

```
vpcmpgtd    ymm8, ymm10, ymm6  
vpcmpgtd    ymm10, ymm7, ymm10  
vpand       ymm8, ymm10, ymm8  
vmovdqu     ymm11, ymmword ptr [rsp]  
vpcmpgtd    ymm10, ymm11, ymm3  
vpcmpgtd    ymm11, ymm4, ymm11  
vpand       ymm10, ymm10, ymm11  
vpand       ymm8, ymm8, ymm10  
vmovdqu     ymm11, ymmword ptr [rsp - 128]  
vpcmpgtd    ymm10, ymm11, ymm1  
vpcmpgtd    ymm11, ymm2, ymm11  
vpand       ymm10, ymm10, ymm11  
vpand       ymm8, ymm8, ymm10  
vpor        ymm13, ymm8, ymm13  
vmovdqu     ymm10, ymmword ptr [rsp + 96]  
vpcmpgtd    ymm8, ymm10, ymm6  
vpcmpgtd    ymm10, ymm7, ymm10  
vpand       ymm8, ymm10, ymm8  
vmovdqu     ymm11, ymmword ptr [rsp - 32]  
vpcmpgtd    ymm10, ymm11, ymm3  
vpcmpgtd    ymm11, ymm4, ymm11
```

5-3: Arch intrinsics and inline assembly (Theory)

Instruction Set Architecture (ISA)

- “Language” in which CPUs talk
- Most prominent examples: x86, Arm64, RISC-V
- ISA users require strong backward compatibility

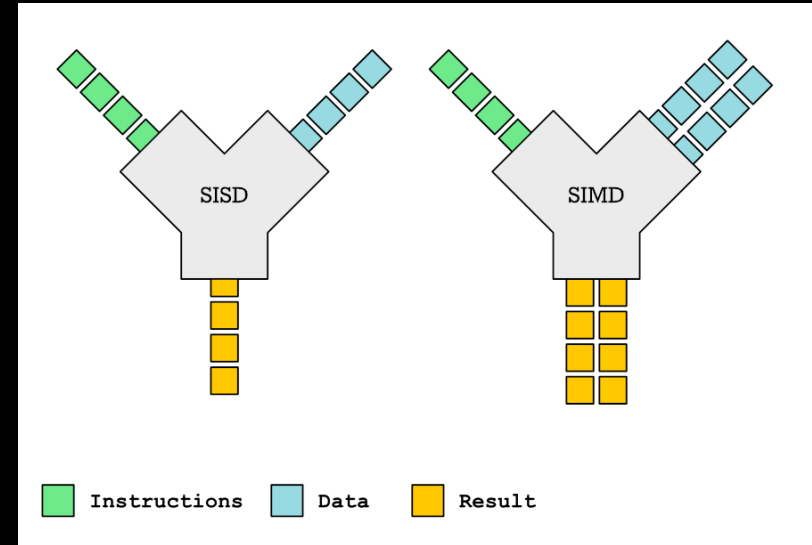
```
1  example::f:
2      test    rsi, rsi
3      je      .LBB0_1
4      mov     ecx, esi
5      and     ecx, 7
6      cmp     rsi, 8
7      jae     .LBB0_4
8      xor     eax, eax
9      xor     edx, edx
10     jmp     .LBB0_6
11  .LBB0_1:
12     xor     eax, eax
13     ret
14  .LBB0_4:
15     and     rsi, -8
16     xor     eax, eax
17     xor     edx, edx
```

ISA extensions

- Instructions sets which extend the “base” set
- x86 examples:
 - SIMD: SSE, SSE2, SSE3, SSSE3, SSE4, AVX, AVX2, AVX-512
 - Cryptography: AES-NI, SHA-NI, CLMUL, RDRAND
 - Bit manipulation: BMI, BMI2
- By default Rust uses only SSE and SSE2

SIMD: Single Instruction, Multiple Data

- Instructions which provide data-level parallelism
- x86 CPUs use SIMD instructions with fixed vector sizes (128, 256, and 512 bits)
- ARM and RISC-V provide “vector” extensions



Problem: using ISA extensions

- Most user CPUs have “modern” extensions (e.g. **AES-NI** and **AVX2**)
- Extensions can make program much faster and even more secure
- But using an unavailable extension will cause CPU exception in the best case, and Undefined Behaviour in the worst

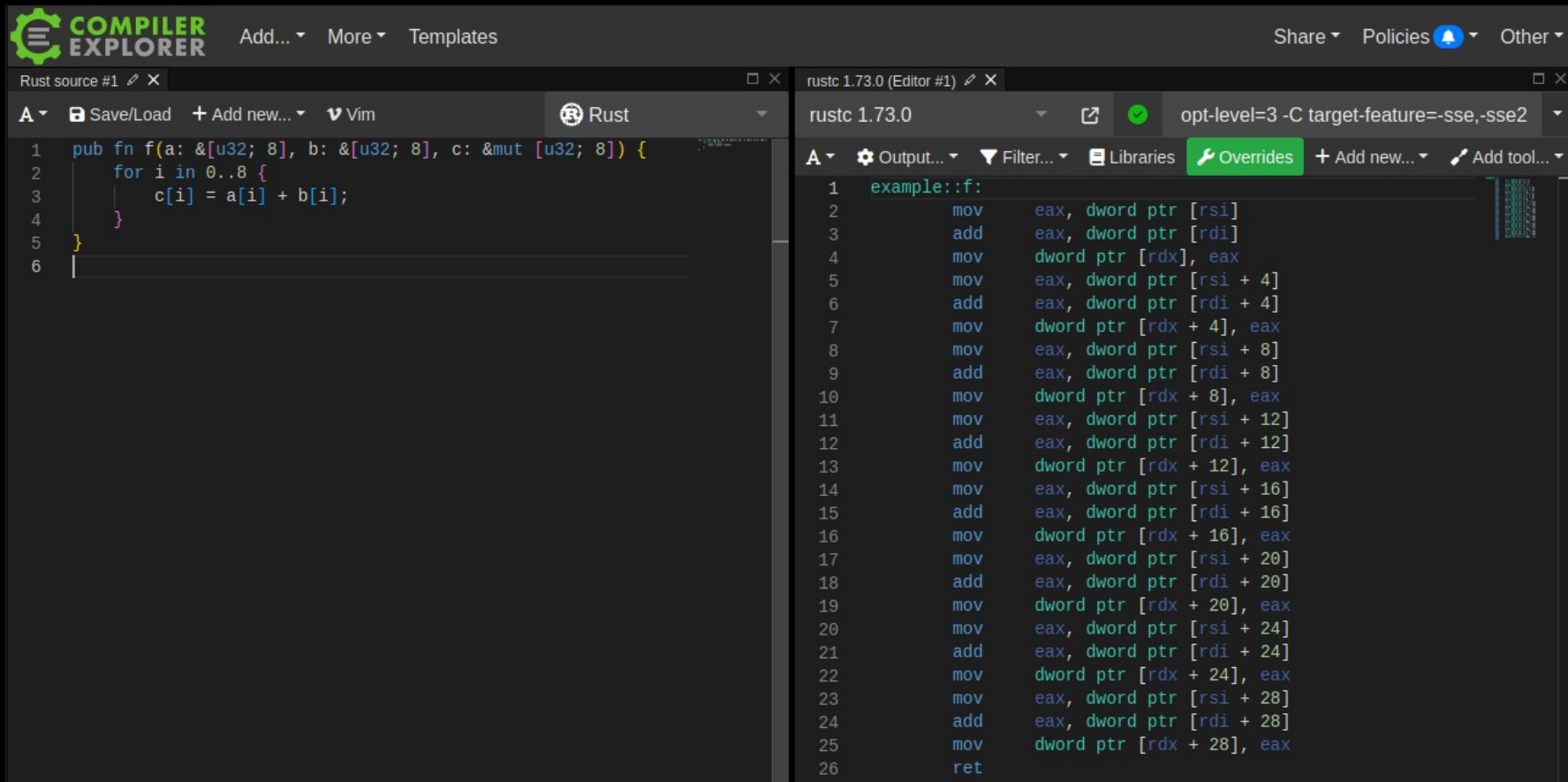
Using CPU extensions in Rust

- Use of ISA extension has to be explicitly enabled
- It's done using target features
- Target feature can be enabled for a whole program or for a separate function
- In the latter case, we can call the function only after we have checked at runtime that CPU supports the necessary target feature

Enabling target feature for a whole program

- Selected target features can be enabled with `-C target-feature=+aes,+avx2` compiler flag
- You can enable all target features supported by your CPU using `-C target-cpu=native` compiler flag
- Warning: the latter may result in a worse performance in some cases. Do not forget to benchmark!

Example: scalar code



The image shows the Compiler Explorer interface with the Rust source code on the left and the generated assembly on the right.

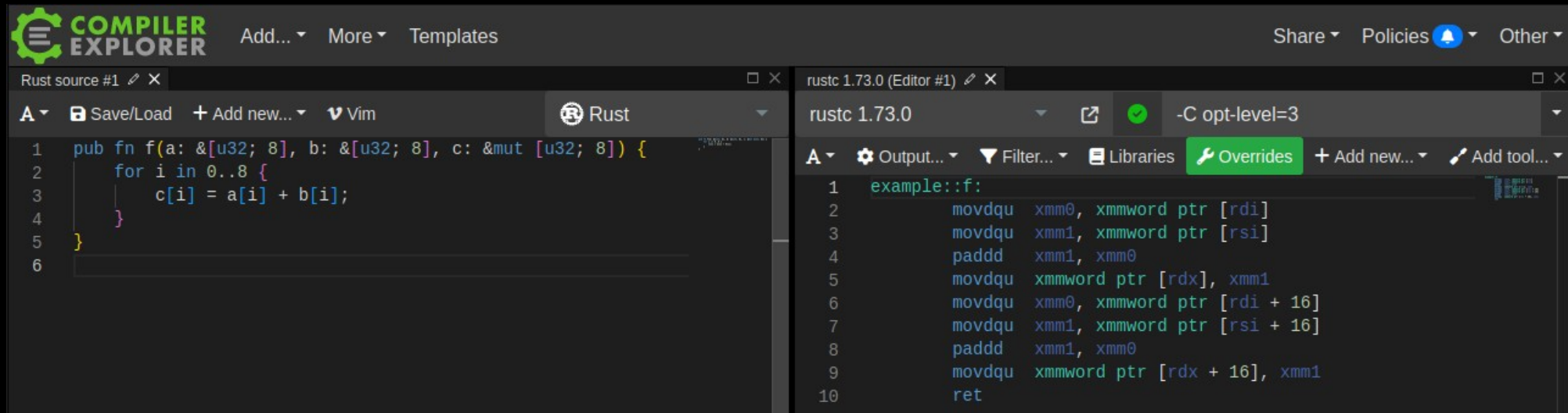
Rust source code (Rust source #1):

```
1 pub fn f(a: &[u32; 8], b: &[u32; 8], c: &mut [u32; 8]) {  
2     for i in 0..8 {  
3         c[i] = a[i] + b[i];  
4     }  
5 }  
6
```

Assembly output (rustc 1.73.0):

```
1 example::f:  
2     mov     eax, dword ptr [rsi]  
3     add     eax, dword ptr [rdi]  
4     mov     dword ptr [rdx], eax  
5     mov     eax, dword ptr [rsi + 4]  
6     add     eax, dword ptr [rdi + 4]  
7     mov     dword ptr [rdx + 4], eax  
8     mov     eax, dword ptr [rsi + 8]  
9     add     eax, dword ptr [rdi + 8]  
10    mov     dword ptr [rdx + 8], eax  
11    mov     eax, dword ptr [rsi + 12]  
12    add     eax, dword ptr [rdi + 12]  
13    mov     dword ptr [rdx + 12], eax  
14    mov     eax, dword ptr [rsi + 16]  
15    add     eax, dword ptr [rdi + 16]  
16    mov     dword ptr [rdx + 16], eax  
17    mov     eax, dword ptr [rsi + 20]  
18    add     eax, dword ptr [rdi + 20]  
19    mov     dword ptr [rdx + 20], eax  
20    mov     eax, dword ptr [rsi + 24]  
21    add     eax, dword ptr [rdi + 24]  
22    mov     dword ptr [rdx + 24], eax  
23    mov     eax, dword ptr [rsi + 28]  
24    add     eax, dword ptr [rdi + 28]  
25    mov     dword ptr [rdx + 28], eax  
26    ret
```


Example: autovectorization (SSE2)

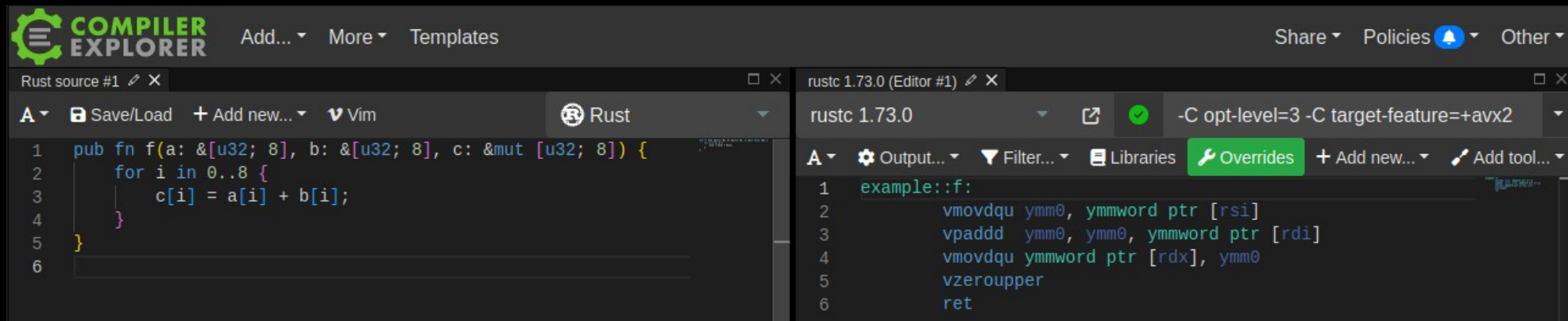


The image shows the Compiler Explorer interface. The left pane displays the Rust source code for a function `f` that iterates over an array `c` and adds corresponding elements from arrays `a` and `b`. The right pane shows the generated assembly code for the `rustc 1.73.0` compiler, which includes SSE2 instructions like `movdqu` and `padd` to perform the vectorized addition.

```
Rust source #1
1 pub fn f(a: &[u32; 8], b: &[u32; 8], c: &mut [u32; 8]) {
2     for i in 0..8 {
3         c[i] = a[i] + b[i];
4     }
5 }
6
```

```
rustc 1.73.0 (Editor #1)
rustc 1.73.0 -C opt-level=3
example::f:
2     movdqu    xmm0, xmmword ptr [rdi]
3     movdqu    xmm1, xmmword ptr [rsi]
4     padd     xmm1, xmm0
5     movdqu    xmmword ptr [rdx], xmm1
6     movdqu    xmm0, xmmword ptr [rdi + 16]
7     movdqu    xmm1, xmmword ptr [rsi + 16]
8     padd     xmm1, xmm0
9     movdqu    xmmword ptr [rdx + 16], xmm1
10    ret
```

Example: autovectorization (AVX2)

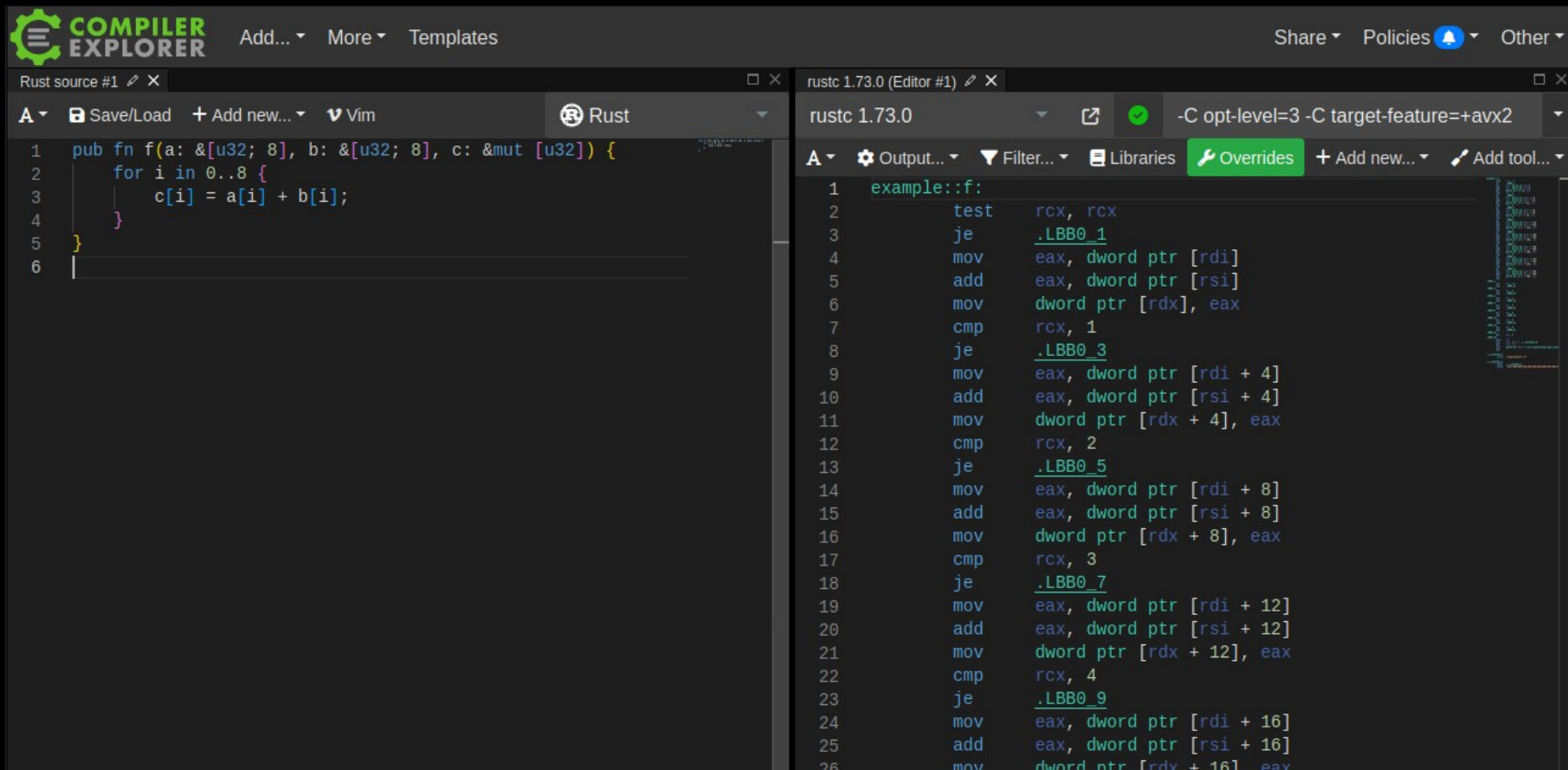


The screenshot displays the Compiler Explorer interface. The left pane shows the Rust source code for a function `f` that iterates over an array of 8 `u32` elements and adds them. The right pane shows the generated assembly for `rustc 1.73.0` with optimization level 3 and the `avx2` target feature enabled. The assembly output shows the use of `ymm` registers and `vpaddd` instructions, indicating successful autovectorization to AVX2.

```
Rust source #1 X
A Save/Load + Add new... Vim Rust
1 pub fn f(a: &[u32; 8], b: &[u32; 8], c: &mut [u32; 8]) {
2     for i in 0..8 {
3         c[i] = a[i] + b[i];
4     }
5 }
6

rustc 1.73.0 (Editor #1) X
rustc 1.73.0 -C opt-level=3 -C target-feature=+avx2
A Output... Filter... Libraries Overrides + Add new... Add tool...
1 example::f:
2     vmovdqu ymm0, ymmword ptr [rsi]
3     vpaddd ymm0, ymm0, ymmword ptr [rdi]
4     vmovdqu ymmword ptr [rdx], ymm0
5     vzeroupper
6     ret
```

Autovectorization fragility

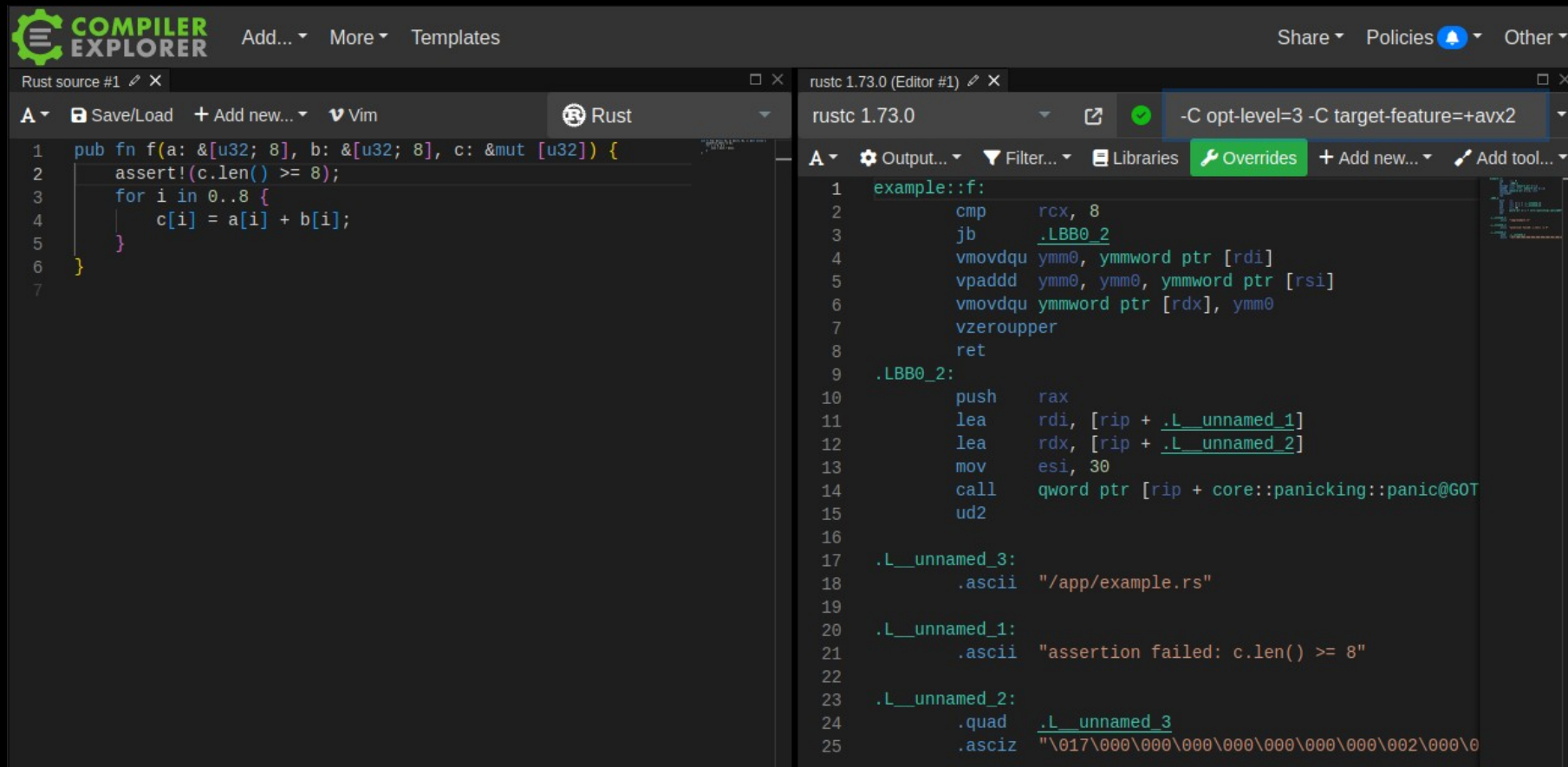


The screenshot displays the Visual Studio Code interface with two panels. The left panel shows the Rust source code for a function `f` that iterates over an array `c` and updates its elements based on `a` and `b`. The right panel shows the assembly output for the same function, demonstrating how the compiler has autovectorized the loop into SIMD instructions using SSE registers.

```
1 pub fn f(a: &[u32; 8], b: &[u32; 8], c: &mut [u32]) {  
2     for i in 0..8 {  
3         c[i] = a[i] + b[i];  
4     }  
5 }  
6
```

```
1 example::f:  
2     test    rcx, rcx  
3     je      .LBB0_1  
4     mov     eax, dword ptr [rdi]  
5     add     eax, dword ptr [rsi]  
6     mov     dword ptr [rdx], eax  
7     cmp     rcx, 1  
8     je      .LBB0_3  
9     mov     eax, dword ptr [rdi + 4]  
10    add     eax, dword ptr [rsi + 4]  
11    mov     dword ptr [rdx + 4], eax  
12    cmp     rcx, 2  
13    je      .LBB0_5  
14    mov     eax, dword ptr [rdi + 8]  
15    add     eax, dword ptr [rsi + 8]  
16    mov     dword ptr [rdx + 8], eax  
17    cmp     rcx, 3  
18    je      .LBB0_7  
19    mov     eax, dword ptr [rdi + 12]  
20    add     eax, dword ptr [rsi + 12]  
21    mov     dword ptr [rdx + 12], eax  
22    cmp     rcx, 4  
23    je      .LBB0_9  
24    mov     eax, dword ptr [rdi + 16]  
25    add     eax, dword ptr [rsi + 16]  
26    mov     dword ptr [rdx + 16], eax
```

“Fixing” autovectorization



The screenshot displays the Visual Studio Code interface with two panels. The left panel shows a Rust source file named 'Rust source #1' with the following code:

```
1 pub fn f(a: &[u32; 8], b: &[u32; 8], c: &mut [u32]) {  
2     assert!(c.len() >= 8);  
3     for i in 0..8 {  
4         c[i] = a[i] + b[i];  
5     }  
6 }  
7
```

The right panel shows the 'rustc 1.73.0 (Editor #1)' output window. The command line at the top is `rustc 1.73.0 -C opt-level=3 -C target-feature=+avx2`. The assembly output for the function `example::f` is shown below:

```
1 example::f:  
2     cmp     rcx, 8  
3     jb     .LBB0_2  
4     vmovdqu ymm0, ymmword ptr [rdi]  
5     vpaddd  ymm0, ymm0, ymmword ptr [rsi]  
6     vmovdqu ymmword ptr [rdx], ymm0  
7     vzeroupper  
8     ret  
9 .LBB0_2:  
10    push    rax  
11    lea     rdi, [rip + .L__unnamed_1]  
12    lea     rdx, [rip + .L__unnamed_2]  
13    mov     esi, 30  
14    call    qword ptr [rip + core::panicking::panic@GOT  
15    ud2  
16  
17 .L__unnamed_3:  
18     .ascii  "/app/example.rs"  
19  
20 .L__unnamed_1:  
21     .ascii  "assertion failed: c.len() >= 8"  
22  
23 .L__unnamed_2:  
24     .quad   .L__unnamed_3  
25     .asciz  "\017\000\000\000\000\000\000\000\002\000\0"
```

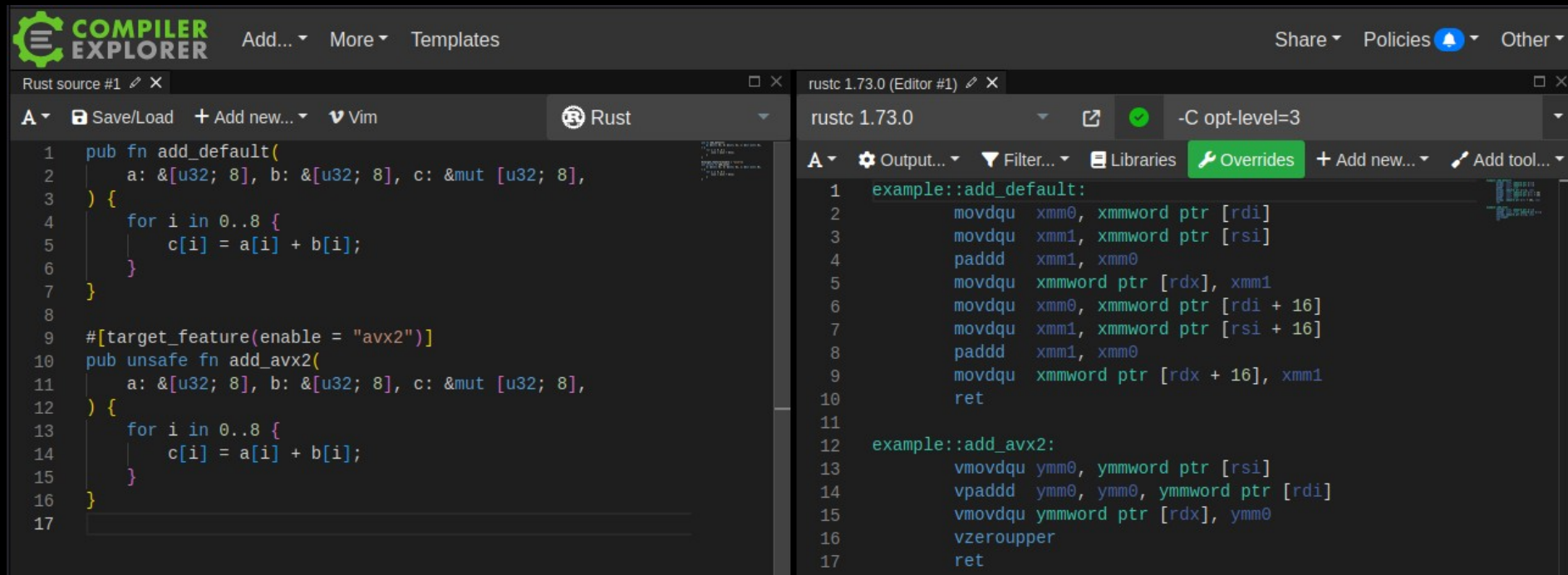
Target feature detection at runtime

- Target features can be detected at runtime using `std::is_x86_feature_detected!` Macro
- The macro is available (for now) only on `std` and `x86` targets
- An alternative: the `cpufeatures` crate

Selectively enabling target features

- Target feature can be enabled for a function with `#[target_feature(enable = "...")]` attribute
- Such function has to be `unsafe`, since to call it we have to check availability of the required ISA extensions
- Inside the function compiler can use instruction from enabled extensions
- RFC: <https://rust-lang.github.io/rfcs/2045-target-feature.html>

Example: `#[target_feature(enable = "...")]`



The image shows the Compiler Explorer interface. The left pane displays Rust source code for a function `add_default` and a feature-gated version `add_avx2`. The right pane shows the assembly output for the `add_avx2` function, which uses AVX instructions like `movdqu`, `paddq`, and `vzeroupper`.

```
1 pub fn add_default(  
2     a: &[u32; 8], b: &[u32; 8], c: &mut [u32; 8],  
3 ) {  
4     for i in 0..8 {  
5         c[i] = a[i] + b[i];  
6     }  
7 }  
8  
9 #[target_feature(enable = "avx2")]  
10 pub unsafe fn add_avx2(  
11     a: &[u32; 8], b: &[u32; 8], c: &mut [u32; 8],  
12 ) {  
13     for i in 0..8 {  
14         c[i] = a[i] + b[i];  
15     }  
16 }  
17
```

```
1 example::add_default:  
2     movdqu    xmm0, xmmword ptr [rdi]  
3     movdqu    xmm1, xmmword ptr [rsi]  
4     paddq     xmm1, xmm0  
5     movdqu    xmmword ptr [rdx], xmm1  
6     movdqu    xmm0, xmmword ptr [rdi + 16]  
7     movdqu    xmm1, xmmword ptr [rsi + 16]  
8     paddq     xmm1, xmm0  
9     movdqu    xmmword ptr [rdx + 16], xmm1  
10    ret  
11  
12 example::add_avx2:  
13    vmovdqu    ymm0, ymmword ptr [rsi]  
14    vpaddq     ymm0, ymm0, ymmword ptr [rdi]  
15    vmovdqu    ymmword ptr [rdx], ymm0  
16    vzeroupper  
17    ret
```

Example: autodetection 1

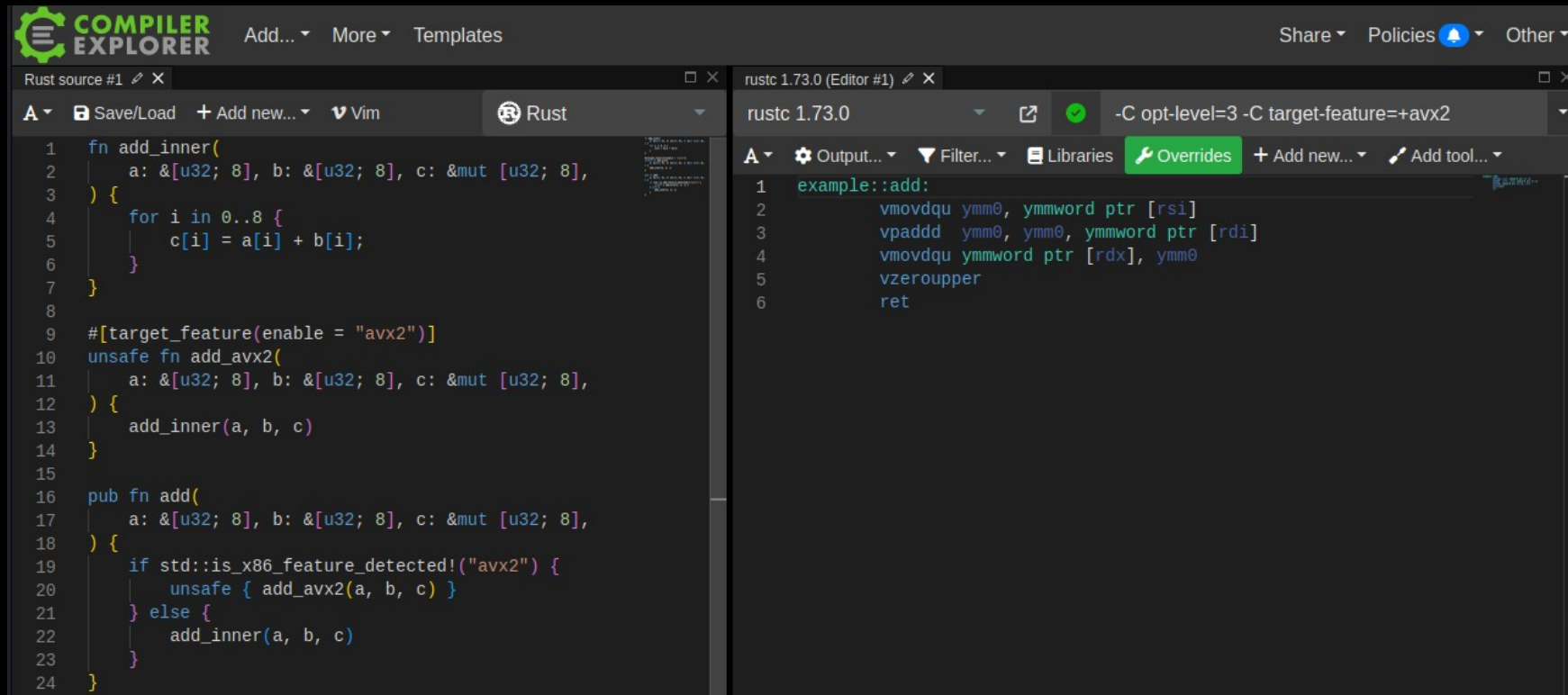
```
1  fn add_inner(  
2      a: &[u32; 8], b: &[u32; 8], c: &mut [u32; 8],  
3  ) {  
4      for i in 0..8 {  
5          c[i] = a[i] + b[i];  
6      }  
7  }  
8  
9  #[target_feature(enable = "avx2")]  
10 unsafe fn add_avx2(  
11     a: &[u32; 8], b: &[u32; 8], c: &mut [u32; 8],  
12 ) {  
13     add_inner(a, b, c)  
14 }  
15  
16 pub fn add(  
17     a: &[u32; 8], b: &[u32; 8], c: &mut [u32; 8],  
18 ) {  
19     if std::is_x86_feature_detected!("avx2") {  
20         unsafe { add_avx2(a, b, c) }  
21     } else {  
22         add_inner(a, b, c)  
23     }  
24 }
```


Example: autodetection 2

```
1  example::add_avx2:
2      vmovdqu ymm0, ymmword ptr [rsi]
3      vpaddd  ymm0, ymm0, ymmword ptr [rdi]
4      vmovdqu ymmword ptr [rdx], ymm0
5      vzeroupper
6      ret
7
8  example::add:
9      push    r15
10     push    r14
11     push    rbx
12     mov     rbx, rdx
13     mov     r14, rsi
14     mov     r15, rdi
15     mov     rax, qword ptr [rip + std_detect::det
16     mov     rax, qword ptr [rax]
17     test    rax, rax
18     je      .LBB1_1
19     test    ax, ax
20     js      .LBB1_4
21 .LBB1_3:
```

```
21 .LBB1_3:
22     movdqu  xmm0, xmmword ptr [r15]
23     movdqu  xmm1, xmmword ptr [r14]
24     paddd   xmm1, xmm0
25     movdqu  xmmword ptr [rbx], xmm1
26     movdqu  xmm0, xmmword ptr [r15 + 16]
27     movdqu  xmm1, xmmword ptr [r14 + 16]
28     paddd   xmm1, xmm0
29     movdqu  xmmword ptr [rbx + 16], xmm1
30     pop     rbx
31     pop     r14
32     pop     r15
33     ret
34 .LBB1_1:
35     call    qword ptr [rip + std_detect::det
36     test    ax, ax
37     jns     .LBB1_3
38 .LBB1_4:
39     mov     rdi, r15
40     mov     rsi, r14
41     mov     rdx, rbx
42     pop     rbx
43     pop     r14
44     pop     r15
45     jmp     example::add_avx2
```

Example: autodetection 3



The image shows the Compiler Explorer interface. The left pane displays Rust source code for a function `add_inner` and a public function `add` that uses `std::is_x86_feature_detected!` to conditionally enable AVX2. The right pane shows the generated assembly for the `example::add` function, which uses AVX2 instructions like `vmovdqu`, `vpaddq`, and `vzeroupper`.

```
Rust source #1 X
A Save/Load + Add new... Vim Rust
1 fn add_inner(
2     a: &[u32; 8], b: &[u32; 8], c: &mut [u32; 8],
3 ) {
4     for i in 0..8 {
5         c[i] = a[i] + b[i];
6     }
7 }
8
9 #[target_feature(enable = "avx2")]
10 unsafe fn add_avx2(
11     a: &[u32; 8], b: &[u32; 8], c: &mut [u32; 8],
12 ) {
13     add_inner(a, b, c)
14 }
15
16 pub fn add(
17     a: &[u32; 8], b: &[u32; 8], c: &mut [u32; 8],
18 ) {
19     if std::is_x86_feature_detected!("avx2") {
20         unsafe { add_avx2(a, b, c) }
21     } else {
22         add_inner(a, b, c)
23     }
24 }
```

```
rustc 1.73.0 (Editor #1) X
rustc 1.73.0 -C opt-level=3 -C target-feature=+avx2
A Output... Filter... Libraries Overrides + Add new... Add tool...
1 example::add:
2     vmovdqu ymm0, ymmword ptr [rsi]
3     vpaddq ymm0, ymm0, ymmword ptr [rdi]
4     vmovdqu ymmword ptr [rdx], ymm0
5     vzeroupper
6     ret
```

arch intrinsics

- Special low-level arch-specific **unsafe** functions which compile down to one instruction*
- For example, **_mm256_add_epi32** usually compiles to **vpaddq**
- Defined in the **std::arch** module
- Callers must ensure that CPU has the required target feature

std::arch::x86_64



Module x86_64

Structs

Constants

Functions

Type Aliases

`_fxrstor`[△] (x86 or x86-64) and `fxsr`

`_fxrstor64`[△] `fxsr`

`_fxsave`[△] (x86 or x86-64) and `fxsr`

`_fxsave64`[△] `fxsr`

`_lzcnt_u32`[△] (x86 or x86-64) and `lzcnt`

`_lzcnt_u64`[△] `lzcnt`

`_mm256_abs_epi8`[△] (x86 or x86-64) and `avx2`

`_mm256_abs_epi16`[△] (x86 or x86-64) and `avx2`

`_mm256_abs_epi32`[△] (x86 or x86-64) and `avx2`

`_mm256_add_epi8`[△] (x86 or x86-64) and `avx2`

`_mm256_add_epi16`[△] (x86 or x86-64) and `avx2`

`_mm256_add_epi32`[△] (x86 or x86-64) and `avx2`

`_mm256_add_epi64`[△] (x86 or x86-64) and `avx2`

`_mm256_add_pd`[△] (x86 or x86-64) and `avx`

`_mm256_add_ps`[△] (x86 or x86-64) and `avx`

`_mm256_adds_epi8`[△] (x86 or x86-64) and `avx2`

`_mm256_adds_epi16`[△] (x86 or x86-64) and `avx2`

`_mm256_adds_epu8`[△] (x86 or x86-64) and `avx2`

`_mm256_adds_epu16`[△] (x86 or x86-64) and `avx2`

Restores the XMM, MMX, MXCSR, and x87 FPU registers from the 512-byte-long 16-byte-aligned memory region `mem_addr`.

Restores the XMM, MMX, MXCSR, and x87 FPU registers from the 512-byte-long 16-byte-aligned memory region `mem_addr`.

Saves the x87 FPU, MMX technology, XMM, and MXCSR registers to the 512-byte-long 16-byte-aligned memory region `mem_addr`.

Saves the x87 FPU, MMX technology, XMM, and MXCSR registers to the 512-byte-long 16-byte-aligned memory region `mem_addr`.

Counts the leading most significant zero bits.

Counts the leading most significant zero bits.

Computes the absolute values of packed 8-bit integers in `a`.

Computes the absolute values of packed 16-bit integers in `a`.

Computes the absolute values of packed 32-bit integers in `a`.

Adds packed 8-bit integers in `a` and `b`.

Adds packed 16-bit integers in `a` and `b`.

Adds packed 32-bit integers in `a` and `b`.

Adds packed 64-bit integers in `a` and `b`.

Adds packed double-precision (64-bit) floating-point elements in `a` and `b`.

Adds packed single-precision (32-bit) floating-point elements in `a` and `b`.

Adds packed 8-bit integers in `a` and `b` using saturation.

Adds packed 16-bit integers in `a` and `b` using saturation.

Adds packed unsigned 8-bit integers in `a` and `b` using saturation.

Adds packed unsigned 16-bit integers in `a` and `b` using

Intel Intrinsics Guide

- The official resource about x86 intrinsics:
<https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html>
- Contains detailed description of intrinsics and additional information (e.g. latency, throughput, etc.)

The screenshot shows the Intel Intrinsics Guide interface. On the left, under 'Instruction Set', there are checkboxes for MMX, SSE family, AVX family, AVX-512 family, AMX family, SVMML, and Other. Below this, under 'Categories', there are checkboxes for Application-Targeted, Arithmetic, Bit Manipulation, and Cast. A search bar labeled 'Search Intel Intrinsics' is located at the top right. The main area displays a list of intrinsics, each with its name, arguments, and a category label on the right.

Intrinsic Name	Arguments	Category
<code>_mm256i_mm256_abs_epi16</code>	<code>(__m256i a)</code>	vpabsw
<code>_mm256i_mm256_abs_epi32</code>	<code>(__m256i a)</code>	vpabsd
<code>_mm256i_mm256_abs_epi8</code>	<code>(__m256i a)</code>	vpabsb
<code>_mm256i_mm256_add_epi16</code>	<code>(__m256i a, __m256i b)</code>	vpaddw
<code>_mm256i_mm256_add_epi32</code>	<code>(__m256i a, __m256i b)</code>	vpaddd
<code>_mm256i_mm256_add_epi64</code>	<code>(__m256i a, __m256i b)</code>	vpaddq
<code>_mm256i_mm256_add_epi8</code>	<code>(__m256i a, __m256i b)</code>	vpaddb
<code>_mm256i_mm256_adds_epi16</code>	<code>(__m256i a, __m256i b)</code>	vpaddsw
<code>_mm256i_mm256_adds_epi8</code>	<code>(__m256i a, __m256i b)</code>	vpaddsb
<code>_mm256i_mm256_adds_epu16</code>	<code>(__m256i a, __m256i b)</code>	vpaddusw

Intrinsics vs `std::arch::asm!`

- You do not manually allocate registers with an intrinsics-based code, the compiler can handle stack spilling if necessary
- The compiler “understands” intrinsics to a certain extent
- The compiler may change order of intrinsic calls
- For example, `_mm256_sub_epi32(x, x)` can be compiled as `vxorps xmm0, xmm0, xmm0`

Vector types

- x86 arch intrinsics work with “vector types”
- For example, `__m128`, `__m128i`, `__m256`, `__m256d`, `__m256i`, etc.
- `__m256i` depending on intrinsic can be interpreted as `u8x32`, `i8x32`, `u16x16`, `i16x16`, `u32x8`, `i32x8`, `u64x4`, `i64x4`
- `__m256` is interpreted as `f32x8`, and `__m256d` as `f64x4`
- Variables of vector types are usually processed in associated SIMD registers: `XMM`, `YMM`, and `ZMM`

Example: intrinsics 1

```
1  #[cfg(target_arch = "x86")]
2  use std::arch::x86::*;
3  #[cfg(target_arch = "x86_64")]
4  use std::arch::x86_64::*;
5
6  pub fn add(
7      a: &[u32; 8], b: &[u32; 8], c: &mut [u32; 8],
8  ) {
9      if is_x86_feature_detected!("avx2") {
10         unsafe { add_avx2(a, b, c) }
11     } else if is_x86_feature_detected!("sse2") {
12         unsafe { add_sse2(a, b, c) }
13     } else {
14         add_soft(a, b, c)
15     }
16 }
17
18 fn add_soft(a: &[u32; 8], b: &[u32; 8], c: &mut [u32; 8]) {
19     for i in 0..8 {
20         c[i] = a[i] + b[i];
21     }
22 }
23
```

```
23
24 #[target_feature(enable = "sse2")]
25 unsafe fn add_sse2(a: &[u32; 8], b: &[u32; 8], c: &mut [u32; 8]) {
26     let a1: __m128i = _mm_loadu_si128(a.as_ptr().cast());
27     let b1: __m128i = _mm_loadu_si128(b.as_ptr().cast());
28     let res1: __m128i = _mm_add_epi32(a1, b1);
29     _mm_storeu_si128(c.as_mut_ptr().cast(), res1);
30
31     let a2: __m128i = _mm_loadu_si128(a.as_ptr().add(4).cast());
32     let b2: __m128i = _mm_loadu_si128(b.as_ptr().add(4).cast());
33     let res2: __m128i = _mm_add_epi32(a2, b2);
34     _mm_storeu_si128(c.as_mut_ptr().add(4).cast(), res2);
35 }
36
37 #[target_feature(enable = "avx2")]
38 unsafe fn add_avx2(a: &[u32; 8], b: &[u32; 8], c: &mut [u32; 8]) {
39     let a: __m256i = _mm256_loadu_si256(a.as_ptr().cast());
40     let b: __m256i = _mm256_loadu_si256(b.as_ptr().cast());
41     let res: __m256i = _mm256_add_epi32(a, b);
42     _mm256_storeu_si256(c.as_mut_ptr().cast(), res);
43 }
44
```


Example: intrinsics 2

```
mov    eax, dword ptr [r14]
add    eax, dword ptr [r15]
mov    dword ptr [rbx], eax
mov    eax, dword ptr [r14 + 4]
add    eax, dword ptr [r15 + 4]
mov    dword ptr [rbx + 4], eax
mov    eax, dword ptr [r14 + 8]
add    eax, dword ptr [r15 + 8]
mov    dword ptr [rbx + 8], eax
mov    eax, dword ptr [r14 + 12]
add    eax, dword ptr [r15 + 12]
mov    dword ptr [rbx + 12], eax
mov    eax, dword ptr [r14 + 16]
add    eax, dword ptr [r15 + 16]
mov    dword ptr [rbx + 16], eax
mov    eax, dword ptr [r14 + 20]
add    eax, dword ptr [r15 + 20]
mov    dword ptr [rbx + 20], eax
mov    eax, dword ptr [r14 + 24]
add    eax, dword ptr [r15 + 24]
mov    dword ptr [rbx + 24], eax
mov    eax, dword ptr [r14 + 28]
add    eax, dword ptr [r15 + 28]
mov    dword ptr [rbx + 28], eax
```

example::add_sse2:

```
movdqu xmm0, xmmword ptr [rdi]
movdqu xmm1, xmmword ptr [rsi]
padd    xmm1, xmm0
movdqu  xmmword ptr [rdx], xmm1
movdqu  xmm0, xmmword ptr [rdi + 16]
movdqu  xmm1, xmmword ptr [rsi + 16]
padd    xmm1, xmm0
movdqu  xmmword ptr [rdx + 16], xmm1
ret
```

example::add_avx2:

```
vmovdqu ymm0, ymmword ptr [rsi]
vpadd    ymm0, ymm0, ymmword ptr [rdi]
vmovdqu  ymmword ptr [rdx], ymm0
vzeroupper
ret
```

cfg-based switching

```
1  pub fn add(a: &[u32; 8], b: &[u32; 8], c: &mut [u32; 8]) {
2      cfg_if::cfg_if! {
3          if #[cfg(target_feature = "avx2")] {
4              unsafe { add_avx2(a ,b, c) }
5          } else if #[cfg(target_feature = "sse2")] {
6              unsafe { add_sse2(a ,b, c) }
7          } else {
8              add_default(a ,b, c)
9          }
10     }
11 }
```

Example: bounding boxes

- We have $8*N$ points given as `u32` and list of bounding boxes
- We want to find points which are inside of at least one bounding box

```
pub unsafe fn foo(  
    x: &[_m256i; N],  
    y: &[_m256i; N],  
    z: &[_m256i; N],  
    bboxes: &[[_m256i; 6]],  
) -> [_m256i; N] {  
    let mut res = [_mm256_setzero_si256(); N];  
    for bbox in bboxes {  
        for i in 0..N {  
            let tx = _mm256_and_si256(  
                _mm256_cmpgt_epi32(x[i], bbox[0]),  
                _mm256_cmpgt_epi32(bbox[1], x[i]),  
            );  
            let ty = _mm256_and_si256(  
                _mm256_cmpgt_epi32(y[i], bbox[2]),  
                _mm256_cmpgt_epi32(bbox[3], y[i]),  
            );  
            let t = _mm256_and_si256(tx, ty);  
            let tz = _mm256_and_si256(  
                _mm256_cmpgt_epi32(z[i], bbox[4]),  
                _mm256_cmpgt_epi32(bbox[5], z[i]),  
            );  
            let t = _mm256_and_si256(t, tz);  
            res[i] = _mm256_or_si256(res[i], t);  
        }  
    }  
    res  
}
```

Target feature disadvantages

- `#[target_feature(enable = "...")]` have to be marked unsafe (changed in Rust 1.86)
- Runtime target feature checks are not enforced by compiler
- It's currently impossible to specify that target feature will NOT be present during execution
- Compiler sometimes generates suboptimal code
- No vector extensions support
- On x86 SIMD-based code badly interacts with soft-float targets

Future of target features

- Reduction of **unsafe** amount in target feature v1.1:
<https://rust-lang.github.io/rfcs/2396-target-feature-1.1.html>
- Vector extensions support:
<https://github.com/rust-lang/rfcs/pull/3268>
- Portable simd (a.k.a **std::simd**):
<https://github.com/rust-lang/portable-simd>

Questions?