1 – Wiederholung IT-Grundlagen

1.1 – Der Von-Neumann-Rechner

Von-Neumann-Rechner
Komponenten

ALU (Arithmetic Logic Unit) – Rechenwerk

- Führt Rechenoperationen und logische Verknüpfungen durch

Control Unit – Steuerwerk oder Leitwerk

- Interpretiert Anweisungen eines Programms
- Verschaltet Daten und notwendige ALU-Komponenten

CPU

Bus – System

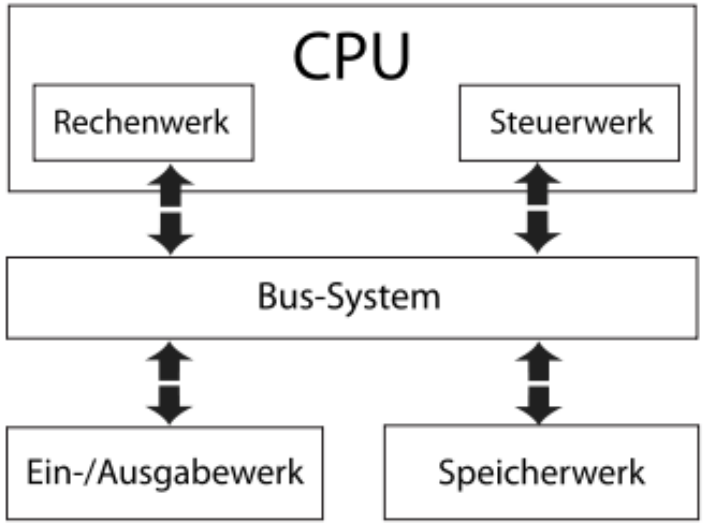
- Datenbus und Adressbus

I/O Unit – Eingabe-/Ausgabewerk

- Steuert Ein- und Ausgabe von Daten

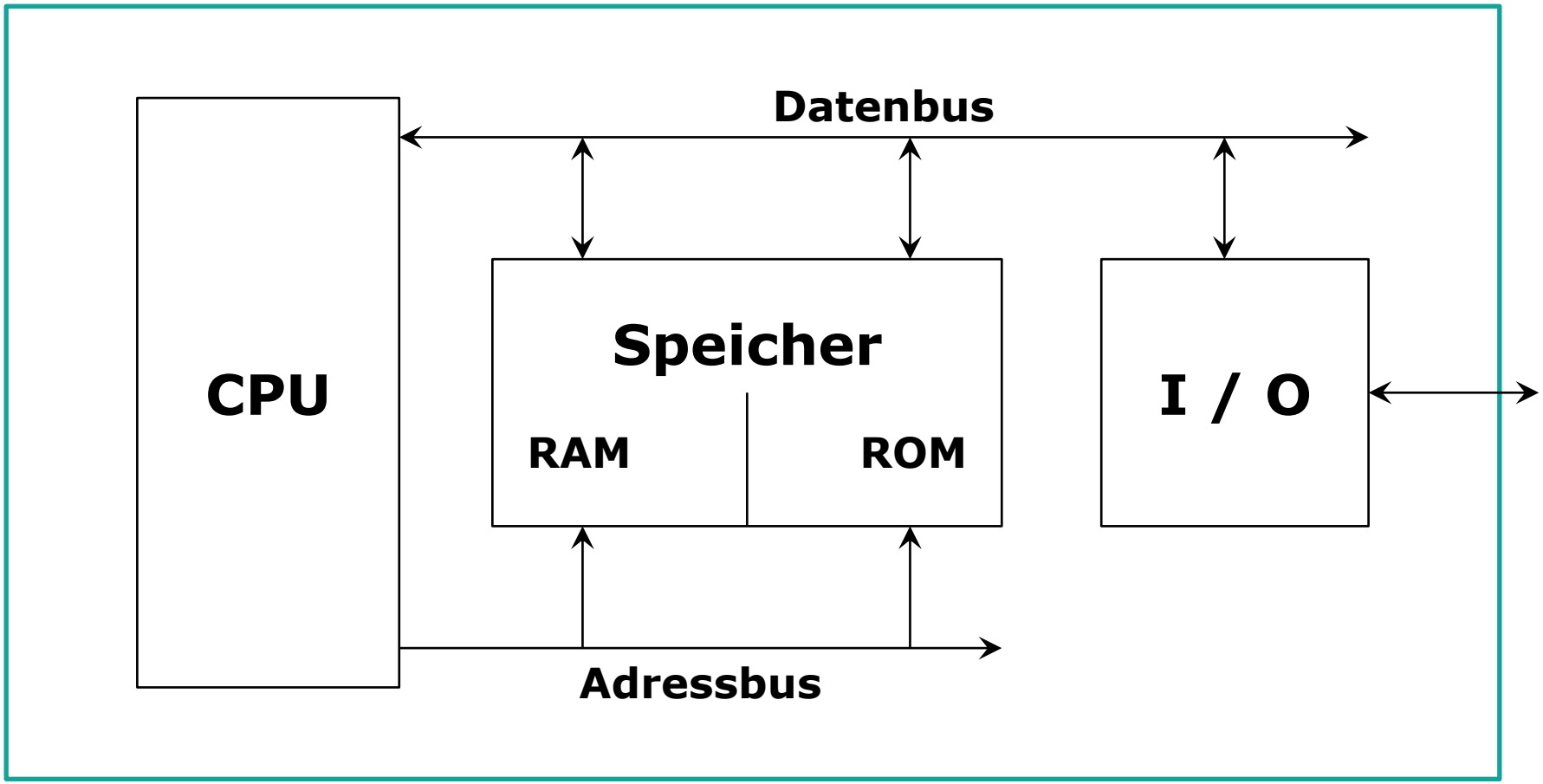
Memory – Speicherwerk

- Speichert Programme und Daten
  - > RAM (Random Access Memory)
  - > ROM (Read-Only Memory)



Von-Neumann-Rechner
Komponenten

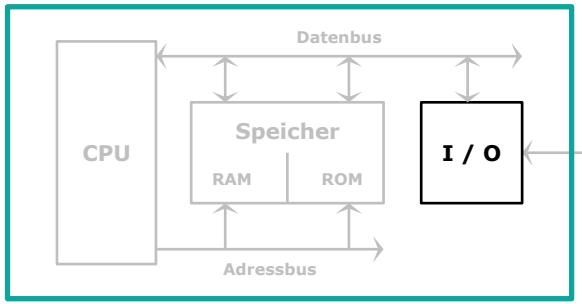
Übersicht



Von-Neumann-Rechner
Komponenten

Ein- und Ausgabegeräte

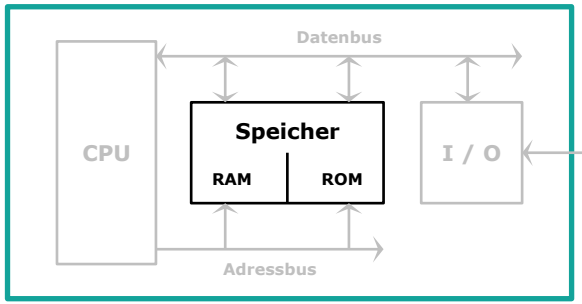
- Transport von Programm und Daten in den Hauptspeicher, DMA (Direct Memory Access)
- Beispiel: Ausgabe der im Hauptspeicher abgelegten Ergebnisse des Programmlaufs

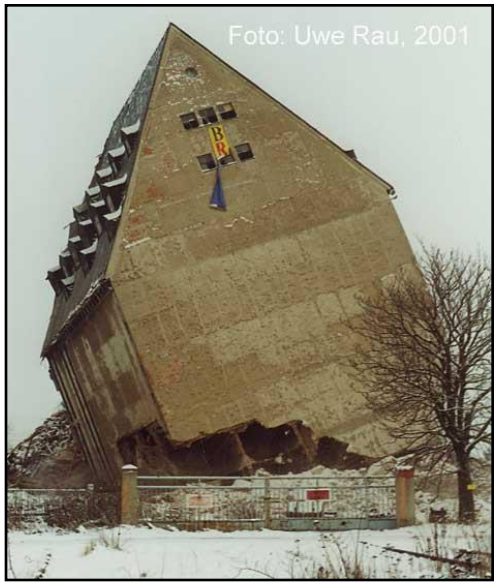


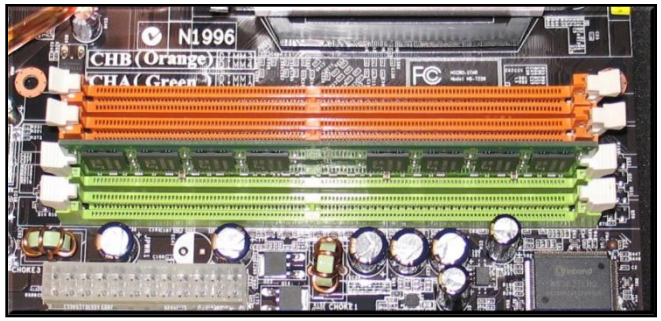
Von-Neumann-Rechner
Komponenten

Hauptspeicher

- gleichgeartete Speicherplätze
- Adresse je Speicherplatz
- Aufnahme von Daten und Programmen
- BIOS (Basic I / O System)



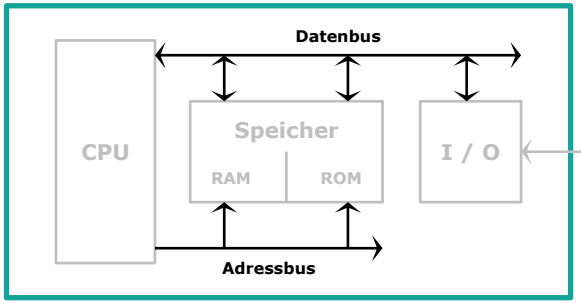


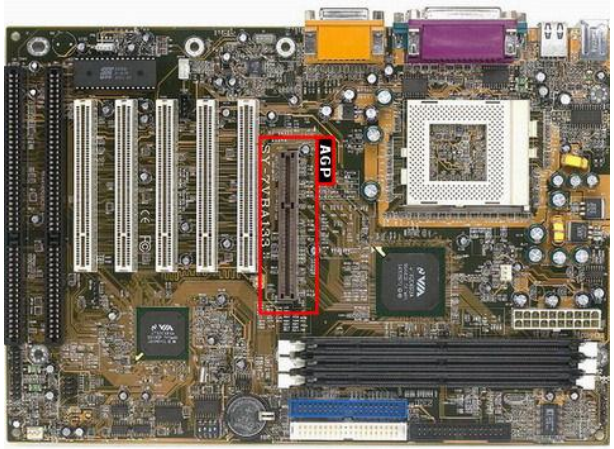


Von-Neumann-Rechner
Komponenten

Busse

- Ermöglichen die Kommunikation zwischen Speicher, CPU und I/O Einheit
- Man unterscheidet Daten- und Adressbus

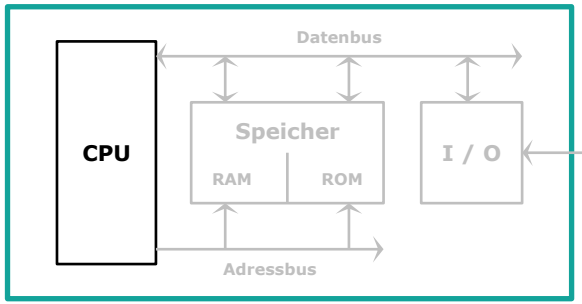


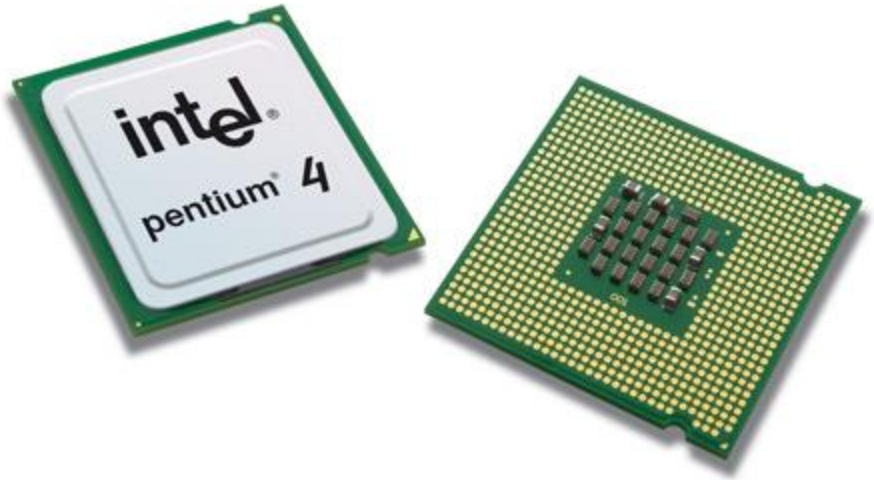


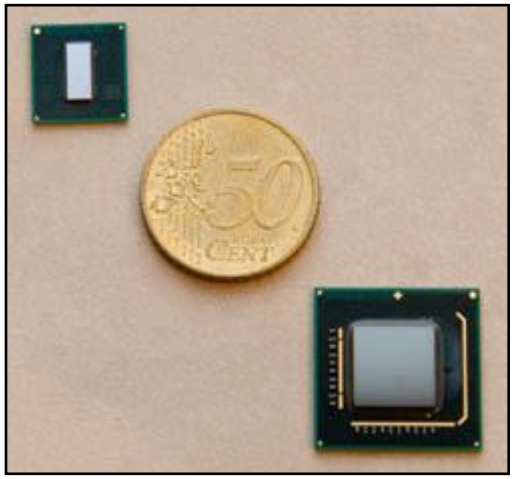
Von-Neumann-Rechner
Komponenten

Prozessor

- Leitwerk und Rechenwerk
- Bsp. Intel Pentium 4, Atom







Von-Neumann-Rechner
Eigenschaften

Der Von-Neumann-Flaschenhals:

- Die Ausführungszeit eines Befehls ist meistens sehr kurz
- Die Zugriffszeit auf den Speicher ist im Vergleich sehr lang
- Die CPU wartet, während Dinge aus dem Speicher geladen oder in den Speicher geschrieben werden

⇨ **Die Ausführungszeit eines Programms hängt (auch) von der Anzahl der Speicherzugriffe ab**

⇨ **Viele Ideen und Strategien, um Teilbereiche zu parallelisieren und zu beschleunigen, z.B. Befehls-Queues/Pipelines, Branch-Predictions etc.**

Von-Neumann-Rechner
CPU-Übersicht

CISC (Complex Instruction Set Computer)

- Komplexe Befehle, Bsp. Division
- Befehlssatz meist in Form von Microcode
- Wenig Register, Bsp. x86 (9 Register)

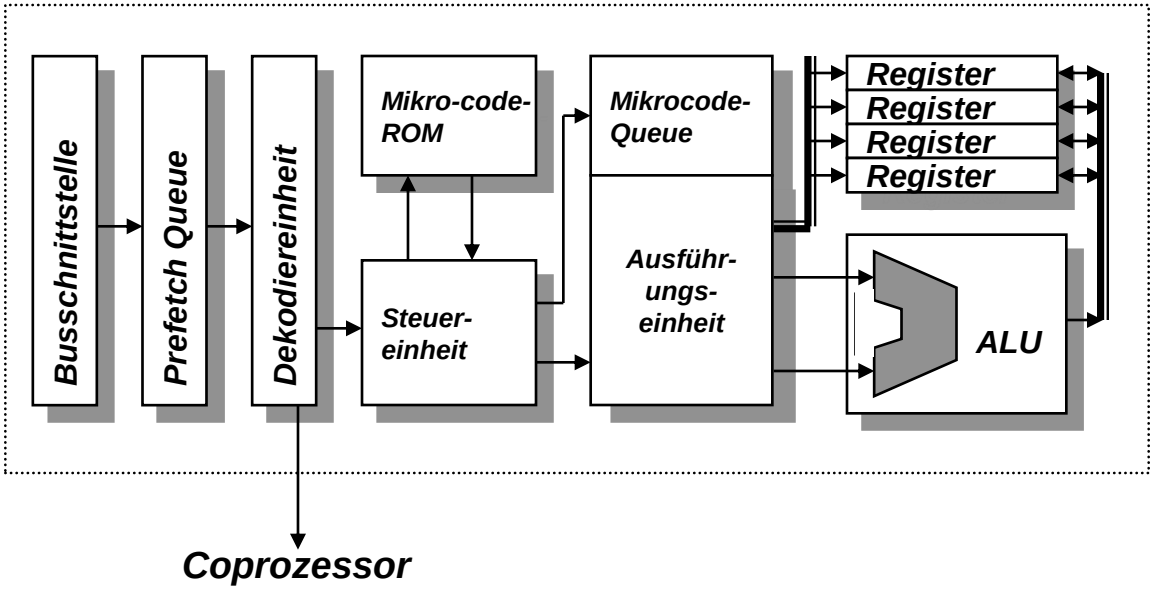
RISC (Reduced Instruction Set Computer)

- Verzichtet auf komplexe Befehle
- Einzelnen Befehle sind fest verdrahtet
- Viele Register, Bsp. PowerPC (32 GPR Register)

Von-Neumann-Rechner
CISC CPU

CISC

- Prozessor stellt eine große Anzahl zum Teil sehr komplexer Befehle zur Verfügung
- Abarbeitungszeit verschiedener Befehle ist unterschiedlich lang
- Befehle haben unterschiedliche Länge (Speicherplatz)
- CISC-Mikroprozessoren sind mikroprogrammiert



Von-Neumann-Rechner
CISC CPU

Argumente für CISC:

- Komplexe Befehle verringern den Hauptspeicherbedarf für Programmcode
- Komplexe Befehle vereinfachen den Compiler
- Oftmals erhält man so fehlerfreiere Programme
- Befehlskomplexität bedingt weniger Hauptspeicherzugriffe, was die Geschwindigkeit erhöht
- Ein Rechner kann mikroprogrammiert verschiedene Architekturen darstellen

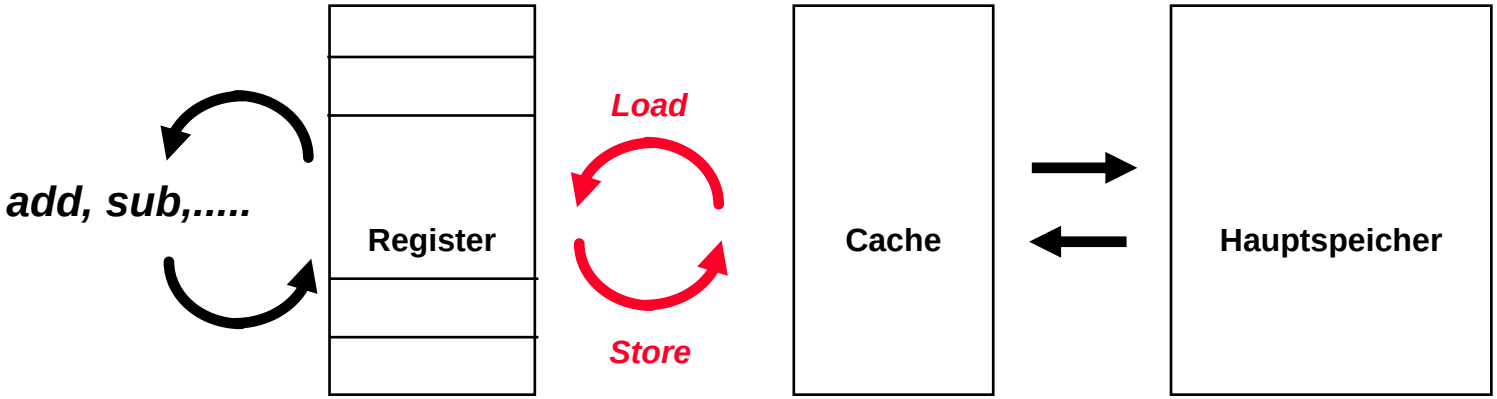
Argumente gegen CISC:

- Die komplexen Befehle, die eigentlich die Compiler vereinfachen sollen, werden kaum von den Compilern benutzt
- Hauptspeicher ist keine Mangelware mehr (Ausnahme: Embedded Systems)
- Caching verringert die Anzahl der Hauptspeicher Zugriffe ebenfalls effizient

Von-Neumann-Rechner
RISC CPU

RISC

- Prozessor stellt weniger, elementare Funktionen zur Verfügung
- Alle Befehle haben die gleiche Länge
- Dekodieraufwand kleiner
- Befehle sind „fest verdrahtet" und von daher optimiert
- Ziel ist es, pro Prozessorzyklus einen Befehl abzuarbeiten
- Prozessor-Pipeline erhöht den Verschachtelungsgrad
- Viele Register ermöglichen ein Register-Window -> Context-Switch schneller



Von-Neumann-Rechner
RISC CPU

Kriterien für RISC-Architekturen:

- Mindestens 80% der Befehle werden in einem Taktzyklus ausgeführt
- Alle Befehle werden mit einem Maschinenwort codiert
- weniger als 128 Maschinenbefehle
- weniger als 4 Adressierungsarten
- weniger als 4 Befehlsformate
- Speicherzugriffe nur über LOAD/ STORE-Befehle
- Register-Register Architektur (ALU)
- mehr als 32 Prozessorregister
- festverdrahtete Maschinenbefehle (keine Mikroprogrammierung)
- höhere Programmiersprachen werden durch optimierende Compiler unterstützt

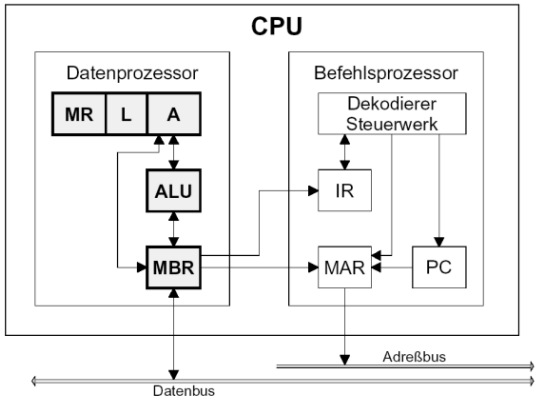
Nach D. Tabak ("RISC Architecture", 1995) müssen mindestens 5 der folgenden Kriterien erfüllt sein, damit eine RISC-Architektur vorliegt

Von-Neumann-Rechner
RISC vs. CISC

	RISC	CISC
Ausführungszeit	1 Datenpfadzyklus (fetch, decode, execute)	≥ 1 Datenpfadzyklus
Instruktionsanzahl	klein	groß
Instruktionsformat	einfach/einheitlich	variabel
Steuerung über	Hardware	Mikroprogramm
Hauptspeicherzugriffe	LOAD/STORE-Architektur	keine Einschränkungen
Pipelining	möglich	nicht möglich
Verlagerung der Komplexität	Compiler	Hardware

Rechenwerk (Datenprozessor)

- Ausführung von arithmetischen und logischen Operationen **(ALU)**
- Zwischenspeicherung mittels Registern:
  - Akkumulator-Register **(A)**
  - Puffer-Register **(MBR)**
  - Übertrags-Register **(L)**



ALU (**A**rithmetic **L**ogic **U**nit): führt die Rechenoperationen durch

A (**A**kkumulator): dient zur Aufnahme von Operanden

L (**L**ink Register): z.B. zur Aufnahme eines Additionsübertrags

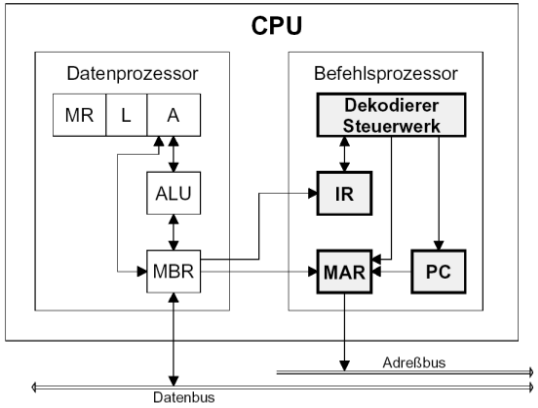
MR (**M**ultiplikator **R**egister): z.B. zur Aufnahme von Multiplikationsergebnissen

MBR (**M**emory **B**uffer **R**egister): dient zur Kommunikation mit dem Speicher

Leitwerk (Befehlsprozessor) – Übersicht

- Überwachung der Programm-durchführung, also der im ge-speicherten Programm festgelegten Arbeitsanweisungen

1. Adresse des ersten Befehls in den Befehlszähler laden **(PC)**
2. Leitwerk holt den Befehl in das Befehlsregister **(IR)**
3. Befehlszähler (PC) um 1 erhöht.
4. Ausführung des Befehls
5. Weiter bei 2.



IR (**I**nstruction **R**egister): enthält den jeweils aktuell bearbeiteten Befehl

MAR (**M**emory **A**ddress **R**egister): enthält die Adresse des als nächsten anzusprechenden Speicherplatzes

PC (**P**rogram **C**ounter): beinhaltet die Adresse des als nächsten auszuführenden Befehls

Dekodierer/Steuerwerk: Dekodierung von Befehlen und Steuerung der Ausführung

Von-Neumann-Rechner
Leitwerk

Befehlsarten

- Arithmetische Befehle (Grundrechenarten)
- Logische Befehle (Vergleichsoperationen, …)
- Transferbefehle (Speicherung, …)
- Steuerbefehle (Sprungbefehle, …)
- …

Befehlsaufbau

- Jeder Befehle bestehet aus zwei Abschnitten:
  - > Operationsteil (Welcher Befehl?)
  - > Argumente (Womit, Wohin, …?)

Von-Neumann-Rechner
Pipelining

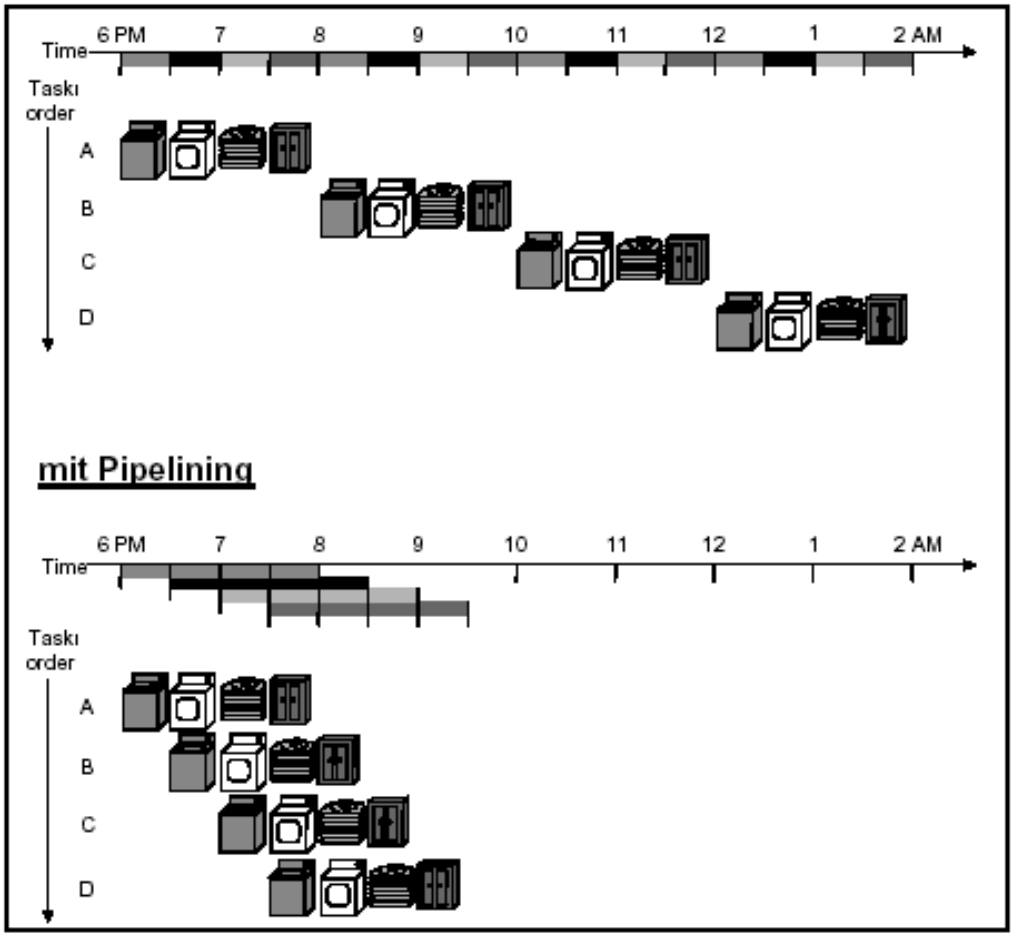
Idee: „Never waste time“

Sie kommen aus dem Urlaub und es ist viel schmutzige Wäsche zu waschen!

- Zur Verfügung stehen:
  - > Eine Waschmaschine (0.5 Std. Laufzeit)
  - > Ein Trockner (0.5 Std. Laufzeit)
  - > Eine Bügelmaschine (0.5 Std. Arbeit zum Bügeln)
  - > Ein Wäscheschrank (0.5 Std. Arbeit zum Einräumen)
- Jede Person (A, B, C, D) wäscht seine Wäsche selbst.

Von-Neumann-Rechner
Pipelining

- **Dauer: 8 Stunden**
- **Dauer: 3,5 Stunden**



Von-Neumann-Rechner
Pipelining

Pipelining auf RISC-Architekturen:

- Die Befehlsausführung funktioniert immer nach demselben Prinzip
- Es gibt unabhängig Teile einer Befehlsabarbeitung
  - > Diese können parallelisiert werden
  - > Diese benutzen im Allgemeinen unabhängige Hardwarekomponenten

5-stage pipeline:

- Instruction fetch (***IF***): Befehl lesen
- Instruction decode (***ID***): Befehl identifizieren und Daten bereitstellen
- Execute (***EX***): Ausführen
- Memory Access (***MEM***): Speicherzugriff
- Writeback (***WB***): Ergebnis schreiben

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Situationen, in denen die regelmäßige Abarbeitung beim Pipelining unterbrochen werden muss:

- Structural-Hazards:
  - > Instruktionen benötigen dieselbe Hardware und können nicht überlappend ausgeführt werden
- Data-Hazards:
  - > Datenabhängigkeiten zwischen aufeinander folgenden Instruktionen
- Control-Hazards:
  - > Änderung der sequentiellen Abarbeitung von Befehlsfolgen durch bedingte Sprünge

Structural-Hazards

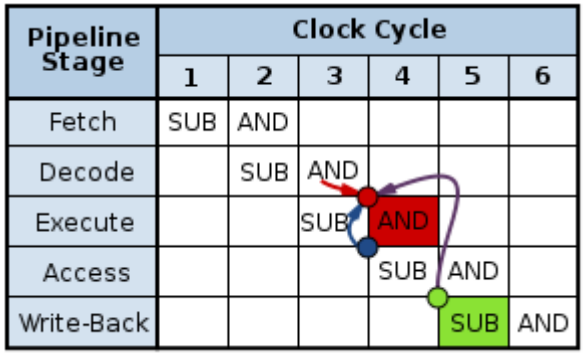
- Konsekutive Instruktionen benötigen dieselbe Hardware und können nicht überlappend ausgeführt werden

Lösung

- Vermeidung typischerweise durch Replizieren der Hardware, Bsp. ALU

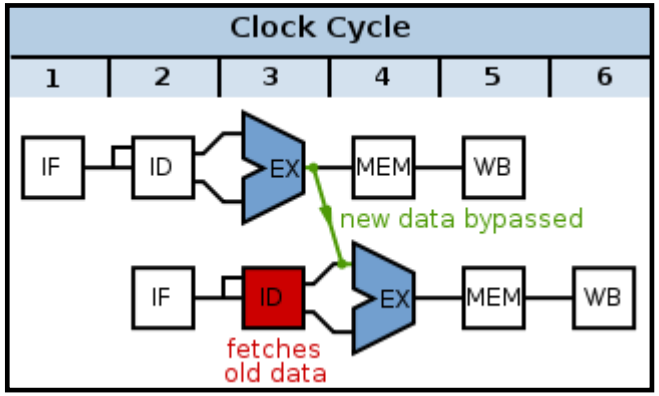
Data-Hazards

- Datenabhängigkeiten zwischen aufeinander folgenden Instruktionen, Bsp.:
  - > SUB r3,r4 -> r10
  - > AND r10,*adr* -> r11
- Cycle 3:
  - > SUB berechnet Wert für r10, und
  - > AND wird decodiert und (alter) Wert von r10 geholt
  - > SUB Ergebnis noch nicht in r10 ⇨ Ergebnis falsch



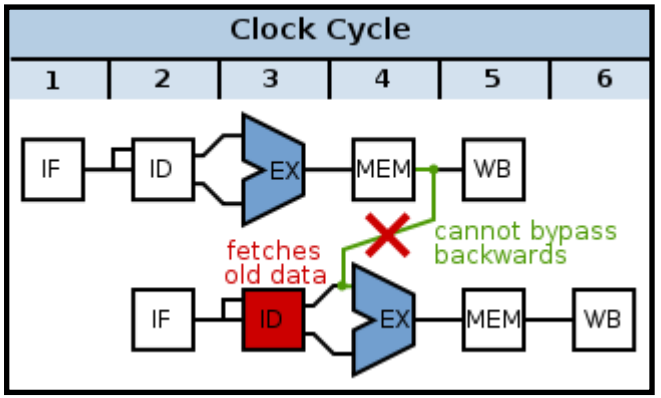
Lösung: *data bypassing*

- Eingangsdaten für Befehle stammen aus Registern oder „Zwischenergebnissen“ (roter, blauer oder grüner Kreis)



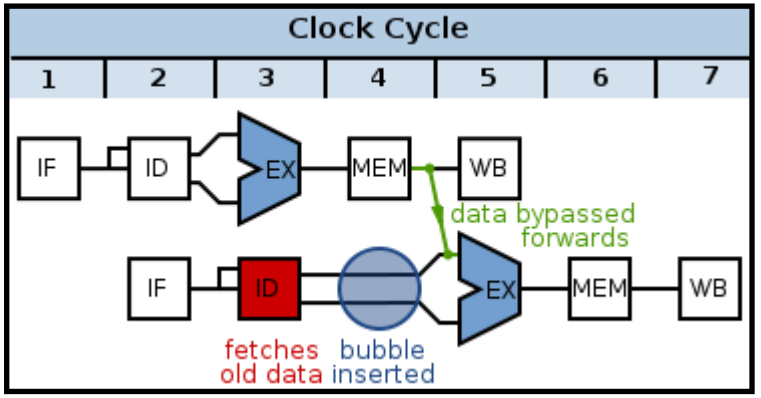
Data-Hazards

- Datenabhängigkeiten zwischen aufeinander folgenden Instruktionen, Bsp.:
  - > LOAD *adr2* -> r10
  - > AND r10, *adr* -> r11
- Data aus *adr2* steht ab MEM stage zur Verfügung, zu dem Zeitpunkt ist AND schon verarbeitet



Lösung: *NOP Befehl oder stall*

- Befehl NOP (no operation) einfügen
- Pipeline Ausführung anhalten bis aktuelle Daten vorhanden
- Verhält sich wie Luftblase (bubble)



Control-Hazards

- Bedingte Sprungbefehle ändern den als nächstes auszuführenden Befehl
  - > Nächster Befehl steht erst am Ende des vorherigen Befehls fest
  - > Die Pipeline hat bereits begonnen unnötige Befehle abzuarbeiten

Lösung:

- Stall / Freeze (Nops einfügen)
- Dynamic Branch Prediction (Berücksichtige vormaligen Ausgang)
  - Predict Taken
  - Predict Not Taken
- Branch Delay Slot (Füge bedingungsunabhängige Instruktion ein)

1.2 – Speicher und Cache

Speicher
Allgemeines

Speicher (Memory)

- Funktionseinheit, die Daten aufnimmt, aufbewahrt und abgibt

Random Access Memory (RAM)

- Flüchtiger Speicher, welcher von Programmen (beliebig)geschrieben und gelesen werden kann

Read Only Memory (ROM)

- Festwertspeicher
- Kann nur gelesen werden

Cache

- Speicher, der die Daten und Befehle bereithält, die vom Prozessor wahrscheinlich als nächstes benötigt werden
- Aufgebaut aus Speicherbausteinen mit niedriger Zugriffszeit

Es gibt verschiedene technische Realisierungen von Speicher ...

Speicher
Kenngrößen

Typ

- DRAM (Dynamic Random Access Memory), SRAM, ROM, PROM, EPROM ...

Speicherkapazität

- Gibt an, wie viel Informationen gespeichert werden kann (in Byte)

Technologie

- DDR-SDRAM (Double Data Rate Synchronous RAM)

Formfaktor

- SIMM, DIMM (Dual Inline Memory Module)

Bus-Geschw.

Chip	Modul	Speichertakt	I/O-Takt [2]	Effektiver Takt [3]	Übertragungsrate pro Modul	Übertragungsrate Dual-Channel
DDR-200	PC-1600	100 MHz	100 MHz	200 MHz	1,6 GB/s	3,2 GB/s
DDR-266	PC-2100	133 MHz	133 MHz	266 MHz	2,1 GB/s	4,2 GB/s
DDR-333	PC-2700	166 MHz	166 MHz	333 MHz	2,7 GB/s	5,4 GB/s
DDR-400	PC-3200	200 MHz	200 MHz	400 MHz	3,2 GB/s	6,4 GB/s

[2]) Geschwindigkeit der Anbindung an den Speichercontroller von CPU oder Mainboard

[3]) Effektiver Takt im Vergleich zu SDR-SDRAM (theoretisch)

PC-XXXX: Das XXXX berechnet sich durch (2 × Speichertakt × Busbreite)/8 (Busbreite = 64 bit) und entspricht der Datenrate in MB/

Speicher
Kenngrößen

Latenzzeit

- Column Address Strobe Latency (CL bzw. „CAS Latency“)
- Verzögerung zwischen der Adressierung und der Bereitstellung der an dieser Adresse gespeicherten Daten

Zugriffszeit

- Durchschnittliche Zeitspanne vom Anlegen einer Adresse zum Lesen einer Information

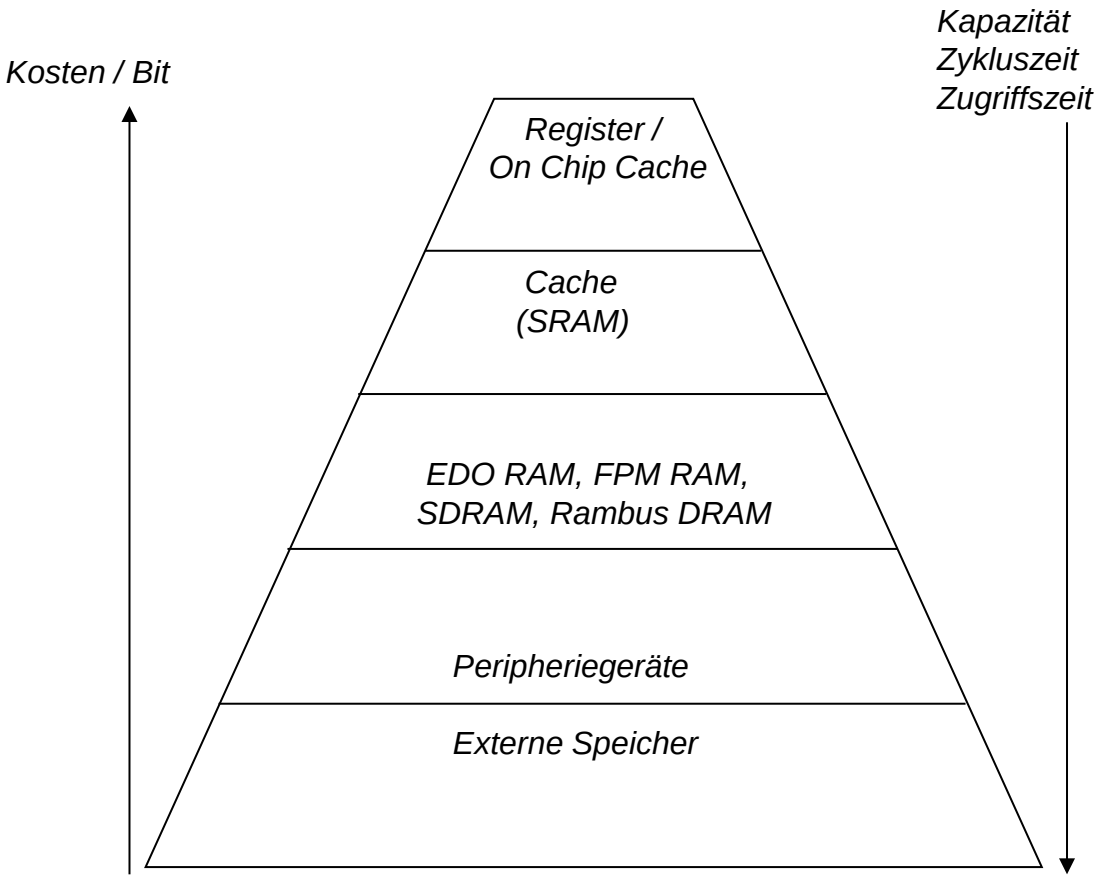
Zykluszeit

- Zeit, nach der ein Lese- oder Schreibvorgang wiederholt werden kann (Zeit zwischen dem Anlegen einer Adresse)
- Umfasst Zugriffszeit und „Erholzeit“

Datenintegrität

- ECC (Error Correcting Code); Fehlerkorrekturmodus, kann sowohl einzelne Bitfehler erkennen, als auch korrigieren (vgl. Parity-Bit)

Speicher
Kosten und Zeiten



Es war einmal…
Expertenmeinung

„640 KBytes [Arbeitsspeicher] sollten für jeden genug sein…“

(Bill Gates, 1981)

Speicher
Dynamic RAM (DRAM)

Hält Informationen nur kurze Zeit

- Erhaltung der Information durch Auffrischung (Refresh)

Die Information eines Bits wird in einem Kondensator gehalten

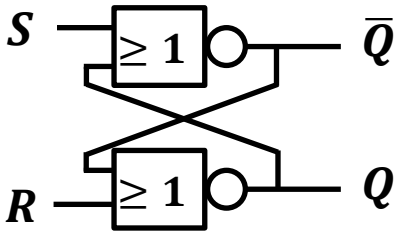
- Pro Bit sind ein Transistor (Schalten, Lesen/Schreiben) und ein Kondensator (Energiespeicher) erforderlich
  - > Hohe Integrationsdichte, also wenig Chip-Fläche
  - > Geringer Stromverbrauch

Die Ladung des Kondensators wird durch das Lesen zerstört

- Erneutes Einschreiben der Information nach dem Lesevorgang nötig

Die Information wird in einem Flip-Flop (bistabile Kippstufe) gehalten

Set und Reset



R	S	Q	Kommentar
0	0	Q	„Speichern“
0	1	1	Setzen
1	0	0	Zurücksetzen
1	1	X	Undefiniert

Unterschiede zu DRAM

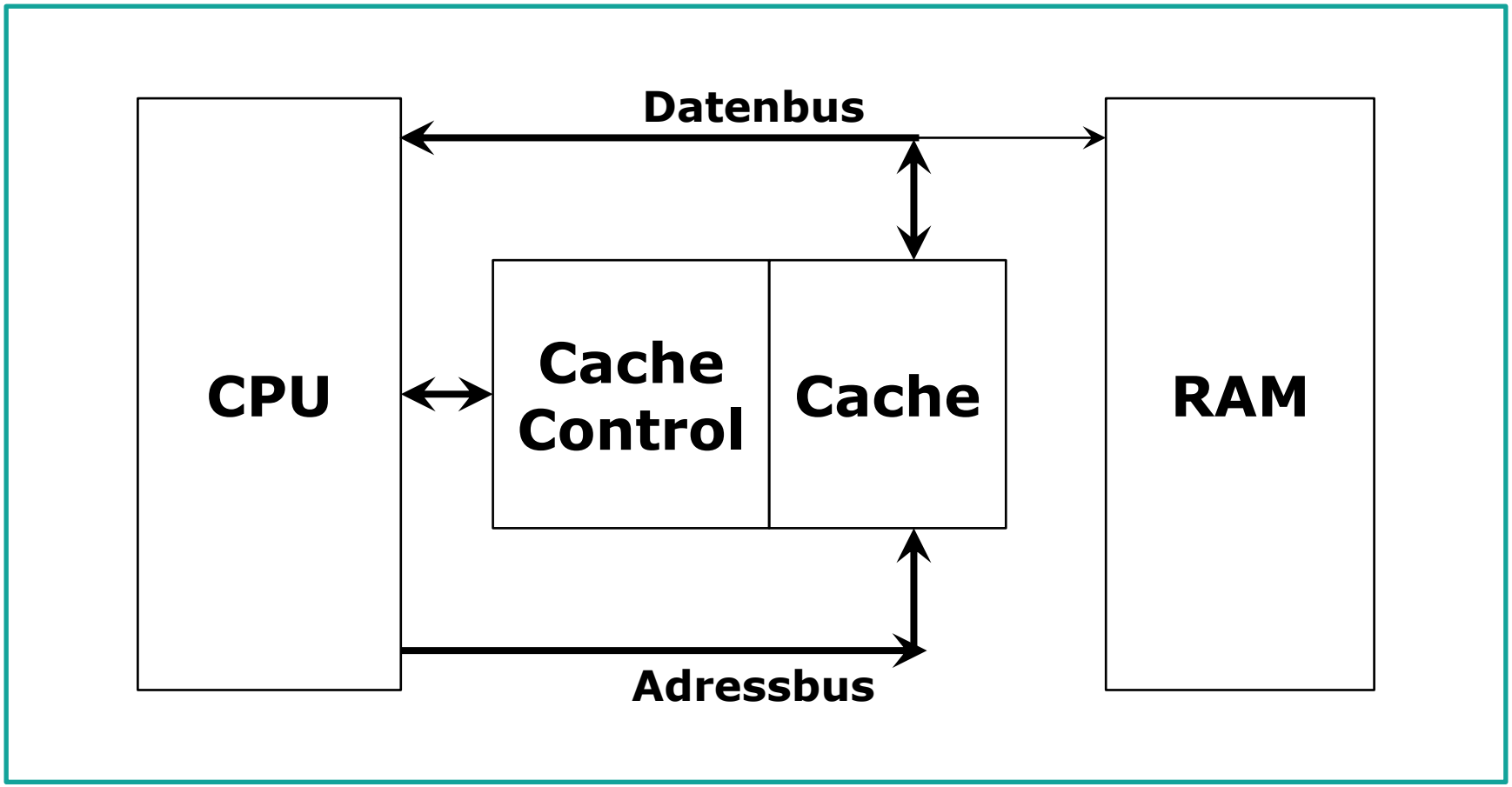
- Benötigt kein Refresh
- Zugriffszeiten kürzer
- Kleinere Integrationsdichte (≈ ¼)
- Höhere Kosten

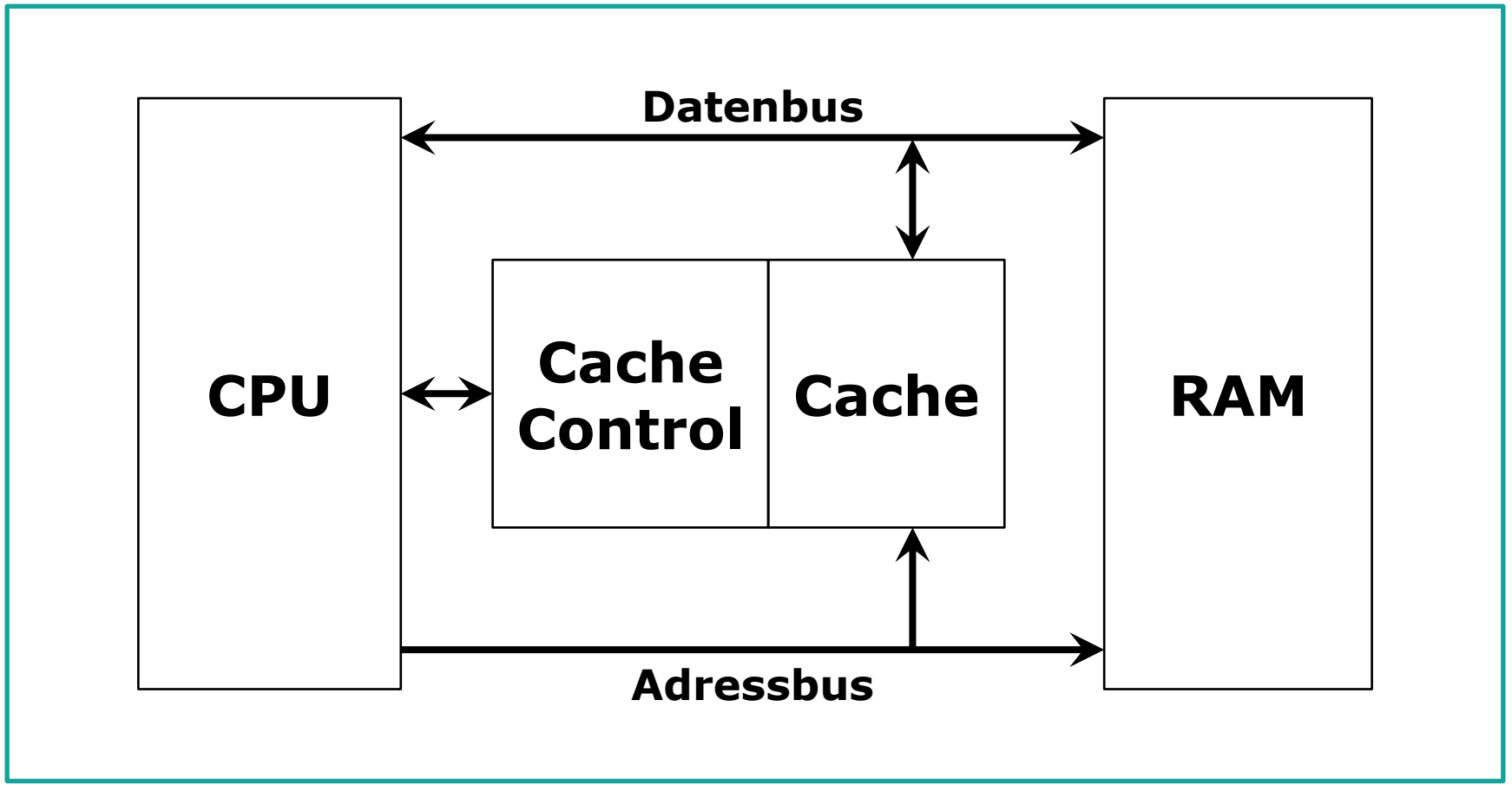
Problem

- Schnelle SRAM-Bausteine sind teuer
- Schnelle SRAM-Bausteine nehmen viel Platz in Anspruch

→ In Größenordnung von mehreren GB Speicher schwer zu realisieren

Speicher
Cache Hit





Wartezyklen

Cache ist aus SRAM-Bausteinen aufgebaut

- Wenig Speicher, im Vergleich zum Hauptspeicher
- Schneller Speicher, im Vergleich zum Hauptspeicher

Ein Cache enthält nur kleine Datenmengen (*Cacheblocks*) aus dem Hauptspeicher

- Problem: Welche Daten sollen aus dem Hauptspeicher geladen werden?

Ist im Cache keinen Platz mehr für einen neuen Block, muss zuerst ein Block verdrängt werden – Wie?

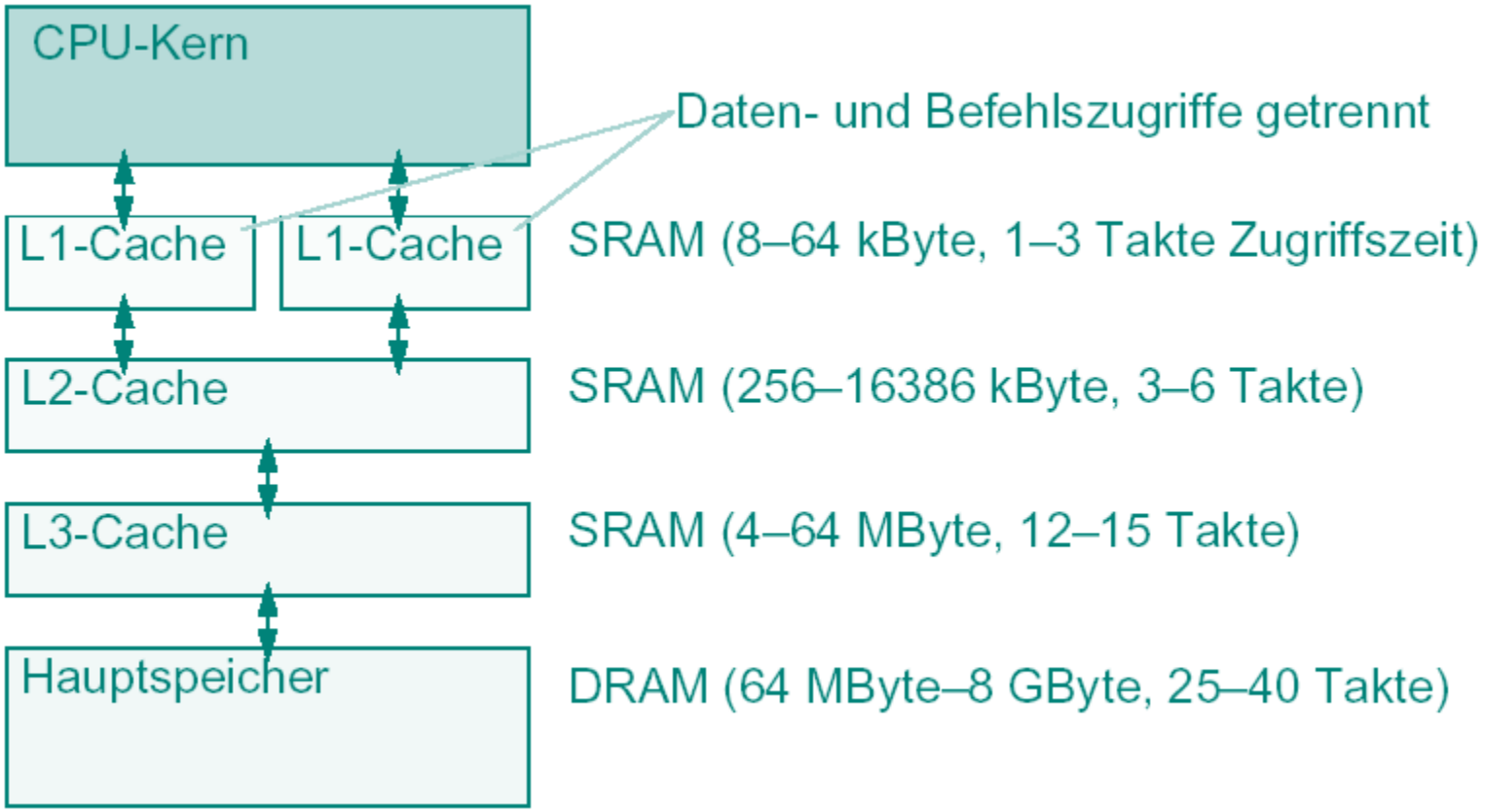
Zeitliche Lokalität

- Es ist, bei entsprechender Programmierung, sehr wahrscheinlich, dass auf eine Speicherzelle nicht nur einmal zugegriffen wird, sondern dass sie zeitlich nah mehrmals gelesen/geschrieben wird

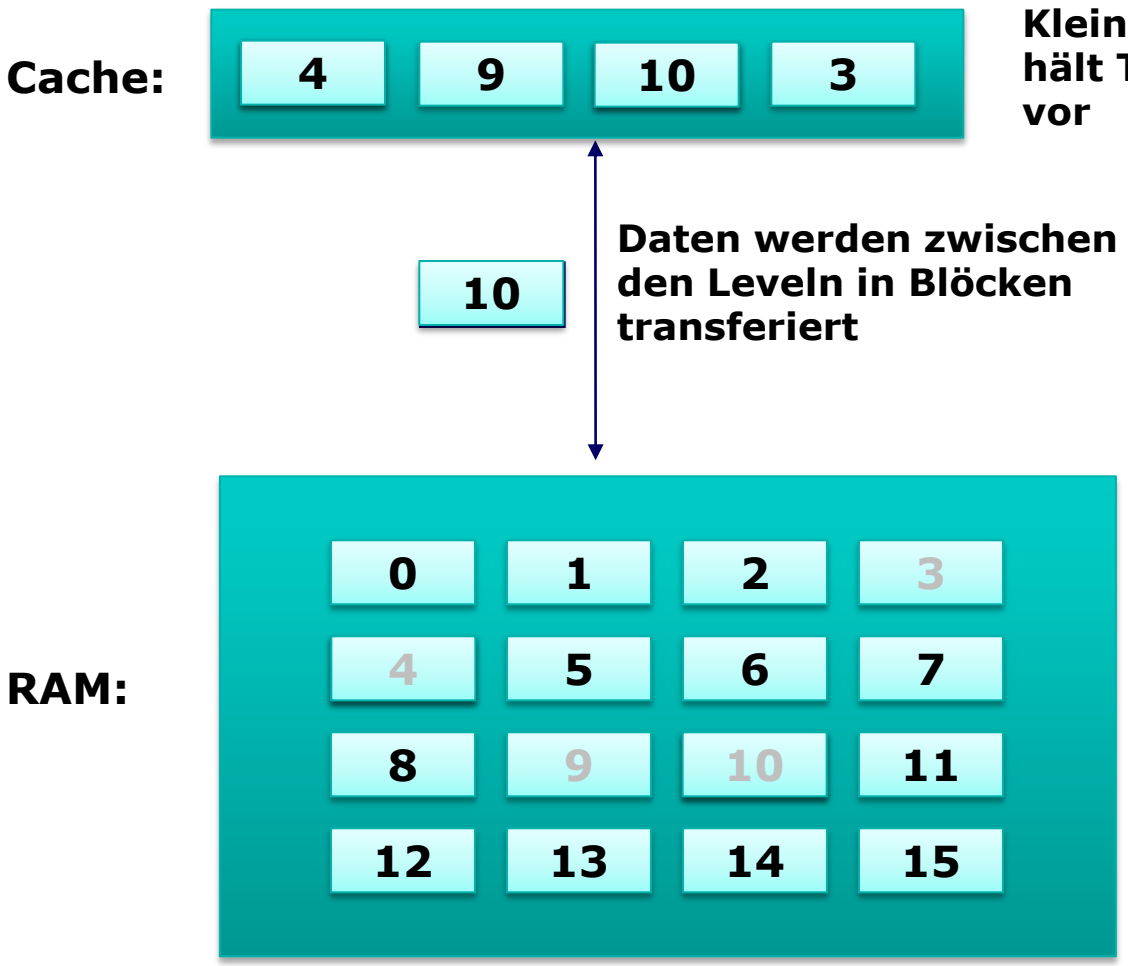
Örtliche Lokalität

- Es ist, bei entsprechender Programmierung, sehr wahrscheinlich, dass nach einem Zugriff auf eine Speicherzelle A zeitlich nah auch ein Zugriff auf eine Speicherzelle B = A + k, für kleine k, geschieht

Speicher
Cache-Hierarchie



Speicher
Cache-Idee



Kleinerer, schneller Speicher – hält Teile des Speichers als Kopie vor

Großer, langsamer Haupt-speicher - ist in Blöcken unterteilt

Speicher
Aufbau eines Caches

Cache ist ein Assoziativspeicher für Blöcke des Hauptspeichers

- Tag (*Key/Schlüssel*) entspricht der Speicheradresse

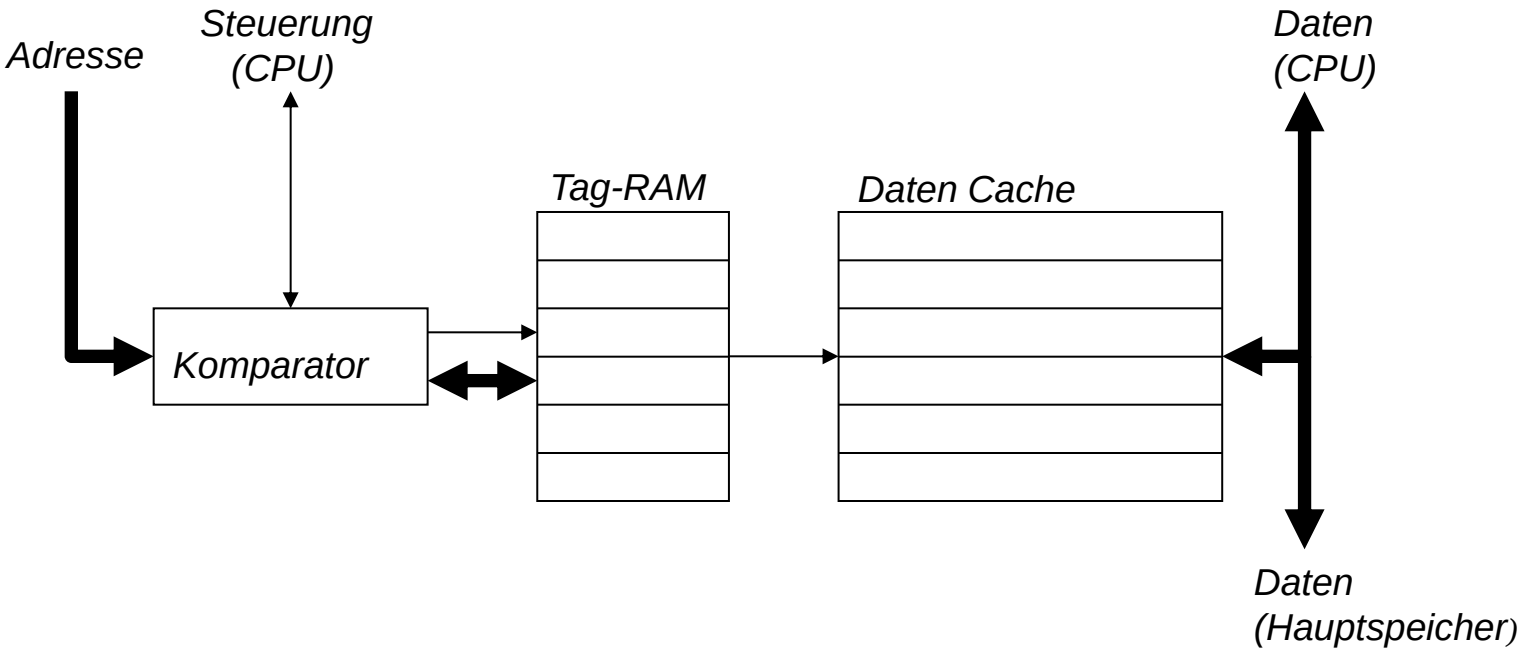
Ungültige Einträge werden nicht aus dem Cache gelöscht

- Markierung ungültiger Blocks durch Löschen des Valid-Flags (V-Flag)
- Löschen erst durch überschreiben

Aufbau eines Cacheeintrags

- Die Daten selbst
- Tag (oberer Teil der Adresse, der einen Block adressiert)
- Statusbits, u.a.
  - > Valid-Flag (gültige Daten)
  - > Modified bzw. Dirty (wurde geändert, wichtig für Write-Back)

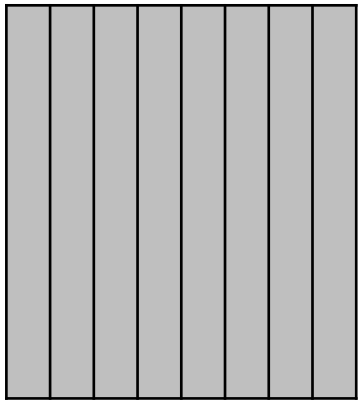
Speicher
Aufbau eines Caches



Blöcke und Sets

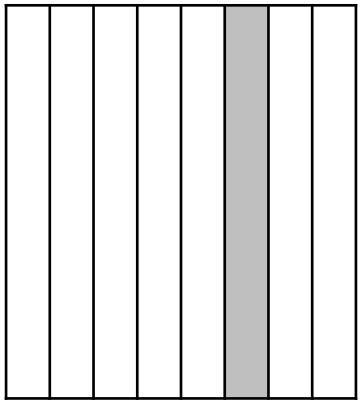
- Blöcke eines Caches werden in sog. Sets zusammengefasst
- *m* sei die Gesamtanzahl der Cacheblöcke
- *n* sei die Anzahl der Blöcke pro Set, die so genannte *Assoziativität*

Es gibt drei Arten von Caches



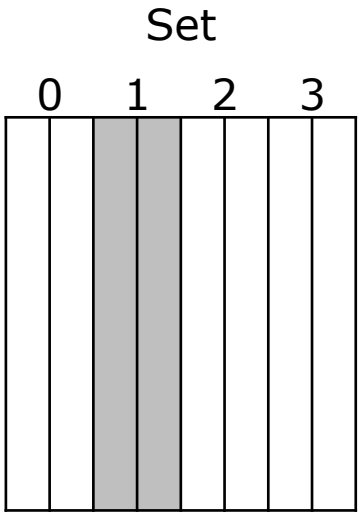
0 1 2 3 4 5 6 7

fully associative



0 1 2 3 4 5 6 7

direct mapped



0 1 2 3 4 5 6 7

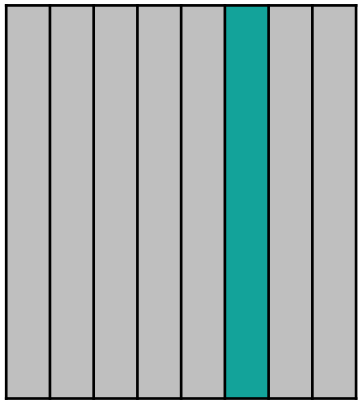
set associative

Die Stelle, an der ein Block im Cache eingefügt wird, ergibt sich direkt aus der Adresse im Hauptspeicher

- Berechnung durch Modulo-Operation

Beispiel

- Cache besteht aus 8 Blöcken
- Speicherblock 13 soll in den Cache aufgenommen werden
- Cacheblock: 13 mod 8 = 5

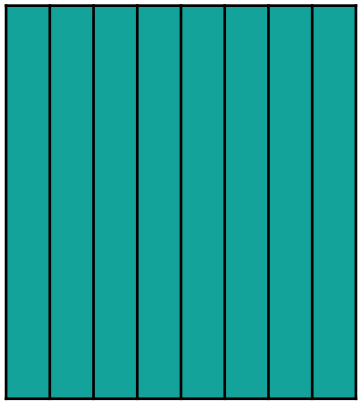


Speicher
Fully-associative Cache

Ein Block kann in jeder Stelle in den Cache eingefügt werden

- Beim Einfügen in den Cache wird die Adresse des Blocks in das Tag-RAM geschrieben

Bei der Überprüfung, ob ein Block im Cache liegt, wird das Tag-RAM durchsucht, ob es die entsprechende Speicheradresse enthält



Mehrere Fully-associative Caches arbeiten parallel

- **Der Fully-associative Cache, in den ein Block eingefügt wird, wird anhand der Speicheradresse bestimmt**
  > Berechnung durch Modulo-Operation

Beispiel

- Es gibt 8 Fully-associative Caches zu je 10 Blöcken
- Speicherblock 13 soll in den Cache aufgenommen werden
- Bestimmung des Fully-associative Caches: 13 mod 8 = 5
  > Der fünfte Cache wird verwendet
- Einfügen des Blockes in diesen Cache wie bei Fully-associative Caches üblich
- Tag-RAM

Findet ein Lesezugriff auf Speicherzelle A statt...

- **... wird zunächst geprüft, ob Speicherzelle A bereits im Cache liegt**
  1. A liegt im Cache (***cache hit***)
     > Der Datensatz kann direkt aus dem Cache gelesen werden
  2. A liegt nicht im Cache (***cache miss***)
     > Der Datensatz muss aus dem Hauptspeicher in den Cache geladen werden
     > Aus dem Cache wird der Datensatz in die CPU geladen

Drei Arten von Cache Misses

- ***Compulsory (zwangsweise, zwangsläufig):***
  - > erstmaliger Zugriff auf eine Adresse, dazugehörigen Daten noch nicht im Cache
  - > Mögliche Lösung: Pre-Fetch Einheiten laden selbständig Cache
- ***Capacity:***
  - > Cache ist zu klein, Daten waren im Cache vorrätig, wurden aber wieder aus dem Cache entfernt
  - > Mögliche Lösung: größerer Cache
- ***Conflict:***
  - > in einem Set ist nicht mehr genug Platz und Block wird entfernt, obwohl Platz in anderen Sätzen wäre
  - > erneuter Zugriff auf entfernten Block ist ein „Conflict Miss"
  - > Mögliche Lösung: Erhöhung der Cacheblocks pro Set – also eine Erhöhung der Assoziativität.

Findet ein Schreibzugriff auf Speicherzelle A statt…

- **… wird zunächst geprüft, ob Speicherzelle A bereits im Cache liegt**
  1. A liegt im Cache (***cache hit***)
     > Der Datensatz wird im Cache aktualisiert
     > (Der Datensatz wird im Hauptspeicher aktualisiert)
  2. A liegt nicht im Cache (***cache miss***)
     > Der Datensatz wird direkt in den Hauptspeicher geschrieben
     > Der Inhalt des Caches wird nicht verändert

Es gibt zwei Verfahren zum Aktualisieren von Datensätzen im Cache

- **Write-Through**
  - > Der Datensatz wird im Cache und direkt anschließend auch im Hauptspeicher aktualisiert
  - > Keine Probleme mit der Datenkonsistenz im Hauptspeicher

- **Write-Back**
  - > Der Datensatz wird im Cache aktualisiert und erst in den Hauptspeicher zurückgeschrieben, wenn der entsprechende Cacheblock aus dem Cache verdrängt wird
  - > Niedrige Belastung der Systembusse und keine Wartezyklen

Speicher
Verdrängungsstrategien

- **FIFO (First In, First Out)**
- Der Block, welcher chronologisch gesehen zuerst in den Cache geladen wurde, wird aus dem Cache verdrängt

- **LRU (Least Recently Used)**
- Der Block, auf den am längste nicht mehr zugegriffen wurde, wird aus dem Cache verdrängt

- **Optimale Ersetzungsstrategie**
- Der Block, der chronologisch gesehen am spätesten wieder gebraucht wird, wird aus dem Cache verdrängt
- Nachteil: Diese Information steht meistens nicht zur Verfügung

- **Varianten der genannten Strategien**

1.3 – Speicherverwaltung

Speicherverwaltung
Übersicht

Die Speicherverwaltung eines Betriebssystems ermöglicht den Prozessen einen Zugriff auf den Arbeitsspeicher des Computers

- Der Zugriff sollte effizient und komfortabel sein

Es wird zwischen realer / direkter und virtueller Speicherverwaltung unterschieden

- Welche von beiden Varianten verwendet wird, hängt vom Einsatzgebiet des Systems ab

Speicherzugriffe unterschiedlicher Prozesse

- Der Zugriff auf den Speicherbereich eines anderen Prozesses muss durch das Betriebssystem explizit unterbunden werden

Speicherverwaltung
Reale Speicherverwaltung

Der Arbeitsspeicher wird aus den Prozessen heraus direkt adressiert

- Die Summe des von den gleichzeitig geladenen Prozessen verbrauchten Arbeitsspeichers kann nicht größer sein, als der physikalisch vorhandene Speicher
- Nutzung von *Swapping*, damit mehr Prozesse „parallel" betrieben werden können (Summe der Prozessspeicher kann über dem verfügbaren Hauptspeicher liegen)

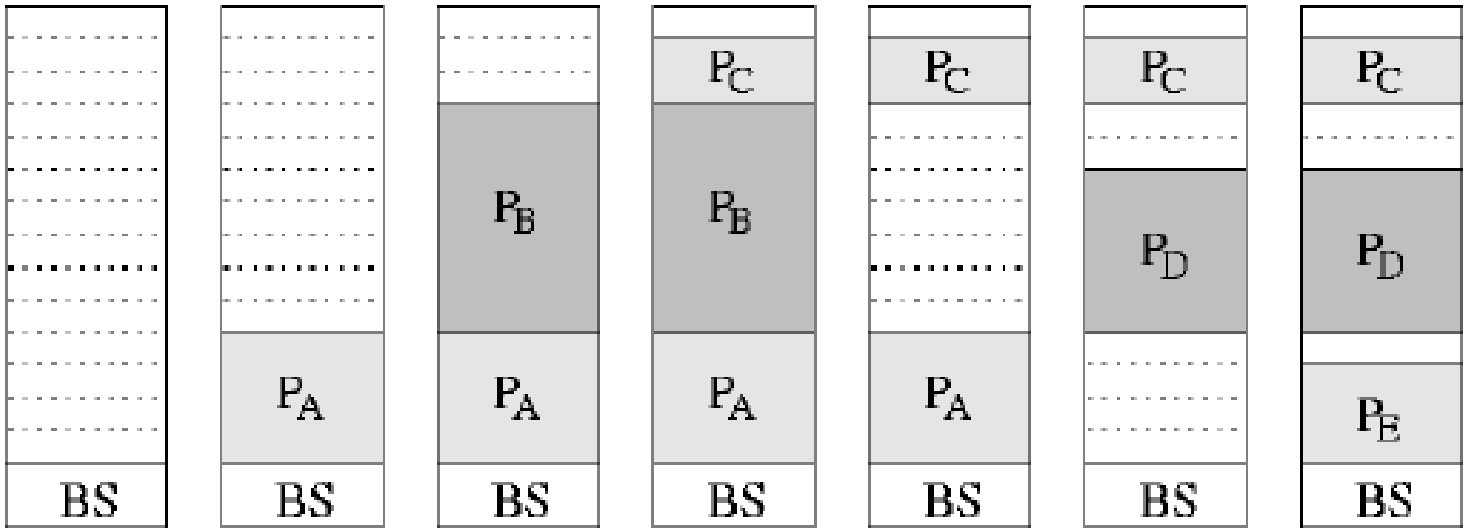
Probleme realer Speicherverwaltung

- Fragmentierung des Speichers
- Suche nach freien Speicherblöcken mittels Belegungstabelle sehr aufwändig

Speicherverwaltung
Fragmentierung des Speichers

Beim Ersetzen von Speicherblöcken können viele kleine, freie Bereiche entstehen

- Neue Prozesse finden keine ausreichend großen zusammenhängenden Bereiche mehr



Nutzung des Speicherplatzes

- Es muss Platz für das gesamte Programm und die Daten gefunden werden, obwohl diese wahrscheinlich nicht alle gleichzeitig benötigt werden

Beschränkung des Speicherplatzes

- Es kann insgesamt nicht mehr Speicher genutzt werden, als physikalisch vorhanden

Belegung des Speichers

- Die Anforderung zusammenhängende Speicherblöcke für Prozesse zu finden, verschärft das Problem der Fragmentierung

Virtuelle Speicherverwaltung behebt die Nachteile realer Speicherverwaltung

Jedem Prozess wird ein (scheinbar) zusammenhängender Speicherbereich der Größe *n* zur Verfügung gestellt

- Tatsächlich besteht der Speicher des Prozesses aus nicht zwangsläufig zusammenhängenden *virtuellen Pages*
- Der Prozess kann seinen Speicher mit den *virtuellen Adressen* von *0* bis (*n-1*) adressieren
- Das Betriebssystem bildet diese auf die realen Adressen ab und sorgt zugleich dafür, dass diese vor dem Zugriff dann auch verfügbar sind
- Pages, auf die aktuell nicht zugegriffen werden, müssen auch nicht im realen Hauptspeicher liegen

Die Gesamtheit aller virtuellen Adressen wird als *virtueller Adressraum* bezeichnet

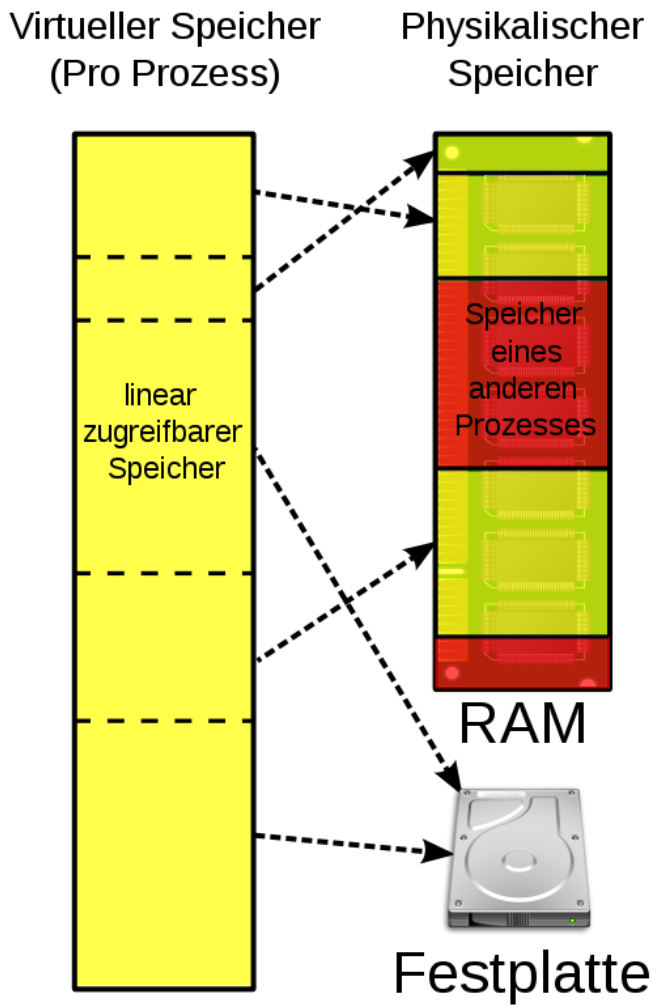
Virtuelle Pages werden rechnerintern auf physikalisch vorhandene Pages gleicher Größe abgebildet

- Die physikalischen Pages können irgendwo im Arbeitsspeicher oder, sofern aktuell nicht im Zugriff, sogar in einer Auslagerungsdatei auf der Festplatte liegen

- **Beim Zugriff eines Prozesses auf eine virtuelle Speicheradresse in seinem Adressraum …**
- … wird zunächst die zu dieser Adresse gehörige virtuelle Page ermittelt
- Anschließend wird die zu dieser virtuellen Page gehörige physikalische Page im Arbeitsspeicher ermittelt
  - > Die Zuordnung von virtuellen und physikalischen Pages wird in der sogenannten *Pagetable* gespeichert
  - > Liegt diese nicht im Speicher, so sorgt ein Page-Fault-Interrupt dafür, dass diese eingelagert wird
- Die relative Speicheradresse innerhalb einer Page ist in beiden Pages dieselbe

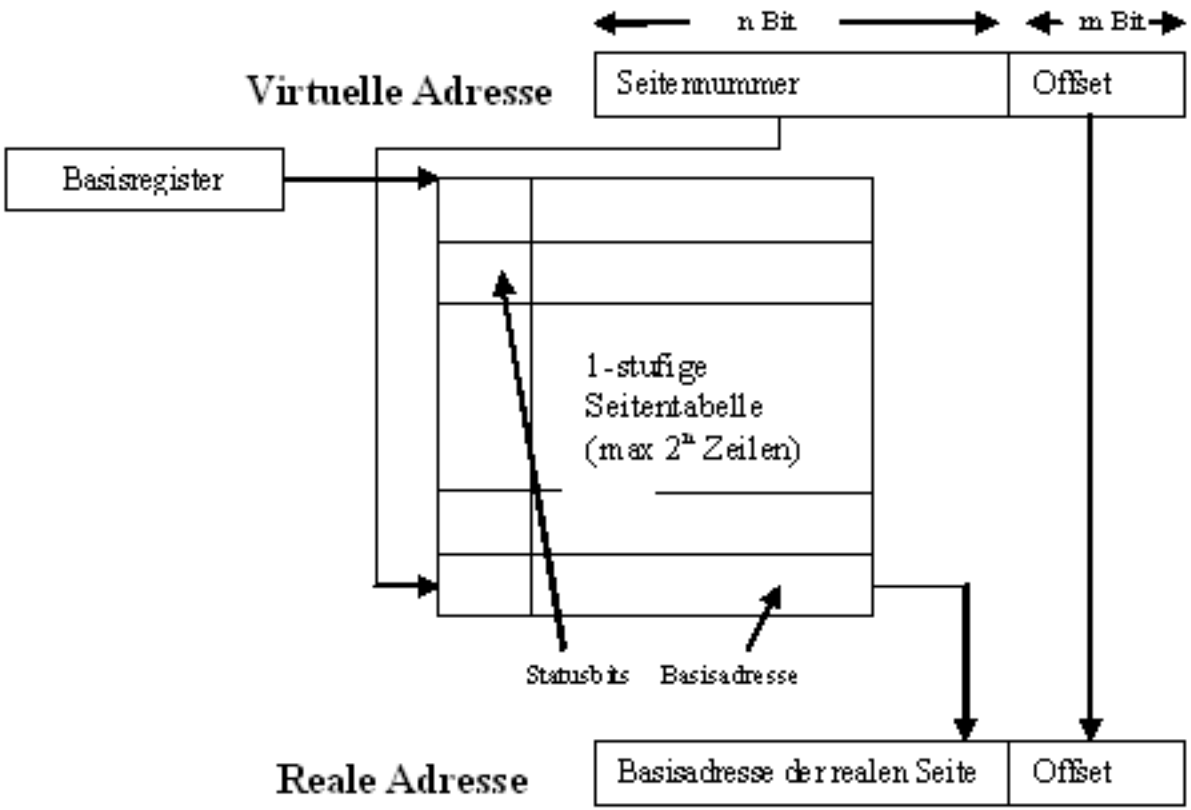
Wie groß ist eine Page?

Speicherverwaltung
Pages



Quelle: http://de.wikipedia.org/wiki/Virtuelle_Speicherverwaltung

Berechnung der physikalischen Speicheradresse



Quelle: http://de.wikipedia.org/wiki/Paging

Berechnung der physikalischen Speicheradresse: Beispiel Adresslänge 16 Bit, 8 Bit Offset (low) und 8 Bit Seitennummer (high)

Seitentabelle:

Eintrag	Gültig	Seitenrahmen
0	Nein	-
1	Ja	0x17
2	Ja	0x20
3	Ja	0x08
4	Nein	-
5	Ja	0x10

Zu übersetzen:

virtuelle Adr.	physische Adr.
0x083A	ungültig (Seite 8 ex. nicht)
0x01FF	0x17FF (Seite 1, Rahmen 0x17)
0x0505	0x1005 (Seite 5, Rahmen 0x10)
0x043A	ungültig (Seite 4 ungültig)

Quelle: http://de.wikipedia.org/wiki/Paging

Speicherverwaltung
Paging on Demand

Zu einem Prozess gehörige, jedoch zu einem bestimmten Zeitpunkt nicht genutzte Pages werden bei Engpässen auf den Hintergrundspeicher (z.B. Festplatte) ausgelagert

- Es wird Arbeitsspeicher für andere Prozesse freigegeben

Braucht der Prozess die Page zu einem späteren Zeitpunkt wieder, muss sie wieder in den Arbeitsspeicher geladen werden

- Es muss eine andere Page aus dem Arbeitsspeicher verdrängt werden

Es gibt mehrere Möglichkeiten, zu bestimmen, welche Page aus dem Arbeitsspeicher verdrängt wird

Speicherverwaltung
Verdrängungsstrategien

FIFO (First-In, First-Out)

- Die Page, welche bereits am längsten im Speicher liegt, wird verdrängt. Ineffizient, da das nichts über die Häufigkeit aussagt.

Least recently/frequently used (LRU/LFU)

- Die Page, auf welche am längsten nicht mehr bzw. am seltensten zugegriffen wurde, wird verdrängt.

Unversehrtheit der Speicherseite

- Es werden Pages ausgelagert, welche sich im Arbeitsspeicher nicht geändert haben, sodass die teure Schreiboperation der Aktualisierung im Hintergrundspeicher entfällt

Not recently used (NRU)

- Die Page, die innerhalb eines Zeitintervalls nicht benutzt und nicht modifiziert wurde, wird ausgelagert. Danach die, die entweder nicht benutzt oder nicht modifiziert wurde.

Nachteil all dieser Verfahren

- Keins der genannten Verfahren garantiert, dass eine „gute" Page ausgelagert wird, d.h. eine Page, welche in der Zukunft am längsten nicht mehr zugegriffen wird.
- Diese Information ist meistens nicht verfügbar …

Speicherverwaltung
Paging on Demand – Beispiel

FIFO-Strategie

- Seitenanforderungen: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Referenzfolge		1	2	3	4	1	2	5	1	2	3	4	5
Arbeitsspeicher	Page 1	1	1	1	4	4	4	5	5	5	5	5	5
	Page 2		2	2	2	1	1	1	1	1	3	3	3
	Page 3			3	3	3	2	2	2	2	2	4	4
Kontrollzustand (wie lange im Speicher in Zyklen)	Page 1	0	1	2	0	1	2	0	1	2	3	4	5
	Page 2	-	0	1	2	0	1	2	3	4	0	1	2
	Page 3	-	-	0	1	2	0	1	2	3	4	0	1

- Anzahl Einlagerungen (Page laden): **9**

LRU-Strategie

- Seitenanforderungen: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Referenzfolge		1	2	3	4	1	2	5	1	2	3	4	5
Arbeitsspeicher	Page 1	1	1	1	4	4	4	5	5	5	3	3	3
	Page 2		2	2	2	1	1	1	1	1	1	4	4
	Page 3			3	3	3	2	2	2	2	2	2	5
Kontrollzustand (wie lange nicht drauf zugegriffen)	Page 1	0	1	2	0	1	2	0	1	2	0	1	2
	Page 2	-	0	1	2	0	1	2	0	1	2	0	1
	Page 3	-	-	0	1	2	0	1	2	0	1	2	0

- Anzahl Einlagerungen (Page laden): **10**

Speicherverwaltung
Paging on Demand – Beispiel

„Future"-Strategie

- Seitenanforderungen: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Referenzfolge		1	2	3	4	1	2	5	1	2	3	4	5
Arbeitsspeicher	Page 1	1	1	1	1	1	1	1	1	1	3	4	4
	Page 2		2	2	2	2	2	2	2	2	2	2	2
	Page 3			3	4	4	4	5	5	5	5	5	5
Kontrollzustand (in wie viel Zyklen wird zugegriffen)	Page 1	4	3	2	1	3	2	1	-	-	-	-	-
	Page 2	-	4	3	2	1	3	2	1	-	-	-	-
	Page 3	-	-	7	7	6	5	5	4	3	2	1	-

- Anzahl Einlagerungen (Page laden): **7**

Speicherverwaltung
Pageflattering

Wenn einem Prozess nicht genügend Pages zur Verfügung stehen, kann es passieren, dass sehr oft Pages nachgeladen bzw. ersetzt werden müssen

- Der Prozess verbringt mehr Zeit mit dem Warten auf den Speicher, als mit der eigentlichen Ausführung

Ursachen

- Prozess ohne Lokalität: Random Access auf große Speichermengen
- Zu viele Prozesse
- Schlechter Ersetzungsstrategie

Lösung

- Zuteilung einer genügend großen Anzahl von Pages (nicht immer möglich)
- Begrenzung der Prozessanzahl
- Codeoptimierung, sodass der Prozess lokaler arbeitet

Speicherverwaltung
Thrashing

Der Durchsatz des Gesamtsystems leidet, ist es doch in erheblichem Maße mit Paging beschäftigt

- Fehlende Lokalität stört den Gesamtdurchsatz

Ursachen

- Prozess ohne Lokalität: Random Access auf große Speichermengen
- Zu viele Prozesse
- Schlechter Ersetzungsstrategie
- Zu wenig Speicher

Lösung

- Zuteilung einer genügend großen Anzahl von Pages (nicht immer möglich)
- Begrenzung der Prozessanzahl
- Codeoptimierung, sodass der Prozess lokaler arbeitet
- Verbesserung der Speicherausstattung
- Direct Memory Access

Lokale Ersetzungsstrategie

- Ein Prozess ersetzt immer nur seine eigenen Pages
  - > Statische Zuteilung von Pages an Prozesse
  - > Nachladen / Ersetzen von Pages liegt in der Verantwortung der Prozesse

Globale Ersetzungsstrategie

- Ein Prozess ersetzt ggf. auch Pages anderer Prozesse
  - > Dynamisches Verhalten der Prozesse kann berücksichtigt werden
  - > Im Schnitt bessere Effizienz, da ungenutzte Seiten von anderen Prozessen verwendet werden können

Speicherverwaltung
Speicherbereinigung

Wird in einem Prozess Speicher dynamisch (zur Laufzeit) allokiert, muss dieser auch irgendwann wieder freigegeben werden, da dem Prozess sonst irgendwann kein Speicher mehr zur Verfügung steht

- Der Prozess kann dies selbst tun (z.B. in C/C++ mittels *free* oder *delete*)
  - > Volle Kontrolle über belegten Speicher, aber kompliziert bzw. fehleranfällig)
  - > Es existieren Konzepte, um Speicherlecks zu detektieren und zu verhindern, z.B.
    - *new/delete* in Klassen kapseln, garantiert Freigabe via Destruktoren
    - *auto_ptr/unique_ptr Instanz* „besitzt" Speicher und gibt automatisch frei, wenn Ptr-Instanz freigegeben wird
    - Markieren von Speicherbereichen durch besondere Muster
- Es kann eine automatische Speicherbereinigung (engl. *Garbage Collection*) verwendet werden (z.B. in Java)
  - > Bequem, entzieht den Programmierer die Kontrolle über den Speicher

1.4 – Betriebssysteme (Prozesse, Scheduling)

Prozess

- Abstraktion eines sich in Ausführung befindlichen Programms
- Besteht aus Programmbefehlen, welche unter der Kontrolle des Betriebssystems ablaufen

Prozessverwaltung (*Task Management, Task Scheduling*)

- Steuert die Zuteilung von Betriebsmitteln (CPU, Speicher, I/O, …) an die Prozesse

Ein Prozess besteht aus …

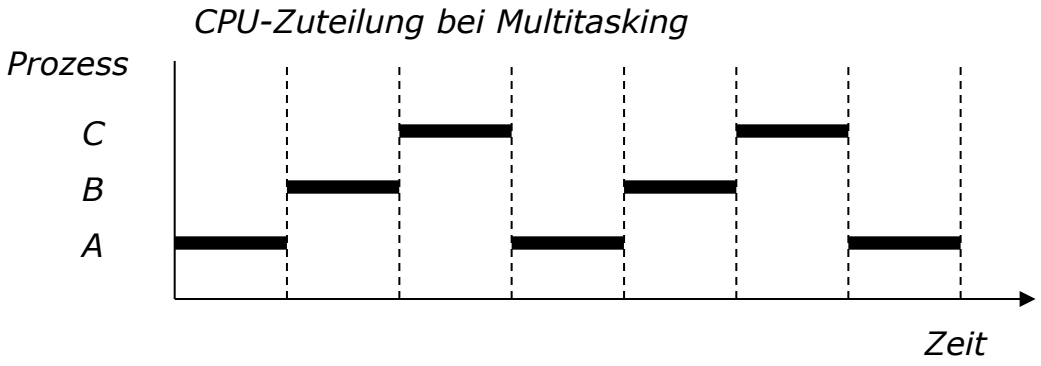
- Dem Programmcode, des in Ausführung befindlichen Programms
- Der Ausführungsumgebung (Kontext)

Der Kontext beinhaltet …

- Den privaten Adressraum des Programms inkl. globaler Variablen
- Dem Status der verwendeten Register (diese behalten ihre Werte)
- Statusinformationen zu externen Einheiten (Dateien, Sockets, …)
- Abhängige Prozesse (Kindsprozesse)

Multitasking

- Der Prozessor kann zwischen mehreren Prozessen hin und her geschaltet werden
- Im Allgemeinen wird sog. **preemptives Multitasking** verwendet
  - > Das Betriebssystem entscheidet, wann welcher Prozess zur Ausführung kommt und wann zwischen den Prozessen gewechselt wird (via Interrupts)
  - > Jeder Prozess wird für einige Millisekunden (*Zeitscheibe, timeslice*) ausgeführt, spätestens mit Ablauf der Zeitscheibe wird er unterbrochen
  - > Der Benutzer erhält dadurch den Eindruck von Parallelität



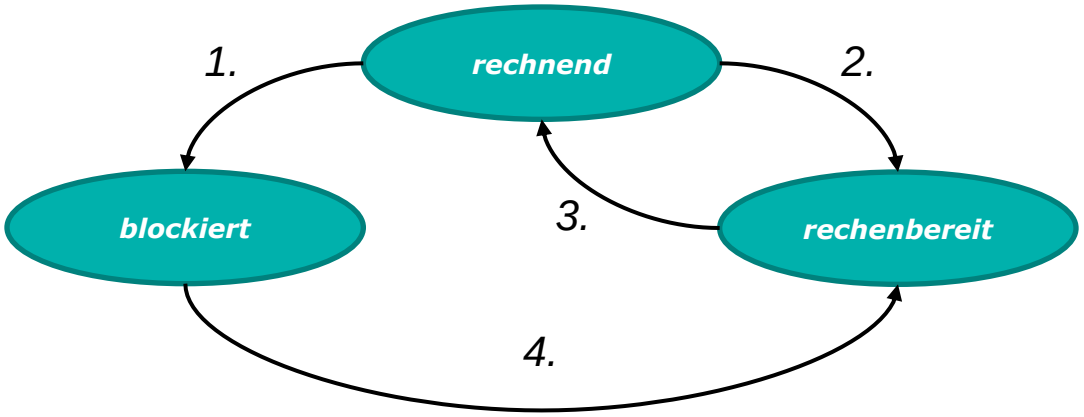
Im Gegensatz zum preemptiven Multitasking bestimmt ein Prozess selbst, wann er den Prozessor an andere Prozesse abgibt

Nachteile

- Ein einzelner Prozess kann bei bestimmten Fehlern (z.B. Endlosschleifen) das gesamte System zum Absturz bringen
- Selbst das Betriebssystem kann nicht mehr rechnen, wenn ein Prozess den Prozessor nicht wieder freigibt

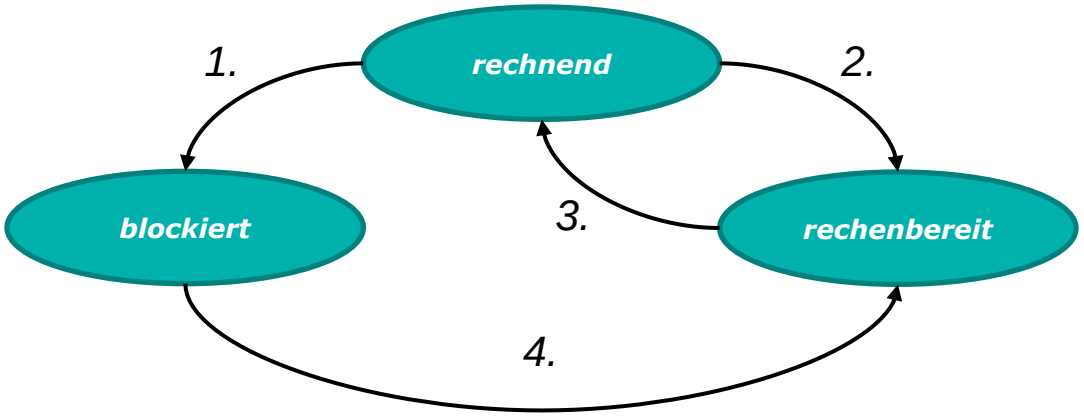
Ein Prozess kann sich in folgenden Zuständen befinden

- ***rechnend*** Der Prozessor ist dem Prozess zugeteilt
- ***blockiert*** Der Prozess kann nicht ausgeführt werden, bis ein externes Ereignis auftritt (z.B. I/O)
- ***rechenbereit*** Der Prozess ist ausführbar, aber der Prozessor ist einem anderen Prozess zugeteilt



Der Prozess-Scheduler organisiert die Übergänge der Prozesse zwischen den einzelnen Zuständen

1. Der Prozess wartet z.B. auf Eingaben des Benutzers
2. Die Zeitscheibe des Prozesses ist abgelaufen oder ein höher priorisierter Prozess muss ausgeführt werden
3. Der Prozess bekommt eine neue Zeitscheibe zugeteilt
4. Das Ereignis, auf welches ein Prozess nach 1. gewartet hat, ist eingetreten



Ein guter Scheduling-Algorithmus muss einige Anforderungen erfüllen

- ***Fairness*** Jeder Prozess erhält einen gerechten Anteil der CPU-Zeit (Fair-Share)
- ***Effizienz*** Die CPU und andere Ressourcen sind möglichst ausgelastet
- Einhaltung der Systemregeln

Weitere Anforderungen an den Scheduling-Algorithmus hängen vom Einsatzgebiet ab

- Batch-Systeme
- Interaktive Systeme / Dialogsysteme
- Echtzeitsysteme

Zusätzliche Anforderungen an einen Scheduler auf Batch-Systemen

- Möglichst Hohe CPU-Auslastung
  > Es sollte nicht vorkommen, dass die CPU Leertakte hat
- Job-Durchsatz
  > Die Anzahl der bearbeiteten Aufgaben pro Zeiteinheit sollte maximal sein
- Minimale Durchlaufzeit
  > Die Zeit, welche ein Job von Ankunft in der Job-Queue bis zur Fertigstellung des Jobs benötigt, sollte minimal sein

Zusätzliche Anforderungen an einen Scheduler auf Dialogsystemen

- Kurze Antwortzeiten
  - > Der Benutzer sollte nicht das Gefühl haben, auf Reaktionen des Systems, z.B. auf Mausklicks oder Tastatureingaben, warten zu müssen
  - > Prozesse, die Interaktion mit dem Benutzer erfordern, sollte vor anderen Prozessen bevorzugt werden

- Proportionalität
  - > Die Antwortzeit verschiedener Prozesse sollte mit der Benutzererwartung übereinstimmen
  - > Aus Benutzersicht einfache Aufgaben sollten schneller erledigt werden, als aus Benutzersicht schwierige

Zusätzliche Anforderungen an einen Scheduler auf Echtzeitsystemen

- Einhaltung von Zeitfenstern
  > Der Scheduler muss einen Überblick über die Zeitfenster verschiedener Prozesse haben und diesen entsprechend Rechenzeit zuweisen

- Vorhersagbarkeit
  > Das System muss deterministisch reagieren

First-Come, First-Serve (FCFS, FIFO)

- Die Prozesse werden der Reihenfolge ihres Starts nach bearbeitet
- Die Zuteilung des Prozessors an andere Prozesse findet nur statt, wenn ein laufender Prozess zu warten beginnt oder sich beendet

Shortest Job First (SJF)

- Wird auf Batch-Systemen verwendet
- Jobs werden aufsteigend nach geschätzten Ausführungszeit bearbeitet
- Große Jobs bekommen möglicherweise nie Rechenzeit, wenn immer wieder kleinere Jobs gestartet werden

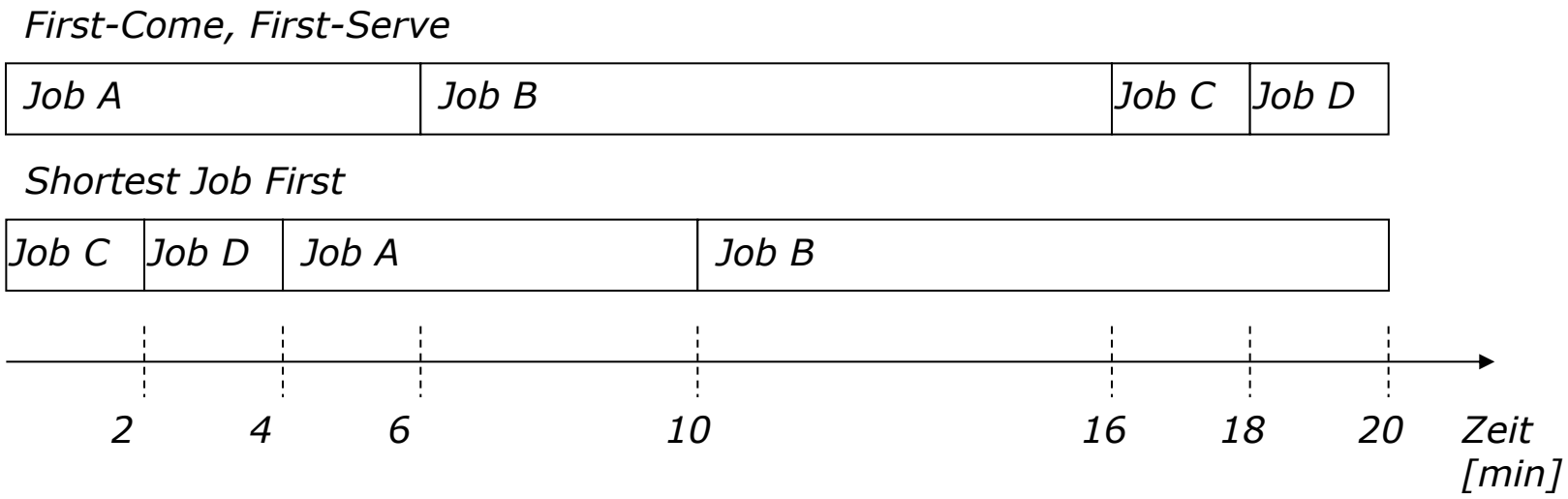
Round-Robin

- Prozessen wird der Reihe nach eine (gleichgroße) Zeitscheibe auf der CPU zugeteilt

Prioritätsscheduling auf Dialogsystemen

Ausführungszeiten

- Job A: 6 Minuten
- Job B: 10 Minuten
- Job C: 2 Minuten
- Job D: 2 Minuten



„First Come, First Serve" ergibt folgende Verweilzeiten:

- Job A: 6 Minuten
- Job B: 16 Minuten
- Job C: 18 Minuten
- Job D: 20 Minuten
- Mittlere Verweilzeit: $\frac{6+16+18+20}{4} = 15$ Minuten

„Shortest Job First" ergibt folgende Verweilzeiten:

- Job A: 10 Minuten
- Job B: 20 Minuten
- Job C: 2 Minuten
- Job D: 4 Minuten
- Mittlere Verweilzeit: $\frac{10+20+2+4}{4} = 9$ Minuten

Jedem Prozess wird eine Priorität zugewiesen

- Zuweisung der Priorität wird vom Betriebssystem gesteuert
- Es gibt dynamische und statische Zuweisung

Es wird immer dem lauffähigen Prozess mit der höchsten Priorität die nächste Zeitscheibe zugeteilt

Neu hinzukommende Prozesse hoher Priorität verdrängen ggf. rechnende Prozesse niedriger Priorität

- Auch der Übergang nach „rechenbereit" eines hoch priorisierten Prozesses verdrängt nieder priorisierte Prozesse

Statische Priorität

- Jeder Prozess erhält bei seinem Start eine feste Priorität
- Es wird immer der Prozess mit der höchsten Priorität ausgeführt
- Gibt es mehrere Prozesse mit gleicher Priorität werden diese im Round-Robin-Verfahren bearbeitet
- Oft in Echtzeitsystemen verwendet

Dynamische Priorität

- Jedem Prozess wird eine Anfangspriorität zugeordnet
- Es rechnet immer der Prozess mit der höchsten Priorität
  - > Die Prioritäten der Prozesse werden dynamisch geändert
  - > Es gibt verschiedene Verfahren dies zu realisieren, z.B. in Abhängigkeit von der gerade genutzten CPU-Zeit

Beispiel für ein dynamisches Scheduling

- Die Prozesse werden anhand ihres bisherigen Ressourcenverbrauchs priorisiert
- z.B. in Windows und Linux (bis Kernel-Version 2.4) verwendet

Es werden mehrere FIFO-Queues benutzt

- Neue Prozesse werden in der höchst priorisierten FIFO-Queue eingefügt
- Maßnahmen auf den Warteschlangen:
    1. Der Prozess beendet sich und verlässt das System
    2. Der Prozess gibt den Prozessor freiwillig ab
        - Der Prozess wird in derselben FIFO-Queue wieder eingefügt
    3. Der Prozess verbraucht seine gesamte Zeitscheibe
        - Der Prozess wird in der nächst niedriger priorisierten FIFO-Queue wieder eingefügt

1.5 – Massenspeicher

Massenspeicher

RAID = "Redundant Array of Independent Disks"

- Ehemals „Redundant Array of Inexpensive Disks"

Dient zur Organisation mehrerer physikalischer Festplatten in einem Verbund

Kann hardware- oder softwareseitig verwaltet werden

Es werden gezielt redundante Daten erzeugt, um beim Ausfall einer Festplatte die Integrität und Funktionalität des Gesamtsystems trotzdem gewährleisten zu können

- RAID0 Sonderfall

Massenspeicher
Aufbau eines RAID

Es werden mindestens 2 Festplatten benötigt

Alle in das RAID eingebundenen Festplatten werden gemeinsam betrieben und zielen auf die Verbesserung minndestens einer Eigenschaft ab

- Erhöhung der Ausfallsicherheit
- Steigerung der Datentransferrate
- Erweiterung der Speicherkapazität
- Möglichkeit des Austauschs von Festplatten im laufenden Betrieb
- Kostenreduktion durch Einsatz mehrerer kostengünstiger Festplatten

Die genau Art des Betriebs wird durch das sog. *RAID-Level* angegeben

Größenänderungen

- Es ist nicht ohne Weiteres möglich die Kapazität eines RAID im Nachhinein durch einfaches Zuschalten einer weiteren Festplatte zu erhöhen

Austausch von Datenträgern

- Muss eine Festplatte im RAID ausgetauscht werden, so muss sie durch eine Platte gleicher oder größerer Kapazität ausgetauscht werden (oftmals identisch)

Defekte Controller (sofern existent)

- Defekte RAID-Controller können Datenverluste erzeugen
- Defekte RAID-Controller können i.A. nur durch identische Controller ersetzt werden

Fehlerhaft produzierte Datenträgerserien

- Kommen mehrere Festplatten einer fehlerhaften Charge im selben RAID zum Einsatz können sich deren Fehler multiplizieren

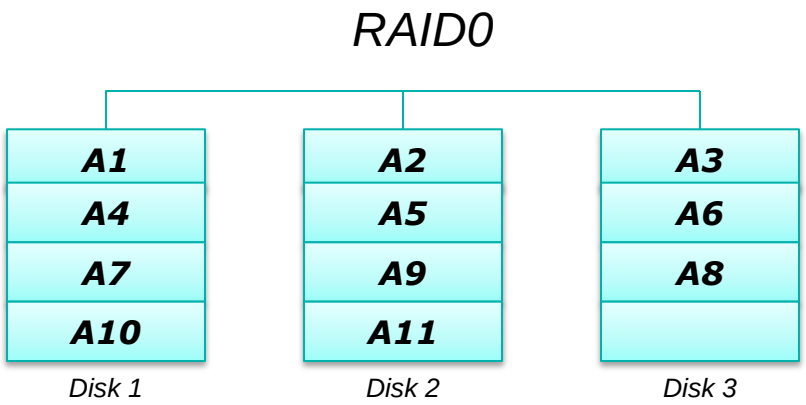
Statistische Fehlerrate bei großen Festplatten

- Die vom Hersteller angegebenen Fehlerraten können sich beim Betrieb der Platte in einem RAID akkumulieren

Kein RAID im herkömmlichen Sinne, da keine redundanten Daten erzeugt werden

Bietet höhere Transferraten durch *Striping*

- Alle beteiligten Festplatten werden in gleich große Blöcke unterteilt
  - > Häufig 64 kB pro Block (große Blöcke)
- Diese Blöcke werden im Reißverschlussverfahren zu einer großen Festplatte angeordnet
- Zugriff auf Blöcke kann in gewissem Maße parallel geschehen



Nachteile

- **Ist eine Festplatte im RAID 0 defekt, so können die Nutzdaten nicht mehr vollständig rekonstruiert werden**
  - > Kleine Dateien können möglicherweise gerettet werden, wenn keine Datenblöcke auf der kaputten Festplatte lagen

- **Der laufende Betrieb muss bei Ausfall einer Festplatte zwangsläufig unterbrochen werden**

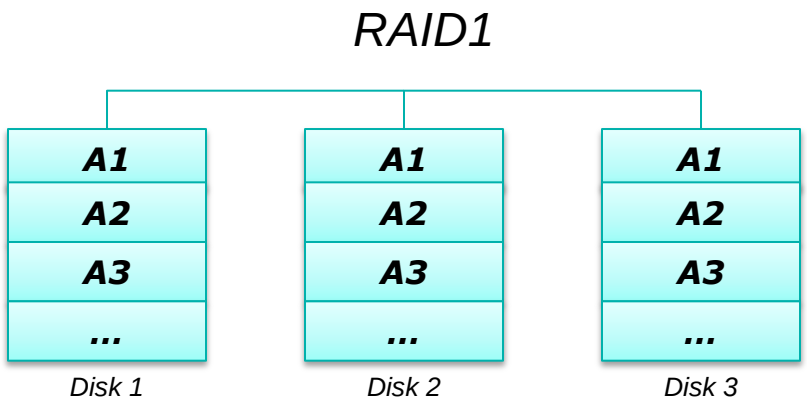
Die Ausfallwahrscheinlichkeit eines RAID 0 mit n Festplatten beträgt

$$1 - (1 - p)^n$$

- Die Ausfallwahrscheinlichkeiten p einer einzelnen Festplatte müssen statistisch unabhängig voneinander sein

Bietet hohe Datenredundanz durch *Mirroring*

- Alle Daten werden identisch auf alle verfügbaren Festplatten geschrieben
- Lesezugriffe können, ähnlich wie bei RAID0, parallel ablaufen



Nachteile von RAID1

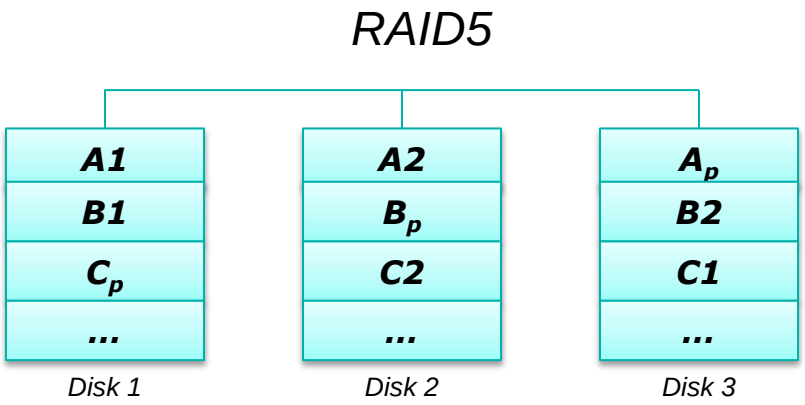
- Das RAID kann maximal so groß sein, wie die kleinste beteiligte Festplatte

RAID1 ist KEIN Ersatz für Datensicherung, da alle Schreib-operationen auch auf allen Festplatten vorgenommen werden

- Versehentlich gelöschte oder geänderte Daten sind trotzdem verloren
- Programmfehler, die fehlerhafte Daten erzeugen haben somit ohne Backup unwiderrufliche Konsequenzen

Bietet höhere Lesetransferraten und hohe Datenredundanz

- Daten werden, wie bei RAID0, in gleich großen Blöcken auf die beteiligten Platten gelegt
- Jeweils eine Platte enthält statt eines Datenblocks einen Paritätsblock
  - > Die Berechnung des Paritätsblocks erfolgt durch XOR-Verknüpfung aller zugehöriger Datenblöcke
  - > Datenintegrität ist bei Ausfall von maximal einer Platte gewährleistet (Rekonstruktion möglich)
- Der Paritätsblock rotiert zyklisch über die Platten (RAID3: Feste Zuordnung)



Datenredundanz, Beispiel (xor-Verknüpft)

- Original bei drei Platten

D1	D2	D3	par
1	1	0	0
0	1	1	0
0	1	0	1

- Wiederherstellung bei Ausfall durch Auswertung der Parität

D1	D2	D3	par
1	?	0	0
?	1	1	0
0	1	?	1

Die Speicherkapazität eines RAID 5 mit n Festplatten errechnet sich wie folgt

- *(n-1) * (Kapazität der kleinsten Festplatte)*

Zweiphasiges Schreibverfahren

1. Schreiben der Daten
2. Schreiben / Aktualisieren des Paritätsblocks

Schreiboperationen können langsamer werden, insbesondere dann, wenn RAID5 in Software realisiert wird

Massenspeicher
RAID 5 - Schreiboperationen

1. *Lese alte Daten*
2. *Schreibe neue Daten*
3. *XOR alte und neue Daten -> "Partial Product"*
4. *Lese alte Paritätsdaten*
5. *Xor alte Paritäts-daten mit dem Partial Product; Sichere das Ergebnis als neue Parität*