

# 1. Betriebssysteme und Systemprogrammierung

- Teil 2** → {
- 1.1. Einführung
  - 1.2. Computer-Hardware: Ein Kurz-Überblick
  - 1.3. Instruktionsarchitektur (Instruction Set Architecture, ISA)
  - 1.4. Virtuelle Maschinen
  - 1.5. Java und die Java Virtual Machine
  - 1.6. Zusammenfassung (Kapitel 1)

# 1.3. Instruktionsarchitektur und Assembler-Sprachen

Die Schnittstelle zwischen der Hardware und der Software eines Computers wird durch die sog. „ISA-Ebene“ (**ISA = Instruction Set Architecture**) gebildet.

Bei neuen Prozessoren muss meist „**Abwärtskompatibilität**“ gewährleistet sein: Die vorhandenen Programme (incl. der Compiler) müssen unverändert lauffähig bleiben.

Eine neue ISA sollte also eine **Obermenge des „alten“ Befehlssatzes** umfassen.

[1.3.1. Instruktionsarchitektur \(Instruction Set Architecture, ISA\)](#)

[1.3.2. Prinzipielle Gestalt von Maschinenbefehlen](#)

[1.3.3. Assembler-Sprachen](#)

[1.3.4. Adressierung](#)

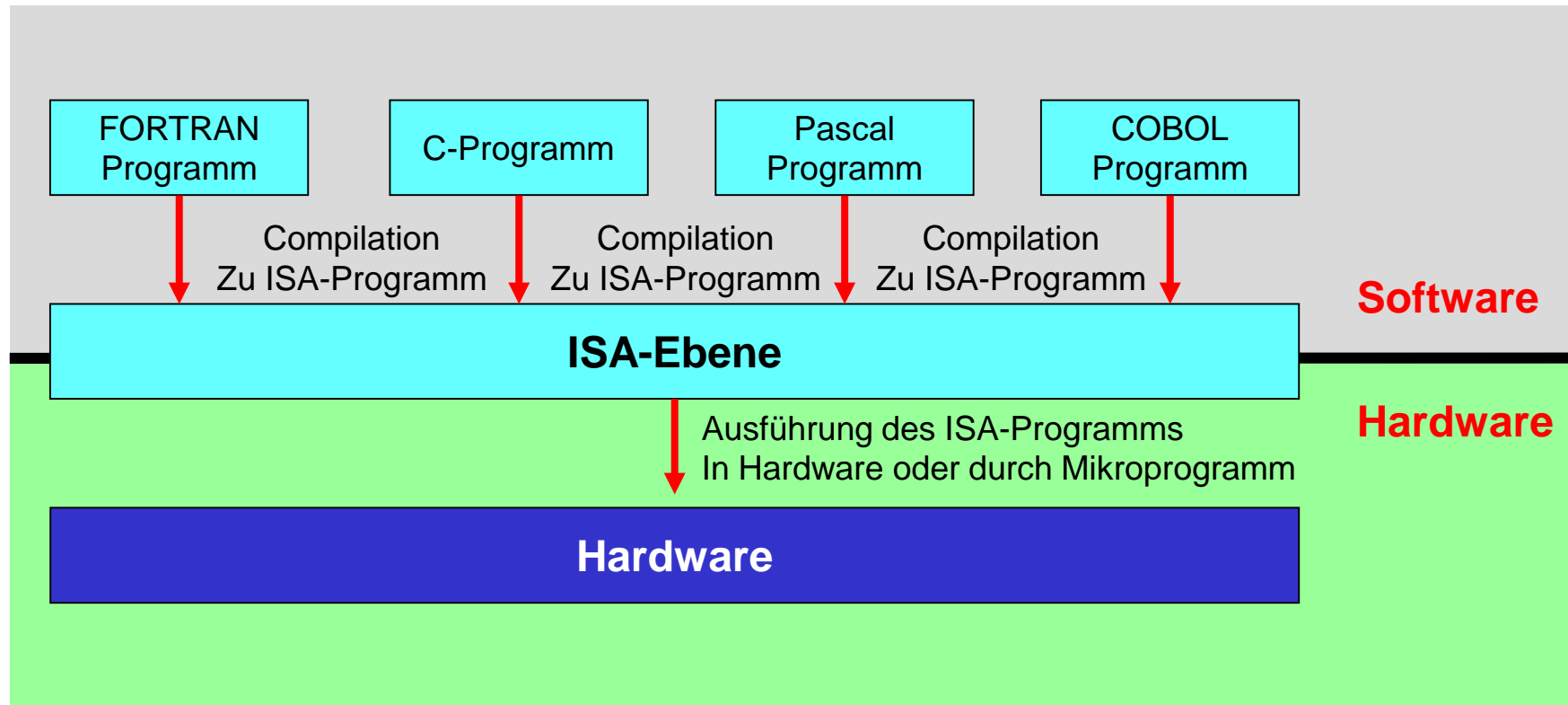
[1.3.5. Unterprogramme](#)

[1.3.6. Vom Assemblerprogramm zum Maschinen-Code](#)

## 1.3.1. Instruktionsarchitektur (Instruction Set Architecture, ISA)

Die Befehlsmenge (Instruction Set) umfasst **alle Befehle, die der entsprechende Prozessor bearbeiten kann**:  
Die Operations-Codes (Maschinensprache).

Computer mit **verschiedenen Mikroarchitekturen** können den **gleichen Befehlssatz** bzw. den gleichen Basis-Befehlssatz (mit unterschiedlichen Erweiterungen) unterstützen.



# Beispiel: ISA der Intel 64 und IA-32 Prozessoren

Instruction Set Architecture	Intel 64 and IA-32 Processor Support
General Purpose	All Intel 64 and IA-32 processors
x87 FPU	Intel486, Pentium, Pentium with MMX Technology, Celeron, Pentium Pro, Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors
x87 FPU and SIMD State Management	Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors
MMX Technology	Pentium with MMX Technology, Celeron, Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors
SSE Extensions	Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors

Quelle: Intel 64 and IA-32 Architecture Software Developer's Manual, Volume 1: Basic Architecture, Feb. 2008  
<http://www.intel.com/products/processor/manuals/index.htm>

## Beispiel: ISA der Intel 64 und IA-32 Prozessoren (contd)

Instruction Set Architecture	Intel 64 and IA-32 Processor Support
SSE2 Extensions	Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors
SSE3 Extensions	Pentium 4 supporting HT Technology (built on 90nm process technology), Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Xeon processor 7100 Series
SSSE3 Extensions	Intel Xeon processor 3000, 3200, 5100, 5300, 7300 series, Intel Core 2 Extreme processors QX6000 series, Intel Core 2 Duo, Intel Core 2 Quad processors, Intel Pentium Dual-Core processors
SSE4.1 Extensions	Intel Xeon processor 5200, 5400 series, Intel Core 2 Extreme processors QX9000 series.
IA-32e mode: 64-bit mode instructions	All Intel 64 processors
System Instructions	All Intel 64 and IA-32 processors
VMX Instructions	All Intel 64 and IA-32 processors supporting Intel Virtualization Technology
SMX Instructions	Intel Core 2 Duo processor E6x50

# Data Transfer Instructions (Intel 64 and IA-32)

The data transfer instructions move data between memory and the general-purpose and segment registers. They also perform specific operations such as conditional moves, stack access, and data conversion.

MOV	Move data between general-purpose registers; move data between memory and general-purpose or segment registers;
CMOVE/CMOVZ	move immediates to general-purpose registers
CMOVNE/CMOVNZ	Conditional move if equal/Conditional move if zero
CMOVA/CMOVNB	Conditional move if not equal/Conditional move if not zero
CMOVAE/CMOVNB	Conditional move if above/Conditional move if not below or equal
CMOVBE/CMOVNAE	Conditional move if above or equal/Conditional move if not below
CMOVBE/CMOVNA	Conditional move if below/Conditional move if not above or equal
CMOVGE/CMOVNLE	Conditional move if below or equal/Conditional move if not above
CMOVGE/CMOVNL	Conditional move if greater/Conditional move if not less or equal
CMOVL/CMOVNGE	Conditional move if greater or equal/Conditional move if not less
CMOVL/CMOVNG	Conditional move if less/Conditional move if not greater or equal
CMOVLE/CMOVNG	Conditional move if less or equal/Conditional move if not greater
CMOVC	Conditional move if carry
CMOVNC	Conditional move if not carry
CMOVO	Conditional move if overflow
CMOVNO	Conditional move if not overflow
CMOVS	Conditional move if sign (negative)
CMOVNS	Conditional move if not sign (non-negative)
CMOVP/CMOVPE	Conditional move if parity/Conditional move if parity even
CMOVNP/CMOVPO	Conditional move if not parity/Conditional move if parity odd
XCHG	Exchange
BSWAP	Byte swap
XADD	Exchange and add
CMPXCHG	Compare and exchange
CMPXCHG8B	Compare and exchange 8 bytes
PUSH	Push onto stack
POP	Pop off of stack
PUSHA/PUSHAD	Push general-purpose registers onto stack
POPA/POPAD	Pop general-purpose registers from stack
CWD/CDQ	Convert word to doubleword/Convert doubleword to quadword
CBW/CWDE	Convert byte to word/Convert word to doubleword in EAX register
MOVSX	Move and sign extend
MOVZX	Move and zero extend

# ISAs – Eine Auswahl

## ISAs, die in Hardware implementiert wurden:

- Alpha
- ARM
- Burroughs B5000/B6000/B7000 series
- IA-64 (Itanium)
- MIPS
- Motorola 68k
- PA-RISC
- IBM 700/7000 series
  - System/360
  - System/370
  - System/390
  - z/Architecture
- Power Architecture
  - POWER
  - PowerPC
- PDP-11
  - VAX
- SPARC
- SuperH
- Tricore
- Transputer
- UNIVAC 1100/2200 series
- x86
  - IA-32 (i386, Pentium, Athlon)
  - x86-64 (64-bit Obermenge von IA-32)
- EISC (AE32K)

## ISAs, üblicherweise in Software implementiert, einige Inkarnationen in Hardware:

**p-Code** (UCSD p-System Version III on Western Digital Pascal MicroEngine)  
**Java virtual machine** (ARM Jazelle, PicoJava, JOP)  
**FORTH**

## ISAs, die nicht in Hardware implementiert wurden:

**ALGOL** object code  
**SECD machine**, a virtual machine used for some functional programming languages.  
**MMIX**, a teaching machine used in Donald Knuth's *The Art of Computer Programming*



# „Undocumented Features“ ?

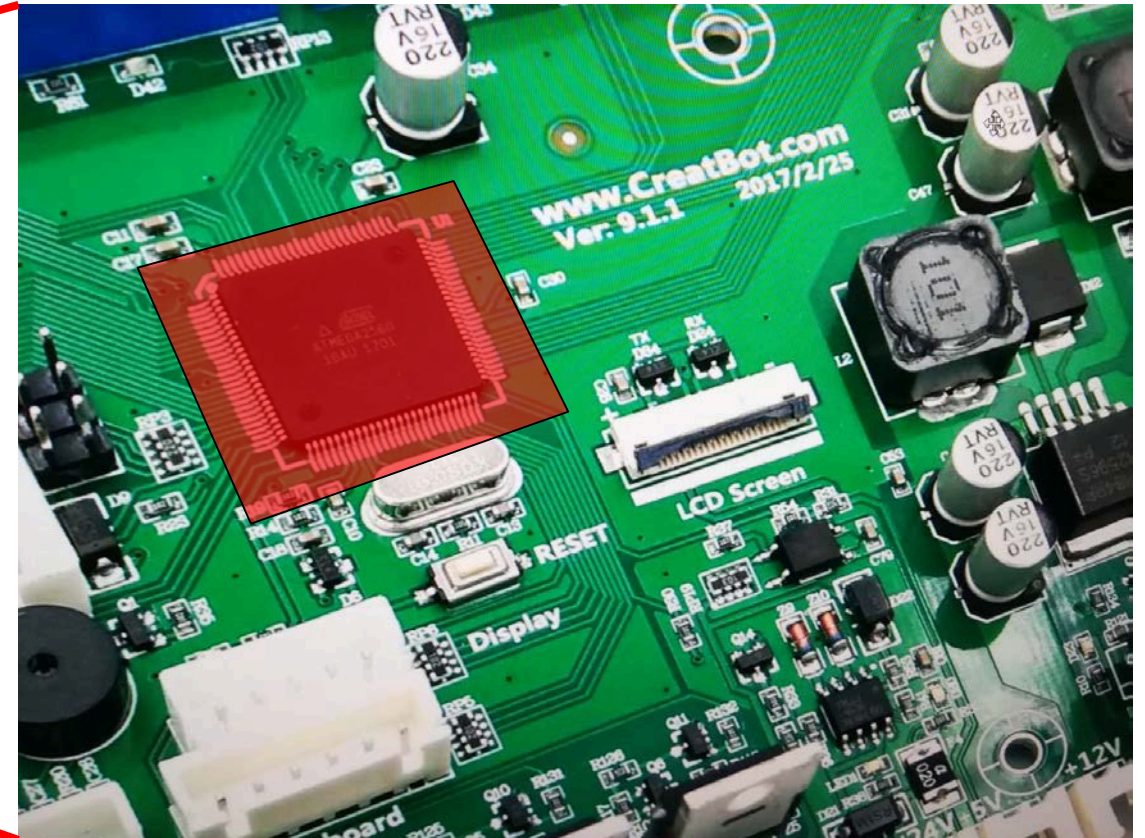
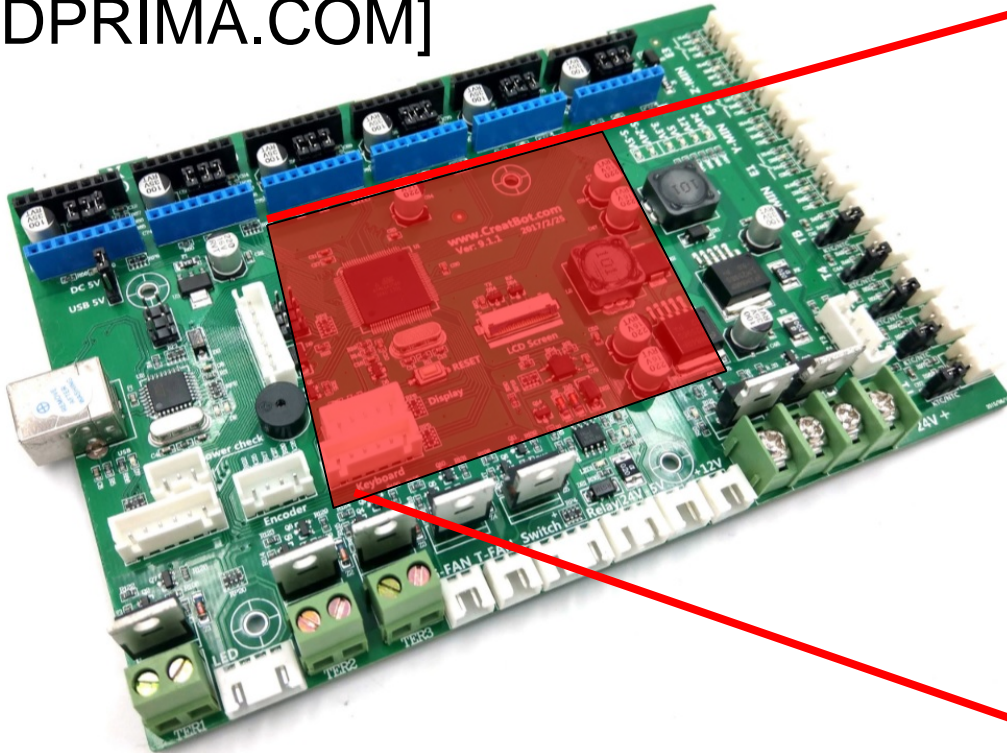
Woher weiß ich, dass ein Chip **nur das tut, was ich in Auftrag gebe**?

Woher weiß ich, dass der Chip das **immer in gleicher Weise** tut?

... Ohne „Kill Switch“ oder andere versteckte Funktionalität?

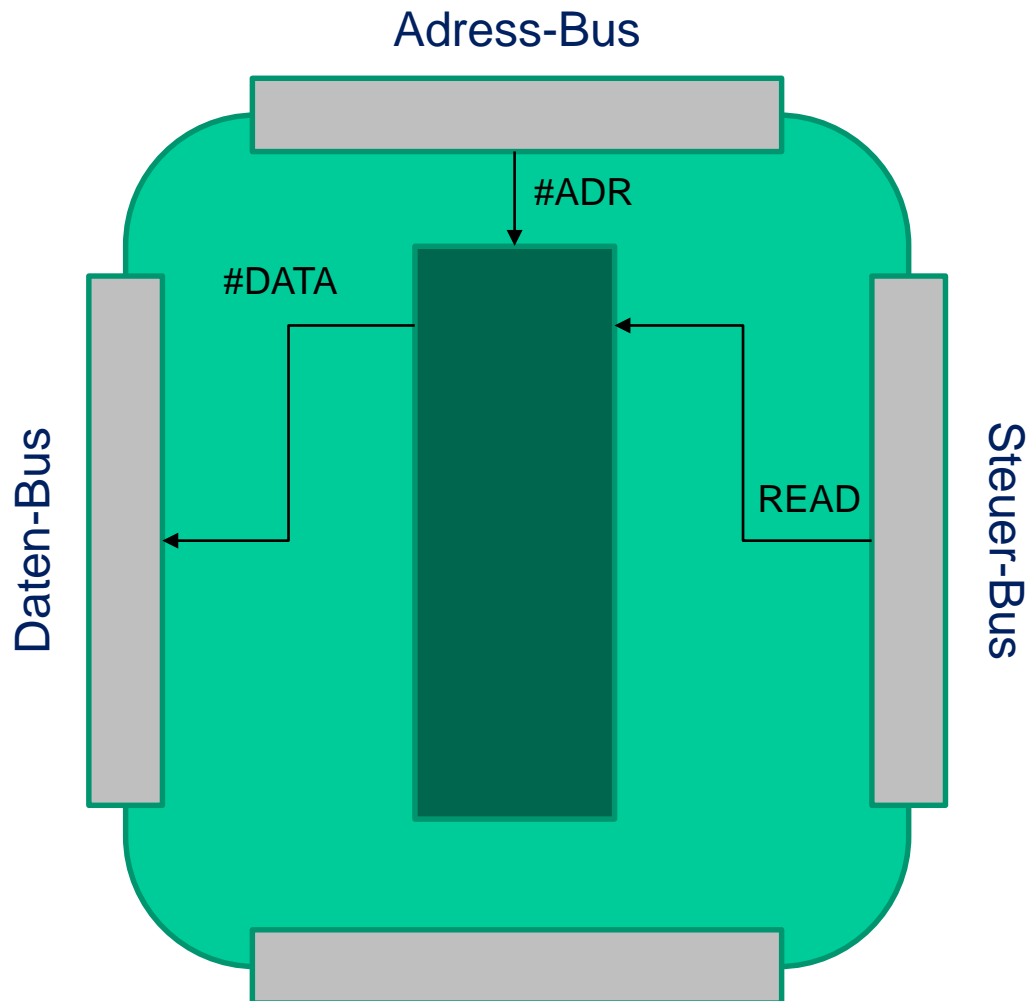
**Beispiel:** Blick auf die Ansteuerung eines 3D-Druckers:

[3DPRIMA.COM]



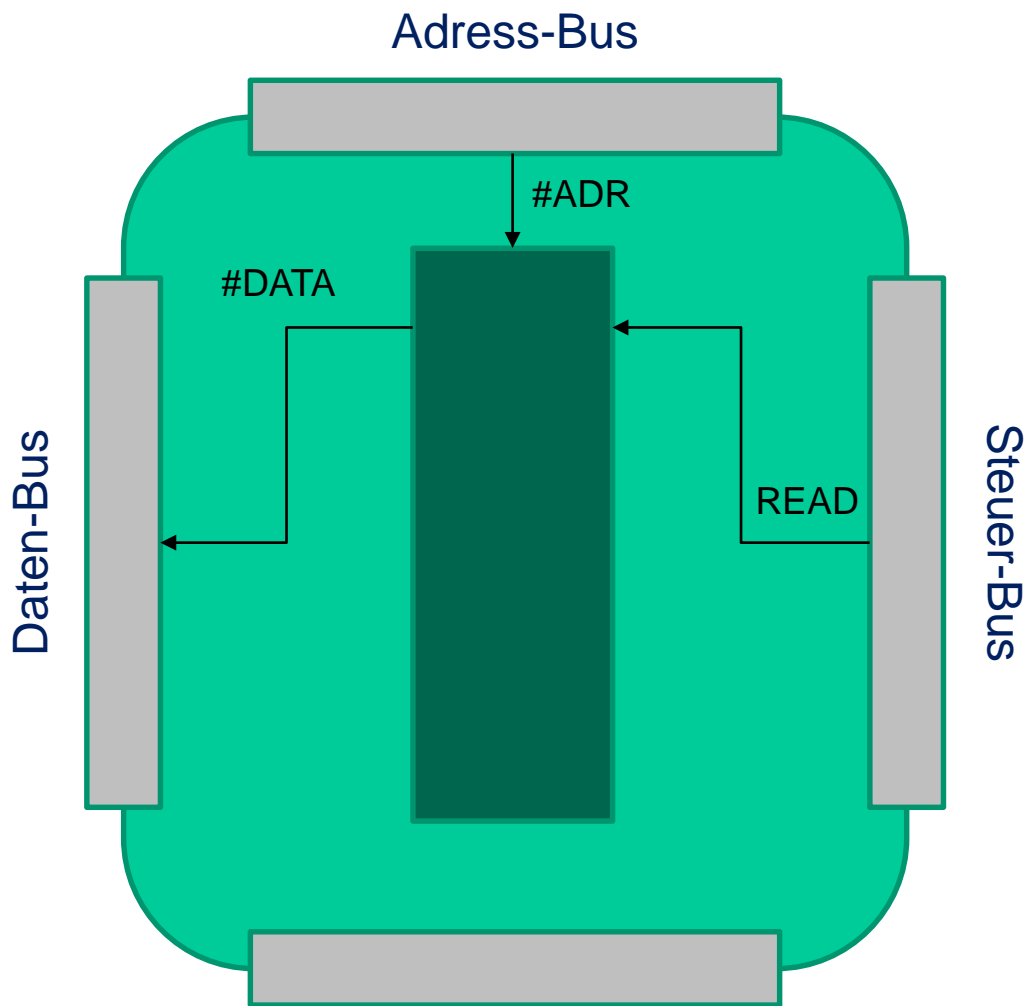


# Spezifikation (SOLL)

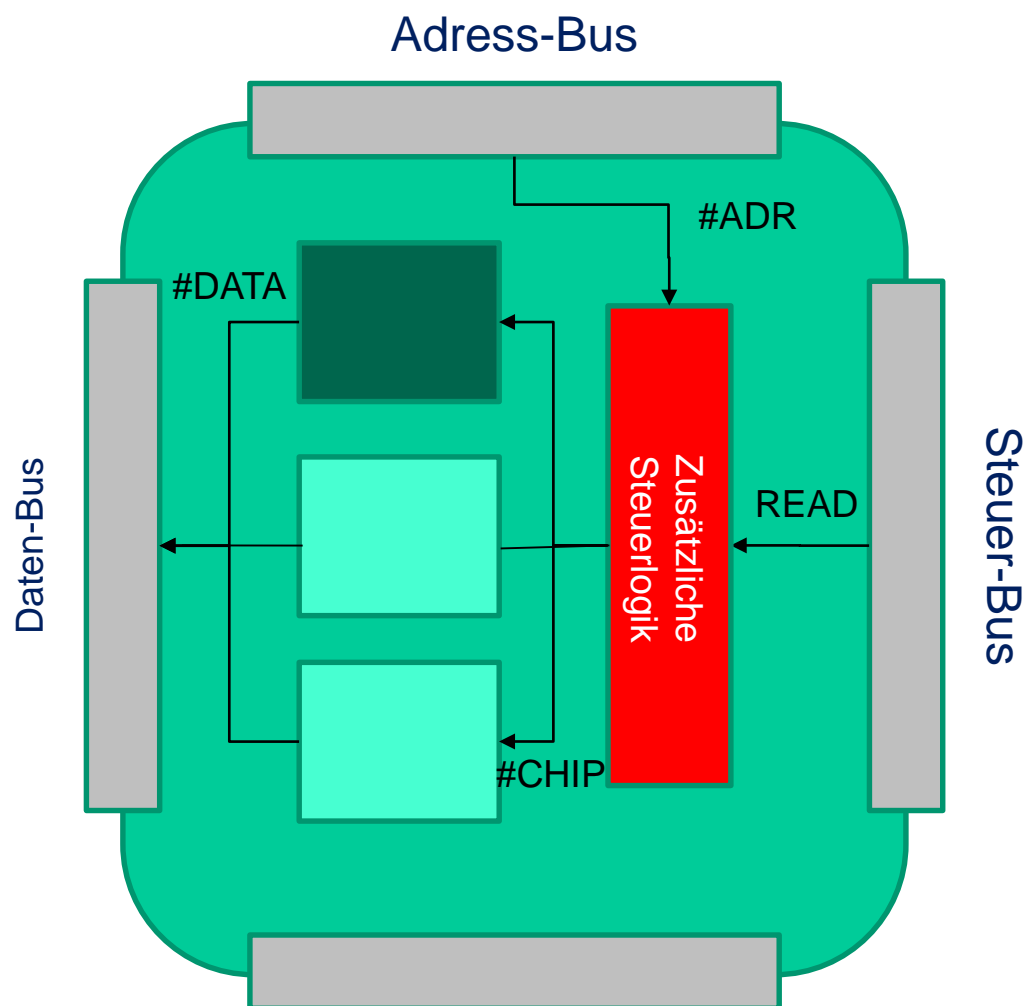


- Funktionalität gem. technischem Datenblatt (Chip-Spezifikation)
- Herausforderungen:
  - Undokumentierte Features
  - Nicht-spezifizierte Adressbereiche
  - Nicht-spezifizierte Steuerbefehle

# Spezifikation (SOLL)

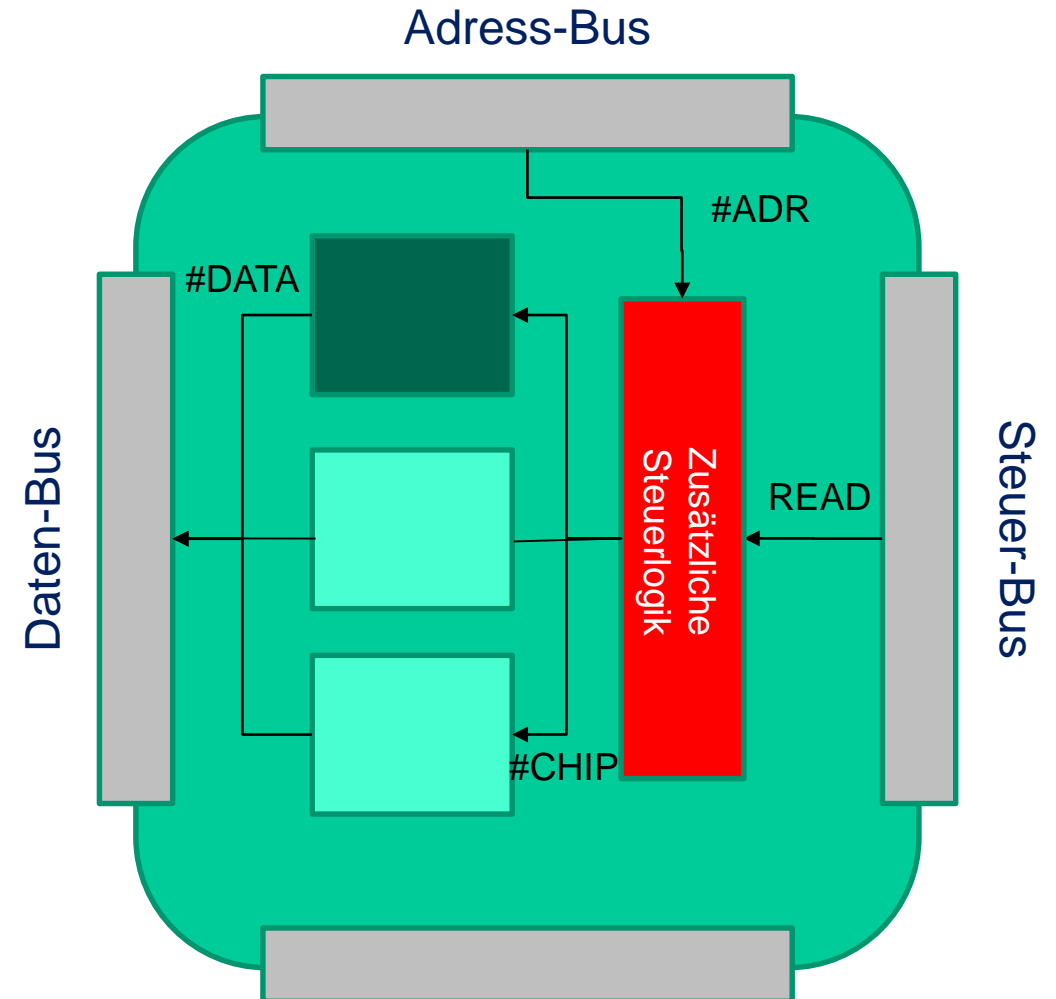


# (IST)??



# Mögliche Implementierung einer „zusätzlichen Steuerlogik“

- Steuer-Bus
  - Bedingt entsprechende „Aktivierungssequenz“ in der Firmware
- Adress-Bus
  - „Aktivierungssequenz“ durch Adressierung (Pre-Defined Patterns, „Port-Knocking“)



## 1.3.2. Prinzipielle Gestalt von Maschinenbefehlen

Maschinenbefehle werden (wie Daten) in Worten gespeichert, manchmal auch in Halbworten, Doppelworten oder Mehrfachworten.

Ein Befehl hat i.W. die folgende Gestalt:

Format	Op-Code	Daten 1	Daten 2	...	Daten k	Ziel	Folge
--------	---------	---------	---------	-----	---------	------	-------

**Format** (kann entfallen, falls Op-Code eindeutig):

- Angabe der **Länge und** der **Positionen der** einzelnen **Felder**.

**Op-Code:**

- Angabe der auszuführenden **Operation**

**Daten** (Wo sind die Operanden?):

- **unmittelbar** („immediate“, Angabe von Konstanten im Maschinencode selbst),
- **implizit** (Akkumulator oder Stack, Art der Bereitstellung folgt aus Op-Code),
- **direkt** (Angabe der Adressdarstellung im Befehl) oder
- **indirekt** (Angabe der Adresse der Speicherzelle, welche die Adresse des Operanden enthält)

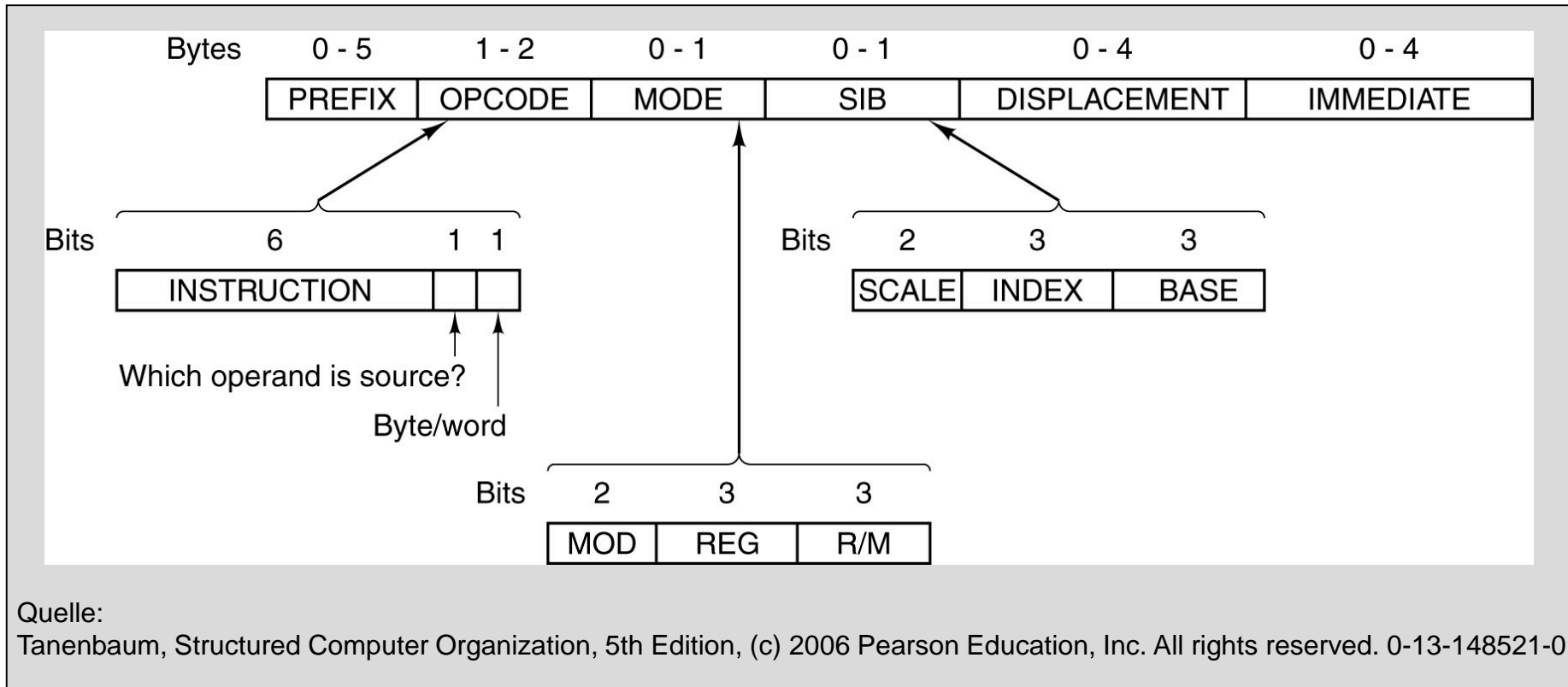
**Ziel** (Wo soll das Ergebnis gespeichert werden?):

- ggf. „Überdeckung“ (eine Quelle ist auch Ziel)
- ggf. „Implizierung“ (z.B. Akkumulator oder Stack als Ziel)

**Folge:**

- **Adresse des nächsten Befehls** („nächster“ Befehl oder Sprung),
- Angabe **entfällt bei sequentieller Abarbeitung**.

# Beispiel: Befehlsformate des Pentium 4



Die Befehlsformate des Pentium 4 spiegeln seine Entwicklung wider, insbes. die Fehlentscheidungen früherer Jahre (**Abwärtskompatibilität!**). So ergab sich

- **hohe Komplexität**
- **große Unregelmäßigkeit / variable Felder / z.T. vielfache Optionen**

Verwendet ein Befehl mehrere Operanden, dann darf **höchstens einer im Speicher** stehen.

# 1.3.3. Assembler-Sprachen

Die sog. „Assembler\*-Sprachen“ sind **maschinenorientierte Programmiersprachen**.

Im Gegensatz zu reinen Maschinensprachen gestatten Sie es dem Menschen, **statt** der **binär dargestellten Befehle** sehr viel leichter verständliche **mnemotechnische\*\* Symbole** einzusetzen.

Ein **Befehl einer Assembler-Sprache umfasst**

- **immer** die **Bezeichnung** der durchzuführenden **Operation**,
- **meist** maschinenspezifische **Angaben zu den Operanden**,
- **häufig** eine „**Marke**“ (label, symbolische Adresse) zur Kennzeichnung der Programmzeile.  
Marken können u.a. als Operanden in Sprungbefehlen verwendet werden.

Vor der Ausführung muss ein Assembler-Programm in ein entsprechendes Maschinenprogramm übersetzt werden. Der hierzu erforderliche „**Assemblierer**“ muss

- **Befehle und Operanden in Binärcode umsetzen**,
- **Marken in Adressen umrechnen** und
- bestimmte „**Pseudobefehle**“ **bearbeiten** (z.B. Reservierung von Speicherplatz).

\* to assemble: zusammenbringen, -tragen, -setzen, -bauen, montieren.

\*\*Mnemonik = Mnemotechnik: Die Kunst, das Einprägen von Gedächtnisstoff durch Lernhilfen zu erleichtern.

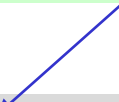


# Pseudoassembler ( $\alpha$ -Notation)

Ein Assembler ist stark maschinenabhängig. Gleiches gilt für die in der zugehörigen Sprache geschriebenen Programme.

Im Rahmen der **Vorlesung** benutzen wir die mnemotechnischen Symbole der sog.  **$\alpha$ -Notation**.

Später lassen wir z.T. auch mehrere Akkumulatoren zu



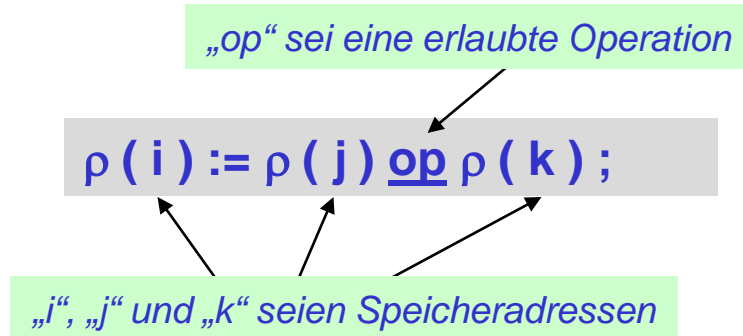
$\alpha$	Akkumulator bzw. dessen Inhalt
$\rho(i)$	Inhalt der Speicherzelle mit Adresse i
$+, -, \cdot, /$	Erlaubte Operationen

Bei der  $\alpha$ -Notation ist die Verwendung von **Marken** zulässig

- sowohl zur Kennzeichnung von **Zellen des Datenspeichers**
- als auch zur Kennzeichnung von **Zellen des Programmspeichers**.

# Wertzuweisungen und Sprungbefehle

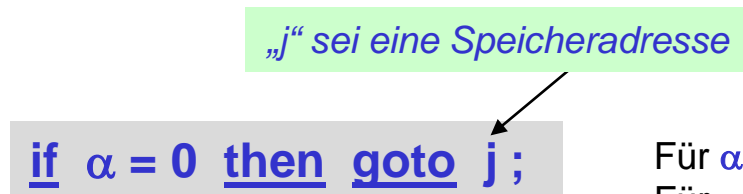
**Wertzuweisungen** haben in der  $\alpha$ -Notation die Gestalt:



## Anmerkungen:

- $\text{op} \in \{ \bullet, /, +, - \}$
- „op“ kann entfallen (dann auch der 2. Operand)
- Statt des Inhaltes einer Speicherzelle ist möglich:
  - Akkumulator oder
  - Konstante

**Als Sprungbefehle seien erlaubt:**



Für  $\alpha = 0$  erfolgt **Sprung** zum Befehl in Zelle j.  
Für  $\alpha \neq 0$  erfolgt **kein Sprung**.

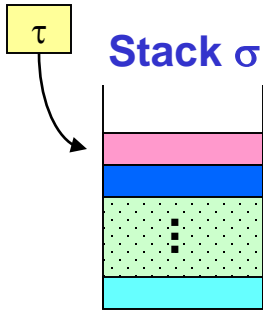
goto j ;

Hier erfolgt in jedem Fall ein **Sprung** zum Befehl in Zelle j.  
Es kann **auch** eine **symbolische Adresse** (z.B. „meinLabel“) verwendet werden.

# Stackoperationen

Ferner wird ein **Stack  $\sigma$**  (Last-in-First-Out, LIFO) mit  $\tau$  als Zeiger auf die oberste Position verwendet.

Operation	Wirkung	Erläuterung
<u>push</u>	$\tau := \tau + 1$ $\sigma(\tau) := \alpha$	Der aktuelle Inhalt des Akkumulators wird auf den Stack gelegt.
<u>pop</u>	$\alpha := \sigma(\tau)$ $\tau := \tau - 1$	Das oberste Element des Stacks wird in Akkumulator geholt.
<u>stack op</u>	$\alpha := \sigma(\tau)$ $\tau := \tau - 1$ $\alpha := \sigma(\tau) \text{ op } \alpha$ $\sigma(\tau) := \alpha$	Die beiden obersten Elemente des Stacks werden gemäß „op“ miteinander verknüpft und durch das Ergebnis der Operation ersetzt.



## 1.3.4. Adressierung

In geringem Umfang können die **Operanden** zu einem Befehl unmittelbar **in dem Befehl selbst oder im darauffolgenden Wort** angegeben werden:



### „Immediate“-Operanden

(Schneller Zugriff auf Konstanten; keine Modifikation während des Programmlaufs ! )

**Mehr Flexibilität** ergibt sich bei **Angabe der Adressen der Speicherzellen**, in denen die Operanden zu finden sind.

[1.3.4.1. Eine, zwei oder drei Adressen ?](#)

[1.3.4.2. Direkte \(= absolute\) Adressierung](#)

[1.3.4.3. Basisregister und Displacement](#)

[1.3.4.4. Indexregister](#)

[1.3.4.5. Indirekte Adressierung](#)

## 1.3.4.1. Eine, zwei oder drei Adressen ?

**Prinzipiell** kann ein Befehl **sehr viele Adressen** von Operanden beinhalten.

**In der Praxis** werden Befehle mit **sehr wenigen Adressen** verwendet, weil

- die **Adresslänge** in direktem Zusammenhang zur **Größe des** adressierbaren **Speicherbereiches** steht,
- **sehr lange Befehle keine effiziente Bearbeitung** zulassen.

**Die häufigsten Befehlsformen sind:**

**3-Adress-Format:**

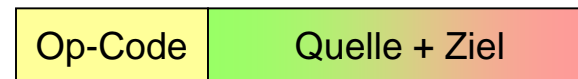


**2-Adress-Format:**



(„Überdeckung“: Der zweite Operand wird zerstört)

**1-Adress-Format:**



(„Überdeckung“: Der Operand wird zerstört; ggf. zusätzlich „Implizierung“)

# Beispiel: Problemlösung mit 3-Adressmaschine

**Gegeben:** Werte  $A, B, C$  in den Speicherzellen  $a, b$  und  $c \in \mathbb{N}$ .

**Ziel:** Berechnung von  $D = (A - B \cdot C) \cdot (A + B \cdot C)$  und Ablage in Zelle  $d$ .

**Sonstiges:** Die Zellen  $h_1, h_2, \dots$  dürfen zur Abspeicherung von **Zwischenergebnissen** genutzt werden. Die Inhalte der Zellen  $a, b$  und  $c$  dürfen **nicht verändert** werden.

## Lösung mittels 3-Adressmaschine

$i:$	$\rho(h_1) := \rho(b) \cdot \rho(c);$	<i>speichert <math>B \cdot C</math> nach <math>h_1</math></i>
$i+1:$	$\rho(h_2) := \rho(a) - \rho(h_1);$	<i>speichert <math>A - B \cdot C</math> nach <math>h_2</math></i>
$i+2:$	$\rho(h_3) := \rho(a) + \rho(h_1);$	<i>speichert <math>A + B \cdot C</math> nach <math>h_3</math></i>
$i+3:$	$\rho(d) := \rho(h_2) \cdot \rho(h_3);$	<i>speichert <math>D</math> nach <math>d</math></i>

## Anmerkungen:

- Hier wurden die **Befehle durchnummeriert**.
- **Allgemein** seien für die **Kennzeichnung von Speicherzellen** beliebige alphanumerische Zeichenketten zulässig. Speicherzellen seien unter ihrem auf diese Art zugeordneten Namen (dem „**Label**“) ansprechbar (z.B. goto meinLabel).



# Beispiel: Problemlösung mit 2-Adressmaschine

Bei einer 2-Adressmaschine gibt es für die Speicherung des Ergebnisses zweistelliger Rechenoperationen **drei Möglichkeiten**:

$$\begin{array}{ll} \alpha := \rho(x) \text{ op } \rho(y) & \text{Implizierung} \\ \rho(x) := \rho(x) \text{ op } \rho(y) & \left. \begin{array}{l} \\ \rho(y) := \rho(x) \text{ op } \rho(y) \end{array} \right\} \text{Überdeckung} \end{array}$$

## Lösung mittels 2-Adressmaschine

i:	$\alpha := \rho(b) \cdot \rho(c);$	<i>speichert <math>B \cdot C</math> in den Akkumulator</i>
i+1:	$\rho(h_1) := \alpha;$	<i>sichert den Akkumulator nach Zelle <math>h_1</math></i>
i+2:	$\alpha := \rho(a) - \rho(h_1);$	<i>speichert <math>A - B \cdot C</math> in den Akkumulator</i>
i+3:	$\rho(d) := \alpha;$	<i>sichert den Akkumulator nach Zelle <math>d</math></i>
i+4:	$\rho(h_1) := \rho(a) + \rho(h_1);$	<i>speichert <math>A + B \cdot C</math> nach <math>h_1</math></i>
i+5:	$\rho(d) := \rho(h_1) \cdot \rho(d);$	<i>speichert <math>D</math> nach <math>d</math></i>

### Anmerkung:

Wir gehen hier davon aus, dass bei Operationen mit zwei Speicherzellen entweder die zweite Speicherzelle oder der Akkumulator überschrieben wird.

# Beispiel: Problemlösung mit 1-Adressmaschine

Die 1-Adressmaschine wird durch Überdeckung und Implizierung realisiert:

## Lösung mittels 1-Adressmaschine

i: $\alpha := \rho ( b ) ;$	<i>speichert B in den Akkumulator</i>
i+1: $\alpha := \alpha \bullet \rho ( c ) ;$	<i>speichert <math>B \bullet C</math> in den Akkumulator</i>
i+2: $\rho ( h_1 ) := \alpha ;$	<i>sichert <math>B \bullet C</math> nach Zelle <math>h_1</math></i>
i+3: $\alpha := \rho ( a ) ;$	<i>speichert A in den Akkumulator</i>
i+4: $\alpha := \alpha - \rho ( h_1 ) ;$	<i>speichert <math>A - B \bullet C</math> in den Akkumulator</i>
i+5: $\rho ( h_2 ) := \alpha ;$	<i>speichert <math>A - B \bullet C</math> nach Zelle <math>h_2</math></i>
i+6: $\alpha := \rho ( a ) ;$	<i>speichert A in den Akkumulator</i>
i+7: $\alpha := \alpha + \rho ( h_1 ) ;$	<i>speichert <math>A + B \bullet C</math> in den Akkumulator</i>
i+8: $\alpha := \alpha \bullet \rho ( h_2 ) ;$	<i>speichert D in den Akkumulator</i>
i+9: $\rho ( d ) := \alpha ;$	<i>speichert D nach d</i>

# Beispiel: Problemlösung mit 0 Adressmaschine

i:	$\alpha := \rho ( a ) ;$	<i>A in den Akkumulator</i>
i+1:	<u>push</u> ;	<i>A ist einziges Element im Stack</i>
i+2:	$\alpha := \rho ( b ) ;$	<i>B in den Akkumulator</i>
i+3:	<u>push</u> ;	<i>B / A sind auf dem Stack</i>
i+4:	$\alpha := \rho ( c ) ;$	<i>C in den Akkumulator</i>
i+5:	<u>push</u> ;	<i>C / B / A sind im Stack</i>
i+6:	<u>stack •</u> ;	<i>B • C / A sind im Stack</i>
i+7:	<u>stack -</u> ;	<i>A - B • C sind im Stack</i>
i+8:	$\alpha := \rho ( a ) ;$	<i>speichert A in den Akkumulator</i>
i+9:	<u>push</u> ;	<i>A / A - B • C sind im Stack</i>
i+10:	$\alpha := \rho ( b ) ;$	<i>speichert B in den Akkumulator</i>
i+11:	<u>push</u> ;	<i>B / A / A - B • C sind im Stack</i>
i+12:	$\alpha := \rho ( c ) ;$	<i>speichert C in den Akkumulator</i>
i+13:	<u>push</u> ;	<i>C / B / A / A - B • C sind im Stack</i>
i+14:	<u>stack •</u> ;	<i>B • C / A / A - B • C sind im Stack</i>
i+15:	<u>stack +</u> ;	<i>A + B • C / A - B • C sind im Stack</i>
i+16:	<u>stack •</u> ;	<i>D ist einziges Element im Stack</i>
i+17:	<u>pop</u> ;	<i>D in den Akkumulator</i>
i+18:	$\rho ( d ) := \alpha ;$	<i>speichert D nach d</i>

## 1.3.4.2. Direkte (= absolute) Adressierung

Bei der sog. „**direkten Adressierung**“ wird die **Adresse** des/der Operanden direkt **im Befehlswort\*** angegeben.

**Beispiel:**



**Vorteil:** **Schneller Zugriff** (schneller als bei indirekter Adressierung)

**Nachteile:**

- **Eingeschränkter Adressraum** bei Angabe im Befehlswort  
Mit  $k$  Adress-Bits sind  $2^k$  Speicherzellen adressierbar.
- **Keine Verschiebung der Daten**  
Da die Adresse als fester Wert im Programm steht.
- **Keine Modifikation von Adressen** beim Programmablauf  
Programme werden mit expliziter Adressierung ggf. sehr unübersichtlich.

\* bzw. unmittelbar im Anschluss an das Befehlswort.

# Beispiel: Multiplikation eines Vektors mit einem Skalar

Ein **Vektor**, der in den **Speicherzellen 0 bis 99** gespeichert ist, wird **mit** dem **Skalar in Zelle 100** multipliziert.

**Direkte Adressierung** macht ein wenig elegantes Programm erforderlich:

```
101:   $\alpha := \rho(100);$ 
102:   $\alpha := \alpha \bullet \rho(0);$ 
103:   $\rho(0) := \alpha;$ 
104:   $\alpha := \rho(100);$ 
105:   $\alpha := \alpha \bullet \rho(1);$ 
106:   $\rho(1) := \alpha;$ 
...
398:   $\alpha := \rho(100);$ 
399:   $\alpha := \alpha \bullet \rho(99);$ 
400:   $\rho(99) := \alpha;$ 
```

### 1.3.4.3. Basisregister und Displacement

Hier wird die Adresse aufgeteilt in:

- **Basisadresse** (gespeichert im „Basisregister“)
- **relative Distanz** („Displacement“)

**Beispiel:**

Op-Code	Adresse des Basisregisters	Displacement
---------	----------------------------	--------------

Die tatsächliche Adresse wird berechnet durch **Addition**

- des im **Basisregister** angegebenen Wertes und
- des im Befehl angegebenen **Displacements**.

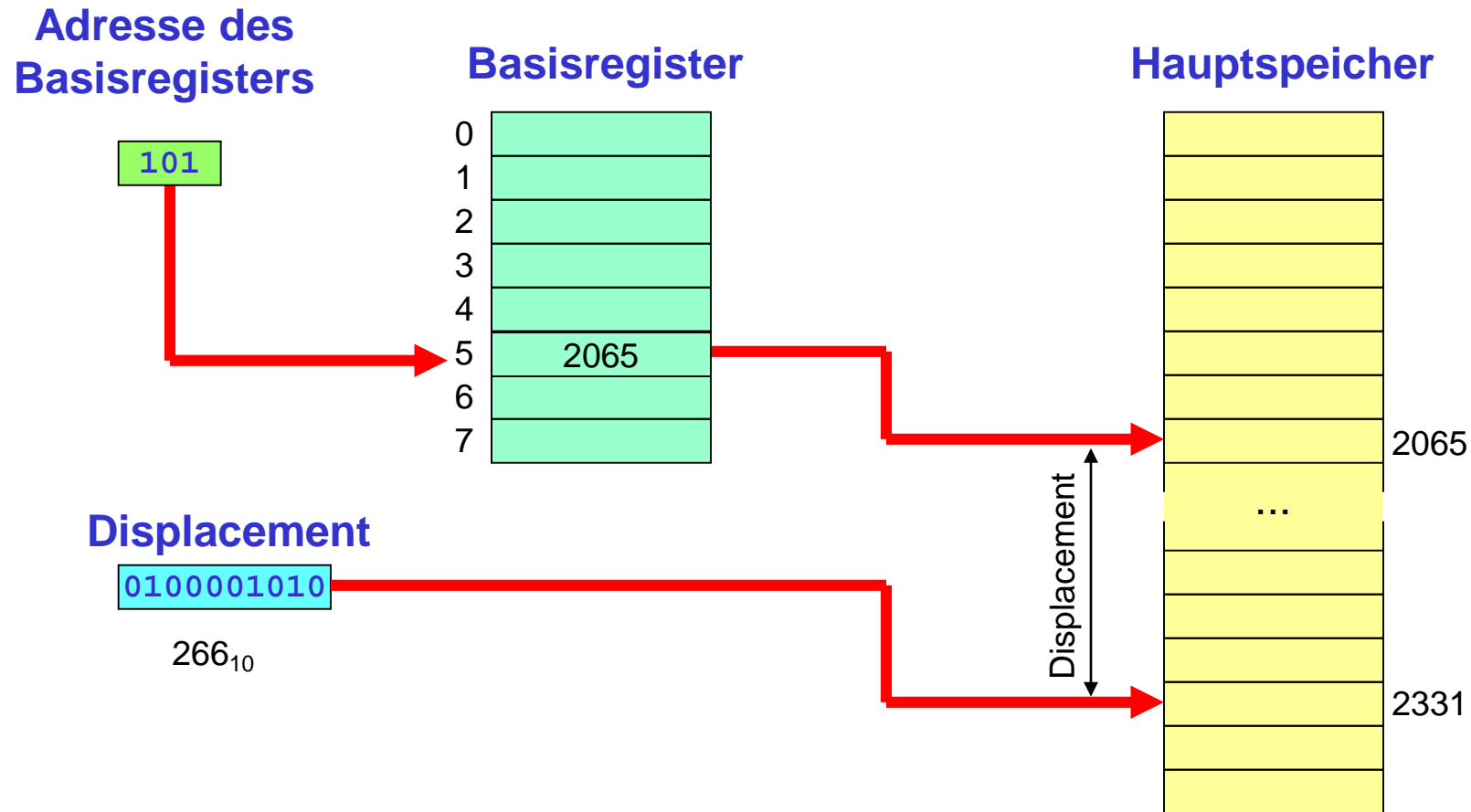
**Hinweis:**

Sind in einem Programm alle Adressen relativ zu einer Basisadresse, dann **kann** das Programm sehr **leicht im Speicher verschoben werden**.

Adressumrechnung erfolgt durch Modifikation des Inhaltes des Basisregisters.



# Beispiel: Basisregister und Displacement



## Basisregister ermöglichen

- die **leichte Verschiebbarkeit** von Programmen
- die Realisierung eines **sehr großen Adressraums**.

## 1.3.4.4. Indexregister

### Bisher ungelöstes Problem:

- **Adressmodifikation** durch das aktuell laufende Programm  
(Basisregister sind hierfür nicht verfügbar, da sie unter Kontrolle des Betriebssystems zur Lösung des Adressierungsproblems bei Programmverschiebung im Speicher eingesetzt werden).

### Idee:

Wähle als zusätzliche additive Komponente ein sog. „**Indexregister**“ (nachfolgend: „ $\gamma$ “).

### Beispiel:



Die **tatsächliche Adresse** ergibt sich nun als **Summe aus**

- dem **Inhalt des Indexregisters** und
- der (ggf. relativen) „**Adresse**“.

Im Folgenden gehen wir davon aus, dass das **Indexregister** - analog zum Akkumulator - **mit seinem Inhalt identifiziert** wird:

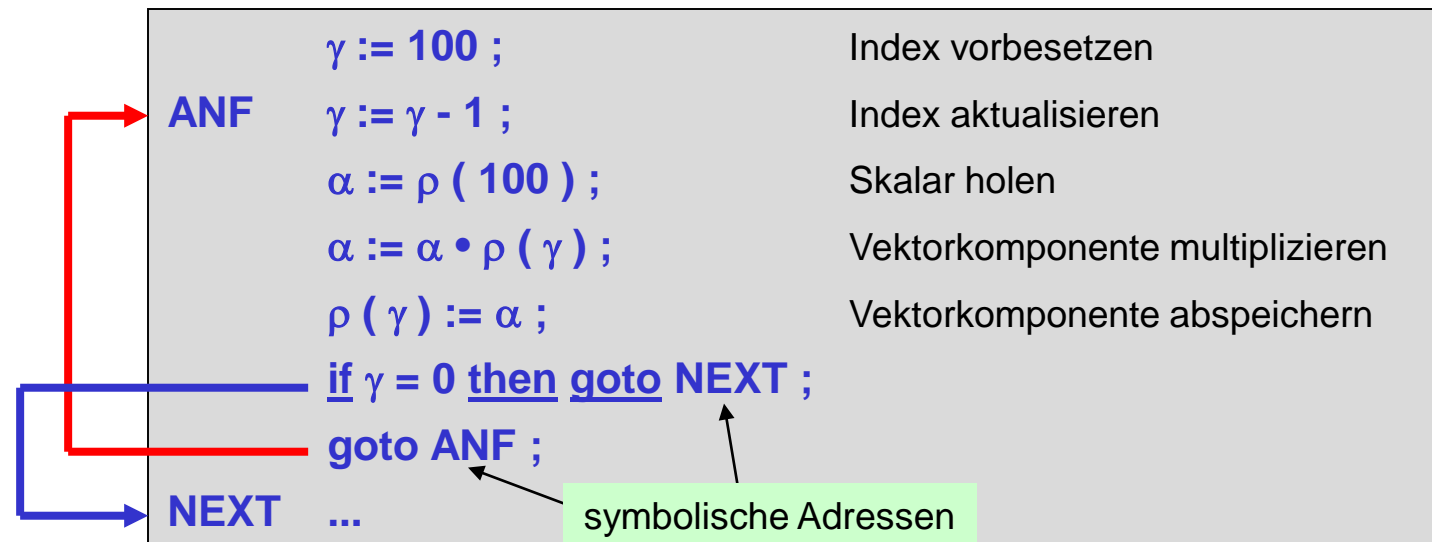
$\gamma$  : Inhalt des Indexregisters

$\rho(\gamma)$  : Inhalt der Speicherzelle, auf die der Inhalt des Indexregisters zeigt.  
(„Register indirekt“).

# Beispiel: Einsatz eines Indexregisters

Bei Einsatz eines Indexregisters kann das aus dem Abschnitt „direkte Adressierung“ bekannte Programm zur **Multiplikation eines Vektors mit einem Skalar** stark vereinfacht werden. Als Befehle seien neben Wertzuweisung und Arithmetik mit Konstanten zulässig:

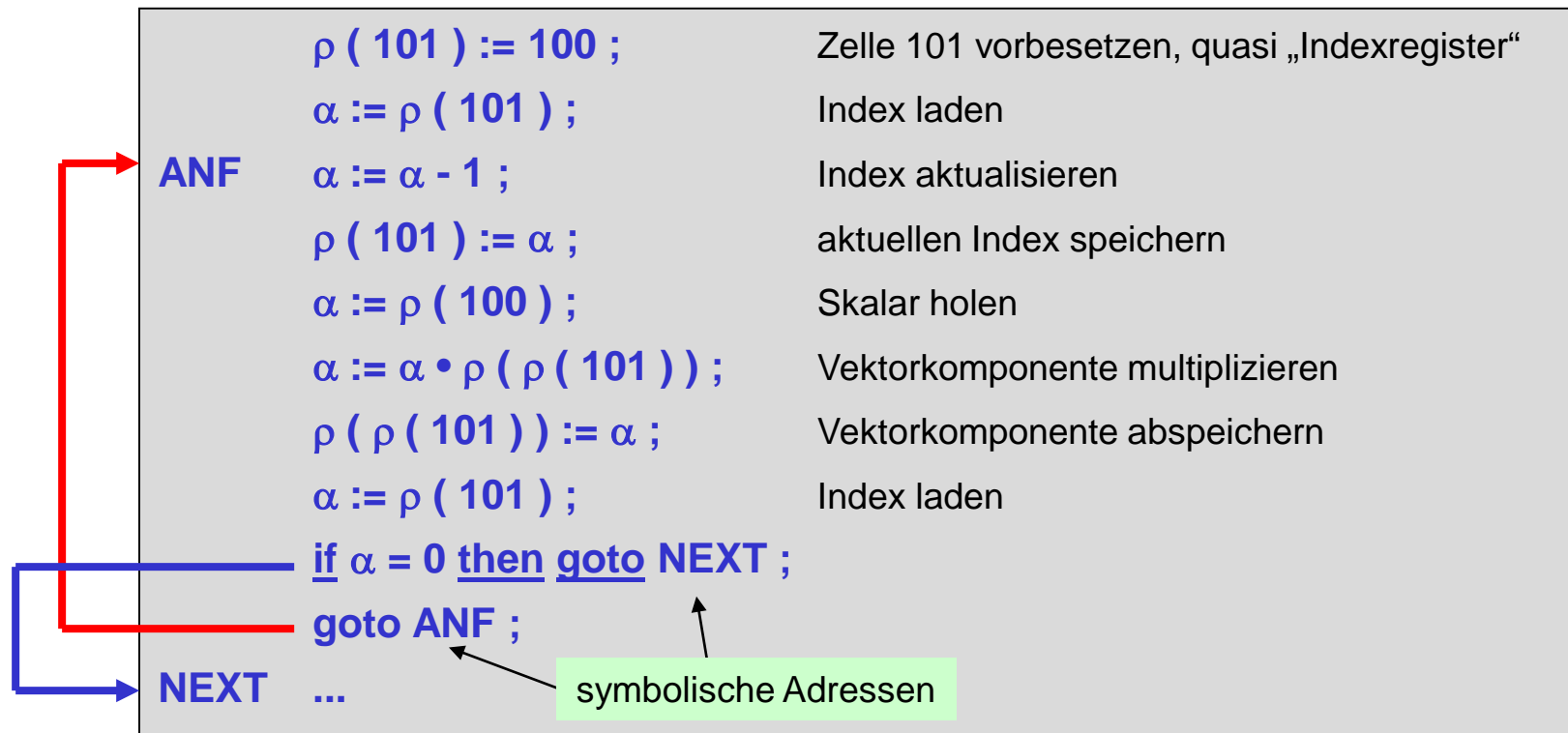
$\alpha := \rho(i);$	$\alpha := \rho(\gamma);$	$\rho(i) := \alpha;$	$\rho(\gamma) := \alpha;$	$\alpha := \gamma;$	$\gamma := \alpha;$
$\alpha := \alpha \text{ op } \rho(i);$	$\alpha := \alpha \text{ op } \rho(\gamma);$	$i \in \mathbb{N}; \text{ op } \in \{ \cdot, /, +, - \}$			
<u>goto</u> j ;	<u>if</u> $\alpha = 0$ <u>then goto</u> j ;	<u>if</u> $\gamma = 0$ <u>then goto</u> j ;			



## 1.3.4.5. Indirekte Adressierung

Es ist möglich, auch den **Inhalt beliebiger Speicherzellen als Adresse** zu **interpretieren**. Für das nachfolgende Beispiel seien folgende Befehle zulässig:

$\alpha := \rho(i);$        $\alpha := \rho(\rho(i));$        $\rho(i) := \alpha;$        $\rho(\rho(i)) := \alpha;$   
 $\alpha := \alpha \text{ op } \rho(i);$        $\alpha := \alpha \text{ op } \rho(\rho(i));$        $i \in \mathbb{N}; \text{ op } \in \{ \bullet, /, +, - \}$   
goto j ;      if  $\alpha = 0$  then goto j ;



Bei indirekter Adressierung wird der Inhalt der Speicherzelle häufig durch „**Autoinkrement**“ bzw. „**Autodekrement**“ modifiziert.

## 1.3.5. Unterprogramme

1.3.5.1. Grundlegende Betrachtungen

1.3.5.2. Einstufige Unterprogramme

1.3.5.3. Mehrstufige, nicht erneut aufrufbare Unterprogramme

1.3.5.4. Mehrstufige, erneut aufrufbare Unterprogramme

## 1.3.5.1. Grundlegende Betrachtungen

Offenbar muss - spätestens - nach Ausführung eines Befehls bekannt sein, **in welcher Speicherzelle der nächste Befehl** zu finden ist:

- Die **nächste Zelle** im Programmspeicher (bei sequentieller Bearbeitung).  
Achtung: Befehle können unterschiedliche Länge haben!
- Die im aktuellen Befehl **adressierte Zelle** (z.B. bei goto-Anweisung).

Schwieriger wird die Situation bei **Einsatz von Unterprogrammen**:

### Aufruf eines Unterprogramms („call“):

1. „**Rette**“ die Rücksprungadresse (nächster Befehl bei sequentieller Bearbeitung)
2. „**Springe**“ zum ersten Befehl des Unterprogramms.

### Rückkehr aus einem Unterprogramm („return“):

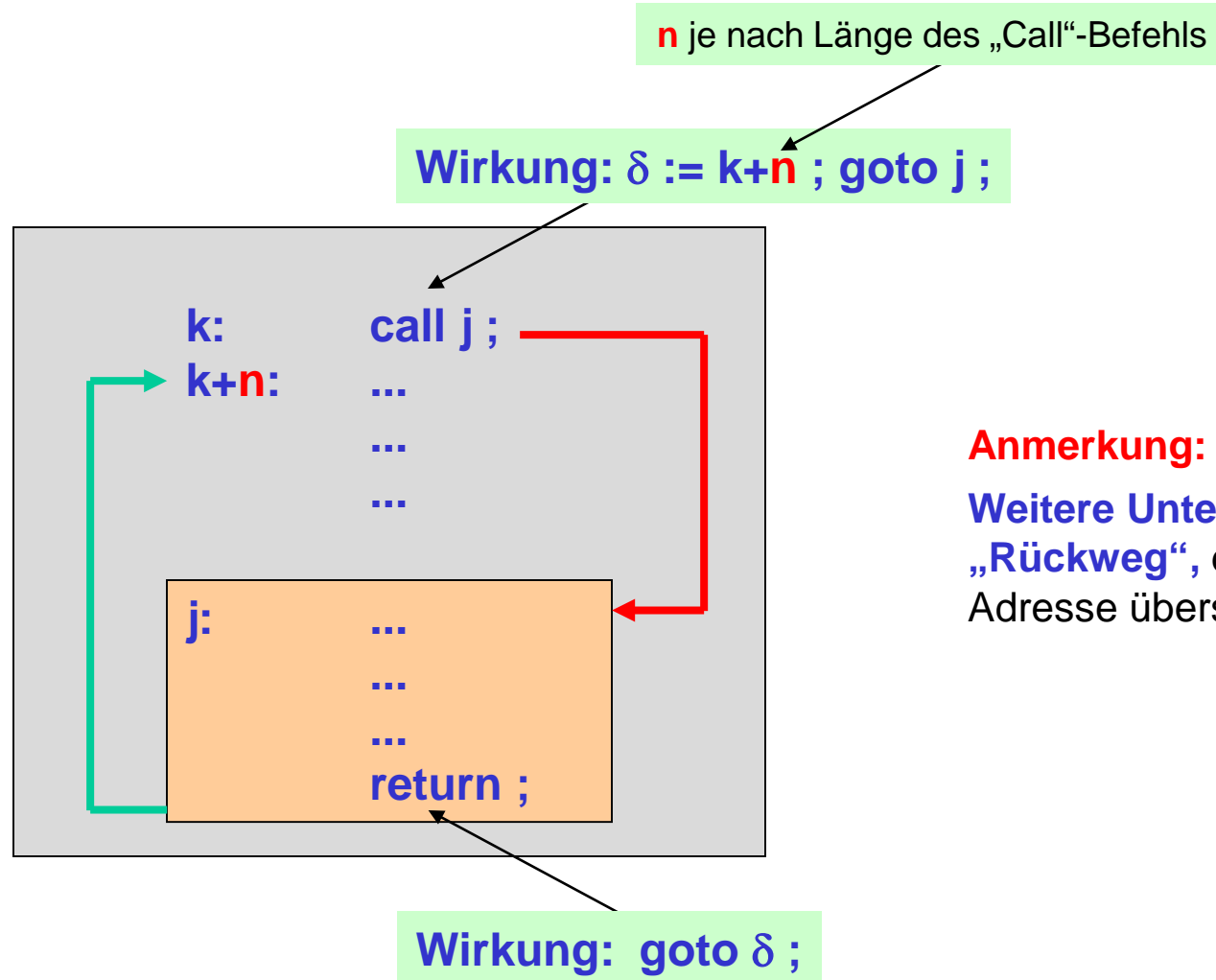
1. „**Hole**“ die zugehörige Rücksprungadresse
2. „**Springe**“ zurück in das Programm, aus dem der Aufruf erfolgte.

**Anmerkung:** Auch die **Datenübergabe** muss in geregelter Form erfolgen.



## 1.3.5.2. Einstufige Unterprogramme

Eine sehr einfache Realisierung von Unterprogrammtechnik kann dadurch erfolgen, dass ein **spezielles Register  $\delta$**  zur „Rettung“ der Rücksprungadresse eingesetzt wird.



### Anmerkung:

Weitere Unterprogrammaufrufe zerstören den „Rückweg“, da das Register  $\delta$  mit einer neuen Adresse überschrieben wird.

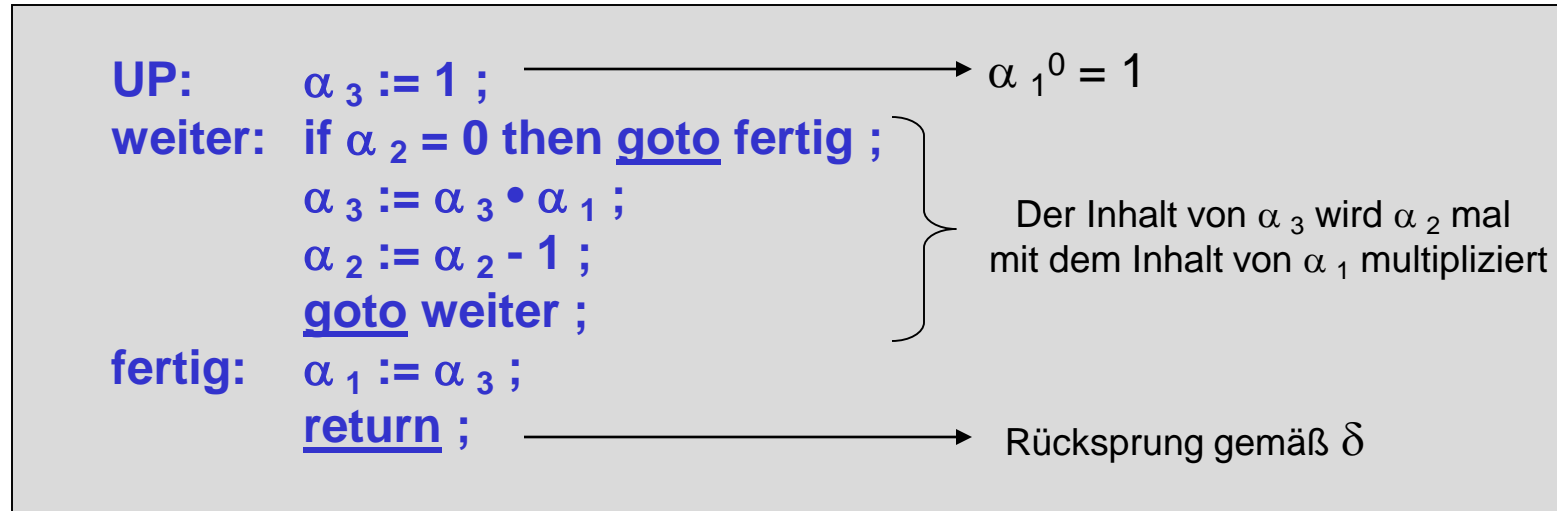
# Beispiel: Einstufige Unterprogrammtechnik

## Aufgabenstellung:

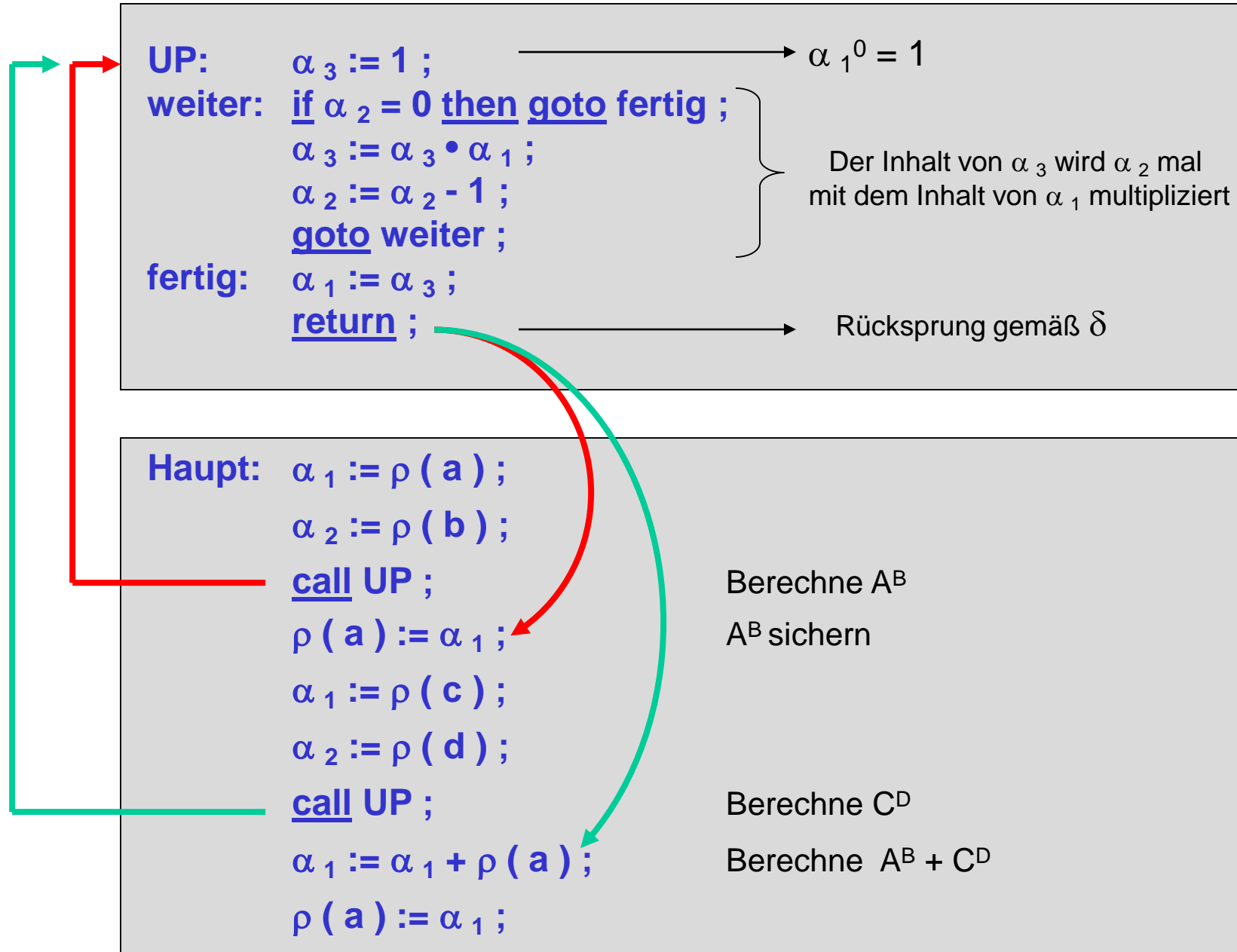
- Die Speicherzellen a, b, c und d seien den Variablen A, B, C und D zugeordnet.
- Berechne  $A^B + C^D$ .
- Speichere das **Ergebnis in Speicherzelle a**.
- Die Register  $\alpha_1, \alpha_2, \dots, \alpha_8$  seien **frei verfügbar**.

Wir schreiben zunächst ein **Unterprogramm**, das  $\alpha_1^{\alpha_2}$  berechnet.

Das Register  $\alpha_1$  wird auch zur Ausgabe eingesetzt:

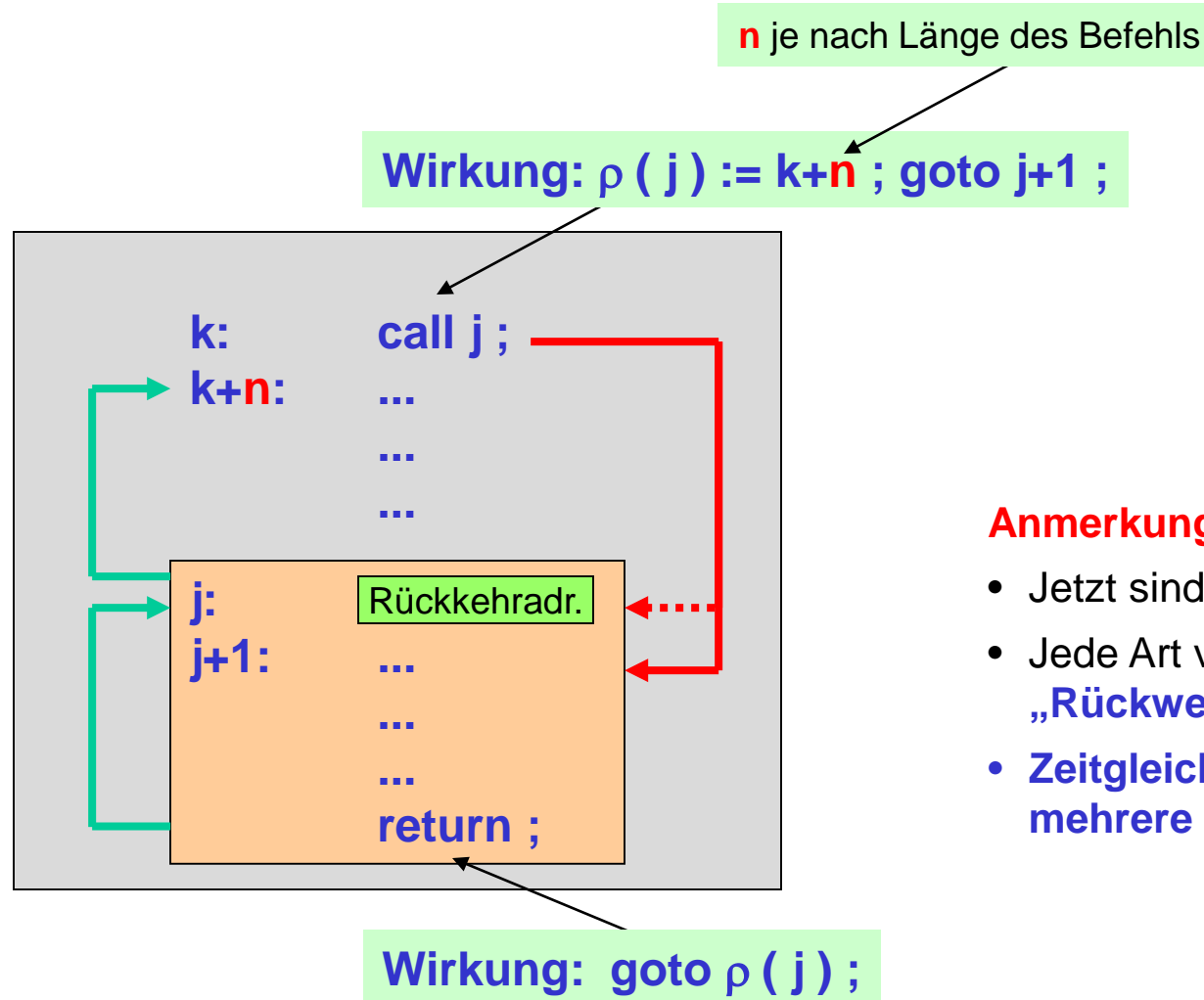


## Beispiel: Einstufig (2)



### 1.3.5.3. Mehrstufige, nicht erneut aufrufbare Unterprogramme

Wird die **Rücksprungadresse in der ersten Zelle des Unterprogramms** gespeichert (Unterprogramm beginnt mit Leerzelle), dann ergibt sich mehr Flexibilität:



#### Anmerkungen:

- Jetzt sind **weitere Unterprogramme aufrufbar**.
- Jede Art von **Rekursion** (direkt oder indirekt) **zerstört den „Rückweg“**.
- **Zeitgleiche Verwendung** des Unterprogramms **durch mehrere Programme** würde im **Chaos** enden.

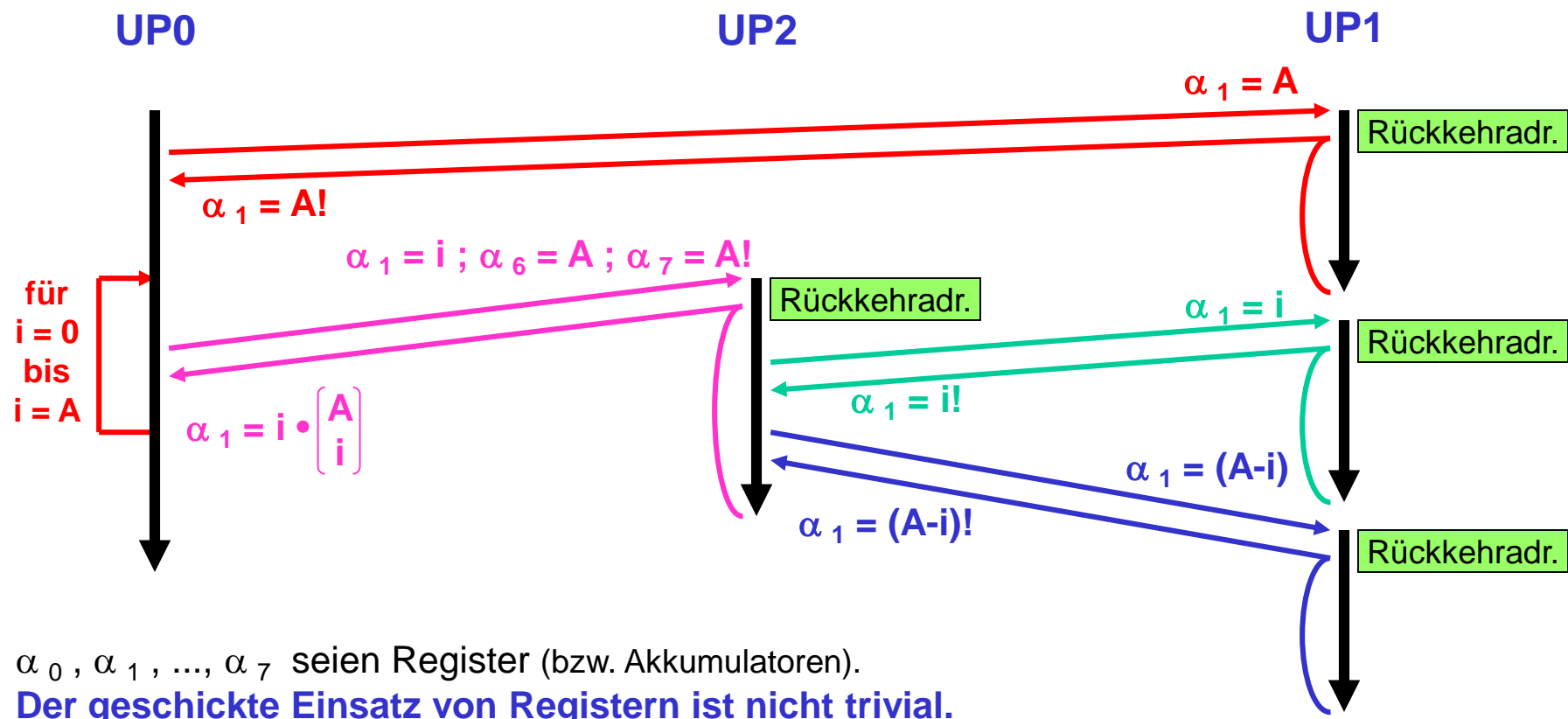
# Beispiel: Mehrstufig, nicht erneut aufrufbar

In Speicherzelle a stehe der Wert A. Es soll

$$\sum_{i=0}^A i \binom{A}{i} \text{ mit } \binom{A}{i} = \frac{A!}{i! (A-i)!}$$

mittels Unterprogramm „UP0“ berechnet und in Zelle a abgelegt werden.

Wir lösen das Problem mit mehrstufigen Unterprogrammen:



$\alpha_0, \alpha_1, \dots, \alpha_7$  seien Register (bzw. Akkumulatoren).  
**Der geschickte Einsatz von Registern ist nicht trivial.**

## Beispiel: Mehrstufig, nicht erneut aufrufbar(2)

UP1: Rückkehradresse

$\alpha_2 := 1 ;$

Hilfsregister

Weiter: if  $\alpha_1 \leq 1$  then goto Fertig ;  $0! = 1! = 1$

$\alpha_2 := \alpha_2 \cdot \alpha_1 ;$

$\alpha_1 := \alpha_1 - 1 ;$

Multiplikator dekrementieren

goto Weiter ;

Fertig:  $\alpha_1 := \alpha_2 ;$

Ergebnis in den Akkumulator 1

return ;

UP2: Rückkehradresse

$\alpha_4 := \alpha_1 ;$

Akkumulator 4 als Hilfsregister; enthält i

call UP1 ;

i! berechnen

$\alpha_5 := \alpha_1 ;$

Hilfsregister; enthält i!

$\alpha_1 := \alpha_6 - \alpha_4 ;$

Setze  $\alpha_1$  auf (A-i)

call UP1 ;

(A-i)! berechnen

$\alpha_1 := \alpha_1 \cdot \alpha_5 ;$

i! (A-i)! berechnen

$\alpha_1 := \alpha_7 / \alpha_1 ;$

A über i berechnen

$\alpha_1 := \alpha_1 \cdot \alpha_4 ;$

berechne „i mal A über i“

return ;

## Beispiel: Mehrstufig, nicht erneut aufrufbar (3)

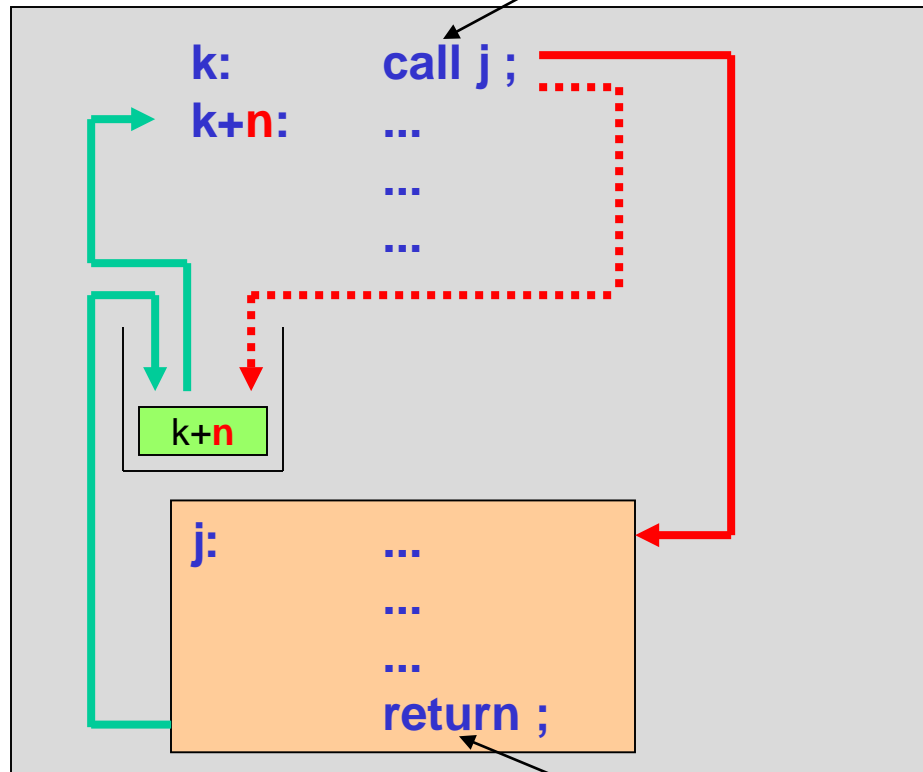
<b>Haupt:</b>	$\alpha_6 := \rho(a);$ $\rho(a) := 0;$ $\alpha_1 := \alpha_6;$ <u>call</u> <b>UP1</b> ; $\alpha_7 := \alpha_1;$ $\alpha_0 := 0;$	<b>A</b> in den Akkumulator 6 laden $\rho(a)$ initialisieren für Addition Akkumulator 1 erhält <b>A</b> Akkumulator 1 erhält <b>A</b> ! Akkumulator 7 erhält <b>A</b> ! Initialisiere Zähler für <b>i</b> (Schleifensteuerung)
<b>Loop:</b>	<u>if</u> $\alpha_0 > \alpha_6$ <u>then goto</u> <b>Vor</b> ; $\alpha_1 := \alpha_0;$ <u>call</u> <b>UP2</b> ; $\rho(a) := \rho(a) + \alpha_1;$ $\alpha_0 := \alpha_0 + 1;$ <u>goto</u> <b>Loop</b> ;	Ist <b>i</b> > <b>A</b> ? Aktuelles <b>i</b> in den Akkumulator 1 laden Aktuellen Summanden berechnen Zur bisherigen Zwischensumme hinzu addieren Zähler <b>i</b> inkrementieren
<b>Vor:</b>	<u>return</u> ;	Berechnung beendet

#### 1.3.5.4. Mehrstufige, erneut aufrufbare Unterprogramme

Wird die **Rücksprungadresse in einem Stack** gespeichert, dann ergibt sich noch mehr Flexibilität.

**n** je nach Länge des Befehls

Wirkung: push k+n ; goto j ;



**Wirkung:** `pop address ; goto address;`

### Anmerkungen:

- Jetzt ist auch **Rekursion möglich**.
- Ein **Stack hat** in der Praxis **eine maximale Größe**. Bei Überschreitung wird der Rückweg zerstört.
- Bei **Einsatz mehrerer Stacks** sind **mehrere Kontrollflüsse** möglich.



# Beispiel 3: Mehrstufig, erneut aufrufbar

Obwohl **Rekursion zur Berechnung von  $n!$**  wenig sinnvoll ist (warum ?), eignet sich eine rekursive Lösung zur Veranschaulichung wesentlicher Aspekte der Technik erneut aufrufbarer Unterprogramme.

## Verwendung der Register:

- $\alpha_1$ : Parameterübergabe
- $\alpha_2$ : (vorbesetzter) Stackpointer
- $\alpha_3$ : Hilfsregister

Adressen im Programmspeicher  
(„Offset“)

Die Lösung verwendet einen selbst verwalteten Daten-Stack mit dem Stack-Pointer  $\alpha_2$ .

<b>Fakultaet:</b>	<u>if</u> $\alpha_1 = 0$ <u>then goto</u> Vor ;	0	$0! = 1$
<b>Daten-Stack aufbauen</b>	$\alpha_2 := \alpha_2 + 1 ;$	1	Stackpointer vorbereiten
	$\rho(\alpha_2) := \alpha_1$	2	$n - j$ auf den Stack legen
	$\alpha_1 := \alpha_1 - 1 ;$	3	$n - j - 1$ bilden
	<u>call</u> Fakultaet ;	4	
<b>Berechne:</b>	$\alpha_3 := \rho(\alpha_2) ;$	5	oberstes Stackelement holen ...
<b>Daten-Stack abarbeiten</b>	$\alpha_1 := \alpha_1 \cdot \alpha_3 ;$	6	... und zu Akkumulator multiplizieren
	$\alpha_2 := \alpha_2 - 1 ;$	7	Stackpointer vorbereiten (Daten-Stack abbauen)
	<u>return</u> ;	8	... wird häufig ausgeführt.
<b>Vor:</b>	$\alpha_1 := 1 ;$	9	... da $0! = 1$ ; beendet die Rekursion bei $\alpha_1 = 0$
	<u>return</u> ;	10	... wird einmal ausgeführt.

# Beispiel 3: Mehrstufig, erneut aufrufbar (2)

Bei der hier gezeigten Lösung werden zwei Stacks verwendet:

- **Ein Stack für die  $n - j$  („selbst“)**  
Rettung des aktuellen Wertes von Akkumulator 1
- **Ein Stack für die Rücksprungadressen („automatisch“)**  
Rettung des Befehlszählers (PC = Program Counter)

Sei SP ein Pointer auf das oberste Element im Stack der Rücksprungadressen.

Dann bewirkt ein **call-Befehl**:

**SP := SP + 1 ;**

Erhöhe Stackpointer

**$\rho ( SP ) := PC + n ;$**

Sichere Befehlszähler

**PC := „Adresse“ ;**

Setze Befehlszähler  
Springe ins Unterprogramm

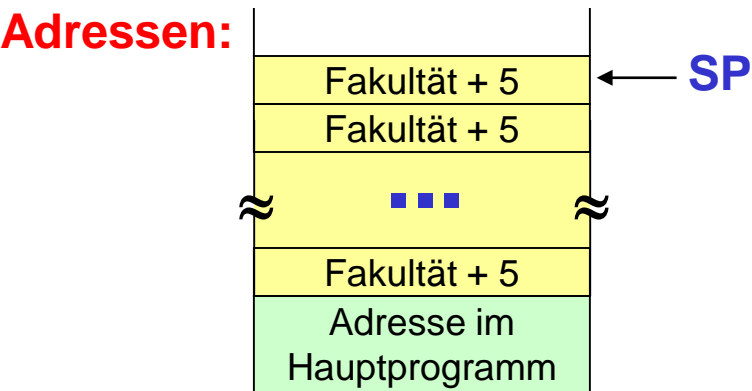
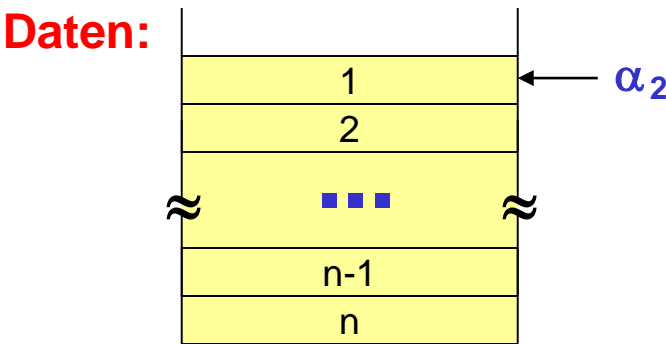
Ein **return-Befehl** hat die Wirkung:

**PC :=  $\rho ( SP )$  ;**

Setze Befehlszähler

**SP := SP - 1 ;**

Setze Stackpointer  
Springe „zurück“

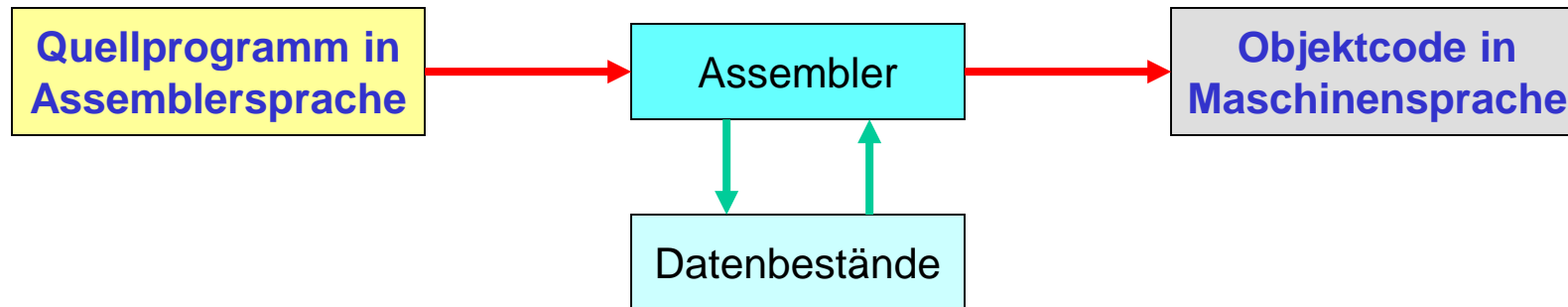


## 1.3.6. Vom Assemblerprogramm zum Maschinen-Code

Programme in Assemblersprache sind

- zwar **maschinennah**,
- aber **nicht unmittelbar lauffähig** (da Assemblersprachen zu „luxuriös“ sind).

Die Übersetzung in ein Objektprogramm (d.h. in Maschinsprache) erfolgt durch ein spezielles Programm: „**Assembler**“ (**Assembler**).



Neben der Verarbeitung von „Pseudo-Operationen“ (z.B. Auflösung externer Referenzen) übernimmt der Assembler i.w. die folgenden Aufgaben:

- **Umsetzung** der mnemotechnischen Darstellung **in Binärcode**,
- **Umsetzung** symbolischer Adressen („Label“) **in Speicheradressen**,
- **Erzeugung von Daten** (Umwandlung von Literalen in Binärdarstellung der Daten).

# Ein Durchgang ist nicht genug

Der typischer Assembler benötigt mindestens zwei Durchgänge:

## 1. Durchgang:

- Bestimmung der **Länge** von **Maschineninstruktionen**
- **Verwaltung** eines **Adresszählers** (Befehle und Daten)
- **Zuordnung symbolischer Adressen**
- **Zuordnung** von **Literalen**
- Verarbeitung einiger Assembler-Instruktionen

## 2. Durchgang:

- **Heranziehung** der **Symbolwerte** (= Speicheradressen)
- **Erzeugung** von **Maschineninstruktionen**
- **Erzeugung** von **Daten** (= Konstanten)
- Verarbeitung der restlichen Assembler-Instruktionen

## Benötigte Tabellen:

- Tabelle der **Maschineninstruktionen** (statisch)
- Tabelle der **Assemblerinstruktionen** (statisch)
- **Symboltabelle** (dynamisch)
- ggf. **weitere Tabellen** (z.B. Basisregistertabelle)

# Makros

Assembler-Programmierung macht häufig die Wiederholung von Code-Blöcken erforderlich. Hier können „Makros“ zum Einsatz kommen:

Ein **Makro** fasst **mehrere Befehle oder Deklarationen** zu einer Einheit zusammen.

Einem Makro wird bei seiner Definition ein eindeutiger **Bezeichner zugeordnet**.

Wo immer dieser Bezeichner im Programmtext auftaucht, wird er vom Assembler bei der „**Makroexpansion**“ durch den zugehörigen Text ersetzt, bevor die Umwandlung in Maschinencode vorgenommen wird.

Es ist auch möglich, **ähnliche Makrobefehle** zu **parametrisieren**: Die Parameter werden dann jeweils durch die aktuellen Parameter ersetzt.

## Beispiel: Definition eines Makros

<b>MACRO;</b>	Pseudo-Operation
<b>Potenzieren &amp;Bas, &amp;Exp</b>	Übergabe symbolischer Adressen
$\alpha_3 := \rho ( \&Bas ) ;$	} Setzen der Akkumulatoren und Aufruf des Unterprogramms
$\alpha_2 := \rho ( \&Exp ) ;$	
<u>call</u> UP ;	
<b>MEND ;</b>	Pseudo-Operation

# Beispiel: Einsatz von Makros

**MACRO ;**  
**Potenzieren &Bas, &Exp**

$\alpha_1 := \rho ( \&Bas ) ;$

$\alpha_2 := \rho ( \&Exp ) ;$

call UP ;

**MEND ;**

Pseudo-Operation

Übergabe symbolischer Adressen

} Setzen der Akkumulatoren und  
Aufruf des Unterprogramms

Pseudo-Operation

UP:  $\alpha_3 := 1 ;$   $\longrightarrow \alpha_1^0 = 1$

weiter: if  $\alpha_2 = 0$  then goto fertig ;

$\alpha_3 := \alpha_3 \cdot \alpha_1 ;$

$\alpha_2 := \alpha_2 - 1 ;$

goto weiter ;

fertig:  $\alpha_1 := \alpha_3 ;$

return ;

} Der Inhalt von  $\alpha_3$   
wird  $\alpha_2$  mal  
mit dem Inhalt  
von  $\alpha_1$  multipliziert

Haupt: **Potenzieren a, b ;**

$\rho ( a ) := \alpha_1 ;$

Sichere  $A^B$

**Potenzieren c, d ;**

$\alpha_1 := \alpha_1 + \rho ( a ) ;$

Berechne  $A^B + C^D$

$\rho ( a ) := \alpha_1 ;$

## Anmerkung:

Durch Einsetzen und Aktualisieren der Parameter entsteht das bereits bekannte Hauptprogramm.

# 1.4. Virtuelle Maschinen

Die ISA stellt in gewissem Sinne eine **Virtualisierung** dar: Es ist möglich, die zugehörigen **Befehle auf unterschiedlichster Hardware** auszuführen, ohne dass der Nutzer dies bemerkt: Er nutzt eine **virtuelle CPU**.

Dieses Konzept der Virtualisierung kann massiv erweitert werden:

- Virtuelle CPU,
- Virtueller Speicher,
- Virtuelle Ein-/Ausgabegeräte (z.B. Drucker, virtuell sind sie viel schneller...),
- ...

können auf einem nur einem Computer mehrere parallele „**Ausführungsumgebungen**“ (execution environments) entstehen lassen:



**Die Illusion völlig getrennter Maschinen mit jeweils eigener Hardware.**



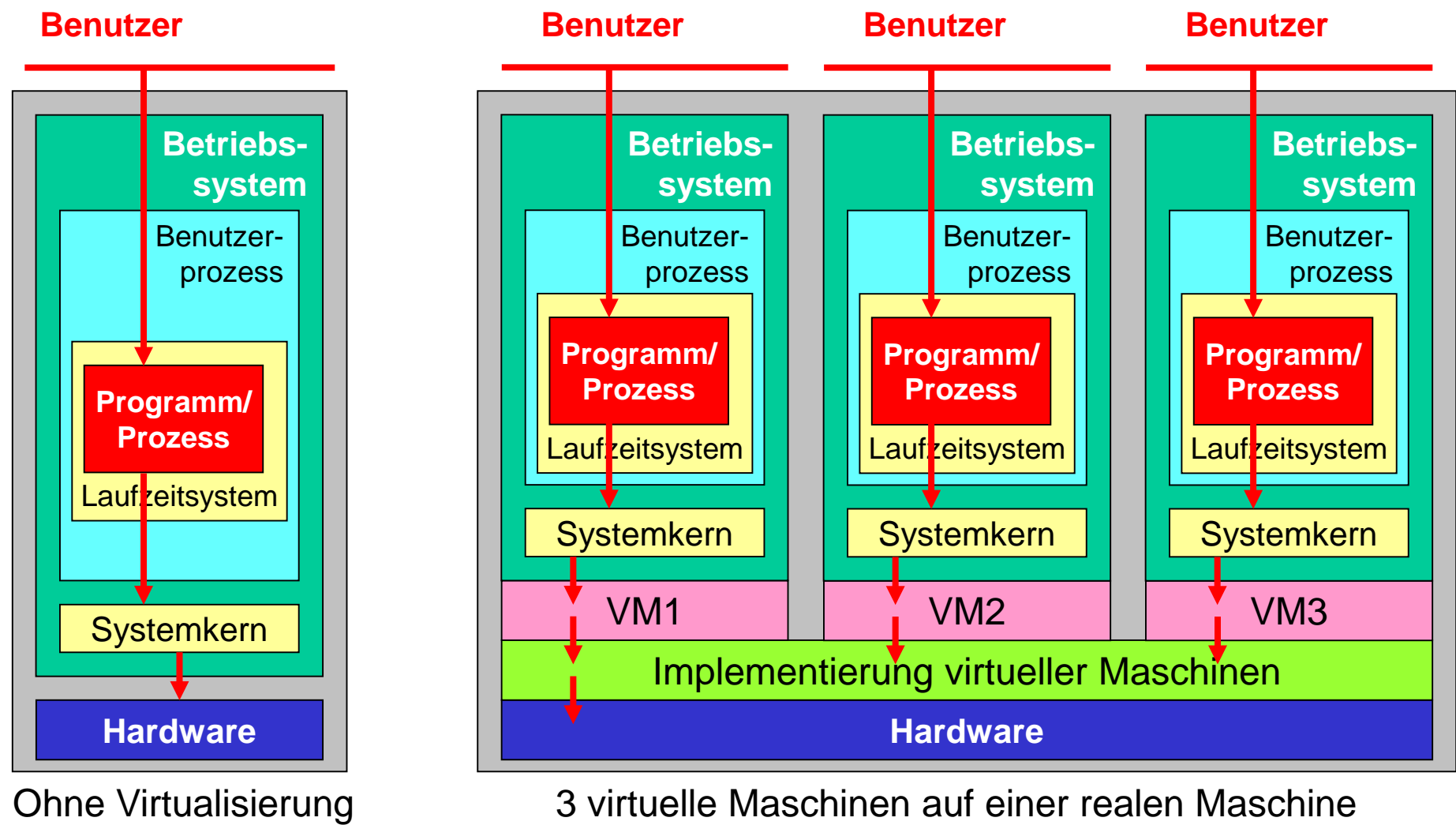
Das Programm erhält zu seiner Ausführung die **eigene (virtuelle) Kopie** eines realen (oder auch nur gedachten) Computersystems.



Die virtuelle Maschine (guest system) kann hierbei massiv vom „Wirtssystem“ (host system) abweichen.

# Nicht-virtuelle und virtuelle Maschine im Vergleich

Bei nicht-virtuellen Maschinen haben Prozesse Zugriff auf reale Ressourcen.  
Bei virtuellen Maschine „tun wir nur so“, als ob dies der Fall wäre.





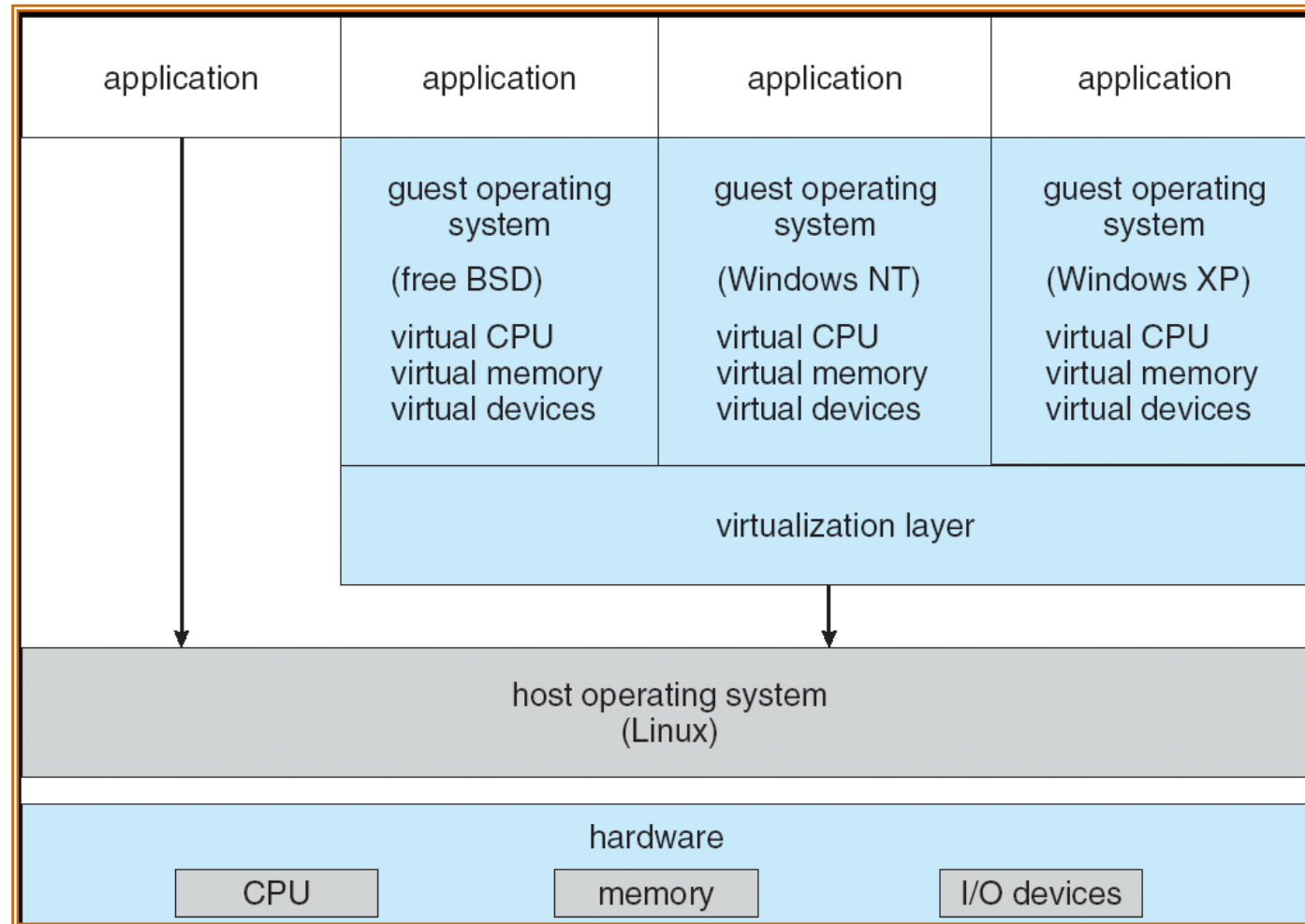
# Vorteile und Nachteile virtueller Maschinen

Vorteile	Nachteile
<b>Vollständige Isolation / Kapselung</b> ➡ Gewinn an Sicherheit	<b>Vollständige Isolation / Kapselung</b> ➡ Ineffiziente Interprozesskommunikation (bei reiner Lehre: kein gemeinsamer Speicher)
	<b>Ineffizienz / zusätzlicher Overhead</b> ➡ Verbrauch von Speicher und CPU-Zeit
<b>„Sandkasten“ zum Experimentieren</b> ➡ Entwicklungsarbeiten am Betriebssystem	
<b>Verschiedene Systeme in einem</b> ➡ z.B. Linux, Windows, Symbian auf einer gemeinsam genutzten realen Maschine ➡ Lösung von Kompatibilitätsproblemen	

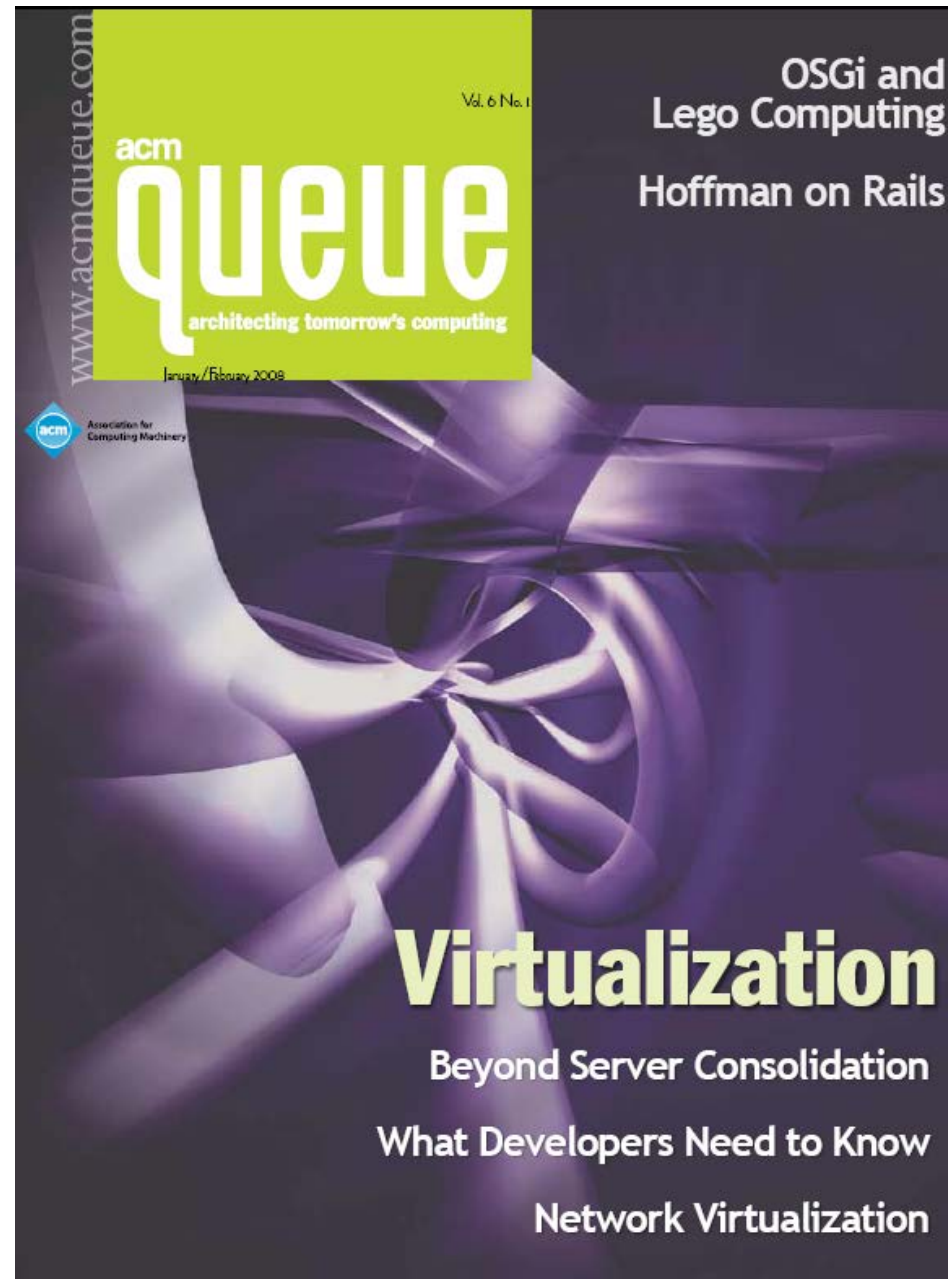
Das Konzept virtueller Maschinen ist seit Jahrzehnten bekannt.

Praktisch bedeutsam wurde es aber erst Ende der 1990er-Jahre mit **VMware** und der **Java Virtual Machine**. Auch das **.NET Framework** basiert auf dem Konzept der virtuellen Maschine.

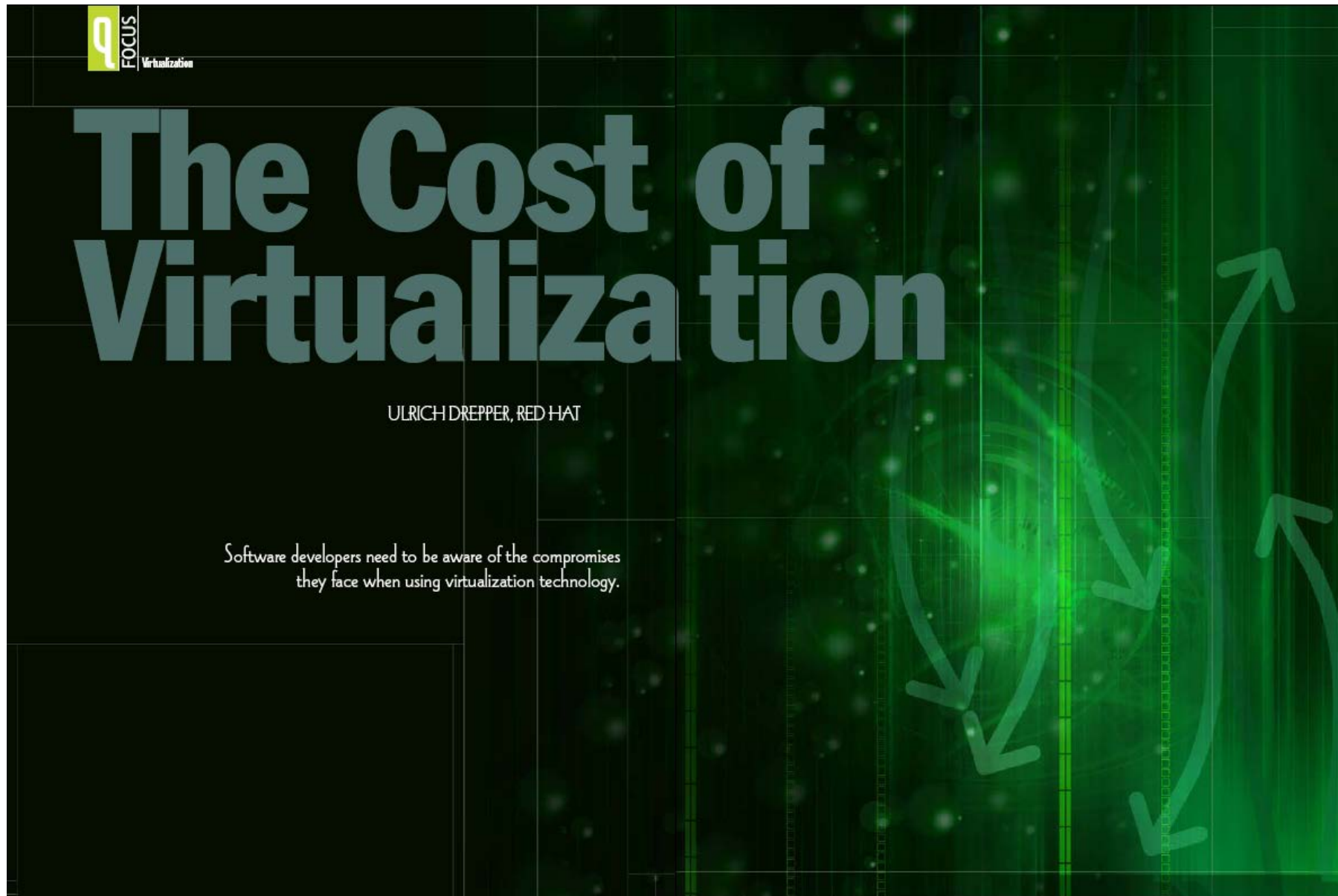
# Die Architektur von VMware (hier: Linux als Wirtssystem)



Quelle: Silberschatz, Galvin, Gagne, "Operating System Concepts"  
... ein sehr empfehlenswertes Buch ...



# TINSTAAFL\*: There Is No Such Thing As A Free Lunch



\*aka: TANSTAAFL (There Ain't No Such Thing As A Free Lunch)