1 Einheiten, Größen

1.1 Datenmengen

	Dezimalpräfix ⟶				
↓ Binärpräfix	**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

$$f = \frac{1}{T}$$

$$\frac{1}{Hz} = \text{Taktzykluszeit}$$

1.3 Vorsätze für Maßeinheiten

Symbol	Name	Wert
T	Tera	10^{12}
G	Giga	10^9
M	Mega	10^6
k	Kilo	10^3
h	Hekto	10^2
da	Deka	10^1
–	–	10^0
d	Dezi	10^{-1}
c	Zenti	10^{-2}
m	Milli	10^{-3}
μ	Mikro	10^{-6}
n	Nano	10^{-9}

2 Leistungsbewertung

2.1 Kenngrößen

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

Wichtig:
MIPS-Vergleiche ergeben lediglich Sinn inerhalb der gleichen ISA (Instruction Set Architecture).

$$\text{MIPS} = \frac{10^3}{\text{Instruktionszeit in ns}}$$

2.2 Amdahlsches Gesetz

S = (Gesamt-)Beschleunigung des Programmes
s = Beschleunigung des Programmteils welches von der Verbesserung profitiert
p = Anteil der Zeit, in dem die Verbesserung benutzt wird
T_0 = Zeit vor der Beschleunigung
T_S = Zeit nach der Beschleunigung

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

$$\text{Beschleunigung} = \frac{\text{Zeit vor der Beschleunigung}}{\text{Zeit nach der Beschleunigung}}$$

Damit kann man folgende Gesetzmäßigkeit aufstellen:

$$S = \frac{1}{(1-p) + \frac{p}{s}}$$

$$s = \frac{p \cdot S}{(p \cdot S - S + 1)}$$

$$p = \frac{s \cdot (S-1)}{(s-1) \cdot S}$$

Aus diesem Zusammenhang kann man nun auch folgendes herleiten:

$$\frac{T_0}{T_S} = \frac{1}{(1-p) + \frac{p}{s}}$$

Wenn sich z.B. die Anzahl der Prozessor Kerne ver-n-facht dann ist die Beschleunigung des Programmteils $\frac{1}{n}$ wenn es vorher auf einem Kern lief.

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 $w = \lceil \frac{m}{2} \rceil$ representiert.

$$p_s = \sum_{i=0}^{w} \left(\binom{m}{m-i} \cdot p_c^{m-i} \cdot (1-p_c)^i \right) \cdot p_v$$

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

$$p_s = \left(2p_c - p_c^2\right) \cdot p_v$$

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

$$p_s = \left(3p_c^2 - 2p_c^3\right) \cdot p_v$$

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

$$p_s = \left(3p_c^4 - 8p_c^3 + 6p_c^2\right) \cdot p_v$$

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

$$p_s = \left(6p_c^5 - 15p_c^4 + 10p_c^3\right) \cdot p_v$$

4 Paritätsprüfung

4.1 Paritätsbit

Die Wahrscheinlichkeit das ein Wort der Länge n wirklich fehlerfrei ist, wird bereits durch 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 wiefolt berechnet:

$$\begin{aligned} P_{\text{parity}} &= p^{n+1} + \binom{n+1}{n} \cdot p^n \cdot (1-p) \\ &= p^{n+1} + (n+1) \cdot \left(p^n - p^{n+1}\right) \\ &= (n+1) \cdot p^n - n \cdot p^{n+1} \end{aligned}$$

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

$$P_{\text{no\_parity}} = p^n$$

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 korrigiren. Daraus ergibt sich die allgemeine Ungleichung:

$$2^r \geq m + r + 1$$

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 1100100100_2 und 1001100011_2 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:

$$\begin{aligned} x &= \texttt{00110} & h_{xy} &= 2 \\ y &= \texttt{00101} & h_{xz} &= 1 \\ z &= \texttt{01110} & h_{yz} &= 3 \end{aligned}$$

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:

$$\begin{aligned} h &= d+1 \\ d &= h-1 \end{aligned}$$

5.4 Fehlerkorrektur

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

$$\begin{aligned} h &= 2d+1 \\ d &= \left\lfloor \frac{h-1}{2} \right\rfloor \end{aligned}$$

5.5 Bündelfehlerkorrektur

Um mehrere aufeinanderfolgende Bitfehler zu korrigiren 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 korrigiren, 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
a	1	1	0	0	0	0	1	①	⓪	1	①	1	0	0	①	0	0	1
m	1	1	0	1	1	0	1	①	①	1	⓪	1	0	1	⓪	1	0	0
m	1	1	0	1	1	0	1	①	①	1	⓪	1	0	1	⓪	1	0	0
i	1	1	0	1	0	0	1	⓪	①	1	⓪	1	0	1	①	0	0	1
n	1	1	0	1	1	1	0	⓪	①	1	⓪	1	0	1	⓪	1	1	0
g	1	1	0	0	1	1	1	⓪	①	1	①	1	0	0	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, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59
- Paritätsbit p_2 prüft 2, 3, 6, 7, 10, 11, 14, 15, 18, 19, 22, 23, 26, 27, 30, 31, 34, 35, 38, 39, 42, 43, 46, 47, 50, 51, 54, 55, 58, 59
- Paritätsbit p_4 prüft 4, 5, 6, 7, 12, 13, 14, 15, 20, 21, 22, 23, 28, 29, 30, 31, 36, 37, 38, 39, 44, 45, 46, 47, 52, 53, 54, 55
- Paritätsbit p_8 prüft 8, 9, 10, 11, 12, 13, 14, 15, 24, 25, 26, 27, 28, 29, 30, 31, 40, 41, 42, 43, 44, 45, 46, 47, 56, 57, 58, 59
- Paritätsbit p_{16} prüft 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59
- Paritätsbit p_{32} prüft 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59

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	1	?	0	0	0	0	1	0	1	?	0	1	1	1	0
+ Parität	**0**	**0**	1	**0**	1	1	1	**0**	0	0	0	0	1	0	1	**1**	0	1	1	1	0

5.7 Erweiterungen

SECDED
Single **E**rror **C**orrection **D**ouble **E**rror **D**etection 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, korrigirbar mit Hamming
- Hamming-Prüfbits ✗ Extra-Prüfbit ✓
  ⟶ 2-Bitfehler, nicht lokalisier- bzw. korrigirbar

6 IJVM

6.1 Ausführungsdauer

$$\text{Ausführungszeit} = \frac{\mu\text{-Instruktionen}}{\text{Takt}}$$

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 unterscheidliche Anzahl an μ-Instruktionenbesitzt. Ebenfalls muss beachtet werden ob inerhalb der μ-Instruktionenauch 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 verscheiedenen Ausführungseinheiten eines superskalaren Prozessors werden Abhängigkeitskette gebildet, wobei die Anzahl der Glieder in der längsten Kette der Mindestanzahl der benötigten Taktzyklen entspricht.

$$\mathbf{R1} = \text{R2} + \text{R3} \quad (1)$$
$$\text{R4} = \mathbf{R1} + \text{R5} \quad (2)$$

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

$$\text{R1} = \mathbf{R2} + \text{R3} \quad (1)$$
$$\mathbf{R2} = \text{R5} + \text{R6} \quad (2)$$

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

$$\mathbf{R1} = \text{R2} + \text{R3} \quad (1)$$
$$\mathbf{R1} = \text{R5} + \text{R6} \quad (2)$$

6.3 Adressierungsarten

Unmittelbare Adressierung Bei der unmittelbaren Adressierung folgt ein Operand direkt dem Opcode. Er wird als Konstante betrachtet.
`LOAD 20` → `20` (`Load Immediate`)

Direkte Adressierung Bei der direkten Adressierung wird der Operand der dem Befehl folgt oder das angegebene Register als Adresse angesehen.
`LOAD 20` → `40` (`Load Direct`)

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` → `60` (`Load Indirect`)
`LOAD R2` → `40` (`Load Indirect-Register-Mode`)

Registeradressierung Bei der Registeradressierung ist der Operand ein Register, welches den Wert selbst beinhaltet.
`LOAD R2` → `20` (`Load Register-Mode`)

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)` → `70` (`Load Indexed Mode`)
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)` → `50`
(`Load Based-Indexed-Mode`)

Beispielwerte:

Register	Wert
R1	10
R2	20

Adresse	Wert
20	40
30	50
40	60
50	70

6.4 Umwandlung von MicX in MicY

Bei der Umwandlung darauf achten das der Versatzt der einzelnen Mikroinstruktionen richtig erfolgt, d.h. es muss darauf geachtet werden das ein Register erst dann wieder verwendet werden darf wenn es von der Vorherigen Instruktion nicht mehr benötig wird.
Bei einem **read** wird aus dem `MAR` die Adresse gelesen und der Wert aus dem Speicher in das `MDR` gelegt. (MDR = mem)
Bei einem **write** wird aus dem `MAR` der Adresse gelesen an die der Wert aus dem `MDR` geschrieben wird. (mem = MDR)

6.5 Ausführung von Pipelines

Es besteht eine RAW-Abhängigkeit zwischen der Anweisung I2, die in R4 schreibt, und der Anweisung I3, die R4 liest. Deshalb kann I3 die Eingaberegister erst lesen, wenn I2 das Zurückschreiben abgeschlossen hat, was zu einer Pipelineverzögerung führt (Pipeline Stall).

	1	2	3	4	5	6	7	8	9	10
IF	I1	I2	I3	I4						
ID		I1	I2	I3	I4					
RR			I1	I2	I3	I4				
EX				I1	I2	I3	I4			
WB					I1	I2	I3	I4		

Ohne Berücksichtigung der Abhängigkeiten

	1	2	3	4	5	6	7	8	9	10
IF	I1	I2	I3	I4						
ID		I1	I2	I3	I4	I4	I4			
RR			I1	I2	I3	I3	I3	I4		
EX				I1	I2	...	...	I3	I4	
WB					I1	I2	...	...	I3	I4

Mit Berücksichtigung der Abhängigkeiten

7 Hard Disk Drive

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 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:

$$\text{Spuren} = \frac{\text{Datei}_{\text{Gr}}}{\text{Oberflächen}_{\text{Anz}} \cdot \text{Sektoren}_{\text{Anz}} \cdot \text{Sektor}_{\text{Gr}}}$$

7.3 Speicherdirektzugriff

Der Speicherdirektzugriff ermöglicht es angeschlossenen Peripheriegeräten ohne Umweg über die CPU mit dem Arbeitsspeicher zu kommuniziren. Ein Vorteil des Speicherdirektzugriffs ist die schneller Datenübertargung 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 benötigten 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 Solid State Drive

- 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
- P_F = Prozentanteil der freien Blöcke

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

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

$$\text{Anzahl}_{\text{Datein}} = \frac{\text{Anzahl}_{\text{Dynamischener Blöcke}}}{A_{\text{DBF}}}$$

$$B_T = A_{\text{DBF}} \cdot \text{Anz}_{\text{Datein}} \cdot \text{Zeiteinheit}$$

8.1 Static Wear Levelling

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

8.2 Dynamic Wear Levelling

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

9 Caches

9.1 Zugriffszeit

Die Trefferwahrscheinlichkeit für einen Wert in eimem Level-i-Cache berechnet siech wie folgt:

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

Dabei bezeichnet $p_{\text{L}i}$ 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 berechen:

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

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.

$$\text{Bits}_{\texttt{TAG}} = \text{Bits}_{\texttt{ADDRESS}} - \text{Bits}_{\texttt{LINE}} - \text{Bits}_{\texttt{WORD}}$$

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

$$\text{Bits}_{\texttt{LINE}} = \left\lceil \log_2 \left(\frac{\text{Cachegröße}}{\text{Cachezeilengröße}} \right) \right\rceil$$

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 zusammengefast werden.

$$\text{Bits}_{\texttt{WORD}} = \lceil \log_2 (\text{Cachezeilengröße}) \rceil$$

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

$$\text{Anzahl}_{\text{Mengen}} = \frac{\text{Cachegröße}}{\text{Cachezeilengröße} \cdot \text{Assoziativität}}$$

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:

$$\text{Bits}_{\texttt{LINE}} = \lceil log_2 \left(\text{Anzahl}_{\text{Mengen}} \right) \rceil$$

9.3 Schreib-Operationen

Schreibt der Prozess ein Wort und das Wort befindet sich im Cache, muss der Cache-Eintrag aktualisirt werden, heizu gibt es folgende möglichkeiten:

1. Cache-Treffer
   (a) **Write Through** Die modifizierten Daten werden sofort in den Hauptspeicher zurück geschrieben.
   (b) **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*)
2. Cache-Verfehlen
   (a) **Write Allocation** Erzeugen eines neuen Cache-Eintrags, in den die Daten geschrieben werden.
   (b) **Write Around** Schreiben von Daten direkt in den Hauptspeicher, ohne einen Cache-Eintrag zu erzeugen.

9.4 Cache-Hits und Cache-Misses

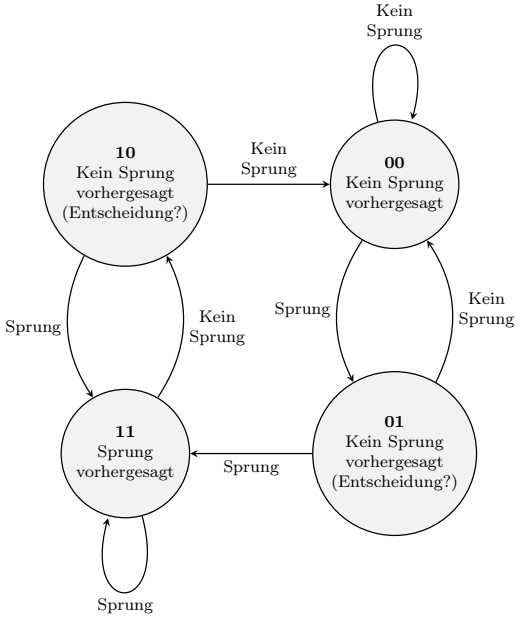
Um die Hits und Misses zu bestimmen muss zunächst die Aufteilung des Speicheradresse berechnet werden. (*Siehe 9.2*) Danach müssen die Tag und Zeilen Bits der Speicheradressen verglichen werden.

1. Zwei Variablen sind in *derselben* Cache-Zeile, wenn ihre Speicheradressen identische Tag- und Line-Einträge besitzen.
2. Zwei Variablen stehen im Konflikt, wenn ihre Speicheradressen identische Line-Einträge, aber verschiedene Tag-Einträge besitzen

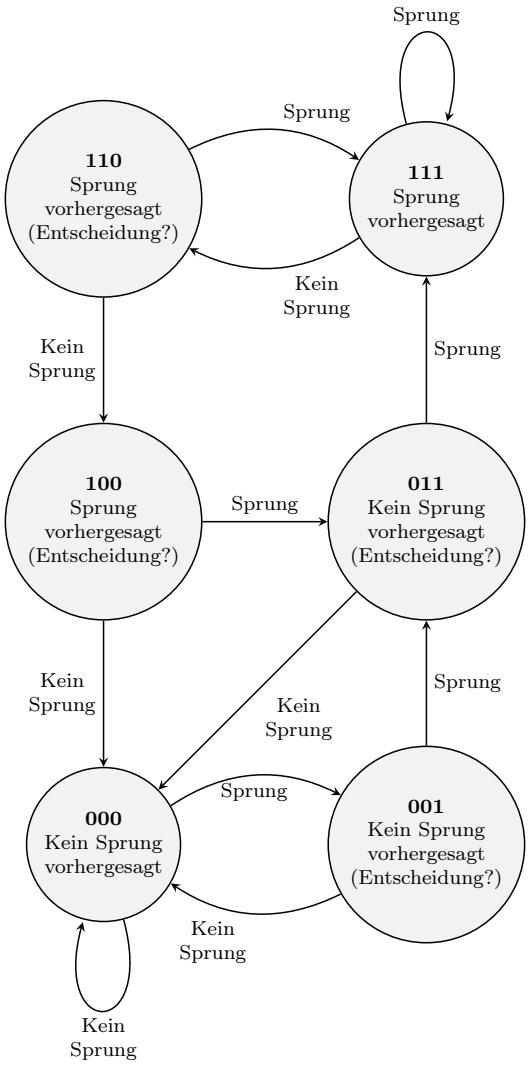
10 Sprungvorhersage

Dynamische Sprungvorhersage mit hilfe einer History-Tabelle mit n Entscheidungs-Bits pro Eintrag kann als ein endlicher Automat dargestellt werden.

10.1 2-Bit-Sprungvorhersage



10.2 3-Bit-Sprungvorhersage



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.

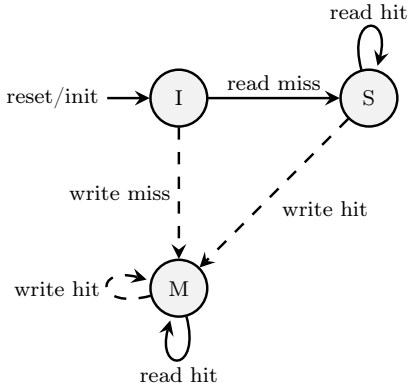
Shared Cache-Zeile ist gültig und die Kopie im Hauptspeicher ist gültig. Der Eintrag kann möglicherweiße in einem andern Cache existiren.

Invalid Cache-Zeile ist nicht gültig oder ist im aktuellen Cache nicht vorhanden.

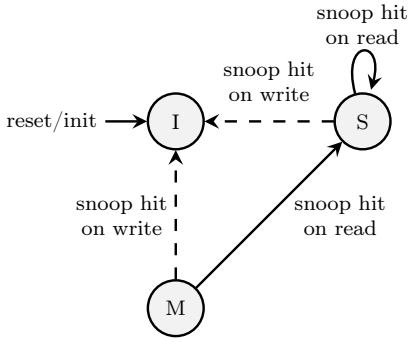
Owned Ist eine Cache-Zeile *Modified* und ein anderer Prozess möchte den aktuellen Wert lesen, so versetzt er die aktuell modifzierte Zeile in den Zustand Owned. Die Caches tauschen dabei ihre Werte aus, jedoch beibt der Wert im Hauptspeicher unverändert und somit ungültig.

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 modifzierte wurde.

11.1 MSI

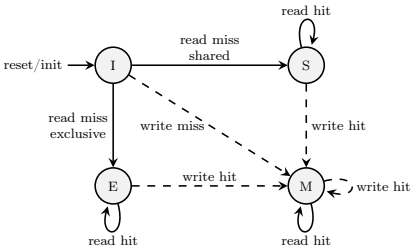


Zustandsübergänge des initiierenden Prozessors

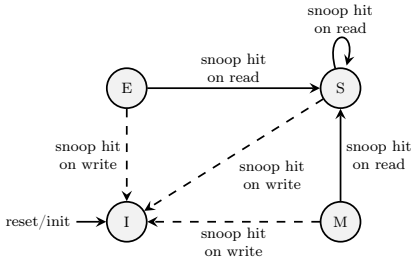


Zustandsübergänge des passiven Prozessors

11.2 MESI

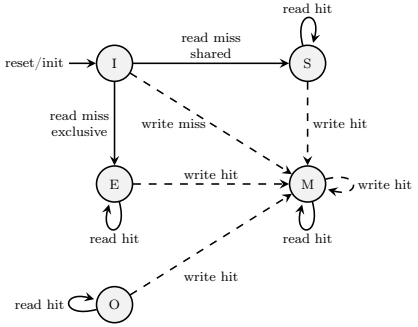


Zustandsübergänge des initiierenden Prozessors

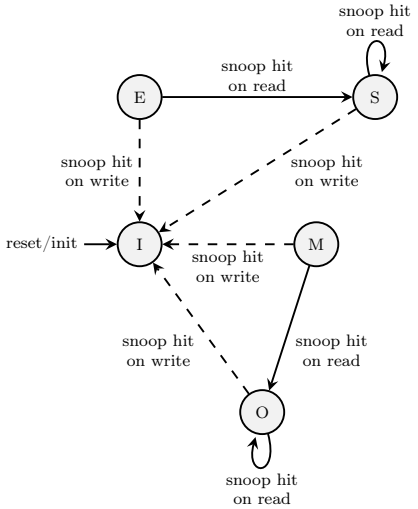


Zustandsübergänge des passiven Prozessors

11.3 MOESI

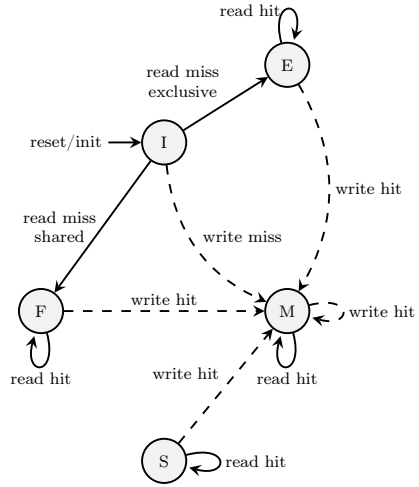


Zustandsübergänge des initiierenden Prozessors

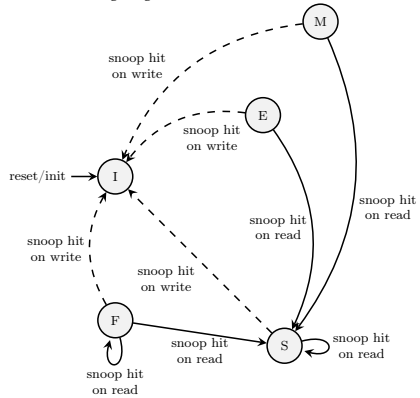


Zustandsübergänge des passiven Prozessors

11.4 MESIF



Zustandsübergänge des initiierenden Prozessors



Zustandsübergänge des passiven Prozessors

12 Multithreading

Beispiel Verteilung von 3 Threads auf 4 Ausfüehrungseinheiten, 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	-		-		-

12.1 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							

12.2 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	-		-		-		-	

12.3 Simultanes Multithreading

Mit Round-Robin Ausführungsreihenfolge werden die Threads aus alle Ausfüehrungseinheiten 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	