15.

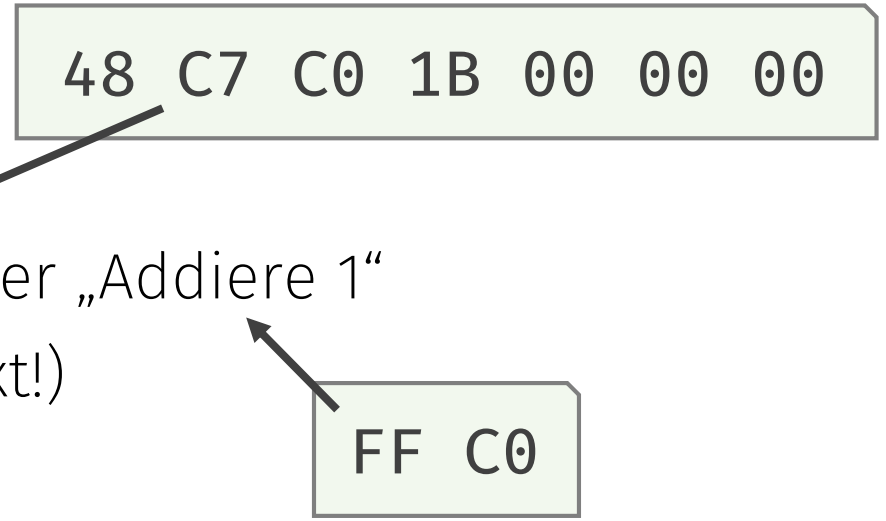
Low Level: Speicher, Assembly, ...

Motivation für Low Level

- Hilft, Rust Konzepte und Designentscheidungen zu verstehen
- Hilft, besseren Code in C/C++/Rust/… zu schreiben
  - Schneller/effizienter
  - In C und C++: Sicherer (in Rust müssen wir uns darum keine Sorgen machen ;-))
- Sinnvoll für andere Uni-Kurse
  - „Programmiersprache C++“, „Info C“, „Betriebssysteme“, …
- Assembly verstehen immer noch wichtig!
- Geschwindigkeit von Programmen/Sprachen besser verstehen

Maschinencode und Instructions

- *Maschinencode*: Reihe von *Instructions*
- *Instruction*: Primitiver Befehl für CPU
  - *Beispiel*: „Lade den Wert 27 an diese Stelle“ oder „Addiere 1“
  - Wird sehr kompakt kodiert (binär, nicht als Text!)
    - In x86_64: 1 bis 15 Bytes
  - Wird von CPU nacheinander ausgeführt (…)
- Hängt von CPU-Architektur ab
  - **x86_64:** Zurzeit quasi alle Desktop/Notebook CPUs (in diesen Slides genutzt)
  - **ARM**: Smartphones, Raspberry Pi und viele mehr…
  - …




```
55 48 89 E5 31 C0 48 85 FF 48 8D 47 FF 48 8D 4F
FE 48 F7 E1 48 0F A4 C2 3F 48 8D 44 3A FF 5D C3
```

Assembly

- Darstellung von *Maschinencode* als Text (für Menschen)
- Grundsätzliche zwei Syntaxen:
  - AT&T und Intel
  - Wir nutzen Intel-Syntax in den Slides
- Wir betrachten immer Assembly
  - **Aber**: „Alles ist binär kodiert“ im Hinterkopf behalten

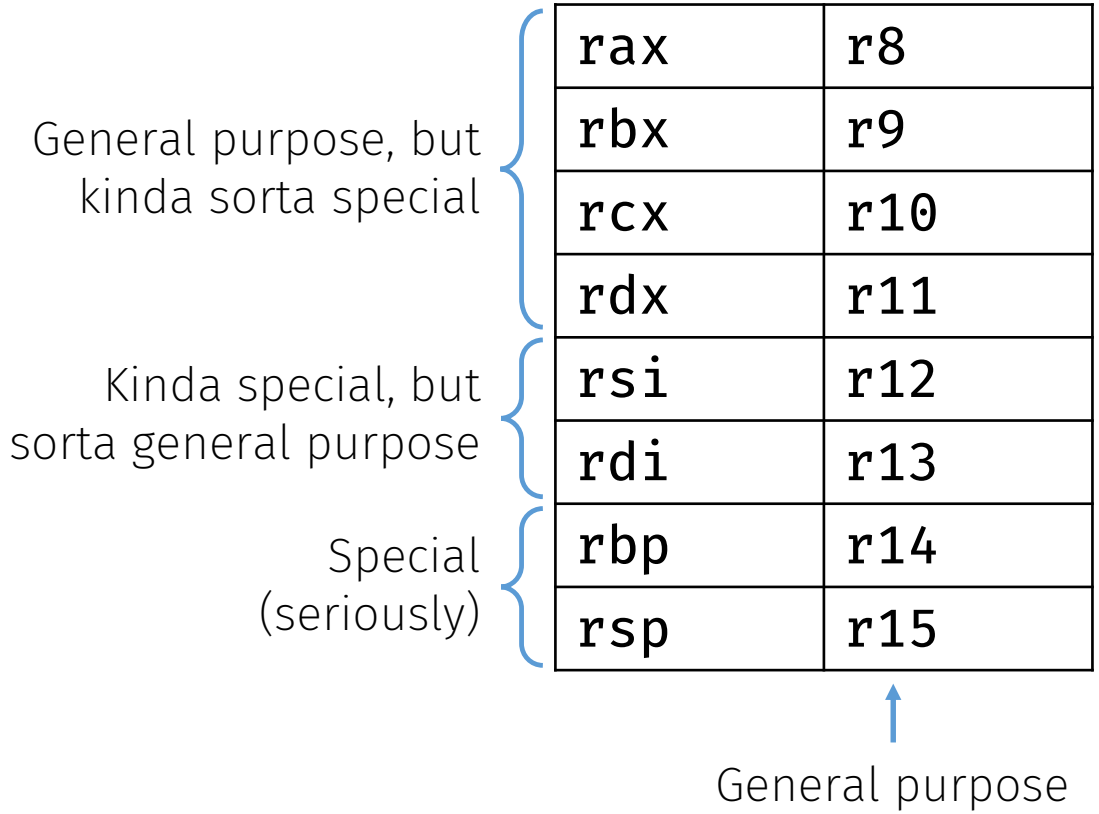
```
triangle:
        push    rbp
        mov     rbp, rsp
        xor     eax, eax
        test    rdi, rdi
        je      .LBB0_2
        lea     rax, [rdi - 1]
        lea     rcx, [rdi - 2]
        mul     rcx
        shld    rdx, rax, 63
        lea     rax, [rdx + rdi - 1]
.LBB0_2:
        pop     rbp
        ret
```

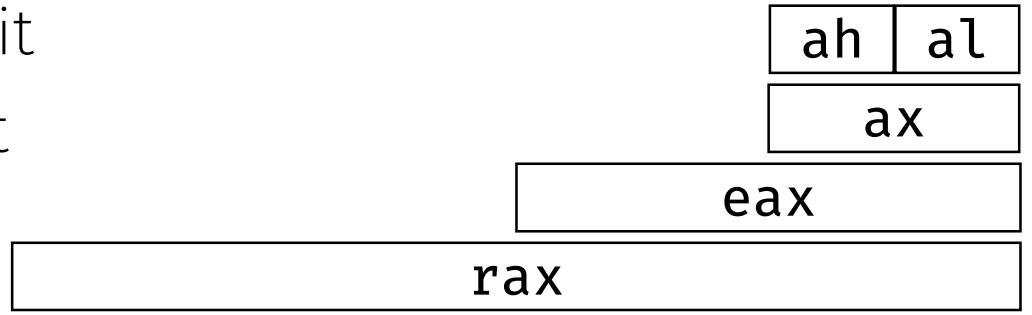
„Assembilieren“ ↓ ↑ „**De**assembilieren“

```
55 48 89 E5 31 C0 48 85 FF 48 8D 47 FF 48 8D 4F
FE 48 F7 E1 48 0F A4 C2 3F 48 8D 44 3A FF 5D C3
```

Register

- Speicher direkt in der CPU
- Extrem schneller Zugriff
- Für x86_64:
  - 16 Register
  - 64 Bit groß
- Teile der Register haben Namen:
  - `e??` bzw. `r??d` → unteren 32 Bit
  - `??` bzw. `r??w` → unteren 16 Bit
  - `wat` bzw. `r??b` → unteren 8 Bit





Beispiel-Instructions

- *Zuerst*: Art der Operation
- *Danach*: Argumente
  - Mit Komma getrennt
  - Register oder „*Intermediates*“
- Argumentreihenfolge manchmal merkwürdig
  - `mov` *`ziel, quelle`*

```
inc rax
```

```
FF C0
```

- Inkrementiert den Wert in **`rax`**

```
mov rax, 27
```

```
48 C7 C0 1B 00 00 00
```

- Füllt **`rax`** mit dem Wert 27

```
add rax, rbx
```

```
48 01 D8
```

- Addiere **`rbx`** auf **`rax`** auf (Ergebnis also in **`rax`**)

Sprünge

- Ausführung springt zu einer Adresse
  - Instruktion an dieser Adresse wird als nächstes ausgeführt
  - Addresse relativ zur jetztigen Instruktion
  - In Assembly werden Sprungziele benannt

```
main:                           ; comments start with ';'
    mov   rax, 27               ; load 27 into rax
    jmp   .loop_forever         ; unconditional jump
    mov   rbx, rax              ; would load the value of rax into
                                ; rbx, but won‘t be executed!

.loop_forever:
    jmp   .loop_forever         ; hihihi
```

Flags & Arten von Sprüngen

ZF	zero
SF	sign
CF	carry
…	

- Bedingungslose Sprünge: **`jmp`**
- *Flags*: Werden von Instructions gesetzt und gelesen
  - **`cmp rax, rbx`**: Subtrahiert **`rax`** von **`rbx`** und setzt *ZF*=1 wenn Ergebnis 0
- Bedingter Sprung:
  - **`jz`** (jump zero), **`je`** (jump equal): Sprung wenn *ZF* == 1
  - **`jnz`** (jump not zero), **`jne`** (jump not equal): Sprung wenn *ZF* == 0
  - Weitere: **`jb`** (below), **`jnb`** (not below), **`ja`** (above), **`jna`** (not above), …
- Dynamischer Sprung
  - **`jmp rax`**: Springt an die Adresse, die in **`rax`** gespeichert ist

Do-While-Schleife

```rust
let mut i = 0;    // will be in rax
let mut sum = 0;  // will be in rbx
do {  // there is no do-while in Rust!
    sum += i;
    i += 1;
} while (i != 10);
```

```
main:
    mov   rax, 0          ; load 0 into rax
    mov   rbx, 0          ; load 0 into rbx
.start_loop:
    add   rbx, rax        ; rbx += rax
    inc   rax             ; rax += 1
    cmp   rax, 10         ; compare (sets ZF=1 if equal)
    jne   .start_loop     ; if ZF==0 jump!

    nop                   ; Nothing to do anymore. Just for fun:
                          ; nop (no operation) does nothing :-)
```

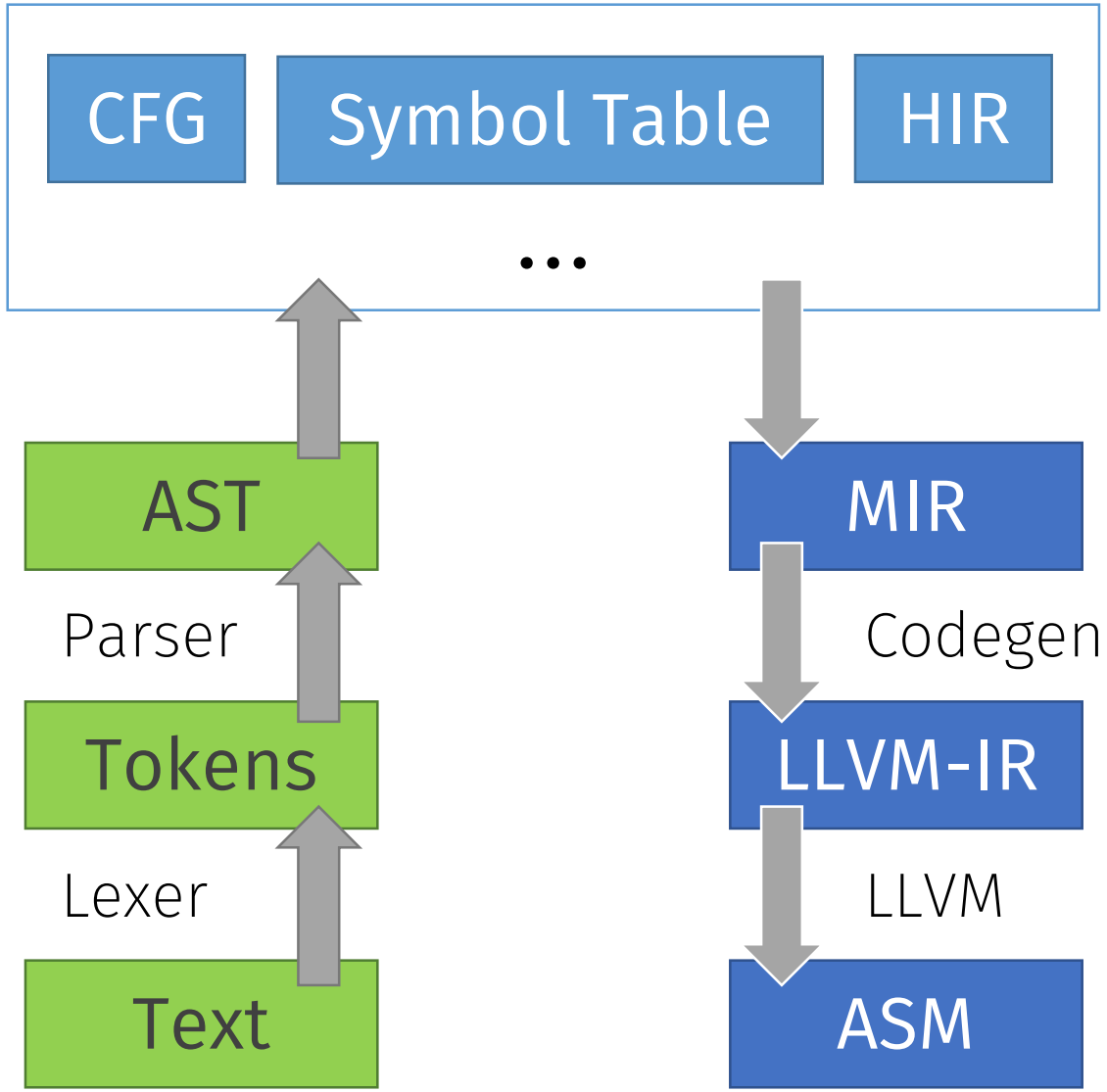
While-Schleife

```asm
main:
    mov   rax, 0          ; load 0 into rax
    mov   rbx, 0          ; load 0 into rbx
.start_loop:
    cmp   rax, 10         ; compare (sets ZF=1 if equal)
    je    .end            ; if ZF==1 jump!
    add   rbx, rax        ; rbx += rax
    add   rax, 1          ; rax += 1
    jmp   .start_loop     ; jump back up again

.end:
    nop                   ;-)
```

```rust
let mut i = 0;    // will be in rax
let mut sum = 0;  // will be in rbx
while i != 10 {
    sum += i;
    i += 1;
}
```

Von Rust zu Maschinencode



- **IR*: Intermediate Representation
- *LLVM-IR* ≈ CPU-unabhängiger Maschinencode
- *LLVM*
  - Von *LLVM-IR* zu Maschinencode
  - Unterstützt **viele** CPUs
  - Optimiert Code (!!!)
  - Wird auch von clang (C++) benutzt

Demo

Komische Instruktionen?

```
xor rax, rax
```

- Setzt **`rax`** auf 0 (diese Instruction ist kürzer als **`mov rax, 0`**)

```
lea rax, [eine Rechnung]
```

- Schreibt Ergebnis der Rechnung in **`rax`**
  - CPU kann gewisse, einfache Rechnungen so schneller ausführen, als mit mehreren **`mov`**, **`add`**, … Instruktionen

```
mul rdi
```

- Rechenoperationen mit einem Operanden nutzen meist **`rax`** als ersten Operanden