Kapitel 2: Rechner im Überblick

Rechnersichten

Rechnerorganisation: Aufbau und Funktionsweise

Assembler

Lernziele

- Die Assembler – Sprache als letzte lesbare Sprachebene kennenlernen
- Die Zusammensetzung von Befehlen aus meist elementaren Operationen verstehen und in einfachen Beispielen anwenden können.
- Register des Prozessors als Instrument zur Datenmanipulation kennenlernen
- Den Unterschied der Adressierungsarten für Operanden kennenlernen

Noch einmal: Ebenensicht

Höhere Programmiersprachen	C++, Java, Fortran
Assembler	Symbolische Notation der bisher vorhandenen Befehle
Betriebssystemebene	Zusätzliche Dienste (z.B. Speicherorganisation, Dateiverwaltung
Maschinensprache	Unterste frei zugängliche Sprache, Befehle sind Folgen über 0 und 1
Mikroprogrammebene	Instruktionen zum Setzen von Steuersignalen
Digitale Ebene	Die eigentliche Hardware

Noch einmal: Ebenensicht

Höhere Programmiersprachen	C++, Java, Fortran
Assembler	**Symbolische Notation der bisher vorhandenen Befehle**
Betriebssystemebene	Zusätzliche Dienste (z.B. Speicherorganisation, Dateiverwaltung
Maschinensprache	Unterste frei zugängliche Sprache, Befehle sind Folgen über 0 und 1
Mikroprogrammebene	Instruktionen zum Setzen von Steuersignalen
Digitale Ebene	Die eigentliche Hardware

Der Unterschied zwischen Assembler und höheren Programmiersprachen

Assembler

- Zusammengang mehrerer Befehle oft schwer erkennbar
- Einfache Befehle
- Direkter Speicherzugriff
- Maschinenabhängige Programme

Höhere Programmiersprache

- Gute Lesbarkeit des Quelltextes
- Komplexe Sprachkonstrukte
- Konstruktion komplexer Datentypen
- Weitgehend von Maschinen unabhängige Programme

Befehlssatz

Befehlssatz wird beim Entwurf eines Prozessors festgelegt

- Kompromiss:
  - Wünsche aus Anwendersicht
  - Technische Machbarkeit

- Von Bedeutung sind dabei
  - Bitbreite zur Kodierung eines Befehls
  - Zulässige Adressierungsarten von Befehlen
  - ...

Klassen von Befehlen

Arithmetische Befehle

Add, Sub, Mult, Div, Srl

Klassen von Befehlen

Arithmetische Befehle	Add, Sub, Mult, Div, Srl
Datentransportbefehle	Load, Store, Move, Push, Pop, I/O

Klassen von Befehlen

Arithmetische Befehle	Add, Sub, Mult, Div, Srl
Datentransportbefehle	Load, Store, Move, Push, Pop, I/O
Logische Befehle	And, Or, Not

Klassen von Befehlen

Arithmetische Befehle	Add, Sub, Mult, Div, Srl
Datentransportbefehle	Load, Store, Move, Push, Pop, I/O
Logische Befehle	And, Or, Not
Bitverarbeitende Befehle	Setzen/Rücksetzen von Bits, Statusflags

Klassen von Befehlen

Arithmetische Befehle	Add, Sub, Mult, Div, Srl
Datentransportbefehle	Load, Store, Move, Push, Pop, I/O
Logische Befehle	And, Or, Not
Bitverarbeitende Befehle	Setzen/Rücksetzen von Bits, Statusflags
Sprungbefehle	jmp, bgez, bnez

Klassen von Befehlen

Arithmetische Befehle	Add, Sub, Mult, Div, Srl
Datentransportbefehle	Load, Store, Move, Push, Pop, I/O
Logische Befehle	And, Or, Not
Bitverarbeitende Befehle	Setzen/Rücksetzen von Bits, Statusflags
Sprungbefehle	jmp, bgez, bnez
Systembefehle	

Klassen von Befehlen

Arithmetische Befehle	Add, Sub, Mult, Div, Srl
Datentransportbefehle	Load, Store, Move, Push, Pop, I/O
Logische Befehle	And, Or, Not
Bitverarbeitende Befehle	Setzen/Rücksetzen von Bits, Statusflags
Sprungbefehle	jmp, bgez, bnez
Systembefehle	

Im Folgenden: Befehle RISC-V RV32**IM**

RISC-V Foundation

Exkurs: RISC-V

- Modulare 32/64/128-Bit RISC-Architektur
  - Basisinstruktionen RVd**I** (d=32,64,128)
  - Optionale Befehlssatzerweiterungen, u.a.
    - RVd**M**: Multiplikation/Division
    - RVd{**F**|**D**|**Q**}: Gleitkommaarithmetik mit einfacher (**F**), doppelter (**D**) und vierfacher Genauigkeit (**Q**)
- LOAD/STORE-Architektur:
  - Lediglich durch LOAD- bzw. STORE-Befehle kann auf den Hauptspeicher zugegriffen werden
  - Alle anderen Befehle arbeiten nur auf Registern
- Die einfache Architektur erlaubt einen Einblick in Assemblerprogrammierung

Register (1)

- Ein 32-Bit Datentyp wird als Wort bezeichnet
- Konventionen legen fest, wie diese Register heißen und wie sie verwendet werden sollen:
  - 12 Register für Variablen des Quellprogramms:
    s0, ..., s11
  - 7 Register für temporäre Variablen:
    t0, ..., t6
- Compiler (oder Programmierer/Anwender) muss sich nicht unbedingt an solche Konventionen halten
- Derartige Konventionen sind aber notwendig, damit getrennt übersetzte Programmteile zusammenarbeiten können

Register (2)

Register	Name	Verwendung
x0	zero	Konstante 0
x1	ra	Rücksprungadresse
x2	sp	Stackpointer
x3	gp	Globaler Pointer
x4	tp	Threadpointer
x5-7	t0-2	Temporäre Register
x8	s0/fp	Registerzwischenspeicher oder Framepointer
x9	s1	Registerzwischenspeicher
x10-11	a0-1	Funktionsargumente und Rückgabewerte
x12-17	a2-7	Funktionsargumente
x18-27	s2-11	Registerzwischenspeicher
x28-31	t3-6	Temporäre Register

Register (3)

- Beschränkte Anzahl von Registern: Unzureichend, um die Variablen realistischer Programme aufzunehmen

Größere Mengen von Variablen bzw. komplexere Datentypen (z.B. Arrays) müssen im Speicher abgelegt werden können

Arithmetische Befehle (1)

- Befehle für Festkommaarithmetik (RV32 **IM**)
  - mit oder ohne Vorzeichen
- Befehle für Gleitkommaarithmetik (RV32 **F**)
  - im Prozessor integriert
  - Emulation durch Zerlegung in elementare Operationen
- Vergleichsbefehle
- Schiebe- und Rotationsbefehle

Arithmetische Befehle (2)

- ADD

ADD Rdest, Rsrc1, Rsrc2	Add	Rdest := $Rsrc1 + Rsrc2$
ADDI Rdest, Rsrc, Imm	Add immediate	Rdest := $Rsrc + Imm$

Arithmetische Befehle (2)

- ADD

ADD Rdest, Rsrc1, Rsrc2	Add	Rdest := $Rsrc1 + Rsrc2$
ADDI Rdest, Rsrc, Imm	Add immediate	Rdest := $Rsrc + Imm$

- SUB

SUB Rdest, Rsrc1, Rsrc2	Sub	Rdest := $Rsrc1 - Rsrc2$

Arithmetische Befehle (2)

- ADD

ADD Rdest, Rsrc1, Rsrc2	Add	Rdest := $Rsrc1 + Rsrc2$
ADDI Rdest, Rsrc, Imm	Add immediate	Rdest := $Rsrc + Imm$

- SUB

SUB Rdest, Rsrc1, Rsrc2	Sub	Rdest := $Rsrc1 - Rsrc2$

Kein SUBI in RV32 I

Arithmetische Befehle (2)

- ADD

ADD Rdest, Rsrc1, Rsrc2	Add	Rdest := $Rsrc1 + Rsrc2$
ADDI Rdest, Rsrc, Imm	Add immediate	Rdest := $Rsrc + Imm$

- SUB

SUB Rdest, Rsrc1, Rsrc2	Sub	Rdest := $Rsrc1 - Rsrc2$

- RV32**M** : MUL, DIV, REM (remainder)

Arithmetische Befehle (2)

- ADD

ADD Rdest, Rsrc1, Rsrc2	Add	Rdest := $Rsrc1 + Rsrc2$
ADDI Rdest, Rsrc, Imm	Add immediate	Rdest := $Rsrc + Imm$

- SUB

SUB Rdest, Rsrc1, Rsrc2	Sub	Rdest := $Rsrc1 - Rsrc2$

- RV32**M** : MUL, DIV, REM (remainder)
- Andere Assemblersprachen lassen auch Speicherinhalte als Operanden zu (z.B. CISC)

Beispiele (1)

Bedeutung

- Einfache Ausdrücke:
  - $-a = a + b$
  - $-a = a - b$

RV Instruktion

Beispiele (1)

Bedeutung

- Einfache Ausdrücke:
  - $a = a + b$
  - $a = a - b$

RV Instruktion

```
add a,a,b
sub a,a,b
```

Beispiele (1)

Bedeutung

- Einfache Ausdrücke:
  - $a = a + b$
  - $a = a - b$
- Aufteilung längerer Ausdrücke in einfachere Operationen
  - $a = b + c + d + e$

RV Instruktion

```
add a,a,b
sub a,a,b
```

Beispiele (1)

Bedeutung

- Einfache Ausdrücke:
  - $a = a + b$
  - $a = a - b$
- Aufteilung längerer Ausdrücke in einfachere Operationen
  - $a = b + c + d + e$

RV Instruktion

```
add a,a,b
sub a,a,b
```

```
add a,b,c
add a,a,d
add a,a,e
```

Beispiele (1)

Bedeutung

- Einfache Ausdrücke:
  - $a = a + b$
  - $a = a - b$
- Aufteilung längerer Ausdrücke in einfachere Operationen
  - $a = b + c + d + e$
- Wenn nötig, Einführung temporärer Variablen durch Anwender/Compiler
  - $f = (g + h) - (i + j)$

RV Instruktion

```
add a,a,b
sub a,a,b
```

```
add a,b,c
add a,a,d
add a,a,e
```

Beispiele (1)

Bedeutung

- Einfache Ausdrücke:
  - $a = a + b$
  - $a = a - b$
- Aufteilung längerer Ausdrücke in einfachere Operationen
  - $a = b + c + d + e$
- Wenn nötig, Einführung temporärer Variablen durch Anwender/Compiler
  - $f = (g + h) - (i + j)$

RV Instruktion

```
add a,a,b
sub a,a,b
```

```
add a,b,c
add a,a,d
add a,a,e
```

```
add t0,g,h
add t1,i,j
sub f,t0,t1
```

Beispiele (2)

Befehl: $f = (g + h) - (i + j)$

Annahme:

- Variable **`g`** befindet sich im Register **`s1`**,
- Variable **`h`** im Register **`s2`**,
- Variable **`i`** im Register **`s3`** und
- Variable **`j`** im Register **`s4`**.
- Das Ergebnis soll im Register **`s0`** gespeichert werden.

```
add t0, s1, s2 # t0 = g + h
add t1, s3, s4 # t1 = i + j
sub s0, t0, t1 # f = (g+h)-(i+j)
```

*Anmerkung: Das Zeichen **#** leitet einen Kommentar ein, der bis zum Ende der Zeile geht*

Arithmetische Befehle (3)

- Vergleichsbefehle

SLT Rdest, Rsrc1, Rsrc2 SLTI Rdest, Rsrc, Imm	Set less than	Rdest := $\begin{cases} 1, Rsrc1 < Src2 \\ 0, sonst \end{cases}$
SLTU Rdest, Rsrc1, Rsrc2 SLTIU Rdest, Rsrc1, Imm	SLT unsigned	Rdest := $\begin{cases} 1, Rsrc1 < Src2 \\ 0, sonst \end{cases}$

Notation: Src2 kann Register(Rsrc2) oder Immediate (Imm) sein

Arithmetische Befehle (4)

- Schiebebefehle

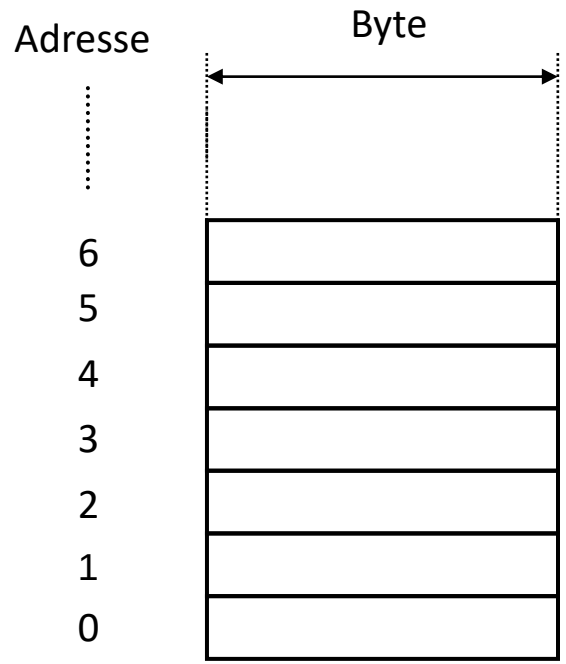
SLL Rdest, Rsrc1, Rsrc2 SLLI Rdest, Rsrc1, Imm	Shift left logical	Rdest := $Rsrc1 \ll Src2$
SRL Rdest, Rsrc1, Rsrc2 SRLI Rdest, Rsrc1, Imm	Shift right logical	Rdest := $Rsrc1 \gg Src2$
SRA Rdest, Rsrc1, Rsrc2 SRAI Rdest, Rsrc1, Imm	Shift right arithmetic	Rdest := $Rsrc1 \cdot 2^{-Src2}$

Datentransportbefehle (1)

- Transport eines Datums von Quelle zu Ziel
- Quelle und Ziel im Hauptspeicher oder Registern
- Übertragung auch zwischen I/O-Schnittstellen und Registern
- „Transport“ eigentlich nicht richtig, da nichts von Quelle entfernt wird

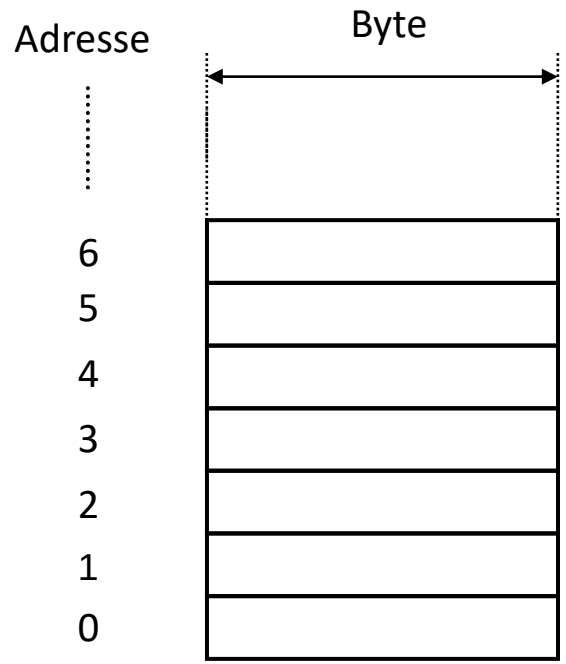
Speicheradressierung

- Speicher wird mit Byteadressen adressiert
  - Speicher ≈ großes, eindimensionales Array von Bytes
  - Adresse einer Speicherzelle entspricht dem Array-Index
  - Niedrigste Adresse ist 0

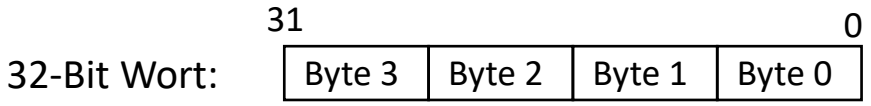


Speicheradressierung

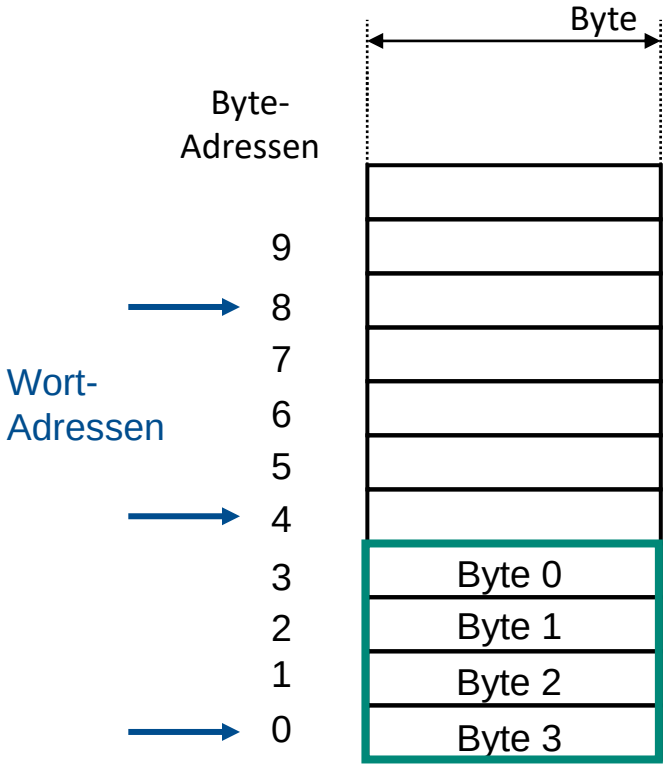
- Speicher wird mit Byteadressen adressiert
  - Speicher ≈ großes, eindimensionales Array von Bytes
  - Adresse einer Speicherzelle entspricht dem Array-Index
  - Niedrigste Adresse ist 0
- Beispiel: Wie wird ein Wort (4 Byte) im Speicher abgelegt?
  - Das höchstwertige Byte des Wortes befindet sich an der niedrigsten Byte-Adresse
  - Ein Wort wird mit der Adresse seines höchstwertigsten Bytes adressiert
  - Wortadressen müssen ein Vielfaches von 4 sein („Alignment Restriction“)



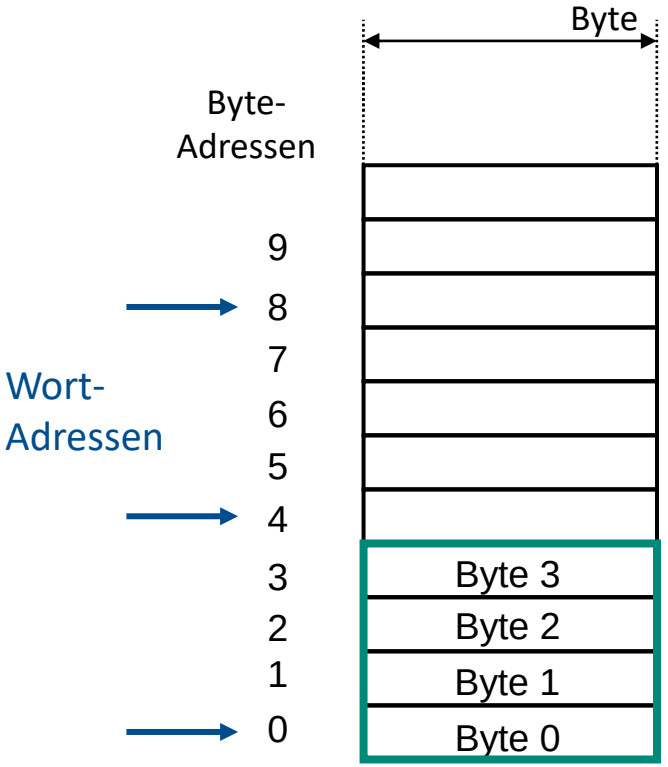
Big Endian vs. Little Endian



Big Endian



Little Endian



Datentransportbefehle (2)

- LOAD- Befehle

LUI Rdest, Imm	Load upper immediate	Rdest := Imm[31:12]
LW Rdest, Rbase, Offs	Load word	Rdest := Mem[Rbase + Offs]
LB Rdest, Rbase, Offs	Load byte	Rdest := Mem[Rbase + Offs]

- STORE- Befehle

SB Rsrc, Rbase, Offs	Store byte	Mem[Rbase + Offs] := Rsrc[7:0]
SW Rsrc, Rbase, Offs	Store word	Mem[Rbase + Offs] := Rsrc

Beispiel (2)

- Befehl: $A[12] = h + A[8]$

- Annahme:
  - A ist ein Array vom Datentyp Wort.
  - Die Variable h steht im Register s2,
  - Die Basisadresse von A steht im Register s3

```
lw   t0, 32(s3)   # t0 = Memory[s3 + 32]
add  t0, s2, t0   # t0 = h + A[8]
sw   t0, 48(s3)   # Memory[s3 + 48] = t0
```

Adressierungsarten (keine RISCV-Befehle)

implizite Adressierung	Ziel wird durch den Befehl bestimmt	`LOADA Rsrc ↔ A := Rsrc`
unmittelbare Adressierung	Operand wird als Wert angegeben	`LI Rdest, 4 ↔ Rdest := 4`
absolute (direkte) Adressierung	Operand ist Inhalt der angegebenen Adresse	`LW Rdest, Rsrc ↔ Rdest := Mem[Rsrc]`
indirekte Adressierung	Operand ist Inhalt des Zeigers an der angegebenen Adresse	`LW Rdest,(Rsrc)↔Rdest:= Mem[Mem[Rsrc]]`
indizierte Adressierung	Operand ist Inhalt der angegebenen Adresse + Index	`LB Rdest, 4(Rsrc)↔Rdest:= Mem[Rsrc+4]`

Logische Befehle

- AND

AND Rdest, Rsrc1, Rsrc2 ANDI Rdest, Rsrc1, Imm	And	Rdest := $Rsrc1 \ \& \ Src2$

- analog dazu: OR, XOR

Beispiel (3)

- Annahme:
  - `s0` enthält `0…00001001`

```
sll   s1, s0, 2          # s1 = ?
or    s1, s1, s0         # s1 = ?
andi  s1, s1, 15         # s1 = ?
andi  s2, s1, 0          # s2 = ?
```

Beispiel (3)

- Annahme:
  - `s0` **enthält** `0…00001001`

```
sll   s1, s0, 2              # s1 = 0…00100100
or    s1, s1, s0             # s1 = ?
andi  s1, s1, 15             # s1 = ?
andi  s2, s1, 0              # s2 = ?
```

Beispiel (3)

- Annahme:
  - `s0` enthält `0…00001001`

```
sll   s1, s0, 2          # s1 = 0…00100100
or    s1, s1, s0         # s1 = 0…00101101
andi  s1, s1, 15         # s1 = ?
andi  s2, s1, 0          # s2 = ?
```

Beispiel (3)

- Annahme:
  - `s0` **enthält** `0…00001001`

```
sll   s1, s0, 2          # s1 = 0…00100100
or    s1, s1, s0         # s1 = 0…00101101
andi  s1, s1, 15         # 15 = 0…00001111
                         # s1 = 0…00001101
andi  s2, s1, 0          # s2 = ?
```

Beispiel (3)

- Annahme:
  - `s0` **enthält** `0…00001001`

```
sll   s1, s0, 2          # s1 = 0…00100100
or    s1, s1, s0         # s1 = 0…00101101
andi  s1, s1, 15         # 15 = 0…00001111
                         # s1 = 0…00001101
andi  s2, s1, 0          #  0 = 0…00000000
                         # s2 = 0…00000000
```

Beispiel (3) – mit Hex

- Annahme:
  - `s0` enthält `0x00000009`

```
sll   s1, s0, 2          # s1 = 0x00000024
or    s1, s1, s0         # s1 = 0x0000002D
andi  s1, s1, 15         # 15 = 0x0000000F
                         # s1 = 0x0000000D
andi  s2, s1, 0          #  0 = 0x00000000
                         # s2 = 0x00000000
```

Sprungbefehle (1)

- Zur Ablaufsteuerung von Programmen
- Klassifikation
  - bedingte / unbedingte Sprungbefehle (Sprung erfolgt relativ zum Programmzähler oder absolut)
  - Unterprogrammaufrufe
  - Rückkehr zum aufrufenden Programmabschnitt (Unterprogrammbeendigung)
  - Unterbrechung (Interrupt)

Sprungbefehle (2)

- Unbedingte Sprünge

JAL Rdest, offs	Jump and Link	Rdest := pc+4, Sprung zu pc+offs
JALR Rdest, Rbase, offs	JAL Register	JAL zu Adresse Rbase+offs

Sprungbefehle (3)

- Bedingte Sprünge

BEQ Rsrc1, Rsrc2, offs	Branch Equal	Sprung, falls $Rsrc1 == Rsrc2$
BNE Rsrc1, Rsrc2, offs	Branch Not Equal	Sprung, falls $Rsrc1 != Rsrc2$
BLT Rsrc1, Rsrc2, offs	Branch Less Than	Sprung, falls $Rsrc1 < Rsrc2$
BGE Rsrc1, Rsrc2, offs	Branch Greater Equal	Sprung, falls $Rsrc1 >= Rsrc2$
BLTU Rsrc1, Rsrc2, offs	Unsigned BLT	Sprung, falls $u(Rsrc1) < u(Rsrc2)$
BGEU Rsrc1, Rsrc2, offs	Unsigned BGE	Sprung, falls $u(Rsrc1) >= u(Rsrc2)$

Beispiel (if-then-else)

- C/C++ - Programm:

```
if (i == j)
   { f = g + h; }
else
   { f = g - h; }
```

- Annahmen:
  - Variable `f` in `s0`
  - Variable `g` in `s1`
  - Variable `h` in `s2`
  - Variable `i` in `s3`
  - Variable `j` in `s4`

```
        bne   s3, s4, Else  # if (i != j) goto Else
        add   s0, s1, s2    # f = g + h
        j     Exit          # goto Exit
Else:   sub   s0, s1, s2    # f = g - h
Exit:   ...
```

`Else, Exit` sind Sprungmarken / Labels.

Beispiel (Schleife)

- C/C++ - Programm:

```
while ( save[i] == k)
    { i = i+1; }
```

- Annahmen:
  - Variable `i` in `s3`
  - Variable `k` in `s5`
  - Basisadresse von `save` in `s6`

```
Loop:   sll   t1, s3, 2       # t1 = 4 * i
        add   t1, t1, s6      # t1 = Adresse von save[i]
        lw    t0, 0(t1)       # t0 = save[i]
        bne   t0, s5, Exit    # if (save[i] != k) goto Exit
        addi  s3, s3, 1       # i = i + 1
        j     Loop            # goto Loop (= jal zero, Loop)
Exit:   …
```

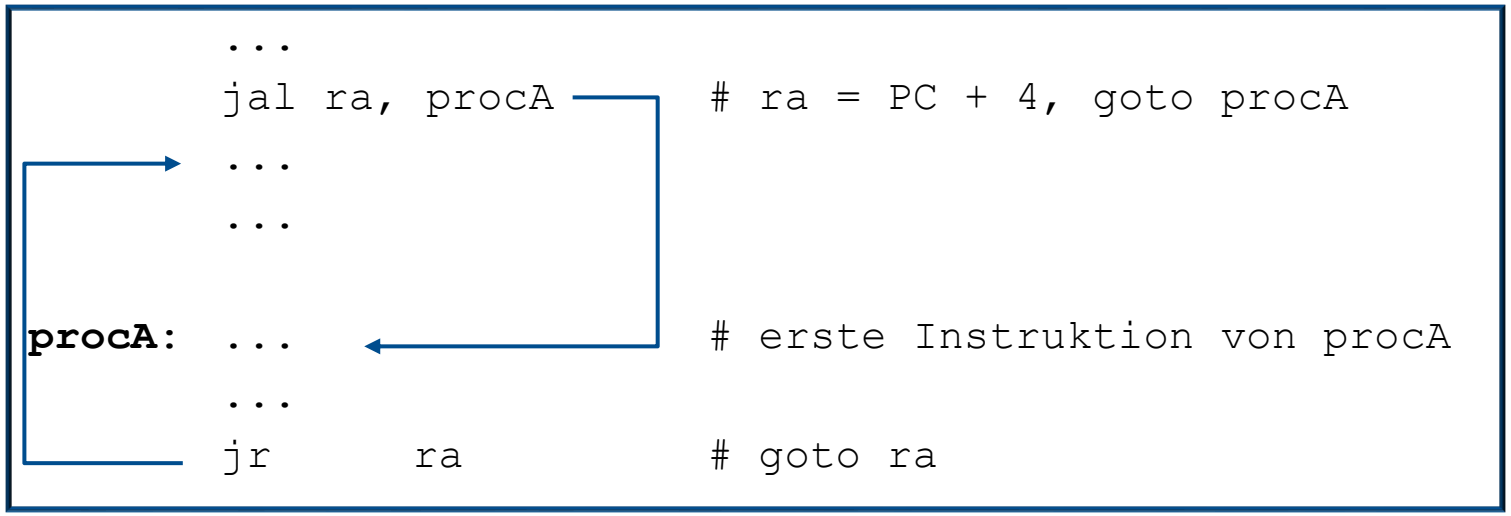
Unterprogramme (1)

- Prinzipieller Ablauf
  - Der Kontrollfluss wird an die aufgerufene Prozedur übergeben
  - Die Prozedur führt Berechnungen aus
  - Der Kontrollfluss wird wieder zurück an die aufrufende Prozedur übergeben
- RISC-V-Unterstützung für Prozeduraufrufe
  - Spezielles Register für die Sicherung der Rücksprungadresse: **`ra`**
  - Sprunginstruktionen:

JAL	Jump and Link	Sichert die Rücksprungadresse in **`ra`** und springt zur Adresse der Prozedur
JR	Jump Register	Springt zur Adresse im Register, d.h. **`jr ra`** springt zurück zum aufrufenden Programm

Unterprogramme (2)

- Unterprogrammaufrufe durch Sprung zu Labeln
- Prozeduren benötigen i.A. Argumente und liefern Resultate. Für eine schnelle Argument- und Resultatübergabe gibt es in RISC-V folgende Konvention:
  - 8 Register für Argumente: `a0,...,a7`
  - 2 Register für Resultate: `a0,a1`



Ein Beispielprogramm: $n!$

- Annahmen:
  - Wert n in Register `a0`
  - Rückgabe von $n!$ in Register `a1`

```
Fak:
    addi sp, sp, -12          # Stack um 12 dekrementieren
    sw fp, 12(sp)             # Reg. retten:   -Framepointer
    sw ra, 8(sp)              #                -Return-Adresse
    addi fp, sp, 12           #       Framepointer aktualisieren
    sw a0, 4(sp)              # Argument speichern
    beq a0, zero, Ret1        # n = 0 ?
    addi a0, a0, -1           # Nein: dekrementiere n
    jal Fak                   # rekursiver Aufruf
    lw a0, 4(sp)              #
    mulh t0, a0, a1           # berechne a0*a1 = n*(n-1)!
    mul t1, a0, a1            #
    mv a1, t1                 #
    bne t0, zero, Error       # Fehler: Overflow, falls t0 != 0
    j Ret
Ret1:                         #  Ja (n=0):
    addi a1, zero, 1          # Rückgabe von 0! = 1
Ret:
    lw fp, 12(sp)             # Register fp, ra, sp
    lw ra, 8(sp)              #   wiederherstellen
    addi sp, sp, 12
    jr ra                     # Rücksprung
```

Systembefehle

- In RISC-V:

ECALL	Systemaufruf

- Control-/Statusregister (CSR):
  - Fehlerbehandlung (Exceptions)
  - Prozessorverhalten steuern

CSRR{W|S|R} Rdest, Rsrc, csr	Lese bzw. überschreibe Control- und Statusregister

Instruktionscodierung

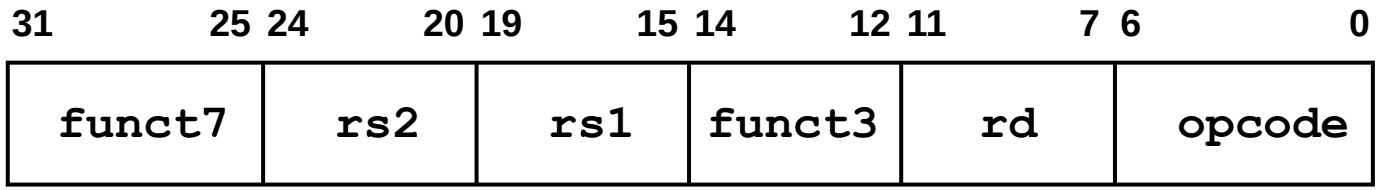
- Die Instruktionen wurden bisher in Assemblersprache notiert, der Prozessor versteht aber nur Maschinensprache.
- Instruktionen müssen binär codiert werden: Instruktionskodierung

```
add  t0, s1, s2 →  0000000 10010 01001 000 00101 0110011
```

- Bei RV32 sind alle Instruktionen 32 Bit lang!
  - Da die verschiedenen Instruktionen unterschiedlich viele Operanden haben, werden verschiedene Instruktionsformate verwendet:
    - R-Typ Instruktionen, I-Typ Instruktionen
    - S/B-Typ Instruktionen, U/J-Typ Instruktionen

Instruktionsformat R-Typ

- Instruktionsformat R-Typ (Register-Format) wird für arithmetische und logische Instruktionen verwendet



opcode	Operationscode (OP-Code)
rd	Register, in dem das Ergebnis gespeichert wird
rs1	Register des ersten Quelloperanden
rs2	Register des zweiten Quelloperanden
funct3/7	Funktionscode (function), Variante einer Operation

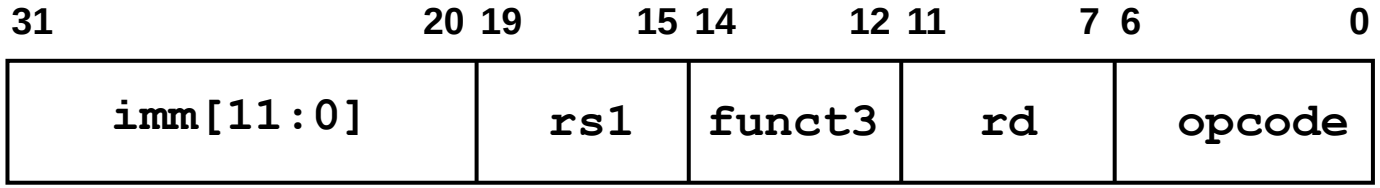
Beispiel

```
add  t0, s1, s2
```

000000	10010	01001	000	00101	0110011

Instruktionsformat I-Typ

- Instruktionsformat I-Typ (Immediate-Format) wird verwendet für
  - Immediate-Versionen der arithmetischen und logischen Instruktionen,
  - Load-Instruktionen und Systembefehle.



  - **imm** ist eine vorzeichenbehaftete 12-bit Zahl im 2er-Komplement und kann Werte zwischen -2^{11} und $+2^{11}-1$ annehmen

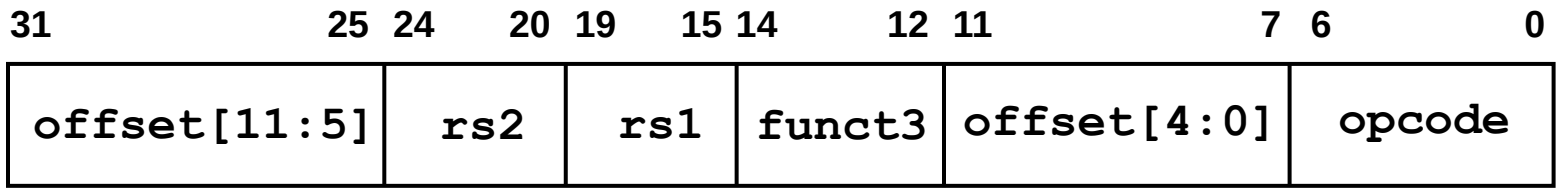
Beispiel

```
lw   s1, -4(s2)
```

111111111100	**10010**	**010**	01001	0000**011**

Instruktionsformat S/B-Typ

- Instruktionsformat S-Typ (Store-Format) wird für Store-Befehle verwendet



  - **`offset`** ist eine vorzeichenbehaftete 12-bit Zahl, auf die der Wert aus Register **`rs2`** addiert wird, um die effektive Speicheradresse zu erhalten

Beispiel

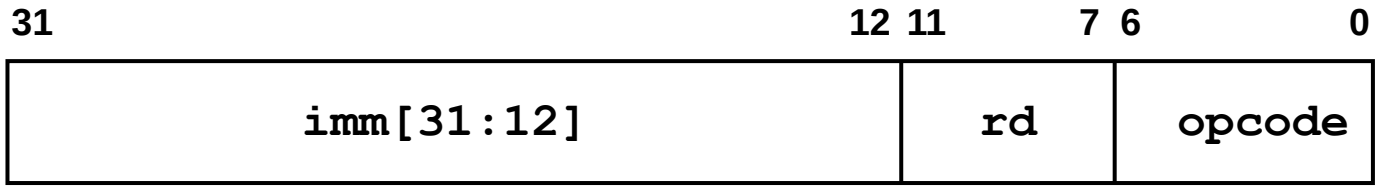
```
sw   s1, -4(s2)
```

1111111	01001	**10010**	010	**11100**	0100011

- Beim B-Typ (Branch-Format, für bedingte Sprünge) andere Interpretation der **`offset`**-Bits

Instruktionsformat U/J-Typ

- Instruktionsformat U-Typ (Upper-Format) wird für spezielle Load-Befehle verwendet



  - **`imm`** ist eine 32-bit Zahl, deren 20 höchstwertige Bits **`imm[31:12]`** ins Register **`rd`** geladen werden

- Beim J-Typ (unbedingte Sprünge/Unterprogrammaufrufe) wird die effektive Sprungadresse aus **`imm[20:1]`** (interpretiert als vorzeichenbehaftete 20-bit Zahl) konstruiert.

Der Unterschied zwischen Assembler und höheren Programmiersprachen

Assembler

- Zusammengang mehrerer Befehle oft schwer erkennbar
- Einfache Befehle
- Direkter Speicherzugriff
- Maschinenabhängige Programme

Höhere Programmiersprache

- Gute Lesbarkeit des Quelltextes
- Komplexe Sprachkonstrukte
- Konstruktion komplexer Datentypen
- Weitgehend von Maschinen unabhängige Programme

Der Unterschied zwischen Assembler und Maschinensprache

Assembler

- Symbolische Bezeichnung von Befehler, Adressen (label) und z.T. der Operanden
- Ein-/Ausgaben- Routinen des Betriebssystems können verwendet werden
- Programme müssen vor der Ausführung assembliert werden

Maschinensprache

- Binäre Darstellung von Befehle (OP-Code), Adressen und Operanden in einem Datenwort
- Bibliotheken für komplexe Befehle
- Programme werden direkt ausgeführt (binaries)

Verarbeitung einer Instruktion (RISC-V)

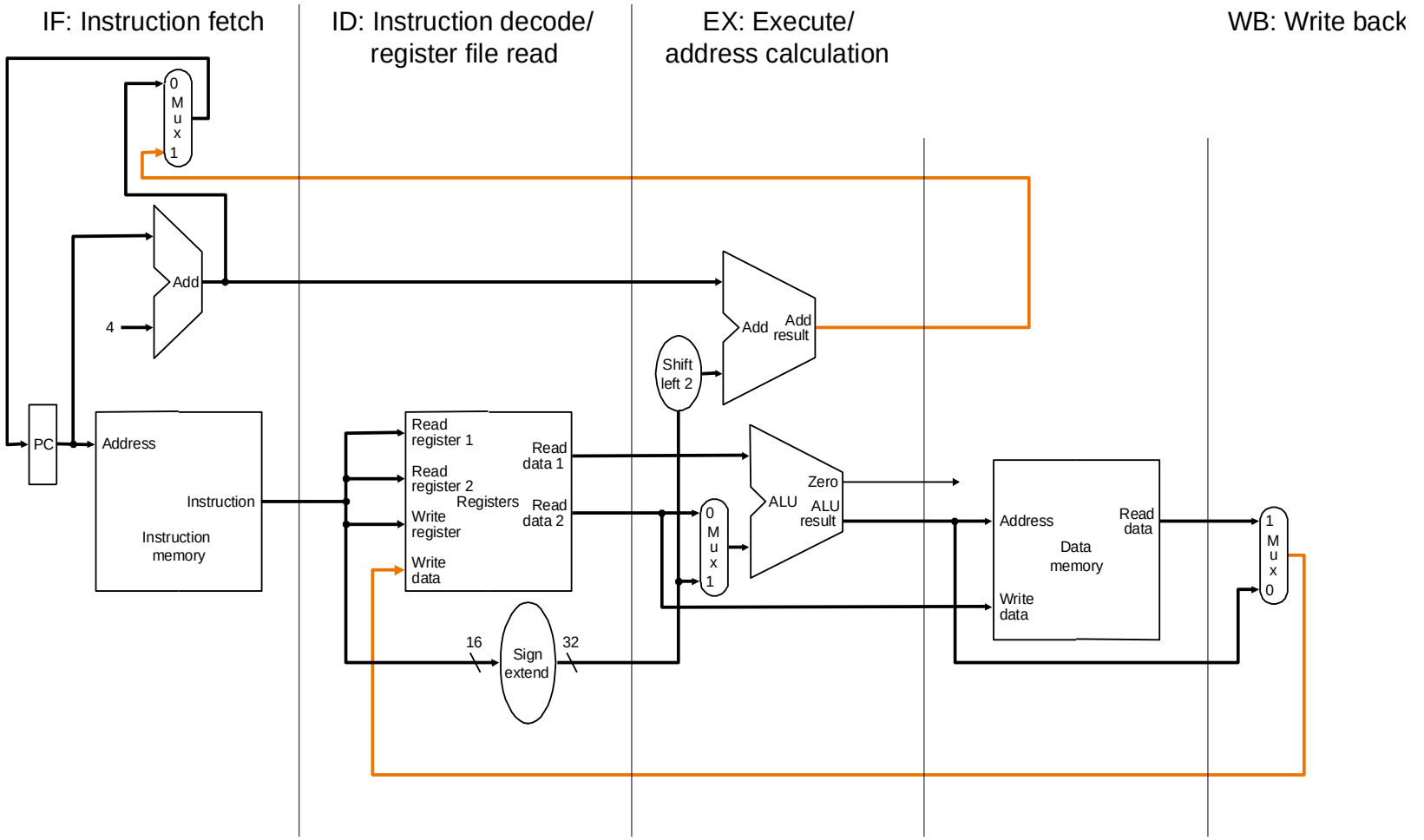
- Im Folgenden bereits um Pipelining erweiterte CPU

  ➡ wird im nächsten Kapitel näher erläutert

- Verarbeitung einer Instruktion verläuft stufenweise
  - Befehl holen
  - Befehl dekodieren / Operanden holen
  - Befehl ausführen
  - Speicherzugriff
  - Ergebnis in Register schreiben

- Hinweis: Je nach Prozessor können einzelne Stufen unterschiedlich ausfallen

Aufbau eines RISC-V-Prozessors



Aufbau eines RISC-V-Prozessors