

**Ausführungszeit**  $t[s] = \frac{\text{Taktzyklen [Takte]}}{\text{Frequenz [Hz]}} = \frac{C}{f}$

**Leistung absolut**  $L_{abs}[MIPS] = \frac{\text{Befehlsanzahl}}{\text{Ausführungszeit [s]} \cdot 10^6} = \frac{n}{t \cdot 10^6}$

**Leistung relativ**  $L_{rel}[MIPS] = \frac{\text{Referenzzeit [s]}}{\text{Ausführungszeit [s]}} \cdot \text{RefLeistung [MIPS]} = \frac{t_{ref}}{t_{mess}} \cdot L_{ref}$

**Clocks per Instruction**  $CPI = \frac{\text{Taktzyklen [Takte]}}{\text{Befehlsanzahl}} = \frac{C}{n}$

**Gewichtete mittlere**  $CPI_G = \sum (CPI_{Bef.gr.} \cdot \text{Rel.Häufigkeit}_{Bef.gr.}) = \sum_{i=1}^n (CPI_i \cdot p_i)$

**Instructions per Clock**  $IPC = \frac{\text{Befehlsanzahl}}{\text{Taktzyklen [Takt]}} = \frac{n}{C}$

**Speedup**  $S_n = \frac{1}{\text{AnteilSeruell} + \text{Overhead} + \frac{\text{AnteilParallel}}{\text{AnzahlProzessoren}}} = \frac{1}{A_{seriell} + o(n) + \frac{A_{parallel}}{n}}$

**Effizienz**  $E_n = \frac{\text{Speedup}}{\text{AnzahlProzessoren}} = \frac{S_n}{n}$

Bezeichnung	Konflikterkennung	Issue-Struktur	Scheduling	Hauptmerkmal	Beispiele
Superskalar	Hardware	Dynamisch	Statisch	In-order Execution	Sun UltraSPARC II/ III
Out of Order	Hardware	Dynamisch	Dynamisch mit Spekulation	Out of Order mit Spekulation	Pentium III, Pentium 4, MIPS 10000
VLIW	Software	Statisch	Statisch	Keine Konflikte	Trimedia, diverse DSPs

## Prozessorarchitekturen

**CISC** complex instruction set computer

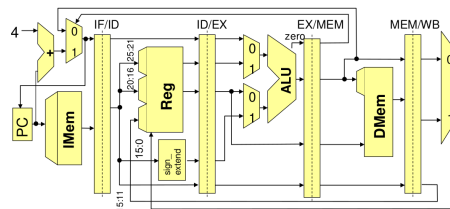
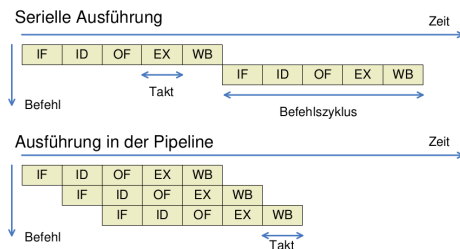
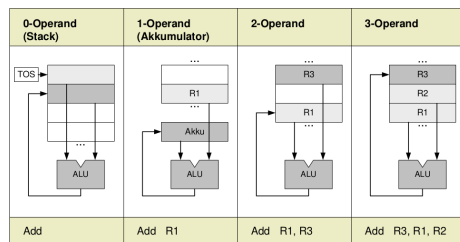
- Einfache und komplexe Befehle
- Heterogener Befehlssatz
- verschiedene Taktzahl pro Befehl
- Viele Befehlscode-Formate mit unterschiedlicher Länge
- Mikroprogrammwerk
- Vermischung von Verarbeitungs & Speicherbefehlen
- schwierig, unter  $CPI = 2$  zu kommen

**RISC** reduced instruction set computer

- wenige, einfache Befehle
- orthogonaler Befehlssatz
- meist 1 Takt pro Befehl
- wenige Befehlscode-Formate mit einheitlicher Länge
- Direktverdrahtung
- Trennung von Verarbeitungs & Speicherbefehlen
- hohe Ausführungsgeschwindigkeit ( $CPI \leq 1$ )

## MIPS

- Microprocessor without interlocked pipeline stages
- 32-bit Architektur/64-bit Erweiterung



Aufgaben der einzelnen Phasen

**Befehlsholphase** Lesen des aktuellen Befehls; separater Speicher, zur Vermeidung von Konflikten mit Datenzugriffen

**Dekodier & Register-Lese-Phase** Lesen der Register möglich wegen fester Plätze für Nr. im Befehlswort

**Ausführungs & Adressberechnungsphase** Berechnung arithmetischer Funktion bzw. Adresse für Speicherzugriff

**Speicherzugriffsphase** Wird nur bei Lade & Speicherbefehlen benötigt

**Abspeicherungsphase** Speichern in Register, bei Speicherbefehlen nicht benötigt

### Hazards

- resource hazards
- data hazards: Datenabhängigkeiten

**Antidatenabhängig** falls Befehl j eine Speicherzelle beschreibt, die von i noch gelesen werden müsste. WAR (write after read)

**Ausgabeabhängig** falls Befehle i und j die selbe Speicherzelle beschreiben. WAW (w.a.w.)

**Datenabhängigkeit** Operation hängt von der vorhergehenden Operation ab. RAW (r.a.w.)

- control hazards: Kontrollabhängigkeiten
  - Gleichheit der Register wird schon in der instruction decode-Stufe geprüft
  - Sprungziel wird in separatem Adressaddierer bereits in instruction decode-Stufe berechnet

### Sprungvorhersage

#### Einfache lokale Prädiktoren

- Vorhersage, ob bedingter Sprung genommen wird oder nicht
- Prädiktion allein anhand der Historie des betrachteten, aktuellen Sprungs
- Historie eines Sprungs wird mit 1, 2 oder n Bits gepuffert

#### Einfache Sprungvorhersage (1 Bit)

- Branch prediction buffer oder branch history table

- Kleiner Speicher, der mit (Teil der) Adresse des Sprungbefehls indiziert wird
- Verwendet nur wenige untere Bits der Adresse
- Enthält 1 Bit: Sprung beim letzten Mal ausgeführt (taken) oder nicht (not taken)
- Prädiktion: Sprung verhält sich wie beim letzten Mal
- Nachfolgebefehle ab vorhergesagter Adresse holen
- Falls Prädiktion fehlerhaft: Prädiktionsbit invertieren
- Alle Sprünge, deren Adressen im Indexteil übereinstimmen, werden derselben Zelle im branch prediction buffer zugeordnet
- Einfachste Art von Puffer (keine Tags, keine Überprüfung, ob Adresse im Puffer)
- Entspricht sehr einfachem Cache
- Hat eine bestimmte Kapazität
- Kann nicht für alle Sprünge Einträge enthalten
- Reduziert branch penalty nur, wenn branch delay länger als Berechnung der Zieladresse mit branch prediction buffer dauert
- Prädiktion kann fehlerhaft sein
- Prädiktion kann von anderem Sprungbefehl stammen (mit gleichen Bits im Indexteil der Adressen)

### Einführung von Tag Bits

- wenn Sprung fast immer ausgeführt (taken) wird, entstehen 2 Fehler anstatt 1
- Tag: gültiger Eintrag, falls Tag-Bits gleich sind
- Alle Sprünge, deren Adressen im Indexteil übereinstimmen, werden derselben Zelle im branch prediction buffer zugeordnet. Überprüfung mittels tags, ob es der richtige Eintrag ist.
- Allgemein: Fehlerrate von 1-Bit Prädiktor ist für Sprünge in Schleifenkonstrukten doppelt so hoch wie die Anzahl ausgeführter Sprünge

### 2 Bit Vorhersagen

- Änderung der Vorhersage nur, wenn 2 falsche Vorhersagen in Folge
- 2-Bit Branch-Prediction Buffer: Speicherung der Historie, Befehlsadressen als Zugriffsschlüssel

### n-Bit Prädiktor

- Verwendet n-Bit Zähler
- Sättigungsarithmetik (kein wrap around bei Überlauf)
- Kann Werte zwischen 0 und  $2^{n-1}$  annehmen
- Wenn Zähler größer als Hälfte des Maximums ( $2^{n-1}$ ): Vorhersagen, dass Sprung ausgeführt wird; ansonsten vorhersagen, dass Sprung nicht genommen wird
- Zähler wird bei ausgeführtem Sprung inkrementiert und bei nicht ausgeführtem dekrementiert
- Praxis: 2-Bit Prädiktor ähnlich gut wie n-Bit Prädiktor

## Korrelierende Prädiktoren

- Betrachtet nur Verhalten eines Sprungs, um dessen Verhalten vorherzusagen
- Arbeitet rein lokal
- Verbesserung durch Betrachtung des Verhaltens anderer Sprünge
- erhält so korrelierenden/zweistufigen Prädiktor
- Prinzip: Aufgrund globaler Information wird einer von mehreren lokalen Prädiktoren ausgewählt
- Beziehen zur Vorhersage des Verhaltens Kontext-Information mit ein, d.h. die Historie anderer Sprungbefehle
- Prädiktor benutzt globale Kontext-Bits, um einen von mehreren lokalen Prädiktoren auszuwählen
- Betrachten wiederholte Ausführung des Codefragments

### Zweistufiger Prädiktor

- Es existieren 2 lokale Prädiktoren, beide je 1-Bit
- Kontext: Letzter Sprung wurde (nicht) ausgeführt
- Anhand des Kontexts wird lokaler Prädiktor für die Vorhersage des aktuell betrachteten Sprungs ausgewählt
- Letzter Sprung ist i.a. nicht gleich aktuellem, vorherzusagendem Sprung
- Notation des Prädiktorstatus:  $\langle X \rangle / \langle Y \rangle$  mit
  - $\langle X \rangle$ : Vorhersage, falls letzter Sprung not taken
  - $\langle Y \rangle$ : Vorhersage, falls letzter Sprung taken
  - $\langle X \rangle \vee \langle Y \rangle$  Vorhersagen: entweder T oder NT

### (m,n)-Prädiktor

- Betrachtet als Kontext das Verhalten der letzten m Sprünge, um aus  $2^m$  vielen lokalen Prädiktoren einen n-Bit Prädiktor auszuwählen
- Höhere Vorhersagegenauigkeit
- Erfordert kaum Hardwareaufwand
- Sprunggeschichte kann in m-Bit Schieberegister gespeichert werden (1 Bit gleich 1 wenn Sprung taken)
- Vorhersagepuffer adressiert via Konkatenation von unteren Adressbits der Sprungbefehlsadresse
- m Bit globaler Sprunggeschichte

## High Performance Befehlsdekodierung

reine Vorhersage eines Sprungs i.d.R. nicht ausreichend

- Insbesondere: Falls mehrere Befehle pro Takt auszugeben sind
- Befehlsstrom mit großer Bandbreite erforderlich
- Kontrollflussabhängigkeiten dürfen nicht „wahrnehmbar“ sein
- Pufferung von Sprungzielen und nicht nur Vorhersage des Sprungverhaltens (branch target buffer)
- Integrierte Einheit für das Holen der Befehle
- Vorhersage von Rücksprungadressen (bei Prozeduraufruf)

## Branch Target Buffer

5-stufige Pipeline, Auswertung von Sprungbedingungen in EX

- Branch delay von 2 Takten
- Mit Sprungvorhersage (branch prediction buffer)
- Zugriff erfolgt in ID (Adresse des Sprungbefehls schon in IF bekannt; aber evtl. angesprungenes Ziel erst nach Befehlsdekodierung)
- Nächste vorhergesagte Instruktion kann erst nach ID geholt werden
- Branch delay = 1, falls Prädiktion korrekt
- Mit Pufferung des Sprungziels (branch target buffer)
- Zugriff auf branch target buffer erfolgt in IF. Verhalten wie „echter“ Cache, adressiert mit Sprungbefehlsadresse
- Liefert vorhergesagte Adresse als Ergebnis

- Keine Verzögerung, falls Prädiktion korrekt
- Zusätzliche Speicherung auch des Sprungziels
- Bei geschickter Organisation kann das Fließband immer gefüllt bleiben; die Sprünge kosten dann effektiv keine Zeit;  $CPI < 1$  möglich.

### Eigenschaften

- Verzögerung durch Sprung kann vollständig vermieden werden, da bereits in IF Entscheidung über nächsten Befehlszähler (PC) getroffen wird.
- Da Entscheidung allein auf Basis des PC getroffen wird, muss überprüft werden, ob Adresse im Puffer
- Speicherung nur für Sprünge notwendig, die als ausgeführt vorhergesagt werden
- Entsteht ursprüngliche Sprung-Verzögerung plus Aufwand zur Aktualisierung des Vorhersagepuffers

## Integrierte Befehls-Hol-Einheit (IF Unit)

Insbesondere mit Blick auf multiple-issue Prozessoren eigene (autonome) funktionale Einheit für Befehlsholphase

- Führt Befehlscodes in Pipeline ein
- Integrierte Funktionalitäten
- Sprungvorhersage wird Teil der Befehlsholphase
- Instruction Pre-fetch: Insbes. um mehrere Befehle pro Takt liefern (und später ausgeben) zu können, läuft Befehlsholen weiterer Dekodierung voraus (= pre-fetch)
- Zugriff auf Befehlsspeicher: Bei mehreren Befehlen pro Takt mehrere Zugriffe erforderlich (bei Cache auf ggfs. mehrere cache lines). Werden hier koordiniert/geplant
- Befehlspuffer: Befehle können hier (lokal im Prozessor!) von Issue-Stufe nach Bedarf abgerufen werden

## Vorhersage von Rücksprungadressen

Allgemeines Ziel: Vorhersage indirekter Sprünge (d.h. bzgl. Basisadresse in Register)

- Hauptverwendung: Rückkehr aus Prozeduraufrufen
- MIPS: Prozeduraufruf per jal proc, Rückkehr per jr \$31
- Vorhersage mit branch target buffer schlecht, da Aufruf aus unterschiedlichen Codeteilen heraus möglich
- Methode: (Stack-) Speicher für Rücksprungadressen
- Push bei Prozeduraufruf (call), und
- Pop bei Rücksprung (return)
- Vorhersagequalität „perfekt“, wenn Stack-Puffer größer als maximale Aufruftiefe

## Multiple-Issue-Architekturen

### Mehrere Ausführungseinheiten

- Weitere Leistungssteigerung:  $CPI < 1$
- Mehrere Befehle pro Takt ausgeben
- Zwei Grundtypen von multiple-issue Prozessoren:
  - Superskalar: variable Anzahl von Befehlen pro Takt
  - VLIW/EPIC: Feste Anzahl von Befehlen ausgegeben, definiert durch Befehlscode (weitgehende Planung der Issue-Phase durch Compiler)

## Superskalar

statisch: Details der Befehlsausgabe

- In IF werden 1-n Befehle von Instruction Fetch Unit geholt (ggfs. Max. von n nicht immer möglich)
- Befehlsgruppe, die potentiell ausgegeben werden kann = issue packet
- Konflikte bzgl. Befehlen im issue packet werden in Issue-Stufe in Programmreihenfolge geprüft

- Befehl ggfs. nicht ausgegeben (und alle weiteren)
- Aufwand für Prüfung in Issue-Stufe groß!
- Wegen Ausgewogenheit der Pipeline-Stufen ggfs. Issue in mehrere Stufen unterteilen = nicht-trivial
- Parallele Ausgabe von Befehlen limitierender Faktor superskalärer Prozessoren

### MIPS mit statischem Scheduling

- Annahme: 2 Befehle pro Takt können ausgegeben werden (1x ALU, Load/Store plus 1x FP)
- Einfacher als 2 beliebige Befehle (wegen „Entflechtung“)
- 2 Befehlsworte holen (64-Bit Zugriff, komplexer als bei nur 1 Befehl)
- Prüfen, ob 0/1/2 Befehle ausgegeben werden können
- Befehl(e) ausgeben an korrespondierende funktionale Einheiten
- Prüfen auf Konflikte durch Entflechtung vereinfacht
- Integer und FP-Operationen nahezu unabhängig (verschiedene Registersätze)
- Abhängigkeiten nur bei Speichertransfers möglich (von Integer-ALU für FP ausgeführt)
- Einschränkung des issue
- Leistungssteigerung nur bei geeignetem Anteil von FP-Operationen sowie geeigneter Verflechtung durch Compiler

## Dynamisches Scheduling - in-order-execution

- Jeder Befehl, der aus der Instruction fetch-Einheit kommt, durchläuft das Scoreboard (Scoreboarding)
- Wenn für Befehl alle Daten/Operanden bekannt sind und Ausführungseinheit frei ist, wird Befehl gestartet
- Alle Ausführungseinheiten melden abgeschlossene Berechnungen dem Scoreboard
- Scoreboard erteilt Befehlen die Berechtigung zum Abspeichern von Ergebnissen, sofern Speichereinheit frei ist und Antidaten- und Ausgabeabhängigkeiten berücksichtigt sind und prüft, ob dadurch neue Befehle ausführbereit werden
- Zentrale Datenstruktur: Scoreboard (für Befehlsstatus)
- load/store-Architektur
- mehrere funktionale Einheiten
- Scoreboarding für MIPS nur sinnvoll wenn
- für FP-Pipeline (Operationen mit mehreren Taktzyklen)
- und mehrere funktionale Einheiten (zB: 2xMult, Div, Add, Int)

## Verfahren von Tomasulo

- erlaubt auch bei Ausgabe- und Antidatenabhängigkeiten, die Reihenfolge zu vertauschen
- Umbenennung der Register
- verschiedenen Benutzungen eines Registers werden verschiedene Speicherzellen zugeordnet
- Jeder funktionalen Einheit wird eine Reservation Station zugeordnet
- Reservation Stations enthalten die auszuführende Operation und die Operanden/tags des Operanden
- Sind alle Operanden bekannt und ist die funktionale Einheit frei, so kann die Bearbeitung beginnen
- Am Ende der Bearbeitung wird das Ergebnis von allen Einheiten übernommen, die das Ergebnis benötigen
- Verteilen der Daten erfolgt vor der Abspeicherung im Registerspeicher
- Aus den tag bits geht hervor, aus welcher Einheit der Operand kommen muss
- Registeradressen werden dynamisch auf größere Anzahl von Plätzen in den Reservation Stations abgebildet, d.h.

- Register effektiv umbenannt
- Performance-Beschränkungen wegen weniger Register werden so umgangen

## Register Renaming

- Verwendung temporärer Register für (logisch) neue möglicherweise interferierende Belegung
- Alle Namenskonflikte durch Umbenennung auflösbar (Voraussetzung: genügend temporäre Register)
- Wichtige Hardwarestruktur: Reservation Stations
- Zugeordnet zu funktionalen Einheiten (i.d.R. pro Einheit)
- Puffern Operanden für Befehle (sobald verfügbar)
- Müssen nicht aus Registern gelesen werden
- Ausstehende Operanden verweisen auf Reservation Station, die Eingabe bereitstellen
- Bei aufeinander folgenden Schreibzugriffen auf Register: Nur letzter für Aktualisierung des Inhalts verwendet
- Wichtige Eigenschaften der Verwendung von Reservation Stations anstelle des zentralen Registersatzes
- Konfliktdetektion und Ausführungskontrolle verteilt
- Informationen in Reservation Stations bei den funktionalen Einheiten bestimmen, wann Ausführung eines Befehls möglich ist
- Ergebnisse werden direkt zu den funktionalen Einheiten (in jeweiliger Reservation Station) weitergereicht
- Erweiterte Form des Forwarding
- Realisiert implizit Register Renaming
- durch gemeinsamen Ergebnisbus (common data bus)

## Multiple-Issue mit dynamischem Scheduling

- Nachteil von statischem Scheduling: Latenzzeiten werden ca. mit Länge des issue packets skaliert
- Längere Verzögerung für Load/Stores bzw. Branches
- Lösung: Erweiterung des Tomasulo-Algorithmus auf Multiple-Issue durch
- Sequentielles Ausgeben mehrerer Befehle an Reservation Stations innerhalb eines Taktes,
- oder „Verbreiterung“ der Ausgabe-Logik (issue logic) zur Behandlung mehrerer Operationen parallel

## VLIW Very Long Instruction Word

- Befehlszuordnung und Konfliktvermeidung durch Compiler
- Compiler muss Zeitbedarf der Speicherzugriffe in Befehlsplanung einbeziehen
- Befehlsstrom mit Tupel von Befehlen
- nicht un-flexibel bei Reaktion auf Laufzeitereignisse
- VLIW hat festes Befehlsformat; höhere Codedichte
- Forwarding Hardware - nach dem EX werden Daten in Pufferregister gespeichert und können vom nächsten Befehl schon genutzt werden
- WB erfolgt erst darauf
- Datenabhängigkeitsproblem wird verringert
- MUX nutzt eher Pufferdaten als Registerdaten
- verschiedene parallele Ausführungseinheiten
- Verteilung von Maschinencode direkt vom Befehlswort im Speicher vorgegeben
- für jede Ausführungseinheit dezidierte Anweisungen
- Meist für stark parallelisierbare Aufgaben verwendet
- Vorteile:
  - parallele Architektur des Prozessors kann während der Programmerstellung zur Optimierung genutzt werden

- Keine aufwendige Prozessorhardware zur Befehlsverteilung erforderlich
- Ausführungszeiten sind im wesentlichen bekannt

### Nachteile:

- Aufwendigere Compiler
- Schlechte Prozessorauslastung bei ungünstigem Code
- Rekompilierung für den Prozessor erforderlich
- Größerer Speicherbedarf, wenn Code nicht parallelisiert werden kann

EPIC = Explicitely Parallel Instruction Computing = IA64

- Im wesentlichen Prinzip des VLIW-Prozessors
- Umsortieren der Befehle und Auflösung der Abhängigkeiten werden durch den Compiler durchgeführt
- Hauptnachteil: Neukompilierung erforderlich
- Keine statische Aufteilung auf Funktionseinheiten
- Effizienteres Befehlswort
- Keine Verwendung von zwangsweise NOPs

Bei der IA64-Architektur werden verschiedene Ansätze verfolgt, um die Prozessorlogik zu vereinfachen.

### 1. Bedingte Befehlsverarbeitung

- Ein Befehl wird abhängig von einem Statusbit ausgeführt
- Dadurch kann die Sprungvorhersage bei einfachen if-then-else Zweigen entfallen
- Die then und else Befehle sind parallel, wobei jeweils nur einer ausgeführt wird

### 2. Statische Sprungvorhersage (Compiler)

Die Optimierung (Finden paralleler Befehle) wird im wesentlichen dem Compiler überlassen.

### 4. Spekulatives Laden von Operanden

- Möglichst geringe Wartezeit auf Operanden
- Schon im Compiler werden entsprechende Ladebefehle vorgezogen.

## Simultaneous Multithreading (SMT)

- Modellprozessor I (2-fach Superskalar)
- Modellprozessor II (2-fach Out-of-Order)

## Speicherhierarchie

- Große Speicher sind langsam
- Anwendung verhalten sich üblicherweise lokal
- Häufig benötigte Speichereinhalte in kleinen Speichern, seltener benötigte Inhalte in großen Speichern ablegen
- Bis zu sechs Ebenen unterscheidbar

Ebene	Latenz	Kapazität
Register	100 ps	1 KB
Cache	1 ns	12 MB
Hauptspeicher/RAM	10 ns	8 GB
Festplatte	10 ms	1 TB
CD-ROM/DVD	100 ms	50 GB
Magnetbänder	100 s	5 TB

## Speicherarchitekturen

### Adresspipelining

- Aufteilen des Speicherzugriffs in mehrere Phasen
- parallele gestaffelte Abarbeitung dieser Phasen für mehrere Speicherzugriffe
- Adresse auslesen/dekodieren; Daten mit Prozessor

Lesezugriff auf Speicher

- Matrixaufbau eines Speichers

- Aufteilen der Speicheradresse in Zeilen- und Spaltenadresse
- Dekodierung der Zeilenadresse bestimmt Select-Leitung
- Komplette Zeile wird in den Zeilenpuffer geschrieben
- Dekodierung der Spaltenadresse bestimmt Datenwort

## Speicher Interlacing

- Speicheraufteilung in mehrere physische Bänder
- Adressen nicht kontinuierlich in den Bändern, sondern wechseln von Band zu Band
- nahezu gleichzeitiges Benutzen der Daten möglich (Daten-/Adressbus verhindern Parallelität)

## Burst Mode Blocktransfer

- Auslesen des kompletten Zeilenpuffers durch automatisches Inkrementieren der Spaltenadresse
- Prozessor gibt eine Adresse, Speicher liefert n Datenworte (Adr, Adr+1,..., Adr+n-1)
- falls folgende Datenworte genutzt werden, war für n Datenworte nur 1 Speicherzugriff (Zeit) nötig

## Typischer DRAM-Speicher

- Adressleitungen werden i.d.R. gemultiplext
- gleiche Adressleitungen werden einmal zur Auswahl der Zeile verwendet, dann zur Auswahl der Spalte
- Einsparung von Leitungen, gerade für große Speicher
- Steuerleitungen RAS/CAS codieren
- RAS (Row Address Strobe): Bei fallenden Flanke auf RAS ist anliegende Adresse Zeilenadresse
- CAS (Column Address Strobe): Bei fallenden Flanke auf CAS ist anliegende Adresse Spaltenadresse
- Zeilenadressdecoder liefert Select-Leitung für eine Zeile
- Komplette Zeile wird in einen Zwischenpuffer übernommen und zurückgeschrieben
- DRAM-Eigenschaften: Weniger Platzbedarf
- Nur 1 Transistor und 1 Kondensator pro Speicherzelle, statt 6 Transistoren bei SRAM
- Integrationsdichte Faktor 4 höher als bei SRAMs
- Aber: Langsamerer Zugriff, Insbes. Lesezugriff wegen Zwischenspeicherung und Auffrischung
- Auf DRAM-Zeile kann während Auffrischung nicht zugegriffen werden
- Hoher Energieverbrauch sowohl bei Aktivität als auch bei Inaktivität
- Ausgleich des Ladungsverlusts durch periodische Auffrischung

## Cache Speicher

- kleiner, schneller Prozessornaher Speicher
- CPU weiß nicht dass Cache zwischengeschaltet ist
- es wird immer zuerst im Cache nachgeschaut, zum Adressvergleich (kostet Zeit)
- 90% der Zeit verbringt ein Programm in 10% des Codes
- Puffer zwischen Hauptspeicher und Prozessor
- Cache besteht aus Cache Tabelle

**voll assoziativ** Adressvergl. der kompletten Adresse jede beliebige Adressfolge im Cache möglich kann zu langsamen Adressvergleich führen

**direct-mapped** Adressvergleich nur über Teiladresse - eine Tabelle im Cache

**mehr-wege-assoziativ** mehrere Adressvergleiche parallel

- Schreibstrategien

**Write Back** Cache sammelt Schreibvorgänge und aktualisiert nur im Cache. Nach einer entsprechenden Anweisung werden Daten in den Hauptspeicher kopiert (aktualisiert)

**Write Trough** Daten werden sowohl im Cache als auch im Hauptspeicher aktualisiert

- Speicherverwaltung mit memory management günstiger vor dem Cache
- Prüfung anhand der Adresse, ob benötigte Daten im Cache vorhanden sind (Treffer: cache hit)
- Falls nicht (cache miss): Zugriff auf den (Haupt-) Speicher, Eintrag in den Cache
- Such-Einheit im Cache: Cache-Zeile (cache line)
- Blockgröße ist Anzahl der Worte, die im Fall eines cache misses aus Speicher nachgeladen werden
- Verbindung Speicher ↔ Cache ist so entworfen, dass Speicher durch zusätzliches Lesen nicht langsamer wird
- Methoden dazu:
  - Schnelles Lesen aufeinanderfolgender Speicherzellen (Burst-Modus der Speicher)
  - Interleaving (mehrere Speicher ICs mit überlappenden Zugriffen)
  - Fließbandzugriff auf den Speicher (EDO-RAM, SDRAM)
  - Breite Speicher, die mehrere Worte parallel übertragen können
- Ersetzungs-Strategien

**Zufall** zu ersetzende Block zufällig ausgewählt

**FIFO** älteste Block ersetzt

**LRU** (least recently used ) Block ersetzt, auf den am längsten nicht mehr zugegriffen wurde

**LFU** (least frequently used ) am seltensten gelesene Block ersetzt

**CLOCK** alle Platzierungen im Kreis auf einem Ziffernblatt angeordnet. Ein Zeiger wird im Uhrzeigersinn weiterbewegt und zeigt den zu ersetzenden Eintrag an

Strategien zum Rückschreiben → (Haupt-) Speicher

- Write-Through (Durchschreiben):
- Jeder Schreibvorgang in den Cache führt zu einer unmittelbaren Aktualisierung des Speichers
- Speicher wird Engpass, es sei denn, der Anteil an Schreiboperationen ist klein oder der Speicher ist nur wenig langsamer als der Cache.
- Copy-Back, conflicting-use-write-back:
- Rückschreiben erfolgt erst, wenn Cache-Zeile bei Miss verdrängt wird
- Funktioniert auch bei großen Geschwindigkeitsunterschieden zwischen Cache und Speicher. Vorkehrungen erforderlich, damit keine veralteten Werte aus dem Speicher kopiert werden.

Trefferquote  $T = \frac{N_C}{N_G}$  mit  $N_G$  Gesamtzahl der Zugriffe auf Speicher und  $N_C$  Anzahl der Zugriffe mit Hit auf Cache

## Spezialrechner

### Einchiprechner

- geringer Stromverbrauch, Wärme
- relativ geringer Befehlsdurchsatz
- einfacher Befehlssatz
- komplexer Rechner auf einem Chip (Ram/Rom intern)
- an Problem angepasst
- so Leistungsfähig wie nötig
- Anwendung: einfache Steuer-/Regelungsaufgaben

### Digital-Signal-Prozessoren

- hohe Leistung, u.a. sich wiederholende, numerisch intensive Aufgaben

- pro Befehlszyklus kann man ausführen
  - mehrere ALU Funktionen (+Shifter)
  - eine/mehrere MAC-Operationen
  - ein/mehrere Speicherzugriffe
  - spezielle Unterstützung effiziente Schleifen
- Die Hardware enthält
  - Eine oder mehrere MAC-Einheiten
  - On-Chip- und Off-Chip-Speicher mit mehreren Ports
  - Mehrere On-Chip-Busse
  - Adressgenerierungseinheit
  - größerer ROM, RAM (mit Stack), Cache
  - externes Interface für Speicher und E/A
  - DMA-Coprozessor
  - mehrere universelle parallele E/A-Ports
  - AD-Wandler meist nicht on-chip
- DSP's benutzung häufig VLIW
- Anwendung: Signal-/Bildverarbeitung

## Pipeline Prozessoren

- Aufteilung eines Befehls in Teilbefehle
- parallele Abarbeitung verschiedener Teilbefehle von unterschiedlichen Befehlen möglich
- Probleme
  - bedingte Sprünge (unvorhergesehen)
  - LSG: Pipeline um 2 Schritte verzögern
  - LSG: Sprungzielspekulation
  - Datenabhängigkeit
  - LSG: Pipeline um 2 Schritte verzögern
  - LSG: Out-of-Order Prinzip nutzen
  - LSG: Forwarding Hardware
- Superpipelining - noch mehr Befehlsaufteilung

## Skalare Prozessoren

- Mikroarchitektur
- Befehlszuordnung und Konfliktvermeidung geschieht durch Hardware
- Speicherzugriffe automatisch von Load/Store Einheit
- ein Befehlsstrom mit einfachem Befehl an Ausführungseinheit
- bessere Reaktion auf Laufzeitergebnisse
- Spekulation möglich
- Superskalar ( $\geq 2$  EX Einheiten)
  - Befehle in Befehlsfenster gespeichert
  - Zuordnungseinheit wählt Befehl aus, die parallel verarbeitet werden können

## Multiprozessorarchitekturen

Klassifikation nach Flynn (Instruction/Data)

	Ein Datenstrom	mehrere Datenströme
ein Befehlsstrom	SISD	SIMD
mehrere Bs	MISD	MIMD

## Kopplung

**enge Kopplung** (shared memory)

- parallelzugriff in Datenbreite des Prozessors
- schneller Datenaustausch (Proz-Speicher-Zugriffszeit)
- neu lokal benachbarte Rechner
- aus Sicht des Prozessors gemeinsamer Speicher
- Arbiters - Zuteiler für Busanforderungen

**lose Kopplung** (distributed memory)

- meist seriell 1 bit breit

- langsamer da über Kommunikations-schnittstelle
- auch global verteilte Rechner
- verschiedene Speicher dem Prozessor bewusst

**Kopplung** verändern

- Wartezeit auf Speicher
- Kommunikationsaufwand
- Kommunikationsfähigkeit
- optimale Prozessor-/Speicher-/Kommunikationspfad anzahl  $< \infty$

## Out-of-Order Architektur

- statt Pipeline bei Datenabhängigen Befehlen um 2 Schritte verzögern, datenunabhängige Befehle einschieben
- möglichst ständige Auslastung aller EX Einheiten

Cache(daten)-Kohärenz

- Kohärenz: welcher Wert wird beim Lesen abgeliefert
- Bezug auf Lesen und Schreiben ein- und derselben Speicherzelle
- Definition: Ein Speichersystem heißt kohärent, wenn
  - geschriebene Werte werden wieder gelesen
  - Schreibvorgänge derselben Zelle serialisiert

- Lösung des I/O-Problems: Zuordnung einer I/O-Einheit zu jedem Prozessor
- Hardware-Lösung: Aufwändig, schlechte Lokalität der Daten
- Gemeinsamer Cache für alle Prozessoren: Hoher Hardware-Aufwand, geringe Effizienz
- Unterscheidung in cacheable/non-cacheable Daten: Hoher Aufwand

Snooping-Protokolle

- Die Caches aller Prozessoren beobachten alle Datenübertragungen von jedem Cache zum Hauptspeicher.
- Voraussetzung: broadcastfähiges Verbindungsnetzwerk
- Write Invalidate: Das Verändern eines Blocks im Speicher führt zur Invalidierung aller Cache-Kopien mit der gleichen Adresse
- Write Update / Write Broadcast: Das Verändern eines Blocks im Speicher führt zur Modifikation aller anderen Cache-Blöcke mit der gleichen Adresse

Copy-Back

- Copy-Back Caches führen zur temp. Inkonsistenz
- Lösung: exklusives Eigentumskonzept durch Zustandsgraph pro Cache-Block
- MESI (Modified, Exclusive, Shared, Invalid)
- Mischung zwischen Write-Through und Copy-Back

## MESI Vier Zustände

- (exclusive) Modified: Cache-Block wurde lokal geändert, die Kopie im Hauptspeicher ist ungültig. Will ein anderer Prozessor dieses Datum im Hauptspeicher lesen, so muss der Cache-Block erst in den Hauptspeicher zurückgeschrieben werden.
- Exclusive (unmodified): Dieser Cache ist der einzige, der den Cache-Block enthält, Wert im Hauptspeicher ist gültig. Liest ein anderer Prozessor dieses Datum im Hauptspeicher, so muss die Zeile als shared markiert werden. Wird das Datum im Hauptspeicher verändert, ist der Cache-Block auf invalid zu setzen.
- Shared (unmodified): Mehrere Caches (mind. 2) enthalten dieses Datum. Da alle bisher nur gelesen haben, ist das Datum im Hauptspeicher gültig.

Schreibzugriffe auf einen shared Cache-Block müssen immer zu einer Bus-Operation führen, damit die Cache-Blocks der anderen Caches auf invalid gesetzt werden können.

- Invalid: Cache-Block ist noch gar nicht geladen bzw. veraltet/ungültig

#### Bus-Operationen

- Bus Read: wenn ein Prozessor Wert eines Speicherblocks lesen will
- Bus Read Exclusive: wenn ein Prozessor Wert eines Speicherblocks überschreiben will
- Flush: wenn ein Prozessor  $P_i$  einen Speicherblock alleinig in seinem Cache hat, ein anderer Prozessor  $P_j$  aber lesend oder schreibend auf diesen Block zugreift. Bei einer Flush-Operation legt  $P_i$  ebenfalls das Datum des Speicherblocks auf den Bus.

#### Steuersignale

- Invalidate-Signal: Invalidieren des Blocks in den Caches anderer Prozessoren
- Shared-Signal: Signalisierung, ob ein zu ladendes Datum bereits als Kopie im Cache vorhanden ist
- Retry-Signal: Aufforderung von Prozessor  $P_i$  an Prozessor  $P_j$ , das Laden eines Datums vom Hauptspeicher abubrechen, da der Hauptspeicher noch ein altes, ungültiges Datum besitzt und vorher aktualisiert werden muss. Das Laden durch  $P_j$  kann danach wiederholt werden.

#### Bewertung von Snooping-Protokollen

- Leichte Implementierbarkeit bei Bus-basierten Shared Memory Systemen
- Snooping skaliert bei Bussen jedoch nicht
- Bei vielen beteiligten Prozessoren sinkt die effektive Bandbreite des Busses, da überproportional viele Invalidierungsnachrichten per Broadcast über den Bus gehen

- Punkt-zu-Punkt Netzwerke sind skalierbar, jedoch ist die Implementierung von Broadcasts hier aufwändig
- Für Snooping-Protokolle daher oft ungeeignet

#### Directory-Protokolle

- Nur wenige Prozessoren teilen sich die gleichen Daten in vielen Anwendungen
- Directory-Protokolle nutzen Lokalitätsinformationen, um die Anzahl an Invalidierungsnachrichten zu minimieren
- Nachrichten gehen nur an Prozessoren, die eine Kopie des Cache-Blocks besitzen
- Directory-Protokolle skalieren daher auch für Netze ohne Broadcast-Fähigkeit
- Presence Flag Vector: Im Hauptspeicher abgelegter Bit-Vektor für jeden einzelnen Speicherblock (1 Bit pro Prozessor/Cache + Statusbits (dirty, modified))
- Problem: Wachstum des Speicherbedarfs linear mit Anzahl der Prozessoren