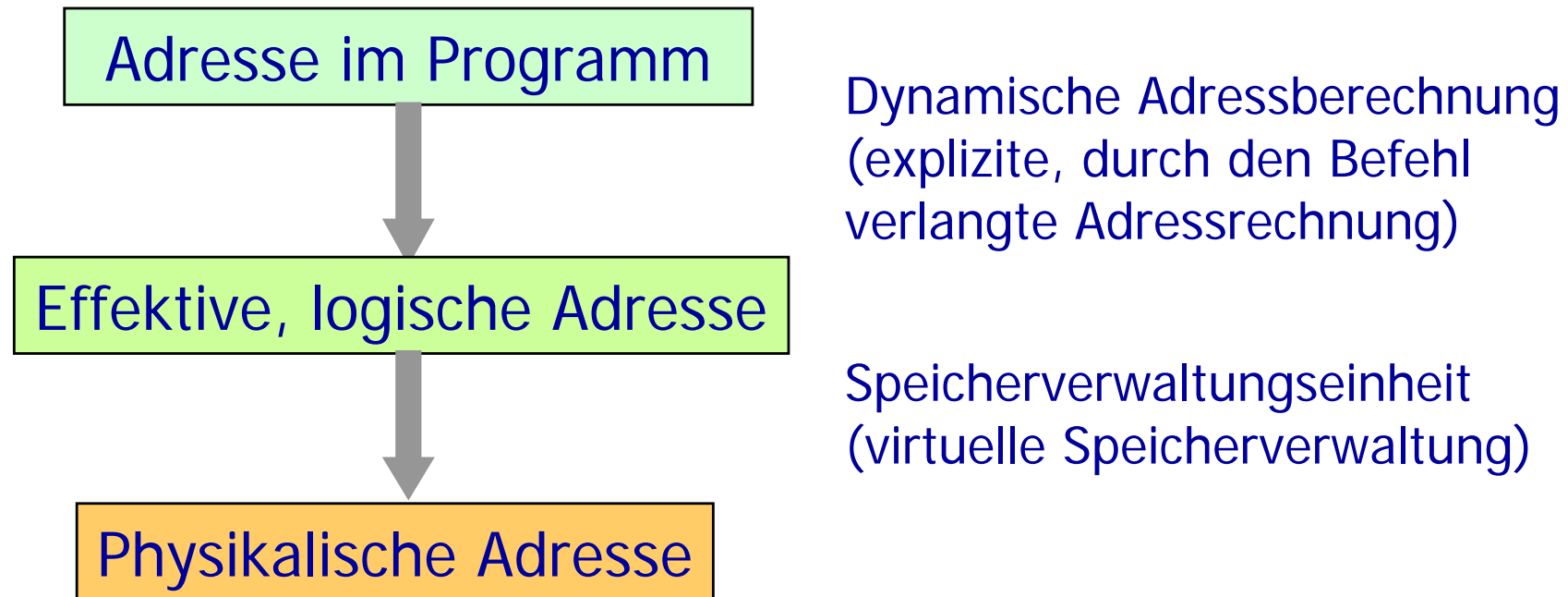


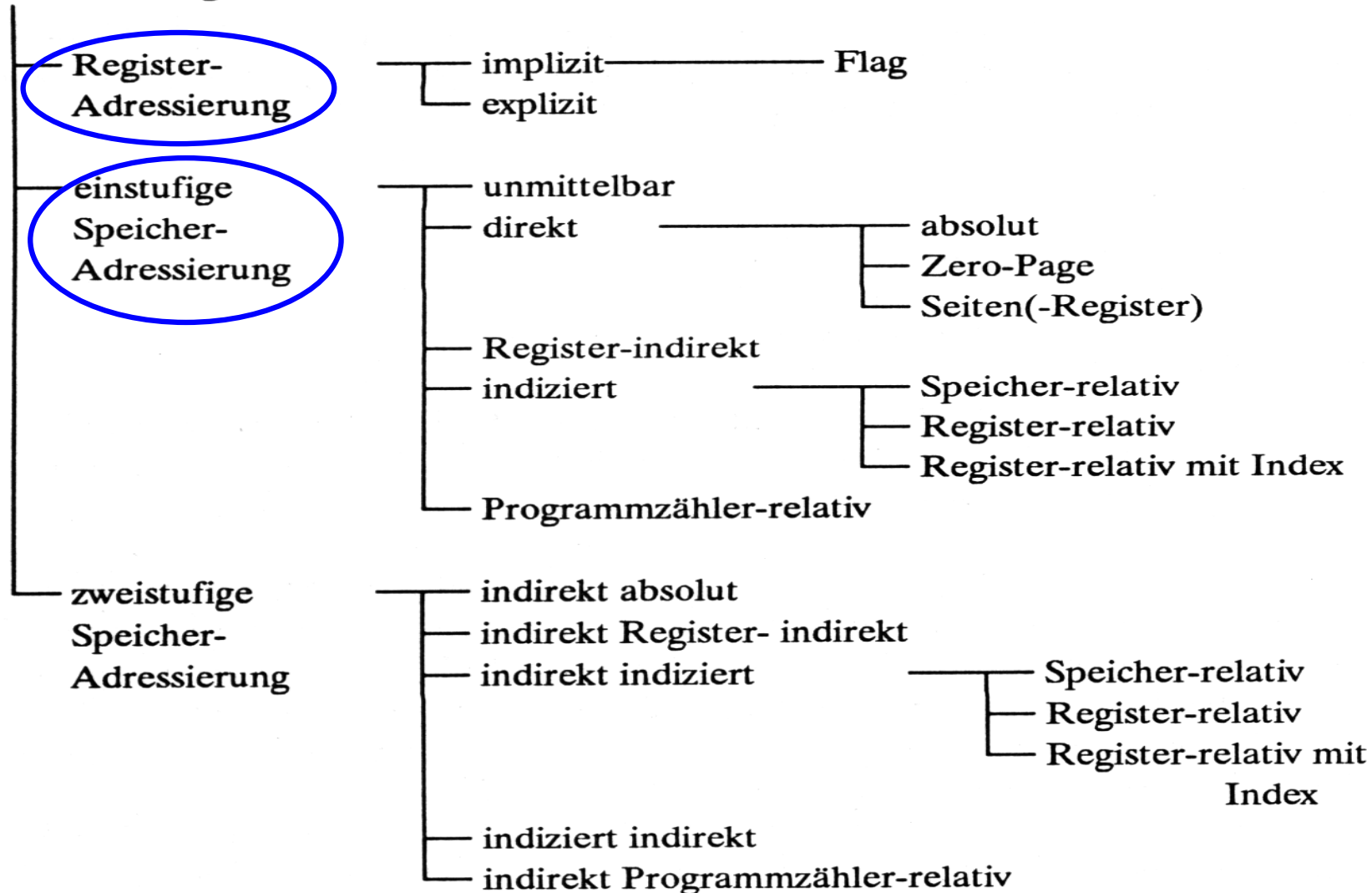
4.3 Adressierungsarten

Ablauf der Adressberechnung:



4.3 Adressierungsarten

Adressierungsarten



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)



Direkte Adressierung (*direct addressing*)

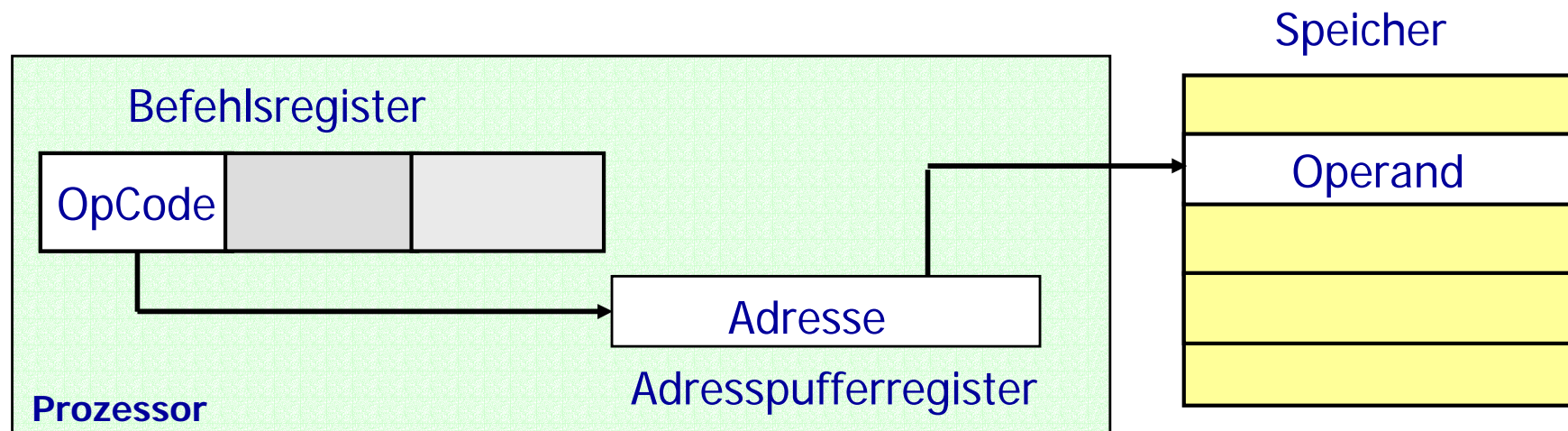
- ❑ Der Befehl enthält im Speicherwort nach dem OpCode die logische Adresse des Operanden, aber keine weiteren Vorschriften zu deren Manipulation, z.B. durch die Addition mit einem Registerinhalt
- ❑ **Assemblerschreibweise:** <Mnemo> <Adresse>
- ❑ **Effektive Adresse:** $EA = ((PC) + 1)$

- ❑ 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



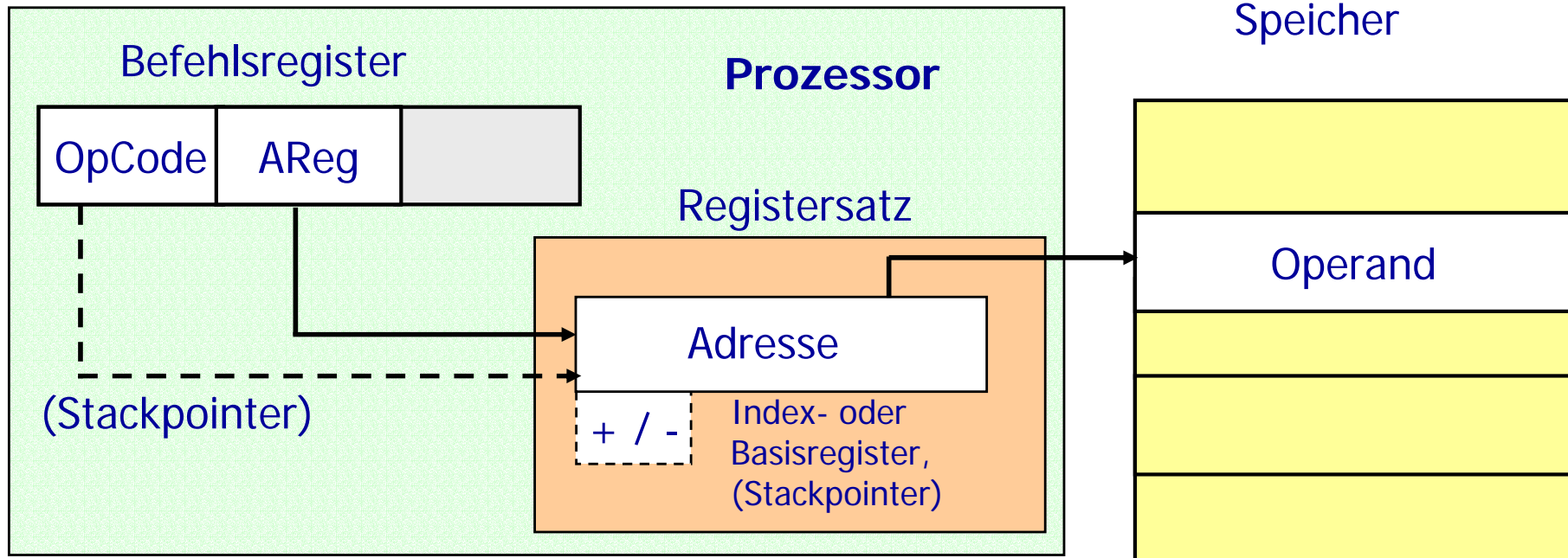
Register-indirekte Adressierung

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:** <Mnemonic> -(Ri)

- **Effektive Adresse:** $EA = (Ri) - 1$

- **Beispiel:** CLR -(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)



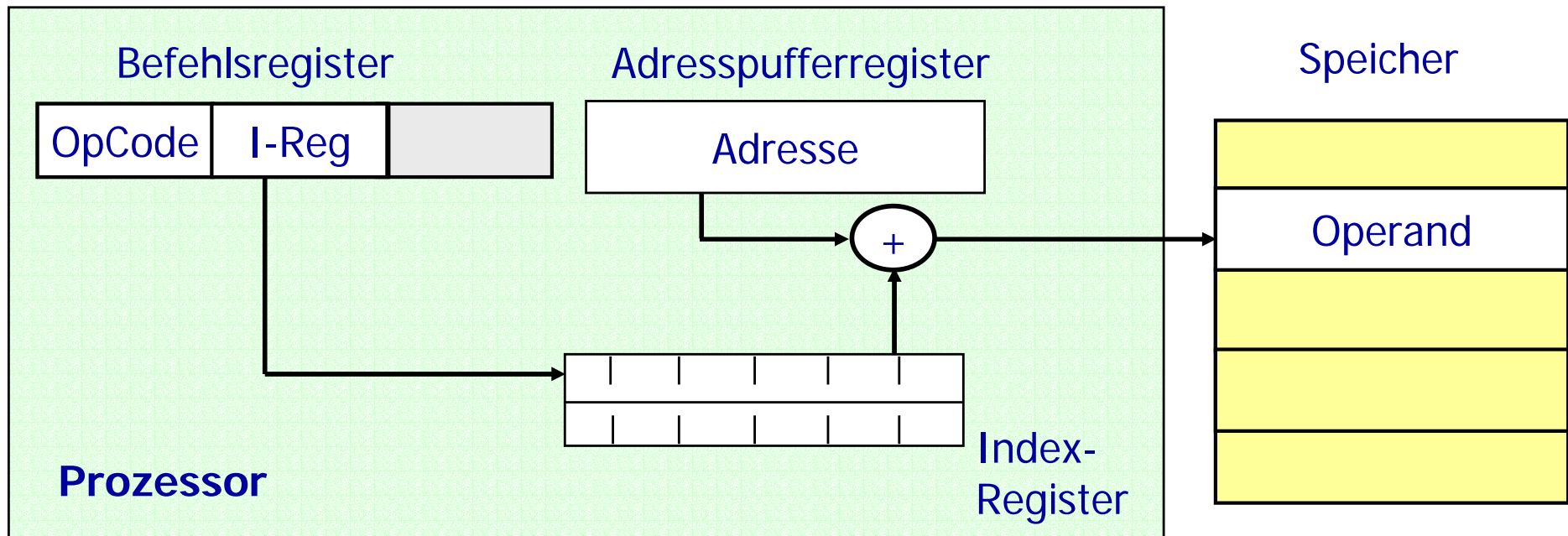
Indizierte Adressierung (*indexed addressing*)

Speicher-relative Adressierung (*memory relative addressing*)

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



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)



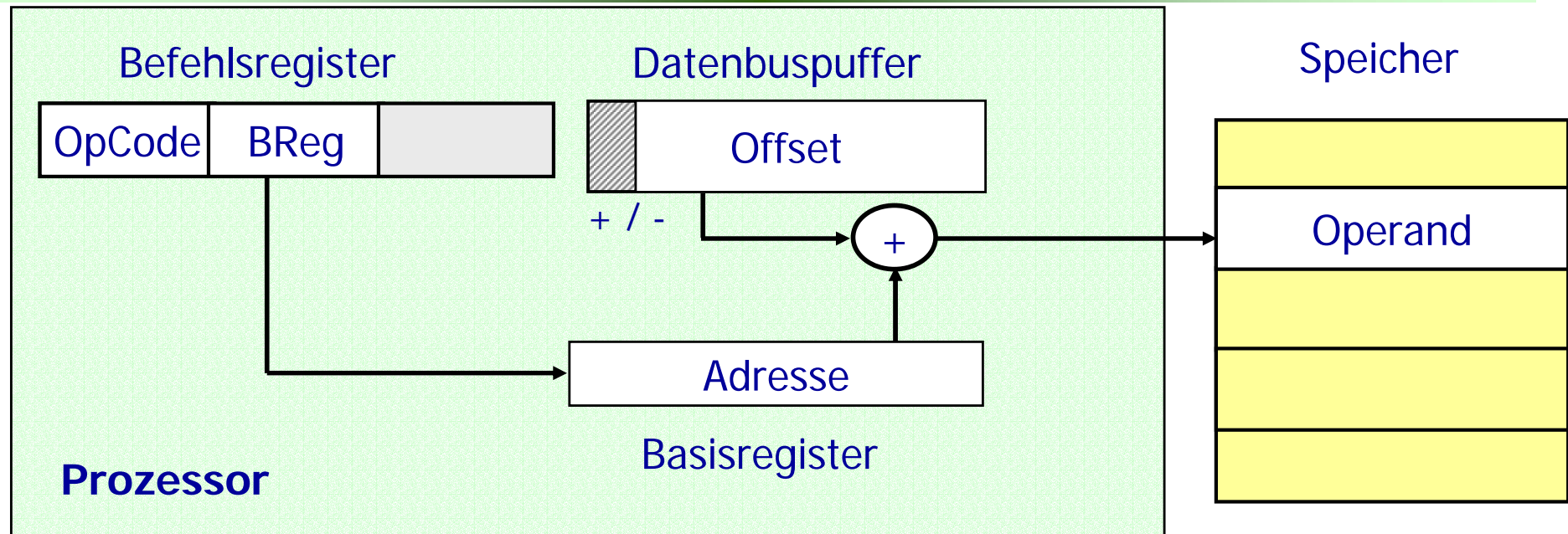
Indizierte Adressierung (*indexed addressing*)

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:** <Mnemo> <Offset> (Bi)
- ❑ **Effektive Adresse:** $EA = (Bi) + ((PC) + 1)$



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)



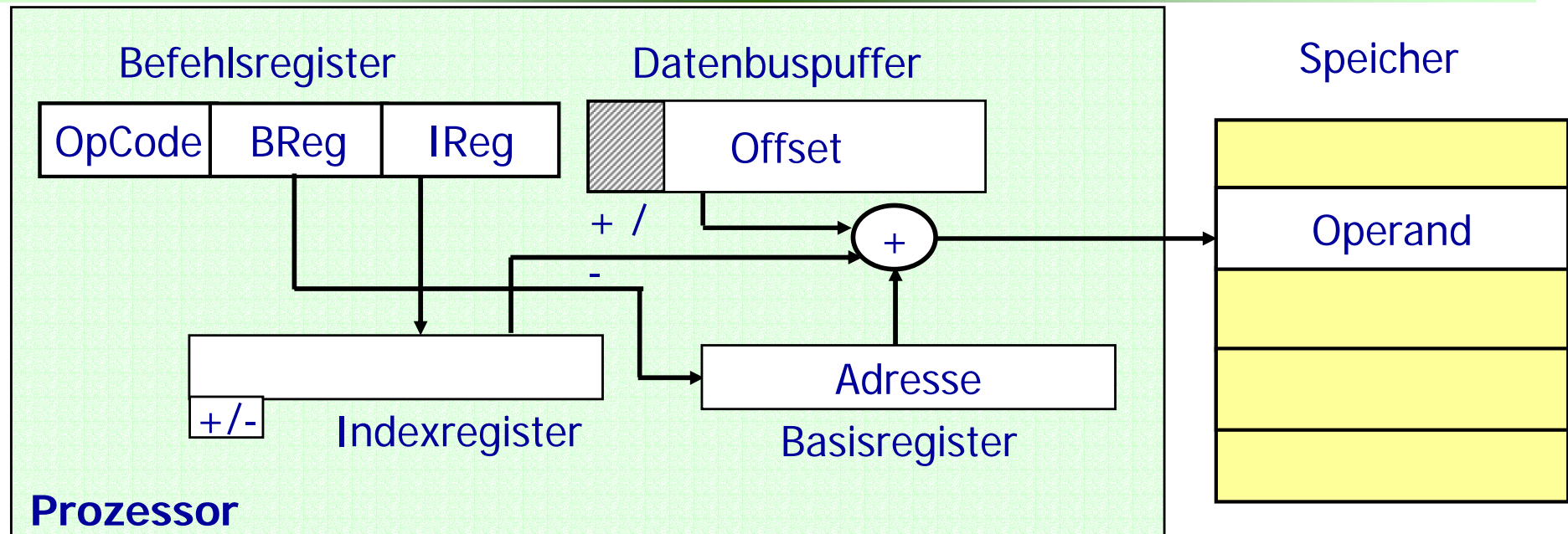
Indizierte Adressierung (*indexed addressing*)

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

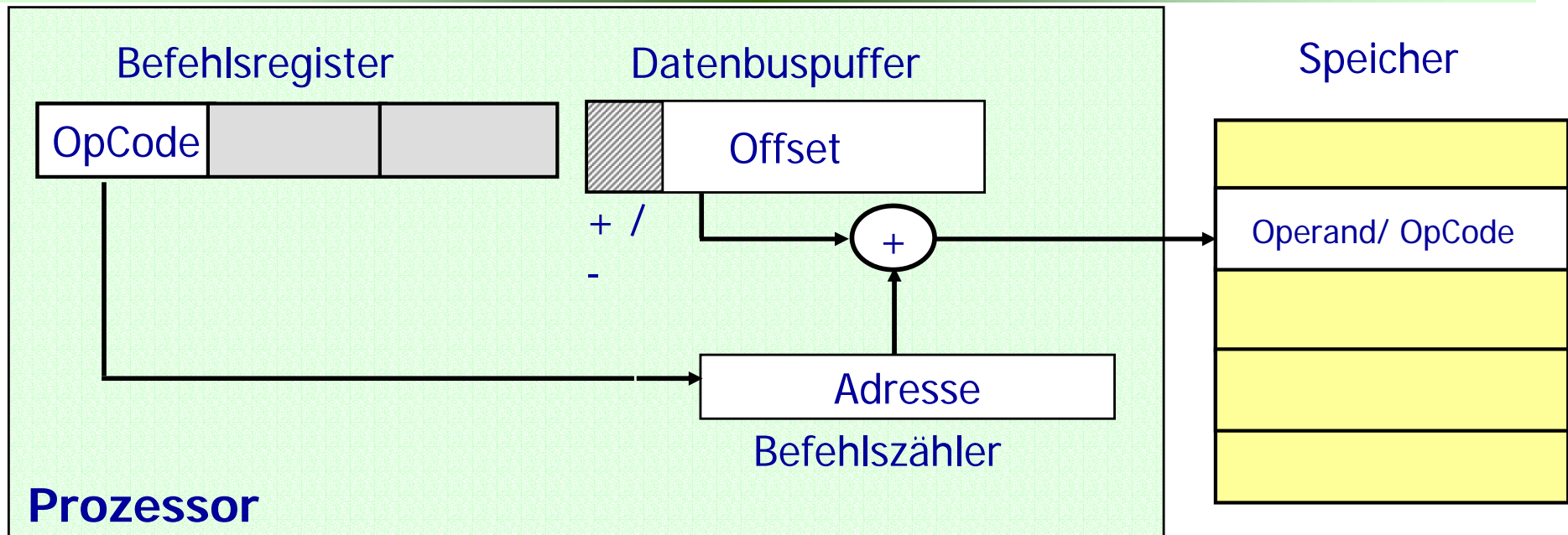
- **Assemblerschreibweise:**

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

- **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://i61www.ira.uka.de/users/asfour/TI/Adressierungsarten/>



4.4 RISC & CISC

CISC

Complex Instruction Set Computers

RISC

Reduced Instruction Set Computers

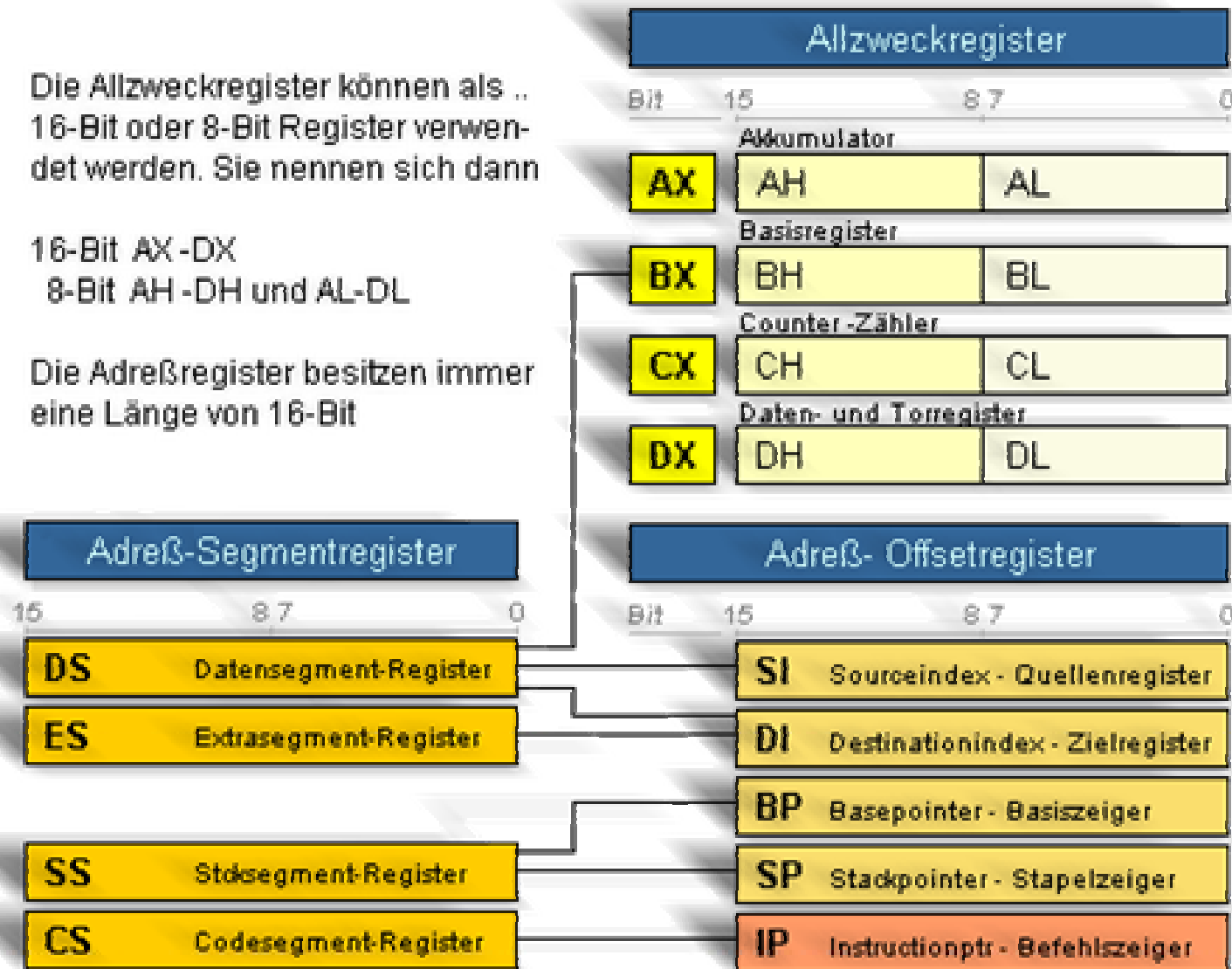


Programmiermodell der Intel 80x86

Die Allzweckregister können als ..
16-Bit oder 8-Bit Register verwendet werden. Sie nennen sich dann

16-Bit AX - DX
8-Bit AH - DH und AL - DL

Die Adreßregister besitzen immer eine Länge von 16-Bit



Programmiermodell der Intel 80x86

Der Registersatz ist eine Aufwärts-Entwicklung der Registersätze von 8080 und 8086:

Register des 8080:

8 Bit Register	16 Bit Register
AL, DL, DH (DX) CL, CH (CX) BL, BH (BX)	SP, IP

Einige 8 Bit Register zusammengefaßt zu 16 Bit Registern (DX, CX, BX)

AL mit dem Statusregister zu einem 16 Bit Register zusammengefaßt (PSW, Processor Status Word)



Programmiermodell der Intel 80x86

Register des 8086:

- Alle Register jetzt 16 Bit Register (AX, DX, CX, BX, SP, IP)
- zusätzliche 16 Bit Register BP, SI, DI
- zusätzliche 16 Bit Register CS, DS, SS

Aus Kompatibilitätsgründen und zur Unterstützung Byte-Orientierter Probleme sind die Register AX, DX, CX und BX weiterhin byteweise ansprechbar

Die zusätzlichen 16 Bit Register CS, DS und SS dienen als Segmentregister zusammen mit SP und IP, sowie CX und DX zur Erweiterung des Adressraums auf 20 Bit (1Mbyte)



Programmiermodell der Intel 80x86

Erweiterung aller Register auf 32 Bit

□ **Allgemeine Register:** Daten- und Adressregister, aber teilweise mit Spezialfunktionen

- Akkumulator (EAX)
- Datenregister für Ein-/Ausgabe (EDX)
- Zählregister für Schleifen (ECX)
- Basisregister (EBX, EBP)
- Indexregister (ESI, EDI)
- Stackregister (ESP)



Programmiermodell der Intel 80x86

□ **Spezialregister:**

- **Segment-Register** zur virtuellen Adressverwaltung (logische → physikalische Adresse) sowie aus Kompatibilitätsgründen zu 8086
- Code-Segment Register (CS)
- Stack-Segment Register (SS)
- Daten-Segment Register (DS - GS)
- **Programmzähler** (EIP)
32 Bit (4 Gbyte Adressraum)



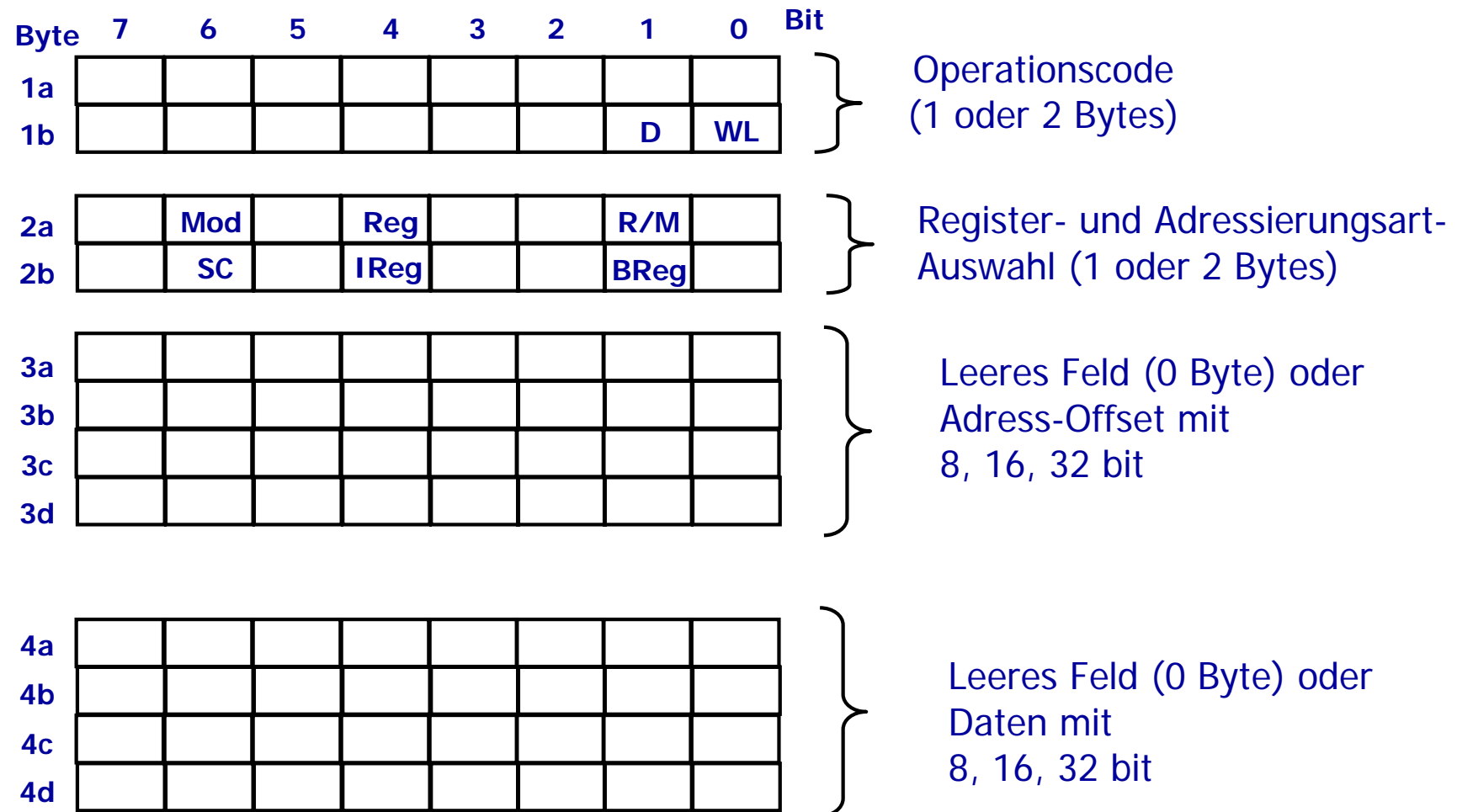
Programmiermodell der Intel 80x86

□ **Spezialregister:**

- **Statusregister** (EFR): Flags, wie CF, PF, AF und DF
- **Steuerregister** (CR0, Control Register 0): enthält Flags, die den Prozessorzustand beschreiben
 - PG: Paging Enable (Speicherverwaltung mit Seitenwechsel)
 - ET: Processor Extension (Art der Coprozessorsteuerung)
 - TS: Task Switch (Prozesswechsel)
 - EM: Emulate Coprozessor (Coprozessor-Emulation)
 - MP: Monitor Coprozessor (Coprozessor-Überwachung)
 - PE: Protection Enable (virtuelle (protected) oder reale (real) Speicherverwaltung)



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)

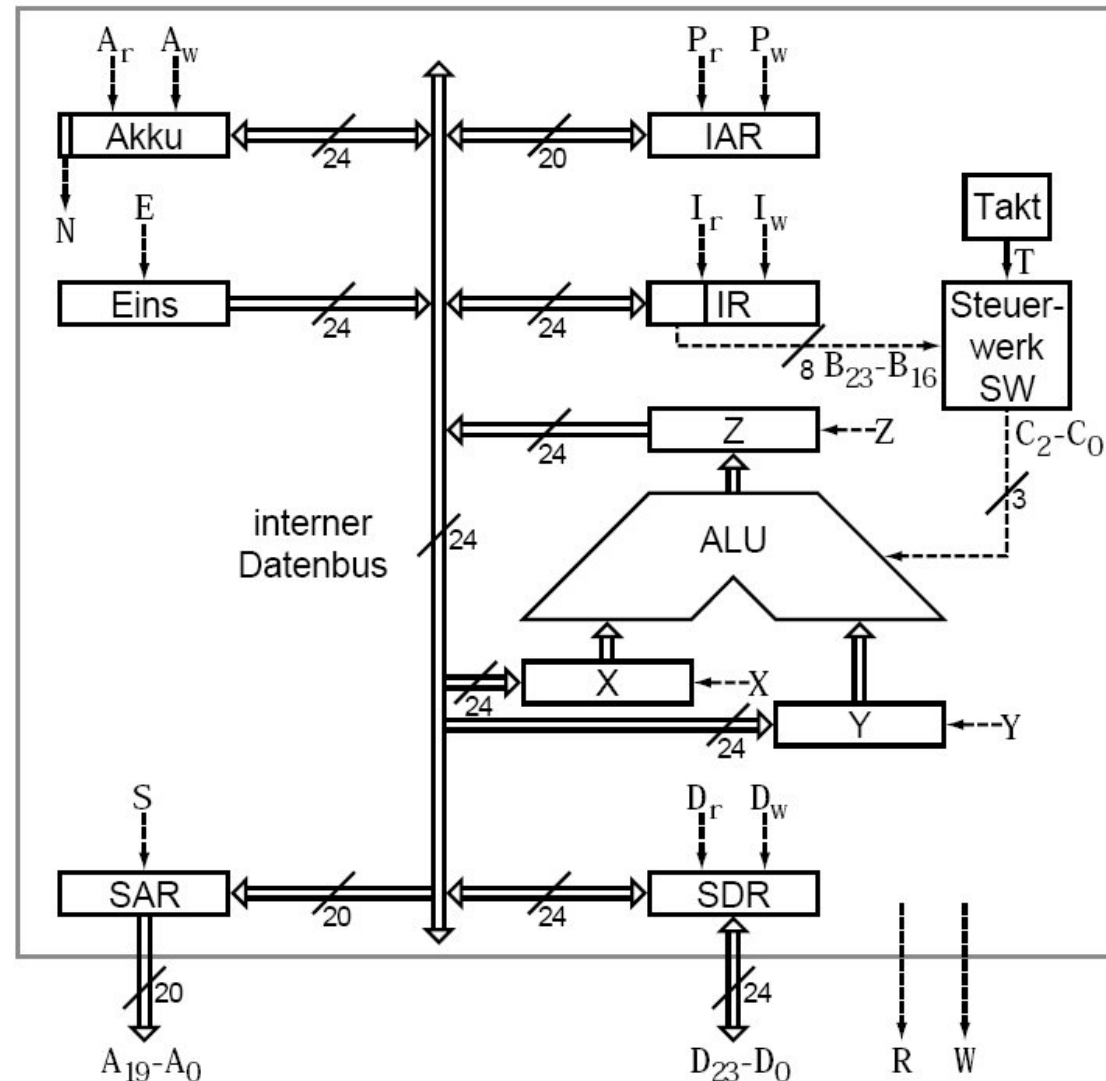
➤ **Mikroprogramme als Befehlsinterpreter**

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



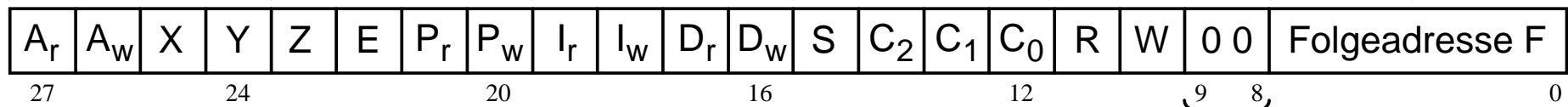
Mima-Architektur (Übungsblatt 3)

- 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:

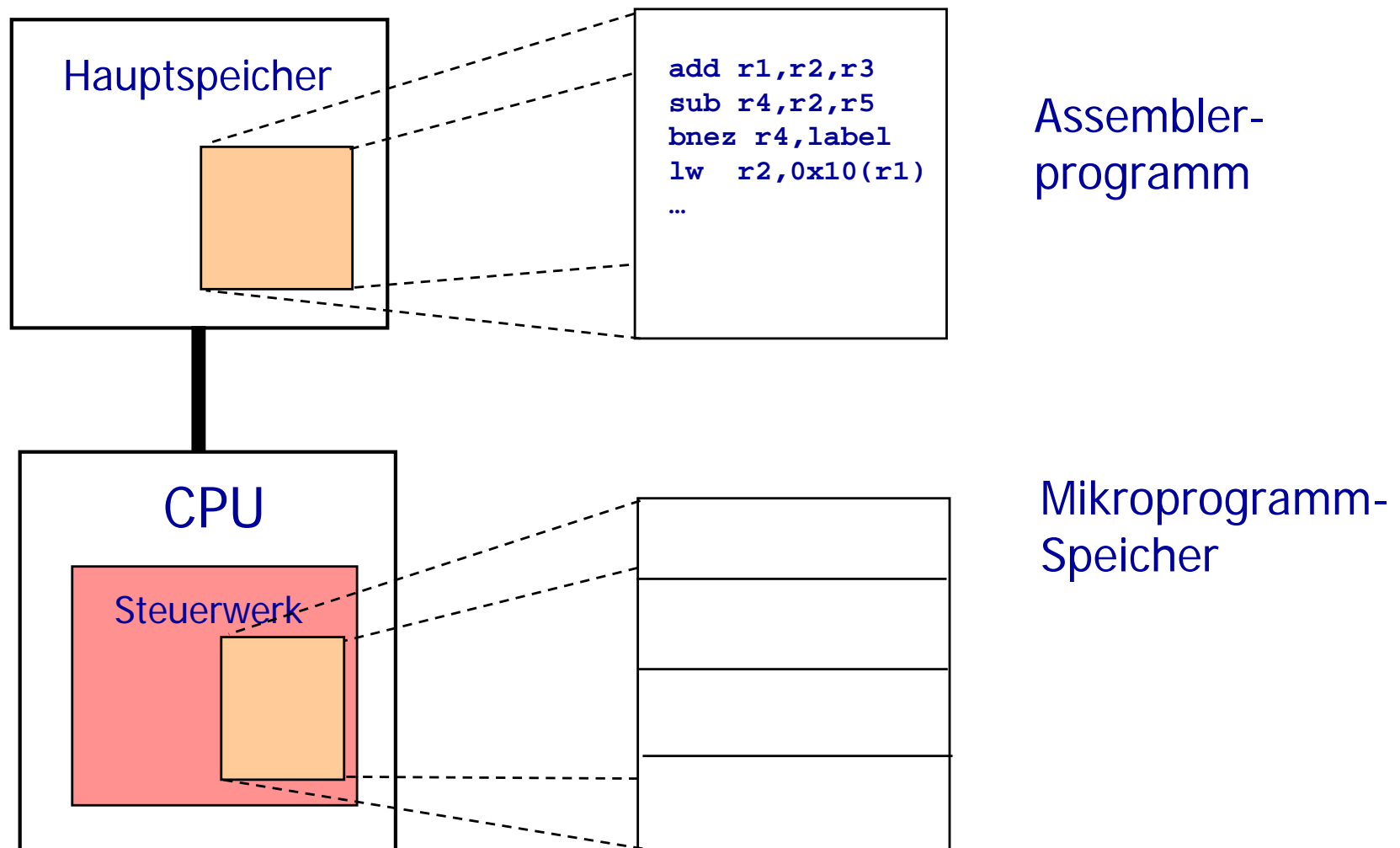
1. Takt: IAR -> SAR; IAR -> X; R = 1
2. Takt: Eins -> Y; ALU auf addieren; R = 1
3. Takt: ALU auf addieren; R = 1
4. Takt: Z -> IAR
5. Takt: SDR -> IR

Mikroprogramm der Fetch Phase

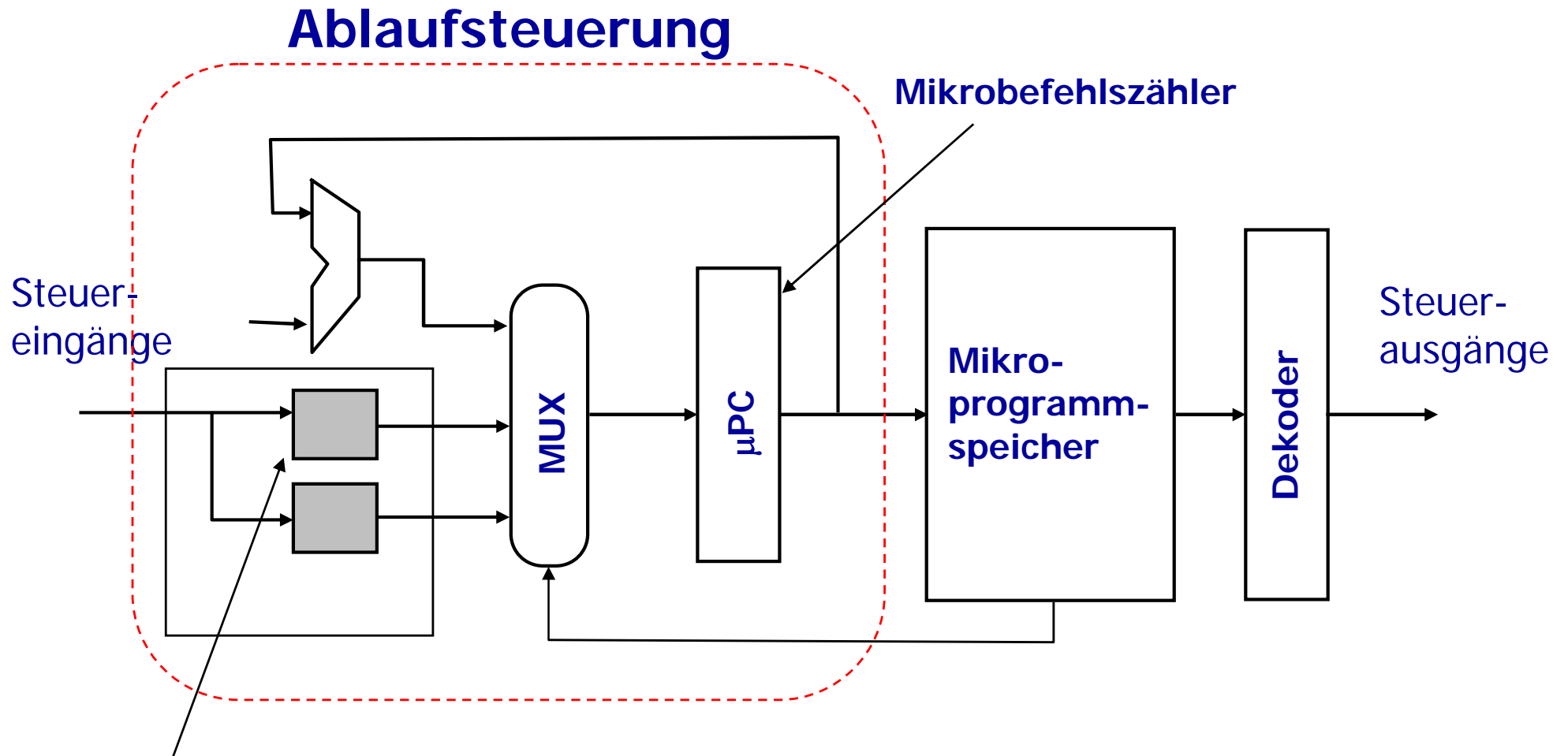
0010	0001	0000	1000	1000	0000	0001
0001	0100	0000	0000	1000	0000	0010
0000	0000	0000	0001	1000	0000	0011
0000	1010	0000	0000	0000	0000	0100
0000	0000	1001	0000	0000	0000	0101



Prinzip der Mikroprogrammierung



Implementierung des Steuerwerks



Sprungspeicher

