Entwurfsmethodik
Institut für Informatik

Lehrstuhl für
Eingebettete Systeme

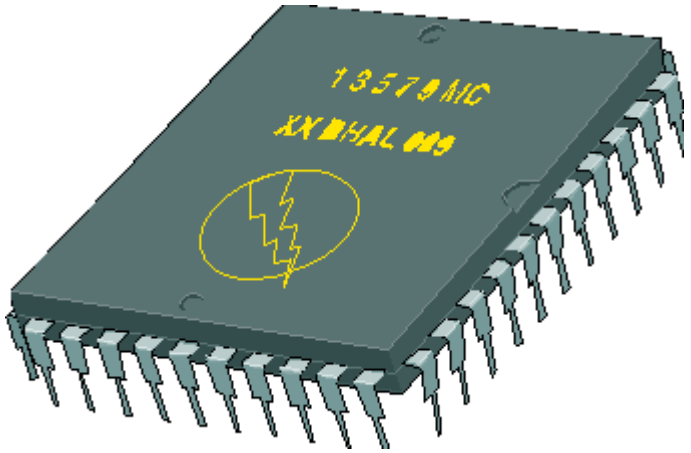
Hardwarearchitekturen und Rechensysteme

2. Rechnerarchitektur

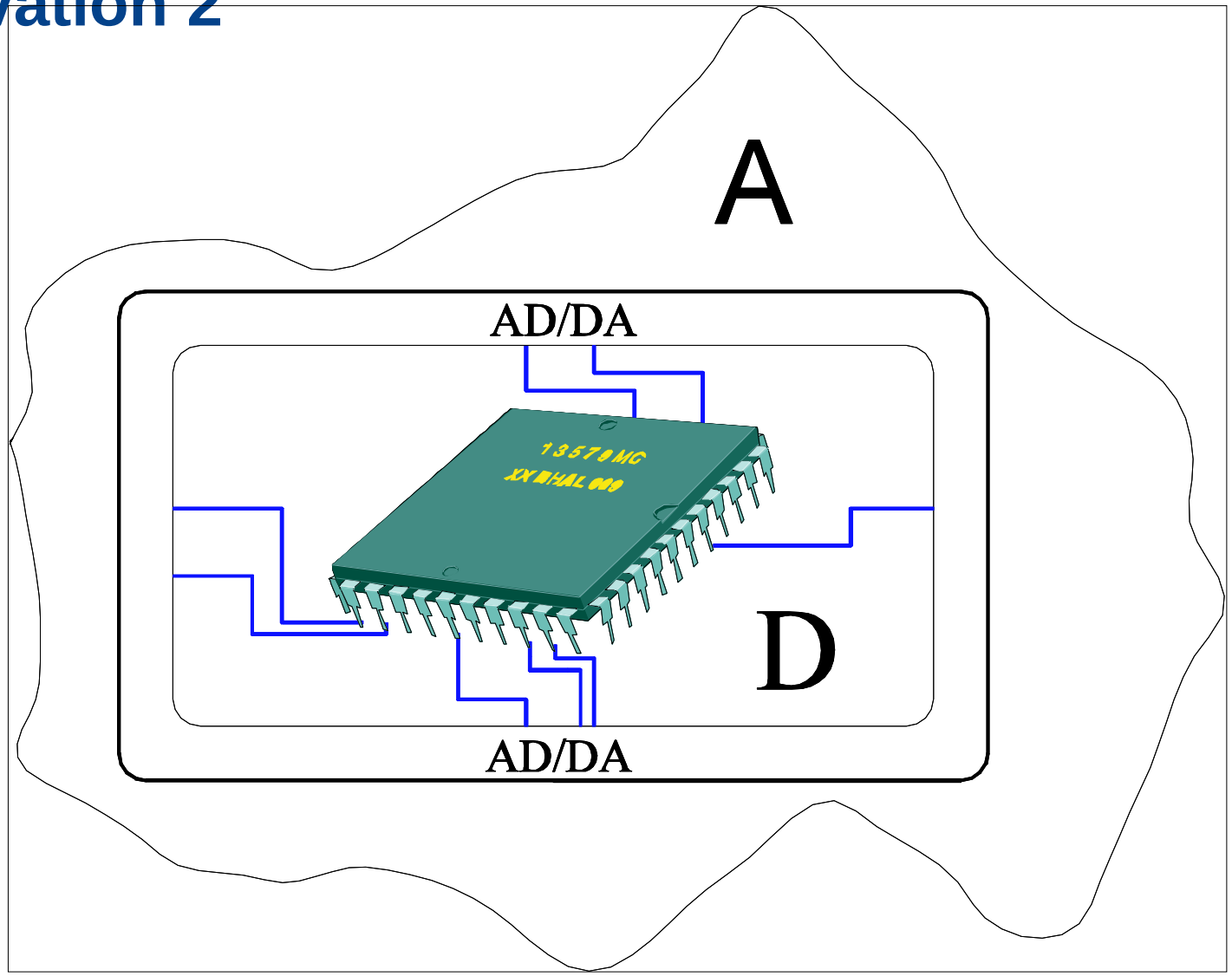
Folien zur Vorlesung Hardwarearchitekturen und Rechensysteme von
Prof. Dr. rer. Nat. U. Brinkschulte
Prof. Dr.-Ing. L. Hedrich

Motivation

- Kernschaltung der modernen digitalen und analogen Schaltungstechnik
- Zunehmende Bedeutung als Maß der Bedeutung des technologischen Fortschritts in der VLSI Technologie
- Technische Komponente der modernen Informationsgesellschaft



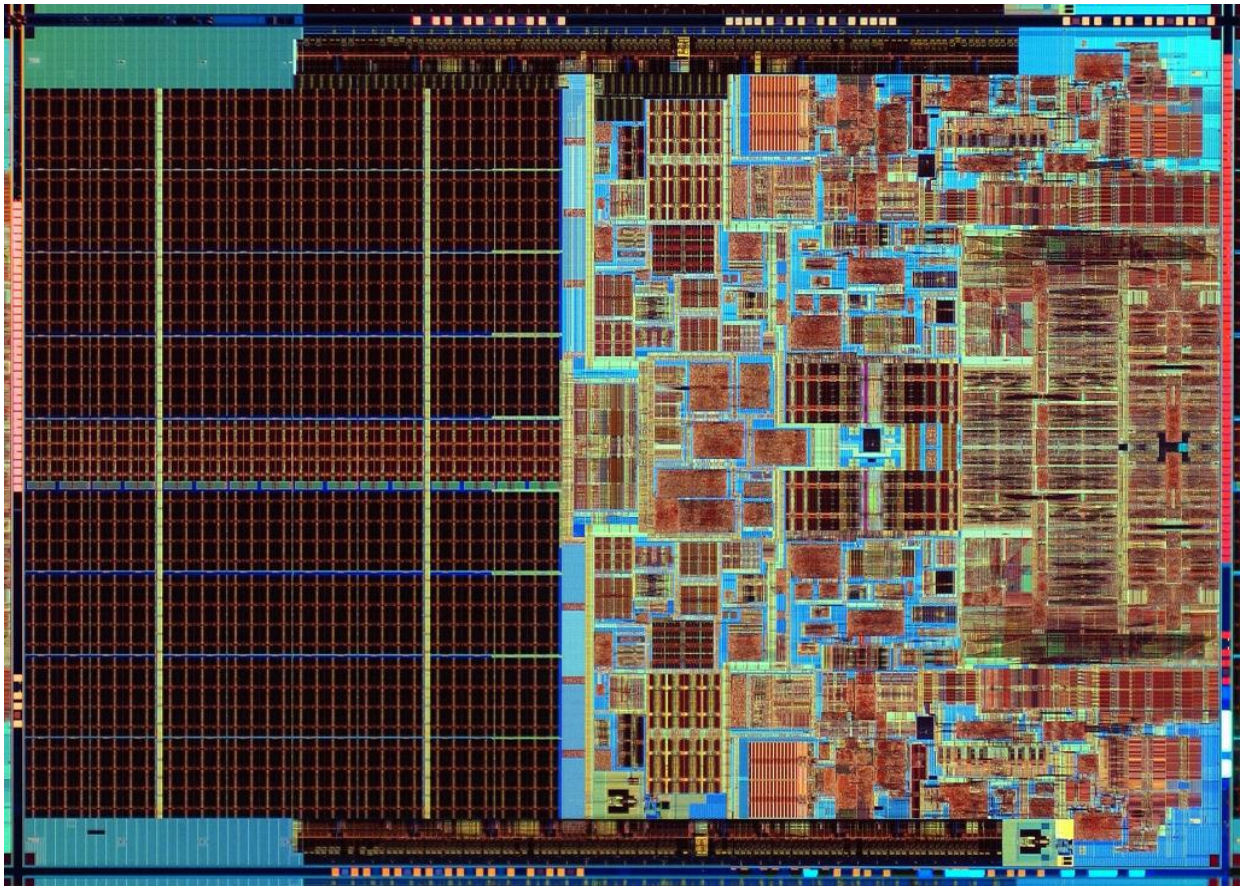
Motivation 2



Heutiger Laptop

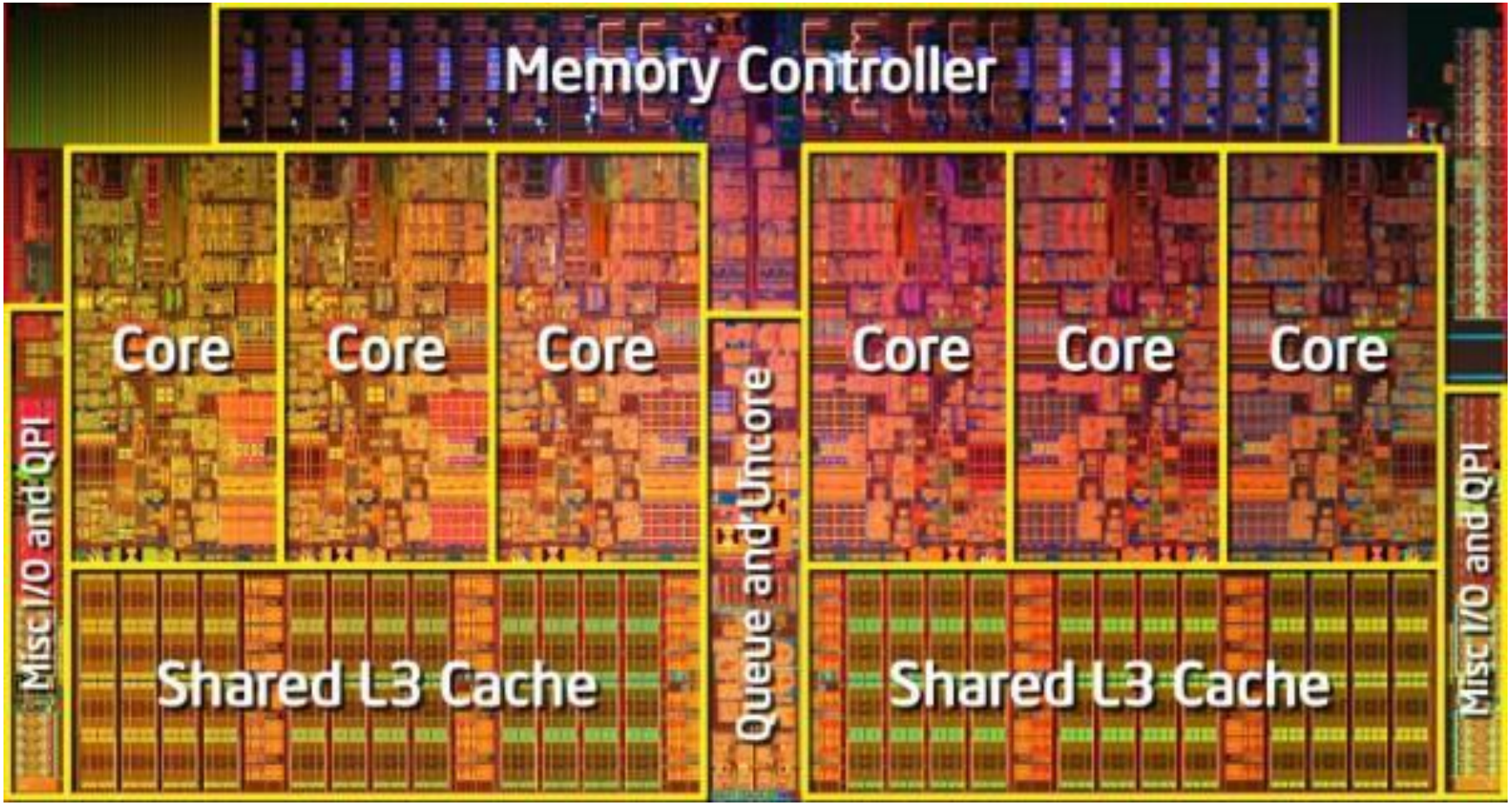
S3 Yoga **[www.laptopultra.com]**

Ein Blick in den Core 2 Duo (291 Mill. Transistoren)



Quelle: Intel

Heute: 6 Kerne möglich



Mikrofotografie eines Core i7-980 6-Kern Prozessors

1.17 Millarden Transistoren - 32nm - 248mm²

Gliederung

2.1 Zahlendarstellung

Menschen arbeiten mit verschiedensten Zahlen. Für den Rechner sind diese effizient zu kodieren. Zum einen sollte

- Speicherplatz

gespart werden. Zum anderen sollte möglichst mit diesen Zahlen eine hohe

- Rechenleistung

erbracht werden. Beides führt zu einer Vielzahl von möglichen Kodierungen. Mit bestimmten kann der Rechner direkt umgehen. Dies sind die Maschinen-Datentypen. Andere sind durch Software zu behandeln.

Systeme zur Informationsdarstellung

- Systeme zur Informationsdarstellung unterscheiden sich im wesentlichen durch die Anzahl der Zustände, welche die kleinstmögliche Informations-Einheit annehmen kann
- Anzahl der Zustände = **Wertigkeit** oder **Entscheidungsgrad** des Systems
- Beipiel: Die kleinste Informationseinheit (Stelle) unseres Zahlensystems kann 10 verschiedene Zustände annehmen
- => es besitzt die Wertigkeit 10
- Systeme der Wertigkeit 10 heißen **Denärsysteme**
- Die kleinstmögliche Wertigkeit eines solchen Systems ist 2
- Systeme der Wertigkeit 2 heißen **Binärsysteme**

Wertigkeit 2 ist technisch am leichtesten zu realisieren

=> Im Rechner benutztes System

Maschinen-Datentypen

Ein Datentyp wird durch eine Wertemenge und Operationen auf den Werten beschrieben. Bei Maschinen-Datentypen sind die Operationen als Maschinenbefehle implementiert. Typische Datentypen sind z.B.:

Bit:

Wertemenge: 0,1

Operationen: AND, OR, XOR, NOT, Vergleich, . . .

Byte/Character:

Wertemenge: Bitmuster (8 Bit) z.B. 00101101

Operationen: Identitätsvergleich, Shift, bitweises XOR, . . .

Integer:

Wertemenge: Ganze Zahlen (endliche Menge (z.B. 16 Bit, 64 Bit))

Operationen: ADD, SUB, MUL, DIV, Vergleich, . . .

Gleitkommazahlen:

Wertemenge: Gleitkommazahlen (Vorzeichen, Mantisse und Exponent:)

Operationen: ADD, SUB, MUL, DIV

ASCII-Codierung

	000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	0	@	P	'	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	´	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	¯
1111	SI	US	/	?	O	_	o	DEL

Binärzahlen

- N-stellige Binärzahlen
- Einzelne Stellen heißen Bits
- Byte:
  - Ein Byte: 8 Bit. Z.B. : 01001101

(MSB, Most Significant Bit)

(LSB, Least Significant Bit)

  - Wertigkeit: 2^{i-1} mit i der i-ten Stelle von rechts gezählt
  - Character $\triangleq$ Byte: kodierte Darstellung alphanumerischer Zeichen z.B. nach dem ASCII-Zeichensatz

- Integer:
  - N-stellige Binärzahl mit häufig 8, 16, 32 oder 64 Bits
  - mit Vorzeichen (signed) oder ohne Vorzeichen (unsigned)

Hexadezimalkodierung

- Kurze Darstellung einer 4-Bit-Integerzahl
- 2 Hexadezimalkodes können effizient ein Byte darstellen
  - Beispiel: 7D = 01111101 = 7*16+13 = 125
- Hexadezimalzahlen werden häufig durch ein vorgestelltes 0x dargestellt, z.B. 0x7D.

Binär	Dezimal	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Umwandlung von Zahlensystemen

- Umwandlung einer Zahl beliebiger Basis b in eine Dezimalzahl nach der Stellenwertformel:

$$X_b = z_n b^n + z_{n-1} b^{n-1} + \ldots + z_1 b^1 + z_0 + z_{-1} b^{-1} + \ldots + z_{-m} b^{-m}$$

- Umwandlung einer Dezimalzahl in eine Zahl beliebiger Basis b:
  - Euklidischer Alogrithmus
  - Horner-Schema

Umwandlung von Zahlensystemen: Euklid

Euklidischer Algorithmus:

$$Z = z_n 10^n + z_{n-1} 10^{n-1} + \ldots + z_1 10^1 + z_0 + z_{-1} 10^{-1} + \ldots + z_{-m} 10^{-m}$$
$$= y_p b^p + y_{p-1} b^{p-1} + \ldots + y_1 b^1 + y_0 + y_{-1} b^{-1} + \ldots + y_{-q} b^{-q}$$

Ziffern werden sukzessive, beginnend mit der höchstwertigsten Ziffer, berechnet:

1. Schritt: Berechne p gemäß der Ungleichung
   $b^p \leq Z < b^{p+1}$
2. Schritt: Ermittle y_i und den Rest R durch Division von Z durch b^i
   $y_i = Z \text{ div } b^i; \quad R = Z \text{ mod } b^i;$
   $y_i \in \{0, 1, \ldots, b-1\}$
3. Schritt: Wiederhole 2. Schritt für i = p, p-1, … , ersetze dabei nach jedem Schritt Z durch R, bis R= 0 oder bis b^i (und damit der Umrechnungsfehler) gering genug ist.

Umwandlung von Zahlensystemen: Beispiel

Beispiel: Umwandlung von 15741,23310 ins Hexadezimalsystem

Schritt : $16^3 \leq 15741{,}233 < 16^4$ => höchste Potenz 16^3
Schritt : $15741{,}233 : 16^3 = 3$ Rest 3453,233
Schritt : $3453{,}233 : 16^2 = D$ Rest 125,233
Schritt : $125{,}233 : 16 = 7$ Rest 13,233
Schritt : $13{,}233 : 1 = D$ Rest 0,233
Schritt : $0{,}233 : 16^{-1} = 3$ Rest 0,0455
Schritt : $0{,}0455 : 16^{-2} = B$ Rest 0,00253
Schritt : $0{,}00253 : 16^{-3} = A$ Rest 0,000088593
Schritt : $0{,}000088593 : 16^{-4} = 5$ Rest 0,000012299 (=> Fehler)

=> **$15741{,}23310 \approx 3D7D{,}3BA5_{16}$**

Umwandlung von Zahlensystemen: Horner

Horner-Schema

→Getrennte Betrachtung des ganzzahligen und des gebrochenen Anteils

a. Umwandlung des ganzzahligen Anteils:

Schreibe ganze Zahl $X_b = \sum_{i=0}^{n} z_i b^i$

durch fortgesetztes Ausklammern in der Form

$X_b = ((\ldots(((y_n b + y_{n-1}) b + y_{n-2}) b + y_{n-3}) b \ldots) b + y_1) b + y_0$

Umwandlung von Zahlensystemen: Beispiel II

Verfahren:

- Die gegebene Dezimalzahl wird sukzessive durch die Basis b dividiert
- Die jeweiligen ganzzahligen Reste ergeben die Ziffern der Zahl X_b in der Reihenfolge von der niedrigstwertigen zur höchstwertigen Stelle

Beispiel:

Wandle 15741_{10} ins Hexadezimalsystem

15741_{10} : 16	=	983	Rest 13	(D)
983_{10} : 16	=	61	Rest 7	(7)
61_{10} : 16	=	3	Rest 13	(D)
3_{10} : 16	=	0	Rest 3	(3)

=> $15741_{10} = 3D7D_{16}$

Umwandlung von Zahlensystemen: Nachkomma

Umwandlung des Nachkommateils gebrochener Zahlen:

Auch der gebrochene Anteil $\sum_{i=-m}^{-1} z_i b^i$ einer Zahl lässt sich entsprechend schreiben

$$Y_b = ((\ldots((y_{-m} b^{-1} + y_{-m+1}) b^{-1} + y_{-m+2}) b^{-1} + \ldots + y_{-2}) b^{-1} + y_{-1}) b^{-1}$$

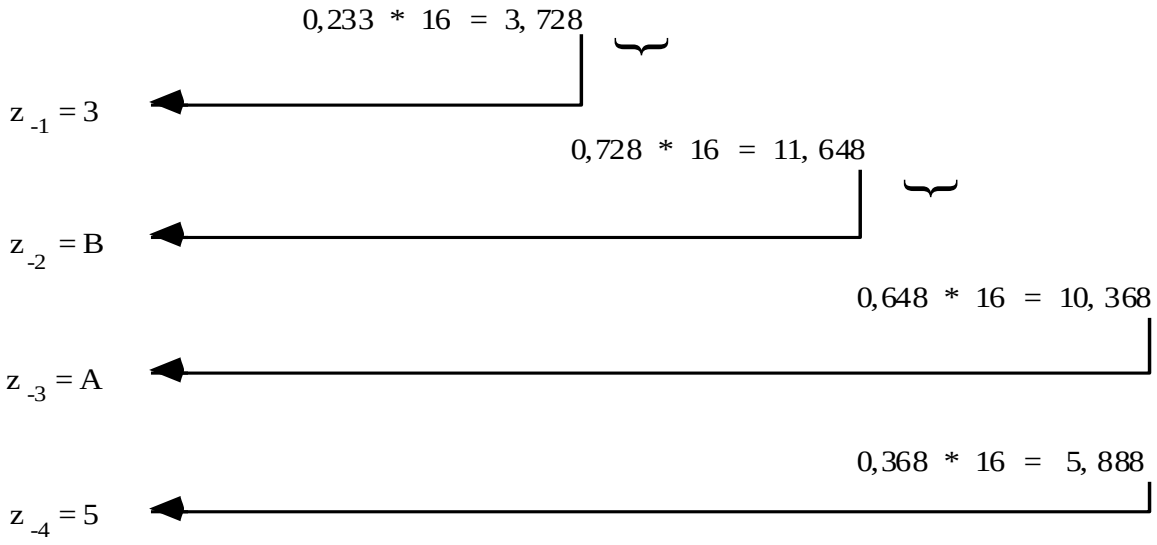
Verfahren:

- Sukzessive Multiplikation des Nachkommateils der Dezimalzahl mit der Basis b des Zielsystems
- Die jeweiligen ganzzahligen Anteile ergeben nacheinander die y_{-i} in der Reihenfolge der höchstwertigsten zur niederwertigsten Nachkommaziffer

Umwandlung von Zahlensystemen: Beispiel III

Beispiel:

Umwandlung von $0{,}233_{10}$ ins Hexadezimalsystem



Abbruch bei genügend hoher Genauigkeit

=> $0{,}233_{10} \approx 0{,}3BA5_{16}$

Komplementdarstellung

Für negative Zahlen werden Komplementdarstellung gewählt. Sie ermöglichen eine einfache Implementierung der Subtraktion und Addition mit negativen Zahlen.

Es gilt für N-stellige Binärzahlen: $a - b = a - b + 2^N = a + 1 + 2^N - 1 - b$

- 1er-Komplement
  - Darstellung der Zahl - b als 2^N -1 –b durch einfaches Invertieren jeder Stelle (aus jeder 0 wird eine 1 und aus jeder 1 eine 0)
  - Die Subtraktion kann durch die Berechnung von a + 1er(b) +1 erfolgen
- 2er-Komplement
  - Darstellung der Zahl - b als 2^N –b durch einfaches Invertieren jeder Stelle und anschließendes Addieren von 1
  - Die Subtraktion kann durch die Berechnung von a+2er(b) erfolgen
  - Konsistentes rechnen mit dieser Darstellung ist auch für negative Zahlen möglich. → Darstellung in **C, C++ ...**

Gleitkommazahlen

In digitalen Computern wird Arithmetik mit endlicher Genauigkeit vollzogen. Die endliche Genauigkeit impliziert, dass Zahlen, die die darstellbare Genauigkeit überschreiten, gerundet werden müssen. Bei der Gleitkommadarstellung wird eine Zahl Z so gespeichert, dass das Komma immer zur ersten von 0 verschiedenen Zahl gleitet. Dies erreicht man durch Abspalten einer entsprechenden Potenz:

$$Z = m * b^e \qquad \text{mit} \qquad m = 0, \text{xxx} \ldots$$

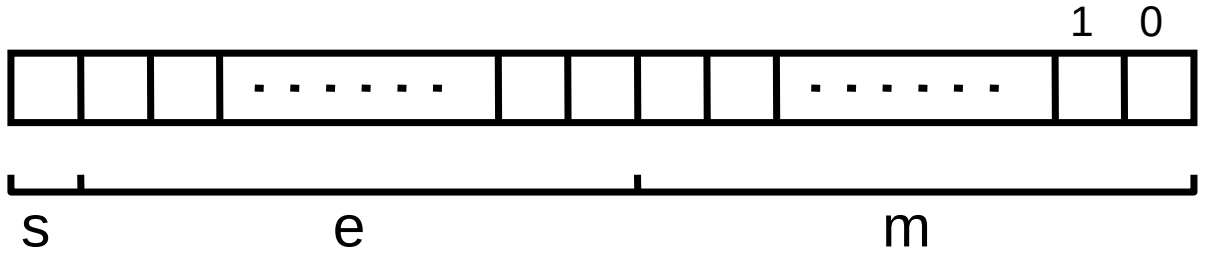
Da die Basis b bekannt ist, kann die Zahl durch die Mantisse m und den Exponenten e, mit jeweils fester Stellenanzahl, dargestellt werden. Die Anpassung der Gleitkomma-Zahl an diese Darstellungsform wird *normalisieren* genannt.

Beispiel:

$$Z = 143,135 = 0,143135 * 10^3 \qquad \Rightarrow \qquad m = 143135$$
$$e = 3$$

IEEE-754

Das *Institute of Electrical and Electronics Engineers (IEEE)* hat einen Standard für Gleitkomma-Zahlen entwickelt, der heutzutage in fast allen Computern Verwendung findet. Gleitkomma-Zahlen werden durch ein Vorzeichen-Bit, einen Exponenten und eine Mantisse, jeweils mit fester Stellenanzahl und zur Basis 2, dargestellt.



s: Vorzeichen

e: Exponent

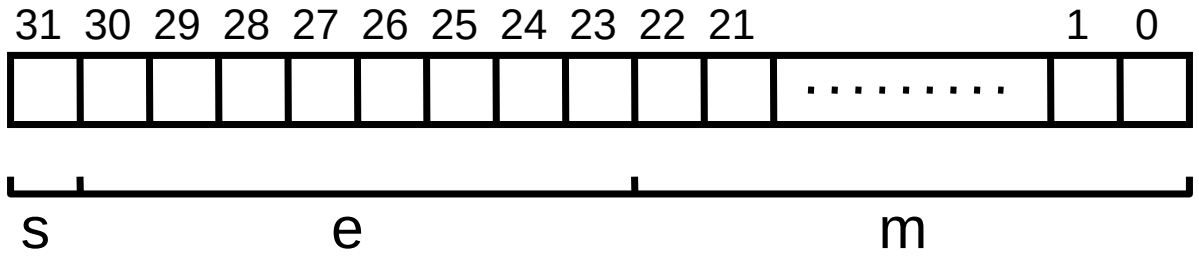
m: Mantisse

Im Standard IEEE-754 werden Gleitkomma-Zahlen mit verschiedenen Wertebereichen und Genauigkeiten definiert:

Format	Bit	Vorzeichen	Exponent	Mantisse
single	32	1Bit	8Bit	23Bit
double	64	1Bit	11Bit	52Bit

IEEE-754 – single

Eine 32Bit Gleitkomma-Zahl besteht aus einem Vorzeichen-Bit, einem 8Bit Exponenten und einer 23Bit Mantisse.



Exponent:

Der 8Bit Exponent e kann Werte aus dem Wertebereich $0, \ldots, 255$ darstellen. Durch Subtraktion des konstanten Wertes 127 wird eine Links- und Rechtsgleitung des Kommas im Wertebereich $-127, \ldots, 128$ ermöglicht.

Mantisse:

Die 23Bit Mantisse m stellt einen 24Bit-Wert (zzgl. Vorzeichen-Bit) dar, indem das Komma über die erste von 0 verschiedene Stelle hinaus gleitet $(1,m)$.

IEEE-754 – single

s	e	m	Wert
0/1	$0 < e < 255$	beliebig	$Z = (-1)^s * 2^{e-127} * 1, m$
0/1	0	0	$Z = (-1)^s * 0,0$ (signed zero)
0/1		$m \neq 0$	$Z = (-1)^s * 2^{-126} * 0, m$
0	255	0	$Z = +\infty$ (positive infinity)
1		0	$Z = -\infty$ (negative infinity)
0/1		$m \neq 0$	$Z = NaN$ (not a Number)

Beispiele -- IEEE-754 – single

s	e	m	Dezimal-Wert
0	0111 1111	000 0000 0000 0000 0000 0000	$1,0$
0	1000 0000	000 0000 0000 0000 0000 0000	$2,0$
0	1111 1110	111 1111 1111 1111 1111 1111	$3,40282347E+38$ (max)
0	0000 0001	000 0000 0000 0000 0000 0000	$1,17549435E-38$ (min)
0	0000 0000	000 0000 0000 0000 0000 0000	$+0,0$
1	0000 0000	000 0000 0000 0000 0000 0000	$-0,0$
0	1111 1111	000 0000 0000 0000 0000 0000	$+\infty$
1	1111 1111	000 0000 0000 0000 0000 0000	$-\infty$
0	1111 1111	100 0000 0000 0000 0000 0000	NaN
0	1000 0000	100 1001 0000 1111 1101 1010	π

Beispiele -- IEEE-754 – single

Wie stellt man z.B. 3,8125 in der IEEE-754 Darstellung dar?

- Vorzeichenbit ist 0, da positiv
- Vorkommateil 3 ist binär dargestellt 11
- Nachkommateil 0,8125 ist $2^{-1} + 2^{-2} + 2^{-4}$, also binär → 1101
- Zusammensetzen des Vor- und Nachkommateils: 11,1101
- Verschieben des Kommas hinter die erste 1 von links: 1,11101 → Exponent ist 1, da um 1 Stelle verschoben.
- Aufaddieren von 127 auf Exponent (= 128) und binär darstellen als 8-Bit-Wert: 10000000
- Auffüllen oder kürzen der Mantisse auf insgesamt 23 Bit, wobei die 1 vor dem Komma nicht codiert wird: 11101000000000000000000
- Zusammenfügen von 3,8125 als IEEE-754 - single:
  (Vorzeichen - Exponent - Mantisse) →
  0-10000000-11101000000000000000000

Gleitkommaarithmetik

- **Gegeben:**

$$A = (e_A, m_A) \qquad \text{und} \qquad B = (e_B, m_B)$$

- **Zur Vereinfachung sei**

$$e_A \leq e_B$$

- Addition

$$C = A + B = (m_A * 2^{e_A - e_B} + m_B) * 2^{e_B}$$

- Subtraktion

$$C = A - B = (m_A * 2^{e_A - e_B} - m_B) * 2^{e_B}$$

- Multiplikation

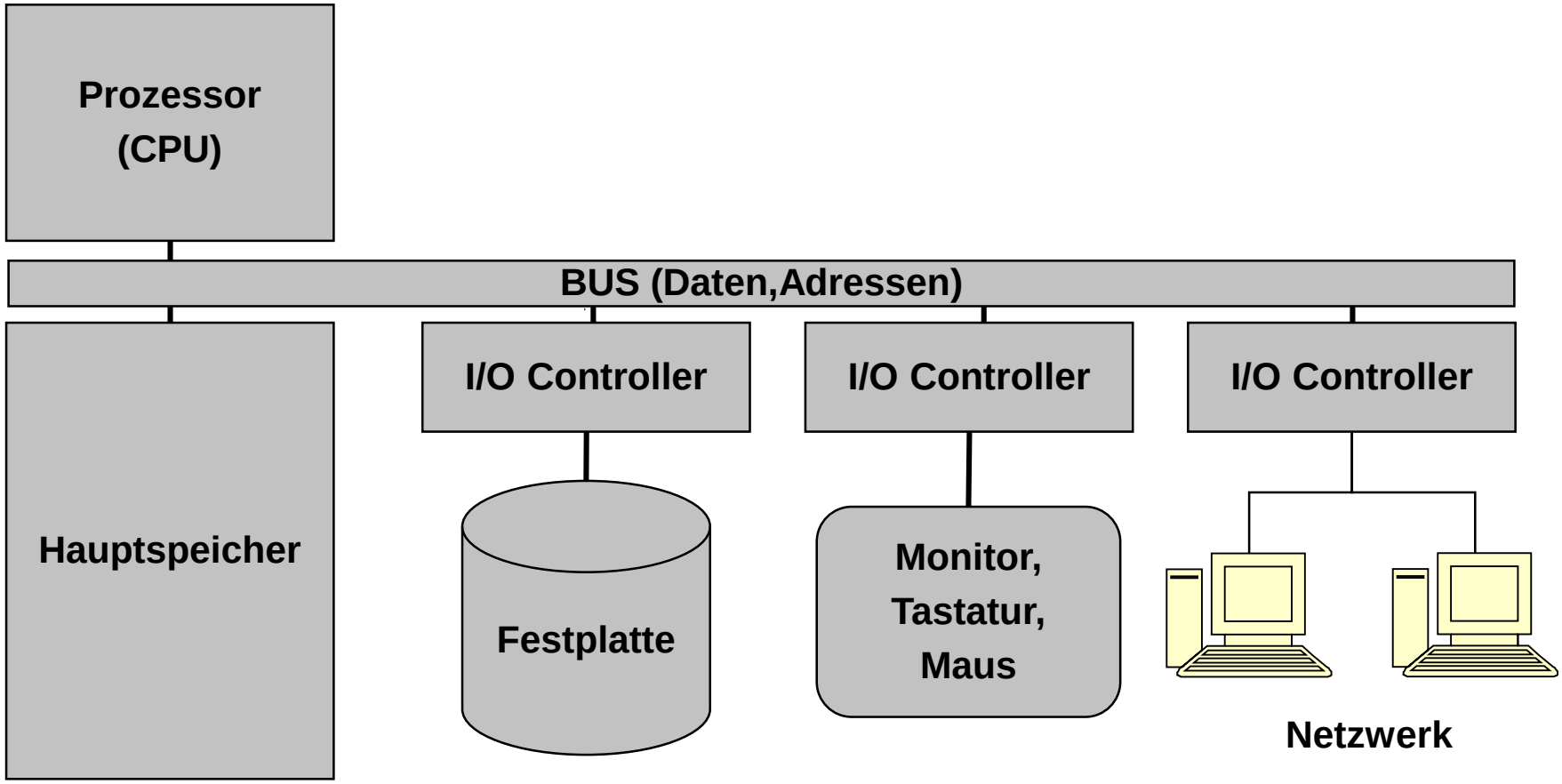
$$C = A * B = (m_A * m_B) * 2^{e_A + e_B}$$

- Division

$$C = A/B = (m_A / m_B) * 2^{e_A - e_B}$$

2.2 Rechnerarchitekturen

Ein Rechner besteht aus folgenden Komponenten



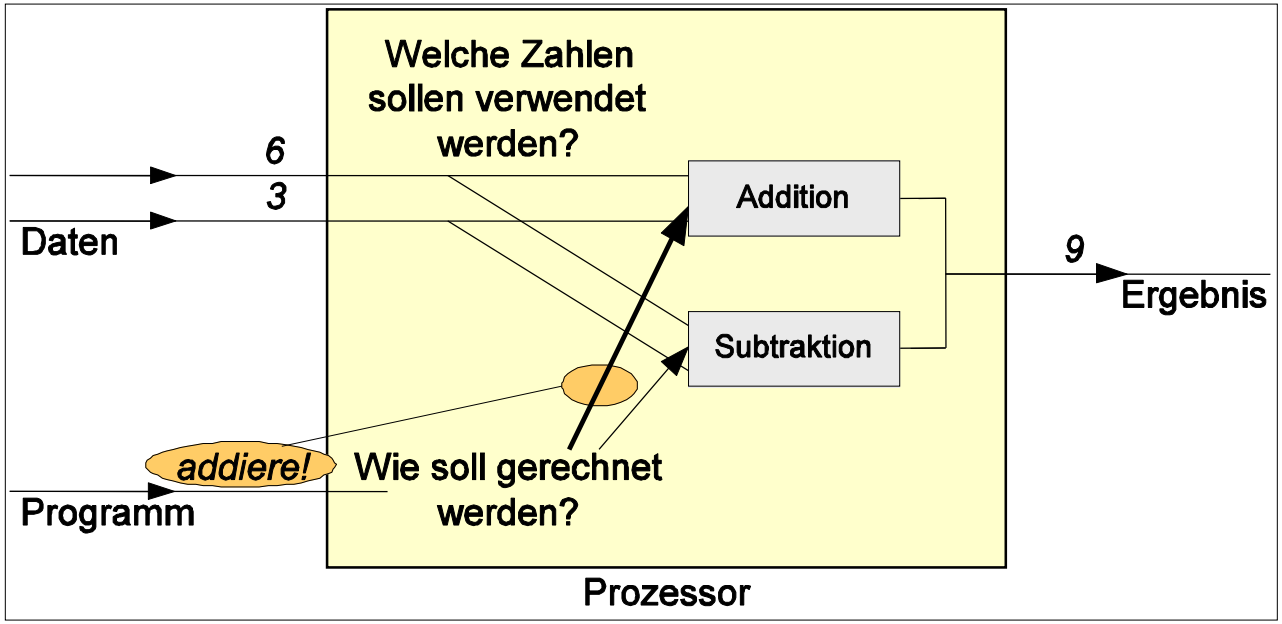
Wo ist das hier?

S3 Yoga **[www.laptopultra.com]**

Prozessor

Ein Prozessor arbeitet ein Programm ab, das aus einzelnen Befehlen besteht. Die Befehle erhalten Daten zur Berechnung und erzeugen Ausgaben.

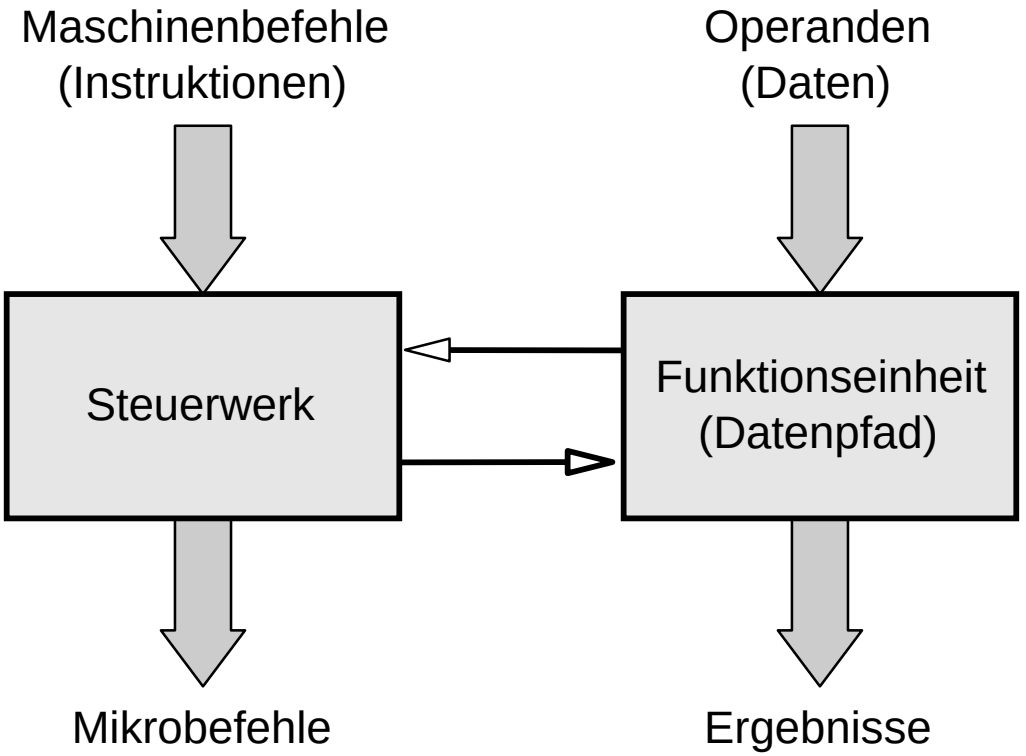
Daten liegen am Dateneingang an. Das Programm wird befehlsweise am Steuereingang angelegt.



Typische Datenquellen sind: Benutzereingaben, Festplatten, Sensordaten...

Rechner

Operationsprinzip eines Rechners ist die sequentielle Programmabarbeitung durch einen Prozessor. Diesem Prinzip liegt die zentrale Steuerung durch ein Steuerwerk und die zentrale Verarbeitung durch ein oder mehrere Rechenwerke zugrunde.



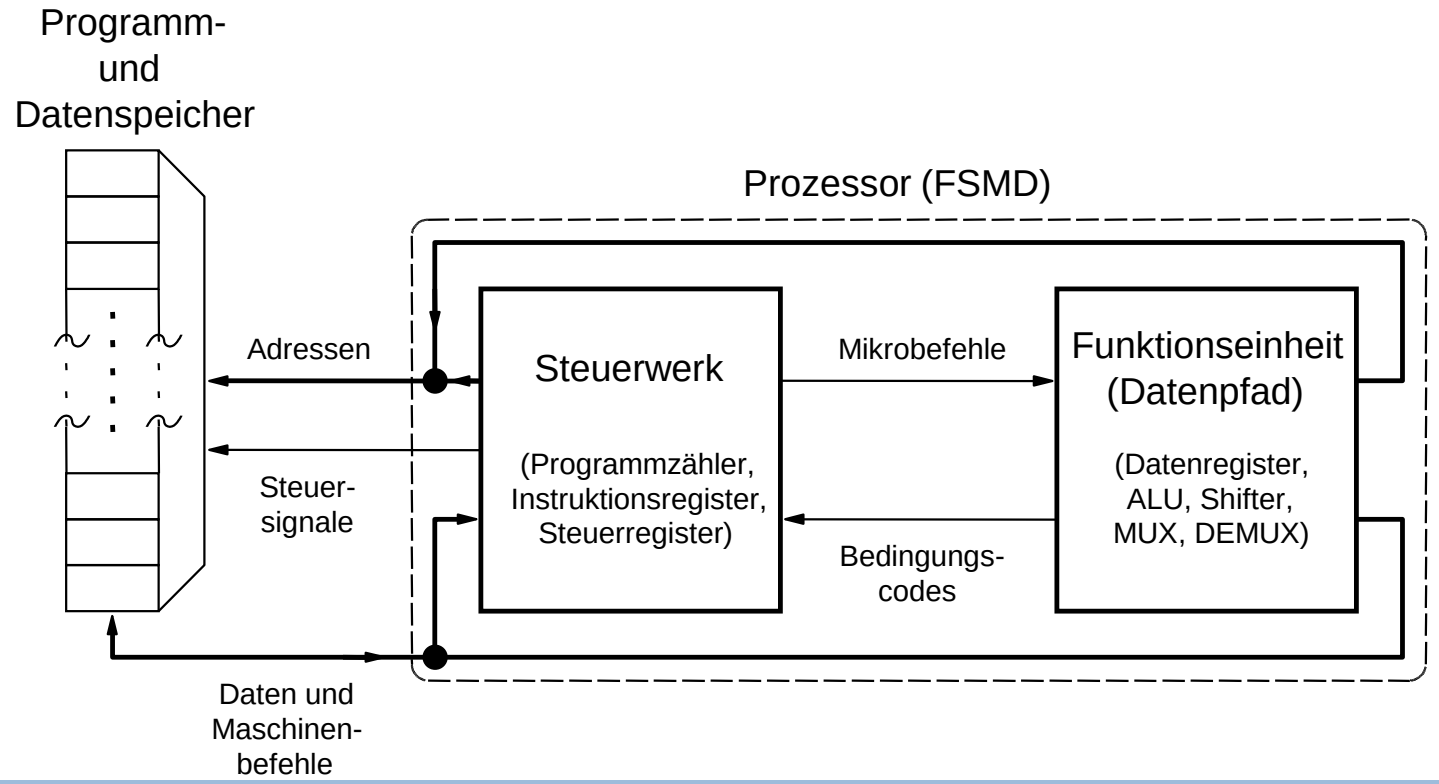
Rechner

In einem Befehlszyklus wird i.d.R. ein Befehl aus dem Hauptspeicher geholt und dann ausgeführt. Ein Befehlszyklus, d.h. die Folge von Aktionen, bis der nächste Befehl aus dem Hauptspeicher geholt wird, besteht im einfachsten Fall aus den Teilzyklen:

- Befehl holen (instruction fetch)
- Befehl dekodieren (instruction decode)
- Befehl ausführen (instruction execute)

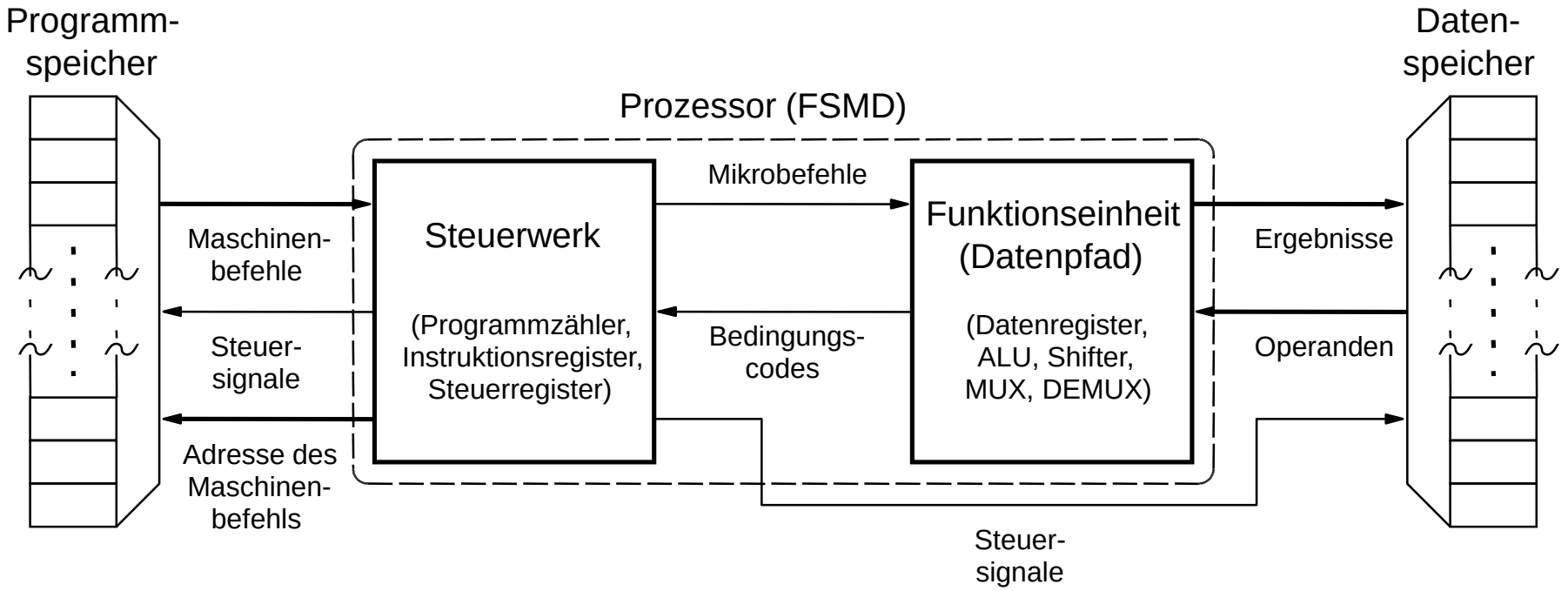
2.2.1 Die von Neumann-Architektur

Die von Neumann-Architektur zeichnet sich durch eine sehr einfache Struktur mit einem gemeinsamen Speicher für Daten und Programme aus. Das Programm und die Daten müssen nacheinander aus dem Speicher gelesen werden.



2.2.2 Die Harvard-Architektur

Bei der Harvard-Architektur handelt es sich um eine kompliziertere Struktur mit getrenntem Speicher für Daten und Programme. Das Programm und die Daten können gleichzeitig aus dem Speicher gelesen werden.



Heutige Rechner entsprechen überwiegend der Harvard-Architektur.

2.2.3 Busse

Sammelschiene

Verbindung mehrere Komponenten eines Rechners über dieselben Leitungen; dabei darf zu einem Zeitpunkt immer nur eine Komponente Informationen auf die Leitungen legen

Adreßbus, Datenbus, Steuerbus

Als Gesamtheit: **Systembus**

Die Informationen werden binär codiert über den Bus übertragen

Bus

Adreßleitungen

Diejenigen Leitungen, auf denen die Adreß-information transportiert wird (unidirektional).

Datenleitungen

Transportieren Daten und Befehle von/zum Prozessor (bidirektional).

Steuerleitungen

Geben Steuerinformationen von/zum Prozessor (uni- oder bidirektional).

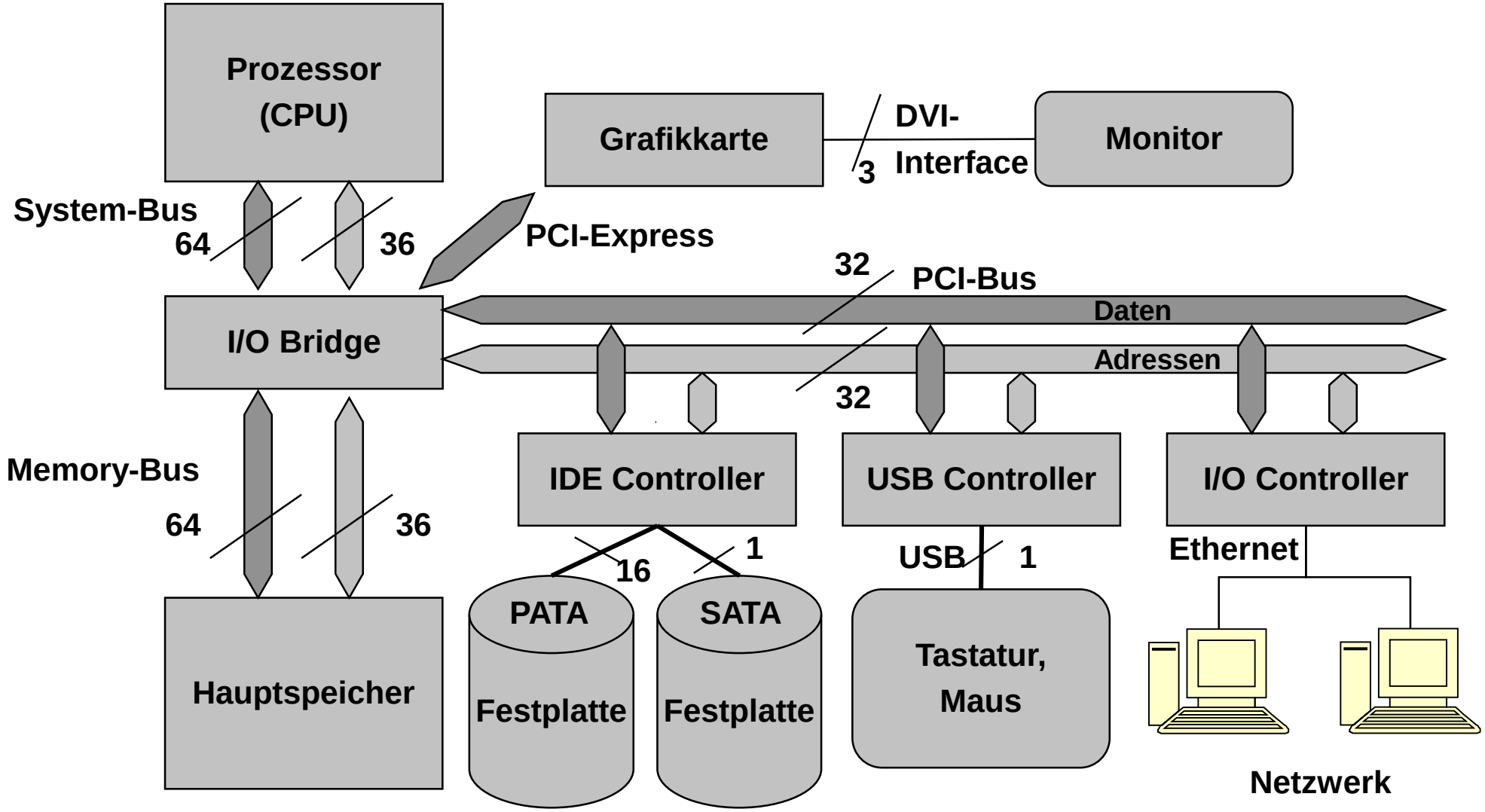
Busse

- In modernen Rechneranlagen sind viele verschiedene Bussysteme vorhanden
- Beispiele
  - PCI
  - SCSI , IDE, S-ATA
  - Systembus
    (Daten und Adressen zum Hauptspeicher)
  - Memory-Bus
  - On-Chip-Busse
    - Amba-Bus
  - USB
  - Trend: von parallelen Bussen mit niedriger Taktfrequenz zu seriellen Punkt-zu-Punkt-Verbindungen mit hoher Taktfrequenz

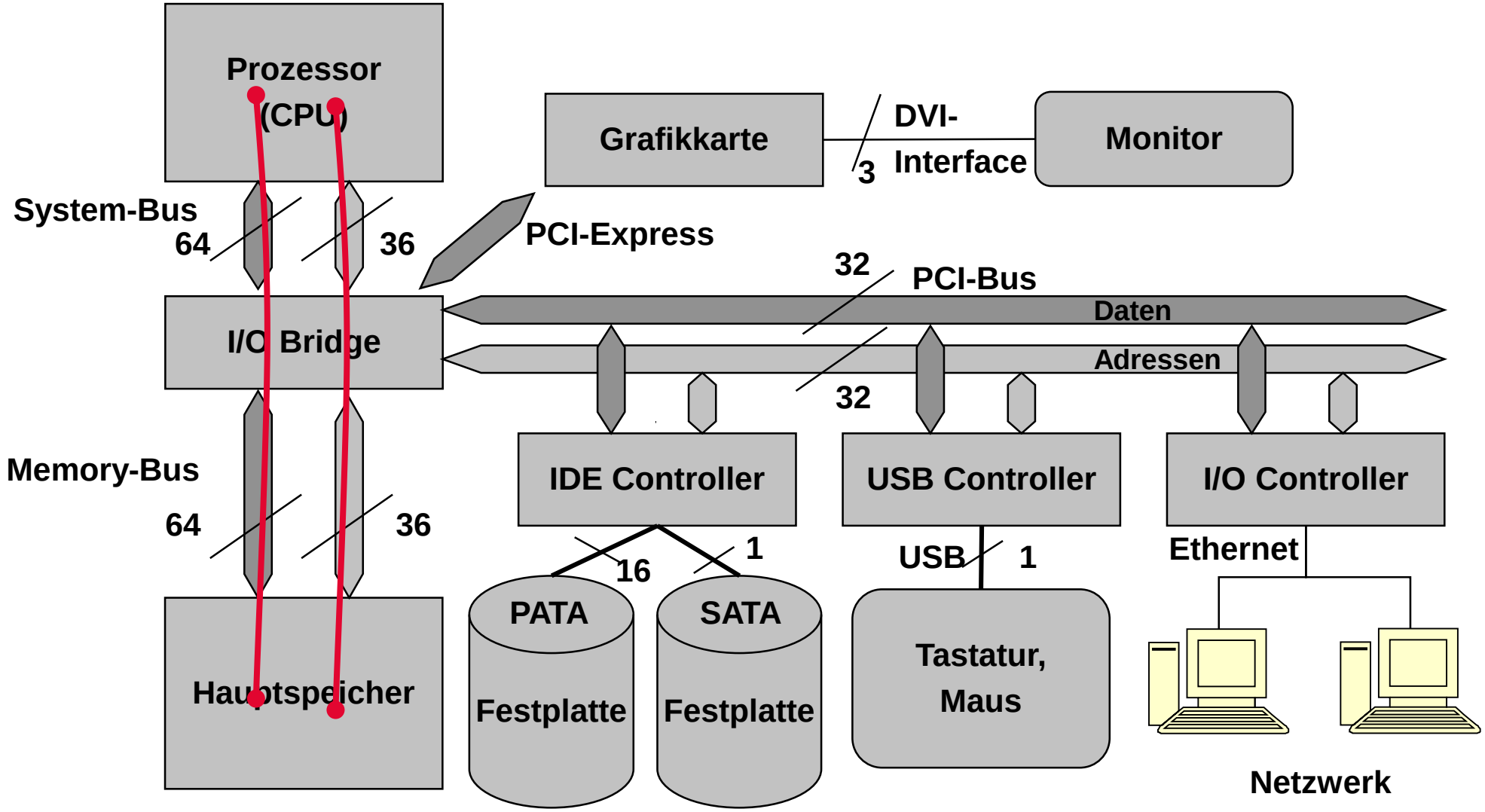
Busse und PTP-Verbindungen: Bandbreiten

Bus	Übertragungsart	Taktrate	Übertragungsrate
Front Side Bus	64 Bit parallel, 4-fache Datenrate	266 MHz	8,5 GByte/s
Memory Bus	64 Bit parallel, 2-fache Datenrate	533 MHz	8,5 GByte/s
PCI Express x16	16 x 1 Bit seriell, vollduplex, 2-fache Datenrate, 8B10B Kode	1250 MHz	8 GByte/s
PCI Express x1	1 x 1 Bit seriell, vollduplex, 2-fache Datenrate, 8B10B Kode	1250 MHz	500 MByte/s
DMI x4	4 x 1 Bit seriell, vollduplex, 2-fache Datenrate, 8B10B Kode	1250 MHz	2 GByte/s
PCI Bus	32 Bit parallel, Adress/Daten-Multiplex	33 MHz	133 MByte/s
USB	1 Bit seriell, halbduplex, 1-fache Datenrate, NRZI Kode	480 MHz	60 MByte/s
Serial ATA	1 Bit seriell, halbduplex, 1-fache Datenrate, 8B10B Kode	1500 MHz	150 MByte/sec
IDE (UDMA/133, ATA/ATAPI-7)	16 Bit parallel, 1-fache Datenrate	66 MHz	133 MByte/sec

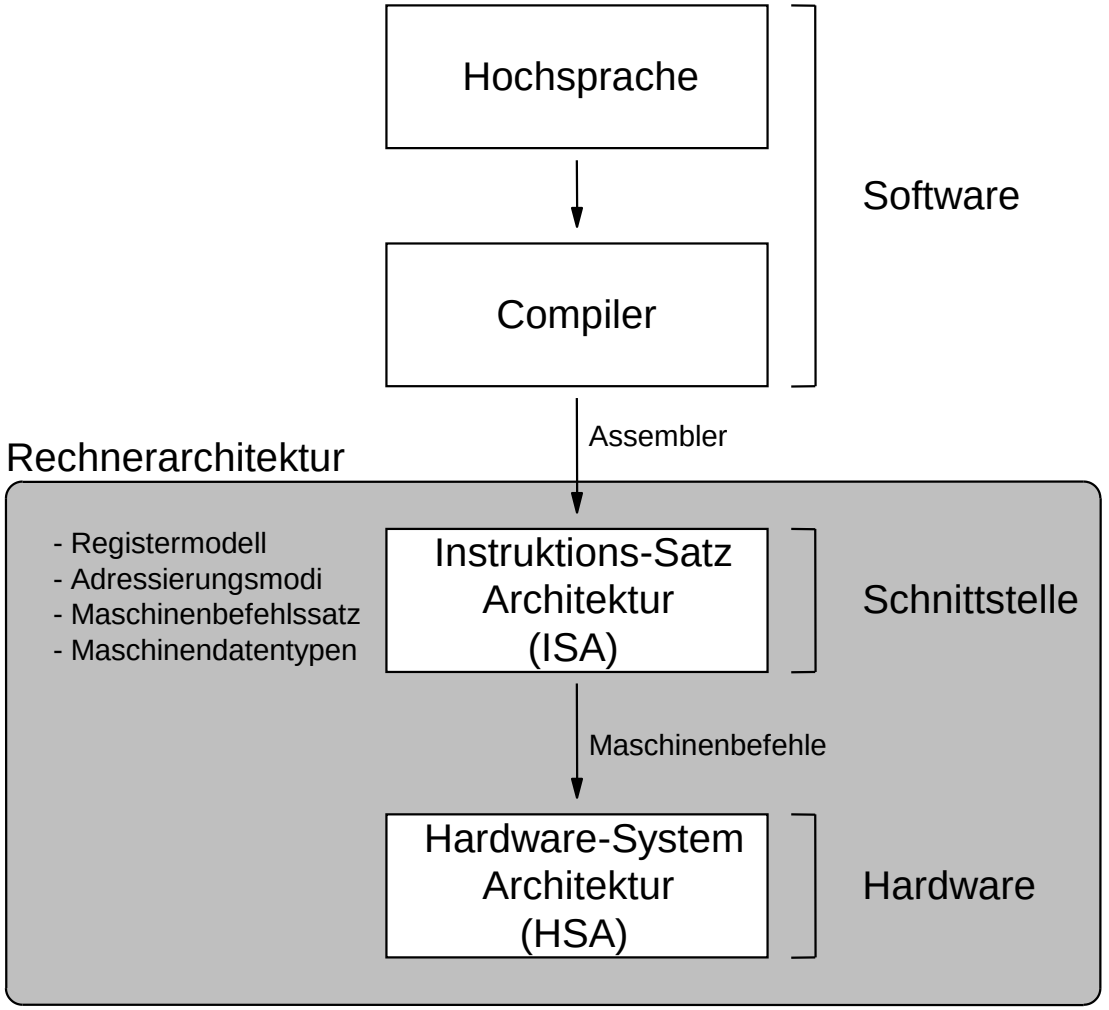
Busse im PC (Pentium 4 - System)



Zugriff auf den Hauptspeicher



2.3 HSA und ISA



HSA und ISA

Hardware-System-Architektur (HSA):

Die HSA bestimmt das Operations- und das Strukturkonzept der betrachteten Rechnerklasse und legt den Aufbau des Rechners aus einzelnen Hardwarekomponenten fest.

Auf Prozessorebene (Mikroarchitekturebene) sind z.B. die Organisation der Befehlsverarbeitung, der Aufbau des Rechen- und des Steuerwerks oder der Aufbau von Caches Gegenstand der HSA. Auf Systemebene (Makroarchitekturebene) zählt u.a. die Festlegung der Anzahl der Prozessoren und deren Verbindung dazu.

Instruktions-Satz-Architektur (ISA):

Die ISA definiert die Hardware-Software-Schnittstelle der Rechnerarchitektur und beschreibt den Rechner aus der Sicht des Assembler-Programmierers und des Compilerbauers. Somit hat sie einen maßgeblichen Einfluss auf die HSA.

Die wichtigsten Komponenten der ISA sind einerseits das Registermodell und die Adressierungsmöglichkeiten des Hauptspeichers und andererseits der Maschinenbefehlssatz und die verfügbaren Maschinendatentypen.

2.3.1 Die Hardware-System-Architektur

Die *Funktionseinheit* besteht aus einem oder mehreren Datenpfaden und führt die einzelnen Operationen aus. Typische Komponenten eines Datenpfades sind:

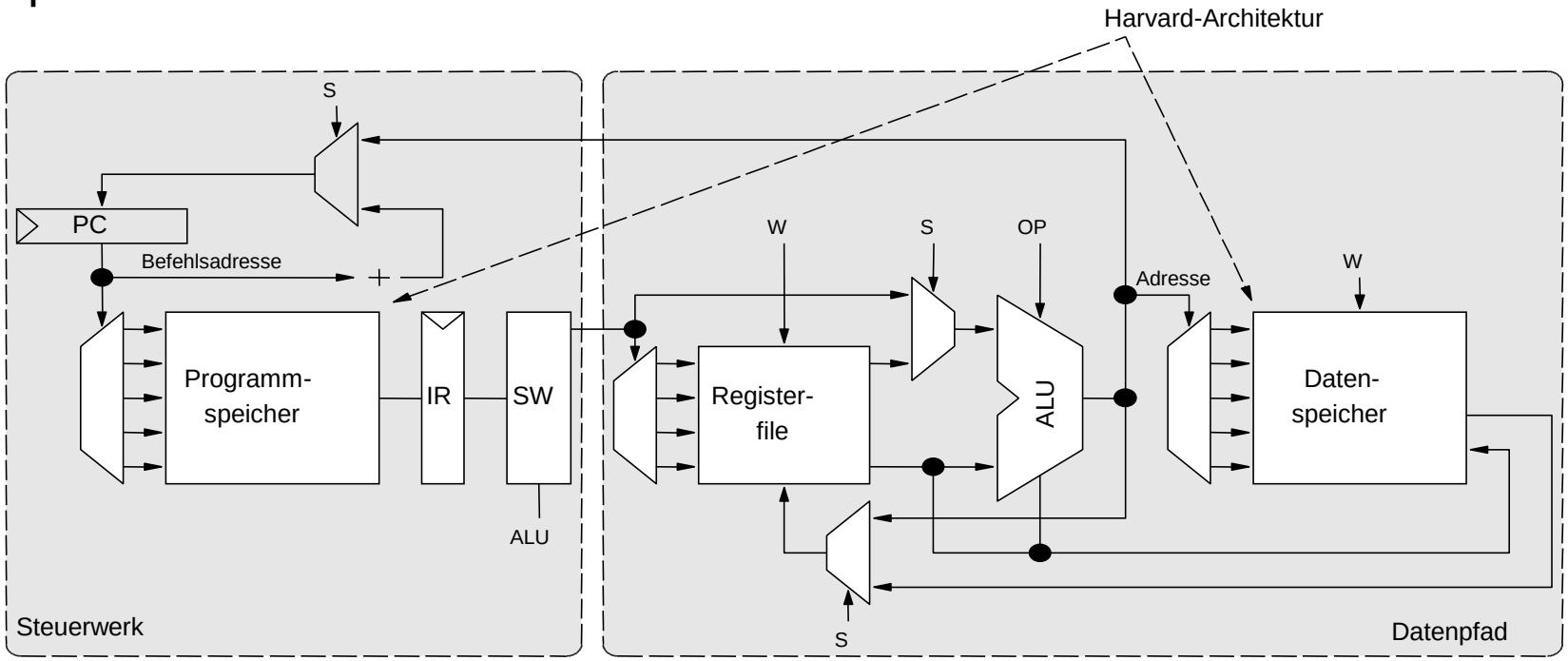
- Busse
- Register/Registerfiles
- ALU
- Shifter
- Multiplexer/Demultiplexer
- . . .

Das *Steuerwerk* steuert die Ausführung der durchzuführenden Berechnungen.

Bei einigen einfachen Rechnern besteht die Hardware-System-Architektur nur aus einem Steuerwerk (z.B. Fahrstuhlsteuerung) oder nur aus einer Funktionseinheit (z.B. Aufgaben der Signalverarbeitung).

Die Hardware-System-Architektur

Multiplexer-basierte 3-Adreß-Load-Store-Architektur:



ALU: Arithmetical Logical Unit

PC: Befehlszähler (Program Counter)

IR: Instruktionsregister (Instruction Register)

SW: Steuerwerk

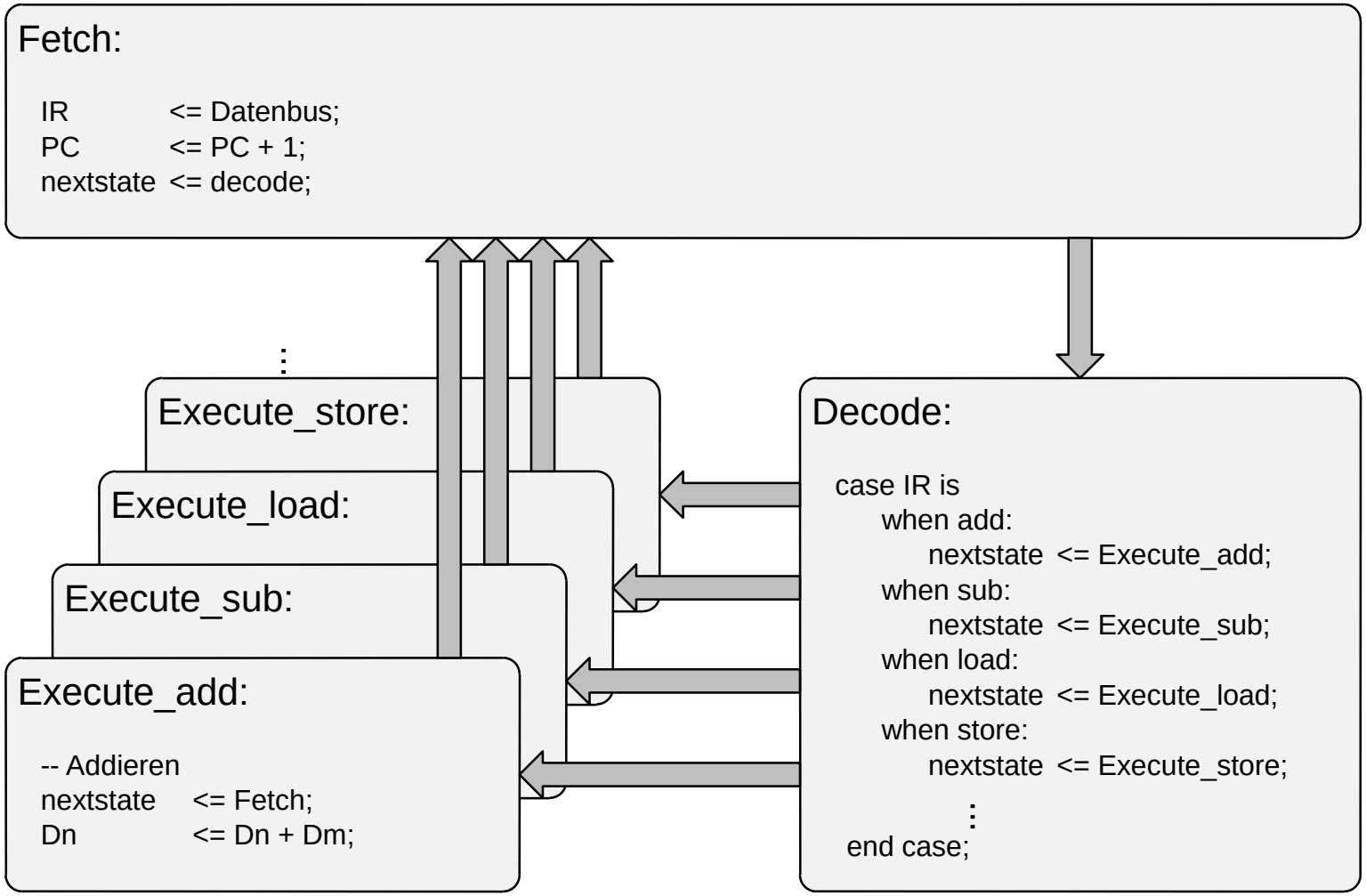
Die Hardware-System-Architektur

Multiplexer-basierte 3-Adreß-Load-Store-Architektur:

Im Fetch-Zyklus wird die Instruktion in das Instruktionsregister *IR* geladen und anschließend durch das Steuerwerk *SW* dekodiert. Aus dem dekodierten Befehl werden die Steuersignale für die ALU (*OP*) und die Multiplexer (*S*), die Schreibsignale (*W*) und gegebenenfalls die Registernummern für einen lesenden bzw. schreibenden Zugriff auf das Registerfile generiert. Die ALU verknüpft zwei Operanden, die aus dem Registerfile geladen werden oder Teil der Instruktion sind. Bei logisch/arithmetischen Operationen wird das Ergebnis in das Registerfile zurückgeschrieben. Bei Load- bzw. Store-Instruktionen berechnet die ALU aus den Operanden die effektive Adresse und führt eine Lese- bzw. Schreiboperation auf dem Registerfile aus.

Der Befehlszähler (*PC*) speichert die Adresse des nächsten auszuführenden Befehls und wird normalerweise automatisch inkrementiert.

Befehlszyklus bei busbasierter Architektur



2.3.2 Die Instruktions-Satz-Architektur

ISA: Registermodell

Prozessoren enthalten eine kleine Mengen an Registern (z.B. 32 Register mit jeweils 64Bit) für die kurzfristige Datenhaltung. Diese können mit wenigen Bits im Befehlscode adressiert werden. Im Vergleich zum Hauptspeicher sind diese Daten sehr schnell verfügbar.

Diese Gründe haben dazu geführt, dass Maschinenbefehle üblicherweise auf Registern und nicht auf Hauptspeicherzellen operieren. Sind alle Maschinenbefehle, bis auf Register-Lade- und -Speicherbefehle von dieser Art, so spricht man von einer *Load-Store-ISA*.

Adressierbare Register:

- Arbeitsregister
- Statusregister (*SR*)

nicht-Adressierbare Register:

- Instruktions- oder Befehlsregister (*IR*)
- Befehlszähler (*PC*)
- Stackpointer (SP)

ISA: Addressierungsarten

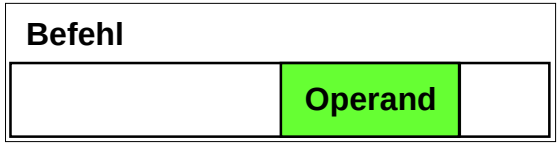
Die größte Speichereinheit, die über eine einzige Adresse erreicht werden kann, ist ein *Word* (i.a. ein Vielfaches eines Bytes).

Informationen über die Adresse der Operanden sind meist in den Maschinenbefehlen enthalten. Diese Adressinformation definiert den Speicherplatz derjenigen Operanden, auf die sich der Befehl bezieht. Die Information kann als Registernummer oder Hauptspeicheradresse (physikalische Adresse) oder als Vorschrift zur Berechnung einer Adresse (Adress-Spezifikation) vorliegen.

Adressierungsarten 1

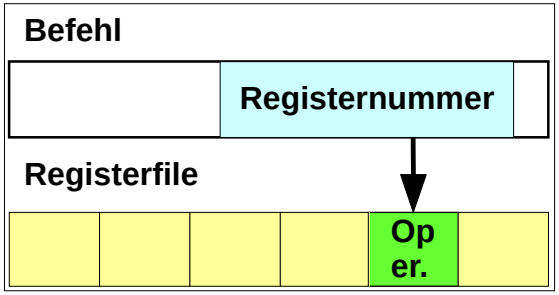
Es wird zwischen den folgenden Adressierungsmodi unterschieden:

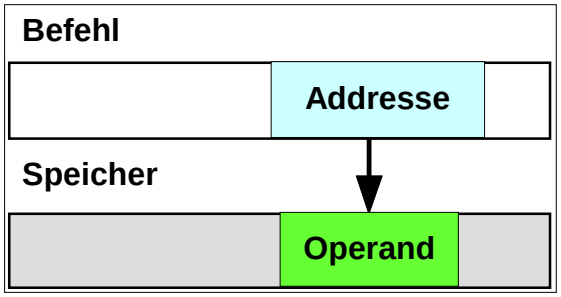
Unmittelbare Adressierung:



Der Operand ist im Maschinenbefehl enthalten.

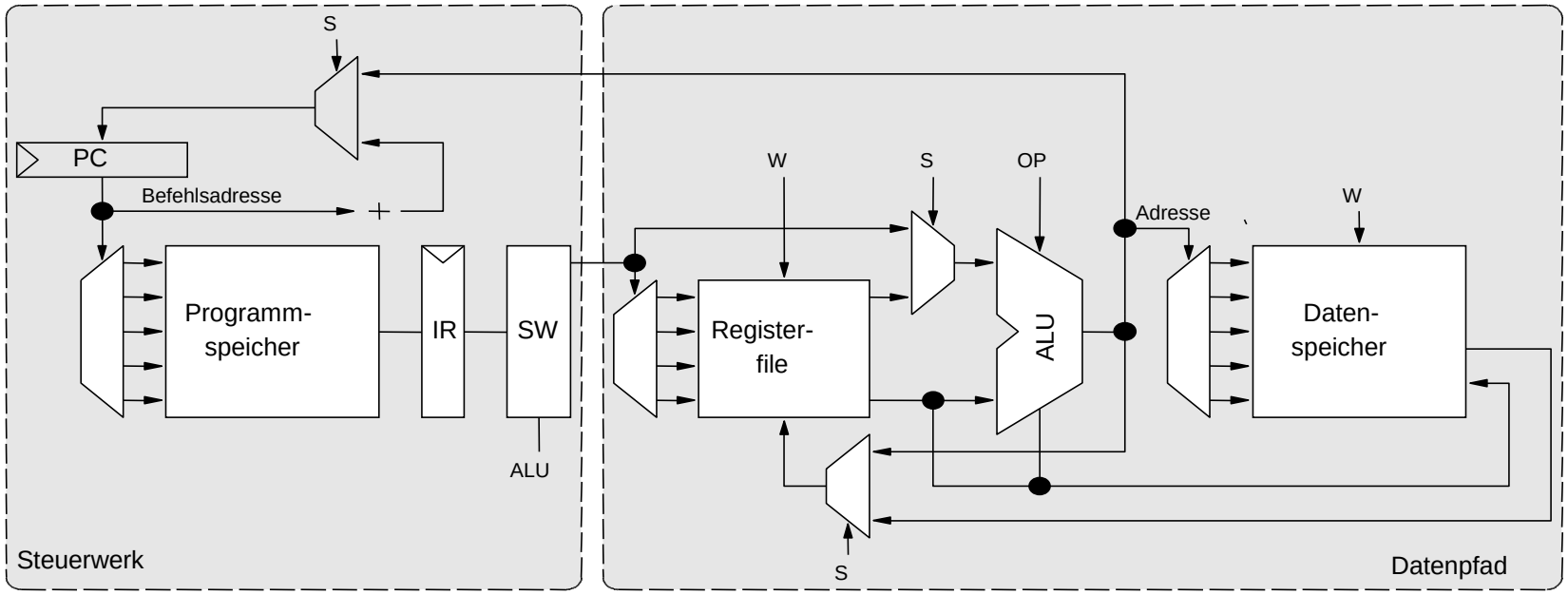
Register-/Speicher-direkte Adressierung:





Die Adresse des Operanden (Registernummer oder Hauptspeicherzelle) ist im Befehl enthalten.

Welche Teile sind bei einer Register-direkten Speicherung aktiv



ALU: Arithmetical Logical Unit

PC: Befehlszähler (Program Counter)

IR: Instruktionsregister (Instruction Register)

SW: Steuerwerk

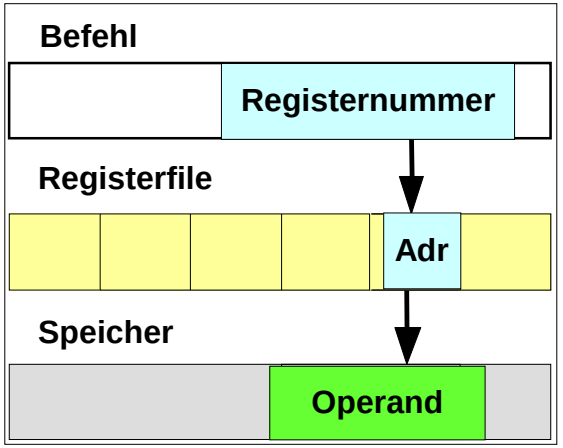
Adressierungsarten 2

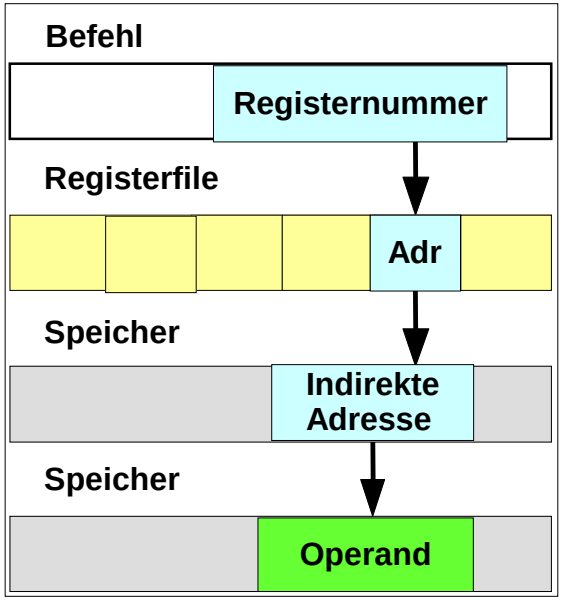
Register-indirekte Adressierung:

Die Adresse des Operanden steht in einem Register, dessen Nummer im Befehl enthalten ist.

Speicher-indirekte Adressierung:

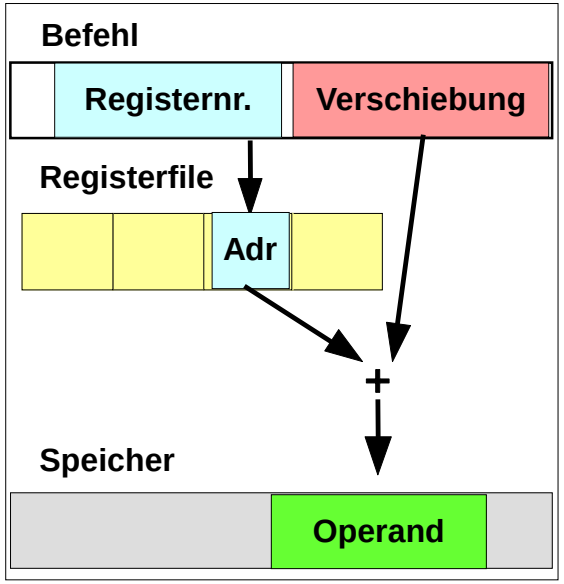
Die Adresse des Operanden steht in einer Hauptspeicherzelle, die durch ein Register adressiert wird.





Adressierungsarten 3

Register-relative Adressierung



Die Operandenadressen werden als Summe aus Registerinhalt und einer Verschiebung, die im Maschinenbefehl angegeben ist, gewonnen.

ISA: Maschinenbefehle

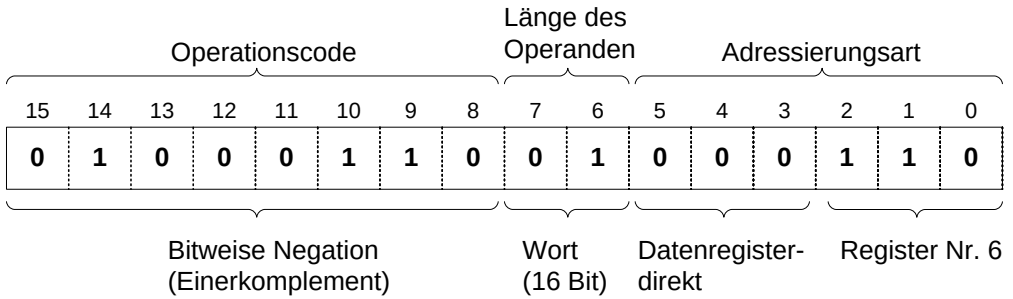
Maschinenbefehlssatz:

Der Maschinenbefehlssatz eines Rechners enthält üblicherweise Befehle unterschiedlichen Formats (z.B. Ein-, Zwei und Dreiadressbefehle). Jeder Befehl ist in Felder aufgeteilt, wobei ein Feld für den Operationscode steht. In den weiteren Feldern stehen - je nach Operationscode - das Quell-/Zielregister, ein Operand oder eine Adresse. Ein Maschinenbefehl ist also eine bestimmte kodierte Bitkette.

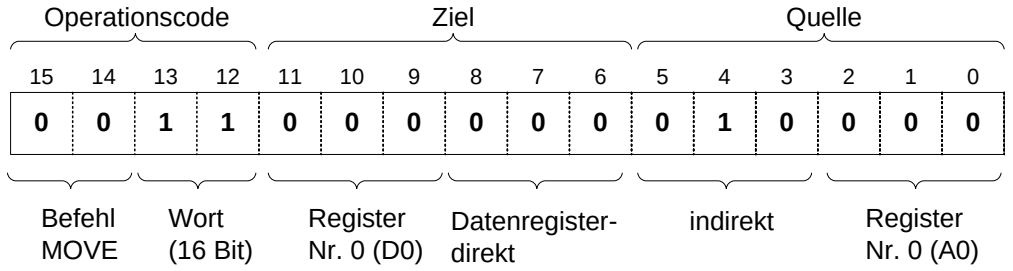
Ein Maschinenbefehl des Motorola 68000 besitzt eine Länge von 16 Bit und ist ein Zwei-Adress-Code.

Beispiel (Motorola 68000):

NOT D6 (Bitweise Negation von Register D6):



MOVE (A0), D0 (Übertrage das Wort an der mit A0 adressierten Speicheradresse in das Register D0):



2.3.3 Assembler

Der Assembler ist eine Kurzschreibweise für Maschinenbefehle, um diese für den Menschen besser lesbar zu machen.

Einige Beispiele für Befehle des 68000-Assemblers:

Wertzuweisung:

```
MOVE D0,D1          (schreibe den 16-Bit Wert von Register D0 nach D1)
```

Arithmetische Befehle:

```
ADD  #35,D0         ( D0 <= D0 + 35),
MULU D2,D3          ( D3 <= D2 * D3),
SUB  (A0),D4        ( D4 <= D4 - (A0)),
EOR  300(A0),D1     ( D1 <= (A0+300) ⊗ D1), …
```

Sprungbefehle:

```
BRA                 (unbedingter Sprung)
CMP D1,D2           (bildet Differenz d = D2 - D1), anschließend Vergleich:
BEQ (falls d = 0), BNE (falls d ≠ 0), BGT (falls d > 0), BLT (falls d < 0), …
BSR subr            (Subroutinen-Aufruf),
RTS                 (Subroutinen-Rückkehr)
```

Registeroperationen:

```
MOVE D1,(A0)        ((A0) <= D1)
```

Assembler

Ein einfaches Assemblerprogramm für die Summe der ersten *n* Zahlen.

Eingabe: *n*.

Hier wird solange *k* erhöht und zur Summe *s* addiert, bis *k* = *n*.

```
        MOVE   #0,D0    | Summe s (Register D0) auf 0 setzen
        MOVE   #0,D1    | Zählwert k (Register D1) auf 0 setzen
        MOVE   n,D2     | Wert n in Register D2 schreiben
marke1  CMP    D2,D1    | vergleiche n und k (in Register D2 und D1)
        BLE    marke2   | falls nicht größer, springe zu marke2
        ADD    #1,D1    | Zählwert k in D1 um 1 erhöhen
        ADD    D1,D0    | Zählwert k zu s addieren
        BRA    marke1   | unbedingter Sprung zu marke1
marke2                  | Ende
```

0,1,2,3 – Adressmaschinen

- Die (max.) Anzahl der Operanden in einem Maschinenbefehl bezeichnet den Typ der Maschine

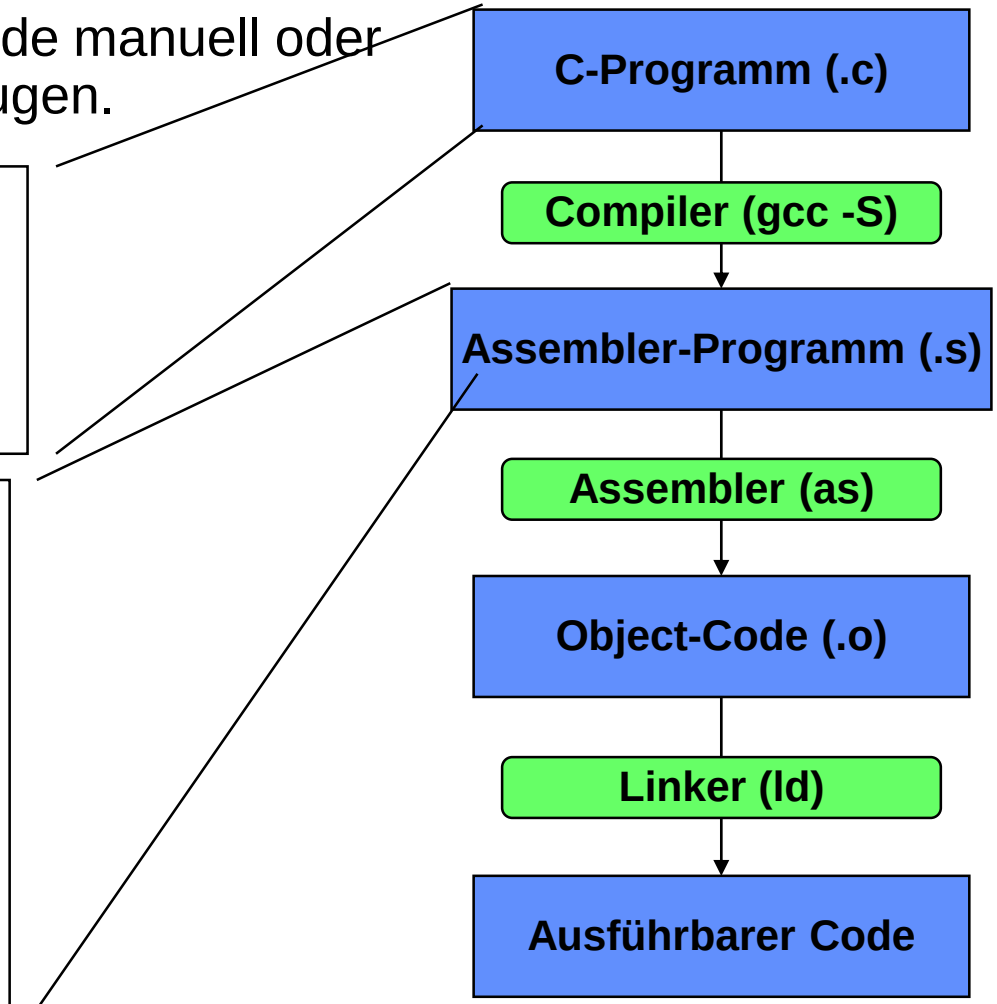
- Add #35, D0, D2 *3-Adressmaschine (z.B. ARM)*
- Add $35, D0 *2-Adressmaschine (z.B. 80X86, 68000)*
- Add D0 *1-Adressmaschine (z.B. 6502)*
  - Nahezu alle Operationen bezüglich eines Akkumulators
- Add *0-Adressmaschine (z.B. Postscript)*
  - Laden u. Speichern bezüglich des Stacks

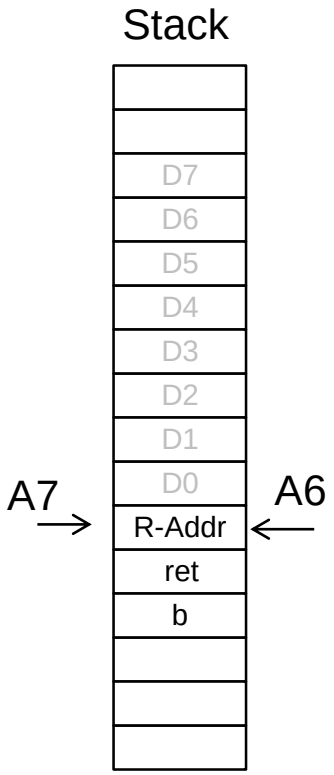
Kompilierung

Man kann Assemblercode manuell oder durch kompilieren erzeugen.

```
int add(int a, int b)
{
    int ret = a + b;
    return ret;
}
```

```
_add
      MOVEA.L A7, A6
      MOVEM D0-D7, -(A7)
      MOVE.L +(A6), D3
      MOVE.L +(A6), D2
      ADD.L D3, D2
      MOVE.L D2, -(A6)
      MOVEM (A7)+, DO-D7
      RTS
```

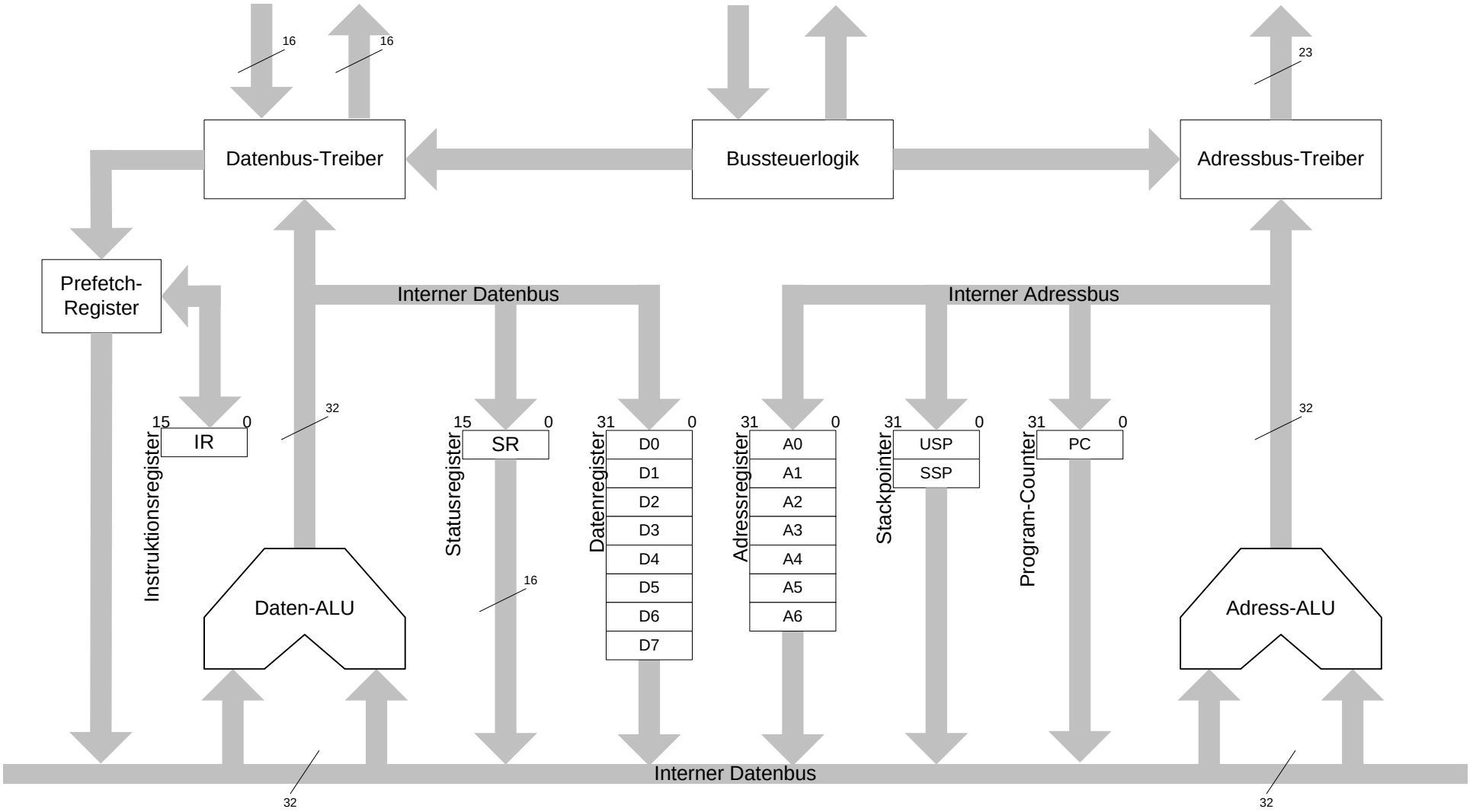




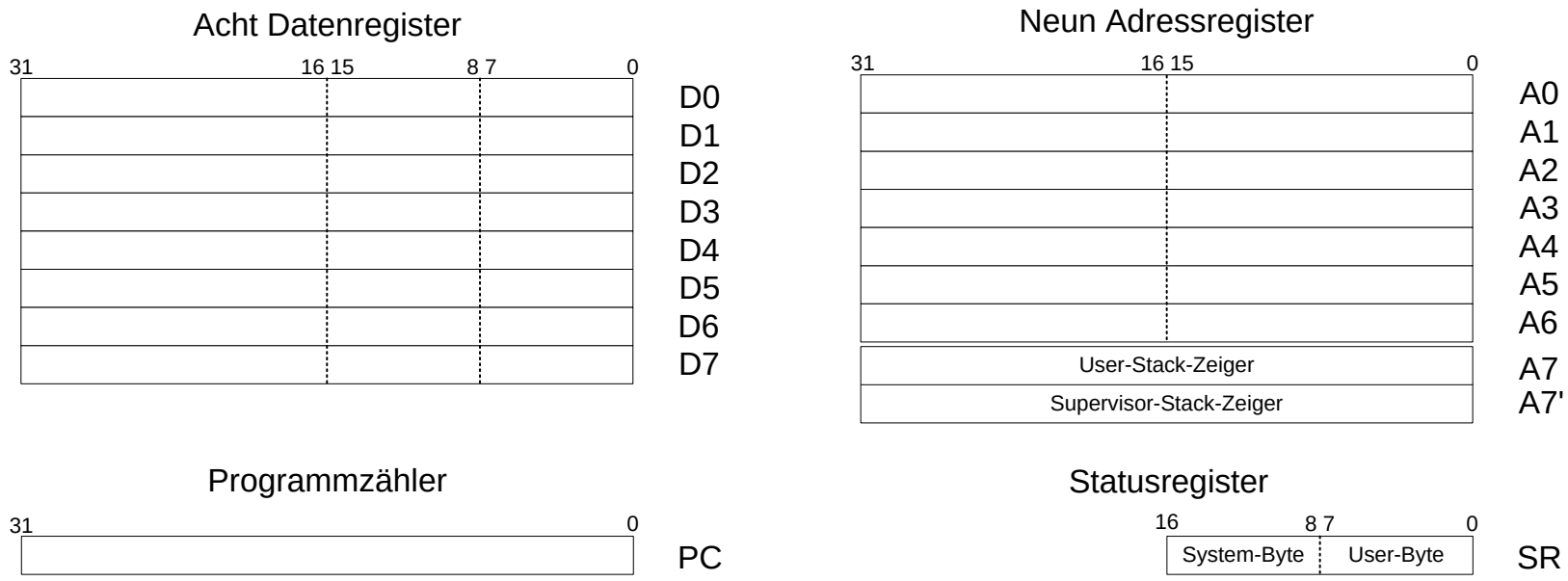
Der Motorola 68000 Prozessor

- Eingeführt 1979
- 68000 Transistoren
- 7 MHz Taktfrequenz, später bis 66 Mhz
- Operationen benötigen zwischen 4 und 158 Taktzyklen
- 16 MB linearer Adressraum (23+1-bit)
- 16-bit ALU, 32-bit Register
- 2-stufige Pipeline
- CISC → „Programmierer-freundliche“ Operationen
- Eingesetzt u. a. in:
  - Apple Macintosh
  - Atari ST
  - Amiga 500 – 2000
  - Sega Megadrive
  - TI 92
  - Palm Handhelds

Operationswerk des 68000 Prozessor (HSA)



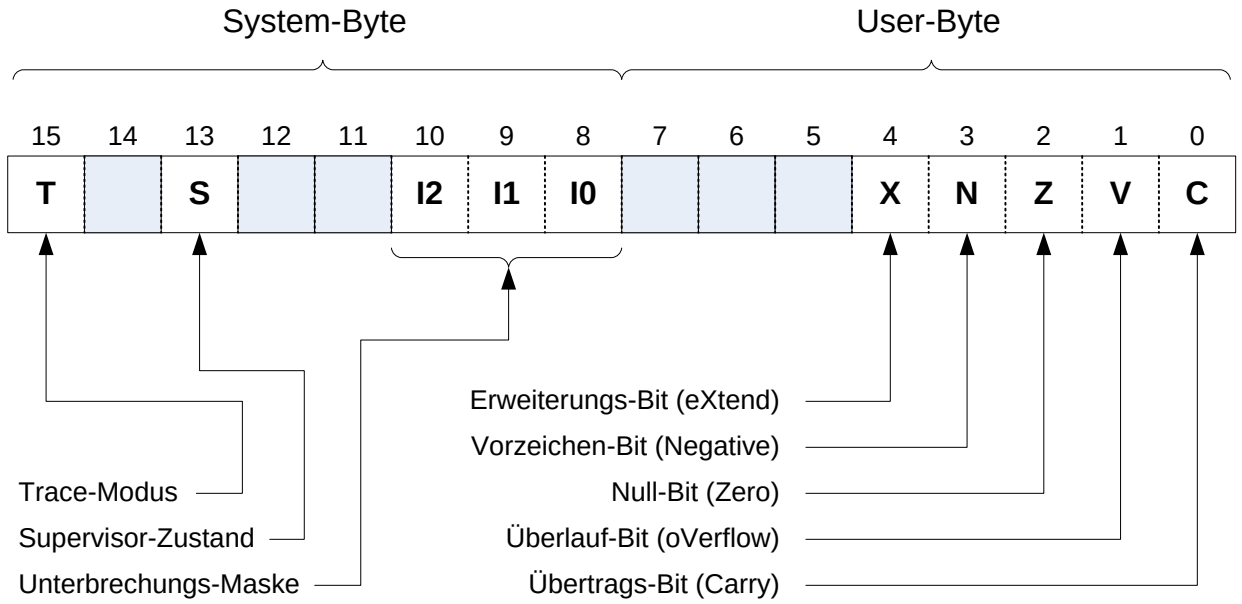
Registermodell des 68000 Prozessor



Das Registermodell beschreibt den für den Programmierer zugänglichen Bereich der Register eines Prozessors. Die acht Datenregister des 68000 haben eine Breite von 32 Bit. Die neun Adressregister sowie der Programmzähler besitzen ebenfalls eine Breite von 32 Bit.

Adressierbar sind beim 68000 das gesamte Register mit Operationszusatz *.L* (Bsp.: MOVE.L), die unteren 16 Bit mit *.W* oder ohne Zusatz (Bsp.: MOVE.W oder MOVE) und das unterste Byte (8 Bit) mit *.B* (Bsp.:MOVE.B).

Das Statusregister des 68000

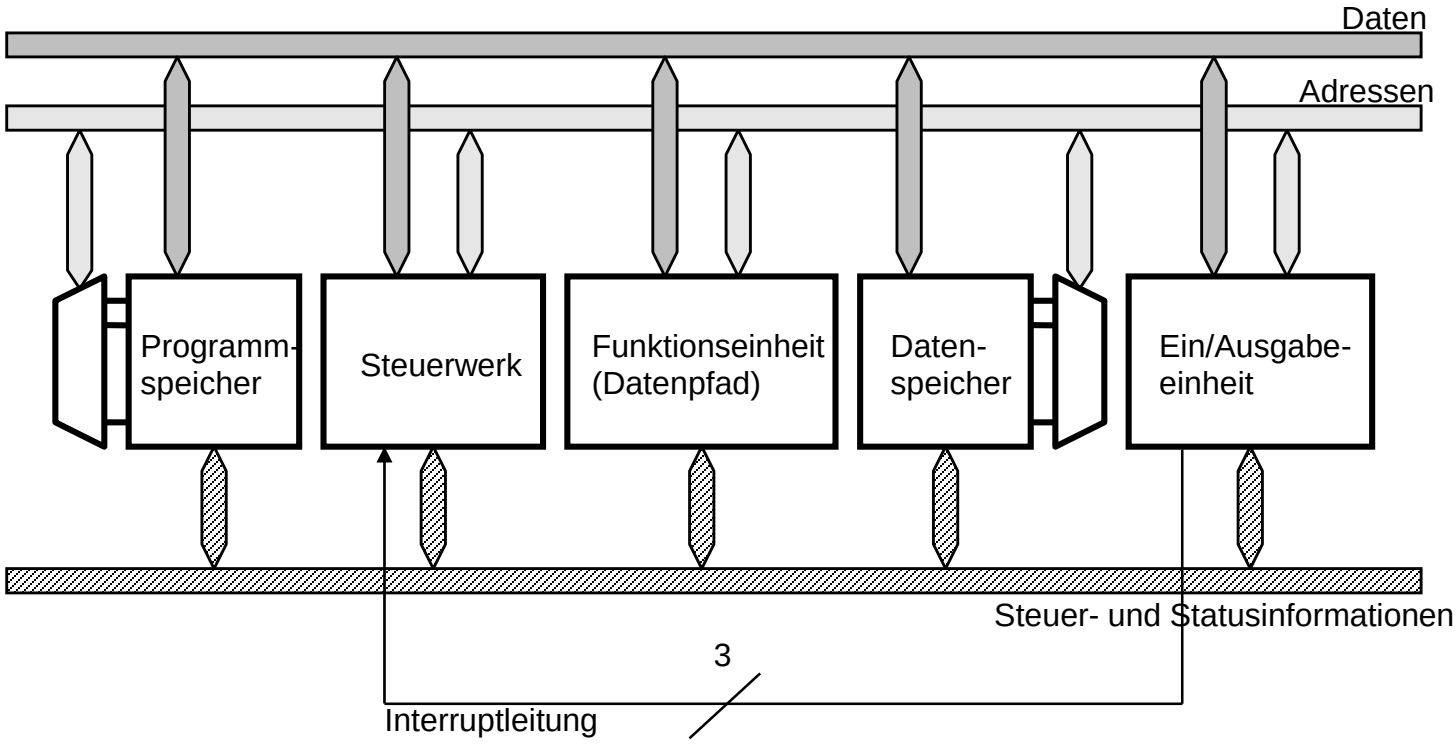


Die niederwertigsten 5 Bit des Statusregisters sind die Bedingungscode-Bits, die von Operationen gesetzt werden. Eine nachfolgende Operation kann davon abhängige Eigenschaften besitzen.

Beispiel: Vergleich zweier Register D0 und D1. Die Operation CMP führt D1-D0 durch. Ist das Subtraktionsergebnis 0 wird das Null-Bit Z gesetzt. Eine nachfolgende bedingte Verzweigung BEQ (branch if equal) erfolgt nur, wenn das Null-Bit gesetzt ist.

2.3.4 Interrupts

Um die Kommunikation mit der externen Hardware effizient zu ermöglichen kann man den Programmablauf durch Interrupts unterbrechen.



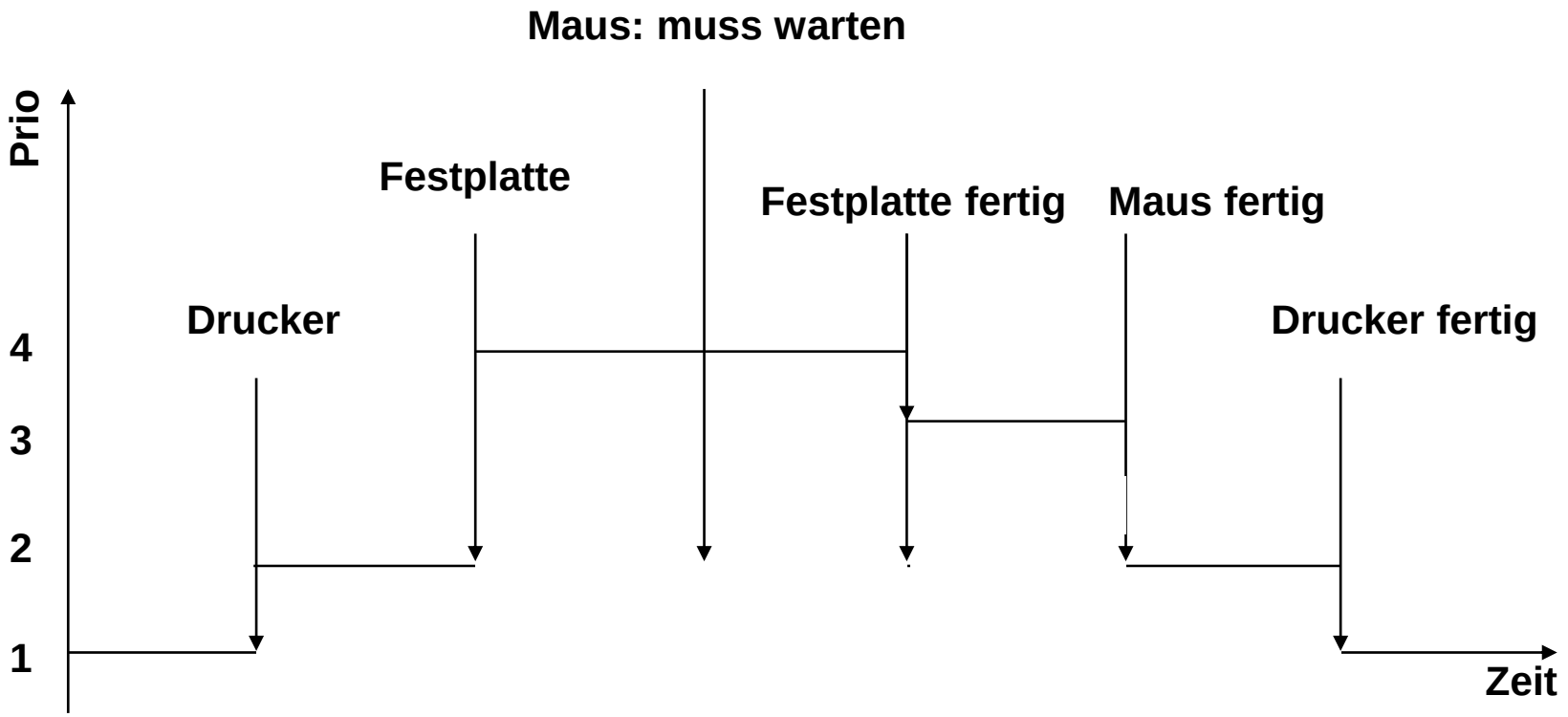
Ablauf eines Interrupt

1. Das Gerät löst einen Interrupt mit einer bestimmten Nummer aus
2. Wenn die CPU den Interrupt annehmen kann, nimmt sie ihn an, sonst wird er in eine Warteschlange eingereiht.
3. Das aktuelle Programm wird unterbrochen, alle Register werden auf den Stack gerettet.
4. Über die Nummer des Interrupts wird in einem Feld die Adresse der Interrupt-Service-Routine angesprungen.

- Ab jetzt sind alle weiteren Interrupts mit niedrigerer Priorität gesperrt
- Es wird das Gerät gesucht, welches den Interrupt ausgelöst hat (bei mehreren Geräten mit der selben Interruptnummer). Die Ein-/Ausgabe des Geräts wird behandelt.

1. Nach Behandlung des Interrupts werden die Register zurück geladen
2. Die Funktion Return from Interrupt springt in den unterbrochenen Code zurück und erlaubt wieder die Interrupts.

Prioritäten bei Interrupts



2.3.5 RISC-/CISC-Architekturen

RISC-Architekturen: (**r**educed **i**nstruction **s**et **c**omputer)

RISC-Architekturen zeichnen sich durch ein einfaches Befehlsformat und orthogonale Befehle aus. Dies führt zu kurzen Taktzyklen und zu einer Reduktion der mittleren CPI (cycles per instruction).

Zu den wichtigsten Aspekten einer RISC-ISA zählen:

- wenige Adressierungsarten
- wenige Befehlsformate
- Load-Store-Architektur
- viele sichtbare Prozessorregister

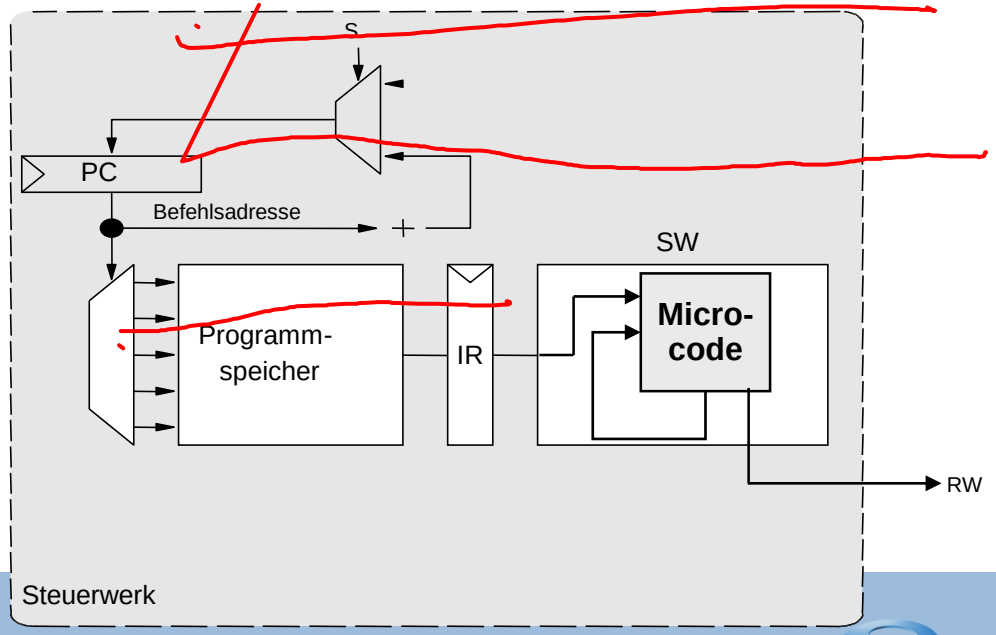
Die wichtigsten Aspekte einer RISC-HSA sind die direkte (festverdrahtete) Steuerung und das Fließbandprinzip (Pipelining).

RISC-Architekturen führen zu einer einfachen Befehlsdekodierung. Komplexe Operationen müssen jedoch als Folge einfacher Maschinenbefehle realisiert werden. => Verlagerung der Abarbeitung komplexer Aufgaben in den Compiler

RISC-/CISC-Architekturen

CISC-Architekturen: (**c**omplex **i**nstruction **s**et **c**omputer)

CISC-Architekturen zeichnen sich durch viele mächtige Befehle und Adressierungsarten aus. Ziel ist eine Reduktion der Anzahl im Mittel von einem Programm ausgeführten Befehle *IC* (instruction count). Dies wird durch eine Funktionsverlagerung in die Hardware erreicht. Die Programmierung wird dadurch vereinfacht. Die Interpretation der Befehle wird üblicherweise durch ein mikroprogrammiertes Steuerwerk vorgenommen.



Load-Store-ISA

RISC

Die Ausführungszeit eines Befehls wird größer, wenn die benötigten Operanden nicht in Registern zur Verfügung stehen und somit aus dem Hauptspeicher geladen werden müssen. In einer Load-Store-Instruktions-Satz-Architektur führen nur Register-Speicher- und -Lade-Befehle zu einem Hauptspeicherzugriff:

- *LOAD < Register, ea >*

  Lade Register mit dem Datum in der Speicherzelle mit der effektiven Adresse *ea*.

- *STORE < Register, ea >*

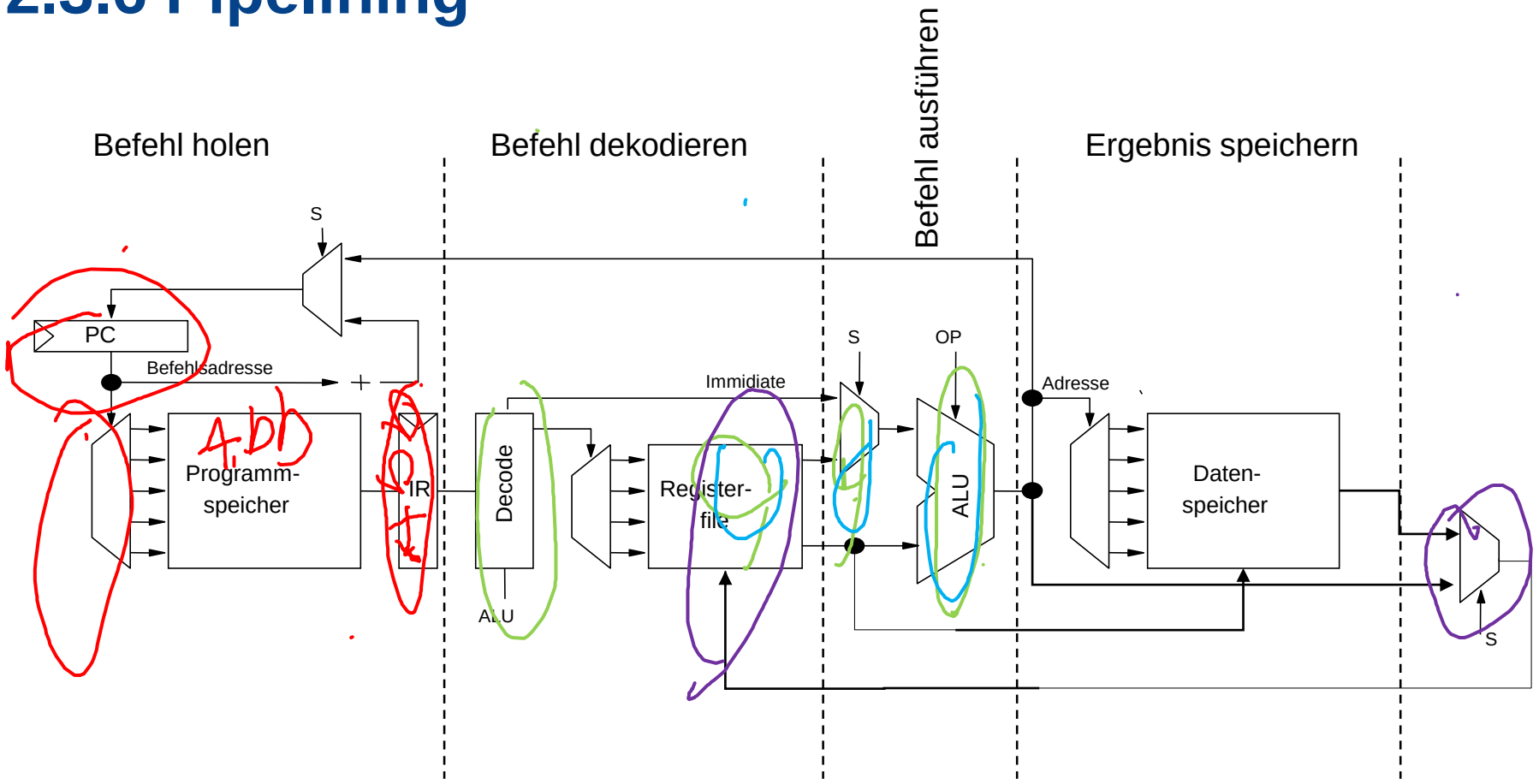
  Speichere Registerinhalt in die Speicherzelle mit der effektiven Adresse *ea*.

Alle anderen Befehle operieren nur mit Registeradressen. Wenn ausreichend viele Register vorhanden sind, verringert die Load-Store-Architektur die Anzahl der Hauptspeicherzugriffe und beschleunigt die Befehlsausführung.

Vergleich RISC/CISC

	RISC	CISC
Ausführungszeit	1 Datenpfadzyklus	>= 1 Datenpfadzyklus
Instruktionszahl	klein	groß
Steuerung	Hardware	Mikroprogramm
Hauptspeicherzugriffe	Load/Store Architektur	Keine Einschränkungen
Pipelining	Einfach	Schwierig
Beispiel	SPARC, ARM, PowerPC	x86, 68000

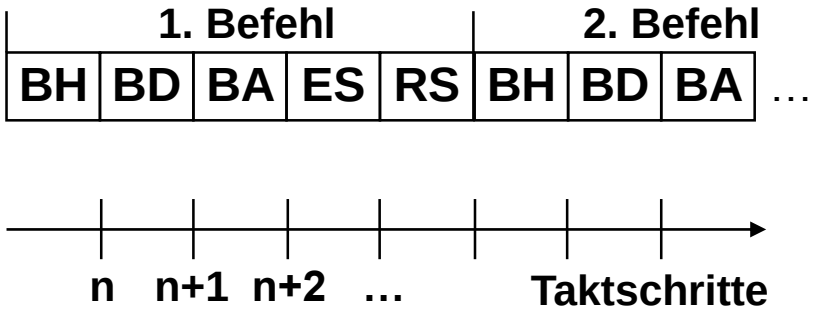
2.3.6 Pipelining



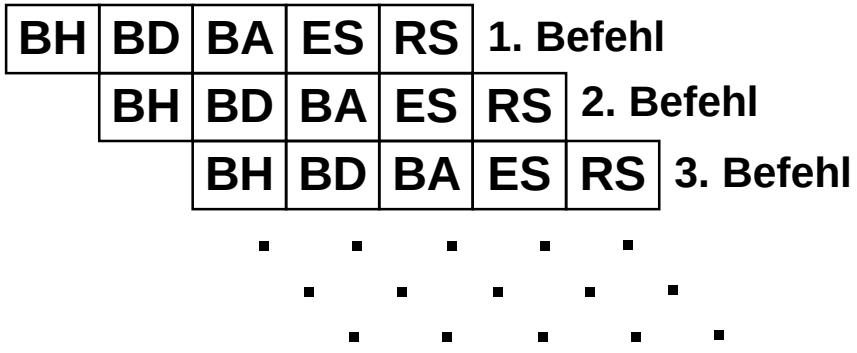
Die Ausführung eines Befehls dauert relativ lange. Eine Erhöhung der Taktfrequenz scheitert, es sei denn man unterteilt die Befehlsausführung in mehrere Schritte (hier 5). Dann kann die Taktfrequenz auf das 5-fache angehoben werden. Dieses schrittweise Ausführen nennt man Pipelining.

Ausführung der Befehle

Auführung ohne Pipelining:



Ausführung mit Pipelining:



Ein Prozessor besitze die Folge der Befehlsabarbeitung:

- BH = Befehl holen
- BD = Befehl decodieren/Operanden holen
- BA = Befehl ausführen
- ES = Ergebnis speichern
- RS = Register zurückschreiben

Ohne paralleles Abarbeiten der Pipeline: Hintereinanderreihen der Einzelschritte.

→ **1 Befehl in 5 Taktschritten**

Paralleles Abarbeiten der Pipeline (Pipelining): Jede der Einheiten arbeitet in jedem Taktschritt auf einem Befehl. Bei ausgelasteter Pipeline:

→ **5 Befehle in 5 Taktschritten**

Problem: Datenabhängigkeiten

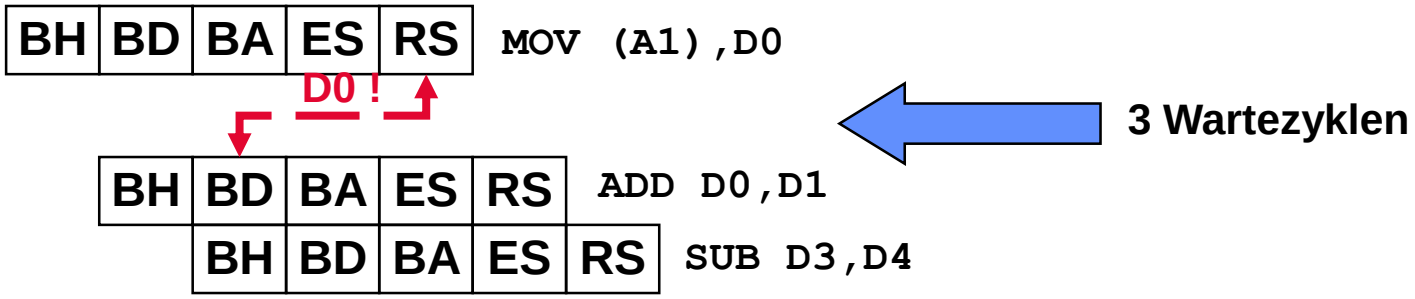
Pipelining: Hazards

- **Datenhazard**
  Ein Datum wird noch in der Pipeline berechnet obwohl es schon gebraucht wird
- **Bedingte Verzweigungen**
  Ein Verzweigung bewirkt, dass ganz anderer Programmcode ausgeführt werden muss
- **Abhilfe:**
  - **Generell:**
    - Löschen des Pipelineinhalts (Flush)
    - Wartezyklen einfügen
  - **Datenhazards: Reordering**
    Die Reihenfolge der Befehle wird dynamisch umgeordnet um Datenkonflikte aufzulösen. Das geht nicht immer.
  - **Verzweigungen: Spekulative Ausführung**
    Es wird eine Entscheidung angenommen und spekulativ ausgeführt. Bei Eintreten der anderen Entscheidung müssen alle Ergebnisse verworfen werden!

Beispiel Hazards

- **Beispielprogramm**

```
      MOV  (A1),D0
LOOP  ADD  D0,D1
      SUB  D3,D4
      CMP  #0,D1
      BNE  LOOP
      MOV  D4,(A4)
      ...
```

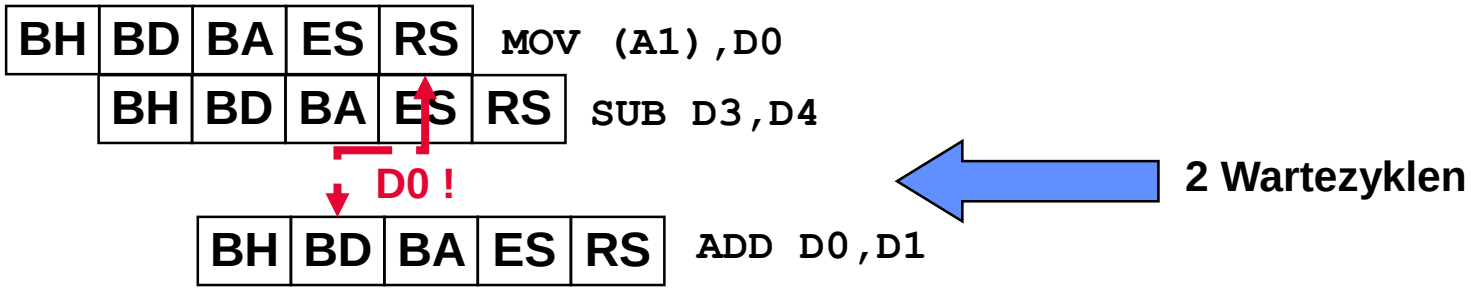


Beispiel Hazards

- **Beispielprogramm**

```
      MOV  (A1),D0
LOOP  ADD  D0,D1
      SUB  D3,D4
      CMP  #0,D1
      BNE  LOOP
      MOV  D4,(A4)
      ...
```

Mit Reordering:

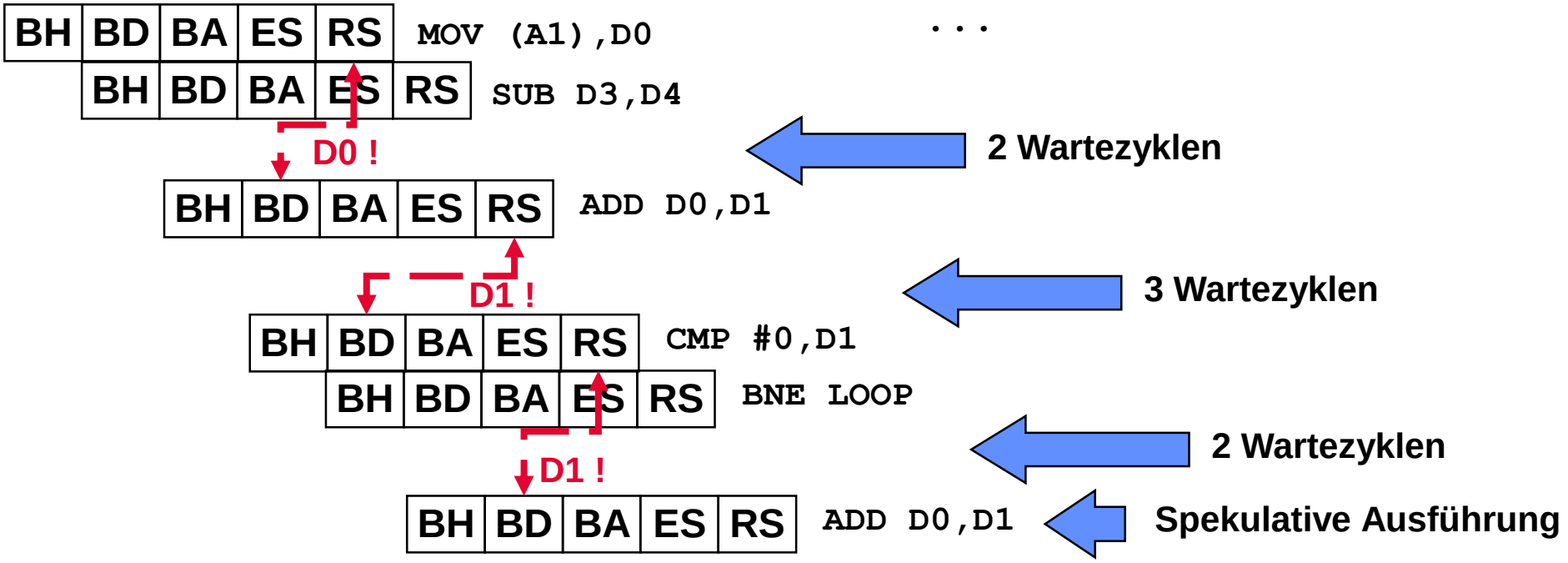


Beispiel Hazards

- **Beispielprogramm**

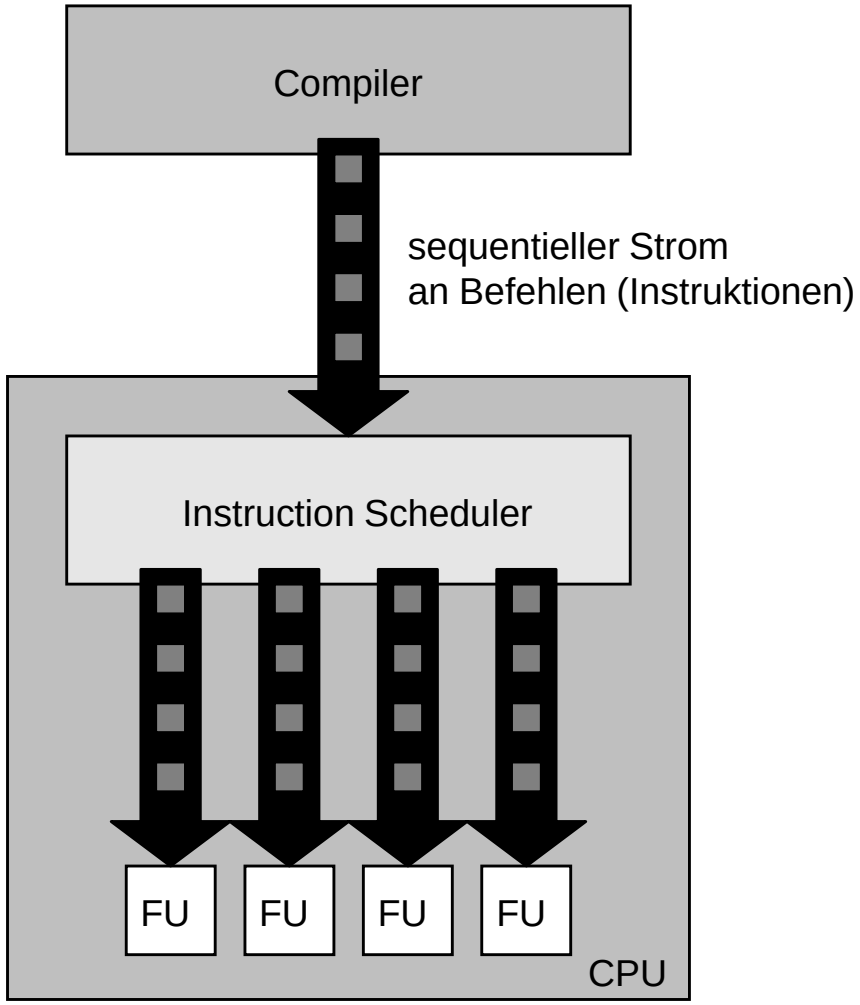
```
      MOV  (A1),D0
LOOP  ADD  D0,D1
      SUB  D3,D4
      CMP  #0,D1
      BNE  LOOP
      MOV  D4,(A4)
      ...
```

Mit Reordering:



2.4 Parallele Strukturen in Prozessoren

2.4.1 Superskalare Prozessoren



Moderne (Mikro)-Prozessoren enthalten mehrere Funktionseinheiten (FUs), so dass mehrere Instruktionen parallel ausgeführt werden können. Dies bezeichnet man auch als Instruktionen-Parallelität (ILP).

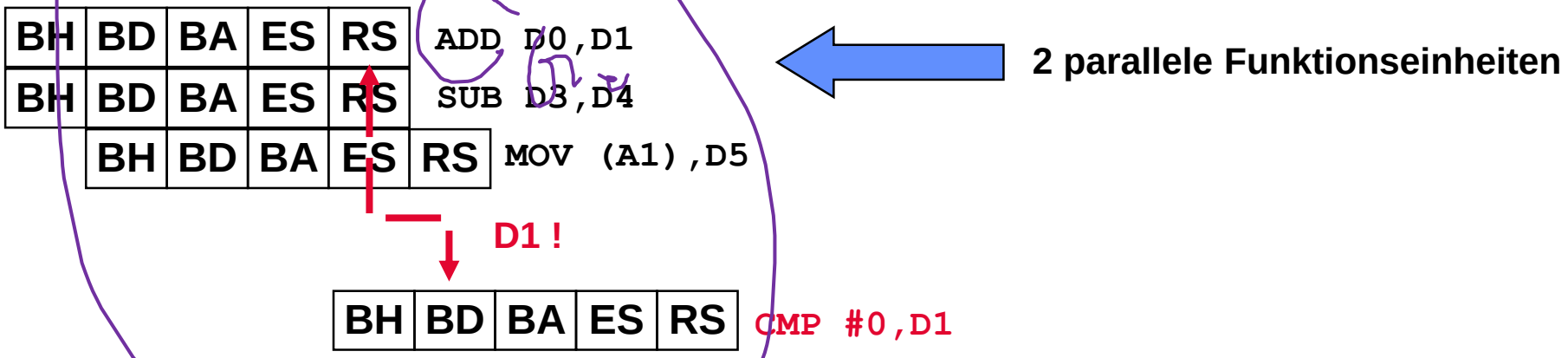
Bei Superskalaren Prozessoren (z.B. Pentium) erfolgt die Zuordnung an die Funktionseinheiten zur Laufzeit des Programms durch einen Instruktions-Scheduler.

In der Darstellung sind zur Vereinfachung Programm- und Datenspeicher, Registerfile, Teile des Steuerwerks sowie die Busse weggelassen.

Beispiel: Instruktionsparallelität

- Beispielprogramm

```
ADD D0,D1
SUB D3,D4
MOV (A1),D5
CMP #0,D1
...
```

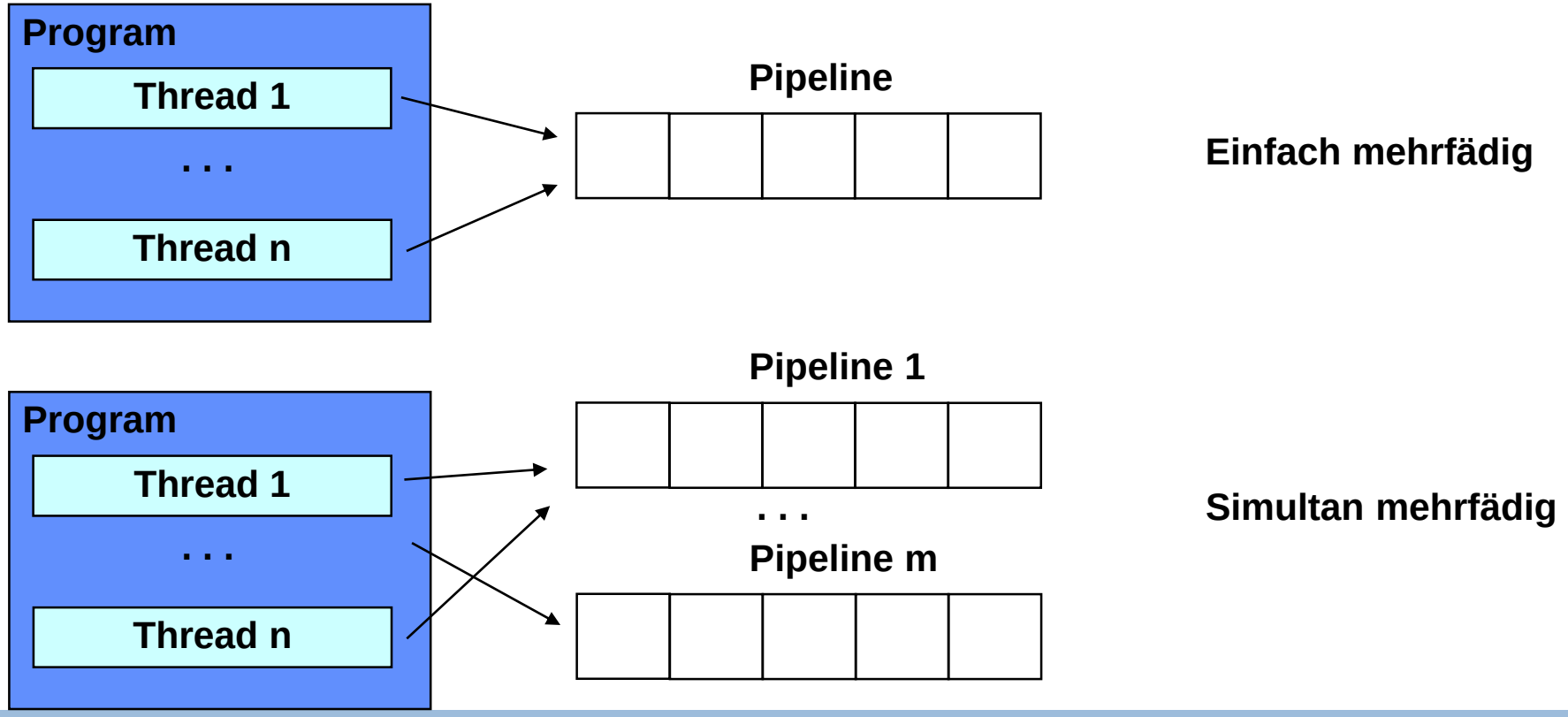


Mehrfädige Prozessoren (Multithreading, Hyperthreading)

Hardware-Unterstützung zur Ausführung mehrerer Kontrollfäden (Threads)

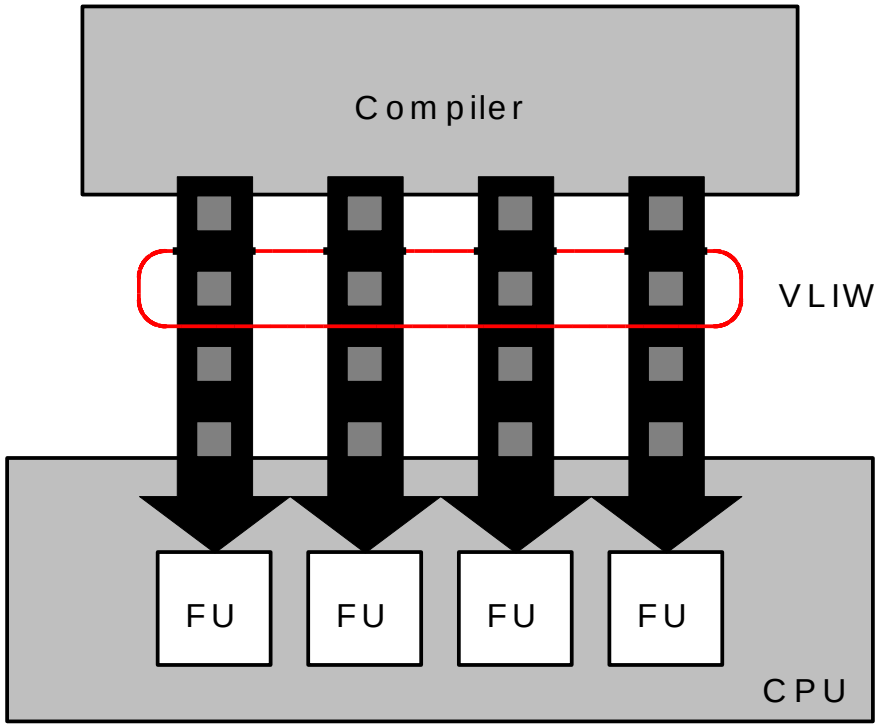
- Mehrfache Registersätze, Threadkennung in der Pipeline, …

Ursprüngliches Ziel: Überbrückung von Latenzen (z.B. Pipelinehazards)



2.4.2 VLIW-Prozessoren

Bei VLIW-Prozessoren (**V**ery **L**ong **I**nstruction **W**ord) existieren ebenfalls mehrere Funktionseinheiten. Im Gegensatz zu den Superskalaren Prozessoren erfolgt jedoch die Zuordnung der Befehle an die Funktionseinheiten zur Compile-Zeit.



Es existiert daher ein sequentieller, aber breiter (VLIW)-Strom an Befehlen. Moderne Prozessoren nach dem EPIC-Konzept (z.B. Itanium) basieren auf dem VLIW-Prinzip.

In der Darstellung sind zur Vereinfachung Programm- und Datenspeicher, Registerfile, Teile des Steuerwerks sowie die Busse weggelassen.

2.4.3 Multicore- / Manycore-Prozessoren

- Mehrere Prozessorkerne auf einem Chip
- Homogen (gleiche Kerne) oder heterogen (unterschiedliche Kerne)
- Meist Speicherkopplung
- Heute 2 – 16 Kerne
- Zukünftig mehr als 1000 Kerne => Manycore-Prozessoren

Problem: wie entwickle ich Programme, die so viele Kerne effizient nutzen?

Werkzeuge, neue Programmierpradigmen sind erforderlich

Designprinzipien moderner Prozessoren

- Alle Instruktionen direkt von der Hardware ausführen lassen
  Microcode von CISC-Prozessoren verlangsamt die Ausführung

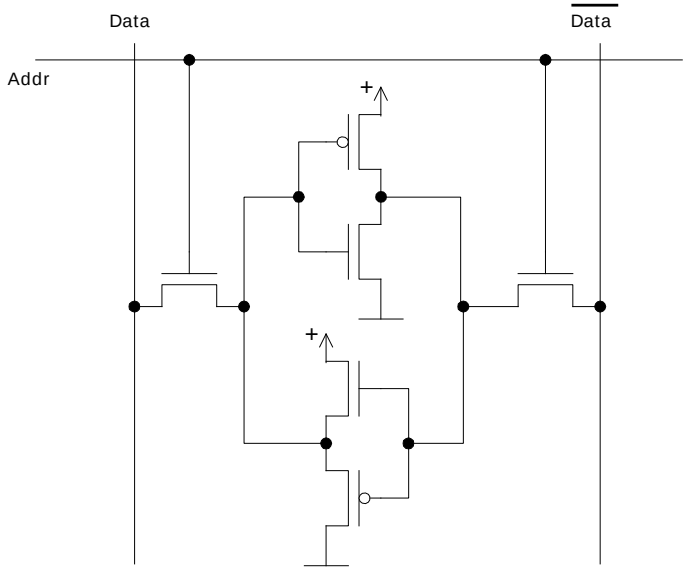
- Maximieren des Instruktionsdurchsatzes
  Die Latenzzeit einer Instruktion ist nicht entscheidend -> Pipelining

- Leicht zu dekodierende Befehle

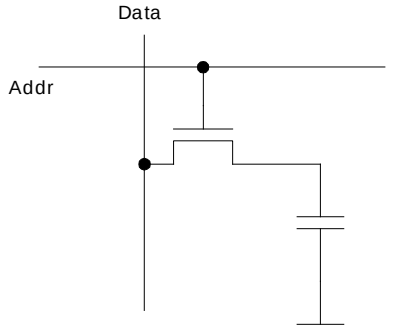
- Nur Lade- und Speicherbefehle sollen auf den Speicher zugreifen
  Sie haben eine große Latenzzeit. Sie können jedoch so besser verschoben werden

2.5 Speicher und Speicherhierarchie:
Hauptspeicher

- RAM (Random Access Memory)
  Schreib- und Lesespeicher
- Zwei wesentliche Typen
  - SRAM Statischer RAM
    - Die Speicherzellen behalten ihren Wert
    - Schnell
    - Viele Transistoren ⇒ teuer

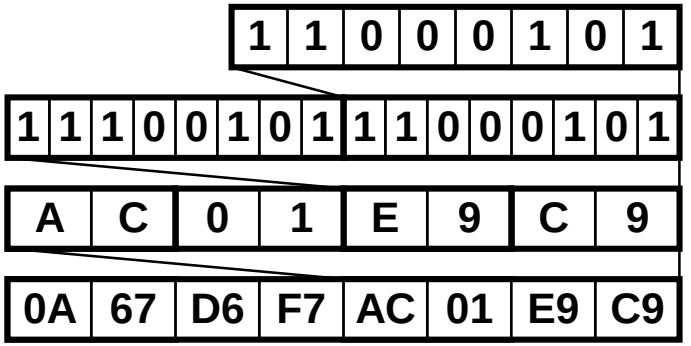


  - DRAM Dynamischer RAM
    - Die Speicherzellen müssen ab und zu aufgeladen werden
    - Langsamer
    - 1 Transistor + 1 Kapazität

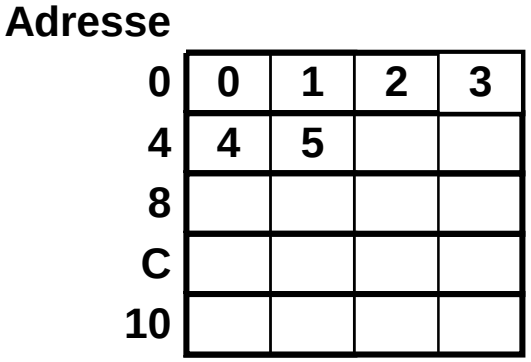


Speicheraufbau

- **Wortbreiten**
  - 8 Bit = 1 Byte
  - 16 Bit = 2 Byte
  - 32 Bit = 4 Byte
  - 64 Bit = 8 Byte

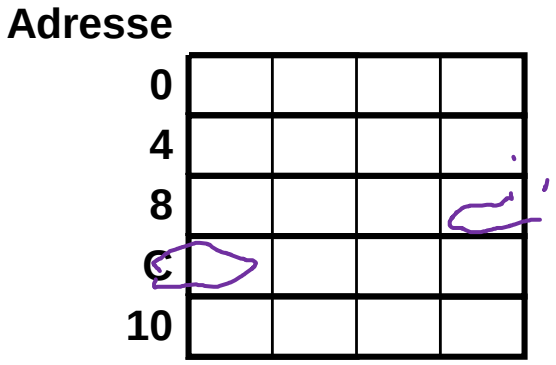


- **Speicheraufbau**
  - Adresszählung immer Byte-weise, auch wenn 64-Bit-Wörter abgelegt werden.

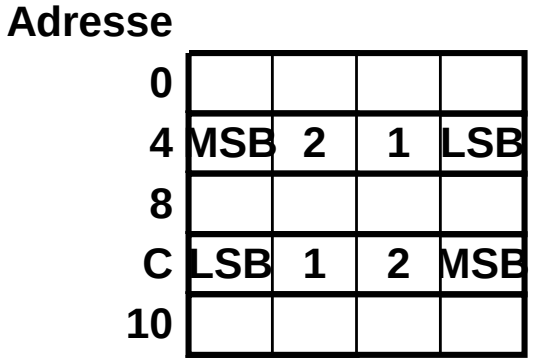


Speicheraufbau II

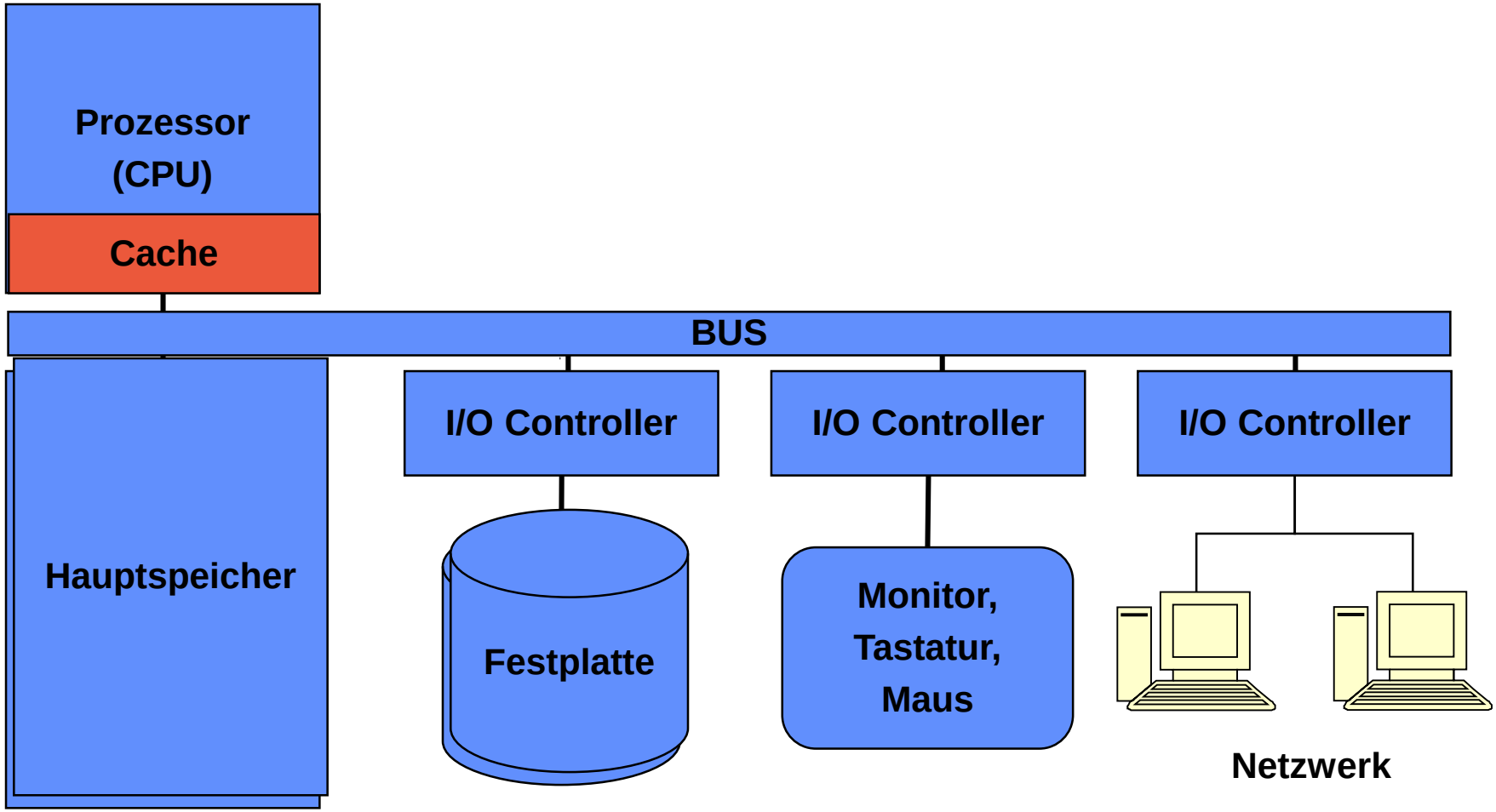
- **Anordnung im Speicher**
  - aligned
  - misaligned
  - je nach Wortgröße



- **Byte-Reihenfolge**
  - Big-Endian
  - Little-Endian

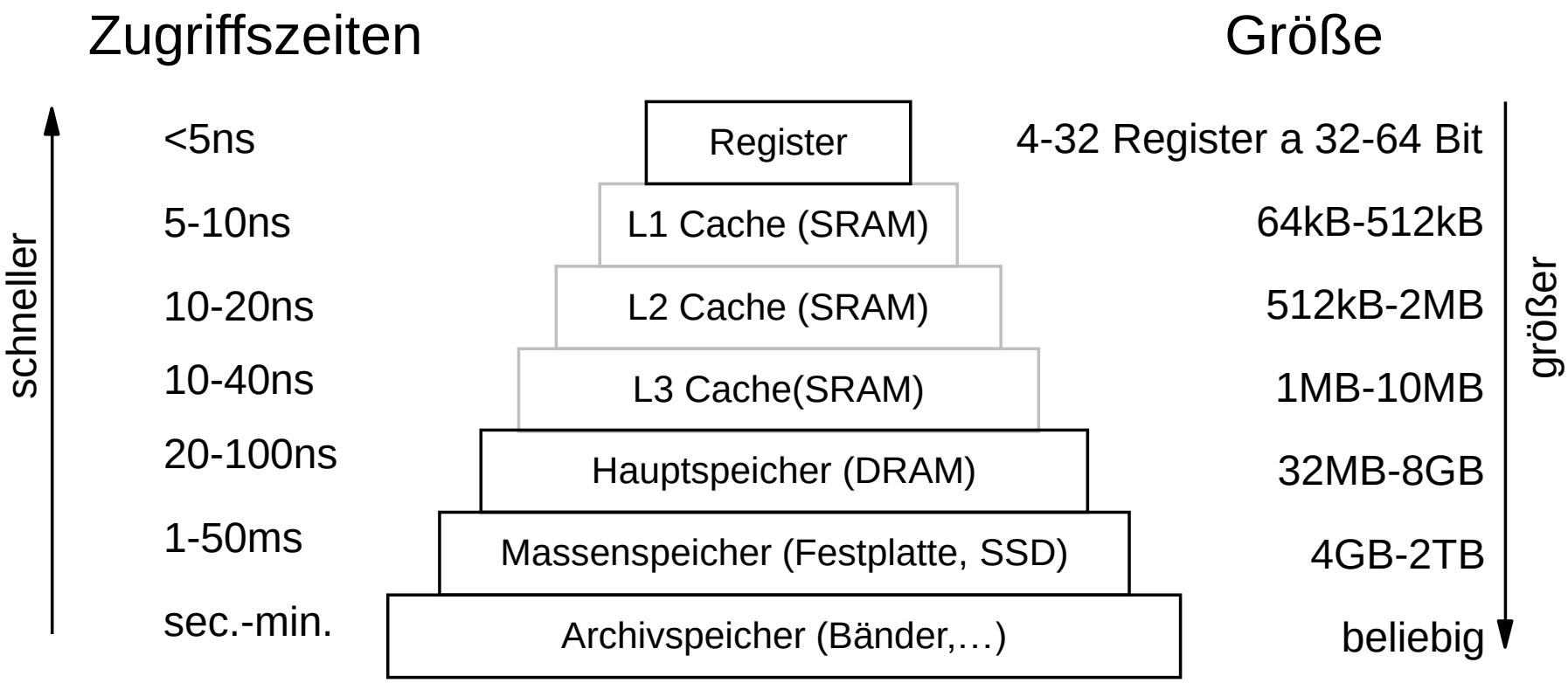


Wo sind Speicher?



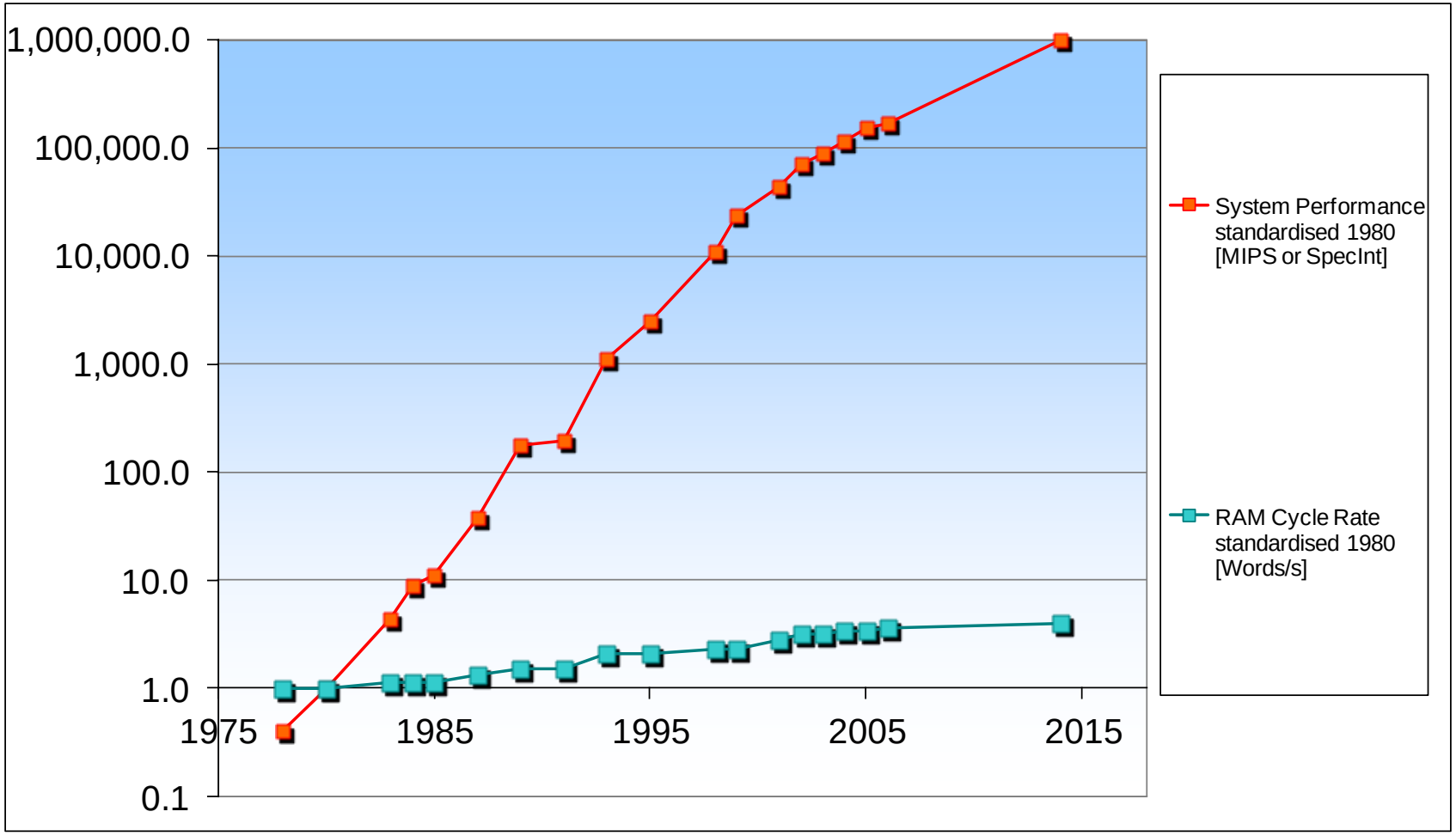
Speicherhierarchie

Die Speicherhierarchie entsteht im Zusammenwirken von Speichern unterschiedlicher Größe und Geschwindigkeit:



Die Caches sind alle transparent, also für den Anwender nicht sichtbar

Warum eine Speicherhierarchie?



Ausnutzung der Speicherhierarchie

Die sequentielle Abarbeitung von Befehlen und die Lokalisierung von Operanden in Datensegmenten führt zu einer *Referenzlokalität*. Darunter versteht man die Tendenz von Prozessoren, über eine Zeitspanne hinweg nur auf Daten und Instruktionen zuzugreifen, die kurz zuvor bereits referenziert wurden oder benachbarte Adressen haben. Man unterscheidet:

Räumliche Lokalität:

Der nächste Zugriff erfolgt auf eine benachbarte Speicherzelle.

Zeitliche Lokalität:

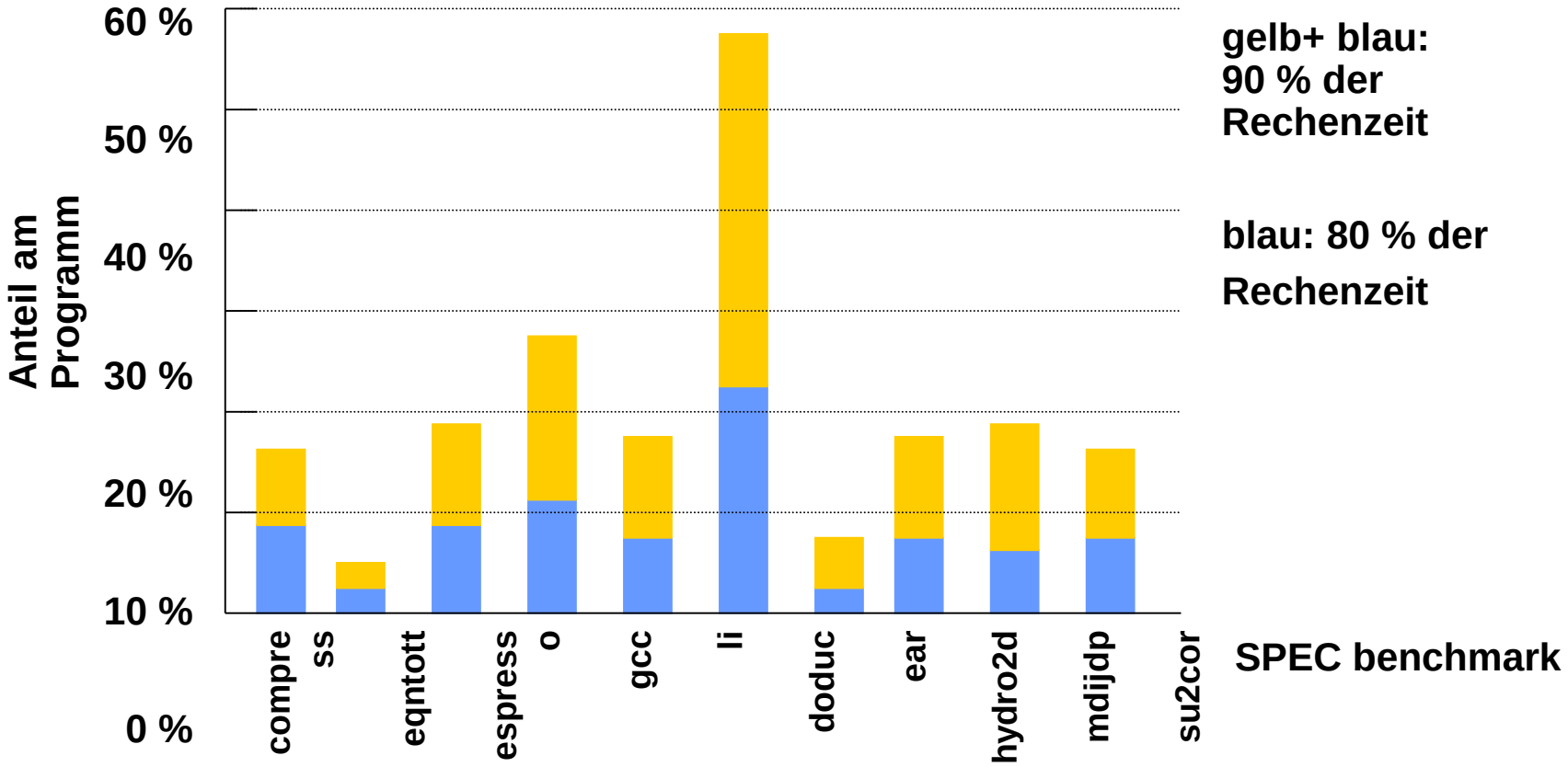
Der nächste Zugriff erfolgt auf eine Speicherzelle, auf die kurz zuvor bereits zugegriffen wurde.

Daraus folgt, dass jeweils nur ein Teil der Daten in den schnellen Speichern gehalten werden muss. Bei Bedarf werden benötigte Teile aus den großen und langsamen Speichern nachgeladen.

Die Lokalität muss bei der Programmierung möglichst erhalten bleiben!

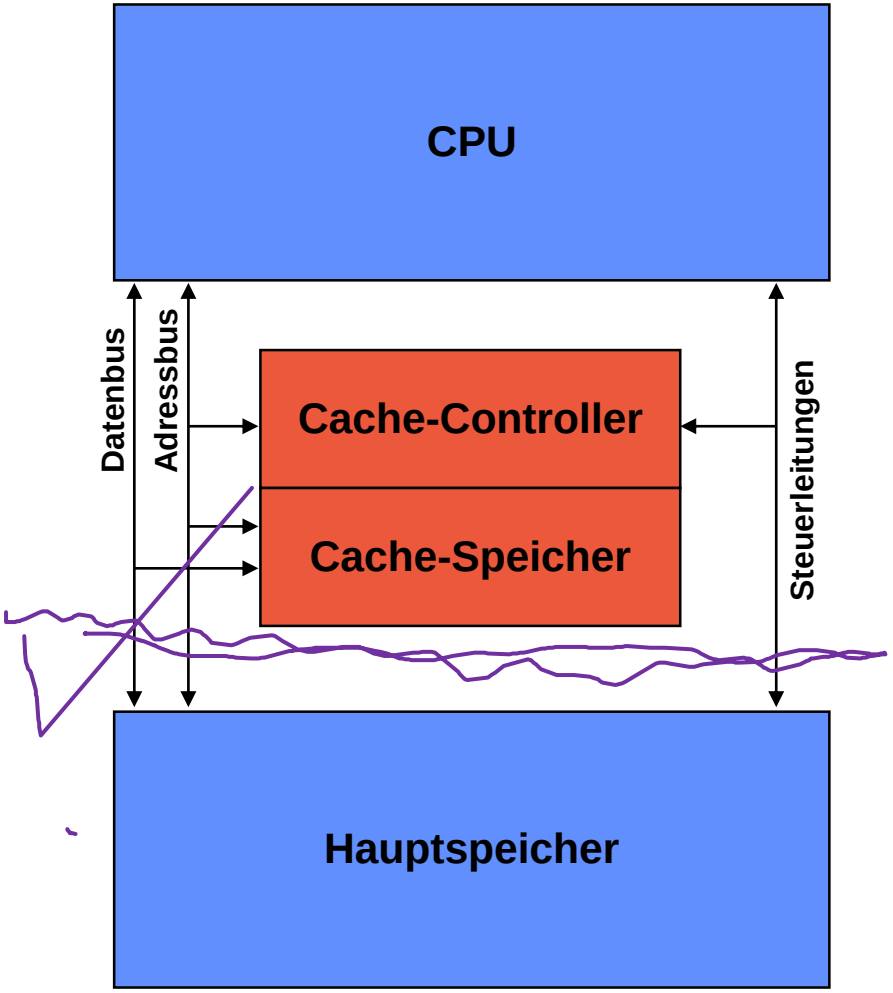
Lokalitätsprinzip

- Circa 10 % der Instruktionen in einem typischen Programm verbrauchen ca 80% der Rechenzeit.



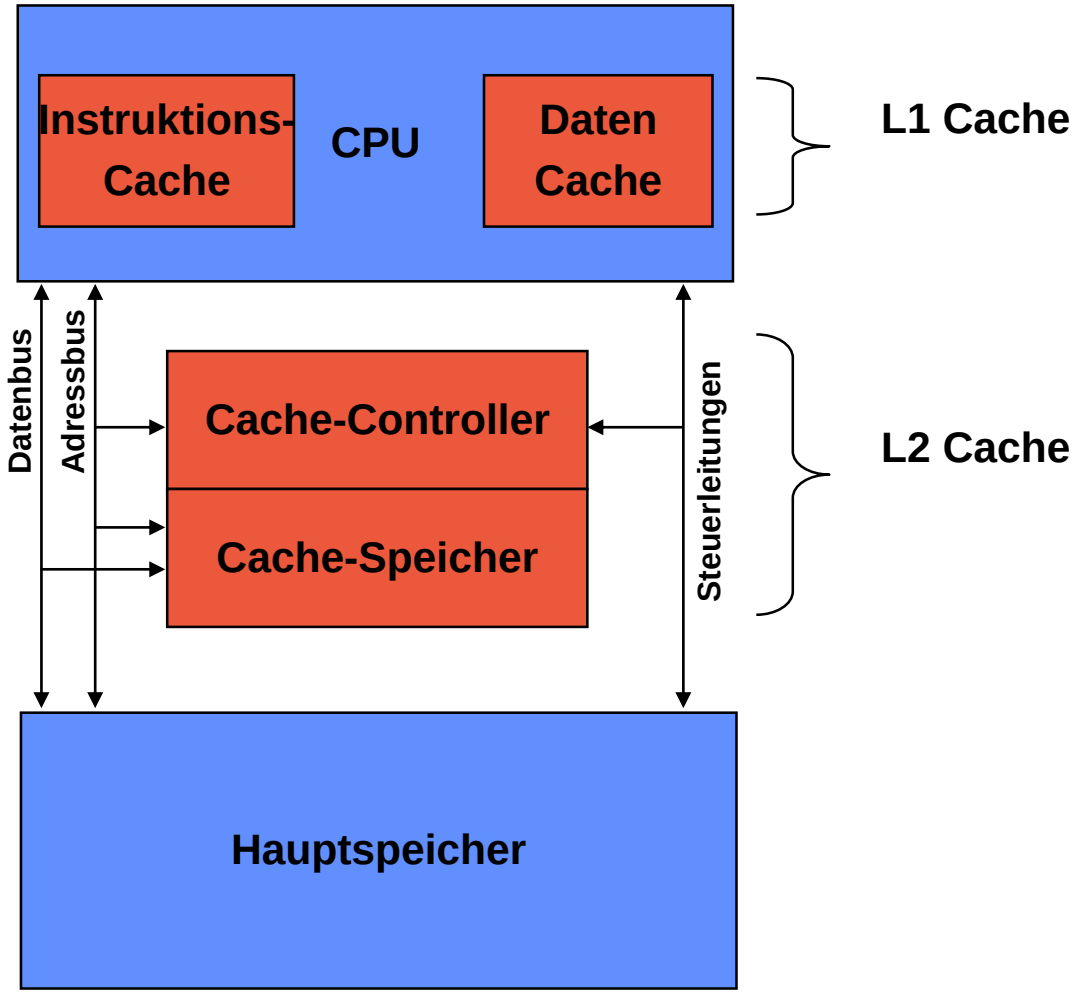
Cache

- Schnell
- Entlastet den langsamen Systembus
- Aufbau
  - Blockweises abspeichern von Daten und Adressen
  - Halten der Adressen und der Veränderung (Dirty-Bit) im Cache-Controller
- cache-hit
  - Datum liegt im Cache
- cache-miss
  - Datum liegt nur im Hauptspeicher

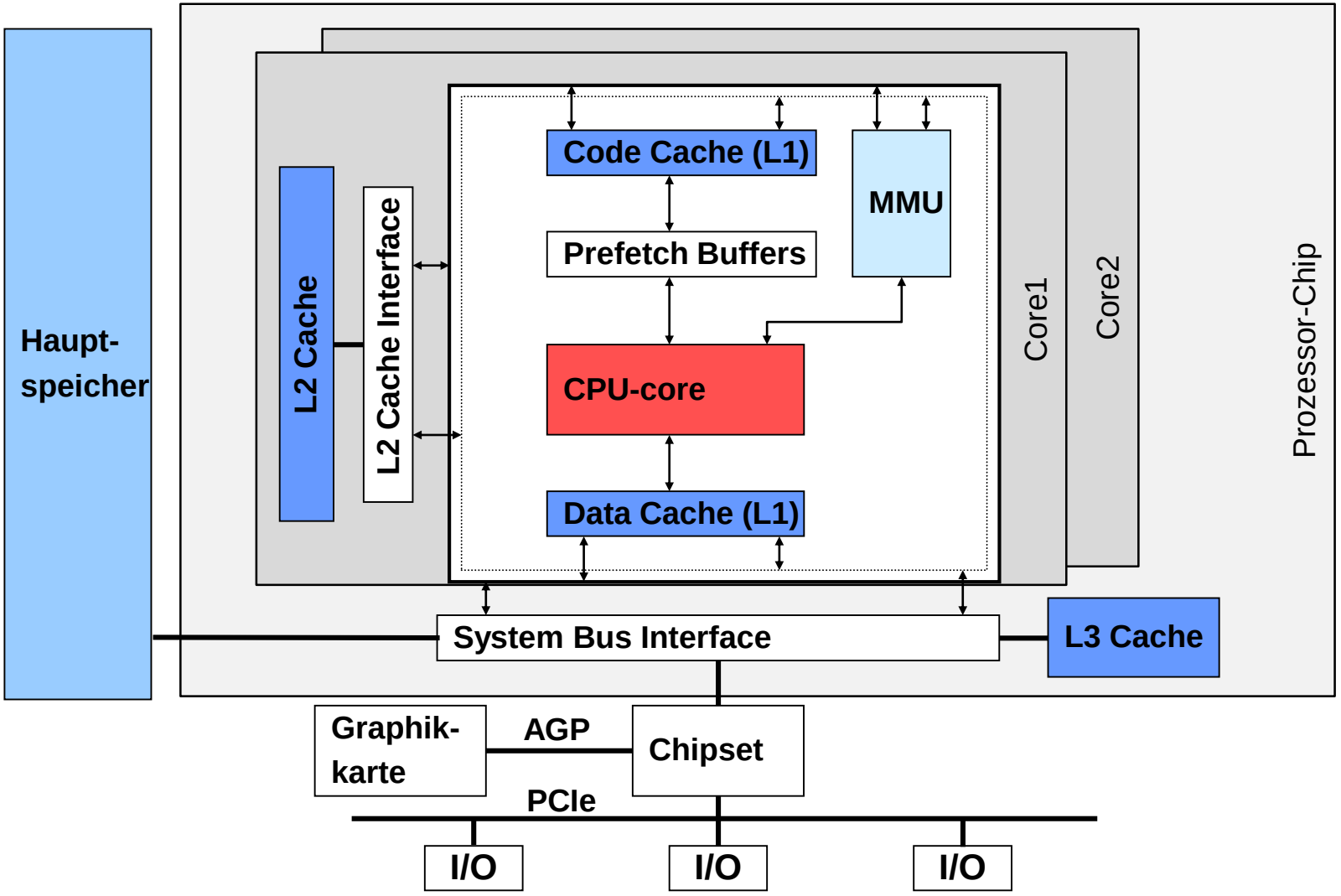


L1/L2-Cache

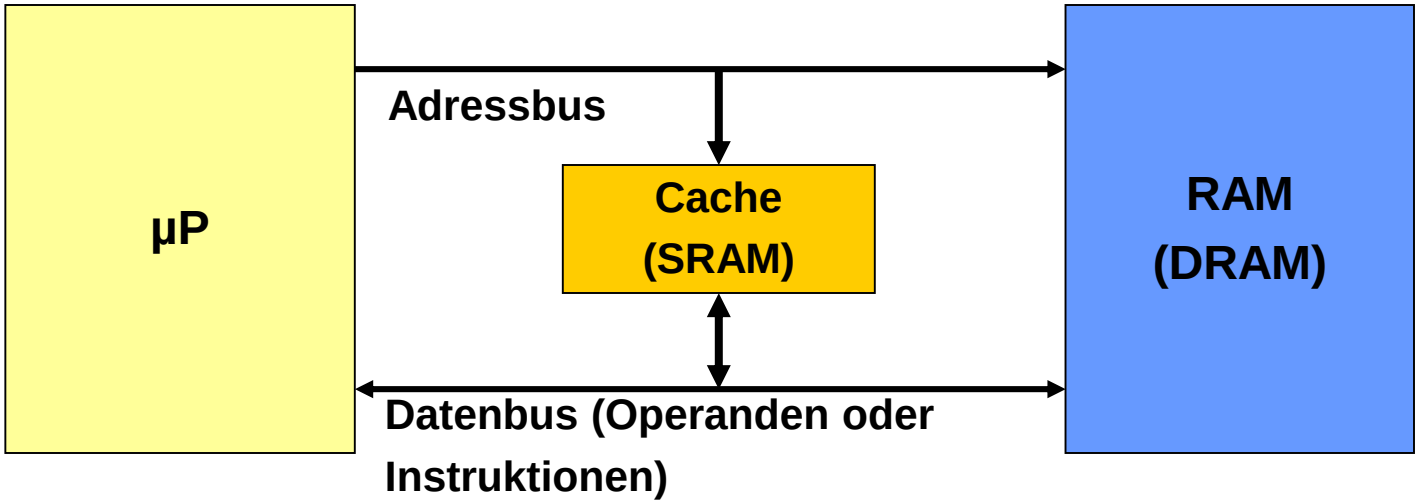
- **Weitere Verbesserung:**
- **Level 1 Cache:**
  Getrennte Caches für Daten und Instruktionen (Harvard Architektur)
- **Level 2 Cache:**
  Langsamer, größer und gemeinsamer Inhalt



Genauere Sicht auf ein PC-basiertes System

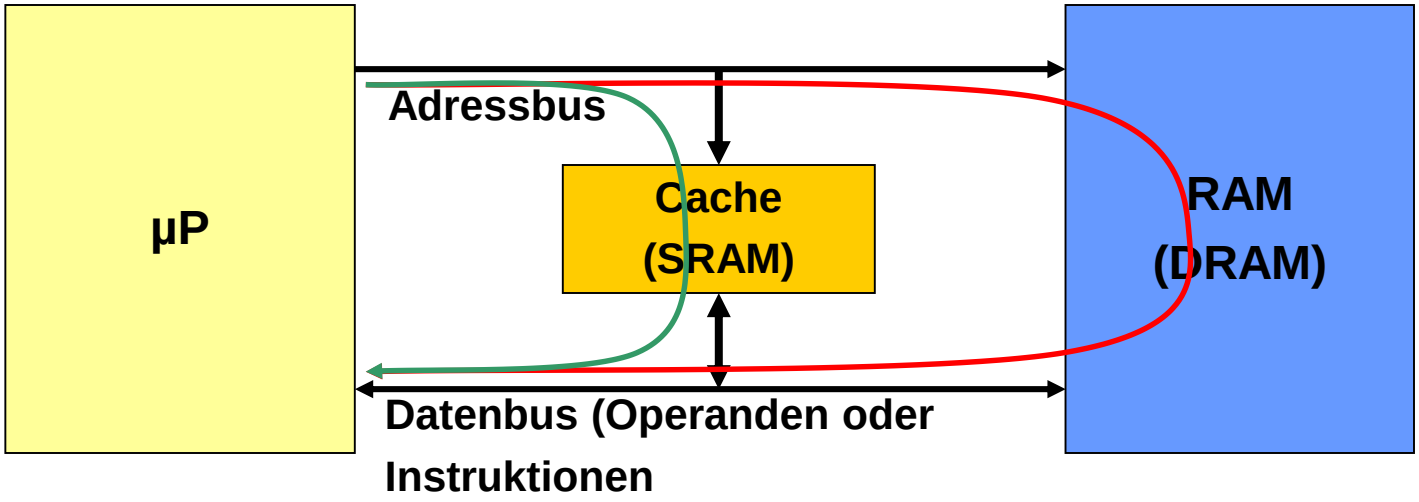


Vereinfachte Sicht: Nur 1 Cache zwischen Main Memory (RAM) und CPU (µP)



- Der Cache kann als Bypass für einen schnellen Zugriff angesehen werden.

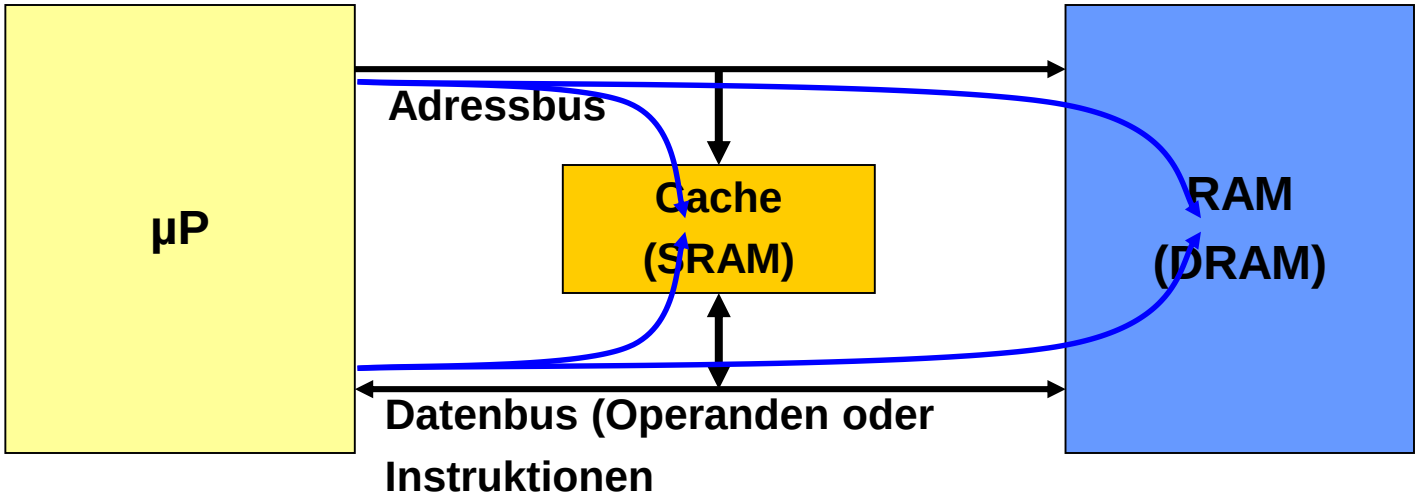
Cache Lesezugriff



- Ein Lesezugriff versucht die Daten im Cache zu finden. Wenn sie nicht vorhanden sind, (Cache Miss) wird auf den Hauptspeicher zugegriffen.

Cache Schreibzugriff

Write Through Strategie

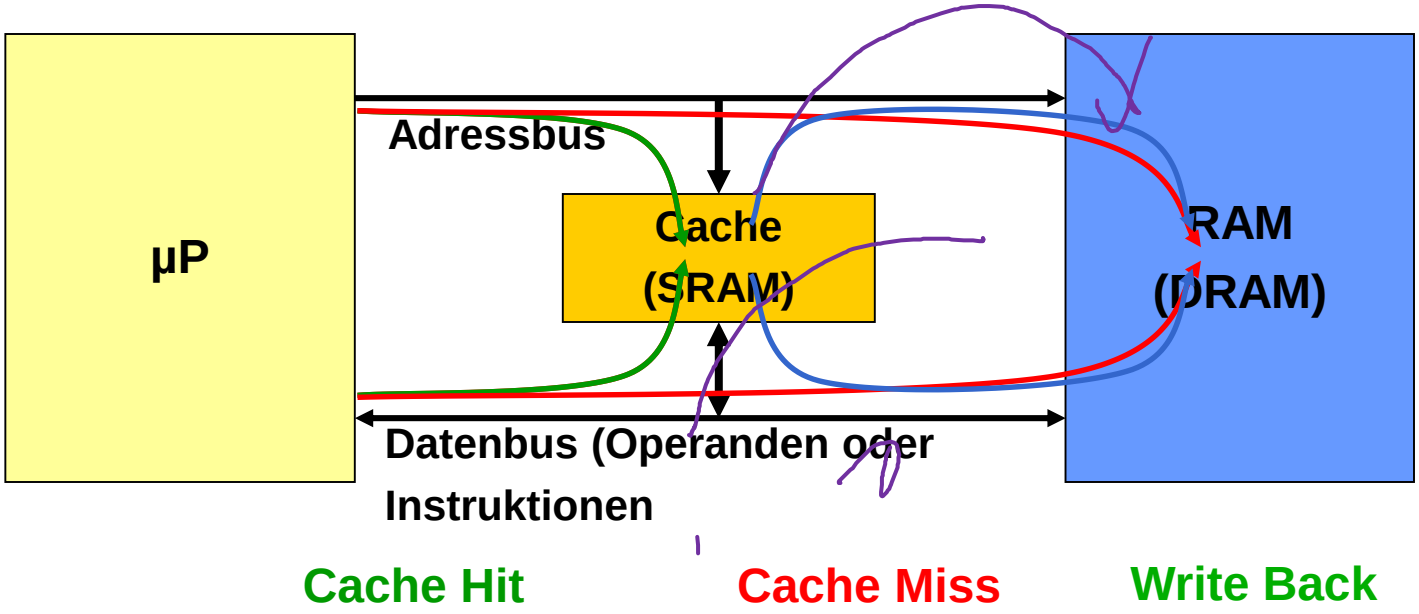


Cache Hit **Cache Miss**

- Eine Schreiboperation nach der Write Trough Strategie schreibt immer sowohl in den Cache als auch in den Hauptspeicher, egal ob ein Cache Hit oder ein Cache Miss vorliegt.
  => langsam, aber starke Kohärenz

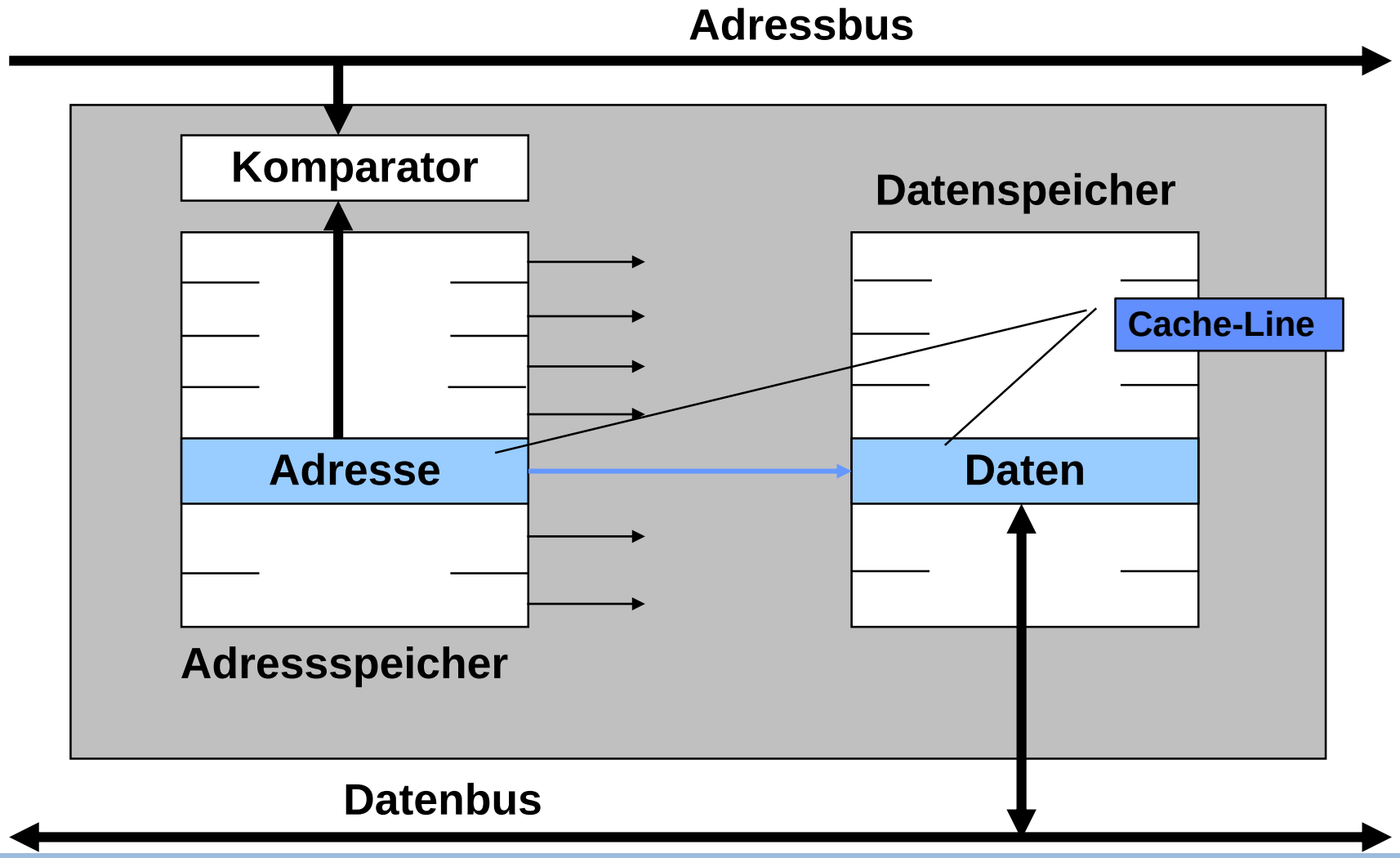
Cache Schreibzugriff II

Write Back Strategie



- Eine Schreiboperation nach der Write Back Strategie schreibt bei einem Cache Hit nur in den Cache. Später werden die modifizierten Daten in den RAM zurückgeschrieben.
  => schnell, schwache Kohärenz

Aufbau eines Caches



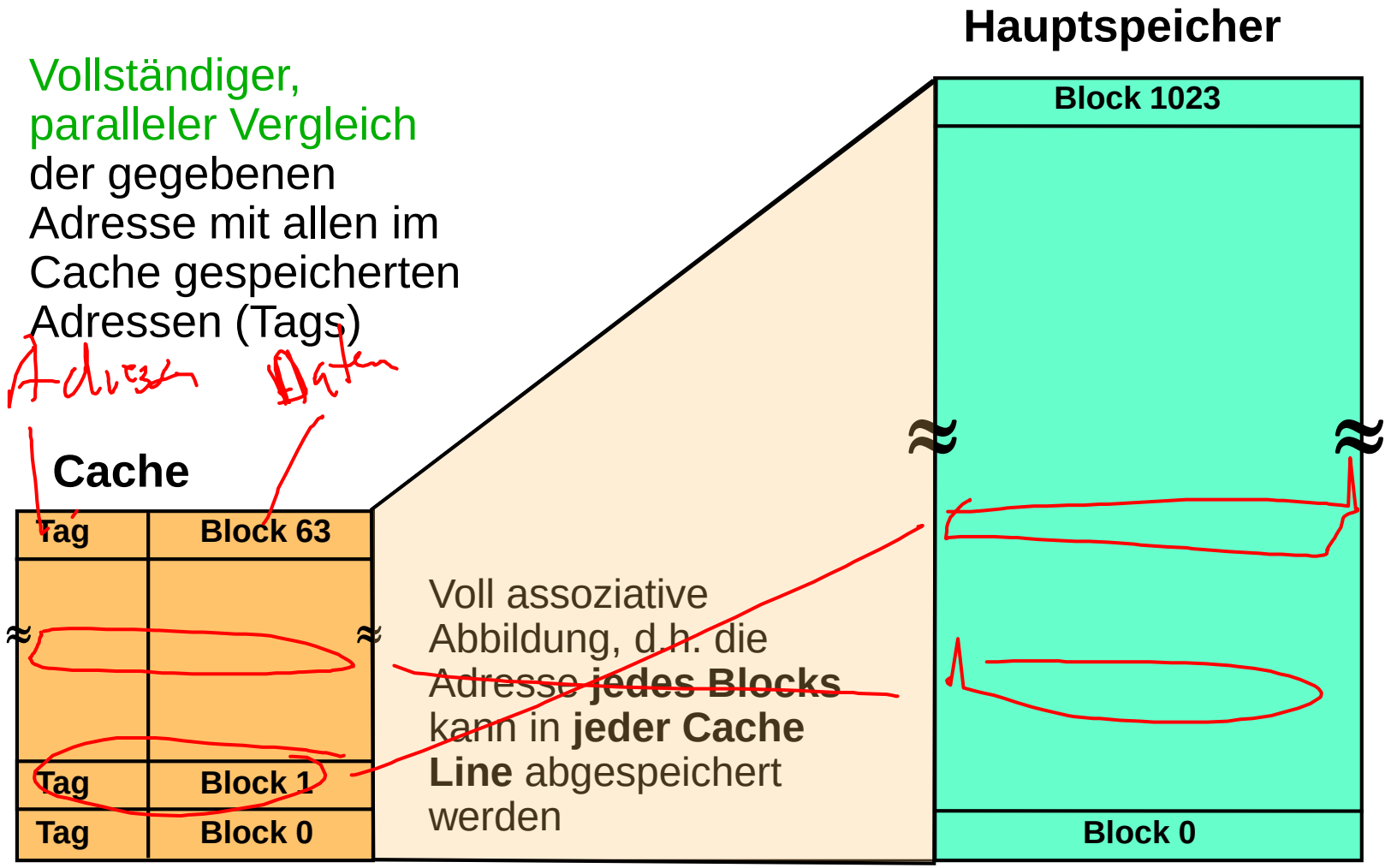
Aufbau eines Caches

- Ein Cache besteht aus einem Daten- und Adressspeicher.
- Der Speicher ist in Blöcken, sogennanten Cache-Lines aufgeteilt.
- Der Adressspeicher enthält die Hauptspeicheradressen der Daten, die im Cache gespeichert sind.
- Ein Vergleich prüft bei einem Hauptspeicherzugriff, ob die Adressen auf dem Adressbus einer gespeicherten Adresse entsprechen.
- Diese Vergleich muss mit allen Adressen gleichzeitig und sehr schnell geschehen, sonst ist der Cache ineffektiv.
- Ein Valid-Bit (V) gibt an ob die Daten in der Cacheline überhaupt gültig sind oder noch gar nicht geladen wurden.
- Ein Dirty-Bit (D) gibt an, ob bei einer Write-Back Strategie die Daten verändert wurden und noch nicht wieder in den Hauptspeicher geschrieben wurden.

Cache-Typen

- Unterschiedliche Varianten zum Vergleich der Adressen existieren, die unterschiedlichen Hardwareaufwand verbrauchen und dabei verschieden starke Leistung versprechen:
  - Fully Associative Cache
  - Direct Mapped Cache
  - n-Way Set Associative Cache

Fully Associative Cache: Prinzip

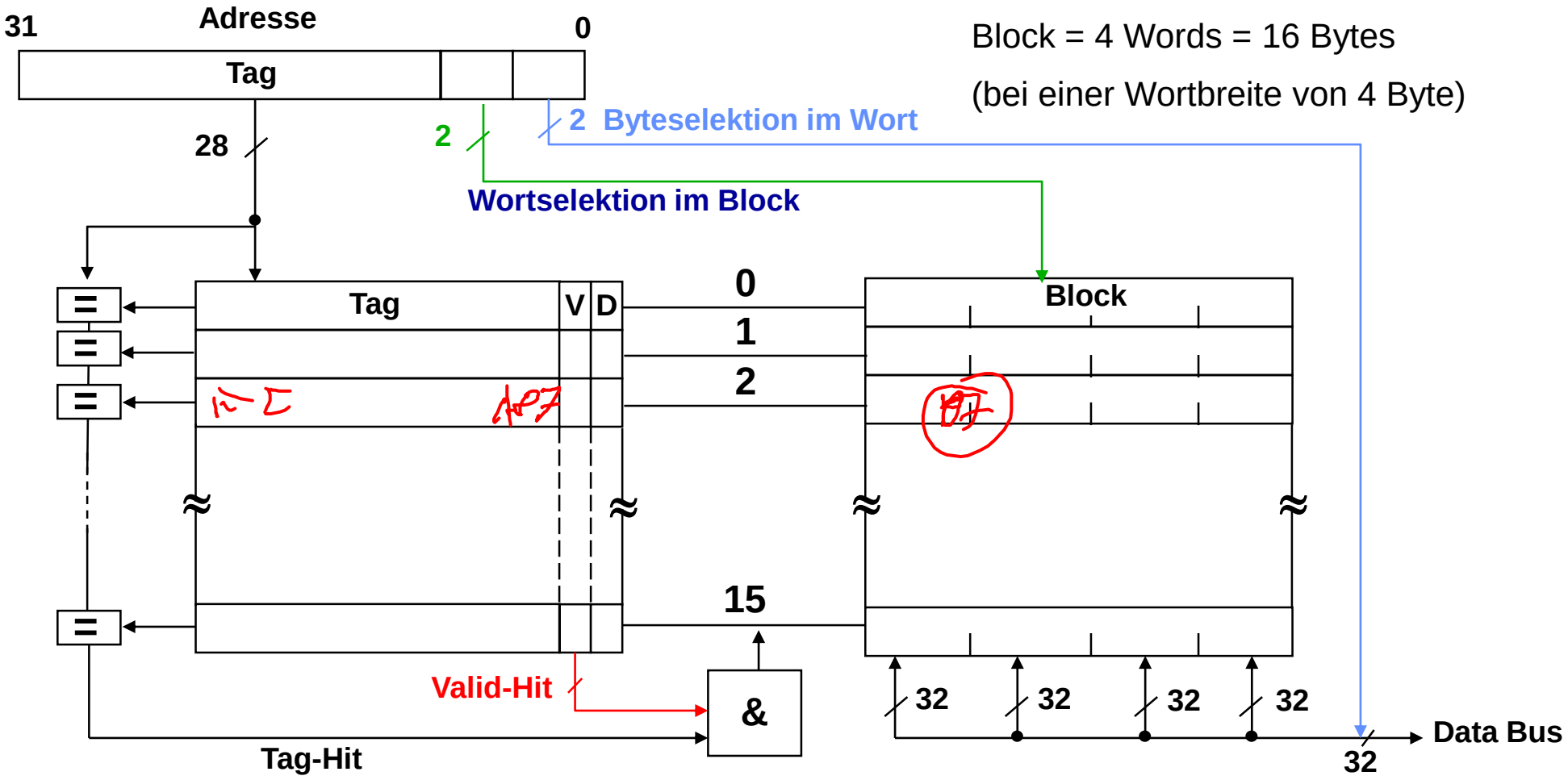


Fully Associative Cache: Aufbau

Kapazität: 256 Byte

Block = 4 Words = 16 Bytes

(bei einer Wortbreite von 4 Byte)



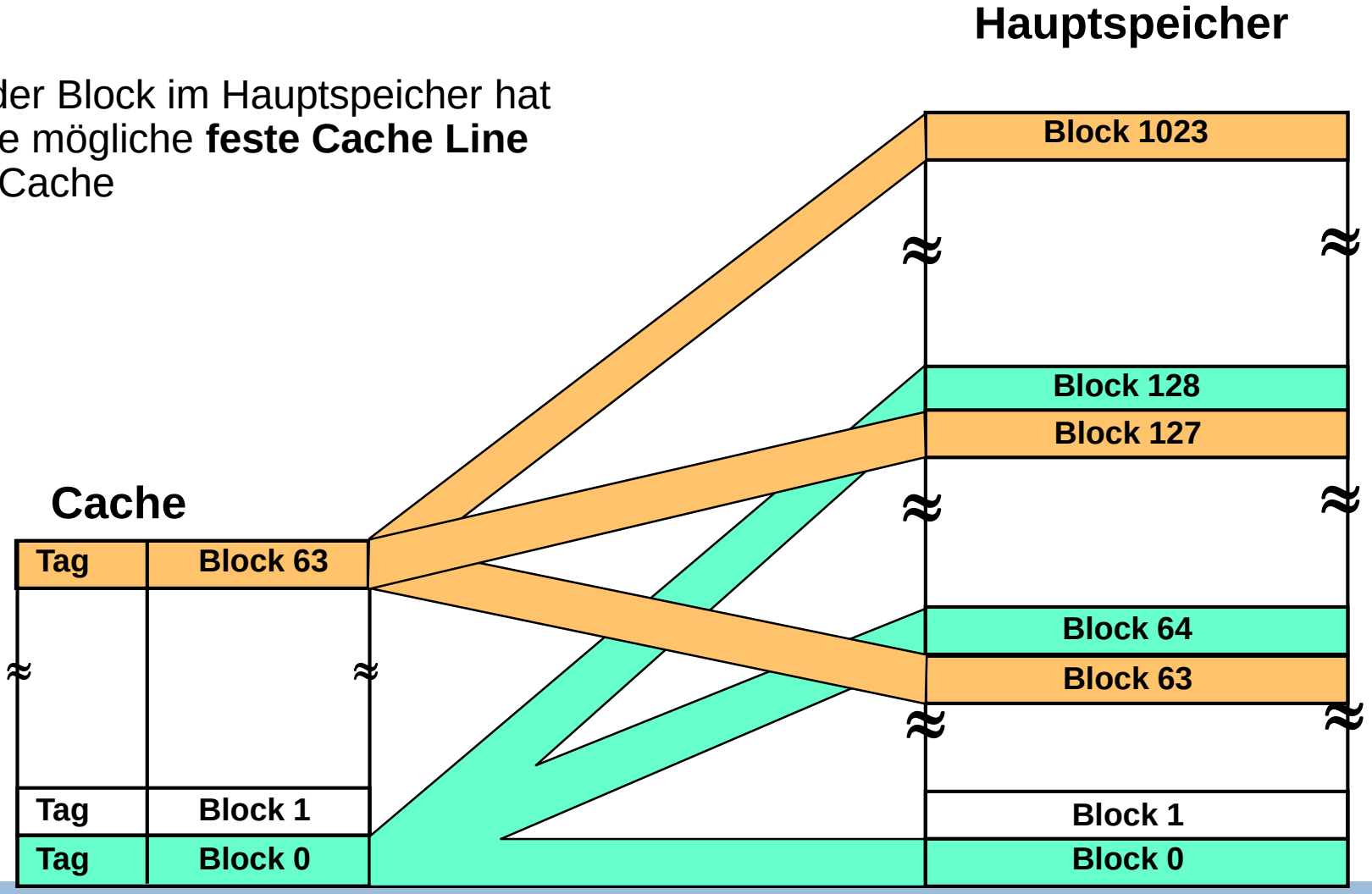
Fully Associative Cache: Eigenschaften

- **Vorteile:**
  - Daten von jeder Speicherstelle können beliebig in den Cache-Lines abgelegt werden
  - Optimale Cache-Nutzung
  - Freie Wahl der Cache Ersetzungsstrategie

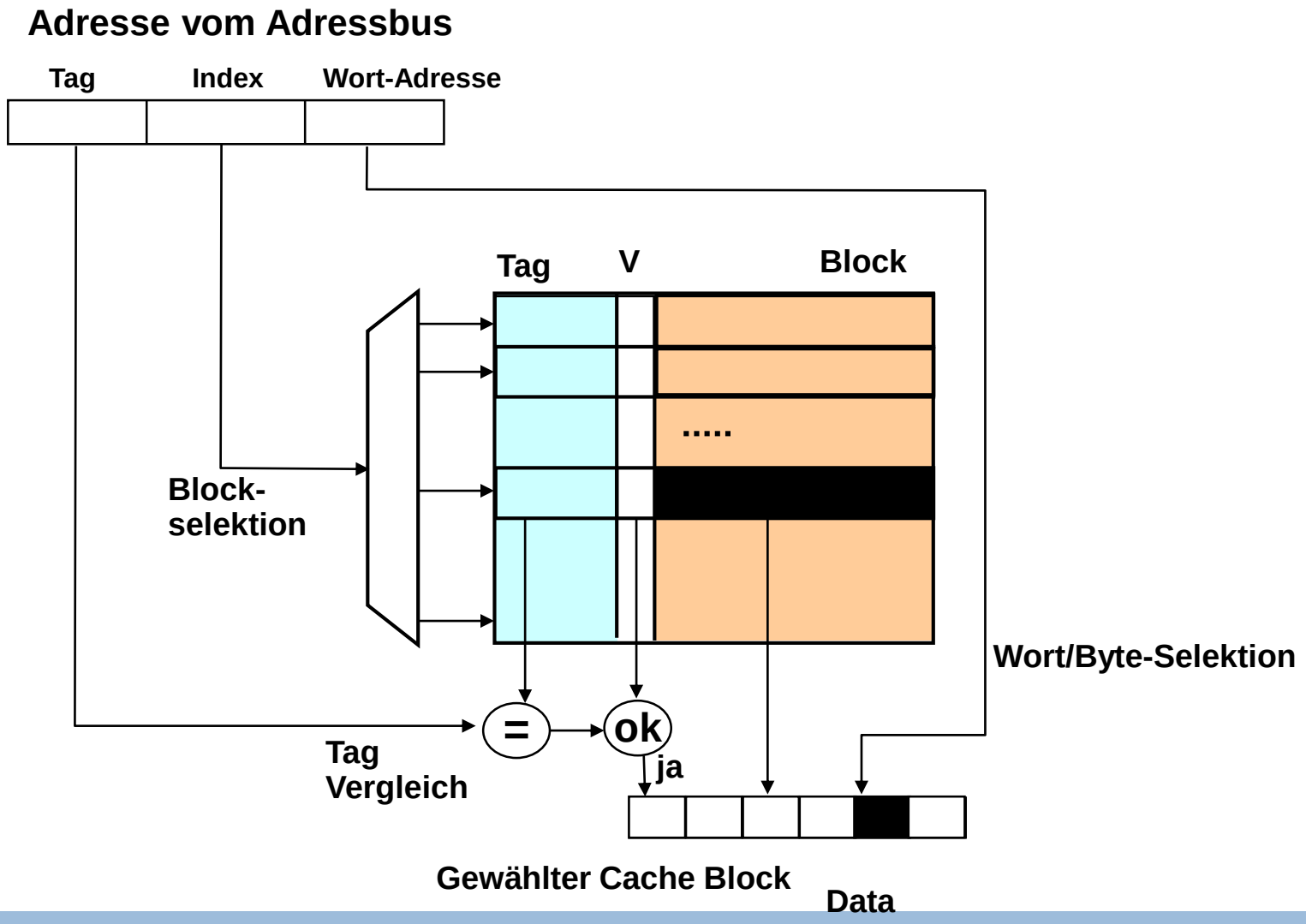
- **Nachteile:**
  - Hoher Hardware Aufwand, da viele Komparatoren
    ➔ nur kleine Caches können realisiert werden.
  - Die flexible Datenspeicherung erfordert mehr Hardwareaufwand bei der Implementierung der Ersetzungsstrategie

Direct Mapped Cache: Prinzip

Jeder Block im Hauptspeicher hat eine mögliche **feste Cache Line** im Cache



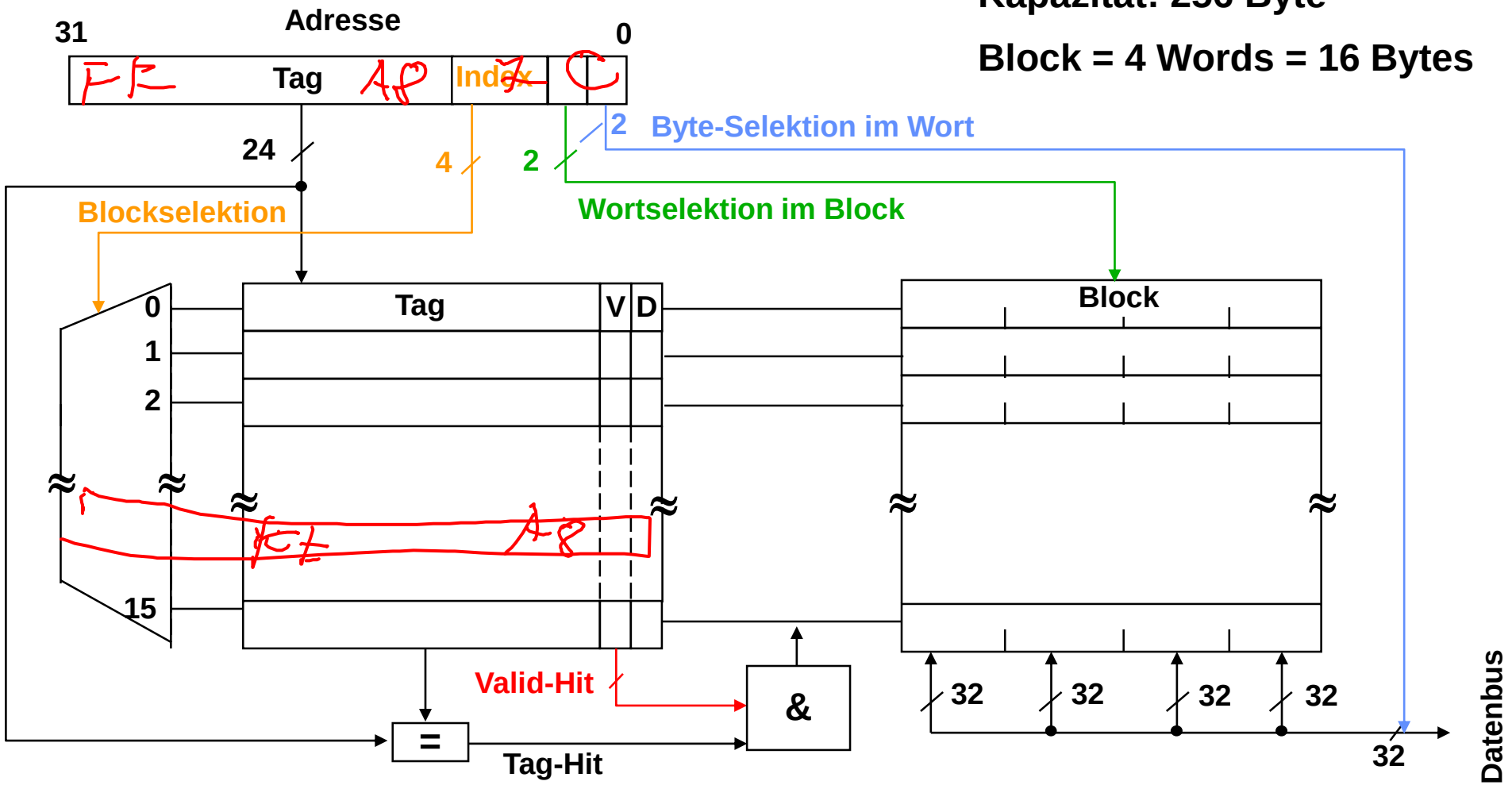
Direct Mapped Cache: Aufbau



Direct Mapped Cache: Aufbau detailliert

Kapazität: 256 Byte

Block = 4 Words = 16 Bytes



Direct Mapped Cache: Eigenschaften

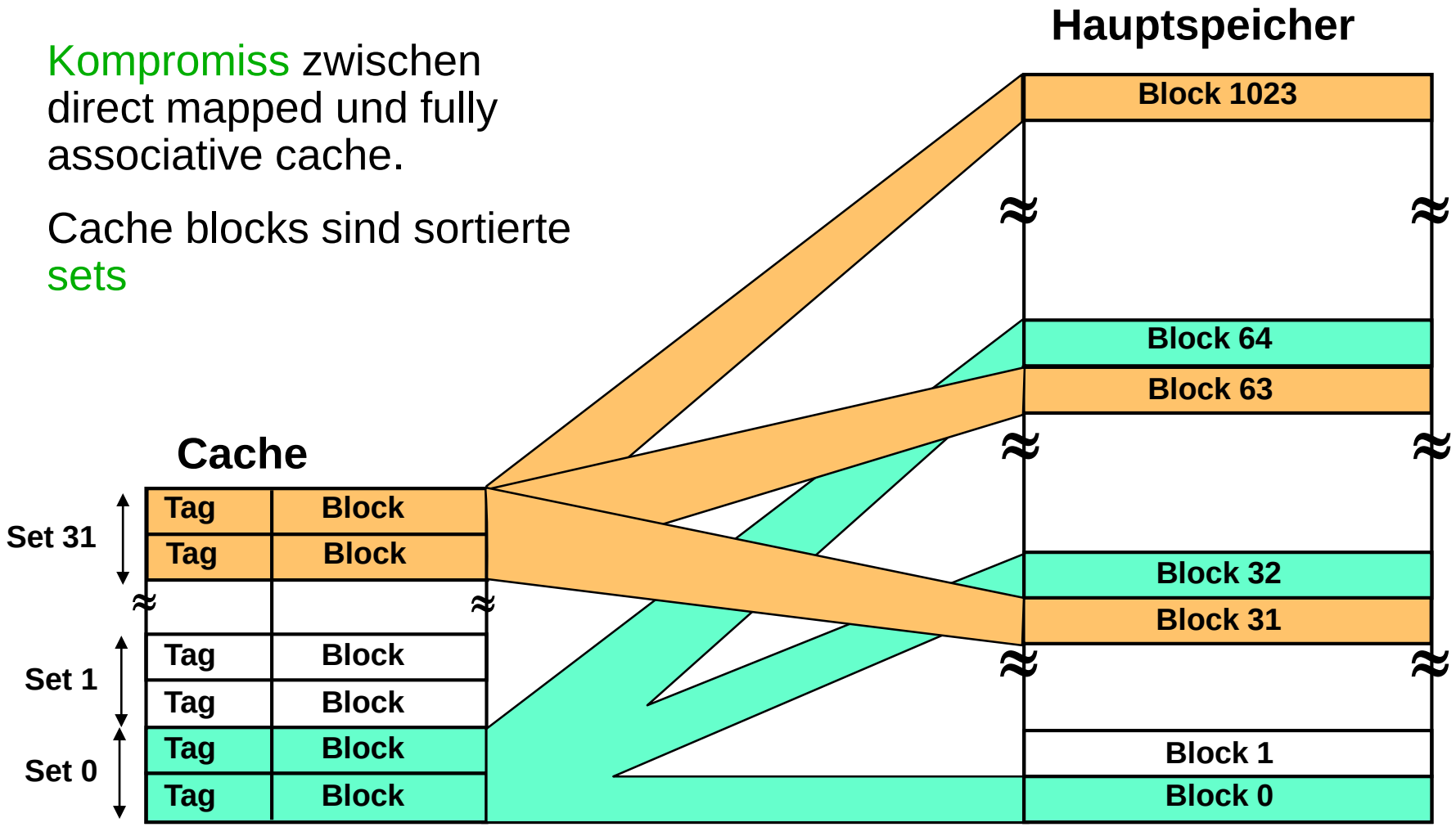
- **Vorteile:**
  - Wenig Hardwardwareaufwand, nur ein Komparator.
  - Schneller Zugriff, da Tag und Datenblock parallel gelesen werden können.
  - Keine Wahl der Cache Ersetzungsstrategie notwendig.

- **Nachteile:**
  - Ständiger Kampf um freie Cache-Lines, .auch wenn noch Platz im Cache wäre, da ungenutzte Blöcke nicht beliebig verwendet werden können.
  - Damit könnten bei Zugriffen z.B. auf Blöcke 64,128 immer abwechselnd die Cache-Lines gefüllt werden => sehr ineffizient.

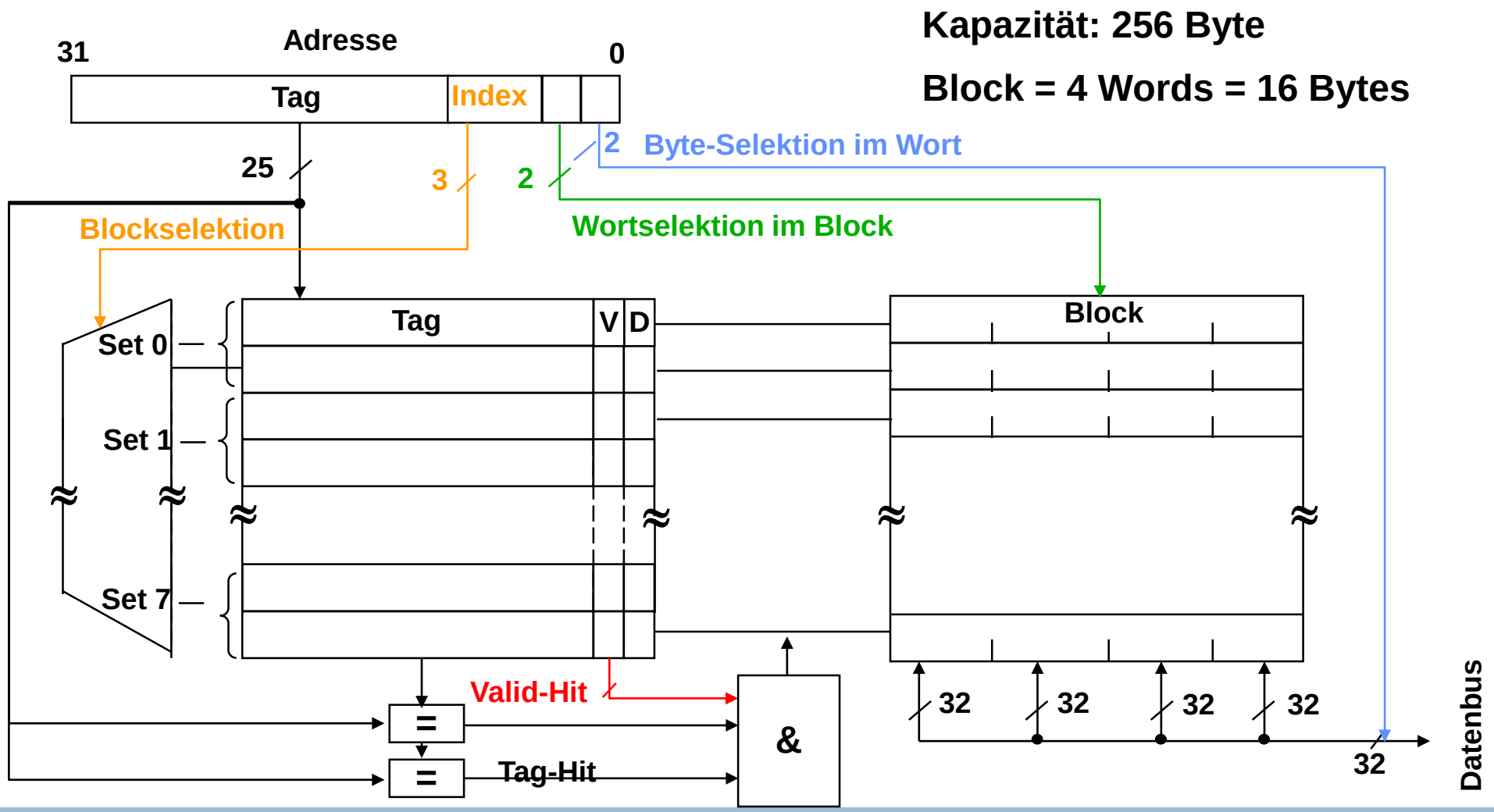
n-Way Set Associative Cache: Prinzip

Kompromiss zwischen direct mapped und fully associative cache.

Cache blocks sind sortierte sets



2-Way Set Associative Cache: Aufbau



n-Way Assiociative Cache: Eigenschaften

- **Vorteile:**
  - Mittlerer Hardwardwareaufwand, nur n Komparatoren.
  - Schneller Zugriff, da Tag und Datenblock parallel gelesen werden können.
  - Abwechselnde Zugriff sind erst bei n-Zyklen problematisch

- **Nachteile:**
  - Teilweise Kampf um freie Cache-Lines, .auch wenn noch Platz im Cache wäre, da ungenutzte Blöcke nicht beliebig verwendet werden können.

Cache: Kenndaten

c = Cache Kapazität (# lines)

s = Anzahl von Sets

n = Größe eines Sets

Es gilt immer c=s*n

- **Direct mapped cache: n = 1, s = c.**
  Ein Block kann nur an einer einzig möglichen Position gespeichert werden
- **full associative cache: n = c, s = 1**
  Ein Block kann an jeder Stelle im Cache gespeichert werden.
- **n-way set associative cache: n = #, s = c/n**
  Ein Block block or line can be stored at n different places (positions) in the cache. This is called a set.

Verhalten eines Caches

- Cache Hit: Der gesuchte Datenblock ist im Cache
- Cache Miss: Der gesuchte Datenblock ist nicht im Cache

- Gründe für Cache Misses:

Typ des Cache Misses	Grund	Strategie zur Behebung
Kaltstart	Erste Verwendung des Blocks	-
Kapazität zu klein	Cache Kapazität zu klein	Cache mit höherer Kapazität
Konflikt	Die Größe eines Sets ist zu klein	Höhere Assoziativität

Speicherzugriffszeiten

- **Level 1 (L1) Cache**

  $T_{access\ (average)} = T_{access} + q_{Cache\ Miss} \cdot T_p$

  T_{access} = Cache Zugriffszeit

  q = Wahrscheinlichkeit eines Cache misses

  T_p = zusätzliche Zeitverzögerung für einen Cache miss (z.B. zus. Takte * Taktperiode)

- **Level 2 (L2) Cache**

  $T_{access\ (average)} = T_{access\ L1} + q_{Cache\ Miss\ L1} (T_{access\ L2} + q_{Cache\ Miss\ L2} \cdot T_{p_{L2}})$

- **… und so weiter für L3, L4**

Caches in Intel-Architekturen:

- **Intel 64 and IA-32 Architekturen:**

Level	Size	Latency	Index Bits	Sets	Associativity	Line Size
L1 Data	32kB	4	6	64	8	64B
L1 Instr.	32kB	4	6	64	8	64B
L2	256kB	7	9	512	8	64B
L3	>3MB	26-31	11	4096	12-20	64B

[Yarom, Ge, Liu, Lee, Heiser,2015]

HS

SJ ~700

Ersetzungsstrategien:

- **FIFO (First In First Out)**
  Der jeweils älteste Eintrag wird verdrängt.
  - Einfach
  - Nicht gut bei wiederkehrenden Zugriffen
- **LRU (Least Recently Used)**
  Der Eintrag, auf den am längsten nicht zugegriffen wurde, wird verdrängt.

- **LFU (Least Frequently Used)**
  Der Eintrag, auf den wenigsten häufig zugegriffen wurde, wird verdrängt.

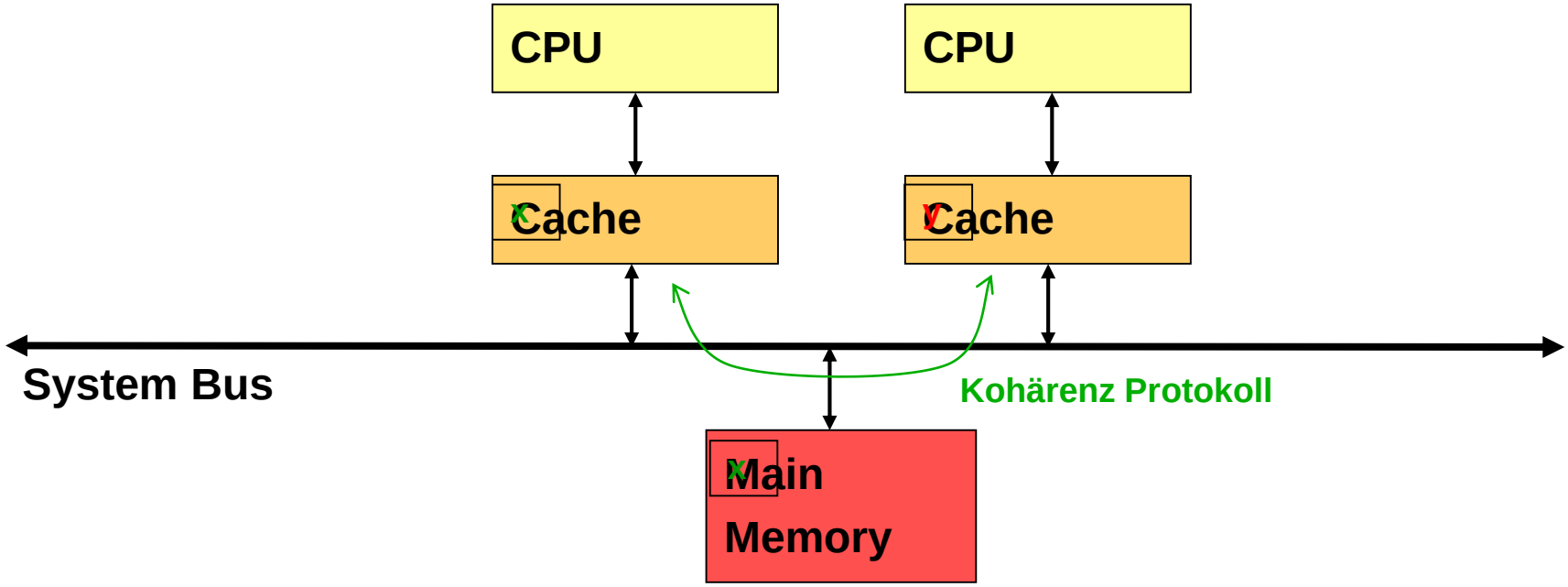
Beispiel Cache

```
#define MSIZE 10000
int i,k,n;
double **m=new double*[MSIZE];
for(int ii= 0; ii < MSIZE; ++ii)
    m[ii] = new double[MSIZE];
for (i=0;i<MSIZE;i++) {
   for (k=0;k<MSIZE;k++) {
     //m[i][k]=i+k; // 24% Cache miss 0.084u seconds time
     m[k][i]=i+k; // 98,5% Cache miss 1.348u seconds time
   }
 }
```

- Größe der Matrix=8*10000*10000=800 MB, passt in keinen Cache
- Cache miss rate beim Schreiben laut valgrind 24% bei *gutem* Zugriff, sonst 98,5%

Cache Kohärenz in Multiprozessorsystemen

- Eine volle Kohärenz zwischen den Caches der Prozessoren ist schwer herzustellen
- Dazu gibt es Kohärenzprotokolle.
- Dies sind in der Lage die Kohärenz sicherzustellen
  => Vorlesung **“Computer-Architectures”**

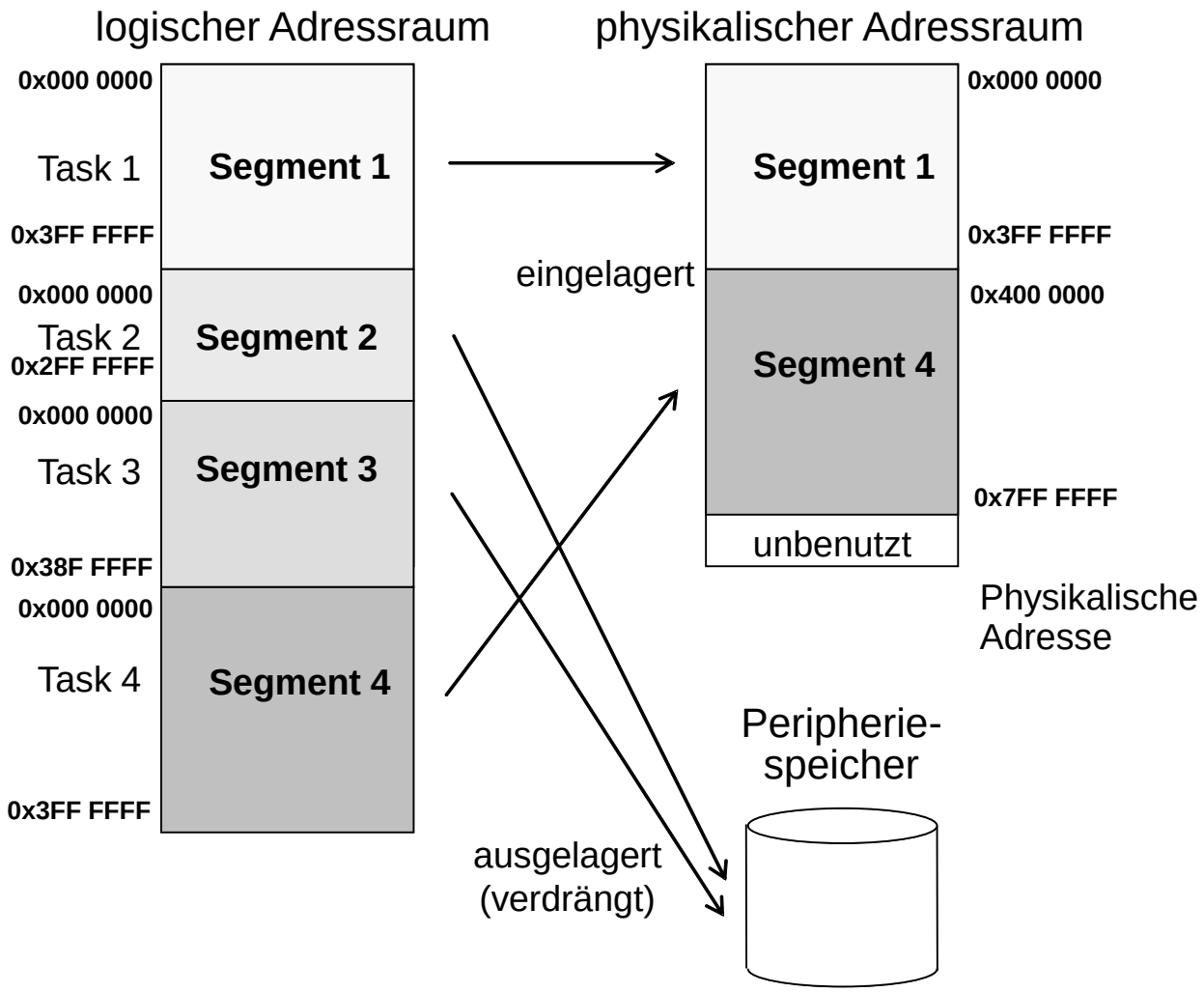


Virtueller Speicher : Segmentierung

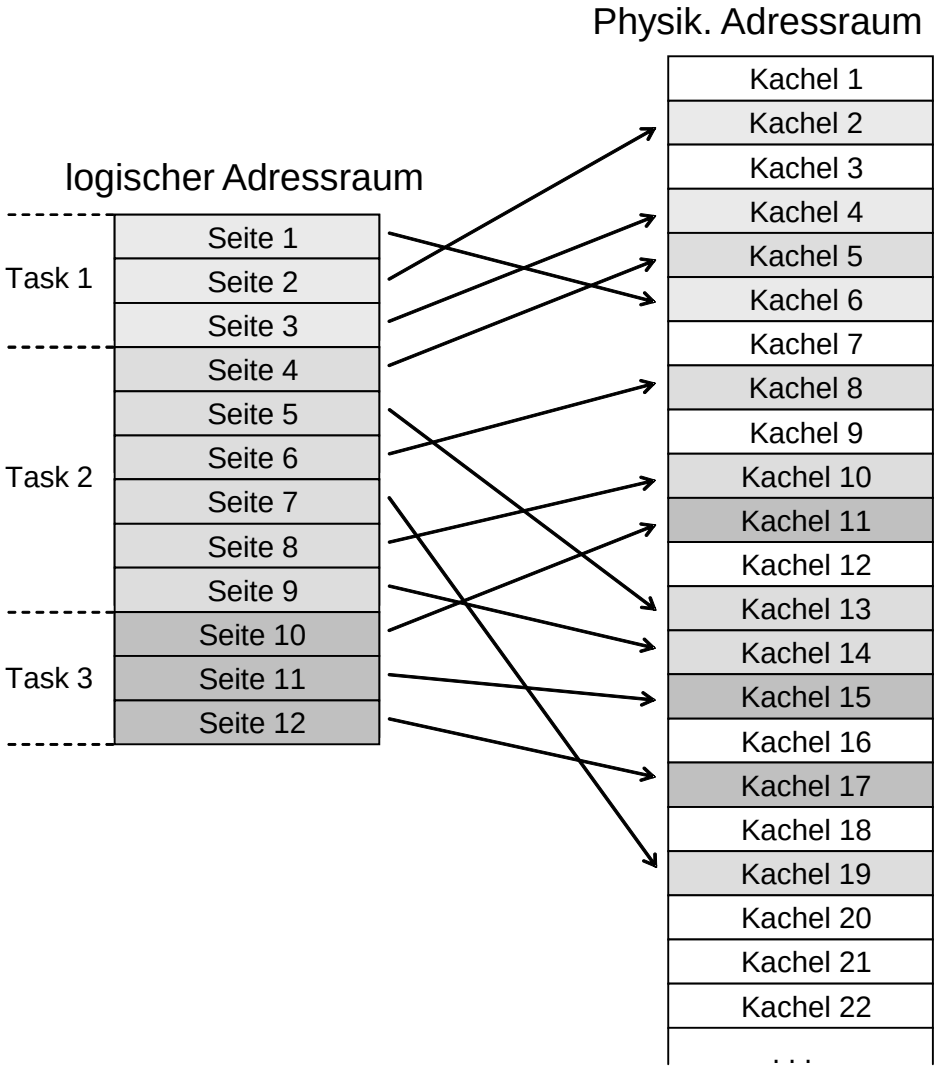
Reicht der Hauptspeicher nicht aus, kann auf die Festplatte ausgelagert werden.

Diesen Vorgang bezeichnet man als Segmentierung oder Paging.

Damit ist es möglich einen sehr großen „virtuellen“ Adressraum zu bekommen: (z.B. 4 GByte bei 32 Bit Adressen).



Virtueller Speicher : Paging



Die Zuordnung von Seiten zu Kacheln erfolgt in der Regel ohne Beachtung der sequentiellen Reihenfolge der Seiten

Vorteile:

- einfache Zuweisung,
- dynamische Taskgrößenänd. einfacher
- keine Speicherbereinigung

Nachteile:

- letzte Seite nur teilweise voll
- höherer Seitenverwaltungsaufwand
- durch kleine Seiten mehr Datentransfers zwischen Haupt- und Peripheriespeicher