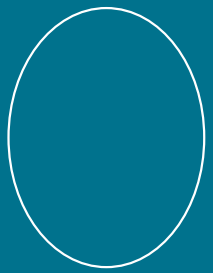
RISC-V CPU Emulator

Christian Krause christian.krause@stud.uni-heidelberg.de

Jugendforum Informatik



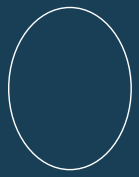


Table of Contents

Übersicht

Was passiert, bevor ein Programm ausgeführt wird?

1. C-Programm (`hello_world.c`)
2. Assembly Datei (`hello_world.s`)
3. Ausführbare Datei (ELF) (`hello_world`)

hello_world.c

```c
#include <stdio.h>
int main() {
  int a = 3;
  int b = 37;
  int c = a + b;
  c += 2;
  printf("Hello World! %d\n",c);
  return 0;
}
```

hello_world.c

```c
#include <stdio.h>
int main() {
  int a = 3;
  int b = 37;
  int c = a + b;
  c += 2;
  printf("Hello World! %d\n",c);
  return 0;
}
```

```
$ gcc -S hello_world.c
$ /opt/riscv/bin/riscv64-unknown-elf-gcc -S hello_world.c -O0 -mabi=lp64 -march=rv64i
```

~~hello_world.s~~

```
main:
...
  li  a4,3
  li  a5,37
  addw  a5,a4,a5
  addiw a5,a5,2
...
  call  printf
...
```

```c
#include <stdio.h>
int main() {
  int a = 3;
  int b = 37;
  int c = a + b;
  c += 2;
  printf("Hello World! %d\n",c);
  return 0;
}
```

Assembly Grundlagen

- CPU:
  - Register (32×64bit) (+ ggf. float Register)
  - Instructions
  - Program Counter (PC) (Position im Programm)
- RAM

Beispiele für Instructions

- `li $rd $imm`

  (Load immediate): Läd *imm* in *rd*

- `addi $rd $rs1 $imm`

  (Add immediate): Addiert *imm* auf *rs1* und schreibt das Ergebnis in *rd*

- `sub $rd $rs1 $rs2`

  Subtrahiert *rs2* von *rs1* und speichert das Ergebnis in *rd*

Speicher

- `ld $rd $offset($rsi)`

  (Load Doubleword): Läd den 64-bit Wert an *rs1* + *offset* aus dem Arbeitsspeicher in *rd*

- `sd $rs2 $offset($rs1)`

  (Store Douleword): Speichert den 64-bit Wert aus *rs2* an die Stelle *offset* + *rs1*

Speicher

Was wenn man nicht ganze 64bit aus dem Speicher laden möchte?

- `lb $rd $offset($rsi)`

TODO visualisierung Was passiert mit den 56 freien Bits?

1. **Zero Extending**: Bei **lbu** (Load Byte unsinged) werden alle oberen Bits auf null gesetzt
2. **Sign Extending**

Two's Complement

(TODO zwischenergebnisse selber überlegen und aufdecken)

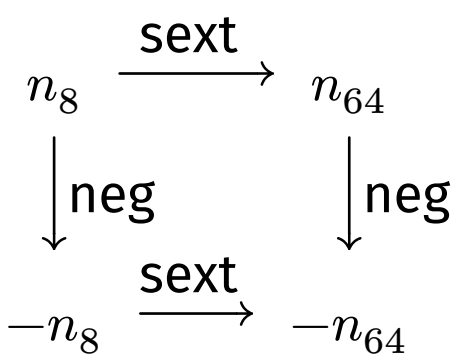
- Ziel: Binärdarstellung von ganzen (insbesondere negativen) Zahlen mit N Bits
- Natürliche Zahlen: $(n : \mathbb{N}) \mapsto \text{bin}(n) \quad 6 \mapsto 00000110$ (für $N = 8$)
- Negative Zahlen: $-n \mapsto \text{bin}(n)^{-1} + 1$
  - $6 \mapsto (00000110)^{-1} + 1 \mapsto 11111001 + 1 \mapsto 11111010$
- Warum?(TODO aufdecken)
  - 0 ist eindeutig
  - Addition funktioniert selbst mit negativen Zahlen
  - TODO gibt bestimmt noch mehr

Sign extending

TODO an der Tafel beispiel machen Wir haben eine Zahl n (als Binärzahl mit 8 Bits) und wollen diese in ein 64-Bit register Speichern (ohne den Wert zu ändern).

- **Zero Extending**: Wenn die Zahl negativ ist, ändert sich der Wert
- **Sign Extending**: Der neue Platz wird mit dem *MSB* (Most significant Bit) aufgefüllt
  - Bei Positiven Zahlen bleibt der Wert gleich (gleich wie Zero-Extending)
  - Bei Negativen Zahlen:

Sign extending



Es ist egal, ob man zuerst negiert oder Sign extended. (TODO Beispiel)

Speicher

Was wenn man nicht ganze 64bit aus dem Speicher laden möchte?

- `lb $rd $offset($rsi)`

TODO visualisierung Was passiert mit den 56 freien Bits?

1. **Zero Extending**: Bei **lbu** (Load Byte unsinged) werden alle oberen Bits auf null gesetzt
2. **Sign Extending**: Bei **lb** (Load Byte) wird der Wert auf 64-Bit Sign Extended und in *rd* geschrieben.

TODO aufdecken: Es gibt ähnliche Instructions für halfwords (16 Bit) und words (32 Bit).

TODO bei anderen Instructions (li addi etc) wird auch sign extended.

Lui und auipc

li kann nur einen 12-Bit Wert laden, was wenn man größere Zahlen möchte?

- `lui $rd $imm`

  (Load upper immediate): Setzt die unteren 12 Bits von *rd* auf null und die nächsten 20 Bits auf *imm*.

Control Flow Instructions

- `jal $rd $offset`

  (Jump and Link): Schreibe den Wert des Program Counters in *rd* und erhöhe den Program Counter um *offset*

- `beq $rs1 $rs2 $offset`

  (Branch if equal): Wenn *rs1* gleich *rs2* ist, dann wird der Program Counter um *offset* erhöht.

Hello World.asm

Jetzt verstehen wir was passiert! (abgesehen von call...)

```
main:
...
  li   a4,3
  li   a5,37
  addw   a5,a4,a5
  addiw a5,a5,2
...
  call   printf
...
```

```
#include <stdio.h>
int main() {
  int a = 3;
  int b = 37;
  int c = a + b;
  c += 2;
  printf("Hello World! %d\n",c);
  return 0;
}
```

ELF Dateien

Assembler konvertiert menschlich lesbaren Assembly Code in eine Ausführbare ELF Datei:

```
   └─λ file hello_world
hello_world: ELF 64-bit LSB executable, UCB RISC-V, soft-float ABI, version 1 (SYSV), statically linked, with debug_info, not stripped
```

- **file** command gibt informationen zu einer Datei aus

Elf Dateien

TODO leute raten lassen

```
hello_world: ELF 64-bit LSB executable, UCB RISC-V, soft-float ABI, version 1 (SYSV), statically linked, with debug_info, not stripped
```

- *hello_world* ist eine ausführbare (**executable**) **ELF** Datei
- **64-bit** Register
- **LSB**: Least Significant Bit (little endian)
- **RISC-V**
- **soft-float ABI**: keine Hardware für float operationen
- **statically linked**: Hängt nicht von anderen Bibliotheken ab

andere ELF Dateien

```
python3.14: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=ddf03afb83452757f6df249b7608c7df7b476e25,
for GNU/Linux 4.4.0, stripped
```

- **pie** (Position independent executable): Prgoramm kann an belibieger Speicheradresse geladen werden
- **x86-84**: Instruction set
- **dynamically linked**: Der **interpreter** läd Bibliotheken befor die Datei ausgeführt wird

ELF Dateien

TODO auf Wikiepdia artikel verlinken https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

- **Header**: Allgemeine Informationen zu der Datei, Entry point, Programm header Tabelle
- **Programm Headers**: Geben einen Teil der Elf Datei an, der an einen bestimmten Ort in den Arbeitsspeicher geladen werden soll

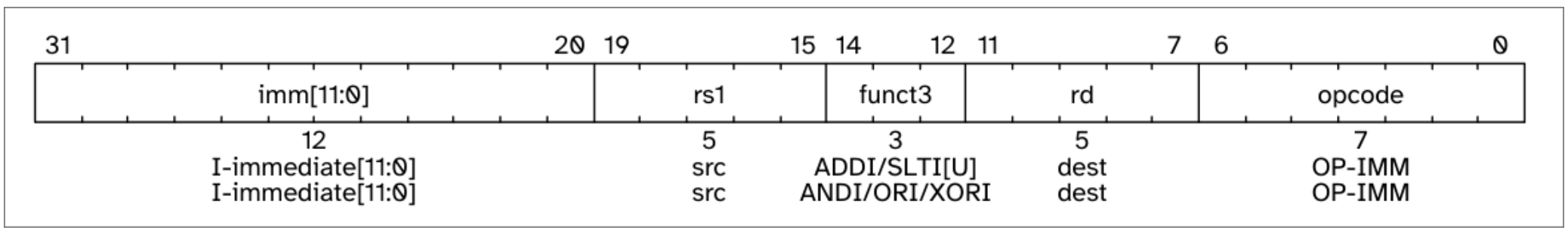
Instructions

Wie werden Instructions codiert?

- 4 Bytes (32-Bits)

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

Beispiel

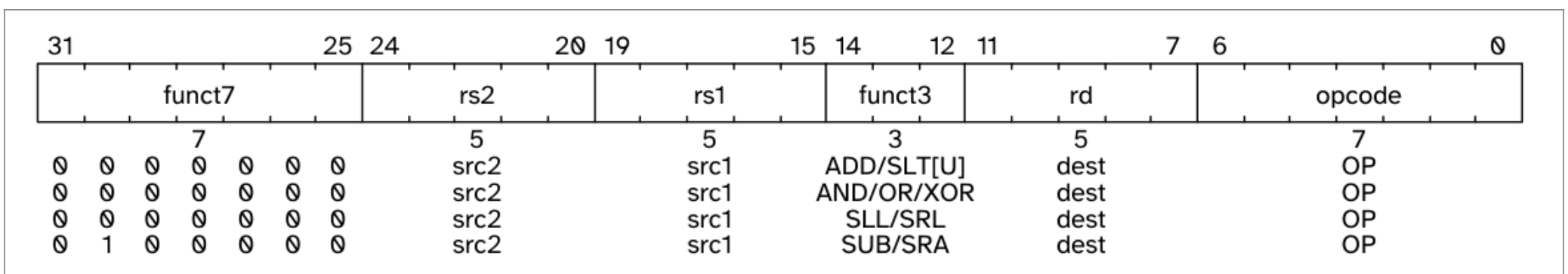



```
Assembly =   addi x3, x2, 42

  Binary =   0000 0010 1010 0001 0000 0001 1001 0011
```

Random Logo Sample

Beispiel



Bitwise Operationen

AND:

$$\begin{array}{rr} & 101101 \\ \& & 000111 \\ \hline = & 000101 \end{array}$$

OR:

$$\begin{array}{rr} & 101101 \\ | & 000111 \\ \hline = & 101111 \end{array}$$

Left Shift (8-Bit werte):

$$00110101 \ll 3 = 10101000$$

Logical Right Shift (8-Bit Werte):

$$10110101 \gg 3 = 00010110$$

Arithmetic Right Shift (8-Bit Werte):

$$10110101 \ggg 3 = 11110110$$

$$00110101 \ggg 3 = 00000110$$

Register

TODO erklären was abi heißt TODO diese slide später

Register	ABI Name	Beschreibung
x0	zero	Ist immer 0

TODO signed vs unsigned

~~a.out~~

TODO mehrere Wege sich a.out anzuschauen.

1. `file a.out`
2. `readelf a.out`

   (TODO welche flags)
3. `objdump a.out`

   (TODO welche flags)

a.out ausführen

1. einfach ausführen (output anschauen) (qemu)
2. gdb

Zum Debugging:

- qemu-linux-riscv64 installieren
- qemu gdb installieren
- schritt für schritt durch
- TODO das muss man nicht selber machen, sondern auf VM (Franz??)
- TODO kurzer reminder zu hexadezimalzahlen und binärzahlen
- TODO erklären wie ecalls funktionieren
- TODO Tabelle mit register namen
- TODO diese internetseite verlinken: https://luplab.gitlab.io/rvcodecjs/#q=0x4035d793&abi=false&isa=AUTO
- TODO bei shift instructions unterschied zwischen 64 und 32 beachten (auch bei encoding zwischen srli und srai)
- TODO rv64i multilib compilen

- TODO Riscv Musl toolchain erklären (bruacht -a extension für threading sachen)