Rethinking Integer Divider Design for FPGA-based Soft-Processors

Eric Matthews, Alec Lu, Zhenman Fang and Lesley Shannon
School of Engineering Science, Simon Fraser University
{ematthew, alec_lu, zhenman_fang, lshannon}@sfu.ca

***Abstract*—Most existing soft-processors on FPGAs today support a fixed-latency instruction pipeline. Therefore, for integer division, a simple fixed-latency radix-2 integer divider is typically used, or algorithm-level changes are made to avoid integer divisions. However, for certain important application domains the simple radix-2 integer divider becomes the performance bottleneck, as every 32-bit division operation takes 32 cycles.**

In this paper, we explore integer divider designs for FPGA-based soft-processors, by leveraging the recent support of variable-latency execution units in their instruction pipeline. We implement a high-performance, data-dependent, variable-latency integer divider called Quick-Div, optimize its performance on FPGAs, and integrate it into a RISC-V soft-processor called Taiga that supports a variable-latency instruction pipeline. We perform a comprehensive analysis and comparison—in terms of cycles, clock frequency, and resource usage—for both the fixed-latency radix-2/4/8/16 dividers and our variable-latency Quick-Div divider with various optimizations. Experimental results on a Xilinx Virtex UltraScale+ VCU118 FPGA board show that our Quick-Div divider can provide over 5x better performance and over 4x better performance/LUT compared to a radix-2 divider for certain applications like random number generation. Finally, through a case study of integer square root, we demonstrate that our Quick-Div divider provides opportunities for reconsidering simpler and faster algorithmic choices.

I. Introduction

Division, one of the fundamental arithmetic operations, is utilized in a range of computations from the simple, such as non-power-of-two indexing and random number generation [1] to complex use cases such as image processing [2]. There are mainly two types of dividers: floating-point/fixed-point dividers that produce a single fraction quotient; and integer dividers that return two integer results, including a quotient and a remainder. Over the years, the floating-point/fixed-point divider has received considerably more attention and design improvements in both commodity hard processors and FPGA-based soft-processors [2]–[8], due to its frequent usage in applications. Meanwhile, integer divider improvement has been overlooked, due to its less frequent usage and the fact that it can be emulated with a floating-point/fixed-point divider with a sufficient degree of numerical precision. As a result, the floating-point dividers in most commodity x64 processors are capable of outperforming the integer dividers by nearly 2x in throughput when calculating 32-bit integer divisions [9].

To overcome the inefficiency of integer division on commodity x86 processors, compiler optimizations and efficient libraries are usually exploited to avoid using the underlying integer dividers directly. This has led to proposals for utilizing double-precision floating point logic to accelerate 32-bit integer division, compiling programs with predefined divisors, or to avoid integer division all together through algorithmic choices [9]. Although these optimizations may sometimes be effective for x64 processors, in an FPGA-based soft-processor, where hardware resources are often constrained, emulating integer division using a floating-point/fixed-point divider is inefficient in terms of performance/resource. For example, a typical FPGA 64-bit floating-point divider occupies a few thousand LUTs, more than 10x that a simple radix-2 32-bit integer divider uses [4], [6].

In fact, most existing FPGA-based soft-processors—including the widely used MicroBlaze [10], NIOSII [11] and the LEON3 processor [12]—use a simple fixed-latency radix-2 divider with 32 cycles of latency, if they support an integer divider at all. The choice of a simple radix-2 divider is made based on its smaller resource usage and the fact that most of these soft-processors implement a simple instruction pipeline that only supports fixed-latency arithmetic logic units (ALUs). While simple arithmetic operations typically only take one or two cycles, the 32-cycle radix-2 divider makes integer division one of the slowest operations in the soft-processors. This can cause a significant performance loss for certain important application domains that heavily use integer divisions, such as random number generation, non-power-of-two indexing, and data shuffling [9]. For example, for random number generation, our experimental results show that the simple radix-2 divider can cause more than 5x performance slowdown compared to an optimized divider implementation.

In this paper, we explore integer divider design for FPGA-based soft-processors, optimizing for performance and performance per LUT with the constraint of not reducing the soft processor's operating frequency. By leveraging the support of variable-latency operations in Taiga [13], we explore both fixed-latency radix-based (radix-2/4/8/16) dividers and variable-latency, data-dependent integer dividers. With a quantitative analysis and comparison in terms of operating cycles, clock frequency and resource usage, we find that our optimized variable-latency data-dependent integer divider, *Quick-Div*, can outperform the widely used radix-2 divider by over 5x in performance and over 4x in performance/LUT, in certain applications like random number generation. Finally, to demonstrate that our *Quick-Div* divider offers opportunities for simpler and faster algorithms for design problems that use integer divisions, we also conduct a case study for integer square root: with our *Quick-Div*, a straightforward algorithm using integer division can now outperform a more complex division-avoiding algorithm by 1.6x in performance.

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 operating cycles, clock frequency, and resource usage.
2. A highly-optimized variable-latency integer divider, called *Quick-Div*, which provides better performance and performance/resource for FPGA-based soft-processors.
3. A case study demonstrating that our *Quick-Div* can provide simpler and faster algorithm choices for soft-processors.

II. Background and Related Work

A. Basics of Integer Dividers

Integer dividers, also called whole dividers, calculate the quotient and remainder of the divisor from the dividend, all of which are integer numbers, such that 1) the dividend equals the sum of the remainder and the product of the divisor and the quotient, and 2) the remainder is less than the divisor and non-negative. Next we describe the two major integer dividers: fixed-latency and variable-latency dividers.

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, since it consumes minimal resources and can be easily integrated into most soft-processor's fixed-latency instruction pipeline.

A general fixed-latency radix-N division algorithm determines the quotient from the most significant bit to the least significant bit. Each cycle, log_2N bits of the quotient is determined, representing the largest divisor multiple 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. At the end of the algorithm, the partial remainder becomes the remainder as no further divisor multiples can be subtracted.

The latency for a radix-N divider is determined by the bit width of the longest operand (i.e., 32 in an 32-bit integer divider), as well as the number of quotient bits that the divider calculates per cycle, i.e., log_2N. Analytically, the latency is ceil($32/log_2N$). Meanwhile, resource usage scales with the radix number as it indicates the number of trial subtractions required to find the quotient. In the context of a FPGA-based soft-processor, where resource is limited, high-radix integer dividers such as radix-16 or higher are not practical options as is shown in this paper.

2) Variable-Latency Data-Dependent Divider: Another alternative 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 dividend 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 set bits in the resulting quotient.

Figure 1 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 most-significant-bit (MSB) for both the divisor and the partial remainder, where MSB is equivalent to the floor of log2 of the number. The difference between the two MSBs is the bits that we should left shift the divisor by to align it with the partial remainder. However, as this difference only compares the MSBs for both numbers, it can overestimate the difference resulting in a subsequent subtraction that overflows, when the shifted trial divisor is larger than the dividend. As such, in the case where the estimated divisor overflows, we left shift the divisor by one bit less and perform a second subtraction.

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

Fig. 1. Variable-latency integer divider algorithm

B. Related Work in Dividers for Soft-Processors

1) Related Work in Floating-Point/Fixed-Point Dividers: Due to the importance of floating-point/fixed-point arithmetic in signal processing and other applications on FPGA-based soft-processors, various aspects of floating-point/fixed-point dividers for FPGAs have been explored [2]–[8]. Hemmert et al. explored floating-point divider design across a wide range of design constraints, from pipelined to iterative, as well as division algorithms such as non-restoring and SRT [3]. Likewise, Sutter et al. presented divider architectures for non-restoring power-of-two radix algorithms [7] and studied SRT dividers for FPGAs [8]. Both non-restoring and SRT algorithms have a shorter critical path allowing dividers to run at a higher operating frequency. However, for radix-r dividers up to radix-8, in this work the soft-processor will be the limiting factor in determining the clock frequency thus optimizations for divider operating frequency do not result in increases in performance. Additionally, for the SRT division algorithms, operands are required to be scaled and normalized, which adds extra overhead to the computation in terms of resource usage and latency. Fang et al. and Liebig et al. investigated floating-point division algorithms for FPGAs [4], [6] that offer lower latencies compared to today's commercial dividers from FPGA vendors, but require extra DSP multiplier blocks and BRAMs. One of Liebig et al's designs was able to achieve latency as low as 8 cycles, same as a radix-16 integer divider; however, it has a hardware resource usage twice as much as the radix-16 divider investigated in this paper. Moreover, additional logic and cycles would be required to compute integer division using a floating-point divider.

2) Related Work in Integer Dividers: Existing soft-processors–including the widely used MicroBlaze [10], NIOSII [11] and the LEON3 processor [12]–use a simple fixed-latency radix-2 divider, and all make the inclusion of the divider optional. While algorithms for hardware are often designed to avoid integer division, next we discuss a few cases where high-performance integer dividers are investigated.

Dinechin et al. looked into ways to efficiently compute division by small constants [14]. However, for a soft-processor, full processor data width divisions and remainder operations are required, preventing the techniques in this work from being applied. Variable-latency data-dependent integer divider designs have been investigated in two master's theses by Khan [15] and Trummer [16]. 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 [16], a per-iteration aligning divider is also presented, which is investigated for standalone 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.

With the introduction of open-source soft-processors (e.g., Taiga [17]) that support variable-latency execution units, it provides opportunities to investigate variable-latency integer dividers in a soft-processor environment. In contrast to Trummer's and Khan's work, we are able to 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.

III. 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 RISC-V soft-processor Taiga.

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 II-A2) is that: the closer the dividend and divisor are in magnitude, the fewer cycles the algorithm will take to complete: if the two (decimal) numbers differ by a factor of N or less, the divider will take at most $log_2 N$ iterations. Additionally, when both the dividend and divisor are power-of-two numbers, it requires only a single iteration.

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 2(a), as the bit-width of the *Quick-Div* divider increases, the average latency of the division increases up to approximately 1.69 cycles and then flattens out, which is much more attractive than the N cycles latency for a radix-2 based N-bit integer divider.

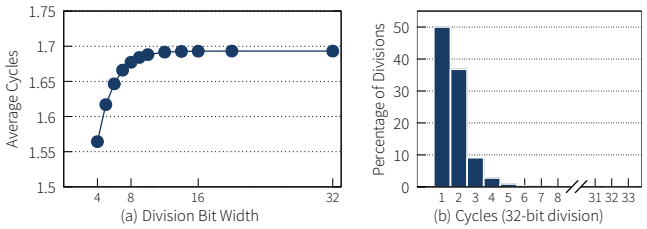

Fig. 2. (a) Average cycles of Quick-Div for division between two uniformly distributed random numbers, with division bit-widths between 4 and 32. (b) Access cycle distribution of Quick-Div for 32-bit division between two uniformly distributed random numbers.

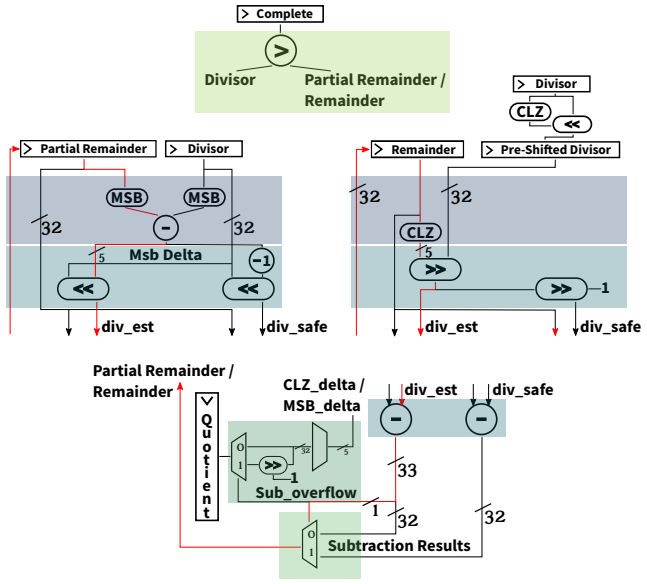

Fig. 3. Variable-latency Quick-Div divider hardware implementation

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 2(b), the vast percentage of divisions take only a few cycles (cycles shown in x-axis) to complete. While most pairings take only a few cycles, there is a long tail to the distribution as 32 iterations is still possible. However, only one number pair, $((2^{32}-1)/1)$, will actually take 32 iterations (i.e., 33 cycles).

A. Quick-Div Divider Design

The initial *Quick-Div* implementation closely follows the pseudo-code that was presented in Figure 1. A schematic of the circuit, along with changes to improve clock frequency and details on the design's critical path is presented in Figure 3. The floor of log2 operations are computed by finding the most significant bit (MSB) for the divisor and partial remainder. In the initial design iteration, these are implemented as 32-bit priority encoders. The difference between the MSBs 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 as the mux 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, highlighted as red line in Figure 3, was found to be from the partial remainder, through the MSB 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.

B. 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 MSB design: The first optimization investigated was to replace the 32-bit MSB priority encoder with a tree structured MSB 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 MSB subtraction.

Trial #2: Avoiding MSB delta and use pre-shift divisor: Instead of using the MSB 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 MSB, then the divisor is left shifted by the partial remainder's MSB 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 MSB. 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 MSB logic with Count Leading Zero (CLZ) logic. CLZ can be calculated with the same kind of circuit (it is equivalent to $32 - MSB$). 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.

Final choice: optimized CLZ, i.e., quick-clz: Finally, we created an optimized version of the CLZ circuit, shown in Figure 4, which computes the upper two bits of the result in one less level of logic 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 indexes 1 and 0, this would result in comparisons needing 30 and 31 bits. As such, we look at computing the CLZ for 4-bit intervals, mux the results pairwise, 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 LUTs required from 30 to 19, and reduces the levels-of-LUTs from three to two for the upper two bits of the CLZ output.

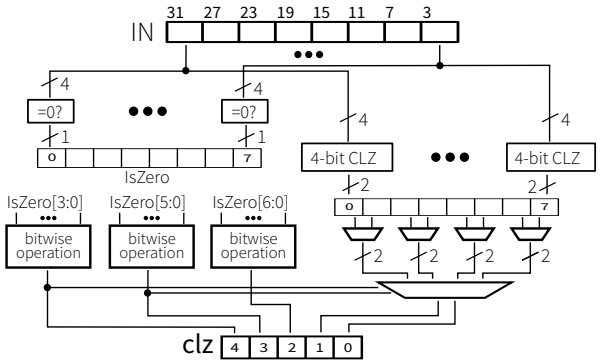

Fig. 4. Optimized CLZ (count leading zero) unit hardware implementation

With this optimized CLZ implementation (called quick-clz), the *Quick-Div* improves the clock frequency by 81MHz, while slightly increasing resource usage, compared to the baseline design without any of these optimizations (called quick-naive). 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.

C. Worst case division optimization: quick-clz-2bit

As discussed in Section III, 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 set bit. One approach to improve such cases would be to try and resolve two consecutive bits per cycle. This is the equivalent of determining if both the estimated divisor and the "safe" divisor can be subtracted from the partial remainder. This adds a third subtractor to the design and increases the final multiplexer size, while reducing some worst case latencies by half, e.g. $(2^{32}-1/1)$ would now require 16 iterations as opposed to the original 32. In Section IV, we will show that while this can help in a few rare corner cases, in practice it has little benefit and it lowers the frequency below what was gained by our initial optimizations (i.e., quick-clz).

D. Integration into Taiga Soft-Processor

Finally, we also integrate our variable-latency *Quick-Div* dividers with the RISC-V soft-processor Taiga [13], [17], which is open-source and supports a seamless integration of variable-latency execution units. As our *Quick-Div* dividers are unsigned, they can require sign conversion before and after completing 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 radix-2 divider. As we wish to evaluate whether the *Quick-Div* divider 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/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 audience to the Taiga papers [13], [17].

IV. Experimental Results

A. Experimental Setup

1) Dividers for Comparison: We evaluate a comprehensive set of radix-based fixed-latency dividers and our variable-latency *Quick-Div* dividers, all of which operate on unsigned integer numbers. The radix-based dividers include the radix-2 divider from Taiga [17] 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-r 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, which is labelled Early-Termination-Full (ETF). For our *Quick-Div* dividers, we evaluate three versions: 1) Quick-Naive, which is the initial design presented in Section III-A without any optimization, 2) Quick-CLZ, which is the version presented in Section III-B with clock frequency optimizations using count leading zeros (CLZ), and 3) Quick-CLZ-2BIT, which is the version presented in Section III-C with worst case optimization.

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 2018.3 synthesis, place and route with all default settings. We leave the detailed description of benchmarks into Section IV-D. While the Taiga processor integrated with these dividers is synthesized to run on-board, the detailed number of instructions and cycles each benchmark takes is collected through the open-source Verilator simulator [18] that simulates the full processor system.

B. Divider Latency Comparison

Table I presents min, max and average cycles for each divider derived by analytical means for the radix-r 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-r 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.

TABLE I
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	32	32	32
Radix-2-ET	0	32	16
Radix-2-ETF	1	32	16.5
Radix-4	16	16	16
Radix-4-ET	0	16	8
Radix-8	11	11	11
Radix-16	8	8	8
Quick-Naive	1	32	1.69
Quick-CLZ	2	33	2.69
Quick-CLZ-2BIT	2	33	2.66

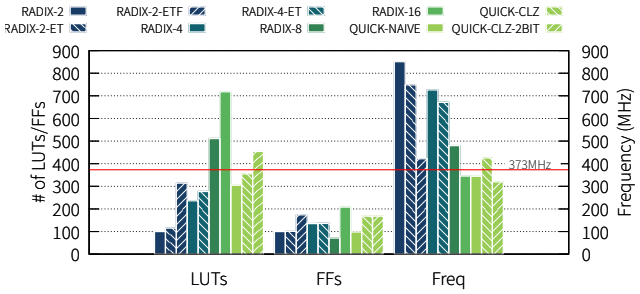

Fig. 5. Operating frequency and resource usage for standalone dividers: the red line is the target frequency

C. Resource Usage and Frequency Comparison

1) Standalone Quick-Div Divider Comparison: Figure 5 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 and clock frequency targets are set separately for each divider as there is a large span in achievable frequencies. 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 be the critical path of the processor.

As shown in Figure 5, the initial implementation of the *Quick-Div* algorithm did not meet the required timing threshold. However, the optimized Quick-CLZ version improved the clock frequency by 81MHz (24%) over the Quick-Naive version, putting its clock frequency at 426MHz, well above the 373MHz threshold. The Quick-CLZ-2BIT version degrades the clock frequency, as it increases the critical path as explained in Section III-C. So, from all three versions of our *Quick-Div*, we choose the Quick-CLZ as our final version.

All radix-based dividers meet the target clock frequency except for radix-16 due to the large number of comparisons required to generate its quotient bits. Additionally, the radix-16 divider requires the largest amount of LUTs approximately double of that the Quick-CLZ divider requires. We still in-

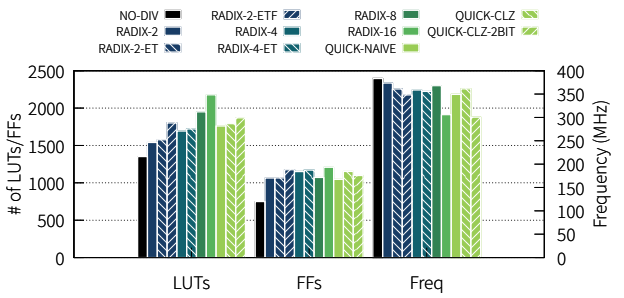

Fig. 6. Operating frequency and resource usage for the Taiga soft-processor integrated with dividers

clude 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 log2 of 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. The Quick-CLZ's resource usage is approximately half way between the radix-4 and radix-8 dividers. But as shown in Table I, the average latency of Quick-CLZ is even lower than the radix-16 divider.

2) Taiga-Quick-Div Integration Comparison: Figure 6 presents the frequency and resource usage for the dividers when integrated into the Taiga processor. We initially found a large variance of more than 20% in the clock frequency obtained by Vivado 2018.3 for this small system, with the no-divider configuration sometimes obtaining the lowest operating frequency. To reduce the variation, we tried to introduce randomness into the placer as discussed in [19]: We added a module to the design—where multiple signals throughout the design are AND-ed with a mask, OR-ed together and assigned to a pin—to prevent the logic from being optimized away. While not a true solution, the frequency variation to within 3%. Additionally, while the Quick-Naive version is reasonably close to the target frequency, it is the critical path in each case; whereas the Quick-CLZ divider is not the critical path in any test case. This further confirms our choice of Quick-CLZ as our final *Quick-Div* implementation.

In terms of the overall resource overhead, the simplest radix-2 divider integration uses 189 more LUTs, or 14% more LUTs, compared to the Taiga processor without any divider. Between radix-2 and Quick-CLZ, there is a further 16% increase in LUT usage, or 252 more LUTs. In this paper, we purposely selected a small configuration for the baseline Taiga processor to show that if Quick-CLZ provides the highest performance per LUT for this configuration it will hold for larger processor configurations as well.

D. Performance and Performance/Resource Comparison

1) Microbenchmark Analysis: To analyze the performance and performance/resource of our *Quick-Div* in relation to the other dividers, we first 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, in order to leverage Taiga's support for out-of-order commits to obtain the highest possible IPC (instruction per cycle). Figure 7 compares the performance (IPC) of all dividers using the following microbenchmarks.

1. *div-by-one* is a worst-case scenario for the *Quick-Div* dividers. Here, one array of numbers is constantly divided by one, (implemented with inline assembly to prevent the compiler optimizing away the division). For *Quick-Div*, it takes $log_2(dividend)$ 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 and Quick-CLZ 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 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.
3. *div-always-less-than* ensures that the divisor is less than the dividend whereas *div-rand* does not. Between them 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%.
4. *divisor>quotient* 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 be up to 228%. For *divisor>quotient* and a few other microbenchmarks we see the original Quick-Naive implementation performs slightly better than Quick-CLZ version: this is due to the extra cycle of latency that Quick-CLZ has. However, we note that the impact is limited to benchmarks where a large proportion of division operations trigger early-termination logic. Additionally, for this benchmark the radix-r-ET algorithms perform slightly better than *Quick-Div* as they have one cycle less latency for the early-termination case, overlapping their early-termination with the cycle before the division algorithm starts. The *Quick-Div* algorithms do not do this as it would require additional resources.
5. *non-power-of-two-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 a most two iterations for the *Quick-Div* dividers providing the large performance benefit we see here for *Quick-Div*. For Quick-CLZ, this results in a 3.8x increase in performance over radix-2.
6. *branch-on-div-result* illustrates the instruction pattern that the next instruction after the division is a branch on the

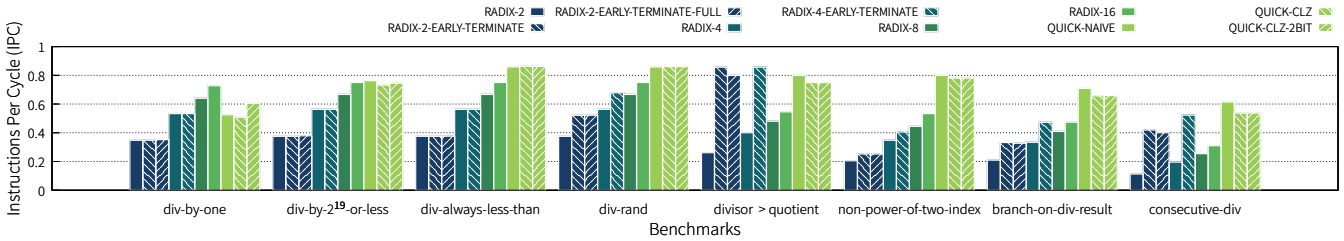

Fig. 7. Microbenchmark performance (IPC) comparison.

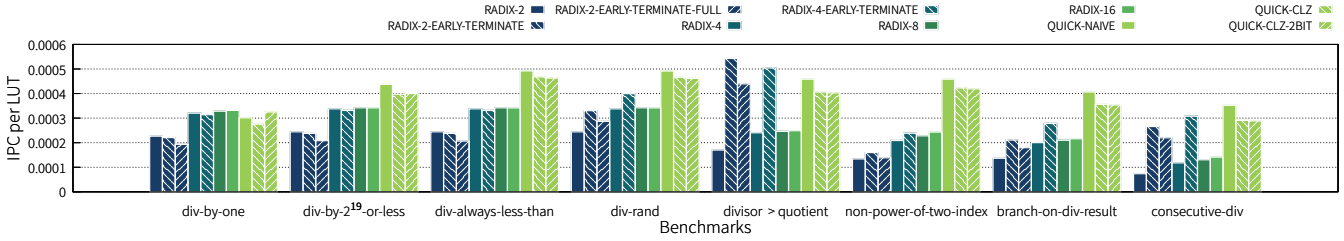

Fig. 8. Microbenchmark performance per LUT (IPC/LUT) comparison

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-r dividers has decreased.

7. *consecutive-div* has four consecutive divisions, each dividing the result of the previous division. It takes the trend in *branch-on-div-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.

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 0.2 to 0.8 in some cases, which leaves little space for further improvement for a single-issue processor design.

Figure 8 presents the results of Figure 7 normalized to each processor configuration's LUT count, i.e., performance/LUT. Here the comparative advantage of the *Quick-Div* approach 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* is always better than the radix-2 divider, which is the most widely used divider choice for soft-processors.

2) Practical Benchmarks: Now we evaluate the performance and performance/resource impact of the dividers for benchmarks that represent 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 terminating early with this approach 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: *Dhrystone* [20], a standard processor integer benchmark; *random*, a random number function based on C++11's `minstd_rand0` [21], which makes use of the remainder; *sqrt*, a square root algorithm using Newton's method [22]; *prime-check*, a benchmark that determines if a number is prime through successive division; and *rsa-32-decrypt*, a benchmark that performs 32-bit RSA decrypt [23], which makes use of the remainder. Figure 9 presents the IPC results for these benchmarks, for the remaining dividers, and Figure 10 presents the performance per LUT results.

Dhrystone is an interesting case. Only approximately 0.24% of dynamically executed instructions are divisions, however, Quick-CLZ can provide a 7% increase in performance over a radix-2 divider. As only multiply and load instructions have latencies more than one cycle, the 32-cycle radix-2 divider can have a reasonable impact on overall performance. Dhrystone is the only case where Quick-CLZ does not provide the highest performance per LUT, but the degradation is within 12%.

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 5x for Quick-CLZ compared to radix-2 and over 4x the improvement in IPC/LUT.

sqrt (newton's) has lower IPC values as this benchmark is similar to *branch-on-div-result*, where, immediately after computing the division operation a branch is evaluated on the result. Additionally, *Quick-Div* requires less 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.

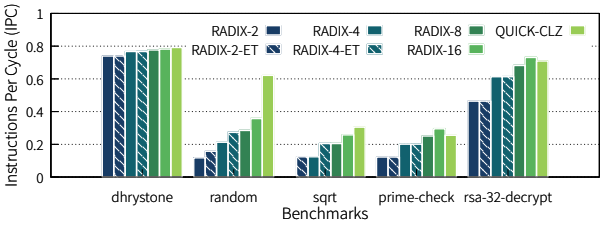

Fig. 9. Practical benchmark performance (IPC) comparison.

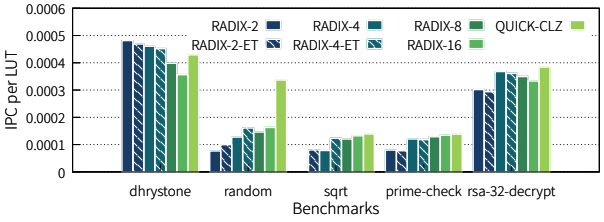

Fig. 10. Practical benchmark performance per LUT comparison.

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-div-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 higher IPC, its performance would be lower as it lowers the processor's clock frequency by more than 20% compared to our target frequency. Radix-16 is included as reference point to compare the average number of cycles per division Quick-CLZ takes for each scenario.

rsa-32-decrypt performs the RSA-32 decryption, which utilizes the remainder operator. It provide a 1.5x speedup compared to radix-2 and has the highest IPC per LUT.

In summary, through the addition of Quick-CLZ, application performance can improve by over 5x in some cases. And even in cases where only a fraction of one percent of instructions are division operations, a respectable 7% improvement in performance can be obtained.

V. Case Study Using Quick-Div

To demonstrate that our *Quick-Div* divider provides opportunities for simpler and faster algorithmic choices on soft-processors, we further investigate the square root benchmark as introduced in Section IV-D. 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 [22] that uses divisions.

The first algorithm *shift-a* [24] 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* [24] 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 less iterations than

Fig. 11. Runtime comparison between different integer square root algorithms: shift-a, shift-b, and multiply use division-avoiding algorithms

the *shift-a* algorithm. The third algorithm, which we labelled as *multiply* [25], 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 11. As all algorithms other than Newton's do not use division, only one data point is provided for them. As shown in Figure 11, 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 1.6x for Quick-CLZ over the *multiply* algorithm, and the Quick-CLZ design achieves the best performance per LUT.

VI. Conclusion and Future Work

In this paper we have shown that large performance gains are made possible by replacing the standard fixed-latency radix-2 dividers widely used in today's soft-processors with our optimized variable-latency divider called Quick-Div. The resulting performance and performance/LUT improvements can be more than 5x and 4x, respectively. Even for benchmarks with a small percent (0.24%) of division instructions, there is still a noticeable (7%) performance improvement. An in-depth characterization and comparison—in terms of performance, resource usage, and clock frequency—for the *Quick-Div* divider and radix-based dividers was conducted to provide insight into their potential use cases. Our case study illustrates that current algorithm choices may need to be reevaluated due to the performance gain obtained with our new *Quick-Div* divider. In future work, we will explore compiler and library changes to increase the use of division operations which are currently assumed to be slow.

VII. Acknowledgments

The authors would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) (NTGP 485577-15 and CWSE 470957-PDF), Xilinx Inc. and Intel for supplying resources and/or funding for this project.

REFERENCES

[1] S. K. Park and K. W. Miller, "Random Number Generators: Good Ones Are Hard to Find," *Commun. ACM*, vol. 31, no. 10, pp. 1192–1201, Oct. 1988. [Online]. Available: http://doi.acm.org/10.1145/63039.63042

[2] X. Wang, "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, Dec 2007.

[3] K. S. Hemmert and K. D. Underwood, "Floating-Point Divider Design for FPGAs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 1, pp. 115–118, Jan 2007.

[4] B. Liebig and A. Koch, "Low-latency double-precision floating-point division for FPGAs," in *2014 International Conference on Field-Programmable Technology (FPT)*, Dec 2014, pp. 107–114.

[5] X. Fang and M. Leeser, "Vendor agnostic, high performance, double precision Floating Point division for FPGAs," in *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2013, pp. 1–5.

[6] ——, "Open-source variable-precision floating-point library for major commercial fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 9, no. 3, pp. 20:1–20:17, Jul. 2016. [Online]. Available: http://doi.acm.org/10.1145/2851507

[7] G. Sutter and J. Deschamps, "High speed fixed point dividers for FPGAs," in *2009 International Conference on Field Programmable Logic and Applications*, Aug 2009, pp. 448–452.

[8] G. Sutter, G. Bioul, and J.-P. Deschamps, "Comparative Study of SRT-Dividers in FPGA," in *Field Programmable Logic and Application*, J. Becker, M. Platzner, and S. Vernalde, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 209–220.

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

[10] *MicroBlaze Processor Reference Guide*, Xilinx Inc. [Online]. Available: xilinx.com/support/documentation /sw_manuals/xilinx2016_4/ug984-vivado-microblaze-ref.pdf

[11] *Nios II Gen2 Processor Reference Guide*, Intel Corp. [Online]. Available: altera.com/en_US/pdfs/literature /hb/nios2/n2cpu-nii5v1gen2.pdf

[12] *GRLIB IP Core User's Manual*, Cobham Gaisler AB. [Online]. Available: gaisler.com/products/grlib/grip.pdf

[13] E. Matthews, Z. Aguila, and L. Shannon, "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, Apr 2018, pp. 1–8. [Online]. Available: doi.ieeecomputersociety.org/10.1109/FCCM.2018.00010

[14] F. de Dinechin and L.-S. Didier, "Table-Based Division by Small Integer Constants," in *Proceedings of the 8th International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, ser. ARC'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 53–63. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28365-9_5

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

[16] R. K. L. Trummer, "A High-performance Data-dependent Hardware Integer Divider," Master's thesis, University of Salzburg, 2005.

[17] E. Matthews and L. Shannon, "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)*, Sep. 2017, pp. 1–4.

[18] W. Snyder, "Verilator 4.008," 2018. [Online]. Available: https://www.veripool.org/ftp/verilator_doc.pdf

[19] X. Forum, "Multiple Seed Place & route," 2018. [Online]. Available: https://forums.xilinx.com/t5/Implementation/Multiple-Seed-Place-amp-route/m-p/717701#M21341

[20] R. P. Weicker, "Dhrystone: a synthetic systems programming benchmark," *Commun. ACM*, vol. 27, pp. 1013–1030, October 1984.

[21] Standard, Sep.

[22] wikipedia, "Square root." [Online]. Available: https://en.wikipedia.org/wiki/Square_root

[23] J. M. T. Palma, "The Simple C RSA-32 implementation," 2016. [Online]. Available: https://github.com/jmtorrespalma/sc-rsa

[24] J. W. Crenshaw, "Integer Square Roots," 1998. [Online]. Available: https://www.embedded.com/electronics-blogs/programmer-s-toolbox/4219659/Integer-Square-Roots

[25] M. T. I. Ross M. Fosler, "Fast Integer Square Root," 2000. [Online]. Available: http://ww1.microchip.com/downloads/en/AppNotes/91040a.pdf