每秒解析千兆字节的 JSON

WONG DING FENG

Created: 2025-02-11 Tue 13:54

TABLE OF CONTENTS

1. 目标

- 为什么 JSON 很慢?
- 位运算和 simd 简明入门
- simdjson 架构
- 使用的 C++ 技术

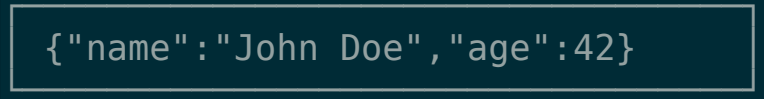
2. 问题

2.1. SBE VS JSON

Binary Format (Schema: string[10], uint8)


```
  |                └─ Age: Fixed 1 byte, parser knows to read exactly 1 byte
  └─ Name: Fixed 10 bytes, parser knows to read exactly 10 bytes
                      (padded with spaces)
```

JSON Format




```
  |    |          |     └─ Parser must scan until it finds closing brace
  |    |          └─ Parser must scan for quotes and ":"
  |    └─ Parser must scan for quotes and ":"
  └─ Parser must scan character by character, looking for valid JSON tokens
```

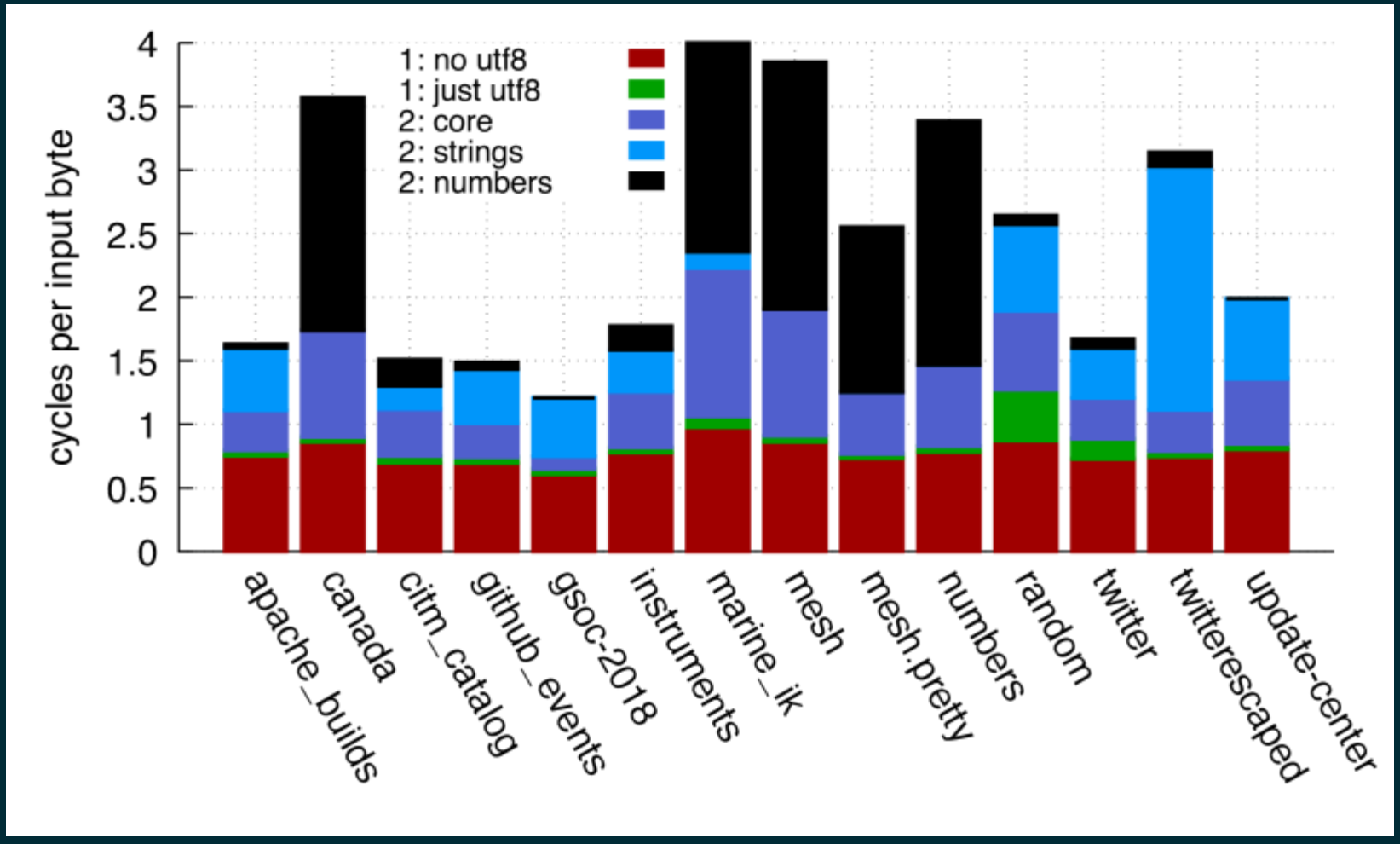
2.2. 为什么 JSON 有趣?

- 大部分数据都是 json 格式

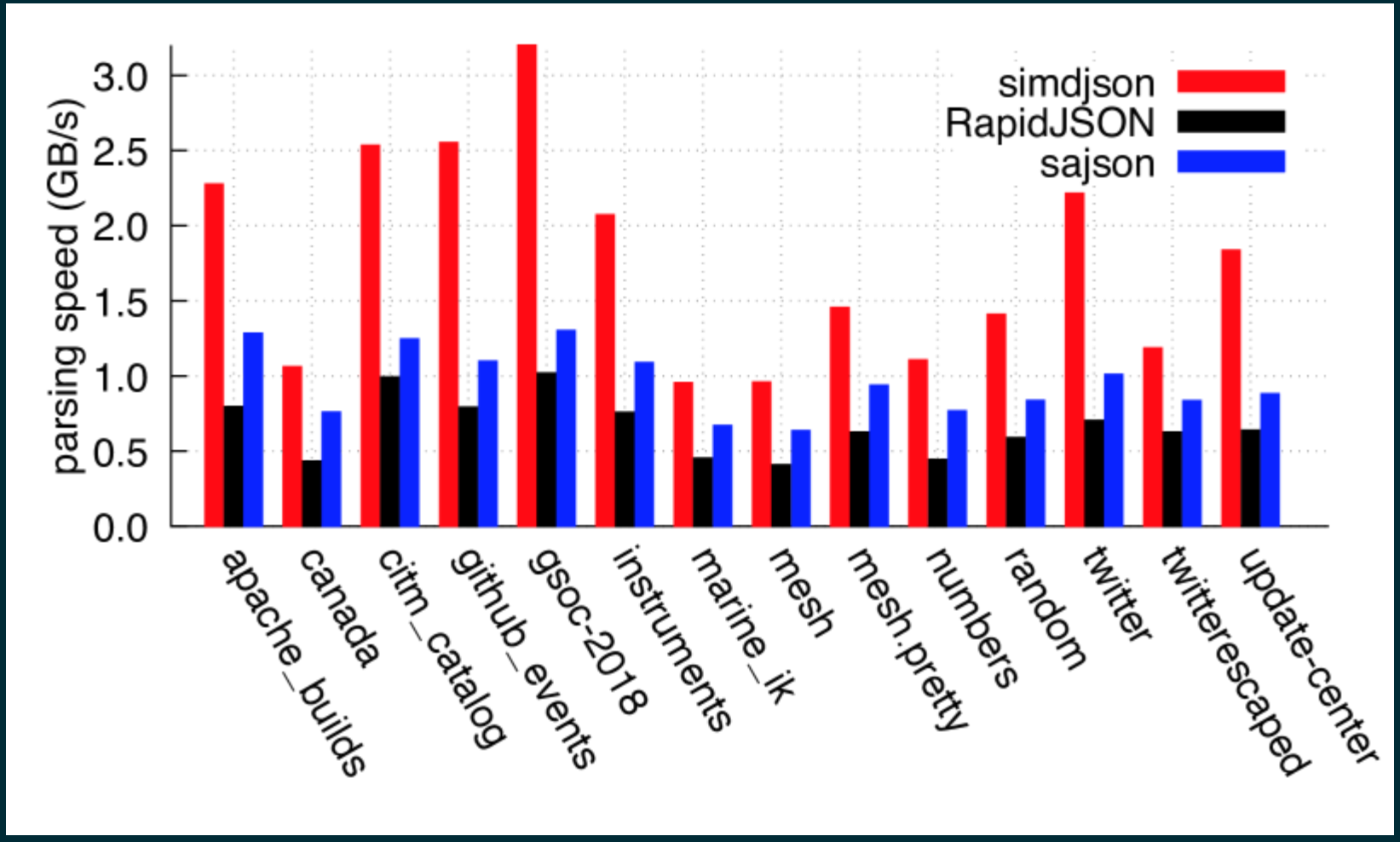
parser	Skylake	Cannon Lake	speed
simdjson	1.4	1.3	fast
RapidJSON	0.56	0.44	slow
sajson	0.93	0.84	normal

2.3. 为什么你应该感兴趣

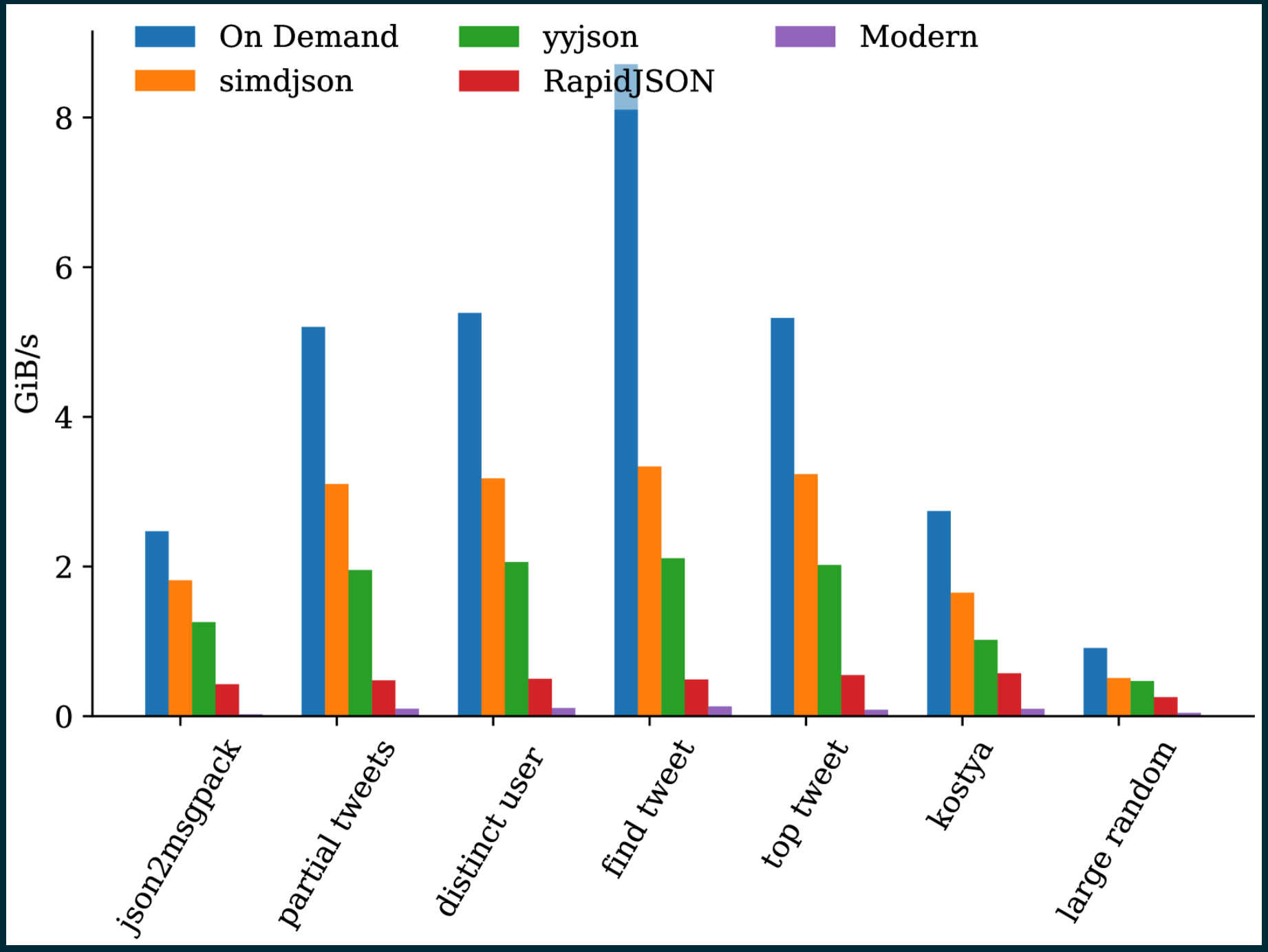
- 可配置,提高速度



2.4. 对比其他方案


parsing speed (GB/s)
3.0
2.5
2.0
1.5
1.0
0.5
0.0
simdjson
RapidJSON
sajson
apache_builds
canada
citm_catalog
github_events
gsoc-2018
instruments
marine_ik
mesh
mesh.pretty
numbers
random
twitter
twitterescaped
update-center

2.5. 按需 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

2.6. 用法

- 读取
  - simdjson
  - simdjson: On Demand (stream)
  - fastjson2
- 写入
  - rapidjson
  - fastjson2

3. 关于如何快速执行的想法?

3.1. 策略

- 取决于使用模式

3.1.1. 查询密集型

- 创建一个数据库 (ElasticSearch, MongoDB, PostgresSQL)
  - 创建一个 KV 存储
  - 加载一次并查询它

3.1.2. 选择性解析

- 选择性解析
  - NoDB
    - 无需解析即可查询数据,无需加载到数据库中
    - 像 grep 一样
  - JIT 技术
    - 查找模式和重复结构,编译用于特定查询的代码
    - 像编译器一样
  - Mison (由 Microsoft 开发)
    - 选择性解析,直接跳转到你想要的字段
    - 使用 SIMD 查找结构性重要字符,例如 "

3.2. 什么是公平竞争?

- JSON 解析类型
  - 非验证 JSON 解析器
    - 假设输入是有效的
    - 更容易
    - 大多数选择性解析是非验证的
  - 验证 JSON 解析器
    - 检查输入是否有效
    - 没有假设或格式错误的输入
      - 安全风险
      - 它只是被解析的错误数字或字符串
    - 更难,更复杂

3.3. JSON 的正确定义

```
/* 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. 强烈定义:BOOL、字符串、数字、NULL、对象和数组

```haskell
data JsonValue
    = Primitive PrimitiveValue
    | Container ContainerValue
-- 6 primitives --------------------------
data PrimitiveValue
    = Boolean Bool  -- true | false
    | String Text   -- "string"
    | Number Double -- 123, 1.23, 123e0, 123E0
    | Null          -- null
data ContainerValue
    = Object Object -- { "string", PrimitiveValue, ... }
    | Array Array   -- [ PrimitiveValue, ... ]
-- END -----------------------------------
newtype Object = Object [(Text, JsonValue)]
newtype Array = Array [JsonValue]
```

3.5. 强烈定义:BOOL、字符串、数字、NULL、对象和数组

3.5.1. 数字限制和整数

```
// 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
};
```

3.5.2. 浮点数和科学计数法

```
// 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
    }
};
```

3.6. 字符串:处理转义引号和 UTF-8

- 一些惰性解析器为了简单起见,假定为 ascii
  - 128 种可能性,只有 8 位
  - 假设输入没有日语、中文或奇怪的字符
- RFC 标准说字符串是 UTF-8
- 转义双引号 “Tom said: \”hello\“.”
  - Tom said: “hello”.
  - ’\’ 的数量
    - 奇数 -> 转义, “\”“ -> ”
    - 偶数 -> 未转义, “\\” -> \
- 在 " 之外,只能有 4 种空格符
  - “ ” | “\t” | “\r” | “\n”

3.6.1. ASCII 码

- 代码点 0x00 - 0xEF 127 种可能性

Bits $b_7 b_6 b_5$ →	000	001	010	011	100	101	110	111
b_4 b_3 b_2 b_1 ↓ / Row ↓ Column →	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

3.6.2. 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
```

3.7. 需求总结

- 数字
  - 负数 + -
  - 浮点数 1.23
- 字符串
  - utf-8
  - 转义引号 \“ | \\”
- 结构良好
  - 有效的空格
  - 有效的括号 {}, []

4. 挑战

4.1.为其编写解析器

- 递归下降类型解析器
- 需要许多 if else,是否可以在没有任何分支的情况下完成?

```
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'
    else:
        raise ValueError(f"Invalid JSON character at position {index}: {char}")
```

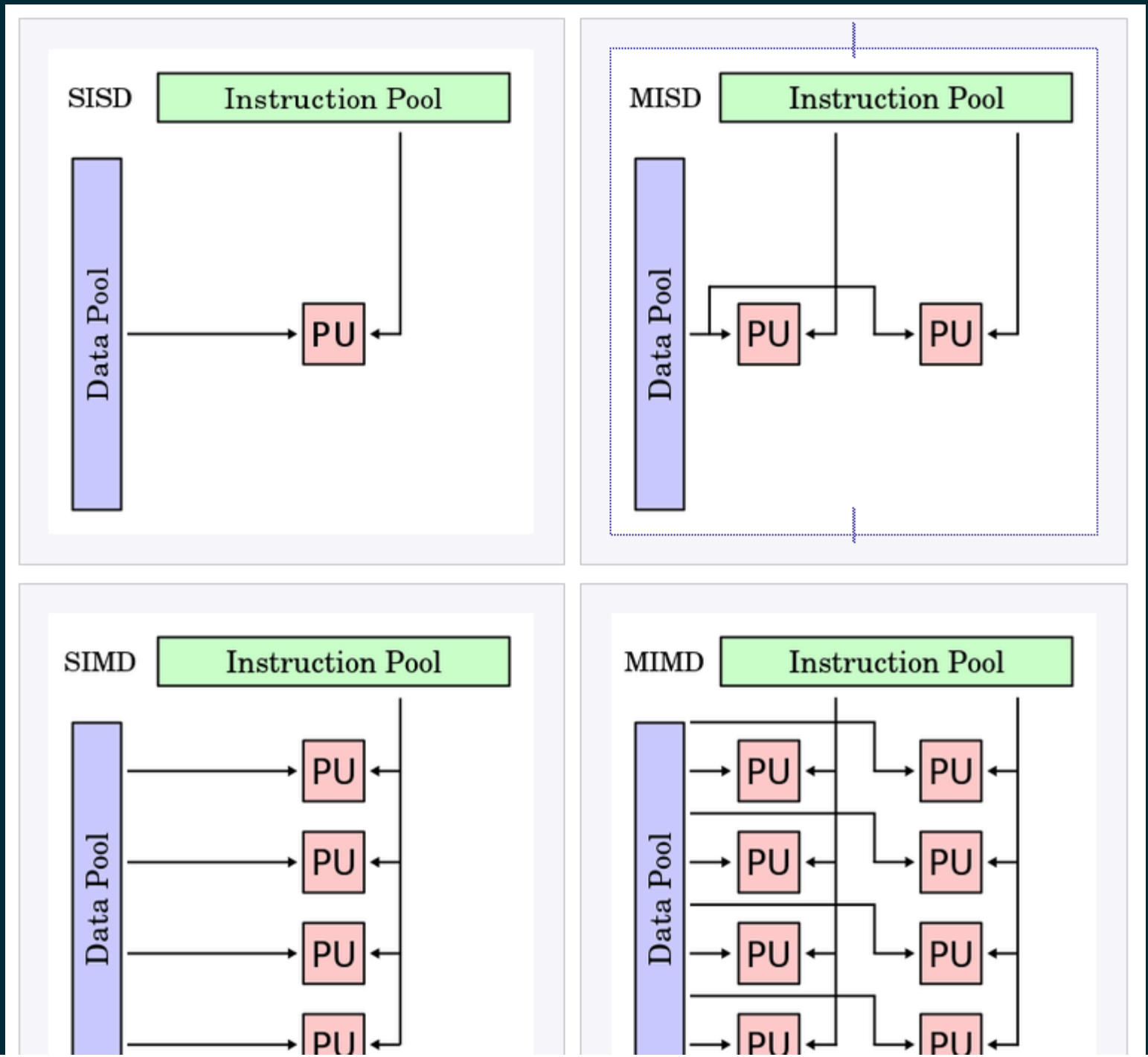
4.2. 鉴于挑战,如何快速完成?

- SIMD,一次处理超过 8 个字节。
  - 无分支代码,没有 if 语句。CPU 错误预测分支。
    - 正确,0-1 个周期
    - 分支错误,20 个周期

5. 关于 SIMD

simd 如何融入所有这些?

5.1. 什么是 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 示例

```
同时添加 4 个数字:
标量:
A: [5] + [3] = [8]      步骤 1
B: [7] + [2] = [9]      步骤 2
C: [4] + [6] = [10]     步骤 3
D: [1] + [8] = [9]      步骤 4
SIMD:
[5|7|4|1] +
[3|2|6|8] =    步骤 1
[8|9|10|9]     完成!
```

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)
```

- 流式 SIMD 扩展
  - XMM0-XMM15
- 高级向量扩展 2
  - YMM0-YMM15
- 高级向量扩展 512
  - ZMM0-ZMM15

5.4. SIMD 代码并没有那么可怕

Westmere 使用 128 位 SSE 指令 (_mm_shuffle_epi8) Haswell 使用 256 位 AVX2 指令 (_mm256_shuffle_epi8) Ice Lake 使用 512 位 AVX-512 指令 (_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. 一些 SIMD 示例

Intrinsic Function	Instruction	Description
_mm256_add_epi8(a, b)	VPADDB	Add packed 8-bit
_mm256_add_epi16(a, b)	VPADDW	Add packed 16-bit
_mm256_add_epi32(a, b)	VPADDD	Add packed 32-bit
_mm256_add_epi64(a, b)	VPADDQ	Add packed 64-bit
_mm256_sub_epi64(a, b)	VPSUBQ	Subtract packed 64-bit
_mm256_mullo_epi32(a, b)	VPMULLD	Multiply packed 32-bit
_mm256_mulhi_epi16(a, b)	VPMULHW	Multiply packed 16-bit
_mm256_and_si256(a, b)	VPAND	Bitwise AND of 256 bits
_mm256_or_si256(a, b)	VPOR	Bitwise OR of 256 bits
_mm256_xor_si256(a, b)	VPXOR	Bitwise XOR of 256 bits
_mm256_andnot_si256(a, b)	VPANDN	Bitwise AND NOT of 256 bits
_mm256_slli_epi64(a, imm8)	VPSLLQ	Shift packed 64-bit
_mm256_srli_epi64(a, imm8)	VPSRLQ	Shift packed 64-bit

5.6. SIMD 的闪光点

- 规则的,可预测的数据模式
- 简单的数学运算
- 连续的内存块
- 多个数据点上的相同操作
- 高吞吐量

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

5.7. SIMD 的阿喀琉斯之踵:分支

- 如果类似解析中的逻辑很复杂,则无法进行 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
    );
}
```

5.7.1. 正确的分支预测

```
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
```

5.7.2. 分支预测错误

- 示例成本 3 个周期,但实际 cpu 成本 7-15 个周期

```
Time →      FLUSH
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 EX -- -- --          | Instruction 3 (wrong path)
         IF ID -- -- --          | Instruction 4 (wrong path)
            IF ID EX ME WB    | Correct Instruction 2
               IF ID EX ME WB | Correct Instruction 3
```

5.7.3. 算术布尔值

- 实际上,当您执行 -o2 和 -o3 时,LLVM 会为您执行此操作

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

5.7.4. 选择索引

- 实际上,当您执行 -o2 和 -o3 时,LLVM 会为您执行此操作

```
// 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;
```

5.7.5. 如果 LLVM 为您执行此操作,那有什么意义?

- LLVM 尽力而为,但它找不到所有内容
  - 擅长小案例
- 一些较大的复杂模式
  - 人工模式识别
  - 批处理操作可以使用 simd

5.8. 编写无分支代码(按位运算)

5.8.1. 棘手的内存布局

```
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
```

5.8.2. 掩码

```
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
```

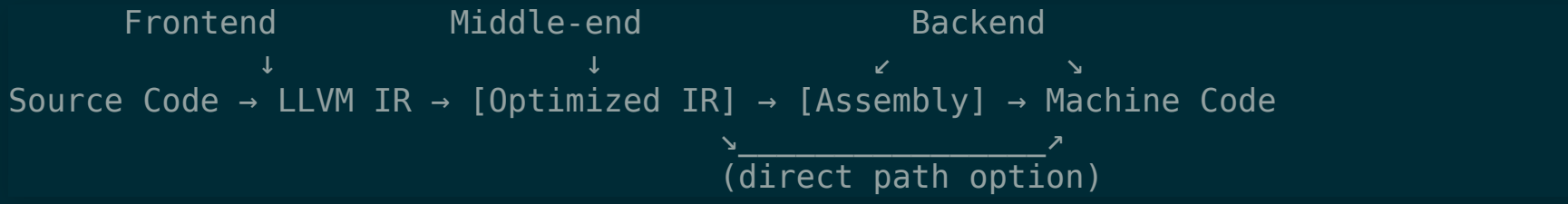
5.8.3. 取消设置最右边的位(BLSR)

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

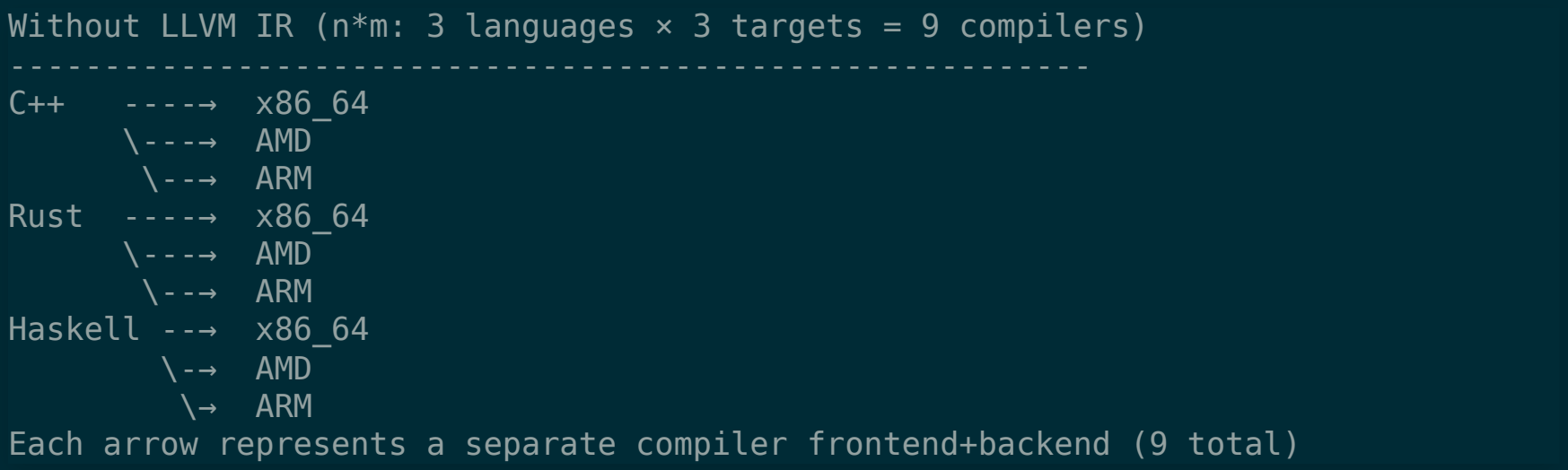
- 通用 cpu 操作,编译器优化为 `blsr`

5.9. LLVM 编译器

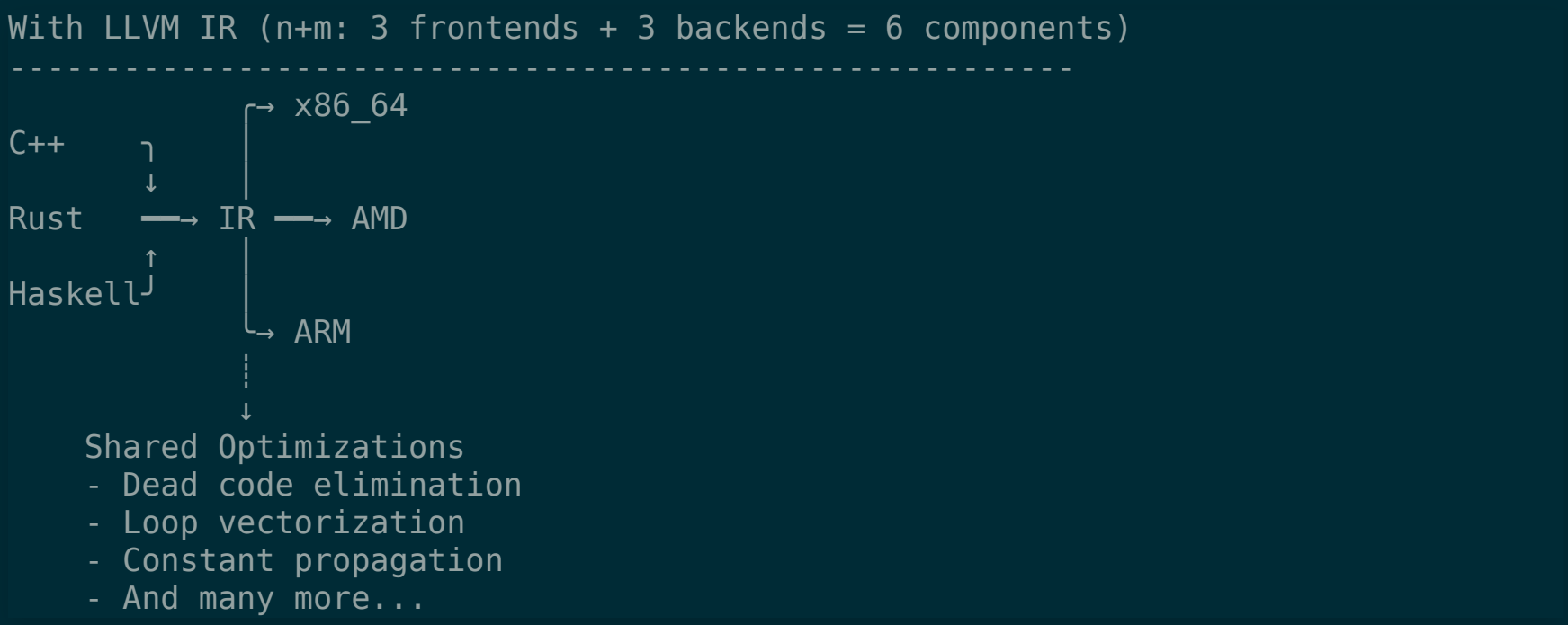
5.9.1. LLVM


Frontend
Middle-end
Backend
Source Code → LLVM IR → [Optimized IR] → [Assembly] → Machine Code
(direct path option)

5.9.2. 没有 LLVM IR



5.9.3. 使用 LLVM IR



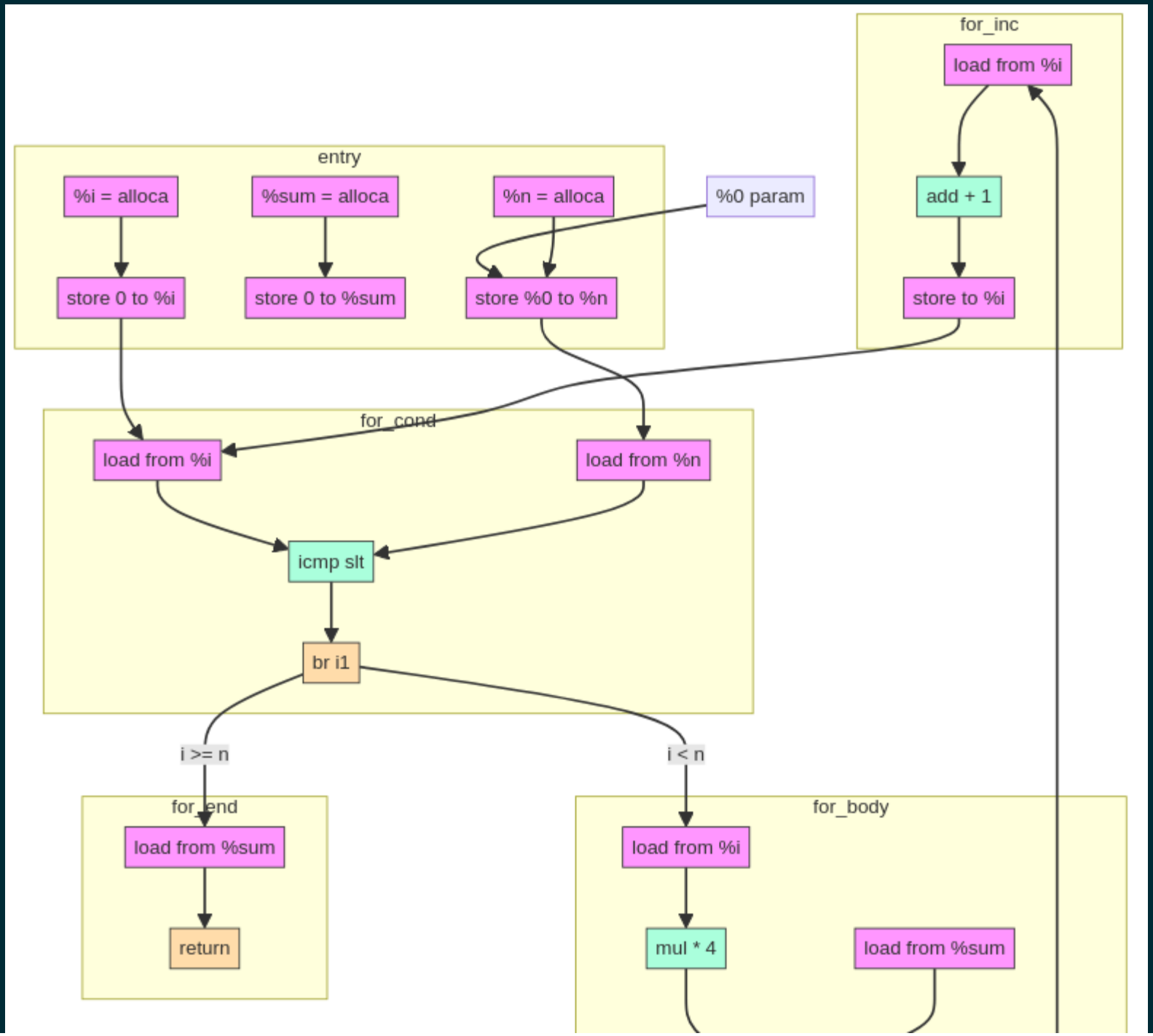
5.9.4. 中间表示示例(IR)

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

5.9.5. 未优化 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
}
```

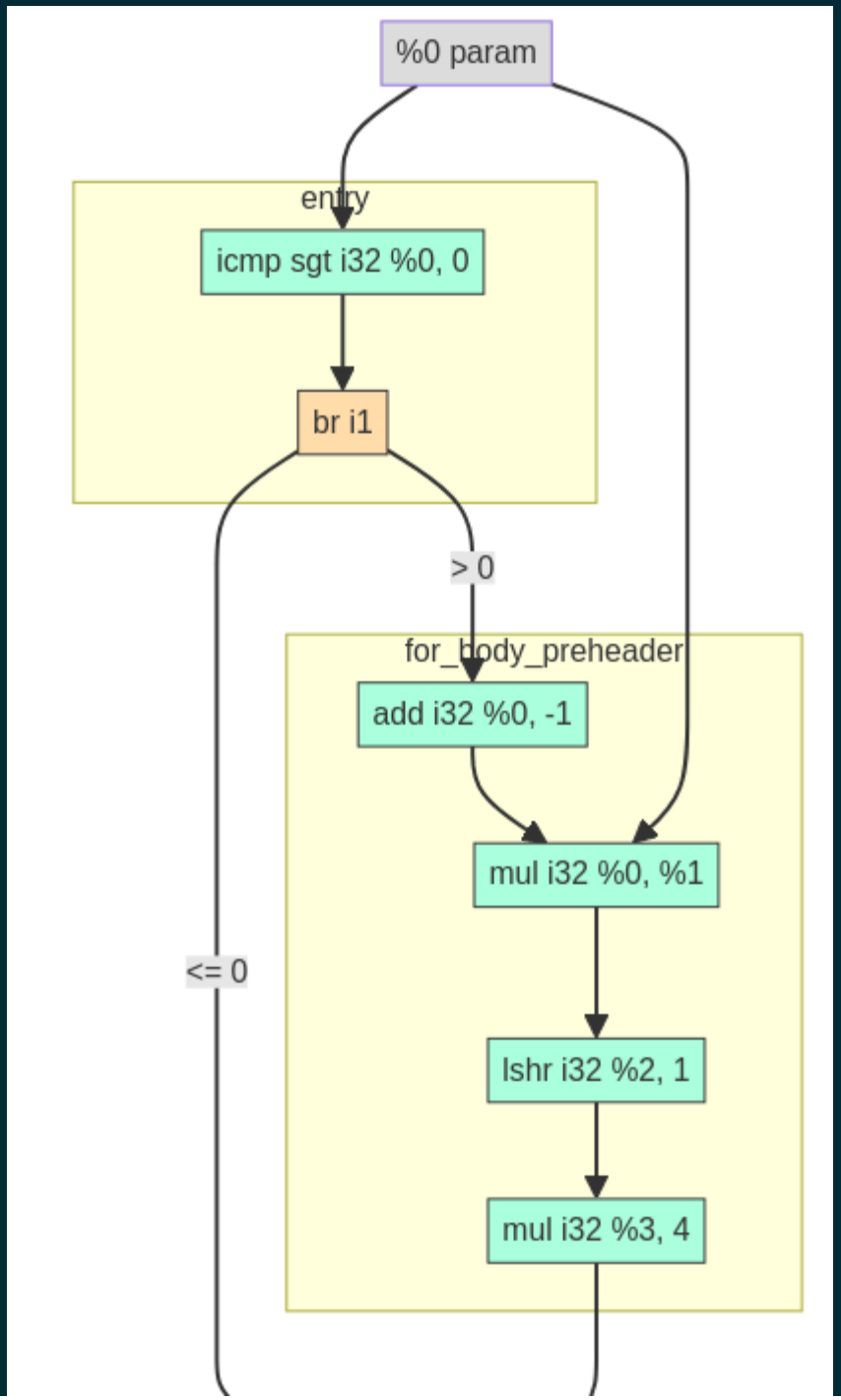
5.9.6. 未优化 IR -O0 图


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

5.9.7. 优化 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
}
```

5.9.8. 优化 IR -O2 图

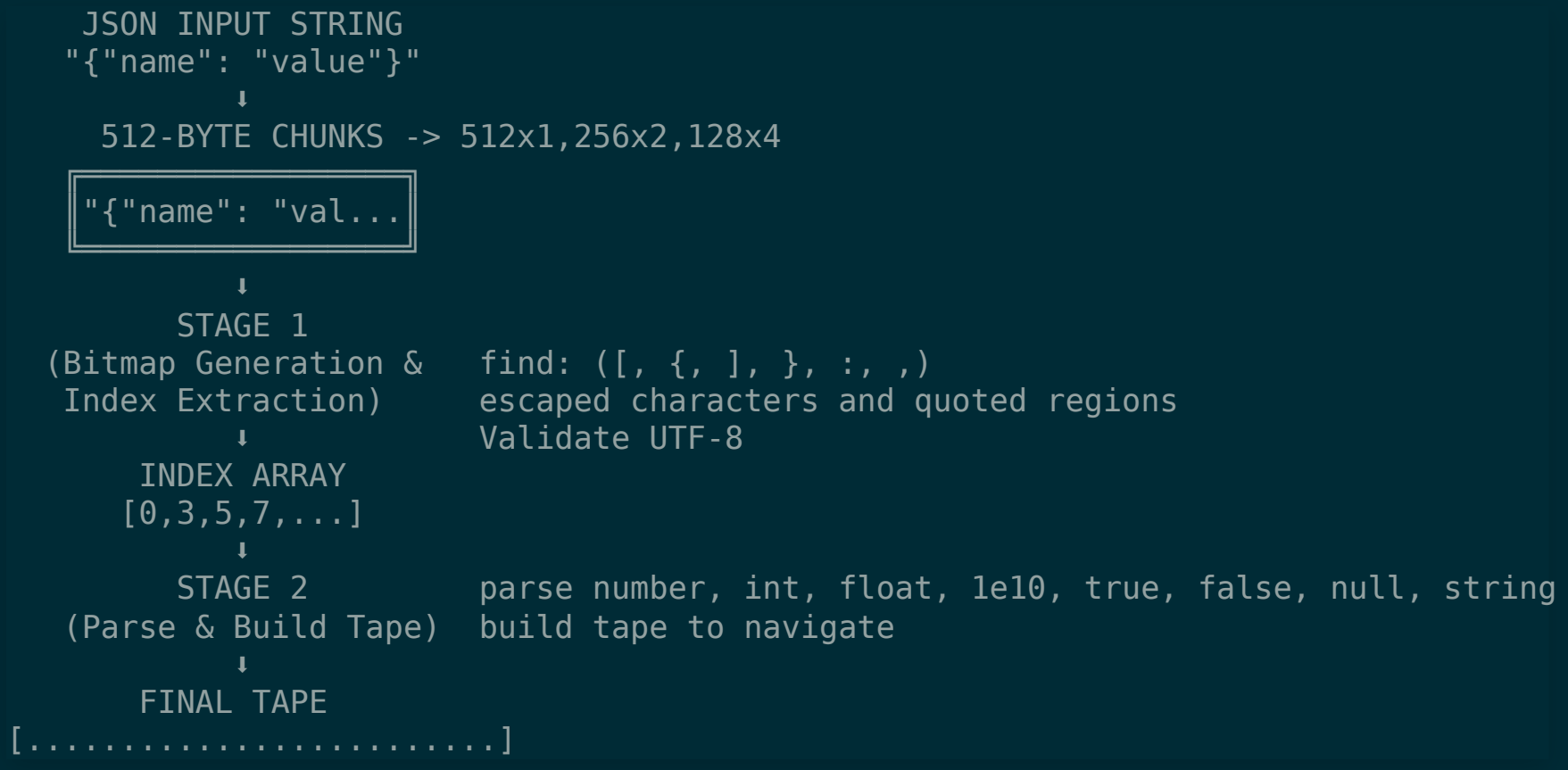

%0 param
entry
icmp sgt i32 %0, 0
br i1
> 0
<= 0
for_body_preheader
add i32 %0, -1
mul i32 %0, %1
lshr i32 %2, 1
mul i32 %3, 4

6. SIMDJSON 实现

6.1. SIMDJSON 架构概述

1. 阶段 1:结构索引创建(查找重要标记的位置)
    1. 查找结构字符 ({,},[,],",,:)
    2. 定位空格
    3. 识别字符串边界
    4. 验证 UTF-8 编码
2. 阶段 2:解析和磁带构建
    1. 验证文档结构
    2. 构建可导航磁带表示
        1. 解析原子值(字符串,数字,true/false/null)
        2. 将数字转换为计算机格式
        3. 将字符串规范化为 UTF-8

6.2. SIMDJSON


JSON INPUT STRING
"{"name": "value"}"
512-BYTE CHUNKS -> 512x1,256x2,128x4
"{"name": "val...
STAGE 1
(Bitmap Generation &
Index Extraction)
find: ([, {,], }, :, ,)
escaped characters and quoted regions
Validate UTF-8
INDEX ARRAY
[0,3,5,7,...]
STAGE 2
(Parse & Build Tape)
parse number, int, float, 1e10, true, false, null, string
build tape to navigate
FINAL TAPE
[.....]

6.3. 阶段 1:结构和伪结构索引构建

6.3.1. 输入与输出

- 输入:原始 JSON 字节
- 输出:
  - 结构字符的位掩码
  - 标记结构元素的整数索引数组

6.3.2. 主要职责

1. 字符编码验证 (UTF-8)
2. 定位结构字符 ([, {,], }, :, ,)
3. 识别字符串边界
    1. 处理转义字符和引用区域
4. 查找伪结构字符(如数字,true,false,null 等原子)

6.4. 阶段 2:结构化导航

6.4.1. 输入与输出

- 输入:来自阶段 1 的结构索引数组
- 输出:在“磁带”(数组)上解析的 JSON 结构
- 目的:构建 JSON 文档的可导航表示

6.4.2. 主要职责

1. 解析字符串并转换为 UTF-8
2. 将数字转换为 64 位整数或双精度数
3. 验证结构规则(匹配括号,正确序列)
4. 构建可导航的磁带结构

6.4.3. 磁带格式

- 每个节点 64 位字
- 不同类型的特殊编码:
  - 原子(null,true,false): n/t/f × 2^56
  - 数字:两个 64 位字
  - 数组/对象:具有导航指针的开始/结束标记
  - 字符串:指向字符串缓冲区的指针

7. 阶段 1:结构和伪结构索引构建

_mm256_shuffle_epi8 == VPSHUFB

```
// _mm256_shuffle_epi8 == VPSHUFB
const auto whitespace_table = simd8<uint8_t>::repeat_16(' ', 100, 100, 100, 17, 100, 113, 2, 10
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. 阶段 1:1 向量化分类和伪结构字符

- 想要获取结构字符的位置 ({, }, [,], :, ,)
  - 伪结构 - 紧跟在结构字符或空格之后的任何非空格字符
  - 用于分析,我们需要此位掩码来构建磁带

```
{ "\\\"Nam[{": [ 116,"\\\\" , 234, "true", false ], "t":"\\\"" }
**1**__________1__1_______1___1______1_______1__1______
_____0_____________________________________________0___ escaped quotes "
```

7.1.1. 向量化分类

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

```
HIGH_4 AND LOW_4 == 0000 0100 // it must be a bracket
{ "\\\"Nam[{": [ 116,"\\\\" , 234, "true", false ], "t":"\\\"" }
____________________1______1___________1_______1_____________ comma mask
_____________1_____________________________________1________ colon mask
1________11___1_____________________________________1_____________1 bracket mask
```

7.1.2. VPSHUFB:向量排列混排字节

- 基本上是一个使用四种最低有效位(半字节)的单指令查找表
  - 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'
    );
    // 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>
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. 简单示例

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. 简单示例

```
"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. 简单示例

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

7.1.6. 阶段 1:位图到数组索引

```
{ "\\\"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
```

- 以 Q 为例,我们要将 Q 的位掩码转换为索引列表
  - [2, 12, 22, 27, 37, 42, 54, 56, 58, 62]

7.1.7. 提取

- 2 条说明
  - TZCNT 计算尾随最不重要的 0 位
  - BLSR 删除最后设置的位。

```
a = 1010000
idx = tzcnt(a) // 4       count 0 after lowest bit
a = blsr(a)    // 1000000 remove lowest set bit
idx = tzcnt(a) // 6       count 0 after lowest bit
[4, 6]
```

7.1.8. 朴素的实现

```
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);
    }
}
```

7.1.9. 最小分支实现

```
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);
        *output++ = __builtin_ctzll(bitset);
        bitset &= (bitset - 1);
    }
    // Reset output pointer to actual end based on real count
    output = next_base;
}
```

7.2. 阶段 1:2 消除转义或引用的子字符串

7.2.1. 获取反斜杠

```
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. 获取以奇数偏移量开头的奇数长度序列

```
{ "\\\"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_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_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_1_1_1_1_1_1: E (constant)
_____1_____________________________________________1___: OD2 = OCO & E
{ "\\\"Nam[{": [ 116,"\\\\" , 234, "true", false ], "t":"\\\"" }: input data
// this shows two odd-length sequence starting on an odd offset
```

7.2.3. 获取以偶数偏移量开头的奇数长度序列

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. 获取具有奇数偏移量的序列

```
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. 消除转义

```
{ "\\\"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. 获取引号之间的位置掩码

```cpp
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);
    }
    // 0000000000000000000100000000000000000000000000000000000000000000
    // after apply
    // 0000000000000000000011111111111111111111111111111111111111111111
    // 0001000000000000000000000000000000000000000000000000000000000000
    // after apply
    // 0001111111111111111111111111111111111111111111111111111111111111
    return result;
}
```

7.2.7. 清扫

```
// 00000000000000000001000000000000000000000000000000000000000000000000
// OR
// 00010000000000000000000000000000000000000000000000000000000000000000
// result
// 00010000000000000001000000000000000000000000000000000000000000000000
// 00000000000000000000111111111111111111111111111111111111111111111111
// XOR
// 00011111111111111111111111111111111111111111111111111111111111111111
// result
// 00011111111111111110000000000000000000000000000000000000000000000000
```

7.2.8. 清扫

```
Final result:
0x00      00111111 11110000 00000111 11000000 00011111 00000000 00001100 11111000
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. 由 CLMUL, PCLMULQDQ 实现的清扫

- 无进位乘法器
- CLMUL(4, 15)
- 4 * 15

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

7.2.10. 由 CLMUL, PCLMULQDQ 实现的清扫

- 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. 最终获取引号掩码

```
{ "\\\"Nam[{": [ 116,"\\\\" , 234, "true", false ], "t":"\\\"" }: input data
**1111111111**_______11111_________11111____________11__11111___: CLMUL(Q,~0)
```

7.3. 阶段 1:3 字符编码验证

1. 初始 ASCII 快速路径,第一位 == 0
2. 主要算法
    1. 范围检查(0xF4 饱和减法)
    2. 连续字节验证

7.3.1. 检查 ASCII 快速路径

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

7.3.2. 连续字节验证

```
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. 映射到值(再次 VPSHUFB!)

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 验证算法

```
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 验证算法:无效示例

```
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. 阶段 2:构建磁带

8.1. 阶段 2:磁带

8.1.1. 磁带条目的三个类别

1. 直接值(原子)
    - null、true、false
    - 数字(整数和浮点数) - 占用 2 个磁带条目
2. 字符串引用
    - 指向单独的字符串缓冲区
    - 不是原始 JSON 字符串
3. 结构化导航
    - 数组括号 [,]
    - 对象大括号 {,}
    - 包含跳转索引

8.1.2. 基本结构

- 磁带是 64 位字的数组
- 每个条目:=TYPE_MARKER × 2^56 + payload=
- 高 8 位:类型信息
- 低 56 位:值或引用

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

8.1.3. 直接値(原子)

```
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. 数字:整数示例(42)

占用 2 个磁带条目: -第一个只是一个类型标记 -第二个是值

```
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. 数字:浮点示例(3.14)

占用 2 个磁带条目:

```
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: 0x4009219940000000
```

8.1.6. 字符串磁带条目

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

- 字符串缓冲区是一个单独的数组,用于存储标准化的 UTF-8 字符串
  1. 这种方法的好处
     - 快速的长度检索 - 无需在磁带中进行可变长度猜测搜索
     - 包含标准化的 UTF-8 字符串

8.1.7. 对象示例

```
{"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. 数组示例

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

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

8.1.9. JSON 文档

```
{
  "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. 磁带优势

- 缓存友好的线性布局
- 使用索引跳转快速导航
- SIMD 友好的处理
- 可预测的内存布局

8.2. 阶段 2:1数字解析

8.2.1. 了解 IS_ALL_DIGITS

快速 8 位数检查

```
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. 主要见解:0x29 到 0x3A 的 ASCII 字符

- 请注意,有效数字的所有高半字节均为 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. 步骤 1:高半字节的初始掩码

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

- 如果您小于 0x3X,则为 0x2F,
- 让我们采用有效输入“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. 低半字节检查的工作原理

- 我们要确保低半字节在 0xX0 - 0xX9 范围内
  - 0xXA - 0xXF 是非法的
    - 使用二进制分析进位检测

8.2.5. 情况 1:有效数字(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. 情况 2:无效字符(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. 步骤 2:添加 0X06 以检测非数字

```
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. 步骤 3:有效数字(0-9)的示例

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
                            ^  ^  ^  ^

If original > 9, carry effects the high nibble >3
Mask high:      30 30 30 30 30 30 30 30
Shift right 4:  03 03 03 03 03 03 03 03
OR together:    33 33 33 33 33 33 33 33
```

8.2.9. 步骤 4:无效字符(';' = 0X3B)的示例

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. 它为什么有效

1. 第一部分 (val & 0xF0F0...):
    - 隔离高半字节
    - 对于有效数字,必须为 0x30
2. 第二部分 ((val + 0x06...) & 0xF0...):
    - 将 0x06 添加到低半字节:
        - 对于 0-9:结果保持在半字节内
        - 对于>9:导致进位
    - 右移 4 位后:
        - 有效数字:始终为 0x03
        - 无效:不同的值
3. 当 OR 在一起时:
    - 有效数字:始终为 0x33
    - 无效:不同的模式

8.2.11. 有效案例

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

8.2.12. 无效案例

```
"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. 性能优势

- 单个比较而不是 8 个单独的检查
- 没有分支(对于现代 CPU 而言很重要)
- 使用本机 64 位操作
- 利用 CPU 并行检查的能力

此算法是位操作的一个很好的示例,它将通常为 8 个比较转变为单个数学测试。

8.2.14. 了解基于 SIMD 的快速八位数字数解析

使用 SIMD 指令将 8 位数的 ASCII 字符串转换为整数。示例:“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. 步骤 1:将 ASCII 转换为数值

```
__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. 步骤 2:将备用数字乘以 10 并添加

```
  __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. 步骤 3:将备用 16 位值乘以 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)
```

- 下一步是什么?10000?

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

8.2.18. 步骤 4:将 32 位值打包到 16 位

- 将值重新解释为 32 位而不是 16 位!?为什么?
- 因此我们可以使用~_mm_setr_epi16~而不是~_mm_setr_epi32~
    - 它更有效

```
    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. 步骤 5:使用乘以 10000 的最终组合

```
  __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. 摘要:为什么这么快

1. 并行处理:
    - 同时处理多个数字
    - 高效地使用 CPU 的 SIMD 功能
2. 指令计数:
    - 传统: ~8 个加载 + ~8 个乘法 + ~7 加法 ~23 个实例
    - SIMD: ~7 个总指令
3. Haswell 的延迟分析:
    - PSUBB(减):1 个周期
    - PMADDUBSW(乘法加法字节):5 个周期
    - PMADDWD(乘法加法字):5 个周期
    - PACKUSDW(打包):1 个周期
    - 总延迟:~17 个周期

9. 代码库中的实际 C++ 代码实现和优化技巧

9.1. SIMD8 零成本“抽象”

9.1.1. 质量至上的抽象

```
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. 质量至上的抽象

```
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. 模板元编程和 CRTP 与虚函数(动态绑定)

- **使用模板/CRTP 进行编译时多态性:**
  - **零成本抽象:** CRTP 模式允许编译器在编译时解析函数调用。
    - 来自 simdjson 的示例:

```
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);
  }
};
```

- 内联和优化
- 没有运行时重定向

9.2.1. 使用虚函数进行动态绑定

- **后期绑定:** 函数调用在运行时通过 vtable 解析。
  - 示例(代价值高的替代方法):

```
struct Base {
  virtual void foo() = 0;
  virtual ~Base() = default;
};
struct Derived : Base {
  void foo() override {
    // ... implementation ...
  }
};
```

- **运行时开销:**
  - 间接寻址
  - 无法内联
- **类似于 Java 界面:**

9.2.2. 为什么 C++ 选择编译时多态性

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. 内联函数和编译时内联

- **技术:** 函数标有 `simdjson_inline` 以鼓励内联。
- **为什么?:** 内联消除了小型,常用函数的函数调用开销。
- **来自 simdjson 的示例:**

```
#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);
}
```

- **注意:** 在所有小型操作(例如,算术,按位运算符)上使用内联可确保最大性能。

9.4. SIMDJSON 中的 C++ 强制转换:性能注意事项

- 在高性能 C++ 代码中,使用正确的强制转换对于安全性和速度至关重要。
- C++ 提供了几个强制转换运算符:
  - **static_cast**: 编译时转换。
  - **reinterpret_cast**: 低级,指针和位重新解释。
  - **const_cast**: 删除 constness。
  - **dynamic_cast**: 运行时检查的强制转换(带有 RTTI)。

9.4.1. CRTP 效率的 STATIC_CAST

- 在编译时已知,确保零成本抽象。

```
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;
  }
};
```

- 注意:
  - `static_cast<Child*>(this)` 将基类指针转换为派生类型。

9.4.2. SIMD 内存操作的 REINTERPRET_CAST

- 将原始内存(例如字节数组)重新解释为 SIMD 寄存器类型。
- 无法对类型检查进行静态强制转换

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

- **注意:**
  - 这些 reinterpret_cast 允许编译器生成有效的 SIMD 加载/存储指令。
  - 它们不会产生运行时成本,因为它们是在编译期间解析的。

9.4.3. 为什么不使用 DYNAMIC_CAST 或 CONST_CAST?

- **dynamic_cast:**
  - 执行运行时类型检查并产生额外的开销。
- **const_cast:**
  - const -> other typr

9.4.4. SIMDJSON 中的强制转换摘要

- **static_cast:**
  - 用于编译时转换(例如,CRTP 基类到派生类指针转换)。
  - 零成本和类型安全。
- **reinterpret_cast:**
  - 用于指针重新解释(例如,将字节数组转换为 SIMD 寄存器指针)。
  - 与低级别内在函数接口的必要组件。
- **避免的强制转换:**
  - **dynamic_cast** 和 **const_cast** 不用于性能关键部分,以防止不必要的运行时开销。

9.5. 为什么错误代码优于异常

- 零成本错误处理:无堆栈展开或 EH 表
- 更好的编译器优化:线性控制流
- 可预测的分支模式:CPU 管道友好
- 更小的代码大小:无异常处理元数据

```
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. 程序集比较:错误代码路径(SIMDJSON 样式)

```
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. 程序集比较:异常路径

```
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. 主要性能因素

1. **无 EH 表开销**
    - 异常处理需要 RTTI 和堆栈展开表
2. **CPU 分支预测**
    - 错误代码使用简单的条件分支
        - 异常创建不可预测的控制流
3. **内联友好**
    - 错误返回路径不会抑制函数内联
    - 对于 SIMD 优化至关重要

9.6. 内存对齐和填充

- 正确的内存对齐(和额外的填充)对于 SIMD 操作至关重要;未对齐的访问会严重损害性能。

```
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. 循环展开和向量化处理

- **关键思路:** 展开循环以手动执行更多操作

```
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. 编译器指令和特殊构建标志

- 编译器标志(例如,-O3 或 -march=native)和特定宏是释放峰值性能的关键。

9.9. C++优化摘要

- 零成本抽象
- 内联函数和强制转换
- 异常的错误代码
- 内存和循环优化

10. 谢谢您