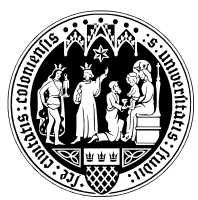
Takum: A New Tapered Precision Machine Number Format

Laslo Hunhold

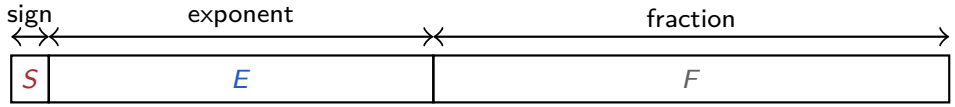
Parallel and Distributed Systems Group
University of Cologne

12th June 2024

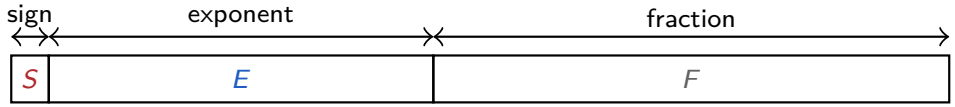


IEEE 754 Floating-Point

versus Desirable Design Criteria (1/2)



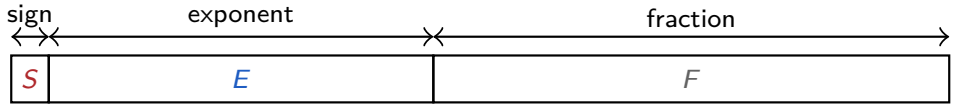
IEEE 754 Floating-Point



versus Desirable Design Criteria (1/2)

- Represented numbers should reflect those used in computations

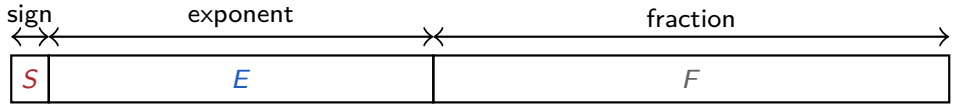
IEEE 754 Floating-Point



versus Desirable Design Criteria (1/2)

- Represented numbers should reflect those used in computations: Observations

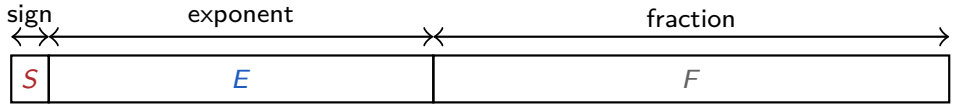
IEEE 754 Floating-Point



versus Desirable Design Criteria (1/2)

- Represented numbers should reflect those used in computations: Observations:
  - Dynamic Range

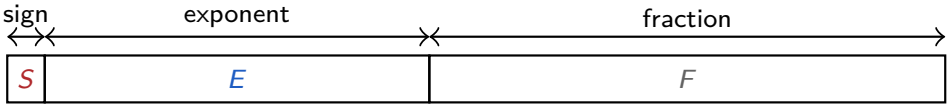
IEEE 754 Floating-Point



versus Desirable Design Criteria (1/2)

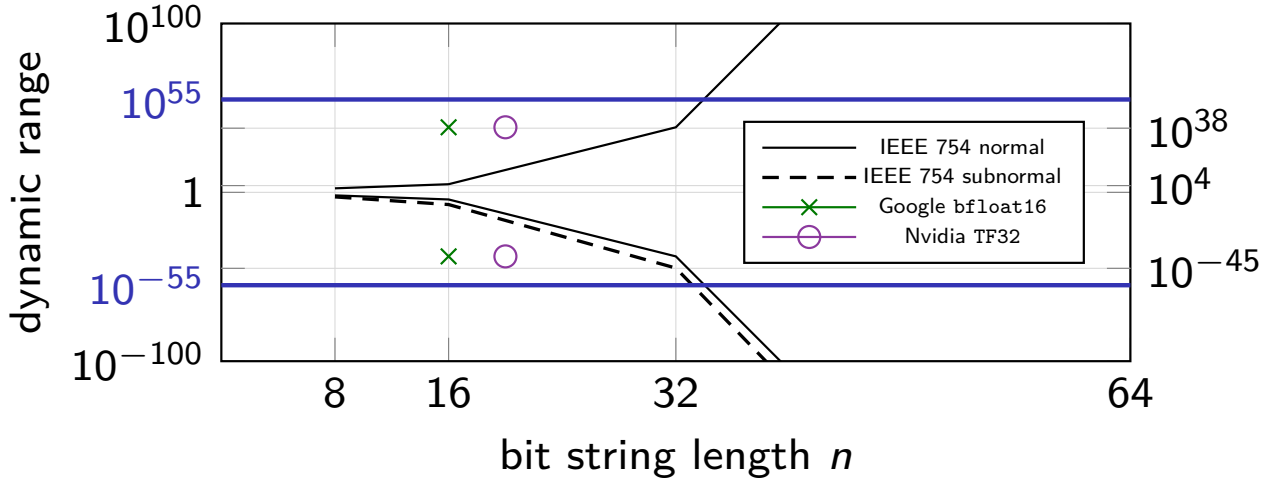
- Represented numbers should reflect those used in computations: Observations:
  - Dynamic Range: General purpose $10^{\pm 55}$ (SI units, constants, etc.)

IEEE 754 Floating-Point

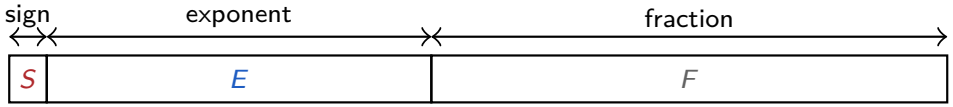


versus Desirable Design Criteria (1/2)

- Represented numbers should reflect those used in computations: Observations:
  - Dynamic Range: General purpose $10^{\pm 55}$ (SI units, constants, etc.)

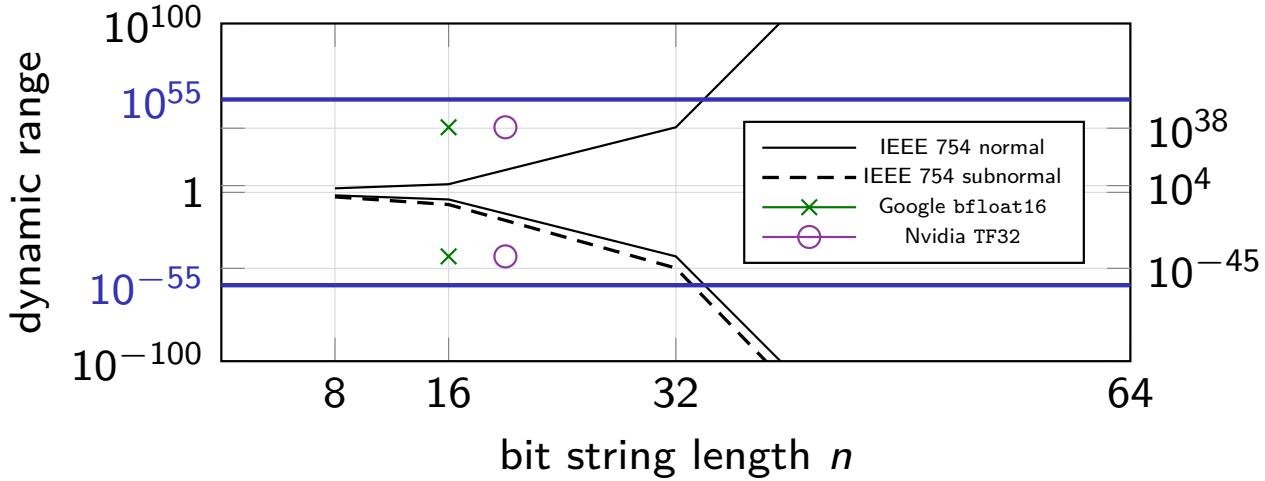


IEEE 754 Floating-Point



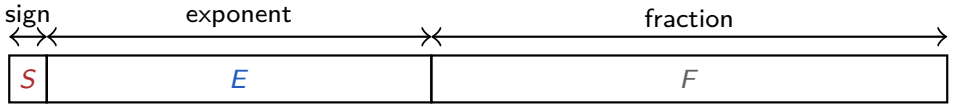
versus Desirable Design Criteria (1/2)

- Represented numbers should reflect those used in computations: Observations:
  - Dynamic Range: General purpose $10^{\pm 55}$ (SI units, constants, etc.)



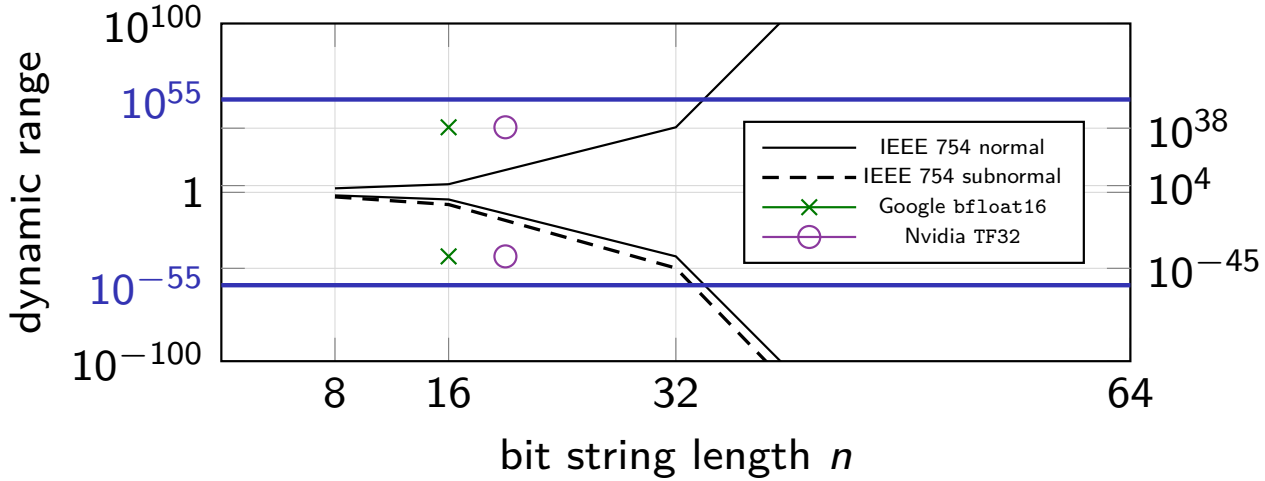
Violated:

IEEE 754 Floating-Point



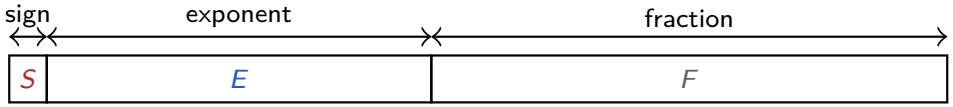
versus Desirable Design Criteria (1/2)

- Represented numbers should reflect those used in computations: Observations:
  - Dynamic Range: General purpose $10^{\pm 55}$ (SI units, constants, etc.)



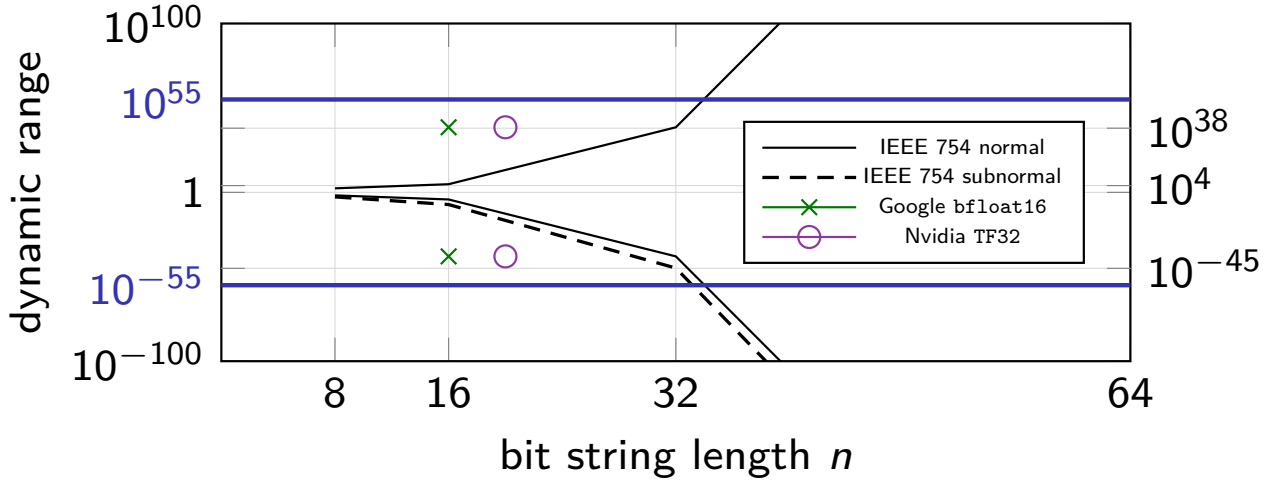
Violated: too small for $n \leq 32$, too large for $n \geq 64$

IEEE 754 Floating-Point



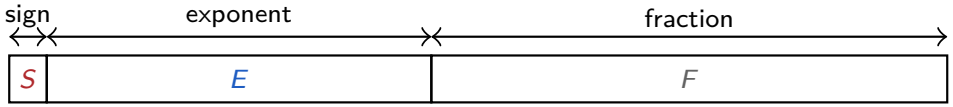
versus Desirable Design Criteria (1/2)

- Represented numbers should reflect those used in computations: Observations:
  - Dynamic Range: General purpose $10^{\pm 55}$ (SI units, constants, etc.)



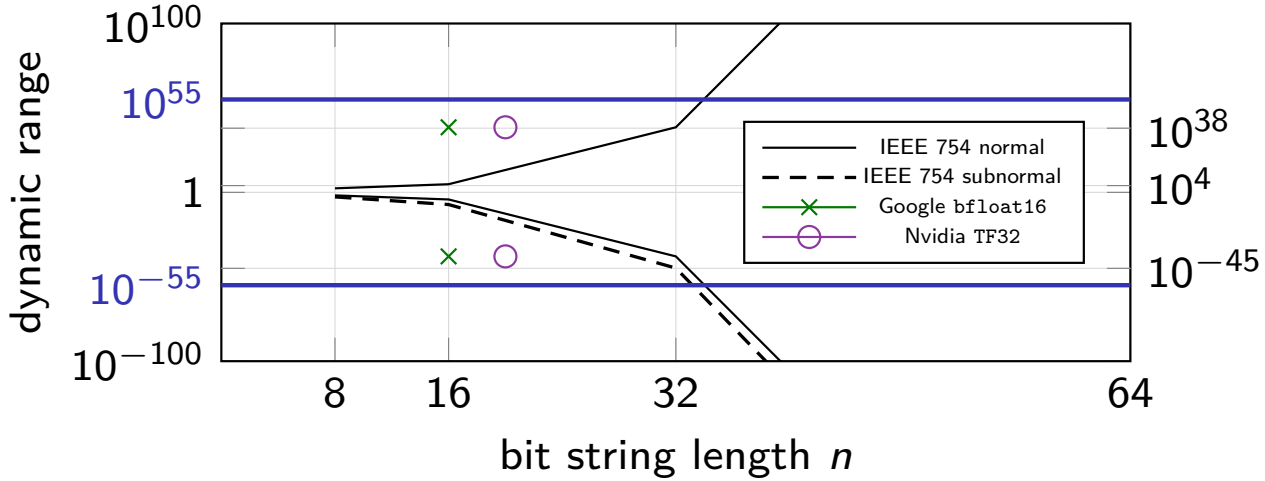
    Violated: too small for $n \leq 32$, too large for $n \geq 64$
  - Tapering

IEEE 754 Floating-Point



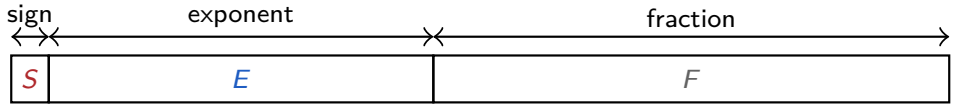
versus Desirable Design Criteria (1/2)

- Represented numbers should reflect those used in computations: Observations:
  - Dynamic Range: General purpose $10^{\pm 55}$ (SI units, constants, etc.)



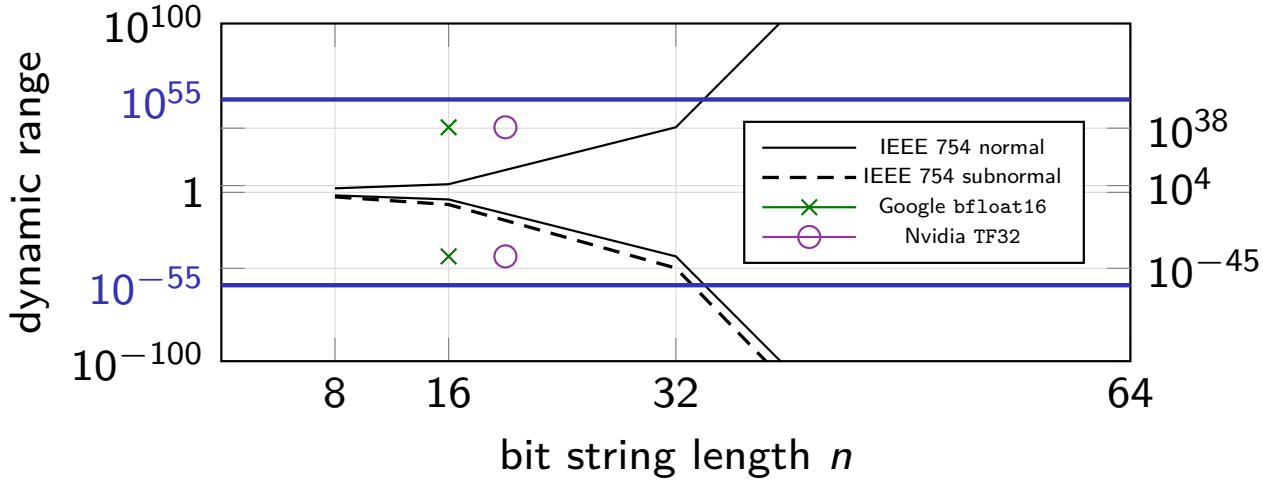
    Violated: too small for $n \leq 32$, too large for $n \geq 64$
  - Tapering: Numbers in magnitude close to one are used much more frequently than very large or very small numbers

IEEE 754 Floating-Point

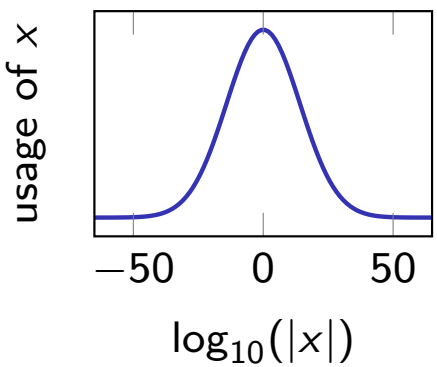


versus Desirable Design Criteria (1/2)

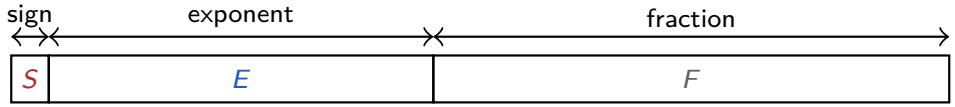
- Represented numbers should reflect those used in computations: Observations:
  - Dynamic Range: General purpose $10^{\pm 55}$ (SI units, constants, etc.)



    Violated: too small for $n \leq 32$, too large for $n \geq 64$
  - Tapering: Numbers in magnitude close to one are used much more frequently than very large or very small numbers

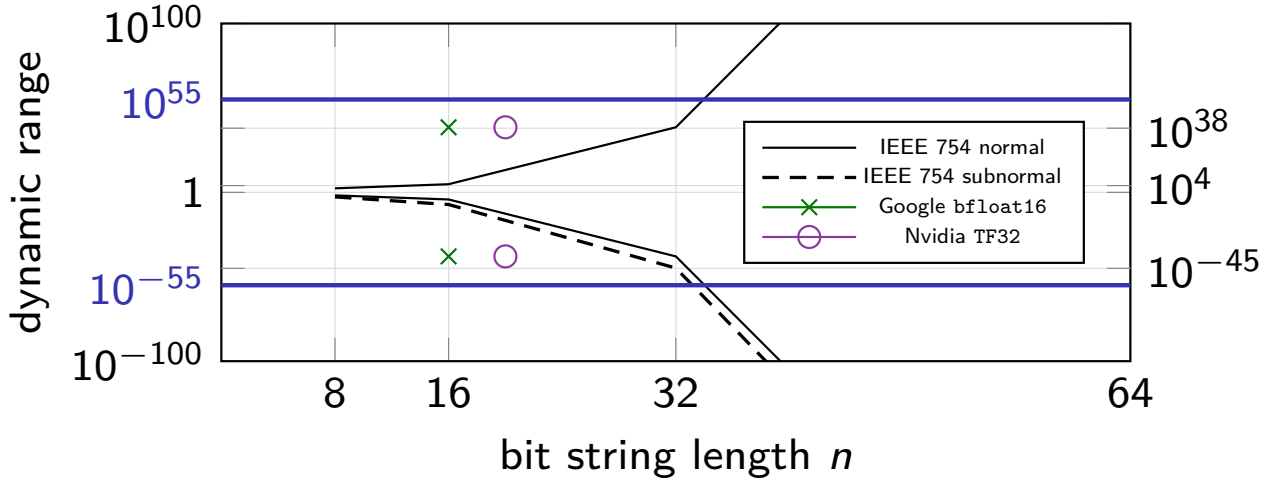


IEEE 754 Floating-Point

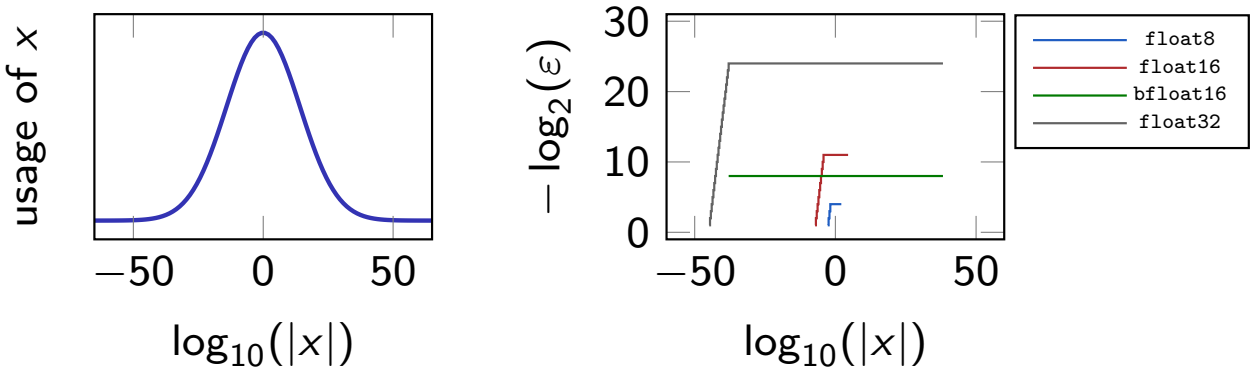


versus Desirable Design Criteria (1/2)

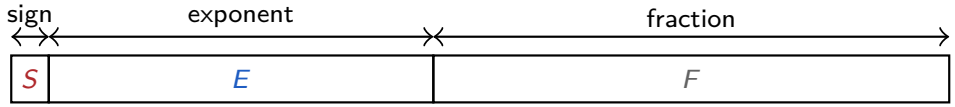
- Represented numbers should reflect those used in computations: Observations:
  - Dynamic Range: General purpose $10^{\pm 55}$ (SI units, constants, etc.)



    Violated: too small for $n \leq 32$, too large for $n \geq 64$
  - Tapering: Numbers in magnitude close to one are used much more frequently than very large or very small numbers

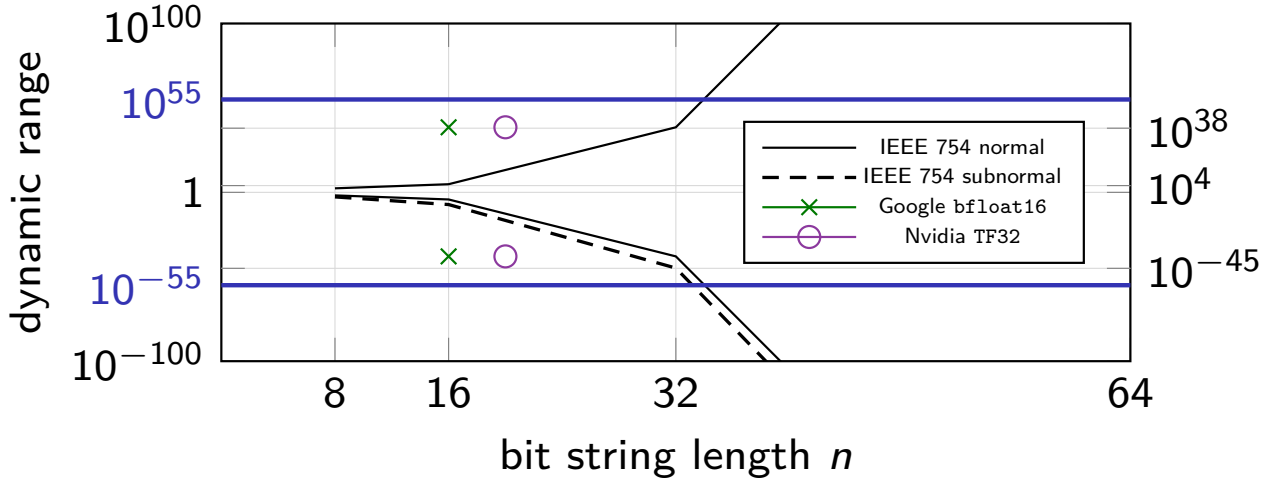


IEEE 754 Floating-Point



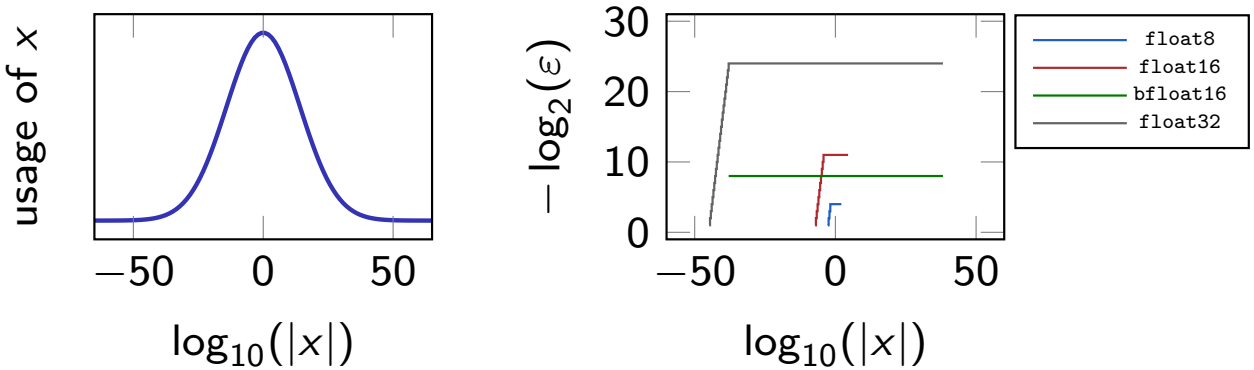
versus Desirable Design Criteria (1/2)

- Represented numbers should reflect those used in computations: Observations:
  - Dynamic Range: General purpose $10^{\pm 55}$ (SI units, constants, etc.)



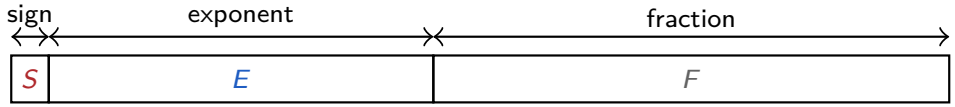
Violated: too small for $n \leq 32$, too large for $n \geq 64$

  - Tapering: Numbers in magnitude close to one are used much more frequently than very large or very small numbers



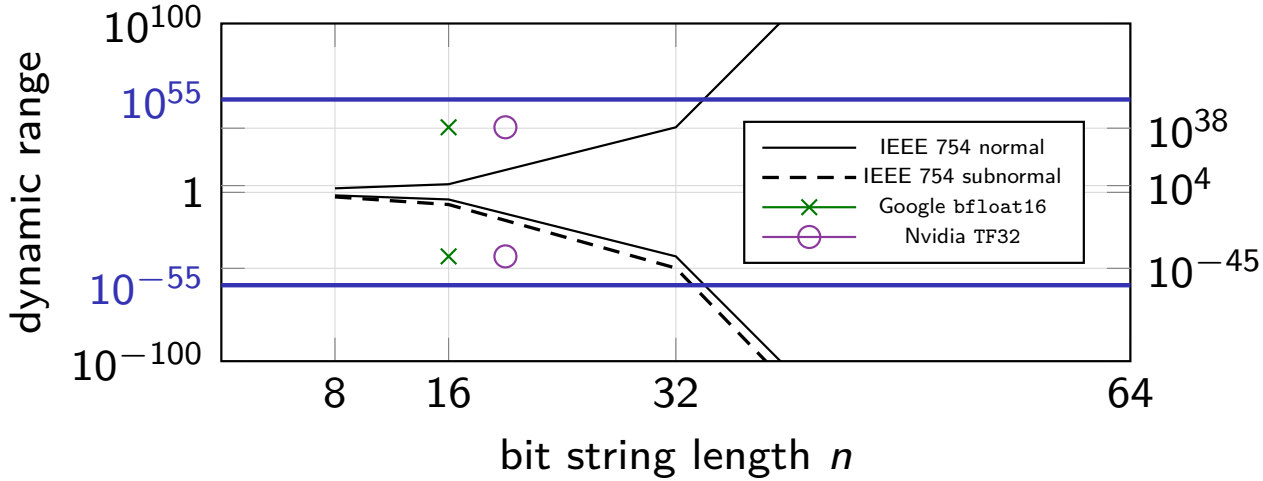
Violated

IEEE 754 Floating-Point



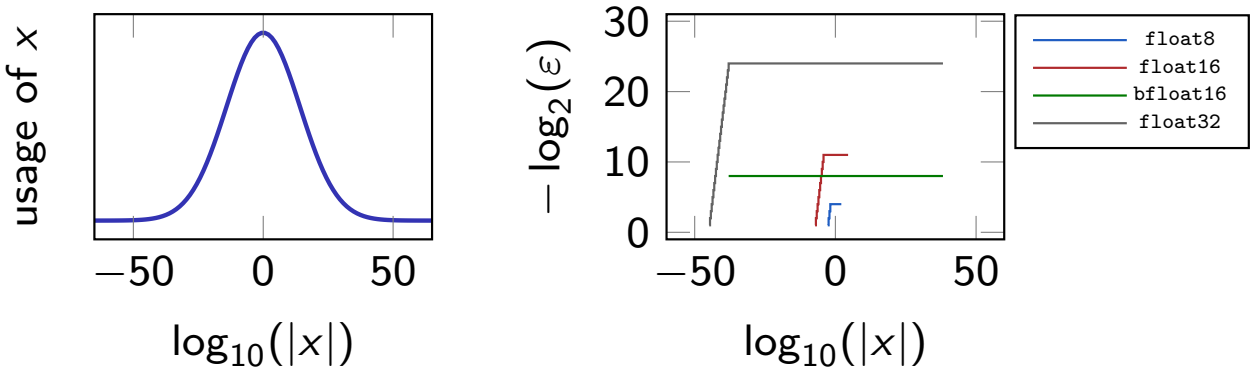
versus Desirable Design Criteria (1/2)

- Represented numbers should reflect those used in computations: Observations:
  - Dynamic Range: General purpose $10^{\pm 55}$ (SI units, constants, etc.)



Violated: too small for $n \leq 32$, too large for $n \geq 64$

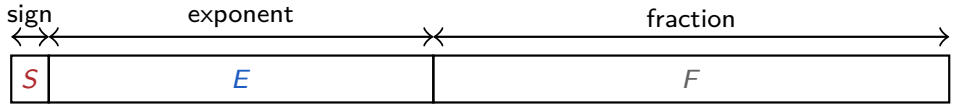
  - Tapering: Numbers in magnitude close to one are used much more frequently than very large or very small numbers



Violated: no tapering, uniform density (except subnormals)

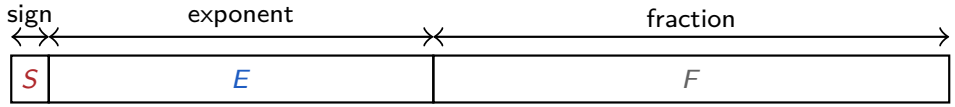
IEEE 754 Floating-Point

versus Desirable Design Criteria (2/2)



IEEE 754 Floating-Point

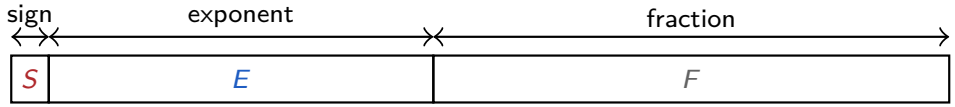
versus Desirable Design Criteria (2/2)



- No redundant encodings

IEEE 754 Floating-Point

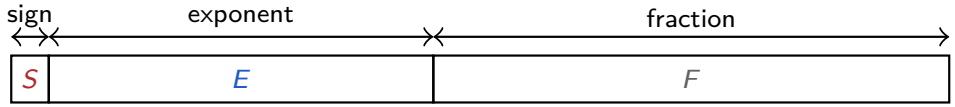
versus Desirable Design Criteria (2/2)



- No redundant encodings
  Violated

IEEE 754 Floating-Point

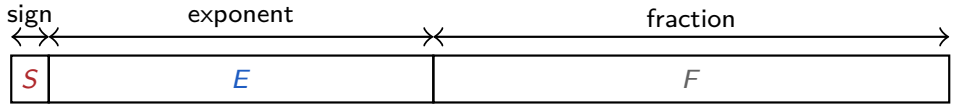
versus Desirable Design Criteria (2/2)



- No redundant encodings
  Violated: signed zero

IEEE 754 Floating-Point

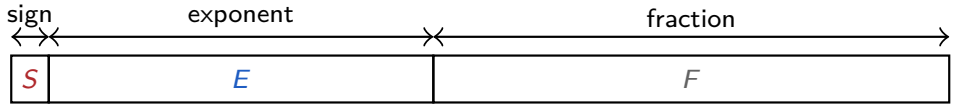
versus Desirable Design Criteria (2/2)



- No redundant encodings
  Violated: signed zero, redundant NaNs

IEEE 754 Floating-Point

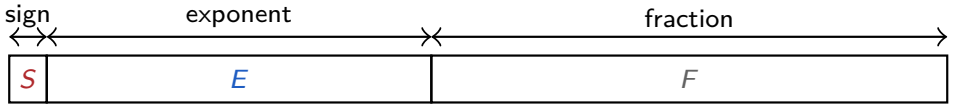
versus Desirable Design Criteria (2/2)



- No redundant encodings
  Violated: signed zero, redundant NaNs, excess numbers ($n \geq 64$)

IEEE 754 Floating-Point

versus Desirable Design Criteria (2/2)

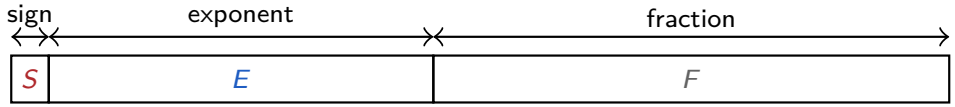


- No redundant encodings
  Violated: signed zero, redundant NaNs, excess numbers ($n \geq 64$)

format	float8	float16	bfloat16	TF32	float32	float64	float128
waste/%	5.08	3.12	0.78	0.78	0.39	82.03	98.88

IEEE 754 Floating-Point

versus Desirable Design Criteria (2/2)



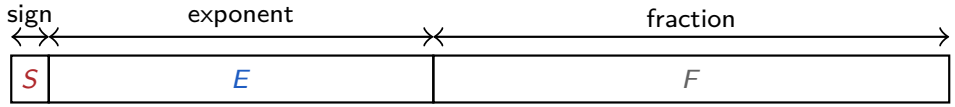
- No redundant encodings
  Violated: signed zero, redundant NaNs, excess numbers ($n \geq 64$)

format	`float8`	`float16`	`bfloat16`	`TF32`	`float32`	`float64`	`float128`
waste/%	5.08	3.12	0.78	0.78	0.39	82.03	98.88

- Defined for bit strings of any length

IEEE 754 Floating-Point

versus Desirable Design Criteria (2/2)



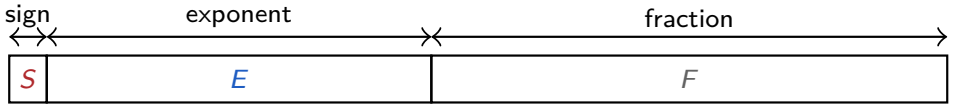
- No redundant encodings
  Violated: signed zero, redundant NaNs, excess numbers ($n \geq 64$)

format	`float8`	`float16`	`bfloat16`	`TF32`	`float32`	`float64`	`float128`
waste/%	5.08	3.12	0.78	0.78	0.39	82.03	98.88

- Defined for bit strings of any length
  Violated

IEEE 754 Floating-Point

versus Desirable Design Criteria (2/2)



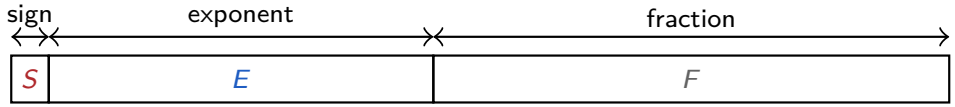
- No redundant encodings
  Violated: signed zero, redundant NaNs, excess numbers ($n \geq 64$)

format	`float8`	`float16`	`bfloat16`	`TF32`	`float32`	`float64`	`float128`
waste/%	5.08	3.12	0.78	0.78	0.39	82.03	98.88

- Defined for bit strings of any length
  Violated: Only defined for $n \in \{8, 16, 32, 64, \dots\}$

IEEE 754 Floating-Point

versus Desirable Design Criteria (2/2)



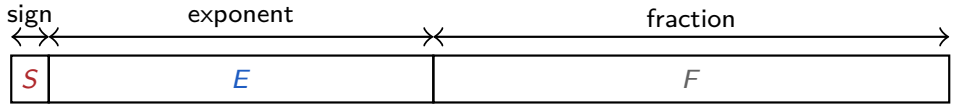
- No redundant encodings
  Violated: signed zero, redundant NaNs, excess numbers ($n \geq 64$)

format	`float8`	`float16`	`bfloat16`	`TF32`	`float32`	`float64`	`float128`
waste/%	5.08	3.12	0.78	0.78	0.39	82.03	98.88

- Defined for bit strings of any length
  Violated: Only defined for $n \in \{8, 16, 32, 64, \dots\}$
- Straightforward conversion between lengths

IEEE 754 Floating-Point

versus Desirable Design Criteria (2/2)



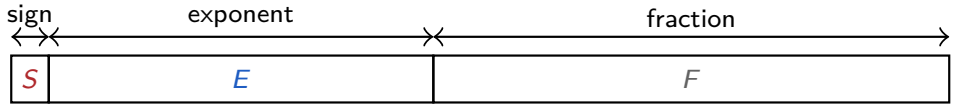
- No redundant encodings
  Violated: signed zero, redundant NaNs, excess numbers ($n \geq 64$)

format	`float8`	`float16`	`bfloat16`	`TF32`	`float32`	`float64`	`float128`
waste/%	5.08	3.12	0.78	0.78	0.39	82.03	98.88

- Defined for bit strings of any length
  Violated: Only defined for $n \in \{8, 16, 32, 64, \dots\}$
- Straightforward conversion between lengths
  Violated

IEEE 754 Floating-Point

versus Desirable Design Criteria (2/2)



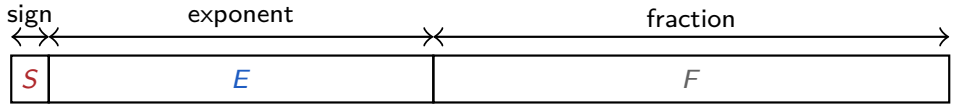
- No redundant encodings
  Violated: signed zero, redundant NaNs, excess numbers ($n \geq 64$)

format	`float8`	`float16`	`bfloat16`	`TF32`	`float32`	`float64`	`float128`
waste/%	5.08	3.12	0.78	0.78	0.39	82.03	98.88

- Defined for bit strings of any length
  Violated: Only defined for $n \in \{8, 16, 32, 64, \dots\}$
- Straightforward conversion between lengths
  Violated: Requires exponent-reencoding

IEEE 754 Floating-Point

versus Desirable Design Criteria (2/2)



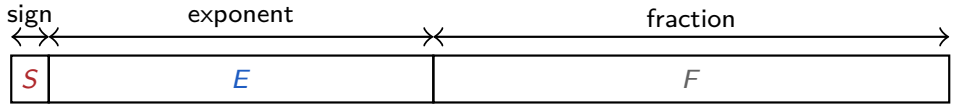
- No redundant encodings
  Violated: signed zero, redundant NaNs, excess numbers ($n \geq 64$)

format	`float8`	`float16`	`bfloat16`	`TF32`	`float32`	`float64`	`float128`
waste/%	5.08	3.12	0.78	0.78	0.39	82.03	98.88

- Defined for bit strings of any length
  Violated: Only defined for $n \in \{8, 16, 32, 64, \ldots\}$
- Straightforward conversion between lengths
  Violated: Requires exponent-reencoding
- Simple/efficient hardware implementation (transistor count, energy efficiency, latency, throughput, etc.)

IEEE 754 Floating-Point

versus Desirable Design Criteria (2/2)



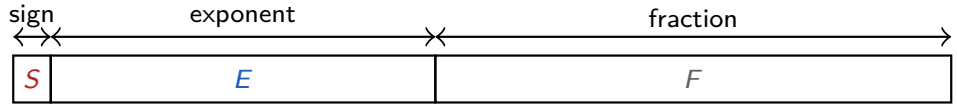
- No redundant encodings
  Violated: signed zero, redundant NaNs, excess numbers ($n \geq 64$)

format	float8	float16	bfloat16	TF32	float32	float64	float128
waste/%	5.08	3.12	0.78	0.78	0.39	82.03	98.88

- Defined for bit strings of any length
  Violated: Only defined for $n \in \{8, 16, 32, 64, \dots\}$
- Straightforward conversion between lengths
  Violated: Requires exponent-reencoding
- Simple/efficient hardware implementation (transistor count, energy efficiency, latency, throughput, etc.)
  Violated

IEEE 754 Floating-Point

versus Desirable Design Criteria (2/2)



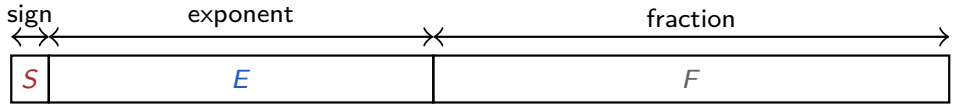
- No redundant encodings
  Violated: signed zero, redundant NaNs, excess numbers ($n \geq 64$)

format	float8	float16	bfloat16	TF32	float32	float64	float128
waste/%	5.08	3.12	0.78	0.78	0.39	82.03	98.88

- Defined for bit strings of any length
  Violated: Only defined for $n \in \{8, 16, 32, 64, \ldots\}$
- Straightforward conversion between lengths
  Violated: Requires exponent-reencoding
- Simple/efficient hardware implementation (transistor count, energy efficiency, latency, throughput, etc.)
  Violated: Special cases complicate hardware

IEEE 754 Floating-Point

versus Desirable Design Criteria (2/2)



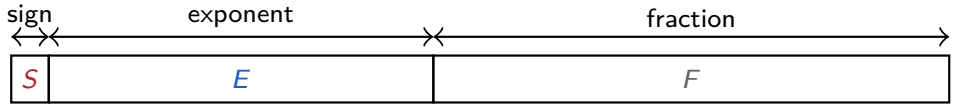
- No redundant encodings
  Violated: signed zero, redundant NaNs, excess numbers ($n \geq 64$)

format	float8	float16	bfloat16	TF32	float32	float64	float128
waste/%	5.08	3.12	0.78	0.78	0.39	82.03	98.88

- Defined for bit strings of any length
  Violated: Only defined for $n \in \{8, 16, 32, 64, \dots\}$
- Straightforward conversion between lengths
  Violated: Requires exponent-reencoding
- Simple/efficient hardware implementation (transistor count, energy efficiency, latency, throughput, etc.)
  Violated: Special cases complicate hardware
  - Subnormals (increase FPU energy consumption by 20%)

IEEE 754 Floating-Point

versus Desirable Design Criteria (2/2)



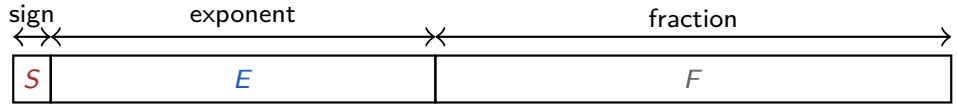
- No redundant encodings
  Violated: signed zero, redundant NaNs, excess numbers ($n \geq 64$)

format	float8	float16	bfloat16	TF32	float32	float64	float128
waste/%	5.08	3.12	0.78	0.78	0.39	82.03	98.88

- Defined for bit strings of any length
  Violated: Only defined for $n \in \{8, 16, 32, 64, \dots\}$
- Straightforward conversion between lengths
  Violated: Requires exponent-reencoding
- Simple/efficient hardware implementation (transistor count, energy efficiency, latency, throughput, etc.)
  Violated: Special cases complicate hardware
  - Subnormals (increase FPU energy consumption by 20%)
  - Signed zero

IEEE 754 Floating-Point

versus Desirable Design Criteria (2/2)



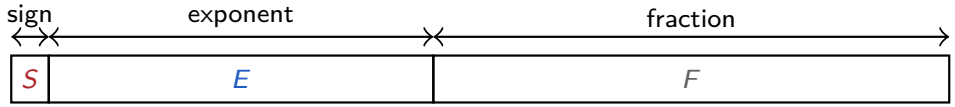
- No redundant encodings
  Violated: signed zero, redundant NaNs, excess numbers ($n \geq 64$)

format	float8	float16	bfloat16	TF32	float32	float64	float128
waste/%	5.08	3.12	0.78	0.78	0.39	82.03	98.88

- Defined for bit strings of any length
  Violated: Only defined for $n \in \{8, 16, 32, 64, \dots\}$
- Straightforward conversion between lengths
  Violated: Requires exponent-reencoding
- Simple/efficient hardware implementation (transistor count, energy efficiency, latency, throughput, etc.)
  Violated: Special cases complicate hardware
  - Subnormals (increase FPU energy consumption by 20%)
  - Signed zero ($0 = -0$

IEEE 754 Floating-Point

versus Desirable Design Criteria (2/2)



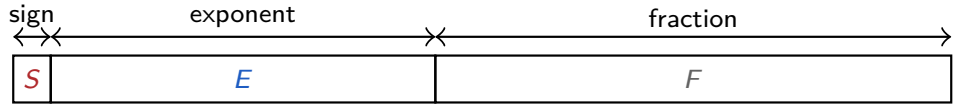
- No redundant encodings
  Violated: signed zero, redundant NaNs, excess numbers ($n \geq 64$)

format	float8	float16	bfloat16	TF32	float32	float64	float128
waste/%	5.08	3.12	0.78	0.78	0.39	82.03	98.88

- Defined for bit strings of any length
  Violated: Only defined for $n \in \{8, 16, 32, 64, \dots\}$
- Straightforward conversion between lengths
  Violated: Requires exponent-reencoding
- Simple/efficient hardware implementation (transistor count, energy efficiency, latency, throughput, etc.)
  Violated: Special cases complicate hardware
  - Subnormals (increase FPU energy consumption by 20%)
  - Signed zero ($0 = -0$, $\sqrt{-0} = -0$

IEEE 754 Floating-Point

versus Desirable Design Criteria (2/2)



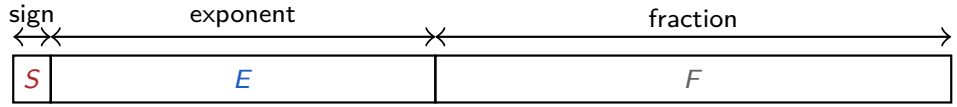
- No redundant encodings
  Violated: signed zero, redundant NaNs, excess numbers ($n \geq 64$)

format	float8	float16	bfloat16	TF32	float32	float64	float128
waste/%	5.08	3.12	0.78	0.78	0.39	82.03	98.88

- Defined for bit strings of any length
  Violated: Only defined for $n \in \{8, 16, 32, 64, \dots\}$
- Straightforward conversion between lengths
  Violated: Requires exponent-reencoding
- Simple/efficient hardware implementation (transistor count, energy efficiency, latency, throughput, etc.)
  Violated: Special cases complicate hardware
  - Subnormals (increase FPU energy consumption by 20%)
  - Signed zero ($0 = -0$, $\sqrt{-0} = -0$, etc.)

IEEE 754 Floating-Point

versus Desirable Design Criteria (2/2)



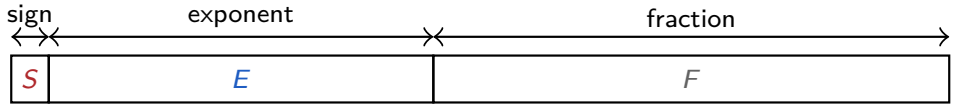
- No redundant encodings
  Violated: signed zero, redundant NaNs, excess numbers ($n \geq 64$)

format	float8	float16	bfloat16	TF32	float32	float64	float128
waste/%	5.08	3.12	0.78	0.78	0.39	82.03	98.88

- Defined for bit strings of any length
  Violated: Only defined for $n \in \{8, 16, 32, 64, \dots\}$
- Straightforward conversion between lengths
  Violated: Requires exponent-reencoding
- Simple/efficient hardware implementation (transistor count, energy efficiency, latency, throughput, etc.)
  Violated: Special cases complicate hardware
  - Subnormals (increase FPU energy consumption by 20%)
  - Signed zero ($0 = -0$, $\sqrt{-0} = -0$, etc.)
  - Signaling/quiet NaN

IEEE 754 Floating-Point

versus Desirable Design Criteria (2/2)



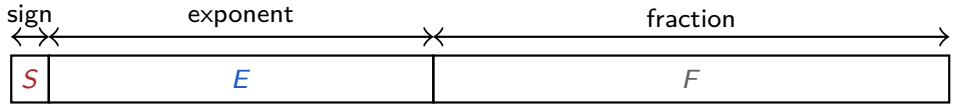
- No redundant encodings
  Violated: signed zero, redundant NaNs, excess numbers ($n \geq 64$)

format	float8	float16	bfloat16	TF32	float32	float64	float128
waste/%	5.08	3.12	0.78	0.78	0.39	82.03	98.88

- Defined for bit strings of any length
  Violated: Only defined for $n \in \{8, 16, 32, 64, \dots\}$
- Straightforward conversion between lengths
  Violated: Requires exponent-reencoding
- Simple/efficient hardware implementation (transistor count, energy efficiency, latency, throughput, etc.)
  Violated: Special cases complicate hardware
  - Subnormals (increase FPU energy consumption by 20%)
  - Signed zero ($0 = -0$, $\sqrt{-0} = -0$, etc.)
  - Signaling/quiet NaN ($\mathrm{NaN} \neq \mathrm{NaN}$

IEEE 754 Floating-Point

versus Desirable Design Criteria (2/2)



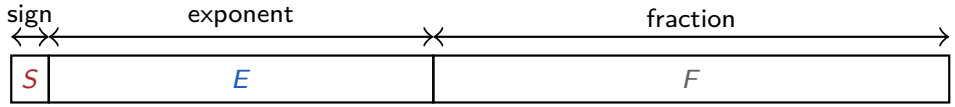
- No redundant encodings
  Violated: signed zero, redundant NaNs, excess numbers ($n \geq 64$)

format	float8	float16	bfloat16	TF32	float32	float64	float128
waste/%	5.08	3.12	0.78	0.78	0.39	82.03	98.88

- Defined for bit strings of any length
  Violated: Only defined for $n \in \{8, 16, 32, 64, \dots\}$
- Straightforward conversion between lengths
  Violated: Requires exponent-reencoding
- Simple/efficient hardware implementation (transistor count, energy efficiency, latency, throughput, etc.)
  Violated: Special cases complicate hardware
  - Subnormals (increase FPU energy consumption by 20%)
  - Signed zero ($0 = -0$, $\sqrt{-0} = -0$, etc.)
  - Signaling/quiet NaN ($\mathrm{NaN} \neq \mathrm{NaN}$, $\mathrm{NaN}^0 = 1$

IEEE 754 Floating-Point

versus Desirable Design Criteria (2/2)



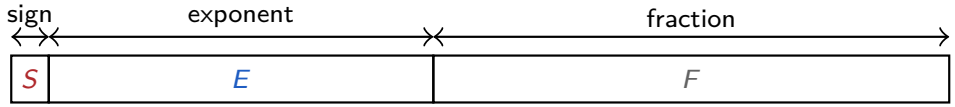
- No redundant encodings
  Violated: signed zero, redundant NaNs, excess numbers ($n \geq 64$)

format	float8	float16	bfloat16	TF32	float32	float64	float128
waste/%	5.08	3.12	0.78	0.78	0.39	82.03	98.88

- Defined for bit strings of any length
  Violated: Only defined for $n \in \{8, 16, 32, 64, \dots\}$
- Straightforward conversion between lengths
  Violated: Requires exponent-reencoding
- Simple/efficient hardware implementation (transistor count, energy efficiency, latency, throughput, etc.)
  Violated: Special cases complicate hardware
  - Subnormals (increase FPU energy consumption by 20%)
  - Signed zero ($0 = -0$, $\sqrt{-0} = -0$, etc.)
  - Signaling/quiet NaN ($\mathrm{NaN} \neq \mathrm{NaN}$, $\mathrm{NaN}^0 = 1$, $1^{\mathrm{NaN}} = 1$

IEEE 754 Floating-Point

versus Desirable Design Criteria (2/2)



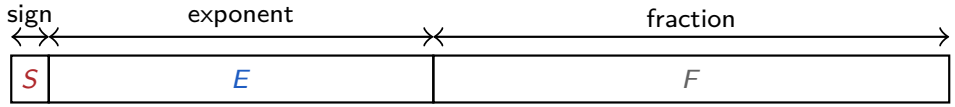
- No redundant encodings
  Violated: signed zero, redundant NaNs, excess numbers ($n \geq 64$)

format	float8	float16	bfloat16	TF32	float32	float64	float128
waste/%	5.08	3.12	0.78	0.78	0.39	82.03	98.88

- Defined for bit strings of any length
  Violated: Only defined for $n \in \{8, 16, 32, 64, \dots\}$
- Straightforward conversion between lengths
  Violated: Requires exponent-reencoding
- Simple/efficient hardware implementation (transistor count, energy efficiency, latency, throughput, etc.)
  Violated: Special cases complicate hardware
  - Subnormals (increase FPU energy consumption by 20%)
  - Signed zero ($0 = -0$, $\sqrt{-0} = -0$, etc.)
  - Signaling/quiet NaN ($\mathrm{NaN} \neq \mathrm{NaN}$, $\mathrm{NaN}^0 = 1$, $1^{\mathrm{NaN}} = 1$, etc.)

IEEE 754 Floating-Point

versus Desirable Design Criteria (2/2)



- No redundant encodings
  Violated: signed zero, redundant NaNs, excess numbers ($n \geq 64$)

format	float8	float16	bfloat16	TF32	float32	float64	float128
waste/%	5.08	3.12	0.78	0.78	0.39	82.03	98.88

- Defined for bit strings of any length
  Violated: Only defined for $n \in \{8, 16, 32, 64, \dots\}$
- Straightforward conversion between lengths
  Violated: Requires exponent-reencoding
- Simple/efficient hardware implementation (transistor count, energy efficiency, latency, throughput, etc.)
  Violated: Special cases complicate hardware
  - Subnormals (increase FPU energy consumption by 20%)
  - Signed zero ($0 = -0$, $\sqrt{-0} = -0$, etc.)
  - Signaling/quiet NaN ($\mathrm{NaN} \neq \mathrm{NaN}$, $\mathrm{NaN}^0 = 1$, $1^{\mathrm{NaN}} = 1$, etc.)
  - Four rounding modes

IEEE 754 Floating-Point

Conclusion

IEEE 754 Floating-Point

Conclusion

End of ‘double precision everywhere’ era

IEEE 754 Floating-Point

Conclusion

End of ‘double precision everywhere’ era

- Deep learning/LLMs

IEEE 754 Floating-Point

Conclusion

End of ‘double precision everywhere’ era

- Deep learning/LLMs
- Mixed Precision

IEEE 754 Floating-Point

Conclusion

End of ‘double precision everywhere’ era

- Deep learning/LLMs
- Mixed Precision
- Hardware/energy constraints (IoT, green HPC)

IEEE 754 Floating-Point

Conclusion

End of ‘double precision everywhere’ era

- Deep learning/LLMs
- Mixed Precision
- Hardware/energy constraints (IoT, green HPC)

The standard is failing

IEEE 754 Floating-Point

Conclusion

End of 'double precision everywhere' era

- Deep learning/LLMs
- Mixed Precision
- Hardware/energy constraints (IoT, green HPC)

The standard is failing

- Not suitable for low precision applications ($n \leq 16$)

IEEE 754 Floating-Point

Conclusion

End of 'double precision everywhere' era

- Deep learning/LLMs
- Mixed Precision
- Hardware/energy constraints (IoT, green HPC)

The standard is failing

- Not suitable for low precision applications ($n \leq 16$)
- Dilution with proprietary formats: `bfloat16` and `TF32`, however still suffer from inherent issues

IEEE 754 Floating-Point

Conclusion

End of 'double precision everywhere' era

- Deep learning/LLMs
- Mixed Precision
- Hardware/energy constraints (IoT, green HPC)

The standard is failing

- Not suitable for low precision applications ($n \leq 16$)
- Dilution with proprietary formats: `bfloat16` and `TF32`, however still suffer from inherent issues
- A lot of new hardware (e.g. GPUs) openly violates the standard

IEEE 754 Floating-Point

Conclusion

End of 'double precision everywhere' era

- Deep learning/LLMs
- Mixed Precision
- Hardware/energy constraints (IoT, green HPC)

The standard is failing

- Not suitable for low precision applications ($n \leq 16$)
- Dilution with proprietary formats: `bfloat16` and `TF32`, however still suffer from inherent issues
- A lot of new hardware (e.g. GPUs) openly violates the standard

The race is on

IEEE 754 Floating-Point

Conclusion

End of 'double precision everywhere' era

- Deep learning/LLMs
- Mixed Precision
- Hardware/energy constraints (IoT, green HPC)

The standard is failing

- Not suitable for low precision applications ($n \leq 16$)
- Dilution with proprietary formats: `bfloat16` and `TF32`, however still suffer from inherent issues
- A lot of new hardware (e.g. GPUs) openly violates the standard

The race is on

- IEEE 754 is a dead end with very high inertia

IEEE 754 Floating-Point

Conclusion

End of 'double precision everywhere' era

- Deep learning/LLMs
- Mixed Precision
- Hardware/energy constraints (IoT, green HPC)

The standard is failing

- Not suitable for low precision applications ($n \leq 16$)
- Dilution with proprietary formats: `bfloat16` and `TF32`, however still suffer from inherent issues
- A lot of new hardware (e.g. GPUs) openly violates the standard

The race is on

- IEEE 754 is a dead end with very high inertia
- Machine Numbers are the backbone of all computations, even small improvements have large high-level effects

IEEE 754 Floating-Point

Conclusion

End of 'double precision everywhere' era

- Deep learning/LLMs
- Mixed Precision
- Hardware/energy constraints (IoT, green HPC)

The standard is failing

- Not suitable for low precision applications ($n \leq 16$)
- Dilution with proprietary formats: `bfloat16` and `TF32`, however still suffer from inherent issues
- A lot of new hardware (e.g. GPUs) openly violates the standard

The race is on

- IEEE 754 is a dead end with very high inertia
- Machine Numbers are the backbone of all computations, even small improvements have large high-level effects
- Time for a new paradigm?

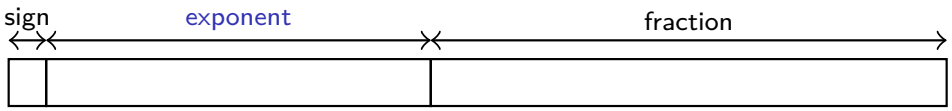
Tapered Precision Machine Numbers

Motivation

Tapered Precision Machine Numbers

Motivation

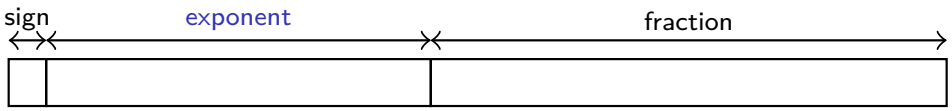
- Consider general floating-point format



Tapered Precision Machine Numbers

Motivation

- Consider general floating-point format

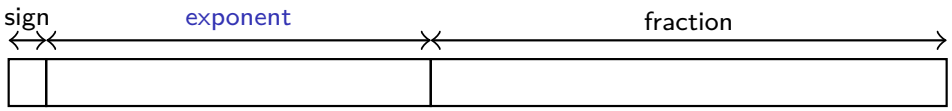


- Sign and fraction bits already contain maximum information

Tapered Precision Machine Numbers

Motivation

- Consider general floating-point format



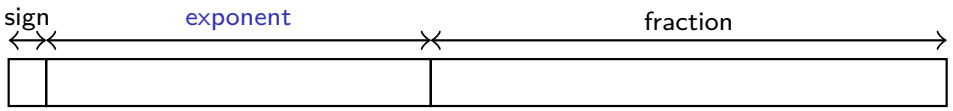
- Sign and fraction bits already contain maximum information

Angle of approach: exponent bits

Tapered Precision Machine Numbers

Motivation

- Consider general floating-point format



- Sign and fraction bits already contain maximum information

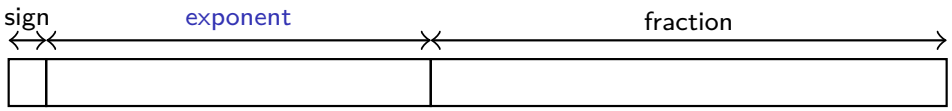
Angle of approach: exponent bits

- Want higher density of numbers (i.e. more fraction bits, density) for exponent values close to zero

Tapered Precision Machine Numbers

Motivation

- Consider general floating-point format



- Sign and fraction bits already contain maximum information

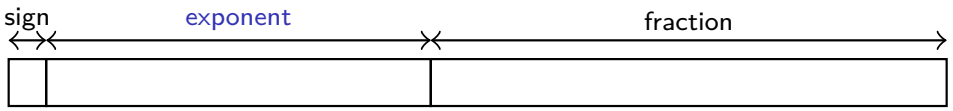
Angle of approach: exponent bits

- Want higher density of numbers (i.e. more fraction bits, density) for exponent values close to zero
- Solution: variable-length exponent encoding

Tapered Precision Machine Numbers

Motivation

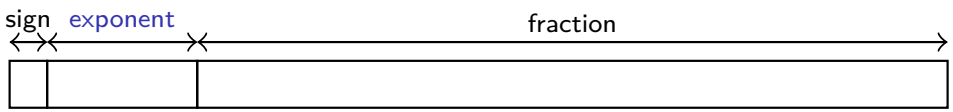
- Consider general floating-point format



- Sign and fraction bits already contain maximum information

Angle of approach: exponent bits

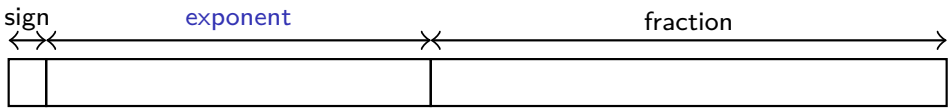
- Want higher density of numbers (i.e. more fraction bits, density) for exponent values close to zero
- Solution: variable-length exponent encoding
  - Small magnitude: shorter exponent, longer fraction, higher density



Tapered Precision Machine Numbers

Motivation

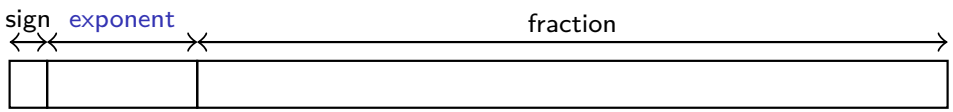
- Consider general floating-point format



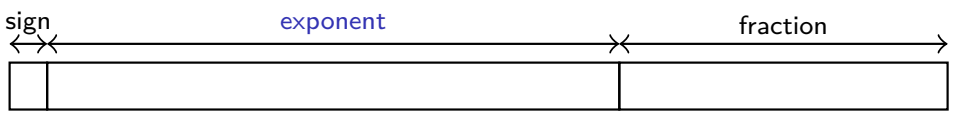
- Sign and fraction bits already contain maximum information

Angle of approach: exponent bits

- Want higher density of numbers (i.e. more fraction bits, density) for exponent values close to zero
- Solution: variable-length exponent encoding
  - Small magnitude: shorter exponent, longer fraction, higher density



  - Large magnitude: longer exponent, shorter fraction, lower density



Posit Arithmetic

Definition

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

Posit Arithmetic

Definition

- State of the art tapered machine number format, in standardisation[1]

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

Posit Arithmetic

Definition

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

Posit Arithmetic

Definition

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

Posit Arithmetic

Definition

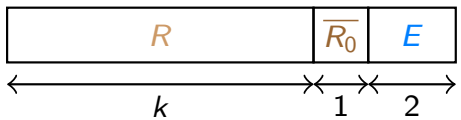
- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

Posit Arithmetic

Definition

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding

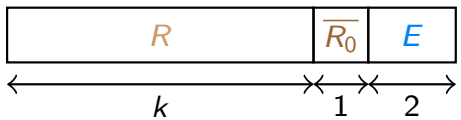


[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

Posit Arithmetic

Definition

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



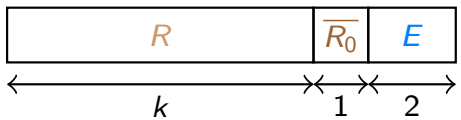
  - R is run of zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

Posit Arithmetic

Definition

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



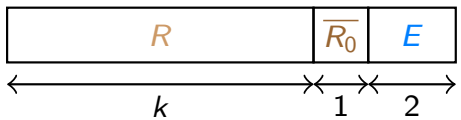
  - R is run of zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

Posit Arithmetic

Definition

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



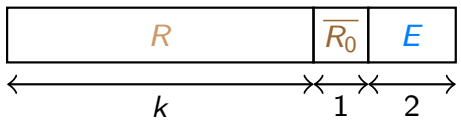
  - R is run of zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$
  - Efficient for small exponents (e.g. $1010 \equiv 2$, $0111 \equiv -1$)

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

Posit Arithmetic

Definition

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



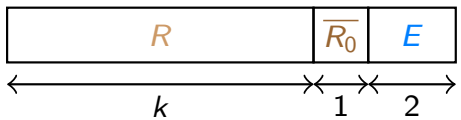
  - R is run of zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$
  - Efficient for small exponents (e.g. $1010 \equiv 2$, $0111 \equiv -1$)
  - Inefficient for large exponents

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

Posit Arithmetic

Definition

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



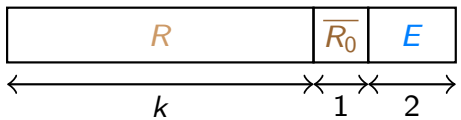
  - R is run of zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$
  - Efficient for small exponents (e.g. $1010 \equiv 2$, $0111 \equiv -1$)
  - Inefficient for large exponents (e.g. 48-bit $111111111111111111111111111111111111111111111011 \equiv 183$)

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

Posit Arithmetic

Definition

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



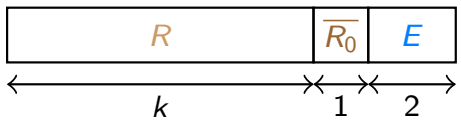
  - R is run of zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$
  - Efficient for small exponents (e.g. $1010 \equiv 2$, $0111 \equiv -1$)
  - Inefficient for large exponents (e.g. 48-bit $111111111111111111111111111111111111111111111011 \equiv 183$)
- Properties

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

Posit Arithmetic

Definition

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



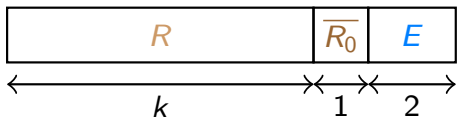
  - R is run of zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$
  - Efficient for small exponents (e.g. $1010 \equiv 2$, $0111 \equiv -1$)
  - Inefficient for large exponents (e.g. 48-bit $111111111111111111111111111111111111111111111011 \equiv 183$)
- Properties
  - No redundant representations

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

Posit Arithmetic

Definition

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



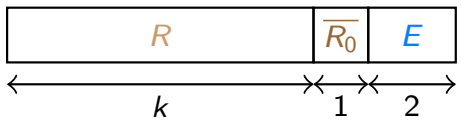
  - R is run of zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$
  - Efficient for small exponents (e.g. $1010 \equiv 2$, $0111 \equiv -1$)
  - Inefficient for large exponents (e.g. 48-bit $111111111111111111111111111111111111111111111011 \equiv 183$)
- Properties
  - No redundant representations
  - Symmetry with two's complement integers

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

Posit Arithmetic

Definition

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



  - R is run of zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$
  - Efficient for small exponents (e.g. $1010 \equiv 2$, $0111 \equiv -1$)
  - Inefficient for large exponents (e.g. 48-bit $111111111111111111111111111111111111111111111011 \equiv 183$)
- Properties
  - No redundant representations
  - Symmetry with two's complement integers (unsigned zero of all zero bits

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

Posit Arithmetic

Definition

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding

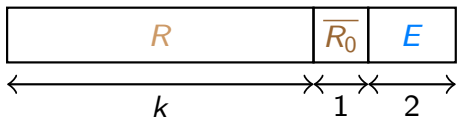
  - R is run of zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$
  - Efficient for small exponents (e.g. $1010 \equiv 2$, $0111 \equiv -1$)
  - Inefficient for large exponents (e.g. 48-bit $111111111111111111111111111111111111111111111011 \equiv 183$)
- Properties
  - No redundant representations
  - Symmetry with two's complement integers (unsigned zero of all zero bits, negation

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

Posit Arithmetic

Definition

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



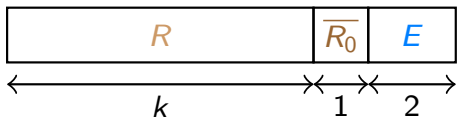
  - R is run of zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$
  - Efficient for small exponents (e.g. $1010 \equiv 2$, $0111 \equiv -1$)
  - Inefficient for large exponents (e.g. 48-bit $111111111111111111111111111111111111111111111011 \equiv 183$)
- Properties
  - No redundant representations
  - Symmetry with two's complement integers (unsigned zero of all zero bits, negation, ordering

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

Posit Arithmetic

Definition

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



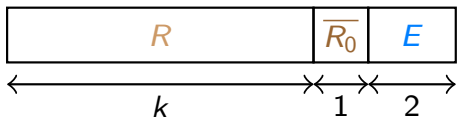
  - R is run of zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$
  - Efficient for small exponents (e.g. $1010 \equiv 2$, $0111 \equiv -1$)
  - Inefficient for large exponents (e.g. 48-bit $111111111111111111111111111111111111111111111011 \equiv 183$)
- Properties
  - No redundant representations
  - Symmetry with two's complement integers (unsigned zero of all zero bits, negation, ordering, sign)

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

Posit Arithmetic

Definition

- State of the art tapered machine number format, in standardisation[1]
- Active research field (hundreds of publications since 2017)
- Numerous implementations, latest by Calligo with RISC-V SoC 'TUNGA' (2024)
- Exponent coding



  - R is run of zeros or ones, followed by one or zero ($\overline{R_0}$) (prefix code)
  - Coded exponent is $e = \begin{cases} -4k + \text{uint}(E) & \overline{R_0} = 0 \\ 4(k-1) + \text{uint}(E) & \overline{R_0} = 1 \end{cases}$
  - Efficient for small exponents (e.g. $1010 \equiv 2$, $0111 \equiv -1$)
  - Inefficient for large exponents (e.g. 48-bit $111111111111111111111111111111111111111111111011 \equiv 183$)
- Properties
  - No redundant representations
  - Symmetry with two's complement integers (unsigned zero of all zero bits, negation, ordering, sign)
  - Defined for all n, conversion between lengths simple rounding or expansion

[1] John L. Gustafson et al. 'Standard for Posit Arithmetic (2022)'. Mar. 2022

Posit Arithmetic

Discussion

Posit Arithmetic

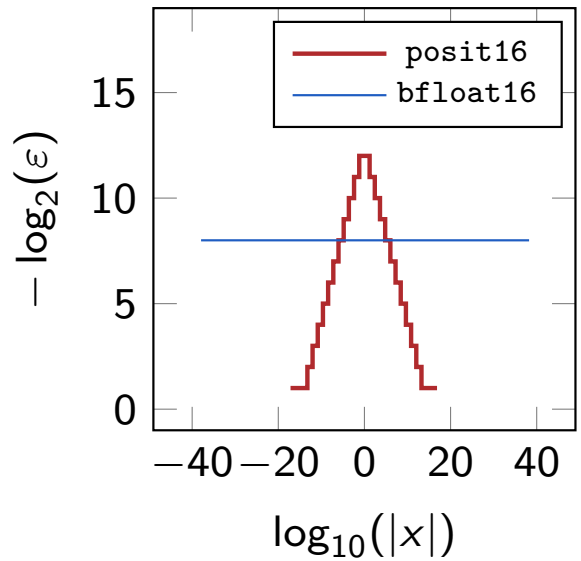
Discussion

Tapering

Posit Arithmetic

Discussion

Tapering

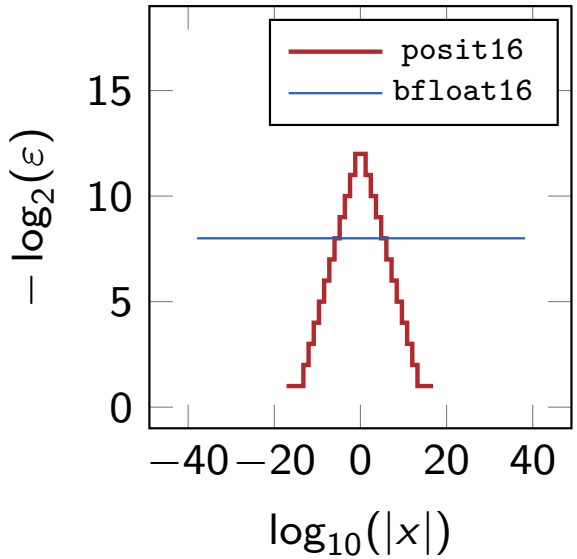


Posit Arithmetic

Discussion

Tapering

- Linear tapering

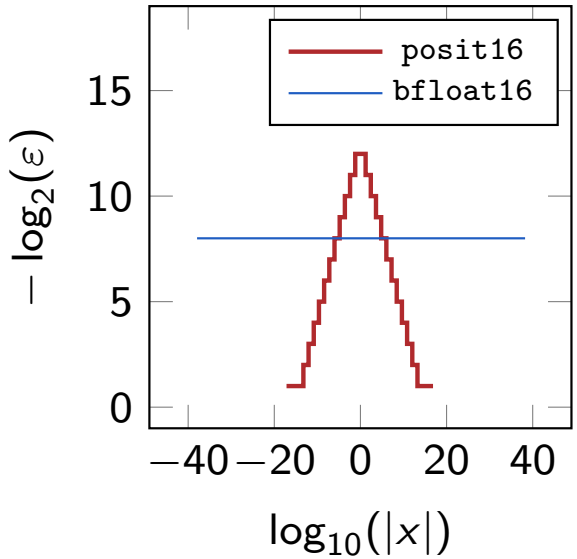


Posit Arithmetic

Discussion

Tapering

- Linear tapering
- Up to 16 times higher density for $n = 16$ compared to `bfloat16`



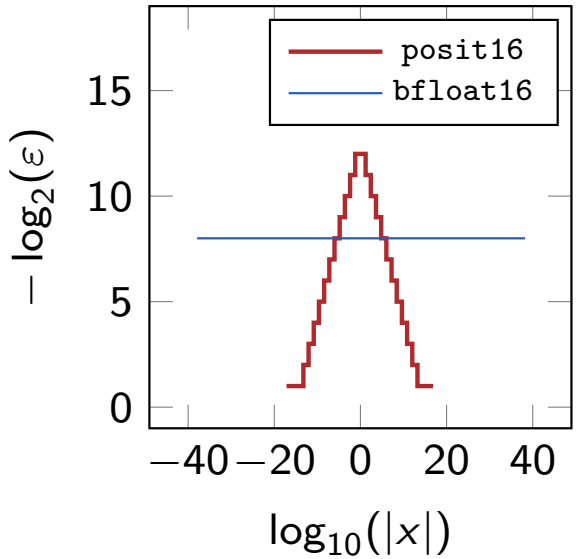
Posit Arithmetic

Discussion

Tapering

- Linear tapering
- Up to 16 times higher density for $n = 16$ compared to `bfloat16`

Dynamic Range

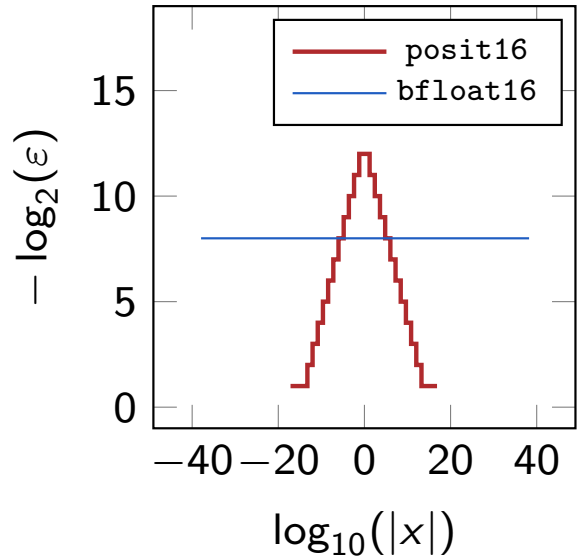


Posit Arithmetic

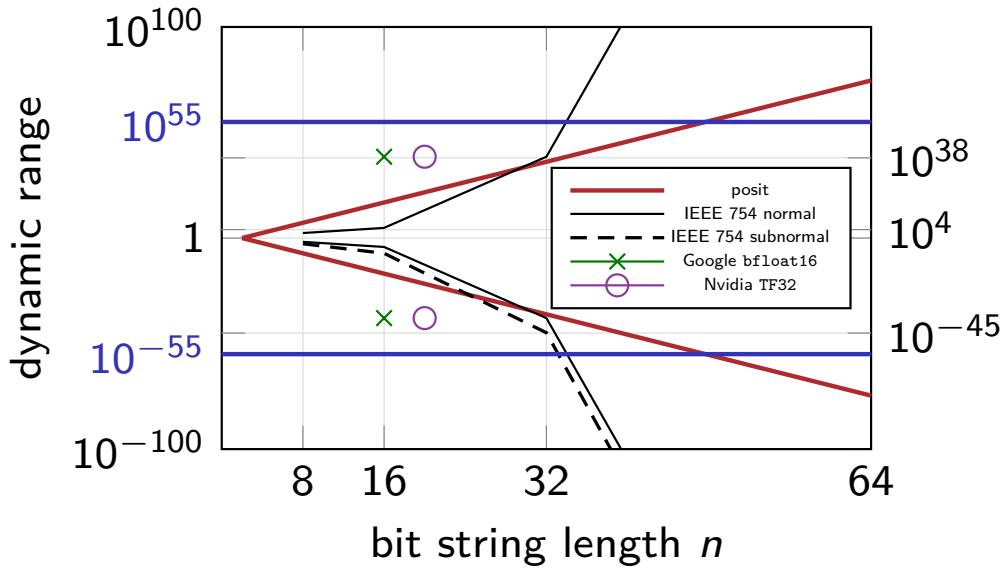
Discussion

Tapering

- Linear tapering
- Up to 16 times higher density for $n = 16$ compared to `bfloat16`



Dynamic Range



Posit Arithmetic

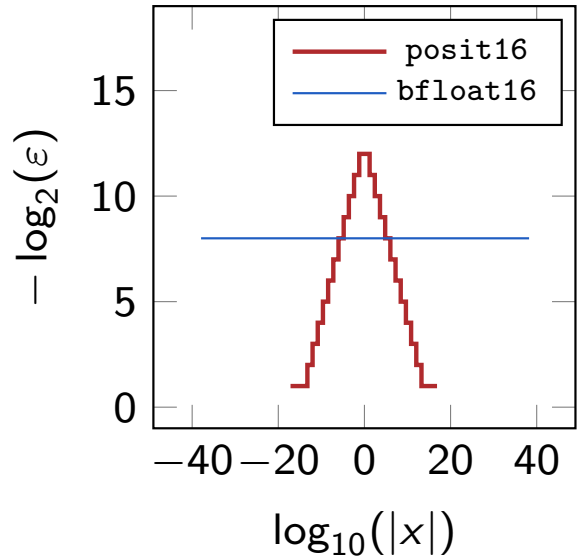
Discussion

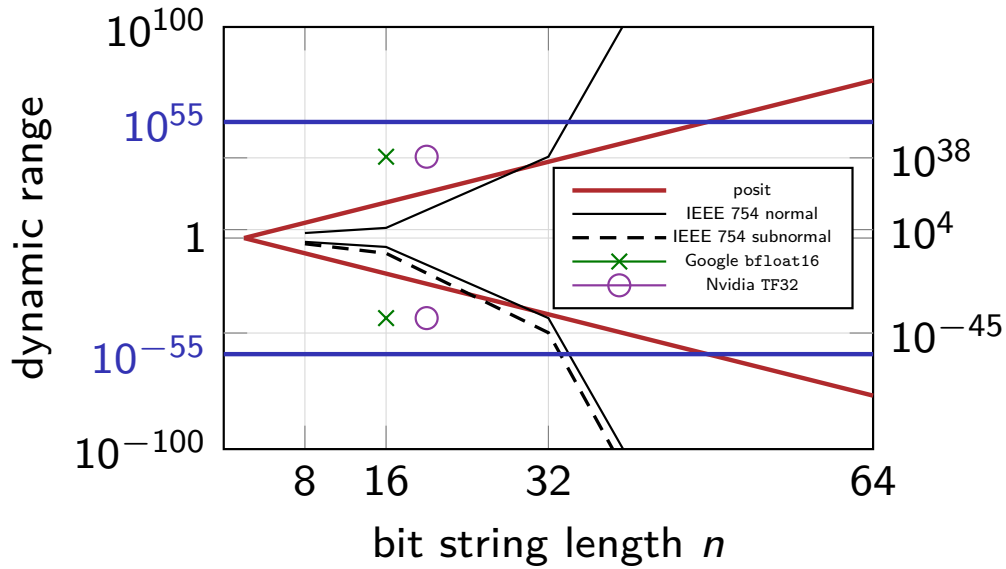
Tapering

- Linear tapering
- Up to 16 times higher density for $n = 16$ compared to `bfloat16`

Dynamic Range

- Very limited dynamic range



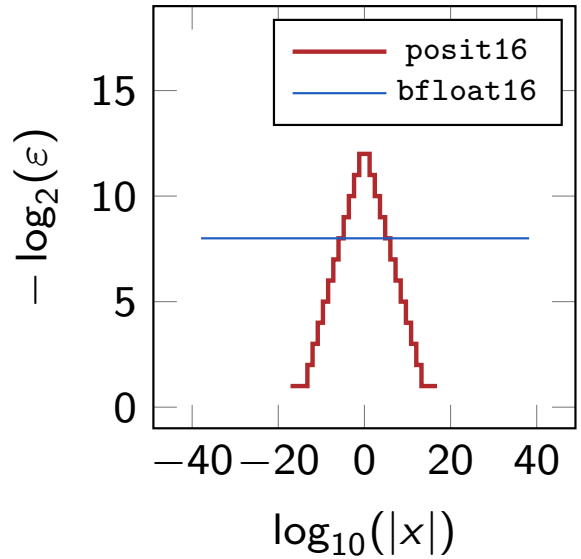


Posit Arithmetic

Discussion

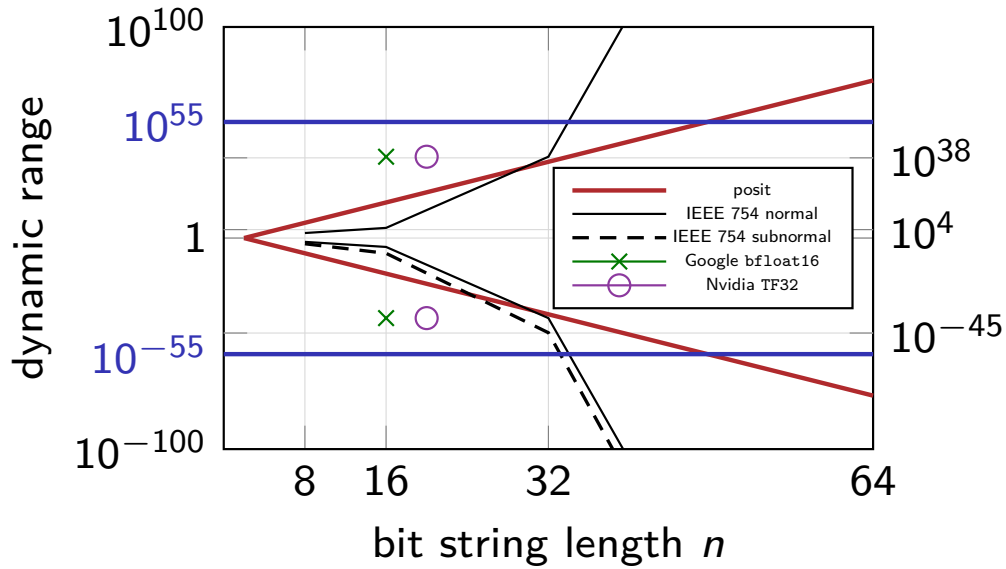
Tapering

- Linear tapering
- Up to 16 times higher density for $n = 16$ compared to `bfloat16`



Dynamic Range

- Very limited dynamic range
- Rejected by Google in favour of `bfloat16` for that reason

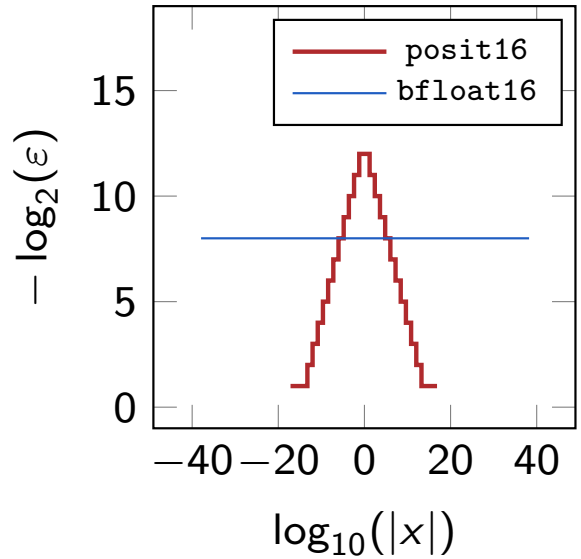


Posit Arithmetic

Discussion

Tapering

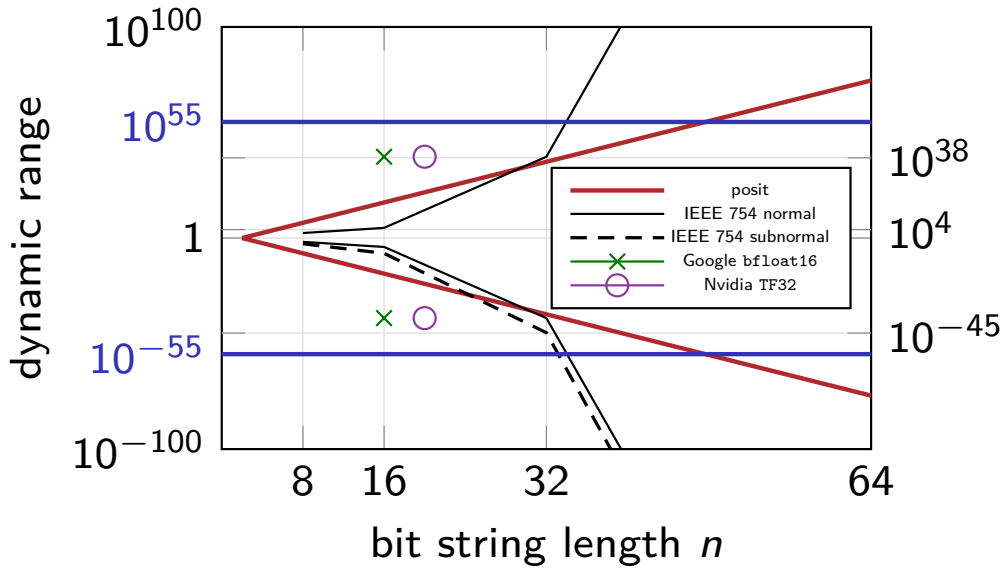
- Linear tapering
- Up to 16 times higher density for $n = 16$ compared to `bfloat16`



Dynamic Range

- Very limited dynamic range
- Rejected by Google in favour of `bfloat16` for that reason

Conclusion

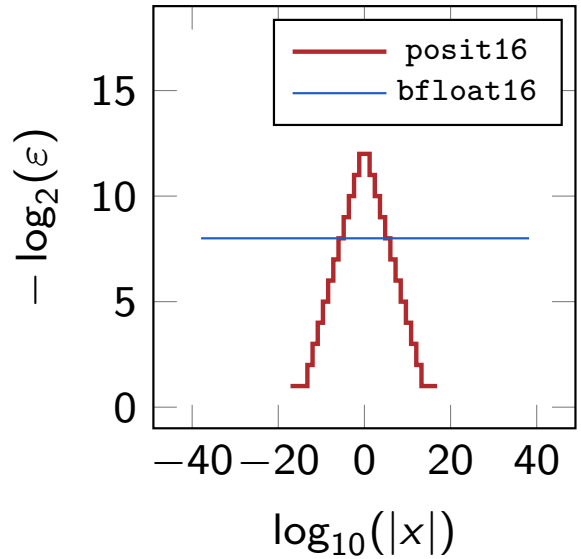


Posit Arithmetic

Discussion

Tapering

- Linear tapering
- Up to 16 times higher density for $n = 16$ compared to `bfloat16`

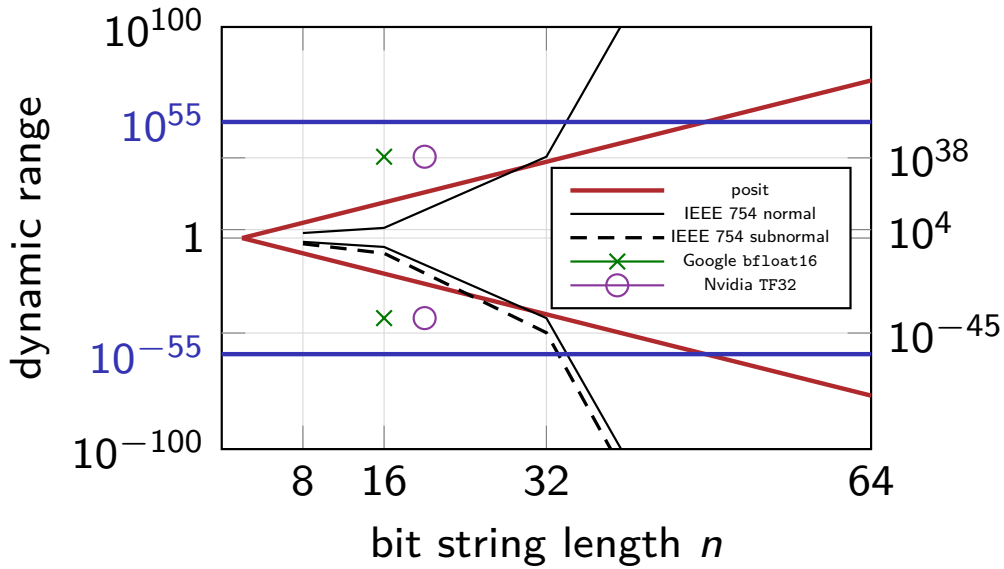


Dynamic Range

- Very limited dynamic range
- Rejected by Google in favour of `bfloat16` for that reason

Conclusion

- interesting properties

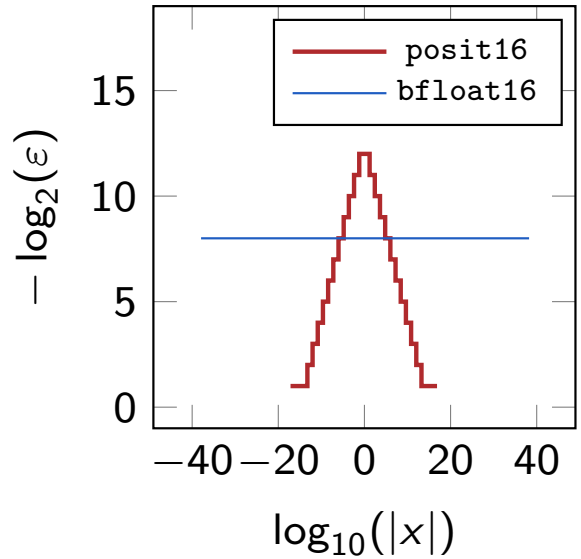


Posit Arithmetic

Discussion

Tapering

- Linear tapering
- Up to 16 times higher density for $n = 16$ compared to `bfloat16`

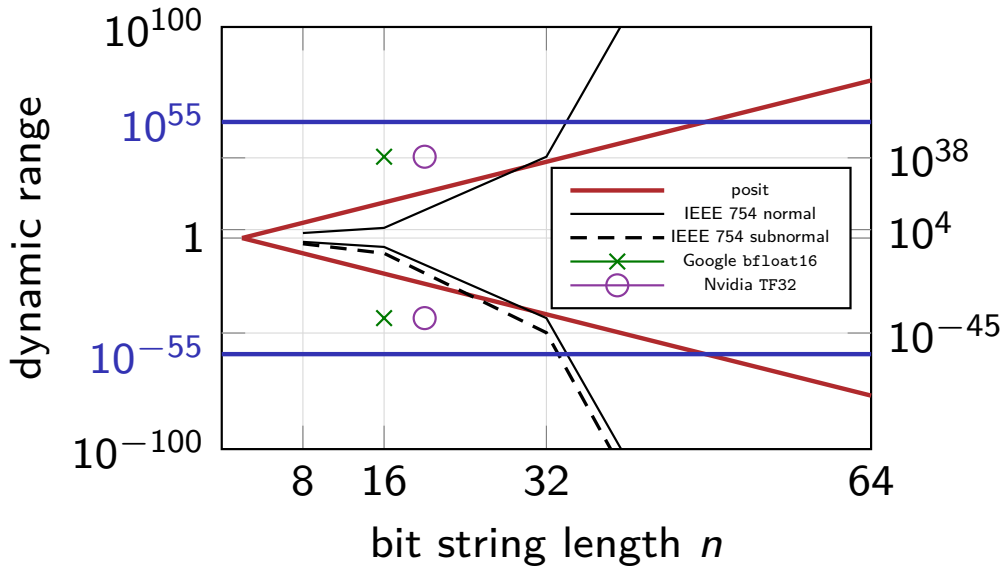


Dynamic Range

- Very limited dynamic range
- Rejected by Google in favour of `bfloat16` for that reason

Conclusion

- interesting properties
- not for general purpose arithmetic



Takum Arithmetic

Overview and Exponent Coding

Takum Arithmetic

Overview and Exponent Coding

- Goals

Takum Arithmetic

Overview and Exponent Coding

- Goals
  - More efficient exponent code

Takum Arithmetic

Overview and Exponent Coding

- Goals
  - More efficient exponent code
  - Logarithmic instead of linear tapering

Takum Arithmetic

Overview and Exponent Coding

- Goals
  - More efficient exponent code
  - Logarithmic instead of linear tapering
  - Preserve useful posit properties

Takum Arithmetic

Overview and Exponent Coding

- Goals
  - More efficient exponent code
  - Logarithmic instead of linear tapering
  - Preserve useful posit properties
- Approach

Takum Arithmetic

Overview and Exponent Coding

- Goals
  - More efficient exponent code
  - Logarithmic instead of linear tapering
  - Preserve useful posit properties
- Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value

Takum Arithmetic

Overview and Exponent Coding

- Goals
  - More efficient exponent code
  - Logarithmic instead of linear tapering
  - Preserve useful posit properties
- Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value
  - Tapered format: Maximum exponent length must also be saturated

Takum Arithmetic

Overview and Exponent Coding

- Goals
  - More efficient exponent code
  - Logarithmic instead of linear tapering
  - Preserve useful posit properties
- Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value
  - Tapered format: Maximum exponent length must also be saturated
  - Take the subsequence $2^{2^k - 1} - 1$ of integers whose bit length is an incremented saturated integer $(3, 15, 255, 65535, 4294967295, \dots)$

Takum Arithmetic

Overview and Exponent Coding

- Goals
  - More efficient exponent code
  - Logarithmic instead of linear tapering
  - Preserve useful posit properties
- Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value
  - Tapered format: Maximum exponent length must also be saturated
  - Take the subsequence $2^{2^k - 1} - 1$ of integers whose bit length is an incremented saturated integer $(3, 15, 255, 65535, 4294967295, \dots)$
  - Maximum exponent value 255 (with bit length 8) follows naturally

Takum Arithmetic

Overview and Exponent Coding

- Goals
  - More efficient exponent code
  - Logarithmic instead of linear tapering
  - Preserve useful posit properties
- Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value
  - Tapered format: Maximum exponent length must also be saturated
  - Take the subsequence $2^{2^k - 1} - 1$ of integers whose bit length is an incremented saturated integer $(3, 15, 255, 65535, 4294967295, \dots)$
  - Maximum exponent value 255 (with bit length 8) follows naturally
- 3-bit regime (0-7) encodes exponent bit count, followed by 0 to 7 exponent bits.

Takum Arithmetic

Overview and Exponent Coding

- Goals
  - More efficient exponent code
  - Logarithmic instead of linear tapering
  - Preserve useful posit properties
- Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value
  - Tapered format: Maximum exponent length must also be saturated
  - Take the subsequence $2^{2^k - 1} - 1$ of integers whose bit length is an incremented saturated integer $(3, 15, 255, 65535, 4294967295, \dots)$
  - Maximum exponent value 255 (with bit length 8) follows naturally
- 3-bit regime (0-7) encodes exponent bit count, followed by 0 to 7 exponent bits.
- Exponent value has implicit leading 1-bit (1-8 bits), subtract 1 for range 0-254

Takum Arithmetic

Overview and Exponent Coding

- Goals
  - More efficient exponent code
  - Logarithmic instead of linear tapering
  - Preserve useful posit properties
- Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value
  - Tapered format: Maximum exponent length must also be saturated
  - Take the subsequence $2^{2^k-1} - 1$ of integers whose bit length is an incremented saturated integer $(3, 15, 255, 65535, 4294967295, \dots)$
  - Maximum exponent value 255 (with bit length 8) follows naturally
- 3-bit regime (0-7) encodes exponent bit count, followed by 0 to 7 exponent bits.
- Exponent value has implicit leading 1-bit (1-8 bits), subtract 1 for range 0-254

value	0	1	2	3	4	...	254
value bits	1	10	11	100	101	...	11111111
encoding	000	0010	0011	01000	01001	...	1111111111

Takum Arithmetic

Overview and Exponent Coding

- Goals
  - More efficient exponent code
  - Logarithmic instead of linear tapering
  - Preserve useful posit properties
- Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value
  - Tapered format: Maximum exponent length must also be saturated
  - Take the subsequence $2^{2^k - 1} - 1$ of integers whose bit length is an incremented saturated integer $(3, 15, 255, 65535, 4294967295, \dots)$
  - Maximum exponent value 255 (with bit length 8) follows naturally
- 3-bit regime (0-7) encodes exponent bit count, followed by 0 to 7 exponent bits.
- Exponent value has implicit leading 1-bit (1-8 bits), subtract 1 for range 0-254

value	0	1	2	3	4	...	254
value bits	1	10	11	100	101	...	11111111
encoding	000	0010	0011	01000	01001	...	1111111111

- Icelandic 'takmarkað umfang', meaning 'limited range'

Takum Arithmetic

Overview and Exponent Coding

- Goals
  - More efficient exponent code
  - Logarithmic instead of linear tapering
  - Preserve useful posit properties
- Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value
  - Tapered format: Maximum exponent length must also be saturated
  - Take the subsequence $2^{2^k-1} - 1$ of integers whose bit length is an incremented saturated integer $(3, 15, 255, 65535, 4294967295, \dots)$
  - Maximum exponent value 255 (with bit length 8) follows naturally
- 3-bit regime (0-7) encodes exponent bit count, followed by 0 to 7 exponent bits.
- Exponent value has implicit leading 1-bit (1-8 bits), subtract 1 for range 0-254

value	0	1	2	3	4	...	254
value bits	1	10	11	100	101	...	11111111
encoding	000	0010	0011	01000	01001	...	1111111111

- Icelandic 'takmarkað umfang', meaning 'limited range'
- Separate 'direction' bit D for exponent sign

Takum Arithmetic

Overview and Exponent Coding

- Goals
  - More efficient exponent code
  - Logarithmic instead of linear tapering
  - Preserve useful posit properties
- Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value
  - Tapered format: Maximum exponent length must also be saturated
  - Take the subsequence $2^{2^k - 1} - 1$ of integers whose bit length is an incremented saturated integer $(3, 15, 255, 65535, 4294967295, \dots)$
  - Maximum exponent value 255 (with bit length 8) follows naturally
- 3-bit regime (0-7) encodes exponent bit count, followed by 0 to 7 exponent bits.
- Exponent value has implicit leading 1-bit (1-8 bits), subtract 1 for range 0-254

value	0	1	2	3	4	...	254
value bits	1	10	11	100	101	...	11111111
encoding	000	0010	0011	01000	01001	...	1111111111

- Icelandic 'takmarkað umfang', meaning 'limited range'
- Separate 'direction' bit D for exponent sign
- Interlude: Comparison for value 254 (posit $\rightarrow$ takum):

Takum Arithmetic

Overview and Exponent Coding

- Goals
  - More efficient exponent code
  - Logarithmic instead of linear tapering
  - Preserve useful posit properties
- Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value
  - Tapered format: Maximum exponent length must also be saturated
  - Take the subsequence $2^{2^k-1} - 1$ of integers whose bit length is an incremented saturated integer $(3, 15, 255, 65535, 4294967295, \dots)$
  - Maximum exponent value 255 (with bit length 8) follows naturally
- 3-bit regime (0-7) encodes exponent bit count, followed by 0 to 7 exponent bits.
- Exponent value has implicit leading 1-bit (1-8 bits), subtract 1 for range 0-254

value	0	1	2	3	4	...	254
value bits	1	10	11	100	101	...	11111111
encoding	000	0010	0011	01000	01001	...	1111111111

- Icelandic 'takmarkað umfang', meaning 'limited range'
- Separate 'direction' bit D for exponent sign
- Interlude: Comparison for value 254 (posit → takum):

11111111111111111111111111111111111111111111111111111111111111010 (67b)

Takum Arithmetic

Overview and Exponent Coding

- Goals
  - More efficient exponent code
  - Logarithmic instead of linear tapering
  - Preserve useful posit properties
- Approach
  - Candidate sequence of 'saturated' integers $2^k - 1$ with $k \in \mathbb{N}_1$ $(1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots)$ for largest exponent value
  - Tapered format: Maximum exponent length must also be saturated
  - Take the subsequence $2^{2^k-1} - 1$ of integers whose bit length is an incremented saturated integer $(3, 15, 255, 65535, 4294967295, \dots)$
  - Maximum exponent value 255 (with bit length 8) follows naturally
- 3-bit regime (0-7) encodes exponent bit count, followed by 0 to 7 exponent bits.
- Exponent value has implicit leading 1-bit (1-8 bits), subtract 1 for range 0-254

value	0	1	2	3	4	...	254
value bits	1	10	11	100	101	...	11111111
encoding	000	0010	0011	01000	01001	...	1111111111

- Icelandic 'takmarkað umfang', meaning 'limited range'
- Separate 'direction' bit D for exponent sign
- Interlude: Comparison for value 254 (posit → takum):

11111111111111111111111111111111111111111111111111111111111111010 (67b)

↓

11111111111 (11b)

Logarithmic Number System

[2] Jeff Johnson. 'Rethinking floating point for deep learning'. arXiv: 1811.01721 (Nov. 2018)

[3] J. Nicholas Coleman and Rizalafande Che Ismail. 'LNS with Co-Transformation Competes with Floating-Point'. In: IEEE Transactions on Computers 65.1 (2016), pp. 136-146. DOI: 10.1109/TC.2015.2409059

[4] Jeff Johnson. 'Efficient, arbitrarily high precision hardware logarithmic arithmetic for linear algebra'. In: 2020 IEEE 27th Symposium on Computer Arithmetic (ARITH). Los Alamitos, CA, USA: IEEE Computer Society, June 2020, pp. 25-32. DOI: 10.1109/ARITH48897.2020.00013

Logarithmic Number System

- Used internally at Meta for AI-ASICs (MTIA)

[2] Jeff Johnson. 'Rethinking floating point for deep learning'. arXiv: 1811.01721 (Nov. 2018)

[3] J. Nicholas Coleman and Rizalafande Che Ismail. 'LNS with Co-Transformation Competes with Floating-Point'. In: IEEE Transactions on Computers 65.1 (2016), pp. 136-146. DOI: 10.1109/TC.2015.2409059

[4] Jeff Johnson. 'Efficient, arbitrarily high precision hardware logarithmic arithmetic for linear algebra'. In: 2020 IEEE 27th Symposium on Computer Arithmetic (ARITH). Los Alamitos, CA, USA: IEEE Computer Society, June 2020, pp. 25-32. DOI: 10.1109/ARITH48897.2020.00013

Logarithmic Number System

- Used internally at Meta for AI-ASICs (MTIA)
- Instead of $(-1)^s(1+f)\cdot 2^e$ express numbers as $(-1)^s b^{e+f} =: (-1)^s b^{\ell}$ for basis $b > 1$ (usually $b = 2$)

[2] Jeff Johnson. 'Rethinking floating point for deep learning'. arXiv: 1811.01721 (Nov. 2018)

[3] J. Nicholas Coleman and Rizalafande Che Ismail. 'LNS with Co-Transformation Competes with Floating-Point'. In: IEEE Transactions on Computers 65.1 (2016), pp. 136-146. DOI: 10.1109/TC.2015.2409059

[4] Jeff Johnson. 'Efficient, arbitrarily high precision hardware logarithmic arithmetic for linear algebra'. In: 2020 IEEE 27th Symposium on Computer Arithmetic (ARITH). Los Alamitos, CA, USA: IEEE Computer Society, June 2020, pp. 25-32. DOI: 10.1109/ARITH48897.2020.00013

Logarithmic Number System

- Used internally at Meta for AI-ASICs (MTIA)
- Instead of $(-1)^s(1+f)\cdot 2^e$ express numbers as $(-1)^s b^{e+f} =: (-1)^s b^{\ell}$ for basis $b > 1$ (usually $b = 2$)
- ℓ is fixed-point number, exponent $e \rightarrow$ characteristic c, fraction $f \rightarrow$ mantissa m

[2] Jeff Johnson. 'Rethinking floating point for deep learning'. arXiv: 1811.01721 (Nov. 2018)

[3] J. Nicholas Coleman and Rizalafande Che Ismail. 'LNS with Co-Transformation Competes with Floating-Point'. In: IEEE Transactions on Computers 65.1 (2016), pp. 136-146. DOI: 10.1109/TC.2015.2409059

[4] Jeff Johnson. 'Efficient, arbitrarily high precision hardware logarithmic arithmetic for linear algebra'. In: 2020 IEEE 27th Symposium on Computer Arithmetic (ARITH). Los Alamitos, CA, USA: IEEE Computer Society, June 2020, pp. 25-32. DOI: 10.1109/ARITH48897.2020.00013

Logarithmic Number System

- Used internally at Meta for AI-ASICs (MTIA)
- Instead of $(-1)^s(1+f)\cdot 2^e$ express numbers as $(-1)^s b^{e+f} =: (-1)^s b^{\ell}$ for basis $b > 1$ (usually $b = 2$)
- ℓ is fixed-point number, exponent $e \rightarrow$ characteristic c, fraction $f \rightarrow$ mantissa m
- Advantages

[2] Jeff Johnson. 'Rethinking floating point for deep learning'. arXiv: 1811.01721 (Nov. 2018)

[3] J. Nicholas Coleman and Rizalafande Che Ismail. 'LNS with Co-Transformation Competes with Floating-Point'. In: IEEE Transactions on Computers 65.1 (2016), pp. 136-146. DOI: 10.1109/TC.2015.2409059

[4] Jeff Johnson. 'Efficient, arbitrarily high precision hardware logarithmic arithmetic for linear algebra'. In: 2020 IEEE 27th Symposium on Computer Arithmetic (ARITH). Los Alamitos, CA, USA: IEEE Computer Society, June 2020, pp. 25-32. DOI: 10.1109/ARITH48897.2020.00013

Logarithmic Number System

- Used internally at Meta for AI-ASICs (MTIA)
- Instead of $(-1)^s(1+f)\cdot 2^e$ express numbers as $(-1)^s b^{e+f} =: (-1)^s b^{\ell}$ for basis $b > 1$ (usually $b = 2$)
- ℓ is fixed-point number, exponent $e \to$ characteristic c, fraction $f \to$ mantissa m
- Advantages
  - Multiplication, Division, Square Root, Squaring, Inversion are simple fixed-point addition, subtraction, right and left shifts and negation, e.g.

$$b^{\ell_1}\cdot b^{\ell_2} = b^{\ell_1+\ell_2},\ \sqrt{b^{\ell}} = b^{\ell/2},\ \left(b^{\ell}\right)^{-1} = b^{-\ell}$$

[2] Jeff Johnson. 'Rethinking floating point for deep learning'. arXiv: 1811.01721 (Nov. 2018)

[3] J. Nicholas Coleman and Rizalafande Che Ismail. 'LNS with Co-Transformation Competes with Floating-Point'. In: IEEE Transactions on Computers 65.1 (2016), pp. 136-146. DOI: 10.1109/TC.2015.2409059

[4] Jeff Johnson. 'Efficient, arbitrarily high precision hardware logarithmic arithmetic for linear algebra'. In: 2020 IEEE 27th Symposium on Computer Arithmetic (ARITH). Los Alamitos, CA, USA: IEEE Computer Society, June 2020, pp. 25-32. DOI: 10.1109/ARITH48897.2020.00013

Logarithmic Number System

- Used internally at Meta for AI-ASICs (MTIA)
- Instead of $(-1)^s(1+f)\cdot 2^e$ express numbers as $(-1)^s b^{e+f} =: (-1)^s b^{\ell}$ for basis $b > 1$ (usually $b = 2$)
- ℓ is fixed-point number, exponent $e \to$ characteristic c, fraction $f \to$ mantissa m
- Advantages
  - Multiplication, Division, Square Root, Squaring, Inversion are simple fixed-point addition, subtraction, right and left shifts and negation, e.g.
    $$b^{\ell_1}\cdot b^{\ell_2} = b^{\ell_1+\ell_2},\ \sqrt{b^{\ell}} = b^{\ell/2},\ \left(b^{\ell}\right)^{-1} = b^{-\ell}$$
  - Lower relative approximation error

[2] Jeff Johnson. 'Rethinking floating point for deep learning'. arXiv: 1811.01721 (Nov. 2018)

[3] J. Nicholas Coleman and Rizalafande Che Ismail. 'LNS with Co-Transformation Competes with Floating-Point'. In: IEEE Transactions on Computers 65.1 (2016), pp. 136-146. DOI: 10.1109/TC.2015.2409059

[4] Jeff Johnson. 'Efficient, arbitrarily high precision hardware logarithmic arithmetic for linear algebra'. In: 2020 IEEE 27th Symposium on Computer Arithmetic (ARITH). Los Alamitos, CA, USA: IEEE Computer Society, June 2020, pp. 25-32. DOI: 10.1109/ARITH48897.2020.00013

Logarithmic Number System

- Used internally at Meta for AI-ASICs (MTIA)
- Instead of $(-1)^s(1+f)\cdot 2^e$ express numbers as $(-1)^s b^{e+f} =: (-1)^s b^{\ell}$ for basis $b > 1$ (usually $b = 2$)
- ℓ is fixed-point number, exponent $e \rightarrow$ characteristic c, fraction $f \rightarrow$ mantissa m
- Advantages
  - Multiplication, Division, Square Root, Squaring, Inversion are simple fixed-point addition, subtraction, right and left shifts and negation, e.g.
    $$b^{\ell_1}\cdot b^{\ell_2} = b^{\ell_1+\ell_2},\ \sqrt{b^{\ell}} = b^{\ell/2},\ \left(b^{\ell}\right)^{-1} = b^{-\ell}$$
  - Lower relative approximation error
  - Simpler hardware, more energy efficient, especially for AI[2]

[2] Jeff Johnson. 'Rethinking floating point for deep learning'. arXiv: 1811.01721 (Nov. 2018)

[3] J. Nicholas Coleman and Rizalafande Che Ismail. 'LNS with Co-Transformation Competes with Floating-Point'. In: IEEE Transactions on Computers 65.1 (2016), pp. 136-146. DOI: 10.1109/TC.2015.2409059

[4] Jeff Johnson. 'Efficient, arbitrarily high precision hardware logarithmic arithmetic for linear algebra'. In: 2020 IEEE 27th Symposium on Computer Arithmetic (ARITH). Los Alamitos, CA, USA: IEEE Computer Society, June 2020, pp. 25-32. DOI: 10.1109/ARITH48897.2020.00013

Logarithmic Number System

- Used internally at Meta for AI-ASICs (MTIA)
- Instead of $(-1)^s(1+f)\cdot 2^e$ express numbers as $(-1)^s b^{e+f} =: (-1)^s b^{\ell}$ for basis $b > 1$ (usually $b = 2$)
- ℓ is fixed-point number, exponent $e \rightarrow$ characteristic c, fraction $f \rightarrow$ mantissa m
- Advantages
  - Multiplication, Division, Square Root, Squaring, Inversion are simple fixed-point addition, subtraction, right and left shifts and negation, e.g.
    $$b^{\ell_1}\cdot b^{\ell_2} = b^{\ell_1+\ell_2},\ \sqrt{b^{\ell}} = b^{\ell/2},\ \left(b^{\ell}\right)^{-1} = b^{-\ell}$$
  - Lower relative approximation error
  - Simpler hardware, more energy efficient, especially for AI[2]
- Disadvantages

[2] Jeff Johnson. 'Rethinking floating point for deep learning'. arXiv: 1811.01721 (Nov. 2018)

[3] J. Nicholas Coleman and Rizalafande Che Ismail. 'LNS with Co-Transformation Competes with Floating-Point'. In: IEEE Transactions on Computers 65.1 (2016), pp. 136-146. DOI: 10.1109/TC.2015.2409059

[4] Jeff Johnson. 'Efficient, arbitrarily high precision hardware logarithmic arithmetic for linear algebra'. In: 2020 IEEE 27th Symposium on Computer Arithmetic (ARITH). Los Alamitos, CA, USA: IEEE Computer Society, June 2020, pp. 25-32. DOI: 10.1109/ARITH48897.2020.00013

Logarithmic Number System

- Used internally at Meta for AI-ASICs (MTIA)
- Instead of $(-1)^s(1+f)\cdot 2^e$ express numbers as $(-1)^s b^{e+f} =: (-1)^s b^{\ell}$ for basis $b > 1$ (usually $b = 2$)
- ℓ is fixed-point number, exponent $e \rightarrow$ characteristic c, fraction $f \rightarrow$ mantissa m
- Advantages
  - Multiplication, Division, Square Root, Squaring, Inversion are simple fixed-point addition, subtraction, right and left shifts and negation, e.g.
    $$b^{\ell_1}\cdot b^{\ell_2} = b^{\ell_1+\ell_2},\ \sqrt{b^{\ell}} = b^{\ell/2},\ \left(b^{\ell}\right)^{-1} = b^{-\ell}$$
  - Lower relative approximation error
  - Simpler hardware, more energy efficient, especially for AI[2]
- Disadvantages
  - Addition/subtraction more involved, but on-par with floating-point for $n \leq 32$ in regard to latency and area[3]

[2] Jeff Johnson. 'Rethinking floating point for deep learning'. arXiv: 1811.01721 (Nov. 2018)

[3] J. Nicholas Coleman and Rizalafande Che Ismail. 'LNS with Co-Transformation Competes with Floating-Point'. In: IEEE Transactions on Computers 65.1 (2016), pp. 136-146. DOI: 10.1109/TC.2015.2409059

[4] Jeff Johnson. 'Efficient, arbitrarily high precision hardware logarithmic arithmetic for linear algebra'. In: 2020 IEEE 27th Symposium on Computer Arithmetic (ARITH). Los Alamitos, CA, USA: IEEE Computer Society, June 2020, pp. 25-32. DOI: 10.1109/ARITH48897.2020.00013

Logarithmic Number System

- Used internally at Meta for AI-ASICs (MTIA)
- Instead of $(-1)^s(1+f)\cdot 2^e$ express numbers as $(-1)^s b^{e+f} =: (-1)^s b^{\ell}$ for basis $b > 1$ (usually $b = 2$)
- ℓ is fixed-point number, exponent $e \rightarrow$ characteristic c, fraction $f \rightarrow$ mantissa m
- Advantages
  - Multiplication, Division, Square Root, Squaring, Inversion are simple fixed-point addition, subtraction, right and left shifts and negation, e.g.
    $$b^{\ell_1} \cdot b^{\ell_2} = b^{\ell_1+\ell_2},\ \sqrt{b^{\ell}} = b^{\ell/2},\ \left(b^{\ell}\right)^{-1} = b^{-\ell}$$
  - Lower relative approximation error
  - Simpler hardware, more energy efficient, especially for AI[2]
- Disadvantages
  - Addition/subtraction more involved, but on-par with floating-point for $n \leq 32$ in regard to latency and area[3]
  - Binary zero representation challenge

[2] Jeff Johnson. 'Rethinking floating point for deep learning'. arXiv: 1811.01721 (Nov. 2018)

[3] J. Nicholas Coleman and Rizalafande Che Ismail. 'LNS with Co-Transformation Competes with Floating-Point'. In: IEEE Transactions on Computers 65.1 (2016), pp. 136-146. DOI: 10.1109/TC.2015.2409059

[4] Jeff Johnson. 'Efficient, arbitrarily high precision hardware logarithmic arithmetic for linear algebra'. In: 2020 IEEE 27th Symposium on Computer Arithmetic (ARITH). Los Alamitos, CA, USA: IEEE Computer Society, June 2020, pp. 25-32. DOI: 10.1109/ARITH48897.2020.00013

Logarithmic Number System

- Used internally at Meta for AI-ASICs (MTIA)
- Instead of $(-1)^s(1+f)\cdot 2^e$ express numbers as $(-1)^s b^{e+f} =: (-1)^s b^{\ell}$ for basis $b > 1$ (usually $b = 2$)
- ℓ is fixed-point number, exponent $e \rightarrow$ characteristic c, fraction $f \rightarrow$ mantissa m
- Advantages
  - Multiplication, Division, Square Root, Squaring, Inversion are simple fixed-point addition, subtraction, right and left shifts and negation, e.g.
    $$b^{\ell_1}\cdot b^{\ell_2} = b^{\ell_1+\ell_2},\ \sqrt{b^{\ell}} = b^{\ell/2},\ \left(b^{\ell}\right)^{-1} = b^{-\ell}$$
  - Lower relative approximation error
  - Simpler hardware, more energy efficient, especially for AI[2]
- Disadvantages
  - Addition/subtraction more involved, but on-par with floating-point for $n \leq 32$ in regard to latency and area[3]
  - Binary zero representation challenge
- All previous work for $b = 2$, advances[4] in mixed-base $(-1)^s 2^e e^f$ for $n > 32$

---

[2] Jeff Johnson. 'Rethinking floating point for deep learning'. arXiv: 1811.01721 (Nov. 2018)

[3] J. Nicholas Coleman and Rizalafande Che Ismail. 'LNS with Co-Transformation Competes with Floating-Point'. In: IEEE Transactions on Computers 65.1 (2016), pp. 136-146. DOI: 10.1109/TC.2015.2409059

[4] Jeff Johnson. 'Efficient, arbitrarily high precision hardware logarithmic arithmetic for linear algebra'. In: 2020 IEEE 27th Symposium on Computer Arithmetic (ARITH). Los Alamitos, CA, USA: IEEE Computer Society, June 2020, pp. 25-32. DOI: 10.1109/ARITH48897.2020.00013

Logarithmic Number System

- Used internally at Meta for AI-ASICs (MTIA)
- Instead of $(-1)^s(1+f)\cdot 2^e$ express numbers as $(-1)^s b^{e+f} =: (-1)^s b^{\ell}$ for basis $b > 1$ (usually $b = 2$)
- ℓ is fixed-point number, exponent $e \to$ characteristic c, fraction $f \to$ mantissa m
- Advantages
  - Multiplication, Division, Square Root, Squaring, Inversion are simple fixed-point addition, subtraction, right and left shifts and negation, e.g.
    $$b^{\ell_1} \cdot b^{\ell_2} = b^{\ell_1+\ell_2},\ \sqrt{b^{\ell}} = b^{\ell/2},\ \left(b^{\ell}\right)^{-1} = b^{-\ell}$$
  - Lower relative approximation error
  - Simpler hardware, more energy efficient, especially for AI[2]
- Disadvantages
  - Addition/subtraction more involved, but on-par with floating-point for $n \leq 32$ in regard to latency and area[3]
  - Binary zero representation challenge
- All previous work for $b = 2$, advances[4] in mixed-base $(-1)^s 2^e e^f$ for $n > 32$

Define takum as LNS with pure base $\sqrt{e}$ (transform to and from base e with ℓ-shift)

[2] Jeff Johnson. 'Rethinking floating point for deep learning'. arXiv: 1811.01721 (Nov. 2018)

[3] J. Nicholas Coleman and Rizalafande Che Ismail. 'LNS with Co-Transformation Competes with Floating-Point'. In: IEEE Transactions on Computers 65.1 (2016), pp. 136-146. DOI: 10.1109/TC.2015.2409059

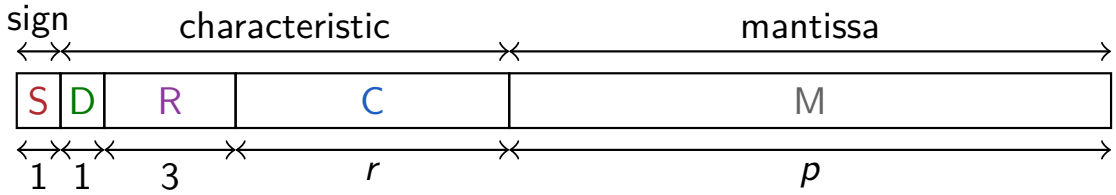
[4] Jeff Johnson. 'Efficient, arbitrarily high precision hardware logarithmic arithmetic for linear algebra'. In: 2020 IEEE 27th Symposium on Computer Arithmetic (ARITH). Los Alamitos, CA, USA: IEEE Computer Society, June 2020, pp. 25-32. DOI: 10.1109/ARITH48897.2020.00013

Takum

Definition

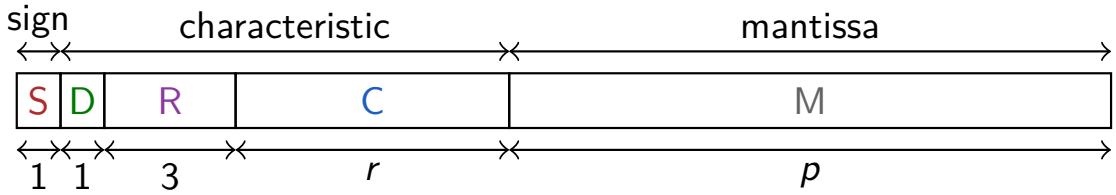
Takum

Definition



Takum

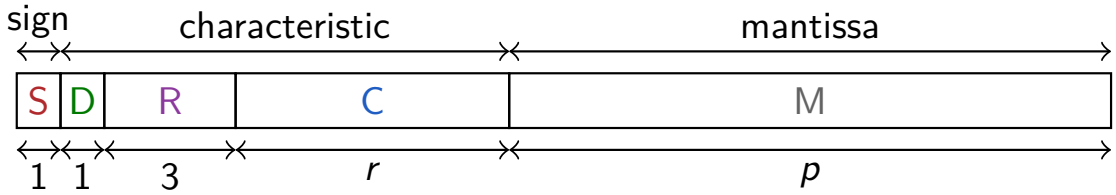
Definition



Sign bit S, direction bit D, regime bits R, characteristic bits C, mantissa bits M

Takum

Definition

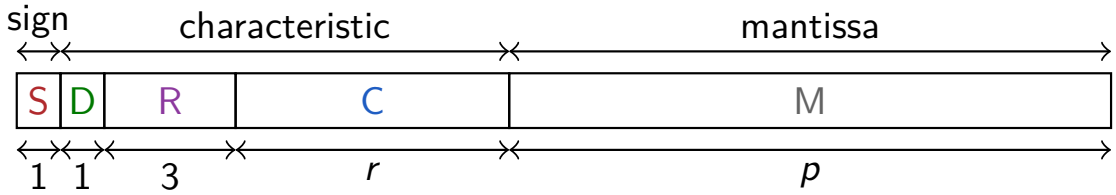


Sign bit S, direction bit D, regime bits R, characteristic bits C, mantissa bits M

$$r := \begin{cases} \text{uint}\left(\overline{R}\right) & D = 0 \\ \text{uint}(R) & D = 1 \end{cases} \in \{0,\ldots,7\} \qquad \text{: regime}$$

Takum

Definition



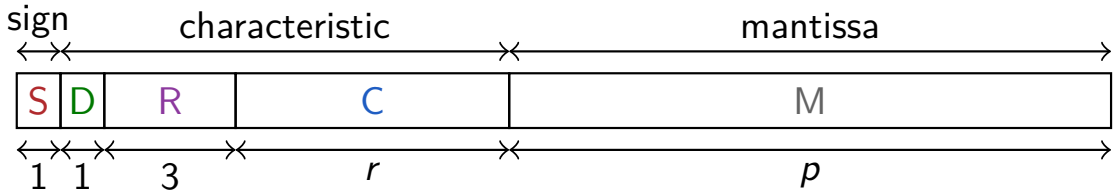
Sign bit S, direction bit D, regime bits R, characteristic bits C, mantissa bits M

$$r := \begin{cases} \text{uint}\left(\overline{R}\right) & D = 0 \\ \text{uint}(R) & D = 1 \end{cases} \in \{0,\ldots,7\} \qquad : \text{regime}$$

$$c := \begin{cases} -2^{r+1} + 1 + \text{uint}(C) & D = 0 \\ 2^{r} - 1 + \text{uint}(C) & D = 1 \end{cases} \qquad : \text{characteristic}$$

Takum

Definition



Sign bit S, direction bit D, regime bits R, characteristic bits C, mantissa bits M

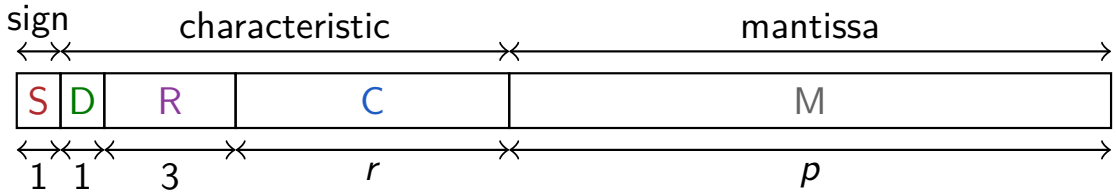
$$r := \begin{cases} \text{uint}\left(\overline{R}\right) & D = 0 \\ \text{uint}(R) & D = 1 \end{cases} \in \{0,\ldots,7\} \qquad : \text{regime}$$

$$c := \begin{cases} -2^{r+1} + 1 + \text{uint}(C) & D = 0 \\ 2^{r} - 1 + \text{uint}(C) & D = 1 \end{cases} \qquad : \text{characteristic}$$

$$p := n - r - 5 \in \{n - 12, \ldots, n - 5\} \qquad : \text{mantissa bit count}$$

Takum

Definition



Sign bit S, direction bit D, regime bits R, characteristic bits C, mantissa bits M

$$r := \begin{cases} \text{uint}\left(\overline{R}\right) & D = 0 \\ \text{uint}(R) & D = 1 \end{cases} \in \{0,\ldots,7\} \quad : \text{regime}$$

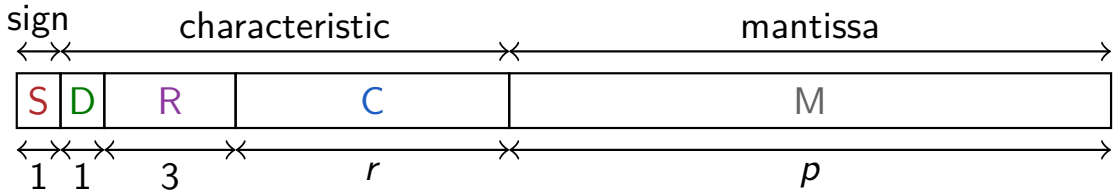
$$c := \begin{cases} -2^{r+1} + 1 + \text{uint}(C) & D = 0 \\ 2^{r} - 1 + \text{uint}(C) & D = 1 \end{cases} \quad : \text{characteristic}$$

$$p := n - r - 5 \in \{n - 12, \ldots, n - 5\} \quad : \text{mantissa bit count}$$

$$m := 2^{-p}\,\text{uint}(M) \in [0, 1) \quad : \text{mantissa}$$

Takum

Definition



Sign bit S, direction bit D, regime bits R, characteristic bits C, mantissa bits M

$$r := \begin{cases} \text{uint}\left(\overline{R}\right) & D = 0 \\ \text{uint}(R) & D = 1 \end{cases} \in \{0,\ldots,7\} \quad : \text{regime}$$

$$c := \begin{cases} -2^{r+1} + 1 + \text{uint}(C) & D = 0 \\ 2^{r} - 1 + \text{uint}(C) & D = 1 \end{cases} \quad : \text{characteristic}$$

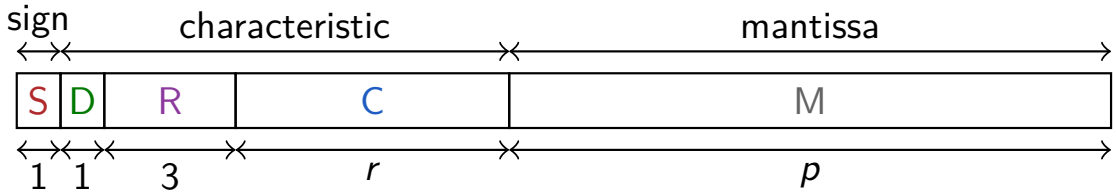
$$p := n - r - 5 \in \{n - 12, \ldots, n - 5\} \quad : \text{mantissa bit count}$$

$$m := 2^{-p}\,\text{uint}(M) \in [0, 1) \quad : \text{mantissa}$$

$$\ell := (-1)^{S}(c + m) \in (-255, 255) \quad : \text{logarithmic value}$$

Takum

Definition



Sign bit S, direction bit D, regime bits R, characteristic bits C, mantissa bits M

$$r := \begin{cases} \text{uint}\left(\overline{R}\right) & D = 0 \\ \text{uint}(R) & D = 1 \end{cases} \in \{0,\ldots,7\} \qquad : \text{regime}$$

$$c := \begin{cases} -2^{r+1} + 1 + \text{uint}(C) & D = 0 \\ 2^r - 1 + \text{uint}(C) & D = 1 \end{cases} \qquad : \text{characteristic}$$

$$p := n - r - 5 \in \{n - 12, \ldots, n - 5\} \qquad : \text{mantissa bit count}$$

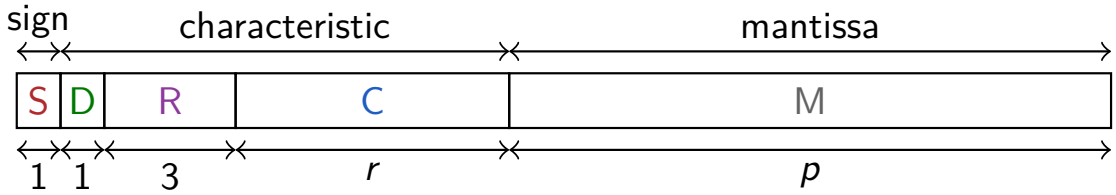
$$m := 2^{-p}\,\text{uint}(M) \in [0, 1) \qquad : \text{mantissa}$$

$$\ell := (-1)^S (c + m) \in (-255, 255) \qquad : \text{logarithmic value}$$

$$\tau((S, D, R, C, M)) := \begin{cases} \begin{cases} 0 & S = 0 \\ \text{NaR} & S = 1 \end{cases} & D = R = C = M = \mathbf{0} \\ (-1)^S \sqrt{\mathrm{e}^{\ell}} & \text{otherwise} \end{cases}$$

Takum

Definition



Sign bit S, direction bit D, regime bits R, characteristic bits C, mantissa bits M

$$r := \begin{cases} \text{uint}\left(\overline{R}\right) & D = 0 \\ \text{uint}(R) & D = 1 \end{cases} \in \{0,\ldots,7\} \qquad : \text{regime}$$

$$c := \begin{cases} -2^{r+1} + 1 + \text{uint}(C) & D = 0 \\ 2^r - 1 + \text{uint}(C) & D = 1 \end{cases} \qquad : \text{characteristic}$$

$$p := n - r - 5 \in \{n - 12, \ldots, n - 5\} \qquad : \text{mantissa bit count}$$

$$m := 2^{-p}\,\text{uint}(M) \in [0, 1) \qquad : \text{mantissa}$$

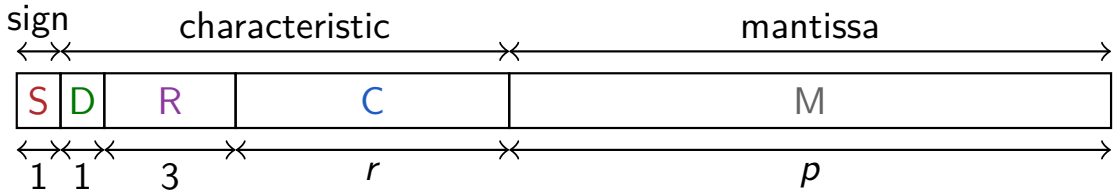
$$\ell := (-1)^S (c + m) \in (-255, 255) \qquad : \text{logarithmic value}$$

$$\tau((S, D, R, C, M)) := \begin{cases} \begin{cases} 0 & S = 0 \\ \text{NaR} & S = 1 \end{cases} & D = R = C = M = \mathbf{0} \\ (-1)^S \sqrt{\mathrm{e}^{\ell}} & \text{otherwise} \end{cases}$$

Decoding

Takum

Definition



Sign bit S, direction bit D, regime bits R, characteristic bits C, mantissa bits M

$$r := \begin{cases} \text{uint}\left(\overline{R}\right) & D = 0 \\ \text{uint}(R) & D = 1 \end{cases} \in \{0,\ldots,7\} \qquad : \text{regime}$$

$$c := \begin{cases} -2^{r+1} + 1 + \text{uint}(C) & D = 0 \\ 2^r - 1 + \text{uint}(C) & D = 1 \end{cases} \qquad : \text{characteristic}$$

$$p := n - r - 5 \in \{n - 12, \ldots, n - 5\} \qquad : \text{mantissa bit count}$$

$$m := 2^{-p}\,\text{uint}(M) \in [0, 1) \qquad : \text{mantissa}$$

$$\ell := (-1)^S (c + m) \in (-255, 255) \qquad : \text{logarithmic value}$$

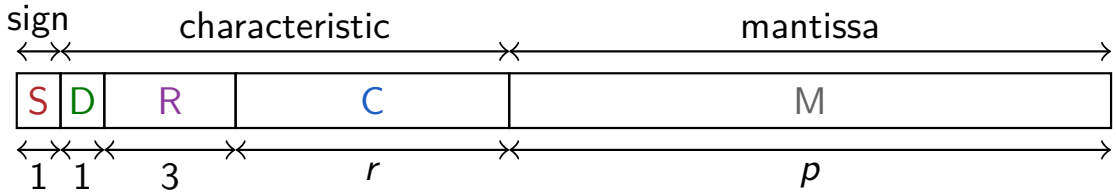
$$\tau((S, D, R, C, M)) := \begin{cases} \begin{cases} 0 & S = 0 \\ \text{NaR} & S = 1 \end{cases} & D = R = C = M = \mathbf{0} \\ (-1)^S \sqrt{\mathrm{e}^{\ell}} & \text{otherwise} \end{cases}$$

Decoding

- Only 16 distinct cases via (D, R), just two LUTs (32 byte, 16 byte)

Takum

Definition



Sign bit S, direction bit D, regime bits R, characteristic bits C, mantissa bits M

$$r := \begin{cases} \text{uint}\left(\overline{R}\right) & D = 0 \\ \text{uint}(R) & D = 1 \end{cases} \in \{0,\ldots,7\} \qquad : \text{regime}$$

$$c := \begin{cases} -2^{r+1} + 1 + \text{uint}(C) & D = 0 \\ 2^{r} - 1 + \text{uint}(C) & D = 1 \end{cases} \qquad : \text{characteristic}$$

$$p := n - r - 5 \in \{n - 12, \ldots, n - 5\} \qquad : \text{mantissa bit count}$$

$$m := 2^{-p}\,\text{uint}(M) \in [0, 1) \qquad : \text{mantissa}$$

$$\ell := (-1)^{S}(c + m) \in (-255, 255) \qquad : \text{logarithmic value}$$

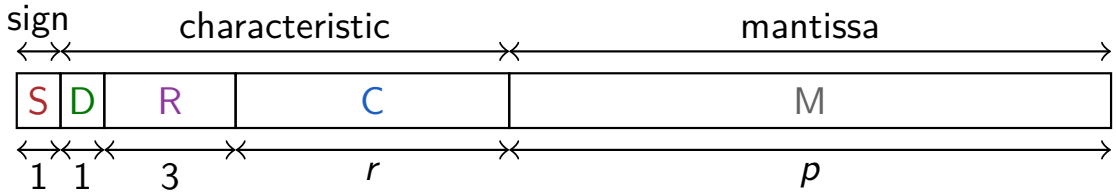
$$\tau((S, D, R, C, M)) := \begin{cases} \begin{cases} 0 & S = 0 \\ \text{NaR} & S = 1 \end{cases} & D = R = C = M = \mathbf{0} \\ (-1)^{S}\sqrt{\mathrm{e}^{\ell}} & \text{otherwise} \end{cases}$$

Decoding

- Only 16 distinct cases via (D, R), just two LUTs (32 byte, 16 byte)
- (D, R, C) is at most 11 bits long $\rightarrow$ shared logic for all n

Takum

Definition



Sign bit S, direction bit D, regime bits R, characteristic bits C, mantissa bits M

$$r := \begin{cases} \text{uint}\left(\overline{R}\right) & D = 0 \\ \text{uint}(R) & D = 1 \end{cases} \in \{0,\ldots,7\} \qquad : \text{regime}$$

$$c := \begin{cases} -2^{r+1} + 1 + \text{uint}(C) & D = 0 \\ 2^r - 1 + \text{uint}(C) & D = 1 \end{cases} \qquad : \text{characteristic}$$

$$p := n - r - 5 \in \{n - 12, \ldots, n - 5\} \qquad : \text{mantissa bit count}$$

$$m := 2^{-p}\,\text{uint}(M) \in [0, 1) \qquad : \text{mantissa}$$

$$\ell := (-1)^S (c + m) \in (-255, 255) \qquad : \text{logarithmic value}$$

$$\tau((S, D, R, C, M)) := \begin{cases} \begin{cases} 0 & S = 0 \\ \text{NaR} & S = 1 \end{cases} & D = R = C = M = \mathbf{0} \\ (-1)^S \sqrt{\mathrm{e}^{\ell}} & \text{otherwise} \end{cases}$$

Decoding

- Only 16 distinct cases via (D, R), just two LUTs (32 byte, 16 byte)
- (D, R, C) is at most 11 bits long $\rightarrow$ shared logic for all n
- Computation of c is conditional negation, bitwise OR and increment

Takum

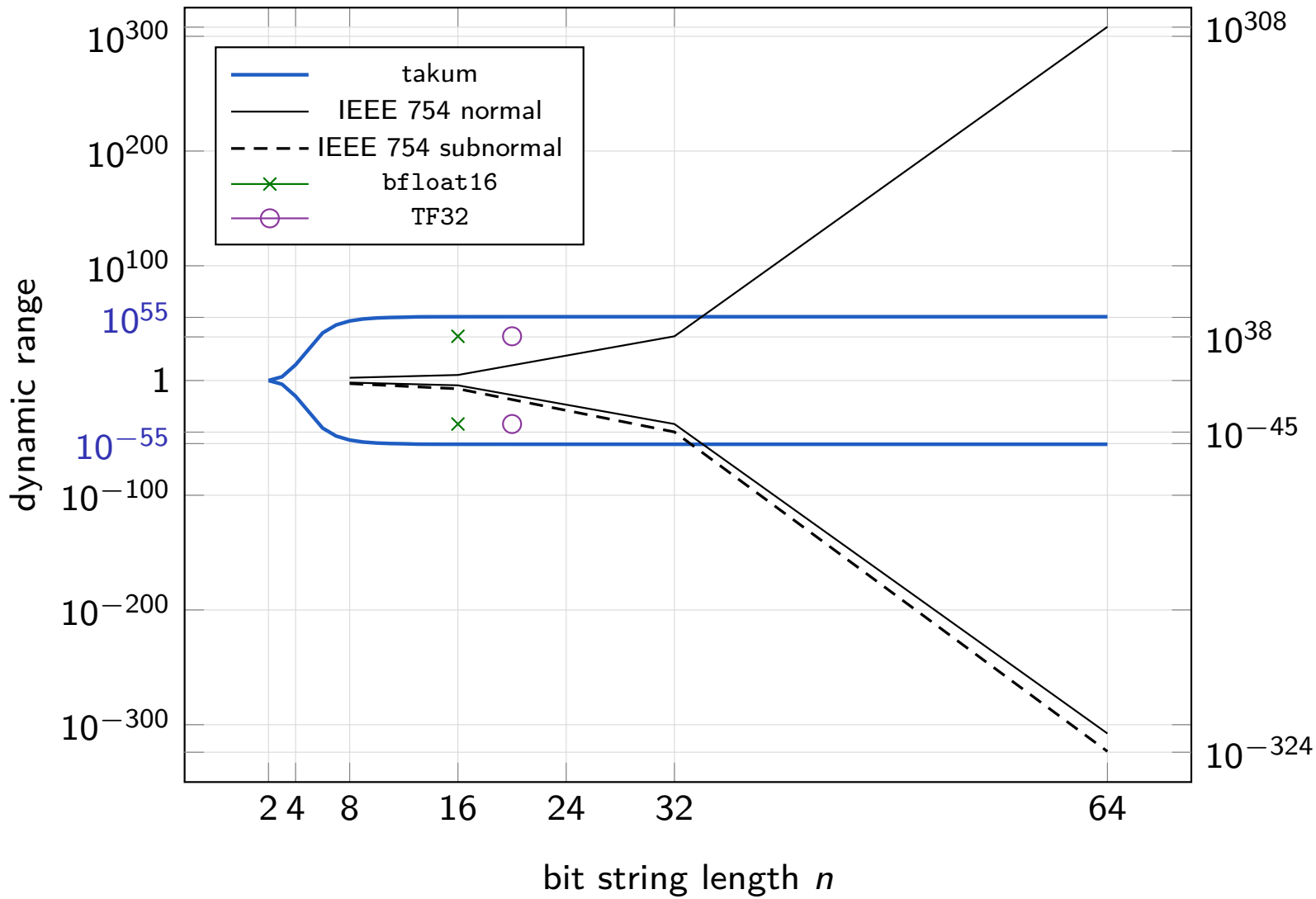
Dynamic Range

- Dynamic range $(\sqrt{\mathrm{e}}^{-255}, \sqrt{\mathrm{e}}^{255}) \approx (4.2 \times 10^{-56}, 2.4 \times 10^{55})$ for $n \geq 12$

Takum

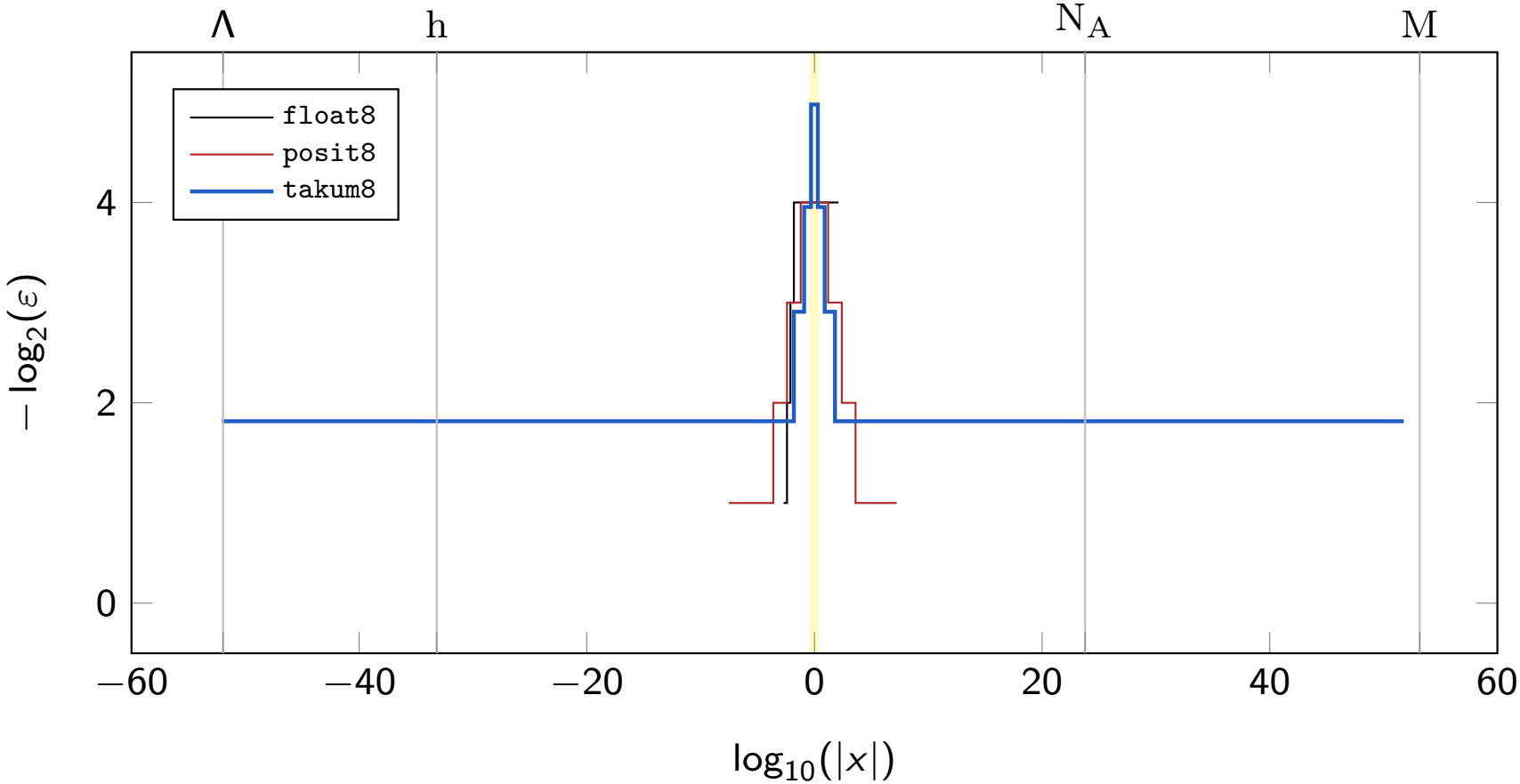
Dynamic Range

- Dynamic range $(\sqrt{\mathrm{e}}^{-255}, \sqrt{\mathrm{e}}^{255}) \approx (4.2 \times 10^{-56}, 2.4 \times 10^{55})$ for $n \geq 12$



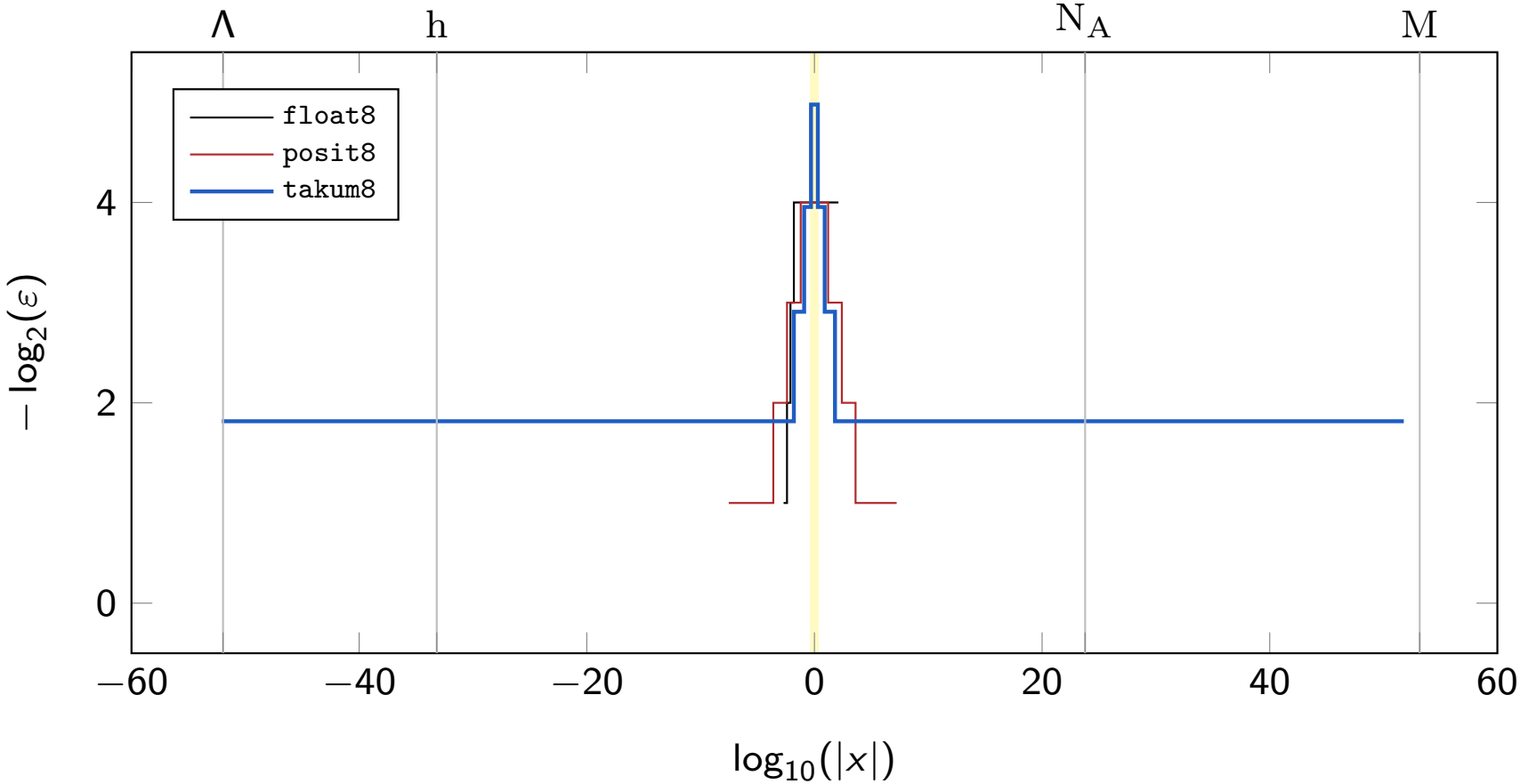
Takum

Density (8 Bit)



Takum

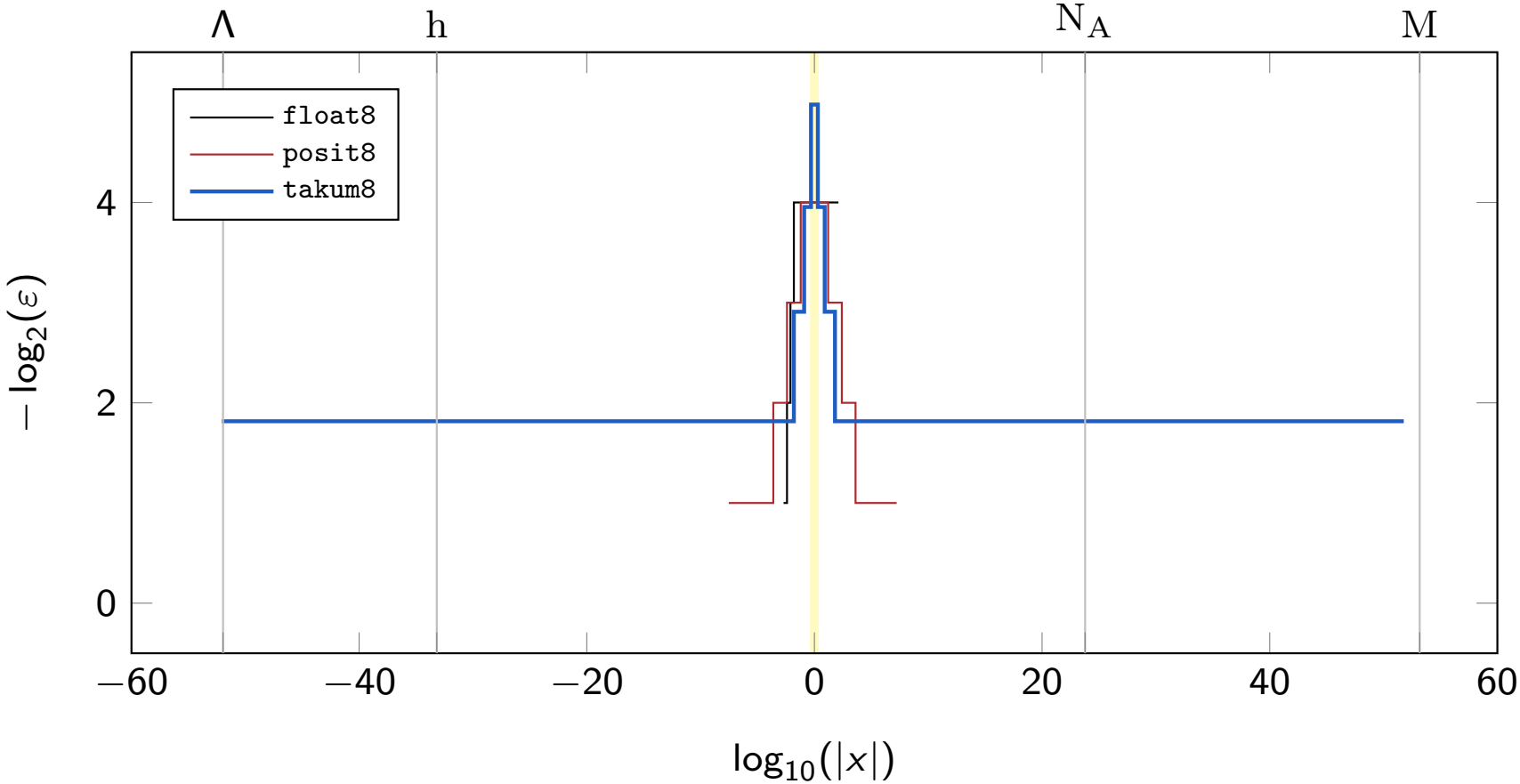
Density (8 Bit)



- `takum8` exceeds `float8` ('golden zone') and `posit8` near unity (2 times denser)

Takum

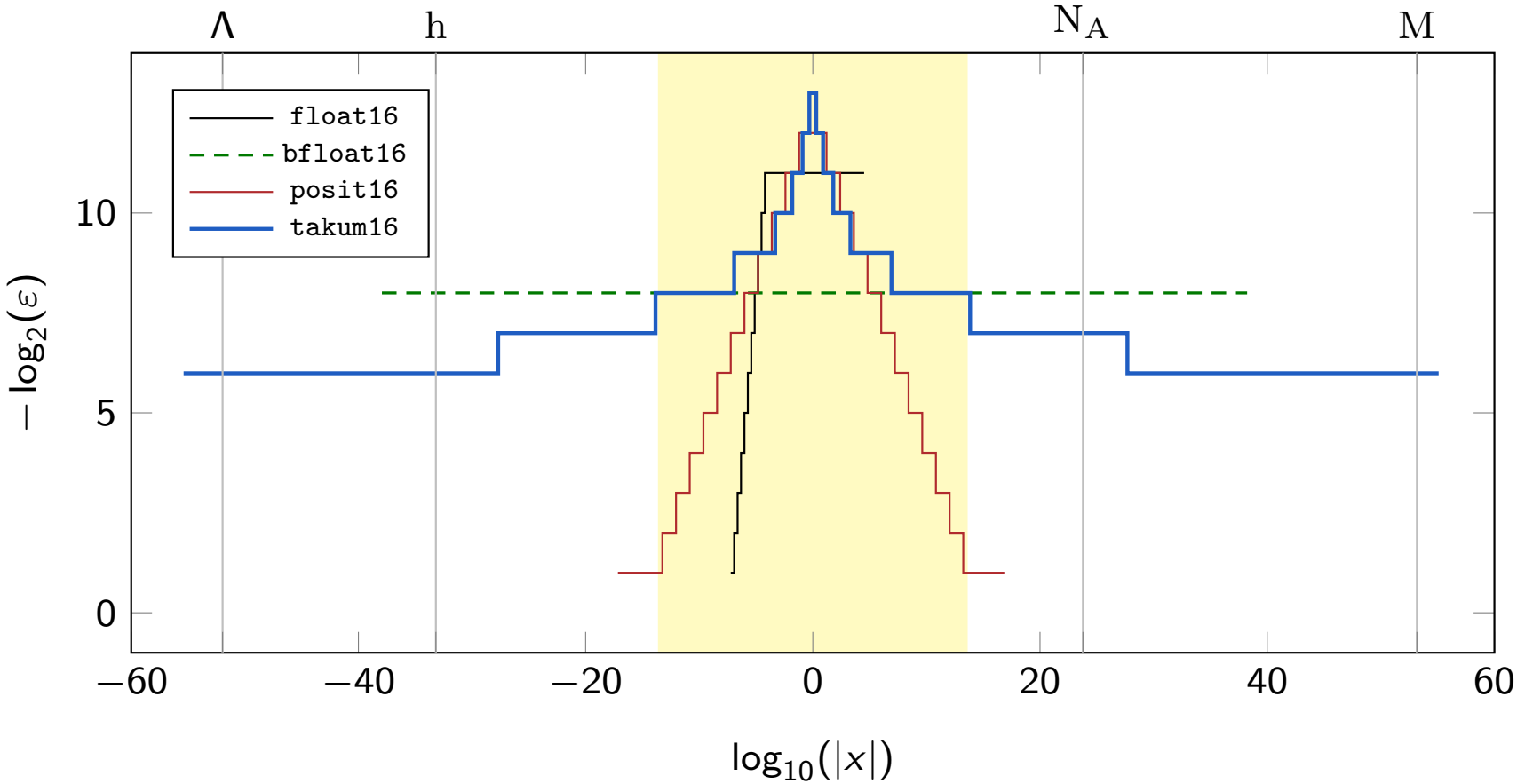
Density (8 Bit)



- `takum8` exceeds `float8` ('golden zone') and `posit8` near unity (2 times denser)
- Almost full dynamic range across benchmark constants Λ (cosmological constant), h (PLANCK constant), N_A (AVOGADRO constant), M (mass of the universe)

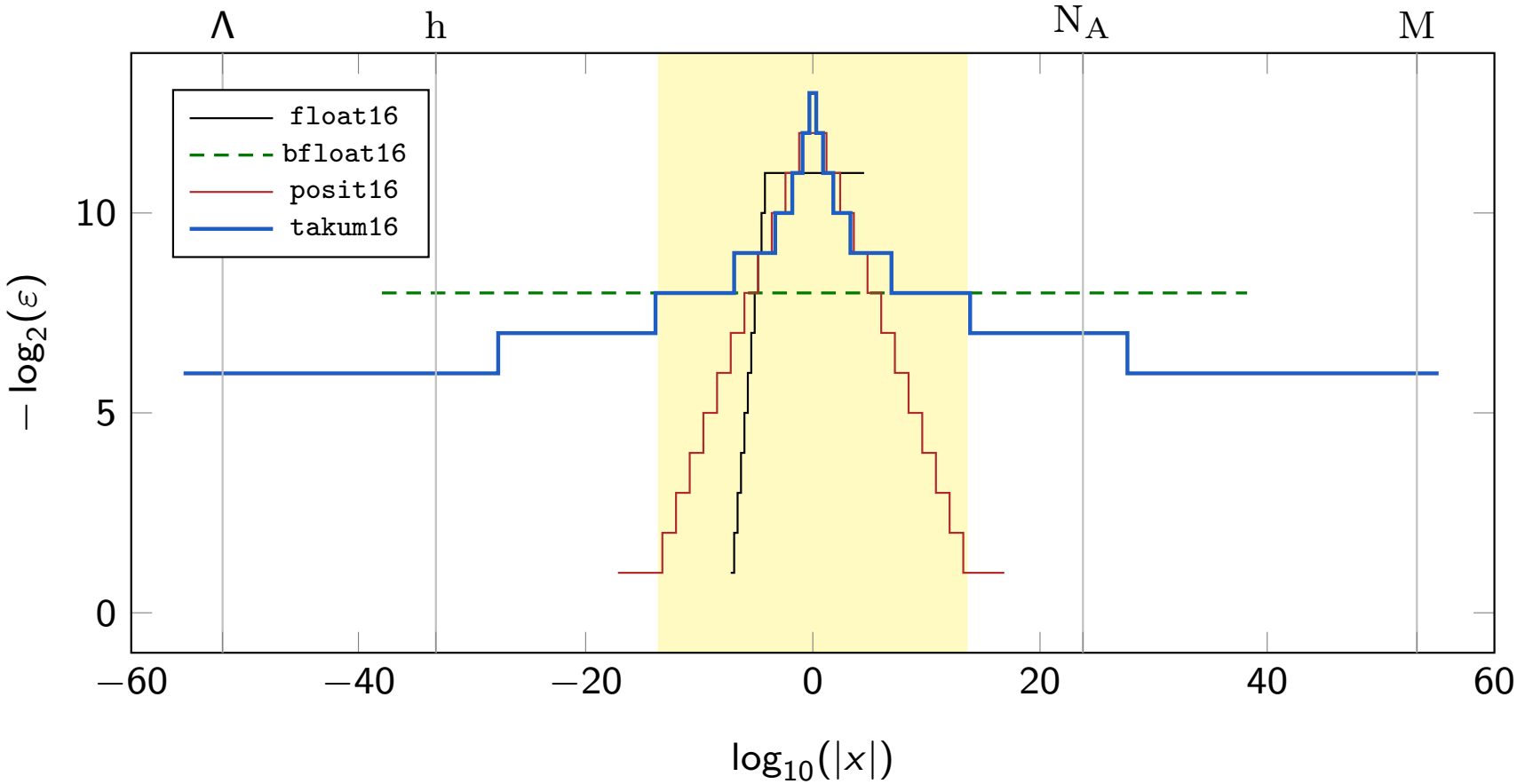
Takum

Density (16 Bit)



Takum

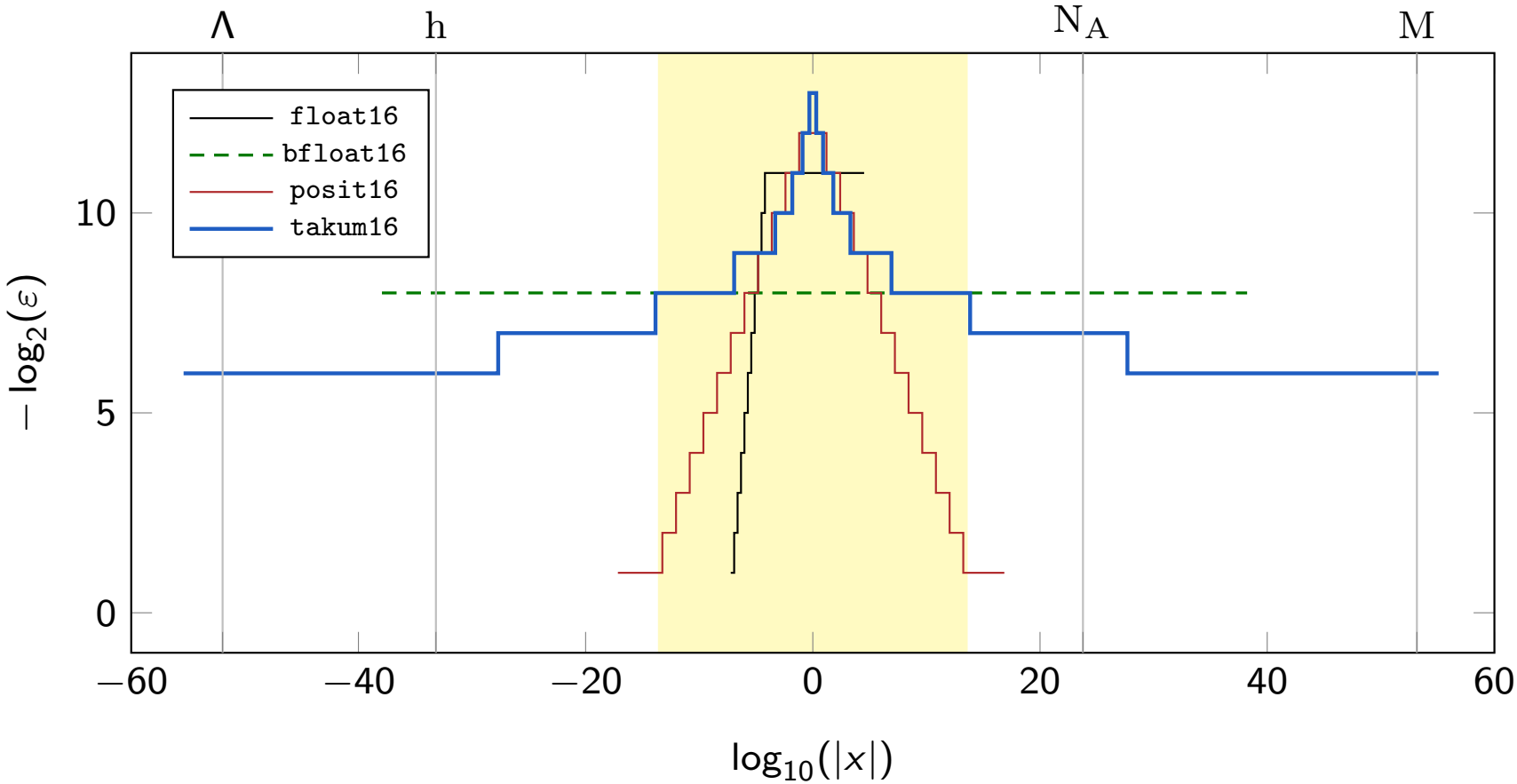
Density (16 Bit)



- Perfect logarithmic tapering

Takum

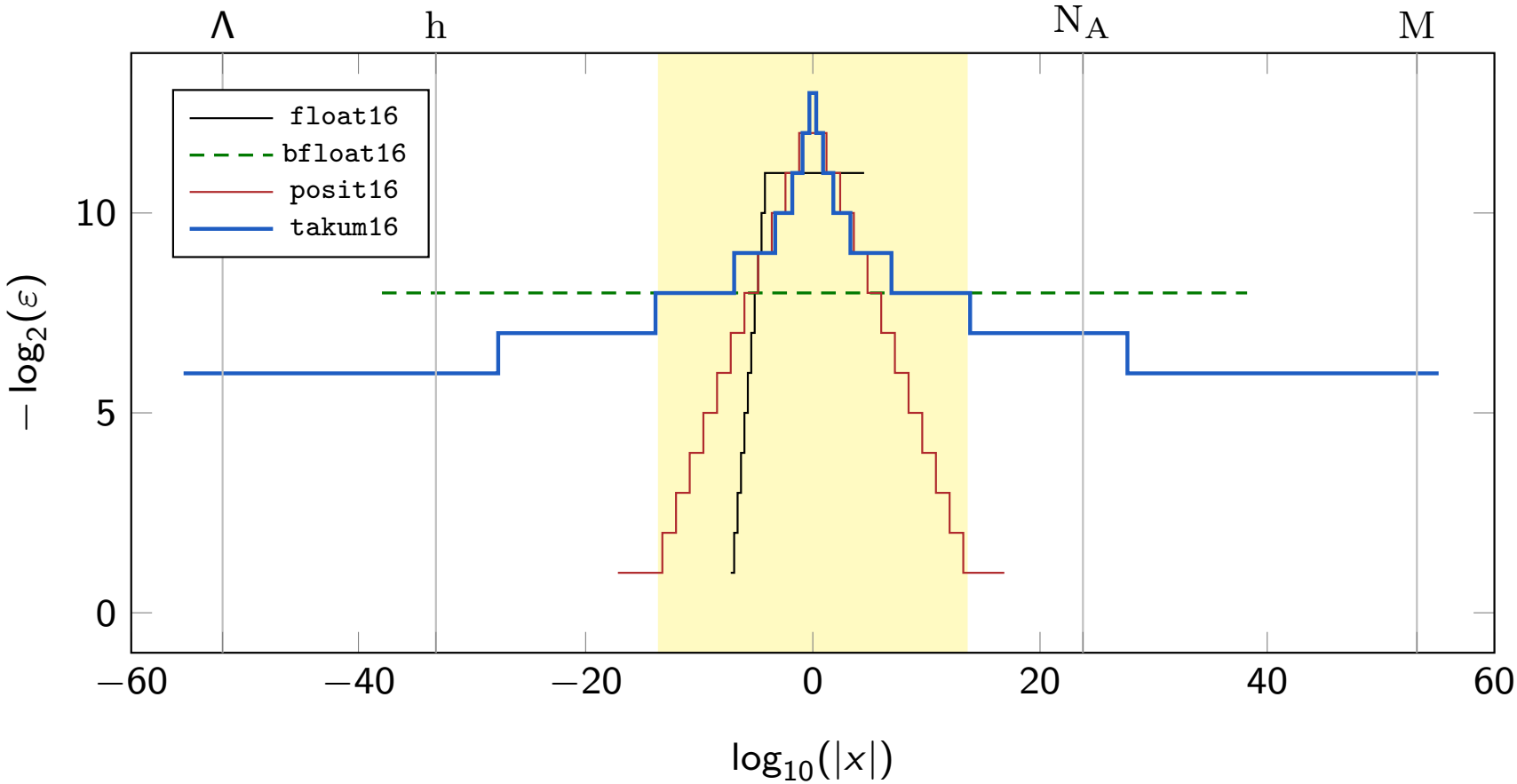
Density (16 Bit)



- Perfect logarithmic tapering
- At least the same density as `bfloat16` in 'golden zone' $(10^{-17}, 10^{17})$, up to 32 times denser

Takum

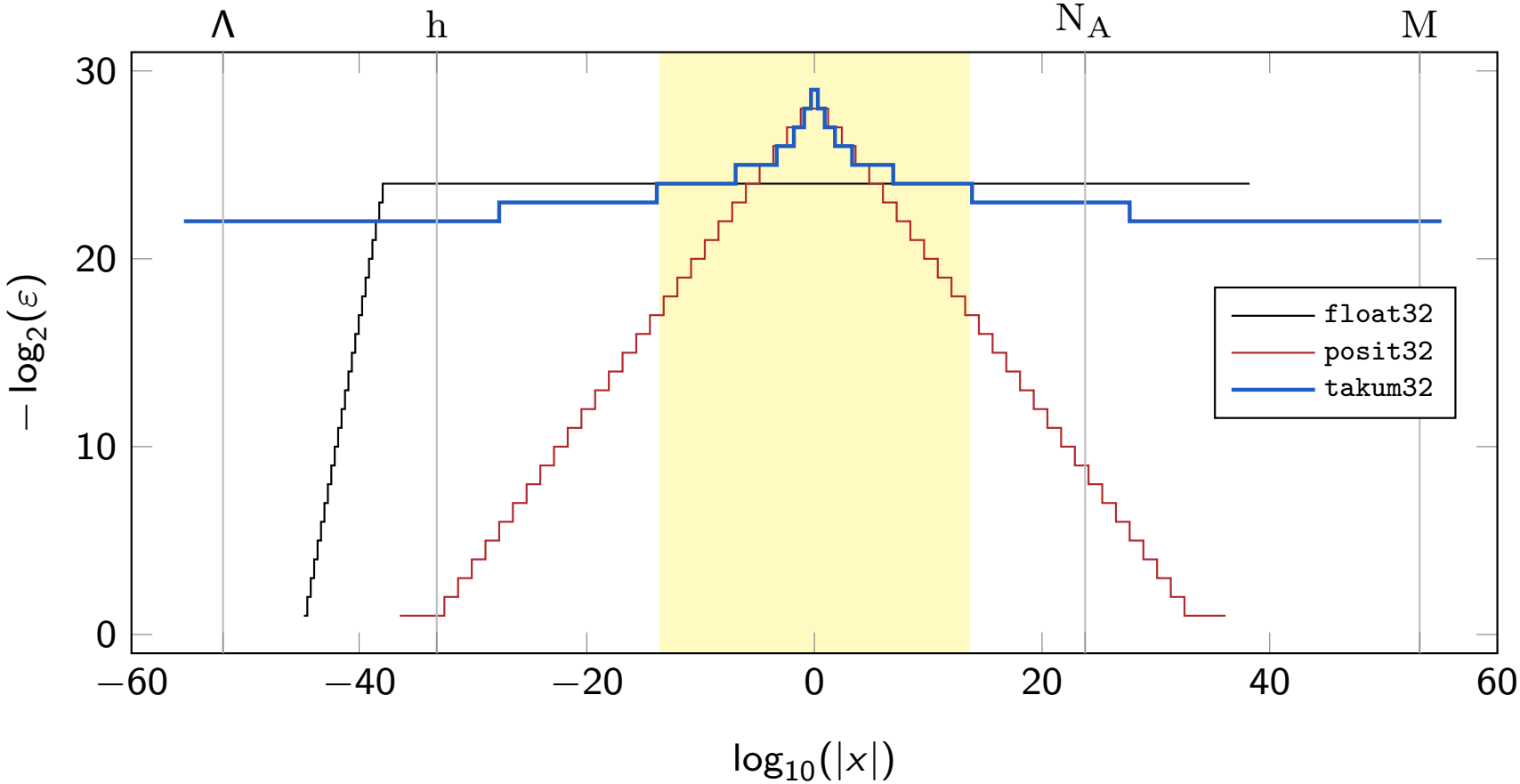
Density (16 Bit)



- Perfect logarithmic tapering
- At least the same density as `bfloat16` in 'golden zone' $(10^{-17}, 10^{17})$, up to 32 times denser
- Up to 2 times denser than `posit16`

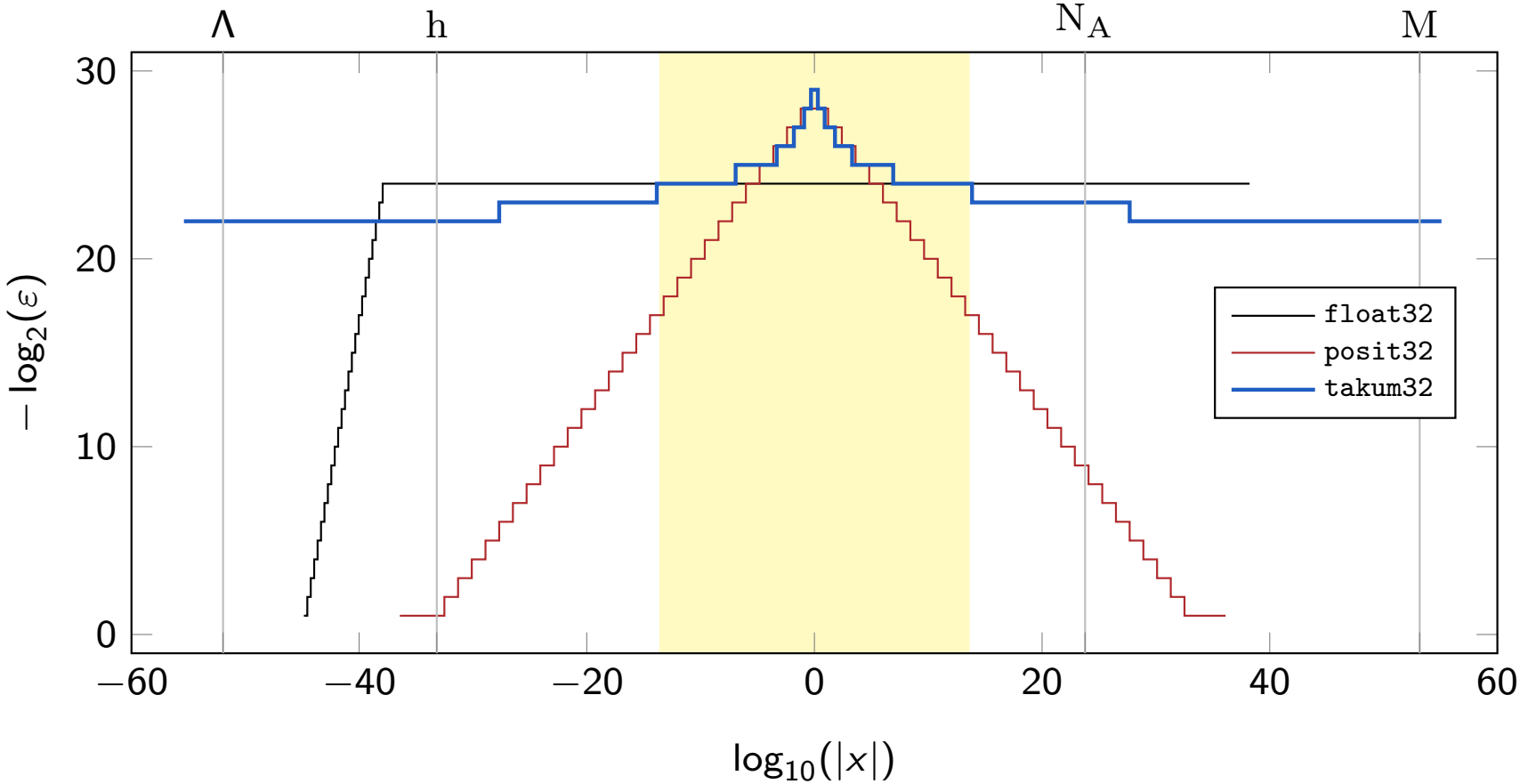
Takum

Density (32 Bit)



Takum

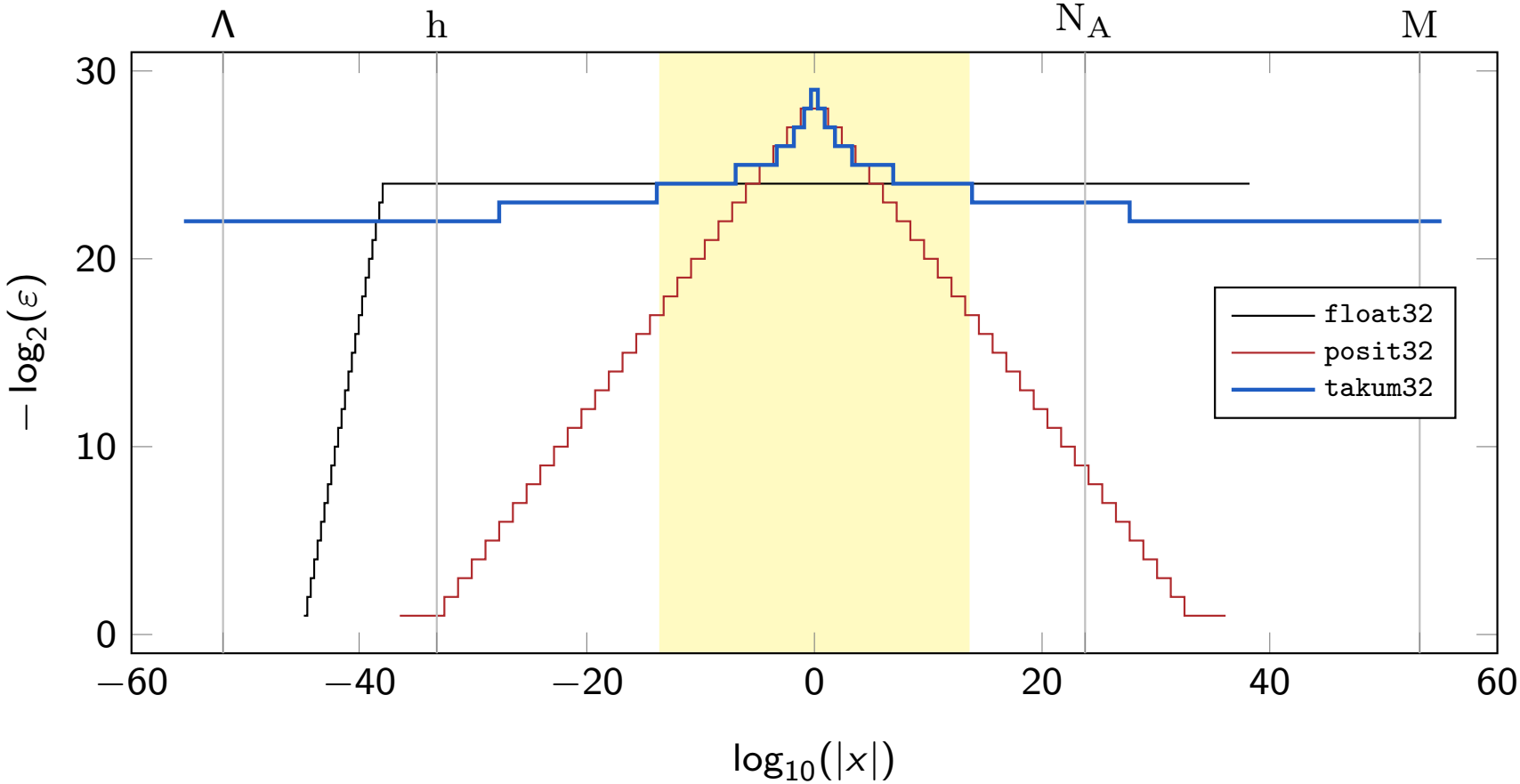
Density (32 Bit)



- Same golden zone as with $n = 16$

Takum

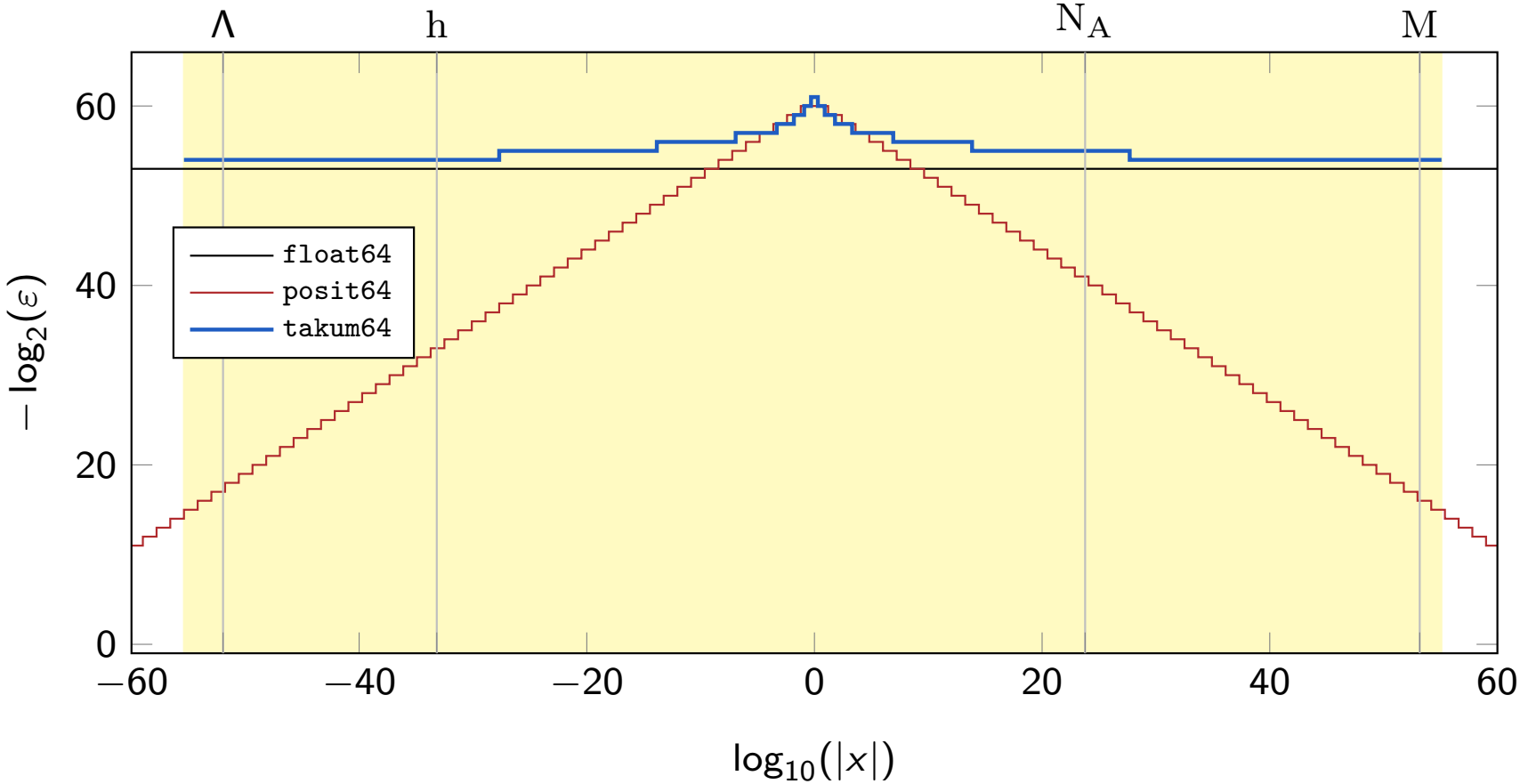
Density (32 Bit)



- Same golden zone as with $n = 16$
- Up to 32 times denser than `float32`

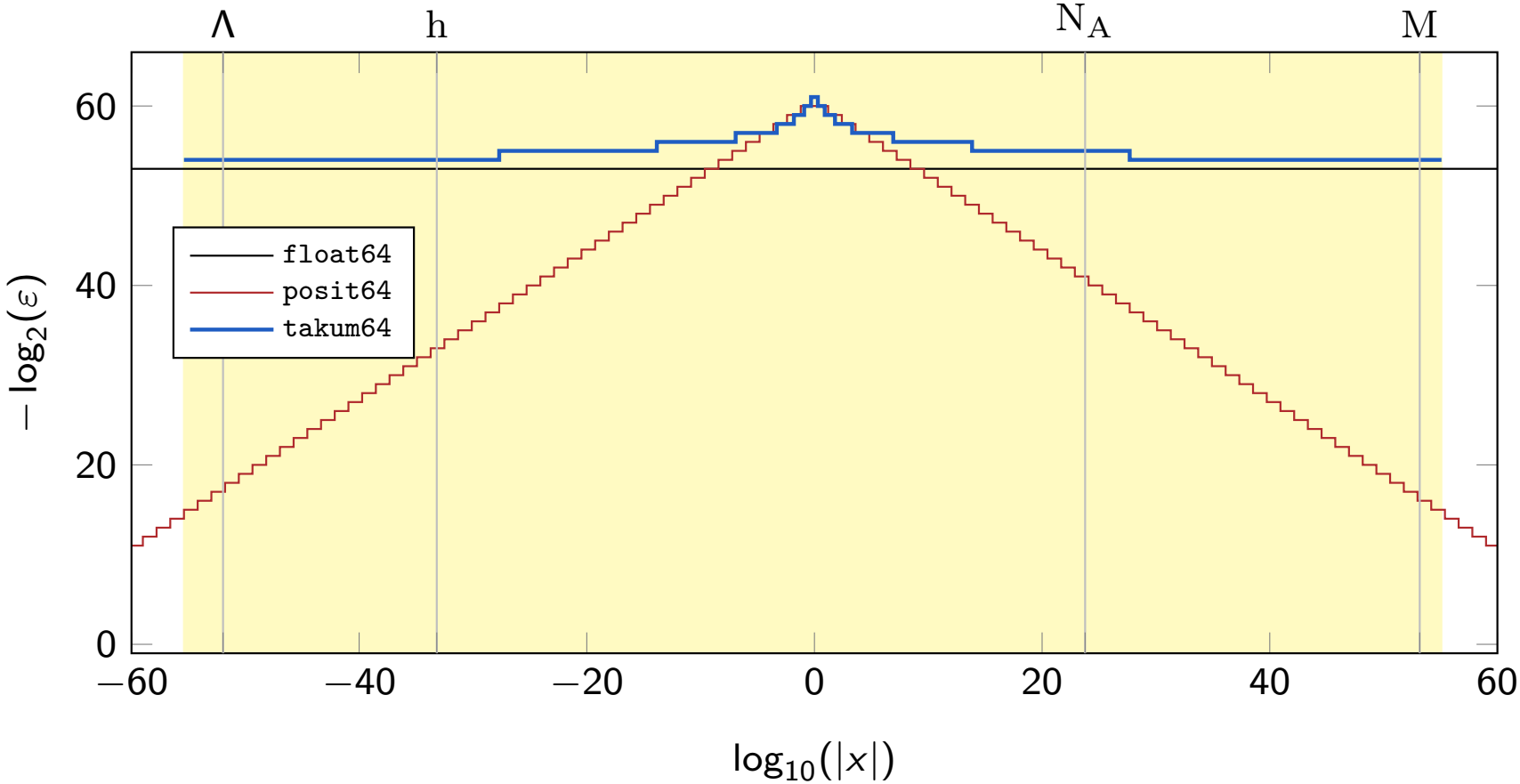
Takum

Density (64 Bit)



Takum

Density (64 Bit)



- At least 2 and up to 256 times denser than `float64`

Takum

Further Properties

Takum

Further Properties

- No redundant representations

Takum

Further Properties

- No redundant representations
- $\tau(\mathbf{0}) = 0$ (unique, unsigned zero)

Takum

Further Properties

- No redundant representations
- $\tau(\mathbf{0}) = 0$ (unique, unsigned zero)
- $\tau(-B) = -\tau(B)$ (two's complement negation, no extra hardware needed)

Takum

Further Properties

- No redundant representations
- $\tau(\mathbf{0}) = 0$ (unique, unsigned zero)
- $\tau(-B) = -\tau(B)$ (two's complement negation, no extra hardware needed)
- $B \preceq \tilde{B} \implies \tau(B) \leq \tau(\tilde{B})$ (total two's complement order, also for NaR (!), no extra hardware needed)

Takum

Further Properties

- No redundant representations
- $\tau(\mathbf{0}) = 0$ (unique, unsigned zero)
- $\tau(-B) = -\tau(B)$ (two's complement negation, no extra hardware needed)
- $B \preceq \tilde{B} \implies \tau(B) \leq \tau(\tilde{B})$ (total two's complement order, also for NaR (!), no extra hardware needed)
- Inversion is a simple bitwise operation

Takum

Further Properties

- No redundant representations
- $\tau(\mathbf{0}) = 0$ (unique, unsigned zero)
- $\tau(-B) = -\tau(B)$ (two's complement negation, no extra hardware needed)
- $B \preceq \tilde{B} \implies \tau(B) \leq \tau(\tilde{B})$ (total two's complement order, also for NaR (!), no extra hardware needed)
- Inversion is a simple bitwise operation
- Defined for all n, conversion between lengths simple rounding or expansion

FPGA Implementation (Artix 7 (100 MHz), VHDL)

[5] Raul Murillo et al. 'Comparing different decodings for posit arithmetic'. Conference on Next Generation Arithmetic, 2022

FPGA Implementation (Artix 7 (100 MHz), VHDL)

Comparison against the state of the art (FloPoCo with improvements, 2018-2023)[5]

[5] Raul Murillo et al. 'Comparing different decodings for posit arithmetic'. Conference on Next Generation Arithmetic, 2022

FPGA Implementation (Artix 7 (100 MHz), VHDL)

Comparison against the state of the art (FloPoCo with improvements, 2018-2023)[5]

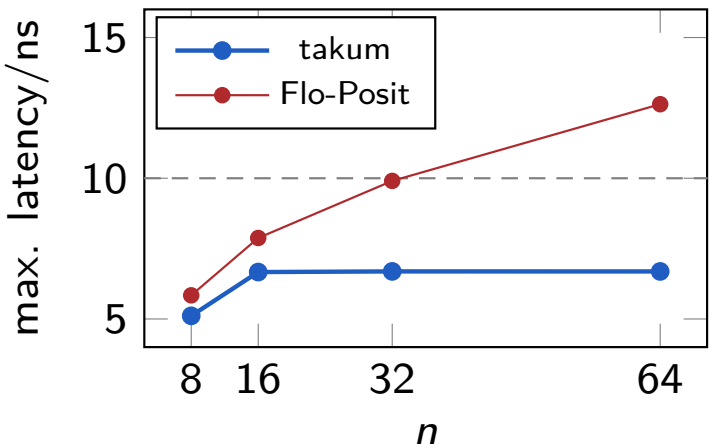
Decoder

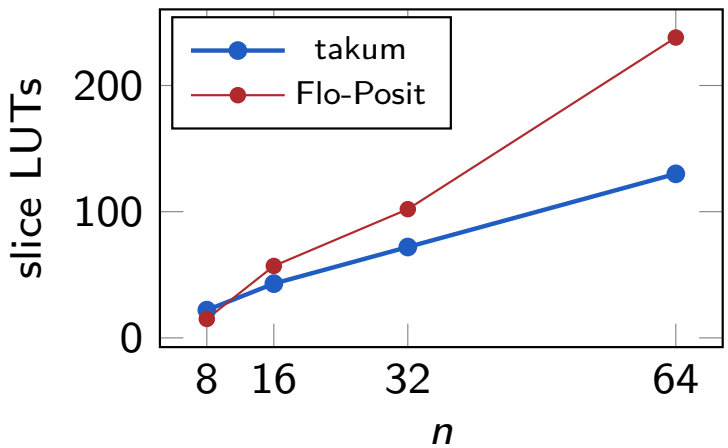
[5] Raul Murillo et al. 'Comparing different decodings for posit arithmetic'. Conference on Next Generation Arithmetic, 2022

FPGA Implementation (Artix 7 (100 MHz), VHDL)

Comparison against the state of the art (FloPoCo with improvements, 2018-2023)[5]

Decoder



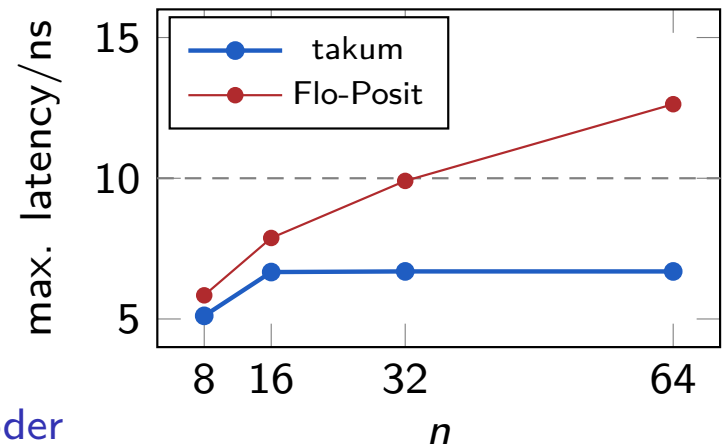


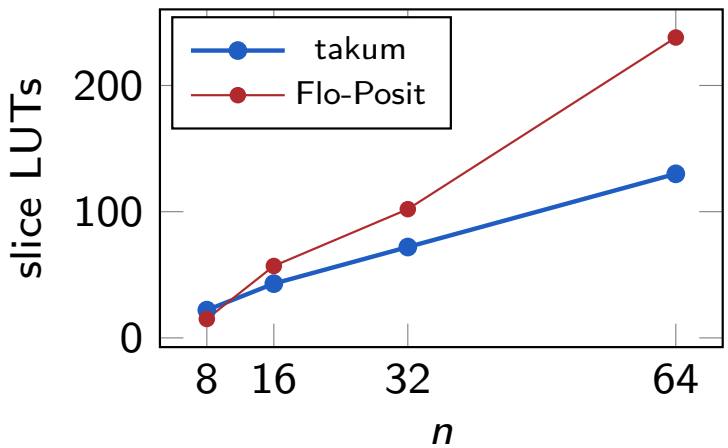
[5] Raul Murillo et al. 'Comparing different decodings for posit arithmetic'. Conference on Next Generation Arithmetic, 2022

FPGA Implementation (Artix 7 (100 MHz), VHDL)

Comparison against the state of the art (FloPoCo with improvements, 2018-2023)[5]

Decoder





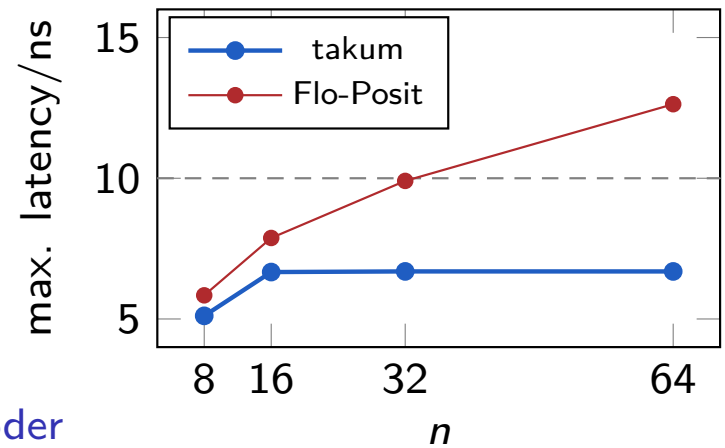
Encoder

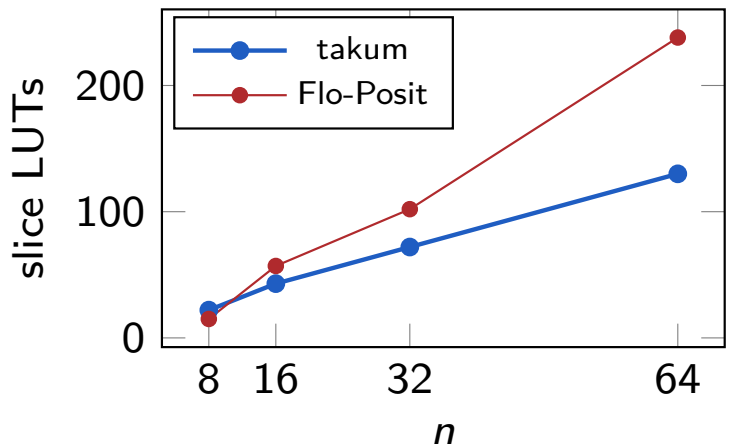
[5] Raul Murillo et al. 'Comparing different decodings for posit arithmetic'. Conference on Next Generation Arithmetic, 2022

FPGA Implementation (Artix 7 (100 MHz), VHDL)

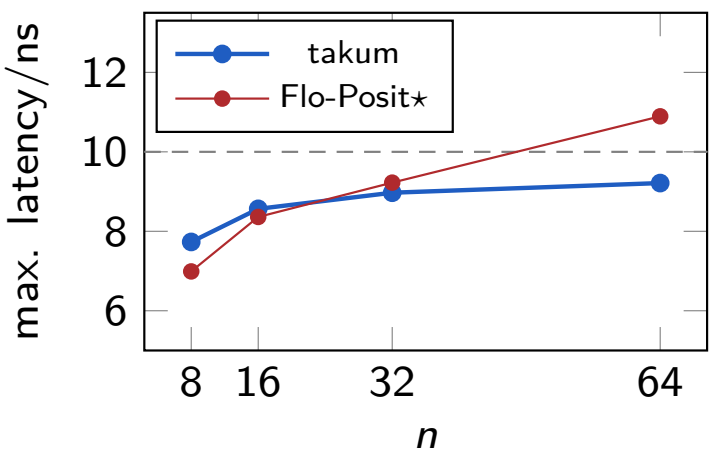
Comparison against the state of the art (FloPoCo with improvements, 2018-2023)[5]

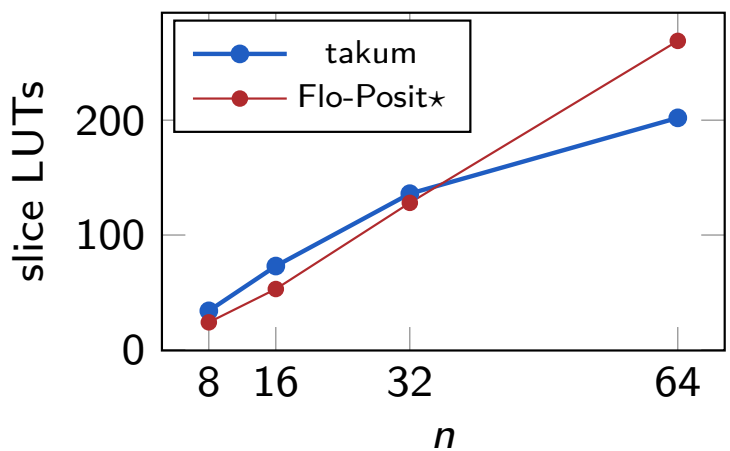
Decoder





Encoder





⋆: no under-/overflow detection

[5] Raul Murillo et al. 'Comparing different decodings for posit arithmetic'. Conference on Next Generation Arithmetic, 2022

Conclusion and Outlook

Conclusion and Outlook

- Takums promising for low-precision (e.g. AI) *and* general purpose arithmetic

Conclusion and Outlook

- Takums promising for low-precision (e.g. AI) *and* general purpose arithmetic
- True mixed-precision workflow

Conclusion and Outlook

- Takums promising for low-precision (e.g. AI) *and* general purpose arithmetic
- True mixed-precision workflow
- Novel base $\sqrt{e}$ LNS approach for simpler and more power efficient hardware

Conclusion and Outlook

- Takums promising for low-precision (e.g. AI) *and* general purpose arithmetic
- True mixed-precision workflow
- Novel base $\sqrt{e}$ LNS approach for simpler and more power efficient hardware
- Takum FPGA implementation superior to fastest FPGA posit codec

Conclusion and Outlook

- Takums promising for low-precision (e.g. AI) *and* general purpose arithmetic
- True mixed-precision workflow
- Novel base $\sqrt{e}$ LNS approach for simpler and more power efficient hardware
- Takum FPGA implementation superior to fastest FPGA posit codec
- C99 Implementation: `https://github.com/takum-arithmetic/libtakum`

Conclusion and Outlook

- Takums promising for low-precision (e.g. AI) *and* general purpose arithmetic
- True mixed-precision workflow
- Novel base $\sqrt{e}$ LNS approach for simpler and more power efficient hardware
- Takum FPGA implementation superior to fastest FPGA posit codec
- C99 Implementation: `https://github.com/takum-arithmetic/libtakum`
- Future work: Further evaluation

Conclusion and Outlook

- Takums promising for low-precision (e.g. AI) *and* general purpose arithmetic
- True mixed-precision workflow
- Novel base $\sqrt{e}$ LNS approach for simpler and more power efficient hardware
- Takum FPGA implementation superior to fastest FPGA posit codec
- C99 Implementation: `https://github.com/takum-arithmetic/libtakum`
- Future work: Further evaluation (deep learning

Conclusion and Outlook

- Takums promising for low-precision (e.g. AI) *and* general purpose arithmetic
- True mixed-precision workflow
- Novel base $\sqrt{e}$ LNS approach for simpler and more power efficient hardware
- Takum FPGA implementation superior to fastest FPGA posit codec
- C99 Implementation: `https://github.com/takum-arithmetic/libtakum`
- Future work: Further evaluation (deep learning, numerical methods

Conclusion and Outlook

- Takums promising for low-precision (e.g. AI) *and* general purpose arithmetic
- True mixed-precision workflow
- Novel base $\sqrt{e}$ LNS approach for simpler and more power efficient hardware
- Takum FPGA implementation superior to fastest FPGA posit codec
- C99 Implementation: `https://github.com/takum-arithmetic/libtakum`
- Future work: Further evaluation (deep learning, numerical methods, etc.)

Conclusion and Outlook

- Takums promising for low-precision (e.g. AI) *and* general purpose arithmetic
- True mixed-precision workflow
- Novel base $\sqrt{e}$ LNS approach for simpler and more power efficient hardware
- Takum FPGA implementation superior to fastest FPGA posit codec
- C99 Implementation: `https://github.com/takum-arithmetic/libtakum`
- Future work: Further evaluation (deep learning, numerical methods, etc.), ASIC implementation

Conclusion and Outlook

- Takums promising for low-precision (e.g. AI) *and* general purpose arithmetic
- True mixed-precision workflow
- Novel base $\sqrt{e}$ LNS approach for simpler and more power efficient hardware
- Takum FPGA implementation superior to fastest FPGA posit codec
- C99 Implementation: `https://github.com/takum-arithmetic/libtakum`
- Future work: Further evaluation (deep learning, numerical methods, etc.), ASIC implementation, 'tapered precision numerical analysis'

Conclusion and Outlook

- Takums promising for low-precision (e.g. AI) *and* general purpose arithmetic
- True mixed-precision workflow
- Novel base $\sqrt{e}$ LNS approach for simpler and more power efficient hardware
- Takum FPGA implementation superior to fastest FPGA posit codec
- C99 Implementation: https://github.com/takum-arithmetic/libtakum
- Future work: Further evaluation (deep learning, numerical methods, etc.), ASIC implementation, 'tapered precision numerical analysis'
- https://takum.org

Conclusion and Outlook

- Takums promising for low-precision (e.g. AI) *and* general purpose arithmetic
- True mixed-precision workflow
- Novel base $\sqrt{e}$ LNS approach for simpler and more power efficient hardware
- Takum FPGA implementation superior to fastest FPGA posit codec
- C99 Implementation: `https://github.com/takum-arithmetic/libtakum`
- Future work: Further evaluation (deep learning, numerical methods, etc.), ASIC implementation, 'tapered precision numerical analysis'
- `https://takum.org`
- Full paper: 'Beating Posits at Their Own Game: Takum Arithmetic'. In: *Conference on Next Generation Arithmetic (CoNGA) 2024*. Lecture Notes in Computer Science. In publication. Berlin, Heidelberg: Springer Berlin Heidelberg, 2024 `https://arxiv.org/abs/2404.18603`

Conclusion and Outlook

- Takums promising for low-precision (e.g. AI) *and* general purpose arithmetic
- True mixed-precision workflow
- Novel base $\sqrt{e}$ LNS approach for simpler and more power efficient hardware
- Takum FPGA implementation superior to fastest FPGA posit codec
- C99 Implementation: `https://github.com/takum-arithmetic/libtakum`
- Future work: Further evaluation (deep learning, numerical methods, etc.), ASIC implementation, 'tapered precision numerical analysis'
- `https://takum.org`
- Full paper: 'Beating Posits at Their Own Game: Takum Arithmetic'. In: *Conference on Next Generation Arithmetic (CoNGA) 2024*. Lecture Notes in Computer Science. In publication. Berlin, Heidelberg: Springer Berlin Heidelberg, 2024 `https://arxiv.org/abs/2404.18603`

IEEE 754 is a dead end.

Conclusion and Outlook

- Takums promising for low-precision (e.g. AI) *and* general purpose arithmetic
- True mixed-precision workflow
- Novel base $\sqrt{e}$ LNS approach for simpler and more power efficient hardware
- Takum FPGA implementation superior to fastest FPGA posit codec
- C99 Implementation: `https://github.com/takum-arithmetic/libtakum`
- Future work: Further evaluation (deep learning, numerical methods, etc.), ASIC implementation, 'tapered precision numerical analysis'
- `https://takum.org`
- Full paper: 'Beating Posits at Their Own Game: Takum Arithmetic'. In: *Conference on Next Generation Arithmetic (CoNGA) 2024*. Lecture Notes in Computer Science. In publication. Berlin, Heidelberg: Springer Berlin Heidelberg, 2024 `https://arxiv.org/abs/2404.18603`

IEEE 754 is a dead end.
Even tiny machine number improvements yield large high-level effects.

Conclusion and Outlook

- Takums promising for low-precision (e.g. AI) *and* general purpose arithmetic
- True mixed-precision workflow
- Novel base $\sqrt{e}$ LNS approach for simpler and more power efficient hardware
- Takum FPGA implementation superior to fastest FPGA posit codec
- C99 Implementation: `https://github.com/takum-arithmetic/libtakum`
- Future work: Further evaluation (deep learning, numerical methods, etc.), ASIC implementation, 'tapered precision numerical analysis'
- `https://takum.org`
- Full paper: 'Beating Posits at Their Own Game: Takum Arithmetic'. In: *Conference on Next Generation Arithmetic (CoNGA) 2024*. Lecture Notes in Computer Science. In publication. Berlin, Heidelberg: Springer Berlin Heidelberg, 2024 `https://arxiv.org/abs/2404.18603`

IEEE 754 is a dead end.
Even tiny machine number improvements yield large high-level effects.

Time for a new paradigm?


takum
arithmetic