Architekturen für moderne Workstations

Der Einsatz von Prozessoren in modernen Hochleistungs-Workstations stellt eine Reihe besonderer Anforderungen an die Prozessorentwicklung.

Mikroprozessoren
aktuelle Einsatztrends

Workstation

- grafische Oberflächen
- parallel ablaufenden Applikationen
- hohe geforderte Systemleistung
- großer Speicherraum
- hohe Systemstabilität

Embedded Control

- Einsatz für Steuerungszwecke
- breites Einsatzspektrum
- kompakt
- stromarm
- kostenoptimal
- hohe Zuverlässigkeit
- kurze Reaktionszeiten

Digitaler Signalprozessor

- numerische Verarbeitung analoger Signale
- einfache Algorithmen
- moderater Speicherbedarf
- sehr kurze Verarbeitungszeiten
- hohe Rechengenauigkeit (16 . . . 48 Bit)

Schwerpunkt: *Prozessoren für Workstations*

Betriebssysteme

carakteristische Kennzeichen und resultierende Aufgaben

HOCHSCHULE MITTWEIDA UNIVERSITY OF APPLIED SCIENCES

Multitasking

- *quasi* gleichzeitige Bearbeitung mehrerer Task auf einem Prozessor
- mehrere Task mit *Code*, *Daten* und *Stack* gleichzeitig im Speicher
- Taskwechsel wird durch Betriebssystem vermittelt
  (Scheduler- und Dispatchercode, Verwaltungsstrukturen, dots)

erweiterte Aufgaben des Betriebssystems

- Zuweisung von Speicher zu Task
- Durchführung Taskwechsel
- Koordinierung von Zugriffswünschen auf Systemressourcen
- Vermittlung von Interprozeßkommunikation
- zentrale Verwaltung der grafischen Oberfläche
- Gewährleistung der Stabilität des Gesamtsystems

Entwicklung moderner Prozessoren

Anforderungen aus dem Anwendungsumfeld

HOCHSCHULE MITTWEIDA UNIVERSITY OF APPLIED SCIENCES

wesentliche Erhöhung der Verarbeitungsleistung

- Verwaltungsoverhead Betriebssystem
- Multitasking

Isolation der Anwenderprogramme

- a) gegeneinander und b) gegenüber dem Betriebssystem
- Fehlerhafte Programme dürfen nur sich selbst beeinflussen

Speicherverwaltung

- Zuweisung freier Speicherbereiche für Code, Daten und Stack
- Verhinderung von Fragmentation
- Unterstützung von *virtuellem* Speicher

dynamische Adreßbindung (*Linking*)

- Code muß zur Laufzeit an aktuell belegten Adreßbereich gebunden werden

Erhöhung der Verarbeitungsleistung

In den letzten Jahren wurden Architektur und Arbeitsweise moderner Rechnersysteme mehrfach überarbeitet um eine wesentliche Steigerung der Verarbeitungsleistung zu erzielen.

Verarbeitungsleistung

hat zwei Komponenten:

Datendurchsatz * Verarbeitungsgeschwindigkeit

Möglichkeiten zur Steigerung:

- schnellere Befehlsbearbeitung
  Erhöhung Taktfrequenz, Änderung der Architektur
- Vergrößerung der Verarbeitungsbreite
  $8 \rightarrow 16 \rightarrow 32 \rightarrow 64$ Bit
- Spezialisierung
  Abkehr vom Universalprozessor, applikationsspezifische Struktur und Befehlssatz
- Parallelverarbeitung
  Problembearbeitung durch mehrere Prozessoren

Leistungsbewertung
benötigte Programmlaufzeit

HOCHSCHULE MITTWEIDA UNIVERSITY OF APPLIED SCIENCES

$\Rightarrow$ einzig reale Vergleichsbasis unterschiedlicher Systeme

Zeit zur Ausführung eines Befehls mit *CPI* Takten pro Befehl bei einer Frequenz von f_{Clock}

$$t_{Befehl} = CPI * \tau_{Clock} = \frac{CPI}{f_{Clock}}$$

Zeit zur Ausführung eines Programms mit n Befehlen

$$T_{gesamt} = \sum_{i=1}^{n} t_{Befehl_i} = \frac{1}{f_{Clock}} \sum_{i=1}^{n} CPI_i$$

Leistungsbewertung
benötigte Programmlaufzeit

HOCHSCHULE MITTWEIDA UNIVERSITY OF APPLIED SCIENCES

Ist H die absolute Häufigkeit von Befehlen mit gleichem *CPI*, so erhält man:

$$T_{ges} = \frac{1}{f_{Clock}} \sum_{j=1}^{m} CPI_j * H_j \quad \text{mit} \quad n = \sum_{j=1}^{m} H_j$$

Kennt man nur die relativen Häufigkeiten $h = \frac{H}{n}$, ergibt sich

$$T_{ges} = \frac{n}{f_{Clock}} \sum_{j=1}^{m} CPI_j * h_j$$

Leistungsbewertung

benötigte Programmlaufzeit

HOCHSCHULE MITTWEIDA UNIVERSITY OF APPLIED SCIENCES

mit $\overline{CPI}$ als mittlere Anzahl Takte pro Befehl

$$\overline{CPI} = \sum_{j=1}^{m} CPI_j * h_j$$

ergibt sich schließlich

$$T_{ges} = \frac{n}{f_{Clock}} \overline{CPI}$$

- Beachte:
  - $\overline{CPI}$ ist wichtige Masszahl für den Vergleich von CPUs
  - hängt vom Befehlsmix ab
- Speicherzugriffe, Adressumsetzung, Busvergabe erhöhen effektives $\overline{CPI}$

Leistung eines Rechnersystems

HOCHSCHULE MITTWEIDA UNIVERSITY OF APPLIED SCIENCES

. . . hängt nur z.T. von der Prozessorleistung ab.

Weitere Einflussfaktoren sind

- I/O-Zugriffe
  - sehr langsam (μs . . . s), aber selten $\Rightarrow$ vernachlässigbar
- Speicherzugriffe
  - Hauptproblem dank häufiger Nutzung

Prozessoren sind gegenwärtig *deutlich* schneller als Speicher

- Typische Werte für Taktperiode bzw. Zugriffszeit:

	1975	2012
CPU-Takt	500 ns	0,25. . . 0,5 ns
Speicherzugriff	150 ns	15..20 ns

Wirkung langsamer Komponenten

Waitstates bremsen die Verarbeitungsleistung

- Systemleistung hängt von *allen* Komponenten ab
- langsame Komponenten (I/O, Speicher) bremsen System
  - $\Rightarrow$ Einfügung unproduktiver Leerlauftakte (Wait States) bis Daten bereit stehen / übernommen werden
  - $\Rightarrow$ Erhöhung des effektiven $\overline{CPI}$
- besonders kritisch $\Rightarrow$ langsamer Speicher (10 . . . 1000 WS)

Leistungseinbuße infolge n eingefügter Waitstates:

$$1 - \frac{\overline{CPI}}{\overline{CPI} + n\,WS}$$

Beispiel: Wirkung von 10 Wait States / Befehl für verschiedene $\overline{CPI}$

$\overline{CPI} = 10$, $WS = 10 \Rightarrow$ Einbusse 50 %
$\overline{CPI} = 1$, $WS = 10 \Rightarrow$ Einbusse ca. 91 %

Verarbeitungsleistung

Unsere Schwerpunkte

hat zwei Komponenten:

Datendurchsatz * Verarbeitungsgeschwindigkeit

Möglichkeiten zur Steigerung:

- schnellere Befehlsbearbeitung
  Erhöhung Taktfrequenz, Änderung der Architektur
- Vergrößerung der Verarbeitungsbreite
  $8 \rightarrow 16 \rightarrow 32 \rightarrow 64$ Bit
- Spezialisierung
  Abkehr vom Universalprozessor, applikationsspezifische Struktur und Befehlssatz
- Parallelverarbeitung
  Problembearbeitung durch mehrere Prozessoren

Erhöhung der Taktfrequenz

Bei konstantem $\overline{CPI}$ ist die Befehlsdauer umgekehrt proportional zur Taktfrequenz.

Erhöhung der Taktfrequenz

erfordert Änderung in Schaltungsdesign und Herstellung

- maximal erreichbarer Prozessortakt wird begrenzt durch:
  - Schaltungsdesign
    (Länge des *kritischen Pfades*)
  - technologische und phys. Randbedingungen
    (Elektronenbeweglichkeit, Laufzeiten, Leitfähigkeit der Materialien, Kapazitäten, Wärmeabfuhr)

Einige historische Werte

Jahr	f_{Clk}	typischer Vertreter
1975	2 MHz	Z80, 6502, 8080
1980	8 MHz	8086, 68000
1985	12 MHz	80286, 68020
1990	33 MHz	80386, 68030
1998	450 MHz	DEC Alpha, PowerPC, Pentium II
2003	ca. 3 GHz	Athlon, Pentium 4, ...

Architekturänderung

Eine Erhöhung der Verarbeitungsleistung bei gleicher Taktfrequenz läßt sich durch eine Verringerung des mittleren CPI-Wertes erreichen.

- Pipelining – Befehlsbearbeitung im Fließbandprinzip
- Superskalare Architektur – mehrere Befehle pro Takt begonnen

Hauptwege zur Senkung $\overline{CPI}$

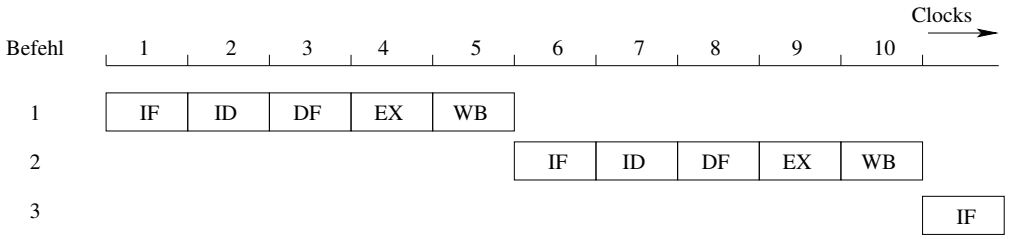
Bessere Auslastung der CPU

- höhere interne Parallelität auf Befehlsebene
  ⇒ *Instruction Level Parallelism (ILP)*
  - Pipelining — CPI > 1
    Fließbandprinzip als Grundlage der Parallelisierung
  - Reduced Instruction Set Computer — CPI = 1.0
    Konsequente Umsetzung des Fließbandprinzips.
  - Superskalare Prozessoren — CPI < 1
    Beginn mehrerer Befehle in jedem Takt
  - Very Long Instruction Word
    Vom Compiler gesteuerte parallele Ausführung von Befehlsgruppen
- höhere interne Parallelität auf Threadebene
  ⇒ Multi-Threading (oder Thread Level Parallelism)

von-Neumann Ausgangspunkt

Befehlsbearbeitung klassisch

- CPU bearbeitet einen Befehl nach dem anderen.
- Befehlsablauf eines Prozessors ist eine *sequentielle* Folge von Teiloperationen:
  1. Instruction Fetch IF
  2. Instruction Decode ID
  3. Data Fetch DF
  4. Execute EX
  5. Write Back WB

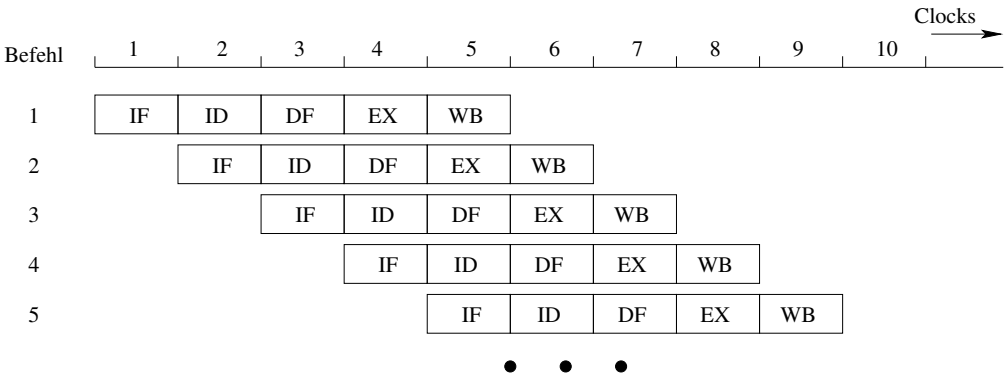


- 1. . . n Takte / Stufe
  - jeweils von separater Hardware in CPU bearbeitet

Pipelining

zeitlich parallele Bearbeitung nach dem Fließbandprinzip.

Pipelining
Idealschema



- Im Mittel wird 1 Befehl/Takt fertiggestellt (scheinbar $\overline{CPI} = 1$)
- Bearbeitung jedes einzelnen Befehls jedoch nicht schneller!

Probleme der Umsetzung
Pipeline Konflikte

1. ungleich lange Teiloperationen
2. Strukturengpässe (*Structural Hazards*)
3. Datenabhängigkeiten (*Data Hazards*) aufeinanderfolgender Befehle
4. Änderung des Programmflusses (*Control Hazards*)

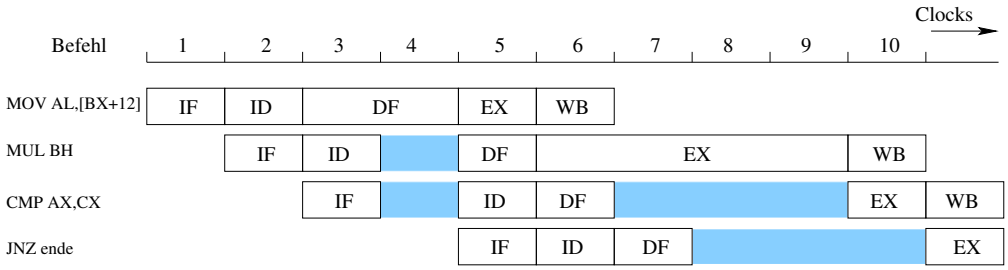
notwendige Massnahmen:

- Pipeline Stall – Einfügung von Leerlauftakten
- Pipeline Flush – Leeren und Neubefüllen der Pipeline
- durch technische und organisatorische Maßnahmen z.T. vermeidbar
  - Änderung Befehlssatz
  - Entflechtung der Abhängigkeiten
  - zusätzliche Hardware

Ungleich lange Teiloperationen

Ausführungszeit der Phasen variiert je nach Befehl

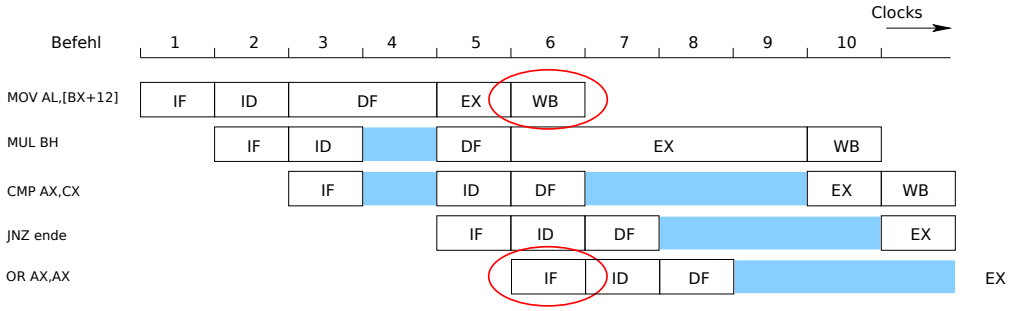
- Lesen langer Befehlscodes
- Berechnung der effektiven Adresse in DF und WB
- Datentransporte vom/zum Speicher
- Lange Ausführungszeit (Multiple Shift, Multiply, Divide, . . .)



Strukturengpässe

zeitgleiche Nutzung einer Hardwareeinheit durch mehrere Befehle

- z.B. Nutzung der ALU zur Adressberechnung in DF und in EX.
- z.B. Speicherzugriffe zum Datentransport (DF oder WB) behindern Befehlslesen (Häufigkeit: ca. 30 %)



Gegenmaßnahmen:

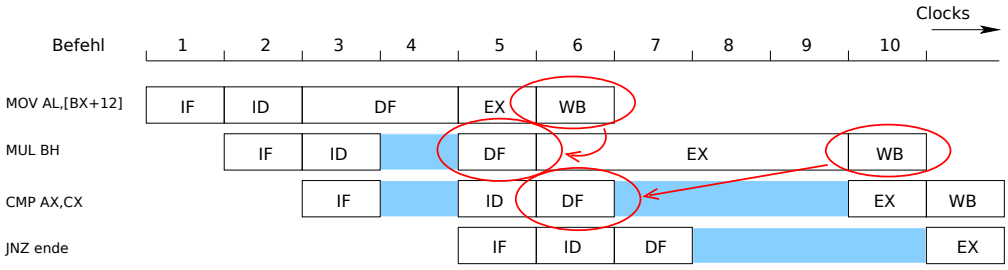
- Einbau zusätzlicher Funktionseinheiten (z.B. *Adress ALU*)
- Große Registerzahl ($\geq$ 32) zur Reduzierung der Speicherzugriffe
- parallele Daten- und Codezugriffe durch (virtuelle) Harvard-Architektur

Datenabhängigkeiten
zwischen aufeinanderfolgenden Befehlen

HOCHSCHULE MITTWEIDA UNIVERSITY OF APPLIED SCIENCES

Read-After-Write-Hazard RAW

Ein Befehl darf Registerinhalte erst auslesen, nachdem ein vorhergehender Befehl das Ergebnis geschrieben hat.

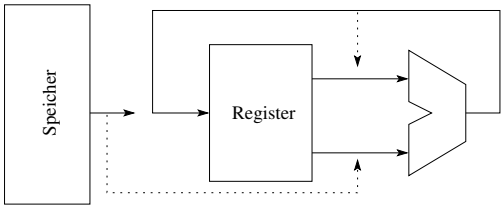


Im Beispiel werden weitere 7 (= 2 + 5) zusätzliche Waitstates nötig.

Datenabhängigkeiten
mögliche Gegenmaßnahmen

HOCHSCHULE MITTWEIDA UNIVERSITY OF APPLIED SCIENCES

- Forwarding
  Zusätzliche Datentransportwege liefern ALU-Ergebnisse oder Speicherinhalte direkt zum ALU-Eingang

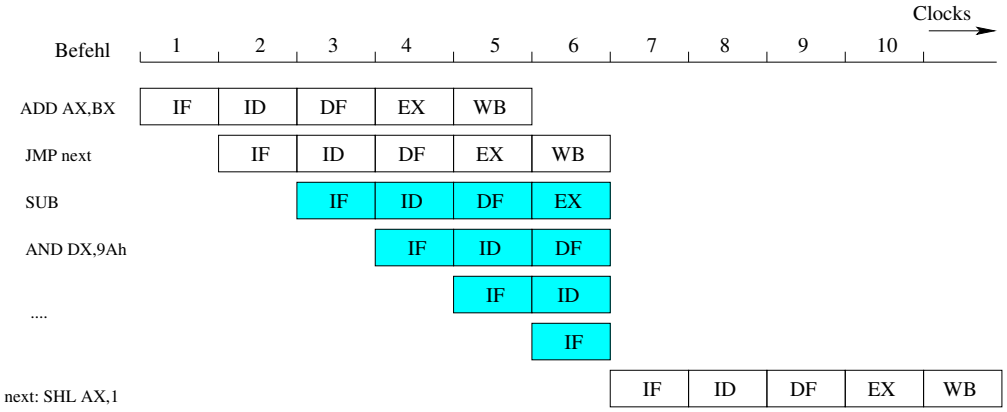


- statisches Befehlsscheduling
  Umordnen der Befehlsreihenfolge (z.B. durch Compiler)

```
ADD AX,BX          ADD AX,BX
CMP AX,99    ⇒     SHL BX,5
SHL BX,5           ADD CX,DX
ADD CX,DX          CMP AX,99
```

Programmflußänderung

machen vorverarbeitete Pipelineinhalte ungültig



- Leeren der Pipeline (*Pipeline Flush*)
- Neubeginn am Sprungziel

Häufigkeit ca. 15 . . . 25 % $\Longrightarrow$ Verarbeitungsleistung sinkt auf ca. 50 %

Programmflußänderung

Gegenmaßnahmen

- Vermeidung von Sprüngen
  - durch Compiler (*Loop Unrolling*, *Function Inlining*, . . .)
  - *bedingte* Befehlsausführung (z.B. `add if Cy set`)
- Früherkennung von JMP, CALL, RET in ID-Phase
  - nicht für bedingte Sprünge geeignet
- spekulative Sprungausführung $\Rightarrow$ *Branch Prediction*
  - CPU verfolgt *wahrscheinlichen* Programmpfad
  - Korrektur (*Pipeline Flush*) nur bei falscher Vorhersage

Branch Prediction

spekulative Sprungausführung

- *statische Vorhersage*
  - z.B. durch Compiler
  - fixe Einstellung
- *dynamische Vorhersage*
  - auf Basis Sprungrichtung
  - anhand beobachtetem Programmverlauf ⇒ Cache-Prinzip

Branch Target Cache

- kombiniert
  - Branch History Table – Vorhersage Sprungentscheidung
  - Branch Target Buffer – Vorhersage Sprungziel
- Adresse Sprungbefehl ist Index in Cache
  - Spekulation schon in IF Phase möglich
- Einbeziehung Sprung-Vorgeschichte verbessert Trefferrate (*gshare*)

Reduced Instruction Set Computers

RISC

Konsequente Umstrukturierung des Prozessors auf zeitoptimale Befehlsausführung.

Reduced Instruction Set Computers

Historie und Hintergründe

HOCHSCHULE MITTWEIDA UNIVERSITY OF APPLIED SCIENCES

. . . beruht auf Arbeiten (ca. 1975 . . . 1980) in *Stanford University* (MIPS-Design) und *Berkeley* (RISC-Design, heute SPARC-Prozessoren).

Basis sind Beobachtungen zur Softwareentwicklung:

- Software wird zunehmend per Compiler erzeugt
- Compiler verwenden vorwiegend einige wenige, einfache Befehle
- Speicher nicht mehr knapp (hohe Codedichte nicht mehr so wichtig)
- komplexe Befehle und Adressierungsarten machen Prozessor kompliziert und langsam

Lösungsprinzip:

- radikale Vereinfachung des Prozessors
- nur wenige, einfache Befehle werden implementiert
- komplexe Befehle werden vom Compiler aus Folge einfacher Befehle nachgebildet

Reduced Instruction Set Computers

typische Architekturmerkmale

HOCHSCHULE MITTWEIDA UNIVERSITY OF APPLIED SCIENCES

- nur wenige Befehle (25 . . . 40)
- wenige Adressierungsarten (≤ 4)
- ALU-Operationen nur zwischen Registern (*Load/Store-Architektur*)
- große Registerzahl (≥ 32) und Harvard-Architektur
  $\Rightarrow$ wenig Konflikte bei Speicherzugriff
- festverdrahtete, zeitoptimale Ablaufsteuerung
- (fast) alle Befehle haben 1 CPI
- einheitliches Befehlsformat $\Rightarrow$ einfacher Befehlsdekoder
- kein Support für Pipeline Stall und Flush (z.T. *Delayed Branches*)
- technische Besonderheiten, Datenabhängigkeiten, Engpässe usw. werden durch angepaßten Compiler automatisch berücksichtigt

Reduced Instruction Set Computers

zwei Beispiele

MIPS

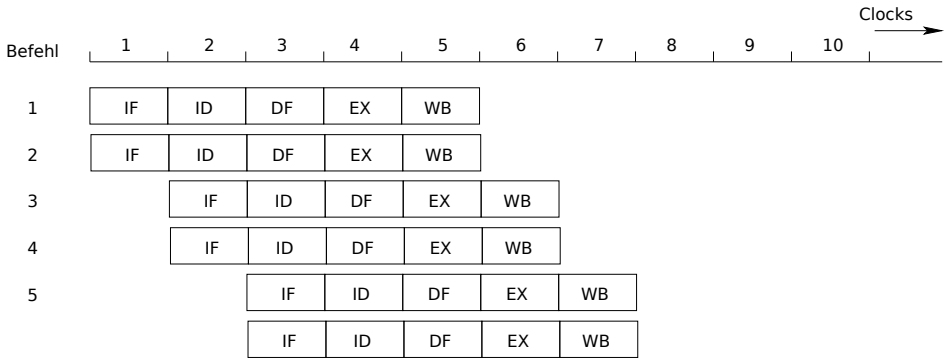
- Stanford University, Ende 70er Jahre
- 32-Bit Prozessor, 32 32-Bit Registern
- einfacher, leistungsfähiger Befehlssatz
- fünfstufige Pipeline (1 Takt/Stufe)
- Harvard-Architektur
- seit 1984 vermarktet (*MIPS R2000 ...*)
- Einsatz in eingebetteten Systemen (z.B. Audio- und Videoverarbeitung)
- breit für Lehre eingesetzt
- `http://www.mips.com`
- s. Hennessy, Patterson; Computer Organization & Design; Morgan Kaufmann,1997

ARM – Advanced RISC Machine

- Acorn Computer Limited, 1983
- 32 Bit Prozessor, 16 32-Bit Register
- nur ca. 35.000 Transistoren im Kern
- meist 1 Takt/Befehl
- bedingte ALU-Operationen vermeiden z.T. Control Hazards
- leistungsarmes Design ($<$ 1W bei 250 MHz)
- breit als IP-Core vermarktet
- im Embedded Bereich weit verbreitet (ca. 2/3 aller 32-Bit CPU's)
- `http://www.arm.com`
- s. St. Furber; ARM System Architecture; Addison Wesley, 1996

Superskalare Prozessoren

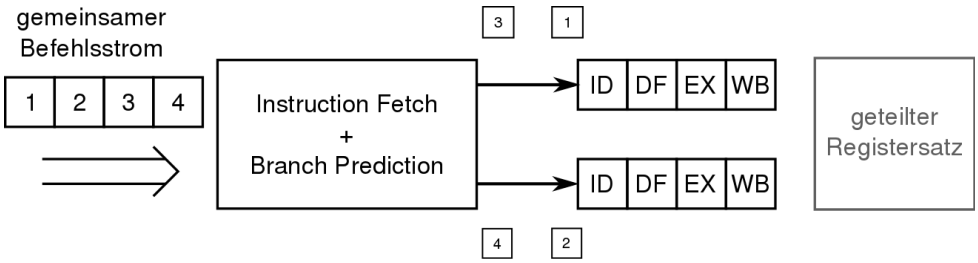
Start mehrerer Befehle / Zeiteinheit $\Rightarrow$ Multi-Issue



- 2 ... 4 Befehle / Takt gestartet
- intensive Nutzung des Pipelining

Superskalare Prozessoren

In Order Execution



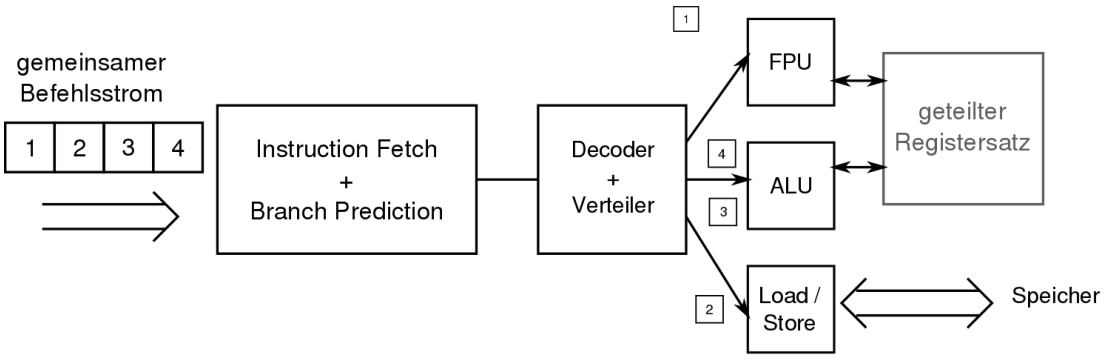
- gemeinsames Frontend liest Befehlsstrom
- Verteilung der Befehle auf mehrere parallele Pipelines

Probleme

- Datenabhängigkeiten stärker wirksam
- hoher Aufwand, wenig Beschleunigung

Superskalare Prozessoren

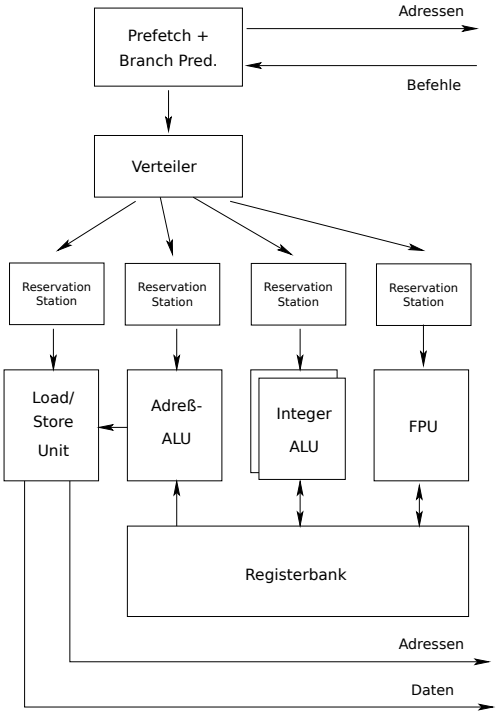
Out Of Order Execution



- Abkehr vom strikten Pipelining
- Verteilung der Befehle auf spezialisierte Verarbeitungseinheiten
  - entscheiden selbständig über Reihenfolge ihrer Teilaufträge
    ⇒ dynamische Umordnung der Befehle durch CPU

Superskalare Prozessoren

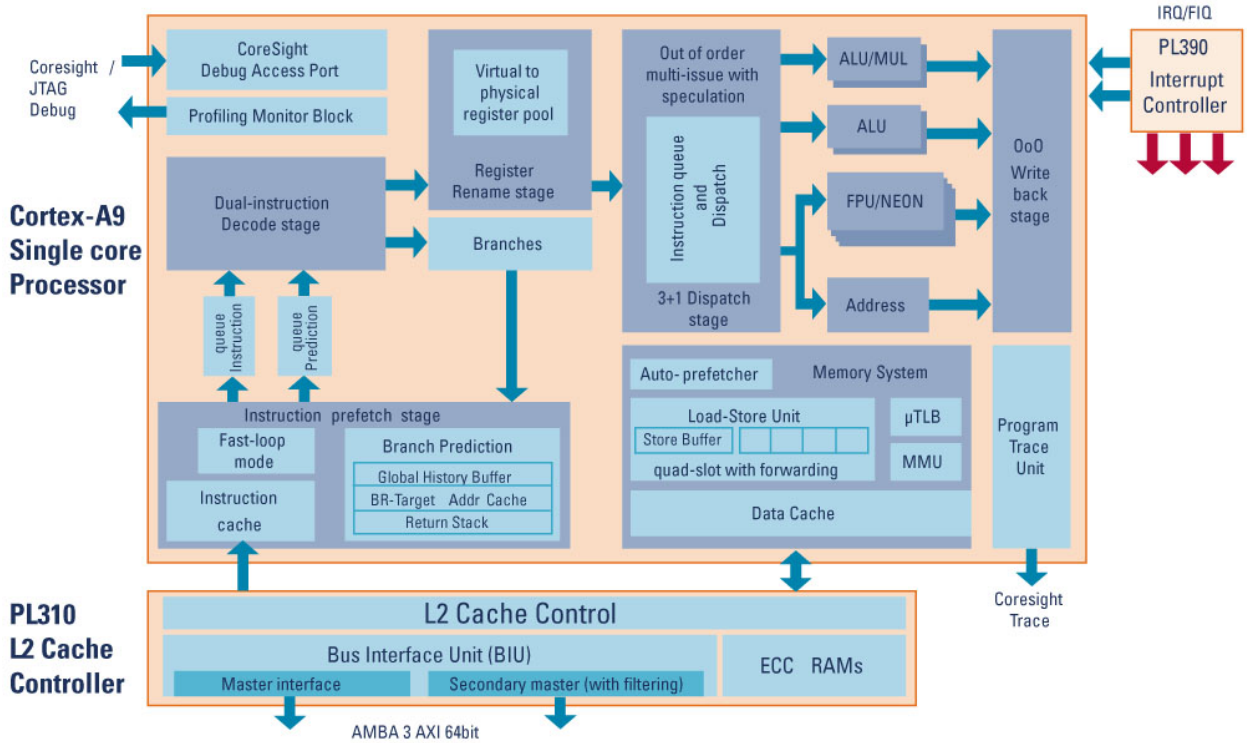
Out of Order Processing



- Abkehr vom strikten Fließbandprinzip
- gemeinsame Prefetch Unit
  - liest und dekodiert Befehlsstrom
  - macht Sprungvorhersage
- Verteilung des Befehlsstroms auf spezialisierte Verarbeitungseinheiten
  - Reservation Stations als Auftragspuffer
  - selbständige Prüfung wann Aufträge zur Ausführung bereit sind
    ⇒ dynamisches Befehlsscheduling
- zentrales *Reordering* fertiger Befehle

Superskalare Prozessoren

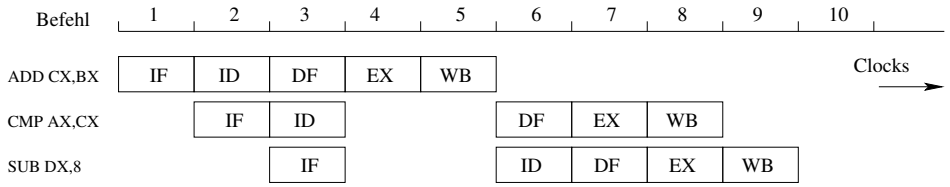
ARM Cortex A9 als Beispiel



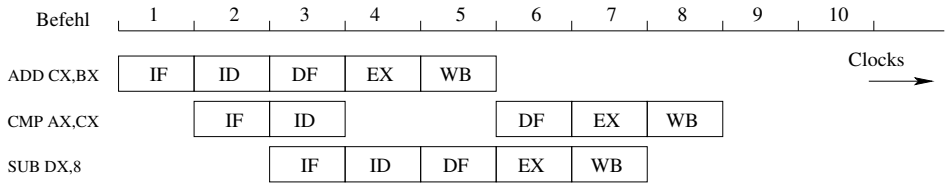
Out of Order Execution

Bearbeitung von Befehlen sobald Vorbedingungen erfüllt sind

In Order Execution:



Out of Order Execution:



unechte Datenabhängigkeiten

Register Renaming

Diese Hazards treten nur bei OOEx-Prozessoren auf.

Write After Read Schreibziel wird noch als Datenquelle benötigt

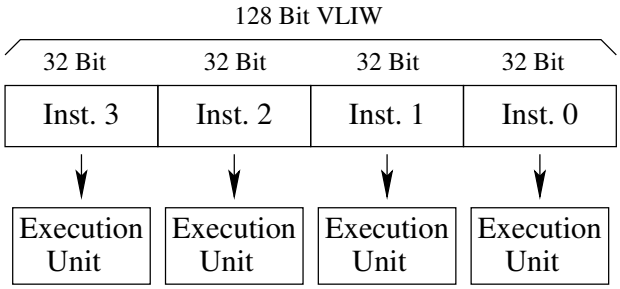
Write After Write Einhaltung korrekter Schreibreihenfolge

Register Renaming

- Zuordnung Ergebnisoperanden zu CPU-internen Schattenregistern
- mehr Schattenregister als logische Register vorhanden
- Rückspeicherung in logische Register bei Befehlsabschluß (Commit)

Very Long Instruction Word

- Explizite Parallelisierung durch Compiler (In Order Execution)
- Aufbau sehr großer Befehlswörter (VLIW), die mit mehreren parallel auszuführende Teilbefehle einzelne Funktionseinheiten parallel ansteuern.



Probleme:

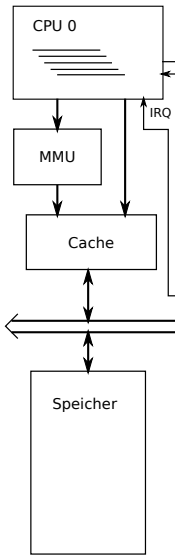
- Programmverzweigungen
- mangelhafte Codedichte
- Behandlung von Waitstates und Exceptions

Speichersystem

Die technische Realisierung des Speichers eines Rechnersystems beeinflußt wesentlich seine Leistungsfähigkeit.

Aufgaben

- Bereitstellung großer Datenmengen bei kurzer Zugriffszeit
  - Latenzzeit
  - Speicherbandbreite
- Unterstützung von Betriebssystem-konzepten
  - Zugriffsüberwachung
  - virtueller Speicher
  - dynamische Adressbindung

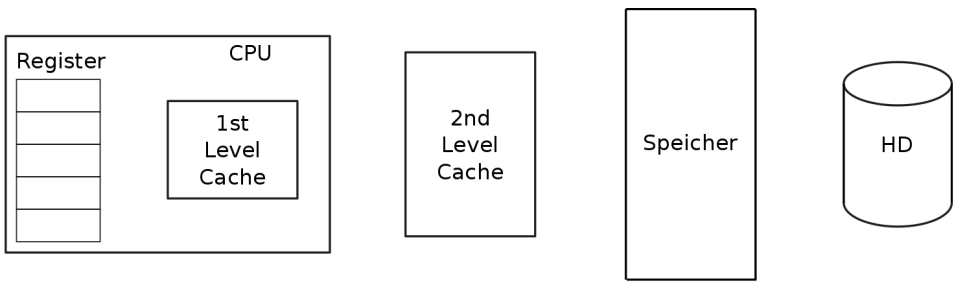


Speicherhierarchie

gestaffelte Speicherung von Daten

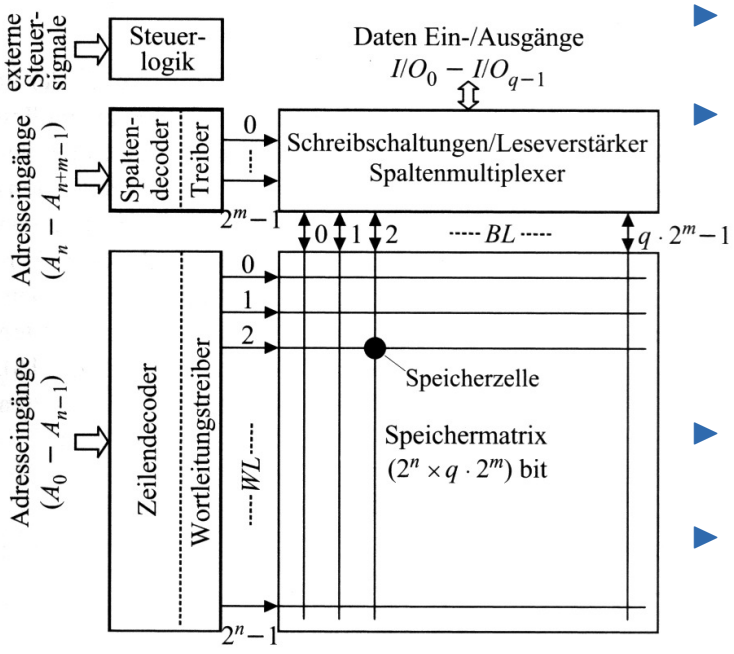
Ziel: Nutzung großer Datenvolumina bei minimalen Kosten und maximaler Geschwindigkeit

- unterschiedliche Datenmengen
- verschiedene Zugriffszeiten



Byte	10..100	8..128 k	128k..1M	1 M .. 16 G	1..10 T
Takte	1	1..4	5..10	100..200	Mio..Mrd

Grundaufbau eines Speichers

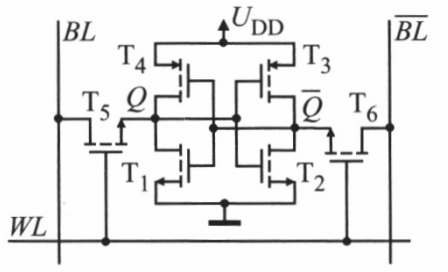


- Matrixstruktur – Aufspaltung der Adresse in Spalte und Zeile
- prinzipieller Ablauf:
  1. Auswahl und Kopie einer ganzen Zeile in Leseverstärker
  2. Auswahl einer Spalte aus Leseverstärker
  3. Rückschreiben der Zeile (nur DRAM)
- zusätzliche Steuersignale ⇒ Timing und Übertragungsrichtung
- 1 Bit / Speicherzelle (gleichzeitige Ansprache von 1, 4, 8 oder 16 Speicherzellen möglich)

SRAM- und DRAM-Speicherzelle

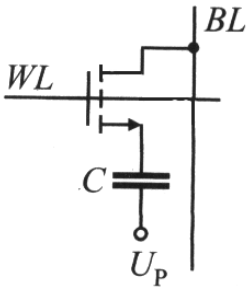
Prinzipieller Aufbau

SRAM



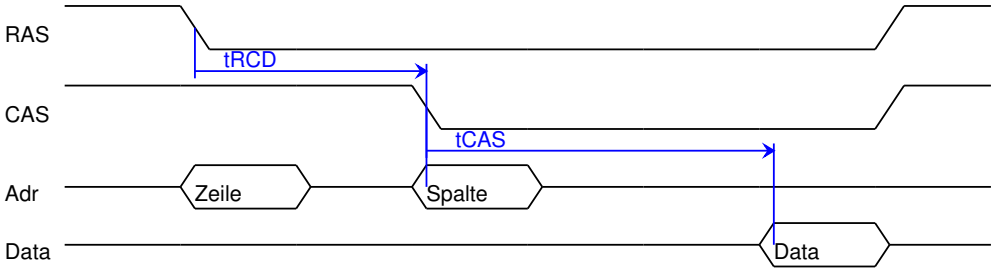
- 6-Transistorzelle
- statischer Datenerhalt bei anliegender Versorgungsspannung
- kurze Schreib-/Lesezeiten

DRAM



- 1-Transistorzelle
- billige Herstellung
- relativ langsamer Zugriff (Umladevorgänge, Rückschreiben)
- erfordert regelmäßige Auffrischung der Daten (16..64 ms)

Timingdiagramm DRAM



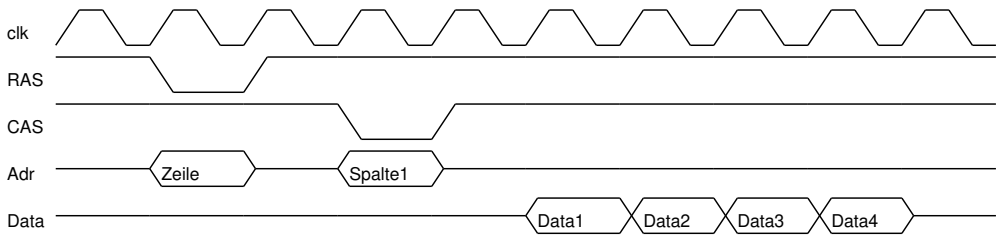
- t_{RCD} - *RAS to CAS delay* — 3 . . . 4 Bustakte
- t_{CL} - *CAS Latency* — 2 . . . 4 Bustakte
- t_{RP} - *Row Precharge time** — >= 8 Takte
- bei f_{Bus} = 200 . . . 400 MHz beträgt Zugriffszeit ca. 10 . . . 35 ns, die Zykluszeit aber ca. 20 . . . 40 ns
- max. 50 Mio. Zugriffe/s ⇒ 200 MByte/s bei 32 Bit/Zugriff

*CAS Low bis erneut RAS Low

Timingdiagramm

DRAM im Fast page Mode

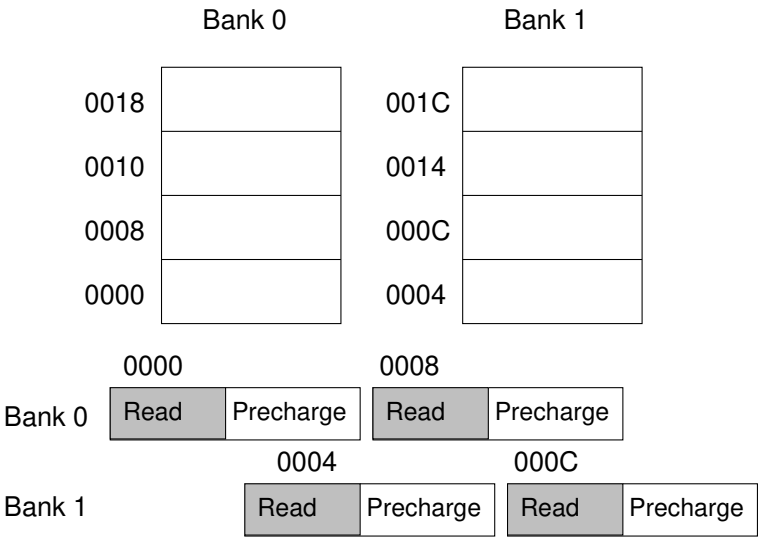
. . . optimierte Betriebsart mit höherem Datendurchsatz



- nach erstem Datenwert liefert Speicher bei jedem Bustakt einen Datenwert von Folgeadresse
- i.d.R. 2, 4 oder 8 facher Burst
- bei f_{Bus} = 200 . . . 400 MHZ beträgt Zugriffszeit auf ersten Wert ca. 10 . . . 35 ns, weitere Daten aller 2,5 . . . 5 ns
- max. 40 . . . 50 Mio. Zugriffe/s (4er Burst) ⇒ 560 . . . 800 MByte/s

Interleaving – Verschachtelung

Reduzierung der mittleren Zykluszeit bei sequentiellem Speicherzugriff



- Aufteilung des Speichers in 2 (oder 4) unabhängige Bänke
- zeitversetzter Zugriff auf beide Bänke abwechselnd
- bei sequentiellem Zugriff $\Rightarrow$ Halbierung der Zugriffszeit

Cache

Zur Realisierung
kurzer Zugriffszeiten und hoher Speicherbandbreiten
finden Caches breite Verwendung.

„Cache – ein sicherer Platz zum Verstecken bzw. Aufbewahren von Dingen."
(Webster's Dictionary.)

HOCHSCHULE MITTWEIDA UNIVERSITY OF APPLIED SCIENCES

Lokalität des Programmflusses

- zeitliche Lokalität
  erneute Nutzung momentaner Speicherinhalte in Kürze (Schleifen, UP, . . .)
- räumliche Lokalität
  Nutzung benachbarter Speicherinhalte wahrscheinlich (normaler Befehlsablauf, Feldzugriffe, . . .)

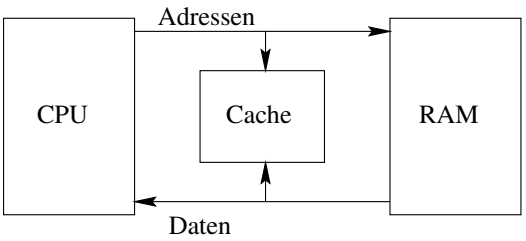
Lösungsansätze:

- Merke die letzen *n* gelesenen Daten in einem kleinen, schnellen Zwischenspeicher (SRAM) zur erneuten Verwendung.
- Lies ganze Speicherblöcke *auf Verdacht* (DRAM-Fast Page-Mode)

Arbeitsprinzip

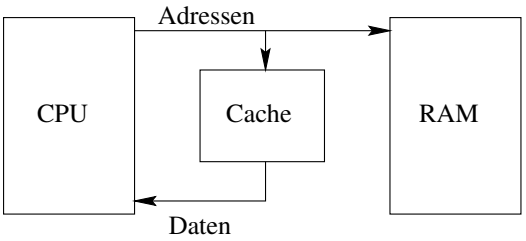
HOCHSCHULE MITTWEIDA UNIVERSITY OF APPLIED SCIENCES

Phase 1 Daten nicht im Cache enthalten ⇒ *Cache Miss*



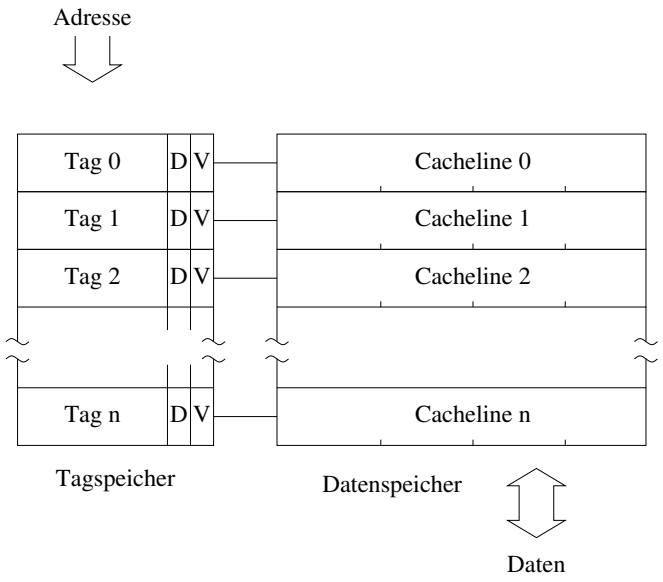
- Daten werden aus Speicher geholt und dem Prozessor zur Verfügung gestellt (Waitstates notwendig)
- Daten werden zusätzlich (samt Adresse) im Cache notiert

Phase 2 Daten im Cache enthalten ⇒ *Cache Hit*



- Daten können schnell aus Cache an CPU geliefert werden
- vorzeitiger Abbruch des Hauptspeicherzugriff

Aufbau eines Cache



- **Datenspeicher**
  Blöcke von 8. . . 64 aufeinander folgenden Bytes ⇒ *Cacheline*
- **Tag**
  Anfangsadresse der Cacheline
- **Valid Bit – V**
  Gültigkeit der Daten in Cacheline
- **Dirty Bit – D**
  wurde Cacheinhalt geändert (nur bei Datacache nötig)

Trefferrate

Maß für die Wirksamkeit eines Cache

Trefferrate (*Hitrate*) — Anzahl erfolgreicher Cachezugriffe relativ zur Anzahl aller Speicherzugriffe.

$$Hit\ Rate = \frac{Cache\ Hits}{Cache\ Hits + Cache\ Misses} = \frac{Cache\ Hits}{Number\ of\ Reads}$$

- Trefferrate hängt ab von
  - Cachegröße,
  - Bauprinzip des Cache,
  - Cache Ersetzungsstrategie,
  - Lokalität der Software . . .
- typische Trefferraten
  - Befehlscache — $\sim 98 \ldots 99,9\%$
  - Datencache — $\sim 90 \ldots 96\%$
- nur noch 10 . . . 50 % Leistungseinbuße

Platzierung und Ersetzung

Strategie kritisch für Leistungsfähigkeit

- **Platzierung**
  Auswahl einer Cacheline zur Aufnahme neuer Daten.
- **Ersetzung**
  Rückschreiben oder Verwerfen einer ausgewählten CacheLine

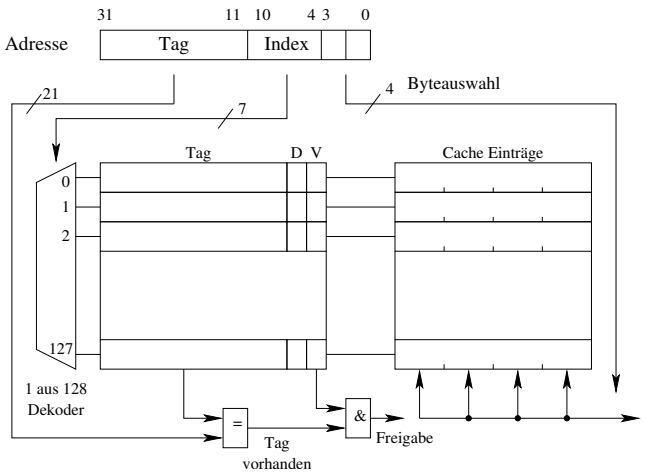
Direkt abbildend

- Teil der Adresse gibt Cacheline eindeutig vor (*Index*).
- Zu ersetzen falls belegt

Assoziativ

- Platzierung an jeder freien Stelle
- Ersetzung nur falls Cache voll
  - Random
  - Roundrobin
  - Least Recently Used – LRU

Direkt abbildender Cache



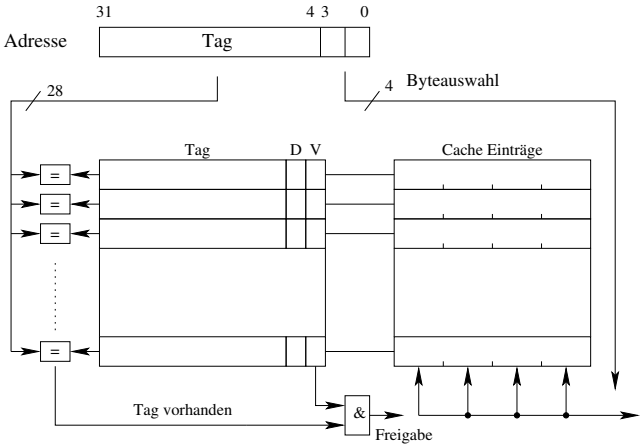
Bestandteile:

- Datenspeicher mit 128 *Cachelines* zu je 16 Byte
- 128facher Adreßspeicher für *Tags* und *Valid*-Bit
- *ein* Komparator zum Vergleich *eines* Tags mit höherwertigen Adreßbits

Funktionsprinzip:

- *Index* wählt eindeutig eine Cacheline aus.
- Vergleich des zugehörigen Tag mit höherwertigen Adreßbits ⇒ Cache Hit.
- niederwertige Adreßbits – Auswahl innerhalb Cacheline
- Valid-Bit V – Gültigkeit der Cacheline, Dirty-Bit D – Cacheinhalt geändert

Vollassoziativer Cache



Bestandteile:

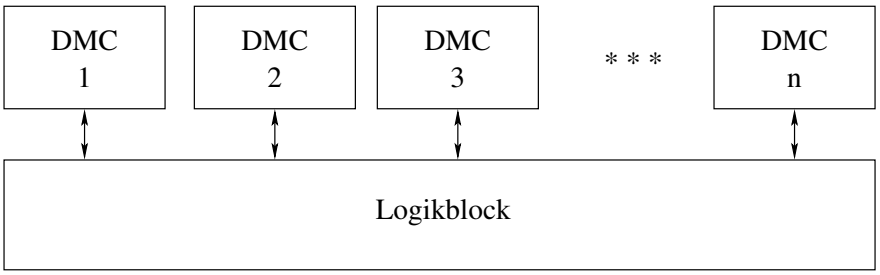
- Datenspeicher mit 128 *Cachelines* zu je 16 Byte
- 128facher Adreßspeicher für *Tags* und *Valid*-Bit
- 128 Komparatoren zum Vergleich aller Tags mit höherwertigen Adreßbits

Funktionsprinzip:

- gleichzeitiger Vergleich aller Tags mit höherwertigen Adreßbits $\Rightarrow$ Cache Hit
- niederwertige Adreßbits – Auswahl innerhalb Cacheline
- Valid-Bit V – Gültigkeit der Cacheline, Dirty-Bit D – Cacheinhalt geändert

n-Wege Assoziativ-Cache

Kompromiss aus Aufwand und Trefferrate



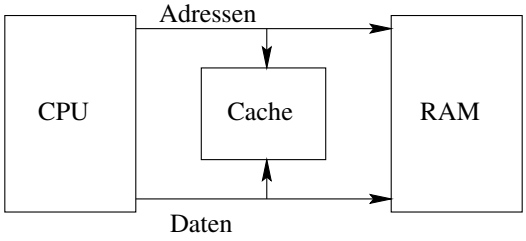
- n DM Caches werden parallel angeordnet
- Suche und Ablage in 1 aus n DM Caches
- bessere Cacheauslastung
- nur n Komparatoren nötig
- typisch 2 . . . 8 fach assoziativ

Cache
einige Ergänzungen

- i.d.R. Trennung von Befehls- und Daten-Cache
  - Befehls-Cache benötigt nur Lesefunktion
  - unterschiedliche Lokalität
- Einsatz von Cache-Hierarchien
  - gestaffelte Zugriffszeiten und Speichervolumina
  - interne Caches in Größe begrenzt
  - externer Bus langsam
- Cache auch aus anderen Gründen eingesetzt
  - virtuelle Harvard-Struktur
  - virtuelle Vergrößerung der Speicherzugriffsbreite
  - Entlastung Speicherbus in Mehrprozessor-Systemen

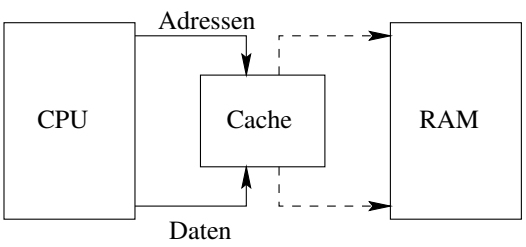
Ergänzungen
Cache beim Schreiben

Write Through



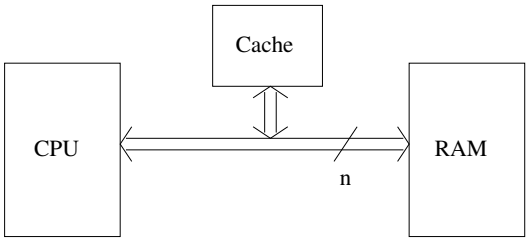
- Daten in Cache *und* Speicher schreiben
- Zeitvorteil bei erneutem Lesen
- Cache und Speicher sind kohärent

Copy Back



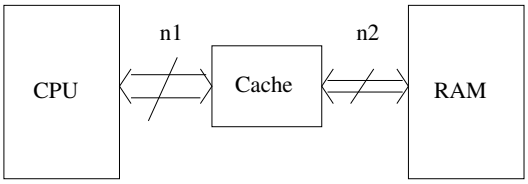
- Daten nur in Cache schreiben
- schnell, HS-Bus meist frei
- Rückschreiben in Speicher wenn
  - Platz für neue Daten benötigt wird
  - andere CPUs Daten benötigen

Lookaside Cache



- schnelle Bereitstellung des Datenwertes

Lookthrough Cache



- Entlastung des Hauptspeicherbus
- Busbreite n1 und n2 kann unterschiedlich sein

Effektive Speicherzugriffszeit
Basis zur Bewertung der Cache-Wirksamkeit

mittlere Speicherzugriffszeit (in Takten)

$$\overline{T_{mem}} = h * T_{Cache} + (1 - h) * T_{mem}$$

h - Trefferrate, T_{Cache} - Zugriffszeit Cache und
T_{mem} - Zeit zum Füllen der Cache Line

analog für Cache-Hierarchie

$$\overline{T_{mem}} = h_1 * T_{L_1} + (1 - h_1) * h_2 * T_{L_2} + (1 - h_1)(1 - h_2) * T_{mem}$$

Achtung: Speicherzugriffszeit $> \tau_{Clock} \Rightarrow$ Einfügung von Waitstates

Parallelverarbeitung

. . . erzielt eine drastische Erhöhung des Datendurchsatzes durch Verteilung der Arbeit auf mehrere Prozessoren.

Einführung
Klassifikation nach *Flynn*

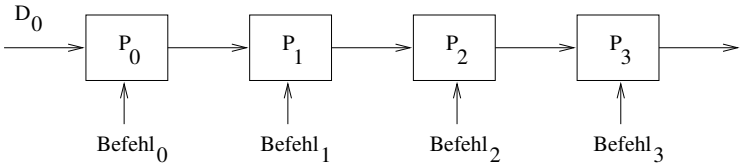
Techniken zur Parallelverarbeitung $\Rightarrow$ seit den 60er Jahren bekannt

- SISD – Single Instruction Single Data
  klassisches Singleprozessorsystem.
- MISD –Multiple Instruction Single Data
  Ein Datenelement wird nach dem Fließbandprinzip von *n* Prozessoren bearbeitet.
- SIMD – Single Instruction Multiple Data
  n Prozessoren bearbeiten *n* verschiedene Datenelemente mit *identischem Programm*.
- MIMD – Multiple Instruction Multiple Data
  n Prozessoren bearbeiten *n* Datenelemente mit jeweils *individuellem Programm*.

MISD

Multiple Instruction Single Data

Bearbeitung jeweils eines Datenelementes nach dem Fließbandprinzip. Jeder Prozessor bearbeitet den Wert mit *eigenem* Befehl.

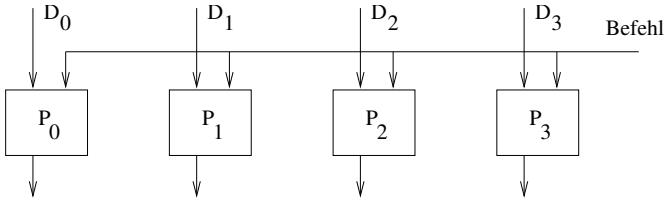


- Aufgrund geringer Flexibilität kaum für universelle Verarbeitungsaufgaben geeignet.
- Breiter Einsatz in fest konfigurierten hierarchischen Datenerfassungs- und Verarbeitungssystemen.
  (z.B. Sensorvorverarbeitung, Regelung, . . .)

SIMD

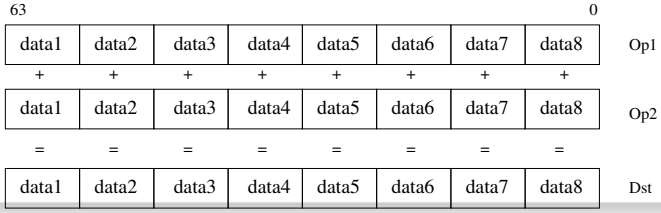
Single Instruction Multiple Data

n Prozessoren bearbeiten gleichzeitig *n* verschiedene Datenelemente mit *einem identischen* Befehl.



- Vektorrechner zur Lösung von Matrixoperationen (z.B. $A = B * C + D$);
  In 60er Jahren verbreitet eingesetzt.
- Effektive Verarbeitung gepackter Audio- und Grafikdaten in modernen Prozessoren (*MMX*, *SSE2*, *3DNow*)

  z.B. Addition zweier gepackter Operanden mit je 8 8-Bit Werten



MIMD

Multiple Instruction Multiple Data

n Prozessoren bearbeiten *n* Datenelemente mit jeweils *individuellem Programm*.
Varianten:

- MIMD mit zentralem Speicher und gemeinsamen Adreßraum
  (UMA-Architektur – Uniform Memory Access, SMP – Symmetric Multiprocessing)
- MIMD mit verteiltem Speicher und gemeinsamen Adreßraum
  (NUMA-Architektur – Non Uniform Memory Access
- MIMD mit privatem Speicher
  Cluster

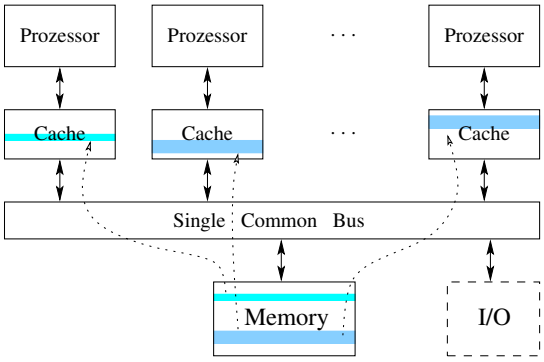
Hauptprobleme beim Einsatz von MIMD-Rechnern:

- Vollständige Parallelisierung, Verteilung der Arbeit
- Synchronisation von Datenzugriffen und Befehlsreihenfolgen
- Bandbreite des Kommunikationsmediums

MIMD

zentraler Speicher, gemeinsamer Adreßraum

Mehrere CPU's teilen sich *einen* Speicher mit *homogenem* Datenbestand.

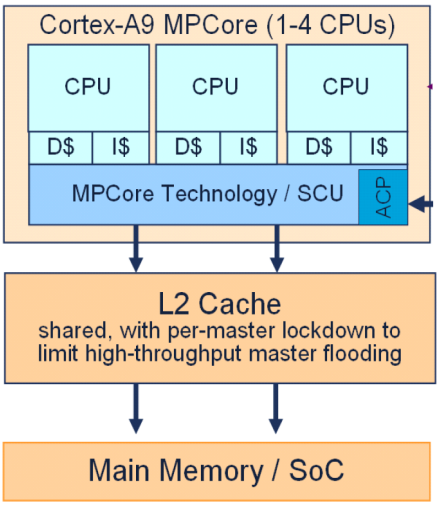


- Jeder Prozessor hat Zugriff auf den gesamten Adressraum.
- *Arbitration* regelt exklusiven Buszugriff ⇒ Multimaster-Bus
- Exklusive Speichernutzung via Locks
- Bandbreite von Bus und Speicher stellt Engpass dar.
- Lokale Caches entlasten gemeinsamen Speicher
  (Beachte Cache-Kohärenz).

Systeme aus 2 . . . 8 Prozessoren.

Multi-Core CPUs

heutige Realisierung des MIMD in Workstations

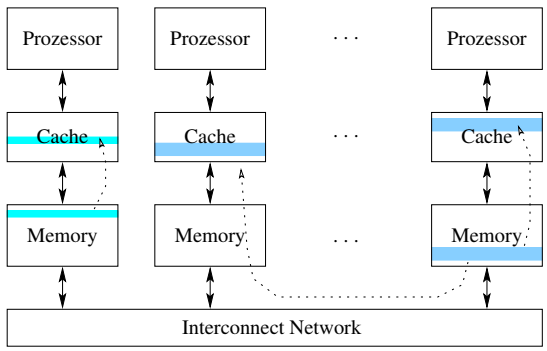


- On-Chip Integration von 2 . . . 16 CPU-Cores
- CPU's z.T. mit Hyper-Threading
- jeweils eigener L1-Cache
- gemeinsamer L2-Cache (1..8 MB) on Chip
- jeweils eigene MMU per CPU

MIMD

verteilter Speicher, gemeinsamer Adreßraum

Ein Speicher per CPU als Teil des gemeinsamen Adreßraums mit *homogenem* Datenbestand.



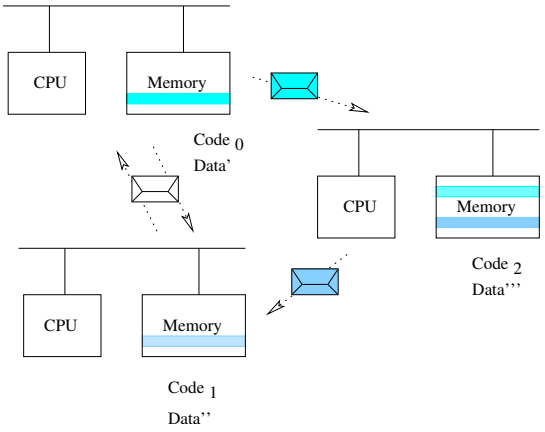
Systeme mit bis zu 256 Knoten (1 . . . 4 CPU's / Knoten).

- Jeder Prozessor hat Zugriff auf den gesamten Adressraum.
- Zugriffszeit variiert mit Position des angesprochenen Datenwertes.
- Geringer Bedarf an Speicher- und Netzwerkbandbreite bei geeigneter Verteilung der Daten.
- Datentransport übers Netzwerk
  - hardwaregesteuert (für SW transparent)
  - vom Betriebssystem gekapselt (z.B. analog virtueller Speicher)

MIMD

verteilter, privater Speicher

Jede CPU besitzt einen *privaten* Speicher mit separatem Adressraum.



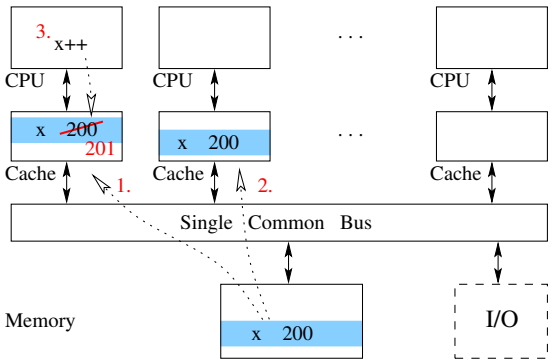
- Datenbestand ist zwischen *n* Prozessoren aufgeteilt.
- Datenaustausch erfolgt über explizites *Message Passing*.
- Synchronisation ist separat zu organisieren.
- Voraussetzung effektiver Arbeit $\Rightarrow$ schnelle Kommunikation mit universeller Struktur
  - Crossbar Switch
  - Routing
  - Bustopologie

Cluster mit 32 . . . 16384 Knoten

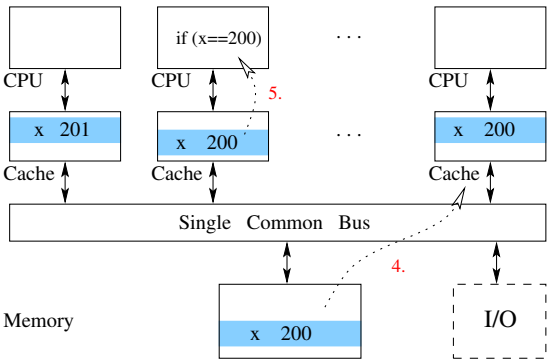
Cache und Multi Core

Cache Kohärenz

Ausgangspunkt:



Mögliche Konflikte:



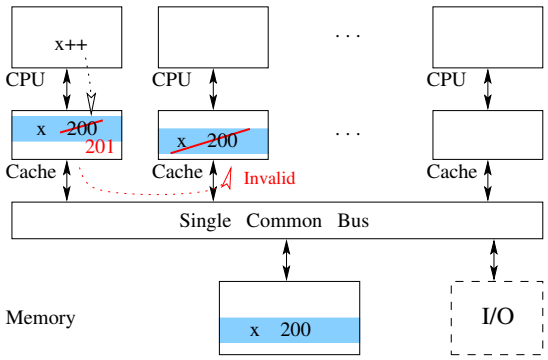
1. CPU 1 liest *x*
2. CPU 2 liest *x*
3. CPU 1 inkrementiert *x*
4. CPU n liest *x*
5. CPU 2 benutzt *x* erneut

Bus Snooping

Überwachung Busverkehr durch jeden Cache – *Snooping* Logik

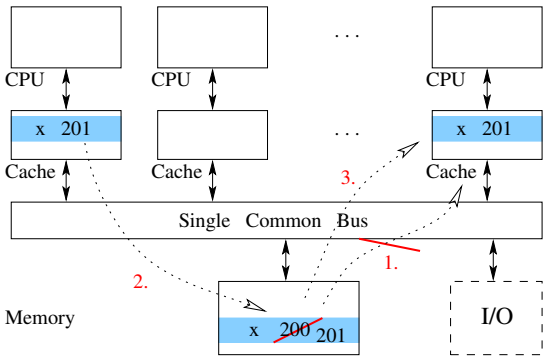
HOCHSCHULE MITTWEIDA UNIVERSITY OF APPLIED SCIENCES

Schreiben von Daten:



1. Alle Kopien des Wertes als *invalid* markieren (Broadcast)
2. Cacheinhalt verwerfen

Cache Miss:



1. Lesen Hauptspeicher abbrechen
2. Write-Back Cache Line
3. Erneutes Lesen des Speichers

HOCHSCHULE MITTWEIDA UNIVERSITY OF APPLIED SCIENCES

Betriebssystem-Support

Betriebssysteme benötigen Hardware-Support zur Durchsetzung von Schutz- und Verwaltungsaufgaben

Schutzmechanismen

Sicherung der korrekten Funktion des Rechners und der Datensicherheit

unerwünschte Wechselwirkung

Möglichkeiten zur gegenseitigen Beeinflussung von Prozessen

Änderung von *Speicherbereichen eines anderen Task*
verfälschte Daten, Änderung des Ablaufes, Absturz, Datendiebstahl, ...

Zugriff auf systemkritische Funktionseinheiten
CPU-Betriebsart, Interruptsystem, Reset, Taskwechsel, Power Down, ...

Nutzung von anderweitig belegten Systemressourcen
I/O-Schnittstellen, Dateien, globale Datenstrukturen, ...

Beeinflussung aus technischer Sicht:

- Zugriff auf *unzulässige* Speicherbereiche bzw. *inkorrekte Nutzung* zulässiger Bereiche
- Ausführung *systemkritischer Befehle* (I/O, Sperren Interrupts, ...)

Schutzmechanismen

Grundkonzept

- Trennung von *Betriebssystem-Software* und *Anwender-Software*
- abgestuftes System von Nutzungsrechten bzw. *Privilegien*
  - Zugriff auf bestimmte Speicherbereiche
  - Nutzung ausgewählter Hardwarekomponenten
  - Manipulation von Dateien und Datenstrukturen
- *hardwareseitige Überwachung* der Einhaltung der Privilegien
- bei Verletzung
  - Auslösung einer Exception
  - Behandlung durch Betriebssystem

Betriebssystem- versus Anwender-Software

Betriebssystem

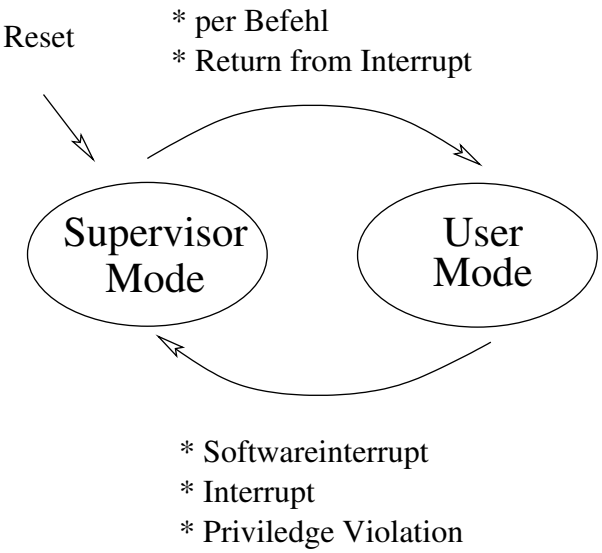
- ausführlich getestet
  $\Rightarrow$ zuverlässig
- darf von Anwendersoftware nicht manipuliert werden
- richtet Prozesse ein, koordiniert Speicher- und Ressourcenvergabe, übernimmt Ein-/Ausgabe
- organisiert Überwachung der Anwendersoftware
- besitzt alle Rechte

Anwendersoftware

- weniger zuverlässig
- Zugriff nur auf *explizit zugewiesenen* Speicher erlaubt
- *systemkritische* Operationen ***müssen*** übers Betriebssystem ausgeführt werden
- kann Privilegien und deren Überwachung nicht ändern
- Verletzung der Nutzungsrechte wird durch Betriebssystem behandelt

Supervisor- / User - Konzept

CPU-Betriebsarten mit unterschiedlichen Rechten



Supervisor Mode

- jegliche Speicherzugriffe sind erlaubt
- alle Befehle sind ausführbar

User Mode

- Überwachung von Speicherzugriffen
- keine *privilegierten Befehle* ausführbar

privilegierte Befehle

unterstützen systemkritische Operationen

- erfordern ausreichende Privilegierung der CPU (*Supervisor Mode*) zur Ausführung
- Ausführung im *User Mode* löst Exception aus $\Rightarrow$ *Priviledge Violation*

typische Vertreter sind:

- Prozessor Halt
- Enable / Disable Interrupt
- Taskumschaltung
- IN- / OUT-Befehle
- Betriebsartumschaltung
- Verwaltung von Speicherschutz, Cache . . .

Speicherverwaltung

Memory Management Units (MMU)

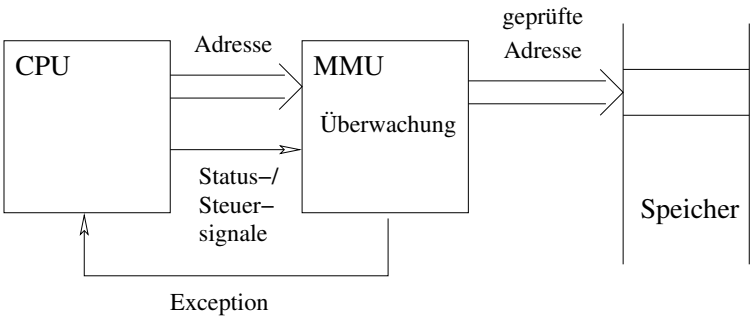
- Speicherschutz – Prüfung von Adressen auf korrekte Verwendung
- *Address Translation* – Umsetzung virtueller in physische Adressen
  - dynamische Adressbindung
- Verwaltung von Speicherbereichen mittels
  - Segmentierung
  - Paging
- Unterstützung *virtuellen Speichers*

MMU - Aufgaben

Überwachung von Speicherzugriffen

Prüfung der von der CPU generierten Adresse auf *Zulässigkeit* und *korrekte Verwendung*

- andernfalls Auslösung einer Exception



Steuersignale:

- Supervisor-/User-Mode
- Read/Write
- Program/Data

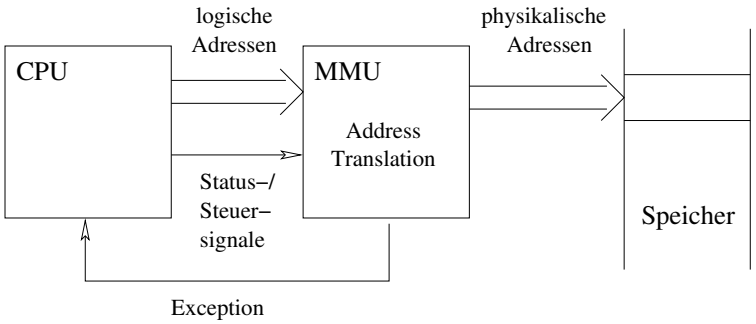
Prüfkriterien:

- Zugriff generell erlaubt
- Executable
- Read Only

MMU - Aufgaben

Adressumsetzung – Address Translation

- Umsetzung logische/virtuelle Adresse ⇒ physikalische Adresse
  - programmierbare 1:1 Transformation



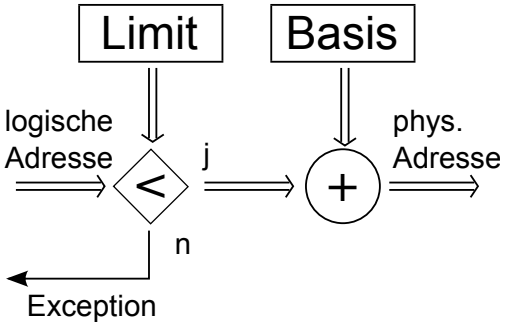
- erlaubt wahlfreie Anordnung von Programm und Daten im Speicher
- *dynamische Adressbindung* ⇒ Verschiebung zur Laufzeit möglich
- eingesetzte Verfahren
  - Paging
  - Segmentierung

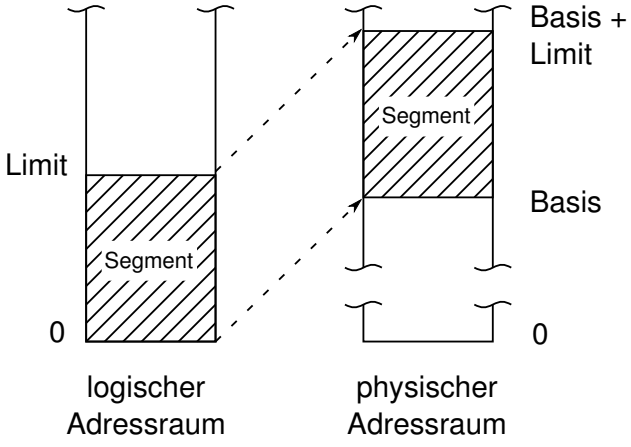
Segmentierung

ein Verfahren der Adressumsetzung

Segment kontinuierlicher Speicherbereich mit

- Limit – variable Länge
- Basis – wählbare phys. Startadresse

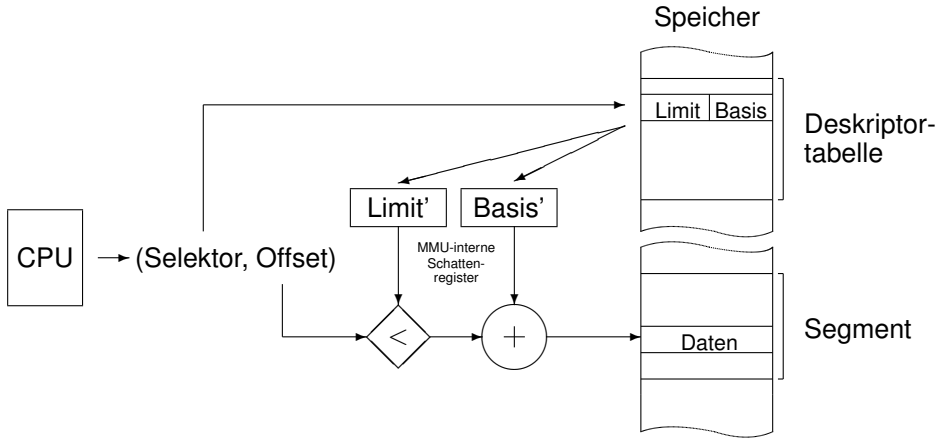




Segmentierung

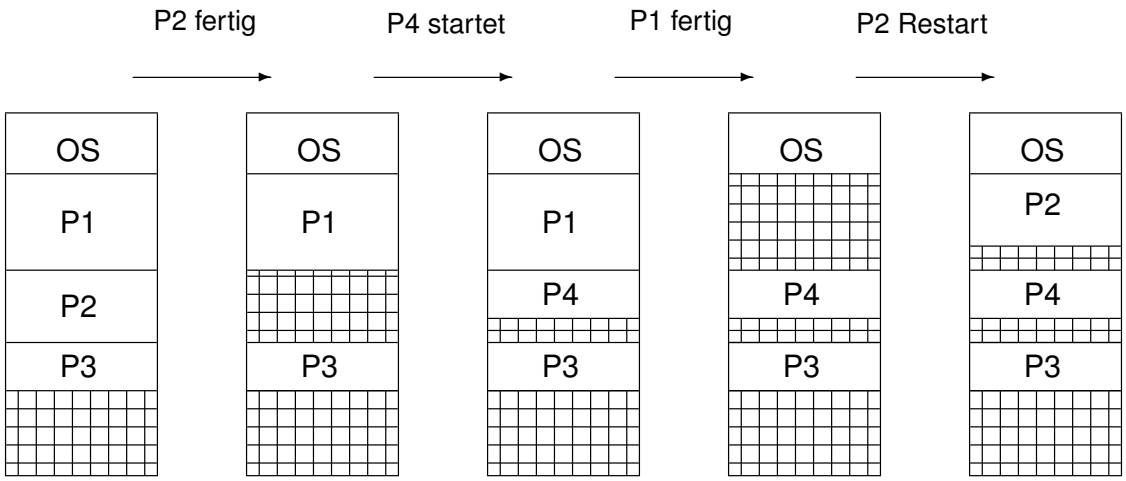
etwas näher betrachtet

- Verwaltung der Speichersegmente erfolgt über *Deskriptor-Tabelle*
- Deskriptor enthält
  - Segmentlage und- größe
  - Schutzrechte des Segmentes



Nachteil der Segmentierung

mögliche Fragmentation des physischen Speichers



OS — Operating System

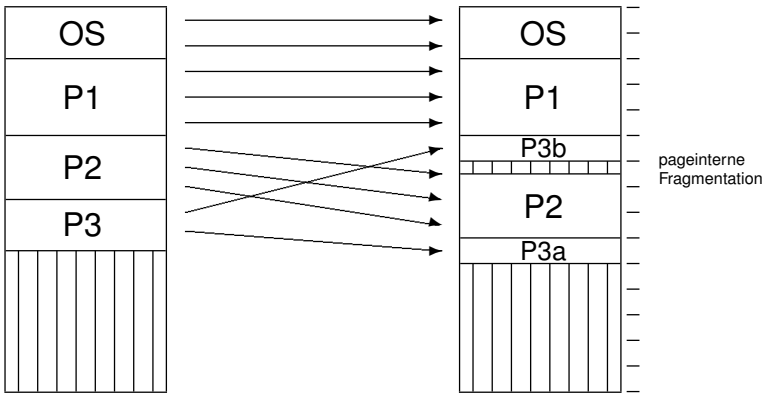
P*n* — Prozeß *n*

freier Speicher

Adressumsetzung mittels Paging

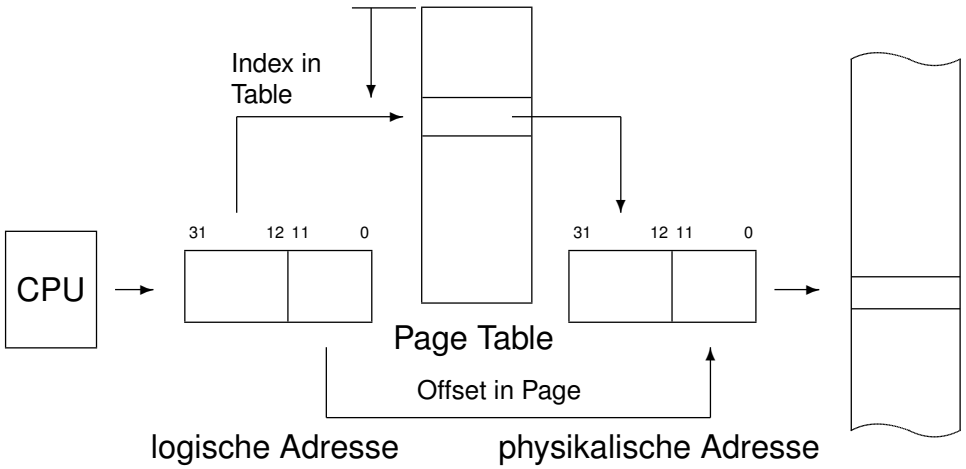
alternatives Verfahren zur Adressumsetzung

- Unterteilung des Speichers in Einheiten fester Größe ⇒ Pages
  - typ. 4, 16, 64 kB od. 1 MB
- wahlfreie Zuordnung virtueller zu physikalischer Page-Nummer
- Zuordnung mehrere Pages zu einem logischen Bereich



Page Translation Table

tabellengesteuerte Adressumsetzung



- Page Table liegt im Speicher (auf fester Adresse)
  - logische Seitennummer ist Index in Tabelle
  - Tabelle enthält u.a. physische Seitennummer
- häufige Einträge in Translation Lookaside Buffer TLB gehalten

Aufbau der Page Table

- Tabelle enthält für jede logische/virtuelle Page einen Eintrag
  - logische Page-Nummer ist Index in die Tabelle

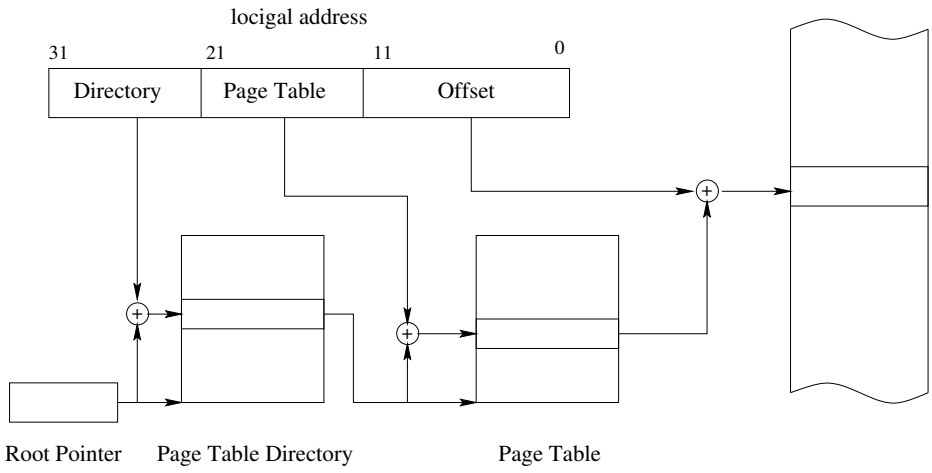
Page Table Eintrag

(für 32 Bit CPU)

- 4 Byte / Eintrag
- enthält
  - physische Page-Nummer (12...20 Bit)
  - Zugriffsrechte der Seite (typ. 3...5 bits)
  - Zusatz-Informationen für die Unterstützung *virtuellen Speichers*
    - A – Accessed
    - D – Dirty
    - P – Present

mehrstufige Umsetzung

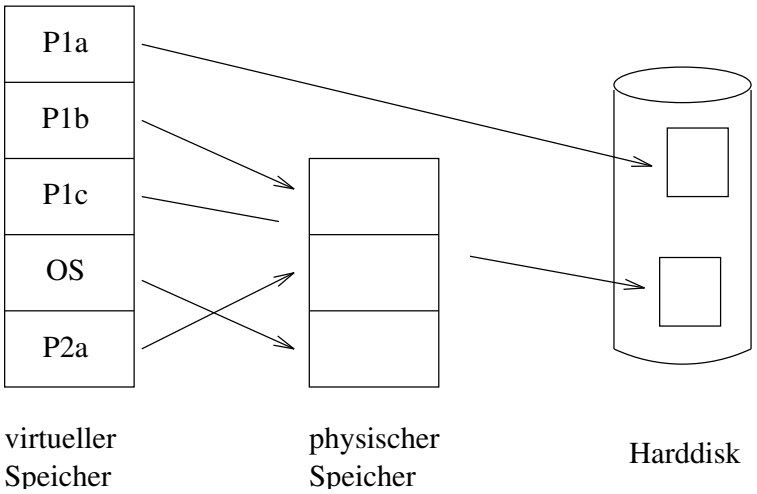
Reduzierung Tabellengröße



- Aufteilung der logischen Adresse in mehrere Teile (Offset, Pageauswahl, Auswahl der Page Table aus Page Table Directory)
- nur für real existierenden Speicher müssen Teil-Tabellen vorhanden sein

Virtueller Speicher

Grundidee

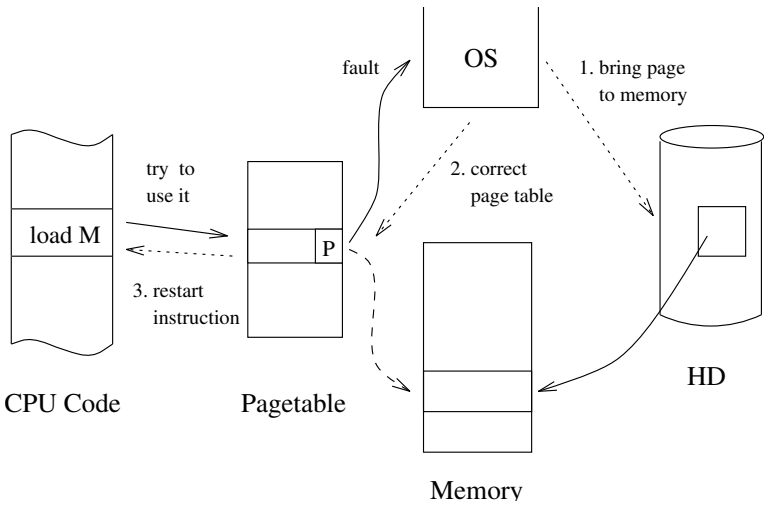


- Programme müssen nicht als ganzes im Speicher sein
- Stückelung erfolgt auf Page-Basis
  (einige Pages sind im Speicher, andere auf HD/SSD ausgelagert)

Virtueller Speicher

Paging on Demand

- *Present Bit* im Page Entry zeigt an, ob Seite im Speicher ist



- Zugriff auf eine ausgelagerte Page ⇒ Page Fault Exception
  - Exceptionhandler (Betriebssystem) führt Schritte 1 bis 3 aus