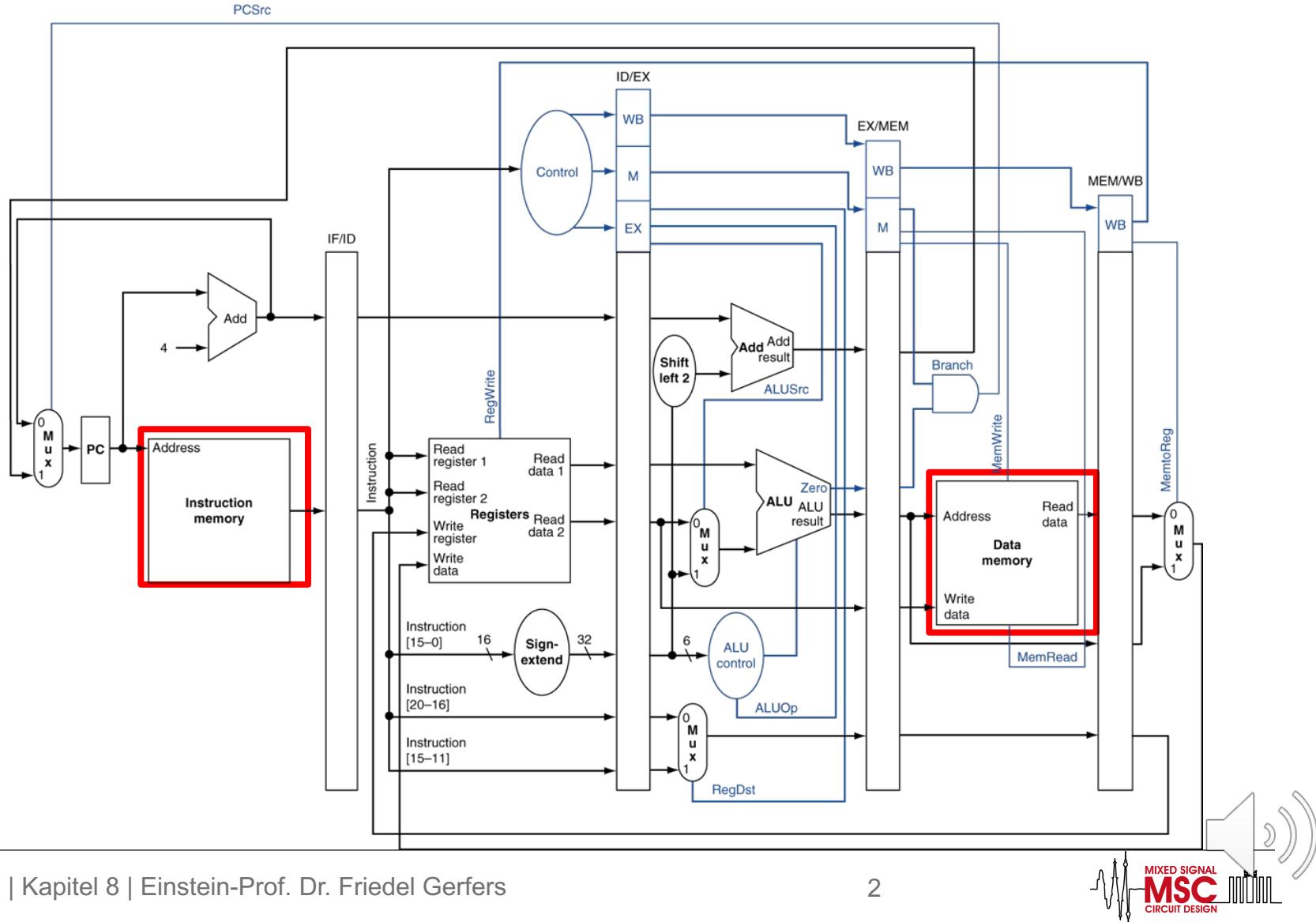


Rechnerorganisation

Einstein-Prof. Dr.-Ing. Friedel Gerfers



Speicher in unserem Prozessor



Kapitel 8: Die Speicherhierarchie

Nach dieser Vorlesung sind Sie in der Lage:

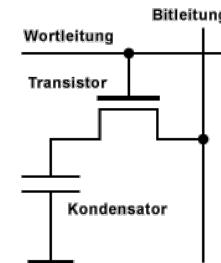
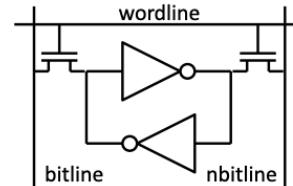
- Folgende Begriffe zu erklären:
 - Cache, Block/Zeile, temporale und räumliche Lokalität, Treffer, Fehlzugriff, Tag, Fehlzugriffsaufwand, Durch-/Rückschreibe-technik, Gültigkeits- und *dirty*-Bit, satz-assoziativer Cache, ...
- Beschreibung eines Caches geben:
 - Parameter wie Größe des Caches, der Index, des Tags, ... zu berechnen
 - Für eine Reihe von Speicherzugriffen: Für jeden Zugriff angeben, ob es ein Treffer oder Fehlzugriff ist
- CPU-Zeit zu berechnen, bei gegebener Fehlzugriffsrate und – aufwand
- Durchschnittliche Zugriffszeit und optimale Blockgröße zu berechnen
- Zu berechnen, wie viele Bit man braucht um einen Cache zu implementieren

Warum Speicherhierarchie?

- Benutzer wollen unbegrenzt großen und unendlich schnellen Speicher
 - Schnellere Speicher sind teurer und brauchen mehr Platz auf dem Chip
 - RAM (Random Access Memory)
 - Static RAMs (**SRAM**) sind statische Direktzugriffsspeicher
 - Dynamic RAMs (**DRAM**) das sind dynamische Direktzugriffsspeicher

Technologie	Zugriffszeit	\$ pro GB (2008)
SRAM	0.5-2.5 ns (1-5 cc)	\$2k-\$5k
DRAM	50-70 ns (50-150 cc)	\$20-\$75
Festplatte	5M-20M ns	\$0.20-\$2

cc = Clock Cycle



Warum Speicherhierarchie?

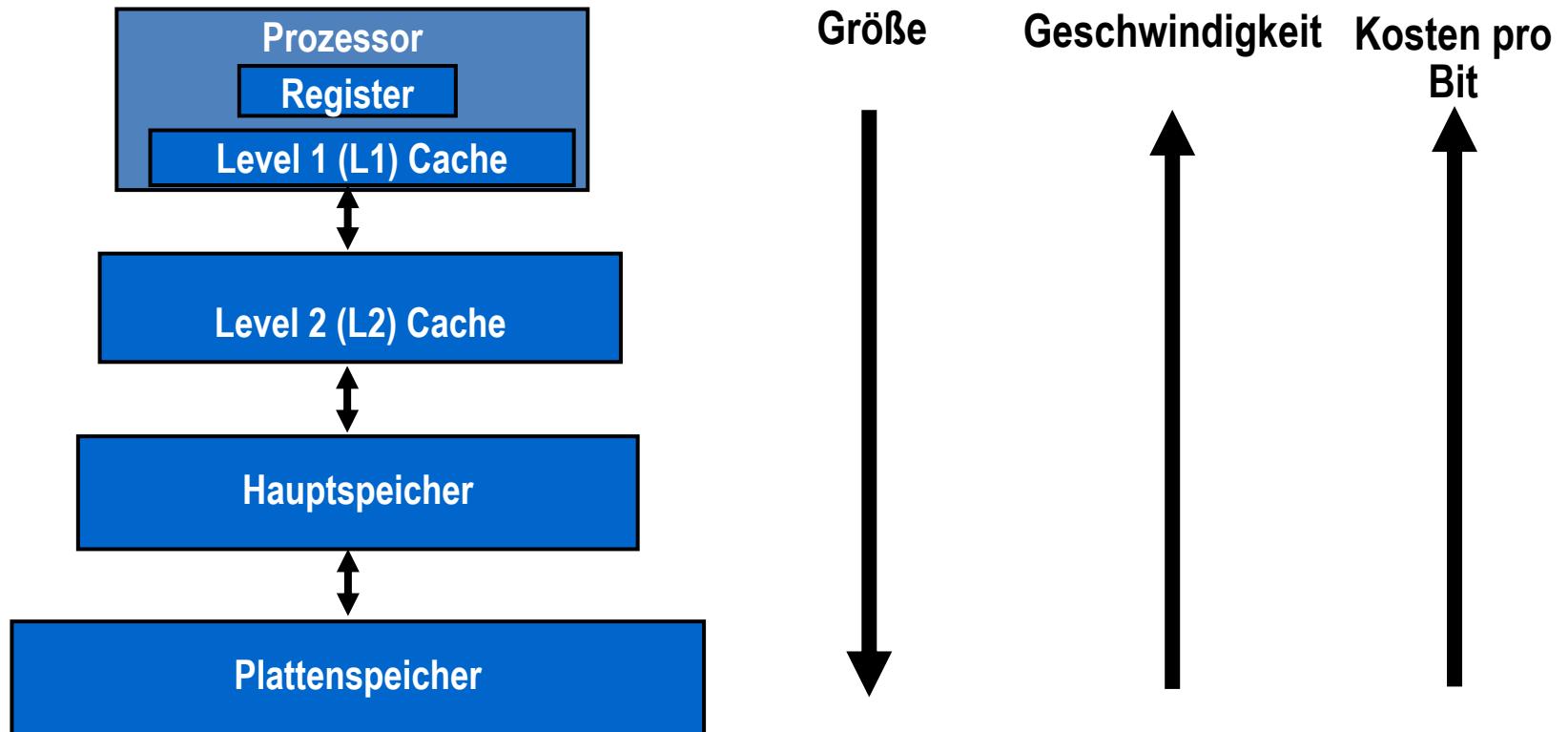
- Benutzer wollen unbegrenzt großen und unendlich schnellen Speicher
 - Schnellere Speicher sind teurer und brauchen mehr Platz auf dem Chip
 - RAM (Random Access Memory)
 - Static RAMs (**SRAM**) sind statische Direktzugriffsspeicher
 - Dynamic RAMs (**DRAM**) das sind dynamische Direktzugriffsspeicher

Technologie	Zugriffszeit	\$ pro GB (2008)
SRAM	0.5-2.5 ns (1-5 cc)	\$2k-\$5k
DRAM	50-70 ns (50-150 cc)	\$20-\$75
Festplatte	5M-20M ns	\$0.20-\$2

cc = Clock Cycle

- Aufbau einer Speicherhierarchie:
 - Mehrere Speicherebenen mit verschiedenen Geschwindigkeiten und Größen
 - Schnellere Speicher sollten häufig genutzte Daten enthalten

Aufbau einer Speicherhierarchie

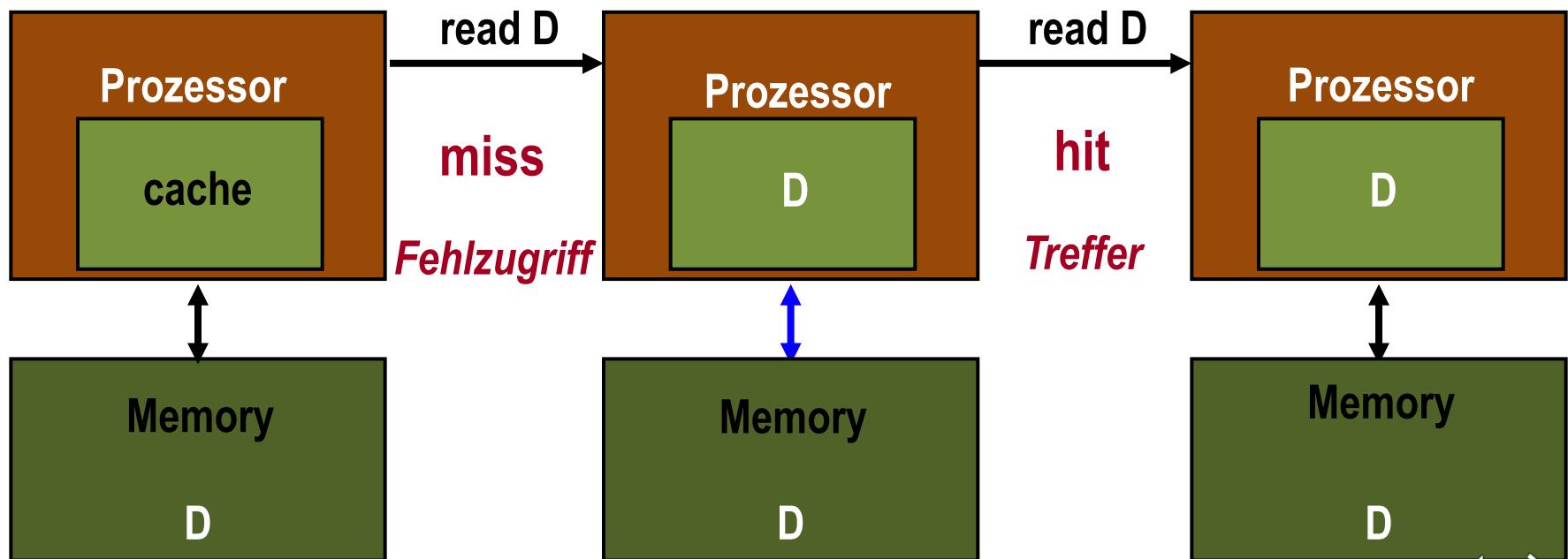


Cache (\$)

- Cache (\$) = Hochgeschwindigkeitsspeicher zwischen Hauptspeicher und Prozessor (CPU)
- Hardware-kontrolliert
 - Hardware bestimmt welcher Code/Daten im Cache gespeichert werden
 - Auf Abruf (*on demand*): wenn auf Code/Daten zugegriffen wird, die sich nicht im Cache befinden, werden sie in den Cache abgelegt
- Software-kontrollierte Hochgeschwindigkeitsspeicher werden oft als Notizblockspeicher (*scratchpad memory*) bezeichnet
 - IBM/Sony/Toshiba Cell Prozessor: Lokaler Speicher (LS)
 - OpenCL: Local Memory

Treffer und Fehlzugriffe (*hit and miss*)

- Cache-Hit-Rate h
- Frage: Wann werden Befehle/Daten in den Cache geladen?
- Vorläufige Antwort: auf Anfrage (*on demand*)



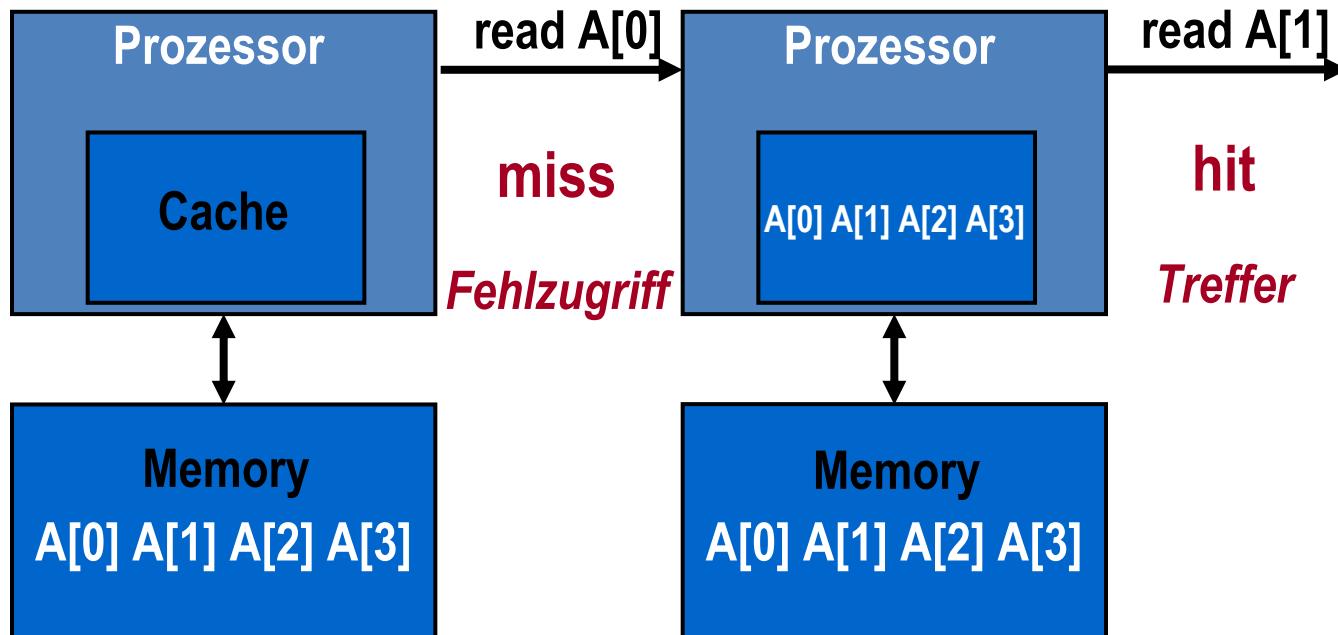
Lokalitätsprinzip

- Typische Programme:
 - Instruktionen werden der Reihe nach ausgeführt, nur unterbrochen durch Sprünge hauptsächlich in Schleifen oder als Prozeduraufruf
 - Dadurch wiederholtes Holen kürzlich geholter Instruktionen
 - Dadurch auch wiederholter Zugriff kürzlich zugegriffener Befehle.
- Zeitliche (temporale) Lokalität (*temporal locality*)
 - Wenn auf etwas zugegriffen wird, erfolgt bald wieder Zugriff darauf
- Räumliche Lokalität (*spatial locality*)
 - Wenn auf ein Element zugegriffen wird, erfolgt bald Zugriff auf in der Nähe befindliche Elemente
- Prinzip gilt für Daten und Instruktionen gleichermaßen

```
for (i=0; i<n; i++)
    sum += a[i];
```

Räumliche Lokalität

- Räumliche Lokalität besagt, dass nach einem Fehlzugriff auf einen Adressbereich **A[0]** mit hoher Wahrscheinlichkeit der nächste Zugriff auf eine Adresse in unmittelbarer Nachbarschaft z.B. **A[1]** erfolgt
 - Effizienter ganzen Block mit n Worten zu laden anstatt n Mal 1 Wort



Cache Begriffe

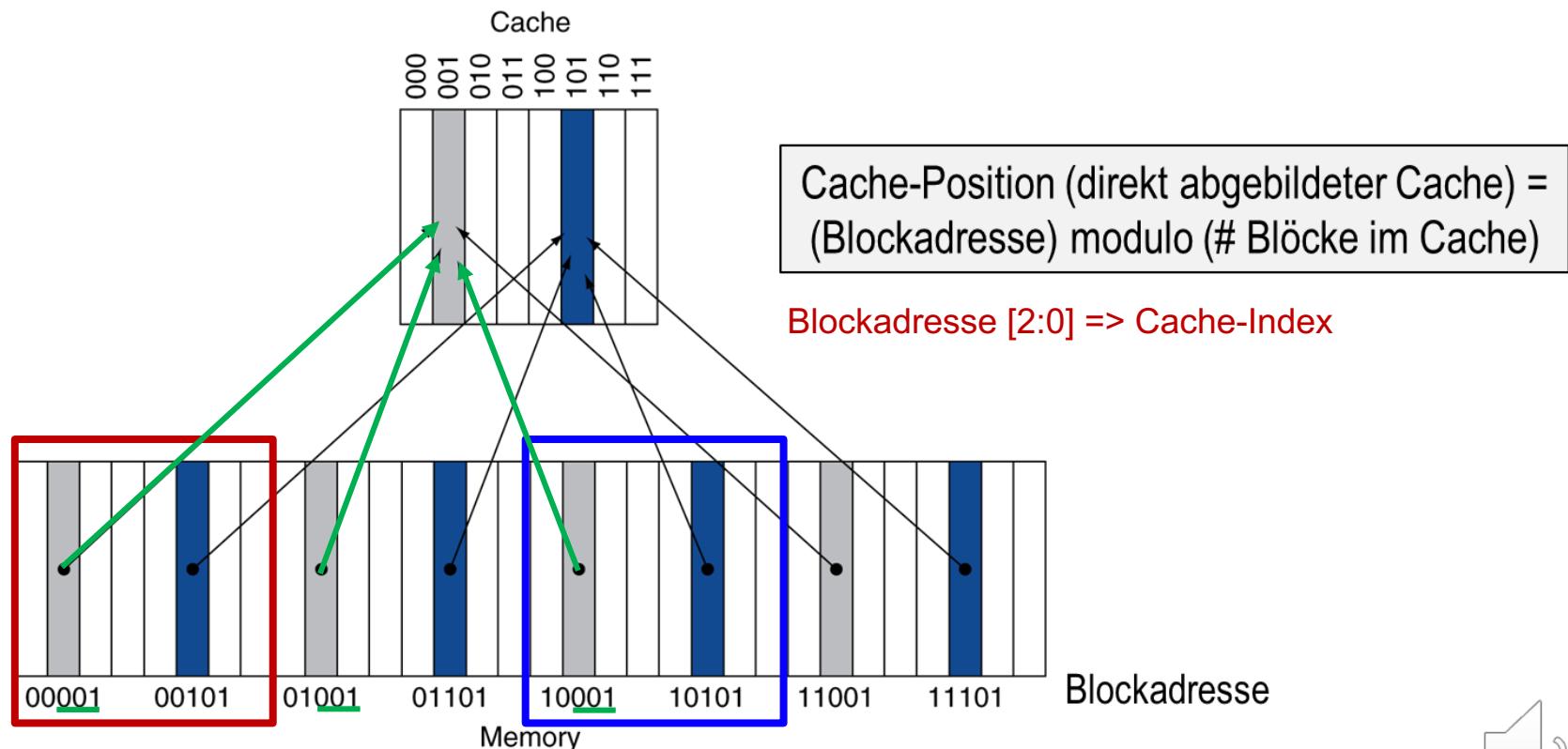
- **Block oder Zeile (*cache line*)**: Informationseinheit, die zwischen 2 Ebenen transportiert wird
- **Treffer (*hit*)**: angeforderte Daten werden auf der oberen Ebene gefunden
- **Fehlzugriff (*miss*)**: angeforderte Daten nicht gefunden
- **Trefferrate (*hit rate*)**: # Treffer / # Zugriffe
- **Fehlzugriffsrate (*miss rate*)**: # Fehlzugriffe / # Zugriffe = 1 – Trefferrate

Cache-Leistung

- Trefferrate **keine** gute Indikation der Cache-Leistung
 - Cache mit höhere Trefferrate kann schlechtere Leistung aufweisen
- **Durchschnittliche Zugriffszeit** (*average memory access time*)
 - **AMAT = hit time + miss rate * miss penalty**
 - Zugriffszeit bei Treffer (*hit time*): Zeit für einen Treffer (inkl. feststellen ob Treffer)
 - **Fehlzugriffsaufwand** (*miss penalty*): Zeit die benötigt wird um Block von unteren Ebene in höhere Ebene zu laden

Wo wird ein Block im Cache abgelegt?

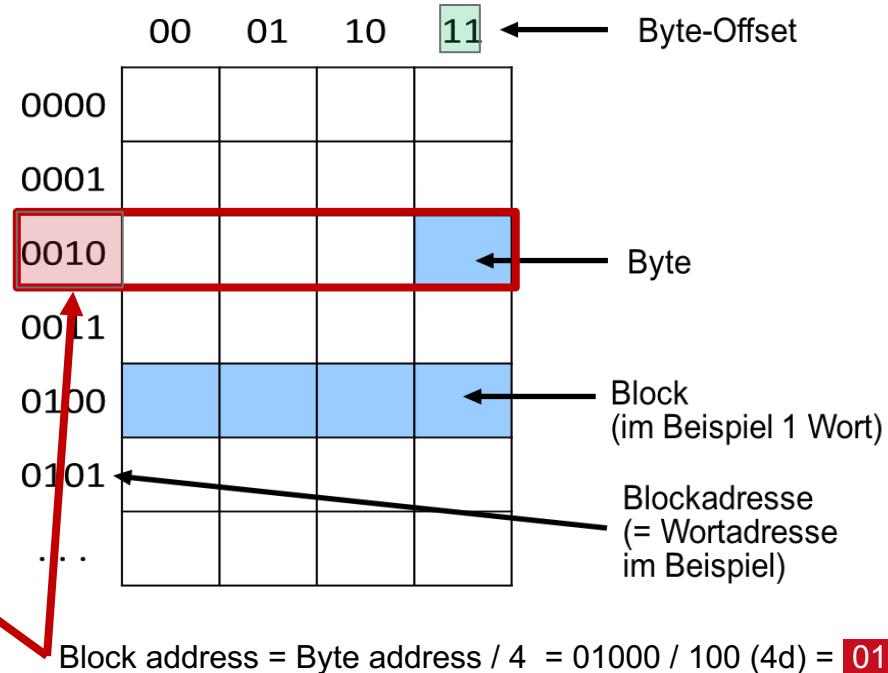
- Cache-Position ist abhängig von Adresse des Blocks im Speicher
 - Einfachste Methode: **Direkt abgebildeter Cache** (*direct-mapped cache*): jede Speicheradresse wird auf genau eine Cache-Position abgebildet



Blockadresse

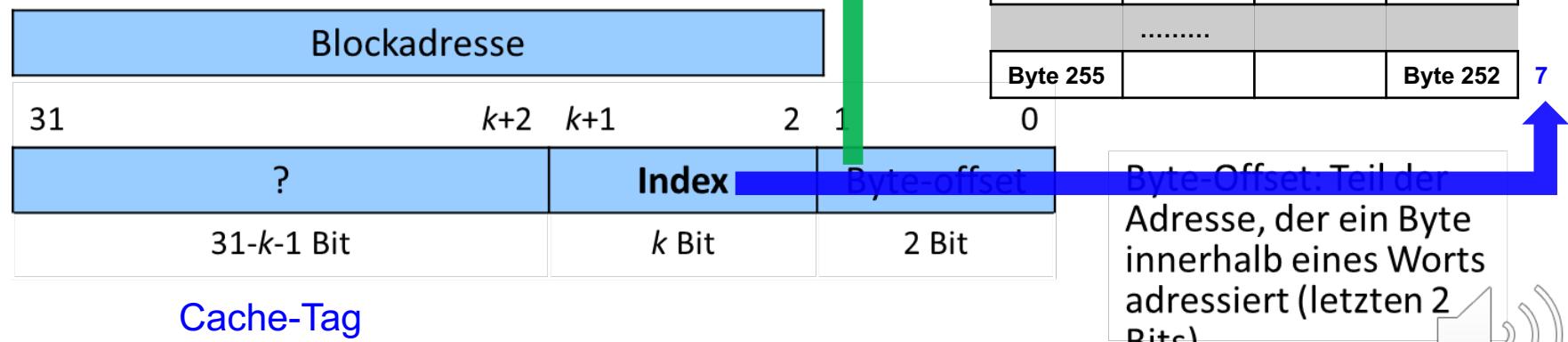
- Blockadresse = (Byteadresse) / (Blockgröße in Bytes) [=Integer Division]
- Beispiel (für 4-Byte (= 1 Wort) Blöcke): Cache-Line von 4 Bytes (schreiben & lesen nur mit Startadressen 0,4,8,12,16, ...)

byte address		block address	
decimal	binary	binary	decimal
0	00000	000	0
1	00001	000	0
2	00010	000	0
3	00011	000	0
4	00100	001	1
5	00101	001	1
6	00110	001	1
7	00111	001	1
8	01000	010	2
9	01001	010	2
10	01010	010	2
11	01011	010	2
12	01100	011	3
13	01101	011	3
...			



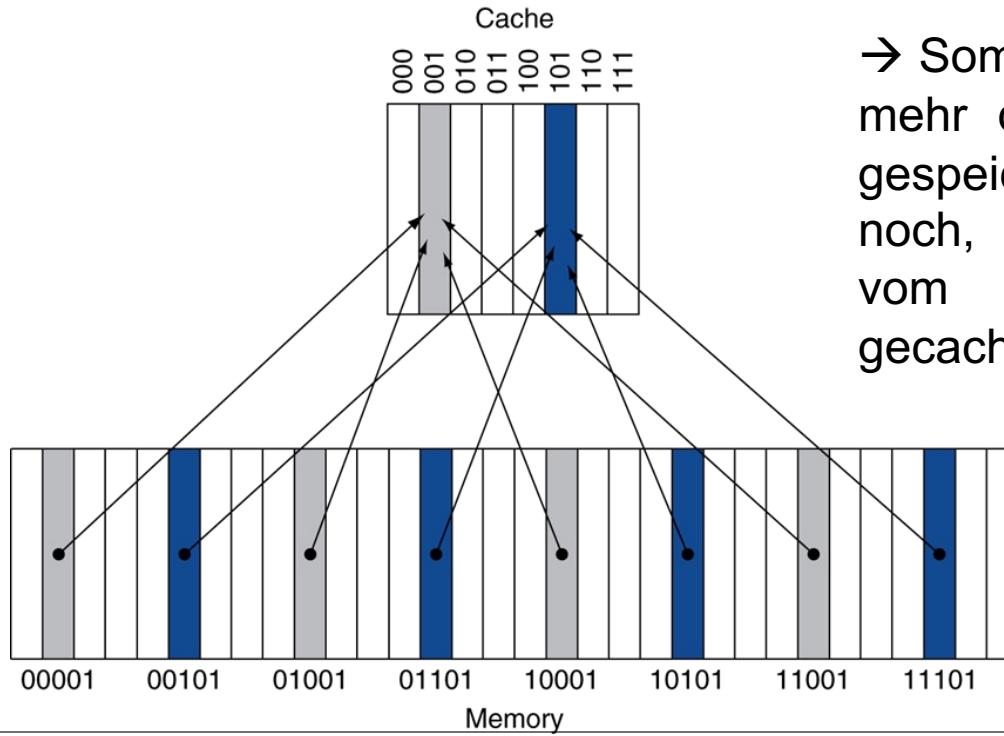
Caches mit direkt Abbildung

- Direkt abgebildete Caches verwenden Abbildung
 - Cache-Position = Cache-Index = (Blockadresse) modulo (# Blöcke im Cache)
- Wenn (# Blöcke im Cache) = $n = 2^k$
 - $k = \log_2 n \rightarrow$ unteren Bits der Blockadresse bilden den Cache-Index
- Beispiel:
 - Blockadresse = 34 = 100010
 - # Blöcke im Cache = 8 = 2^3
 - Cache-Index = $34 \bmod 8 = 2 = 010$



Finden von Daten im Cache

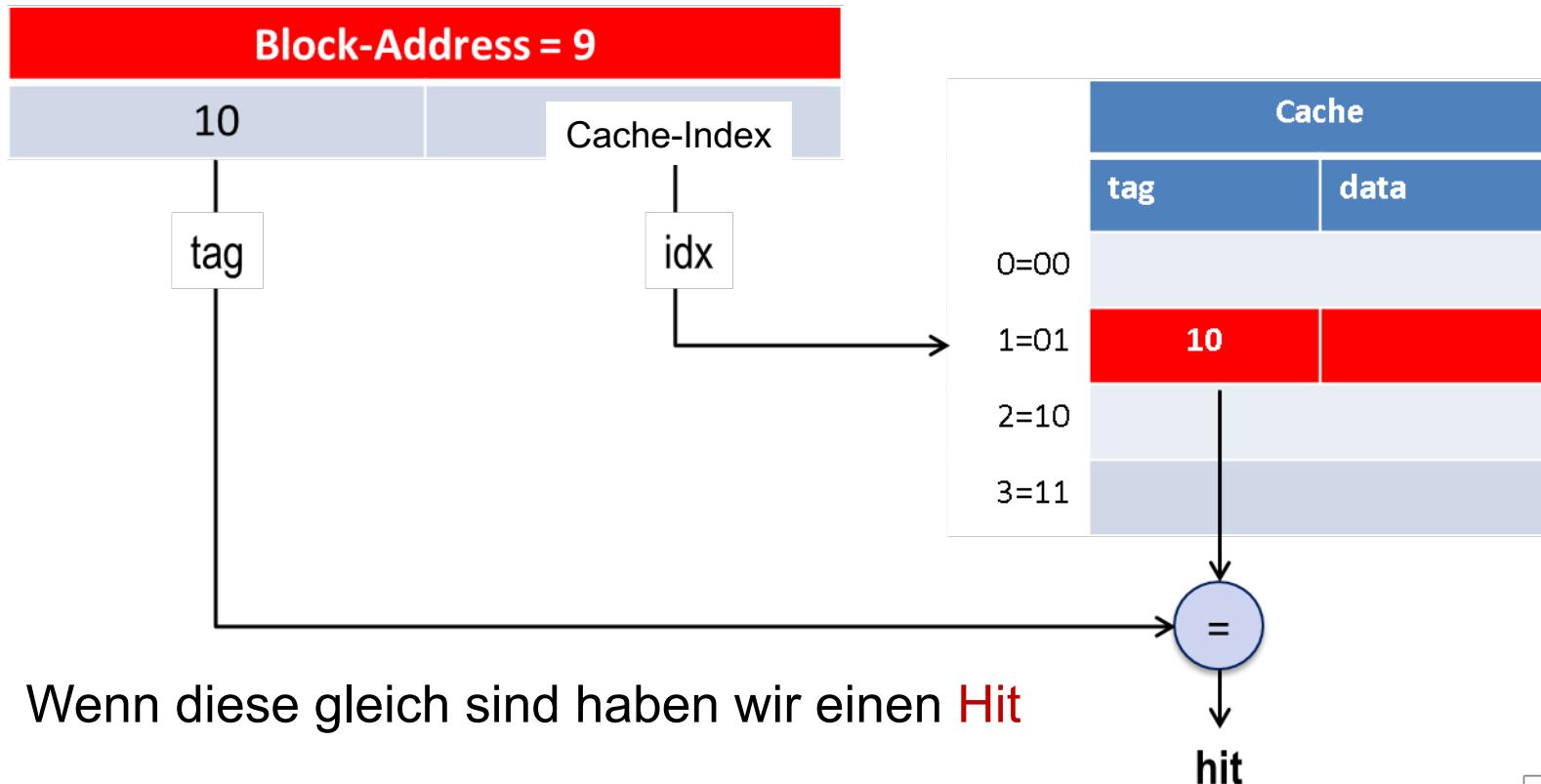
- Jede Cache-Position kann mehrere Speicherblöcke enthalten
- Wie wissen wir, ob Daten im Cache angeforderte Daten entsprechen?
 - **Tag**: Adressinformation nötig um zu erkennen ob Daten im Cache angeforderte Daten entsprechen
 - Obere Teil der Adresse, der **nicht** als Index für den Cache verwendet wird



→ Somit muss im **Adresstag** nicht mehr die gesamte Startadresse gespeichert werden, sondern nur noch, der wievielte Datenblock vom Hintergrund-medium gecached wurde

Beispiel: Finden von Daten im Cache

- Um Daten zu **finden** bzw. zu **prüfen** ob diese den angeforderten entsprechen → vergleiche oberen Teil der Blockadresse (**tag**) mit dem durch den Cache-Index definierten Inhalt des Caches



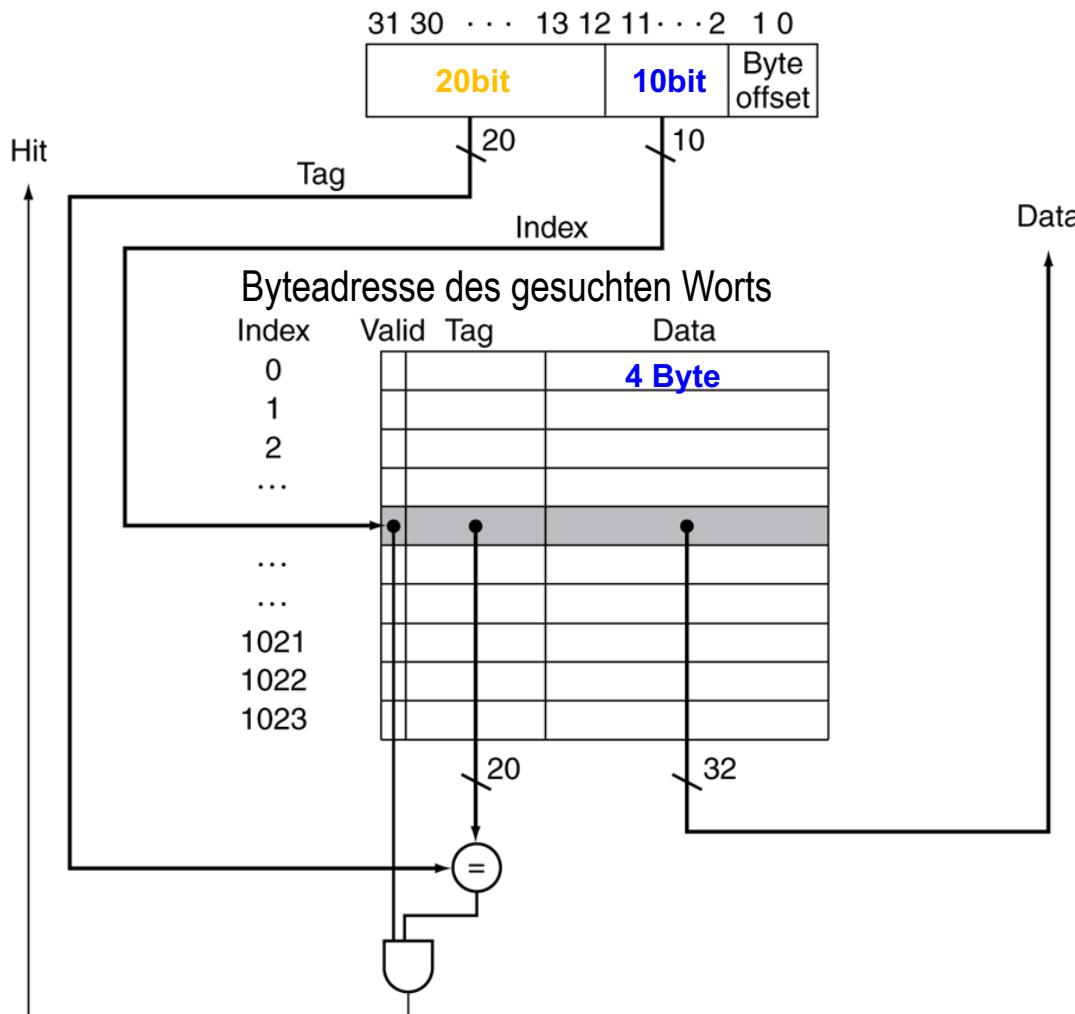
- Wenn diese gleich sind haben wir einen **Hit**

Gültigkeits-Bit

- Müssen erkennen können, ob Cache-Block gültige Information enthält
 - Z. B. beim Programmstart
 - Beim Prozesswechsel (*context switch*)
- Jeder Cache-Block hat **Gültigkeits-Bit** (*valid bit*)
 - gibt an, ob Block gültige Daten enthält
- Block im Cache (Cache-Block):



Direkt abgebildeter (DA) 4KB Cache



- Blockgröße = $2^2 = 4$ Bytes
- # Cache-Positionen = Index = # Blöcke = $2^{10} = 1024$
- Cachegröße = $1024 \times 4 = 4096 = 4\text{KB}$ ($1\text{K} = 1024$)
- Tag-Größe = $32 - 10 - 2 = 20$ Bit

Cache Gleichungen für DA Caches

- BlockAddress = ByteAddress / BytesPerBlock ($/$ = integer division)
- Cache-Index = BlockAddress % #CacheBlocks ($\%$ = modulo)
- Tag = BlockAddress / #CacheBlocks



- Block-offset = Teil der Adresse, der ein Byte innerhalb eines **Blocks** adressiert (unteren $b = \log_2(\text{BytesPerBlock})$ Bits)

Treffer/Fehlzugriff - Beispiel / 1

- Direkt abgebildeter Cache, anfänglich leer
- 8 Wörter groß, Blockgröße = 1 Wort (4 Byte)
- Gib für nächste Folge von Speicherzugriffen an
 - ob Treffer oder Fehlzugriff
 - Cache-Inhalt nach jedem Zugriff

Byteadresse	Blockadresse	Blockadresse Binär	Cache-Index	
88	$88/4 = 22$	10110	$22 \bmod 8 = 6$	Miss
104	$104/4 = 26$	11010	$26 \bmod 8 = 2$	Miss
88	$88/4 = 22$	10110	$22 \bmod 8 = 6$	Hit
104	$104/4 = 26$	11010	$26 \bmod 8 = 2$	Hit
64	$64/4 = 16$	10000	$16 \bmod 8 = 0$	Miss
12	$12/4 = 3$	00011	$3 \bmod 8 = 3$	
64	$64/4 = 16$	10000	$16 \bmod 8 = 0$	
74	$74/4 = 18$	10010	$18 \bmod 8 = 2$	

Beispiel / 2

- Anfangszustand:

Index	Valid	Tag	Daten
0	N		
1	N		
2	J	11=3	MemBlock[26]
3	N		
4	N		
5	N		
6	J	10=2	MemBlock[22]
7	N		

- Nächste Zugriff:

Blockgrösse = 4
Anzahl Blöcke im Cache = 8

Byteadr.	Blockadr.	Binär	Cache-Idx	Tag	T/F
88	88/4 = 22	1011 0	22 mod 8 = 6	22/8= 2	F
104	104/4 = 26	1101 0	26 mod 8 = 2	26/8= 3	F

Beispiel / 3

- Inhalt:

Index	Valid	Tag	Daten
0	J	10=2	MemBlock[16]
1	N		
2	J	11=3	MemBlock[26]
3	J	00=0	MemBlock[3]
4	N		
5	N		
6	J	10=2	MemBlock[22]
7	N		

- Nächste Zugriffe:

Byteadr.	Blockadr.	Binär	Cache-Idx	Tag	T/F
64	64/4 = 16	1000 0	16 mod 8 = 0	16/8= 2	F
12	12/4 = 3	0001 1	3 mod 8 = 3	3/8=0	F

Beispiel / 4

- Inhalt:

Index	Valid	Tag	Daten
0	J	10=2	MemBlock[16]
1	N		
2	J	11=3	MemBlock[26]
3	J	00=0	MemBlock[3]
4	N		
5	N		
6	J	10=2	MemBlock[22]
7	N		

- Nächste Zugriffe:

Byteadr.	Blockadr.	Binär	Cache-Idx	Tag	T/F
64	$64/4 = 16$	1000 0	$16 \bmod 8 =$ 0	$16/8 =$ 2	T
72	$74/4 = 18$	1001 0	$18 \bmod 8 =$ 2	$18/8 =$ 2	F

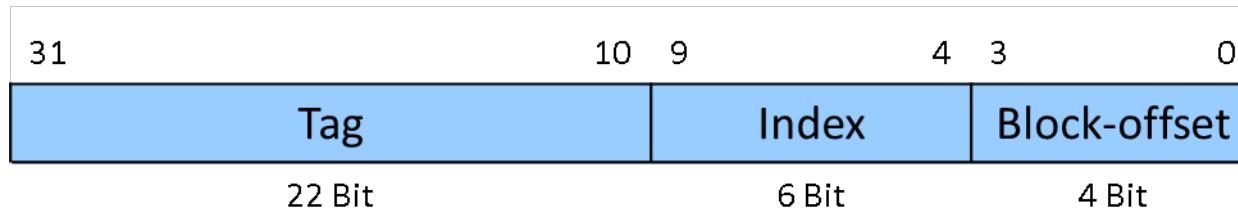
Beispiel / 5

- Folglich ergibt sich von dem anfänglich leeren Cache, durch die entsprechenden Nachladezyklen folgender Inhalt
- Inhalt:

Index	Valid	Tag	Daten
0	J	10=2	MemBlock[16]
1	N		
2	J	10=2	MemBlock[18]
3	J	00=0	MemBlock[3]
4	N		
5	N		
6	J	10=2	MemBlock[22]
7	N		

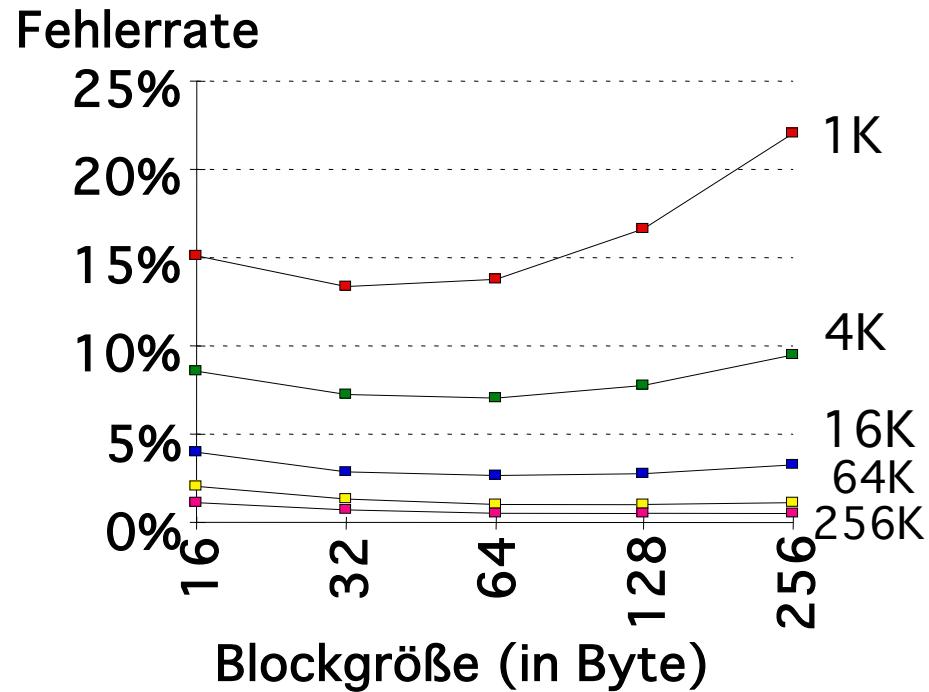
Größere Blöcke

- Blöcke von 1 Wort nutzen räumliche Lokalität **nicht** aus
- Größerer Blöcke nutzen räumliche Lokalität um Fehlzugriffsrate zu senken
 - Typische Blockgröße: 32-64 Bytes
 - Block von *n Wörtern auf einmal* laden kostet weniger Zeit als *n x 1 Wort laden*
- Gleichungen ändern sich nicht:
 - Blockadresse = (Byteadresse) / (Bytes pro Block)
 - Cache-Index = (Blockadresse) modulo (#Cache-Blöcke)
- Beispiel: direkt abgebildeter Cache mit **64** Blöcken je **16-Byte**
 - Frage: Auf welchen Cache-Index wird Byteadresse 1204 abgebildet?
 - Blockadresse = $1204 / 16 = 75$
 - Cache-Index = $75 \text{ modulo } 64 = 11$



Optimale Blockgröße – Abwägungen / 1

- Größere Blöcke senken die Fehlerrate
 - Aufgrund räumlicher Lokalität
- Aber wenn zu groß im Verhältnis zu Cache-Größe:
 - Größere Blöcke → weniger Blöcke → mehr Wettbewerb → erhöhte Fehlerrate
- Größter Nachteil ist, dass der **Fehlzugriffsaufwand steigt**



Bockgröße – Abwägungen / 2

- Größere Blöcke steigern Fehlzugriffsaufwand
- Annahmen
 - hit time = 1 Taktzyklus
 - miss penalty = 10 + (Wörter pro Block) Taktzyklen
 - miss rate:

Blockgröße	4B	8B	16B	32B	64B	128B
Fehlzugriffsrate	10%	6%	4%	3%	2.5%	2.3%

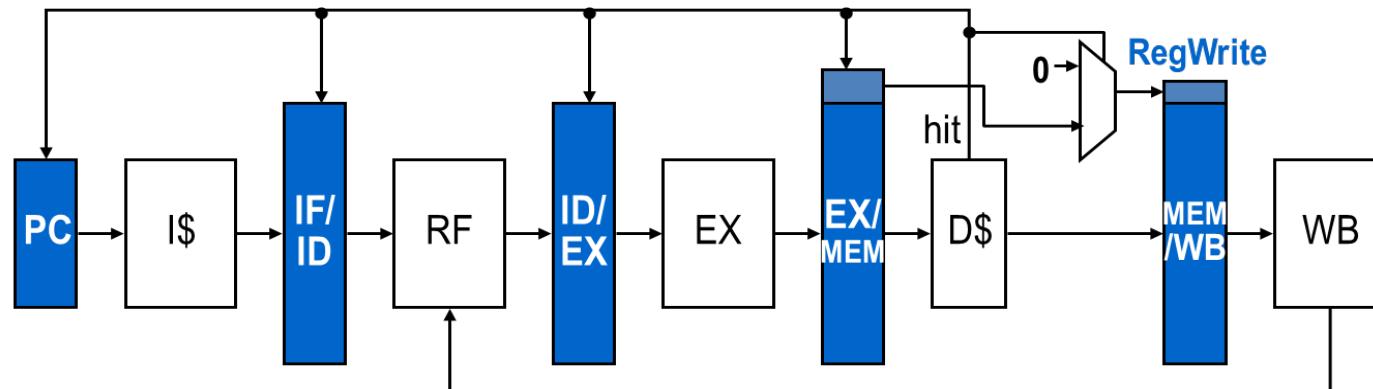
- Welche Blockgröße hat minimale AMAT?
 - AMAT = (hit time) + (miss rate) x (miss penalty)?

AMAT = average memory access time

Blockgröße	4B	8B	16B	32B	64B	128B
miss penalty	11	12	14	18	26	42
AMAT	2.1	1.72	1.56	1.54	1.65	1.97

Verarbeitung von Fehlzugriffen

- Miss → Pipeline verzögern/stall, Block aus nächster Hierarchieebene holen
- Befehls-Chefzugriff
 - Ursprünglicher PC (aktueller PC-4) an Speicher senden, IF/ID Register löschen
 - Warten bis Speicher Leseoperation abschließt
 - Cache-Block füllen, tag & valid bit setzen
 - Befehl erneut laden und dann Instruktion ausführen
- Fehlerhafter Datenzugriff → Pipeline verzögern bis Datenzugriff vollendet

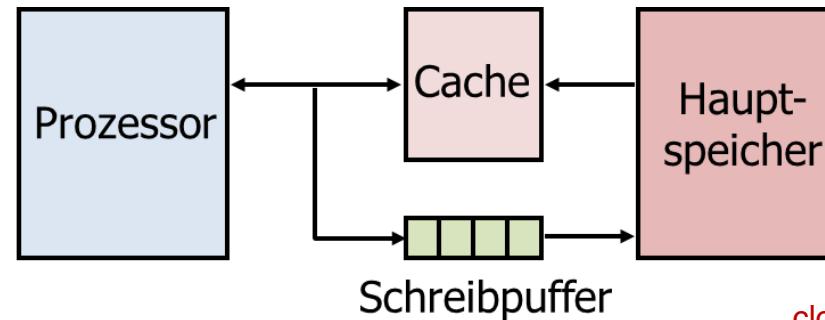


Schreiboperationen

- Schreiben wir neue Daten nur in den Cache oder auch in den Speicher?
- Beide Methoden sind möglich:
 - Durchschreibetechnik (*write-through*)
 - sowohl in den Cache als auch in den Speicher
 - Vorteil: Cache und Speicher sind konsistent
 - Nachteil: schlechte Leistung (CPU wird angehalten bis Schreiboperation fertig)
 - Rückschreibetechnik (*write-back*)
 - neue Daten werden nur in den Cache geschrieben
 - Vorteil: bessere Leistung da mehrere Schreiboperationen zum gleichen Block kombiniert werden
 - Nachteile: Cache und Speicher inkonsistent, Implementierung komplexer

Durchschreibetechnik - Schreibpuffer

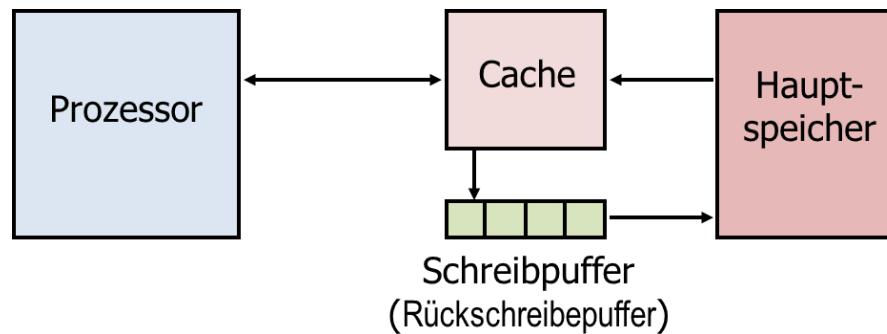
- Durchschreibetechnik: schlechtere CPU Leistung
- Beispiel:
 - CPI ohne Cache-Fehlzugriffe: 1,0
 - Schreiboperation: 100 Taktzyklen (Hauptspeicher ist langsam!)
 - 10% der Befehle Schreiboperationen (SPEC2000)
 - $CPI = 1,0 + 0,1 \times 100 = 11 \rightarrow 10x$ langsamer
- *Idee: Verbesserung durch Schreibpuffer (write buffer):*
 - Speichert Daten bis sie in den Hauptspeicher geschrieben werden
 - CPU macht weiter und hält nur an wenn Schreibpuffer voll



clock cycles per instruction ==

Rückschreibetechnik – dirty bit

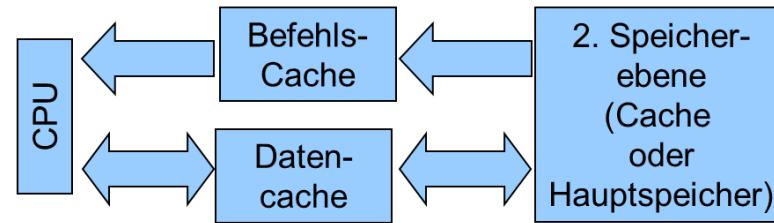
- Brauchen Block nur zurückzuschreiben, wenn er verändert wurde
 - *Dirty Bit*: gibt an ob Block geschrieben wurde
- Rückschreibe-Caches haben Schreibpuffer für ausgetauschte „dirty-Blöcke“
 - auch Rückschreibepuffer (*write-back buffer*) genannt



- Write-through Cache kann direkt in Hauptspeicher schreiben
- Write-back Cache muss Daten vorher im Cache haben (d.h. ggf. laden)

Beispiel: Intrinsity FastMATH CPU / 1

- Eingebetteter MIPS-Prozessor mit 12-Stufige Pipeline + Cache
- Kann in jedem Takt ein Befehl und ein Datenwort anfordern
 - Getrennte Befehls- und Daten Cache (*split instruction/data cache*)
 - Verdopplung der Cache-Bandbreite
 - Befehls-Cache könnte größere Blöcke nutzen



Fehlzugriffsraten Intrinsity FastMATH mit SPEC2K-Benchmarks

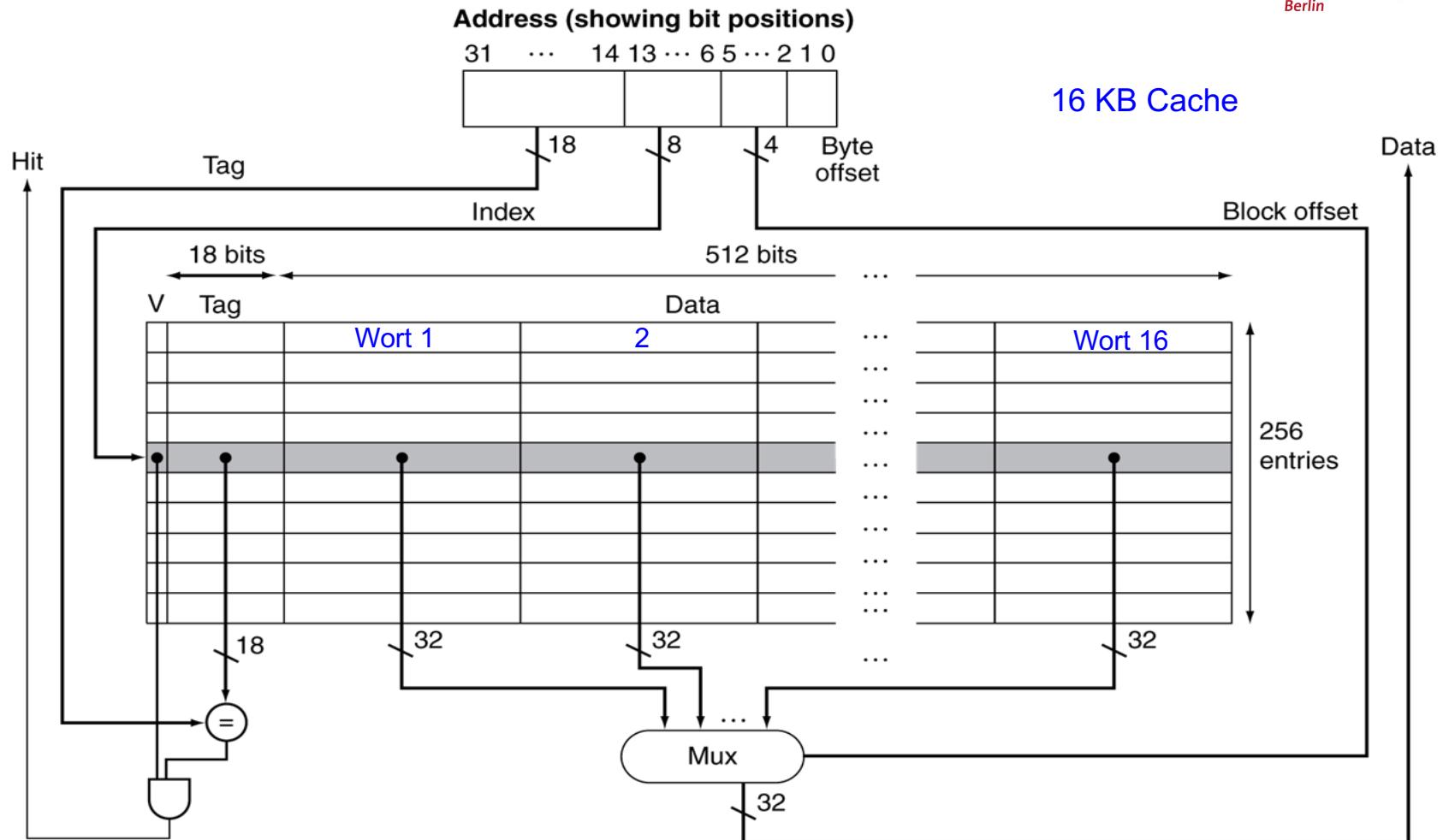
Fehlzugriffsrate I\$ (16 KB)	Fehlzugriffsrate D\$ (16 KB)	Effektive Fehlzugriffsrate
0,4%	11,4%	3,2%

Beispiel: Intrinsity FastMATH CPU / 2

- Sowohl Befehls- als auch Daten-Cache 16 KB groß ($1K = 1024$)
- Direkt abgebildet
- Blockgröße 64 Byte (16 Wort-Block) → 6bit Offset
- Adresslänge 32 Bit

- Wie groß ist der Index und wie groß der Tag?
 - #Blöcke = $\frac{Cache - Größe}{Blockgröße} + \frac{16KB}{64B} = \frac{16 * 1024}{64} = 256$
 - Index = $\log_2(256) = 8$ Bit
 - Tag-Größe = Adresslänge – (Index-Größe) – Block-Offset
= 32 – 8 – 6 = 18 Bit
 - Block-Offset = $\log_2(Blockgröße) = \log_2(64) = 6$

Beispiel: Intrinsity FastMATH CPU / 3



Komponenten einer Adresse

Blockadresse		Block-offset	
Tag	Index	Wort-offset	Byte-offset

- **Byte-offset:** letzte 2 Bit, die einen Byte im Wort adressieren
- **Wort-offset:** nächste $\log_2(\text{Blockgröße in Wörter})$ Bit,
die ein Wort im Block adressieren
- **Block-offset:** Wort-offset und Block-offset zusammen
 $(\log_2(\text{Blockgröße in Byte}))$ Bit)
- **Index:** gibt Position im Cache an
- **Tag:** übrige Bit

Lehrbuch (3. Auflage) undeutlich über Block-Offset

- manchmal was wir Wort-Offset nennen (Abb. 7.8)
- manchmal was wir Block-Offset nennen (Abb. 7.12)

Implementierungskosten

- Wie viele Bits braucht man **insgesamt** für einen direkt abgebildeten Cache mit 16 KB Daten und Blöcke von 4 Wörtern bei 32-Bit-Adressen?
 - Cache : $16 \times 1024 / (4 \times 4) = 1024$ Blöcken
 - Block: $4 \times 32 = 128$ Bit Daten (+ Tag und Gültigkeits-Bit)
 - Tag: $32 - 10$ $- 2$ $- 2 = 18$ Bit

Cache-Index Byte-Offset Wort-Offset
 - Bits insgesamt: $1024 \times (128 + 18 + 1)$ Bit = 147KBit = 18,4 KByte

Valid Bit