

# Kapitel 4

---

## **Befehlssatzarchitektur** *(Instruction Set Architektur ISA)* **Die Hardware-Software-Schnittstelle**

- Datentypen, Datenformate
- Befehlsformat, Befehlssatz
- Adressierungsarten
- Diskussion: RISC & CISC; Fallstudien (MIPS)



## 4.3 Adressierungsarten

---

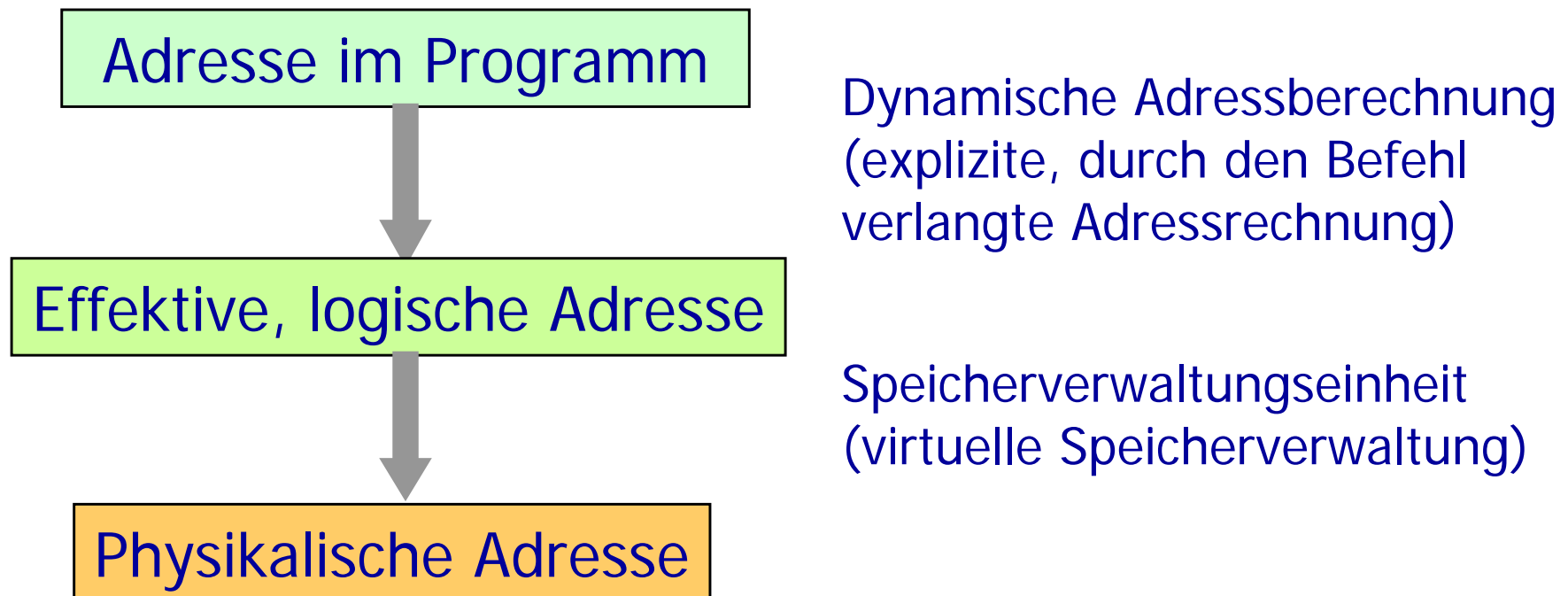
- **Adressierungsarten:** die verschiedenen Möglichkeiten eines Prozessors die Adresse eines Operanden oder eines Sprungziels im Speicher zu berechnen.
- **Früher:** Adresse der Operanden und Sprungziele absolut im Befehl vorgegeben
  - **Nachteile:**
    - absolute Adressen müssen bereits zur Programmierzeit festgelegt werden  
➔ Programme sind lageabhängig im Speicher
    - Bei Tabellenzugriffen im Speicher muss die Adresse im Befehl geändert werden ➔ keine Festwertspeicher als Programmspeicher möglich

# 4.3 Adressierungsarten

## □ Abhilfe:

- Adresse wird zur Laufzeit berechnet (dynamische Adressberechnung)

## Ablauf der Adressberechnung:



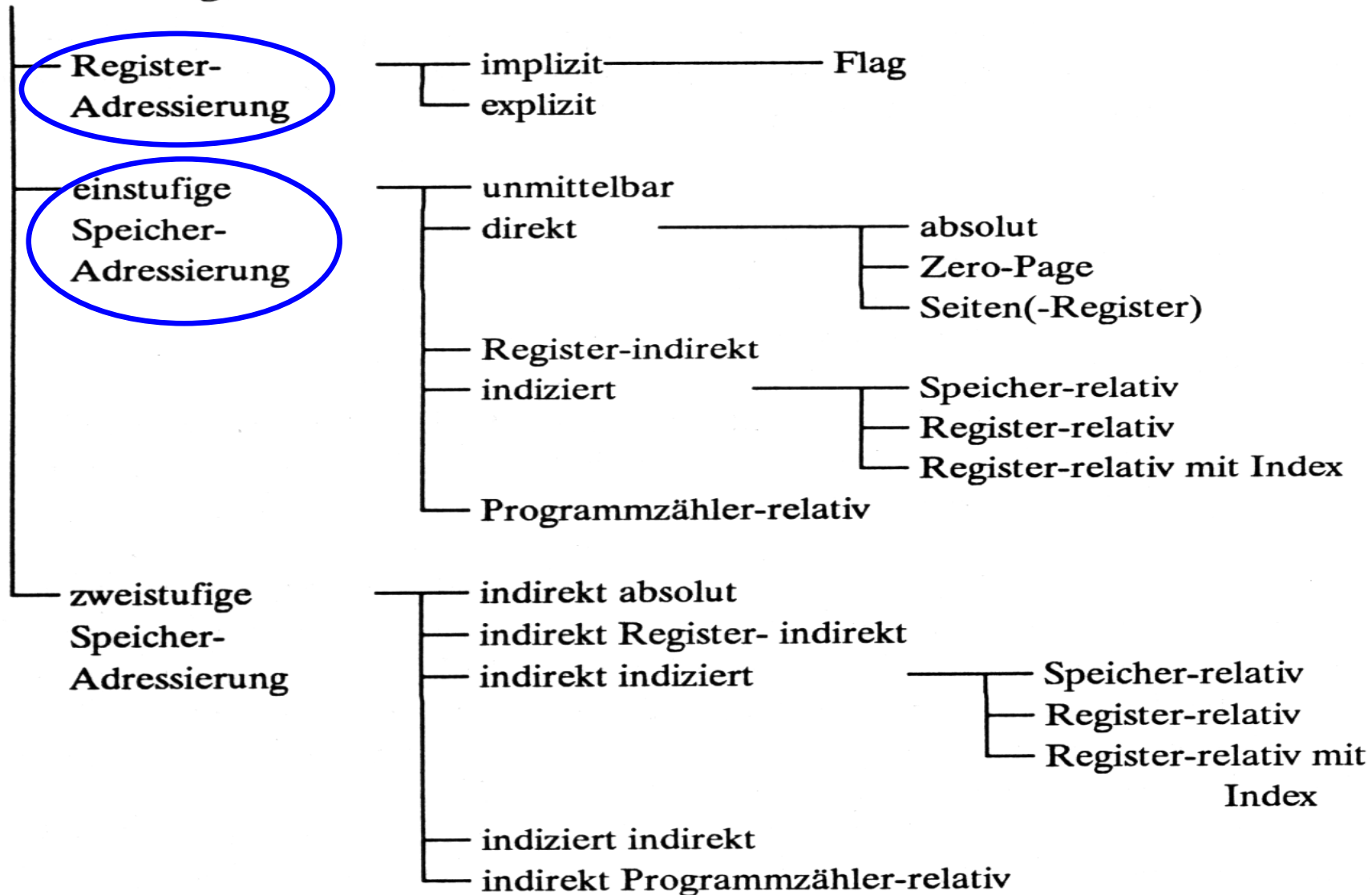
## 4.3 Adressierungsarten

---

- **Effektive Adresse:** die durch die Adressierungsart spezifizierte Speicheradresse im Hauptspeicher
  - Eine effektive Adresse entsteht im Prozessor nach Ausführung der Adressrechnung.
- Bei virtueller Speicherverwaltung gilt:  
**effektive Adresse = logische Adresse**,  
diese wird weiteren Speicherverwaltungsoperationen in einer **Speicherverwaltungseinheit** (*Memory Management Unit MMU*) unterworfen, um letztendlich eine **physikalische Adresse** zu erzeugen, mit der dann auf den Hauptspeicher zugegriffen wird.
- Im folgenden betrachten wir nur die Erzeugung einer effektiven Adresse aus den Angaben in einem Maschinenbefehl.

# 4.3 Adressierungsarten

## Adressierungsarten



# 4.3.1 Register-Adressierung

---

## 4.3.1 Register-Adressierung

Operand steht bereits im Register

→ kein Speicherzugriff erforderlich

## 4.3.2 Einstufige Speicher-Adressierung

Eine Adressberechnung zur Ermittlung der effektiven Adresse notwendig, d. h. keine mehrfachen Speicherzugriffe zur Adressermittlung

## 4.3.3 Zweistufige Speicher-Adressierung

Mehrere sequentielle Adressberechnungen und Speicherzugriffe. Ergebnis der ersten Berechnung liefert die Adresse einer Speicherzelle, deren Inhalt wieder eine Adresse oder ein Offset zur weiteren Berechnung ist

# 4.3.1 Register-Adressierung

---

## 4.3.1 Register-Adressierung

Operand steht bereits im Register →  
kein Speicherzugriff erforderlich

- ❑ Implizite Adressierung
- ❑ Flag-Adressierung
- ❑ Explizite Register-Adressierung

# Implizite Adressierung

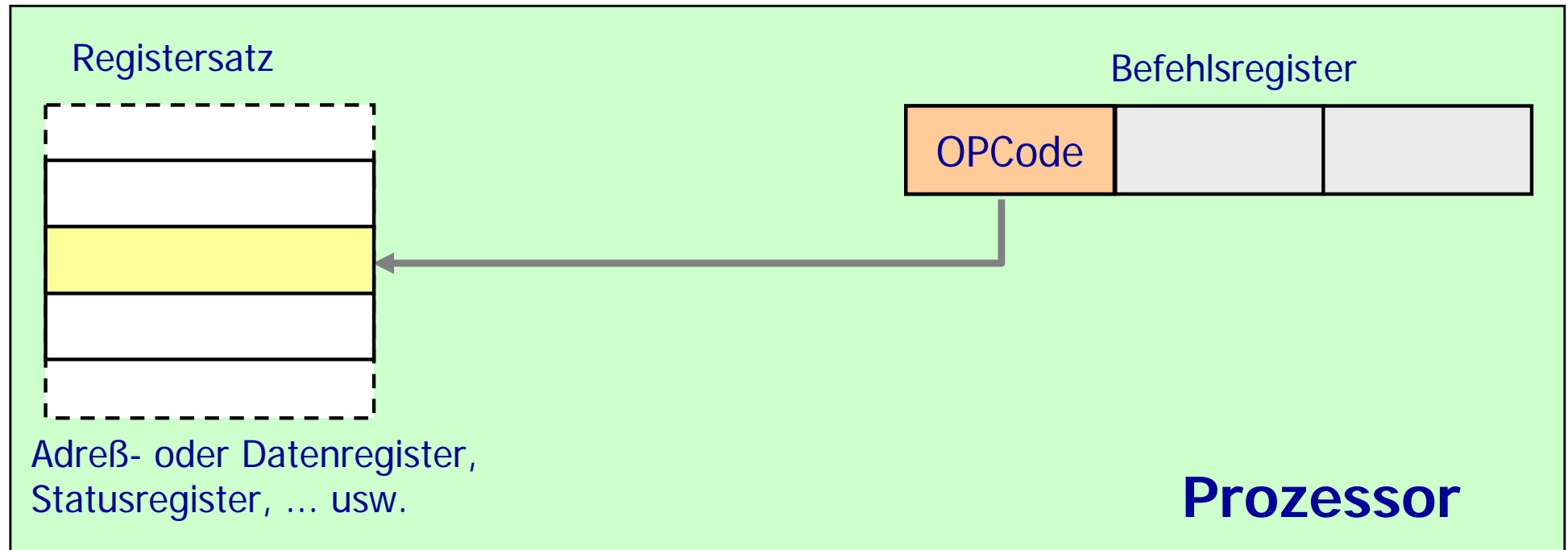
---

- **Implizite Adressierung** (*inherent Adressierung, implied-, inherent addressing*)

Die Nummer, d. h. die effektive Adresse des angesprochenen Registers ist **codiert im Operations-Feld des OpCodes** enthalten

- **Assemblerschreibweise:**  
    <Mnemo> A   (A Akkumulator)
- **Effektive Adresse:**  
    EA ist codiert im OpCode enthalten

# Implizite Adressierung



## Beispiel:

LSRA (*logical shift right accumulator*)

(*Verschiebe den Inhalt des Akkumulators A eine Bitposition nach rechts*)

# Flag-Adressierung

---

## □ Flag-Adressierung:

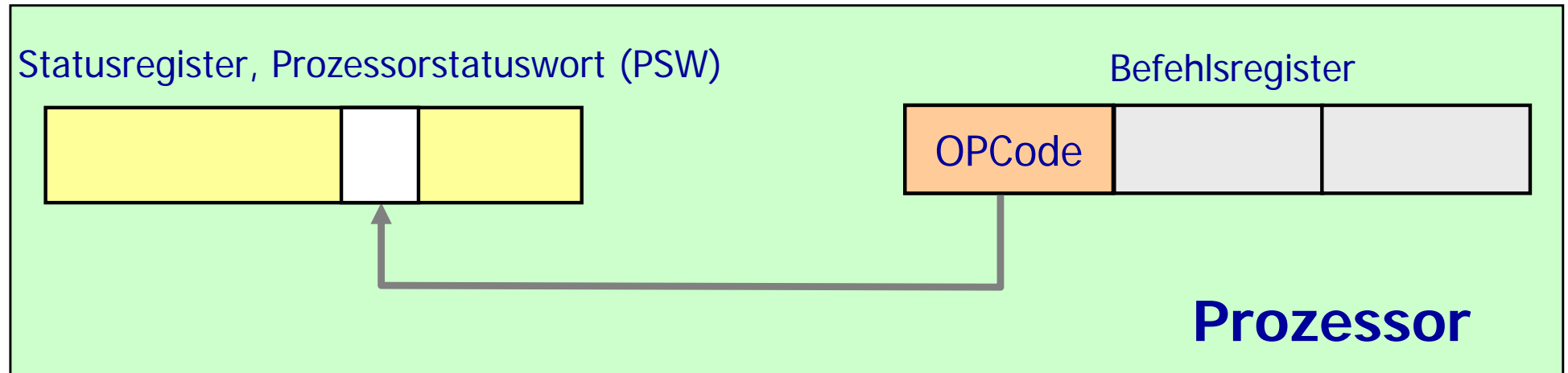
Sie ist ein Spezialfall der impliziten Adressierung. Bei ihr wird nicht ein ganzes Register angesprochen, sondern nur ein einzelnes Bit (Flag) in einem Register

### ➤ **Assemblerschreibweise:**

- SE <flag>            (Flag setzen)
- CL <flag>            (Flag rücksetzen)

### ➤ **Effektive Adresse:** EA ist codiert im OpCode enthalten

# Flag-Adressierung



## Beispiele:

- SEI/CLI *(set / clear interrupt flag)*
  - SEC/CLC *(set / clear carry flag)*
- (Setzen / Zurücksetzen des Interrupt Enable Flags bzw. des Carry Flags.)*

# Explizite Register-Adressierung

---

- **Explizite Register-Adressierung** (*register operand addressing*):

Die Adresse (Nummer) des Registers wird im Operandenfeld des Befehls angegeben

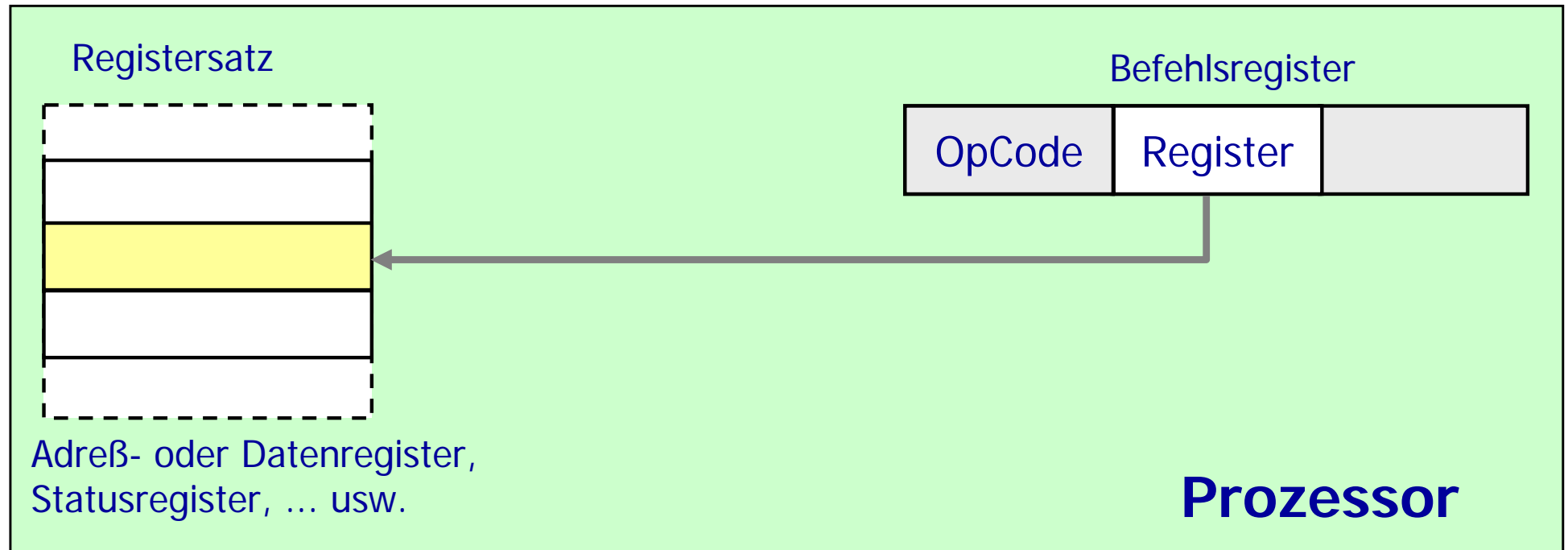
- **Assemblerschreibweise:**            <Mnemo> Ri    (Register i)
- **Effektive Adresse:**                EA = i

## Beispiel:

DEC R0                            (*Decrement R0*)

(*Dekrementiere den Inhalt des Registers R0*)

# Explizite Register-Adressierung



- **Assemblerschreibweise:**  $\langle \text{Mnemo} \rangle \text{ Ri} \quad (\text{Register } i)$
- **Effektive Adresse:**  $\text{EA} = i$
- **Beispiel:**  $\text{DEC R0}$

## 4.3.2 Einstufige Speicher-Adressierung

---

### 4.3.1 Register-Adressierung

Operand steht bereits im Register

→ kein Speicherzugriff erforderlich

### 4.3.2 Einstufige Speicher-Adressierung

Eine Adressberechnung zur Ermittlung der effektiven Adresse notwendig, d. h. keine mehrfachen Speicherzugriffe zur Adressermittlung

### 4.3.3 Zweistufige Speicher-Adressierung

Mehrere sequentielle Adressberechnungen und Speicherzugriffe. Ergebnis der ersten Berechnung liefert die Adresse einer Speicherzelle, deren Inhalt wieder eine Adresse oder ein Offset zur weiteren Berechnung ist

# 4.3.2 Einstufige Speicher-Adressierung

---

## 4.3.2 Einstufige Speicher-Adressierung

Eine Adressberechnung zur Ermittlung der effektiven Adresse notwendig, d. h. keine mehrfachen Speicherzugriffe zur Adressermittlung

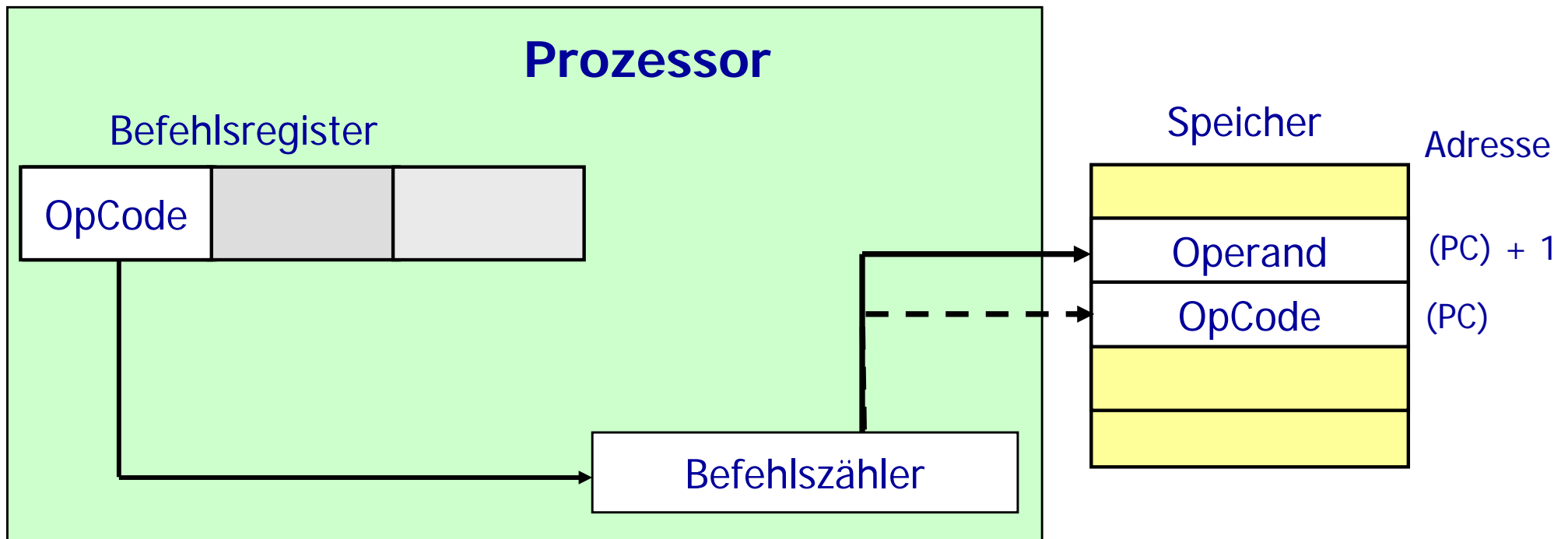
- ❑ Unmittelbare Adressierung (immediate addressing)
- ❑ Direkte Adressierung (direct addressing)
  - Absolute Adressierung
  - Seiten-Adressierung
- ❑ Register-indirekte Adressierung (register indirect addressing)
- ❑ Indizierte Adressierung (indexed addressing)
  - Speicher-relative Adressierung (memory relative addressing)
  - Register-relative Adressierung (register relative addressing)
  - Register-relative Adressierung mit Index (Based indexed mode)
- ❑ Befehlszähler-relative Adressierung (PC relative addressing)

# Unmittelbare Adressierung (immediate addressing)

---

- ❑ Der Befehl enthält nicht die Adresse des Operanden oder einen Zeiger darauf, sondern den Operanden selbst.
- ❑ OpCode und Operand belegen im Speicher hintereinander folgende Speicherworte
- ❑ **Assemblerschreibweise:**       $\langle \text{Mnemo} \rangle \ \# \langle \text{Operand} \rangle$
- ❑ **Effektive Adresse:**               $EA = (PC) + 1$

# Unmittelbare Adressierung (immediate addressing)



## Beispiel:

**LDA   #\$A3           (load accumulator)**

*(Lade den Akkumulator A mit dem Hexadezimalwert \$A3)*

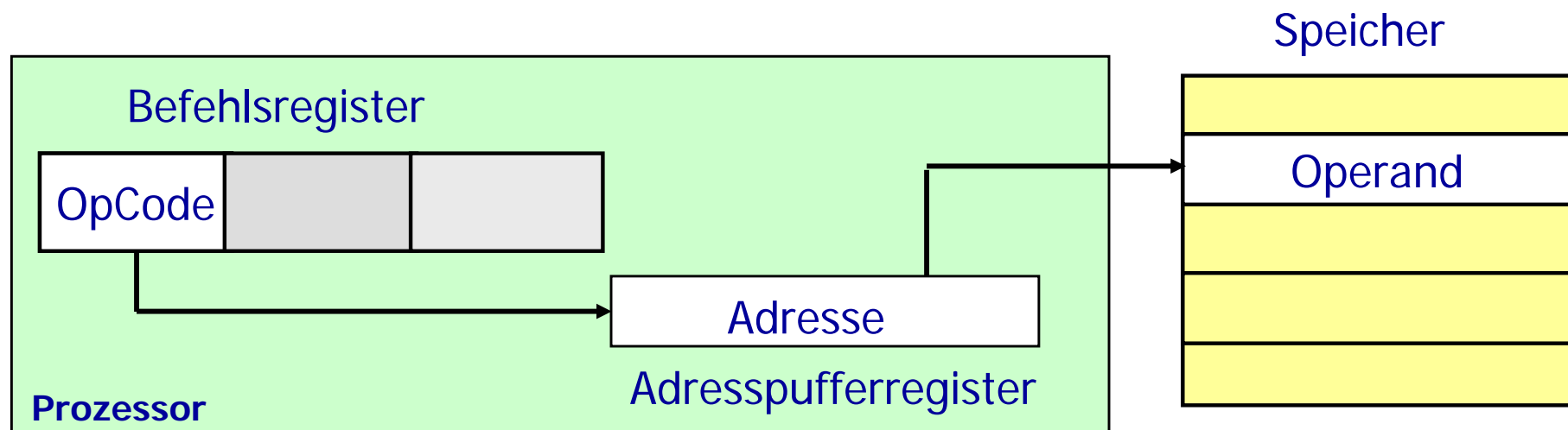
# Direkte Adressierung (*direct addressing*)

---

- Der Befehl enthält im Speicherwort nach dem OpCode die logische Adresse des Operanden, aber keine weiteren Vorschriften zu deren Manipulation, z.B. durch die Addition mit einem Registerinhalt
- **Assemblerschreibweise:**      <Mnemo> <Adresse>
- **Effektive Adresse:**       $EA = ((PC) + 1)$
- Zwei Fälle können unterschieden werden:
  - Absolute Adressierung
  - Seiten-Adressierung

# Absolute Adressierung (*extended direct addressing*)

- Der Befehl enthält im Speicherwort, das dem OpCode folgt, die absolute, d. h. vollständige Adresse des Operanden im (logischen) Adressraum



## Beispiel:

**JMP \$07FE (jump)**

*(Springe zur Adresse \$07FE)*

# Seiten-Adressierung (*direct page addressing*)

---

- Im Befehl steht als Kurz-Adresse nur der niederwertige Teil der Operandenadresse (*Low-Adresse*)

- **Zero-page-Adressierung:**

der höherwertige Adressteil wird durch die entsprechende Anzahl von '0'-Bits ersetzt. Dadurch wird im Adressraum nur die "unterste Seite" (*zero page*) angesprochen

- **Seiten-Register-Adressierung** (*„direct“ page register addressing*):

der höherwertige Adressteil wird in einem Register des Prozessors (DP-Register, direct page register) zur Verfügung gestellt

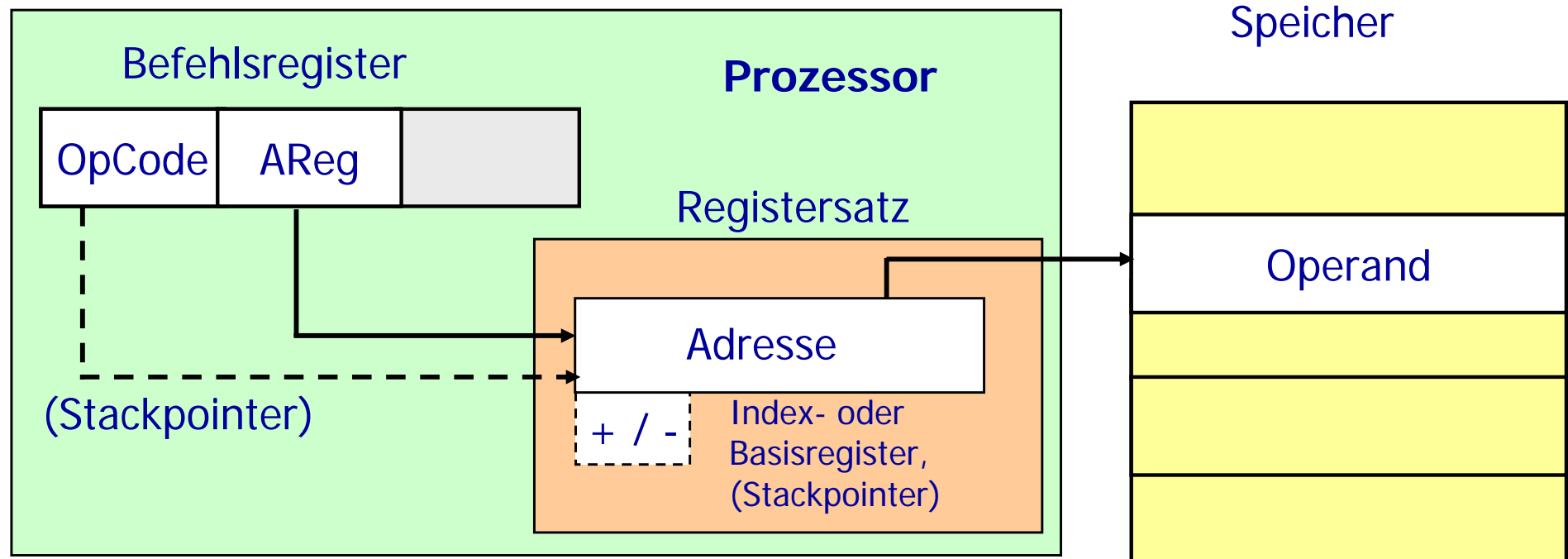
# Register-indirekte Adressierung

---

## Register-indirekte Adressierung (*register indirect addressing*)

- Hier enthält das durch seine Nummer im Register-Feld des OpCodes angegebene Adressregister die Adresse des Operanden (*pointer*, "Zeiger", deshalb auch: Zeigeradressierung)
- **Assemblerschreibweise:**            <Mnemo> (Ri)
- **Effektive Adresse:**                 $EA = (Ri)$

# Register-indirekte Adressierung



## Beispiel:

LD R1, (A0) (*load*)

*(Lade das Register R1 mit dem Inhalt des durch das Adressregister A0 gegebenen Speicherwortes)*

# Register-indirekte Adressierung

Bei der im Register stehenden Adresse handelt es sich oft um die Anfangs- oder Endadresse eines Tabellenbereichs im Speicher → Registerinhalt automatisch modifizieren

- **postincrement:** Nach der Ausführung des Befehls wird der Inhalt des Registers (um 1) erhöht und zeigt danach auf die nachfolgende Speicherzelle

- **Assemblerschreibweise:**  $\langle \text{Mnemo} \rangle (\text{Ri}) +$

- **Effektive Adresse:**  $\text{EA} = (\text{Ri})$

- **Beispiel:**  $\text{INC} (\text{R0}) +$  (increment)

*(Inkrementiere zunächst den Inhalt des Speicherwortes, das durch das Register R0 adressiert wird, und danach den Inhalt von R0)*

# Register-indirekte Adressierung

---

- **predecrement:** Vor der Ausführung des Befehls wird der Inhalt des Registers erniedrigt und zeigt danach auf die vorhergehende Speicherzelle

- **Assemblerschreibweise:**  $\langle \text{Mnemo} \rangle -(\text{Ri})$

- **Effektive Adresse:**  $\text{EA} = (\text{Ri}) - 1$

- **Beispiel:**  $\text{CLR} -(\text{R0})$  (clear)

*(Dekrementiere zuerst den Inhalt des Registers R0 und lösche danach das Speicherwort, das durch R0 adressiert wird)*

# Indizierte Adressierung (*indexed addressing*)

---

- ❑ Die effektive Adresse wird durch die Addition des Inhalts eines Registers zu einem angegebenen Basiswert berechnet. (Adressdistanz zu einem Basiswert, Tabellenverarbeitung)
  
- ❑ Je nachdem, in welcher Form der Basiswert vorgegeben wird, kann man unterscheiden zwischen:
  - Speicher-relative Adressierung (memory relative addressing)
  - Register-relative Adressierung (register relative addressing)
  - Register-relative Adressierung mit Index (Based indexed mode)

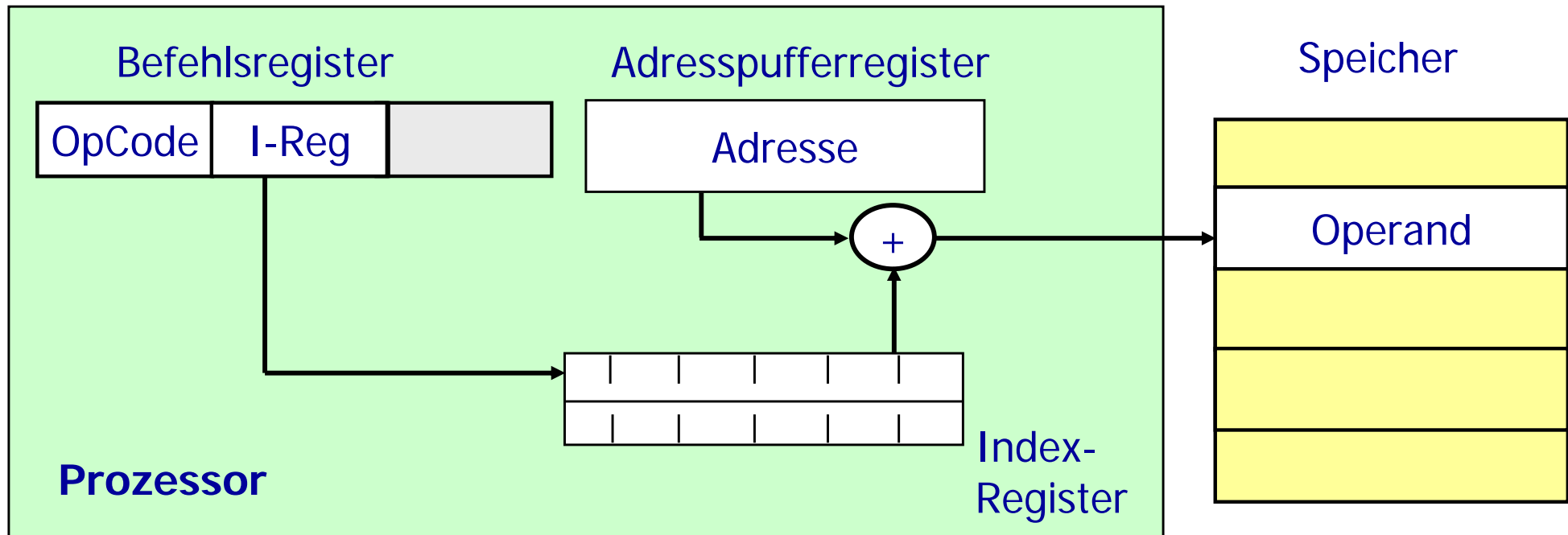
# Indizierte Adressierung (*indexed addressing*)

---

## Speicher-relative Adressierung (*memory relative addressing*)

- Der Basiswert wird als absolute Adresse im Befehl vorgegeben
- **Assemblerschreibweise:**      <Mnemo> <Adresse> (Ii)
- **Effektive Adresse:**               $EA = ((PC) + 1) + (Ii)$

# Speicher-relative Adressierung



## Beispiel:

ST R1,\$A704(R0) (store)

*(Speichere den Inhalt von R1 in das Speicherwort, dessen Adresse sich durch Addition des Inhaltes von R0 zur Basis \$A704 ergibt)*

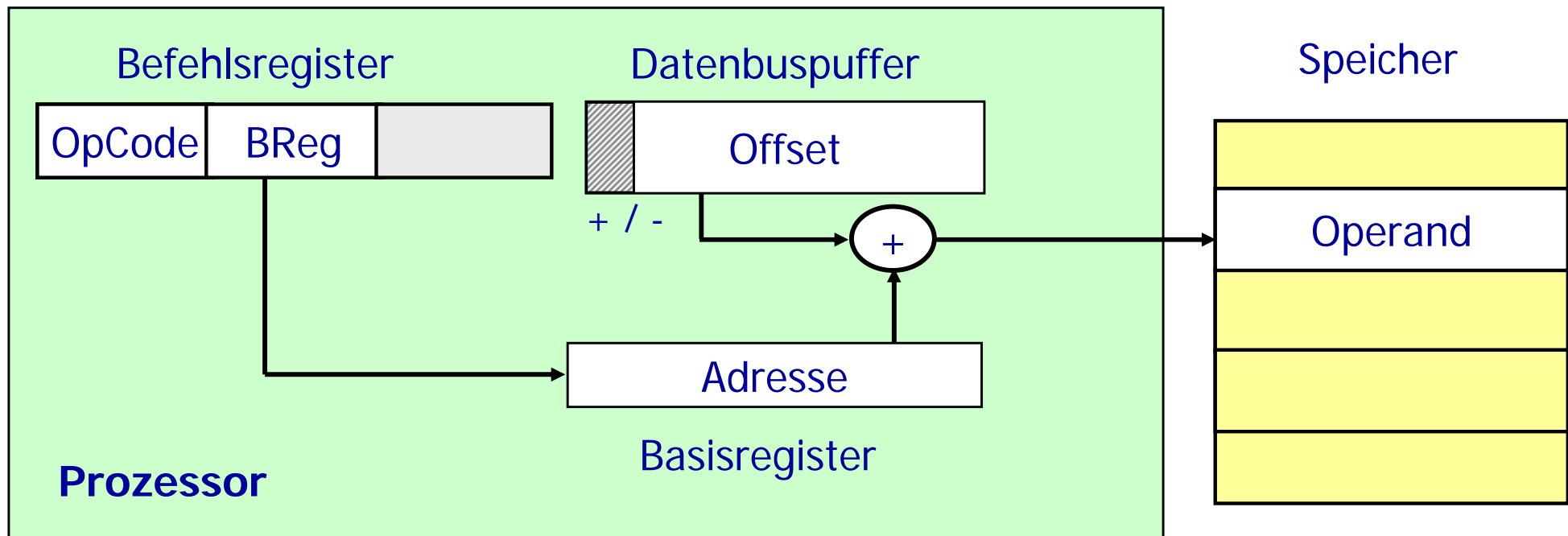
# Indizierte Adressierung (*indexed addressing*)

---

## Register-relative Adressierung (*register relative addressing, based mode*)

- Der Basiswert befindet sich in einem Basisregister, auf das durch das BReg-Feld im OpCode verwiesen wird. Im Befehl wird ein Offset angegeben, der zum Inhalt des Basisregisters addiert wird.
- **Assemblerschreibweise:**       $\langle \text{Mnemo} \rangle \langle \text{Offset} \rangle (\text{Bi})$
- **Effektive Adresse:**               $\text{EA} = (\text{Bi}) + ((\text{PC}) + 1)$

# Register-relative Adressierung



## Beispiel:

CLR \$A7(B0) (clear)

*(Lösche das Speicherwort, dessen Adresse sich durch die Addition des hexadezimalen Offsets \$A7 zum Inhalt des Basisregisters B0 ergibt)*

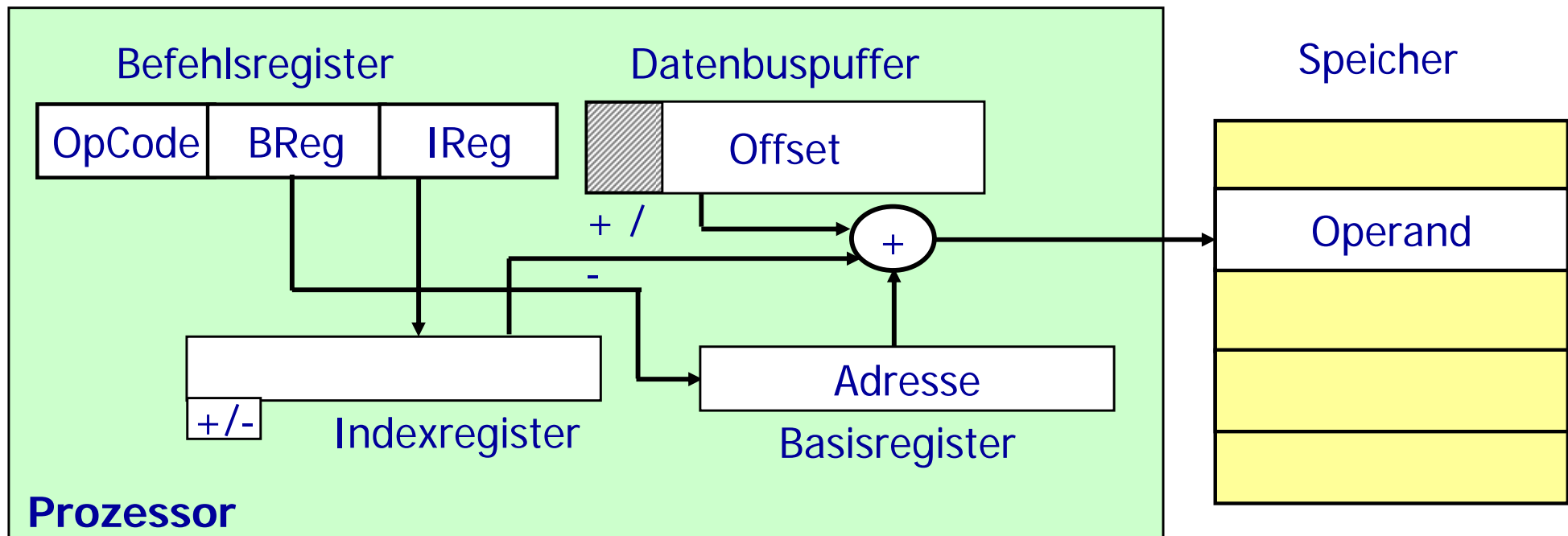
# Indizierte Adressierung (*indexed addressing*)

---

## Register-relative Adressierung mit Index (*based index mode*)

- der Basiswert wird in einem Basisregister übergeben. Dazu wird der Inhalt eines Indexregisters addiert. Für dieses Indexregister kann wieder die automatische Veränderung „autoincrement/autodecrement“ gewählt werden. Zusätzlich kann häufig im Befehl ein Offset angegeben werden, der hinzuaddiert wird.
- **Assemblerschreibweise:** <Mnemo> <Offset> (Bi)(Ii)
- **Effektive Adresse:**  $EA = (Bi) + (Ii) + ((PC) + 1)$

# Register-relative Adressierung mit Index



## Beispiel:

DEC \$A7(B0)(I0)+ (decrement)

*(Dekrementiere das Speicherwort, dessen Adresse sich durch Addition der Inhalte der Register I0 und B0 zum Offset \$A7 ergibt, und erhöhe danach den Inhalt des Registers I0)*

# Indizierte Adressierung (*indexed addressing*)

---

## Befehlszähler-relative Adressierung (*program counter relative addressing*)

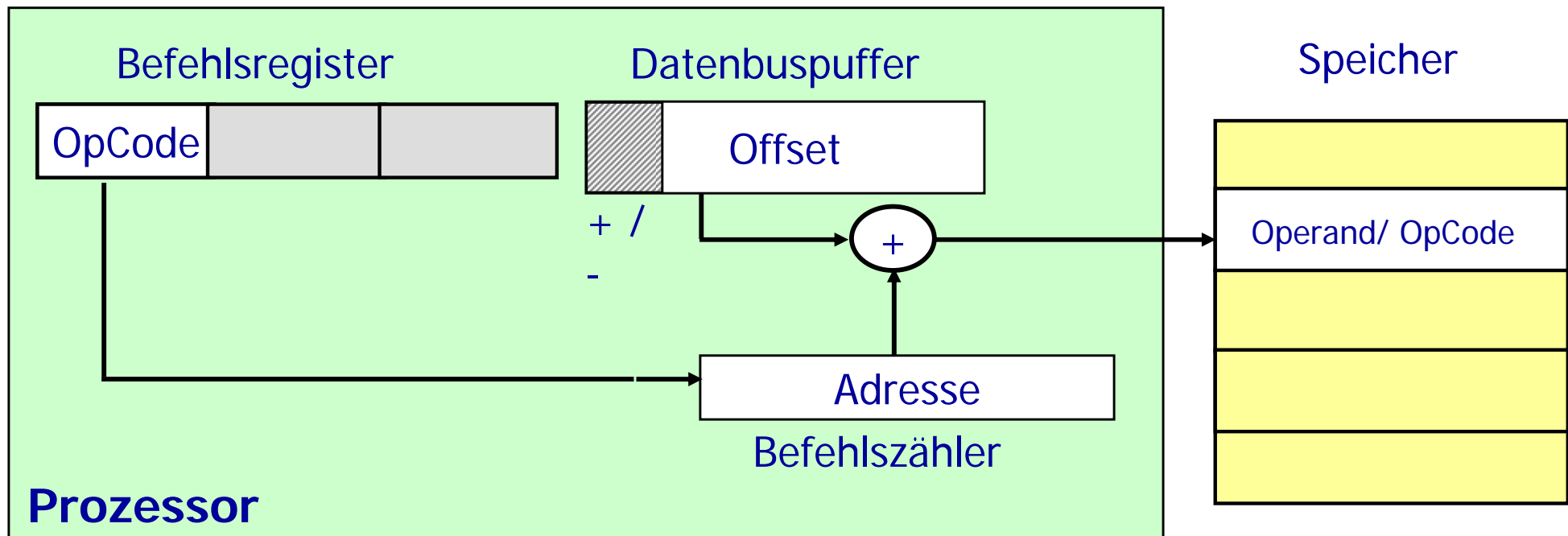
- Die effektive Adresse entsteht durch die Addition eines im Befehl angegebenen Offsets zum aktuellen Befehlszählerstand.
- Diese Adressierungsart erlaubt es, Programme im Hauptspeicher "frei" zu verschieben

- **Assemblerschreibweise:**

$\langle \text{Mnemo} \rangle \quad \langle \text{Offset} \rangle (\text{PC}) \quad \text{oder nur}$   
 $\langle \text{Mnemo} \rangle \quad \langle \text{Offset} \rangle$

- **Effektive Adresse:**  $EA = (\text{PC}) + 2 + ((\text{PC}) + 1)$

# Befehlszähler-relative Adressierung



## Beispiel:

LBRA \$7FFF (*long branch always*)

(Verzweige "unbedingt" zu der Speicherzelle, deren Adressdistanz zum aktuellen Programmzähler \$7FFF ist)

---

# Visualisierung

- Register-Adressierung und
- Einstufige Speicher-Adressierung

Siehe TI-Homepage

<http://ti.ira.uka.de/Adressierungsarten/>



## 4.4 RISC & CISC

---

**CISC**

**Complex Instruction Set Computers**

**RISC**

**Reduced Instruction Set Computers**



# Programmiermodell: Intel 80x86

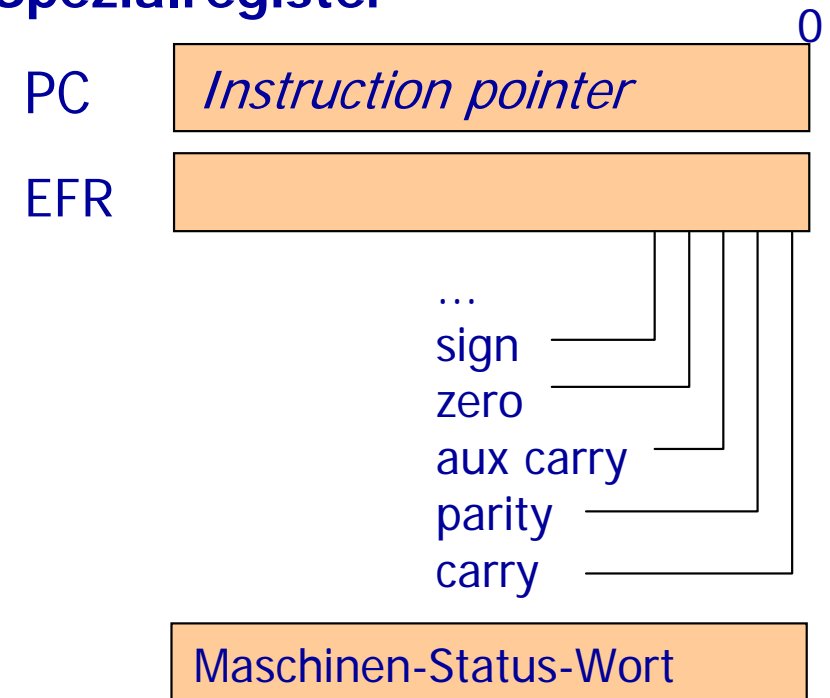
## Allgemeine Register

EAX	Arithm. Ergeb.	AX
EDX	& Ein/Ausgabe	DX
ECX	Zählregister	CX
EBX	Basis-Register	BX
EBP	Basis-Register	BP
ESI	Index-Register	SI
EDI	Index-Register	DI
ESP	Stack pointer	SP

## Segmentregister

CS	Code-Segment
SS	Stack-Segment
DS	Daten-Segment
ES	Daten-Segment
FS	Daten-Segment
GS	Daten-Segment

## Spezialregister



In der Übung:  
MIPS-Programmiermodell

# Befehlsaufbau der Intel-x-86-Prozessoren



# Befehlsaufbau der Intel-x-86-Prozessoren

---

- ❑ Ein Befehl besteht aus maximal 12 Bytes.
- ❑ Der Opcode belegt je nach Befehlstyp 1 oder 2 Bytes
  - Das WL-Bit (*Word Length*) bestimmt die Länge eines im Befehl „unmittelbar“ angegebenen Datum oder eines Offsets
- ❑ Die folgenden beiden Bytes dienen zur Auswahl der Adressierungsart und der dabei benutzten Register
- ❑ Falls ein Offset für die Adressberechnung erforderlich ist, wird dieser in den folgenden 1 bis 4 Bytes angegeben
- ❑ In den nächsten Bytes kann ein „unmittelbar adressiertes“ Datum mit einer Länge von 1, 2, 4 byte auftreten.

# CISC & RISC

---

## Zwei Techniken zur Implementierung von Befehlen im Rechner

### ➤ **Direkt durch Hardware**

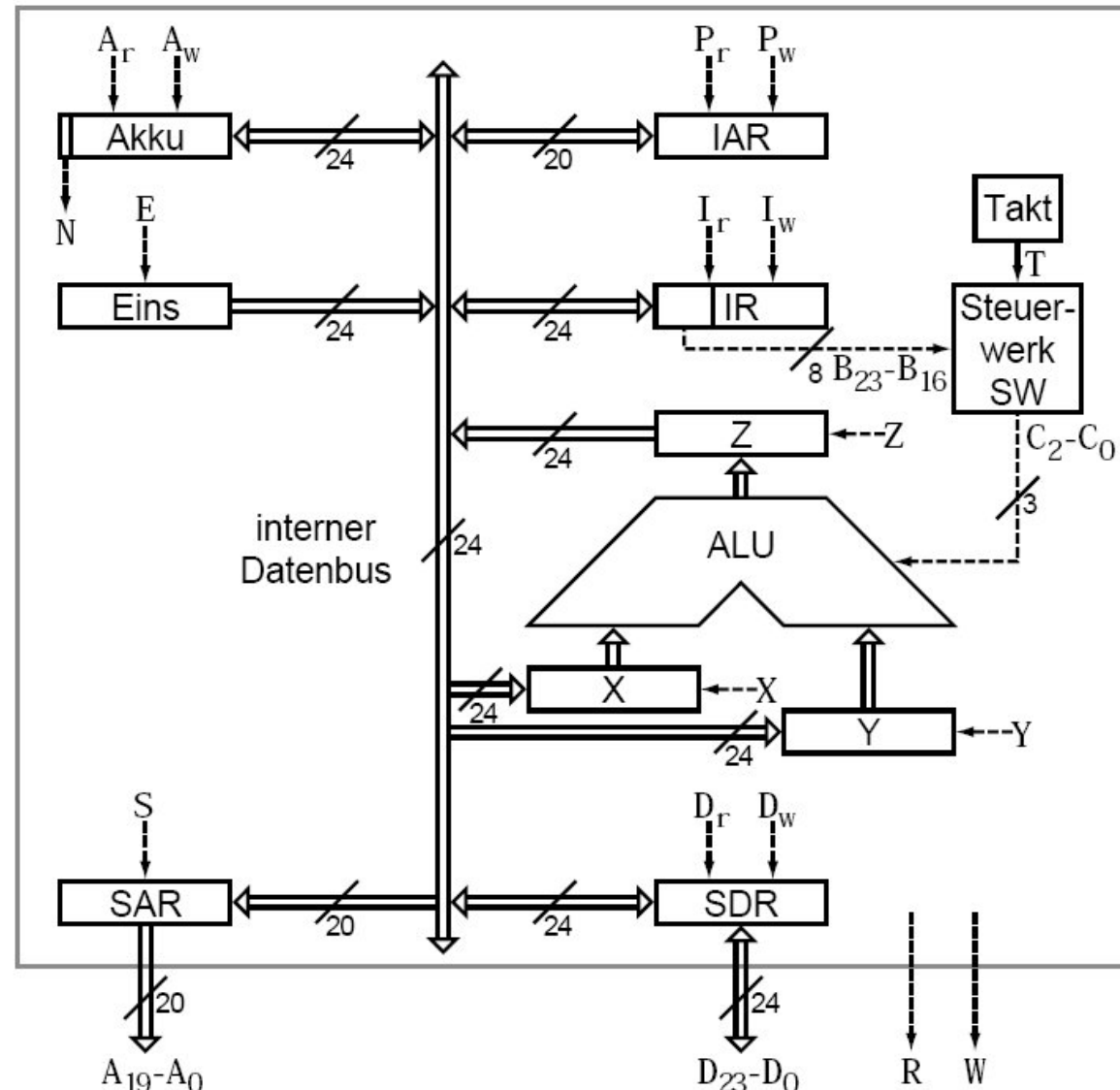
- aufwendige Schaltnetze und Schaltwerke bei großer Anzahl von Befehlen (200-300)

### ➤ **Mikroprogramme als Befehlsinterpreter**

- Mikroprogrammspeicher im Steuerwerk, der neu geladen werden kann
- verschiedene Befehlssätze können implementiert werden

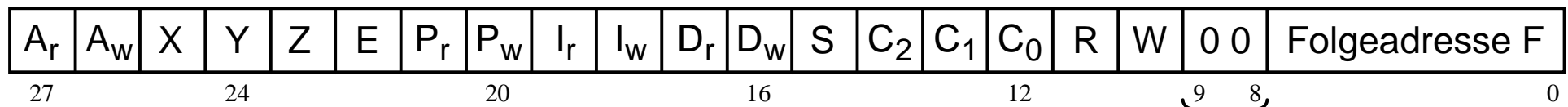
# Mima-Architektur (Übungsblatt 2)

- Mikroprogrammierte Minimalmaschine (von-Neumann-Prinzip)
- SW mit 10 Meldesignale, 18 Steuersignale und Mikroprogrammspeicher für maximal 256 Mikrobefehle
- Befehlsabarbeitung:
  - Lese-Phase
  - Dekodierphase
  - Ausführungsphase
- 3 Taktzyklen für Lese- und Schreibzugriffe



# Mikroprogrammsteuerwerke

## Mikrobefehlsformat:



Jedes Bit des Mikrobefehls entspricht einem Steuersignal

- **Horizontale Mikroprogrammierung:**  
Steuerungsfeld im Mikrobefehl für jedes Steuersignal im Rechner → Kein Dekoder
- **Vertikale Mikroprogrammierung:**  
Zusammenfassung von Steuersignalen zu Feldern.

# Mikroprogrammsteuerwerke

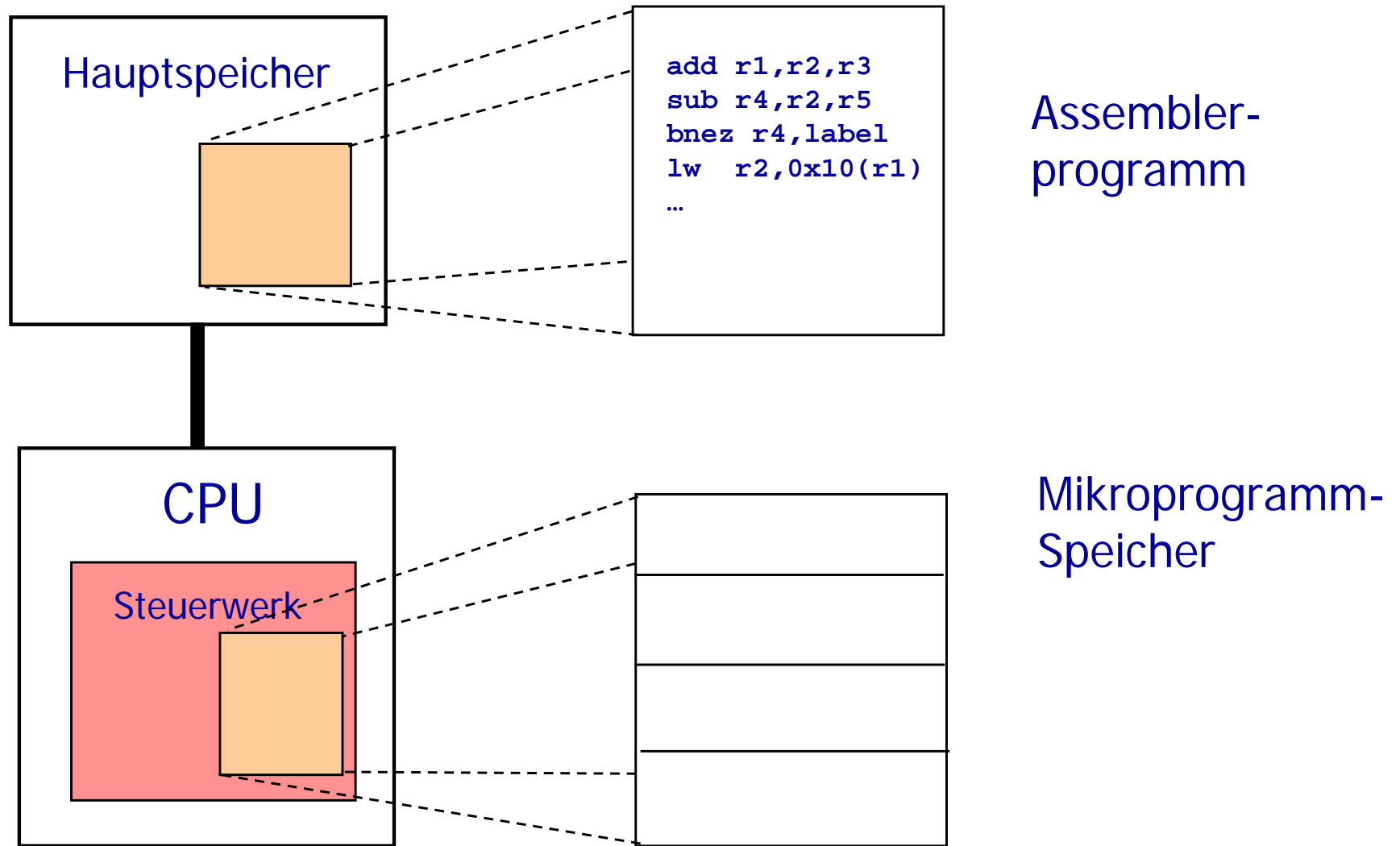
## Fetch-Phase bei der MIMA:

1. Takt: IAR -> SAR; IAR -> X; R = 1
2. Takt: Eins -> Y; ALU auf addieren; R = 1
3. Takt: ALU auf addieren; R = 1
4. Takt: Z -> IAR
5. Takt: SDR -> IR

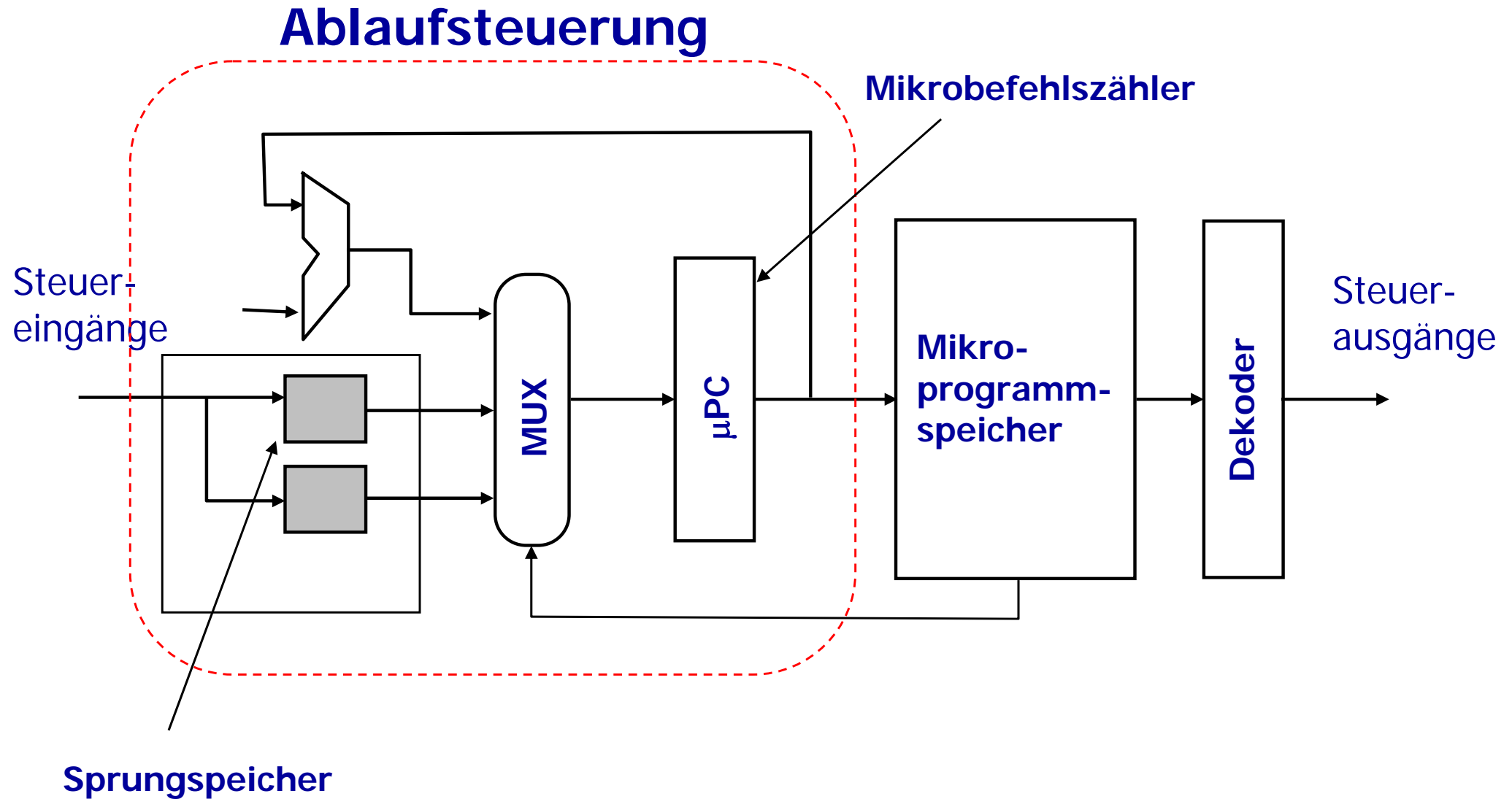
## Mikroprogramm der Fetch Phase

0010	0001	0000	1000	1000	0000	0001
0001	0100	0000	0000	1000	0000	0010
0000	0000	0000	0001	1000	0000	0011
0000	1010	0000	0000	0000	0000	0100
0000	0000	1001	0000	0000	0000	0101

# Prinzip der Mikroprogrammierung



# Implementierung des Steuerwerks



# Vorteile und Nachteile

---

## ➤ Vorteile der Mikroprogrammierung:

- Mehrere Befehlssätze auf einem Rechner ➔ Anpassung des Befehlssatz an der Anwendung
- Mehrere Rechnertypen mit dem gleichen Befehlssatz

## ➤ Nachteile der Mikroprogrammierung:

- Aufwendig
- Langsam

# CISC (complex instruction set computers)

---

## Gründe dafür:

- Ausführung komplizierter Befehle ist immer noch schneller als die Ausführung von Programmen gleicher Funktion
- Mikroprogrammierung begünstigt komplizierte Befehle
- komplizierte Befehle führen zu kurzen Programmen
- Umfang des Befehlssatzes wird oft als Werbeargument verwendet
- Unterstützung höherer Programmiersprachen durch komplizierte Befehle (Direkte Abbildung: *Sprachkonstrukt* → *Befehl*)

# CISC (complex instruction set computers)

---

## Gründe dafür:

- Unterstützung von Compilern durch entsprechende Befehle
- Unterstützung spezieller Einsatzgebiete

## Fazit:

**Entwicklung von Hardware,  
Programmiersprachen und Einsatzgebieten  
begünstigt „komplizierte“ Befehle.**

# CISC (complex instruction set computers)

---

## Gründe dagegen:

- Schnellere Hauptspeicher und die Verwendung von Cache-Speichern beschleunigen die Programmausführung
- Mikroprogramme wurden immer umfangreicher; Verlängerte Entwurfszeit, Komplexe Steuerwerke (> 50% der Chipfläche)
- Nur relativ kleine Teile des großen Befehlssatzes werden häufig benutzt
- Größere Fehlerhäufigkeit auf der Mikroprogrammebene
- Schwieriger Compilerbau

# CISC (complex instruction set computers)

---

- **Systemprogramme in XPL auf IBM/360:**

90 % aller ausgeführten Befehle: 10 verschiedene Befehle

95 % aller ausgeführten Befehle: 21 verschiedene Befehle

99 % aller ausgeführten Befehle: 30 verschiedene Befehle

- **COBOL-Programme auf IBM/370:**

90,28 % aller ausgeführten Befehle: 26 verschiedene Befehle

99,08 % aller ausgeführten Befehle: 48 verschiedene Befehle

*(nur 84 verschiedene Befehle wurden überhaupt benutzt)*

# Limitationen der CISC Architekturen

---

- **Befehlsausnutzung (80/20 Regel):**

viele mächtige Befehle, komplexes Befehlsformat, Mikroprogrammierung, nur 20 % der Befehle werden überwiegend benutzt

- **Kritisches Problem: Anzahl der Zyklen pro Instruktion (CPI)**

bei allen heutigen CISC Architekturen ist  $CPI \gg 2$

# Prozentualer Anteil von Anweisungen in Hochsprachenprogrammen

Großteil der in Hochsprachen verwendeten Anweisungen ist sehr einfach:

Anweisung	Mittlerer zeitlicher Anteil
Zuweisung	47 %
if	23%
call	15 %
loop	6 %
goto	3 %
Andere	7 %

# RISC (reduced instruction set computers)

---

## Grundprinzipien:

- Viel benutzte einfache Befehle so schnell wie möglich machen (Ausführung möglichst in einer Taktphase. Keine Mikroprogrammierung mehr, Befehls-Pipeline)
- Der größte Teil der Arbeit soll durch optimierende Compiler zur Übersetzungszeit erledigt werden
- Operanden werden nach Möglichkeit in großen Registersätzen gehalten → schneller Zugriff → schnelle Verarbeitung
- Einheitliche Befehlsformate → schnelle Decodierung
- Pipelining anwenden, so gut es geht

# RISC (reduced instruction set computers)

---

## Entwurfsziele:

- Ausführung jedes Befehls in einem Taktzyklus  
(*Befehl  $\approx$  bisheriger Mikrobefehl bei CISC*)
- Alle Befehle gleich lang:  
Decodierschaltung wird einfacher  
Programme länger, aber Ausführungszeit kürzer
- Nur Load-Store und Register-Register-Befehle:  
weniger Adressierungsarten → schnelle Ausführung
- Koprozessorarchitektur für komplexe Befehle

# RISC (reduced instruction set computers)

---

## Keine Entwurfsziele sind z. B.:

- Unterstützung von Gleitkomma-Arithmetik
- Unterstützung von Betriebssystemfunktionen

# Zielvorstellungen für RISC-Rechner

---

- Ein-Zyklus-Befehle
- Einheitliches Befehlsformat
- Wenige Maschinenbefehle
- Load/Store-Architektur
- Großer Registersatz
- Verzicht auf Mikroprogrammierung
- 32-Bit-Architektur
- Pipeline-gerechter Maschinenbefehlssatz (gleiche Befehlsausführzeiten)
- Keine Unterstützung für Betriebssystem und Gleitkomma-Arithmetik

# Forderungen an RISC-Systeme

---

- Mindestens 75% aller Befehle sind Ein-Zyklus-Befehle
- Einheitliche Länge aller Befehle entsprechend der Datenbusbreite
- Nicht mehr als 128 Befehle
- Nicht mehr als 4 Befehlsformate
- Nicht mehr als 4 Adressierungsarten
- Load/Store-Architektur
- Festverdrahtete Steuereinheit, keine Mikroprogrammierung
- Mindestens 32 allgemein verwendbare Register

# RISC-Rechner aus heutiger Sicht

---

**Geblieden ist von der RISC-Idee im wesentlichen:**

- das Befehlspipelining
- die Load/Store-Architektur
- ein großer Registersatz: 32 allgemeine und 32 Gleitpunkt-Register
- ein einheitliches Befehlsformat
- die Verwendung weniger Adressierungsarten
- der Verzicht auf Mikroprogrammierung

# RISC & CISC

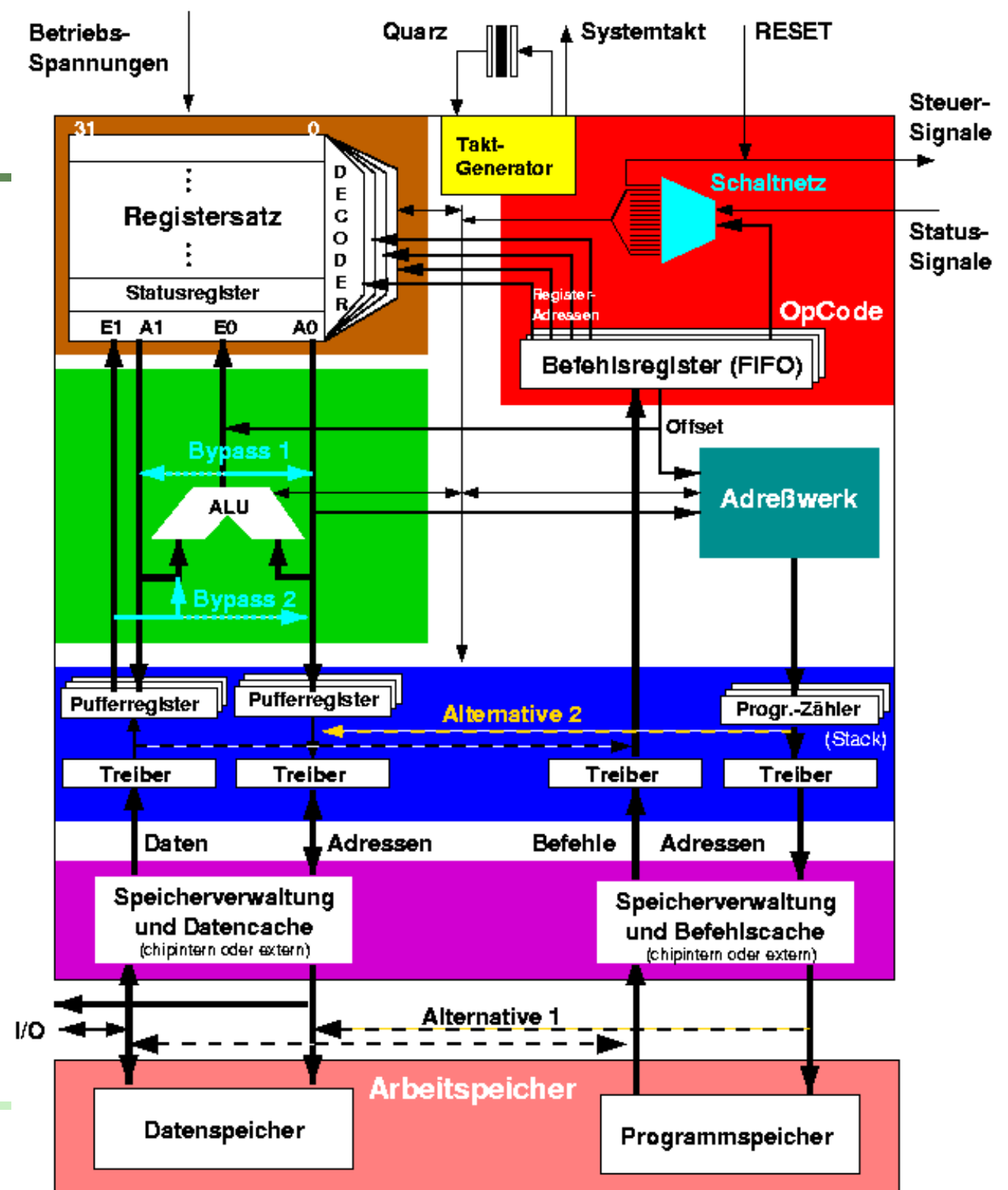
CISC	RISC
Komplexe Befehle, Ausführung in mehreren Taktzyklen	Einfache Befehle, Ausführung in einem Taktzyklen
Jeder Befehl kann auf den Speicher zugreifen	Nur Lade- und Speicherbefehle greifen auf den Speicher zu
Wenig Pipelining	Intensives Pipelining
Befehle werden von einem Mikroprogramm interpretiert	Befehle werden durch festverdrahtete Hardware ausgeführt
Befehlsformat variabler Länge	Alle Befehle mit fester Länge
Die Komplexität liegt im Mikroprogramm	Die Komplexität liegt im Compiler
Einfacher Registersatz	Mehrere Registersätze

# Vergleich CISC & RISC

Vergleich von drei typischen CISC-Rechnern mit den ersten drei RISC-Rechnern [Tanenbaum]:

	CISC			RISC		
	IBM 370/168	VAX 11/780	Xerox Dorado	IBM 801	Berkeley RISC I	Stanford MIPS
Fertigstellungsjahr	1973	1978	1978	1980	1981	1983
Instruktionen	208	303	270	120	<b>31</b>	55
Mikrocodegröße	54k	61k	17k	0	0	0
Instruktionsgröße	2-6 Bytes	2-57 Bytes	1-3 Bytes	4 Bytes	4 Bytes	4 Bytes
Operationsmodell	Reg-Reg Reg-Mem Mem-Mem	Reg-Reg Reg-Mem Mem-Mem	Stack	Reg-Reg	Reg-Reg	Reg-Reg

# Aufbau eines RISC-Prozessors



# Aufbau eines RISC-Prozessors

---

## Havard Architektur:

getrennter Programm- und Datenspeicher, deshalb  
zwei Adress- und Datenbusse

→ paralleles Holen von Operanden und Instruktionen

## Vereinfachende Varianten:

1. zwei getrennte Bussysteme bis zu den Cache-Speichern, jedoch nur ein Arbeitsspeicher (niedrigere Kosten)
2. nur ein Bussystem wie bei Standard-Mikroprozessoren

# Aufbau eines RISC-Prozessors

---

## **Systembusschnittstelle:**

enthält Registerblocks sowohl für Daten als auch für Adressen (gleichzeitiges Lesen eines Datums und Zwischenspeichern eines Ergebnisses)

## **Befehlszähler:**

ist manchmal als Hardware-Stack ausgebildet (beschleunigt Unterprogrammaufrufe)

# Aufbau eines RISC-Prozessors

---

## Steuerwerk:

- festverdrahtet
- Das Befehlsregister als Warteschlange (FIFO) realisiert
- Für jede Pipeline-Stufe ist dort ein Register vorhanden
- Die OpCodes jeder Stufe können vom Schaltnetz des Steuerwerks ausgewertet werden

## Registersatz:

- besteht aus einer großen Zahl von Registern
- erlaubt gleichzeitige Auswahl von 3 bis 4 Registern  
(z. B. 4 Port Registersatz, gleichzeitiges Schreiben (E0, E1) und Lesen (A0, A1) von jeweils 2 Registern)

# Aufbau eines RISC-Prozessors

---

## Rechenwerk:

- Besitzt eine Load/Store-Architektur.
- Die Operanden werden über 2 Operandenbusse aus dem Registersatz herbeigeführt, das Ergebnis (noch im selben Taktzyklus) über den Ergebnisbus in den Registersatz geschrieben.
- Normalerweise gibt es keine direkte Verbindung zwischen ALU und Systemdatenbus
- Datentransfer läuft über die Register (Load/Store-Architektur)

# Befehlsverarbeitung in RISC-Prozessoren

---

## Einfacher Befehlssatz von RISC-Prozessoren

➔ Maschinenprogramme sind länger als bei CISC Prozessoren

*(komplexe Befehle und Adressierungsarten müssen aus den einfachen RISC-Befehlen zusammengesetzt werden)*

Trotzdem arbeitet ein RISC-Prozessor meist schneller als ein CISC-Prozessor. Der Grund liegt in der nahezu **vollständigen Parallelarbeit aller Komponenten** eines RISC-Prozessors (extreme Pipeline-Verarbeitung)

Es wird mit großer Wahrscheinlichkeit **in jedem Taktzyklus ein Befehl** beendet

# RISC - superskalar

---

- ❑ RISC-Prozessoren, die das Entwurfsziel von durchschnittlich einer Befehlsausführung pro Takt (CPI – *cycles per instruction* oder IPC – *instructions per cycle* von eins) erreichen, werden als **skalare RISC-Prozessoren** bezeichnet.
- ❑ Die Superskalar-Technik ermöglicht es, pro Takt mehrere Befehle den Ausführungseinheiten zuzuordnen und eine gleiche Anzahl von Befehlsausführungen pro Takt zu beenden.
- ❑ Solche Prozessoren werden als **superskalare (RISC)-Prozessoren** bezeichnet, da die oben definierten RISC-Charakteristika auch heute noch weitgehend beibehalten werden.
- ❑ Heutige Mikroprozessoren nutzen Befehlsebenenparallelität durch die Pipelining- und Superskalartechnik.