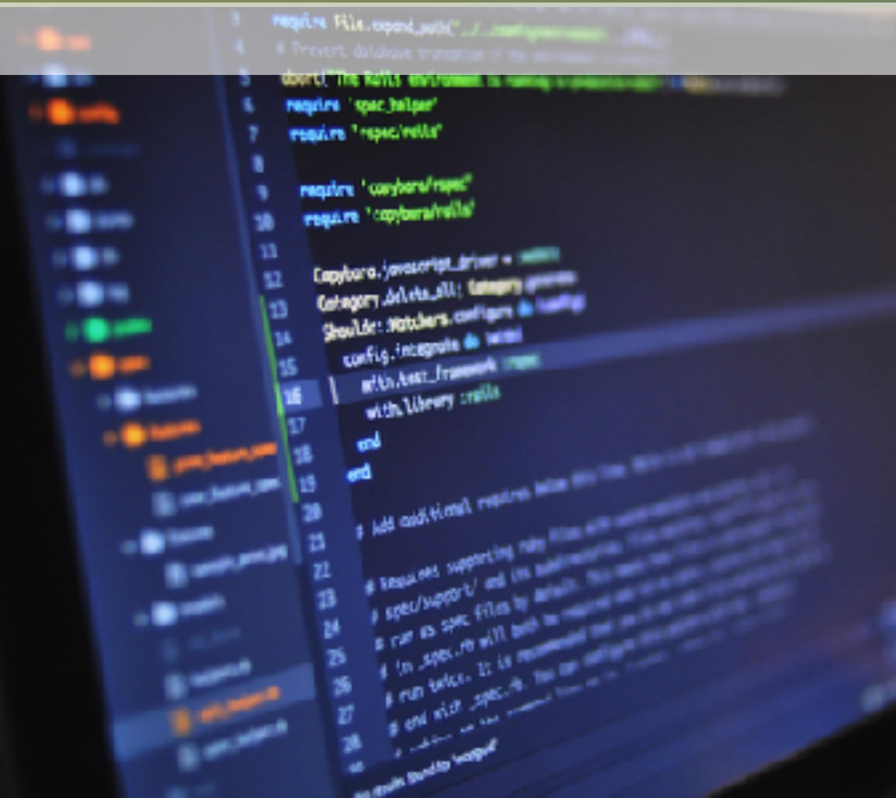
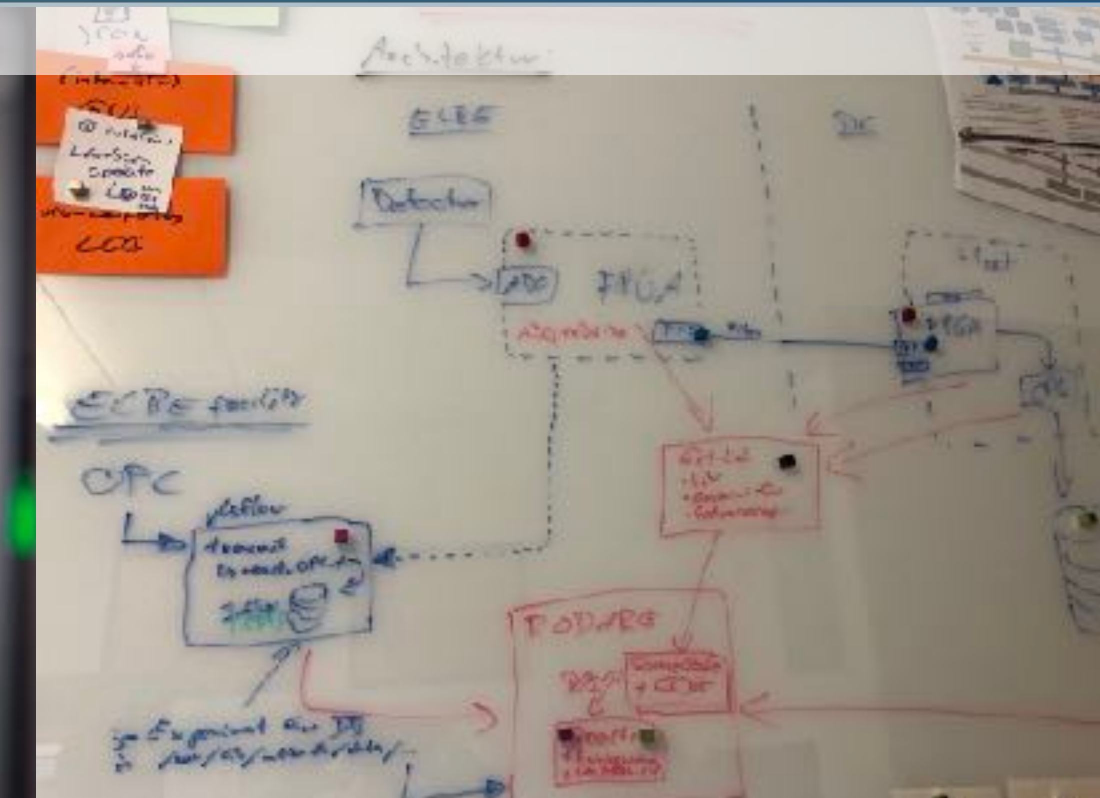
HZDR HELMHOLTZ ZENTRUM DRESDEN ROSSENDORF

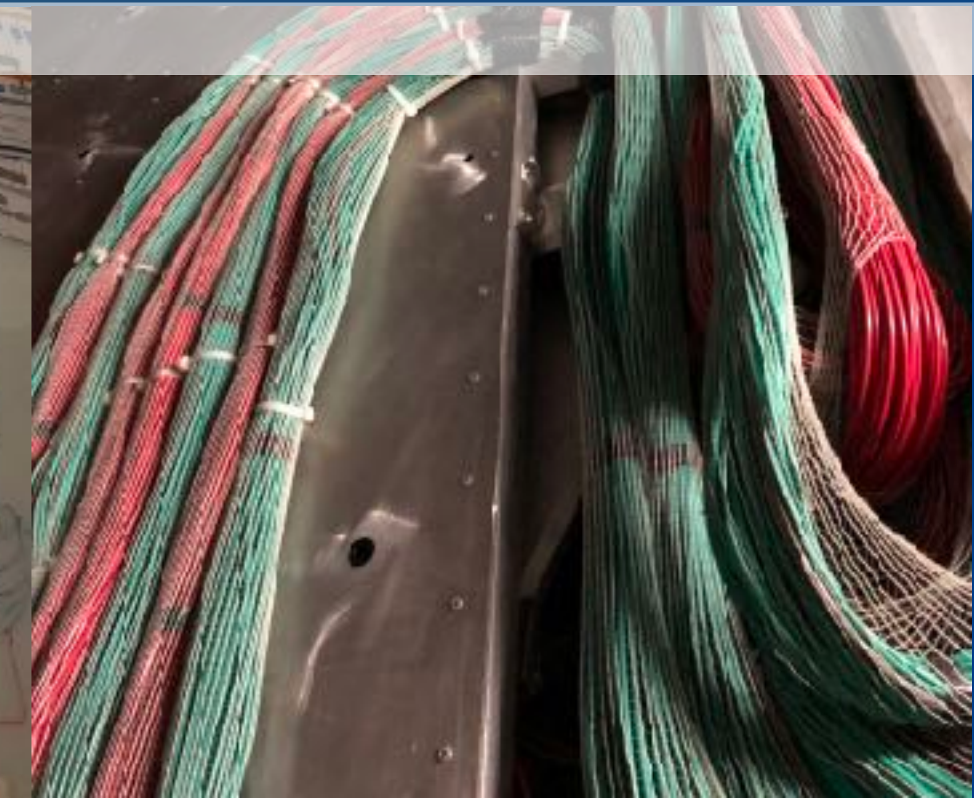
Dr.-Ing. Oliver Knodel

# Befehlssatzarchitektur (ISA)

Dresden // April, 2024







# Begriffserklärung I

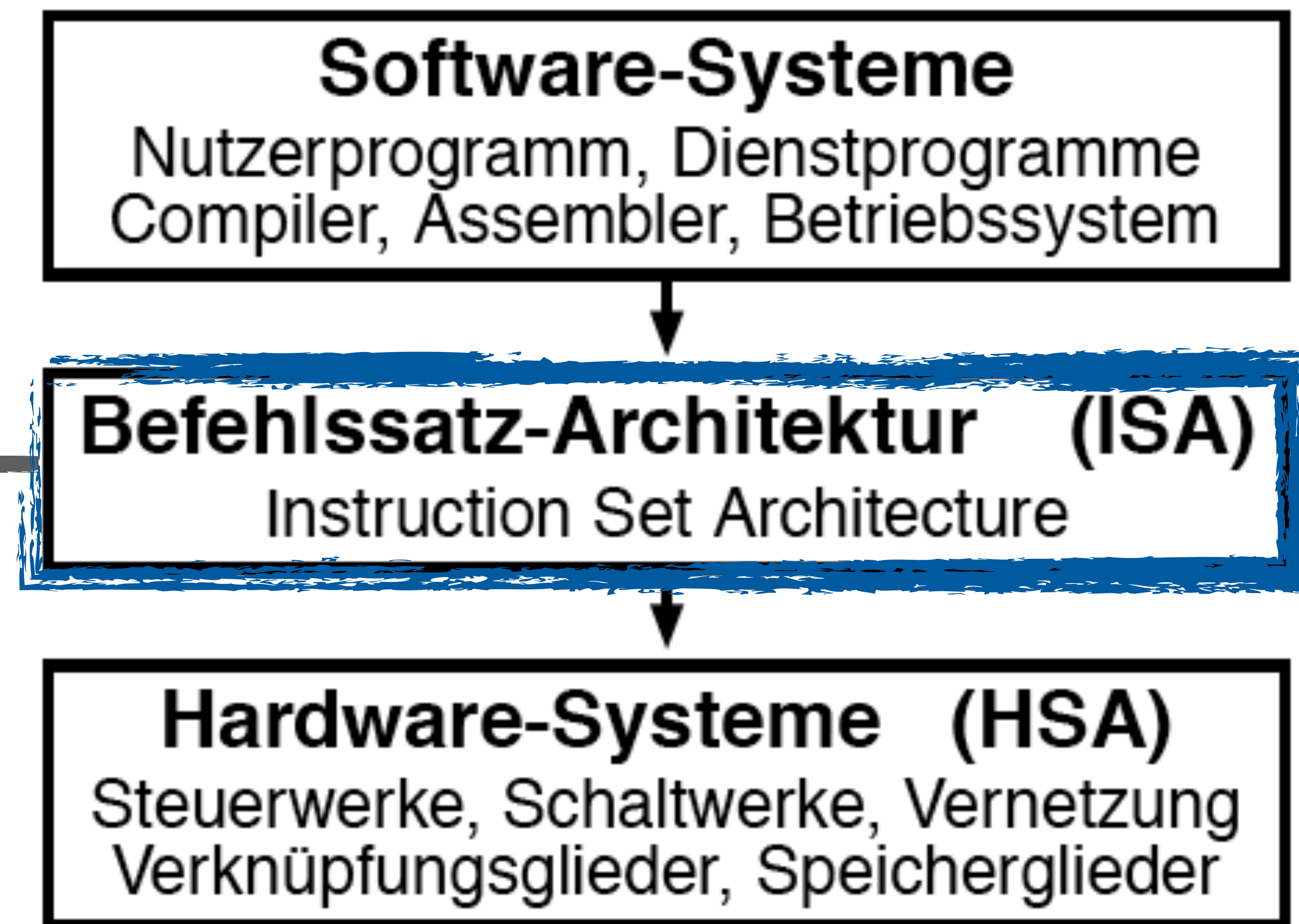
**Algorithmus:** Die Informationsverarbeitung in einem Rechner erfordert die schrittweise Umsetzung eines definierten Algorithmus. Der Algorithmus wird als geordnete Folge von Anweisungen, Befehlen dargestellt.

**Befehl:** Ein Befehl (instruction) ist eine eindeutig spezifizierte Arbeitsanweisung an den Prozessor (CPU). Er ordnet eine Operation an, die in der Regel an spezifizierten Daten (Operanden) vorzunehmen ist und ein Ergebnis (Resultat) liefert. (maschinenlesbar → Maschinenbefehl)

**Befehlssatz:** Der Menge aller in einem Prozessor implementierten Befehle bildet den Befehlssatz. Die Architektur eines Rechners wird wesentlich durch den Befehlssatz des verwendeten Prozessors bestimmt (ISC-Instruction Set Computer → wesentliches Architekturmerkmal).
Die Menge aller Maschinenbefehle definiert die Maschinensprache.

# Der Befehlssatz im Vereinfachten Schichtenmodell

- Von Anfang der sechziger bis Ende der siebziger versuchte man, die interne Struktur und Organisation eines Rechners vor dem Nutzer zu verbergen.
- Die ISA-Schnittstelle oder Befehlssatz-Architektur hat bis dahin die konkrete Hardwarearchitektur weitestgehend verdeckt.
- Für Effiziente Anwendungen ist allerdings gerade in Bezug auf Rechenleistung eine Berücksichtigung der internen Rechnerstruktur und Organisation erforderlich.



# Begriffserklärung II

**Befehlskomponenten:** Die Komponenten eines Befehlssatzes sind die Operation, der Datentyp und die Operanden, Adressierung des Befehls.

**Maschinensprache:** Durch die Menge der im Prozessor realisierbaren Maschinenbefehle ist eine hardwareabhängige Programmiersprache für den Rechner, die Maschinensprache gegeben → Maschinenprogramme. Rechner einer Prozessorfamilie realisieren eine weitestgehend ähnliche Maschinensprache (Binärkompatibilität).

## Hauptmerkmale des Befehlssatzes

**Befehlsvorrat:** Die Menge aller verfügbaren Maschinenbefehle bildet den Befehlsvorrat. Sie werden in der Befehlsliste geordnet zusammengefaßt.

**Befehlsformat:** Die innere Struktur der Maschinenbefehle, dargestellt durch Binärworte, wird durch das Befehlsformat bestimmt. Entsprechend dem Befehlsformat werden die einzelnen Komponenten des Befehls binär codiert im Befehlswort zusammengefaßt.

# Begriffserklärung III

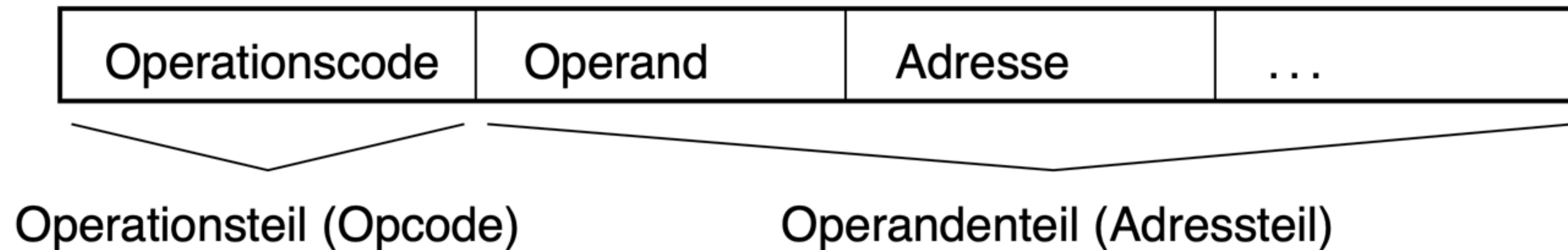
**Orthogonalität:** Ein Befehlssatz heißt orthogonal, wenn eine möglichst kleine Anzahl von grundlegenden Befehlen existiert, die beliebig miteinander kombinierbar sind und sich in ihrer Funktionalität nicht oder nur wenig überschneiden. Die Komponenten eines Befehlssatzes (Operation, Datentyp, Adressierung) sollten orthogonal zueinander sein (voneinander unabhängig). Jede Operation sollte jede relevante Adressierungsart bzw. jeden relevanten Datentyp zulassen.

**Symmetrie:** Ein Befehlssatz heißt symmetrisch, wenn jeder Befehl mit jedem relevanten Datentyp ausgeführt werden kann, sowie jede zulässige Adressierungsart benutzt werden kann.

**Regularität:** Ein Befehlssatz heißt regulär, wenn er nach konsistent anwendbaren Regeln strukturiert ist. Regularität und Orthogonalität stehen in enger Wechselbeziehung.

# Befehlsstruktur

Die Befehlskomponenten ( Operation, Datentyp und Operanden, Adressierung) werden im Befehlswort strukturiert zusammengefaßt und binär codiert. Die Befehlswortlänge ist allgemein nicht für jeden Befehl des Befehlssatzes einheitlich (byteweise abgestuft).




Die Positionen von Opcode und Adressteil sind nicht fest (der Opcode kann am Anfang oder Ende des Befehlswortes stehen aber auch über das Befehlswort verteilt sein).

Der Operandenteil kann je nach Befehlsformat gleichzeitig auch mehrere Operanden und oder Adressen, wie auch andere Informationen enthalten.

# CISC und RISC

Je nach Umfang des im Prozessor realisierten Befehlssatzes können zwei Kategorien von Befehlssatz-Architekturen (ISA) unterschieden werden:

1. **CISC** (Complex Instruction Set Computer)
   Befehlsvorrat: 400..500 Befehle/Befehlsformate
   z.B.: DEC VAX, IBM 360, Intel x86, …
   — Viele mächtige Einzelbefehle unterschiedlicher Länge und Pipelinestufen
   — Intern Nutzung von komplexen Mikroprogrammsteuerwerken

2. **RISC** (Reduced Instruction Set Computer)
   Befehlsvorrat: 40..50 Befehle/Befehlsformate
   z.B.: Sun SPARC, SGI MIPS, DEC ALPHA, HP PARISC, IBM PowerPC, ARM, RISC-V, …
   — Entkopplung von Speicherlese-Operationen und Arithmetik („Load/Store-Architektur“)
   — Die Befehle haben eine konstante Länge

# Motivation für RISC

## 90/10 Regel beim Befehlssatz

Bei einem komplexen Befehlssatz (CISC) werden 90% aller Operationen mit nur 10% der Befehle des Befehlssatzes durchgeführt.

R. Chou und M. Horowitz: „The goal of any instruction format should be: (1) simple decode, (2) simple decode, (3) simple decode."

Albert Einstein: „Keep it simple, as simple as possible, but no simpler."

## Compiler

Modernere Compiler erzeugten schnelleren Code, durch vielschichtige Optimierungen (Registervergabe, statisches Auswerten von Ausdrücken, Entfernen von totem Programmcode).

Viele dieser Optimierungen können auf mikroprogrammierten Prozessoren (CISC) nicht ihr volles Potential ausschöpfen.

# CISC — Complex Instruction Set Computer

- Befehlsworte und Opcode in komplexen Befehlsformaten mit variablen Längen und vielen komfortablen Adressierungsarten

- leistungsfähige, komplexe Befehle führen zu einer Verkürzung des Maschinenprogramms und damit zur Erhöhung der Codedichte

- Realisierung der Maschinenbefehle durch Ausführen von Mikroprogrammen im Prozessor (der Befehlszyklus wird durch eine Mikroprogramm-Steuerung realisiert)

- Anzahl der benötigten Taktzyklen pro Befehl ist unterschiedlich (mehr als 1 Taktzyklus/Befehl)

# RISC — Reduced Instruction Set Computer

- stark reduzierter Umfang an Befehlsformaten und Adressierungsarten (meist weniger als 4 Befehlsformate und 4 Adressierungsarten)
- einfache wenige Basisbefehle, aus denen komplexe Operationen zusammengestellt werden können
- Load/Store-Architektur, ALU-Befehle realisieren keine Speicherzugriffe, Speicherzugriffe erfolgen nur über Load/Store-Befehle
- Universalregister-Architektur (meist 32 oder mehr Universalregister)
- festverdrahtete Maschinenbefehle und fester Befehlszyklus, keine Mikroprogramm-Steuerung
- Ausführung der meisten Befehle in nur einem Taktzyklus

# Klassifikationen von Befehlssatz-Architekturen

# Klassifikationen von Befehlssatz-Architekturen

## Merkmale der Klassifikation

- Operandenspeicherung innerhalb der CPU, wo und wie
- Zahl der explizit im Befehl adressierten Operanden
- Operandenspeicherung, Adressierung, wie spezifiziert
- Operationen des Befehlssatzes
- Typ und Länge der Operanden, wie spezifiziert

Alle bekannten Rechner-Architekturen stellen temporären Operandenspeicher innerhalb der CPU bereit (Register, Stack, Akkumulator).

# Klassifikationen — Operandenspeicherung innerhalb der CPU

## Hauptvarianten, Alternativen der Operandenspeicherung

| temporärer Operandenspeicher | explizite Operanden | Quelle für Operanden | Ziel für Resultate | Zugriff auf Operanden |
|---|---|---|---|---|
| Stack | 0 | Stack | Stack | PUSH/POP auf Stack |
| Akkumulator (ACCU) | 1 | ACCU/ Speicher | ACCU | LOAD/STORE auf ACCU |
| Registersatz (Universalregister) | 2 oder 3 | Register/ Speicher | Register/ Speicher | LOAD/STORE auf Register |

# Stack-Architektur (Null-Adress-Maschine)

Speicher

A

B

C

PUSH

POP

SP

Stack

A

B

C

ALU

(ADD - PUSH)

$C := A + B$

| | |
|---|---|
| PUSH | A |
| PUSH | B |
| ADD | |
| POP | C |

| | | |
|---|---|---|
| Stack | - | Speicher nach dem LIFO Prinzip (last in - first out) |
| SP | - | stack pointer (Zeiger auf zuletzt eingetragene Daten) |
| PUSH | - | Operand vom Speicher in den Stack transportieren<br>automatisches pre-dekrement |
| POP | - | Operand vom Stack in den Speicher transportieren<br>automatisches post-inkrement |

# Akkumulator-Architektur (Ein-Adress-Maschine)

Speicher

B

A

C

(ADD)

LOAD

STORE

ACCU

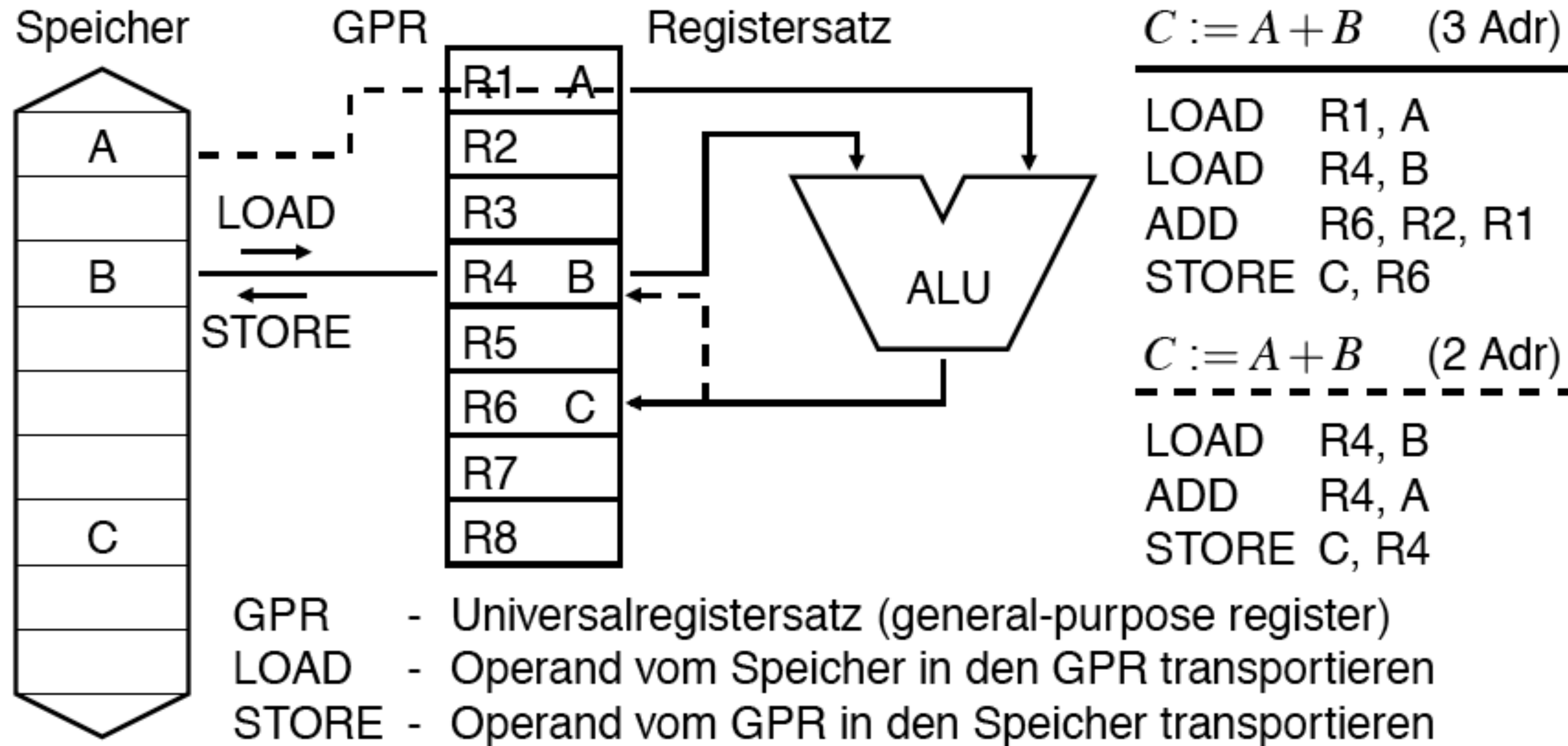
ALU

$C := A + B$

| | |
|---|---|
| LOAD | A |
| ADD | B |
| STORE | C |

ACCU - Akkumulator (Universalregister)
LOAD - Operand vom Speicher in den ACCU transportieren
STORE - Operand vom ACCU in den Speicher transportieren

# Universalregister-Architektur (Zwei/Drei-Adress-Maschine)



# Vor- und Nachteile der Architekturtypen

| Typ | Vorteile | Nachteile |
|---|---|---|
| Stack | einfachstes Modell, gute Codedichte | kein direkter Zugriff auf Stack, nur relativ zum SP $\longrightarrow$ Engpass |
| ACCU | kurze Befehle minimale Hardware | ACCU ist einziger temporärer Speicher, höchster Speicherverkehr $\longrightarrow$ Engpass |
| GRP | allgemeinstes Modell, Zwischenspeicherung der Operanden | alle Operanden explizit adressieren, lange, komplexe Befehlswörter, schlechte Codedichte |

# Motivation für Universalregister-Architekturen

1. Register erlauben einen schnelleren Zugriff auf die Operanden.
2. Einfachere Adressierung der Register (kurze Adresslängen).
3. Einbeziehung der Spezialregister in den Universalregistersatz.
4. Nutzbar als zusätzliche Ebene in der Speicherhierarchie.
5. Vielfältige Möglichkeiten der Zwischen-speicherung von Operanden.
6. Für Compiler einfacher und effektiver nutzbar (z.B. Variablenübergabe).

# Explizit im Befehl adressierte Operanden I

Grundsätzlich werden unterschieden:

**monadische Operation:** (unäre, einstellige Operation) $B := op\ A$

**dyadische Operation:** (binäre, zweistellige Operation) $C := A\ op\ B$

Für eine zweistellige Operation (Verknüpfung $\ll op \gg$ von zwei Operanden zu einem Resultat) sind mindestens folgende Angaben erforderlich:

- Art der Operation $\rightarrow op$
- Adresse des 1. Operanden (1. Quelloperand) $\rightarrow A$
- Adresse des 2. Operanden (2. Quelloperand) $\rightarrow B$
- Adresse für das Resultat (Zieladresse) $\rightarrow C$

# Explizit im Befehl adressierte Operanden II

Folgende zusätzliche Angaben sind bei vollständiger Beschreibung eines Befehls mit Programmverzweigungen noch denkbar:

- Adresse des 1. Folgebefehls ohne Verzweigung $\rightarrow NI0$
- Adresse des 2. Folgebefehls bei Verzweigung $\rightarrow NI1$

Operanden und Resultate (Adressen und Befehle) stehen entweder in prozessorinternen temporären Speichern, z.B. Registern oder im Hauptspeicher. Beide werden durch Adressen angesprochen.

Die Codierung aller Angaben in einem Befehlswort führt zu einen 5-Adress-Befehl. Der Operandenteil umfaßt dabei die 1. und 2. Quelladresse der Operanden, die Zieladresse für das Resultat und die beiden Adressen der möglichen Folgebefehle.

# 5-Adress-Befehlsformat

| Operationsteil | Operandenteil (Adressteil) | | | | |
|---|---|---|---|---|---|
| Opcode | 1. Quelladr. | 2. Quelladr. | Zieladr. | 1. Folgebef. | 2. Folgebef. |
| $op$ | $A$ | $B$ | $C$ | $NI0$ | $NI1$ |

**Beispiel**

| | | | | |
|---|---|---|---|---|
| Opcode | : | 8 Bit | → | 256 verschiedene Befehle codierbar |
| Adressraum | : | 32 Bit | → | 4 GByte Speicher adressierbar |
| Befehlswortlänge | : | 168 Bit | → | 6 32-Bit-Worte pro Befehl |

Übliche Befehlswortlängen z.B. 32 . . . 64 Bit (→ 168 Bit sehr uneffektiv)

→ Reduzierung des Operanden-/Adressteils (Erhöhung der Codedichte)

# Massnahmen zur Reduzierung des Operandenteils

- Befehlszähler: enthält die Adresse für den unmittelbaren Folgebefehl
- Verzweigungs-/Sprungbefehle: enthalten Adressen der Folgebefehle
- implizite Adressierung: Quell oder Zieladressen implizit im Operationsteil
- überdeckte Adressierung: gleichzeitige Nutzung als Quell-/Zieladresse
- Direktoperanden: Operanden werden direkt im Operandenteil codiert
- Registeradressen: gesonderter Adressraum, wesentlich kürzere Adressen
- mehrstufige Adressierung: Umrechnung der Quell- und Zieladressen

# Übliche Befehlsformate

## 3-Adress-Befehlsformat

| Operationsteil | Operandenteil (Adressteil) | | |
|---|---|---|---|
| Opcode | 1. Quelladresse | 2. Quelladresse | Zieladresse |
| $op$ | $A$ | $B$ | $C$ |

$C := A\ op\ B$ alle Quell- und Zieladressen explizit adressiert

## 2-Adress-Befehlsformat

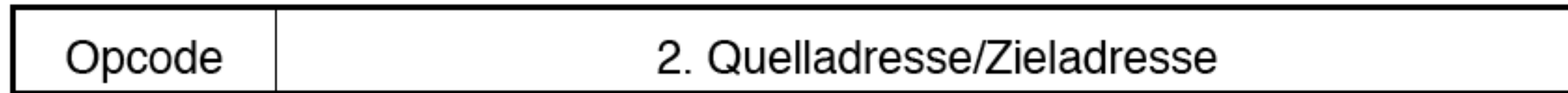
| Operationsteil | Operandenteil (Adressteil) | |
|---|---|---|
| Opcode | 1. Quelladresse | 2. Quelladresse/Zieladresse |
| $op$ | $A$ | $C$ |

$C := C\ op\ A$ C - überdekt adressiert

# Übliche Befehlsformate II

## 1-Adress-Befehlsformat

Operationsteil | Operandenteil (Adressteil)

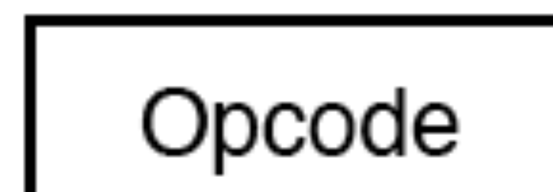



$op$ | $C$

$ACCU := ACCU \; op \; C$ | ACCU - implizit, C-überdeckt adressiert

## 0-Adress-Befehlsformat

Operationsteil

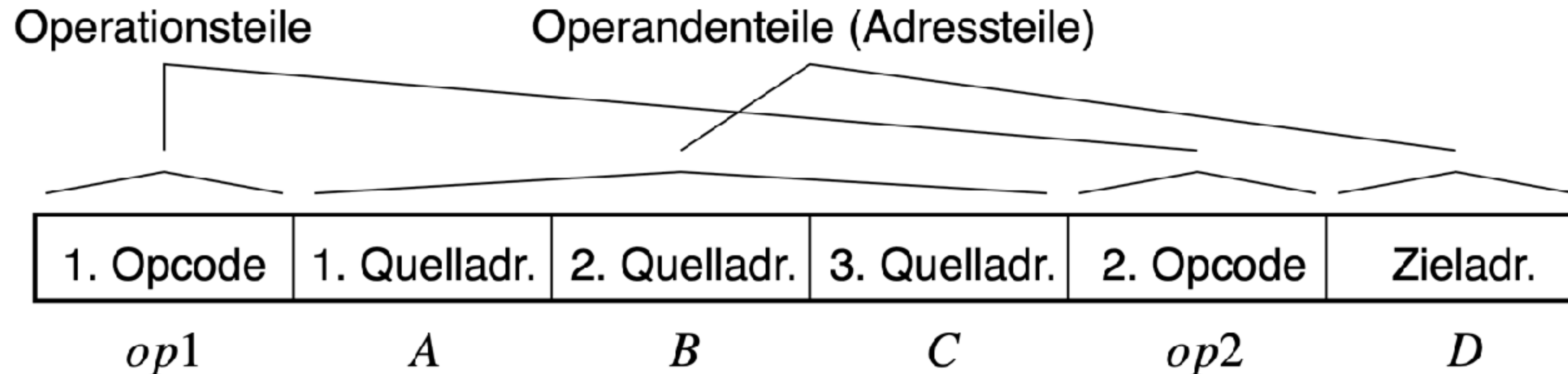



kein Operandenteil (Adressteil)

$op$

alle Quell- und Zieladresssen implizit adressiert

# VLIW-Befehlsformat

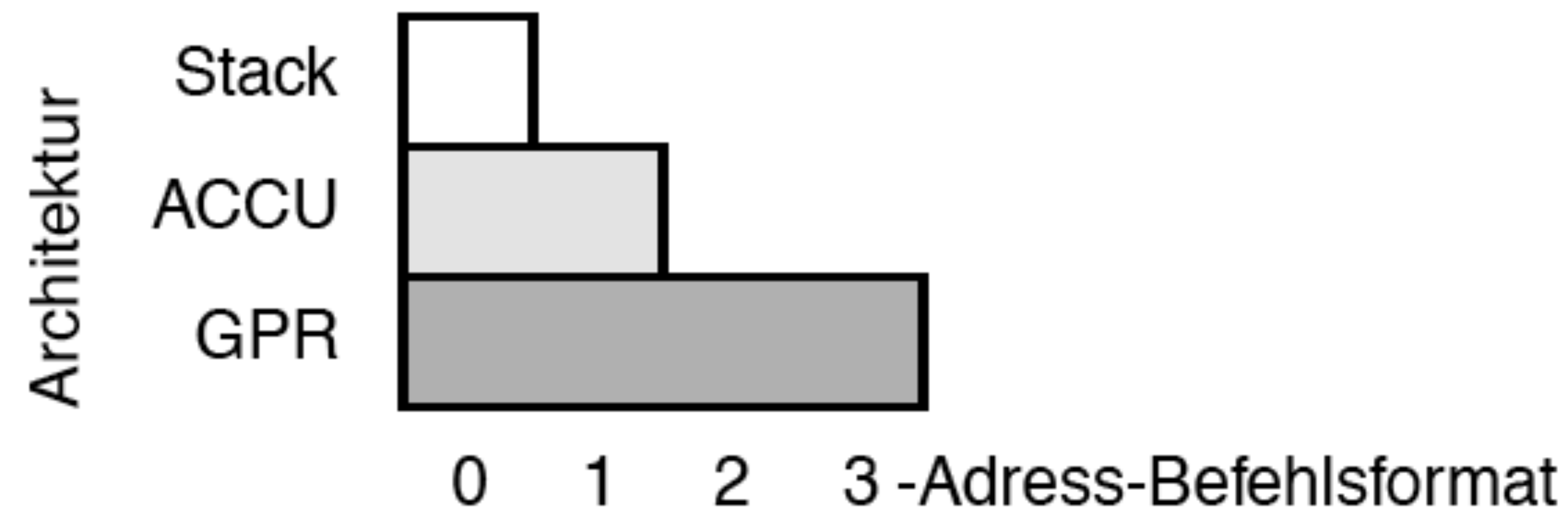



Aus mehreren Einzelbefehlen zusammengesetzter komplexer Befehl

kein festes Befehlsformat, an Operation und Architektur angepasst

speziell für die Nutzung von Parallelität, (z.B. DSP-Architekturen)

# Befehlsformat - Architekturtyp




Die GPR-Architektur (3-Adress-Maschine) ist bzgl. der verschiedenen Befehlsformate am universellsten, alle Formate sind prinzipiell möglich.

Bei der Stack-Architektur (0-Adress-Maschine) ist die Universalität nicht direkt sichtbar, Organisation und Funktionalität des Stacks im Zusammenhang mit der ALU sind entscheidend.

# Aufgaben ISA I — Überblick Architekturen

1. Was ist ein Befehlssatz?
2. Was unterscheidet CISC und RISC voneinander?
3. Was sind die Unterschiede zwischen Stack-, Akkumulator, und Universalregister-Architektur?
4. Was sind 2-, und 3-Adressmaschinen?

# Aufgaben ISA II — Adressmaschine

5. Folgendes Programm für eine 3-Adressmaschine ist gegeben:

```
ADD   R3,R1,R2
SUB   R4,R1,R2
MUL   R1,R3,R4
MUL   R3,R1,R1
MUL   R3,R3,R1
```

Assemblersyntax: `<Operation><Ziel><Quelle1><Quelle2>`

a) Übersetzung auf eine 2-Adressmaschine?
b) Übersetzung auf eine Akkumulator-Architektur?
c) Ergänzung der notwendigen Anweisungen bei der 2- und 3-Adressmaschine, wenn alle Operanden zuerst aus dem Hauptspeicher geladen werden müssen und das Ergebnis dort wieder abgelegt wird. Anzahl der notwendigen Befehle?

# Aufgaben ISA III — Stackarchitektur

6. Eine 8-Bit-Stackarchitektur mit 12-Bit-Adressraum ist gegeben:

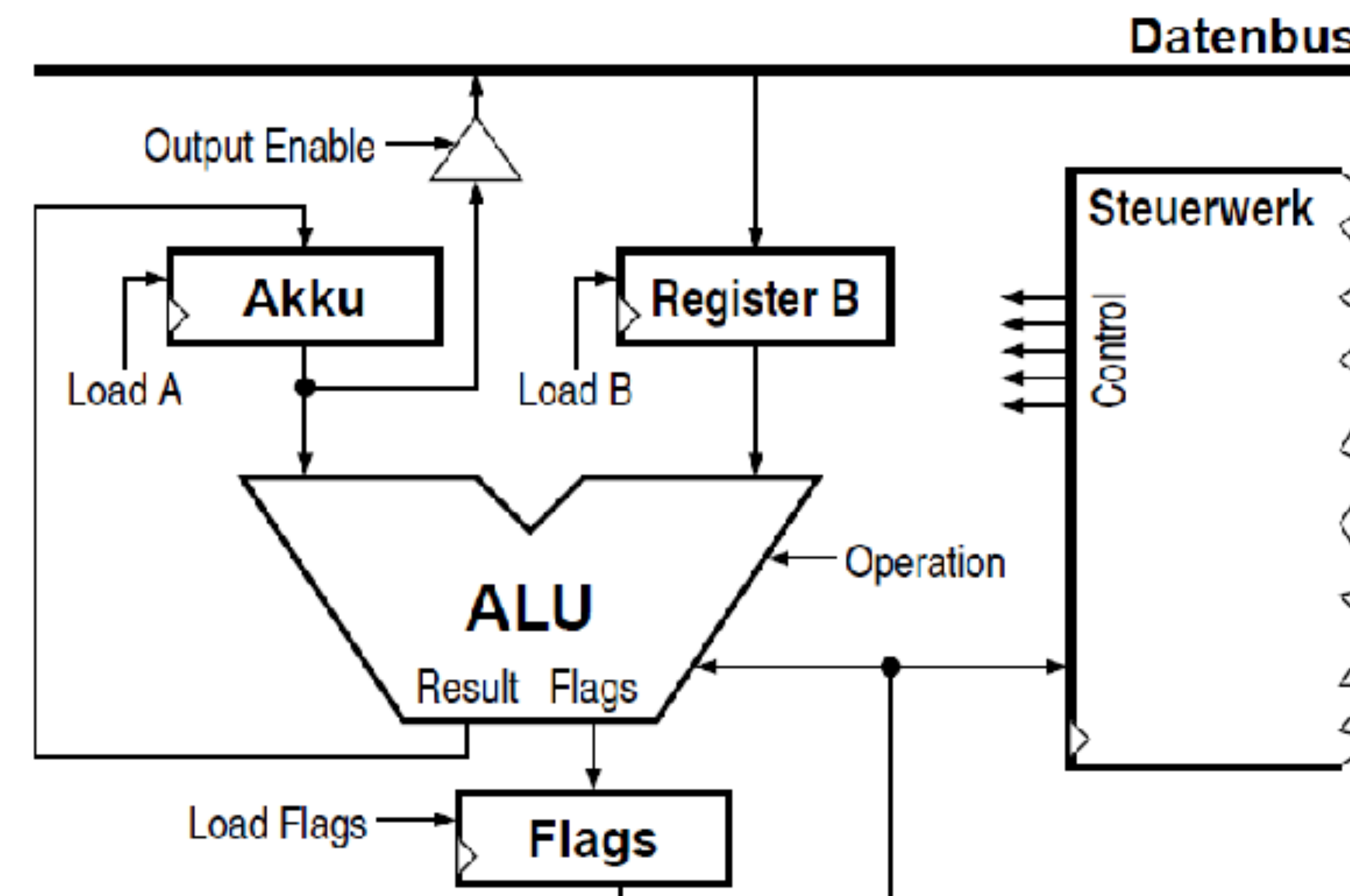
| Mnemonik | Maschinencode | Beschreibung |
|---|---|---|
| NOP | 0000 0000 | Tue nichts. |
| POP | 0001 0000 | Verwerfe oberstes Stackelement. |
| AND | 0001 0001 | Nehme oberen beiden Werte vom Stack, UND-verknüpfe sie bitweise und lege das Ergebnis wieder auf den Stack. |
| OR | 0001 0010 | Nehme oberen beiden Werte vom Stack, ODER-verknüpfe sie bitweise und lege das Ergebnis wieder auf den Stack. |
| XOR | 0001 0011 | Nehme oberen beiden Werte vom Stack, XOR-verknüpfe sie bitweise und lege das Ergebnis wieder auf den Stack. |
| ADD | 0001 0100 | Nehme oberen beiden Werte vom Stack, addiere sie und lege das Ergebnis wieder auf den Stack. |
| PUSH #i | 1100 <i> | Lege vorzeichenerweiterte 4-Bit-Konstante *i* auf den Stack. |
| DUP | 1101 1101 | Dupliziere obersten Stackwert noch einmal auf den Stack. |
| PUSH [A] | 1110 <A.hi>, <A.lo> | Kopiere Speicherinhalt von Adresse *A* auf den Stack. |
| POP [A] | 1111 <A.hi>, <A.lo> | Verschiebe obersten Stackwert an die Speicheradresse *A*. |

(a) Wie lassen sich das Einer- bzw. das Zweierkomplett des obersten Stackelementes $x$ bilden? Geben Sie die erforderlichen Befehlssequenzen in Mnemoniken und im hexadezimalen Maschinencode an! Dokumentieren Sie den Stackzustand!

(b) An den Speicheradressen 0x230 ...0x232 liegen die Werte der Variablen $a$, $b$ und $c$ in dieser Reihenfolge. Geben Sie wie oben ein Programm zur Berechnung von $y = 3(a - b) - c$ an! Das Ergebnis $y$ soll abschließend an der Adresse 0x240 abgelegt werden.

# Aufgaben ISA IV — Akkumulatorarchitektur I

7. Gegeben ist folgendes Rechenwerk mit Akkumulator:




(a) Wie viele Operandenadressen enthält ein typischer arithmetischer Befehl einer solchen Akkumulatorarchitektur?

(b) Auf die Angabe welcher Adressen kann durch welche Adressierungsmodi verzichtet werden?

(c) Beschreiben Sie den zeitlichen Ablauf der Ausführung des Befehls `ADD [0x200]` *im Rechenwerk*! Der Speicher führt gerade den Speicherzugriff aus, der Speicheroperand liegt auf dem Datenbus. Welche Operation führt die ALU aus?

(d) Wie sieht im Vergleich dazu der Ablauf für den Befehl `LOAD [0x240]` aus?

(e) Wozu verwendet das Steuerwerk die im Rechenwerk gebildeten Flags?

# Adressierung

# Adressierung

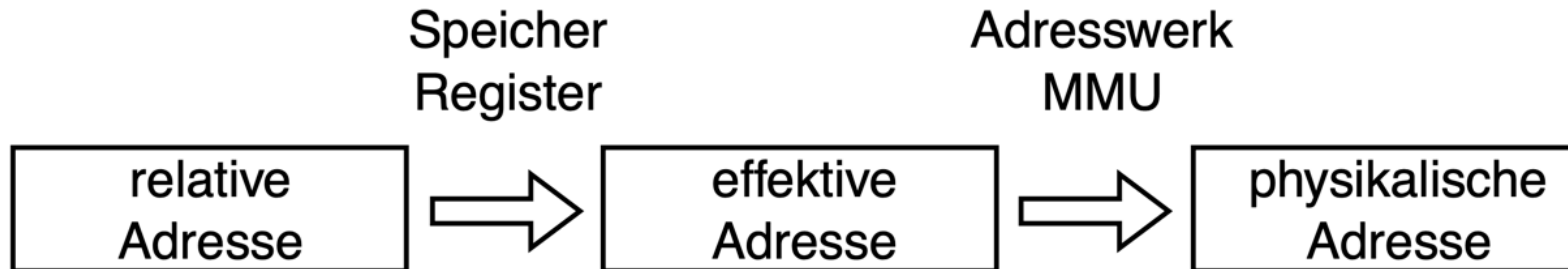
Operanden, Adressen und Befehle können im Hauptspeicher oder im Registerspeicher stehen. Sie werden über ihre Adressen angesprochen, die aus verschiedenen Komponeten zusammengesetzt sein können.

**Adressen:** Adressen beziehen sich auf Konstante (Direktoperand), Register (Registeradresse) oder den Hauptspeicher (Speicheradresse).

**Statische, absolute Adressierung (physikalische Adressen):** Die physikalischen Adressen werden fest, statisch zur Programmierzeit angegeben. Programme und Daten sind vollständig lageabhängig.

**Dynamische, relative Adressierung (effektive Adressen):** Die effektive Adresse wird erst zur Laufzeit durch eine Adressrechnung gewonnen. Im Adresswerk werden dann aus den effektiven Adressen die eigentlichen physikalischen Adressen für die Adressierung gebildet.

# Relative Adressierung




**Formale Notation der Adressrechnung**

| | | | |
|---|---|---|---|
| Hauptspeicheradresse | (Assembler) | : | A |
| Registeradresse | (Assembler) | : | RA |
| Direktoperand | (Assembler) | : | # Operand |
| Hauptspeicherinhalt von A | (Assembler) | : | (A) oder @(A) auch ((A)) |
| Hexadezimalwert | (Assembler) | : | $ HEX |
| Hauptspeicherinhalt von A | | : | M[A] |
| Registerinhalt von RA | | : | RA |

# Adressierungsarten

Adressierungsarten sind alle Möglichkeiten eines Prozessors aus relativen Adressen effektive/physikalische Adressen zu berechnen (zur Laufzeit).

**Vorteile des effektiven Einsatzes der Adressierungsarten:**

- Einsparung von Hauptspeicherplatz, Rechenzeit und Programmierzeit
- Entlastung des Programmierers von aufwendiger Adressrechnung
- Lageunabhängigkeit der Daten und Programme (relative Adressierung)
- Wiederverwendbarkeit von Programmteilen (Unterprogrammtechnik, ...)
- Ermöglichung der wiederholten Befehlsausführung auf verschiedene Daten (z.B. Tabellen, Schleifen), sowie bedingten Operationen

# Adressierungsarten

| Addressierung | Beschreibung |
|---|---|
| implizit | Adressen, Operanden sind druch den Opcode selbst festgelegt |
| immediate | Operand wird im Befehl explizit mitgeführt (Direktoperand) |
| direkt | Adresse wird im Befehl explizit mitgeführt (Direktadresse) |
| indirekt | Befehl enthält die Adresse des Speicherplatzes in dem sich die eigentliche Adresse befindet (Adresse von Adresse) |
| relativ (based) | Befehl enthält einen Offset (Verschiebung) mit dem die Adresse relativ zu einer Basisadresse (Basisregister) gebildet wird |
| indiziert | Befehl enthält eine Basisadresse, die durch Addition eines Index (Indexregister) modifiziert wird |
| segmentiert | Adresse wird an den Inhalt eines Segmentregisters angehängt (concatenate, Seitenadressierung) |
| virtuell | Umsetzung einer virtuellen Adresse in eine physikalische |

# Übersicht: Befehlt – Register - Speicher

| Adressieurngsart | Befehl | | Register | | Speicher |
|---|---|---|---|---|---|
| implizit | Befehl | → | Operand | | |
| immediate | Befehl | | | | |
| | Operand | | | | |
| direkt (Register) | Befehl | → | Operand | | |
| direkt (Speicher) | Befehl | | | | |
| | Adresse | → | | → | Operand |
| indirekt (Register) | Befehl | → | Adresse | → | Operand |

# Übersicht: Befehlt – Register – Speicher II

| Adressieurngsart | Befehl | Register | Speicher |
| --- | --- | --- | --- |
| indirekt (Speicher) | Befehl | | Adresse |
| | Adresse | | Operand |
| relativ | Befehl | Basisadresse | |
| | Verschiebung | | Operand |
| indiziert | Befehl | Index | |
| | Adresse | | Operand |
| segmentiert | Befehl | Segmentadr. | |
| | Adresse | | Operand |

# Gebräuchliche Adressierungsarten

Adressierungsarten

- Register-Adressierung
  - implizit — Flag
  - explizit
- einstufige Speicher-Adressierung
  - immediate
  - direkt
    - absolut
    - Zero-Page
    - Seite(-Register)
  - Register-indirekt
  - indiziert
    - Speicher-relativ
    - Register-relativ
    - Register-relativ mit Index
  - PC-relativ
- zweistufige Speicher-Adressierung
  - indirekt absolut
  - indirekt Register-indirekt
  - indirekt indiziert
    - Speicher-relativ
    - Register-relativ
    - Register-relativ mit Index
  - indiziert indirekt
  - indirekt PC-relativ

# Beispiele zur Adressierung

| Adressierungsart | Befehl (Assembler) | Ergebnis (Zuweisung) |
|---|---|---|
| immediate | ADD R2, #7 | $R2 := R2 + 7$ |
| direkt | ADD R2, R3 | $R2 := R2 + R3$ |
| indirekt (Register) | ADD R2, (R3) | $R2 := R2 + M[R3]$ |
| indirekt (Speicher) | ADD R2, @(R3) | $R2 := R2 + M[M[R3]]$ |
| relativ (Displacement) | ADD R2, 10(R3) | $R2 := R2 + M[10 + R3]$ |
| relativ-indiziert | ADD R2, (R3+R4) | $R2 := R2 + M[R3 + R4]$ |

# Beispielbefehlssatz

# Operationen des Befehlssatzes

Bezüglich der Funktionalität sind vier Klassen von Befehlen unterscheidbar:

| Operationstyp | Beispiel |
|---|---|
| Datenübertragung | Register-Register<br>Register-Speicher<br>Register-E/A |
| Datenmanipulation | arithmetische Operationen<br>logische Operationen<br>Schiebe-/Rotationsoperationen |
| Verzweigungen (bedingt, unbedingt) | Sprung- und Verzweigungsoperationen<br>Prozedurrufe und -rückkehr<br>Traps |
| Systemsteuerung | Betriebssystemrufe<br>Speicherverwaltung<br>Interrupts, Traps |

# Datenübertragung (Beispiele)

| Befehl | Bedeutung | |
|---|---|---|
| LD | Laden eines Registers | load |
| ST | Speichern des Inhalts eines Registers | store |
| MOVE | Übertragung (beliebige Richtung) | move |
| EXC | Vertauschen der Inhalte | exchange |
| TFR | Übertragung eines Registers in ein anderes | transfer |
| PUSH | Ablegen eines oder mehrerer Register in den Stack | push |
| POP | Laden eines oder mehrerer Register aus dem Stack | pop |
| READ | Lesen des Prozessor-Statusregisters | read |
| WRITE | Schreiben des Prozessor-Statusregisters | write |
| IN | Laden eines Registers aus einem Peripheriebaustein | input |
| OUT | Übertragen eines Registers in einen Peripheriebaustein | output |

# Datenmanipulation, arithmetisch Befehle (Beispiele)

| Befehl | Bedeutung | |
|---|---|---|
| ABS | Absolutbetrag | absolute |
| ADD | Addition | addition |
| SUB | Subtraktion | substract |
| MUL | Multiplikation | multiply |
| DIV | Division | devide |
| COM | Einerkomplement | complement |
| NEG | Vorzeichenwechsel, Zweierkomlement | negate |
| CLR | Löschen | clear |
| CMP | Operandenvergleich | compare |
| DAA | Dual-Dezimal-Umwandlung | decimal adjust |
| DEC | Dekrement | decrement |
| INC | Inkrement | increment |

# Datenmanipulation, boolesche Befehle (Beispiele)

| Befehl | Bedeutung | |
|---|---|---|
| AND | AND-Verknüpfung | and |
| OR | OR-Verknüpfung | or |
| EOR | XOR-Verknüpfung | exclusive or |
| NOT | NOT-Verknüpfung | not |

# Datenmanipulation, Flag- und Bit-Befehle (Beispiele)

| Befehl | Bedeutung | |
|---|---|---|
| SEF | Setzen eines Bedingungs-Flag | flag set |
| CLF | Löschen eines Bedingungs-Flag | flag clear |
| TST | Prüfen eines Flags oder Bits | test |
| BSET | Setzen eines Bit | bit set |
| BCLR | Rücksetzen eines Bit | bit clear |
| BCHG | Invertieren eines Bit | bit change |
| BFCLR | Rücksetzen der Bits eines Bitfeldes | clear bits |
| BFSET | Setzen der Bits eines Bitfeldes | set bits |
| BFFFO | Finden der ersten 1 im Bitfeld | find first one |
| BFEXT | Lesen eines Bitfeldes | extract bits |
| BFINS | Einfügen eines Bitfeldes | insert bits |

# Datenmanipulation, String- oder Block-Befehle (Beispiele)

| Befehl | Bedeutung | |
|---|---|---|
| MOVS | Transferieren eines Blocks | move string |
| INS | Einlesen eines Blocks von der Peripherie | input string |
| OUTS | Ausgabe eines Blocks an die Peripherie | ooutput string |
| CMPS | Vergleich zweier Blöcke | compare string |
| COPS | Kopieren eines Blockes | copy string |
| SCAS | Suchen eines Zeichens in einem Block | scan string |

# Datenmanipulation, Schiebe- und Rotation-Befehle (Beispiele)

| Befehl | Bedeutung | |
|---|---|---|
| SHF | Verschieben eines Registerinhlates | shift |
| ASL | arithmetische Links-Verschiebung | arith. shift left |
| ASR | arithmetische Rechts-Verschiebung | arith. shift right |
| LSL | logische Links-Verschiebung | shift left |
| LSR | logische Rechts-Verschiebung | shift right |
| ROT | Rotation eines Registerinhaltes | rotate |
| ROL | Rotation nach links | rotate left |
| ROR | Rotation nach rechts | rotate right |
| SWAP | Vertauschen der beiden Hälften eines Registers | swap |

# Verzweigungen (Beispiele)

| Befehl | Bedeutung | |
|---|---|---|
| JMP | unbedingter Sprung zu einer Adresse | jump |
| BCC | Verzweigen falls Bedingung cc erfüllt | branch |
| BRA | Verzweigen ohne Bedingungsabfrage | branch always |
| CALL, JSR | Sprung in ein Unterprogramm | jump to subroutine |
| BSRCC | JSR, wenn Bedingung cc erfüllt | branch to subr. |
| RTS | Rücksprung aus einem Unterprogramm | return from subr. |
| TRAP, INT | Sprung in Unterbrechungsroutine | software interrupt |
| RTI, RTE | Rücksprung aus Unterbrechungsroutine | return from int. |

# cc-Bedingungen für Verzweigungen (Beispiele)

| cc | Bedingung | Bedeutung |
|---|---|---|
| CS | CF=1 | branch on carry set |
| CC | CF=0 | branch on carry clear |
| VS | OF=1 | branch on overflow |
| VC | OF=0 | branch on not overflow |
| EQ | ZF=1 | branch on zero/equal |
| NE | ZF=0 | branch on not zero/equal |
| MI | SF=1 | branch on minus |
| PL | SF=0 | branch on plus |
| PA | PF=1 | branch on parity/parity even |
| NP | PF=0 | branch on not parity/parity odd |

| cc | Bedingung | Bedeutung |
|---|---|---|
| vorzeichenlose Operanden | | |
| LO | CF=1 | branch on lower than |
| HS | CF=0 | branch on higher or same |
| LS | $CF \vee ZF=1$ | branch on lower or same |
| HI | $CF \vee ZF=0$ | branch on higher than |
| vorzeichenbehaftete Operanden | | |
| LT | $SF \neq OF=1$ | branch on less than |
| GE | $SF \neq OF=0$ | branch on greater or equal |
| LE | $ZF \vee (SF \neq OF)=1$ | branch on less or equal |
| GT | $ZF \vee (SF \neq OF)=0$ | branch on less greater than |

# Systemsteuerung (Beispiele)

| Befehl | Bedeutung | |
|---|---|---|
| NOP | keine Operation | no operation |
| WAIT | Warten auf spezielles Eingangssignal | wait |
| SYNC | Wareten auf einen Interrupt | sync |
| HALT, STOP | Anhalten des Prozessors | stop |
| RESET | Rücksetzsignal für Pripherie | reset |
| SVC | Betriebssystem-Aufruf | supervisor call |

# Operandentypen (Datentypen)

Der Typ der im Befehl adressierten Operanden wird allgemein im Befehl selbst festgelegt. Die Codierung erfolgt dabei im Opcode zusammen mit der durchzuführenden Operation.

Alternativ wird auch die Abspeicherung von Typkennungen (Tag) zusammen mit den Daten verwendet (Datenflußmaschinen, . . . ) verwendet. Sogenannte „Tagged“ Architekturen sind jedoch eher die Ausnahme.

**Numerische Daten:**

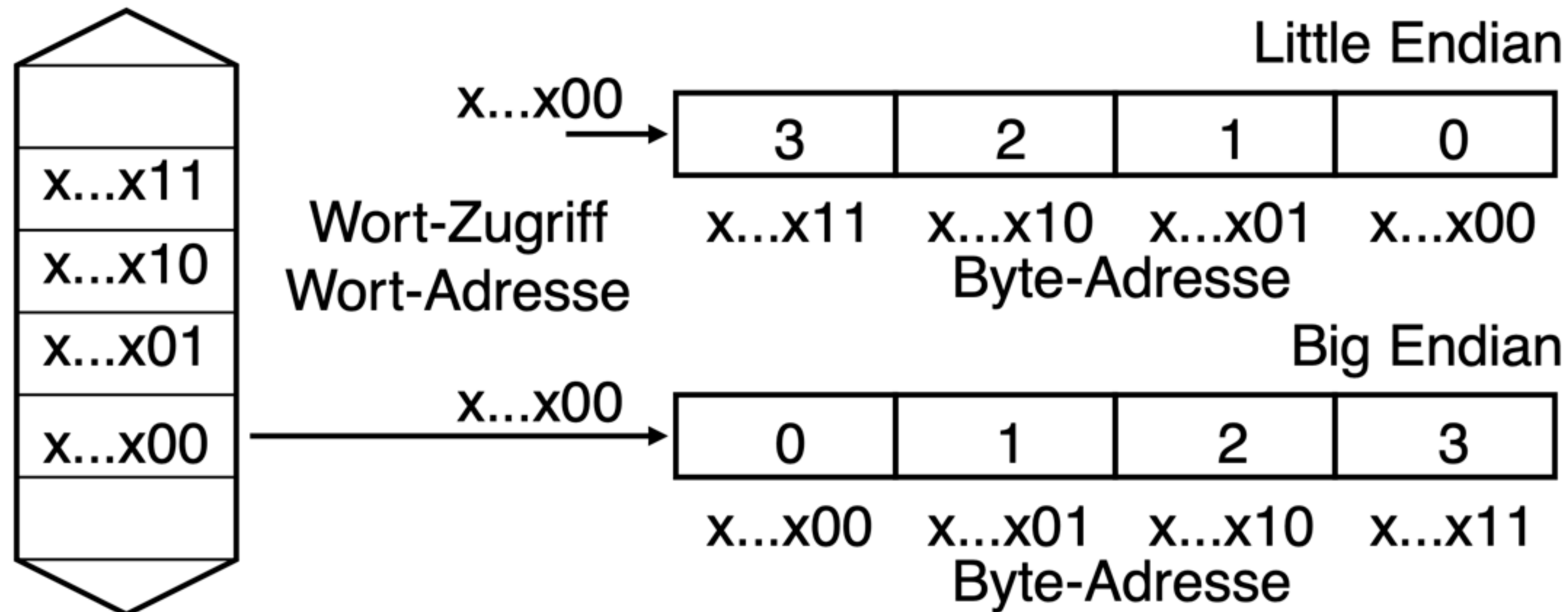
- vorzeichenlose ganze Zahlen (unsigned integer)
- vorzeichenbehaftete ganze Zahlen (signed integer)
- binär codierte Dezimalzahlen (binary coded decimal integer, BCD)
- Gleitkommazahlen (floating point)

# Speicher und Alignment

# Speicheradressierung

**Interpretation von Speicheradressen**

Byte-orientierter Speicherzugriff (kleinste adressierbare Einheit)
→ Probleme bzgl. Byte-Reihenfolge bei Zugriff auf größere Einheiten
(Halbwort-, Wort-, Doppelwort-Zugriff, . . . )



# Vereinbarungen zur Bytereihenfolge in einem Wort . . .

| Bezeichnung | Bedeutung | Beispiel |
| --- | --- | --- |
| little endian byte ordering | Byte mit niederwertigsten Bits an kleinster Speicheradresse | Intel 80x86, DEC VAX |
| big endian byte ordering | Byte mit höhenwertigsten Bits an größerer Speicheradresse | MIPS, SPARC |

Für die Verarbeitung im Rechner ist die Byte-Reihenfolge meist unbedeutend. Probleme allgemein nur bei Byte-Wort-Umrechnungen.

Beim Datenaustausch zwischen Rechnern unterschiedlicher Byte-Reihenfolge ist diese unbedingt zu beachten (ggf. Konvertierungen).

# Byte-Reihenfolge II

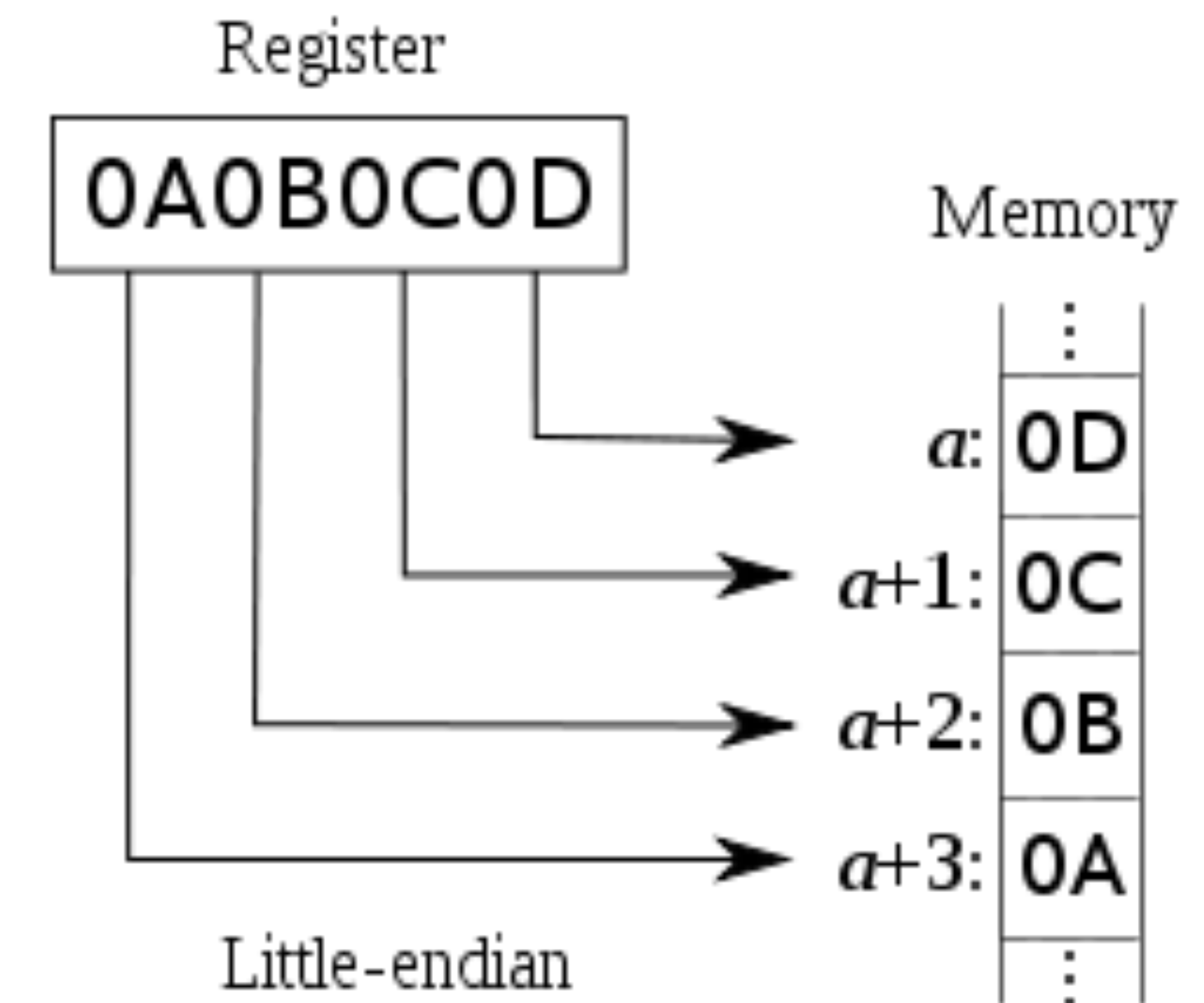
Erstreckt sich ein Datenelement über mehrere Speicherstellen, muss eine Reihenfolge festgelegt sein, in der die einzelnen Bytes abgelegt sind:

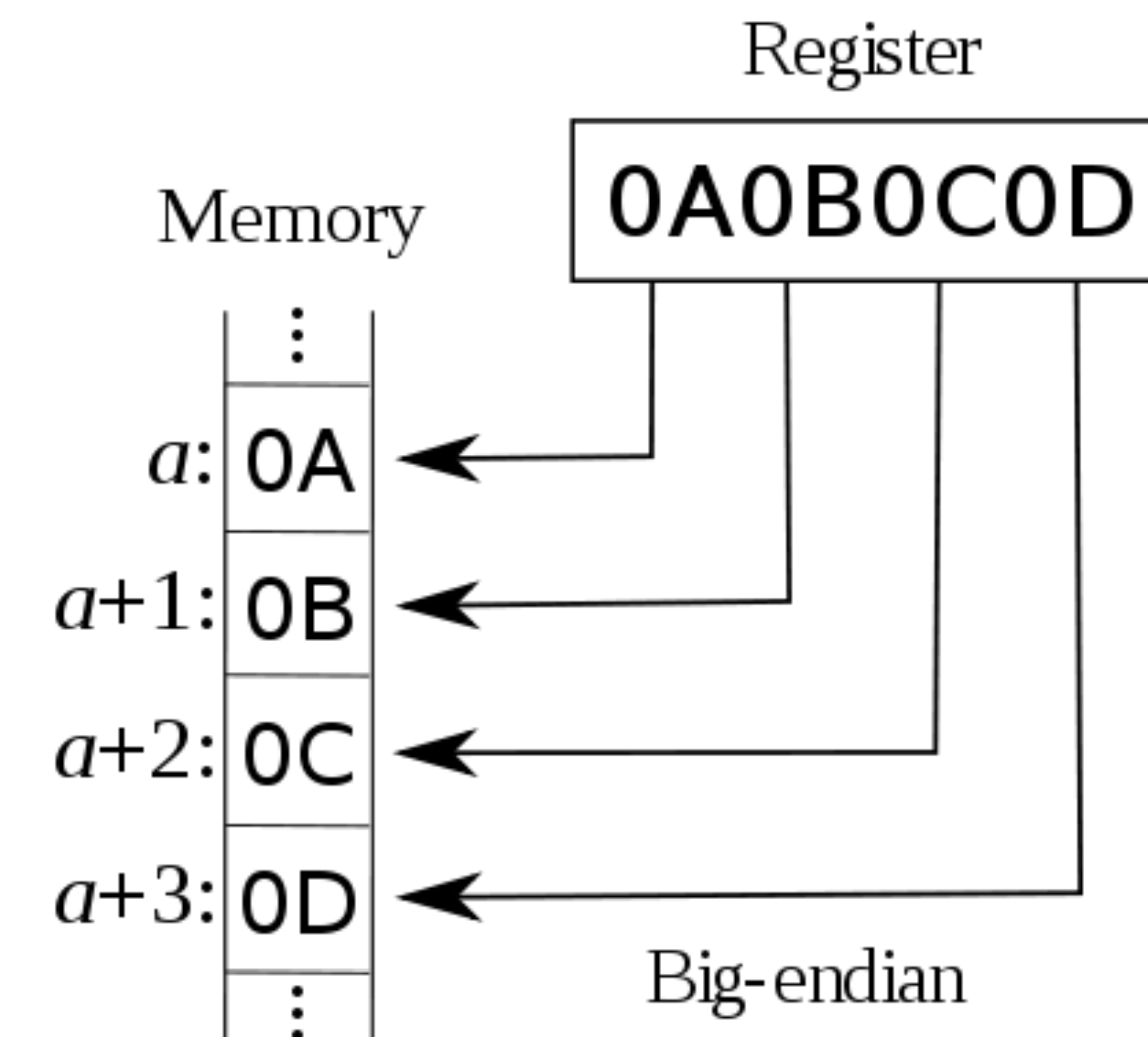
**Little-Endian:**

— Das Byte mit den niederwertigsten Bits wird an der kleinsten Speicheradresse abgelegt.
— Realisiert in Intel x86 und kompatiblen Systemen.

**Big-Endian:**

— Das Byte mit den höherwertigsten Bits wird an der kleinsten Speicherstelle abgelegt.
— Realisiert in IBM PowerPC, MIPS, …

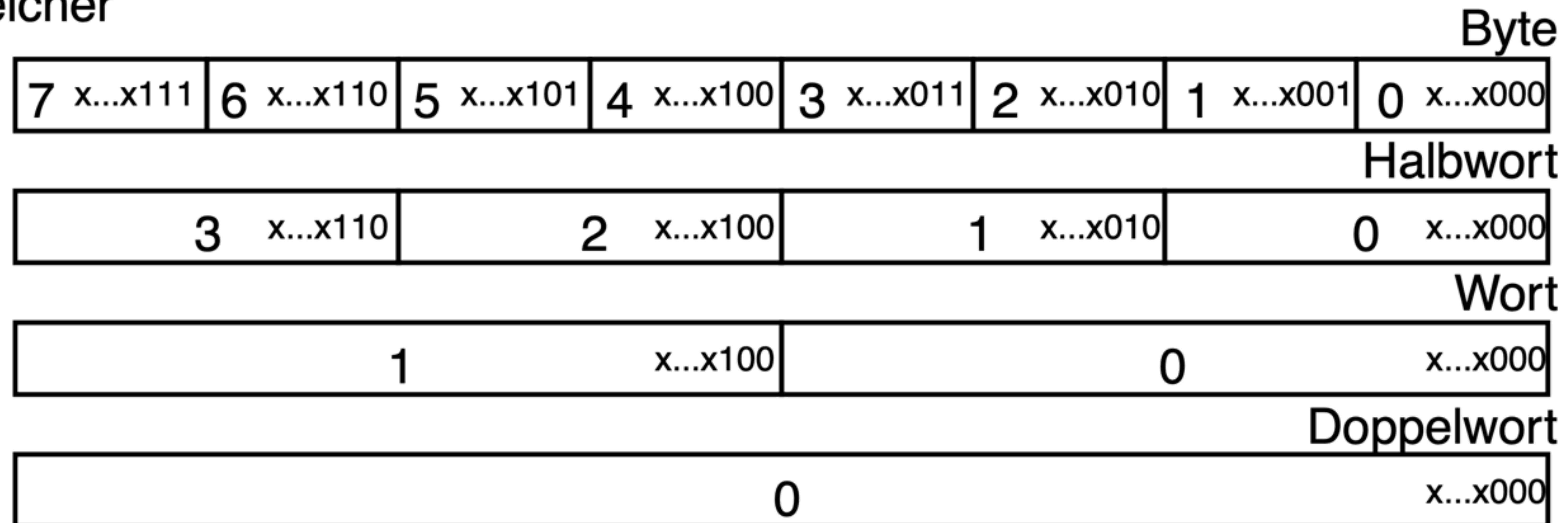


# Speicher-Alignment

Der Zugriff auf Einheiten die größer als ein Byte sind kann ausgerichtet (aligned) oder nicht ausgerichtet (misaligned) erfolgen. Ein Zugriff zu einer Einheit der Länge m Bytes ab der Byte-Adresse A ist ausgerichtet, wenn gilt: $A$ **mod** $m = 0$.

**Ausgerichtete Speicherzugriffe bis Doppelwort-Grenze**



# Speicher-Alignment II

Ein Element mit **m** Bytes ist im Speicher ausgerichtet, wenn dessen Adresse **A** ein ganzzahliges Vielfaches von m ist:

$$A \bmod m = 0$$

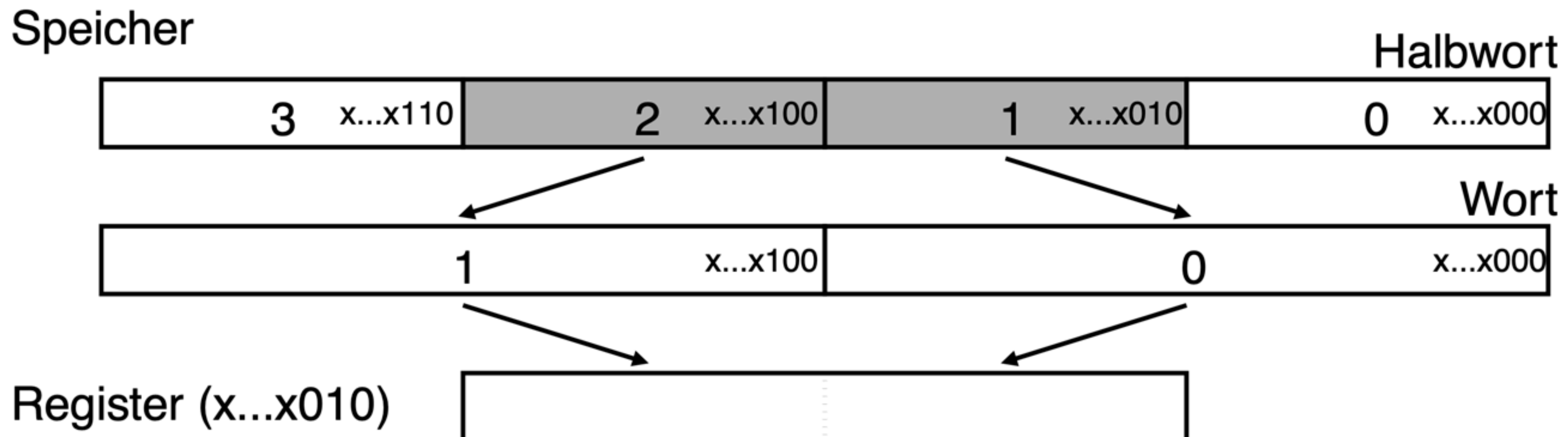
Falls n jedoch keine Potenz von 2 ist, muss für die Berechnung n auf die nächsthöhere Potenz von 2 aufgerundet werden.

- Typischerweise erfolgt eine byteweise Adressierung des Arbeitsspeichers, aber die Ausgabe umfasst bis zu 8 Byte. Es kann also auf eine Folge von mehreren Bytes in einem Takt zugegriffen werden.
- Sind die Operanden im Speicher ausgerichtet, ist für Datenelemente größer als ein Byte nur jeweils ein Speicherzugriff nötig.

# Alignment-Restriktionen I

Soll z.B. auf ein Wort auf einer nicht ausgerichteten Halbwordgrenze (x...x010) zugegriffen werden, so müssen bei ausgerichtetem Wortzugriff die beiden benachbarten Worte (x...x000 und x...x100) eingelesen werden und anschließend die darin enthaltenen Halbworte (x...x010 und x...x100) wieder zu einem Wort zusammengesetzt (ausgerichtet) werden.

**Ausgerichteter Wort-Speicherzugriff auf nichtausgerichtetes Wort**



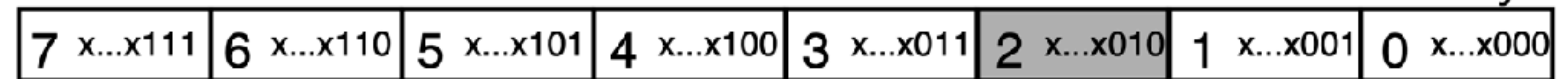
# Alignment-Restriktionen II

Ein nicht ausgerichteter Speicherzugriff erfordert im allgemeinen mehrere ausgerichtete Zugriffe und eine anschließende Ausrichting (Ausrichtungsnetzwerk). Ausgerichtete Zugriffe sind allgemein schneller. Ausgerichtete Zugriffe erfordern bei kürzeren Einheiten als die Registergröße ebenfalls ein Ausrichtungsnetzwerk (Multiplexer-Netzwerk, Barrel-shifter).

**Ausgerichteter Byte-Speicherzugriff bei Wort-Zugriffsbreite**

Speicher

Byte




Wort

Wort-Zugriff (x...x000)

Ausrichtung auf 0. Byte

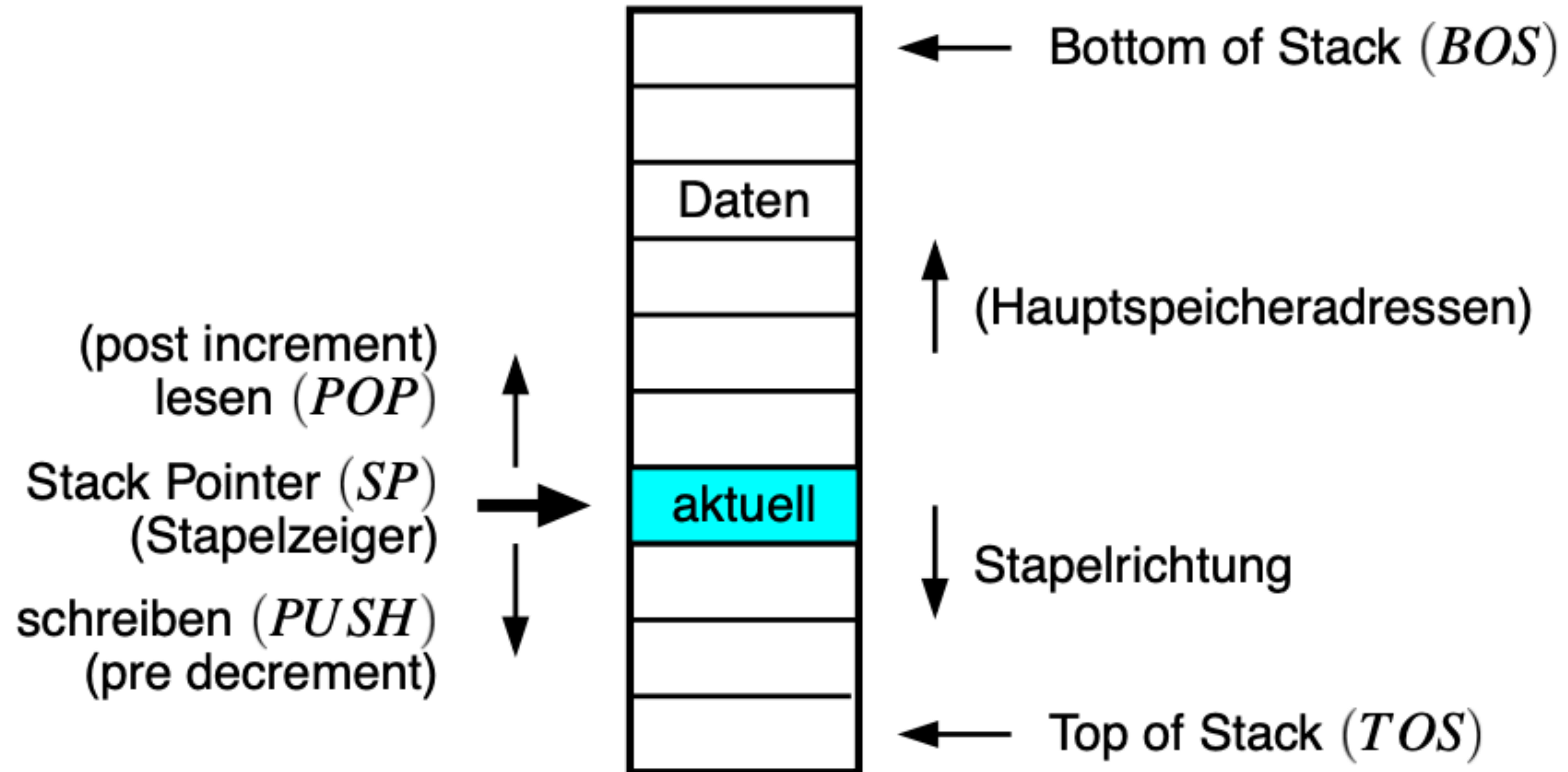
Register (Byte, x...x010)

x...x010

# Stapelspeicher, Stack, Kellerspeicher

- SAM-Speicher (Serial Addressable Memory) nach dem LIFO-Prinzip (Last In First Out). Als Hardware-Stack innerhalb der CPU oder als Software-Stack im Hauptspeicher realisierbar. Des Stack, Stapel, wächst immer nach unten.
- Der Stapelzeiger (Stack Pointer, $SP$) zeigt immer auf das Stapelende, letzte aktuelle Daten. Die Daten im Stack können nicht direkt adressiert werden.
- $PUSH$ – Dekrementieren (erniedrigen) des Stack Pointers und Speichern der Daten auf den Stack (pre decrement).
- $POP$ – Lesen der Daten vom Stack und anschließendes Inkrementieren (erhöhen) des Stack Pointers (post increment).
- Nutzung des Stack zur effektiven Abspeicherung bzw. Zwischenspeicherung von Daten: Prozessorstatus, Unterprogrammparameter, rekursive Programme, Unterbrechungsroutinen, . . .
- Stack-Typen: system stack, data stack, user stack, programm stack, . . .

# Verwaltung des Stapelspeichers



# Aufgaben Alignment I

1. Gegeben ist folgender Speicherauszug aus einem byteweise addressierbaren Speicher:

| Adresse | Inhalt |
|---|---|
| 0x0FD | 0x02 |
| 0x0FE | 0xD5 |
| 0x0FF | 0x68 |
| 0x100 | 0x4F |
| 0x101 | 0x73 |
| 0x102 | 0xC9 |
| 0x103 | 0x1B |

(a) Welche Werte werden bei folgenden Zugriffen von der Speicheradresse 0x100 gelesen?

| Endian | Byte | 16-Bit-Wort | 32-Bit-Wort |
|---|---|---|---|
| Little | | | |
| Big | | | |

(b) An welcher Adresse befindet sich, je nach Endian, das niederwertigste Byte des 32-Bit-Wortes an Adresse 0x100?

# Aufgaben Alignment II

2. Ein Record besteht aus mehreren Datenfeldern für Ganzzahlen unterschiedlicher Bitbreite:

```
record
  a : int16;      // 16 Bit
  b : int8;       //  8 Bit
  c : int16;      // 16 Bit
  d : int32;      // 32 Bit
end record;
```

In einem Programm werde nun eine Instanz dieses Records ab der Adresse `0xA38` im byteadressierten Speicher mit {`.a=0x341A,.b=0x4D,.c=0x90F2,.d=0xA37B672E`} initialisiert. Zeigen Sie den Auszug des belegten Speichers nach dessen Initialisierung, wenn:

(a) das Record dicht in Little-Endian gepackt wird, und

(b) jedes Datenfeld des Records (ohne Umordnung!) gemäß seiner Größe *aligned* (= *natürliches Alignment*) in Big-Endian abgelegt wird!

# Aufgaben Stack

1. Ein Stack wird in der Regel durch eine Speicheradresse repräsentierten Stackpointer (SP) realisiert.
   a) Welche beiden naheliegenden Alternativen Speicherplätze kann der Stackpointer identifizieren?
   b) Wie ist der Stackpointer bei den Operationen `PUSH` und `POP` zu korrigieren?
2. Ein Stack kann auch als Aufrufstack eine Hierarchie von Unterprogrammaufrufen realisieren.
   a) Wie kann auf einem Stack weiterer Speicher für lokale Variablen reserviert werden?
   b) Über welchen Adressierungsmodus kann auf die so alliierten Variablen zugegriffen werden?
   c) Wie können die Variablen wieder frei gegeben werden?
   d) Für welches Aufrufmuster in einem Programm ist eine derartige Vorgehensweise zwingend erforderlich?

# END