

# Software-Based Approximate Multiplication on Multiplierless CPUs Using Custom Instruction

Shalu Prathmesh Rajiv

Dept. of CSE, IIT Hyderabad, India

Email: cs23mtech12008@iith.ac.in

Rajesh Kedia

Dept. of CSE, IIT Hyderabad, India

Email: rkedia@cse.iith.ac.in

**Abstract**—Many lightweight, low-power microcontrollers contain CPU cores without hardware multipliers. When deployed for applications involving multiplication, such devices emulate multiplication using computationally expensive software techniques. While existing works used approximate hardware multipliers to reduce the complexity of multiplication, there are limited works on approximate multiplication in software. Translating ideas from approximate hardware multipliers directly to a software code does not provide considerable improvements. In this work, we study common approaches for approximate hardware multipliers and then propose a custom instruction for leading-one detection (LOD) which significantly reduces the complexity of performing approximate multiplication in software.

We implement LOD instruction in a RISC-V based core and use it to develop seven different software routines for approximate multiplication, and evaluate them on four different popular kernels. These routines are based on existing approximate hardware multipliers and three newly proposed techniques. These routines provide multiple points of trade-off between error rate and computation cycles, enabling configurable choices to the designer. When multiplying 1 million random pairs of numbers, one of the proposed techniques, RoBA-(UpDn)2, enables a very low error rate of 0.05% on average and 0.83% as maximum error; while consuming only 51% of the original CPU cycles. When deployed on two image processing applications, RoBA-(UpDn)2 can provide a signal to noise ratio (SNR) larger than 50 dB; while consuming about 70% of the original computation cycles.

## I. INTRODUCTION

Central Processing Unit (CPU) cores used in many low-power, lightweight microcontrollers like MSP430 [1] and Ix86 [2], or instruction set architectures (ISA) like RISC-V [3] provide configurations without hardware based multiplication in order to reduce the area and power consumption [4]. When executing applications such as image processing, audio processing, ML inference, etc. [4]–[6] which might require multiplication, these devices use standard software libraries based on shift-and-add (e.g., `__mulsi3` in `libgcc`). However, such software based implementation incurs high computational cost; for example, our experiment indicates that up to 189 RISC-V CPU cycles are needed to multiply two 32-bit numbers. This portrays a trade-off between area and power reduction versus more computation cycles.

One popular approach to reduce the computational complexity as well as the area/power consumption of multiplication is to allow approximations. Many works like Mitchell’s algorithm [7], [8], DRUM [9], Rounding based approximate multiplier (RoBA) [10] propose techniques to design efficient and approximate hardware multipliers. But, they are not

suitable for lightweight devices whose ISA does not support multiplication. When directly implemented as software, these approximation multiplier techniques provide limited improvements. The key bottleneck lies in identifying the position of the leading bit with a value of 1 (leading-one) in the input operands, which requires serial processing in software.

In order to address this bottleneck, we propose a custom CPU instruction named LOD (leading-one detector) for identifying the leading-one position. Using the LOD instruction, we implement a total of seven routines for approximate multiplication which are based on popular techniques for approximate hardware multipliers and three newly proposed techniques. These routines provide configurable design choices as per the application requirements. The proposed framework enables complete flexibility for developers to use exact multiplication, approximate multiplication using proposed techniques, or including newer routines in future. Three proposed techniques, RoBA-Dn2, RoBA-UpDnDn, and RoBA-(UpDn)2 consume only 30%, 44%, and 51% CPU cycles on a multiplication intensive kernel compared to using exact multiplication. The approximation leads to a small error, with average error being 0.65%, 0.19%, and 0.05% and the maximum error being 6.24%, 2.74% and 0.83% for RoBA-Dn2, RoBA-UpDnDn, and RoBA-(UpDn)2 respectively. For an image sharpening application, these three techniques provide signal to noise ratio (SNR) around 45 dB, 51 dB, and 54 dB while consuming only 41%, 70% and 75% of the original computation cycles. In terms of hardware, our design reduces area by 35.1%, critical path delay by 10.0%, and power by 17.1% compared to a single-cycle multiplier configuration.

In summary, this work makes the following contributions:

- 1) This is the **first** work towards a generic approximate multiplication in software, on a multiplierless CPU, with addition of just one custom instruction (LOD)
- 2) We implement a library of seven different approximate multiplication routines depicting trade-offs between accuracy and computation cycles
- 3) We perform a comprehensive evaluation considering accuracy, latency, and power consumption using various open-source hardware and tools; deployed on four popular kernels and two full applications

## II. RELATED WORK

Approximate multiplication, though initially proposed long back [7], still remains an active research topic [11]–[14].

Most of these works propose efficient hardware multipliers to improve the latency, area, or power consumption of multiplication with a slight error in the result. One of the popular approaches for building approximate multipliers uses logarithm, which converts multiplication into addition. It was proposed by Mitchell [7] and many variants of it have emerged to improve power, latency, or error rate [15]–[18]. RoBA [10] improves the error rate by rounding a number to its nearest up or down power of 2, by analyzing the bit next to the leading 1. DRUM [9] and LoBA [19] extract few bits starting from leading-1 bit and multiplies them, using a smaller bit-width multiplier compared to an exact multiplication. Another approach to implement approximate hardware multipliers is to use compressors which accumulate various partial products with approximation [11], [13], [20]. One common aspect in these works is that they modify the multiplier hardware and are more suitable for custom designs. In contrast, we enable approximate multiplication in software itself, which allows configurable error rate in lightweight microcontrollers.

For certain microcontrollers lacking hardware multipliers, there are proposals [1] to reduce the complexity of multiplication by storing the number in a modified representation or using approaches like Horner’s method; both of which assume that the multiplier values are apriori known. Voronenko et al. [21] and Aksoy et al. [22] improve multiplication with large constants on multiplierless devices assuming one of the operands is known upfront. However, our work is applicable more widely as we do not require any prior knowledge of the input numbers. Verma et al. [14] implement an approximate multiplier inside a RISC-V core for error-tolerant applications, but they cannot perform exact multiplication operations, if needed. Our proposed software-based approach allows for full flexibility in choosing different multiplication algorithms as per the application requirements.

In summary, to the best of our knowledge, there are no prior works on approximate multiplication of generic operands in software, useful for multiplierless microcontrollers.

### III. BACKGROUND

#### A. Multiplication in Multiplierless Devices

While hardware multipliers are usually present in CPU implementations, many low-end microcontrollers like MSP430 [1], Ibex [2] allow multiplierless configurations to save area and power consumption [4] (data shown in Fig. 1). Also, RISC-V (a popular open-source ISA) does not include multiplication in its base instruction set (RV32I), enabling a simple implementation wherever needed.

In such systems, multiplication of two arbitrary numbers is realized using library routines like `__mulsi3` (Listing 1), which uses shift-and-add. The CPU cycles consumed by such implementations depend upon the multiplicand value (position of leading-1 in the input `a`) and can be high for large numbers.

#### B. Techniques for Approximate Multiplication

Most techniques for approximate multiplication propose a hardware multiplier which calculates an approximate prod-

```

1 unsigned int __mulsi3
2   (unsigned int a, unsigned int b) {
3     unsigned int r = 0; // Store the result
4     while (a) {
5       if (a & 1)
6         r += b; // Add b if a's LSB is 1
7       a >>= 1; // Shift-right to check next bit
8       b <<= 1; // Shift-left
9     }
10    return r;
11  }

```

Listing 1. `__mulsi3` routine for multiplication used by gcc

uct but reduces the area, latency, or power consumption of the multiplier. Logarithmic technique (e.g., Mitchell’s approach [7]) is used in many such approximate multipliers [8], [9], [15], [17], [23]. Other approximation techniques [11], [20] use approximate compressors or adders and are closely tied to hardware implementation; thus, not discussed further.

The logarithmic approach uses the fact that the logarithm of product of two numbers is equal to the sum of their individual logarithms. Therefore, Mitchell’s proposal calculates the logarithm of multiplicand and multiplier, adds them, and then calculates the anti-logarithm of the sum. In binary number system, the position of the leading-1 in the input number provides an approximate logarithm value. Various works using logarithm-based approximate multiplication implement a leading-1 detector (LOD) in the hardware [8], [10], [15], [23].

Another approximate multiplier named RoBA (**R**ounding-**B**ased **A**pproximate multiplier) [10] rounds the operands to their nearest powers of two, converting a multiplication into shifts. RoBA re-writes the multiplication of  $A$  and  $B$  as:

$$A \times B = (A_r - A) \times (B_r - B) + A_r \times B + A \times B_r - A_r \times B_r,$$

where  $A_r$  and  $B_r$  are  $A$  and  $B$  rounded to nearest powers of two. If we ignore the first term in the right hand side, the resulting approximate product of  $A$  and  $B$  can be written as:

$$A \times B \approx A_r \times B + A \times B_r - A_r \times B_r$$

Instead of ignoring, we could approximate the error term  $(A_r - A) \times (B_r - B)$  through another iteration of RoBA algorithm to reduce the error, but increased computations. We explore such trade-offs in this work.

Similar to Mitchell’s algorithm, the calculation of  $A_r$  or  $B_r$  will require LOD implementation. While it is easy to implement LOD directly on the hardware using either a shift-and-count approach [7] or a priority encoder [10], LOD implementation in software requires executing multiple shift and compare instructions and is not efficient. Therefore, we propose adding LOD as a custom instruction to enable efficient approximate multiplication in software. Including such custom instructions is nicely supported by the RISC-V ecosystem. While CLZ (count leading zeros) instruction in RISC-V Bit manipulation ISA extension [3], [24] can help to identify leading-1 position, CLZ is not a part of the RISC-V base ISA and will require an additional subtraction for LOD. Even the minimal variant of RISC-V Bit manipulation extension increases the area by 14.3% on our experimental platform.

```

1 LOD rd, rs ; rd <- Position of leading-1 in rs

```

Listing 2. Syntax and semantics of the proposed LOD instruction

```

1 // Multiplies a and b, returns approximate product
2 uint32_t RoBA-Dn1(uint32_t a, uint32_t b){
3     uint32_t apos, bpos;
4     uint32_t b_apos, a_bpos;
5     uint32_t res;
6     if (a==0 || b==0) return 0;
7     // Returns MSB bit position of a in apos
8     asm volatile ("LOD %0, %1\n"
9         : "=r" (apos)
10        : "r" (a));
11    // Returns MSB bit position of b in bpos
12    asm volatile ("LOD %0, %1\n"
13        : "=r" (bpos)
14        : "r" (b));
15    b_apos = b << apos;
16    a_bpos = a << bpos;
17    res = b_apos + a_bpos - ((1<<apos)<<bpos);
18    return res;
19 }

```

Listing 3. Routine for RoBA-Dn1 approximate multiplication (unsigned) [10]

#### IV. PROPOSED APPROACH FOR APPROXIMATE MULTIPLICATION

As discussed earlier, hardware approximate multipliers implement LOD which is not efficient when realized in software. Therefore, we include LOD as a custom CPU instruction to enable efficient approximate multiplication in software. We implement the LOD instruction as an R-format (register format) instruction in an open-source RISC-V core (Ibex [2]). The instruction takes a destination register (`rd`) and a source register (`rs`) as its operands, with the syntax and semantics shown in Listing 2. The internal hardware implementation of LOD uses a priority encoder [10].

With the CPU hardware supporting the proposed instruction, we develop various routines in C language for approximate multiplication, using inline assembly to instantiate the LOD instruction<sup>1</sup>. We develop routines for approximate multiplication techniques proposed in popular prior works, as well as some newer techniques to expand the trade-off space. We implement the following techniques for approximate multiplication:

1) *Exact*: It uses the default multiplication technique from the compiler. We enable the highest optimization level (O3) so that the best possible compiler optimizations for multiplication are invoked and considered as a baseline for comparing the performance of various approximate techniques.

2) *Mitchell*: It is based on the Mitchell’s logarithmic algorithm [7], implemented on similar lines as presented in [15].

3) *DRUM6*: This approach is based on DRUM [9], configured with 6 bits as relevant<sup>2</sup>. Our implementation identifies these relevant bits and multiplies them using the exact

<sup>1</sup>We also implemented the routines entirely using assembly code, but the performance obtained from C code with O3 (and O2) optimization levels was similar or better and hence the final routines are based on C language, with LOD being used through inline assembly.

<sup>2</sup>We experimented using higher and lower number of relevant bits and identified that they do not add any new Pareto point (see Fig. 7). Hence, due to lack of space, we present DRUM with only one configuration (6 bits).

approach. This technique reduces the complexity of exact multiplication to a maximum of 6 bits.

The original RoBA algorithm [10] (Section III) considers rounding operands to their nearest power of two, in down- or up-direction, to convert multiplication into shift operations. We implement and evaluate multiple variants of the algorithm to depict the trade-off between accuracy and computation cycles.

4) *RoBA-Dn1*: RoBA-Dn1 implements RoBA with operands being always rounded **down** to their nearest power of two. The C code for the implementation is shown in Listing 3, which also illustrates an example usage of LOD instruction for various proposed routines.

5) *RoBA-UpDn*: This approach follows the original proposal from the RoBA algorithm [10], by identifying the nearest power of two considering both up and down directions. The operands are rounded up if their bit next to the leading one bit is set; else, the operands are rounded down. This executes extra instructions to check additional bits, but reduces the error compared to rounding only in the down direction.

The following three techniques are **newly proposed in this work** by modifying the existing RoBA approach.

6) *RoBA-Dn2*: This is an enhancement to the RoBA algorithm by applying an additional iteration of RoBA to the error term, so that we do not entirely ignore the error term. In this second iteration, the difference between the exact values and the rounded (down) values of the operands is computed, and multiplied using RoBA-Dn1, and the computed error term is added to the previously computed result (Section III). While this additional iteration increases computation cycles, it significantly reduces the error.

7) *RoBA-(UpDn)2*: We extend RoBA-UpDn by using RoBA-UpDn for an additional iteration to calculate the error term, as was done in RoBA-Dn2. Rounding to nearest value in either up or down direction could result in the error term being either negative or positive. Therefore, RoBA-(UpDn)2 involves additional computations for an additional iteration as well as identifying the sign of the error term. These additional computations help to drastically reduce the error rate.

8) *RoBA-UpDnDn*: This is a hybrid approach involving rounding up or down for the first iteration, but only rounding down for the second iteration. Rounding down is simpler compared to rounding up or down, and hence this approach uses fewer CPU cycles than RoBA-(UpDn)2.

*Unsigned and Signed Operands*: We implement two variants of routines for each of the approaches, one for unsigned numbers and the other for signed numbers. For the signed numbers, we multiply the absolute values of the numbers and then assign the sign as per whether the sign of the operands were same or different. This requires additional instructions compared to handling unsigned numbers.

## V. EXPERIMENTS AND RESULTS

### A. Experimental Setup

1) *Hardware Platform*: We use a lightweight RISC-V 32-bit core named Ibex [2], originally called Zero-riscy (a

part of the PULP platform [25]). It is an in-order single-issue core, with full support for the RISC-V 32-bit base integer instruction set (RV32I) and optionally, the RISC-V M-extension (RV32IM). The System Verilog implementation of Ibex supports different variants of hardware multipliers, providing area-latency trade-offs. We extend the Ibex code to support the LOD instruction, using a priority encoder to detect the leading-one. We use this extended core inside an Ibex demo system SoC supporting a Verilator simulation model which can execute a binary file from the SRAM memory.

2) *Compiler*: All proposed multiplication routines and the application code are compiled using the RISC-V GCC toolchain (v10.2.0) for RV32I. We enable the highest compiler optimization level (O3) so that the best and commonly adopted compiler optimizations are automatically included in the baseline (exact multiplication).

3) *Kernels and Applications*: We implement four kernels, that are commonly used in many different applications, to evaluate the proposed approximate multiplication routines. The first kernel (*MULT*) multiplies 10K pair of numbers in sequence. These numbers are randomly generated in a way that ensures – (i) the product does not exceed 32 bits, and (ii) cover various cases of multiplier being either much larger, much smaller, or being of similar order as the multiplicand. This kernel being multiplication-intensive helps us establish some form of upper bound on the improvements.

The second kernel (*MATMUL*) implements matrix multiplication to multiply two  $32 \times 32$  matrices. The third kernel (*IDCONV*) implements a 1-D convolution for a signal of length 1024 and with a filter of size 11. The fourth kernel (*2DCONV*) performs 2-D convolution of an image of size  $64 \times 64$  with a filter of  $3 \times 3$ . The input values are randomly generated but do not cause overflow in 32 bits, to mimic the situation with an actual hardware multiplier.

We also use two popular image processing applications, with real images and actual filter values to evaluate the proposed work. The first application performs image smoothing using a  $7 \times 7$  Gaussian filter, with 8-bit fixed point values. The second application sharpens an image using a  $5 \times 5$  filter. We execute these applications for 5 different standard images (Lena, Airplane, Couple, Sailboat, and Baboon) of  $50 \times 50$  size and measure signal to noise ratio (SNR) as the error metric. The noise is calculated as per the deviation from using an exact multiplication.

4) *Measuring Cycle Counts and Error Rate*: The cycle count is measured using internal performance counters available within the Ibex core. We use `pcount_enable(1)` and `pcount_enable(0)` to enable and disable the counters. To measure the error rate with approximation, we compare the output from using approximate multiplication routines versus using the exact multiplication, for the same set of inputs.

5) *Measuring Hardware Design Metrics*: We use open-source EDA tools (Yosys [26] and OpenSTA [27]) to compare area and clock period of the Ibex demo SoC for various existing multiplier configurations and also with the proposed LOD instruction. Since the included tool flow does not directly

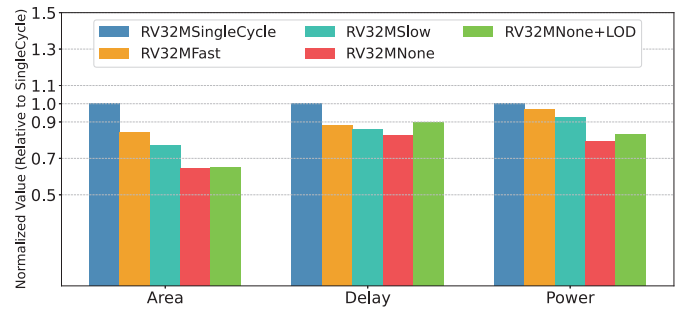


Fig. 1. Hardware metrics for various multiplier configurations

provide the power consumption of the SoC, we used Cadence Genus synthesis tool with 65 nm library (fast cells) to obtain the power consumption of various multiplier configurations.

## B. Results

We present quantitative results in three different parts – (i) Hardware overheads (area, delay, and power) of proposed custom instruction, (ii) Performance benefits (CPU cycles) and error rate for four kernels, (iii) Performance benefits (CPU cycles) and SNR for the two image processing applications.

1) *Hardware Metrics*: We compare the area, delay, and power consumption for the following multiplier configurations.

- RV32MSingleCycle: A single cycle multiplier
- RV32MFast: A 3-cycle multiplier
- RV32MSlow: A bit-serial multiplier
- RV32MNone: No hardware multiplier
- RV32MNone+LOD: No hardware multiplier, but proposed LOD instruction implemented

As shown in Fig. 1, Ibex core without the M-extension support consumes 35.1%, 22.9%, and 15.9% lesser area compared to RV32MSingleCycle, RV32MFast, and RV32MSlow configurations, respectively. We also observe that including the proposed LOD instruction has a negligible effect (0.35%) on the area of the Ibex core.

When comparing the critical path delay (which defines the clock cycle), we observe that the proposed LOD instruction increases the delay by 8% compared to RV32MNone. Despite the increase, the critical path delay of our proposed design is 10% lower than the RV32MSingleCycle configuration and very close to that of the RV32MFast or RV32MSlow configurations. In terms of power consumption, the LOD instruction increases the power consumption of RV32MNone configuration by 4.5%. Still, the proposed design consumes 17.1%, 14.3%, and 10.3% lesser power than RV32MSingleCycle, RV32MFast, and RV32MSlow configurations respectively.

In summary, the hardware overheads of including LOD instruction are significantly lower than supporting M-extension.

2) *Cycle Count and Error Rate for Kernels*: In this section, the cycle counts correspond to the RV32MNone+LOD configuration. For each of the proposed multiplication routines, Fig. 2 shows the number of clock cycles required for four common kernels (Section V-A3). Compared to baseline (exact multiplication using standard libraries), the proposed approximate multiplication routines consume significantly lesser

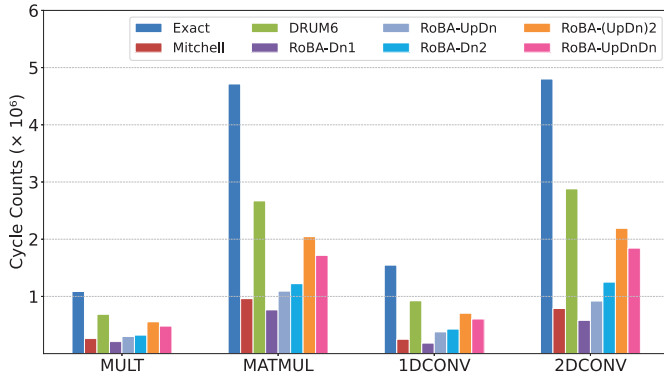


Fig. 2. Cycle count for various multiplication routines across kernels

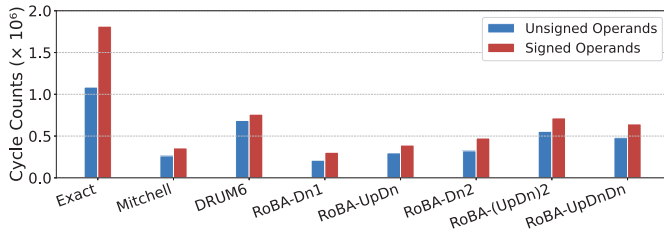


Fig. 3. Cycle count for proposed routines for unsigned and signed operands number of clock cycles. Different approaches for approximate multiplication consume varying amount of cycles, with RoBA-Dn1 consuming the lowest and DRUM6 consuming the largest. DRUM6 [9] needs to perform an exact multiplication of 6 bits. This reduced bit-width multiplication makes DRUM6 better than other approaches for hardware implementation [9], [10]. However, when it comes to software implementation, DRUM6 consumes more cycles than other approaches as it performs the exact multiplication of 6 bits using shift-and-add technique. RoBA-Dn1 results in reducing computation cycles to  $0.19\times$ ,  $0.16\times$ ,  $0.11\times$ , and  $0.12\times$  of the baseline for MULT, MATMUL, 1DCONV, and 2DCONV kernels respectively. We also observe more branch instructions in shift-and-add based multiplication compared to the almost linear code used in our approximate multiplication routines. This acts as a secondary factor in further reducing the CPU cycles.

Fig. 3 shows the computation cycles for routines corresponding to the unsigned and signed operands for the MULT kernel. For the approximate routines, the signed operands incur a slightly larger number of cycles than the unsigned operands due to additional computations to handle the sign. However, for the baseline approach, the signed operands consume very large number of cycles as more bits towards the MSB are ‘1’.

*Effect of Operand Values:* Since the standard library implementation of the multiplication involves shift-and-add until a becomes 0 (Listing 1), the cycle count of exact multiplication depends upon the actual value of the operands. In order to better quantify the cycle count reduction, we generate random numbers by restricting their values within certain number of bits and then evaluate the MULT kernel.

As shown in Fig. 4, when operand values are small (all bits are 0 except four lower bits), the cycle count for the baseline is similar to many approximate multiplication routines, and

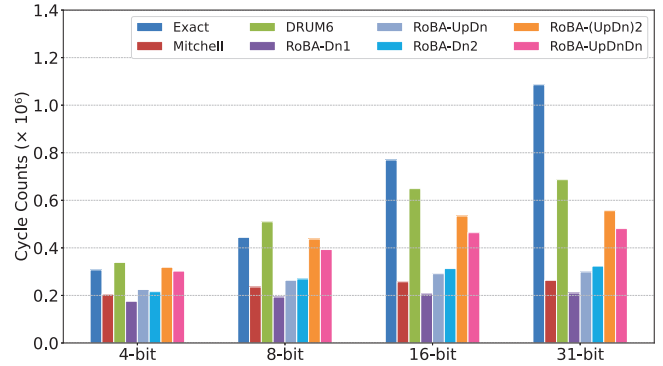


Fig. 4. Cycle count for various multiplication routines, across input bit-widths

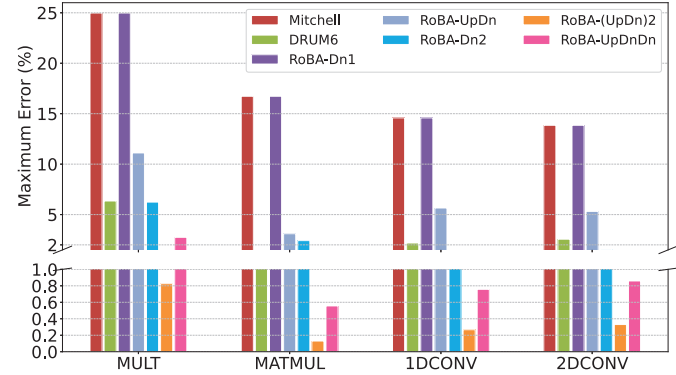


Fig. 5. Maximum error with proposed routines, across kernels

slightly lesser than DRUM6 and RoBA-(UpDn)2. This is because the exact multiplication involve only 4 iterations of shift-and-add for such small numbers. However, as we increase the value of the operands, Fig. 4 shows that the computation cycles needed for the exact approach increases proportionally with operand width, whereas the increase is smaller for the approximate techniques. Therefore, the approximate multiplication techniques will provide much larger benefits for even larger 64-bit or 128-bit RISC-V cores.

*Error Rate:* We compare the error rate (percentage error) of different approximation techniques for four different kernels. We use 1 million random pairs of inputs to calculate the error rate for the MULT kernel. For other kernels, we use the same configurations from Section V-A3. Fig. 5 shows the maximum error rate in the output for each approximate multiplication approach across different kernels. Mitchell and RoBA-Dn1 perform similar and have the highest error rate compared to other approximate routines. RoBA-(UpDn)2 has the lowest error rate due to additional iteration for the error term. However, it consumes much larger CPU cycles compared to Mitchell or RoBA-Dn1 (Fig. 2). The maximum error rate for MATMUL, 1DCONV, and 2DCONV kernels is lower than MULT as a few intermediate products cancel each other’s errors during accumulation.

Fig. 6 shows the fraction of output values with an error rate less than certain threshold, for unsigned operands. As an example, for RoBA-(UpDn)2, 98.9% of outputs have less than 0.5% error and all outputs have less than 1% error. For RoBA-(UpDn), 82.6% of outputs have less than 5% error. We clearly

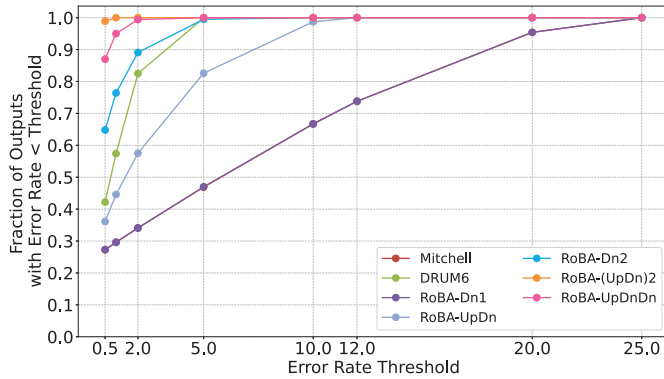


Fig. 6. Percentages of outputs with error rate less than a threshold for the MULT kernel (for unsigned operands)

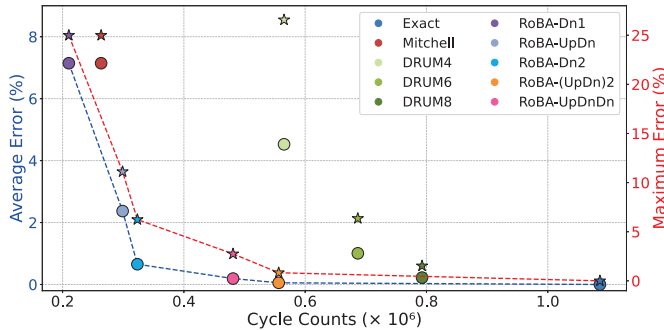


Fig. 7. Error rate versus cycle counts for different approximation routines for the MULT kernel

observe that the error rate reduces significantly with the second iteration of RoBA algorithm – RoBA-Dn1 versus RoBA-Dn2, and RoBA-UpDn versus RoBA-(UpDn)2 or RoBA-UpDnDn. The error rates for these proposed techniques are significantly better than existing works. Similar behavior is also observed for signed operands (omitted due to limited space).

For the MULT kernel, Fig. 7 shows the trade-off between error rate (average and maximum) versus computation cycles for different approximate multiplication techniques (we also include DRUM4 and DRUM8 variants of DRUM [9]). Most of the proposed techniques form *Pareto* points, providing a unique advantage either in error rate or computation cycles. The error rate for RoBA-(UpDn)2 is very close to 0, but with half the number of cycles compared to exact multiplication. Other techniques also provide similar trade-off between error rate and computation cycles. DRUM does not show up as a Pareto point which indicates that very efficient hardware techniques may not be as efficient for software implementation.

3) *Cycle Count and Error Metrics for Image Processing Applications:* Table I presents the cycle count and error metrics (average values for five different images) for image smoothening and image sharpening applications, as described in Section V-A. Similar to previous results, we observe multiple points with trade-off between cycle count and signal to noise ratio (SNR), defined as  $SNR = 10 * \log \frac{\text{signal power}}{\text{noise power}}$ . As an example, RoBA-UpDnDn provides an SNR of 53.05 dB and 51.16 dB respectively for the two applications, while requiring only 75% and 70% of the original computation cycles. Other

TABLE I  
SNR AND CYCLE COUNT FOR TWO IMAGE PROCESSING APPLICATIONS WHEN DIFFERENT APPROXIMATE ROUTINES ARE USED

Approximation Technique	Image smoothening			Image sharpening		
	SNR (dB)	Cycle count ( $\times 10^6$ )	Norm. cycle count	SNR (dB)	Cycle count ( $\times 10^6$ )	Norm. cycle count
Exact	$\infty$	8.86	1.00	$\infty$	4.75	1.00
Mitchell	22.69	3.04	0.34	23.33	1.85	0.39
DRUM6	47.09	10.11	1.14	43.88	5.10	1.07
RoBA-Dn1	22.69	2.13	0.24	23.33	1.40	0.29
RoBA-UpDn	41.42	2.91	0.33	38.20	1.92	0.40
RoBA-Dn2	<b>43.64</b>	3.79	<b>0.43</b>	<b>45.52</b>	1.95	<b>0.41</b>
RoBA-(UpDn)2	57.62	7.52	0.85	54.08	3.54	0.75
RoBA-UpDnDn	<b>53.05</b>	6.66	<b>0.75</b>	<b>51.16</b>	3.33	<b>0.70</b>

approximate multiplication techniques can provide higher or lower SNR, while consuming more or lesser cycles respectively, enabling configurable options to the developer.

Despite having only 8-bit values in the images, we observed a considerable cycle count reduction. For larger bit-width images, we expect even larger speedup over exact multiplication.

## VI. CONCLUSION

To perform approximate multiplication on multiplierless CPU cores, we proposed a new instruction named LOD which identifies the position of the leading ‘1’ in the source operands, which is needed for many approximate multiplication techniques. We developed seven different software routines for approximate multiplication using LOD, which present trade-off between calculation error versus the computation cycles. A RISC-V based Ibex core was used to implement LOD instruction and evaluate the proposed techniques. Three proposed techniques named RoBA-Dn2, RoBA-UpDnDn, and RoBA-(UpDn)2 consume only 30%, 44%, and 51% of original computation cycles with an average error of only 0.65%, 0.19%, and 0.05% respectively. These three techniques support an SNR of 45 dB, 51 dB, and 54 dB for an image sharpening application while consuming only 41%, 70% and 75% of the computation cycles with exact multiplication. We achieve 35.1%, 10.0%, and 17.1% reduction in area, delay, and power respectively compared to using a single-cycle multiplier.

In terms of limitations, the proposed work does not handle overflow which we plan to incorporate in future. This work also cannot generalize to SIMD/vector operations, but such features are anyways not available in lightweight devices.

By enabling trade-off between error rate and computation cycles through different multiplication algorithms, the proposed work opens up research problems in programming languages and compilers in terms of annotating the required error tolerance (or inferring it by the compiler) and accordingly, identify an appropriate multiplication routine for use. We would also like to explore approximate division in software, using LOD.

## ACKNOWLEDGEMENT

Vishal Vijay Devadiga (IIT Hyderabad) performed initial error analysis of RoBA algorithm with multiple iterations. We thank MeitY for enabling access to Cadence tools through Chips to Startup (C2S) scheme under the Grant G576.

## REFERENCES

- [1] Texas Instruments, "Efficient multiplication and division using MSP430™ MCUs," in *Texas Instruments Application Report*, 2018. [Online]. Available: <https://www.ti.com/lit/an/slaa329a/slaa329a.pdf>
- [2] "Ibex RISC-V Core," accessed: 2025-01-31. [Online]. Available: <https://github.com/lowRISC/ibex>
- [3] "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA," accessed: 2025-03-31. [Online]. Available: <https://riscv.org/specifications/ratified/>
- [4] B. H. Spies, M. C. Michelotti, L. L. de Oliveira, and E. A. Carara, "Evaluating multiplier-less CNNs in RISC-V architecture," in *IEEE 16th Latin America Symposium on Circuits and Systems (LASCAS)*, vol. 1, 2025, pp. 1–5.
- [5] H. You, X. Chen, Y. Zhang, C. Li, S. Li, Z. Liu, Z. Wang, and Y. Lin, "ShiftAddNet: a hardware-inspired deep network," in *Proceedings of the 34th International Conference on Neural Information Processing Systems (NeurIPS)*, 2020.
- [6] H. K. Fung and K. H. Wong, "A multiplier-less implementation of the canny edge detector on FPGA and microcontroller," *International Journal of Computer Theory and Engineering*, vol. 9, no. 3, pp. 172–178, 2017.
- [7] J. N. Mitchell, "Computer multiplication and division using binary logarithms," *IRE Transactions on Electronic Computers*, vol. EC-11, no. 4, pp. 512–517, 1962.
- [8] Z. Babić, A. Avramović, and P. Bulić, "An iterative logarithmic multiplier," *Microprocessors and Microsystems*, vol. 35, no. 1, pp. 23–33, 2011. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0141933110000438>
- [9] S. Hashemi, R. I. Bahar, and S. Reda, "DRUM: A dynamic range unbiased multiplier for approximate applications," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015, pp. 418–425.
- [10] R. Zendegani, M. Kamal, M. Bahadori, A. Afzali-Kusha, and M. Pedram, "RoBA multiplier: A rounding-based approximate multiplier for high-speed yet energy-efficient digital signal processing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 2, pp. 393–401, 2017.
- [11] L. Sayadi, S. Timarchi, and A. Sheikh-Akbari, "Two efficient approximate unsigned multipliers by developing new configuration for approximate 4:2 compressors," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70, no. 4, pp. 1649–1659, 2023.
- [12] B. Rashidi, "Efficient and low-cost approximate multipliers for image processing applications," *Integration*, vol. 94, p. 102084, 2024. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167926023001268>
- [13] S. Hwang, K.-W. Kwon, and Y. Kim, "Design of a hardware-efficient approximate 4-2 compressor for multiplications in image processing," *IEEE Embedded Systems Letters*, pp. 1–1, 2025.
- [14] A. Verma, P. Sharma, and B. P. Das, "RISC-V core with approximate multiplier for error-tolerant applications," in *2022 25th Euromicro Conference on Digital System Design (DSD)*, 2022, pp. 239–246.
- [15] Z. Babić, A. Avramović, and P. Bulić, "An iterative Mitchell's algorithm based multiplier," in *2008 IEEE International Symposium on Signal Processing and Information Technology*, 2008, pp. 303–308.
- [16] M. S. Kim, A. A. D. Barrio, L. T. Oliveira, R. Hermida, and N. Bagherzadeh, "Efficient Mitchell's approximate log multipliers for convolutional neural networks," *IEEE Transactions on Computers*, vol. 68, no. 5, pp. 660–675, 2019.
- [17] M. S. Kim, A. A. Del Barrio, R. Hermida, and N. Bagherzadeh, "Low-power implementation of Mitchell's approximate logarithmic multiplication for convolutional neural networks," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2018, pp. 617–622.
- [18] V. Mahalingam and N. Ranganathan, "Improving accuracy in Mitchell's logarithmic multiplication using operand decomposition," *IEEE Transactions on Computers*, vol. 55, no. 12, p. 1523–1535, Dec. 2006. [Online]. Available: <https://doi.org/10.1109/TC.2006.198>
- [19] B. Garg, S. K. Patel, and S. Dutt, "LoBA: A leading one bit based imprecise multiplier for efficient image processing," *Journal of Electronic Testing*, vol. 36, no. 3, pp. 429–437, 2020.
- [20] T. Kong and S. Li, "Design and analysis of approximate 4–2 compressors for high-accuracy multipliers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 10, pp. 1771–1781, 2021.
- [21] Y. Voronenko and M. Püschel, "Multiplierless multiple constant multiplication," *ACM Trans. Algorithms*, vol. 3, no. 2, p. 11–es, May 2007. [Online]. Available: <https://doi.org/10.1145/1240233.1240234>
- [22] L. Aksoy, D. B. Roy, M. Imran, and S. Pagliarini, "Multiplierless design of high-speed very large constant multiplications," in *Proceedings of the 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024, p. 957–962. [Online]. Available: <https://doi.org/10.1109/ASP-DAC58780.2024.10473954>
- [23] S. E. Ahmed, S. Kadam, and M. B. Srinivas, "An iterative logarithmic multiplier with improved precision," in *2016 IEEE 23rd Symposium on Computer Arithmetic (ARITH)*, 2016, pp. 104–111.
- [24] B. Koppelman, P. Adelt, W. Mueller, and C. Scheytt, "RISC-V extensions for bit manipulation instructions," in *International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2019, pp. 41–48.
- [25] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flaman, F. K. Gürkaynak, and L. Benini, "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.
- [26] "Yosys Open SYnthesis Suite," accessed: 2025-01-31. [Online]. Available: <https://github.com/YosysHQ/yosys>
- [27] "Static Timing Analyzer," accessed: 2025-01-31. [Online]. Available: <https://github.com/The-OpenROAD-Project/OpenSTA>