



# The Conversion of Source Code to Machine Code

---

Silas Groh, Mik Müller

17. Mai 2023

Carl-Fuhlrott-Gymnasium

# Zentrales Problem

```
1 fn main() {  
2     foo(2);  
3 }  
4  
5 fn foo(n: int) {  
6     let mut m = 3;  
7     exit(n + m);  
8 }
```

Übersetzung



```
1 ; RISC-V binary  
2 00000000 457f 464c 0102  
3 00000010 0002 00f3 0001  
4 00000020 0040 0000 0000  
5 00000030 0005 0000 0040  
6 00000040 0003 7000 0004  
7 00000050 0000 0000 0000  
8 00000060 0048 0000 0000  
9 00000070 0001 0000 0000  
10 00000080 0000 0000 0000
```

- Programme sollten **einfach** zu schreiben sein
- ⇒ Ein Computer muss diese jedoch auch **einfach** verarbeiten

**TODO**

---

- **TODO: translate everything into German**
- **TODO: fix listings captions**

## **Einstieg & Motivation**

---

- Programme werden in speziellen Sprachen verfasst
- Vorteile eines hohen Abstraktionsgrades [Dan05a, S. 9]



Erweiterbarkeit &  
Reperatur



Portabilität &  
Plattformunabhängigkeit


[Goo23]



Geschwindigkeit &  
Einfachheit

- Man unterscheidet zwischen **Compilern** und **Interpretern**
- Compiler: übersetzt das Programm in ein Zielformat
- Interpreter: führt das Programm direkt aus (keine Übersetzung)

```
1 fn main() {  
2     foo(2);  
3 }  
4  
5 fn foo(n: int) {  
6     let mut m = 3;  
7     exit(n + m);  
8 }
```

Übersetzung 



```
1 ; RISC-V binary  
2 00000000 457f 464c 0102  
3 00000100 0002 00f3 0001  
4 00000200 0040 0000 0000  
5 00000300 0005 0000 0040  
6 00000400 0003 7000 0004  
7 00000500 0000 0000 0000  
8 00000600 0048 0000 0000  
9 00000700 0001 0000 0000  
10 00000800 0000 0000 0000
```

- Rust, C, Go, usw.
  - Zusätzlicher Prozess
  - Umwandlung in ein anderes Format
- ⇒ Muss vor der Ausführung ablaufen



```
1 fn main() {  
2     foo(2);  
3 }  
4  
5 fn foo(n: int) {  
6     let mut m = 3;  
7     exit(n + m);  
8 }
```

Ausführung



Exit code: 5

- Python, Javascript, PHP, usw.
  - Keine Übersetzung notwendig
- ⇒ Interpretiert den Syntaxbaum direkt



**Abbildung 1** – Stufen der Übersetzung.



**Abbildung 2** – Stufen der Übersetzung (angepasst).

**Tabelle 1** – Die wichtigsten Fähigkeiten von rush.

Bezeichnung	Beispiel
Schleife	<code>loop { }</code>
while-Schleife	<code>while x &lt; 5 { }</code>
for-Schleife	<code>for i = 0; i &lt; 5; i += 1 { }</code>
if-Verzweigung	<code>if true { } else { }</code>
Funktionsdefinition	<code>fn foo(n: int) { }</code>
infix-expression	<code>1 + n; 5 ** 2</code>
prefix-expression	<code>!false; -n</code>
let-statement	<code>let mut answer = 42</code>
cast-expression	<code>42 as float</code>

**Tabelle 2** – Datentypen in rush.

Bezeichnung	Instanziierung einer Variable
„int“	<code>let a: int = 0;</code>
„float“	<code>let b: float = 3.14;</code>
„bool“	<code>let c: bool = true;</code>
„char“	<code>let d: char = 'a';</code>
„()“	<code>let e: () = main();</code>
„!“	<code>let f = exit(42);</code>

## Beispiel Programmtext in rush

```
1  fn main() {  
2      exit(fib(10));  
3  }  
4  
5  fn fib(n: int) -> int {  
6      if n < 2 {  
7          n  
8      } else {  
9          fib(n - 2) + fib(n - 1)  
10     }  
11 }
```

**Listing 1.1** – Beispiel: Berechnung von Fibonaccizahlen in rush.

- ca. fünf Monate intensive Entwicklung
- 814 Git Commits
- 17526 Zeilen Programmtext<sup>1</sup> in Git Commit '9953dd8'
- Inhalte des Projektes
  - Lexer
  - Parser
  - zwei Interpreter
  - ein Transpiler
  - vier Compiler

---

<sup>1</sup>Leerzeilen und Kommentare werden nicht gezählt.

**Tabelle 3** – Zeilen Programmtext pro Komponente.

Komponente	Zeilen Programmtext
Lexer / Parser	2737
Tree-walking interpreter	578
VM compiler / runtime	1288
WASM compiler	1641
LLVM compiler	1450
RISC-V compiler	2234
x86 compiler	2751

## **Lexikalische & Syntaktische Analyse**

---





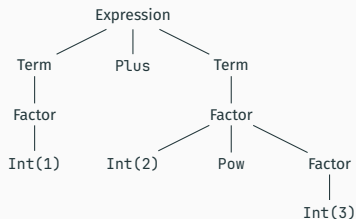
**Abbildung 3** – Etappen der Übersetzung: Syntaxanalyse.

[Wir05, S. 6–7]

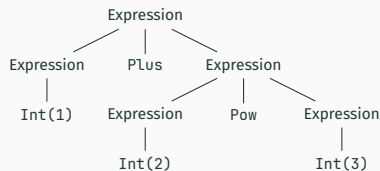
- Gruppieren des Programmtextes in Tokens
- Analyse der Syntax des Programms
- Erzeugung eines abstrakten Syntaxbaums
- Festlegen der formalen Regeln in Form einer Grammatik

```
1 Expression = Term , { ( '+' | '-' ) , Term } ;
2 Term       = Factor , { ( '*' | '/' ) , Factor } ;
3 Factor      = ( integer
4               | '(' , Expression , ')' ) , [ '**' , Factor ] ;
5 integer     = { '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8'
6               | '9' } - ;
```

**Listing 1.2** – Ein Beispiel für eine kontextfreie Grammatik (EBNF).



**Abbildung 4** – Abstrakter Syntaxbaum für '1+2\*3'.



**Abbildung 5** – Abstrakter Syntaxbaum für '1+2\*3', erstellt durch Pratt-Parsing.

## **Semantische Analyse**

---



**Abbildung 6** – Etappen der Übersetzung: Semantische Analyse.

[Wir05, S. 6–7]

- Validiert die semantische Eigenschaften
- Meistens: Definition in einer natürlichen Sprache

- Jede Variable, jede Funktion und jeder Parameter besitzt einen Datentyp, der nach Definierung nicht mehr geändert werden kann
- Eine Funktion muss immer mit den Argumenten aufgerufen werden, die zu den Parametern passen
- Jeder Funktionsname muss eindeutig sein
- Die „main“ Funktion liefert immer den „()“ Datentyp und akzeptiert keine Parameter
- Jede Variable muss Definiert sein, bevor diese Verwendbar ist
- Logische und mathematische Operationen erfolgen nur, wenn die Operanten den selben Datentypen besitzen
- Eine definierte Variable *sollte* verwendet werden

...

## Beispiel: Invalides rush Programm

```
1 fn main() {  
2     let num = 3.1415;  
3     num + 1;  
4 }
```



Fehlerausgabe

**TypeError** at incompatible\_types.rush:3:5

```
2 |     let num = 3.1415;  
3 |     num + 1;  
   |           ^^^^^^^  
4 | }
```

**infix expressions require equal types on both sides, got `float` and `int`**



## Beispiel 2: Invalides rush Programm

```
1 fn main(_n: int) -> bool {  
2     return true;  
3 }
```



Fehlerausgabe

**SemanticError** at invalid\_main\_fn.rush:1:8

```
1 | fn main(_n: int) -> bool {  
    ^^^^^^^^^^^  
2 |     return true;
```

the ``main`` function must have 0 parameters, however 1 is defined

**note:** remove the parameters: ``fn main() { ... }``

**SemanticError** at invalid\_main\_fn.rush:1:21

```
1 | fn main(_n: int) -> bool {  
    ^^^^^  
2 |     return true;
```

the ``main`` function's return type must be ``()``, but is declared as ``bool``

**note:** remove the return type: ``fn main() { ... }``

## Beispiel 3: Warnung aufgrund einer unbenutzten Variable

```
1 fn main() {  
2     let variable = 42;  
3     let mut code = 3;  
4     exit(code)  
5 }
```



Ausgabe

**Warning** at unused\_var.rush:2:9

```
1 | fn main() {  
2 |     let variable = 42;  
   |           ~~~~~  
3 |     let mut code = 3;
```

**unused variable** `variable`

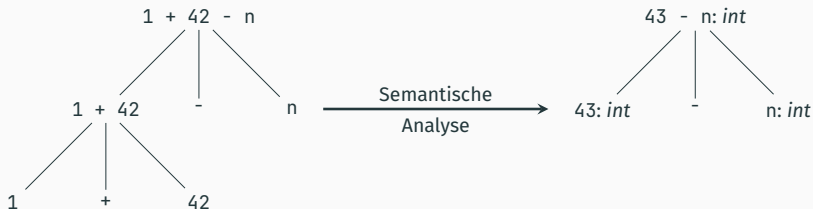
**note:** if this is intentional, change the name to `\_variable` to hide this  
↪ warning

**Info** at unused\_var.rush:3:13

```
2 |     let variable = 42;  
3 |     let mut code = 3;  
   |               ~~~~  
4 |     exit(code)
```

**variable** `code` does not need to be mutable

- Unterscheidung zwischen validen und invaliden Programmen
- Liefern von hilfreiche Warnungen und Informationen
- Hinzufügen von Typinformationen zu dem AST
- Triviale Optimierungen der Programmstruktur



**Abbildung 7** – Auswirkungen der semantischen Analyse auf den AST.

## Tree-walking Interpreter

---

**TODO: RubixDev Write this**

## **Virtuelle Maschine**

---

- Meistens: Eine *Virtuelle Maschine* (VM) simuliert echte Computer
  - Display
  - Lautsprecher
  - Festplatte
  - ...
- Hier: Software, die wie die CPU eines Rechners funktioniert



- Die meisten Prozessoren basieren auf der *von Neumann Architektur* [Led20, p. 172]
- Eine CPU enthält nach von Neumann ein *Rechenwerk*<sup>2</sup>, *Steuerwerk*<sup>3</sup>, *Speicherwerk*, *Ein- / Ausgabewerk* und ein Bussystem [Led20, p. 172]
- Die Programmausführung wird durch den sog. *Befehlszyklus*<sup>4</sup> modelliert [Led20, pp. 208-209]:
  1. **Fetch** (Befehl laden): Das Steuerwerk lädt die nächste Anweisung aus dem Speicher
  2. **Decode** (Befehl dekodieren): Der Befehlscode und die Operanden werden ermittelt
  3. **Execute** (Befehl ausführen): Die zuständige Einheit im Prozessor wird verwendet, um den Befehl zu verarbeiten. Beispielsweis wird das Rechenwerk für logische und mathemtische Befehle aufgerufen.

---

<sup>2</sup>Engl: „arithmetic logic unit“ (ALU).

<sup>3</sup>Engl: „control unit“.

<sup>4</sup>Engl: „fetch-decode-execute cycle“.

# Übertragung der Konzepte auf die rush VM **TODO: DELETE**

---

```
crates/rush-interpreter-vm/src/vm.rs
```

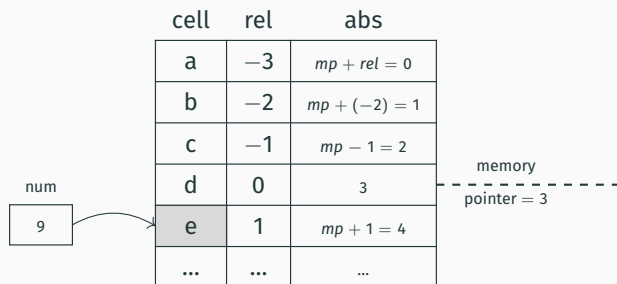
```
16 pub struct Vm<const MEM_SIZE: usize> {
17     /// Working memory for temporary values
18     stack: Vec<Value>,
19     /// Linear memory for variables.
20     mem: [Option<Value>; MEM_SIZE],
21     /// The memory pointer points to the last free location in memory.
22     /// The value is always positive, but using `isize` does not require
    ↪ casts.
23     mem_ptr: isize,
24     /// Holds information about the current position (like ip / fp).
25     call_stack: Vec<CallFrame>,
26 }
```

---

**Listing 1.3** – Struct Definition der VM.

- **TODO: fix broken caption**
- „stack“: Speicher für temporäre Werte bei komplexeren Operationen
- „mem“: Anhaltender Speicher mit einer festen Größe für Variablen
- „mem\_ptr“: Hält den Index der letzten freien Speicherzelle in „mem“
- „call\_stack“: Aufrufstapel, welcher den *Befehlszähler* und den *Funktionszähler* für jeden Aufruf speichert

- Unterscheidung zwischen zwei Arten der Adressierung
- *relative Adressierung*: „svari \*rel[0]“
- *absolute Adressierung*: „svari \*abs[0]“



**Abbildung 8** – Speicherstruktur der rush VM.

## Ein Beispielprogramm der rush VM **TODO: Delete**

```
1  0: (prelude)
2      setmp 0
3      call 1
4  1: (main)
5      setmp 0
6      push 1000
7      call 2
8      exit
9  2: (rec)
10     setmp 1
11     svari *rel[0]
12     push *rel[0]
13     gvar
14     push 0
15     eq
16     jmpfalse 9
17     push 0
18     jmp 14
19     push *rel[0]
20     gvar
21     push 1
22     sub
23     call 2
24     setmp -1
25     ret
```

**Listing 1.4** – Beispielprogramm der rush VM.

- Stack für temporäre Operationen
- Weiterer Stack für Funktionsaufrufe
- Unterteilung in Funktionen
  - Ohne Namen
  - numerische Identifizierung
  - Enthält mehrere Anweisungen
- Git Commit „9953dd8“: ca. 30 verschiedene Befehlscodes
- Struktur der Anweisungen: „call 2“
  - Befehlscode (call)
  - Optionaler Operand (2)

- Ca. 2.7 mal schneller als der Tree-walking Interpreter
- Einfache Implementierung des Compilers
  - Stack-basierte Architektur
  - Gleichzeitige Entwicklung von VM und Compiler
  - Hoher Abstraktionsgrad

## Demonstration: Eingabe

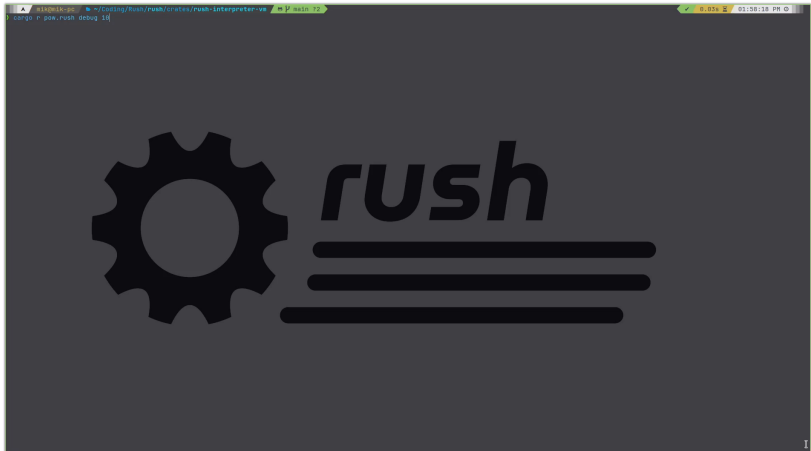
```
1  fn main() {
2      exit(pow(2, 4)); // 2 ** 4 = 16
3  }
4
5  fn pow(mut base: int, mut exp: int) -> int {
6      if exp == 0 {
7          return 1;
8      }
9      if exp < 0 {
10         return 0;
11     }
12
13     let mut acc = 1;
14
15     while exp > 1 {
16         if (exp & 1) == 1 {
17             acc *= base
18         }
19         exp /= 2;
20         base *= base;
21     }
22
23     acc * base
24 }
```

# Demonstration: Ausgabe

```
1  0: (prelude)
2      setmp 0
3      call 1
4  1: (main)
5      setmp 0
6      push 2
7      push 4
8      call 2
9      exit
10 2: (pow)
11     setmp 3
12     svari *rel[0]
13     svari *rel[-1]
14     push *rel[0]
15     gvar
16     push 0
17     # ...
60    push *rel[-1]
61    gvar
62    mul
63    svar
64    jmp 21
65    push *rel[-2]
66    gvar
67    push *rel[-1]
68    gvar
69    mul
70    setmp -3
71    ret
```



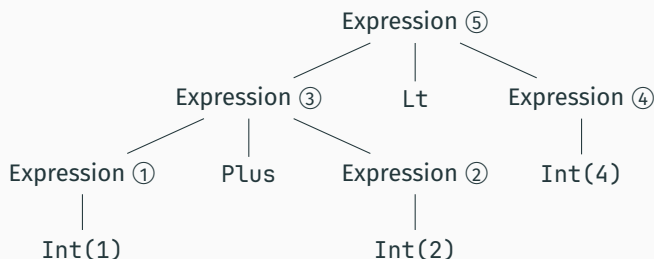
# Demonstration: Laufzeitverhalten



## **Kompilierung zu high-level Architekturen**

---

## Wie ein Compiler den AST traversiert



**Abbildung 9** – AST zu „1 + 2 < 4“.

---

```
1  r0 = 1
2  r1 = 2
3  r2 = add r0, r1
4  r3 = 4
5  r4 = lt r2, r3
```

---

**Listing 1.5** – Beispielausgabe zu „1 + 2 < 4“.

## Kompilierung zu WebAssembly

---

**TODO: @RubixDev Write this**

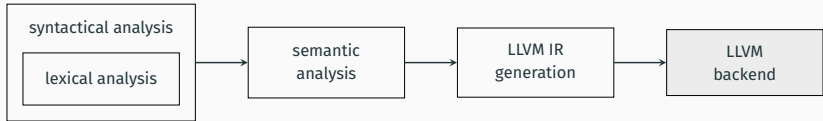
## Kompilierung zu LLVM

---

- Startete als Forschungsprojekt von *Chris Lattner* [Lat02]
  - Auch *Rust* und *Swift* nutzen LLVM<sup>5</sup>.
  - Erzeugung von Code aus einer Zwischendarstellung
  - Aggressive Optimisierungsmaßnahmen
  - Die sogenannte *intermediate representation* (IR) kann mittels APIs erzeugt werden [Hsu21, preface]
- ⇒ LLVM ist das backend eines Compilers

---

<sup>5</sup>[McN21, p. 373], [Hsu21, preface]



**Abbildung 10** – Etappen der Übersetzung mit Verwendung von LLVM



- Verwendung einer Rust library names **Inkwell**
- Erzeugung von LLVM IR

## Ein LLVM Beispielprogramm: Eingabe

```
1  fn main() {  
2      let mut res = 2f;  
3      for i = 0; i < 10; i += 1 {  
4          res /= 2f;  
5      }  
6      exit((res * 100f) as int)  
7  }
```

# Ein LLVM Beispielprogramm: Ausgabe

```
1  ; ModuleID = 'main'
2  source_filename = "main"
3  target triple = "x86_64-alpine-linux-musl"
4
5  define i32 @main() {
6  entry:
7      br label %for_head
8
9  for_head:                                ; preds = %for_body,
    ↪ %entry
10     %res2 = phi double [ %f_prod, %for_body ], [ 2.000000e+00, %entry ]
11     %i3 = phi i64 [ %i_sum, %for_body ], [ 0, %entry ]
12     %i_lt = icmp slt i64 %i3, 10
13     br i1 %i_lt, label %for_body, label %after_for
14
15  for_body:                                ; preds = %for_head
16     %f_prod = fmul double %res2, 5.000000e-01
17     %i_sum = add i64 %i3, 1
18     br label %for_head
19
20  after_for:                                ; preds = %for_head
21     %f_prod5 = fmul double %res2, 1.000000e+02
22     %fi_cast = fptosi double %f_prod5 to i64
23     call void @exit(i64 %fi_cast)
24     unreachable
25 }
26
27 declare void @exit(i64)
```

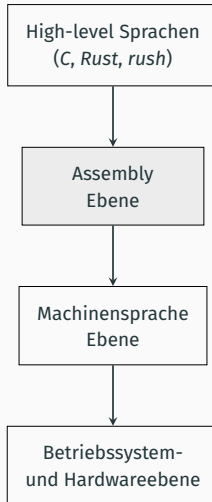
**Tabelle 4** – Vor- und Nachteile von LLVM

Vorteile	Nachteile
hoher Abstraktionsgrad	Aufwendige Installation der LLVM Libraries
Unabhängigkeit von der Zielmaschine	signifikante Größe der ausführbaren Datei
aggressive Optimierungsmaßnahmen	unvollständige Dokumentation von Inkwell
Ausgabeprogramm ca. 1,7 mal schneller (vgl. x86_64 Compiler)	Abhängigkeit von einer C++ Codebase

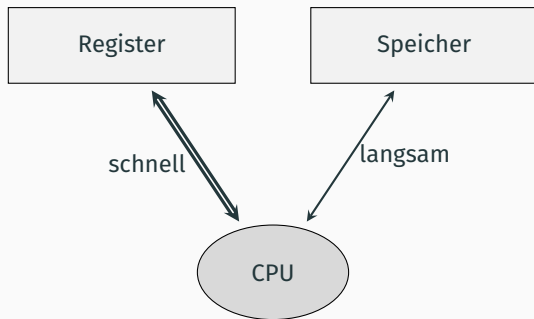
## **Kompilierung zu low-level Architekturen**

---

- Zielmaschine ist spezifisch
- Betriebssystem ist spezifisch
- Ziel: Lösen der Probleme vieler CISC Architekturen
- Hier: die Compiler generieren Assembly



**Abbildung 11** – Abstraktionsgrad von Assembly [Dan05b, p. 5–6].



**Abbildung 12** – Register, Speicher und die CPU [Dan05b, pp. 20–21].



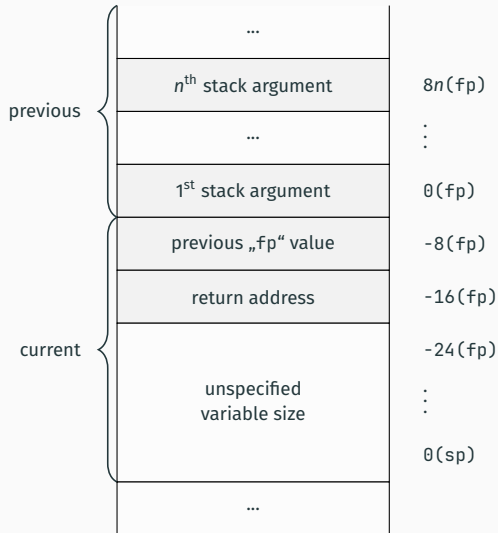
## Kompilierung zu RISC-V

---

- **Reduced instruction set computer (RISC)**
- Einfachheit und Erweiterbarkeit
- Forschungsprojekt der UC Berkeley
- Unterstützung durch: Google Microsoft, Samsung und IBM

**Tabelle 5** – Register der RISC-V Architektur [WA19, p. 155].

Register	Verwendung
zero	hardwired zero
ra	return address
sp	stack pointer
t0–t6	temporary storage
fp	frame pointer
a0, a1	function arguments, return values
a2–a7	function arguments
s1–s11	saved register
fa0, fa1	float arguments, return values
fa2–fa7	float arguments
fs0–fs11	float saved registers
ft0–ft11	float temporaries



**Abbildung 13** – Stack layout der RISC-V Architektur.

---

```
1 fn main() {  
2     let a = 1;  
3     let b = 2;  
4     exit(a + b);  
5 }
```

---

# Beispiel: Ausgabe

```
1  .global _start
2
3  .section .text
4
5  _start:
6      call main..main
7      li a0, 0
8      call exit
9
10 main..main:
11     # begin prologue
12     addi sp, sp, -32
13     sd fp, 24(sp)
14     sd ra, 16(sp)
15     addi fp, sp, 32
16     # end prologue
17     # begin body
18     li a0, 1
19     sd a0, -24(fp)    # let a = a0
20     li a0, 2
21     sd a0, -32(fp)    # let b = a0
22     ld a0, -24(fp)    # a
23     ld a1, -32(fp)    # b
24     add a0, a0, a1
25     call exit
26     # end body
27
28 epilogue_0:
29     ld fp, 24(sp)
30     ld ra, 16(sp)
31     addi sp, sp, 32
32     ret
```

## Kompilierung zu x86\_64

---

- **TODO: @RubixDev Write this**



**Tabelle 6** – Fazit zu x64

Vorteile	Nachteile
Sehr neu und modern	geringe Verbreitung
komplett open-source und Gemeinschaftlich verwaltet	eher experimentell
Sehr übersichtliche und simple Architektur	einige Operationen sind aufwendiger
Weniger Online-Ressourcen	sehr gute und übersichtliche Dokumentation

**Tabelle 7** – Fazit zu x64

Vorteile	Nachteile
höherer Abstraktionsgrad als RISC-V	Kompliziertere Übersetzung von z.B. Division und Multiplikation
Weite Verbreitung	Sehr alt und unübersichtlich
Viele Online-Ressourcen	Weniger übersichtliche Dokumentation

- Deutlich anspruchsvoller
- Signifikanter Lernaufwandt
- Detailliertes Verständnis notwendig
- Sehr fehleranfällig
- Benötigt keine Abhängigkeiten

## **Finale Anmerkungen & Fazit**

---

**TODO: @RubixDev @MikMuellerDev Write this**

## Literatur

---

- [Lat02] Chris Lattner. „LLVM: An Infrastructure for Multi-Stage Optimization“. Magisterarb. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, Dez. 2002.
- [Dan05a] Sivarama P Dandamudi. *Guide to RISC processors*. Ottawa, Canada: Springer International Publishing, Feb. 2005. ISBN: 0-387-21017-2.
- [Dan05b] Sivarama P. Dandamudi. *Introduction to Assembly Language Programming: For Pentium and RISC Processors*. 2. Aufl. Springer International Publishing, 2005. ISBN: 0-387-20636-1.
- [Wir05] Niklaus Wirth. *Compiler Construction*. Zürich, 2005. ISBN: 0-201-40353-6.
- [WA19] Andrew Waterman und Krste Asanović. „The RISC-V Instruction Set Manual Volume I: Unprivileged ISA“. In: (Dez. 2019). Hrsg. von Andrew Waterman, Krste Asanović und Sive Inc. URL: <https://riscv.org/technical/specifications/>.
- [Led20] Jim Ledin. *Modern Computer Architecture and Organization*. Birmingham, UK: Packt Publishing, Apr. 2020. ISBN: 978-1-83898-439-7.
- [Hsu21] Min-Yih Hsu. *LLVM Techniques, Tips, and Best Practices Clang and Middle-End Libraries*. Birmingham, UK: Packt Publishing, Apr. 2021. ISBN: 978-1-83882-495-2.
- [McN21] Timothy Samuel McNamara. *Rust in Action*. In Action. New York, NY: Manning Publications, Aug. 2021. ISBN: 978-1-61729-455-6.
- [Goo23] Google Fonts. *Google Fonts Icons*. Mai 2023. URL: <https://fonts.google.com/icons>.