

1 Einheiten, Größen

1.1 Datenmengen

← Binärpräfix	Dezimalpräfix →					
		B	kB	MB	GB	TB
	B	1	10 ³	10 ⁶	10 ⁹	10 ¹²
	KiB	2 ¹⁰	1	10 ³	10 ⁶	10 ⁹
	MiB	2 ²⁰	2 ¹⁰	1	10 ³	10 ⁶
	GiB	2 ³⁰	2 ²⁰	2 ¹⁰	1	10 ³
	TiB	2 ⁴⁰	2 ³⁰	2 ²⁰	2 ¹⁰	1

1.2 Frquenz f

f = 1 / T

2 Leistungsbewertung

MIPS millions of instructions per second
FLOPS floating point operations per second

Wichtig:

MIPS-Vergleiche ergeben lediglich Sinn innerhalb der gleichen ISA (Instruction Set Architecture).

MIPS = 10^3 / Instruktionszeit in ns

2.1 Amdahlsches Gesetz

Die Beschleunigung eines Systems wird berechnet durch ein vergleich der Zeit:

Beschleunigung = (Zeit vor der Beschleunigung) / (Zeit nach der Beschleunigung)

Damit kann man folgende Gesetzmäßigkeit aufstellen:

S = 1 / ((1 - p) + p/s)

Aus diesem Zusammenhang kann man nun auch folgendes herleiten:

T0 / Ts = 1 / ((1 - p) + p/s)

S = (Gesamt-)Beschleunigung des Programmes
s = Beschleunigung des Programnteils welches von der Verbesserung profitiert
p = Anteil der Zeit, in dem die Verbesserung benutzt wird
T0 = Zeit vor der Beschleunigung
Ts = Zeit nach der Beschleunigung

3 Fehlertoleranz

3.1 N-Modulare-Redundanz

Die Wahrscheinlichkeit eines fehlerfreien Funktionieren eines Systemes p_s , mit m Komponenten, ist abhängig von der Verfügbarkeit der Einzelkomponenten p_c und der Verfügbarkeit des Voters p_v . Für einen perfekten bzw. nicht vorkommenden Voter ist $p_v = 1$. Die maximale Anzahl der ausfallenden Komponenten w , wird falls nicht gegeben, durch $w = \lceil \frac{m}{2} \rceil$ repräsentiert.

ps = sum_{i=0}^w (C(m, i) * p_c^{m-i} * (1 - p_c)^i) * p_v

2MR Wahrscheinlichkeit des fehlerfreien Funktionierens eines 2MR Systems:
2 Komponenten mit der Verfügbarkeit p_c und einem Voter mit der Verfügbarkeit p_v :

ps = (2 * pc - pc^2) * pv

3MR Wahrscheinlichkeit des fehlerfreien Funktionierens eines 3MR Systems:
3 Komponenten mit der Verfügbarkeit p_c und einem Voter mit der Verfügbarkeit p_v :

ps = (3 * pc^2 - 2 * pc^3) * pv

4MR Wahrscheinlichkeit des fehlerfreien Funktionierens eines 4MR Systems:
4 Komponenten mit der Verfügbarkeit p_c und einem Voter mit der Verfügbarkeit p_v :

ps = (3 * pc^4 - 8 * pc^3 + 6 * pc^2) * pv

5MR Wahrscheinlichkeit des fehlerfreien Funktionierens eines 5MR Systems:
5 Komponenten mit der Verfügbarkeit p_c und einem Voter mit der Verfügbarkeit p_v :

ps = (6 * pc^5 - 15 * pc^4 + 10 * pc^3) * pv

4 Paritätsprüfung

4.1 Paritätsbit

Die Wahrscheinlichkeit das ein Wort der Länge n wirklich fehlerfrei ist, wird bereits urch die Fehlererkennung mittels eines Paritätsbits erhöht. Dabei ist die *Wahrscheinlichkeit für ein* korrektes Bit p , wobei gilt das $0 < p < 1$ ist. Die Bitfolge mit der Länge $n + 1$ wird dann als korrekt angesehen wenn alle $n + 1$ oder n Bits korrekt sind. Die Wahrscheinlichkeit für das Wort mit der Länge n und einem Paritätsbit wird wie folgt berechnet:

Pparity = p^{n+1} + C(n, 1) * p^n * (1 - p)
= p^{n+1} + (n + 1) * (p^n - p^{n+1})
= (n + 1) * p^n - n * p^{n+1}

Für ein Wort der Länge n ist die Wahrscheinlichkeit das es korrekt angesehen wird:

Pno_parity = p^n

5 Hamming-Code

5.1 Anzahl der Prüf- und Datenbits

Für ein Wort der länge n mit m Datenbits benötigt man r Prüfbits um einen 1-Bit-Fehler pro Wort zu korrigieren. Daraus ergibt sich die allgemeine Ungleichung:

2^r ≥ m + r + 1

Und für einen *perfekten* Hamming-Code ergibt sich:

2^r = m + r + 1

Der *perfekte* Hamming-Code besitzt die Wortlänge $2^r - 1$.

5.2 Hamming-Abstand

Der Hamming-Abstand gibt die Anzahl der Bits zwischen zwei beliebige Wörtern aus dem Code.

Beispiel:

Die zwei Wörter 1100100100₂ und 1001100011₂ haben den Hamming-Abstand $h = 5$.

Hamming-Abstand eines Codes

Der Hamming-Abstand eines Codes ist die kleinstmögliche Anzahl an Bits, die man verändern muss um ein neues gültiges Wort aus dem vorliegenden Code zu bekommen.

Beispiel:

x = 00110 hxy = 2
y = 00101 hxz = 1
z = 01110 hyz = 3

Da der kleinste Abstand 1 ist, ist der Hamming-Abstand des Codes ebenfalls 1.

5.3 Fehlererkennung

Um d Bitfehler zu erkennen, braucht man den Hamming-Abstand h :

h = d + 1
d = h - 1

5.4 Fehlerkorrektur

Um d Bitfehler zu korrigieren, braucht man den Hamming-Abstand h :

h = 2d + 1
d = floor((h - 1) / 2)

5.5 Bündelfehlerkorrektur

Um mehrere aufeinanderfolgende Bitfehler zu korrigieren ordnet man die k mit Hamming codierten Wörter der Länge n in einer $k \cdot n$ Matrix an. *Dabei ist zu beachten das nun zuerst die erste Spalte der Matrix übertragen bzw. gespeichert wird.*

Somit kann man einen **maximal** einen Bündelfehler der Länge k Bits korrigieren, dabei ist bei k aufeinanderfolgenden fehlerhaften Bits maximal 1 Bit pro Wort fehlerhaft.

Beispiel:

	ASCII	Codewort											
H	1 0 0 1 0 0 0	0	0	1	1	0 0 1	0	0 0 0	0	0 0 0	0	0 0 0	0
a	1 1 0 0 0 0 1	1	0	1	1	1 0 0	1	0 0 1	1	0 0 1	1	0 0 1	1
m	1 1 0 1 1 0 1	1	1	1	0	1 0 1	0	1 0 0	0	1 0 0	0	1 0 0	0
m	1 1 0 1 1 0 1	1	1	1	0	1 0 1	0	1 0 1	0	1 0 1	0	1 0 1	0
i	1 1 0 1 0 0 1	0	1	1	0	1 0 1	1	0 0 1	1	0 0 1	1	0 0 1	1
n	1 1 0 1 1 1 0	0	1	1	0	1 0 1	0	1 1 0	0	1 1 0	0	1 1 0	0
g	1 1 0 0 1 1 1	0	1	1	1	1 0 0	1	1 1 1	1	1 1 1	1	1 1 1	1

Die übertragenen Bits lauten:
0111000 0011111 1111111 1100001 0111111
0000000
1011110 0100101 0011011 0000011 0100101

5.6 Hamming-Algorithmus

Paritäts-Kontrollbits

Paritäts-Kontrollbits p an Zweierpotenz Positionen

- Paritätsbit p_1 prüft 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21
- Paritätsbit p_2 prüft 2, 3, 6, 7, 10, 11, 14, 15, 18, 19
- Paritätsbit p_4 prüft 4, 5, 6, 7, 12, 13, 14, 15, 20, 21
- Paritätsbit p_8 prüft 8, 9, 10, 11, 12, 13, 14, 15
- Paritätsbit p_{16} prüft 16, 17, 18, 19, 20, 21

Beispiel:

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Wert	?	?	1	?	?	1	1	?	0	0	0	0	0	1	0	1	?	0	1	1	1
+ Parität	0	0	1	0	1	1	1	0	0	0	0	0	1	0	1	1	0	1	1	1	1

5.7 Erweiterungen

SECDED

Single Error Correction Double Error Detection füget ein weiteres Paritätsbit an der Position 0 welches die gesamten anderen Bits auf Parität überprüft.

- Hamming-Prüfbits ✓ Extra-Prüfbit ✓
→ fehlerfrei oder mehr als 2 Fehler
- Hamming-Prüfbits ✗ Extra-Prüfbit ✗
→ 1-Bitfehler, korrigierbar mit Hamming
- Hamming-Prüfbits ✗ Extra-Prüfbit ✓
→ 2-Bitfehler, nicht lokalisier- bzw. korrigierbar

6 IJVM

6.1 Ausführungsdauer

Ausführungszeit = (μ-Instruktionen) / Takt

Wichtig:

Bei der Auswärtung der μ -Instruktionenist darauf zu achten dass der Programm fluss beachtet wird, denn

je nach Aufbau des Programmes kann es bei bedingten Sprüngen in der Architektur sein das Befehle eine unterschiedliche Anzahl an μ -Instruktionen besitzt. Ebenfalls muss beachtet werden ob innerhalb der μ -Instruktionen auch Sprünge stattfinden, welche zusätzliche Operation zum selben Takt haben.

6.2 Anweisungsabhängigkeit

Read after Write (RAW) Instruktion 2 versucht aus dem Register einen Wert zu lesen bevor dieser durch Instruktion 1 geschrieben wurde. Diese Art der Abhängigkeit ist entscheidend für Aussagen über die Mindestanzahl der benötigten Taktzyklen. Für die Aufteilung der einzelnen Instruktionen auf die verschiedenen Ausführungseinheiten eines superskalaren Prozessors werden Abhängigkeitsketten gebildet, wobei die Anzahl der Glieder in der längsten Kette der Mindestanzahl der benötigten Taktzyklen entspricht.

$$\mathbf{R1} = \mathbf{R2} + \mathbf{R3} \quad (1)$$

$$\mathbf{R4} = \mathbf{R1} + \mathbf{R5} \quad (2)$$

Write after Read (WAR) Instruktion 2 versucht in ein Register einen Wert zu schreiben bevor es von Instruktion 1 gelesen wurde.

$$\mathbf{R1} = \mathbf{R2} + \mathbf{R3} \quad (1)$$

$$\mathbf{R2} = \mathbf{R5} + \mathbf{R6} \quad (2)$$

Write after Write (WAW) Instruktion 2 versucht in ein Register einen Wert zu schreiben bevor in dieses ein Wert von Instruktion 1 geschrieben wurde.

$$\mathbf{R1} = \mathbf{R2} + \mathbf{R3} \quad (1)$$

$$\mathbf{R1} = \mathbf{R5} + \mathbf{R6} \quad (2)$$

6.3 Adressierungsarten

Unmittelbare Adressierung Bei der unmittelbaren Adressierung folgt ein Operand direkt dem Opcode. Er wird als Konstante betrachtet.
LOAD 20 \rightarrow 20

Direkte Adressierung Bei der direkten Adressierung wird der Operand der dem Befehl folgt oder das angegebene Register als Adresse angesehen.
LOAD 20 \rightarrow 40

Indirekte Adressierung Bei der indirekten Adressierung enthält die Speicherzelle dagegen erneut eine Adresse und erst die Speicherzelle die durch diese Adresse angesprochen wird den Wert selbst.
LOAD 20 \rightarrow 60
LOAD R2 \rightarrow 40

Registeradressierung Bei der Registeradressierung ist der Operand ein Register, welches den Wert selbst beinhaltet.
LOAD R2 \rightarrow 20

Indizierte Adressierung Dies entspricht der direkten Adressierung, nur wird vor dem Zugriff zu der Adresse noch ein fester Wert addiert und das Resultat als Adresse interpretiert.
LOAD (R1 + 40) \rightarrow 70

Anstelle des festen Wert kann auch ein Indexregister zu einem Basisregister addiert werden, dies liefert dann die Adresse für den direkten oder indirekten Zugriff.
LOAD (R1 + R2) \rightarrow 50

Beispielwerte:

Register	Wert	Adresse	Wert
R1	10	20	40
R2	20	30	50
		40	60
		50	70

7 HDD

7.1 Begriffe

Spur Ein Kreis auf der Festplatte auf dem die Sektoren nacheinander liegen. Die Spuren sind konzentrisch auf der Festplatte gelegt.

Sektor Ein Stück aus einer Spur mit fester Breite. Die Sektorlänge ist z.B. in Bytes (512 oder 4096 Bytes) oder Mikrometer angegeben. Jeder Sektor enthält zudem noch 40 bis 100 Bytes für Verwaltungsinformationen (*Präambel*) und Fehlererkennung/-korrektur.

Zylinder Wenn die Festplatte aus mehreren übereinander liegenden Scheiben besteht, besteht ein Zylinder aus allen Spuren die genau übereinander auf den Scheiben liegen. Wenn es z.B. 2 Scheiben gibt, dann sind alle 2 Spuren die ganz außen auf jeder Scheibe liegen ein Zylinder; genauso machen alle Spuren Nr. 16 auf jeder Scheibe zusammen ein Zylinder aus: Zylinder 16. Ein Zylinder ist also eine Erweiterung der Spur auf vertikalen Ebene wenn es mehrere Scheiben gibt. Es folgt, auf einem Zylinder der aus n übereinander liegenden Spuren besteht und wenn jede dieser Spuren k Bytes hat, dann hat ein Zylinder $n \cdot k$ Bytes. Folgt auch, dass eine Angabe wie 32 Zylinder bedeutet dass jede Scheibe 32 Spuren hat.

7.2 Datenübertragungszeit

Die mittlere Datenübertragungszeit T_{average} einer Festplatte hinsichtlich der auf einer Spur befindlichen Daten kann vereinfacht mittels der folgenden Funktion dargestellt werden:

$$T_{\text{average}} := T_s + T_r + T_t$$

Die mittlere Positionierungszeit T_s setzt sich dabei zusammen aus der Positionierungszeit MT_{sr} des

Schreib/Lese-Kopfs in die richtige radiale Position zum Lesen/Schreiben und der Zeit MT_{sc} zum wechseln auf benachbarte Spuren S_c , daraus leiten sich folgenden Funktionen ab:

$$T_s := T_{\text{sr}} + T_{\text{sc}}$$

$$T_{\text{sr}} := MT_{\text{sr}} \cdot (S - S_c)$$

$$T_{\text{sc}} := MT_{\text{sc}} \cdot S_c$$

Sollten jedoch alle Spuren S bis auf die erste benachbart sein, dann gilt für die Positionierungszeit folgenden Funktion:

$$T_s := MT_{\text{sr}} + MT_{\text{sc}} \cdot (S - 1)$$

Der Drehverzug T_r fällt bei jedem Spurwechsel an und ist die Verzögerung, welche anfällt bis sich der gesuchte Sektor auf der rotierenden Festplatte unter dem Schreib/Lese-Kopf befindet. Die Drehverzugszeit ist dabei abhängig von der Rotationsgeschwindigkeit v welche in *RPM* angegeben wird.

$$T_r := \left(\frac{60 \cdot 1000}{2v} \right) \cdot S$$

Die Übertragungszeit T_t ist die eigentliche Zeit welche benötigt wird und die Daten zu übertragen; diese ist abhängig von der Rotationsgeschwindigkeit v sowie der Anzahl der genutzten Sektoren N_u pro Spur und der Gesamtanzahl der Sektoren N_t pro Spur.

$$T_t := \left(\frac{60 \cdot 1000}{v} \right) \cdot \frac{N_u}{N_t} \cdot S$$

Die Anzahl der benötigten Spuren lässt wie folgt berechnen:

$$\text{Spuren} = \frac{\text{DateiGr}}{\text{Oberflächen}_{\text{Anz}} \cdot \text{Sektoren}_{\text{Anz}} \cdot \text{SektorGr}}$$

7.3 Speicherdirektzugriff

Der Speicherdirektzugriff ermöglicht es angeschlossenen Peripheriegeräten ohne Umweg über die CPU mit dem Arbeitsspeicher zu kommunizieren. Ein Vorteil des Speicherdirektzugriffs ist die schnellere Datenübertragung bei einer gleichzeitigen Entlastung des Prozessors. Der DMA-Controller muss jedoch die Daten zwangsläufig über das gleiche Bussystem übertragen wie die CPU.

Die Buszyklen C welche für einen n -Bit-DMA-Transfer benötigt werden kann mittels folgender Funktion, in Abhängigkeit von der Umdrehungsdauer T_r (in ms), der Sektorgröße S und der Sektorenanzahl A , ermittelt werden:

$$C = \left\lceil \frac{A \cdot S \cdot 8}{n \cdot T_r} \right\rceil$$

8 SSD

C = Lösch/Schreib-Zyklen pro Speicherzelle
 T_L = Lebensdauer
 B_G = Gesamtanzahl der Blöcke
 B_T = Anzahl der Blöcke welche pro Zeiteinheit geschrieben werden
 P_D = Prozentanteil der dynamischen Daten
 P_S = Prozentanteil der reservierten Blöcke für das Bad Block Management

$$B_G = \frac{\text{Kapazität}}{\text{Blockgröße}} \cdot (1 - P_S)$$

Wichtig:

Wenn es heißt: *Eine typische Datei, die zu den Dynamischen Daten gerechnet wird, belegt im Mittel x Blöcke ...*, dann muss wie folgt vorgegangen werden:
 A_{DBF} = Anzahl Dynamischer Blöcke pro Datei

$$\text{Anzahl}_{\text{Datei}} = \frac{\text{Anzahl}_{\text{Dynamischer Blöcke}}}{\text{Anzahl}_{\text{Dynamischer Blöcke pro Datei}}}$$

$$B_T = \text{Anz}_{\text{Dyn. Blöcke pro Datei}} \cdot \text{Anz}_{\text{Datei}} \cdot \text{Zeiteinheit}$$

Static Wear Levelling

$$T_L = \frac{C \cdot B_G}{B_T}$$

Dynamic Wear Levelling

$$T_L = \frac{C \cdot (B_G \cdot P_D)}{B_T}$$

9 Caches

9.1 Zugriffszeit

Die Trefferwahrscheinlichkeit für einen Wert in einem Level- i -Cache berechnet sich wie folgt:

$$\begin{aligned} P(L1) &= p_{L1} \\ P(L2) &= p_{L2} \cdot \underbrace{(1 - P(L1))}_{\text{L1 Miss}} \\ P(L3) &= p_{L3} \cdot \underbrace{(1 - P(L1) - P(L2))}_{\text{L1 und L2 Miss}} \\ P(\text{MEM}) &= 1 - \underbrace{(P(L1) - P(L2) - P(L3))}_{\text{nicht im Cache}} \end{aligned}$$

Dabei bezeichnet p_{L_i} den Prozentsatz der Speicher-Lesezugriff, die Cache-Treffer des Level- i -Caches sind. Die mittlere Zugriffszeit auf Werte aus dem Hauptspeicher bzw. dem Cache lässt sich dann wie folgt berechnen:

$$T_{\text{avg}} = P(\text{MEM}) \cdot (T_{i-1} + T_{\text{MEM}}) + \sum (T_i \cdot P(L_i))$$

Wobei T_i die Zugriffszeit für den Level- i -Cache in Nanosekunden ist, T_{MEM} ist die Zugriffszeit auf den Hauptspeicher und T_{i-1} ist der letzte Cache vor dem Hauptspeicher.

Wichtig:
 $T_{i-1} \neq 0$, wenn erst *nach* dem letzten Cache auf den Hauptspeicher zugegriffen wird. Bei einem parallelen Zugriff auf die Caches und den Hauptspeicher gilt: $T_{i-1} = 0$.

9.2 Virtuelle Speicheradresse

TAG	LINE	WORD	BYTE
-----	------	------	------

TAG Das Feld Tag beshetht aus einem eindeutigen n-Bit-Wert. Er kennzeichnet die entsprechenden Zeile im Hauptspeicher, woher die Daten stammen. Die Anzahl der Bist wird meist so gewählt das eine volle Speicheradresse (32/64 Bits) entsteht.

$$\text{BitSTAG} = \text{BitsADDRESS} - \text{BitsLINE} - \text{BitsWORD}$$

LINE Das Line Feld gibt an, in welcher Zeile sich die entsprechenden Daten befinden, falls diese im Cache vorhanden sein sollten.

$$\text{BitsLINE} = \left\lceil \log_2 \left(\frac{\text{Cachegröße}}{\text{Cachezeilengröße}} \right) \right\rceil$$

WORD Mit dem Word Feld wird angegeben welches Wort aus der Cache Zeile angefordert wird. Dies fällt oftmals zusammen mit dem *BYTE* Feld, da der Cache lediglich eine Byte-Adressierung zulässt.

BYTE Das Byte Feld gibt an welches Byte aus einem Speicherwort geldaen werden soll. Dies kann bei einem Cache der lediglich eine Byte-Adressierung zulässt auch mit dem *WORD* Feld zusammengefasst werden.

$$\text{BitsWORD} = \lceil \log_2 (\text{Cachezeilengröße}) \rceil$$

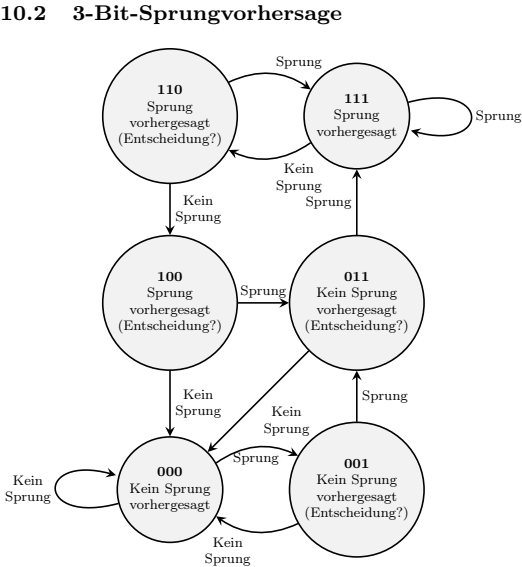
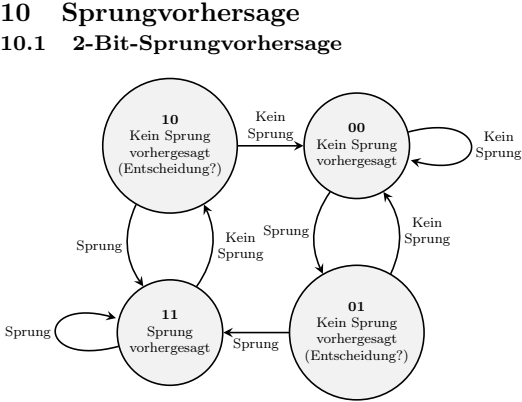
N-Weg-Cache
Ermöglicht es für eine Speicheradresse mehrere Einträge abzulegen. Dies wird benötigt wenn ein Programm häufig Wörter an zwei weit auseinander liegenden Adressen holt und dadurch ständig Konflikte auftreten welche möglicherweise die vorherige Referenz aus dem Cache verdrängt. Ein Zusammenfassung *n* solcher Cache-Einträge nennt man *Menge*.

$$\text{AnzahlMengen} = \frac{\text{Cachegröße}}{\text{Cachezeilengröße} \cdot \text{Assoziativität}}$$

Da die Kapazität des Caches zunimmt jedoch nicht die Anzahl der adressierbaren Cachezeilen, dadurch muss die Anzahl der benötigten Bist für die Cachezeilen wiefolgt lauten:

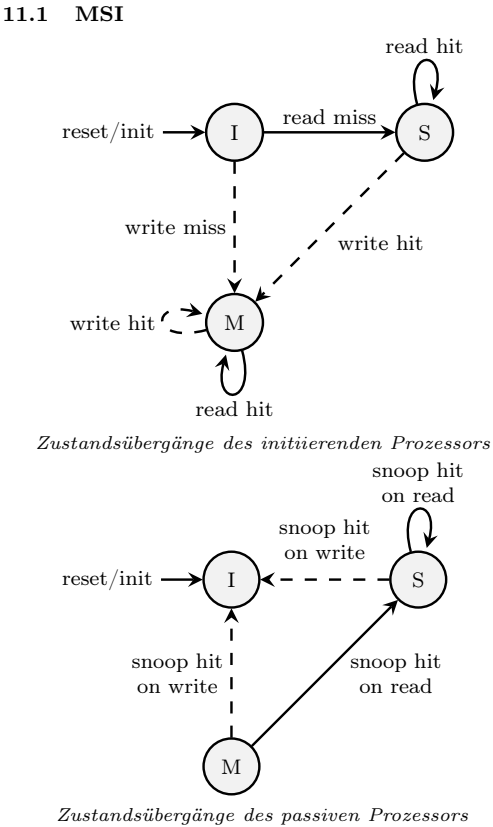
$$\text{BitsLINE} = \lceil \log_2 (\text{AnzahlMengen}) \rceil$$

- 9.3 Schreib-Operationen
- Schreibt der Prozess ein Wort und das Wort befindet sich im Cache, muss der Cache-Eintrag aktualisiert werden, heizu gibt es folgende möglichkeiten:
- Cache-Treffer
 - Write Through** Die modifizierten Daten werden sofort in den Hauptspeicher zurück geschrieben.
 - Write Back** Die modifizierten Daten werden erst in den Hauptspeicher zurück geschrieben, wenn die betroffene Cache-Zeile aus dem Cache entfernt wird. (LRU-Algorithmus *Last Recently Used*)
 - Cache-Verfehlen
 - Write Allocation** Erzeugen eines neuen Cache-Eintrags, in den die Daten geschrieben werden.
 - Write Around** Schreiben von Daten direkt in den Hauptspeicher, ohne einen Cache-Eintrag zu erzeugen.

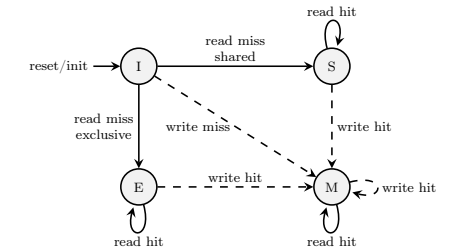


- 11 Cache-Kohärenz
- Modified** Cache-Zeile ist gültig und die Kopie im Hauptspeicher ist nicht gültig. Der Eintrag kann in keinem andern Cache existiren.
- Exclusive** Cache-Zeile ist gültig und die Kopie im Hauptspeicher ist gültig. Der Eintrag kann in keinem andern Cache existiren.
- Shared** Cache-Zeile ist gültig und die Kopie im Hauptspeicher ist gültig. Der Eintrag kann möglicherweise in einem andern Cache existiren.
- Invalid** Cache-Zeile ist nicht gültig oder ist im aktuellen Cache nicht vorhanden.
- Owned** Ist eine Cache-Zeile *Modified* und ein anderer Prozess möchte den aktuellen Wert lesen, so versetzt er die aktuell modifizierte Zeile in den Zustand *Owned*. Die Caches tauschen dabei ihre Werte aus, jedoch beibt der Wert im Hauptspeicher unverändert und somit ungültig.

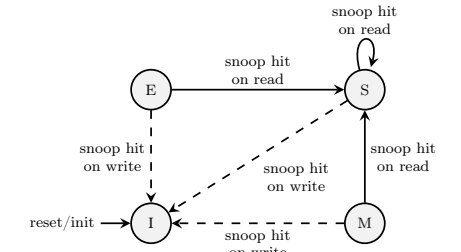
Forward Die Cache-Zeile agiert in diesem Fall als ein Vermittler für andere Caches in welchem die selbe Cache-Zeile mit dem Zustand *Shared* ist. Dabei ist die Cache-Zeile mit dem Zustand *Forward* immer die welche als letztes modifizierte wurde.



11.2 MESI

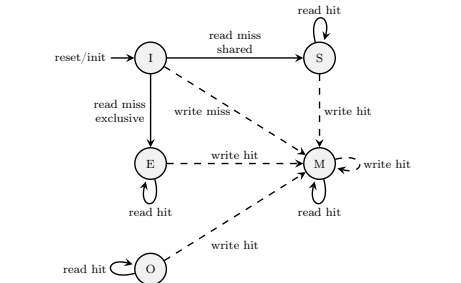


Zustandsübergänge des initiiierenden Prozessors

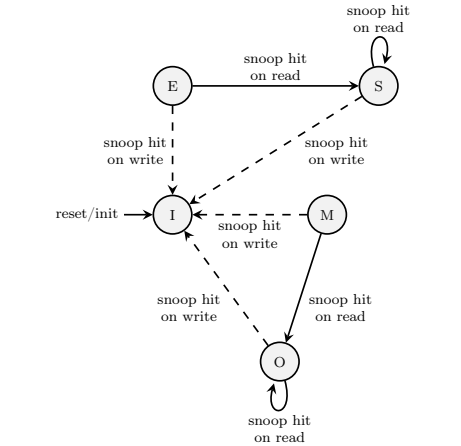


Zustandsübergänge des passiven Prozessors

11.3 MOESI

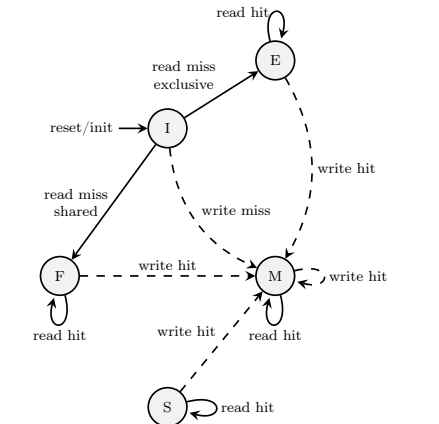


Zustandsübergänge des initiiierenden Prozessors

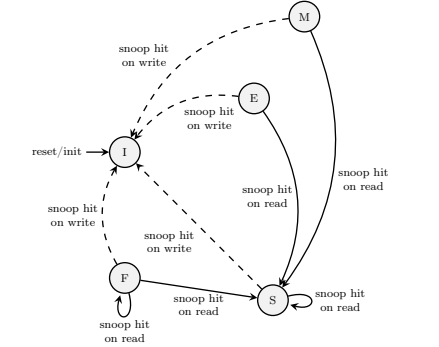


Zustandsübergänge des passiven Prozessors

11.4 MESIF



Zustandsübergänge des initiiierenden Prozessors



Zustandsübergänge des passiven Prozessors

12 Multithreading

Beispiel Verteilung von 3 Threads auf 4 Ausführungseinheiten, dabei benötigt die Ausführung einer Instruktion genau eine Taktzyklus:

Thread X

X	1	2	3	4	5	6
AE 1	X1	X4	×	×	×	X6
AE 2	X2	X5	×	×	×	X7
AE 3	X3		×	×	×	
AE 4			×	×	×	

Thread Y

Y	1	2	3	4	5	6
AE 1	Y1	Y2	Y4	×	×	X7
AE 2		Y3	Y5	×	×	X8
AE 3			Y6	×	×	
AE 4				×	×	

Thread Z

Z	1	2	3	4	5	6
AE 1	Z1	×	Z5	×	Z7	×
AE 2	Z2	×	Z6	×		×
AE 3	Z3	×		×		×
AE 4	Z4	×		×		×

Feinkörniges Multithreading Mit Round-Robin Ausführungsreihenfolge wird in jedem Taktzyklus ein anderer Thread ausgeführt. Falls ein Thread keine Aufgaben hat, wird nicht zurück gegangen.

×	1	2	3	4	5	6	7	8	9	10
AE 1	X1	Y1	Z1	X4	Y2	Z5	Y4	Z7	X6	Y7
AE 2	X2		Z2	X5	Y3	Z6	Y5		X7	Y8
AE 3	X3		Z3				Y6			
AE 4			Z4							

Grobkörniges Multithreading Führt eine Thread aus, bis eine Verzögerung auftritt, dann wird nach Round-Robin die Ausführungsreihenfolge geändert. Im Normalfall tritt eine Verzögerung auf, hier ist es 1 Taktzyklus.

×	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
AE 1	X1	X4	×	Y1	Y2	Y4	×	Z1	×	X6	×	Y7	×	Z5	×	Z7
AE 2	X2	X4	×		Y3	Y5	×	Z2	×	X7	×	Y8	×	Z6	×	
AE 3	X3		×			Y6	×	Z3	×		×	×	×		×	
AE 4			×			×	Z4	×		×	×	×			×	

Simultanes Multithreading Mit Round-Robin Ausführungsreihenfolge werden die Threads aus alle Ausführungseinheiten aufgeteilt. Dabei gibt es eine Verzögerung falls alle Threads zu einem Taktzyklus keine Aufgaben haben.

×	1	2	3	4	5	6	7	8
AE 1	X1	Y2	Z3	Y4	Z5	×	X6	Z7
AE 2	X2	Y3	Z4	Y5	Z6	×	X7	
AE 3	X3	Z1	X4	Y6		×	Y7	
AE 4	Y1	Z2	X5		×	Y8		