WEID: Worst-case Error Improvement in Approximate Dividers

Hassaan Saadat†, Haris Javaid*, Aleksandar Ignjatovic†, and Sri Parameswaran†

†The University of New South Wales, Sydney, Australia
*Xilinx Inc., Singapore
†{h.saadat, a.ignjatovic, sri.parameswaran}@unsw.edu.au, *harisj@xilinx.com

Abstract—Approximate integer dividers suffer from unreasonably high worst-case relative errors (such as 50% or 100%), which can adversely affect the application-level output. In this paper, we propose WEID, which is a novel lightweight method to improve the worst-case relative errors in approximate integer dividers. We first present an in-depth analysis to gain insights into the cause of the high worst-case relative error. Based on our insights, we propose a novel method to detect when an error occurs in an approximate divider, and modify the output to reduce the error. Further, we present the hardware realization of WEID method and demonstrate that it can be generically coupled with several state-of-the-art approximate dividers. Our results show that for 32-by-16 dividers, WEID reduces worst-case relative errors from 100% to ∼20%, while still achieving ∼80% and ∼70% reduction in delay and energy compared to an accurate array divider.

I. Introduction

Approximate arithmetic units are designed by modifying their hardware logic, so that they become simple and efficient while producing possibly erroneous outputs [1]. The goal is to achieve as much efficiency as possible with the least error. Research has shown that several modern compute-intensive applications from the domain of machine learning, deep learning, and multimedia processing are resilient to computational errors [2]. Thus, approximate arithmetic units can be used to achieve significant improvements in performance and energy-efficiency. Among the four basic arithmetic units, the dividers are notorious for having long critical path delays and high energy-consumption [3]. They are often the implementation bottleneck when their use is inevitable [4]. Therefore, designing fast and energy-efficient approximate dividers has been an active research area in the past five years [4], [5], [6].

When designing an approximate divider (or any other arithmetic unit), error behavior is regarded as one of the most important design considerations. Various error metrics can be used to quantify and capture the error behavior. In Fig. 1, we present three error metrics for several state-of-the-art 16-by-8 approximate integer dividers. We can observe that, although the *mean error* and *error bias* are reasonable (∼6% or less), *the worst-case relative error is unreasonably high for all the dividers* ($\geq$ *50%)*. Worst-case error[1] is an important error metric as it defines the upper bound of the error induced by an approximate arithmetic unit. A high worst-case error can, therefore, adversely affect the application-level output.

Motivational Example: To demonstrate the adverse effects of a high worst-case error, we use the Contrast Stretching (Histogram Stretching) application, which is a common pre-

[1]We use *worst-case relative error* and *worst-case error* interchangeably.

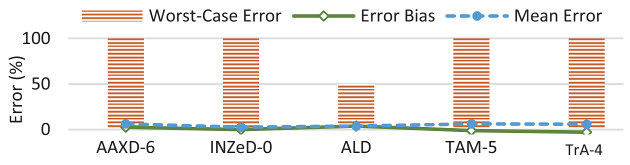


Fig. 1. Error metrics of state-of-the-art approximate dividers. The mean error and error bias are reasonable (∼ 6%) but the worst-case error is very high.

processing step in image processing applications, such as x-ray imaging [7]. It is used to increase the dynamic range, i.e., improve contrast of an image (see Section IV-D for details).

For our experiment, a low-contrast image is shown in Fig. 2 (a), while its contrast-stretched images using an accurate integer divider and an approx. integer divider INZeD-0 [4] (mean error = 2.8%) are illustrated in Fig. 2 (b) and (c), respectively. We observe that, when using an accurate divider, the output is a visually pleasant image, and its histogram (Fig. 2 (e)) is spread over the full dynamic range of 8-bit numbers. Whereas, when using the approximate divider (even with a small mean error) the output image is too dark, has poor contrast, and looks worse than the input image (Fig. 2 (f)).

The above motivational example demonstrates that a high worst-case error can have adverse effects in an application, and thus there is a need to improve the worst-case errors in state-of-the-art approximate dividers. In this paper, we aim to address this problem. Our **novel contributions** are as follows.

- We present an in-depth analysis to gain insights into the worst-case error behavior of approximate dividers.
- Using the mathematical properties of integer division, we derive mathematical conditions to detect incorrect quotients.
- Based on the insights from our analysis and the derived mathematical conditions, we propose WEID, which is a lightweight method to improve the worst-case errors in approximate integer dividers.

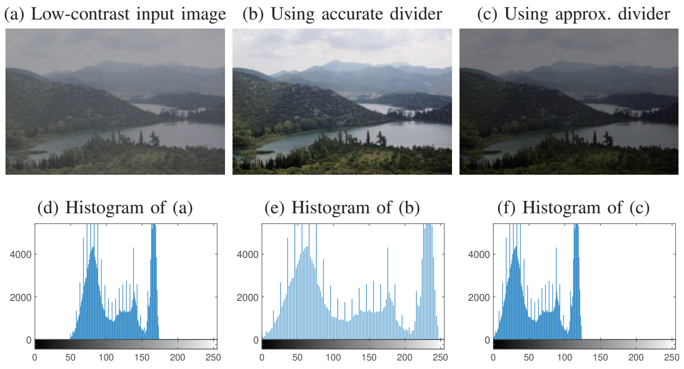


Fig. 2. Contrast Stretching (CS) example: (a) Input image; (b) CS using accurate divider; (c) CS using approx. divider; (d,e,f) Histograms of (a)–(c).

- We present a hardware design of the proposed WEID method, which is configurable for trade-offs between improvement in worst-case error and resource-overhead. We show that WEID can be generically coupled with several state-of-the-art approximate dividers; that is, it is independent of the internal architecture of the dividers.

Our experimental evaluation shows that WEID significantly improves the worst-case errors with acceptable overheads in latency (delay) and energy. Moreover, it also improves other error metrics such as mean error and error bias.

Paper Organization: Section II summarizes the recent approximate dividers. In Section III, we present an analysis for insights into the worst-case error behavior and derive the error detection conditions. Then, the proposed WEID method and its hardware design are elaborated. The experiments and results to evaluate WEID and its overheads are presented in Section IV. The paper is concluded in Section V.

II. Related Work

In recent years, approximation of integer dividers has become an active research topic. A class of approximate integer dividers use approximate adder or subtractor cells in combinational $2N$-by-N array dividers [8], [9] or higher radix dividers [10], [11]. It is reported in [5] that the resource improvements offered by this approach are insignificant.

Another class of approximate dividers perform algorithmic/architectural approximations in the divider designs. A few approximate dividers [12], [13] are look-up table based, however, the power-consumption of such dividers is high [5]. In TruncApp [6] (TrA), approximate inverse of the divisor is determined and multiplied with the truncated dividend. The TruncApp-AM [6] (TAM) is an improved version of the TrA design, in which the multiplication is also approximated. SAADI [14] computes approximate inverse of the divisor iteratively and multiplies it with the dividend.

The DAXD design [15] involves dynamically selecting bits from input operands, and uses a smaller accurate sub-divider. However, DAXD generates large errors due to overflow problem, as identified by [5]. The overflow problem is solved in the AAXD design [5], by using a larger sub-divider.

The classical approximate log based divider (ALD) [16] computes approximate log of inputs, and then performs division using log-division property. A hybrid design AXHD [17] combines an array divider and the ALD divider. The INZeD divider [4] is designed by coupling an error-correction mechanism with ALD to achieve near-zero error bias.

As we shall see in Section IV, the state-of-the-art approx. dividers suffer from very high worst-case errors when implemented in hardware. In this work, for the first time, we propose a method to improve the worst-case errors in these dividers.

III. The Proposed WEID Methodology

A. Worst-Case Error Analysis

We implemented 16-by-8 versions of several state-of-the-art approximate integer dividers: AAXD [5], AXHD [17], ALD [16], INZeD [4], TrA [6], and TAM [6]. We then performed exhaustive simulations of these 16-by-8 dividers to record accurate quotient Q, approximate quotient $\widetilde{Q}$, actual error E_a, relative error E_r, and positive, negative and absolute worst-case errors E_{wc} as follows.

$$E_a = \widetilde{Q} - Q \tag{1}$$

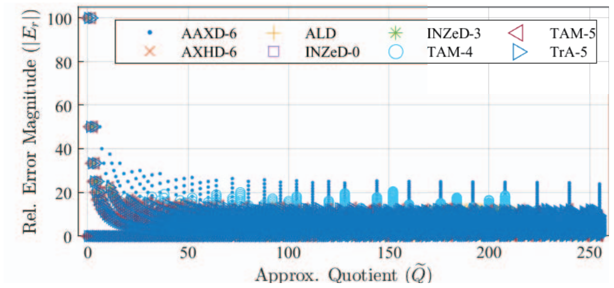


Fig. 3. Relative Error Magnitude vs Approximate Quotient. The relative error is high only when approximate quotient $\widetilde{Q}$ is small.

$$E_r(\%) = E_a/Q \times 100 \tag{2}$$

$$E_{wc}^{neg} = min(E_r), \qquad E_{wc}^{pos} = max(E_r) \tag{3}$$

$$E_{wc} = max(|E_{wc}^{neg}|,|E_{wc}^{pos}|) \tag{4}$$

Insight-1: Fig. 3 plots the magnitude of relative error $|E_r|$ against the approx. quotient ($\widetilde{Q}$) for all the dividers mentioned above. Note that the purpose of the figure is to depict the overall trend of relative error rather than showing results for the individual input combinations. From the figure, we make a key observation: *The relative error has a high magnitude only when the approximate quotient ($\widetilde{Q}$) is small.*

Insight-2: Fig. 4 illustrates the actual error E_a against the approximate quotient ($\widetilde{Q}$) zoomed in for $\widetilde{Q}\leq15$ (when the relative error is high). We make another key observation here: *the magnitude of the actual error is small when the relative error is high*. For example, $|E_a|\leq1$ when $\widetilde{Q}\leq3$, and $|E_a|\leq3$ when $\widetilde{Q}\leq15$. The reason for high relative error when actual error is low is that small values of $\widetilde{Q}$ and E_a imply that Q is also small. Consequently, the small denominator in Equation (2) may lead to high relative error.

From the two insights above, we make the following statements. If we can perform error-reduction when $\widetilde{Q}$ is small, then we can reduce the overall worst-case error (Insight-1). Furthermore, the amount of error-reduction needed for the approximate quotient in such cases will be small (Insight-2).

Note that, from Fig. 4, the actual error can be positive, negative (which means that the approximate quotient is incorrect), or zero (which means that the approximate quotient is correct). The above idea can be practically applied only if we know when the approximate quotient $\widetilde{Q}$ from a divider is incorrect; however, the correctness of an approximate quotient is not readily available since the accurate quotient Q is unknown in approximate division. In the next subsection, we derive mathematical conditions to detect when the approximate quotient is incorrect.

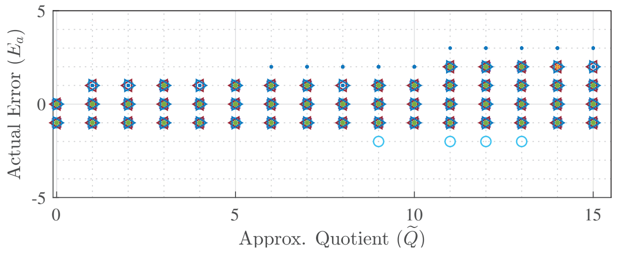


Fig. 4. Actual Error vs Approximate Quotient (same legend as Fig. 3).

B. Deriving Error Detection Conditions

Let us represent the actual quotient by Q and approximate quotient by $\widetilde{Q}$, while A denotes the dividend (numerator), B denotes the divisor (denominator), and R is the remainder. From the accurate unsigned integer division property [3],

$$A = QB + R \qquad where \qquad 0 \leq R < B \tag{5}$$

From this, we can deduce the following two inequalities,

$$A \geq QB \tag{6}$$

$$A - B < QB \tag{7}$$

The inequality (6) is deduced by substituting R=0 in Equation (5), while the inequality (7) is obtained by replacing R with B in Equation (5). These inequalities hold true when $\widetilde{Q}$ is correct (i.e., hold true for the accurate quotient). We will show that, when $\widetilde{Q}$ is incorrect (greater or smaller than Q even by a value of 1), one of the above two inequalities will not hold true.

Proposition-1: *The inequality (6) will not hold true when the approx. quotient is greater than the actual quotient ($\widetilde{Q}$>Q).*
Proof: When $\widetilde{Q}>Q$, it can be represented as $\widetilde{Q}=Q+k$ where k is an integer such that $k\geq 1$. The minimum possible error occurs when $k = 1$, so substituting this in inequality (6) yields

$$A \geq \widetilde{Q}B \implies A \geq (Q+1)B \implies (A-B) \geq QB$$

which contradicts the inequality (7).

Proposition-2: *The inequality (7) will not hold true when the approx. quotient is less than the actual quotient ($\widetilde{Q}$<Q).*
Proof: When $\widetilde{Q}<Q$, it can be represented as $\widetilde{Q}=Q-k$ where k is an integer such that $k\geq 1$. The minimum possible error occurs when $k = 1$, so substituting this in inequality (7) yields

$$A-B < \widetilde{Q}B \implies A-B < (Q-1)B \implies A < QB$$

which contradicts the inequality (6).

We use these conditions in our proposed WEID method to detect when the approximate quotient is incorrect, and needs correction/improvement.

C. The Proposed WEID Algorithm

The error improvement algorithm used in the proposed WEID method is illustrated in Algorithm 1. Recall that the relative error is high only when $\widetilde{Q}$ is small, so we introduce a parameter S, and apply the error improvement only when $\widetilde{Q} \leq S$. Varying this parameter allows for making trade-offs between error improvement and resource-overhead in WEID. The algorithm first checks if the approximate quotient $\widetilde{Q}$ is less than or equal to S. If so, then the product $\widetilde{Q}B$ is computed and the two conditions (inequalities 6 and 7) are checked. If either of the conditions is false, then $\widetilde{Q}$ is incorrect, and is modified by adding or subtracting 1 from it.

Note that the goal of the proposed WEID algorithm is not to fully fix the error (because it is an approximate divider), but to improve the worst-case relative error. Even though the actual error could be more than ±1 (Section III-A), the WEID algorithm adds only ±1. As we shall see from the results, this is enough to improve the worst-case relative error significantly. To fix the error completely, one needs to repeatedly check the error-detection conditions and keep modifying the quotient until both conditions are met. However, this could be extremely expensive and not suitable for approximate dividers.

Algorithm 1 : WEID Algorithm

Input : Approx. Quotient ($\widetilde{Q}$), Dividend (A), Divisor (B), and Parameter S.
Output : Approx. Quotient $\widetilde{Q}$ (updated).

```
$\widetilde{Q} := approx\_divider(A, B)$
if ($\widetilde{Q} \leq S$) then
    $prod := \widetilde{Q}B$
    if $prod > A$ then
        $\widetilde{Q} := \widetilde{Q} - 1$
    end if
    if $prod \leq (A - B)$ then
        $\widetilde{Q} := \widetilde{Q} + 1$
    end if
end if
```

D. Hardware Design of WEID Algorithm

The hardware design of WEID algorithm is shown in Fig. 5. Three comparators are used to implement the if-conditions in the algorithm. The product of $\widetilde{Q}B$ is computed using an N-by-$log_2(S{+}1)$-bit multiplier, where N is the size of the divisor. A subtractor computes $A{-}B$, and a $log_2(S{+}1)$-bit adder/subtractor pair computes $Q{+}1$ and $Q{-}1$. The output of comparators are used as the single-bit inputs of the adder/subtractor, so that $\widetilde{Q}$ is unaffected when both the comparator-outputs are false. At the end, two $(log_2(S{+}1){+}1)$-bit 2×1 multiplexers are used to select the appropriate output. Considering the above description of each component for a given value of S (which will be constant), we can deduce that the overall design complexity is $O(N)$. All these components can be described at the behavioral level in an HDL, which can then be synthesized to the most optimal implementation depending upon the implementation platform.

The WEID hardware can be coupled to an approximate divider as shown in Fig. 6. Most importantly, it is independent of the internal working of the divider. Furthermore, when a high worst-case error is not a concern for the given application, it can be power-gated at run-time to avoid its overhead. Nonetheless, the experimental evaluation of such application-level power-gating is beyond the scope of this paper.

E. Handling Special Cases

Accurate quotient is zero: Some approximate dividers (such as AAXD, TrA and TAM) may produce a non-zero output (+1) when the accurate quotient Q is zero. As an example, consider integer division using a 16-by-8 AAXD-6 (k=3) divider [5] when A=254 and B=255. The binary representations of these integers as 16-bit dividend and 8-bit divisor are

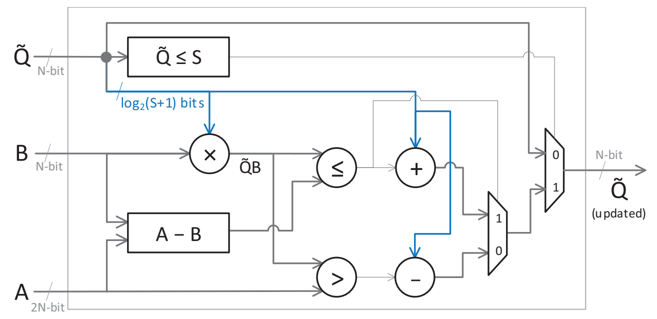


Fig. 5. The proposed WEID hardware design. Implementation of Algorithm 1.

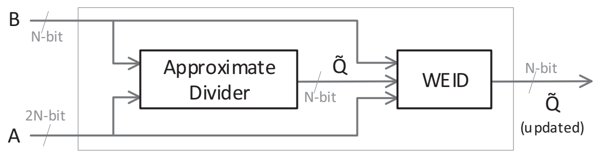


Fig. 6. Coupling WEID hardware with an approximate divider.

A=$(0000\ 0000\ \mathbf{1}111\ 1110)_2$ and B=$(\mathbf{1}111\ 1111)_2$, respectively. The positions of leading-ones are l_A=7, l_B=7. After input pruning, the truncated integers will be A_p=$(111111)_2$ and B_p=$(111)_2$, which implies A_p/B_p=$(1001)_2$. Using the formula given in [5], the approximate integer quotient is $\widetilde{Q}=(1001)_2\times 2^{l_A-l_B-k} = (1001)_2\times 2^{7-7-3} = 1$. Note that the accurate integer quotient is 0 in this case.

Such cases result in an *infinite* relative error according to Equation (2). The proposed WEID algorithm is able to automatically detect and resolve such cases by subtracting 1 from the approximate quotient.

Overflow issue: For a $2N$-by-N divider, the quotient is an N-bit integer [3]. Overflow occurs when a computed approximate quotient is higher than the maximum possible value, i.e., $2^N - 1$. Consequently, overflow is one of the reasons for high relative errors in some approximate dividers, e.g. DAXD divider [5], [15]. The AAXD divider [5] includes an overflow check, and saturates the output (i.e., sets it to $2^N - 1$) when an overflow occurs.

We observed that the TrA [6] and TAM [6] dividers also suffer from the overflow problem when implemented in hardware. Hence, we modified these dividers by adopting the overflow solution (saturation) proposed in [5]. In the previous analysis (Section III-A) and the rest of the paper, we have used the updated designs of TrA and TAM unless stated otherwise.

IV. Experiments and Results:

A. Experimental Setup

We evaluated our proposed WEID method using 32-by-16 and 16-by-8 versions of several recent approximate dividers from the literature: AAXD [5], AXHD [17], ALD [16], INZeD [4], TrA [6], and TAM [6]. These are listed in the first column of Table I.

We developed exact simulation models of the dividers and the WEID module in MATLAB. We performed the error analysis of (i) the original dividers (without WEID); and, (ii) the approximate dividers coupled with WEID for S=3, 7, and 15. The mean error (defined as the mean of absolute relative errors [4]), error bias (defined as the mean of relative errors [4]), negative worst-case error (Neg-WCE) and positive worst-case error (Pos-WCE) are presented in Table I. For the 16-by-8 dividers, we performed exhaustive simulations. Only those input combinations are valid (and thus chosen) for which no overflow occurs in the accurate $2N$-by-N integer divider, i.e., $A < 2^N B$ [3], [4], [5]. For the 32-by-16 dividers, we performed Monte-Carlo simulations of $\sim$250 million uniformly distributed, valid random inputs, since exhaustive simulations are computationally infeasible for 32-by-16 dividers. Our results show that this is enough to stimulate the worst-case error in dividers (columns 4 and 5 in Table I). For fairness, we evaluated the original and WEID-coupled versions of an approximate divider using the same set of random inputs.

In the special case of $\widetilde{Q}>0$ and $Q=0$ (see Section III-E), we set relative error to zero for the original dividers when calculating the error metrics in columns 2–5 of Table I. This special case is solved by the proposed WEID in the WEID-coupled dividers. We do not include the overflow fix in the original versions of TrA and TAM, whereas saturation is applied in the WEID-coupled versions of TrA and TAM (2nd special case, see Section III-E).

For the overhead evaluation, we implemented the above-discussed dividers in Verilog HDL. All designs are implemented as single-cycle combinational designs. The logic for overflow fix (saturation) is added in the WEID-coupled versions of TrA and TAM. For timing analysis, we placed registers at the inputs and outputs, however only combinational-logic energy and area numbers are reported. We synthesized the designs using Synopsys Design Compiler for TSMC 45nm

TABLE I
Error results of approx. dividers: for original and with the proposed WEID for various S. All errors are in percentages (%)

Approx. Dividers	Original Dividers				with WEID: S=3				with WEID: S=7				with WEID: S=15			
	Error Bias	Mean Error	**Neg. WCE**	**Pos. WCE**	Error Bias	Mean Error	**Neg. WCE**	**Pos. WCE**	Error Bias	Mean Error	**Neg. WCE**	**Pos. WCE**	Error Bias	Mean Error	**Neg. WCE**	**Pos. WCE**
32-by-16 Dividers																
AAXD-8	0.9	3.0	**-11.1**	**100.0**	0.9	3.0	**-11.1**	**33.3**	0.9	3.0	**-11.1**	**25.0**	0.9	3.0	**-11.1**	**18.8**
AAXD-10	0.4	1.5	**-5.9**	**100.0**	0.4	1.5	**-5.9**	**33.3**	0.4	1.5	**-5.9**	**14.3**	0.4	1.5	**-5.9**	**11.8**
AXHD-14	1.6	1.6	**0.0**	**50.0**	1.6	1.6	**0.0**	**25.0**	1.6	1.6	**0.0**	**22.2**	1.6	1.6	**0.0**	**16.7**
ALD	3.9	3.9	**0.0**	**50.0**	3.9	3.9	**0.0**	**25.0**	3.9	3.9	**0.0**	**22.2**	3.9	3.9	**0.0**	**16.7**
INZeD-0	0.0	2.7	**-100.0**	**50.0**	0.0	2.7	**-20.0**	**25.0**	0.0	2.7	**-12.5**	**20.0**	0.0	2.7	**-11.1**	**15.0**
INZeD-8	0.1	2.8	**-100.0**	**50.0**	0.1	2.8	**-20.0**	**25.0**	0.1	2.8	**-11.8**	**20.0**	0.1	2.8	**-11.1**	**15.0**
TAM-4	-5.6	7.0	**-100.0**	**100.0**	-3.5	5.0	**-20.8**	**33.3**	-3.5	5.0	**-20.8**	**22.2**	-3.5	5.0	**-20.8**	**16.7**
TAM-5	-1.6	6.5	**-100.0**	**100.0**	1.5	3.5	**-20.0**	**33.3**	1.5	3.5	**-12.5**	**22.2**	1.5	3.5	**-12.5**	**16.7**
TA-4	-3.1	6.2	**-100.0**	**100.0**	-1.0	4.2	**-20.0**	**33.3**	-1.0	4.2	**-16.7**	**22.2**	-1.0	4.2	**-16.7**	**16.7**
16-by-8 Dividers																
AAXD-6	2.7	6.3	**-19.5**	**100.0**	2.6	6.2	**-19.5**	**50.0**	2.4	6.1	**-19.5**	**37.5**	2.3	5.9	**-19.5**	**33.3**
AXHD-6	1.6	1.6	**0.0**	**50.0**	1.5	1.5	**0.0**	**25.0**	1.5	1.5	**0.0**	**22.2**	1.3	1.3	**0.0**	**16.7**
ALD	3.9	3.9	**0.0**	**50.0**	3.9	3.9	**0.0**	**25.0**	3.8	3.8	**0.0**	**22.2**	3.7	3.7	**0.0**	**16.7**
INZeD-0	0.0	2.8	**-100.0**	**50.0**	0.0	2.7	**-20.0**	**25.0**	0.0	2.7	**-11.8**	**20.0**	0.0	2.6	**-11.1**	**15.0**
INZeD-4	1.8	3.5	**-33.3**	**100.0**	1.7	3.5	**-20.0**	**33.3**	1.6	3.4	**-11.1**	**25.0**	1.5	3.3	**-8.6**	**21.1**
TAM-4	-5.0	6.7	**-100.0**	**100.0**	-3.1	4.8	**-20.4**	**33.3**	-3.1	4.7	**-20.4**	**22.2**	-3.1	4.6	**-20.4**	**16.7**
TAM-5	-1.1	6.4	**-100.0**	**100.0**	1.7	3.6	**-20.0**	**33.3**	1.7	3.5	**-12.0**	**22.2**	1.6	3.4	**-12.0**	**16.7**
TrA-4	-2.8	6.0	**-100.0**	**100.0**	-0.8	4.1	**-20.0**	**33.3**	-0.9	4.0	**-16.3**	**22.2**	-0.9	3.9	**-16.3**	**16.7**

standard-cell library at the maximum possible frequency for each design. All results are reported in the form of percentage reductions with respect to an accurate array divider (which has been mostly used as a reference design in previous works [4], [5], [15]). The percentage reductions are calculated as $(d_{acc}-d_{appx})/d_{acc}\times 100$, where d_{acc} and d_{appx} denote the energy/delay/area of the accurate and approximate dividers, respectively.

B. Error Results

Columns 2–5 in Table I present the error results for the original dividers, and columns 6–17 present the results for the WEID-coupled dividers for S=3, 7, and 15. The worst-case errors (positive and negative) are highlighted in bold. It can be seen that the magnitude of worst-case error for each original approximate divider is very high[2], i.e., 50% or 100%.

We observe that the WEID method reduces the worst-case errors significantly for all dividers. Specifically, when $S{=}3$, the magnitudes of WCE become $\leq$33.3%; when $S{=}7$, they become $\leq 25\%$; and when $S{=}15$, the magnitudes of WCE are close to or less than 20% for all the dividers. An exception to these improvements is the 16-by-8 version of AAXD-6, where the absolute WCE is slightly higher than the limits mentioned above. However, we can observe that AAXD-6 has one of the highest mean errors ($\sim$6%) among all the dividers, so a higher WCE is expected. Nonetheless, the improvement in the worst-case error is still evident for AAXD-6 as S increases.

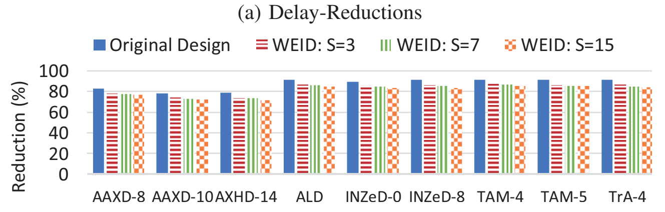
It is noteworthy that the WEID method also improves the mean error and error bias. However, this is only visible for 16-by-8 dividers. The reason is that the $\widetilde{Q}{\leq}S$ condition (when WEID is applied) occurs more frequently in 16-by-8 dividers than in their 32-by-16 counterparts, and consequently, the improvement in the average-based error metrics (mean error and error bias) is more prominent. For the TAM and TrA dividers, the improvements in mean error and error bias are also because of the applied overflow fix in the WEID-coupled versions.

C. WEID Design Overhead

Delay and Energy: The results for the delay- and energy-reductions are shown in Fig. 7 for all the implemented 32-by-16 dividers. The first bar (solid blue) in each bar-group represents the delay/energy-reduction achieved by the original divider design with respect to an accurate array divider. The following (patterned) bars depict the achievable reductions when WEID module is used with different values of the parameter S.

In Fig. 7(a), we observe that even after including WEID module, the resulting delay-reductions for all the approximate dividers are high. In other words, a substantial reduction in delay (nearly 85% for ALD, INZeD, TrA, and TAM and nearly 75% for others) is achievable using WEID with $S = 15$ which reduces the absolute worst-case errors to less than 20% in most cases. In Fig. 7(b), we see that the effect of the WEID module on energy-reductions is slightly more considerable (than on delay-reductions). Nonetheless, nearly 70% reduction is still achievable for most dividers. For AXHD-14, the effect on energy-reduction is comparatively more noticeable, however, the original AXHD-14 also has the lowest energy-reduction among all the original dividers.

[2]The author of ALD in [16] analytically demonstrated that the worst-case error is 12.5%. However, the analysis neither considers the limitations of finite precision for the fractional part of the approximate log values, nor considers the fact that the output of an integer divider must be an integer (also endorsed by recent works [4], [5]). Therefore, when practically implemented in hardware, the worst-case error for ALD is 50%. In other words, errors need to calculated considering integer division with integer quotients.



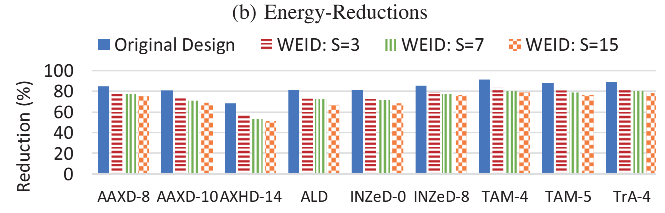
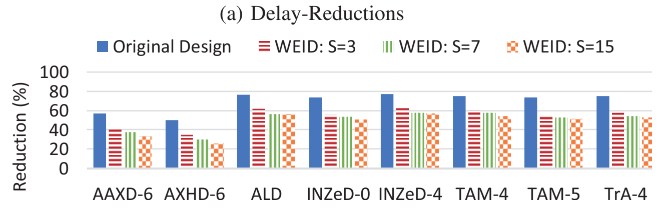


Fig. 7. Delay- and Energy-reductions for 32-by-16 dividers compared with accurate divider (having energy=1736 fJ, delay=6.3ns and area=4580μm^2).

The delay and energy results for 16-by-8 dividers are shown in Fig. 8. The achievable delay-reductions for 16-by-8 dividers are slightly less than that for 32-by-16 dividers. Nonetheless, we can still achieve more than 50% delay-reductions for most of the WEID-coupled dividers (ALD, INZeD, TAM, TrA). The impact of WIED-overhead on energy-reductions for 16-by-8 dividers is more considerable, and specifically, for the AXHD-6 divider, energy-reduction becomes negative (it consumes more energy than the accurate divider) for $S{=}7$ and $S{=}15$. However, note that even for the original AXHD-6, the energy-reduction is the lowest among all original dividers. Nonetheless, we suggest that $S{=}15$ is not suitable for 16-by-8 dividers, as far as overheads are concerned.

The effect of WEID on resource-reductions for 16-by-8 dividers is relatively more than for 32-by-16 dividers because the design complexity of an accurate divider is $O(N^2)$ and of approx. dividers is $O(NlogN)$ (in the best case). On the other hand, the design complexity of the WEID hardware for a given S is $O(N)$. Therefore, the overhead of WEID scales down slower than the dividers themselves for smaller N.



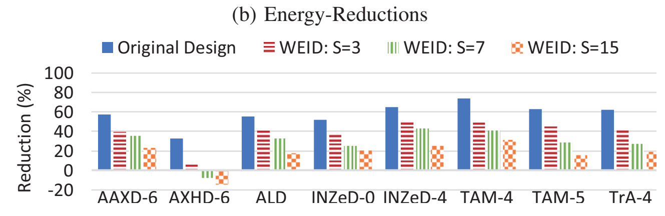


Fig. 8. Delay- and Energy-reductions for 16-by-8 dividers compared with accurate dividers (having energy=322fJ, delay=1.7ns and area=1243μm^2).

TABLE II
AREA AND AREA-DELAY PRODUCT (ADP) REDUCTIONS WITH RESPECT TO ACCURATE DIVIDER. ALL RESULTS ARE IN PERCENTAGES (%).

Approximate Dividers	Area-Reductions				ADP-Reductions			
	Orig.	S=3	S=7	S=15	Orig.	S=3	S=7	S=15
32-by-16 Dividers								
AAXD-8	69.9	53.6	51.3	47.6	94.7	89.8	89.3	87.8
AAXD-10	65.7	48.0	44.4	39.6	92.5	86.5	85.0	83.2
AXHD-14	49.8	30.1	27.5	25.3	89.3	81.8	81.0	78.8
ALD	55.6	33.5	36.3	26.8	96.3	91.4	91.1	89.0
INZeD	61.9	38.8	38.3	31.8	96.0	91.0	90.6	89.0
INZeD-8	68.0	49.7	48.1	46.3	97.2	93.0	92.5	91.3
TAM-4	78.2	57.2	50.8	49.3	98.2	94.7	93.5	92.7
TAM-5	73.1	53.3	50.3	43.1	97.6	93.6	92.8	91.7
TrA-4	72.7	54.5	53.7	47.9	97.7	93.9	93.1	91.9
16-by-8 Dividers								
AAXD-6	34.7	10.1	3.2	-12.7	71.8	47.8	39.4	24.9
AXHD-6	20.1	-15.5	-23.4	-28.9	60.1	24.3	13.5	4.5
ALD	28.7	-1.0	-8.6	-30.5	83.2	61.7	52.5	43.0
INZeD-0	28.3	2.5	-14.4	-19.4	81.1	56.8	46.7	41.0
INZeD-4	49.1	19.9	13.3	-9.5	88.3	70.1	63.1	52.8
TAM-4	60.0	16.4	10.6	-1.0	90.1	66.8	62.0	53.5
TAM-5	48.2	18.6	-0.3	-15.7	86.3	63.1	52.7	43.5
TrA-4	46.2	14.2	3.1	-12.2	86.4	63.5	55.5	47.1

Area and Area-Delay-Product: The area and area-delay-product (ADP) results are shown in Table II. We observe that WEID-coupled dividers can achieve nearly 40% area-reductions for most 32-by-16 dividers even for S=15. For 16-by-8 dividers, the area-reductions are negative for some WEID-coupled approximate dividers (area is greater than the accurate divider). Note that we suggested that S=15 is not suitable for 16-by-8 dividers. However, we have also presented the results for percentage reductions in area-delay-product in the last four columns of the table, and we can observe that even for the cases when area-reduction is negative, the reduction in ADP is substantial (other than the 16-by-8 AXHD-6).

As mentioned before, the primary aspect dividers are notorious for is their long latency (delay); the latency of dividers ($O(N^2)$) is much higher than that of multipliers ($O(logN)$) [3]. Therefore, the primary goal of approximating dividers is minimizing the delay (and energy consumption). We believe that the area overhead of WEID is acceptable, given the reduction in ADP is still significant.

D. Application Example: Contrast Stretching

Suppose we have an image $I(x, y)$ with 8-bits per pixel per channel, where x and y represent pixel coordinates. The contrast-stretched image $G(x, y)$ is computed as,

$$G(x,y) = \frac{255}{max(I) - min(I)}(I(x,y) - min(I)) \quad (8)$$

where $max(I)$ and $min(I)$ represent the highest and lowest pixel value in the image $I(x, y)$, respectively. Note that we perform the division $255/(max(I) - min(I))$ before multiplications because it limits the number of required divisions (slower operation) only to one. If the multiplication $255\times(I(x, y) - min(I))$ is performed first, then we need p divisions, where p is the number of pixels in the image.

In Section I, we showed that when using INZeD-0 for contrast stretching, the output image looks worse than the low-contrast input image due to its high worst-case error. To evaluate the performance of the WEID-coupled dividers at the application-level, we performed contrast stretching of the same input image with several WEID-coupled approximate dividers.

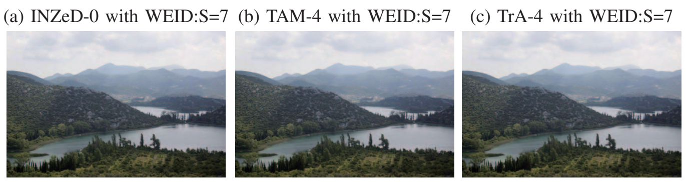


Fig. 9. Contrast Stretching using WEID-coupled approximate dividers.

The resulting images are shown in Fig. 9. We observe that the quality of the image is now as good as when using the accurate divider, described in Section I.

V. CONCLUSIONS

Most state-of-the-art approximate integer dividers exhibit unreasonably high worst-case errors. In this paper, we propose WEID: a lightweight method to improve the worst-case errors in approximate dividers. We first present an analysis to gain insight into the worst-case error behavior of approximate dividers. Then, using the insights from the analysis and the mathematical properties of integer division, we propose the WEID method to improve the worst-case errors in approximate dividers. Further, we propose a hardware design of the proposed method, and then demonstrate its efficacy by coupling it to several state-of-the-art approximate dividers. The proposed method is configurable, and can achieve different trade-offs between error improvement and resource-overhead. The results show that WEID is generically applicable to most state-of-the-art approximate dividers, and it improves the worst-case errors significantly with acceptable design overheads.

REFERENCES

[1] H. Saadat *et al.*, “Minimally biased multipliers for approximate integer and floating-point multiplication,” *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2623–2635, 2018.
[2] S. Venkataramani *et al.*, “Approximate computing and the quest for computing efficiency,” in *Proc. 52nd DAC*, 2015.
[3] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*. New York, NY, USA: Oxford University Press, Inc., 2000.
[4] H. Saadat *et al.*, “Approximate integer and floating-point dividers with near-zero error bias,” in *Proc. 56th DAC*, 2019.
[5] H. Jiang *et al.*, “Adaptive approximation in arithmetic circuits: A low-power unsigned divider design,” in *Proc. DATE*, 2018.
[6] S. Vahdat *et al.*, “TruncApp: A truncation-based approximate divider for energy efficient dsp applications,” in *Proc. DATE*, 2017.
[7] “Contrast Stretching,” http://what-when-how.com/embedded-image-processing-on-the-tms320c6000-dsp/contrast-stretching-image-processing/, 2019, [Online; accessed 09-July-2019].
[8] L. Chen *et al.*, “Design of approximate unsigned integer non-restoring divider for inexact computing,” in *Proc. 25th GLSVLSI*, 2015.
[9] L. Chen *et al.*, “On the design of approximate restoring dividers for error-tolerant applications,” *IEEE Trans. Computers*, vol. 65, no. 8, pp. 2522–2533, 2016.
[10] L. Chen *et al.*, “Design of approximate high-radix dividers by inexact binary signed-digit addition,” in *Proc. 27th GLSVLSI*, 2017.
[11] L. Chen *et al.*, “Design, evaluation and application of approximate high-radix dividers,” *IEEE Trans. Multi-Scale Computing Systems*, vol. 4, no. 3, pp. 299–312, 2018.
[12] R. Zendegani *et al.*, “SEERAD: A high speed yet energy-efficient rounding-based approximate divider,” in *Proc. DATE*, 2016.
[13] M. Vaeztourshizi *et al.*, “An energy-efficient, yet highly-accurate, approximate non-iterative divider,” in *Proc. ISLPED*, 2018.
[14] S. Behroozi *et al.*, “SAADI: A scalable accuracy approximate divider for dynamic energy-quality scaling,” in *Proc. 24th ASP-DAC*, 2019.
[15] S. Hashemi *et al.*, “A low-power dynamic divider for approximate applications,” in *Proc. 53rd DAC*, 2016.
[16] J. N. Mitchell, “Computer multiplication and division using binary logarithms,” *IRE Trans. on Electronic Computers*, vol. EC-11, no. 4, pp. 512–517, 1962.
[17] W. Liu *et al.*, “Combining restoring array and logarithmic dividers into an approximate hybrid design,” in *Proc. 25th ARITH*, 2018.