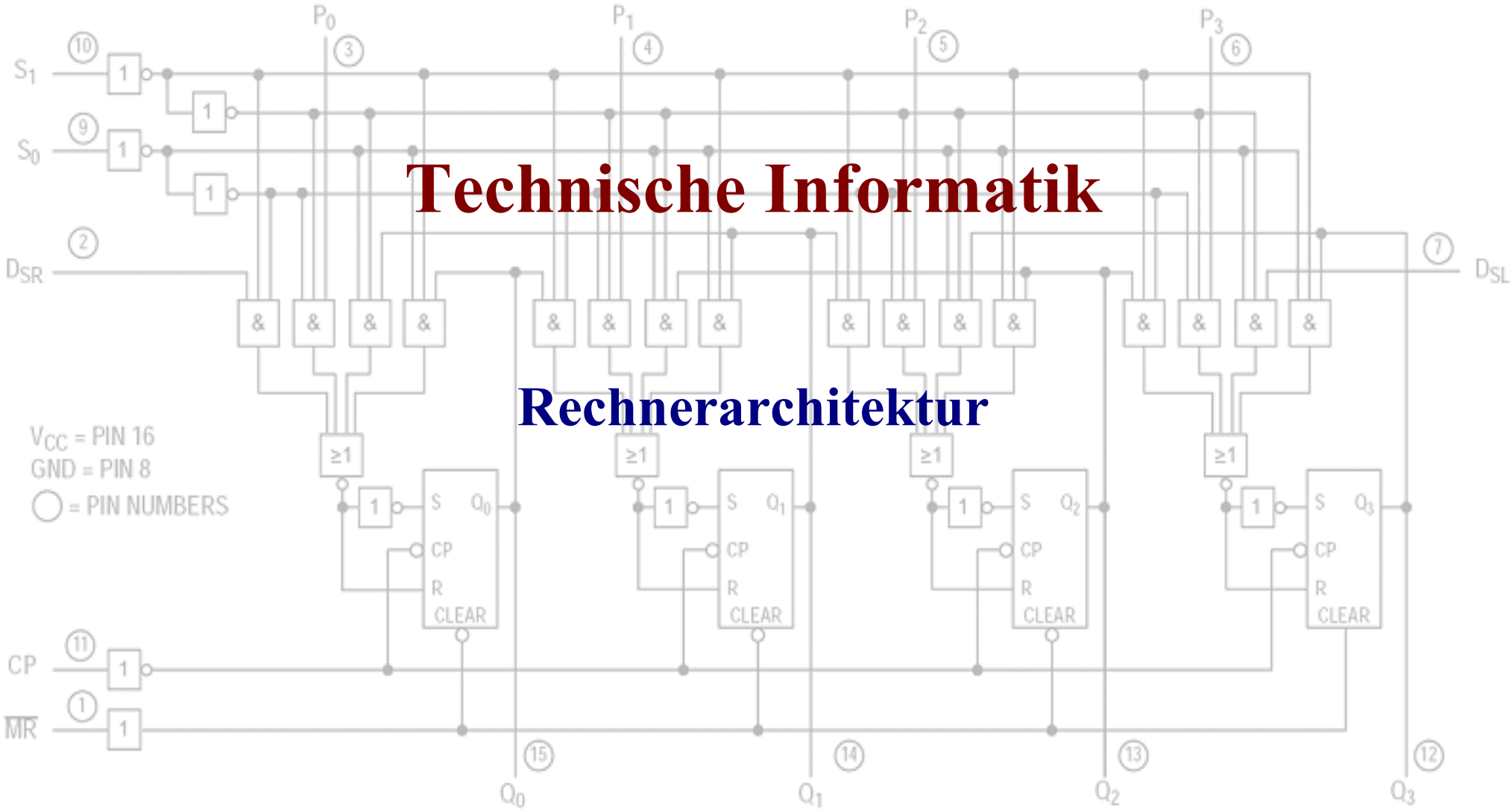
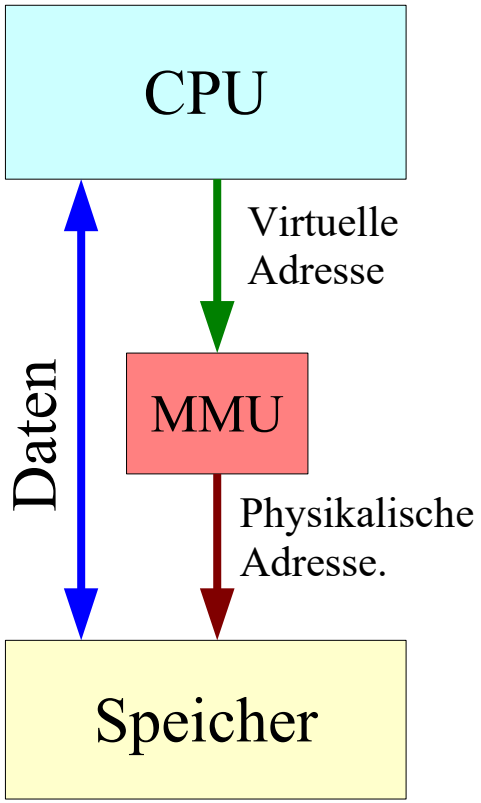



# Technische Informatik

## Rechnerarchitektur

# Memory Management Unit

- Die Memory Management Unit (MMU) ist eine Hardwarekomponente zur Speicherverwaltung
- Aufgabe: Bereitstellung eines virtuellen Adressraumes für jeden Prozess. Jede Applikation „sieht“ ihren eigenen logischen Speicherbereich unabhängig
  - von den anderen Applikationen
  - vom verfügbaren physikalischen Adressraum
- Hardware
  - Übersetzt virtuelle Adressen zu physikalische Adressen
  - Jeder Adressbereich beginnt bei 0
  - Jeder Applikation wird ein maximaler Adressumfang zugeteilt

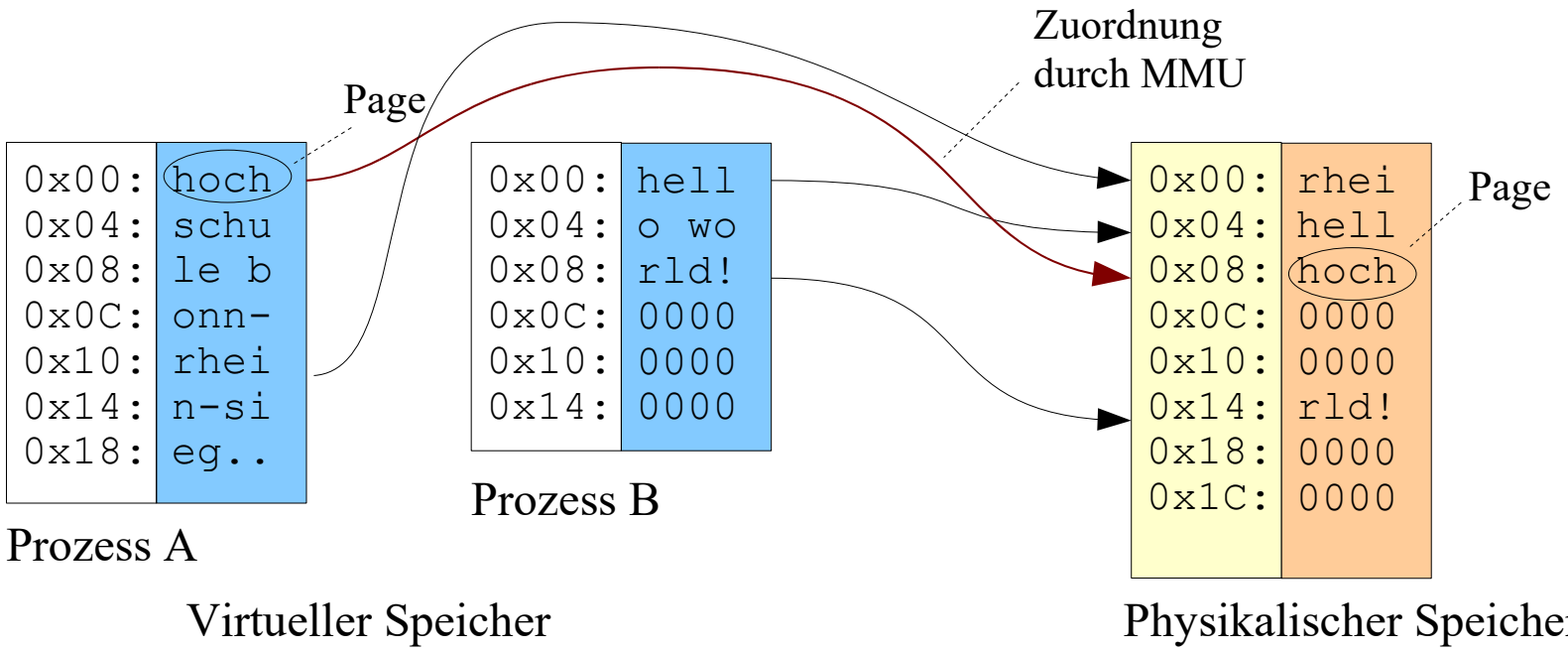


# Memory Management Unit

- Jeder Prozess hat eigenen virtuellen Adressraum
  - Beispiel: 32-Bit-Adresse => $2^{32}$ Speicherstellen
- Rechnersystem hat einen linearen physikalischen Adressraum
  - Beispiel: 1 GB => $2^{30}$ Speicherstellen
- Paging:
  - Virtueller und physikalischer Adressraum werden in gleich große Bereiche (Seiten) eingeteilt
  - Beispiel: 4 KB => $2^{12}$ Speicherstellen
- Page Table (Seitentabelle):
  - Gibt an, an welcher Stelle sich eine virtuelle Seite im physikalischen Speicher befindet
  - Beispiel: Ein Prozess benötigt $2^{20}$ Tabelleneinträge

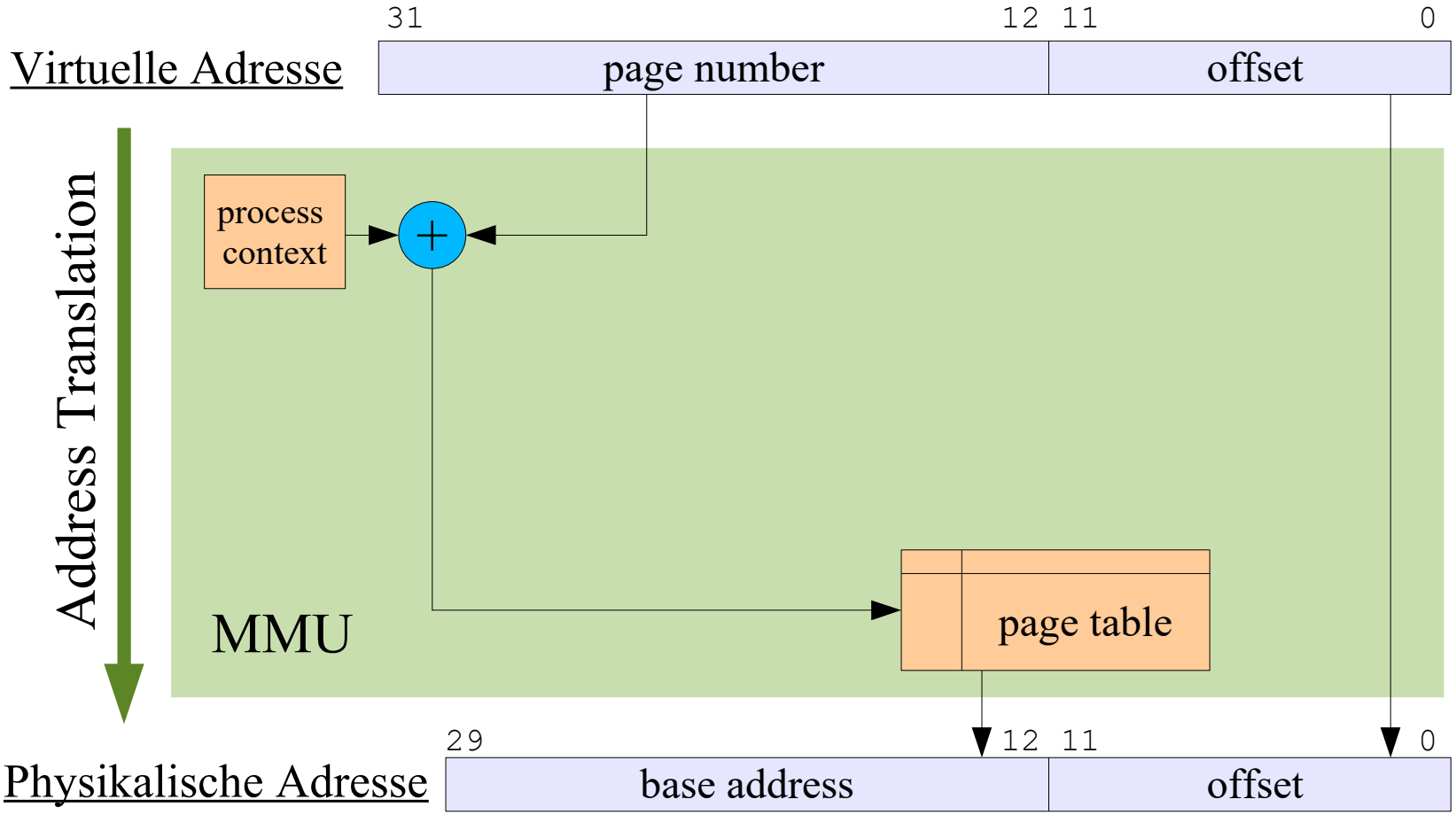
# Memory Management Unit

- Die MMU bildet eine virtuelle Adresse auf eine physikalische Adresse ab
  - Basis: Page Table und aktueller Prozess-Kontext
  - Jeder Prozess hat einen eigenen Block innerhalb der Page Table
  - Beliebige Reihenfolge der Seiten im physikalischen Speicher
  - Teile des Hauptspeichers können auf anderen Datenträgern ausgelagert werden.
    Falls angeforderte Seite nicht im Hauptspeicher liegt, muss diese nachgeladen werden



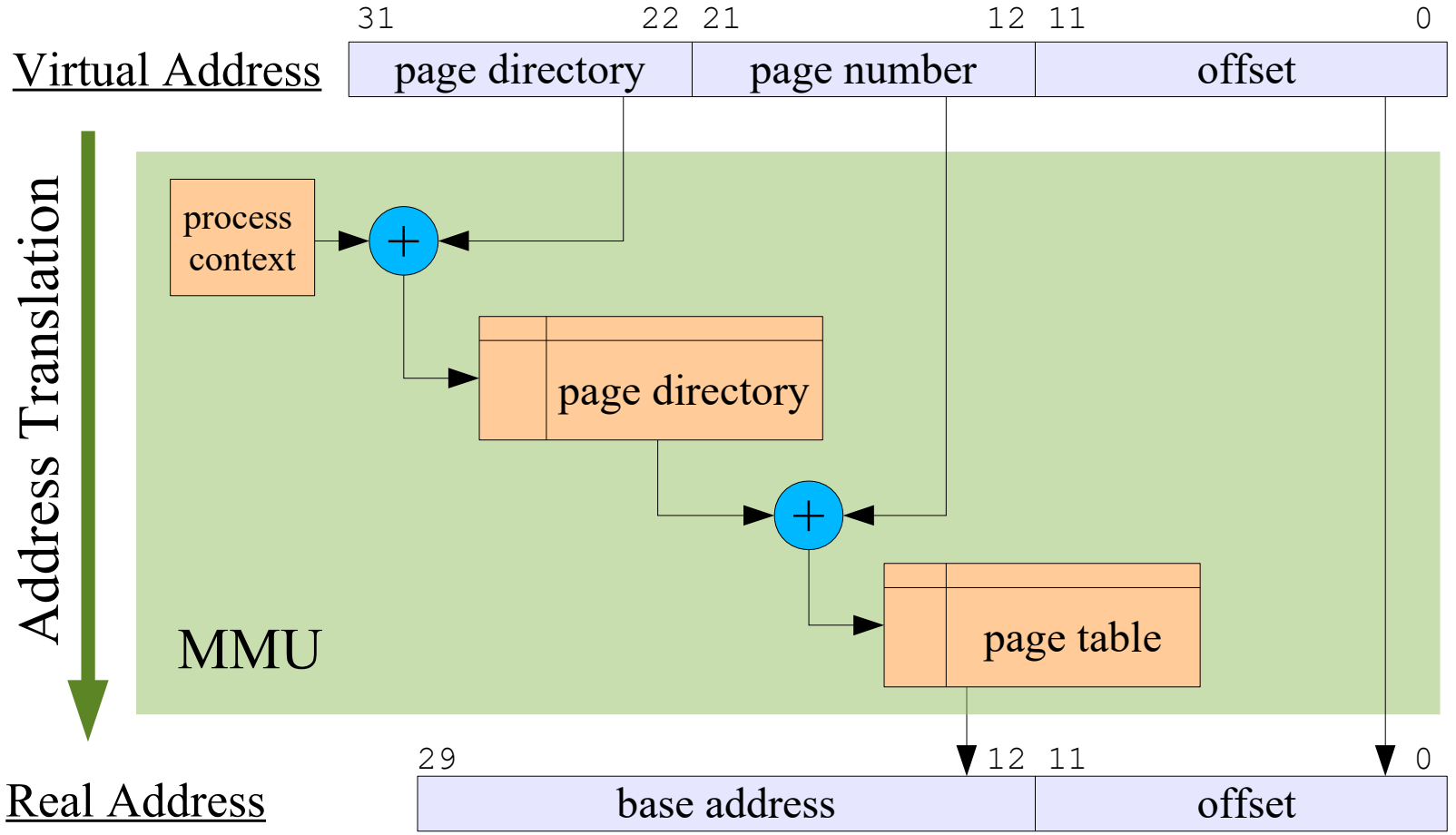
# Memory Management Unit

- Prinzipschaltbild



# Memory Management Unit

- Prinzipschaltbild mit mehrstufiger Zuordnung
  - Page Directory (Seitentabellenverzeichnis) verbleibt im Speicher
  - Page Table (Seitentabelle) wird ggf. nachgeladen

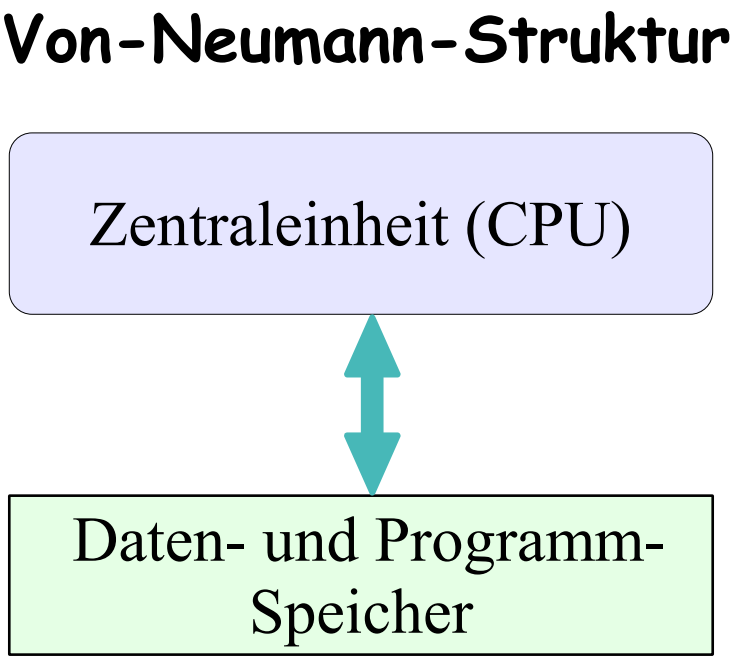


# Memory Management Unit

- Optimierungen:
  - Wahl der Seitengröße
    - Kleine Seiten: Seitentabelle wird zu groß
    - Große Seiten: Speicherfragmentierung
  - Seitentabellen können auch durch (kleinere) Seitenregister (= invertierte Seitentabelle) ersetzt werden, dies erfordert jedoch einen Suchalgorithmus
  - Mehrstufige Verfahren erfordern sequenzielle Berechnungen, diese können über Seitencaches (Translation Lookaside Buffer, TLB) beschleunigt werden

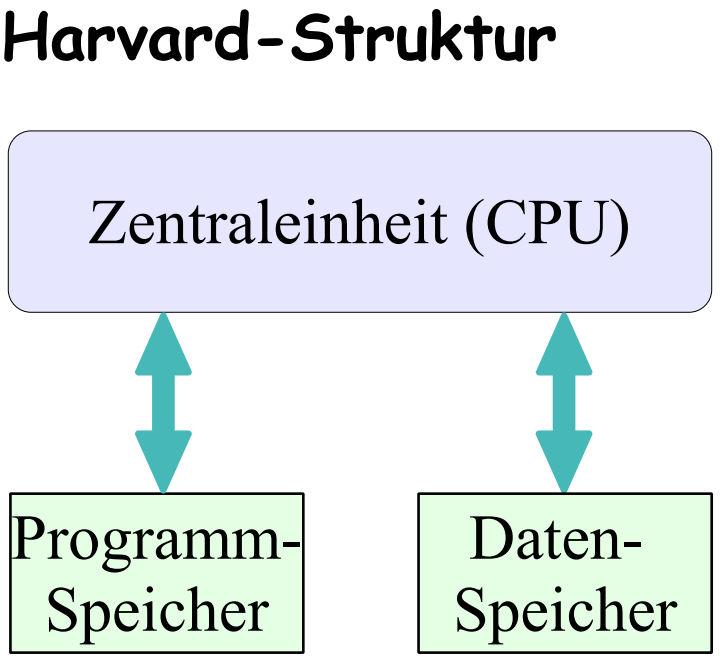
# Von-Neumann-Struktur

- Daten und Programm-Code liegen in einem **gemeinsamen** Speicher
  - Programm-Code
  - Daten
  - I/O-Bereiche
- Vorteil:
  - Vereinheitliche Betrachtungsweise
  - Übersichtlich, strukturiert, ...
  - Speicheraufteilung kann je nach Anwendung optimiert werden
- Nachteil:
  - Speicherzugriff wird oft zum Flaschenhals
- Abhilfe:
  - Speicherhierarchie



# Harvard-Struktur

- Daten und Programm-Code liegen in **getrennten** Speichern
  - 1. Bereich: Programm-Code, 2. Bereich: Daten
  - I/O-Bereiche wahlweise im 1. oder 2. Bereich
- Vorteil:
  - Größere Bandbreite beim Zugriff, da gleichzeitiger Zugriff auf beide Bereiche möglich
  - Programmspeicher vor Zugriffsfehlern geschützt
- Nachteile:
  - Feste Aufteilung der Speicherbereiche
  - Suboptimale Nutzung der Speicherressourcen
  - Unübersichtliche Programmierung in manchen Anwendungen
- Abhilfe:
  - "Super-Harvard" z.B. *Analog-Devices SHARC*



# CISC-Prozessoren

- **CISC** = **C**omplex **I**nstruction **S**et **C**omputer
- Beispiel: *80x86*, *680x0*, C51, ...
- Eigenschaften
  - Umfangreicher Befehlssatz
    - z.B.: Gleitkomma, verschiedene Adressierungsarten
  - Gut geeignet für Assembler-Programmierung
  - Kurzer Maschinencode
  - Befehle kombinieren Verknüpfung und Transfer
  - Aber:
    - Meist werden nur 5% - 10% der Befehle genutzt
    - µ-Programmierung sehr aufwendig
    - Lange µ-Programme -> Reaktion auf Unterbrechung langsamer

```
ADD eax,[0x1234]
```

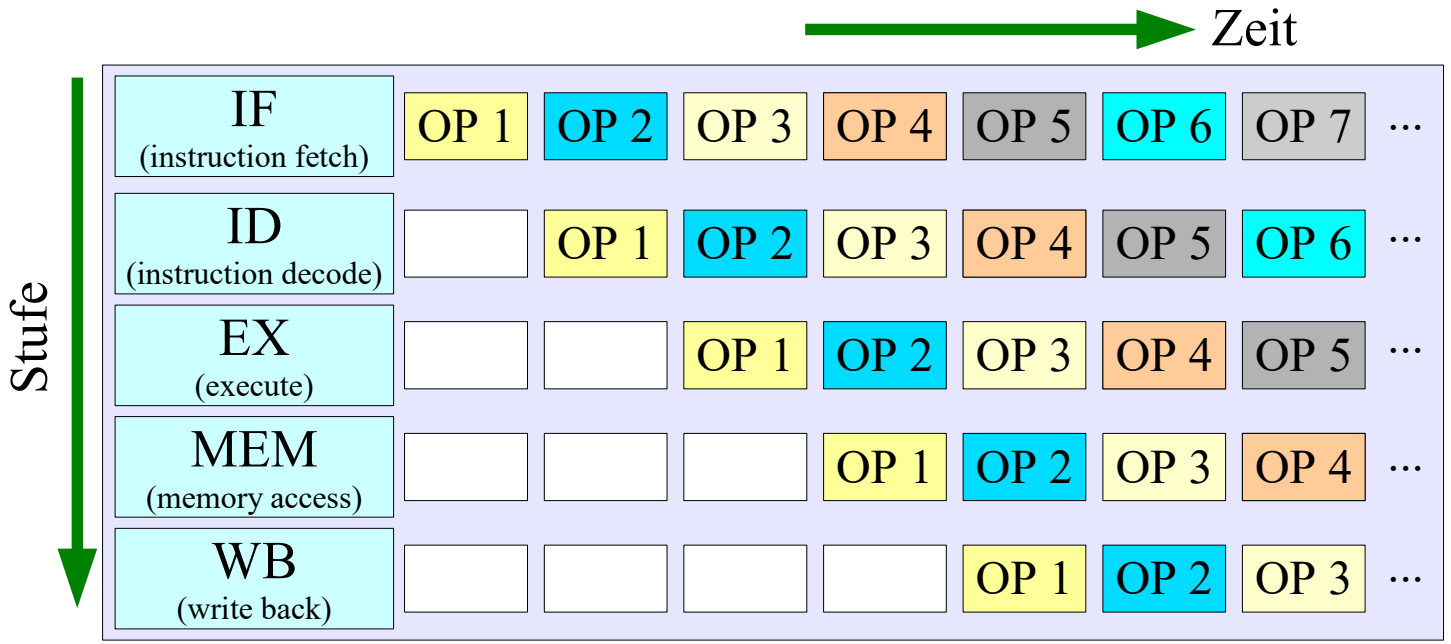
# RISC-Prozessoren

- **RISC** = **R**educed **I**nstruction **S**et **C**omputer
- Beispiel: *Power-PC*, *Alpha*, ARM, AVR, ...
- Eigenschaften
  - Nur elementare (orthogonale) Befehle
  - Befehle kombinieren nicht Verknüpfung und Transfer
  - Load/Store-Architektur (Register/Register-Architektur)
  - Wenige Adressierungsarten
  - Pipeline-Struktur
  - Steuerung „fest verdrahtet“
  - Aber:
    - Langer Maschinencode
    - Nicht für Assembler-Programmierung geeignet

```
MOV ebx,[0x1234]
ADD eax,ebx
```

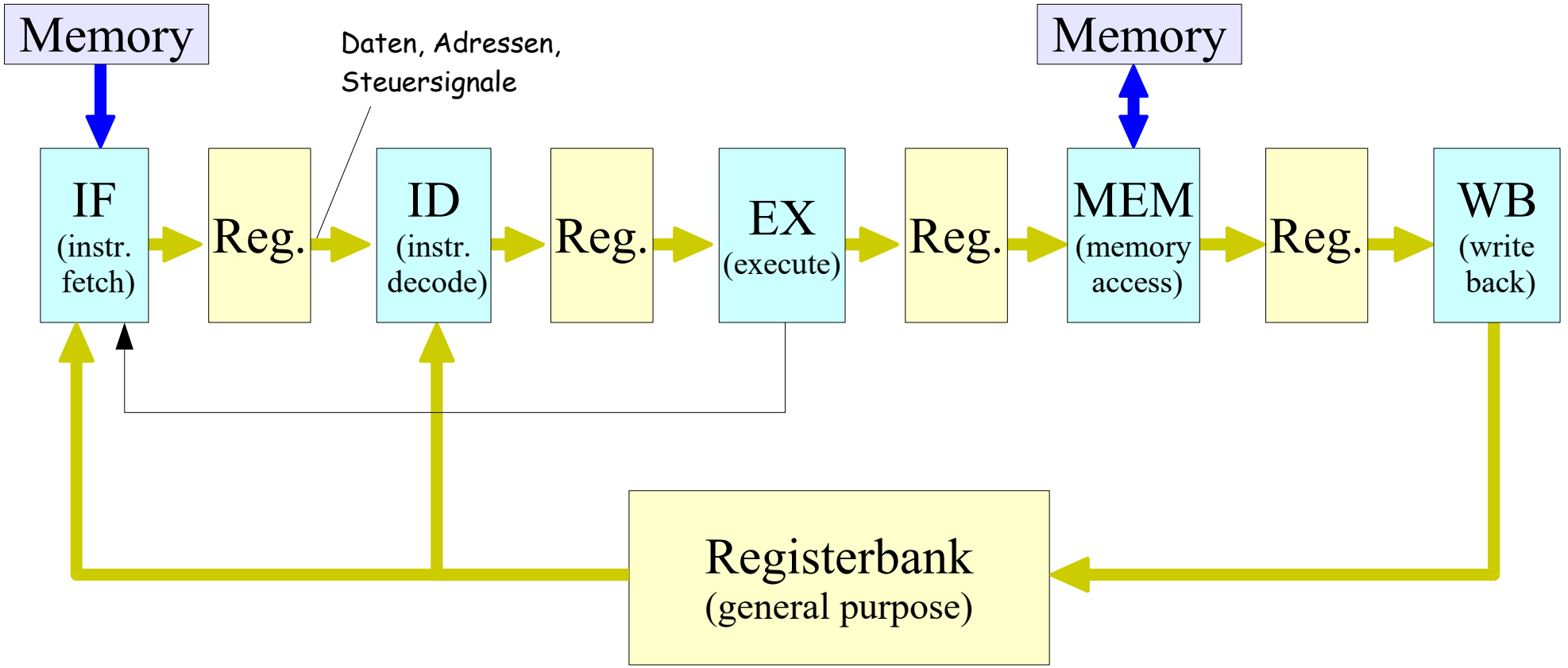
# Pipeline in RISC-Prozessoren

- Alle Befehle werden in gleichen Schritten bearbeitet
  - z.B.: *instruction fetch - instruction decode – execute - memory access - write back*
- Prinzip:
  - Während der zweite Schritt eines Befehls bearbeitet wird, kann schon der erste Schritt des nächsten Befehls bearbeitet werden usw.
  - Mit jedem Taktzyklus (Idealfall) wird ein Ergebnis produziert
  - Aber: Konflikte müssen behandelt werden



# Pipeline in RISC-Prozessoren

- Pipeline-Struktur



# Pipeline-Konflikte

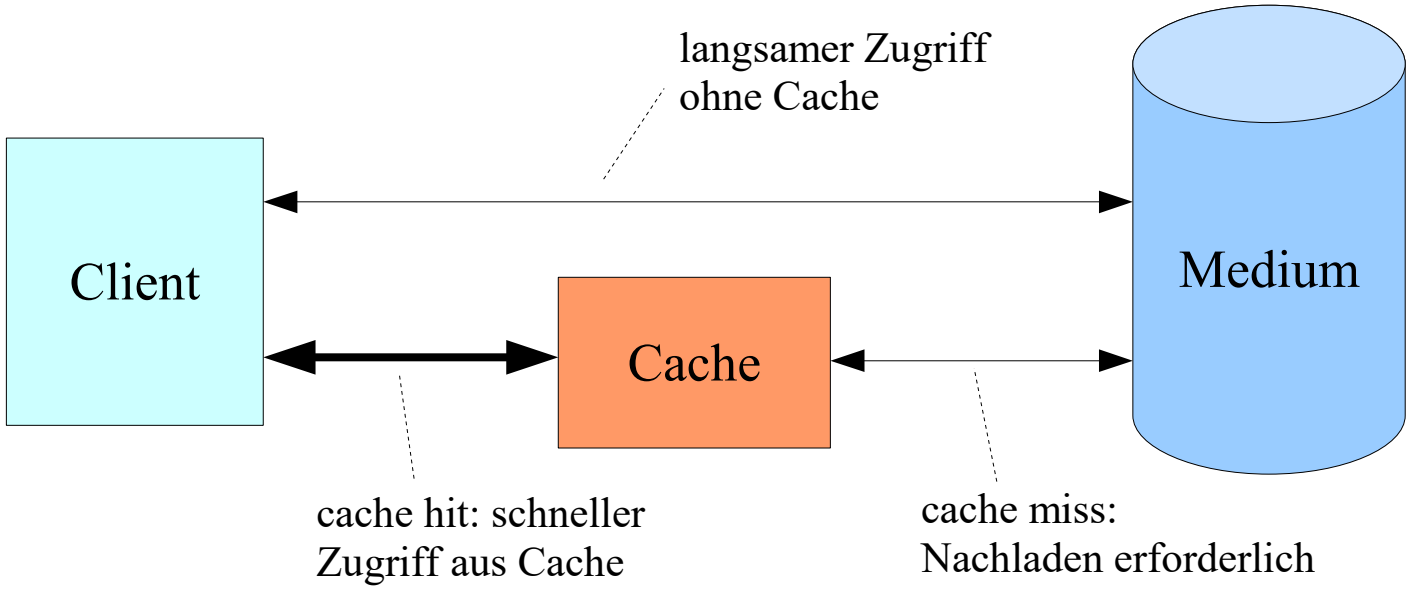
- Strukturelle Konflikte:
  - Gleichzeitiger Zugriff auf Ressourcen
  - z.B.: Register oder  Speicherzugriff bei gemeinsamen Daten- und Programmspeicher
- Datenfluss-Konflikte:
  - Aufeinanderfolgende Befehle benötigen gleichen Speicherinhalt
  - z.B.: RAW = Read-After-Write, Inhalt darf erst nach Scheiben gelesen werden
- Laufzeit-Konflikte:
  - Zugriffszeit auf Ressourcen (z.B. Hauptspeicher) länger als 1 Taktzyklus
- Steuerfluss-Konflikte:
  - Verzweigung im Programmablauf (erst erkennbar, wenn „falscher“ Befehl schon in der Pipeline steht)

# Lösung von Pipeline-Konflikten

- Hardware:
  - Hardware muss Konflikt erkennen
    - Unterdrückung von Taktsignalen der Pipeline-Register für einige Zyklen
      - Pipelineverarbeitung wird eingestellt = *stalled*
    - Zwischenspeichern (Bypass) und Zurückführen von Registerinhalten
- Software:
  - Compiler erkennt Konflikte und beseitigt sie durch
    - Einfügen von NOP („no operation“-Befehle, verlängert Maschinencode)
    - Umsortieren von unabhängigen Befehlen
  - Erfordert enges Zusammenspiel von Rechnerarchitektur und Compilerbau!

# Cache

- Cache
  - Schneller Pufferspeicher, mit dem Zugriffe auf langsame Medien beschleunigt werden
  - Im Cache werden häufig verwendete Datensätze zwischengespeichert
    - Anfragen werden direkt aus dem Cache bedient,
      wenn der nachgefragte Datensatz vorhanden ist (cache hit)
    - Erst wenn ein Datensatz nicht im Cache vorhanden ist (cache miss), muss dieser vom langsamen Medium nachgeladen werden
    - Begrenzte Speicherkapazität erfordert optimale Ersetzungsstrategie



# Parallelität

- Performance-Verbesserung durch parallele Strukturen
  - Vektorrechner (SIMD = Single Instruction Multiple Data)
    - Ein Befehl bearbeitet mehrere Daten (Vektor), z.B. Multimedia, Simulation, ...
    - Vorauss.: Daten liegen in Registern vor, mehrere Verarbeitungseinheiten
  - Superskalarer Prozessor
    - Aufeinanderfolgende Befehle eines Threads werden gleichzeitig bearbeitet
    - Vorauss.: Keine Abhängigkeiten der Ergebnisse, parallele Pipelinestrukturen bzw. Ausführungseinheiten
  - Multithreading
    - Gleichzeitige Bearbeitung mehrerer Threads, jedoch auch gemeinsame Ressourcen (FPU,...)
    - Vorauss.: weitgehend parallele Strukturen
  - Parallelrechner, Multicore (MIMD = Multiple Instructions Multiple Data)
    - Gleichzeitige Bearbeitung mehrerer Threads
    - Vorauss.: Mehrfache, unabhängige Prozessoren, gemeinsames Bussystem

Thread:
Aufeinanderfolgende Befehle,
'Ausführungsfaden'