OPTIMIZED PROGRAMMING FOR REAL-TIME APPLICATIONS AND BEYOND

#1: INTRODUCTION TO SOFTWARE OPTIMIZATION

Stefano D'Angelo

Università di Udine

4th June 2024

WHAT I DO FOR A LIVING

(a) (b)

Figure 1.1. (a) Photograph of the original Minimoog Voyager synthesizer (image in the public domain) and (b) screenshot of the user interface of the Bristol Moog Voyager MIDI-controlled VA emulator[1] (image copyright by Nick Copeland, used with permission).

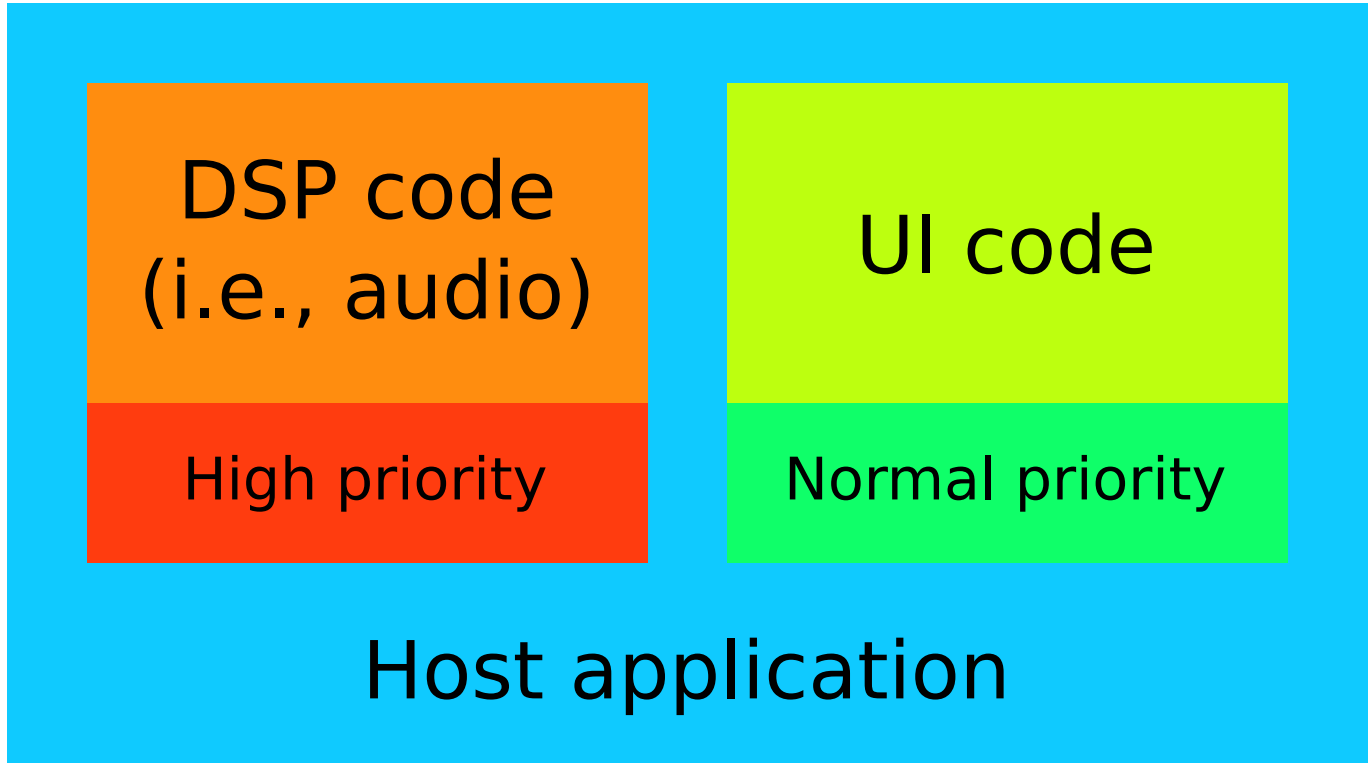
From my doctoral thesis:
S. D'Angelo, "Virtual Analog Modeling of Nonlinear Musical Circuits", Aalto University, Espoo, Finland, November 2014.

DIGITAL AUDIO WORKSTATIONS

Ableton Live

STUDIOS

HOW VIRTUAL GEAR WORKS



AUDIO DSP CODE

- Usually 32-bit floating point
- Processes at least 44100 audio samples per channel per second
- $\longrightarrow$ typically MFLOPS range
- Should use ***as little CPU and memory as possible***
- Should be ***real-time safe***
- Should be ***low latency***
- The last three are **different concepts**

LOW CPU AND MEMORY

- Many potential metrics for CPU usage, e.g.,
  - Number of instructions
  - Code size
  - Throughput or latency
  - Data dependencies
- Same goes for memory, e.g..,
  - Total amount used (e.g., max at any time)
  - Locality
  - Number and pattern of reads/writes
- Objective: run fast both in isolation and not

CPU VS MEM EXAMPLE: TAKE #1

Let's compute the population function (a.k.a. Hamming weight) of a 32-bit unsigned integer number, i.e., the number of 1s in its binary representation.

```
int count_ones(uint32_t x) {
  int result = 0;
  while (x != 0) {
    x = x & (x - 1);
    result++;
  }
  return result;
}
```

Operation count: up to 32 cmp/jmp, and, sub, inc

Memory usage: 2 values, probably CPU registers

CPU VS MEM EXAMPLE: TAKE #2

Precompute and store in a lookup table

```
int count_ones(uint32_t x) {
  static char lut[256] = { 0, 1, 1, 2, 1, 2, 2, 3, ... };
  union {
    uint32_t x;
    char b[4];
  } v = x;
  return lut[v.b[0]] + lut[v.b[1]]
    + lut[v.b[2]] + lut[v.b[3]];
}
```

Operations count: 3 add + perhaps a cast and some indirect addressing

Memory usage: 256 bytes (LUT) + maybe a couple of CPU registers

CPU VS MEM EXAMPLE: VERDICT

- Often metrics are conflicting
- No clear winner, best depends on the context
- You can almost always imagine custom approaches that work better in a specific context
- Not the same as space-time or time-memory tradeoff!
- (learn to) Benchmark!

REAL-TIME CONSTRAINTS

Real-time programs must guarantee response within specified time constraints, often referred to as "deadlines".

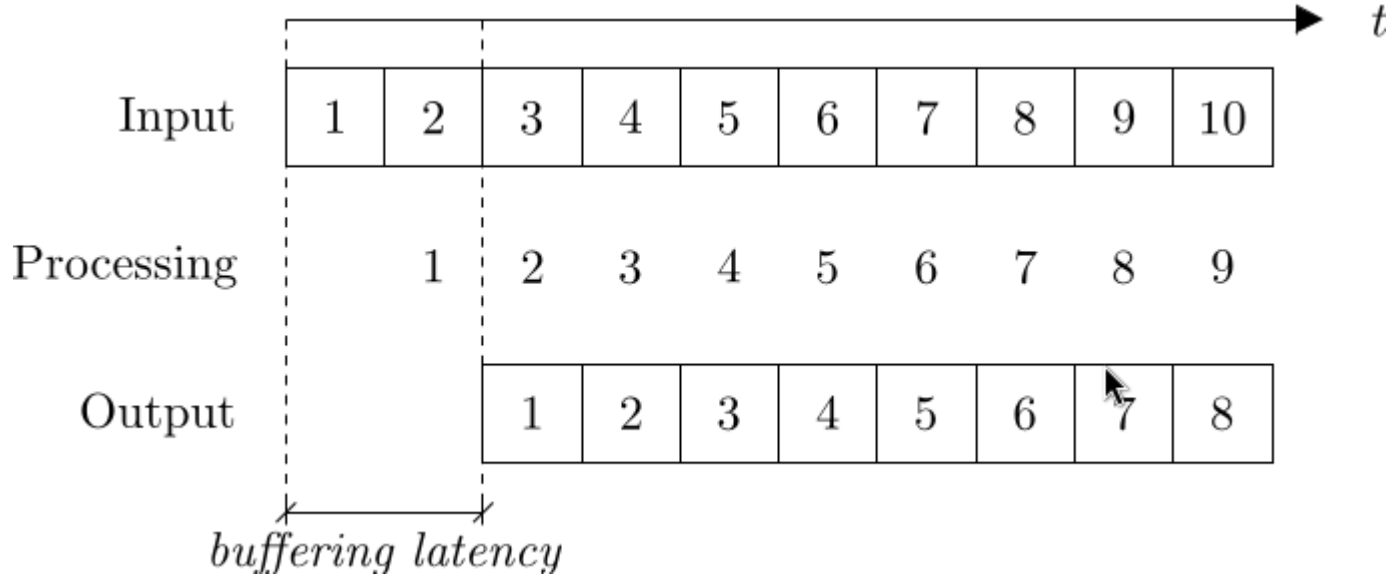
- Hard real-time: guranatee is critical (e.g., medical devices, engine control systems)
- Soft real-time: if not met degradation of performance (e.g., audio, video games)

THE 4 REAL-TIME AUDIO COMMANDMENTS

1. Thou shalt process audio in a time < duration of I/O signals
2. Honour the $O(N)$ algorithm limit, where N is the number of I/O samples
3. Thou shalt never block, including `malloc()`, `rand()`, disk read, network, etc.
4. Thou shalt not try to read the future

COMMANDMENT #1

Thou shalt process audio in a time < duration of I/O signals



(audio is typically processed in buffers corresponding to a given time duration)

COMMANDMENT #2

Honour the $O(N)$ algorithm limit, where N is the number of I/O samples

- Bounded "processing rate" (time to process over I/O duration)
- Hence time to process cannot depend more than linearly on I/O duration
- Sort of follows from commandment #1
- Small sins might only lead you to purgatory (e.g., FFT)

COMMANDMENT #2 IN PRACTICE

OK

```
void process(float *in, float *out, int
  for (int i = 0; i < n_samples; i++)
    float x = in[i];
    float a = ...;
    float b = ...;
    float c = ...;
    ...
    out[i] = ...;
  }
}
```

KO

```
void process(float *in, float *out, int
  for (int i = 0; i < n_samples; i++)
    float x = in[i];
    float a = ...;
    float b = ...;
    float c = ...;
    ...
    for (int j = 0; j < n_samples; j++
      ...
    }
    ...
    out[i] = ...;
  }
}
```

COMMANDMENT #3

Thou shalt never block, including `malloc()`, `rand()`, *disk read, network, etc.*

- Blocking operations are *non-deterministic*, their duration depends on external factors (potentially indefinite wait, priority inversion, …)
- OK to perform them in non-RT threads/processes and perform non-blocking synchronization/ communication
- But must know what to do/how to continue without results $\rightarrow$ asynchronous design

EXAMPLE

```
typedef struct {
  sync s;        // some sync mechanism
  data_in in;    // input data for RT thread, written by non-RT thread
  data_out out; // output data for RT thread, read by non-RT thread
} data;

void process(data *external, data *local, float *in, float *out, int n_samples)
  // possibly copy external->in to local->in
  data_update_in(external, local); // might do nothing but will not block

  // only use local here
  for (int i = 0; i < n_samples; i++)
    ...

  // possibly copy local->out to external->out
  data_update_out(local, external); // might do nothing but will not block
}
```

EXAMPLE SYNC: TRYLOCK

- Used in the context of **mut**ual **ex**clusion (mutex) of critical sections in concurrent code
- `lock()`: lock and return if free, otherwise block until lock
- `trylock()`: lock and return if free, otherwise don't lock and return
- Non-RT thread: `lock()`, read/write on shared memory, `unlock()`
- RT thread: `if (trylock()) {` read/write on shared memory `} unlock()`

ALTERNATIVE: LOCK-FREE QUEUES

- FIFO message queues of fixed predetermined size (also messages of fixed size and preallocated)
- Messages are *atomically* added/removed from/to queue
- RT and non-RT threads read/write such messages
- Hard to implement primitives, arguably easier to use or understand
- Queues need to be big enough, otherwise messages get lost or you need to supplement

EXAMPLE USING LOCK-FREE QUEUES

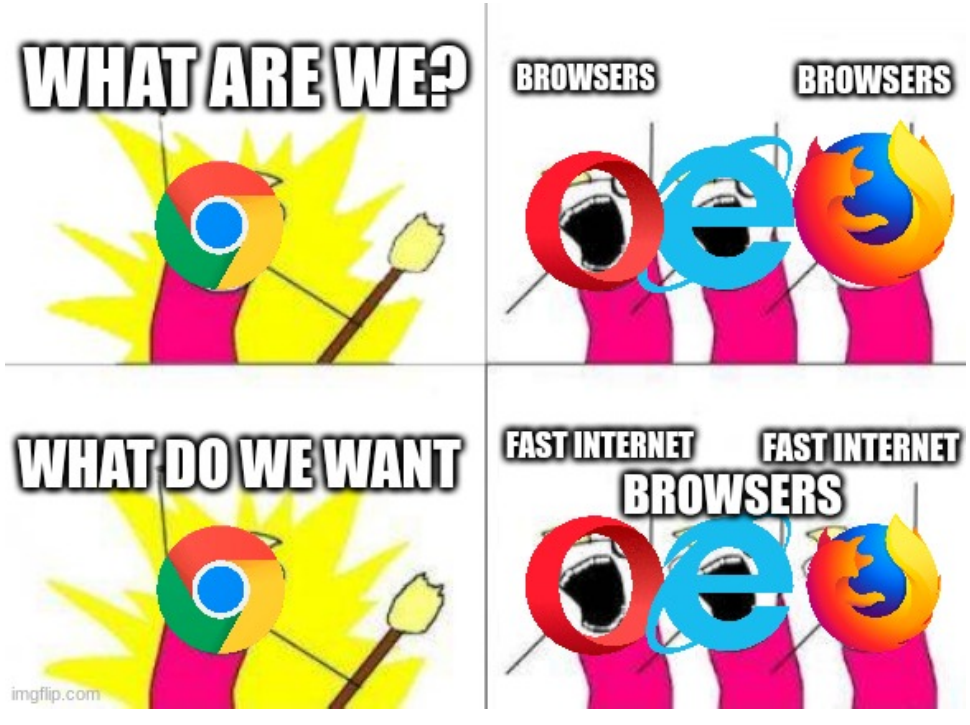
```
void process(queue *qin, queue *qout, data *data, float *in, float *out, int n_
  msg *msg;
  while ((msg = queue_read(qin)) != NULL)
    ... // update data using msg from non-RT thread

  // can directly read/write data here
  for (int i = 0; i < n_samples; i++)
    ...

  // send messages to non-RT thread
  if (data_to_write) {
    queue_write(qout, ...);
    ...
  }
}
```

COMMANDMENT #4

Thou shalt not try to read the future

- Even though audio is processed in buffers, only "present" and "past" input can be used (unless you're really weird)
- Otherwise what would you do close to buffer end?
- This makes your algorithm **causal**
- But many audio filters need future input
- You can artificially insert delay to simulate reading from the future while retaining causality
- You are now introducting *algorithmic latency*

LATENCY



- Undesirable time delay between input and corresponding output
- Low latency allows for live usage
- Algorithmic latency is only part of the total latency

LATENCY IN AUDIO

- 10 ms max RTT rule (~17 m echo distance, ~340 m/s speed of sound)
- Sources of latency:
  - sound card: A/D and D/A filtering, buffering
  - OS audio stack: resampling, buffering
  - host app: resampling, buffering
  - DSP algorithm: resampling, buffering, causality
- Requirements can be slightly relaxed for sound synthesis
- Totally different requirements from, e.g., video

PROGRAMMING LANGUAGES

- RT is all about time predictability (*determinism*)
- Good languages do not introduce or allow you to strictly control sources of nondeterminism, like:
  - Unpredictable stalls, e.g., due to automatic memory management
  - Execution of non-native code (interpreted or bytecode) and sometimes JIT compilation
  - Large dependencies (including the eventual VM itself), which contribute opacity
- In practice you want to be relatively close to metal

GARBAGE COLLECTION

- Characterizes languages featuring automatic memory management
- From time to time memory that was (automatically?) allocated is automatically freed
- In some languages object destruction, which might be blocking, is performed as well
- Most often you can't control when GC takes place
- In some cases it blocks the whole execution, other times it is performed concurrently
- RT-safe GC exists — does anybody use it?

VM AND JIT

- Typical computing platforms (PCs, mobile, high-end embedded) are already not strictly deterministic platforms
  - CPU and memory status when execution of our code begins is unpredictable
  - OS will handle multitasking, interrupts, peripherals, etc.
- VMs that interpret ASTs or bytecode inject even more jitter, not to mention versioning
- Usually JIT can fire up at any time and has overhead

MATLAB, PYTHON, JAVA, JS, ETC.

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

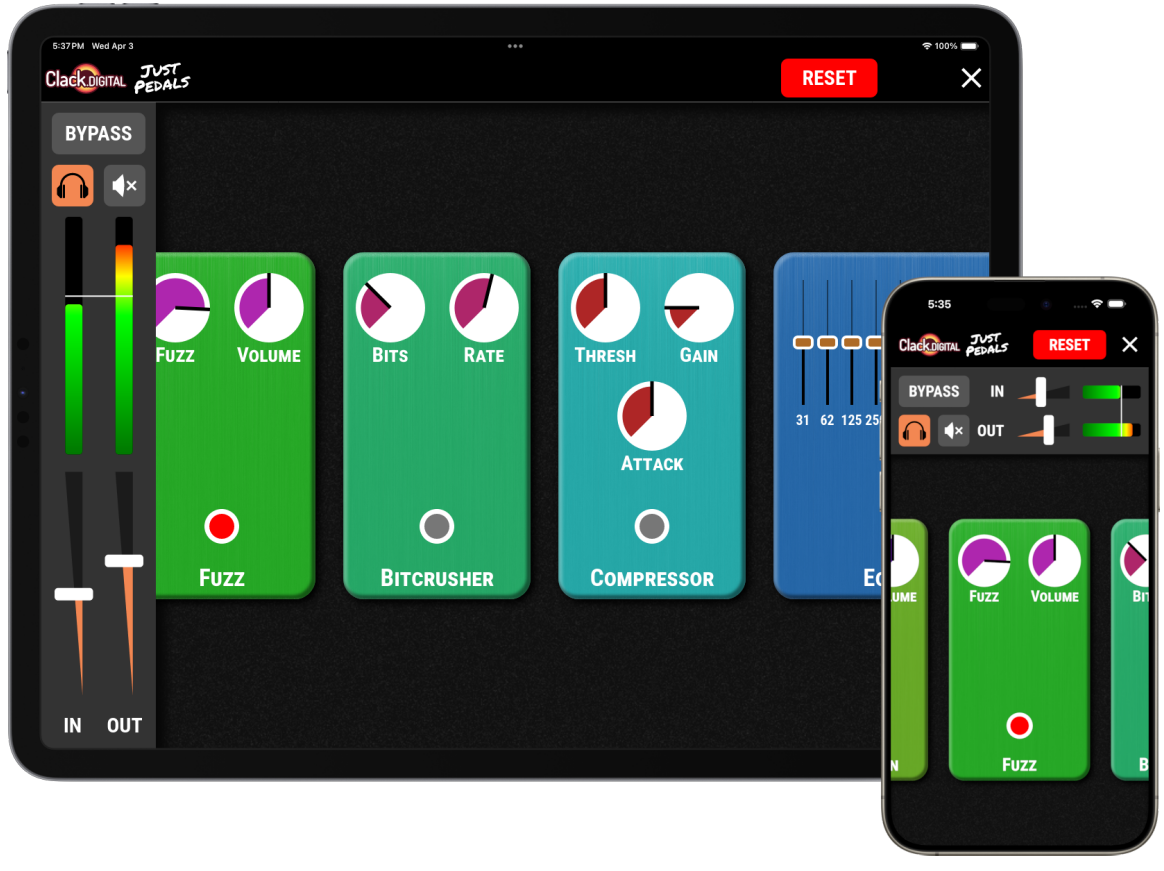
C, C++, ASSEMBLY, D, RUST, ETC.

- Typically require more coding effort
- Can achieve "quasi-deterministic" performance
- Compiler output is native code
- Better for benchmarking
- Real-time constraints can be satisfied
- Actually used in real products

YOU CAN GO HYBRID

- Low-level languages for the RT part, high-level ones for application logic
- Examples:
  - Unity (video game engine) is written in C++ and uses C# for scripting
  - Blender (3D graphics) is written in C/C++ but has Python APIs
  - SuperCollider (live audio) is written in C++ and scripted in sclang

SHAMELESS SELF-PROMOTION



Clack.DIGITAL Just Pedals

Audio: C only, UI: HTML/CSS/JS + some Swift

TRANSPILED LANGUAGES

- Some non-RT languages can be transpiled to RT-capable ones or to machine code
  - I'm no expert, my guess is that under certain conditions it could work but not in general
- Domain-specific languages are often specifically designed for transpilation. Examples (audio):
  - Ciaramella: https://ciaramella.dev/
  - FAUST: https://faust.grame.fr/
  - Cmajor: https://cmajor.dev/

  Plus, they are easier to use but have limits

OTHER AD-HOC SOLUTIONS

- Often require specific execution environments
- Not always embeddable
- Audio examples:
  - Reaktor: https://www.native-instruments.com/en/products/komplete/synths/reaktor-6/
  - Pure Data: https://puredata.info/
  - Max/gen~: https://cycling74.com/products/max

FOOD FOR THOUGHT

- Given RT contraints, plausible algorithms tend to "simply" implement state machines
- Modeled and discretized physical systems are state machines as well
- Math behind RT code, if any, tends to be crucial
- As for all optimization, algorithm design is more important than implementation

(FINITE?) STATE MACHINES

	State A	State B	State C
Input X	…	…	…
Input Y	…	**State C**	…
Input Z	…	…	…

- Input, state, and output can be arbitrarily complex
- Output can be a combinatorial function of input and state
- In automata theory this is enough to parse regular languages, not enough for context-free languages

STATE-SPACE SYSTEMS, CONTINUOUS-TIME DOMAIN

$$\begin{aligned}\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} &= \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} + \mathbf{C}\mathbf{i},\\ \mathbf{i} &= f(\mathbf{v}),\\ \mathbf{v} &= \mathbf{D}\mathbf{x} + \mathbf{E}\mathbf{u} + \mathbf{F}\mathbf{i},\\ \mathbf{y} &= \mathbf{L}\mathbf{x} + \mathbf{M}\mathbf{u} + \mathbf{N}\mathbf{i},\end{aligned}$$

$\mathbf{x}$ is state, $\mathbf{y}$ is output, $\mathbf{u}$ is input

STATE-SPACE SYSTEMS, DISCRETE-TIME DOMAIN

$$
\begin{aligned}
\mathbf{x}[n] &= \mathbf{A}\mathbf{x}[n-1] + \mathbf{B}\mathbf{u}[n] + \mathbf{C}\mathbf{i}[n], \\
\mathbf{i}[n] &= f(\mathbf{v}[n]), \\
\mathbf{v}[n] &= \mathbf{D}\mathbf{x}[n] + \mathbf{E}\mathbf{u}[n] + \mathbf{F}\mathbf{i}[n], \\
\mathbf{y}[n] &= \mathbf{L}\mathbf{x}[n] + \mathbf{M}\mathbf{u}[n] + \mathbf{N}\mathbf{i}[n],
\end{aligned}
$$

$\mathbf{x}[n]$ is state, $\mathbf{y}[n]$ is output, $\mathbf{u}[n]$ is input

Structurally similar to (F)SMs

OPTIMIZED PROGRAMMING FOR REAL-TIME APPLICATIONS AND BEYOND

#2: DESIGN AND ANALYSIS

Stefano D'Angelo

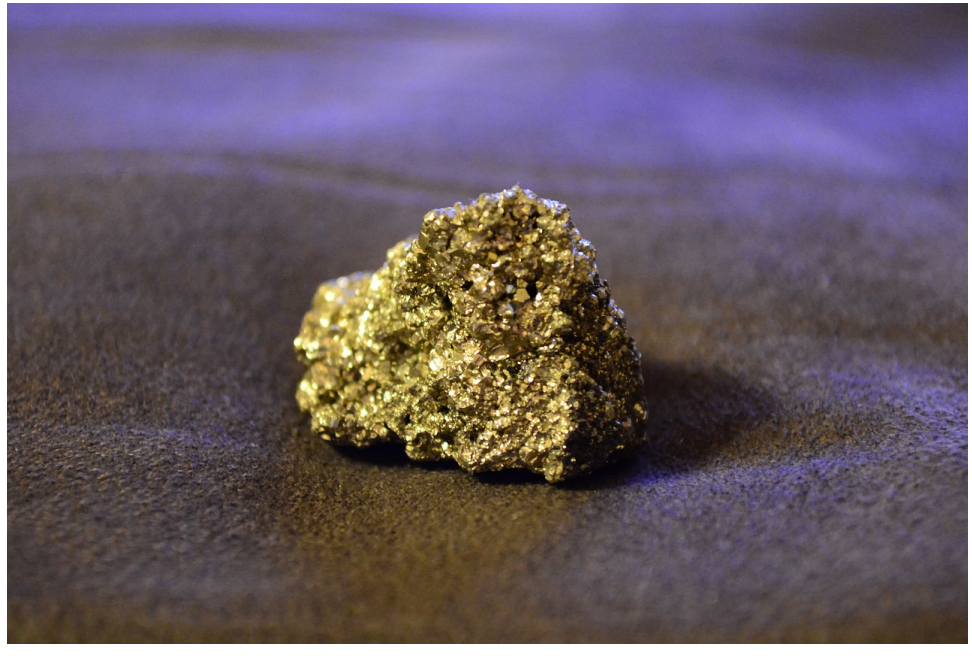
Università di Udine

5th June 2024

WHY OPTIMIZE?

- Tasteful design allows/accounts for optimization
- Optimize when output is good but not performance
- Otherwise **optimize your own time** and do nothing

FOOL'S GOLD



- People love the latest shiny things (advanced compilation techniques, vectorization, parallelization, hardware acceleration, etc.)
- Don't start there! **It's a trap!**

WHERE TO START OPTIMIZING

Mr.donovan, CC BY-SA 4.0

- Non-optimzed good design >> optimized bad design
- Best to proceed top-down IMO

WHAT IS DESIGN BTW?

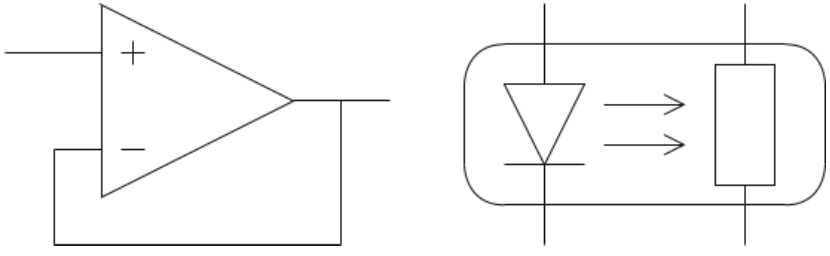
- Description of the various components (including users and I/O) and their interactions, alongside a handful highly-relevant implementation details
- Is good optimizable design compatible with AGILE, SCRUM, waterfall, etc.?

LET'S GET REAL, TOP-DOWN

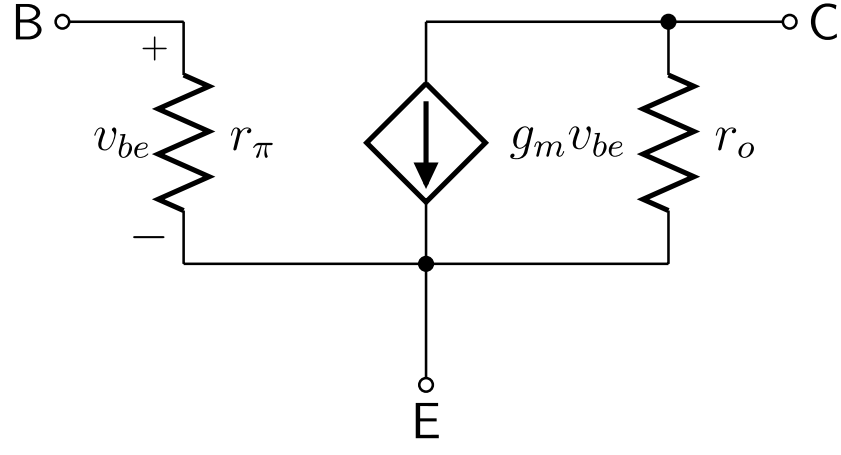
First improve effectiveness rather than efficiency (i.e., make the problem simpler)

- Divide-and-conquer:
  - easier to manipulate multiple smaller systems
  - also good for maintainability
  - define and use clean interfaces
  - cfr. UNIX philosophy
- Simplify as much as possible but no further
- Cleverly rearrange operations
- May also apply to software design itself (early opt?)

CIRCUIT ANALYSIS EXAMPLES



Isolating circuit parts



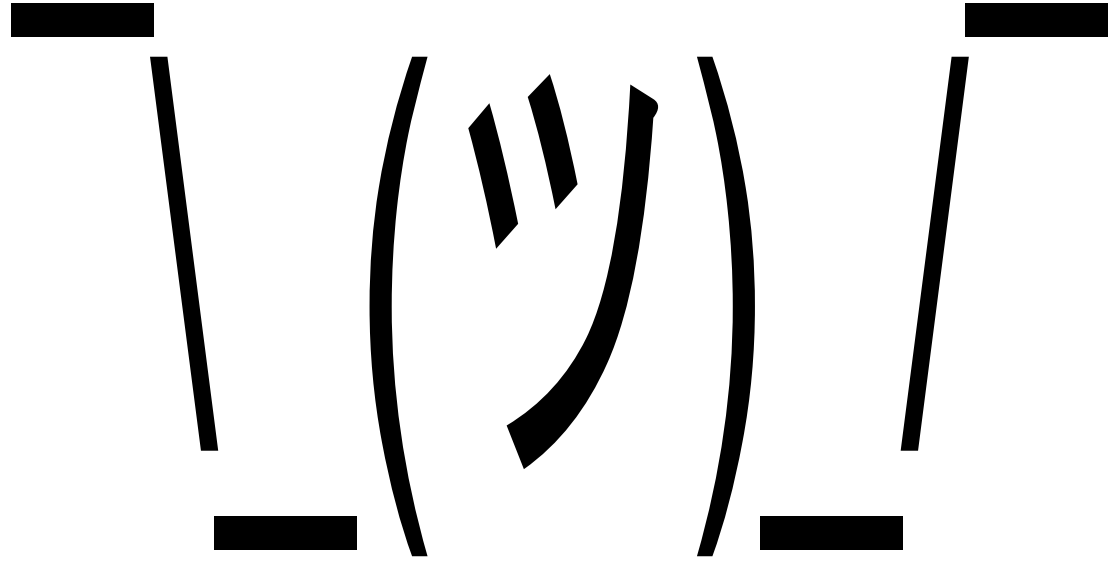
Small-signal BJT model

Circuit model complexity usually grows more-than-linearly w.r.t. number of components

STUPID REARRANGING OPTIMIZATION

- Say you need to filter an array of N elements and sort the remaining $R < N$ elements
- If you sort then filter, you'll examine $2N$ in the best case
- If you filter then sort, you'll exmaine $N + R < 2N$ elements in the best case
- Even better to combine both operations, exactly N visits

BUT SOME SYSTEMS ARE INHERENTLY COMPLEX

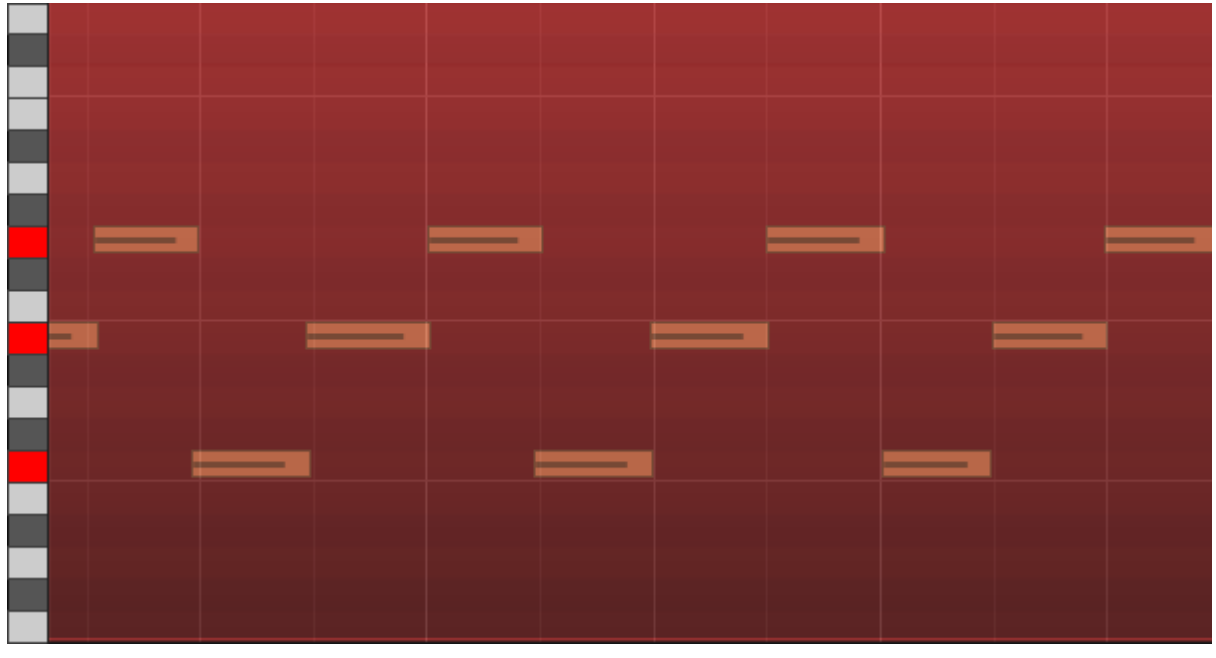


- You do what you can do
- Move to lower levels

TIME TO BE EFFICIENT

- Now that you're doing the right thing, do it fast
- Find out if there's an inherently faster approach (e.g., more convenient data structures)
- Consider using or even writing specific tools
- You might even contemplate code generation
- Notable success stories:
  - Parser generators
  - Neural networks
  - Compilers

ARPEGGIATOR



- User presses P keys on the keyboard
- One note from the set of pressed keys is played at any given time
- Duration of each note is the same

NAIVE IMPLEMENTATION

- Track pressed notes in an array of N elements, where $N \geq P$ is the number of keyboard notes
- If not time to change note, keep playing the last
- Otherwise scan the keyboard to find which is pressed and pick the next
- Insertion is $O(1)$, scan is $O(N)$
- (MIDI has 128 keys and you have typically 10 fingers)

OPTIMIZED APPROACH

- Track pressed notes in a balanced binary search tree
- If not time to change note, keep playing the last
- Otherwise at worst just walk the tree, which has P leaves, and pick the next
- Insertion and search are $O(\log P)$, walk is $O(P)$
- In our application, in the worst case scenario we have improved 17x, but usually more than 31x

NETMIX (AUTOMATED CIRCUIT ANALYSIS)

1. "Annotated" SPICE netlist (manual)
2. Circuit equations
3. Discretization
4. Solution of system of equations (interactive)
5. Ciaramella code
6. Output code (C++, MATLAB, JS, D)

START MESSING WITH CODE NOW

Only do really obvious improvements right now

POINT LIES WITHIN A CIRCLE?

- Naive, just apply $\sqrt{(x - x_0)^2 + (y - y_0)^2} \leq r$:

```
char point_is_in_circle(
    float x, float y,
    float x0, float y0,
    float r) {
  return sqrtf(powf(x - x0, 2.0f) + powf(y - y0, 2.0f)) <= r;
}
```

- Smarter:

```
char point_is_in_circle(
    float x, float y,
    float x0, float y0,
    float r) {
  float dx = x - x0;
  float dy = y - y0;
  return dx * dx + dy * dy <= r * r;
}
```

PARETO "OPTIMUM"?

80% of consequences come from 20% of causes (YMMV)

- Depends on the case
- Many times, if you optimize top-down properly, more effort leads to "diminishing returns" (outcome and/or maintainability)
- But we haven't addressed yet something fundamental…

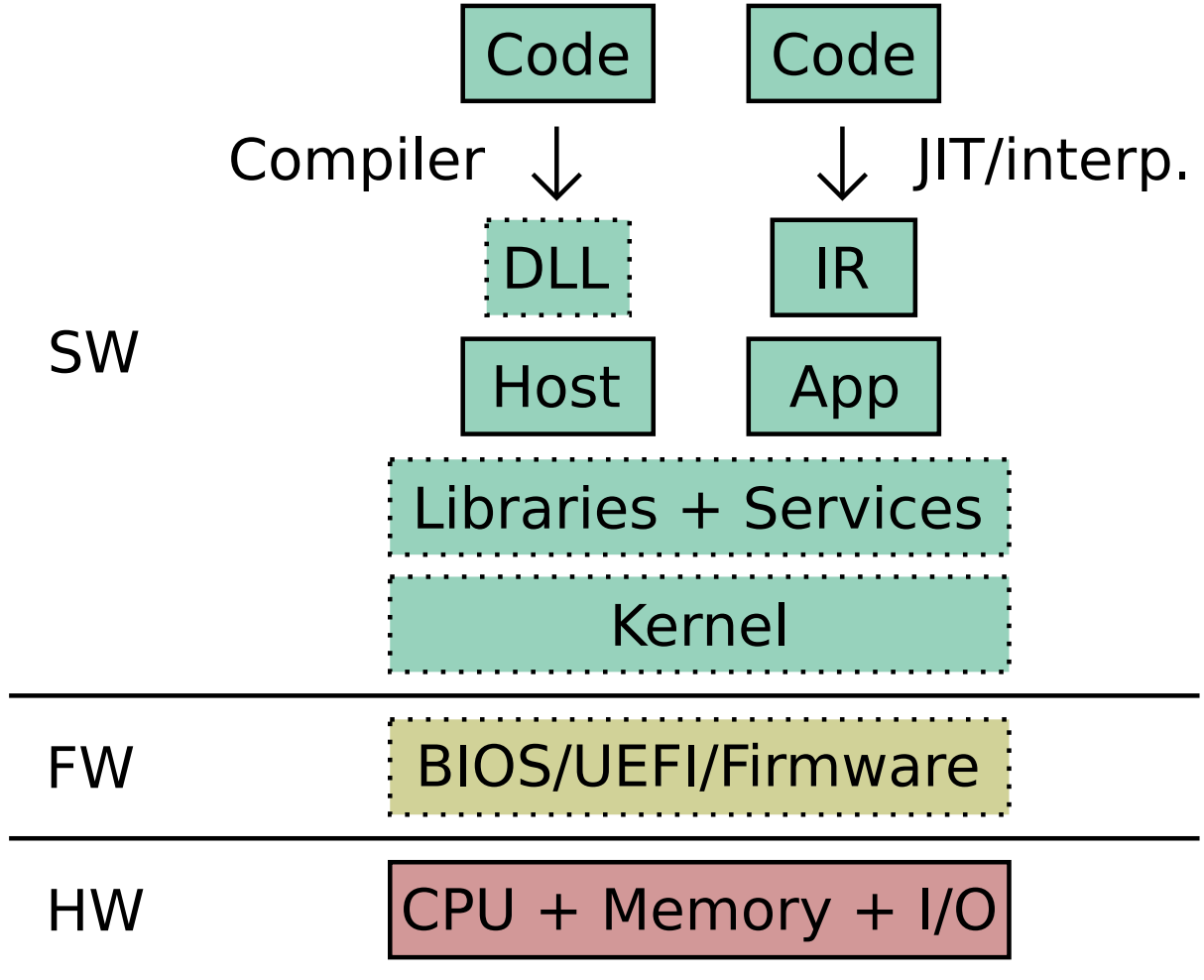
HOW TO IDENTIFY REAL ISSUES

- Code profiling: runtime program analysis measuring, e.g., execution time, memory usage, of whole program or parts (e.g., functions)
- Don't only measure in isolation, try real use cases
- First understand what are the costly operations
- If something is slow, observe metrics and try to understand which matter
- (Re)Consider complexity in terms of costly operations?

HOW PROFILERS WORKS

- The program is interrupted while running to collect data based on
  - Events (e.g., function calls, object creation/ destruction, memory allocation, even line-by-line)
  - Sampling, that is at regular intervals
- Compiler optimizations can mess things up (code lines and order do not correspond 1:1 to what is executed)
- Profiling is invasive and can invalidate results

SIMPLIFIED MODERN STACK



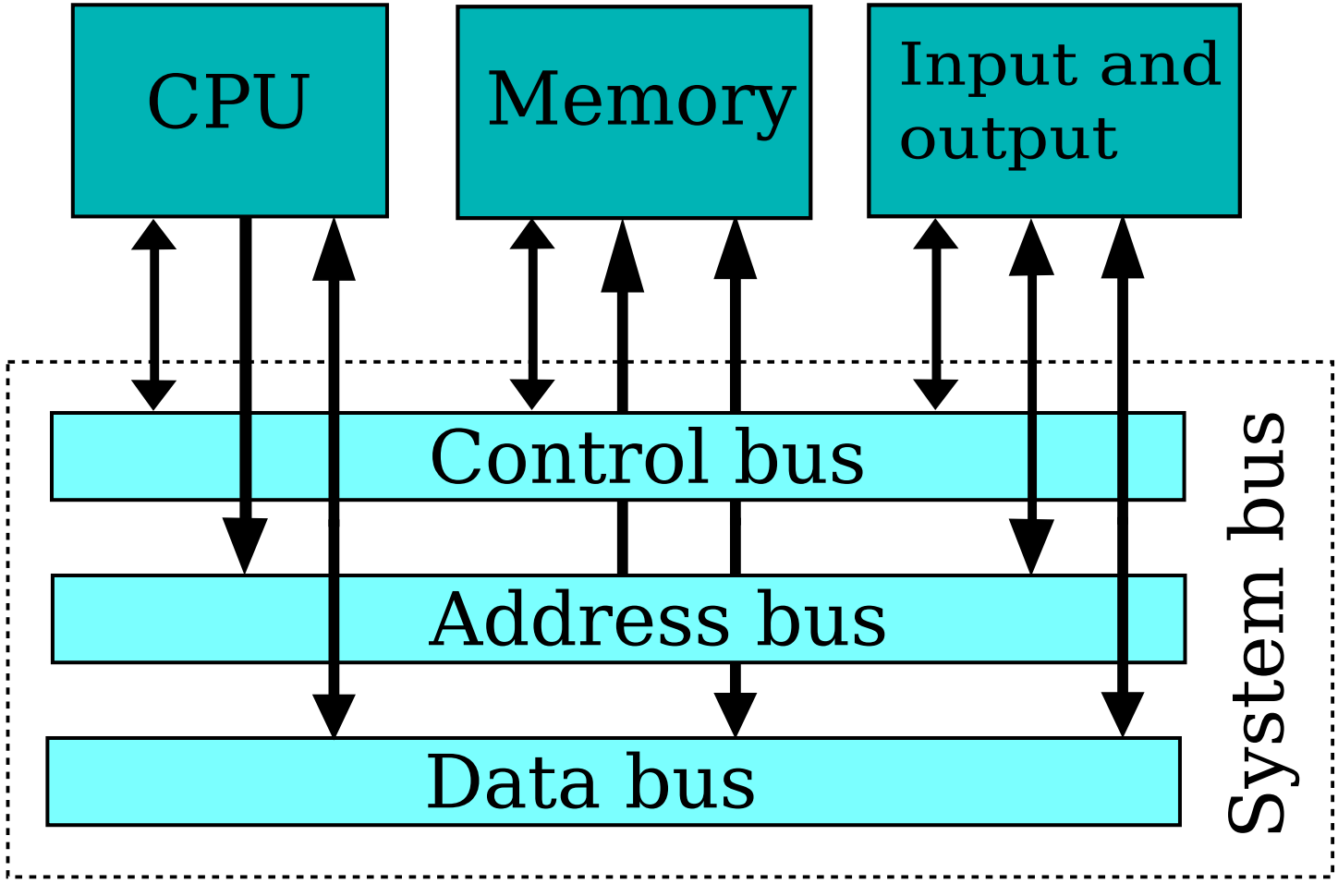
OPTIMIZATION PARADOX

Optimization is always target-dependent but often target is at least partially unknown.

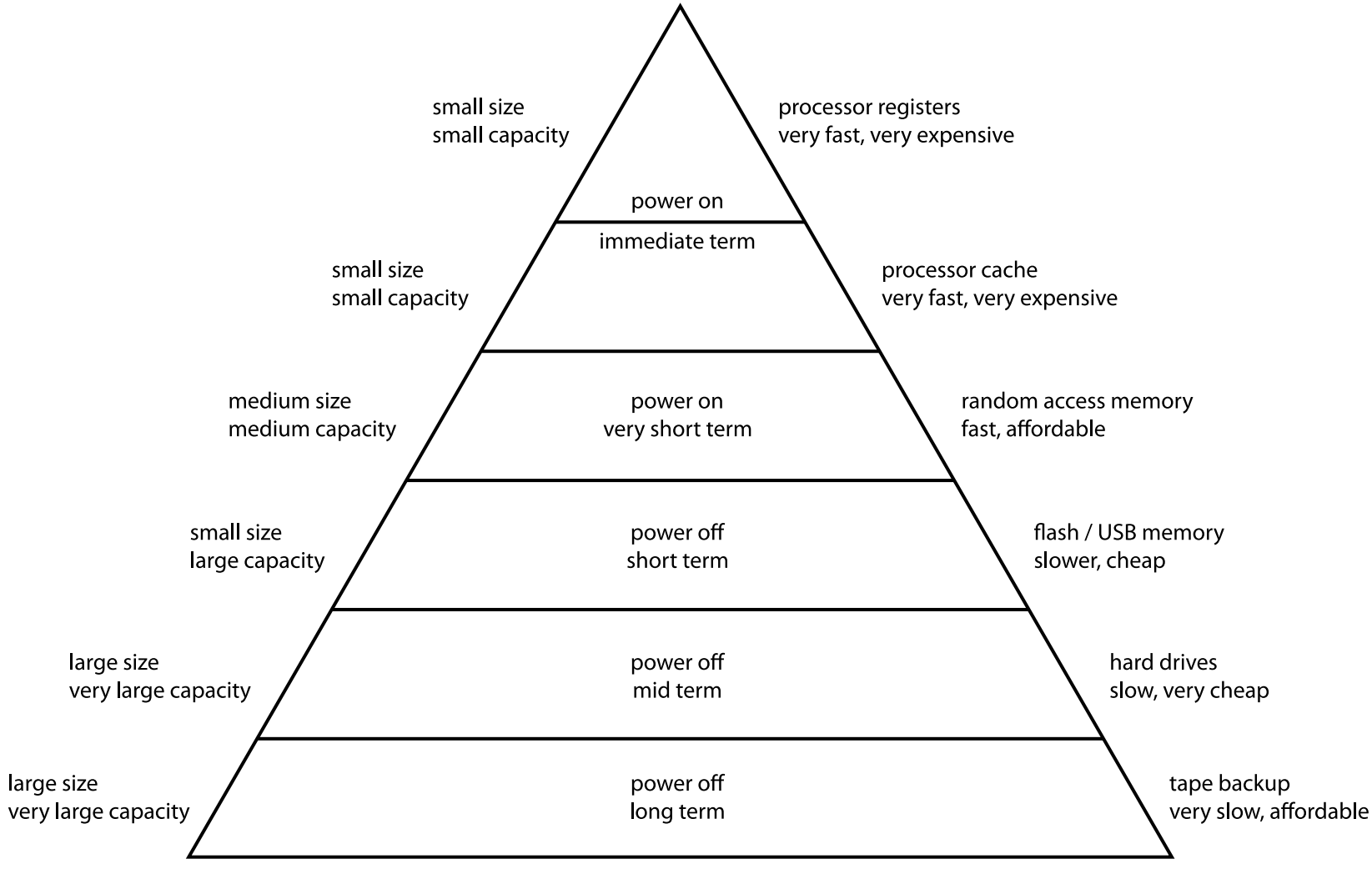
I'm hopefully giving you good general advice from now on, as well as a feel of how it's practically done, and also some theoretical context.

Anyway, always do your homework!

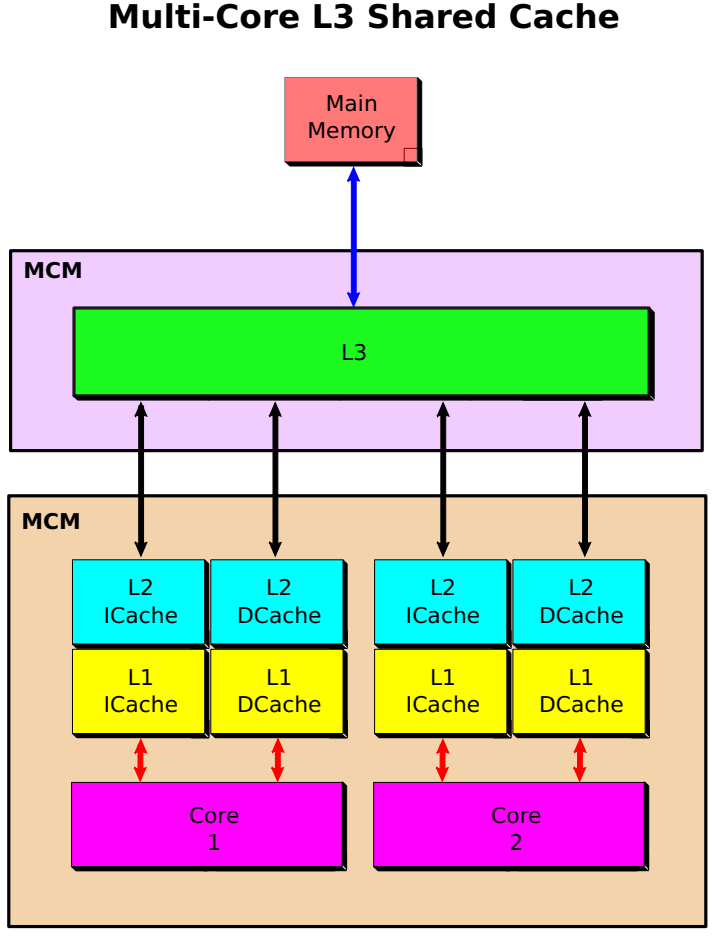
BUS



MEMORY HIERARCHY



CPU CACHE



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

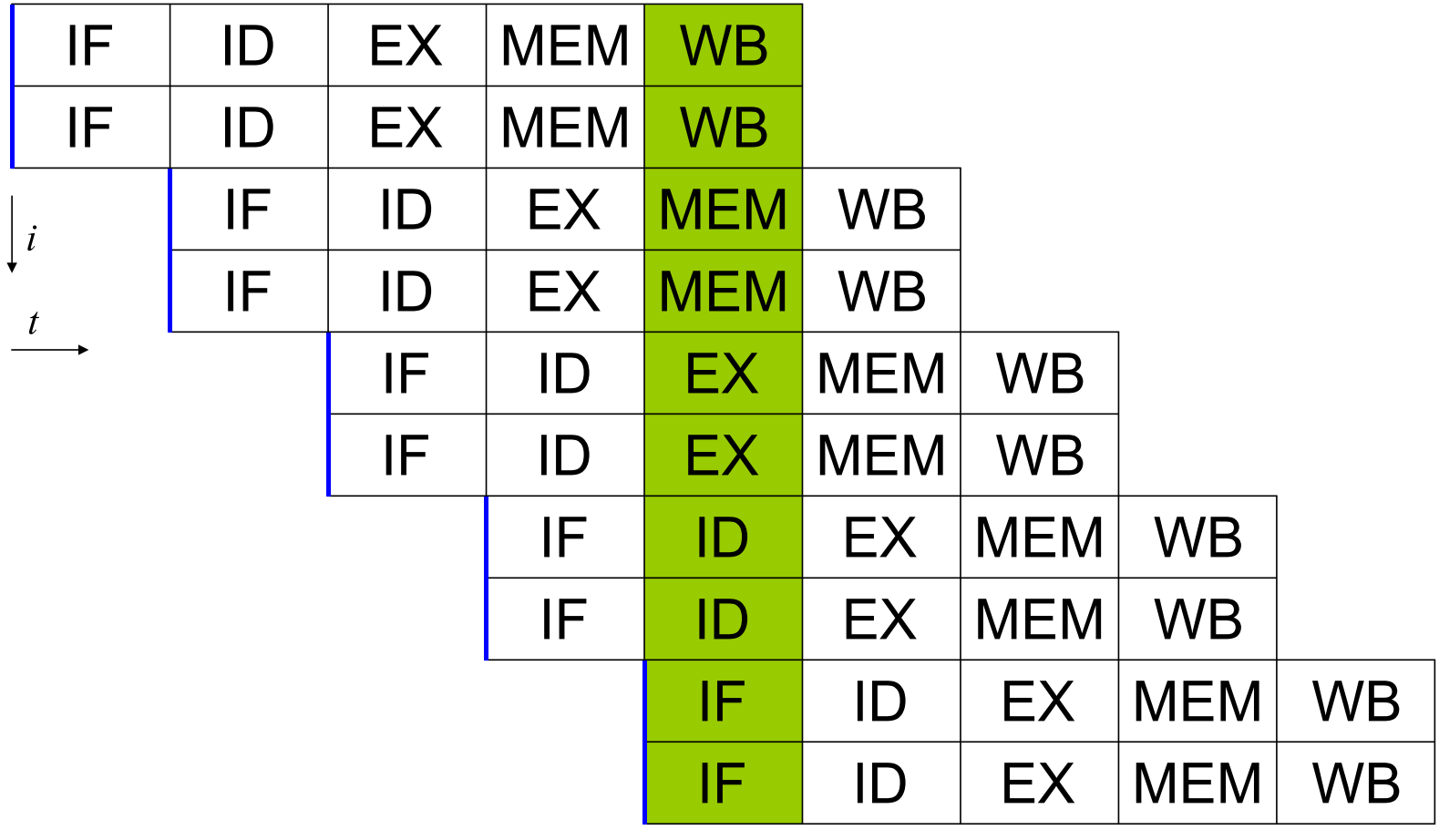
- Limit memory usage (code and data)
- Improve memory locality (code and data)
- Pre-allocate all needed memory if possible
- Inline small functions
- Lock code to a given core
- Avoid writing to memory, prefer local variables
- Many small loops preferable (beware of buffering)
- Avoid concurrent operations 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

OPTIMIZED PROGRAMMING FOR REAL-TIME APPLICATIONS AND BEYOND

#3: IMPLEMENTATION

Stefano D'Angelo

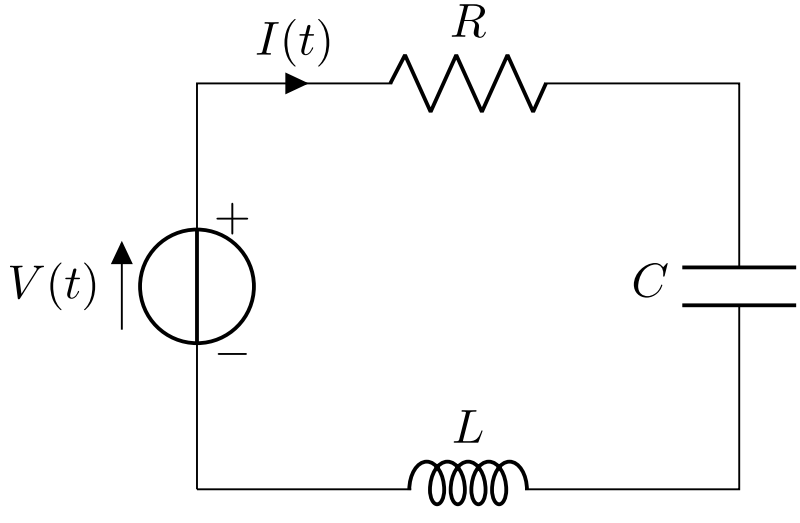
Università di Udine

6th June 2024

UPDATE RATES IN AUDIO

- Absolute constants
- Sample-rate-dependent constants
- Control rate signals, asynchronous, buffer-level
- Audio rate signals, synchronous, sample-level
- There could be many sample/audio rates

AN RLC CIRCUIT MODEL



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

- $V[n], I[n], s1[n], s2[n]$ are audio rate
- B_0, A_1, A_2, R, L, C are control rate
- f_s is the sample rate

STEP 1

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

STEP 2

$$
\begin{aligned}
x[n] &= B_0 V[n] \\
I[n] &= x[n] + s1[n-1] \\
s1[n] &= s2[n-1] - A_1 I[n] \\
s2[n] &= -x[n] - A_2 I[n] \\
k &= \frac{1}{1 + 2f_s RC + 4{f_s}^2 LC} \\
B_0 &= k\left(2f_s C\right) \\
A_1 &= k\left(2 - 8{f_s}^2 LC\right) \\
A_2 &= k\left(1 - 2f_s RC + 4{f_s}^2 LC\right)
\end{aligned}
\quad \rightarrow \quad
\begin{aligned}
x[n] &= B_0 V[n] \\
I[n] &= x[n] + s1[n-1] \\
s1[n] &= s2[n-1] - A_1 I[n] \\
s2[n] &= -x[n] - A_2 I[n] \\
k_2 &= 2f_s C \\
k &= \frac{1}{1 + k_2 R + 2f_s k_2 L} \\
B_0 &= k k_2 \\
A_1 &= k\left(2 - 4f_s k_2 L\right) \\
A_2 &= k\left(1 - k_2 R + 2f_s k_2 L\right)
\end{aligned}
$$

STEP 3

$$
\begin{aligned}
x[n] &= B_0 V[n] \\
I[n] &= x[n] + s1[n-1] \\
s1[n] &= s2[n-1] - A_1 I[n] \\
s2[n] &= -x[n] - A_2 I[n] \\
k_2 &= 2 f_\mathrm{s} C \\
k &= \frac{1}{1 + k_2 R + 2 f_\mathrm{s} k_2 L} \\
B_0 &= k k_2 \\
A_1 &= k\left(2 - 4 f_\mathrm{s} k_2 L\right) \\
A_2 &= k\left(1 - k_2 R + 2 f_\mathrm{s} k_2 L\right)
\end{aligned}
\quad \rightarrow \quad
\begin{aligned}
x[n] &= B_0 V[n] \\
I[n] &= x[n] + s1[n-1] \\
s1[n] &= s2[n-1] - A_1 I[n] \\
s2[n] &= -x[n] - A_2 I[n] \\
k_2 &= 2 f_\mathrm{s} C \\
k_3 &= 2 f_\mathrm{s} k_2 L \\
k &= \frac{1}{1 + k_2 R + k_3} \\
B_0 &= k k_2 \\
A_1 &= k\left(2 - 2 k_3\right) \\
A_2 &= k\left(1 - k_2 R + k_3\right)
\end{aligned}
$$

STEP 4

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

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

$$
\begin{aligned}
I[n] &= B_0 V[n] + s1[n-1] \\
s1[n] &= s2[n-1] - A_1 I[n] \\
s2[n] &= -B_0 V[n] - A_2 I[n] \\
B_0 &= \frac{2 f_\mathrm{s} C}{1 + 2 f_\mathrm{s} RC + 4 {f_\mathrm{s}}^2 LC} \\
A_1 &= \frac{2 - 8 {f_\mathrm{s}}^2 LC}{1 + 2 f_\mathrm{s} RC + 4 {f_\mathrm{s}}^2 LC} \\
A_2 &= \frac{1 - 2 f_\mathrm{s} RC + 4 {f_\mathrm{s}}^2 LC}{1 + 2 f_\mathrm{s} RC + 4 {f_\mathrm{s}}^2 LC}
\end{aligned}
\quad \rightarrow \quad
\begin{aligned}
x[n] &= B_0 V[n] \\
I[n] &= x[n] + s1[n-1] \\
s1[n] &= s2[n-1] - A_1 I[n] \\
s2[n] &= \widehat{A_2} I[n] - x[n] \\
k_1 &= 2 f_\mathrm{s} \\
k_2 &= k_1 C \\
k_3 &= k_1 k_2 L \\
k_4 &= 1 + k_3 \\
k_5 &= k_2 R \\
k &= \frac{1}{k_4 + k_5} \\
B_0 &= k k_2 \\
A_1 &= k \left(2 - 2 k_3\right) \\
\widehat{A_2} &= k \left(k_5 - k_4\right)
\end{aligned}
$$

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)

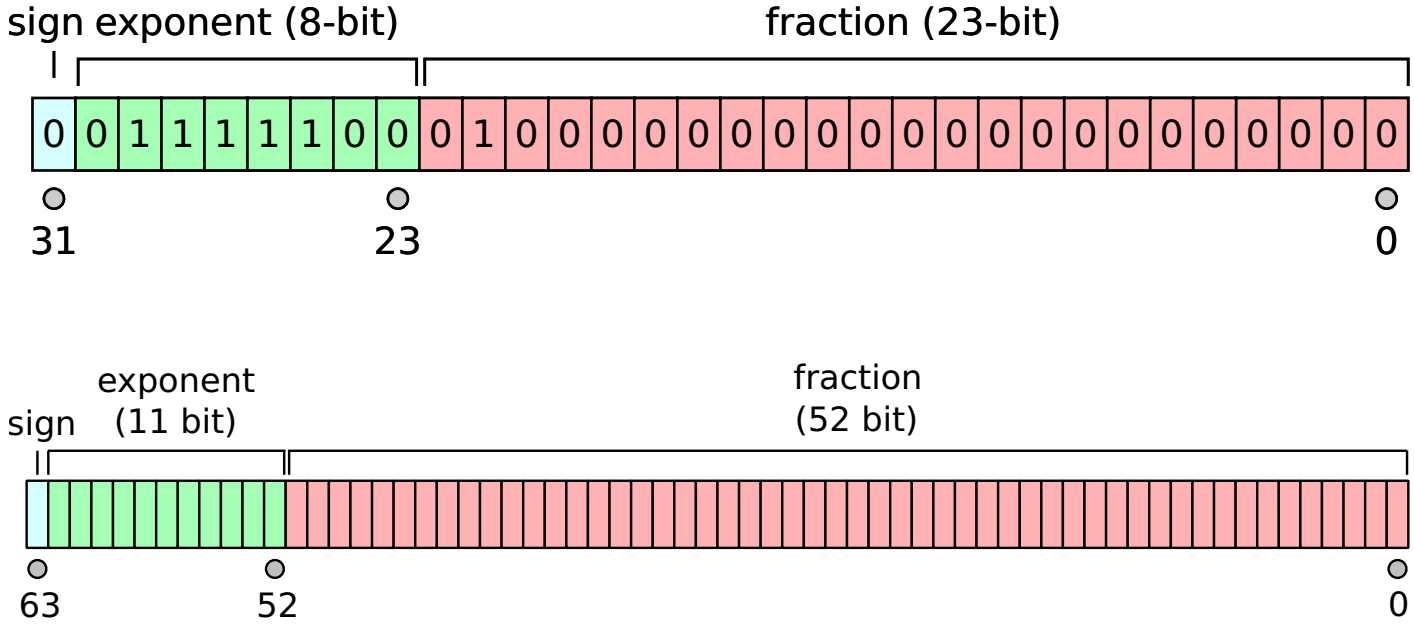
LET'S DIVE DEEPER #2

- $k_1 = 2f_\mathrm{s}$ 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

IEEE-754 REPRESENTATION



User Codekaizen on Wikipedia, CC BY-SA 4.0

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

$b = 127$ (32 bit), $b = 1023$ (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 time-recursive systems
- 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

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 * 0.164042561333452f));
}
```

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

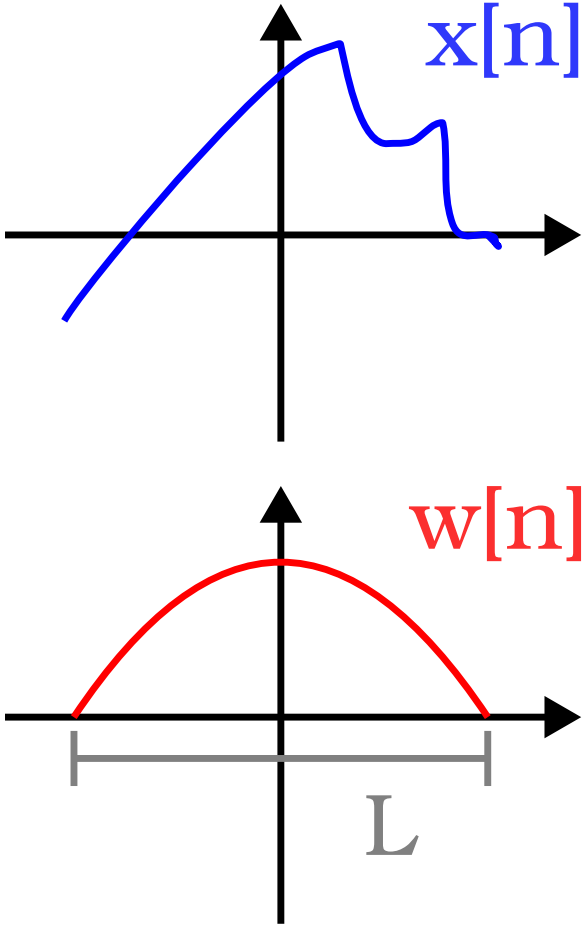
LIBM CONSIDERED HARMFUL

- IEEE-754 compliance requires handling lots of corner cases
- Usually this is implemented in software (slow)
- "Unsafe" compiler optimizations can alleviate this

OPTIMIZE INNER LOOPS FIRST

- They are executed more times, hence even small savings add up significantly
- Factor out branching and maybe duplicate loops
- Exploit symmetries

A WEIGHTED MOVING AVERAGE



$$y[n] = \sum_{m=-\lfloor L/2 \rfloor}^{\lfloor L/2 \rfloor} x[m]w[n-m]$$

$$\sum_{m=-\lfloor L/2 \rfloor}^{\lfloor L/2 \rfloor} w[n] = 1$$

$x[n]$ is input, $y[n]$ is output, $w[n]$ is weighting function

$w[n]$ is symmetrical and of support $[-\lfloor L/2 \rfloor, \lfloor L/2 \rfloor]$, with L odd

a.k.a. discrete-time convolution with a symmetrical odd-length kernel with DC gain 1

NAIVE IMPLEMENTATION

```
#define W_LEN ... // assuming it's odd and >= 3
#define W_CENTER (W_LEN / 2)

float w[W_LEN] = { ... };

void wma(float *x, float *y, int len) {
  for (int i = 0; i < len; i++) {
    y[i] = 0.0f;
    for (int j = 0; j < W_LEN; j++) {
      int d = j - W_CENTER;
      int idx = i + d;
      if (idx >= 0 && idx < len)
        y[i] += w[j] * x[idx];
    }
  }
}
```

OPTIMIZED IMPLEMENTATION, PART 1

```
#define W_LEN ... // assuming it's odd and >= 3
#define W_CENTER (W_LEN / 2)

float w[W_CENTER + 1] = { ... }; // only store first half + peak

void wma(float *x, float *y, int len) {
  // a few corner cases to start with
  if (len == 0)
    return;
  if (len == 1) {
    y[0] = w[W_CENTER] * x[0];
    return;
  }
  if (len == 2) {
    y[0] = w[W_CENTER] * x[0] + w[W_CENTER - 1] * x[1];
    y[1] = w[W_CENTER - 1] * x[0] + w[W_CENTER] * x[1];
    return;
  }

  memset(y, 0, sizeof(float) * len); // zero out y
```

OPTIMIZED IMPLEMENTATION, PART 2

```
if (len <= (W_CENTER + 1)) { // w fully covers x
  int lh = len >> 1; // = len / 2
  int i = 0, j0 = W_CENTER;

  for (int kns = 1; i < lh; i++, kns += 2) {
    // symmetrical part
    for (int j = j0, k = 0, k2 = W_LEN - 1; k < i; j++, k++, k2--)
      y[i] += w[j] * (x[k] + x[k2]);
    // peak
    y[i] += w[W_CENTER] * x[i];
     // non-symmetrical part
    j0--;
    for (int k = kns, j = j0; k < len; j--, k++)
      y[i] += w[j] * x[k];
  }
```

OPTIMIZED IMPLEMENTATION, PART 3

```
  if (len & 1) { // len is odd
    // symmetrical part
    for (int j = j0, k = 0, k2 = W_LEN - 1; k < i; j++, k++, k2--)
      y[i] += w[j] * (x[k] + x[k2]);
    // peak
    y[i] += w[W_CENTER] * x[i];
    j0--; i++;
  }
  for (int kns = 1; i < len; i++, j0--, kns++) {
    int j = j0, k = 0;
    // non-symmetrical part
    for (; k < kns; j++, k++)
      y[i] += w[j] * x[k];
    // symmetrical part
    for (int k2 = W_LEN - 1; k < i; j++, k++, k2--)
      y[i] += w[j] * (x[k] + x[k2]);
    // peak
    y[i] += w[W_CENTER] * x[i];
  }
  return;
}
```

OPTIMIZED IMPLEMENTATION, PART 4

```
  // leaving this for you to optimize if you want
  for (int i = 0; i < len; i++) {
    y[i] = 0.0f;
    for (int j = 0; j < W_LEN; j++) {
      int d = j - W_CENTER;
      int idx = i + d;
      if (idx >= 0 && idx < len)
        y[i] += w[j] * x[idx];
    }
  }
}
```

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 $x \approx 0.75$

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

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
    * (0.0121807326003771 6f * axm - 0.2750231331124371f);
}
```

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

ROOT-FINDING ALGORITHMS

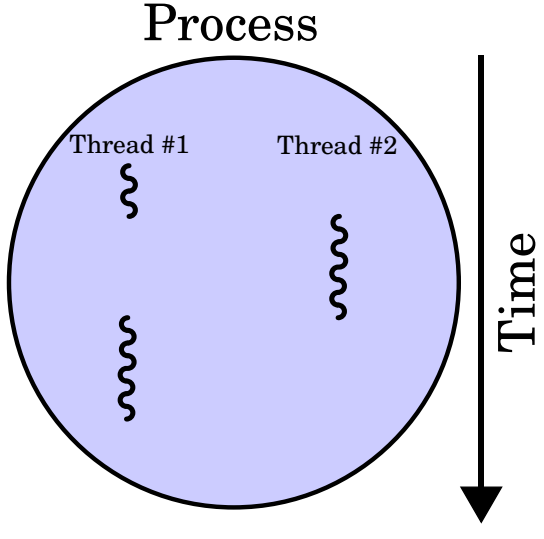
- Needed for implicitly-defined functions
- Also useful to approximate closed-form expressions
- They need a good enough first guess
- Famous methods: bisection, Newton-Raphson (needs derivative), secant

A RECIPROCAL ALGORITHM

```
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)

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

OPTIMIZED PROGRAMMING FOR REAL-TIME APPLICATIONS AND BEYOND

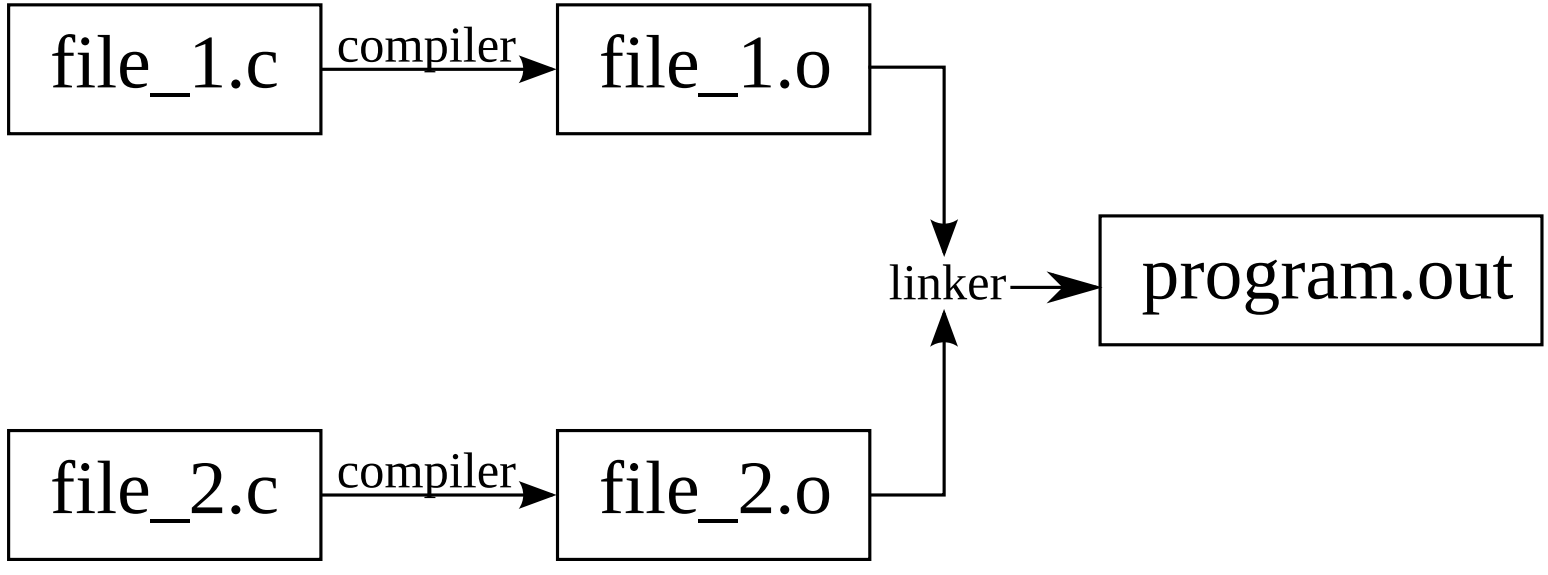
#4: COMPILERS AND LOW-LEVEL OPTIMIZATIONS

Stefano D'Angelo

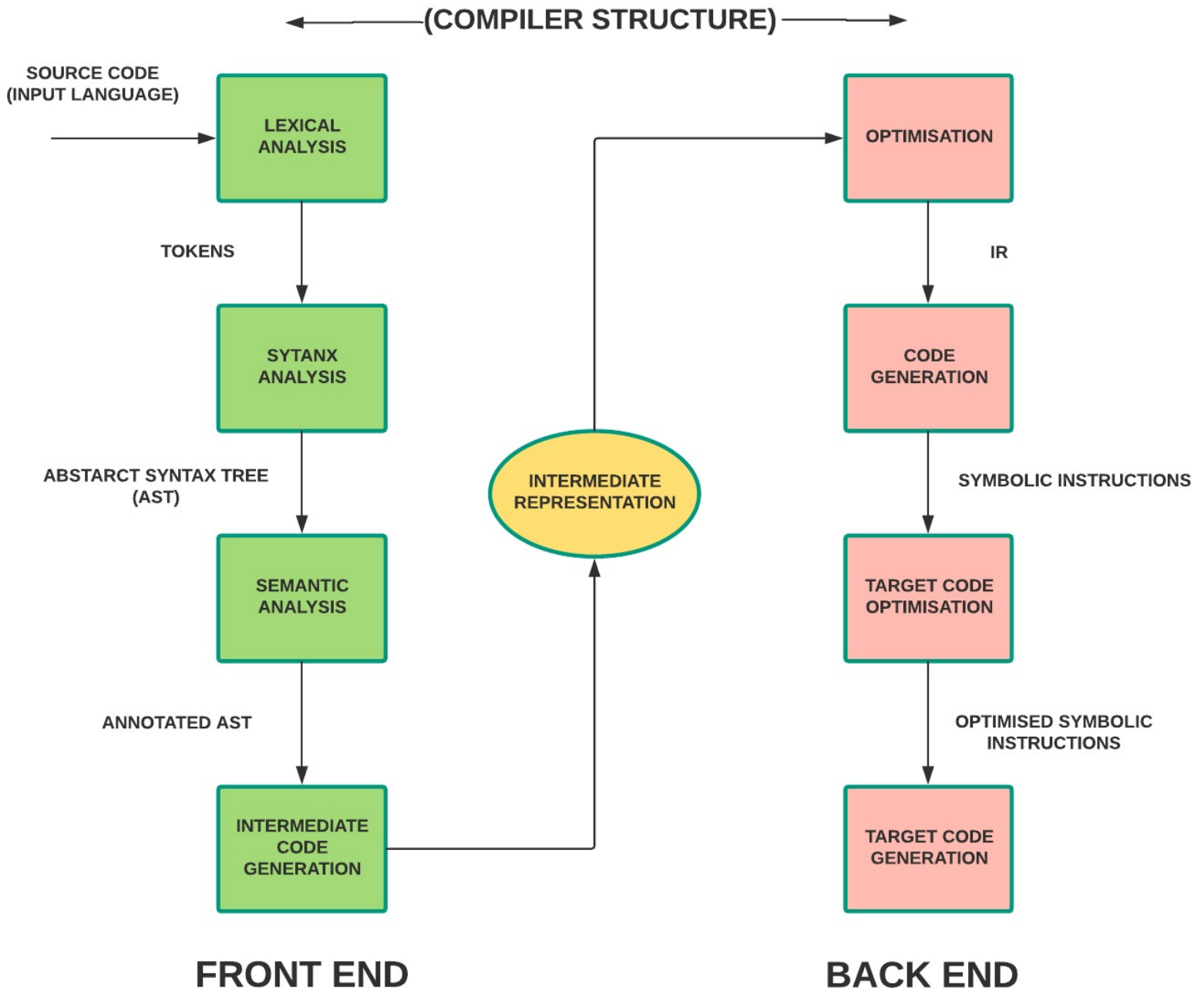
Università di Udine

7th June 2024

BUILDING PROCESS



- .o files contain machine code and metadata
- Not executable, miss "function references"


(COMPILER STRUCTURE)
SOURCE CODE
(INPUT LANGUAGE)
LEXICAL ANALYSIS
TOKENS
SYTANX ANALYSIS
ABSTARCT SYNTAX TREE
(AST)
SEMANTIC ANALYSIS
ANNOTATED AST
INTERMEDIATE CODE GENERATION
INTERMEDIATE REPRESENTATION
OPTIMISATION
IR
CODE GENERATION
SYMBOLIC INSTRUCTIONS
TARGET CODE OPTIMISATION
OPTIMISED SYMBOLIC INSTRUCTIONS
TARGET CODE GENERATION
FRONT END
BACK END

COMPILER OPTIMIZATIONS

- An incredible amount and variety
- Language/machine-dependent/independent
- Loop, data-flow, code generation, interprocedural, link-time…
- No miracles: languages don't express problem-level context
- Basically they apply many of the lower-level tricks discussed
- One more reason to focus on the bigger picture

TARGET CODE GENERATION/ OPTIMIZATION

- Consist essentially of 3 (complicated, and thus normally separated) phases:
  - Register allocation, based on liveness analysis
  - Instruction selection, based on hardcoded rules, pattern-matching, …
  - Execution order, subject to optimization

STATIC SINGLE-ASSIGNMENT

- Internal code representation used by many compilers (GCC, LLVM, MSVC, etc.)
- Rename variables to only assign once, e.g.,

$$
\begin{array}{ll}
x \leftarrow 1 & x_1 \leftarrow 1 \\
x \leftarrow 2 \quad \rightarrow & x_2 \leftarrow 2 \\
y \leftarrow x + 3 & y_1 \leftarrow x_2 + 3
\end{array}
$$

- Allows for efficient implementation of many optimization algorithms (constant propagation, elimination of dead/redundant code, strength reduction)

REGISTER ALLOCATION

- CPUs have a very limited number of registers (up to 32 normally) but programs needs to manipulate large sets of data
- Data needs to be loaded from memory to registers for operations to be performed, results also live in registers until written back to memory
- Need to be very efficient allocating registers and minimize moves
- Liveness analysis, linear scan allocation (fast, greedy allocation, JIT), graph-coloring allocation (slower, complex, better results)

COMPILER OPTIONS (GCC)

- `-O0…  -O3`: no optimization to most optimizations (more space, less time)
- `-Os/-Oz`: optimize for size (less space, more time, yet faster than `-O1` typically)
- `-Ofast`: `-O3` + unsafe optimizations
- `-ffast-math`: unsafe math optimizations
- `-march, -mcpu, -mtune`: specify target architecture and processor
- `man gcc` to get scared

RESULTS FOR WMA FROM PREV SESSION

- $L = 31, len = 16$, 1M iterations
- `gcc wma.c -o wma -O0`
  - Naive: 1.842 us avg
  - Optimized: 0.684 us avg
- `gcc wma.c -o wma -O3 -ffast-math`
  - Naive: 0.448 us avg
  - Optimized: 0.123 us avg

Still better - how could GCC know about symmetry?

MEMORY ALIGNMENT AND PADDING

- Memory is actually read/write at addresses that are multiples of a *quantum*, usually processor word size $\longrightarrow$ naturally aligned
- Non-aligned data needs mutliple reads/writes
- Compilers add padding between data structure members
- This can be controlled with compiler hints (non-standard)

MORE COMPILER HINTS (C)

- `register` keyword
- `inline` keyword
- Branch-prediction, GCC:
  `__builtin_expect(long exp, long c)`

NOT SATISFIED YET?

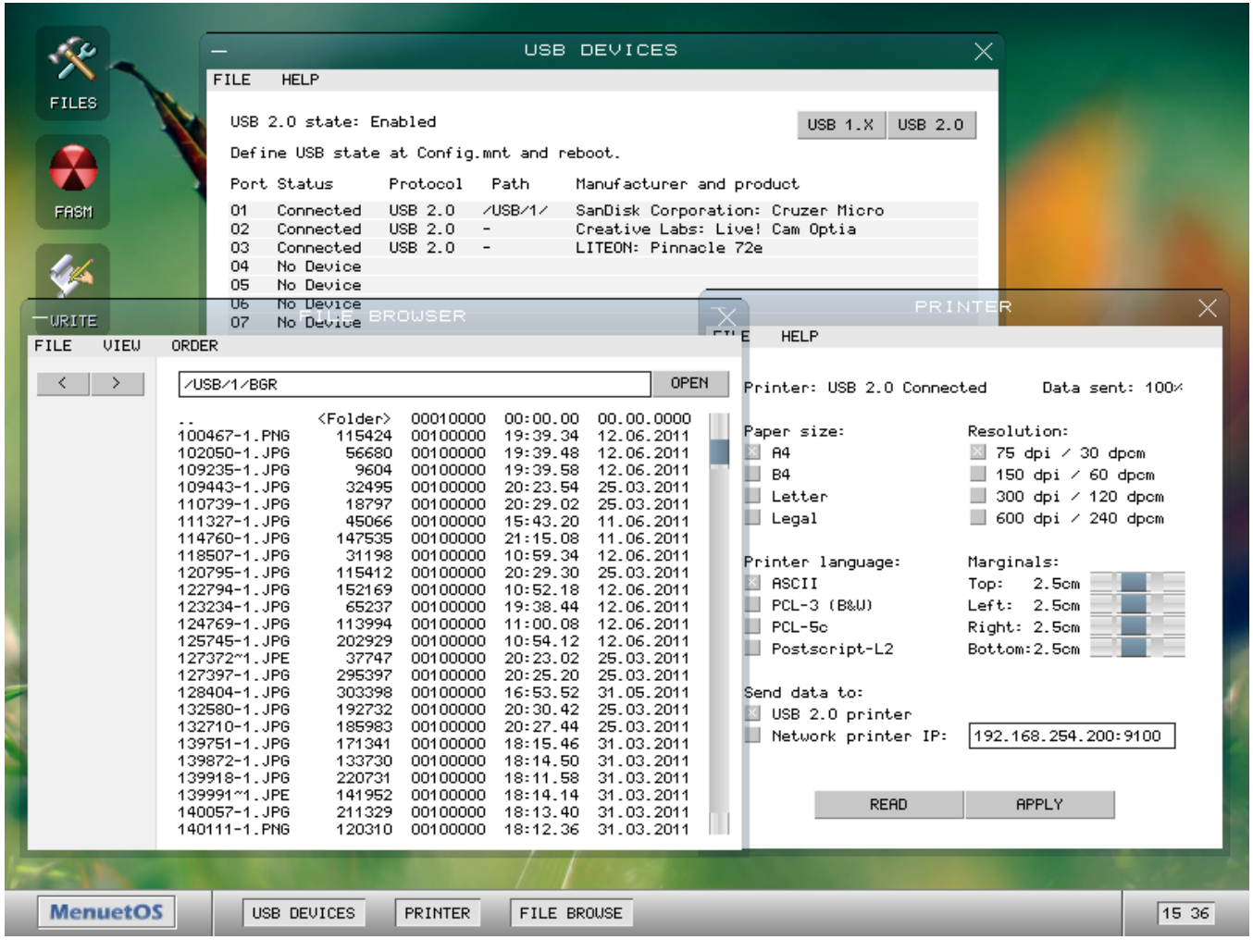
- Code in assembly ("textual machine code") if you think you can beat your compiler
- You can check what your compiler produces by disassembling (and learn hopefully)
- Deal with CPU instructions, registers, memory locations, etc., explicitly
- Non portable by definition
- Can't go lower level than that
- Current C/C++ compilers also allow you to inject assembly code directly

WHAT IT LOOKS LIKE

```
  mov ecx, 5    ; ecx = 5
  mov eax, ecx  ; eax = ecx

loop:
  cmp ecx, 1    ; compare ecx and 1
  jle end       ; if (ecx) <= (1) then jump to end

  sub ecx, 1    ; ecx = ecx - 1
  mul ecx       ; edx:eax = eax * ecx (unsigned mul, edx discarded)
  jmp loop      ; jump to loop

end:
  ; result is in eax
```

Adapted from https://gist.github.com/bieniekmateusz/3845880

HOW FAR CAN I PUSH THIS?



Vmt1970, CC BY-SA 3.0

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

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/

CPUID

- Instruction to determine CPU at runtime on x86
- Alternatives: system registers, `/proc/cpuinfo`, ...
- Nothing stops you from having multiple copies of code and dynamically pick the best one
- Or even invoke a compiler at runtime supplying machine information or similar

A-SID (2022)

- "Effect plugin" for the C64
- Uses sound chip audio input to "turn your C64 into a wah effect"
- Need to build yourself some cables
- Supports expression pedals
- No DSP, it "just" controls the sound chip
- www.orastron.com/asid

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

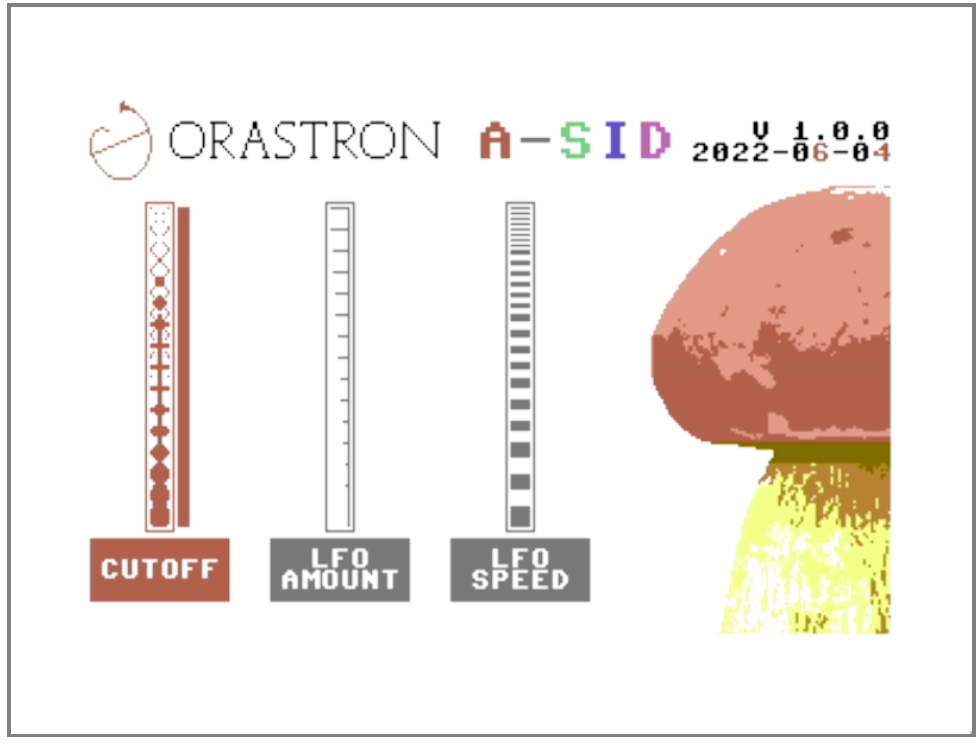
  $ab = ((a+b)/2)^2 - ((a-b)/2)^2$

  512 lookup entries, 16-bits each → 1 kB

25 Commodore c64 Games still great to play in 2020.

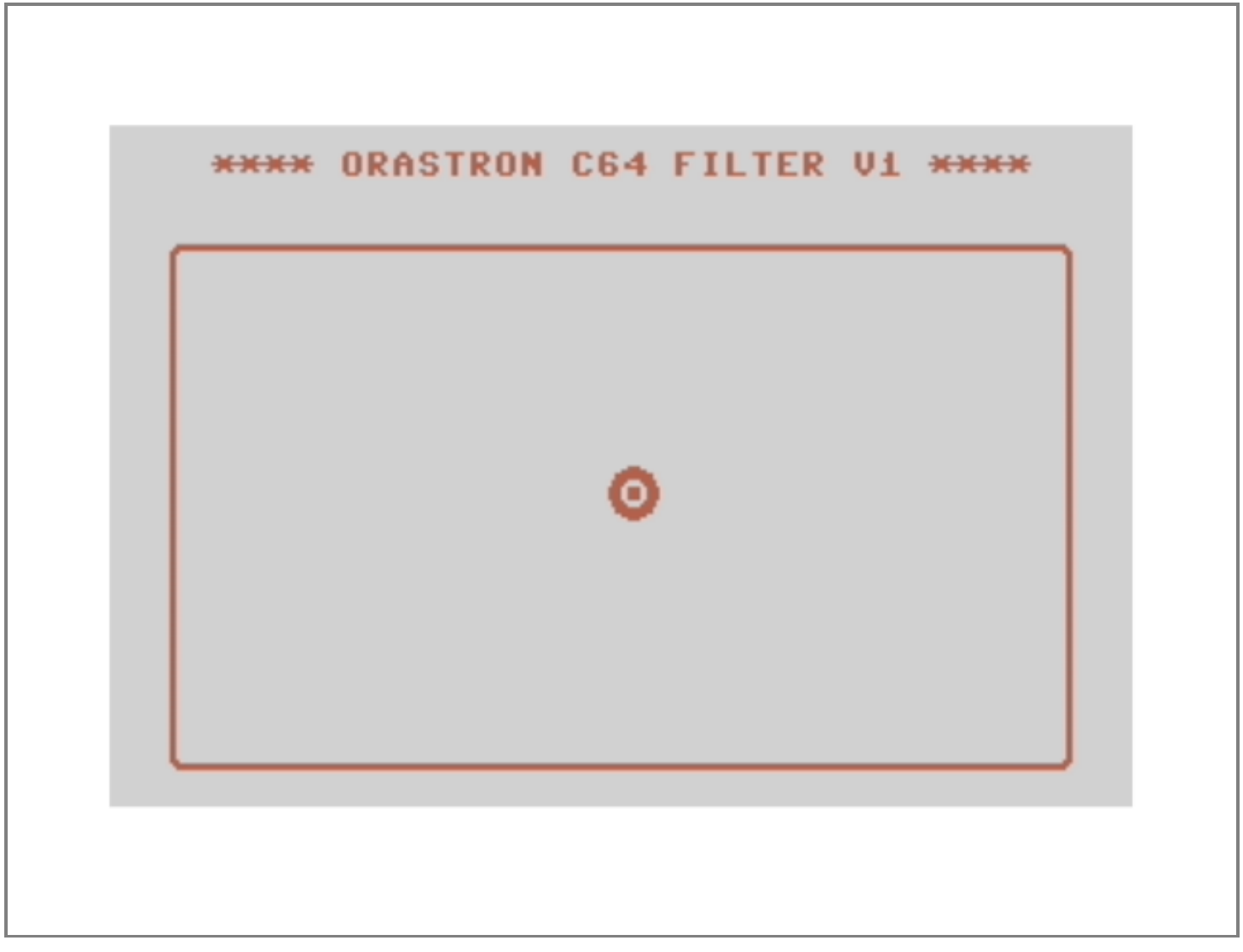
PRODUCT DESIGN

Demo



Turn your C64 into a wah effect with Orastron A-SID

A FIRST PROTOTYPE IN BASIC



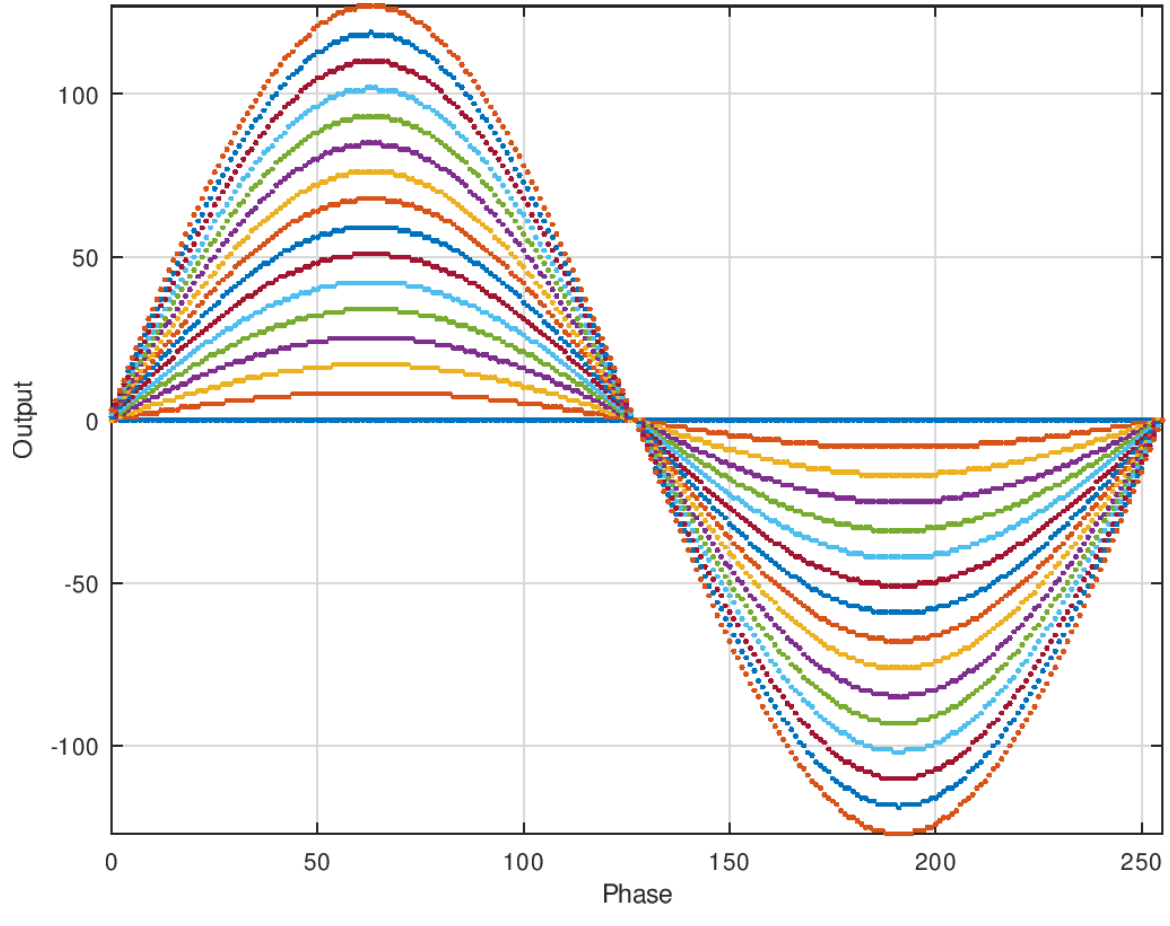
BOOT PROGRAM

- Written in BASIC - no real need for optimization
- Implements expression pedal calibration procedure and launches the main program

MAIN PROGRAM

- Written in 6510 assembly
- First a big comment containing a memory map
- Starts by initializing the gfx and sound chips and setting initial values
- Main loop:
  1. read joystick and expr. pedal, update current param and value and mod variables, 4 times
  2. LFO, compute actual cutoff, update GUI and sound chip registers, 1 time
- Should have used interrupts? This is good enough!

LFO

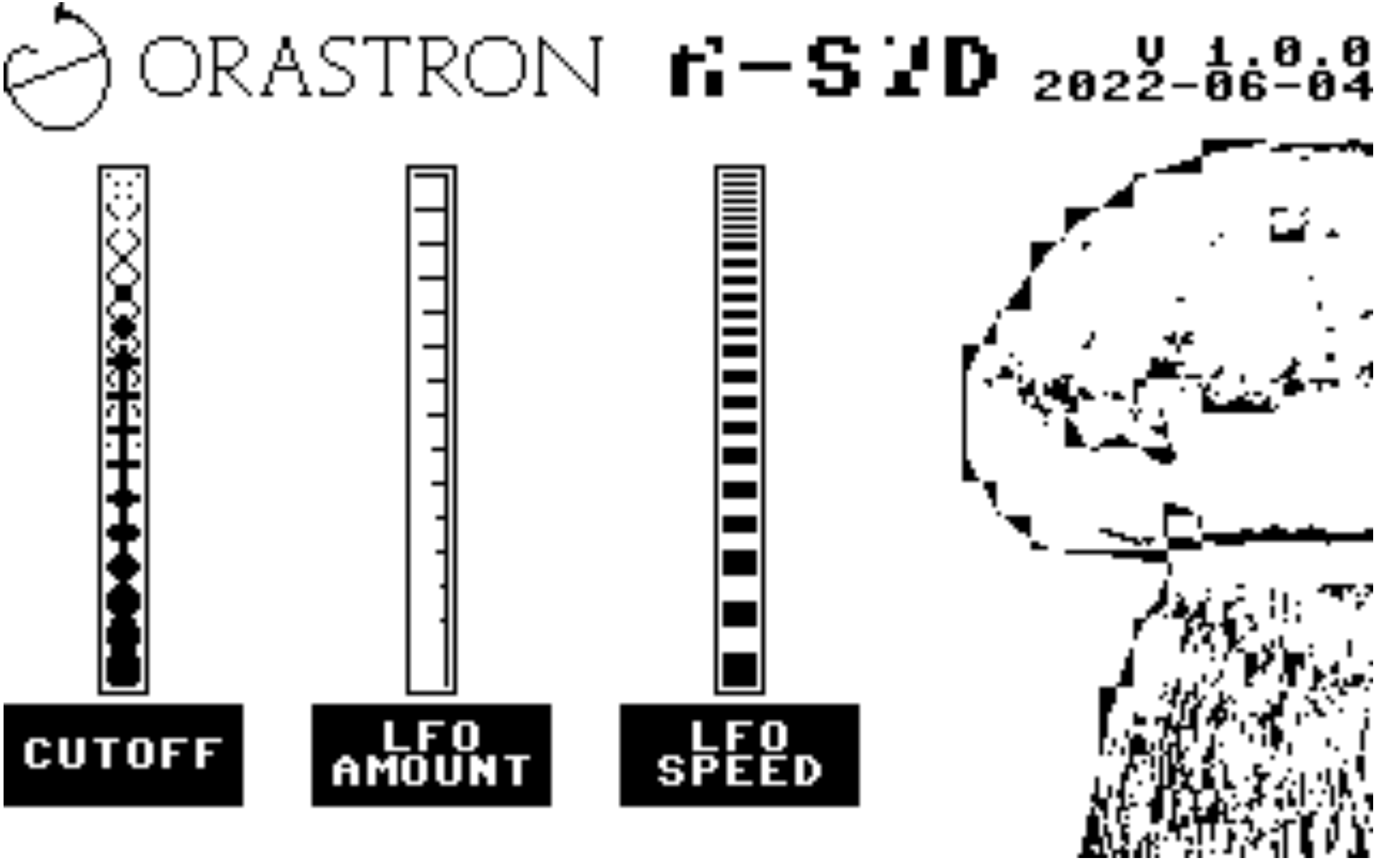


Phase generator (modulo increment exploiting overflow, increments in 16 values LUT tuned to 0.4-25 Hz range at the end) + sinusoidal waveshaper (16x256 LUT, LFO amt + phase)

STANDARD BITMAP MODE

- 320x200 resolution, 16 colors
- The screen is a grid of 8x8 pixel tiles, 2 colors per tile
- Bitmap data: 8 kB, indicating for each pixel whether to use fg or bg color
- Colormap data: 1 kB, indicating fg and bg color per tile
- We developed an HTML application specifically to draw/manipulate this data

BITMAP DATA



COLORMAP

FAST GRAPHICS

- Actually using 4 colormaps
- Only the colormap address needs to be changed
- Sliders and labels need runtime coloring though (colormaps edited on the fly)
- Modulated cutoff bar also needs to be dynamically drawn (live bitmap editing) and colored