PARSING GIGABYTES OF JSON PER SECOND

WONG DING FENG

Created: 2025-02-06 Thu 23:59

TABLE OF CONTENTS

1. OBJECTIVES

- Describe the real problem with JSON
    - Techniques and strategies to make it fast
- Simple primer on bitwise operations and simd
- simdjson architecture
- How this is used to make simdjson

2. PROBLEM

2.1. HOW FAST IS YOUR HARDWARE

CPU: AMD Eng Sample: 100-000000534-40_Y: 4.6 GHz Network: 1Gb~100MB/s, 10Gb~1000MB/s

One core can read direct text at this speed:

```
Testing dd read:
1280+0 records in
1280+0 records out
10737418240 bytes (11 GB, 10 GiB) copied, 3.88076 s, 2.8 GB/s
```

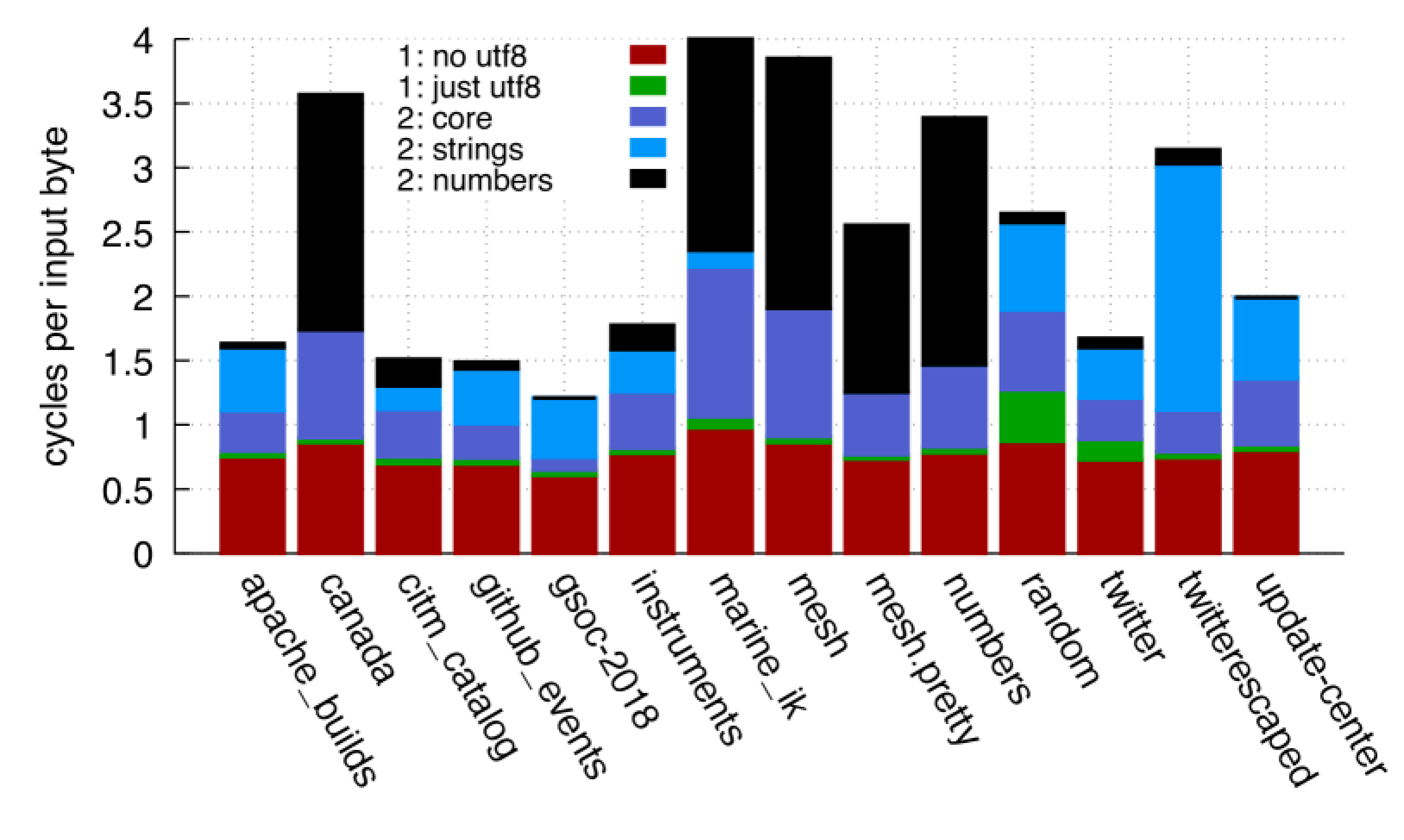
2.2. WHY IS JSON INTERESTING?

- most data is in json
- basically double the speed means you can parse 2x as much data
    - duplicating hardware for free

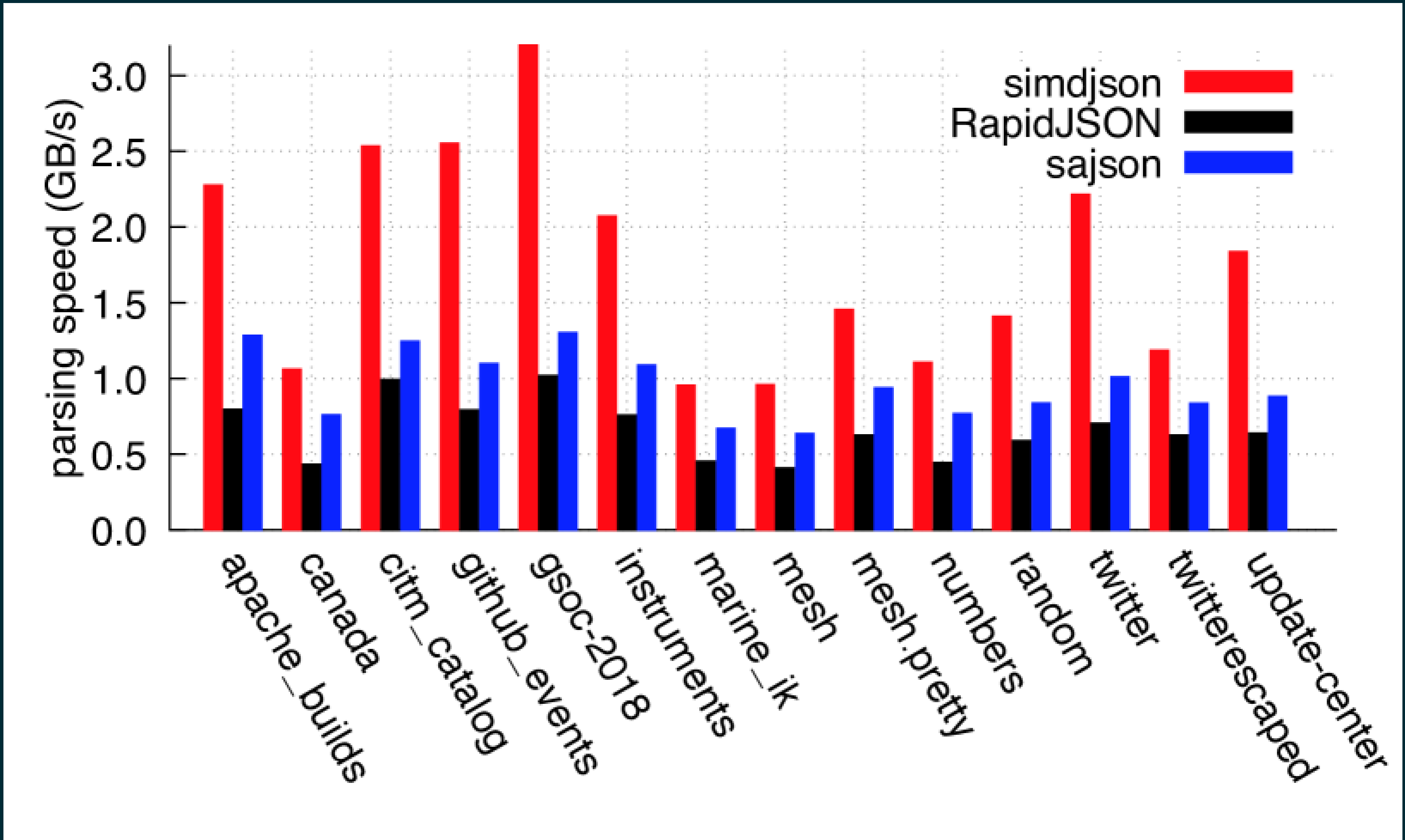
parser	Skylake	Cannon Lake	speed
simdjson	1.4	1.3	fast
RapidJSON	0.56	0.44	slow
sajson	0.93	0.84	normal

2.3. WHY YOU SHOULD BE INTERESTED

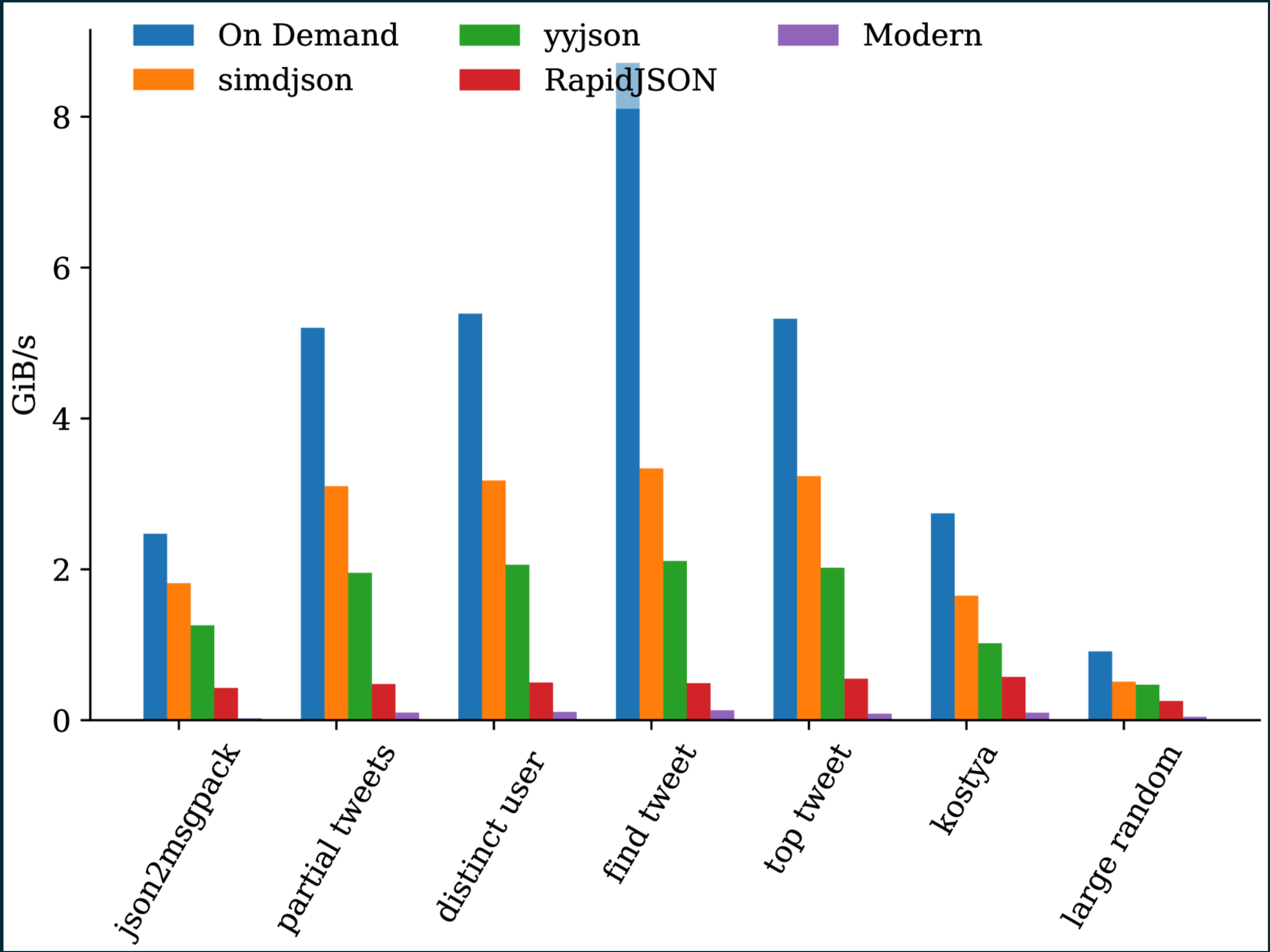
- configurable, increase speed



2.4. AGAINST OTHERS



2.5. ON DEMAND JSON



On Demand
simdjson
yyjson
RapidJSON
Modern
GiB/s
0
2
4
6
8
json2msgpack
partial tweets
distinct user
find tweet
top tweet
kostya
large random

3. IDEAS ON HOW TO DO IT FAST?

3.1. STRATEGIES

- depends on the usage pattern

3.1.1. QUERY INTENSIVE

- Create a database (ElasticSearch, MongoDB, PostgresSQL)
  - create a KV store
  - load once and query it

3.1.2. SELECTIVE PARSING

- Selective parsing
  - NoDB
    - query the data without parsing it, without loading into a DB
    - like grep
  - JIT techniques
    - find patterns and repetitive structures, compile the code for the specific query
    - like a compiler
  - **Mison** (by Microsoft)
    - selective parsing, jump directly to the field you want
    - use SIMD to find structural important characters like "

3.2. WHAT IS FAIR GAME?

- Types of json parsing
  - Non-validating json parser
    - assume the input is valid
    - easier
    - most selective parsing is non-validating
  - Validating json parser
    - check the input is valid
    - no assumptions or malformed input
      - security risk
      - its just wrong number or string being parsed
    - harder more complex

3.3. PROPER DEFINITION OF JSON

```
/* JSON EBNF Grammar Specification */

/* Root JSON structure */
json = ws , (object | array) , ws ;

/* Objects */
object = "{" , ws , [ members ] , ws , "}" ;
members = pair , { "," , ws , pair } ;
pair = string , ws , ":" , ws , value ;

/* Arrays */
array = "[" , ws , [ elements ] , ws , "]" ;
elements = value , { "," , ws , value } ;

/* Values */
value = string | number | object | array | "true" | "false" | "null" ;

/* Strings */
string = '"' , { char | escape } , '"' ;
char = ? any Unicode character except " or \ or control characters ? ;
escape = "\" , ('"' | "\" | "/" | "b" | "f" | "n" | "r" | "t" | unicode) ;
unicode = "u" , hexdigit , hexdigit , hexdigit , hexdigit ;
hexdigit = digit | "A" | "B" | "C" | "D" | "E" | "F" | "a" | "b" | "c" | "d" | "e" | "f"

/* Numbers */
number = [ "-" ] , (zero | integer) , [ fraction ] , [ exponent ] ;
integer = nonzero , { digit } ;
nonzero = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
digit = "0" | nonzero ;
zero = "0" ;
```

```
fraction = "." , digit , { digit } ;
exponent = ("E" | "e") , [ "+" | "-" ] , digit , { digit } ;

/* Whitespace */
ws = { whitespace } ;
whitespace = " " | "\t" | "\n" | "\r" ;

/* Comments and Explanation */
```

3.4. MY REDUCED DEFINITION OF JSON

```
json_value ::= primitive_value | container_value

primitive_value ::= boolean | string | number | null

boolean ::= "true" | "false"

string ::= " utf8_char* "
utf8_char ::= ascii_char | unicode_char

number ::= integer | decimal | scientific
integer ::= ["+" | "-"] digit+
decimal ::= integer "." digit+
scientific ::= decimal "e" ["+"|"-"] digit+
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

null ::= "null"

container_value ::= object | array

object ::= "{" string:json_value, string:json_value "}"

array ::= "[" json_value, json_value "]"

(* Notes:
   - Integer limits: [-2^53+1 to 2^53-1] for safe integers
   - Scientific notation limit: approximately ±2^1024
   - NaN and Infinity are not valid JSON numbers
   - Strings must be UTF-8 encoded
*)
```

3.5. STRONGLY DEFINE: BOOL, STRING, NUMBER, NULL, OBJECT AND ARRAY

```
data JsonValue
    = Primitive PrimitiveValue
    | Container ContainerValue

-- 6 primitives -------------------------
data PrimitiveValue
    = Boolean Bool
    | String Text
    | Number Double
    | Null

data ContainerValue
    = Object Object
    | Array Array
-- END ----------------------------------

newtype Object = Object [(Text, JsonValue)]

newtype Array = Array [JsonValue]
```

3.6. STRONGLY DEFINE: BOOL, STRING, NUMBER, NULL, OBJECT AND ARRAY

3.6.1. NUMBER LIMITS AND INTEGERS

```
// 1. Integer Limits
const INTEGER_EXAMPLES = {
    // Maximum safe integer in JavaScript (2^53 - 1)
    max_safe_integer: 9007199254740991,
    // Minimum safe integer in JavaScript (-(2^53 - 1))
    min_safe_integer: -9007199254740991,

    // Zero representations
    zero: 0,
    negative_zero: -0,  // JSON preserves negative zero

    // Common boundary values
    max_32bit_int: 2147483647,
    min_32bit_int: -2147483648,

    // Integer examples
    positive: 42,
    negative: -42
};
```

3.6.2. FLOATS AND SCIENTIFIC NOTATION

```
// 2. Floating Point Examples
const FLOAT_EXAMPLES = {
    // Precision examples (up to 15-17 significant digits)
    high_precision: 1.234567890123456,

    // Edge cases
    very_small_positive: 2.2250738585072014e-308, // Near smallest possible double
    very_large_positive: 1.7976931348623157e+308  // Near largest possible double
};

// 3. Scientific Notation Examples
const SCIENTIFIC_NOTATION = {
    // Positive exponents
    large_scientific: 1.23e+11,
    very_large: 1.23E+308,  // Note: Both 'e' and 'E' are valid

    // Negative exponents
    small_scientific: 1.23e-11,
    very_small: 1.23E-308,

    // Zero with exponent
    zero_scientific: 0.0e0,

    // Various representations
    alternative_forms: {
        standard: 1230000000,
        scientific: 1.23e9,
        another_form: 123e7
    }
};
```

3.7. STRING: HANDLE ESCAPED QUOTES AND UTF-8

- some lazy parsers assume ascii for simplicity
  - 128 possibilities, 8 bits only
  - assume that input does not have japanese or chinese or weird characters
- RFC standard says strings are UTF-8
- escaped double quotes “Tom said: \”hello\“.”
  - Tom said: “hello”.
  - number of ’\’
    - odd -> escaped, “\”“ -> ”
    - even -> not escaped, “\\” -> \
- outside of '',there can only be 4 types of white space
  - “ ” | “\t” | “\r” | “\n”

3.7.1. ASCII CODE

- code ponits 0x00 - 0xEF 127 possibilities

b7 b6 b5 →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
Bits b4 ↓	b3 ↓	b2 ↓	b1 ↓	Column → / Row ↓	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	´	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	¦
1	1	0	1	13	CR	GS	—	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

3.7.2. UTF-8

```
Single byte (ASCII):
0xxxxxxx                     (values 0-127)
Values start with 0, remaining 7 bits for data

Two bytes:
110xxxxx 10xxxxxx           (values 128-2047)
First byte starts with 110

Three bytes:
1110xxxx 10xxxxxx 10xxxxxx  (values 2048-65535)
First byte starts with 1110

Four bytes:
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx   (values 65536+)
First byte starts with 11110
```

- normally outside of strings, no bytes start with 1 in front.

3.8. SUMMARY OF REQUIREMENTS

- numbers
  - negative
  - floats
- string
  - utf-8
  - escaped quotes \"
- Rest of structure well formed
  - valid whitespace
  - valid bracket {}, []

4. CHALLENGES

4.1. WRITING A PARSER FOR IT

- Recursive Descent type parser
- Many if else required, is it possible to do it without any branches?

```python
def peek_token_type(json_str, index):
    char = json_str[index]

    # Skip whitespace
    while index < len(json_str) and is_whitespace(char):
        index += 1
        char = json_str[index]

    # Check data type based on first character
    if char == '{':
        return 'object'
    elif char == '[':
        return 'array'
    elif char == '"':
        return 'string'
    elif is_digit(char):
        return 'number'
    elif char == 't' or char == 'f':
        return 'boolean'
    elif char == 'n':
        return 'null'
```

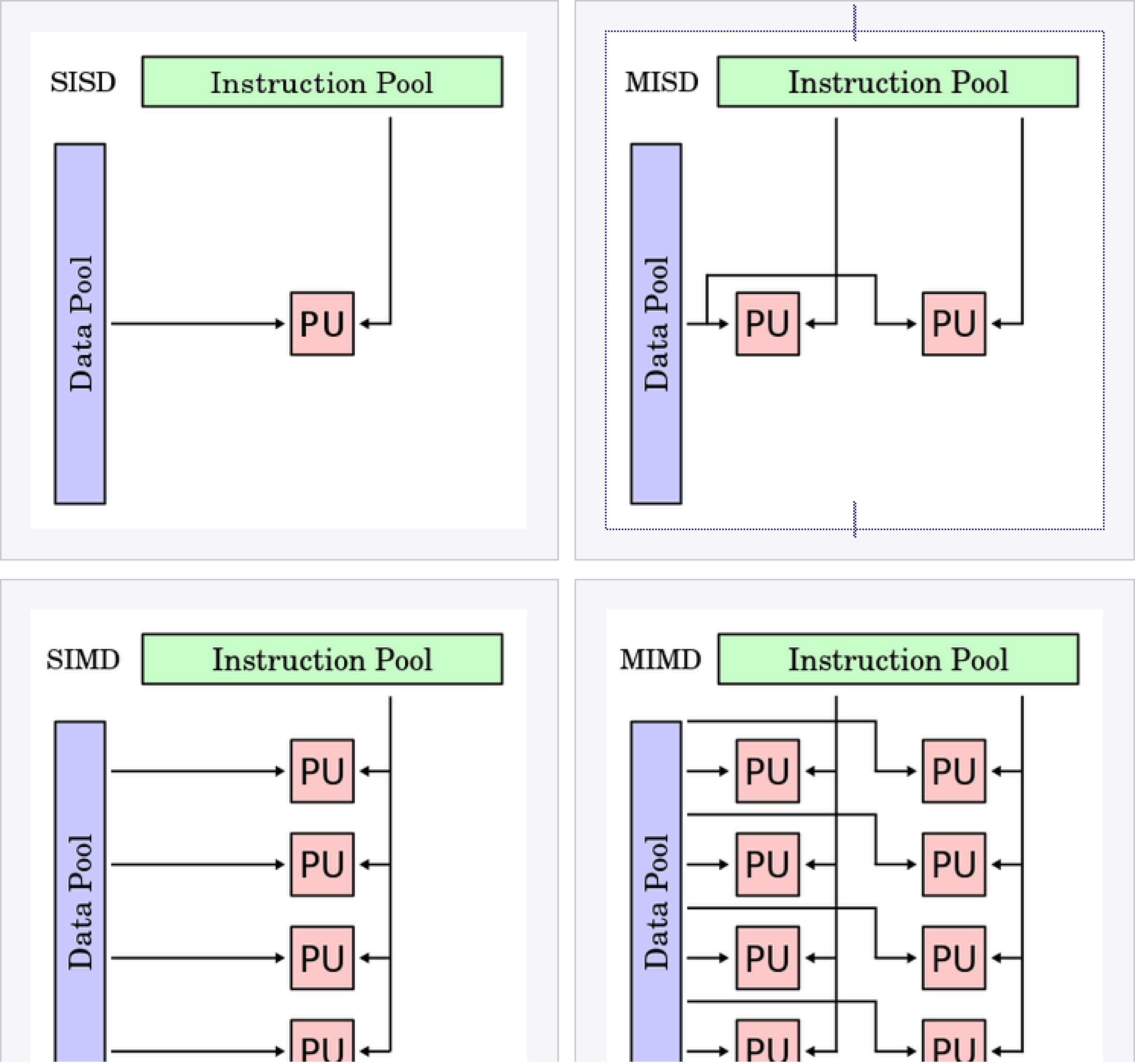
4.2. GIVEN THE CHALLENGE, HOW TO DO IT FAST?

- Parallelization, split work across multiple cores.
- SIMD, process more than 8 bytes at a time.
    - Branchless code, no if statements. CPU missed branch prediction.
        - correct, 0-1 cycles
        - branch miss, 20 cycles

5. ABOUT SIMD

how does simd fit into all of this?

5.1. WHAT IS SIMD


SISD
Instruction Pool
Data Pool
PU
MISD
Instruction Pool
Data Pool
PU
PU
SIMD
Instruction Pool
Data Pool
PU
PU
PU
PU
MIMD
Instruction Pool
Data Pool
PU
PU
PU
PU
PU
PU
PU
PU

5.2. SIMD EXAMPLE

```
Adding 4 numbers simultaneously:

Scalar:
A: [5] + [3] = [8]      Step 1
B: [7] + [2] = [9]      Step 2
C: [4] + [6] = [10]     Step 3
D: [1] + [8] = [9]      Step 4

SIMD:
[5|7|4|1] +
[3|2|6|8] =   Step 1
[8|9|10|9]    Done!
```

5.3. CPU

```
Year:           2010          2013          2019
Architecture: Westmere  ->  Haswell   ->  Ice Lake
Process:        32nm          22nm          10nm
Vector ISA:     SSE2      ->  AVX2      ->  AVX512
Vec Width:      128-bit       256-bit       512-bit
               (16 bytes)    (32 bytes)    (64 bytes)
```

- Streaming SIMD Extensions
  - XMM0-XMM15
- Advanced Vector Extensions 2
  - YMM0-YMM15
- Advanced Vector Extensions 512
  - ZMM0-ZMM15

5.4. SIMD CODE IS NOT THAT SCARY

Westmere uses 128-bit SSE instructions (_mm_shuffle_epi8) Haswell uses 256-bit AVX2 instructions (_mm256_shuffle_epi8) Ice Lake uses 512-bit AVX-512 instructions (_mm512_shuffle_epi8)

```
// Westmere
const uint64_t whitespace = in.eq({
    _mm_shuffle_epi8(whitespace_table, in.chunks[0]),
    _mm_shuffle_epi8(whitespace_table, in.chunks[1]),
    _mm_shuffle_epi8(whitespace_table, in.chunks[2]),
    _mm_shuffle_epi8(whitespace_table, in.chunks[3])
});

// Haswell (2 x 256-bit chunks)
const uint64_t whitespace = in.eq({
    _mm256_shuffle_epi8(whitespace_table, in.chunks[0]),
    _mm256_shuffle_epi8(whitespace_table, in.chunks[1])
});

// Ice Lake (1 x 512-bit chunk)
const uint64_t whitespace = in.eq({
    _mm512_shuffle_epi8(whitespace_table, in.chunks[0])
});
```

5.5. WHEN SIMD SHINES

- Regular, predictable data patterns
- Simple mathematical operations
- Continuous blocks of memory
- Identical operations on multiple data points
- High throughput

```
Perfect for SIMD:
[1|2|3|4] × 2  = [2 |4 |6 |8 ] ✓
[R|G|B|A] + 10 = [R'|G'|B'|A'] ✓
```

5.6. SIMD'S ACHILLES HEEL: BRANCHING

- if logic is complex like in parsing unable to do simd

```
if (char_at == '{') {
    return "object";
} else if (char_at == '[') {
    return "array";
} else if (char_at == '"') {
    return "string";
} else if (is_digit(char_at)) {
    return "number";
} else if (char_at == 't' || char_at == 'f') {
    return "boolean";
} else if (char_at == 'n') {
    return "null";
} else {
    throw std::invalid_argument(
        "Invalid JSON character at position " +
        std::to_string(index) +
        ": " + char_at
    );
}
```

5.6.1. CORRECT BRANCH PREDICTION

```
IF = Instruction Fetch
ID = Instruction Decode
EX = Execute
MEM = Memory Access
WB = Write Back

Time →
1  2  3  4  5  6  7  8  9
IF ID EX ME WB          | Instruction 1 (branch)
   IF ID EX ME WB       | Instruction 2 (correctly predicted)
      IF ID EX ME WB    | Instruction 3
         IF ID EX ME WB | Instruction 4
```

5.6.2. BRANCH PREDICTION MISS

- example cost 3 cycles but real cpu cost 7-15 cycles

```
Time →
1  2  3  4  5  6  7  8  9 10 11 12 13
IF ID EX ME WB                | Instruction 1 (branch)
   IF ID EX -- -- --          | Instruction 2 (wrong path)
      IF ID -- -- --          | Instruction 3 (wrong path)
         IF -- -- --          | Instruction 4 (wrong path)
            IF ID EX ME WB    | Correct Instruction 2
               IF ID EX ME WB | Correct Instruction 3
```

5.6.3. ARITHMETIC BOOLEANS

- actually LLVM does this for you when you do -o2 and -o3

```
// Example 1: Arithmetic with booleans
bool condition = true;
int a = 10;
int b = 20;

// Branched version
int x;
if (condition) {
    x = a;
} else {
    x = b;
}
std::cout << x << std::endl;  // Output: 10

// Branchless version 1
x = condition * a + (!condition) * b;
// Step by step:
// true * 10 + (!true) * 20
// 1 * 10 + 0 * 20
// 10 + 0 = 10
std::cout << x << std::endl;  // Output: 10

// Branchless version 2
x = b + (a - b) * condition;
// Step by step:
// 20 + (10 - 20) * true
// 20 + (-10) * 1
```

```cpp
// 20 - 10 = 10
std::cout << x << std::endl;  // Output: 10
```

5.6.4. SELECTION INDEXING

- actually LLVM does this for you when you do -o2 and -o3

```
// Example 2: Tuple indexing
bool condition = true;
int a = 10;
int b = 20;

// Branched version
int x;
if (condition) {
    x = a;
} else {
    x = b;
}
std::cout << x << std::endl;  // Output: 10

// Branchless version
std::array<int, 2> values = {b, a};  // Note: array order is {b, a} to match Python'
x = values[condition];
// Step by step:
// {20, 10}[true]
// {20, 10}[1]     // true converts to 1
// 10
std::cout << x << std::endl;  // Output: 10

return 0;
```

5.6.5. IF LLVM DOES IT FOR YOU, WHATS THE POINT?

- LLVM does it’s best, but it cannot find everything
  - good at small cases
- some larger complex patterns
  - human pattern recognition
  - batching operations you can use simd

5.7. WRITE BRANCHLESS CODE (BITWISE OPERATIONS)

5.7.1. TRICKY MEMORY LAYOUT

```
number = 305,419,896
number << 1 # shift left logical
Number: 305,419,896
Hex: 0x12345678
Physical Memory Layout (lowest bit → highest bit)
   Addr Low                                  Addr High
     0x1200                                     0x1203
        |                                         |
        v                                         v
Before: 00011110 01101010 00110100 00010010
           ↓↓↓↓↓    ↓↓↓↓↓    ↓↓↓↓↓    ↓↓↓↓↓
After:  00001111 00110101 00010110 00100100
        ↑
        0 enters
Decimal: 610,839,792
Hexadecimal: 0x2468ACF0
```

5.7.2. MASKING

```
a = 00001111
b = 11111100

and_op = a & b
and_op = 00001100

 or_op = a | b
 or_op = 11111111

xor_or = a ^ b
xor_or = 11110011
```

5.7.3. UNSET RIGHT MOST BIT(BLSR)

```
s = s & (s-1)

a =     00101100
b =     (a - 1)
a =     00101100
b =     00101011
a & b = 00101000
// rightmost bit is unset
```

- common cpu operation, compiler optimize to `blsr`

5.8. LLVM COMPILER

5.8.1. LLVM

```
      Frontend          Middle-end                Backend
             ↓                 ↓              ↙         ↘
Source Code → LLVM IR → [Optimized IR] → [Assembly] → Machine Code
                                     ↘_______________↗
                                     (direct path option)
```

5.8.2. WITHOUT LLVM IR

```
Without LLVM IR (n*m: 3 languages × 3 targets = 9 compilers)
---------------------------------------------------------
C++    ----→  x86_64
      \---→  AMD
       \--→  ARM

Rust   ----→  x86_64
      \---→  AMD
       \--→  ARM

Haskell --→  x86_64
        \-→  AMD
         \→  ARM

Each arrow represents a separate compiler frontend+backend (9 total)
```

5.8.3. WITH LLVM IR

```
With LLVM IR (n+m: 3 frontends + 3 backends = 6 components)
--------------------------------------------------------

            ╭→ x86_64
C++    ╮    │
       ↓    │
Rust   ━━→ IR ━━→ AMD
       ↑    │
Haskell╯    │
            ╰→ ARM

            ┊
            ↓
    Shared Optimizations
    - Dead code elimination
    - Loop vectorization
    - Constant propagation
    - And many more...
```

5.8.4. INTERMEDIATE REPRESENTATION EXAMPLE(IR)

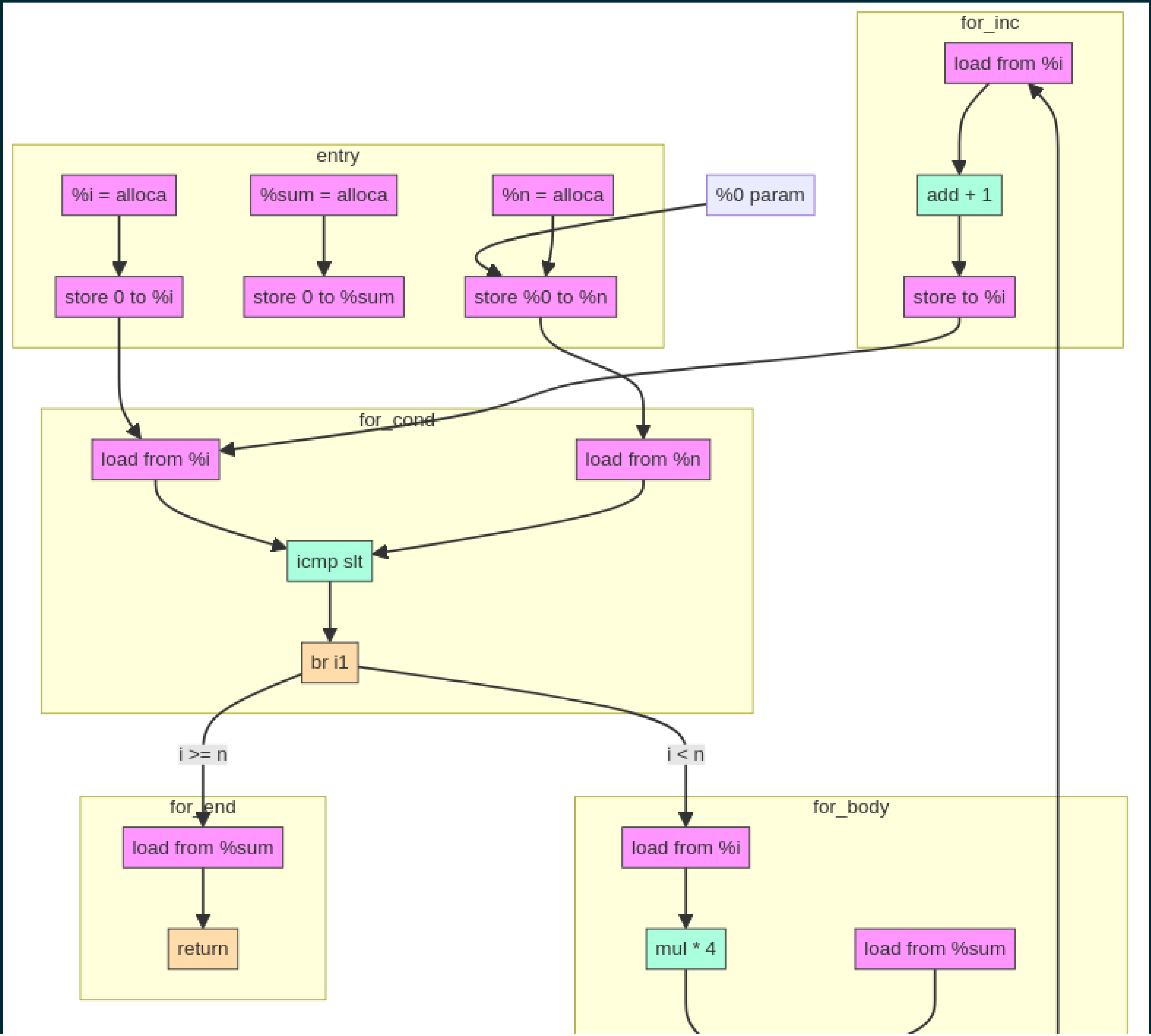
```
int example2(int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += i * 4;  // Multiplication in loop
    }
    return sum;
}
```

5.8.5. UNOPTIMIZED IR -O0

```
define dso_local i32 @_Z8example2i(i32 %0) {
entry:
  %n = alloca i32, align 4
  %sum = alloca i32, align 4
  %i = alloca i32, align 4
  store i32 %0, ptr %n, align 4
  store i32 0, ptr %sum, align 4
  store i32 0, ptr %i, align 4
  br label %for.cond

for.cond:
  %1 = load i32, ptr %i, align 4
  %2 = load i32, ptr %n, align 4
  %cmp = icmp slt i32 %1, %2
  br i1 %cmp, label %for.body, label %for.end

for.body:
  %3 = load i32, ptr %i, align 4
  %mul = mul nsw i32 %3, 4
  %4 = load i32, ptr %sum, align 4
  %add = add nsw i32 %4, %mul
  store i32 %add, ptr %sum, align 4
  br label %for.inc

for.inc:
  %5 = load i32, ptr %i, align 4
  %inc = add nsw i32 %5, 1
  store i32 %inc, ptr %i, align 4
  br label %for.cond

for.end:
```

```
  %6 = load i32, ptr %sum, align 4
  ret i32 %6
}
```

5.8.6. UNOPTIMIZED IR -O0 GRAPH

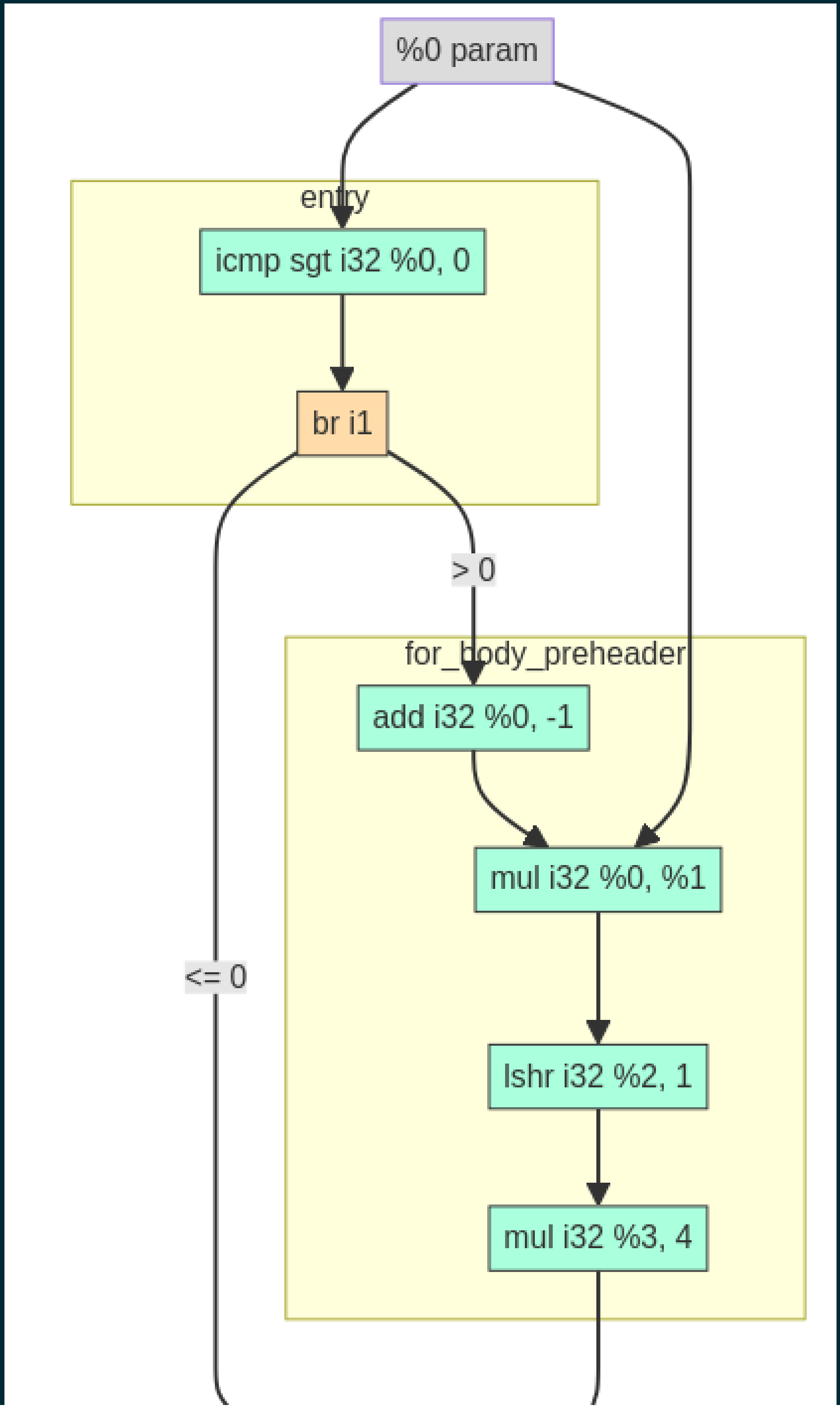


for_inc
load from %i
add + 1
store to %i
entry
%i = alloca
%sum = alloca
%n = alloca
%0 param
store 0 to %i
store 0 to %sum
store %0 to %n
for_cond
load from %i
load from %n
icmp slt
br i1
i >= n
i < n
for_end
load from %sum
return
for_body
load from %i
mul * 4
load from %sum

5.8.7. OPTIMIZED IR -O2

```
define dso_local i32 @_Z8example2i(i32 %0) local_unnamed_addr #0 {
entry:
  %cmp6 = icmp sgt i32 %0, 0
  br i1 %cmp6, label %for.body.preheader, label %for.end

for.body.preheader:
  %1 = add i32 %0, -1
  %2 = mul i32 %0, %1
  %3 = lshr i32 %2, 1
  %4 = mul i32 %3, 4
  br label %for.end

for.end:
  %sum.0.lcssa = phi i32 [ 0, %entry ], [ %4, %for.body.preheader ]
  ret i32 %sum.0.lcssa
}
```

5.8.8. OPTIMIZED IR -O2 GRAPH


%0 param
entry
icmp sgt i32 %0, 0
br i1
> 0
<= 0
for_body_preheader
add i32 %0, -1
mul i32 %0, %1
lshr i32 %2, 1
mul i32 %3, 4

6. SIMDJSON IMPLEMENTATION

6.1. SIMDJSON ARCHITECTURE OVERVIEW

1. Stage 1: Structural Index Creation (find location of important markers)
    1. Find structural characters ({,},[,],",,:)
    2. Identify string boundaries
    3. Locate whitespace
    4. Validate UTF-8 encoding
    5. Detect pseudo-structural characters
2. Stage 2: Parsing & Tape Building
    1. Parse atomic values (strings, numbers, true/false/null)
    2. Validate document structure
    3. Build navigable tape representation
    4. Convert numbers to machine formats
    5. Normalize strings to UTF-8

6.2. SIMDJSON DIAGRAM

```
    JSON INPUT STRING
   "{"name": "value"}"
            ⬇
     64-BYTE CHUNKS
   ╔══════════════════╗
   ║ "{"name": "val...║
   ╚══════════════════╝
            ⬇
        STAGE 1
  (Bitmap Generation &   find: ([, {, ], }, :, ,)
   Index Extraction)     escaped characters and quoted regions
            ⬇            Validate UTF-8
      INDEX ARRAY
     [0,3,5,7,...]
            ⬇
        STAGE 2          parse number, int, float, 1e10, true, false, null, string
   (Parse & Build Tape)  build tape to navigate
            ⬇
      FINAL TAPE
[root, {, "name", "value", }]
```

6.3. STAGE 1: STRUCTURAL AND PSEUDO STRUCTURAL INDEX CONSTRUCTION

6.3.1. INPUT AND OUTPUT

- Input: Raw JSON bytes
- Output:
    - Bitmask of structural chars
    - Array of integer indices marking structural elements

6.3.2. KEY RESPONSIBILITIES

1. Character encoding validation (UTF-8)
2. Locate structural characters ([, {,], }, :, ,)
3. Identify string boundaries
    1. Handles escaped characters and quoted regions
4. Find pseudo-structural characters (atoms like numbers, true, false, null)

6.4. STAGE 2: STRUCTURED NAVIGATION

6.4.1. INPUT AND OUTPUT

- Input: Array of structural indices from Stage 1
- Output: Parsed JSON structure on a “tape”(array)
- Purpose: Build navigable representation of JSON document

6.4.2. KEY RESPONSIBILITIES

1. Parse strings and convert to UTF-8
2. Convert numbers to 64-bit integers or doubles
3. Validate structural rules (matching braces, proper sequences)
4. Build navigable tape structure

6.4.3. THE TAPE FORMAT

- 64-bit words for each node
- Special encoding for different types:
    - Atoms (null, true, false): n/t/f × 2^56
    - Numbers: Two 64-bit words
    - Arrays/Objects: Start/end markers with navigation pointers
    - Strings: Pointer to string buffer

7. STAGE 1: STRUCTURAL AND PSEUDO STRUCTURAL INDEX CONSTRUCTION

```
const auto whitespace_table = simd8<uint8_t>::repeat_16(' ', 100, 100, 100, 17, 100, 1

const auto op_table = simd8<uint8_t>::repeat_16(
  0, 0, 0, 0,
  0, 0, 0, 0,
  0, 0, ':', '{', // : = 3A, [ = 5B, { = 7B
  ',', '}', 0, 0  // , = 2C, ] = 5D, } = 7D
);

const uint64_t whitespace = in.eq({
  _mm256_shuffle_epi8(whitespace_table, in.chunks[0]),
  _mm256_shuffle_epi8(whitespace_table, in.chunks[1])
});
// Turn [ and ] into { and }
const simd8x64<uint8_t> curlified{
  in.chunks[0] | 0x20,
  in.chunks[1] | 0x20
};
const uint64_t op = curlified.eq({
  _mm256_shuffle_epi8(op_table, in.chunks[0]),
  _mm256_shuffle_epi8(op_table, in.chunks[1])
});

return { whitespace, op };
```

7.1. STAGE 1: 1 VECTORIZED CLASSIFICATION AND PSEUDO-STRUCTURAL CHARACTERS

- Want to obtain location of structural characters ({, }, [,], :, ,)
    - pseudo-structural - Any non-whitespace character that immediately follows a structural character or whitespace
    - useful for parsing, we need this bit mask to build tape

```
{ "\\\"Nam[{": [ 116,"\\\\" , 234, "true", false ], "t":"\\\"" }
__1______________1___1________1____1_______1________1___1_______
```

7.1.1. VECTORIZED CLASSIFICATION

code points	character	desired value	bin
0x2c	`,` (comma)	1	00001
0x3a	`:` (colon)	2	00010
0x5b	`[`	4	00100
0x5d	`]`	4	00100
0x7b	`{`	4	00100
0x7d	`}`	4	00100
0x09	TAB	8	01000
0x0a	LF	8	01000
0x0d	CR	8	01000
0x20	SPACE	16	10000
others	any other	0	00000

- PMOVMSKB
    - _mm256_movemask_epi8 to extract the bits into bitmap

7.1.2. VPSHUFB: VECTOR PERMUTE SHUFFLE BYTES

- basically a one instruction lookup table using the 4 lowest bit(nibble)
    - 0000 XXXX

```
int main() {
    // Lookup table for hex digits "0123456789abcdef"
    __m256i lut = _mm256_setr_epi8(
        '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f',
        '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
    );

    // Example 2: Alternating normal/zeroed values (0x00,0x80,0x01,0x81...)
    __m256i indices2 = _mm256_setr_epi8(
        0x00, 0x80, 0x01, 0x81, 0x02, 0x82, 0x03, 0x83, 0x04, 0x84, 0x05, 0x85, 0x06, 0x
        0x08, 0x88, 0x09, 0x89, 0x0A, 0x8A, 0x0B, 0x8B, 0x0C, 0x8C, 0x0D, 0x8D, 0x0E, 0x
    );

    printf("\nAlternating with zeroes (. represents zero):\n");
    print_bytes(_mm256_shuffle_epi8(lut, indices2));
    // Alternating with zeroes (. represents zero):
    // 0.1.2.3.4.5.6.7.8.9.a.b.c.d.e.f.

    return 0;
}

#pragma GCC target("avx2")
#include <immintrin.h>
#include <stdio.h>
```

```c
void print_bytes(__m256i v) {
    unsigned char bytes[32];
    _mm256_storeu_si256((__m256i*)bytes, v);
    for(int i = 0; i < 32; i++) {
        if (bytes[i]) {
            printf("%c", bytes[i]);
        } else {
            printf(".");  // Print dot for zero bytes
        }
    }
    printf("\n");
}
```

7.1.3. SIMPLE EXAMPLE

code points	character	desired value	bin
0x3a	`:` (colon)	2	00010
0x0a	LF	8	01000

- use vpshufb to match low nibble a
- could be both : and LF so it must match 0010 | 1000 = 1010
- low nibble at position A = 10
  - high nibble 0x3 vs 0x0
    - 0x3 = 2
    - 0x0 = 8

7.1.4. SIMPLE EXAMPLE

```
"LF:"

Low nibble table
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
xx xx xx xx xx xx xx xx xx xx 10 xx xx xx xx xx
1010

high nibble table
00 .. 02 03 04 05 06 07 08 09 10 11 12 13 14 15
08 .. 02 xx xx xx xx xx xx xx xx xx xx xx xx xx
0100,  0010
```

7.1.5. SIMPLE EXAMPLE

		LF	:
	low	1010	1010
	high	1000	0010
AND		1000	0010
		8	2

7.1.6. STAGE 1: BITMAP TO ARRAY INDEX

7.1.7. INPUT DATA

```
{ "\\\"Nam[{": [ 116,"\\\\" , 234, "true", false ], "t":"\\\"" }: input data
__1_________1________1____1________1____1___________1_1_1____1__: Q
1_________11_1_1____1_______1____1_______1_______11____1_______1: S
_1____________1_1__________1_1____1_______1_____1__1__________1_: W
```

- take Q for example, we want to convert Q’s bit mask into a list of indexes
  - [2, 12, 22, 27, 37, 42, 54, 56, 58, 62]

7.1.8. EXTRACTION

- 2 instructions
  - tzcnt count trailing least significant 0 bits
  - blsr which delete the last bit.

```
a = 1010000
idx = tzcnt(a) // 4
a = blsr(a)    // 1000000
idx = tzcnt(a) // 6
```

7.1.9. NAIVE IMPLEMENTATION

```
void extract_set_bits_unoptimized(uint64_t bitset, uint32_t* output) {
    uint32_t pos = 0;

    // This while loop is the source of unpredictable branches
    while (bitset) {
        // Find position of lowest set bit
        uint32_t bit_pos = __builtin_ctzll(bitset);
        // Store the position
        *output++ = bit_pos;
        // Clear the lowest set bit
        bitset &= (bitset - 1);
    }
}
```

7.1.10. MINIMAL BRANCHING IMPLEMENTATION

```
void extract_set_bits_optimized(uint64_t bitset, uint32_t* output) {
    // Get total number of set bits
    uint32_t count = __builtin_popcountll(bitset);
    uint32_t* next_base = output + count;

    // Process 8 bits at a time unconditionally
    while (bitset) {
        // Extract next 8 set bit positions, even if we don't have 8 bits
        *output++ = __builtin_ctzll(bitset);
        bitset &= (bitset - 1);  // Clear lowest set bit (blsr instruction)

        *output++ = __builtin_ctzll(bitset);
        bitset &= (bitset - 1);

        *output++ = __builtin_ctzll(bitset);
        bitset &= (bitset - 1);

        *output++ = __builtin_ctzll(bitset);
        bitset &= (bitset - 1);

        *output++ = __builtin_ctzll(bitset);
        bitset &= (bitset - 1);

        *output++ = __builtin_ctzll(bitset);
        bitset &= (bitset - 1);

        *output++ = __builtin_ctzll(bitset);
        bitset &= (bitset - 1);

        *output++ = __builtin_ctzll(bitset);
        bitset &= (bitset - 1);
    }

    // Reset output pointer to actual end based on real count
```

```
    output = next_base;
}
```

7.2. STAGE 1: 2 ELIMINATED ESCAPED OR QUOTED SUBSTRING

7.2.1. GET BACKSLASH

```
am[{": [ 116,"\\\\" , 234, "true", false ], "t":"\\\"" }: input data
______________1111_______________________________111____: B = backslash_bits
________________1111_______________________________111___: bits_shifted_left = backslash_bits << 1

______________1111_______________________________111____: bits
________________0000_______________________________000___: inverted = ~bits_shifted_left
______________1__________________________________1______: S = starts = bits & inverted
t the first backslash of every group
```

7.2.2. GET ODD LENGTH SEQUENCES STARTING ON AN ODD OFFSET

```
_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1: O (constant)
___111________________1111_______________________________111____: B = backslash_bits
___1__________________1__________________________________1______: S = starts = bits & inverted
_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1: O (constant)
___1_____________________________________________________1______: OS = S & O

// add B to OS, yielding carries on backslash sequences with odd starts
___1_____________________________________________________1______: OS = S & O
___111________________1111_______________________________111____: B = backslash_bits
   -->                                                   -->
______1_______________1111__________________________________1___: OC = B + OS

// filter out the backslashes from the previous addition, getting carries only
___111________________1111_______________________________111____: B = backslash_bits
___000________________0000_______________________________000____: ~B
______1_______________1111__________________________________1___: OC = B + OS
______1_____________________________________________________1___: OCO = OC & ~B

// get the odd-length sequence starting on an odd offset and ending on even offset
______1_____________________________________________________1___: OCO = OC & ~B
1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1: E (constant)
______1_____________________________________________________1___: OD2 = OCO & E
// this shows two odd-length sequence starting on an odd offset
```

7.2.3. GET ODD LENGTH SEQUENCES STARTING ON AN EVEN OFFSET

its just the reverse of what we done just now

```
Nam[{": [ 116,"\\\\" , 234, "true", false ], "t":"\\\"" }: input data
_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_: E (constant)
______________1__________________________________1______: S = starts = bits & inverted
______________1_________________________________________: ES = S & E
______________1111_______________________________111____: B = backslash_bits
B to ES, yielding carries on backslash sequences with even starts
              --->
__________________1______________________________111____: EC = B + ES
er out the backslashes from the previous addition, getting carries only
__________________1_____________________________________: ECE = EC & ~B
ct only the end of sequences ending on an odd offset
__________________1_____________________________________: ECE = EC & ~B
1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1: O (constant)
________________________________________________________: OD1 = ECE & ~E
e are no odd-length sequences of backslashes starting on an even offset
```

7.2.4. GET SEQUENCES WITH ODD OFFSET

```
merge results, yielding ends of all odd-length sequence of backslashes
______________________________________________________________: OD1 = ECE & ~E
___1_____________________________________________________1____: OD2 = OCO & E

___1_____________________________________________________1____: OD = OD1 | OD2
\\\"Nam[{": [ 116,"\\\\" , 234, "true", false ], "t":"\\\"" }: input data

these " are escaped and thus are counted as text instead of structural characters
```

7.2.5. ELIMINATED ESCAPE

```
{ "\\\"Nam[{": [ 116,"\\\\" , 234, "true", false ], "t":"\\\"" }: input data
__1___1_____1________1____1________1____1___________1_1_1___11__: Q = quotes
______1_____________________________________________________1___: OD
// we remove the escaped " quotes
__1_________1________1____1________1____1___________1_1_1____1__: Q &= ~OD
__1111111111_________11111_________11111____________11__11111___: CLMUL(Q,~0)
```

7.2.6. SWEEPING

```
uint64_t xorShiftOperations(uint64_t num, bool rightShift, bool tutorial = false) {
    uint64_t result = num;
    for (int i = 1; i <= 64; i++) {
        result ^= (num << i);
    }
    // 0000000000000000000010000000000000000000000000000000000000000000
    // after apply
    // 0000000000000000000011111111111111111111111111111111111111111111

    // 0001000000000000000000000000000000000000000000000000000000000000
    // after apply
    // 0001111111111111111111111111111111111111111111111111111111111111
    return result;
}
```

7.2.7. SWEEPING

```
// 0000000000000000000010000000000000000000000000000000000000000000
// OR
// 0001000000000000000000000000000000000000000000000000000000000000
// result
// 0001000000000000000010000000000000000000000000000000000000000000

// 0000000000000000000011111111111111111111111111111111111111111111
// XOR
// 0001111111111111111111111111111111111111111111111111111111111111
// result
// 0001111111111111111100000000000000000000000000000000000000000000
```

7.2.8. SWEEPING

```
Testing left shift:
Initial number:
0x00      00100000 00001000 00000100 00100000 00010000 10000000 00001010 10000100
After left shift by 1:
0x00      00110000 00001100 00000110 00110000 00011000 11000000 00001111 11000110
After left shift by 2:
0x00      00111100 00001111 00000111 10111100 00011110 11110000 00001100 00110111
After left shift by 4:
0x00      00111111 11001111 11110111 11000111 11011111 00011111 00001100 11110100
After left shift by 8:
0x00      00111111 11110000 00111000 00110000 00011000 11000000 00010011 11111000
After left shift by 16:
0x00      00111111 11110000 00000111 11000000 00100000 11110000 00001011 00111000
After left shift by 32:
0x00      00111111 11110000 00000111 11000000 00011111 00000000 00001100 11111000
```

7.2.9. SWEEPING IMPLEMENTED BY CLMUL, PCLMULQDQ

- Carry Less Multiply
- CLMUL(4, 15)
- 4 * 15

```
        4
X      15
----------
        4
X(8+4+2+1)
----------
        4
        8
       16
+      32
----------
       60
----------
```

7.2.10. SWEEPING IMPLEMENTED BY CLMUL, PCLMULQDQ

- CLMUL(4, 15)
- XOR ~= ADD

```
         0100  (4)
   X     1111  (15)
-------------
        00100  (X1 means 4 << 0)
XOR    00100_  (X2 means 4 << 1)
XOR   00100__  (X4 means 4 << 2)
XOR  00100___  (X8 means 4 << 3)
-------------
       111100  (all XORed together)
-------------
```

7.2.11. SWEEPING IMPLEMENTED BY CLMUL, PCLMULQDQ

- CLMUL(inputJsonBitmask, EvenBitMask)

```
uint64_t result = num;
for (int i = 1; i <= 64; i++) {
    result ^= (num << i);
}
// 11111111 11111111 11111111 11111111

//            input
// XOR (input <<  1)
// XOR (input <<  2)
// XOR (input <<  4)
// XOR (input <<  8)
// XOR (input << 16)
// XOR (input << 32)
// ----------------
//       quote mask
// ----------------
```

7.2.12. FINALLY GET QUOTE MASK

```
{ "\\\"Nam[{": [ 116,"\\\\" , 234, "true", false ], "t":"\\\"" }: input data
__1111111111_________11111_________11111____________11__11111___: CLMUL(Q,~0)
```

7.3. STAGE 1: 3 CHARACTER-ENCODING VALIDATION

1. Initial ASCII Fast Path, first bit == 0
2. Main algorithm
    1. Range check(0xF4 saturated subtract)
    2. Continuation Byte validation

7.3.1. CHECK FOR ASCII FAST PATH

```
Single byte (ASCII):
0xxxxxxx                     (values 0-127)
Values start with 0, remaining 7 bits for data
```

7.3.2. CONTINUATION BYTE VALIDATION

```
Single byte (ASCII):
0xxxxxxx                     (values 0-127)
Values start with 0, remaining 7 bits for data

Two bytes:
110xxxxx 10xxxxxx           (values 128-2047)
First byte starts with 110

Three bytes:
1110xxxx 10xxxxxx 10xxxxxx  (values 2048-65535)
First byte starts with 1110

Four bytes:
1111xxxx 10xxxxxx 10xxxxxx 10xxxxxx   (values 65536+)
First byte starts with 11110
```

7.3.3. MAP TO VALUES

high	Dec	high	Dec
0000	1	1000	0
0001	1	1001	0
0010	1	1010	0
0011	1	1011	0
0100	1	1100	2
0101	1	1101	2
0110	1	1110	3
0111	1	1111	4

```
1111xxxx 10xxxxxx 10xxxxxx 10xxxxxx   (values 65536+)
4 0 0 0

1110xxxx 10xxxxxx 10xxxxxx  (values 2048-65535)
3 0 0
```

7.3.4. SIMD VALIDATION ALGORITHM

```
4 0 0 0 3 0 0 2 0 1 1 1
  4 0 0 0 3 0 0 2 0 1 1 1 // <<= 1 byte, shift left by 1 byte
  3 0 0 0 2 0 0 1 0 0 0 0 // saturated subtract 1 from each byte

4 0 0 0 3 0 0 2 0 1 1 1
  3 0 0 0 2 0 0 1 0 0 0 0
4 3 0 0 3 2 0 2 1 1 1 1   // add it back into the original mapping

4 3 0 0 3 2 0 2 1 1 1 1   // add it back into the original mapping
    4 3 0 0 3 2 0 2 1 1 1 1   // <<= 2 byte, shift left by 2 bytes
    2 1 0 0 1 0 0 0 0 0 0 0   // saturated subtract 2
4 3 2 1 3 2 1 3 1 1 1 1   // add it back
// the end result will have no 0
// none of the numbers are bigger than the original
```

7.3.5. SIMD VALIDATION ALGORITHM: INVALID EXAMPLE

```
2 0 0 0 4 3 0 0
  2 0 0 0 4 3 0 // shift left 1
  1 0 0 0 3 2 0 // saturated subtract 1
2 1 0 0 4 6 2 0

2 1 0 0 4 6 2 0
    0 0 2 1 0 0 4 6 // shift left 2
    0 0 0 0 0 0 2 4 // saturated subtract 2
2 1 0 0 4 6 4 4

2 0 0 0 4 3 0 0
2 1 0 0 4 6 4 4
    --- zeros found here invalid
          - 6 > 3
```

8. STAGE 2: BUILDING THE TAPE

8.1. STAGE 2: THE TAPE

8.1.1. THREE CATEGORIES OF TAPE ENTRIES

1. Direct Values (Atoms)
    - null, true, false
    - numbers (integers and floats) - takes 2 tape entries
2. String References
    - Points to separate string buffer
    - Not original JSON string
3. Structural Navigation
    - Array brackets [,]
    - Object braces {,}
    - Contains jump indices

8.1.2. BASIC STRUCTURE

- Tape is array of 64-bit words
- Each entry: `TYPE_MARKER × 2^56 + payload`
- High 8 bits: Type information
- Low 56 bits: Value or reference

```
63      56 55                    0
+--------+---------------------+
|  TYPE  |     PAYLOAD         |
+--------+---------------------+
   8 bits      56 bits
```

8.1.3. DIRECT VALUES (ATOMS)

```
01101110 00000000 00000000 00000000 00000000 00000000 00000000 00000000
   ^'n' null
Hex: 0x6E00000000000000

01110100 00000000 00000000 00000000 00000000 00000000 00000000 00000000
   ^'t' true
Hex: 0x7400000000000000

01100110 00000000 00000000 00000000 00000000 00000000 00000000 00000000
   ^'f' false
Hex: 0x6600000000000000
```

8.1.4. NUMBER: INTEGER EXAMPLE (42)

Takes 2 tape entries:

- first one is just a type marker
- second is the value

```
Entry 1 (type marker):
01101100 00000000 00000000 00000000 00000000 00000000 00000000 00000000
   ^'l'
Hex: 0x6C00000000000000

Entry 2 (value):
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00101010
                                                                 ^42
Hex: 0x000000000000002A
```

8.1.5. NUMBER: FLOAT EXAMPLE (3.14)

Takes 2 tape entries:

```
Entry 1 (type marker):
01100100 00000000 00000000 00000000 00000000 00000000 00000000 00000000
   ^'d'
Hex: 0x6400000000000000

Entry 2 (value in IEEE 754):
01000000 00001001 00011110 10111000 01010100 01000000 00000000 00000000
Hex: 0x400921994000000
```

8.1.6. STRING TAPE ENTRY

```
Example for "..........hello":
Binary:
00100010 00000000 00000000 00000000 00000000 00000000 00000000 00001010
   ^'"'                                                         ^offset=10
Hex: 0x220000000000000A
```

- The string buffer is a separate array that stores normalized UTF-8 strings

1. Benefits of This Approach
    - Fast length retrieval - no variable length guessing search in tape
    - Contains normalized UTF-8 strings

8.1.7. OBJECT EXAMPLE

```
{"name": "John"}

Opening brace (points forward):
Binary:
01111011 00000000 00000000 00000000 00000000 00000000 00000000 00000010
   ^'{'                                                          ^next=2
Hex: 0x7B00000000000002

Closing brace (points backward):
Binary:
01111101 00000000 00000000 00000000 00000000 00000000 00000000 00000000
   ^'}'                                                          ^prev=0
Hex: 0x7D00000000000000
```

8.1.8. ARRAY EXAMPLE

```
array = [1,2,3]
```

addr	type	char	tape entry
0	array	[–8	0x5B00000000000008
1	integer	l	0x6C00000000000000
2	value		0x0000000000000001
3	integer	l	0x6C00000000000000
4	value		0x0000000000000002
5	integer	l	0x6C00000000000000
6	value		0x0000000000000003
7	array	]–0	0x5D00000000000000
8	other		other

#+END_SRC

8.1.9. JSON DOCUMENT

```
{
  "name": "John",
  "age": 42,
  "active": true
}
```

```
Idx  Type    Payload   Description
0:   'r'     12        Root (points to end)
1:   '{'     12        Object start (points to end)
2:   '"'     100       String "name" (points to string buffer offset 100)
3:   '"'     150       String "John" (points to string buffer offset 150)
4:   '"'     200       String "age" (points to string buffer offset 200)
5:   'l'     0         Integer marker
6:   -       42        Integer value
7:   '"'     250       String "active" (points to string buffer offset 250)
8:   't'     0         true value
9:   '}'     1         Object end (points to start)
10:  'r'     0         Root end (points to start)
```

8.1.10. TAPE BENEFITS

- Cache-friendly linear layout
- Fast navigation with index jumping
- SIMD-friendly processing
- Predictable memory layout

8.2. STAGE 2: 1 NUMBER PARSING

8.2.1. UNDERSTANDING THE IS_ALL_DIGITS

Fast 8 digit check

```
uint64 high_nibble = val & 0xF0F0F0F0F0F0F0F0;
uint64 low_nibble = ((val + 0x0606060606060606) & 0xF0F0F0F0F0F0F0F0) >> 4;
uint64 combined = high_nibble | low_nibble;
bool is_all_digits = combined == 0x3333333333333333;
```

8.2.2. KEY INSIGHT: ASCII CHARACTERS FROM 0X29 TO 0X3A

- notice all high nibble of valid digits are 3

Char	Hex	Binary	Description	
'/'	0x2F	0010 1111	Forward Slash	
'0'	0x30	0011 0000	Digit Zero	<– Valid digits start
'1'	0x31	0011 0001	Digit One	
'2'	0x32	0011 0010	Digit Two	
'3'	0x33	0011 0011	Digit Three	
'4'	0x34	0011 0100	Digit Four	
'5'	0x35	0011 0101	Digit Five	
'6'	0x36	0011 0110	Digit Six	
'7'	0x37	0011 0111	Digit Seven	
'8'	0x38	0011 1000	Digit Eight	
'9'	0x39	0011 1001	Digit Nine	<– Valid digits end
':'	0x3A	0011 1010	Colon	

8.2.3. STEP 1: INITIAL MASKING OF HIGH NIBBLES

```
uint64 high_nibble = val & 0xF0F0F0F0F0F0F0F0;
```

- if you are lesser than 0x3X, you are 0x2F,
- Let’s take valid input “12345678”:

```
Input bytes:    31 32 33 34 35 36 37 38
                || || || || || || || ||
                v| v| v| v| v| v| v| v|
High nibble:    3  3  3  3  3  3  3  3
                |  |  |  |  |  |  |  |
Mask:           F0 F0 F0 F0 F0 F0 F0 F0
                =  =  =  =  =  =  =  =
Result1:        30 30 30 30 30 30 30 30
```

8.2.4. HOW THE LOW NIBBLE CHECK WORKS

- we want to ensure that low nibble is within 0xX0 - 0xX9
    - 0xXA - 0xXF is illegal
        - Analyzing Carry Detection with Binary

8.2.5. CASE 1: VALID DIGIT (0X39 = '9')

```
0x39 = 0011 1001  (Original value '9')
0x06 = 0000 0110  (Value we add)
      -----------
      0011 1111  (Result = 0x3F)
Low nibble does not overflow into high nibble and affect the 0x3 in high nibble

After masking high nibble (& 0xF0):
0x3F = 0011 1111
0xF0 = 1111 0000
      -----------
      0011 0000  (= 0x30)

After right shift by 4:
0x30 >> 4 = 0000 0011  (= 0x03) ✓ Valid!
```

8.2.6. CASE 2: INVALID CHARACTER (0X3A = ':')

```
0x3A = 0011 1010  (Original value ':')
0x06 = 0000 0110  (Value we add)
      -----------
       0011 0000
          1 0000
      -----------
      0100 0000  (Result = 0x40) <- Notice the carry!
                                   The '1' carried into the high nibble

After masking high nibble (& 0xF0):
0x40 = 0100 0000
0xF0 = 1111 0000
      -----------
      0100 0000  (= 0x40)

After right shift by 4:
0x40 >> 4 = 0000 0100  (= 0x04) ✗ Invalid!

 0x3X
|0xX4
-----
 0x34 <- INVALID
-----
```

8.2.7. STEP 2: ADD 0X06 TO DETECT NON-DIGITS

```
Low nibbles:    1  2  3  4  5  6  7  8
Add 0x06:       7  8  9  A  B  C  D  E
                ^  ^  ^  ^  ^  ^  ^  ^
                |  |  |  |  |  |  |  |
If original <= 9: No carry to high nibble
If original > 9: Carry affects high nibble
```

8.2.8. STEP 3: EXAMPLE WITH VALID DIGITS (0-9)

Take “12345678”:

```
Original:       31 32 33 34 35 36 37 38
                v  v  v  v  v  v  v  v
high nibble:    30 30 30 30 30 30 30 30

Original:       31 32 33 34 35 36 37 38
After +0x06:    37 38 39 3A 3B 3C 3D 3E
Mask high:      30 30 30 30 30 30 30 30
low nibble:     03 03 03 03 03 03 03 03

high nibble:    30 30 30 30 30 30 30 30
low nibble:     03 03 03 03 03 03 03 03
OR together:    33 33 33 33 33 33 33 33
```

8.2.9. STEP 4: EXAMPLE WITH INVALID CHARACTER (';' = 0X3B)

Take "1234;678":

```
Original:        31 32 33 34 3B 36 37 38
After +0x06:     37 38 39 3A 41 3C 3D 3E
                                ^
                                |
Mask high:       30 30 30 30 40 30 30 30
                                ^ Different!
Shift right 4:   03 03 03 03 04 03 03 03
high nibble:     30 30 30 30 30 30 30 30
OR together:     33 33 33 33 34 33 33 33 ≠ 0x3333...
                                ^ Caught!
```

8.2.10. WHY IT WORKS

1. First part (val & 0xF0F0…):
    - Isolates high nibbles
    - Must be 0x30 for valid digits
2. Second part ((val + 0x06…) & 0xF0…):
    - Adding 0x06 to low nibble:
        - For 0-9: Result stays within nibble
        - For >9: Causes carry
    - After shift right 4:
        - Valid digits: Always 0x03
        - Invalid: Different value
3. When OR’d together:
    - Valid digits: Always 0x33
    - Invalid: Different pattern

8.2.11. VALID CASES

```
"00000000" -> 0x3333333333333333 ✓
"99999999" -> 0x3333333333333333 ✓
"12345678" -> 0x3333333333333333 ✓
```

8.2.12. INVALID CASES

```
"A" (0x41):
Original:  41
+0x06:     47
High:      40 ≠ 30 -> Fails

"/" (0x2F):
Original:  2F
+0x06:     35
High:      20 ≠ 30 -> Fails

":" (0x3A):
Original:  3A
+0x06:     40
High:      40 ≠ 30 -> Fails
```

8.2.13. PERFORMANCE BENEFITS

- Single comparison instead of 8 individual checks
- No branches (important for modern CPUs)
- Uses native 64-bit operations
- Exploits CPU’s ability to do parallel checks

This algorithm is a beautiful example of bit manipulation that turns what would normally be 8 comparisons into a single mathematical test.

8.2.14. UNDERSTANDING SIMD-BASED FAST EIGHT-DIGIT NUMBER PARSING

Convert ASCII string of 8 digits to integer using SIMD instructions.
Example: “12345678” -> 12345678

```
uint32_t parse_eight_digits_unrolled(char *chars) {
  __m128i ascii0 = _mm_set1_epi8('0');
  __m128i mul_1_10 = _mm_setr_epi8(10, 1, 10, 1, 10, 1, 10, 1, 10, 1, 10, 1, 10, 1, 10,
  __m128i mul_1_100 = _mm_setr_epi16(100, 1, 100, 1, 100, 1, 100, 1);
  __m128i mul_1_10000 = _mm_setr_epi16(10000, 1, 10000, 1, 10000, 1, 10000, 1);
  __m128i number_ascii = _mm_loadu_si128((__m128i *)chars);
  __m128i in = _mm_sub_epi8(number_ascii, ascii0);
  __m128i t1 = _mm_maddubs_epi16(in, mul_1_10);
  __m128i t2 = _mm_madd_epi16(t1, mul_1_100);
  __m128i t3 = _mm_packus_epi32(t2, t2);
  __m128i t4 = _mm_madd_epi16(t3, mul_1_10000);
  return _mm_cvtsi128_si32(t4);
}
```

8.2.15. STEP 1: CONVERT ASCII TO NUMERIC VALUES

```
__m128i ascii0 = _mm_set1_epi8('0');
__m128i number_ascii = _mm_loadu_si128((__m128i *)chars);
__m128i in = _mm_sub_epi8(number_ascii, ascii0);
```

```
Input:          "12345678"
ASCII values:   31 32 33 34 35 36 37 38
Subtract:       30 30 30 30 30 30 30 30
Subtract '0':   01 02 03 04 05 06 07 08  (numeric values)
                |  |  |  |  |  |  |  |
Instruction:    _mm_sub_epi8 (PSUBB - packed subtract bytes)
```

8.2.16. STEP 2: MULTIPLY ALTERNATE DIGITS BY 10 AND ADD

```
__m128i mul_1_10 = _mm_setr_epi8(10, 1, 10, 1, 10, 1, 10, 1, 10, 1, 10, 1, 10, 1, 10,
__m128i t1 = _mm_maddubs_epi16(in, mul_1_10);
```

```
Values:          1  2  3  4  5  6  7  8
Multipliers:    10  1 10  1 10  1 10  1
                 |  |  |  |  |  |  |  |
Results:        10  2 30  4 50  6 70  8
                 \ /   \ /   \ /   \ /
Sums:            12    34    56    78     (as 16-bit values)

Instruction: _mm_maddubs_epi16 (PMADDUBSW - multiply and add unsigned bytes to signed wo
```

8.2.17. STEP 3: MULTIPLY ALTERNATE 16-BIT VALUES BY 100

```
__m128i mul_1_100 = _mm_setr_epi16(100, 1, 100, 1, 100, 1, 100, 1);
__m128i t2 = _mm_madd_epi16(t1, mul_1_100);
```

```
Values:         12   34   56   78
Multipliers:   100    1  100    1
                 |    |    |    |
Results:      1200   34 5600   78
                  \ /       \ /
Sums:            1234      5678    (as 32-bit values)

Instruction: _mm_madd_epi16 (PMADDWD - multiply and add packed words)
```

- what is the next step? 10000?

```
__m128i mul_1_10000 = _mm_setr_epi16(10000, 1, 10000, 1, 10000, 1, 10000, 1);
```

8.2.18. STEP 4: PACK 32-BIT VALUES TO 16-BIT

- reinterpret value as 32 bit instead of 16 bits!? why?
- so we can use `_mm_setr_epi16` instead of `_mm_setr_epi32`
  - its more efficient

```
  uint16 max_value = 65536;
__m128i t3 = _mm_packus_epi32(t2, t2);
```

```
Before:   1234(32-bit)  5678(32-bit)
After:    1234(16-bit)  5678(16-bit)

Instruction: _mm_packus_epi32 (PACKUSDW - pack with unsigned saturation)
```

8.2.19. STEP 5: FINAL COMBINE WITH MULTIPLY BY 10000

```
__m128i mul_1_10000 = _mm_setr_epi16(10000, 1, 10000, 1, 10000, 1, 10000, 1);
__m128i t4 = _mm_madd_epi16(t3, mul_1_10000);
```

```
Values:         1234     5678
Multipliers:   10000        1
                   |        |
Results:    12340000     5678
                    \   /
Sum:            12345678    (final 32-bit result)

Instruction: _mm_madd_epi16 (PMADDWD again)
```

8.2.20. SUMMARY: WHY THIS IS FAST

1. Parallel Processing:
    - Processes multiple digits simultaneously
    - Uses CPU’s SIMD capabilities efficiently
2. Instruction Count:
    - Traditional: ~8 loads + ~8 multiplies + ~7 adds
    - SIMD: ~7 total instructions
3. Latency Analysis on Haswell:
    - PSUBB (subtract): 1 cycle
    - PMADDUBSW (multiply-add bytes): 5 cycles
    - PMADDWD (multiply-add words): 5 cycles
    - PACKUSDW (pack): 1 cycle
    - Total latency: ~17 cycles

9. ACTUAL C++ CODE IMPLEMENTATION AND OPTIMIZATION TRICKS IN THE CODE BASE

9.1. SIMD8 ZERO COST “ABSTRACTION”

```
                          +---------------------+
                          |    base<Child>      |
                          +---------------------+
                                    |
                                    |
                     +--------------┴--------------+
                     |                             |
              (for T ≠ bool)                   (for bool)
                     |                             |
              +------▼------+                +------▼------+
              |   base8<T>  |                | base8<bool> |
              +-------------+                +-------------+
                     |                             |
                     |                             |
            +--------▼--------+                    |
            | base8_numeric<T>|                    |
            +-----------------+                    |
                     |                       +-----▼-----+
           +---------┴---------+             |simd8<bool>|
           |                   |             +-----------+
           |                   |
   +-------▼-------+   +-------▼-------+
   | simd8<int8_t> |   | simd8<uint8_t>|
   +---------------+   +---------------+

base<simd8<uint8_t>>     // Base template
    ↑
base8<uint8_t>          // Adds common SIMD operations
    ↑
base8_numeric<uint8_t>  // Adds numeric operations
    ↑
simd8<uint8_t>         // Final implementation
```

9.1.1. QUALITY OF LIFE ABSTRACTIONS

```
int8_t>: base8_numeric<uint8_t> {
 math
ine simd8<uint8_t> saturating_add(const simd8<uint8_t> other) const { return _mm256_adds_epu8(*
ine simd8<uint8_t> saturating_sub(const simd8<uint8_t> other) const { return _mm256_subs_epu8(*

cific operations
ine simd8<uint8_t> max_val(const simd8<uint8_t> other) const { return _mm256_max_epu8(*this, ot
ine simd8<uint8_t> min_val(const simd8<uint8_t> other) const { return _mm256_min_epu8(other, *t
, but only guarantees true is nonzero (< guarantees true = -1)
ine simd8<uint8_t> gt_bits(const simd8<uint8_t> other) const { return this->saturating_sub(othe
, but only guarantees true is nonzero (< guarantees true = -1)
ine simd8<uint8_t> lt_bits(const simd8<uint8_t> other) const { return other.saturating_sub(*thi
ine simd8<bool> operator<=(const simd8<uint8_t> other) const { return other.max_val(*this) == o
ine simd8<bool> operator>=(const simd8<uint8_t> other) const { return other.min_val(*this) == o
ine simd8<bool> operator>(const simd8<uint8_t> other) const { return this->gt_bits(other).any_b
ine simd8<bool> operator<(const simd8<uint8_t> other) const { return this->lt_bits(other).any_b
```

9.1.2. QUALITY OF LIFE ABSTRACTIONS

```
fic operations
ine simd8<bool> bits_not_set() const { return *this == uint8_t(0); }
ine simd8<bool> bits_not_set(simd8<uint8_t> bits) const { return (*this & bits).bits_not_set();
ine simd8<bool> any_bits_set() const { return ~this->bits_not_set(); }
ine simd8<bool> any_bits_set(simd8<uint8_t> bits) const { return ~this->bits_not_set(bits); }
ine bool is_ascii() const { return _mm256_movemask_epi8(*this) == 0; }
ine bool bits_not_set_anywhere() const { return _mm256_testz_si256(*this, *this); }
ine bool any_bits_set_anywhere() const { return !bits_not_set_anywhere(); }
ine bool bits_not_set_anywhere(simd8<uint8_t> bits) const { return _mm256_testz_si256(*this, bi
ine bool any_bits_set_anywhere(simd8<uint8_t> bits) const { return !bits_not_set_anywhere(bits)
 N>
ine simd8<uint8_t> shr() const { return simd8<uint8_t>(_mm256_srli_epi16(*this, N)) & uint8_t(0
 N>
ine simd8<uint8_t> shl() const { return simd8<uint8_t>(_mm256_slli_epi16(*this, N)) & uint8_t(0
f the bits and make a bitmask out of it.
e.get_bit<7>() gets the high bit
 N>
ine int get_bit() const { return _mm256_movemask_epi8(_mm256_slli_epi16(*this, 7-N)); }
```

9.2. TEMPLATE METAPROGRAMMING & CRTP VS. VIRTUAL FUNCTIONS (DYNAMIC BINDING)

- **Compile-Time Polymorphism with Templates/CRTP:**
  - **Zero-Cost Abstraction:** The CRTP pattern lets the compiler resolve function calls at compile time.
    - **Example from simdjson:**

```
template<typename Child>
struct base {
  // Overloaded operator (inline, no vtable overhead)
  simdjson_inline Child operator|(const Child other) const {
    return _mm256_or_si256(*this, other);
  }
};
```

- **Inlining & Optimization**
- **No Runtime Indirection**

9.2.1. DYNAMIC BINDING WITH VIRTUAL FUNCTIONS

- **Late Binding:** Function calls are resolved at runtime via a vtable.
  - **Example (the costly alternative):**

```cpp
struct Base {
  virtual void foo() = 0;
  virtual ~Base() = default;
};

struct Derived : Base {
  void foo() override {
    // ... implementation ...
  }
};
```

- **Runtime Overhead:**
  - indirection
  - cannot inline
- **Comparable to Java Interfaces:**

9.2.2. WHY C++ CHOOSES COMPILE-TIME POLYMORPHISM

Java	C++
• Runtime method dispatch via JIT	• Compile-time resolution via templates
• Variable latency due to GC	• No GC = predictable latency
• Performance changes during execution	• Performance known at compile time
• Requires “warm up” for optimization	• Consistent from first call

9.3. INLINE FUNCTIONS & COMPILE-TIME INLINING

- **Technique:** Functions are marked with `simdjson_inline` to encourage inlining.
- **Why?** Inlining eliminates function call overhead for tiny, frequently used functions.
- **Example from simdjson:**

```
#elif defined(__GNUC__) && !defined(__OPTIMIZE__)
  // If optimizations are disabled, forcing inlining can lead to significant
  // code bloat and high compile times. Don't use simdjson_really_inline for
  // unoptimized builds.
  #define simdjson_inline inline
#else

// Overloaded bitwise OR operator
simdjson_inline Child operator|(const Child other) const {
  return _mm256_or_si256(*this, other);
}
```

- **Note:** The use of inlining on all small operations (e.g. arithmetic, bitwise operators) ensures maximum performance.

9.4. C++ CASTS IN SIMDJSON: PERFORMANCE CONSIDERATIONS

- In high-performance C++ code, using the proper cast is essential for both safety and speed.
- C++ provides several cast operators:
  - **static_cast**: Compile-time conversions.
  - **reinterpret_cast**: Low-level, pointer and bit-reinterpretation.
  - **const_cast**: Remove constness.
  - **dynamic_cast**: Runtime-checked casts (with RTTI).

9.4.1. STATIC_CAST FOR CRTP EFFICIENCY

- known at compile-time, ensuring zero-cost abstraction.

```
template<typename Child>
struct base {
  __m256i value;
  // Overloaded compound assignment using CRTP
  simdjson_inline Child& operator|=(const Child other) {
    auto this_cast = static_cast<Child*>(this);
    *this_cast = *this_cast | other;
    return *this_cast;
  }
};
```

- **Notes:**
    - The `static_cast<Child*>(this)` converts the base class pointer to the derived type.

9.4.2. REINTERPRET_CAST FOR SIMD MEMORY OPERATIONS

- Reinterpret raw memory (such as an array of bytes) as SIMD register types.
- cannot static cast, type checked

```
static simdjson_inline simd8<T> load(const T values[32]) {
  return _mm256_loadu_si256(reinterpret_cast<const __m256i *>(values));
}
```

- **Notes:**
  - These reinterpret_casts allow the compiler to generate efficient SIMD load/store instructions.
  - They incur no runtime penalty as they are resolved during compilation.

9.4.3. WHY NOT DYNAMIC_CAST OR CONST_CAST?

- **dynamic_cast:**
    - Performs runtime type checking and incurs additional overhead.
- **const_cast:**
    - const -> other type

9.4.4. SUMMARY OF CASTS IN SIMDJSON

- **static_cast:**
  - Used for compile-time conversions (e.g. CRTP base-to-derived pointer conversion).
  - Zero-cost and type-safe.
- **reinterpret_cast:**
  - Used for pointer re-interpretation (e.g. converting a byte array to a SIMD register pointer).
  - Necessary for interfacing with low-level intrinsics.
- **Avoided Casts:**
  - **dynamic_cast** and **const_cast** are not used in performance-critical sections to prevent unnecessary runtime overhead.

9.5. WHY ERROR CODES OUTPERFORM EXCEPTIONS

- Zero-cost error handling: No stack unwinding or EH tables
- Better compiler optimizations: Linear control flow
- Predictable branch patterns: CPU pipelining friendly
- Smaller code size: No exception handling metadata

```
simdjson_warn_unused error_code minify(const uint8_t *buf, size_t len, uint8_t *dst, siz
  return set_best()->minify(buf, len, dst, dst_len);
}
```

9.5.1. ASSEMBLY COMPARISON: ERROR CODE PATH (SIMDJSON STYLE)

```
check_ascii:
  vptest %ymm0, %ymm1
  jne .error          ; Single conditional branch
  ; ... normal path ...

.error:               ; simd branchless way if possible
  mov eax, 1          ; Set error code
  ret
```

9.5.2. ASSEMBLY COMPARISON: EXCEPTION PATH

```
check_ascii:
  vptest %ymm0, %ymm1
  jne .exception
  ; ... normal path ...

.exception:
  call __cxa_allocate_exception  ; Heavy EH machinery
  ; ... stack unwinding setup ...
  ; - Exception table lookups
  ; - Destructor calls
  ; - Catch handler matching
  ; - Stack unwinding
```

9.5.3. KEY PERFORMANCE FACTORS

1. **No EH Table Overhead**
    - Exception handling requires RTTI and stack unwinding tables
2. **CPU Branch Prediction**
    - Error codes use simple conditional branches
        - Exceptions create unpredictable control flow
3. **Inlining Friendly**
    - Error return paths don’t inhibit function inlining
    - Critical for SIMD optimizations

9.6. MEMORY ALIGNMENT & PADDING

- Correct memory alignment (and extra padding) is crucial for SIMD operations; unaligned accesses can severely hurt performance.

```
simdjson::padded_string_view get_padded_string_view(const char *buf, size_t len,
                                                    simdjson::padded_string &jsonbuffer)
  if (need_allocation(buf, len)) { // unlikely case
    jsonbuffer = simdjson::padded_string(buf, len);
    return jsonbuffer;
  } else { // no allocation needed (most common)
    return simdjson::padded_string_view(buf, len, len + simdjson::SIMDJSON_PADDING);
  }
}
```

9.7. LOOP UNROLLING AND VECTORIZED PROCESSING

- **Key idea:** Unroll loops to manually do more things in one loop

```
void extract_set_bits_optimized(uint64_t bitset, uint32_t* output) {
    // Get total number of set bits
    uint32_t count = __builtin_popcountll(bitset);
    uint32_t* next_base = output + count;

    // Process 8 bits at a time unconditionally
    while (bitset) {
        // Extract next 8 set bit positions, even if we don't have 8 bits
        *output++ = __builtin_ctzll(bitset);
        bitset &= (bitset - 1);  // Clear lowest set bit (blsr instruction)

        *output++ = __builtin_ctzll(bitset);
        bitset &= (bitset - 1);
```

9.8. COMPILER DIRECTIVES & SPECIAL BUILD FLAGS

- Compiler flags (for instance, -O3 or -march=native) and specific macros are key to unlocking peak performance.

9.9. C++ OPTIMIZATIONS SUMMARY

- Zero cost abstractions
- inline functions and casting
- Error code over exceptions
- memory and loop optimizations

10. THANK YOU