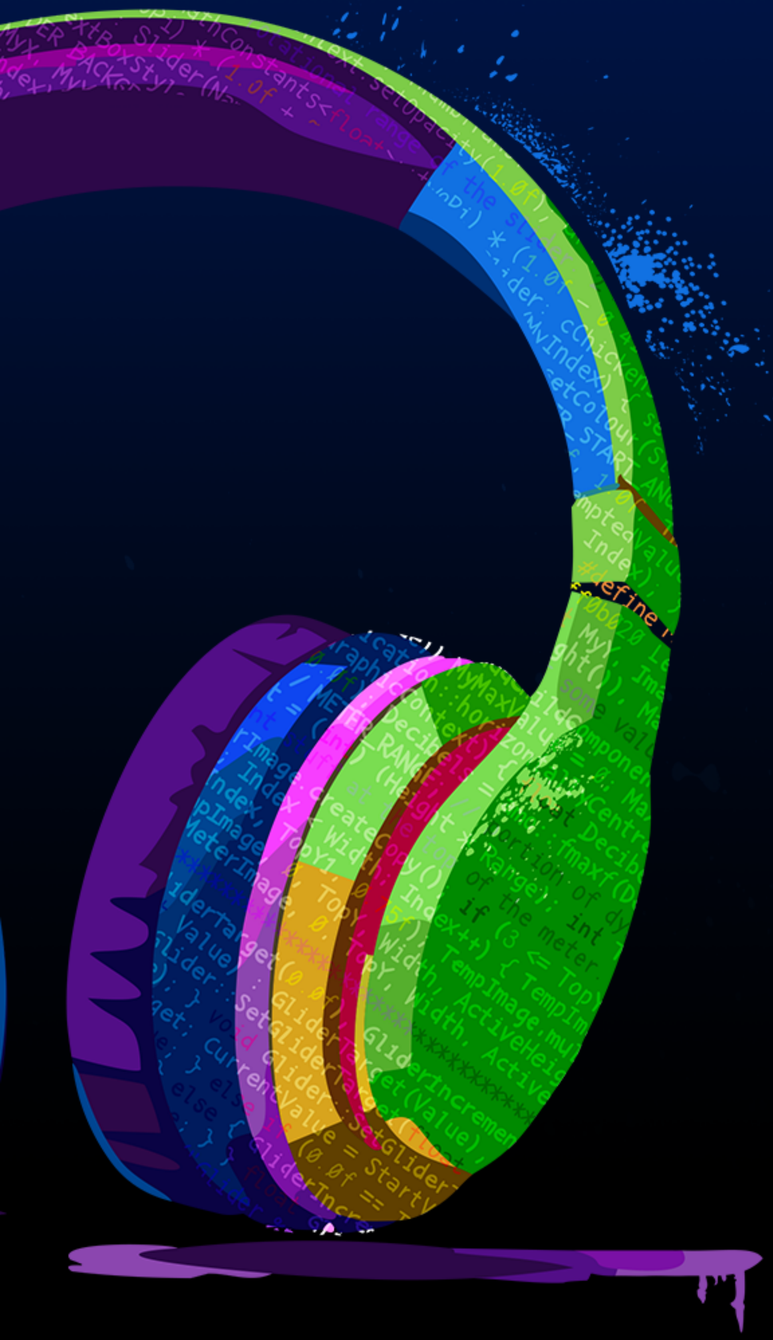


ADC24 Bristol

BRANCH-FREE OSCILLATORS FOR FUN AND PROFIT

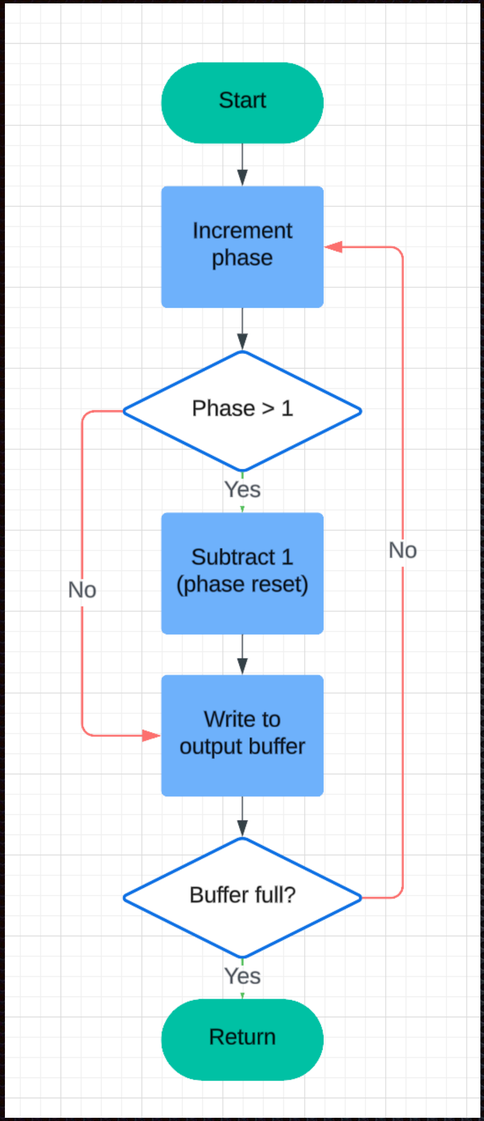
ANGUS HEWLETT


if

Single **P**rogram **M**ultiple **D**ata

To DSL or not to DSL?

Branches? What branches?




```
void naive_sawtooth(float inc, float* buffer, int smps, float& ph)
{
    for (int i = 0; i < smps; i++)
    {
        ph += inc;
        if (ph >= 1.f)
        {
            ph -= 1.f;
        }
        buffer[i] = ph;
    }
}
```

Language features:
if
switch
for
do/while
“?” ternary operator
min/max (maybe.)
memory operations (most.)
synchronisation primitives
vtable lookups

“Language VM”

Aside: The C++ virtual machine

Observable behaviour is well-defined…

.. but compilers may achieve this **however they please**.

If performance depends on the
compiler's decisions, this may be *fragile*.

Optimising compilers make a best effort…
we can often help by giving it more to work with,
but sometimes their "help" is counterproductive.

"Language VM"

Machine instructions and instruction flow

https://godbolt.org

```
void naive_sawtooth(float inc, float* buffer, int smps, float& ph)
{
    for (int i = 0; i < smps; i++)
    {
        ph += inc;
        if (ph >= 1.f)
        {
            ph -= 1.f;
        }
        buffer[i] = ph;
    }
}
```

```
.LBB3_1:
        ldr     w8, [sp, #12]
        ldr     w9, [sp, #28]
        subs    w8, w8, w9
        b.ge    .LBB3_6
        b       .LBB3_2
.LBB3_2:
        ldr     s1, [sp, #44]
        ldr     x8, [sp, #16]
        ldr     s0, [x8]
        fadd    s0, s0, s1
        str     s0, [x8]
        ldr     x8, [sp, #16]
        ldr     s0, [x8]
        fmov    s1, #1.00000000
        fcmp    s0, s1
        b.lt    .LBB3_4
        b       .LBB3_3
.LBB3_3:
        ldr     x8, [sp, #16]
        ldr     s0, [x8]
        fmov    s1, #1.00000000
        fsub    s0, s0, s1
        str     s0, [x8]
        b       .LBB3_4
.LBB3_4:
        ldr     x8, [sp, #16]
        ldr     s0, [x8]
        ldr     x8, [sp, #32]
        ldrsw   x9, [sp, #12]
        str     s0, [x8, x9, lsl #2]
        b       .LBB3_5
.LBB3_5:
        ldr     w8, [sp, #12]
        add     w8, w8, #1
        str     w8, [sp, #12]
```

Add

Compare

Branch

Subtract

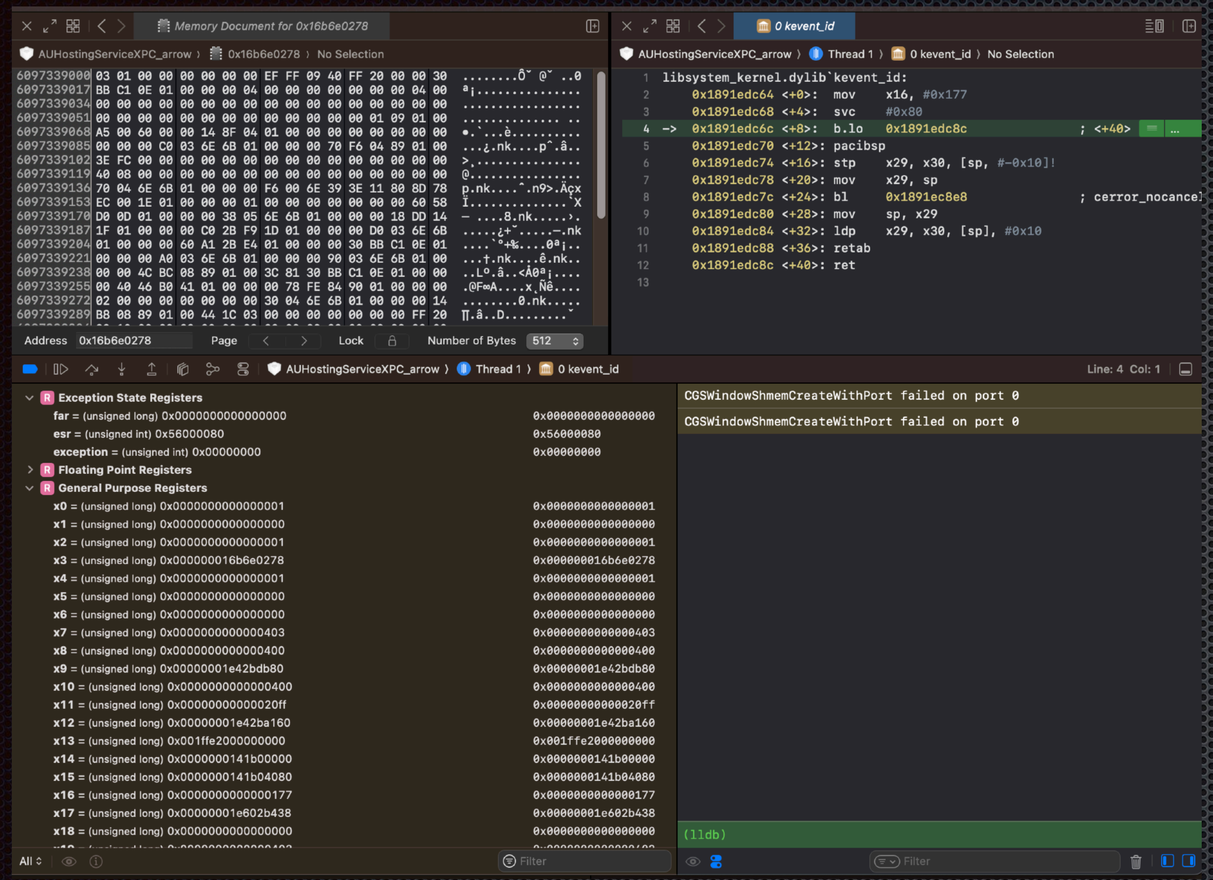
Screenshot

AMD 64

M1

“Machine-language VM”

Aside: The machine-language virtual machine

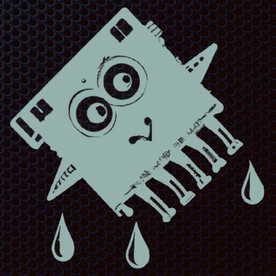


NOT PICTURED:

Instruction fetch address
Execution pipeline
Address-space remapping
Caches (L1I, L1D, L2, L3…)
Rename registers
Branch predictors
Load / store queues / buffers
Shared execution resources (SMT)

SPECTRE

MELTDOWN



Machine-language VM

Why branchless? Parallelism recap.

Apple M1: “An octa/deca-core superscalar CPU with 14-wide dispatch and 128 bit NEON SIMD units.”

~~**TLP**: Thread- (or process)- level parallelism: Cores~~

ILP: Instruction-level parallelism: Execution pipeline (“front end”, “back end”)

DLP: Data-level parallelism: SIMD lanes

Core designs are (mostly) common across a given CPU family / generation.

Designs vary but these principles are generally applicable (ARM “A”-class & x64).

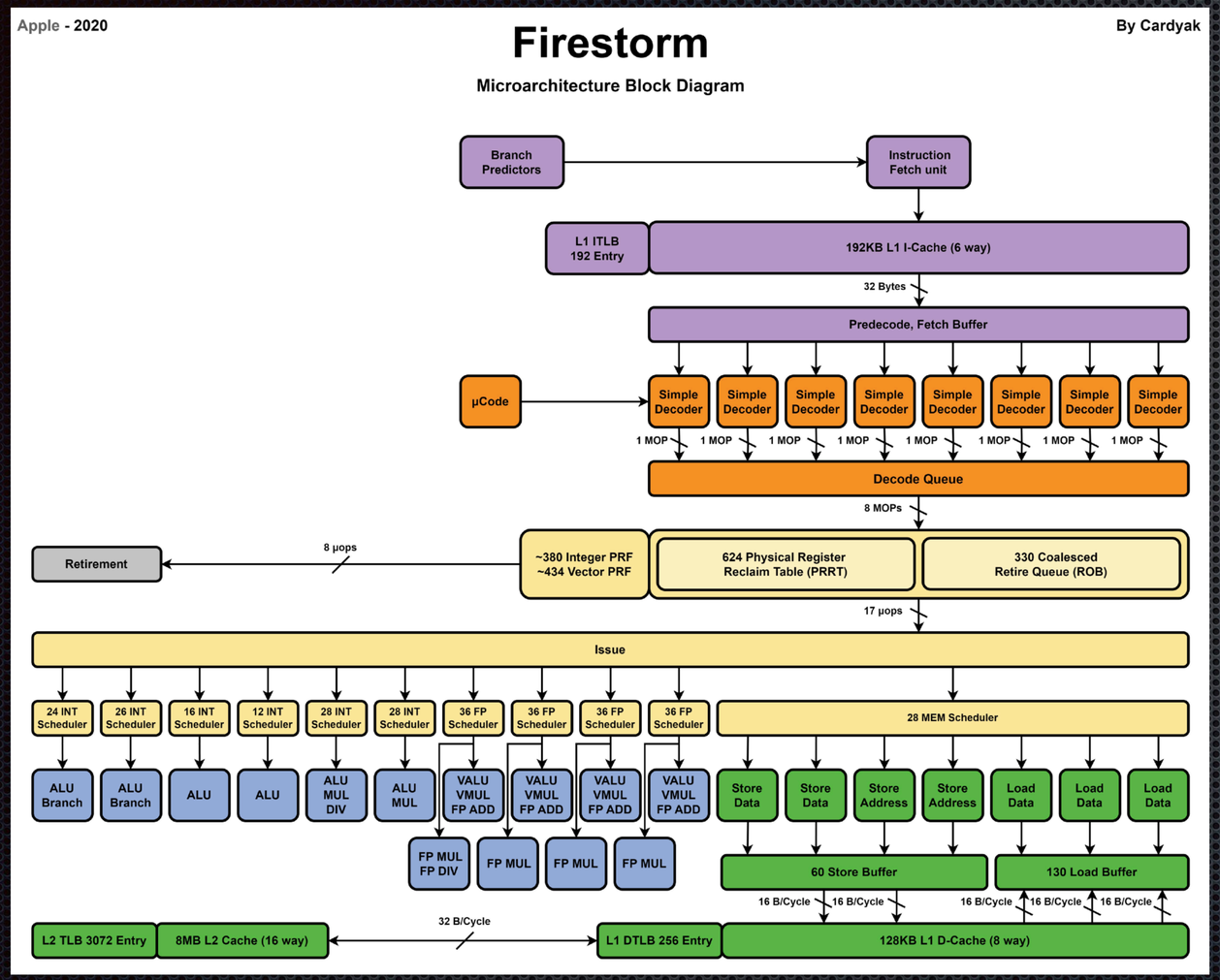
Thread-level parallelism? Why not?

- Already used by the host & the OS
- Optimal for big chunks of work (>100µs / 10^6 instructions)
- Data / core / cache synchronisation and safety can be complex & expensive
- Doesn't belong in your inner loop

Each type of parallelism tends to be one-shot: best used at one hierarchical level only.

Assumption: No SMT. No in-lane vectorisation.

CPU architecture: The Apple M1 “Firestorm” superscalar RISC CPU core.



Front end: organises the work

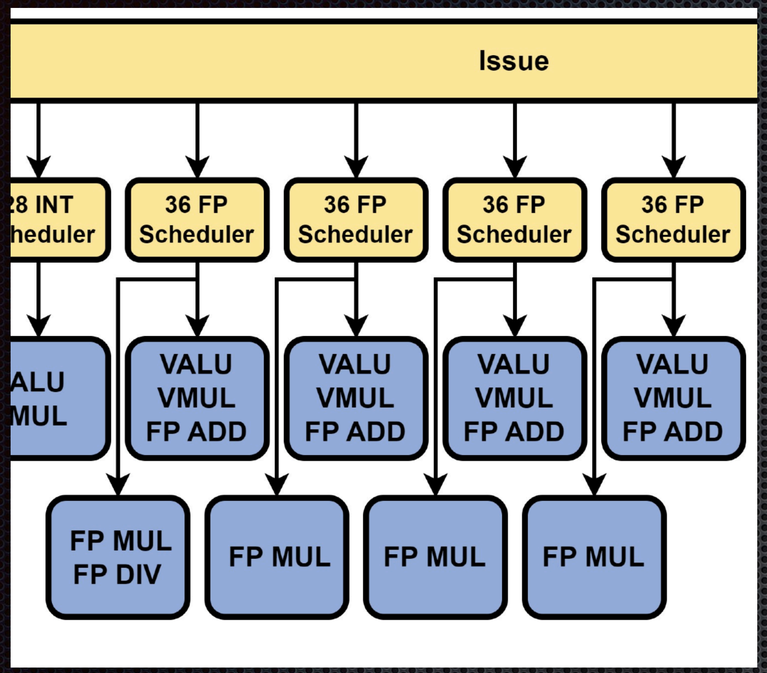
Fetch, Decode/“Crack”, Schedule, Rename, Predict

Back end: does the work

Load, Store, Integer/Float Math, Logic, Vector SIMD

CPU scheduler make a best effort - but we can help by giving it more to work with.

CPU architecture: zoom and enhance…



Four NEON SIMD units each 128 bits wide

(4x float32 single-precision: also int32 & double)

Four instructions per clock (multiply, add, logic, compare…)

16 float32's per clock tick

Why branchless?

Back-end execution: Instruction latency and data dependencies

a = a * b;

```
fmul    v23.4s, v27.4s, v23.4s
```

Executes at t=0

Result → Input

c = c - a;

```
fsub    v27.4s, v27.4s, v23.4s
```

Cannot execute until t=3

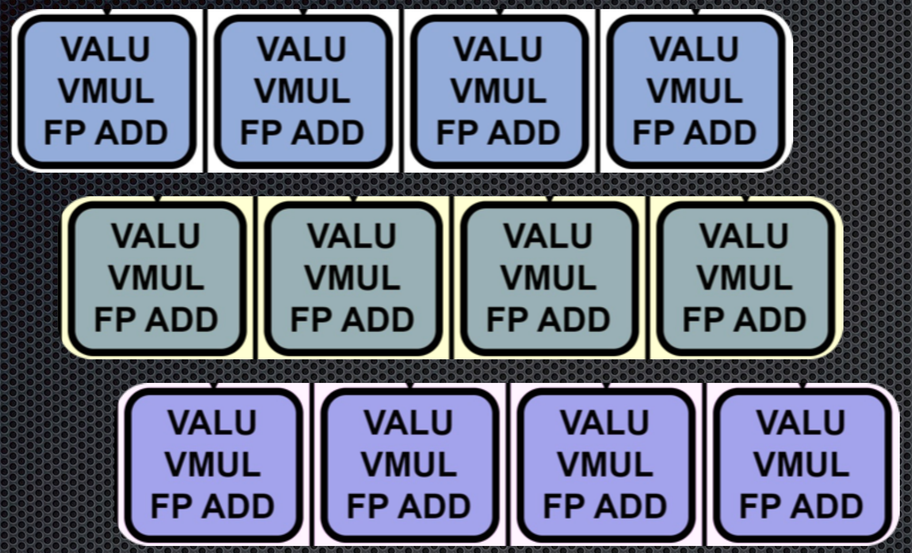
(typical figures for simple operations on modern CPUs. 1/x, sqrt(x), log(x), x^n may be much slower)

Simple operations: add, subtract, multiply, min, max, abs, comparisons, bitwise logic, shift/shuffle.

1-10ms	10-25μs	0.25ns
NOT audio latency!	NOT filter unit-sample delay!	One CPU clock-tick.

Why branchless?

Imagining the CPU as a slower, wider machine



T=0

Ready at T~=3

T=1

Ready at T~=4

T=2

Ready at T~=5

- One-third the clock rate.
- 48 data streams with single-cycle latency.
- Potential for 16x to 48x greater throughput?

Note: compile- and runtime instruction re-ordering

Limits to width: the register file.

ARM: **32** visible 128-bit registers

X64: **16** visible 256-bit registers

AVX512f: **32** visible 512-bit registers

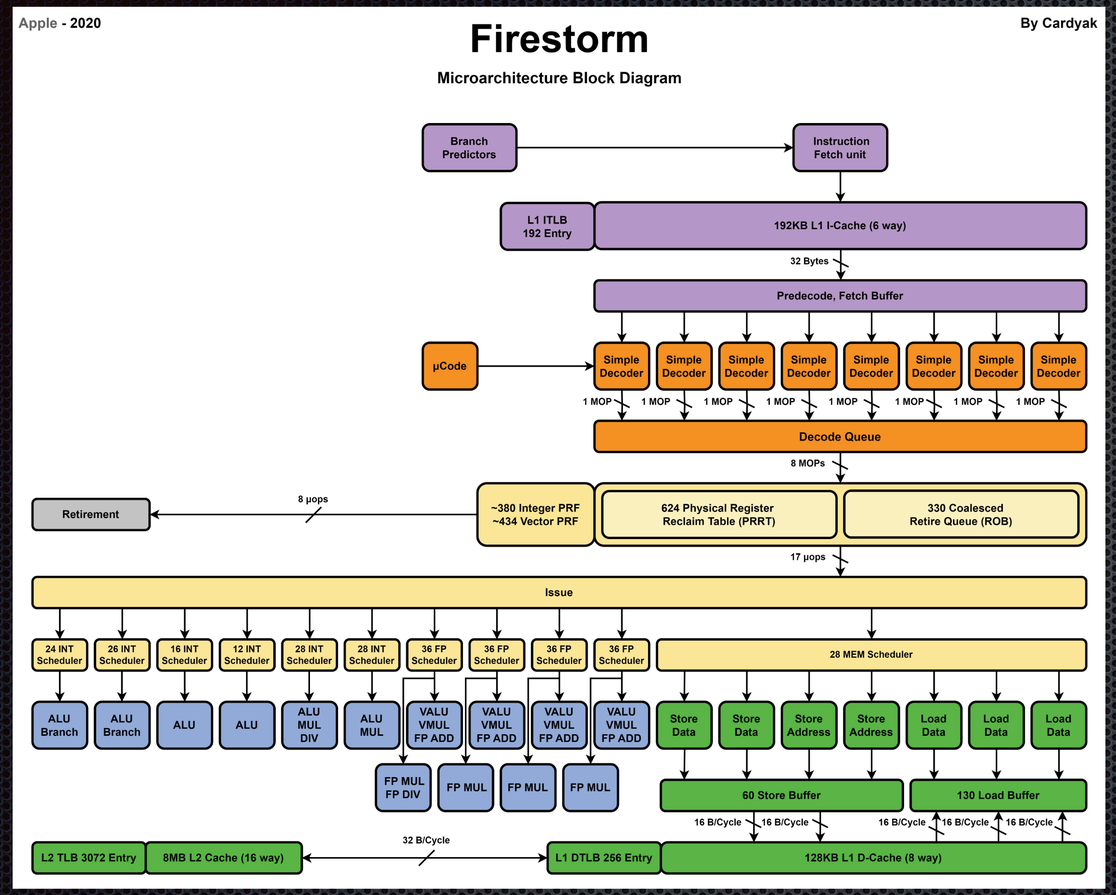
ARM64EC: **16** visible 128-bit registers

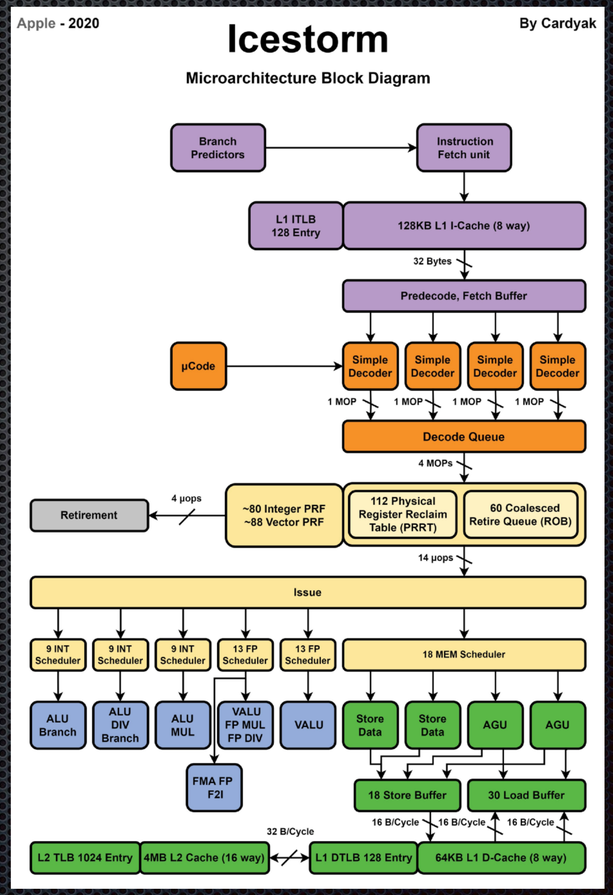
12-wide instruction stream (48-wide data stream) may be too wide: 2.5 registers per stream.

“Spills and refills” result in additional machine instructions, more work for the CPU, and may degrade performance.

Note: compile- and runtime instruction re-ordering

CPU heterogeneity: “Performance”, “Efficiency”, “big.LITTLE”





Performance characteristics - and ideal instruction sequences! - can vary even on the same device.

Evaluate!

During development

At runtime?

Use high resolution timer

Aim for a run-size in the ~tens of microsecond range (64 samples x 1000 iterations?)

Too small: sampling error. Too large: thread interrupts

Run the whole test 100+ times (=> sub 1s), sort results, discard upper and lower quartiles and average.

Core reassignments can cause negative times; thread interrupts can cause long times.

```
Avg runtime :107.592003 Interleave width: 1 (x4) 4      (for iter):262144       Normalised exec time (1 osc, 1smp, pSec) 410.430939      (MOscSamps/s: 2436)     (64vc 88k Est CPU%: 0.231155)
Avg runtime :130.412003 Interleave width: 2 (x4) 8      (for iter):524288       Normalised exec time (1 osc, 1smp, pSec) 248.741165      (MOscSamps/s: 4020)     (64vc 88k Est CPU%: 0.140091)
Avg runtime :147.000000 Interleave width: 4 (x4) 16     (for iter):1048576      Normalised exec time (1 osc, 1smp, pSec) 140.190140      (MOscSamps/s: 7133)     (64vc 88k Est CPU%: 0.0789551)
Avg runtime :187.000015 Interleave width: 8 (x4) 32     (for iter):2097152      Normalised exec time (1 osc, 1smp, pSec) 89.168556       (MOscSamps/s: 11214)    (64vc 88k Est CPU%: 0.0502197)
Avg runtime :236.000015 Interleave width: 10 (x4) 40    (for iter):2621440      Normalised exec time (1 osc, 1smp, pSec) 90.026863       (MOscSamps/s: 11107)    (64vc 88k Est CPU%: 0.0507031)
Avg runtime :278.430023 Interleave width: 12 (x4) 48    (for iter):3145728      Normalised exec time (1 osc, 1smp, pSec) 88.510521       (MOscSamps/s: 11298)    (64vc 88k Est CPU%: 0.0498491)
Avg runtime :364.542023 Interleave width: 16 (x4) 64    (for iter):4194304      Normalised exec time (1 osc, 1smp, pSec) 86.913589       (MOscSamps/s: 11505)    (64vc 88k Est CPU%: 0.0489497)
Avg runtime :1219.344116        Interleave width: 32 (x4) 128   (for iter):8388608      Normalised exec time (1 osc, 1smp, pSec) 145.357147      (MOscSamps/s: 6879)     (64vc 88k Est CPU%: 0.0818651)
```

Note: compile- and runtime instruction re-ordering

Foundational techniques

- SIMD intrinsics (wrapped)
- Data interleaving (wrapped)

- Compare-and-mask ops
- Clip-and-scale window functions
- Polynomial approximations

- Avoid unrolling the inner-loop: code size, per-sample dependencies, book-keeping

SIMD & interleaved intrinsics wrapper

```
template <class simd_t> class QuadratureOscillator
{
    typedef simd_t::vec_float vf;

    vf A, B, sinOut, cosOut;

    void process_sample()
    {

        vf temp  = B * sinOut + A * cosOut;
        cosOut = B * cosOut - A * sinOut;
        sinOut = temp;
        write_output (0, sinOut);
        write_output (1, cosOut);
    }
}


typedef simd_wrap<arm_neon, 2> simd;
QuadratureOscillator<simd> x;
x.process_sample();
```

- Write clean, readable code
- “No” performance penalty vs intrinsics or asm
- Trivial to generate different layouts & ISAs for evaluation

- Control-flow statements (if, else…) are unavailable.
- Relational compare operators must evaluate to ‘bool’.
- “?” (Conditional ternary) operator cannot be overloaded.
- Substitute with template-function constructs (“compare_greater” instead of “>”)
- … but dependent name lookups require import via ‘using’.
- Some developer overhead when moving between AoS & SoA.

Moving target (std::**experimental**::simd in standard library) .

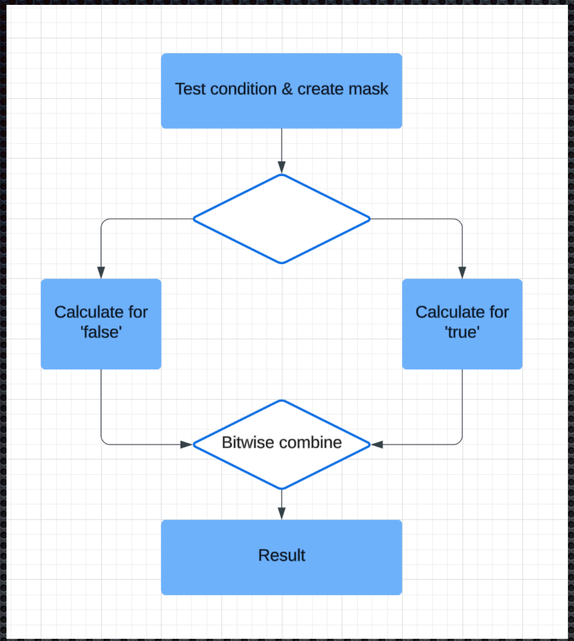
SIMD & interleaved intrinsics wrapper

```
// Define an array-of-Neon-vector class with a conversion from float, and an add operator.
template <int N> class alignas(16) vf
{
public:
    static constexpr int vector_width = N*4;
    float32x4_t m[N];

    simd_forceinline vf (float x)
    {
        force_unroll for (int i = 0; i < N; i++) m[i] = vdupq_n_f32(x);     // <= correct:
    }

    simd_forceinline const vf& operator+=(const vf<N>& other)
    {
        force_unroll for (int i = 0; i < N; i++) m[i] = vaddq_f32(m[i], other.m[i]);
        return *this;
    }
}
```

- Implements simple math operations (+-*, min/max, comparisons, logic), pick, floor, clip etc.
- Conversion from float - rely on compiler to deduplicate
- Requires some compiler cooperation… 🐊 🐊 🐊
- If in any doubt, use Compiler Explorer or ‘clang -S’

Compare-and-mask operations

```
if (a > b)
{
   x = c;
}
else
{
   x = d;
}
```




```
a:        1.5     2.0     0.6    -0.2
b:        0.8     1.3     3.1     0.3
temp = CMPGT (a, b);  // a > b
temp:   11111111  11111111  00000000  00000000
c:        0.4     0.5     0.6     0.7
d:        0.8     0.9     1.0     1.1
x = AND (temp, c) + ANDNOT (temp, d);
X:        0.4     0.5     1.0     1.1
```

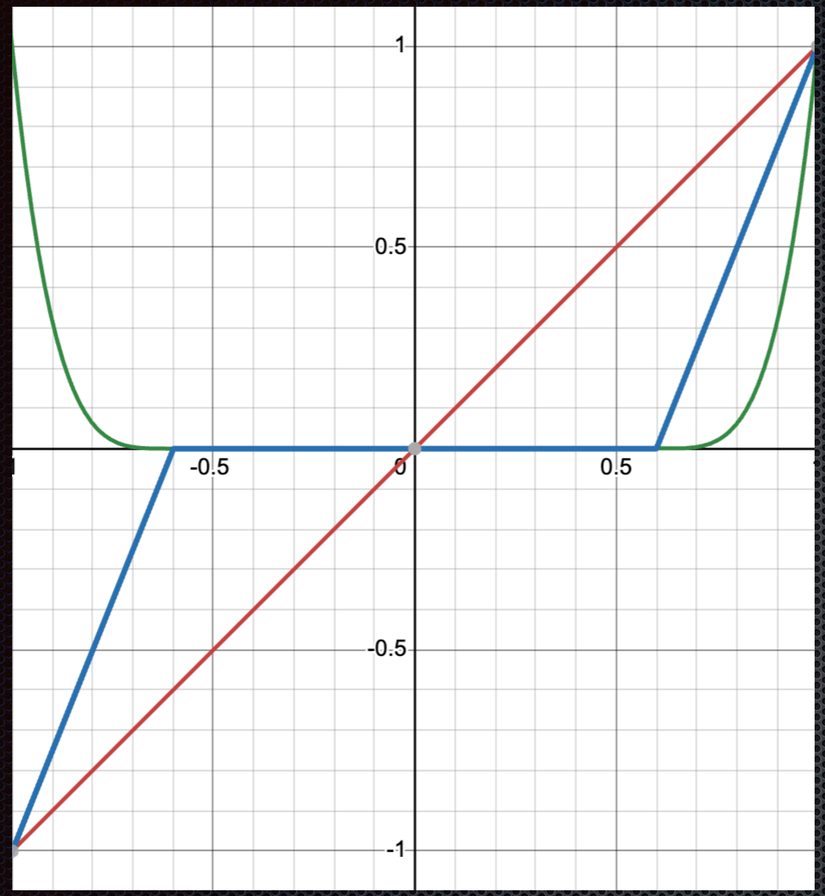
```
inline vec_t pick_gt(a, b, c, d)
{
    vec_t temp = CMPGT(a,b);
    return AND(temp, c) + ANDNOT(temp, d);
}
```

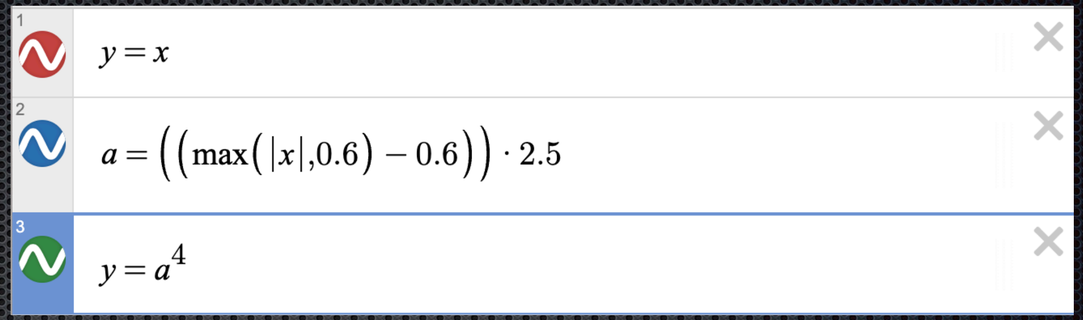
Convenience function: pick_xx (input, comparison, val_if_true, val_if_false).

Works for all widths. **max**, **min**, **clip** functions available for simpler cases.

Can use bitwise OR or vector FADD for the final combination step.

Clip-and-scale window functions



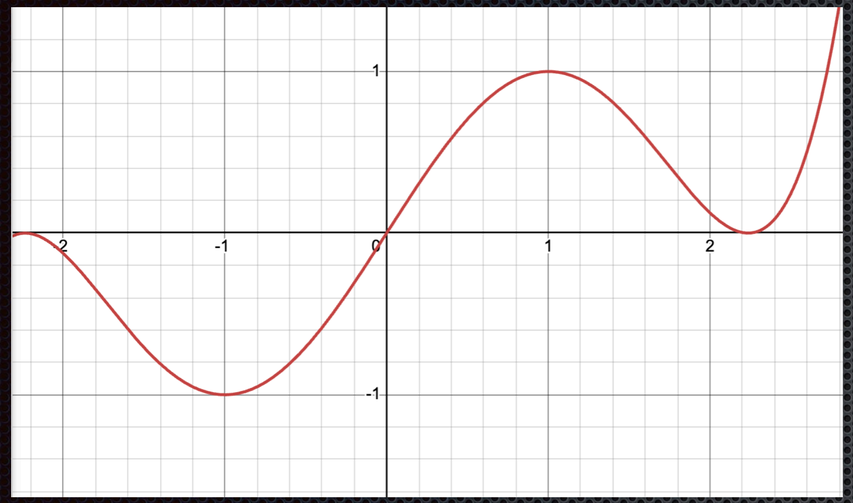


fmul (fsub (fmax (fabs (x) ,a), a),scale);

Generates a window function across multiple lanes in four instructions

Polynomial approximations

$$b = \frac{\left(x^5 - 10x^3 + 25x\right)}{16}$$



5th order polynomial (half sine)

4x multiply, 2x FMA

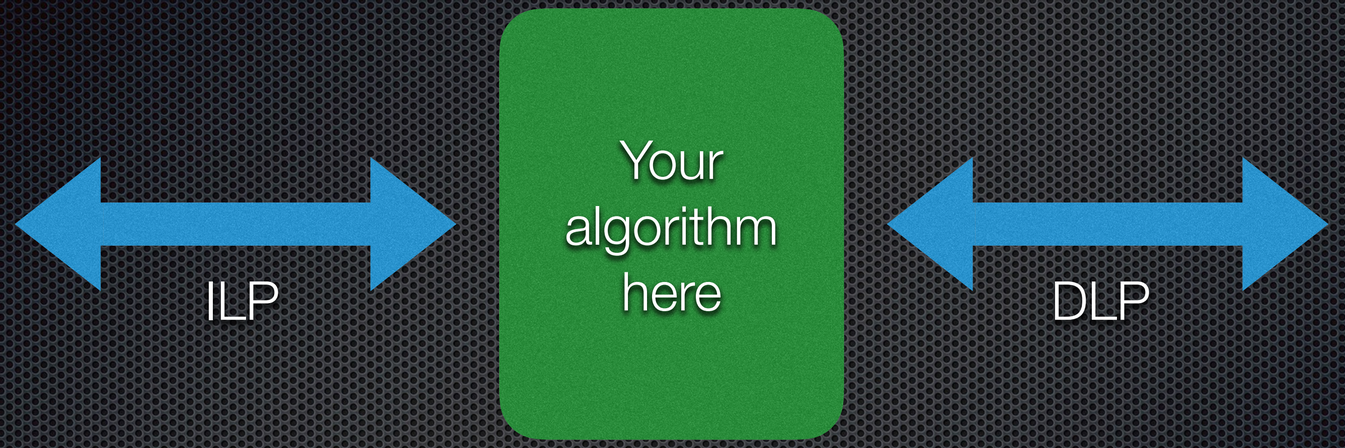
- Do not parallelise well
- Memory-intensive (harmful to cache?)
- 1024-entry LUT: 4kb
- L1 cache: 32-128kb, ~3 cycles
- L2 cache: 15 cycles *per lane*

- Inherently serial*
- Function-call overhead*
- >100 cycles on many CPUs

SIMD & interleaved intrinsics wrapper: Recap!

Voice channel layout

Interleaving factor



SSE4.1

AVX2

AVX512f

NEON

SVE2

Decouples algorithm design from instruction sets and decisions about optimal interleaving - while maintaining “close to optimal” performance.

Interleaving maximises use of available CPU resources and hides instruction dependency / data latency.

Ease of arbitrary code generation x lightweight benchmarking/profiling = runtime code path selection

Let’s make an oscillator: Hello World

```cpp
///
// Naive phase increment "sawtooth" - original branching version
void process_block(std::vector<simd>& out_buffer )
{

    simd ph = m_phase;
    const simd phase_increment = m_inc;

    for (auto& o0 : out_buffer)
    {
        ph += phase_increment;
        if (ph >= 1.f) ph -= 1.f;
        o0 = ph;
    }
    m_phase = ph;
}
```

```cpp
///
// Naive phase increment "sawtooth" - branchless version
void process_block(std::vector<simd>& out_buffer )
{
    simd ph = m_phase;
    const simd phase_increment = m_inc;
    const simd reset_threshold = simd(1.f) - phase_inc;
    const simd reset_increment = simd(-1.f) + phase_inc;


    // Branchless
    for (auto& o0 : out_buffer)
    {
        // Three machine instructions:
        // - compare
        // - select
        // - add
        // ph += (ph >= reset_threshold) ? reset_increment : phase_increment;
        ph += simd::pickge(ph, reset_threshold, reset_increment, phase_increment);
        o0 = ph;
    }
    m_phase = ph;
}
```

Branching - cannot work with SIMD vectors!

Branchless version using **pickge** compare-and-mask.
Pre-subtract for phase reset.

Let’s make an oscillator: Code generation

1x interleaved

```
LBB1_4:
        fcmgt   v4.4s, v2.4s, v0.4s
        bsl     v4.16b, v1.16b, v3.16b
        fadd    v0.4s, v4.4s, v0.4s
        str     q0, [x9, x10]
        add     x10, x10, #16
        cmp     x10, #2048
        b.ne    .LBB1_4
```

7 instructions (4 lanes)

1.75 inst / lane

2x interleaved

```
.LBB2_4:
        fcmgt   v5.4s, v2.4s, v0.4s
        fcmgt   v6.4s, v2.4s, v4.4s
        subs    x10, x10, #1
        bsl     v5.16b, v1.16b, v3.16b
        bsl     v6.16b, v1.16b, v3.16b
        fadd    v0.4s, v5.4s, v0.4s
        fadd    v4.4s, v6.4s, v4.4s
        stp     q0, q4, [x9, #-16]
        add     x9, x9, #32
        b.ne    .LBB2_4
```

10 instructions (8 lanes)

1.25 inst / lane

4x interleaved

```
.LBB3_4:
        fcmgt   v7.4s, v2.4s, v0.4s
        fcmgt   v16.4s, v2.4s, v4.4s
        subs    x10, x10, #1
        fcmgt   v17.4s, v2.4s, v5.4s
        fcmgt   v18.4s, v2.4s, v6.4s
        bsl     v7.16b, v1.16b, v3.16b
        bsl     v16.16b, v1.16b, v3.16b
        bsl     v17.16b, v1.16b, v3.16b
        bsl     v18.16b, v1.16b, v3.16b
        fadd    v0.4s, v7.4s, v0.4s
        fadd    v4.4s, v16.4s, v4.4s
        fadd    v5.4s, v17.4s, v5.4s
        fadd    v6.4s, v18.4s, v6.4s
        stp     q0, q4, [x9, #-32]
        stp     q5, q6, [x9], #64
        b.ne    .LBB3_4
```

15 instructions (16 lanes)

1.066 inst / lane

- Spreads the cost of flow control.
- Hides instruction latency / data dependency.

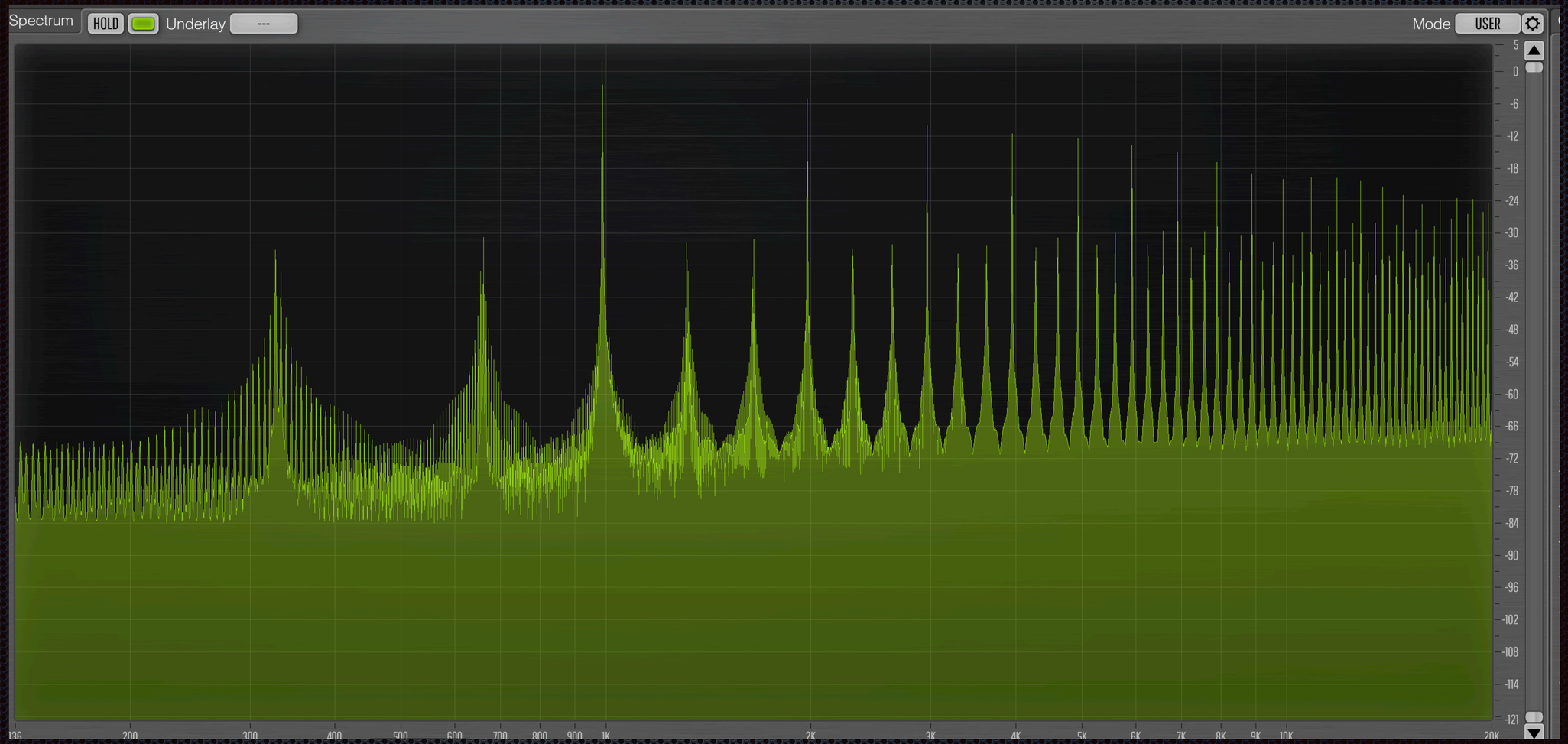
Let’s make an oscillator: Performance

```
Avg runtime :96.088005  Interleave width: 1 (x4) 4      (for iter):262144      Normalised exec time (1 osc, 1smp, pSec) 366.546661     (MOscSamps/s: 2728)     (64vc 88k Est CPU%: 0.206439)
Avg runtime :122.818008 Interleave width: 2 (x4) 8      (for iter):524288      Normalised exec time (1 osc, 1smp, pSec) 234.256760     (MOscSamps/s: 4268)     (64vc 88k Est CPU%: 0.131933)
Avg runtime :141.000000 Interleave width: 4 (x4) 16     (for iter):1048576     Normalised exec time (1 osc, 1smp, pSec) 134.468094     (MOscSamps/s: 7436)     (64vc 88k Est CPU%: 0.0757324)
Avg runtime :167.000015 Interleave width: 8 (x4) 32     (for iter):2097152     Normalised exec time (1 osc, 1smp, pSec) 79.631813      (MOscSamps/s: 12557)    (64vc 88k Est CPU%: 0.0448486)
Avg runtime :181.000015 Interleave width: 10 (x4) 40    (for iter):2621440     Normalised exec time (1 osc, 1smp, pSec) 69.046028      (MOscSamps/s: 14483)    (64vc 88k Est CPU%: 0.0388867)
Avg runtime :206.324005 Interleave width: 12 (x4) 48    (for iter):3145728     Normalised exec time (1 osc, 1smp, pSec) 65.588638      (MOscSamps/s: 15246)    (64vc 88k Est CPU%: 0.0369395)
Avg runtime :279.318024 Interleave width: 16 (x4) 64    (for iter):4194304     Normalised exec time (1 osc, 1smp, pSec) 66.594604      (MOscSamps/s: 15016)    (64vc 88k Est CPU%: 0.0375061)
Avg runtime :900.436035 Interleave width: 32 (x4) 128   (for iter):8388608     Normalised exec time (1 osc, 1smp, pSec) 107.340340     (MOscSamps/s: 9316)     (64vc 88k Est CPU%: 0.0604541)
```

```
Interleave width: 1 (x4) 4      (64vc 88k Est CPU%: 0.206439)
Interleave width: 2 (x4) 8      (64vc 88k Est CPU%: 0.131933)
Interleave width: 4 (x4) 16     (64vc 88k Est CPU%: 0.0757324)
Interleave width: 8 (x4) 32     (64vc 88k Est CPU%: 0.0448486)
Interleave width: 10 (x4) 40    (64vc 88k Est CPU%: 0.0388867)
Interleave width: 12 (x4) 48    (64vc 88k Est CPU%: 0.0369395)
Interleave width: 16 (x4) 64    (64vc 88k Est CPU%: 0.0375061)
Interleave width: 32 (x4) 128   (64vc 88k Est CPU%: 0.0604541)
```

12-way interleaved (48 voices): 5.5x faster than “classic” SIMD, 22x faster than scalar code.

This oscillator sucks.



Why “only” 22x faster?

```
LBB1_4:
        fcmgt   v4.4s, v2.4s, v0.4s
        bsl     v4.16b, v1.16b, v3.16b
        fadd    v0.4s, v4.4s, v0.4s
        str     q0, [x9, x10]
        add     x10, x10, #16
        cmp     x10, #2048
        b.ne    .LBB1_4
```

```
        fcmgt   v4.4s, v2.4s, v0.4s
        bsl     v4.16b, v1.16b, v3.16b
        fadd    v0.4s, v4.4s, v0.4s
        str     q0, [x9, x10]
        add     x10, x10, #16
        cmp     x10, #2048
        b.ne    .LBB1_4
```

```
        fcmgt   v4.4s, v2.4s, v0.4s
        bsl     v4.16b, v1.16b, v3.16b
        fadd    v0.4s, v4.4s, v0.4s
        str     q0, [x9, x10]
        add     x10, x10, #16
        cmp     x10, #2048
        b.ne    .LBB1_4
```

Pipelining - second iteration can begin before first completes.

1x interleave = 1.75 per lane

4x interleave = 1.06 per lane

```
        ldp     q20, q10, [sp, #352]
        subs    x8, x8, #1
        ldp     q25, q27, [sp, #96]
        fcmgt   v14.4s, v20.4s, v0.4s
        ldp     q20, q21, [sp, #288]
        fcmgt   v15.4s, v21.4s, v1.4s
        ldr     q21, [sp, #400]
        fcmgt   v28.4s, v20.4s, v2.4s
        ldr     q20, [sp, #272]
        bsl     v14.16b, v21.16b, v27.16b
        fcmgt   v31.4s, v20.4s, v3.4s
        ldp     q21, q20, [sp, #224]
        fcmgt   v8.4s, v21.4s, v5.4s
        ldr     q21, [sp, #384]
        fcmgt   v20.4s, v20.4s, v4.4s
        bsl     v15.16b, v21.16b, v25.16b
        ldp     q27, q25, [sp, #64]
        fadd    v0.4s, v14.4s, v0.4s
        bsl     v28.16b, v10.16b, v25.16b
        ldp     q25, q21, [sp, #192]
        fcmgt   v21.4s, v21.4s, v6.4s
        fadd    v1.4s, v15.4s, v1.4s
        fcmgt   v10.4s, v25.4s, v7.4s
        ldr     q25, [sp, #336]
        bsl     v31.16b, v25.16b, v27.16b
        ldr     q25, [sp, #176]
        ldr     q27, [sp, #48]
        fadd    v2.4s, v28.4s, v2.4s
        stp     q0, q1, [x9, #-96]
        fcmgt   v14.4s, v25.4s, v16.4s
        ldr     q25, [sp, #320]
        bsl     v20.16b, v25.16b, v27.16b
        ldr     q25, [sp, #160]
        ldr     q27, [sp, #32]
        fadd    v3.4s, v31.4s, v3.4s
```

```
        fcmgt   v15.4s, v25.4s, v17.4s
        ldr     q25, [sp, #256]
        bsl     v8.16b, v25.16b, v27.16b
        ldr     q25, [sp, #144]
        fadd    v4.4s, v20.4s, v4.4s
        mov     v20.16b, v14.16b
        stp     q2, q3, [x9, #-64]
        fcmgt   v28.4s, v25.4s, v18.4s
        ldr     q25, [sp, #16]
        bsl     v21.16b, v11.16b, v25.16b
        ldr     q25, [sp, #128]
        fadd    v5.4s, v8.4s, v5.4s
        mov     v8.16b, v15.16b
        bsl     v20.16b, v22.16b, v29.16b
        fcmgt   v31.4s, v25.4s, v19.4s
        ldr     q25, [sp]
        fadd    v6.4s, v21.4s, v6.4s
        mov     v21.16b, v28.16b
        bsl     v10.16b, v24.16b, v25.16b
        bsl     v8.16b, v9.16b, v30.16b
        fadd    v16.4s, v20.4s, v16.4s
        stp     q4, q5, [x9, #-32]
        mov     v28.16b, v31.16b
        bsl     v21.16b, v26.16b, v12.16b
        fadd    v7.4s, v10.4s, v7.4s
        bsl     v28.16b, v23.16b, v13.16b
        fadd    v17.4s, v8.4s, v17.4s
        fadd    v18.4s, v21.4s, v18.4s
        stp     q6, q7, [x9]
        fadd    v19.4s, v28.4s, v19.4s
        stp     q16, q17, [x9, #32]
        stp     q18, q19, [x9, #64]
        add     x9, x9, #192
        b.ne    .LBB10_2
```

12x = 68 instructions (1.42 per lane)

‘ldp’, ‘stp’ - loads and stores - indicate register spills & refills

Synthetic / artificial case - always profile!

8-wide: 4.5x 1-wide (18x scalar)

12-wide: 5.5x 1-wide (22x scalar)

Improvement 1: triangle

Triangle formula

$$2\{\mathrm{abs}(x) < 0.5 : x\ ,\ \mathrm{sign}(x) - x\}$$

1x absolute
2x compare
2x mask
1x subtract
1x scale

sign function: pick (x >= 0) ? 1 : -1;

```
simd triangle = 2.f * (simd::picklt(simd::fabs(x), 0.5f, x, simd::sign(x)-x));
```

```
Avg runtime :300.944000 Interleave width: 1 (x4) 4      (for iter):524288      Normalised exec time (1 osc, 1smp, pSec) 574.005188     (MOscSamps/s: 1742)     (64vc 88k Est CPU%: 0.32328)
Avg runtime :344.110016 Interleave width: 2 (x4) 8      (for iter):1048576     Normalised exec time (1 osc, 1smp, pSec) 328.168884     (MOscSamps/s: 3047)     (64vc 88k Est CPU%: 0.184825)
Avg runtime :471.224030 Interleave width: 4 (x4) 16     (for iter):2097152     Normalised exec time (1 osc, 1smp, pSec) 224.697128     (MOscSamps/s: 4450)     (64vc 88k Est CPU%: 0.126549)
Avg runtime :841.338013 Interleave width: 8 (x4) 32     (for iter):4194304     Normalised exec time (1 osc, 1smp, pSec) 200.590622     (MOscSamps/s: 4985)     (64vc 88k Est CPU%: 0.112973)
Avg runtime :1047.624023        Interleave width: 10 (x4) 40    (for iter):5242880      Normalised exec time (1 osc, 1smp, pSec) 199.818436     (MOscSamps/s: 5004)     (64vc 88k Est CPU%: 0.112538)
Avg runtime :1257.088013        Interleave width: 12 (x4) 48    (for iter):6291456      Normalised exec time (1 osc, 1smp, pSec) 199.808777     (MOscSamps/s: 5004)     (64vc 88k Est CPU%: 0.112532)
Avg runtime :1809.314087        Interleave width: 16 (x4) 64    (for iter):8388608      Normalised exec time (1 osc, 1smp, pSec) 215.687042     (MOscSamps/s: 4636)     (64vc 88k Est CPU%: 0.121475)
Avg runtime :5050.236328        Interleave width: 32 (x4) 128   (for iter):16777216     Normalised exec time (1 osc, 1smp, pSec) 301.017548     (MOscSamps/s: 3322)     (64vc 88k Est CPU%: 0.169533)
```

At 4-16x interleaving, still ~9x faster than a naive sawtooth implemented in scalar code.

Still not antialiased! But provides a solid foundation for symmetric, band limited polynomials…

Improvement 2: polynomial sine wave

5th order polynomial, half-cycle sine wave

$$y = \frac{\left(x^5 - 10x^3 + 25x\right)}{16}$$

6x multiply (5x?)
1x add
1x subtract

```
simd y = triangle;
simd y2 = y*y;
simd y3 = y2*y;
simd sine_wave = ((y2*y3) - (10.f * y3) + (25.f * y)) * 0.625f;
```

```
Avg runtime :329.464020 Interleave width: 1 (x4) 4      (for iter):524288       Normalised exec time (1 osc, 1smp, pSec) 628.402771     (MOscSamps/s: 1591)     (64vc 88k Est CPU%: 0.353916)
Avg runtime :417.248016 Interleave width: 2 (x4) 8      (for iter):1048576      Normalised exec time (1 osc, 1smp, pSec) 397.918732     (MOscSamps/s: 2513)     (64vc 88k Est CPU%: 0.224108)
Avg runtime :690.966003 Interleave width: 4 (x4) 16     (for iter):2097152      Normalised exec time (1 osc, 1smp, pSec) 329.478271     (MOscSamps/s: 3035)     (64vc 88k Est CPU%: 0.185562)
Avg runtime :1353.282104        Interleave width: 8 (x4) 32     (for iter):4194304      Normalised exec time (1 osc, 1smp, pSec) 322.647583     (MOscSamps/s: 3099)     (64vc 88k Est CPU%: 0.181715)
Avg runtime :1691.318115        Interleave width: 10 (x4) 40    (for iter):5242880      Normalised exec time (1 osc, 1smp, pSec) 322.593323     (MOscSamps/s: 3099)     (64vc 88k Est CPU%: 0.181685)
Avg runtime :2032.918091        Interleave width: 12 (x4) 48    (for iter):6291456      Normalised exec time (1 osc, 1smp, pSec) 323.123657     (MOscSamps/s: 3094)     (64vc 88k Est CPU%: 0.181983)
Avg runtime :2843.722168        Interleave width: 16 (x4) 64    (for iter):8388608      Normalised exec time (1 osc, 1smp, pSec) 338.998108     (MOscSamps/s: 2949)     (64vc 88k Est CPU%: 0.190924)
Avg runtime :7165.370117        Interleave width: 32 (x4) 128   (for iter):16777216     Normalised exec time (1 osc, 1smp, pSec) 427.089355     (MOscSamps/s: 2341)     (64vc 88k Est CPU%: 0.240537)
```

At 4-16x interleaving, still >5.5x faster *than a naive sawtooth* implemented in scalar code.

Total error: **-54dB**

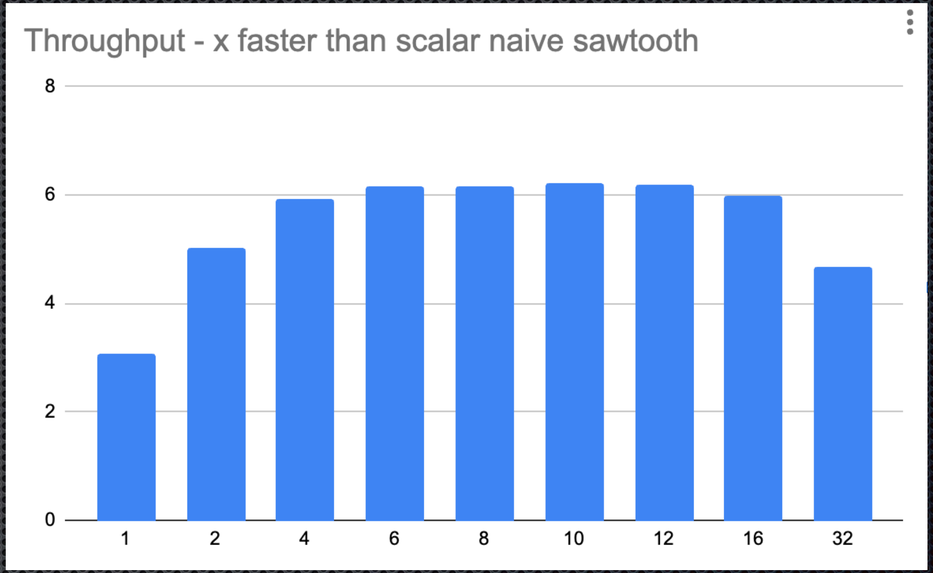
Excluding first four harmonics: **-80dB** (good to Fs / 8)

Improvement 2: polynomial sine wave

Assembly (1x interleave / 4-wide)

```
LBB10_2:
        fcmge   v18.4s, v5.4s, v1.4s
        subs    x8, x8, #1
        bsl     v18.16b, v0.16b, v6.16b
        fadd    v1.4s, v18.4s, v1.4s
        fcmlt   v18.4s, v1.4s, #0.0
        fabs    v19.4s, v1.4s
        bsl     v18.16b, v3.16b, v2.16b
        fcmge   v19.4s, v19.4s, v4.4s
        fsub    v18.4s, v18.4s, v1.4s
        bif     v18.16b, v1.16b, v19.16b
        fadd    v19.4s, v18.4s, v18.4s
        fmul    v18.4s, v18.4s, v17.4s
        fmul    v20.4s, v19.4s, v19.4s
        fmul    v19.4s, v20.4s, v19.4s
        fadd    v20.4s, v20.4s, v7.4s
        fmla    v18.4s, v19.4s, v20.4s
        fmul    v18.4s, v18.4s, v16.4s
        str     q18, [x0], #16
        b.ne    .LBB10_2
```

Assembly (1x interleave / 4-wide)



Improvement 3: Quadrature oscillator

Computationally simple technique producing a pure sine & cosine wave using rotation

```
for (int i = 0; i < smps; i++)
{
    simd newX = cosAngle * x - sinAngle * y;
    simd newY = sinAngle * x + cosAngle * y;
    x = newX;
    y = newY;
    buffer[i] = y;
}
```

- ✅ Clean waveforms.
- ✅ Algorithm is inherently branchless.
- ✅ sin & “free” cos wave.
- ❌ Coefficient calculation is costly (needs trig or close appro
- ❌ Iterative - potential for stability problems.

```
Avg runtime :657.864014 Interleave width: 1 (x4) 4      (for iter):524288       Normalised exec time (1 osc, 1smp, pSec) 1254.776123    (MOscSamps/s: 796)      (64vc 88k Est CPU%: 0.70669)
Avg runtime :810.534058 Interleave width: 2 (x4) 8      (for iter):1048576      Normalised exec time (1 osc, 1smp, pSec) 772.985535     (MOscSamps/s: 1293)     (64vc 88k Est CPU%: 0.435345)
Avg runtime :1335.934082        Interleave width: 4 (x4) 16     (for iter):2097152      Normalised exec time (1 osc, 1smp, pSec) 637.023010     (MOscSamps/s: 1569)     (64vc 88k Est CPU%: 0.358771)
Avg runtime :1942.556152        Interleave width: 6 (x4) 24     (for iter):3145728      Normalised exec time (1 osc, 1smp, pSec) 617.521973     (MOscSamps/s: 1619)     (64vc 88k Est CPU%: 0.347788)
Avg runtime :2405.724121        Interleave width: 8 (x4) 32     (for iter):4194304      Normalised exec time (1 osc, 1smp, pSec) 573.569336     (MOscSamps/s: 1743)     (64vc 88k Est CPU%: 0.323034)
Avg runtime :3089.466064        Interleave width: 10 (x4) 40    (for iter):5242880      Normalised exec time (1 osc, 1smp, pSec) 589.268921     (MOscSamps/s: 1697)     (64vc 88k Est CPU%: 0.331876)
Avg runtime :3643.606201        Interleave width: 12 (x4) 48    (for iter):6291456      Normalised exec time (1 osc, 1smp, pSec) 579.135620     (MOscSamps/s: 1726)     (64vc 88k Est CPU%: 0.326169)
Avg runtime :5089.434082        Interleave width: 16 (x4) 64    (for iter):8388608      Normalised exec time (1 osc, 1smp, pSec) 606.707825     (MOscSamps/s: 1648)     (64vc 88k Est CPU%: 0.341698)
Avg runtime :12212.930664       Interleave width: 32 (x4) 128   (for iter):16777216     Normalised exec time (1 osc, 1smp, pSec) 727.947388     (MOscSamps/s: 1373)     (64vc 88k Est CPU%: 0.40998)
```

Improvement 3: Quadrature oscillator

```
for (int i = 0; i < smps; i++)
{
    simd newX = cosAngle * x - sinAngle * y;
    simd newY = sinAngle * x + cosAngle * y;
    x = newX;
    y = newY;
    buffer[i] = y;
}
```

```
.LBB10_2:
        fmul    v5.4s, v0.4s, v2.4s
        subs    x8, x8, #1
        fmls    v5.4s, v4.4s, v3.4s
        fmul    v4.4s, v4.4s, v2.4s
        fmla    v4.4s, v0.4s, v3.4s
        mov     v0.16b, v5.16b
        str     q4, [x0], #16
        b.ne    .LBB10_2
```

4 cycles after ‘fmul’.
4 cycles until result.

Result to ‘v0’ for next iteration

	AX			AY			BX			BY	
R04	R03	R00	R00	R01	R02	R09	R08	R05	R05	R06	R07
	CX			CY			DX			DY	
R14	R13	R10	R10	R11	R12	R19	R18	R15	R15	R16	R17
	EX			EY			FX			FY	
R24	R23	R20	R20	R21	R22	R29	R28	R25	R25	R26	R27
	GX			GY			HX			HY	
R34	R33	R30	R30	R31	R32	R39	R38	R35	R35	R36	R37

Two halves (x, y) execute independently on adjacent units

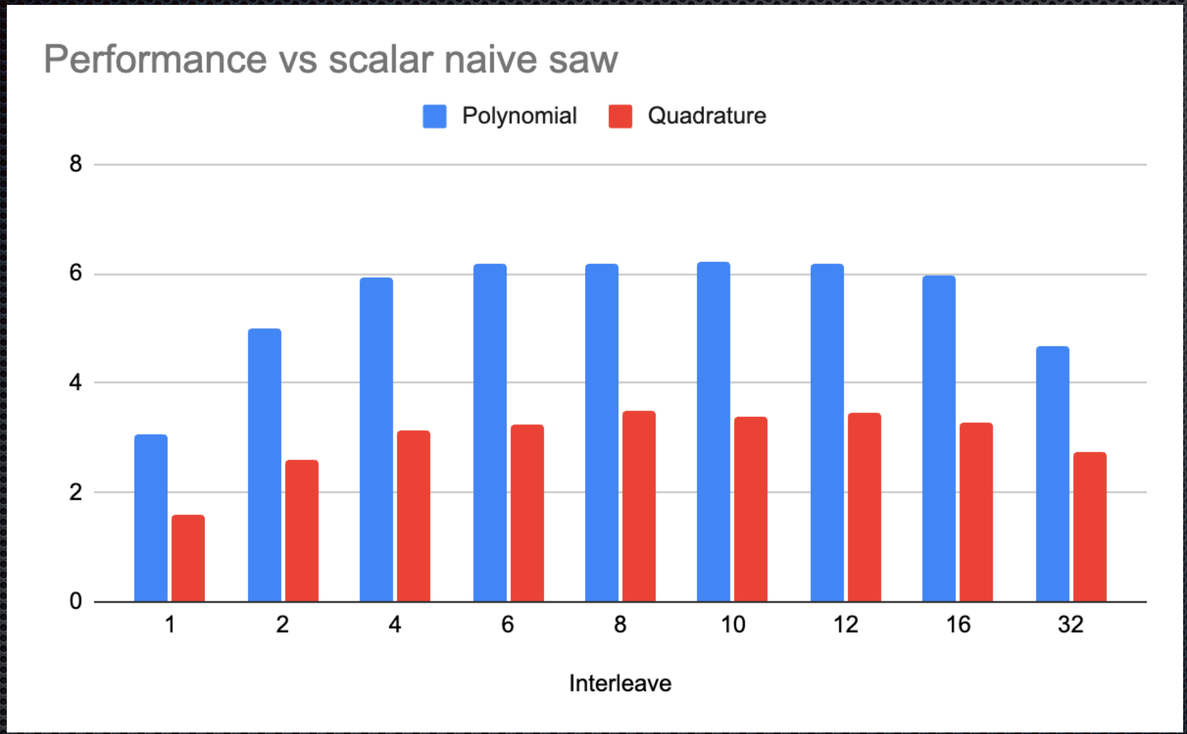
Multiply-accumulates (FMLA/FLMS) wait 4 cycles for FMULs to complete.

Next iteration must wait 4 cycles for multiply-accumulates to complete.

Five registers required per lane

… but only 32 available!

Results and comparison



✅ Polynomial

Performance

Numerical stability

Coefficients

✅ Quadrature

Fidelity

Simplicity

Free cos wave!

Improvement three: PolyBLEP

Polynomial **B**and**l**imited St**ep**

“Generate a naive sawtooth, then correct the step-function to reduce | remove aliasing while preserving harmonics.”

Use a polynomial function to approximate a band-limited step.

Sawtooth, triangle, square, PWM, hardsync …

Improvement three: PolyBLEP

Polynomial **B**and**l**imited St**ep**

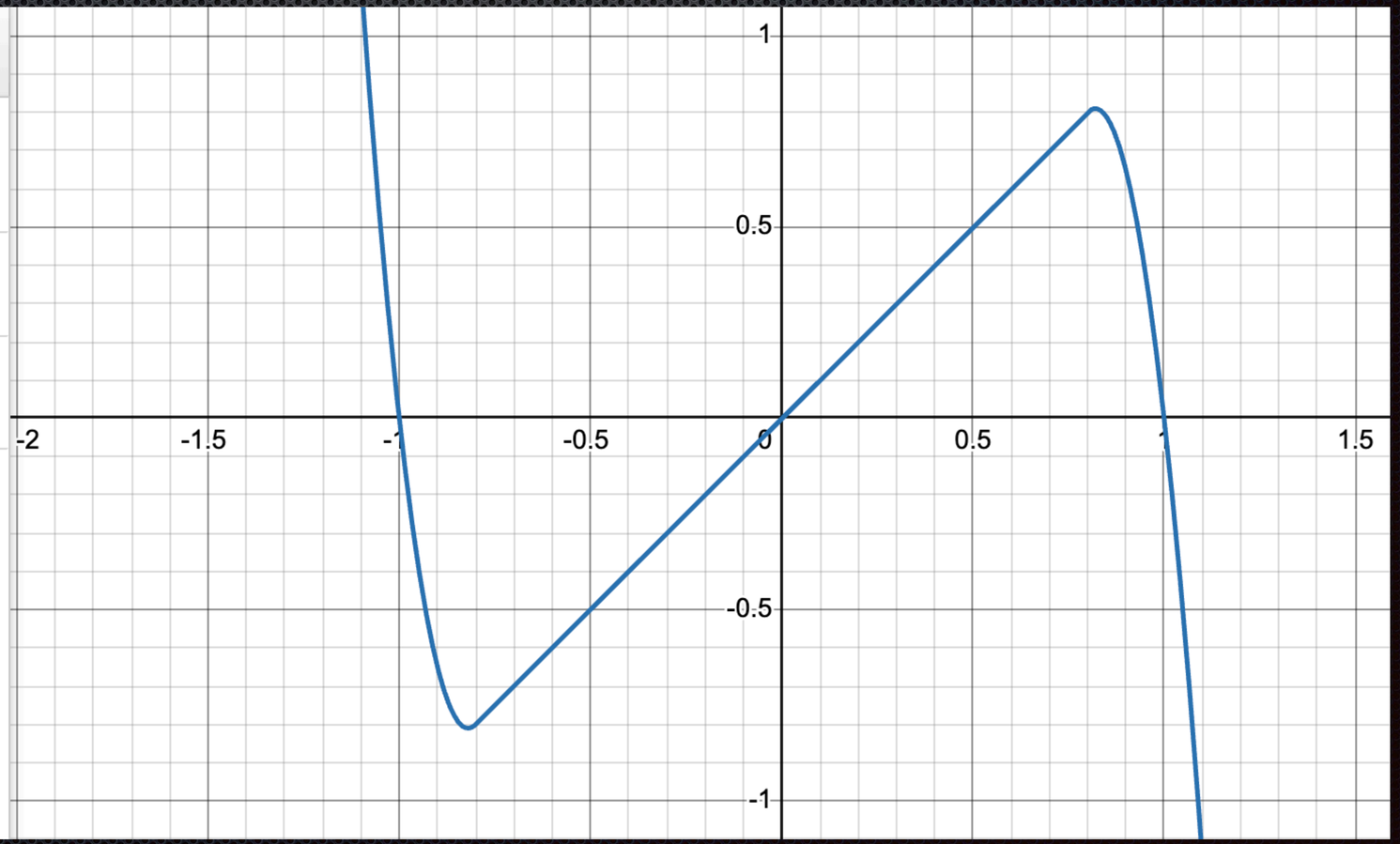
"Lookahead-free" using windowed x^2

1 $a = \left(\left(\max\left(|x|, 0.8\right) - 0.8\right)\right) \cdot 5$

2 $b = x$

3 $c = b - (a \cdot a) \cdot \mathrm{sign}(b)$

4



Improvement three: PolyBLEP

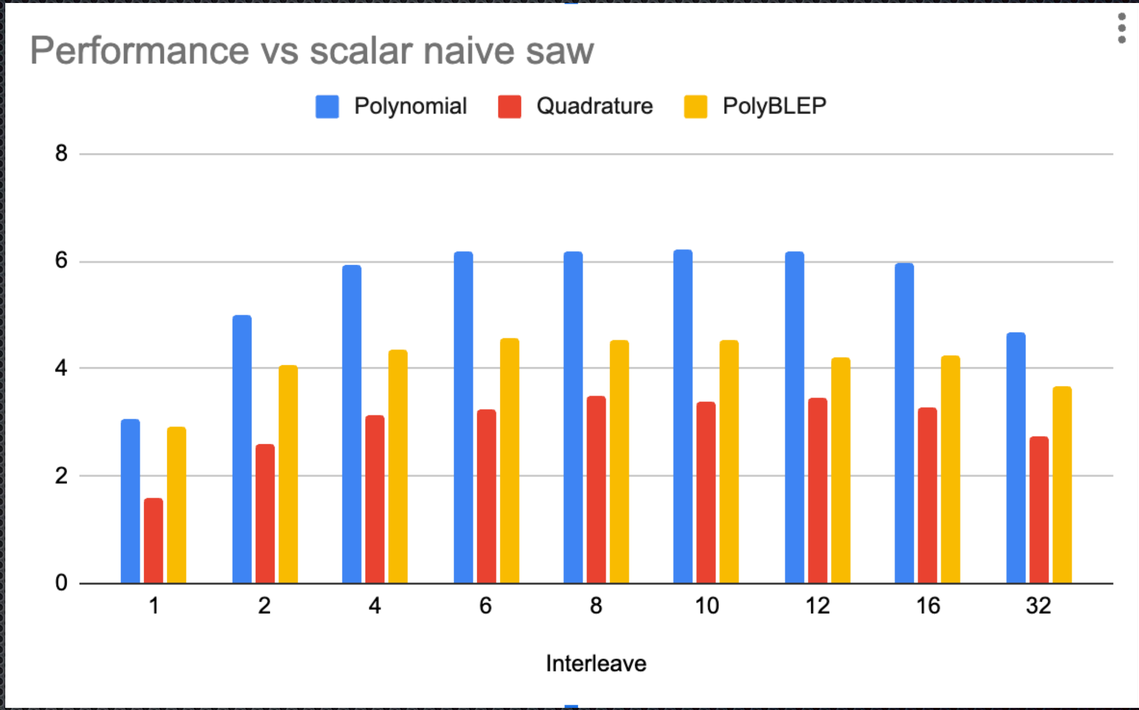
Polynomial **B**and**l**imited St**ep**

```
for (int i = 0; i < smps; i++)
{
    ph += simd::pickgt(ph, rsp, rst, phi);              // Phase increment: (ph > rsp) ? rst : phi;

    simd saw = ph *2;                                   // Raw sawtooth scaled -1..1
    simd sqr = simd::pickgt(ph, pwm, 1.f, -1.f);        // Raw squarewv scaled -1..1

    simd window_hilo = window_size_inverse * simd::fmax(0.f, (simd::fabs(ph) - window_size_one_minus));       // Window around +-1
    simd window_hilo_q = window_hilo * window_hilo * simd::sign(saw);  // x^2 window, positive for ph > 0.5, negative for ph < 0.5

    simd window_lohi = window_size_inverse * simd::fmax(0.f, (simd::fabs(ph) - window_size_one_minus));       // Window around +-1
    simd window_lohi_q = window_lohi * window_lohi * simd::sign(sqr);  // x^2 window, positive for sqr > 0, negative for sqr < 1

    buffer[i] = sqr_level * (sqr - window_hilo_q - window_lohi_q) + saw_level * (saw - window_hilo_q);
}
```

Improvement three: PolyBLEP

Polynomial **B**and**l**imited St**ep**

```
.LBB10_2:
        fcmge   v20.4s, v5.4s, v1.4s
        subs    x8, x8, #1
        bsl     v20.16b, v0.16b, v6.16b
        fadd    v1.4s, v20.4s, v1.4s
        fabs    v20.4s, v1.4s
        fadd    v21.4s, v1.4s, v1.4s
        fcmge   v22.4s, v4.4s, v1.4s
        fadd    v20.4s, v17.4s, v20.4s
        fcmlt   v23.4s, v21.4s, #0.0
        mov     v25.16b, v22.16b
        bsl     v25.16b, v3.16b, v2.16b
        fmax    v20.4s, v20.4s, v18.4s
        fmul    v20.4s, v20.4s, v19.4s
        fmul    v20.4s, v20.4s, v20.4s
        fneg    v24.4s, v20.4s
        bsl     v23.16b, v24.16b, v20.16b
        bif     v20.16b, v24.16b, v22.16b
        fadd    v20.4s, v20.4s, v25.4s
        fsub    v21.4s, v21.4s, v23.4s
        fsub    v20.4s, v20.4s, v23.4s
        fmul    v21.4s, v21.4s, v16.4s
        fmla    v21.4s, v20.4s, v7.4s
        str     q21, [x0], #16
        b.ne    .LBB10_2
```

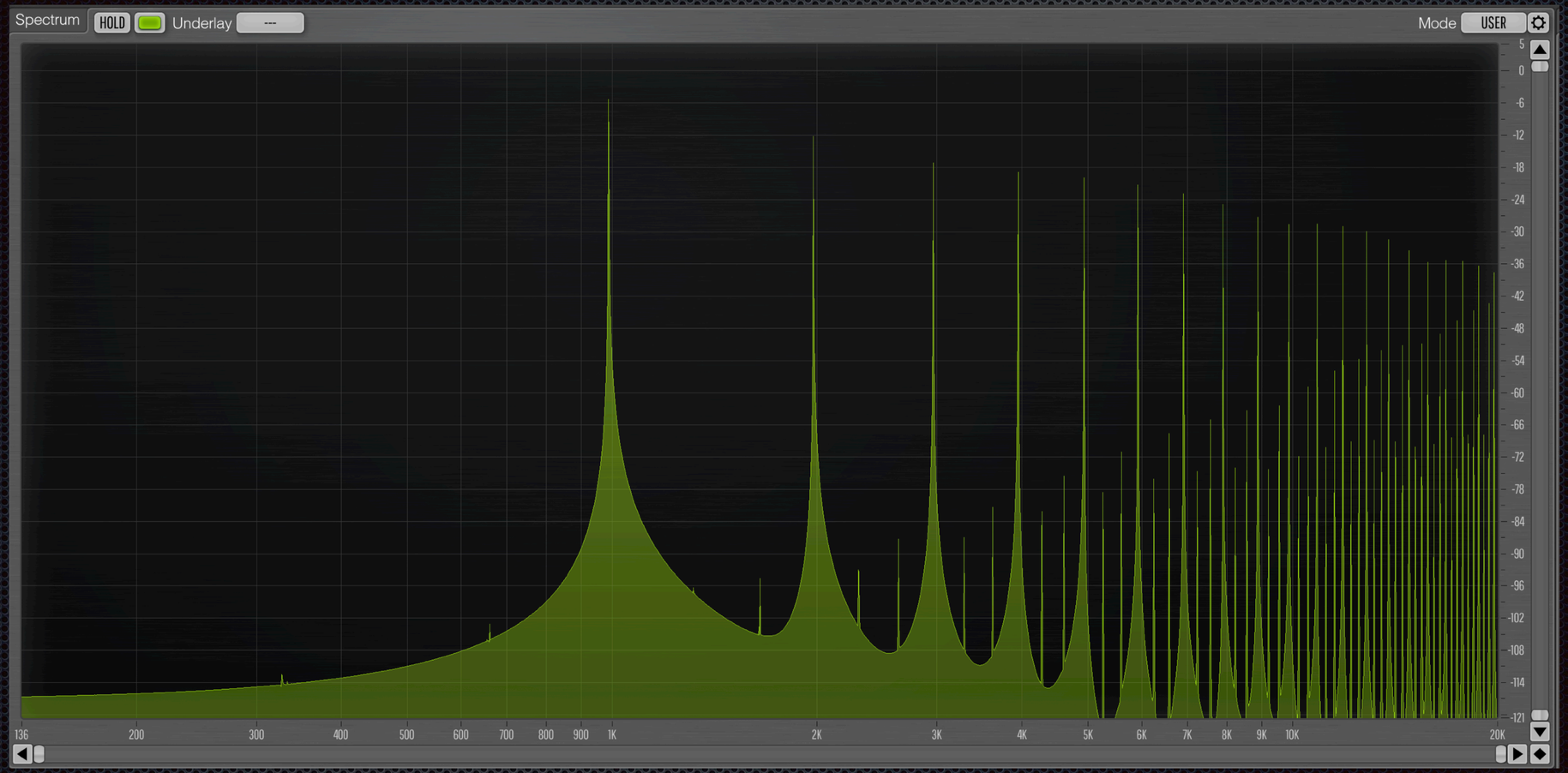



```
=== Test op 3 (PolyBLEP saw/sqr): ===
Avg runtime :359.624023 Interleave width: 1 (x4) 4      (for iter):524288       Normalised exec time (1 osc, 1smp, pSec) 685.928406     (MOscSamps/s: 1457)     (64vc 88k Est CPU%: 0.386315)
Avg runtime :517.182007 Interleave width: 2 (x4) 8      (for iter):1048576      Normalised exec time (1 osc, 1smp, pSec) 493.223206     (MOscSamps/s: 2027)     (64vc 88k Est CPU%: 0.277783)
Avg runtime :964.546021 Interleave width: 4 (x4) 16     (for iter):2097152      Normalised exec time (1 osc, 1smp, pSec) 459.931396     (MOscSamps/s: 2174)     (64vc 88k Est CPU%: 0.259033)
Avg runtime :1381.078125        Interleave width: 6 (x4) 24     (for iter):3145728      Normalised exec time (1 osc, 1smp, pSec) 439.032928     (MOscSamps/s: 2277)     (64vc 88k Est CPU%: 0.247263)
Avg runtime :1848.480103        Interleave width: 8 (x4) 32     (for iter):4194304      Normalised exec time (1 osc, 1smp, pSec) 440.712006     (MOscSamps/s: 2269)     (64vc 88k Est CPU%: 0.248209)
Avg runtime :2311.291992        Interleave width: 10 (x4) 40    (for iter):5242880      Normalised exec time (1 osc, 1smp, pSec) 440.843994     (MOscSamps/s: 2268)     (64vc 88k Est CPU%: 0.248283)
Avg runtime :2985.920166        Interleave width: 12 (x4) 48    (for iter):6291456      Normalised exec time (1 osc, 1smp, pSec) 474.599274     (MOscSamps/s: 2107)     (64vc 88k Est CPU%: 0.267294)
Avg runtime :3949.286133        Interleave width: 16 (x4) 64    (for iter):8388608      Normalised exec time (1 osc, 1smp, pSec) 470.791595     (MOscSamps/s: 2124)     (64vc 88k Est CPU%: 0.26515)
Avg runtime :9166.251953        Interleave width: 32 (x4) 128   (for iter):16777216     Normalised exec time (1 osc, 1smp, pSec) 546.351257     (MOscSamps/s: 1830)     (64vc 88k Est CPU%: 0.307705)
```

4-5x antialiased saw & square for the price of one naive sawtooth

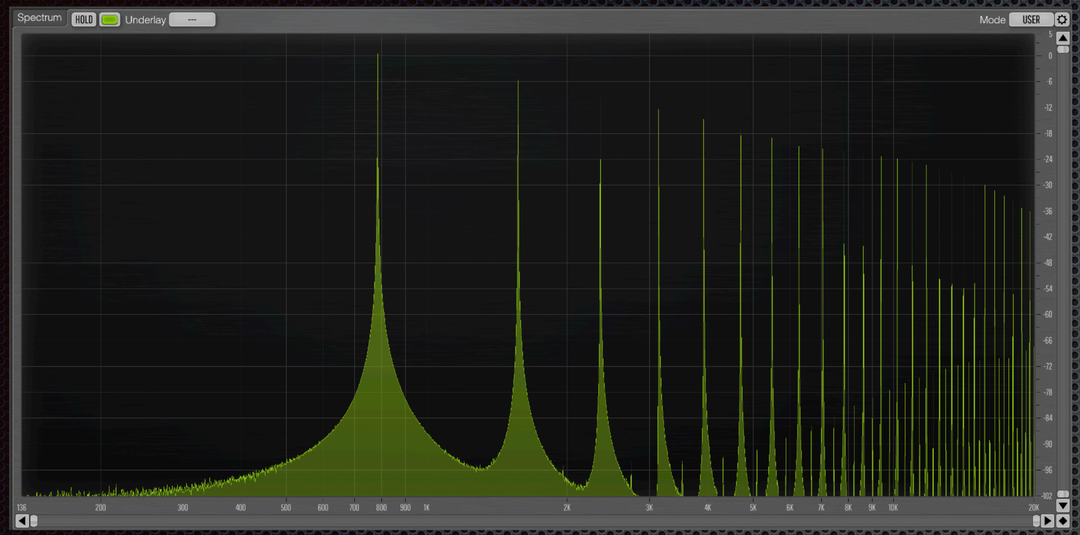
Improvement three: PolyBLEP

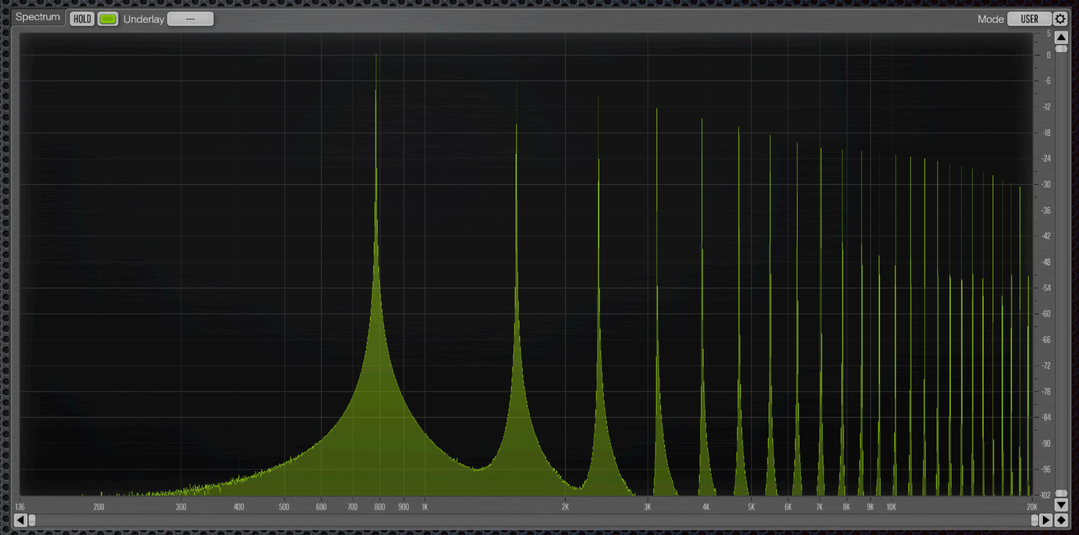
Crude window - unacceptable roll-off and aliasing at 1x



Better polynomials may be available!

Improvement three: PolyBLEP





2x: much better than naive.

-66dB > 10kHz 😿

-78dB <= 10kHz 😷

4x: approaching perfection

-78dB > 10kHz

-90dB <= 10kHz

Clean sawtooth & pulse at 4x for the price of a naive sawtooth at 1x

Better polynomials may be available…

Results and conclusion

Branch-free & parallel code offers potentially significant efficiency gains.

Visible register file limits maximum efficiency (SVE2 to the rescue?)

Beware compilers with good intentions…

Profile, profile, profile!

Thank you.

Thanks to:

Dougall Johnson	(M1 instruction timings)	https://dougallj.github.io/applecpu/firestorm.html
Matt Godbolt	(Compiler Explorer)	https://godbolt.org/
Cardyak	(M1 architectural diagrams)	https://x.com/cardyak
Desmos	(Graph Plotter)	https://www.desmos.com/calculator
Voxengo	(SPAN)	https://www.voxengo.com