

VO 050041:

Technische Grundlagen der Informatik

Begleitende Folien zur Vorlesung
Wintersemester 2016/17

Vortragende: Peter Reichl, Andreas Janecek

Zuletzt aktualisiert: 30. November 2016

Teil 5:

Rechnerarchitekturen

Outline: Teil 5

- 1 Einführung und Formale Aspekte
 - Klassifikation
 - Design und Performance
 - Quantitative Methoden
- 2 Standard-Rechnerarchitektur
 - Abstraktionsebenen
 - Prozessor
- 3 Speicherverwaltung
 - Virtueller Speicher und Paging
 - Caching
- 4 Externer Speicher
 - Festplatten
 - Disk Scheduling
 - SSD
- 5 Skalare und superskalare/parallele Architekturen
 - CISC und RISC
 - Pipelining
 - Superskalare Architekturen

Literatur

- Mikroprozessortechnik (Wüst): Kapitel 6, 7, 10, 11, 13
- Computer Architecture (Hennessy, Patterson): Kapitel 1, 2, Appendix A, B, C

Überblick

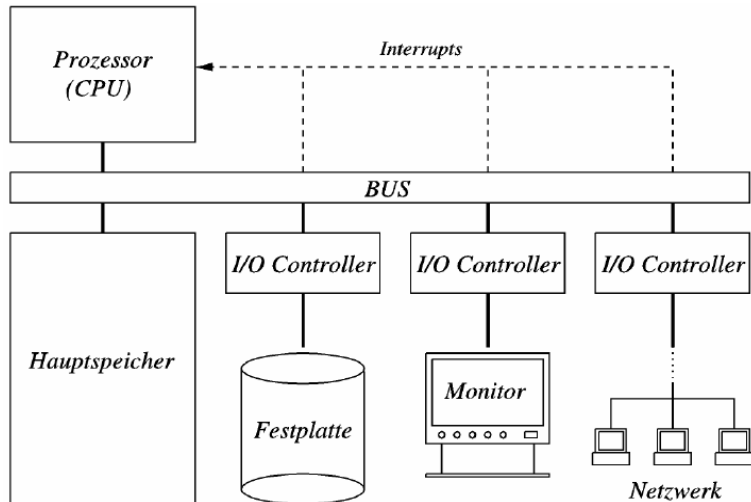
- 1 Einführung und Formale Aspekte
 - Klassifikation
 - Design und Performance
 - Quantitative Methoden
- 2 Standard-Rechnerarchitektur
 - Abstraktionsebenen
 - Prozessor
- 3 Speicherverwaltung
 - Virtueller Speicher und Paging
 - Caching
- 4 Externer Speicher
 - Festplatten
 - Disk Scheduling
 - SSD
- 5 Skalare und superskalare/parallele Architekturen
 - CISC und RISC
 - Pipelining
 - Superskalare Architekturen

Aufbau von Rechnern

Grundlegende Komponenten eines Computers

- **(Mikro-)Prozessor (CPU):** verarbeitet Daten
- **Speicher:** Arbeitsspeicher, Massenspeicher
- **Ein-/Ausgabegeräte:** Tastatur, Display, Drucker, ...
- **Busse:** Interne Kommunikationsleitungen

Aufbau von Rechnern



Klassifikationskriterien

Rechner lassen sich nach mehreren Kriterien klassifizieren:

- Preis/Leistungsfähigkeit
 - **Laptop:** mobile und/oder nomadische Nutzung durch einen Anwender, Netzanbindung über WLAN
 - **Personal Computer (PC):** typischerweise ein Anwender, netzwerkfähig (d.h. Ressourcen können von anderen Benutzern mitverwendet werden)
 - **Workstation (Arbeitsplatzrechner):** in der Regel mit Mehrbenutzer-Betriebssystem und Netz ausgestattet
 - **Mainframe (Grossrechner):** Hochgeschwindigkeitsrechner, von vielen Anwendern gleichzeitig benutzt
 - Grenzen zwischen diesen Kategorien sind fließend!
- Maschinenbefehlssatz (CISC vs RISC) → später!!
- Anzahl der Daten- und Befehlsströme → Flynn

Rechner-Klassifikation nach Flynn

Anzahl der vorhandenen Datenströme

- SD: Single Data Stream
- MD: Multiple Data Streams

Anzahl der vorhandenen Befehlsströme

- SI: Single Instruction Stream
- MI: Multiple Instruction Streams

- **SISD**
- **SIMD**
- MISD
- **MIMD**

SISD

SISD - Single Instruction, Single Data

- **Sequenzielle Programmausführung (ein Prozessor)**
- **Princeton-Architektur („von Neumann-Rechner“)**
 - **Gemeinsamer Speicher für Instruktionen und Daten**
 - Streng sequentieller (deterministischer) Programmablauf garantiert
 - Bei wechselndem Daten-/Programmspeicher-Bedarf gute Auslastung: “Von Neumann Bottleneck”
- **Harvard-Architektur:**
 - **Getrennter Speicher für Instruktionen und Daten**
 - Zwei unabhängige (unterschiedlich breite) Busse
 - Daten und Befehle können gleichzeitig geladen werden
 - Aber: Race conditions, nicht-deterministischer Programmablauf

SIMD

SIMD - Single Instruction, Multiple Data

- **Instruktion gleichzeitig auf mehrere Datensätze ausführen**

- $$\begin{pmatrix} 3 \\ -2 \\ 0 \\ 7 \end{pmatrix} \cdot 9 = \begin{pmatrix} 27 \\ -18 \\ 0 \\ 63 \end{pmatrix}$$

- Vektorrechner: 1 Operation; SISD: 4 Operationen
- Mathematische, physikalische Berechnungen, auch Grafik
- General Purpose Graphics Processing Unit (GPGPU)
- Auch viele moderne Mikroprozessoren besitzen inzwischen SIMD-Erweiterungen

MIMD

MIMD - Multiple Instruction, Multiple Data

- Mehrere Prozessoren, mit Leitwerk und Rechenwerk
- Jeder arbeitet eigenständig
- MIMD Computer kann mehrere unabhängige Programme gleichzeitig ausführen

Computerdesign

Computerdesign

- **Wir wollen einen neuen, *optimalen* Computer kaufen/bauen**
- **Oder wir wollen verschiedene existierende vergleichen**
- **Was aber ist eigentlich *optimal*?**
 - Welche Tasks?
 - Welche Programme?
 - Welcher Benutzer?
 - Schnelligkeit?
 - Zuverlässigkeit?
 - Physische Robustheit?
 - ...
- **Wir brauchen eine Metrik, um Vergleichbarkeit herzustellen**

Beispiele

Desktops

- Optimum an Preis und Performance (Preis/Leistung)
- Oft: Preis für Mindestanforderungen (z.B. Office)

Server

- Verfügbarkeit und Zuverlässigkeit
- Skalierbarkeit: Rechenleistung, Speicher, I/O, ...
- Durchsatz: Transaktionen pro Zeiteinheit

Dynamische Entwicklung

- Stromverbrauch pro Durchsatz
- ...

Performance

Wichtige Kriterien

- **„Computer A ist schneller als Computer B“ muss quantifizierbar sein**
- **Durchsatz (Throughput, X [Aufträge/s]):** Mittlere Anzahl von erledigten Aufträgen pro Zeiteinheit. Je größer, desto besser.
- **Ausführungszeit (Execution Time, ET , [s]):** Zeit zwischen Beginn und Ende eines Auftrags. Je kleiner, desto besser.

Execution time und Performance

Rechner X ist n-mal so schnell wie Rechner Y

$$\frac{ET_Y}{ET_X} = n$$

ET ist reziprok zu Performance

$$n = \frac{ET_Y}{ET_X} = \frac{Performance_X}{Performance_Y}$$

Beispiel

Durchsatz von X ist 1.3 mal so hoch wie von Y \Rightarrow auf Rechner X können 1.3 mal so viele Tasks pro Zeiteinheit erledigt werden wie auf Rechner Y.

Zeit-Begriffe

Execution Time

- **Verschiedene Definitionen möglich**
- **Response time** (wall clock time): Gesamte Zeit vom Absenden der Anfrage bis zum Eintreffen der Antwort.
- **CPU time**: Nur Zeit, in der CPU aktiv ist (z.B. keine Wartezeiten)
- **User CPU time**: Nur Zeit, die CPU für das Programm aufwendet (user mode)
- **System CPU time**: Nur Zeit, die CPU für Betriebssystemaufgaben (für Programm) aufwendet (kernel mode)
- **Linux/UNIX**: `time` Kommando, z.B.: `time firefox`

Leistungsvergleich

- Welcher der Computer hat die höchste Performance?

	Computer A	Computer B	Computer C
Programm 1	1s	10s	60s
Programm 2	199s	100s	60s
Programm 3	50s	140s	130s
Total ET	250s	250s	250s

- **Gesamt-Ausführungszeit vereinfacht Interpretation**

Leistungsvergleich - Möglichkeiten

Arithmetisches Mittel

$$\overline{ET} = \frac{1}{n} \cdot \sum_{i=1}^n ET_i \quad [s]$$

Gewichtete Ausführungszeit (weighted ET)

- Für ungleich verteilte Workloads
- Z.B. 20% der Tasks betreffen P1, 80% P2, dann sind Gewichte $g_1 = 0.2$ bzw. $g_2 = 0.8$

$$WET = \sum_{i=1}^n g_i \cdot ET_i \quad [s]$$

Designprinzipien

Designprinzipien

- „Make the common case fast“
- **Häufig auftretende Ereignisse schnell abarbeiten**
- Seltene Ereignisse nicht übermäßig beachten
- **Welche Gesamt-Verbesserung können wir erwarten?**

Gesetz von Amdahl

Definiert den Gesamt-Speedup S , der mit einer Teil-Verbesserung erzielt werden kann!

$$S = \frac{ET_{alt}}{ET_{neu}}$$

$$ET_{neu} < ET_{alt}$$

$$S > 1$$

Gesetz von Amdahl

Fraction Enhanced F_E

- **Anteil** der Gesamt-ET, der verbessert (beschleunigt) ist
- Bsp: Von einer Gesamt-ET von 60s wurden 20s verbessert
 $\Rightarrow F_E = \frac{20}{60} = \frac{1}{3}$
- Es gilt: $F_E \leq 1$

Speedup Enhanced S_E

- **Faktor**, um den der erwähnte Anteil der Gesamt-ET (F_E) verbessert (beschleunigt) ist
- Bsp: Für das gleiche Teil-Problem statt 5s nur noch 2s benötigt: $\Rightarrow S_E = \frac{5}{2}$
- Es gilt: $S_E > 1$

Gesetz von Amdahl

Herleitung

$$S = \frac{ET_{alt}}{ET_{neu}} = \frac{ET_{alt}}{ET_{alt} \cdot \left(\underbrace{(1 - F_E)}_{\text{unverbessert}} + \underbrace{\frac{F_E}{S_E}}_{\text{verbessert}} \right)} \Rightarrow$$

Amdahl's Law

$$\Rightarrow S = \frac{1}{(1 - F_E) + \frac{F_E}{S_E}}$$

Gesetz von Amdahl

Anwendung

- **Gibt Hinweise, inwieweit eine Verbesserung überhaupt die Gesamt-Performance steigert (steigern kann)**
- **Gibt eine Grenze des Speedups an, der vor allem von der F_E abhängt** ($\lim_{S_E \rightarrow \infty} S = \frac{1}{1-F_E}$)
- Hilft auch bei der Beurteilung, welche von zwei Verbesserungsvarianten besser ist!

Amdahl: Beispiel

Beispiel fernab der IT (?)

- Wir arbeiten alleine an einem Projekt
- 40% der Zeit für Meetings, Koordination, etc. drauf
- 60% der Zeit kann produktiv gearbeitet werden
- Projekt verzögert sich, 5 neue Mitarbeiter eingestellt
- Einfache Annahme: 40:60-Verhältnis gilt weiterhin
- Mit welchem Speedup ist maximal zu rechnen?
- Warum wird dieser praktisch wahrscheinlich nicht halten?

Amdahl: Beispiel

$$\begin{aligned} S &= \frac{1}{(1 - F_E) + \frac{F_E}{S_E}} = \frac{1}{(1 - 0.6) + \frac{0.6}{6}} \\ &= \frac{1}{0.4 + 0.1} = \frac{1}{0.5} = \mathbf{2} \end{aligned}$$

- **Mitarbeiterzahl versechsfacht** ($1 \Rightarrow 6$), **aber nur Speedup von 2!**
- Absurdes Experiment: Mitarbeiterzahl unendlich groß machen: $S = \frac{1}{1 - F_E + 0} = 2.5$

Amdahl: Beispiel 2

Beispiel aus der IT

- Gesamt-ET eines Programm: 30% CPU-Berechnungen, 70% Warten auf Daten aus Arbeitsspeicher
- Zwei alternative Varianten der Verbesserung:
 - 1 Neue CPU kaufen, die 1.5 mal so schnell rechnen kann
 - 2 Neues Mainboard/Speicher kaufen, Transfer zwischen Speicher/CPU 1.4 mal so schnell
- Zeigen Sie, welcher Ansatz besser ist, indem Sie den jeweiligen Speedup berechnen und dann vergleichen!

Amdahl: Beispiel 2 Lösung

Variante 1, CPU-Verbesserung

$$S = \frac{1}{(1 - 0.3) + \frac{0.3}{1.5}} = 1.11 \dots$$

Variante 2, Speicherverbesserung

$$S = \frac{1}{(1 - 0.7) + \frac{0.7}{1.4}} = 1.25 \dots$$

Lösung

Variante 2 ist besser!

(solange man Anschaffungskosten nicht miteinbezieht...)

CPU-Performance

Taktung

- **Alle Einheiten und Operationen in einer CPU müssen zeitlich synchronisiert werden**
- Taktgeber mit konstanter Frequenz erzeugt diskrete Zeitschritte
- Begriffe: *ticks*, *cycles*, *clock cycles*, ...
- **Zeitdauer eines Takts** t_{cc} , [s]
- **Taktfrequenz** (Taktrate) f , [Hz] = Anzahl an Cycles pro Sekunde (entscheidend für die in einer Sekunde ausführbaren Instruktionen/Operationen)
- Jede Instruktion benötigt eine gewisse Anzahl an Cycles
- **Ein Programm ist eine Abfolge von Instruktionen**

CPU-Zeit

CPU-Zeit t_{CPU}

- $t_{CPU} = (\text{CPU clock cycles fuer ein Programm}) \cdot t_{cc}$
- $t_{CPU} = \frac{(\text{CPU clock cycles fuer ein Programm})}{f}$
- $t_{cc} \dots$ Zeitdauer eines Takts [s]

Clock Cycles Per Instruction (CPI)

- Programm = Anzahl von Instruktionen \Rightarrow Instruction Count (IC)
- Mittlere Anzahl an **Zyklen pro Instruktion** oft wichtig
- Auch invers (Instructions per Cycle, IPC) verwendet

$$CPI = \frac{(\text{CPU clock cycles pro Programm})}{IC}$$

CPU-Zeit per CPI

CPU-Zeit

- $t_{CPU} = IC \cdot t_{cc} \cdot CPI$
- $t_{CPU} = \frac{IC \cdot CPI}{f}$

Überprüfung

$$\frac{\text{Instruktionen}}{\text{Programm}} \cdot \frac{\text{ClockCycles}}{\text{Instruktion}} \cdot \frac{\text{Sekunden}}{\text{ClockCycle}} = \frac{\text{Sekunden}}{\text{Programm}} = ET_{cpu}$$

Wichtige Kriterien für CPU-Performance

- Clock cycle time bzw. invers: Taktrate
- Cycles per Instruction bzw. invers: Instructions per Cycle
- Instruction count, also “Größe” des Programms

CPU-Zeit bei verschiedenen Instruktionen

CPU-Zeit bei verschiedenen Instruktionen

- **Programme bestehen fast immer aus mehreren, verschiedenen Arten von Instruktionen**
- **Verschiedene Instruktionen benötigen oft unterschiedliche Mengen an Zyklen**
- IC_i : Anzahl der Instruktionen vom Typ i in einem Programm
- CPI_i : Mittlere Anzahl der Zyklen/Instruktion vom Typ i

CPU-Zeit

$$\text{CpuZeit} = \left(\sum_{i=1}^n IC_i \cdot CPI_i \right) \cdot t_{cc}$$

Überblick

- 1 Einführung und Formale Aspekte
 - Klassifikation
 - Design und Performance
 - Quantitative Methoden
- 2 Standard-Rechnerarchitektur**
 - Abstraktionsebenen**
 - Prozessor**
- 3 Speicherverwaltung
 - Virtueller Speicher und Paging
 - Caching
- 4 Externer Speicher
 - Festplatten
 - Disk Scheduling
 - SSD
- 5 Skalare und superskalare/parallele Architekturen
 - CISC und RISC
 - Pipelining
 - Superskalare Architekturen

Abstraktionsebenen

Was ist überhaupt ein Rechner?

Antwort hängt stark von der Sichtweise ab:

- **Anwendungs-Ebene**
- **Höhere Programmiersprache-Ebene**
- **Assembler-Ebene**
- **Betriebssystem-Ebene**
- **Maschinensprache-Ebene**
- **Mikroprogramm-Ebene**
- **Hardware-Ebene**

Abstraktionsebenen

Hardware-Ebene:

- Physikalische, elektrotechnische Ebene
- Wie ist Prozessor (i.e., Steuer- und Recheneinheiten) hardwaremäßig aufgebaut (Transistoren, Gatter, etc.)?

Mikroprogramm-Ebene:

- Wie werden einzelne Bausteine der Steuer- und Recheneinheit von Maschinenbefehlen angesteuert?
- Fetch–Decode–Execute
- Nur bei CISC-Architektur, bei RISC-Architektur kann diese Ebene entfallen

Abstraktionsebenen

Maschinensprache-Ebene:

- Unterste für Programmierer frei zugängliche Ebene
- *Software/Hardware-Schnittstelle* eines Prozessors
- Satz der verfügbaren Maschinenbefehle eines Prozessors bestimmt die “Architektur” des Rechners (CISC vs RISC)

Betriebssystem-Ebene:

- Verwaltung von Speicher
- Ansteuerung der Ein-/Ausgabe-Geräte
- Ausführung von Programmen
- Kommunikation zwischen Mensch und Computer

Abstraktionsebenen

Assembler-Ebene:

- Stellt symbolische Notation (Mnemonics) der auf Maschinensprache- und Betriebssystem-Ebene vorhandenen Befehle zur Verfügung
- Pseudoinstruktionen, die sich einfach durch wenige Maschinenbefehle realisieren lassen

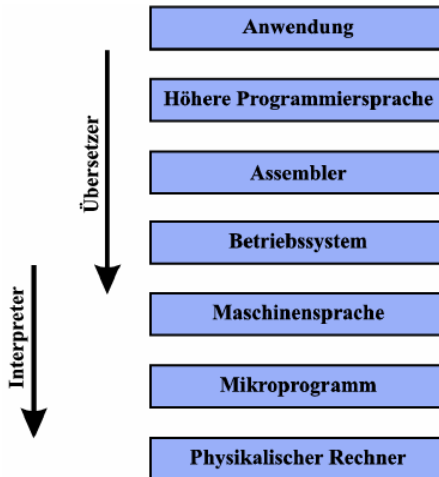
Höhere Programmiersprache-Ebene (Fortran, C, Java,...):

- Rechner ist eine Black Box, tatsächlicher technischer Ablauf bzw. Hardware-Struktur bleiben verborgen

Anwendungs-Ebene:

- Nur Anwendung sichtbar

Abstraktionsebenen



Compiler vs. Interpreter

Programmausführung:

Jede Ebene muss auf Hardware-Ebene abgebildet werden

Compiler

L_i -**Compiler**: Ein Maschinenprogramm, das ein Programm P_i der Sprache L_i in ein Programm P_j der Sprache L_j transformiert (**übersetzt**), das äquivalent zu P_i ist, d.h. das gleiche Ein-/Ausgabeverhalten wie P_i hat.

Interpreter

L_i -**Interpreter**: Ein Maschinenprogramm, das ein Programm P_i der Sprache L_i **Anweisung für Anweisung ausführt**.
Kein zu P_i äquivalentes Maschinenprogramm wird erzeugt.

Compiler vs. Interpreter

Anschaulich

Wir wollen ein Rezept nachkochen, das auf französisch verfasst ist, wir verstehen diese Sprache allerdings nicht.

Variante 1

Mit Rezept und Wörterbuch in den Supermarkt, jede Zutat einzeln nachschauen. Zuhause jeden Arbeitsschritt einzeln nachschauen. ⇒ Interpretieren

Variante 2

Das Rezept vorab von jemandem, der der Sprache mächtig ist, in eine Sprache, der wir mächtig sind, übersetzen lassen, sodann die Einkaufsliste und Befehlsliste abarbeiten. pause ⇒ Übersetzen (Kompilieren)

Maschinenbefehlssatz

Maschinenbefehlssatz

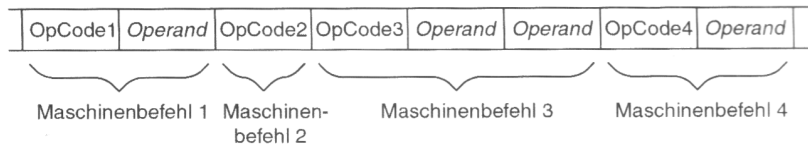
- Compiler übersetzt Programm von Hochsprache in Maschinenbefehle
- **Maschinenbefehlssatz:** Direktes Abbild aller vom Prozessor durchführbaren Operationen → Menge der Maschinenbefehle
- Transportbefehle: Speicher-Speicher, Register-Speicher, PUSH/POP, ...
- Arithmetische Befehle: Addition, Inkrement, Dekrement, ...
- Bitweise logische Befehle: AND, OR, XOR, NOT, ...
- Schiebe- und Rotierbefehle
- Sprungbefehle ...

Maschinencode / Maschinenprogramm

Maschinencode / Maschinenprogramm

- **Programm in Maschinenbefehlsdarstellung**
- **Sequenz von OpCodes und Operanden**
- Schwer merkbare Bitmuster
- Platzsparender: Hex-Darstellung
- Programme schwer zu lesen, unflexibel, schwer veränderlich; keine Kommentare möglich
- **Assemblersprache:** Jeder Maschinenbefehl durch leicht merkbare Abkürzung dargestellt, z.B. ADD, SHL, MOV, DEC, ...

Maschinencode / Maschinenprogramm



Maschinencode / Maschinenprogramm

Maschinencode / Maschinenprogramm

- **Mikroprozessor kann exakt bestimmte Menge von Aktionen ausführen**
- **Aktion = Maschinenbefehl**
- Bsp: Schreiben in Hauptspeicher, Bitmuster invertieren, ...
- Maschinenbefehl im ausführbaren Code **durch Bitmuster dargestellt**: Operationscode, **Opcode**
- Meist zusätzliche **Operanden** notwendig, z.B. Adresse
- **Maschinencode** = Sequenz von zusammenhängenden Maschinenbefehlen, die ablaufendes Programm darstellen
- **Programm = Abfolge von Instruktionen**

Maschinencode / Maschinenprogramm

Ein Maschinenprogramm kann nicht mehr übersetzt werden

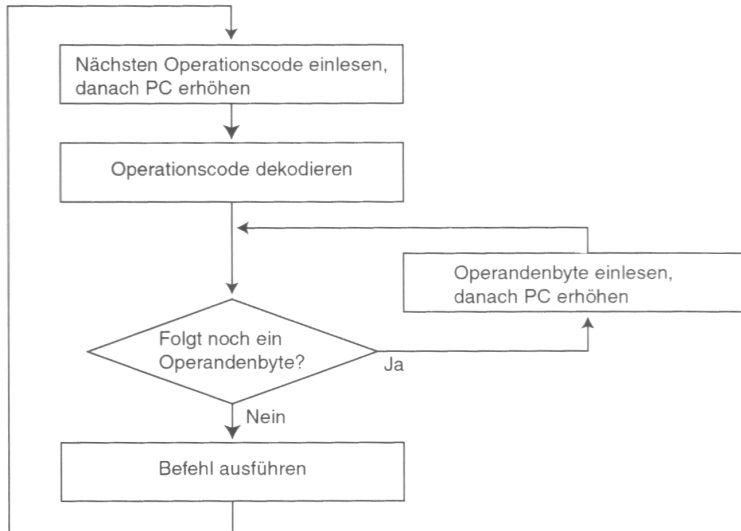
- Befehl für Befehl des Maschinenprogramms muss durch ein Mikroprogramm (Ebene 2) bzw. die Hardware (Ebene 1) abgearbeitet werden, d.h.
 - aus dem Hauptspeicher in den Prozessor **geholt**,
 - **dekodiert**, und letztendlich
 - **ausgeführt** werden,
... bevor der nächste Maschinenbefehl durch den Prozessor betrachtet und verarbeitet werden kann

Fetch – Decode – Execute

Programmausführung

- **Fetch:** Nächster auszuführender Opcode wird aus Programmspeicher gelesen
 - Program (auch Instruction) Counter (PC): Spezialregister, das die Adresse des nächsten Befehls enthält
 - PC wird inkrementiert, zeigt damit auf nächsten Befehl
- **Decode:** OpCode wird bitweise mit bekannten Mustern (Befehlssatz) verglichen, um Bedeutung herauszufinden
 - Falls OpCode mit Operand, wird PC inkrementiert, um Operanden auf nachfolgendem Speicherplatz zu lesen (bis alle Operanden gelesen)
- **Execute:** OpCode wird ausgeführt;

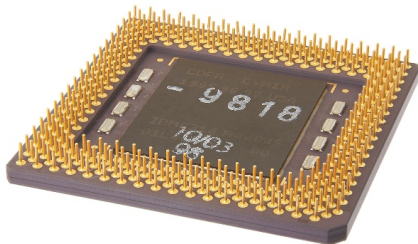
Fetch – Decode – Execute



Der Prozessor

...auch bekannt als

- Central Processing Unit, Central Processor Unit (CPU)
- Hauptprozessor
- Mikroprozessor

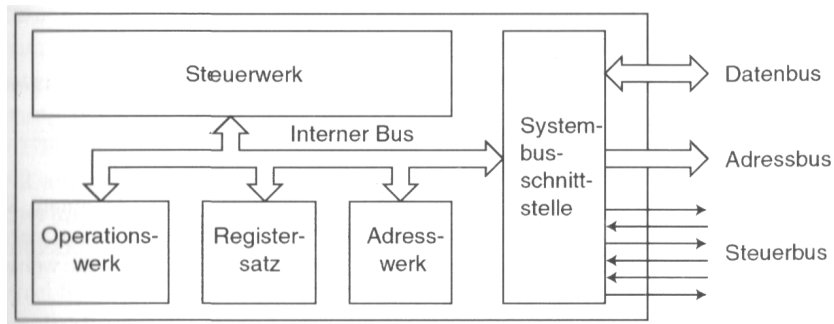


Interner Aufbau

Interner Aufbau einer CPU

- **Registersatz:** Register, um Daten innerhalb des Prozessors speichern zu können
- **Steuerwerk:** Verantwortlich für Ablaufsteuerung
- **Operationswerk** (Rechenwerk): Eigentliche Datenverarbeitung
- **Adresswerk:** Um auf Daten und Code im Hauptspeicher zugreifen zu können
- **Systembus**-Schnittstelle: Datenverkehr mit Rest des Systems

Schematisch



Registersatz

Register

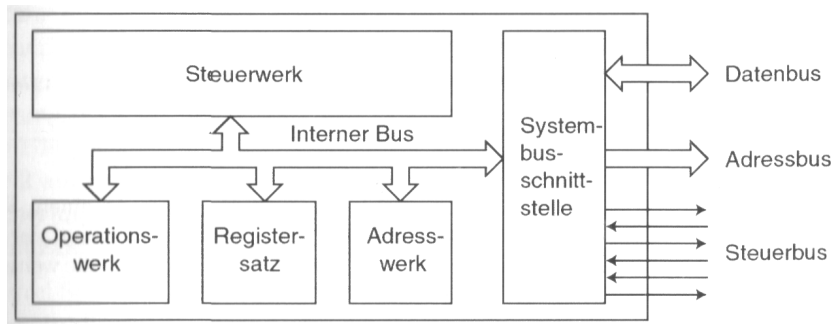
- Ein Speicherbereich innerhalb eines Prozessors
- Durch internen Datenbus meist direkt mit ALU verbunden
- Wesentlich schneller (und kleiner) als Hauptspeicher
- Gruppe von Flip-Flops mit gemeinsamer Steuerung
- 1 Flip-Flop kann 1 Bit speichern
- Register haben gewisse Breite, meist 8, 16, 32, 64... Bit
- **Registersatz:** Nach außen „sichtbare“ Register

Registersatz

Registertypen

- Datenregister (Akkumulator): Speichern Operanden für die ALU und deren Resultate
- Adressregister: Speichern Speicheradressen eines Operanden oder Befehls
- Universalregister: Für verschiedene Inhalte verwendbar
- Spezialregister: Für bestimmte Zwecke vorgesehen
 - Befehlszählregister: Speicheradresse des nächsten auszuführenden Befehls
 - Befehlsregister (Instruction register): (Zwischen)-Speicherung des aktuellen Befehls
 - Statusregister: z.B. Auftreten eines Überlaufs

Schematisch



Steuerwerk

Steuerwerk – Control Unit (CU)

- Steuert den Ablauf der Befehlsverarbeitung
- **Decodierung** der OpCodes (Instruction Decoder)
- Ansteuerung und Koordination der anderen Elemente
- Opcode wird im Befehlsregister abgelegt und dekodiert
- Mittels Schaltwerk werden notwendige Signale für Ausführung erzeugt

Steuerwerk: Beispiel (siehe K. Wüst, Mikroprozessortechnik, 4. Auflage)

Programmzähler zeigt auf Speicherplatz, an dem folgender
OpCode liegt: `Kopiere Register 1 in Register 2`

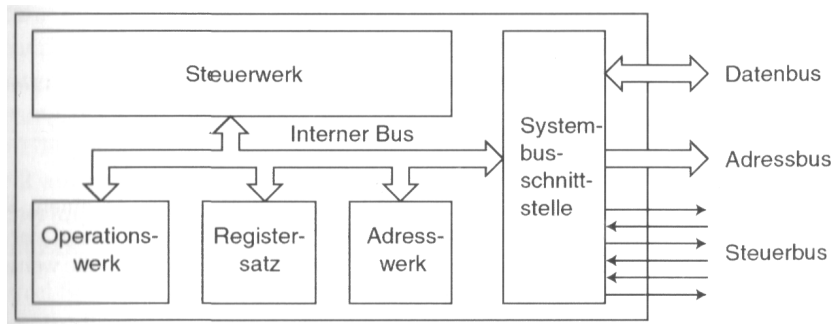
- Programmzähler auf Adressbus legen
- Aktivierung ext. Steuerleitungen f. Lesezugriff im Speicher
- Einspeicherimpuls für Befehlsregister erzeugen, OpCode vom Datenbus entnehmen und im Befehlsregister einspeichern
- Dekodierung des Opcodes
- Register 1 auf Senden einstellen und auf internen Datenbus aufschalten
- Register 2 auf internen Datenbus aufschalten
- Einspeicherimpuls an Register 2 geben
- Programmzähler inkrementieren

Steuerwerk: Beispiel

Zu beachten

- Wartezeiten + Buszugriffe in Ausführungsphase
- Komplizierte Sequenzen von Signale zu erzeugen!
- Exakter zeitlicher Ablauf \Rightarrow Synchronisierung!
- Statusflags von Prozessor zu berücksichtigen
- Externe Signale, z.B. Unterbrechungen (Interrupts)
- ...

Schematisch



Operationswerk

Operations-/Rechenwerk, Arithmetic & Logic Unit (ALU)

- Führt die vom Steuerwerk verlangten **logischen und arithmetischen** Operationen aus
- Wird vom Steuerwerk nach Dekodierung einer entsprechenden Instruktion angesprochen
- Verfügbare Operationen unterschiedlich, z.B.:
 - $a \wedge b$
 - $\neg b$
 - $a + b$
 - $a + 1$
 - $a - b$
 - bit shift left/right
 - ...
- Zum Ausprobieren: <http://tams-www.informatik.uni-hamburg.de/research/software/applets/hades/webdemos/20-arithmetic/50-74181/SN74181.html>

Operationswerk

Operations-/Rechenwerk, Arithmetic & Logic Unit (ALU)

- Über Steuereingänge auf bestimmte Anzahl von arithmetisch/logischen Operationen einstellbar
- Hängt vom Befehl ab, wird vom Steuerwerk nach dem OpCode-Decode gemacht
- Schaltwerk ohne eigene Speicherzellen:
Operandenregister (Hilfsregister) auf Dateneingänge für Zeit der Berechnung aufgeschaltet
- Ergebnis wird über Datenbus z.B. in Universalregister geschrieben
- Kann aber auch erneut in Operandenregister geladen werden ⇒ komplexe Operationen in mehreren Schritten ausführen

Operationswerk

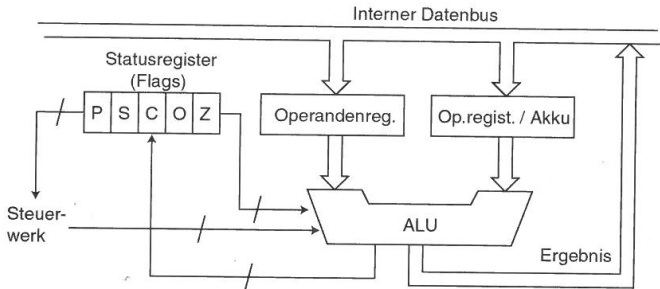
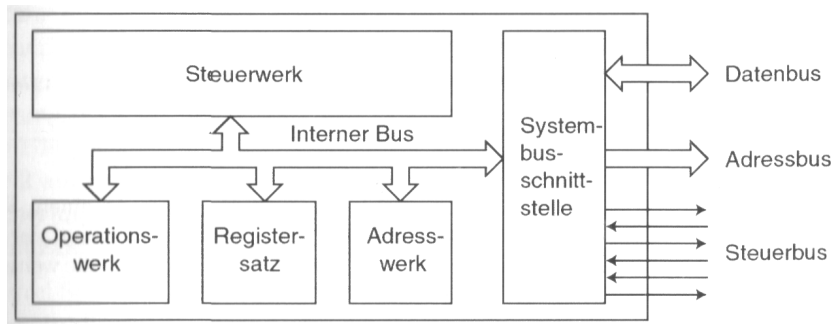


Abbildung 7.8: Zentraler Teil des Operationswerkes.

Schematisch



Adresswerk

Aufgaben

- Berechnet nach den Vorschriften des Steuerwerks die Adresse eines Befehls oder eines Operanden

Adressierungsarten

- Unmittelbare Adressierung: Operand als Konstante im Befehl enthalten (folgt unmittelbar auf den OpCode)
- Registeradressierung: Register **direkt** angesprochen

Adresswerk

Adressierungsarten

- **Direkte** Adressierung: Speicheradresse beim Kompilieren bekannt und als Operand im Maschinencode
- Z.B. Sortialgorithmen: Adressen müssen zur Laufzeit festlegbar sein
- Register-**indirekte** Adressierung: **Inhalt eines Registers** als Adresse interpretiert
- **Basisregister**: Volle Adressbusbreite, für Anfangsadresse
- **Displacement** (aus Maschinencode) kann addiert werden
- Adresswerk hat eigenen Adressrechner

Adresswerk

Adressierungsarten

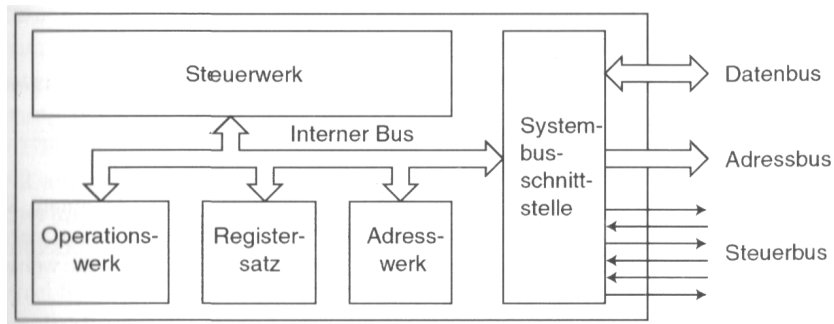
- **Indexregister:** Kann z.B. Autoinkrement, Autodekrement
- Für **indizierte Adressierung** ideal, um zusammenhängende Datenblöcke sequenziell zu adressieren
- **Speicherindirekte Adressierung:** Register verweist auf Speicherplatz, diese wird gelesen und als Adresse verwendet
- Bis zu zwei Displacements möglich

Adresswerk

Stackpointer

- **Spezialregister, SP, Stapelzeiger**
- Verwendung eines Stacks in Hardware unterstützt
- Speicherstruktur mit **Last-In–First-Out** Prinzip (LIFO)
- Anschaulich: ein Stapel Teller oder Bücher, ...
- Temporäre Daten, z.B. Programmverlauf
- Wächst meist zu kleineren Speicheradressen hin
- Neues Wort im Stack ablegen: Stackpointer dekrementieren (**PUSH**)
- Wort von Stack holen: Stackpointer inkrementieren (**POP**)

Schematisch

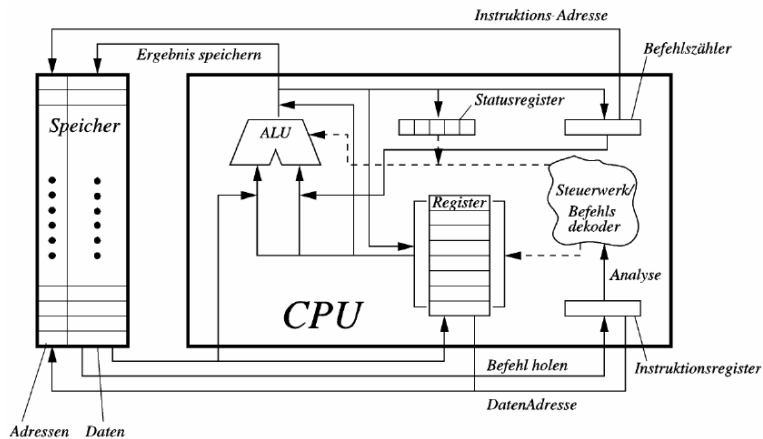


Systembusschnittstelle

Systembus (!= interner CPU Bus)

- **Adressleitungen** (Adressbus): unidirektional, Prozessor immer Sender
 - Adressbus-Breite gibt adressierbaren Bereich an
 - 32 bit $\Rightarrow 2^{32} \cdot 1 \text{ Byte} = 4096 \text{ MiB} = 4 \text{ GiB}$ (386 bis Pentium)
 - 48 bit $\Rightarrow 2^{48} \cdot 1 \text{ Byte} = 256 \text{ TiB}$ (AMD64)
...wenn an jeder Adresse 1 Byte gespeichert wird.
- **Datenleitungen** (Datenbus): bidirektional
 - Datenbus-Breite: Wie viele Daten können in einem Schritt transferiert werden
- **Steuerleitungen** (Steuerbus): Jede Leitung hat eine andere Aufgabe

The Big Picture



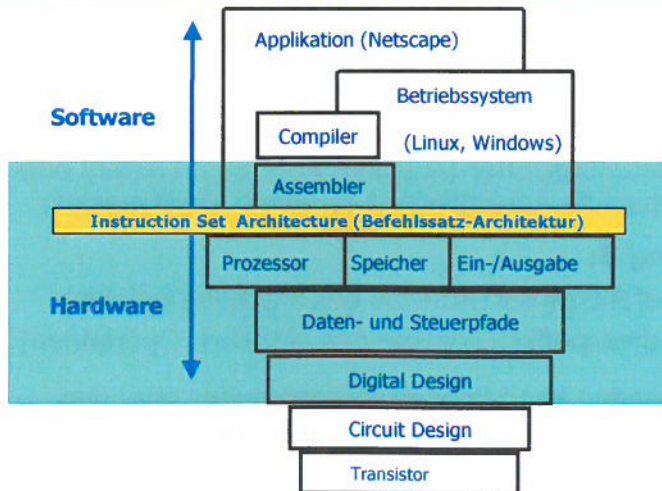
Sehenswert: http://www.youtube.com/watch?v=cNN_tTXABUA

ISA

ISA - Instruction Set Architecture / Befehlssatz

- **Gesamte nach außen hin sichtbare Architektur**
 - 1 Befehlssatz (s.o.)
 - 2 Registersatz (s.o.)
 - 3 Speichermodell (Größe des Adressraums, Breite der Busse, etc.)
- Muss zur Erstellung von Maschinenprogrammen für diesen Prozessor bekannt sein
- Für Assemblerprogrammierung, Aufbau von Compilern, ...
- **ISA = Schnittstelle zwischen Hard- und Software**

ISA



Überblick

- 1 Einführung und Formale Aspekte
 - Klassifikation
 - Design und Performance
 - Quantitative Methoden
- 2 Standard-Rechnerarchitektur
 - Abstraktionsebenen
 - Prozessor
- 3 Speicherverwaltung**
 - Virtueller Speicher und Paging**
 - Caching**
- 4 Externer Speicher
 - Festplatten
 - Disk Scheduling
 - SSD
- 5 Skalare und superskalare/parallele Architekturen
 - CISC und RISC
 - Pipelining
 - Superskalare Architekturen

Virtueller Speicher

Mangel an Arbeitsspeicher – Mögliche Lösungen

- Früher: *Overlays*, Programmteile manuell ausgelagert
- Probleme: Manuell, ist für jede Maschine anders
- **Virtueller (logischer) Adressraum: unabhängig vom physikalisch vorhandenen Arbeitsspeicher**
- Betriebssystem lagert bei Speichermangel Bereiche (**Pages**) auf Massenspeicher aus (**Paging, Swapping**)
- Paging ist **transparent** für Programmierer (muss keine Rücksicht nehmen, ob Adressen im *real* vorhandenen *physikalischen Arbeitsspeicher* wirklich existieren)
- **Virtuelle Adresse** ⇒ **physikalische Adresse**
- Hardware-Baustein: Memory Management Unit auf CPU

Virtueller Speicher

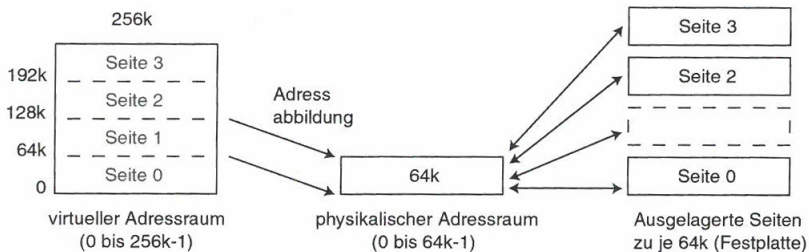
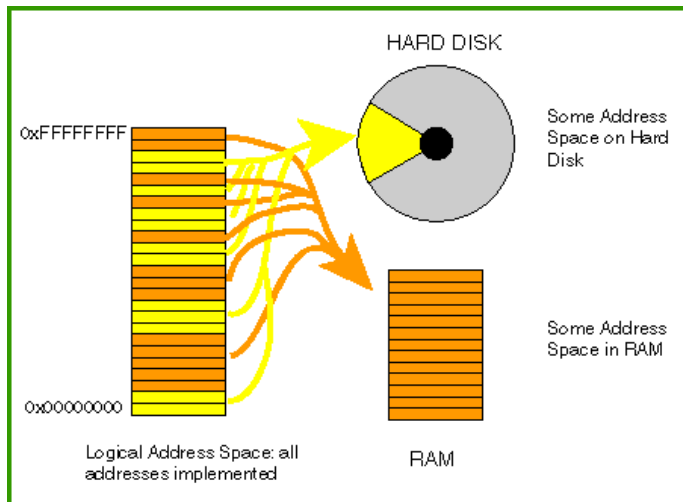


Abbildung 10.1: Die Seitenauslagerung ermöglicht einen beliebig großen virtuellen Adressraum. In diesem Beispiel befindet sich gerade Seite 1 im physikalischen Speicher, die anderen Seiten sind ausgelagert.

Virtueller Speicher



Virtueller Speicher – Beispiel

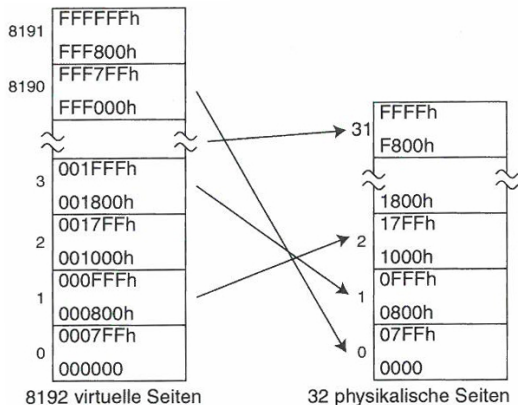


Abbildung 10.2: Beispiel zum Paging: Ein virtueller Adressraum von 16 MByte, wird mit einer Seitengröße von 2 KByte auf einem physikalischen Adressraum von 64 KByte abgebildet.

Paging

Page size: 512 Byte bis 4 MiB, typisch sind 4 KiB

⇒ zu groß: Ein-/Auslagerung dauert zu lange

⇒ zu klein: HDD Seek Time problematisch

- **Page table:** Verwaltung der Pages
- **Page fault:** Page wird gebraucht, wurde aber ausgelagert
- **Demand paging:** Physikalischer Speicher wird sukzessive mit benötigten Speicherbereichen befüllt (per page faults). Liefert nach einiger Zeit **working set** (Arbeitsmenge) des Programms.
- **Dirty bit:** Wurde eine Seite überhaupt geändert, oder muss sie bei Auslagerung gar nicht auf die Disk zurückgeschrieben werden?

Paging

Was passiert wenn physikalischer Speicher voll ist?

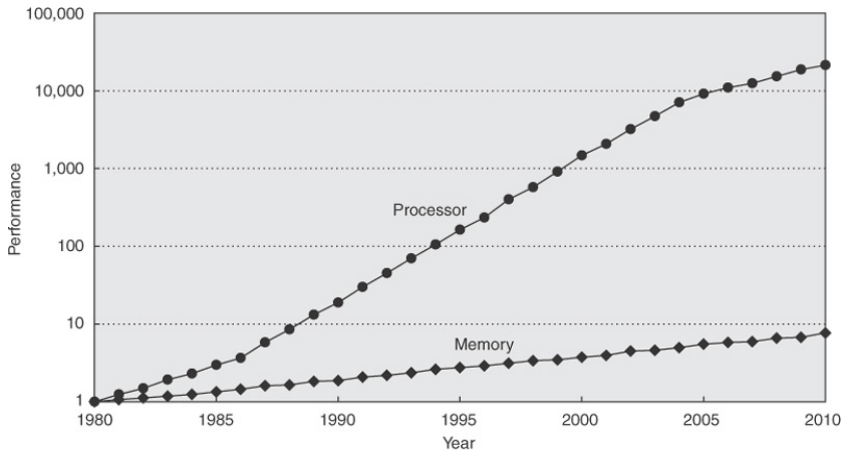
- Wenn physikalischer Speicher voll ist, muss vor dem Einlagern einer angeforderten Page (Page-In) eine andere Page ausgelagert (Page-out) werden. **Welche?**
- Diejenige, die möglichst lange nicht mehr gebraucht wird
- Lässt sich aber nur per Wahrscheinlichkeit ausdrücken:
 - **LRU**-Ersetzung: die am längsten nicht gebraucht wurde
 - **LFU**-Ersetzung: die am seltensten gebraucht wurde
 - **FIFO**-Ersetzung: die am längsten im Speicher ist
- **Thrashing**: System ist (fast) nur mit Einlagerung und Auslagerung beschäftigt und kann kaum (gar nicht) Programm abarbeiten

Warum Caches?

Motivation

- CPUs in letzten 30 Jahren um Größenordnungen schneller
 - (Haupt-)Speicher auch schneller, aber nicht in diesem Maße ⇒ Speicherzugriff dauert oft mehr als 10 CPU-Takte
 - **CPU-Designziel: in jedem Takt Befehl abarbeiten**
 - Bei superskalaren CPUs sogar noch mehr Befehle/Takt
 - Daten und Code müssen schnell genug aus Speicher geliefert werden, sonst sind alle CPU-Fortschritte sinnlos!
- ⇒ **Speicher-Latenzzeit:** Wartezeit, bis Speicherzugriff anläuft
- ⇒ **Speicherbandbreite:** Übertragungsrate
... müssen beide verbessert werden
- ⇒ **Caches: kleine, schnelle Zwischenspeicher**

CPU Memory Gap



© 2007 Elsevier, Inc. All rights reserved.

Lokalitätsprinzip

Auf (Haupt-)Speicher wird meist nicht völlig zufällig zugegriffen

- 1 **Räumliche Lokalität:** Häufig Zugriffe auf Adressen, die in der Nähe kürzlich benutzter Adressen liegen
- 2 **Zeitliche Lokalität:** Folgezugriffe auf kürzliche benutzte Adressen

⇒ **Kürzlich benutzte Daten möglichst lange im Cache halten**

- Nach Hauptspeicherzugriff wird nicht nur Inhalt der adressierten Speicherzelle im Cache aufbewahrt (zeitliche Lokalität), sondern gleich ganzer Speicherblock (räumliche Lokalität), in dem die Speicherzelle liegt

Cache Hits, Cache Misses

“Datum”: Eine kleine Datenmenge im Cache

- Cache zunächst leer, füllt sich bei Zugriffen: Kopie des Speicherblocks wird im Cache abgelegt
- Lesezugriff auf Hauptspeicher: **Cache auf Kopie prüfen**
- **Cache-Hit: Datum vorhanden, lesen aus Cache**
- **Cache-Miss: Datum nicht vorhanden, lesen aus Hauptspeicher – mehrfacher Aufwand**
- **Hit-Rate: Wahrscheinlichkeit für Cache Hit**

Look-Through-Cache vs. Look-aside-Cache

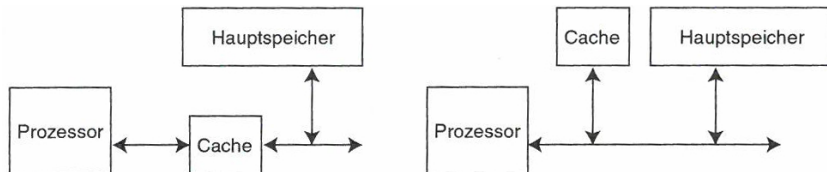


Abbildung 10.8: Ein Cache kann zwischen Hauptspeicher und Prozessor liegen (Look-Through-Cache, links) oder parallel zum Hauptspeicher (Look-aside-Cache, rechts).

K. Wüst, Mikroprozessortechnik, 4. Auflage

Look-Through-Cache vs. Look-Aside-Cache

Look-Through-Cache

- Zwischen CPU und Hauptspeicher
 - Schnell an CPU angebunden (höherer Bustakt als Hauptspeicher)
 - Cache und Hauptspeicher nicht gleichzeitig aktivierbar (Hauptspeicherzugriff erfolgt erst, wenn Daten im Cache nicht gefunden wurden)
- ⇒ Bei guter Hit-Rate fast nur Zugriffszeit des Caches entscheidend
- ⇒ Bei schlechter Hit-Rate hohe Wartezeiten

Look-Through-Cache vs. Look-Aside-Cache

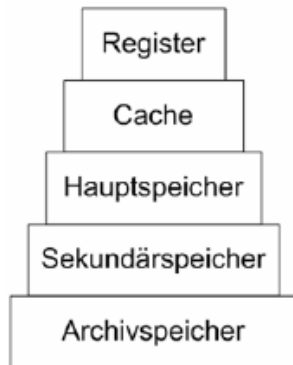
Look-Aside-Cache

- Parallel zum Hauptspeicher
 - Cache-Zugriff und Hauptspeicher-Zugriff werden gleichzeitig aktiviert
- ⇒ Kein Zeitverlust bei Fehltreffer im Cache
- ⇒ Aber: Cache wird durch langsamen Speicherbus getaktet

Speicherhierarchie

“Ideally one would desire an indefinitely large memory capacity such that any particular word would be immediately available. We are forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding one but which is less quickly accessible.”

– *A.W.Burks, H.H. Goldstine and J. von Neumann (1946)*



Mehrstufige Caches

Einstufige Caches oft ineffizient \Rightarrow mehrstufige Caches

1 First-Level (L1) Cache:

- Kleiner Cache, mit CPU-Takt betrieben; ca. 16-64 KiB
- Meist in Prozessorchip integriert
- *Split cache*: Daten / Code; gleichzeitiger Zugriff möglich

2 L2-Cache

- Hinter L1-Cache; $f < \text{CPU-Takt}$; ca. 256-512 KiB
- Meist in Prozessorchip integriert
- *Unified cache*: Daten und Code gemeinsam

3 L3-Cache

- Hinter L2-Cache; oft schnelles SRAM auf Systemplatine
- Bei MultiCore-CPU's meist geteilt zwischen einzelnen Cores (L1 und L2 extra für jeden Core)

4 L4-Cache Prinzip: Hauptspeicher als Cache für HDD ...

Memory Stall Cycles, Miss Penalty

MSC, MP

- Memory Stall Cycles (MSC): Anzahl der Zyklen, die CPU auf Speicherzugriff warten muss
- Hängt von der Anzahl der Cache Misses und den Kosten pro Cache Miss (Miss Penalty) ab
- $MSC = \text{Anzahl der Misses} \cdot MP$
- $= IC \cdot MPI \cdot MP = IC \cdot \frac{\text{Misses}}{\text{Instruktion}} \cdot MP$
- $= IC \cdot \frac{\text{Speicherzugriff}}{\text{Instruktion}} \cdot \frac{\text{Misses}}{\text{Speicherzugriff}} \cdot MP$
- $MPI = \frac{\text{Misses}}{\text{Instruktion}} = \frac{\text{Speicherzugriffe}}{\text{Instruktion}} \cdot MR$
- $MR = \frac{\text{Misses}}{\text{Speicherzugriff}}$ (Miss Rate)

Performance

Mittlere Speicherzugriffszeit

- $\bar{t}_{SZ} = t_H + MR \cdot MP$
- $t_H \dots$ **Hit time**, Zeit um **Treffer** im Cache zu erzielen
- $MR \dots$ Miss rate, Wahrscheinlichkeit für Cache Miss
- $MP \dots$ Extra-Aufwand bei Cache Miss (zB L2, RAM, ..)

Für mehrstufige Caches

- $\bar{t}_{SZ} = t_{H,L1} + MR_{L1} \cdot MP_{L1}$
- $MP_{L1} = t_{H,L2} + MR_{L2} \cdot MP_{L2}$
- $\bar{t}_{SZ} = t_{H,L1} + MR_{L1} \cdot (t_{H,L2} + MR_{L2} \cdot MP_{L2})$

Performance

Average Memory Stalls per Intruction (AMSPI)

- Durchschnittliche Anzahl an Zyklen pro Instruktion, in denen auf Daten aus dem Speicher gewartet wird
- Annahme: L1-Cache wird mit vollem Prozessortakt betrieben \Rightarrow erfolgreicher L1-Cache Zugriff braucht also keinen Extra-Takt (Zyklus)
- $AMSPI = MPI_{L1} \cdot t_{H,L2} + MPI_{L2} \cdot MP_{L2}$

Cache-Optimierungen

Grob: 3 Varianten

1 **Miss Penalty (MP) senken**

- Mehrstufige Caches

2 **Miss Rate (MR) senken**

- Blockgröße erhöhen, nutzt räumliche Lokalität, erhöht MP

3 **Hit Time (HT) senken**

- Kleiner, einfacher, schneller Cache; On-chip, CPU-Takt, direct mapped

Cache-Organisation

Bestandteile eines Caches

1 Datenbereich

- **Besteht aus Anzahl von Cache-Zeilen** (*Cache Lines*)
- Alle Lines haben **einheitliche Blocklänge** (*block/line size*)
- Immer ganze Cache Line laden/ersetzen

2 Identifikationsbereich

- **Enthält tags** (Etikett): Aus welchem Abschnitt des Hauptspeichers wurde Cache-Zeile geladen
- Stellt fest ob ein Datum im Cache ist oder nicht

3 Verwaltungsbereich: Mindestens zwei Bits

- **Valid** bit: Ist Cache-Line ungültig (veraltet, 0) oder gültig (1)
- **Dirty** bit: Wurde auf Cache-Line geschrieben (1) oder nicht (0) – für Copy-Back-Ersetzungsstrategie wichtig

Cache-Organisation

Mögliche Organisationsformen

- 1 **Vollassoziativ** (fully associative)
- 2 **Direkt abbildend** (direct mapped)
- 3 **Mehrfach assoziativer Cache (n-way set associative, teilassoziativ)**, am häufigsten zu finden

Fully associative

Fully associative

- **Datenblock in beliebiger Cache-Zeile abspeicherbar**
- \Rightarrow **Alle** Tags mit angefragter Adresse zu vergleichen
- Muss aus Performance-Gründen gleichzeitig erfolgen
- \Rightarrow Vergleichereinheit für **jede** Cache-Zeile
- \Rightarrow **Sehr hoher Hardwareaufwand**
- **Dafür keine Einschränkungen bei Ersetzung!**
- \Rightarrow **Höchste Trefferquote aller Organisationen**

Direct mapped

Direct mapped

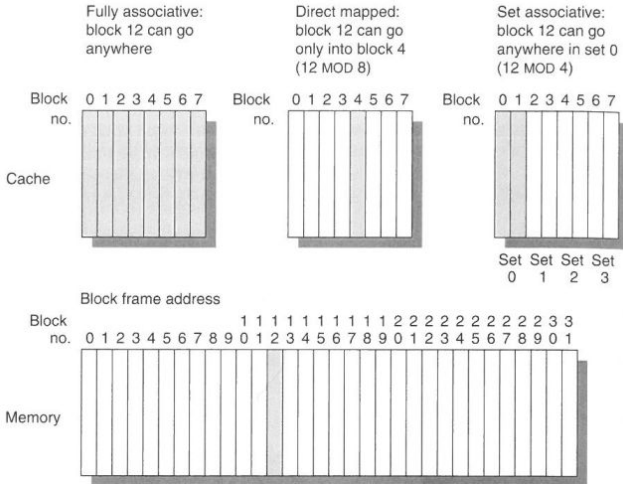
- **Teil der Speicheradresse wird als Index benutzt um dem Datenblock eine Cache-Zeile zuzuordnen.**
- **⇒ Block kann nur dort gespeichert werden**
- **⇒ Es gibt keine Ersetzungsstrategie**
- Kann zu Thrashing führen = schlechter als gar kein Cache
- Aus Index sofort klar, wo Block stehen muss (falls im Cache)
- Einfacher Aufbau, nur ein Vergleich notwendig
- **⇒ Schlechteste Trefferquote aller Organisationen**

n-way set associative

n-way set associative

- **Kompromiss zw. fully associativ und direct mapped**
- Cache-Zeilen werden zu einem *Satz* zusammengefasst
- **Satznummer wird als Index benutzt:**
 - Aus Adresse ergibt sich nur Nummer des Satzes
 - Innerhalb des Satzes kann frei gewählt werden
 - Ersetzungsstrategien möglich
- **Trefferbestimmung:**
 - Index verweist auf Satz, wo Datum stehen könnte
 - Innerhalb von Satz kann Datum in n Einträgen liegen
 - n Vergleiche: parallel Tags der n Einträge vergleichen

Cache Organisationen im Überblick



Cache Organisationen – Beispiel

AMD K10

- ① L1
 - 64 KB 2-way set associative instruction cache
 - 64 KB 2-way set associative data cache
- ② 512 KB 16-way set associative cache
- ③ 32-way set associative L3 shared cache

Vergleich per Benchmark

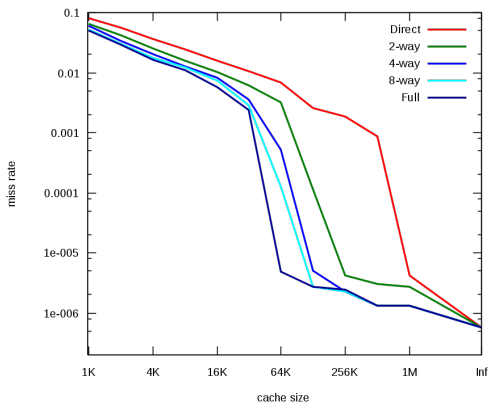


Abbildung: Miss rate versus cache size on Integer portion of SPEC CPU2000 –<http://en.wikipedia.org/wiki/File:Cache,misrate.png>

Ersetzungsstrategien

Ersetzungsstrategien

- Analog zu virtuellem Speicher
- LRU, LFU, FIFO, ...
- Random: Zufallsprinzip → erstaunlich gute Ergebnisse

Aktualisierungsstrategien

Wichtige Begriffe

- **Konsistenz:** **Alle** Kopien eines Datums inhaltsgleich
- **Kohärenz:** Korrektes Fortschreiten des Systemzustands
- Inkohärenz, wenn Dateninkonsistenz nicht bereinigt wird
- Lesen vs. **Schreiben**

Lesen

- Lese-Treffer: Daten zeitsparend aus Cache nehmen; keine weiteren Aktionen
- Lese-Fehlertreffer: Block, der Datum enthält, in Cache laden; angefordertes Datum in CPU laden

Aktualisierungsstrategien: Schreiben

Schreib-Treffer

Zwei Möglichkeiten:

- ① **Write-Trough-Strategie:** Datum wird in Cache und in Hauptspeicher geschrieben: Einfach, garantiert Konsistenz
- ② **Copy-Back (Write-Later)-Strategie:** Datum wird nur im Cache aktualisiert und betroffene Cache-Zeile als verändert markiert (Dirty Bit wird auf 1 gesetzt).
 - Spätestens vor Ersetzung muss Datum in Hauptspeicher zurückgeschrieben werden (write back, copy back)
 - Vorteilhaft z.B. bei viel benutzter Variable (z.B. Schleifencounter)
 - System jedoch teilweise inkonsistent

Aktualisierungsstrategien: Schreiben

Schreib-Fehlertreffer

Auch zwei Möglichkeiten:

- **No-Write-Allocate-Strategie:** Datum nur in Hauptspeicher schreiben, liegt danach nicht im Cache vor
- **Write-Allocate-Strategie:** Datum in Hauptspeicher schreiben und zusätzlich den entsprechenden Block in Cache laden

Überblick

- 1 Einführung und Formale Aspekte
 - Klassifikation
 - Design und Performance
 - Quantitative Methoden
- 2 Standard-Rechnerarchitektur
 - Abstraktionsebenen
 - Prozessor
- 3 Speicherverwaltung
 - Virtueller Speicher und Paging
 - Caching
- 4 Externer Speicher**
 - Festplatten**
 - Disk Scheduling**
 - SSD**
- 5 Skalare und superskalare/parallele Architekturen
 - CISC und RISC
 - Pipelining
 - Superskalare Architekturen

Sekundärspeicher

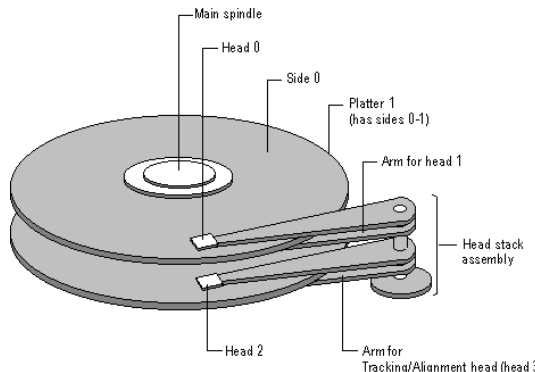
Sekundärspeicher

- **Speichermedien mit persistenter (dauerhafter) Speicherung**
- Optische Medien: CD, DVD, BluRay, ...
- **Magnetische Medien:** Festplatte (HDD), Floppy, ...
- Elektronische Medien: v.a. Flash-Speicher (USB-Stick, SSD, ...)
 - Keine mechanischen Verzögerungen
 - Dafür meist unterschiedliche Zeiten für Lesen/Schreiben
 - Vor dem Schreiben muss meist gelöscht werden

Festplatten

Aufbau

- Mehrere Scheiben (Platter) aus nicht magnetisierbarem Material (Substrate)
- Magnetisierbares Material wird aufgebracht
- Für jeden Platter: Schreib/Leseköpfe auf beweglichem Arm



Festplatten

Schreiben

- **Bewegte Ladungen (=Strom) erzeugen ein Magnetfeld**
- Definierter Bereich der Oberfläche wird magnetisiert
- Richtung des Stroms bestimmt Richtung der Magnetisierung (logisch 0/1)

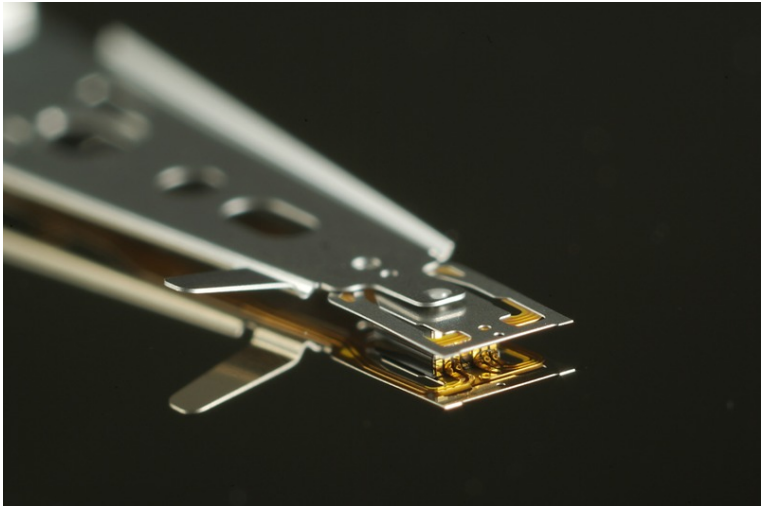
Lesen

- **Veränderliches Magnetfeld induziert Strom in einem Leiter**
- Magnetoresistiver (MR) Sensor, Widerstand hängt von Richtung der Magnetisierung ab. Per Lesestrom wird Widerstand (über Spannungsabfall) gemessen

└ Externer Speicher

└ Festplatten

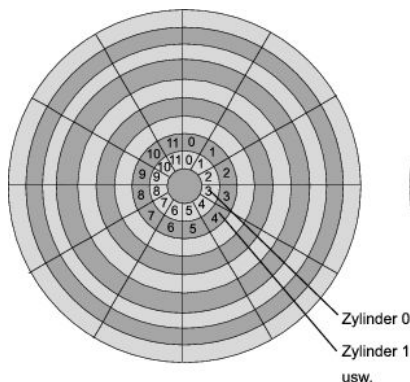
Disk head



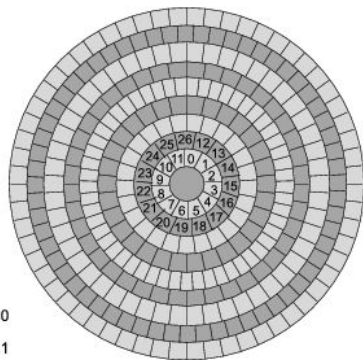
└ Externer Speicher

└ Festplatten

Aufbau – CHS (Cylinder, Head, Sector) vs. LBA (Logical Block Addressing)



CHS-Adressierung

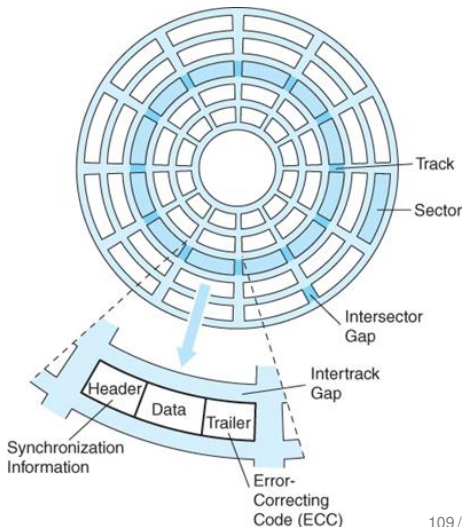


LBA-Adressierung

Datenorganisation bei CHS-Adressierung

Vereinfacht:

- Jeder Platter in viele Spuren (Tracks, konzentrische Kreise) aufgeteilt
- Jede Spur in viele Sektoren unterteilt, Einheit für Datentransfer
- Sektorgröße früher meist 512 Byte, heute auch oft 4 KiB



Performance bei CHS-Adressierung

Seek Time T_S

- **Aktuator (Lese/Schreibkopf) muss auf jeweilige Spur bewegt werden**
- Abschätzung $T_S = m \cdot n + s$
- $n \dots$ Anzahl der überquerten Spuren
- $m \dots$ Zeit pro Spur (Bruchteile einer ms)
- $s \dots$ Startup-Zeit (wenige ms)
- **Meist wird durchschnittliche Seek Time angegeben, ca. 8-15ms**

Performance bei CHS-Adressierung

Rotationsverzögerung T_r

- **Ist Aktuator auf richtiger Spur, dauert es, bis sich entsprechender Sektor unter dem Aktuator befindet**
- Abschätzung $T_r = \frac{1}{2r}$
- $r \dots$ Umdrehungen pro Minute (RPM)
- **Im Durchschnitt ist man eine halbe Rotation von Sektor entfernt**
- Bsp $r=10000$ RPM; $T_r = \frac{1}{2 \cdot 10000 \text{ min}} = 5 \cdot 10^{-5} \text{ min} = 5 \cdot 10^{-5} \cdot 60 \text{ s} = 3 \cdot 10^{-3} \text{ s} = 3 \text{ ms}$

Performance bei CHS-Adressierung

Transferzeit T

- **Reine Übertragungszeit der Daten, Aktuator ist schon an richtiger Position**
- Hängt v.a. von RPMs und Datendichte ab
- $T = \frac{b}{r \cdot N}$
- $b \dots$ Anzahl Bytes, die transferiert werden müssen
- $N \dots$ Anzahl an Bytes auf der Spur

Gesamte Transferzeit T_a

$$T_a = T_s + T_r + T = T_s + \frac{1}{2r} + \frac{b}{rN}$$

(De-)Fragmentierung

Fragmentierung

- Verstreute Speicherung logisch zusammengehörender Datenblöcke des Dateisystems auf einem Datenträger
- Führt speziell bei hohen Zugriffszeiten (seek time, HDD) zu spürbarer Verlangsamung der Lese- und Schreibvorgänge

Defragmentierung

- Neuordnung von fragmentierten Datenblöcken
- Logisch zusammengehörende Datenblöcke sollten aufeinander folgen
- Kann den sequentiellen Zugriff beschleunigen

Disk Scheduling

Disk Scheduling

- Leistungssteigerung nicht nur durch mechanische Verbesserungen oder Defragmentierung erzielbar
- Scheduling: Gezieltes Umreihen der Anfragen, mechanische Verzögerungen möglichst minimieren
- Strategien: FIFO/FCFS, SSTF, (C-)SCAN, (C)-LOOK

Scheduling: FIFO/FCFS

FIFO/FCFS

- First-In-First-Out; First-Come-First-Served
- Anfragen in Reihenfolge des Eintreffens bearbeiten
- Vorteil: Faire Abarbeitung
- Nachteil: Gesamtanzahl an überquerten Spuren sehr hoch

Scheduling: SSTF

SSTF

- Shortest Seek Time First
- Immer Anfrage bearbeiten, deren Spur der momentanen Aktuator-Position am nächsten ist
- Vorteil: Jeweilige Seek Time wird verringert
- Nachteil: Durchschnittliche Seek Time nicht garantiert minimal

Scheduling: (C-)SCAN

SCAN

- Aktuator läuft immer zwischen Endpositionen auf und ab
- Verhindert, dass Aufträge „verhungern“ (*Starvation*)

C-SCAN

- **C**: Circular
- Wie SCAN, beim Zurückkehren an Anfangsposition wird nichts abgearbeitet
- Reduziert Verzögerung für neu eintreffende Anfragen

Scheduling: (C-)LOOK

LOOK

- Ähnlich zu SCAN, aber intelligenter
- Sobald letzte angefragte Spur einer Richtung erreicht wurde, ändert Aktuator die Laufrichtung
- Restliche, unnötige Spuren müssen also nicht überquert werden
- C-LOOK: Analog

Beispiel

Beispiel

- **Disk hat insgesamt 200 Spuren**
- In der Warteschlange (Queue) sind mehrere Anfragen
- Angefragte Daten belegen jeweils maximal eine Spur
- Aktuator befindet sich gerade auf Spur 100, war davor auf Spur 85 \Rightarrow **aufsteigende** Laufrichtung bei LOOK/SCAN
- Folgende Spuren sind zu Lesen:
- 55, 58, 39, 18, 90, 160, 150, 38, 184

FIFO

Nächste Spur	Differenz
55	45
58	3
39	19
18	21
90	72
160	70
150	10
38	112
184	146
Gesamt	498
Schnitt	55,3

SSTF

Nächste Spur	Differenz
90	10
58	32
55	3
39	16
38	1
18	20
150	132
160	10
184	24
Gesamt	248
Schnitt	27,6

SCAN

Nächste Spur	Differenz
150	50
160	10
184	24
200	16
90	110
58	32
55	3
39	16
38	1
18	20
Gesamt	282
Schnitt	31,3

C-SCAN

Nächste Spur	Differenz
150	50
160	10
184	24
200	16
1	199
18	17
38	20
39	1
55	16
58	3
90	32
Gesamt	388
Schnitt	43,1

LOOK

Nächste Spur	Differenz
150	50
160	10
184	24
90	94
58	32
55	3
39	16
38	1
18	20
Gesamt	250
Schnitt	27,8

C-LOOK

Nächste Spur	Differenz
150	50
160	10
184	24
18	166
38	20
39	1
55	16
58	3
90	32
Gesamt	322
Schnitt	35,8

Bespiel: Vergleich

	FCFS	SSTF	SCAN	C-SCAN	LOOK	C-LOOK
Spuren	498	248	282	388	250	322
Schnitt	55,3	27,6	31,3	43,1	27,8	35,8

Nota bene

Es muss immer durch die Anzahl der zu bearbeitenden Spuren (hier 9) dividiert werden. Wenn ein Verfahren (hier SCAN und C-SCAN) **Wege** zurücklegt, **bei denen keine Arbeit erledigt wird, müssen die in die Summe eingerechnet werden**, der **Divisor wird aber NICHT erhöht!**

Solid State Drive (SSD)

Aufbau

- Durch Halbleiterbausteine realisiertes, nichtflüchtiges Speichermedium (zB. EEPROM, FLASH)
- Üblicherweise teurer als HDDs (per GB)
- Aber auch schneller! (**Seek time entfällt**)
- Geringer Stromverbrauch, geräuschfrei
- Unempfindlich gegen Erschütterungen, sehr leicht
- **Limitierte Anzahl an Schreibvorgängen**
 - Nur ca. 10^4 – 10^6 Schreibvorgänge pro Speicherzelle
 - “Wear Leveling”-Algorithmen: Daten werden gleichmäßig auf alle Speicherzellen verteilt

Überblick

- 1 Einführung und Formale Aspekte
 - Klassifikation
 - Design und Performance
 - Quantitative Methoden
- 2 Standard-Rechnerarchitektur
 - Abstraktionsebenen
 - Prozessor
- 3 Speicherverwaltung
 - Virtueller Speicher und Paging
 - Caching
- 4 Externer Speicher
 - Festplatten
 - Disk Scheduling
 - SSD
- 5 Skalare und superskalare/parallele Architekturen**
 - CISC und RISC**
 - Pipelining**
 - Superskalare Architekturen**

Wiederholung: ISA

ISA - Instruction Set Architecture / Befehlssatz

- **Gesamte nach außen hin sichtbare Architektur**
 - 1 Befehlssatz (s.o.)
 - 2 Registersatz (s.o.)
 - 3 Speichermodell (Größe des Adressraums, Breite der Busse, etc.)
- Muss für Erstellung von Maschinenprogrammen für diesen Prozessor bekannt sein
- Für Assemblerprogrammierung, Aufbau von Compilern, ...
- **ISA = Schnittstelle zwischen Hard- und Software**

CISC vs. RISC

CISC

- **Complex** Instruction Set Computer
- Viele, verhältnismäßig mächtige Einzelbefehle
- **Mikroprogrammierung:** Sequenzen für Steuerung der CPU werden aus Mikrocode-ROM abgerufen
- Befehlssatz wurde immer größer (oft mehrere Hundert)
- Immer kompliziertere Befehle kamen dazu

RISC

- **Reduced** Instruction Set Computer
- Wenige, einfache Maschinenbefehle

CISC

CISC: Vorteile

- Flexibilität: Befehlssatz auf Software-Ebene erweiterbar
- Fehlerbehebung: Neuer Mikrocode auch beim Kunden einspielbar
- **Kompatibilität** – Emulation: Befehlssatz von Vorgängern auf SW-Ebene nachbildbar

CISC: Nachteile

- **Dekodierung der vielen komplexen Befehle aufwändig**
 - Dekodierungseinheit brauchte mehr Zeit und Platz auf Chip
- ⇒ Bsp. Intel 386: Addition: 2 Takte; Multiplikation: bis zu 38 Takte

RISC

RISC

- **Kein Mikrocode, keine algorithmische Abarbeitung**
- **Befehl muss in Hardware implementiert sein**
- Befehlssatz ist kleiner, enthält keine komplizierten Befehle
- IBM, Mitte 70er: 10 einfache Befehle machen 2/3 des Programmcodes aus
- Rest durch Folge einfacher RISC-Befehle ersetzen
- **Skalarität:** Möglichst in jedem Takt einen Befehl bearbeiten. ($> 1 \Rightarrow$ superskalar)

Ohne Pipelining

Takt	Instr-Fetch	Decode	Opnd-Fetch	Execute	Write
1	Befehl 1	×	×	×	×
2	×	Befehl 1	×	×	×
3	×	×	Befehl 1	×	×
4	×	×	×	Befehl 1	×
5	×	×	×	×	Befehl 1
6	Befehl 2	×	×	×	×
7	×	Befehl 2	×	×	×
8	×	×	Befehl 2	×	×
9	×	×	×	Befehl 2	×
10	×	×	×	×	Befehl 2

Pipelining

Ohne Pipelining

- Jede Stufe nur 1 von 5 Takten beschäftigt
- Auslastung 20%

Pipelining Prinzip

- Fließbandverarbeitung
- **Befehle überlappend und parallel durch verschiedene Bearbeitungsstufen schleusen**
- **In jedem Takt:**
 - 1 Fertigen Befehl herausnehmen
 - 2 Jeden andren Befehl in nächste Bearbeitungsstufe
 - 3 Neuen Befehl in erste Stufe

Pipelining

Takt	Instr-Fetch	Decode	Opnd-Fetch	Execute	Write
1	Befehl 1	×	×	×	×
2	Befehl 2	Befehl 1	×	×	×
3	Befehl 3	Befehl 2	Befehl 1	×	×
4	Befehl 4	Befehl 3	Befehl 2	Befehl 1	×
5	Befehl 5	Befehl 4	Befehl 3	Befehl 2	Befehl 1
6	Befehl 6	Befehl 5	Befehl 4	Befehl 3	Befehl 2
7	Befehl 7	Befehl 6	Befehl 5	Befehl 4	Befehl 3
8	Befehl 8	Befehl 7	Befehl 6	Befehl 5	Befehl 4

- **Takt 1-5: Latenzzeit - nur beim Befüllen relevant**
- **Takt 6-8: Ein Befehl pro Takt!**
- Nach Befüllen **theoretisch** 100% Auslastung!

Praktische Probleme

Interlocks

- Z.B. Hauptspeicherzugriffe, kann nicht immer in einem Takt erfolgen
- Nur dann, wenn Datum in L1-Cache und dieser mit vollem Prozessortakt betrieben!

Data hazards (Datenabhängigkeiten)

- Z.B.: Befehl 1 schreibt in Register, das von Befehl 2 gelesen werden muss.
- Befehl 2 muss warten, bis Befehl 1 fertig ist
- Read-After-Write-Hazard (RAW-Hazard)

RAW-Hazard

Bsp Code

- 1 ADD R1, R1, R2 ; R1=R1+R2
- 2 SUB R3, R3, R1 ; R3=R3-R1
- 3 XOR R2, R2, R4 ; R2=R2 xor R4

Takt	Instr-Fetch	Decode	Opnd-Fetch	Execute	Write
1	ADD	×	×	×	×
2	SUB	ADD	×	×	×
3		SUB	ADD	×	×
4			SUB	ADD	×
5				SUB	ADD
6					SUB

SUB würde veraltete Daten verwenden!

Lösung über Leerbefehl

Takt	Instr-Fetch	Decode	Opnd-Fetch	Execute	Write
1	ADD	×	×	×	×
2	NOP	ADD	×	×	×
3	SUB	NOP	ADD	×	×
4		SUB	NOP	ADD	×
5			SUB	NOP	ADD
6				SUB	NOP
7					SUB

- Wenn SUB ausgeführt wird, liegt Ergebnis von ADD bereits vor
- Leertakte: Leistungseinbußen

Lösung über Compiler

- **Compiler versucht, Datenabhängigkeiten zu erkennen und ordnet Instruktionen falls möglich entsprechend um**
- **In diesem, und *nur* diesem konkreten Beispiel: XOR vor SUB, da XOR nicht datenabhängig ist!**
- **Keine Leistungseinbußen!**
- Zusätzlich auf CPU-Ebene: **Out of Order Execution**

Takt	Instr-Fetch	Decode	Opnd-Fetch	Execute	Write
1	ADD	×	×	×	×
2	XOR	ADD	×	×	×
3	SUB	XOR	ADD	×	×
4		SUB	XOR	ADD	×
5			SUB	XOR	ADD
6				SUB	XOR
7					SUB

Pipelining: Bedingte Sprungbefehle

Bedingter Sprungbefehl (`goto`, `jmp`, Schleifenabbruch)

- Eine Anweisung, die dazu dient, die Ausführung des Programms an anderer Stelle fortzusetzen

⇒ Programmteile können übersprungen werden

Probleme

- Sprungziel (Program Counter!) erst nach Auswertung (Ausführung) bekannt
- Nicht-sequenzielle Ordnung der Befehle

⇒ Je länger Pipeline, desto mehr zu verwerfen

Pipelines: Bedingte Sprungbefehle

Bsp: BREQ (Branch if equal) verzweigt zu Befehl 80. Befehle 2-5 zu verwerfen, Pipeline neu füllen (Dauer: Latenzzeit)

Takt	Instr-Fetch	Decode	Opnd-Fetch	Execute	Write
1	BREQ	×	×	×	×
2	Befehl 2	BREQ	×	×	×
3	Befehl 3	Befehl 2	BREQ	×	×
4	Befehl 4	Befehl 3	Befehl 2	BREQ	×
5	Befehl 5	Befehl 4	Befehl 3	Befehl 2	BREQ
6	Befehl 80	×	×	×	×
7	Befehl 81	Befehl 80	×	×	×
8	Befehl 82	Befehl 81	Befehl 80	×	×
9	Befehl 83	Befehl 82	Befehl 81	Befehl 80	×
10	Befehl 84	Befehl 83	Befehl 82	Befehl 81	Befehl 80

Sprungvorhersage und spekulative Ausführung

Branch Prediction

- Macht Annahme, ob Sprung genommen wird oder nicht
- Setzt dementsprechend Ausführung fort
- \Rightarrow Speculative Execution
- Wenn Annahme richtig: kein Zeitverlust
- \Rightarrow Idee: möglichst gute Annahmen treffen

Branch History Table

- Sprungbefehle im aktuellen Programmlauf beobachten und Statistik aufstellen
- Aufwändige Implementierungen erreichen Trefferquoten bis zu 99%

Superskalare Architekturen

Skalarität

⇒ Fähigkeit, in jedem Takt eine Instruktion ausführen

Superskalarität

- ⇒ Fähigkeit, Durchsätze von **mehr als einem Befehl** pro Takt zu erreichen. *“Instruction-level parallelism (ILP)”*
- Nur durch echte Parallelisierung erreichbar, die über Befehlspipelining hinausgeht
 - Superskalare Architekturen besitzen mehrere Dekodierungsstufen und mehrere Ausführungseinheiten die parallel arbeiten können

Ansätze

Zusätzlich (zu den parallelen Dekodierungsstufen und Ausführungseinheiten) verwendet man oft:

- **Out-of-Order-Execution:** Reihenfolge der Abarbeitung zwecks Beschleunigung verändern
- **Register Renaming:** Datenabhängigkeiten können durch Verwendung anderer Register (die keine Datenabhängigkeiten aufweisen) teilweise vermieden werden
- Verschiedene Pipeline Längen
- Mehrfache spekulative Ausführung
- **VLIW** (Very Long Instruction Word)
- Hyper-Threading

... bis hin zu mehreren Kernen/Prozessoren

└ Skalare und superskalare/parallele Architekturen

└ Superskalare Architekturen

Beispiel einer Superskalare Architektur

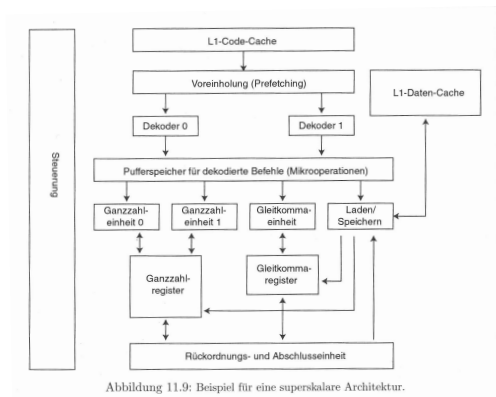


Abbildung 11.9: Beispiel für eine superskalare Architektur.

Abbildung: Die beiden Dekoder begrenzen die Leistung auf maximal zwei Instruktionen pro Takt

Beispiel-Befehlsfolge

```
;Beispiel-Befehlsfolge für die superskalare Architektur von Abb. 11.9
      AND R1,R2,R3      ; R1=R2 AND R3
      INC R0             ; Increment R0
      SUB R5,R4,R1       ; R5=R4-R1
      ADD R6,R5,R1       ; R6=R5+R1
      COPY R5, 1200h     ; R5=1200h
      LOAD R7,[R5]       ; Laden mit registerindirekter Adressierung
      FADD F0,F1,F2      ; F0=F1+F2
      FSUB F0,F0,F3      ; F0=F0-F3
```

Abbildung: R1-R31: Ganzzahlregister; F0-F15: Gleitkommaregister;
Datenabhängigkeiten beachten!

Beispiel der Abarbeitung in strikter Reihenfolge

Taktzyklus	Dekoder0	Dekoder1	Ganzzahl-Einheit0	Ganzzahl-Einheit1	Gleitkomma-einheit	Load/Store
1	AND	INC				
2	SUB	ADD	AND	INC		
3			AND	INC		
4			AND	INC		
5	COPY		SUB			
6			SUB			
7			SUB			
8	LOAD		ADD			
9			ADD			
10			ADD			
11	FADD		COPY			
12			COPY			
13			COPY			
14	FSUB	...			FADD	LOAD
15					FADD	LOAD
16					FADD	LOAD
17					FADD	LOAD
18	...				FSUB	
19					FSUB	
20					FSUB	
21					FSUB	

Abbildung 11.10: Ausführung der Beispiel-Befehlsfolge, wenn der Prozessor die Instruktionen in der richtigen Reihenfolge ausgibt. „...“ steht für einen nachfolgenden Befehl.

Abbildung: FADD(FloatADD) & FSUB sind unabhängig von vorhergehenden Befehlen, werden aber Aufgrund d. Reihenfolge erst später ausgeführt

└ Skalare und superskalare/parallele Architekturen

└ Superskalare Architekturen

Bsp. der Abarbeitung mit Out-of-Order Execution

Takt- zyklus	Dekoder0	Dekoder1	Ganzzahl- Einheit0	Ganzzahl- Einheit1	Gleit- komma- einheit	Load/ Store
1	AND	INC				
2	SUB	ADD	AND	INC		
3	COPY	LOAD	AND	INC		
4	FADD	FSUB	AND	INC		
5	SUB		FADD	
6			SUB		FADD	
7			SUB		FADD	
8	...		ADD		FADD	
9	...		ADD		FSUB	
10			ADD		FSUB	
11	...		COPY		FSUB	
12			COPY		FSUB	
13			COPY			
14	...					LOAD
15						LOAD
16						LOAD
17						LOAD

Abbildung 11.11: Ausführung der Beispiel-Befehlsfolge, wenn der Prozessor die Instruktionen in veränderter Reihenfolge ausgibt.

Abbildung: Vier Takte schneller, aber erheblicher Hardwareaufwand

Abarbeitung mit OoO Exec. + Register Renaming

Takt-zyklus	Dekoder0	Dekoder1	Ganzzahl-Einheit0	Ganzzahl-Einheit1	Gleit-komma-einheit	Load/Store
1	AND	INC				
2	SUB	ADD	AND	INC		
3	COPY	LOAD	AND	INC		
4	FADD	FSUB	AND	INC		
5	SUB	COPY	FADD	
6	...		SUB	COPY	FADD	
7			SUB	COPY	FADD	
8	ADD		FADD	LOAD
9	...		ADD		FSUB	LOAD
10			ADD		FSUB	LOAD
11					FSUB	LOAD
12					FSUB	

Abbildung 11.12: Ausführung der Beispiel-Befehlsfolge, bei veränderter Reihenfolge und Benutzung von Register-Umbenennung.

Abbildung: Register Renaming. Verwendung eines anderen Registers als R5 im COPY und im folgenden LOAD Befehl (siehe Beispiel-Befehlsfolge) bringt weitere fünf Takte Performance-Gewinn, da Datenabhängigkeiten vermieden werden können

Verschiedene Pipeline Längen und mehrfache spekulative Ausführung

Super-pipelined Prozessoren \Rightarrow mehr als 10 Pipeline-Stufen

- Höherer Durchsatz, da höherer Prozessortakt möglich
- Problem 1: Zugriffe auf Speicherbausteine lassen sich nicht weiter unterteilen
- Problem 2: Latenzzeit wächst mit Pipeline-Länge

Spekulative Ausführung bei superskalaren Architekturen

- Weiterhin ein Problem (evtl. höhere Latenzzeit!)
- Bei ausreichend Systemressourcen wird spekulative Ausführung in mehreren Zweigen durchgeführt
- Nach der Sprungentscheidung wird ein Zweig der nicht gebraucht wird ungültig gemacht

VLIW-Prozessoren

Very Long Instruction Word

- VLIW enthält mehrere Befehle, die zur Übersetzungszeit gebündelt wurden
 - Schon bei Übersetzung wird der Code analysiert und unter Kenntnis der Hardware entschieden, welche Maschinenbefehle parallel ausgeführt werden können
 - Übersetztes Programm enthält dann schon die Anweisungen zur Parallelausführung (muss nicht mehr vom Prozessor übernommen werden)
- ⇒ “explicit parallelism” (control over the parallel execution)
- ⇒ Gegenstück “implicit parallelism” (*automatic* parallel execution)

Hyper-Threading (Ausblick VO “Betriebssysteme”)

Thread (“Faden”): eigenständiger Teil eines Prozesses

- Hat seinen eigenen Programmfluss samt Programmzähler, Registerinhalten, und Stack
- Aber: Alle Threads eines Prozesses teilen sich denselben Adressraum

Hyper-Threading: schnelleres Umschalten zwischen 2 Threads

- Hintergrundregister und Registerzuordnungstabelle doppelt vorhanden
- Zweiter Registersatz enthält Daten des zweiten Threads

⇒ Umschalten zwischen zwei Threads geht damit schneller

Taktfrequenz der Prozessoren ist limitiert

Max. Taktfrequenz liegt seit Jahren bei ca. 3.5 GHz und kann kaum noch gesteigert werden

- Gründe: Wärmeentwicklung, Abstrahlung, ...
- Weitere Verbreiterung der Register bring keine große Leistungssteigerung – schon 64-Bit-Reg. kaum ausgenutzt
- Ebenso bringen mehrere parallele Ausführungseinheiten keine weiteren Vorteile, da zu viele Datenabhängigkeiten

Trend zu Parallelrechnern

- Prozessoren mit mehreren Kernen, oder
- Rechner mit mehreren Prozessoren im System
- Damit auch *“task-parallelism”* möglich

Mehrprozessorsystem vs. Mehrkernsystem

Mehrprozessorsystem

- Für ein echtes Mehrprozessorsystem braucht man
 - mehrere Sockel (“Steckplatz” für einen Prozessor) und
 - einen dementsprechenden Chipsatz (mehrere zusammengehörende integrierte Schaltkreise)
- ⇒ Aufwändig und teuer

Mehrkernsystem – SMP

- Günstiger ist es, mehrere Prozessorkerne in einem Chip unterzubringen
- Ein Chip ist ein integrierter Schaltkreis auf einem einzigen Halbleiterplättchen
- Bei mehreren gleichartigen Kernen spricht man von “Symmetrischen Multiprocessing” SMP

Mehrkernprozessoren (Multi-Core)

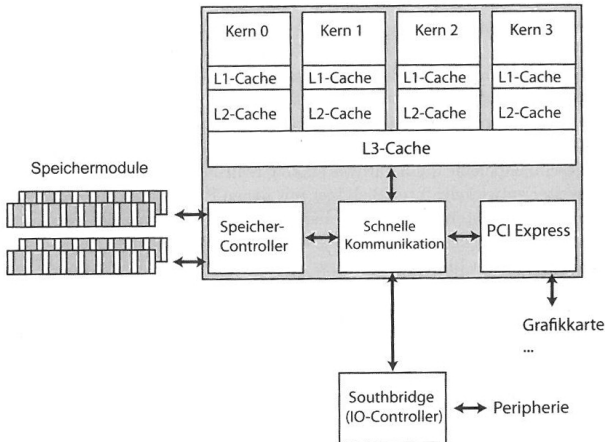


Abbildung 11.16: Vierkernprozessor mit integrierter PCI-Express-Schnittstelle und Southbridge

Mehrkernprozessoren (Multi-Core)

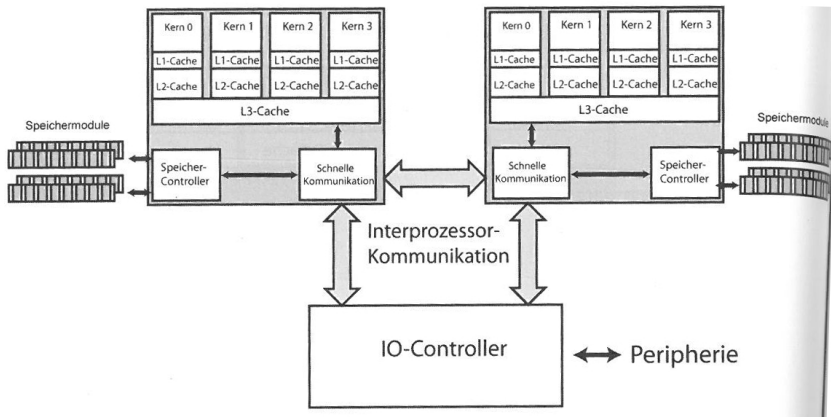
Cache-Designfragen und Cache-Organisation

- Gibt es private Caches für die Kerne?
 - Bei einem gemeinsamen Cache wäre Datentransfer sehr groß und langsam
 - Besser: jeder Prozessor(-kern) hat seinen eigenen Cache
- Wie werden die Caches konsistent gehalten?
 - Problem: In zwei Caches von verschiedenen Kernen liegt eine Kopie des gleichen Speicherblocks, eine davon wird durch Schreibzugriff verändert
 - “Cache coherency”, “bus snooping”

Shared Memory / Uniform Memory Access (UMA)

- Prozessoren der Multi-Prozessor-Hauptplatine können auf den vorhandenen Arbeitsspeicher “gleichwertig” zugreifen

Mehrprozessorsystem (Multi-CPU)



Mehrprozessorsystem (Multi-CPU)

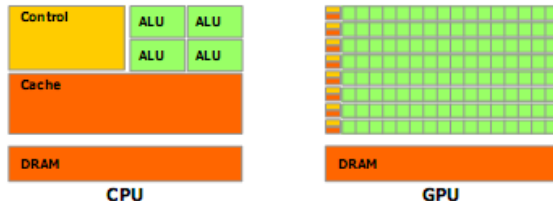
Inter-Prozessor-Kommunikation (IPC/IPK)

- Leistungsstarke Schnittstelle um untereinander (zwischen Prozessoren) schnell Daten auszutauschen
- zB: *Quick Path Interconnect* (Intel); *Hypertransport* (AMD)

Distributed (Shared) Memory / Non-Uniform Memory Access (NUMA)

- Jeder Prozessor besitzt eigenen Arbeitsspeicher
- Zugriff auf “fremde” Arbeitsspeicher nur mittels Arbeitsspeicher-Verwalter (Bsp. MESI-/MOESI-Protokoll)

Manycore Prozessor



Relative Allocation of Transistors for Computation, Data Caching, and Flow Control - NVIDIA CUDA C Programming Guide - Version 4.2 - 4/5/2012 - copyright NVIDIA Corporation 2012

- Große Anzahl an einfachen, unabhängigen Kernen
- Beispiel: nvidia GPGPU (General Purpose Computation on Graphics Processing Unit)
- Programmier-APIs: OpenCL, CUDA, C++ AMP
- Typisch: *“Data-parallel”* job execution

Speedup durch Parallelisierung

Amdahl's Law

- Nur gewisser Anteil des Codes kann parallel ausgeführt werden (Vergleich F_E von Kapitel Computerdesign)
- Der Rest muss sequenziell abgearbeitet werden
 - Mögliche Gründe: Vorbereitung der Parallelausführung, Erfassen der Ergebnisse, Datenabhängigkeiten, nicht parallelisierbarer Code, IPC, ...
- Nur der **parallele** Teil kann statt auf einem Prozessor auf n Prozessoren (Vergleich S_E) ausgeführt werden
- Sequentieller Anteil f , parallelisierbarer Anteil $(1 - f)$
- *Theoretisch* erreichbarer Speedup: $S = \frac{n}{1+(n-1)f}$
- Erfolgt hängt u.a. von Caches und Speicherlatenzen ab

Speedup durch Parallelisierung

Beispiel zu Amdahl's Law bei System mit 4 Prozessoren ($n=4$)

- **Achtung!** Tippfehler in K. Wüst, Mikroprozessortechnik, 4. Aufl. p218.
Wenn f gleich 0.1 ist, dann muss der parallelisierbare Anteil 0.9 sein

Bsp. 1: Speedup bei einem parallelisierbaren Anteil von 90%

- $(1 - f) = 0.9 \Rightarrow f = 0.1 \Rightarrow S = \frac{4}{1+(4-1)0.1} \Rightarrow S = 3.077$
- Alternative Berechnung mittels F_E und S_E :

$$S = \frac{1}{(1-F_E)+F_E/S_E} = \frac{1}{(1-0.9)+0.9/4} = \frac{1}{(0.1+0.225)} = 3.077$$

Bsp. 2: Speedup bei einem parallelisierbaren Anteil von 10%

- $(1 - f) = 0.1 \Rightarrow f = 0.9 \Rightarrow S = \frac{4}{1+(4-1)0.9} \Rightarrow S = 1.08$