Design, Implementation and Evaluation of a New Variable Latency Integer Division Scheme

Marco Angioli, Marcello Barbirotta, Abdallah Cheikh, Antonio Mastrandrea, Francesco Menichelli, Saeid Jamili, and Mauro Olivieri, *Senior Member, IEEE*

***Abstract*—Integer division is key for various applications and often represents the performance bottleneck due to its inherent mathematical properties that limit its parallelization. This paper presents a new data-dependent variable latency division algorithm derived from the classic non-performing restoring method. The proposed technique exploits the relationship between the number of leading zeros in the divisor and in the partial remainder to dynamically detect and skip those iterations that result in a simple left shift. While a similar principle has been exploited in previous works, the proposed approach outperforms existing variable latency divider schemes in average latency and power consumption. We detail the algorithm and its implementation in four variants, offering versatility for the specific application requirements. For each variant, we report the average latency evaluated with different benchmarks, and we analyze the synthesis results for both FPGA and ASIC deployment, reporting clock speed, average execution time, hardware resources, and energy consumption, compared with existing fixed and variable latency dividers.**

***Index Terms*—Variable-latency divider, integer division, high-speed arithmetic, computer arithmetic, real-time and embedded systems, low-power design.**

I. Introduction

INTEGER division is one of the fundamental operations in computer arithmetic, used in a wide range of applications such as digital signal processing [1], random number generation [2], cryptography [3], artificial intelligence [1], [4], [5], matrix factorization [6], [7], and image processing [8], [9].

Compared to addition and multiplication, division is inherently slower due to the absence of associative and commutative properties that does not allow factorization and parallelization [1], [5], [10], [11], [12], resulting in an expensive hardware implementation orsevere performance bottleneck for many applications [10], [13]. The implementation of dedicated integer division units is sometimes avoided and replaced by alternative methods, emulating integer divisions with floating-point dividers [12], yet requiring significantly large area and power consumption [14] that make them unsuitable for Field Programmable Gate Array (FPGA) implementation, or Integrated Circuit (IC) microarchitectures with limited hardware resources. In these contexts, low-hardware-cost dividers with a fixed execution latency are often used [15], [16], possibly resulting in performance limitation for those embedded applications where computational speed is critical such as automotive, video processing, and industrial control.

Variable latency arithmetic units have been studied for decades, covering addition [17], [18], [19], multiplication [20], division [5], and more complex operations [21]. Variable latency represents a valid alternative to fixed-latency when the average execution time of the target application, resulting from the average latency and the sustainable clock speed, is significantly shorter than in a fixed latency implementation, with negligible hardware overhead.

In this paper, we propose a variable latency data-dependent integer divider that significantly improves the average convergence time and power consumption, outperforming existing fixed [22] and variable latency alternatives [5], [12], [23], [24] in the literature, maintaining hardware requirements of the latter. These properties make the presented approach perfect for low-power embedded applications where the execution time is critical, making it suitable for on-the-edge machine learning and edge computing applications [25], [26], [27], [28]. The contributions of the proposed study are as follows:

- Introducing a novel variable latency integer division algorithm derived from the classic non-performing restoring technique;
- Proposing an efficient baseline hardware implementation of the algorithm, based on high-speed Count Leading Zeros (CLZ) units and a single-cycle barrel shifter with the reuse of the same register to store the remainder and the quotient;
- Detailing the hardware architectures of the algorithm in four different variants, specifically designed for targeting different application contexts;
- Comparing all the proposed hardware schemes with the reference designs in literature [12], [24], in terms of average latency, hardware cost, operating frequency and energy

Manuscript received 24 August 2023; revised 21 February 2024; accepted 30 March 2024. Date of publication 8 April 2024; date of current version 11 June 2024. Recommended for acceptance by J. Hormigo. *(Corresponding author: Marco Angioli.)*

The authors are with the Department of Information Engineering, Electronics and Telecommunications (DIET), Sapienza University of Rome, 00184 Rome, Italy (e-mail: marco.angioli@uniroma1.it; marcello.barbirotta@uniroma1.it; abdallah.cheikh@uniroma1.it; antonio.mastrandrea@uniroma1.it; francesco.menichelli@uniroma1.it; saeid.jamili@uniroma1.it; mauro.olivieri@uniroma1.it).

Digital Object Identifier 10.1109/TC.2024.3386060

consumption on FPGA and ASIC, demonstrating lower latency and energy consumption per division compared to the reference designs;
- Reporting, to the best of our knowledge, the first energy consumption analysis and ASIC synthesis of variable latency dividers in the literature;
- Showing and discussing a comprehensive evaluation of the performance on six benchmark applications.

The rest of the work is organized as follows: Section II reviews division techniques and analyzes the restoring algorithm that forms the basis of our work. Section III presents the adopted methodology and the proposed novel restoring algorithm. In Section IV, we describe the basic hardware implementation scheme for the proposed algorithm, and three alternative implementation schemes optimized for latency, clock frequency, and hardware resources, respectively. Section V analyzes the average latency as a dependence of the input operands, by Monte Carlo functional simulation. Section VI reports synthesis and implementation results of all the versions of the algorithm on FPGA using Vivado 2022.2 while Section VII reports ASIC synthesis results. In Section VIII, we test the performance of each of the proposed dividers using six real application benchmarks, comparing the overall performance with state-of-the-art variable-latency dividers and fixed-latency radix-n dividers. Finally, Section IX discusses the main outcomes of the work.

II. Background and Related Works

Integer division involves a dividend A and a divisor B, resulting in two integer values, namely the remainder W and the quotient Q, which are always less or equal to the divisor B and the dividend A, respectively, and satisfy (1).

$$A = B \cdot Q + W \tag{1}$$

Like all the division algorithms used for comparison with this work [5], [12], [22], [24], we target unsigned integer division. In the case of signed operands, the signs of the results are determined separately from the division operation, according to (2).

$$s_Q = s_A \,\mathrm{XOR}\, s_B \;; \quad s_W = s_A \tag{2}$$

Integer divisions can be implemented in hardware by different algorithms, depending on the design requirements such as computing speed and available resources [10]. The algorithms can be broadly classified into fixed and variable-latency schemes.

In a fixed-latency divider, a constant number of quotient bits is computed in each iteration of the algorithm, starting from the most significant bit (MSB). Like the classic pen-and-paper technique, the algorithm looks for the largest multiple of the divisor that can be subtracted from a partial remainder. The number of iterations of a fixed-latency divider depends on the radix used to represent the dividend and divisor, and not on their values. Due to the minimal hardware requirements, a widely used fixed-latency divider is radix-2, with one quotient bit computed at each iteration, thus taking n clock cycles to perform an n-bit division. Other typical values for the radix are 4, 8, and 16, which imply a constant division latency of $n/2$, $n/3$, and $n/4$ clock cycles, respectively. However, in resource-constrained designs, in both FPGAs and ASICs, the hardware complexity typically limits the implementation to radix-2 and 4 [10], [12].

1: **Input** $\rightarrow$
2: $Divider : A \in [0, 2^n - 1]$
3: $Divisor : D \in [0, 2^n - 1]$
4: $Partial\ Remainder : W = A$
5: $Quotient : Q = 0$
6: $count = 0$
7:
8: **Procedure** $\rightarrow$
9: **while** $count < n$
10: $W = (W << 1) - D$
11: **if** $W > 0$ **then**
12: $Q_{n-1-count} = 1$
13: **else**
14: $Q_{n-1-count} = 0$
15: $W = W + D$ # Restoration step
16: **end if**
17: $count++$
18: **end while**

Fig. 1. n-bit restoring division algorithm.

A common way to implement a radix-2 fixed-latency divider is the Restoring Division algorithm [22], [29], which we detail here as a useful basis for discussing the proposed novel schemes. The method is presented in Fig. 1. To perform a division between n-bit integer numbers, this algorithm uses two n-bits registers, Q and W, to respectively store the quotient and the partial remainder. W is initially set equal to the dividend. Then, at each iteration, a tentative remainder is computed by left-shifting the W register by one bit and subtracting the divisor. If the tentative remainder is non-negative, the new quotient bit is set to one. Otherwise, the quotient bit is set to zero, and the divisor is added back to W. When the iteration count reaches n, the division finishes with the final quotient stored in the Q register and the remainder stored in W. Notably, in hardware implementations, the registers Q and W are commonly concatenated in a single $2n$-bits register R, a technique we will refer to in the following.

The main drawback of the approach is the restoration step, which slows the execution. Several solutions have been proposed to avoid this procedure. In the Non-Restoring algorithm [22], the recovery step is omitted in exchange for additional operations and more complexity. In the Non-Performing Divider (NPD) [22], the tentative remainder is stored only when the subtraction result is non-negative. Thus, the addition in the restoring procedure is unnecessary because the R register is updated only when required. However, these two solutions do not change the total latency of the operation, which is fixed and equal to n clock cycles.

Conversely, variable-latency dividers can determine a variable number of quotient bits per iteration, so that the required cycle count for completing the division is not fixed but depends on the input data values. Notably, differently from

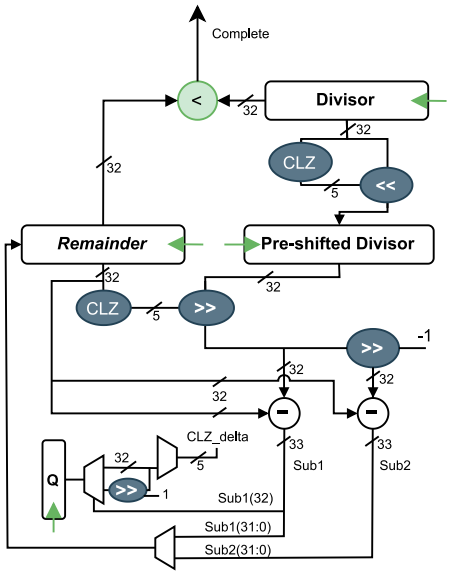

Fig. 2. Hardware implementation of the algorithm presented in [12].

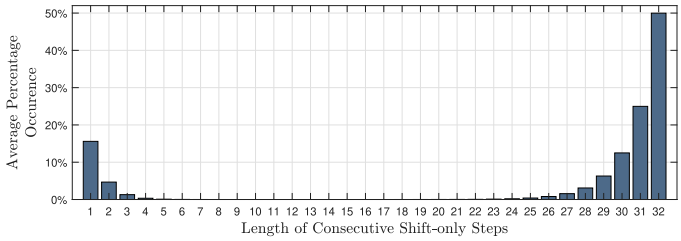

Fig. 3. Statistical analysis of "shift-only" consecutive steps in 32-bit integer divisions. For more than 98% of the divisions, the consecutive shifts are between 27 and 31, underlying the importance and impact of a dynamic shift mechanism on the NPD.

approximation-based dividers [30], these solutions produce an exact result in a variable number of clock cycles. In [5], three variable-latency integer dividers are proposed starting from the restoring division algorithm. In the first, the subtraction is skipped until the MSB of the dividend reaches the MSW of the remainder. In the second, the divisor is shifted to the left to be aligned with the dividend, then the shift is reversed, and the classical restoring division is performed. In the third, the remainder and the divisor are shifted dynamically (to the right and the left, respectively) using priority encoders. The performance of the three solutions has been evaluated in terms of speedup over the restoring division algorithm using algorithmic simulations with randomly generated operands. The work does not provide any results on real benchmarks, hardware requirements, or operating frequency.

Authors in [23] discuss and analyze the Needy Restoring division algorithm, which does not need to perform subtractions under some conditions, preventing the execution of the restoration step of the algorithm. The work does not provide performance or hardware implementation data. Furthermore, the algorithm requires an additional n-bit register and iteratively checks the value of the shifted R register, which would significantly limit the operating frequency when implemented in hardware.

The work in [24] proposes a variable latency divider based on the dynamic shift of the divisor, which skips unnecessary steps by exploiting the relationship between the remainder and the divisor itself. Priority encoders are used to compute and implement the log_2 function in hardware. The technique achieves 2.73 clock cycles on average per 32-bit integer division, assuming uniform operand distribution and an operating frequency of 90 MHz with 316 LUTs in the Virtex-7 FPGA. However, no performance data based on real application benchmarks are provided.

In [12], a similar approach is used to realize a variable-latency integer divider, denoted as Quick-Div. The algorithm was specifically designed for FPGA-based soft processors and compared in detail with fixed-latency radix-n dividers. Similar to [24], this approach performs dynamic shifting of the divisor but exploits a highly optimized hardware design depicted in Fig. 2 that uses a Count Leading Zeros (CLZs) technique and splits the shift into two steps. During the first iteration, the divisor is left-shifted by its number of leading zeros, whereas in the remaining ones, it is right-shifted by the number of leading zeros in the remainder. This approach maximizes the operating frequency and allows using a 32-bit register to store the divisor, without losing bits during the shift procedure. The division is complete when the divisor is greater than the current partial remainder. Notably, this requires an additional clock cycle to set the completion signal at the end of the division. In our solution, we avoid this by using an internal counter, which is dynamically incremented without increasing the critical path. Additionally, we use a single 64-bit register for storing the remainder and the quotient on which we perform the dynamic shift without losing bits. The Quick-Div algorithm allows an average number of clock cycles per 32-bit integer division of 1.69, assuming uniform operand distribution. However, the hardware implementation requires a fixed additional latency of two clock cycles: one for storing input signals in registers and one for performing the first shifting iteration, for an average of 3.69. The performance is tested on many application benchmarks, and the implementation results report 365 LUTs, 129 FFs and an operating frequency of 426 MHz, making this architecture the reference design for the performance comparison of the proposed work.

III. Proposed Divider

As summarized in Section II, the fixed-latency NPD [22] represents an optimized version of the Restoring divider in which the tentative remainder is stored only in the case of positive subtractions. Whenever the difference between the MSW of $R << 1$ and the divisor is negative, the previous value of R simply shifts one position to the left. As a result, several iterations will only perform a left shift of the R register, while the subtraction result is not used. Fig. 3 shows the statistical distribution of consecutive "shift-only" iterations within a 32-bit division using the NPD. In more than 98% of the cases, the

Input $\rightarrow$
  $Divider : A \in [0, 2^{32} - 1]$
  $Divisor : D \in [0, 2^{32} - 1]$
  $R_{(63\,:\,32)} = 0 \times 0000$
  $R_{(31\,:\,0)} = A$
  $count = 0$
  $division\_complete = 0$

Procedure $\rightarrow$
while division_complete == 0
    leading_D = CLZ(D) $\in [0, 31]$ # D Leading Zeros
    leading_R = CLZ(R) $\in [0, 63]$ # R Leading Zeros

    # Dynamic Shift:
    shift_amount = leading_R − leading_D − 1
    if shift_amount> 0
        if shift_amount > 31 − count
            shift_amount = 31 − count
        end if
        $R = R <<$ shift_amount
        count = count + shift_amount
    end if

    # Classic division step:
    difference = $R_{(62\,:\,31)} - D$
    if (difference< 0)
      $R = R << 1$
    else
      $R_{(63\,:\,32)}$ =difference
      $R_0 = '1'$
    end if

    if (++count==32)
      division_complete=1
    end if
end while

Fig. 4. Algorithmic description of the presented VLNPD.

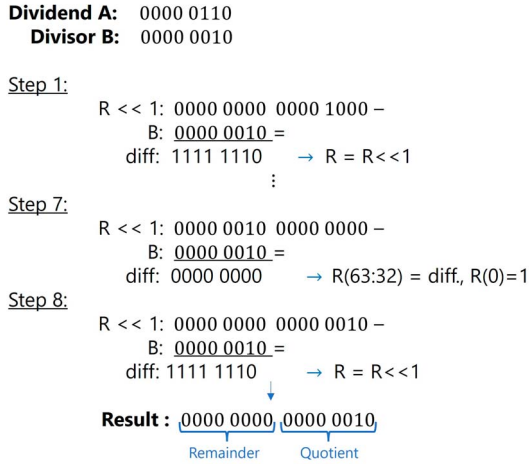

Fig. 5. Example of an 8-bit integer division with the NPD. The R register is just shifted to the left for six subsequent steps.

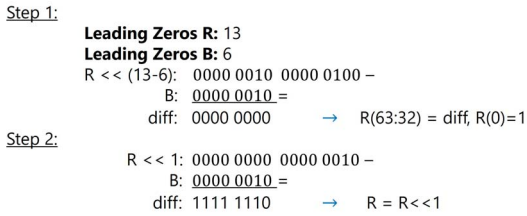

Fig. 6. Example of an 8-bit integer division with the proposed VLNPD. This approach allows skipping six steps.

division performs 27 to 32 *consecutive* steps in which the R register is just shifted to the left.

The variable latency data-dependent divider proposed in this work is based on the NPD in conjunction to exploiting the relationship between the number of leading zeros in the divisor and in the partial remainder, to dynamically detect and skip the consecutive iterations that would perform only a left shift.

Fig. 4 depicts the step-by-step overview of the presented division technique. At each iteration, the CLZ function computes the position of the most significant non-zero bit for the R register and the divisor D. The difference between these two quantities represents the number of division steps in which the subtraction result is certainly negative, and it is used to shift the R register dynamically by $shift\_amount$ bit positions. The term -1 in the calculation of $shift\_amount$ accounts for a classical division step, which is always performed after the dynamic shift. If $shift\_amount > 0$, an equivalent amount of quotient bits is evaluated in a single step, skipping the same number of "shift-only" steps. The iteration counter is updated accordingly. Note that when $shift\_amount$ results greater than the remaining division iterations, i.e. $32 - 1 - count$, the actual shift amount is limited accordingly.

Figs. 5 and 6 illustrate an example of 8-bit integer division performed with the standard NPD algorithm and the proposed technique, respectively. In the first step of the original algorithm (Fig. 5), the difference between the 8 most significant bits of $R << 1$ and the divisor D is computed and results to be negative, so R is left-shifted by one bit position. The same situation occurs for the following six iterations. The whole procedure consists of 8 iterations, with subtraction results used in only 2 of them.

In the first step of the proposed algorithm (Fig. 6), R and D have 13 and 6 leading zeros, respectively, which means that for $(13 - 6) - 1 = 6$ iterations the difference between the shifted remainder and the divisor will be negative. Thus, R is left-shifted by 6 bit positions and the classic division iteration is executed, obtaining a non-negative difference between the 8 most significant bits of $R << 1$ and D. The subtraction result is stored in the 32 most significant bit of R, and the least significant bit R_0 is set to one. In the second step, count=7 and only the classic division step is executed. The whole procedure consists of 2 iterations.

While the presented algorithm can be extended to high-precision computations, from 64 to 1024 bits, the proposed

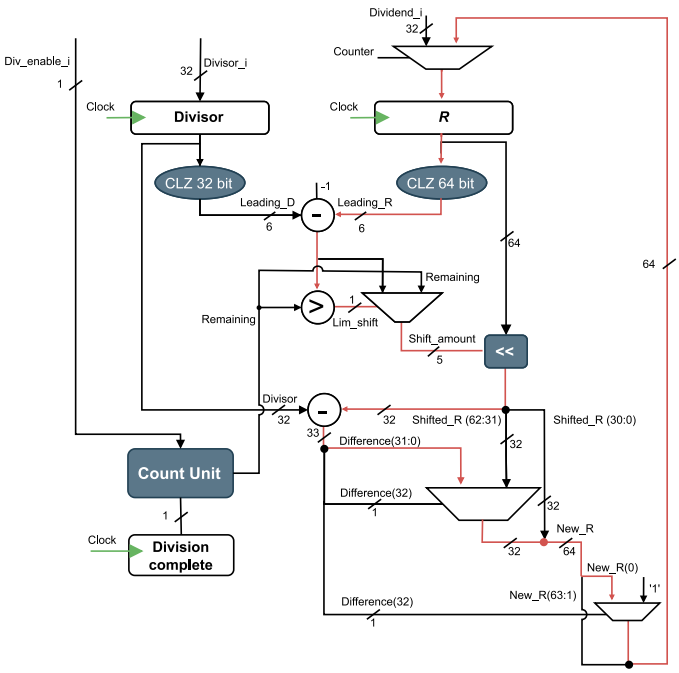

Fig. 7. Basic hardware implementation for the proposed variable latency divider (VLNPD-Std). The difference between the leading zeros is used to set the dynamic shift. D is subtracted from the output of the dynamic shifter, and the result is used to update R.

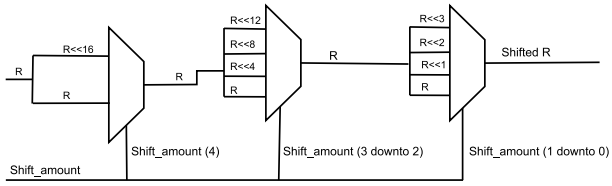

Fig. 8. Schematic of the dynamic barrel-shifter implemented by a set of three multiplexers.

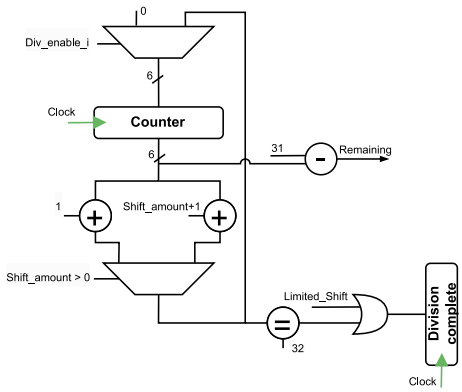

Fig. 9. Hardware scheme for the iteration count unit.

work focuses on medium or low-precision embedded applications, and we go into the details of hardware synthesis of the 32-bit implementation.

In the following, we will refer to the proposed division algorithm and its hardware implementation variants as Variable Latency Non-Performing Divider (VLNPD).

IV. Hardware Implementation

A. Baseline Version (VLNPD-Std)

The schematic diagram in Fig. 7 depicts the baseline hardware implementation (VLNPD-Std) for the proposed algorithm, with the critical path highlighted in red. The hardware unit has two 32-bit input data signals, an enable signal Div_enable, a clock signal, and a reset signal. An output signal $Division\_completed$ flags that the division has been completed, and the result (quotient and remainder) is available on the output of register R. The architecture includes a 64-bit and a 32-bit registers for R and for the divisor D, respectively, a 6-bit register in the iteration count logic, and a 1-bit output register sampling the $Division\_completed$ flag, for a total of 103 sequential logic elements, which is exactly the same amount as for the implementation of the NPD. The division starts when Div_enable is set, the input operands are stored in the registers, and the counter is set to zero. At each iteration, corresponding to one clock cycle, the value stored in the R register is updated according to Fig. 4. A 32-bit CLZ unit is used to find the number of leading zeros in the divisor D, while a 64-bit one is used for the R register. The difference between the number of leading zeros defines how many bit positions the R register should be dynamically left-shifted, according to Fig. 4. When $shift\_amount$ is greater than the remaining division steps, i.e. $32 - 1 - counter$, it is accordingly limited, and a control signal named $limited\_shift$ is set. Conversely, no dynamic shift is performed when this difference is less than or equal to zero. Finally, the classic division step is implemented by subtraction between the 32 most significant bits of the shifted R register and the divisor.

The effectiveness of the proposed division scheme relies on the availability of a fast single-cycle barrel shifter and single-cycle CLZ units, in order to avoid increasing the clock cycle time. The CLZ units were implemented according to [31], featuring a high-speed design tailored for small hardware overhead on FPGAs. The best-performing barrel shifter implementation for the chosen 32-bit operand length was found to be the multiplexing structure shown in Fig. 8, taking a 64-bit signal as the input and performing any left-shift between 1 and 31 bit positions. For 64-bit division implementations and above, a better implementation could be a multi-level architecture, as explored in [32]. The Count Unit detailed in Fig. 9 updates the iteration count, calculates the remaining steps and sets the $Division\_completed$ flag bit. If a dynamic shift is performed, the count is updated by adding $shift\_amount + 1$ to the previous value; otherwise, it is incremented by 1. When the updated count value reaches 32, or the $limited\_shift$ control signal is '1', the division ends and the $Division\_completed$ flag is set. The final remainder and the quotient are stored in the 32 most significant bits and the 32 less significant bits of the R register, respectively.

Notably, unlike the architectures presented in [12] and [24], in the proposed implementation, the output control logic uses

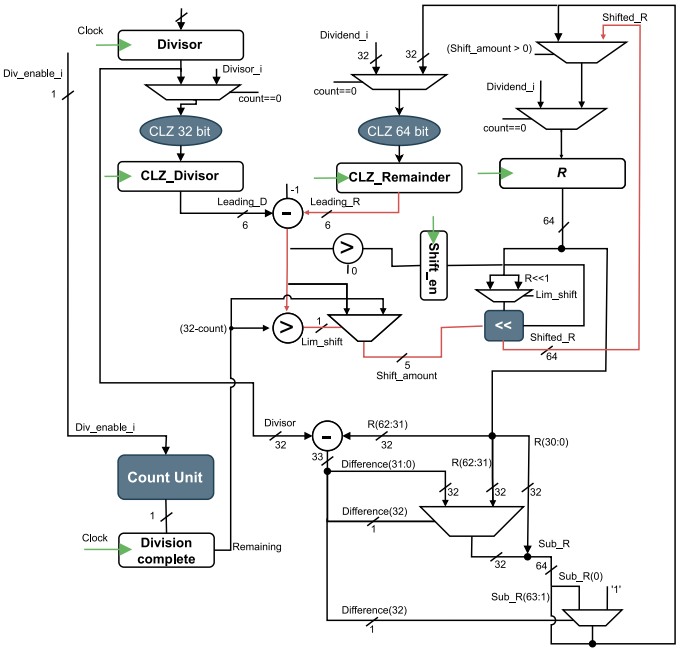

Fig. 10. Hardware architecture for the high-frequency variant (VLNPD-HF) of the proposed variable latency divider. The critical path is split as highlighted with red lines at the cost of increased average latency.

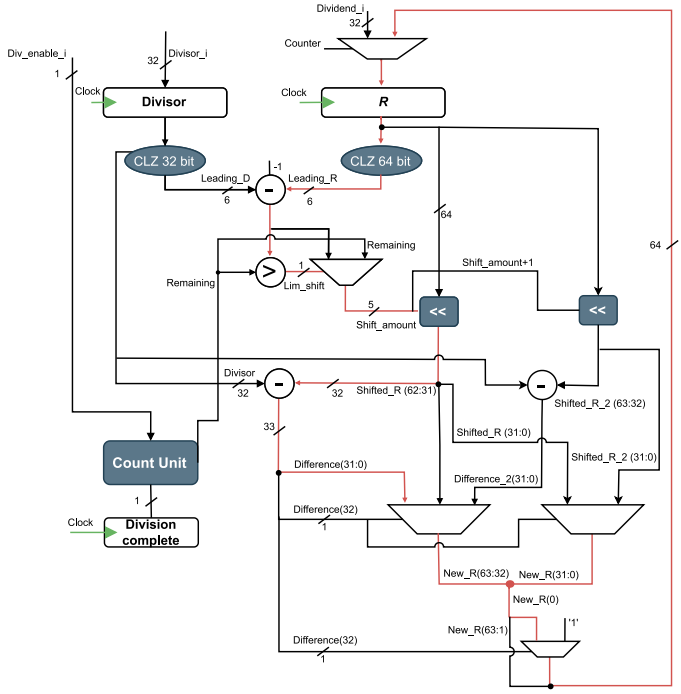

Fig. 11. Hardware architecture for the high-performance variant (VLNPD-HP). In this unit, one step is saved each time a dynamic shift is performed, and the subsequent difference is negative.

the updated value of the internal counter to detect the division completion, rather than checking the condition $remainder < divisor$. This allows to avoid an additional clock cycle to set the completion signal without increasing the critical path.

B. High-Frequency Version (VLNPD-HF)

Since the basic architecture has the CLZ units and the barrel shifter on its register-to-register critical path, a first optimization can be obtained by splitting that combinational path into two different clock cycles for a higher operating frequency, paying in terms of the average latency. In the first cycle, the classic division step and CLZ of the subtraction result are executed, while in the second clock cycle, the dynamic shift is performed, aligning the operands and making them ready again for the next step. Compared to the VLNPD-Std, this solution requires an additional clock cycle every time the operands are not aligned.

The actual hardware implementation is shown in Fig. 10. With respect to the baseline implementation, the R register has been moved after the calculation of the dynamic shift, a 6-bit register called $CLZ\_Remainder$ has been inserted to contain the 64-bit CLZ output, a 5-bit one named $CLZ\_Divisor$ contains the 32-bit CLZ output and a 1-bit register called $shift\_en$ has been introduced to disable the shifter every time the operands have been already aligned in the previous clock cycle. Whenever $shift\_amount > 0$ and $shift\_en = 0$, the dynamic shift of R is carried out, and the output of the shifter is directly written back into the R register instead of being used in the subtractor. At the next iteration, the most significant '1' of register R is already aligned with that of the divisor, and the divisor is directly subtracted from the 32 most significant bits of R. Note that if $shift\_amount > (32 - count)$, the shifter takes $R << 1$ as input to perform up to 32 iterations in one clock cycle.

Notably, this approach makes it possible to move the remainder's leading zeros count in a different clock cycle with respect to the dynamic shift, decreasing the maximum number of division steps per clock cycle from shift_amount+1 to shift_amount, splitting the previous critical path and creating the new one highlighted in Fig. 10.

C. High-Performance Version (VLNPD-HP)

In the described VLNPD-Std variable latency divider, the dynamic shift of the R register is done to align its most significant one with that of the divisor. Then, the classic division step is performed, and the sign of the result is used to choose the new partial remainder value. However, it is possible to observe that whether the result of the subtraction is negative after a dynamic shift, at the next clock cycle, the new $R << 1$ will certainly be greater than the divisor, the dynamic shift will not be performed, and the results of the subtraction will always be greater than zero. When this condition occurs, R can be directly assigned to the result of the subtraction between $R << (shift\_amount + 1)$ and the divisor, saving one clock cycle. Fig. 12 depicts this technique on an 8-bit integer division example.

From the hardware point of view, this version is depicted in Fig. 11 in which it is possible to highlight that it requires two parallel subtractions at each clock cycle and additional control logic to select which result chooses as new R register. The additional logic results in a higher area occupation and lower

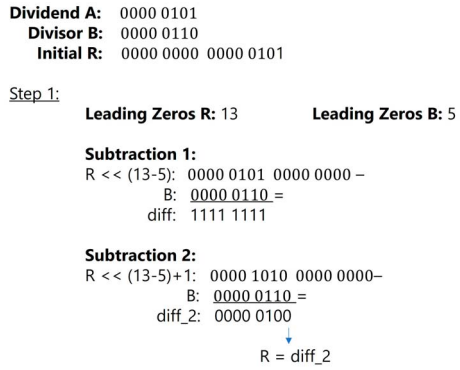

Fig. 12. Example of an 8-bit integer division in the VLNPD-HP. The HP version performs a total of 7 steps in one clock cycle.

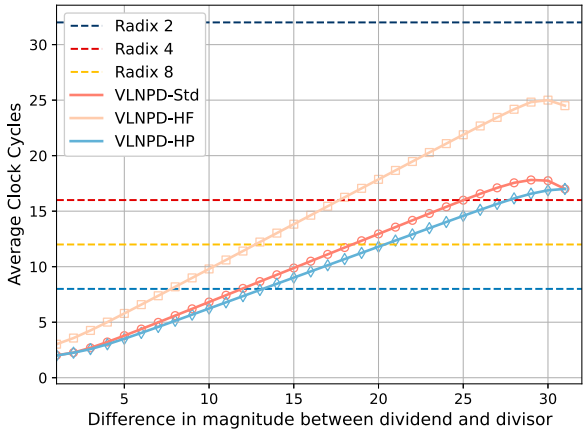

Fig. 13. Trend of the VLNPD's average clock cycle against the difference in magnitude (as powers of two) between the operands.

frequency (due to the longer critical path length) but allows for a significant reduction of the average latency per division, as will be described in Section V.

D. Limited Area Version (VLNPD-LA)

In the baseline VLNPD-Std architecture, the dynamic shifter requires the most hardware resources. This unit takes 64-bit data as input and can shift up to 32 positions to the left dynamically at each iteration of the algorithm and is represented by a 5-bit control signal (shift_amount). The implementation of this unit is optimized by a series of three multiplexers, as shown in Fig. 8, but despite that, it requires 151 LUTs. An alternative divider version with significantly lower area occupation paid in performance can be obtained by limiting the possible shifting range. Observing the statistical distribution of the average number of subsequent shift-only steps in a 32-bit division (Fig. 3), it is possible to note that this number appears to be gathered in the range and 24-31. Therefore, reducing the required area for the dynamic shifter is possible by limiting the shift in this range. The performance deviation with this solution depends heavily on the application domain, as will be discussed in Section VIII, but from a statistical point of view, this solution allows a significant reduction in the hardware resources in exchange for a small deviation in the division average latency.

V. Results

This section analyzes the performance in terms of clock cycle latency for each variant of the presented VLNPD, to verify the effectiveness of the proposed approach. Section V-A analyzes the properties of the latency as a function of the input data, showing how it changes with the amplitude ratio between the operands. In Section V-B, Montecarlo simulations for 4, 6, 8, 12, 16, and 32-bit integer divisions evaluate the average latency of the proposed dividers for uniformly distributed input values, for direct comparison with the other dividers available in the literature.

A. Latency Behavior Analysis

All the VLNPD implementation variants have data-dependent latency, meaning that the number of clock cycles required to complete a division depends on the input data values. Here we discuss some particular cases to understand the expected behaviour of the VLNPD:

- $dividend < divisor$: If the dividend is smaller than the divisor, the correct result is the quotient equals zero, and the remainder equals the dividend. However, it is unnecessary to check this condition, as the number of leading zeros in the dividend will certainly be less than in the divisor. In the first iteration, the dividend is placed in the LSW of the R register, the shift amount is always greater than or equal to 32, and the shift limiting mechanism is enabled, shifting R left by 31 positions and subtracting the divisor. This last step always returns a negative result, resulting in an additional shift of R. In this way, in a single clock cycle, the dividend is shifted from the LSW to the MSW, becoming the remainder of the division. On the other hand, the quotient in the LSW, is zero;
- $dividend = divisor$: when the dividend and divisor are equal, the difference between the leading zeros is exactly 32. As in the previous case, these divisions automatically end in one clock cycle with the difference that the result of the final subtraction will be non-negative and equal to zero. Consequently, the quotient is set to 1 and the remainder to zero, which gives the correct result;
- $dividend = 1$: the special case of the dividend equal to 1 may fall into one of the two previous cases, and both are completed in one clock cycle, as described;
- $divisor = 1$: this case is the most critical for variable latency algorithms since this division requires as many clock cycles as the number of '1's in the dividend. The latency for these cases ranges from 1 (when dividend equals 1) to 32 (when dividend equals $2^{32} - 1$). However, it is important to note that, unlike other architectures in the literature, we can solve this situation by checking whether the number of leading zeros in the divisor equals 31; still, we did not implement that to avoid adding extra hardware overhead. By adding this extra condition to the division_complete flag, divisions by one are completed in one clock cycle. The produced result is automatically the

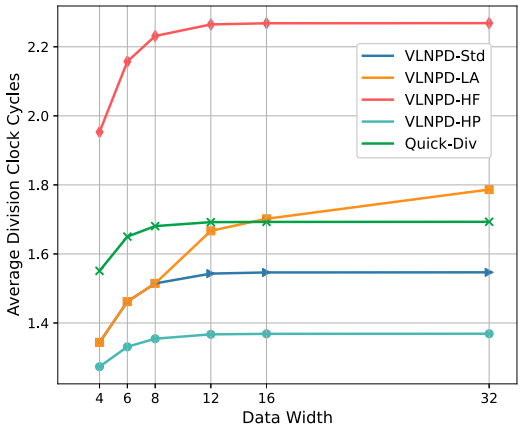

Fig. 14. Average clock cycles under varying bit width of operands.

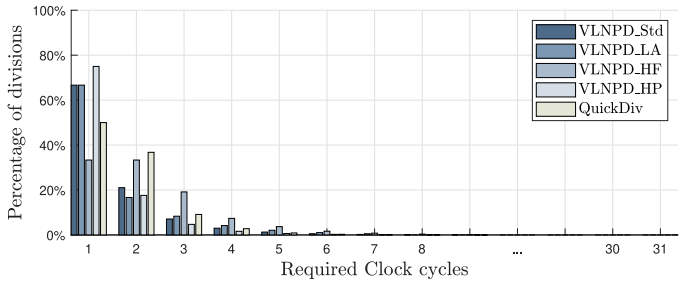

Fig. 15. Statistical distribution of division latency in 32-bit integer divisions. VLNPD-Std: 66.7% of divisions completed in 1 clock cycle, 21% in 2 cycles, 7% in 3 clock cycles. VLNPD-HP: 75% in 1 clock cycle.

correct one since, in the first iteration, the quotient is equal to the dividend and the remainder is equal to zero;

- $log_2(dividend) = log_2(divisor)$: if the base-2 logarithm of the dividend is equal to that of the divisor, the operands have the same n number of leading zeros. At the first iteration, R has $n + 32$ leading zeros, and it is, then, dynamically left-shifted by $(n + 32) - n - 1 = 31$ positions. The most significant ones are aligned in this way, and the final subtraction gives the correct result of the operation. In this case, the difference is always positive (because dividend $>$ divisor and $log_2(dividend) = log_2(divisor)$): the quotient is set to 1, and the result of the subtraction is the new remainder;
- $(dividend/divisor) \mod 2 = 0$: if the dividend is equal to the divisor multiplied by any power of two, the proposed algorithm performs integer division in two clock cycles. In the first one, the leading ones are aligned, the result of the subtraction is zero, and the quotient bit is set to one. After this operation, all subsequent division steps consist only of a left shift of the R register and can be dynamically performed in the second clock cycle. Note that this case also includes the divisions where both the dividend and the divisor are powers of two.

Fig. 13 shows how the average number of cycles required by the proposed algorithm varies as a function of the distance (in powers of 2) between the dividend and the divisor. For distances between 0 and 2^{13}, the clock cycles required by the VLNPD-Std algorithm are always less than or equal to 8, the latency expected from a radix-16 divisor. After this value, the number of cycles remains less than 12 (latency of a radix-8) up to a distance of 2^{19}. The VLNPD-Std variable latency divider is also better than a radix-4 one for distances up to 2^{25}. The number of clock cycles as a function of the distance is lower in the VLNPD-HP version and higher in the VLNPD-HF one, for the reasons outlined in Section IV.

B. Monte-Carlo Simulation Analysis

To compare and evaluate the performance between the proposed algorithm and the fixed latency solutions, we tested more

TABLE I
AVERAGE LATENCY OF INTEGER DIVIDERS FOR UNIFORMLY DISTRIBUTED NUMBERS, INCLUDING THE ONES REQUIRED BY THE HARDWARE IMPLEMENTATION

	Average Latency
NPD	33.000
VLNPD-Std	2.546
VLNPD-LA	2.786
VLNPD-HF	3.268
VLNPD-HP	2.370
Quick-Div [12]	3.693
Priority Encoders [24]	3.730

than 10^{10} dividend and divisor pairs through Monte-Carlo simulations in the case of 4, 6, 8, 12, 16 and 32 bit operand width. Fig. 14 shows the results obtained for each version. The results were obtained by simulating until the first and second decimal places of the average latency remained constant over 10 billion iterations. In all the implementation variants the increase in the average latency becomes negligible for bit-widths greater than 12, except for the case of VLNPD-LA in which the limited range of the shifter causes a steeper increase, yet remaining below 1.8 cycles. The VLNPD-Std version of the proposed divider presented in Section III requires an average number of clock cycles converging at a value of 1.55. This average latency is achieved thanks to the presented dynamic shifting method. In detail, Fig. 15 reports the clock cycle distribution required by a 32-bit integer division performed with the VLNPD-Std algorithm. Overall, 66.67% of the divisions are completed in one clock cycle. The 21% and the 7% of divisions are completed in two and three clock cycles, respectively, while the number of divisions requiring more than five clock cycles is less than 1%. The optimization technique described in Section IV-C allows a lower average number of clock cycles for the VLNPD-HP version, which converges at 1.36. In this case, thanks to the additional step that can be performed at each dynamic shift, 75% of the divisions are completed in one clock cycle. In contrast, the average latency increases in the VLNPD-HF divider, reaching a value of 2.27, in exchange for a higher operating frequency. As explained in Section IV-B, this is due to the additional cycle

TABLE II
OPERATING FREQUENCY, EXECUTION TIME, AREA AND ENERGY CONSUMPTION RESULTS OBTAINED ON XILINX VIRTEX ULTRASCALE+ VCU118 BOARD

	Operating Frequency [MHz]	Number of LUTs	Number of FFs	Average Dynamic Energy per Division [pJ]	Average Static Energy per Division [pJ]	Average Execution Time per division [ns]
NPD	800	136	103	2969.010	10.189	41.250
VLNPD-Std	300	385	103	178.270	6.290	8.489
VLNPD-LA	315	290	103	195.056	4.370	8.846
VLNPD-HF	460	463	115	326.846	7.020	7.105
VLNPD-HP	268	438	103	165.820	8.733	8.839
Quick-Div [12]	426	365	129	332.391	6.424	8.670
Priority Encoders [24]	286	446	103	261.100	13.162	13.321

at each dynamic shift, which allows the splitting of the critical path. With the VLNPD-LA version, the number of clock cycles statistically slightly increases and reaches 1.78. As mentioned before, the performance of this version strongly depends on the application, as will be detailed in Section VIII. Finally, Fig. 14 also shows the performance of Quick-Div [12], which exhibits to an average number of clock cycles of 1.69 and completes 50% of divisions in one clock cycle.

All the results reported in Fig. 14 refer to the algorithmic-level performance, assuming the operands are already available in the local input registers and without considering any additional latency overhead related to the implementation. For the architecture in [12], in fact, one more cycle is required to split the shift into two steps and increase the operating frequency. The average number of clock cycles, including the ones required by the hardware implementation, are summarized in Table I for all the VLNPD variants and the designs in [12], and [24].

Notably, in VLNPD schemes, the latency might be further reduced by avoiding one clock cycle to load the operands in internal registers, introducing a multiplexer to directly pass the dividend at the input of the shifter and the CLZ unit. In the present analysis, we did not implement this further improvement to limit the hardware overhead and to assume the same operation setup for all the compared dividers.

VI. IMPLEMENTATION ON FPGA

We synthesized and implemented the VLNPD divider, in all its variants, on the Xilinx Virtex UltraScale+ VCU118 board (XCVU9P-L2FLGA2104E) using Vivado 2022.2. We also replicated the design in [24], for which the data available in the literature are related to a Virtex-7 FPGA board. Table II reports the results of the implementations in terms of hardware resources, i.e. Look-Up Tables (LUTs) and Flip-Flops (FFs), operating frequency, average execution time per division at the maximum operating frequency, and average energy consumption per division.

The average execution time per 32-bit division is computed according to (3) and is reported in Table II.

$$Execution\_time = \frac{average\_latency}{frequency} \tag{3}$$

The average dynamic energy consumption per division and average static energy consumption per division is calculated according to (4) and (5).

$$E\_dynamic = \frac{P_{dynamic}}{frequency} * average\_latency \tag{4}$$

$$E\_static = \frac{P_{static}}{frequency} * LUT_{\%} * average\_latency \tag{5}$$

where $P_{dynamic}$ and P_{static} are the dynamic and static power data, respectively, which have been obtained by the Vivado power estimation based on the switching activity trace file extracted from the gate-level simulation of actual division operations; $LUT_{\%}$ is the percentage of the total LUTs of the device occupied by the division unit. For the design in [12], the energy results were not available in the literature and therefore they were produced by replicating the microarchitecture on the same target FPGA.

The original NPD shows an operating frequency of $800\ MHz$ and a very low area occupancy. It requires an average execution time of $41.25\ ns$ and an average dynamic energy of $2969.01\ pJ$ due to its fixed latency of 33 cycles.

The VLNPD designs significantly improve the execution time and the average dynamic energy per division, at the cost of more hardware resources. The VLNPD-Std version requires 385 LUTs and 103 FFs with an operating frequency of $300\ MHz$. Despite the lower frequency, thanks to an average of 2.55 cycles per division, it exhibits an average execution time of $8.489\ ns$, providing a speedup of $4.86\times$ over the original algorithm and 1.02 compared to [12]. The latency difference also affects the average dynamic energy, which in the case of the VLNPD-Std equals $178.270\ pJ$, 93.99% less than the original division algorithm and 46.37% less than [12]. These values are also lower than all the compared variable latency architectures, making the proposed VLNPD the most performing in terms of the average latency, average execution time, and average static and dynamic energy.

The VLNPD-HF version described in Section IV-B allows the operating frequency to be increased by 53.33% over the VLNPD-Std version, resulting in a value of $460\ MHz$. This version requires 20.25% more LUTs, 12 more FFs, and a higher average number of clock cycles (Table I). Nevertheless, this architecture has the highest frequency and the lowest execution time of all the compared variable latency dividers. In fact,

TABLE III
OPERATING FREQUENCY, EXECUTION TIME, AREA AND ENERGY CONSUMPTION RESULTS OBTAINED ON SYNOPSYS FUSION COMPILER IN GF22FDX TECHNOLOGY

	Operating Frequency [GHz]	Total Cell Area [μm^2]	Average Dynamic Energy per Division [pJ]	Average Static Energy per Division [pJ]	Average Time per division [ns]
NPD	1.66	310.97	13.464	1.171	19.800
VLNPD-Std	1.00	727.10	3.081	0.686	2.546
VLNPD-LA	1.06	619.27	2.111	0.484	2.451
VLNPD-HF	1.40	750.80	2.735	0.665	2.334
VLNPD-HP	0.98	893.30	3.375	1.008	2.407
Quick-Div [12]	1.22	722.78	3.690	0.775	3.026

the VLNPD-HF version exhibits an average execution time speedup of $5.80\times$ over the original algorithm, $1.20\times$ compared to the VLNPD-Std version and $1.22\times$ over [12].

The VLNPD-LA design described in Section IV-C reduces the number of LUTs by 24.67%, making this version the one with the smallest area occupation and the lowest static energy per operation.

Finally, although the VLNPD-HP version has the lowest operating frequency and the highest hardware resource utilization, it has the lowest average latency per division, providing significantly better performance.

VII. IMPLEMENTATION ON ASIC

We further evaluated the ASIC implementation of the proposed VLNPD dividers by synthesizing them with Synopsys Fusion Compiler on GlobalFoundries 22FDX (GF22FDX) technology. Table III contains the comparison between the proposed dividers and the existing reference dividers, again in terms of area, dynamic and static energy consumption per division operation, maximum operating frequency in typical process corner, and resulting average time per operation. Also, for the ASIC implementation, energy consumption data were obtained by the power calculator tool based on switching activity trace files extracted from gate-level simulations of real operations.

With respect to the original NPD, the four VLNPD dividers proposed in this work provide an average execution time speedup that ranges from $7.77\times$ for the VLNPD-Std version to $8.48\times$ for the VLNPD-HF version. Correspondingly, the average energy per division operation is reduced by 77.12% for the VLNPD-Std implementation up to 84.32% for the VLNPD-LA implementation.

To compare with the reference variable-latency design Quick-Div, we replicated and synthesized the design reported in [12] on the target technology. The execution time speed-up obtained by the VLNPD dividers ranges from $1.18\times$ to $1.29\times$, while the average energy per division operation is reduced by 16.50% up to 42.79%.

The VLNPD-Std version of the variable latency divider shows an operating frequency of $1\ GHz$ (40% less than the original algorithm), which is increased up to $1.40\ GHz$ (only 15.66% less than the original algorithm and 12.83% higher than the reference design Quick-div) by the VLNPD-HF architecture. The area occupation range from $619.27\ um^2$ for the VLNPD-LA design to $893.30\ um^2$ in the VLNPD-HP occupying less area than the reference design Quick-div. Overall, the VLNPD-HF Variable Latency Divider results to be the best version for ASIC implementation.

VIII. PERFORMANCE ANALYSIS ON BENCHMARK PROGRAMS

Since the actual speed of variable latency dividers is data-dependent, it is relevant to explore the performance of the proposed design as well as other reference designs in the execution of real-world computation kernels. We implemented the execution of six representative benchmarks in C++ using the target division algorithms. Also, since it is very likely that in a system-on-chip architecture, the critical path that imposes the clock frequency is not in a small sequential divider, we compared the actual performance of the dividers in a continuous range of frequencies from $100\ MHz$ up to the maximum operating frequency specific to each divider. The range of considered clock frequencies is representative of an FPGA implementation; equivalent results may be obtained for frequency ranges related to ASIC implementation.

A. Benchmark Set

Details on the six benchmark routines adopted for the analysis are the following:

RNG: pseudo-random number generation function. At each iteration, a new random number, X_{n+1}, is calculated starting from the previous one, according to (6). Like the minstd_rand functions in C++, we used $m = 2^{31} - 1$, $a = 16807$ and $c = 0$. In this benchmark, 23% of the executed instructions are divisions.

$$X_{n+1} = (aX_n + c) \bmod m \tag{6}$$

SQRT: Newton-Raphson method used to find the approximate value of the square root of a number x according to (7). The iterative method starts with an initial guess g and uses it to improve the estimation, g', until the desired accuracy is achieved. In this routine, 26% of the operations are divisions.

$$g' = 0.5 * (g + x/g) \tag{7}$$

PRIME: function that checks whether an integer input n is a prime number. If at least one of the divisions between n and

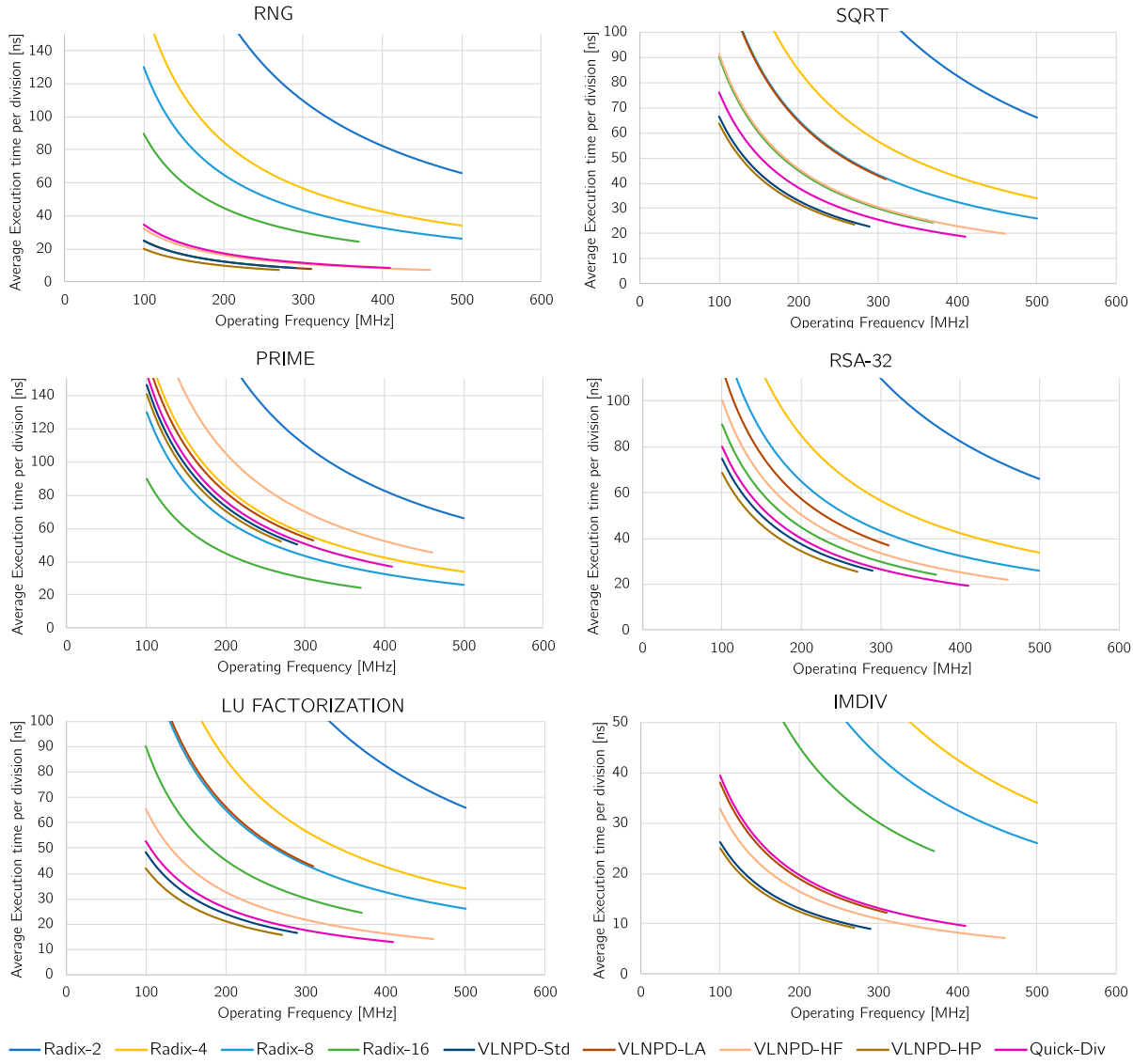

Fig. 16. Average execution time per division required in six different benchmarks as a function of the operating frequency.

every possible divisor in the range $[0, \sqrt{n}]$ has a remainder equal to zero, the number is not prime. In this benchmark, 25% of instructions are divisions. RSA: function for decrypting text files using RSA technique (8), [33]. The private key is indicated with d, and the modulus is indicated with n. The percentage of divisions in this benchmark depends on n; on average, circa 13% are divisions.

$$m = \left(c^d \bmod n\right) \tag{8}$$

LU_FACT: function for performing the LU factorization of fixed-point matrices. This technique is used for efficiently solving linear systems and inverting matrices in AI tasks such as regressions and recommendation systems. For 3×3 matrices, in this benchmark 16% of the operations are divisions.

IMDIV: pixel-wise image division used to report the fractional change or ratio for each pixel. In this benchmark, 33% of operations are divisions.

B. Results

The results of the analysis are expressed as the average division absolute execution time when running the benchmark routines, in the examined clock frequency range. Fig. 16 summarizes the obtained data. The general most relevant outcome of the analysis is that - in all the benchmarks - the proposed VLNPD dividers can obtain the same or better average execution time at a frequency lower than the other compared dividers. This is particularly relevant as it would allow running the entire system-on-chip architecture - in which the divider is to be integrated - at a lower frequency while maintaining the same division operation performance, in the view of low power consumption. Other specific details on the results are discussed below. In the RNG benchmark, the VLNPD-Std version of our variable latency divider shows an average number of clock cycles of 2.49, providing a speedup equal to $13.25\times$ over the original algorithm and $1.40\times$ over Quick-div. The low average

is due to how this benchmark works. Considering that in (6), m is equal to $2^{31}-1$, this function performs two types of division:

1) $dividend \leq m$: as described in Section V-A, the proposed algorithm always completes these divisions in one clock cycle;
2) $2^{31} < dividend < 2^{32}$: in this case, the distance (in powers of two) between the dividend and the divisor can be at most 1, and these divisions require two clock cycles (Fig. 13). Such a range for the dividend means that the first performed dynamic shift equals $32-1-1=30$. The subtraction result will certainly be negative and another shift will be required. However, this is exactly the case optimized in the VLNPD-HP version that, in this benchmark takes an average of 1 clock cycle.

To this quantity, an additional clock cycle must be added due to the hardware implementation, as described in Section V. This discussion also explains why the VLNPD-LA exhibits the same performance of the VLNPD-Std in this benchmark.

In the SQRT benchmark, the latency increases with the difference in magnitude between input operands, reaching its maximum in the last iteration when the estimation g' matches the actual number's square root. In this case, the following condition holds: $log_2(g') = \frac{1}{2} log_2(x)$, meaning that the maximum latency according to Fig. 13, will be equal to 10 clock cycles. On average, the VLNPD-Std version requires 6.65 clock cycles for this benchmark, providing a speedup equal to $4.96\times$ compared to the NPD and $1.15\times$ over Quick-div.

The PRIME benchmark is the worst for variable-latency dividers since the dividend is fixed and the divisor varies in $[2, \sqrt{n}]$. Also, the procedure is interrupted on the first division with a non-zero remainder, meaning that, in most cases, only a small part of the available range is tested. For this benchmark, the VLNPD-Std presents an average clock cycle of 14.61, providing a speedup of $2.26\times$ and $1.06\times$ over NPD and Quick-div, respectively. Note that this example shows how the workload strongly impacts the performance of the VLNPD-LA version that is almost never capable of performing the dynamic shift.

In the RSA benchmark, no particular observations are required since the latency depends on the distance of the powers of c and the modulus n. The VLNPD-Std presents an average number of clock cycles equal to 7.49, with a speedup equal to $4.41\times$ compared to NPD and $1.07\times$ over Quick-div.

In the LU_FACT routine, the elements of the lower triangular matrix are computed through the division by the corresponding diagonal elements of the upper triangular one. In this benchmark, the VLNPD-Std requires 4.82 clock cycles, with a speedup equal to $6.84\times$ relative to the NPD and $1.1\times$ over Quick-div.

Finally, in the IMDIV program, the division is usually performed between two pixels with similar values. The VLNPD-Std has an average of 2.62 cycles and a speedup of $12.60\times$ compared to NPD and $1.50\times$ over Quick-div.

IX. Conclusion

The presented variable latency data-dependent division exploits the relationship between the leading zeros in the divisor and the partial remainder to reduce the average execution time and energy consumption. These features make it suited for low-power embedded applications with high speed requirements.

We presented the algorithm, its hardware implementation and a detailed performance analysis to evaluate its effectiveness. Experimental results show that the approach achieves an average of 1.55 clock cycles per 32-bit integer division, providing a speedup of $20.65\times$ over the starting NPD algorithm, and in the range of $2.26\times$ up to $13.25\times$ when applied to real benchmarks, depending on the input data.

We illustrated four implementation versions of the presented divider, and we synthesized all of them on a Xilinx Virtex UltraScale+ VCU118 FPGA, as well as on ASIC GF22FDX technology. The FPGA VLNPD-Std version achieved the lowest execution time per division in the literature with an average of $8.489\ ns$, also reaching the lowest average dynamic energy consumption, with a reduction of 93.99% compared to the original algorithm and 46.37% over the reference design. The same properties were observed on ASIC in GF22FDX technology, with an execution time of 2.546 ns and dynamic and static energy consumption of $3.081\ pJ$ and $0.686\ pJ$, respectively. The VLNPD-HF version improved the operating frequency by 53.33% on FPGA and 40% on ASIC, leading to a further reduction in the average execution time compared to the VLNPD-Std version, with $7.105\ ns$ on FPGA and $2.334\ ns$ on ASIC despite a higher average latency in clock cycles. The best performance in terms of latency was achieved by the VLNPD-HP version with an average of 1.36 clock cycles and the 75% of divisions completed in just one iteration. In contrast, the VLNPD-LA version reduced the area occupation by 26.67%, leaving performance highly dependent on the target application.

The availability of different versions allow selecting the implementation according to the system requirements. Overall, the VLNPD-Std is the most versatile hardware implementation of the algorithm that balances operating frequency, power consumption and hardware requirements, offering a general excellent alternative to fixed-latency solutions in embedded systems. The VLNPD-HF version should be preferred when the system performance is limited by the operating frequency of the variable latency arithmetic units. The VLNPD-HP results particularly suited when clock frequency is imposed by other parts of the system, so that minimizing the clock cycle count per division is primary. Finally, the VLNPD-LA version is recommended in scenarios with very strict hardware resource constraints and the workload profiling can be used to tune the shifter range.

References

[1] U. S. Patankar, M. E. Flores, and A. Koel, "Division algorithms-from past to present chance to improve area time and complexity for digital applications," in *Proc. IEEE Latin Am. Electron Devices Conf. (LAEDC)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 1–4.
[2] 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, 1988.
[3] E. Milanov, "The RSA algorithm," RSA Laboratories, Washington, DC, USA, Tech. Rep. 2009, pp. 1–11.

[4] K. P. Sinaga and M.-S. Yang, "Unsupervised k-means clustering algorithm," *IEEE Access*, vol. 8, pp. 80716–80727, 2020.
[5] R. K. L. Trummer, "A high-performance data-dependent hardware integer divider," M.S. thesis, Inst. Comput. Sci. Syst. Anal., Paris Lodron Univ., Salzburg, Austria, 2005.
[6] R. Mittal and A. Al-Kurdi, "LU-decomposition and numerical structure for solving large sparse nonsymmetric linear systems," *Comput. Math. Appl.*, vol. 43, nos. 1–2, pp. 131–155, 2002.
[7] W. Gander, "Algorithms for the QR decomposition," *Res. Rep.*, vol. 80, no. 2, pp. 1251–1268, 1980.
[8] X. Wang, "Variable Precision Floating-Point divide and square root for efficient FPGA implementation of image and signal processing algorithms," PhD dissertation, Northeastern University, Boston, Massachusetts, 2007.
[9] D. G. Bailey, "Space efficient division on FPGAs," in *Proc. Electron. New Zealand Conf. (EnzCon'06)*, 2006, pp. 206–211.
[10] U. S. Patankar and A. Koel, "Review of basic classes of dividers based on division algorithm," *IEEE Access*, vol. 9, pp. 23035–23069, 2021.
[11] R. S. Hongal and D. Anita, "Comparative study of different division algorithms for fixed and floating point arithmetic unit for embedded applications," *Int. J. Comput. Sci. Eng.*, vol. 4, no. 9, pp. 48–54, 2016.
[12] E. Matthews, A. Lu, Z. Fang, and L. Shannon, "Rethinking integer divider design for FPGA-based soft-processors," in *Proc. IEEE 27th Annu. Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 289–297.
[13] S. F. Obermann and M. J. Flynn, "Division algorithms and implementations," *IEEE Trans. Comput.*, vol. 46, no. 8, pp. 833–854, Aug. 1997.
[14] X. Fang and M. Leeser, "Open-source variable-precision floating-point library for major commercial FPGAs," *ACM Trans. Reconfigurable Technol. Syst. (TRETS)*, vol. 9, no. 3, pp. 1–17, 2016.
[15] Advanced Micro Devices, Inc. (AMD), Microblaze Processor Reference Guide. 2021. Accessed: Apr. 14, 2024. [Online]. Available: https://www.amd.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2021_2/ug984-vivado-microblaze-ref.pdf
[16] Intel Corporation, Nios II Processor Reference Guide. 2020. Accessed: Apr. 14, 2024. [Online]. Available: https://cdrdv2-public.intel.com/666887/n2cpu-nii5v1gen2-683836-666887.pdf
[17] A. De Gloria and M. Olivieri, "Completion-detecting carry select addition," *Comput. Digital Tech. IEE Proc.*, vol. 147, pp. 93–100, Apr. 2000.
[18] M. Olivieri and A. Mastrandrea, "A general design methodology for synchronous early-completion-prediction adders in nano-CMOS DSP architectures," in *Proc. VLSI Des.*, 2013, pp. 785281:1–785281:12.
[19] S. Ghosh, D. Mohapatra, G. Karakonstantis, and K. Roy, "Voltage scalable high-speed robust hybrid arithmetic units using adaptive clocking," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 18, no. 9, pp. 1301–1309, Sep. 2010.
[20] M. Olivieri, "Design of synchronous and asynchronous variable-latency pipelined multipliers," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 9, no. 2, pp. 365–376, Apr. 2001.
[21] R. Andraka, "A survey of cordic algorithms for FPGA based computers," in *Proc. ACM/SIGDA 6th Int. Symp. Field Programmable Gate Arrays (FPGA'98)*, New York, NY, USA: ACM, 1998, pp. 191–200.
[22] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. San Francisco, CA: Morgan Kaufmann Publishers, 2004.
[23] N. Aggarwal, K. Asooja, S. S. Verma, and S. Negi, "An improvement in the restoring division algorithm (needy restoring division algorithm)," in *Proc. 2nd IEEE Int. Conf. Comput. Sci. Inf. Technol.,* Piscataway, NJ, USA: IEEE Press, 2009, pp. 246–249.
[24] F. Hassan, A. Ammar, and H. Drennen, "A 32-bit integer division algorithm based on priority encoder," in *Proc. 27th IEEE Int. Conf. Electron. Circuits Syst. (ICECS)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 1–4.
[25] M. Olivieri, A. Cheikh, G. Cerutti, A. Mastrandrea, and F. Menichelli, "Investigation on the optimal pipeline organization in RISC-V multi-threaded soft processor cores," in *Proc. New Gener. CAS (NGCAS)*, Piscataway, NJ, USA: IEEE Press, 2017, pp. 45–48.
[26] A. Cheikh, S. Sordillo, A. Mastrandrea, F. Menichelli, G. Scotti, and M. Olivieri, "Klessydra-T: Designing vector coprocessors for multi-threaded edge-computing cores," *IEEE Micro*, vol. 41, no. 2, pp. 64–71, Mar./Apr. 2021.
[27] M. Barbirotta, A. Cheikh, A. Mastrandrea, F. Menichelli, and M. Olivieri, "Design and evaluation of buffered triple modular redundancy in interleaved-multi-threading processors," *IEEE Access*, vol. 10, pp. 126074–126088, 2022.
[28] M. Angioli, M. Barbirotta, A. Mastrandrea, S. Jamili, and M. Olivieri, "Automatic hardware accelerators reconfiguration through linearUCB algorithms on a RISC-V processor," in *Proc. 18th Conf. Ph.D. Res. Microelectron. Electron. (PRIME)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 169–172.
[29] M. M. Mano, C. R. Kime, and T. Martin, *Logic and Computer Design Fundamentals*, 5th ed., Hoboken, NJ, USA: Pearson, 2015.
[30] I. Tsiokanos, L. Mukhanov, and G. Karakonstantis, "Low-power variation-aware cores based on dynamic data-dependent bitwidth truncation," in *Proc. Des., Autom. Test Europe Conf. Exhib. (DATE)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 698–703.
[31] S. Perri, F. Spagnolo, F. Frustaci, and P. Corsonello, "Design of leading zero counters on FPGAs," *IEEE Embedded Syst. Lett.*, vol. 15, no. 3, pp. 149–152, Sep. 2023.
[32] A. Ammar, H. Drennen, and F. Hassan, "High-precision priority encoder based integer division algorithm," in *Proc. IEEE Int. Midwest Symp. Circuits Syst. (MWSCAS)*, 2021, pp. 494–497.
[33] J. M. Torres-Palma, "SC-RSA: Basic and minimal implementation of RSA-32 in C." GitHub Repository, 2016. [Online]. Available: https://github.com/jmtorrespalma/sc-rsa