

# ERA Tutorium 2

Leopold  
Carlo

Termin:

- Mittwoch 10:00
- Donnerstag 12:00

Raum:

- 03.13.010
- 00.08.059

Zulip:

- ERA Tutorium - Mi - 1000 - 1
- ERA Tutorium - Do - 1200 - 1

Lehrstuhl für  
Rechnerarchitektur & Parallele Systeme  
Prof. Dr. Martin Schulz  
Dominic Prinz  
Jakob Schäffeler

Lehrstuhl für  
Design Automation  
Prof. Dr.-Ing. Robert Wille  
Stefan Engels  
Benjamin Hien

# Einführung in die Rechnerarchitektur

Wintersemester 2025/2026

Übungsblatt 2: RISC-V Assembly

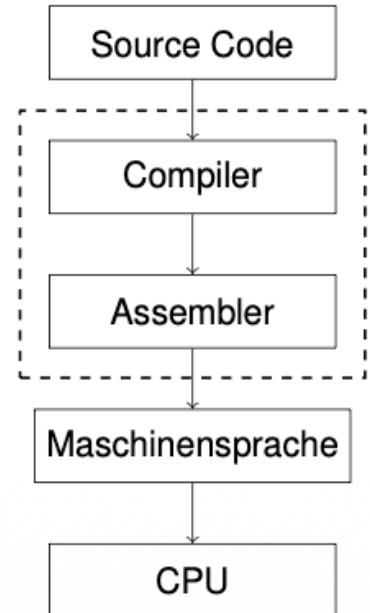
27.10.2025 – 31.10.2025

## 1 RISC-V Simulator Einrichtung

In ERA wird zur Simulation eines RISC-V Prozessors QtRvSim – ein Projekt der Tschechischen Technischen Universität – verwendet. Für die Einrichtung folgen Sie bitte den folgenden Schritten:

1. Laden Sie sich die passende Installationsdatei für Ihr Betriebssystem herunter: <https://github.com/cvut/qtrvsim/releases/tag/v0.9.8>
  - Ubuntu-User können auch folgendes PPA verwenden: ppa:qtrvsimteam/ppa
  - Windows-User benutzen die Datei mit mingw32 im Namenoder verwenden Sie die Web-Version (experimentell): <https://comparch.edu.cvut.cz/qtrvsim/app>
2. Belassen Sie die Einstellungen wie sie sind: „No pipeline no cache“ und klicken Sie auf „Example“.
3. In der oberen Hälfte sehen Sie die Register inklusive der zugehörigen Mnemonics und Werte.
4. Links sehen Sie die auszuführenden Instruktionen. Die Instruktionen eines Programms beginnen wie im RISC-V Ökosystem üblich, bei der Adresse 0x200.
5. Sie können mithilfe der Reiter „Core“ und „template.S“ zwischen der Prozessor- und Source Code-Ansicht wechseln.
6. Klicken Sie auf „Compile Source and update memory“ (blauer Pfeil nach unten), um den aktuell ausgewählten Source Code zu kompilieren und zu laden.
7. Anschließend können Sie das Programm mit dem Play-Button starten. Als Beispiel wird der Text „Hello world.“ rechts im Terminal ausgegeben.

- Code in einer Hochsprache (C, Java, ...) ist lediglich eine Abstraktion
- Compiler: Hochsprache → Assemblersprache
- Assembler: Assemblercode → Maschinensprache (1:1 Übersetzung)
- Maschinensprache ist plattformspezifisch!
- ISA: „Bedienungsanleitung“ einer CPU
- RISC vs. CISC



(Abbildung stark vereinfacht)

# 7 Referenzmaterial

| Name  | Register Number | Use                              |
|-------|-----------------|----------------------------------|
| zero  | x0              | Constant value 0                 |
| ra    | x1              | Return address                   |
| sp    | x2              | Stack pointer                    |
| gp    | x3              | Global pointer                   |
| tp    | x4              | Thread pointer                   |
| t0-2  | x5-7            | Temporary registers              |
| s0/fp | x8              | Saved register/Frame pointer     |
| s1    | x9              | Saved register                   |
| a0-1  | x10-11          | Function arguments/Return values |
| a2-7  | x12-17          | Function arguments               |
| s2-11 | x18-27          | Saved registers                  |
| t3-6  | x28-31          | Temporary registers              |

Abbildung 1: RISC-V 32-Bit Register

| op            | funct3 | funct7   | Type | Instruction          | Description                   | Operation  |
|---------------|--------|----------|------|----------------------|-------------------------------|--|
| 0000011 (3)   | 000    | -        | I    | lb rd, imm(rs1)      | load byte                     | $rd = \text{SignExt}([\text{Address}]_{7:0})$                      |
| 0000011 (3)   | 001    | -        | I    | lh rd, imm(rs1)      | load half                     | $rd = \text{SignExt}([\text{Address}]_{15:0})$                     |
| 0000011 (3)   | 010    | -        | I    | lw rd, imm(rs1)      | load word                     | $rd = [\text{Address}]_{31:0}$                                     |
| 0000011 (3)   | 100    | -        | I    | lbu rd, imm(rs1)     | load byte unsigned            | $rd = \text{ZeroExt}([\text{Address}]_{7:0})$                      |
| 0000011 (3)   | 101    | -        | I    | lhu rd, imm(rs1)     | load half unsigned            | $rd = \text{ZeroExt}([\text{Address}]_{15:0})$                     |
| 0010011 (19)  | 000    | -        | I    | addi rd, rs1, imm    | add immediate                 | $rd = rs1 + \text{SignExt}(imm)$                                   |
| 0010011 (19)  | 001    | 0000000* | I    | slli rd, rs1, uimm   | shift left logical immediate  | $rd = rs1 \ll< uimm$   |
| 0010011 (19)  | 010    | -        | I    | slti rd, rs1, imm    | set less than immediate       | $rd = (rs1 < \text{SignExt}(imm))$                                 |
| 0010011 (19)  | 011    | -        | I    | sltiu rd, rs1, imm   | set less than imm. unsigned   | $rd = (rs1 < \text{SignExt}(imm))$                                 |
| 0010011 (19)  | 100    | -        | I    | xori rd, rs1, imm    | xor immediate                 | $rd = rs1 ^ \text{SignExt}(imm)$                                   |
| 0010011 (19)  | 101    | 0000000* | I    | srlt rd, rs1, uimm   | shift right logical immediate | $rd = rs1 \gg uimm$  |
| 0010011 (19)  | 101    | 0100000* | I    | srai rd, rs1, uimm   | shift right arithmetic imm.   | $rd = rs1 \gg> uimm$   |
| 0010011 (19)  | 110    | -        | I    | ori rd, rs1, imm     | or immediate                  | $rd = rs1   \text{SignExt}(imm)$                                   |
| 0010011 (19)  | 111    | -        | I    | andi rd, rs1, imm    | and immediate                 | $rd = rs1 \& \text{SignExt}(imm)$                                  |
| 0010111 (23)  | -      | -        | U    | auipc rd, upimm      | add upper immediate to PC     | $rd = (upimm, 12'b0) + PC$   |
| 0100011 (35)  | 000    | -        | S    | sb rs2, imm(rs1)     | store byte                    | $[\text{Address}]_{7:0} = rs2_{7:0}$                               |
| 0100011 (35)  | 001    | -        | S    | sh rs2, imm(rs1)     | store half                    | $[\text{Address}]_{15:0} = rs2_{15:0}$                             |
| 0100011 (35)  | 010    | -        | S    | sw rs2, imm(rs1)     | store word                    | $[\text{Address}]_{31:0} = rs2$                                    |
| 0110011 (51)  | 000    | 0000000  | R    | add rd, rs1, rs2     | add                           | $rd = rs1 + rs2$   |
| 0110011 (51)  | 000    | 0100000  | R    | sub rd, rs1, rs2     | sub                           | $rd = rs1 - rs2$   |
| 0110011 (51)  | 001    | 0000000  | R    | sll rd, rs1, rs2     | shift left logical            | $rd = rs1 \ll< rs2_{4:0}$  |
| 0110011 (51)  | 010    | 0000000  | R    | slt rd, rs1, rs2     | set less than                 | $rd = (rs1 < rs2)$   |
| 0110011 (51)  | 011    | 0000000  | R    | sltu rd, rs1, rs2    | set less than unsigned        | $rd = (rs1 < rs2)$   |
| 0110011 (51)  | 100    | 0000000  | R    | xor rd, rs1, rs2     | xor                           | $rd = rs1 ^ rs2$   |
| 0110011 (51)  | 101    | 0000000  | R    | srl rd, rs1, rs2     | shift right logical           | $rd = rs1 \gg rs2_{4:0}$   |
| 0110011 (51)  | 101    | 0100000  | R    | sra rd, rs1, rs2     | shift right arithmetic        | $rd = rs1 \gg> rs2_{4:0}$  |
| 0110011 (51)  | 110    | 0000000  | R    | or rd, rs1, rs2      | or                            | $rd = rs1   rs2$   |
| 0110011 (51)  | 111    | 0000000  | R    | and rd, rs1, rs2     | and                           | $rd = rs1 \& rs2$  |
| 0110111 (55)  | -      | -        | U    | lui rd, upimm        | load upper immediate          | $rd = (upimm, 12'b0)$  |
| 1100011 (99)  | 000    | -        | B    | beq rs1, rs2, label  | branch if =                   | $\text{if } (rs1 == rs2) \text{ PC} = \text{BTA}$                  |
| 1100011 (99)  | 001    | -        | B    | bne rs1, rs2, label  | branch if ≠                   | $\text{if } (rs1 \neq rs2) \text{ PC} = \text{BTA}$                |
| 1100011 (99)  | 100    | -        | B    | blt rs1, rs2, label  | branch if <                   | $\text{if } (rs1 < rs2) \text{ PC} = \text{BTA}$                   |
| 1100011 (99)  | 101    | -        | B    | bge rs1, rs2, label  | branch if ≥                   | $\text{if } (rs1 \geq rs2) \text{ PC} = \text{BTA}$                |
| 1100011 (99)  | 110    | -        | B    | bltu rs1, rs2, label | branch if < unsigned          | $\text{if } (rs1 < rs2) \text{ PC} = \text{BTA}$                   |
| 1100011 (99)  | 111    | -        | B    | bgeu rs1, rs2, label | branch if ≥ unsigned          | $\text{if } (rs1 \geq rs2) \text{ PC} = \text{BTA}$                |
| 1100111 (103) | 000    | -        | I    | jalr rd, rs1, imm    | jump and link register        | $\text{PC} = rs1 + \text{SignExt}(imm), \text{rd} = \text{PC} + 4$ |
| 1101111 (111) | -      | -        | J    | jal rd, label        | jump and link                 | $\text{PC} = \text{JTA}, \text{rd} = \text{PC} + 4$                |

\*Encoded in instr<sub>31:25</sub>: the upper seven bits of the immediate field

Abbildung 2: RISC-V 32-Bit Integerbefehle

## Arithmetische Befehle

Signed Immediate:  $-2^{11}$  bis  $2^{11}-1$   
 -2048 bis 2047

### ► Addition/Subtraktion

- kein subi verfügbar
- negative Immediates bei addi möglich

Sign extension: 0011.1000.0010 → 0000.0000.0000.0000.0011.1000.0010  
 1011.1000.0010 → 1111.1111.1111.1111.1011.1000.0010

| op           | funct3 | funct7  | Type | Instruction       | Description   | Operation                        |
|--------------|--------|---------|------|-------------------|---------------|----------------------------------|
| 0010011 (19) | 000    | -       | I    | addi rd, rs1, imm | add immediate | $rd = rs1 + \text{SignExt}(imm)$ |
| 0110011 (51) | 000    | 0000000 | R    | add rd, rs1, rs2  | add           | $rd = rs1 + rs2$                 |
| 0110011 (51) | 000    | 0100000 | R    | sub rd, rs1, rs2  | sub           | $rd = rs1 - rs2$                 |

### ► Multiplikation/Division (3 Varianten für die oberen Bits)

| op           | funct3 | funct7  | Type | Instruction         | Description                     | Operation                  |
|--------------|--------|---------|------|---------------------|---------------------------------|----------------------------|
| 0110011 (51) | 000    | 0000001 | R    | mul rd, rs1, rs2    | multiply                        | $rd = (rs1 * rs2)_{31:0}$  |
| 0110011 (51) | 001    | 0000001 | R    | mulh rd, rs1, rs2   | multiply high signed signed     | $rd = (rs1 * rs2)_{63:32}$ |
| 0110011 (51) | 010    | 0000001 | R    | mulhsu rd, rs1, rs2 | multiply high signed unsigned   | $rd = (rs1 * rs2)_{63:32}$ |
| 0110011 (51) | 011    | 0000001 | R    | mulhu rd, rs1, rs2  | multiply high unsigned unsigned | $rd = (rs1 * rs2)_{63:32}$ |
| 0110011 (51) | 100    | 0000001 | R    | div rd, rs1, rs2    | divide (signed)                 | $rd = rs1 / rs2$           |
| 0110011 (51) | 101    | 0000001 | R    | divu rd, rs1, rs2   | divide unsigned                 | $rd = rs1 / rs2$           |
| 0110011 (51) | 110    | 0000001 | R    | rem rd, rs1, rs2    | remainder (signed)              | $rd = rs1 \% rs2$          |
| 0110011 (51) | 111    | 0000001 | R    | remu rd, rs1, rs2   | remainder unsigned              | $rd = rs1 \% rs2$          |

### ► Multiplikation von 32-Bit Zahlen kann maximal 64-Bit groß sein ( $2^{32} \cdot 2^{32} = 2^{64}$ )

## Logische Befehle

### ► Logisches UND, ODER und XOR verfügbar mit 12 Bit Immediates und Registern

| op           | funct3 | funct7  | Type | Instruction       | Description   | Operation                             |
|--------------|--------|---------|------|-------------------|---------------|---------------------------------------|
| 0010011 (19) | 100    | -       | I    | xori rd, rs1, imm | xor immediate | $rd = rs1 \wedge \text{SignExt}(imm)$ |
| 0010011 (19) | 110    | -       | I    | ori rd, rs1, imm  | or immediate  | $rd = rs1 \mid \text{SignExt}(imm)$   |
| 0010011 (19) | 111    | -       | I    | andi rd, rs1, imm | and immediate | $rd = rs1 \& \text{SignExt}(imm)$     |
| 0110011 (51) | 100    | 0000000 | R    | xor rd, rs1, rs2  | xor           | $rd = rs1 \wedge rs2$                 |
| 0110011 (51) | 110    | 0000000 | R    | or rd, rs1, rs2   | or            | $rd = rs1 \mid rs2$                   |
| 0110011 (51) | 111    | 0000000 | R    | and rd, rs1, rs2  | and           | $rd = rs1 \& rs2$                     |

## Schiebebefehle

### ► Logische Shifts um $n$

- Jedes Bit wird um  $n$  nach links/rechts geschoben
- Entstehende Bit-Positionen werden mit Nullen aufgefüllt

### ► Arithmetischer Rechtsshift um $n$

- Jedes Bit wird um  $n$  nach rechts geschoben
- Kopien des Vorzeichenbits an der Vorzeichenstelle eingeschoben

Linksshift:  $\text{sll rd, rs1, rs2}$        $0000.0111 \Rightarrow 0000.1110$       Zero-Extension

Rechtsshift:  $\text{srl rd, rs1, rs2}$        $1100.1010 \Rightarrow 0110.0101$       Zero-Extension  
arithmetisch  
 $\text{sra rd, rs1, rs2}$        $1100.1010 \Rightarrow 1110.0101$   
 $0100.1010 \Rightarrow 0010.0101$       Sign-Extension  
Sign-Extension

## Load/Store Befehle

- ▶ Load und Store in 8, 16 und 32 Bit Varianten
- ▶ Bei 8 und 16 Bit Load Varianten für Sign- und Zero-Extension
- ▶ Wert wird von Adresse `rs1 + imm` aus Hauptspeicher geladen

| op           | funct3 | funct7 | Type | Instruction      | Description        | Operation                                      |
|--------------|--------|--------|------|------------------|--------------------|--|
| 0000011 (3)  | 000    | -      | I    | lb rd, imm(rs1)  | load byte          | $rd = \text{SignExt}([\text{Address}]_{7:0})$  |
| 0000011 (3)  | 001    | -      | I    | lh rd, imm(rs1)  | load half          | $rd = \text{SignExt}([\text{Address}]_{15:0})$ |
| 0000011 (3)  | 010    | -      | I    | lw rd, imm(rs1)  | load word          | $rd = [\text{Address}]_{31:0}$                 |
| 0000011 (3)  | 100    | -      | I    | lbu rd, imm(rs1) | load byte unsigned | $rd = \text{ZeroExt}([\text{Address}]_{7:0})$  |
| 0000011 (3)  | 101    | -      | I    | lhu rd, imm(rs1) | load half unsigned | $rd = \text{ZeroExt}([\text{Address}]_{15:0})$ |
| 0100011 (35) | 000    | -      | S    | sb rs2, imm(rs1) | store byte         | $[\text{Address}]_{7:0} = rs2_{7:0}$           |
| 0100011 (35) | 001    | -      | S    | sh rs2, imm(rs1) | store half         | $[\text{Address}]_{15:0} = rs2_{15:0}$         |
| 0100011 (35) | 010    | -      | S    | sw rs2, imm(rs1) | store word         | $[\text{Address}]_{31:0} = rs2$                |

8. Weitere Beispiele finden Sie hier: <https://gitlab.fel.cvut.cz/b35apo/stud-support/-/tree/master/seminaries/qtrvsim>

9. Tipp: Probieren Sie sich hier aus!

## 2 Registerbenutzung und (symbolische) Konstanten

a) Machen Sie sich noch einmal mit den Registern und deren üblicher Verwendung vertraut.  
Die Register sind in Tabelle 1 abgebildet.

12

b) Alle für uns relevanten Befehle sind in Tabelle 2 gelistet. Nutzen Sie möglichst wenige Befehle, um folgende Aufgaben zu bewältigen.

- Laden Sie in das Register a0 den Wert 0 (sog. Nullen).

addi a0, x0, 0

li a0, 0 /

add a0, x0, x0/

xor a0, a0, a0

- Laden Sie die Konstante 0xF0000000 in das Register t1.

lui t1, 0xF0000 = 0xF0000 000

- Laden Sie die Konstante 0xABAD1DEA in das Register a3. Verwenden Sie keine Pseudobefehle.

0xABAD1DEA :

lui a0, 0xABAD1  
addi a0, a0, 0xDEA

$$0_{\text{x}} \text{DEA} = 3562 > 2047 \quad X$$
$$0_{\text{x}} \text{DEA} = -534 = -0_{\text{x}} 216 \quad \checkmark$$

$0_{\text{x}} \text{FFFFFDEA}$

$$0_{\text{x}} \text{ABAD1000} + 0_{\text{x}} \text{DEA} = 0_{\text{x}} \text{ABAD0DEA}$$

$$\begin{array}{r} 1010.1011.1010.1101.0001.0000.0000.0000 \\ + 1111.1111.1111.1111.1101.1110.1010 \\ \hline 1010.1011.1010.1101.0000.1101.1110.1010 \end{array}$$

$$0_{\text{x}} \text{ABAD1DEA} \doteq (?) + 0_{\text{x}} \text{FFFF.FDEA}$$

$$(?) = 0_{\text{x}} \text{ABAD.1DEA} - 0_{\text{x}} \text{FFFF.FDEA}$$

$$(?) = 0_{\text{x}} \text{ABAD.2000}$$

$$\begin{array}{r} 1010.1011.1010.1101.0001.1101.1110.1010 \\ - 1111.1111.1111.1111.1101.1110.1010 \\ \hline 1010.1011.1010.1101.0010.0000.0000.0000 \end{array}$$

=> lui a0, 0xABAD1DEA  
addi a0, a0, -534

### 3 Einfache Befehle

Übersetzen Sie die folgenden Terme in RISC-V Assembler. Verwenden Sie dabei nur Befehle aus Tabelle 2.

- $a_0 := a_0 - a_1$

*sub a0, a0, a1*

- $a_0 := a_0 + 2^{10}$

*1024*

*addi a0, a0, 0x400*

- $a_0 := a_0 + 2^{11}$

*$2^{11} = 1000.0000.0000$*

*addi a1, zero, 1*

*slli a1, a1, 11*

*add a0, a0, a1*

*$a1 = 1000.0000.0000$*

- $a_0 := a_0 \cdot 2 + 123$  (vorzeichenlose Multiplikation)

*slli a0, a0, 1*

*addi a0, a0, 123*

- $a_0 := a_0 \cdot 5$  (vorzeichenlose Multiplikation)

*= a0 · 4 + a0*

*ohne mul*

*slli a1, a0, 2*

*add a0, a0, a1*

### 4 Bitmasken und -Hacks

|                                    |  |
|------------------------------------|--|
| $X \text{ or } 1 = 1$              | z.B.:<br><br>$\begin{array}{r} XXXX.XXXX \\ \text{and } 0000.1111 \\ \hline 0000.XXXX \end{array}$ |
| $X \text{ and } 1 = X$             |  |
| $X \text{ and } 0 = 0$             |  |
| $X \text{ xor } 1 = \text{not } X$ |  |

Setzen Sie die folgenden Aufgaben um, indem Sie Bitmasken und -hacks verwenden.

- a)  $a_0 = \lfloor a_0 / 16 \rfloor$  (Ergebnis abgerundet, vorzeichenlose Division)

*srl i a0, a0, 4*

- b)  $a_0 = a_0 \bmod 256$  (vorzeichenlos)

*0111.1010.0001.1100.0000.1111.1010.1011*

*andi a0, a0, 255*

*and 0b1111.1111*

- c) Kopieren Sie die unteren 16 Bit von a1 in die unteren 16 Bit von a0 und die unteren 16 Bit von a2 in die oberen 16 Bit von a0.

`slli a1, a1, 16`

`slli a1, a1, 16`

`slli a2, a2, 16`

`or a0, a1, a2`

$a1: \underline{1000000} \underline{\text{xxxx}}$

$\text{or } a2: \underline{\text{xxxx}} \underline{000000}$

$= a0: \underline{\text{xxxx}} \underline{\text{xxxx}}$

- d) Setzen Sie das a1-te Bit in a0 auf Eins. Die restlichen Bits sollen unverändert bleiben.

`add a2, zero, 1`       $a2=1$

`sll a2, a2, a1`

`or a0, a0, a2`

$a0 = \text{xxxxxx} \text{xx}$

$\text{or } a2 = 00000100$

$\Rightarrow a0 = \text{xxxxxx} \text{1xx}$

- e) Bonus: Setzen Sie das a1-te Bit in a0 auf Null. Die restlichen Bits sollen unverändert bleiben.

`add a2, zero, 1`

`sll a2, a2, a1`

`or a0, a0, a2`

`xor a0, a0, a2`

$\text{xor } a2 = 00000100$

$\Rightarrow a0 = \text{xxxxxx} \text{0xx}$

## 5 Vorüberlegungen zur Hausaufgabe

- a) Was passiert, wenn man die in der Zentralübung erwähnte Formel zur Berechnung des Werts einer Binärzahl ( $a = \sum_{i=0}^n a_i \cdot 2^i$ ) auf negative Indizes erweitert?

$$\begin{array}{cccccc} 10^{-1} & 10^0 & \frac{1}{10} & \frac{1}{100} & \frac{1}{1000} \\ 25,231 & & & & \end{array}$$

$$\begin{array}{ccccccccc} 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & 2^{-1} & 2^{-2} & 2^{-3} & 2^{-4} \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array}$$

- b) Welchen Wertebereich kann man mit 8 binären Vorkommastellen (ohne Vorzeichen) und 8 binären Nachkommastellen (8.8) erreichen?

$$1111.1111,1111.1111$$

$$2^8 = 256 \quad 2^{-8} = \frac{1}{256} = 0,00390625$$

$$256 - 0,00390625 \quad [0,255\ldots99609375]$$

- c) Wie viele Bit bräuchte man mindestens, um Zahlen von 0 bis 100 darzustellen, sodass der Abstand zwischen zwei darstellbaren Werten maximal 0.005 beträgt?

$$2^{-x} = 0,005$$

$$x = -\lceil \log_2(0,005) \rceil = 8$$

$$2^x = 100 \quad \lceil \log_2 100 \rceil = 7$$

- d) Wie sieht die Addition bzw. Subtraktion in Festkommarechnung auf einem Blatt Papier aus? Was muss man beachten?

- e) Wie sieht die Multiplikation in Festkommarechnung auf einem Blatt Papier aus? Was muss man beachten?