New Algorithms and Proofs for High-Precision Arithmetic

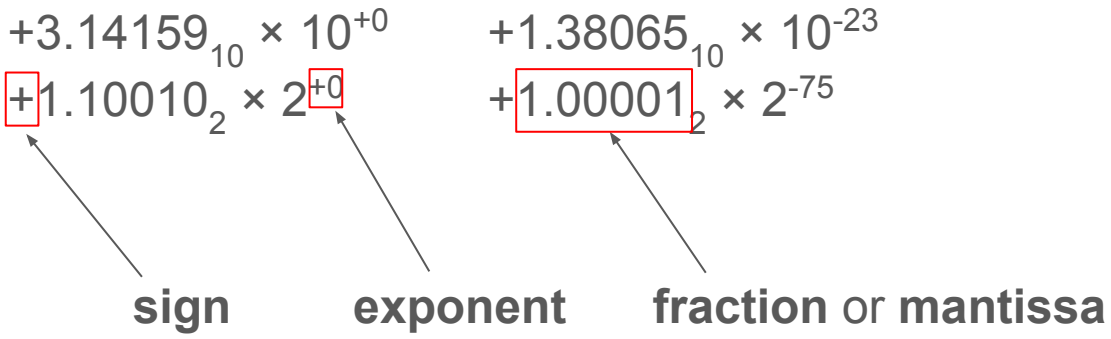
LA/Opt Seminar 2025-02-05

David K. Zhang
Stanford University - ICME

Floating-Point Numbers

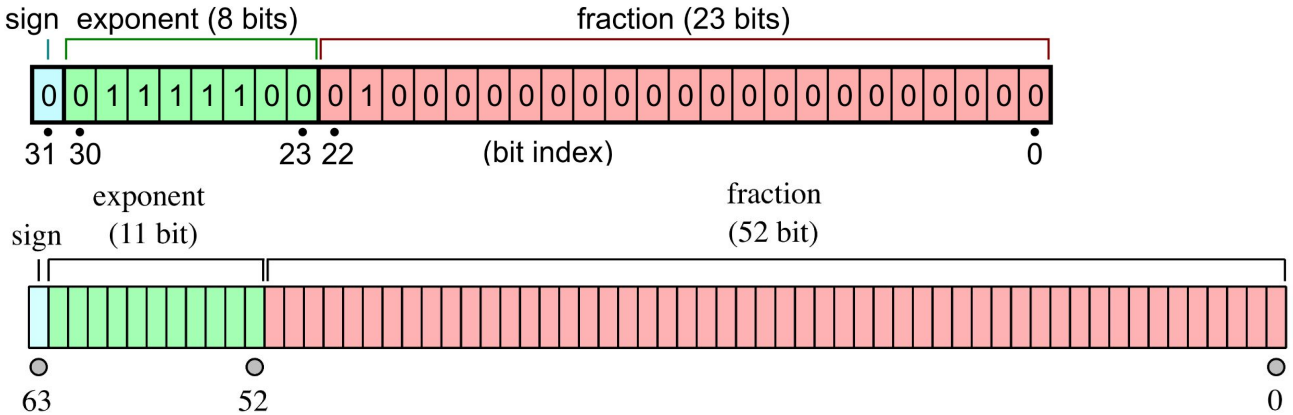
- Floating-point numbers approximately represent **finite-precision real numbers** inside digital computers.

- They generalize **scientific notation** from base 10 to base 2.



Floating-Point Formats

- Floating-point processors usually support IEEE 754 binary**32** (`float`) and binary**64** (`double`)



- What do we do when binary**32**/binary**64 isn't enough?**

Why High Precision?

- Some applications demand final answers to very high precision.

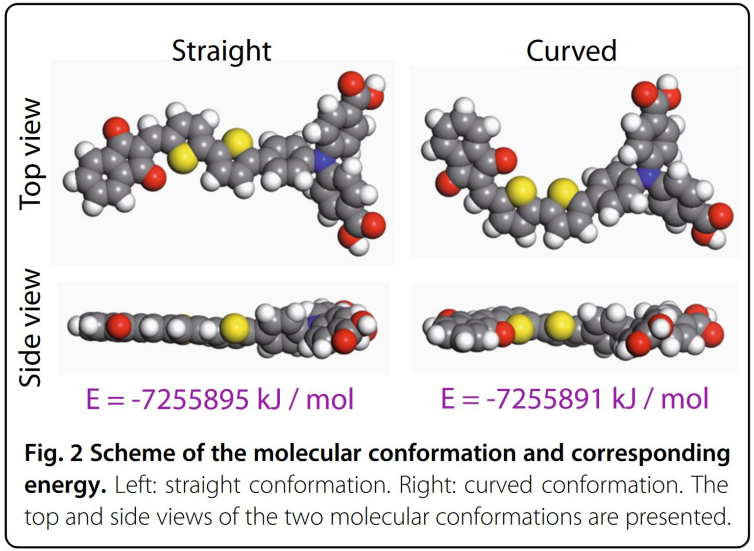


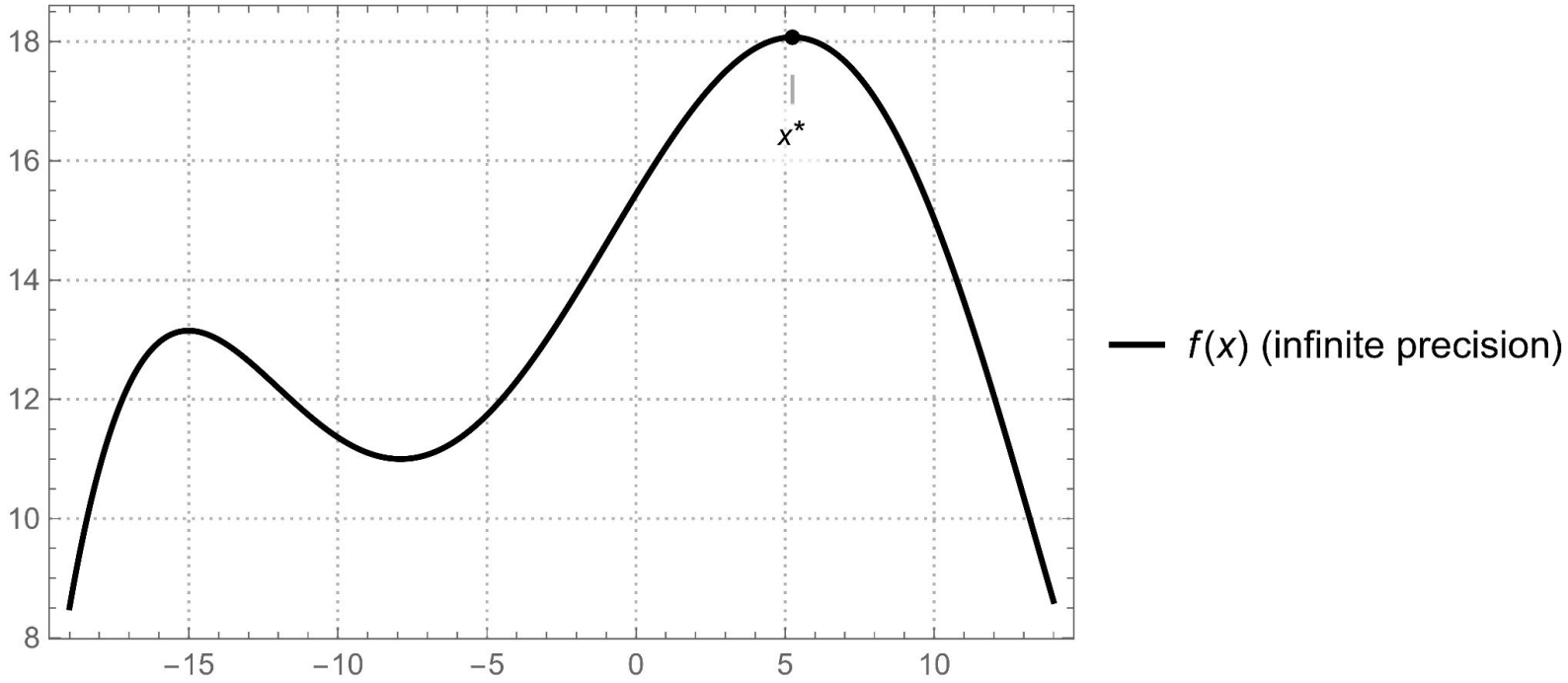
Fig. 2 Scheme of the molecular conformation and corresponding energy. Left: straight conformation. Right: curved conformation. The top and side views of the two molecular conformations are presented.

Why High Precision?

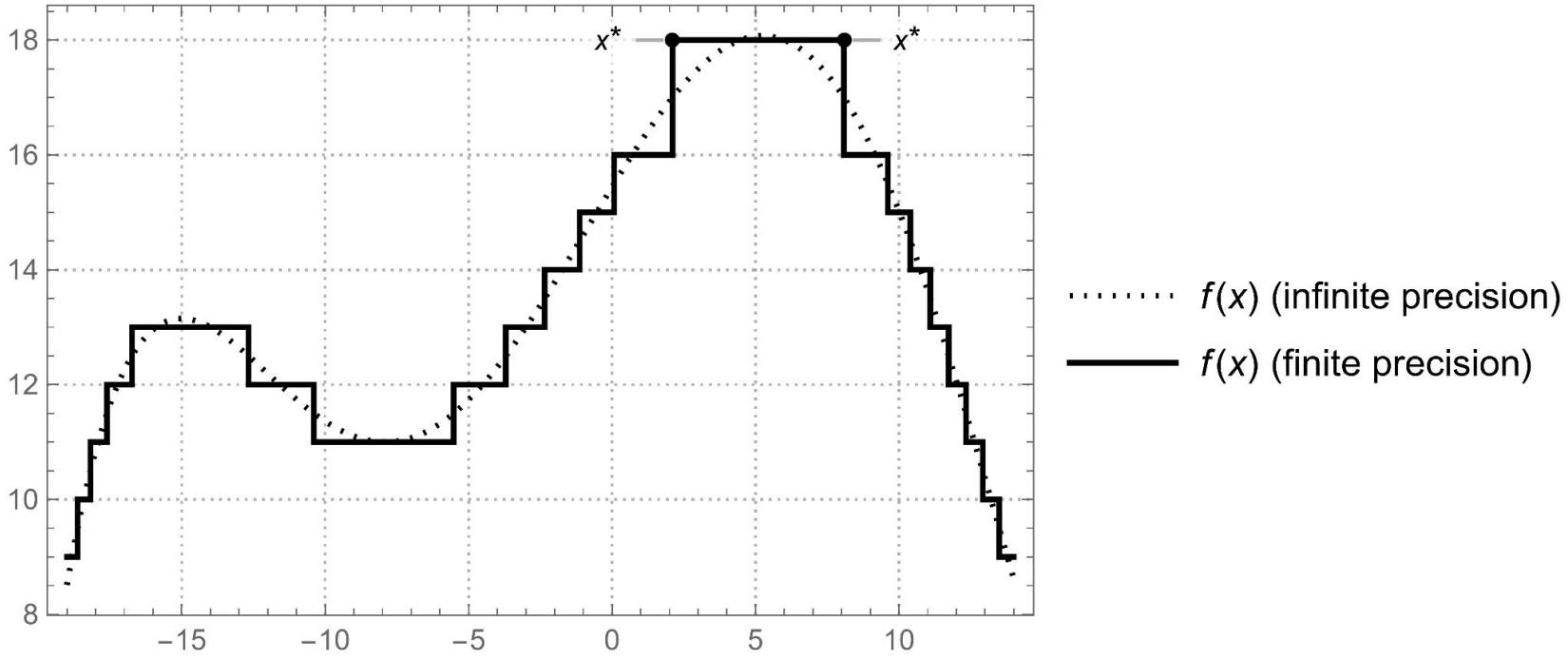
- Some applications demand final answers to very high precision.

- Even when you only need the **final** answer to **moderate** precision, you may need **intermediate** calculations to **high internal precision.**

Example: Function Optimization



Example: Function Optimization



Approximate Optimizers in Finite Precision

Theorem: Let $f : \mathbb{R} \to \mathbb{R}$ be twice differentiable at a local optimizer $x^* \in \mathbb{R}$ with $f(x^*) > 0$ and $f''(x^*) \neq 0$. For all sufficiently small $\epsilon > 0$, every value of x contained in the interval

$$x^* - K\sqrt{\epsilon} < x < x^* + K\sqrt{\epsilon}$$

satisfies the condition

$$(1 - \epsilon)f(x^*) < f(x) < (1 + \epsilon)f(x^*)$$

where $K = \sqrt{f(x^*)/|f''(x^*)|}$.

Proof: By Taylor's theorem, we can write

$$f(x) = f(x^*) + \frac{1}{2}f''(x^*)(x - x^*)^2 + R_2(x)(x - x^*)^2$$

where $R_2 : \mathbb{R} \to \mathbb{R}$ is a function satisfying $\lim_{x \to x^*} R_2(x) = 0$. Thus, there exists an ϵ_0 such that for all $\epsilon \in (0, \epsilon_0)$, if $|x - x^*| < K\sqrt{\epsilon}$, then $R_2(x) < \frac{1}{2}|f''(x^*)|$. This implies that

$$|f(x) - f(x^*)| \leq \frac{1}{2}f(x^*)\epsilon + R_2(x)K^2\epsilon < \epsilon f(x^*)$$

which completes the proof. ■

To locate *x** with precision ***p***, you must compute *f*(*x*) with precision **2*p***.

That's assuming no **internal** precision loss inside of *f*(*x*); **worse in practice.**

Operation	Desired Precision	Required Internal Precision
function optimization $\arg\min_x f(x)$	p bits	$2p + \log_2 \kappa_f$ bits
finite-difference first derivative $\frac{f(x+h) - f(x)}{h}$	p bits	$2p + \log_2 \kappa_f$ bits
finite-difference second derivative $\frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$	p bits	$2p + 2\log_2 \kappa_f$ bits
numerical definite integral $\int_a^b f(x)\,\mathrm{d}x$	p bits	$p + \log_2 \frac{\int_a^b \lvert f(x)\rvert\,\mathrm{d}x}{\left\lvert\int_a^b f(x)\,\mathrm{d}x\right\rvert}$ bits
solve linear system $A\mathbf{x} = \mathbf{b} \implies \mathbf{x} = A^{-1}\mathbf{b}$	p bits	$p + \log_2 \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$ bits
compute eigenvalue $A\mathbf{v} = \lambda\mathbf{v}$	p bits	$p + \log_2 \frac{1}{\lvert\mathbf{v}_L \cdot \mathbf{v}_R\rvert}$ bits
evaluate transcendental function $\sin(x),\ \log(x),\ \Gamma(x),\ \ldots$	p bits	usually $2p$ to $3p$ bits

Complex Numerical Applications

- Combining inexact operations **compounds** loss of precision.
  - Insufficient numerical precision is a **barrier to composability**.
- How do applied mathematicians, scientists, and engineers overcome this?
- Use sophisticated numerical methods to **reformulate problems** and **reduce error sensitivity**
  - each problem type requires its own error analysis
  - requires ingenuity and numerical analysis expertise
  - not always possible

Arbitrary-Precision Arithmetic Libraries

GMP

«Arithmetic without limitations»

Arb

CLN - Class Library for Numbers

- Use **arbitrary-precision libraries** to do arithmetic at any precision level
- Mature, well-tested, proven accuracy
- **CPU-only** and **2,000-4,000x slower** than native machine arithmetic

```
void mpf_add(mpf_ptr r, mpf_srcptr u, mpf_srcptr v) {

    /* Allocate temp space for the result.  Allocate
       just vsize + ediff later???  */
    tp = TMP_ALLOC_LIMBS(prec);

    if (ediff >= prec) {
        /* V completely cancelled.  */
        if (rp != up) MPN_COPY_INCR(rp, up, usize);
        rsize = usize;
    } else {
        /* uuuu     |  uuuu     |  uuuu     |  uuuu     |  uuuu     */
        /* vvvvvvv  |  vv       |    vvvvv  |    v      |       vv */
        if (usize > ediff) {
            /* U and V partially overlaps.  */
            if (vsize + ediff <= usize) {
                /* uuuu     */
                /*   v      */
                mp_size_t size;
                size = usize - ediff - vsize;
                MPN_COPY(tp, up, size);
                cy = mpn_add(tp + size, up + size, usize - size, vp, vsize);
                rsize = usize;
            } else {
                /* uuuu     */
                /*   vvvvv  */
```

GMP

«Arithmetic without limitations»

STOP USING ARBITRARY PRECISION

This is REAL code, written by REAL programmers:

```
2 + 2 = malloc()
```

They have played us for absolute fools

Arbitrary-Precision Design Tradeoffs

- GMP, MPFR, Arb, and CLN are excellent libraries
  designed for **massive** numbers with **thousands** to **millions** of digits.

- In this regime, complex control structures and
  memory allocation are perfectly reasonable costs.

- We often need a **small increase** in precision (128-bit, 256-bit, …)
  without going all the way to arbitrary precision.

Floating-Point is Branchy

- Floating-point representation is **defined** in terms of complex branching case analysis.

- IEEE 754: The representation r of the floating-point datum, and value v of the floating-point datum represented, are inferred from the constituent fields as follows:

  a) If $E = 2^w - 1$ and $T \neq 0$, then r is qNaN or sNaN and v is NaN regardless of S and then d_1 shall exclusively distinguish between qNaN and sNaN (see 6.2.1).

  b) If $E = 2^w - 1$ and $T = 0$, then r and $v = (-1)^S \times (+\infty)$.

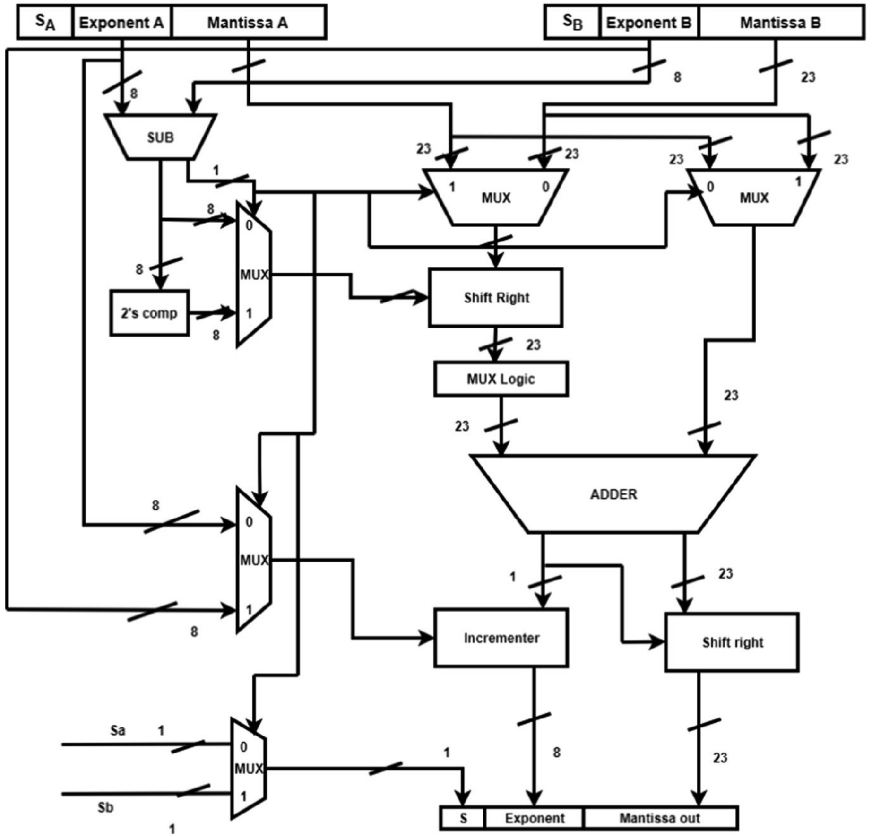
  c) If $1 \leq E \leq 2^w - 2$, then r is $(S, (E - bias), (1 + 2^{1-p} \times T))$;
  the value of the corresponding floating-point number is $v = (-1)^S \times 2^{E-bias} \times (1 + 2^{1-p} \times T)$;
  thus normal numbers have an implicit leading significand bit of 1.

  d) If $E = 0$ and $T \neq 0$, then r is $(S, emin, (0 + 2^{1-p} \times T))$;
  the value of the corresponding floating-point number is $v = (-1)^S \times 2^{emin} \times (0 + 2^{1-p} \times T)$;
  thus subnormal numbers have an implicit leading significand bit of 0.

  e) If $E = 0$ and $T = 0$, then r is $(S, emin, 0)$ and $v = (-1)^S \times (+0)$ (signed zero, see 6.3).

Floating-Point is Branchy

- Native floating-point operations are fast because this branching is implemented in **hardware**.
- Even if we eliminate memory allocation, these control structures must remain.
- Can we leverage these circuits to handle the logic for us?



Floating-Point Expansions

- **Idea**: Represent a high-precision number using a sequence of **successive machine-precision approximations**.

3.14159265359

$3.14159 \times 10^{+0}$

2.65359×10^{-6}

$$
\begin{aligned}
x_0 &:= \mathsf{RNE}(C) \\
x_1 &:= \mathsf{RNE}(C - x_0) \\
x_2 &:= \mathsf{RNE}(C - x_0 - x_1) \\
&\vdots \\
x_{n-1} &:= \mathsf{RNE}(C - x_0 - x_1 - \cdots - x_{n-2})
\end{aligned}
$$

Nonoverlapping Invariant

- To achieve full precision, a floating-point expansion must be **nonoverlapping.**

$$\begin{aligned}
&\text{high-precision constant} && C = 1\,0\,1\,.\,0\,1\,1\,1\,0\,1\,1\,0\,1\,0\,1\,1\,\ldots \\
&\text{expansion with } |x_1| > \mathsf{ulp}(x_0) && \left\{ \begin{array}{ll} x_0 = & 1\,0\,1\,.\,0\,0\,0 \\ x_1 = & \quad 0\,.\,0\,1\,1\,1\,0\,1\,1 \end{array} \right\} \text{ 10-bit precision} \\
&\text{expansion with } |x_1| \leq \mathsf{ulp}(x_0) && \left\{ \begin{array}{ll} x_0 = & 1\,0\,1\,.\,0\,1\,1 \\ x_1 = & \quad 0\,.\,0\,0\,0\,1\,0\,1\,1\,0\,1 \end{array} \right\} \text{ 12-bit precision} \\
&\text{expansion with } |x_1| \leq \tfrac{1}{2}\,\mathsf{ulp}(x_0) && \left\{ \begin{array}{ll} x_0 = & 1\,0\,1\,.\,1\,0\,0 \\ x_1 = & -0\,.\,0\,0\,0\,0\,1\,0\,0\,1\,0\,1 \end{array} \right\} \text{ 13-bit precision}
\end{aligned}$$

Floating-Point Expansion Arithmetic

- How do we add two floating-point expansions?
- This is easy with integers — just **carry** one bit.
  - Hardware support: x86 `adc`, arm `adc`
- Floating-point addition **rounds** instead of carrying.

```
   937        93.7
+   75      +  7.54
-----      -------
1012        101.24
```

(carry bit) 1 **(rounded off)**

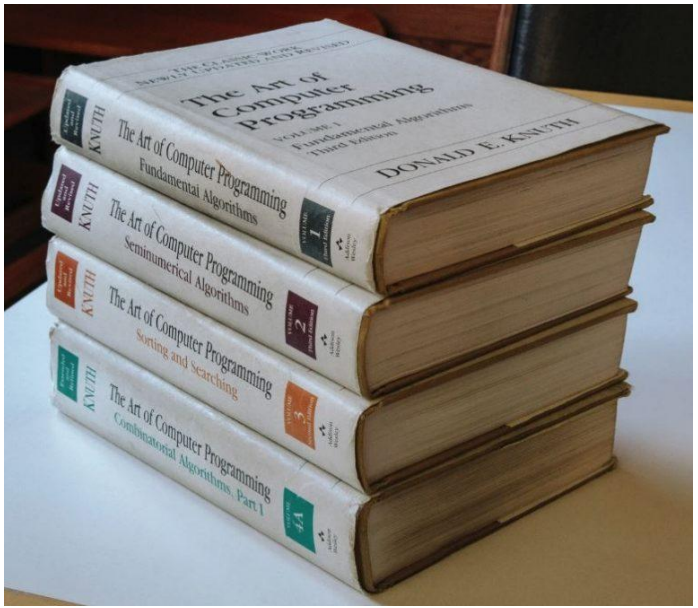
TwoSum

- `(s: FP, e: FP) = TwoSum(a: FP, b: FP)`
  - Here, `FP` denotes any floating-point type (`float`, `double`, etc.)

- TwoSum computes the **rounded sum** `s := a ⊕ b` and the **exact roundoff error** `e := (a + b) - (a ⊕ b)`.
  - `a + b`: true mathematical sum of a and b
  - `a ⊕ b`: rounded floating-point sum of a and b

- Miraculously, it does this using **only rounded operations**.
  - must **round to nearest** (with any tie-breaking rule)

TwoSum Origins



Theorem A. *Let u and v be normalized floating point numbers. Then*

$$\big((u \oplus v) \ominus u\big) + \big((u \oplus v) \ominus \big((u \oplus v) \ominus u\big)\big) = u \oplus v, \tag{40}$$

provided that no exponent overflow or underflow occurs.

This rather cumbersome-looking identity can be rewritten in a simpler manner: Let

$$\begin{aligned} u' &= (u \oplus v) \ominus v\,, \qquad & v' &= (u \oplus v) \ominus u\,; \\ u'' &= (u \oplus v) \ominus v', & v'' &= (u \oplus v) \ominus u'. \end{aligned} \tag{41}$$

Intuitively, u' and u'' should be approximations to u, and v' and v'' should be approximations to v. Theorem A tells us that

$$u \oplus v = u' + v'' = u'' + v'. \tag{42}$$

This is a stronger statement than the identity

$$u \oplus v = u' \oplus v'' = u'' \oplus v', \tag{43}$$

which follows by rounding (42).

TwoSum Algorithm

```
function TwoSum(a: FP, b: FP)
    s := a ⊕ b
    a_eff := s ⊖ b
    b_eff := s ⊖ a_eff # a_eff + b_eff == s
    a_err := a ⊖ a_eff
    b_err := b ⊖ b_eff
    e := a_err ⊕ b_err
    return (s, e)
end
```

TwoSum Example

```
function TwoSum(a := 93.7, b := 7.54)
    s := a ⊕ b = 93.7 + 7.54 = 101.24 => 101
    a_eff := s ⊖ b = 101 - 7.54 = 93.46 => 93.5
    b_eff := s ⊖ a_eff = 101 - 93.5 = 7.5
    a_err := a ⊖ a_eff = 93.7 - 93.5 = 0.2
    b_err := b ⊖ b_eff = 7.54 - 7.5 = 0.04
    e := a_err ⊕ b_err = 0.2 + 0.04 = 0.24
    return (101, 0.24)
end
```

Floating-Point Accumulation Networks

- TwoSum allows us to exactly add **two** floating-point numbers, but computing with floating-point expansions requires **more**.

- **Example:** How do we add two expansions of length 2?
  Need to accumulate **4 inputs** into **2 outputs**.
  $(z_0, z_1) := (x_0, x_1) + (y_0, y_1)$

- Length-2 expansions are called “**double-double arithmetic**” and have an interesting history…

Dekker's add2 Algorithm

A Floating-Point Technique
for Extending the Available Precision

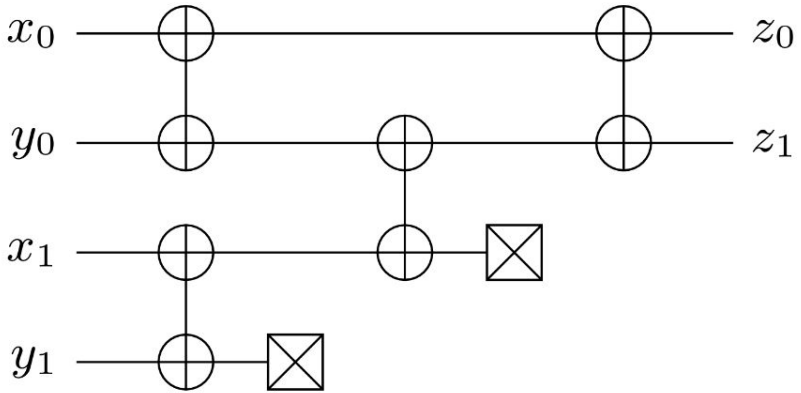
Abstract. A te
metic in terms o
available (say: si
algorithms are e
numbers, deliveri
forward applicati
arithmetic which

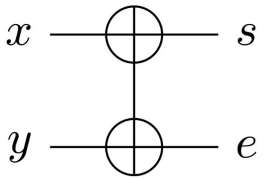
```
comment add2 calculates the doublelength sum of (x, xx) and (y, yy), the result
        being (z, zz);

procedure add2 (x, xx, y, yy, z, zz);
value x, xx, y, yy; real x, xx, y, yy, z, zz;
begin real r, s;
        r := x + y;
        s := if abs (x) > abs (y) then
             x - r + y + yy + xx else y - r + x + xx + yy;
        z := r + s;
       zz := r - z + s
end add2;
```

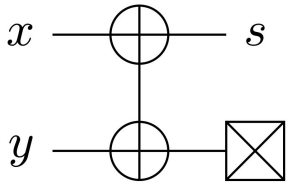
Dekker's add2 Algorithm

```
function add2(x0, x1, y0, y1)
    (s0, s1) := TwoSum(x1, y1)
    v := x1 ⊕ y1
    w := s1 ⊕ v
    (z0, z1) := TwoSum(s0, w)
    return (z0, z1)
end
```





$(s, e) := \mathsf{TwoSum}(x, y)$

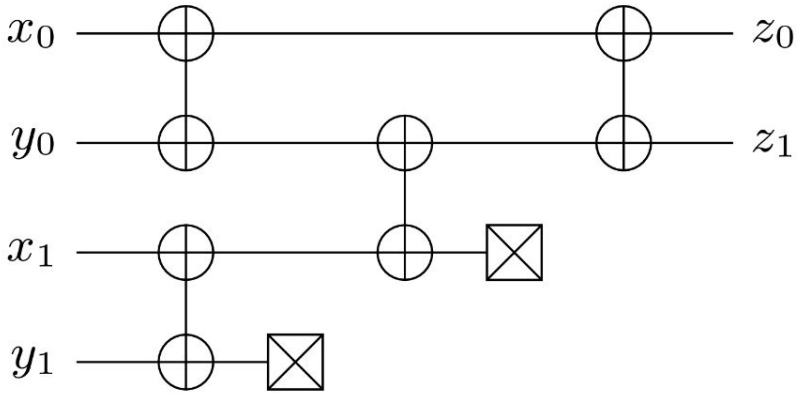


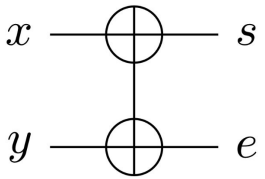
$s := x \oplus y$

Dekker's add2 Algorithm

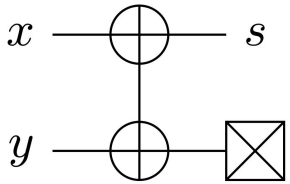
```
function add2(x0, x1, y0, y1)
    (s0, s1) := TwoSum(x1, y1)
    v :=
    w :=
    (z0,                    0, w)
    return (z0, z1)
end
```

Totally incorrect!
add2 can produce
relative error > 1.





$(s, e) := \mathsf{TwoSum}(x, y)$

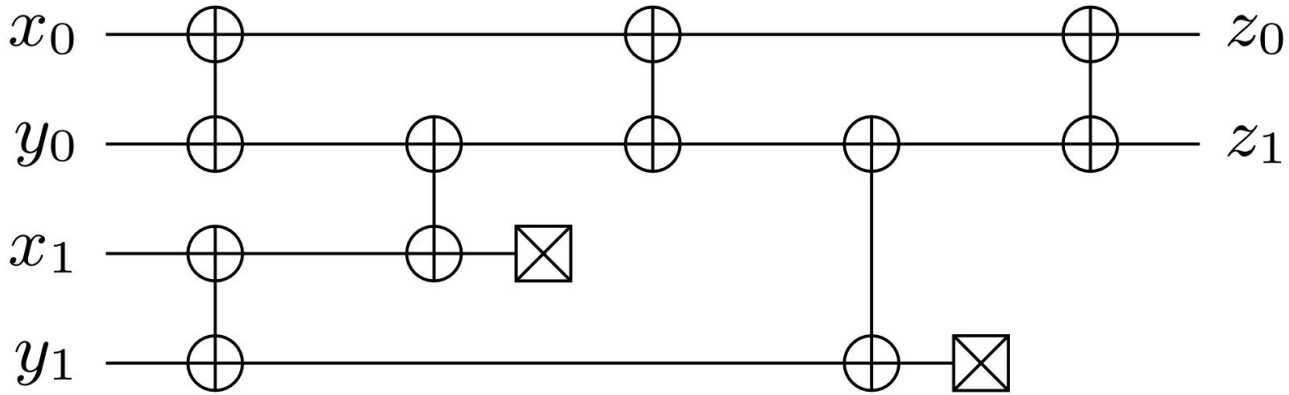


$s := x \oplus y$

Li et al.’s ddadd Algorithm

```
function ddadd(x0, x1, y0, y1)
    (s0, s1) := TwoSum(x0, y0)
    (t0, t1) := TwoSum(x1, y1)
    c := s1 ⊕ t0
    (v0, v1) := TwoSum(s0, c)
    w := t1 ⊕ v1
    (z0, z1) := TwoSum(v0, w)
    return (z1, z2)
end
```

Li et al.’s ddadd Algorithm



Is it correct?

A long story…

Correctness of ddadd

- Li et al. (2000) claimed this addition algorithm to have relative error $2 \cdot 2^{-106}$.
  - Recall that IEEE `double` has 53 bits of precision.
- Joldes et al. (2017) **refuted** this claim, giving a counterexample with relative error $2.25 \cdot 2^{-106}$, conjecturing that this is the upper bound.
- This claim is **also** wrong; I found a counterexample where the discarded terms have relative error $3 \cdot 2^{-106}$.

Proof. First, we exclude the straightforward case in which one of the operands is zero. We can also quickly proceed with the case $x_h + y_h = 0$: The returned result is $2\mathrm{Sum}(x_\ell, y_\ell)$, which is equal to $x + y$, that is, the computation is errorless. Now, without loss of generality, we assume $1 \le x_h < 2$, $1 \le x_h \le 2$

Define ϵ_1

and ϵ_2 as th

1. If $-x_h$

$s_h = x_h + y_h$

Define

We have $-x_h < y_h \le (1-\sigma) + \frac{x_h}{2}(\sigma - 2)$, so $0 \le x_h + y_h \le 1 + \sigma \cdot (\frac{x_h}{2} - 1) \le 1 - \sigma u$. Also, since x_h is a multiple of $2u$ and y_h is a multiple of σu, $s_h = x_h + y_h$ is a multiple of σu. Since s_h is nonzero, we finally obtain

We have $|x_\ell| \le u$ a

From Equation (6),
Equation (7), the fl
algorithm introduc

Equations (6) and (

so $|v_h| \le 1$ and $|v_\ell$

2. If $-x_h/2 < y_h \le x_h$

Notice that we have $x_h/2 < x_h + y_h \le 2x_h$, so $x_h/2 \le s_h \le 2x_h$. Also notice that we have $|x_\ell| \le u$.

- If $\frac{1}{2} < x_h + y_h \le 2 - 4u$. Define

$$\sigma = \begin{cases} 1 \text{ if } x_h + y_h \le 1 - 2u, \\ 2 \text{ if } 1 - 2u < x_h + y_h \le 2 - 4u. \end{cases}$$

We have

$$\frac{\sigma}{2}(1-2u) \le s_h \le \sigma(1-2u) \quad \text{and} \quad |s_\ell| \le \frac{\sigma}{2}u. \tag{11}$$

When $\sigma = 1$, we necessarily have $-x_h/2 < y_h < 0$, so $|y_\ell| \le u/2$. And when $\sigma = 2$, $|y_h| \le x_h \le 2 - 2u$ implies $|y_\ell| \le u$. Hence we always have $|y_\ell| \le \frac{\sigma}{2}u$. This implies $|x_\ell + y_\ell| \le (1 + \sigma/2)u$, therefore

$$|t_h| \le \left(1 + \frac{\sigma}{2}\right)u \quad \text{and} \quad |t_\ell| \le u^2. \tag{12}$$

Now, $|s_\ell + t_h| \le (1+\sigma)u$, so

$$|c| \le (1+\sigma)u \quad \text{and} \quad |\epsilon_1| \le \sigma u^2. \tag{13}$$

Since $s_h \ge 1/2$ and $|c| \le 3u$, if $p \ge 3$, then Algorithm Fast2Sum introduces no error at line 4 of the algorithm, that is,

Formalizing Floating-Point Proofs

- To prove the correctness of a floating-point accumulation network, you need:
  - **Error bound** on discarded terms ($|x_3| \leq 2^{-106} |x_0|$)
  - **Nonoverlapping invariant** on non-discarded terms ($|x_1| \leq 2^{-53} |x_0|$)

- Need to **rule out invalid outputs** assuming valid (nonoverlapping) inputs.

- In principle, a **SAT solver** can be used to prove these bounds…
  …but not before the heat death of the universe.

Formalizing Floating-Point Proofs

- We simplify the problem using our **SELTZO abstraction**.
  Instead of the full bitwise representation of x, consider only:
  - s_x: **sign** of x
  - e_x: **exponent** of x
  - nlz_x, nlo_x: number of **leading zeros/ones** in mantissa of x
  - ntz_x, nto_x: number of **trailing zeros/ones** in mantissa of x

- These variables suffice to express the bounds we're interested in.

  $$|y| \leq 2^{-106} |x| \quad \rightarrow \quad e_y < e_x - 106$$

SELTZO Lemmas

- **TwoSum** is well-behaved under the SELTZO abstraction.
  Knowing SELTZO **inputs**, we can bound possible SELTZO **outputs**.

Lemma SETZ-2AB0: If $s_x = s_y$, $e_x = f_y + p$, $f_x = e_y$, $e_x < f_x + (p-1)$, and $e_y < f_y + (p-1)$, then one of the following statements holds:

1. $s_s = s_x$, $e_s = e_x$, $e_x - (p-2) \leq f_s \leq e_y - 1$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
2. $s_s = s_x$, $e_s = e_x$, $f_s = e_y$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
3. $s_s = s_x$, $e_s = e_x$, $e_y + 1 \leq f_s \leq e_x - 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
4. $s_s = s_x$, $e_s = e_x + 1$, $e_x - (p-2) \leq f_s \leq e_y - 1$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
5. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_y$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
6. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma SETZ-2AB1: If $s_x = s_y$, $e_x = f_y + p$, $f_x = e_y$, and $e_x = f_x + (p-1)$, then one of the following statements holds:

1. $s_s = s_x$, $e_s = e_x$, $e_y + 1 \leq f_s \leq e_x - 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
2. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma SETZ-2AB2: If $s_x = s_y$, $e_x = f_y + p$, $f_x = e_y$, and $e_y = f_y + (p-1)$, then one of the following statements holds:

1. $s_s = s_x$, $e_s = e_x + 1$, $e_x - (p-2) \leq f_s \leq e_y - 1$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
2. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_y$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
3. $s_s = s_x$, $e_s = e_x + 1$, $f_s = e_x + 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma SETZ-3BC0: If $s_x \neq s_y$, $e_x = f_y + p$, $f_x = e_y$, $e_x > f_x + 1$, and $e_y > f_y + 1$, then one of the following statements holds:

1. $s_s = s_x$, $e_s = e_x - 1$, $f_s = f_y$, and $e = +0.0$.
2. $s_s = s_x$, $e_s = e_x$, $e_x - (p-2) \leq f_s \leq e_y - 1$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
3. $s_s = s_x$, $e_s = e_x$, $f_s = e_y$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
4. $s_s = s_x$, $e_s = e_x$, $e_y + 1 \leq f_s \leq e_x - 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma SETZ-3BC1: If $s_x \neq s_y$, $e_x = f_y + p$, $f_x = e_y$, and $e_y = f_y + 1$, then one of the following statements holds:

1. $s_s = s_x$, $e_s = e_x - 1$, $f_s = f_y$, and $e = +0.0$.
2. $s_s = s_x$, $e_s = e_x$, $e_y + 1 \leq f_s \leq e_x - 1$, $s_e = s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma SETZ-3CD0: If $s_x \neq s_y$, $e_x = f_y + p$, $f_x = e_y + 1$, $e_x > f_x$, and $e_y > f_y + 1$, then one of the following statements holds:

1. $s_s = s_x$, $e_s = e_x$, $e_x - (p-2) \leq f_s \leq e_y - 1$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
2. $s_s = s_x$, $e_s = e_x$, $f_s = e_y$, $s_e = s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.
3. $s_s = s_x$, $e_s = e_x$, $e_y + 2 \leq f_s \leq e_x$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Lemma SETZ-3CD1: If $s_x \neq s_y$, $e_x = f_y + p$, $f_x = e_y + 1$, and $e_y < f_y + 2$, then one of the following statements holds:

1. $s_s = s_x$, $e_s = e_x$, $e_y + 2 \leq f_s \leq e_x$, $s_e \neq s_y$, $f_y \leq e_e \leq e_x - p$, and $f_e = f_y$.

Correctness Proofs Using SELTZO

- We set up an **integer satisfiability problem** (i.e., integer linear program) in the **SELTZO variables** (s_x, e_x, nlz_x, nlo_x, ntz_x, nto_x).

- We impose three kinds of constraints:
  - **Consistency constraints** on the validity of (s_x, e_x, nlz_x, nlo_x, ntz_x, nto_x)
  - **Execution constraints** on the inputs and outputs of TwoSum
  - **Counterexample constraints** to describe the negation of the property we want

- If these constraints are **unsatisfiable**, then no counterexample exists.
  - If they are **satisfiable**, a counterexample **may** exist;
    but there may be no concrete values that realize an abstract counterexample.

Example: Consistency Constraints

1. The sign bit must be zero or one, and the exponent must be bounded below.

$$(s_v = 0) \vee (s_v = 1) \qquad e_v \geq e_{\min} - 1 \tag{9}$$

2. If a floating-point variable is zero (i.e., $e_v = e_{\min} - 1$), then its mantissa must consist entirely of zeros.

$$(e_v = e_{\min} - 1) \rightarrow [(\mathsf{nlz}_v = \mathsf{ntz}_v = p - 1) \wedge (\mathsf{nlo}_v = \mathsf{nto}_v = 0)] \tag{10}$$

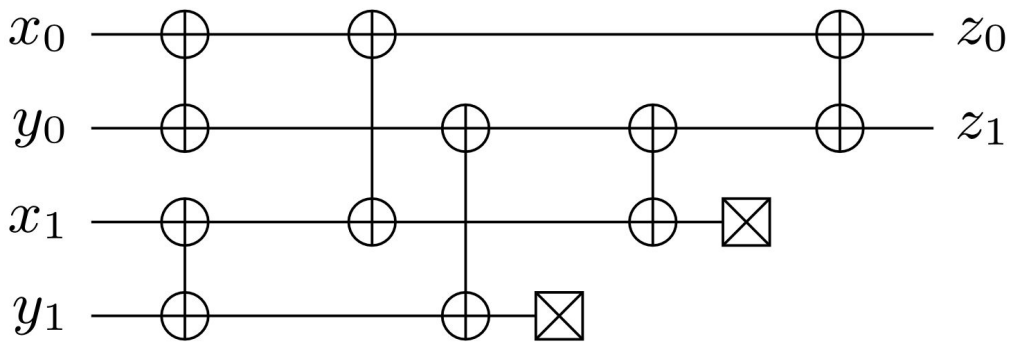
3. The leading and trailing bits of the mantissa are either 0 or 1.

$$[(\mathsf{nlz}_v > 0) \wedge (\mathsf{nlo}_v = 0)] \vee [(\mathsf{nlz}_v = 0) \wedge (\mathsf{nlo}_v > 0)] \tag{11}$$
$$[(\mathsf{ntz}_v > 0) \wedge (\mathsf{nto}_v = 0)] \vee [(\mathsf{ntz}_v = 0) \wedge (\mathsf{nto}_v > 0)] \tag{12}$$

4. The number of leading and trailing bits must be bounded by $p-1$, the width of the mantissa.

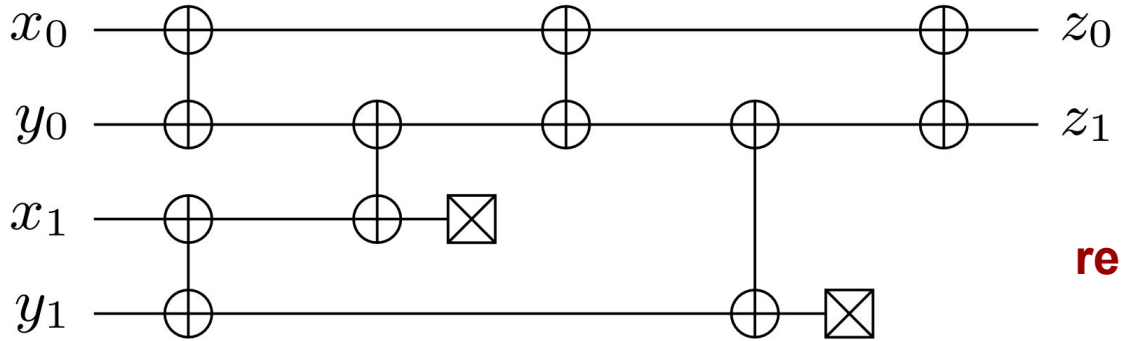
$$(\mathsf{nlz}_v = \mathsf{ntz}_v = p - 1) \vee (\mathsf{nlz}_v + \mathsf{ntz}_v < p - 1) \tag{13}$$
$$(\mathsf{nlo}_v = \mathsf{nto}_v = p - 1) \vee (\mathsf{nlo}_v + \mathsf{nto}_v < p - 1) \tag{14}$$
$$(\mathsf{nlz}_v + \mathsf{nto}_v = p - 1) \vee (\mathsf{nlz}_v + \mathsf{nto}_v < p - 2) \tag{15}$$
$$(\mathsf{ntz}_v + \mathsf{nlo}_v = p - 1) \vee (\mathsf{ntz}_v + \mathsf{nlo}_v < p - 2) \tag{16}$$

madd - More Accurate Double-Double Addition



relative error $2 \cdot 2^{-106}$

more accurate
than ddadd using
lower depth.



relative error $3 \cdot 2^{-106}$

Efficient Verification

FPAN	Format	Z3	CVC5	MathSAT	Bitwuzla	Colibri2	SELTZO
ddadd	binary16	DNF	153 min	DNF	72 min	N/A	0.927 sec
madd	binary16	DNF	120 min	3898 min	72 min	N/A	0.713 sec
ddadd	bfloat16	DNF	704 min	DNF	71 min	N/A	0.838 sec
madd	bfloat16	DNF	946 min	DNF	99 min	N/A	0.689 sec
ddadd	binary32	DNF	1088 min	DNF	640 min	N/A	0.774 sec
madd	binary32	DNF	1019 min	DNF	518 min	N/A	0.722 sec
ddadd	binary64	DNF	DNF	DNF	DNF	N/A	0.623 sec
madd	binary64	DNF	DNF	DNF	DNF	N/A	0.923 sec
ddadd	binary128	DNF	DNF	DNF	DNF	N/A	0.880 sec
madd	binary128	DNF	DNF	DNF	DNF	N/A	0.991 sec

Simpler Abstractions

- Are all six of the variables (s_x, e_x, nlz_x, nlo_x, ntz_x, nto_x) **necessary**?
- What if we try using only (s_x, e_x) or (s_x, e_x, ntz_x,)?

FPAN	SE	SETZ	SELTZO
ddadd	$2^{-(2p-7)} = 128\mathbf{u}^2$	$2^{-(2p-4)} = 16\mathbf{u}^2$	$2^{-(2p-2)} = 4\mathbf{u}^2$
madd	$2^{-(2p-6)} = 64\mathbf{u}^2$	$2^{-(2p-3)} = 8\mathbf{u}^2$	$2^{-(2p-1)} = 2\mathbf{u}^2$

- We still get **nontrivial** bounds, but not **precise to the last bit**.