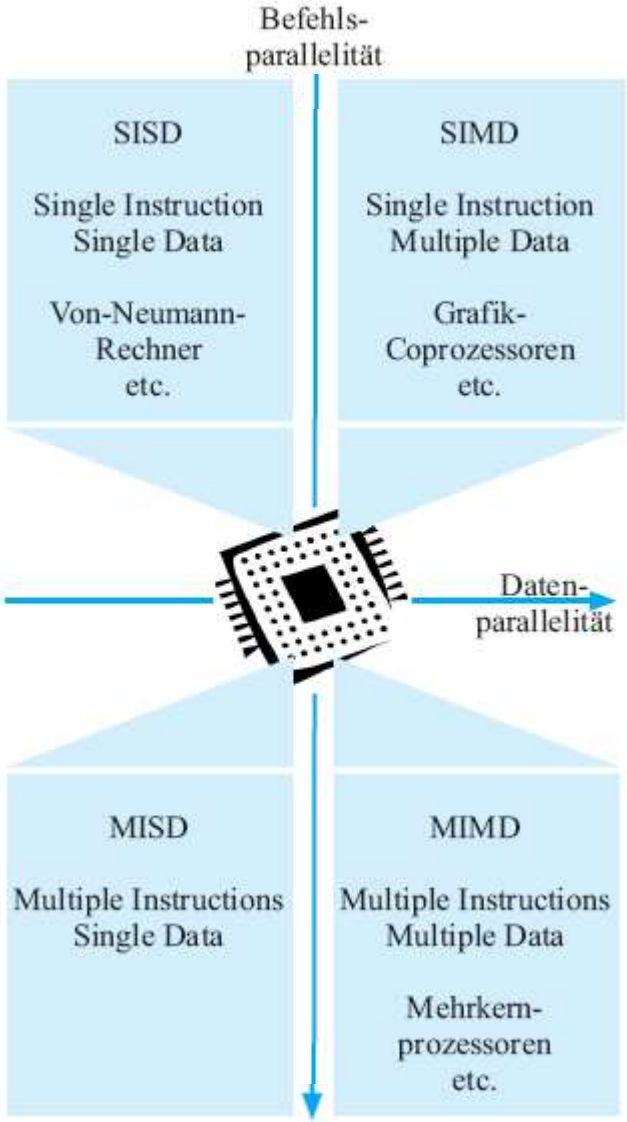
8. Rechnerstrukturen

- **bisher**
  - extrem einfacher Modellprozessor
- **moderne Prozessoren**
  - große Zahl von unterschiedlichen Typen
  - interner Aufbau unterscheidet sich erheblich
  - Ordnungsprinzipien, um Vielfalt übersichtlicher zu gestalten
- **Taxonomie nach Flynn**
  - Taxonomie: Klasseneinteilung
    - griechisch: *taxis* "Ordnung", *nomia* "Verwaltung"
  - Flynn schlug 1966 ein einfaches Modell zur Einteilung der Computer vor, das heute immer noch benutzt wird
  - Klassifikationsmerkmal
    - Anzahl der Daten und Befehlsströme, die gleichzeitig verarbeitet werden

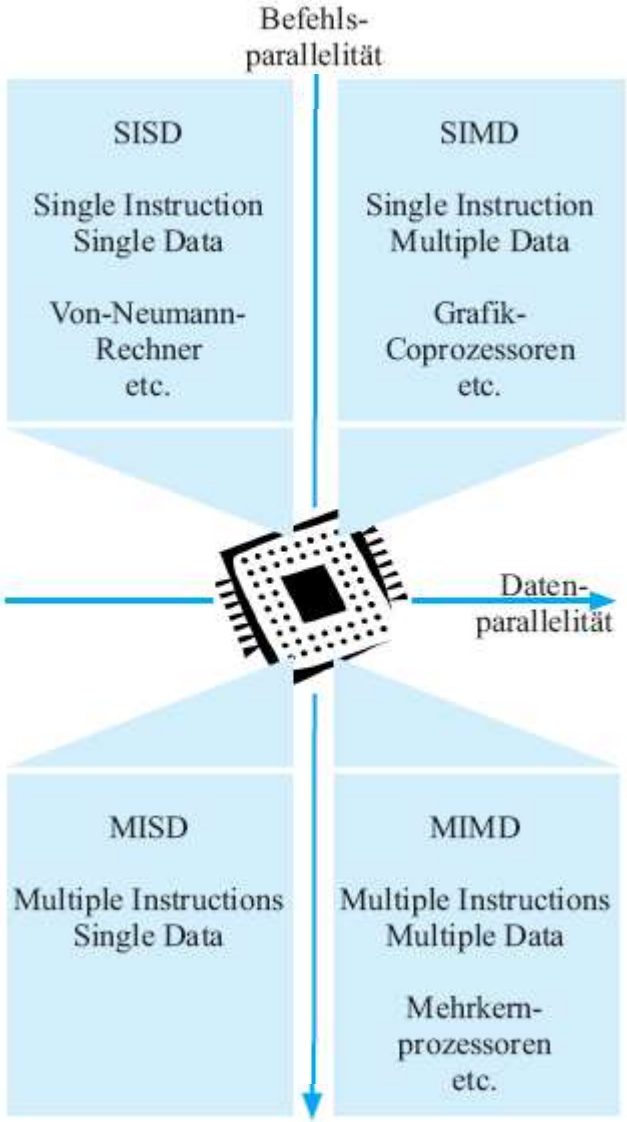
Rechnerklassifikation nach Flynn

- **Single Instruction stream, Single Data stream**
  - SISD (sprich: „sisdi“)
  - normaler Einzelprozessor
- **Single Instruction stream, Multiple Data streams**
  - SIMD (sprich: „simdi“)
  - dieselbe Instruktion steuert mehrere Rechenwerke, die ihre eigenen Datenströme verarbeiten
  - Beispiele
    - Multimedia Extensions
    - Vektorcomputer
    - Grafikprozessoren



Einteilung der parallelen Architekturen (2)

- **Multiple Instruction streams, Single Data stream**
  - MISD
  - z.B. fehlertolerante Systeme, die dieselben Daten mit mehreren Funktionseinheiten bearbeiten und Ergebnisse vergleichen
  - FPGA basierte Systeme (Daten fließen nacheinander durch mehrere programmierbare Funktionseinheiten)
- **Multiple Instruction streams, Multiple Data streams**
  - MIMD
  - jeder Prozessor führt sein eigenes Programm aus und verarbeitet dabei seine eigenen Daten
  - z.B. Mehrkernprozessoren



Instruktionsarchitekturen

- **grobe Einteilung**
  - CISC (Complex Instruction Set Computer)
    - umfangreicher Befehlssatz
    - Programme können mit relativ wenigen Maschinenbefehlen realisiert werden
    - intern werden die Maschinenbefehle in mehrere Mikroinstruktionen zerlegt
    - mehr oder weniger viele Takte zur Bearbeitung eines Befehls
  - RISC (Reduced Instruction Set Computer)
    - Befehlssatz besteht nur aus relativ wenigen elementaren Maschinenbefehlen
    - Instruktionen können in einem einzigen Takt sehr effizient ausgeführt werden
    - überraschend ist, dass RISC Prozessoren insgesamt schneller sind als CISC Prozessoren
- **Details: siehe Vorlesung Rechnerorganisation**

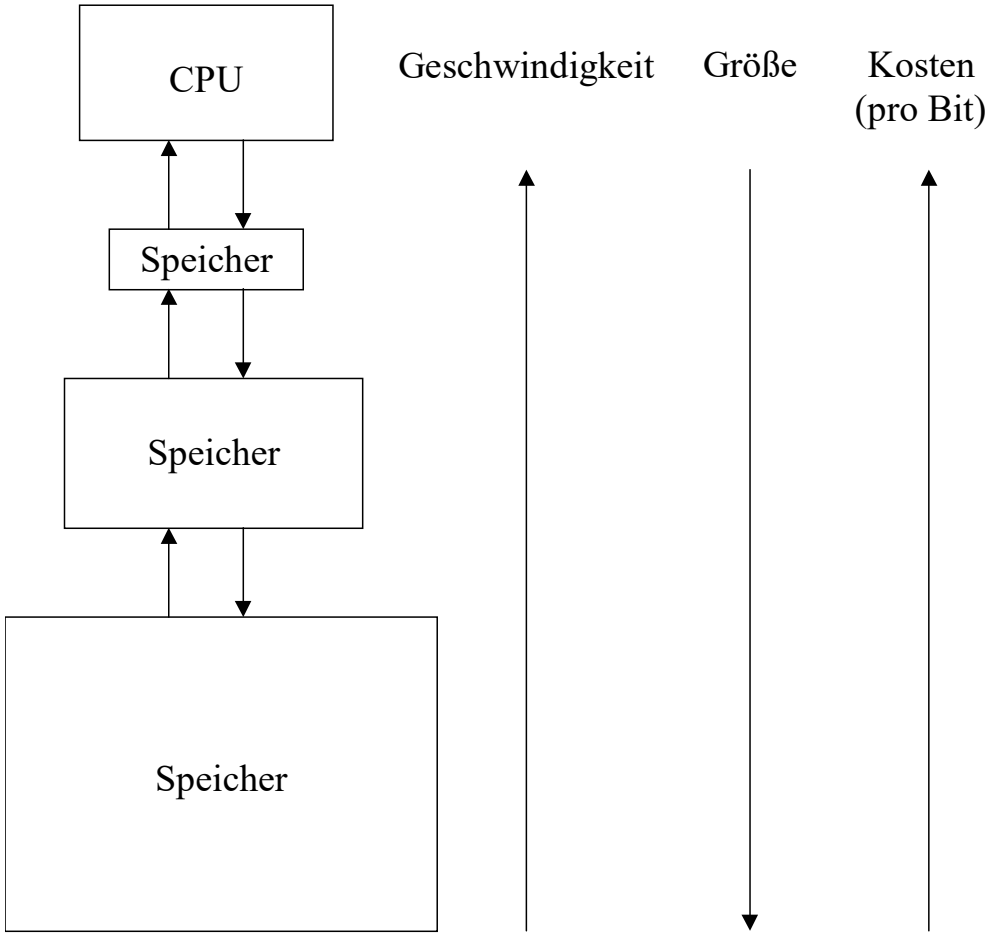
Methoden zur Leistungssteigerung

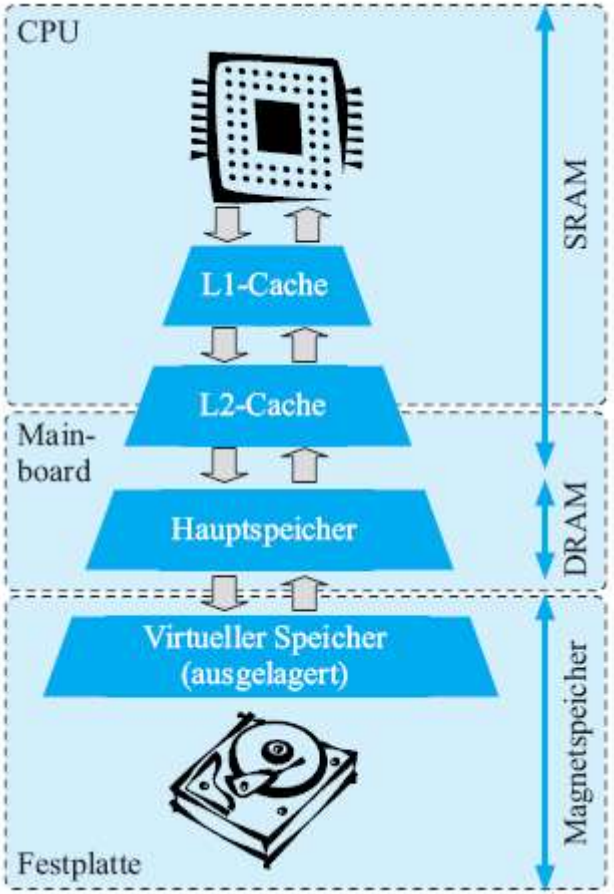
- **Cache**
  - Arbeitsgeschwindigkeit moderner CPUs ist erheblich größer als die Geschwindigkeit der Hauptspeicher
  - kleiner, schneller Zwischenspeicher
    - Anzahl Zugriffe auf großen, langsamen Hauptspeicher wird reduziert
- **Pipelining**
  - Idee wurde in der industriellen Produktion durch Henry Ford Anfang des 20. Jahrhunderts eingeführt (Fließbandproduktion)
    - Produktion eines Autos wird in kleine Schritte unterteilt und an aufeinander folgenden Arbeitsstationen ausgeführt
      - jede Station ist hoch effizient, da immer die gleichen Tätigkeiten ausgeführt werden
      - Produktion wird parallelisiert, da z.B. ein Auto lackiert wird, während bei einem anderen gerade gleichzeitig der Motor eingebaut wird

Cache

- **Problem**
  - Arbeitsgeschwindigkeit moderner CPUs ist erheblich größer als die der Hauptspeicher
    - Hauptspeicher ist räumlich entfernt von der CPU
    - Hauptspeicher ist langsames DRAM
      - SRAM wäre viel schneller als DRAM, ist aber sehr viel teurer
  - Folge: CPUs werden durch Speicherzugriffe ausgebremst
- **Lösung**
  - Cache-Speicher
    - kleiner, aber extrem schneller SRAM-Speicher
    - meist auf dem Chip des Mikroprozessors, also in räumlicher Nähe
    - dient nur als kleiner Zwischenspeicher für den großen Hauptspeicher
  - es entsteht eine Speicherhierarchie

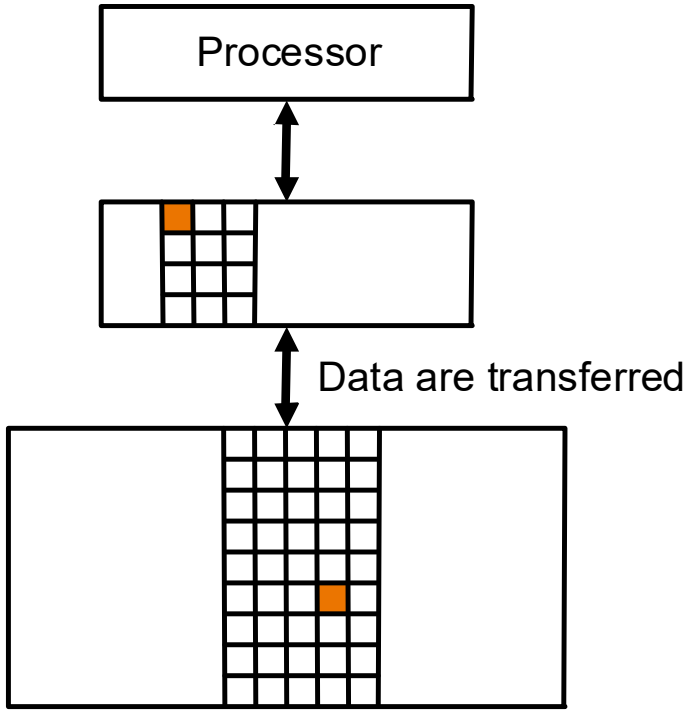
Speicherhierarchie





Obere und untere Ebene

- wir betrachten zwei benachbarte Ebenen (obere und untere)
- Daten werden zwischen oberer und unterer Ebene in Blöcken ausgetauscht



oberer Speicher

unterer Speicher

Begriffe

- **Block**
  - die kleinste Datenmenge, die in einer Ebene vorhanden oder nicht vorhanden sein kann
- ***hit* (Treffer)**
  - das angeforderte Datum befindet sich in einem Block des oberen Speichers
- ***miss* (Fehlwurf)**
  - das angeforderte Datum, befindet sich in keinem Block des oberen Speichers
  - der untere Speicher wird dann nach der Information befragt

Begriffe (2)

- ***hit rate*, *hit ratio* (Trefferquote)**
  - Bruchteil der Speicherzugriffe, die auf der oberen Ebene Erfolg haben
- ***miss rate*, *miss ratio* (1-Trefferquote)**
  - Bruchteil der auf der oberen Ebene erfolglosen Speicherzugriffe
- ***hit time***
  - Gesamtzeit für einen erfolgreichen Speicherzugriff
    - also Daten befinden sich tatsächlich im oberen Speicher (ein hit)
  - einschließlich der Zeit zum Feststellen, dass es sich um einen *hit* handelt
- ***miss penalty* (Strafe, Kosten für Fehlwurf)**
  - Gesamtzeit für das Ersetzen eines Blocks im oberen Speicher durch einen Block aus dem unteren Speicher plus der Übertragungszeit in die obere Ebene

Prinzip der Lokalität in Programmen

- **zeitliche Lokalität**
  - wenn eine Speicherzelle benutzt wird, ist die Wahrscheinlichkeit groß, dass sie bald noch einmal gebraucht wird, z.B.
    - Programmcode enthält viele Schleifen und häufig benutzte Unterprogramme
    - Daten werden gelesen, manipuliert und wieder geschrieben, dieselben Variablen werden für viele Operationen benutzt
- **räumliche Lokalität**
  - wenn eine Speicherzelle benutzt wird, ist die Wahrscheinlichkeit groß, dass bald auch eine Speicherzelle mit einer in der Nähe liegenden Adresse gebraucht wird, z.B.
    - Programmcode wird im wesentlichen sequentiell abgearbeitet
    - Daten liegen logisch gruppiert im Speicher vor (array's, struct's, Objekte)

Ausnutzen der Lokalität

- **zeitliche Lokalität**
  - einmal benutzte Daten werden im schnellen Speicher nahe der CPU zwischengespeichert und dort gehalten
    - schneller Zugriff, wenn sie noch einmal benötigt werden
- **räumliche Lokalität**
  - Daten werden zwischen den Hierarchie-Stufen in großen Blöcken transferiert
    - höherer Durchsatz als bei Transfer einzelner Worte (siehe z.B. Nibble Mode beim DRAM)
    - anschließend schneller Zugriff auch auf benachbarte Daten

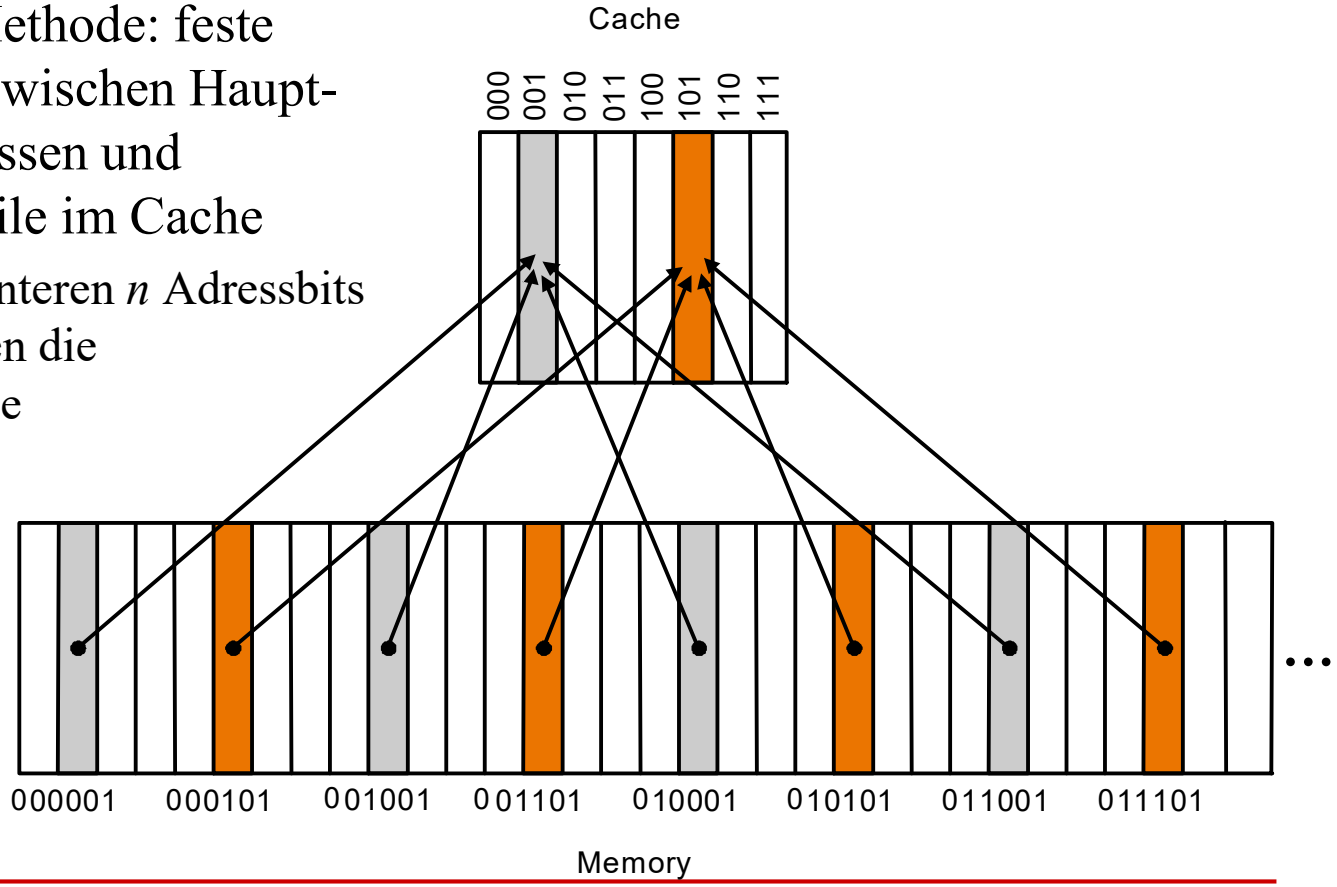
Cache

- engl.: Versteck, geheimes Lager
- zwei Bedeutungen
  - Speicher zwischen CPU und Hauptspeicher
  - jeder Speicher, der die Lokalität von Zugriffen ausnutzt, z.B.
    - Festplatten-Cache
    - Software-Cache, der Rückgabewerte von Funktionen zwischenspeichert

- **Probleme, die gelöst werden müssen**
  - wie weiß man, ob sich ein Datenelement bereits im Cache befindet?
  - falls es sich dort befindet: wie findet man es?

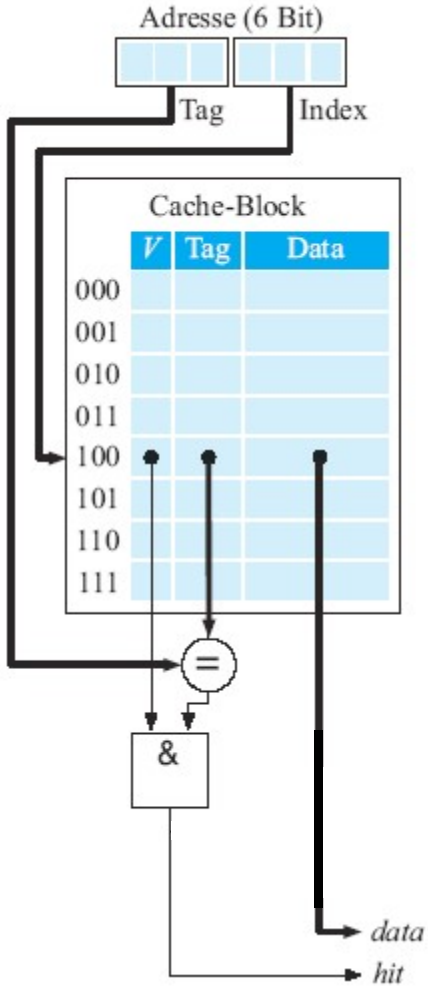
Direkt abgebildeter Cache

- ***Direct Mapped Cache***
  - viele Adressen aus dem Hauptspeicher müssen sich dieselbe Cachezeile teilen
  - einfachste Methode: feste Zuordnung zwischen Hauptspeicheradressen und benutzter Zeile im Cache
    - z.B. die unteren *n* Adressbits adressieren die Cachezeile



Direkt abgebildeter Cache (2)

- die untersten *n* Bits der Adresse werden als Zeilennummer verwendet
- die restlichen Bits der Adresse werden zusammen mit den Daten im Cache abgelegt
  - dieses *Tag* (engl. Etikett) zusammen mit der Zeilennummer identifiziert die Daten im Cache eindeutig
- beim Lesen wird das *Tag* mit dem oberen Teil der Adresse verglichen, um einen *Hit* oder einen *Miss* festzustellen
- Valid-Bit *V* zeigt an, ob die Cache-Zeile überhaupt gültige Daten enthält
  - nach Einschalten: alle *V*=0



Ablauf

- **Lesezugriff**
  - bei jedem Zugriff auf den Hauptspeicher sieht der Controller zunächst nach, ob sich die Daten schon im Cache befinden
    - wenn ja (*Cache-Hit*) werden die Daten ohne Verzögerung extrem schnell aus dem Cache gelesen
    - wenn nein (*Cache-Miss*) werden die Daten aus dem Hauptspeicher gelesen und eine Kopie im Cache abgelegt
      - sollten die Daten später noch einmal gebraucht werden, können sie dann sehr schnell aus dem Cache (Cache-Hit) gelesen werden
- **Schreibzugriff**
  - Daten werden in den schnellen Cache geschrieben
  - Konsistenz von Hauptspeicher- und Cache-Inhalten muss aber sichergestellt werden

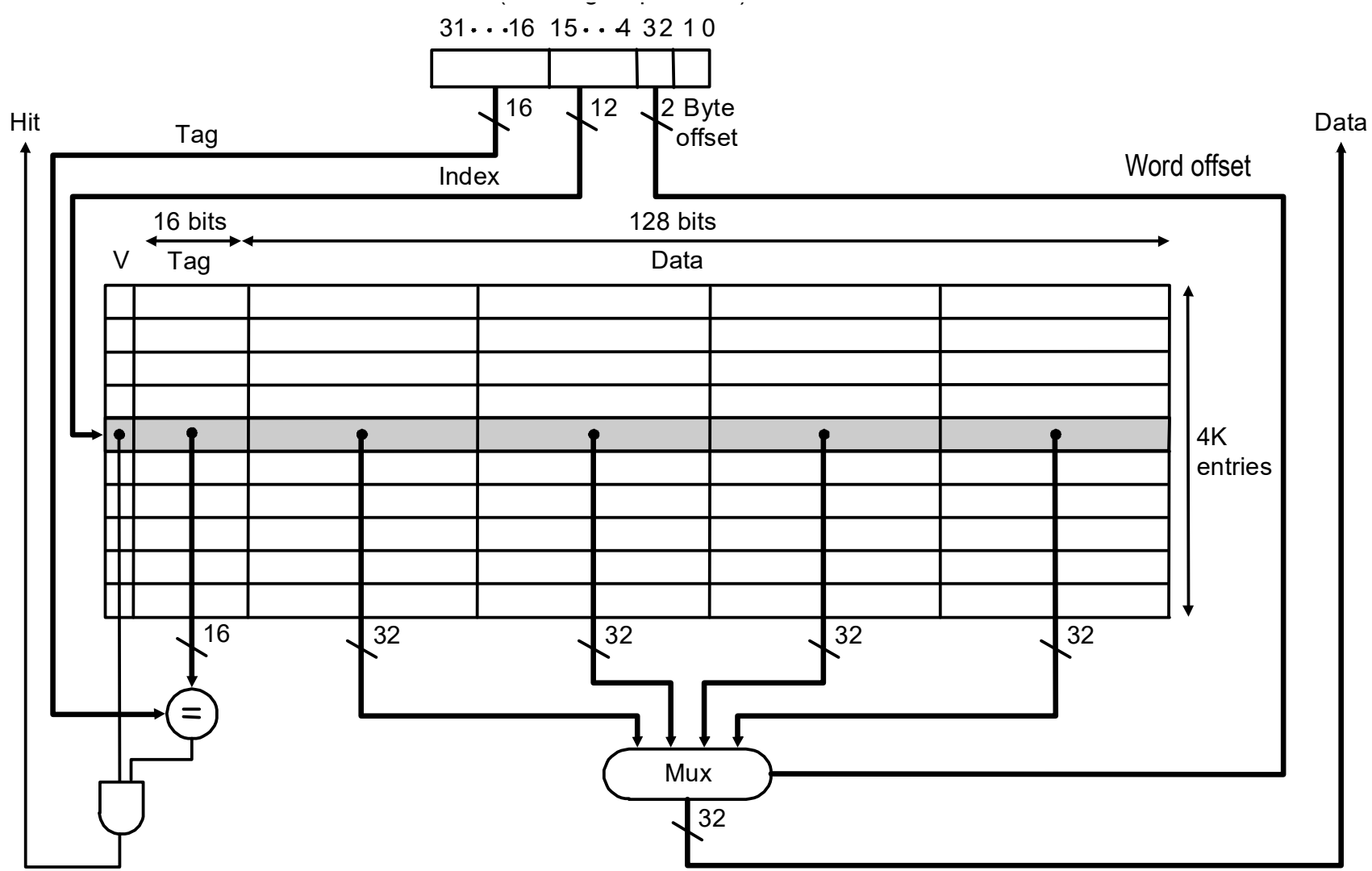
Datenkonsistenz beim Schreiben

- zwei Möglichkeiten, Cache-Inhalt und Hauptspeicher konsistent zu halten
  - *Write-Through* Strategie
    - die Daten werden beim Schreiben in den Cache auch sofort in den Hauptspeicher geschrieben
  - *Write-Back* Strategie
    - die Daten werden erst dann zurück geschrieben, wenn der Platz im Cache für andere Daten benötigt wird
      - » Dirty-Bit notwendig, um anzuzeigen, dass Cache-Inhalt nicht mit Hauptspeicher übereinstimmt
      - » Ist die Cache-Zeile nicht dirty, muss gar nicht zurückgeschrieben werden
    - ist das effizientere Verfahren
    - aufwändiger, insbesondere bei Mehrprozessorsystemen, bei denen jeder Prozessor seinen eigenen Cache hat

Ausnutzung der räumlichen Lokalität

- **Direct Mapped Cache**
  - Block umfasst ***mehrere*** Worte
    - z.B. 1 Block = 4 Worte (im Beispiel 1 Wort = 4 Byte)
  - Daten werden immer als ganzer Block aus dem Hauptspeicher gelesen oder dorthin zurückgeschrieben
  - Zugriff auf Worte über
    - Byte offset (Byteadresse innerhalb eines Wortes, 2 Bits)
      - immer 00 beim Zugriff auf Worte, niederwertigste Bits
    - Word offset (Wortadresse innerhalb Block, 2 Bits)
      - niederwertige Bits
    - Index (Blockadresse im Cache, Cachezeile)
      - mittlere Bits
    - Tag (Blockadresse im Hauptspeicher)
      - höherwertige Bits
      - Vergleich mit gespeichertem Tag, um *hit/miss* festzustellen

Beispiel MIPS-Prozessor



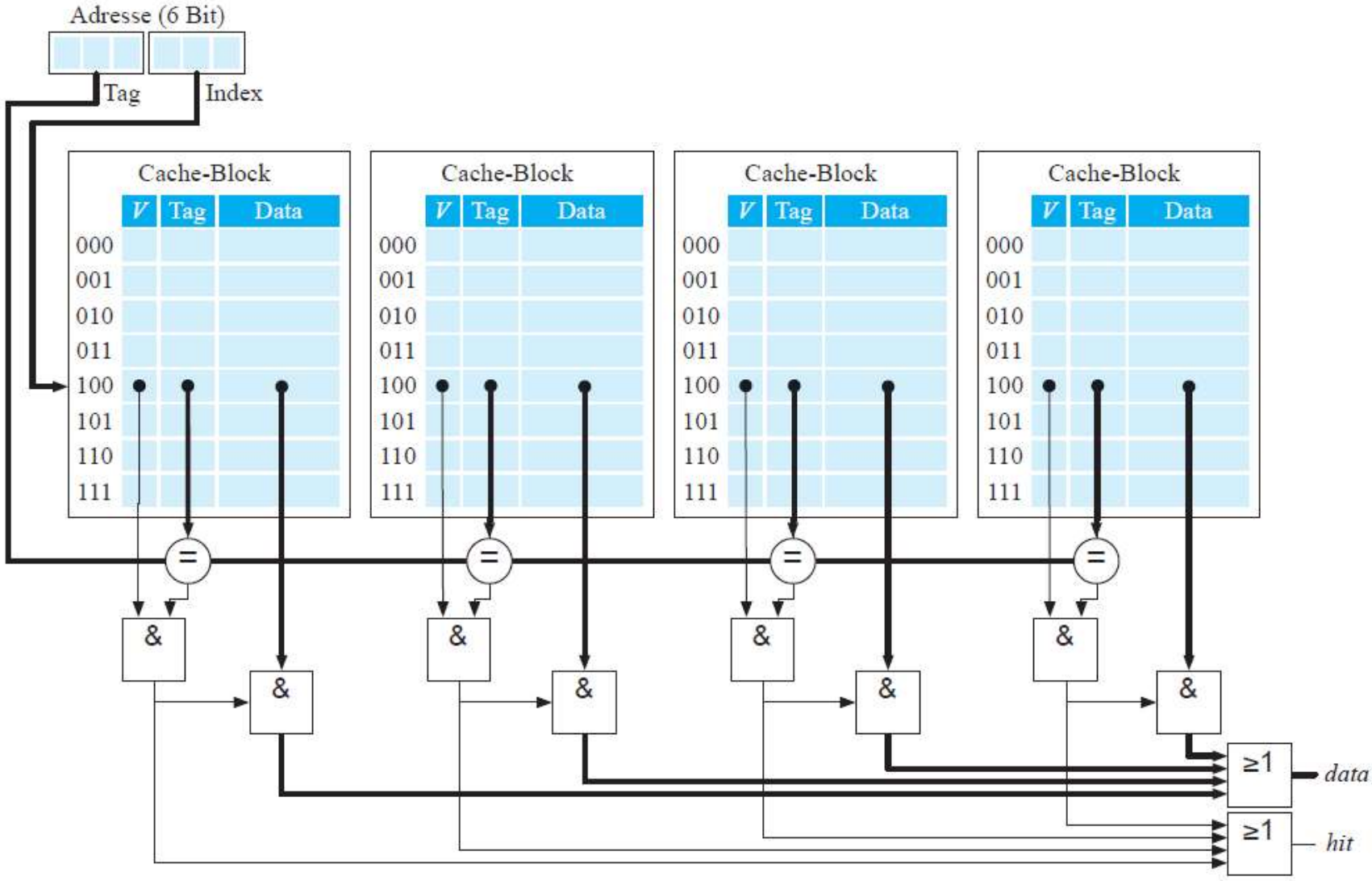
Hits vs. Misses

- ***read***
  - *read hit*: Wort aus Cache lesen
  - *read miss*
    - bei *write back* und gesetztem *dirty bit*: alten Block in Hauptspeicher schreiben
    - neuen Block aus Hauptspeicher lesen
- ***write***
  - es reicht nicht, *tag* und Wort zu schreiben, da die anderen drei Worte zu einer anderen Adresse gehören können
    - auch beim Schreiben *tag* mit Adresse vergleichen (wie beim Lesen)
  - *write hit*: in Cache schreiben (später: *write back*)
  - *write miss*: vor Schreiben Block aus Speicher laden
    - bei *write-back* und gesetztem *dirty bit* Block vorher sogar noch in den Hauptspeicher zurückschreiben

Weitere Cache-Varianten

- **Voll assoziativer Cache**
  - Blöcke können überall im Cache stehen
  - sehr aufwändig
- **Mehrwege Cache (Mehrfach assoziativer Cache)**
  - Blöcke können an mehreren Stellen in Cache stehen
  - Kompromiss zwischen Aufwand und Nutzen
  - meist verwendete Variante
  - verringert die Anzahl der Misses
  - Neues Problem
    - Welcher der Blöcke soll verdrängt werden?
    - Verschiedene Strategien (siehe Vorlesung Rechnerorganisation)

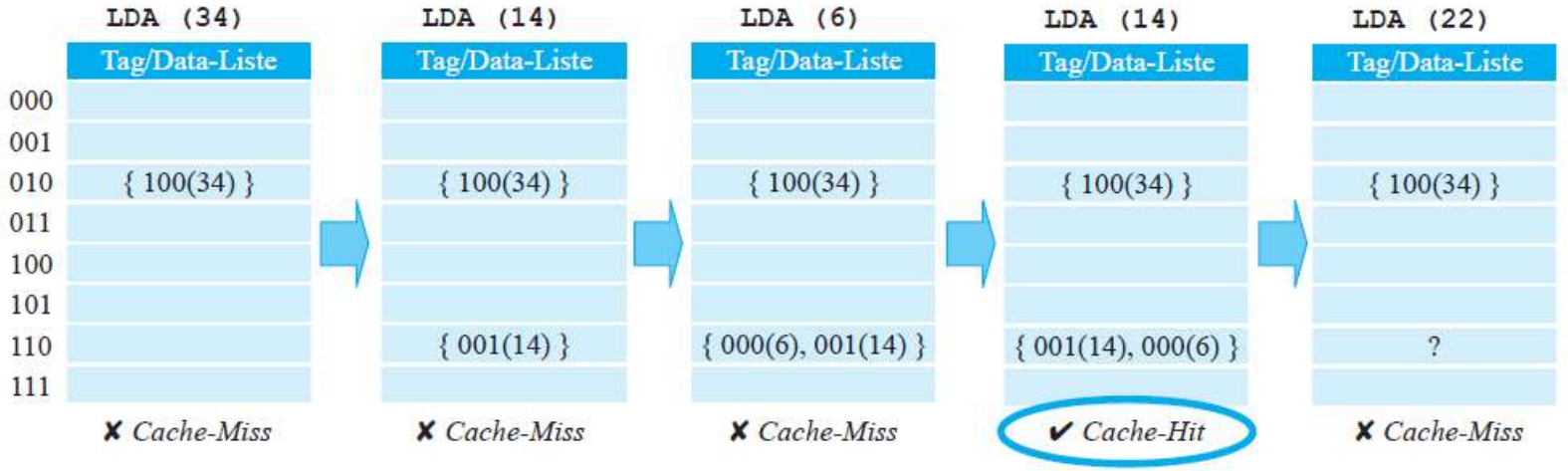
Beispiel: 4-Wege Cache



Vergleich: Direct Mapped vs. 2-Wege Cache

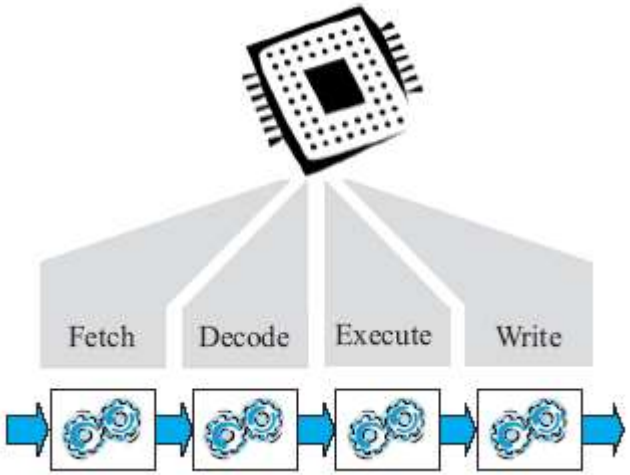
- Direkt abgebildeter Cache-Speicher:

- Zweifach assoziativer Cache-Speicher:



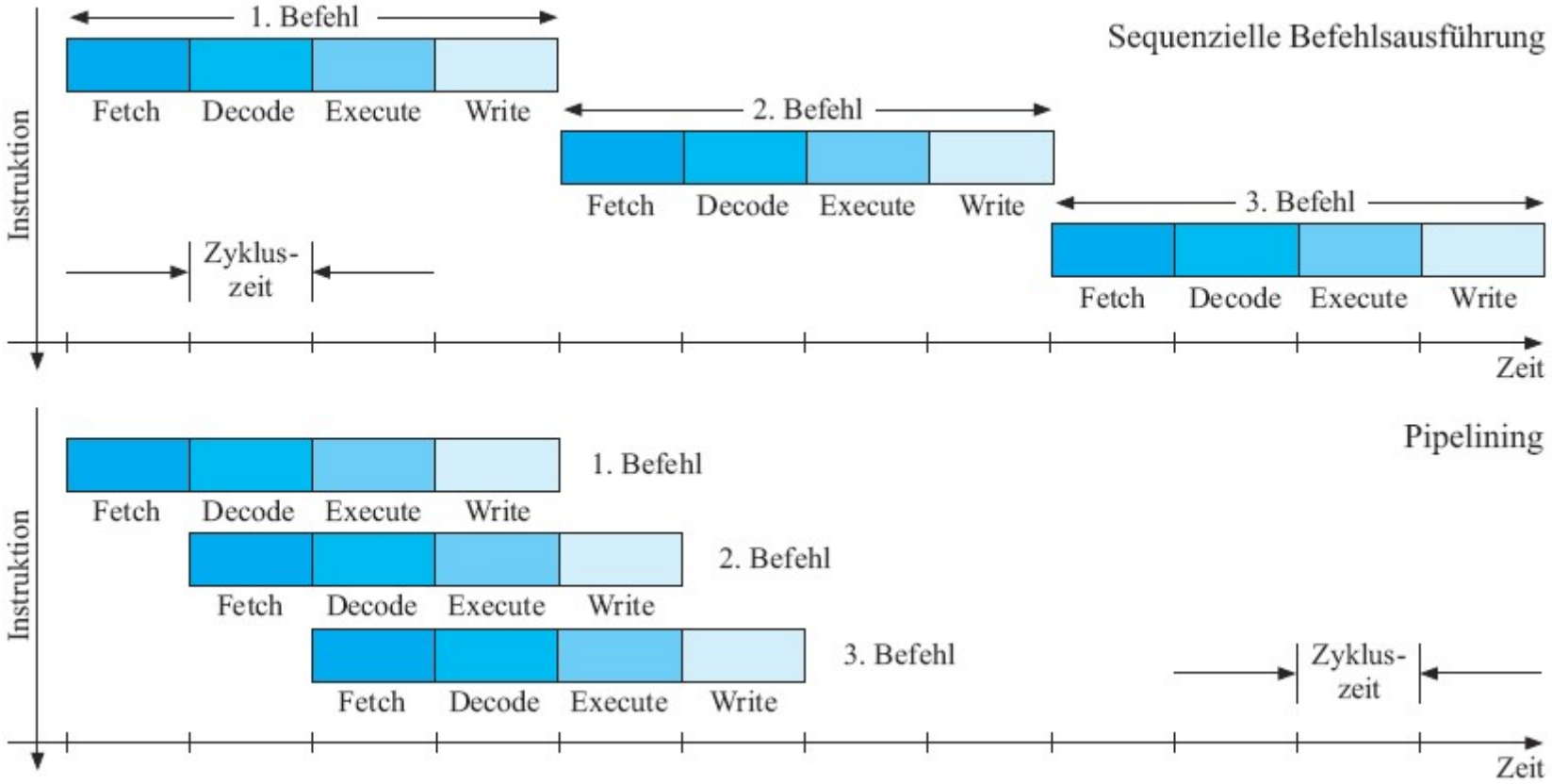
Pipelining

- **Einführung in der industriellen Produktion durch Henry Ford Anfang des 20. Jahrhunderts (Fließbandproduktion)**
  - Produktion eines Autos wird in kleine Schritte unterteilt und an aufeinander folgenden Arbeitsstationen ausgeführt
    - jede Station ist hoch effizient, da immer die gleichen Tätigkeiten ausgeführt werden
    - Produktion wird parallelisiert, da z.B. ein Auto lackiert wird, während bei einem anderen gerade gleichzeitig der Motor eingebaut wird
- **bei Prozessoren ist das Pipelining die wesentliche technische Innovation, um die Performance zu steigern**
  - dazu werden z.B. die vier Bearbeitungsphasen eines Maschinenbefehls in verschiedenen Stationen ausgeführt



Pipelining (2)

- **Befehlsdurchsatz wird durch Pipelining drastisch erhöht**
  - Beschleunigung (*speedup*) bis Faktor 4 (bei 4 Pipelinestufen)



Pipelining (3)

- **Latenz**
  - Anzahl der Takte, die ein Befehl zur Fertigstellung braucht
  - im Beispiel: 4 Takte
  - man muss also 4 Takte warten, bis das Ergebnis feststeht
- **Durchsatz**
  - Anzahl der Befehle, die pro Zeiteinheit bearbeitet werden
  - im Beispiel: 1 Befehl pro Takt
  - man bekommt also in jedem Takt (außer ganz am Anfang beim Füllen der Pipeline) ein neues Ergebnis
- **Eigenschaften nach Einführung des Pipelinings**
  - Latenz erhöht sich unter Umständen
  - Durchsatz wird im Idealfall um einen Faktor erhöht, der der Anzahl der Pipelinestufen entspricht
    - Ausnahme: Pipeline Hazards (s.u.)

Pipeline Hazards (Pipeline Konflikte)

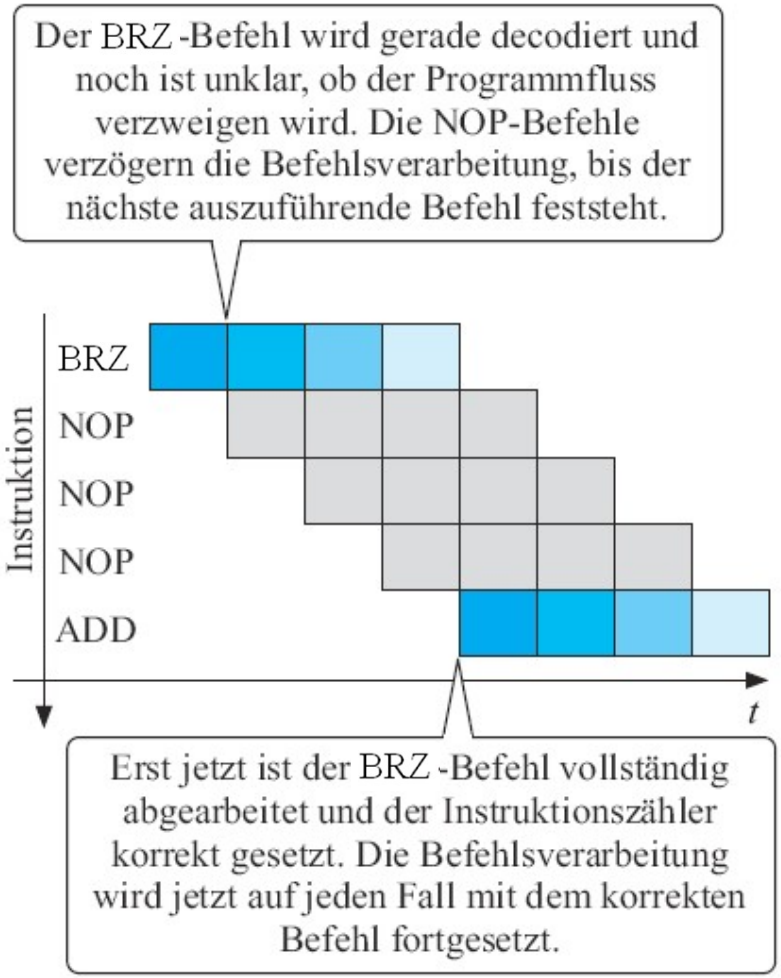
- **Beispiel für einen Kontrollflusskonflikt**

```
start: BRZ else      // if (A != 0)
       ADD #1        //     A = A + 1
       JMP next      // else
else:  ADD #5        //     A = A + 5
next:  …
```

  - erst wenn der bedingte Sprung in der Write-Stufe die neue Adresse „`next`“ in den PC (*program counter*) geschrieben hat, kann der nächste Befehl von der korrekten Adresse gelesen werden
  - in der Zwischenzeit könnten aber drei weitere Befehle gelesen und in die Pipeline gesteckt werden
    - es ist nicht klar, von wo aus diese Befehle gelesen werden müssen
      - es steht nicht fest, ob verzweigt wird oder nicht
      - die Zieladresse muss erst berechnet werden

Pipeline Hazards (Pipeline Konflikte) (2)

- **Abhilfe**
  - Einfügen von Wartezyklen (NOP's)
    - durch HW nach jedem Sprungbefehl
    - durch SW, also Compiler
      - der dann genau wissen müsste, um welche Hardware-Architektur es sich handelt (Anzahl der Pipeline-Stufen)
  - Spekulative Berechnung
    - die wahrscheinlicheren Folgebefehle werden in die Pipeline eingespeist
      - normale sequentielle Reihenfolge, wenn das Nicht-Springen wahrscheinlicher ist
      - Befehle ab der Sprungziel-Adresse, wenn das Springen wahrscheinlicher ist



Pipeline Hazards (Pipeline Konflikte) (3)

- stellt sich heraus, dass das die falschen Befehle waren, werden sie vor der Write-Phase aus der Pipeline entfernt (*flushing*)
- moderne Prozessoren überwachen den Kontrollfluss, um gute Sprungvorhersagen machen zu können
  - z.B. merkt sich der Prozessor, ob beim letzten Mal an dieser Stelle gesprungen wurde oder nicht, und benutzt dies für eine Vorhersage
- ist die Sprungvorhersage korrekt, werden im Idealfall keine Takte verschwendet
- nur im Falle einer falschen Vorhersage, gehen durch das Entleeren der Pipeline (*flushing*) Taktzyklen verloren

- **Weitere Aspekte (siehe Vorlesung Rechnerorganisation)**
  - Details der Pipelineimplementierung
  - weitere Pipeline Hazards
    - Strukturkonflikte (Lösung: Struktur ändern)
    - Datenabhängigkeitskonflikte (Lösung: Data Forwarding)

Zusammenfassung