



Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG

MIN-Fakultät
Fachbereich Informatik



Rechnerstrukturen und Betriebssysteme

64-040 Vorlesung RSB
Kapitel 14: Rechnerarchitektur (3)

Norman Hendrich



Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Technische Aspekte Multimodaler Systeme

Wintersemester 2025/2026

Rechnerarchitektur II

Pipelining

- Befehlspipeline

- MIPS

- Bewertung

Hazards

Parallelität

- Amdahl's Gesetz

- Superskalare Rechner

- Parallelrechner

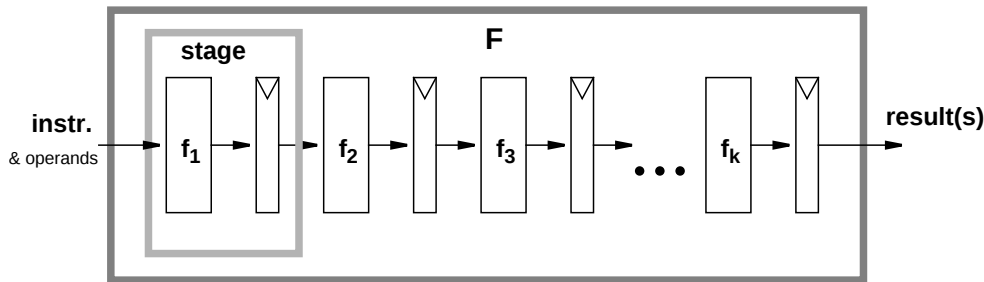
- Symmetric Multiprocessing

Speicherhierarchie

- Speichertypen

- Cache Speicher

Literatur



Grundidee

- ▶ Operation F kann in Teilschritte zerlegt werden
- ▶ jeder Teilschritt f_i braucht ähnlich viel Zeit
- ▶ Teilschritte $f_1 \dots f_k$ können parallel zueinander ausgeführt werden
- ▶ Trennung der Pipelinestufen („stage“) durch Register
- ▶ Zeitbedarf für Teilschritt $f_i \gg$ Zugriffszeit auf Register (t_{FF})

Pipelining-Konzept

- ▶ Prozess in unabhängige Abschnitte aufteilen
- ▶ Objekt sequenziell durch diese Abschnitte laufen lassen
 - ▶ zu jedem Zeitpunkt werden zahlreiche Objekte bearbeitet
 - ▶ —"— sind alle Stationen ausgelastet

Konsequenz

- ▶ Pipelining lässt Vorgänge gleichzeitig ablaufen
- ▶ reale Beispiele: Autowaschanlagen, Fließbänder in Fabriken

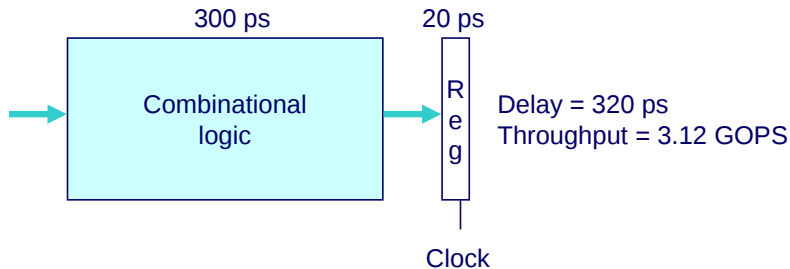
Arithmetische Pipelines

- ▶ Idee: lange Berechnung in Teilschritte zerlegen
- ▶ wichtig bei komplizierteren arithmetischen Operationen
 - ▶ die sonst sehr lange dauern (weil ein großes Schaltnetz)
 - ▶ die als Schaltnetz extrem viel Hardwareaufwand erfordern
 - ▶ Beispiele: Multiplikation, Division, Fließkommaoperationen ...
- + Erhöhung des Durchsatzes, wenn die Berechnung mehrfach hintereinander ausgeführt wird / die Pipeline ausgelastet ist

Befehlspipeline im Prozessor

- ▶ Idee: die Phasen der von-Neumann Befehlsabarbeitung (Befehl holen, Befehl decodieren ...) als Pipeline implementieren
- folgt in *Befehlspipeline*, ab Folie 14

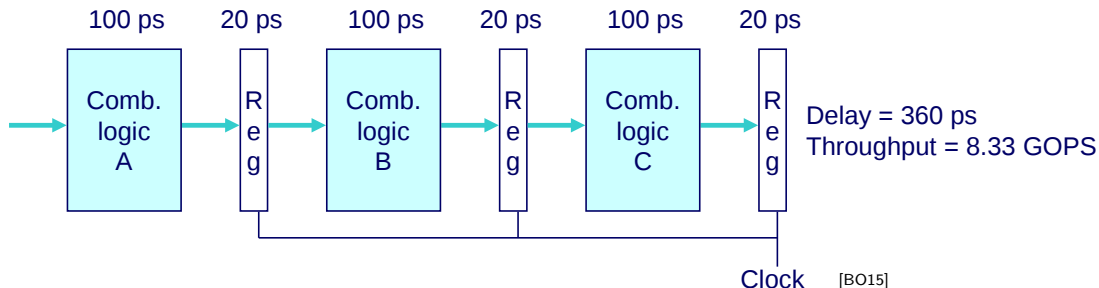
Beispiel: Schaltnetz ohne Pipeline



[B015]

- ▶ Verarbeitung erfordert 300 ps
- ▶ weitere 20 ps um das Resultat im Register zu speichern
- ▶ Zykluszeit: mindestens 320 ps

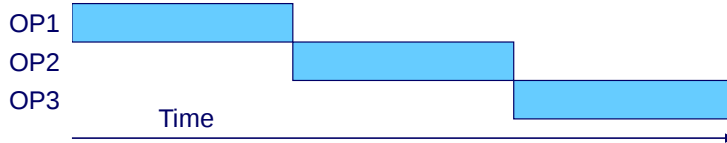
Beispiel: Version mit 3-stufiger Pipeline



- ▶ Kombinatorische Logik in 3 Blöcke zu je 100 ps aufgeteilt
- ▶ neue Operation, sobald vorheriger Abschnitt durchlaufen wurde
⇒ alle 120 ps neue Operation
- ▶ allgemeine Latenzzunahme
⇒ 360 ps von Start bis Ende

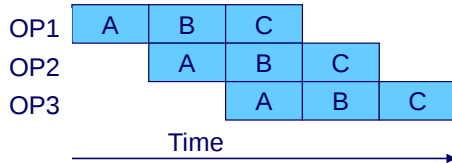
Prinzip: 3-stufige Pipeline

► ohne Pipeline

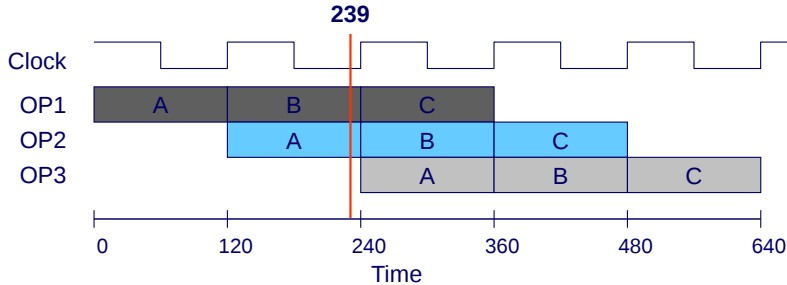


[BO15]

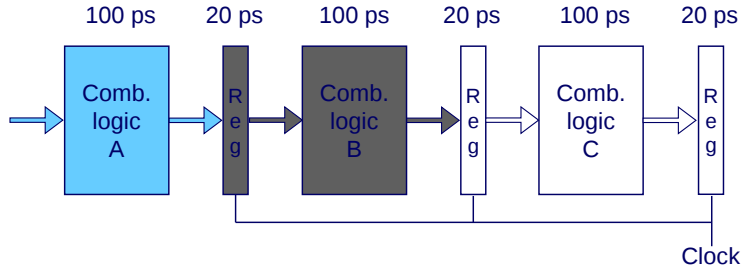
► 3-stufige Pipeline



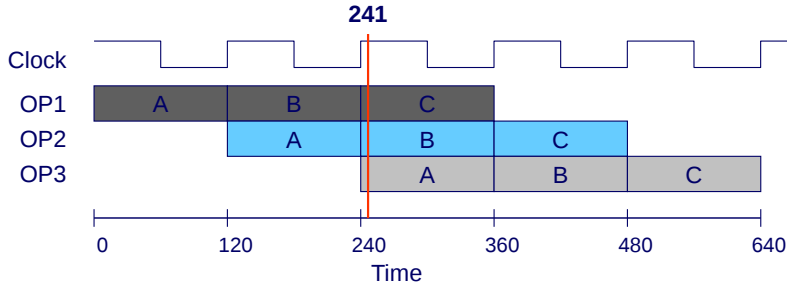
Timing: 3-stufige Pipeline



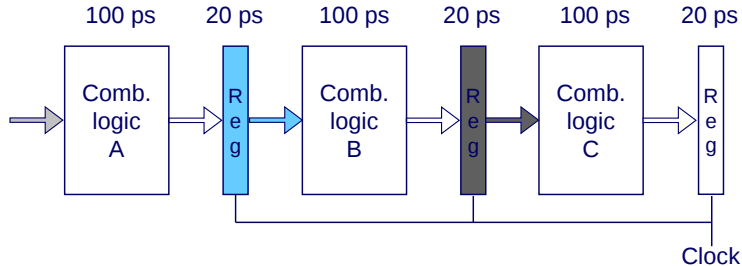
[BO15]



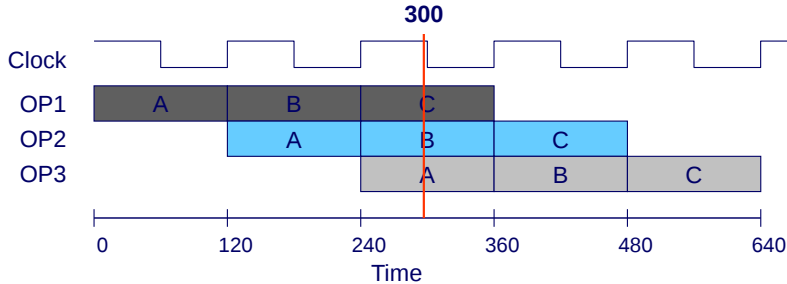
Timing: 3-stufige Pipeline



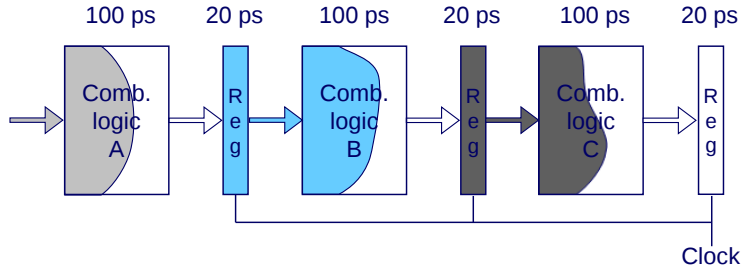
[BO15]



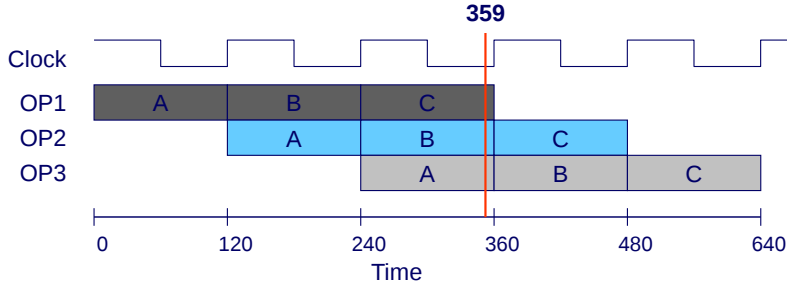
Timing: 3-stufige Pipeline



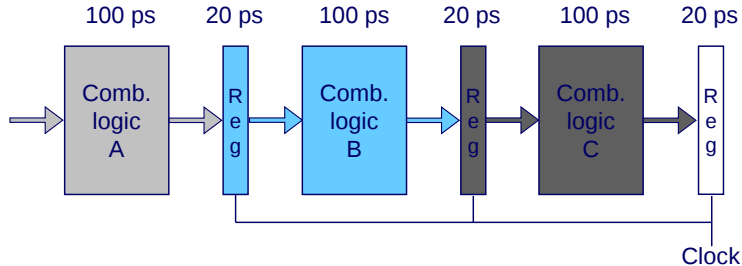
[BO15]



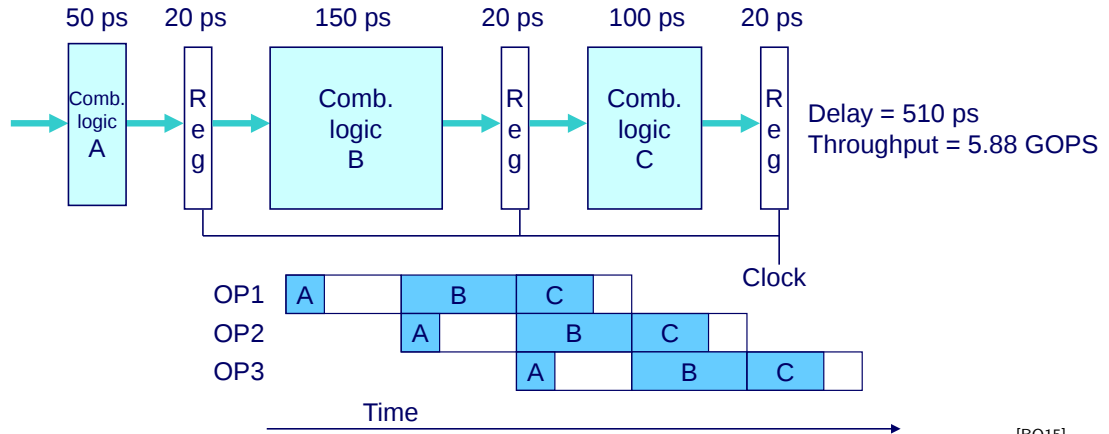
Timing: 3-stufige Pipeline



[BO15]



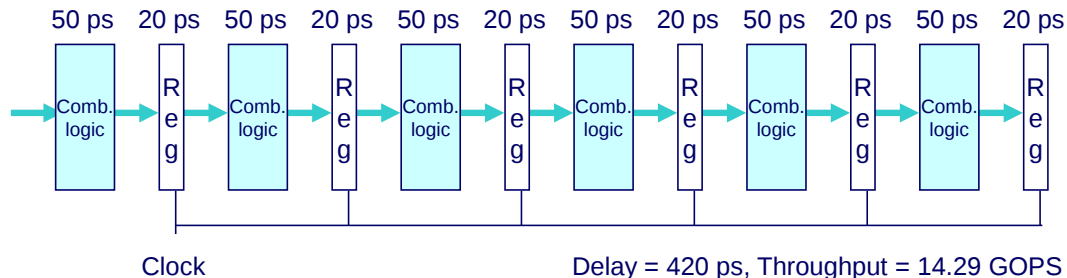
Limitierungen: nicht uniforme Verzögerungen



[BO15]

- ▶ Taktfrequenz limitiert durch langsamste Stufe
- ▶ Schaltung in möglichst gleich schnelle Stufen aufteilen

Limitierungen: Register „Overhead“

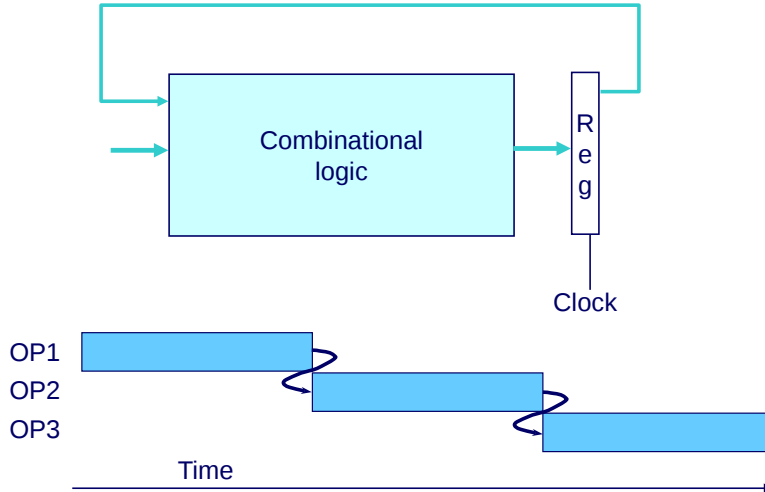


[BO15]

- ▶ registerbedingter Overhead wächst mit Pipelinelänge
- ▶ (anteilige) Taktzeit für das Laden der Register

	Overhead	Taktperiode
1-Register:	6,25% 20 ps	320 ps
3-Register:	16,67% 20 ps	120 ps
6-Register:	28,57% 20 ps	70 ps

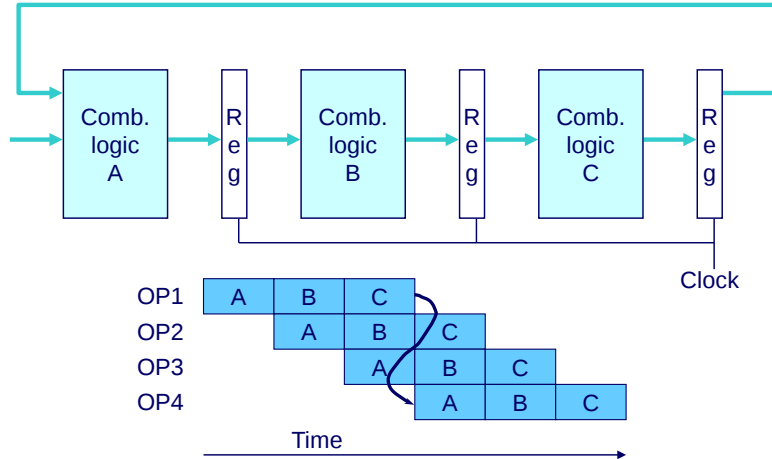
Limitierungen: Datenabhängigkeiten



[BO15]

- ▶ jede Operation hängt vom Ergebnis der Vorhergehenden ab

Limitierungen: Datenabhängigkeiten (cont.)



[BO15]

- ⇒ Resultat-Feedback kommt zu spät für die nächste Operation
- ⇒ Pipelining ändert Verhalten des gesamten Systems

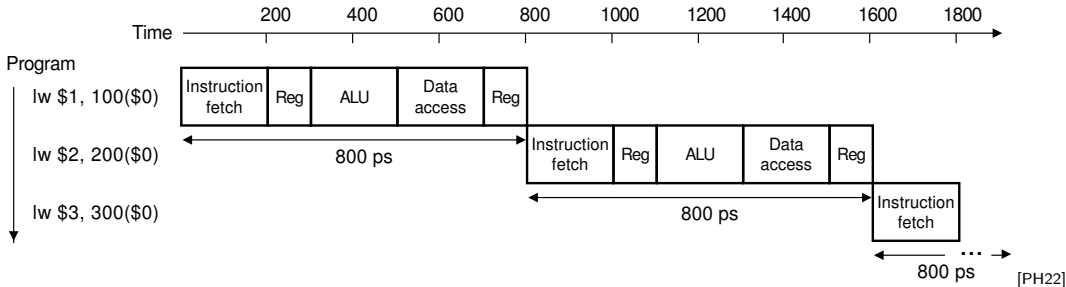
typische Schritte der Befehlsabarbeitung (von ISA abhängig)

▶ IF	I nstruction F etch Instruktion holen, in Befehlsregister laden
ID	I nstruction D ecode Instruktion decodieren
OF	O perand F etch Operanden aus Registern holen
EX	E xecute ALU führt Befehl aus
MEM	M emory access Speicherzugriff: Daten laden/abspeichern
WB	W rite B ack Ergebnis in Register zurückschreiben

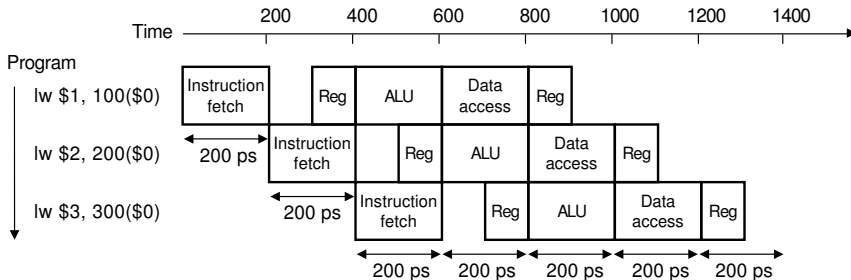
von-Neumann Befehlszyklus (cont.)

- ▶ je nach Instruktion sind nicht alle Schritte notwendig
 - ▶ *nop*: nur Instruction-Fetch
 - ▶ *jump*: kein Speicher- und Registerzugriff
 - ▶ *aluOp*: kein Speicherzugriff
- ▶ Pipeline kann auch feiner unterteilt werden (meist mehr Stufen)

serielle Bearbeitung ohne Pipelining



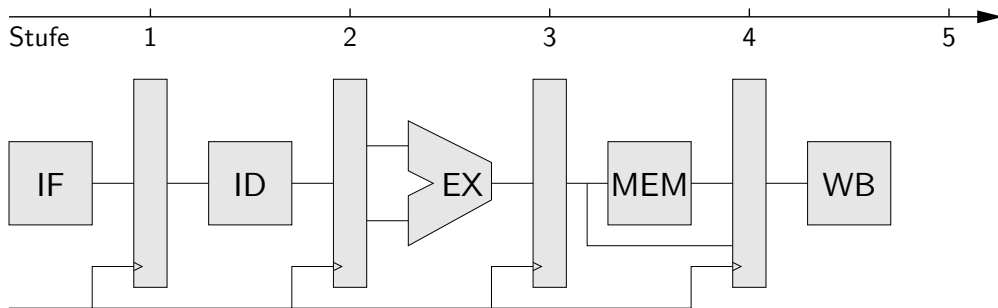
Pipelining für die einzelnen Schritte der Befehlsausführung



[PH22]

- Befehle überlappend ausführen: neue Befehle holen, dann decodieren, während vorherige noch ausgeführt werden
- Register trennen Pipelinestufen

Klassische 5-stufige Pipeline



- ▶ Grundidee der ursprünglichen RISC-Architekturen
- + Durchsatz ca. $3 \dots 5 \times$ besser als serielle Ausführung
- + guter Kompromiss aus Leistung und Hardwareaufwand
- ▶ MIPS-Architektur (aus Patterson, Hennessy [PH22])

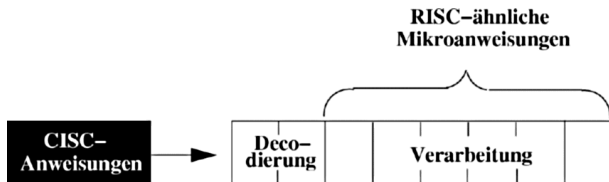
▶ MIPS ohne Pipeline

▶ MIPS Pipeline

▶ Pipeline Schema

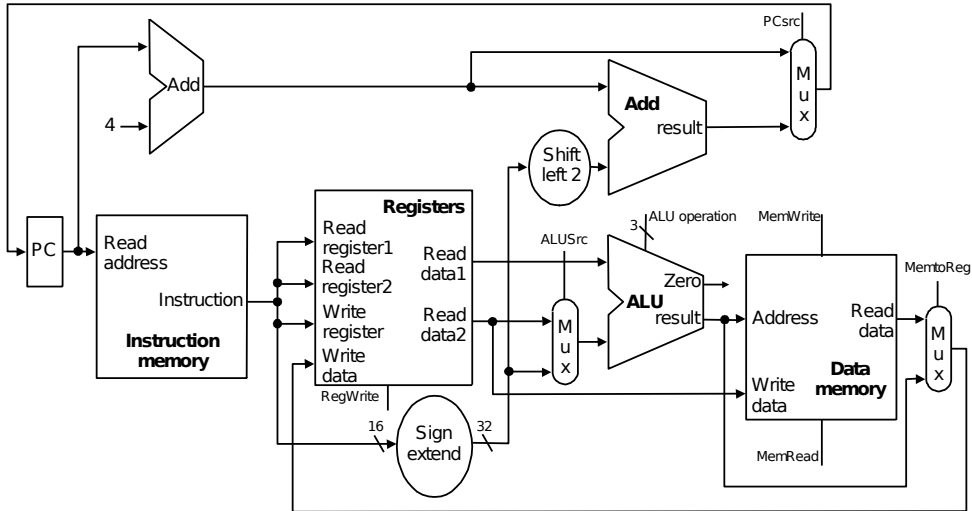
Klassische 5-stufige Pipeline (cont.)

- ▶ RISC ISA: Pipelining wird direkt umgesetzt
 - ▶ Befehlssätze auf diese Pipeline hin optimiert
 - ▶ IBM-801, MIPS R-2000/R-3000 (1985), SPARC (1987)
- ▶ CISC-Architekturen heute ebenfalls mit Pipeline
 - ▶ Motorola 68020 (zweistufige Pipeline, 1984), Intel 486 (1989), Pentium (1993) ...
 - ▶ Befehle in Folgen RISC-ähnlicher Anweisungen umsetzen



- + CISC-Software bleibt lauffähig
- + Befehlssatz wird um neue RISC Befehle erweitert

MIPS: serielle Realisierung ohne Pipeline

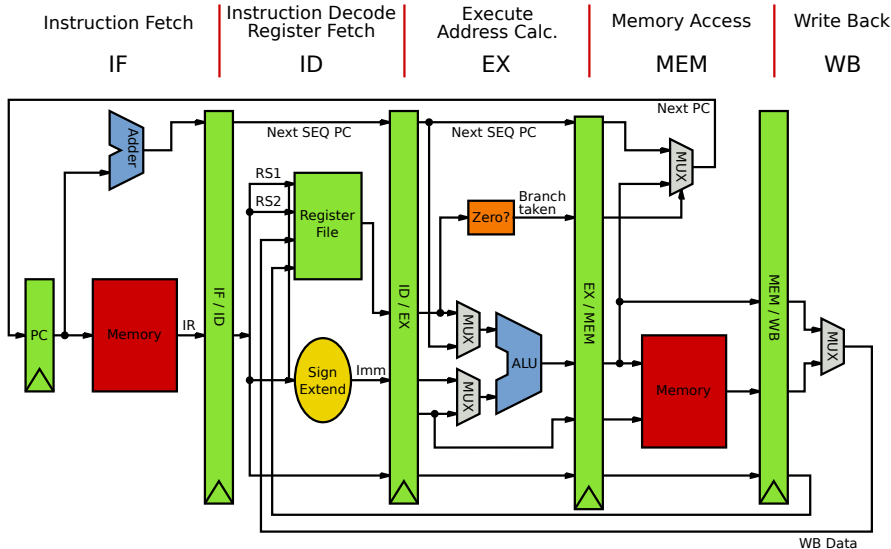


[PH22]

längster Pfad: PC - IM - REG - MUX - ALU - DM - MUX - PC/REG

◀ RISC Pipelining

MIPS: mit 5-stufiger Pipeline



MIPS: mit 5-stufiger Pipeline (cont.)

- ▶ die Hardwareblöcke selbst sind unverändert
 - ▶ PC, Addierer fürs Inkrementieren des PC
 - ▶ Registerbank
 - ▶ Rechenwerke: ALU, sign-extend, zero-check
 - ▶ Multiplexer und Leitungen/Busse
- ▶ vier zusätzliche Pipeline-Register
 - ▶ die (decodierten) Befehle
 - ▶ alle Zwischenergebnisse
 - ▶ alle intern benötigten Statussignale
- ▶ längster Pfad zwischen Registern jetzt eine der 5 Stufen
- ▶ aber wie wirkt sich das auf die Software aus?!

Begriffe

- ▶ **Pipeline-Stage:** einzelne Stufe der Pipeline
- ▶ **Pipeline Machine Cycle:** Instruktion kommt einen Schritt in Pipeline weiter
- ▶ **Durchsatz:** Anzahl der Instruktionen, die je Zeiteinheit abgeschlossen werden
- ▶ **Latenz:** Zeit, die eine Instruktion benötigt, um alle Pipelinestufen zu durchlaufen

Vor- und Nachteile

- + Schaltnetze sind in kleinere Blöcke aufgeteilt
⇒ höhere Taktfrequenzen
- + im Idealfall wird ein neuer Befehl pro Takt gestartet
⇒ höherer Durchsatz, bessere Performanz
- + geringer Zusatzaufwand an Hardware
- + Pipelining ist für den Programmierer nicht direkt sichtbar!
 - Achtung: Daten-/Kontrollabhängigkeiten (s.u.)
- Latenz wird nicht verbessert, bleibt bestenfalls gleich
- Pipeline Takt limitiert durch langsamste Pipelinestufe
unausgewogene Pipelinestufen reduzieren den Takt und damit den Durchsatz
- zusätzliche Zeiten, um Pipeline zu füllen bzw. zu leeren

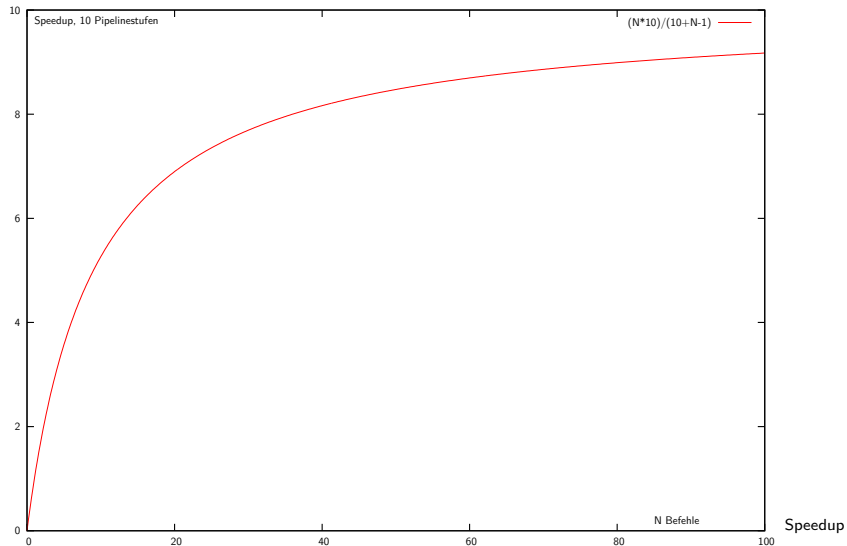
Analyse

- ▶ N Instruktionen; K Pipelinestufen
- ▶ ohne Pipeline: $N \cdot K$ Taktzyklen
- ▶ mit Pipeline: $K + N - 1$ Taktzyklen
- ▶ „Speedup“ $S = \frac{N \cdot K}{K + N - 1}$, $\lim_{N \rightarrow \infty} S = K$

⇒ ein großer Speedup wird erreicht durch

- ▶ große Pipelinetiefe: K
- ▶ lange Instruktionssequenzen: N
- ▶ wegen Daten- und Kontrollabhängigkeiten nicht erreichbar
- ▶ außerdem: Register-Overhead nicht berücksichtigt

Prozessorpipeline – Bewertung (cont.)



- ▶ größeres K wirkt sich direkt auf den Durchsatz aus
- ▶ weniger Logik zwischen den Registern, höhere Taktfrequenzen
- ▶ zusätzlich: technologischer Fortschritt (1985 ... 2017)

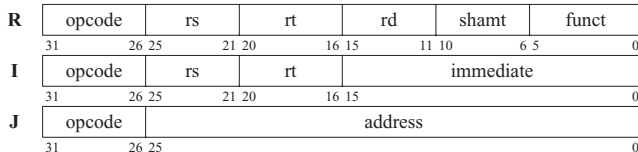
▶ Beispiele	CPU	Pipelinestufen	Taktfrequenz [MHz]
	80386	1	33
	Pentium	5	300
	Motorola G4	4	500
	Motorola G4e	7	1 000
	Pentium II/III	12	1 400
	Athlon XP	10/15	2 500
	Athlon 64, Opteron	12/17	$\leq 3\,000$
	Pentium 4	20 ... 31	$\leq 3\,800$
	Core i-..	14/19	$\leq 4\,200$
	Ryzen ..	19	$\leq 4\,000$

Architekturentscheidungen, die sich auf das Pipelining auswirken

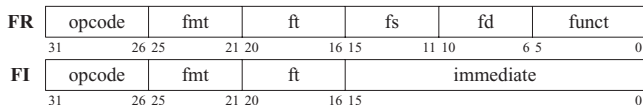
gut für Pipelining

- ▶ gleiche Instruktionslänge
- ▶ wenige Instruktionsformate
- ▶ Load/Store Architektur

BASIC INSTRUCTION FORMATS



FLOATING-POINT INSTRUCTION FORMATS



MIPS-Befehlsformate [PH22]

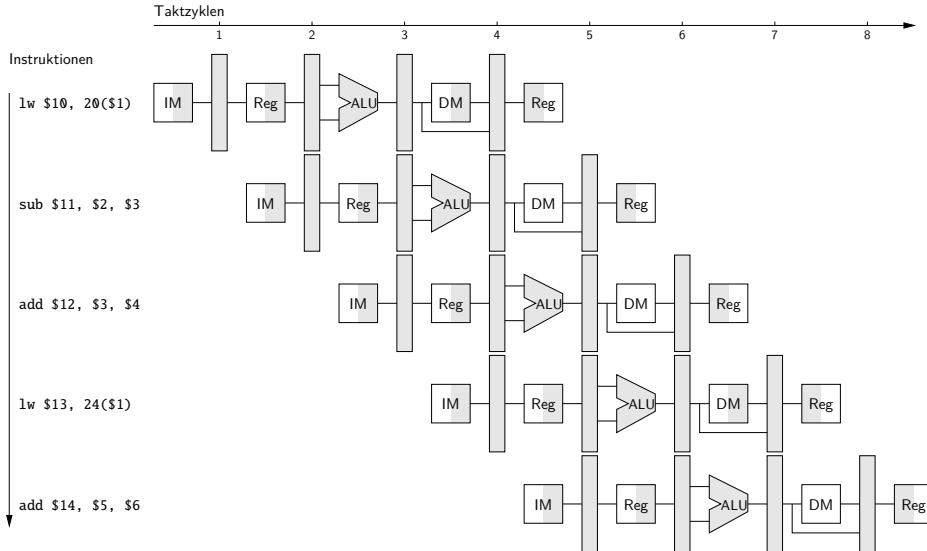
schlecht für Pipelining: *Pipelinekonflikte / -Hazards*

- ▶ Strukturkonflikt: gleichzeitiger Zugriff auf eine Ressource durch mehrere Pipelinestufen
- ▶ Datenkonflikt: Ergebnisse von Instruktionen werden vor dem Write-Back bereits in der Pipeline benötigt
- ▶ Steuerkonflikt: (bedingte) Sprungbefehle verändern die Befehlssequenz innerhalb der Pipeline

sehr schlecht für Pipelining

- ▶ Unterbrechung des Programmkontexts („*Context Switch*“): Interrupt, System-Call, Exception, Prozesswechsel . . .

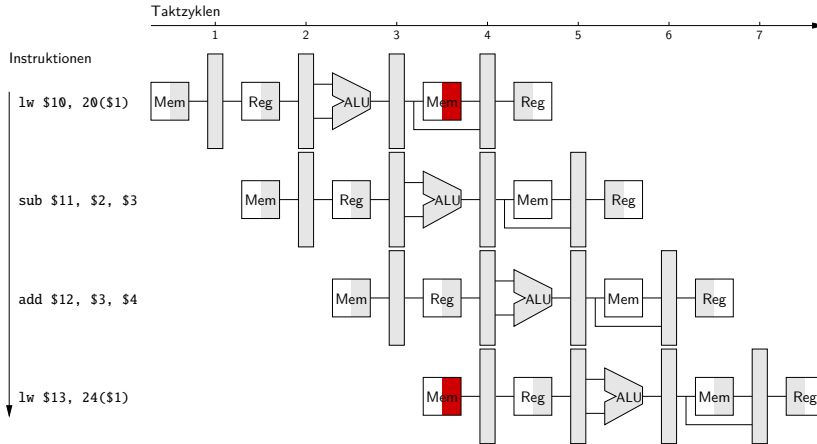
Pipeline Schema



Strukturkonflikt / Structural Hazard

- ▶ mehrere Stufen wollen gleichzeitig auf eine Ressource zugreifen
 - ▶ Beispiel: gleichzeitiger Zugriff auf Speicher
- ⇒ Mehrfachauslegung der betreffenden Ressourcen
- ▶ Harvard-Architektur (getrennte Speicher für Programme und Daten) vermeidet Strukturkonflikt aus Beispiel
 - ▶ Multi-Port Register
 - ▶ mehrfach vorhandene Busse und Multiplexer...

Beispiel für Strukturkonflikt



gleichzeitiges Laden aus *einem* Speicher, zwei verschiedene Adressen

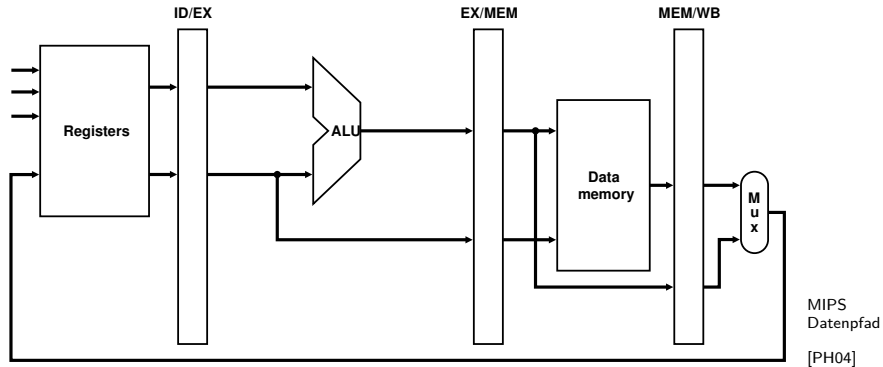
Datenkonflikt / Data Hazard

- ▶ eine Instruktion braucht die Ergebnisse einer vorhergehenden, diese wird aber noch in der Pipeline bearbeitet
 - ▶ Datenabhängigkeiten aufeinanderfolgender Befehle
 - ▶ Operanden während ID-Phase aus Registerbank lesen
 - ▶ Resultate werden erst in WB-Phase geschrieben
- ⇒ aber: Resultat ist schon nach EX-/MEM-Phase bekannt

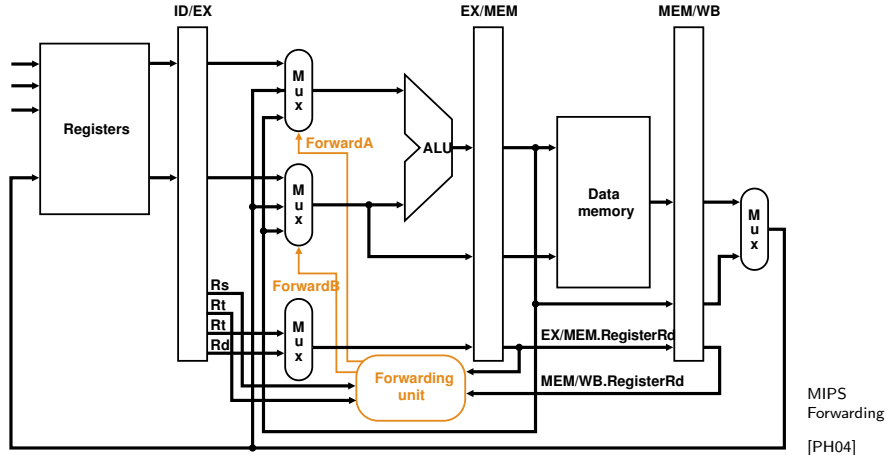
Forwarding

- ▶ zusätzliche Hardware („*Forwarding-Unit*“) kann Datenabhängigkeiten auflösen
- ▶ Änderungen in der Pipeline Steuerung
- ▶ neue Datenpfade und Multiplexer

Pipeline Datenkonflikte (cont.)



Pipeline Datenkonflikte (cont.)



Rückwärtsabhängigkeiten

- ▶ spezielle Datenabhängigkeit
- ▶ Forwarding-Technik funktioniert nicht, da die Daten erst *später* zur Verfügung stehen
 - ▶ bei längeren Pipelines
 - ▶ bei Load-Instruktionen (s.u.)

Auflösen von Rückwärtsabhängigkeiten

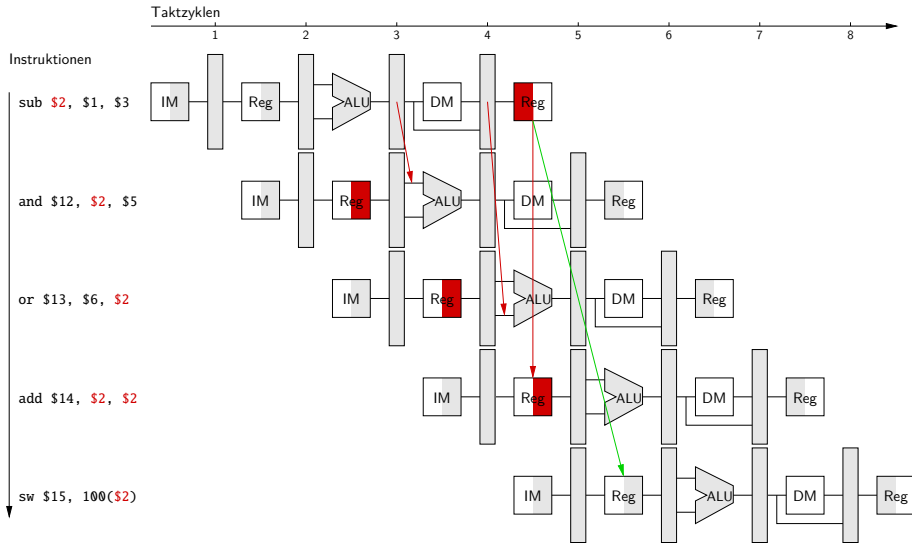
1. Softwarebasiert, durch den Compiler, Reihenfolge der Instruktionen verändern
 - ▶ andere Operationen (ohne Datenabhängigkeiten) vorziehen
 - ▶ nop-Befehl(e) einfügen

2. „Interlocking“

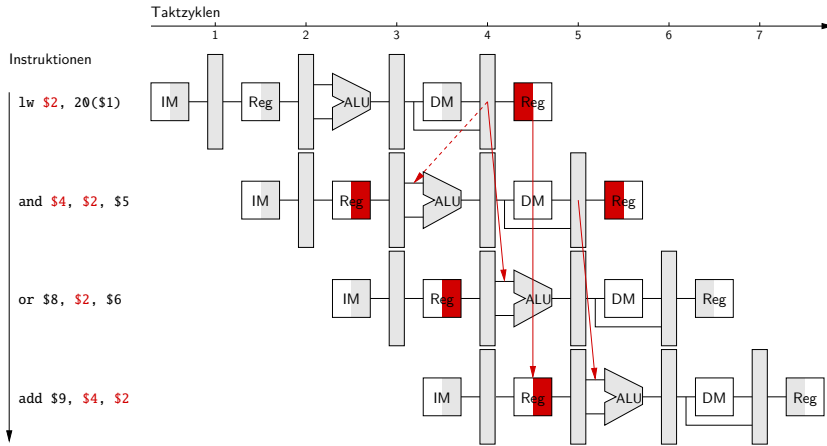
- ▶ zusätzliche (Hardware) Kontrolleinheit: komplexes Steuerwerk
- ▶ automatisches Stoppen der Pipeline, bis die benötigten Daten zur Verfügung stehen – Strategien:
 - ▶ in Pipeline werden keine neuen Instruktionen geladen
 - ▶ Hardware erzeugt: Pipelineleerlauf / „*pipeline stall*“

„Scoreboard“

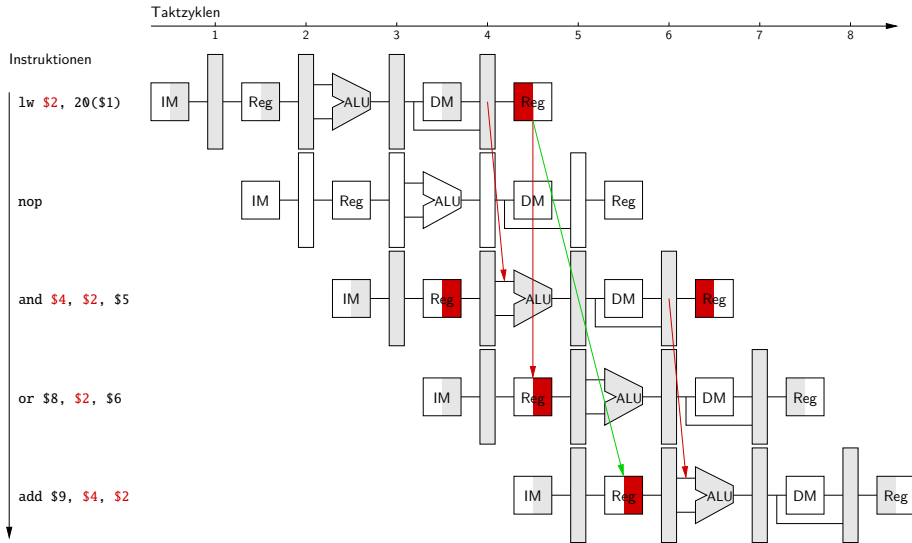
- ▶ Hardware Einheit zur zentralen Hazard-Erkennung und -Auflösung
- ▶ Verwaltet Instruktionen, benutzte Einheiten und Register der Pipeline (siehe „*Superskalare Rechner*“, ab Folie 1223)



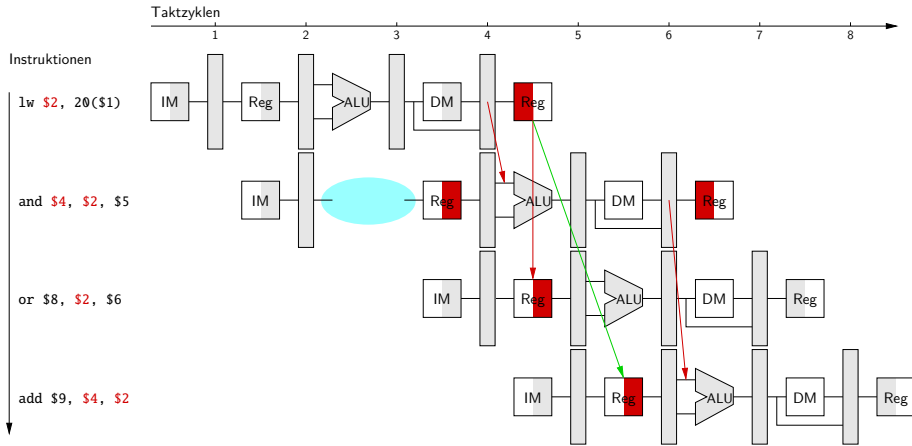
Befehle wollen R2 lesen, während es noch vom ersten Befehl berechnet wird



Befehle wollen R2 lesen, bevor es aus Speicher geladen wird



Compiler kennt Hardware und hat einen nop-Befehl eingefügt



Hardware verzögert Bearbeitung, bis Konflikte beseitigt sind („bubbles“)

Steuerkonflikt / Control Hazard

- ▶ Unterbrechung des sequenziellen Ablaufs durch Sprungbefehle `jmp` und Unterprogrammaufrufe: `call` und `ret`
 - ▶ Instruktionen die auf (bedingte) Sprünge folgen, werden bereits in die Pipeline geschoben
 - ▶ Sprungadresse und Status (*branch taken/untaken*) sind aber erst am Ende der EX-Phase bekannt
 - ▶ einige Befehle wurden bereits teilweise ausgeführt, Resultate eventuell „ge-forwarded“
- alle Zwischenergebnisse müssen verworfen werden
 - ▶ inklusive aller Forwarding-Daten
 - ▶ Pipeline an korrekter Zieladresse neu starten
 - ▶ erfordert sehr komplexe Hardware
- jeder (ausgeführte) Sprung kostet enorm Performance

Lösungsmöglichkeiten für Steuerkonflikte

- ▶ „*Interlocking*“: Pipeline prinzipiell bei Sprüngen leeren
 - ineffizient: ca. 19 % der Befehle sind Sprünge
- 1. Annahme: nicht ausgeführter Sprung („*untaken branch*“)
 - + kaum zusätzliche Hardware
 - im Fehlerfall
 - ▶ Pipeline muss geleert werden („*flush instructions*“)
- 2. Sprungentscheidung „vorverlegen“
 - ▶ Software: Compiler zieht andere Instruktionen vor
Verzögerung nach Sprungbefehl („*delay slots*“)
 - ▶ Hardware: Sprungentscheidung durch Zusatz-ALU
(nur Vergleiche) während Befehlsdecodierung (z.B. MIPS)

3. Sprungvorhersage („branch prediction“)

- ▶ Beobachtung: ein Fall tritt häufiger auf;
Schleifendurchlauf, Datenstrukturen durchsuchen etc.
- ▶ mehrere Vorhersageverfahren; oft miteinander kombiniert
- + hohe Trefferquote: bis 90 %

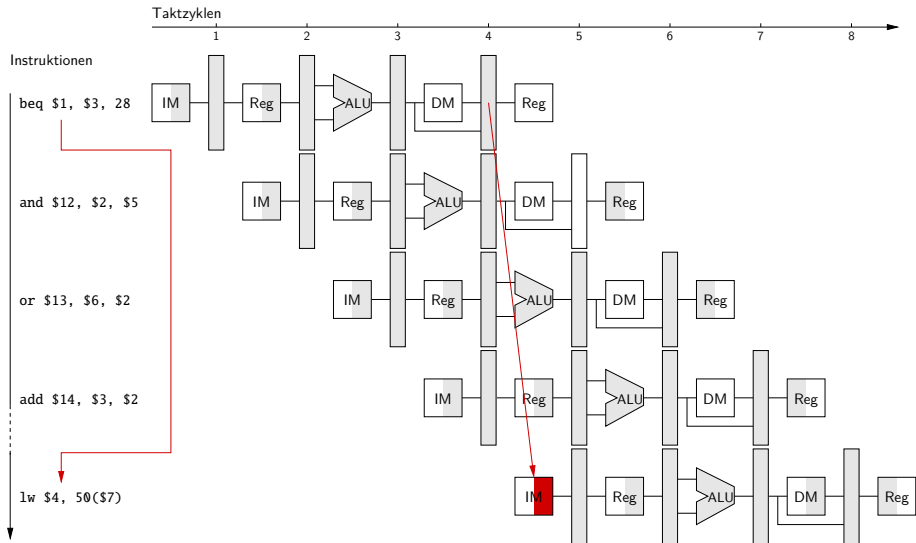
Statische Sprungvorhersage (softwarebasiert)

- ▶ Compiler erzeugt extra Bit in Opcode des Sprungbefehls
- ▶ Methoden: Codeanalyse, Profiling...

Dynamische Sprungvorhersage (hardwarebasiert)

- ▶ Sprünge durch Laufzeitinformation vorhersagen:
Wie oft wurde der Sprung in letzter Zeit ausgeführt?
- ▶ viele verschiedene Verfahren:
History-Bit, 2-Bit Prädiktor, korrelationsbasierte Vorhersage, Branch History Table, Branch Target Cache...

- ▶ Schleifen abrollen / „*Loop unrolling*“
 - ▶ zusätzliche Maßnahme zu allen zuvor skizzierten Verfahren
 - ▶ bei statischer Schleifenbedingung möglich
 - ▶ Compiler iteriert Instruktionen in der Schleife (teilweise)
 - längerer Code
 - + Sprünge und Abfragen entfallen
 - + erzeugt sehr lange Codesequenzen ohne Sprünge
 - ⇒ Pipeline kann optimal ausgenutzt werden



- ▶ von-Neumann Zyklus auf separate Phasen aufteilen
- ▶ überlappende Ausführung von mehreren Befehlen
 - ▶ einfachere Hardware für jede Phase $\Rightarrow$ höherer Takt
 - ▶ mehrere Befehle in Bearbeitung $\Rightarrow$ höherer Durchsatz
 - ▶ 5-stufige RISC-Pipeline: IF $\rightarrow$ ID/OF $\rightarrow$ Exe $\rightarrow$ Mem $\rightarrow$ WB
 - ▶ mittlerweile sind 9...20 Stufen üblich
- ▶ Struktur-, Daten- und Steuerkonflikte
 - ▶ Lösung durch mehrfache/bessere Hardware
 - ▶ Data-Forwarding umgeht viele Datenabhängigkeiten
 - ▶ Sprungbefehle sind ein ernstes Problem
- ▶ Pipelining ist prinzipiell unabhängig von der ISA
 - ▶ einige Architekturen basieren auf Pipelining (MIPS)
 - ▶ Compiler/Tools/Programmierer sollten CPU Pipeline kennen

- ▶ Simulationen, Wettervorhersage, Gentechnologie ...
 - ▶ Datenbanken, Transaktionssysteme, Suchmaschinen ...
 - ▶ Softwareentwicklung, Schaltungsentwurf ...

 - ▶ Performanz eines einzelnen Prozessors ist begrenzt
- ⇒ Hardware: Verteilen eines Programms auf mehrere Prozessoren
- ⇒ Software: kommunizierende Prozesse und Multithreading

Vielfältige Möglichkeiten

- ▶ wie viele und welche Prozessoren?
- ▶ Kommunikation zwischen den Prozessoren?
- ▶ Programmierung und Software/Tools?

- ▶ **Antwortzeit:** die Gesamtzeit zwischen Programmstart und -ende, inklusive I/O-Operationen, Unterbrechungen etc.
(„wall clock time“, „response time“, „execution time“)

$$\text{performance} = \frac{1}{\text{execution time}}$$

- ▶ **Ausführungszeit:** reine CPU-Zeit

Unix time-Befehl: $\frac{597.07\text{u } 0.15\text{s } 9:57.61 \text{ } 99.9\%}{597.07 \text{ user CPU time [sec.]}$
0.15 system CPU time
9:57.61 elapsed time
99.9 CPU/elapsed [%]

- ▶ **Durchsatz:** Anzahl der bearbeiteten Programme / Zeit
- ▶ **Speedup:** $s = \frac{\text{performance } x}{\text{performance } y} = \frac{\text{execution time } y}{\text{execution time } x}$

Wie kann man Performanz verbessern?

- ▶ Ausführungszeit = $\langle \text{Anzahl der Befehle} \rangle \cdot \langle \text{Zeit pro Befehl} \rangle$

- ▶ weniger Befehle
 - ▶ *gute* Algorithmen
 - ▶ bessere Compiler
 - ▶ mächtigere Befehle (CISC)

- ▶ weniger Zeit pro Befehl
 - ▶ bessere Technologie
 - ▶ Architektur: Pipelining, Caches ...
 - ▶ einfachere Befehle (RISC)

- ▶ parallele Ausführung
 - ▶ superskalare Architekturen, SIMD, MIMD

Möglicher Speedup durch Beschleunigung einer Teilfunktion?

1. **System** Programm P wird ausgeführt,
darin Funktion X mit Anteil $0 < f < 1$ der Gesamtzeit
2. **System** Programm P wird ausgeführt,
Funktion X' ist schneller als X mit Speedup S_X

Amdahl's Gesetz

Gene Amdahl, Architekt der IBM S/360, 1967

► Speedup

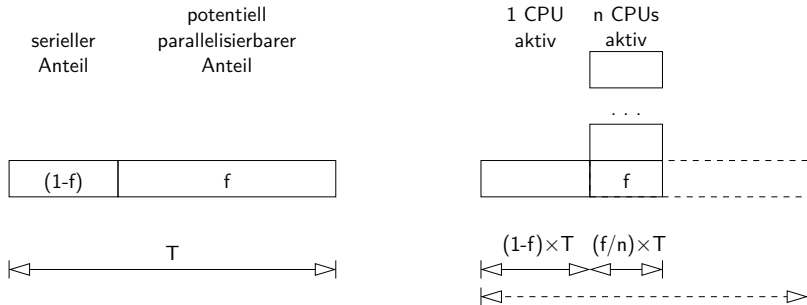
$$S_{gesamt} = \frac{1}{(1 - f) + f/S_X}$$

Amdahl's Gesetz (cont.)

Speedup

$$S_{gesamt} = \frac{1}{(1-f) + f/S_x}$$

- ▶ nur ein Teil f des Gesamtproblems wird beschleunigt



⇒ möglichst großer Anteil f

⇒ Optimierung lohnt nur für relevante Operationen

allgemeingültig: entsprechend auch für Projektplanung, Verkehr ...

- ▶ ursprüngliche Idee: Parallelrechner mit n -Prozessoren

Speedup
$$S_{gesamt} = \frac{1}{(1 - f) + k(n) + f/n}$$

n # Prozessoren als Verbesserungsfaktor

f Anteil parallelisierbarer Berechnung

$1 - f$ Anteil nicht parallelisierbarer Berechnung

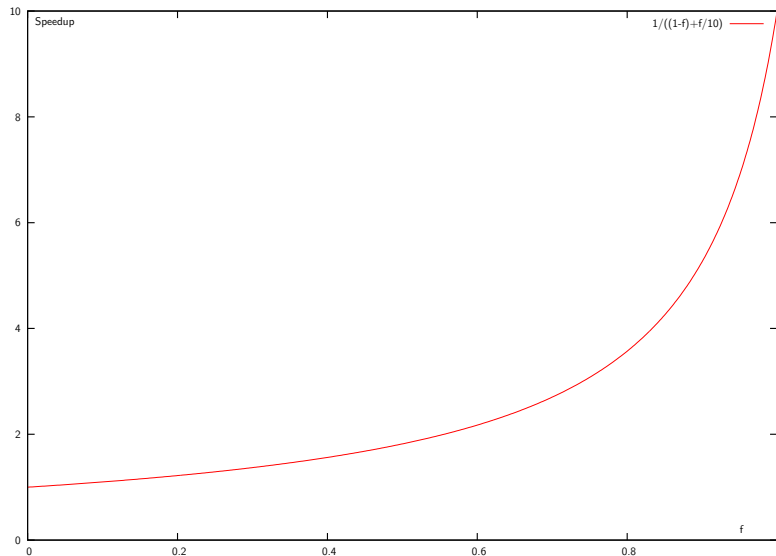
$k()$ Kommunikationsoverhead zwischen den Prozessoren

- ▶ Aufgaben verteilen
- ▶ Arbeit koordinieren
- ▶ Ergebnisse zusammensammeln

S_X	f	S_{gesamt}
10	0,1	$1/(0,9 + 0,01) = 1,1$
2	0,5	$1/(0,5 + 0,25) = 1,33$
2	0,9	$1/(0,1 + 0,45) = 1,82$
1,1	0,98	$1/(0,02 + 0,89) = 1,1$
4	0,5	$1/(0,5 + 0,125) = 1,6$
4 536	0,8	$1/(0,2 + 0,0 \dots) = 5,0$
9 072	0,99	$1/(0,01 + 0,0 \dots) = 98,92$

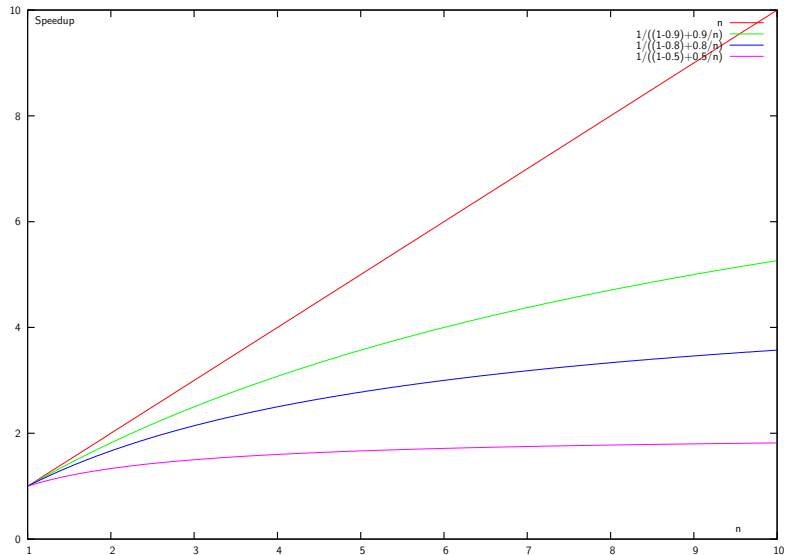
- ▶ Optimierung bringt nichts, wenn der nicht beschleunigte „serielle“ Anteil $(1 - f)$ eines Programms überwiegt
- n -Prozessoren (große S_X) wirken *nicht linear*
- geringe erreichbare Parallelität in Hochsprachen-Programmen, typisch $S_{gesamt} \leq 4$
- + oft viele Prozesse/Tasks, jeweils unabhängig voneinander: Serveranwendungen, virtuelle Maschinen, Container ...

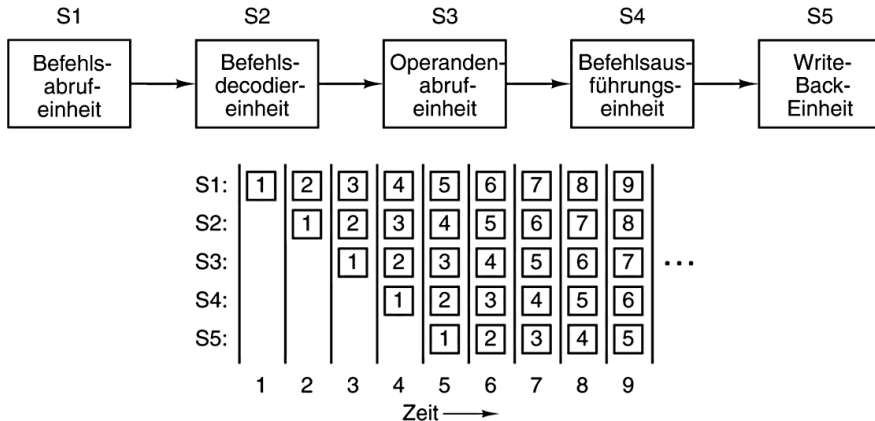
Amdahl's Gesetz: Beispiele (cont.)



$$S_X = 10$$

Amdahl's Gesetz: Beispiele (cont.)

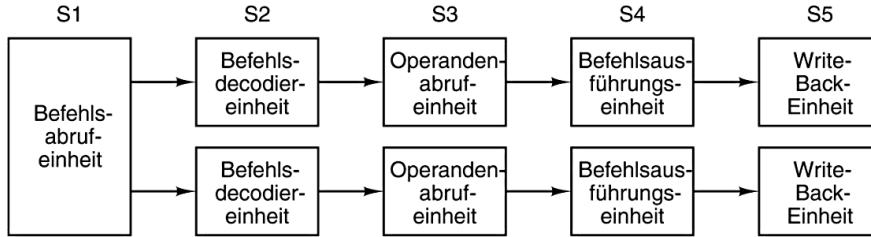




[TA14]

- ▶ Befehl in kleinere, schnellere Schritte aufteilen $\Rightarrow$ höherer Takt
- ▶ mehrere Instruktionen überlappt ausführen $\Rightarrow$ höherer Durchsatz

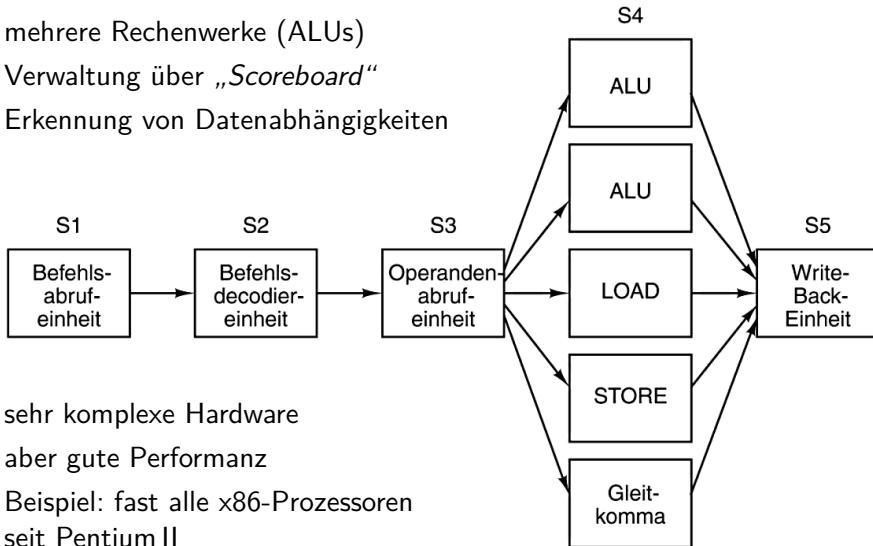
Parallele Pipelines



[TA14]

- ▶ im Bild jeweils zwei Operationen pro Pipelinestufe
- ▶ parallele („superskalare“) Ausführung
- ▶ komplexe Hardware (Daten- und Kontrollabhängigkeiten)
- ▶ Beispiel: Pentium

- ▶ mehrere Rechenwerke (ALUs)
 - ▶ Verwaltung über „Scoreboard“
- Erkennung von Datenabhängigkeiten



- ▶ sehr komplexe Hardware
- ▶ aber gute Performanz
- ▶ Beispiel: fast alle x86-Prozessoren seit Pentium II

- ▶ Superskalare CPUs besitzen mehrere Recheneinheiten: 4 . . . 12
 - ▶ in jedem Takt werden (dynamisch) mehrere Instruktionen eines konventionell linearen Instruktionsstroms abgearbeitet
- ⇒ ILP (Instruction **L**evel **P**arallelism)
- ▶ Hardware verteilt initiierte Instruktionen auf Recheneinheiten
 - ▶ pro Takt kann *mehr als eine* Instruktion initiiert werden
Die Anzahl wird dynamisch von der Hardware bestimmt:
0 . . . „*Instruction Issue Bandwidth*“
- + sehr effizient, alle modernen CPUs sind superskalar
 - Abhängigkeiten zwischen Instruktionen sind der Engpass,
das Problem von Hazards wird verschärft

Datenabhängigkeiten

- ▶ RAW – **R**ead **A**fter **W**rite
Instruktion I_x darf Datum erst lesen, wenn I_{x-n} geschrieben hat
- ▶ WAR – **W**rite **A**fter **R**ead
Instruktion I_x darf Datum erst schreiben, wenn I_{x-n} gelesen hat
- ▶ WAW – **W**rite **A**fter **W**rite
Instruktion I_x darf Datum erst überschreiben, wenn I_{x-n} geschrieben hat

Datenabhängigkeiten superskalarer Prozessoren

- ▶ RAW: echte Abhängigkeit; Forwarding ist kaum möglich und in superskalaren Pipelines extrem aufwändig
- ▶ WAR, WAW: „*Register Renaming*“ als Lösung

- ▶ Hardware löst (einige) Datenabhängigkeiten der Pipeline auf
- ▶ zwei Arten von Registersätzen
 1. Architektur-Register: „logische Register“ der ISA
 2. viele Hardware-Register: „Rename Register“ (180 Int, 168 FP)
 - ▶ dynamische Abbildung von ISA- auf Hardware-Register
- Kontextwechsel aufwändig: „Rename Register“ speichern
- ▶ Beispiel

▶ Originalcode

```
tmp  = a + b;  
res1 = c + tmp;  
tmp  = d + e;  
res2 = tmp - f;
```

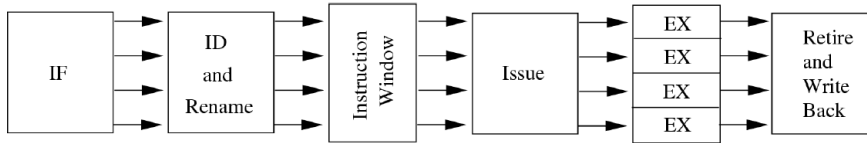
nach Renaming

```
tmp1 = a + b;  
res1 = c + tmp1;  
tmp2 = d + e;  
res2 = tmp2 - f;  
tmp  = tmp2;
```

▶ Parallelisierung des modifizierten Codes

```
tmp1 = a + b;      tmp2 = d + e;  
res1 = c + tmp1;   res2 = tmp2 - f;      tmp = tmp2;
```

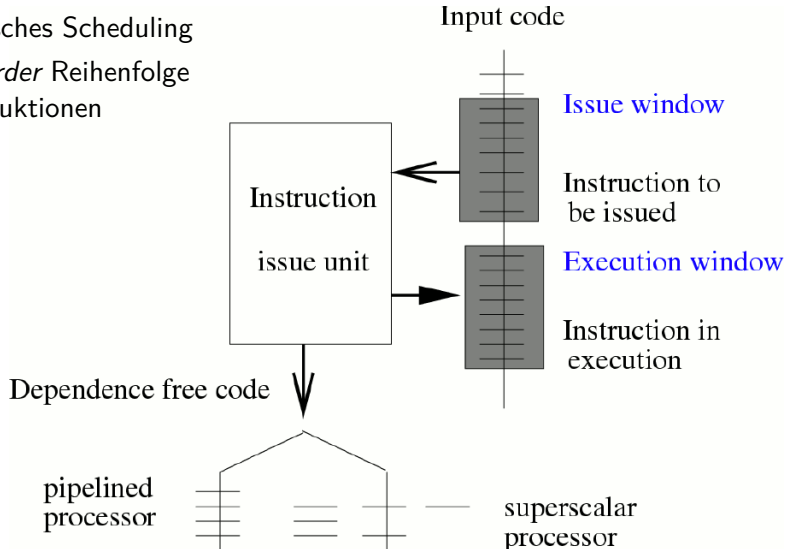
Aufbau der superskalaren Pipeline



- ▶ lange Pipelines mit vielen Phasen: Fetch (Prefetch, Predecode), Decode / Register-Renaming, Issue, Dispatch, Execute, Retire (Commit, Complete / Reorder), Write-Back
- ▶ je nach Implementation unterschiedlich aufgeteilt
- ▶ entscheidend für superskalare Architektur sind die Schritte
vor den ALUs: Issue, Dispatch $\Rightarrow$ *out-of-order* Ausführung
nach -"- : Retire $\Rightarrow$ *in-order* Ergebnisse

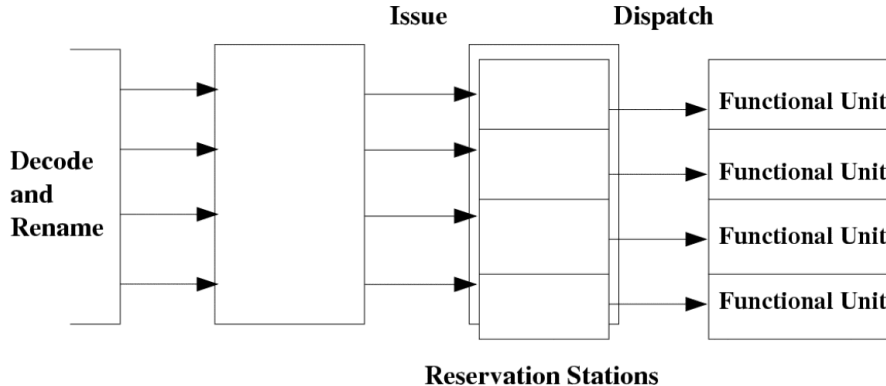
Superskalar – Pipeline (cont.)

- Dynamisches Scheduling
- ⇒ *out-of-order* Reihenfolge der Instruktionen



Superskalar – Pipeline (cont.)

- Issue: globale Sicht
- Dispatch: getrennte Ausschnitte in „*Reservation Stations*“



- ▶ Reservation Station für jede Funktionseinheit
 - ▶ speichert: initiierte Instruktionen die auf Recheneinheit warten
 - ▶ –"– zugehörige Operanden
 - ▶ –"– ggf. Zusatzinformation
 - ▶ Instruktion bleibt blockiert, bis alle Parameter bekannt sind
wird dann an die zugehörige ALU weitergeleitet
- ▶ ggf. „Retire“-Stufe
 - ▶ Reorder-Buffer: erzeugt wieder *in-order* Reihenfolge
 - ▶ commit: „richtig ausgeführte“ Instruktionen gültig machen
 - ▶ abort: Instruktionen verwerfen, z.B. Sprungvorhersage falsch
- ▶ Dynamisches Scheduling: zuerst '67 in IBM 360 (R. Tomasulo)
 - ▶ Forwarding
 - ▶ Registerumbenennung und Reservation Stations

Spezielle Probleme superskalarer Pipelines: weitere Hazard-Möglichkeiten

- komplexe Datenabhängigkeiten
 - ▶ die verschiedenen ALUs haben unterschiedliche Latenzzeiten
 - ▶ Befehle „warten“ in den Reservation Stations
- ⇒ Datenabhängigkeiten können sich mit jedem Takt ändern
- Kontrollflussabhängigkeiten
 - ▶ Anzahl der Instruktionen zwischen bedingten Sprüngen limitiert Anzahl parallelisierbarer Instruktionen
- ⇒ Kontextwechsel noch aufwändiger, muss ggf. warten
- ⇒ Optimierungstechniken wichtig
 - ▶ optimiertes (dynamisches) Scheduling: Scoreboard, Tomasulo-Algorithmus
 - ▶ „*Loop Unrolling*“ (längere Codesequenzen ohne Sprünge)

Softwareunterstützung für Pipelining superskalarer Prozessoren: „*Software Pipelining*“

- ▶ Codeoptimierungen beim Compilieren als notwendiger Ersatz/Ergänzung der Pipelineunterstützung durch Hardware
- ▶ Compiler hat „globalen“ Überblick

⇒ zusätzliche Optimierungsmöglichkeiten

- ▶ superskalare Architektur (mehrere ALUs)
- ▶ CISC-Befehle werden dynamisch in „ μ OPs“ (1...3) umgesetzt
- ▶ Ausführung der μ OPs mit „Out of Order“ Maschine, wenn
 - ▶ Operanden verfügbar sind
 - ▶ funktionelle Einheit (ALU) frei ist
- ▶ Ausführung wird durch „Reservation Stations“ kontrolliert
 - ▶ beobachtet die Datenabhängigkeiten zwischen μ OPs
 - ▶ teilt Ressourcen zu
- ▶ „Trace“ Cache
 - ▶ ersetzt traditionellen Anweisungs-cache
 - ▶ speichert Anweisungen in decodierter Form: Folgen von μ OPs
 - ▶ reduziert benötigte Rate für den Anweisungsdecoder
- ▶ „Double pumped“ ALUs (2 Operationen pro Taktzyklus)

Beispiel: Pentium 4 / NetBurst Architektur (cont.)

- ▶ große Pipelinelänge $\Rightarrow$ sehr hohe Taktfrequenzen

Basic Pentium III Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

Basic Pentium 4 Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC	Nxt IP	TC	Fetch	Drive	Alloc	Rename	Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive	

- ▶ umfangreiches Material von Intel unter:
ark.intel.com, www.intel.com

Beispiel: Pentium 4 / NetBurst Architektur (cont.)

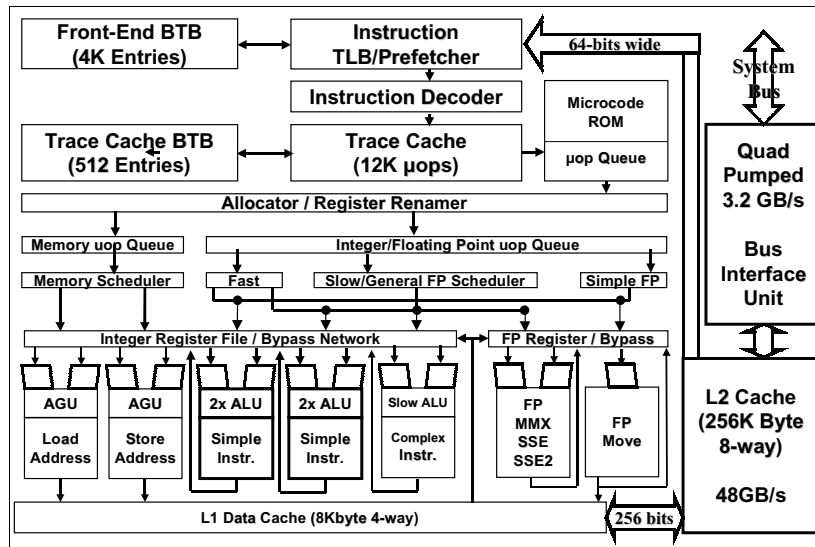
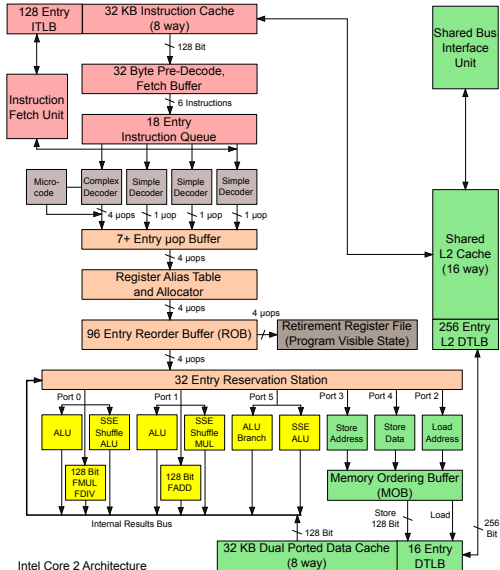


Figure 4: Pentium® 4 processor microarchitecture

Intel: Q1, 2001 [Int]

Beispiel: Core 2 Architektur



Intel Core 2 Architecture

- ▶ *Moore's Law* – technischer Fortschritt
 - ▶ immer schnellere Schaltungen
 - ▶ immer mehr Transistoren pro IC möglich
- ▶ Taktfrequenzen > 10 GHz nicht sinnvoll realisierbar
 - ▶ hoher Takt nur bei einfacher Hardware möglich
 - ▶ Stromverbrauch bei CMOS proportional zum Takt

⇒ höhere Rechenleistung durch Mehrprozessorsysteme

- ▶ Datenaustausch
 1. gemeinsamer Speicher („*Shared-memory*“) oder
 2. Verbindungsnetzwerk („*Message-passing*“)
- ▶ Probleme
 - ▶ Overhead durch Kommunikation
 - ▶ Parallelität in den Algorithmen
 - ▶ Komplexität bei der Programmierung

SISD „*Single Instruction, Single Data*“

- ▶ jeder klassische von-Neumann Rechner, z.B. PC

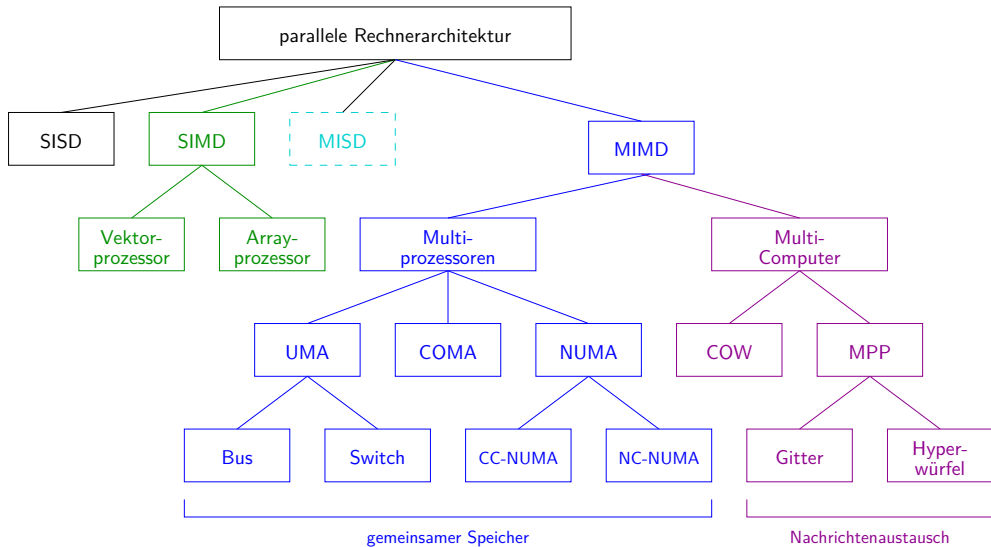
SIMD „*Single Instruction, Multiple Data*“

- ▶ Vektorrechner/Feldrechner
z.B. Connection-Machine 2: 65 536 Prozessoren
- ▶ Befehlssatzerweiterungen um superskalare Recheneinheiten direkt anzusprechen
z.B. x86 MMX, SSE, VLIW-Befehle: 2 ... 8 fach parallel

MIMD „*Multiple Instruction, Multiple Data*“

- ▶ Multiprozessormaschinen
z.B. Compute-Cluster, aber auch Multi-Core CPU

MISD „*Multiple Instruction, Single Data*“ :-)

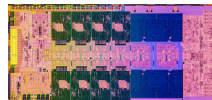


- ▶ Programmierung: ein ungelöstes Problem
 - ▶ Aufteilung eines Programms auf die CPUs/Rechenknoten?
 - ▶ insbesondere bei komplexen Kommunikationsnetzwerken
- ▶ Programme sind nur teilweise parallelisierbar
 - ▶ Parallelität einzelner Programme meist < 8
gilt für Desktop-, Server-, Datenbank Anwendungen etc.
 - ⇒ hochgradig parallele Rechner sind dann Verschwendung
- ▶ *Wohin mit den Transistoren aus „Moore's Law“?*
 - ⇒ SMP-/Mehrkern-CPU's (2...64 Proz.) sind technisch attraktiv
- ▶ Grafikprozessoren (GPUs)
 - ▶ neben 3D-Grafik zunehmender Computing-Einsatz (OpenCL)
 - ▶ Anwendungen: Numerik, Simulation, „Machine Learning“ ...
 - ▶ hohe Fließkomma-Rechenleistung

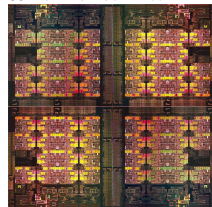
SMP: *Symmetric Multiprocessing*

- ▶ mehrere (gleichartige) Prozessoren
- ▶ gemeinsamer Hauptspeicher und I/O-Einheiten
- ▶ Zugriff über Verbindungsnetzwerk oder Bus
- ▶ geringer Kommunikationsoverhead
- + Bus-basierte Systeme sind sehr kostengünstig
- aber schlecht skalierbar: Bus als Flaschenhals!
- Konsistenz der Daten
 - ▶ lokale Caches für gute Performanz notwendig
 - ▶ Hauptspeicher und Cache(s): Cache-Kohärenz MESI-Protokoll und „*Snooping*“
- siehe 1.4 *Speicherhierarchie – Cache Speicher*
 - ▶ Registerinhalte: ? **problematisch**
- Prozesse wechseln CPUs: „*Hopping*“
- ▶ Multi-Core Prozessoren sind „SMP on-a-chip“

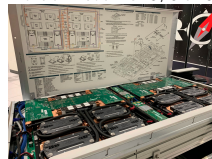
8+16-Kern Core-i9 14900K



60-Kern 4th Gen. Xeon

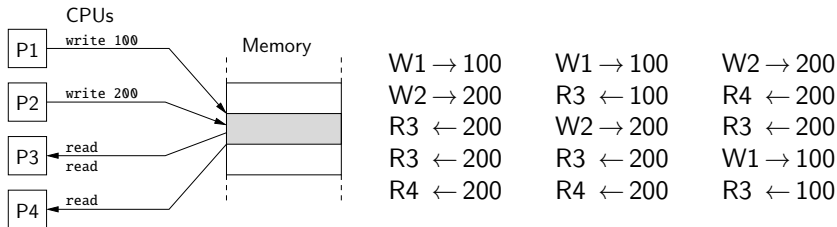


Frontier Blade 2 CPU, 8 GPU



Symmetric Multiprocessing

- ▶ alle CPUs gleichrangig, Zugriff auf Speicher und I/O
- ▶ Konsistenz: *Gleichzeitiger Zugriff auf eine Speicheradresse?*



⇒ „Locking“ Mechanismen und Mutexe

- ▶ spez. Befehle, atomare Operationen, Semaphore etc.
- ▶ explizit im Code zu programmieren

→ siehe ?? Betriebssysteme – Synchronisation und Kommunikation

Cache für schnelle Prozessoren notwendig

- ▶ jede CPU hat eigene Caches: L1, L2
- ▶ aber gemeinsamer Hauptspeicher

Problem der *Cache-Kohärenz*

- ▶ Prozessor P_2 greift auf Daten zu, die im Cache von P_1 liegen
 - ▶ P_2 Lesezugriff: P_1 muss seinen Wert P_2 liefern
 - ▶ P_2 Schreibzugriff: P_1 muss Wert von P_2 übernehmen
oder seine Caches ungültig machen
 - ▶ *Was ist mit gleichzeitigen Zugriffen von P_1 , P_2 ?*
 - ▶ diverse Protokolle zur Cache-Kohärenz
 - ▶ z.B. MESI-Protokoll mit „*Snooping*“: *Modified, Exclusive, Shared, Invalid*
 - ▶ Caches enthalten Wert, Tag und 2 bit MESI-Zustand
- siehe 1.4 *Speicherhierarchie – Cache Speicher*, ab Folie 93

- ▶ MESI-Verfahren garantiert Cache-Kohärenz für Werte in Cache und Hauptspeicher

Vorsicht: Was ist mit den Registern?

- ▶ Variablen in Registern werden von MESI nicht erkannt
- ▶ Compiler versucht, häufig benutzte Variablen soweit möglich in Registern zu halten
- ▶ globale/*shared*-Variablen niemals in Registern halten
- ▶ Java, C: Deklaration als `volatile`

SMP: Erreichbarer Speedup (bis 32 Threads)

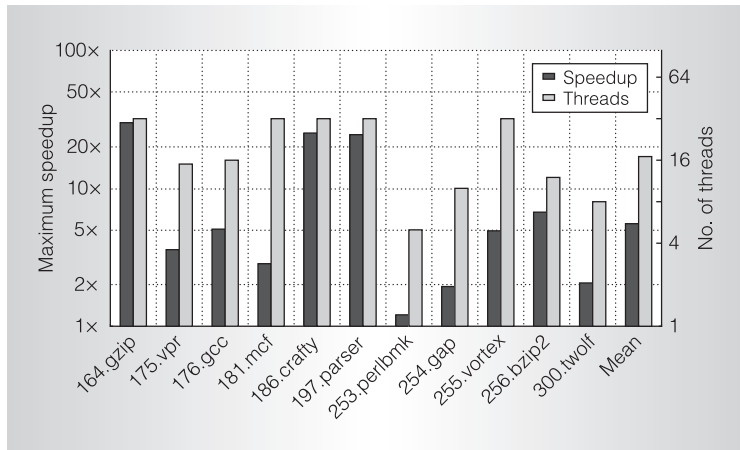
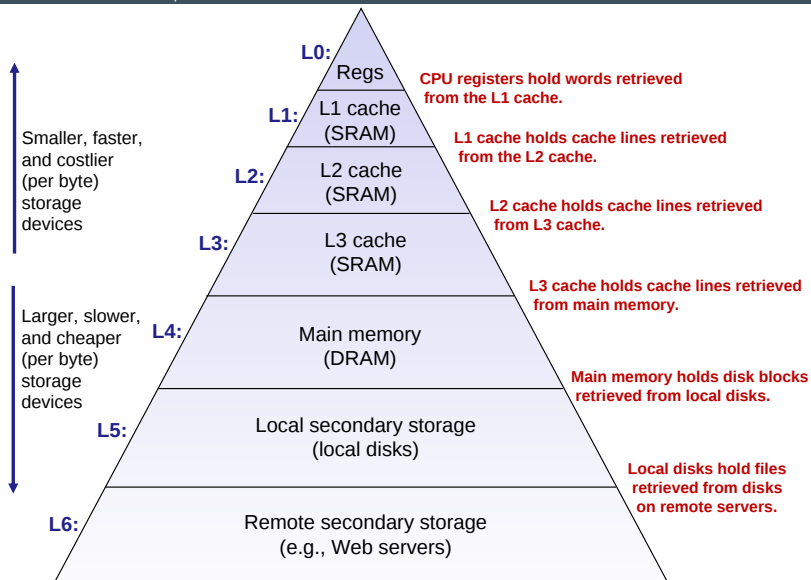


Figure 4. Maximum speedup achieved on up to 32 threads over single-threaded execution (black bars) and minimum number of threads at which the maximum speedup occurred (gray bars).

Speicherhierarchie



Gesamtsystem kombiniert verschiedene Speicher

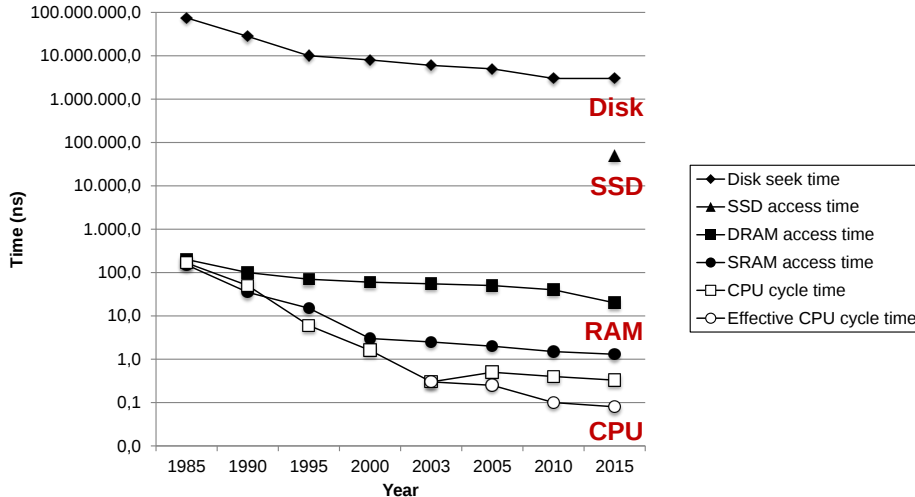
- ▶ wenige KByte Register (-bank) im Prozessor
- ▶ einige MByte SRAM als schneller Zwischenspeicher
- ▶ einige GByte DRAM als Hauptspeicher
- ▶ einige TByte Festplatte als nichtflüchtiger Speicher
- ▶ Hintergrundspeicher (CD/DVD/BR, Magnetbänder)
- ▶ das WWW und Cloud-Services

Kompromiss aus Kosten, Kapazität, Zugriffszeit

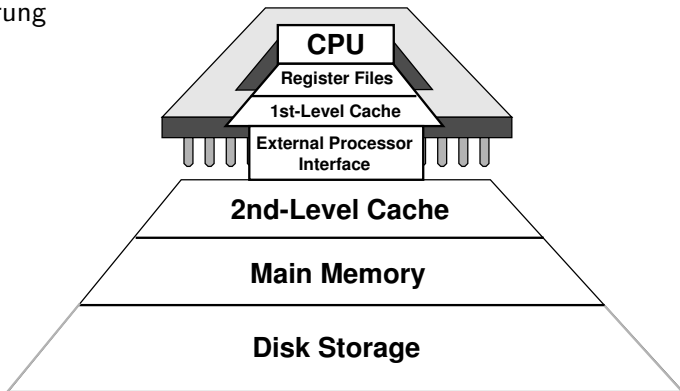
- ▶ Illusion aus großem schnellem Speicher
- ▶ funktioniert nur wegen räumlicher/zeitlicher Lokalität

Prozessor-Memory Gap

- ▶ stetig wachsende Lücke zwischen CPU-, Memory- und Disk-Geschwindigkeiten



- ▶ Geschwindigkeit der Prozessoren vs. Speicher
- ▶ Kosten für den Speicherplatz
- ▶ permanente Speicherung
 - ▶ magnetisch
 - ▶ optisch
 - ▶ mechanisch



- ▶ schnelle vs. langsame Speichertechnologie
schnell : hohe Kosten/Byte geringe Kapazität
langsam : geringe –"– hohe –"–
- ▶ wachsender Abstand zwischen CPU und Speichergeschwindigkeit
 - ▶ Prozessor läuft mit mehreren GHz Takt
 - ▶ Register können mithalten, aber nur einige KByte Kapazität
 - ▶ DRAM braucht 50 ... 100 ns für Zugriff: 100 × langsamer
 - ▶ Festplatte braucht 2,5 ... 10 ms für Zugriff: 1 000 000 × langsamer
- ▶ Lokalität der Programme wichtig
 - ▶ kleiner Adressraum im Programmkontext
 - ▶ aufeinanderfolgende Speicherzugriffe sind meistens „lokal“
 - ▶ gut geschriebene Programme haben meist eine gute Lokalität

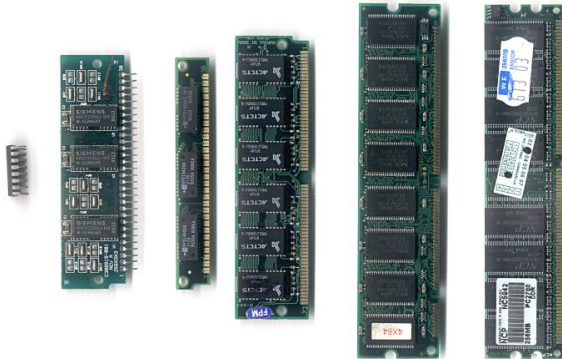
⇒ Motivation für spezielle Organisation von Speichersystemen = **Speicherhierarchie**

- ▶ Register $\leftrightarrow$ Memory
 - ▶ Compiler
 - ▶ Assembler-Programmierer
- ▶ Cache $\leftrightarrow$ Memory
 - ▶ Hardware
- ▶ Memory $\leftrightarrow$ Disk
 - ▶ Hardware und Betriebssystem: Paging, Segmentation
 - ▶ Programmierer und –"– : Dateien

- ▶ Register im Prozessor integriert
 - ▶ Program-Counter und Datenregister für Programmierer sichtbar
 - ▶ weitere Register für Systemprogrammierung
 - ▶ zusätzliche unsichtbare Register im Steuerwerk
- ▶ Flipflops oder Registerbank mit 6 Trans.-Speicherzellen
 - ▶ Lesen und Schreiben in jedem Takt möglich
 - ▶ ggf. mehrere parallele Lesezugriffe in jedem Takt
 - ▶ Zugriffszeiten ca. 100 ps
- ▶ typ. Größe einige KByte, z.B. 16 Register à 64-bit (x86-64)

L1-L4: Halbleiterspeicher RAM

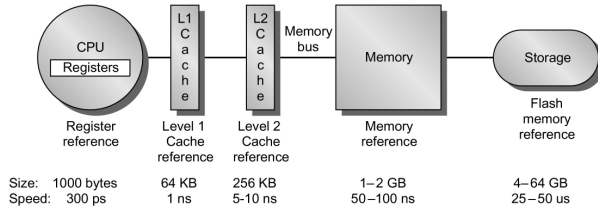
- ▶ „*Random-Access Memory*“ (RAM) aufgebaut aus Mikrochips
- ▶ Grundspeichereinheit ist eine Zelle (ein Bit pro Zelle)
- ▶ SRAM (6T-Zelle) oder DRAM (1T-Zelle) Technologie
- ▶ mehrere RAM Chips bilden einen Speicher



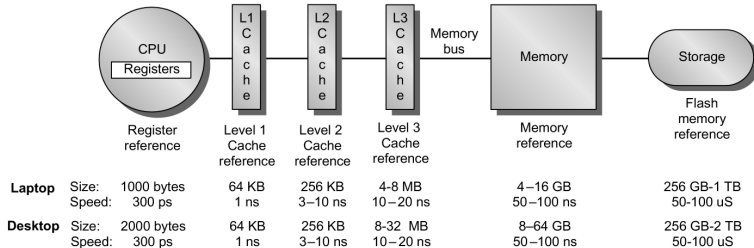
- ▶ dominierende Technologie für nichtflüchtigen Speicher
- ▶ hohe Speicherkapazität, derzeit bis 24 TB
 - ▶ Daten bleiben beim Abschalten erhalten
 - ▶ aber langsamer Zugriff
 - ▶ besondere Algorithmen, um langsamen Zugriff zu verbergen
- ▶ Einsatz als Speicher für dauerhafte Daten
- ▶ Einsatz als erweiterter Hauptspeicher („*virtual memory*“)
- ▶ FLASH/SSD zunehmend als Ersatz für Festplatten
 - ▶ Halbleiterspeicher mit sehr effizienten multibit-Zellen
 - ▶ Verwaltung wie Festplatten
 - ▶ signifikant schnellere Zugriffszeiten (Faktor > 20)

- ▶ enorme Speicherkapazität
- ▶ langsame Zugriffszeiten
- ▶ Archivspeicher und Backup für (viele) Festplatten
 - ▶ Magnetbänder
 - ▶ RAID-Verbund aus mehreren Festplatten
 - ▶ optische Datenspeicher: CD-ROM, DVD-ROM, BlueRay
- ▶ WWW und Internet-Services, Cloud-Services
 - ▶ Cloud-Farms ggf. ähnlich schnell wie L5 Festplatten
Netzwerk schneller als der Zugriff auf eine lokale (Magnet-) Festplatte
- ▶ in dieser Vorlesung nicht behandelt

Speicherhierarchie: zwei Beispiele



(A) Memory hierarchy for a personal mobile device



(B) Memory hierarchy for a laptop or a desktop

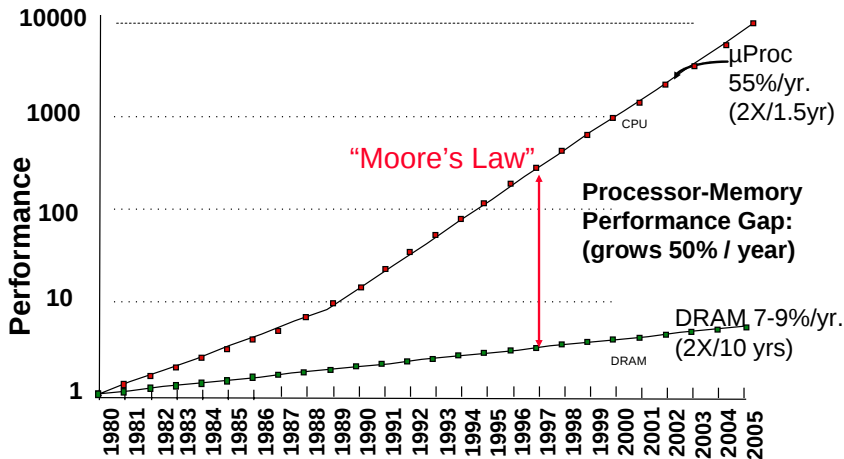
[HP17]

Eigenschaften der Speichertypen

► Speicher	Vorteile	Nachteile	
Register	sehr schnell	sehr teuer	
SRAM	schnell	teuer, große Chips	
DRAM	hohe Integration	Refresh nötig, langsam	
Platten	billig, Kapazität	sehr langsam, mechanisch	
► Beispiel	Hauptspeicher	Festplatte	SSD
Latenz	6 ns	2 ms	10/40 μ s
Bandbreite	64 GB/sec (pro Kanal, bis 12)	1,5 GB/sec	7/4 GB/sec (r/w)
Kosten/GB	1,7 €	1,5 ct. 1 TB: 15 €	4,1 ct. 43 €

- „Memory Wall“: DRAM zu langsam für CPU

Processor-DRAM Memory Gap (latency)

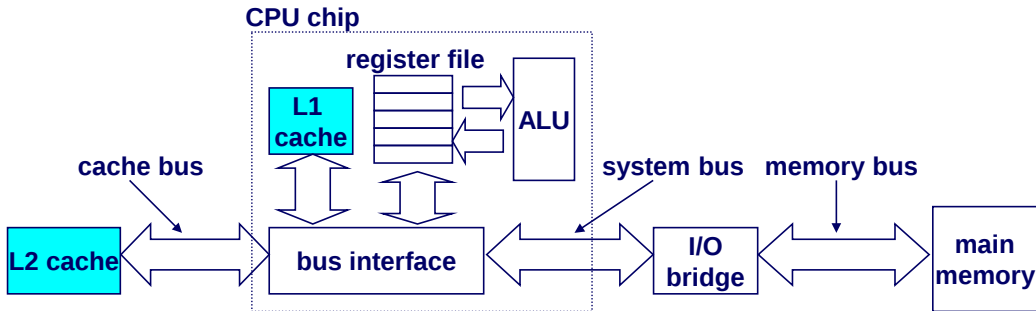


[PH22]

⇒ Cache als schneller Zwischenspeicher zum Hauptspeicher

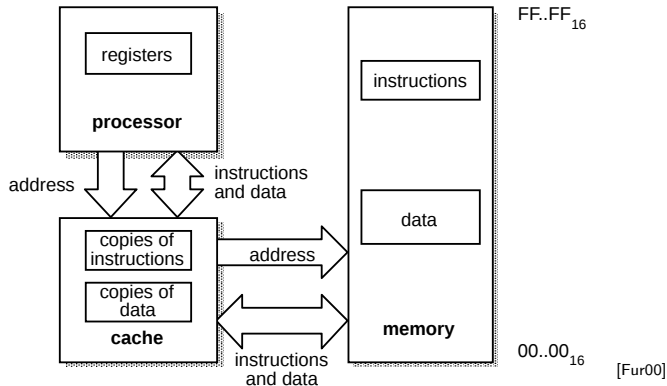
- ▶ technische Realisierung: SRAM
- ▶ transparenter Speicher
 - ▶ Cache ist für den Programmierer nicht sichtbar!
 - ▶ wird durch Hardware verwaltet
- ▶ ggf. getrennte Caches für Befehle und Daten
- ▶ enthält Hauptspeicherblöcke mit erhöhter Zugriffswahrscheinlichkeit
- ▶ nutzt die Lokalität von Speicherzugriffen durch ein laufendes Programm
 - ▶ ca. 80% der Zugriffe greifen auf 20% der Adressen zu
 - ▶ manchmal auch 90% / 10% oder noch besser
- ▶ de.wikipedia.org/wiki/Cache
en.wikipedia.org/wiki/CPU_cache
[en.wikipedia.org/wiki/Cache_\(computing\)](https://en.wikipedia.org/wiki/Cache_(computing))

- ▶ CPU referenziert Adresse
 - ▶ parallele Suche in L1 (level 1), L2 ... und Hauptspeicher
 - ▶ erfolgreiche Suche liefert Datum, Abbruch laufender Suchen

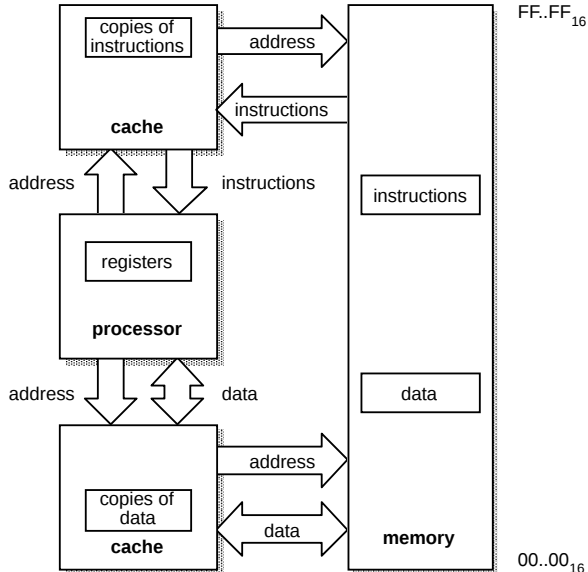


[BO15]

gemeinsamer Cache / „unified Cache“

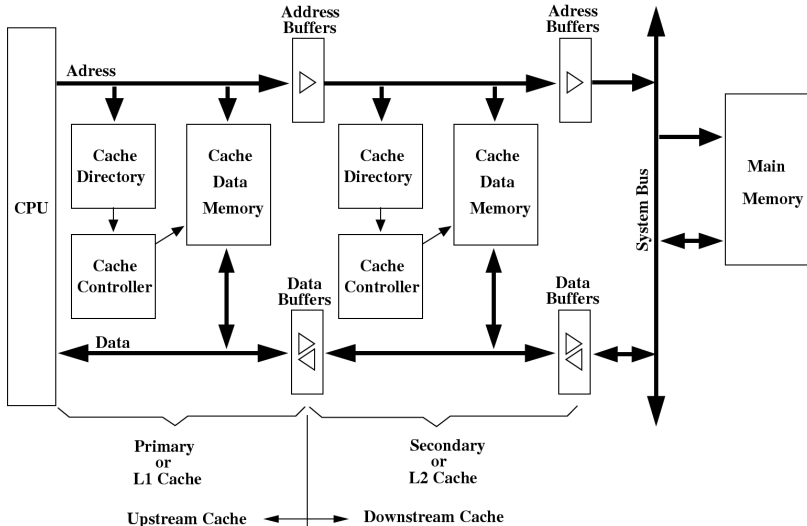


separate Instruction-/Data Caches



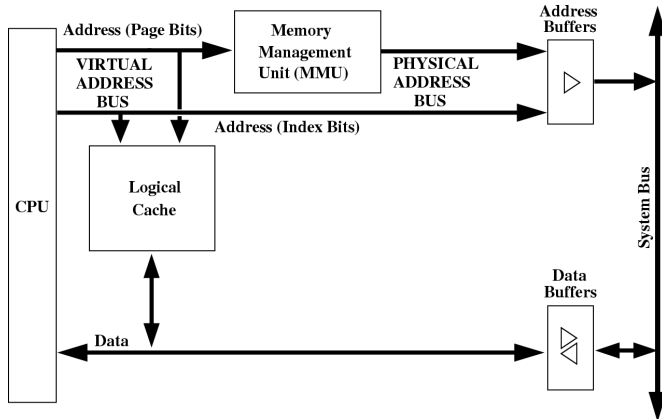
[Fur00]

► First- und Second-Level Cache



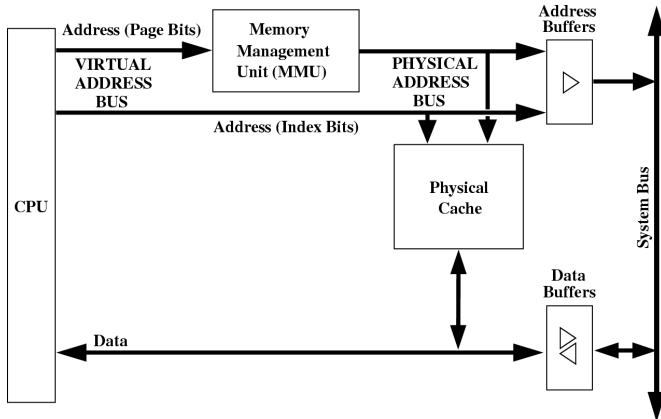
► Virtueller Cache

- + Adressumrechnung durch MMU oft nicht nötig
- Cache leeren bei Kontextwechseln



► Physischer Cache

- + Cache muss nie geleert werden
- Adressumrechnung durch MMU immer nötig



- ▶ typische Cache Organisation
 - ▶ First-Level Cache: getrennte Instruktions- und Daten-Caches
 - ▶ Second-Level Cache: gemeinsamer Cache je Prozessorkern
 - ▶ Third-Level Cache: gemeinsamer Cache für alle Prozessorkerne
- ▶ bei mehreren Prozessoren / Prozessorkernen
 - ⇒ Cache-Kohärenz wichtig
 - ▶ gemeinsam genutzte Daten konsistent halten (s.u.)

Cachestrategie: *Welche Daten sollen (präventiv) in den Cache?*

Diejenigen, die bald benötigt werden!

- ▶ *temporale Lokalität*: die Daten, die zuletzt häufig gebraucht wurden
- ▶ *räumliche Lokalität*: die Daten, die nahe den zuletzt gebrauchten liegen
- ▶ verschiedene Platzierungs-, Ersetzungs- und Rückschreibestrategien für den Cache

Cacheperformanz

► Begriffe

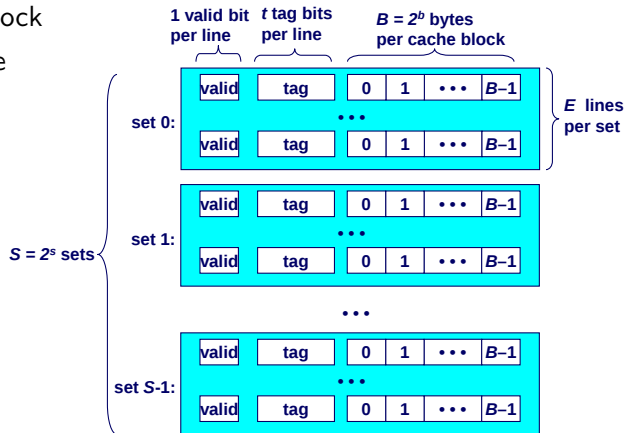
Treffer (Hit)		Zugriff auf Datum, ist bereits im Cache
Fehler (Miss)		–"– ist nicht –"–
Treffer-Rate	R_{Hit}	Wahrscheinlichkeit, Datum ist im Cache
Fehler-Rate	R_{Miss}	$1 - R_{Hit}$
Hit-Time	T_{Hit}	Zeit, bis Datum bei Treffer geliefert wird
Miss-Penalty	T_{Miss}	zusätzlich benötigte Zeit bei Fehler

► Mittlere Speicherzugriffszeit = $T_{Hit} + R_{Miss} \cdot T_{Miss}$

► Beispiel $T_{Hit} = 1$ Takt, $T_{Miss} = 20$ Takte, $R_{Miss} = 5\%$ $\Rightarrow$ Mittlere Speicherzugriffszeit = 2 Takte

Cache – Organisation

- ▶ Cache ist ein Array von Speicher-Bereichen: „sets“
- ▶ jeder Bereich enthält eine oder mehrere Zeilen: „Cache-Line“
- ▶ jede Zeile enthält einen Datenblock
- ▶ jeder Block enthält mehrere Byte

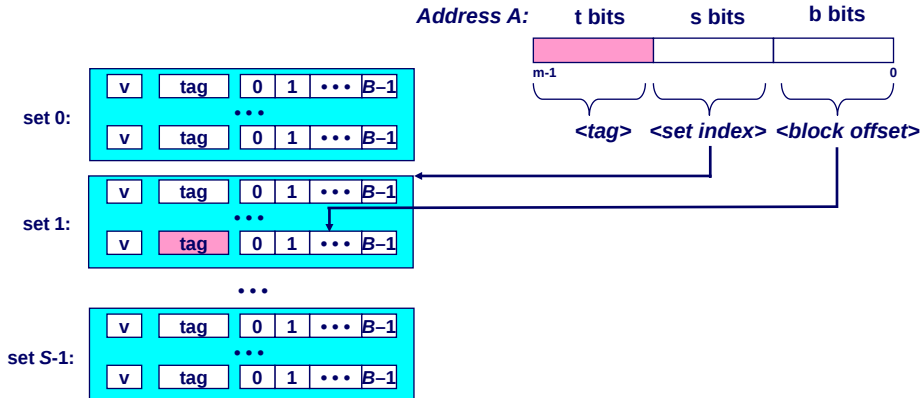


Cache size: $C = B \times E \times S$ data bytes

[BO15]

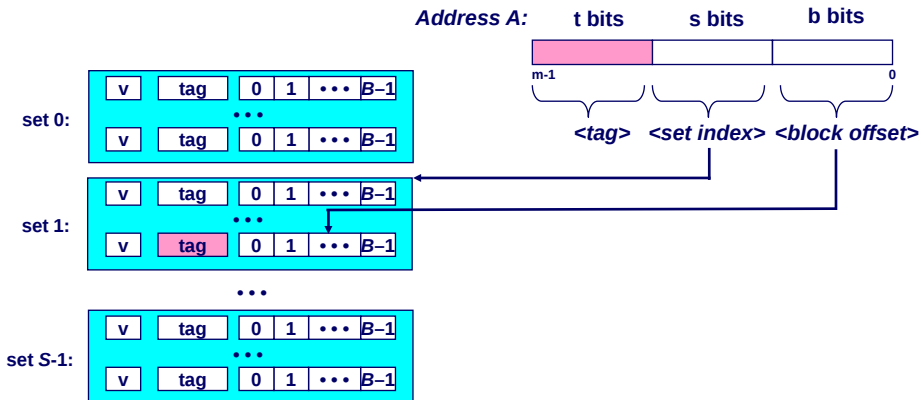
Cache – Adressierung

- ▶ Adressteil $\langle \text{set index} \rangle$ von A bestimmt Bereich („set“)
- ▶ Adresse A ist im Cache, wenn
 1. Cache-Zeile ist als gültig markiert („valid“)
 2. Adressteil $\langle \text{tag} \rangle$ von A = „tag“ Bits des Bereichs



Cache – Adressierung (cont.)

- ▶ Cache-Zeile („cache line“) enthält Datenbereich von 2^b Byte
- ▶ gesuchtes Wort mit Offset $\langle \text{block offset} \rangle$



► *Welchen Platz im Cache belegt ein Datum des Hauptspeichers?*

► drei Verfahren

direkt abgebildet / direct mapped jeder Speicheradresse ist genau eine Cache-Speicherzelle zugeordnet

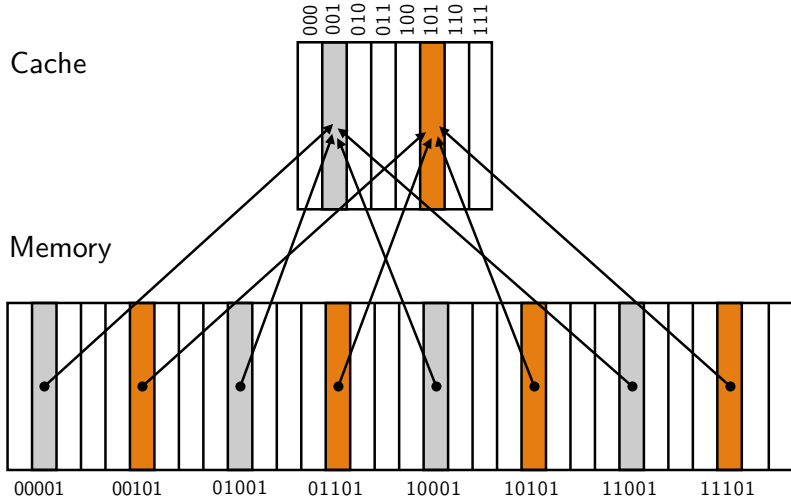
n-fach bereichsassoziativ / set associative

jeder Speicheradresse ist eine von n möglichen Cache-Speicherzellen zugeordnet, Cache-Line E in den Grafiken [BO15]

voll-assoziativ jeder Speicheradresse kann jede beliebige Cache-Speicherzelle zugeordnet werden

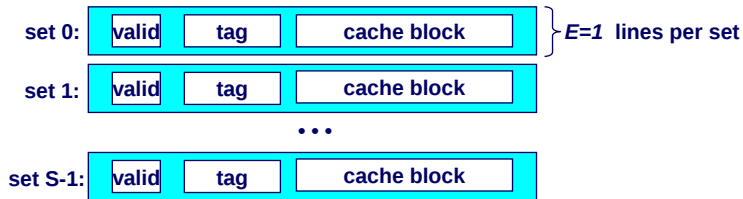
Cache: direkt abgebildet / „direct mapped“

- ▶ jeder Adresse ist genau eine Speicherzelle im Cache zugeordnet



Cache: direkt abgebildet / „direct mapped“ (cont.)

- ▶ genau 1 Zeile („cache line“) pro Bereich, S Bereiche (**S**ets)



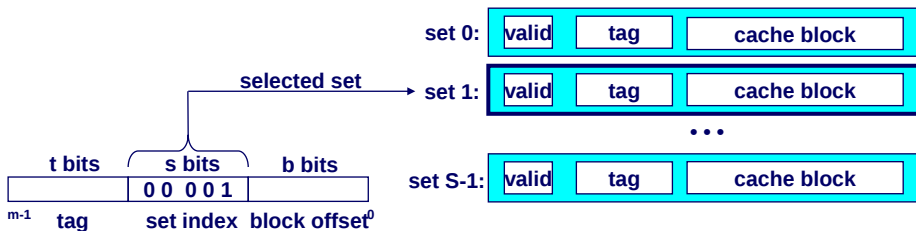
[BO15]

- + einfachste Cache-Art
- + große Caches möglich
- Effizienz, z.B. Zugriffe auf $A, A + n \cdot S \dots \Rightarrow$ „Cache Thrashing“
Beispiel (s.o.): Zugriff auf Adressen „00101“, „01101“, „10101“, „11101“

Cache: direkt abgebildet / „direct mapped“ (cont.)

Zugriff auf direkt abgebildete Caches

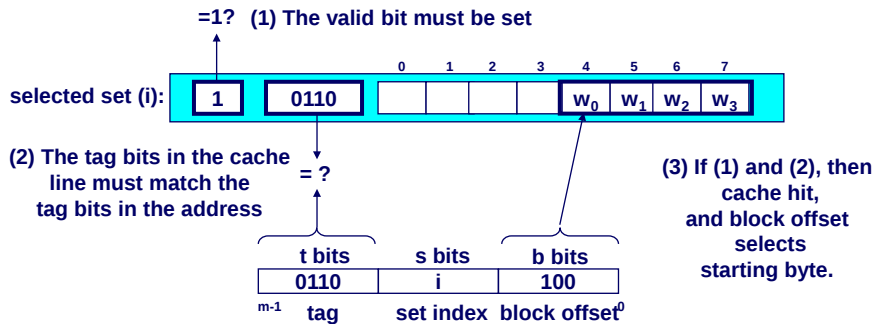
1. Bereichsauswahl durch Bits $\langle \text{set index} \rangle$



[BO15]

Cache: direkt abgebildet / „direct mapped“ (cont.)

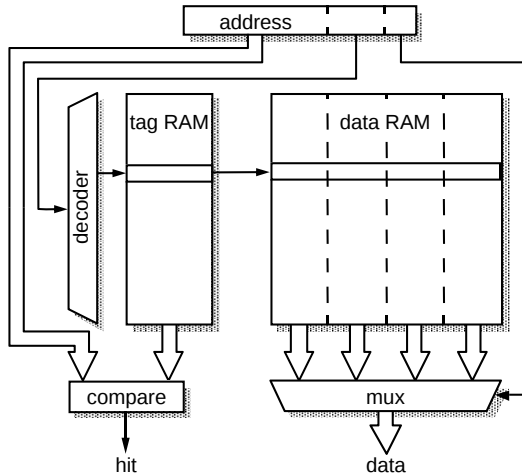
2. $\langle \text{valid} \rangle$: sind die Daten gültig?
3. „Line matching“: stimmt $\langle \text{tag} \rangle$ überein?
4. Wortselektion extrahiert Wort unter Offset $\langle \text{block offset} \rangle$



[BO15]

Cache: direkt abgebildet / „direct mapped“ (cont.)

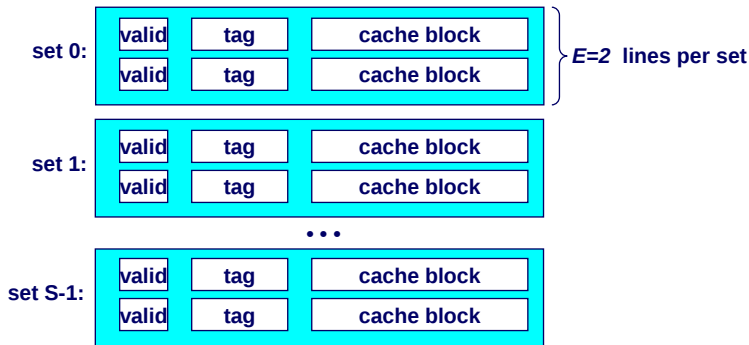
Prinzip



[Fur00]

Cache: bereichsassoziativ / „set associative“

- ▶ jeder Speicheradresse ist ein Bereich S mit mehreren Cachezeilen E zugeordnet
- ▶ n-fach assoziative Caches: $E=2, 4 \dots$ „2-way set associative cache“, „4-way ...“

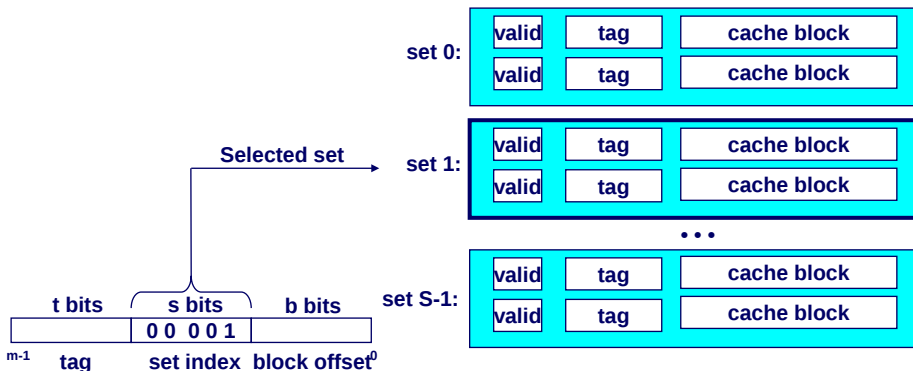


[BO15]

Cache: bereichsassoziativ / „set associative“ (cont.)

Zugriff auf n-fach assoziative Caches

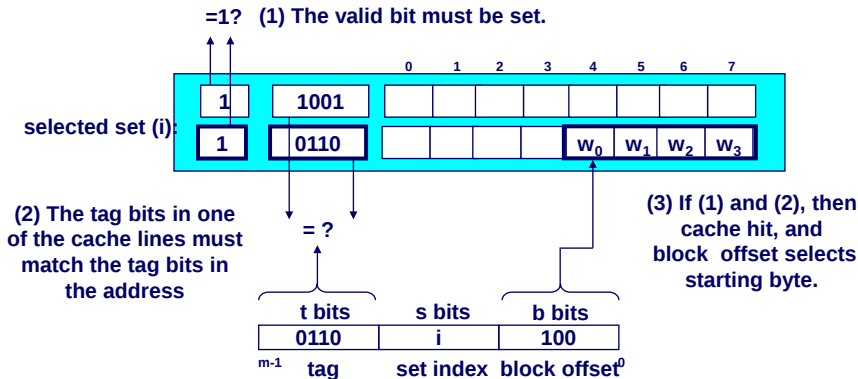
1. Bereichsauswahl durch Bits $\langle \text{set index} \rangle$



[BO15]

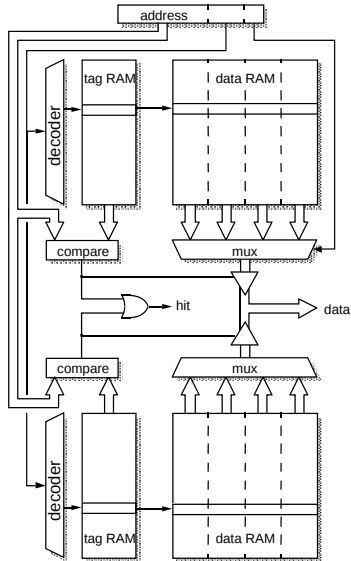
Cache: bereichsassoziativ / „set associative“ (cont.)

2. $\langle \text{valid} \rangle$: sind die Daten gültig?
3. „Line matching“: Cache-Zeile mit passendem $\langle \text{tag} \rangle$ finden?
dazu Vergleich aller „tags“ des Bereichs $\langle \text{set index} \rangle$
4. Wortselektion extrahiert Wort unter Offset $\langle \text{block offset} \rangle$



Cache: bereichsassoziativ / „set associative“ (cont.)

Prinzip

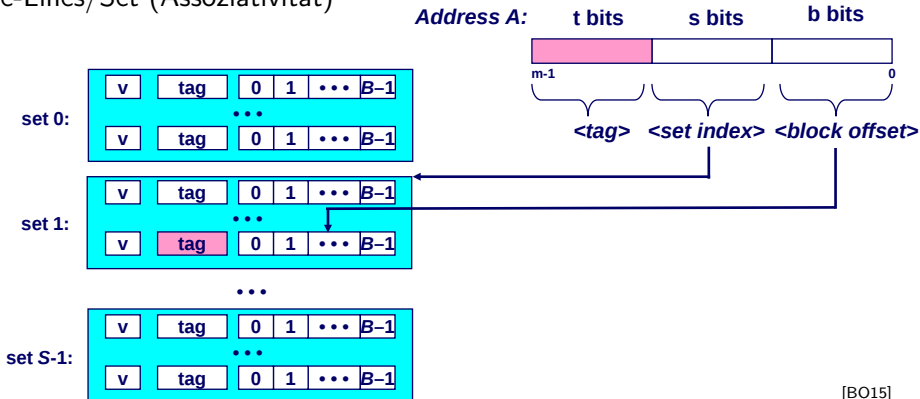


[Fur00]

- ▶ jeder Adresse des Speichers kann jede beliebige Cachezeile zugeordnet werden
- ▶ Spezialfall: nur ein Cachebereich S
- benötigt E -Vergleicher
- nur für sehr kleine Caches realisierbar

Parameter

- ▶ $S = \#$ Cache Bereiche Teil der Adresse
- ▶ $B = \text{Byte/Block}$ Teil der Adresse
- ▶ $E = \text{Cache-Lines/Set (Assoziativität)}$

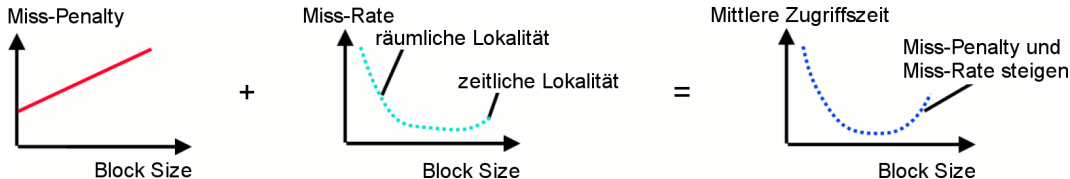


Vor- und Nachteile des Cache

- + nutzt räumliche Lokalität aus
Speicherzugriffe von Programmen (Daten und Instruktionen) liegen in ähnlichen/aufeinanderfolgenden Adressbereichen
- + breite externe Datenbusse, es werden ganze Bereiche übertragen
- + nutzt Burst-Adressierung des Speichers: Adresse nur für erstes Wort vorgeben, dann automatisches Inkrement
- + kürzere interne Cache-Adressen
- Hardwareaufwand und Kosten

Cache – Dimensionierung (cont.)

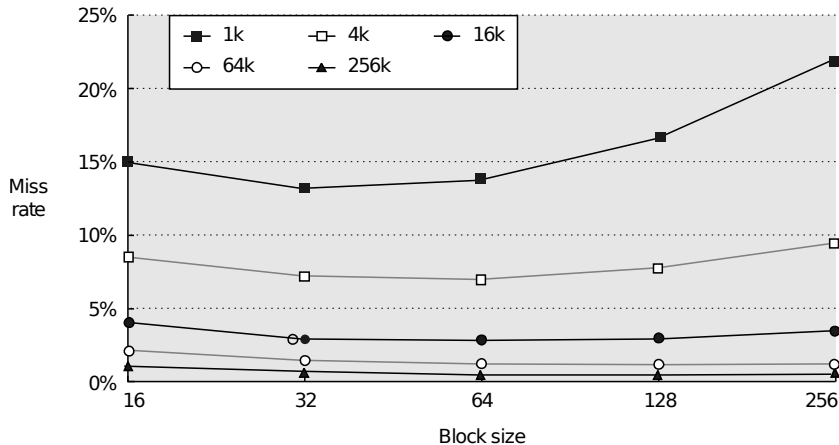
Cache- und Block-Dimensionierung



[PH22]

- ▶ Blockgröße klein, viele Blöcke
 - + kleinere Miss-Penalty
 - + temporale Lokalität
 - räumliche Lokalität
- ▶ Blockgröße groß, wenig Blöcke
 - größere Miss-Penalty
 - temporale Lokalität
 - + räumliche Lokalität

Cache – Dimensionierung (cont.)



[HP17]

- Block-Size: 32... 128 Byte
- L1-Cache: 4... 256 KiByte
- L2-Cache: 256... 4 096 KiByte

- ▶ **cold miss**
 - ▶ Cache ist (noch) leer
- ▶ **conflict miss**
 - ▶ wenn die Kapazität des Cache eigentlich ausreicht, aber unterschiedliche Daten in den selben Block abgebildet werden
 - ▶ Beispiel für „Trashing“ beim direct-mapped Cache mit $S=8$:
abwechselnder Zugriff auf Blöcke 0, 8, 0, 8, 0, 8 ... ist jedesmal ein Miss
- ▶ **capacity miss**
 - ▶ wenn die Menge der aktiven Blöcke („working set“) größer ist als die Cache Kapazität

Wenn der Cache gefüllt ist, welches Datum wird entfernt?

- ▶ zufällige Auswahl
- ▶ **LRU** (**L**east **R**ecently **U**sed): der „älteste“ nicht benutzte Cache Eintrag
 - ▶ echtes LRU als Warteschlange realisiert
 - ▶ Pseudo LRU mit baumartiger Verwaltungsstruktur:
Zugriff wird paarweise mit einem Bit markiert, die Paare wieder zusammengefasst usw.
- ▶ **LFU** (**L**east **F**requently **U**sed): der am seltensten benutzte Cache Eintrag
 - ▶ durch Zugriffszähler implementiert

Wann werden modifizierte Daten des Cache zurückgeschrieben?

- ▶ **Write-Through**: beim Schreiben werden Daten sofort im Cache und im Hauptspeicher modifiziert
 - + andere Bus-Master sehen immer den „richtigen“ Speicherinhalt: *Cache-Kohärenz*
 - Werte werden unnötig oft in Speicher zurückgeschrieben
- ▶ **Write-Back**: erst dann in den Speicher schreiben, wenn das Datum des Cache ersetzt werden würde
 - + häufig genutzte Werte (z.B. lokale Variablen) werden nur im Cache modifiziert
 - Cache-Kohärenz ist nicht gegeben
 - ⇒ spezielle Befehle für „Cache-Flush“
 - ⇒ „non-cacheable“ Speicherbereiche

- ▶ Daten zwischen Cache und Speicher konsistent halten
- ▶ notwendig wenn mehrere Einheiten (Bus-Master: Prozessor, DMA-Controller) auf den Speicher zugreifen können: wichtig für „*Symmetric Multiprocessing*“
- ▶ Harvard-Architektur hat getrennte Daten- und Instruktions-Speicher
 - ▶ Instruktionen sind read-only
 - ⇒ einfacherer Instruktions-Cache
 - ⇒ Cache-Kohärenz Problem betrifft D-Cache
- ▶ Cache-Kohärenz Protokolle und „*Snooping*“
 - ▶ alle Prozessoren($P_1, P_2 \dots$) überwachen alle Bus-Transaktionen
Cache „schnüffelt“ am Speicherbus
 - ▶ Prozessor P_2 greift auf Daten zu, die im Cache von P_1 liegen
 P_2 Schreibzugriff $\Rightarrow P_1$ Cache aktualisieren / ungültig machen
 P_2 Lesezugriff $\Rightarrow P_1$ Cache liefert Daten
 - ▶ Was ist mit gleichzeitige Zugriffen von P_1, P_2 ?

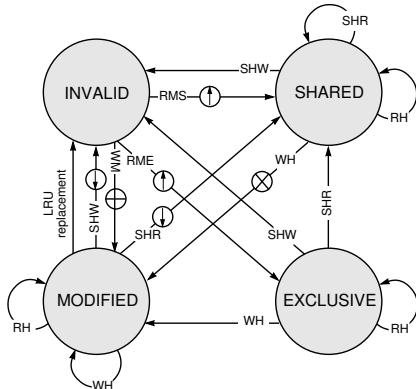
- ▶ viele verschiedene Protokolle: Hersteller- / Prozessor-spezifisch
 - ▶ SI („*Write Through*“)
 - ▶ MSI, MOSI,
 - ▶ MESI: *Modified, Exclusive, Shared, Invalid*
 - ▶ MOESI: *Modified (exclusive), Owned (Modified shared), Exclusive, Shared, Invalid*
 - ▶ ...

siehe z.B.: en.wikipedia.org/wiki/Cache_coherence

- ▶ Caches enthalten Wert, Tag und zwei Statusbits für die vier Protokollzustände
 - ▶ **Modified:** gültiger Wert, nur in diesem Cache, gegenüber Hauptspeicher-Wert verändert
 - ▶ **Exclusive:** gültiger Wert, nur in diesem Cache nicht verändert (unmodified)
 - ▶ **Shared:** gültiger Wert, in mehreren Caches vorhanden nicht verändert (unmodified)
 - ▶ **Invalid:** ungültiger Inhalt, Initialzustand
- ▶ alle Prozessoren überwachen alle Bus-Transaktionen
- ▶ bei Speicherzugriffen Aktualisierung des Status'
- ▶ Zugriffe auf „modified“-Werte werden erkannt:
 1. fremde Bus-Transaktion unterbrechen
 2. eigenen (=modified) Wert zurückschreiben
 3. Status auf shared ändern
 4. unterbrochene Bus-Transaktion neu starten

MESI Protokoll (cont.)

- ▶ erfordert spezielle Snoop-Logik im Prozessor
- ▶ garantiert Cache-Kohärenz
- ▶ gute Performanz, aber schlechte Skalierbarkeit
- ▶ Zustandsübergänge: MESI Protokoll



*PowerPC 604 RISC Microprocessor
User's Manual [Motorola / IBM]*

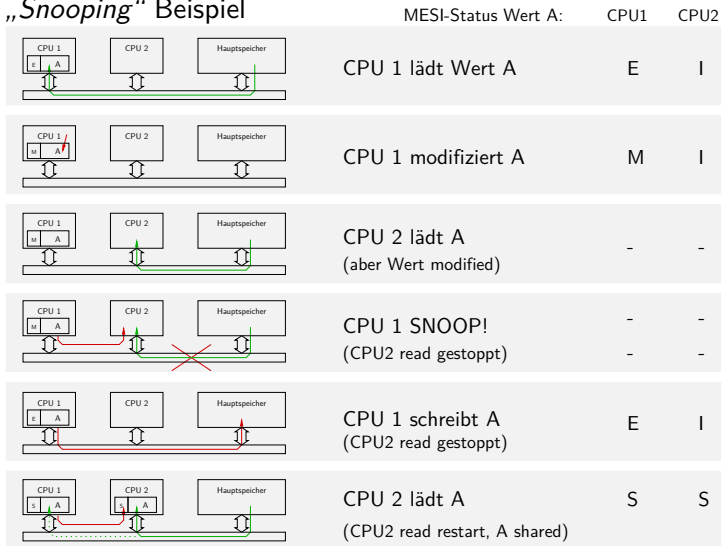
Bus Transactions

RH = Read hit
RMS = Read miss, shared
RME = Read miss, exclusive
WH = Write hit
WM = Write miss
SHR = Snoop hit on a read
SHW = Snoop hit on a write or
read-with-intent-to-modify

⬇ = Snoop push
⊗ = Invalidate transaction
⊕ = Read-with-intent-to-modify
⬆ = Read

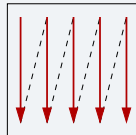
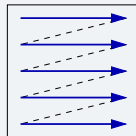
MESI Protokoll (cont.)

► „Snooping“ Beispiel



Cache Effekte bei Matrixzugriffen

```
public static double sumRowCol( double[][] matrix ) {  
    int rows = matrix.length;  
    int cols = matrix[0].length;  
    double sum = 0.0;  
    for( int r = 0; r < rows; r++ ) {  
        for( int c = 0; c < cols; c++ ) {  
            sum += matrix[r][c];  
        }  
    }  
    return sum;  
}
```



Matrix creation (5000×5000)

2105 msec.

Matrix row-col summation

75 msec.

Matrix col-row summation

383 msec.

⇒ 5 × langsamer

Sum = 600,8473695346258 / 600,8473695342268

⇒ andere Werte

Programmierer kann für maximale Cacheleistung optimieren

- ▷ Datenstrukturen werden fortlaufend alloziert
- 1. durch entsprechende Organisation der Datenstrukturen
- 2. durch Steuerung des Zugriffs auf die Daten
 - ▷ Geschachtelte Schleifenstruktur
 - ▷ Blockbildung ist eine übliche Technik

Cache-freundlicher Code

- ▷ Erreichen der optimalen Leistung ist plattformspezifisch
 - ▷ Cachegrößen, Zeilengrößen, Assoziativität etc.
 - ▷ Optimierung durch Compiler / manuell fehleranfällig
- ▷ generelle Empfehlungen
 - ▷ „working set“ klein $\Rightarrow$ zeitliche Lokalität
 - ▷ kleine Adressfortschaltungen („strides“) $\Rightarrow$ räumliche Lokalität

- [BO15] Randal E. Bryant and David R. O'Hallaron.
Computer systems – A programmers perspective.
Pearson Education Limited, Harlow, third, global ed. edition, 2015.
- [BVZ⁺08] Matthew J. Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin, and David I. August.
Revisiting the sequential programming model for the multicore era.
IEEE Micro Magazine, 28(1):12–20, 2008.
- [Fur00] Steve Furber.
ARM System-on-Chip Architecture.
Pearson Education Limited, Harlow, second edition, 2000.

- [HP17] John L. Hennessy and David A. Patterson.
Computer architecture – A quantitative approach.
Morgan Kaufmann Publishers Inc., San Mateo, CA, sixth edition, 2017.
- [Int] Santa Clara, CA.
- [PH04] David A. Patterson and John L. Hennessy.
Computer Organization and Design – The Hardware/Software Interface.
Morgan Kaufmann Publishers Inc., San Mateo, CA, third edition, 2004.
- [PH14] David A. Patterson and John L. Hennessy.
Computer Organization and Design – The Hardware/Software Interface.
Morgan Kaufmann Publishers Inc., San Mateo, CA, fifth edition, 2014.

- [PH20] David A. Patterson and John L. Hennessy.
Computer Organization and Design – The Hardware Software Interface – RISC-V Edition.
Morgan Kaufmann Publishers Inc., San Mateo, CA, second edition, 2020.
- [PH22] David A. Patterson and John L. Hennessy.
Rechnerorganisation und Rechnerentwurf – Die Hardware/Software-Schnittstelle – MIPS Edition.
De Gruyter Oldenbourg, Berlin, sixth edition, 2022.
- [TA14] Andrew S. Tanenbaum and Todd Austin.
Rechnerarchitektur – Von der digitalen Logik zum Parallelrechner.
Pearson Deutschland GmbH, Hallbergmoos, sixth edition, 2014.