64-040 Modul IP7: Rechnerstrukturen

15 Performance
Befehlspipeline und Parallelverarbeitung

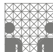
Norman Hendrich

Universität Hamburg
MIN Fakultät, Department Informatik
Vogt-Kölln-Str. 30, D-22527 Hamburg
hendrich@informatik.uni-hamburg.de

WS 2013/2014

Inhalt

1. Pipelining
   - Motivation und Konzept
   - Befehlspipeline
   - Pipeline-Hazards
2. Parallelrechner
   - Amdahl's Gesetz
   - Klassifikation
   - Multimedia-Befehlssätze
   - Symmetric Multiprocessing
   - Supercomputer
   - Literatur



Pipelining / Fließbandverarbeitung

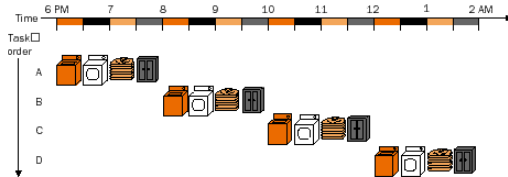
Allgemeine Idee:

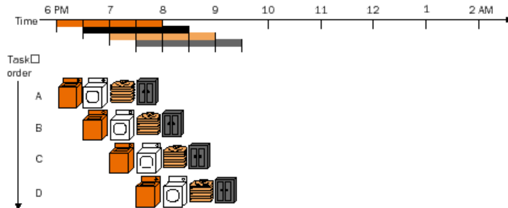
- Prozess in unabhängige Abschnitte aufteilen
- Objekte sequenziell durch diese Abschnitte laufen lassen
  - zu jedem Zeitpunkt werden zahlreiche Objekte bearbeitet
  - zu jedem Zeitpunkt sind alle Stationen ausgelastet
- Beispiele:
  - Autowaschanlage
  - Fabrik mit Fließbandverarbeitung
  - Halbleiterherstellung

⇒ Pipelines zur Beschleunigung arithmetischer Operationen

⇒ Befehlspipeline im Rechner

Pipelining: Grundidee





Pipelining im Digitalrechner

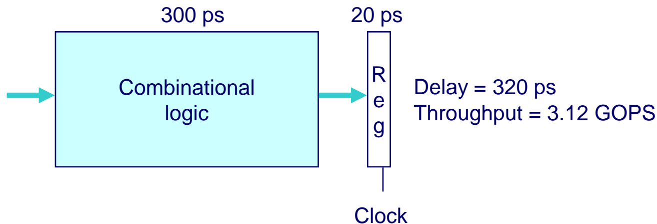
Arithmetische Pipelines

- 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

Befehlspipeline im Rechner

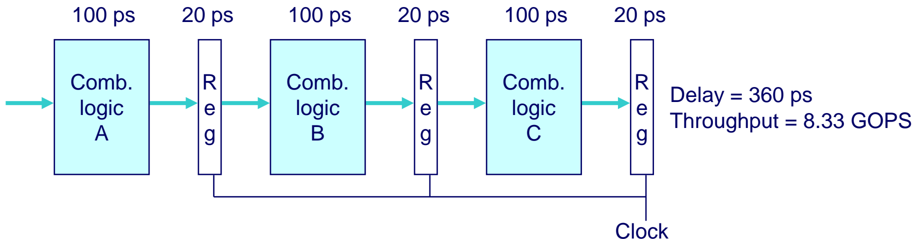
- Idee: die Phasen der von-Neumann Befehlsabarbeitung
  (Befehl holen, Befehl decodieren …) als Pipeline implementieren

Beispiel: Schaltnetz ohne Pipeline



- Verarbeitung im Schaltnetz erfordert 300 ps
- weitere 20 ps um das Resultat im Register zu speichern
- Zykluszeit: mindestens 320 ps
- eine Operation alle 320 ps, Durchsatz $1/320\,\text{ps} = 3.12\,\text{GOPS}$

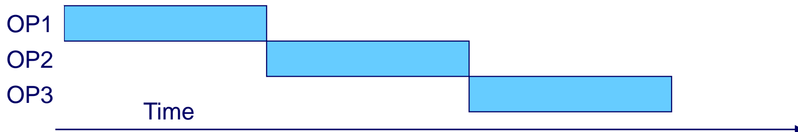
Beispiel: Version mit 3-stufiger Pipeline

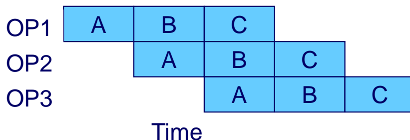


- Schaltnetz in 3 Blöcke zu je 100 ps aufgeteilt
- neue Operation starten, sobald vorheriger Abschnitt durchlaufen wurde ⇒ alle 120 ps neue Operation
- Durchsatz: $1/120\,\text{ps} = 8.33\,\text{GOPS}$
- aber höhere Latenz ⇒ 360 ps von Start bis Ende

Prinzip: 3-stufige Pipeline

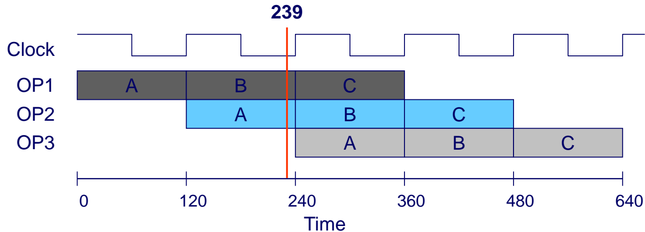
- ohne Pipeline

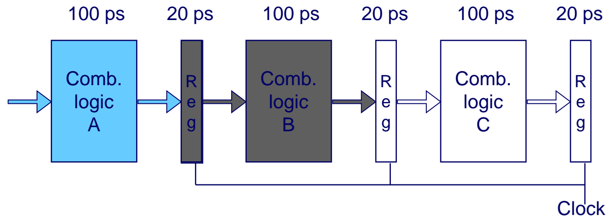




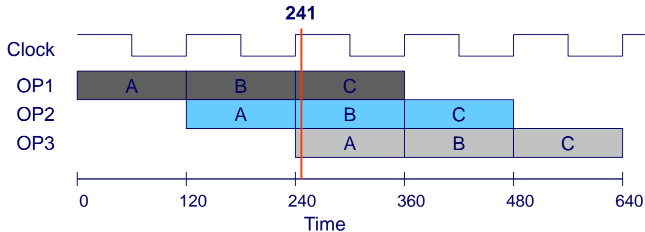
- 3-stufige Pipeline

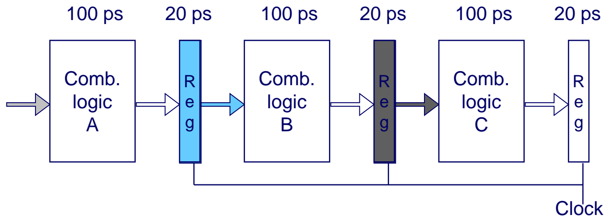
Timing: 3-stufige Pipeline



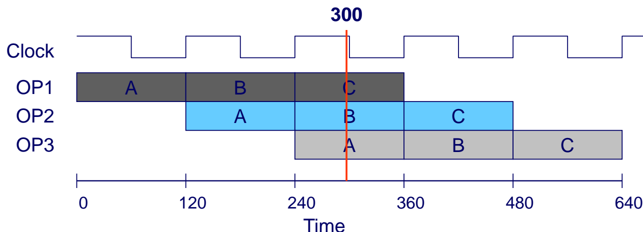


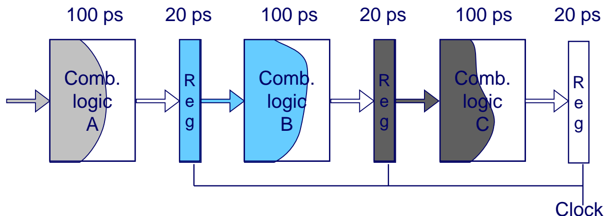
Timing: 3-stufige Pipeline



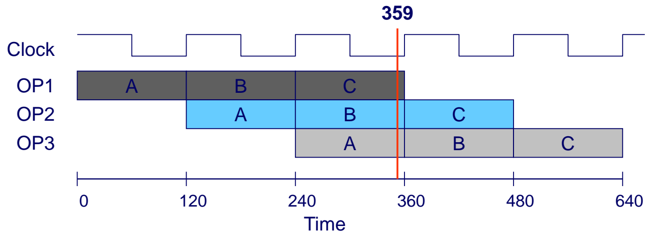


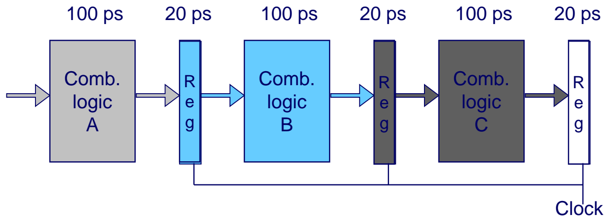
Timing: 3-stufige Pipeline



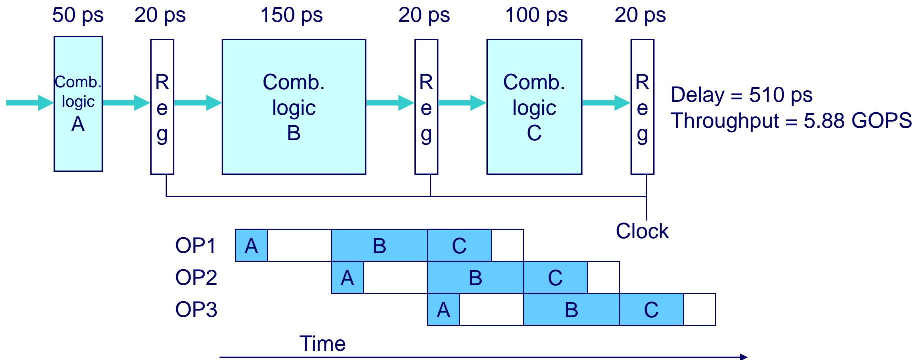


Timing: 3-stufige Pipeline



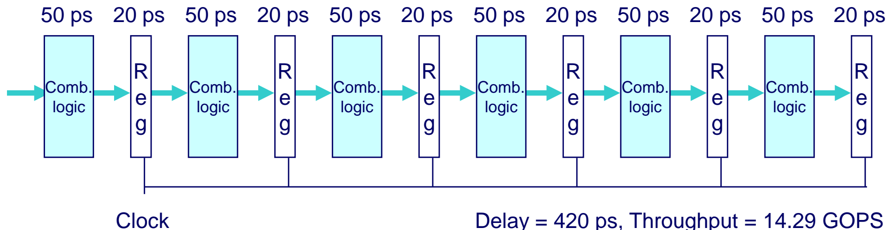


Limitierungen: nicht uniforme Verzögerungen



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

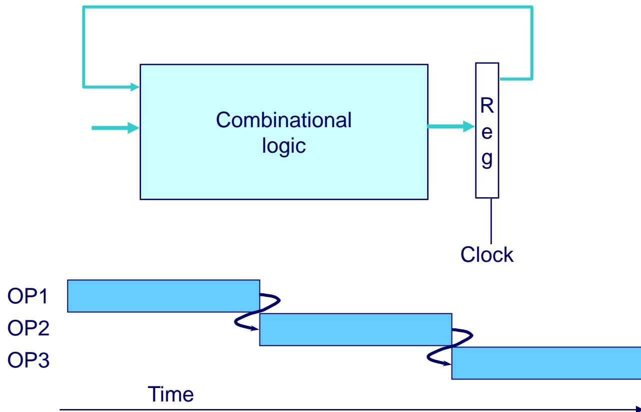
Limitierungen: Register-„Overhead“



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

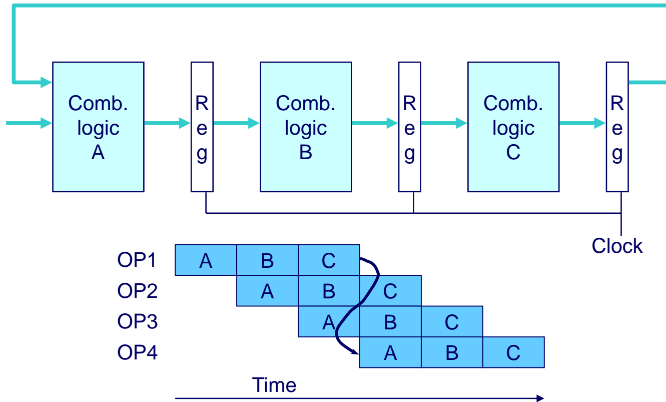
		Overhead	Taktperiode
1-Register:	6,25%	20 ps	320 ps
3-Register:	16,67%	20 ps	120 ps
6-Register:	28,57%	20 ps	70 ps

Limitierungen: Datenabhängigkeiten / „Data Hazards“



- jede Operation hängt vom Ergebnis der Vorhergehenden ab

Limitierungen: Datenabhängigkeiten / „Data Hazards“



⇒ Resultat-Feedback kommt zu spät für die nächste Operation

⇒ Pipelining ändert Verhalten des gesamten Systems

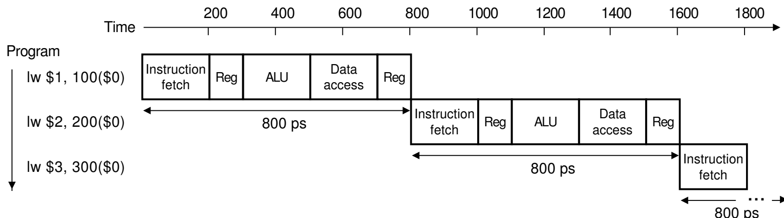
von-Neumann Befehlszyklus

typische Schritte (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	**Ex**ecute ALU führt Befehl aus
MEM	**Mem**ory access Speicherzugriff bei Load-/Store-Befehlen
WB	**W**rite **B**ack Ergebnisse in Register zurückschreiben

Serielle Befehlsausführung

- serielle Bearbeitung ohne Pipelining:

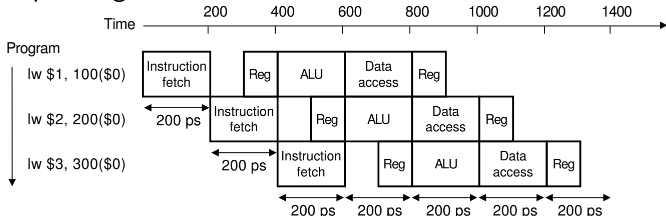


- einige Befehle benötigen nicht alle Schritte
  - nop: nur instruction-fetch :-)
  - jump: kein Speicher-/Registerzugriff

  ...
- Schritte können auch feiner unterteilt werden (mehr Stufen)

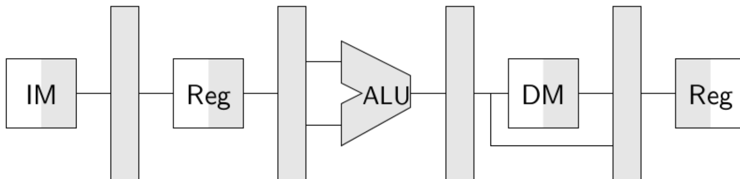
Befehlspipeline

- Pipelining für die einzelnen Schritte der Befehlsausführung



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

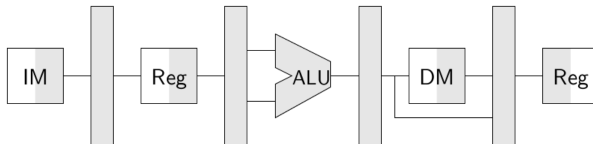
Klassische 5-stufige Pipeline



- 5 Stufen: Durchsatz ca. 3..5x besser als serielle Ausführung
- verschiedene Namen/Bezeichnungen für die gleiche Sache

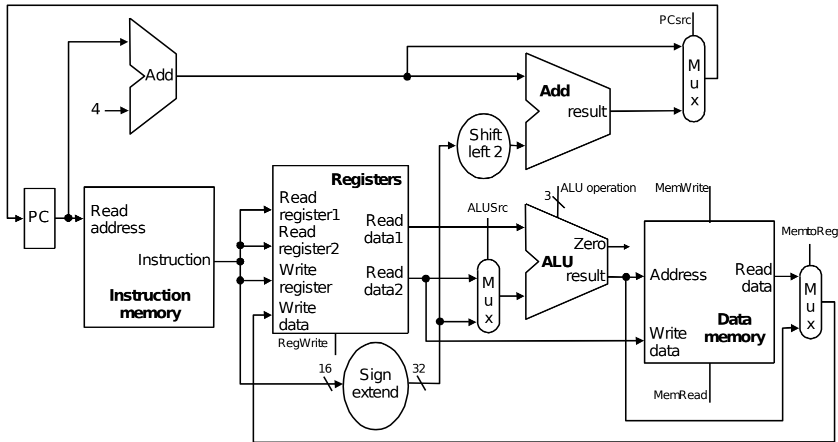
F	IF	IM	instruction fetch	instruction memory: Befehl holen
D	ID	Reg	instruction decode	Operanden aus Registern holen
E	EX	ALU	instruction execute	ALU führt Berechnung aus
M	MEM	DM	data memory	Daten laden/abspeichern
W	WB	Reg	write back	Ergebnis in Register schreiben

Klassische 5-stufige Pipeline



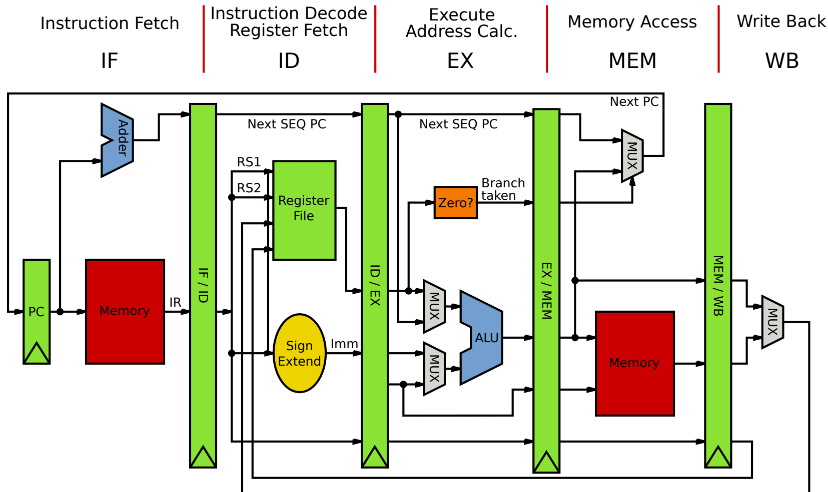
- guter Kompromiss aus Leistung und Hardwareaufwand
- Grundidee der ursprünglichen RISC-Architekturen
  - IBM-801, MIPS R-2000/R-3000 (1985), SPARC (1987)
  - Single-Chip Realisierung möglich (ab ca. 1985)
  - Befehlssätze auf diese Pipeline hin optimiert
- CISC-Architekturen heute ebenfalls mit Pipeline
  - Motorola 68020 (zweistufige Pipeline, 1984)
  - Intel 486 (1989), Pentium (1993), ...

MIPS: serielle Realisierung ohne Pipeline



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

MIPS: mit 5-stufiger Pipeline



MIPS: mit 5-stufiger Pipeline

- 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 (dekodierten) 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?!

Befehlspipeline: Bewertung

+ Schaltnetze in kleinere Blöcke aufgeteilt ⇒ höherer Takt
+ im Idealfall ein neuer Befehl pro Takt gestartet ⇒ höherer Durchsatz, bessere Performance
+ geringer Zusatzaufwand an Hardware
+ Pipelining ist für den Programmierer nicht direkt sichtbar
  - Warnung: Daten-/Kontrollabhängigkeiten (s.u.)

− Latenz wird nicht verbessert, bleibt bestenfalls gleich
− zusätzliche Zeiten, um Pipeline zu füllen bzw. zu leeren
− Takt der Pipeline limitiert durch langsamste Pipelinestufe sorgfältiger Entwurf der Hardware notwendig

Befehlspipeline: Analyse

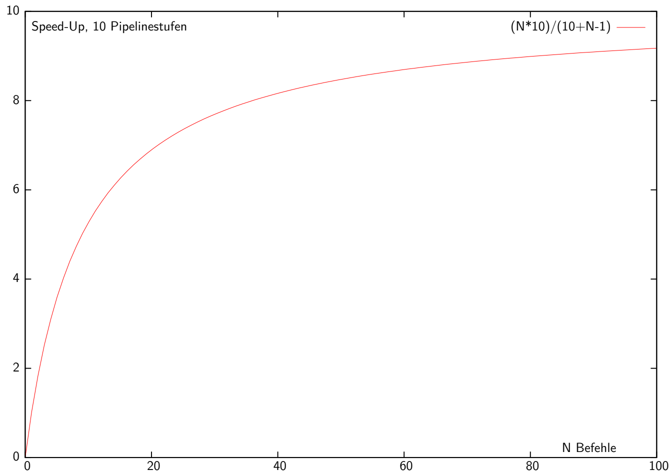
- N Instruktionen; K Pipelinestufen
- ohne Pipeline: $N \cdot K$ Taktzyklen
- mit Pipeline: $K + N - 1$ Taktzyklen

- „Speed-Up" $S = \frac{N \cdot K}{K+N-1}$, $\lim_{N\to\infty} S = K$

$\Rightarrow$ ein großer Speed-Up wird erreicht durch
  - große Pipelinetiefe: K
  - lange Instruktionssequenzen: N

  - wegen Daten- und Kontrollabhängigkeiten nicht erreichbar
  - außerdem: Register-Overhead nicht berücksichtigt

Befehlspipeline: theoretischer Speed-Up



Befehlspipeline: Dimensionierung

- größeres K wirkt sich direkt auf den Durchsatz aus
- weniger Logik zwischen den Registern, höhere Taktfrequenzen
- natürlich auch stark technologieabhängig (1985..2010)

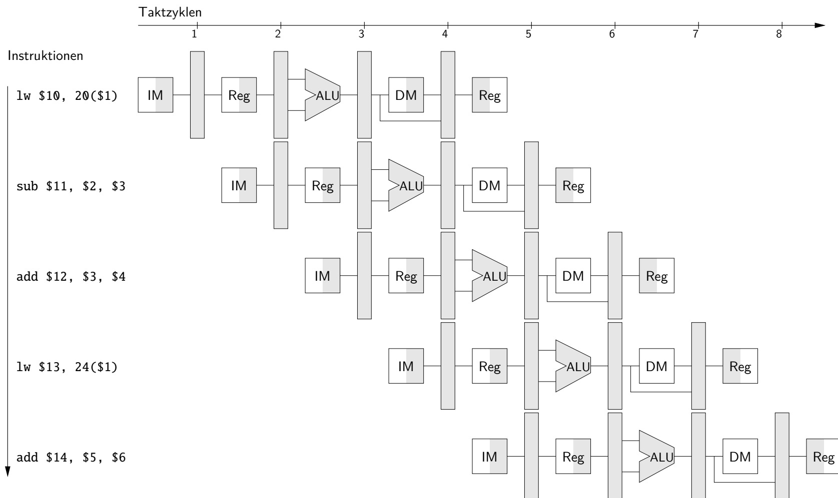
- Beispiele

CPU	Pipelinestufen	Taktfrequenz $[MHz]$
386	1	40
Pentium	5	300
Motorola G4	4	500
Motorola G4e	7	1000
Pentium II/III	12	1400
Athlon XP	10/15	2500
Athlon 64, Opteron	12/17	$\leq$ 3000
Pentium 4	20	$\leq$ 5000

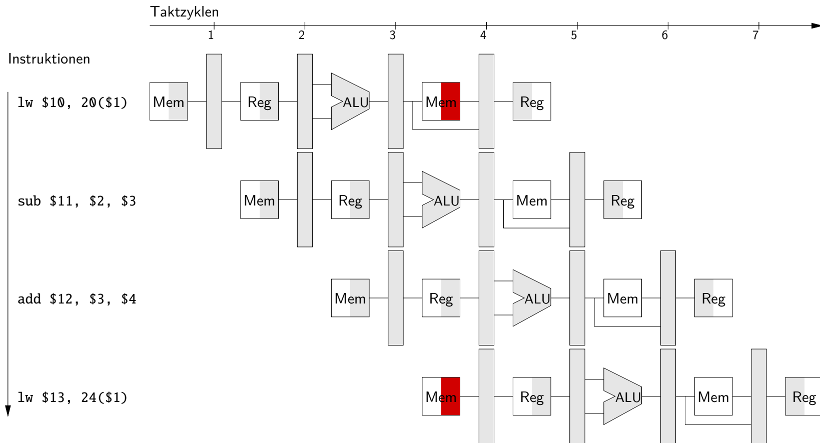
Befehlspipeline: Probleme

- Strukturkonflikt „structural hazard"
  - mehrere Stufen wollen gleichzeitig auf eine Resource zugreifen
  - Beispiel: gleichzeitiger Schreibzugriff auf den Speicher
  - teilweise durch zusätzliche Hardware zu vermeiden
- Datenkonflikt „data hazard"
  - Zugriff auf noch nicht berechnete / modifizierte Daten
  - RAW: read-after-write
  - WAR: write-after-read
  - WAW: write-after-write
  - forwarding: schnelle Zwischenergebnisse bereitstellen (rote Pfeile)
- Steuerkonflikt „control hazard"
  - Sprungbefehle, Exceptions, Interrupts

MIPS Pipeline: Idealfall ohne Konflikte

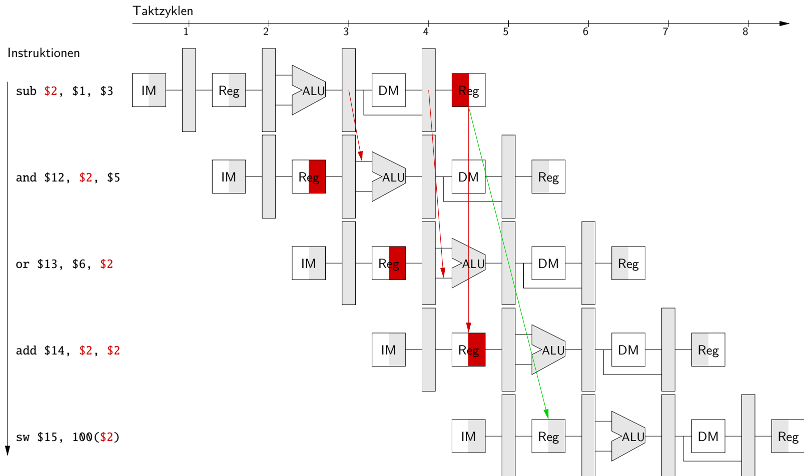


MIPS Pipeline: Strukturkonflikt



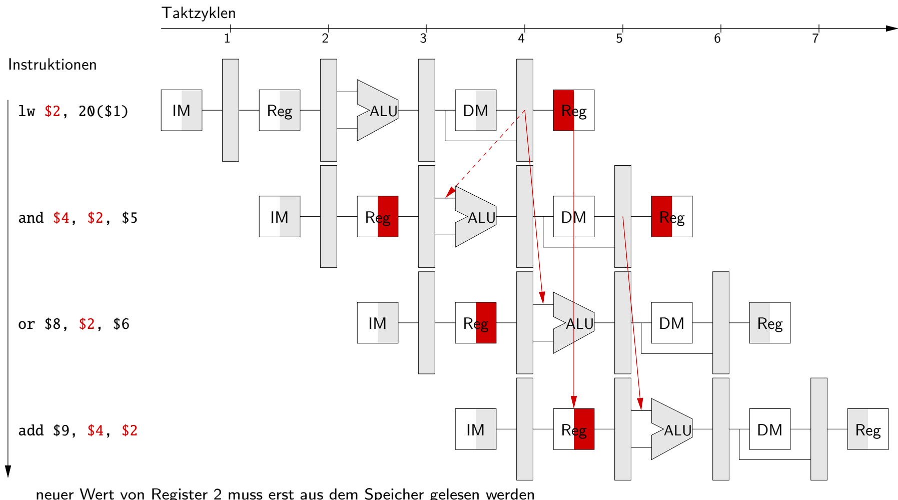
gleichzeitiges Laden aus dem Speicher, zwei verschiedene Adressen

MIPS Pipeline: Datenkonflikte und Forwarding



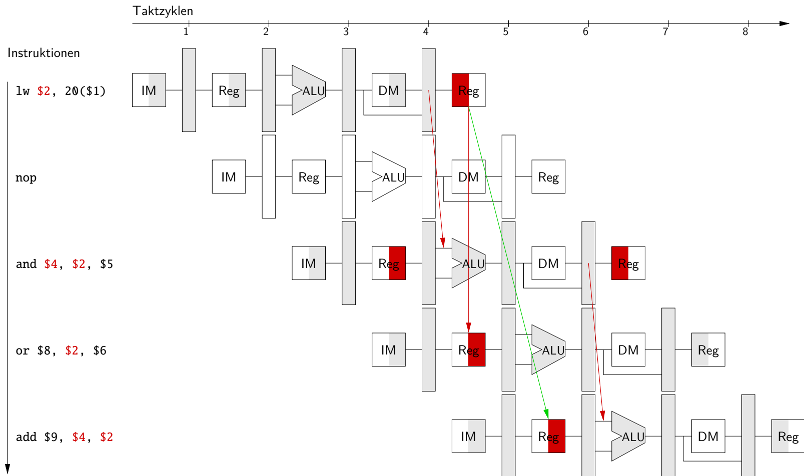
andere Befehle wollen Register 2 lesen, während dieses noch vom ersten Befehl berechnet wird.

MIPS Pipeline: Datenkonflikt mit Rückwärtsabhängigkeit



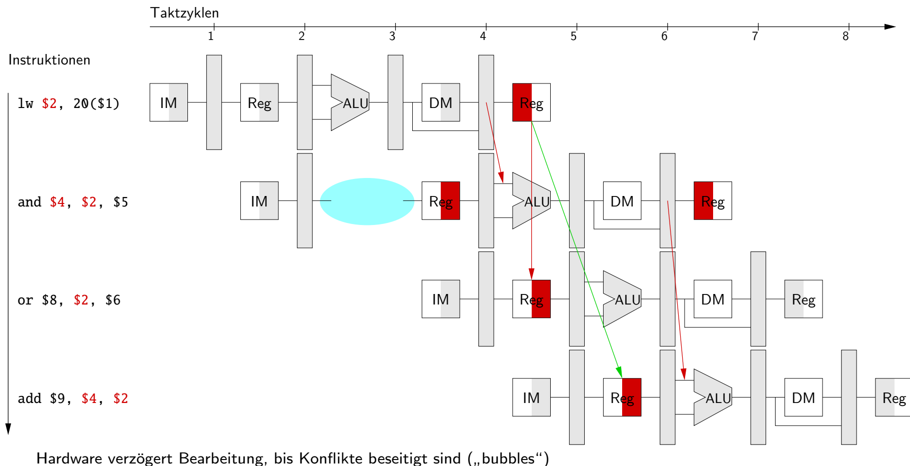
neuer Wert von Register 2 muss erst aus dem Speicher gelesen werden

MIPS Pipeline: Datenkonflikt mit Softwarelösung



Compiler kennt Hardware, und hat einen nop-Befehl eingefügt

MIPS Pipeline: Datenkonflikt mit „Interlocking“



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

Datenkonflikte

- treten auf, wenn nachfolgende Befehle zu früh (vor WB) auf Zwischenergebnisse zugreifen
- können in vielen Fällen durch zusätzliche Hardware vermieden werden („forwarding“ Einheiten)

- Rückwärtsabhängigkeiten sind nicht lösbar
  - Einfügen von nop-Operationen in Software
  - Compiler/Programmierer muss HW genau kennen

  - automatisches Stoppen der Pipeline („bubbles“), bis die benötigten Daten zur Verfügung stehen
  - komplexes Steuerwerk im Prozessor

„Data Forwarding“

Naive Pipeline

- Operanden während ID-Phase aus Registerbank gelesen
- Resultate werden erst in der WB-Phase geschrieben
- aber: Resultat ist nach der EX- oder MEM-Phase bekannt

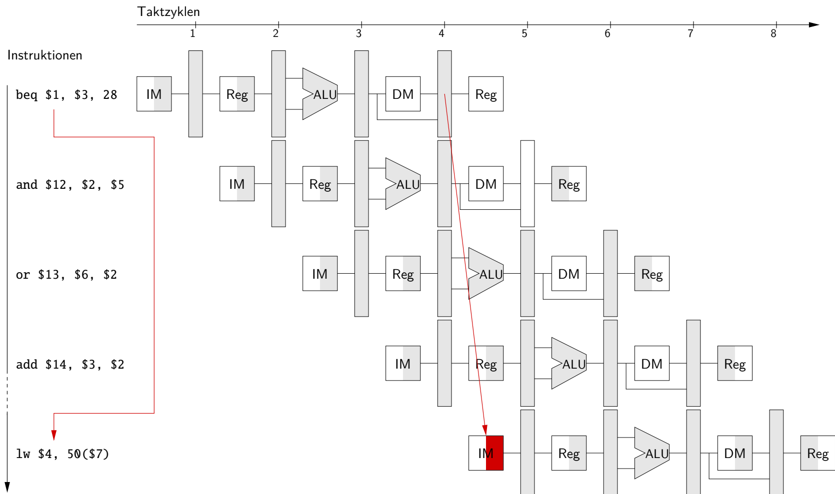
„Data Forwarding“-Trick

- Resultate direkt in die Dekodierstufe leiten
- und in Pipeline-Registern zwischenspeichern
- muss erst zum Ende der Dekodierphase verfügbar sein (5-stufige Pipeline ist de-facto halb-stufig aufgebaut)

Steuerkonflikte

- Sprungbefehle unterbrechen den sequenziellen Ablauf
- ebenso für `call` und `ret`
  - Problem: Instruktionen die auf (bedingte) Sprünge folgen, werden bereits in die Pipeline geschoben
  - Sprungadresse und Status (taken/not-taken) sind aber erst am Ende der EX-Phase gekannt
  - einige Befehle wurden bereits teilweise ausgeführt, und 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

Steuerkonflikte: bedingter Sprung



Steuerkonflikte: Lösungmöglichkeiten

- Interlocking: Pipeline prinzipiell bei Sprüngen leeren
  - ineffizient: ca 19 % aller Befehle sind Sprünge

- statische Sprungvorhersage
  - Annahme: nicht ausgeführter Sprung / „untaken branch“
  - Pipeline leeren, falls Sprung doch ausgeführt wird

  - Compiler kann Sprungwahrscheinlichkeit im Code annotieren (z.B. 1 Bit im Opcode für Sprungbefehle)
  - Compiler kann Code erzeugen, dass nur wenige Sprünge auch tatsächlich ausgeführt werden
  - vorherige Code-Analyse oder Profiling des Programms

Steuerkonflikte: Entscheidung verzögern

- Compiler versucht, die Befehle umzusortieren
  - nach jedem Sprungbefehl solche Befehle einfügen, die in beiden Fällen (taken/not-taken) ausgeführt werden

  - diese Befehle werden von der Pipeline also korrekt ausgeführt
  - in Hardware kostengünstig umzusetzen

  - aber: compilierte Programme sind auf eine bestimmte Pipeline optimiert (z.B. MIPS+SPARC: 1 branch delay slot), späterer Wechsel ist problematisch

Steuerkonflikte: Sprungvorhersage

- viele Sprungbefehle haben starken „bias“ (taken/not-taken)
  - Beispiel: viele Schleifen werden mehrfach durchlaufen
  - jeder Sprungebefehl an eindeutiger Adresse im Programm

- Hardware/Software-Mechanismus zur Sprungvorhersage
  - Compiler annotiert Programm mit Sprungwahrscheinlichkeit
  - Hardware speichert bisheriges Verhalten des Programms
  - entsprechende Steuerung der Pipeline
  - viele verschiedene Verfahren:
    History-Bit, 2-Bit Prädiktor, korrelationsbasierte Vorhersage,
    Branch History Table, Branch Target Cache...

Pipeline: Zusammenfassung

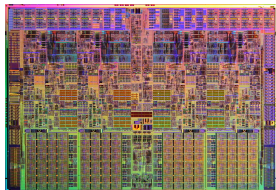
- von-Neumann Zyklus auf separate Phasen aufteilen
- überlappende Ausführung von mehreren Befehlen
  - einfachere Hardware für jede Phase: höherer Takt
  - mehrere Befehle in Bearbeitung: höherer Durchsatz
  - klassische RISC-Pipeline: fünf-Stufen I-D-E-M-W
  - mittlerweile sind 9..20 Stufen üblich

- 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 basiseren auf Pipelining (MIPS)
  - Compiler/Tools/Progammierer sollten CPU Pipeline kennen

Parallelrechner

- Motivation
- Amdahl's Gesetz
- Merkmale und Klassifikation
- Performance-Abschätzungen



Drei Beispiele:

- Befehlssätze für Multimedia (SIMD)
- Symmetric Multiprocessing und Cache-Kohärenz (SMP)
- Supercomputer (MIMD)

Motivation: ständig steigende Anforderungen

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

- Performance eines einzelnen Prozessors ist begrenzt
- also: Verteilen eines Programms auf mehrere Prozessoren

Vielfältige Möglichkeiten:
- wie viele und welche Prozessoren?
- Kommunikation zwischen den Prozessoren?
- Programmierung und Software/Tools?

Performance: Antwortzeit

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

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

- **Ausführungszeit** (reine CPU-Zeit)
  - user-time CPU-Zeit für Benutzerprogramm
  - system-time CPU-Zeit für Betriebssystem
  - Unix: `time make` `7.950u 2.390s 0:22.98 44.9%`

- **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 Performance verbessern?

Ausführungszeit := (Anzahl der Befehle) × (Zeit pro Befehl)

▶ weniger Befehle	besserer Compiler mächtigere Befehle (CISC)
▶ weniger Zeit pro Befehl	bessere Technologie Pipelining, Caches einfachere Befehle (RISC)
▶ parallele Ausführung	superskalar, SIMD, MIMD

Amdahl's Gesetz

Möglicher Speedup durch Beschleunigung einer Teilfunktion?

System 1: berechnet Programm P, darin Funktion X
mit Anteil $0 < f < 1$ der Gesamtzeit

System 2: Funktion X' ist schneller als X mit speedup s_X

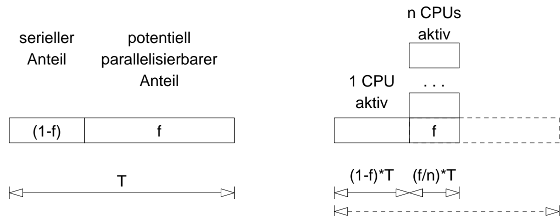
- Amdahl's Gesetz:

$$s_{\text{gesamt}} = \frac{1}{(1-f) + f/s_X}$$

(Gene Amdahl, Architekt der IBM S/360, 1967)

Amdahl's Gesetz

Nur ein Teil des Gesamtproblems wird beschleunigt:



$$s_{\text{gesamt}} = \frac{1}{(1-f) + f/s_X}$$

- Optimierung lohnt nur für relevante Operationen
- gilt entsprechend auch für Projektplanung, Verkehr, ...

Amdahl's Gesetz: Beispiele

$$s_X = 10, f = 0.1 \qquad s_{\text{gesamt}} = 1/(0.9 + 0.01) = 1.09$$
$$s_X = 2, f = 0.5 \qquad s_{\text{gesamt}} = 1/(0.5 + 0.25) = 1.33$$
$$s_X = 2, f = 0.9 \qquad s_{\text{gesamt}} = 1/(0.1 + 0.45) = 1.82$$
$$s_X = 1.1, f = 0.98 \qquad s_{\text{gesamt}} = 1/(0.02 + 0.89) = 1.10$$

- Optimierung bringt nichts, wenn der nicht beschleunigte „serielle" Anteil $(1 - f)$ eines Programms überwiegt
- die erreichbare Parallelität in Hochsprachen-Programmen (z.B. Java) ist gering, typisch z.B. $s_{\text{gesamt}} \leq 4$

Benchmarking

Rechner werden für ganzunterschiedliche Aufgaben genutzt:
Office, 3D-Gaming, Softwareentwicklung, usw.

⇒ Auswahl von „typischen" Standardprogrammen: Benchmarks

- skriptgesteuerter Ablauf dieser Programme
- Auswertung der Antwortzeit oder des Durchsatzes
- Normierung und Mittelung der Werte ergibt Leistungsindex
- ermöglicht Vergleich verschiedener Systeme

- Drystone/Whetstone/Coremark (synth. Benchmarks)
- BABCO Sysmark (PC Office)
- SPECint/SPECfp (wiss. Applikationen)
- 3DMark, Furmark, usw. (Grafik)

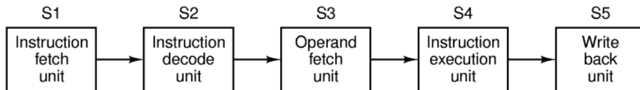
www.bapco.com, www.spec.org, usw.

Benchmarking: für den Systementwurf

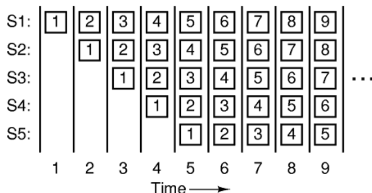
- relevante Benchmark-Programme auswählen
- mit Analysefunktionen erweitern, z.B. Instruction Tracing
- verschiedene Versionen des System-Designs testen
  - Compileroptimierungen, falls Benchmark-Quelltext verfügbar
  - verbesserte Befehlssätze, optimierte Befehle
  - optimierte Hardwarestruktur, Pipeline-Details (s.u.)
  - Cache-Organisation und Details der Speicherhierarche
  - Bus-System und Systemanbindung

- diese Analysen erfolgen, lange bevor das System fertig ist
  - spez. Tools für statische Analysen und Abschätzungen
  - Simulation oder Emulation des Gesamtsystems
  - deutlich langsamer als die spätere echte Hardware
  - Intel & Co. nutzen ganze Rechenzentren für diese Aufgaben

Befehls-Pipeline



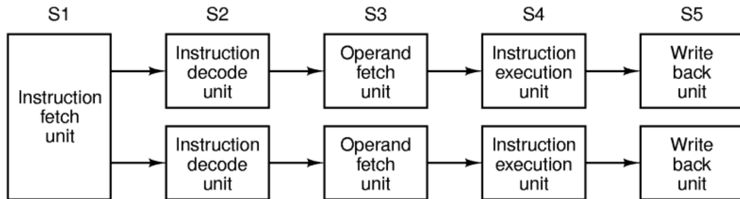
(a)



(b)

- Aufteilen eines Befehls in kleinere (=schnellere) Schritte
- überlappte Ausführung für höheren Durchsatz

Parallele Pipelines

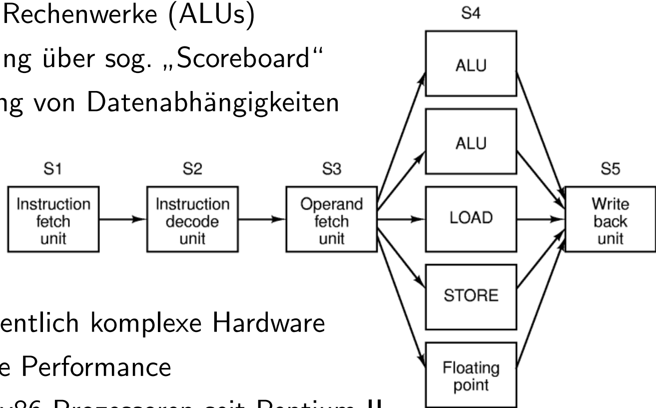


- parallele („superskalare") Ausführung
- im Bild jeweils zwei Operationen pro Pipelinestufe

- komplexe Hardware (Daten- und Kontrollabhängigkeiten)
- Beispiel: Pentium-I

Superskalarer Prozessor

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



- außerordentlich komplexe Hardware
- aber gute Performance
- fast alle x86-Prozessoren seit Pentium-II

Parallelrechner mit mehreren Prozessoren

- Taktfrequenzen $> 10\,\text{GHz}$ nicht sinnvoll realisierbar
  - hoher Takt nur bei einfacher Hardware möglich
  - Stromverbrauch bei CMOS proportional zum Takt

$\Rightarrow$ mehrere Prozessoren

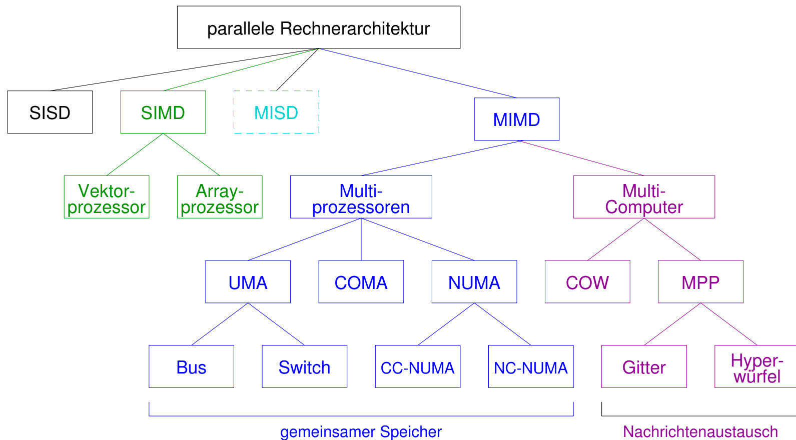
- Datenaustausch: shared-memory oder Verbindungsnetzwerk

- aber Overhead durch Kommunikation
- Programmierung ist ungelöstes Problem
- aktueller Kompromiss: bus-basierte „SMPs“ mit 2..16 CPUs

Flynn-Klassifikation

SISD	„single instruction, single data“ jeder klassische von-Neumann Rechner (z.B. PC)
SIMD	„single instruction, multiple data“ Vektorrecher/Feldrechner z.B. Connection-Machine 2: 65536 Prozessoren z.B. x86 MMX/SSE: 2..8 fach parallel
MIMD	„multiple instruction, multiple data“ Multiprozessormaschinen z.B. Quad-Core PC, Compute-Cluster
MISD	„multiple instruction, single data“ :-)

Detaillierte Klassifikation



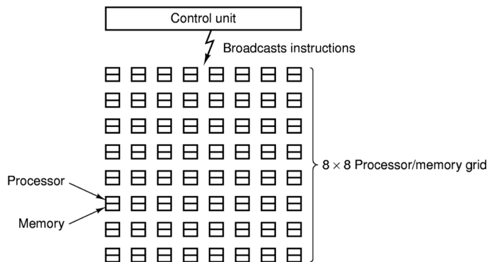
(Tanenbaum, Structured Computer Organization)

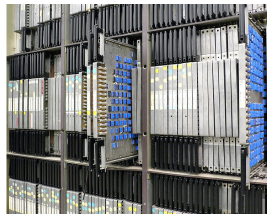
SIMD: Vektorrechner: Cray-1 (1976)

- Prinzip: Anwendung eines Rechen-Befehls auf alle Elemente von Vektoren/Matrizen
- Adressberechnung mit „Stride“
- „Chaining“ von Vektorbefehlen

- schnelle skalare Befehle
- ECL-Technologie, Freon-Kühlung
- 1662 Platinen (Module), über Kabel verbunden
- 80 MHz Takt, 136 MFLOPS

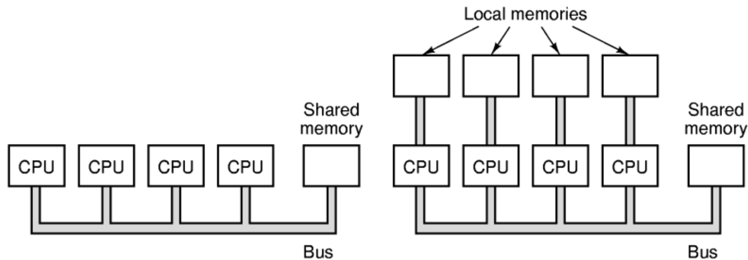
SIMD: Feldrechner Illiac-IV (1964..1976)





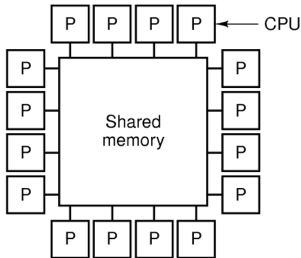
- ein zentraler Steuerprozessor
- 64 Prozessoren/ALUs und Speicher, 8x8 Matrix
- Befehl wird parallel auf allen Rechenwerken ausgeführt
- aufwendige und teure Programmierung
- oft schlechte Auslastung (Parallelität Algorithmus vs. #CPUs)

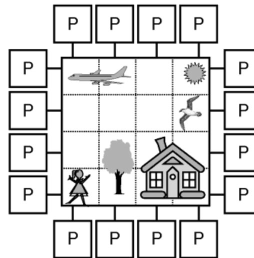
MIMD: Konzept



- mehrere Prozessoren, über Bus/Netzwerk verbunden
- gemeinsamer („shared“) oder lokaler Speicher
- unabhängige oder parallele Programme / Multithreading
- sehr flexibel, zunehmender Markterfolg

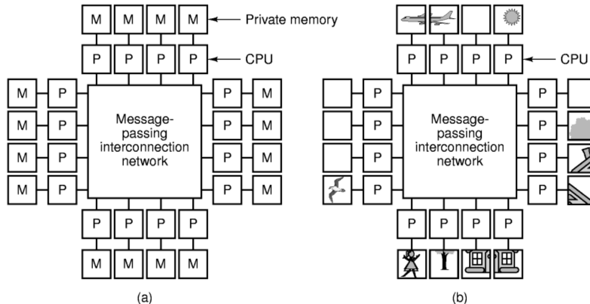
MIMD: Shared-Memory





- mehrere CPUs, aber gemeinsamer Speicher
- jede CPU bearbeitet nur eine Teilaufgabe
- CPUs kommunizieren über den gemeinsamen Speicher
- Zuordnung von Teilaufgaben/Speicherbereichen zu CPUs

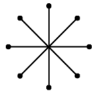
MIMD: Message-Passing

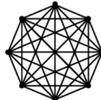


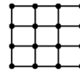
- jede CPU verfügt über eigenen (privaten) Speicher
- Kommunikation über ein Verbindungsnetzwerk
- Zugriff auf Daten anderer CPUs evtl. recht langsam

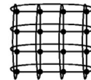
Verbindungs-Netzwerke

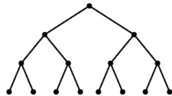
Stern | vollständig vernetzt | Gitter | Torus

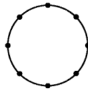

(a)

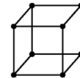

(b)

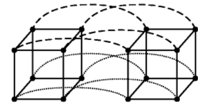

(e)


(f)


(c)

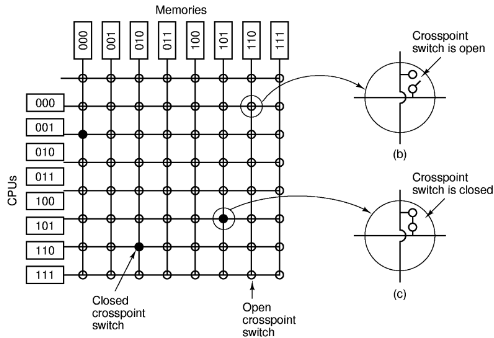

(d)


(g)


(h)

Baum | Ring | Würfel | Hyperwürfel

Kreuzschienenverteiler („crossbar switch“)



- jede CPU kann auf jeden Speicher zugreifen
- hoher Hardwareaufwand: $O(N^2)$ Schalter und Verbindungen
- Konflikte bei gleichzeitigem Zugriff auf einen Speicher

Omega-Netzwerk

3 Stages

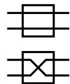
CPUs: 000, 001, 010, 011, 100, 101, 110, 111

Stage 1: 1A, 1B, 1C, 1D
Stage 2: 2A, 2B, 2C, 2D
Stage 3: 3A, 3B, 3C, 3D

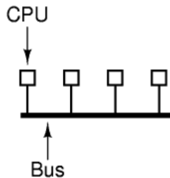
Memories: 000, 001, 010, 011, 100, 101, 110, 111

a, b

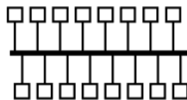
- Schalter „gerade" oder „gekreuzt"
- jede CPU kann auf jeden Speicher zugreifen
- aber nur bestimmte Muster, Hardwareaufwand $O(N \ln N)$



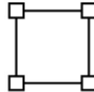
Skalierbarkeit



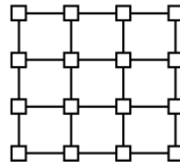
(a)



(b)



(c)



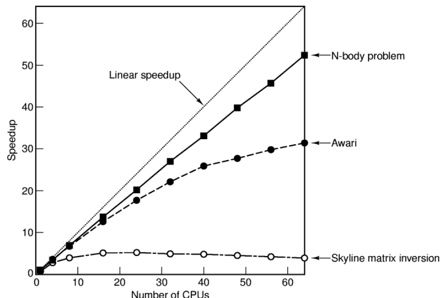
(d)

Wie viele CPUs kann man an ein System anschliessen?

- Bus: alle CPUs teilen sich die verfügbare Bandbreite
- daher normalerweise nur 2..8 CPUs sinnvoll

- Gitter: verfügbare Bandbreite wächst mit Anzahl der CPUs

Speedup



- Maß für die Effizienz einer Architektur / eines Algorithmus'
- wegen Amdahl's Gesetz maximal linearer Zuwachs
- je nach Problem oft wesentlich schlechter

Programmierung: ein ungelöstes Problem

- Aufteilung eines Programms auf die CPUs/nodes?
- insbesondere bei komplexen Kommunikationsnetzwerken

- Parallelität typischer Programme (gcc, spice, ...): kleiner 8
- hochgradig parallele Rechner sind dann Verschwendung
- aber SMP-Lösungen mit 4..16 Prozessoren attraktiv

- z.B. Datenbankanwendungen nur teilweise parallelisierbar
- Vektor-/Feld-Rechner für Numerik, Simulation, ...
- Graphikprozessoren (GPUs) für 3D-Graphik: Feld-Rechner

SIMD für Multimedia

Multimedia-Verarbeitung mit dem PC?

- hohe Anforderungen (Audio, Video, Image, 3D)
- große Datenmengen (z.B. DVD-Wiedergabe)
- einzelne Datenworte klein (8-bit Pixel, 16-bit Audio)
- Parallelverarbeitung wünschenswert

- „Multimedia"-SIMD-Befehle zum Befehlssatz hinzufügen
- Trick: vorhandene ALUs/Datenpfade für SIMD verwenden

- MMX Intel *multimedia extension* 1996
- 3Dnow! 1998
- SSE *SIMD streaming extension* 1999
- AVX 256-bit Rechenwerke 2010

MMX: multimedia extension für x86 (1996)

- Kompatibilität zu alten Betriebssystemen / Applikationen:
  - keine neuen Register möglich $\Rightarrow$ FP-Register nutzen
  - keine neuen Exceptions $\Rightarrow$ Überlauf ignorieren
    $\Rightarrow$ saturation Arithmetic
  - bestehende Datenpfade nutzen $\Rightarrow$ 64 bit
  - möglichst wenig neue Opcodes

  - Test-Applikationen (Stand 1996) $\Rightarrow$ 16 bit dominiert
  - zunächst keine Tools $\Rightarrow$ Assembler
  - aber Bibliotheken mit optimierten Grundfunktionen

- Kompromisse schränken Performance stark ein
- SSE/SSE2/...definiert komplett neuen Befehlssatz

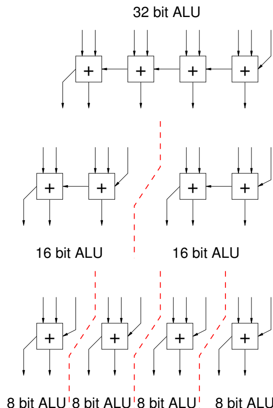
MMX: Grundidee

- 32/64-bit Datenpfade vorhanden,
- aber bei Medien nur 8..16-bit genutzt

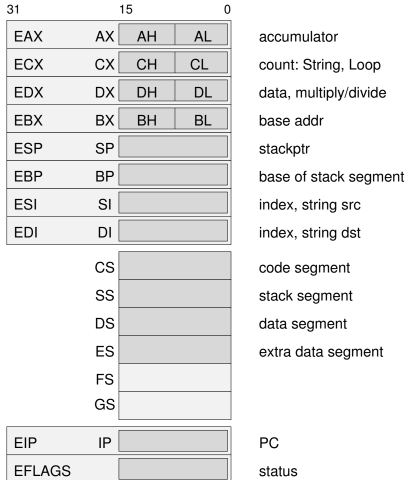
- ALUs auch parallel nutzbar
- carry-chain auftrennen
- parallele Berechnungen, z.B. achtmal 8-bit, viermal 16-bit

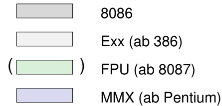
- geringer Zusatzaufwand $\approx 10\%$ Fläche beim Pentium/MMX

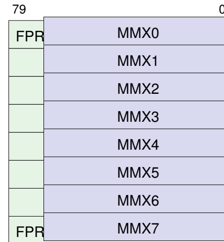
- Performance 2..8x für MMX-Befehle
- Performance 1.5..2x für Applikationen

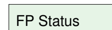


MMX: keine neuen Register...

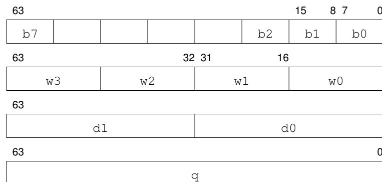








MMX: Packed Data



64-bit Register, 4 Datentypen:

- 8× packed-byte, 4× packed-word, 2× packed double-word,
- quad-word
- Zugriff abhängig vom jeweiligen MMX-Befehl

MMX: Befehlssatz

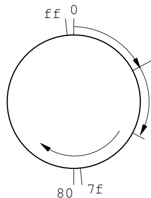
```
EMMS     (FSAV / FRESTOR)        clear MMX state (handle FP regs)
MOVD     mm1, mm2/mem32          move 32 bit data
MOVQ     mm1, mm2/mem64          move 64 bit data
PACKSSWB mm1, mm2/mem64          pack 8*16 into 8*8 signed saturate
PUNPCKH  mm1, mm2/mem64          fancy unpacking (see below)
PACKSSDW mm1, mm2/mem64          pack 4*32 into 4*16 signed saturate
PAND     mm1, mm2/mem64          mm1 AND mm2/mem64 / auch OR/XOR/NAND
PCMPEQB  mm1, mm2/mem64          8*a==b, create bit mask / auch GT
PADDB    mm1, mm2/mem64          8*add 8 bit data
PSUBD    mm1, mm2/mem64          2*sub 32 bit data / signed wrap
PSUBUSD  mm1, mm2/mem64          2*sub 32 bit data / unsigned saturate
PSLL     mm1,mm2/mem64/imm8      shift left mm1 / auch PSRA/PSRL
PMULL/HW mm1, mm2/mem64          4*mul 16*16 store low/high 16 bits
PMADDWD  mm1, mm2/mem64          MAC 4*16 -> 2*32

insgesamt 57 Befehle             (Varianten B/W/D S/US)
```

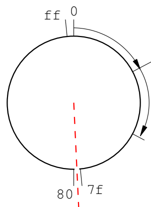
MMX: „saturation arithmetic“

was soll bei einem Überlauf passieren?

- wrap-around
  ..., 125, 126, 127, -128, -127, ...



- saturation
  ..., 125, 126, 127, 127, 127, ...
- Zahlenkreis "aufgeschnitten"
- gut für DSP-Anwendungen



paddw (wrap around):

a3	a2	a1	7FFFh
+	+	+	+
b3	b2	b1	0004h
a3+b3	a2+b2	a1+b1	8003h

paddusw (saturating):

a3	a2	a1	7FFFh
+	+	+	+
b3	b2	b1	0003h
a3+b3	a2+b2	a1+b1	7FFFh

MMX: „packed multiply add word“

für Skalarprodukte:

pmaddwd

a3	a2	a1	a0
*	*	*	*
b3	b2	b1	b0

a3*b3+a2*b2	a1*b1+a0*b0

```
vector_x_matrix_4x4( MMX64* v, MMX64 *m ) {
  MMX64  v0101, v2323, t0, t1, t2, t3;

  v0101 = punpckldq( v, v );  // unpack v0/v1
  v2323 = punpckhdq( v, v );  // unpack v2/v3

  t0    = pmaddwd( v0101, m[0] );  // v0|v1 * first 2 rows
  t1    = pmaddwd( v2323, m[1] );  // v2|v3 * first 2 rows
  t2    = pmaddwd( v0101, m[2] );  // v0|v1 * last 2 rows
  t3    = pmaddwd( v2323, m[3] );  // v2|v3 * last 2 rows

  t0    = paddd(    t0, t1 );      // add
  t2    = paddd(    t2, t3 );      //
  v     = packssdw( t0, t2 );      // pack 32->16, saturate
}
```

MMX: „packed compare“

pcmpgtw:

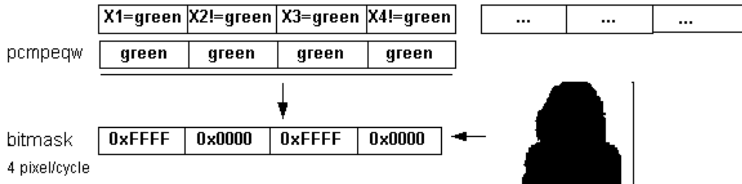
23	45	16	34
>	>	>	>
31	7	16	67
0000h	FFFFh	0000h	0000h

Vergleichsbefehle für packed data?

- schlecht parallelisierbar
- Beispiel: $a > b$?
- was soll passieren, wenn einige Vergleiche wahr sind, andere falsch?

- keine Sprungbefehle in MMX
- Vergleichsbefehle setzen Bit-Masken
- dann logische Operationen und (paralleles) Weiterrechnen

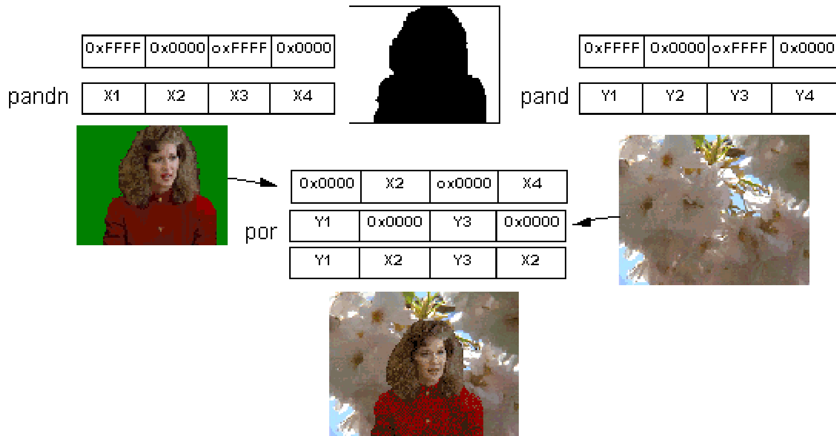
MMX: Beispiel Chroma-Keying (1)



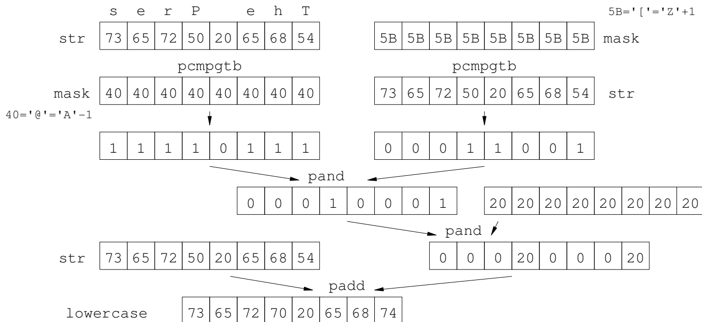
- Überblenden von Person (Objekt) vor einfarbigem Hintergrund („blue screen“)

- MMX berechnet 4 Pixel/Takt
- Schritt 1: Maske erstellen (Beispiel: 16 bit/Pixel „high-color“)
- keine Branch-Befehle notwendig

MMX: Beispiel Chroma-Keying (2)



MMX: Beispiel toLowerCase()



aber: Probleme mit Umlauten...

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0																
1																
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

(Intel MMX appnote)

SSE: zweiter SIMD-Befehlssatz für x86 (1999)

- MMX wegen der Kompromisse nur wenig effektiv
- aber Multimedia/3D-Spiele zunehmend wichtig

Einführung eines zweiten SIMD-Befehlssatzes für x86:

SIMD-Streaming Extension

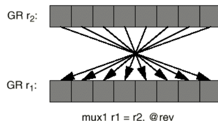
- neue Register, 128 bit
- 70 neue Befehle, 4-fach SIMD
- insbesondere auch schnelle Gleitkommarechnung
- benötigt Unterstützung vom Betriebssystem
- seitdem mehrfach erweitert (SSE2, SSE3, SSE4, AVX)

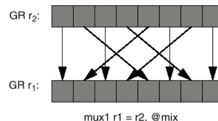
SSE: mux1-Befehl

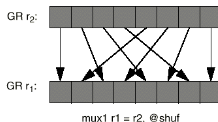
- Verschieben von Daten
- je acht 8-bit Pixel

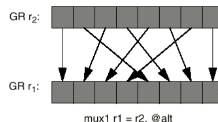
- „Pixel-Reversal“
- „Butterfly“
- achtfache Kopie

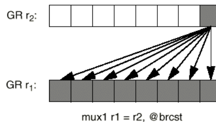
- Bildverarbeitung
- Fast-Fourier Transform





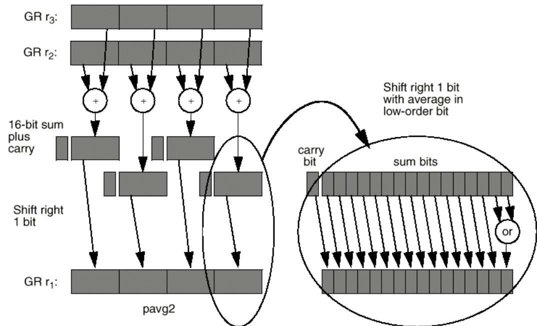






SSE: pavg-Befehl: parallel average

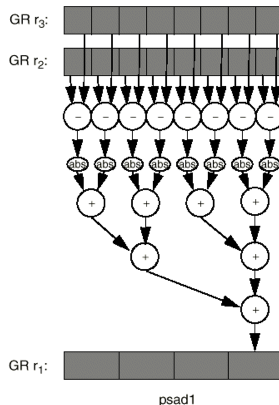
- pro Register
  je 4 16-bit Werte
- Mittelwertbildung
- Rundung
- ein Takt



SSE: psadbw-Befehl: parallel-sum of absolute differences

- je acht 8-bit Pixelwerte pro Register
- parallele Berechnung der Differenz von korrespondierenden Pixeln
- Berechnung der Absolutwerte
- Summation
- alles in einem Takt

- für MPEG/H.26x Videokodierung



SMP: Symmetric Multiprocessing

- mehrere Prozessoren teilen gemeinsamen Hauptspeicher
- Zugriff über Verbindungsnetzwerk oder Bus
- geringer Kommunikationsoverhead

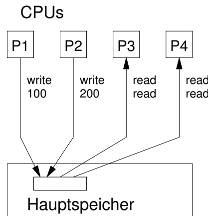
- bus-basierte Systeme sind sehr kostengünstig
- aber schlecht skalierbar (Bus wird Flaschenhals)
- lokale Caches für gute Performance notwendig
- MESI-Protokoll und Snooping für Cache-Kohärenz

- Dual-/Multi-Core Prozessoren sind „SMP on-a-chip“

SMP: Eigenschaften

„symmetric multiprocessing“:

- alle CPUs gleichrangig, Zugriff auf Speicher und I/O
- gleichzeitiger Zugriff auf eine Speicheradresse?
- „strikte“ / sequentielle / Prozessor- / schwache Konsistenz:



W1 100	W1 100	W2 200
W2 200	R3 = 100	R4 = 200
R3 = 200	W2 200	W1 100
R3 = 200	R3 = 200	R3 = 100
R4 = 200	R4 = 200	R4 = 100
R4 = 200	R4 = 200	R4 = 100

SMP: Cache-Kohärenz

aus Performancegründen:

- jeder Prozessor hat seinen eigenen Cache (L1, L2, ...)
- aber gemeinsamer Hauptspeicher

Problem der *Cache-Kohärenz*:

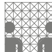
- Prozessor X greift auf Daten zu, die im Cache von Y liegen
  - Lesezugriff von X: Y muss seinen Wert liefern
  - Schreibzugriff von X: Y muss Wert von X übernehmen
  - gleichzeitiger Zugriff: problematisch
- MESI-Protokoll mit Snooping
- Caches enthalten Wert, Tag, und 2 bit MESI-Zustand

SMP: MESI

MESI := modified, exclusive, shared, invalid

- jede Cache-Speicherstelle wird um 2 Statusbits erweitert
- alle Prozessoren überwachen die Zugriffe anderer Prozessoren
- entsprechende Aktualisierung der Statusbits

Zustand:	Bedeutung (grob):
invalid	Wert ist ungültig (z.B. noch nie geladen)
exclusive	gültiger Wert, nur in diesem Cache vorhanden
modified	gültiger Wert, nur in diesem Cache vorhanden, gegenüber Hauptspeicher-Wert verändert
shared	gültiger Wert, in mehreren Caches vorhanden

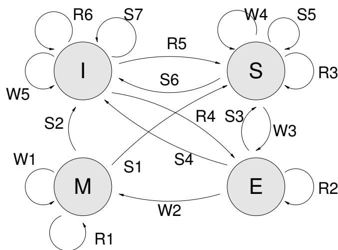


MESI: Zustände

MESI-Zustand	Cache-Eintrag gültig?	Wert im Speicher gültig?	Kopien in anderen Caches?	Zugriff betrifft
M	ja	nein	nein	Cache
E	ja	ja	nein	Cache
S	ja	ja	möglich	Speicher
I	nein	unbekannt	möglich	Speicher

- Cache-Strategie: write-back, kein write-allocate
- Schreibzugriffe auf M führen nicht zu Bus-Transaktionen
- Werte in E stimmen mit Hauptspeicherwerten überein
- Werte in S sind aktuell, Lesezugriff ohne Bus-Transaktion
- Schreibzugriff auf S: lokal S, fremde auf I, Wert abspreichern
- bei write-through: Zustände S/I, kein M/E

MESI: Übergänge



Lesezugriffe:

M-M	R1	Cache-Hit, CPU bekommt Daten
E-E	R2	Cache-Hit, CPU bekommt Daten
S-S	R3	Cache-Hit, CPU bekommt Daten
I-E	R4	Miss, Speicher liefert Daten
I-S	R5	Miss, externer Cache liefert Daten
I-I	R6	Miss, Adresse nicht cacheable

Schreibzugriffe:

M-M	W1	Hit, CPU aktualisiert Cache
E-M	W2	Hit, CPU aktualisiert Cache
S-E	W3	Hit (write-back): Cache aktualisiert, Buszyklus markiert fremde Kopien als invalid
S-S	W4	Hit (write-through): Caches und Speicher aktualisiert
I-I	W5	Miss, Speicher schreiben, aber kein write-allocate

Snoop-Zyklen

M-S	S1	Hit, Speicher schreiben
M-I	S2	Hit, Speicher schreiben
E-S	S3	Hit, aber nicht modifiziert

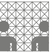
usw.

MESI: Snooping

„schnüffeln"-Prinzip:

- alle Prozessoren überwachen alle Bus-Transaktionen
- 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

- erfordert spezielle Snoop-Logik im Prozessor
- garantiert Cache-Kohärenz
- gute Performance, aber schlechte Skalierbarkeit



MESI: Snooping

MESI-Status Wert A:	CPU1	CPU2
CPU 1 lädt Wert A	E	I
CPU 1 modifiziert A	M	I
CPU 2 lädt A (aber Wert modified)	-	-
CPU 1 SNOOP! (CPU2 read gestoppt)	- -	- -
CPU 1 schreibt A (CPU2 read gestoppt)	E	I
CPU 2 lädt A (CPU2 read restart, A shared)	S	S

MESI: beim Pentium-III

The following section describes the cache control protocol currently defined for the Intel Architecture processors. This protocol is used by the P6 family and Pentium® processors. The Intel486™ processor uses an implementation defined protocol that does not support the MESI four-state protocol, but instead uses a two-state protocol with valid and invalid states defined.

In the L1 data cache and the P6 family processors' L2 cache, the MESI (modified, exclusive, shared, invalid) cache protocol maintains consistency with caches of other processors. The L1 data cache and the L2 cache has two MESI status flags per cache line. Each line can thus be marked as being in one of the states defined in Table 9-3. In general, the operation of the MESI protocol is transparent to programs.

The L1 instruction cache implements only the "SI" part of the MESI protocol, because the instruction cache is not writable. The instruction cache monitors changes in the data cache to maintain consistency between the caches when instructions are modified. See Section 9.7., "Self-Modifying Code", for more information on the implications of caching instructions.

Table 9-3. MESI Cache Line States

Cache Line State	M (Modified)	E (Exclusive)	S (Shared)	I (Invalid)
This cache line is valid?	Yes	Yes	Yes	No
The memory copy is…	…out of date	…valid	…valid	—
Copies exist in caches of other processors?	No	No	Maybe	Maybe
A write to this line …	…does not go to bus	…does not go to bus	…causes the processor to gain exclusive ownership of the line	…goes directly to bus

SMP: volatile

- MESI-Verfahren garantiert Cache-Kohärenz
- für Werte im Cache und im Hauptspeicher

Vorsicht: was ist mit den Registern?

- Variablen in Registern werden von MESI nicht erkannt
- Compiler versucht, häufig benutzte Variablen soweit wie möglich in Registern zu halten

- shared-Variablen niemals in Registern halten
- Java/C: Deklaration als `volatile`

SMP: Atomic Operations

7.1. LOCKED ATOMIC OPERATIONS

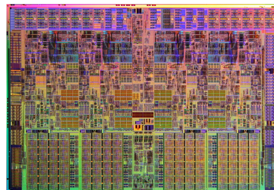
The 32-bit Intel Architecture processors support locked atomic operations on locations in system memory. These operations are typically used to manage shared data structures (such as semaphores, segment descriptors, system segments, or page tables) in which two or more processors may try simultaneously to modify the same field or flag. The processor uses three interdependent mechanisms for carrying out locked atomic operations:

- Guaranteed atomic operations.
- Bus locking, using the LOCK# signal and the LOCK instruction prefix.
- Cache coherency protocols that insure that atomic operations can be carried out on cached data structures (cache lock). This mechanism is present in the P6 family processors.

These mechanisms are interdependent in the following ways. Certain basic memory transactions (such as reading or writing a byte in system memory) are always guaranteed to be handled atomically. That is, once started, the processor guarantees that the operation will be completed before another processor or bus agent is allowed access to the memory location. The processor also supports bus locking for performing selected memory operations (such as a read-modify-write operation in a shared area of memory) that typically need to be handled atomically, but are not automatically handled this way. Because frequently used memory locations are often cached in a processor's L1 or L2 caches, atomic operations can often be carried out inside a processor's caches without asserting the bus lock. Here the processor's cache coherency protocols insure that other processors that are caching the same memory locations are managed properly while atomic operations are performed on cached memory locations.

Note that the mechanisms for handling locked atomic operations have evolved as the complexity of Intel Architecture processors has evolved. As such, more recent Intel Architecture processors (such as the P6 family processors) provide a more refined locking mechanism than earlier Intel Architecture processors, as is described in the following sections.

SMP: Multi-Core Prozessoren



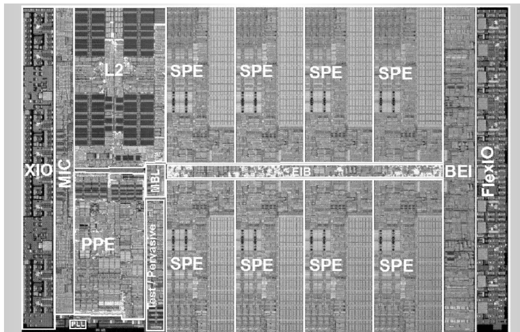
- aktuelle CMOS-Technologie erlaubt mehrere Prozessoren pro Chip

- entweder einige Hochleistungsprozessoren
- oder bis Hunderte einfacher Prozessoren
- aber: „normale“ Anwendungen wenig parallelisierbar

- Dual-Core / Multi-Core Prozessoren: on-chip SMP
- separate L1-Caches, separate/shared L2, (shared L3)
- MESI und Snooping on-chip effizient realisierbar

(Photo: Intel Core-i7 mit 4/8 CPUs, 8 MB L3-Cache on-chip, 2009)

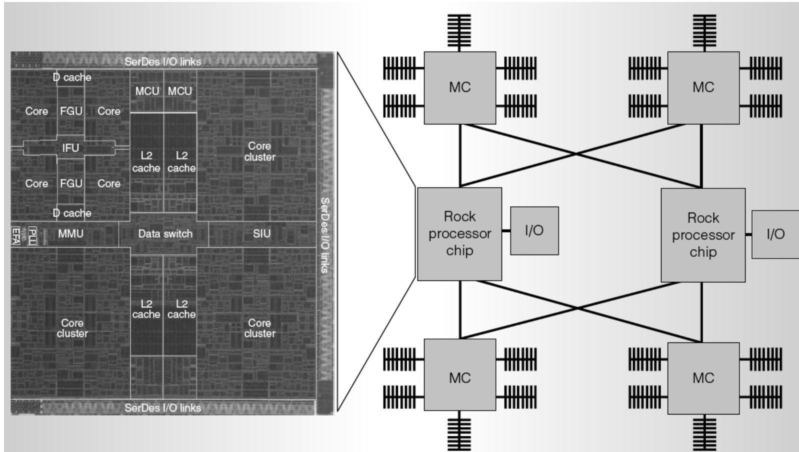
SMP: IBM/Sony/Toshiba „Cell“ processor: 1+8 Core



BEI	Broadband engine interface	MBL	MIC bus logic
EIB	Element interconnect bus	PPE	Power processor element
FlexIO	High-speed I/O interface	SPE	Synergistic processor element
L2	Level 2 cache	XIO	Extreme data rate I/O cell
MIC	Memory interface controller		Test control unit/pervasive logic

(M.Kistler et.al., Cell multiprocessor communication network, IEEE Micro 2006)

SMP: Sun „Rock“ processor: 16-Core, 32-Threads



(S.Chaudhry et.al., A high-performance SPARC CMT processor, IEEE Micro 2009)

SMP: Erreichbarer Speedup (bis 32 Threads)

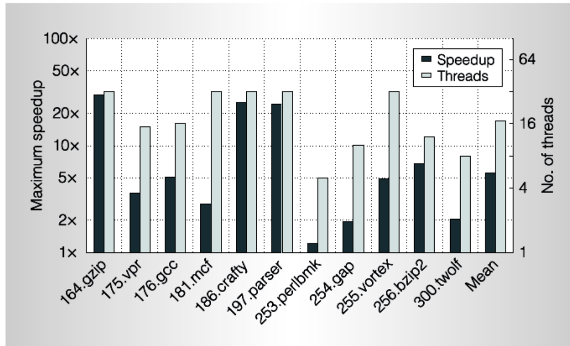


Figure 4. Maximum speedup achieved on up to 32 threads over single-threaded execution (black bars) and minimum number of threads at which the maximum speedup occurred (gray bars).

(M.J.Bridges et.al., Revisiting the sequential programming model for the multicore era, IEEE Micro 2009)

Supercomputer

Bezeichnung für die jeweils schnellsten Maschinen

- Prestigeprojekte / Forschung
- Ranking nach Linpack-Benchmark (Matrix-Multiplikation)

- derzeit Systeme mit 10K..100K Prozessoren/Cores
- MIMD
- maßgeschneiderte Verbindungsnetzwerke
- erfordert entsprechend leistungsfähige I/O-Geräte
- und entsprechende externe Netzwerkanbindung

(Top-500 Listen und Diagramme: www.top500.org)

Leistungssteigerung der Spitzenrechner seit 1993

Jahr	Rechner	Linpack in Gflop/s	Zahl der Prozessoren
1993	Fujitsu NWT	124	140
1994	Intel Paragon XP/S MP	281	6.768
1996	Hitachi CP-PACS	368	2.048
1997	Intel ASCI Red (200 MHz Pentium Pro)	1.338	9.152
1998	ASCI Blue-Pacific (IBM SP 640E)	2.144	5.808
1999	ASCI Intel Red (Pentium II Xeon)	2.379	9.632
2000	ASCI White, IBM (SP Power 3)	4.903	7.424
2002	Earth Simulator, NEC	35.610	5.104
2006	JUBL	45.600	16.384
2008	IBM Roadrunner	1.105.000	124.400[1]
2009	Jaguar am ORNL, Cray	1.759.000	224.162[2]

[1] Anzahl der Kerne (6.480 Opteron, 12.960 Cell)
[2] Anzahl der Kerne (Basis: Opteron)

Beispiel: ASCI Red (1997)

9216 CPUs   594 GB RAM   1 TB Disk   I/O: 1.0 GB/s   1.8 TFLOPS

ASCI Red: Architektur

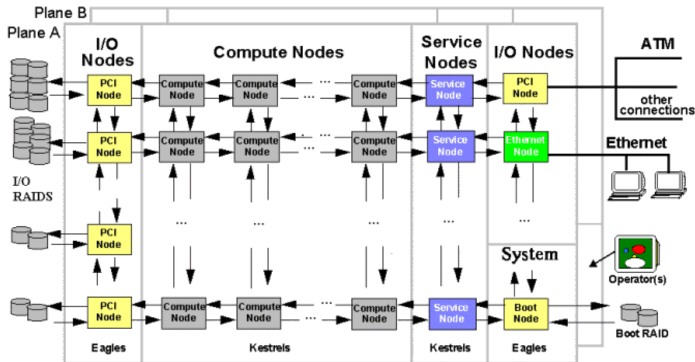


Figure 5: Logical System Block Diagram for the ASCI Option Red Supercomputer. This system uses a split-plane mesh topology and has 4 partitions: System, Service, I/O and Compute. Two different kinds of node boards are used and described in the text: the Eagle node and the Kestrel node. The operators console (the SPS station) is connected to an independent ethernet network that ties together patch support boards on each card cage.

ASCI Red: Verbindungsnetzwerk

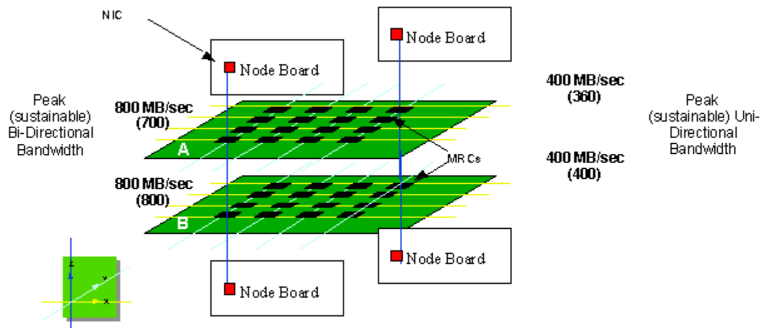
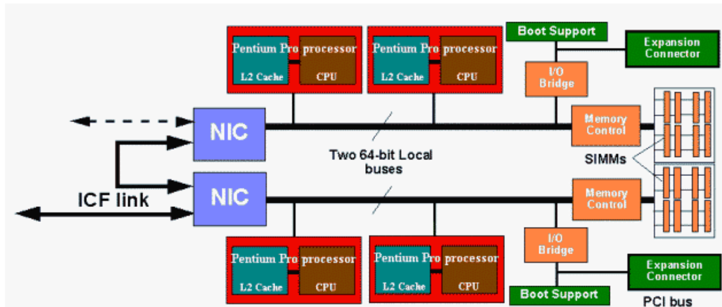


Figure 4: ASCI Option Red Supercomputer 2 Plane Interconnection Facility (ICF). Bandwidth figures are given for NIC-MRC and MRC-MRC communication. Bi-directional bandwidths are given on the left side of the figure while directional bandwidths are given on the right side. In both cases, sustainable (as opposed to peak) numbers are given in parentheses.

ASCI Red: Compute Nodes



Platine enthält zwei Nodes mit je 2 CPUs und 128 MByte Speicher

Figure 3: The ASCI Option Red supercomputer Kestrel Board. This board includes two compute nodes chained together through their NIC's. One of the NIC's connects to the MRC on the backplane through the ICF Link.

ASCI Red: Performance

- 200 MHz PentiumPro: 200 MFLOPS peak
- 9200 CPUs: 1.8 TFLOPS peak

- 1 TFLOPS Grenze am 07.12.1996 erreicht:
  - handoptimierter Algorithmus (LRU blocked, pivoting)
  - handoptimierter Assemblercode
  - Maschine 80% vollständig: 140 MFLOPS/node
- speicherlimitierte Programme: $<$ 20 MFLOPS / node
- compilierte Programme: 20..80 MFLOPS / node

- 640 Festplatten, 1540 Netzteile, 616 „backplanes“
- MTBF $>$ 50 hours (bzw. 97% nodes aktiv für $>$ 4 Wochen)

Jaguar (2009)

„petascale“ supercomputer

- Cray XT5, SuSE Enterprise Linux
- Oak Ridge National Laboratory

- 224.256 AMD Opteron Cores
- 16 GB Hauptspeicher pro CPU
- „SeaStar2“ Verbindungsnetzwerk: doppelter Torus
  Bandbreite 7.6 GByte/s zwischen Nachbarn
- 1750 TFLOPS (Nov. 2009)

(www.nccs.gov/computing-resources/jaguar/#XT5-6-Core-Upgrade)

Literatur: Vertiefung

- Tanenbaum SCO (Kapitel 8)
- Hennessy & Patterson (Kapitel 8, Anhang A)

- IEEE Micro Magazine, *Special issue on Multicore systems*, Vol. 28-3, 2008

- Linux parallel-processing-HOWTO
- www.top500.org