technische universität
dortmund

fakultät für informatik
informatik 12

Multiple Issue

Peter Marwedel
Informatik 12

2012/05/15

Multiple Issue

Techniken der vorangegangenen Abschnitte geeignet, um Daten- und Kontrollkonflikte zu lösen

☞ Idealer CPI $\rightarrow$ ~1

Weitere Leistungssteigerung: CPI < 1

☞ mehrere Befehle pro Takt ausgeben (☞ fertigstellen)

Zwei Grundtypen von *multiple-issue* Prozessoren:

Superskalar

- Geben variable Anzahl von Befehlen pro Takt aus
- Mit statischem (vom Compiler erzeugtem) oder dynamischem Scheduling in HW (*Scoreboard*, Tomasulo)

VLIW/EPIC

- Feste Anzahl von Befehlen ausgegeben, definiert durch Befehlscode (Weitgehende Planung der *Issue*-Phase durch Compiler)

5 Primäre Ansätze für *multiple-issue* Prozessoren:

Name	*Issue*-Struktur	Konflikt-detektion	Scheduling	Eigenschaft	Beispiele
Superskalar (statisch)	dynamisch	Hardware	statisch	*in-order* Ausführung	MIPS, ARM Cortex A8
Superskalar (dynamisch)	dynamisch	Hardware	dynamisch	*out-of-order* Ausführung	-
Superskalar (spekulativ)	dynamisch	Hardware	dynamisch+ Spekulation	*out-of-order*+ Spekulation	Core i3, i5, i7 AMD Phenom, IBM Power PC
VLIW	statisch	Compiler	statisch	keine Abhäng. zwischen *Issue*-Paketen	Trimedia (TI 6x)
EPIC	überwiegend statisch	überw. Compiler	überw. statisch	Abhäng. vom Compiler markiert	Itanium

Superscalar mit statischem Scheduling
- Details der Befehlsausgabe (*issue*) -

- In IF 1- n Befehle von IF-Unit geholt
  (ggf. max. von n nicht immer möglich, z.B. bei Sprüngen)
- Befehlsgruppe, die potentiell ausgegeben werden kann
  = *issue packet*
- Konflikte bzgl. Befehlen im *issue packet* werden in Issue-Stufe in Programmreihenfolge (*in-order*) geprüft
  ☞ Befehl ggf. nicht ausgegeben (und alle weiteren)

Aufwand für Prüfung in *Issue*-Stufe groß!
☞ wg. Ausgewogenheit der Pipelinestufen ggf. Issue weiter "*pipelinen*", d.h. in mehrere Stufen unterteilen ☞ nicht-trivial

Parallele Ausgabe von Befehlen limitierender Faktor superskalarer Prozessoren!

Superscalarer MIPS mit statischem Scheduling

Annahme: 2 Befehle pro Takt können ausgegeben werden
(1x ALU, *Load/Store* + 1x FP)

- Einfacher als 2 beliebige Befehle (wegen „Entflechtung”)
- Ähnlich HP7100 Prozessor

Befehlsstart umfasst:

- 2 Befehlsworte holen (64-bit Zugriff, d.h. komplexer als bei nur 1 Befehl ☞ ggf. *Prefetch*?)
- Prüfen, ob 0, 1 oder 2 Befehle ausgegeben werden können
- Befehl(e) ausgeben an korrespondierende funktionelle Einheiten

Prüfen auf Konflikte vereinfacht:

- Integer und FP-Operationen nahezu unabhängig (verschiedene Registersätze)
- Abhängigkeiten nur bei Speichertransfers möglich (von Integer-ALU für FP ausgeführt) ☞ Einschränkung des *issue*

Superscalarer MIPS (statisch)

Beispielsituation der 2-*issue Pipeline*
(Annahme: FP-Operation ist Addition mit 3xEX):

Befehlstyp	Pipelinestufen								
Integer	IF	ID	EX	ME	WB				
FP	IF	ID	EX	EX	EX	ME	WB		
Integer		IF	ID	EX	ME	WB			
FP		IF	ID	EX	EX	EX	ME	WB	
Integer			IF	ID	EX	ME	WB		
FP			IF	ID	EX	EX	EX	ME	WB
Integer				IF	ID	EX	ME	WB	
FP				IF	ID	EX	EX	EX	ME WB

Leistungssteigerung nur bei „geeignetem" Anteil von FP-Operationen im Programm!
(sowie geeigneter „Verflechtung" durch Compiler)

Multiple Issue mit dynamischem Scheduling

Wesentlicher Nachteil von statischem Scheduling für superskalare Prozessoren: Latenzzeiten werden ca. mit Länge des *issue packets* skaliert

- ☞ „längere" Verzögerung (in Anzahl Befehlen) für *Load* bzw. *Branches*

Lösung: Erweiterung des Tomasulo-Algorithmus auf *Multiple Issue* durch

- Sequentielles Ausgeben mehrerer Befehle an *Reservation Stations* innerhalb eines Taktes, oder
- „Verbreiterung" der Ausgabe-Logik (*issue logic*) zur Behandlung mehrerer Operationen parallel (Alle Abhängigkeiten gleichzeitig überprüfen!)

Spekulative Befehlsausführung

- Mehr ILP → Kontrollabhängigkeiten als limitierender Faktor
- Sprungvorhersage reduziert *stalls* aufgrund von Sprungbefehlen, kann aber insbes. für *multiple issue* nicht ausreichend effizient sein!

☞ Nicht nur Vorhersage von Sprüngen, sondern Ausführung des Programms, als ob Vorhersage korrekt

- Unterscheidung zu Sprungvorhersage
  - Normales dynamisches Scheduling: Befehl holen + ausgeben
  - Mit Spekulation: holen, ausgeben + ausführen
- Kann vom Compiler unterstützt, oder ...
- ... komplett in Hardware erfolgen (= erweitertes dynamisches Scheduling)

Spekulative Befehlsausführung (2)

Kombiniert 3 Grundideen:

1. Dynamische Sprungvorhersage zur Bestimmung der (potentiell) auszuführenden Befehle
2. „Spekulation", um Ausführung der Befehle zu erlauben, bevor Kontrollflussabhängigkeiten aufgelöst (mit Möglichkeit, Effekte „spekulativer" Befehle rückgängig zu machen, falls Vorhersagen falsch sind)
3. Dynamisches Scheduling

Hardware-basierte „Spekulation"

- folgt vorhergesagtem Daten- und Kontrollfluss für die Auswahl der auszuführenden Befehle
- Als Erweiterung des Tomasulo-Verfahrens realisierbar

Spekulative Befehlsausführung (3)

Verfahren zur spekulativen Befehlsausführung (*hardware-based speculation*) auf der Basis des Tomasulo-Algorithmus realisiert (wie z.B. in Alpha21264, MIPS R12000, Pentium 4, AMD Athlon)

Dafür erforderlich: Trennung zwischen

- Weiterleitung von Ergebnissen (damit Folgeinstruktionen ausgeführt werden können, insbesondere spekulativ) und
- Fertigstellung eines Befehls

☞ Befehl kann:

- Ergebnis berechnen und Folgebefehlen bereitstellen,
- aber nicht Prozessorzustand so verändern, dass dies nicht rückgängig zu machen ist.

Spekulative Befehlsausführung (4)

Spekulativer Befehl = Befehl, für dessen Ausführung Annahmen über Kontrollfluss gemacht, aber nicht bestätigt sind (erst danach „real”, d.h. nicht spekulativ)

Lesen von weitergeleiteten Ergebnissen durch spekulativen Befehl = spekulatives Lesen von Registern
(korrekt, sobald Quellbefehl nicht mehr spekulativ)

Sobald Befehl nicht mehr spekulativ:
Darf Veränderung / Aktualisierung des Prozessorzustands (Register/Speicher) vornehmen

☞ zusätzlicher Schritt in der Befehlsauswertung!
Bezeichnet als *instruction commit* (deutsch etwa „festlegen”)

Spekulative Befehlsausführung (5)

Grundidee spekulativer Dekodierung:

- Erlauben, dass Befehle *out-of-order* ausgeführt, aber
- Erzwingen, dass *commit in-order* erfolgt
  (d.h. Effekte entstehen in seq. Programmreihenfolge!)
- Sicherstellen, dass keine nicht rückgängig zu machende Aktion vor *commit* stattfindet

Erweiterung der Dekodierung um *Commit*-Phase erfordert:

- Anpassung in übrigen Dekodierungsphasen
- Puffer für berechnete aber nicht "*committete*" Ergebnisse

☞ *Reorder Buffer*: Dient auch zur Weiterleitung [spekulativ] berechneter Zwischenergebnisse

In-order commit erlaubt auch präzise Ausnahmen (*exception* erst bei *commit* behandelt)

Spekulative Befehlsausführung: *Reorder Buffer*

Eigenschaften:

- Stellt wie *Reservation Stations* zusätzliche Register bereit
- Speichert Ergebnisse einer Instruktion in der Zeit zwischen Abschluss der Operation und *commit* der Instruktion
- Ist auch Quelle für Operanden

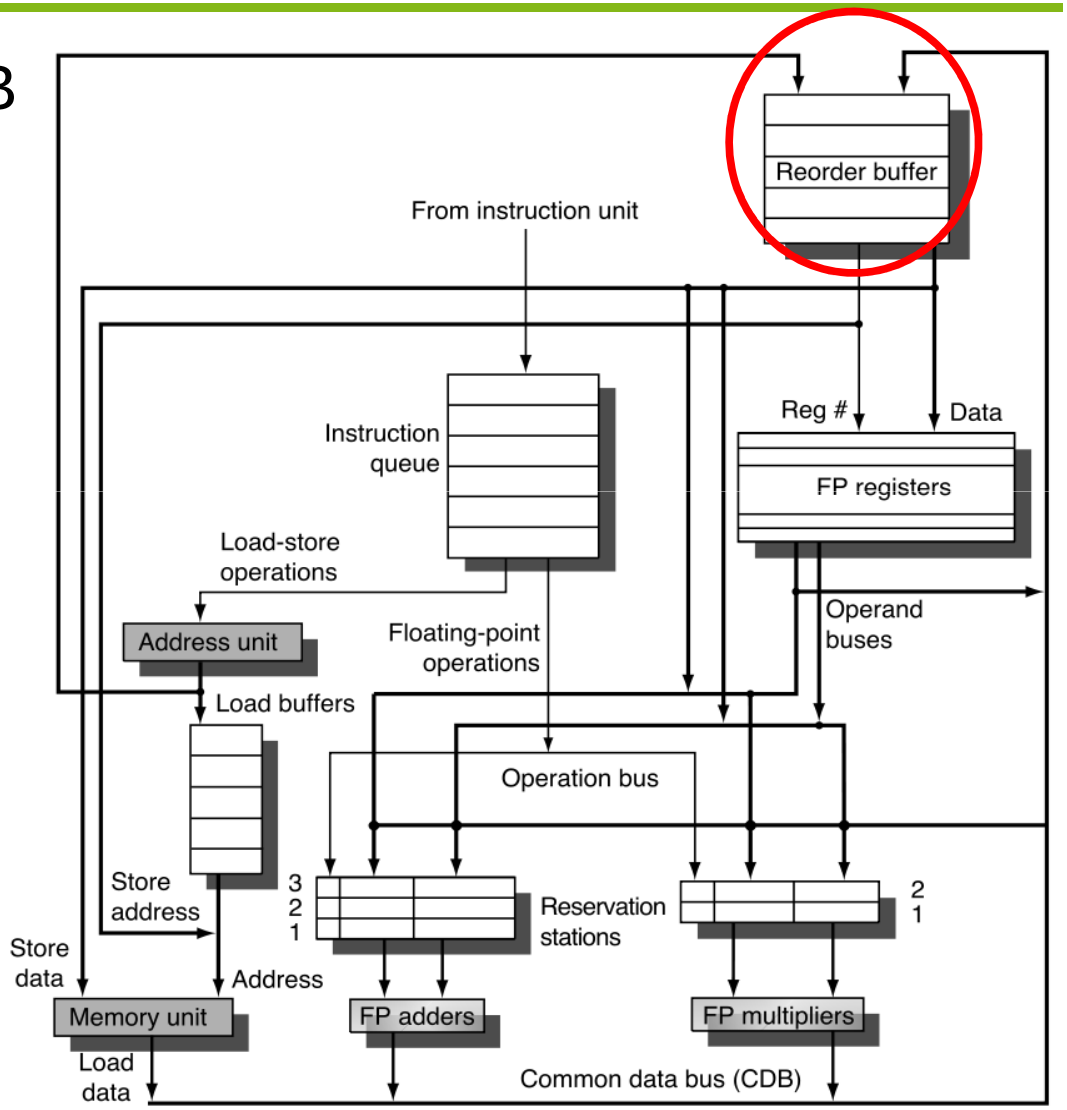
Einträge im Reorder Buffer (ROB):

- Instruktionstyp: Sprung (kein Ziel f. Ergebnis), ALU-Op. oder *Load* (Ziel = Register), *Store* (Ziel = Speicher)
- Ziel: Registerspezifikation oder Speicheradresse
- Wert: Zwischengespeichertes Berechnungsergebnis
- Flag: Auswertung des Befehls fertig, Ergebnis vorhanden?

Spekulative Befehlsausführung: *Reorder Buffer* (3)

Hardwarestruktur mit ROB

- *Register Renaming* jetzt durch ROB realisiert
- *Reservation Stations* puffern Operation + Operanden zwischen *Issue* und Beginn der Ausführung



Spekulative Befehlsausführung: *Reorder Buffer* (4)

Teilschritte der Befehlsausführung

1. *Issue*
   - Befehl aus Befehlswarteschlange holen (beliefert von IF-Unit)
   - Ausgeben, falls ROB-Eintrag und notwendige *Reservation Station* frei
   - Operanden in *Reservation Station* laden, falls in Registern oder ROB vorhanden

Reservation Station erhält Referenz auf ROB-Eintrag als Ziel für Berechnungsergebnis

2. *Execute*
   - *Reservation Stations* beobachten Ergebnisbus (CDB), falls noch nicht alle Operanden verfügbar
     ☞ Auflösung von RAW-Konflikten
   - Sobald beide Operanden verfügbar in *Reservation Station* Befehl in zugehöriger funktioneller Einheit ausführen

Spekulative Befehlsausführung: *Reorder Buffer* (5)

3. *Write Results*
   - Sobald Berechnungsergebnis vorliegt, dieses via CDB (markiert mit Referenz auf ROB-Eintrag) in ROB und alle wartenden *Reservation Stations* schreiben
   - *Reservation Station* wieder freigeben
4. *Commit*
   - Unterscheidung nach: Branch mit falscher Vorhersage, Schreiboperation (Ziel: Speicher) bzw. andere Befehle
   - Branch mit falscher Vorhersage: Spekulation falsch
     - ☞ ROB wird geleert und Dekodierung wird mit korrektem Nachfolgebefehl neu gestartet
   - Sonst: Befehl erreicht Anfang des ROB, Ergebnis verfügbar
     - Berechnungsziel (Register/Speicher) aktualisieren
     - Befehl aus ROB entfernen

Reorder Buffer: Beispiel

Beispiel (bekanntes Codefragment):

```
L.D   F6,32(R2)
L.D   F2,96(R3)
MUL.D F0,F2,F4
SUB.D F8,F6,F2
DIV.D F10,F0,F6
ADD.D F6,F8,F2
```

- Annahme: Latenzen DIV >> MUL >> ADD
  (z.B. ADD.D -> 2, MUL.D -> 10, DIV.D -> 40)
- Betrachtete Situation:
  - MUL.D hat *Write Results* Phase gerade erreicht
  - Wegen kürzerer Latenz liegen Ergebnisse von ADD/SUB auch schon vor, konnten aber nicht *committen.*
- Beachte: Hier kein *Branch*, daher auch keine Spekulation

Reorder Buffer: Beispiel (2)

Reservation Stations

Name	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*	*Dest*
Load1	nein						
Load2	nein						
Add1	nein						
Add2	nein						
Add3	nein						
Mult1	ja	MUL	Mem[96+Regs[R3]]	Regs[F4]			#3
Mult2	ja	DIV		Mem[32+Regs[R2]]	#3		#5

Reorder Buffer

Nr.	*Busy*	*Instruktion*		*Zustand*	*Ziel*	*Wert*
1	nein	`L.D`	`F6,32(R2)`	Commit	F6	Mem[32+Regs[R2]]
2	nein	`L.D`	`F2,96(R3)`	Commit	F2	Mem[96+Regs[R3]]
3	ja	`MUL.D`	`F0,F2,F4`	Write Result	F0	#2 x Regs[F4]
4	ja	`SUB.D`	`F8,F6,F2`	Write Result	F8	#1 - #2
5	ja	`DIV.D`	`F10,F0,F6`	Execute	F10	
6	ja	`ADD.D`	`F6,F8,F2`	Write Result	F6	#4 + #2

Register-Ergebnis-Status

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*...*
ROB	#3			#6	#4	#5	
Busy	ja	nein	nein	ja	ja	ja	

Reorder Buffer: Beispiel (3)

Beispiel mit Schleife/Spekulativer Ausführung

```
Loop: L.D    F0,0(R1)
      MUL.D  F4,F0,F2
      S.D    F4,0(R1)
      DADDI  R1,R1,-8
      BNE    R1,R2,Loop
```

- Multiplikation eines Vektors (0(R1)) mit Konstante (F2)
- Betrachtete Situation:
  - Laden und Multiplikation in aktuellem Durchlauf der Schleife abgeschlossen
  - Weitere Operationen aus aktueller Iteration und der folgenden (Annahme: *branch taken*) sind (teilweise spekulativ) ausgegeben worden und haben Ergebnisse berechnet

Reorder Buffer: Beispiel (4)

Reorder Buffer

Nr.	*Busy*	*Instruktion*		*Zustand*	*Ziel*	*Wert*
1	nein	`L.D`	`F0,0(R1)`	Commit	F0	Mem[0+Regs[R1]]
2	nein	`MUL.D`	`F4,F0,F2`	Commit	F4	#1 x Regs[F2]
3	ja	`S.D`	`F4,0(R1)`	Write Result	0+Regs[R1]	#2
4	ja	`DADDI`	`R1,R1,-8`	Write Result	R1	Regs[R1]-8
5	ja	`BNE`	`R1,R2,Loop`	Write Result	---	
6	ja	`L.D`	`F0,0(R1)`	Write Result	F0	Mem[#4]
7	ja	`MUL.D`	`F4,F0,F2`	Write Result	F4	#6 x Regs[F2]
9	ja	`S.D`	`F4,0(R1)`	Write Result	0+#4	#7
9	ja	`DADDI`	`R1,R1,-8`	Write Result	R1	#4 - 8
10	ja	`BNE`	`R1,R2,Loop`	Write Result	---	

Register-Ergebnis-Status

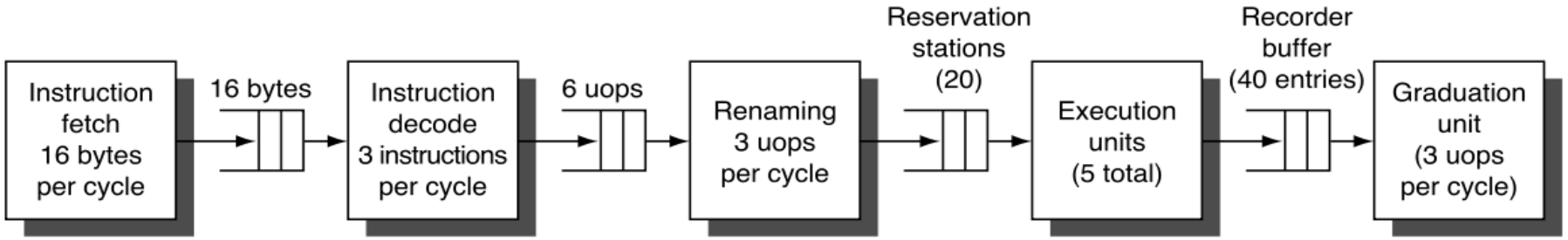
	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*...*
ROB	#6		#7				
Busy	ja	nein	ja	nein	nein	nein	

Falls Sprungvorhersage falsch, ROB löschen, sobald BNE Anfang erreicht bzw. zum *commit* ansteht.

Fallbeispiel: Die P6 Mikroarchitektur

- Basis von Pentium Pro, Pentium II und Pentium III (Unterschiede in Taktrate, Cache-Architektur und tw. Befehlssatz [MMX, SSE])
- Dynamisches Scheduling und *multiple issue*
- Übersetzt IA-32 Instruktionen in Folge von Mikro-Operationen (µops)
  - Werden von Pipeline ausgewertet
  - Ähnlich zu RISC-Befehlen
- Bis zu 3 IA-32 Operationen pro Takt, max. 6 µops
- µops werden von *out-of-order* spekulativer Pipeline mit *register renaming* und *reorder buffer* (ROB) ausgeführt
- max. 3 *issues* bzw. 3 *commits* pro Takt

Fallbeispiel: Die P6 Mikroarchitektur II



8 Stufen für *in-order* Befehlsholen, Dekodieren und Ausgeben

- Sprungvorhersage mit 2-stufigem Prädiktor (512 Einträge)
- *Register renaming* mit 40 virtuellen Registern
- Ausgeben an 20 *Reservation Stations* bzw. 40 ROB-Einträge

3 Stufen für *out-of-order* Ausführung in 5 funktionellen Einheiten

- Integer, FP, Branch, Adressrechnung, Speichertransfer
- Erfordert 1-32 Zyklen (für z.B. Einfache ALU-Op bzw. FP-Division)

3 Stufen für *graduation* = *commit*-Phase (max. 3 µops pro Takt)

technische universität
dortmund

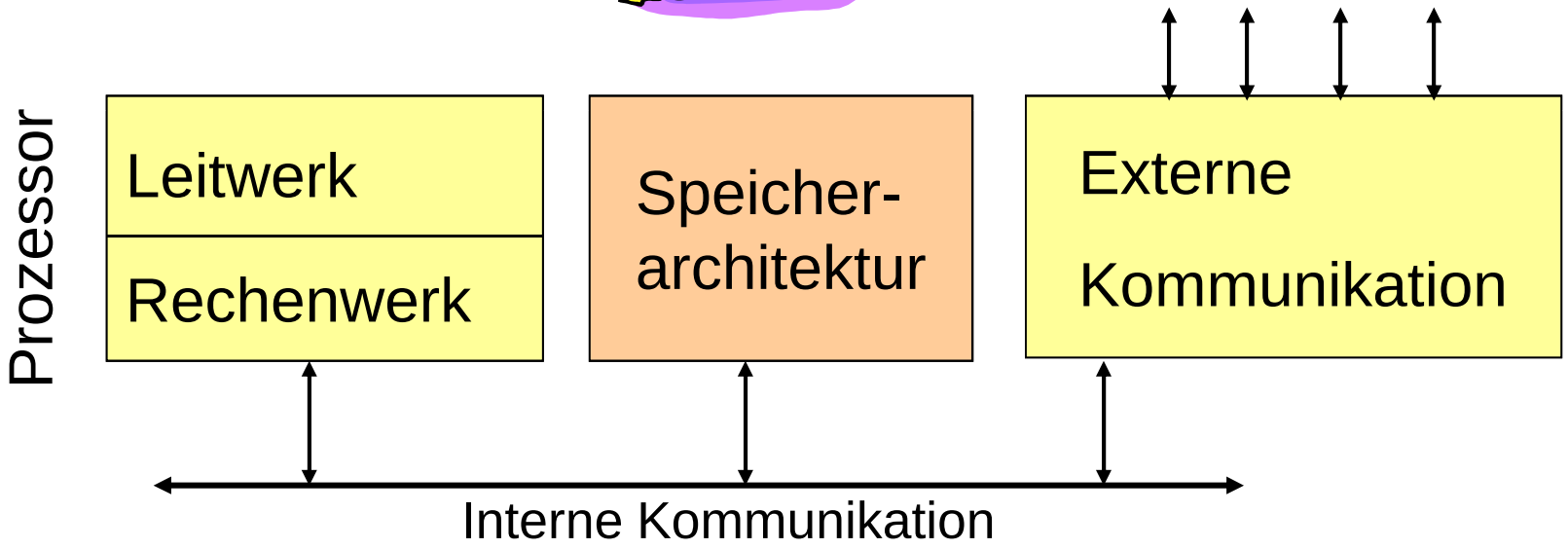
fakultät für informatik
informatik 12

Speicherhierarchie

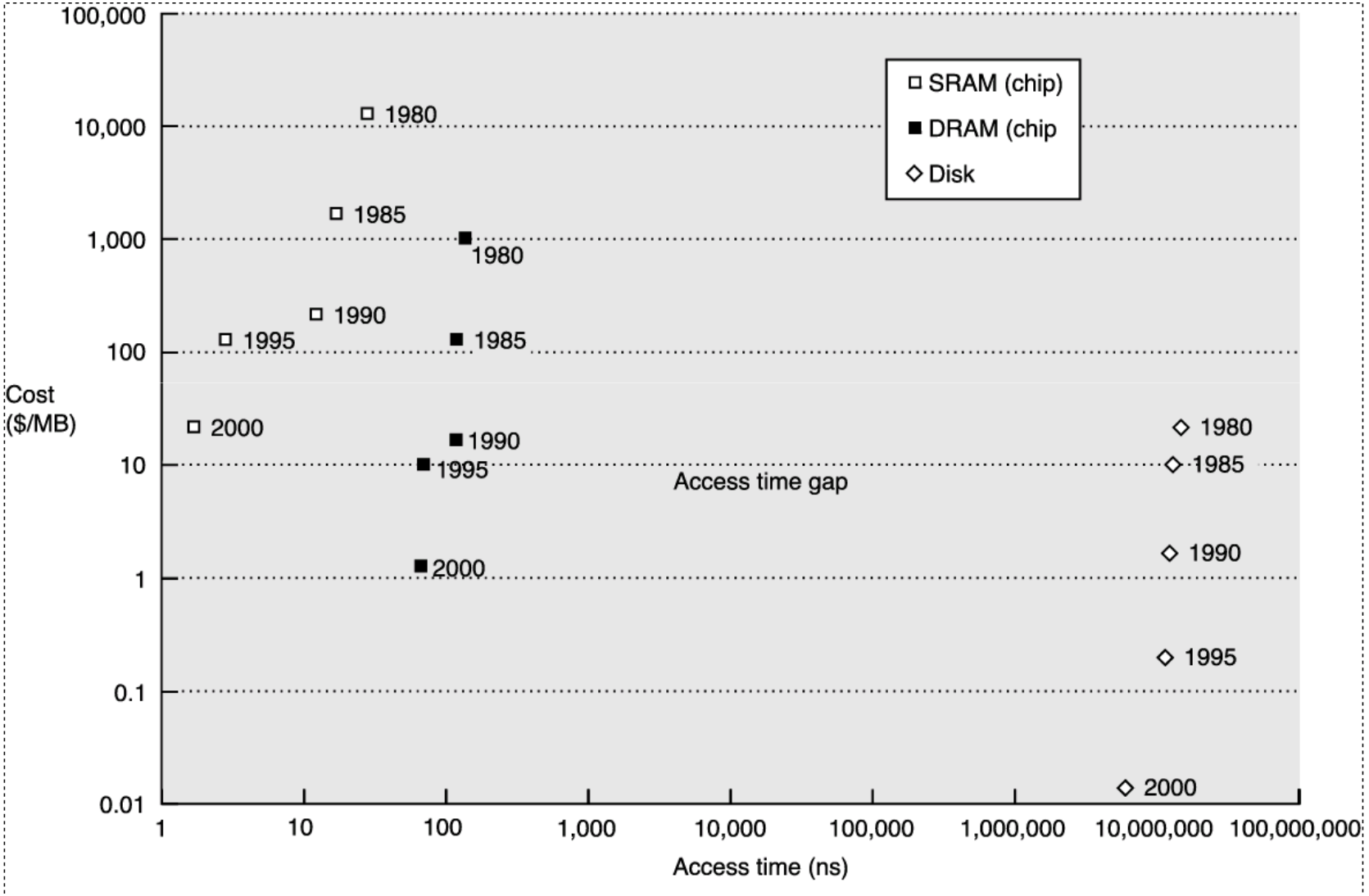
Peter Marwedel
Informatik 12

2011/05/12

Kontext


Prozessor
Leitwerk
Rechenwerk
Speicher-
architektur
Externe
Kommunikation
Interne Kommunikation

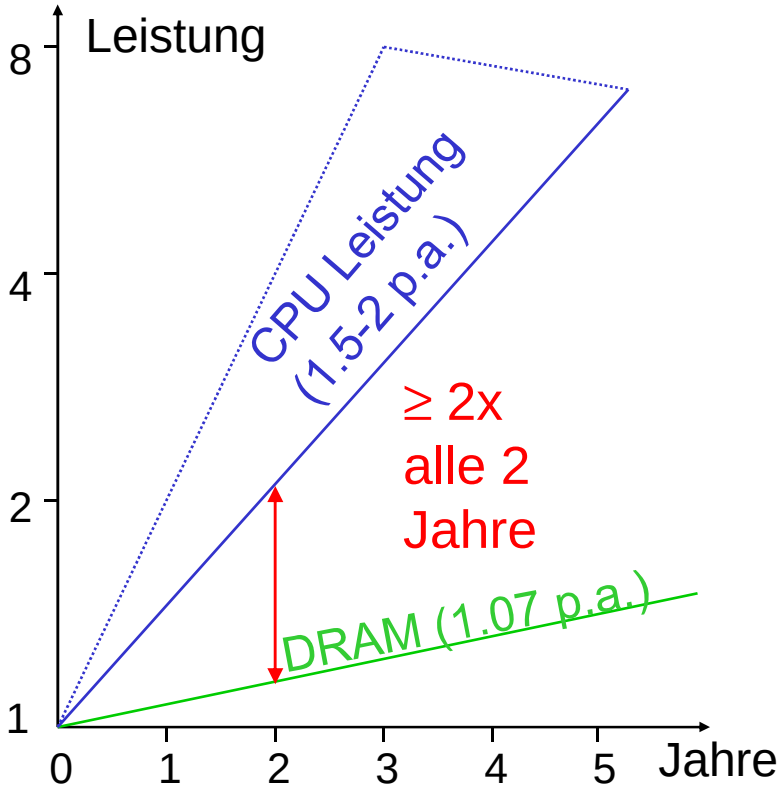
Die Realität: Kosten/Mbyte und Zugriffszeiten für verschiedene Speichermedien



[Hennessy/Patterson, *Computer Architecture*, 3. Aufl.]

Trend der Leistungen von DRAM

Die Performanzlücke zwischen Prozessoren und DRAM wächst



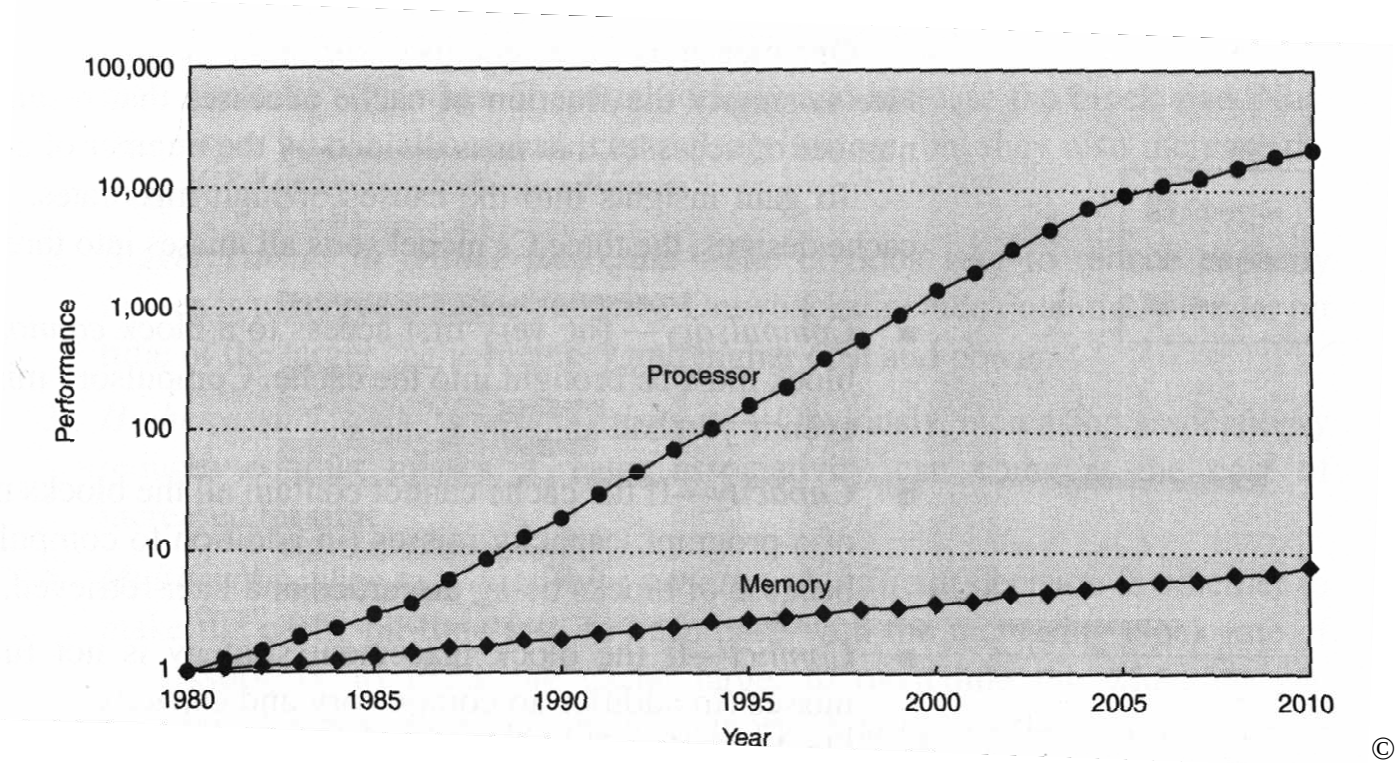
Ähnliche Probleme auch für Eingebettete Systeme

☞ In Zukunft:
Speicherzugriffszeiten ≫ Prozessorzykluszeiten

☞ „*Memory wall*" Problem

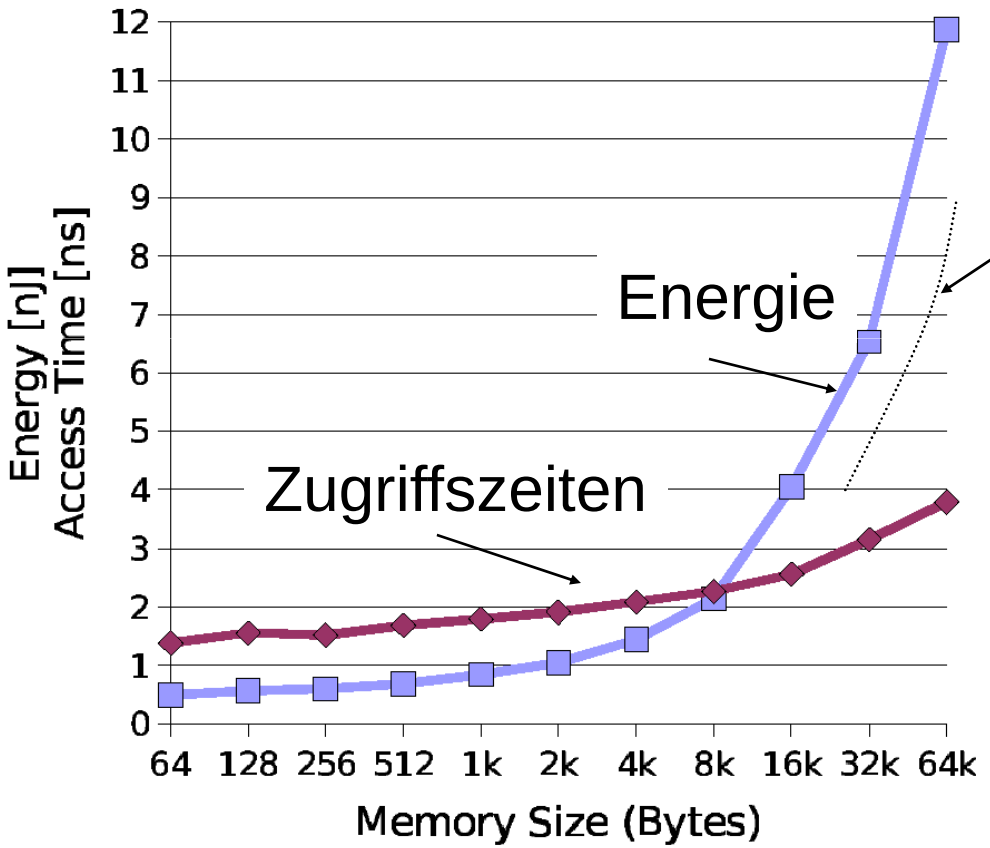
[P. Machanik: Approaches to Addressing the Memory Wall, TR Nov. 2002, U. Brisbane]

Memory wall



- Früher (1980er): Prozessoren ohne Cache entwickelt
- Seit 2000: mindestens 2 Cache-Ebenen
- Problem wird sich weiter verschärfen
- Speicherhierarchien sind Versuch zu kompensieren

Abhängigkeit von der Speichergröße

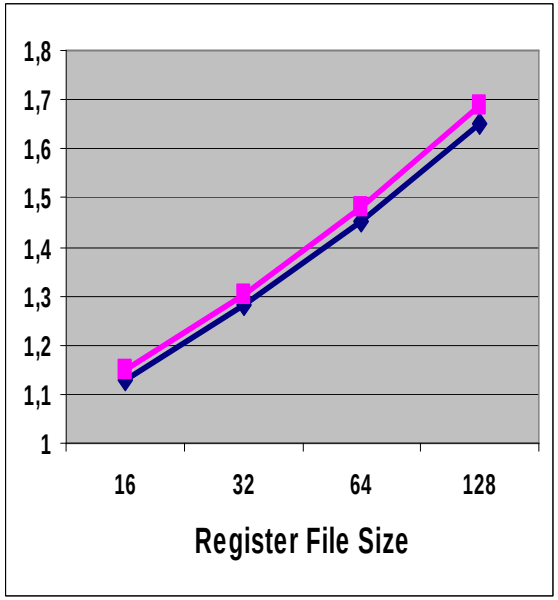


Anwendungen werden größer und größer …

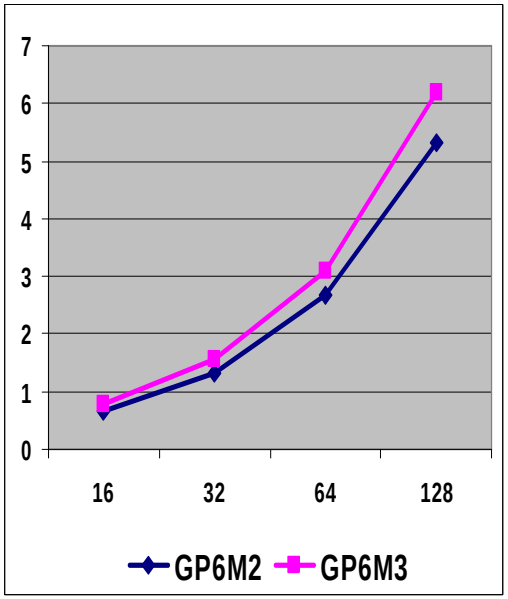
Quelle: CACTI

„Alles“ ist groß für große Speicher

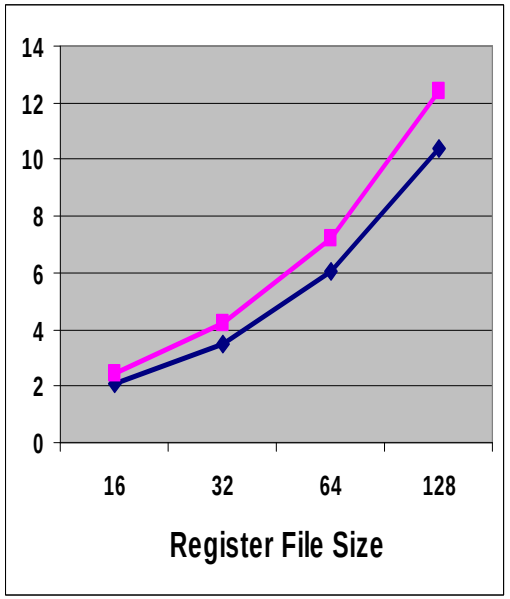
Zykluszeit (ns)*



Fläche (λ^2x10^6)



El. Leistung (W)



* Monolithic register file; Rixner’s et al. model [HPCA’00], Technology of 0.18 mm; VLIW configurations for a certain number of ports („GP*x*M*y*REG*z* where: x={6}, y={2, 3} and z={16, 32, 64, 128“}; Based on slide by and ©: Harry Valero, U. Barcelona, 2001

Speicherhierarchie

- Große Speicher sind langsam
- Anwendungen verhalten sich üblicherweise lokal
- ☞ Häufig benötigte Speicherinhalte in kleinen Speichern, seltener benötigte Inhalte in großen Speichern ablegen!
- ☞ Einführung einer „Speicherhierarchie“
- ☞ *Illusion* eines **großen Speichers** mit (durchschnittlich) **kleinen Zugriffszeiten**
- Enge Grenze für die Zugriffszeit wird selten garantiert.
- ☞ Spezielle Überlegungen bei Realzeitsystemen

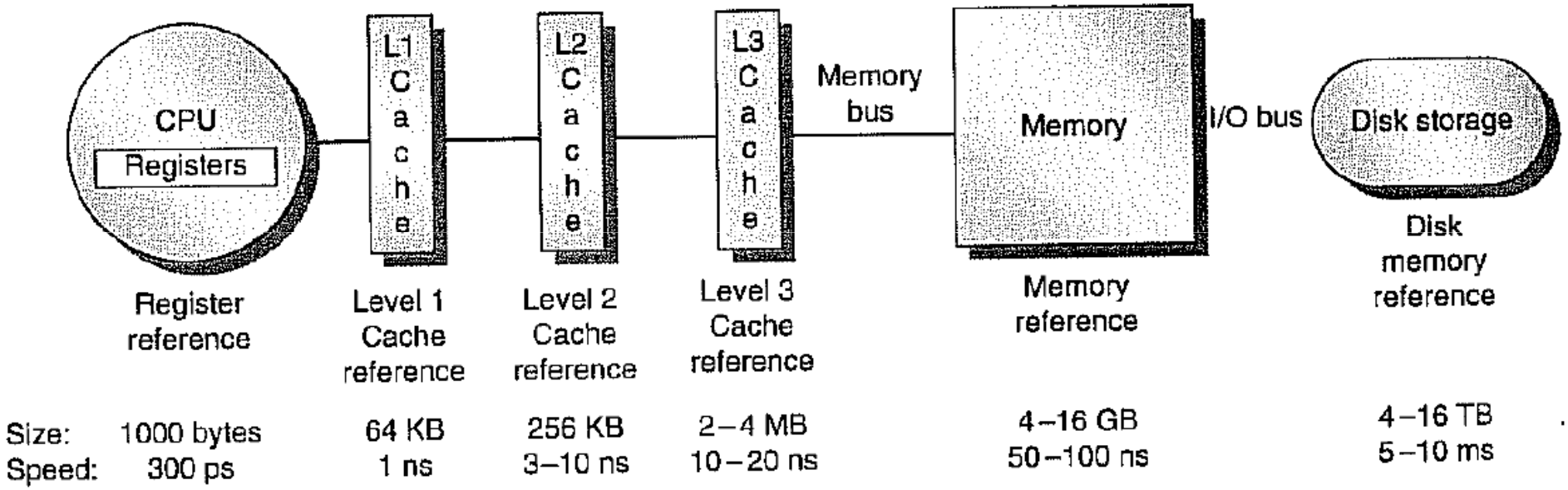
Entwurf von Speicherhierarchien

„Speicherhunger” von Applikationen/Entwicklern/...

“Ideally one would desire an indefinitely large memory capacity such that any particular ... word would be immediately available ... We are ... forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible”

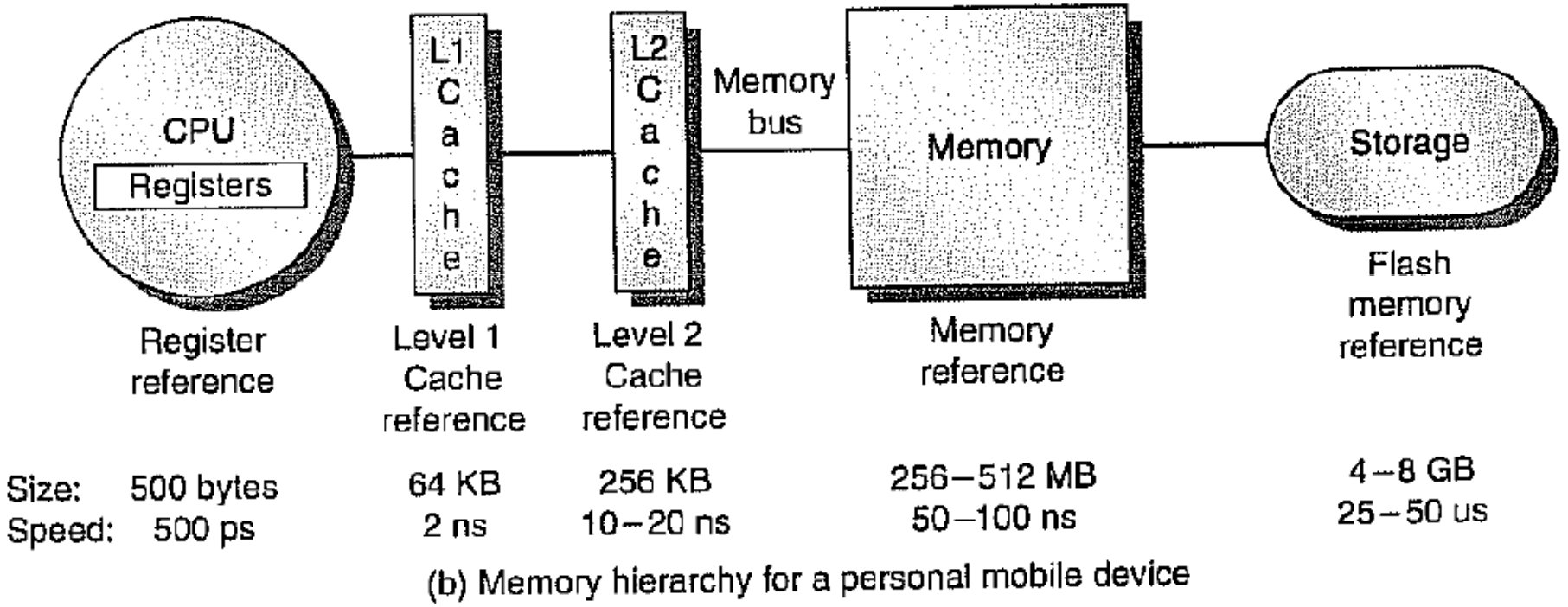
Burks, Goldstine, von Neumann
Preliminary Discussion of the Logical Design of an Electronic Computing Instrument (1946)

Größenordnungen von Kapazitäten und Zugriffszeiten (Server, 2011)



(a) Memory hierarchy for server

Größenordnungen von Kapazitäten und Zugriffszeiten (Mobilgeräte, 2011)

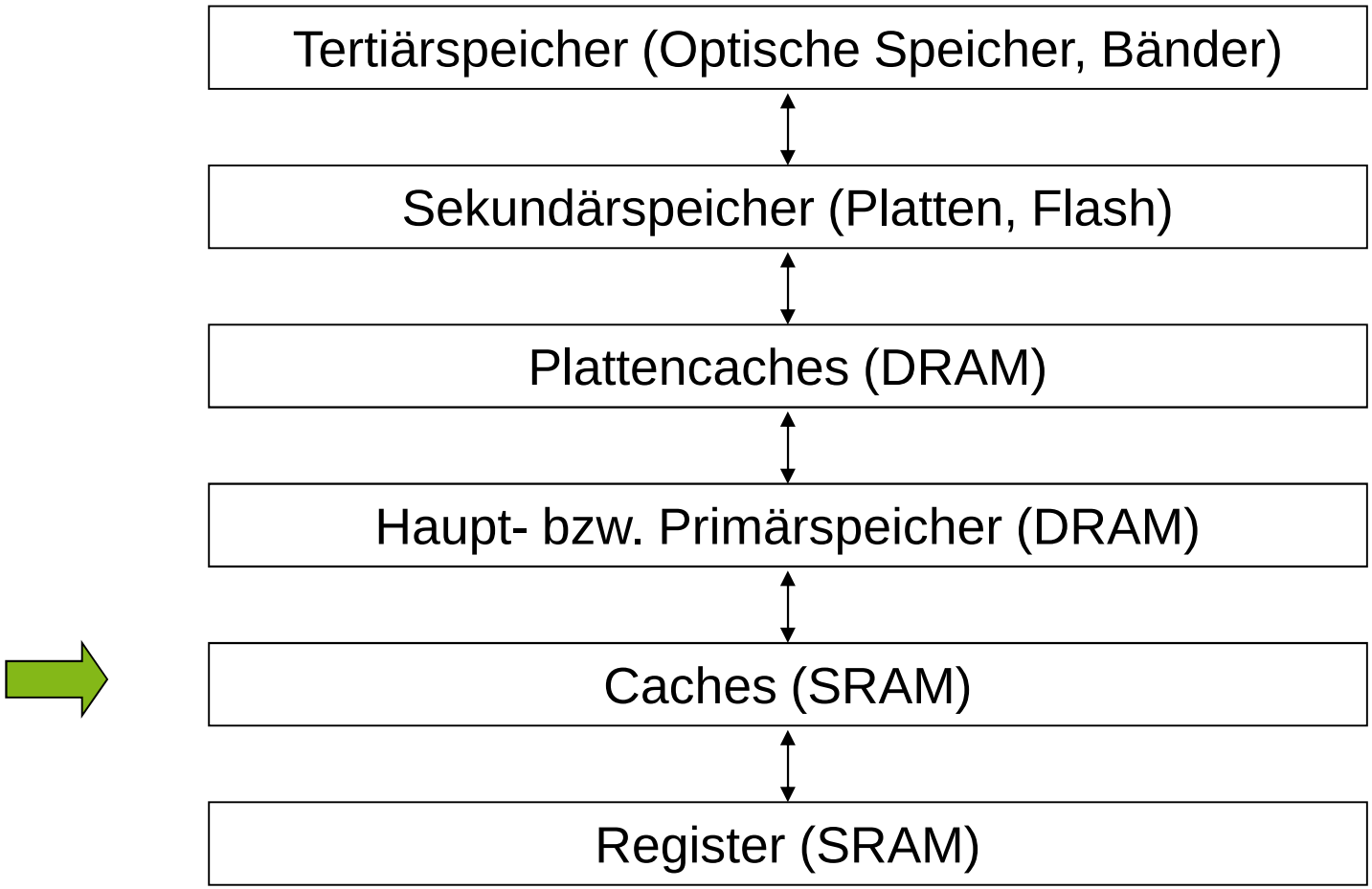


(b) Memory hierarchy for a personal mobile device

Entwurfsziele

- Möglichst großer Speicher
- Möglichst große Geschwindigkeit
  - Geringe Zeit bis zur Verfügbarkeit eines Speicherinhalts (kleine *latency*)
  - Hoher Durchsatz (großer *throughput*)
- Persistente (nicht-flüchtige) Speicherung
- Geringe Energieaufnahme
- Hohe Datensicherheit
- Geringer Preis
- Klein

Mögliche Stufen der Speicherhierarchie
und derzeit eingesetzte Technologien



Caches

- Daten im *Cache* i.d.R. in größeren Einheiten organisiert
  ☞ *Cache*-Block (oder *cache line*)
- Bei erstem Zugriff (temporale Lokalität) in *Cache* geladen
- Örtliche Lokalität: Auch andere Daten im *Cache*-Block mit großer Wahrscheinlichkeit bald gebraucht
- Zugriff bei *cache miss* bestimmt durch
  - Latenzzeit des Speichers
    (Dauer des Zugriffs auf 1. Element im Cache Block) und
  - Bandbreite (Dauer des Transfers der weiteren Daten)
- Bei virtuellem Speicher weitere Hierarchieebene

Cache-Performanz

Bewertungsmaß für die Leistungsfähigkeit einer Speicherhierarchie:

Mittlere Zugriffszeit = *Hit Time* + *Miss Rate* x *Miss penalty*

- *Hit Time* = Zeit für erfolgreichen Zugriff (in CPU-Leistungsgleichung Teil der „normalen" Ausführungszeit)
- *Miss Rate* = Anteil der Fehlzugriffe
- *Miss Penalty* = Wartezyklen der CPU auf Speicher, die pro Fehlzugriff entstehen

Maßeinheiten: Zeit oder Taktzyklen möglich

Cache-Performanz (2)

- Einbeziehung von Speicher-Wartezyklen (*memory stall cycles*) = Takte, in denen CPU auf Speicher wartet
- CPU-Ausführungszeit = (# CPU-Takte + #Speicher-Wartezyklen) x Taktzeit
- Annahmen:
  - CPU-Takte enthalten Zeit für erfolgreichen Cache-Zugriff
  - *Memory stalls* treten nur bei Fehlzugriffen auf

Cache-Performanz (3)

Anzahl der Speicher-Wartezyklen abhängig von:

- Anzahl der Fehlzugriffe (*cache misses*) und
- „Kosten" pro *cache miss* (= *miss penalty*)

Speicher-Wartezyklen =

= # *Cache misses* x *miss penalty*

= IC x (misses / Befehl) x *miss penalty*

= IC x (# Speicherzugriffe / Befehl) x
x Fehlzugriffsrate x *miss penalty*

Beachte:

- Alle Größen messbar (ggf. durch Simulation)
- # Speicherzugriffe / Befehl > 1, da immer 1x Befehlsholen
- *miss penalty* ist eigentlich Mittelwert

Vier Fragen für Caches

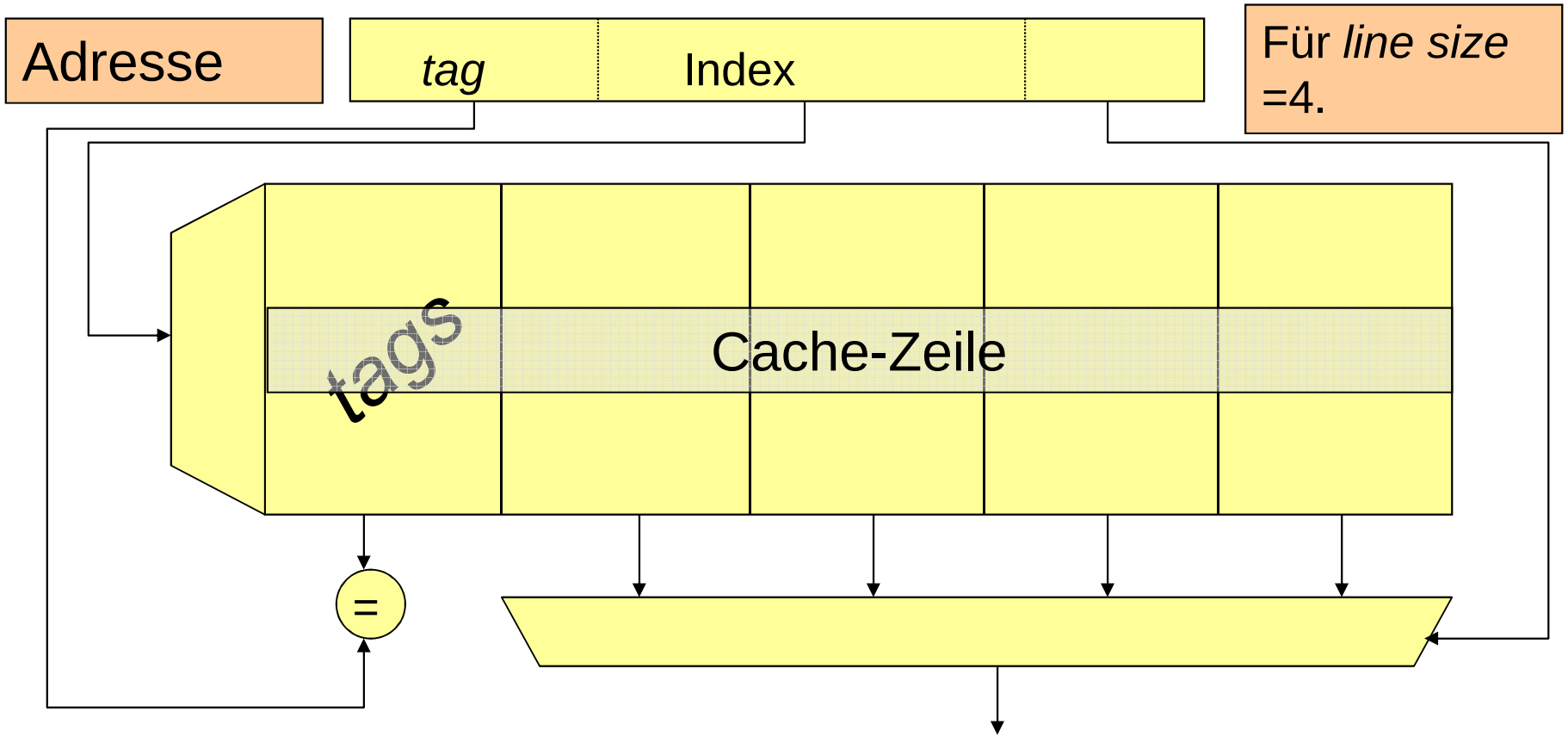
1. Wo kann ein Speicherblock im Cache abgelegt werden (*block placement*)
2. Wie wird ein Speicherblock gefunden (*block identification*)
3. Welcher Block sollte bei einem Fehlzugriff (*cache miss*) ersetzt werden? (*block replacement*)
4. Was passiert beim Schreiben von Daten in den Cache? (*write strategy*)

Block Placement

- ***Direct mapping***
  Für *caching* von Befehlen besonders sinnvoll, weil bei Befehlen *aliasing* sehr unwahrscheinlich ist.
- ***Set associative mapping***
  Sehr häufige Organisationsform, mit Set-Größe=2 oder 4, selten 8.
- ***Associative mapping***
  Wegen der Größe eines Caches kommt diese Organisationsform kaum in Frage.

Block Identification bei *direct mapping*

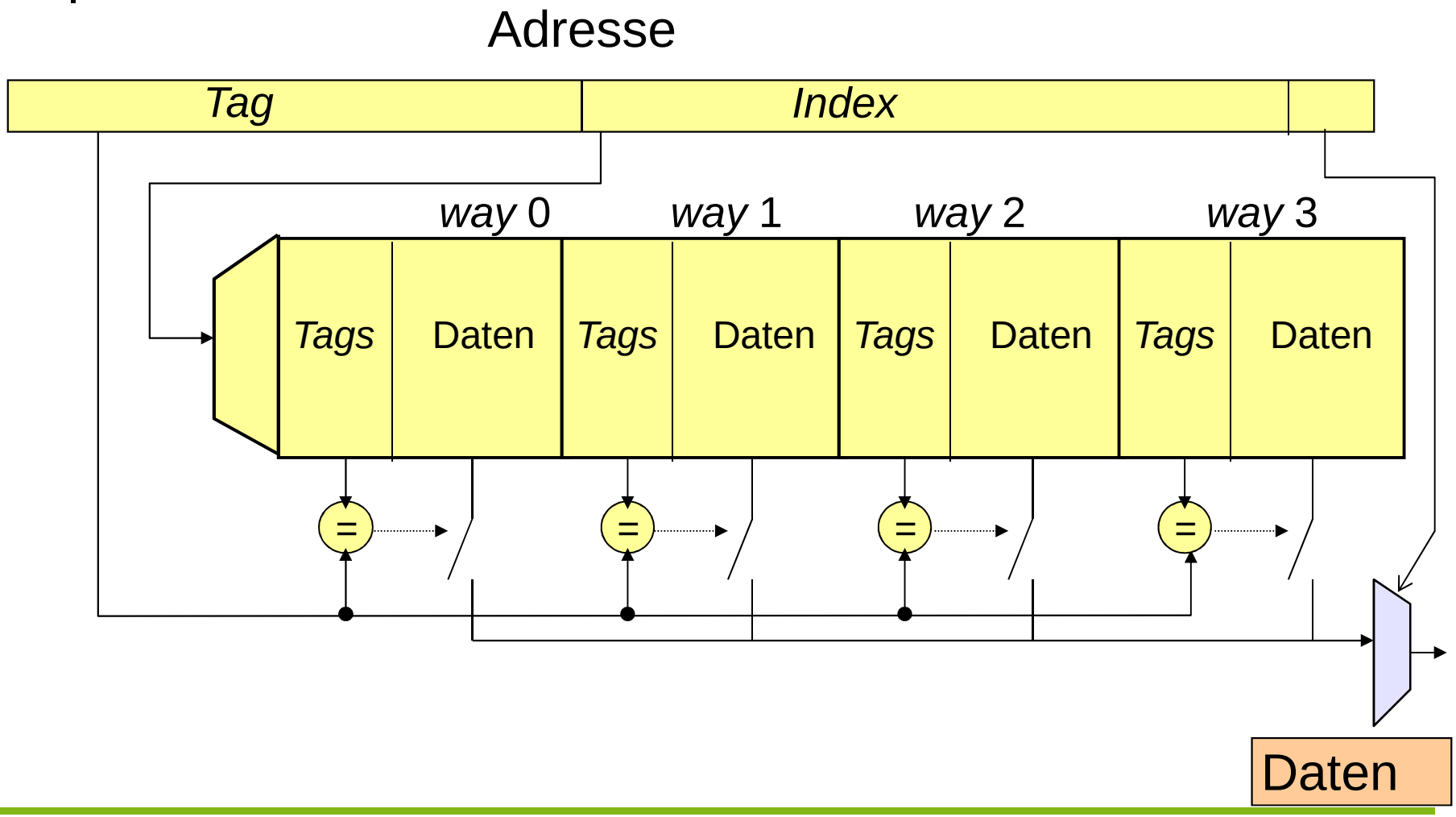
Such-Einheit im Cache: ***Cache*-Zeile** (***cache line***).



Weniger *tag bits*, als wenn man jedem Wort *tag bits* zuordnen würde.

Set-associative cache
n-way cache, *block size = line size*

|Set| = 4



Caches: Ersetzung von Cache-Blöcken

Bei fehlgeschlagenem Cache-Zugriff (*cache miss*) muss Cache-Block ausgewählt und Inhalt mit angeforderten Daten ersetzt werden

Bei *direct mapped*: trivial

- Nur ein Block auf *hit* geprüft
- Nur dieser kann ersetzt werden
- ☞ extrem einfache / schnelle Hardware-Realisierung

Bei voll/*n*-Wege assoziativem Cache:

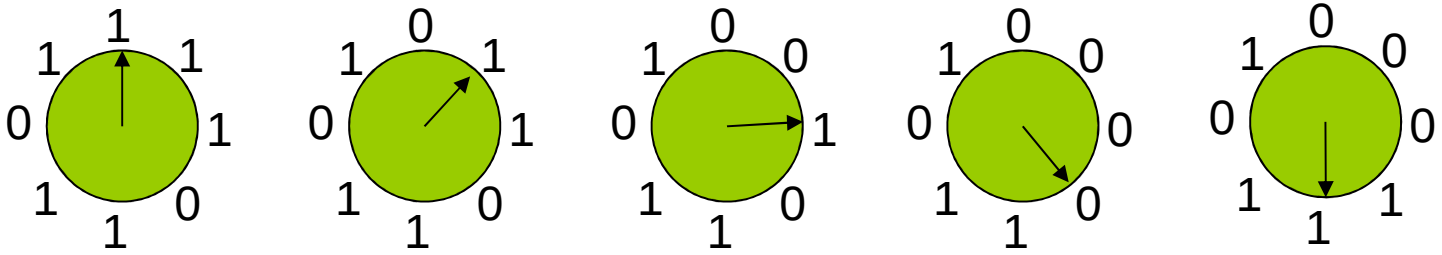
- Viele mögliche Blöcke für Ersetzung zur Auswahl
- Welcher Block soll ersetzt (d.h. entfernt) werden?
- Gefahr, dass diese Daten bald wieder gebraucht werden

Caches: Ersetzungsstrategien für Böcke

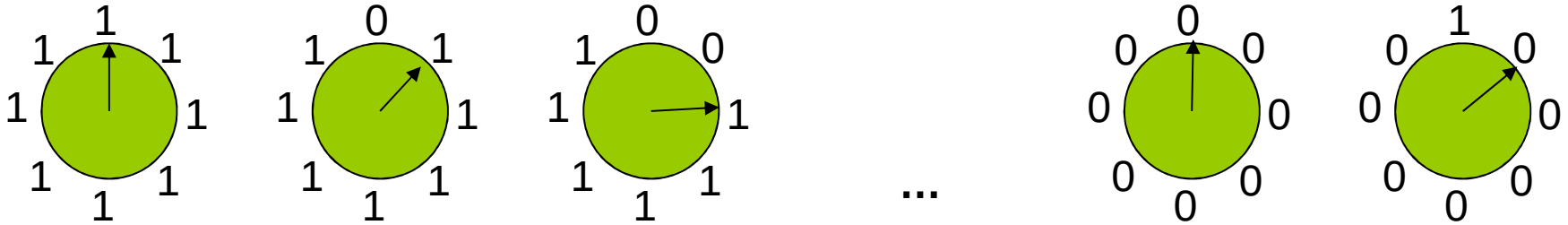
- **Zufällig (*random*)**
  - Ziel: Belegung des Caches möglichst uniform verteilen
  - Ggf. werden Pseudozufallszahlen(folgen) verwendet für reproduzierbares Verhalten (*debugging*!)
  - Für Realzeitsysteme eine Katastrophe ☹
- ***Least-recently used* (LRU)**
  - Der am längsten nicht benutzte Block wird ersetzt
  - Realisierung erfordert einen gewissen Aufwand
  - Für Realzeitsysteme die beste Lösung ☺
- ***First in, first out* (FIFO)**
  - Ältester Block wird ersetzt (auch wenn gerade benutzt)
- ***Clock***
  - Suche nach Block mit *Used-bit*=‘0‘ gleichzeitigem Rücksetzen des Bits. Stop nach max. 1 Runde.

Clock

Annahme: Manche Einträge unbenutzt, Platz benötigt



Annahme: Alle Einträge benutzt, Platz benötigt



Realisierung von LRU mittels verketteter Liste

Datenstruktur: verkettete Liste von Einträgen, sortiert nach der Reihenfolge des Zugriffs.

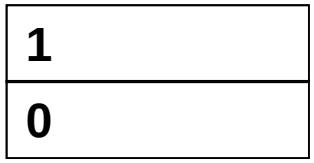
Operationen:

- Zugriff auf Eintrag i:
  - i wird am Anfang der Liste eingefügt und,
  - falls i sonst noch in der Liste vorkommt, dort entfernt.
- Verdrängen eines Eintrags:
  - Der Eintrag, der zum letzten Element der Liste gehört, wird überschrieben.

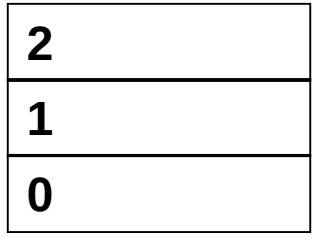
Beispiel: Dynamischer Ablauf

Zugriff auf *tag*=0

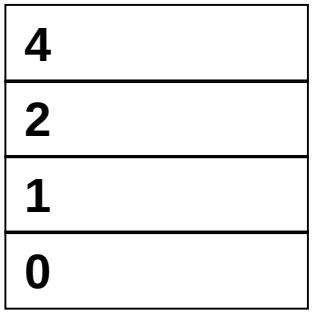
Zugriff auf *tag*=1



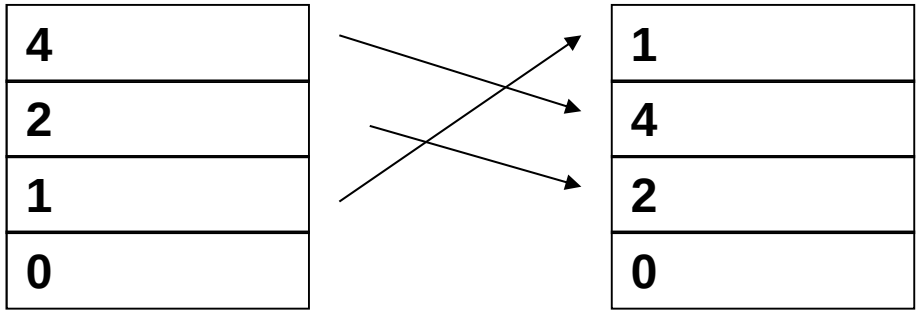
Zugriff auf *tag*=2



Zugriff auf *tag*=4



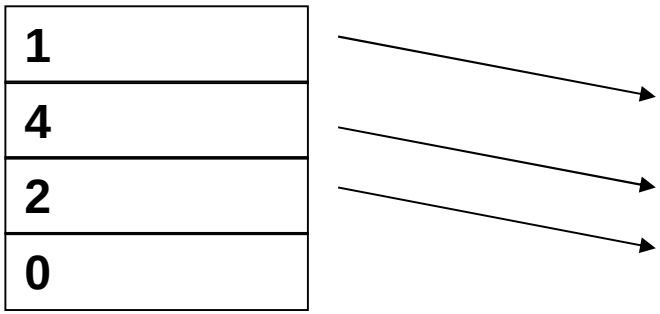
Zugriff auf *tag*=4



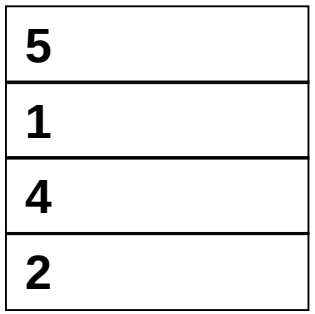
Zugriff auf *tag*=1

Beispiel: Dynamischer Ablauf (2)

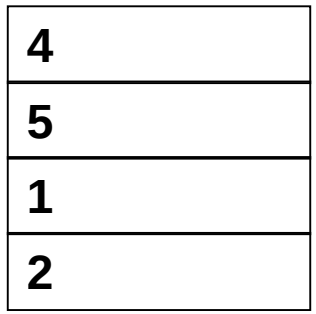
lt. letzter Folie



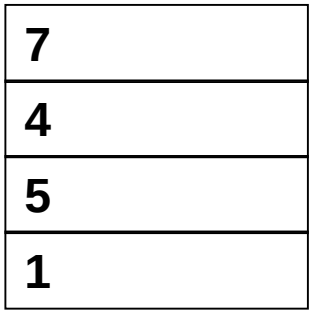
Zugriff auf *tag*=5



Zugriff auf *tag*=4



Zugriff auf *tag*=7



Zusammenfassung

- *Multiple issue*
  - VLIW, EPIC
  - Superskalar
    - *„Statisch“*
    - *„Dynamisch“*
    - *„Spekulativ“*
- Speicherhierarchien
  - Motivation
  - Caches
    - Durchschnittliche Laufzeiten
    - Realisierung von LRU