Hardware implementation of linear algebra operators for small floating point formats

Orégane Desrentes

1. Kalray
2. Floating point linear algebra
3. 8 bits formats

Kalray

A complete offer or data-intensive applications

A UNIFIED VALUE PROPOSITION

PCIe Cards

Reference Solutions

KALRAY

THE POWER OF MORE

MPPA® DPU
Manycore Processors

pixstor™ beyond storage | ngenea® beyond cloud

Data Management & Storage
Software Solutions

A complete offer or data-intensive applications

Two application domains:

Data Center acceleration

Computing

A complete offer or data-intensive applications

Two application domains:

Data Center acceleration

- Compression and decompression
- Encryption and decryption
- Erasure coding
- De-duplication

Computing

A complete offer or data-intensive applications

Two application domains:

Data Center acceleration

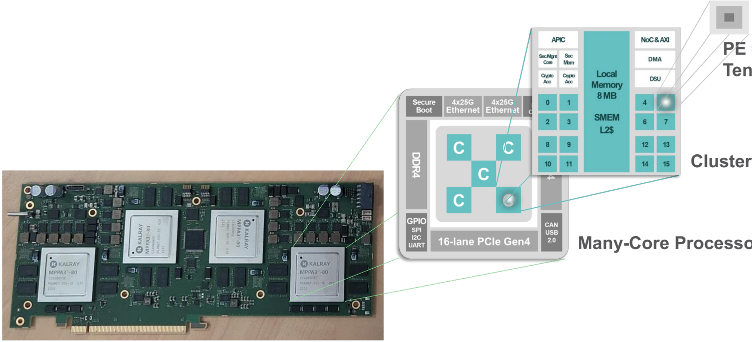
- Compression and decompression
- Encryption and decryption
- Erasure coding
- De-duplication

Computing

- Machine Learning
- Computer vision
- Pre/post processing
- Signal processing

Kalray MPPA® scalable many-core architecture

3rd-gen MPPA® processor: in TSMC 16nm technology, up to 1.2 GHz



PE = VLIW Core + Tensor Coprocessor

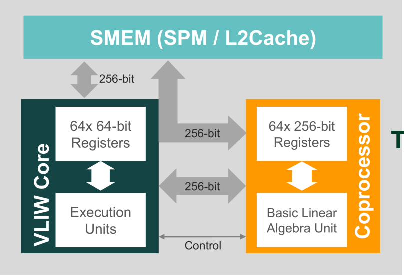
Cluster of 16 PEs

Many-Core Processor

Multiple Processors per Card

MPPA3 V2 Coolidge™ processing element (PE)

6-issue 64-bit VLIW core with a tightly-coupled tensor coprocessor



VLIW Core

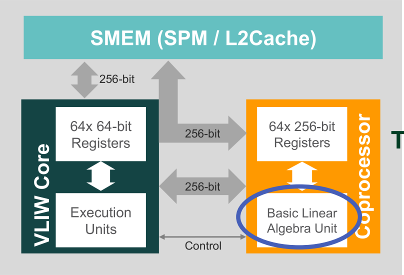
- Scalar 32-bit and 64-bit INT & FP
- 8× 8-bit, 4× 16-bit, 2× 32-bit SIMD
- 128-bit 256-bit SIMD operations by bundling multiple instructions
- 256-bit load/store unit with masking

Tensor Coprocessor

- Matrix multiply-add on 4 × 4 tiles
- 512-bit multiply and add operands
- Matrix zip/unzip & transpose
- 256-bit load/store unit with masking
- Blocks of 256-bit registers used as circular buffer or as lookup table

MPPA3 V2 Coolidge™ processing element (PE)

6-issue 64-bit VLIW core with a tightly-coupled tensor coprocessor



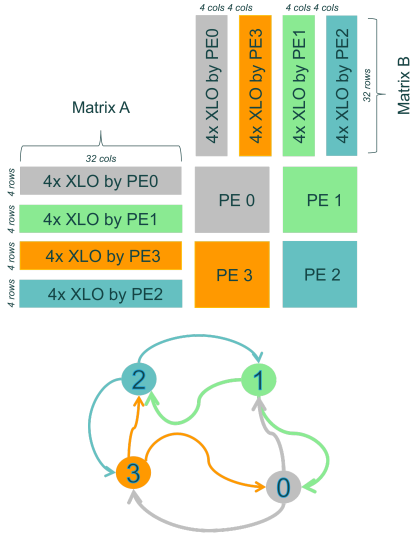
VLIW Core

- Scalar 32-bit and 64-bit INT & FP
- 8× 8-bit, 4× 16-bit, 2× 32-bit SIMD
- 128-bit 256-bit SIMD operations by bundling multiple instructions
- 256-bit load/store unit with masking

Tensor Coprocessor

- Matrix multiply-add on 4×4 tiles
- 512-bit multiply and add operands
- Matrix zip/unzip & transpose
- 256-bit load/store unit with masking
- Blocks of 256-bit registers used as circular buffer or as lookup table

Coprocessor collective tensor operations



- PE operation: INT8.32 $(4 \times 16) \cdot (16 \times 4) + = (4 \times 4)$
- Macro-scheme executed by 4 PEs
    - $8\times$ 256-bit memory loads (XLO) per PE
    - $8\times$ 256-bit data exchanges per PE
    - $8\times$ matrix multiply-add operations per PE
- Matrix A and B are loaded by quarter by each PE which exchange one quarter with 2 different PEs
- Kernel for INT8.32: $(16 \times 32) \cdot (32 \times 16) + = (16 \times 16)$

Coprocessor Matrix multiply-accumulate operations

8-bit to 32-bit int matrix multiply-add: $(4 \times 16)\mathrm{int8} \cdot (16 \times 4)\mathrm{int8} += (4 \times 4)\mathrm{int32}$

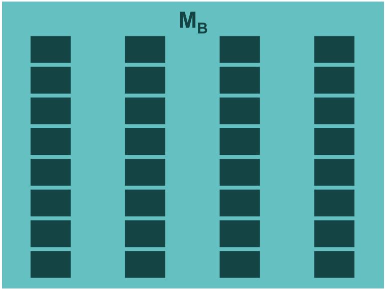
16-bit to 32-bit FP matrix multiply-add: $(4 \times 8)\mathrm{FP16} \cdot (8 \times 4)\mathrm{FP16} += (4 \times 4)\mathrm{FP32}$

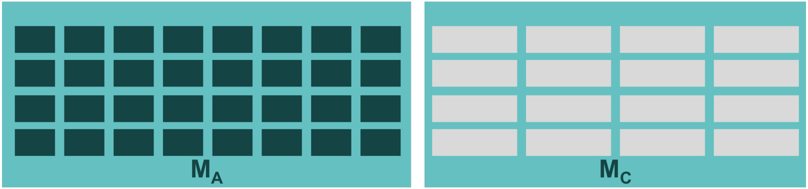
Signature

- 512-bit × 512-bit += 512-bit
- 256-bit register-pair multiplicands
- 256-bit register-pair accumulator

Performances

- 256 MADD eq. per cycle, 512 ops/c
- 128 FMA eq. per cycle, 256 flops/c
- 50 TOPS @1.2 GHz for 80 cores
- 25 TFLOPS @1.2 GHz for 80 core





Floating point linear algebra

Floating-Point numbers

- Computer representation for real numbers

Floating-Point numbers

- Computer representation for real numbers
- $\mathbb{R}$ is infinite in range and precision

Floating-Point numbers

- Computer representation for real numbers
- $\mathbb{R}$ is infinite in range and precision
- A computer is finite, in base 2

Floating-Point numbers

- Computer representation for real numbers
- $\mathbb{R}$ is infinite in range and precision
- A computer is finite, in base 2

$$(-1)^S \times 2^E \times 1.F$$

S, E, F are stored in binary, in a finite format.

Floating-Point numbers

- Computer representation for real numbers
- $\mathbb{R}$ is infinite in range and precision
- A computer is finite, in base 2

$$(-1)^S \times 2^E \times 1.F$$

S, E, F are stored in binary, in a finite format.

For floats: $w_S = 1, w_E = 8, w_F = 23$

Floating-Point special values

- $+0, -0$

Floating-Point special values

- $+0, -0$
  - $S =$ sign
  - $E = 0$
  - $F = 0$

Floating-Point special values

- $+0, -0$
  - $S =$ sign
  - $E = 0$
  - $F = 0$
- $+\infty, -\infty$

Floating-Point special values

- $+0, -0$
  - $S = \text{sign}$
  - $E = 0$
  - $F = 0$
- $+\infty, -\infty$
  - $S = \text{sign}$
  - $E = 2^{w_E} - 1$ (max number)
  - $F = 0$

Floating-Point special values

- $+0, -0$
    - $S = \text{sign}$
    - $E = 0$
    - $F = 0$
- $+\infty, -\infty$
    - $S = \text{sign}$
    - $E = 2^{w_E} - 1$ (max number)
    - $F = 0$
- NaN

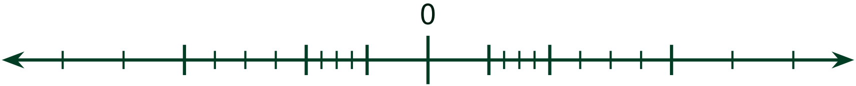
Floating-Point special values

- $+0, -0$
  - $S = \text{sign}$
  - $E = 0$
  - $F = 0$
- $+\infty, -\infty$
  - $S = \text{sign}$
  - $E = 2^{w_E} - 1$ (max number)
  - $F = 0$
- NaN ($2^{w_F+1} - 2$)
  - $S = \text{any}$
  - $E = 2^{w_E} - 1$
  - $F \neq 0$

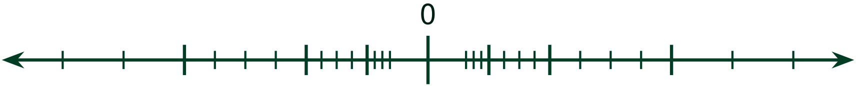
Floating-Point special values

- $+0, -0$
    - $S = \text{sign}$
    - $E = 0$
    - $F = 0$
- $+\infty, -\infty$
    - $S = \text{sign}$
    - $E = 2^{w_E} - 1$ (max number)
    - $F = 0$
- NaN $(2^{w_F+1} - 2)$
    - $S = \text{any}$
    - $E = 2^{w_E} - 1$
    - $F \neq 0$

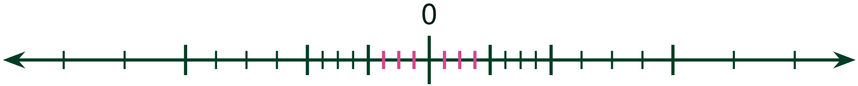
Subnormals



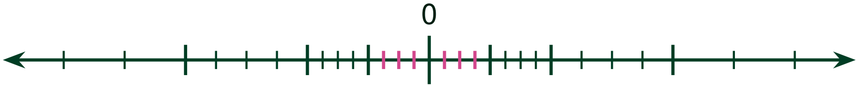
Subnormals



Subnormals



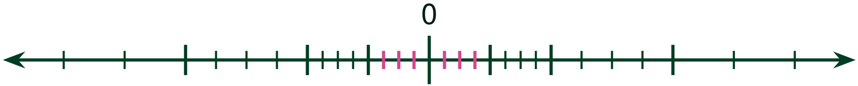
Subnormals



Why we need them

Avoid numbers being flushed to 0 abruptly: gradual underflow.
Guarantees if $x \neq y$ then $x - y \neq 0$ and other useful properties.

Subnormals



Why we need them

Avoid numbers being flushed to 0 abruptly: gradual underflow.
Guarantees if $x \neq y$ then $x - y \neq 0$ and other useful properties.

Why they are annoying

They have less precision than w_F.
They are encoded differently: $(-1)^S \times 2^E \times 0.F$
Where the first significant bit ? $0.F = 0.000000011001100$

$\Rightarrow$ They used to be treated in micro-code but now we do subnormal hardware

Short and biased history of floating point units

Short and biased history of floating point units

- In the 70s, adders and multipliers
  - $R = \circ(X + Y)$
  - $R = \circ(X \times Y)$

Short and biased history of floating point units

Short and biased history of floating point units

- In the 70s, adders and multipliers
- In 1985, IEEE-754 standard normalises the rounding $\circ(\dots)$

Short and biased history of floating point units

Short and biased history of floating point units

- In the 70s, adders and multipliers
- In 1985, IEEE-754 standard normalises the rounding $\circ(\dots)$
- In the 90s, FMA (fused multiply add): $R = \circ(X \times Y + Z)$
  - two operations in one instruction: *faster*
  - one single rounding: *more accurate*

Short and biased history of floating point units

Short and biased history of floating point units

- In the 70s, adders and multipliers
- In 1985, IEEE-754 standard normalises the rounding $\circ(\dots)$
- In the 90s, FMA (fused multiply add): $R = \circ(X \times Y + Z)$
- These days: $R \approx Z + \sum_{i=0}^{N-1} X_i \times Y_i$

Short and biased history of floating point units

Short and biased history of floating point units

- In the 70s, adders and multipliers
- In 1985, IEEE-754 standard normalises the rounding $\circ(\dots)$
- In the 90s, FMA (fused multiply add): $R = \circ(X \times Y + Z)$
- These days: $R = \circ\left(Z + \sum_{i=0}^{N-1} X_i \times Y_i\right)$

Short and biased history of floating point units

Short and biased history of floating point units

- In the 70s, adders and multipliers
- In 1985, IEEE-754 standard normalises the rounding $\circ(\dots)$
- In the 90s, FMA (fused multiply add): $R = \circ(X \times Y + Z)$
- These days: $R = \circ\left(Z + \sum_{i=0}^{N-1} X_i \times Y_i\right)$

Multiple applications

- Matrix multiplication (neural networks, graphical applications, scientific computing,…)
- Complex arithmetic (FFT,…)

Short and biased history of floating point units

Short and biased history of floating point units

- In the 70s, adders and multipliers
- In 1985, IEEE-754 standard normalises the rounding $\circ(\dots)$
- In the 90s, FMA (fused multiply add): $R = \circ(X \times Y + Z)$
- These days: $R = \circ\left(Z + \sum_{i=0}^{N-1} X_i \times Y_i\right)$

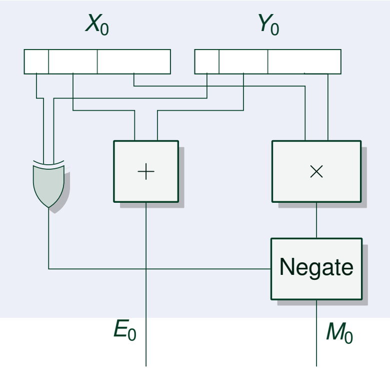
Multiple applications

- Matrix multiplication (neural networks, graphical applications, scientific computing,. . .)
- Complex arithmetic (FFT,. . .)

Objective: better than FMA chains

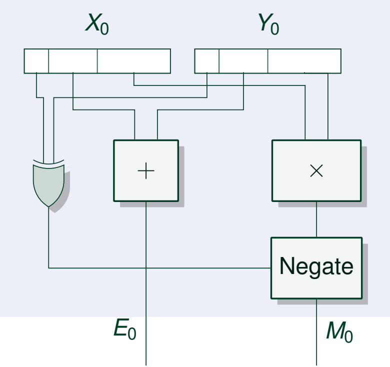
$$Z + \sum_{i=0}^{N-1} X_i \times Y_i \approx \circ(\dots \circ(\circ(Z + X_0 \times Y_0) + X_1 \times Y_1) \dots + X_N \times Y_N)$$

Product of floating point numbers



Easy, but the result is not a IEEE-754 floating point number:

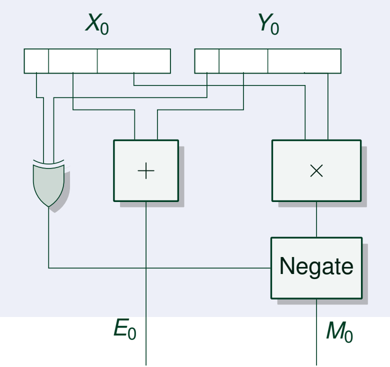
Product of floating point numbers



Easy, but the result is not a IEEE-754 floating point number:

- Not rounded
- Signed significands

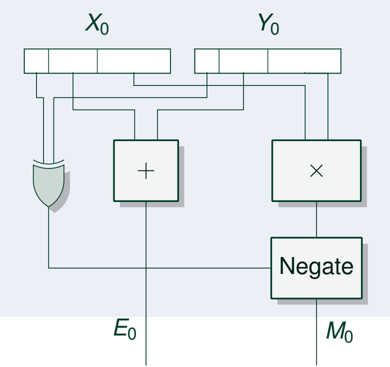
Product of floating point numbers



Easy, but the result is not a IEEE-754 floating point number:

- Not rounded
- Signed significands
- Significands are not normalised

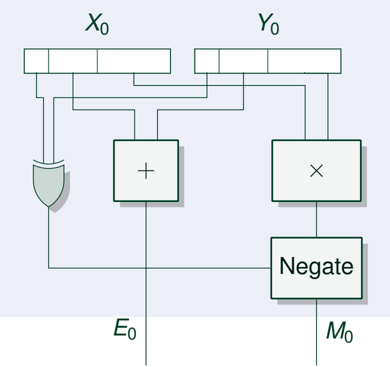
Product of floating point numbers



Easy, but the result is not a IEEE-754 floating point number:

- Not rounded
- Signed significands
- Significands are not normalised
  - Float is in $[1, 2[$ but product is in $[1, 4[$

Product of floating point numbers



Easy, but the result is not a IEEE-754 floating point number:

- Not rounded
- Signed significands
- Significands are not normalised
  - Float is in $[1, 2[$ but product is in $[1, 4[$
  - Product of normal and subnormal

Round a floating point number

Let's consider a decimal float with 5 digits precision.
Round by adding $\frac{1}{2}$ ulp (Unit in the Last Place) and truncating.

$\pi \approx 3.1415927\ldots$

Round a floating point number

Let’s consider a decimal float with 5 digits precision.
Round by adding $\frac{1}{2}$ ulp (Unit in the Last Place) and truncating.

$\pi \approx 3.1415927\ldots$

$$\begin{array}{rl} & \mathbf{3}.\mathbf{1415}927 \\ + & 0.00005 \\ \hline & \mathbf{3}.\mathbf{1416}427 \end{array}$$

Round a floating point number

Let's consider a decimal float with 5 digits precision.
Round by adding $\frac{1}{2}$ ulp (Unit in the Last Place) and truncating.

$\pi \approx 3.1415927\ldots$

$$\begin{array}{rl} & \mathbf{3.1415}927 \\ + & 0.00005 \\ \hline & \mathbf{3.1416} \end{array}$$

Round a floating point number

Let's consider a decimal float with 5 digits precision.
Round by adding $\frac{1}{2}$ ulp (Unit in the Last Place) and truncating.

$\pi \approx 3.1415927\ldots$

$x = -15416500$

$$
\begin{array}{rl}
 & \mathbf{3}.\mathbf{1415}927 \\
+ & 0.00005 \\
\hline
 & \mathbf{3}.\mathbf{1416}
\end{array}
$$

Round a floating point number

Let's consider a decimal float with 5 digits precision.
Round by adding $\frac{1}{2}$ ulp (Unit in the Last Place) and truncating.

$\pi \approx 3.1415927\ldots$

$$\begin{array}{rl} & \mathbf{3}.\mathbf{1415}927 \\ + & 0.00005 \\ \hline & \mathbf{3}.\mathbf{1416} \end{array}$$

$x = -15416500$

$$\begin{array}{rl} & \mathbf{1}.\mathbf{5416}500 \\ + & 0.00005 \\ \hline & \mathbf{1}.\mathbf{5417}000 \end{array}$$

Round a floating point number

Let's consider a decimal float with 5 digits precision.
Round by adding $\frac{1}{2}$ ulp (Unit in the Last Place) and truncating.

$\pi \approx 3.1415927\ldots$

$$\begin{array}{rl} & \mathbf{3}.\mathbf{1415}927 \\ + & 0.00005 \\ \hline & \mathbf{3}.\mathbf{1416} \end{array}$$

$x = -15416500$

$$\begin{array}{rl} & \mathbf{1}.\mathbf{5416}500 \\ + & 0.00005 \\ \hline & \mathbf{1}.\mathbf{5417} \end{array}$$

Round a floating point number

Let's consider a decimal float with 5 digits precision.
Round by adding $\frac{1}{2}$ ulp (Unit in the Last Place) and truncating.

$\pi \approx 3.1415927\ldots$

$$\begin{array}{rl} & \mathbf{3}.\mathbf{1415}927 \\ + & 0.00005 \\ \hline & \mathbf{3}.\mathbf{1416}427 \end{array}$$

$x = -15416500$

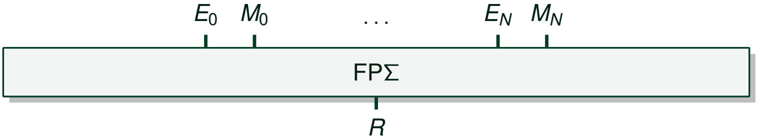
$$\begin{array}{rl} & \mathbf{1}.\mathbf{5416}500 \\ + & 0.00005 \\ \hline & \mathbf{1}.\mathbf{5417}000 \end{array}$$

If exactly in the middle, round to an even float.
$\Rightarrow$ We need the following information to round:

$$\underbrace{3.14159}_{\text{goes in } \circ(\pi)}\underset{\uparrow}{2}\underbrace{654}_{\text{boolean information: is this 0 ? "sticky bit"}}$$

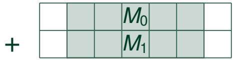
boolean information: is this digit over or under 5 ? "round bit"

Sum of floating point numbers



Rounding the sum of two floating point numbers

We sort $(E_0, M_0), (E_1, M_1)$ such that $E_0 \geq E_1$

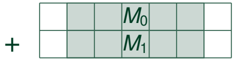


$E_0 = E_1$

Sum like integers

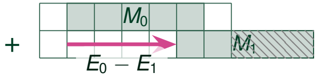
Rounding the sum of two floating point numbers

We sort $(E_0, M_0), (E_1, M_1)$ such that $E_0 \geq E_1$



$E_0 = E_1$

Sum like integers

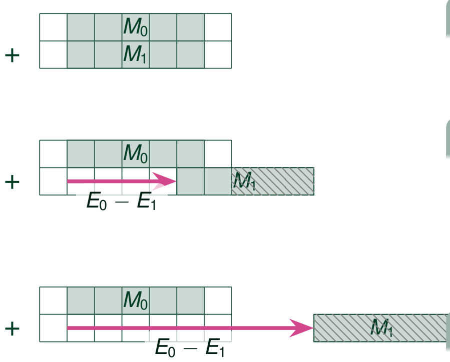


$E_0 \simeq E_1$

Part of (shifted) M_1 is added to M_0, and the rest is compressed in a "sticky bit"

Rounding the sum of two floating point numbers

We sort $(E_0, M_0), (E_1, M_1)$ such that $E_0 \geq E_1$



$E_0 = E_1$

Sum like integers

$E_0 \simeq E_1$

Part of (shifted) M_1 is added to M_0, and the rest is compressed in a "sticky bit"

$E_0 \gg E_1$

M_1 is completely compressed in a "sticky bit"

Rounding the sum of more than two floating point numbers - problem !!

Problem: cancellation

$M_0 = -M_1$ and $E_0 \gg E_2$

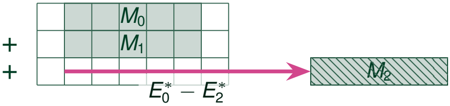
$M_0 + M_1 + M_2 = M_2$

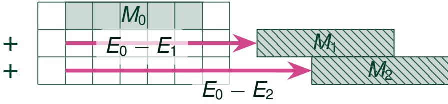
If M_2 has been totally compressed in a sticky bit, we cannot retrieve the result.

Problem: multi-sticky

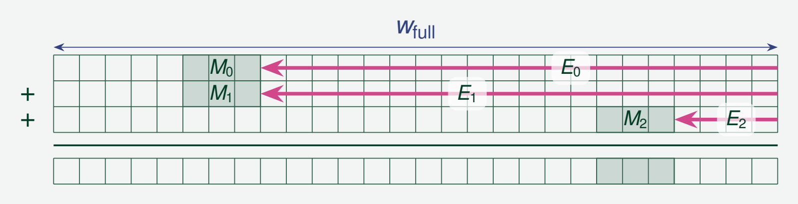
$E_0 \gg E_1$ and $E_0 \gg E_2$

If M_1 et M_2 were compressed, we cannot round the result M_0





An easy but expensive solution: the Kulisch accumulator[1]

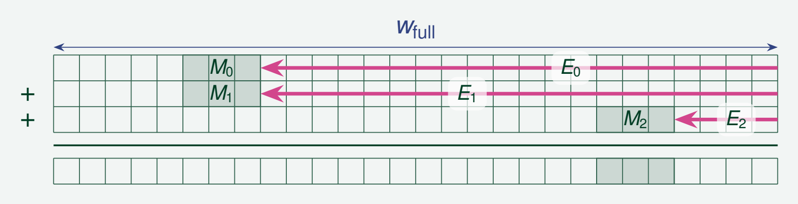


- Method:
  - Convert to fixpoint
  - Sum like integer
  - Use Leading Zero Count to get final exponent and significand

---

[1] U. W. Kulisch, "Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units", 2002

An easy but expensive solution: the Kulisch accumulator[1]

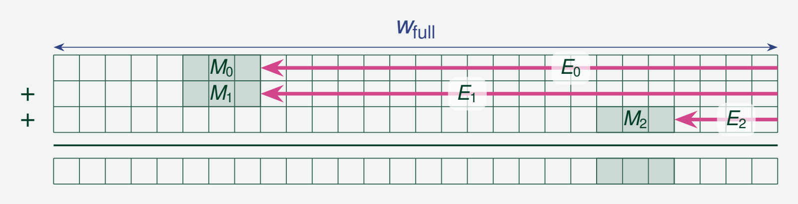


- Method:
    - Convert to fixpoint
    - Sum like integer
    - Use Leading Zero Count to get final exponent and significand
- $w_{full} = 2^{w_e} + w_f - 1$

---

[1] U. W. Kulisch, "Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units", 2002

An easy but expensive solution: the Kulisch accumulator[1]

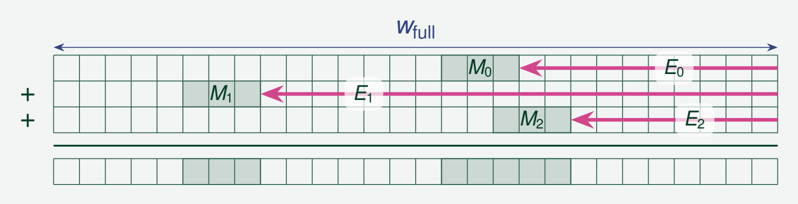


- Method:
  - Convert to fixpoint
  - Sum like integer
  - Use Leading Zero Count to get final exponent and significand
- $w_{full} = 2^{w_e} + w_f - 1$
- Good when w_e is small (or for a format with little range like Int or Posit when $es \leq 2$)

---

[1] U. W. Kulisch, "Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units", 2002

An easy but expensive solution: the Kulisch accumulator[1]

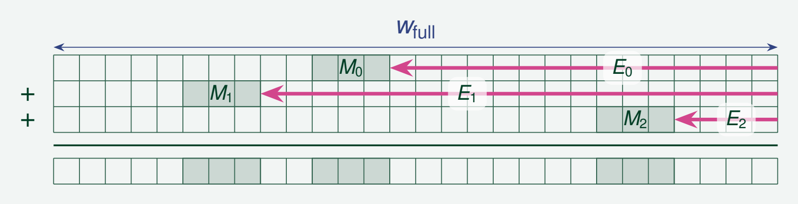


- Method:
    - Convert to fixpoint
    - Sum like integer
    - Use Leading Zero Count to get final exponent and significand
- $w_{full} = 2^{w_e} + w_f - 1$
- Good when w_e is small (or for a format with little range like Int or Posit when $es \leq 2$)

---

[1] U. W. Kulisch, "Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units", 2002

An easy but expensive solution: the Kulisch accumulator[1]



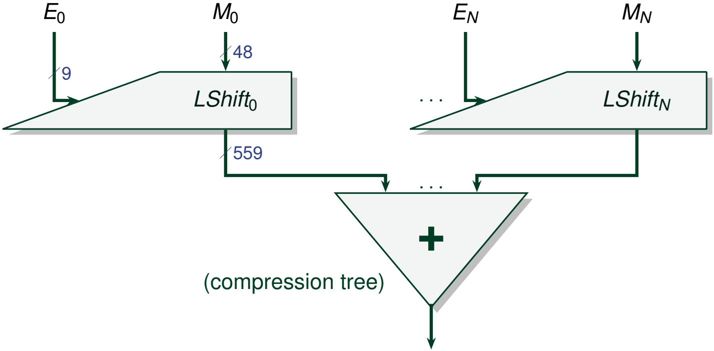
- Method:
  - Convert to fixpoint
  - Sum like integer
  - Use Leading Zero Count to get final exponent and significand
- $w_{full} = 2^{w_e} + w_f - 1$
- Good when w_e is small (or for a format with little range like Int or Posit when $es \leq 2$)

---

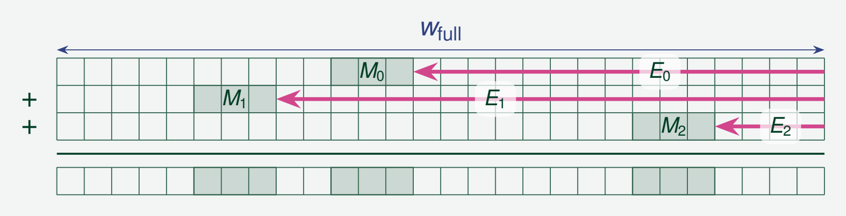
[1] U. W. Kulisch, "Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units", 2002

Kulisch accumulator: Architecture

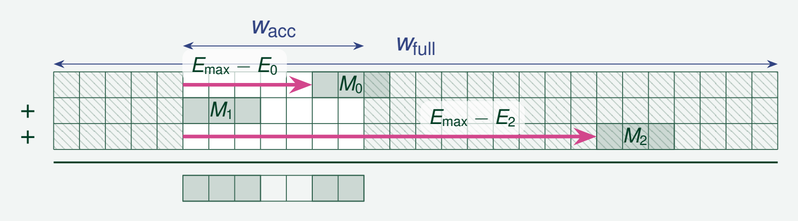
Size in bits for FP32 dot-product



Kulisch accumulator: Expensive and empty

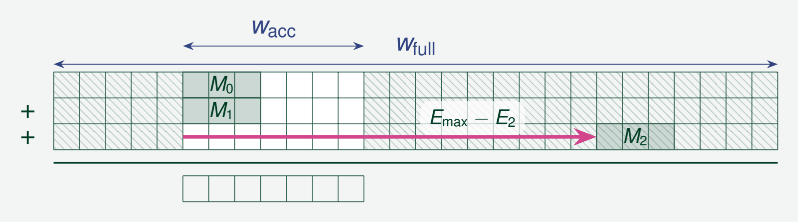


A less expensive version: truncated Kulisch



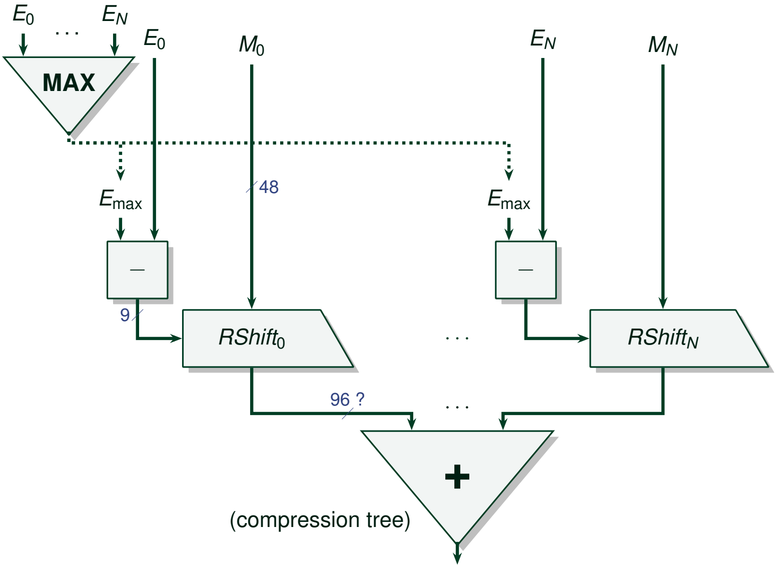
- Method similar to floating point sum:
    - Choose $w_{acc} < w_{full}$ (arbitrarily)
    - Align all numbers on the biggest one, throw away any bits that don't fill in w_{acc} (no sticky, we don't care)
    - Sum like integer, round to float

A less expensive version: truncated Kulisch



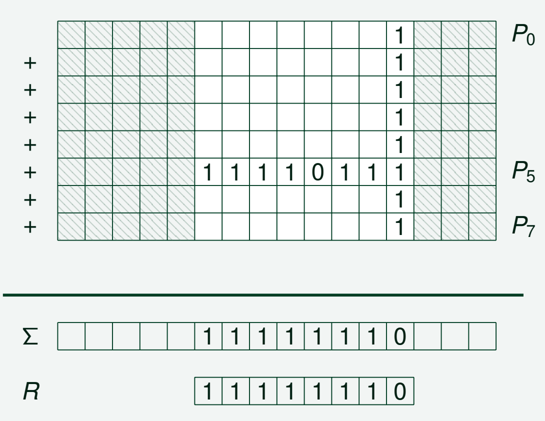
- Method similar to floating point sum:
  - Choose $w_{acc} < w_{full}$ (arbitrarily)
  - Align all numbers on the biggest one, throw away any bits that don't fill in w_{acc} (no sticky, we don't care)
  - Sum like integer, round to float
- Inexact computation

Truncated Kulisch accumulator: Architecture

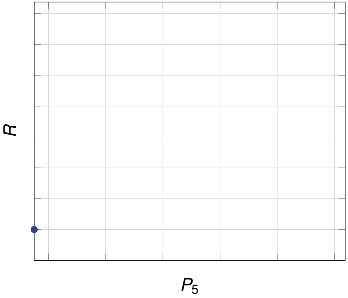


Non monotonicity of truncated Kulisch

Recently presented in detail by Mantas Mikaitis.

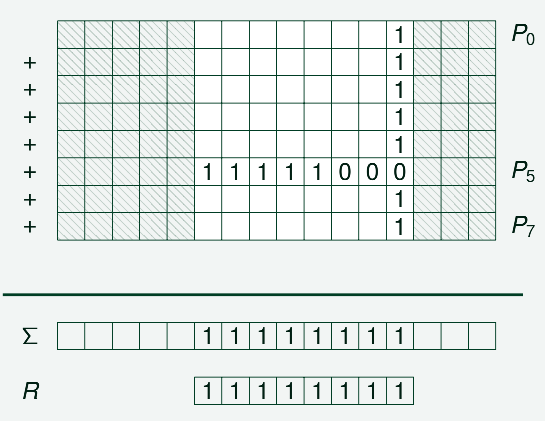


	▨	▨	▨	▨	▨								1	▨	▨	▨	P_0
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨	1	1	1	1	0	1	1	1	▨	▨	▨	P_5
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	P_7
Σ						1	1	1	1	1	1	1	0				
R						1	1	1	1	1	1	1	0				

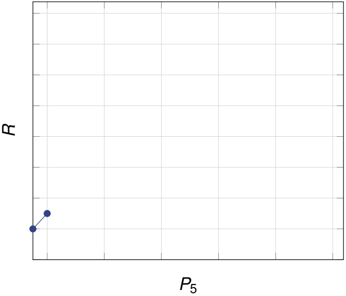


Non monotonicity of truncated Kulisch

Recently presented in detail by Mantas Mikaitis.

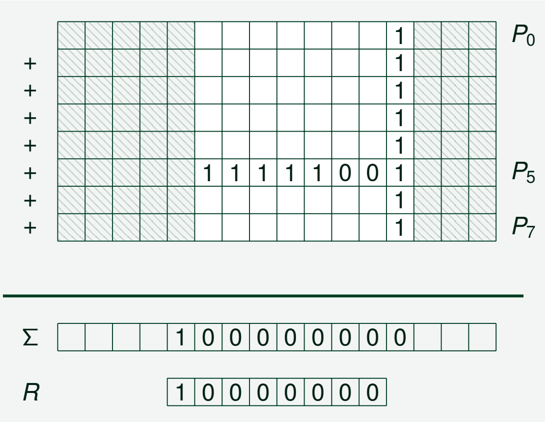


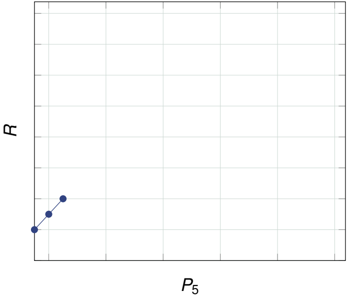
	▨	▨	▨	▨	▨								1	▨	▨	▨	P_0
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨	1	1	1	1	1	0	0	0	▨	▨	▨	P_5
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	P_7
Σ						1	1	1	1	1	1	1	1				
R						1	1	1	1	1	1	1	1				



Non monotonicity of truncated Kulisch

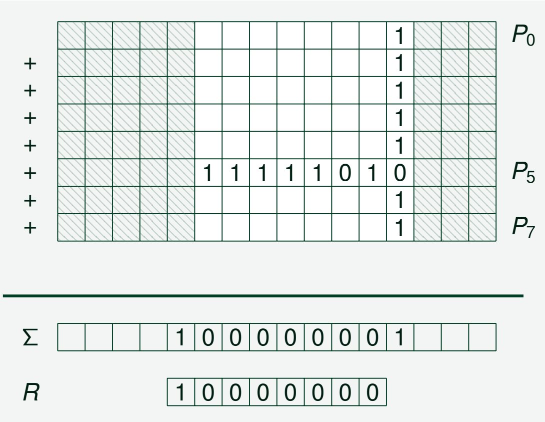
Recently presented in detail by Mantas Mikaitis.



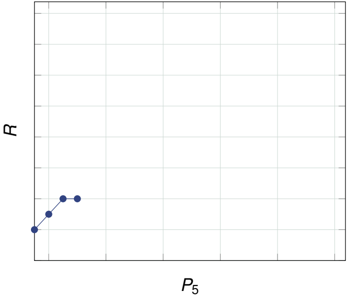


Non monotonicity of truncated Kulisch

Recently presented in detail by Mantas Mikaitis.

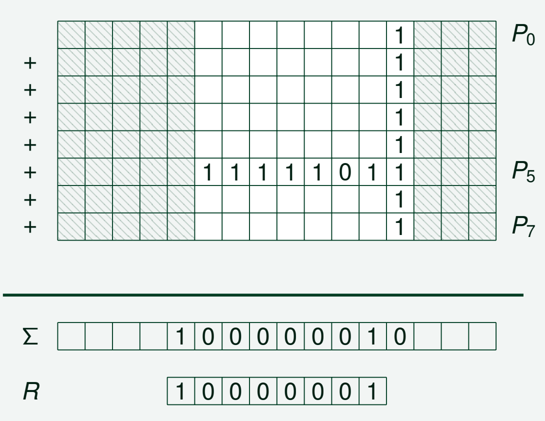


	▨	▨	▨	▨	▨								1	▨	▨	▨	P_0
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨	1	1	1	1	1	0	1	0	▨	▨	▨	P_5
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	P_7
Σ					1	0	0	0	0	0	0	0	1				
R					1	0	0	0	0	0	0	0					

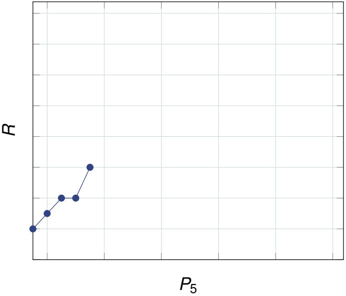


Non monotonicity of truncated Kulisch

Recently presented in detail by Mantas Mikaitis.



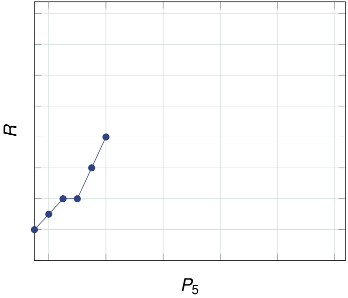
	▨	▨	▨	▨	▨								1	▨	▨	▨	P_0
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨	1	1	1	1	1	0	1	1	▨	▨	▨	P_5
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	P_7
Σ					1	0	0	0	0	0	0	1	0				
R					1	0	0	0	0	0	0	1					



Non monotonicity of truncated Kulisch

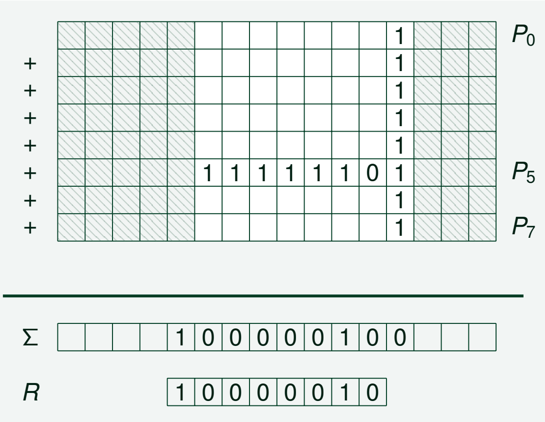
Recently presented in detail by Mantas Mikaitis.

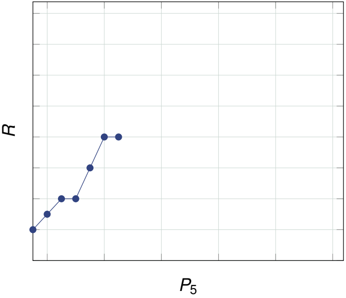
	▨	▨	▨	▨	▨								1	▨	▨	▨	P_0
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨	1	1	1	1	1	1	0	0	▨	▨	▨	P_5
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	P_7
Σ					1	0	0	0	0	0	0	1	1				
R					1	0	0	0	0	0	1	0					



Non monotonicity of truncated Kulisch

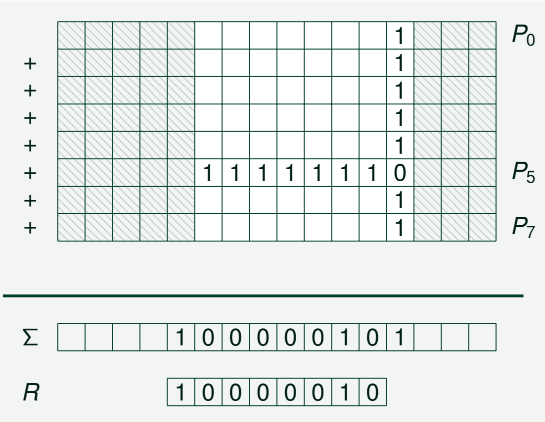
Recently presented in detail by Mantas Mikaitis.



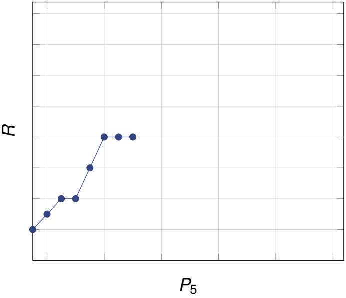


Non monotonicity of truncated Kulisch

Recently presented in detail by Mantas Mikaitis.

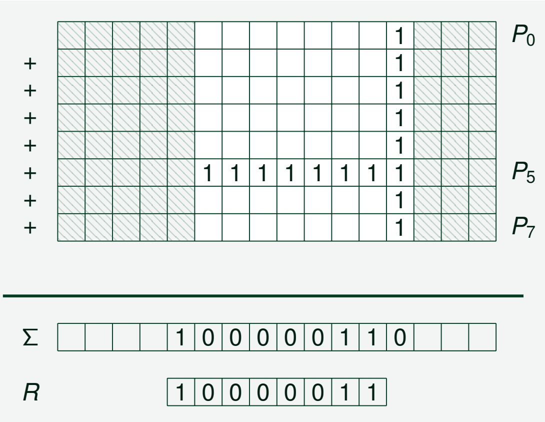


	▨	▨	▨	▨	▨								1	▨	▨	▨	P_0
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨	1	1	1	1	1	1	1	0	▨	▨	▨	P_5
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	P_7
Σ					1	0	0	0	0	0	1	0	1				
R					1	0	0	0	0	0	1	0					

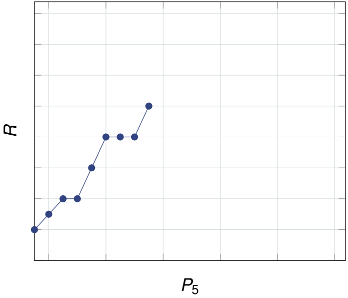


Non monotonicity of truncated Kulisch

Recently presented in detail by Mantas Mikaitis.

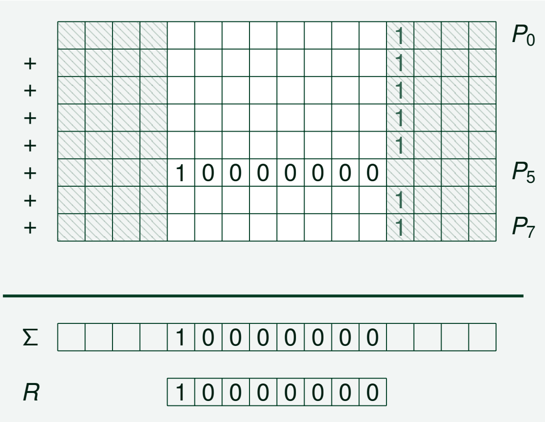


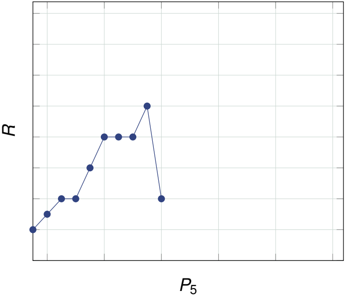
	▨	▨	▨	▨	▨								1	▨	▨	▨	P_0
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨	1	1	1	1	1	1	1	1	▨	▨	▨	P_5
+	▨	▨	▨	▨	▨								1	▨	▨	▨	
+	▨	▨	▨	▨	▨								1	▨	▨	▨	P_7
Σ					1	0	0	0	0	0	1	1	0				
R					1	0	0	0	0	0	1	1					



Non monotonicity of truncated Kulisch

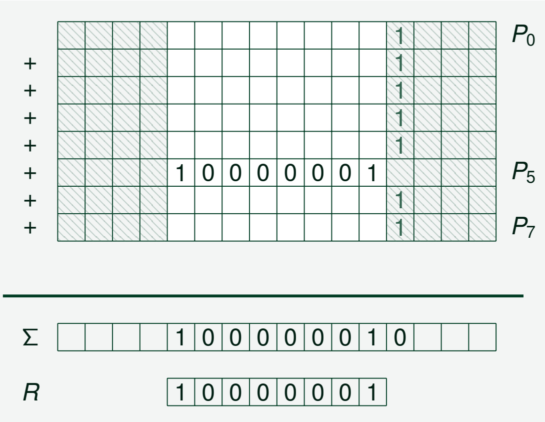
Recently presented in detail by Mantas Mikaitis.

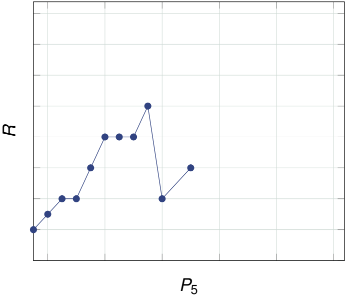




Non monotonicity of truncated Kulisch

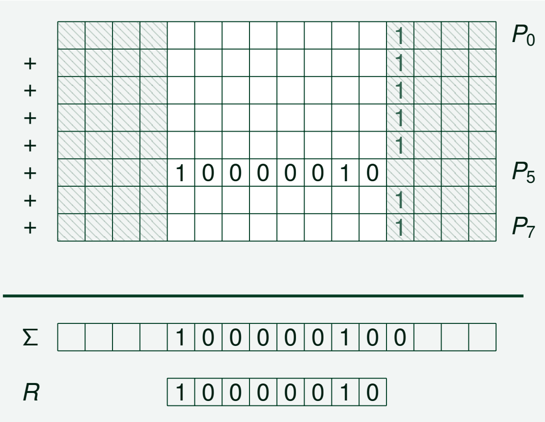
Recently presented in detail by Mantas Mikaitis.



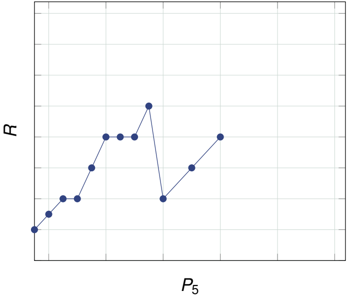


Non monotonicity of truncated Kulisch

Recently presented in detail by Mantas Mikaitis.

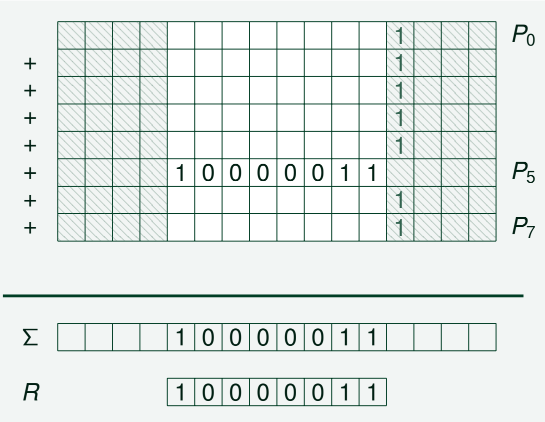


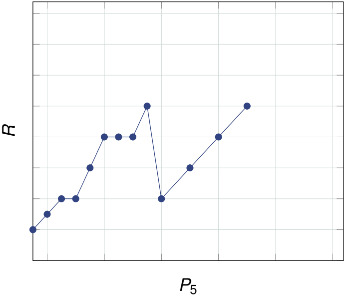
	▨	▨	▨	▨									1	▨	▨	▨	P_0
+	▨	▨	▨	▨									1	▨	▨	▨	
+	▨	▨	▨	▨									1	▨	▨	▨	
+	▨	▨	▨	▨									1	▨	▨	▨	
+	▨	▨	▨	▨									1	▨	▨	▨	
+	▨	▨	▨	▨	1	0	0	0	0	0	1	0	▨	▨	▨	▨	P_5
+	▨	▨	▨	▨									1	▨	▨	▨	
+	▨	▨	▨	▨									1	▨	▨	▨	P_7
Σ					1	0	0	0	0	0	1	0	0				
R					1	0	0	0	0	0	1	0					



Non monotonicity of truncated Kulisch

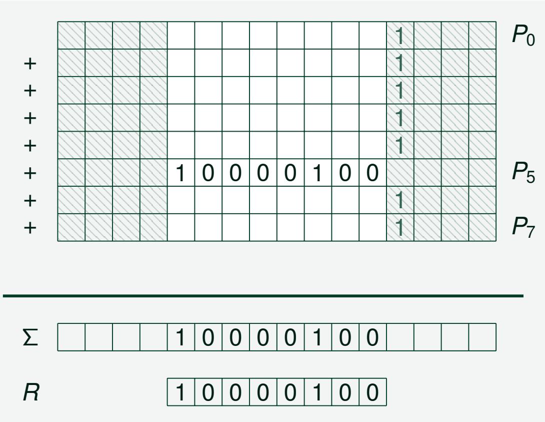
Recently presented in detail by Mantas Mikaitis.

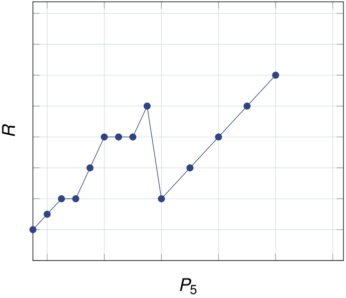




Non monotonicity of truncated Kulisch

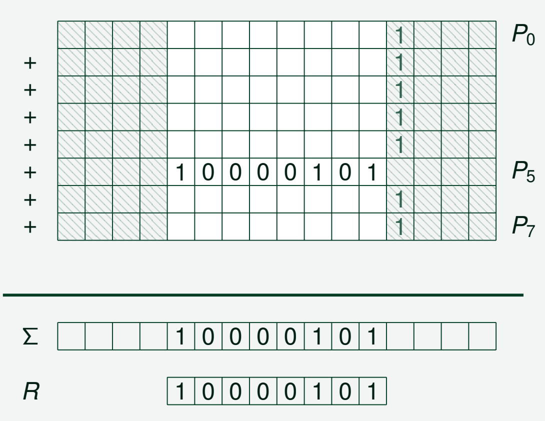
Recently presented in detail by Mantas Mikaitis.



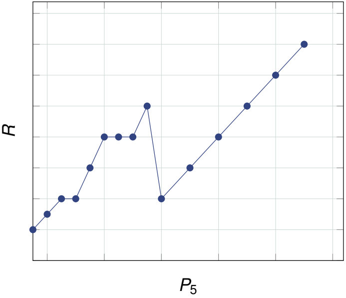


Non monotonicity of truncated Kulisch

Recently presented in detail by Mantas Mikaitis.

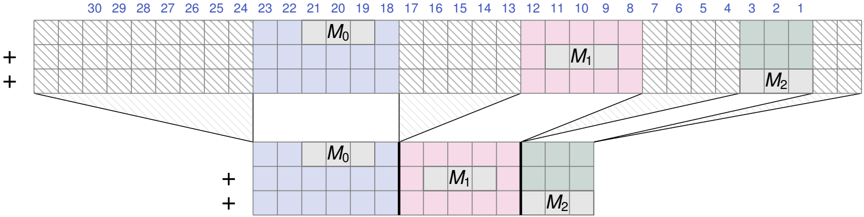


	▨	▨	▨	▨									▨ 1	▨	▨	▨	P_0
+	▨	▨	▨	▨									▨ 1	▨	▨	▨	
+	▨	▨	▨	▨									▨ 1	▨	▨	▨	
+	▨	▨	▨	▨									▨ 1	▨	▨	▨	
+	▨	▨	▨	▨									▨ 1	▨	▨	▨	
+	▨	▨	▨	▨	1	0	0	0	0	1	0	1	▨	▨	▨	▨	P_5
+	▨	▨	▨	▨									▨ 1	▨	▨	▨	
+	▨	▨	▨	▨									▨ 1	▨	▨	▨	P_7
Σ					1	0	0	0	0	1	0	1					
R					1	0	0	0	0	1	0	1					



Compressed Kulisch (not truncated)[1]

Correctly rounded sum of products

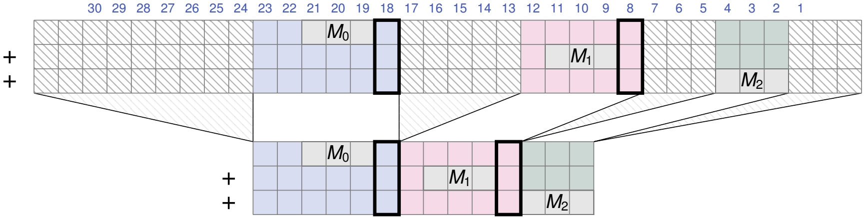


---

[1] O. Desrentes, B. Dupont de Dinechin, F. de Dinechin, "Exact Fused Dot Product Add Operators"

Compressed Kulisch (not truncated)[1]

Correctly rounded sum of products

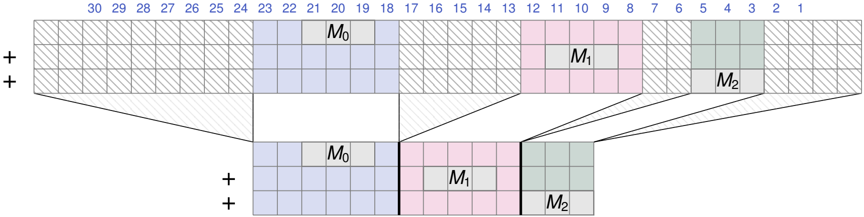


---

[1] O. Desrentes, B. Dupont de Dinechin, F. de Dinechin, "Exact Fused Dot Product Add Operators"

Compressed Kulisch (not truncated)[1]

Correctly rounded sum of products

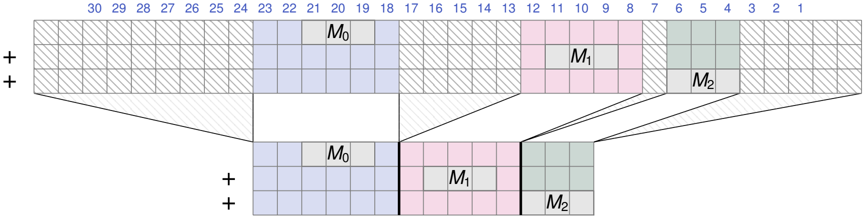


---

[1] O. Desrentes, B. Dupont de Dinechin, F. de Dinechin, "Exact Fused Dot Product Add Operators"

Compressed Kulisch (not truncated)[1]

Correctly rounded sum of products

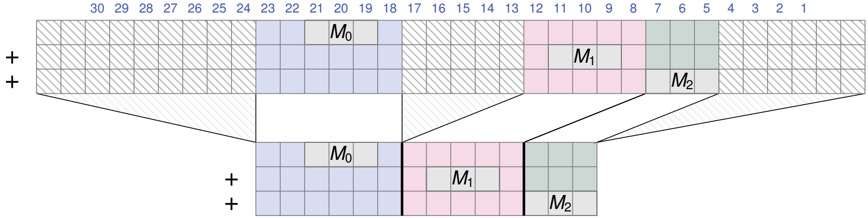


---

[1] O. Desrentes, B. Dupont de Dinechin, F. de Dinechin, "Exact Fused Dot Product Add Operators"

Compressed Kulisch (not truncated)[1]

Correctly rounded sum of products

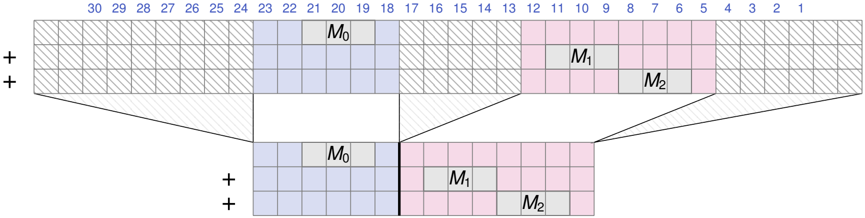


---

[1] O. Desrentes, B. Dupont de Dinechin, F. de Dinechin, "Exact Fused Dot Product Add Operators"

Compressed Kulisch (not truncated)[1]

Correctly rounded sum of products

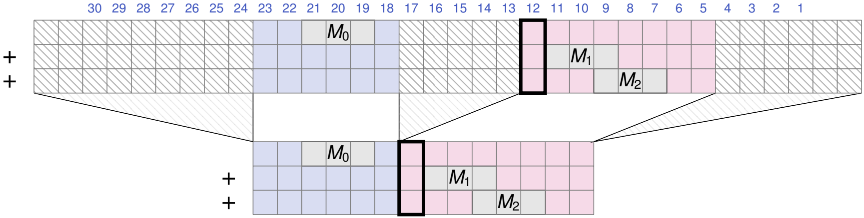


---

[1] O. Desrentes, B. Dupont de Dinechin, F. de Dinechin, "Exact Fused Dot Product Add Operators"

Compressed Kulisch (not truncated)[1]

Correctly rounded sum of products

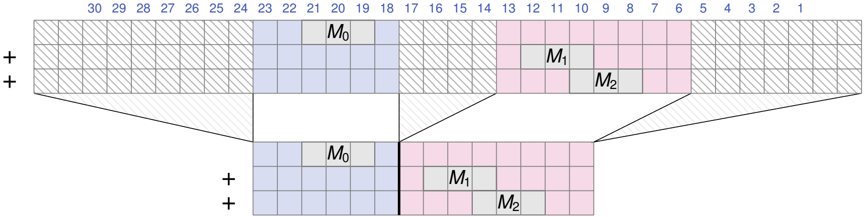


---

[1] O. Desrentes, B. Dupont de Dinechin, F. de Dinechin, "Exact Fused Dot Product Add Operators"

Compressed Kulisch (not truncated)[1]

Correctly rounded sum of products

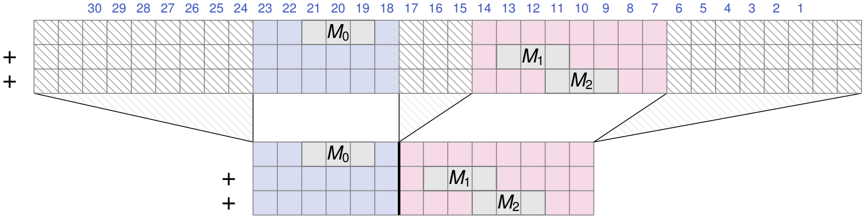


---

[1] O. Desrentes, B. Dupont de Dinechin, F. de Dinechin, "Exact Fused Dot Product Add Operators"

Compressed Kulisch (not truncated)[1]

Correctly rounded sum of products

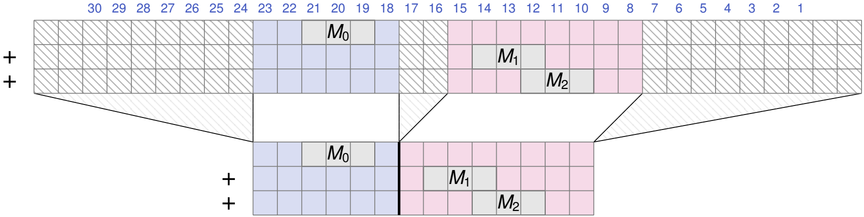


---

[1] O. Desrentes, B. Dupont de Dinechin, F. de Dinechin, "Exact Fused Dot Product Add Operators"

Compressed Kulisch (not truncated)[1]

Correctly rounded sum of products

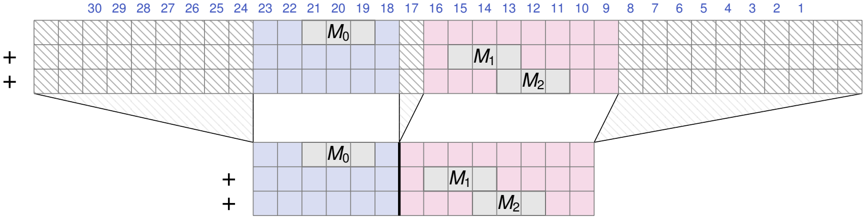


---

[1] O. Desrentes, B. Dupont de Dinechin, F. de Dinechin, "Exact Fused Dot Product Add Operators"

Compressed Kulisch (not truncated)[1]

Correctly rounded sum of products

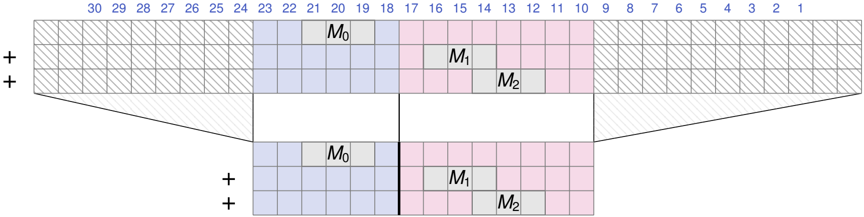


---

[1] O. Desrentes, B. Dupont de Dinechin, F. de Dinechin, "Exact Fused Dot Product Add Operators"

Compressed Kulisch (not truncated)[1]

Correctly rounded sum of products

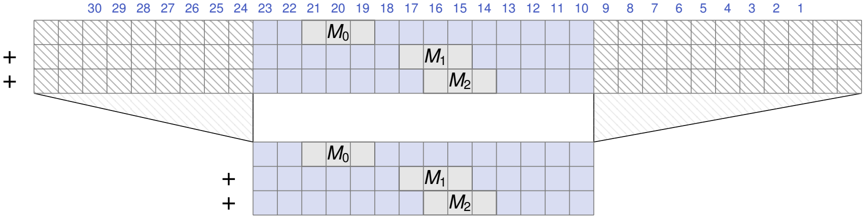


---

[1] O. Desrentes, B. Dupont de Dinechin, F. de Dinechin, "Exact Fused Dot Product Add Operators"

Compressed Kulisch (not truncated)[1]

Correctly rounded sum of products

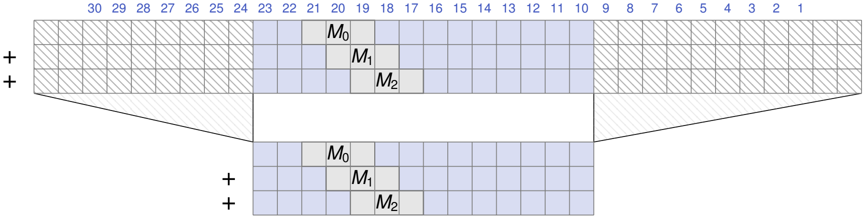


---

[1] O. Desrentes, B. Dupont de Dinechin, F. de Dinechin, "Exact Fused Dot Product Add Operators"

Compressed Kulisch (not truncated)[1]

Correctly rounded sum of products

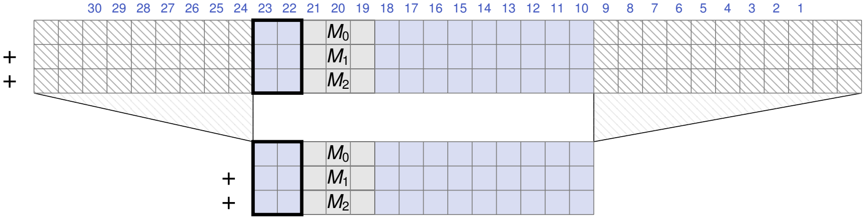


---

[1] O. Desrentes, B. Dupont de Dinechin, F. de Dinechin, "Exact Fused Dot Product Add Operators"

Compressed Kulisch (not truncated)[1]

Correctly rounded sum of products

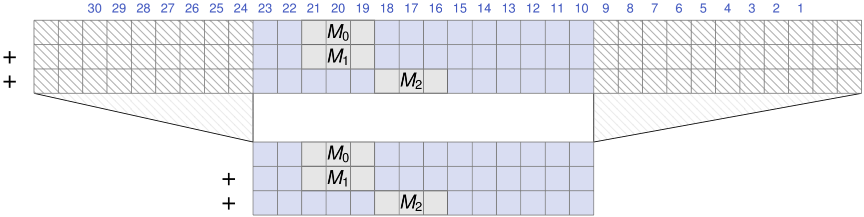


---

[1] O. Desrentes, B. Dupont de Dinechin, F. de Dinechin, "Exact Fused Dot Product Add Operators"

Compressed Kulisch (not truncated)[1]

Correctly rounded sum of products

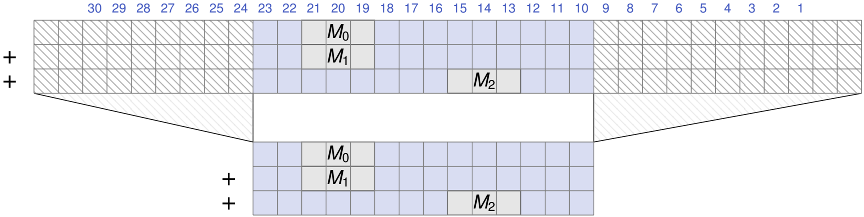


---

[1] O. Desrentes, B. Dupont de Dinechin, F. de Dinechin, "Exact Fused Dot Product Add Operators"

Compressed Kulisch (not truncated)[1]

Correctly rounded sum of products

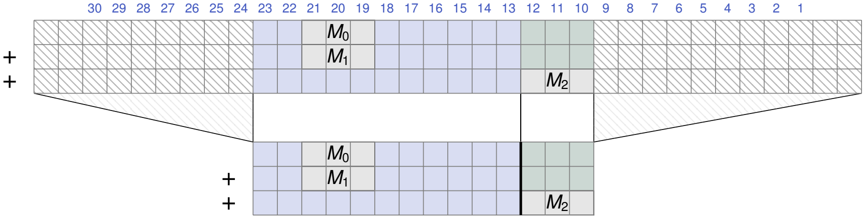


---

[1] O. Desrentes, B. Dupont de Dinechin, F. de Dinechin, "Exact Fused Dot Product Add Operators"

Compressed Kulisch (not truncated)[1]

Correctly rounded sum of products

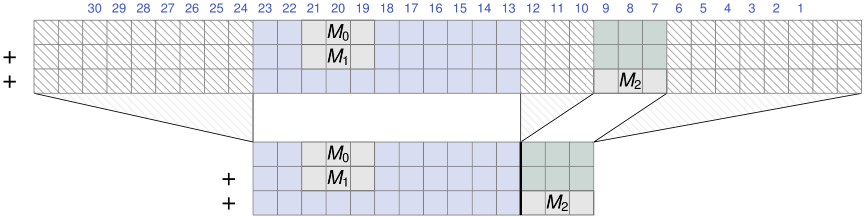


---

[1] O. Desrentes, B. Dupont de Dinechin, F. de Dinechin, "Exact Fused Dot Product Add Operators"

Compressed Kulisch (not truncated)[1]

Correctly rounded sum of products

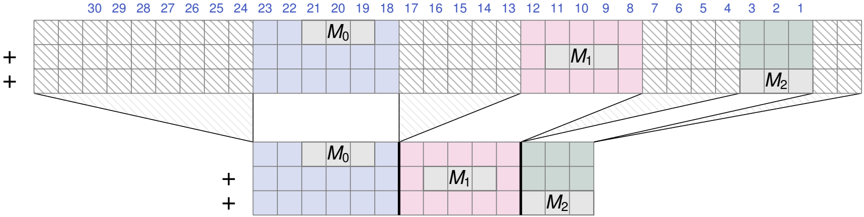


---

[1] O. Desrentes, B. Dupont de Dinechin, F. de Dinechin, "Exact Fused Dot Product Add Operators"

Compressed Kulisch (not truncated)[1]

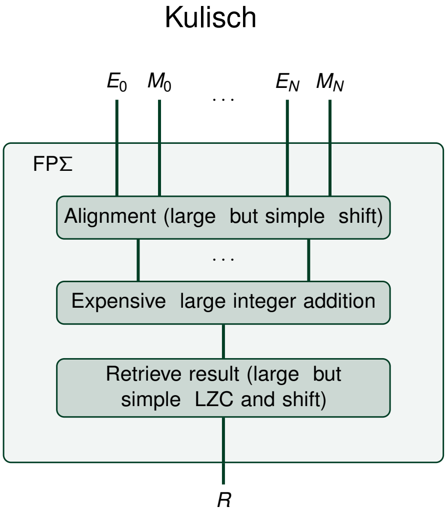
Correctly rounded sum of products



---

[1] O. Desrentes, B. Dupont de Dinechin, F. de Dinechin, "Exact Fused Dot Product Add Operators"

Architecture comparison

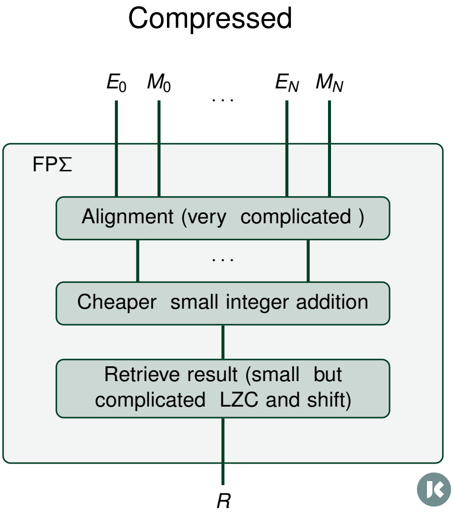


Kulisch

Alignment (large but simple shift)

Expensive large integer addition

Retrieve result (large but simple LZC and shift)



Compressed

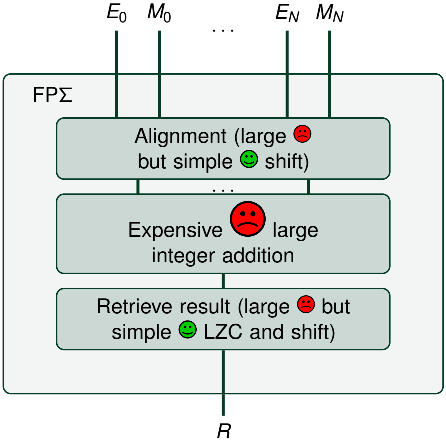
Alignment (very complicated)

Cheaper small integer addition

Retrieve result (small but complicated LZC and shift)

Architecture comparison

Kulisch



E_0 M_0 … E_N M_N

FPΣ

Alignment (large ☹ but simple ☺ shift)

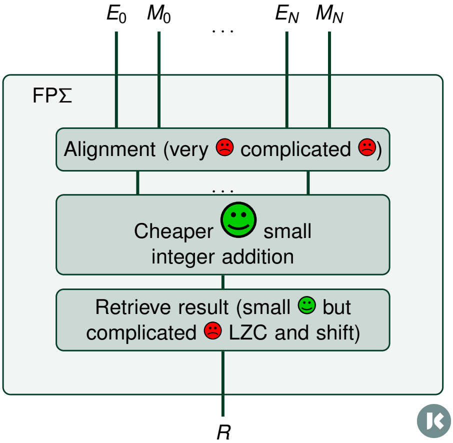
…

Expensive ☹ large integer addition

Retrieve result (large ☹ but simple ☺ LZC and shift)

R

Compressed



E_0 M_0 … E_N M_N

FPΣ

Alignment (very ☹ complicated ☹)

…

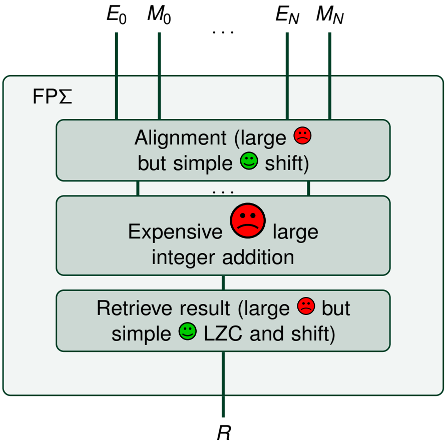
Cheaper ☺ small integer addition

Retrieve result (small ☺ but complicated ☹ LZC and shift)

R

Architecture comparison

Kulisch



E_0 M_0 . . . E_N M_N

FPΣ

Alignment (large ☹ but simple ☺ shift)

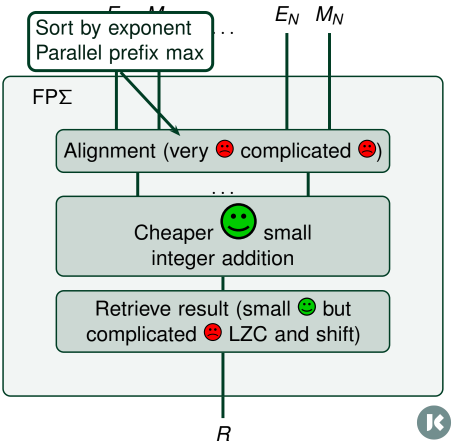
. . .

Expensive ☹ large integer addition

Retrieve result (large ☹ but simple ☺ LZC and shift)

R

Compressed



Sort by exponent
Parallel prefix max

. . . E_N M_N

FPΣ

Alignment (very ☹ complicated ☹)

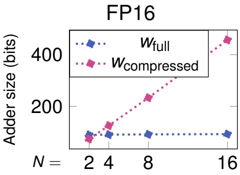
. . .

Cheaper ☺ small integer addition

Retrieve result (small ☺ but complicated ☹ LZC and shift)

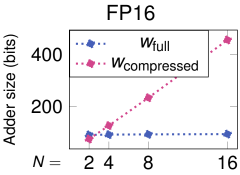
R

Results



$$w_{\text{full}} \sim 2^{w_E}$$
$$w_{\text{compressed}} \sim N \times w_F$$

Results

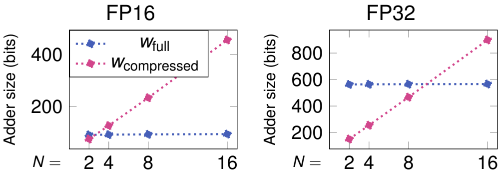


$$w_{\text{full}} \sim 2^{w_E}$$
$$w_{\text{compressed}} \sim N \times w_F$$

3 situations:

- The format has a lot of precision compared to the range, $w_{\text{compressed}} > w_{\text{full}}$ even for small N

Results

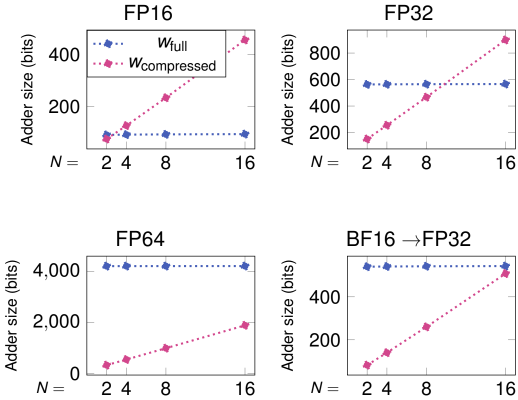


$w_{\text{full}} \sim 2^{w_E}$
$w_{\text{compressed}} \sim N \times w_F$

3 situations:

- The format has a lot of precision compared to the range, $w_{\text{compressed}} > w_{\text{full}}$ even for small N
- The format is more balanced, $w_{\text{compressed}} < w_{\text{full}}$ for small N

Results



$w_{\text{full}} \sim 2^{w_E}$

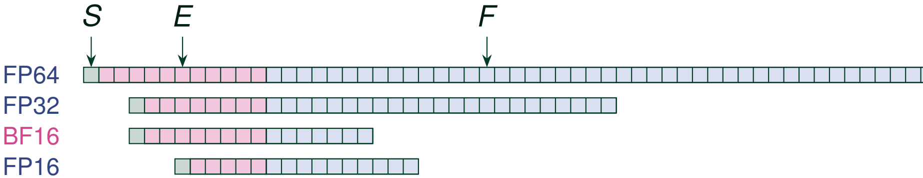
$w_{\text{compressed}} \sim N \times w_F$

3 situations:

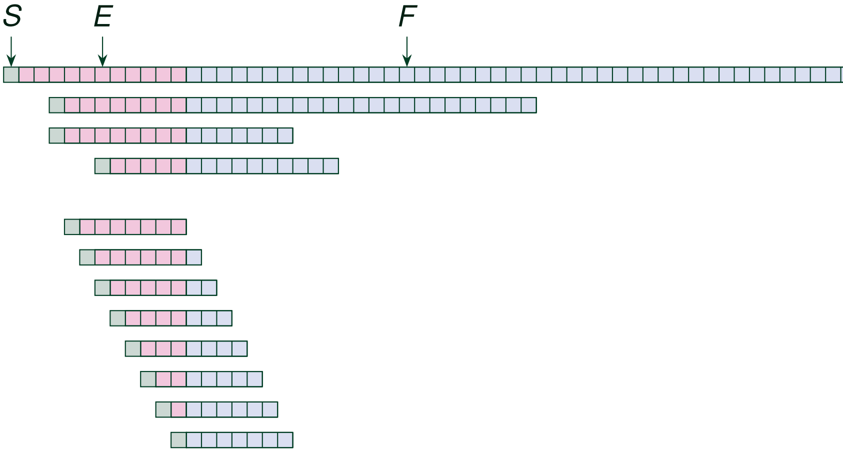
- The format has a lot of precision compared to the range, $w_{\text{compressed}} > w_{\text{full}}$ even for small N
- The format is more balanced, $w_{\text{compressed}} < w_{\text{full}}$ for small N
- The format has a lot of range compared to the precision $w_{\text{compressed}} < w_{\text{full}}$ until larger N

8 bits formats

Formats



Formats



S *E* *F*

FP64

FP32

BF16

FP16

E7M0

E6M1

E5M2

E4M3

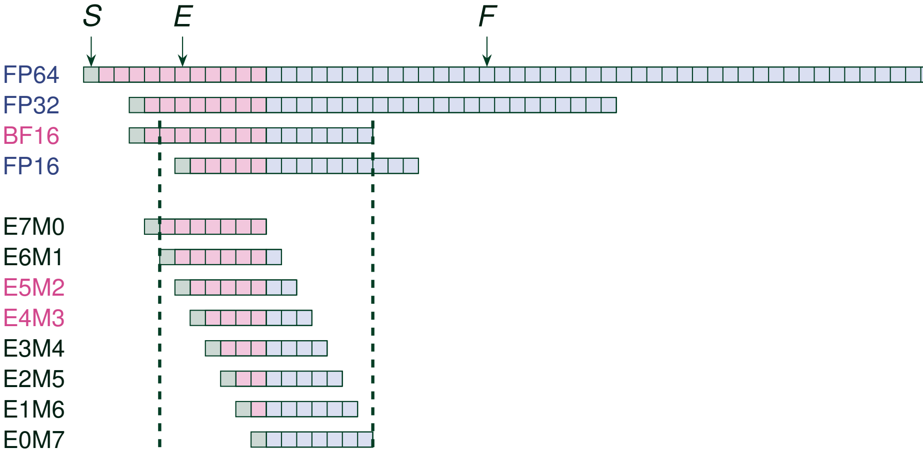
E3M4

E2M5

E1M6

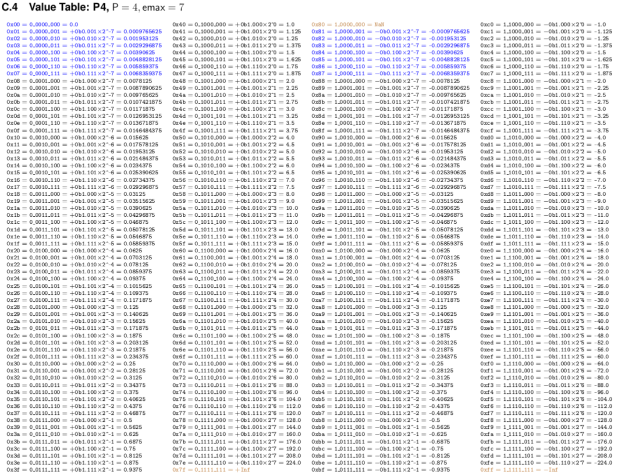
E0M7

Formats


S
E
F
FP64
FP32
BF16
FP16
E7M0
E6M1
E5M2
E4M3
E3M4
E2M5
E1M6
E0M7

Special values ?

8 bits is not a lot of bits (256 different values)



C.4 Value Table: P4, $\mathrm{P} = 4, \mathrm{emax} = 7$

```
0x00 = 0_0000_000 = 0.0
0x01 = 0_0000_001 = +0b0.001×2^-7 = 0.0009765625
0x02 = 0_0000_010 = +0b0.010×2^-7 = 0.001953125
0x03 = 0_0000_011 = +0b0.011×2^-7 = 0.0029296875
0x04 = 0_0000_100 = +0b0.100×2^-7 = 0.00390625
0x05 = 0_0000_101 = +0b0.101×2^-7 = 0.0048828125
0x06 = 0_0000_110 = +0b0.110×2^-7 = 0.005859375
0x07 = 0_0000_111 = +0b0.111×2^-7 = 0.0068359375
0x08 = 0_0001_000 = +0b1.000×2^-7 = 0.0078125
0x09 = 0_0001_001 = +0b1.001×2^-7 = 0.0087890625
0x0a = 0_0001_010 = +0b1.010×2^-7 = 0.009765625
0x0b = 0_0001_011 = +0b1.011×2^-7 = 0.0107421875
0x0c = 0_0001_100 = +0b1.100×2^-7 = 0.01171875
0x0d = 0_0001_101 = +0b1.101×2^-7 = 0.0126953125
0x0e = 0_0001_110 = +0b1.110×2^-7 = 0.013671875
0x0f = 0_0001_111 = +0b1.111×2^-7 = 0.0146484375
0x10 = 0_0010_000 = +0b1.000×2^-6 = 0.015625
0x11 = 0_0010_001 = +0b1.001×2^-6 = 0.017578125
0x12 = 0_0010_010 = +0b1.010×2^-6 = 0.01953125
0x13 = 0_0010_011 = +0b1.011×2^-6 = 0.021484375
0x14 = 0_0010_100 = +0b1.100×2^-6 = 0.0234375
0x15 = 0_0010_101 = +0b1.101×2^-6 = 0.025390625
0x16 = 0_0010_110 = +0b1.110×2^-6 = 0.02734375
0x17 = 0_0010_111 = +0b1.111×2^-6 = 0.029296875
0x18 = 0_0011_000 = +0b1.000×2^-5 = 0.03125
0x19 = 0_0011_001 = +0b1.001×2^-5 = 0.03515625
0x1a = 0_0011_010 = +0b1.010×2^-5 = 0.0390625
0x1b = 0_0011_011 = +0b1.011×2^-5 = 0.04296875
0x1c = 0_0011_100 = +0b1.100×2^-5 = 0.046875
0x1d = 0_0011_101 = +0b1.101×2^-5 = 0.05078125
0x1e = 0_0011_110 = +0b1.110×2^-5 = 0.0546875
0x1f = 0_0011_111 = +0b1.111×2^-5 = 0.05859375
0x20 = 0_0100_000 = +0b1.000×2^-4 = 0.0625
0x21 = 0_0100_001 = +0b1.001×2^-4 = 0.0703125
0x22 = 0_0100_010 = +0b1.010×2^-4 = 0.078125
0x23 = 0_0100_011 = +0b1.011×2^-4 = 0.0859375
0x24 = 0_0100_100 = +0b1.100×2^-4 = 0.09375
0x25 = 0_0100_101 = +0b1.101×2^-4 = 0.1015625
0x26 = 0_0100_110 = +0b1.110×2^-4 = 0.109375
0x27 = 0_0100_111 = +0b1.111×2^-4 = 0.1171875
0x28 = 0_0101_000 = +0b1.000×2^-3 = 0.125
0x29 = 0_0101_001 = +0b1.001×2^-3 = 0.140625
0x2a = 0_0101_010 = +0b1.010×2^-3 = 0.15625
0x2b = 0_0101_011 = +0b1.011×2^-3 = 0.171875
0x2c = 0_0101_100 = +0b1.100×2^-3 = 0.1875
0x2d = 0_0101_101 = +0b1.101×2^-3 = 0.203125
0x2e = 0_0101_110 = +0b1.110×2^-3 = 0.21875
0x2f = 0_0101_111 = +0b1.111×2^-3 = 0.234375
0x30 = 0_0110_000 = +0b1.000×2^-2 = 0.25
0x31 = 0_0110_001 = +0b1.001×2^-2 = 0.28125
0x32 = 0_0110_010 = +0b1.010×2^-2 = 0.3125
0x33 = 0_0110_011 = +0b1.011×2^-2 = 0.34375
0x34 = 0_0110_100 = +0b1.100×2^-2 = 0.375
0x35 = 0_0110_101 = +0b1.101×2^-2 = 0.40625
0x36 = 0_0110_110 = +0b1.110×2^-2 = 0.4375
0x37 = 0_0110_111 = +0b1.111×2^-2 = 0.46875
0x38 = 0_0111_000 = +0b1.000×2^-1 = 0.5
0x39 = 0_0111_001 = +0b1.001×2^-1 = 0.5625
0x3a = 0_0111_010 = +0b1.010×2^-1 = 0.625
0x3b = 0_0111_011 = +0b1.011×2^-1 = 0.6875
0x3c = 0_0111_100 = +0b1.100×2^-1 = 0.75
0x3d = 0_0111_101 = +0b1.101×2^-1 = 0.8125
0x3e = 0_0111_110 = +0b1.110×2^-1 = 0.875
0x3f = 0_0111_111 = +0b1.111×2^-1 = 0.9375
0x40 = 0_1000_000 = +0b1.000×2^0 = 1.0
0x41 = 0_1000_001 = +0b1.001×2^0 = 1.125
0x42 = 0_1000_010 = +0b1.010×2^0 = 1.25
0x43 = 0_1000_011 = +0b1.011×2^0 = 1.375
0x44 = 0_1000_100 = +0b1.100×2^0 = 1.5
0x45 = 0_1000_101 = +0b1.101×2^0 = 1.625
0x46 = 0_1000_110 = +0b1.110×2^0 = 1.75
0x47 = 0_1000_111 = +0b1.111×2^0 = 1.875
0x48 = 0_1001_000 = +0b1.000×2^1 = 2.0
0x49 = 0_1001_001 = +0b1.001×2^1 = 2.25
0x4a = 0_1001_010 = +0b1.010×2^1 = 2.5
0x4b = 0_1001_011 = +0b1.011×2^1 = 2.75
0x4c = 0_1001_100 = +0b1.100×2^1 = 3.0
0x4d = 0_1001_101 = +0b1.101×2^1 = 3.25
0x4e = 0_1001_110 = +0b1.110×2^1 = 3.5
0x4f = 0_1001_111 = +0b1.111×2^1 = 3.75
0x50 = 0_1010_000 = +0b1.000×2^2 = 4.0
0x51 = 0_1010_001 = +0b1.001×2^2 = 4.5
0x52 = 0_1010_010 = +0b1.010×2^2 = 5.0
0x53 = 0_1010_011 = +0b1.011×2^2 = 5.5
0x54 = 0_1010_100 = +0b1.100×2^2 = 6.0
0x55 = 0_1010_101 = +0b1.101×2^2 = 6.5
0x56 = 0_1010_110 = +0b1.110×2^2 = 7.0
0x57 = 0_1010_111 = +0b1.111×2^2 = 7.5
0x58 = 0_1011_000 = +0b1.000×2^3 = 8.0
0x59 = 0_1011_001 = +0b1.001×2^3 = 9.0
0x5a = 0_1011_010 = +0b1.010×2^3 = 10.0
0x5b = 0_1011_011 = +0b1.011×2^3 = 11.0
0x5c = 0_1011_100 = +0b1.100×2^3 = 12.0
0x5d = 0_1011_101 = +0b1.101×2^3 = 13.0
0x5e = 0_1011_110 = +0b1.110×2^3 = 14.0
0x5f = 0_1011_111 = +0b1.111×2^3 = 15.0
0x60 = 0_1100_000 = +0b1.000×2^4 = 16.0
0x61 = 0_1100_001 = +0b1.001×2^4 = 18.0
0x62 = 0_1100_010 = +0b1.010×2^4 = 20.0
0x63 = 0_1100_011 = +0b1.011×2^4 = 22.0
0x64 = 0_1100_100 = +0b1.100×2^4 = 24.0
0x65 = 0_1100_101 = +0b1.101×2^4 = 26.0
0x66 = 0_1100_110 = +0b1.110×2^4 = 28.0
0x67 = 0_1100_111 = +0b1.111×2^4 = 30.0
0x68 = 0_1101_000 = +0b1.000×2^5 = 32.0
0x69 = 0_1101_001 = +0b1.001×2^5 = 36.0
0x6a = 0_1101_010 = +0b1.010×2^5 = 40.0
0x6b = 0_1101_011 = +0b1.011×2^5 = 44.0
0x6c = 0_1101_100 = +0b1.100×2^5 = 48.0
0x6d = 0_1101_101 = +0b1.101×2^5 = 52.0
0x6e = 0_1101_110 = +0b1.110×2^5 = 56.0
0x6f = 0_1101_111 = +0b1.111×2^5 = 60.0
0x70 = 0_1110_000 = +0b1.000×2^6 = 64.0
0x71 = 0_1110_001 = +0b1.001×2^6 = 72.0
0x72 = 0_1110_010 = +0b1.010×2^6 = 80.0
0x73 = 0_1110_011 = +0b1.011×2^6 = 88.0
0x74 = 0_1110_100 = +0b1.100×2^6 = 96.0
0x75 = 0_1110_101 = +0b1.101×2^6 = 104.0
0x76 = 0_1110_110 = +0b1.110×2^6 = 112.0
0x77 = 0_1110_111 = +0b1.111×2^6 = 120.0
0x78 = 0_1111_000 = +0b1.000×2^7 = 128.0
0x79 = 0_1111_001 = +0b1.001×2^7 = 144.0
0x7a = 0_1111_010 = +0b1.010×2^7 = 160.0
0x7b = 0_1111_011 = +0b1.011×2^7 = 176.0
0x7c = 0_1111_100 = +0b1.100×2^7 = 192.0
0x7d = 0_1111_101 = +0b1.101×2^7 = 208.0
0x7e = 0_1111_110 = +0b1.110×2^7 = 224.0
0x7f = 0_1111_111 = +Inf
0x80 = 1_0000_000 = NaN
0x81 = 1_0000_001 = −0b0.001×2^-7 = -0.0009765625
0x82 = 1_0000_010 = −0b0.010×2^-7 = -0.001953125
0x83 = 1_0000_011 = −0b0.011×2^-7 = -0.0029296875
0x84 = 1_0000_100 = −0b0.100×2^-7 = -0.00390625
0x85 = 1_0000_101 = −0b0.101×2^-7 = -0.0048828125
0x86 = 1_0000_110 = −0b0.110×2^-7 = -0.005859375
0x87 = 1_0000_111 = −0b0.111×2^-7 = -0.0068359375
0x88 = 1_0001_000 = −0b1.000×2^-7 = -0.0078125
0x89 = 1_0001_001 = −0b1.001×2^-7 = -0.0087890625
0x8a = 1_0001_010 = −0b1.010×2^-7 = -0.009765625
0x8b = 1_0001_011 = −0b1.011×2^-7 = -0.0107421875
0x8c = 1_0001_100 = −0b1.100×2^-7 = -0.01171875
0x8d = 1_0001_101 = −0b1.101×2^-7 = -0.0126953125
0x8e = 1_0001_110 = −0b1.110×2^-7 = -0.013671875
0x8f = 1_0001_111 = −0b1.111×2^-7 = -0.0146484375
0x90 = 1_0010_000 = −0b1.000×2^-6 = -0.015625
0x91 = 1_0010_001 = −0b1.001×2^-6 = -0.017578125
0x92 = 1_0010_010 = −0b1.010×2^-6 = -0.01953125
0x93 = 1_0010_011 = −0b1.011×2^-6 = -0.021484375
0x94 = 1_0010_100 = −0b1.100×2^-6 = -0.0234375
0x95 = 1_0010_101 = −0b1.101×2^-6 = -0.025390625
0x96 = 1_0010_110 = −0b1.110×2^-6 = -0.02734375
0x97 = 1_0010_111 = −0b1.111×2^-6 = -0.029296875
0x98 = 1_0011_000 = −0b1.000×2^-5 = -0.03125
0x99 = 1_0011_001 = −0b1.001×2^-5 = -0.03515625
0x9a = 1_0011_010 = −0b1.010×2^-5 = -0.0390625
0x9b = 1_0011_011 = −0b1.011×2^-5 = -0.04296875
0x9c = 1_0011_100 = −0b1.100×2^-5 = -0.046875
0x9d = 1_0011_101 = −0b1.101×2^-5 = -0.05078125
0x9e = 1_0011_110 = −0b1.110×2^-5 = -0.0546875
0x9f = 1_0011_111 = −0b1.111×2^-5 = -0.05859375
0xa0 = 1_0100_000 = −0b1.000×2^-4 = -0.0625
0xa1 = 1_0100_001 = −0b1.001×2^-4 = -0.0703125
0xa2 = 1_0100_010 = −0b1.010×2^-4 = -0.078125
0xa3 = 1_0100_011 = −0b1.011×2^-4 = -0.0859375
0xa4 = 1_0100_100 = −0b1.100×2^-4 = -0.09375
0xa5 = 1_0100_101 = −0b1.101×2^-4 = -0.1015625
0xa6 = 1_0100_110 = −0b1.110×2^-4 = -0.109375
0xa7 = 1_0100_111 = −0b1.111×2^-4 = -0.1171875
0xa8 = 1_0101_000 = −0b1.000×2^-3 = -0.125
0xa9 = 1_0101_001 = −0b1.001×2^-3 = -0.140625
0xaa = 1_0101_010 = −0b1.010×2^-3 = -0.15625
0xab = 1_0101_011 = −0b1.011×2^-3 = -0.171875
0xac = 1_0101_100 = −0b1.100×2^-3 = -0.1875
0xad = 1_0101_101 = −0b1.101×2^-3 = -0.203125
0xae = 1_0101_110 = −0b1.110×2^-3 = -0.21875
0xaf = 1_0101_111 = −0b1.111×2^-3 = -0.234375
0xb0 = 1_0110_000 = −0b1.000×2^-2 = -0.25
0xb1 = 1_0110_001 = −0b1.001×2^-2 = -0.28125
0xb2 = 1_0110_010 = −0b1.010×2^-2 = -0.3125
0xb3 = 1_0110_011 = −0b1.011×2^-2 = -0.34375
0xb4 = 1_0110_100 = −0b1.100×2^-2 = -0.375
0xb5 = 1_0110_101 = −0b1.101×2^-2 = -0.40625
0xb6 = 1_0110_110 = −0b1.110×2^-2 = -0.4375
0xb7 = 1_0110_111 = −0b1.111×2^-2 = -0.46875
0xb8 = 1_0111_000 = −0b1.000×2^-1 = -0.5
0xb9 = 1_0111_001 = −0b1.001×2^-1 = -0.5625
0xba = 1_0111_010 = −0b1.010×2^-1 = -0.625
0xbb = 1_0111_011 = −0b1.011×2^-1 = -0.6875
0xbc = 1_0111_100 = −0b1.100×2^-1 = -0.75
0xbd = 1_0111_101 = −0b1.101×2^-1 = -0.8125
0xbe = 1_0111_110 = −0b1.110×2^-1 = -0.875
0xbf = 1_0111_111 = −0b1.111×2^-1 = -0.9375
0xc0 = 1_1000_000 = −0b1.000×2^0 = -1.0
0xc1 = 1_1000_001 = −0b1.001×2^0 = -1.125
0xc2 = 1_1000_010 = −0b1.010×2^0 = -1.25
0xc3 = 1_1000_011 = −0b1.011×2^0 = -1.375
0xc4 = 1_1000_100 = −0b1.100×2^0 = -1.5
0xc5 = 1_1000_101 = −0b1.101×2^0 = -1.625
0xc6 = 1_1000_110 = −0b1.110×2^0 = -1.75
0xc7 = 1_1000_111 = −0b1.111×2^0 = -1.875
0xc8 = 1_1001_000 = −0b1.000×2^1 = -2.0
0xc9 = 1_1001_001 = −0b1.001×2^1 = -2.25
0xca = 1_1001_010 = −0b1.010×2^1 = -2.5
0xcb = 1_1001_011 = −0b1.011×2^1 = -2.75
0xcc = 1_1001_100 = −0b1.100×2^1 = -3.0
0xcd = 1_1001_101 = −0b1.101×2^1 = -3.25
0xce = 1_1001_110 = −0b1.110×2^1 = -3.5
0xcf = 1_1001_111 = −0b1.111×2^1 = -3.75
0xd0 = 1_1010_000 = −0b1.000×2^2 = -4.0
0xd1 = 1_1010_001 = −0b1.001×2^2 = -4.5
0xd2 = 1_1010_010 = −0b1.010×2^2 = -5.0
0xd3 = 1_1010_011 = −0b1.011×2^2 = -5.5
0xd4 = 1_1010_100 = −0b1.100×2^2 = -6.0
0xd5 = 1_1010_101 = −0b1.101×2^2 = -6.5
0xd6 = 1_1010_110 = −0b1.110×2^2 = -7.0
0xd7 = 1_1010_111 = −0b1.111×2^2 = -7.5
0xd8 = 1_1011_000 = −0b1.000×2^3 = -8.0
0xd9 = 1_1011_001 = −0b1.001×2^3 = -9.0
0xda = 1_1011_010 = −0b1.010×2^3 = -10.0
0xdb = 1_1011_011 = −0b1.011×2^3 = -11.0
0xdc = 1_1011_100 = −0b1.100×2^3 = -12.0
0xdd = 1_1011_101 = −0b1.101×2^3 = -13.0
0xde = 1_1011_110 = −0b1.110×2^3 = -14.0
0xdf = 1_1011_111 = −0b1.111×2^3 = -15.0
0xe0 = 1_1100_000 = −0b1.000×2^4 = -16.0
0xe1 = 1_1100_001 = −0b1.001×2^4 = -18.0
0xe2 = 1_1100_010 = −0b1.010×2^4 = -20.0
0xe3 = 1_1100_011 = −0b1.011×2^4 = -22.0
0xe4 = 1_1100_100 = −0b1.100×2^4 = -24.0
0xe5 = 1_1100_101 = −0b1.101×2^4 = -26.0
0xe6 = 1_1100_110 = −0b1.110×2^4 = -28.0
0xe7 = 1_1100_111 = −0b1.111×2^4 = -30.0
0xe8 = 1_1101_000 = −0b1.000×2^5 = -32.0
0xe9 = 1_1101_001 = −0b1.001×2^5 = -36.0
0xea = 1_1101_010 = −0b1.010×2^5 = -40.0
0xeb = 1_1101_011 = −0b1.011×2^5 = -44.0
0xec = 1_1101_100 = −0b1.100×2^5 = -48.0
0xed = 1_1101_101 = −0b1.101×2^5 = -52.0
0xee = 1_1101_110 = −0b1.110×2^5 = -56.0
0xef = 1_1101_111 = −0b1.111×2^5 = -60.0
0xf0 = 1_1110_000 = −0b1.000×2^6 = -64.0
0xf1 = 1_1110_001 = −0b1.001×2^6 = -72.0
0xf2 = 1_1110_010 = −0b1.010×2^6 = -80.0
0xf3 = 1_1110_011 = −0b1.011×2^6 = -88.0
0xf4 = 1_1110_100 = −0b1.100×2^6 = -96.0
0xf5 = 1_1110_101 = −0b1.101×2^6 = -104.0
0xf6 = 1_1110_110 = −0b1.110×2^6 = -112.0
0xf7 = 1_1110_111 = −0b1.111×2^6 = -120.0
0xf8 = 1_1111_000 = −0b1.000×2^7 = -128.0
0xf9 = 1_1111_001 = −0b1.001×2^7 = -144.0
0xfa = 1_1111_010 = −0b1.010×2^7 = -160.0
0xfb = 1_1111_011 = −0b1.011×2^7 = -176.0
0xfc = 1_1111_100 = −0b1.100×2^7 = -192.0
0xfd = 1_1111_101 = −0b1.101×2^7 = -208.0
0xfe = 1_1111_110 = −0b1.110×2^7 = -224.0
0xff = 1_1111_111 = −Inf
```

Special values ?

8 bits is not a lot of bits (256 different values)

NaN

Having $2^{w_F+1} - 2$ NaN values is a waste of encoding space.
Should we keep at least one ? Yes

Special values ?

8 bits is not a lot of bits (256 different values)

NaN

Having $2^{w_F+1} - 2$ NaN values is a waste of encoding space.
Should we keep at least one ? Yes

∞

Having 2 infinities is a waste of encoding space.
Should we keep them anyway ? Maybe

Special values ?

8 bits is not a lot of bits (256 different values)

NaN

Having $2^{w_F+1} - 2$ NaN values is a waste of encoding space.
Should we keep at least one ? Yes

∞

Having 2 infinities is a waste of encoding space.
Should we keep them anyway ? Maybe

-0

Having 2 zeros is a waste of encoding space.
Should we keep -0 ? Maybe

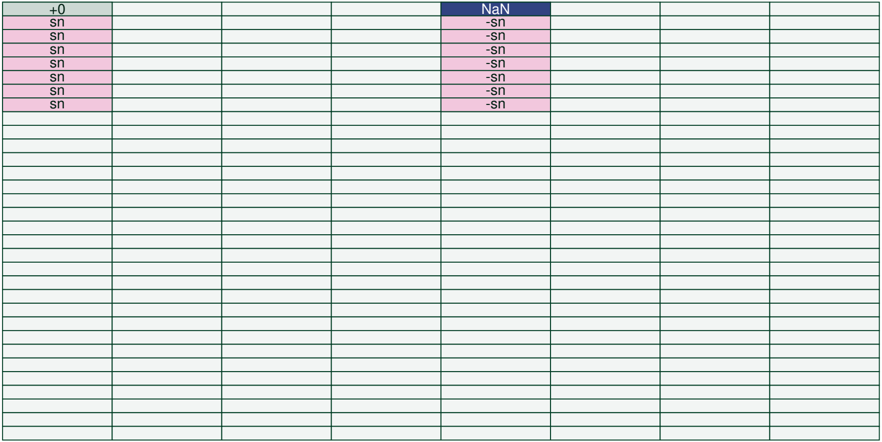
Use of encoding space (E4M3 example)

IEEE-754-like:

+0				-0			
sn				-sn			
sn				-sn			
sn				-sn			
sn				-sn			
sn				-sn			
sn				-sn			
sn				-sn			
			$+\infty$				$-\infty$
			NaN				NaN
			NaN				NaN
			NaN				NaN
			NaN				NaN
			NaN				NaN
			NaN				NaN
			NaN				NaN

Use of encoding space (E4M3 example)

Graphcore[1]:



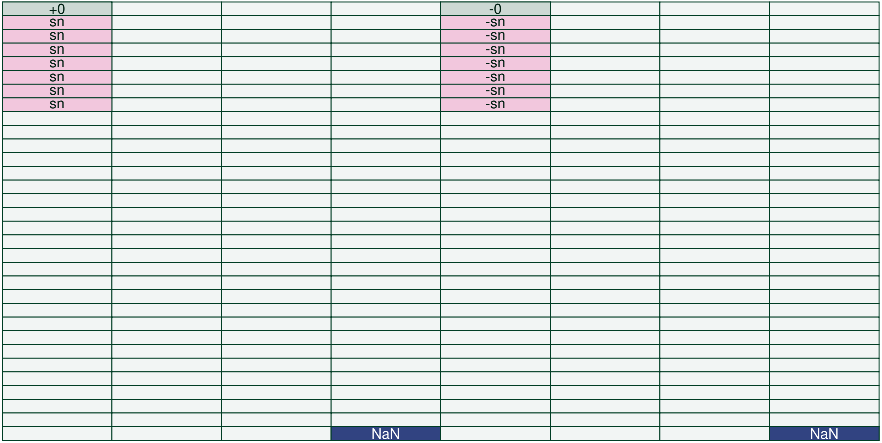
+0				NaN			
sn				-sn			
sn				-sn			
sn				-sn			
sn				-sn			
sn				-sn			
sn				-sn			
sn				-sn			

---

[1] B. Noune, P. Jones, D. Justus, D. Masters, and C. Luschi, "8-bit numerical formats for deep neural networks," 2022

Use of encoding space (E4M3 example)

Intel Arm NVIDIA[2]:



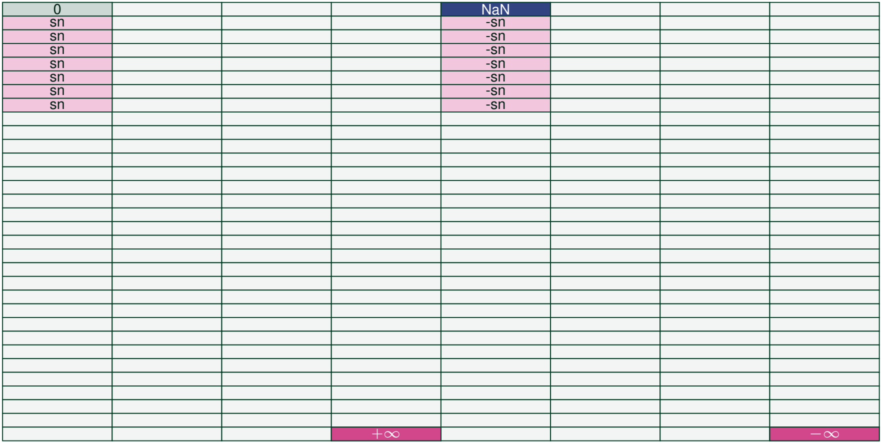
+0				-0			
sn				-sn			
sn				-sn			
sn				-sn			
sn				-sn			
sn				-sn			
sn				-sn			
sn				-sn			
			NaN				NaN

---

[2] P. Micikevicius, D. Stosic, N. Burgess, M. Cornea, P. Dubey, R. Grisenthwaite, S. Ha, A. Heinecke, P. Judd, J. Kamalu, N. Mellempudi, S. Oberman, M. Shoeybi, M. Siu, and H. Wu, "Fp8 formats for deep learning," 2022

Use of encoding space (E4M3 example)

IEEE WG P3109 standard[3]:



0				NaN			
sn				-sn			
sn				-sn			
sn				-sn			
sn				-sn			
sn				-sn			
sn				-sn			
sn				-sn			
			$+\infty$				$-\infty$

---

[3] "IEEE Working Group P3109 Interim Report on 8-bit Binary Floating-point Formats"

Use of encoding space (E4M3 example)

IEEE WG P3109 standard (with saturation)[3]:

+0				NaN			
sn				-sn			
sn				-sn			
sn				-sn			
sn				-sn			
sn				-sn			
sn				-sn			
sn				-sn			

---

[3] "IEEE Working Group P3109 Interim Report on 8-bit Binary Floating-point Formats"

Bias ?

For a real exponent E, it is encoded in the format like a positive number, by adding a bias.

	FP16 (IEEE-754)
w_E	5
Bias	$2^{w_E-1}-1=15$
Max exp	$2^{w_E-1}-1=15$
Min exp	$-2^{w_E-1}+2=-14$

Bias ?

For a real exponent E, it is encoded in the format like a positive number, by adding a bias.

	FP16 (IEEE-754)	E5M2 (until recently)
w_E	5	5
Bias	$2^{w_E-1}-1=15$	$2^{w_E-1}-1=15$
Max exp	$2^{w_E-1}-1=15$	$2^{w_E-1}=16$
Min exp	$-2^{w_E-1}+2=-14$	$-2^{w_E-1}+2=-14$

Bias ?

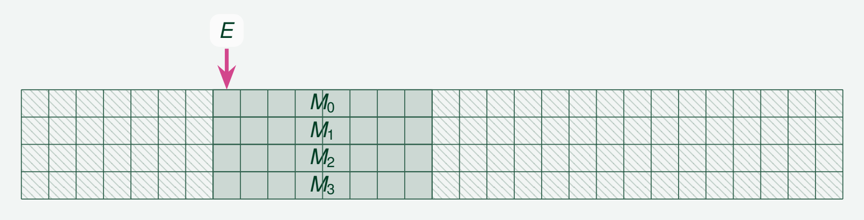
For a real exponent E, it is encoded in the format like a positive number, by adding a bias.

	FP16 (IEEE-754)	E5M2 (until recently)	E5M2 (IEEE WG P3109)
w_E	5	5	5
Bias	$2^{w_E-1}-1=15$	$2^{w_E-1}-1=15$	$2^{w_E-1}=16$
Max exp	$2^{w_E-1}-1=15$	$2^{w_E-1}=16$	$2^{w_E}-1=15$
Min exp	$-2^{w_E-1}+2=-14$	$-2^{w_E-1}+2=-14$	$-2^{w_E-1}+1=-15$

Block floating point

History

A vector of fixpoint numbers that share an exponent.



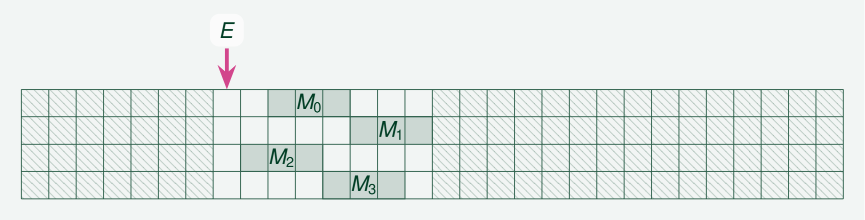
Block floating point

History

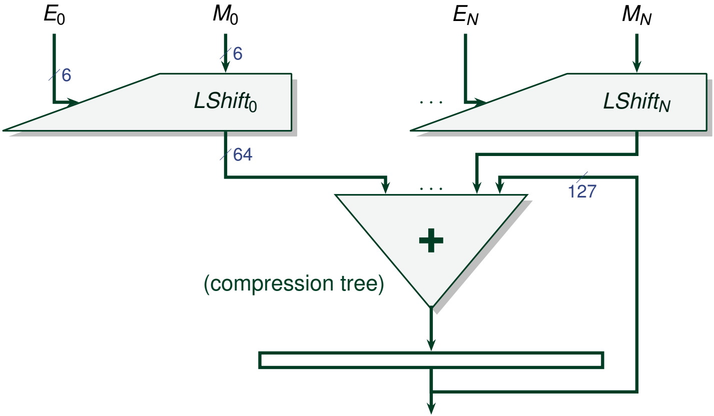
A vector of fixpoint numbers that share an exponent.

Recent variant

A vector of small floating point numbers that share a bias modifier/scaling factor



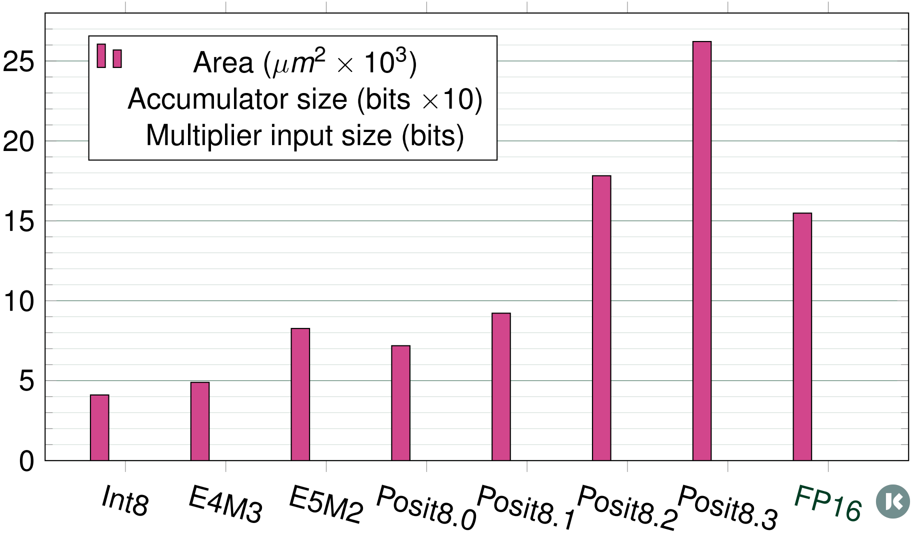
Kulisch architecture for exact 8 bits dot product



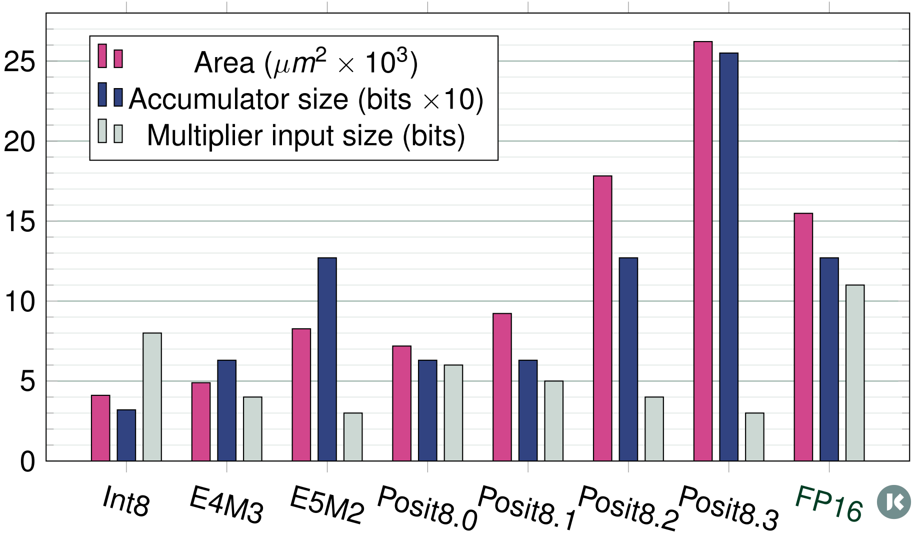
Kulisch architecture for exact 8 bits dot product

Format	Product				Accumulator	
	size	LSB	MSB	w (in bits)	MSB	w_{acc} (in bits)
INT8	8×8	0	15	16	31	32
E4M3	4×4	-18	16	36	44	63+1
E5M2	3×3	-32	30	64	94	127+1
Posit8.0	6×6	-6	6	26	50	63+1
Posit8.1	5×5	-24	24	50	38	63+1
Posit8.2	4×4	-48	48	98	78	127+1
Posit8.3	3×3	-96	96	194	158	255+1
FP16	11×11	-48	30	80	78	127+1

Comparison with other 8 bits formats: posits and integers

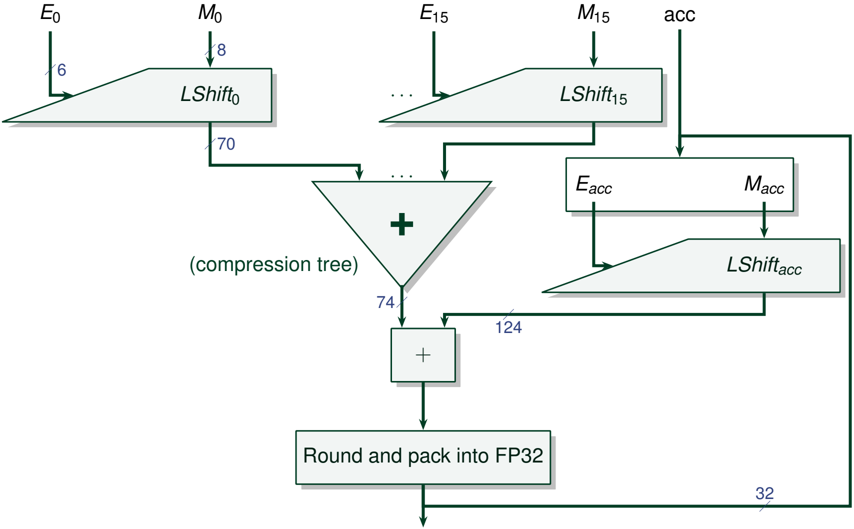


Comparison with other 8 bits formats: posits and integers



How it is implemented in the MPPA

Instead of accumulating in large fixpoint, use a FP32



Conclusion

Back to the 70s

Conclusion

Back to the 70s

- NVIDIA[1] does some sort of truncated Kulisch with their E5M2 and E4M3

---

[a] B. Hickmann and D. Bradford, "Experimental analysis of matrix multiplication functional units", 2019

Conclusion

Back to the 70s

- NVIDIA[1] does some sort of truncated Kulisch with their E5M2 and E4M3
- Intel[2] is similar

---

[b] B. Hickmann, J. Chen, M. Rotzin, A. Yang, M. Urbanski, S. Avancha, "Intel Nervana Neural Network Processor-T (NNP-T) Fused Floating Point Many-Term Dot Product", 2020

Conclusion

Back to the 70s

- NVIDIA[1] does some sort of truncated Kulisch with their E5M2 and E4M3
- Intel[2] is similar
- Arm[3] is now doing some exact Kulisch with rounding by block of 4 products, on E5M2 and E4M3 with scaling factor

---

[c] D. Lutz, A. Saini, M. Kroes, T. Elmer, H. Valsaraju, "Fused FP8 4-Way Dot Product with Scaling and FP32 Accumulation", 2024

Conclusion

Back to the 70s

- NVIDIA[1] does some sort of truncated Kulisch with their E5M2 and E4M3
- Intel[2] is similar
- Arm[3] is now doing some exact Kulisch with rounding by block of 4 products, on E5M2 and E4M3 with scaling factor
- Kalray is continuing the exact Kulisch with rounding by block of 8? products, with E5M2 and E4M3 (but which ones ?)

Conclusion

Back to the 70s

- NVIDIA[1] does some sort of truncated Kulisch with their E5M2 and E4M3
- Intel[2] is similar
- Arm[3] is now doing some exact Kulisch with rounding by block of 4 products, on E5M2 and E4M3 with scaling factor
- Kalray is continuing the exact Kulisch with rounding by block of 8? products, with E5M2 and E4M3 (but which ones ?)
- IEEE WG P3109 formats are a description of everything that can be done