Quick-Div: Rethinking Integer Divider Design for FPGA-based Soft-Processors

ERIC MATTHEWS, ALEC LU, ZHENMAN FANG, and LESLEY SHANNON, Simon Fraser University, Canada

In today's FPGA-based soft-processors, one of the slowest instructions is integer division. Compared to the low single-digit latency of other arithmetic operations, the fixed 32-cycle latency of radix-2 division is substantially longer. Given that today's soft-processors typically only implement *radix-2* division—if they support hardware division at all—there is significant potential to improve the performance of integer dividers.

In this work, we present a set of high-performance, data-dependent, variable-latency integer dividers for FPGA-based soft-processors that we call *Quick-Div*. We compare them against various radix-N dividers and provide a thorough analysis in terms of latency and resource usage. In addition, we analyze the frequency scaling for such divider designs when 1) treated as a stand-alone unit and 2) integrated as part of a high-performance soft-processor. Moreover, we provide additional theoretical analysis of different dividers' behaviour and develop a new better-performing *Quick-Div* variant, called *Quick-radix-4*. Experimental results show that our *Quick-radix-4* design can achieve up to 6.8x better performance and 6.1x better performance-per-LUT over the *radix-2* divider for applications such as random number generation. Even in cases where division operations constitute as little as 1% of all executed instructions, *Quick-radix-4* provides a performance uplift of 16% compared to the radix-2 divider.

CCS Concepts: • **Hardware → Reconfigurable logic and FPGAs**; **Arithmetic and datapath circuits**; • **Computer systems organization → Pipeline computing**.

Additional Key Words and Phrases: arithmetic operator, integer divider, variable-latency pipeline, soft-processor

ACM Reference Format:
Eric Matthews, Alec Lu, Zhenman Fang, and Lesley Shannon. 2021. Quick-Div: Rethinking Integer Divider Design for FPGA-based Soft-Processors. *ACM Trans. Reconfig. Technol. Syst.* 1, 1, Article 1 (March 2021), 28 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

While not the most common arithmetic operation, integer division still has significant usage in a wide range of computations. These include cases as simple as non-power-of-two indexing and random number generation [25] to complex use cases including computationally demanding linear algebra and image processing [32]. In today's processors, division hardware is typically split between floating-point/fixed-point and integer dividers. The main difference between the two is that integer dividers produce both a quotient and an integer remainder, whereas the floating point divider will produce a fractional quotient as its only output. Between the two, there has been substantially more research into floating-point dividers both for commodity (hard) processors and FPGA-based soft-processors [8, 9, 12, 17, 28, 29, 32].

Authors' address: Eric Matthews, eric_matthews@sfu.ca; Alec Lu, alec_lu@sfu.ca; Zhenman Fang, zhenman@sfu.ca; Lesley Shannon, lesley_shannon@sfu.ca, Simon Fraser University, School of Engineering Science, Canada.

1936-7406/2021/3-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

For commodity x86 processors, the focus on floating-point optimization has led to the situation where re-purposing floating-point dividers to compute 32-bit integer division can achieve about 2x the performance of directly using integer dividers [16]. This imbalance in performance has led to a series of workarounds, from changing libraries to avoid integer division, to leveraging existing floating-point hardware for integer division in x86 systems [16]. In a fixed architecture system like the commodity x64 processors, these optimizations may offer performance gains with little to no resource penalty. However, in the context of an FPGA-based soft-processor, where the system architecture is highly configurable and needs to meet the resource constraints, calculating integer division with a floating-point divider is inefficient in terms of performance/resource. For instance, a typical 32-bit integer divider such as *radix-2* is merely 1/10 of the size of a 64-bit floating-point divider that typically requires at least a few thousand LUTs [9, 17].

To the best of our knowledge, today's FPGA-based soft-processors, such as the widely used MicroBlaze [34], NIOSII [13] and the LEON3 processor [1], only support one type of integer divider, which is a *radix-2* integer divider with a fixed latency of 32 cycles. There are two intuitive reasons for this situation. First, the *radix-2* divider requires fewer hardware resources than most (if not all) other dividers, making it a suitable choice where resource constraints are the limiting factor. Second, for simple fixed-latency pipeline processors, *radix-2*'s fixed-latency operation is easier to integrate than more complex variable-latency dividers.

On the other hand, as today's FPGAs continue to grow in logic capacity, there is a shift in design paradigms from being primarily resource optimized, to being more performance or performance-per-resource optimized. Moreover, newer soft-processors, such as Taiga [19], have emerged to provide support for variable-latency execution units in the instruction pipeline, and thus enable the exploration of more efficient integer divider designs. These two trends motivate us to rethink the integer divider design for FPGA-based soft-processors.

To understand the impact that division operations can have on the application performance, we illustrate the potential slowdown in terms of Instructions-Per-Cycle (IPC) of a processor in Figure 1 (left sub-figure), as the percentage of division instructions is increased from zero to one-hundred percent. Note that in many soft-processors, the majority of instructions typically have latencies of only a few cycles, and the ideal IPC is one (for single-issue processors). With a latency of 32-cycles and a pipeline Initiation Interval (II) of 32-cycles, a *radix-2* divider significantly affects the throughput of any application, even if the percentage of division instructions is small. Additionally, a zoomed-in view of zero to ten percent on the right-hand side is provided to show more detail in the most representative range for the majority of benchmarks used in this work. As we can see, at just 5% division instructions, a *radix-2* divider can degrade throughput to just 40% of the ideal case. Higher radix dividers that have lower (fixed) latency, such as radix-4, radix-8, and radix-16, still leave a significant performance gap to the ideal case. Instead of using a fixed-latency radix-N divider, when using a fast variable-latency divider, like the ones introduced in this work, the integer division latency can be reduce to around 1.7 cycles on average. And thus, the processor can maintain a much higher IPC. In some microbenchmarks, such as random number generation, such fast variable-latency divider can result in performance uplift of over 6x than that of a *radix-2* divider.

In this work, building upon our work presented in [20], we explore a set of data-dependent, variable-latency integer divider designs for FPGA-based soft-processors, called *Quick-Div*, with a particular focus on both overall performance and performance-per-LUT. The *Quick-Div* dividers are based on the long division algorithm by hand, which will be explained in detail in Section 2.4. We also compare them to a range of radix-based dividers from *radix-2* to *radix-16* along with variants of the radix dividers that allow for early termination, in terms of clock frequency, resource usage and latency. Both theoretical analysis and experimental evaluation of the divider designs

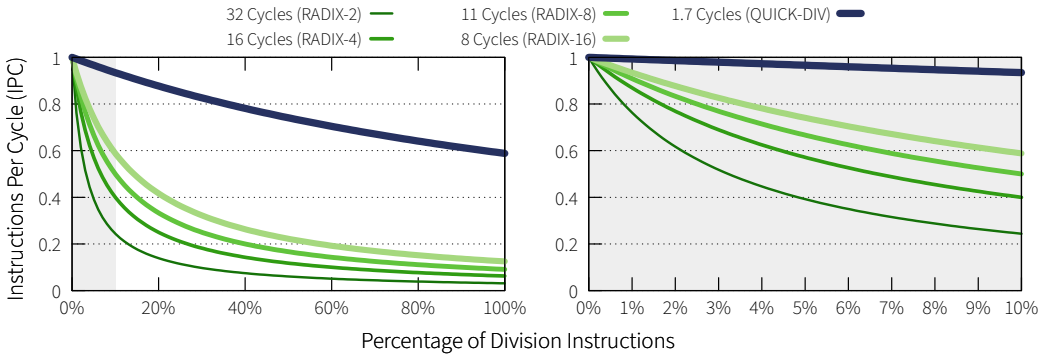

Fig. 1. Potential impact of integer divider latency on an application that otherwise can achieve an ideal Instructions-Per-Cycle (IPC) of one. The left sub-figure shows the impact when the percentage of division instructions increase from 0 to 100%, while the right sub-figure further zooms in the 0-10% range where the majority of the benchmarks fall in.

with a wide range of benchmarks are provided. In our results, we find that the *Quick-Div* dividers can achieve up to 6.8x better performance and 6.1x better performance-per-LUT compared to the commonly used *radix-2* divider. Furthermore, as soon as the percentage of division instructions executed in a program becomes more than 1%, *Quick-Div* dividers becomes more efficient in terms of performance per LUT, while providing more than 16% performance improvement. Finally, we present a case study on integer square root algorithms. With our *Quick-Div* dividers we show that the best performing algorithms can change when comparing to systems with only radix-2 division. With our best performing *Quick-Div* divider, called *Quick-radix-4*, a square root algorithm that uses integer division can outperform other more commonly used techniques by about 1.9x times.

In summary, this paper makes the following contributions.

(1) A comprehensive analysis and comparison of various fixed-latency and variable-latency integer dividers, including their output latency, clock frequency, and resource usage.
(2) A detailed analysis on the behaviour of *Quick-Div*'s data-dependent nature.
(3) A highly-optimized integer divider called *Quick-radix-4*, which is a hybrid of our base *Quick-Div* and *radix-4*, providing the best performance and performance-per-LUT for FPGA-based soft-processors.
(4) A case study demonstrating that our *Quick-Div* dividers performance can changes optimal algorithm selection for soft-processors.

The remainder of the paper is organized as follows. Section 2 discusses background and related work on various types of high performance dividers. Section 3 describes our techniques for reducing integer division latency and design implementation details. Section 4 provides an evaluation across our *Quick-Div* algorithms along with multiple radix-N dividers. Section 5 follows up with a case study on integer square root to highlight the impact of a faster divider on algorithm choice for embedded processor system. And finally, Section 6 concludes the paper and discusses future work.

2 BACKGROUND AND RELATED WORK

First, we present the basics of the common radix-based integer dividers, followed by integration considerations for soft-processors. Next, we discuss related work in optimizing floating/fixed-point dividers and integer dividers, before ending this section with a discussion on variable latency dividers.

```
Counter = 0                                    *BPI – Bit Per Iteration
BPI = FLOOR(log2(N)) //# of bits per iteration *PR – Partial Remainder
PR = 0 //partial remainder                     *bitWidth – Input Width
Quotient = Dividend
do {
  PR = {PR[bitWidth-BPI-1:0], Q[bitWidth-1:bitWidth-BPI]}
  CycleQuotientBits = 0
  //largest divisor multiple less or equal to partial remainder
  for (i = N-1; i >= 0; --i)
    if (PR - i*Divisor >= 0)
      CycleQuotientBits = i
      PR = PR - i*Divisor
      break
  Quotient = {Quotient[bitWidth-BPI-1:0], CycleQuotientBits}
  Counter += BitsPerIteration
} while (Counter < bitWidth)
Remainder = PR
```

Fig. 2. Fixed-latency restoring *radix-N* integer divider algorithm

2.1 Basics of Integer Dividers

As shown in (1), all integer dividers accept two integer operands as inputs, a dividend and a divisor, and produce two integer results, a quotient and a remainder. While keeping both quotient and remainder positive, the divider maximizes the quotient and minimizes the remainder.

$$Dividend = Divisor * Quotient + Remainder \tag{1}$$

Most integer hardware division techniques focus on algorithms that have fixed latency and resolve one or more bits of a fixed length quotient per cycle i.e., radix-based dividers (also called digit recurrence dividers). An additional type, convergent dividers, which start with an initial estimate and double their resolved bits with each iteration are commonly used in high-performance floating point dividers, but require large resource usage on FPGAs [8, 9, 17]. However, as this paper will discuss, there are other data-dependent variable-latency approaches which can offer a higher performance potential for soft-processors in an FPGA environment.

2.1.1 Fixed-Latency Radix-based Divider. A (32-bit) *radix-2* integer divider has a fixed latency of 32 cycles and is the most widely used integer divider in today's soft-processors. It consumes minimal resources and can be easily integrated into most soft-processor's fixed-latency instruction pipeline. Figure 2 provides the pseudo code for a general restoring, radix-N division algorithm. The algorithm determines the quotient from the Most-Significant-Bit (MSB) to the least significant bit, with the Partial Remainder (PR) and quotient stored in shift-registers. Each cycle, log_2N bits (Bits Per Iteration) of the quotient are determined (starting with the MSB), representing the largest divisor multiple that can be subtracted from the corresponding log_2N bits of the dividend. The remainder of that is called the partial remainder and is carried over to the new dividend bits for the next cycle. This process continues until all quotient bits have been determined. Naturally, at the end of the algorithm, the partial remainder becomes the remainder as no further divisor multiples can be subtracted.

Two determining factors for the latency of a radix-N integer divider are: the number of quotient bits that can be determined per cycle (i.e., log_2N bits), and the maximum bit-width of the operands

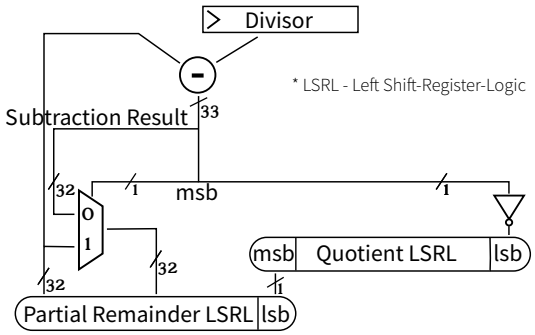

Fig. 3. Fixed-latency *radix-2* integer divider hardware implementation

(i.e., 32 for a 32-bit integer divider). So the latency of radix-N divider can be formulated as such:

$$radix\_N\_latency = ceil(32/log_2 N) \tag{2}$$

For example, a high-radix integer divider like radix-16 would only require 8 cycles to complete a 32-bit integer division (ceil(32/$log_2$16) = 8), whereas a *radix-2* integer divider takes 32 cycles (ceil(32/$log_2$2) = 32). To reduce latency, higher radix dividers such as *radix-16* could potentially be used. However, radix-based dividers' resource usage and complexity grow substantially as the number of bits resolved increases. For example, as shown in Figure 3, the core hardware implementation of a *radix-2* integer divider consists of 1) a subtractor for calculating the difference between the partial remainder and the divisor, 2) a 2:1 multiplexer for the partial remainder to choose a subtraction result between the two divisor multiples, and 3) two shift-registers for storing the partial remainder and quotient. When increasing the radix number, the number of subtractors along with the size of the multiplexer scales directly with the radix number. As we will show in this paper, resource usage becomes quite significant at *radix-16*, and at this point a standard restoring radix-based divider cannot meet the minimum timing requirements of our selected soft-processor.

2.2 Soft-Processors and Integer Division

Existing FPGA-based soft-processors share many attributes, from high levels of configurability to similar resource usages and clock frequencies. Industry designs, such as the MicroBlaze [34] and the NIOSII [13], along with 32-bit RISC-V designs including ORCA [31], VexRISCV [24], and Taiga [19] all support configurations with either a radix-2 divider or no divider at all. Additionally, they all have similar clock frequencies and resource usage (ranging from 1-4k LUTs for higher performance configurations). With a resource usage of 241 LUTs to support radix-2 integer division in the Taiga processor (the FPGA-based soft-processor used in this paper) this small baseline integer divider represents around 6-24% of a typical soft-processor's resource usage. Given the typically low percentage of division operations in applications, its resource overhead is one of the reasons why it is often not included.

In a stand-alone environment, a divider's performance can be evaluated by its latency, initiation interval (often the same as its latency) and its clock frequency. However, in a soft-processor environment, the clock frequency is fixed, and likely lower than what some stand-alone dividers can achieve, (due to other data paths in the processor). Thus, latency becomes the key factor in improving division throughput. As a result, there is more flexibility in a soft-processor environment to explore trading-off clock frequency for latency improvements. Furthermore, due to a focus on resource efficiency for soft-processors, fully-pipelined dividers (which can initiate a new division

operation each cycle) are not suitable due to high resource overheads [12, 29] compared to sequential dividers (with initiation intervals equal to their latency).

Among the open-source processors, Taiga has been found to provide the highest runtime performance [11]. Combining this with single-issue out-of-order execution and variable latency [19] makes Taiga a good choice for exploration of variable latency division for soft-processors. As such, we have selected it as the baseline processor for the purposes of this work. Due to the fact that other FPGA-based soft-processors, such as the MicroBlaze and VexRISCV, have similar operating frequencies, our best performing divider, *Quick-radix-4* (which exceeds Taiga's clock frequency by 41%), could also be used with these processors.

2.3 Related Work in Dividers for FPGAs

Over the years, many studies have looked into improving performance, resource utilization, and other aspects of floating-point/fixed-point dividers for FPGAs [8, 9, 12, 17, 28, 29, 32]. This is due to the crucial role these dividers play in many fields and applications that require complex numerical computation.

2.3.1 Radix-N Dividers. As most research on dividers focuses on performance in a stand-alone environment, there has been a lot of interest in improving the clock frequency of radix-based dividers. A common approach is through SRT-based radix-N division. An SRT divider still produces bits through digit-recurrence like the restoring radix-N divider presented earlier, however, it attempts to reduce the delay in the logic to determine the quotient bits on each cycle. It does this with a redundant digit set, allowing for the quotient bits to be estimated with corrections made on future cycles. As divider use cases can be quite varied, research on dividers for FPGAs also include work on purely combinational and fully-pipelined designs [12, 28, 29] in addition to the sequential designs most relevant to this work.

Hemmert et al. explored radix-based floating-point divider design (up to radix-4) across a wide range of design constraints, from fully-pipelined to sequential, for digit recurrence algorithms such as non-restoring and SRT [12]. In terms of sequential designs, the highest radix-N design explored was radix-4 and was found to have the best throughput. A comparative study of SRT dividers on FPGAs by Sutter et al. [28] (comparing from radix-2 to radix-16 based designs), found that a radix-4 SRT design provided the best latency for a sequential design. Later work by Sutter et al on FPGA-optimized non-restoring radix-2 to radix-32 dividers [29] found a radix-16 non-restoring divider to offer the highest throughput for a sequential divider design. While these works provide optimized radix-N dividers for FPGAs (in terms of stand-alone throughput), the variable-latency designs in this work exceed the latency characteristics of radix-16 dividers (with resource usage only slightly higher than radix-4). Furthermore, frequency optimizations are not necessary for radix-N dividers below radix-16 in order to meet timing considerations for the Taiga soft-processor. Finally, several of these techniques rely on the inputs to be normalized (as they typically are for floating-point operations) in order to simplify their quotient selection functions. This would result in additional resource overhead and latency for integer division as inputs are not normalized.

2.3.2 Convergent Dividers. Another common divider design for more complex floating-point dividers are convergent algorithms such as Newton-Raphson and Goldschmidt [10]. In these algorithms, starting with an initial approximation, the number of correct bits of the result is doubled on each cycle. On FPGAs, Liebig et al. achieved latencies as low as 8-cycles for double-precision floating-point [17] (same latency as a radix-16 divider for 32-bit integers). However, for use as an integer divider, a similar precision would be required [16], along with additional logic to normalize and denormalize the inputs/outputs. Additionally, these algorithms only produce a quotient, requiring additional multiplication and subtraction logic to produce a remainder. Given

that this design's resource usage of 1525 LUTs is similar to the size of Taiga, without any integer divider (1693 LUTs), suggests these techniques are not suitable for performance-per-resource efficient integer division.

2.3.3 Lookup Table Based Dividers. Taylor series expansion using lookup tables and multipliers [7] is another technique that has been explored on FPGAs. Fang et al. presented a floating-point divider based on Taylor series expansion (resulting in a fixed latency of 14 cycles) with better throughput than vendor cores. However, with a Xilinx resource usage of over 20k LUTs this technique would also not be suitable for resource efficient integer division.

Outside floating-point divider research, Dinechin et al. looked into ways to efficiently compute division by small constants [5]. However, for a soft-processor, full, non-constant, processor data width (32-bit by 32-bit) divisions and remainder operations are required, preventing these techniques from being applicable.

2.3.4 Variable-Latency Dividers. Variable-latency dividers exist primarily in two forms. One type relies on self-timed (asynchronous) operation, and the other on producing a variable number of quotient bits per iteration (cycle). In terms of asynchronous designs, there have been designs that consider fast detection of quotient bits that are zero (to skip adder delays) [22] as well as those that use quotient speculation [2]. However, asynchronous designs are not well supported and would add significant additional overhead and complexity in a synchronous soft-processor design.

In addition to speculation in an asynchronous environment, quotient speculation was explored by Cortadella et al [3] with roll-back in case of a misprediction. Their highest performing design, a radix-512 divider, was able to produce approximately six quotient bits per cycle, on average. This would result in an average latency of 5.33 cycles for 32-bit division whereas the *Quick-radix-4* divider presented in this work averages 2.7 cycles. Furthermore, this design was not resource efficient relative to their non-speculative radix-16 design.

The earliest work on skipping quotient bits that are zero, to the authors' knowledge, was presented by Ligomenides in 1977 [18]. The technique presented is based-off of long division by hand where the largest multiple of the divisor that is less than the partial remainder is attempted to be found on each iteration. (A more detailed explanation of this general approach, and the specific variant explored by this paper is presented in the following section, Section 2.4). In Ligomenides' variant, if the guess is incorrect, an additional iteration is required to correct the overestimate compared to the one presented in this work. In our analysis presented in Section 2.4, we found there to be an average of 0.7 corrections, per division, for uniformly distributed random quotient and divider pairs. As such, this approach would add, on average, 0.7 cycles (increasing average latency by 26%). Additionally, in the worst case, this could occur on every trial subtraction, which would double the latency of some number pairs compared to the approach presented in this work.

On FPGAs, quotient zero-skipping dividers have been investigated in two master's theses by Khan [15] and Trummer [30]. In [15], a per-iteration aligning divider is presented. However, it is implemented as a 32-bit dividend, 17-bit divisor for random number generation, and no latency analysis is provided for different data sets. In [30], a per-iteration aligning divider is also presented, which is investigated for stand-alone usage only. However, it does not present resource usage or operating clock frequency. Only weighted average cycles of the dividers are compared through algorithmic simulation. None of the published work has focused on optimizing the variable-latency data-dependent divider designs for FPGAs, provided detailed characterization, or explored their integration into FPGA-based soft-processors.

```
Remainder = Dividend, Quotient = 0
while (Remainder > Divisor) {
  LOIΔ = ⌊log2(Remainder)⌋ - ⌊log2(Divisor)⌋
    EstimatedDivsor = 2^LOIΔ * Divisor
    A = Remainder - EstimatedDivsorMultiple
    B = Remainder - EstimatedDivsorMultiple/2
    Quotient[(A < 0) ? LOIΔ-1 : LOIΔ] = 1
    Remainder = (A < 0) ? B : A
}
```

Fig. 4. Variable-latency integer divider algorithm

2.4 Quotient Zero-Bit Skipping

Quotient zero-bit skipping is a variable-latency data-dependent divider where the algorithm mirrors a standard process for long division by hand. For each iteration, the largest multiple of the divisor that is less than the partial remainder is found. This approach leads to a result that produces one non-zero digit of the quotient per cycle, thus only requiring the same number of cycles as the number of non-zero bits in the resulting quotient, i.e.,

$$Quick\_div\_latency = num\_non\_zero\_bits\_in\_quotient \tag{3}$$

Figure 4 provides pseudo code for the division algorithm. In each cycle, the partial remainder is compared against the divisor as an exit condition, which we call *early-termination*. This comparison provides an early exit if the divisor is greater than the dividend; a common practical case of this is using the modulus operator for non-power-of-two indexing. Next, to find the highest multiple of the divisor that is less than the partial remainder, we first find the Leading-One-Index (LOI) for both the divisor and the partial remainder, where the LOI is equivalent to the floor of *log2* of the number. The difference between the two LOIs is the number of bits that we need to left shift the divisor by to align it with the partial remainder. However, as this difference only compares the LOIs for both numbers, it can overestimate the difference resulting in a subsequent subtraction that overflows. In works such as Ligomenides [18] and Khan [15] the result is allowed to overflow and requires additional cycles to correct when this occurs. However, as the estimate here can only overestimate by a single power-of-two, in the case where the estimated-divisor-multiple overflows, we left shift the divisor by one bit less and perform a second subtraction. Thus, requiring no additional cycle overhead.

2.5 Summary

In summary, much of the work on division for sequential dividers has focused on stand-alone performance with radix-16 typically providing the highest throughput for digit recurrence based algorithms. On FPGAs, lookup table based and convergent dividers can provide better latencies than radix-16 dividers for double-precision floating-point division, but do so with high resource costs that are unacceptable for small soft-processor designs.

With the introduction of open-source soft-processors (e.g., Taiga [21]) that support variable-latency execution units, there exists a new opportunity to investigate variable-latency integer dividers in a soft-processor environment. In contrast to Trummer's and Khan's studies [15, 30],

we provide a comprehensive analysis of various integer dividers, optimize the variable-latency data-dependent integer divider, and integrate it within a soft-processor, with the goal to achieve the best performance and performance-per-LUT for FPGA-based soft-processors.

3 THE QUICK-DIV DIVIDER

In this section, we characterize the variable-latency data-dependent integer divider (called *Quick-Div* in this paper), discuss its implementation and optimization on FPGAs, and integrate it with state-of-the-art FPGA-based RISC-V soft-processor Taiga [21].

Compared to the widely-used *radix-2* integer divider that has a fixed latency of 32 cycles, a key property of the variable-latency *Quick-Div* integer division algorithm (presented in Section 2.4) is that: the closer the dividend and divisor are in magnitude, the fewer cycles the algorithm will take to complete. On *each* iteration the upper bound for the remaining number of cycles is:

$$Quick\_div\_latency <= log_2 \frac{PartialRemainder}{Divisor} \tag{4}$$

In total, the number of cycles is equal to the number of set bits in the quotient (also commonly referred to as the population count). As the algorithm is data dependent, the average latency will depend on the input data set. However, for any pair of independently generated uniformly generated random numbers we can calculate the average latency. With the initial shift we have the index of the leading one (LOI) of the quotient, which provides us with the bit-width of the quotient. As, on average, half the quotient bits will be set, dividing the LOI of the quotient by two provides us with the average latency:

$$Quick\_div\_average\_latency = log_2 \frac{Dividend}{Divisor}/2 + 2 \tag{5}$$

Two additional cycles are required due to the hardware implementation. The first is required due to an initialization cycle before the algorithm starts, and the second for determining when the divisor is greater than the partial remainder. While the divider's performance is the best when the two inputs are close in magnitude, there are still cases where the divider's latency is small regardless of the magnitude separation between the numbers. In the case where both the dividend and divisor are power-of-two, *Quick-Div* requires only a single iteration to produce the quotient.

To characterize the behaviour of the *Quick-Div* divider, we examine how the average latency of the divider scales as the bit-width of the divider is increased. For the smaller bit-widths, all pairs were tested exhaustively; and for larger bit-widths, 100 billion number pairs were randomly generated from a uniform distribution. This test was repeated ten times with different random seeds with negligible variation: less than 1% variation across the runs in terms of the latency distribution. As shown in Figure 5(a), as the bit-width of the *Quick-Div* divider increases, the average iteration requirements increases up to approximately 1.69 iterations and then flattens out, which is much more attractive than the N-cycle latency for a *radix-2* N-bit integer divider. Additionally, we investigated how often the estimated-divisor-multiple is overestimated, finding this averages to 0.7 occurrences per division. Note that Figure 5 characterizes the average division latency of *Quick-Div* on the algorithmic level and does not include the latency overhead introduced by the hardware implementation as it may vary.

To understand how the low average latency is achieved, we further break down the access latency distribution for the 32-bit *Quick-Div* divider. As shown in Figure 5(b), the vast percentage of divisions take only a few cycles (cycles shown in x-axis) to complete. In fact, all iteration counts beyond five cycles represent less than 1% of all pairings. This is due to the fact that the percentage of pairings that have these higher latencies reduces significantly as the latency increases. By 32 cycles, there is only one number pair, $(2^{32} - 1)/1$, that will actually take this many iterations to

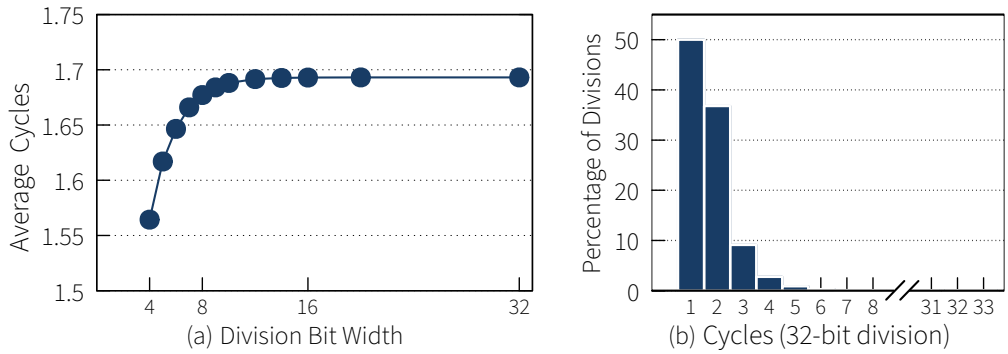

Fig. 5. (a) Average cycles of *Quick-Div* for division between two uniformly distributed random numbers, with division bit-widths between 4 and 32. (b) Average cycle distribution of *Quick-Div* for 32-bit division between two uniformly distributed random numbers.

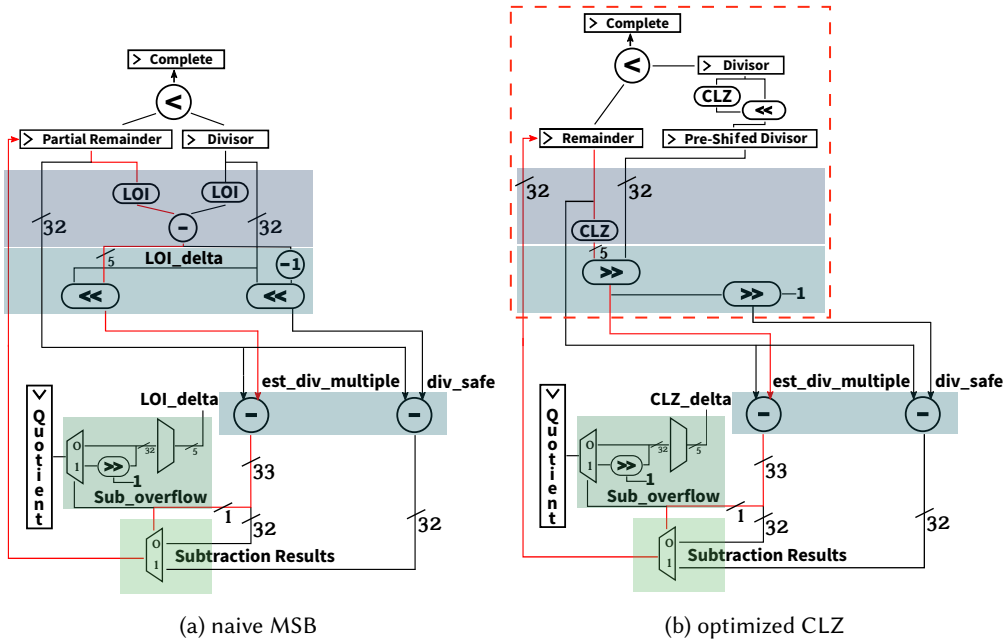

Fig. 6. Variable-latency *Quick-Div* divider hardware implementation. The dotted red box indicates the design changes. Shaded boxes correspond to the pseudo-code in Figure 4.

complete. A further analysis of *Quick-Div's* latency characteristics, particularly for higher latency number pairings is covered later in Section 3.3.3.

3.1 Quick-Div Divider Design

The initial *Quick-Div* implementation closely follows the pseudo-code that was presented in Figure 4. A schematic of the circuit, along with changes to improve clock frequency and details on the design's critical path is presented in Figure 6. The floor of log2 operations are computed by finding the Leading One Index (LOI) for the divisor and partial remainder. In the initial design iteration, Figure 6

(a), these are implemented as 32-bit priority encoders. The difference between the LOIs is used to shift the divisor (the power-of-two multiply) and to generate the new bit to set for the quotient. Two subtractions are performed in parallel, with the overflow bit of the estimated-divisor-multiple as the mux's select bit to determine the final partial remainder and quotient bit. The termination condition is provided by a separate comparator that checks for when the divisor is greater than the current partial remainder.

The critical path of the design (for both designs), highlighted with solid red lines in Figure 6 (a), was found to be from the partial remainder, through the LOI delta computation, shift, subtract and final mux. Splitting this set of operations would result in reducing the throughput of the divider, as such, we focused on investigating other ways of improving the clock frequency.

3.2 Clock Frequency Optimization

The development of this divider was driven by a desire to improve the performance of division for FPGA-based soft-processors. As such, an important consideration is that the divider should not reduce the operating frequency of the processor in order to be suitable as a complete replacement of the standard *radix-2* divider. The initial *Quick-Div* implementation was found to be in the critical path of small processor configurations. To alleviate this performance bottleneck, we investigate several optimizations to the design.

Trial #1: Hierarchical LOI design. The first optimization investigated was to replace the 32-bit LOI priority encoder with a tree structured LOI design. However, this approach provided little benefit as it did not reduce the delay on the least significant bits which are the bits needed first for the delta LOI subtraction.

Trial #2: Avoiding LOI delta and pre-shifting the divisor. Instead of using the LOI delta to perform the divisor shifting, we split the shifting operation into two parts. As an initialization step we right shift the divisor by the divisor's LOI, then the divisor is left shifted by the partial remainder's LOI each cycle. However, this leads to a potential 64-bit result for the right shifted divisor (to avoid losing divisor bits), and a 64-bit to 32-bit left shift when left shifting by the partial remainder's LOI. This negates some of the benefit of pre-shift the divisor.

Trial #3: Using Count Leading Zero (CLZ). The next optimization was to replace the LOI logic with Count Leading Zero (CLZ) logic. CLZ can be calculated with the same kind of circuit (it is equivalent to $32 - LOI$). With CLZs, the directions of the shifts can be reversed: we first left shift the divisor by the divisor's CLZ, and then right shift it by the partial remainder's CLZ. Using CLZs, we know that the divisor will not be shifted outside of a 32-bit bound, and thus reduces the storage to 32-bits and the shifts to 32-bits. The logic for this version of the circuit is shown in Figure 6 (b) (differences between the CLZ pre-shift approach and the original naive design are highlighted with a dotted red box).

Final choice: optimized CLZ, i.e., quick-clz. Finally, we created an optimized version of the CLZ circuit, shown in Figure 7, which computes the upper two bits of the result in one level of logic less than the remaining three bits. As the upper two bits take one less level-of-LUTs, they can drive the select logic for the first stage of the shift operation while the lower bits are being determined thus reducing the overall levels of logic for the critical path. The highest bit, bit index 4, indicates that there are at least 16 leading zeros. This can be implemented by comparing the upper 16 bits against zero. Bit index 3 is one if there are at least 24 leading zeroes (when bit 4 is 1) or if there are between 8 and 15 leading zeros (when bit 4 is 0). Similarly, we can calculate the value for the rest of the bits. For bit index 2, the comparisons are made on groups of four bits. Finally, for bit indices 1 and 0, this would result in comparisons needing 30 and 31 bits. As such, we look at computing the

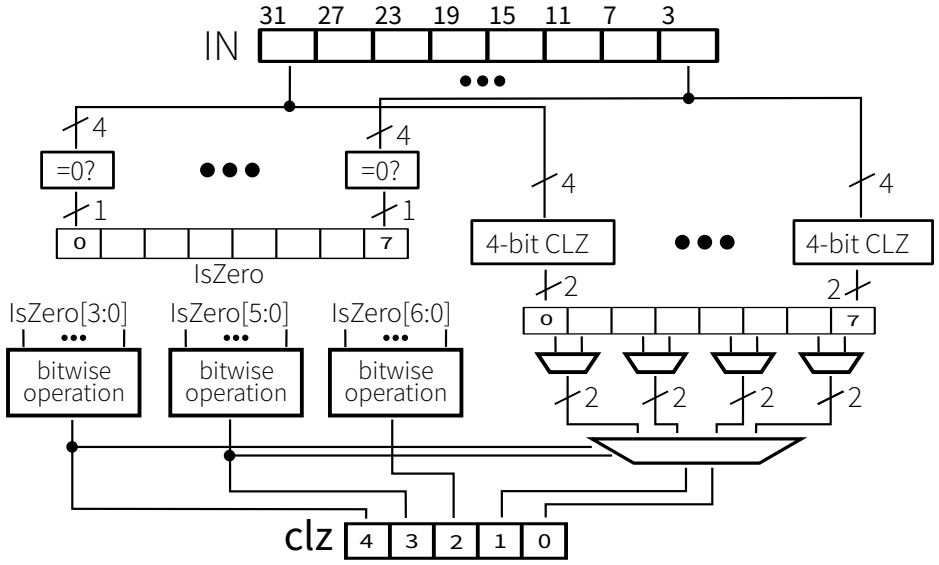

Fig. 7. Optimized CLZ (count leading zero) unit hardware implementation: 19 LUTs required, two levels-of-LUTS for bits [4:3] and three levels required for bits [2:0] allowing the first stage of the following shift logic to resolve while the lower bits are computed.

CLZ for 4-bit intervals, mux the results pair-wise, using whether the upper pair is all zeros, before using the upper two bits of the CLZ result to mux the final result for bits 1 and 0. This approach reduces the required LUTs for the CLZ circuit, from 30 to 19, and reduces the levels-of-LUTs from three to two for the upper two bits of the CLZ output.

With this optimized *quick-clz* design, clock frequency, for the divider as a whole (Figure 6 (b)), is improved by 19% (67MHz) compared to the baseline design (*quick-naive* Figure 6 (a)). Overall, due to the additional pre-shifting logic, only a small increase in resource usage is required (further details and evaluation is provided later in Section 4.3). This places the *Quick-Div's* clock frequency safely above the Taiga soft-processor's clock frequency, such that it does not show up in the critical path of the processor.

3.3 Further Design Exploration

While the *Quick-Div* algorithm has excellent latency characteristics for the average case with randomly generated numbers, there are still some divisor/dividend pairings that have long latency, particularly if a large dividend is divided by a small divisor such as 1. In this section, we explore two different approaches at improving the worst case of the *Quick-Div* algorithm.

3.3.1 Worst case division optimization: Quick-clz-2bit. As the *Quick-Div* divider's worst case is dividing numbers with a large number of set bits by a number with only a few or one bit set. One approach to improve such cases would be to try and resolve two consecutive bits per cycle. As *Quick-Div* already skips over groups of zeros in the quotient, we only need to focus on the case where there are two consecutive one bits in the quotient. This is the equivalent of determining if both the estimated-divisor-multiple and the "safe" divisor multiple can be subtracted from the partial remainder. To implement this functionality, a third subtractor is needed for this calculation and additional muxing for selecting the new partial remainder as shown in Figure 8. Quotient

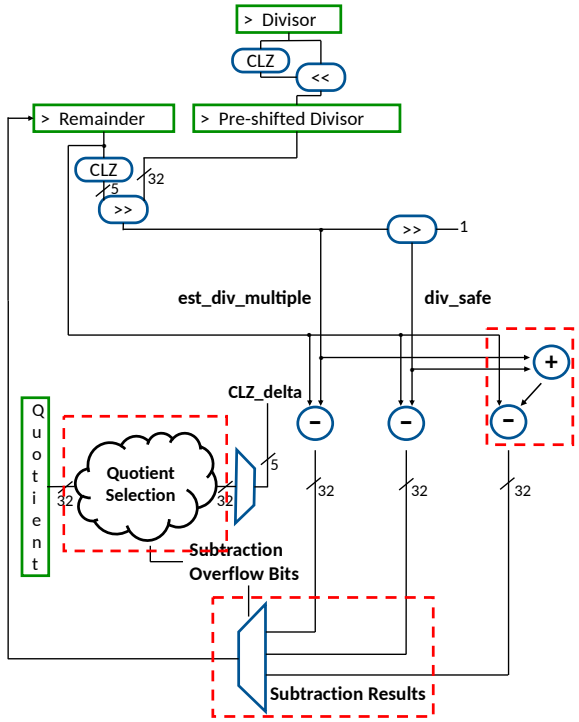

Fig. 8. *Quick-clz-2bit* divider hardware implementation: the bounding boxes indicate the design changes

digit selection, remains the same as in earlier designs, however, it is now possible to set two bits as opposed to one. With this approach, division of numbers with groups of consecutive ones in their quotient will be able to be calculated faster. For example, $(2^{32} - 1)/1$ would now require 16 iterations as opposed to the original 32. However, in cases where there are no consecutive one bits in the quotient, this enhancement will not change the latency of *Quick-Div*. Additionally, in cases where the alignment is an overestimation, it can still only resolve a single bit of the quotient per cycle. Later in this section we show that while this approach can slightly reduce the average latency overall, but only significantly so for division of large ($> 2^{29}$) numbers by small numbers, such as one. In Section 4, we will show that, in practice, the benefits do not outweigh the additional resource costs and impact on clock frequency from the additional logic.

3.3.2 Variability optimization: Quick-radix-4. A downside of the *Quick-Div* approach is that while the average latency is low, the variability increases as the separation between the dividend and the divisor grows, providing performance similar to radix-2 in the worst case. Furthermore, with each iteration requiring a realignment of the divisor to the partial remainder, we found there were only a few improvements that we could make to the clock frequency. A possible alternative would be to only perform an initial alignment before switching to a radix-N divider. This has the benefit of allowing us to remove the alignment from the iterative step of the algorithm and allow for higher radix implementations to be explored. However, its performance will depend on how much of the benefit of aligning the divisor and dividend/remainder comes from the initial alignment vs continual re-alignment. With only an initial alignment, the divider latency will now only depend on the initial separation between the divisor and dividend. However, this approach loses the benefit

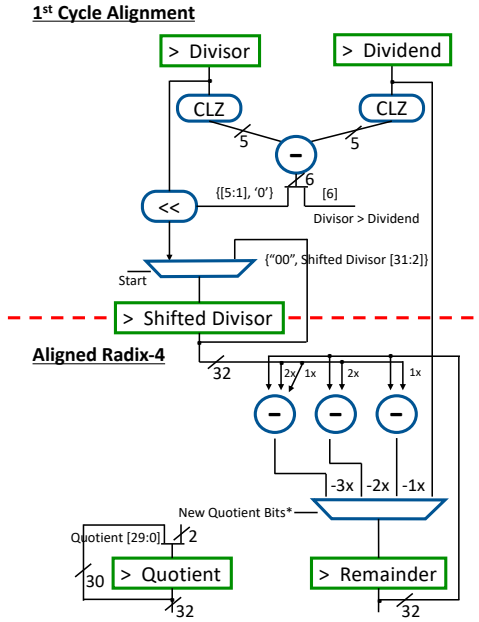

Fig. 9. *Quick-radix-4* divider hardware implementation

of being able to skip over any other group of zeros in the quotient. Thus, to match or exceed the performance of our *quick-clz* design, this approach will require the iterative portion to use a higher radix than radix-2.

To determine the latency of this divider we start by determining the Leading One Index of the quotient, which is simply $log_2 \frac{Dividend}{Divisor}$ and can be determined with the same CLZ delta logic as was used in the *Quick-clz* design. This provides us with the number of bits of the quotient that we need to calculate using radix-N. Therefore, for the radix-N portion, we only need $\lceil log_2 \frac{Dividend}{Divisor} / log_2 N \rceil$ cycles to determine the final result. Note that after the initial alignment, as we know the remaining number of cycles, we do not need to check if the divisor is greater than the partial remainder (as was the case for *Quick-clz*). In summary, the latency for the *Quick-radix-N* divider is:

$$Quick\_radix\text{-}N\_latency = \lceil log_2 \frac{Dividend}{Divisor} / log_2 N \rceil + 1 \tag{6}$$

With radix-2, this results in a latency (N=2 in Equation 6) equal to the upper bound of *Quick-Div* (Equation 4), which would result in worse overall latency. However, if we look at the average latency of *Quick-Div* (Equation 5), it is roughly half of the upper bound (with radix-2). As such, if we combine an initial alignment operation with radix-4, the *Quick-radix-4* divider can achieve the following latency, which is one cycle less than the average latency of *Quick-Div* (Equation 5):

$$Quick\_radix\text{-}4\_latency = \lceil log_2 \frac{Dividend}{Divisor} / 2 \rceil + 1 \tag{7}$$

For this design, worst case latency has now been reduced to 17 cycles. However, there are still cases where the continual re-alignment approach will provide lower latency. While combining an initial alignment with radix-4 gives similar latency to the continual re-alignment approach, this approach could also be combined with radix-8 or radix-16. However, as will be shown in Section 4.3, high resource usage for these two dividers would lower the resource efficiency of the

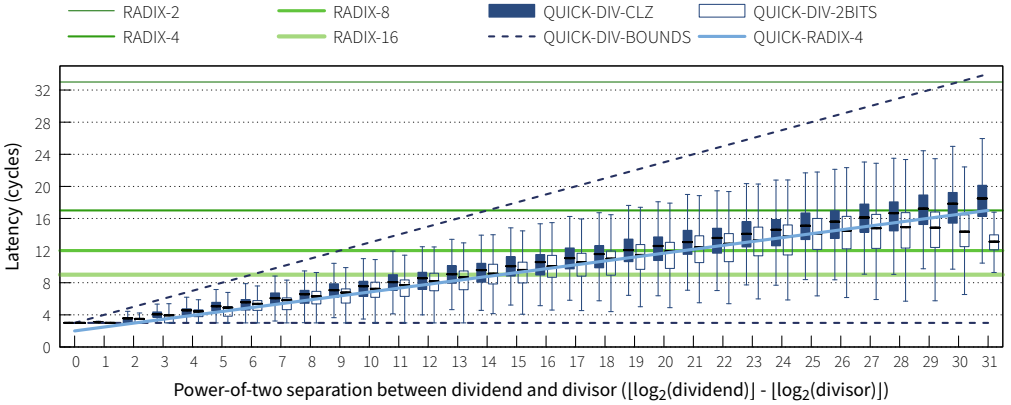

Fig. 10. Boxplot latency breakdown of various dividers while increasing the power-of-two separation between the dividend and divisor.

design. Additionally, radix-16 does not meet the minimum timing requirements for the processor system.

Figure 9 provides an overview of the implementation of this design variant. In the first cycle, the CLZ delta between the divisor and dividend is determined and the divisor is shifted to be aligned with the dividend. In the case where the shift would be odd, we round down to the next even number (through truncation) as, with *radix-4*, we want the shifted divisor to be aligned to a power of two. At this point, the number of cycles remaining is equal to half of the shift amount. In the case where the CLZ delta is negative, we terminate after the first cycle. On each remaining cycle, three trial subtractions are performed to determine the new quotient bits. Unlike with quick-clz, as we know the remaining number of iterations we do not need any extra circuitry to determine when the division is complete. This helps reduce both resource usage and latency, as compared to quick-clz, we do not require a separate comparison to determine if the division is complete. This design has a critical path advantage over the continual re-alignment approach. As it needs only a single alignment, that alignment can be performed in the first cycle, thus separating it from the subtraction path and providing a higher operating frequency.

3.3.3 Latency breakdown. As shown at the beginning of this section, *Quick-Div* has a very low average latency when the inputs are random numbers from a uniform distribution. Not only is the average latency low, but the vast majority of parings take only a few cycles to complete. To provide more insight, we present a more detailed breakdown of latency for *Quick-Div* along with the *Quick-Div-2Bits* and *Quick-radix-4* variants and the standard radix-N dividers in Figure 10. Figure 10 presents the average latency and spread of the dividers as the power-of-two separation ($\lfloor log_2 dividend \rfloor - \lfloor log_2 divisor \rfloor$) is increased from 0 to its maximum of 31. For the *Quick-Div* dividers, where this data is achieved through simulation, 100 million pairs of numbers are used for each power-of-two separation. For the radix-based dividers, this is constant, and thus, they are constant across this figure. (Note, their latencies are all one more than the iteration count of their algorithms due to the cycle needed to load their inputs). The dotted lines represent the minimum and maximum bounds for the *Quick-Div* algorithm with the distribution being represented with a box plot.

From Figure 10, we can see the different trade-off points between the different dividers. For *Quick-Div*, for powers-of-two separation up to 14, the vast majority of number pairings will complete with lower latency than a *radix-16* divider. While the upper bound of the *Quick-Div* algorithm can reach 32-cycles, it will only do so in the case of $(2^{32} - 1)/1$, and even for numbers with a power-of-two separation of 31 (i.e., the dividend has its most significant bit set and the divisor is one), the average latency is 16 cycles (comparable to *radix-4*). In terms of the two variations of *Quick-Div*, we can see that as the power-of-two separation increases, the benefit of *Quick-Div-2Bits* grows, however it only becomes significant for extreme magnitude differences. As will be seen in Section 4, this translates into benefits only being seen in cases such as division by one. For the *Quick-radix-4* variant, we can see that it is effectively equal to the average of *Quick-Div* reducing the variability in latency. With randomly generated numbers we expect *Quick-Div* and *Quick-radix-4* to have the same performance, in later sections we will show if this holds in applications with real data sets.

3.4 Integration into Taiga Soft-Processor

Finally, we also integrate our variable-latency *Quick-Div* dividers with the 32-bit RISC-V soft-processor Taiga [19, 21], which is open-source and supports a seamless integration of variable-latency execution units. As our *Quick-Div* dividers are unsigned, they require sign conversion before and after completion depending on the instruction operands and type. However, we did not need to make any modifications to Taiga for this as its division unit was already structured to perform sign conversions around the existing 32-bit *radix-2* divider. As we wish to evaluate whether the *Quick-Div* dividers can be a complete replacement for the standard *radix-2* divider, we have configured the processor with a minimal single-issue high-performance configuration to show that there are performance and performance-per-LUT benefits even for smaller processor configurations. This configuration includes 16KB of shared instruction/data local memory, a 512 entry branch predictor, an AXI bus interface for a UART and no exception/interrupt support or caches. For more details of the Taiga processor, we refer the reader to the Taiga papers [19, 21].

4 EXPERIMENTAL RESULTS

4.1 Experimental Setup

4.1.1 Dividers for Comparison. As a point of comparison for our variable-latency *Quick-Div* dividers, we implemented a range of restoring radix-based fixed-latency dividers, all of which operate on unsigned integer numbers. The radix-based dividers include the *radix-2* divider from Taiga [21] and radix-4, 8 and 16 dividers that we implemented. Additionally, as one of the performance benefits of the *Quick-Div* design is the early termination when the divisor is larger than the dividend, we also applied these modifications for the radix-N dividers, and labelled them with the suffix Early-Termination (ET). For *radix-2* we also investigated a design that can terminate whenever all bits of the quotient have been determined (i.e. when the divisor is greater than the remainder), which is labelled Early-Termination-Full (ETF). For our *Quick-Div* dividers, we evaluate four versions: 1) *Quick-Naive*, which is the initial design presented in Section 3.1 without any optimization, 2) *Quick-CLZ*, which is the version presented in Section 3.2 with clock frequency optimizations using count leading zeros (CLZ), 3) *Quick-CLZ-2BIT*, which can calculate two consecutive quotient bits in some situations and, 4) *Quick-radix-4*, which performs an initial alignment, then rather than re-aligning on each iteration, performs radix-4 division.

4.1.2 Hardware and Software Setup. All resource usage and frequency numbers have been collected for the Xilinx Virtex UltraScale+ VCU118 board (XCVU9P-L2FLGA2104E) using Vivado 2020.1 synthesis, place and route with all default settings. We leave the detailed description of benchmarks into Section 4.4. While the Taiga processor integrated with these dividers is synthesized to run

Table 1. Min, max, and average cycles for radix-based dividers and *Quick-Div* dividers for random uniformly distributed numbers.

Divider Cycles	**Min**	**Max**	**Avg**
Radix-2	33	33	33
Radix-2-ET	1	33	17
Radix-2-ETF	1	33	17
Radix-4	17	17	17
Radix-4-ET	1	17	9
Radix-8	12	12	12
Radix-16	9	9	9
Quick-Naive	3	34	3.69
Quick-CLZ	3	34	3.69
Quick-CLZ-2BIT	3	34	3.66
Quick-radix-4	1	17	2.56

on-board, the detailed number of instructions and cycles each benchmark takes is collected through the open-source Verilator simulator [27] that simulates the full processor system.

4.2 Divider Latency Comparison

Table 1 presents min, max and average cycles for each divider derived by analytical means for the radix-N dividers and by running 100 billion pairs of two uniformly distributed random numbers for the variable-latency dividers. The *early-terminate* dividers have a lower min and average latency as half of all random number pairs will result in triggering the early-termination paths in those dividers. For the radix-N dividers with early-termination, the termination logic is performed before they start during the sign conversion stage, thus reducing their contribution to the division instruction latency to zero. In terms of instruction execution time, the division latency has an additional 3 cycles over the values in the table due to potential sign conversion operations inside the processor's division execution unit. For the *Quick-Div* dividers, *Quick-Naive* and *Quick-radix-4*, are both able to terminate in the first cycle when the divisor is greater than the dividend, while for resource and clock frequency reasons *Quick-CLZ* and *Quick-CLZ-2Bit* require two cycles. It would be possible to add additional logic to *Quick-CLZ* and *Quick-CLZ-2Bit* to enable them to terminate in the first cycle, however the resource cost (at least 30 LUTs) would be larger than the benefit (in most benchmarks: no change, in some: at best one or two percent).

4.3 Resource Usage and Frequency Comparison

4.3.1 Standalone Quick-Div Divider Comparison. Figure 11 presents the resource usage and frequency for each divider implementation when placed and routed as a standalone component. All I/Os are registered so as not to be impacted by pin assignment. As there is a large span in achievable clock frequencies for the dividers, clock frequency targets for each divider are set separately to be just above their achievable frequency. The red line in the figure—the max frequency that the integrated Taiga-radix-2 configuration can achieve—is the target frequency that all standalone dividers need to reach in order to not immediately become the critical path of the processor.

As shown in Figure 11, the initial implementation of the *Quick-Div* algorithm (*Quick-Naive*) did not meet the required timing threshold. However, the optimized *Quick-CLZ* version improved the clock frequency by 67MHz (19%) over the *Quick-Naive* version, putting its clock frequency at 416MHz, well above the 367MHz threshold. Of the two *Quick-Div* variants, the *Quick-CLZ-2BIT* version has a lower clock frequency than *Quick-CLZ* due to the additional muxing in the critical path. Conversely, the *Quick-radix-4* variant is able to achieve a higher clock frequency of 519MHz

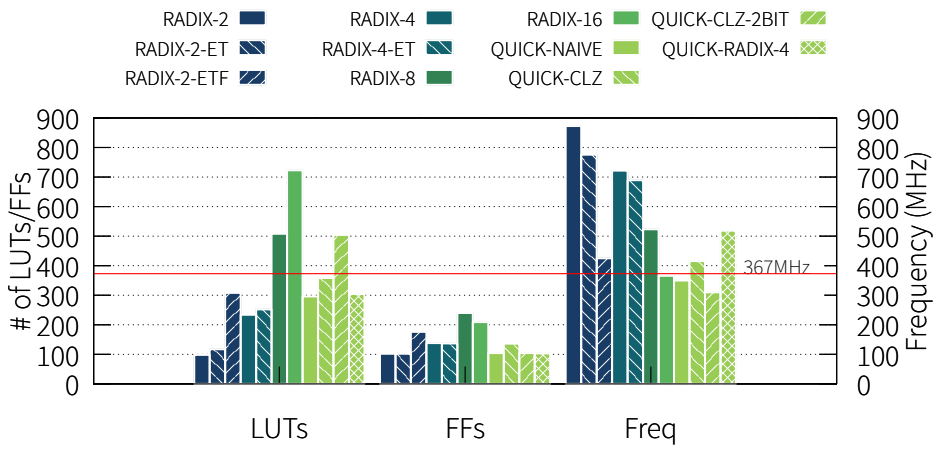

Fig. 11. Operating frequency and resource usage for standalone dividers: the red line is the target frequency.

due to the splitting of the divisor shifting and *radix-4* subtraction into separate cycles (as explained in Section 3.3).

All radix-based dividers meet the target clock frequency except for *radix-16* due to the large number of comparison logic needed to generate their quotient bits. Additionally, the *radix-16* divider requires significantly more LUTs than the other dividers, more than twice of that of the *Quick-CLZ*. We still include the *radix-16* divider for performance evaluation, just to demonstrate the effectiveness of our *Quick-Div*.

In terms of resource usage, for regular radix-N dividers, the LUT usage (the dominating resource usage compared to Flip-Flop) increases almost linearly with $log_2 N$. The early-termination-full (*Radix-2-ETF*) optimization significantly increases the LUT usage due to requiring shifters for early extraction of the quotient and remainder. In comparison to the radix-N dividers, our *Quick-Div* implementations resource usage are approximately half way between the *radix-4* and *radix-8* dividers (with the exception of *Quick-CLZ-2BIT*). But as shown in Table 1, with an average latency of less than 3-cycles a *radix-1024*, would be needed to achieve the same average latency.

4.3.2 Taiga-Quick-Div Integration Comparison. Figure 12 presents the frequency and resource usage for the dividers when integrated into the Taiga processor. When placing and routing the system, we found a large variance (of around 20%) in the clock frequency when making small inconsequential changes to the logic. In particular, we found that the frequency variance was the largest for the smallest configuration (i.e., the no-div case). To help reduce the variation, we built each design with multiple clock frequency targets, with targets spanning 375 to 400MHz (with increments of 5MHz), selecting the highest frequency for each divider system. While the resulting frequency does not vary substantially across many of the designs, several of the dividers are the critical path in their implementations. These include, *radix-2-early-terminate-full*, *radix-16*, *Quick-Naive*, and *Quick-CLZ-2Bit*. The other dividers, including *Quick-CLZ* and *Quick-radix-4*, were not the constraining factor in frequency obtained. Instead, the critical path was found to be either between the load-store unit and the 64KB shared instruction/data BRAM or between the branch predictor logic and the shared instruction/data BRAM.

In terms of the overall resource overhead, the simplest *radix-2* divider integration uses 241 more LUTs, or 14% more LUTs, compared to the Taiga processor without any divider. Between *radix-2*

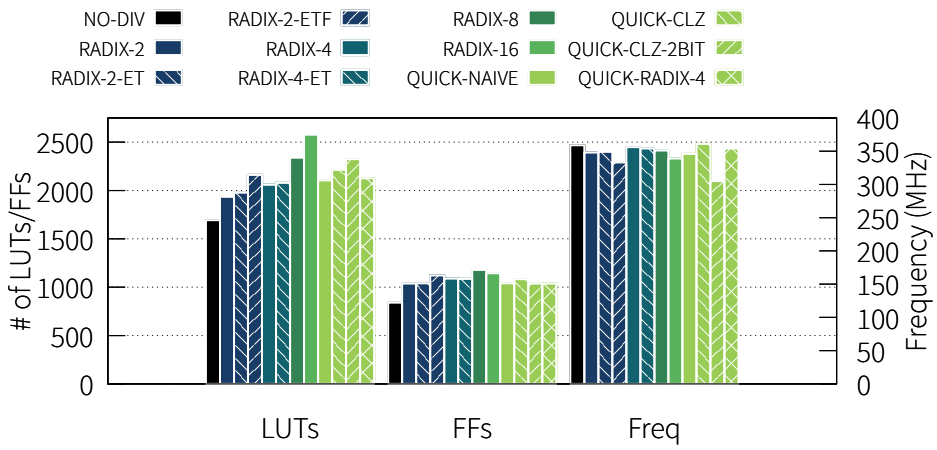

Fig. 12. Operating frequency and resource usage for the Taiga soft-processor integrated with dividers

and *Quick-radix-4*, there is a further 10% increase in LUT usage, or 192 more LUTs. In this paper, we purposely selected a small configuration for the baseline Taiga processor (no caches, exceptions or virtual memory support) to show that if Quick-radix-4 provides the highest performance per LUT for this configuration, then it will hold for larger processor configurations as well.

4.4 Performance and Performance/Resource Comparison

4.4.1 Microbenchmark Analysis. To analyze the performance and performance/resource of our *Quick-Div* dividers in relation to the radix-N based dividers, we created a variety of microbenchmarks to explore different characteristics of the dividers. All tests are performed by iterating over arrays of uniformly randomly distributed integer numbers. Unless stated otherwise, all microbenchmarks have four instructions following each division that are not dependent on the division result. As Taiga supports out-of-order execution, by adding these instructions, we allow the processor to effectively hide the latency of the division operations up to four cycles. With a divider unit minimum of four cycles (three for the unit and one for the algorithm), a divider that always achieves the minimum latency would not incur any dependency stalls. Figure 13 compares the performance (IPC) of all dividers using the following microbenchmarks. For each microbenchmark, the percentage of division instructions (as a percentage of executed instructions) is listed below the benchmark name. As can be seen, division percentages range from 5.55% up to 23.5%. Additionally, as a measure of the upper bound on performance in each of these microbenchmarks, an *upper-bound* bar has been added where the division instructions have been replaced with a single ”add” instruction, which has a latency of one cycle.

(1) ***div-by-one*** is a worst-case scenario for the *Quick-Div* dividers. Here, one array of numbers is constantly divided by one, which is implemented with inline assembly to prevent the compiler optimizing away the division. For *Quick-Div*, it takes $population\_count(dividend)$ cycles, and in the worst case, where the dividend is $2^{32} - 1$, 32 cycles. On average, 16-bits per number will be set. Therefore, *Quick-Naive*, *Quick-CLZ* and *Quick-radix-4* achieve results in line with *radix-4* which has a fixed latency of 16-cycles. This test case also shows the potential benefit of the *Quick-CLZ-2BIT* design which can generate two consecutive bits of

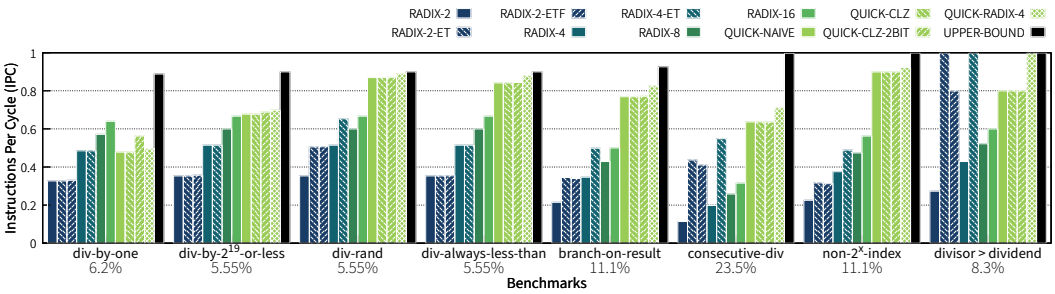

Fig. 13. Microbenchmark performance (IPC) comparison. The percentages underneath each microbenchmark indicate the percentage of instructions executed that were division operations. radix-r-ET designs have logic to perform Early-Termination on the first cycle if the divisor is larger than the dividend. The ETF Early-Termination-Full variant has an extra cycle of latency for termination, but can terminate on any cycle as soon as the Quotient is complete.

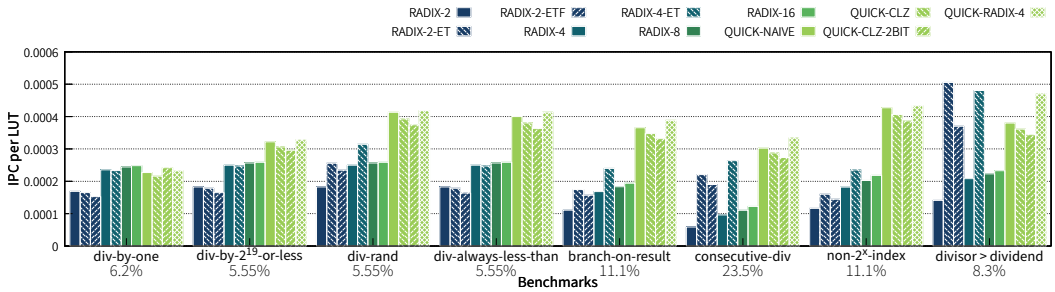

Fig. 14. Microbenchmark performance per LUT (IPC/LUT) comparison

the quotient per cycle. Its IPC is slightly lower than *radix-8* due to an additional fixed cycle of latency it has over the *radix-8* design.

(2) ***div-by-2^{19}-or-less*** is a case where the divisor is always 2^{19} or less: 2^{19} was found as the crossover point where *Quick-Div* starts to perform better than *radix-16*. This was found by increasing the divisor range one power-of-two at a time.

(3) ***div-rand*** performs division on two unconstrained uniformly randomly generated numbers. As such, there is a 50% chance for any pairing that the divisor will be greater than the dividend. In this case, we see the divider designs with early-terminate logic and the *Quick-Div* dividers perform better as they are able to terminate early without performing the division.

(4) ***div-always-less-than*** is the same as *div-rand*, except the divisor is selected to always be less than the dividend. Between *div-always-less-than* and *div-rand*, we see that the performance of the dividers is largely the same with the exception of the early-termination designs. In the *div-rand* benchmark, approximately half of the time, these dividers can terminate before starting their iterative process. In the case of *radix-2*, with an additional 36 LUTs for early termination logic, performance can be improved by 39%. Of note here is that, the *Quick-Div* algorithms achieve results that are quite close to the *reference-add* results (upper bound), indicating that any further latency reduction will bring only minimal improvements to overall throughput.

(5) ***branch-on-result*** illustrates the instruction pattern that the next instruction after the division is a branch on the division result, compared to *div-rand* where there are four subsequent

instructions that do not depend on the division result. This increases the impact the division latency has, and can be seen by the fact that the IPC for the *Quick-Div* dividers have similar IPC to the *div-rand* case whereas the performance of the radix-N dividers has decreased.

(6) ***consecutive-div*** has four consecutive divisions, each dividing the result of the previous division. It takes the trend in *branch-on-result* further, with each division instruction depending on the previous result. As such, the likelihood of the divisor being larger than the dividend increases and thus we see higher performance from early-terminate logic.

(7) ***non-2^x-index*** iterates through an array modulus 763 for an index up to $4 * 763$. As such, the first quarter will trigger early-termination logic, while the next three quarters will be divisions that are very close in magnitude. Thus, these divisions will complete in at most two iterations for the *Quick-Div* dividers providing the large performance benefit we see here for *Quick-Div*. For *Quick-radix-4*, this results in a 4.1x increase in performance over *radix-2*.

(8) ***divisor>dividend*** is a best case scenario where all division operations can utilize the early-termination condition. For the *radix-2* divider, the performance increase with early-termination support can reach an IPC of effectively one. This is due to its latency going down to four cycles (1 cycle for the algorithm and 3 in the remainder of the processor's pipeline). With four non-dependent instructions following each div in the benchmark, at this point the latency of the instruction can be hidden by other execution with the Taiga processor allowing an IPC of one to be obtained.
For *divisor>dividend* and a few other microbenchmarks (particularly those where the early-terminate dividers perform better than their regular radix-N counterparts) we see that the *Quick-radix-4* implementation performs slightly better than the other *Quick-Div* variants. This is due to *Quick-radix-4* performing its divisor > dividend check in its first cycle allowing it to terminate a cycle earlier than the other dividers. It would be possible to add additional logic to the other *Quick-Div* dividers to achieve this, but would not be resource efficient.

Summary. To summarize, besides *division-by-one* or cases where division/remainder operations always terminate early, *Quick-Div* provides the highest performance gains lifting IPC values from as low as 0.22 to 0.92 in some cases. In fact, for some cases, *Quick-Div*'s performance can reach very close to the upper-bound of what can be achieved. Our first design variant, *Quick-clz-2bit* shows only marginal gains in a couple of microbenchmarks, with the only real benefit seen in *division-by-one*. If we factor in the lower clock frequency, *Quick-clz-2bit* actually degrades performance in all cases. Our second design variant, *Quick-radix-4* is able to achieve a slightly higher performance than the original *Quick-CLZ* design due to two factors. Firstly, it can early-terminate a cycle sooner without additional logic and thus performs better when the divisor is greater than the dividend. Secondly, due to the way *Quick-Div* works, the alignment shifts are only an estimate and can overestimate. As *Quick-CLZ* continually re-aligns, the way we determine completion is to check if the divisor is greater than the remainder. If that comparison is true, the done signal is asserted the following cycle. Due to the pre-calculation of the number of cycles required, *Quick-radix-4* can sometimes terminate a cycle sooner as we always know when we are on the last iteration required. Again, we could add additional logic to *Quick-CLZ* to achieve this, but it would further increase the performance efficiency advantage that *Quick-radix-4* has over *Quick-CLZ* as is shown in Figure 14.

Figure 14 presents the results of Figure 13 normalized to each processor configuration's LUT count, i.e., performance-per-LUT. Here the comparative advantage of the *Quick-Div* dividers increases over the higher radix dividers, as both *radix-8* and *radix-16* dividers use more resources than *Quick-Div*. In terms of performance-per-LUT, the *Quick-Div* dividers are always better than the *radix-2* divider, which is the most widely used divider choice for soft-processors.

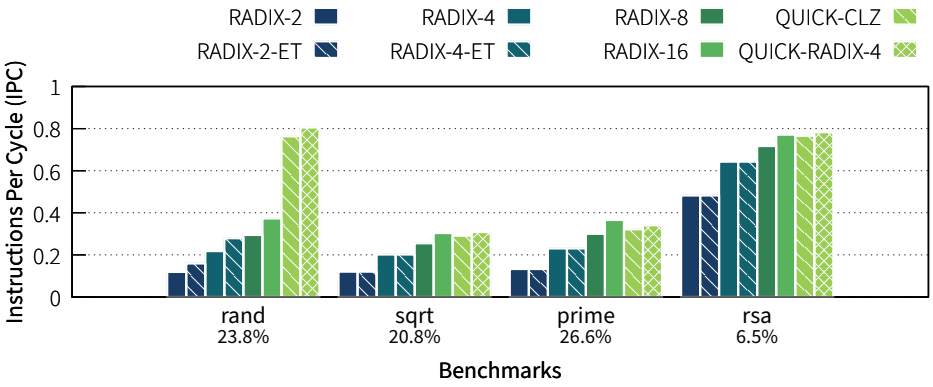

Fig. 15. Practical benchmark performance (IPC) comparison. The percentages underneath each microbenchmark indicate the percentage of instructions executed that were division operations.

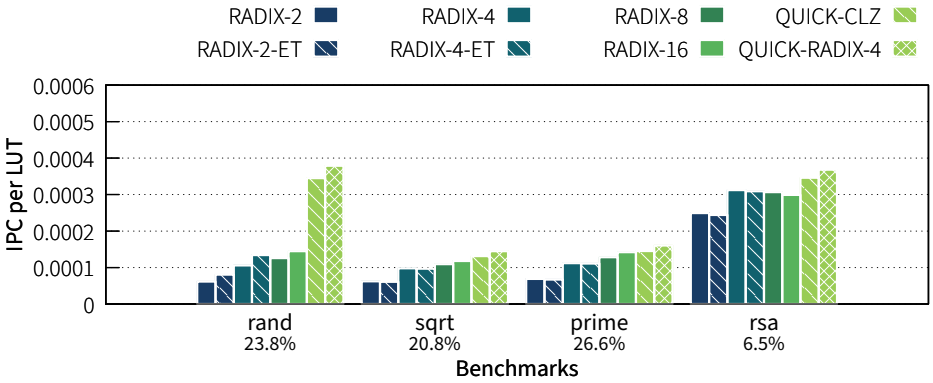

Fig. 16. Practical benchmark performance-per-LUT comparison.

4.4.2 Division-Heavy Benchmarks. Now we evaluate the performance and performance/resource impact of the dividers for benchmarks that represent more practical use cases. We remove the *radix-2-ETF* as its benefit was negligible over *radix-2* in the microbenchmarks examined. This is due to how *radix-2* operates, resolving one bit per cycle starting with the most-significant-bit. Thus, its early termination case (where the divisor is greater than the partial remainder) can only help if none of the low order bits are set for the quotient, which rarely occurs (all odd quotients have the least-significant-bit set). We also remove *Quick-CLZ-2BIT* as it did not meet timing and only showed an improvement for the *div-by-one* microbenchmark. *Quick-Naive* is also removed as it did not meet timing.

The benchmarks we use include:

(1) ***random*** a random number function based on C++11's `minstd_rand0` [14], which makes use of the remainder
(2) ***sqrt*** a square root algorithm using Newton's method [33]
(3) ***prime-check*** a benchmark that determines if a number is prime through successive division
(4) ***rsa-32-decrypt*** a benchmark that performs 32-bit RSA decrypt [23], which makes use of the remainder.

Figure 15 presents the IPC results for these benchmarks, for the remaining dividers, and Figure 16 presents the performance-per-LUT results.

The first benchmark, *random*, performs similarly to the *non-power-of-two* microbenchmark. In this benchmark random numbers are generated with a remainder operation, resulting in a mixed amount of modulus operations where the divisor is larger than the dividend. As a result, the early-terminate designs perform better here than their non early-terminate counterparts. Here we see a performance gain of over 6.8x for *Quick-radix-4* compared to radix-2 and over 6.1x the improvement in performance-per-LUT.

sqrt (Newton's method) has lower IPC values as this benchmark is similar to *branch-on-result*, where, immediately after computing the division operation a branch is evaluated on the result. Additionally, *Quick-Div* requires fewer iterations the closer the log2 difference is between the two numbers. Square root operations have a larger spread between the input and the square root of the input, which lessens the benefit that *Quick-Div* can provide.

prime-check determines if a number is prime by successive division. As the division starts from small numbers and is dependent on previous results, this benchmark is similar to *branch-on-result* and in between the *div-by-one* and the *div-by-*2^{19}*-or-less* microbenchmarks in terms of performance characteristics. Even so, it performs better than *radix-8* and has the highest performance-per-LUT. While *radix-16* has slightly higher IPC, its performance would be lower as it lowers the processor's clock frequency by more than 10% compared to our target frequency.

rsa-32-decrypt performs the RSA-32 decryption, which utilizes the remainder operator. Here, *Quick-radix-4* provides a 1.6x speedup compared to *radix-2* and has the highest performance-per-LUT.

Summary. In summary, through the selection of a *Quick-Div* algorithm, and in particular, *Quick-radix-4*, application performance can improved by over 6.8x in some cases. And even in cases where only a fraction of one percent of instructions are division operations, a respectable 8% improvement in performance can be obtained.

4.4.3 Embench Benchmarks. While the previous benchmark set were selected to include some heavy division use cases, to further examine the impact divider performance can have, in this section we present the impact of divider selection on the Embench [6] benchmark suite. Within this set of 19 benchmarks, eight were found to use hardware division instructions. Of those eight benchmarks, division execution percentages varied from 0.09% up to 3.4% which are lower than the examples in the previous section. The results are presented in Figure 17 for IPC and Figure 18 for IPC-per-LUT.

As can be seen in Figure 17, the *Quick-Div* dividers provide the highest performance in all cases, ranging from a minimum of 3% improvement (*nettle-aes*) to an improvement of 66% in the case of *ud*. While some of these improvements are small, as is the case for *nettle-aes*, the improvement is obtained with less than 0.1% of instructions executed being division operations. Even for small execution proportions, such as 1% (*wikisort*), we can find a 16% improvement. Another interesting observation that can made from these results is the benefit of early-termination logic. In multiple benchmarks (*minver, sglib-combined, ud*), we see improved performance from the early-terminate variants, in some cases surpassing the performance of the next higher radix implementation (*sglib-combined, ud*). Given that the additional logic for early-terminate is small, in these cases *radix-2-early-terminate* would be a good replacement for *radix-2* even in resource constrained systems. Comparing *Quick-radix-4* to *Quick-CLZ*, we can see that *Quick-radix-4* achieves slightly higher performance in all benchmarks with the exception of *wikisort*. As *Quick-CLZ* has the possibility of lower latency compared to *Quick-radix-4* depending on its inputs, it is possible that the number set in this benchmark favours *Quick-CLZ*.

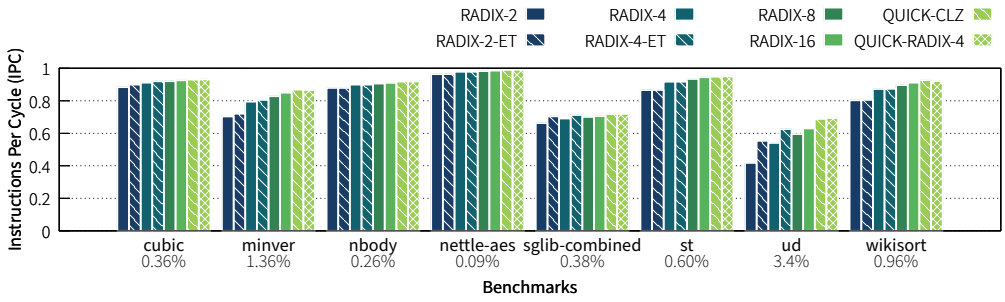

Fig. 17. Embench benchmark performance (IPC) comparison. The percentages underneath each microbenchmark indicate the percentage of instructions executed that were division operations.

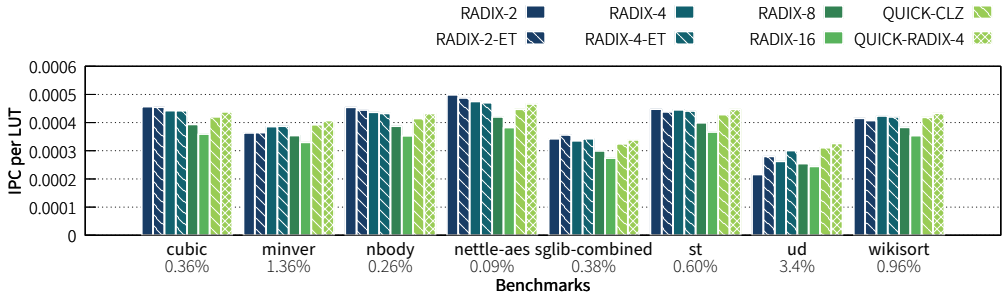

Fig. 18. Embench performance per LUT comparison.

In Figure 18 we present the IPC for each benchmark normalized to the LUT usage of Taiga with each respective divider. We see that in the worst case (*nettle-aes*) the *Quick-CLZ* design has only 94% the performance efficiency of a *radix-2* (although still providing a 3% performance improvement). However, by the time division instructions reaches just 1% of executed instructions, the *Quick-Div* dividers have the highest IPC-per-LUT of all of the dividers.

4.4.4 Power and Energy Commentary. Power estimates were collected using Vivado on the XCVU9P FPGA (at 370 MHz). Approximately 90% of the estimated power for the complete system (~2.7W) was static power draw (in part due to only ~1% of device LUT resources being used for these designs). Dynamic power estimates for Taiga configurations with *radix-2* and *Quick-radix-4* were found to be ~0.11W and ~0.12W respectively for just the processor itself. Between the *radix-2* and *Quick-radix-4* design there is a 10% increase in LUT usage which approximately matches the increase we see in the power estimate. As a result of the variability in place-and-route results that we discussed earlier, we found that power consumption variability with small HDL changes was as large as this 10% difference in LUT requirements. Thus, while the *Quick-radix-4* design can be expected to have slightly higher power requirements, that actual difference can be lost in the place-and-route variability. Additionally, the divider logic will only toggle during division instructions due to clock enables in the divider unit, further reducing the dynamic power impact for the processor as a whole.

In an embedded system, energy usage can be just as, or more important than maximum power consumption. For non-continuous operation, (where a task is run to completion and the system is then powered down), the *Quick-Div* algorithm's shorter runtimes allow for the whole system to be

powered down sooner, reducing the energy requirements. In systems that use caches, this could also include off-chip resources such as DDR memory. Thus, we can conservatively say that in any case where *Quick-radix-4* is more performance efficient per-LUT than *radix-2* that it would also be more energy efficient than *radix-2*.

5 CASE STUDY USING QUICK-DIV

To demonstrate that our *Quick-Div* dividers provide opportunities for simpler and faster algorithmic choices on soft-processors, we further investigate the square root benchmark as introduced in Section 4.4. A quick Internet search produces many different software implementations of integer square root, with almost all of them avoiding the use of the division. We chose the three best performing algorithms that avoid divisions as comparison points against the Newton's method [33] that uses divisions.

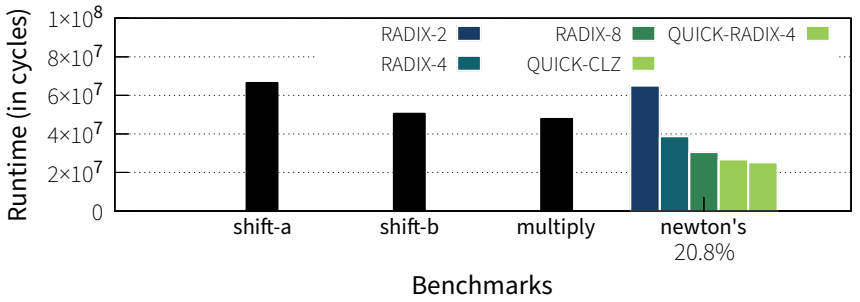

Fig. 19. Runtime comparison between different integer square root algorithms: shift-a, shift-b, and multiply use division-avoiding algorithms

The first algorithm *shift-a* [4] determines one bit of the result per iteration over 16 iterations, as the max square root for a 32-bit number can be at most $2^{16}-1$. The second algorithm *shift-b* [4] finds the highest power-of-four less than the input number, and performs only shifts and subtractions to obtain the result. For smaller numbers, it requires fewer iterations than the *shift-a* algorithm. The third algorithm, which we labelled as *multiply* [26], is one of only a few algorithms that uses multiplications. This algorithm uses multiplication to square the current guess and compare against the input number. Its starting point for the guess is at 2^{16}.

Newton's method requires one division per iteration and uses the following formula where n is the number for which you are finding the square root of: $nextGuess = (guess + n/guess)/2$. The exit condition occurs when $n/guess$ is less than $nextGuess$. We experimented with different initial guesses and approximations before settling on $2^{16}-1$, which provides the best performance for this platform.

The results of the four benchmarks are plotted in Figure 19. Each algorithm's performance was measured by performing the square root of over a million random numbers and measuring overall runtime (in cycles). All non division-based sqrt algorithms are able to achieve a high IPC (> 0.74) and are not bottlenecked by other processor resources. The highest source of stalls for these benchmarks are branch predictor mispredicts, with the Newton's benchmark having a similar quantity of mispredict stalls. As all algorithms other than Newton's do not use division, only one data point is provided for each of them. As shown in Figure 19, with only a *radix-2* divider in the system, the *multiply* algorithm provides the fastest runtime. However, for any divider selection other than *radix-2*, Newton's method results in a better runtime. The overall speedup is 1.9x for

Quick-radix-4 over the *multiply* algorithm and a 1.5x improvement in performance-per-LUT over no divider in the system.

6 CONCLUSION AND FUTURE WORK

In this work, we have found that substantial performance gains can be obtained in soft-processor systems by replacing the commonly used *radix-2* divider with the Quick-Div dividers presented in this work. These performance gains can be as much as 6.8x in applications with heavy division usage when employing our best performing *Quick-Div* divider, *Quick-radix-4*. Further, improvements of 16% can be achieved when about 1% of the workload is composed of division instructions while still being more resource efficient than a *radix-2* divider. Additionally, we presented a detailed analysis of *Quick-Div*, in terms of algorithm analysis, implementation considerations, and performance characteristics. Against comparisons of other dividers, including higher radix dividers, our *Quick-radix-4* design provides the highest achievable performance when factoring in throughput (IPC) and processor clock frequency across a range of benchmarks. Through our case study, we demonstrated that, at the software level, algorithmic choices may need reevaluation when comparing the performance of *Quick-Div* to the commonly used *radix-2* divider. Future work could further explore adapting existing software to better leverage the performance of *Quick-Div* including examining existing math libraries and compiler optimizations.

7 ACKNOWLEDGMENTS

We acknowledge the support from NSERC Discovery Grant RGPIN341516, RGPIN-2019-04613, DGECR-2019-00120, Alliance Grant ALLRP-552042-2020, COHESA (NETGP485577-15), CWSE PDF (470957); CFI John R. Evans Leaders Fund; Simon Fraser University New Faculty Start-up Grant.

REFERENCES

[1] Cobham Gaisler AB 2021. *GRLIB IP Core User's Manual.* Cobham Gaisler AB. gaisler.com/products/grlib/grip.pdf

[2] G. Cornetta and J. Cortadella. 2001. A multi-radix approach to asynchronous division. In *Proceedings Seventh International Symposium on Asynchronous Circuits and Systems. ASYNC 2001.* 25–34. https://doi.org/10.1109/ASYNC.2001.914066

[3] J. Cortadella and T. Lang. 1994. High-radix division and square-root with speculation. *IEEE Trans. Comput.* 43, 8 (1994), 919–931. https://doi.org/10.1109/12.295854

[4] Jack W. Crenshaw. 1998. Integer Square Roots. https://www.embedded.com/electronics-blogs/programmer-s-toolbox/4219659/Integer-Square-Roots

[5] Florent de Dinechin and Laurent-Stéphane Didier. 2012. Table-Based Division by Small Integer Constants. In *Proceedings of the 8th International Conference on Reconfigurable Computing: Architectures, Tools and Applications* (Hong Kong, China) *(ARC'12)*. Springer-Verlag, Berlin, Heidelberg, 53–63. https://doi.org/10.1007/978-3-642-28365-9_5

[6] Embench™ Task Group. 2019. Embench™: Open Benchmarks for Embedded Platforms. https://github.com/embench/embench-iot.

[7] M.D. Ercegovac, T. Lang, J.-M. Muller, and A. Tisserand. 2000. Reciprocation, square root, inverse square root, and some elementary functions using small multipliers. *IEEE Trans. Comput.* 49, 7 (2000), 628–637. https://doi.org/10.1109/12.863031

[8] X. Fang and M. Leeser. 2013. Vendor agnostic, high performance, double precision Floating Point division for FPGAs. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–5. https://doi.org/10.1109/HPEC.2013.6670335

[9] Xin Fang and Miriam Leeser. 2016. Open-Source Variable-Precision Floating-Point Library for Major Commercial FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 9, 3, Article 20 (July 2016), 17 pages. https://doi.org/10.1145/2851507

[10] Robert Goldschmidt. 1964. *Applications of division by convergence.* Ph.D. Dissertation.

[11] C. Heinz, Y. Lavan, J. Hofmann, and A. Koch. 2019. A Catalog and In-Hardware Evaluation of Open-Source Drop-In Compatible RISC-V Softcore Processors. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–8. https://doi.org/10.1109/ReConFig48160.2019.8994796

[12] K. S. Hemmert and K. D. Underwood. 2007. Floating-Point Divider Design for FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 15, 1 (Jan 2007), 115–118. https://doi.org/10.1109/TVLSI.2007.891099

[13] Intel Corp. 2020. *Nios II Gen2 Processor Reference Guide.* Intel Corp. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/n2cpu-nii5v1gen2.pdf

[14] ISO/IEC 14882:2011 2011. *Information technology – Programming languages – C++.* Standard. International Organization for Standardization, Geneva, CH.

[15] Salman Khan. 2015. *VHDL Implementation and Performance Analysis of two Division Algorithms.* Master's thesis. University of Victoria.

[16] Daniel Lemire. 2017. Fast exact integer divisions using floating-point operations. https://lemire.me/blog/2017/11/16/fast-exact-integer-divisions-using-floating-point-operations/.

[17] B. Liebig and A. Koch. 2014. Low-latency double-precision floating-point division for FPGAs. In *2014 International Conference on Field-Programmable Technology (FPT)*. 107–114. https://doi.org/10.1109/FPT.2014.7082762

[18] Ligomenides. 1977. The Skip-and-Set Fast-Division Algorithm. *IEEE Trans. Comput.* C-26, 10 (1977), 1030–1032. https://doi.org/10.1109/TC.1977.1674740

[19] E. Matthews, Z. Aguila, and L. Shannon. 2018. Evaluating the Performance Efficiency of a Soft-Processor, Variable-Length, Parallel-Execution-Unit Architecture for FPGAs Using the RISC-V ISA. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Vol. 00. 1–8. https://doi.org/10.1109/FCCM.2018.00010

[20] E. Matthews, A. Lu, Z. Fang, and L. Shannon. 2019. Rethinking Integer Divider Design for FPGA-Based Soft-Processors. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 289–297. https://doi.org/10.1109/FCCM.2019.00046

[21] E. Matthews and L. Shannon. 2017. TAIGA: A new RISC-V soft-processor framework enabling high performance CPU architectural features. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–4. https://doi.org/10.23919/FPL.2017.8056766

[22] P. Montuschi and L. Ciminiera. 1991. Simple radix 2 division and square root with skipping of some addition steps. In *[1991] Proceedings 10th IEEE Symposium on Computer Arithmetic.* 202–209. https://doi.org/10.1109/ARITH.1991.145560

[23] Juan Manuel Torres Palma. 2016. The Simple C RSA-32 implementation. https://github.com/jmtorrespalma/sc-rsa

[24] Charles Papon. [n.d.]. VexRiscv. https://github.com/SpinalHDL/VexRiscv

[25] S. K. Park and K. W. Miller. 1988. Random Number Generators: Good Ones Are Hard to Find. *Commun. ACM* 31, 10 (Oct. 1988), 1192–1201. https://doi.org/10.1145/63039.63042

[26] Microchip Technology Inc. Ross M. Fosler. 2000. Fast Integer Square Root. http://ww1.microchip.com/downloads/en/AppNotes/91040a.pdf

[27] Wilson Snyder. 2018. Verilator 4.008. https://www.veripool.org/ftp/verilator_doc.pdf
[28] Gustavo Sutter, Gery Bioul, and Jean-Pierre Deschamps. 2004. Comparative Study of SRT-Dividers in FPGA. In *Field Programmable Logic and Application*, Jürgen Becker, Marco Platzner, and Serge Vernalde (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 209–220.
[29] G. Sutter and J. Deschamps. 2009. High speed fixed point dividers for FPGAs. In *2009 International Conference on Field Programmable Logic and Applications*. 448–452. https://doi.org/10.1109/FPL.2009.5272492
[30] Rainer K. L. Trummer. 2005. *A High-performance Data-dependent Hardware Integer Divider.* Master's thesis. University of Salzburg.
[31] VectorBlox. [n.d.]. ORCA: RISC-V by VectorBlox. github.com/VectorBlox/orca(mirror:https://github.com/riscveval/orca-1)
[32] Xiaojun Wang. 2007. *Variable Precision Floating-Point Divide and Square Root for Efficient FPGA, Implementation of Image and Signal Processing Algorithms.* Ph.D. Dissertation. EECS Department, Northeastern University.
[33] Wikipedia. 2021. Square root. https://en.wikipedia.org/wiki/Square_root
[34] Xilinx Inc. 2019. *MicroBlaze Processor Reference Guide.* Xilinx Inc. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug984-vivado-microblaze-ref.pdf