# RISC-V

Implementation einer RISC-V VM in Python

# Inhalt

- Was ist RISC-V?
- Geschichte von RISC-V
- RISC-V Architektur
- Instruktionssatz
- Implementierung in Python

# Was ist RISC-V?

- Freie, offene Befehlssatzarchitektur (ISA)
- Entwickelt an der University of California, Berkeley
- Ursprünglich für Forschung und Lehre gedacht
- Zunehmende Bedeutung als Alternative zu proprietären ISAs
- RISC-V ist **keine** CPU-Architektur, sondern eine ISA

# Geschichte von RISC-V

- 2010 an der UC Berkeley von Andrew Waterman, Krste Asanović und David Patterson initiiert
- Ziel: eine offene ISA für Forschung und Lehre zu schaffen
- 2015 Gründung der RISC-V Foundation

# Ziele von RISC-V

- Vollständig offene ISA, frei verfügbar für Wissenschaft und Industrie
- Eignung für Hardware-Implementierung
- Kleine Basis-ISA mit optionalen Erweiterungen
- Unterstützung des IEEE-754 Floating-Point Standards (2008)
- 32- und 64-Bit-Adressraumvarianten
- Voll virtualisierbare ISA

# Modulares Design

- Basis-ISA definiert die grundlegenden Instruktionen (RV32I, RV64I)
- Erweiterungen für spezielle Anwendungsfälle
  - z.B. Multiplikation (M), Floating Point (F), Atomics (A)
- Erweiterungen sind optional und können je nach Anwendungsfall hinzugefügt werden

# RV32I

- Basis-ISA für 32-Bit RISC-V
- Definiert grundlegende Instruktionen:
  - Integer-Arithmetik
  - Logikoperationen
  - Sprung- und Kontrollinstruktionen
  - Laden und Speichern von Daten

# Register

- 32 Register (x0 bis x31)
- **Register x0 ist Hardwired auf 0** (hat immer den Wert 0)
  - Writes werden ignoriert
  - Das erlaubt es, einige Instruktionen als Pseudo-Instruktionen zu verwenden
  - z.B. `add x1, x0, x2` ist äquivalent zu `mv x1, x2`

# Assembly-Sprache

- Instruktionen sind in der Regel 3-Adress-Form
- Register werden explizit angegeben
- Beispiel:

```
_start:
  li x1, 42       # Lade 42 in Register x1
.loop:
  sub x1, x1, 1   # Subtrahiere 1 von x1
  bne x1, x0, .loop # Springe zurück zur .loop, wenn x1 != 0
  ...
```

# Instruktionsformat

RISC-V Instruktionen sind immer 32 Bit lang und in verschiedene encoding-Varianten unterteilt:

- R-Format: für Register-zu-Register-Operationen
- I-Format: für Immediate-Werte und Laden/Speichern
- S-Format: für Store-Operationen
- B-Format: für Sprung- und Verzweigungsoperationen
- U-Format: für große Immediate-Werte
- J-Format: für Sprungoperationen

# Beispiel: R-Format

```
add x1, x0, x2
```

```
0000000 00010 00000 000 00001 0110011
^funct7 ^rs2  ^rs1  ^f3 ^rd   ^opcode
```

- `opcode`, `funct3`, `funct7`: spezifizieren die genaue Operation
- `rd`: Zielregister (x1)
- `rs1`, `rs2`: Quellregister (x0, x2)

# Beispiel: I-Format

```
addi x1, x0, 10
```

```
000000001010 00000 000 00001 0010011
^imm[11:0]   ^rs1  ^f3 ^rd   ^opcode
```

- `opcode`, `f3`: spezifizieren die genaue Operation
- `rd`: Zielregister (x1)
- `rs1`: Quellregister (x0)
- `imm[11:0]`: Immediate-Wert (10)

# Instruktionssatz

RV32I definiert 47 Instruktionen aus mehreren Kategorien:

- Arithmetik: `add`, `sub`, `mul`, `div`, `rem`
- Logik: `and`, `or`, `xor`, `sll`, `srl`, `sra`
- Vergleich: `eq`, `ne`, `lt`, `le`, `gt`, `ge`
- Vergleich: `slt`, `sltu`
- Laden/Speichern: `lw`, `sw`
- Sprung: `beq`, `bne`, `jal`, `jalr`

# VM in Python

- Ziel: eine einfache RISC-V VM in Python zu implementieren
- Schritte:
  1. Definieren der Register und des Speichers
  2. Implementieren der Instruktionsdekodierung
  3. Ausführen der Instruktionen
  4. Implementieren von I/O-Funktionen

# Projektstruktur

```
src/
├── main.py                 # Entry point, lädt hex-Programme
├── vm.py                   # VM-Hauptklasse, instruction loop
├── state.py                # RVState: Register, Speicher, PC
├── instruction.py          # Instruction: Dekodierung der Instruktionen
├── instruction_impl.py     # InstructionImpl: Abstrakte Basisklasse
├── extension.py            # Extension: Abstrakte Basisklasse
├── nums.py                 # Typ-Aliase für numpy (u32, i32, etc.)
└── extensions/
    ├── rv32i.py            # RV32I Basis-Instruktionssatz
    ├── m.py                # M-Extension (mul, div, rem)
    └── ecall.py            # ECALL für I/O-Funktionen
```

# VM-Loop

Die VM-Loop führt nun folgende Schritte aus, bis `halt` gesetzt ist:

1. Lese die Instruktion am aktuellen Programmzähler (PC)
2. Dekodiere die Instruktion
3. Matche die Instruktion gegen die implementierten Instruktionsklassen
4. Führe die Instruktion aus

# State

Die class `RVState` repräsentiert den Zustand der VM:

```
class RVState:
    mem:  np.ndarray[u8]  # Memory
    rf:   np.ndarray[i32] # Register file
    pc:   u32             # Program counter
    halt: bool            # Halt flag
```

# Instruktionsdekodierung

```
class Instruction:
    instruction_word: u32

    def __init__(self, instruction_word: u32) -> None:
        self.instruction_word = instruction_word

    @property
    def opcode(self) -> u8:
        return u8(self.instruction_word & 0x7F)

    ...
```

# Instruktions-Matching

```python
class InstructionImpl(ABC):
    @abstractmethod
    def match(self, instruction: Instruction) -> bool:
        pass
    @abstractmethod
    def execute(self, state: RVState, instruction: Instruction) -> None:
        pass
```

# Beispiel: `add`

```
class Add(InstructionImpl):
    def match(self, instruction: Instruction) -> bool:
        return instruction.opcode == 0b0110011 \
           and instruction.funct3 == 0b000     \
           and instruction.funct7 == 0b0000000

    def execute(self, state: RVState, instruction: Instruction) -> None:
        raise NotImplementedError()
```

# Beispiel: `add`

```python
def execute(self, state: RVState, instruction: Instruction) -> None:
    # Extract the source and destination
    # registers from the instruction
    rd = instruction.rd
    rs1 = instruction.rs1
    rs2 = instruction.rs2

    # Execute the addition
    state.rf[rd] = state.rf[rs1] + state.rf[rs2]

    # Increment the program counter
    state.pc += 4
```

# ECALL

- Eigentlich für Systemcalls gedacht
- Wird in der VM für I/O verwendet
- Das Register `x17` bestimmt die Art des Aufrufs
- Beispiel: `li x17, 1` für Print, `li x17, 10` für Exit
- Bei Print wird der Wert von `x10` ausgegeben

# Testprogramm

```
_boot:
    li x10, 42          # Lade 42 in Register x10
.loop:
    li  x17, 1          # Setze x17 auf 1 (ECALL Print)
    ecall               # Führe Print aus
    addi x10, x10, -1   # Subtrahiere 1 von x10
    bne x10, x0, .loop  # Springe zurück zur .loop, wenn x10 != 0
    li  x17, 10         # Setze x17 auf 10 (ECALL Exit)
    ecall               # Führe Exit aus
```

In der VM ausgeführt, werden die Zahlen von 42 bis 1 ausgegeben.

# Zusammenfassung

- RISC-V ist eine offene ISA, die für Forschung und Lehre entwickelt wurde
- Modulares Design ermöglicht flexible Erweiterungen
- Basis-ISA relativ simpel zu implementieren
- Python eignet sich gut für die VM-Implementierung
- Projektstruktur ermöglicht einfache Erweiterungen und Anpassungen
- Code auf GitHub: https://github.com/levno-710/rvpy

# Quellen


- Waterman, A. & Asanović, K. (2025). *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. RISC-V Foundation. Document Version 20250508.
- Waterman, A. (2016). *Design of the RISC-V Instruction Set Architecture*. Ph.D. Dissertation, EECS Department, University of California, Berkeley.