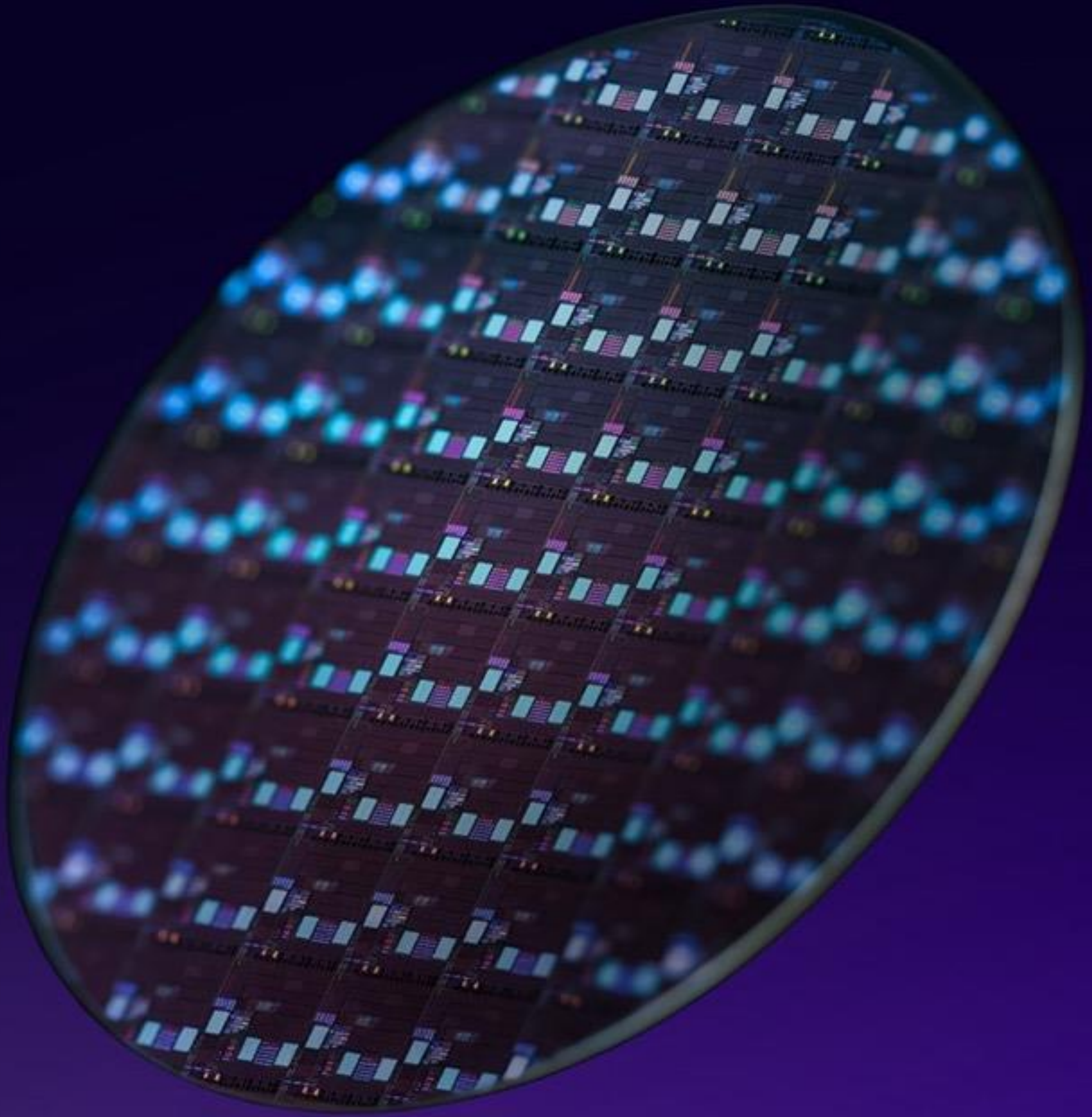
Floating-Point in Transition: Bridging Scientific Computation and AI Acceleration

From Academy to Industry



Javier D. Bruguera

Arm Cambridge, UK

RAIM meeting 2025, A Tribute to Jean-Michel Muller

Lyon, November 6th, 2025

Floating-Point in Transition: Bridging Scientific Computation and AI Acceleration

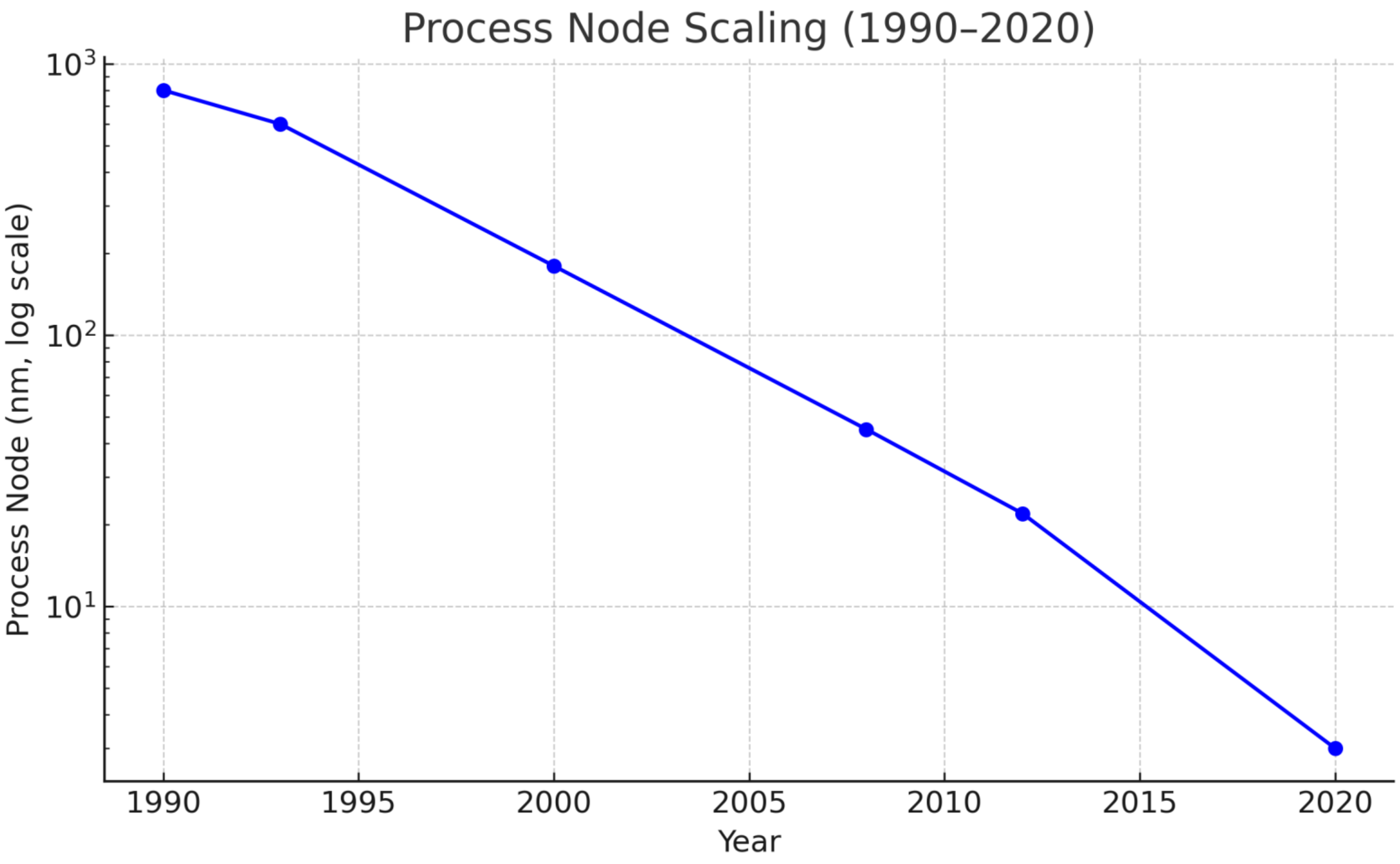
- **Historical context** – IEEE 754 and the dominance of 32- and 64-bit FP for scientific workloads
  - Portability, reproducibility, and precision
  - For decades 32 and 64-bit FP formats dominated HPC, weather simulation, physics and scientific computing
- **The AI shift** – rise of low-bit, workloads-specific formats and new operations
  - Modern AI workloads don't need full precision, they benefit from low-bit formats
  - Hardware tailored for throughput, energy efficiency and domain-specific operations
- **Dual perspective** – academic insights shaping theory, industry needs driving adoption
  - Academia pushes innovation in novel number representations, error analysis, and algorithmic resilience
  - Industry accelerates adoption by embedding these ideas into CPUS, GPUs, TPUs, and custom accelerators
- **Future outlook** – flexible, heterogeneous FP microarchitectures
  - Floating-point design will become more flexible and heterogeneous, combining multiple formats and operations within the same system

Floating-Point in Transition: Bridging Scientific Computation and AI Acceleration

1. Evolution of Processor Technology
2. Floating-Point Arithmetic in Big Cores
3. Future (and Present) of Floating-Point Formats and Operations
4. Conclusions

Evolution of Processor Technology

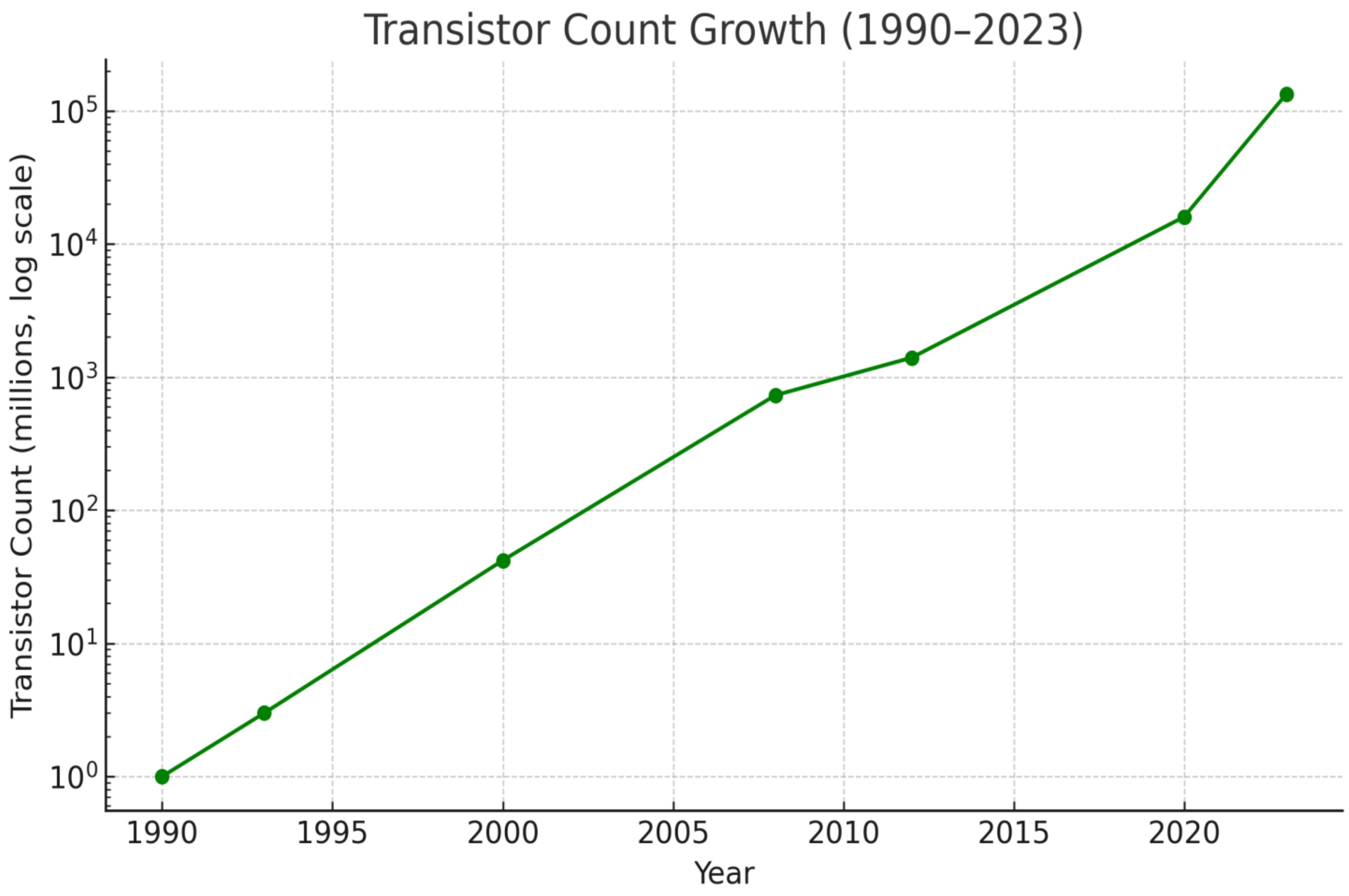
Process Technology (Feature size)



- **1990s:** ~800 nm → 350 nm
- **2000s:** 250 nm → 65 nm
- **2010s:** 45 nm → 14 nm
- **2020s:** 10 nm → 2 nm

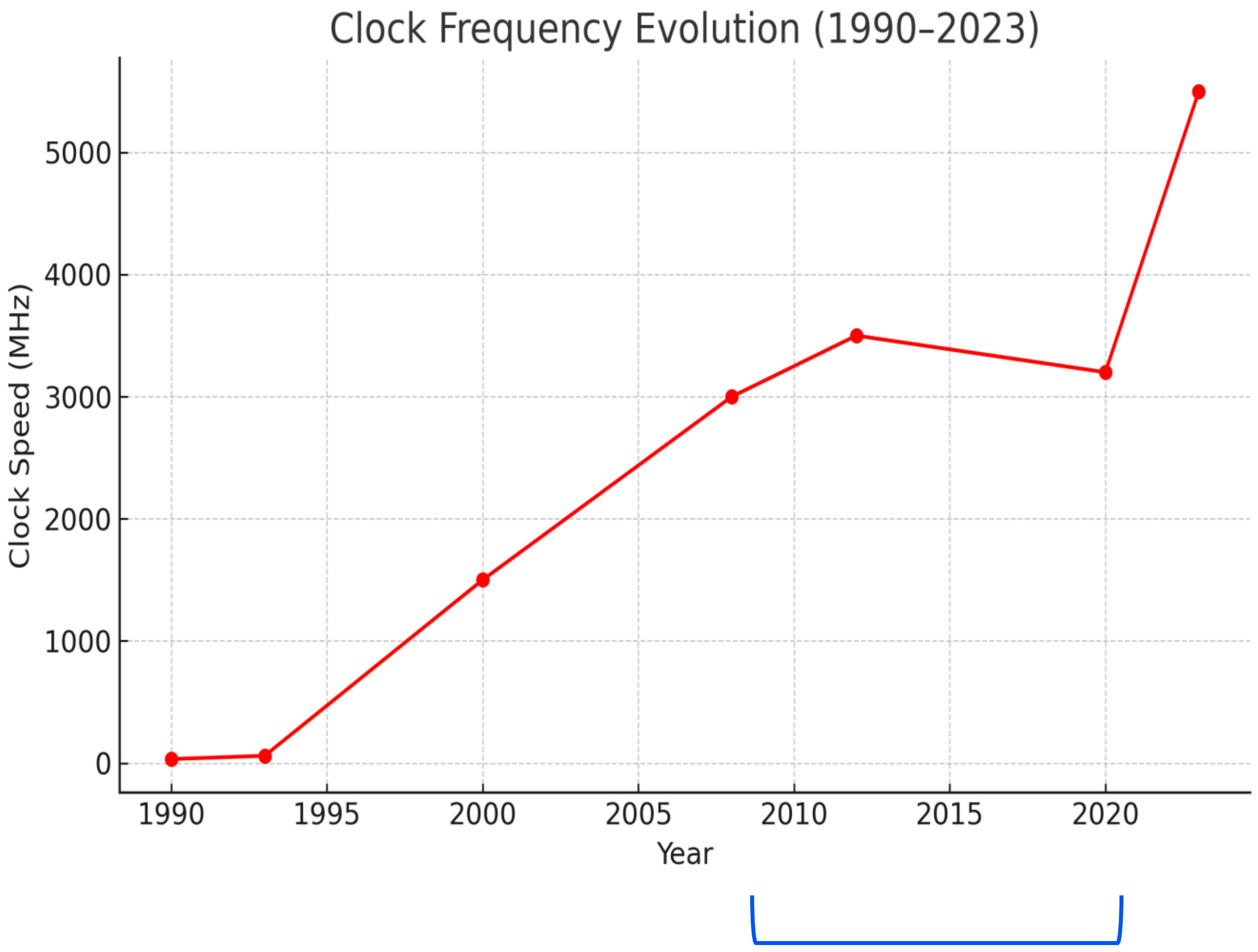
Shrinking nodes,
↑ power density,
↓ yield margins

Transistors Count per Chip (Flagship CPUs)



- **1993**: ~3 million (Intel Pentium, 800 nm)
- **2000:** ~42 million (Pentium 4, 180 nm)
- **2008:** ~731 million (Intel Nehalem, 45 nm)
- **2012:** ~1.4 billion (Ivy Bridge, 22 nm)
- **2020:** ~16 billion (Apple M1, 5 nm)
- **2023:** ~134 billion (Apple M2 Ultra, 5 nm)

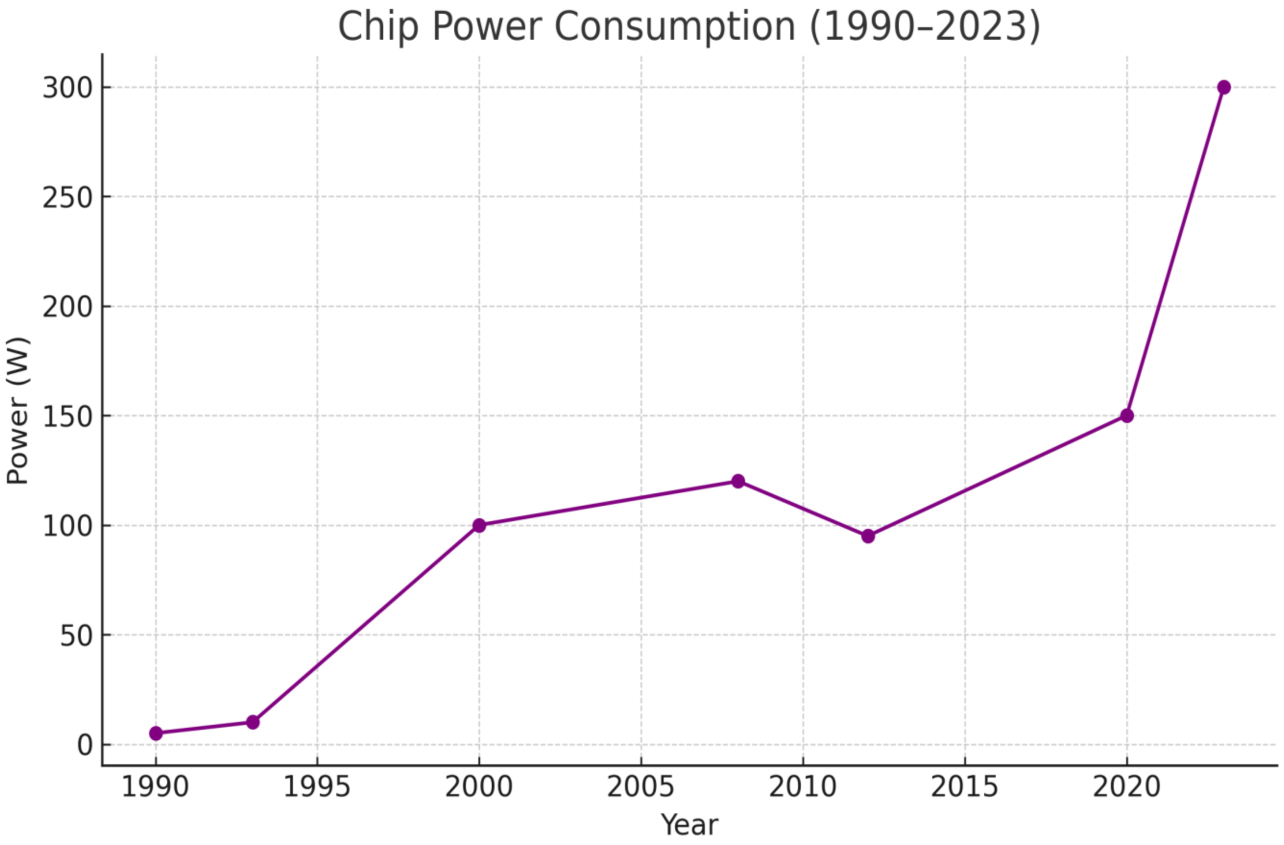
Clock Frequencies



multi-core dominance,
less emphasis on clock scaling

- **1990s:** 20–200 MHz (Intel 486 to Pentium Pro)
- **2000s:** 500 MHz → ~3.5 GHz (Pentium 4)
- **2010s:** ~2.5–4 GHz
- **2020s:** ~3–5.5 GHz (Intel Raptor Lake)

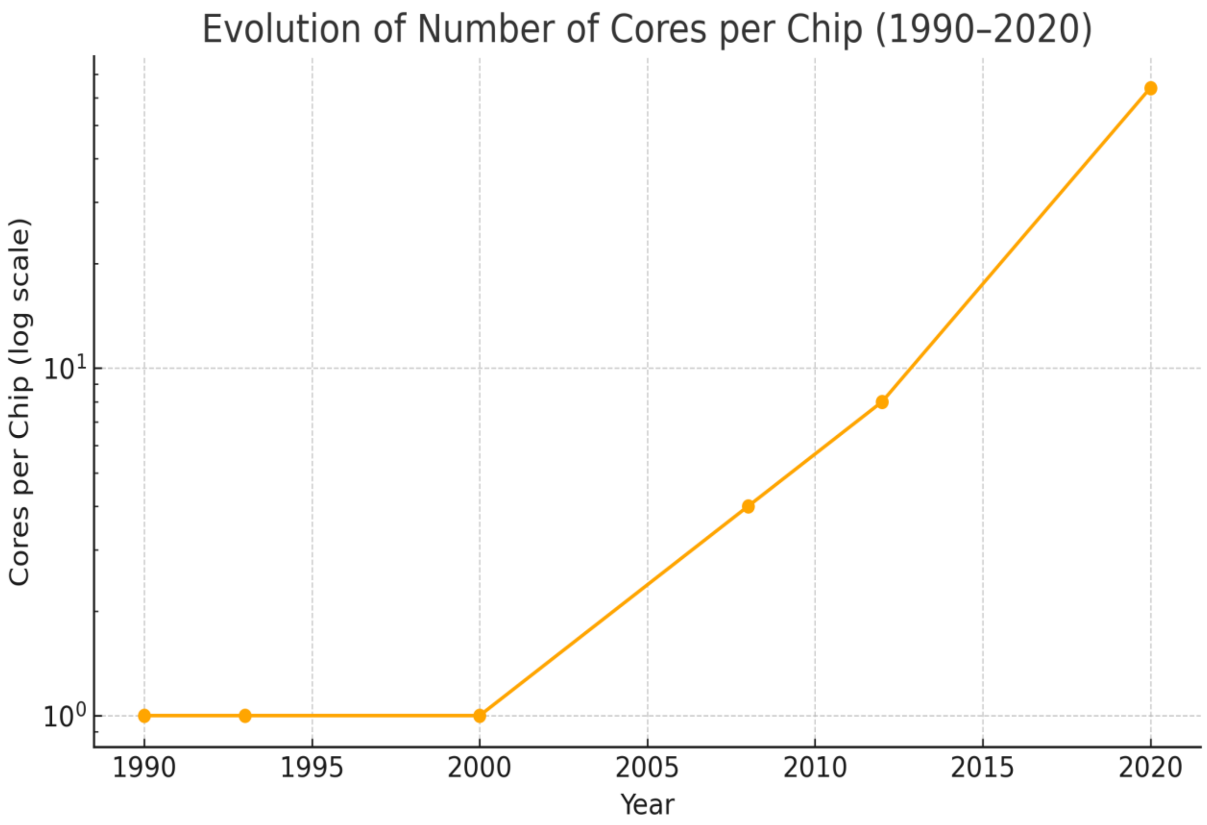
Power Consumption per Chip (High-Performance CPUs)



- **1990s:** ~1–10 W (Pentium)
- **2000s:** ~30–120 W (Pentium 4 “Prescott” peak ~115 W)
- **2010s:** ~65–140 W (Intel/AMD desktop CPUs)
- **2020s:** ~25–300 W (desktop CPUs/GPUs)

Limiting factors are power density and cooling capacity

Number of Cores per Chip (High-Performance CPUs)



- **1990s:** Single core era. Performance improvements came from shrinking process nodes, higher clock frequency, and architecture enhacements
- **Early 2000s:** power wall
- **Mid 2000s:** Transition to multicore (2, 4 cores)
- **2010s:** Scaling core counts in desktop CPUs (4-8 cores) and high-end servers (16-32 cores)
- **2020s:** Many-core era, consumer CPUs with 8-16 cores are common, server CPUs up to 128 cores, mobile SoCs big.LITTLE

Workloads

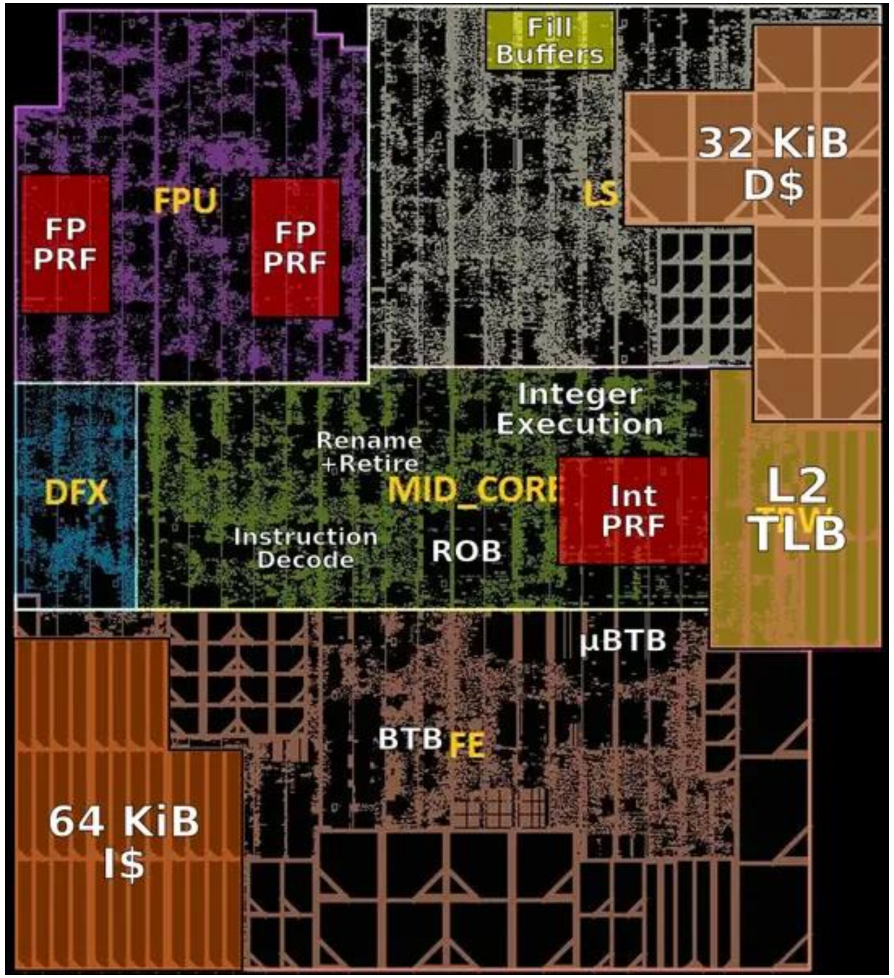
Year	Typical workloads	Characteristics	Hardware implications
1990s	Office apps (Word, Excel), early databases, 2D/3D gaming, dial-up internet, GUIs	Mostly **single-threaded**, modest memory, limited networking	Performance scaled with clock speed and single-core improvements
2000s	Multimedia (MP3, DVD), web browsing (Flash, JavaScript), advanced 3D games, enterprise databases,	More parallel tasks, i.e. video decoding, still desktop apps largely single-threaded	Birth of **multi-core CPUs** (2–4 cores) to handle concurrent server/workloads
2010s	Mobile apps, cloud services, virtualization, big data, social media, video streaming	Heavy **multi-threading** in cloud & analytics, demand for concurrency, GPUs for parallelism	**4–32 cores** common, GPUs rise as accelerators, low-power SoCs dominate mobile
2020s	AI/ML (deep learning, transformers), cloud-native microservices, gaming with ray tracing, VR/AR, exascale simulations	**Massively parallel workloads**, heterogeneous computing (CPU + GPU + NPU/TPU), data- and compute-intensive	**Many-core CPUs (64+ cores)**, GPUs with thousands of cores, **specialized AI accelerators**

Floating-Point Arithmetic in Big Cores

Core Design Key Metrics – PPA (Performance, Power, Area)

- Trade-off among performance, power and area
  - Performance: timing, throughput and latency
  - Power: energy per operation(efficiency) and total power
  - Area: silicon footprint, dictates cost
- More performance often means higher power and increases area
- Lowering power can impact maximum frequency and aggressive power savings may need more area
- Reducing area usually means fewer resources and lower performance
- Pick the sweet spot depending on the workloads
  - Mobile CPUs: power first, then area, with good performance
  - Server CPUS: performance first
  - AI accelerators and GPUs: performance per watt, large area for compute arrays

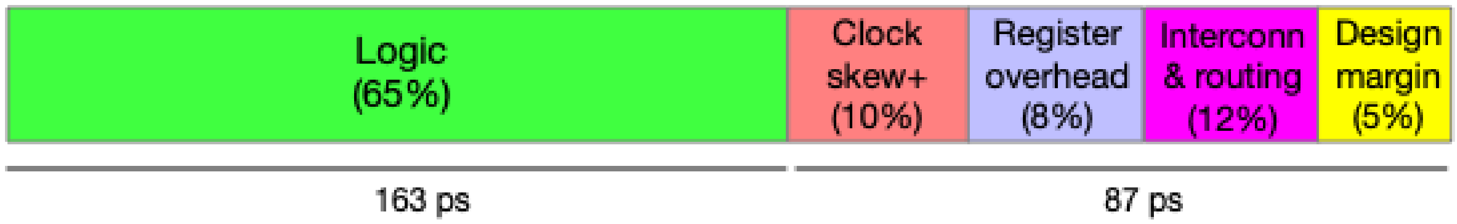
Samsung Exynos M1 (2016, 14nm, @2.6 GHz)



Core Design Key Metrics – PPA (Performance, Power, Area)

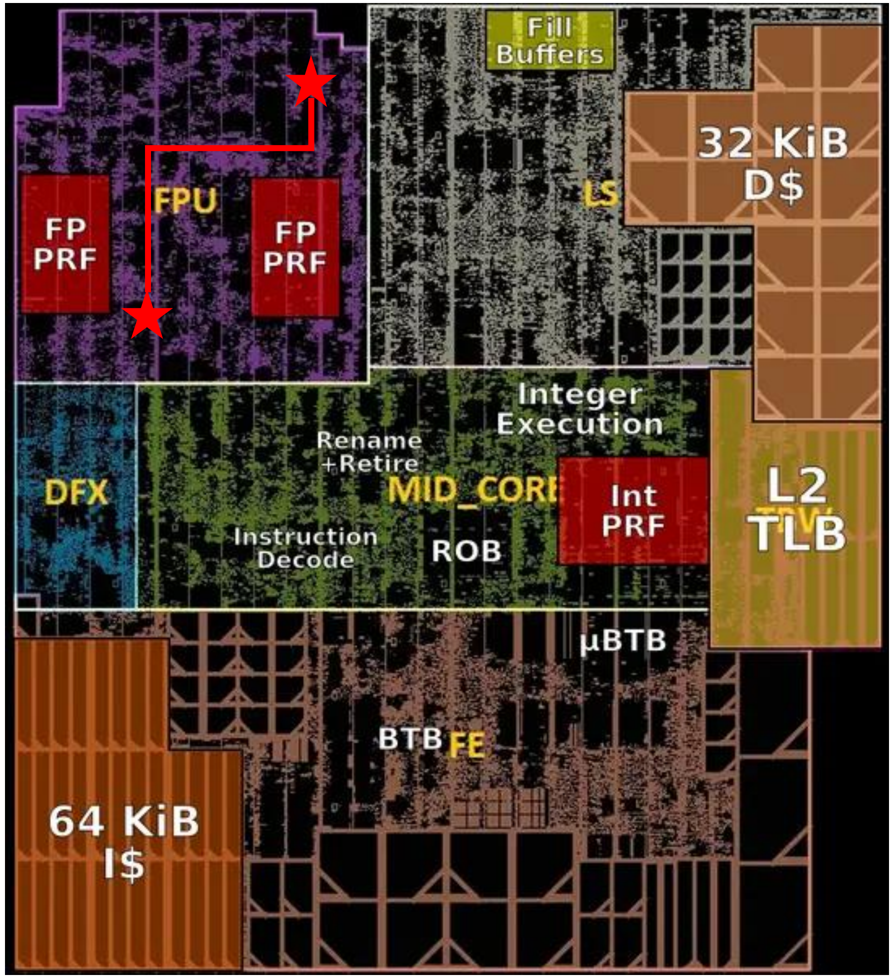
- **Performance - timing**
  - Clock frequency in high-performance cores is $\sim 4GHz$, then cycle period is $\sim 250ps$
  - Less than $250ps$ for logic -> clock skew, flop set-up and hold time, interconnect and routing

Cycle time breakdown example (250ps cycle)



  - Logic depth -> balanced paths
  - Routing -> **Forwarding**
- **Performance – throughput**
  - Pipelined/non-pipelined

Samsung Exynos M1 (2016, 14nm, @2.6 GHz)

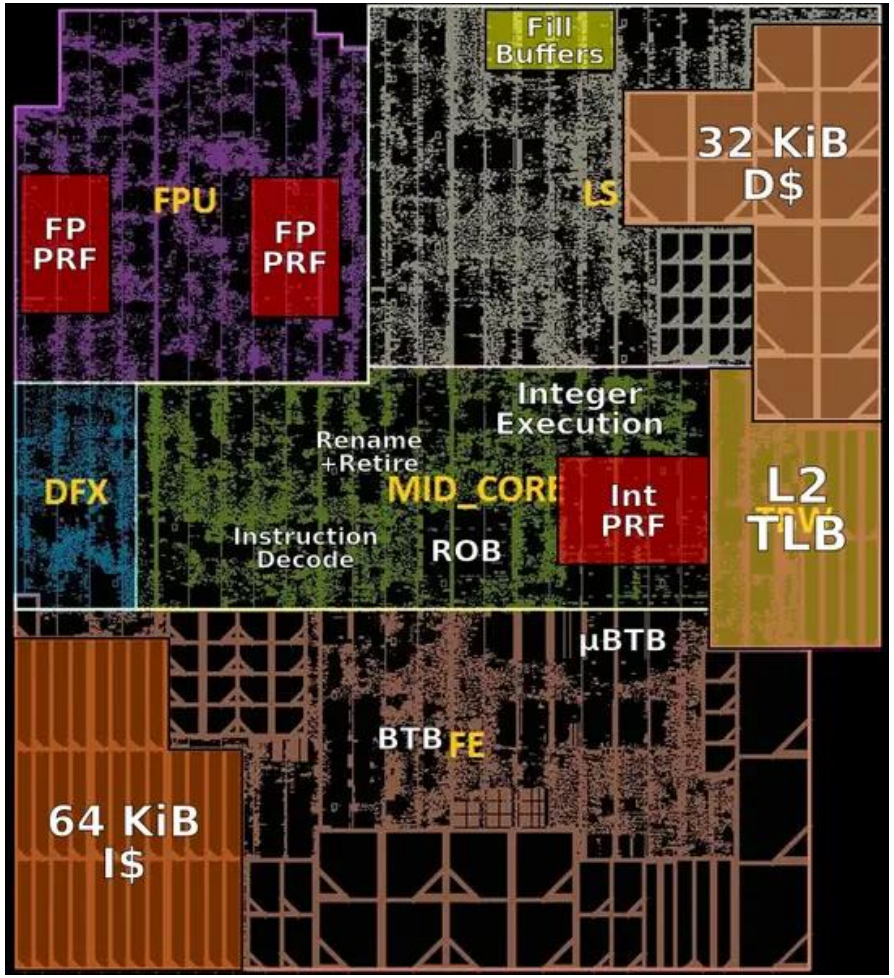


Core Design Key Metrics – PPA (Performance, Power, Area)

- **Power Breakdown**

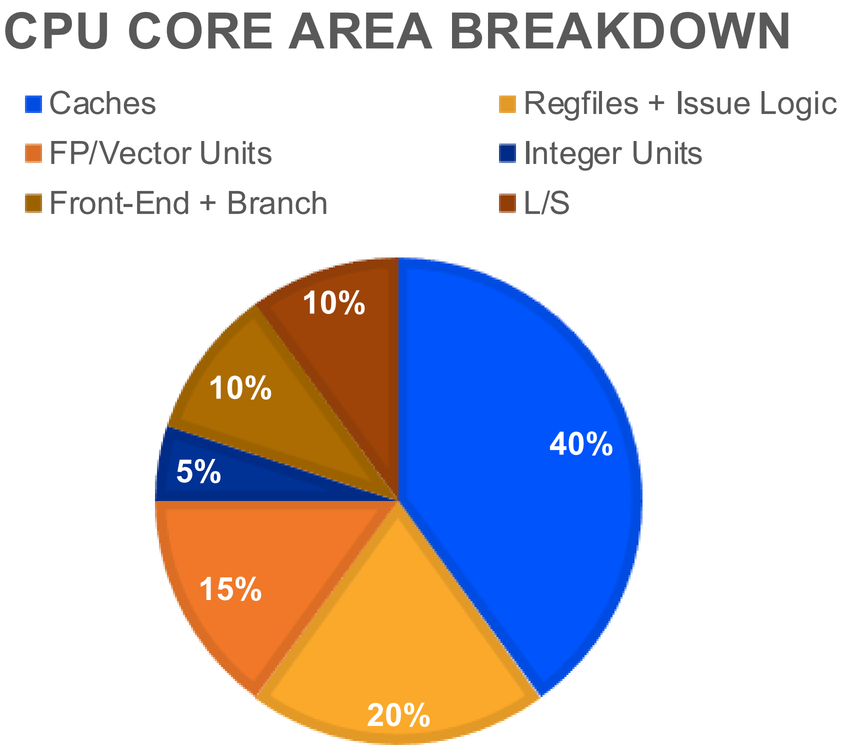
	High-efficiency CPU (Neoverse)	Modern AI (Hopper)	Notes
FP/vector/matrix	10 – 15%	30 – 45%	Compute dominates in AI
Integer	5 – 10%	5 – 10%	
L/S	5 – 15%	5 – 10%	In AI still many loads-stores (cannot be neglected)
L1 & L2 caches	25 – 40%	20 – 35%	Expensive due to many access
Interconnect, data movement	10 – 20%	10 – 20%	In AI less costly than in CPUs
Branch & Front-end	10 – 20%	5 – 15%	
Clock & misc	10 – 20%	5 – 15%	Multi-port register files and bypass are costly

Samsung Exynos M1 (2016, 14nm, @2.6 GHz)

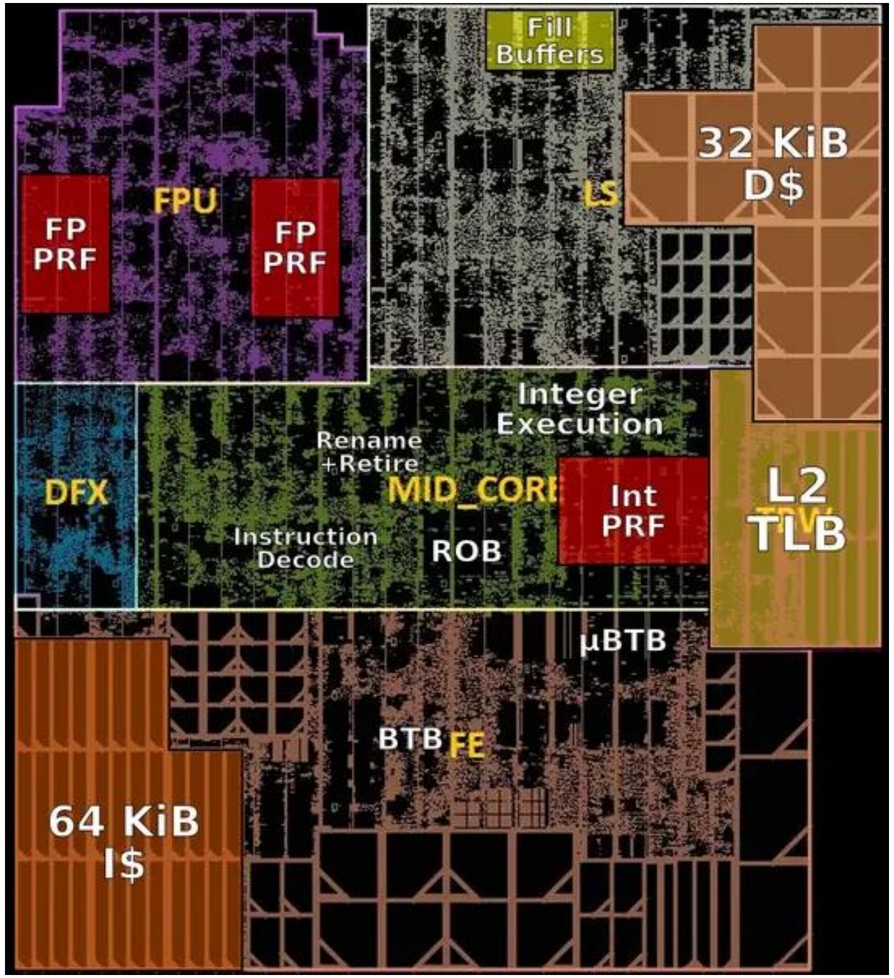


Core Design Key Metrics – PPA (Performance, Power, Area)

- **Area**



Samsung Exynos M1 (2016, 14nm, @2.6 GHz)



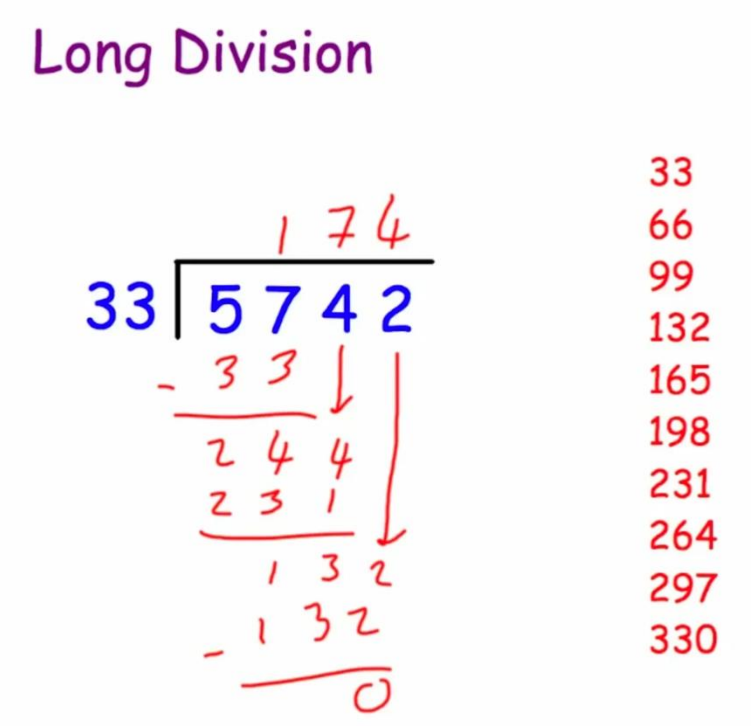
- GPUs and AI accelerators
  - Compute units have much larger share - $30 - 50\%$
  - On-chip SRAM comparable or larger area

What on FPUs?

- The goal is **more operations per second, with less power, and less silicon**
- Increase performance with wider datapath, deep pipelining, specialized units, fused operations (FMA), exploit parallelism within the FPU
- Reduce power with lower precision formats, clock gating, approximate computing
- Area is dominated by multipliers (quadratic growth with bit-width), vector units, register file
- Some trade-offs
  - High-throughput FPUs deliver more FLOPS but consumes more power
  - Wider datapath need more area
  - More area not always mean more power
- Workload driven sweet spots
  - Scientific computing: precision first (FP64), so area/power sacrificed for accuracy
  - Graphics: FP32 dominates, balanced between throughput and power
  - AI/ML: Reduced precision (FP16, BFLOAT16, FP8), massive gains in performance per Watt, accuracy suffers

Example: Floating-Point Division and Square Root

- Iterative algorithm
  - Newton-Raphson: quadratic convergence
  - Digit-recurrence: linear convergence
- Digit-recurrence
  - Iterative method similar to the *paper-and-pencil* method for division and square root
  - Result is produced *Most-significant digit first*
  - Each iteration
    - Obtain one digit of the result
      - Result digits based on the divisor and remainder
      - First remainder is the dividend
    - Update the partial result
      - Concatenation of the digit
    - Update of the remainder,
      - Using the actual digit, the divisor, and the actual remainder
  - The number of iterations depends on the precision



Digit-Recurrence Division and Square Root

- One quotient/root digit per iteration
- Redundant quotient/root
  - It is represented with n *radix-r digits*, being r a power of 2 $(r = 2^b)$

    $Q = q_{n-1} \times r^{n-1} + \cdots + q_1 \times r + q_0$

    $q_i \in \{-r/2, \ldots, -1, 0, +1, \ldots, +r/2\}$
- The larger the radix, the more complex the iteration
- Redundant quotient/root means several different representations
- Faster and simpler implementation

Example: r=4 and n=5, representation of 441

Q= $q_4 \times 4^4 + q_3 \times 4^3 + q_2 \times 4^2 + q_1 \times 4 + q_0$

$q_i = \{-2, -1, 0, +1, +2\}$

$441 = 2 \times 4^4 - 1 \times 4^3 + 0 \times 4^2 - 2 \times 4^1 + 1 \rightarrow (2, -1, 0, -2, 1)$

$441 = 2 \times 4^4 - 1 \times 4^3 - 1 \times 4^2 + 2 \times 4^1 + 1 \rightarrow (2, -1, -1, 2, 1)$

Radix (r)	Digit set* $(q_j = \{-r/2, \ldots, 0, \ldots, +r/2\})$	Bits per iteration $(b = \log_2 r)$
2	$\{-1, 0, +1\}$	1
4	$\{-2, -1, 0, +1, +2\}$	2
8	$\{-4, \ldots, -1, 0, +1, \ldots, +4\}$	3
16	$\{-8, \ldots, -1, 0, +1, \ldots, +8\}$	4
32	$\{-16, \ldots, -1, 0, +1, \ldots, +16\}$	5
64	$\{-32, \ldots, -1, 0, +1, \ldots, +32\}$	6
128	$\{-64, \ldots, -1, 0, +1, \ldots, +64\}$	7
256	$\{-128, \ldots, -1, 0, +1, \ldots, +128\}$	8
512	$\{-256, \ldots, -1, 0, +1, \ldots, +256\}$	9
1024	$\{-512, \ldots, -1, 0, +1, \ldots, +512\}$	10

(*) A radix-r digit set can be maximally redudant, $q_j = \{-(r-1), \ldots, +(r-1)\}$

Digit-Recurrence Division and Square Root

- Several iterations with,
  1. Digit selection
  2. *(Redundant)* Remainder update
  3. Partial result update - digit concatenation

DIVISION

$$q_{j+1} = SEL(\widehat{rem}[j], d)$$

$$rem[j+1] = r \times rem[j] - d \times q_{j+1}$$

$$Q[j+1] = Q[j] + q_{j+1} \times r^{-(j+1)}$$

SQUARE ROOT

$$s_{j+1} = SEL(\widehat{rem}[j], S[j])$$

$$rem[j+1] = r \times rem[j] - s_{j+1} \times (2 \times S[j] + s_{j+1} \times r^{-(j+1)})$$

$$\mathrm{S}[j+1] = S[j] + s_{j+1} \times r^{-(j+1)}$$

Radix ($\boldsymbol{r}$)	**Digit set*** ($\boldsymbol{q_j = \{-r/2, \dots, 0, \dots, +r/2\}}$)	**Bits per iteration** ($\boldsymbol{b = \log_2 r}$)
2	$\{-1, 0, +1\}$	1
4	$\{-2, -1, 0, +1, +2\}$	2
8	$\{-4, \dots, -1, 0, +1, \dots, +4\}$	3
16	$\{-8, \dots, -1, 0, +1, \dots, +8\}$	4
32	$\{-16, \dots, -1, 0, +1, \dots, +16\}$	5
64	$\{-32, \dots, -1, 0, +1, \dots, +32\}$	6
128	$\{-64, \dots, -1, 0, +1, \dots, +64\}$	7
256	$\{-128, \dots, -1, 0, +1, \dots, +128\}$	8
512	$\{-256, \dots, -1, 0, +1, \dots, +256\}$	9
1024	$\{-512, \dots, -1, 0, +1, \dots, +512\}$	10

(*) A radix-r digit set can be maximally redudant, $q_j = \{-(r-1), \dots, +(r-1)\}$

Fast Floating-Point Division and Square-Root in Big Arm Cores

- Choose the radix -> Latency
  - The number of iterations depends on radix (r) and number of bits of the final quotient/root (m)
  - Number of iterations is $t = [m/\log_2 r]$

Radix (r)	**Digit set** $[-r/2, +r/2]$	**Bits/iteration** $(\log_2 r)$	**FP64**	**FP32**	**FP16**
2	$[-1, +1]$	1	54	25	12
4	$[-2, +2]$	2	27	13	6
8	$[-4, +4]$	3	18	9	4
16	$[-8, +8]$	4	14	7	3
32	$[-16, +16]$	5	11	5	3
64	$[-32, +32]$	6	9	5	2
1024	$[-512, +512]$	10	6	3	2

- Best radix: Trade-off between number of iterations (latency) and iteration complexity (area, timing)
  - Low radices are simpler but slower

Fast Floating-Point Division and Square-Root in Big Arm Cores

- Effective radix: A large radix can be obtained by overlapping several simpler low-radix iterations in the same cycle
- Combine the reduced latency of a large radix and the hardware simplicity of a small radix

Arm Cores	FP div	FP sqrt
--	**Radix 16** four radix-2 it/cycle	**Radix 4** two radix-2 it/cycle
--	**Radix 64** three radix-4 it/cycle	**Radix 16** two radix-4 it/cycle
--	**Radix 64** two radix-8 it/cycle	**Radix 16** two radix-4 it/cycle
--	**Radix 64** two radix-8 it/cycle	**Radix 64** two radix-8 it/cycle

- Core in the last row: Common iteration for division and square root

Fast Floating-Point Division and Square-Root in Big Arm Cores

Arm CORE	RADIX		FP DIVISION LATENCY			FP SQUARE ROOT LATENCY			
	FP DIV	FP SQRT	FP64	FP32	FP16	FP64	FP32	FP16	
--	**16** (4xR2)	**4** (2xR2)	17-19	9-11	5-7	29-30	15-16	6-7	Non-pipelined
--	**64** (3xR4)	**16** (2xR4)	12-14	6-9	5-7	15-16	8-9	5-6	
--	**64** (2xR8)	**16** (2xR4)	11-14	6-9	5-7	15-16	8-9	5-6	
--	**64** (2xR8)	**64** (2xR8)	12	7	5	12	7	5	Pipelined

- Non-pipelined cores have a variable number of cycles in pre- and post-processing to deal with denormal inputs and output

Fast Floating-Point Division and Square-Root in Big Arm Cores

		Pipelined div/sqrt throughput	Iterative div/sqrt throughput (2 x fdivsqrt64 and 2 x fdivsqrt32)
Throughput	FP64 div	1	(2/10, 2/13)
	FP32 div	1	(4/5, 4/8)
	FP16 div	1	(1, 4/6)
	FP64 sqrt	1	(2/14, 2/15)
	FP32 sqrt	1	(4/7, 4/8)
	FP16 sqrt	1	(1, 4/5)
Area		7,700 sq.um	9,200 sq.um
Latency	FP64 div, sqrt	12	11-14 (div), 15-16 (sqrt)
	FP32 div, sqrt	7	6-9 (div), 8-9 (sqrt)
	FP16 div, sqrt	5	5-7 (div), 5-6 (sqrt)

Future (and Present) of Floating-Point Formats and Operations

Architecture and Microarchitecture

Future (and present) of FP support: Architecture and Microarchitecture

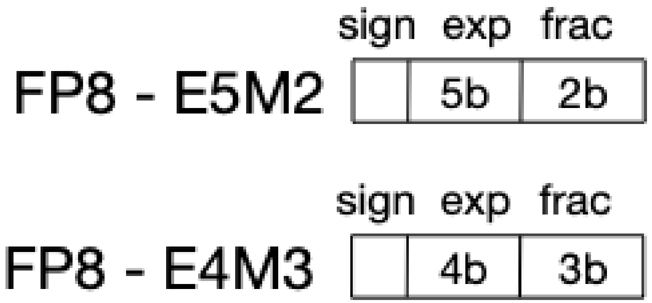
- Architectural and microarchitectural trends are mainly driven by AI/GenAI workload
- In AI Deployment: Model compression & quantization (8-bit, 4-bit)
- Architecture and microarchitecture support
- New formats – small floats
  - BF16, FP8, below FP8 (FP6, FP4), Microscaling, (also small integers formats, i.e., INT8, INT4)
- Mixed precision: new instructions with small floats accumulating to larger FP formats (FP32)
  - Inner product (Dot product), Small matrix multiplications, i.e. (4x2)x(2x4), Standard FMA
- Matrix extension
  - New instructions and hardware
- Determining what is good enough is not an exact science
  - Precision, formats, rounding modes, scaling, …
  - Training and inference use a mix of formats
  - Different layers may use different formats
- The AI application must be within the expected error bounds
  - Designer decides the number of internal fractional bits and rounding modes

Formats for Small Floats

	sign	exponent	fraction	
FP64		11 bits	52 bits	64 bits
FP32		8 bits	23 bits	32 bits
BF16		8 bits	7 bits	16 bits
FP16		5 bits	10 bits	16 bits
	sign	**exp**	**frac**	
FP8 - E5M2		5b	2b	8 bits
FP8 - E4M3		4b	3b	8 bits
FP6 - E3M2		3b	2b	6 bits
FP6 - E2M3		2b	3b	6 bits
FP4		2b	1b	4 bits

FP8 Formats – Example OFP8

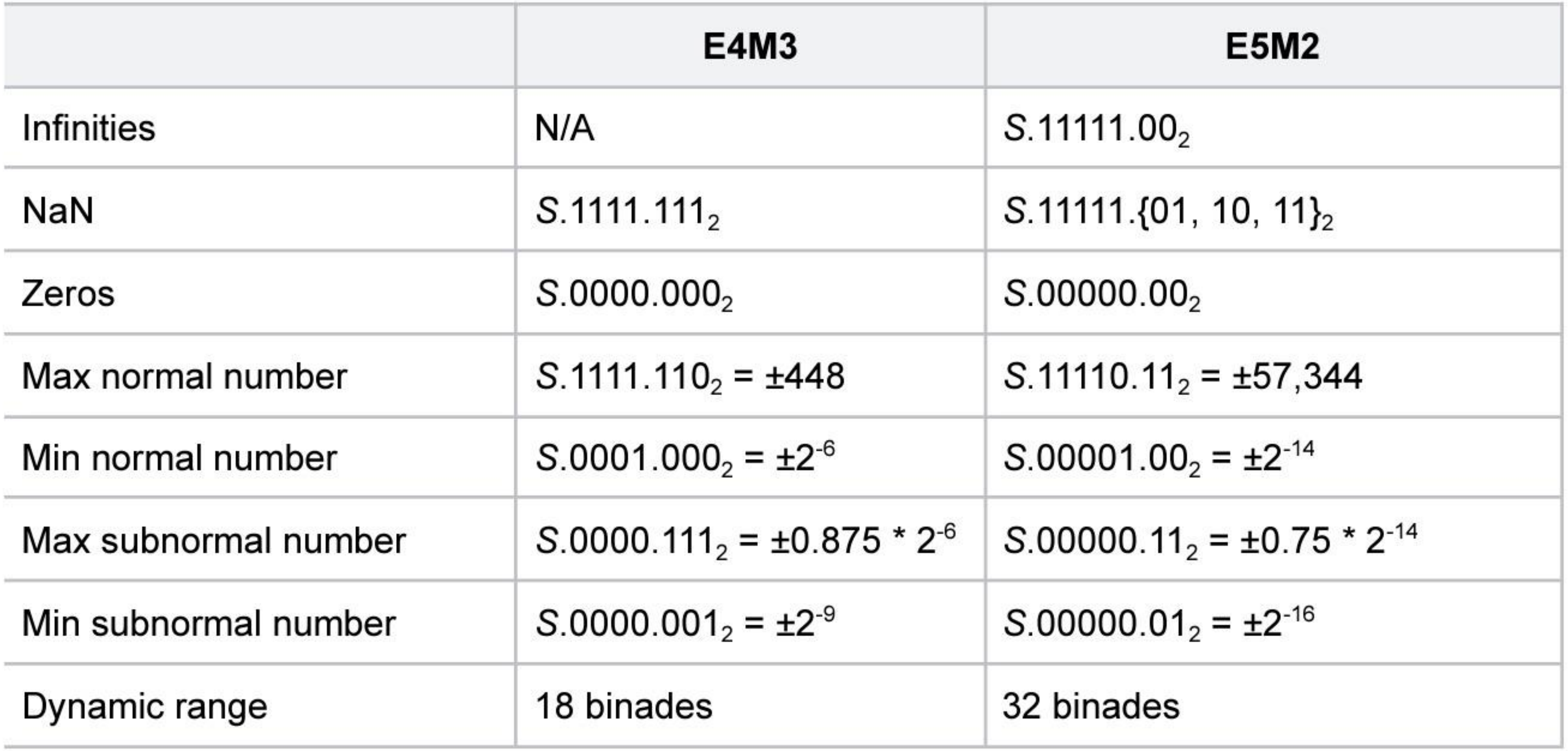
- There are several alternative FP8 formats being used currently in industry
- Example: OCP 8-bit FP specification (OFP8), 2023
  - Nvidia, AMD, Arm, Meta, Google, Intel
- Define two formats



	E4M3	E5M2
Exponent bias	7	15
emax (unbiased)	8	15
emin (unbiased)	-6	-14

- E5M2: Represents infinities and NaNs (3 patterns)
- E4M3: Does not represent infinities and uses one NAN pattern
  - Increase $emax$ and the dynamic range by one binade

FP8 Formats – Example OFP8



	E4M3	E5M2
Infinities	N/A	$S.11111.00_2$
NaN	$S.1111.111_2$	$S.11111.\{01, 10, 11\}_2$
Zeros	$S.0000.000_2$	$S.00000.00_2$
Max normal number	$S.1111.110_2 = \pm448$	$S.11110.11_2 = \pm57{,}344$
Min normal number	$S.0001.000_2 = \pm2^{-6}$	$S.00001.00_2 = \pm2^{-14}$
Max subnormal number	$S.0000.111_2 = \pm0.875 * 2^{-6}$	$S.00000.11_2 = \pm0.75 * 2^{-14}$
Min subnormal number	$S.0000.001_2 = \pm2^{-9}$	$S.00000.01_2 = \pm2^{-16}$
Dynamic range	18 binades	32 binades

Binary FP Formats for Machine Learning (IEEE P3109 Working Group)

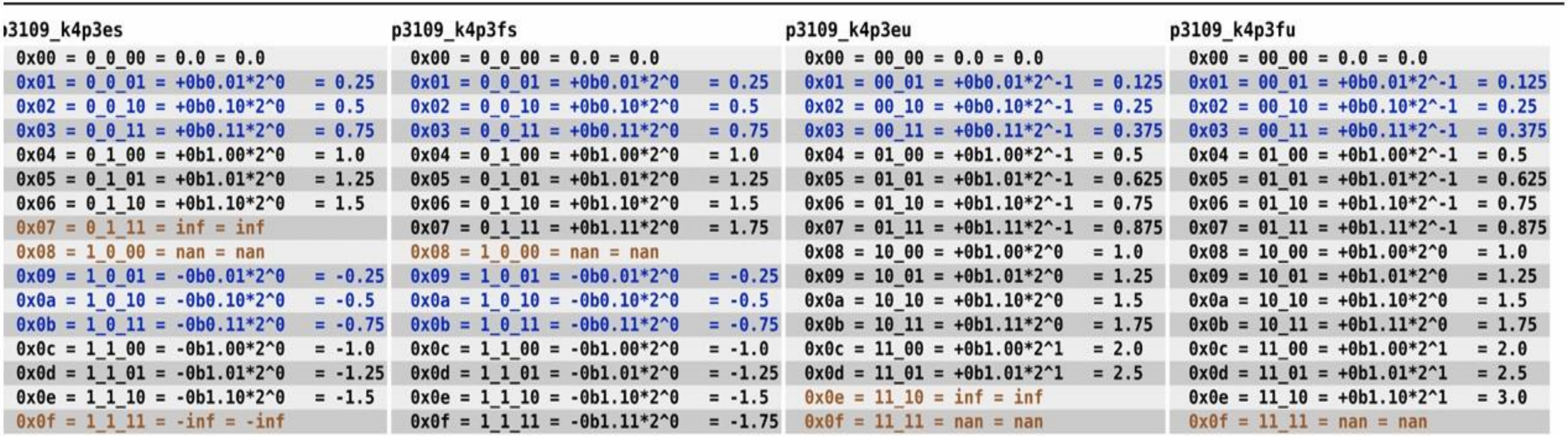
- Binary arithmetic and data format for machine learning-optimized domains
- Aligned with IEEE Std 754-2019 for Floating-Point Arithmetic
  - Biased exponent, subnormals, infinities, NaN
- Define several small floats formats (less than 16 bits)
  - Different number of exponent and fraction bits
  - Signed/unsigned
  - Infinities/no infinities
- Significant differences to the other IEEE FP standard:
  - Only one zero representation $(+0)$
  - Only one NaN (it is encoded at where Std 754 encodes $-0)$
- Why only exactly one zero and one NaN?
  - In machine learning, exceptions are difficult or expensive to deal with: NaN is allowed to propagate
  - The use of multiple NaN is not widely used and would reduce the limited encoding space
  - Negative 0 was included in IEEE Std 754 for consistent implementation of atan2 function and the complex trigonometric functions. These function are rare in AI

Binary FP Formats for Machine Learning (IEEE P3109 Working Group)

- The new standard describes a set of formats

$$binary\langle n\rangle p\langle x\rangle\langle\sigma\rangle\langle\delta\rangle$$

  - n total number of bits
  - x number of precision bits (including hidden bit)
  - $\sigma \in \{s, u\}$ (signed o unsigned) → exponent bias is 2^{n-x-1} or 2^{n-x} respectively
  - $\delta \in \{e, f\}$ (extended, finite)



```
3109_k4p3es
0x00 = 0_0_00 = 0.0 = 0.0
0x01 = 0_0_01 = +0b0.01*2^0    = 0.25
0x02 = 0_0_10 = +0b0.10*2^0    = 0.5
0x03 = 0_0_11 = +0b0.11*2^0    = 0.75
0x04 = 0_1_00 = +0b1.00*2^0    = 1.0
0x05 = 0_1_01 = +0b1.01*2^0    = 1.25
0x06 = 0_1_10 = +0b1.10*2^0    = 1.5
0x07 = 0_1_11 = inf = inf
0x08 = 1_0_00 = nan = nan
0x09 = 1_0_01 = -0b0.01*2^0    = -0.25
0x0a = 1_0_10 = -0b0.10*2^0    = -0.5
0x0b = 1_0_11 = -0b0.11*2^0    = -0.75
0x0c = 1_1_00 = -0b1.00*2^0    = -1.0
0x0d = 1_1_01 = -0b1.01*2^0    = -1.25
0x0e = 1_1_10 = -0b1.10*2^0    = -1.5
0x0f = 1_1_11 = -inf = -inf
```

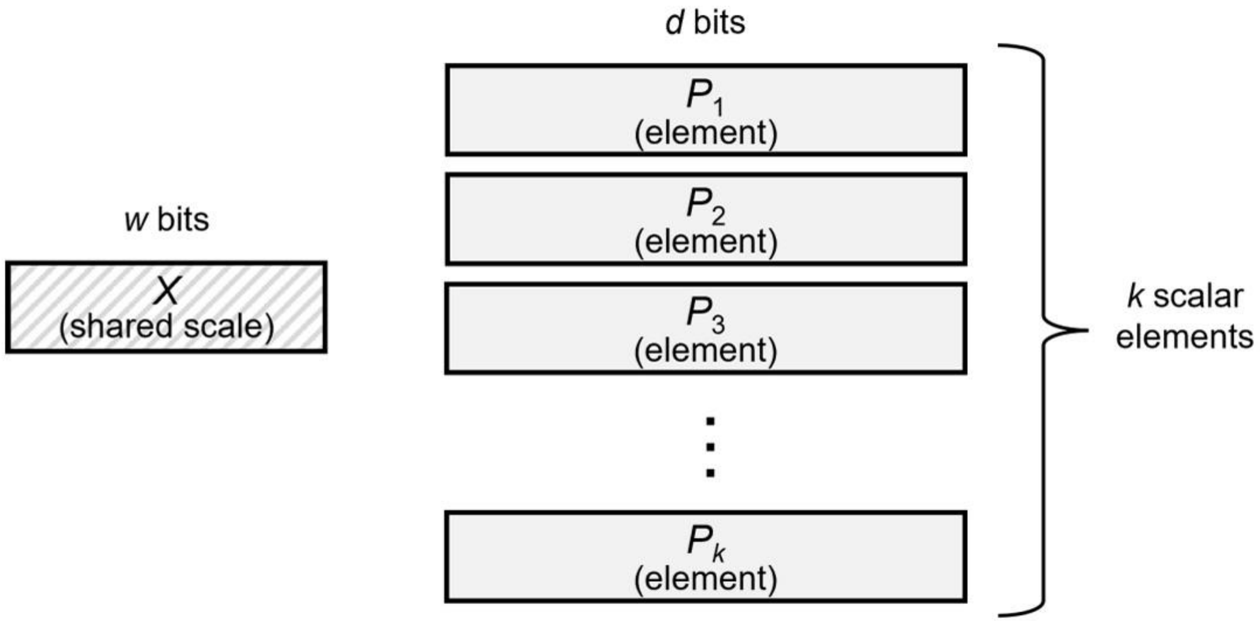
```
p3109_k4p3fs
0x00 = 0_0_00 = 0.0 = 0.0
0x01 = 0_0_01 = +0b0.01*2^0    = 0.25
0x02 = 0_0_10 = +0b0.10*2^0    = 0.5
0x03 = 0_0_11 = +0b0.11*2^0    = 0.75
0x04 = 0_1_00 = +0b1.00*2^0    = 1.0
0x05 = 0_1_01 = +0b1.01*2^0    = 1.25
0x06 = 0_1_10 = +0b1.10*2^0    = 1.5
0x07 = 0_1_11 = +0b1.11*2^0    = 1.75
0x08 = 1_0_00 = nan = nan
0x09 = 1_0_01 = -0b0.01*2^0    = -0.25
0x0a = 1_0_10 = -0b0.10*2^0    = -0.5
0x0b = 1_0_11 = -0b0.11*2^0    = -0.75
0x0c = 1_1_00 = -0b1.00*2^0    = -1.0
0x0d = 1_1_01 = -0b1.01*2^0    = -1.25
0x0e = 1_1_10 = -0b1.10*2^0    = -1.5
0x0f = 1_1_11 = -0b1.11*2^0    = -1.75
```

```
p3109_k4p3eu
0x00 = 00_00 = 0.0 = 0.0
0x01 = 00_01 = +0b0.01*2^-1  = 0.125
0x02 = 00_10 = +0b0.10*2^-1  = 0.25
0x03 = 00_11 = +0b0.11*2^-1  = 0.375
0x04 = 01_00 = +0b1.00*2^-1  = 0.5
0x05 = 01_01 = +0b1.01*2^-1  = 0.625
0x06 = 01_10 = +0b1.10*2^-1  = 0.75
0x07 = 01_11 = +0b1.11*2^-1  = 0.875
0x08 = 10_00 = +0b1.00*2^0   = 1.0
0x09 = 10_01 = +0b1.01*2^0   = 1.25
0x0a = 10_10 = +0b1.10*2^0   = 1.5
0x0b = 10_11 = +0b1.11*2^0   = 1.75
0x0c = 11_00 = +0b1.00*2^1   = 2.0
0x0d = 11_01 = +0b1.01*2^1   = 2.5
0x0e = 11_10 = inf = inf
0x0f = 11_11 = nan = nan
```

```
p3109_k4p3fu
0x00 = 00_00 = 0.0 = 0.0
0x01 = 00_01 = +0b0.01*2^-1  = 0.125
0x02 = 00_10 = +0b0.10*2^-1  = 0.25
0x03 = 00_11 = +0b0.11*2^-1  = 0.375
0x04 = 01_00 = +0b1.00*2^-1  = 0.5
0x05 = 01_01 = +0b1.01*2^-1  = 0.625
0x06 = 01_10 = +0b1.10*2^-1  = 0.75
0x07 = 01_11 = +0b1.11*2^-1  = 0.875
0x08 = 10_00 = +0b1.00*2^0   = 1.0
0x09 = 10_01 = +0b1.01*2^0   = 1.25
0x0a = 10_10 = +0b1.10*2^0   = 1.5
0x0b = 10_11 = +0b1.11*2^0   = 1.75
0x0c = 11_00 = +0b1.00*2^1   = 2.0
0x0d = 11_01 = +0b1.01*2^1   = 2.5
0x0e = 11_10 = +0b1.10*2^1   = 3.0
0x0f = 11_11 = nan = nan
```

Microscaling Floating-Point Formats (MX)

- MX format, three components
  - Scale (X)
  - Private elements (P_i)
  - Scaling block size (k)
- All k elements P_i have same data type (and bit-width)
- Scale factor X is shared across the k elements
- Data type of scale and elements may be different
  - w bits for scale and d bits for elements
  - Each block is encoded in $(w + kd)$ bits



The values $v_1, \dots, v_k$ *represented in a block are*

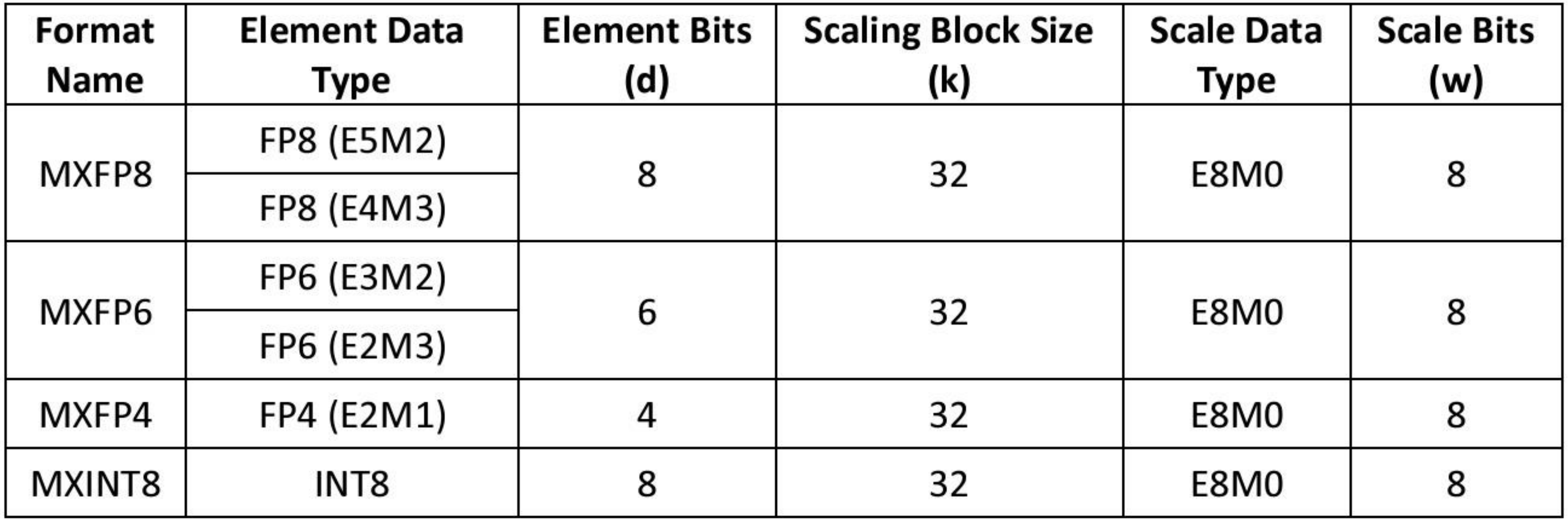
$if\ X = NaN, then\ v_i = NaN\ for\ 1 \leq i \leq k$

$if\ X \neq NaN:$

- $if\ P_i \in \{inf, NaN\}, then\ v_i = P_i$
- $if\ XP_i > V_{max}\ or\ XP_i < -V_{max}, then\ v_i\ is\ impl\ defined$
- otherwise, $v_i = XP_i$

Example: OCP MXFP8, MXFP6, MXFP4

- Different element types
- Scale is a power of 2



Format Name	Element Data Type	Element Bits (d)	Scaling Block Size (k)	Scale Data Type	Scale Bits (w)
MXFP8	FP8 (E5M2)	8	32	E8M0	8
	FP8 (E4M3)				
MXFP6	FP6 (E3M2)	6	32	E8M0	8
	FP6 (E2M3)				
MXFP4	FP4 (E2M1)	4	32	E8M0	8
MXINT8	INT8	8	32	E8M0	8

- Example: dot product with two MX format vectors of length k

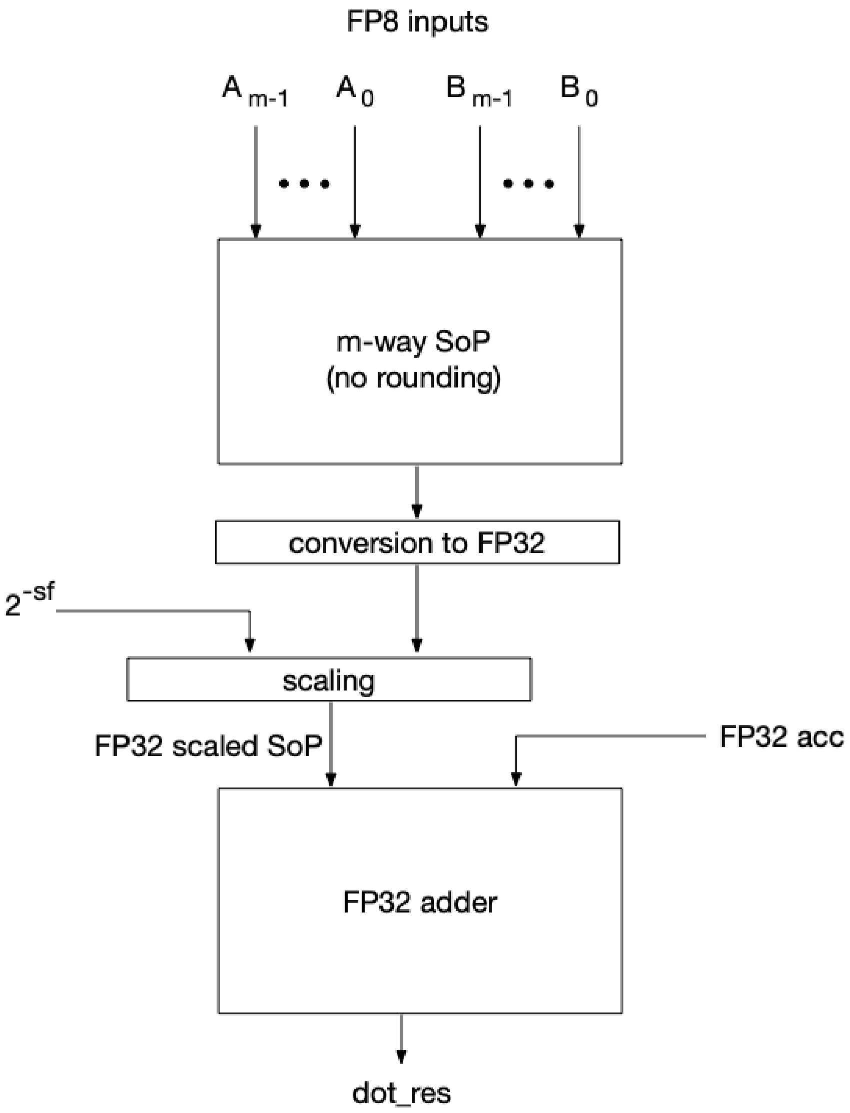
$$A: \left\{X^{(A)}, \left[P_i^{(A)}\right]_{i=1}^{k}\right\}, B: \left\{X^{(B)}, \left[P_i^{(B)}\right]_{i=1}^{k}\right\} \rightarrow \; C = Dot(A,B) = X^{(A)}X^{(B)}\sum_{i=1}^{k}\left(P_i^{(A)} \times P_i^{(B)}\right)$$

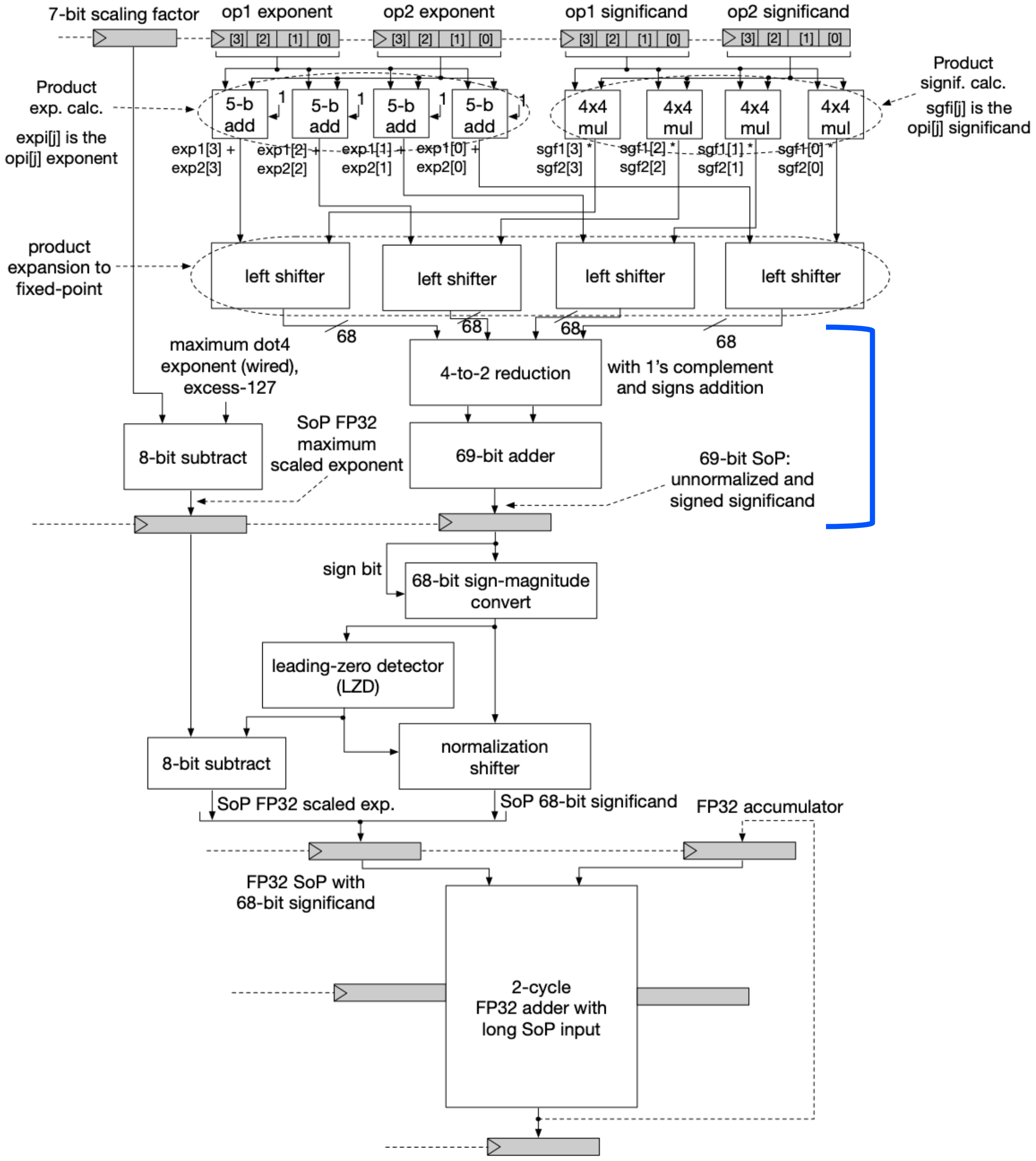
Mixed Precision: Small Floats Accumulating to Larger FP Formats

- Some examples of Armv9 instructions: dot product and small matrix-matrix multiplication
- BF16 and FP16 accumulating to FP32
  - 2-way dot product with accumulation
  - $(4 \times 2) \times (2 \times 4)$ accumulating to a (2×2) matrix → equivalent to four 4-way dot products with accumulation
- FP8 accumulating to FP32
  - 2-way and 4-way dot product with scaling and accumulation
  - FMA with scaling and accumulation
  - $(8 \times 2) \times (2 \times 8)$ accumulating to a (2×2) matrix → equivalent to eight 8-way dot products with scaling and accumulation
- FP8 accumulating to FP16
  - FMA with scaling and accumulation
  - $(4 \times 2) \times (2 \times 4)$ accumulating to a (2×2) matrix → equivalent to four 4-way dot products with scaling and accumulation

- Instructions added to the architecture and new units in the microarchitecture

Example: 4-Way FP8 Dot Product with Scaling and Accumulation





Matrix Extensions

- Special ISA extensions to add instructions (and execution units) optimized for matrix operations
- Many workloads (AI, ML, graphics, scientific HPC) boil down to **matrix multiply + accumulate**
  - Traditional vector extensions (AVX, NEON, SVE) can do multiply element wise; the regular structure of matrices it is not exploited
- Matrix extension: the CPU/accelerator can operate on **tiles** (blocks of data) in a single instruction
  - Exploits data reuse and reduces control overhead
- **TILE**: 2D register ($rows \times columns$) holding a small block of a matrix
  - Arranged as a matrix block, which the hardware can directly operate on
  - Tile registers are the building blocks for matrix instructions
  - The ISA exposes this tile as one “register” that can be loaded from memory, used in a multiply, and stored back to memory
  - A matrix instruction might take two input tiles and an accumulator tile, and perform a number of multiply-adds in one shot
- **Benefits**
  - Higher Throughput: One instruction does hundreds of multiplies and adds
  - Energy Efficiency: Better data reuse → fewer memory accesses → lower energy/op
  - Domain-Specific: Perfect fit for AI

Architecture Matrix Extensions

Architecture	Extension	Key Concept	Precision Focus	Use Cases
ArmV9	SME/SME2	Tile-based ops + scalable vectors (SVE2) Unified vector and matrix ops	FP8, FP16, FP32, INT8	AI inference, DSP, HPC
x86	AMX	Tile registers + AVX-512	BF16, INT8	DL training & inference
RISC-V	RVV + RVM	Scalable vectors + matrix multiply (RVM still evolving)	Configurable	Edge AI, HPC

- Matrix hardware accelerators: Arm CME, Nvidia Tensor Cores, Google TPU, Apple Matrix Coprocessor (AMX)

Conclusions

Conclusions: Floating-Point in Transition

- Floating-point landscape is diversifying
  - For decades, scientific computing was dominated by **IEEE 754 FP32 and FP64**, valuing precision, reproducibility and portability
  - The AI revolution has introduced **a spectrum of low-precision formats** optimized for throughput and energy efficiency, and **mixed-precision accumulation** to maintain accuracy
  - Approximate arithmetic for PPA balance
  - **Energy-aware scheduling** across CPU/GPU/NPU
- Bridging Science and AI
  - Scientific computing and AI are converging, both rely on flexible FP arithmetic, just at different points of the precision–efficiency spectrum
- Looking Ahead
  - The future FP unit will be **heterogeneous, flexible, adaptative, and energy-optimized** and will combine **multiple floating-point formats** in the same architecture to balance performance, precision, and efficiency
  - Floating-point formats are becoming **workload-aware**: precision when needed, efficiency when possible

The floating-point model that powered science for decades is evolving — not disappearing. Scientific computation and AI acceleration share the same silicon foundation, unified by adaptative precision and scalable matrix architectures

Merci

Danke

Gracias

Grazie

谢谢

ありがとう

Asante

Thank You

감사합니다

धन्यवाद

Kiitos

شكرًا

ধন্যবাদ

תודה

ధన్యవాదములు

Köszönöm

Example: OCP MXFP8, MXFP6, MXFP4

FP6

	E2M3	E3M2
Exponent bias	1	3
Infinities	N/A	N/A
NaN	N/A	N/A
Zeros	S 00 000_2	S 000 00_2
Max normal	S 11 111_2 = ± 2^2 × 1.875 = ± 7.5	S 111 11_2 = ± 2^4 × 1.75 = ± 28.0
Min normal	S 01 000_2 = ± 2^0 × 1.0 = ± 1.0	S 001 00_2 = ± 2^{-2} × 1.0 = ± 0.25
Max subnorm	S 00 111_2 = ± 2^0 × 0.875 = ± 0.875	S 000 11_2 = ± 2^{-2} × 0.75 = ± 0.1875
Min subnorm	S 00 001_2 = ± 2^0 × 0.125 = ± 0.125	S 000 01_2 = ± 2^{-2} × 0.25 = ± 0.0625

SCALE

	E8M0
Exponent bias	127
Supported exponent range	-127 to 127
Infinities	N/A
NaN	11111111_2
Zeros	N/A

FP4

	E2M1
Exponent bias	1
Infinities	N/A
NaN	N/A
Zeros	S 00 0_2
Max normal	S 11 1_2 = ± 2^2 × 1.5 = ± 6.0
Min normal	S 01 0_2 = ± 2^0 × 1.0 = ± 1.0
Max subnorm	S 00 1_2 = ± 2^0 × 0.5 = ± 0.5
Min subnorm	S 00 1_2 = ± 2^0 × 0.5 = ± 0.5