Trailing-Ones Anticipation for Reducing the Latency of the Rounding Incrementer in FP FMA Units

Toru Koizumi[*†], Ryota Shioya[*‡], Takuya Yamauchi[*], Tomoya Adachi[*], Ken Namura[*], and Jun Makino[*]

[*] Preferred Networks, Inc.
[†] Nagoya Institute of Technology
[‡] The University of Tokyo

Introduction

- Floating-point fused multiply-add (FMA) operations are fundamental in many fields
  - such as graphics, scientific computing, machine learning.
- One of the most complex arithmetic units in a processor

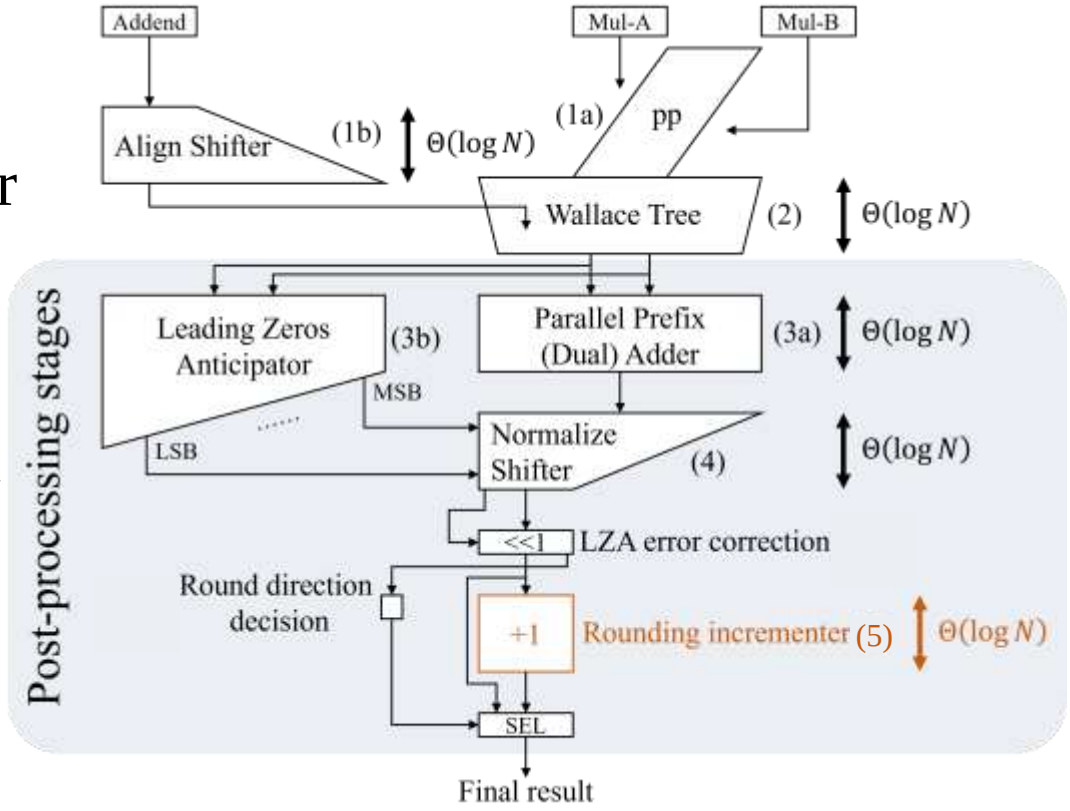
- Many research efforts have been made to design an efficient FMA.
  - Some recent improvements:
    - Leading zeros anticipation that can also handle a denormal number output [Lutz, 2017]
    - Early decision of rounding direction [Lutz, 2017]
    - Booth multiplier that can also handle denormal number inputs [Sohn, 2023]
    - Tracking an all-ones bit in addition to the sticky bit to remove the dual adder [Sohn, 2023]

Conventional FMA architecture

- Conventional FMA architecture: five $\Theta(\log N)$-depth components
  - (1) Partial product generator and align shifter
  - (2) Wallace (or another adder) tree
  - (3) Parallel prefix (dual) adder and leading zeros anticipator
  - (4) Normalize shifter
  - (5) Rounding incrementer

  } ~60% of total latency [1]

- We focused on (5) for the possibility of reducing latency.



[1] S. Srinivasan, K. Bhudiya, R. Ramanarayanan, P. S. Babu, T. Jacob, S. K. Mathew, R. Krishnamurthy, and V. Errgauntla, "Split-path fused floating point multiply accumulate (FPMAC)," in *2013 IEEE 21st Symposium on Computer Arithmetic (ARITH)*, pp. 17–24 (2013).

Idea: using trailing ones

- If we have the *trailing ones* of before-rounding mantissa, the increment is done by a simple XOR operation.
  - Example: 0b110011 [M: before-rounding mantissa]
    0b000011 [T: trailing ones]
    0b000111 [T << 1 | 1]
    0b110100 [M ^ (T << 1 | 1)] this is the M+1
  - The trailing ones indicate the carry-propagation region.

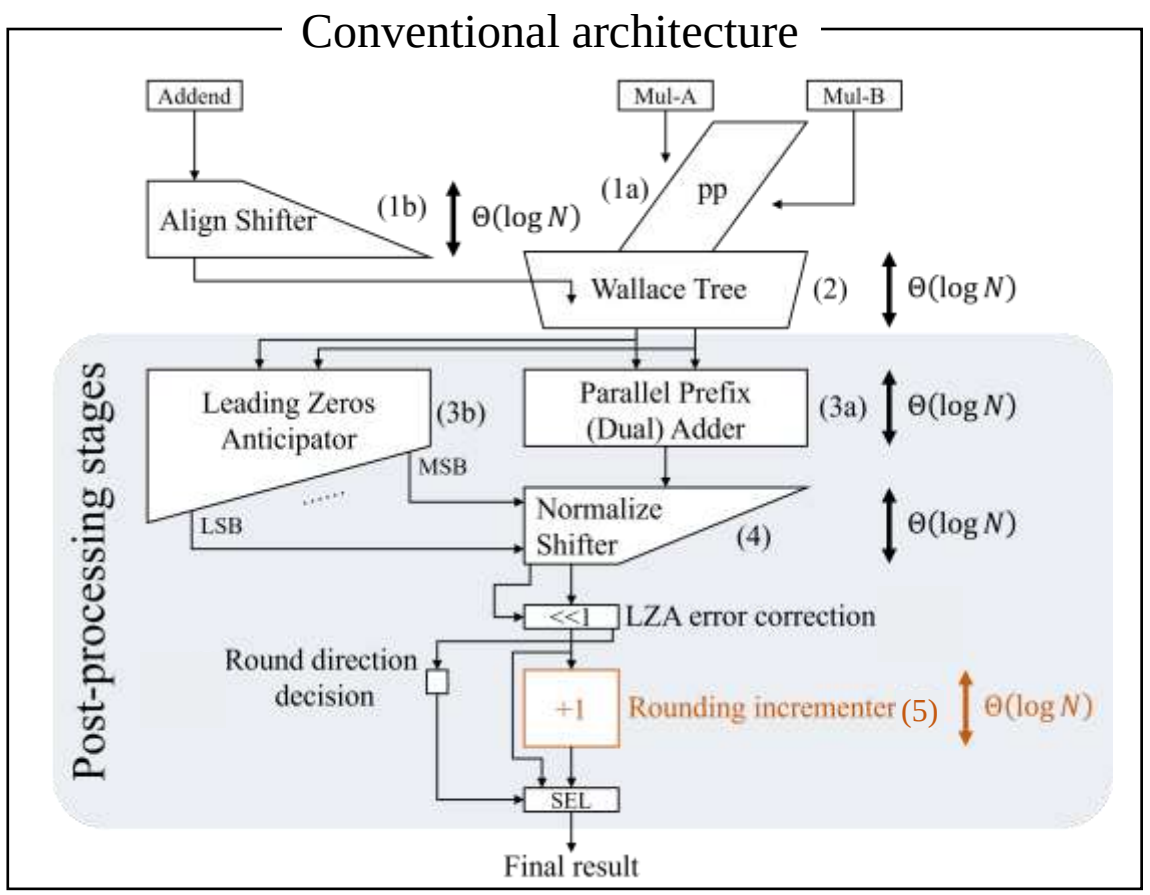
- How can we obtain the trailing ones of before-rounding mantissa?

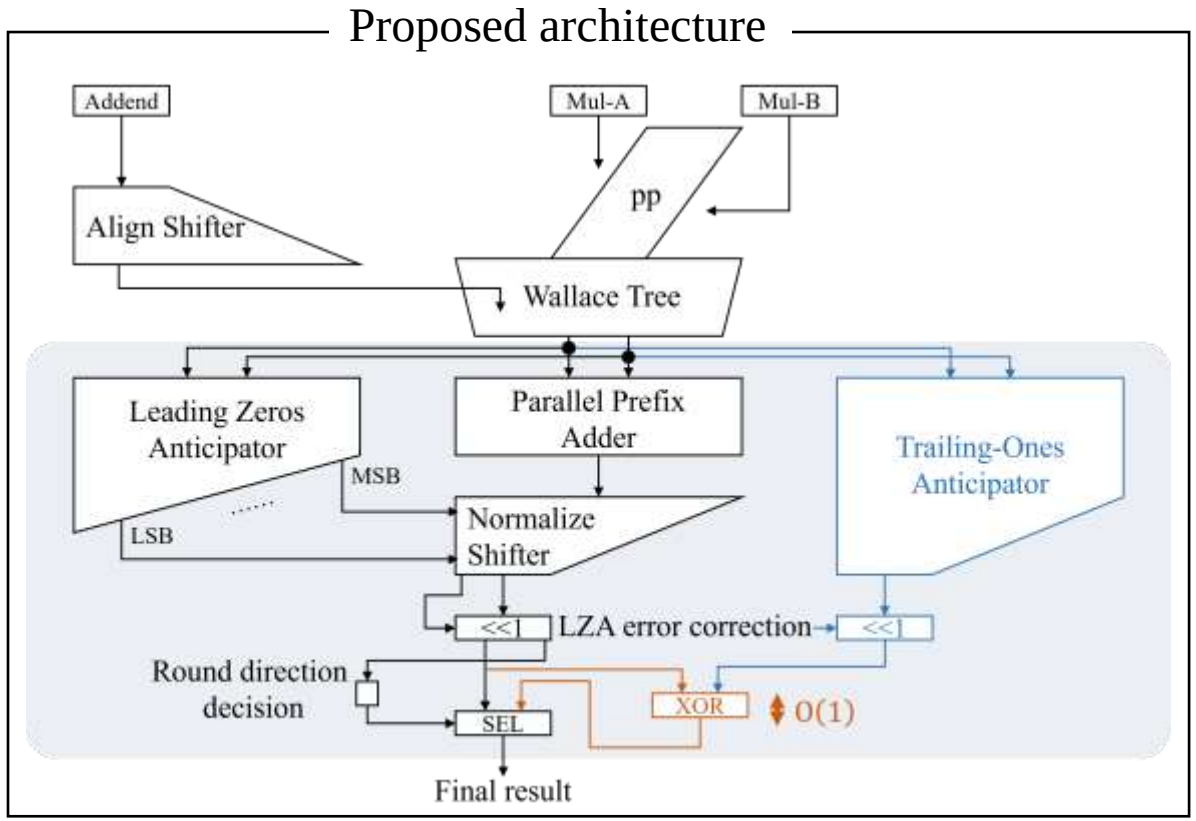
  → Our solution: trailing-ones anticipation

Table of contents

Proposed architecture

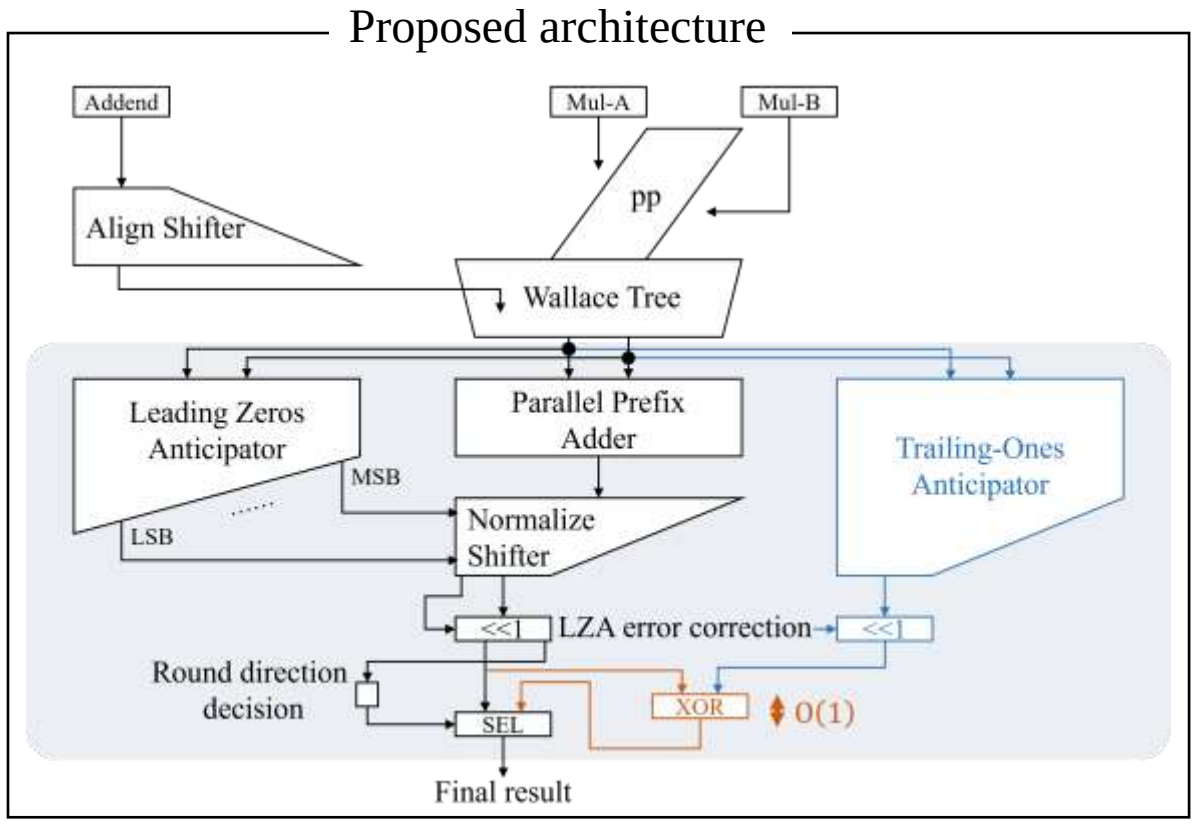
- A trailing-ones anticipator (TOA) is placed in parallel with (3) and (4).





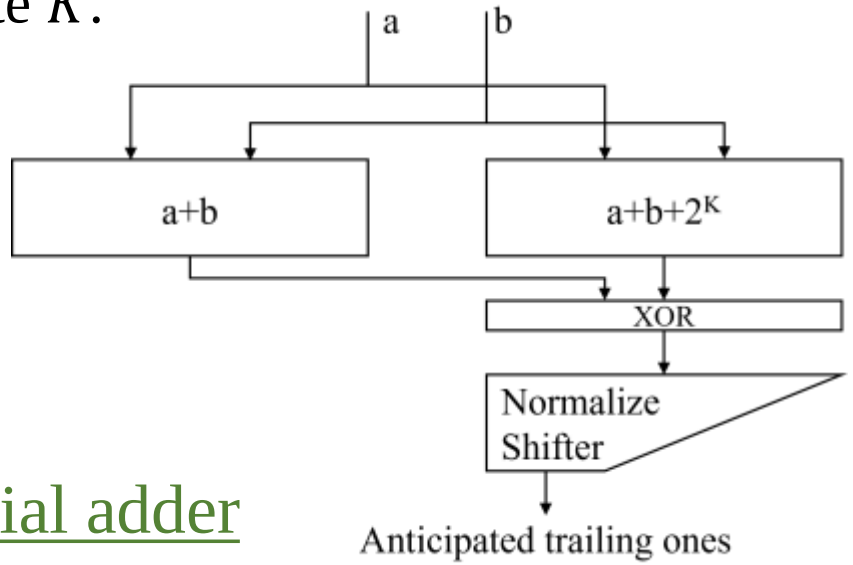
Proposed architecture

- A trailing-ones anticipator (TOA) is placed in parallel with (3) and (4).
- The TOA circuit outputs the trailing ones of the *before-rounding* (= *after-normalized*) mantissa using the output of the Wallace tree.



Trailing-ones anticipator: Overview

- The TOA circuit is built on a simple idea:
  - First, calculate $a + b$ and $a + b + 2^K$ for an appropriate K.
    - 2^K is the last place of before-rounding mantissa.
  - XOR them, then normalize.



- The problem: How to obtain K or 2^K?
  - Explicitly obtaining the value of K or 2^K takes $\Theta(\log N)$ time*; thus, such an architecture will not reduce the total latency.

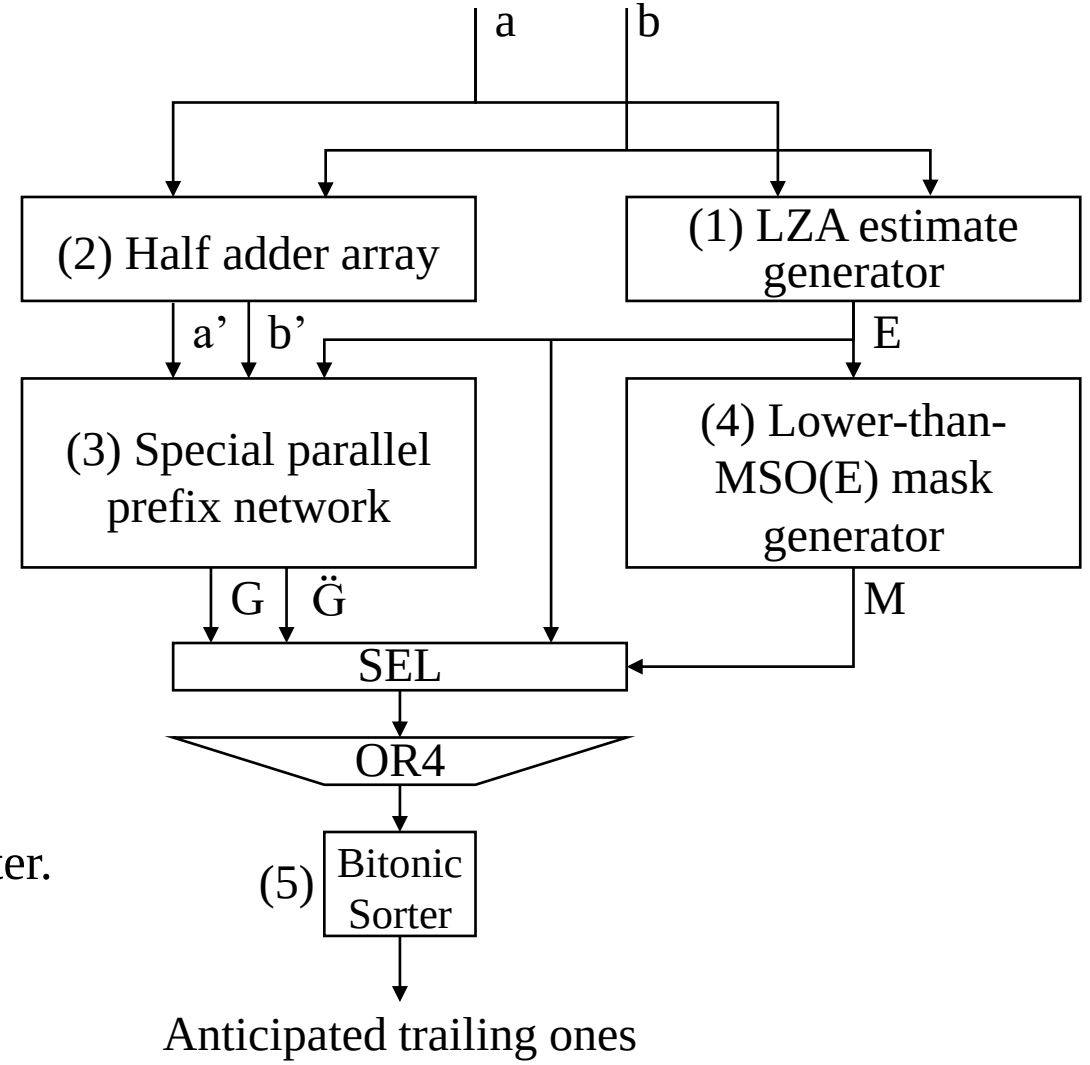
→ Our solution: use an estimate value E and a special adder

  - Our special adder calculate $a + b + \mathrm{MSO}(E)$.
    - $\mathrm{MSO}(x)$ denotes a binary number where all bits except the most significant 1 of x are set to 0.
      Example: $\mathrm{MSO}(\mathrm{0b0000}10011) = \mathrm{0b0000}10000$
  - E needs satisfy $\mathrm{MSO}(E) = 2^K$ *or* 2^{K+1} and can be obtained in $O(1)$ time.

* Since changing any bit in a or b may change the value of K, it needs $\Theta(\log N)$-depth circuit.
Note: Although the dual adder can calculate $a + b + 1$ but it cannot calculate $a + b + 2^K$ for arbitrary K.

Trailing-ones anticipator: Detailed architecture

- Five main components in our architecture:
  - (1) LZA estimate generator
    - The same as in the conventional design
    - Just feed its output into (3) and (4).
  - (2) Half-adder array
    - Constrains the inputs for (3).
  - (3) Special parallel prefix network
    - The most novel part in this architecture
  - (4) Lower-than-MSO(E) mask generator
    - Almost the same as Lutz's mask
  - (5) Bitonic sort-based shifter
    - Normalize shifting can be done by a sorter!
    - The bitonic sorter is faster and smaller than a shifter.



(1) Estimating 2^K

- An LZA estimate L can be obtained in $O(1)$ time.
- It satisfies $2^{N+K} \leq L < 2^{N+K+2}$.
- Thus, we can use $E = L \gg N$.
  - It can be obtained in $O(1)$ time.
  - It satisfies $2^K \leq \mathrm{E} < 2^{K+2}$.
  - This means that $\mathrm{MSO}(E)$ is either 2^K or 2^{K+1}.
  - An LZA error correction is needed; explained later.

(2) Half-adder array

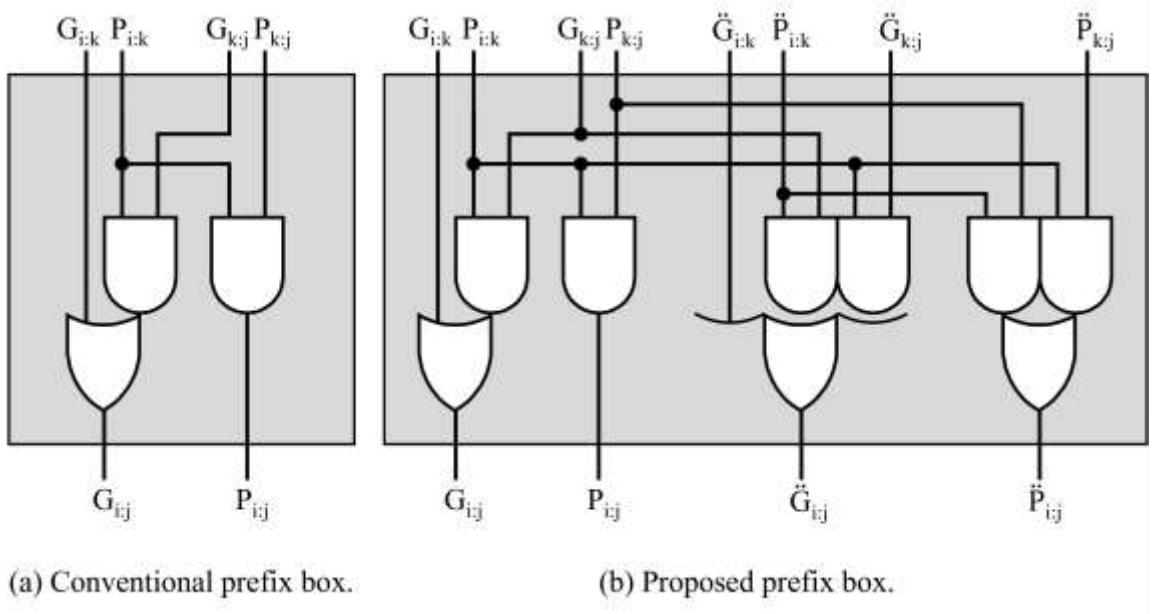
- We want to calculate $a + b + \mathrm{MSO}(E)$.
- Parallel prefix adders cannot handle this directly
  - because two carries can be generated in the same digit.
- To address this issue, a half-adder array is employed.
  - A half-adder array convert a and b to a' and b' where $a + b = a' + b'$ satisfies.
  - For any i, $\sum_{k=0}^{i} 2^k ({a'}_k + {b'}_k) < 3 \cdot 2^i$ (cf. $\sum_{k=0}^{i} 2^k (a_k + b_k) < 4 \cdot 2^i$) holds.
  - It also holds that $\sum_{k=0}^{i} 2^k ({a'}_k + {b'}_k + \mathrm{MSO}(E)_k) < 4 \cdot 2^i$.
  - This guarantees the carry from any digit will be at most 1 (and will not be 2).

  → We can use a parallel prefix adder.

(3) Special parallel prefix adder (1/2)

- The special parallel prefix adder calculates $a + b + \mathrm{MSO}(E)$ from a, b, and E (not $\mathrm{MSO}(E)$).

→ We introduce a special prefix box.

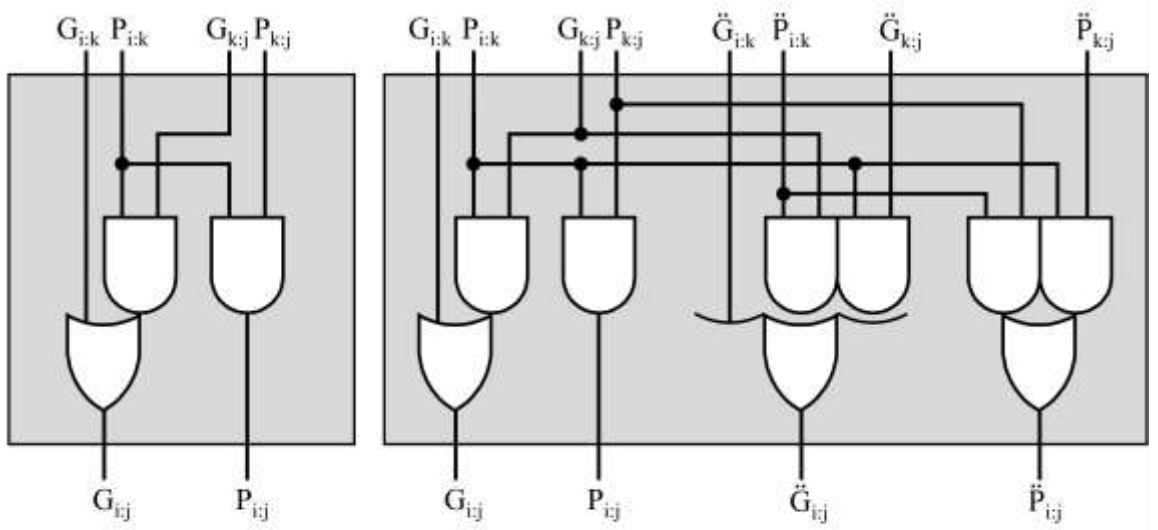


(a) Conventional prefix box. (b) Proposed prefix box.

- Conventional signals: $G_{i:j}$ and $P_{i:j}$
  - The generate and propagate signals
- Novel signals: $\ddot{G}_{i:j}$ and $\ddot{P}_{i:j}$
  - It means that, if MSO(E) is in between i and j, generates carry ($\ddot{G}_{i:j}$) or propagates carry ($\ddot{P}_{i:j}$).

(3) Special parallel prefix adder (2/2)

- Calculation method of novel signals:
  - $\ddot{G}_{i:j} = \ddot{G}_{i:k} \vee \left(P_{i:k} \wedge \ddot{G}_{k:j}\right) \vee \left(\ddot{P}_{i:k} \wedge G_{k:j}\right)$
  - $\ddot{P}_{i:j} = \left(P_{i:k} \wedge \ddot{P}_{k:j}\right) \vee \left(\ddot{P}_{i:k} \wedge P_{k:j}\right)$
- These equations are derived by adding exactly one umlaut to each term and summing up them.
  - $P_{i:k} \wedge P_{k:j} \rightarrow P_{i:k} \wedge \ddot{P}_{k:j}$ and $\ddot{P}_{i:k} \wedge P_{k:j}$
  - $G_{i:k} \vee \left(P_{i:k} \wedge G_{k:j}\right) \rightarrow \ddot{G}_{i:k}$, $P_{i:k} \wedge \ddot{G}_{k:j}$, and $\ddot{P}_{i:k} \wedge G_{k:j}$
  - "Exactly one umlaut" comes from the fact that MSO(E) has exactly one 1 bit.
    - The only 1 cannot simultaneously exist both *between i and k* and *between k and j*.



(a) Conventional prefix box. (b) Proposed prefix box.

(4) Lower-than-MSO(E) mask generator

- The propagate/generate signals are
  - P/G if the position is lower than MSO(E)
  - P̈/G̈ otherwise
- Lower-than-MSO(E) mask *M* is used to select from P/G or P̈/G̈.
  - Example: 0b0001010010 [E]
    0b0000111111 [M]
- The following algorithm can calculate *M*.

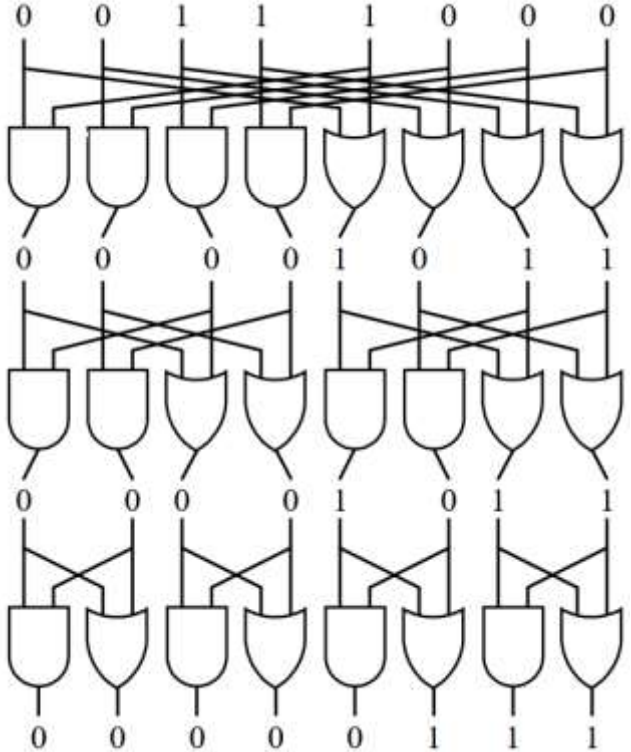
```
M = E >> 1;
for shamt in { 1, 2, 4, 8, 16, ... }:
   M |= M >> shamt;
```

(5) Bitonic sort-based shifter (1/2)

- The output of the special prefix adder is a bitonic sequence.
  - Bitonic sequence: 0*1*0* (or 1*0*1*)
- Trailing ones start at the right end by their nature.
  - Trailing ones: 0*1*

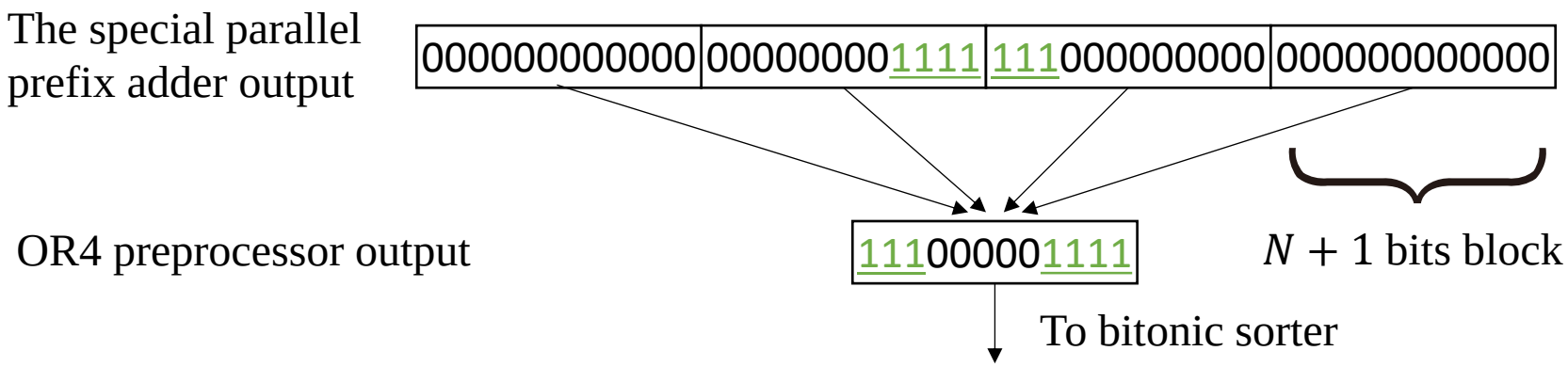
→ Sorting instead of shifting can be used.

- A bitonic sorter is faster than a shifter because it does not have large fan-out structures.



(5) Bitonic sort-based shifter (2/2)

- The width of the sorting circuit can be reduced.
  - The special parallel prefix adder outputs $4N + \Theta(1)$ bits but only an $N + 1$ bits bitonic sorter is needed if OR4 preprocessor is placed before the sorter.
- OR4 preprocessor leverages the fact that
  - trailing-ones cannot contain more than $N + 1$ 1 bits, and
  - a bitonic sorter can sort 1*0*1* sequences as well as 0*1*0* sequences.



Error correction

- The obtained trailing ones may have one bit position error.
- Classifying in three cases can deal this error.

(1) The before-rounding mantissa is an even number

- Simply invert last bit. Trailing ones are not used.

mantissa 000001100110 → 000001100111

(2) The before-rounding mantissa is an odd number and LZA is correct

- Simply XOR the mantissa and the trailing ones.

mantissa 000001100111 → 000001101000
trail. ones 000000001111

(3) The before-rounding mantissa is an odd number and LZA mispredicts

- XOR the mantissa and one-bit shifted trailing ones, then invert last bit.

mantissa 000001100111 → 000001101000
trail. ones 000000000111

Negative case handling

- Positive case: $a + b + 2^K$ is needed.
- Negative case: $-a - b + 2^K$ is needed.
- We have been unable to find a circuit configuration that can simultaneously compute both patterns.

→ Simply prepare both circuits

Table of contents

Evaluation methodology

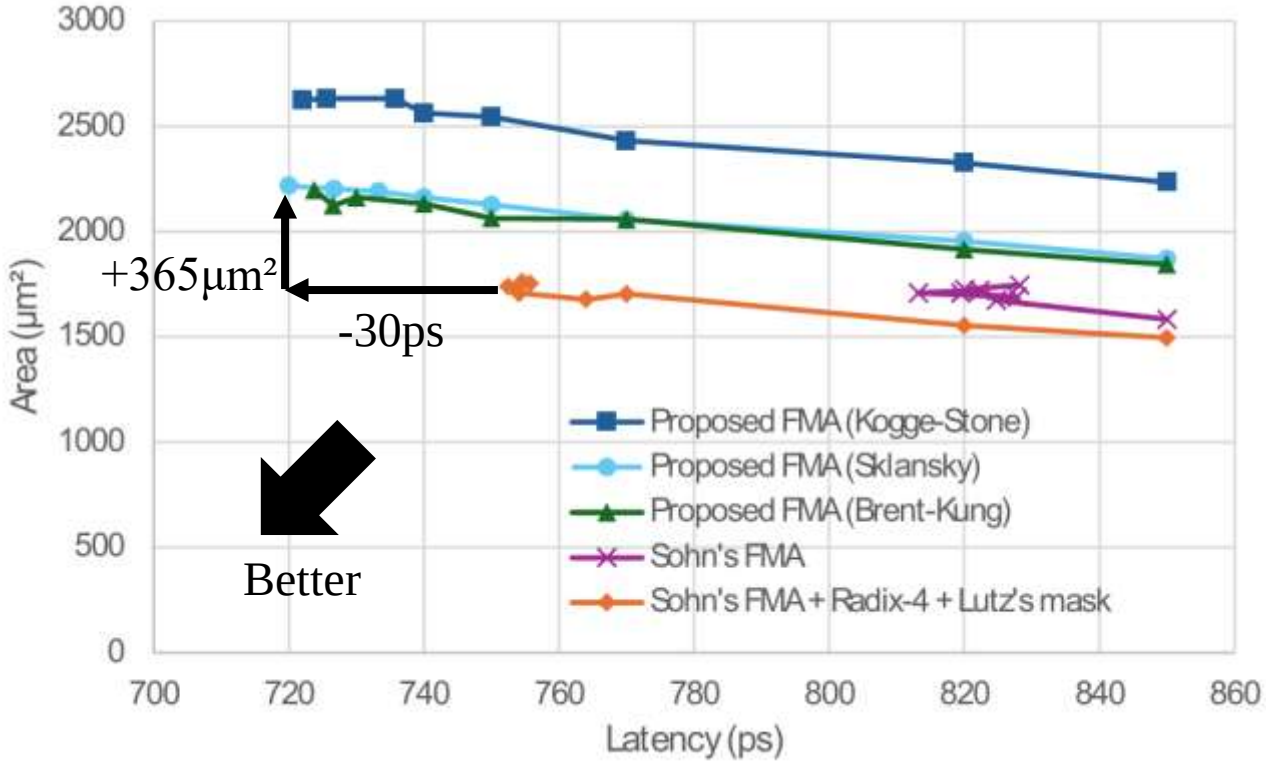
- We implemented the following three FMA circuits in SystemVerilog.
  - Sohn's FMA [2]
    - It seems to be oriented towards saving circuit area.
  - Sohn's FMA with Lutz's technique [3] and Radix-4 Booth multiplier
    - Festest known implementation.
  - Our FMA (with Lutz's technique and Radix-4 Booth multiplier)

- We synthesized the design by Synopsys Design Compiler with a standard cell library derived from the ASAP7 7nm finFET PDK.

[2] J. Sohn, D.K. Dean, E. Quintana, and W.S. Wong, "Enhanced floating point multiply-add with full denormal support," in *2023 IEEE 30th Symposium on Computer Arithmetic (ARITH)*, 2023, pp. 143–150.
[3] D.R. Lutz, "Optimized leading zero anticipators for faster fused multiply-adds," in *2017 51st Asilomar Conference on Signals, Systems, and Computers*, 2017, pp. 741–744.

Evaluation results — latency and area

- Our design improved the total latency by ~30 ps (4%).
- The area increased by 26% (when using Brent-Kung or Sklansky structure for proposed special adder).
- Using Kogge-Stone structure for proposed special adder did not yield additional speed-up and only resulted in an unnecessarily large area.



Conclusion

- We proposed a novel FMA architecture using trailing-ones anticipation.
  - The rounding incrementor can be replaced by a simple XOR operation; thus, the total latency is reduced.
- The key is a special parallel prefix adder using $\ddot{G}_{i:j}$ and $\ddot{P}_{i:j}$.
  - It can calculate $a + b + 2^K$ without explicitly computing K or 2^K.
- The evaluation results show it contributes reducing the latency by 4%.
  - However, the area is increased by 26%.

- Future work: negative case handling in a single circuit

Area breakdown

	Sohn's FMA [2]	Sohn's + Radix-4 + Lutz's mask	Proposed FMA
Exponent Difference & Alignment	Exponent adder (30), Alignment shifter (120) Complement (25), Sticky (10)	Exponent adder (30), Alignment shifter (120) Complement (25), Sticky (10)	Exponent adder (30), Alignment shifter (120) Complement (25), Sticky (10)
Multiplier	Hard multiplier (145) Booth selector (540), Denormal adjust (5) Wallace tree (300)	Booth selector (380), Denormal adjust (5) Wallace tree (455)	Booth selector (370), Denormal adjust (5) Wallace tree (430)
Main Adder	PPA (150), Complement (25)	PPA (120), Complement (30)	PPA (115), Complement (30) **Special PPA (410)**
LZA	LZA input generation (100), LZC (40)	LZA input generation (100), LZC (40) Lutz's mask generation (55)	LZA input generation (100), LZC (40) Lutz's mask generation (55)
Normalization	Normalize shifter (100), Exponent adder (5) Sticky/All ones (10)	Normalize shifter (100), Exponent adder (5) Sticky/All ones (10), Lutz's mask tree (90)	Normalize shifter (85), Exponent adder (5) Sticky/All ones (10), Lutz's mask tree (85) **Bitonic sorter-based shifter (25)**
Rounding	Adjust shifter (10), Round decision (5) Incrementer (25) Result selector (20)	Adjust shifter (10), Round decision (5) Incrementer (25) Result selector (20)	Adjust shifter (15), Round decision (5) **XOR (10)** Result selector (20)
Other	Special case handler (25) Exception flags (5)	Special case handler (25) Exception flags (5)	Special case handler (25) Exception flags (5)
Total	1695	1665	2030

Latency breakdown

	Sohn's FMA [2]	Sohn's + Radix-4 + Lutz's mask	Proposed FMA
Exponent Difference & Alignment	Exponent adder (95) Alignment shifter (135) Complement (15)	Exponent adder (85) Alignment shifter (135) Complement (15)	Exponent adder (95) Alignment shifter (140) Complement (15)
Multiplier	(rest of) Wallace tree (155)	(rest of) Wallace tree (80)	(rest of) Wallace tree (100)
Main Adder		PPA (120) Complement (55)	
LZA	LZA input generation (40) LZC (75)		LZA input generation (70) LZC (70)
Normalization	Normalize shifter (115)	Normalize shifter (95)	Normalize shifter (155)
Rounding	Adjust shifter (40) Incrementer (95) Result selector (10)	Adjust shifter (20) Incrementer (95) Result selector (10)	Adjust shifter (20) XOR (10) Result selector (10)
Other	Flip-flop setup (40)	Flip-flop setup (40)	Flip-flop setup (35)
Total	810	750	**720 (fastest)**