2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SAN JOSE, CA, USA
FEBRUARY 24-27, 2025

Beyond Integers and Floating Point

Designing and Verifying with Alternate Number Representations

Russell Klein

Siemens EDA

SIEMENS

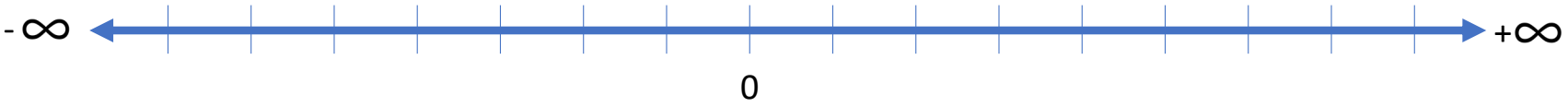
accellera SYSTEMS INITIATIVE

Agenda

- Numbers and their representation
  - Integer, floats, fixed points, and more
- Why change representations?
  - Power, Performance, Area
- Impacts on algorithms
  - Edge conditions – rounding, overflow, and saturation
- Impacts on verification

A Bit of Number Theory

- Numbers are infinite
  - Infinitely large
    - For both the positive and negative values
  - Infinitely small
- Modern (and even some ancient) math relies on this
  - Algebra, Calculus, Trigonometry are all deeply rooted in this concept



Representations

- Any representation we create will be practically finite
  - Only so much room on the paper
- Arabic was the most extensible
  - But still limited for very large and very small numbers

Arabic	Egyptian	Babylonian	Greek	Roman	Chinese	Aztec	Cretan	Mayan
1	𓏺	𒁹	α	I	一	•	𐄇	•
2	𓏺𓏺	𒁹𒁹	β	II	二	••	𐄇𐄇	••
3	𓏺𓏺𓏺	𒁹𒁹𒁹	γ	III	三	•••	𐄇𐄇𐄇	•••
4	𓏺𓏺𓏺𓏺	𒁹𒁹𒁹𒁹	δ	IIII	四	••••	𐄇𐄇𐄇𐄇	••••
5	𓏺𓏺𓏺𓏺𓏺	𒁹𒁹𒁹𒁹𒁹	ε	V	五	•••••	𐄇𐄇𐄇𐄇𐄇	———
6	𓏺𓏺𓏺𓏺𓏺𓏺	𒁹𒁹𒁹𒁹𒁹𒁹	ϛ	VI	六	••••• •	𐄇𐄇𐄇𐄇𐄇𐄇	——— •
7	𓏺𓏺𓏺𓏺𓏺𓏺𓏺	𒁹𒁹𒁹𒁹𒁹𒁹𒁹	ζ	VII	七	••••• ••	𐄇𐄇𐄇𐄇𐄇𐄇𐄇	——— ••
8	𓏺𓏺𓏺𓏺𓏺𓏺𓏺𓏺	𒁹𒁹𒁹𒁹𒁹𒁹𒁹𒁹	η	VIII	八	••••• •••	𐄇𐄇𐄇𐄇𐄇𐄇𐄇𐄇	——— •••
9	𓏺𓏺𓏺𓏺𓏺𓏺𓏺𓏺𓏺	𒁹𒁹𒁹𒁹𒁹𒁹𒁹𒁹𒁹	θ	VIIII	九	••••• ••••	𐄇𐄇𐄇𐄇𐄇𐄇𐄇𐄇𐄇	——— ••••
10	𓎆	𒌋	ι	X	一十	•••••	𐄐	═══
20	𓎆𓎆	𒌋𒌋	κ	XX	二十	𐤟	𐄐𐄐	═══ •
30	𓎆𓎆𓎆	𒌋𒌋𒌋	λ	XXX	三十	•••••𐤟	𐄐𐄐𐄐	═══ •••
40	𓎆𓎆𓎆𓎆	𒌋𒌋𒌋𒌋	μ	XXXX	四十	𐤟𐤟	𐄐𐄐𐄐𐄐	═══ ••••
50	𓎆𓎆𓎆𓎆𓎆	𒌋𒌋𒌋𒌋𒌋	ν	L	五十	•••••𐤟𐤟	𐄐𐄐𐄐𐄐𐄐	═══ •••••
60	𓎆𓎆𓎆𓎆𓎆𓏺	𒁹	ξ	LX	六十	𐤟𐤟𐤟	𐄐𐄐𐄐𐄐𐄐𐄇	═══ •••••
70	𓎆𓎆𓎆𓎆𓎆𓏺𓏺	𒁹𒌋	ο	LXX	七十	•••••𐤟𐤟𐤟	𐄐𐄐𐄐𐄐𐄐𐄇𐄇	═══ ••••• ••
80	𓎆𓎆𓎆𓎆𓎆𓏺𓏺𓏺	𒁹𒌋𒌋	π	LXXX	八十	𐤟𐤟𐤟𐤟	𐄐𐄐𐄐𐄐𐄐𐄇𐄇𐄇	═══ ••••• •••
90	𓎆𓎆𓎆𓎆𓎆𓏺𓏺𓏺𓏺	𒁹𒌋𒌋𒌋	ϟ	LXXXX	九十	•••••𐤟𐤟𐤟𐤟	𐄐𐄐𐄐𐄐𐄐𐄇𐄇𐄇𐄇	═══ ••••• ••••
100	𓍢	𒁹𒌋𒌋𒌋	ρ	C	一百	𐤟𐤟𐤟𐤟𐤟	𐄐	———
200	𓍢𓍢	𒁹𒁹𒌋𒌋𒌋	σ	CC	二百	𐤟𐤟𐤟𐤟𐤟 𐤟𐤟𐤟𐤟𐤟	𐄐𐄐	═══

J. Zhang, D. Norman, “Cognition,” Vol 57, Issue 3, December 1995, Pages 271-295

Representations

- Sciences and engineering require representations for very big and very small numbers
- Scientific notations
  - Exponentiation
- Examples

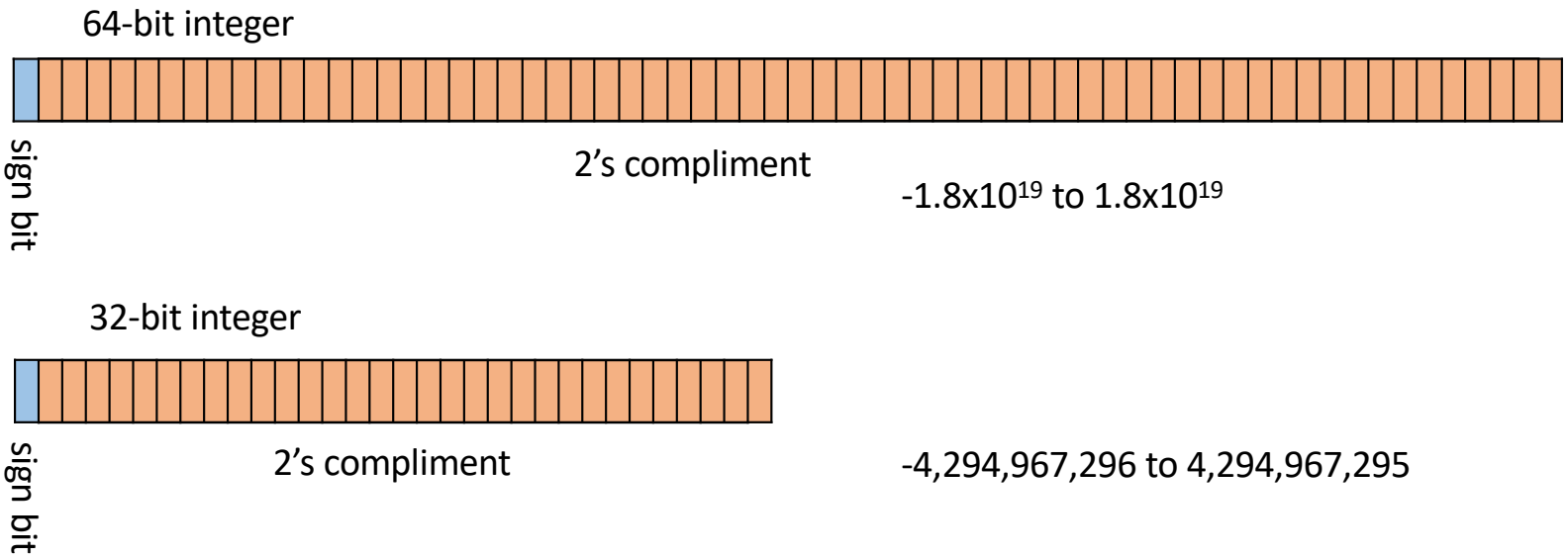
Earth's mass	5,972,400,000,000,000,000,000,000 grams	5.9724×10^{27}
Electron's mass	0.00000000000000000000000000091094 grams	9.1094×19^{-28}

SI Units

Prefix		Base 10	Decimal	Adoption [nb 1]
Name	Symbol			
quetta	Q	10^{30}	1 000 000 000 000 000 000 000 000 000 000	2022[3]
ronna	R	10^{27}	1 000 000 000 000 000 000 000 000 000	
yotta	Y	10^{24}	1 000 000 000 000 000 000 000 000	1991
zetta	Z	10^{21}	1 000 000 000 000 000 000 000	
exa	E	10^{18}	1 000 000 000 000 000 000	1975[4]
peta	P	10^{15}	1 000 000 000 000 000	
tera	T	10^{12}	1 000 000 000 000	1960
giga	G	10^{9}	1 000 000 000	
mega	M	10^{6}	1 000 000	1873
kilo	k	10^{3}	1 000	1795
hecto	h	10^{2}	100	
deca	da	10^{1}	10	
deci	d	10^{-1}	0.1	1795
centi	c	10^{-2}	0.01	
milli	m	10^{-3}	0.001	
micro	µ	10^{-6}	0.000 001	1873
nano	n	10^{-9}	0.000 000 001	1960
pico	p	10^{-12}	0.000 000 000 001	
femto	f	10^{-15}	0.000 000 000 000 001	1964
atto	a	10^{-18}	0.000 000 000 000 000 001	
zepto	z	10^{-21}	0.000 000 000 000 000 000 001	1991
yocto	y	10^{-24}	0.000 000 000 000 000 000 000 001	
ronto	r	10^{-27}	0.000 000 000 000 000 000 000 000 001	2022[3]
quecto	q	10^{-30}	0.000 000 000 000 000 000 000 000 000 001	

https://en.wikipedia.org/wiki/Metric_prefix

Computing Machinery Representations

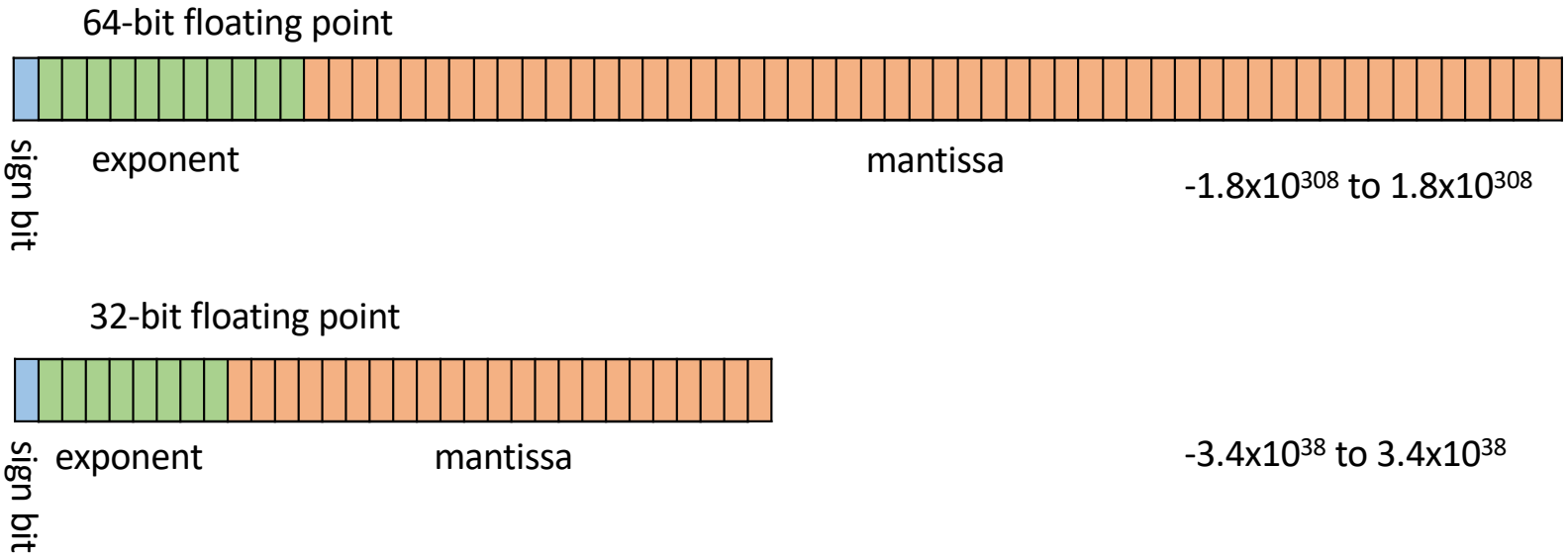


- Integer– 32 bits and 64 bits

Integers

- Consistent precision across the entire range
- Discrete steps between represented values

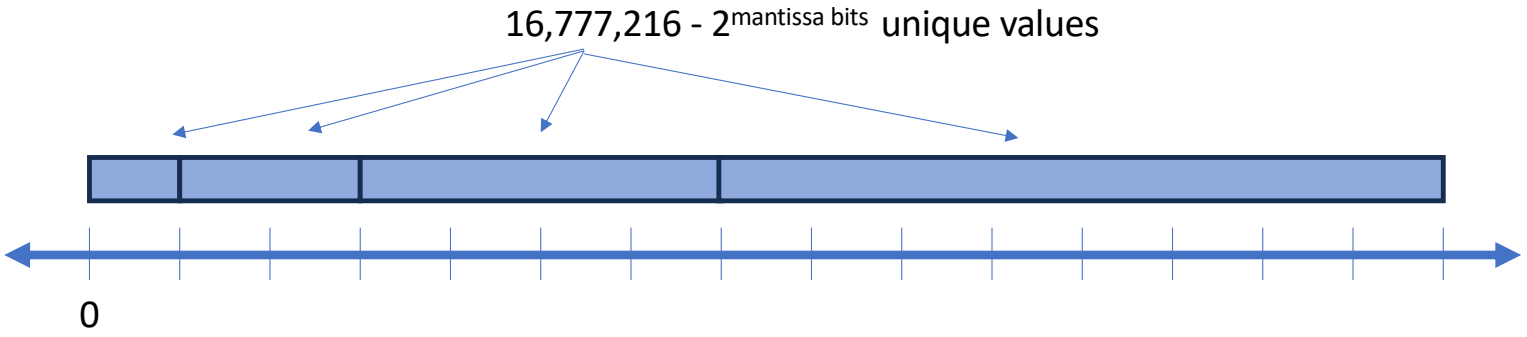
Computing Machinery Representations



- Floating point – IEEE standard 32 bits and 64 bits

Floating point

- Not consistent precision, and also discrete steps between values
  - High precision near 0
  - Lower precision for bigger numbers
  - For IEEE float 32, numbers above 2^{24} (16,777,216) are separated by 2.0



Problems

- Fun with Excel (running on a 64-bit CPU, in 2025)

	1.23E-100
	1.23E-100
sum:	2.46E-100

	1.23E-100
	1.23E-100
	3.45E+100
	7.89E+120
	-3.45E+100
	-7.89E+120
sum:	0

	3.45E+100
	7.89E+120
	-3.45E+100
	-7.89E+120
	1.23E-100
	1.23E-100
sum:	2.46E-100

	1.23E-05
	1.23E-05
	3.45E+05
	7.89E+09
	-3.45E+05
	-7.89E+09
sum:	2.48E-05

Problems

- Fun with Python (really numpy)

```
>>>
>>> import numpy
>>> a = numpy.float32(123)
>>> b = numpy.float32(123*10**9)
>>> a
 123.0
>>> a = a + b
>>> a = a - b
>>> a
 0.0
```

```
>>>
>>> import numpy
>>> a = numpy.float32(12345)
>>> b = numpy.float32(123*10**9)
>>> a
 12345.0
>>> a = a + b
>>> a = a - b
>>> a
 16384.0
>>>
```

```
>>>
>>> import numpy
>>> b = numpy.float32(123*10**9)
>>> b - numpy.float32(4096.0)
 123000000000.0
>>> b - numpy.float32(4097.0)
 122999990000.0
>>>
```

Algorithms Implemented in Hardware

- Fast Fourier transformations
- Audio filters
- Gaussian blurring
- Convolutions
- Inferencing
- Trigonometry
- And many more

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$\cos\alpha + \cos\beta = 2\cos\frac{1}{2}(\alpha+\beta)\cos\frac{1}{2}(\alpha-\beta)$$

$$(x+a)^n = \sum_{k=0}^{n} \binom{n}{k} x^k a^{n-k}$$

$$f(x) = a_0 + \sum_{n=1}^{\infty}\left(a_n \cos\frac{n\pi x}{L} + b_n \sin\frac{n\pi x}{L}\right)$$

Algorithms Implemented in Hardware

- Based on math
- Generally, there will be some software reference implementation
  - Presumably, well verified for some set of inputs
- Usually, computationally complex
  - Simple, lightweight computations can be left in software

The “Range” Problem

- An average function

```
int average(int data[], int count)
{
   int sum = 0;
   int i;

   if (count == 0) return sum;

   for (i=0; i<count; i++) {
     printf("data[%d] = %d \n", i, data[i]);
     sum += data[i];
   }
   return sum/count;
}
```

```
data[0] = 10
data[1] = 20
data[2] = 30
data[3] = 40
data[4] = 50
data[5] = 60
data[6] = 70
data[7] = 80
data[8] = 90
data[9] = 100
average is = 55
```

The “Range” Problem

- An average function

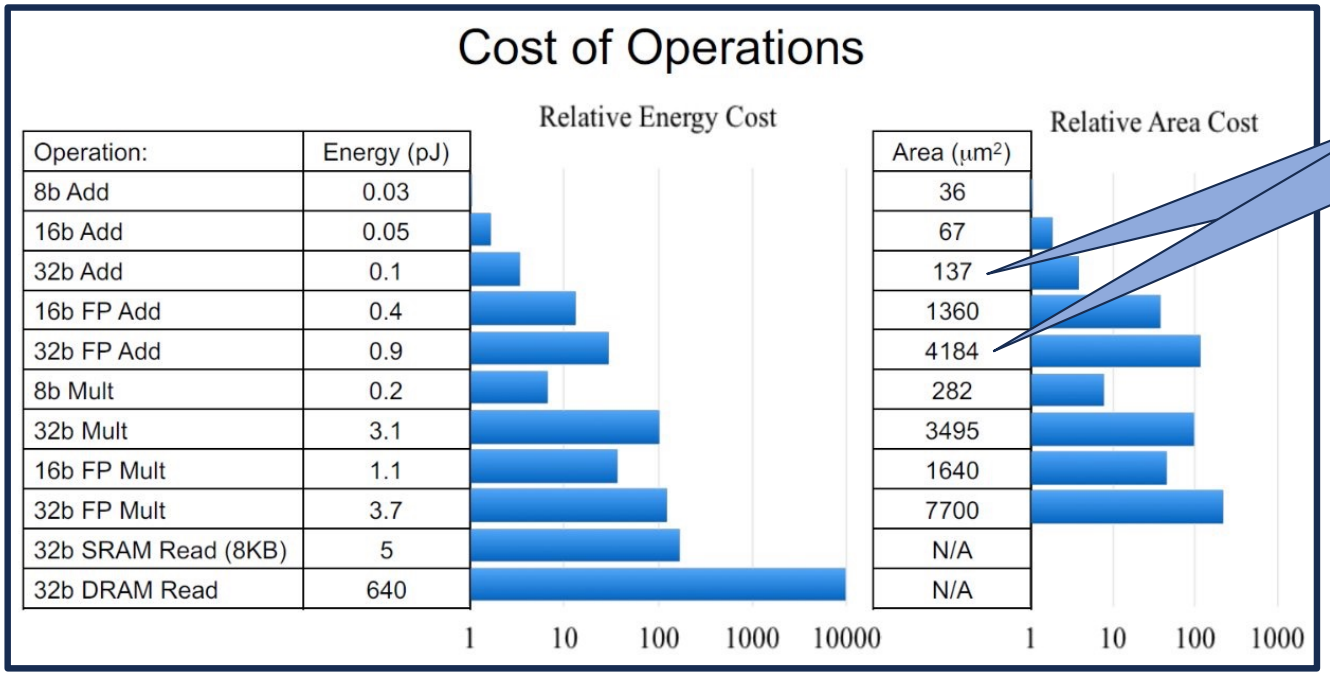
```
int average(int data[], int count)
{
   int sum = 0;
   int i;

   if (count == 0) return sum;

   for (i=0; i<count; i++) {
     printf("data[%d] = %d \n", i, data[i]);
     sum += data[i];
   }
   return sum/count;
}
```

```
data[0] = 217897144
data[1] = 417876878
data[2] = 422077472
data[3] = 418769530
data[4] = 439176704
data[5] = 505877146
data[6] = 596749270
data[7] = 406971490
data[8] = 439084336
data[9] = 430487396
average is = 7
```

Math in Hardware

- In a design, the representation can be anything we want
  - Obviously, integer registers are sized to be just big enough
- Why deviate from the reference implementation?
  - Power
  - Performance
  - Area

Data Sizes and Operators

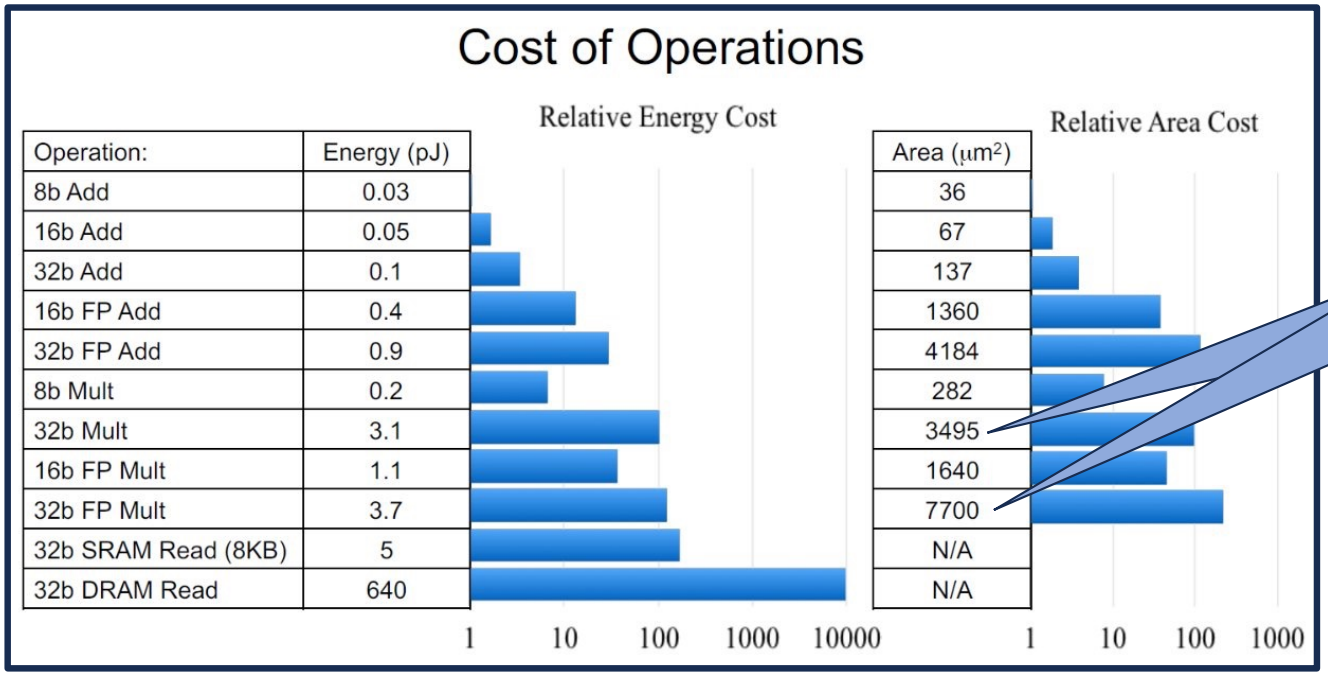


Operation:	Energy (pJ)	Area (μm²)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A

Adder: floating point is 37X bigger than integer

Source: Nvidia DAC2017

Data Sizes and Operators

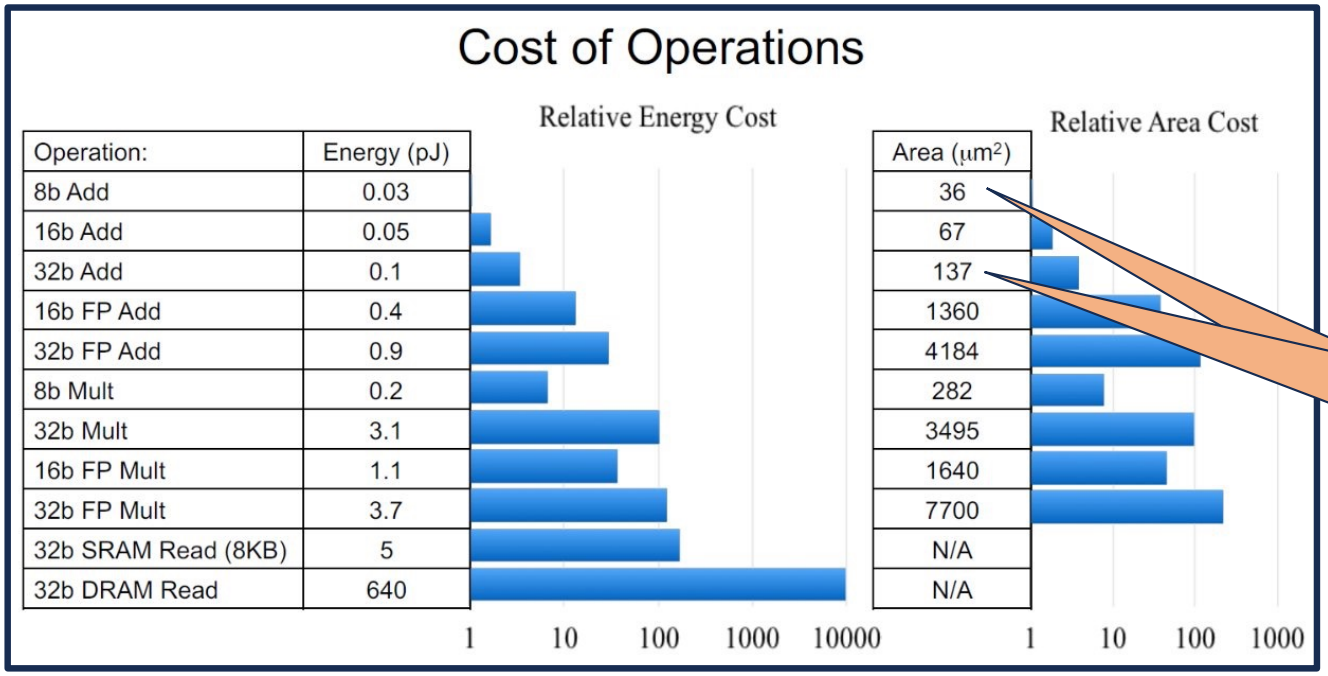


Operation:	Energy (pJ)	Area (μm²)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A

Mult: floating point is 2.2X bigger than integer

Source: Nvidia DAC2017

Data Sizes and Operators

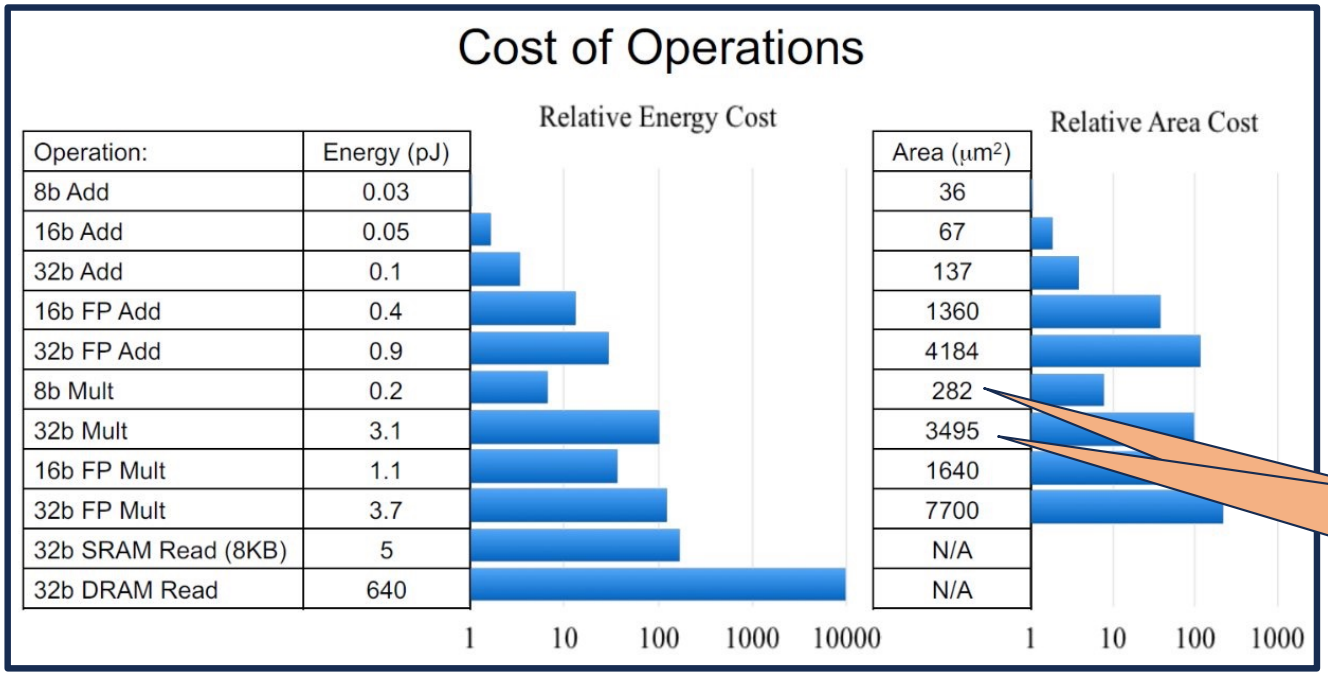


Operation:	Energy (pJ)	Area (μm²)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A

Adder: 32-bit is 3.8 X bigger than 8-bit

Source: Nvidia DAC2017

Data Sizes and Operators

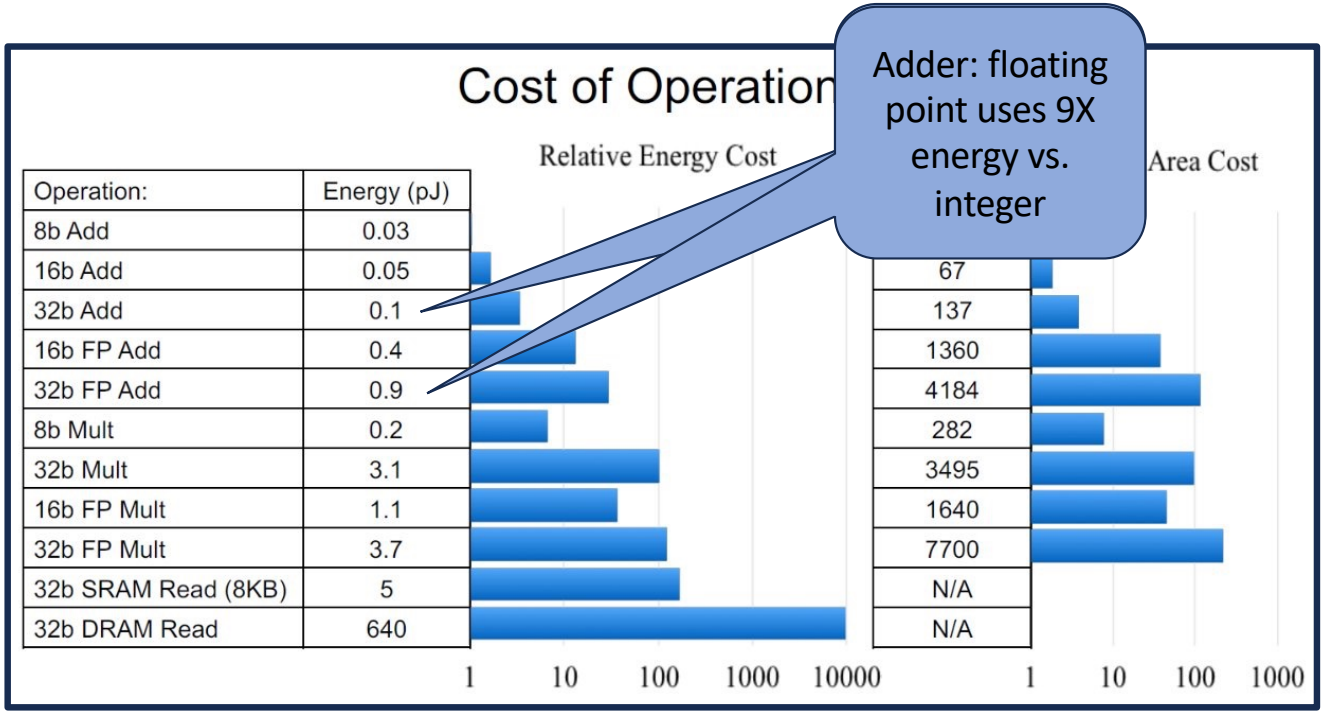


Operation:	Energy (pJ)	Area (μm²)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A

Mult: 32-bit is 12 X bigger than 8-bit

Source: Nvidia DAC2017

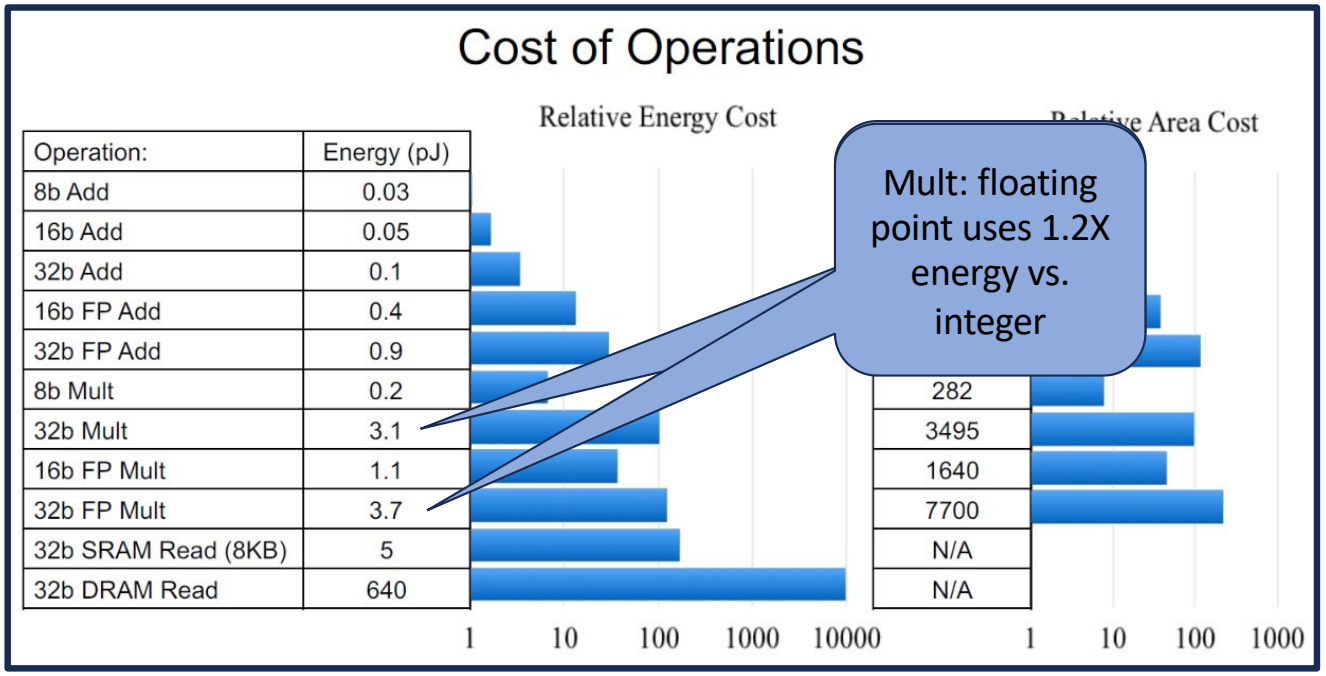
Energy and Operators



Operation:	Energy (pJ)	Area (μm²)
8b Add	0.03	67
16b Add	0.05	137
32b Add	0.1	1360
16b FP Add	0.4	4184
32b FP Add	0.9	282
8b Mult	0.2	3495
32b Mult	3.1	1640
16b FP Mult	1.1	7700
32b FP Mult	3.7	N/A
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	

Source: Nvidia DAC2017

Energy and Operators

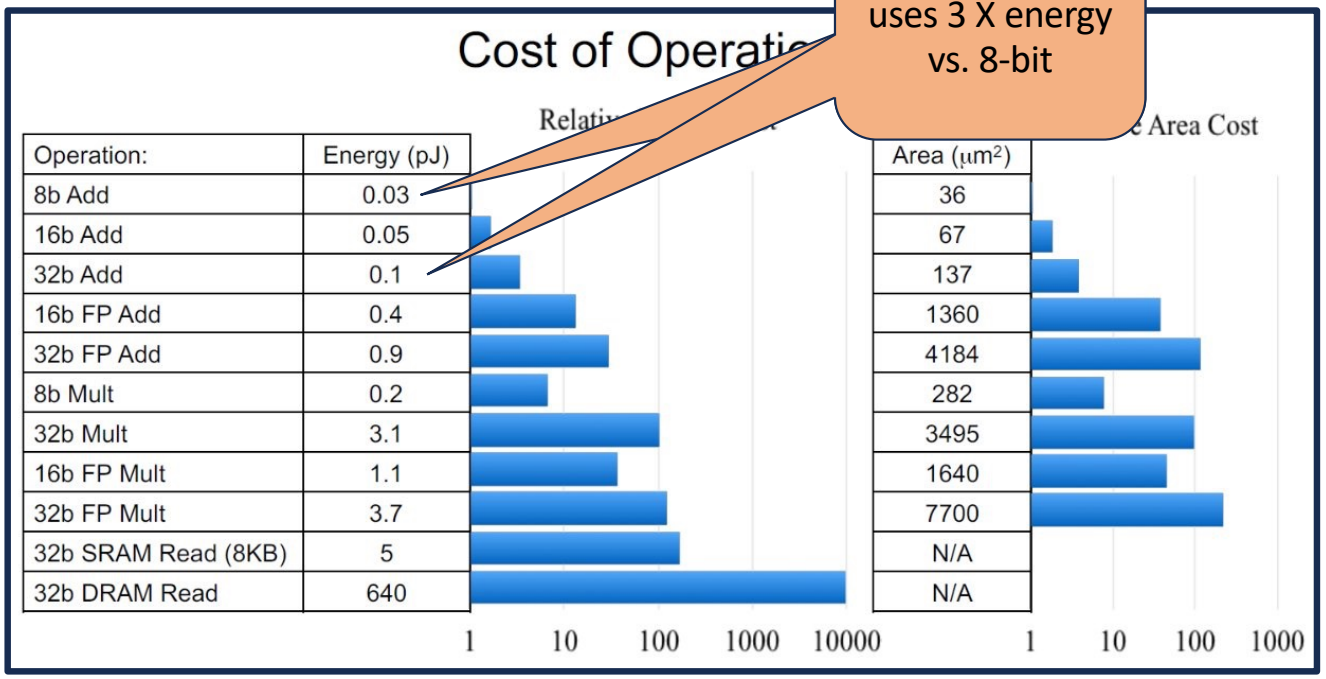


Operation:	Energy (pJ)	Relative Area Cost
8b Add	0.03	
16b Add	0.05	
32b Add	0.1	
16b FP Add	0.4	
32b FP Add	0.9	
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A

Source: Nvidia DAC2017

Energy and Operators

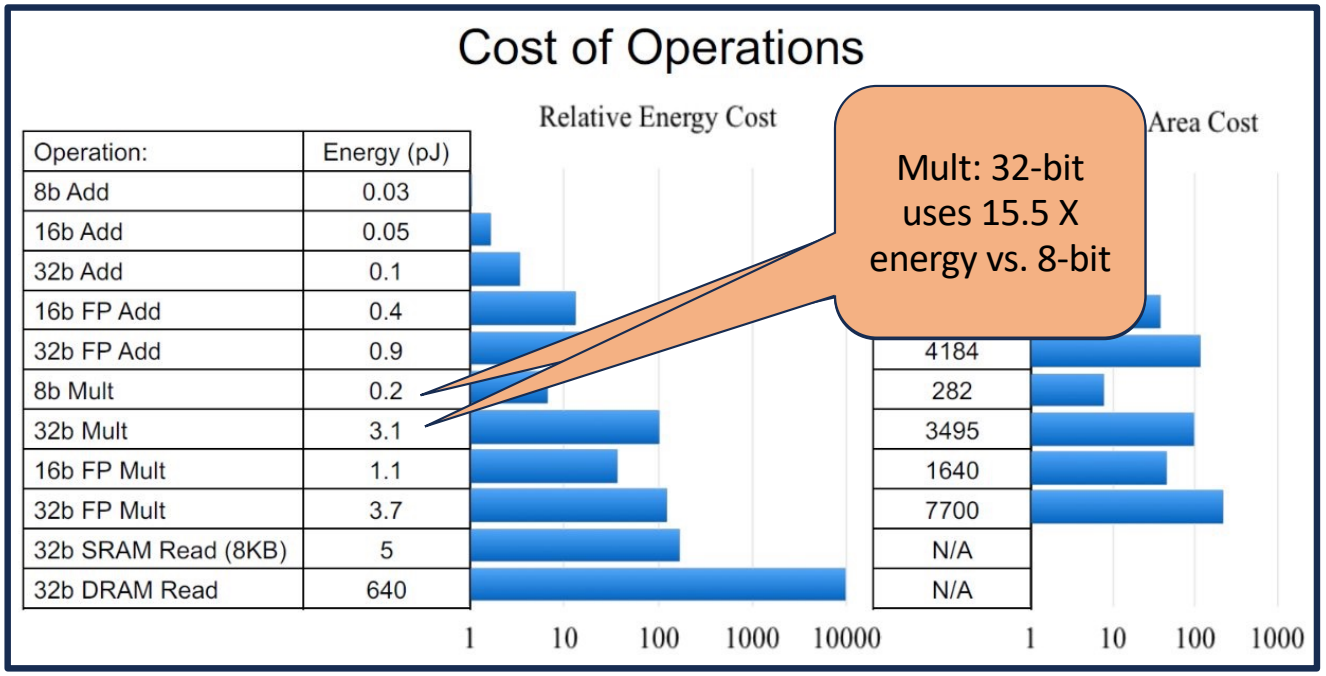
Adder: 32-bit uses 3 X energy vs. 8-bit



Operation:	Energy (pJ)	Area (μm²)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A

Source: Nvidia DAC2017

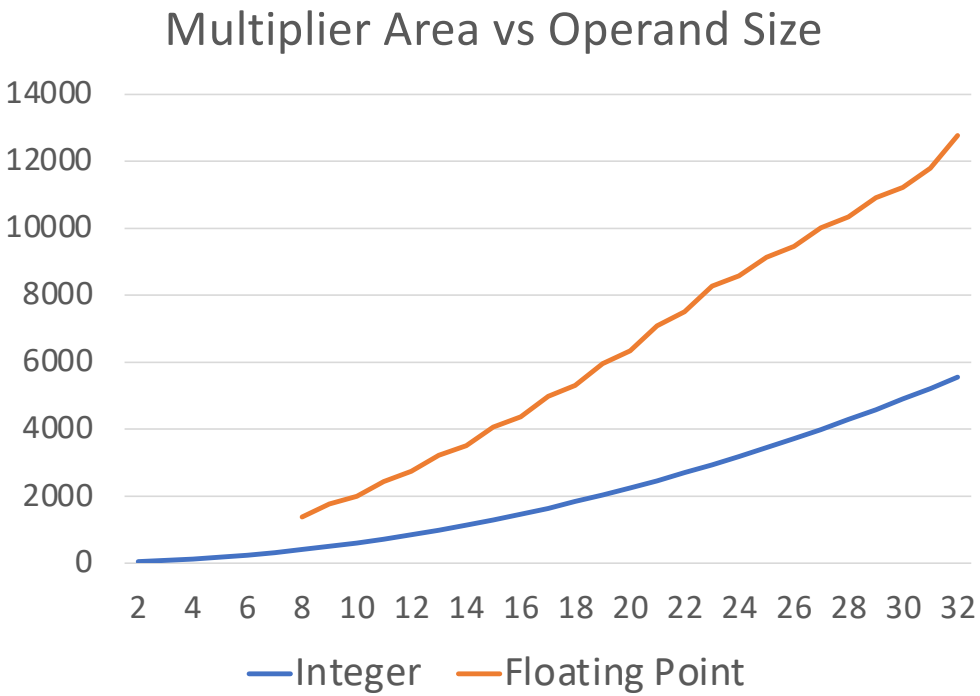
Energy and Operators

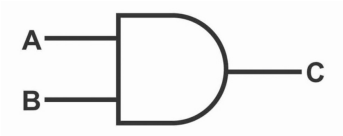


Operation:	Energy (pJ)	Area (μm²)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A

Source: Nvidia DAC2017

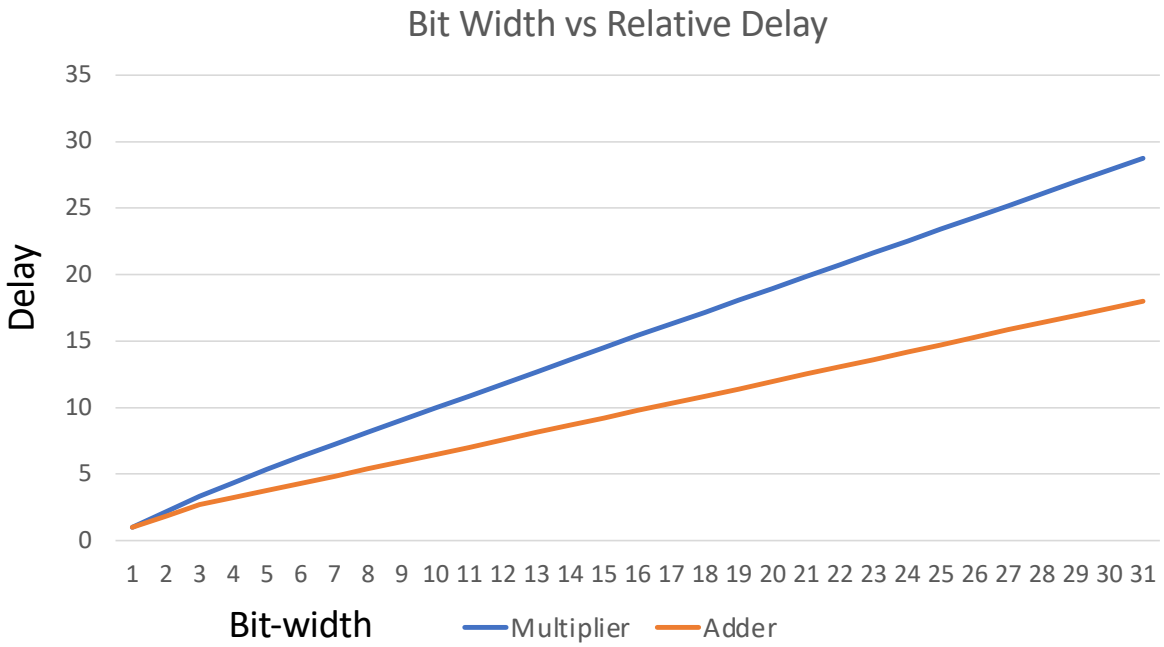
Smaller is Better





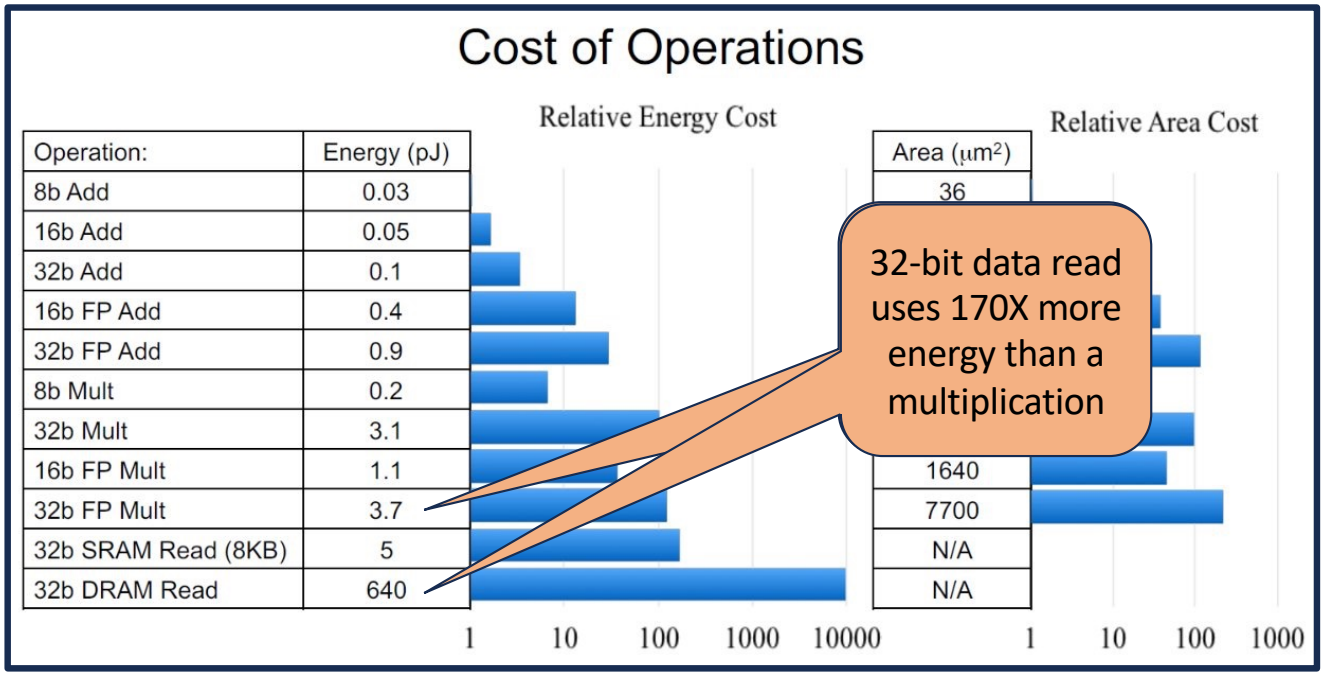
A one-bit integer multiplier is an “and” gate

Smaller is better



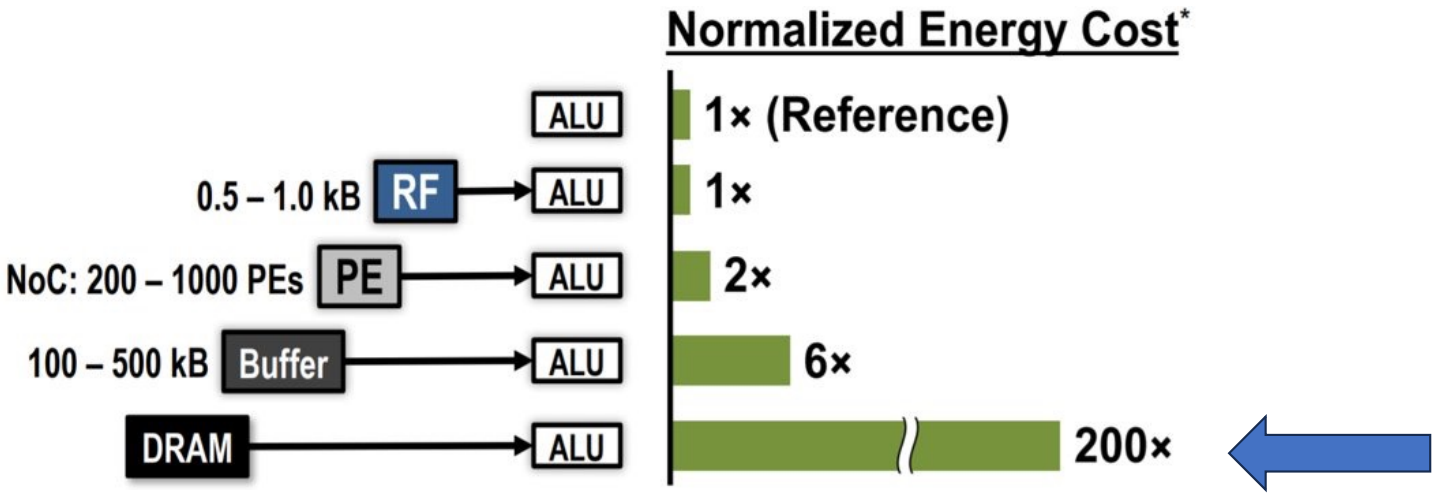
- Propagation delay is shorter when there are fewer gates
- 32-bit adder has a carry chain through 32 operators

Energy and Operators



Source: Nvidia DAC2017

Energy Cost of Data Movement



One of the largest energy consumers in your design

Source: Nvidia DAC2017

Energy Cost of Data Movement

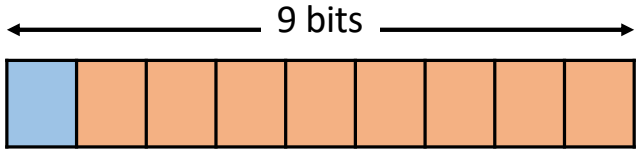
- Reducing the size of the data means:
  - Less data to store
  - Less data to move
- Algorithms that need to be in hardware often have large data sets
  - This is most pronounced in AI algorithms
- Reduce memory footprint by 2-3X
- Reduce movement costs by 2-3X

Alternate Representations

- Smaller integers
- Fixed point
- Varying sized floating point
  - bfloat_16
- Posits
- Indexes

Varying Sized Integers

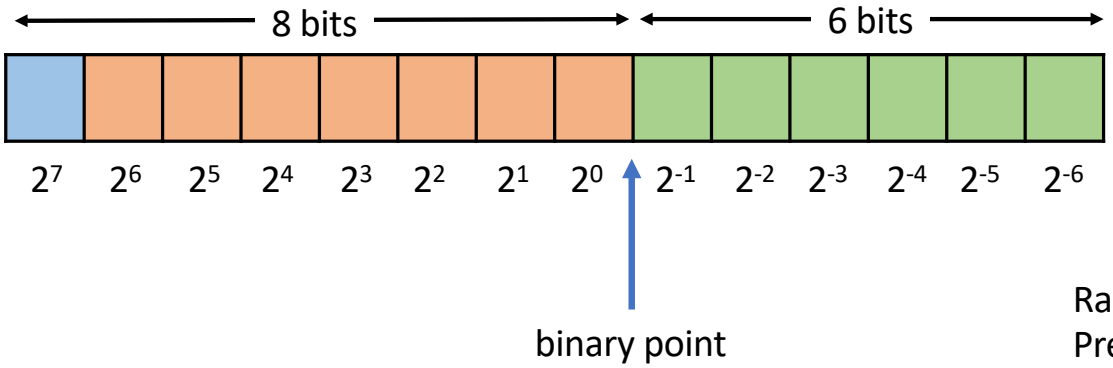
- Obvious (included for completeness)
- You don’t use a 32-bit integer when you have data values from 1 to 10
- Synthesis tools sometimes trim integers for you
  - If it can work out that bits are not needed



Range: -256 to 255
Precision: 1

Fixed Point

- Integer with an implied “binary point”



Range: -128.98 to 127.98
Precision: 2^{-6}, 0.0156

Fixed Point

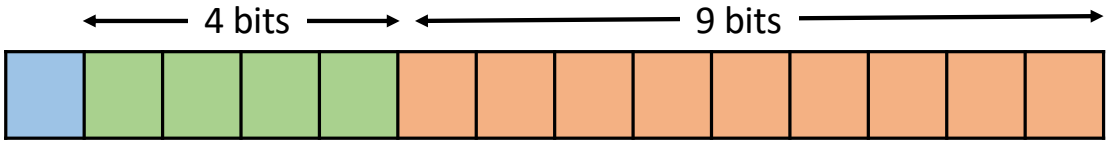
- Uses almost the same operators as integers
  - Multiplication and division operations need to mind the binary point

0.1 x 0.1 = 0.01 1 x 1 = 1 10 x 10 = 100

- Consistent precision across the entire range of value represented

Varying Sized Floating-Point

- Arbitrary sized exponent and mantissa



Range: -64K to 64K
Precision: Varies, best 0.0078

Varying Sized Floating-Point

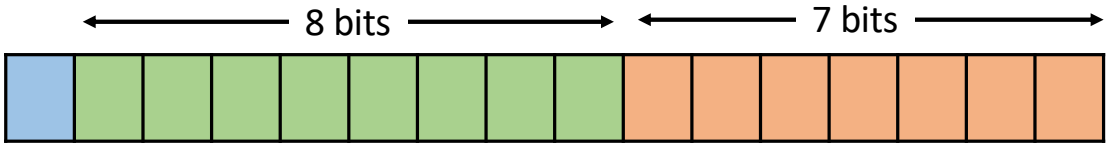
- Better precision near 0 and larger range than equivalent integer or fixed point
- For many computations standard sized floats are overkill
  - Excess bits in data storage, data movement, and operators

Bfloat-16 (or Google's Brain Float)

- Inferencing algorithms usually have numbers near 0
- As numbers get big precision is less important
  - Just knowing that the quantity is "big" is often sufficient
- AI researchers at Google came up with 16-bit floating point optimized for AI training and inferencing
  - Smaller data, less data movement, and smaller silicon have a big impact on training costs
- Google's TPUs support this format
- RISC-V extensions can support this in vector unit

Bfloat-16

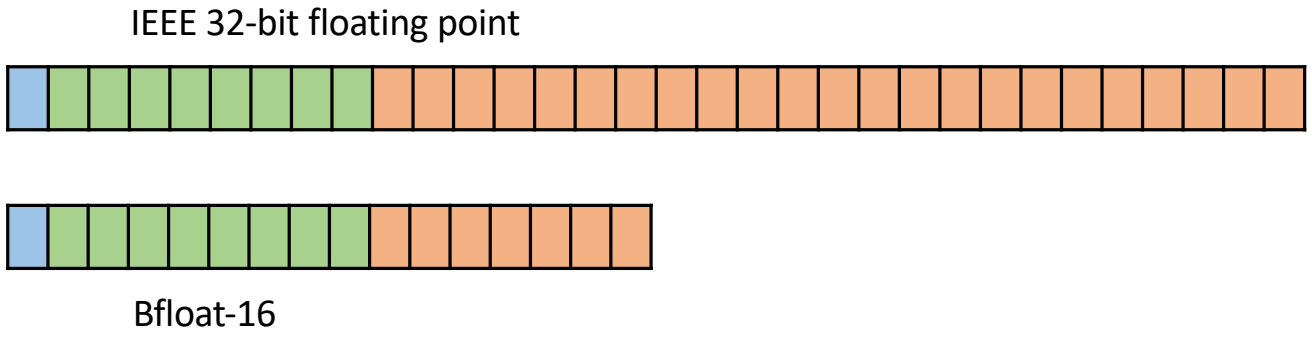
- Brain float



Range: -10^{38} to 10^{38}
Precision: Varies

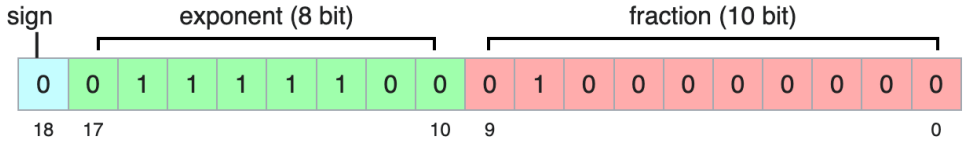
Bfloat-16 to IEEE Float-32

- Key value is fast conversion to and from native float-32

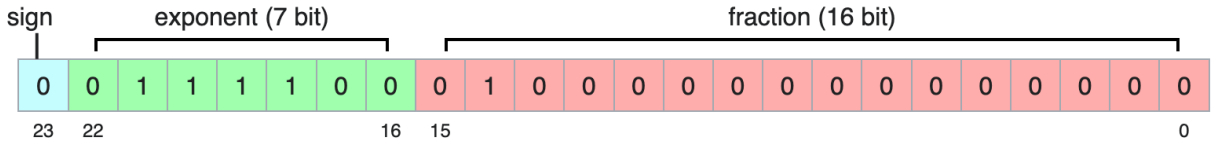


Other Notable Floating-Point Formats

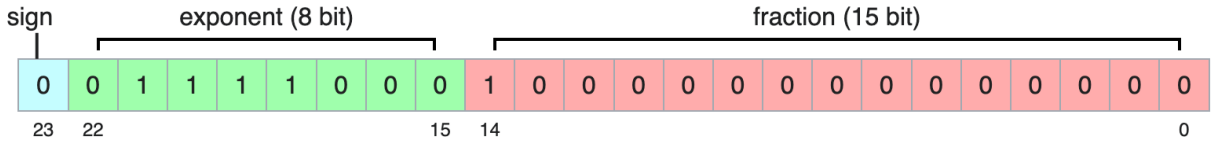
Nvidia's TensorFloat-32 (19 bits)



AMD's fp24 format



Pixar's PXR24 format

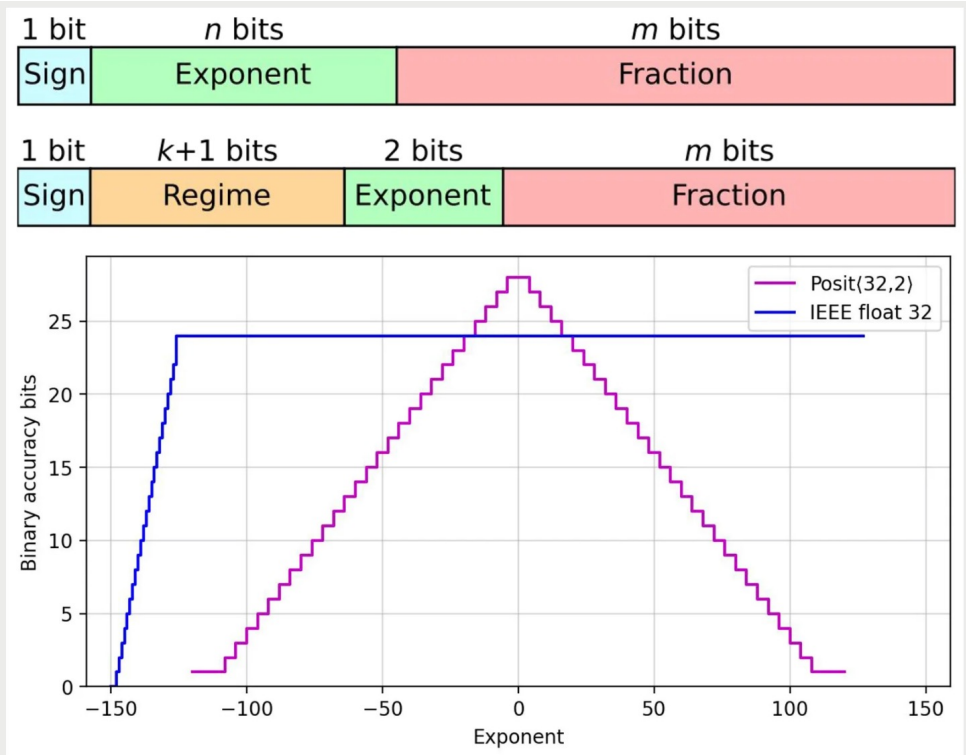


source: Wikipedia

Posits

- Gustafson and Yonemoto developed these in 2017 for AI
- Adds “regime” bits
  - Essentially, an exponent for the exponent
- Gives better precision near zero
  - And larger range than equivalently sized floating point
- Claim significantly better accuracy for training and inferencing with smaller representation

Posits



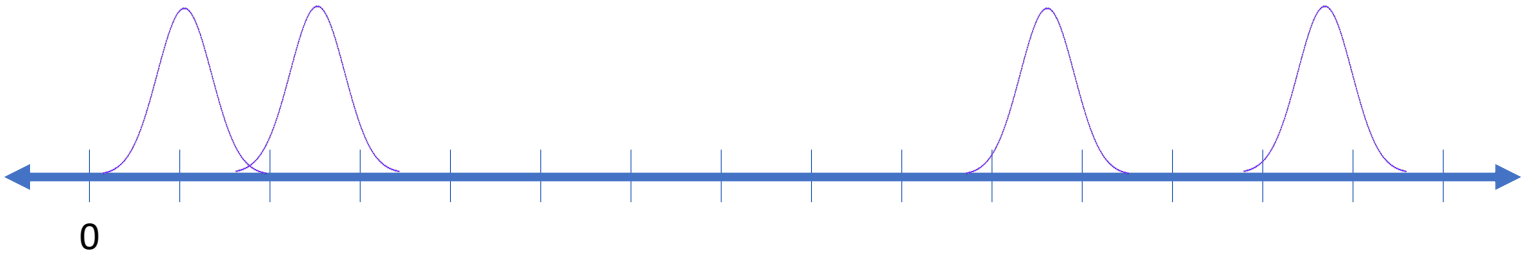
Traditional Float

Posits

Source: https://spectrum.ieee.org/floating-point-numbers-posits-processor

Indexes (or Look-up Table)

- For ”lumpy” non-contiguous irregular data
- Can potentially save space (and movement) by loading values into a memory and storing/moving an index



Alternate Representations

- Smaller integers
- Fixed point
- Varying sized floating point
  - bfloat_16
- Posits
- Indexes

- This was sample of a few ways to represent numbers for computation
- Limited only by designer creativity

Computing with Alternate Formats

- **Any** change to the representation will change the math
  - And can affect the results of computations
  - There will be **less precision** and a **smaller range**

- Key issues
  - Overflows
  - Rounding

Overflow

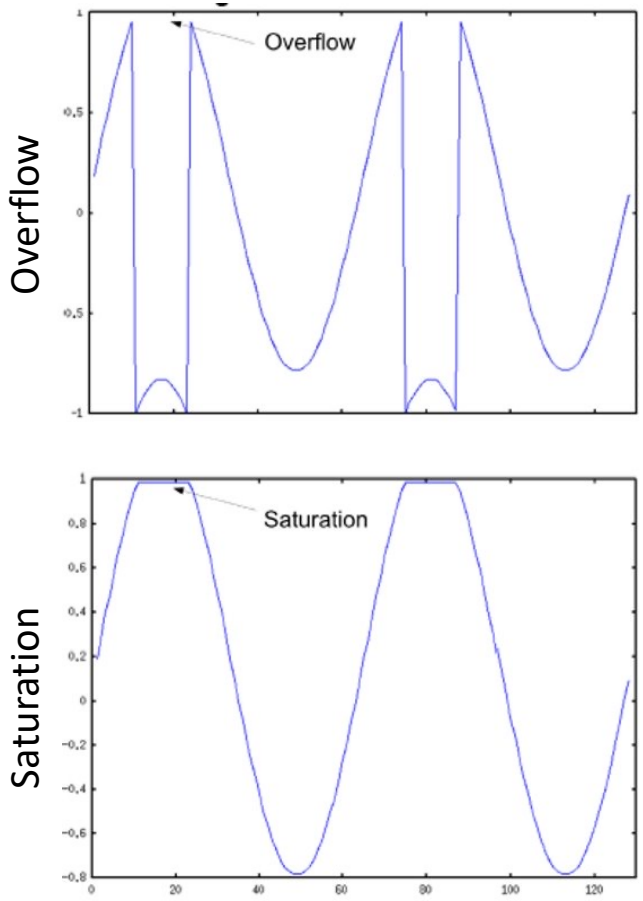
- Overflow happens when the result of the computation is larger than can be represented by the register/variable
- For integers, high order bits are dropped
  - Result is incorrect, all subsequent calculations are corrupted
- For floats, value becomes INF (infinity)
  - Subsequent operations result in either INF or NaN

Overflow

- With smaller representations, overflow is a bigger concern
- With 32-bit and 64-bit CPUs overflow is uncommon
  - Older 8/16 bit CPUs have overflow flags and “add with carry” instructions
- Overflow handling at the HW level is rarely propagated to SW
- Fun Fact
  - RISC-V architecture has no ADD with CARRY instruction
  - And no carry/overflow/borrow bit in ISA

Saturation

- Saturating math sets the output value to the maximum representable value
  - Does not just drop bits
- In some cases, this produces acceptable results



Saturating Math

- Saturating math assigns largest possible value instead of dropping bits
- With 10-bit signed fixed-point number with 7 integer bits:

Overflow:

```
 62.5     0 1 1 1 1 1 1 . 1 0 0
+ 2.0               1 0 . 0 0 0
-----------------------------------
- 1.5     1 1 1 1 1 0 1 . 1 0 0
```

REALLY WRONG!

Saturation:

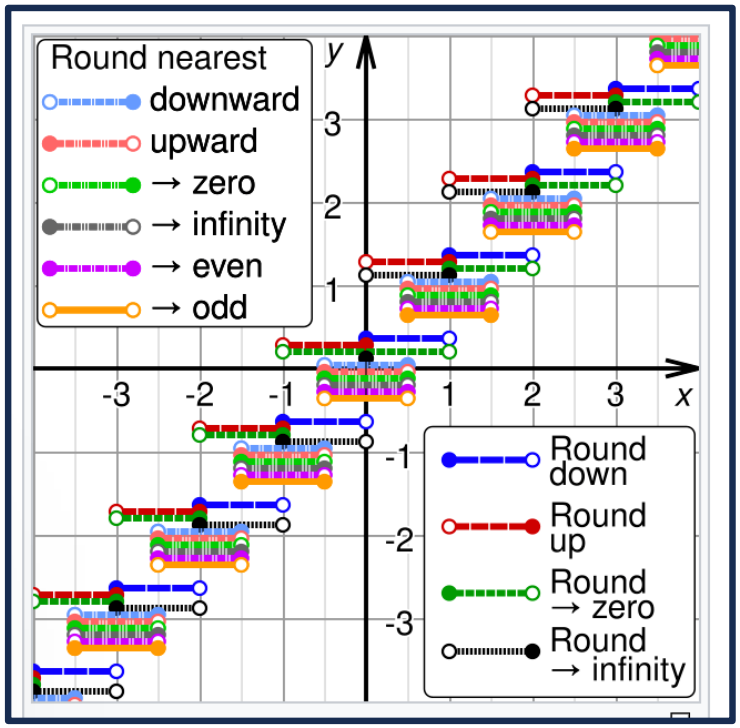
```
 62.5     0 1 1 1 1 1 1 . 1 0 0
+ 2.0               1 0 . 0 0 0
-----------------------------------
 63.875   0 1 1 1 1 1 1 . 1 1 1
```

Close to correct

Rounding

- Integer adders truncate results:
  - In C code: printf(“ %d \n”, 1000 * (99/100));
  - Results in “0”
- Assigning integers from any other type truncates low order bits
  - Consistent in most programming languages and with bit-vector types in Verilog
- As representation’s range shrinks, truncation error becomes more significant

Rounding Models



- Many different rounding models
- Rounding will affect results
- All verification models must be bit-level consistent

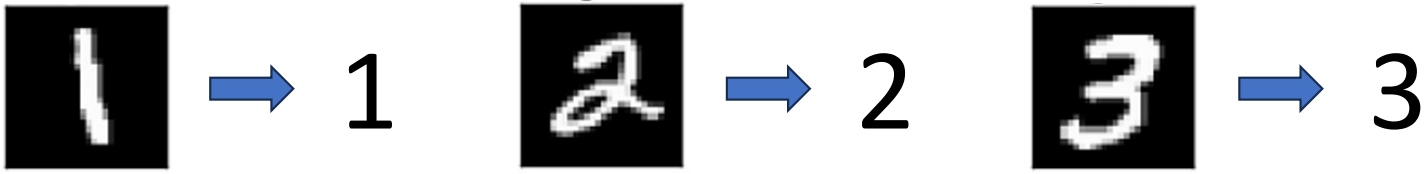
Image source: https://en.wikipedia.org/wiki/Rounding

Examples

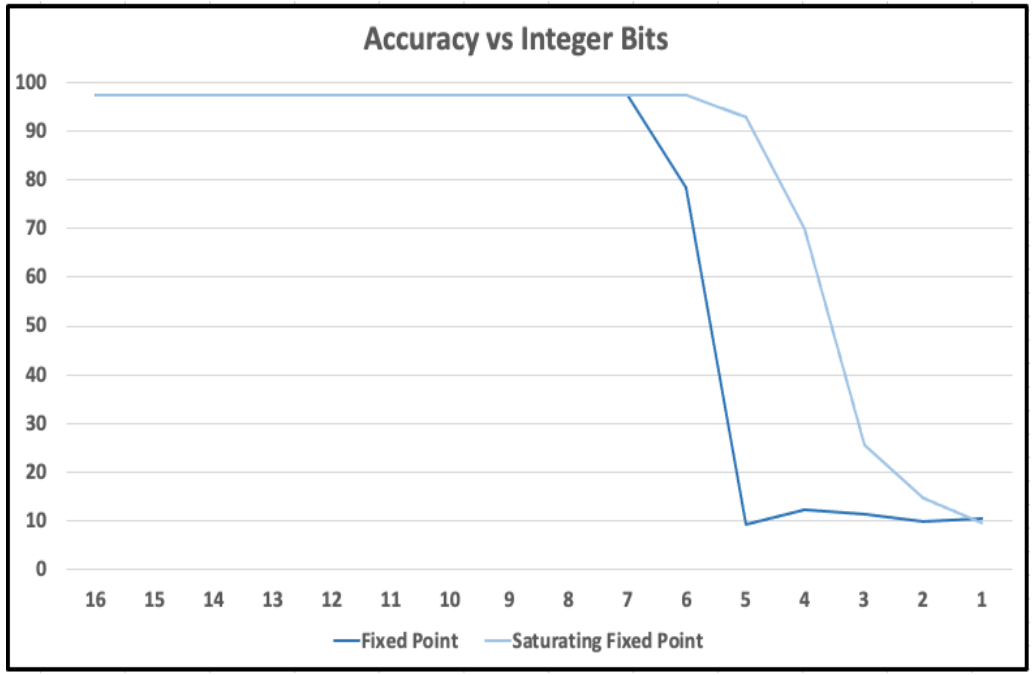
- Inferencing accelerator
  - MNIST handwritten character recognition
  - Conversion from float32 (Python) to 10 bit fixed point (Verilog)
- Audio processing
  - MFCC calculation (MEL Frequency Coefficient Calculation)
    - Fast Fourier transformation of PCM audio to
  - Conversion from double (C++) to bespoke 8-bit float (Verilog)

MNIST Handwritten Character Recognition

- "Hello, World!" example for machine learning
  - Small, but runs millions of multiply accumulate operations
  - Performs many 2d convolutions and vector/matrix multiplications
  - Good platform for exploring implementation and optimization methods



Fixed Point Size Analysis



- High-water mark of data and intermediate values showed range of values was -37 to 56
  - Float32 (+/-10^{38} is excessive)
- Sensitivity analysis performed across varying fixed-point representations

Fixed Point Conversion Results

Integer Bits

Fractional Bits	8	7	6	5	4	3	2	1	0
8	98.05	98.05	98.05	97.55	76.75	28.70	18.00	16.80	14.90
7	97.85	97.85	97.85	97.25	75.39	27.90	17.50	16.60	15.40
6	97.13	97.95	97.91	97.45	75.15	28.30	17.30	15.90	13.90
5	97.21	98.08	98.10	97.40	72.57	24.50	16.90	15.20	14.90
4	96.94	97.79	97.76	95.71	59.90	21.40	16.20	13.10	15.10
3	95.56	96.37	96.35	90.08	38.83	16.70	14.00	11.50	12.70
2	82.31	83.13	83.13	64.73	22.70	14.90	12.30	10.50	8.50
1	30.15	30.97	30.92	33.72	32.07	24.60	34.90	12.30	8.50
0	9.53	9.33	9.50	9.37	9.37	8.50	8.50	8.50	10.00

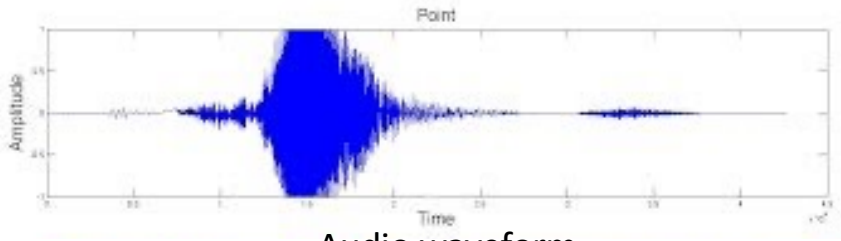
Accuracy

Computational resources: Area = -78% Speed = 2.4X faster Energy = -92%

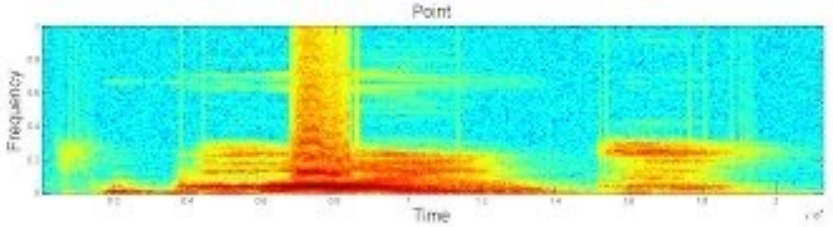
MFCC – MEL Frequency Cepstrum Coefficients

- Coverts audio waveform into an audio 2D spectrogram
  - “MEL” frequencies correspond to human ear sensitivities
  - Computes audio energy in a range of frequencies and their changes over time
  - An FFT is computed for each frequency and timeslice
- Spectrograms are used in voice to text, wakewords, and voiceprinting

Mic



Audio waveform



Spectrogram

Float32 to 8-bit Float

- Fixed Point did not work at all
- Computed average error in spectrogram values
  - Accuracy started dropping at 10 bits
- 8-bit accuracy was good enough for downstream processing

Computational resources: Area = -89% Prop delay = -62% Power = -84%

Verification of Modified Algorithm

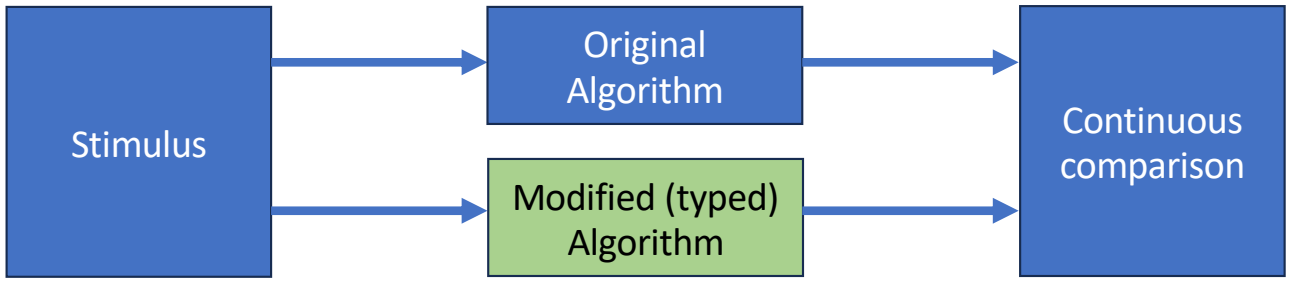
- Need to fully verify operators, computational operations
- Also need to verify that algorithm produces correct results
  - with modified representation
- Algorithms are likely too large to analytically size representation
  - MNIST had > 4 million MAC operations
  - MFCC performed 2,000 FFT calculations on input waveform
- Need to run large data sets
  - Both real and synthetic data
  - Verification will only be as good as the stimuli

Key Verification Points

- High and low watermarks for inputs, output, and intermediate data
  - Need to understand largest and smallest values represented at each point in the algorithm
- When overflows occur
  - Need to know anytime operation results in overflow of the representation
- These observations will change with each alteration to the representation
- Will require multiple runs at different representations
- Must have bit level accuracy matching ultimate RTL implementation

Run in Tandem with Original Algorithm

- Run modified algorithm in parallel with the original algorithm
- Perform on-the-fly comparisons
  - Ensure needed accuracy is maintained
  - Monitor both outputs and intermediate values



Performance Needed

- Algorithms are likely large and complex
  - Else would be left in software
- Logic simulation is probably too slow
  - Yolo-2 – small object recognition algorithm – at RTL takes ~5 hours 1 inference
  - About 6 CPU years for logic simulation of 10,000 inferences
- Raise the abstraction level
  - Can run 1,000X faster
  - But need to ensure bit level activity of math is identical to intended implementation

AC Data Types

- Bit accurate C++ library of numeric representations and operators
  - Templated typedefs for various representations
  - Extensions for bit level manipulation, for hardware operations
- Supports bit level accurate representation of various data formats
  - Overloaded operators for all types perform bit level accurate operations
  - Easy migration from C++ or other high-level languages
- Supports multiple rounding models and saturating math

`http://hlslibs.org`

AC Data Types

- Variable sized int - ac_int<size, signedness>;
- Fixed point numbers - ac_fixed<size, integer_bits, signedness>
- Variable sized float - ac_float<size, exponent_size>;
  - Bfloat-16 - ac_bfloat<>;
- IEEE float 32 and 64 - ac_ieee_float32<>, ac_ieee_float64<>;
- complex numbers - ac_complex<type>;

Value Range Analysis

- Records high and low watermarks for variables throughout algorithm
- Identifies overflow and underflow points

- Essentially, assertions and monitors in the ac_type classes

- This information enables zeroing in on the correct representation quickly

Synthesis from C++ for AC types

```
361  void copy_to_regs(hw_cat_type *dst, index_type dst_offset, raw_memory_line *src, index_type src_offset, index_type size)
362  {
363      // read out of internal memories to an array of registers
364      // *should* make it easy for catapult to pipeline access to internal memories
365
366      index_type count;
367      index_type n;
368
369      static const index_type stride = STRIDE;
370
371      count = 0;
372      while (count < size) {
373          n = stride;
374          if ((size - count) < stride) n = size - count; // mis-aligned at the end of transfer
375          read_line(dst, dst_offset, src, src_offset, n);
376          count += n;
377          src_offset += n;
378          dst_offset += n;
379      }
380  }
```

C/C++ or SystemC

Automated path from C/C++ or SystemC into technology optimized synthesizable RTL

Catapult
High-Level
Synthesis

```
1   module timer (
2       input          clock,
3       input          resetn,
4       input  [1:0]   trans,
5       input  [29:0]  address,
6       input  [3:0]   bl,
7       input          we,
8       input          ce,
9       input  [31:0]  write_data,
10      output [31:0]  read_data,
11      output [1:0]   resp,
12      output         ready
13    );
14
15    reg [31:0]  timer_value;
16    reg [31:0]  rd_reg;
17
18    //reg         ready_out = 1'b1;
19    reg         resp_out  = 2'b00;
20
21    ready_gen #(1) rg (clock, resetn, ce, trans, ready_out);
22
23    assign ready          = ready_out;
24    assign resp           = resp_out;
25    assign read_data      = rd_reg;
26
27    always @(posedge clock or negedge resetn) begin
28      if (resetn == 1'b0) begin
29        timer_value <= 32'h00000000;
30      end else begin
31        timer_value <= timer_value + 32'h00000001;
32      end
33    end
34
35    always @(posedge clock or negedge resetn) begin
36      if (resetn == 1'b0) begin
37        rd_reg <= 32'h00000000;
38      end else begin
39        //if (trans[1] & ce) begin
40        if (ce & !we) begin
41          rd_reg <= timer_value;
42        end
43      end
44    end
45
46  endmodule
```

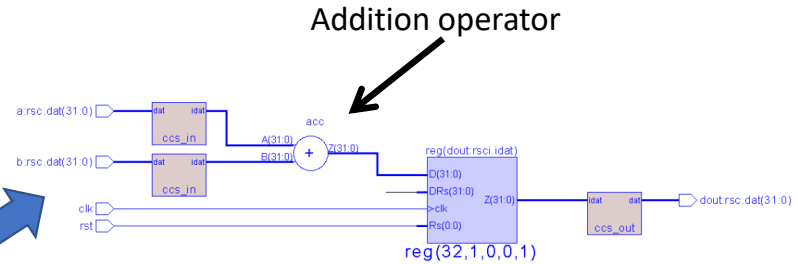
Synthesizable RTL

Catapult Hight-Level Synthesis

- Optimized for ASIC or FPGA
- Output is VHLD or Verilog

```
void add(int a, int b, int &dout){
  dout = a + b;
}
```

Catapult HLS




```
16  module add_core (
17    clk, rst, a_rsc_dat, b_rsc_dat, dout_rsc_dat
18  );
19    input clk;
20    input rst;
21    input [31:0] a_rsc_dat;
22    input [31:0] b_rsc_dat;
23    output [31:0] dout_rsc_dat;
24
25
26    // Interconnect Declarations
27    wire [31:0] a_rsci_idat;
28    wire [31:0] b_rsci_idat;
29    reg [31:0] dout_rsci_idat;
30    wire [32:0] nl_dout_rsci_idat;
31  ...
32    always @(posedge clk) begin
33      if ( rst ) begin
34        dout_rsci_idat <= 32'b0;
35      end
36      else begin
37        dout_rsci_idat <= nl_dout_rsci_idat[31:0];
38      end
39    end
40    assign nl_dout_rsci_idat  = a_rsci_idat + b_rsci_idat;
41  endmodule
```

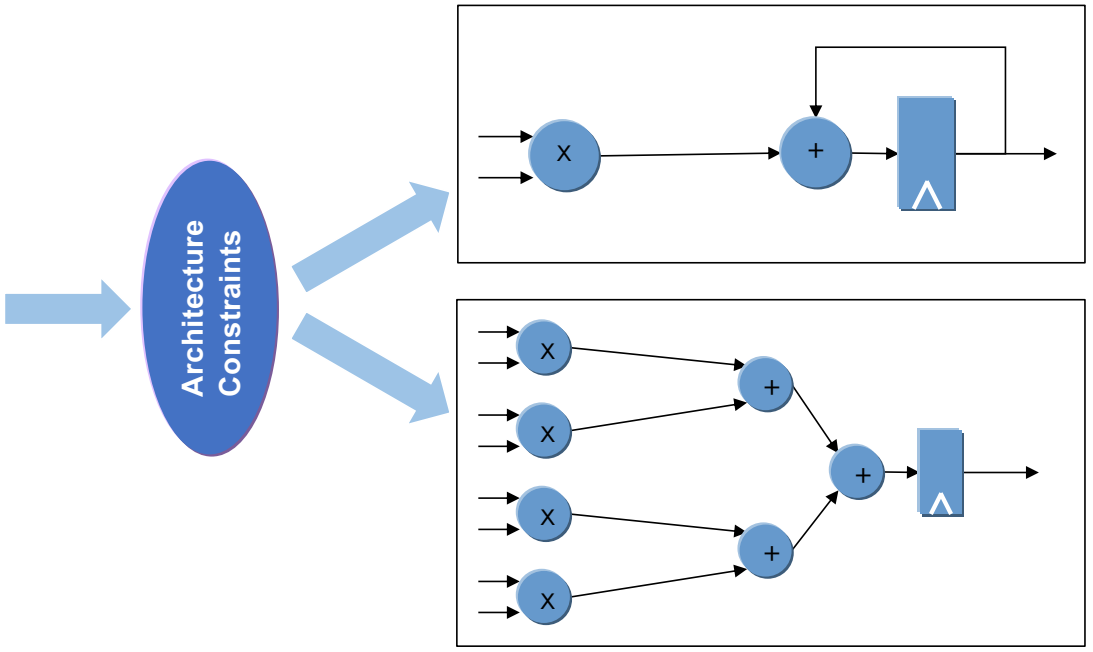
Clock and reset

Addition operator

Architectural Exploration

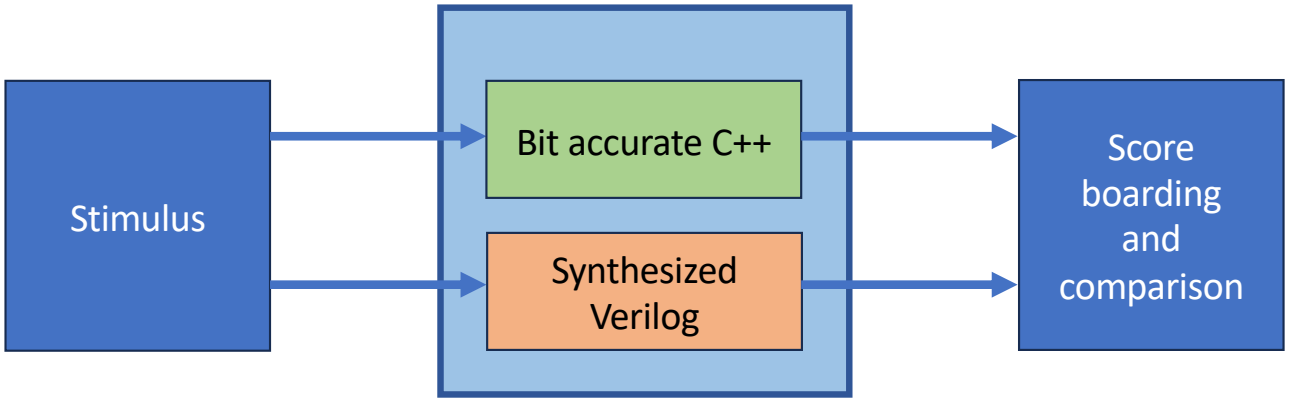
User controls implementation with constraints and pragmas

```
data_t MAC (
  data_t data_in[4],
  coef_t coef_in[4]
) {

  accu_t acc = 0 ;

  for (int i=0;i<4;i++) {
    acc += data_in[i] * coef_in[i] ;
  }
  return acc ;
}
```

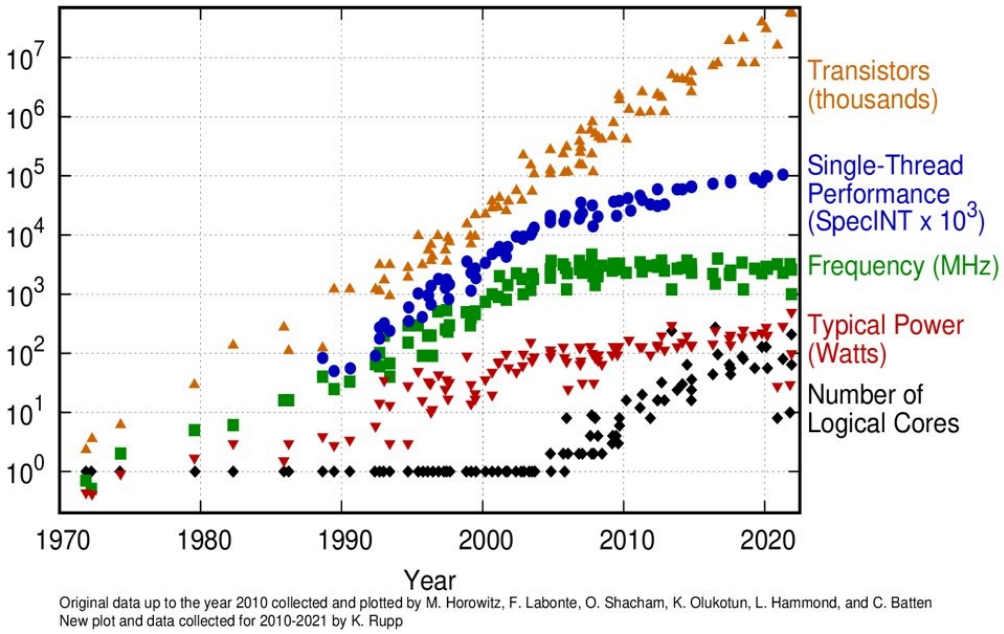


Verification of RTL against C++

- Compare C++ against RTL with
  - Formal equivalency (SLEC)
  - Coverage
  - Dynamic simulation



Technology Trends



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

- CPU single thread performance has stalled
- Performance demands continue to rise
- More algorithms moving to hardware

Conclusion

- Using alternative numeric representations when implementing designs can greatly improve PPA
- Need to carefully migrate from original algorithm to new representations, then to RTL
  - Continuous verification against the prior implementation is essential
  - Compare not just results but intermediate values as well
- Must ensure new representation maintains fidelity of the algorithm

Thank you

?? || //

Questions or comments

Russell.Klein@siemens.com
https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/
http://hlslibs.org

accellera SYSTEMS INITIATIVE