Universität Bamberg

GRABS: Grundlagen der Rechnerarchitektur und Betriebssysteme

Vorlesung 6: Von Bits zu Prozessoren

Literatur:
Harris [2]
Kap. 7

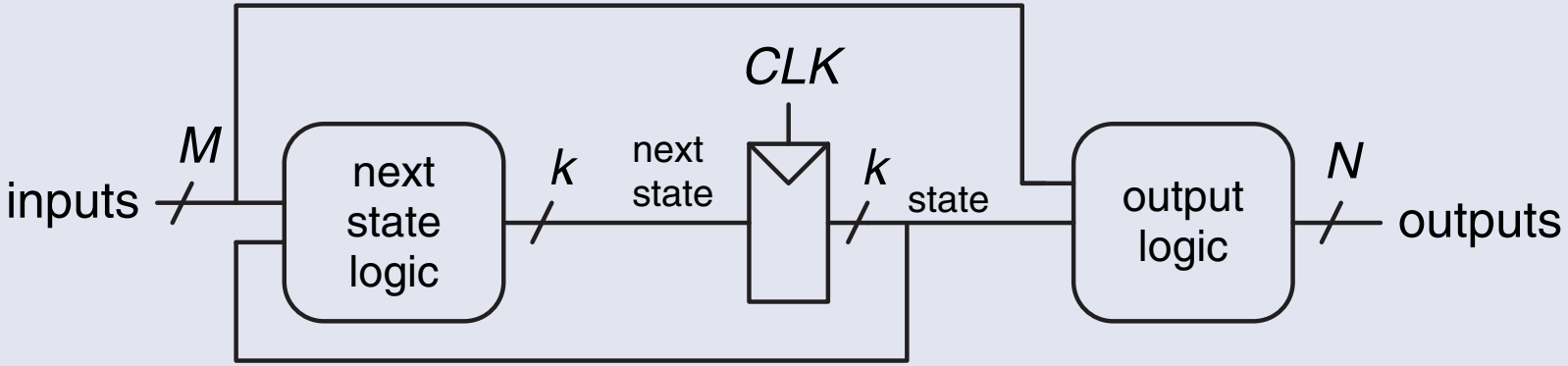
Michael Engel (michael.engel@uni-bamberg.de)
Lehrstuhl für Praktische Informatik, insbes. Systemnahe Programmierung
https://www.uni-bamberg.de/sysnap

Nochmal: Endliche Automaten

Wir haben Endliche Automaten gesehen – in den Beispielen haben EAs eine feste Funktion, z.B. die Ampelsteuerung

Komponenten:

- Register (aus Flipflops)
- Logik für den Folgezustand (rein kombinatorisch)



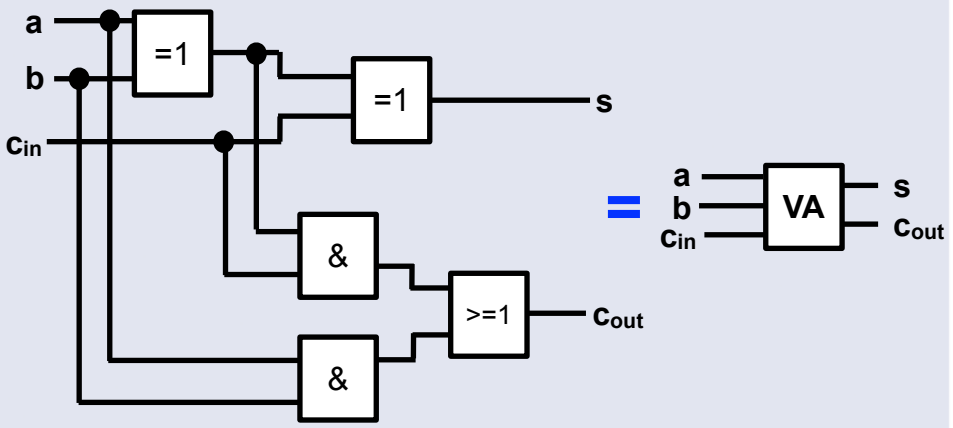
Mealy-Automat

Nochmal: Kombinatorische Logik

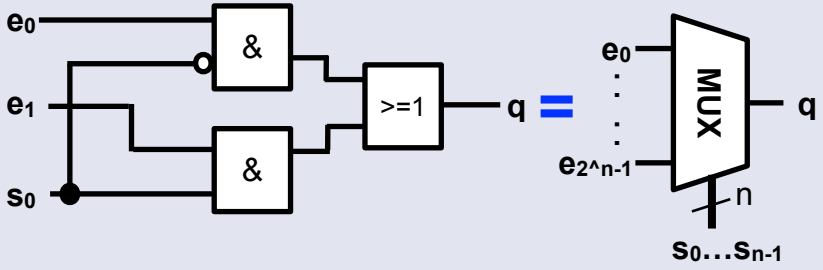
Hardware ändert ihr Verhalten nicht
- Erfüllt immer die in der Schaltung vorgesehene Funktion

Beispiele:

- **Volladdierer**
  - Eingang – 2 Bit + Übertrag
  - Ausgang – Summe + ausgehender Übertrag



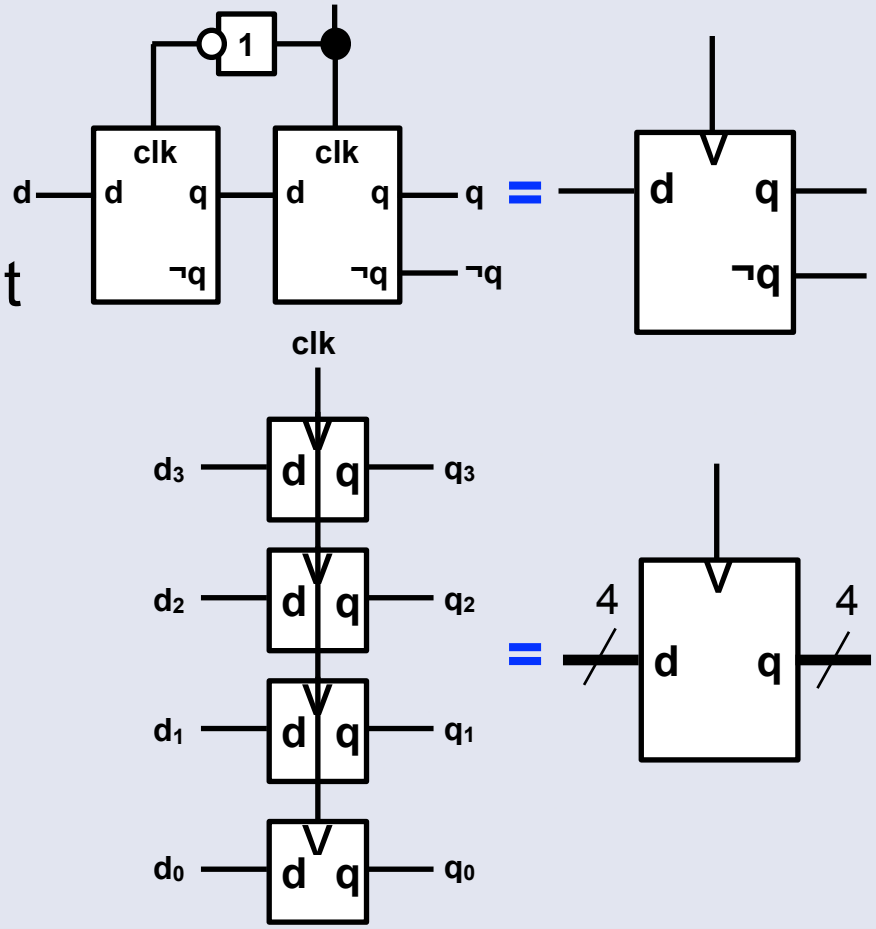
- **Multiplexer** (kurz: **Mux**)
  - Selektiert einen von 2^n Eingängen $\mathbf{e_i}$ abhängig von Binärzahl **n** an Selektions-eingang **s**



Nochmal: Register

Register verwenden Rückkopplung: **Speichern** von Daten

- **D-Flipflop**
  - Eingang – Clock und Daten
    - *flankengesteuert*
  - Ausgang – gespeichertes Bit + evtl. Inverses davon
- **Register**
  - Parallelschaltung mehrerer D-FFs
  - Hier: 4 Bit-Register mit Eingang $d_0 \ldots d_3$, Ausgang $q_0 \ldots q_3$

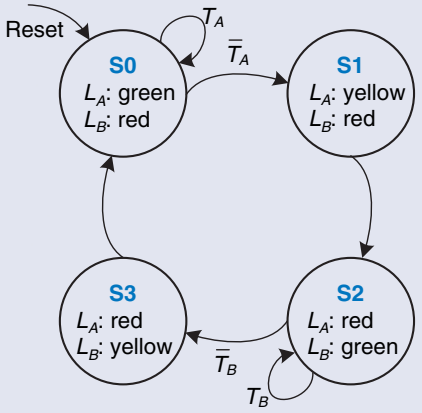


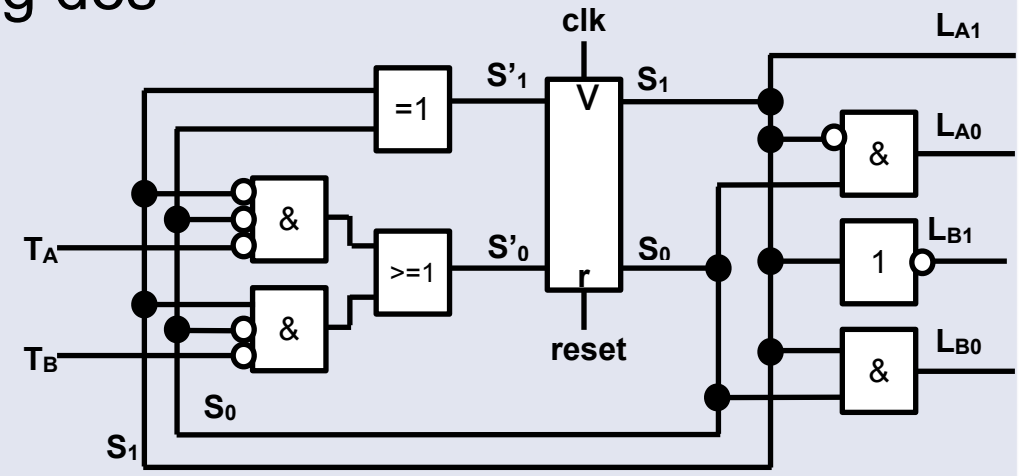
Nochmal: Beschreibung von EAs

Universität Bamberg

Wir haben gesehen:
Direktes Modellieren von EAs mit Logik sehr aufwändig

- **Automatisiertes Entwurfsverfahren**
  - **Spezifikation** des EA-Verhaltens als Zustandsübergangsgraph oder-tabelle
  - **Synthese** der Schaltung des Automaten aus der Spezifikation und der binären Codierung von Zuständen und I/O





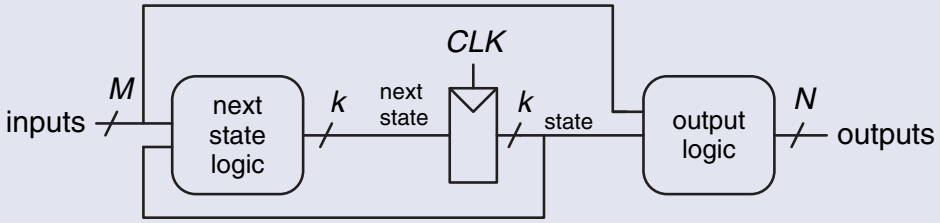
EAs vs. Prozessoren

Prozessoren sind noch komplexer als endliche Automaten
- Modellierung mit bisherigen Methoden zu umständlich!

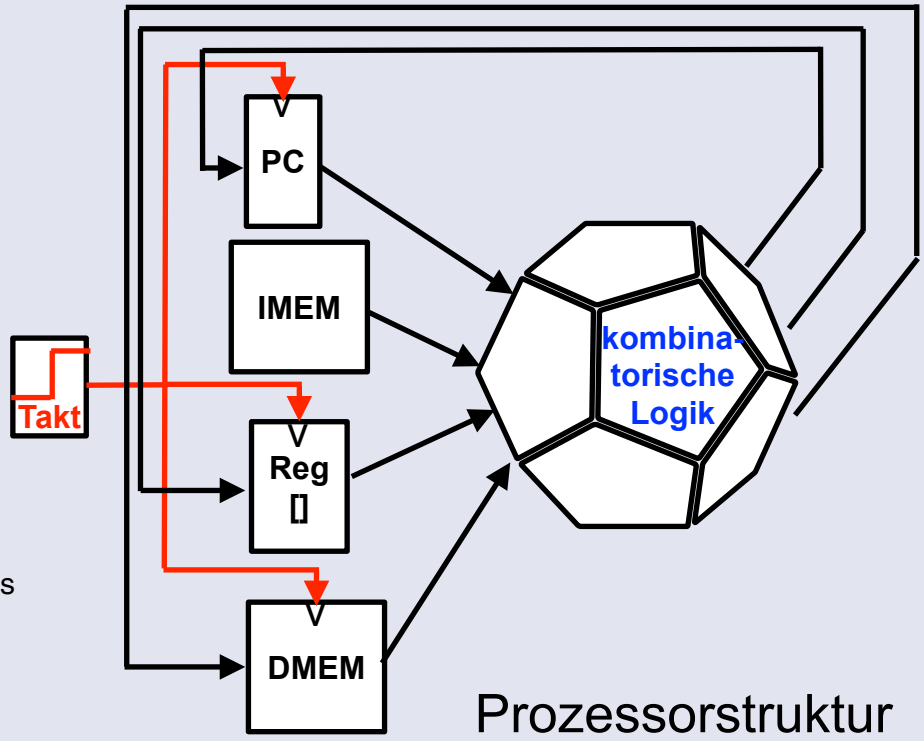
Zustand des Prozessors gespeichert in:
- Prozessorregister (PC, Reg[]), Datenspeicher (DMEM)

Eingaben des Prozessors:
- PC, Reg[], Instruktionsspeicher (IMEM), DMEM

Ausgaben des Prozessors:
- Prozessorregister, DMEM



Mealy-Automat



Prozessorstruktur

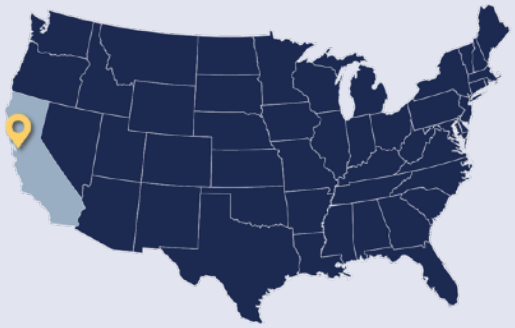
Die RISC-V Prozessorarchitektur

Es gibt viele verschiedene Prozessorarchitekturen
- x86, ARM, PowerPC, SPARC, IA64, MIPS, …

Wir verwenden die **RISC-V** Architektur
- Offene RISC (load/store)-Architektur

Entwickelt seit 2012 an der Berkeley University in Kalifornien, initial von David Patterson, Krste Asanović und Andrew Waterman [3]

Es existieren eine Reihe von Varianten von RISC-V, wir betrachten die grundlegende 32-Bit-Version RV32I (+Erweiterungen)



Idee: Abstraktion durch Interpretation

Universität Bamberg

Software

Programm in einer Hochsprache (z.B. C)

Compiler

Programm in Assembler-Sprache (z.B. RISC-V)

Assembler

Maschinensprache-Programm (z.B. RISC-V)

```
temp = v[k];
v[k] = v[k]+1;
v[k+1] = temp;
```

```
lw      a5,0(sp)
slli    a5,a5,2
addi    a5,a5,48
add     a5,a5,sp
```

```
0000 0000 0000 0001 0010 0111 1000 0011
0000 0000 0010 0111 1001 0111 1001 0011
0000 0011 0000 0111 1000 0111 1001 0011
0000 0000 0010 0111 1000 0111 1011 0011
```

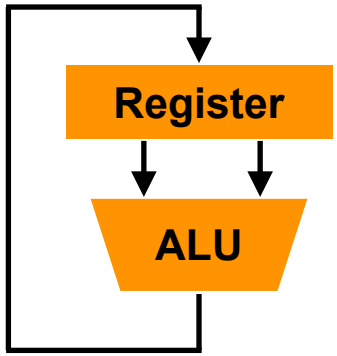
Hardware

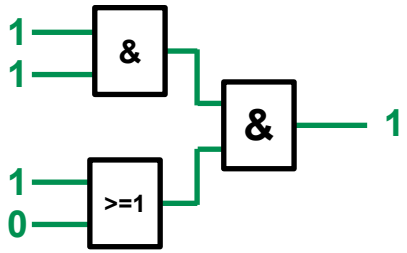
Maschinen-interpretation

Hardwarearchitektur-Diagramm (z.B. Blockschaltbilder)

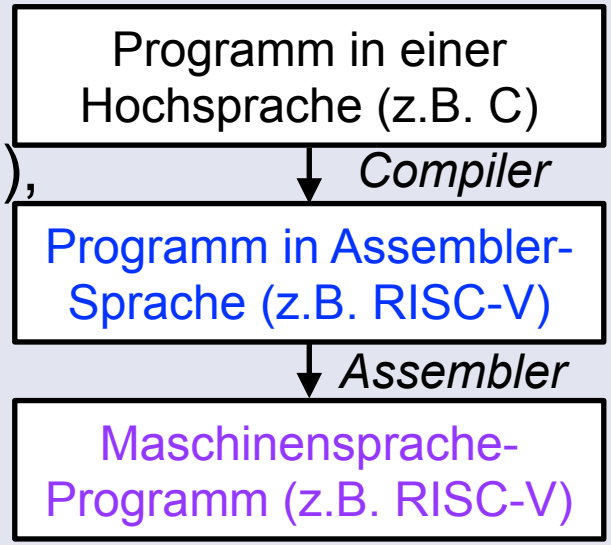
Architektur-implementierung

Logische Schaltungsbeschreibung (Schaltplan)





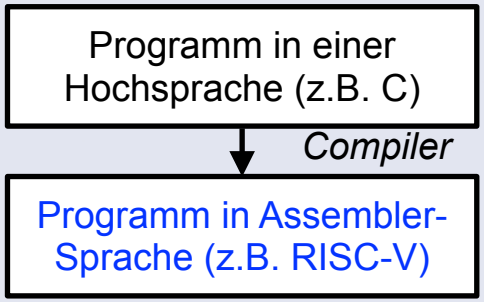
C → Assembler



Assembler ist eine **menschenlesbare Beschreibung** der Instruktionen (Befehle), die ein Prozessor ausführen kann

- Jede Prozessorarchitektur hat einen eigenen **Instruktionssatz**
- Entsprechend sind in Assembler geschriebene Programme ***nicht portabel***, d.h. nicht ohne viel Aufwand für eine andere Architektur übersetzbar!
- Auf Assemblerebene existieren (meist) nur ***einfache*** Operationen (Instruktionen oder Befehle genannt):
  - ***keine*** Datentypen, Schleifen, If-Bedingungen, Funktionen
  - ***aber*** oft Variablen- und Funktionsnamen
    - Diese werden in Binärwerte übersetzt

C → Assembler: Beispiele



Wir gehen im Folgenden von in der Sprache C geschriebenen Beispielen aus

- RISC-V ist eine **load/store-Architektur**
  - Instruktionen haben eine **feste Breite von 32 Bit**
  - Arithmetische und logische Instruktionen arbeiten auf **Registern**
    - Werte von Variablen werden zuvor aus Speicher geladen
    - Nach der Operation werden Werte wieder zurückgeschrieben
  - RISC-V hat **32 Register** mit Namen x0…x31
    - Der Wert von x0 ist immer = 0
  - Lade/Speicheroperationen arbeiten auf **festen Datenlängen**
    - Wort (32 Bit), Halbwort (16 Bit), Byte (8 Bit)
    - Arithmetische und logische Operationen arbeiten **immer auf Wortlänge der CPU** – bei uns also 32 Bit!

RISC-V Assemblerinstruktionen (1)

Universität Bamberg

Klasse	Instruktion	Fmt	RV32-Instruktion	Effekt
Laden	Lade Byte	I	`LB rd, rs1, imm`	`rd[7:0]⇐M[rs1+imm],rd[31:8]⇐rd[7]`
	Lade Halbwort	I	`LH rd, rs1, imm`	`rd[15:0]⇐M[rs1+imm],rd[31:16]⇐rd[15]`
	Lade Wort	I	`LW rd, rs1, imm`	`rd ⇐ M[rs1 + imm]`
	Lade Byte unsigned	I	`LBU rd, rs1, imm`	`rd[7:0]⇐M[rs1+imm],rd[31:8]⇐0`
	Lade Halbwort unsigned	I	`LHU rd, rs1, imm`	`rd[15:0]⇐M[rs1+imm],rd[31:16]⇐0`
Speichern	Speichere Byte	S	`SB rd, rs1, imm`	`M[rs1+imm] ⇐ rd[7:0]`
	Speichere Halbwort	S	`SH rd, rs1, imm`	`M[rs1+imm] ⇐ rd[15:0]`
	Speichere Wort	S	`SW rd, rs1, imm`	`M[rs1+imm] ⇐ rd`
Arithmetik	Addiere	R	`ADD rd, rs1, rs2`	`rd ⇐ rs1 + rs2`
	Addiere Konstante (*Immediate*)	I	`ADDI rd, rs1, imm`	`rd ⇐ rs1 + imm`
	Subtrahiere	R	`SUB rd, rs1, rs2`	`rd ⇐ rs1 - rs2`
	Lade Konstante in obere Bits	U	`LUI rd, imm`	`rd ⇐ imm << 12`
	Addiere Konstante zu PC	U	`AUIPC rd, imm`	`rd ⇐ pc + (imm << 12)`
Logik	XOR	R	`XOR rd, rs1, rs2`	`rd ⇐ rs1 ⊕ rs2`
	XOR mit Konstante	I	`XORI rd, rs1, imm`	`rd ⇐ rs1 ⊕ imm`
	OR	R	`OR rd, rs1, rs2`	`rd ⇐ rs1 ∨ rs2`
	OR mit Konstante	I	`ORI rd, rs1, imm`	`rd ⇐ rs1 ∨ imm`
	AND	R	`AND rd, rs1, rs2`	`rd ⇐ rs1 ∧ rs2`
	AND mit Konstante	I	`ANDI rd, rs1, imm`	`rd ⇐ rs1 ∧ imm`

rd, rs1, rs2 ist Kurzform für R[rd] usw.!

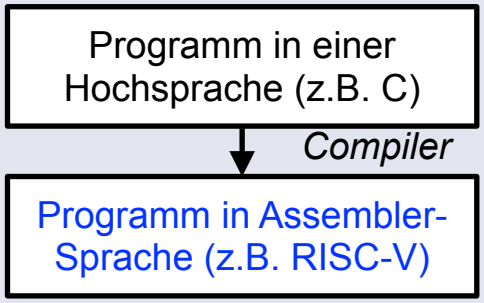
rd[a:b] bedeutet Bits a bis b (inklusive) von Register R[rd]

RISC-V Assemblerinstruktionen (2)

Universität Bamberg

Klasse	Instruktion	Fmt	RV32-Instruktion	Effekt
Schieben	Schiebe Bits links	R	`SLL rd, rs1, rs2`	`rd ⇐ rs1 << rs2 (mit 0 füllen)`
	Schiebe Bits links um Konst.	I	`SLLI rd, rs, shamt`	`rd ⇐ rs1 << imm (mit 0 füllen)`
	Schiebe Bits rechts	R	`SRL rd, rs1, rs2`	`rd ⇐ rs1 >> rs2 (mit 0 füllen)`
	Schiebe Bits rechts um Konst.	I	`SRLI rd, rs1, shamt`	`rd ⇐ rs1 >> imm (mit 0 füllen)`
	Schiebe Bits rechts arithm.	R	`SRA rd, rs1, rs2`	`rd ⇐ rs1 >> rs2 (Bit 31 füllen)`
	Schiebe Bits re. arithm. Kons.	I	`SRAI rd, rs1, shamt`	`rd ⇐ rs1 >> imm (Bit 31 füllen)`
Vergleiche	Setze falls <	R	`SLT rd, rs1, rs2`	`rd ⇐ if (rs1 < rs2) then 1 else 0`
	Setze falls < Konst.	I	`SLTI rd, rs1, imm`	`rd ⇐ if (rs1 < imm) then 1 else 0`
	Setze falls < unsigned	R	`SLTU rd, rs1, rs2`	`rd ⇐ if (rs1 < rs2) then 1 else 0`
	Setze falls < unsigned Konst.	I	`SLTIU rd, rs1, imm`	`rd ⇐ if (rs1 < imm) then 1 else 0`
Bed. Sprünge	Springe falls =	B	`BEQ rs1, rs2, imm`	`if (rs1 = rs2) pc ⇐ pc + (imm<<1)`
	Springe falls ≠	B	`BNE rs1, rs2, imm`	`if (rs1 ≠ rs2) pc ⇐ pc + (imm<<1)`
	Springe falls <	B	`BLT rs1, rs2, imm`	`if (rs1 < rs2) pc ⇐ pc + (imm<<1)`
	Springe falls ≧	B	`BLE rs1, rs2, imm`	`if (rs1 ≧ rs2) pc ⇐ pc + (imm<<1)`
	Springe falls < unsigned	B	`BLTU rs1, rs2, imm`	`if (rs1 < rs2) pc ⇐ pc + (imm<<1)`
	Springe falls ≧ unsigned	B	`BGEU rs1, rs2, imm`	`if (rs1 ≧ rs2) pc ⇐ pc + (imm<<1)`
Unbed. Sprünge	Jump&Link	J	`JAL rd, imm`	`rd ⇐ pc + 4, pc ⇐ pc + (imm<<1)`
	Jump&Link Register	J	`JALR rd, rs1, imm`	`rd ⇐ pc + 4, pc ⇐ rs1 + imm`

C → Assembler: Beispiel



a, b und foo sind **Label**, also symbolische Namen für Adressen von Variablen (a,b) und Code (foo)

```
            .data
a:          .dc.w 42
b:          .dc.w 23

            .text

foo:
            la   x1, a
            lw   x3, 0(x1)
            la   x2, b
            lw   x4, 0(x2)
            add  x10, x3, x4
            ret
```

`.data`, `.dc.w` und `.text` sind **Direktiven**, diese geben Hinweise zur Übersetzung nach Maschinensprache

Üblich: nur eine Instruktion pro Zeile

Instruktionsformat:

```
add x10, x3, x4
```

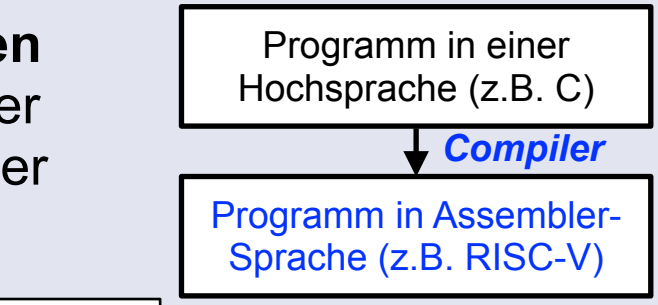
Instruktionsname oder "**Mnemonic**"

Ziel **Quelle(n)**

(optional) Parameter

C → Assembler: Variablen addieren

- Zugriff auf und Rechnen mit **globalen Variablen**
  - Liegen an festen Adressen im Hauptspeicher
  - Oft: **Adresse** von Variablen unter **Namen** der Variablen ansprechbar



RISC-V **Assembler**

```
    .data
a:  .dc.w 42
b:  .dc.w 23

    .text
foo:
    la  x1, a      # Lade Adresse von a → x1
    lw  x3, 0(x1)  # Lade a aus Speicher → x3
    la  x2, b      # Lade Adresse von b → x2
    lw  x4, 0(x2)  # Lade b aus Speicher → x4
    add x10, x3, x4  # Addiere x3+x4 → x10
    ret            # Rücksprung aus Funktion
```

C

```
int a = 42;
int b = 23;

int foo(void) {
  return a+b;
}
```

Ergebniswerte einer Funktion werden in x10 zurückgegeben

Kommentare werden mit '#' eingeleitet und laufen bis zum Zeilenende

…Moment mal?!?

RISC-V **Assembler**

```
    .data
a:  .dc.w 42
b:  .dc.w 23

    .text
foo:
    la  x1, a      # Lade Adresse von a → x1
    la  x2, b      # Lade Adresse von b → x2
    …
```

`la` **ist nicht in der Instruktionsliste!**

Programm in einer Hochsprache (z.B. C)

↓ ***Compiler***

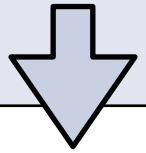
Programm in Assembler-Sprache (z.B. RISC-V)

- **Pseudoinstruktionen** machen uns das Programmieren leichter!
  - Wir laden eine 32-Bit-Konstante (Adresse) in Register `x1` bzw. `x2`
  - Instruktionen haben eine **feste Breite von 32 Bit (!)**
- **32 Bit Adresse lassen keinen Platz für Codierung der Art der Instruktion**
  - `la` muss **aus mehreren Operationen zusammengesetzt** werden

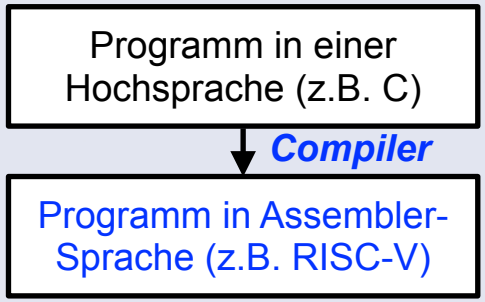
Pseudoinstruktion `la`

RISC-V **Assembler**

```
foo:
    la   x1, a      # Lade Adresse von x1
    ....
```



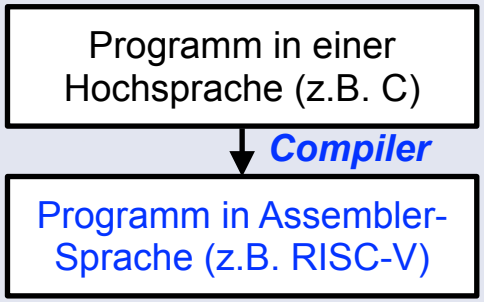
```
foo:
    lui x1, a[31:12]      # Lade Bits 31-12 der Adr von a → x1
    addi x1, x1, a[11:0]  # Add. Bits 11- 0 der Adr von a zu x1
    …
```



- `la` muss **aus mehreren Operationen zusammengesetzt** werden:
  - Zuerst die oberen 20 Bit der Adresse (31-12) laden
    - Dabei werden die unteren 12 Bits des Registers auf 0 gesetzt
  - Dann die restlichen 12 Bit der Adresse (11-0) zum Register addieren

Weitere Pseudoinstruktionen

Weitere Pseudoinstruktionen in RISC-V Assembler (vollständige Liste unter [5])



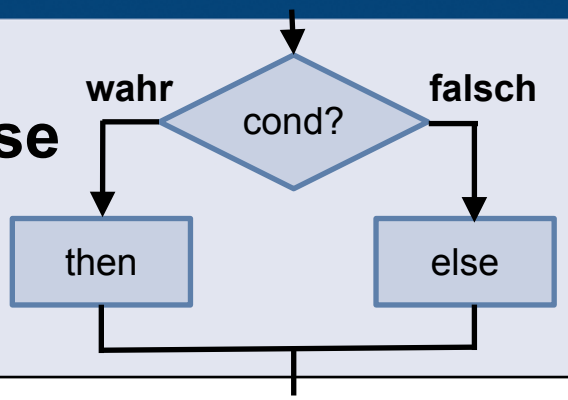
Bedeutung	Pseudoinstruktion	Implementiert durch
No Operation	`NOP`	`ADDI x0, x0 ,0`
Lade 32-Bit-Konstante	`LI rd, imm`	Kombination aus `LUI + ADDI`
Kopiere Daten ("Move")	`MOV rd, rs1`	`ADDI rd, rs1 ,0`
Invertierung	`NOT rd, rs1`	`XORI rd, rs1, -1` (1er-Komplement)
Negation	`NEG rd, rs1`	`SUB rd, x0, rs1` (2er-Komplement)
Springe falls >	`BGT rs1, rs2, imm`	`BLT rs2, rs1, imm`
Springe falls ≦	`BLE rs1, rs2, imm`	`BGE rs2, rs1, imm`
Springe unbedingt	`J imm`	`JAL x0, imm`
Springe unbedingt Register	`JR imm`	`JAL x1, imm`
Return	`RET`	`JALR x0, x1, x0`

Hochsprachenkonstrukte → Assembler

Universität Bamberg

Bedingte Abfragen: **if–else**

Führe *then* aus, falls cond
= wahr – sonst führe *else* aus



Programm in einer Hochsprache (z.B. C)

Compiler

Programm in Assembler-Sprache (z.B. RISC-V)

RISC-V **Assembler**

```
# Annahme:
# Variablen a,b,c sind in
# Registern x1,x2 bzw. x3 abgelegt

    ble  x1, x2, else # springe zu else
                      # …falls x1 <= x2 (!)
    li   x3, 42       # Lade 42 nach x3
    j    end          # Überspringe else-Teil

else:
    li   x3, 23       # Lade 23 nach x3
end:
    …                 # Hier geht's weiter
```

C

```
if (a>b) {
  c = 42;
} else {
  c = 23;
}
```

Die Sprungbedingung ist umgekehrt zum C-Code!
Warum?

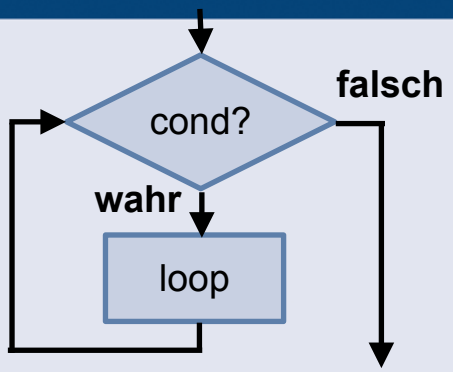
Übrigens: Mnemonics dürfen in Groß- oder Kleinbuchstaben geschrieben werden!

Hochsprachenkonstrukte → Assembler

Universität Bamberg

Schleife: **while**

Führe *loop* aus, solange
cond = wahr ist



Programm in einer Hochsprache (z.B. C)

Compiler

Programm in Assembler-Sprache (z.B. RISC-V)

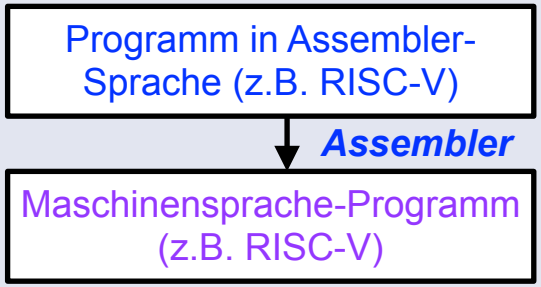
RISC-V **Assembler**

```
# Annahme:
# Variablen a, b sind in
# Registern x1 bzw. x2 abgelegt

    li   x3, 10        # Konst. für Vergleich
while:
    bge  x1, x3, end   # Springe zu end
                       # …falls x1 >= 10 (!)
    add  x1, x1, 1     # Addiere 1 zu x1
    add  x2, x2, 2     # Addiere 2 zu x2
    j    while         # Springe zum Anfang

end:
    …                  # Hier geht's weiter
```

C

```
while (a<10) {
  a = a+1;
  b = b+2;
}
```

Assembler → Maschinensprache



Erinnerung:
Assembler ist die **menschenlesbare** Darstellung von Maschineninstruktionen

Wie erhalten wir **Maschinensprache**, die **für die CPU verständliche** binäre Version von Assembler-Programmen?

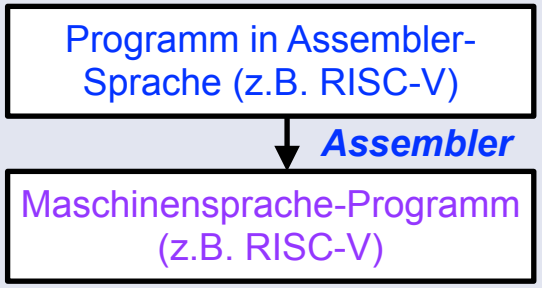
- Wir nutzen ein Werkzeug, das zur Verwirrung 😀 **ebenfalls Assembler genannt wird**

Für reguläre Instruktionen existiert eine 1:1-Abbildung

- Pseudoinstruktionen werden wie beschrieben ersetzt
- Befehle liegen hintereinander im Speicher
- Jede Maschineninstruktion ist 4 Byte lang
  - Beginnt an durch 4 teilbarer Adresse: ***naturally aligned***

Beispiel für Maschinensprache

Der Assembler übersetzt den Assembler-Quellcode und erzeugt eine **Binärdatei**



Dies sind die Instruktionen in Maschinensprache

RISC-V **Assembler**

```
    li   x3, 10
while:
    bge  x1, x3, end
    add  x1, x1, 1
    add  x2, x2, 2
    j    while
end:
    …
```

RISC-V **Maschinensprache**

```
00000000 <while-0x4>:
   0:  00a00193    li   x3,10

00000004 <while>:
   4:  0030d863    bge  x1,x1,14 <end>
   8:  00108093    addi x1,x1,1
   c:  00210113    addi x2,x2,2
  10:  ff5ff06f    j    4 <while>
00000014 <end>:
```

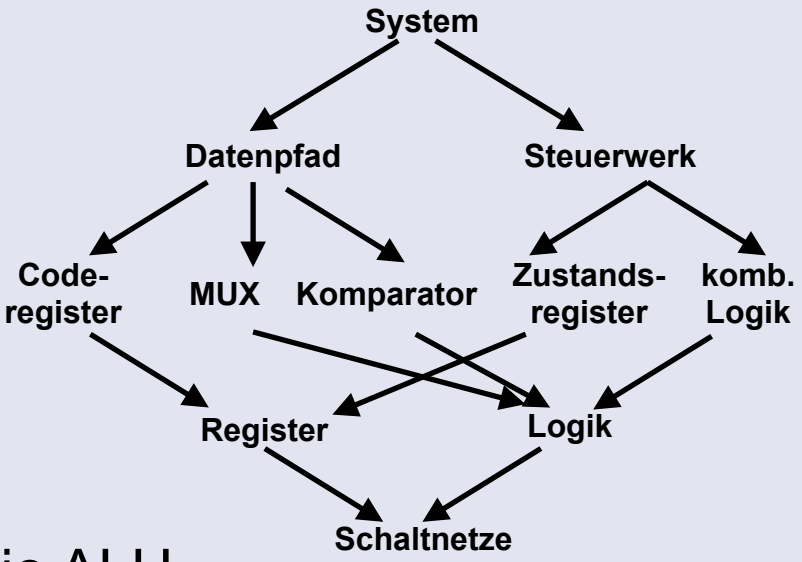
Adresse

```
00000000   93 01 a0 00 63 d8 30 00
00000008   93 80 10 00 13 01 21 00
00000010   6f f0 5f ff
```

Little Endian Bytereihenfolge!

Dies ist die Ausgabe eines *Disassemblers*, der aus den binären (in Hex-Darstellung) Operationen wieder menschenlesbaren Assemblercode erzeugt

Prozessorstruktur



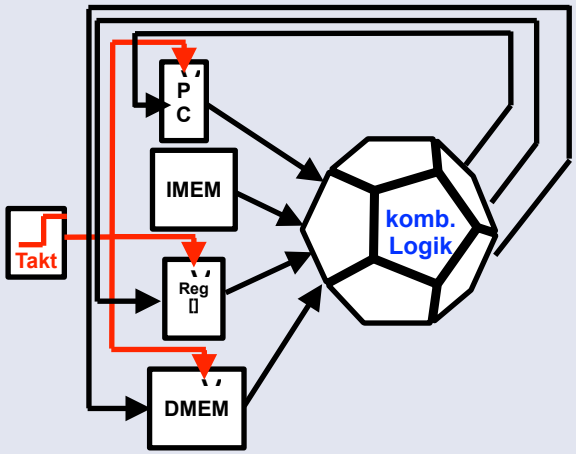
Bei RISC-Prozessoren:

- Befehlssatzarchitektur (Instruction Set Architecture, ISA) direkt in Hardware implementiert
- **Datenpfad:** Teil des Prozessors, der die Hardware zur Durchführung der Instruktionen enthält
  - Sammlung aus Rechenwerken, wie ALU oder Multiplizierer, die Datenverarbeitungsoperationen, Register und Busse
- **Steuerwerk:** Teil des Prozessors, der den Datenpfad für die jeweilige Instruktion konfiguriert
  - Erzeugt Steuersignale zur Konfiguration der Datenpfadkomponenten und des Routings im Datenpfad durch Multiplexer

Instruktionsausführung

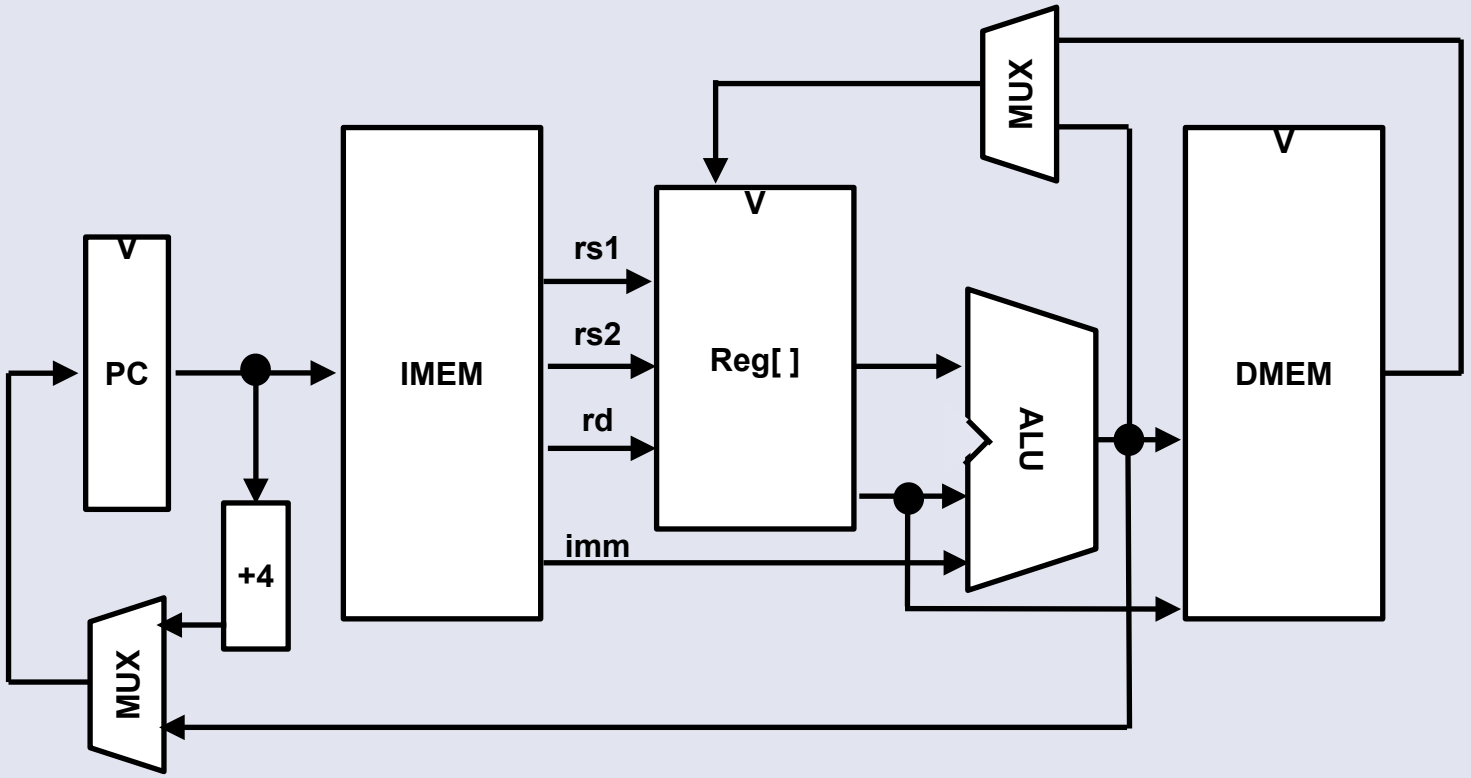
Welche Schritte müssen ausgeführt werden, um das Ergebnis einer RISC-V-Instruktion zu erhalten?

- …und was ist die korrekte Reihenfolge?



1. Hole die Instruktion: `add x10, x11, x12`
   - Aus Instruktionsspeicher IMEM
     - Adresse PC an IMEM, Ausgabe ist Instruktion
2. Welche Instruktion is es? `add`
3. Hole die Daten: lies Register `R[11], R[12]`
   - Aus Register (hier) oder Instruktion (*immediate*)
4. Führe die Operation aus: berechne `R[11]+R[12]`
5. Speichere Ergebnis: schreibe in `R[10]`

Datenpfad eines einfachen RISC-V



1. Instruktion holen → 2. Decodieren/ Register lesen → 3. Ausführen → 4. Speicher → 5. Register schreiben

Prozessorzustand des 32 Bit RISC-V

Jede Instruktion liest und aktualisiert diesen Zustand bei der Ausführung:

- **Register** (x0..x31)
  - *Register file* (or regfile) R hat 32 registers zu 32 Bit: R[0]…R[31]
    - Erstes gelesenes Register (*source*) in rs1-Feld der Instruktion
    - Zweites gelesenes Register in rs2-Feld der Instruktion
    - Schreibregister (Ziel, *destination*) in rd-Feld der Instruktion
    - Register x0 ist **immer** 0 (Schreibzugriffe zu R[0] ignoriert)
- **Programmzähler** (*program counter*, PC)
  - Speicher Adresse der aktuellen Instruktion
- **Speicher** (MEM)
  - Enthält Instruktionen und Daten in einem gemeinsamem 32-Bit Adressraum
  - Wir nutzen separate Speicher für Instruktionen (IMEM) und Daten (DMEM)
  - Load/store-Befehle greifen auf den Datenspeicher zu

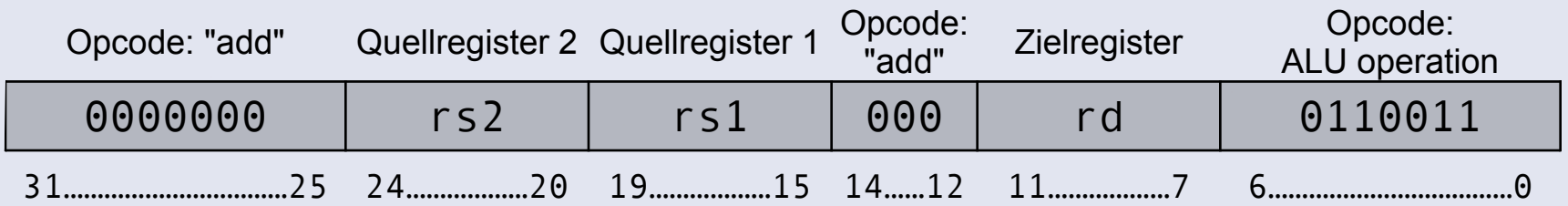
Was ist eine Instruktion?

Wie beim EA: Eingabe, codiert durch eine Menge von Bits

- RISC-V-Instruktionen (oder Befehle) sind 32 Bit breit und enthalten:
  - **Opcode** – gibt auszuführende Operation an
  - **Quellregister** – Eingabewerte für Operationen
  - **Zielregister** – Ausgabeziel für Ergebnis
  - **Direkte Parameter** – in Instruktion enthaltene Konstante

Beispiel: Addition – `add rd, rs1, rs2`

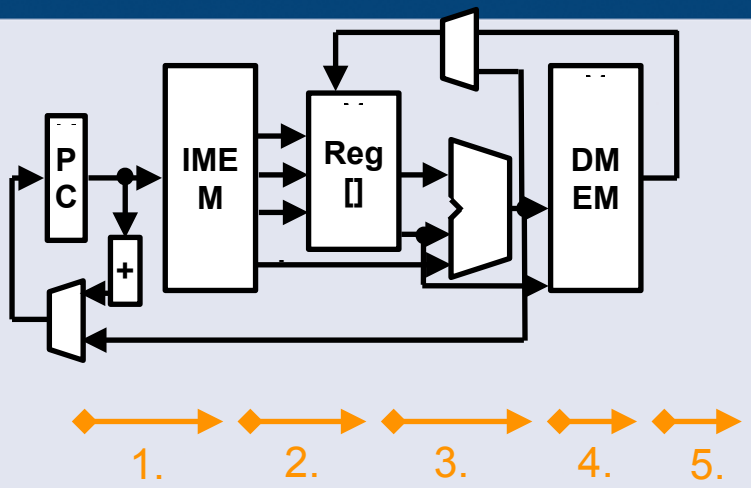
- Die Instruktion ändert Zustand des Prozessors an zwei Stellen:
  - `Reg[rd] = Reg[rs1] + Reg[rs2]`
  - `PC = PC + 4`



Binäre Codierung der `add`-Instruktion bei RISC-V

Ausführung der `add`-Instruktion


```
add x3, x1, x2
```

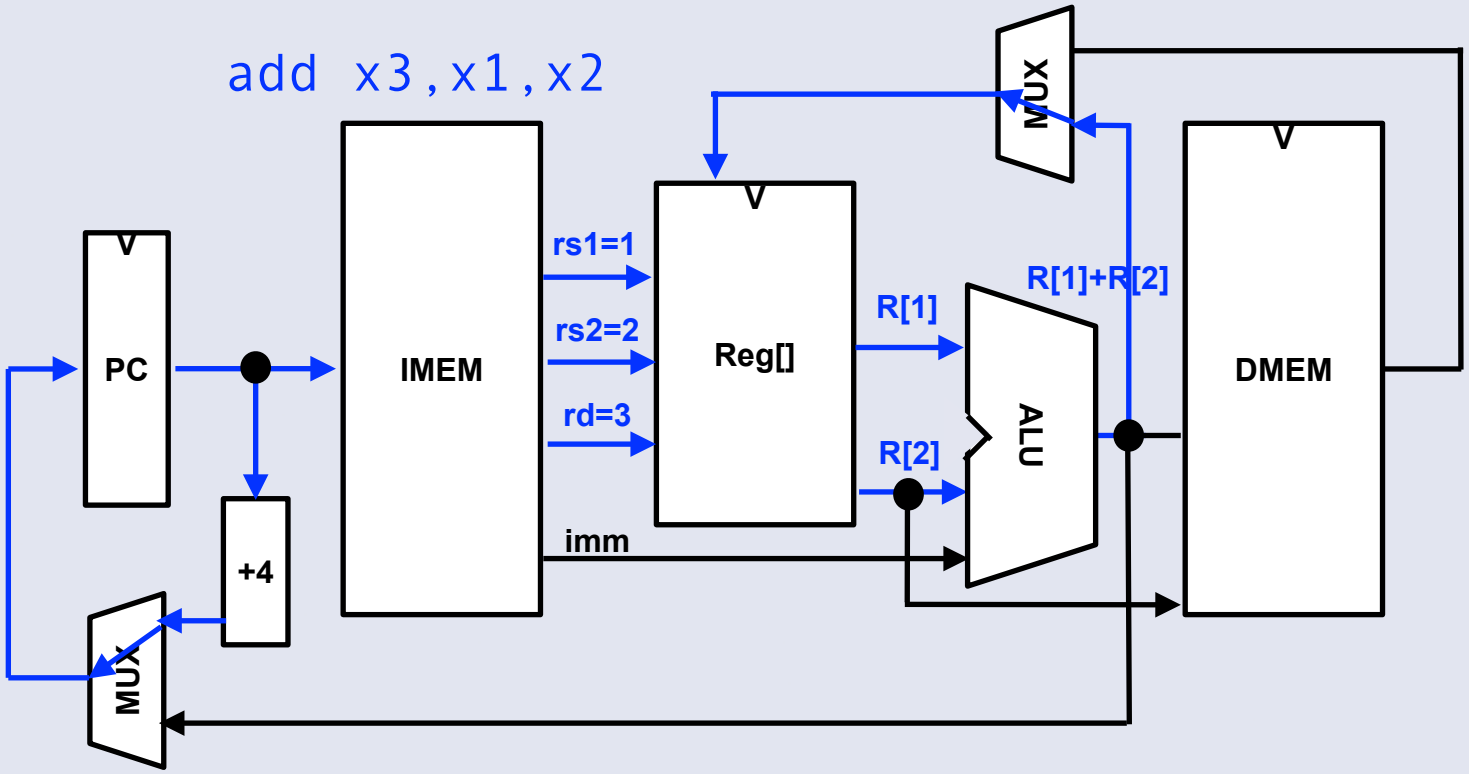


1. IF: Hole die Instruktion,
   inkrementiere PC
2. ID: Decodiere als `add` und
   lies Reg[1] und Reg[2]
3. EX: addiere die beiden von ID geholten Werte
4. MEM: ***ungenutzt*** (kein Speicherzugriff)
5. WB: schreibe Ergebnis von EX in Reg[3]

Opcode: "add"	Quellregister 2: x2	Quellregister 1: x1	Opcode: "add"	Zielregister: x3	Opcode: ALU operation
0000000	00010	00001	000	00011	0110011
31.....25	24.....20	19.....15	14.....12	11.....7	6.....0

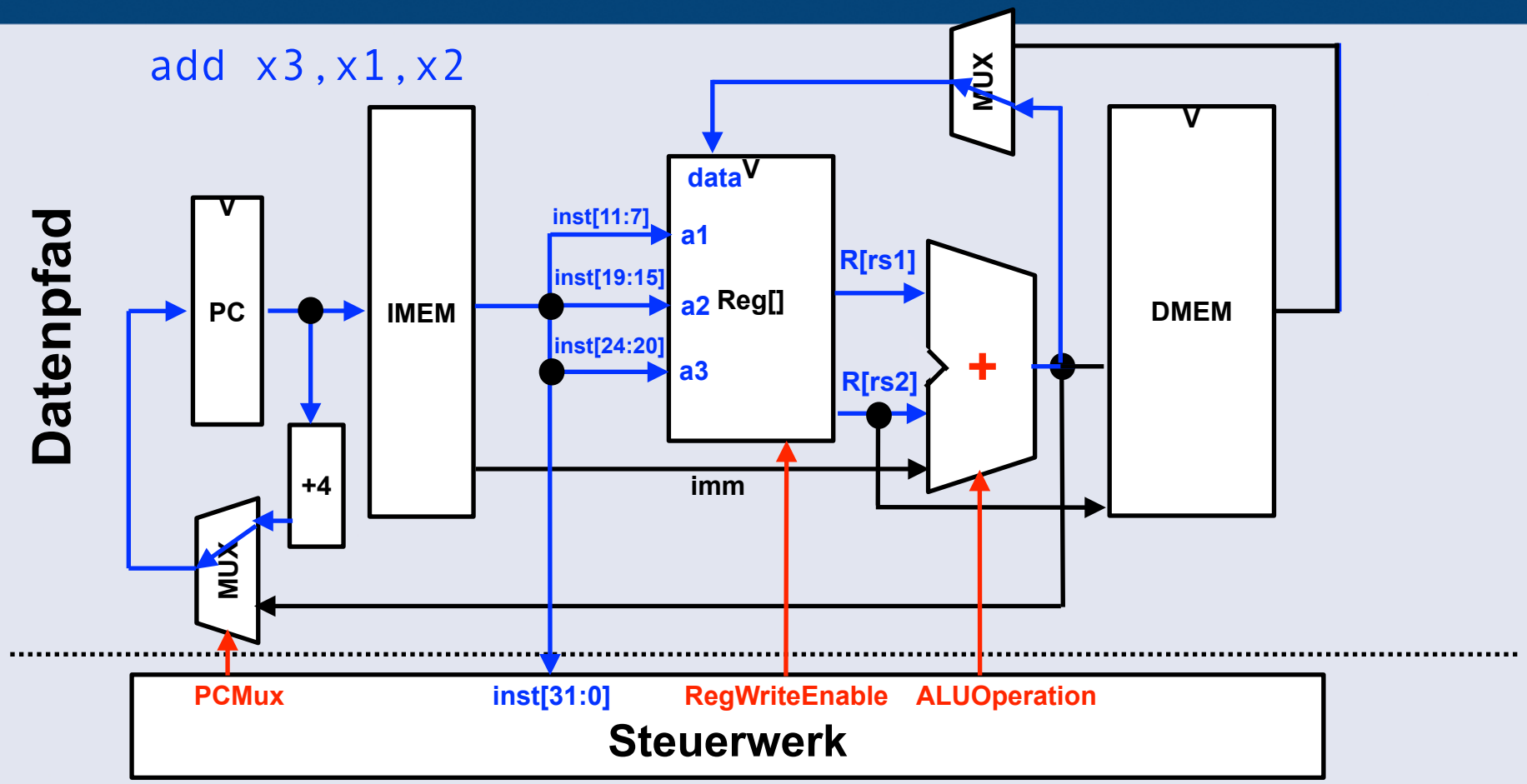
Binäre Codierung der `add x3,x1,x2`-Instruktion bei RISC-V

Add-Instruktion im Datenpfad



	Quellregister 2	Quellregister 1	Opcode: "add"	Zielregister	Opcode: ALU operation
0000000	00010	00001	000	00011	0110011
31.....25	24.....20	19.....15	14.....12	11.....7	6.....0
	=2	=1		=3	

…da fehlt was beim Prozessor?



Quellregister 2	Quellregister 1	Opcode	Zielregister	Opcode
0000000	00010	00001	000	00011	0110011
31……25	24……20	19……15	14……12	11……7	6……0

Sub im Datenpfad


```
sub x3, x1, x2    # r3 = r1-r2
```

Opcode: "add"	Quellregister 2	Quellregister 1	Opcode: "add"	Zielregister	Opcode: ALU operation	
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
31.....25	24.....20	19.....15	14.....12	11.....7	6.....0	

Fast identisch zu `add`:

- Operanden müssen jetzt in der ALU subtrahiert statt addiert werden
- Bit `inst[30]` unterscheidet zwischen Addition und Subtraktion

Datenpfad **ist identisch**:

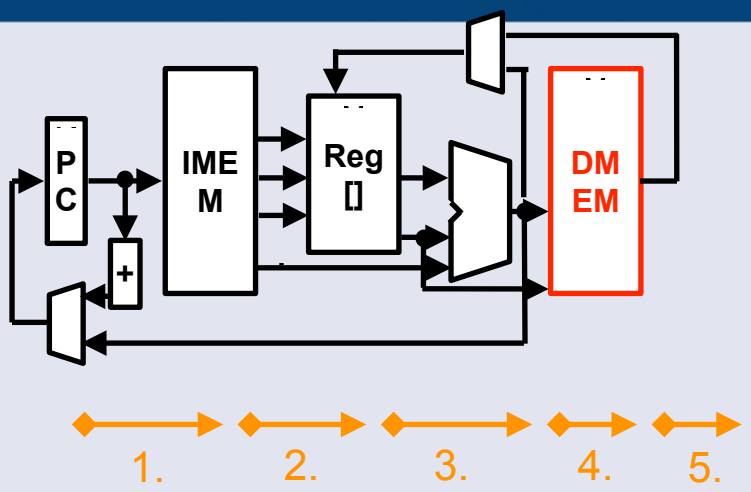
- Steuerwerk muss jetzt **ALUOperation** abhängig von Bit 30 setzen, wenn `inst[6:0] = 0110011` und `inst[14:12] = 000`
- Was ist mit den **restlichen Bits 31 und 29…25?**
  - "don't care"? – andere Kombinationen sind **reserviert**

Warum fünf Stufen?

Wäre eine andere Anzahl von Stufen möglich?

- Ja, andere Architekturen verwenden andere Anzahlen:
  - ARM Cortex M0: 2, M3: 3
  - Intel Pentium 4: 20 (!)

- Warum nutzt RISC-V fünf Stufen, wenn (die meisten) Instruktionen mindestens eine Stufe ungenutzt lassen?
  - Diese fünf Stufen sind die **Vereinigung aller Operationen**, die von allen Instruktionen benötigt werden



1. IF: Hole die Instruktion, inkrementiere PC
2. ID: Decodiere als `add` und lies R[1] und R[2]
3. EX: addiere die beiden von ID geholten Werte
4. MEM: ***ungenutzt*** (kein Speicherzugriff)
5. WB: schreibe Ergebnis von EX in R[3]

RISC-V Instruktionsformate

Die Instruktionen von RISC-V-Prozessoren sind **regulär** und in sechs einfache Formate aufgeteilt:

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

- R: Register
- I: Immediate
- S: Store
- B: Branch
- U: long Immediate
- J: Jump

Übersicht R-Format-Instruktionen

Register-Befehle: arithmetische und logische (ALU-)Befehle

- Zwei Quellregister, ein Zielregister

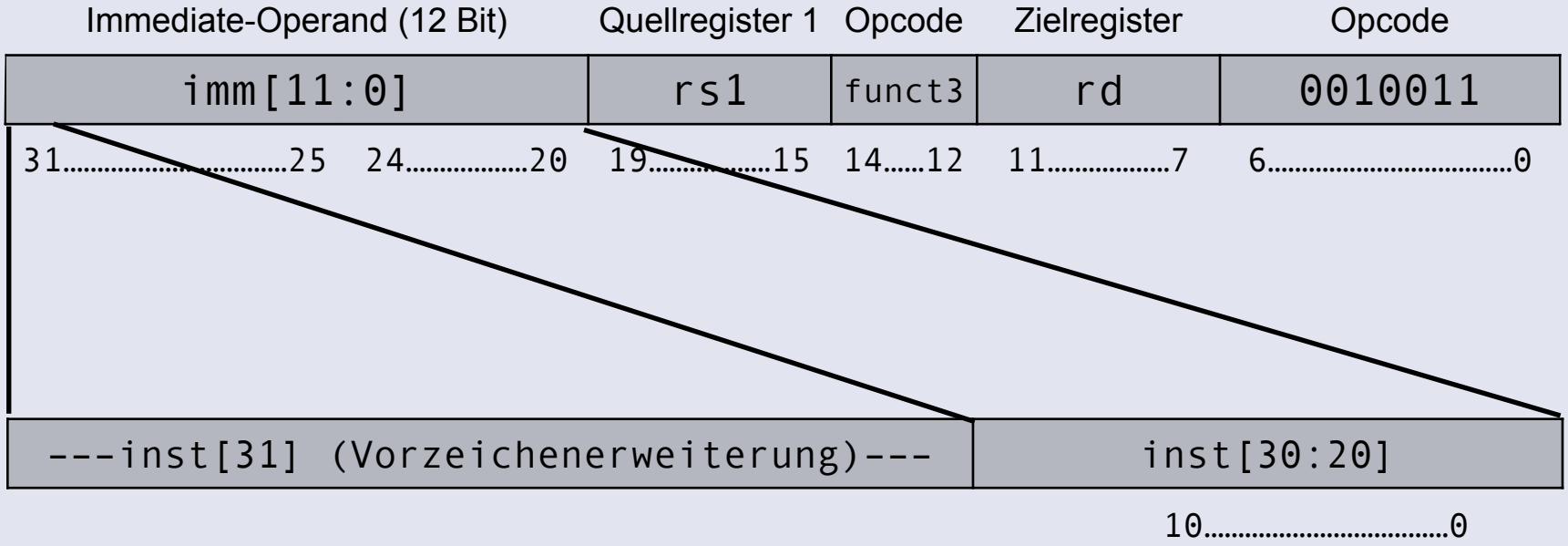
Opcode	Quellregister 2	Quellregister 1	Opcode	Zielregister	Opcode	
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	001	rd	0110011	sll
0000000	rs2	rs1	010	rd	0110011	slt
0000000	rs2	rs1	011	rd	0110011	sltu
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	101	rd	0110011	srl
0100000	rs2	rs1	101	rd	0110011	sra
0000000	rs2	rs1	110	rd	0110011	or
0000000	rs2	rs1	111	rd	0110011	and
31.....25	24.....20	19.....15	14.....12	11.....7	6.....0	

Immediate-Instruktionen

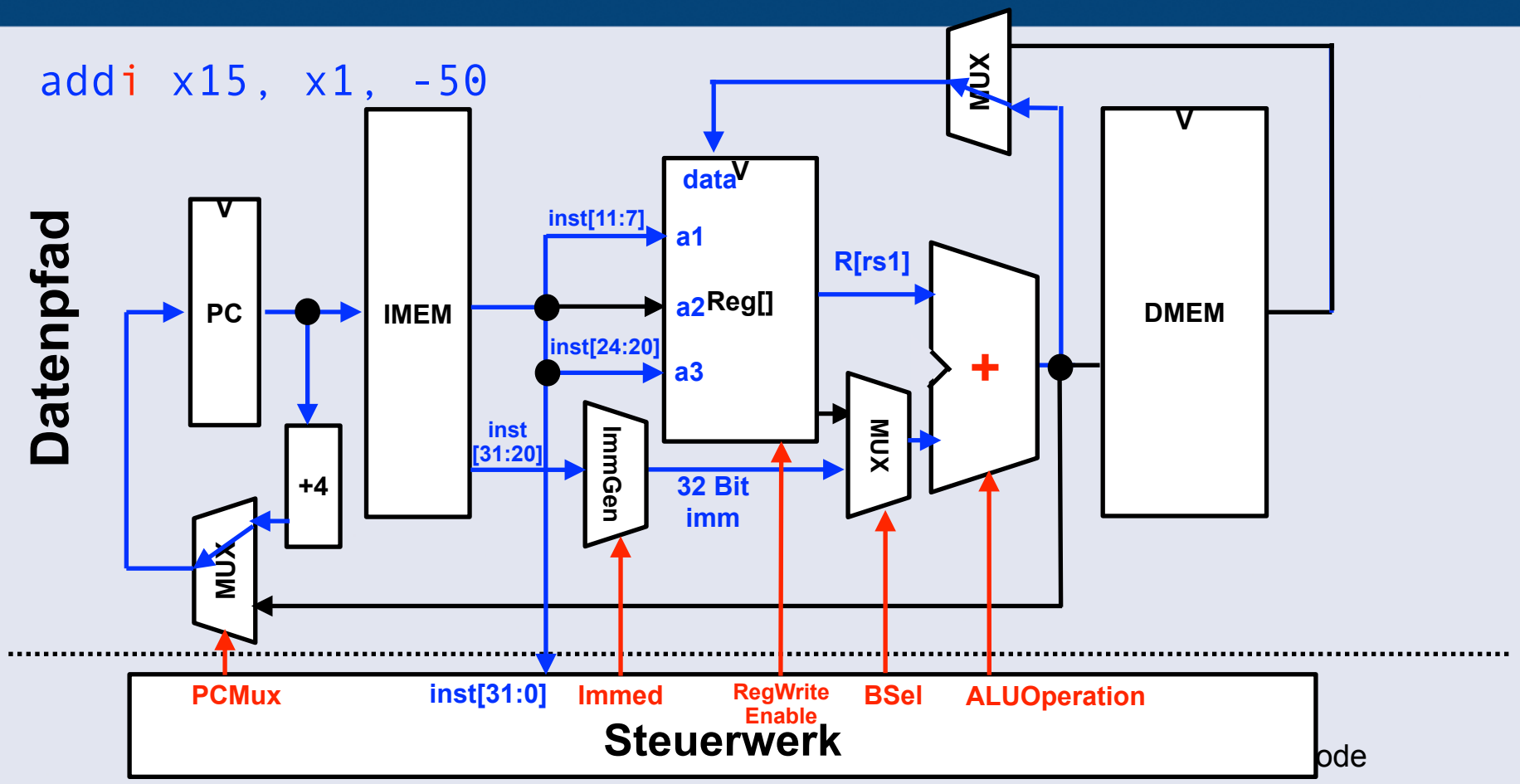

```
addi x15, x1, -50
```

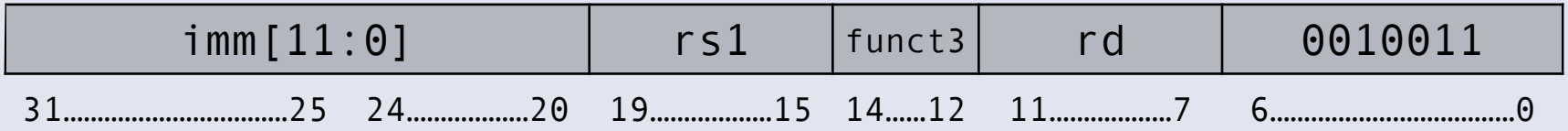
Zweiter Parameter ist jetzt kein Register, sondern direkt als Konstante in der Instruktion codiert (hier: -50)

- Nur 12 Bit für Konstante in Befehl verfügbar
- Wird auf 32 Bit **vorzeichenerweitert** (2er-Komplement)



Immediate-Instruktionen





Binäre Codierung der `addi`-Instruktion bei RISC-V

Übersicht I-Format-Instruktionen

Immediate-Befehle: enthalten eine 12 Bit-Konstante statt `rs2`

Immediate-Operand (12 Bit)		Quellregister 1	Opcode	Zielregister	Opcode	
imm[11:0]		rs1	000	rd	0010011	addi
imm[11:0]		rs1	010	rd	0010011	subi
imm[11:0]		rs1	011	rd	0010011	sltiu
imm[11:0]		rs1	100	rd	0010011	xori
imm[11:0]		rs1	110	rd	0010011	ori
imm[11:0]		rs1	111	rd	0010011	andi
0000000	shamt	rs1	001	rd	0010011	slli
0000000	shamt	rs1	101	rd	0010011	srli
0100000	shamt	rs1	101	rd	0010011	srai

31.....25 24.....20 19.....15 14.....12 11.....7 6.....0

Bit unterscheidet zwischen logischem und arithmetischem Shift

Mehr als 31 Bit schieben ergibt keinen Sinn auf 32 Bit RISC-V
Damit reichen 5 Bit für die Angabe der zu schiebenden Bits aus

Übersicht I-Format-Instruktionen: Loads

Load-Befehle sind **ebenfalls im I-Format** codiert!

Immediate-Operand (12 Bit)	Quellregister 1	Opcode	Zielregister	Opcode	
imm[11:0]	rs1	000	rd	0000011	lb
imm[11:0]	rs1	010	rd	0000011	lh
imm[11:0]	rs1	011	rd	0000011	lw
imm[11:0]	rs1	100	rd	0000011	lbu
imm[11:0]	rs1	101	rd	0000011	lhu

31.....25 24.....20 19.....15 14.....12

Codiert Größe und Format (unsigned/ signed) der geladenen Daten

- LBU: “load unsigned byte”
- LH: “load halfword”, lädt 16 Bit (2 Bytes) und erweitert das Vorzeichen, damit eine vorzeichenkorrekte 32 Bit-Zahl entsteht
- LHU: “load unsigned halfword”, lädt 16 Bit (2 Bytes) und füllt die restlichen Bits mit 0 auf

Referenzen

[1] Frank Slomka, Michael Glaß
Grundlagen der Rechnerarchitektur
Von der Schaltung zum Prozessor

[2] David Harris, Sarah Harris
Digital Design and Computer Architecture, RISC-V Edition

[3] David Patterson, Andrew Waterman
The RISC-V Reader: An Open Architecture Atlas
Strawberry Canyon 2017, ISBN-13: 978-0999249116

[4] RISC-V "Green Card" – Instruktionsübersicht
https://inst.eecs.berkeley.edu/~cs61c/fa17/img/riscvcard.pdf

[5] Michael J. Clark, RISC-V Assembler Reference
https://michaeljclark.github.io/asm.html

[6] Uni Freiburg – RISC-V Processor Design
https://cca.informatik.uni-freiburg.de/riscv-simulator/