Assembler Crashkurs

4.1 Was ist ein Assembler?

- Ein (einfacher) Compiler, der den Code eines Assemblerprogramms in Maschinensprache umsetzt
  - Assemblerprogramm = menschenlesbare Instruktionen
  - Maschinenprogramm = binäre Darstellung der Instruktionen
- Komfortabler zu programmieren:
  - Statt der Bitfolge 000001011110100000000011 kann der Programmierer schreiben: `add ax,1000`
- Es existiert also eine eindeutige Abbildung von Assemblerbefehlen zu binären Maschineninstruktionen:

Symbolische Bezeichnung	Maschinencode
add ax	00000101
1000 (dez.)	0000001111101000

- Jede CPU-Architektur hat also ihren speziellen Assembler

Byte-Order

- Achtung – zusätzlich vertauscht der Assembler noch die Reihenfolge der Bytes des Parameters:

```
00000101 11101000 00000011
```

add ax	low byte	high byte
(1. Byte)	(2. Byte)	(3. Byte)

- little endian byte order (niederwertiges Byte an kleinerer Adresse) bei x86!

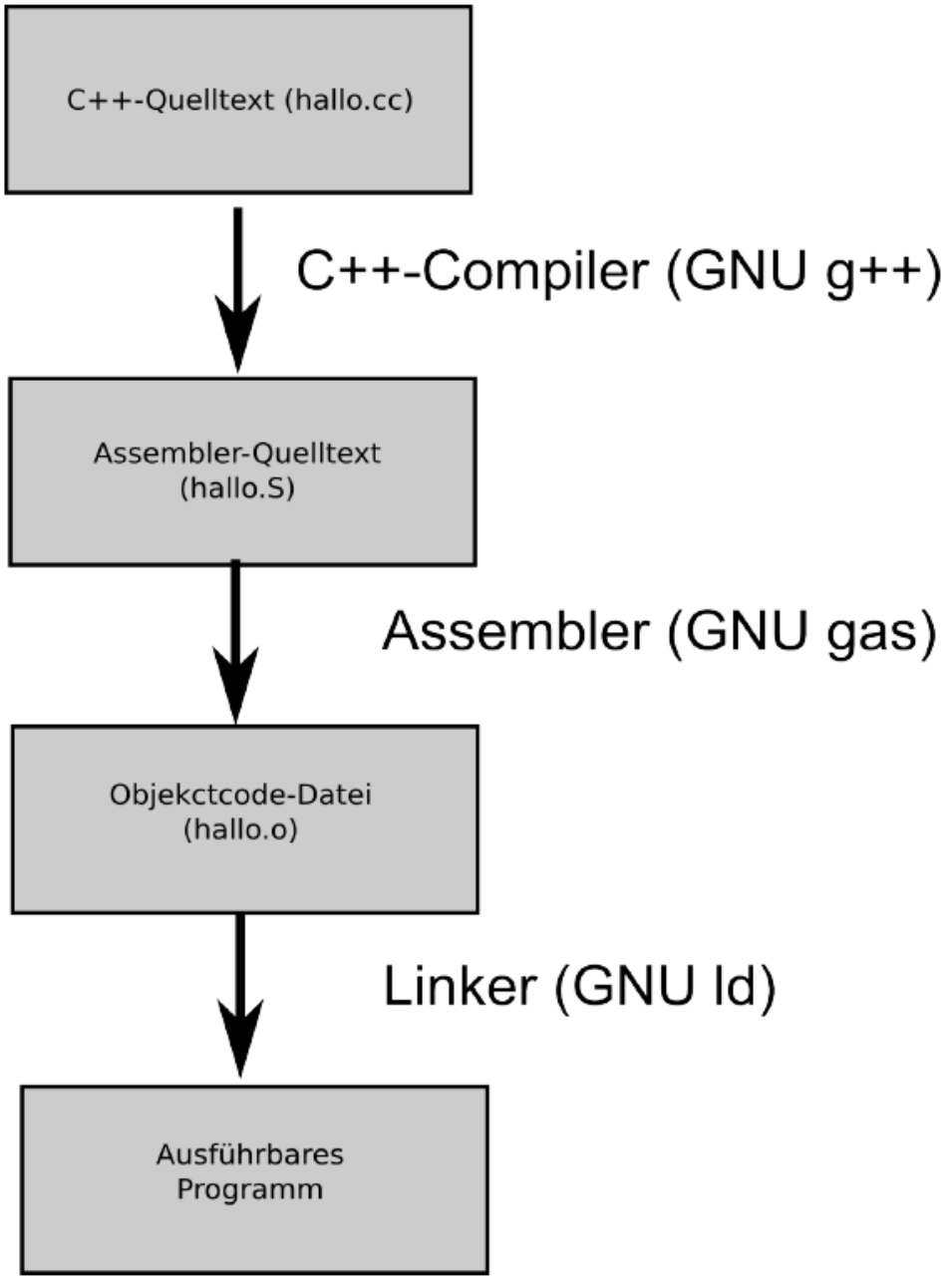
- Unter „Assembler“ versteht man also sowohl ...
  - die symbolische Darstellung der Maschinensprachebefehle wie auch ...
  - das Übersetzerprogramm, das die symbolische Darstellung in die vom Prozessor verstandene Binärdarstellung umsetzt

4.2 Was kann ein Assembler?

- Im Gegensatz zu „echten“ (Hochsprachen-)Compilern kann ein Assembler nur sehr wenige komplexe Ausdrücke umsetzen.
- Die Sprache, die der Assembler „versteht“, entspricht den im Instruktionssatz der jeweiligen CPU verfügbaren Instruktionen
  - Manche Assembler können allerdings zur Assemblier-Zeit auch einfache Berechnungen durchführen und besitzen einen simplen Präprozessor
- Konstrukte höherer Programmiersprachen werden vom Compiler in einfachere Instruktionen umgesetzt:
  - keine komplexen Anweisungen
  - keine komfortablen Schleifen — meist nur „goto“-Äquivalente
  - keine strukturierten Datentypen
  - keine Unterprogramme mit Parameterübergabe

4.3 Übersetzungsvorgang bei C++

- Der Assembler steht als Komponente zwischen Compiler und Linker
- Er liest vom Compiler erzeugten Assembler-Quelltext und erzeugt daraus Objektcode-Dateien, die die zugehörigen binären Maschineninstruktionen und Daten enthalten
- Diese Objektdateien werden vom Linker zu einem fertigen ausführbaren Programm verbunden



Beispiel

- Die C-Anweisung

```
summe = a + b + c + d;
```

  ist für einen Assembler zu kompliziert und muss daher in mehrere Anweisungen aufgeteilt werden

- Der 80x86 Assembler kann immer nur zwei Zahlen addieren und das Ergebnis in einer der beiden verwendeten "Variablen" (Akkumulatorregister) speichern.

- Dieses C-Programm entspricht von der Struktur her also eher einem Assemblerprogramm:

```
summe = a;
summe = summe + b;
summe = summe + c;
summe = summe + d;
```

Beispiel (Fortsetzung)

- Dieses Programm

```
summe = a;
summe = summe + b;
summe = summe + c;
summe = summe + d;
```

  würde in 80x86-Assembler z.B. so aussehen:

```
mov rax, [a]
add rax, [b]
add rax, [c]
add rax, [d]
```

- Assembler unterstützten also nur primitive Operationen
- Die meisten Assembler arbeiten zeilenorientiert
  - eine Zeile entspricht einer Maschinenanweisung
  - kein Semikolon o.ä. am Ende der Zeile notwendig

Kontrollanweisungen: „if“

- Einfache if-then-else-Konstrukte sind für Assembler auch schon zu komplex:

```
if ( a == 4711 ) {
     ...
} else {
     ...
}
```

- In 80x86-Assembler sieht das wie folgt aus:

```
    cmp rax, 4711  ; vergleiche eax mit 4711
    jne ungleich   ; ungleich -> springe

gleich: ...        ; sonst hier weiter (if-Zweig)
    jmp weiter     ; else-Zweig auslassen

ungleich:...       ; else-Zweig

weiter: ...        ; danach geht's weiter
```

Schleifen: einfache „for“-Schleife

- Simple Zählschleifen werden vom 80x86 schon besser unterstützt:

```
for (i=0; i <100; i++) {
     summe = summe + a;
}
```

- ... in Assembler:

```
    mov rcx, 100
schleife:
    add rax, [a]
    loop schleife
```

- Der Loop-Befehl dekrementiert implizit das ecx-Register und führt den Sprung nur dann aus, wenn der Inhalt von ecx anschließend nicht 0 ist

4.4 Register

- In Assembler existieren keine beliebigen Variablen. Werte als Parameter/Ergebnisse aktueller Berechnungen müssen in CPU-Registern gehalten werden
- Ein Register ist ein extrem schneller, sehr kleiner Zwischenspeicher innerhalb der CPU, der (beim 80x86) bis zu 32/64 Bits speichern kann
- Die eigentlichen Variablen der Hochsprache werden vom Compiler zu Speicherplätzen im Datensegment der Objektcode-Datei zugeordnet
- Um eine Berechnung mit Variablen durchführen zu können, muss erst der Wert aus der jeweiligen Speicherzelle in ein Register geladen werden
  - Nicht alle Variablen passen gleichzeitig in die wenigen Register!
  - Es existiert also eine zeitlich sich verändernde Zuordnung Register <=> Variable

4.5 Speicher

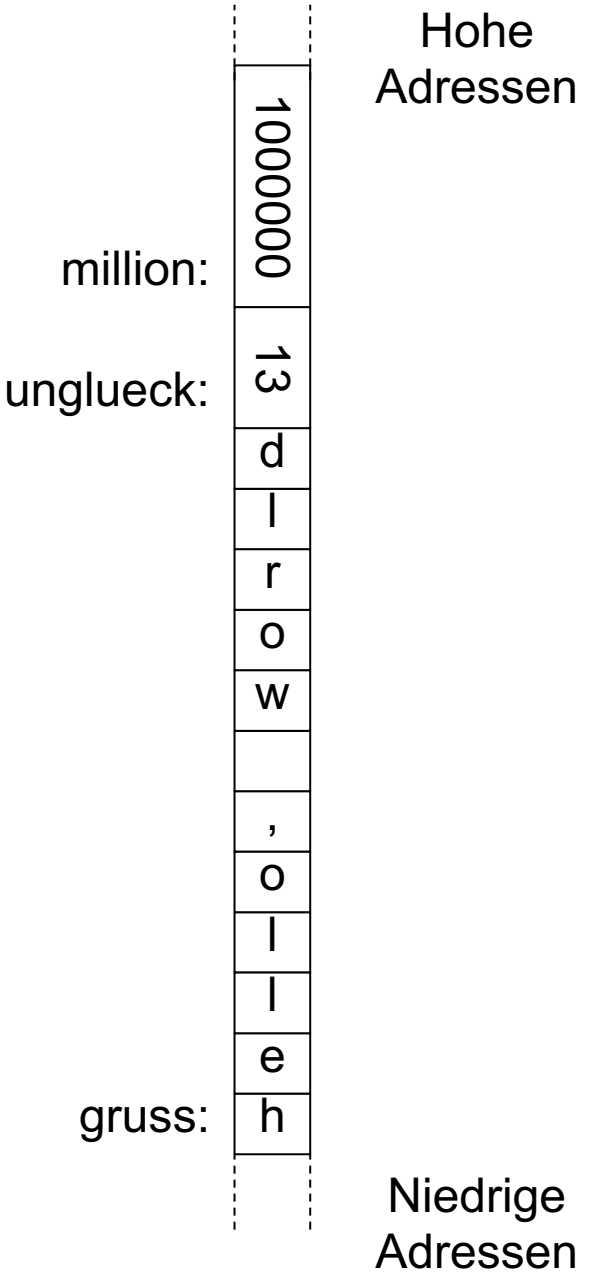
- Meistens reichen Register alleine zur Lösung eines Problems nicht aus
  - Zugriff auf den Hauptspeicher ist erforderlich
- Der Hauptspeicher ist wie ein riesiges Array aus Registern, die wahlweise 8, 16, 32 oder 64 Bit „breit“ sind
  - Ein Byte ist die kleinste adressierbare Einheit
  - Speicherzellen werden durchnummeriert => Index
  - Die Zugriffsgeschwindigkeit auf den Hauptspeicher (~12ns) ist viel langsamer als die Zugriffsgeschwindigkeit auf Register (<1ns)
- Um auf einen Wert im Hauptspeicher zugreifen zu können, muss der Programmierer den Index des Wertes im Speicher kennen. Dieser ist die Adresse des Wertes
- Der Hauptspeicher wird in Bytes von 0 an aufsteigend indiziert

Speicher: Beispiel

- Beispiel

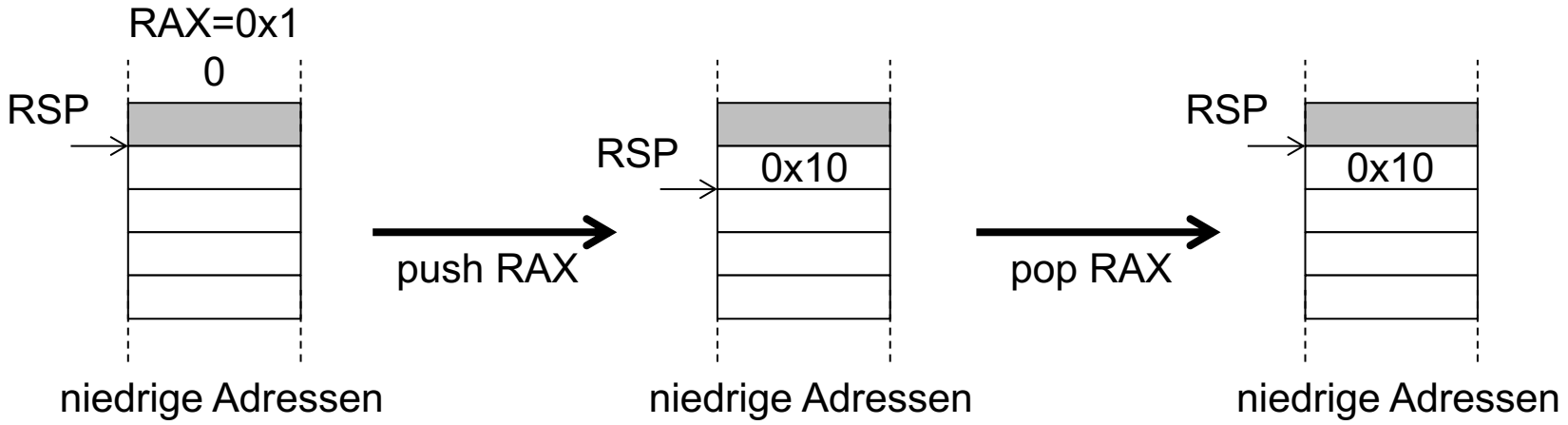
```
[SECTION .data
gruss:     db ’hello world’
unglueck:  dw 13
million:   dq 100000

[SECTION .text]
mov rax, [million]
```

db: define byte
dw define word (2 Byte)
dd: define double word (4 Byte)
dq: define quad word (8 Byte)



Der Stack

- Werte werden mit der `push`-Operation „oben“ auf den Stack gelegt, die `pop`-Operation entfernt den obersten Wert wieder vom Stack
  - dabei ist die aktuelle Adresse, an der `push/pop` operieren, in einem speziellen Register, dem sog. Stack Pointer (Register: `RSP`) gespeichert (`RSP` zeigt immer auf den zuletzt belegten Eintrag)
  - der Programmierer muss sich aber nicht um den konkreten Wert des Stack Pointers kümmern, sondern sich nur die Reihenfolge merken, in der Werte auf dem Stack abgelegt hat
  - Der Stack arbeitet immer mit 32 / 64 Bit Werten (je nach CPU-Modus)



4.6 Adressierungsarten

- Die meisten Befehle des x86 können Operanden wahlweise aus Registern, aus dem Speicher oder direkt einer Konstante entnehmen
- Beim `mov`-Befehl sind (u.a.) folgende Formen möglich, wobei der erste Operand stets das Ziel und der zweite stets die Quelle der Operation angeben:
  - **Registeradressierung** – der Wert eines Registers wird in ein anderes übertragen: `mov rbx, rdi`
  - **Unmittelbare Adressierung** – eine Konstante wird in ein Register geladen: `mov rbx, 1000`
  - **Direkte Adressierung** – der Wert, der an der angegebenen Speicherstelle steht, wird ins Register übertragen: `mov rbx, [1000]`
  - **Register-indirekte Adressierung** – der Wert, der an der Speicherstelle steht, die durch das zweite Register bezeichnet wird, wird in das erste Register geladen: `mov rbx, [rax]`
  - **Basis-Register Adressierung** – Der Wert, der an der Speicherstelle steht, die sich durch die Summe des Inhalts des zweiten Registers und der Konstanten ergibt, wird in das erste Register geladen: `mov rax,[10+rsi]`

4.7 Funktionen

- Aus den höheren Programmiersprachen ist das Konzept der Funktion oder Prozedur bekannt
  - Der Vorteil dieses Konzeptes gegenüber einem goto besteht darin, daß die Prozedur von jeder beliebigen Stelle im Programm aufgerufen werden kann und das Programm anschließend an genau der Stelle fortgesetzt wird, die nach dem Prozeduraufruf folgt
  - Die Prozedur selbst muss nicht wissen, von wo sie aufgerufen wurde und wo es hinterher weiter geht. Das geschieht über den Stack automatisch
- Nicht nur die Daten des Programms, sondern auch das Programm selbst liegt im Hauptspeicher und somit gehört zu jeder Maschinencodeanweisung eine eigene Adresse
- Damit der Prozessor ein Programm ausführt, muss sein Befehlszeiger auf den Anfang des Programms zeigen, also die Adresse der ersten Maschinencodeanweisung in das spezielle Register Befehlszeiger (instruction pointer, eip) geladen werden.

- Der Prozessor wird dann den auf diese Weise bezeichneten Befehl ausführen und im Normalfall anschließend den Inhalt des Befehlszeigers um die Länge des Befehls im Speicher erhöhen, so dass er auf die nächste Maschinenanweisung zeigt

- Bei einem Sprungbefehl wird der Befehlszeiger nicht um die Länge des Befehls, sondern entweder
  - auf die angegebene Zieladresse umgesetzt (absoluter Sprung) oder
  - um die angegebene relative Zieladresse erhöht oder erniedrigt
- Um nun eine Funktion aufzurufen, wird zunächst einmal wie beim Sprungbefehl verfahren, nur dass der alte Wert des Befehlszeigers (+ Länge des Befehls) zuvor auf den Stack geschrieben wird
  = Rücksprungadresse
- Am Ende der Funktion wird die Rücksprungadresse vom Stack genommen und an diese Adresse gesprungen

`call` & `ret`

- Beim 80x86 erfolgt das Speichern der Rücksprungadresse auf dem Stack implizit mit Hilfe des **`call`** Befehls. Genauso führt der **`ret`** Befehl auch implizit einen Sprung (Rücklsprung) an die auf dem Stack liegende Adresse durch:

```
; ----- Hauptprogramm ----
;
main: ...
      call f1

xy:   ...

; ----- Funktion f1
f1:   ...
      ret
```

Application Binary Interface (ABI)

- Wenn die Funktion Parameter erhalten soll, werden diese in Registern und oder dem Stack abgelegt
- Wie dies genau abläuft legt das ABI fest
  - Größe und Byteorder primitiver Datentypen
  - Aufruf-Konvention (engl. calling convention): wo werden Parameter übergeben sowie Rückgabewerte zurückgegeben (Register, Stack)
  - Größe und Byteorder primitiver Datentypen
  - Das ABI beschreibt allgemein ein Interface zwei binären Programmteilen

- Wir verwenden die x64 ABI (sowohl in C++ als auch Rust)
  - https://learn.microsoft.com/en-us/cpp/build/x64-software-conventions

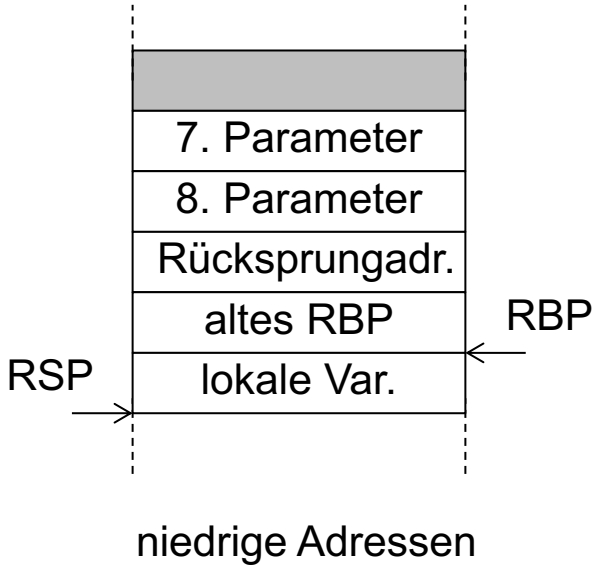
Funktionen mit Parameter

- Wenn die Funktion Parameter erhalten soll, werden diese bei x86_64 in Registern und nur bei Bedarf auf dem Stack abgelegt, natürlich vor dem `call` Befehl.
- Hinterher müssen sie naturlich wieder entfernt werden, entweder mit `pop`, oder durch direktes Umsetzen des Stack Pointers nach dem `call` oder beim `ret`, z.B. `ret 16`
- Parameterübergabe links nach rechts

Parameter Nummer	Register / Stack
1	`rdi`
2	`rsi`
3	`rdx`
4	`rcx`
5	`r8`
6	`r9`
>6	stack

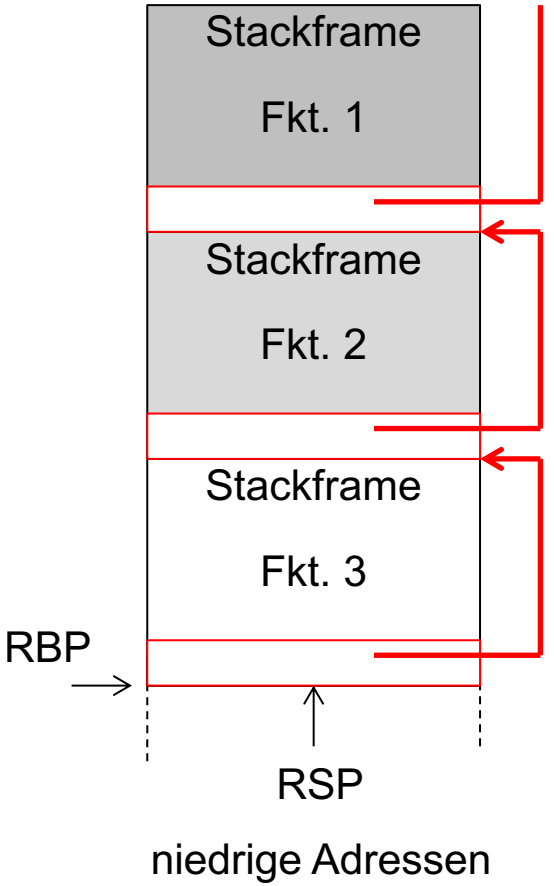
Typischer Stackframe

- Adressierung evt. Parameter über `RBP` und positivem Offset
- Adressierung lokaler Variablen über `RBP` und negativem Offset
- Im 64-Bit Modus hat jeder Stackeintrag 64-Bit



Kette von Funktionsaufrufen

- Beispiel: Aufrufreihenfolge Fkt. 1 → Fkt. 2 → Fkt. 3
- Stackframes sind über alte RBP-Einträge verkettet



4.8 Schnittstelle C zu Assembler

- Ein Label im Assembler-Code kann dem Linker bekannt gemacht werden – auch die Adresse einer Funktion:

```
; EXPORTIERTE FUNKTIONEN

[GLOBAL Coroutine_switch]
[GLOBAL Coroutine_start]

Coroutine_start: ...
```

- Nun kann ein C++-Programm die Funktion aufrufen
  - Allerdings braucht der Compiler eine passende Deklaration in der C++ Datei

```
extern "C" void Coroutine_start  (void* context);
```

- Der 1. Parameter `context` kann vom Assembler-Code über das Register `RDI` entgegengenommen werden.

4.8 Schnittstelle Rust zu Assembler

- Ein Label im Assembler-Code kann dem Linker bekannt gemacht werden – auch die Adresse einer Funktion:

```
; EXPORTIERTE FUNKTIONEN

[GLOBAL _coroutine_switch]
[GLOBAL _coroutine_start]

_coroutine_start: ...
```

- Nun kann ein Rust-Programm die Funktion aufrufen
  - Allerdings braucht der Compiler eine passende Deklaration in der Rust-Datei

```
extern "C" fn _coroutine_start  (context: *mut c_void);
```

- Der 1. Parameter `context` kann vom Assembler-Code über das Register `RDI` entgegengenommen werden.
- Weitere Infos zu Inline-Assembly in Rust: https://doc.rust-lang.org/reference/inline-assembly.html

Rust Aufrufkonvention für Assemblerfunktionen

- Der Rust Compiler geht davon aus, dass für externe Funktionen die Aufrufkonvention der Standard C ABI auf der verwendeten Plattform ist
- Also in unserem Fall x86 ABI
- https://doc.rust-lang.org/reference/items/external-blocks.html#abi