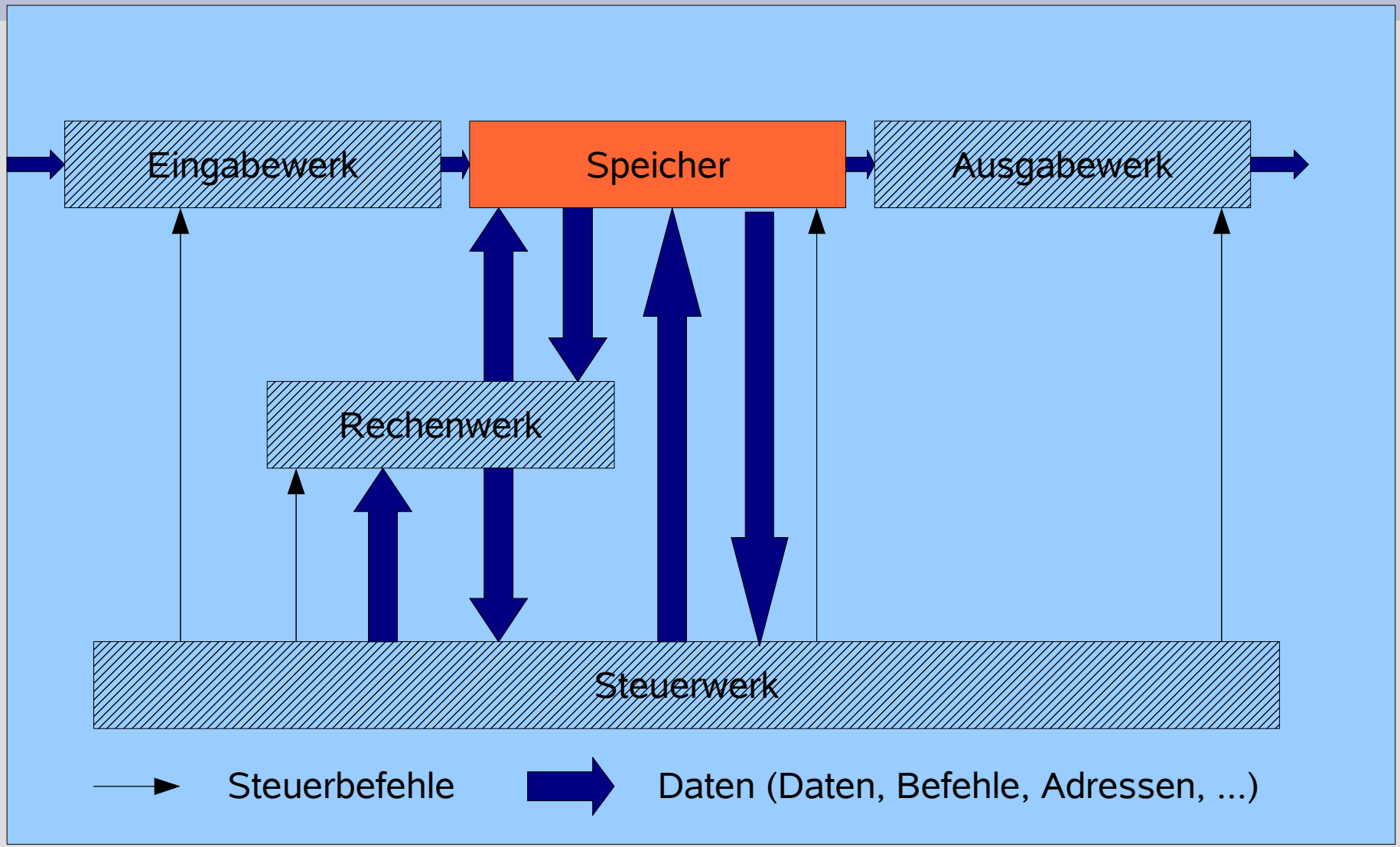
Speicherorganisation

- John von Neumann 1946
  „Ideal wäre ein unendlich großer, unendlich schneller und unendlich billiger Speicher, so dass jedes Wort unmittelbar, d.h. ohne Zeitverlust, zur Verfügung steht ohne dass es etwas kostet.“
- Bisher:
  - Vorlesung 4: Adressierungsmodi
  - „Register sind schneller als Speicher“
  - Systematisierung

v.-Neumann-Architektur



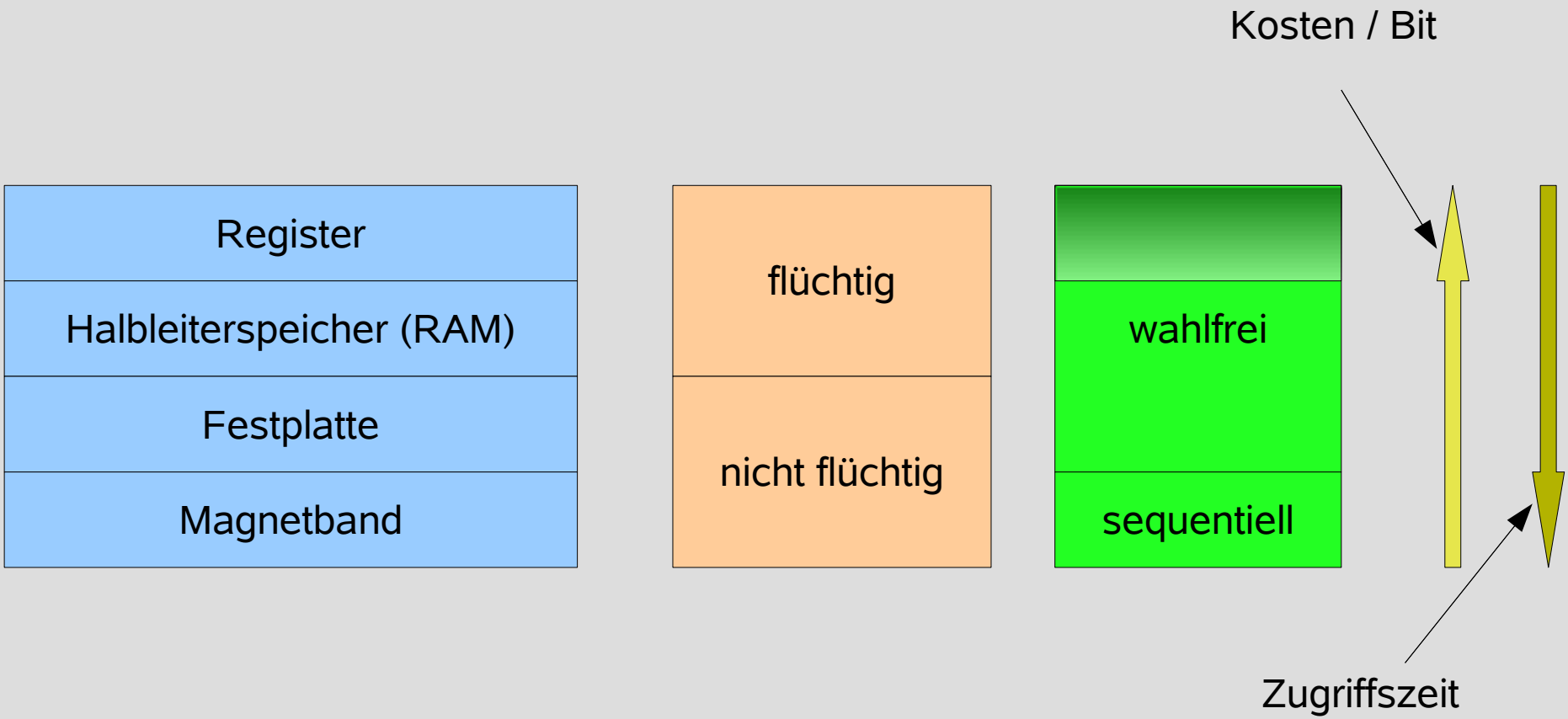
Speicherorganisation (2)

- Klassifikation
  - Flüchtig
    - RAM
  - Nicht flüchtig
    - ROM
  - Zugriffsart: Wahlfrei (Random Access)
    - Festplatte
  - Zugriffsart: sequentiell
    - Magnetband

Speicherorganisation (3)

- Klassifikationsparameter:
  - **Zugriffszeit** (access time): Zeit zwischen der Anforderung eines Lesezugriffs und dem Eintreffen des Inhaltes der adressierten Speicherzelle am Speicherausgang
  - **Kosten** (pro Bit)

Speicherorganisation (4)



Speicherorganisation (5)

- Kosten / Bit sowie Zugriffszeit sind gegenläufig.
- dies führt in der Regel zu einem Optimierungsproblem
- Randbedingung: „Man muss den Speicher gerade noch bezahlen können“
- Anderer Weg: „Verwendung von Registern wie ein Assemblerprogrammierer“.

Beispiel

- Berechnung der Quadratwurzel aus a durch Anwendung der Näherung:

$$x_{m+1} = \frac{1}{2}\left(x_m + \frac{a}{x_m}\right)$$

- x konvergiert gegen die Quadratwurzel
- Abbruchkriterium ist: $|x_m - x_{m+1}| < \text{epsilon}$

Beispiel (2)

- a steht im Speicher, Quadratwurzel aus a soll im Speicher stehen.
- Wo stehen x_m und x_{m+1}?
- ===> Register, außerdem holt sich der **Assemblerprogrammierer** am Anfang den Wert a sowie den Abbruchwert epsilon in ein Register.

Beispiel (3)

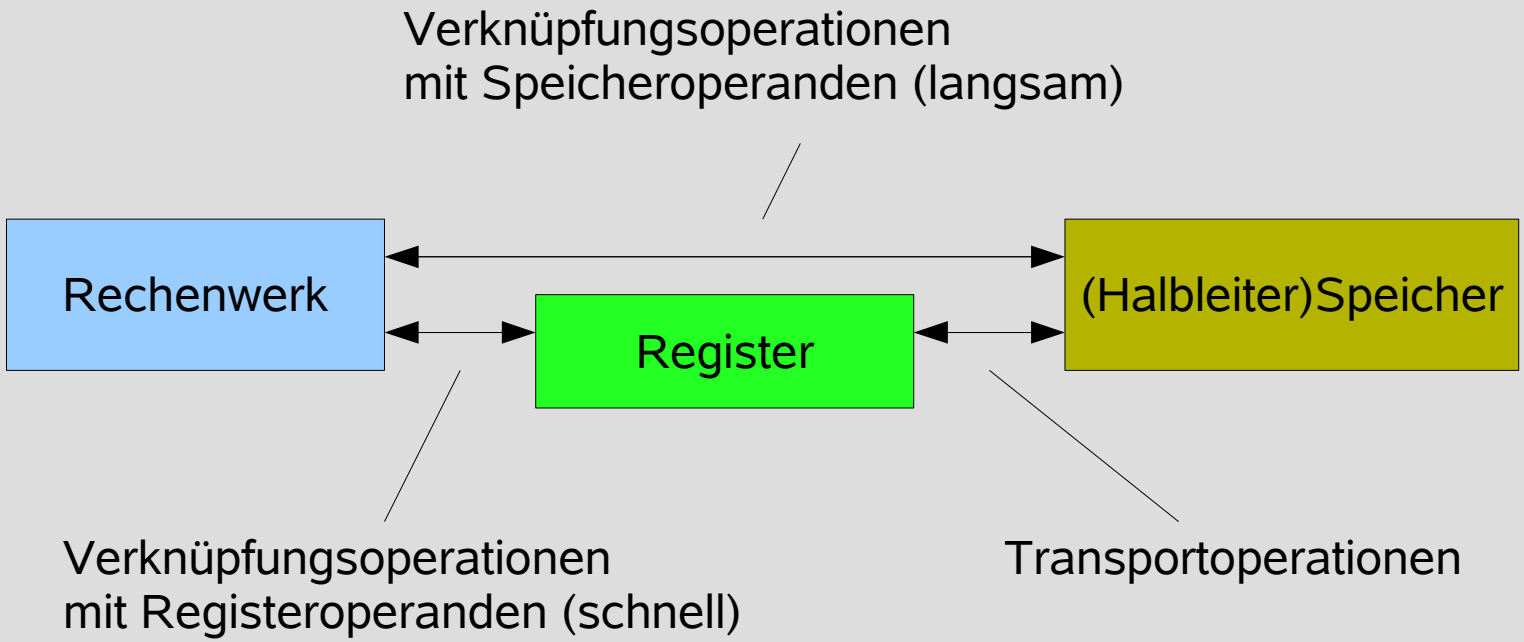
- PASCAL:

```
var a,epsilon,x,x_neu, qwurzel:real;
....
readln(a);
x_neu := 1;
repeat
   x := x_neu;
   x_neu := 0.5 * (x + a/x);
until abs(x-x_neu) < epsilon;
qwurzel := x_neu;
```

Beispiel (4)

- PASCAL-Programmierer hat keinen direkten Zugriff auf Register (zumindest im „Normalfall“).
- Deshalb stehen x, x_neu, a und epsilon im Speicher 
- PASCAL-Programm ist langsamer als Assemblerprogramm

Beispiel (5)


Verknüpfungsoperationen
mit Speicheroperanden (langsam)
Rechenwerk
Register
(Halbleiter)Speicher
Verknüpfungsoperationen
mit Registeroperanden (schnell)
Transportoperationen

Beispiel (6)

- wenn der Befehlssatz des Rechners es dem Programmierer erlaubt zwischen
  - Verknüpfungsoperationen mit Registeroperanden und
  - Verknüpfungsoperationen mit Speicheroperanden

zu wechseln, macht die im Beispiel gezeigte Vorgehensweise Sinn, wenn die Anzahl der Verknüpfungsoperationen sehr viel größer ist als die Anzahl der Transportoperationen.

Lokalitätsprinzip

- wiederholtes Nutzen der selben Daten und Befehle
  - komplexe Berechnungen über gewisse Datenstrukturen erfordern mehrfachen Zugriff auf diese Datenstrukturen
  - Programmschleifen werden mehrfach durchlaufen. Dadurch werden Befehle innerhalb der Schleifenkörper mehrfach zugegriffen.

    Faustregel: Ein Programm verbringt 90 % seiner Abarbeitungszeit in 10 % des gesamten Codes.

Lokalitätsprinzip (2)

- Bereits bekannt:
  - komplexe Datenstrukturen (Felder, Arrays) werden im Speicher dicht hintereinander abgespeichert.
  - sequentiell aufeinanderfolgende Befehle stehen ebenfalls dicht sequentiell im Speicher
- daraus folgt:
  - **mit einiger Wahrscheinlichkeit lässt sich aus dem abgelaufenen Programmteil auf den (in Zukunft) folgenden sowie die folgenden Daten schließen.**

Lokalitätsprinzip (3)

- Dieses *heuristische* Prinzip heißt **Prinzip der Lokalität** oder **Lokalitätsprinzip**.
- 2 Arten von Lokalität:
  - Zeitliche Lokalität:
    - der Zugriff auf eine Speicherzelle lässt es erwarten, dass in naher Zukunft wieder auf sie zugegriffen wird.
  - Räumliche Lokalität
    - Wenn auf eine Speicherzelle zugegriffen wird, erfolgt mit hoher Wahrscheinlichkeit ein Zugriff auf eine benachbarte Speicherzelle.

Speicherhierarchien

- Die Optimierungskriterien
  - Minimale Zugriffszeit
  - Minimale Kosten pro gespeichertes Bit

führen zum Prinzip des

hierarchischen Speichers

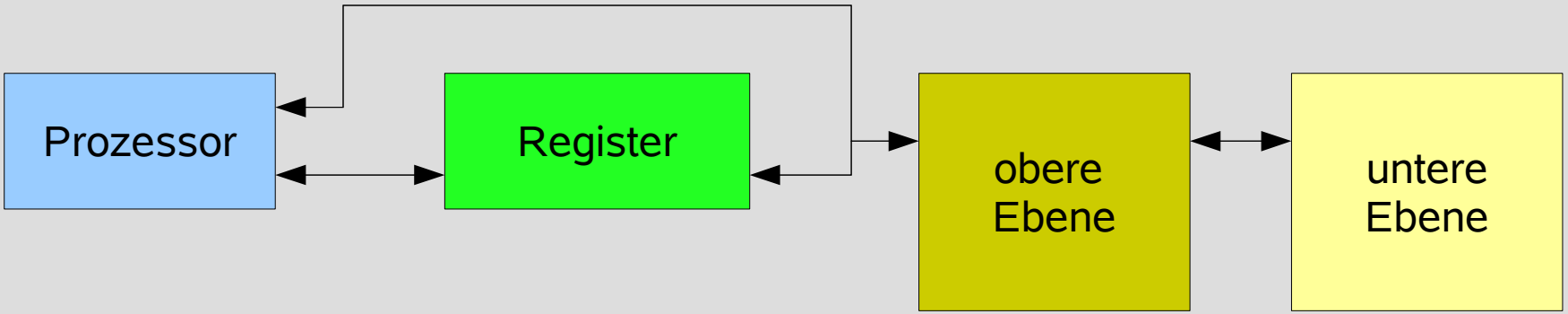
Je teurer der Speicher, umso weniger ist davon im Rechner vorhanden

Speicherhierarchien

Preis pro Bit nimmt zu

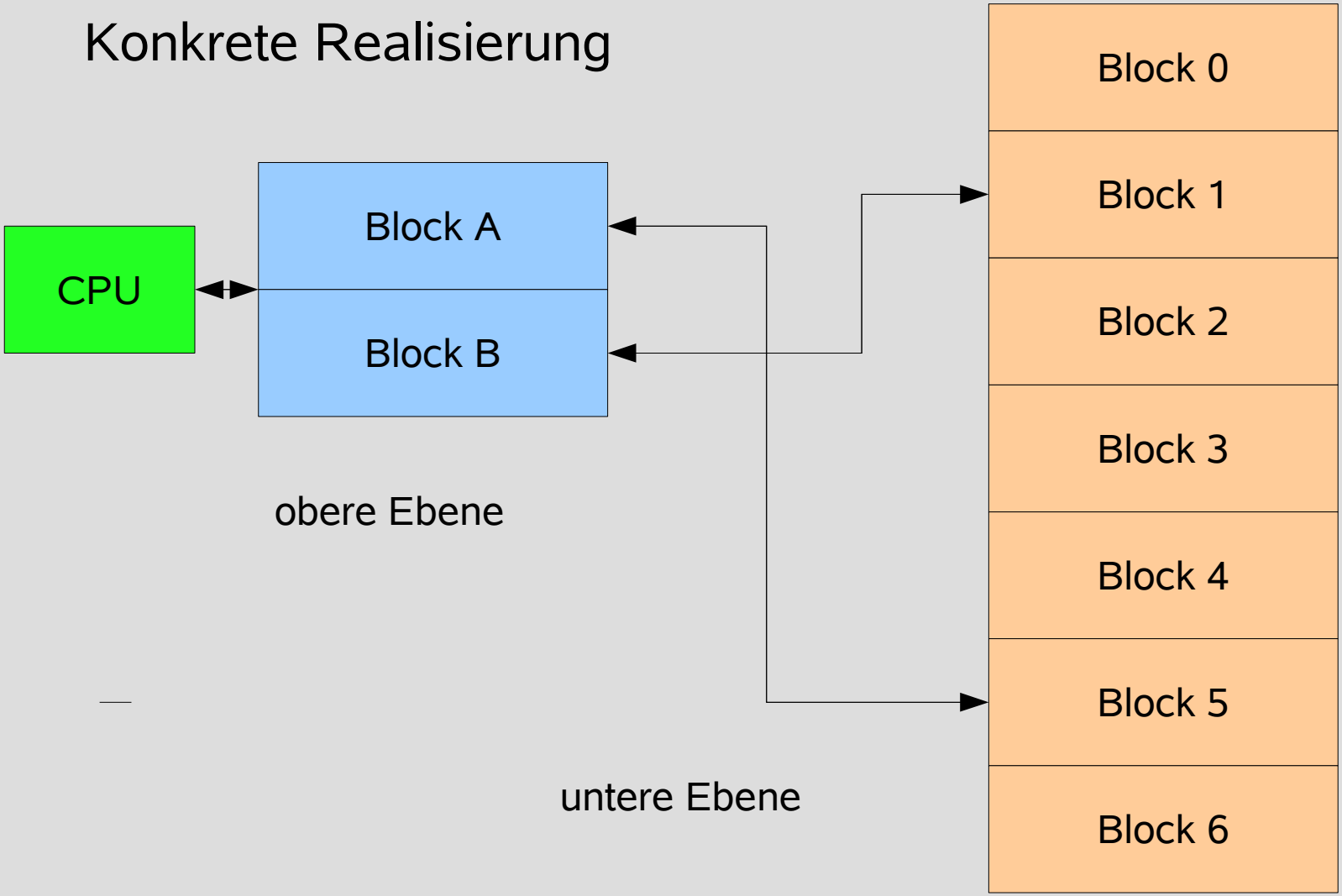
Zugriffszeit nimmt zu

Kapazität nimmt zu, die unterste Ebene realisiert den gesamten Adressraum

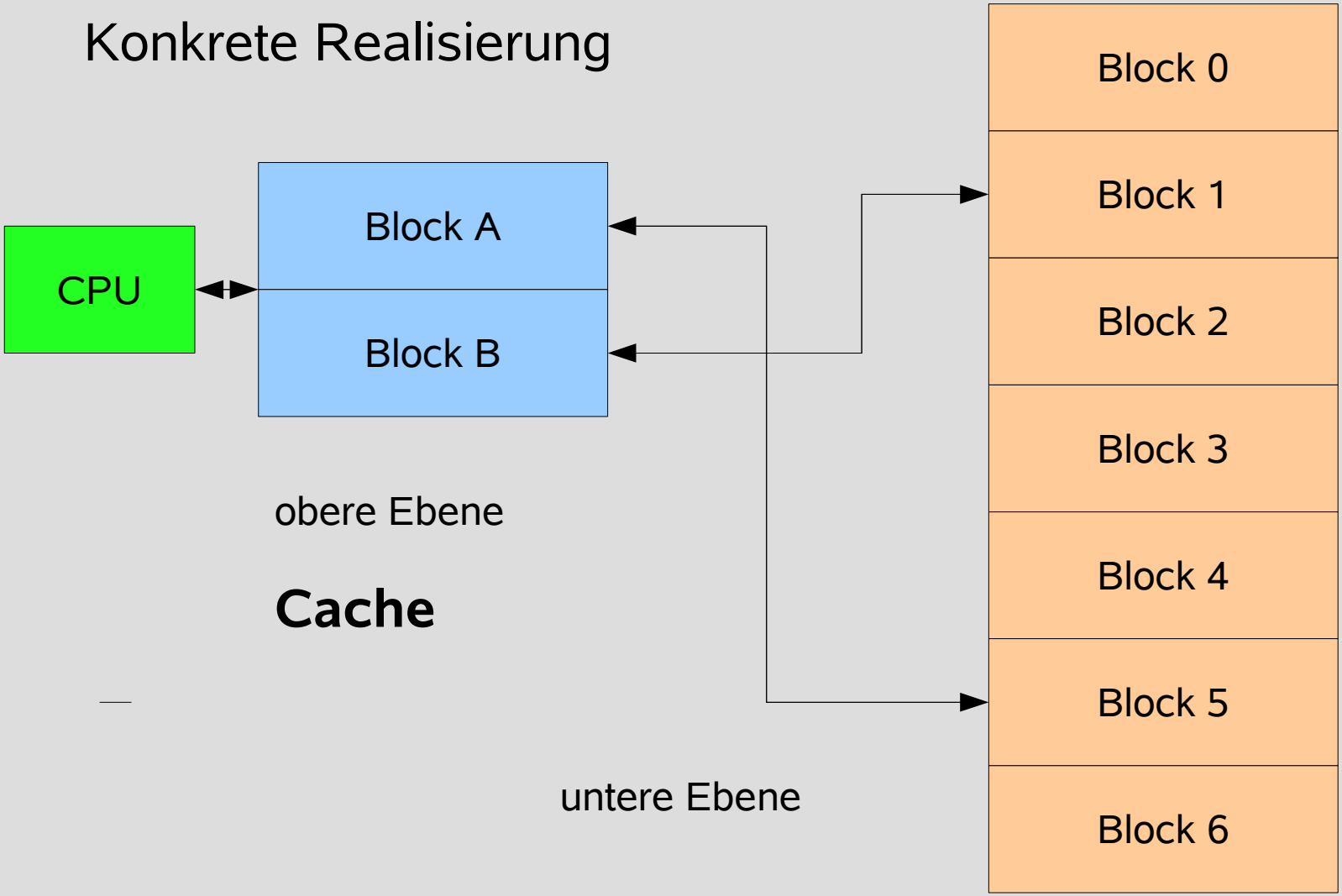


Halbleiterspeicher

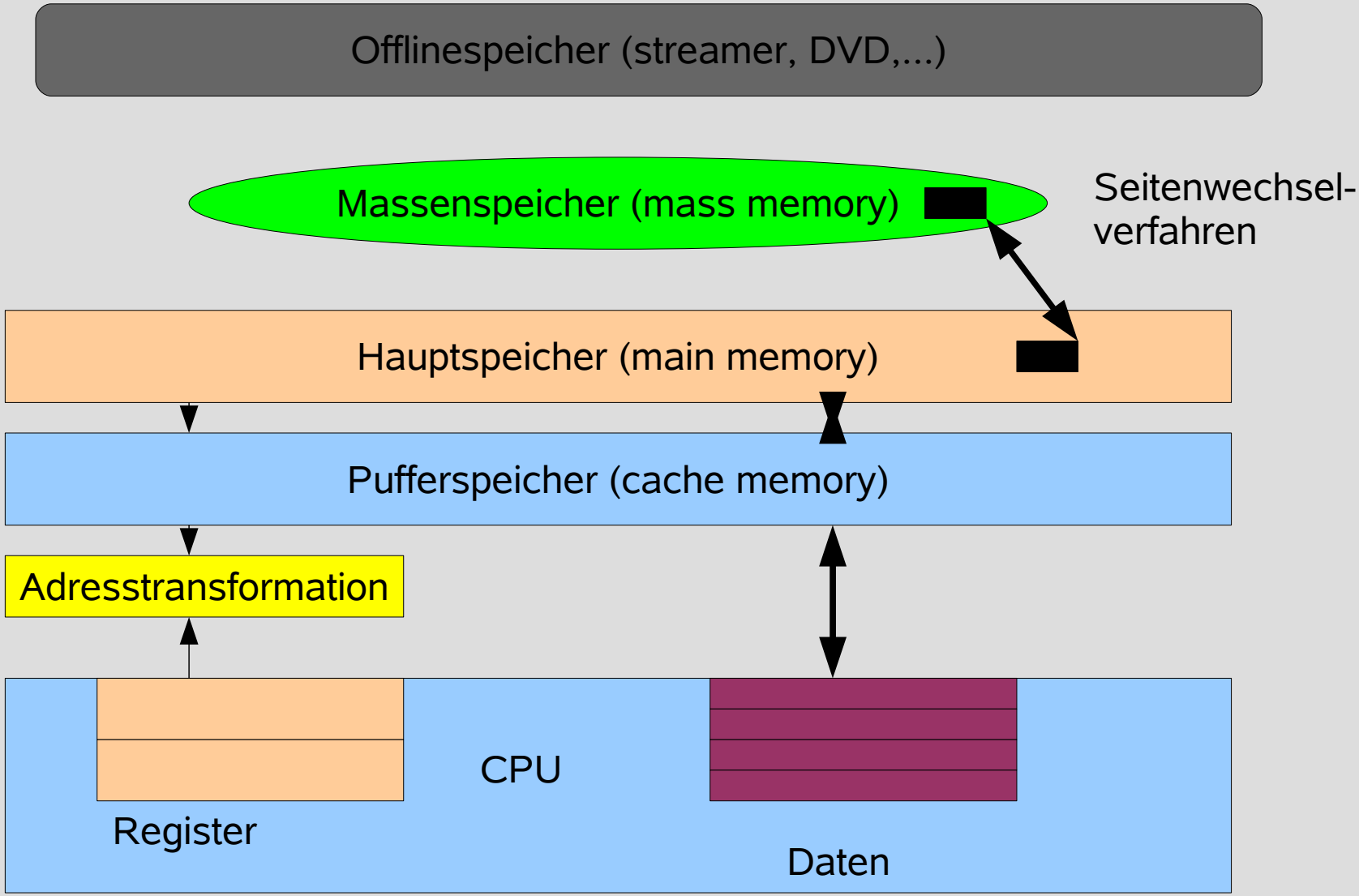
Speicherhierarchien (2)



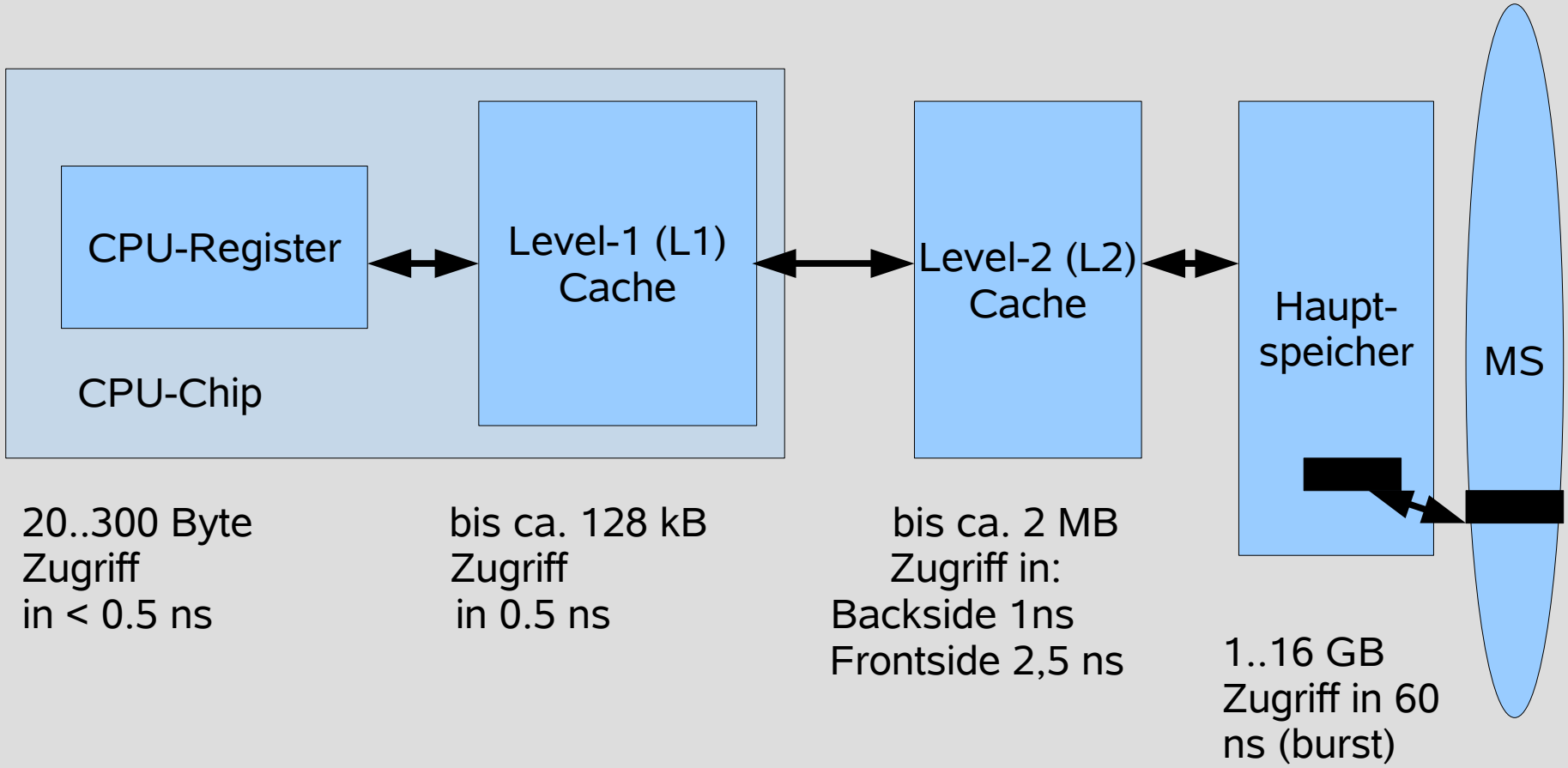
Speicherhierarchien (2)



Speicherhierarchien (3)



Speicherhierarchien (4)



Cachespeicher

- Weitere Verwendung des Cache-Prinzips:
  - Hardware:
    - Pufferspeicher der MMU (Translation Lookaside Buffer)
    - Lokale Pufferspeicher bei periphären Geräten (Netzwerkschnittstelle, Festplatte)
    - *Branch prediction cache, branch target cache* in der Pipelinesteuerlogik der CPU)
  - Software
    - Pufferspeicher des Betriebssystems (*disk cache, buffer cache, file cache*)
    - Pufferspeicher für Datenbanken

Cacheleistung

- In Abhängigkeit davon, ob ein Speicherinhalt sich im Cache oder im Hauptspeicher befindet, gelten die entsprechenden Zugriffszeiten
- Als Trefferrate (*hit rate*) bezeichnet man den Anteil aller Speicherzugriffe, bei denen die gewünschte Information aus dem Cache verfügbar ist.
- Wenn man die Information im Cache nicht gefunden hat, spricht man von einer Niete (*miss*).

Cacheleistung (2)

$$T_{eff} = hT_C + (1-h)T_M$$

mit

T_{eff} : *effektive Zugriffszeit (mittel)*

T_C : *Zugriffszeit des Cache – Speichers*

T_M : *Zugriffsteit des Hauptspeichers*

h : *Trefferrate*

Beispiel

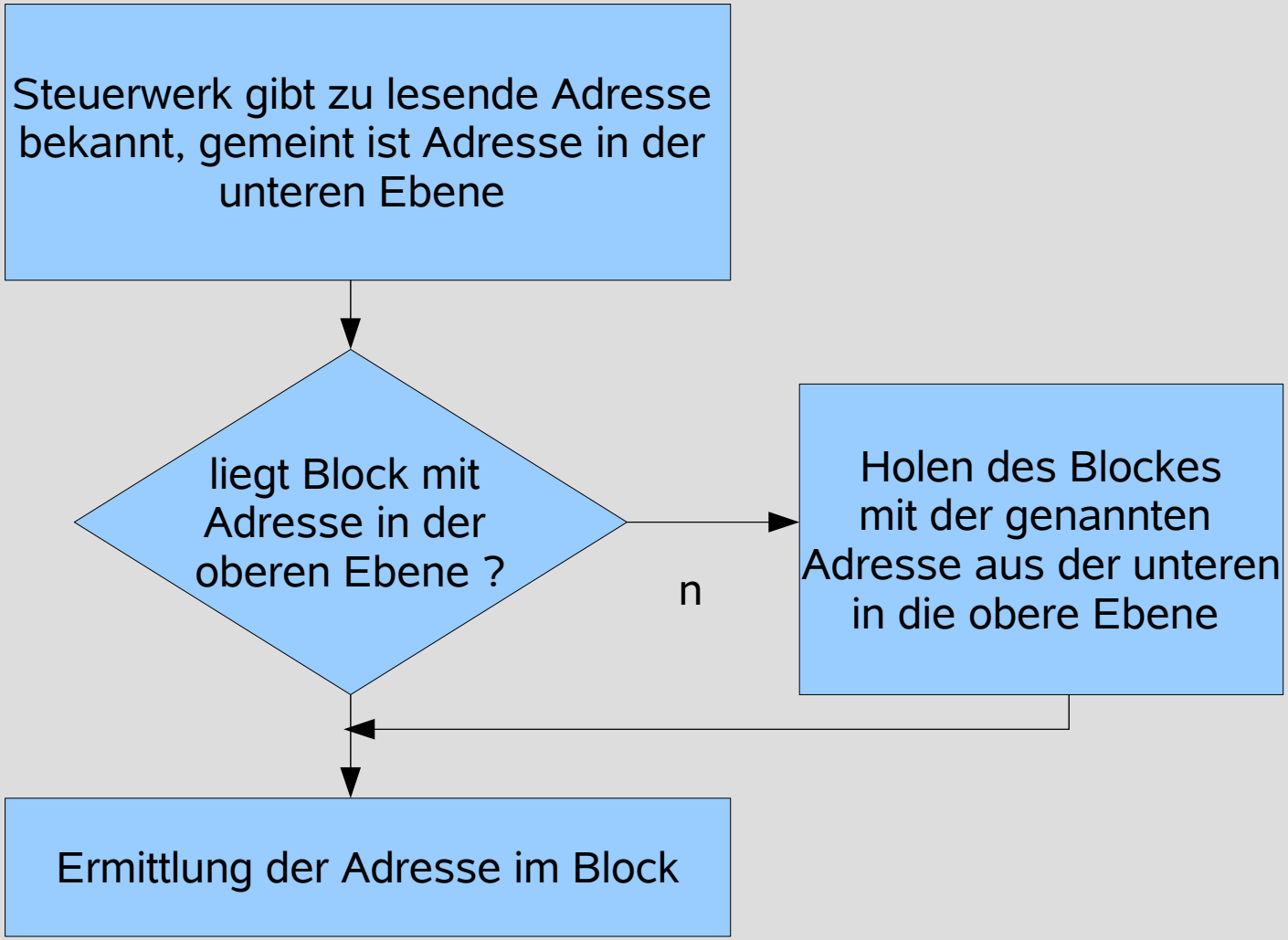
- Virtueller Speicher
  - Zusammenhang Hardware – Betriebssystem anhand von Pagingverfahren

Probleme

- Probleme entstehen durch folgende Umstände:
  - Steuerwerk kennt nur die Adressen der unteren Ebene, während tatsächlich mit Speicherzellen der oberen (und nur mit diesen!) gearbeitet wird.
  - Die obere und die untere Ebene haben nur mittelbar miteinander zu tun.
  - Das führt bei Schreibzugriffen zu Inkonsistenzen zwischen der oberen und der unteren Ebene, die (zumindest langfristig) aufgehoben werden müssen.

Probleme (2)

Lesen:



Probleme (3)

- Beim Verzweigen in den „nein“-Zweig muss der entsprechende Block aus der unteren Ebene in die obere Ebene transportiert werden.
- Die obere Ebene ist voll, deshalb muss ein Block aus der oberen Ebene entfernt werden.
- Aber welcher?

  Blockersetzungsstrategien

  FIFO, Not recently used, Second chance ....

Blockersetzungsstrategien

- **Optimaler Algorithmus**
  - Leicht zu beschreiben, aber unmöglich zu implementieren ;-(
  - Wenn ein Block ersetzt werden muss, sind eine Menge von Blöcken in der oberen Ebene. Jeder dieser Blöcke erhält eine Markierung, die darüber Auskunft gibt, wann in der Zukunft auf diesen Block zugegriffen wird.
  - Der optimale Algorithmus sagt einfach, dass der Block mit der höchsten Markierungsnummer entfernt werden sollte (ein unangenehmes Ereignis wird soweit wie möglich weggeschoben).
  - **Diese Information ist aber unbekannt.**

Blockersetzungsstrategien (2)

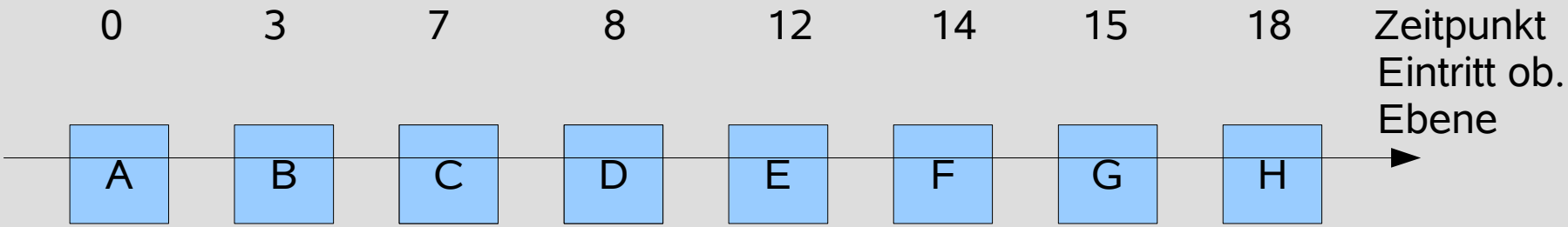
- **Not Recently Used (NRU)**
  - Einteilung der Blöcke in Klassen:
    - Klasse 0: nicht referenziert, nicht modifiziert
    - Klasse 1: nicht referenziert, modifiziert
    - Klasse 2: referenziert, nicht modifiziert
    - Klasse 3: referenziert, modifiziert
  - R-Bit wird periodisch (z.B. bei einem Uhrinterrupt) zurückgesetzt, M-Bit bleibt unverändert (wegen Konsistenzproblemen zwischen oberer und unterer Ebene (siehe später)). Dadurch entsteht auch Klasse 1.
  - Bei Notwendigkeit einer Blockersetzung wird zufällig ein Block aus der kleinsten nichtleeren Klasse entfernt.

Blockersetzungsstrategien (3)

- **First-In-First-Out (FIFO)**
  - Der am längsten in der oberen Ebene befindliche Block wird entfernt
  - Analogie: Supermarkt

Blockersetzungsstrategien (4)

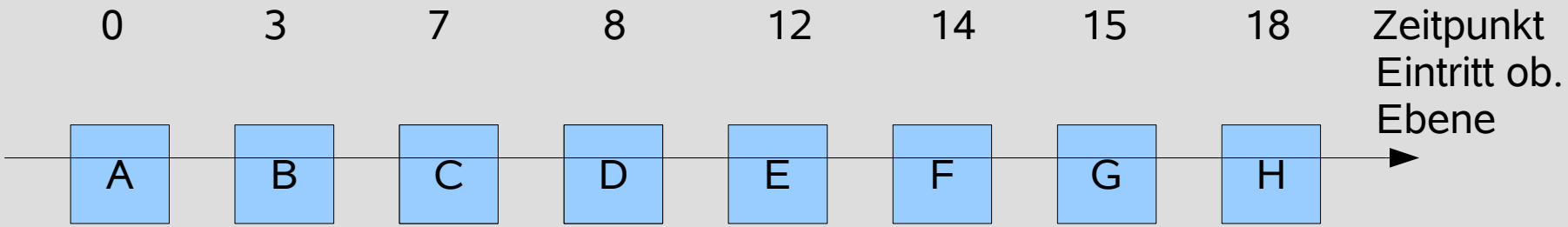
- **Second Chance**
  - Modifikation FIFO



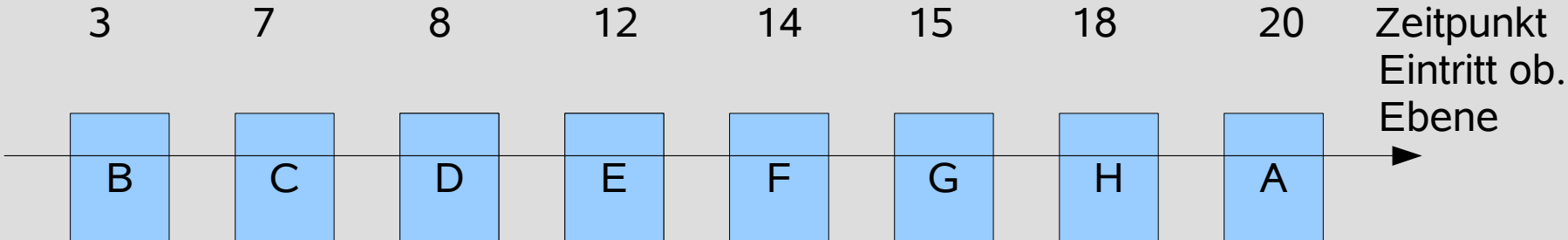
Blockfehler zum Zeitpunkt 20. Wenn R-Bit nicht gesetzt ist, wird der Block A entfernt. Wenn R-Bit gestzt ist, wird A an das Ende der Liste gesetzt, das R-Bit wird zurückgesetzt. Die Suche nach einem Auslagerungskandidaten wird mit B fortgesetzt.

Blockersetzungsstrategien (4)

- **Second Chance**
  - Modifikation FIFO



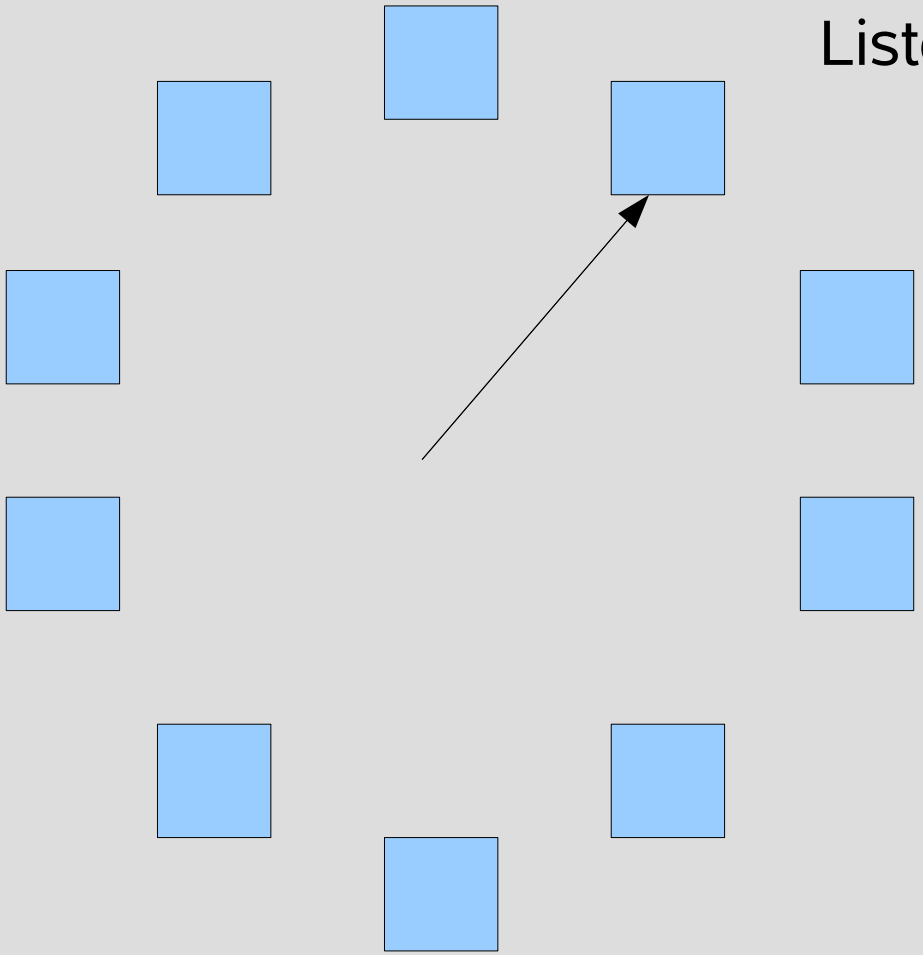
Blockfehler zum Zeitpunkt 20. Wenn R-Bit nicht gesetzt ist, wird der Block A entfernt. Wenn R-Bit gestzt ist, wird A an das Ende der Liste gesetzt, das R-Bit wird zurückgesetzt. Die Suche nach einem Auslagerungskandidaten wird mit B fortgesetzt.



Blockersetzungsstrategien (5)

- **Uhr-Algorithmus**

Modifikation Second Chance: aus linearen Listen werden Ringlisten



Blockersetzungsstrategien (6)

- **Least-Recently-Used**

0	1	2	3
0	1	1	1
0	0	0	0
0	0	0	0
0	0	0	0

0	1	2	3
0	0	1	1
1	0	1	1
0	0	0	0
0	0	0	0

0	1	2	3
0	0	0	1
1	0	0	1
1	1	0	1
0	0	0	0

0	1	2	3
0	0	0	0
1	0	0	0
1	1	0	0
1	1	1	0

Blockersetzungsstrategien (6)

- **Least-Recently-Used**

0	1	2	3
0	1	1	1
0	0	0	0
0	0	0	0
0	0	0	0

0	1	2	3
0	0	1	1
1	0	1	1
0	0	0	0
0	0	0	0

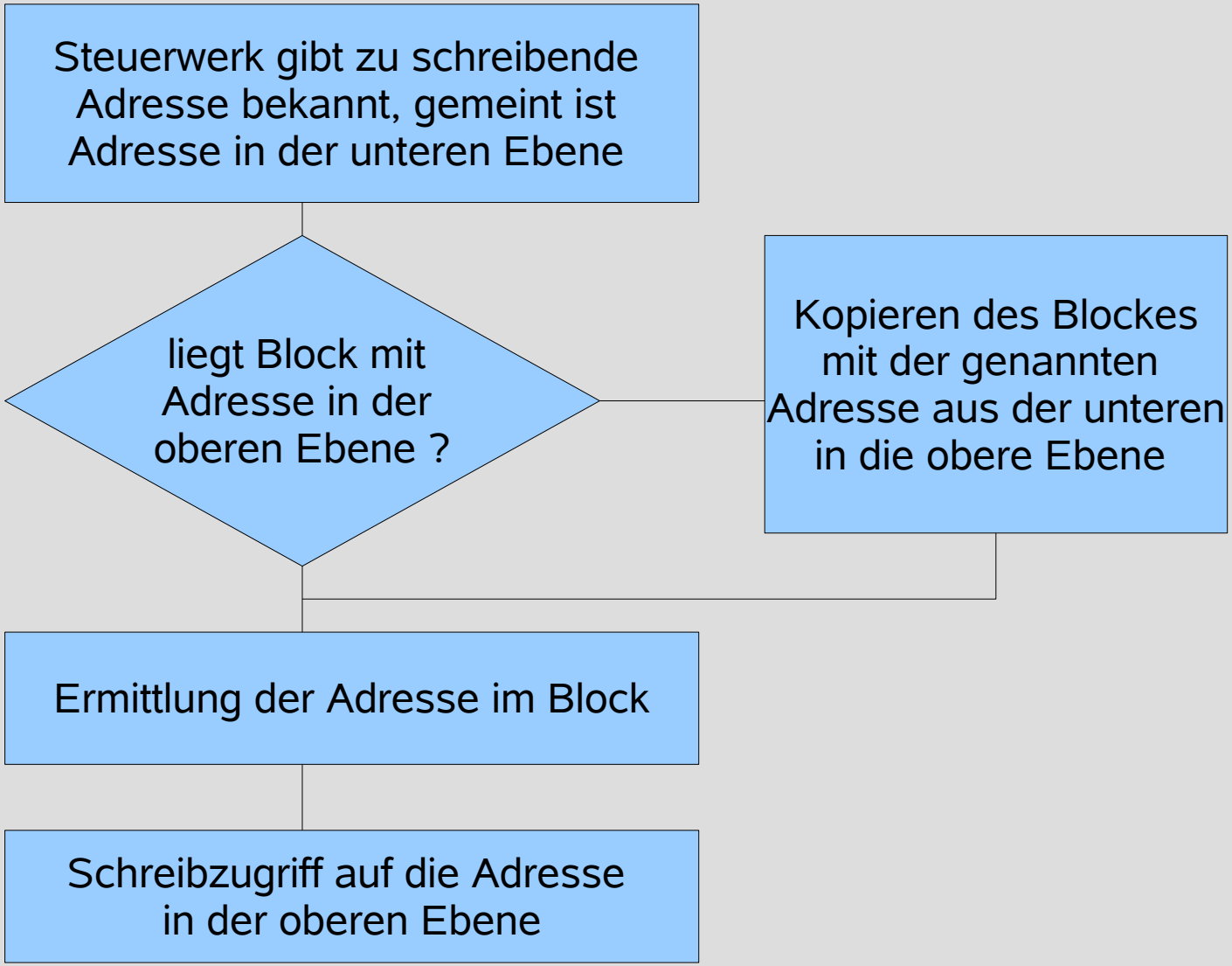
0	1	2	3
0	0	0	1
1	0	0	1
1	1	0	1
0	0	0	0

0	1	2	3
0	0	0	0
1	0	0	0
1	1	0	0
1	1	1	0

Frage: Wie geht es weiter bei Blockzugriffsreihenfolge: 0 1 2 3 **2 1 0 3 2 3**

Probleme (4)

Schreiben:



Probleme (5)

- Beim Schreiben treten folgende Probleme auf:
  - es wird immer versucht, in einem Block der oberen Ebene zu schreiben.
  - wenn dieser nicht da ist, so kann man entweder diesen Block holen (**fetch on write**), oder man verzichtet darauf und schreibt direkt in die untere Ebene (**work around**). Da Statistiken sagen, dass Schreibzugriffe wesentlich seltener auftreten als Lesezugriffe, ist der Zeitverlust verschmerzbar.

Probleme (6)

- Wenn aber der Block in der oberen Ebene präsent war, wird immer dorthin geschrieben.

- weitere Probleme: es gibt nun eine Inkonsistenz zwischen oberer und unterer Ebene:
  - **Write through**
  - **Write back**

Speicherhierarchien

- Weitere Behandlung:
  - Kapitel 8 (Parallele Rechnerarchitekturen)

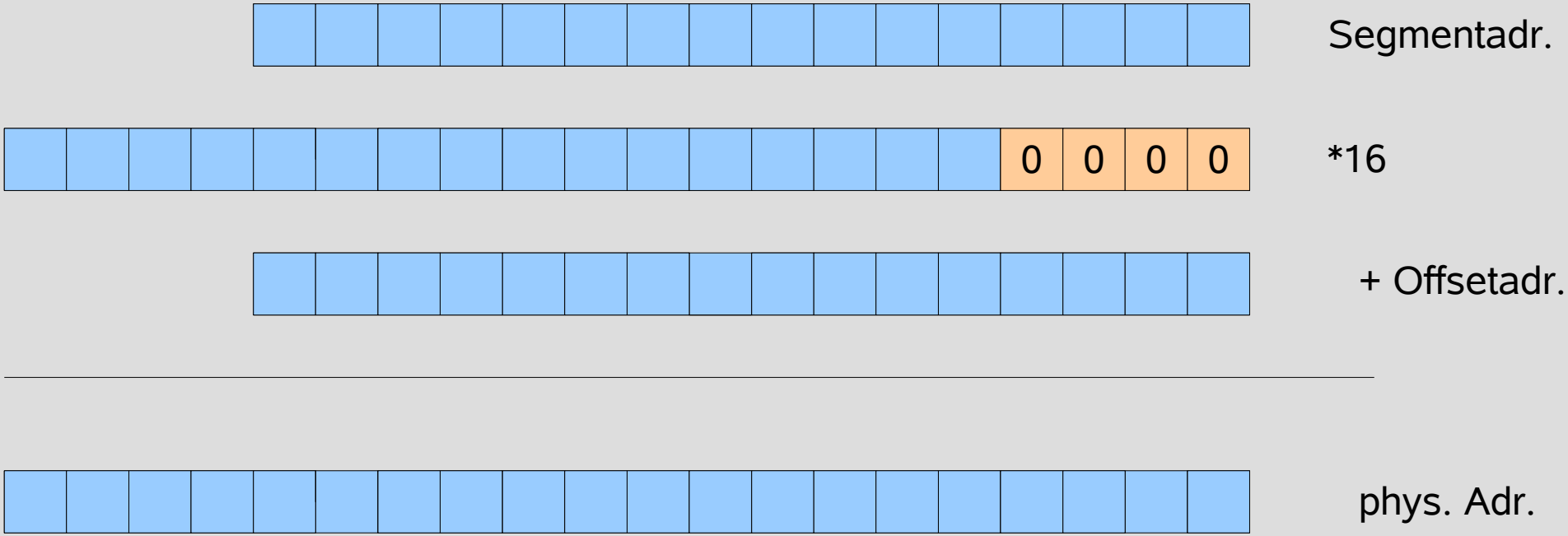
Adressierungsverfahren, Speicherorganisation

- Behandlung anhand i80x86 (bzw. Pentium)
- Kapitel 4: Realmode

Adressierungsverfahren i8086

- Real-Mode
  Adressraum: 1 Mbyte (2^{20})
  Adressregister: 64 kByte (2^{16})
  Wie wird dieser Wiederspruch gelöst?
  <Adressregister> * 16 + Offset
- Segmentierung:

Adressierungsverfahren i8086 (2)



Protected Mode

- Logisches Verfahren zum Zugriff auf Arbeitsspeicher
- In Zusammenhang mit einem geeigneten Betriebssystem können so (quasi)parallele Anweisungen realisiert werden
- (im Realmode kann von jedem Programm der gesamte Speicher adressiert werden)

Protected Mode (2)

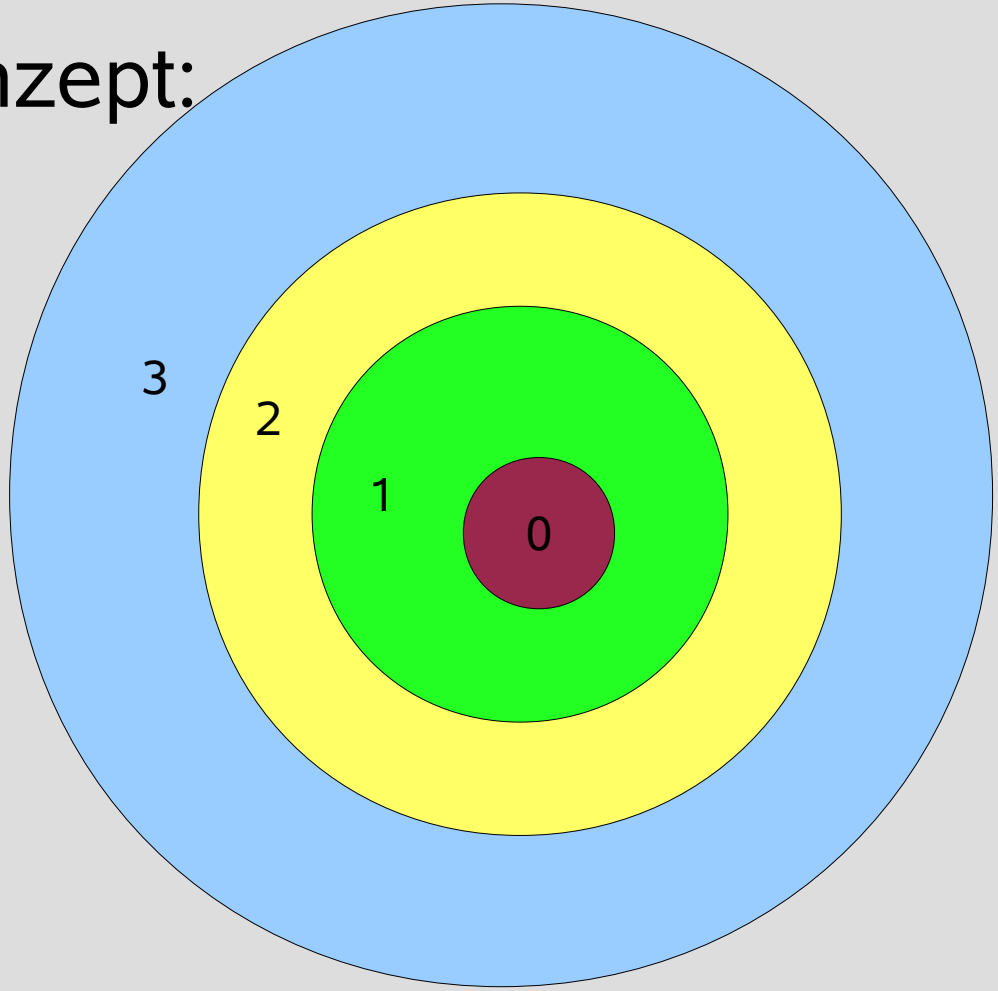
- Schutzkonzept:

0 – höchste Priorität (meist BS)

1 – BS-Dienste (Dateiverwaltung, Geräteverw.,...)

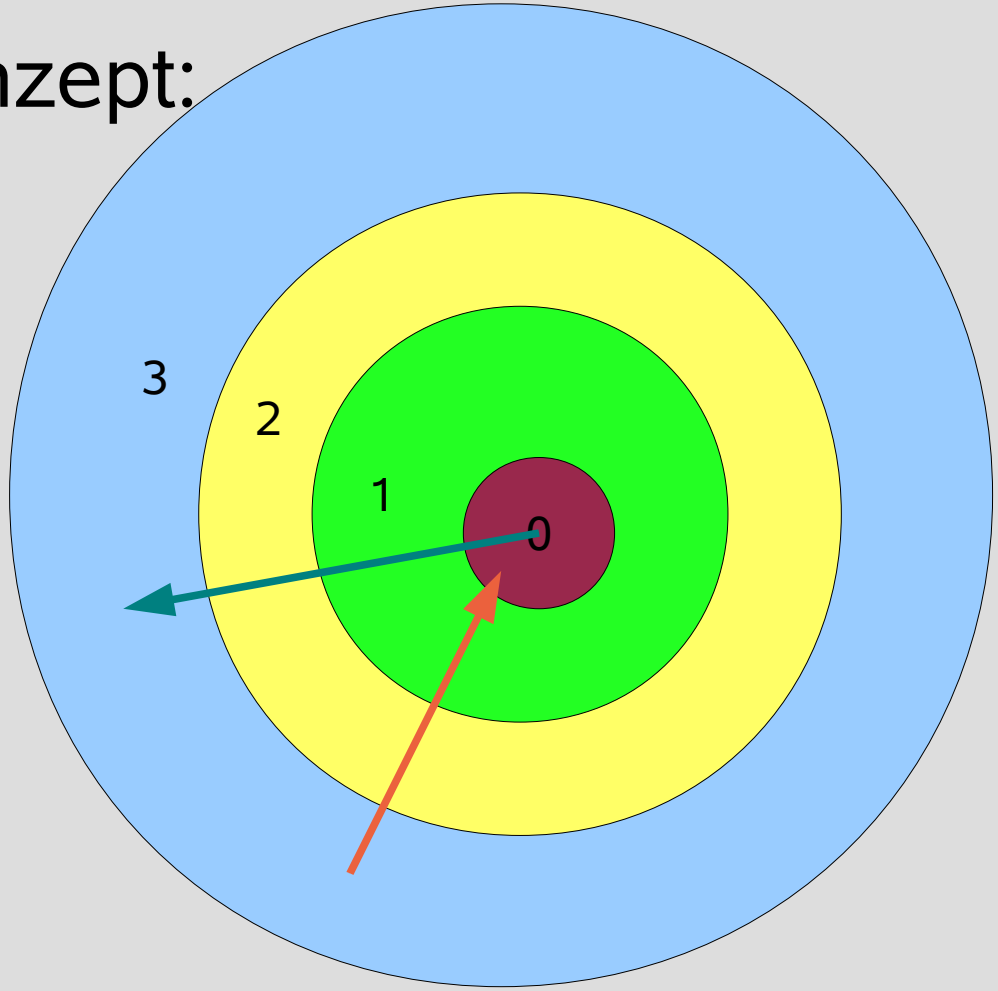
2 – BS-Erweiterungen

3 - Anwendungen



Protected Mode (2)

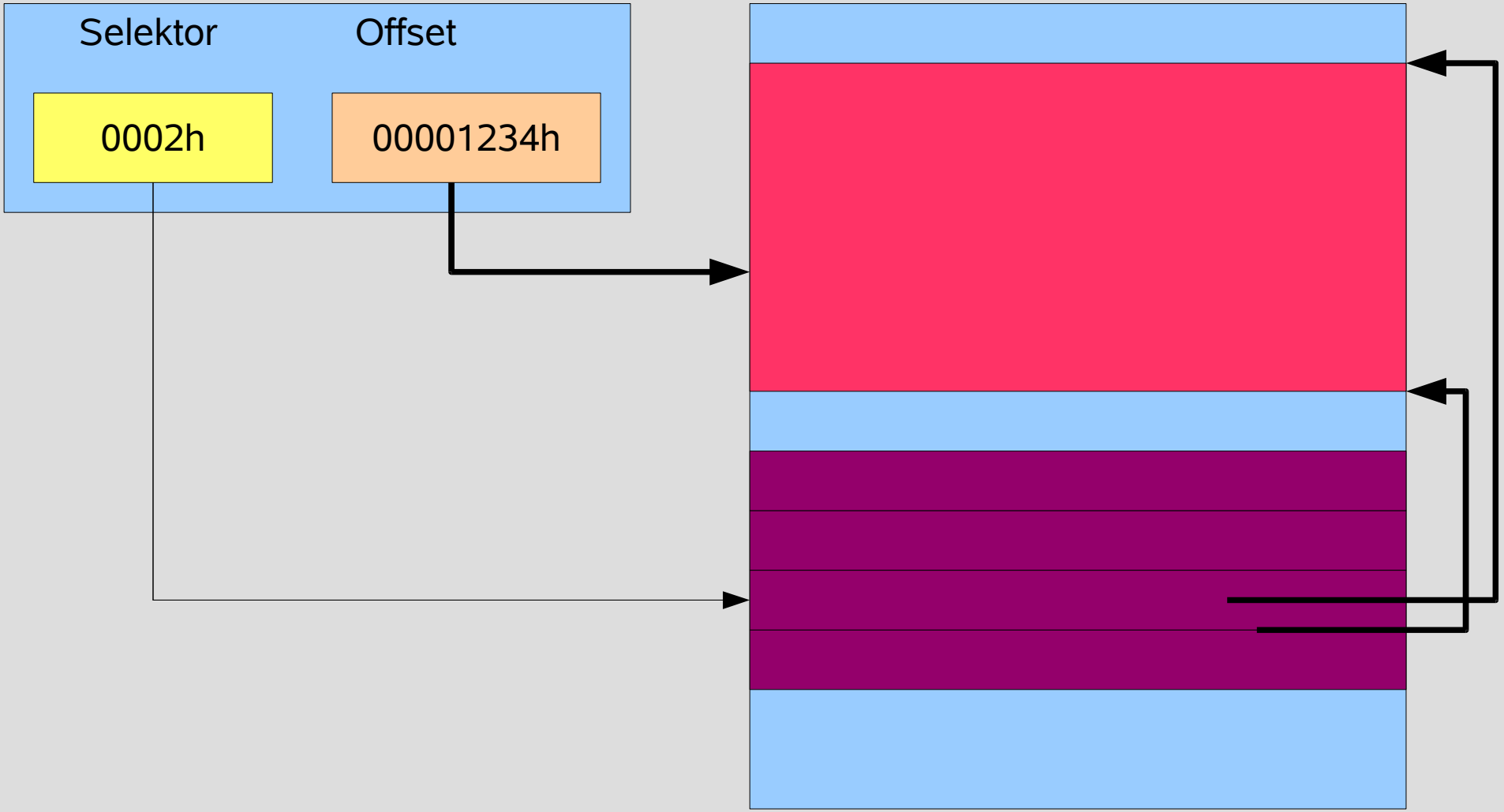
- Schutzkonzept:



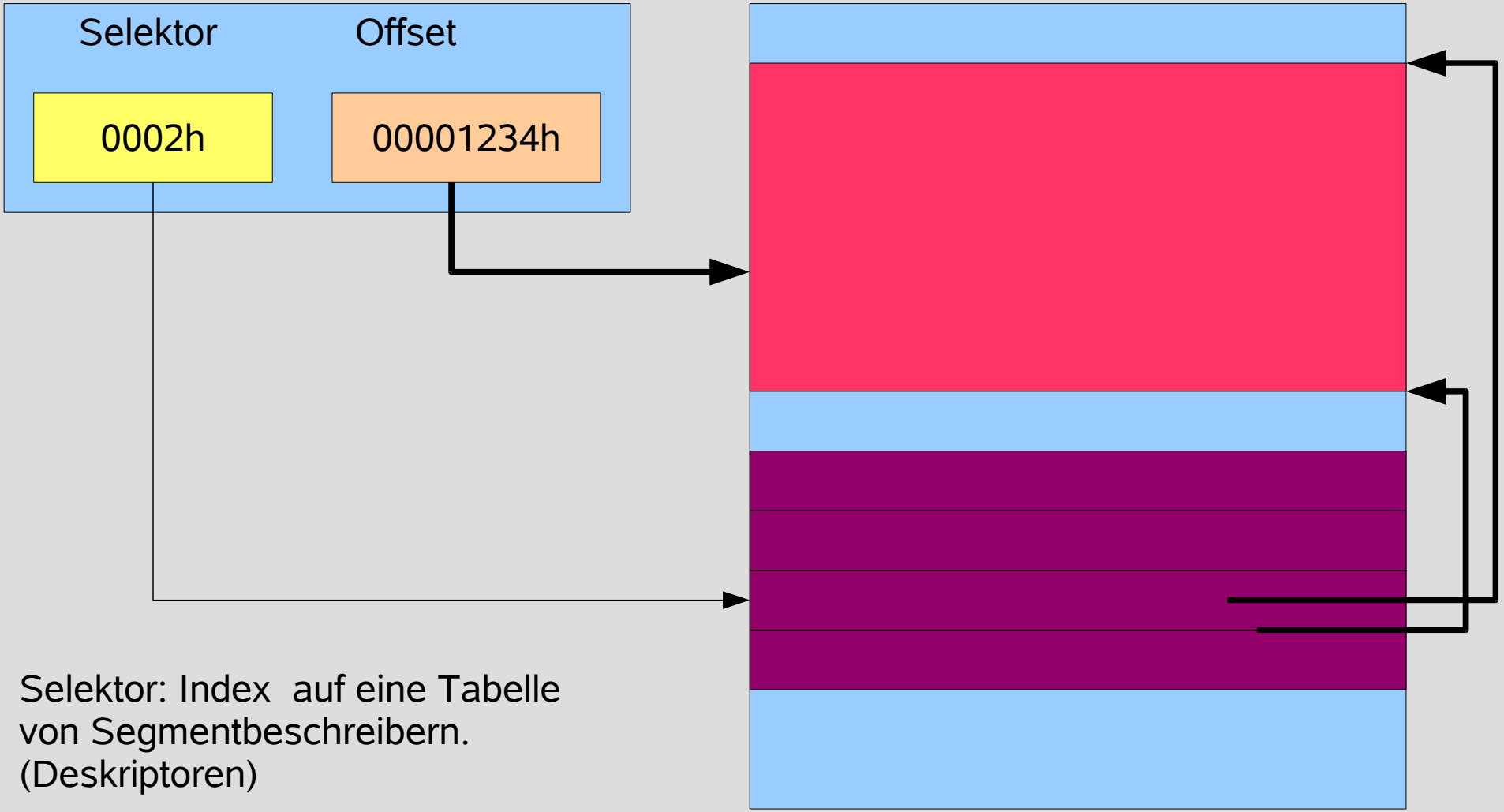
Virtuelle Adressierung

- Wie auch im Realmode exisitieren Segmentadressen
- Speicher wird als virtuell interpretiert, d.h. Prinzipiell spielt pysische Größe des Speichers keine Rolle
- Adresse wird aus 2 Teilen gebildet:
  - Selektor
  - Offset

Protected Mode

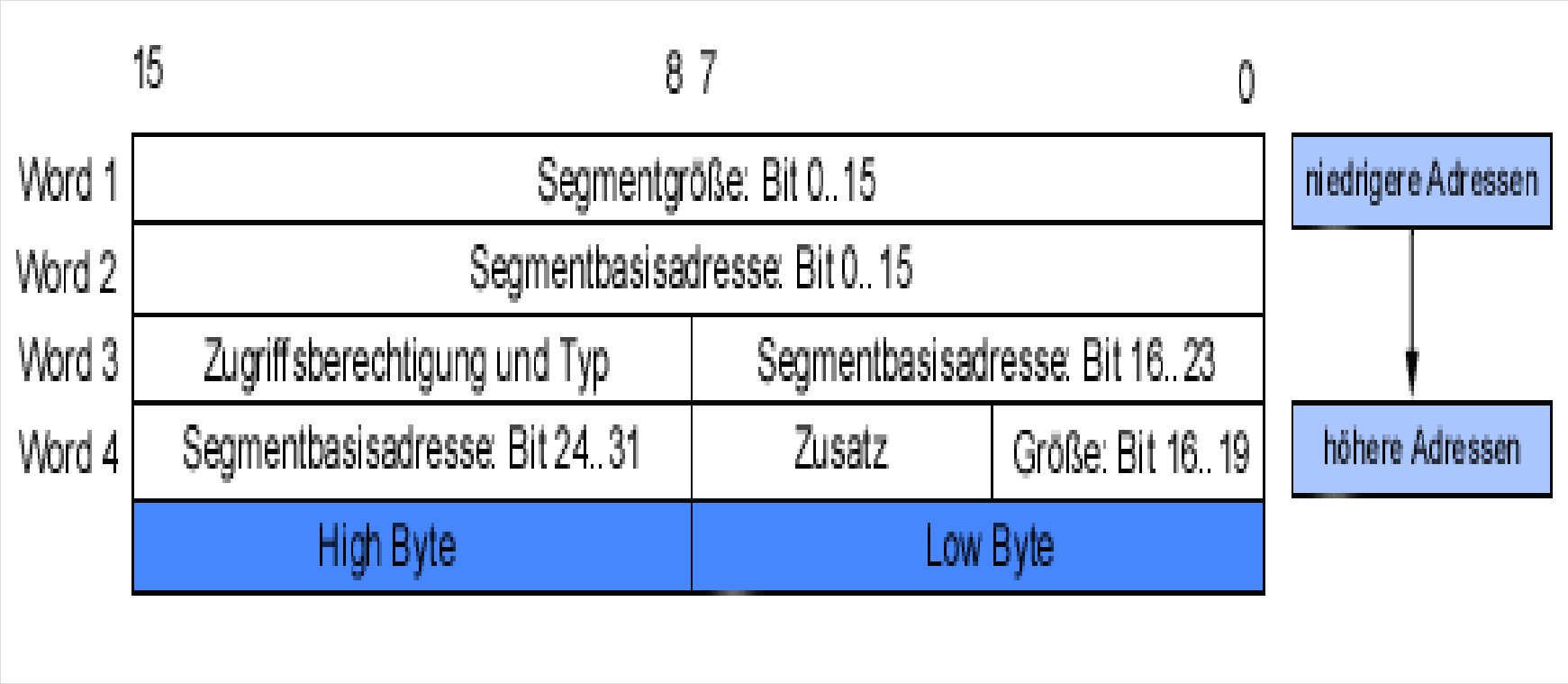


Protected Mode



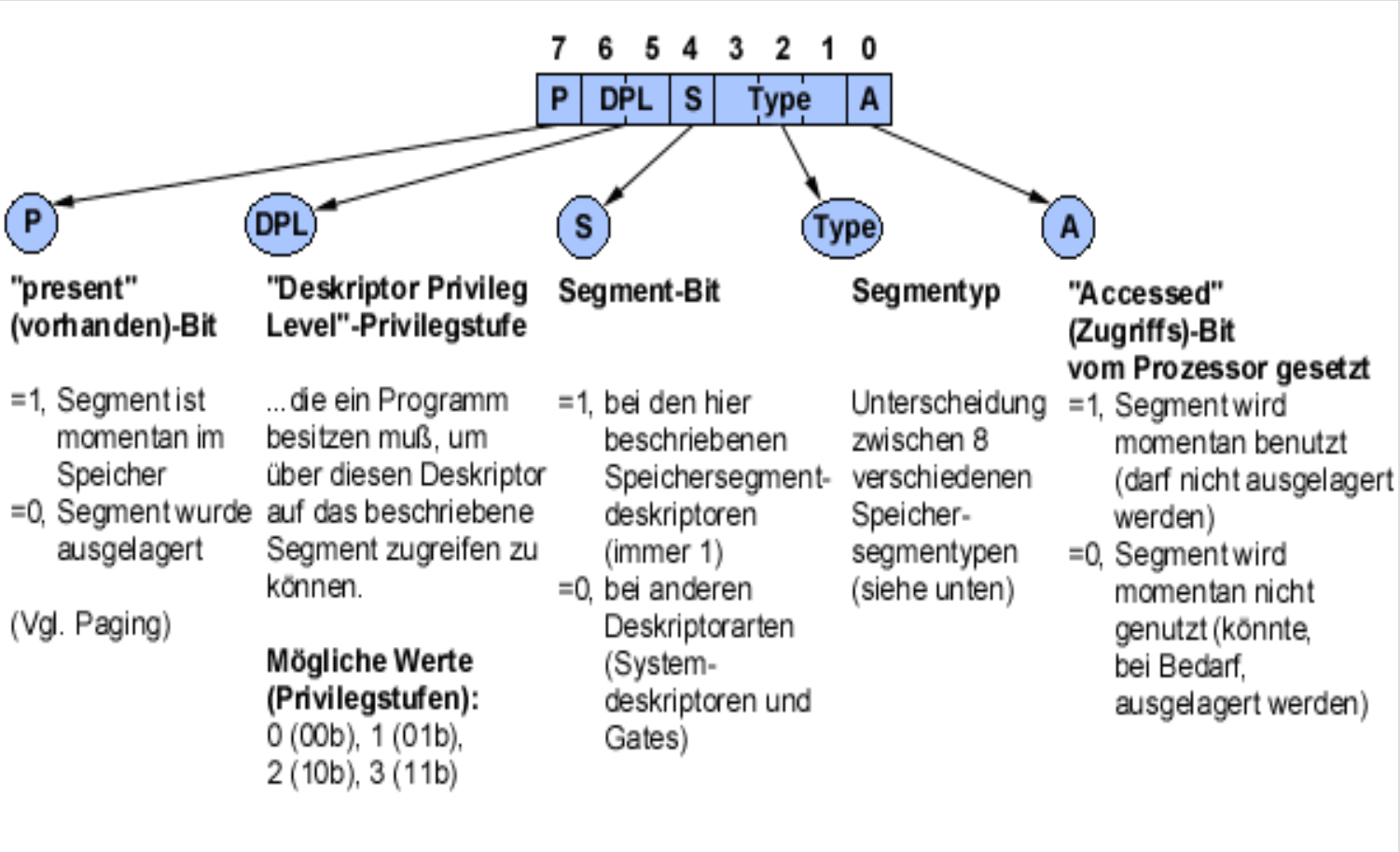
Selektor: Index auf eine Tabelle von Segmentbeschreibern. (Deskriptoren)

Deskriptoren



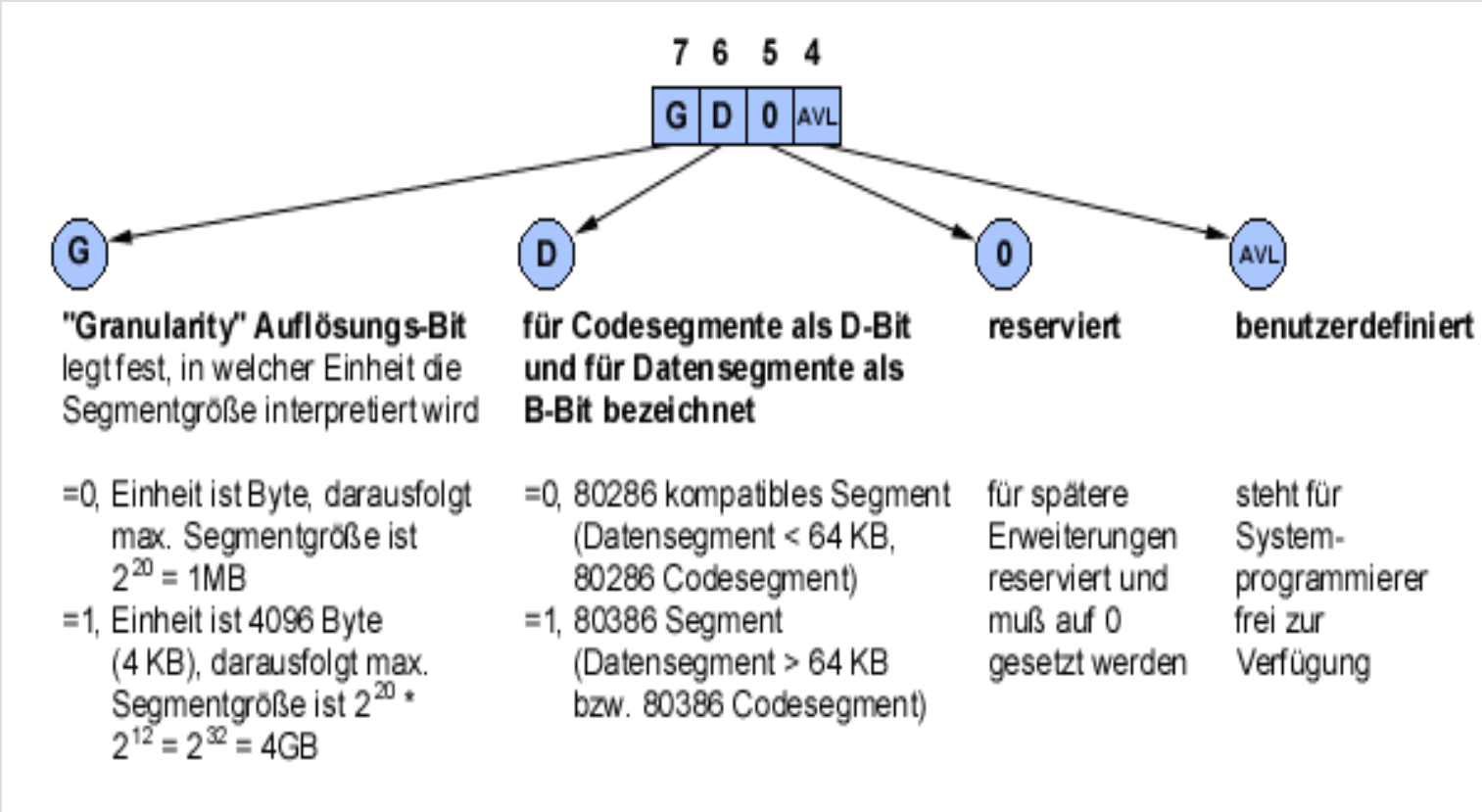
Deskriptoreintrag i80386

Deskriptoren (2)



Zusatzinformationen, Byte 6

Deskriptoren (3)



Zusatzinformationen, Byte 7, Bit 4..7

Deskriptoren-Beispiel

- Maximale Größe: 8192 Einträge (64k / 8Byte)
- 3 Typen
  - Globale Deskriptortabelle (GDT)
  - Interruptdeskriptortabelle
  - Lokale Deskriptortabelle

Deskriptoren-Beispiel

- Segmentstart an der pysischen Adresse 01F2E3Dh
- Länge 2 MB (200000h)
- Datensegment (r/w)
- DPL von 2

Assemblerquelltext:

```
mein_deskritor:
    dw 0200             ; Segmentgröße, Bit 0..15
    dw 2E3Dh            ; Segmentbasisadresse, Bit 0..15
    db  1Fh             ; Segmentbasisadresse, Bit 16..23
    db 11010010b        ; Zugriffsberechtigung und Typ
    db 11000000b        ; Zusatzinformation und Segmentgröße Bit 16..19
    db 0                ; Segmentbasisadresse, Bit 24..31
```

Selektoren

TI

Table Indicator

legt fest, ob sich der angegebene Index auf die Globale Deskriptor-tabelle oder eine Lokale Deskriptor-tabelle bezieht

0 - GDT wird adressiert
1 - LDT wird adressiert

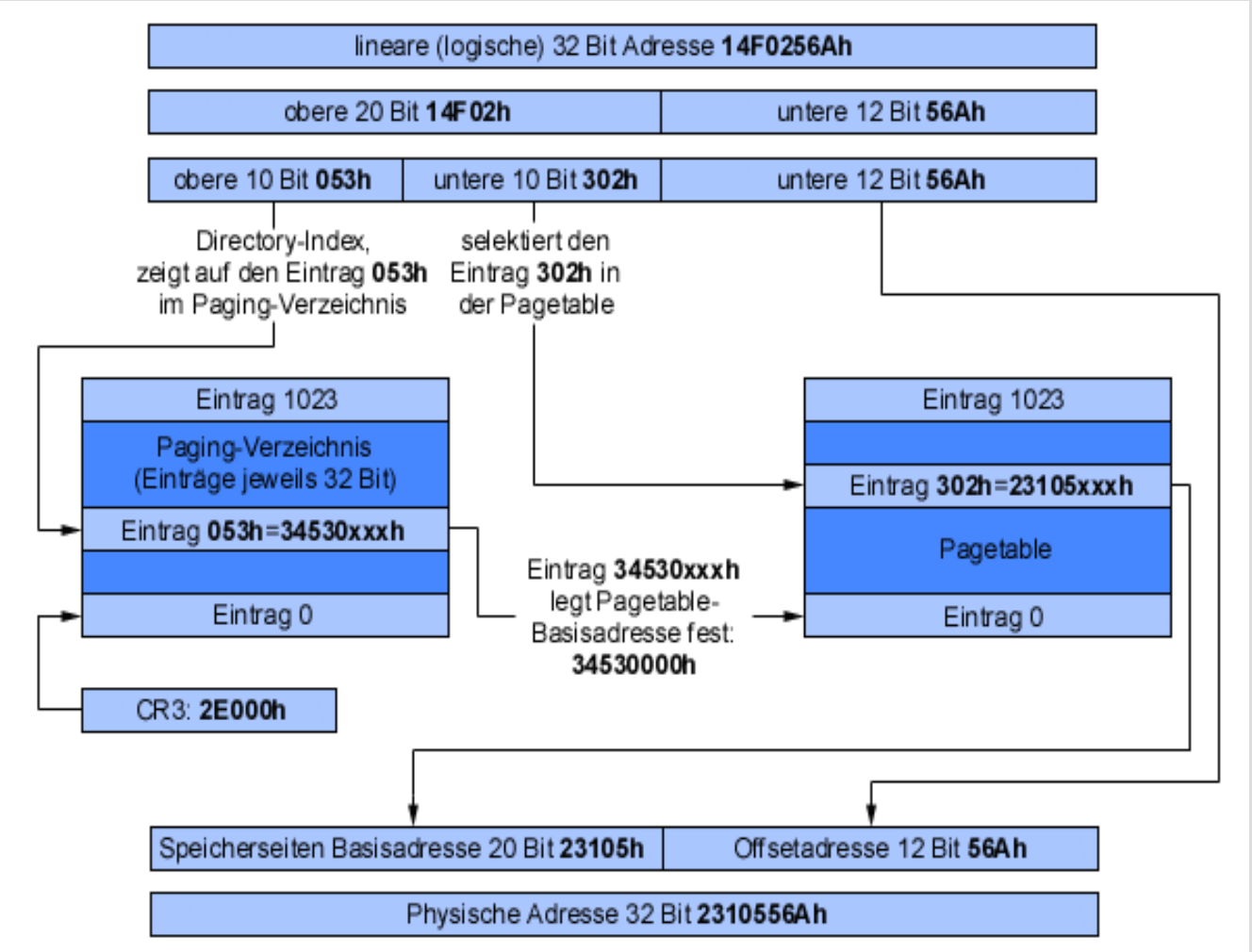
RPL

Requested Privilege Level

wie im Deskriptor definiert

00b - Stufe 0
01b - Stufe 1
10b - Stufe 2
11b - Stufe 3

Paging


lineare (logische) 32 Bit Adresse 14F0256Ah
obere 20 Bit 14F02h
untere 12 Bit 56Ah
obere 10 Bit 053h
untere 10 Bit 302h
untere 12 Bit 56Ah
Directory-Index, zeigt auf den Eintrag 053h im Paging-Verzeichnis
selektiert den Eintrag 302h in der Pagetable
Eintrag 1023
Paging-Verzeichnis (Einträge jeweils 32 Bit)
Eintrag 053h=34530xxxh
Eintrag 0
CR3: 2E000h
Eintrag 34530xxxh legt Pagetable-Basisadresse fest: 34530000h
Eintrag 1023
Eintrag 302h=23105xxxh
Pagetable
Eintrag 0
Speicherseiten Basisadresse 20 Bit 23105h
Offsetadresse 12 Bit 56Ah
Physische Adresse 32 Bit 2310556Ah