More innovations in H.264/AVC software decoding

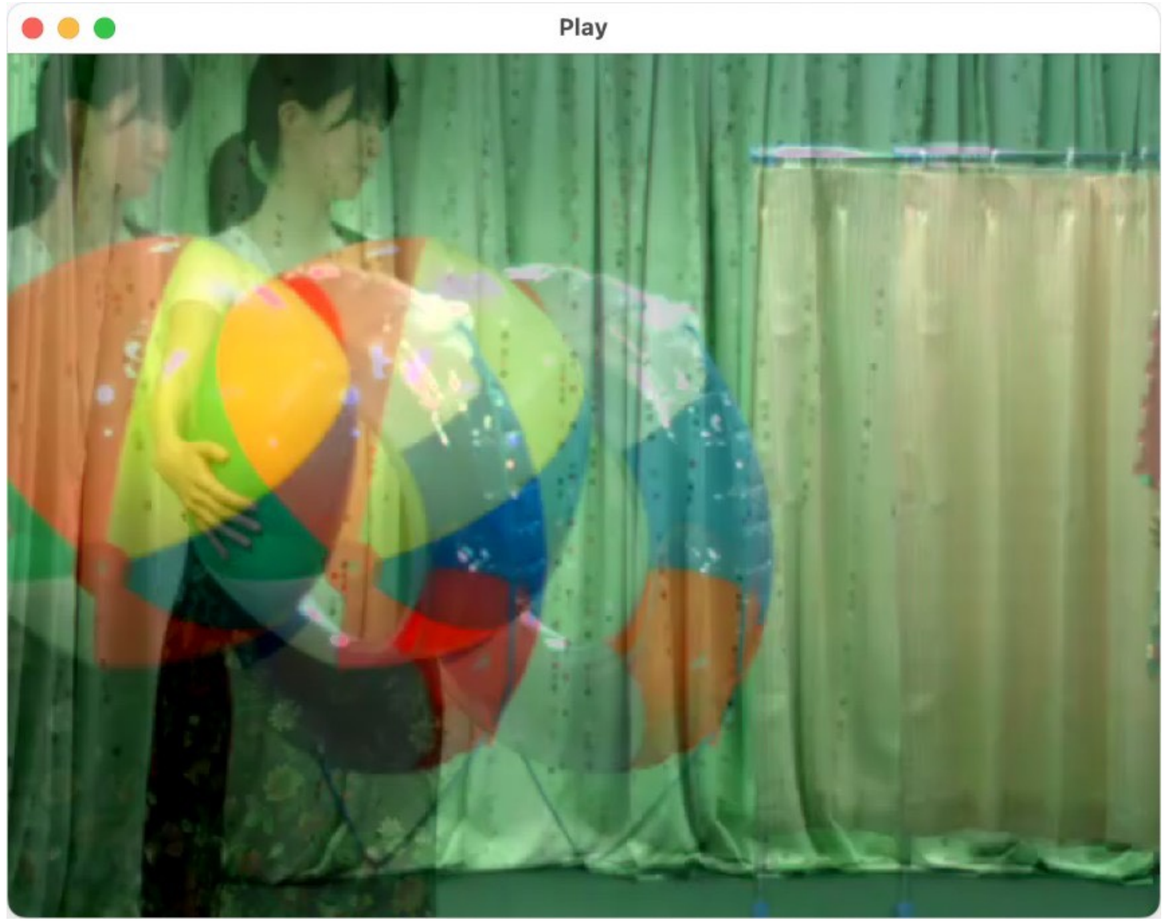
Thibault Raffaillac
PhD in HCI and SE

FOSDEM'25 – 2 February 2025

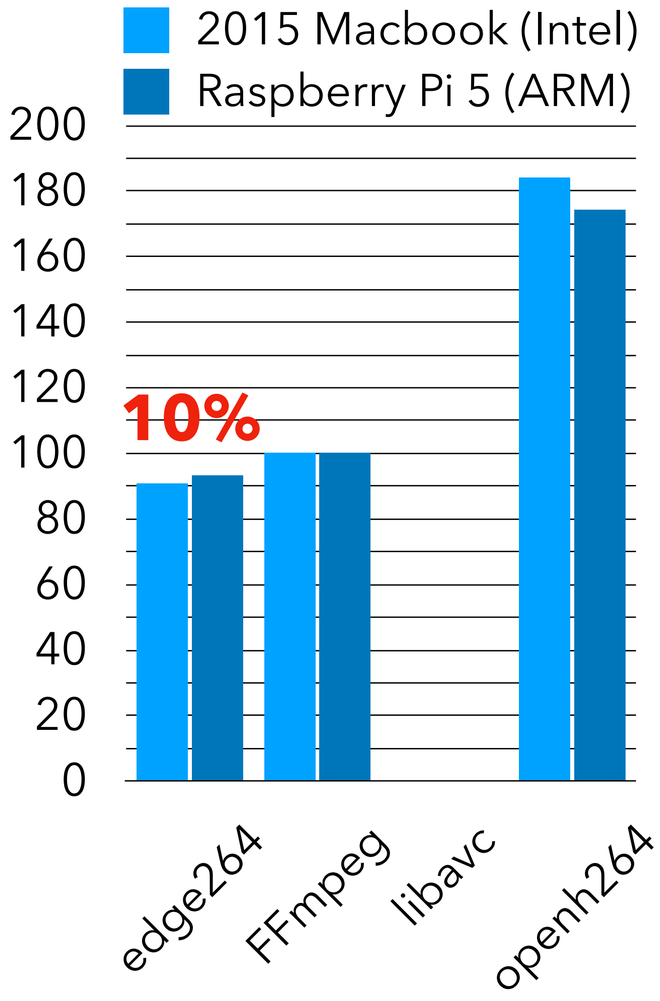
Introduction - edge264

H.264/AVC software decoder for Progressive High & MVC 3D profiles

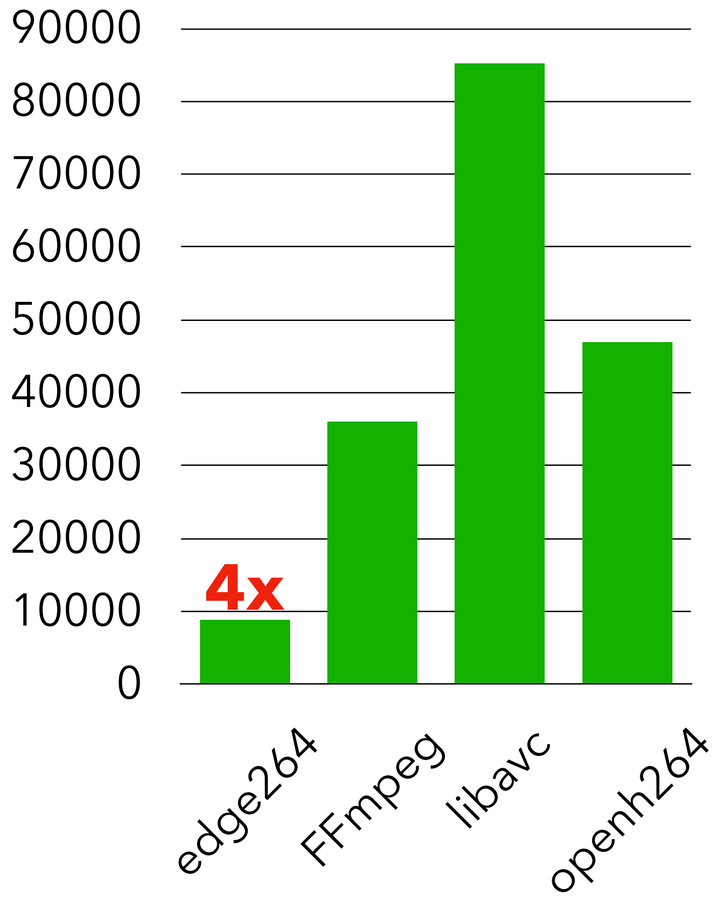
- Intel 32/64 with **SSE2**/SSSE3 (**runtime detection**)
- **ARM64 with NEON**
- **Linux**, **Windows**, Mac
- **Multi-threaded**



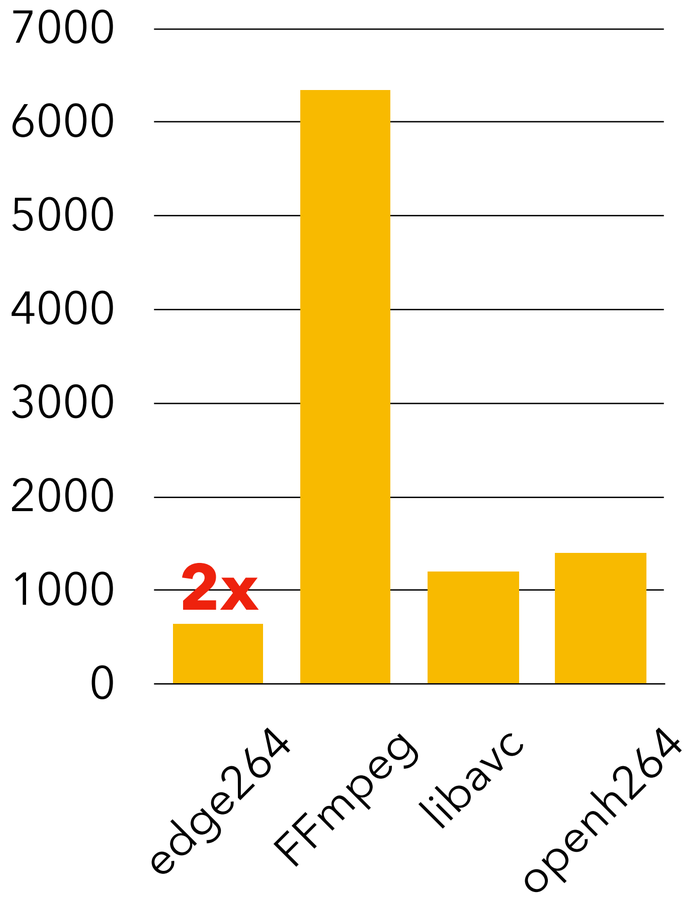
Introduction - Benchmarks



Speed (%)



Code size (loc)



Binary size (KiB)

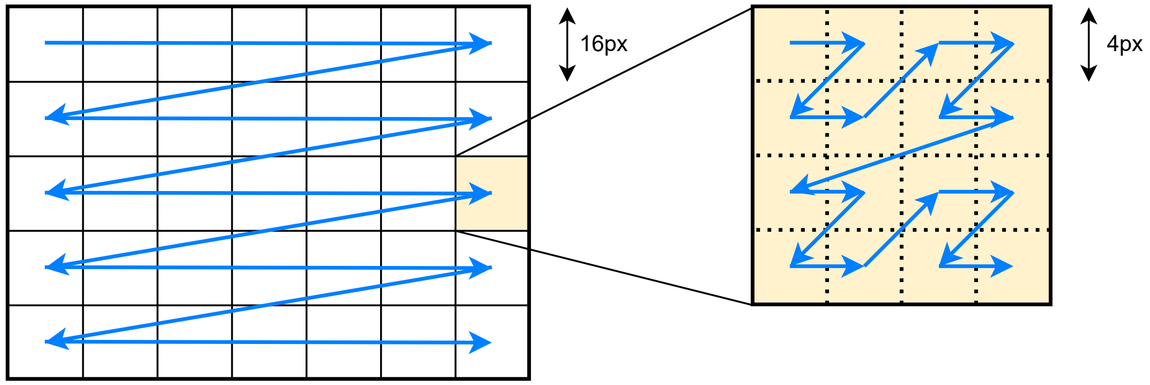
Thank you for your attention!

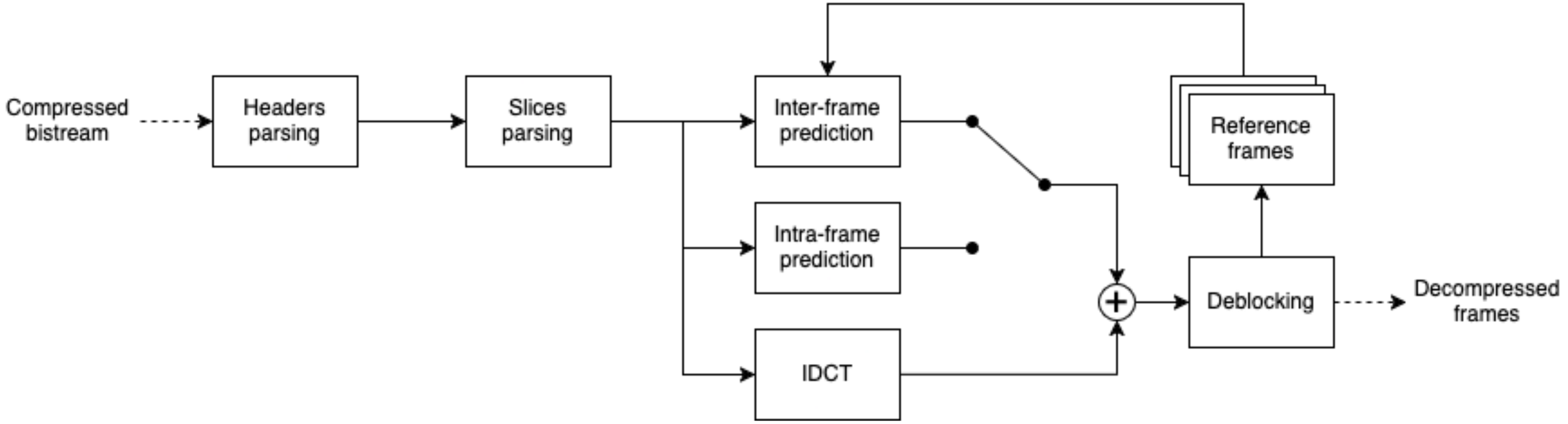
https://github.com/tvlabs/edge264

(end of postdoc in March, will work full time on it then)

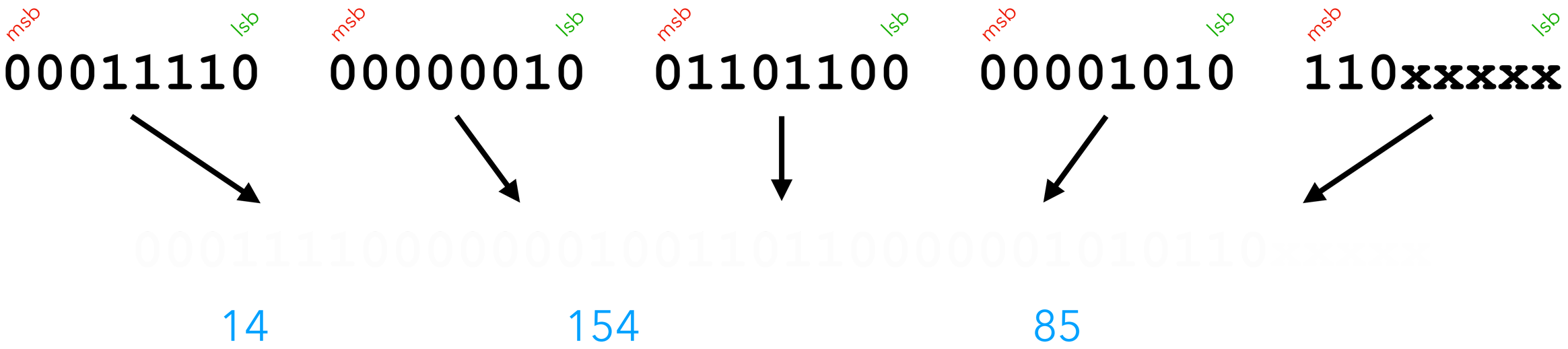
1. Single header file
2. Code blocks instead of functions
3. Tree branching
4. ~~Global context register~~
5. Default neighboring values
6. Relative neighboring offsets
7. Parsing uneven block shapes
8. Using vector extensions
9. Register-saturating SIMD

Introduction - H.264/AVC decoding





Technique 1 - Piston cached bitstream reader



Bitstream reader:

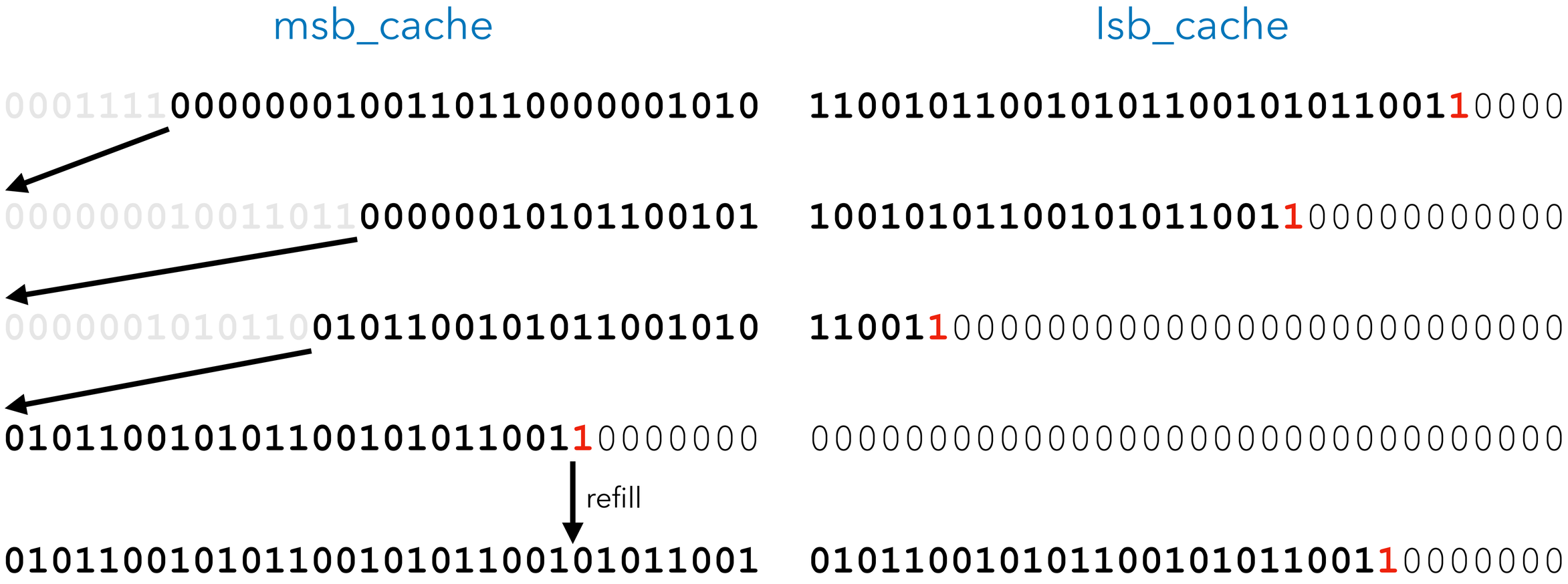
- uint64_t show_bits(ctx);
- void skip_bits(ctx, n);

Technique 1 - Piston cached bitstream reader

approaches	bits	reads	writes	notes
1 unaligned read, big-endian, 0..7 shift	57	2+1	2	branchless
2 aligned reads, big-endian, 0..63 double shift	64	2+2	2	branchless, slower shift
int64 cache, 0..63 shift, refill	32..57	2	1	
int64 cache, 0..63 shift with update, refill	32..57	2	2	shorter dependency
int64[2] cache, 0..63 double shift, refill	64	3	1	slower shift
int64[2] cache, 0..63 double shift with update, refill	64	3	3	shorter dependency, slower shift
int64[2] piston cache, 0..63 double shift with update, refill	64	2	2	shorter dependency, slower shift

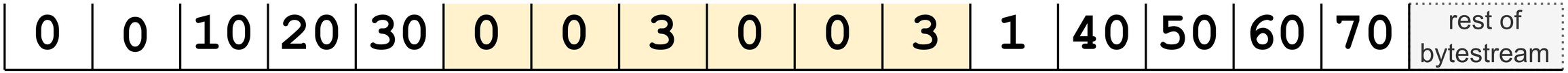
Additional design choices: pre|post-refill, inline|noinline

Technique 1 - Piston cached bitstream reader



Replace shift count with trailing set bit inside cache

Technique 2 - On-the-fly SIMD unescaping



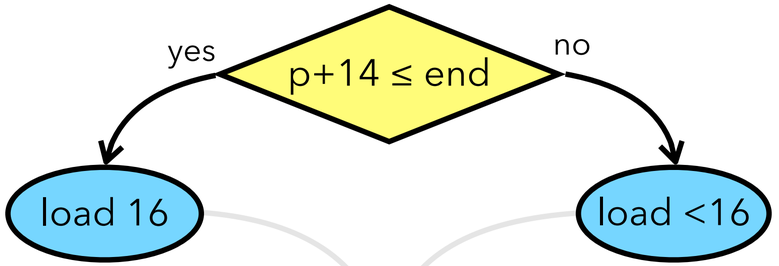
When parsing AVC bytestream: 003 → 00~~3~~

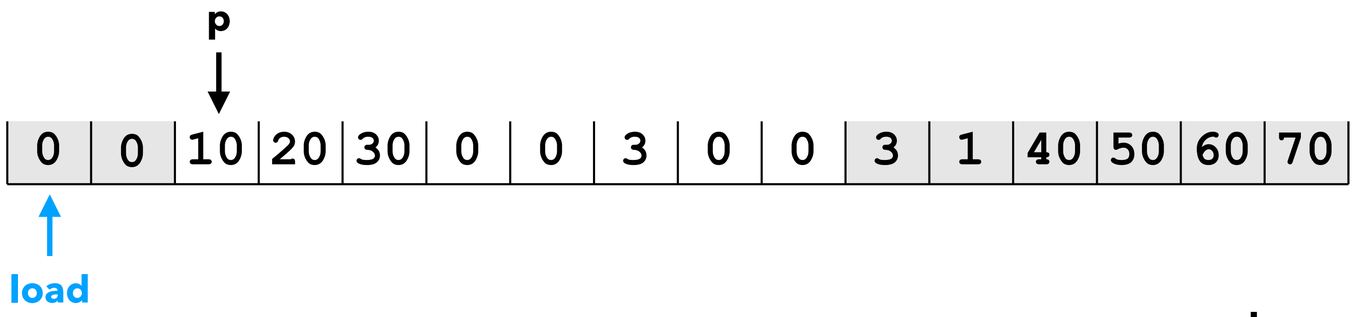
Used for escaping of 000 and 001 delimiters in MPEG-TS

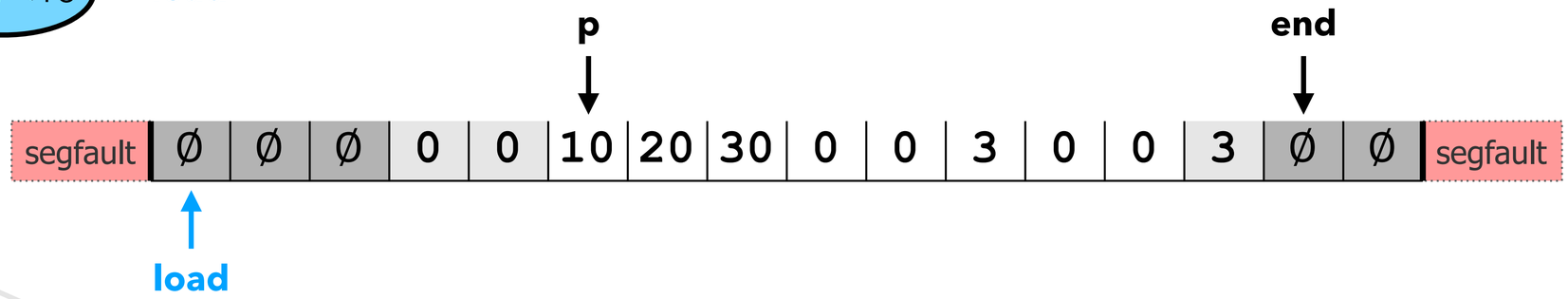
Usually: fast pre-scan to trim escape sequences

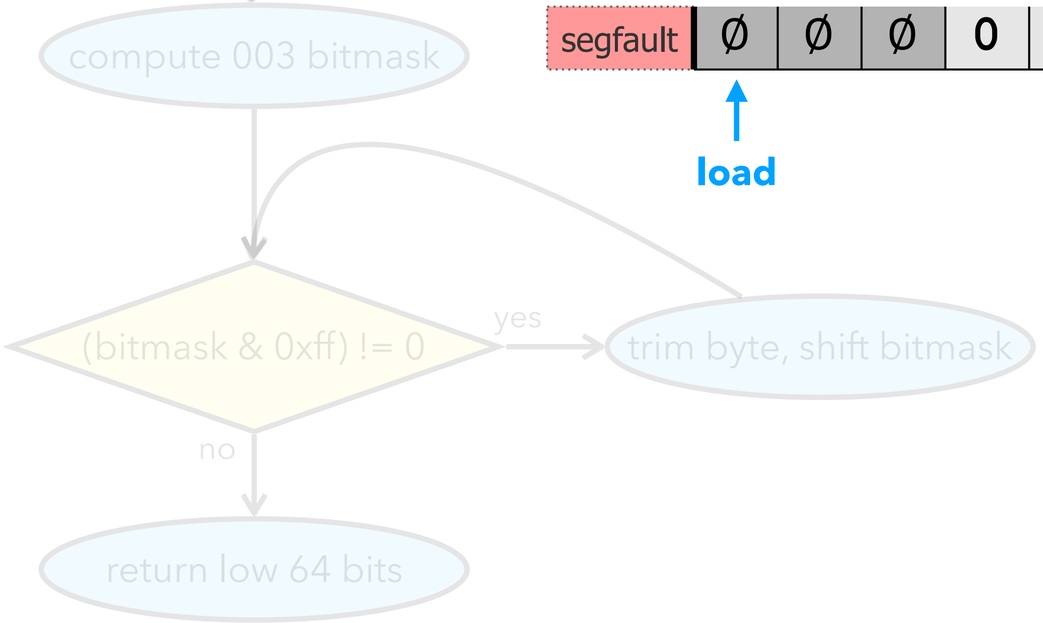
edge264: on-the-fly unescaping with SIMD at refill

Technique 2 - On-the-fly SIMD unescaping

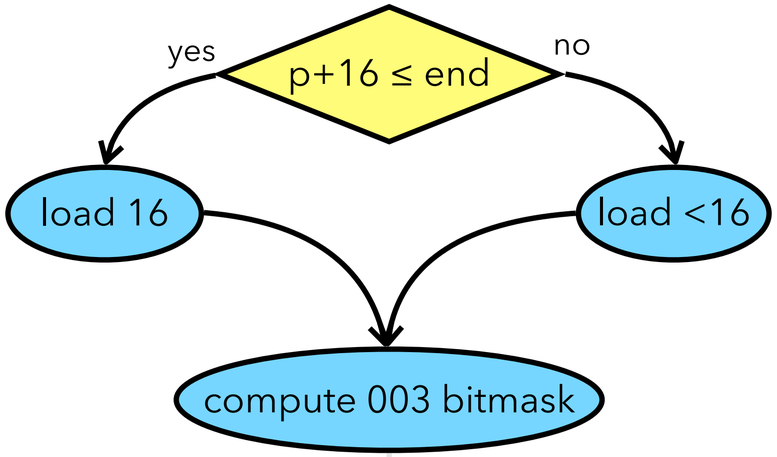


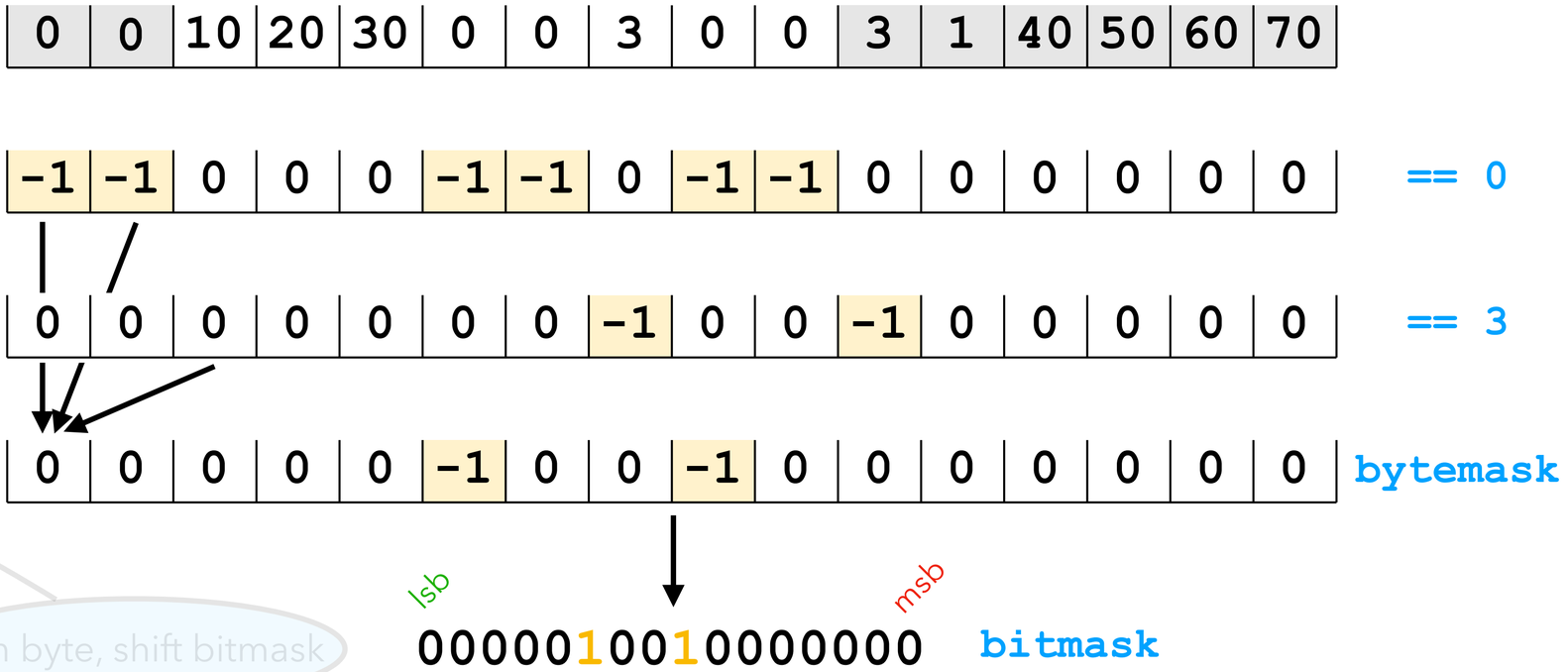


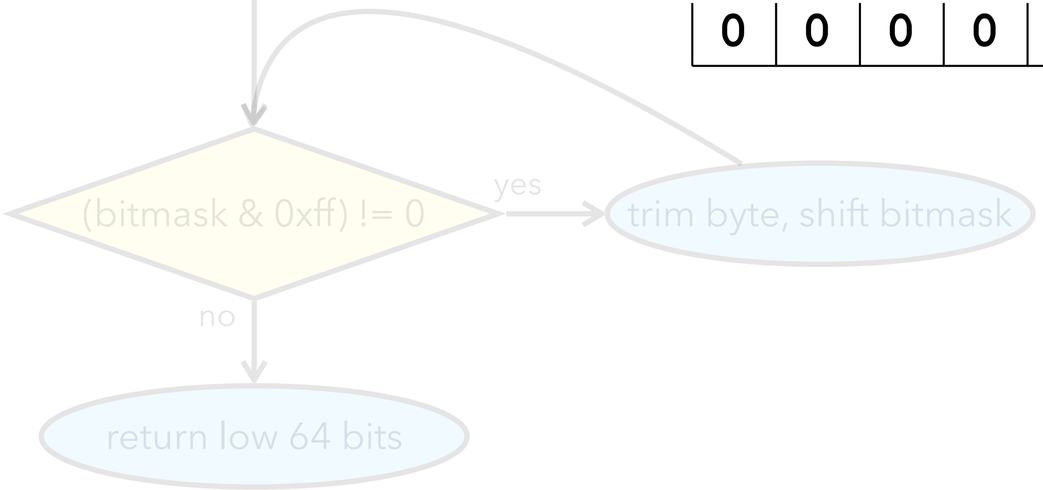




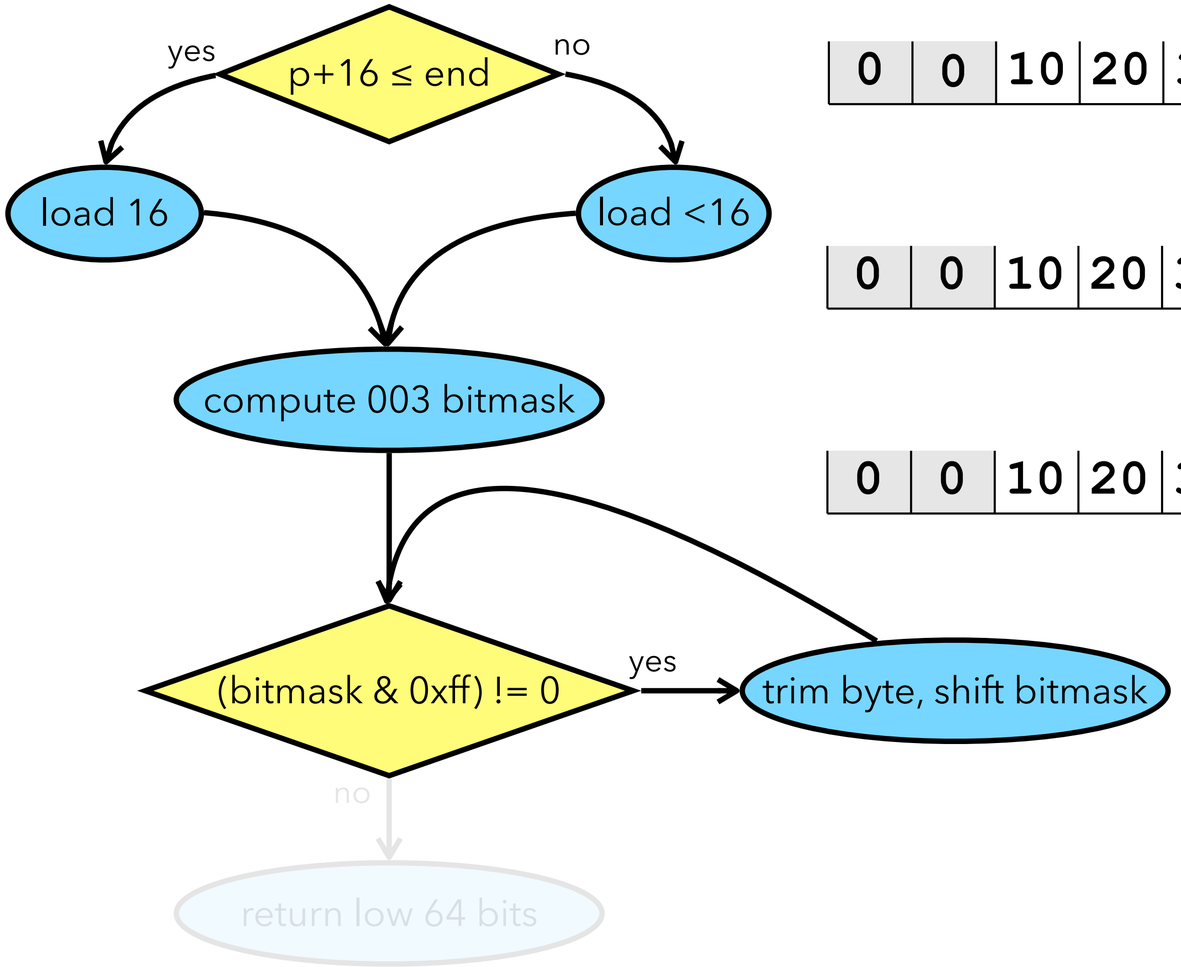
Technique 2 - On-the-fly SIMD unescaping

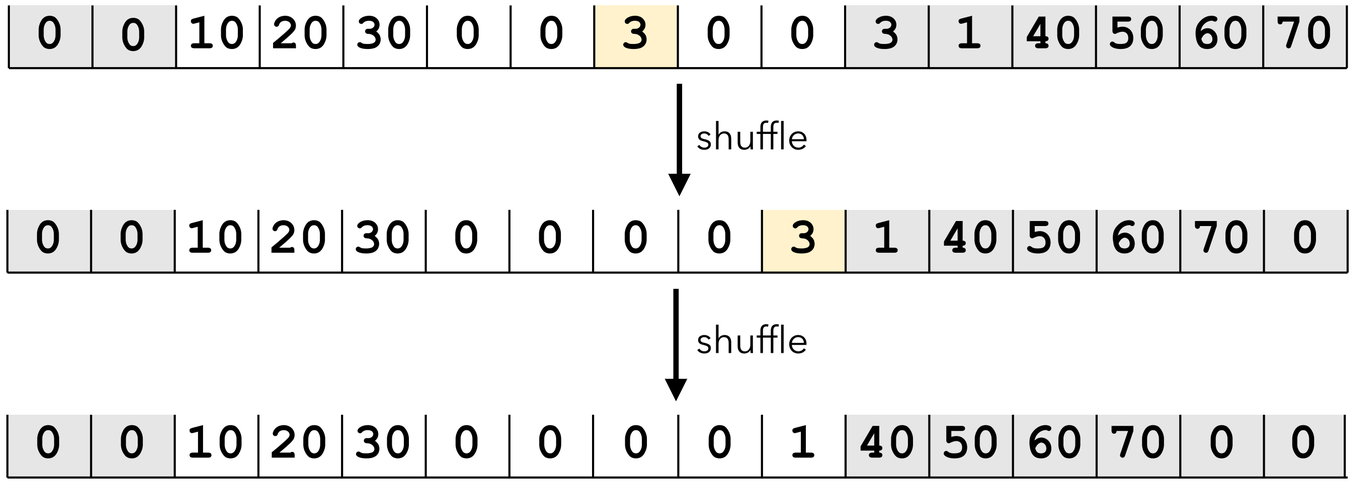


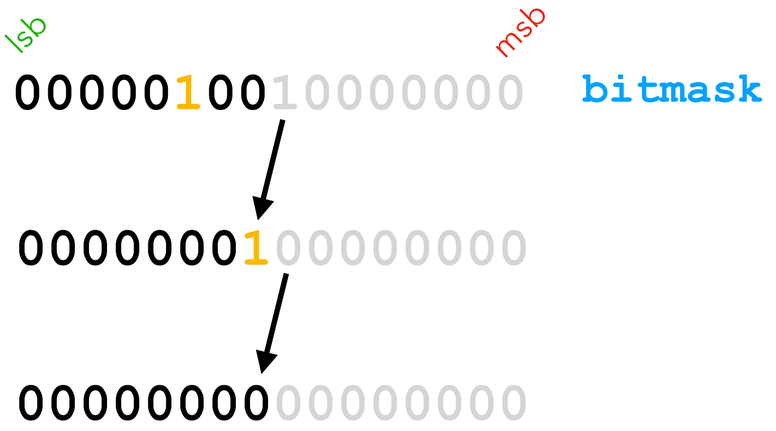




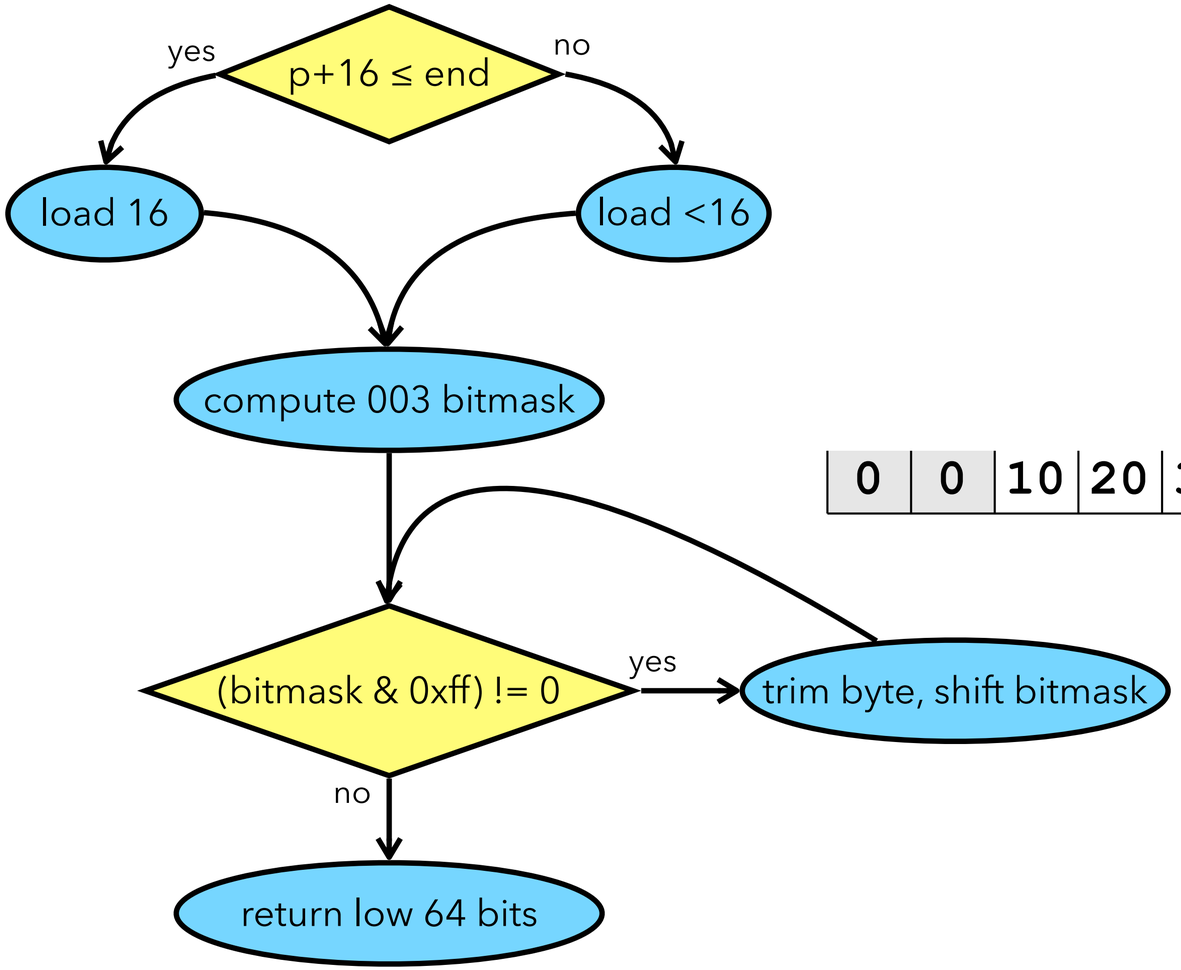
Technique 2 - On-the-fly SIMD unescaping







Technique 2 - On-the-fly SIMD unescaping



0	0	10	20	30	0	0	0	0	1	40	50	60	70	0	0

Technique 3 - Multiarch SIMD programming

edge264 uses **GCC vector extensions** in addition to **vector intrinsics**

```
typedef int8_t i8x16 __attribute__((vector_size(16)));
union { int8_t array[64]; i8x16 array_v[4]; };
#define shuffle8(a, m) (i8x16)_mm_shuffle_epi8(a, m)
```

Multiarch vs **Portable**

- Multiarch → best possible performance for *a set of* architectures
- Portable → good performance for *all* architectures

Clang vector extensions (__builtin_shufflevector, __builtin_convertvector, __builtin_elementwise, __builtin_reduce) are designed as portable → avoided

Technique 3 - Multiarch SIMD programming

Adding A64 NEON – **Workflow**

- Write unit tests using existing SSE code
- *Write best possible NEON intrinsics first*
- For each NEON instruction used, alias with SSE, else fallback to #if #else (~20% of SIMD code → Intra planar, Intra chroma DC, Inter chroma, 3/18 of Inter luma, Hadamard transforms, core deblocking filters)
- Look at files internal.h, intra.c, inter.c, residual.c, deblock.c

Technique 3 - Multiarch SIMD programming

Adding A64 NEON – **The good** (Intel/ARM)

- Aliases for same instructions (abs, min, max, alignr/extq, avg/rhadd, cvt/movl, adds/qadd, subs/qsub, unpack/zip)
- Semantic variations for close instructions (blend/bsl → ifelse_msb/ifelse_mask, shuffle/tbl → shuffle/shufflez/shufflen/shuffle2/shuffle3, sad/addv → sum8/sumh8/sumd8)
- Polyfills for key ARM instructions (dup → set1+shuffle, rshr → add+srli, qshr → srli+pack, addw → cvt+add, mull → cvt+mul)
- Polyfills for key SSE instructions (pack → qmov+qmov, movemask, shld ☹)
- Specialized helper functions for redundant code (e.g. load4x4, maddshr8)

Technique 3 - Multiarch SIMD programming

Adding A64 NEON – **The bad**

- GCC/Clang do not model multiple-latency instructions (pre/post-index, mla)
- Clang splits accumulate intrinsics instead of merely generating them (mla)
- Narrowing intrinsics return 64-bit vector whereas instructions return 128-bit
- Across-vector intrinsics (addv) return int whereas instructions return 128-bit
- Incompatible shifts by element & multiplies by element

Technique 3 - Multiarch SIMD programming

Adding A64 NEON – **The ugly**

- Inline asm to force the use of mla or make addv return to vector

```
asm("mla %0.8h, %1.8h, %2.8h": "+w"(a): "w"(b), "w"(c));
```

- Dead Code Elimination for variable shifts and multiplies by 1 element

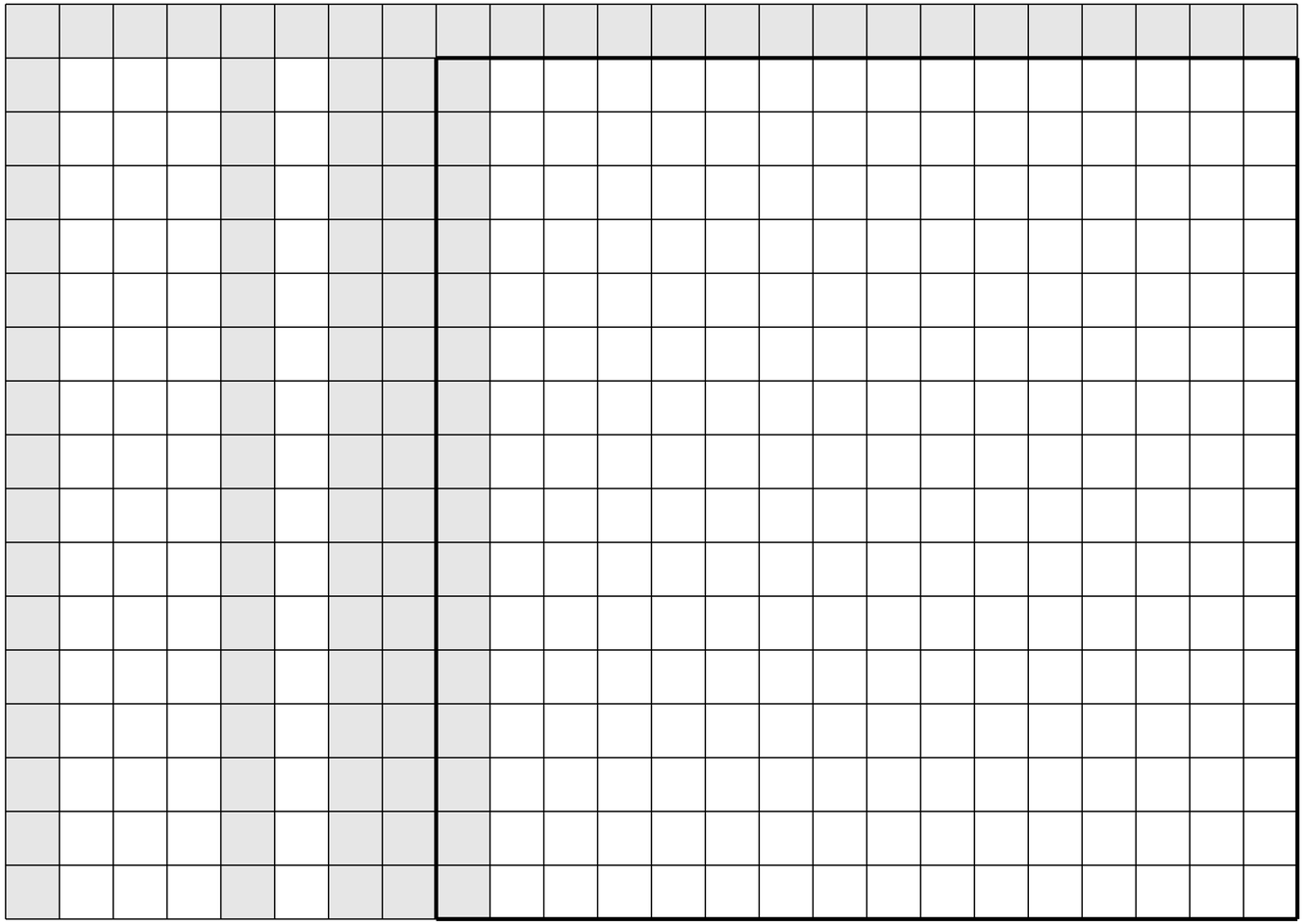
```
i64x2 wd64 = {shift}; // for SSE
i16x8 wd16 = -set16(shift); // for NEON
...
#if defined(__SSE2__)
    ... = shr16(..., wd64);
#elif defined(__ARM_NEON)
    ... = vshlq_s16(..., wd16);
#endif
```

Technique 3 - Multiarch SIMD programming

Adding A64 NEON – **The ugly**

- Address calculation

```
INIT_PX(p, stride);
...
PX(x,y) //p+x+y*stride
```

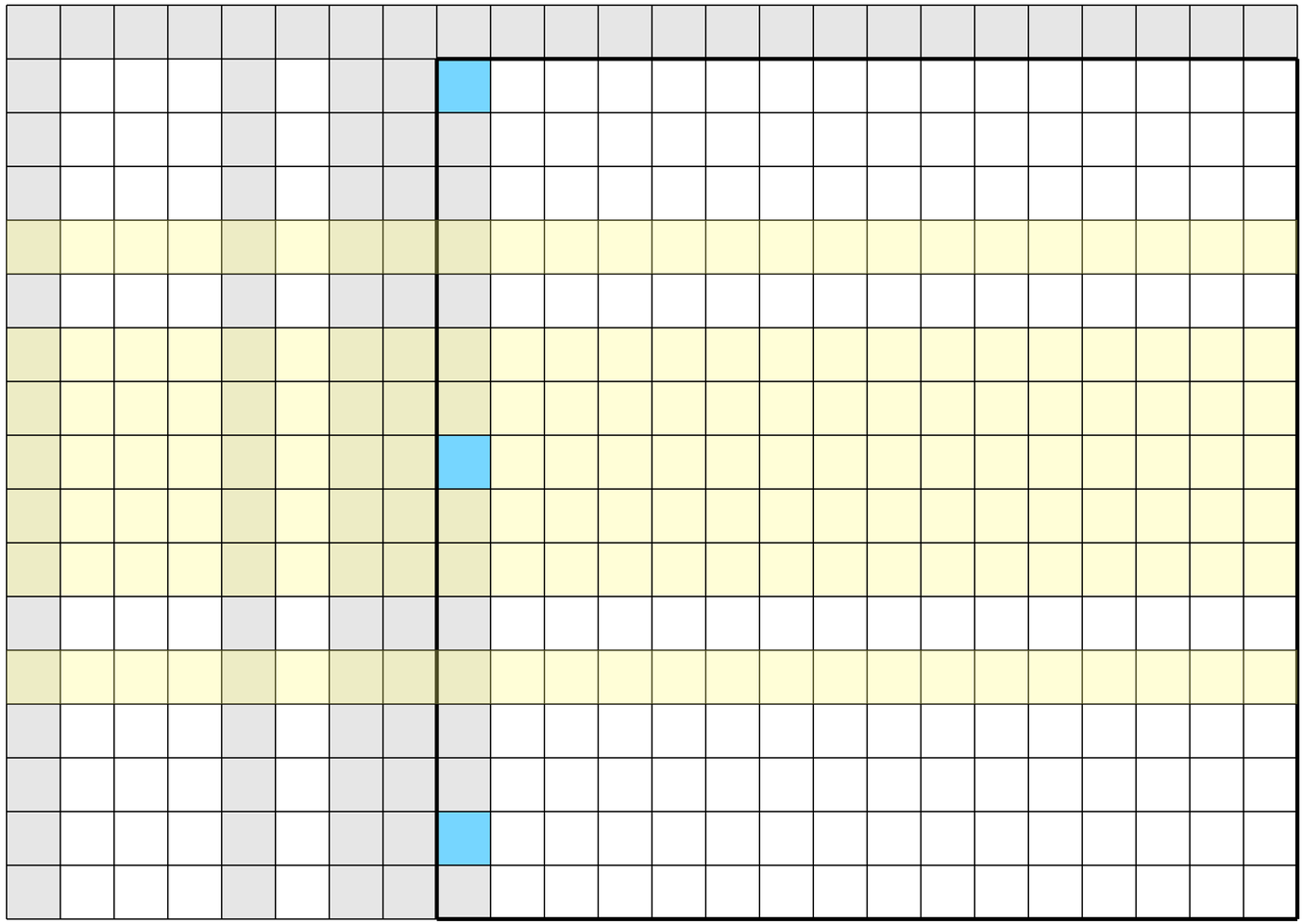


Technique 3 - Multiarch SIMD programming

Adding A64 NEON – **The ugly**

- Address calculation (Intel)

```
_p0 = p;
_p7 = p+stride*7;
_pE = p+stride*14;
_stride = stride;
_nstride = -stride;
```

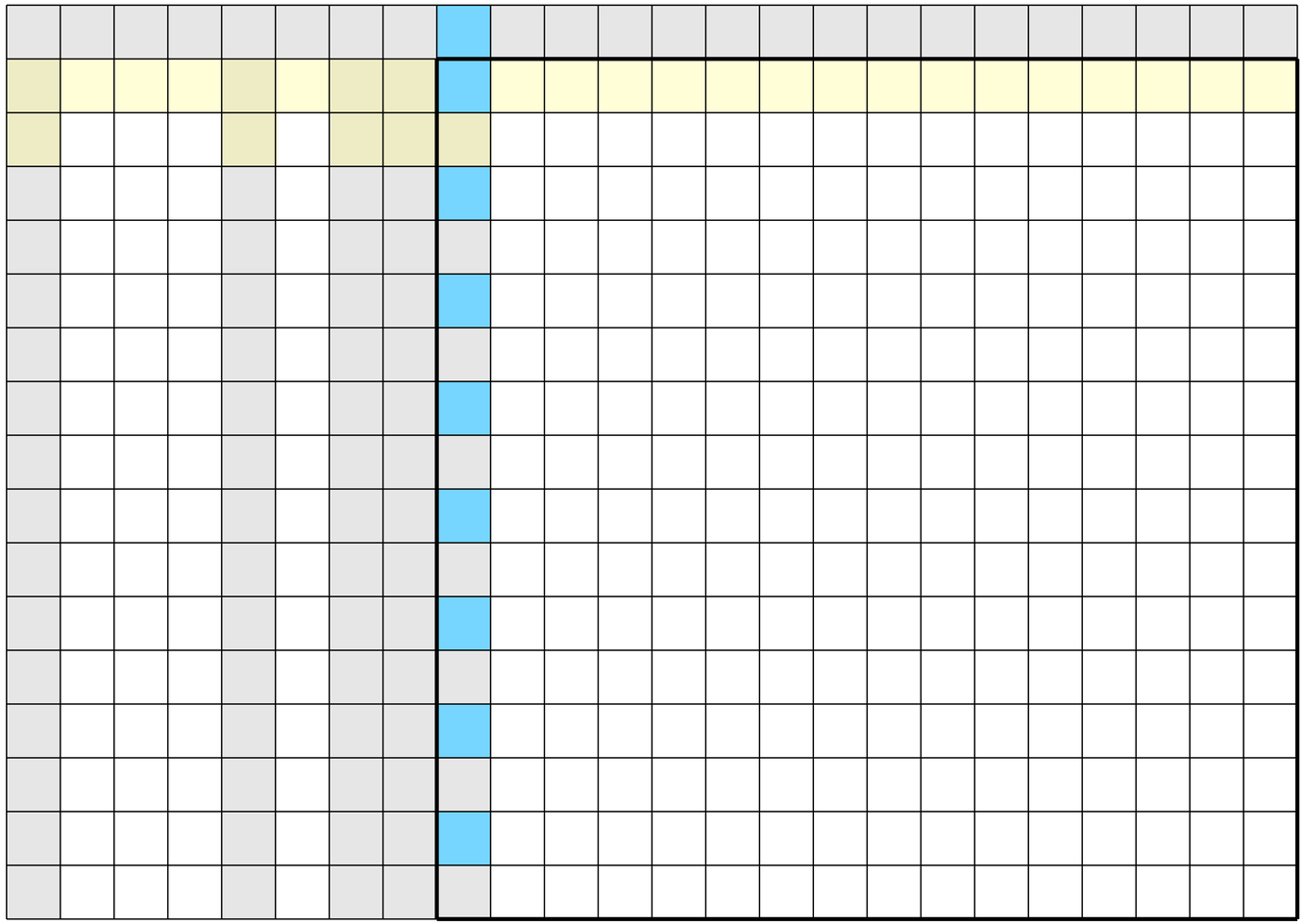


Technique 3 - Multiarch SIMD programming

Adding A64 NEON - **The ugly**

- Address calculation (ARM)

```
_pZ = p-stride;
_p0 = p;
_p2 = p+stride*2;
...
_pE = p+stride*14;
_stride = stride;
_stridem1 = stride-1;
_stridem2 = stride-2;
_stridem4 = stride-4;
_stridem8 = stride-8;
```



Technique 4 - The Structure of Arrays pattern

```
struct Frame {
    uint8_t *frame_buffer;
    int32_t picture_order_count;
    int8_t reference_flag;
    int8_t output_flag;
} frames[32];
```

```
uint8_t *frame_buffers[32];
int32_t picture_order_counts[32];
uint32_t reference_flags;
uint32_t output_flags;
```

Technique 4 - The Structure of Arrays pattern

Exploding arrays of structs into independent arrays

→ *replace loops on structures with vector operations*

Find empty frame buffer:

```
__builtin_ctz(~ctx->reference_flags & ~ctx->output_flags)
```

Count reordered (pending) frames:

```
__builtin_popcount(ctx->output_flags)
```

Loop through reference frames:

```
for (unsigned r = ctx->reference_flags; r; r &= r - 1) {
    int i = __builtin_ctz(r);
    ...
}
```

Technique 4 - The Structure of Arrays pattern

```
uint8_t *frame_buffers[32];
int32_t picture_order_counts[32];
uint32_t reference_flags;
uint32_t output_flags;
```

Pros:

- less allocations & memory fragmentation
- smaller code+binary overall

Cons:

- less intuitive
- cache pollution & false sharing in multithreading

Technique 5 - Deferred error checking

```
sps->bit_depth_luma   = get_ue_golomb_31(gb) + 8;
sps->bit_depth_chroma = get_ue_golomb_31(gb) + 8;

if (sps->bit_depth_chroma != sps->bit_depth_luma) {
    avpriv_request_sample(avctx,
                          "Different chroma and luma bit depth");
    goto fail;
}

if (sps->bit_depth_luma   < 8 || sps->bit_depth_luma   > 14 ||
    sps->bit_depth_chroma < 8 || sps->bit_depth_chroma > 14) {
    av_log(avctx, AV_LOG_ERROR, "illegal bit depth value (%d, %d)\n",
           sps->bit_depth_luma, sps->bit_depth_chroma);
    goto fail;
}
```

Technique 5 - Deferred error checking

Parsing errors are usually handled with **goto** or **return**

- Needs code to bubble up errors from sub-functions
- Many exit paths to test
- Noise for branch predictor

Error reporting needs depend on usage:

- Viewers → none
- Media players → API misuse
- Encoders → everything

Technique 5 - Deferred error checking

validator vs **parser**

- validator → checks if a stream is valid, otherwise helps fix the errors
- parser → decodes a stream if valid, otherwise fails

Drop validation & encoder users → *relax exact location of errors*

Checksums in streams also ensure we rarely see erroneous streams

High chance of desync on error with VLCs:

- `00011110000000100110110000001010110`
- `00011110000000100110110000001010110`

Technique 5 - Deferred error checking

Solution:

- ```
  sps->bit_depth_luma   = get_ue_golomb_31(gb, 6) + 8;
  sps->bit_depth_chroma = get_ue_golomb_31(gb, 6) + 8;
  ```
- Check for desync at end of header

  `10000000000000000000000000000000  00000000000000000000000000000000`

Impact:

- no branches inside parsing code (except refill)
- check for unsupported stuff once at the end of headers
- less exit paths to test & care about
- return values need not bubble errors anymore (slices only)
  ```

Roadmap for release

- Finish multithreading
- Fuzzying & Continuous Integration
- VLC/ffmpeg/GStreamer plugins
- ARM32

Thank you for your attention!

https://github.com/tvlabs/edge264

(Now looking for a job! traf@ik.me)

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

Agner Fog's Intel optimization resources https://www.agner.org/optimize/

Arm Neon Intrinsics https://arm-software.github.io/acle/neon_intrinsics/advsimd.html

Arm A64 ISA (for bitcodes) https://developer.arm.com/documentation/ddi0602/

Cortex-A76 Timings https://developer.arm.com/documentation/pjdoc466751330-7215/

Bit twiddling hacks https://graphics.stanford.edu/~seander/bithacks.html

Programming language needs

- Accessing integers as bit arrays
- Slicing of vectors
- Expressing alternate code blocks
- Expressing procedures
- Custom ABI (multiple return regs, in-place updates, callee/caller-saved)
- Unsigned operators in addition to unsigned variables
- Global context storage (%rbp in non-recursive call stack)
- Explicit variables for condition flags (so functions can return conditions)
- Alias of variable to memory
- Automatic struct reordering for cache locality based on profiling

C vs ASM

C over ASM:

- Easier to recode → incremental gains with new algorithms
- Higher mental threshold to design complex algorithms
- Built-in model of CPU pipelines → efficient scheduling with new CPUs

ASM over C:

- Writing code unobtainable in C (pre/post-index, custom ABI)
- Deterministic result, doesn't degrade with new compiler versions