Core C++ 2025

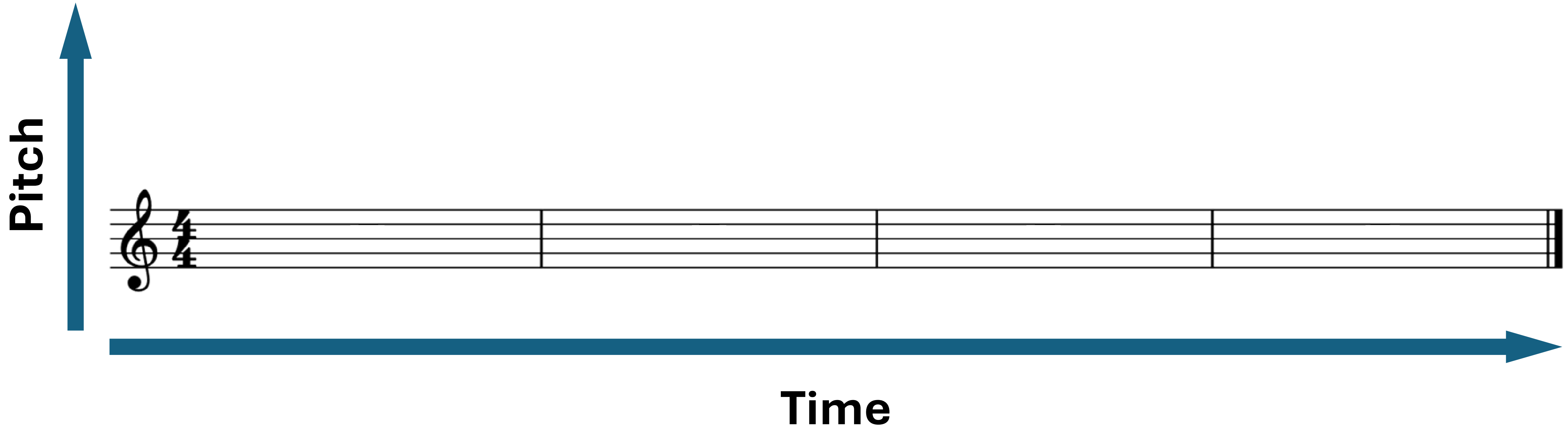
19 Oct. 2025 :: Tel-Aviv

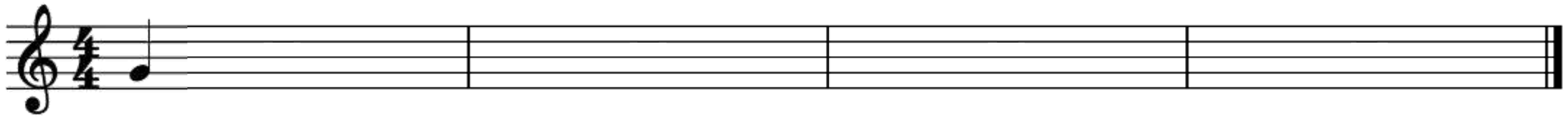
When the Structs Align … And When They Don’t

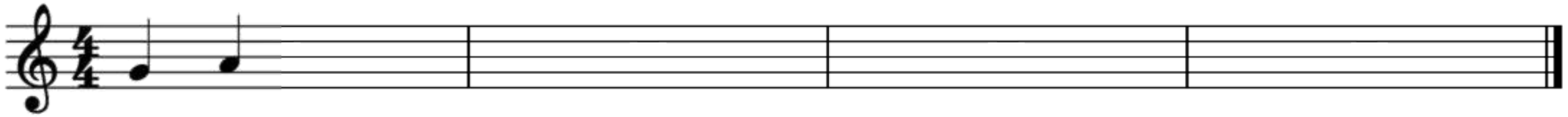
Tomer Vromen

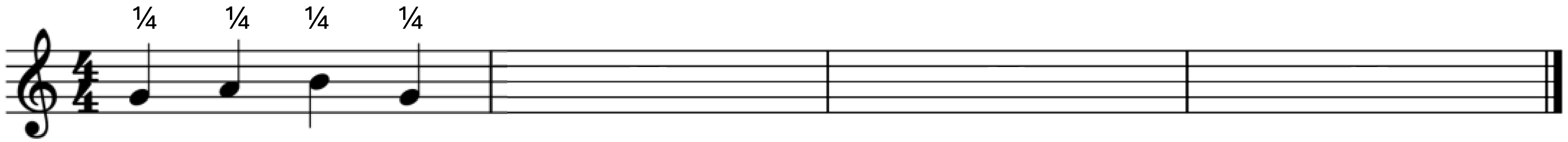
Musical Notation

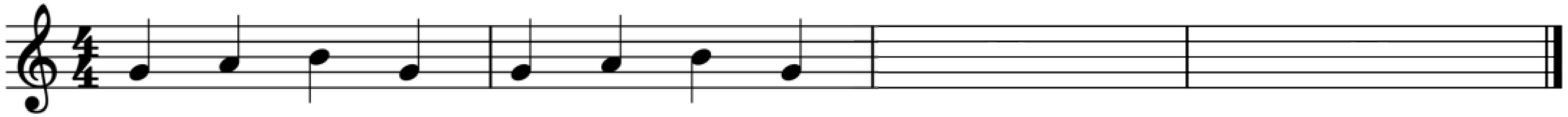
Photo by Mike Castro Demaria on Unsplash


Pitch
Time

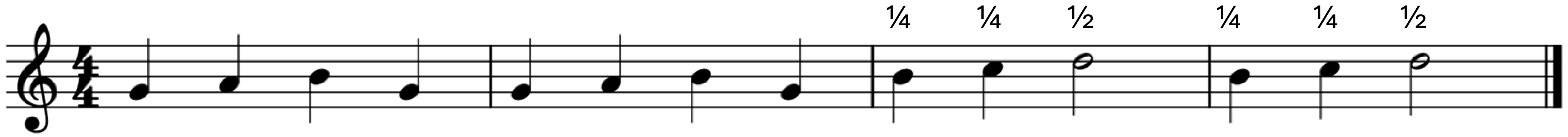




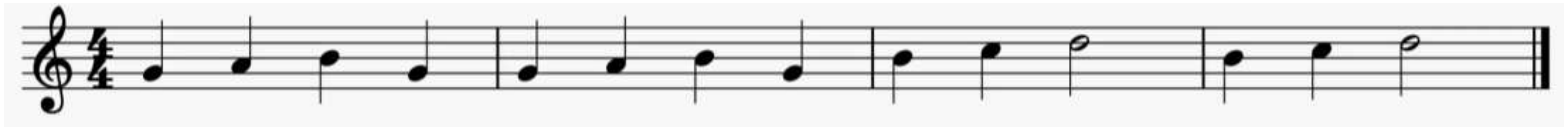

¼
¼
¼
¼



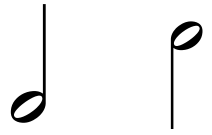
Frère Jacques



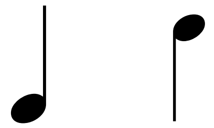
Frère Jacques

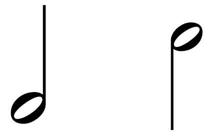
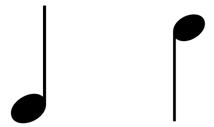
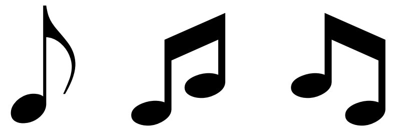


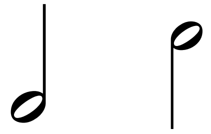
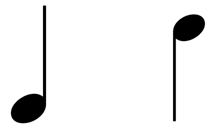
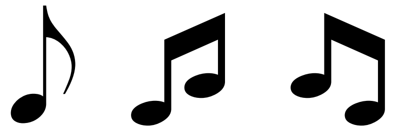
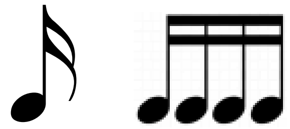
½ note

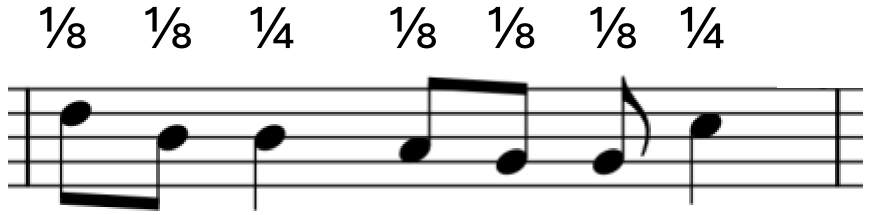


¼ note

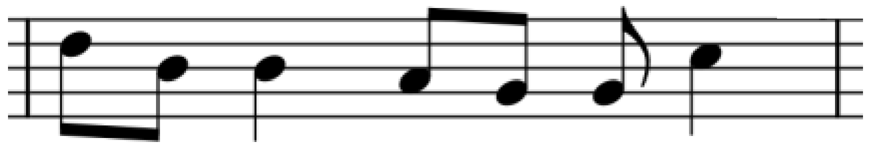


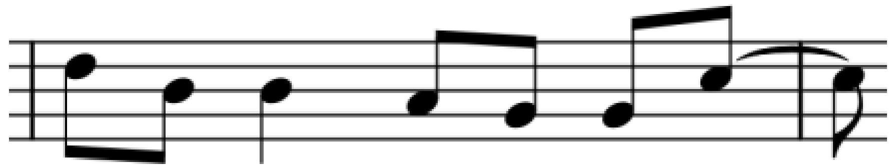
½ note	
¼ note	
⅛ note	

½ note	
¼ note	
⅛ note	
$^1/_{16}$ note	

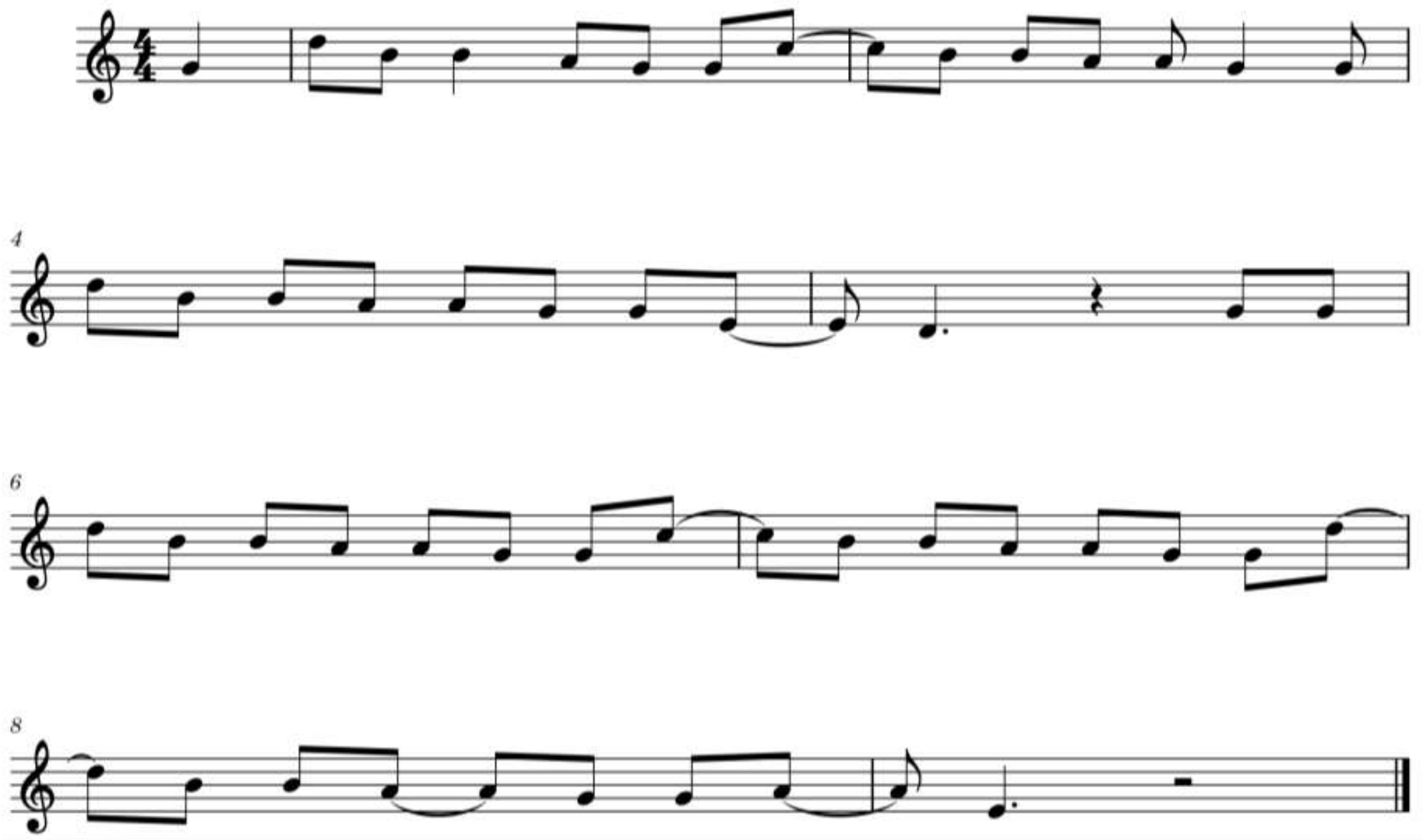

⅛
⅛
¼
⅛
⅛
⅛
¼

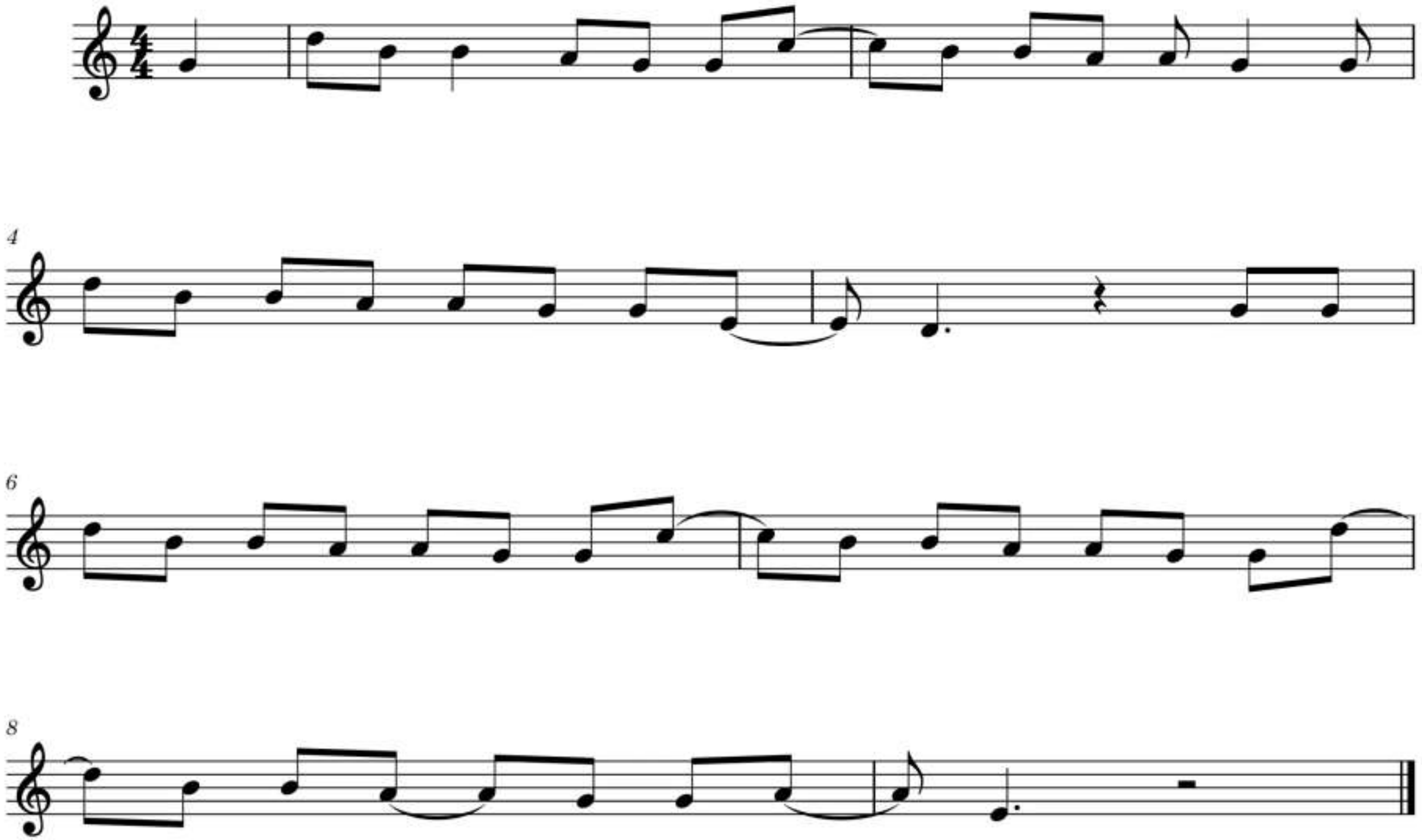
$\frac{1}{8} + \frac{1}{8} + \frac{1}{4} \quad + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{4} = 1\frac{1}{8}$





syncopation


4
6
8


4
6
8

A ♩ note is ***beat-aligned*** if it starts at

a whole multiple of ♩ from the start of the bar.

A ♩ note is ***beat-aligned*** if it starts at

a whole multiple of ♩ from the start of the bar.

Syncopated = not beat-aligned

An object x is ***N-byte-aligned*** if

its memory address is kN

where $N = 2^n$

An object x is ***N-byte-aligned*** if

```
(uintptr_t)&x % (1 << n) == 0
```

where $N = 2^n$

About Me: Tomer Vromen – תומר פרומן

Working @ DELL Technologies

C++, Python

PowerFlex Ultra

We're hiring

Haifa/Glil Yam/Be'er Sheva

→ `Tomer.Vromen@dell.com`

C++ Alignment Rules

Photo by Tom Wilson on Unsplash

Object types have ***alignment requirements*** which place restrictions on the addresses at which an object of that type may be allocated.

[basic.align]

An ***alignment*** is an **implementation-defined** integer value representing the number of bytes between successive addresses at which a given object can be allocated. [...]

Attempting to create an object in storage that does not meet the alignment requirements of the object's type is **undefined behavior**.

[basic.align]

An **`alignof`** expression yields the alignment requirement of its operand type.

[expr.alignof]

In a declaration, an **`alignas(...)`** attribute can be used to **increase** the default alignment requirement.

[dcl.align], paraphrased

Demo

https://godbolt.org/z/cM6exnMvo

Keeping Things Aligned

- Compiler ensures that all created objects are aligned according to C++ rules
- ABI = Abstract Binary Interface
  - Each platform has a different ABI
- ABI defines proper alignment
  - Constraints & invariants
- The x86_64 Stack Frame: “The end of the input argument area shall be aligned on a 16 byte boundary” (x86_64 ABI)

Keeping Things Aligned

- Global variables:
  - Compiler puts them in aligned position
- Stack-allocated (local) objects
  - **ABI promises** that stack is 16-byte aligned when control is transferred to the function entry point.
  - Higher alignment achieved by bitwise **AND**ing the stack register.

Keeping Things Aligned: Heap-Allocated

`MyClass *p = new MyClass{"hello", 42};`

1. Call **`operator new(sizeof(MyClass))`**
2. Call c'tor with arguments
   - The address (`this`) is the value returned by operator new

Calls to **`operator new(std::size_t)`** are guaranteed to be aligned by **`__STDCPP_DEFAULT_NEW_ALIGNMENT__`**

For larger alignment requirements,
`operator new(std::size_t, std::align_val_t)` is called. (since C++17)

Breaking the Rules

Photo by Tom Wilson on Unsplash

Attempting to create an object in storage that does not meet the alignment requirements of the object's type is **undefined behavior**.

[basic.align]

https://godbolt.org/z/KWd8qa5qb

Alignment in Practice

Photo by Tom Wilson on Unsplash

Alignment In Practice

CPU	Allowed?	Performance
Recent x86, x86_64 (Intel, AMD)	Yes	Good
ARMv8+	Yes	Good
POWER9+ (IBM)	Yes	Good

Modern architectures don't mind unaligned memory access!

Alignment In Practice

CPU	Allowed?	Performance
Recent x86, x86_64 (Intel, AMD)	Yes	Good
ARMv8+	Yes	Good
POWER9+ (IBM)	Yes	Good
x86, x86_64, Ivy Bridge & older	Yes	Depends

Modern architectures don't mind unaligned memory access!

Alignment In Practice

CPU	Allowed?	Performance
Recent x86, x86_64 (Intel, AMD)	Yes	Good
ARMv8+	Yes	Good
POWER9+ (IBM)	Yes	Good
x86, x86_64, Ivy Bridge & older	Yes	Depends
POWER8	No	---
SPARC	[illegible]	[illegible]
MIPS	[illegible]	[illegible]
ARM M-ser [illegible]	[illegible]	[illegible]
RISC-V	[illegible]	[illegible]

Modern architectures don't mind unaligned memory access!

Still relevant for older\embedded architectures

Breaks atomicity!

```
int prctl(PR_SET_UNALIGN, [illegible]signed long flag);
```

Pass **PR_UNALIGN_NOPRINT** to silently fix up unaligned user accesses, or **PR_UNALIGN_SIGBUS** to generate SIGBUS on unaligned user access.

Alignment In Practice ☆

Fundamental types

☆ ABI-defined

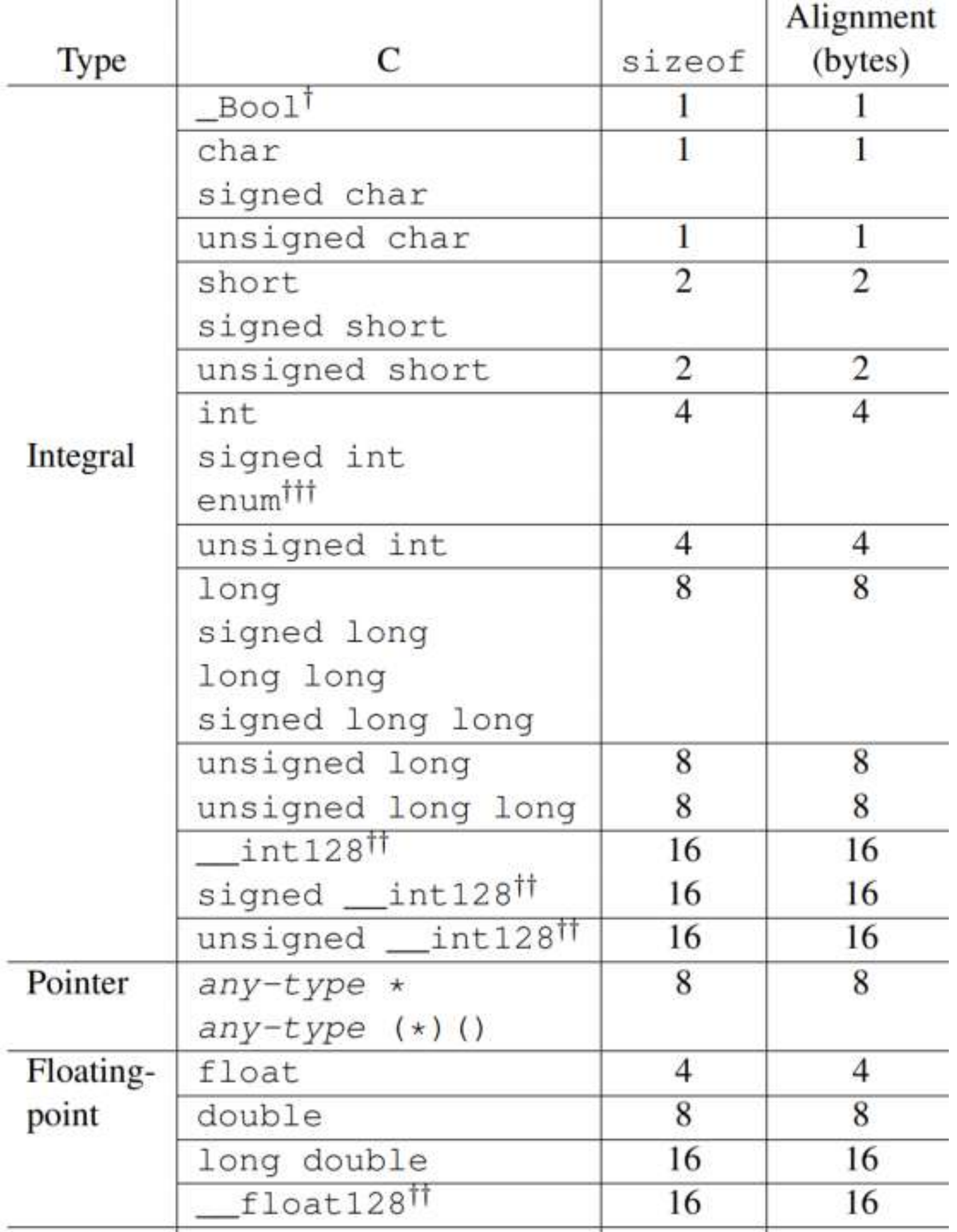
Alignment In Practice ☆

Fundamental types:

```
alignof(T) == sizeof(T)
```

Natural alignment

ABI for x86_64 --->



Type	C	sizeof	Alignment (bytes)
Integral	_Bool†	1	1
	char signed char	1	1
	unsigned char	1	1
	short signed short	2	2
	unsigned short	2	2
	int signed int enum†††	4	4
	unsigned int	4	4
	long signed long long long signed long long	8	8
	unsigned long unsigned long long	8 8	8 8
	__int128†† signed __int128††	16 16	16 16
	unsigned __int128††	16	16
Pointer	*any-type* * *any-type* (*)()	8	8
Floating-point	float	4	4
	double	8	8
	long double	16	16
	__float128††	16	16

☆ ABI-defined

Alignment In Practice ☆

Fundamental types:

```
alignof(T) == sizeof(T)
```

Natural alignment

☆ ABI-defined

Alignment In Practice ☆

Fundamental types:

```
alignof(T) == sizeof(T)
```

Natural alignment

Compound types (struct, class, union):

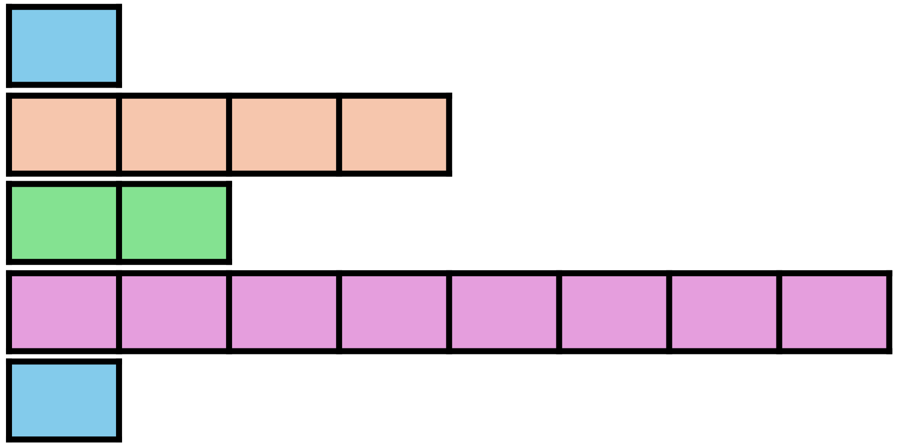
The alignment is that of the largest non-static member

☆ ABI-defined

Struct Alignment

The whole is greater than the sum of its parts

```
struct S
{
    char a;
    int b;
    short c;
    double d;
    char e;
};
```

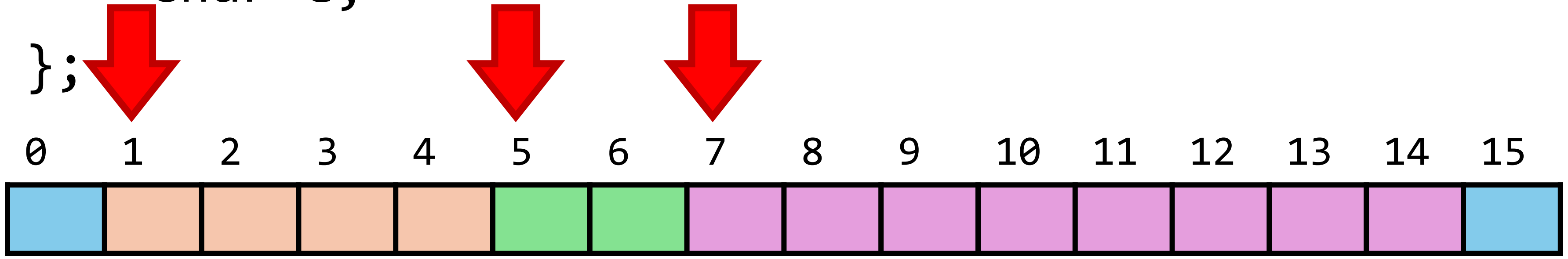


☆ ABI-defined

Struct Alignment

The whole is greater than the sum of its parts

```
struct S
{
    char a;
    int b;
    short c;
    double d;
    char e;
};
```

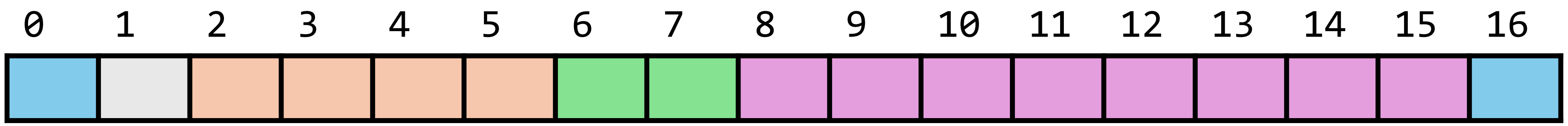


☆ ABI-defined

Struct Alignment

The whole is greater than the sum of its parts

```
struct S
{
    char a;
    int b;
    short c;
    double d;
    char e;
};
```

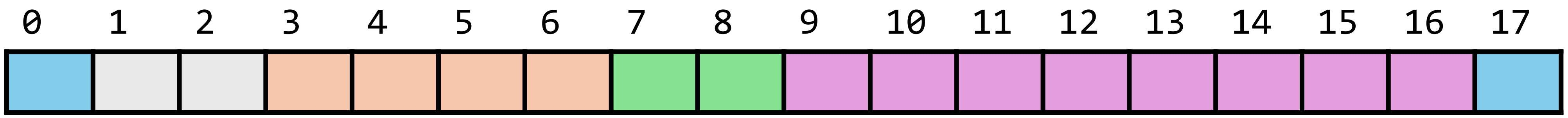


☆ ABI-defined

Struct Alignment

The whole is greater than the sum of its parts

```
struct S
{
    char a;
    int b;
    short c;
    double d;
    char e;
};
```

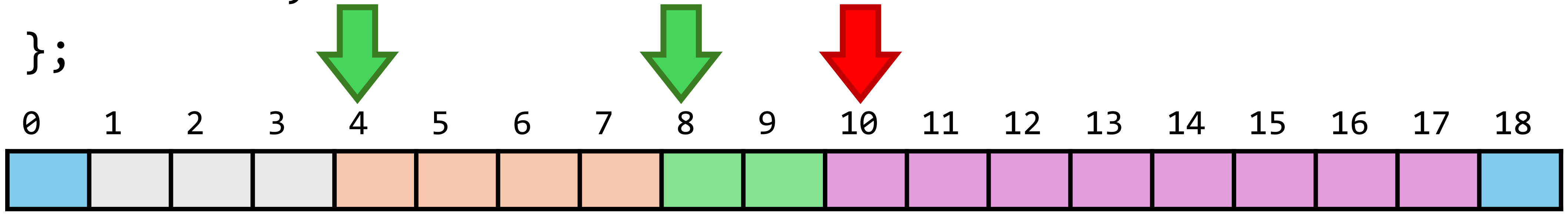


☆ ABI-defined

Struct Alignment

The whole is greater than the sum of its parts

```
struct S
{
    char a;
    int b;
    short c;
    double d;
    char e;
};
```

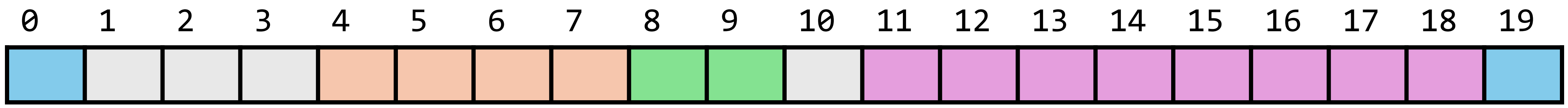


☆ ABI-defined

Struct Alignment

The whole is greater than the sum of its parts

```
struct S
{
    char a;
    int b;
    short c;
    double d;
    char e;
};
```

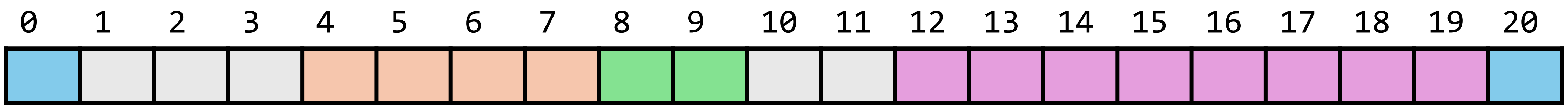


☆ ABI-defined

Struct Alignment

The whole is greater than the sum of its parts

```
struct S
{
    char a;
    int b;
    short c;
    double d;
    char e;
};
```

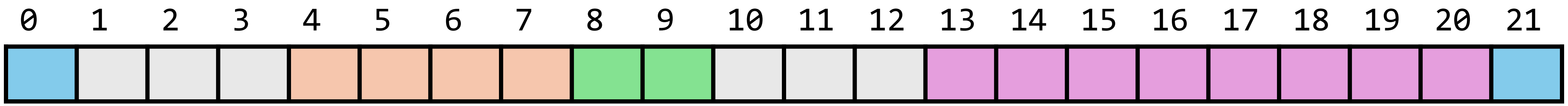


☆ ABI-defined

Struct Alignment

The whole is greater than the sum of its parts

```
struct S
{
    char a;
    int b;
    short c;
    double d;
    char e;
};
```

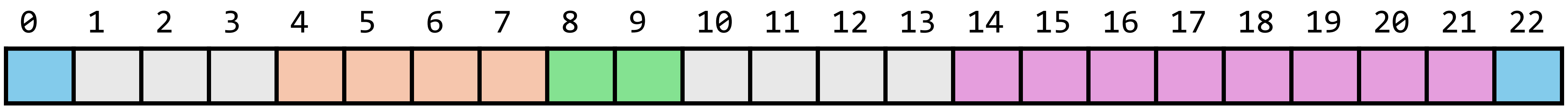


☆ ABI-defined

Struct Alignment

The whole is greater than the sum of its parts

```
struct S
{
    char a;
    int b;
    short c;
    double d;
    char e;
};
```

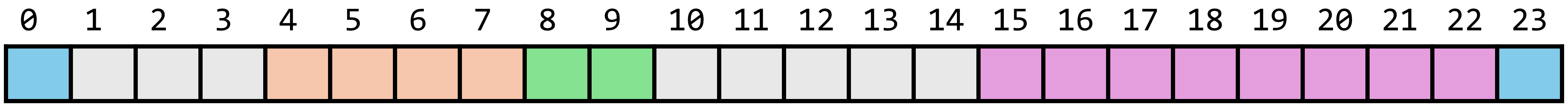


☆ ABI-defined

Struct Alignment

The whole is greater than the sum of its parts

```
struct S
{
    char a;
    int b;
    short c;
    double d;
    char e;
};
```



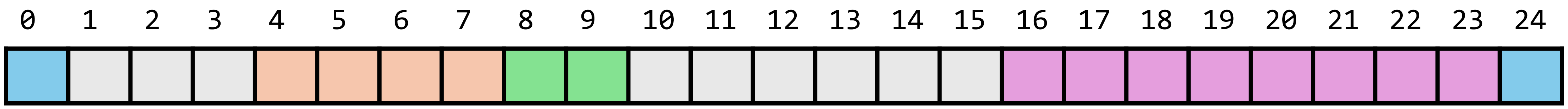
☆ ABI-defined

Struct Alignment

The whole is greater than the sum of its parts

```
struct S
{
    char a;
    int b;
    short c;
    double d;
    char e;
};
```

```
sizeof(S) == 25 ???
```



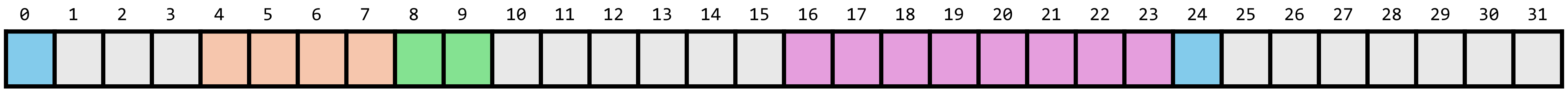
☆ ABI-defined

Struct Alignment

The whole is greater than the sum of its parts

```
struct S
{
    char a;
    int b;
    short c;
    double d;
    char e;
};
```

`sizeof(S) == 32`

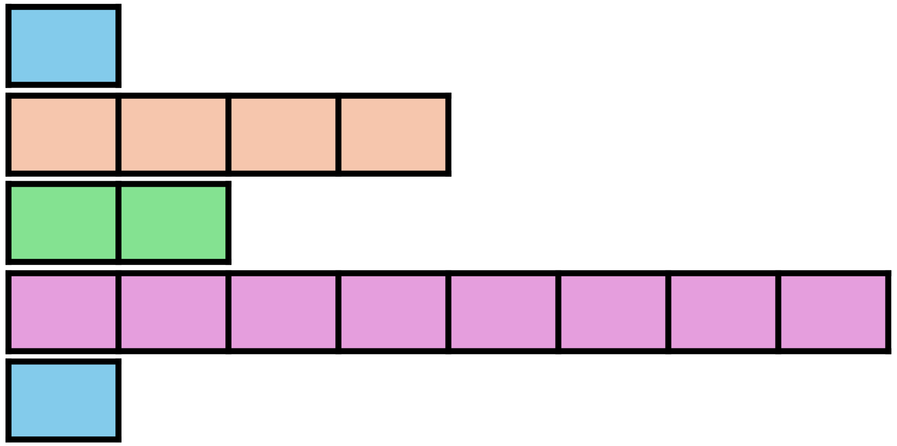


☆ ABI-defined

Struct Alignment

The whole is greater than the sum of its parts

```
struct S
{
    char a;
    int b;
    short c;
    double d;
    char e;
};
```

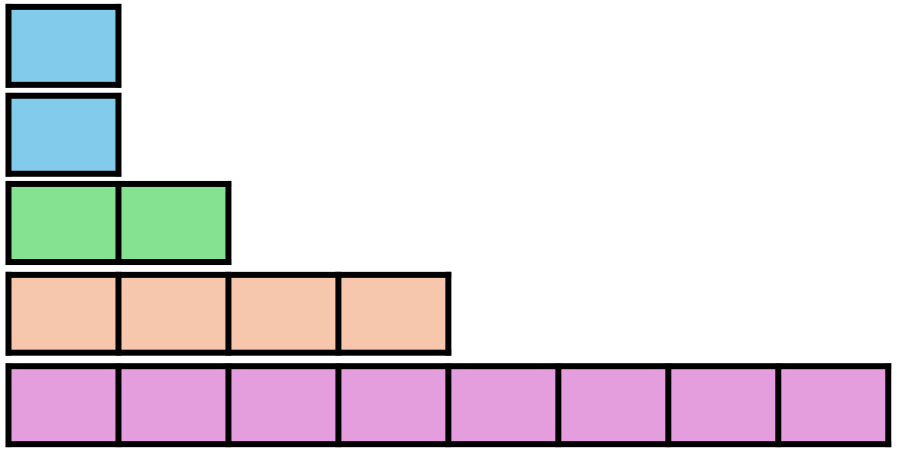


☆ ABI-defined

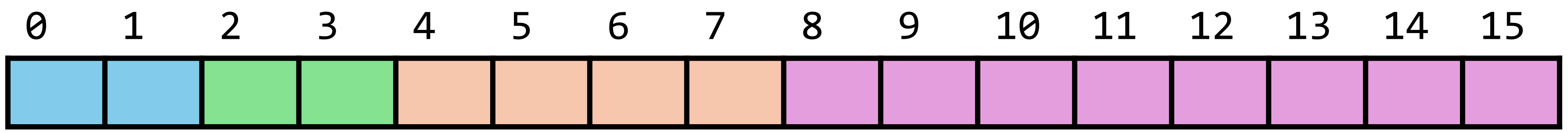
Struct Alignment

The whole is greater than the sum of its parts

```
struct S
{
    char a;
    char e;
    short c;
    int b;
    double d;
};
```



sizeof(S) == 16

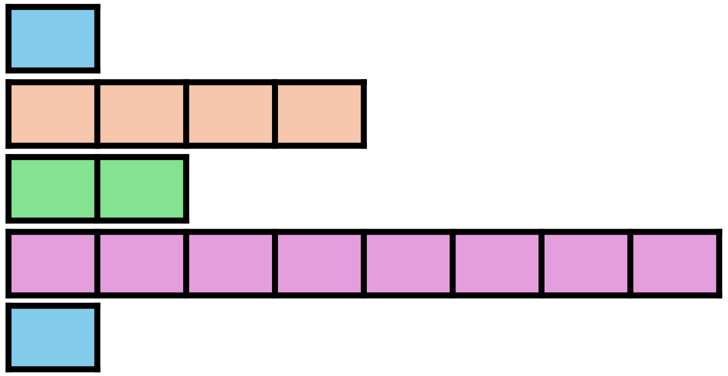


☆ ABI-defined

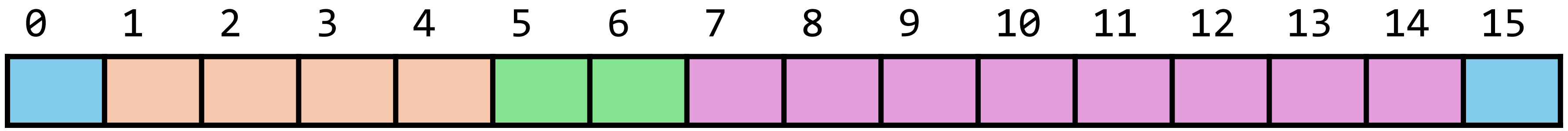
Struct Alignment

The whole is greater than the sum of its parts

```
#pragma pack(push, 1)
struct S
{
    char a;
    int b;
    short c;
    double d;
    char e;
};
#pragma pack(pop)
```



sizeof(S) == 16

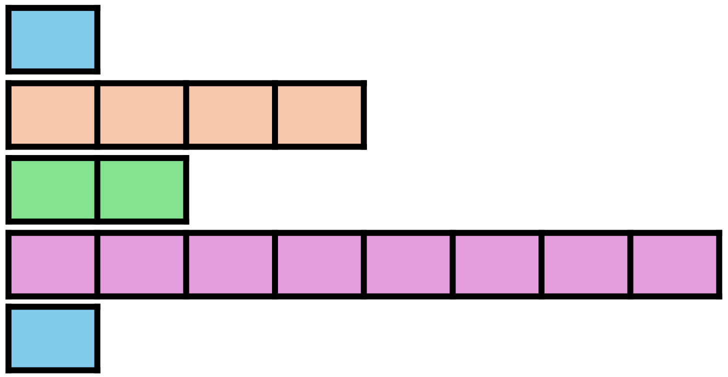


☆ ABI + compiler extension

Struct Alignment

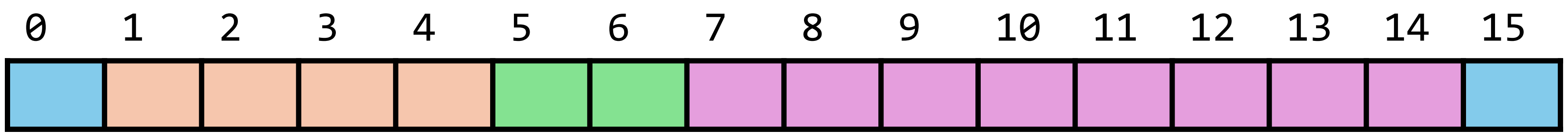
The whole is greater than the sum of its parts

```
#pragma pack(push, 1)
struct S
{
    char a;
    int b;
    short c;
    double d;
    char e;
};
#pragma pack(pop)
```



```
s.b = 42;
```

arm32 disassembly:

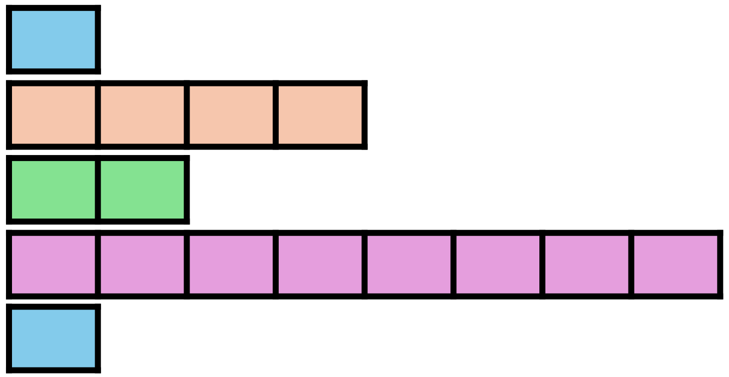


☆ ABI + compiler extension

Struct Alignment

The whole is greater than the sum of its parts

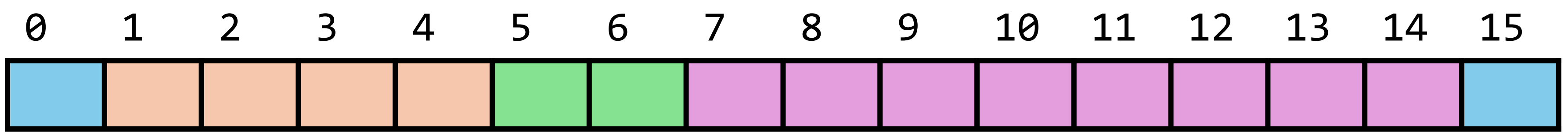
```
#pragma pack(push, 1)
struct S
{
    char a;
    int b;
    short c;
    double d;
    char e;
};
#pragma pack(pop)
```



```
s.b = 42;
```

arm32 disassembly:

```
    movs    r3, #0
    orr     r3, r3, #42
1   strb    r3, [r7, #1]
    movs    r3, #0
2   strb    r3, [r7, #2]
    movs    r3, #0
3   strb    r3, [r7, #3]
    movs    r3, #0
4   strb    r3, [r7, #4]
```

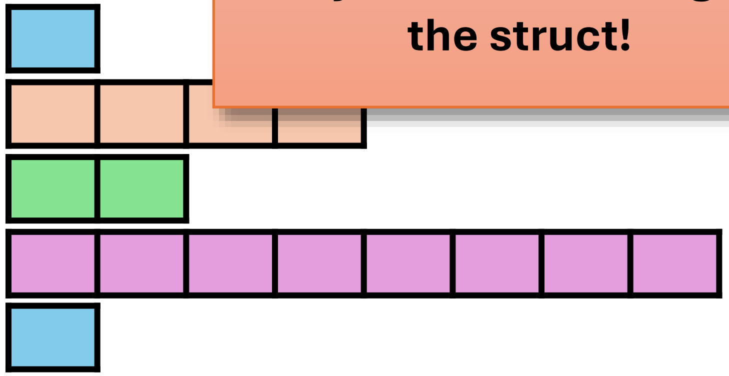


☆ ABI + compiler extension

Struct Alignment

The whole is greater than the sum of its parts

```
#pragma pack(push, 1)
struct S
{
    char a;
    int b;
    short c;
    double d;
    char e;
};
#pragma pack(pop)
```

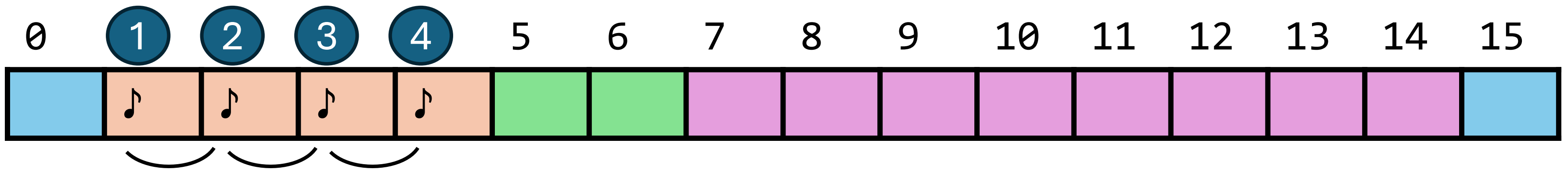


Only when accessing via the struct!

```
s.b = 42;
```

arm32 disassembly:

```
    movs    r3, #0
    orr     r3, r3, #42
    strb    r3, [r7, #1]
    movs    r3, #0
    strb    r3, [r7, #2]
    movs    r3, #0
    strb    r3, [r7, #3]
    movs    r3, #0
    strb    r3, [r7, #4]
```



☆ ABI + compiler extension

SIMD

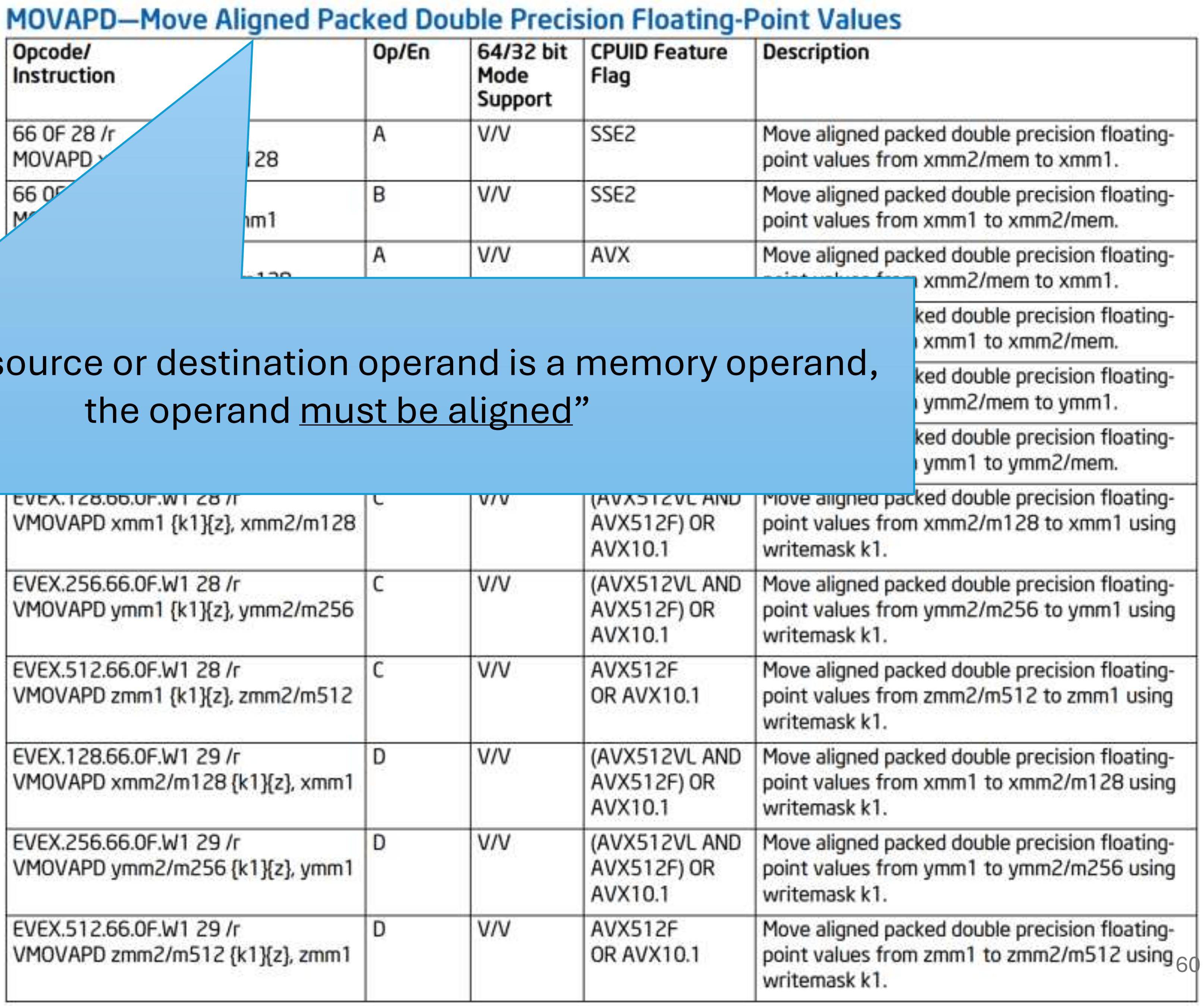
Single

Instruction

Multiple

Data

Intel’s documentation --->



MOVAPD—Move Aligned Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 28 /r MOVAPD x 28	A	V/V	SSE2	Move aligned packed double precision floating-point values from xmm2/mem to xmm1.
66 0F M nm1	B	V/V	SSE2	Move aligned packed double precision floating-point values from xmm1 to xmm2/mem.
	A	V/V	AVX	Move aligned packed double precision floating- xmm2/mem to xmm1.
				ked double precision floating- xmm1 to xmm2/mem.
				ked double precision floating- ymm2/mem to ymm1.
				ked double precision floating- ymm1 to ymm2/mem.
EVEX.128.66.0F.W1 28 /r VMOVAPD xmm1 {k1}{z}, xmm2/m128	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed double precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W1 28 /r VMOVAPD ymm1 {k1}{z}, ymm2/m256	C	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed double precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W1 28 /r VMOVAPD zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F OR AVX10.1	Move aligned packed double precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W1 29 /r VMOVAPD xmm2/m128 {k1}{z}, xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed double precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 29 /r VMOVAPD ymm2/m256 {k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed double precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 29 /r VMOVAPD zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1	Move aligned packed double precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

“When the source or destination operand is a memory operand, the operand must be aligned”

SIMD

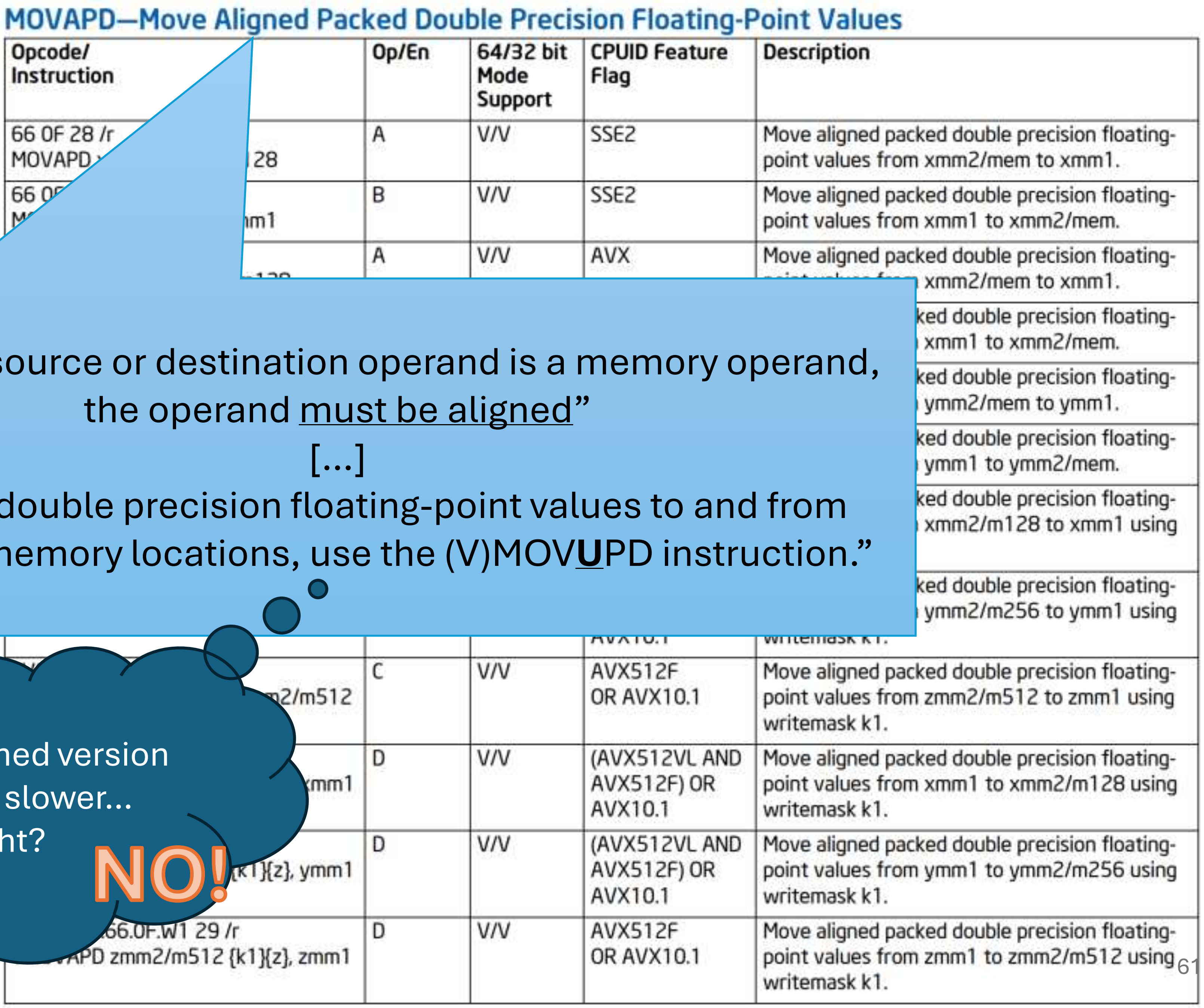
Single

Instruction

Multiple

Data

Intel's documenta



MOVAPD—Move Aligned Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 28 /r MOVAPD x 28	A	V/V	SSE2	Move aligned packed double precision floating-point values from xmm2/mem to xmm1.
66 0 M m1	B	V/V	SSE2	Move aligned packed double precision floating-point values from xmm1 to xmm2/mem.
128	A	V/V	AVX	Move aligned packed double precision floating- xmm2/mem to xmm1.
				ked double precision floating- xmm1 to xmm2/mem.
				ked double precision floating- ymm2/mem to ymm1.
				ked double precision floating- ymm1 to ymm2/mem.
				ked double precision floating- xmm2/m128 to xmm1 using
			AVX10.1	ked double precision floating- ymm2/m256 to ymm1 using writemask k1.
m2/m512	C	V/V	AVX512F OR AVX10.1	Move aligned packed double precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed double precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
{k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed double precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
66.0F.W1 29 /r APD zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1	Move aligned packed double precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

"When the source or destination operand is a memory operand, the operand <u>must be aligned</u>"

[...]

"To move double precision floating-point values to and from <u>unaligned</u> memory locations, use the (V)MOV**<u>U</u>**PD instruction."

The unaligned version must be slower... right?

NO!

SIMD

Single

Instruction

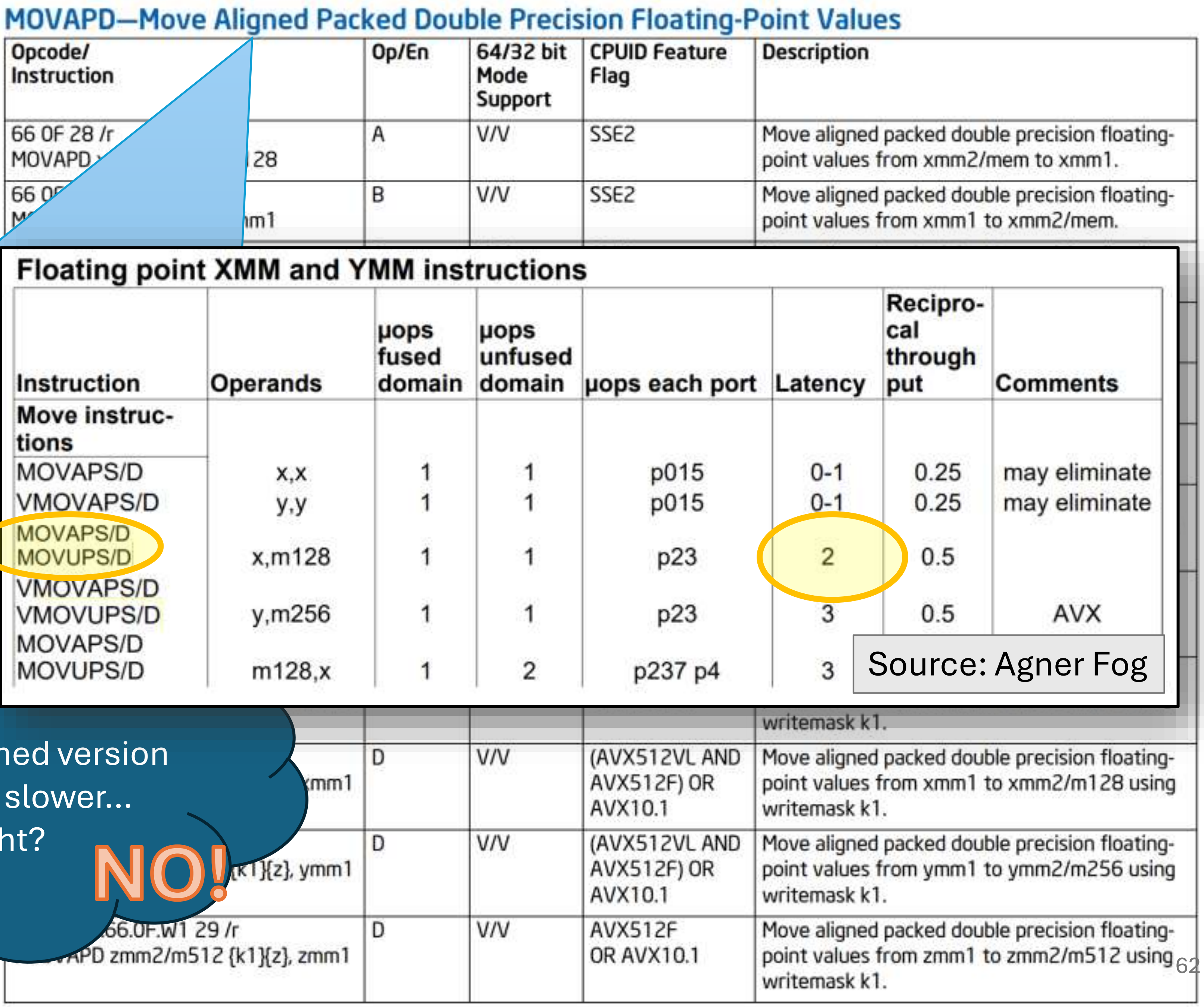
Multiple

Data

Intel’s documenta

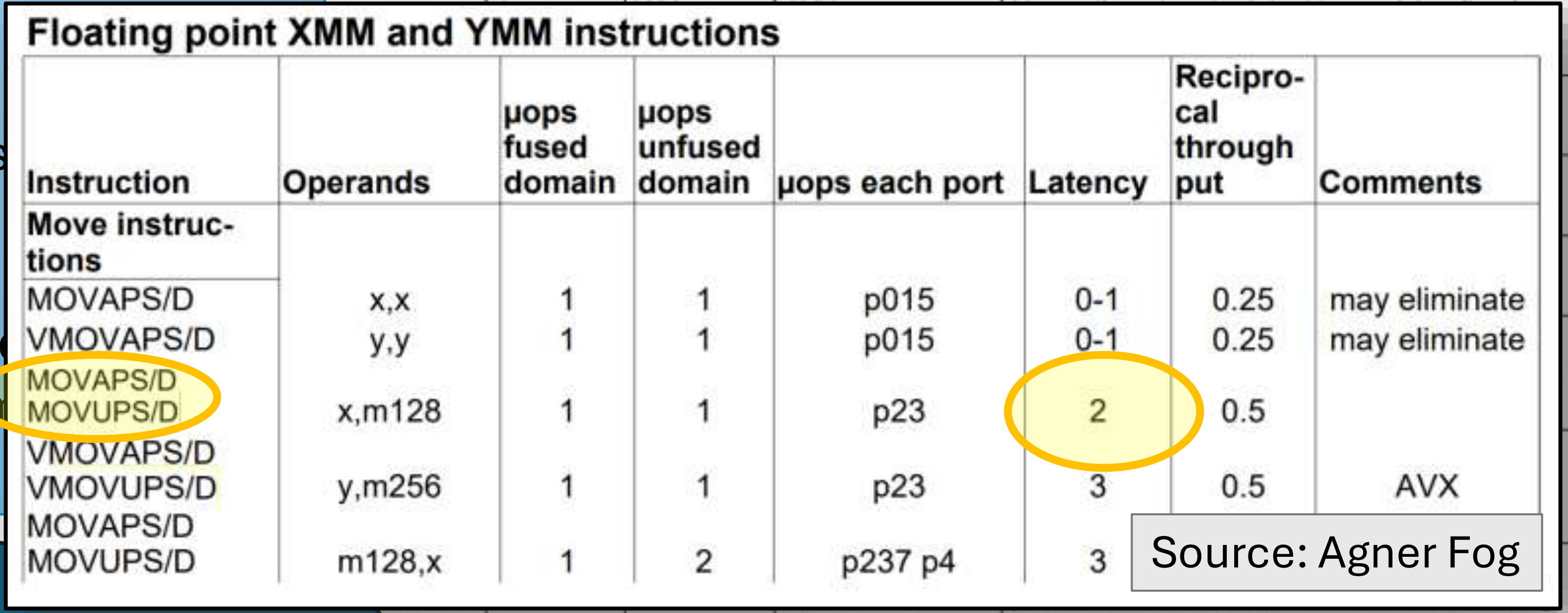
“When the s

“To move
unaligned



MOVAPD—Move Aligned Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 28 /r MOVAPD 28	A	V/V	SSE2	Move aligned packed double precision floating-point values from xmm2/mem to xmm1.
66 0 M m1	B	V/V	SSE2	Move aligned packed double precision floating-point values from xmm1 to xmm2/mem.
				writemask k1.
xmm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed double precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
k1}{z}, ymm1	D	V/V	(AVX512VL AND AVX512F) OR AVX10.1	Move aligned packed double precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
66.0F.W1 29 /r APD zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F OR AVX10.1	Move aligned packed double precision floating-point values from zmm1 to zmm2/m512 using writemask k1.



Floating point XMM and YMM instructions

Instruction	Operands	μops fused domain	μops unfused domain	μops each port	Latency	Reciprocal through put	Comments
Move instructions							
MOVAPS/D	x,x	1	1	p015	0-1	0.25	may eliminate
VMOVAPS/D	y,y	1	1	p015	0-1	0.25	may eliminate
MOVAPS/D MOVUPS/D	x,m128	1	1	p23	2	0.5	
VMOVAPS/D VMOVUPS/D	y,m256	1	1	p23	3	0.5	AVX
MOVAPS/D MOVUPS/D	m128,x	1	2	p237 p4	3		

Source: Agner Fog

The unaligned version must be slower… right?

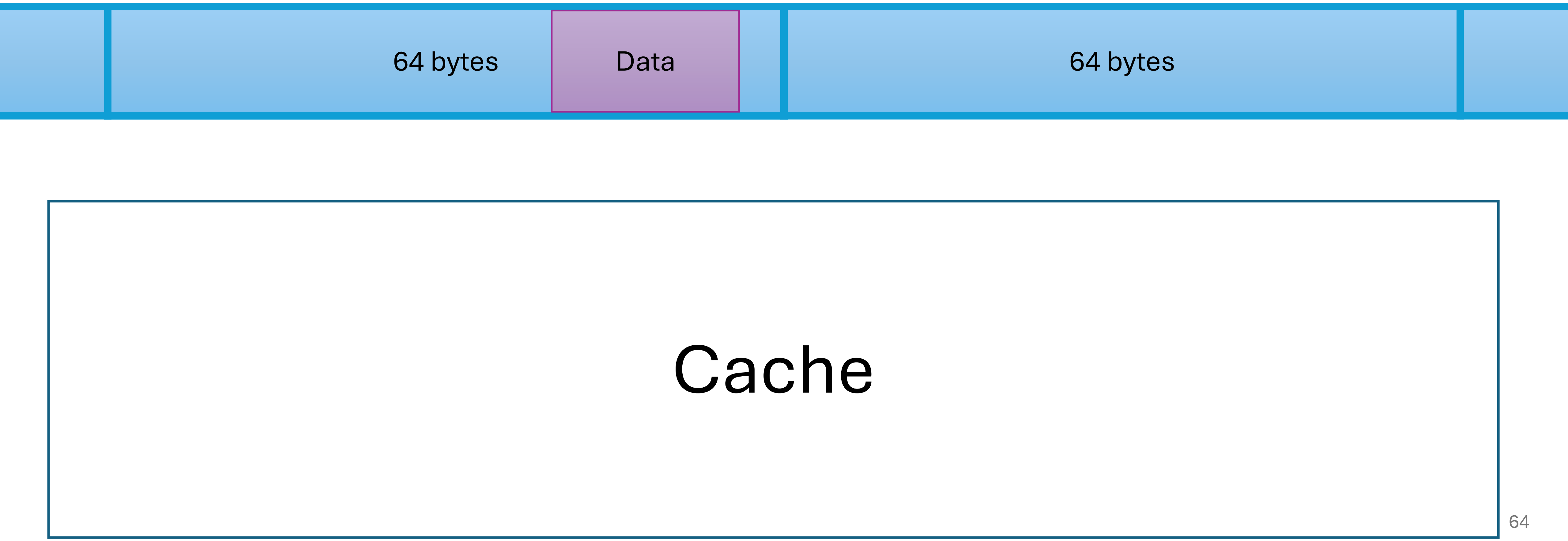
NO!

Alignment is Still Relevant!

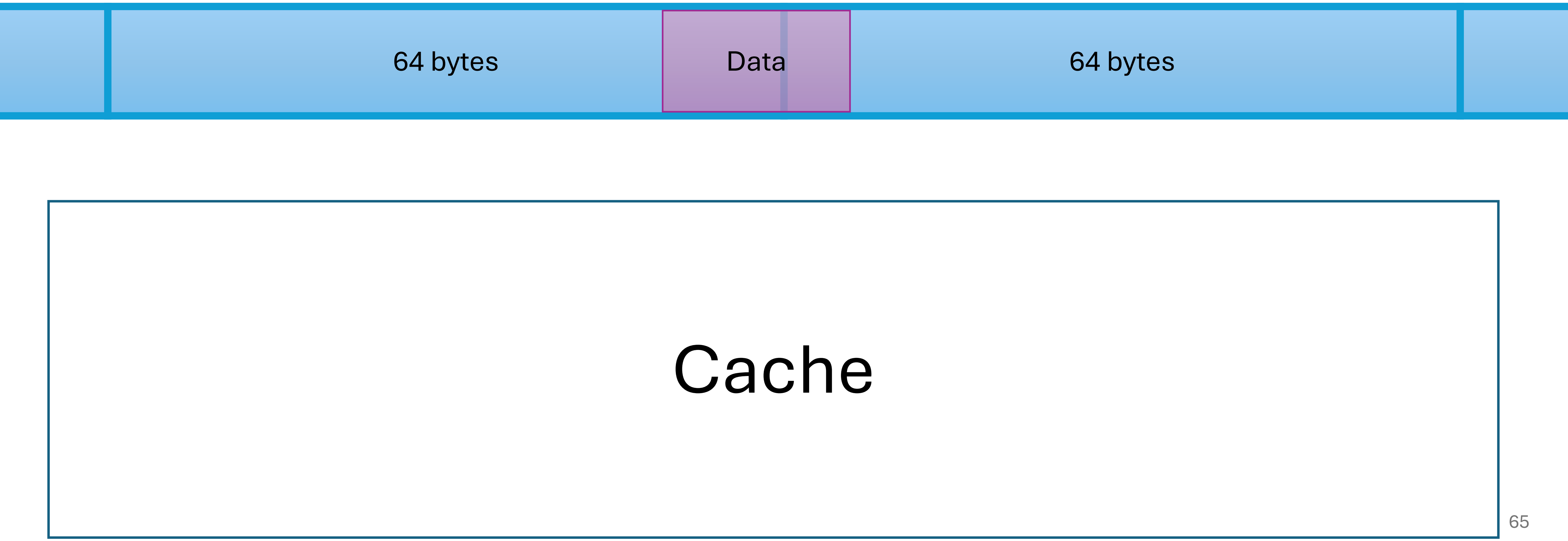
(Even on Modern Platforms)

Photo by Matteo Vistocco on Unsplash

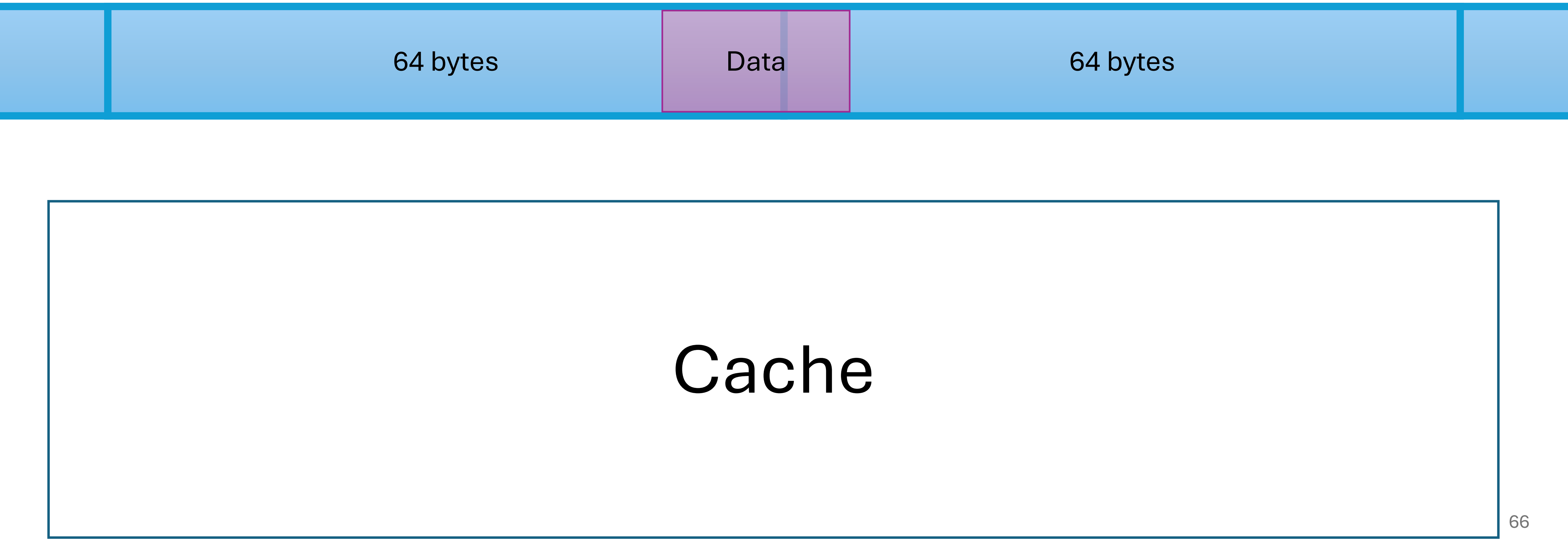
Cache Lines



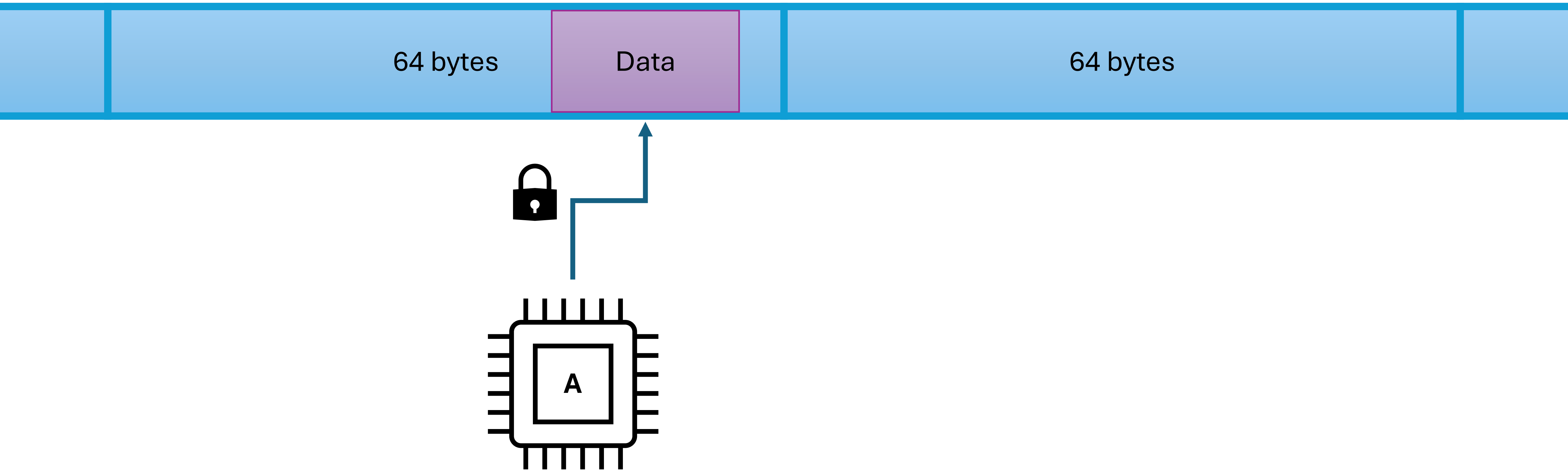
Cache Lines



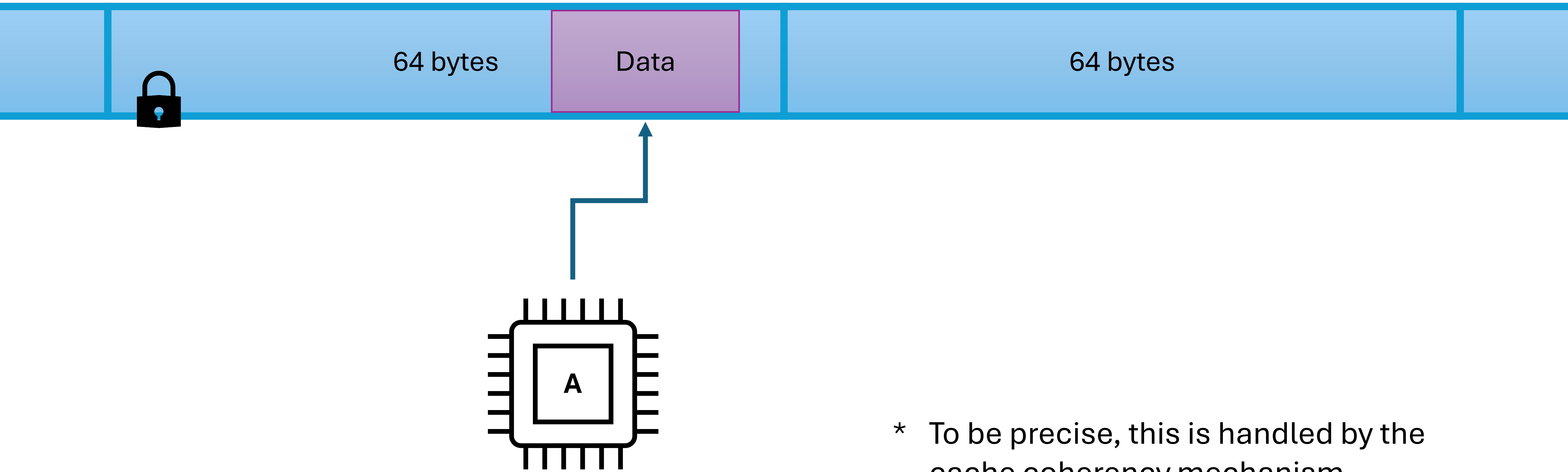
Cache Lines



Cache Lines & Locking

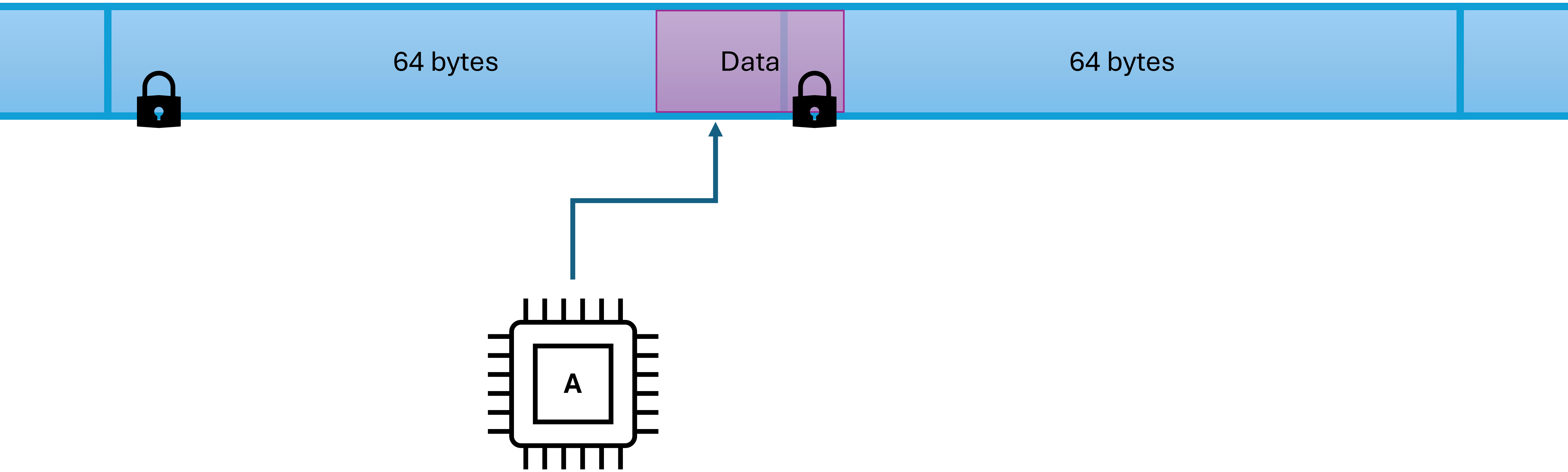


Cache Lines & Locking



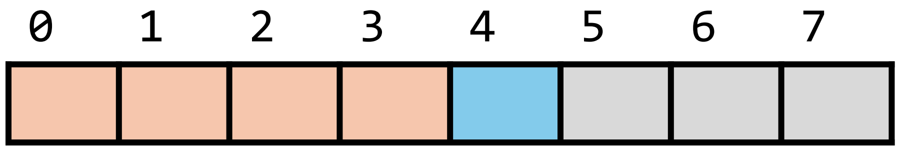
* To be precise, this is handled by the cache coherency mechanism.

Cache Lines & Locking

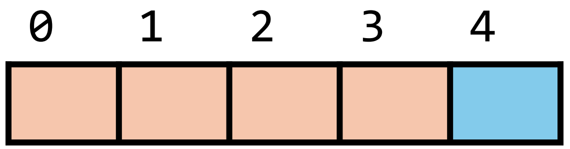


Benchmark

```
struct StructAligned
{
    int a = 42;
    char b = '\0';
};
```

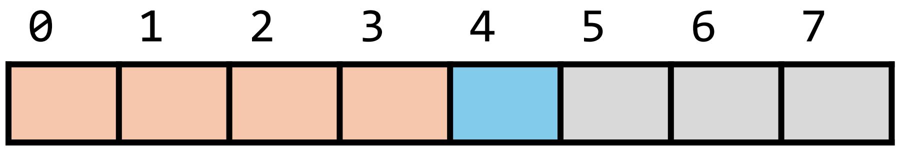


```
#pragma pack(push, 1)
struct StructUnaligned
{
    int a = 42;
    char b = '\0';
};
#pragma pack(pop)
```

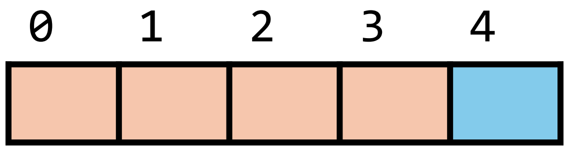


Benchmark

```
struct AtomicAligned
{
    atomic<int> a = 42;
    char b = '\0';
};
```



```
#pragma pack(push, 1)
struct AtomicUnaligned
{
    atomic<int> a = 42;
    char b = '\0';
};
#pragma pack(pop)
```



Benchmark

```
template <typename T>
static void Runner(State& state)
{
    constexpr size_t N = 100;
    T s[N];
    for (auto _ : state) {
        for (int i = 0; i < N; ++i) {
            int t = ++s[i].a;
            DoNotOptimize(t);
        }
    }
}
```

```
BENCHMARK(Runner<StructAligned>);
BENCHMARK(Runner<StructUnaligned>);
BENCHMARK(Runner<AtomicAligned>);
BENCHMARK(Runner<AtomicUnaligned>);
```

Benchmark

```
-------------------------------------------------------------
Benchmark                                    Time          CPU
-------------------------------------------------------------
Runner<StructAligned>                     39.8 ns      39.7 ns
```

Benchmark

```
-------------------------------------------------------------
Benchmark                                 Time            CPU
-------------------------------------------------------------
Runner<StructAligned>                  39.8 ns
Runner<StructUnaligned>                70.8 ns        70.6 ns
```

Cache line split: **78%** slower

Benchmark

```
-------------------------------------------------------
Benchmark                             Time          CPU
-------------------------------------------------------
Runner<StructAligned>              39.8 ns
Runner<StructUnaligned>            70.8 ns      70.6 ns
Runner<AtomicAligned>               669 ns
```

Cache line split: **78%** slower

Atomic write: **9.5x** slower

Benchmark

```
------------------------------------------------------------
Benchmark                                 Time             CPU
------------------------------------------------------------
Runner<StructAligned>                  39.8 ns
Runner<StructUnaligned>                70.8 ns         70.6 ns
Runner<AtomicAligned>                   669 ns
Runner<AtomicUnaligned>             3443049 ns      3434070 ns
```

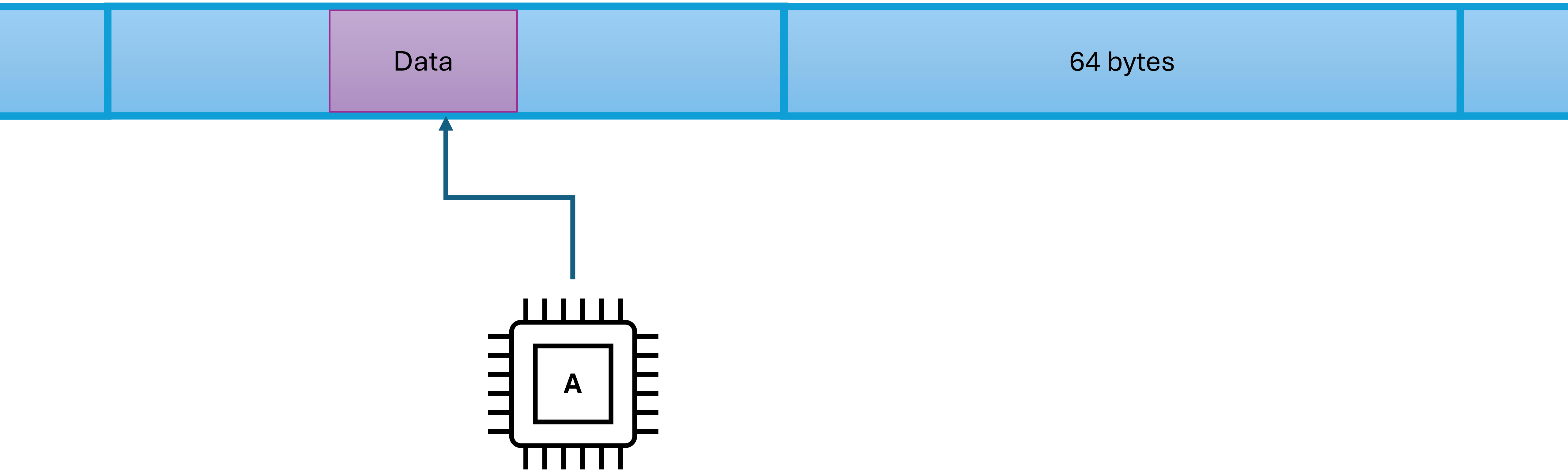
Cache line split: **78%** slower

Atomic write: **9.5x** slower

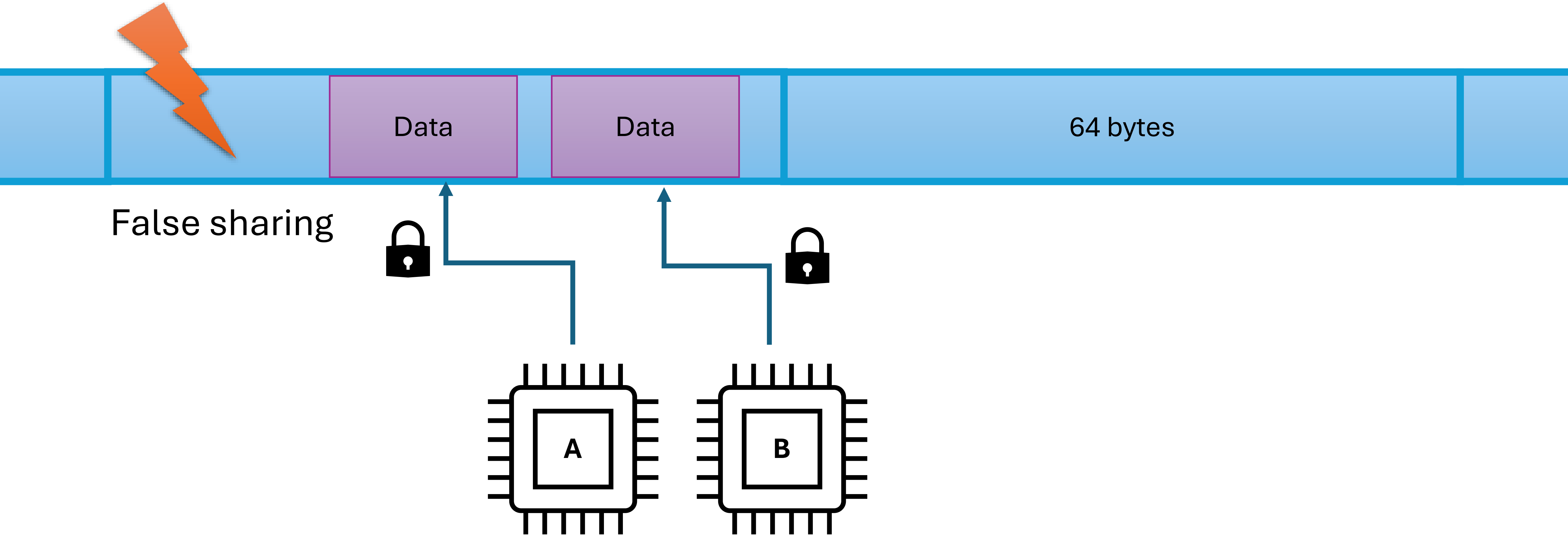
Atomic write with cache line split: **5000x** slower!

Split lock: locks the whole memory bus!

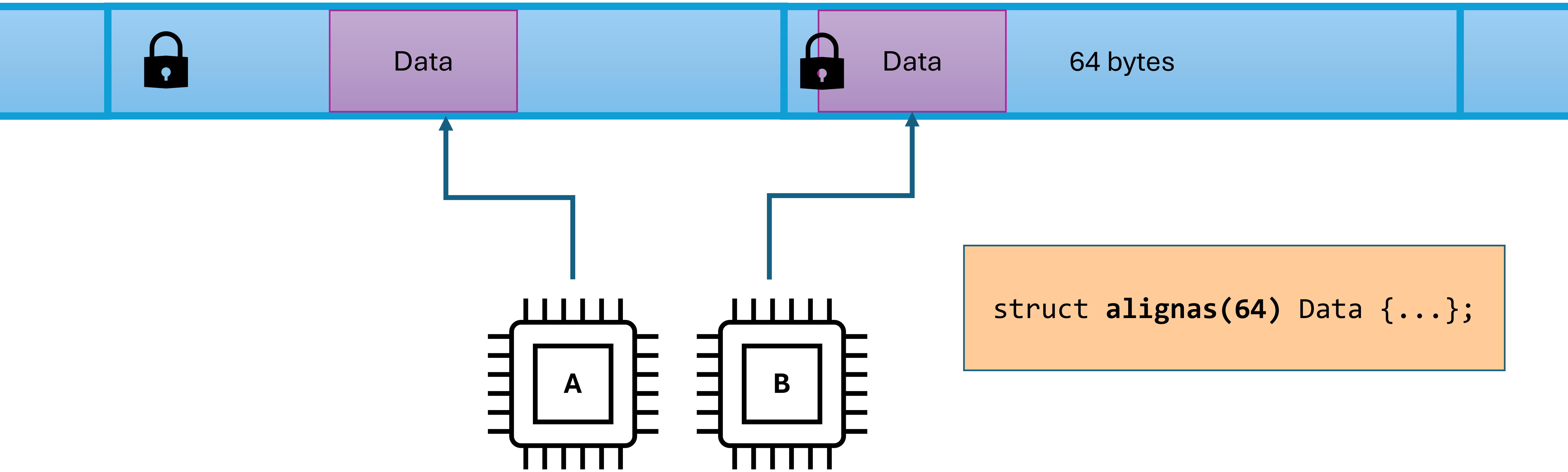
Cache Lines & Locking & Multithread



Cache Lines & Locking & Multithread

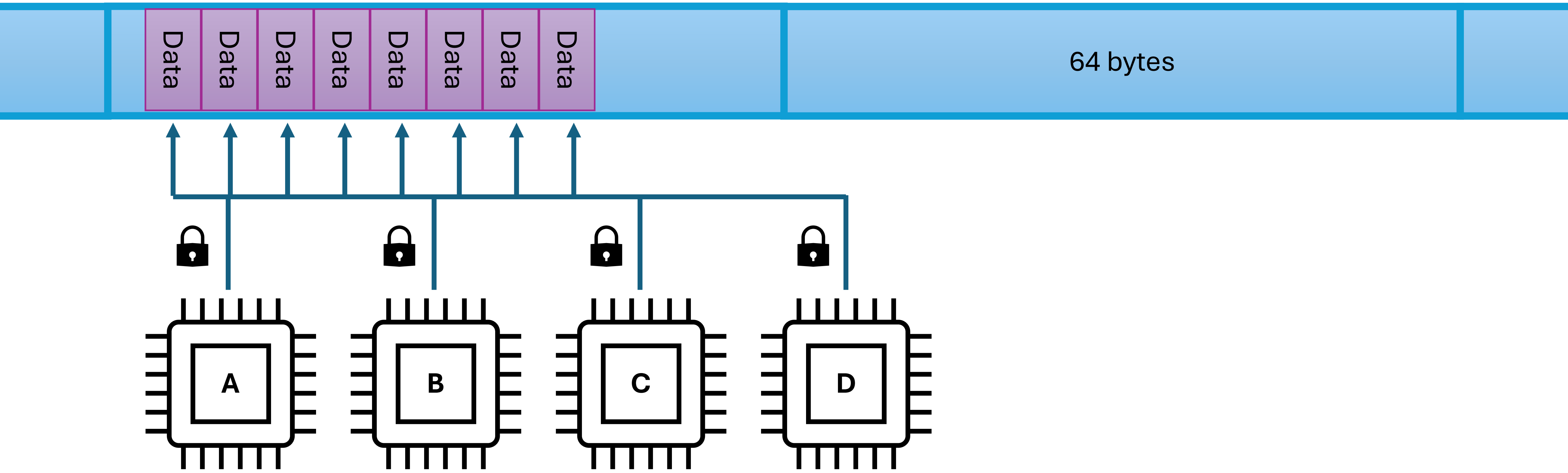


Cache Lines & Locking & Multithread



```
struct alignas(64) Data {...};
```

Benchmark: False Sharing



Benchmark: False Sharing

```
struct AtomicAligned4
{
    atomic<int> a = 42;
};

sizeof(Aligned4) == 4
```

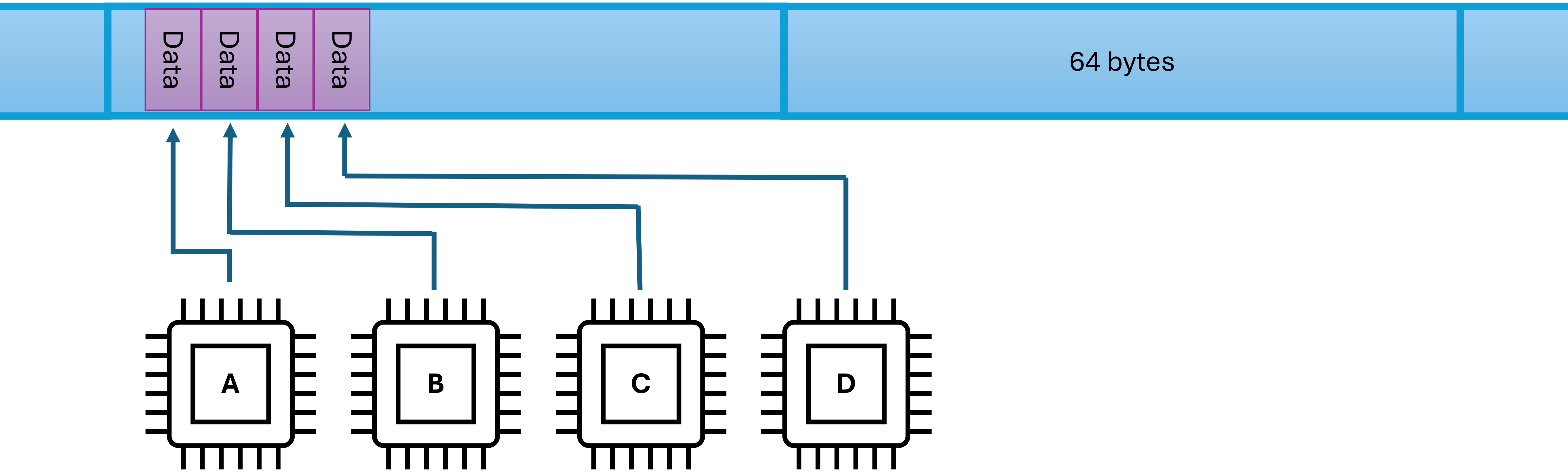
```
alignas(64)
struct AtomicAligned64
{
    atomic<int> a = 42;
};

sizeof(Aligned64) == 64
```

Benchmark: False Sharing

```
--------------------------------------------------
Benchmark                       Time            CPU
--------------------------------------------------
Runner<AtomicAligned4>   1208372885 ns    260510 ns
Runner<AtomicAligned64>   802320730 ns    221603 ns
```

Avoiding false sharing: **33.6% faster**

Benchmark: False Sharing, No Locks



Benchmark: False Sharing, No Locks

```
struct Aligned4
{
    int a = 42;
};

sizeof(Aligned4) == 4
```

```
alignas(64)
struct Aligned64
{
    int a = 42;
};

sizeof(Aligned64) == 64
```

Benchmark: False Sharing, No Locks

```
-----------------------------------------------
Benchmark                          Time              CPU
-----------------------------------------------
Runner<Aligned4>             726761 ns         76867 ns
Runner<Aligned64>            634758 ns         73379 ns
```

Avoiding false sharing: **12.5% faster**

Summary

Alignment – Yes or No?

- C++ alignment rules are simplistic, and maybe outdated
    - Undefined behavior → Implementation-defined?
- Only *really* needed for embedded
- Modern CPUs don't mind unaligned data *too much*
- C++ will pad structs to enforce alignment
    - Good if you need it, but wasteful otherwise
    - Reorder members to reduce padding
    - Use #pragma pack to decrease alignment, *carefully*
- Cache alignment *does* matter for performance!
- Multi-threaded: use `alignas(64)` to avoid false sharing

Thank you.

Thanks to Amir Kirsh

Tomer.Vromen@dell.com