Karlsruher Institut für Technologie

Institut für Technik der Informationsverarbeitung

Prof. Dr.-Ing. Dr. h. c. J. Becker

becker@kit.edu

Karlsruher Institut für Technologie (**KIT**)

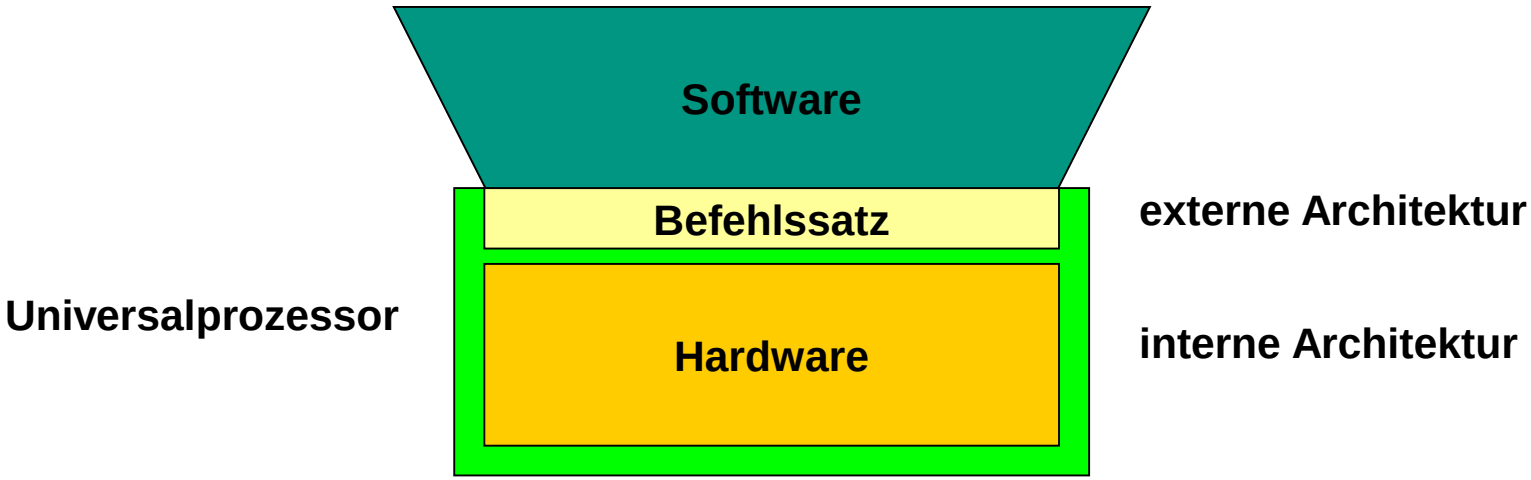
Institut für Technik der Informationsverarbeitung (ITIV)

Digitaltechnik

Rechnersysteme (Einführung)

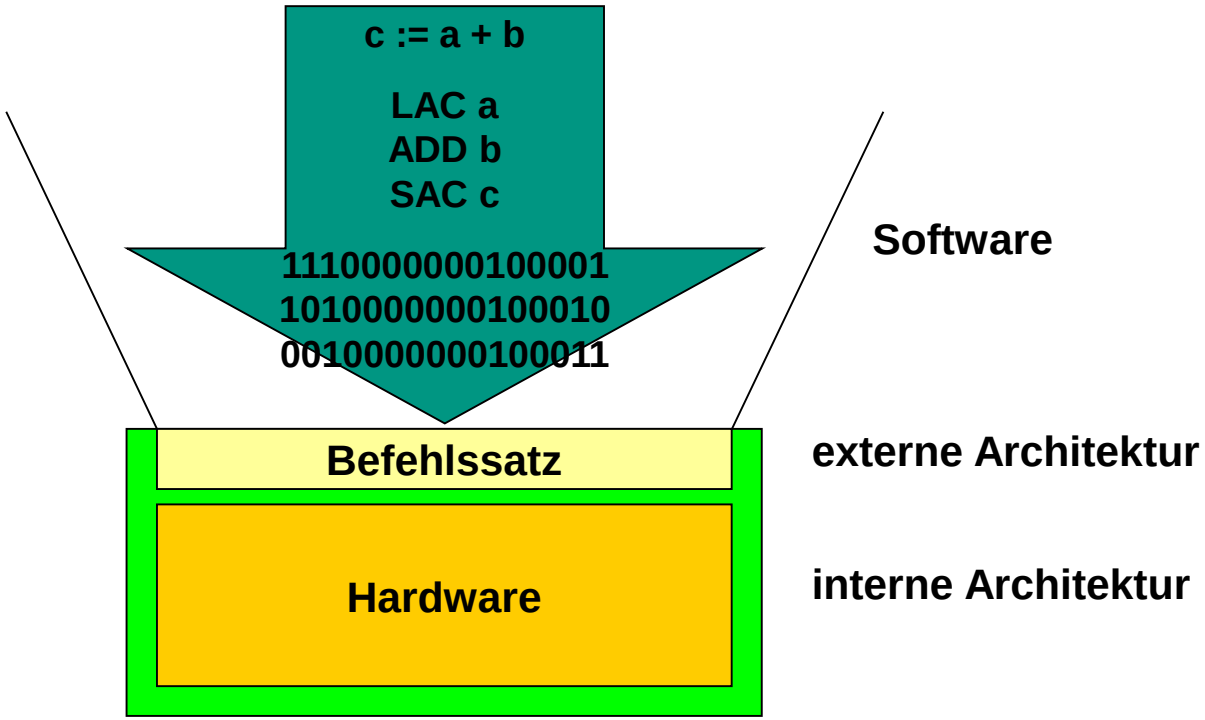
Rechnerarchitekturen

- externe Architektur: Maschinenbefehlssatz
- interne Architektur: der innere Hardwareaufbau eines Rechners
- Universalprozessoren: für alle Anwendungen von einfachem C-Programm bis zum Compiler, Betriebssystem, Datenbank, Textverarbeitung



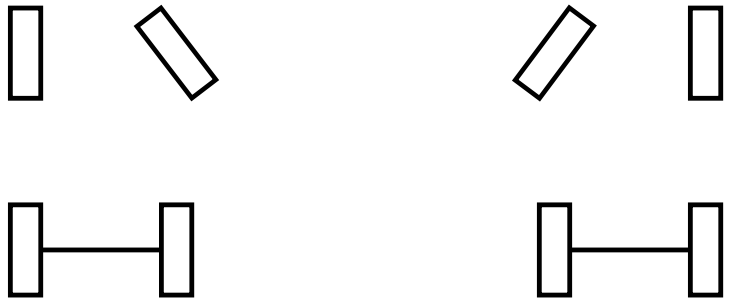
Rechnerarchitekturen

– Gesteuert wird die Hardware mit einem Programm in einer höheren Programmiersprache / Assembler / Maschinensprache

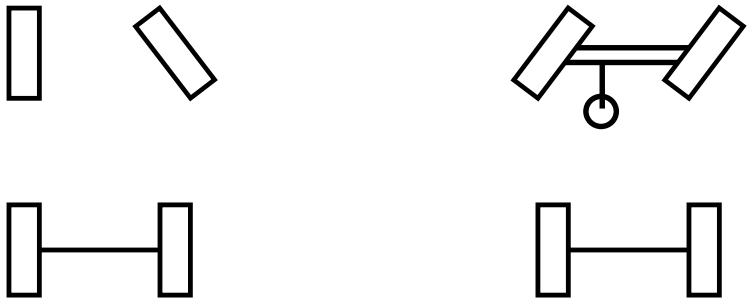


Rechnerarchitekturen

- Analogie (Parnas/Siewiorek): der Maschinenbefehlssatz entspricht der Lenkung eines Autos
  - bei starrer Hinterachse und beweglich aufgehängten Vorderrädern gibt es für die Vorderräder viele Möglichkeiten ... (die freien Schaltmöglichkeiten eines Rechensystems)

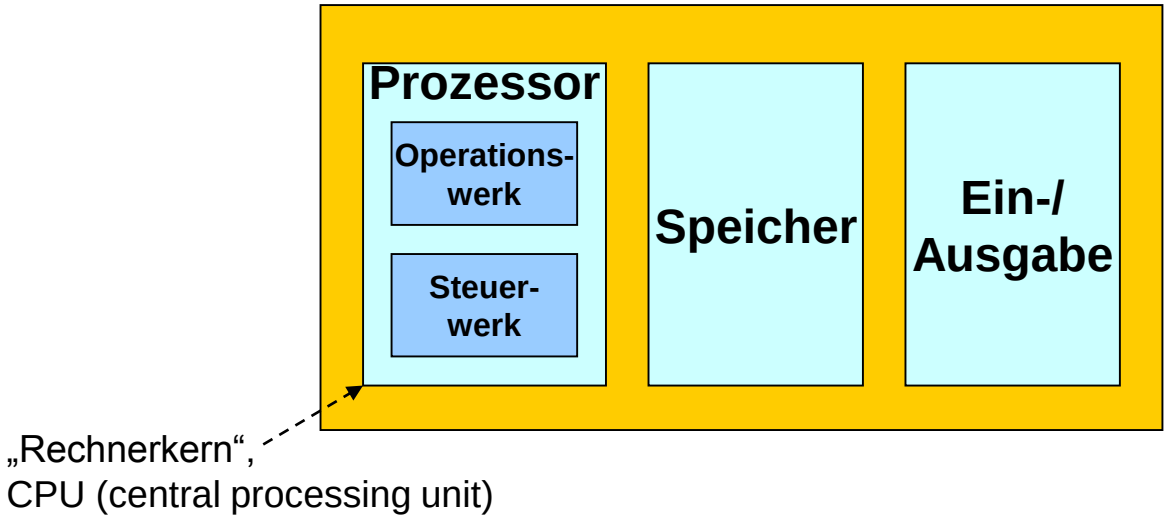


  - durch die Lenkung sind nur noch sinnvolle Radstellungen benutzbar

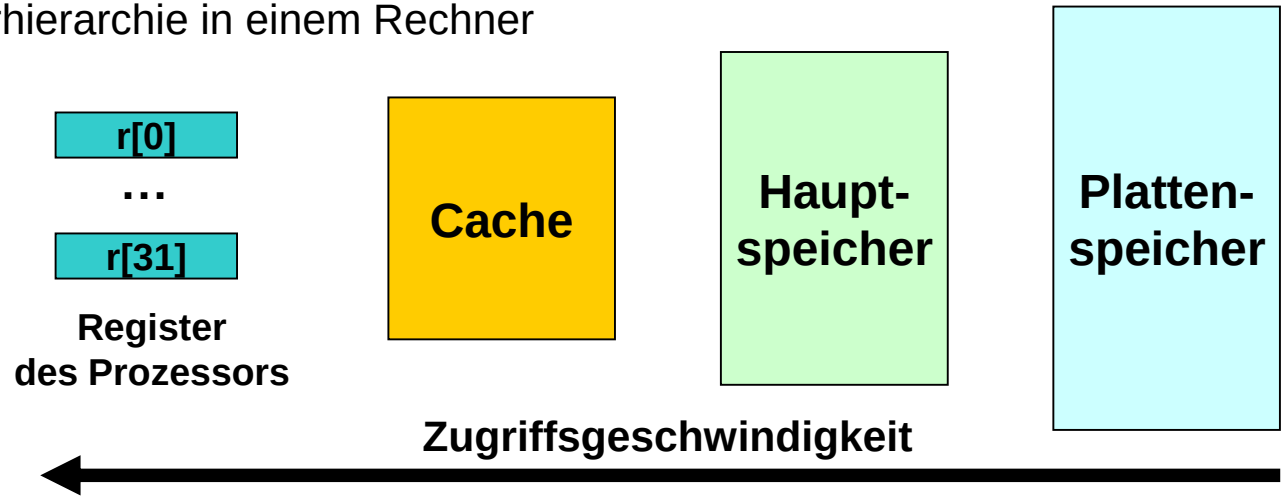


Rechnerarchitekturen

- Interne Architektur: Bestandteile

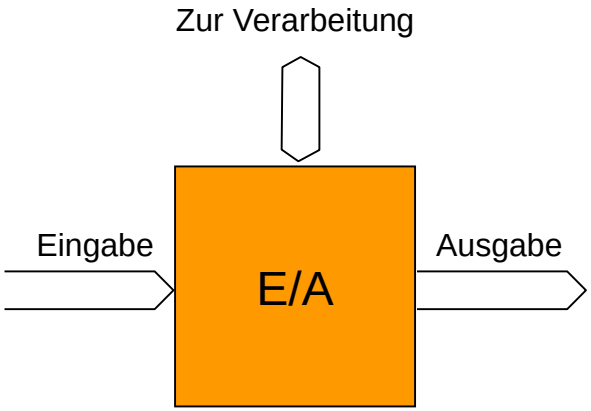


- Speicherhierarchie in einem Rechner



Ein-/Ausgabewerk

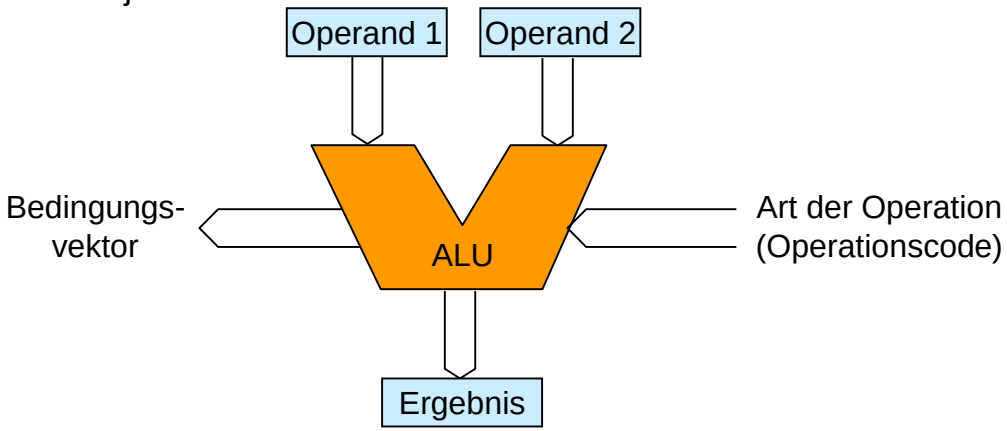
- Kommunikation mit dem Benutzer
- Eingabe:
  - Übernahme von Daten von extern angeschlossenen Geräten
  - Umsetzung in eine geeignete, normierte Darstellung zur weiteren Verarbeitung



- Ausgabe:
  - Datenformatmäßige und elektrische Aufbereitung der Daten
  - Weiterleitung an externe Ausgabegeräte
  - Eventuell Auswahl des Ausgabegerätes

Operationswerk (Arithmetisch/Logische Einheit ALU)

- Verarbeitung der Daten
- Stellt einen Satz von Basisfunktionen aus dem Bereich der numerischen und logischen Verknüpfungen zur Verfügung
- Beispiel für ein einfaches Rechenwerk:
  - Addition
  - Komplementbildung
  - Konjunktion und Diskjunktion
  - Negation



- Üblicherweise Beschränkung auf Funktionen mit zwei Operanden bei gegebener maximalen Stellenzahl → Genauigkeit Verarbeitung der Daten
- Einzelne Merkmale eines Operanden oder Ergebnisses (z.B. Wert gleich Null) können als Binärwerte in einem sogenannten Bedingungsvektor („flags“) zur Verfügung gestellt werden

Steuerwerk

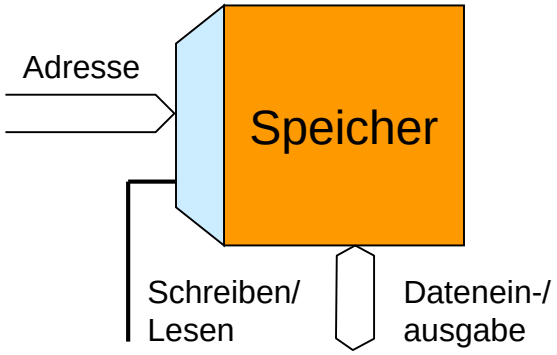
- Koordination der Komponenten des Prozessors
  - Versorgung mit Daten und Adressen
  - Selektion von Geräten
  - Auswahl von Operanden und Operationen
- Ursprünglich Steuerung durch den Bediener, später Automatisierung durch zeitliche Abfolge von Binärinformationen
- Formalisierung der Anweisung nötig:

→ **Befehl:** (Maschinenlesbare) Anweisung, aus der hervorgeht, welche Operation mit welchen Operanden durchgeführt werden soll

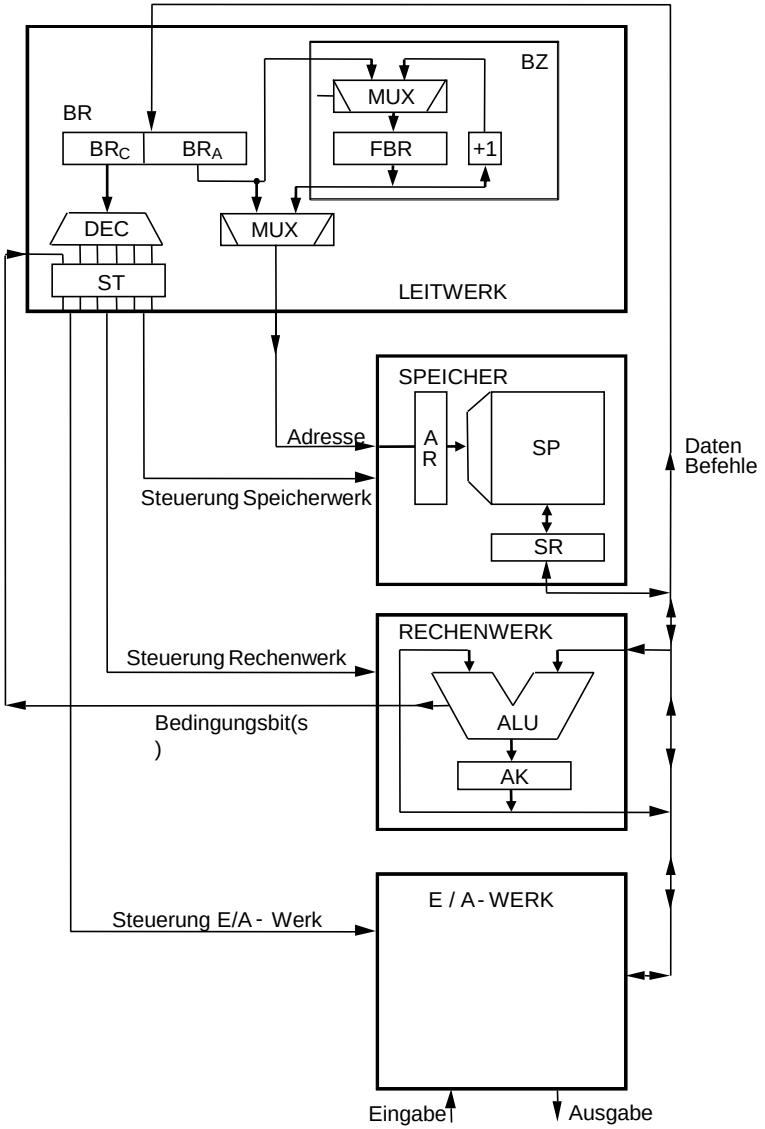
→ **Befehlsformat:** Festlegung, wie die für einen Befehl notwendigen Angaben dargestellt werden

Speicherwerk

- Im einfachsten Fall nur eine Reihe von Registern zur Aufbewahrung von Operanden, Bedingungsvektor und Ergebnis
- I.A. jedoch ein mehr oder weniger umfangreicher, adressierbarer Schreib/Lese-Speicher
- Ermöglicht es, während der Verarbeitung eine große Zahl von Operanden und (Zwischen-)Ergebnissen bereitzuhalten, um so nicht für jede Operation das Ein-/Ausgabewerk benutzen zu müssen
- Die Zahl der Verfügbaren Speicherzellen richtet sich nach der Breite des Adressvektors



Schema eines einfachen Rechners

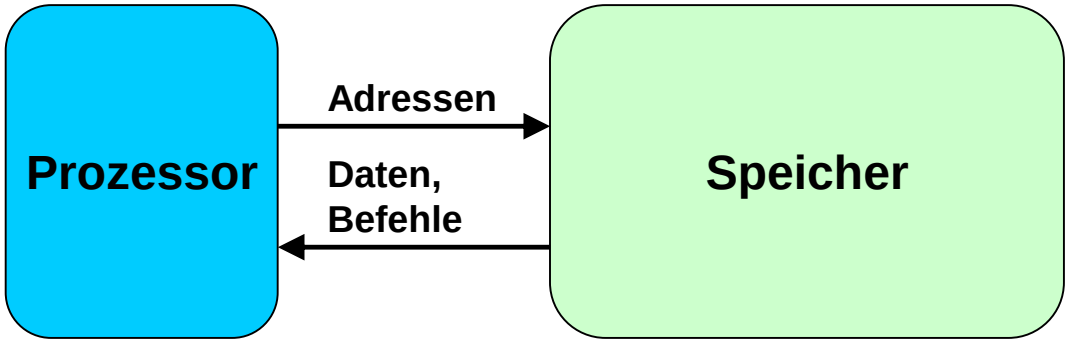


Abkürzungen:

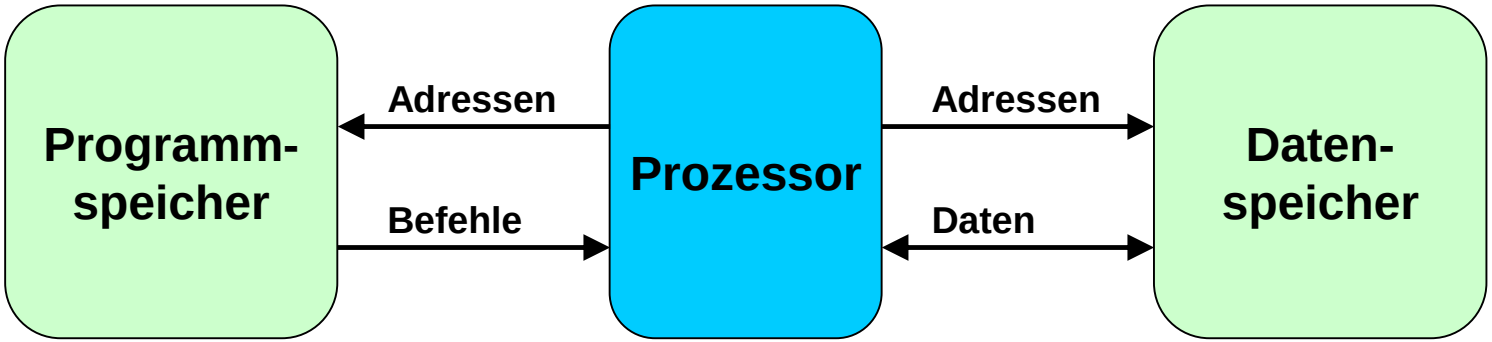
BR	Befehlsregister
BRC	Codeteil von BR
BRA	Adressteil von BR
ST	Register für Steuerbits
BZ	Befehlszähler
AR	Adressregister
SR	Speicherregister
AK	Akkumulator

Rechnerarchitekturen

von Neumann-Architektur



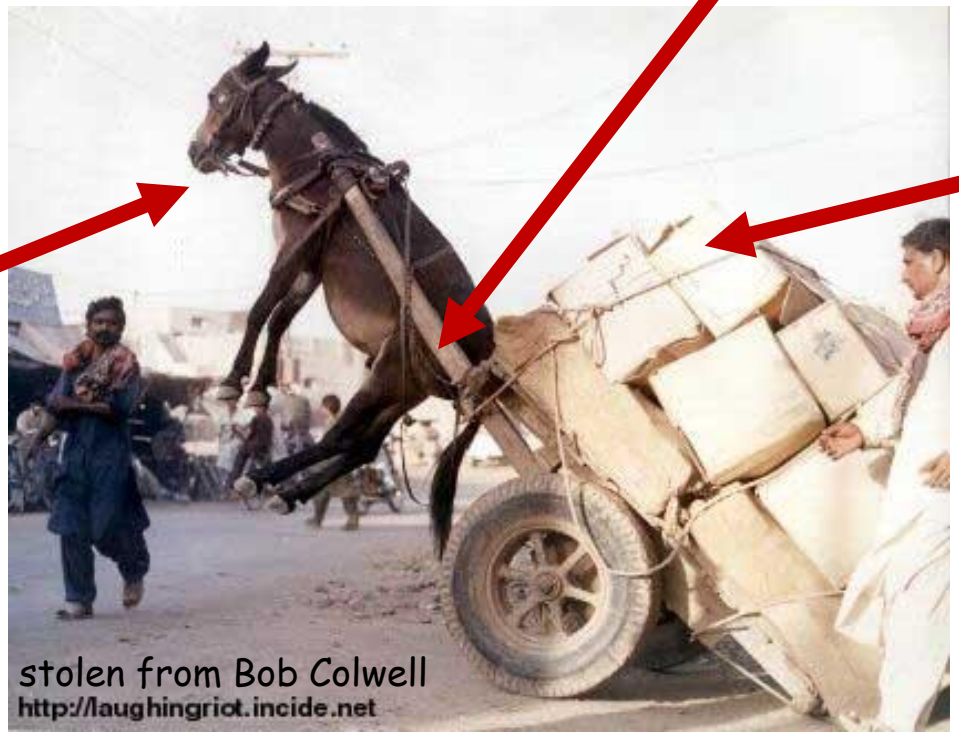
Harvard-Architektur



Von Neumann (vN) Architektur kommt an ihre Grenzen …

vN: unbalanciert

vN Flaschenhals



CPU (Prozessor-Kern)

Cache-Speicher, …

Quelle: R. Hartenstein, Univ. Kaiserslautern

Rechnerarchitekturen

- Klassifizierung externer Architekturen
    - wie werden Operanden von Befehlen im Prozessor gespeichert
    - Anzahl der Operanden pro Befehl
    - Residenz von Operanden
    - Operationsvorrat
    - Typ und Länge der Operanden
    - ...
- Beispielklassen:
    - Akkumulator-Maschine
    - Stack-Maschine
    - Registersatzmaschine
        - CISC-Architektur
        - RISC-Architektur

Rechnerarchitekturen

Akkumulatormaschine:

- Organisationsprinzip vieler einfacher, früherer Rechner, aber auch z.B. Motorola 6809
- die Rechenergebnisse wurden in einem Register, dem Akkumulator (AC), „akkumuliert“
- typische Befehle:

  LAC m: lade Wert unter Adresse m in den AC

  SAC m: speichere den Inhalt von AC nach Adresse m

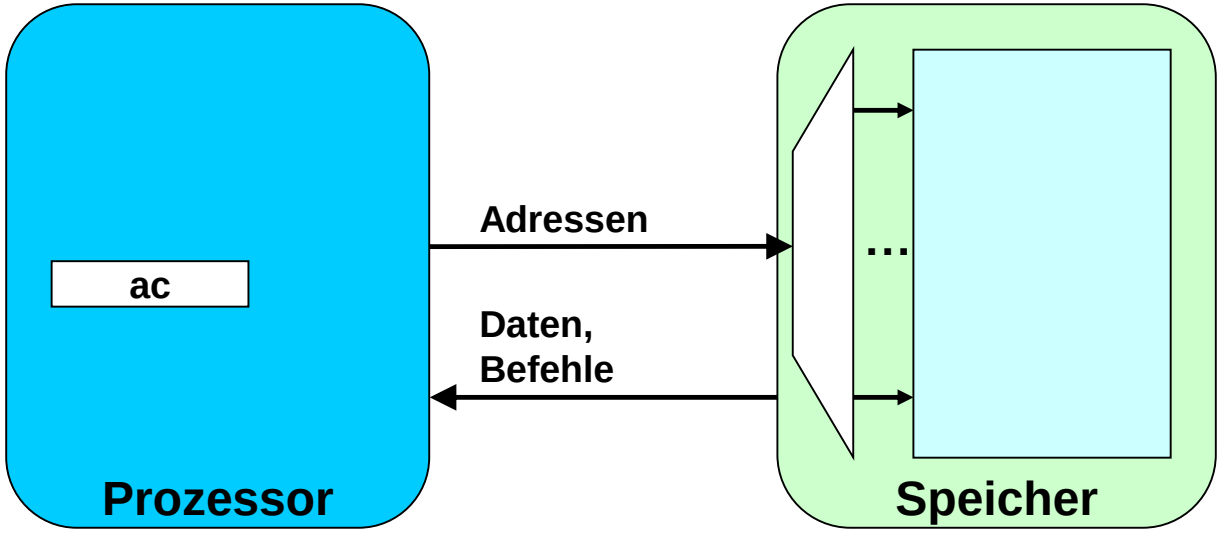
  ADD m: addiere den Inhalt von AC mit dem Wert unter Adresse m und speichere das Resultat nach AC

  Beispiel c := a + b

  ```
  LAC a
  ADD b
  SAC c
  ```

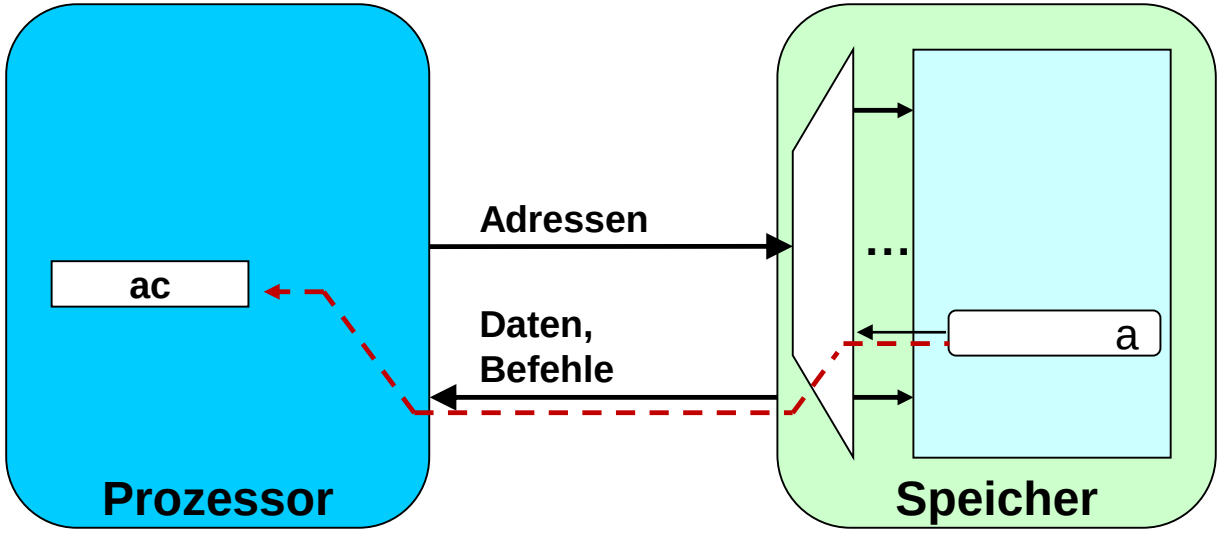
Rechnerarchitekturen

- Struktur:



Rechnerarchitekturen

- Befehlsausführung anhand des Beispiels:

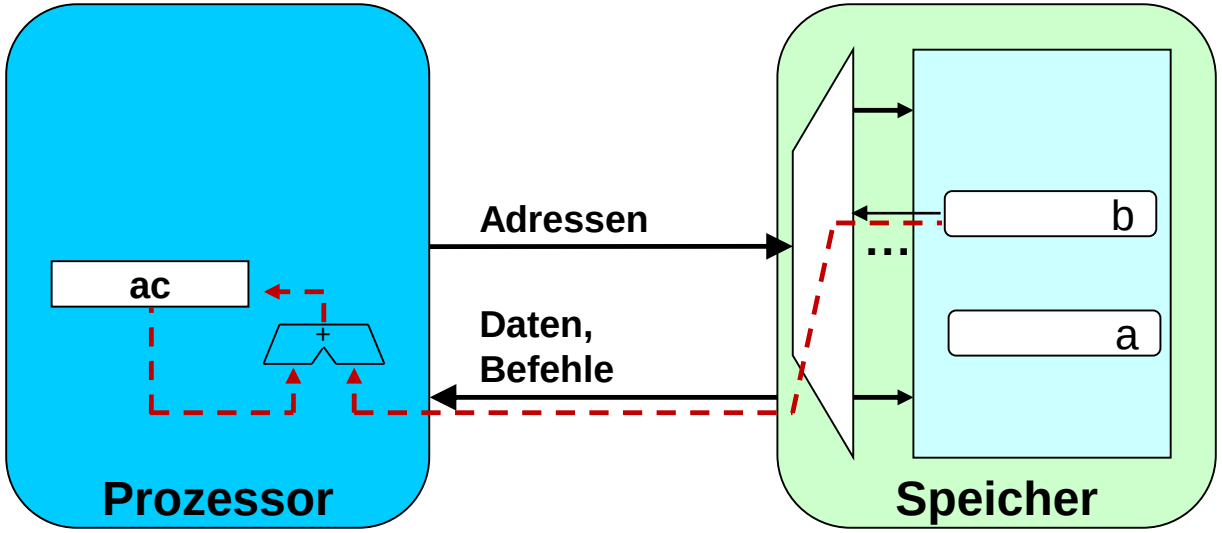


Beispiel c := a + b

```
LAC a
ADD b
SAC c
```

Rechnerarchitekturen

- Befehlsausführung anhand des Beispiels:

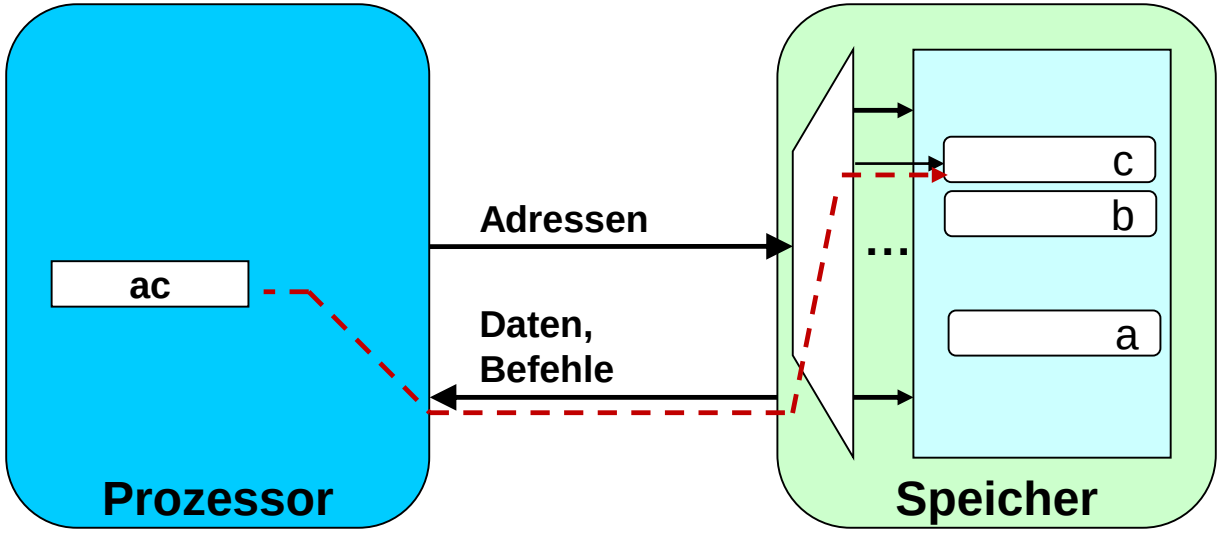


Beispiel c := a + b

```
LAC a
ADD b
SAC c
```

Rechnerarchitekturen

- Befehlsausführung anhand des Beispiels:



Beispiel c := a + b

LAC a

ADD b

SAC c

Rechnerarchitekturen

Stackmaschine:

- Befehle beziehen sich auf einen Stack zur Speicherung u.a. von Zwischenergebnissen
- typische Befehle:

  PUSH m: lege den Wert unter Adresse m auf den Stack

  ADD : addiere die beiden obersten Werte des Stacks, entferne sie und lege die Summe als oberstes Element auf den Stack

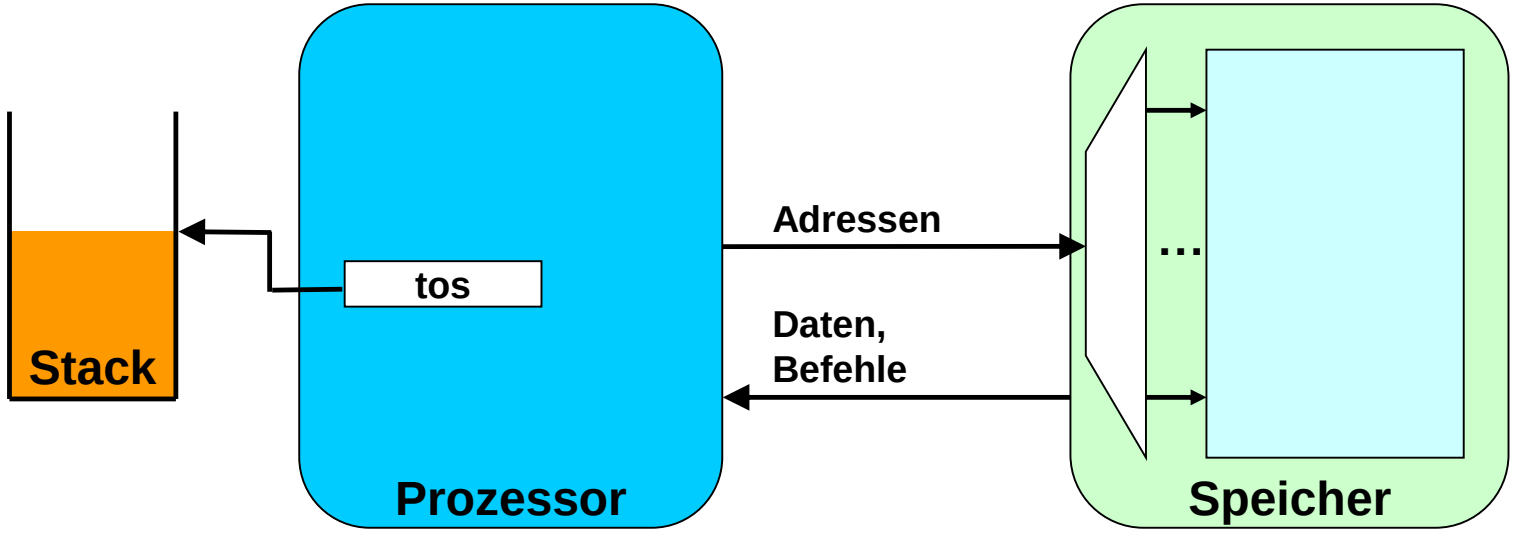
  STORE m: speichere das oberste Element auf dem Stack

- Beispiel c := a + b

```
PUSH a
PUSH b
ADD
STORE c
```

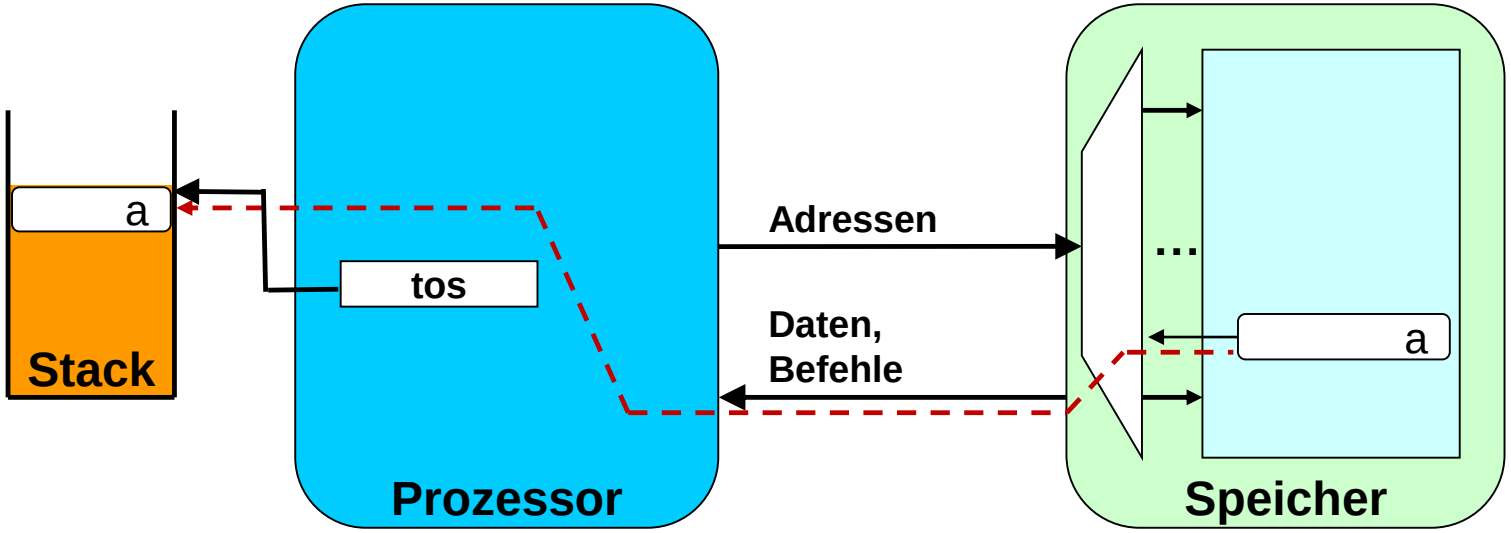
Rechnerarchitekturen

- Struktur:



Rechnerarchitekturen

- Befehlsausführung anhand des Beispiels:

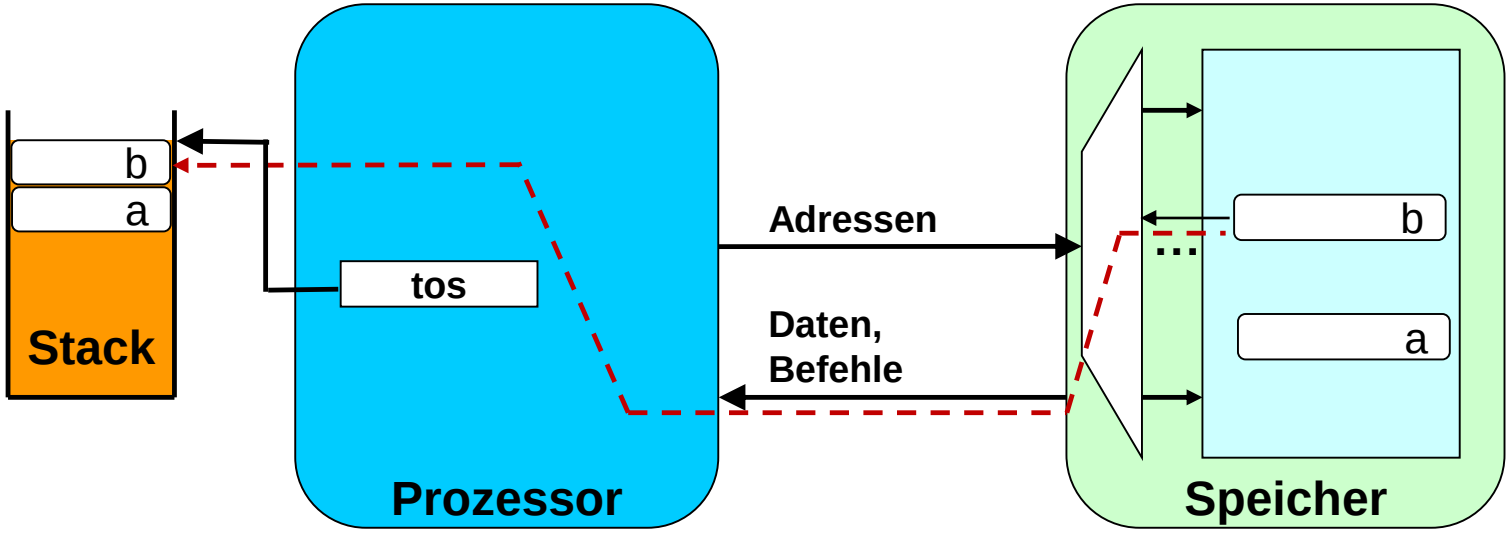


Beispiel c := a + b

```
PUSH a
PUSH b
ADD
STORE c
```

Rechnerarchitekturen

- Befehlsausführung anhand des Beispiels:



Beispiel c := a + b

PUSH a

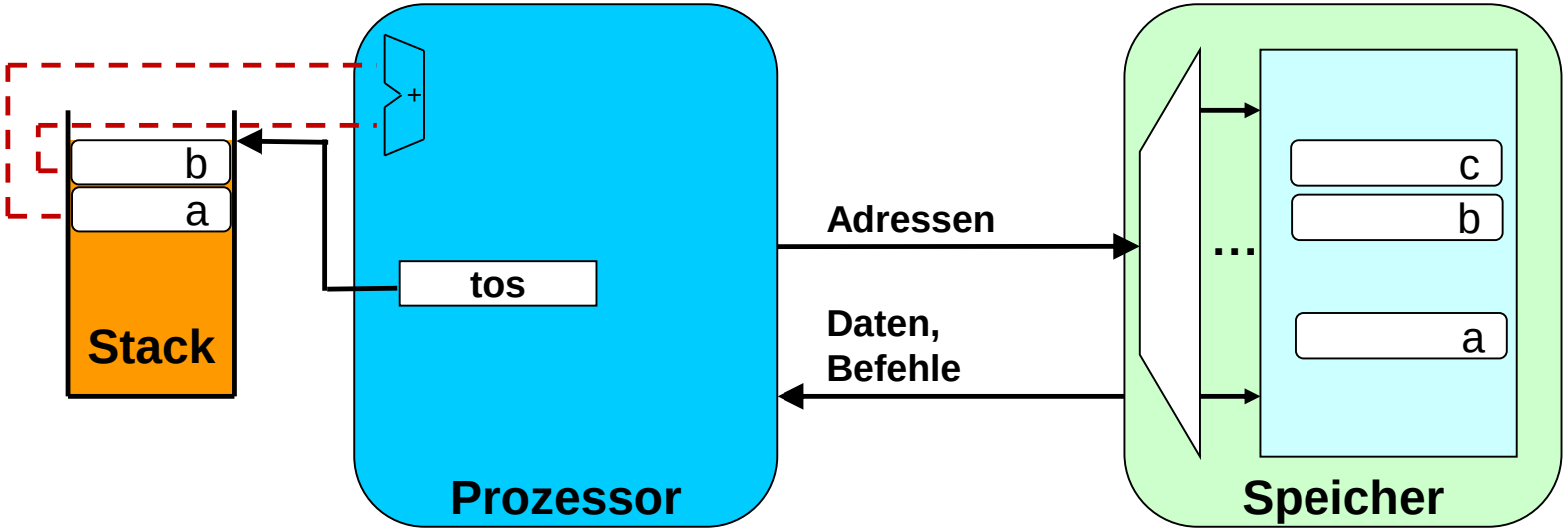
PUSH b

ADD

STORE c

Rechnerarchitekturen

- Befehlsausführung anhand des Beispiels:



Beispiel c := a + b

PUSH a

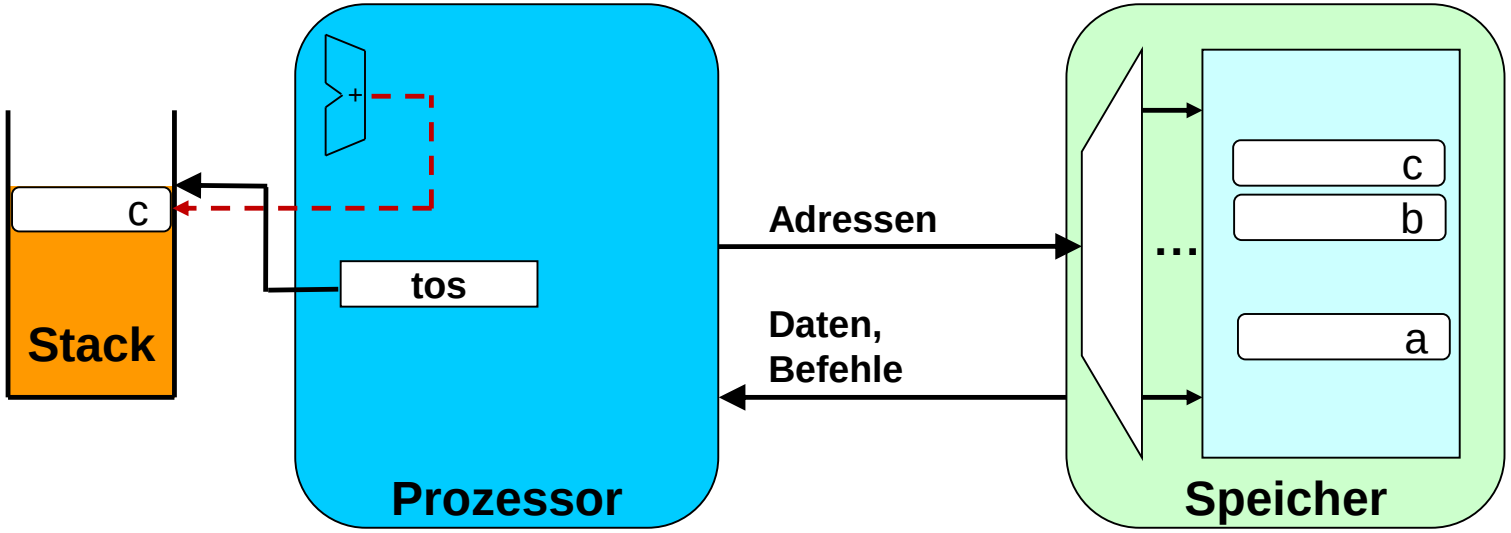
PUSH b

ADD

STORE c

Rechnerarchitekturen

- Befehlsausführung anhand des Beispiels:



Beispiel c := a + b

PUSH a

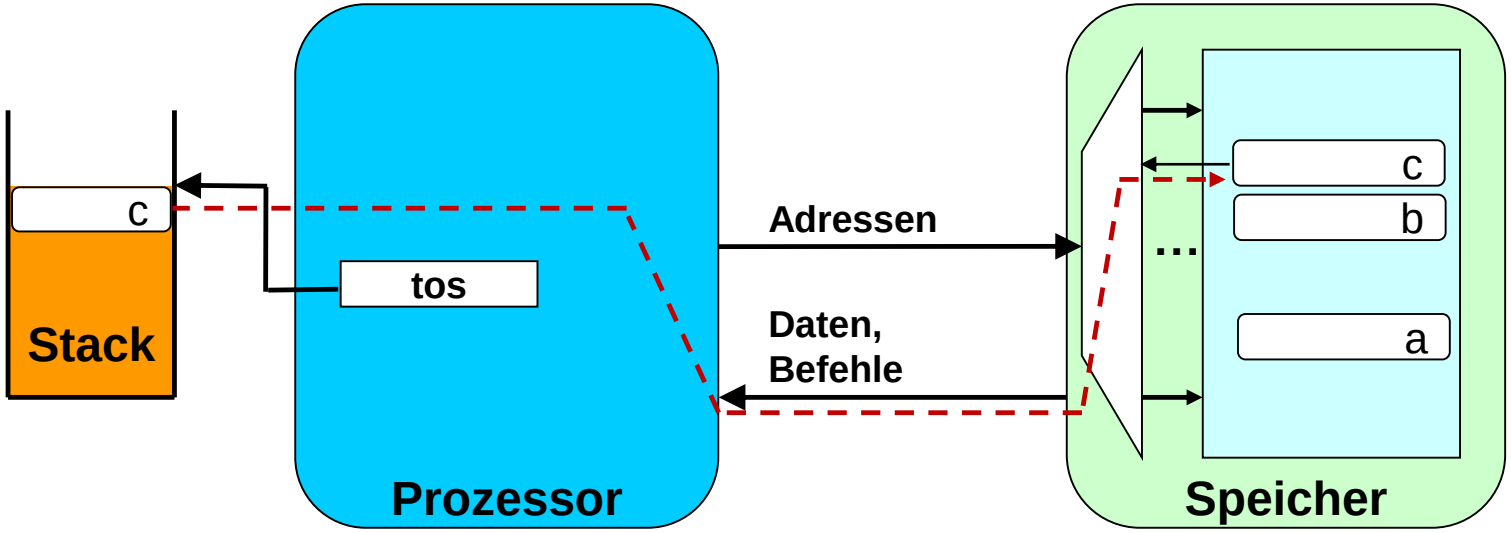
PUSH b

ADD

STORE c

Rechnerarchitekturen

- Befehlsausführung anhand des Beispiels:



Beispiel c := a + b

PUSH a

PUSH b

ADD

STORE c

Rechnerarchitekturen

Registersatzmaschine:

- der Prozessor enthält z.B. 16 oder 32 Universalregister
- üblich sind Dreiadressbefehle

  OP DEST, SRC1, SCR2 d.h. DEST := SRC1 OP SRC2
- oder Zweiadressbefehle

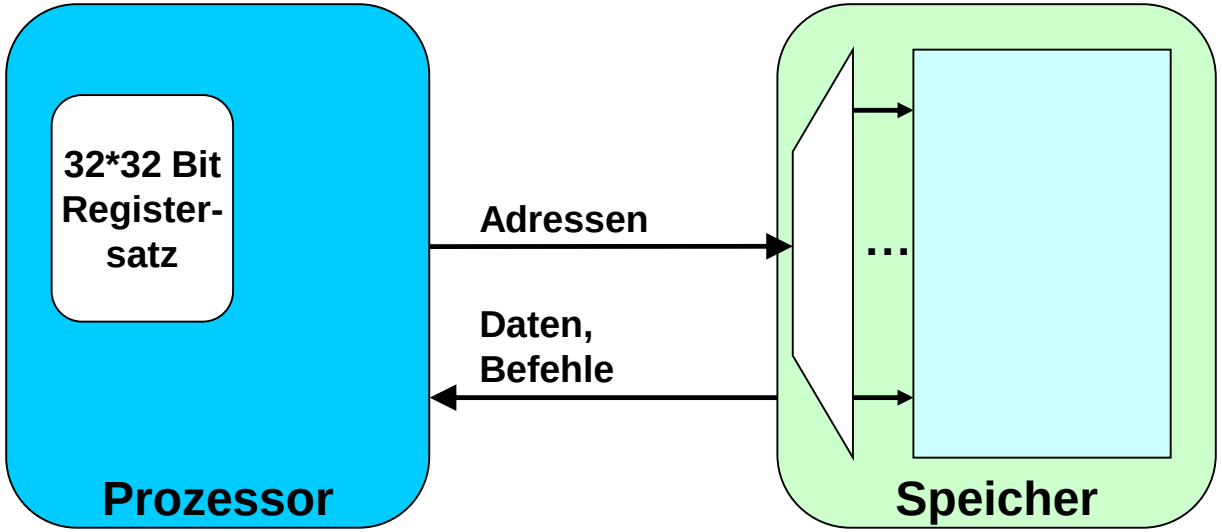
  OP DEST, SRC d.h. DEST := DEST OP SRC

  z.B. ADD R5, R6 bedeutet R5 := R5 + R6

- Beispiel IBM 360 (1964): Rechnerfamilie, d.h. eine externe Architektur, mehrere interne Architekturen, d.h. Hardware mit sehr unterschiedlichem Preis/Leistungsverhältnis

Rechnerarchitekturen

- Struktur:



Rechnerarchitekturen

Zwei Typen von Registersatzmaschinen:

- **CISC (complex instruction set computer)**

  Beispiele: VAX 11/780 (1979), INTEL x86 (1981)

  - sehr umfangreiche Befehlssätze
  - komplizierte Adressiermodi
  - Operanden Register und Speicheradressen
  - Versuch, die Konstrukte höherer Programmiersprachen durch teilweise sehr komplizierte Befehle zu unterstützen
  - einfaches Beispiel c := a + b

```
MOVE R1, a
ADD R1, b
MOVE c, R1
```

Rechnerarchitekturen

- **RISC (reduced instruction set computer),** auch Load/Store-Architektur

  Beispiele: SPARC, Alpha, PowerPC

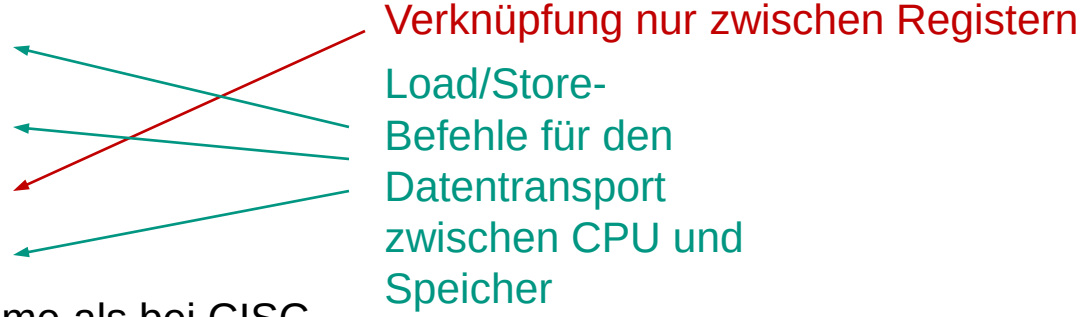
  - wenige, i.a. gleich-lange, einfache Instruktionen
  - nur Register als Operanden von Verknüpfungen wie z.B. Addition
  - Beispiel c := a + b

    LOAD R1, a

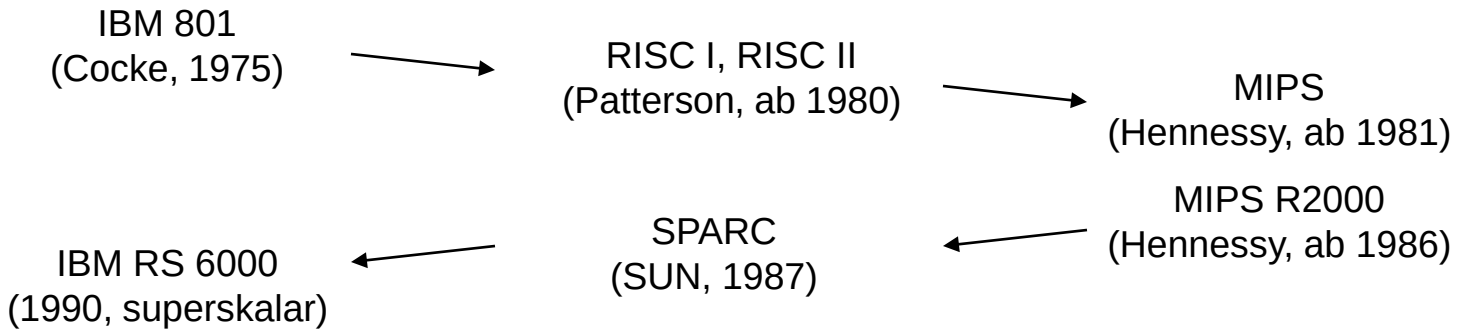
    LOAD R2, b

    ADD R3, R2, R1

    STORE c, R3



  - i.a. größere Programme als bei CISC
  - 2 bis 5 fache Leistung bei gleicher Technologie
  - Historie:



Rechnerarchitekturen

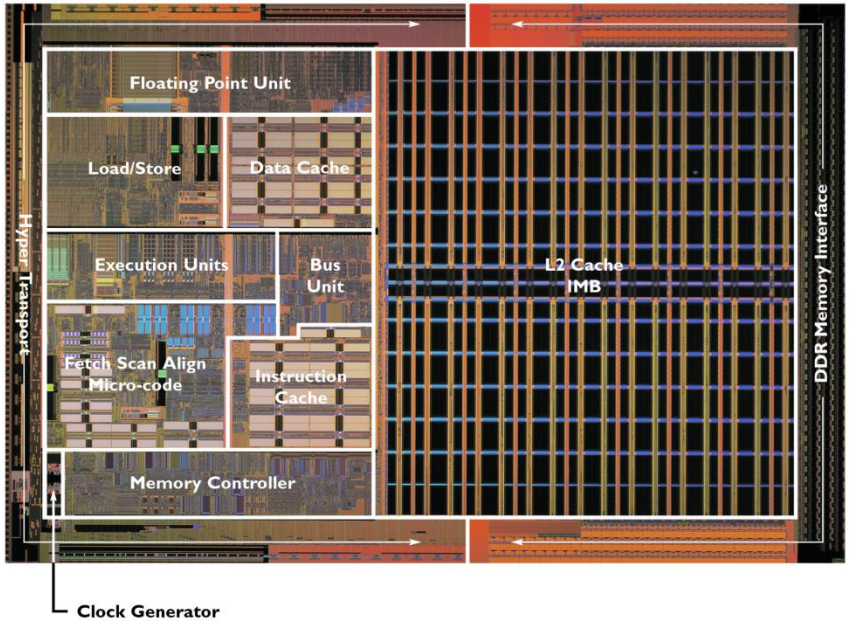
- Programmgröße, transportierte Daten und Code sowie Ausführungszeiten:

	Programmgröße		Transportierte Instr. und Daten		CPU Zeit	
	VAX	DEC 3100	VAX	DEC 3100	VAX	DEC 3100
Gnu C Compiler	410kB	688kB	18MB	21MB	291s	90s
TeX	159kB	217kB	67MB	78MB	449s	95s
Spice	223kB	372kB	99MB	106MB	352s	94s

- CISC Programme (VAX) sind meist kleiner, benötigen jedoch eine längere Ausführungszeit
- RISC Programme (DEC) dagegen sind länger bei gleichzeitig schnellerer Ausführung

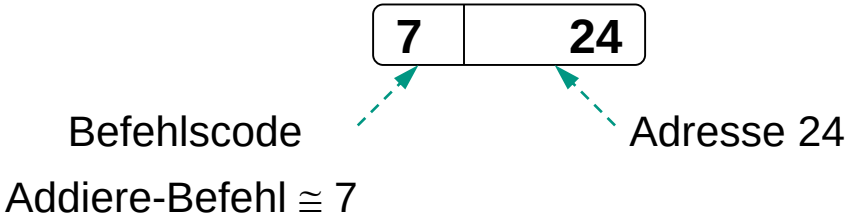
Rechnerarchitekturen

- Beispiel für eine CISC Maschine: AMD Opteron Prozessor
  - Taktfrequenz 1,5 – 2,2 GHz
  - AMD64 Instruktionen, MMX, 3DNow, SSE, SSE2 Support
  - 64 KByte Level 1 Cache, 2-Wege assoziativ
  - 1 MByte Level 2 Cache, 16-Wege assoziativ
  - 48-bit virtueller Adressraum, 40-bit physikalisch
  - HyperTransport Anbindung zwischen CPUs und Systemkomponenten



Befehlsausführung

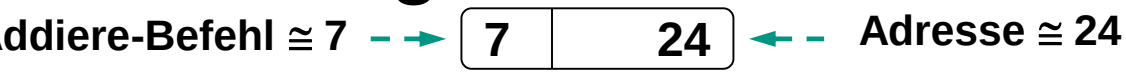
- Verarbeitungsphasen eines Befehls am Beispiel
  - Annahme: einfache Akkumulatormaschine
  - Befehlsformat



  - der Addiere-Befehl holt den Wert unter der im Befehl angegebenen Adresse (im Beispiel 24) und addiert ihn zum Inhalt

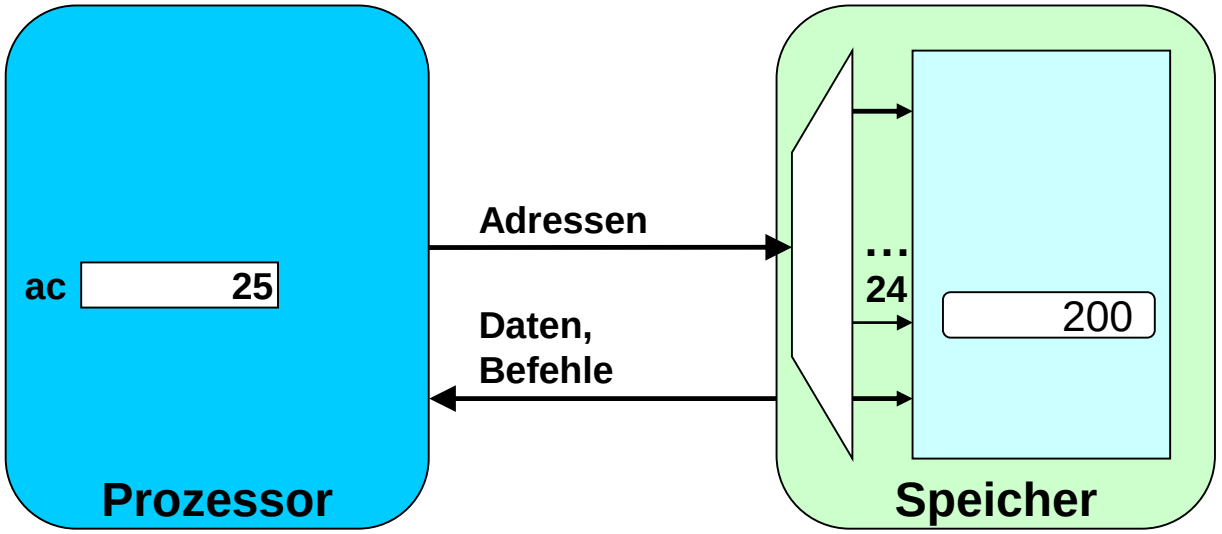
Befehlsausführung

Befehlscode Addiere-Befehl ≅ 7

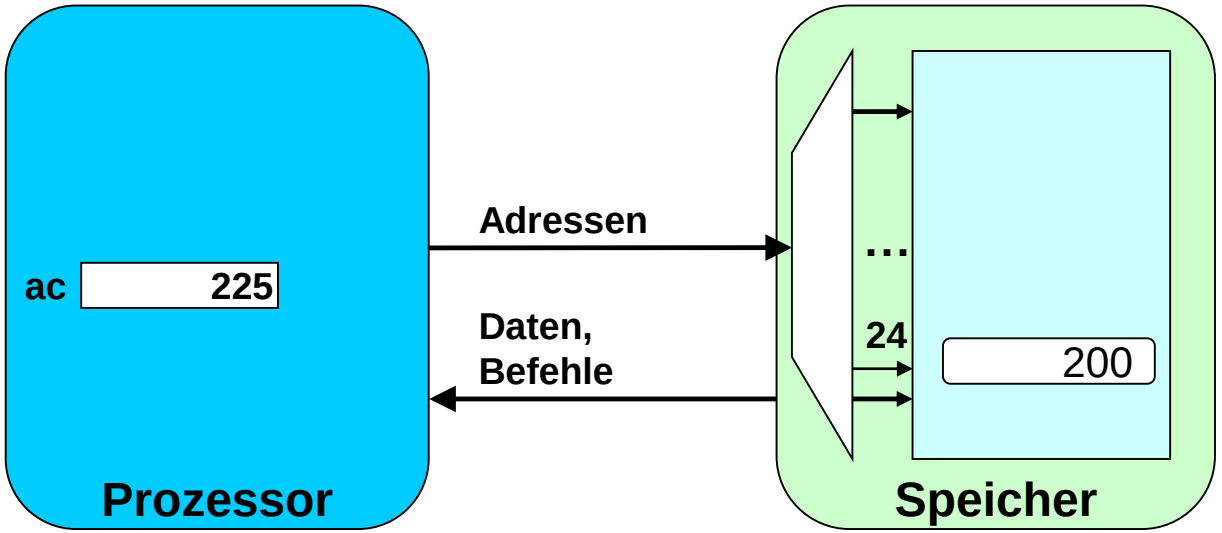


Adresse ≅ 24

Vorher:

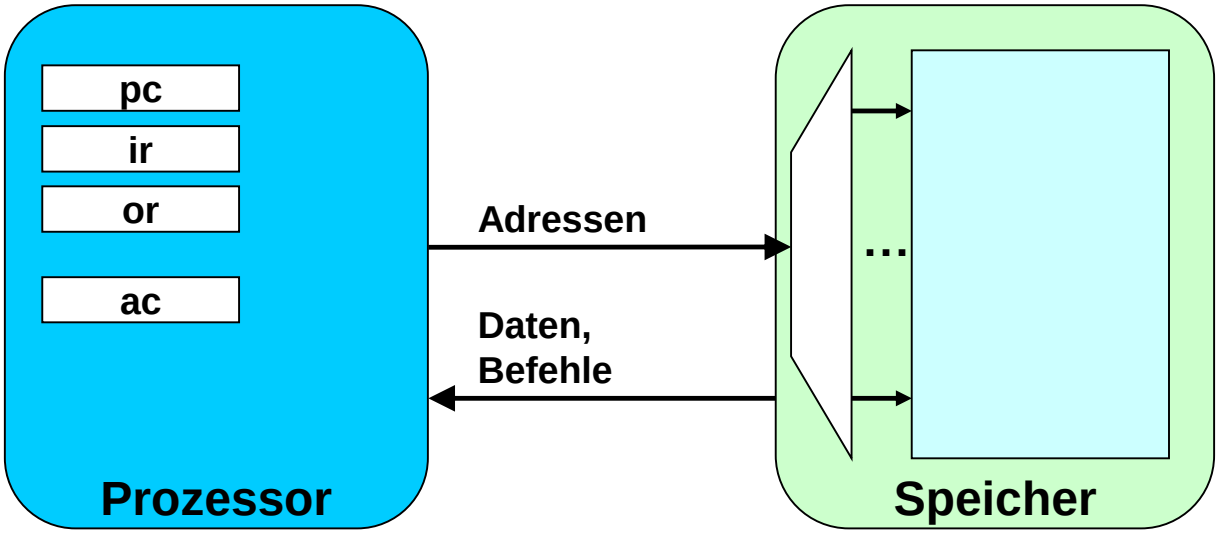


Nachher:



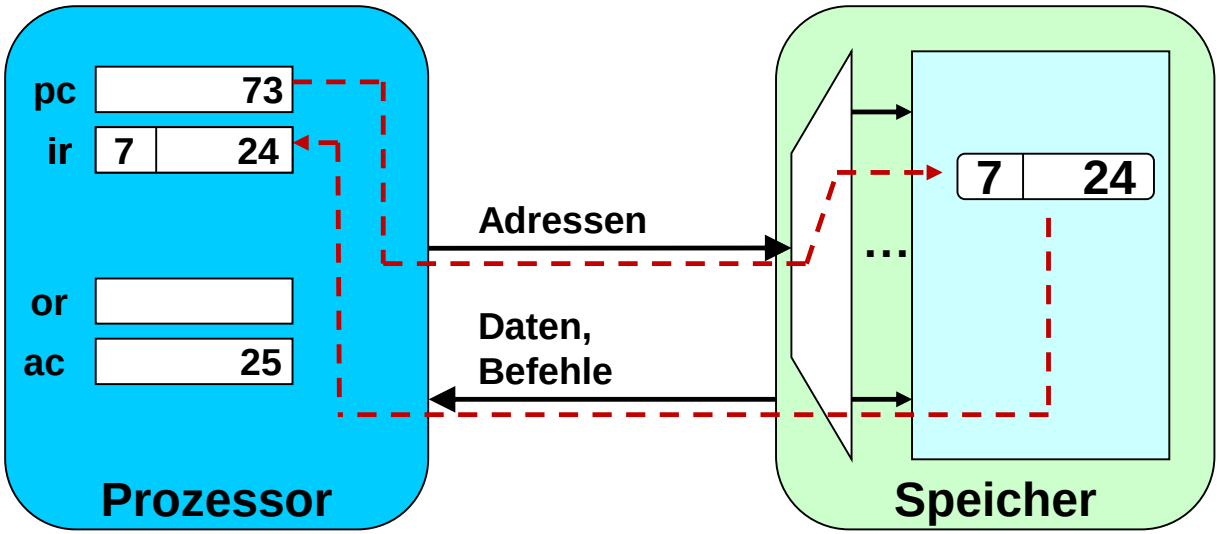
Befehlsausführung

- Struktur:
  - pc (program counter, auch ip: instruction pointer)
  - ir (instruction register)
  - or (operand register)
  - ac (accumulator)



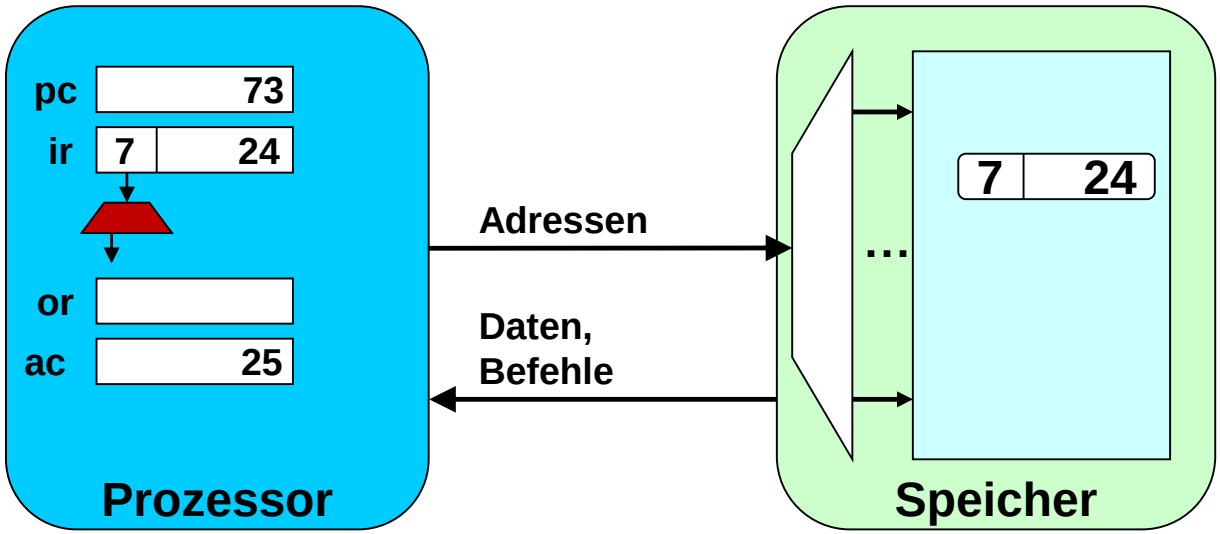
Befehlsausführung

- 1. Befehlsholphase (instruction fetch, IF)



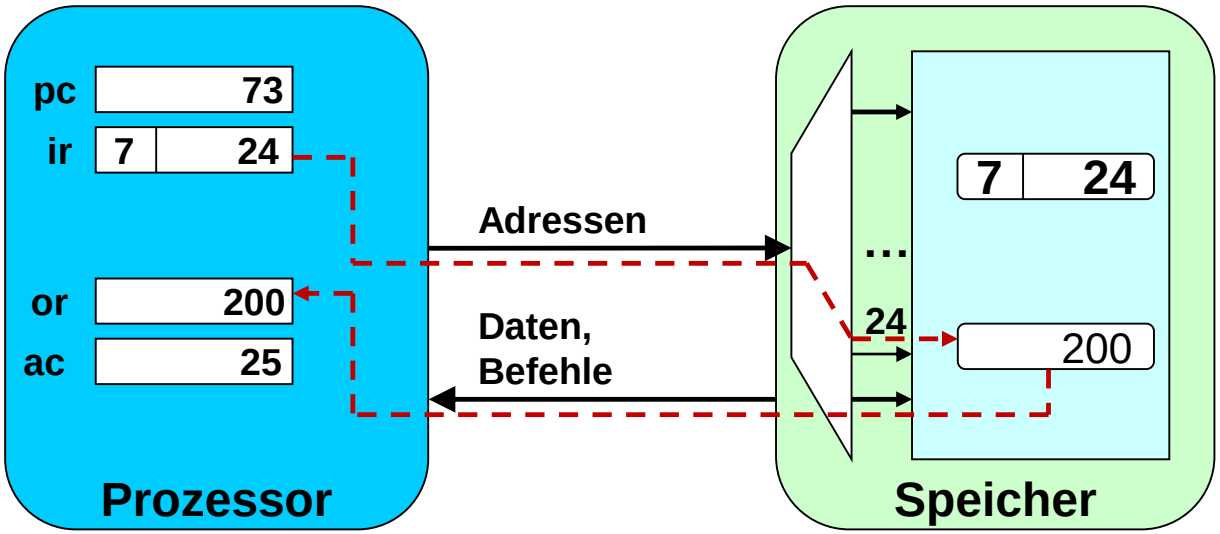
Befehlsausführung

- 2. Befehl dekodieren (instruction decode, ID)



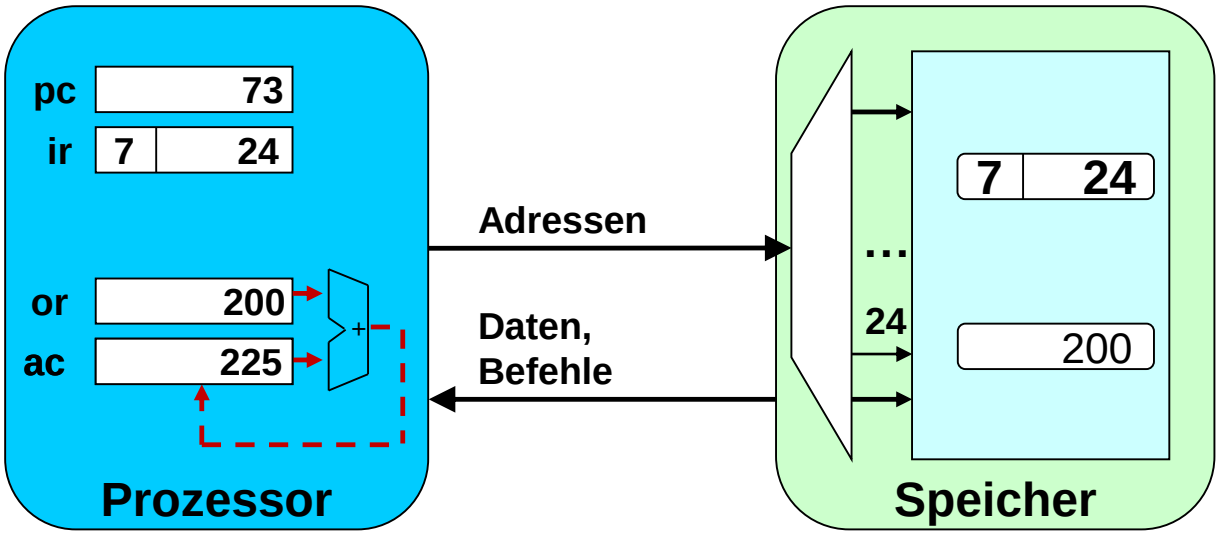
Befehlsausführung

- 3. Operand holen (operand fetch, OF)



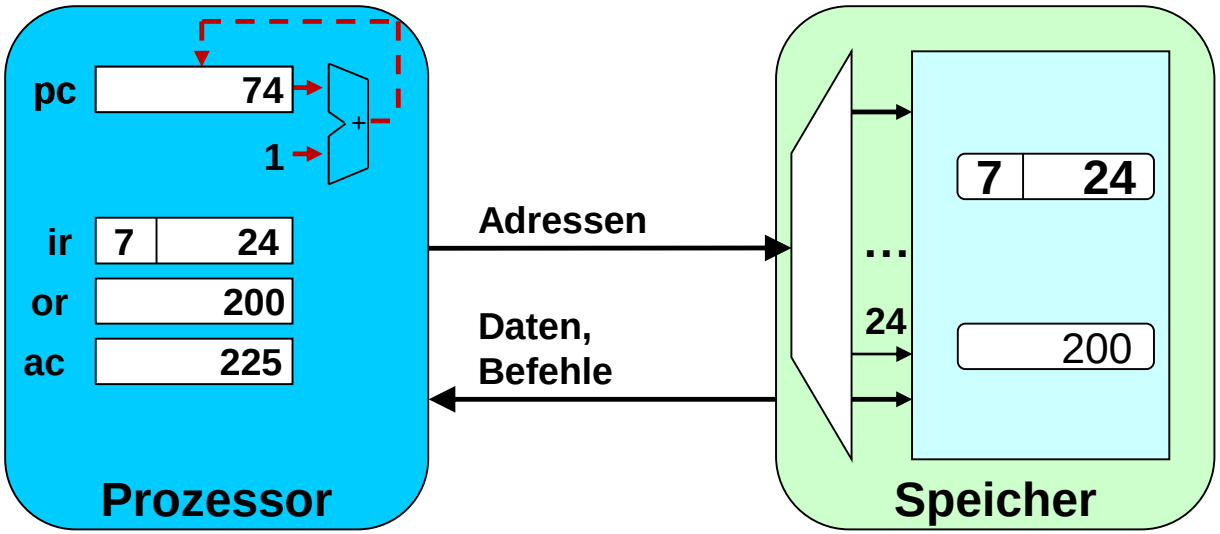
Befehlsausführung

- 4. Befehl ausführen (instruction execute, EX)



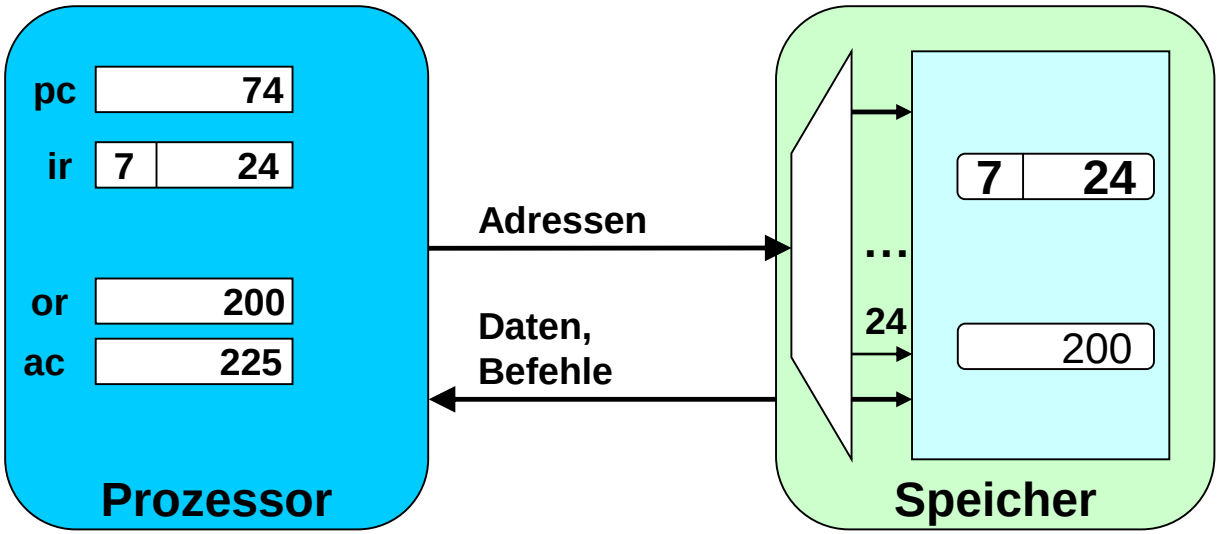
Befehlsausführung

- Befehlszähler inkrementieren
  - kann als separater Schritt oder parallel erfolgen



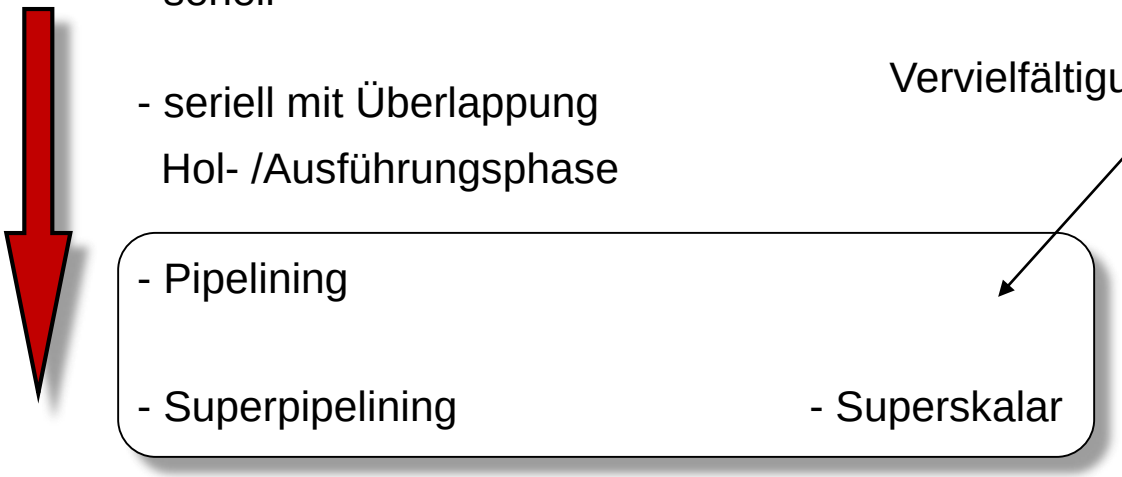
Befehlsausführung

- Resultat:



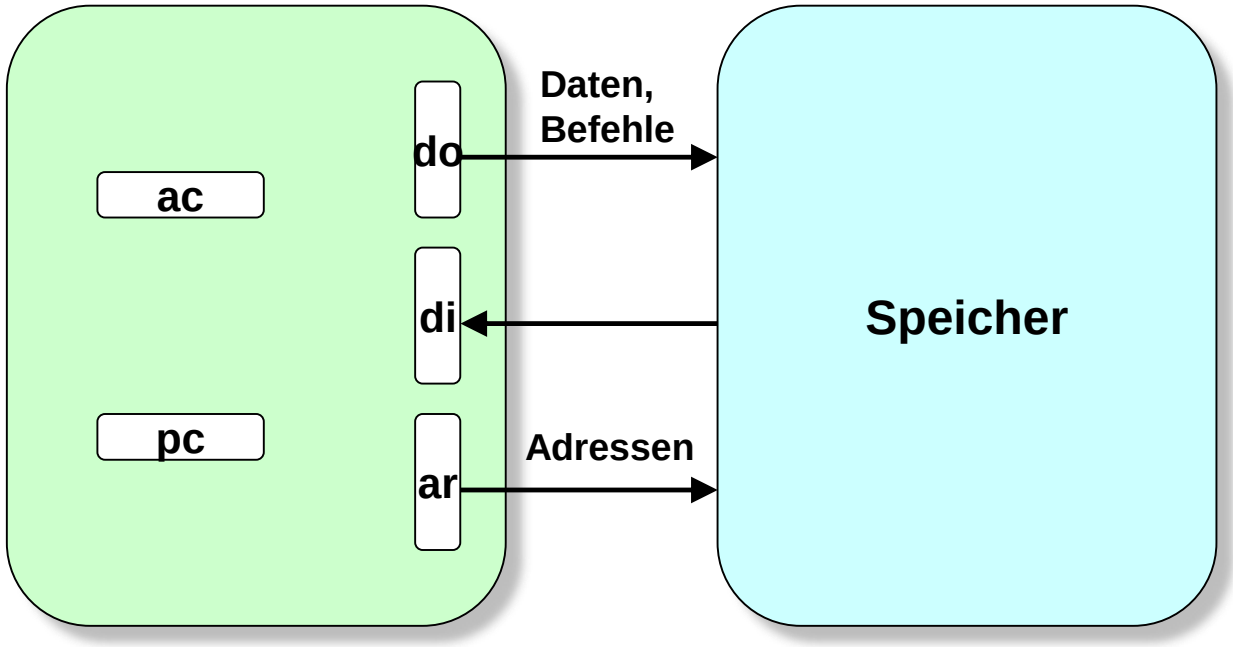
Befehlsausführung

- Überlappung von Befehlshol- und Ausführungsphase
  - Genealogie von Techniken zur Beschleunigung der Instruktionsausführung



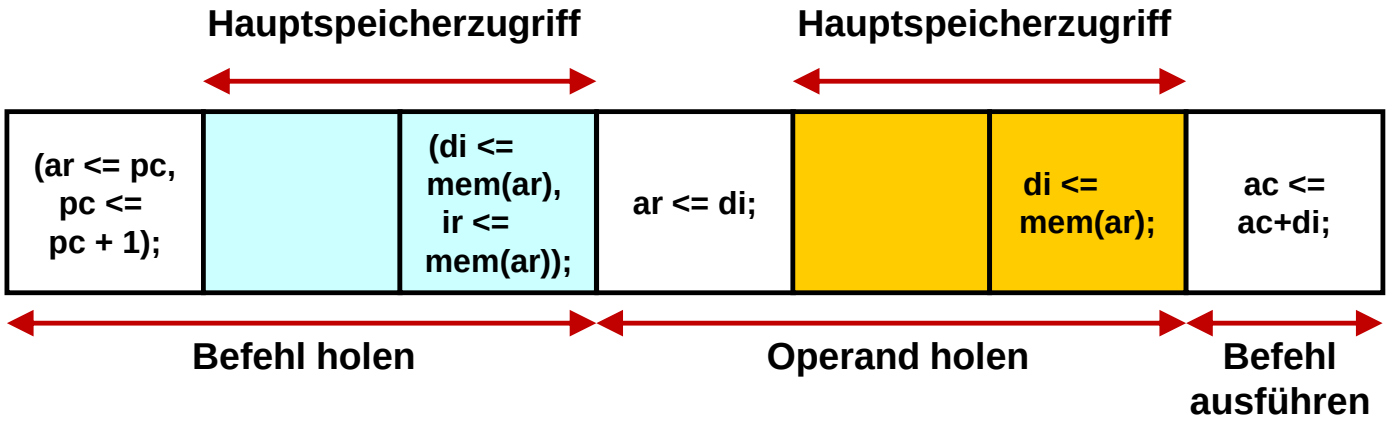
Befehlsausführung

- Beispiel: einfache Akkumulatormaschine



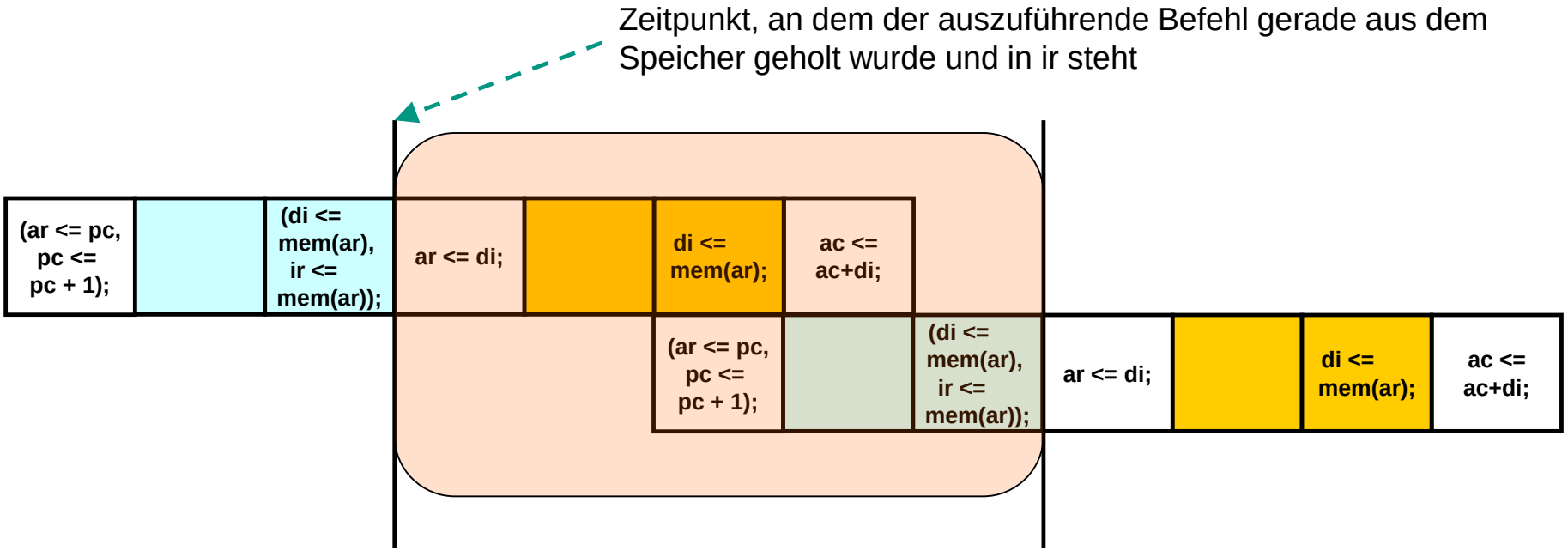
Befehlsausführung

- Annahmen:
  - Hauptspeicherzugriffzeit = 2 * Prozessortaktzeit
  - Befehlsauslegung für größtmögliche Geschwindigkeit
  - Beispiel: Addiere-zu-Akkumulator-Befehl
  - Auslastung bei serieller Ausführung:
    - Hauptspeicher: 57%
    - CPU: 57%



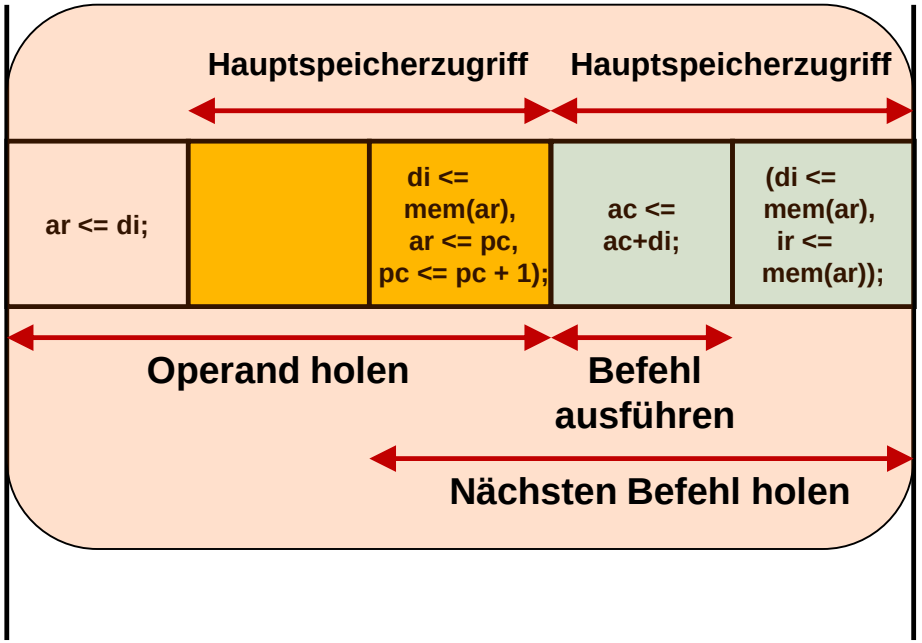
Befehlsausführung

Überlappung der Ausführungsphase mit dem Holen des nächsten Befehls:



- es werden keine zusätzlichen Ressourcen benötigt
- eine zeitliche Einsparung von 2 Takten pro Befehl => 28,6 % Leistungssteigerung

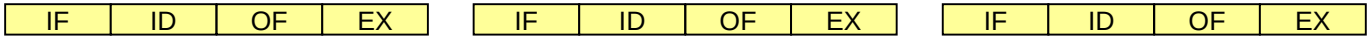
Befehlsausführung



- Während eines Speicherzugriffs wird bereits der nächste Speicherzugriff vorbereitet
- Ebenfalls Parallel zum Speicherzugriff: Kalkulation des Ergebnisses (Befehl ausführen)
- nur noch 5 Takte notwendig
- keine zusätzlichen Kosten
- Auslastung bei überlappter Ausführung:
  - Hauptspeicher: 80%
  - CPU: 80%

Rechnerarchitekturen

- Fortgeschrittene Organisationsprinzipien
  - die Verarbeitung mehrerer Befehle nacheinander erfolgt im Prinzip in einer Schleife

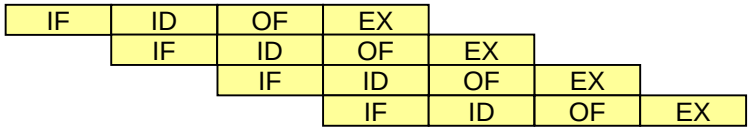


Befehl 1 **Befehl 2** **Befehl 3**

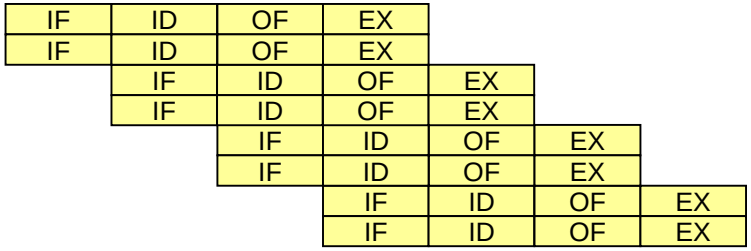
  - die Schleife kann auf verschiedene Arten parallelisiert werden

Pipelining: Überlappung der Ausführungsphase zur Reduktion der Latenzzeiten, resultiert in einer höheren Verarbeitungsgeschwindigkeit

Superskalar: zusätzlich zum Pipelining paralleles Ausführung von mehreren Befehlen, im Idealfall Vervielfachung der Leistung



Pipelining



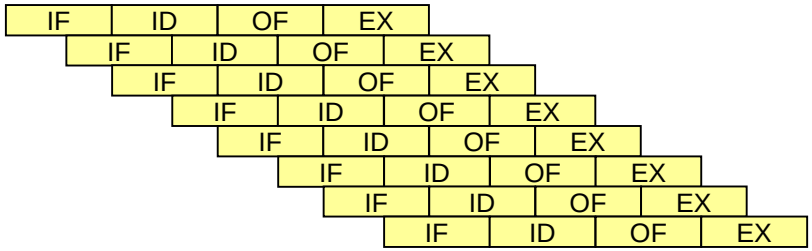
Superskalar

Rechnerarchitekturen

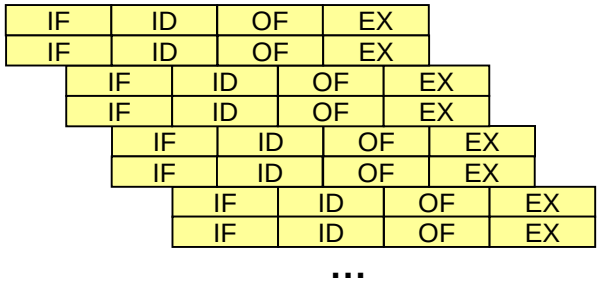
- die Schleife kann auf verschiedene Arten parallelisiert werden

  Superpipelining: Die Überlappung der Ausführungsphasen erfolgt nicht mehr taktweise, resultiert in einem höheren Instruktionsdurchsatz

  Superskalar/
  Superpipelining: zusätzlich zum Superpipelining paralleles Ausführung von mehreren Befehlen, im Idealfall Vervielfachung der Leistung



Superpipelining

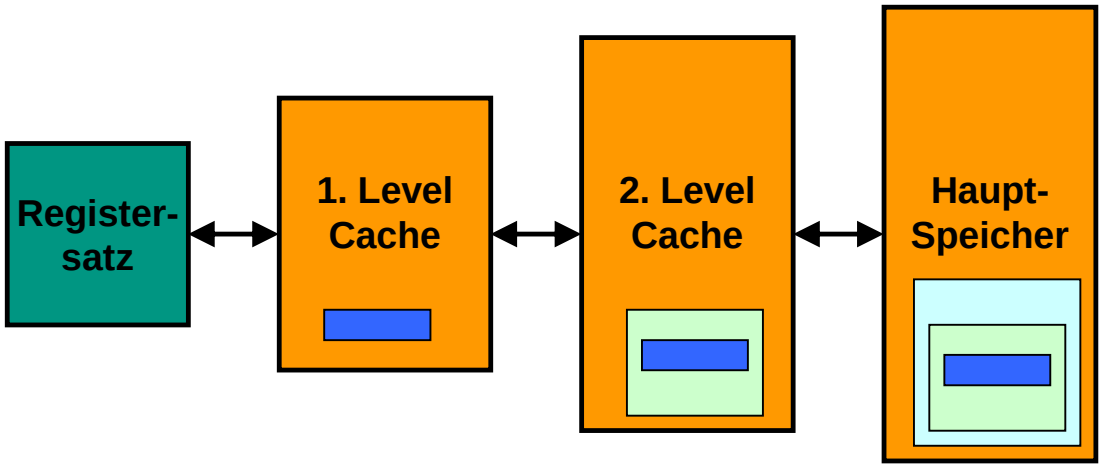


Superskalar / Superpipelining

Speicherorganisation

Speicherhierarchie

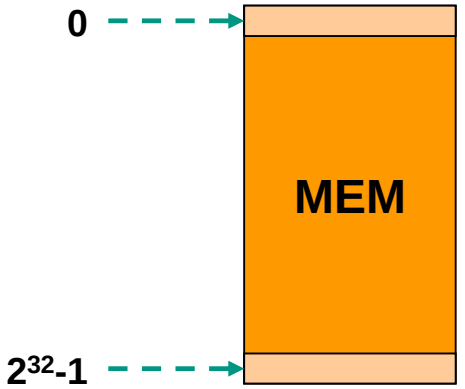
- Prinzip der Lokalität:
  - zeitlich: benutzte Daten/Befehle werden bald wieder benutzt
  - räumlich: auf Daten/Befehle, die im Adressraum nahe zu gerade benutzten liegen, wird wahrscheinlicher verwiesen als auf weiter weg liegende
- räumlich nahe beieinander liegende und oft referenzierende Teile (Blöcke), werden möglichst nahe am Prozessor gehalten



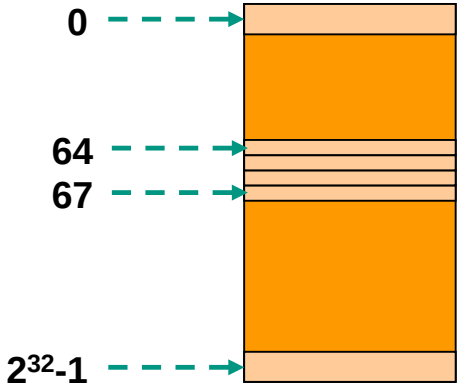
Speicherorganisation

Speicherorganisation

- üblich ist die Organisation in Bytes:
  z.B. 32bit Adresse $\Rightarrow$ 2^{32} Bytes adressierbar

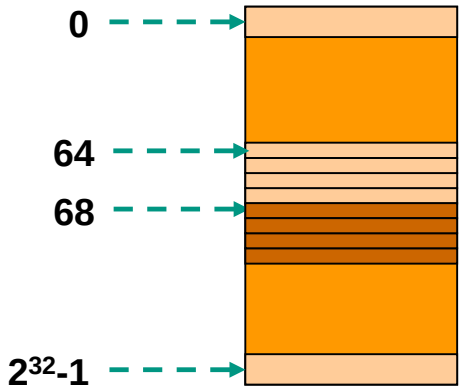


- Organisation z.B. in Bytes, Wörter (2 Bytes), Langwörter (4 Bytes), Quadwörter (8 Bytes), ist auch denkbar
- Ausrichtung (alignment)
  - Daten mit einer festen Länge können nur beginnend mit einer festen Adresse im Speicher untergebracht werden, z.B. Langwörter nur an Adressen, die ein vielfaches von 4 sind

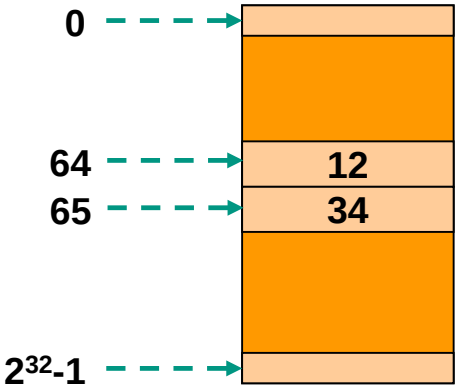


Speicherorganisation

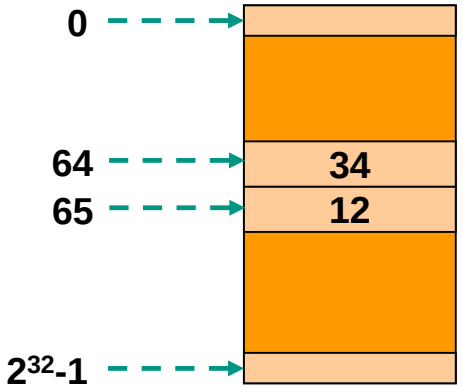
- besteht ein Befehl aus 4 Bytes, so muss die Byte-Adresse um 4 erhöht werden, um den nächsten Befehl zu holen



- dabei ist die Interpretation z.B. als ganze Zahlen zu unterscheiden, ob das niedrigstwerte Byte an der höchsten Adresse („big endian“, z.B. Apple), oder niedrigsten Adresse („little endian“, z.B. Intel x86) gespeichert wird
- Beispiel Speicherung von 1234h in einem Wort:



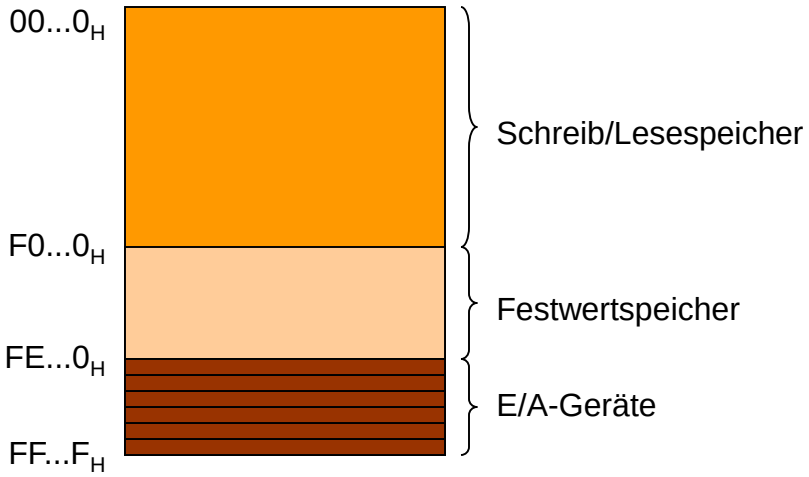
„big endian“



„little endian“

Speicherorganisation

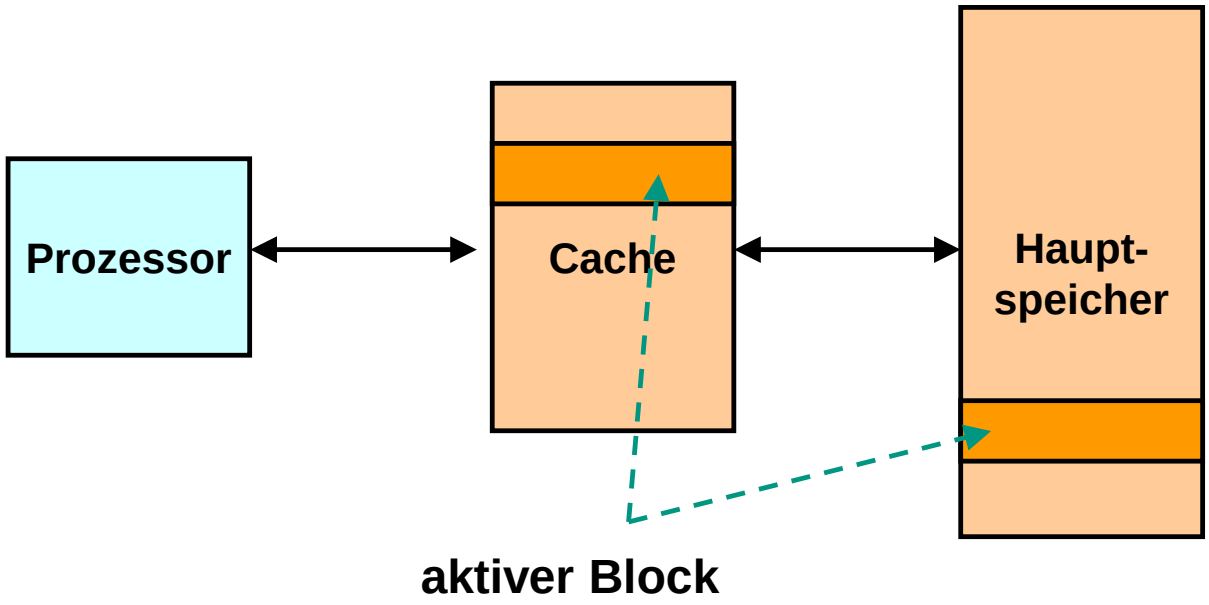
- Neben den Schreib/Lesespeichern existieren weitere Speichertypen
  - Festwertspeicher (ROM), behalten den Inhalt auch nach Abschalten der Spannungsversorgung
  - Speicherung von Konstanten oder grundsätzlichen Befehlsabläufen
- Aufteilung des Speichers in mehrere Teilspeicher
  - Auswahl der Bereiche über einen Teil des Adressvektors
  - Der Rest des Adressvektors adressiert die Speicherzelle innerhalb des Bereiches
- Auch E/A Geräte können wie ein Speicher behandelt werden („memory mapped I/O“)
  - Auswahl der Geräte über die Speicheradresse
  - Festlegung der Richtung über die Schreib/Lese-Steuerung des Speichers



Cacheorganisation

Cacheorganisation

- Prinzip: nur die tatsächlich gebrauchten Blöcke werden im Cache gehalten
- simples Beispiel: zweistufige Hierarchie Cache-Hauptspeicher



Cacheorganisation

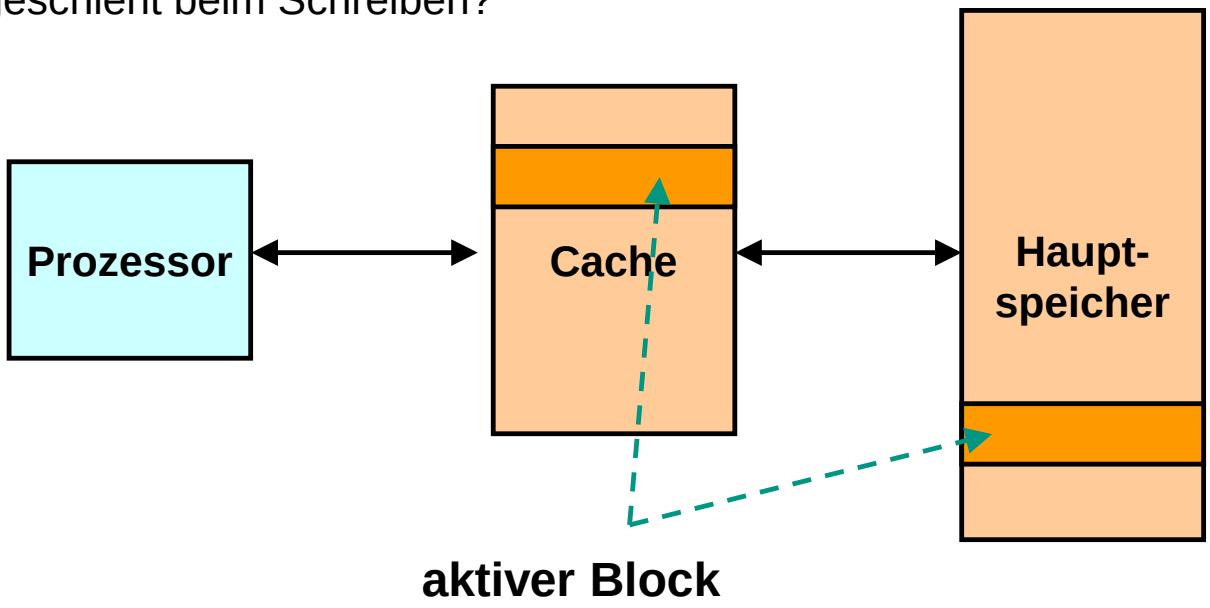
- Antworten auf vier Fragen entscheidend:

  A: wo kann ein Block im Cache plaziert werden?

  B: wie kann festgestellt werden, ob ein Block im Cache ist?

  C: welcher Block soll ersetzt werden, falls ein Block gebraucht wird?

  D: was geschieht beim Schreiben?

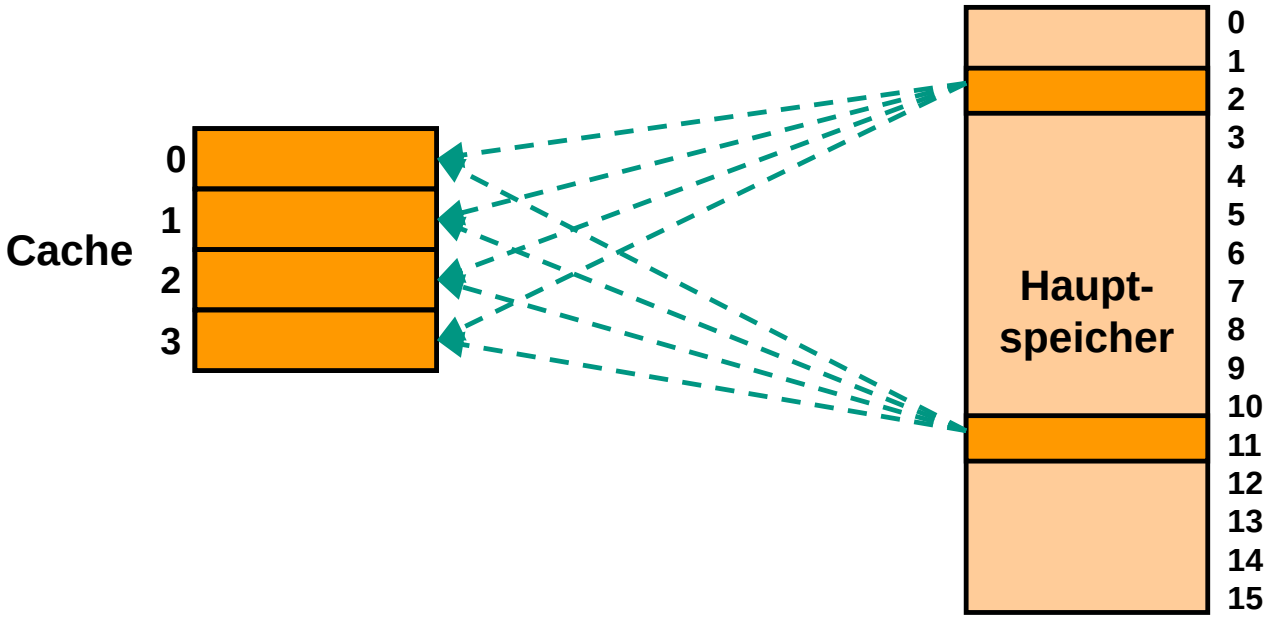


Cacheorganisation

A: wo kann ein Block im Cache plaziert werden?

1. Verfahren: voll-assoziativ

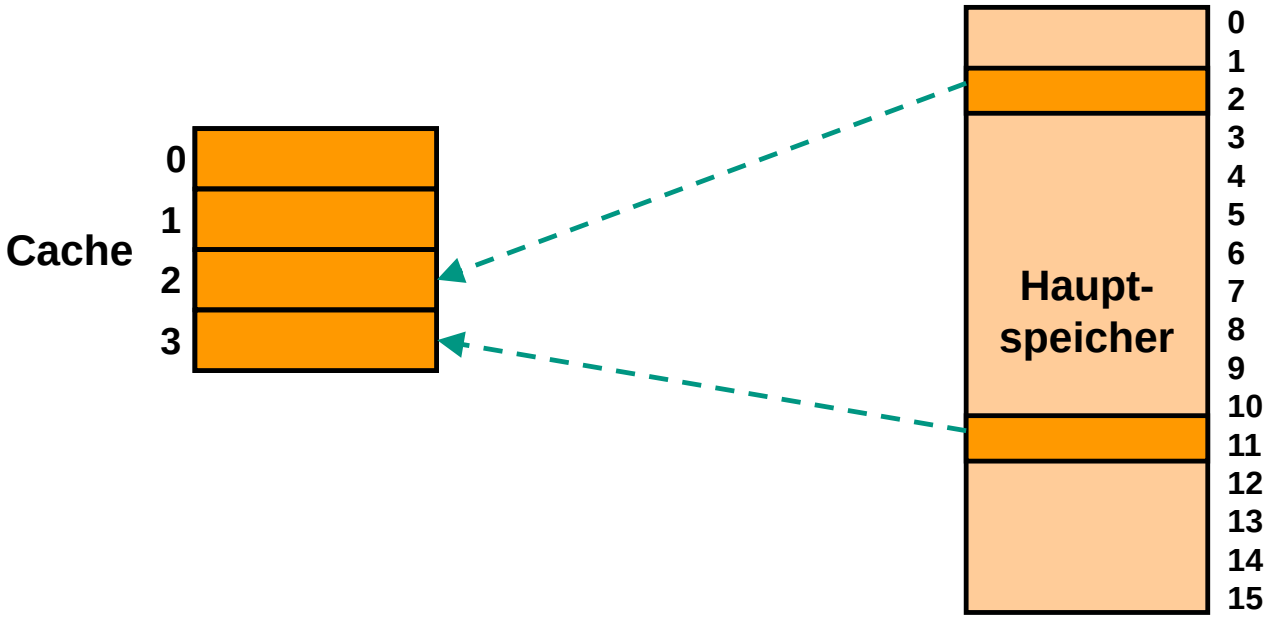
- jeder Block kann an allen Adressen gespeichert werden



Cacheorganisation

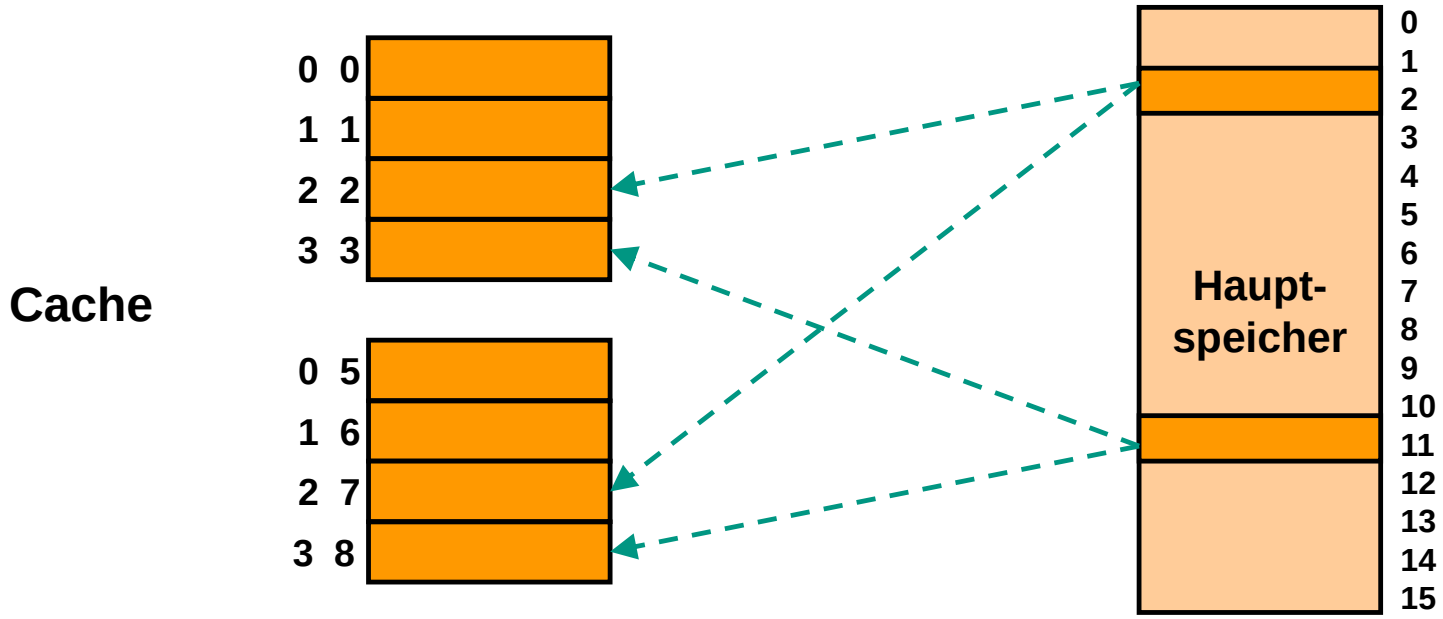
2. Verfahren: direkt-abbildend

- ein Block kann nur an einer Adressen mod n (n Cache-Größe, hier: 4) gespeichert werden



Cacheorganisation

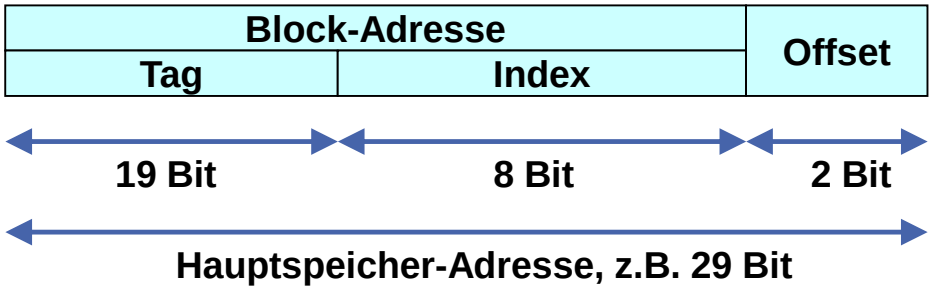
3. Verfahren: n-Wege assoziativ
   - ein Block kann an jeder der n Cache-Adressen gespeichert werden, bei denen die Hauptspeicheradresse mod m gleich der Cache-Adresse mod m ist
   - physikalischer Aufbau von n-unabhängigen Caches
   - Beispiel: 2-Wege assoziativ



Cacheorganisation

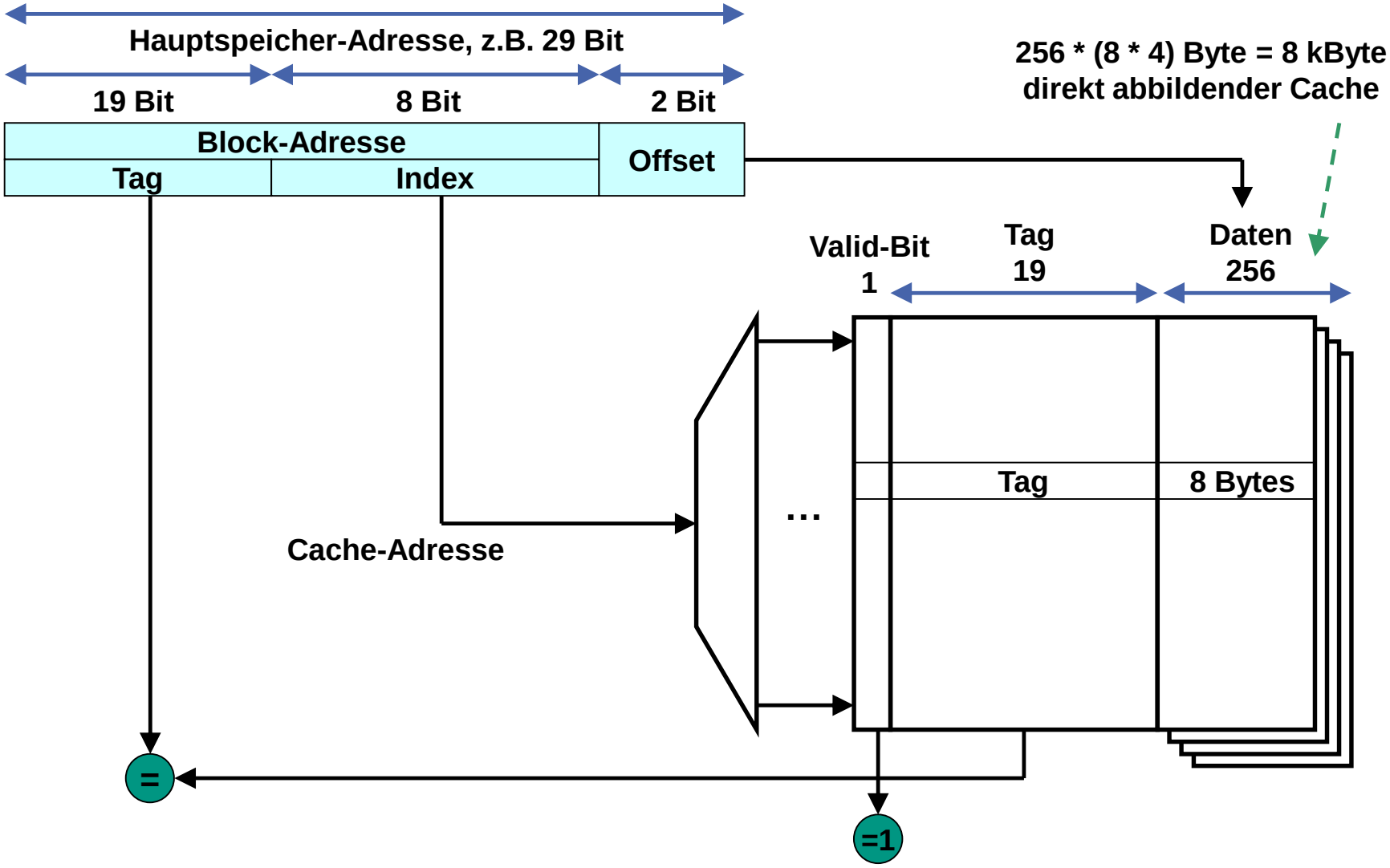
B: wie kann festgestellt werden, ob ein Block im Cache ist?

- Verfahren bei direkt abbildendem Cache
- Annahme: pro Block im Cache werden k Worte gespeichert, z.B. 4 Worte zu 8 Byte
- Offset ist die Adresse innerhalb eines Blockes, z.B. 2 Bits bei 4 Worten pro Block
- Index gibt die Adresse im Cache an



Cacheorganisation

Beispiel: direkt-abbildend



Cacheorganisation

Beispiel: direkt-abbildend

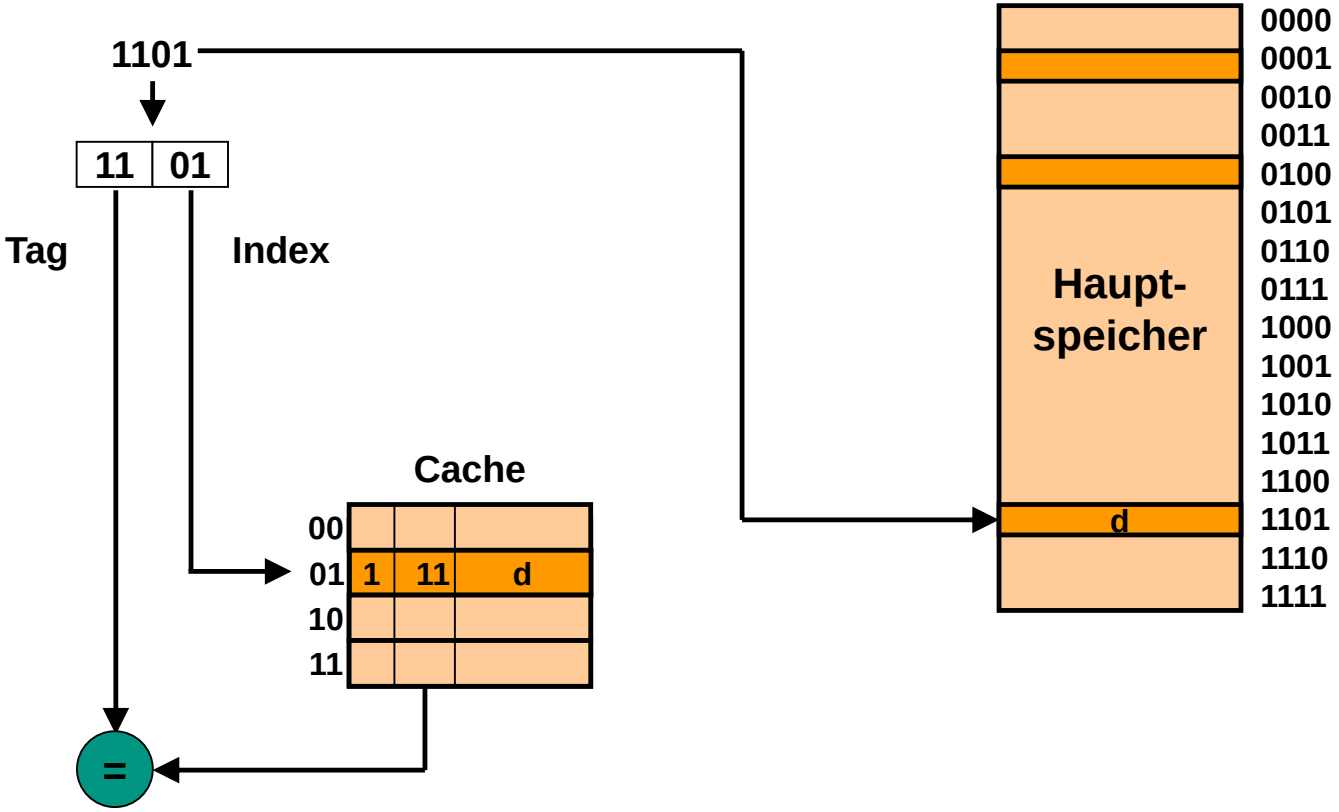
- Cache ist in Form einer Tabelle mit mehreren Datenworten pro Zeile organisiert
- Hauptspeicheradresse von insgesamt 29 Bit Breite setzt sich zusammen aus:

Offset (2 Bits):	adressiert ein Datenwort innerhalb einer Zeile, wobei jede Zeile $2^{2\ \text{Bits}} = 4$ Datenworte enthalten kann.
Index(8 Bit):	adressiert eine Zeile innerhalb des Caches, die Gesamtzahl der Zeilen beträgt $2^8 = 256$.
Tag (19 Bit):	dient als Referenz im Cache, um die Gültigkeit des aktuellen Eintrages festzustellen, ist dies nicht der Fall, werden die alten Werte im Cache bei Bedarf in den Speicher übertragen und die aktuell referenzierten Daten aus dem Speicher in den Cache geschrieben.

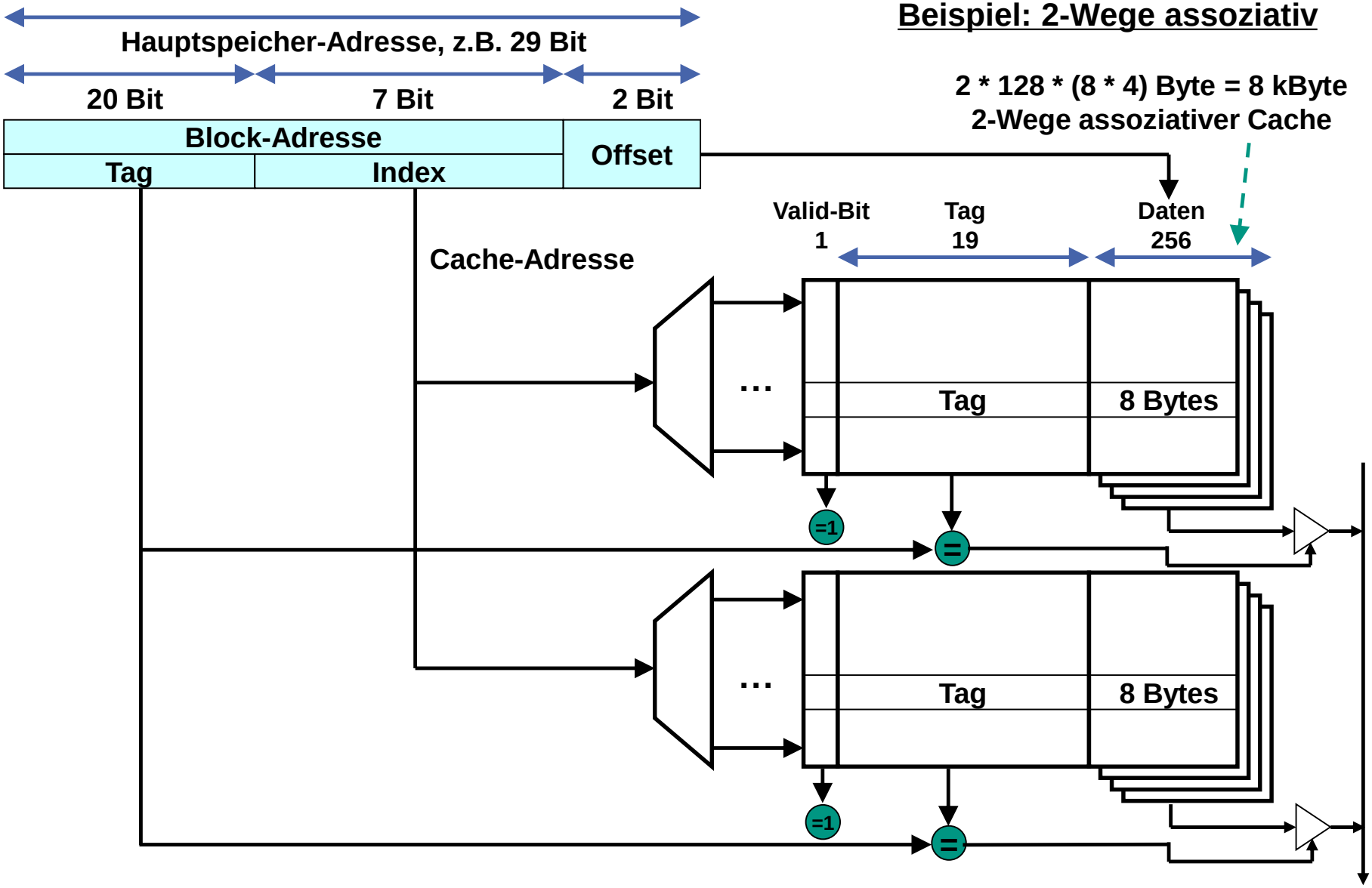
- Das Valid-Bit gibt an, ob in der betreffenden Zeile die Daten gültig sind, d.h. ob die entsprechende Zeile in Benutzung ist
- Die gesamte Größe des Caches errechnet sich aus der Anzahl der Zeilen, Anzahl der Datenworte pro Zeile and der Größe der Datenworte: 256 * (4 * 8) = 8 KBytes

Cacheorganisation

Beispiel: Blockgröße = 1



Cacheorganisation



Cacheorganisation

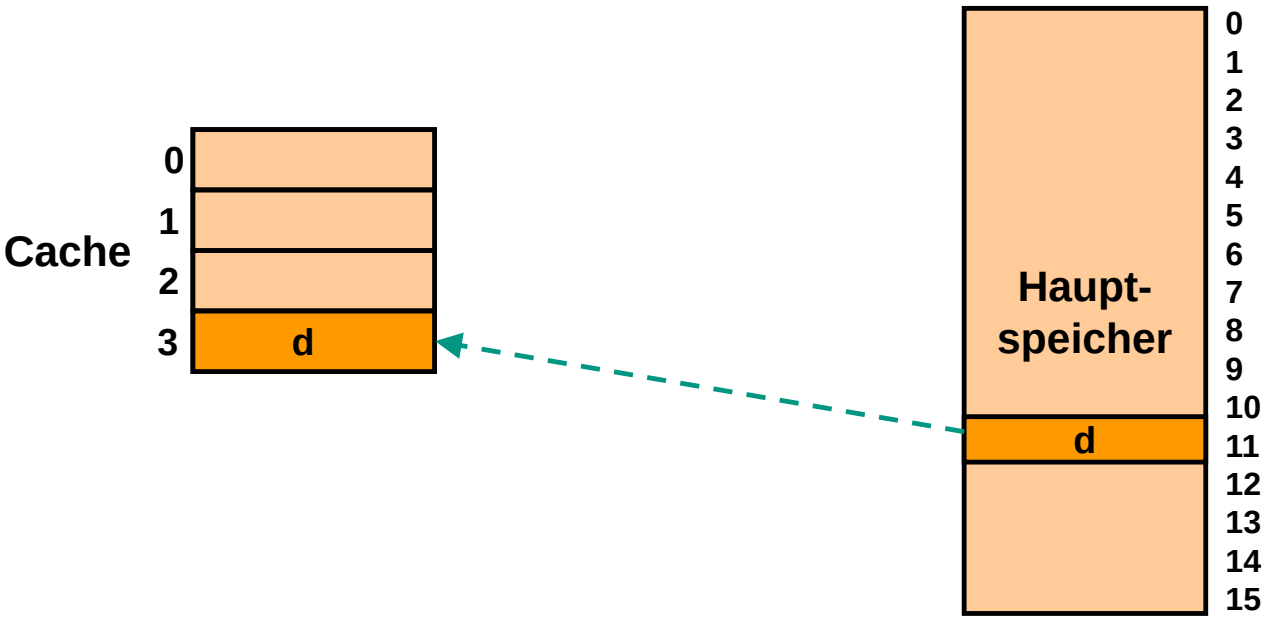
Beispiel: 2-Wege assoziativ

- Das Prinzip ähnelt dem direkt-abbildendem Cache, jedoch enthält diese Organisationsform die zweifache Implementierung
- Über einen Komparator des Tags wird entschieden welches Datenwort auf den Datenbus des Prozessor gelegt wird
- Vorteile:
  - Zwei Datenworte, die den selben Index jedoch unterschiedliche Tags in der Adresse enthalten, können gleichzeitig im Cache gepuffert werden
  - Führt je nach Programmbeschaffenheit zu einer Steigerung der Performanz
- Nachteil:
  - Größerer Aufwand beim Aufbau des Caches / höhere Kosten
- In modernen Prozessoren werden bis zu 8- bzw. 16-fach assoziative Caches eingesetzt

Cacheorganisation

C: welcher Block soll ersetzt werden, falls ein Block gebraucht wird?

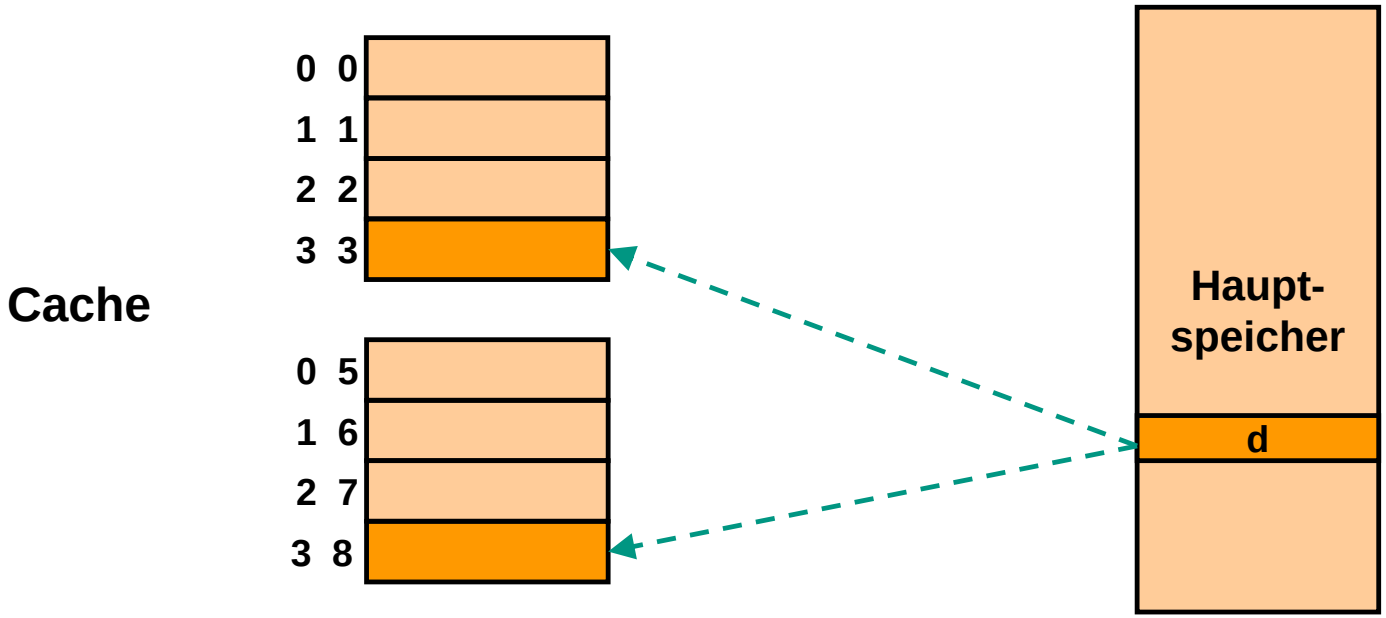
- direkt-abbildend: klar



Cacheorganisation

n-Wege assoziativ:

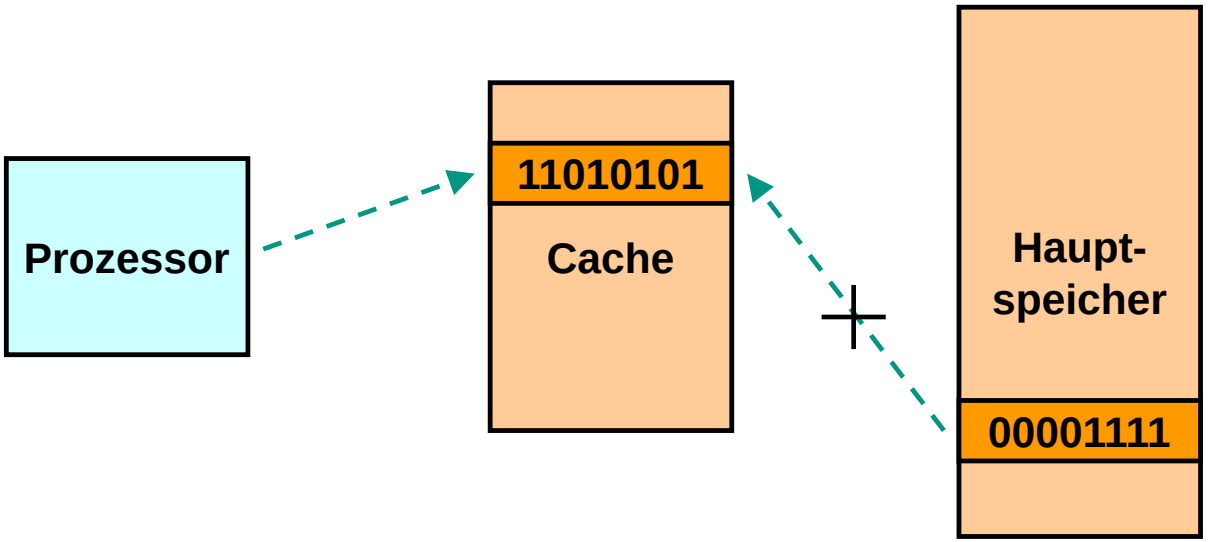
- LRU (least recently used): Versuch die zuletzt benutzten Daten zu erhalten, z.B. PowerPC 603 LRU 4-Wege teilassoziativer Cache
- zufallsgesteuert
- experimentell: LRU nur bei kleinen Caches (16kB) geringfügig (<10%) besser



Cacheorganisation

D: was geschieht beim Schreiben?

- Daten in Cache und Hauptspeicher korrespondieren einander nur solange der Prozessor nicht die Daten im Cache überschreibt
- 2 Strategien:
  - Schreiben in Cache und Speicher (write through)
  - Schreiben nur in Cache, zurückschreiben in Speicher nur bei Austausch des Blocks (write-back)



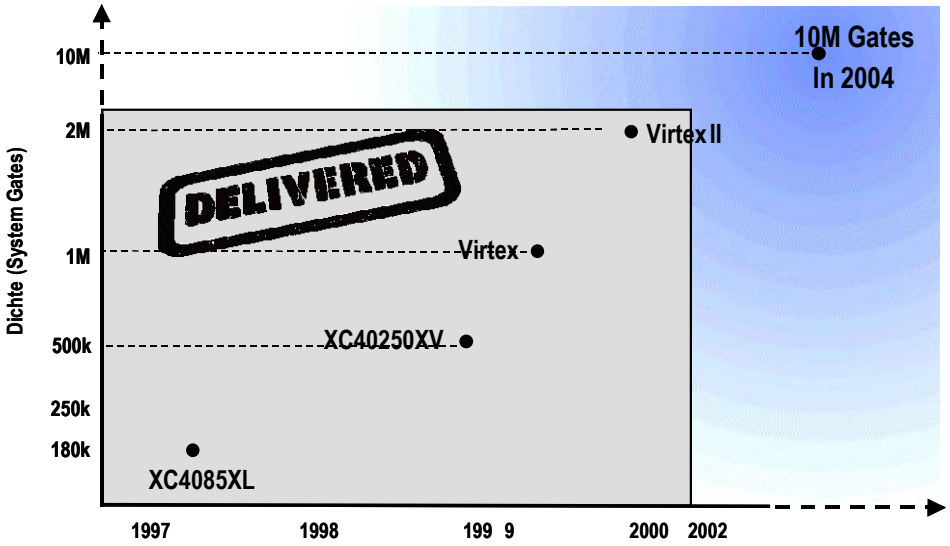
Programmierbare Logik

- ***Fuse* und *Anti-Fuse***
  - „Schmelzsicherung“ unterbricht (-> *Fuse*) oder erzeugt (-> *Anti-Fuse*) Verbindung zwischen zwei Verdrahtungslagen
  - Typische Verbindungen liegen bei 50-300 Ohm
  - **Einmalige Programmierbarkeit** (Tests vor der Programmierung?)
  - Erlaubt sehr hohe Integrationsdichte
- **EPROM und EEPROM**
  - Hoher Leistungsverbrauch
  - Typische Verbindungen liegen bei 2K - 4K Ohm
  - Ziemlich **hohe Integrationsdichte**
- **RAM-basiert**
  - Gesetzte Bits in ***Look-up-Tables*** (LUTs) realisieren Logikfunktionen
  - Gesetzte Bits in einer FlipFlop-Zelle (FF) kontrollieren Schalter, die unterschiedliche Leitungen trennen / verbinden (-> Verdrahtung)
  - Typische Verbindungen liegen bei 0.5K - 1K Ohm
  - Können **unendlich oft (re-) programmiert** werden (auch teilweise dynamisch, während des Betriebs)
  - Geringe Integrationsdichte

Rekonfigurierbare Hardware

- **PALs, PLAs:**
  -> 10 - 100 Gatter-Äquivalente

- **<u>Feinkörnig:</u> FPGAs, FPLDs**
  - Altera MAX Family
    -> FPLD (*EEPROM*)
  - Actel Programmable Gate-Array
    -> FPGA (*Anti-Fuse*)
  - Xilinx Logical Cell Array
    -> FPGA (*RAM-basiert*)

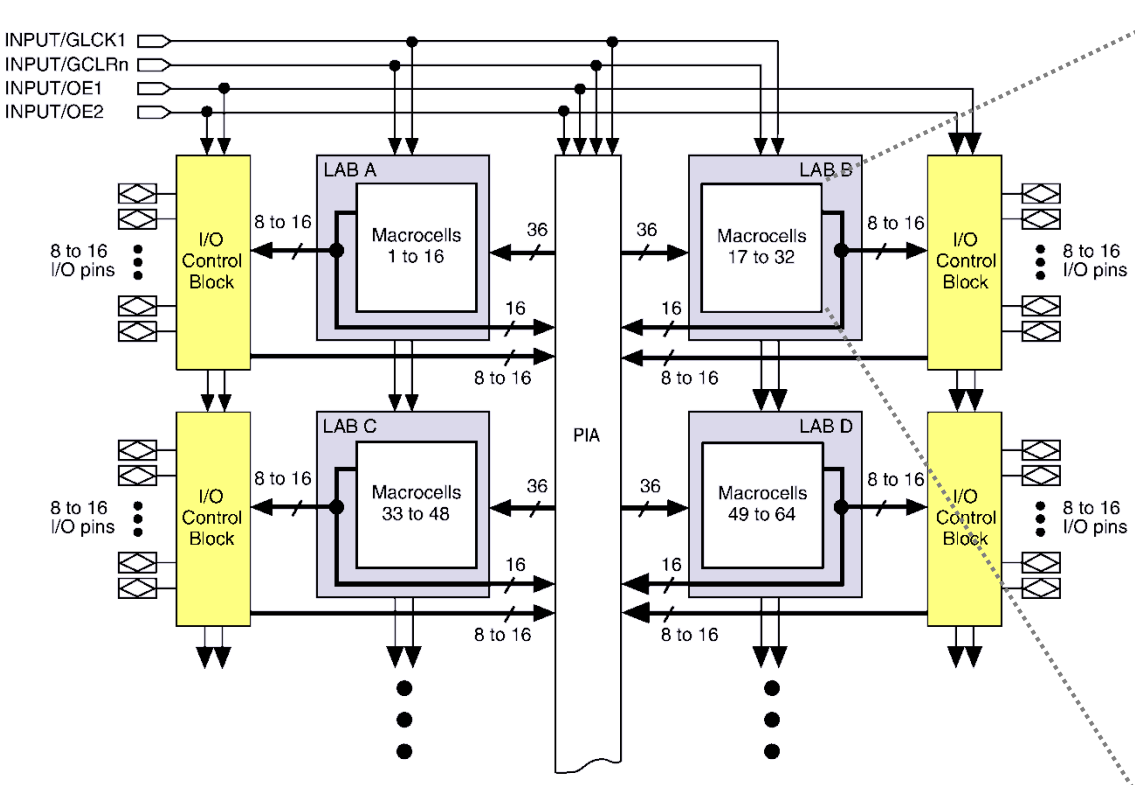


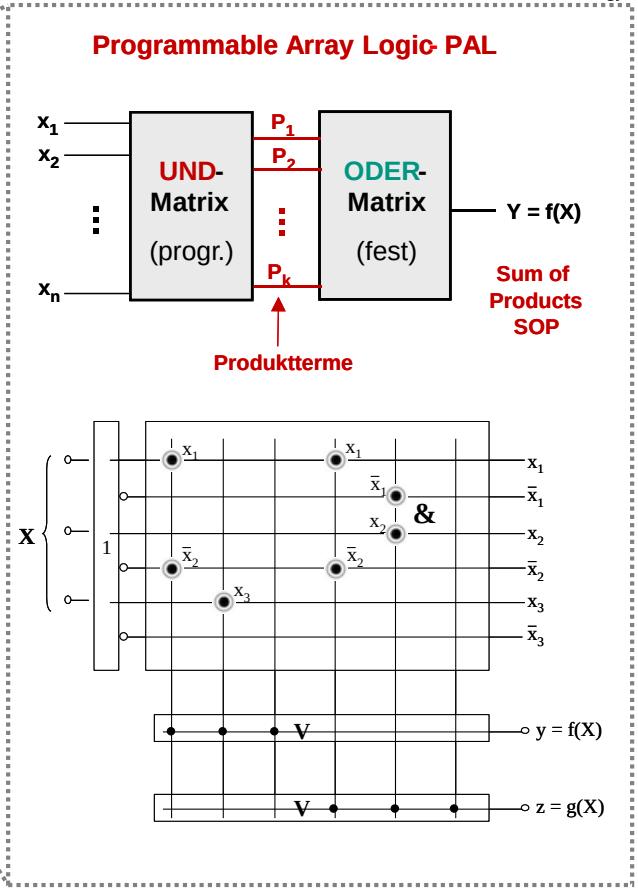
10 Million System Gates in 2002!

- Einige Tausend bis mehrere Millionen Gatter-Äquivalente
  -> z. B Xilinx Virtex XCV2000

- **<u>Grobkörnig:</u>** Field Programmable Functional Arrays = FPFAs

Feingranular Rekonfigurierbare Hardware: *Altera* MAX 7000 CPLD





– **L**ogic **A**rray **B**locks (LAB), I/O Blocks, und **P**rogrammable **I**nterconnect **A**rray (PIA)

– **Programmierbarkeit** basiert auf ***EEPROM*** Floating-Gate (*Flash*) Technology (nicht-flüchtig)

– 16 Makrozellen in jedem LAB

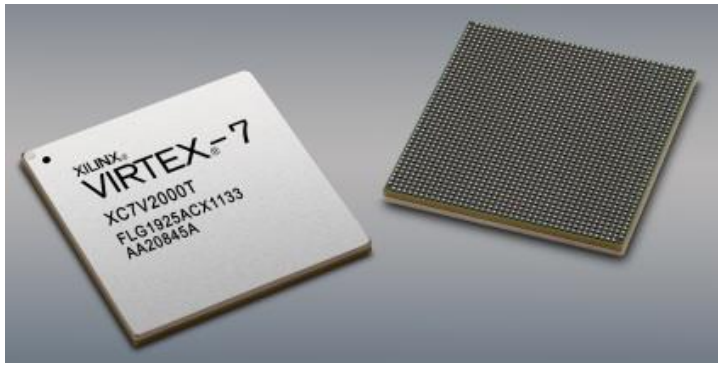
  • Flip-Flops + kombinatorische Basis-Logik

• *Programmierbares Interconnect-Array* (***PIA***) enthält Busse, welche die Eingänge/ Ausgänge der Makrozellen verbinden

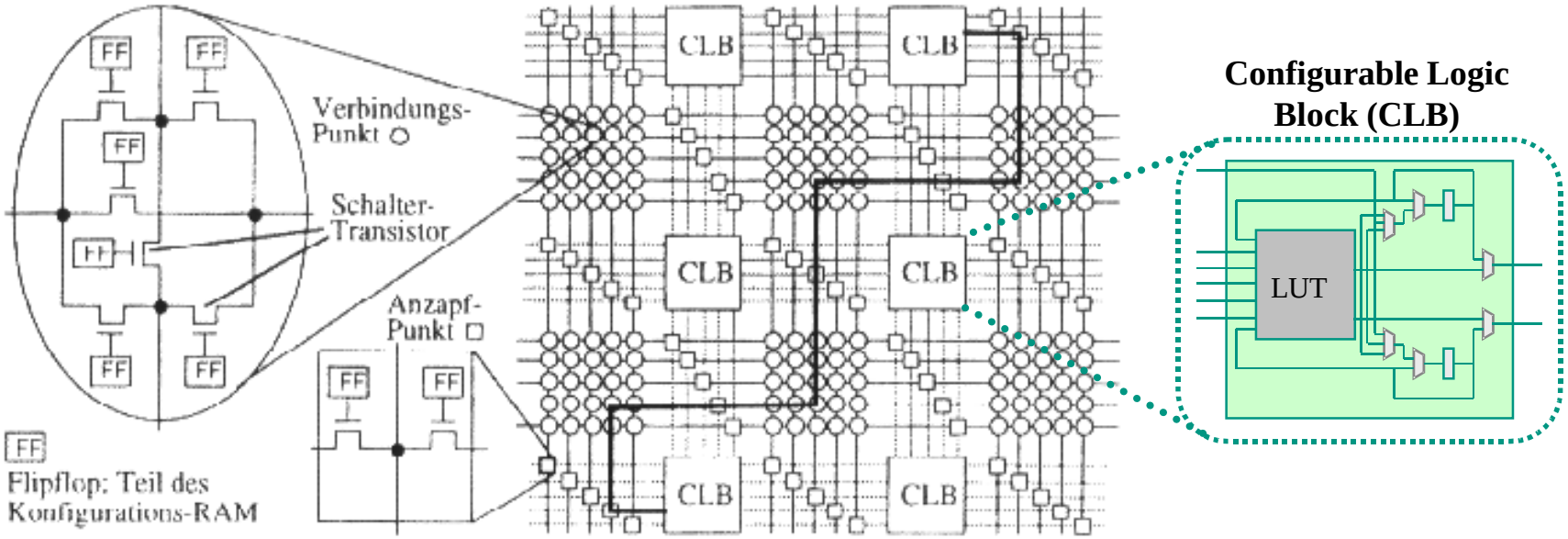
• I/O Control-Blöcke verbinden alle benachbarten LABs sowie extern zu Pins

Feingranular Rekonfigurierbare Hardware: *Xilinx* FPGA

- FPGA-Architekturen
  - SRAM-basierende Look-up Tables (LUTs)
  - Probleme:
    - Verdrahtung: reduziert Performanz
    - Verhältnis: aktive / passive Elemente

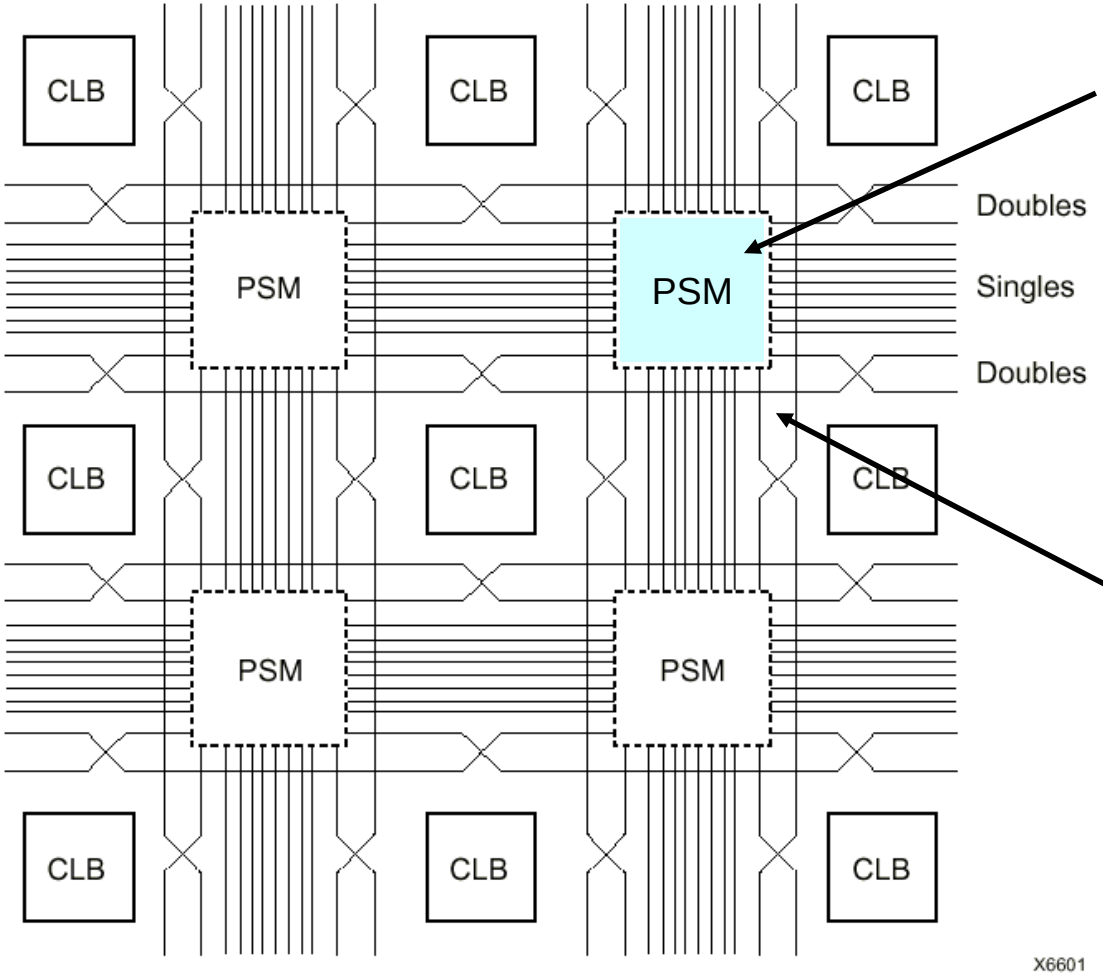


rekonfigurierbare Verbindungen (*Switching Boxes*)



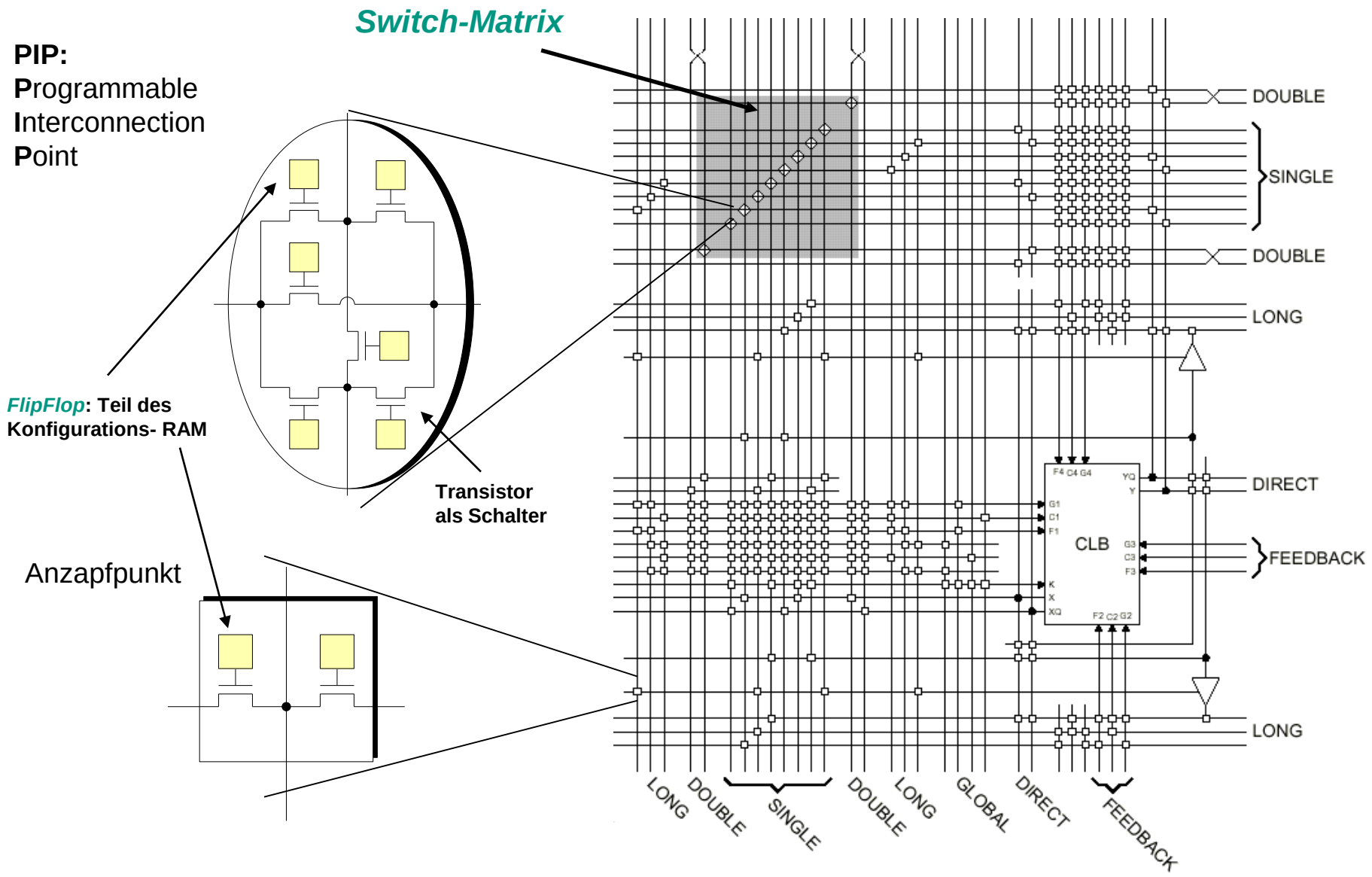
Source: R. Hartenstein

Xilinx FPGA: Switch-Matrix (I)

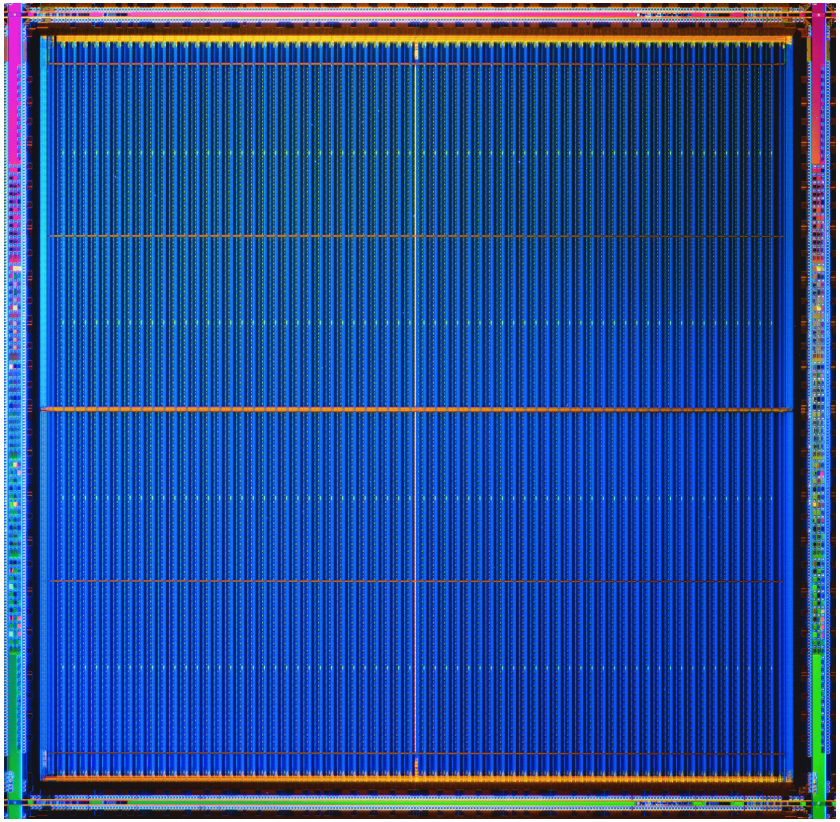


- PSM = Programmable Switch Matrix
- 10 Verbindungspunkte pro Matrix
- Jede Verbindung enthält 6 Pass- Transistoren für eine vollständige Ver-bindung in alle Richtungen
- Erlaubt Verbindung zwischen Single und Double Leitungen
  - Single: Verbinden benachbarte PSM
  - Double: Verbinden zwei PSM mit 1 unverbundenen PSM in der Mitte

Xilinx FPGA: Switch-Matrix (II)



Xilinx FPGA – Beispiel: *Virtex* Familie



Xilinx Virtex XCV 1000

- CMOS: 2.5 V, 0.22 µm ,
  5 Metallebenen CMOS Prozeß
- CLB Matrix 64 x 96 = 6144
- f_{clock} bis 200 MHz (abh. von Applikation)
- Max. Anzahl Logik Gatter 1,124,022
- Max. Anzahl RAM Bits 393,216
- Benötigte Konfigurationsbits 686,184
- 5.0 V TTL kompatibles I/O
- 5.0 V und 3.0 V PCI fähiges I/O
- Max. I/O 512 , Unterstützung
  von 20 I/O Standards
- Dual-Port RAM Option
- LUTs als 16 / 32 Bit RAM konfigurierbar
- 4 primäre und 24 sekundäre Low-Skew
  Takt oder Signal- Verteilungsnetzwerke
- FPGA- Programmierung kann auch
  zurückgelesen werden, zur Programm-
  verifikation + Beobachtung interner Knoten
- Dedizierte High-Speed Carry Logik
- Interne Tri-State-Bus Fähigkeit

Zusammenfassung

- Wichtige Rechnerkomponenten und deren „Zusammenspiel“
- Unterschiedliche von Neumann / Harvard basierte Rechnerarchitekturen
  - Akkumulatormaschine
  - Stackmaschine
  - Registersatzmaschine (CISC, RISC)
- Befehlsausführungszyklus
  - Pipelining-Prinzip
- Speicherorganisation und Hierarchie
  - Cachehierarchie
  - Cachetypen
- Spezialarchitekturen
  - Rekonfigurierbare Hardware: PLDs, FPGAs