OPTIMIZING PERFORMANCE FOR DSP ALGORITHMS: BRIDGING THEORY AND PRACTICE ON MODERN PLATFORMS

Stefano D'Angelo
Orastron srl, Sessa Cilento, Italy

Copenhagen, Denmark
Aalborg University
4th September 2023

WHY SHOULD I CARE?

- Papers sometimes indicate measured execution time or operations count
- You'll hopefully learn those metrics can be of limited significance in the real world
- Perhaps you'll improve your algorithm design skills

COMMODORE 64

Bill Bertram, CC BY-SA 2.5

- MOS 6510 CPU
- 64 kB RAM (shared/mmaped)
- VIC-II gfx (320x200, 16 colors)
- SID synth-on-chip (3 osc, filter, ADSR, ringmod)

MOS 6510 CPU

- 8-bit, 16-bit address bus, 1 MHz
- Registers:
  - 1x 8-bit accumulator
  - 2x 8-bit index
  - 1x 8-bit stack pointer
  - 1x 16-bit program counter
  - 7 status flag bits
- ISA has 54 instructions
- Math instructions: ADC and SBC

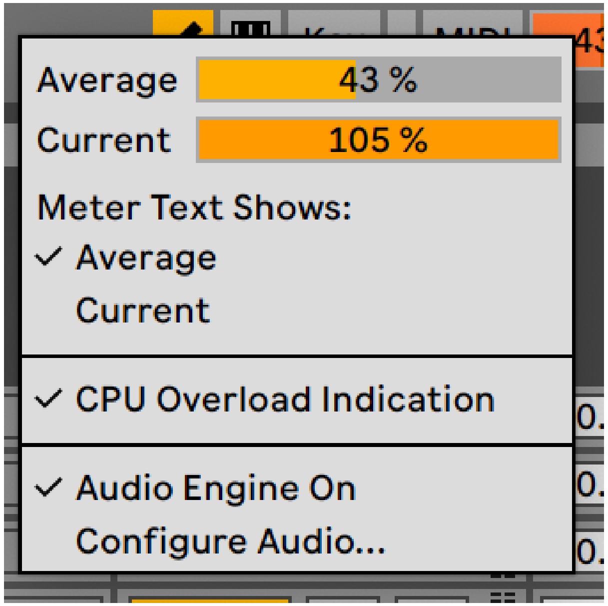
MUL ALGORITHMS

Source: https://codebase64.org/doku.php?id=base:multiplication_and_division

- A (8-bit) × B (8-bit) → C (16-bit)
- #1: Adding in a loop
  → up to 255 (×2) sums + loop logic
- #2: Bit-shifting
  → up to 8× left shifts, 16 bit sums, etc. + loop logic
- #3: Lookup
  $$ab = ((a+b)/2)^2 - ((a-b)/2)^2$$
  512 lookup entries, 16-bits each → 1 kB

25 Commodore c64 Games still great to play in 2020.

WHAT WENT WRONG?

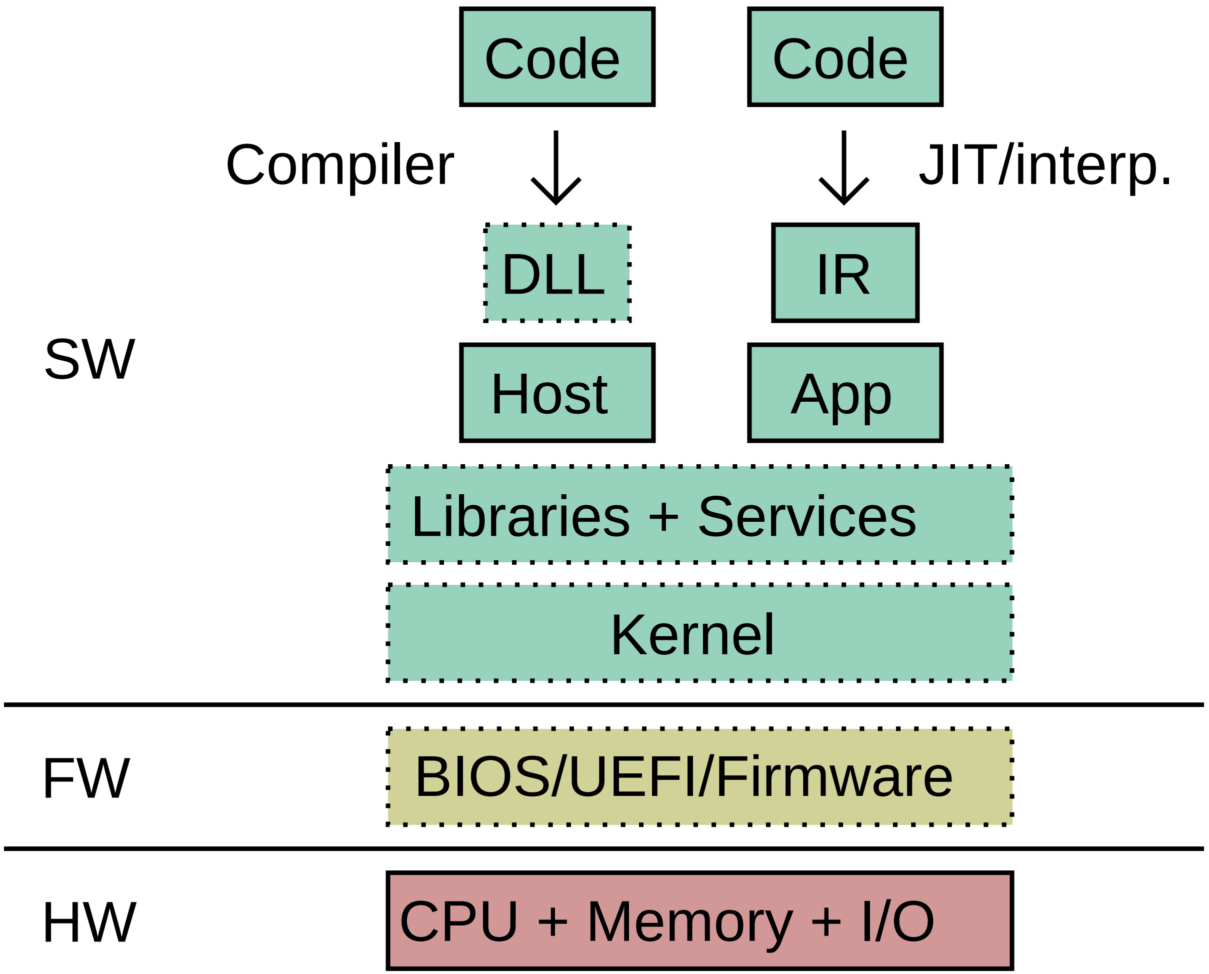


SR	Prev	Prev w/ gc	New	New w/ gc
96 kHz	29.87%	31.99%	**11.02%**	14.80%
192 kHz	45.69%	53.11%	**20.89%**	29.29%
384 kHz	94.21%	107.71%	**42.80%**	49.72%

TABLE IV: Average CPU usage detected by processing a set of input signals at different sample rates, after previous oversampling, and at different input gain levels.

Spoiler: not a fair comparison

SIMPLIFIED MODERN STACK



OPTIMIZATION PARADOX

Optimization is always target-dependent

but often target is at least partially unknown.

I'm hopefully giving you good general advice

and a feel of how it's practically done

and also some theoretical context.

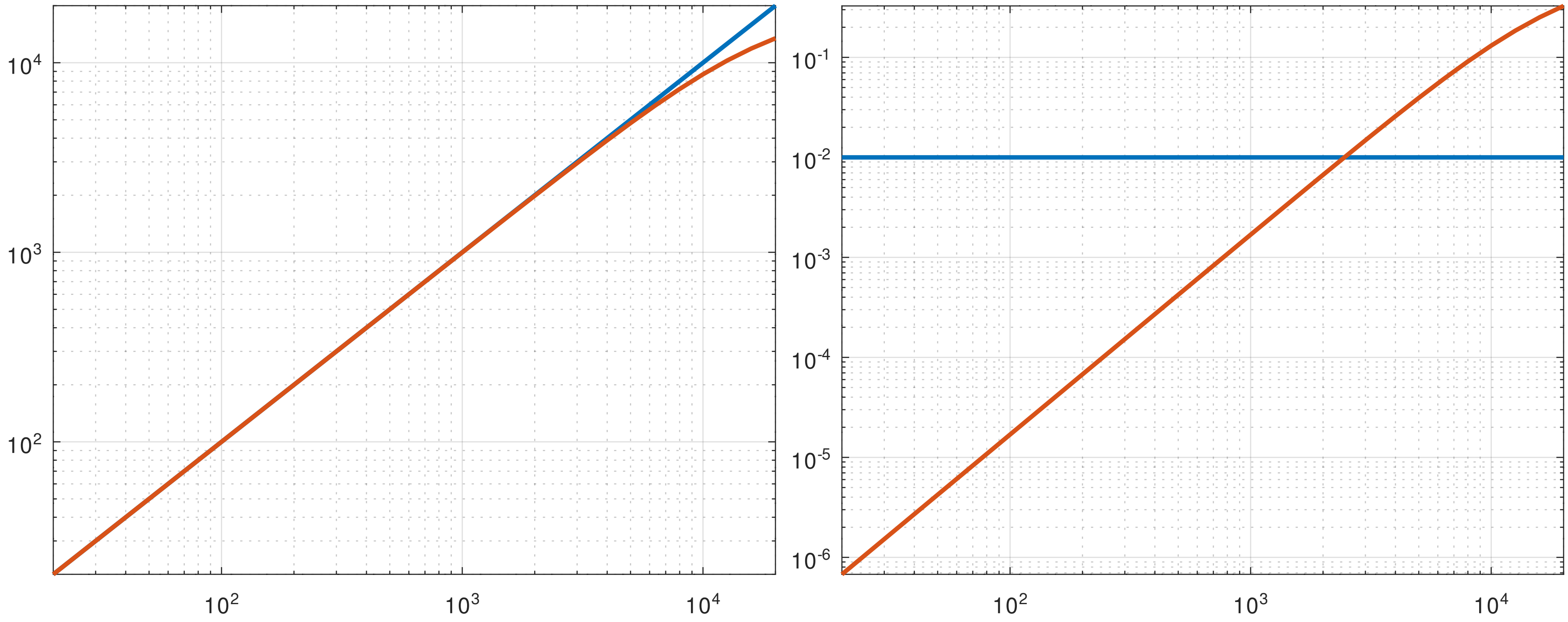
Anyway, always do your homework!

HOW TO OPTIMIZE

1. Understand application requirements
2. Use appropriate technology
3. (Re)structure for effectiveness and efficiency
   i.e., choose the right algorithms and use them smartly

4. Avoid recomputing
5. Improve memory usage
6. Exploit HW and data types
7. Approximate
8. Vectorize
9. Parallelize

FREQUENCY WARPING



$f_{\mathrm{s}} = 44100$ Hz, bilinear w/o prewarping

Relative error < 1% up to about 2.5 kHz

REAL-TIME AUDIO CONTRAINTS

- $O(N)$ algorithm, N number of I/O samples
- Never block, including `malloc()`, `rand()`, disk read, network, etc. (use a worker thread)
- You can't read the future (but you can add latency)

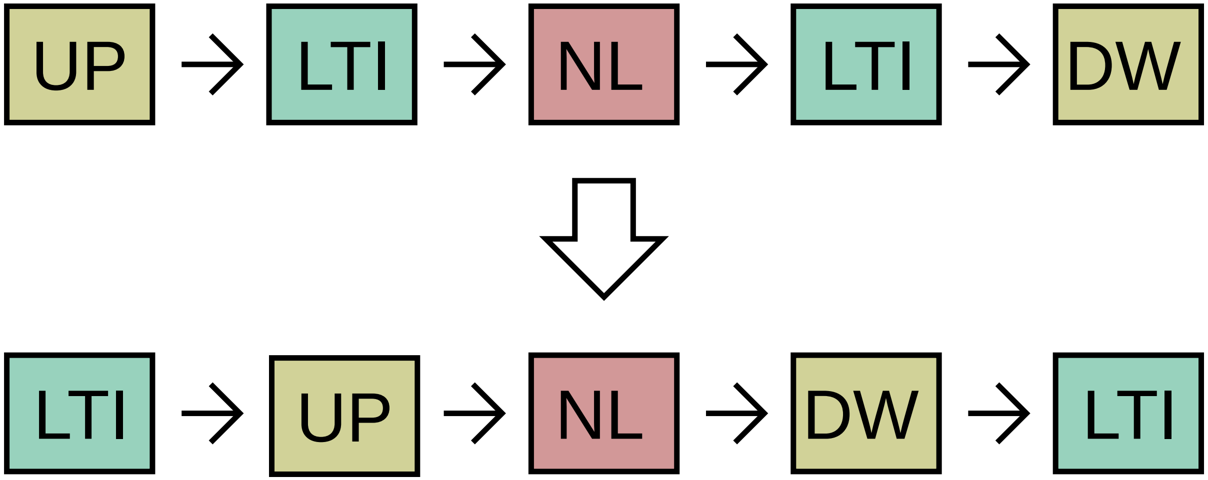
MATLAB, PYTHON, JAVA, JS, ETC.

- Quick coding, excellent for prototyping
- Automatic memory management vs determinism
- Normally need VMs/ecosystems which vary wildly
- Typically unsuitable for benchmarking
- Hard to satisfy real-time audio constraints

C, C++, ASSEMBLY, D, RUST, ETC.

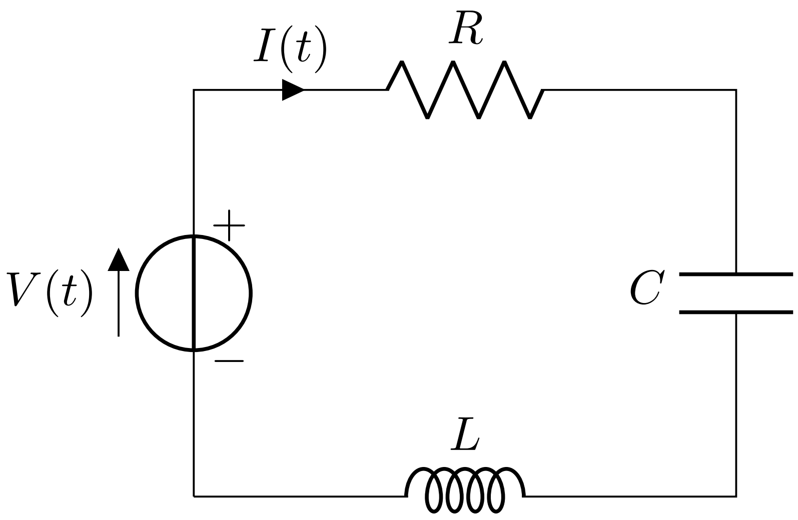
- Typically require more coding effort
- Can achieve "quasi-deterministic" behavior
- Compiler output is native code (w/ limited dependencies)
- Better for benchmarking
- Real-time audio constraints can be satisfied
- Actually used in real products

SMART RESTRUCTURING EXAMPLE



- Consider ADAA (lower OS factor) and IIR resampling (cheaper)
- Choose OS factor based on sample rate

AN RLC CIRCUIT MODEL



$$I[n] = B_0 V[n] + s1[n-1]$$
$$s1[n] = s2[n-1] - A_1 I[n]$$
$$s2[n] = -B_0 V[n] - A_2 I[n]$$
$$B_0 = \frac{2 f_\mathrm{s} C}{1 + 2 f_\mathrm{s} RC + 4 {f_\mathrm{s}}^2 LC}$$
$$A_1 = \frac{2 - 8 {f_\mathrm{s}}^2 LC}{1 + 2 f_\mathrm{s} RC + 4 {f_\mathrm{s}}^2 LC}$$
$$A_2 = \frac{1 - 2 f_\mathrm{s} RC + 4 {f_\mathrm{s}}^2 LC}{1 + 2 f_\mathrm{s} RC + 4 {f_\mathrm{s}}^2 LC}$$

STEP 1

$$
\begin{aligned}
I[n] &= B_0 V[n] + s1[n-1] \\
s1[n] &= s2[n-1] - A_1 I[n] \\
s2[n] &= -B_0 V[n] - A_2 I[n] \\
B_0 &= \frac{2 f_\mathrm{s} C}{1 + 2 f_\mathrm{s} RC + 4 {f_\mathrm{s}}^2 LC} \\
A_1 &= \frac{2 - 8 {f_\mathrm{s}}^2 LC}{1 + 2 f_\mathrm{s} RC + 4 {f_\mathrm{s}}^2 LC} \\
A_2 &= \frac{1 - 2 f_\mathrm{s} RC + 4 {f_\mathrm{s}}^2 LC}{1 + 2 f_\mathrm{s} RC + 4 {f_\mathrm{s}}^2 LC}
\end{aligned}
$$

→

$$
\begin{aligned}
x[n] &= B_0 V[n] \\
I[n] &= x[n] + s1[n-1] \\
s1[n] &= s2[n-1] - A_1 I[n] \\
s2[n] &= -x[n] - A_2 I[n] \\
k &= \frac{1}{1 + 2 f_\mathrm{s} RC + 4 {f_\mathrm{s}}^2 LC} \\
B_0 &= k \left(2 f_\mathrm{s} C\right) \\
A_1 &= k \left(2 - 8 {f_\mathrm{s}}^2 LC\right) \\
A_2 &= k \left(1 - 2 f_\mathrm{s} RC + 4 {f_\mathrm{s}}^2 LC\right)
\end{aligned}
$$

STEP 2

$$\begin{aligned}
x[n] &= B_0 V[n] \\
I[n] &= x[n] + s1[n-1] \\
s1[n] &= s2[n-1] - A_1 I[n] \\
s2[n] &= -x[n] - A_2 I[n] \\
k &= \frac{1}{1 + 2f_\mathrm{s}RC + 4{f_\mathrm{s}}^2 LC} \\
B_0 &= k\left(2f_\mathrm{s}C\right) \\
A_1 &= k\left(2 - 8{f_\mathrm{s}}^2 LC\right) \\
A_2 &= k\left(1 - 2f_\mathrm{s}RC + 4{f_\mathrm{s}}^2 LC\right)
\end{aligned}$$

$\rightarrow$

$$\begin{aligned}
x[n] &= B_0 V[n] \\
I[n] &= x[n] + s1[n-1] \\
s1[n] &= s2[n-1] - A_1 I[n] \\
s2[n] &= -x[n] - A_2 I[n] \\
k_2 &= 2f_\mathrm{s}C \\
k &= \frac{1}{1 + k_2 R + 2f_\mathrm{s}k_2 L} \\
B_0 &= k k_2 \\
A_1 &= k\left(2 - 4f_\mathrm{s}k_2 L\right) \\
A_2 &= k\left(1 - k_2 R + 2f_\mathrm{s}k_2 L\right)
\end{aligned}$$

STEP 3

$$\begin{aligned} x[n] &= B_0 V[n] \\ I[n] &= x[n] + s1[n-1] \\ s1[n] &= s2[n-1] - A_1 I[n] \\ s2[n] &= -x[n] - A_2 I[n] \\ k_2 &= 2 f_\mathrm{s} C \\ k &= \frac{1}{1 + k_2 R + 2 f_\mathrm{s} k_2 L} \\ B_0 &= k k_2 \\ A_1 &= k\left(2 - 4 f_\mathrm{s} k_2 L\right) \\ A_2 &= k\left(1 - k_2 R + 2 f_\mathrm{s} k_2 L\right) \end{aligned}$$

$\rightarrow$

$$\begin{aligned} x[n] &= B_0 V[n] \\ I[n] &= x[n] + s1[n-1] \\ s1[n] &= s2[n-1] - A_1 I[n] \\ s2[n] &= -x[n] - A_2 I[n] \\ k_2 &= 2 f_\mathrm{s} C \\ k_3 &= 2 f_\mathrm{s} k_2 L \\ k &= \frac{1}{1 + k_2 R + k_3} \\ B_0 &= k k_2 \\ A_1 &= k\left(2 - 2 k_3\right) \\ A_2 &= k\left(1 - k_2 R + k_3\right) \end{aligned}$$

STEP 4

$$
\begin{aligned}
x[n] &= B_0 V[n] \\
I[n] &= x[n] + s1[n-1] \\
s1[n] &= s2[n-1] - A_1 I[n] \\
s2[n] &= -x[n] - A_2 I[n] \\
k_2 &= 2 f_\mathrm{s} C \\
k_3 &= 2 f_\mathrm{s} k_2 L \\
k &= \frac{1}{1 + k_2 R + k_3} \\
B_0 &= k k_2 \\
A_1 &= k\left(2 - 2k_3\right) \\
A_2 &= k\left(1 - k_2 R + k_3\right)
\end{aligned}
$$

→

$$
\begin{aligned}
x[n] &= B_0 V[n] \\
I[n] &= x[n] + s1[n-1] \\
s1[n] &= s2[n-1] - A_1 I[n] \\
s2[n] &= -x[n] - A_2 I[n] \\
k_1 &= 2 f_\mathrm{s} \\
k_2 &= k_1 C \\
k_3 &= k_1 k_2 L \\
k_4 &= 1 + k_3 \\
k_5 &= k_2 R \\
k &= \frac{1}{k_4 + k_5} \\
B_0 &= k k_2 \\
A_1 &= k\left(2 - 2k_3\right) \\
A_2 &= k\left(k_4 - k_5\right)
\end{aligned}
$$

STEP 5

$$\begin{aligned}
x[n] &= B_0 V[n] \\
I[n] &= x[n] + s1[n-1] \\
s1[n] &= s2[n-1] - A_1 I[n] \\
s2[n] &= -x[n] - A_2 I[n] \\
k_1 &= 2f_\mathrm{s} \\
k_2 &= k_1 C \\
k_3 &= k_1 k_2 L \\
k_4 &= 1 + k_3 \\
k_5 &= k_2 R \\
k &= \frac{1}{k_4 + k_5} \\
B_0 &= k k_2 \\
A_1 &= k\left(2 - 2k_3\right) \\
A_2 &= k\left(k_4 - k_5\right)
\end{aligned}$$

→

$$\begin{aligned}
x[n] &= B_0 V[n] \\
I[n] &= x[n] + s1[n-1] \\
s1[n] &= s2[n-1] - A_1 I[n] \\
s2[n] &= \widehat{A_2} I[n] - x[n] \\
k_1 &= 2f_\mathrm{s} \\
k_2 &= k_1 C \\
k_3 &= k_1 k_2 L \\
k_4 &= 1 + k_3 \\
k_5 &= k_2 R \\
k &= \frac{1}{k_4 + k_5} \\
B_0 &= k k_2 \\
A_1 &= k\left(2 - 2k_3\right) \\
\widehat{A_2} &= k\left(k_5 - k_4\right)
\end{aligned}$$

SUMMING UP

$$
\begin{aligned}
I[n] &= B_0 V[n] + s1[n-1] \\
s1[n] &= s2[n-1] - A_1 I[n] \\
s2[n] &= -B_0 V[n] - A_2 I[n] \\
B_0 &= \frac{2 f_\mathrm{s} C}{1 + 2 f_\mathrm{s} R C + 4 {f_\mathrm{s}}^2 L C} \\
A_1 &= \frac{2 - 8 {f_\mathrm{s}}^2 L C}{1 + 2 f_\mathrm{s} R C + 4 {f_\mathrm{s}}^2 L C} \\
A_2 &= \frac{1 - 2 f_\mathrm{s} R C + 4 {f_\mathrm{s}}^2 L C}{1 + 2 f_\mathrm{s} R C + 4 {f_\mathrm{s}}^2 L C}
\end{aligned}
$$

→

$$
\begin{aligned}
x[n] &= B_0 V[n] \\
I[n] &= x[n] + s1[n-1] \\
s1[n] &= s2[n-1] - A_1 I[n] \\
s2[n] &= \widehat{A_2} I[n] - x[n] \\
k_1 &= 2 f_\mathrm{s} \\
k_2 &= k_1 C \\
k_3 &= k_1 k_2 L \\
k_4 &= 1 + k_3 \\
k_5 &= k_2 R \\
k &= \frac{1}{k_4 + k_5} \\
B_0 &= k k_2 \\
A_1 &= k\left(2 - 2 k_3\right) \\
\widehat{A_2} &= k\left(k_5 - k_4\right)
\end{aligned}
$$

Saved 5 pow2, 2 div, 21 mul, 5 add, 1 sign

LET'S DIVE DEEPER #1

- Let's assume R, L, C are parameters
- This part is audio-rate:

$$
\begin{aligned}
x[n] &= B_0 V[n] \\
I[n] &= x[n] + s1[n-1] \\
s1[n] &= s2[n-1] - A_1 I[n] \\
s2[n] &= \widehat{A_2} I[n] - x[n]
\end{aligned}
$$

  That is 3 add and 3 mul
- Everything else (1 div, 9 mul, 4 add) can be computed outisde of the audio loop…
- … unless smoothing, which can however be done at a lower rate (e.g., each 4 samples)

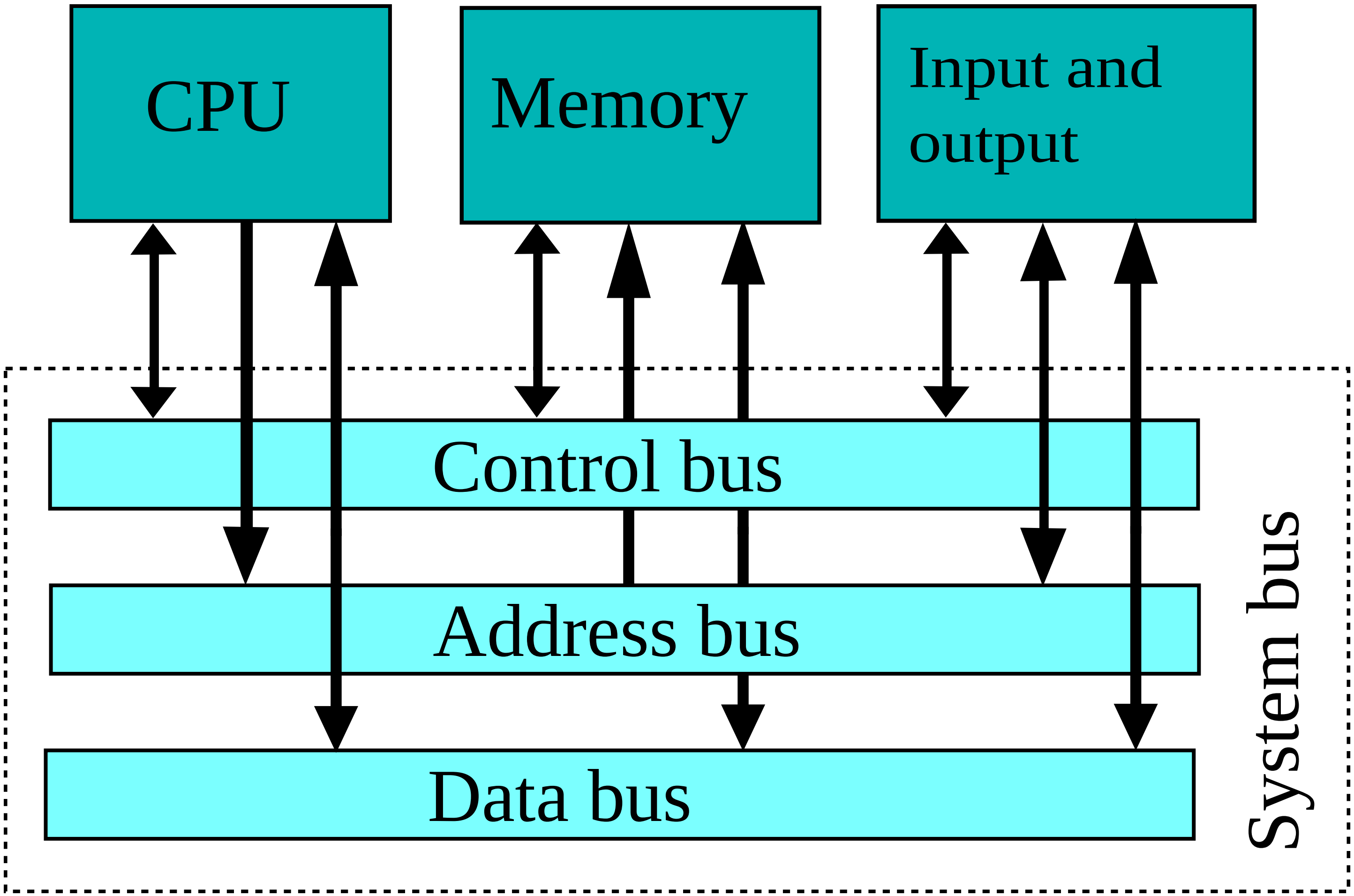
LET'S DIVE DEEPER #2

- $k_1 = 2f_\mathrm{s}$ is sample-rate-constant (1 mul)
- $k_2 = k_1 C$ depends on C
- $k_3 = k_1 k_2 L$ and $k_4 = 1 + k_3$ depend on L and C
- $k_5 = k_2 R$ depends on R and C
- k, B_0, A_1, and $\widehat{A_2}$ depend on R, L, and C

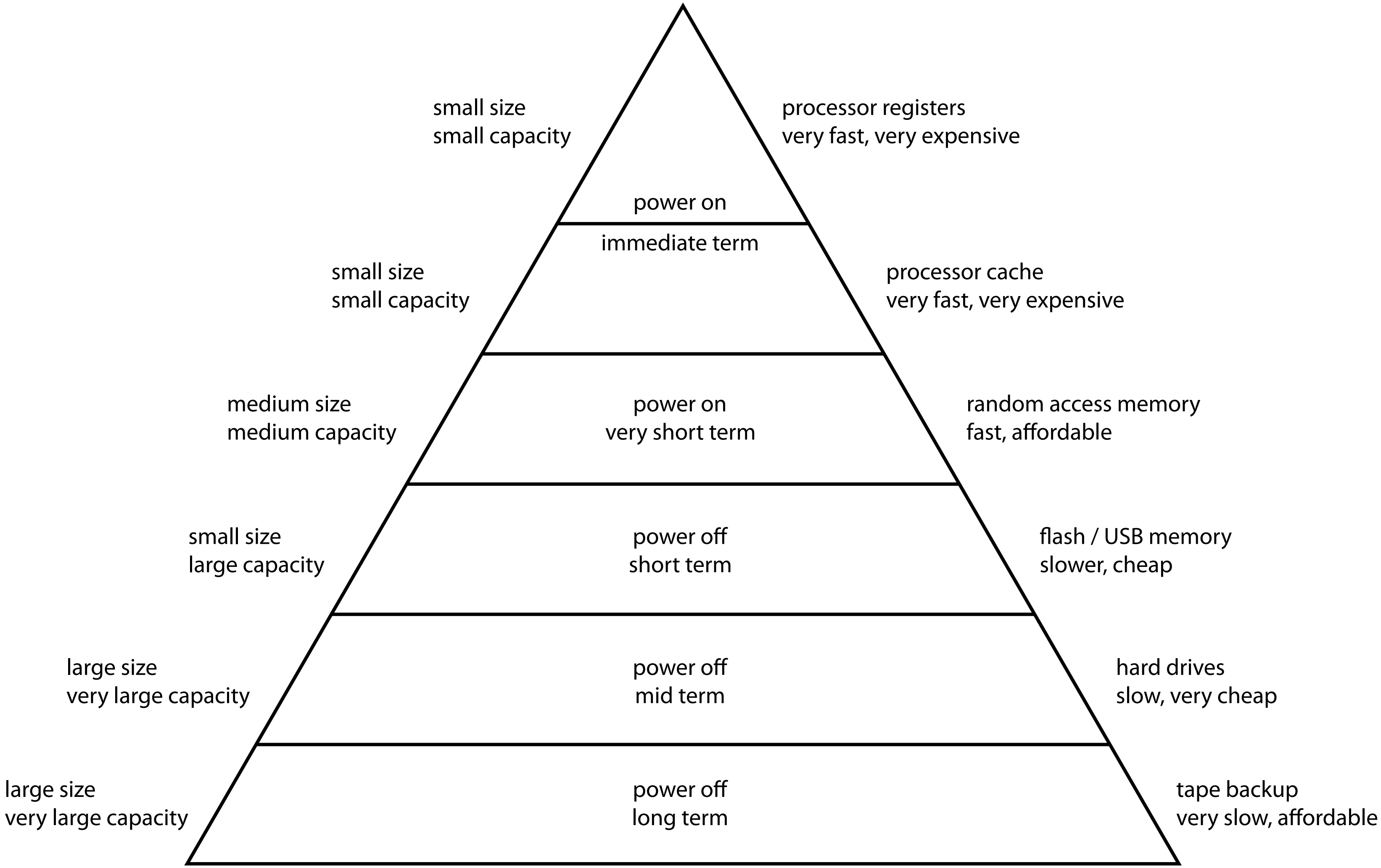
VALIDITY TABLE

What changed	k_2	k_3	k_4	k_5	k, B_0, etc.	Cost
Nothing	✔	✔	✔	✔	✔	0
R	✔	✔	✔	✘	✘	1 div, 5 mul, 3 add
L	✔	✘	✘	✔	✘	1 div, 5 mul, 4 add
C	✘	✘	✘	✘	✘	1 div, 8 mul, 4 add
R, L	✔	✘	✘	✘	✘	1 div, 7 mul, 4 add
L, C	✘	✘	✘	✘	✘	1 div, 8 mul, 4 add
R, L, C	✘	✘	✘	✘	✘	1 div, 8 mul, 4 add

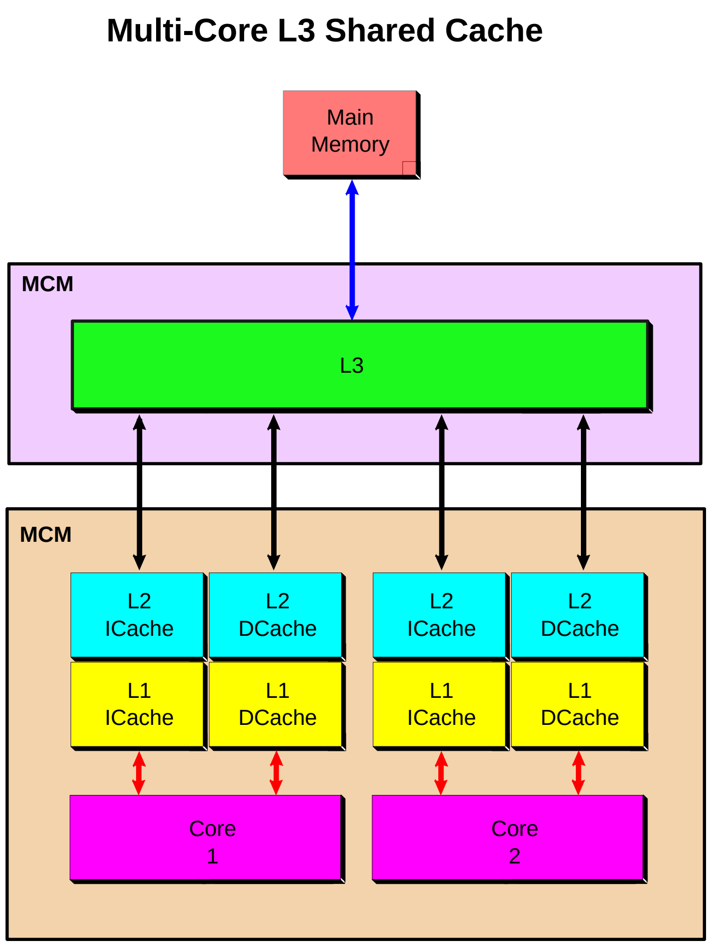
BUS


CPU
Memory
Input and output
Control bus
Address bus
Data bus
System bus

MEMORY HIERARCHY



CPU CACHE


Multi-Core L3 Shared Cache
Main Memory
MCM
L3
MCM
L2 ICache
L2 DCache
L2 ICache
L2 DCache
L1 ICache
L1 DCache
L1 ICache
L1 DCache
Core 1
Core 2

CACHE STORAGE ARRAY

Tag	Line
Address 1	Page 1
Address 2	Page 2
...	...
Address N	Page N

READING FROM MEMORY

value = read(address)

1. Address is examined
2. Check if address is part of a line in cache
3. If so, return the data in cache (*cache hit*) → FAST
4. Otherwise, replace a line with the needed line from memory and return data (*cache miss*) → SLOW

WRITING TO MEMORY

write(address, value)

1. *Write-through*: both cache and memory are updated → SLOW
2. *Write-back*: only update in cache and associate a *dirty bit* to each cache line → FAST but . . .
3. The system needs to maintain *cache coherency* (think multi-core or SMP) → SLOW

MEMORY USAGE TIPS

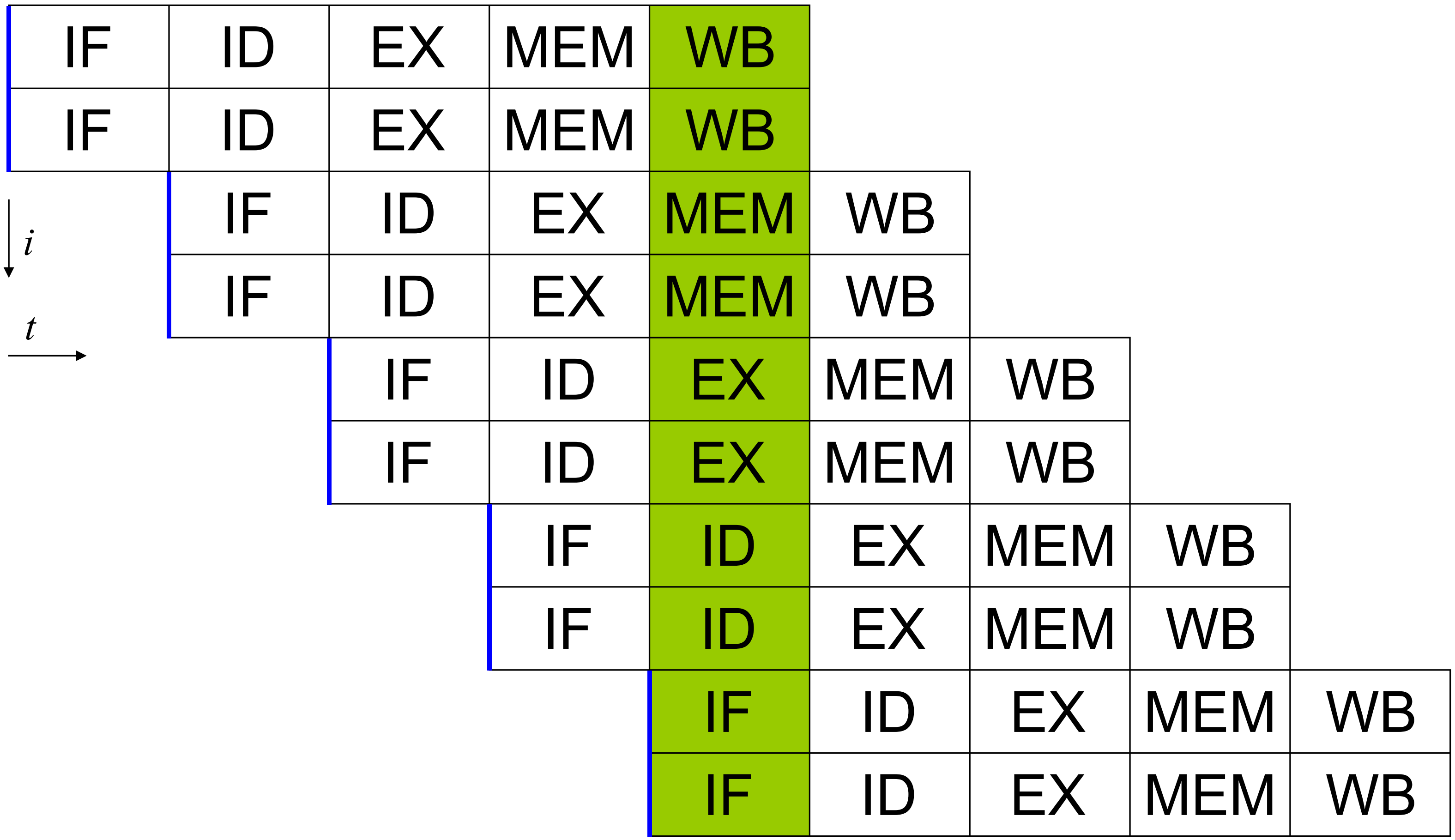
- Limit memory usage (code and data)
- Improve memory locality (code and data)
- Pre-allocate all needed memory if possible
- Inline small functions
- Lock code to a given core
- Avoid writing to memory, prefer local variables
- Many small loops preferable (beware of buffering)
- Avoid concurrent operations at this level
- Lookup tables are ok as long as they are small
- Benchmarking outside of context can be misleading

POINTER ALIASING

```
void f(int *a, int *b, int *c) {
  *a += *c;
  *b += *c;
}
```

- Compiler cannot exclude that arguments refer to the same memory location
- This prevents many compiler optimizations
- Solution: use `restrict` (C only, kind of)

SUPERSCALAR PIPELINE



PIPELINE HAZARDS

- Structural hazards:
  - 2 instr. use same CPU resources at the same time
  - CPU/ISA design issue, don't care
- Data hazards:
  - instr. uses data before it is available in regs
  - not an issue in modern CPUs… sort of…
- Control hazards:
  - branching
  - can cause pipeline stall and inconsistent performance

CONDITIONAL MOVE

- y = c ? a : b
- Does not stall pipeline
- cmov (x86), blend (SSE/AVX), csel (ARM), bsl (Neon)
- Only chooses between computed values
- Compilers normally generate these in obvious cases
- Example:

```
int min(int a, int b) {
  return a < b ? a : b;
}
```

BRANCHLESS ALGORITHMS

- Idea: avoid branching altogether
- Example:

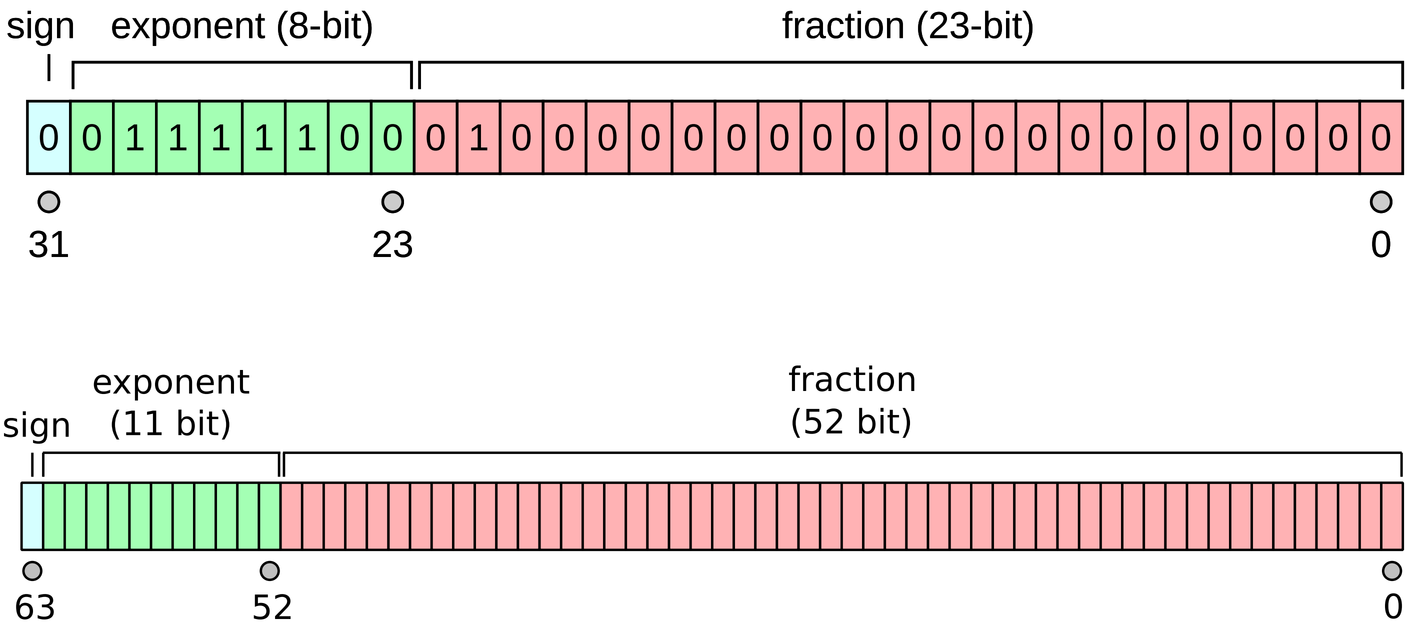
```
int abs(int x) {
  return x * ((x > 0) - (x < 0));
}
```

- Often bad in such simple cases

DIGITAL SIGNAL PROCESSORS

- Optimized for streaming data
- Peculiar architectures (e.g., Harvard)
- MAC instructions (polynomials, FIR, FFT)
- Modulo addressing (circular buffers)
- Saturation arithmetic

IEEE-754 REPRESENTATION



$$\text{value} = (-1)^s 2^{e-b} 1.m$$

$$b = 127 \text{ (32 bit)}, b = 1023 \text{ (64 bit)}$$

SPECIAL VALUES

- Zero(s): $e =$ all 0, $m =$ all 0
- Infinities: $e =$ all 1, $m =$ all 0
- NaN: $e =$ all 1, $m =$ not all 0
- Denormals: $e =$ all 0, $m =$ not all 0

DEALING WITH DENORMALS

- Operations on denormals can be REALLY SLOW
- They occur naturally in IIR filters
- Largest denormal (32 bit):
  $(1 - 2^{-23}) \times 2^{-126} \approx 1.18 \times 10^{-38}$
- You just don't want them around
- Simple solution: enable flush-to-zero and denormals-are-zero CPU flags
- (Clunky) alternatives exist

NOT ALL INSTRUCTIONS ARE CREATED EQUAL

Instruction	Throughput	Latency
add, sub	0.5	2/4
mul	0.5	4
div	3	11
~rcp	1	4
sqrt	3	12
round	1/1.03	8
and, or (int)	0.33	1
shift (int)	0.5*	1*

From Intel Intrinsics Guide, * = typical

FAST $\mathrm{trunc}(x)$

$$\mathrm{trunc}(x) = x < 0\,?\lceil x \rceil : \lfloor x \rfloor$$

```
float trunc(float x) {
  union { float f; uint32_t u; } v;
  v.f = x;
  const int32_t ex = (v.u & 0x7f800000u) >> 23;
  int32_t m = (~0u) << clipi32(150 - ex, 0, 23);
  m &= ex > 126 ? ~0 : 0x80000000;
  v.u &= m;
  return v.f;
}
```

Adapted from Brickworks (https://www.orastron.com/brickworks)

FAST $\log_2()$

```
float log2(float x) {
  union { float f; int32_t i; } v;
  v.f = x;
  int e = v.i >> 23;
  v.i = (v.i & 0x007fffff) | 0x3f800000;
  return (float)e - 129.213475204444817f
    + v.f * (3.148297929334117f
    + v.f * (-1.098865286222744f
    + v.f * 0.1640425613334452f));
}
```

Adapted from Brickworks (https://www.orastron.com/brickworks)

LIBM CONSIDERED HARMFUL

```
static const float
ln2 = 0.69314718055994530942,
two25 =    3.355443200e+07, /* 0x4c000000 */
Lg1 = 6.6666668653e-01, /* 3F2AAAAB */
Lg2 = 4.0000000596e-01, /* 3ECCCCCD */
Lg3 = 2.8571429849e-01, /* 3E924925 */
Lg4 = 2.2222198546e-01, /* 3E638E29 */
Lg5 = 1.8183572590e-01, /* 3E3A3325 */
Lg6 = 1.5313838422e-01, /* 3E1CD04F */
Lg7 = 1.4798198640e-01; /* 3E178897 */

static const float zero   =  0.0;

float
  ieee754_log2f(float x)
```

Taken from the GNU C Library (https://www.gnu.org/software/libc/)

"Unsafe" compiler optimization can alleviate this

TAYLOR AND PADÉ

- Horner's method:
  $a_0 + x(a_1 + x(a_2 + \ldots + x(a_{n-1} + xa_n)))$
- Polynomials are typically fast to compute
- Division is slower, while reciprocal is ok
- Good around a single point
- Example:
  $\tan(x) \approx x + \frac{x^3}{3} + \frac{2x^5}{15} = x\left(1 + x^2\left(\frac{1}{3} + x^2\frac{2}{15}\right)\right)$
  1% relative error at $x \approx 0.75$
  Prewarping at $f_\mathrm{s} = 44100$ Hz → $f \approx 10$ kHz

LINEAR INTERPOLATION

- $p(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0)$, $x \in [x_0, x_1]$
- Easy and cheap if $x_1 - x_0$ is "fixed"
- Never overshoots
- Preserves C^0 differentiability
- Piecewise approximation passes through chosen points
- Good for dense lookup tables (which are often bad for caching)

CUBIC SPLINE INTERPOLATION

- 3rd-degree polynomial in $[x_0, x_1]$ passing through $(x_0, f(x_0)), (x_1, f(x_1))$ and with matching first-derivative values in such points (unique)
- Can overshoot
- Preserves C^1 differentiability
- Cheap, smooth, can cover wider ranges than linear interpolation but requires more data per interval
- Better for analytical nonlinearities

NUMERICAL OPTIMIZATION

Classical optimization methods can be used, e.g.:

- to directly approximate by regression analysis, NN, etc.
- to optimize parameters of underdetermined systems
- to find optimal points to divide into piecewise-defined approximations
- to find "magic numbers"

A tanh() APPROXIMATION

```
float tanh(float x) {
  const float xm =
    clip(x, -2.115287308554551f, 2.115287308554551f);
  const float axm = abs(xm);
  return xm + xm * axm
    * (0.01218073260037716f * axm - 0.2750231331124371f);
}
```

Adapted from Brickworks (https://www.orastron.com/brickworks)

ROOT-FINDING ALGORITHMS

- Needed for implicitly-defined functions
- Also useful to approximate closed-form expressions
- They need a good enough first guess
- Famous methods: bisection, Newton-Raphson (needs derivative), secant

A RECIPROCAL ALGORITHM

```c
float rcp(float x) {
  union { float f; int32_t i; } v;
  v.f = x;
  v.i = 0x7ef0e840 - v.i;
  v.f = v.f + v.f - x * v.f * v.f;
  return v.f + v.f - x * v.f * v.f;
}
```

Adapted from Brickworks (https://www.orastron.com/brickworks)

VECTORIZATION

- $x = y \text{ op } z \Rightarrow \vec{x} = \vec{y} \text{ op } \vec{z}$ (2…16 elems)
- a.k.a., Single Instruction Multiple Data (SIMD)
- x86/x64: SSE, AVX - ARM: Neon
- Not all algorithms can be easily/fully vectorized
- Data to be aligned and moving more complicated
- Same memory speed, code size increases
- Actual speedup less than vector size

CPU VS FPU VS SIMD

- FPU: floating point unit
- CPU vs CPU + FPU vs CPU + FPU + SIMD
- Each has its own registers, ISA, computing model, some overlap
- CPU only: compiler translates to soft float (SLOW) or emulated by kernel on exception (REALLY SLOW)
- CPU + FPU: all float in FPU, for ints it depends…
- CPU + FPU + SIMD: all float in SIMD (FPU kept for compatibility), better int support than FPU

SIMD INTRINSICS

C API mapping directly to CPU instructions

Example (SSE):

```
__m128 _mm_add_ps (__m128 a, __m128 b)
```

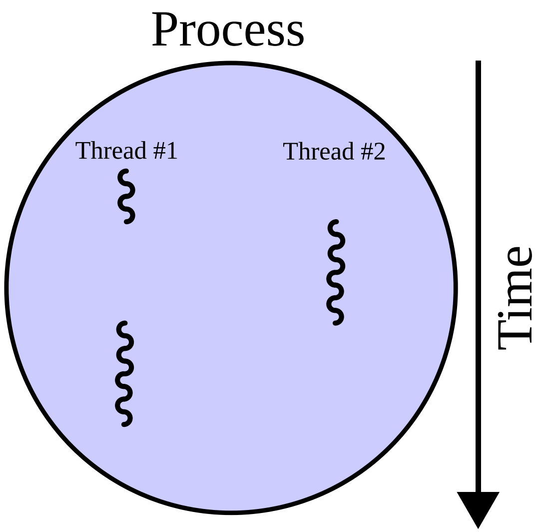
maps to

```
addps xmm, xmm
```

Intel Intrinsics Guide:
https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

ARM Intrinsics:
https://developer.arm.com/architectures/instruction-sets/intrinsics/

MULTITHREADING



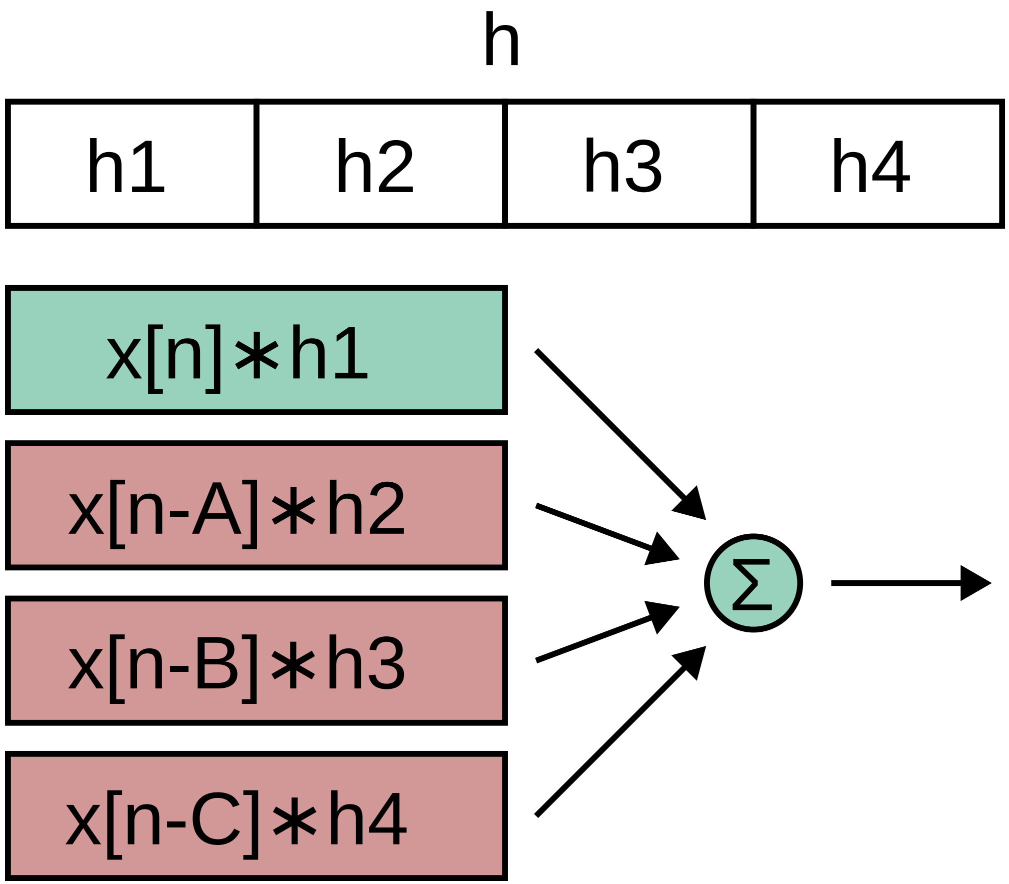
User Cburnett on Wikipedia,
CC BY-SA 3.0

- Multiple concurrent threads of execution per process, supported by OS
- Threads can be mapped to different CPUs/cores
- Threads share the same memory space
- Synchronization to be explicitly coded (can be hard)

MULTITHREADED DSP

- Pure multithreaded DSP only makes sense if large degree of parallelism (low granularity, few interdependencies)
- UI and audio thread on non-embedded platforms
- UI thread can be used to compute coefficients if changes can be slow, sporadic and don't need smoothing

PARTITIONED CONVOLUTION



THANK YOU!

ORASTRON

www.orastron.com

info@orastron.com