

# Einführung in die Technische Informatik

## General Purpose Prozessoren

Zellescher Weg 12  
Willers-Bau A 205  
Tel. +49 351 - 463 - 35450

Nöthnitzer Straße 46  
Raum 1044  
Tel. +49 351 - 463 - 38246

Wolfgang E. Nagel (wolfgang.nagel@tu-dresden.de)



## Einführung in die Technische Informatik

Modulnummer:	INF-BAS5
Modulname:	Basismodul Technische Informatik
Verantwortlicher	Prof. Dr. Wolfgang E. Nagel für Arbeitsgebiet Parallelverarbeitung
Lehr- und Lernformen:	4 SWS Vorlesung 2 SWS Übung 2 SWS Praktikum
Verwendbarkeit:	Wahlpflichtmodul in - Master-Studiengang Informatik - Diplomstudiengang Informatik - Diplomstudiengang Informationssystemtechnik
Leistungspunkte:	12
Noten:	Mündliche Prüfung (Prüfungsvorleistung ist Protokollsammlung)
Moduldauer:	1 Semester

# Einführung in die Technische Informatik

Vorlesung: Fr.: 5. DS, INF/E006  
Fr.: 6. DS, INF/E006

Übung und Praktikum: werden je nach Themenbereich aufgeteilt  
Do.: 4.DS und 5.DS INF/E069

**Beginn von Übung/Praktikum: Do.: 16. Januar 2014, 13:00 Uhr, INF/E069**

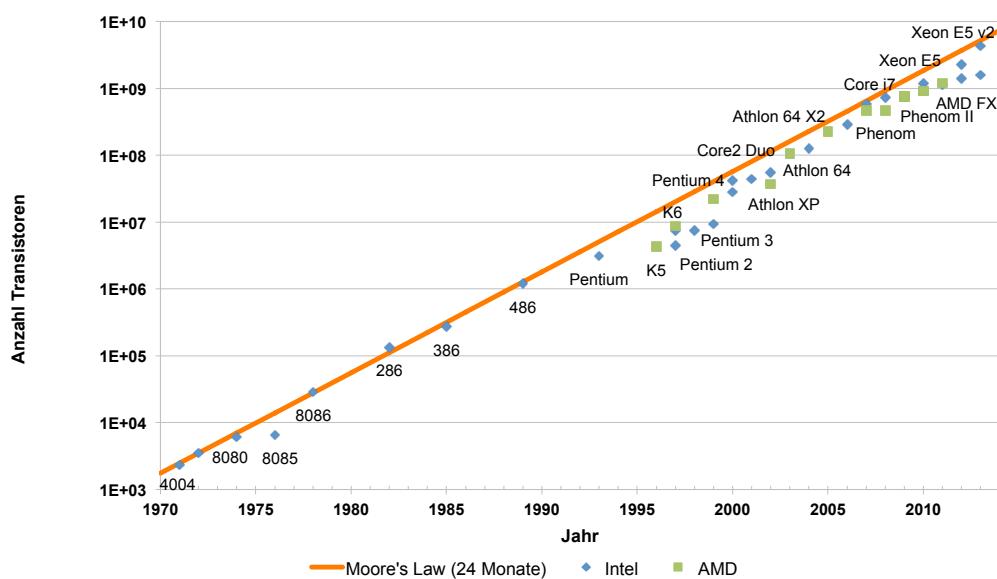
Materialien im OPAL verfügbar:

<https://bildungsportal.sachsen.de/opal/url/RepositoryEntry/5382635529>

Themenbereiche:

- General Purpose Prozessoren
- Hardware-Beschleuniger
- Parallelrechner
- Programmierung paralleler Rechnerarchitekturen
- Leistungsbewertung und Optimierung von Parallelrechnern

## Motivation: Moore's Law



- Herausforderung: steigende Transistorzahl zur Erhöhung der Rechenleistung nutzen

## Inhalt

---

- Steigerung der Rechenleistung bei sequenzieller Verarbeitung (SISD)
  - Pipelining und Superskalare Ausführung
  - Cache
  - Out-of-Order Execution
  - Sprungvorhersage
- Parallelverarbeitung
  - Befehlssatz-Erweiterungen (SIMD)
  - Multicore Prozessoren (MIMD)
  - Energie-Effizienz

## Flynn'sche Klassifizierung (siehe RAI1-Vorlesung)

---

- **SISD** - single instruction stream, single data stream
- **MISD** - multiple instruction streams, single data stream
- **SIMD** - single instruction stream, multiple data streams
- **MIMD** - multiple instruction streams, multiple data streams

---

# PIPELINING/SUPERSKALAR

## Pipelining

- Ausführung der Befehle in mehrere Phasen unterteilt:
  - Instruction Fetch (IF): nächsten Befehl laden (mittels program counter)
  - Decode (D): aus Opcode Steuersignale für die ALU generieren
  - Execute (EX): Berechnung durchführen
  - Memory (M): Speicherzugriff durchführen (optional)
  - Write Back (WB): Ergebnisse in Registerfile übertragen
- Mehrere Befehle gleichzeitig in der Pipeline
  - Start jeweils um einen Takt versetzt → 1 Befehl in jeder Phase
  - Ein Befehl pro Takt wird fertiggestellt (im eingelaufenen Zustand)

Takt	1	2	3	4	5	6	7
Inst 1	IF	D	EX	M	WB		
Inst 2		IF	D	EX	M	WB	
Inst 3			IF	D	EX	M	WB

Basis Pipeline

## Pipelining

---

- Phase mit längster Bearbeitungszeit bestimmt maximale Taktfrequenz
  - Häufig weitere Unterteilung der einzelnen Phasen
    - Mehrstufiges Dekodieren (Decode, Operand Fetch)
    - Berechnung (EX) in mehreren Schritten

Takt	1	2	3	4	5	6	7	8	9	10
Inst 1	IF	D	OF	EX1	EX2	EX3	M	WB		
Inst 2		IF	D	OF	EX1	EX2	EX3	M	WB	
Inst 3			IF	D	OF	EX1	EX2	EX3	M	WB

- Variable Bearbeitungszeiten schlecht für effektives Pipelining
  - RISC: ähnliche Bearbeitungszeiten der Befehle
  - CISC: Decoder zerlegen komplexe Befehle in mehrere RISC-artige („Microops“)

## Superskalare Befehlausführung

---

- Verdopplung der Verarbeitungs-Ressourcen
  - Mehrere Befehle in jeder Pipeline-Stufe
  - In n-fach superskalarer Architekturen werden bis zu n Befehle pro Takt fertig gestellt:

Takt	1	2	3	4	5	6	7
Inst 1	IF	D	EX	M	WB		
Inst 2	IF	D	EX	M	WB		
Inst 3		IF	D	EX	M	WB	
Inst 4		IF	D	EX	M	WB	
Inst 5			IF	D	EX	M	WB
Inst 6			IF	D	EX	M	WB

2-fach superskalare Pipeline

- Selten voll ausgelastet wegen Konflikten (**Hazards**) zwischen Instruktionen

## Strukturelle Hazards

- Verschiedene Befehle benötigen die selben Ressourcen
  - Instruction Fetch und Memory-Phase nutzen Speicherinterface

Takt	1	2	3	4	5	6	7	8	9	10
Inst 1	IF	D	EX	M	WB					
Inst 2		IF	D	EX	M	WB				
Inst 3			IF	D	EX	M	WB			
Inst 4				stall	IF	D	EX	M	WB	
Inst 5						IF	D	EX	M	WB

- Speicherzugriffe benötigen mehrere Takte

Takt	1	2	3	4	5	6	7	8	9	10
Inst 1	IF	D	EX	M	M	M	M	M	WB	
Inst 2		IF	D	EX	stall				M	WB

- Folgebefehle müssen warten bis nächste Pipeline Stufe frei wird
- „mehrere“ sind in aktuellen Systemen 200 - 300 Takte

## Daten-Abhängigkeiten

- RAW – Read After Write:

$$R8 = R8 + R9$$

$$R10 = R10 * R8$$

$$R7 = R7 + R9$$

Takt	1	2	3	4	5	6	7	8	
Inst 1	IF	D	EX	M	WB				
Inst 2		IF	D	stall		EX	M	WB	
Inst 3			IF	stall		D	EX	M	WB

- Zweiter Befehl muss auf Resultate des ersten Befehls warten
- Folgebefehle werden ebenfalls verzögert
  - Inst 2 bleibt in Decode Phase bis Operanden verfügbar sind
  - Inst 3 kann Decode erst beginnen wenn Inst 2 in EX-Phase ist

## Control Flow Hazards

- Durch Verzweigungen im Programmablauf (z.B. durch if-then-else) können Folgebefehle übersprungen werden

Takt	1	2	3	4	5	6	7	8	9	
Inst1	IF	D1	D2	EX	WB					
Inst2	IF	D1	D2	EX	WB					
jnz Inst8		IF	D1	D2	EX	WB				
Inst4		IF	D1	D2	EX	WB				
Inst5		IF	D1	D2	EX	WB				
Inst6		IF	D1	D2	EX	WB				
Inst7		IF	D1	D2	EX	WB				
Inst8		IF	D1	D2	EX	WB				
Inst9		IF	D1	D2	EX	WB				
Inst10					IF	D1	D2	EX	WB	
Inst8					IF	D1	D2	EX	WB	
Inst9					IF	D1	D2	EX	WB	

- Ob Sprung ausgeführt wird, entscheidet sich erst in Execution-Phase (abhängig vom Ergebnis von Inst 2)

## Erhöhung der Rechenleistung

- Caches
  - Schnelle Zwischenspeicher im Prozessor
  - Reduzieren Wartezeiten bei Speicherzugriffen
  - Getrennte Befehls- und Daten-Caches um Konflikte zwischen Instruction Fetch und Load-Store-Anweisungen zu vermeiden
- Out-of-order Execution
  - Ausführung nachfolgender Befehle, falls ein Befehl noch auf Berechnung seiner Operanden warten muss
  - Reduziert Wartezeiten bei Datenabhängigkeiten zwischen Befehlen
- Sprungvorhersage
  - Spekulative Fortsetzung der Programmausführung nach Verzweigungen an wahrscheinlicherer Ziel-Adresse
  - Verringert Häufigkeit von Control Flow Hazards

---

# CACHE

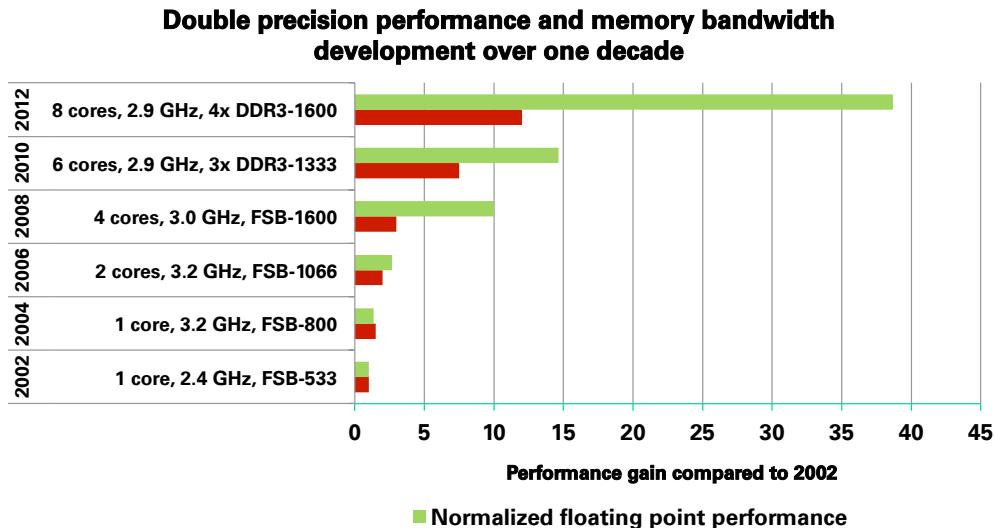
---

## Memory Gap

- Hauptspeicher:
  - Viel Speicherplatz (mehrere GB)
  - Begrenzte Bandbreite (DDR3-1600: 12,8 GB/s pro Speicher-Kanal)
- Prozessor:
  - Rechenwerke verarbeiten mehrere Befehle pro Takt
  - Double Precision Gleitkomma-Operationen benötigen 24 Byte pro Befehl  
→ 1 GFLOP/s benötigt 24 GB/s
- Beispiel: 4-Kern Prozessor (Intel Sandy Bridge):
  - 8 FLOPs/Takt @ 3+ GHz  
→ ca. 25 GFLOP/s pro Kern  
→ ca. 100 GFLOP/s maximal Rechenleistung  
→ erforderliche Speicherbandbreite: ca. 2,4 TB/s
  - 2-Kanal DDR3-1600: maximal 25,6 GB/s Speicherbandbreite

**⇒ Speicherbandbreite reicht nur für ca. 1% der Rechenleistung**

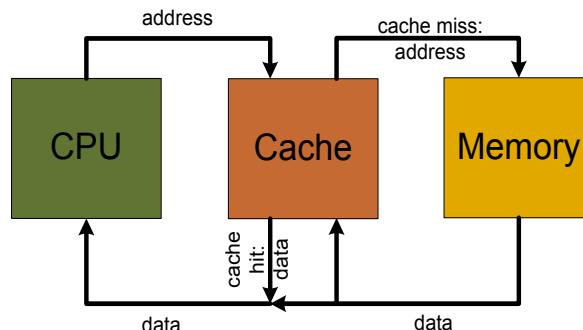
## Memory Gap vergrößert sich stetig



- Rechenleistung steigt schneller an als die Speicherbandbreite  
→ Immer mehr Anwendungen sind durch den Speicher limitiert

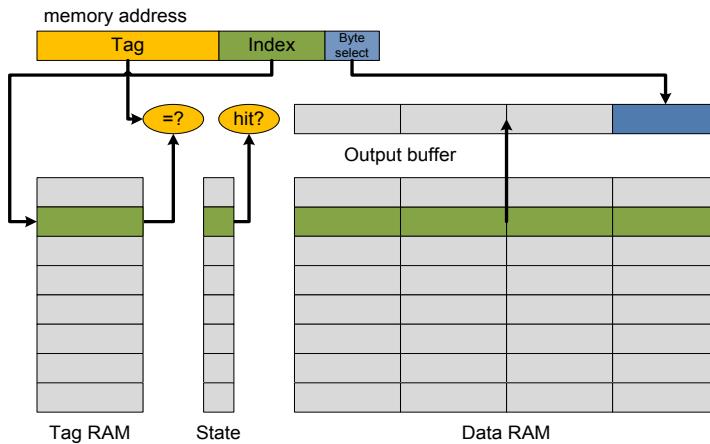
## Cache Konzept

- Schnelle Zwischenspeicher für häufig genutzte Daten
  - höhere Bandbreite und geringere Latenz als Hauptspeicher



- Speicherinhalte werden häufig mehrfach benutzt
  - Daten: Variablen, Konstanten, Datenstrukturen, etc.
  - Code: Schleifen und Funktionen

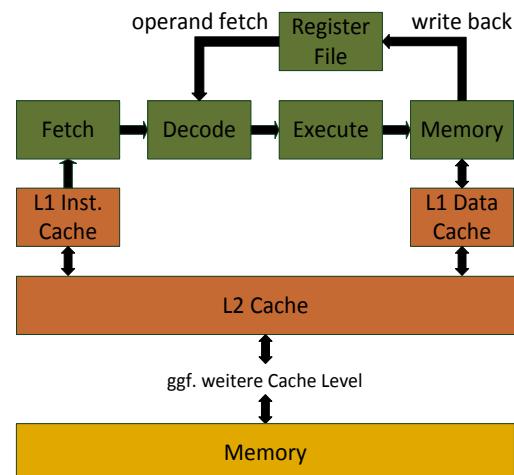
# Inhaltsadressierter Speicher



- Data RAM: Kopien des Hauptspeicher-Inhalts  
Blöcke (Cachelines) mit fester Größe
- Tag RAM: Speicheradressen der gespeicherten Blöcke
- Status: gibt an ob der Eintrag gültig ist

## Speicherhierarchie

- Meist mehrere Cache Level
  - Typisch sind z.Zt. 3 Cache Level
  - First-, Mid- und Last Level Cache
- Kleiner, schneller Level 1 Cache
  - Häufig mehrere Lese- oder Schreib-Operationen pro Takt aus L1D
  - geringe Latenz (~3 Takte)
  - i.d.R. unterteilt in Befehlscache und Datencache
    - Jeweils nur wenige KiB
    - nah an den entsprechenden Verarbeitungseinheiten



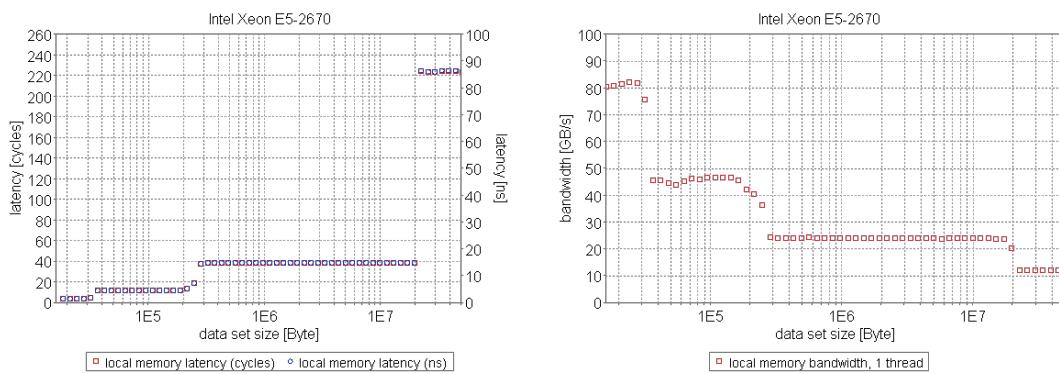
- Weitere Cache-Level
  - Höhere Kapazität: mehrere MiB
  - Geringere Bandbreite, höhere Latenz

## Entwicklung der Cache-Hierarchie

Prozessor	Jahr	L1	L2	L3
Intel i486	1989	8 KiB	-	-
Intel Pentium (P5)	1993	2x 8 KiB	-	-
Intel Pentium Pro (P6)	1995	2x 8 KiB	256 KiB	-
Intel Pentium 2 (Klamath)	1997	2x 16 KiB	512 KiB	-
Intel Pentium 3 (Katmai)	1999	2x 16 KiB	512 KiB	-
Intel Pentium M (Banias)	2003	2x 32 KiB	1 MiB	-
Intel Core2 Duo (Conroe)	2006	2x 32 KiB	4 MiB	-
1st gen Intel Core i7 (Nehalem)	2008	2x 32 KiB	256 KiB	8 MiB
2nd gen Intel Core i7 (Sandy Bridge)	2011	2x 32 KiB	256 KiB	8 MiB
4th gen Intel Core i7 (Haswell)	2013	2x 32 KiB	256 KiB	8 MiB

- L2 Cache zunächst als zusätzliche Chips
- Seit späteren Pentium 3 Modellen im Prozessor integriert
- L3 Größe seit längerem konstant, da steigende Transistorzahl derzeit bevorzugt für bessere integrierte Grafikeinheit genutzt wird

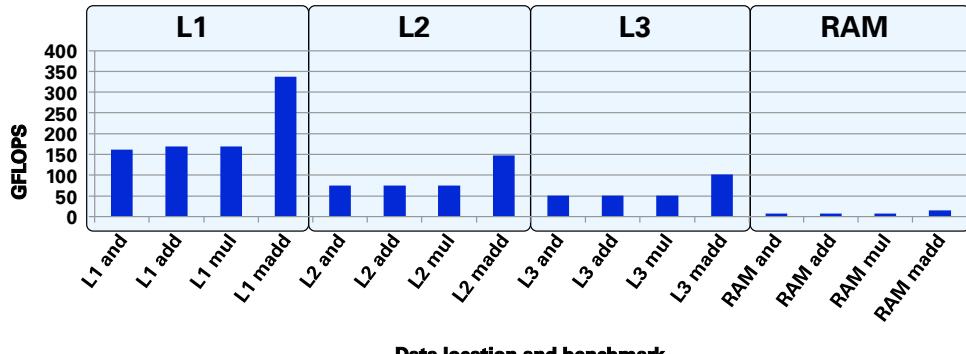
## Beispiel: Intel Xeon E5-2670



- 32 KiB L1: 4 Takte Latenz; 83,2 GB/s (32 Byte/Takt)
- 256 KiB L2: 12 Takte Latenz; 46,8 GB/s
- 20 MiB L3: 39 Takte Latenz; 24,0 GB/s
- 64 GiB RAM: 224 Takte Latenz; 12,1 GB/s

# Einfluss der Lokalität auf Rechenleistung

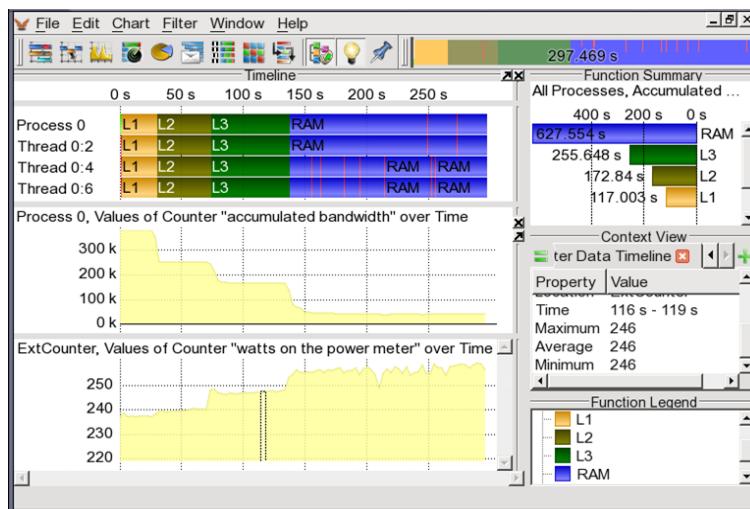
- 2x Xeon E5 2,7 GHz: 345 GFLOPS maximale Rechenleistung



Data location and benchmark

- Benchmarks laden Daten aus den Ebenen der Speicherhierarchie und:
  - and/add/mul: führen eine Operation pro gelesenem Operand aus
  - madd: führen zwei Operationen pro gelesenem Operand aus
- Sogar wenn die Daten im L1 liegen, sind zwei Operationen pro Operand erforderlich um die volle Rechenleistung abzurufen

## Energieverbrauch von Speicherzugriffen [I]



- Je mehr Ebenen der Speicherhierarchie genutzt werden umso höher ist die Leistungsaufnahme
- Gleichzeitig sinkt die verfügbare Speicherbandbreite

## Energieverbrauch von Speicherzugriffen [II]

Operation	$E_{trans}$ AMD Opteron 2435	$E_{trans}$ Intel Xeon X5670
Zugriff in L1 Cache	105 pJ/Byte	64 pJ/Byte
Zugriff in L2 Cache	357 pJ/Byte	121 pJ/Byte
Zugriff in L3 Cache	654 pJ/Byte	254 pJ/Byte
Hauptspeicher-Zugriff	3590 pJ/Byte	1250 pJ/Byte

- Energieverbrauch für Laden von Daten aus dem Hauptspeicher
  - Faktor >20 höher als L1 Zugriff
  - Faktor 10 bzw. 5 höher als L2 bzw. L3
- Gleitkomma-Operation (Intel Xeon X5670):
  - Berechnung ca. 400 pJ pro Operation (double precision)
  - 24 Byte Datentransfer: 1500 pJ (L1) – 30000 pJ (Hauptspeicher)

⇒ **Datentransfers sind der limitierende Faktor sowohl bei der Rechenleistung als auch bei der Energie-Effizienz**

## Auswirkung von Cache auf Pipelining

- Getrennte L1 Caches lösen Ressourcen-Konflikt zwischen Instruction Fetch und Memory Phase
- Wartezeiten bei Speicherzugriffen werden deutlich reduziert
- Aber:
  - nur der L1 Cache ist schnell genug um Pipelining nicht zu stark zu stören
  - L1 Cache meist sehr klein
  - Latenzen von L2 und L3 Cache nicht vernachlässigbar
- → **Out-of-Order Execution um verbleibende Wartezeiten für andere Berechnungen zu nutzen**

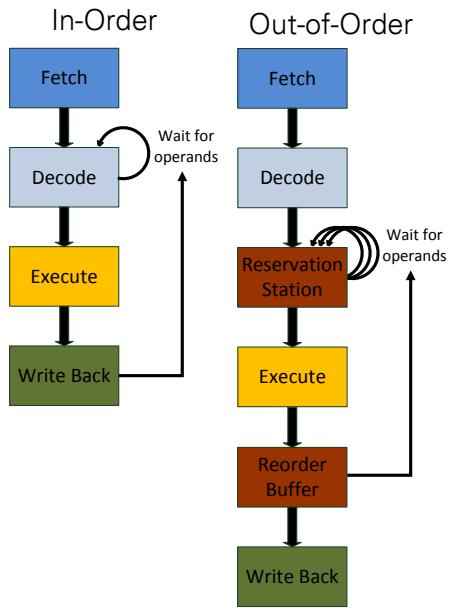
---

# OUT-OF-ORDER EXECUTION

## Out-of-Order Execution

- Abarbeitung der Befehle abweichend von der Reihenfolge im Programm
  - Verdecken der Latenz beim Speicherzugriff
  - Vermeidung von Leerlauf durch Abhängigkeiten
  - → Bessere Auslastung der Rechenwerke
  
- Nach außen sichtbares Ergebnis muss der Abarbeitung in Programmreihenfolge entsprechen
  - Zwischenspeicherung der Ergebnisse in zusätzlichen Registern
  - Architekturregister erst aktualisieren wenn alle vorhergehenden Befehle erfolgreich abgeschlossen sind

# Out-of-Order Execution



## ● In-Order

- Ein Befehl wartet auf seine Operanden
- Fetch und Decode werden angehalten

## ● Out-of-Order

- Reservation Station (RS)
  - Mehrere Befehle warten auf Operanden
  - Decode läuft weiter bis Puffer voll ist
- Execute
  - Abarbeitung der Befehle sobald Operanden verfügbar sind
  - Es werden bevorzugt die ältesten Befehle ausgeführt
- Reorder Buffer (ROB)
  - Speichert Ergebnisse
  - Write Back in Programmreihenfolge (in-order completion)

# Out-of-Order Execution: Beispiel

## ● In-Order:

Takt	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R8= R8+R9	IF	D	EX 1	EX 2	EX 3	WB									
R10= R10-R8		IF	D	stall			EX 1	EX 2	EX 3	WB					
R7= R7+R9			IF	stall			D	EX 1	EX 2	EX 3	WB				
R11= R11-R7				stall			IF	D	stall			EX 1	EX 2	EX 3	WB

## ● Out-of-Order:

Takt	1	2	3	4	5	6	7	8	9	10	11	12	13	14
R8= R8+R9	IF	D	RS	EX 1	EX 2	EX 3	RB	WB						
R10= R10-R8		IF	D	RS	RS-wait			EX 1	EX 2	EX 3	RB	WB		
R7= R7+R9			IF	D	RS	EX 1	EX 2	EX 3	RB	ROB-wait			WB	
R11= R11-R7				IF	D	RS	RS-wait			EX 1	EX 2	EX 3	RB	WB

## Daten-Abhangigkeiten

---

- Out-of-Order Execution erzeugt neue Abhangigkeiten
  - WAR – Write After Read
$$R9 = R9 + R8$$
$$R8 = R10 * R11$$
    - R8 darf erst berschrieben werden wenn erster Befehl gelesen hat
  - WAW – Write After Write
$$R8 = R9 + R10$$
$$R8 = R11 + R12$$
    - Sicherstellen, dass letztes Resultat sichtbar wird
- Keine echten Datenabhangigkeiten
  - Ressourcen-Konflikt zwischen genutzten Registern
  - → Anzahl der Register muss erhoht werden

## Register Renaming

---

- Anzahl adressierbarer Register bleibt gleich
  - keine nderung am Befehlssatz
- Intern mehr Register als ber Befehlssatz zugreifbar
  - Ergebnisse werden in temporren Registern zwischengespeichert
  - Lst Ressourcen-Konflikte bzgl. der architekturellen Register

### Ohne Renaming:

$$R10 = (ptr+8)$$

$$R8 = R8 + R10$$

$$R10 = (ptr+16)$$

$$R9 = R9 + R10$$

### Mit Renaming:

$$R10 = (ptr+8)$$

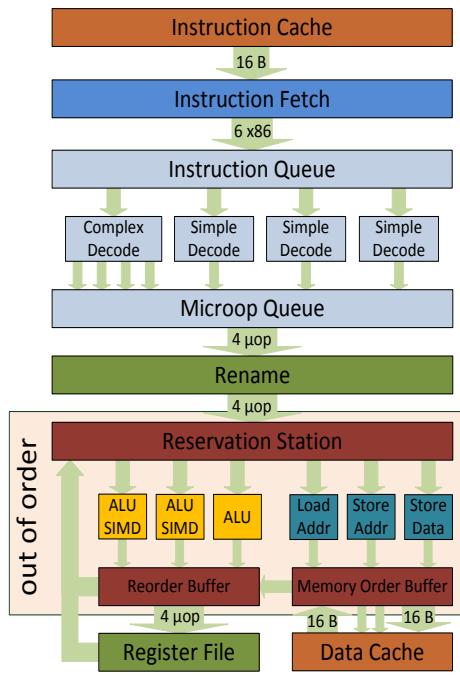
$$R8 = R8 + R10$$

$$R10' = (ptr+16)$$

$$R9 = R9 + R10'$$

- Ermglicht effektiveres Umsortieren, z.B. im Falle eines Cache-Misses beim Lesen von ptr+8

## Beispiel: Intel Core Mikroarchitektur



- 16 Byte Fetch / Takt
- 4x x86-Decode / Takt
  - 3 Decoder für einfache x86-Befehle, die nur eine Microop erfordern
  - 1 Decoder für komplexe x86-Befehle, die mehrere Microops erzeugen
- Registerumbenennung für 4 Microops / Takt
- Bis zu 32 Microops warten in RS
- Verarbeitung von bis zu 6 Microops / Takt
  - 3 Microops für Berechnungen
  - 3 Microops für Speicherzugriffe
- ROB speichert bis zu 96 Ergebnisse
- 4 Microops können pro Takt abgeschlossen werden (in Program-Reihenfolge)
- MOB sorgt für in-order stores

## Entwicklung der out-of-order Ressourcen

Prozessor	Jahr	ROB Einträge	RS Einträge	Microops/Takt
Pentium Pro (P6)	1995	40	20	5
Pentium 2 (Klamath)	1997	40	20	5
Pentium 3 (Katmai)	1999	40	20	5
Pentium M (Banias)	2003	40	20	5
Core2 Duo (Conroe)	2006	96	32	6
1st gen Core i7 (Nehalem)	2008	128	36	6
2nd gen Core i7 (Sandy Bridge)	2011	168	54	6
4th gen Core i7 (Haswell)	2013	192	60	8

- Bis zur Einführung der Core Mikroarchitektur Leistungssteigerung im Wesentlichen über höheren Takt
- Danach Leistungssteigerung über Erhöhung der IPC

## Implementierungen

---

- Prinzip in allen Out-of-Order Prozessoren identisch
- Unterschiede im Detail:
  - Reservation Station
    - Eine RS für alle Verarbeitungseinheiten
    - Getrennte RS für Integer und FPU
    - Eine RS für jede Verarbeitungseinheit
  - Reorder Buffer / Register Renaming
    - Getrennte Register Sätze für Renaming und architekturelle Register
    - Ein „Physical Register File“ für Beides
- i.d.R. mehr Ausführungseinheiten als Befehle dekodiert werden können
  - Hohe IPC für unterschiedliche Kombinationen von Befehlen
  - Einzelne Pipelines selten voll ausgelastet
- **Herausforderung: Umsortieren über bedingte Sprünge hinweg**  
→ **Sprungvorhersage**

---

# SPRUNGVORHERSAGE

## Motivation

---

- im Mittel jede 5. - 7. Operation ein Sprungbefehl
- am häufigsten bedingte Verzweigungen
  - if-then-else Konstrukte
  - Schleifen mit dynamischer Abbruchbedingung
- Unterbrechung des sequenziellen Programmablaufs
  - Ohne Vorhersage wird auf Ergebnis des Sprungbefehls gewartet

Takt	1	2	3	4	5	6	7	8	9	10
<b>inst_x</b>	IF	D1	D2	EX	WB					
<b>jnz Inst_y</b>		IF	D1	D2	EX	WB				
<b>inst_y   inst_x+1</b>			stall			IF	D1	D2	EX	WB

- Verhindert effektives Pipelining und Out-of-Order Execution  
→ schlechte Auslastung der Rechenwerke

## Gründe für Verzögerungen

---

- Feststellen ob es sich um einen Sprungbefehl handelt
  - → Decodierung des Befehls abwarten
- ggf. Bedingung auswerten.
  - Abhängig von vorhergehendem Befehl
  - → warten bis Ergebnis vorliegt
- Sprungziel (Adresse des Folgebefehls) bestimmen
  - Abhängig von Adressierungsart
  - → Berechnen oder Laden

## Sprungvorhersage

---

- Ziel: Verzögerungen bei Verzweigungen minimieren
  - 1) Verzweigungen schnell erkennen:
    - Befehlsstrom frühzeitig auf entsprechende Opcodes untersuchen
  - 2) Vorhersage ob Sprung ausgeführt wird:
    - Statisch anhand fester Vorhersage für verschiedene Arten von Verzweigungen
    - Dynamisch auf Grundlage des bisherigen Verhaltens
  - 3) Bestimmung der Zieladresse beschleunigen
    - Speicherung von früheren Sprungzielen
- Prinzip: spekulative Fortsetzung der Befehlsabarbeitung
  - Ausführung des wahrscheinlichsten Pfades
  - Verwerfen der Ergebnisse bei falscher Vorhersage

## Spekulative Fortsetzung der Befehlsabarbeitung

---

- Korrekte Vorhersage: Keine Unterbrechung in der Verarbeitung

Takt	1	2	3	4	5	6	7	8	9	10
inst_1	IF	D1	D2	EX	WB					
jnz inst_8		IF	D1	D2	EX	WB				
inst_3			IF	D1	D2	EX	WB			
inst_4				IF	D1	D2	EX	WB		
inst_5					IF	D1	D2	EX	WB	
inst_6						IF	D1	D2	EX	WB

- Falsche Vorhersage: Verwerfen unnötig berechneter Ergebnisse

Takt	1	2	3	4	5	6	7	8	9	10
inst_1	IF	D1	D2	EX	WB					
jnz inst_8		IF	D1	D2	EX	WB				
inst_3			IF	D1	D2	EX	WB			
inst_4				IF	D1	D2	EX	WB		
inst_5					IF	D1	D2	EX	WB	
inst_8						IF	D1	D2	EX	WB

- Resultate spekulativer Befehle dürfen nicht sichtbar werden

## Sprungvorhersage

---

- Anforderungen:
  - Hoher Anteil richtiger Vorhersagen
  - oft erneute Vorhersage nötig, bevor tatsächlicher Ausgang eines früheren Sprungs bekannt ist
  - schnelles Rückrollen der Befehlsabarbeitung bei Fehlspkulationen
- Implementierung
  - Zusätzliche Logik im Front-End, die Sprünge erkennt und Adresse vom wahrscheinlichsten Folgebefehl liefert
    - Fortsetzung von Fetch und Decode sobald Vorhersage vorliegt
    - Sprungbefehl normal ausführen und mit Vorhersage vergleichen
  - Nutzung der Ressourcen der Out-of-Order Execution
    - Reservation Station verdeckt kurze Lücken in Fetch und Decode
    - Reorder Buffer erlaubt Verwerfen von unnötigen Ergebnissen

## Statische Vorhersage

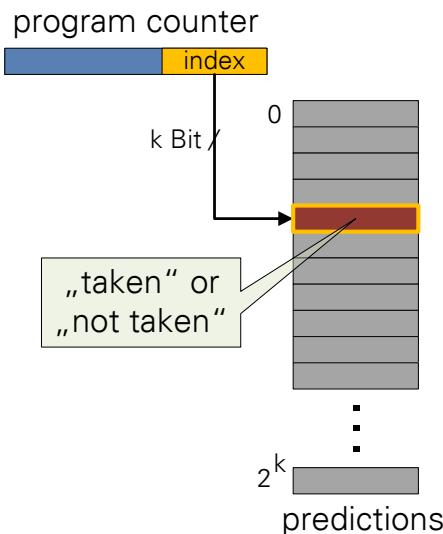
---

- Entscheidung aufgrund statistischer Werte
  - etwa 2/3 aller Sprünge verzweigen tatsächlich
    - einfache statische Vorhersage nimmt an, dass alle Sprünge ausgeführt werden
    - → ca. 67% Trefferquote
  - Vorwärtssprünge gleichverteilt, Rückwärtssprünge werden fast immer ausgeführt
    - Optimierung:
      - Vorwärtssprünge: predict not taken
      - Rückwärtssprünge: predict taken
    - Gleiche Trefferquote, weniger L1 I-Cache und TLB Misses
- Compilerbasierte Vorhersage
  - Im Opcode codierte Hinweise zur Sprungwahrscheinlichkeit
  - Basierend auf automatischer Analyse oder Annotation durch Programmierer
    - z.B.: if (\_\_builtin\_expect (<condition>, <0|1>)) foo()

## Dynamische Vorhersage

- Dynamische Verfahren sammeln zur Laufzeit Informationen über Sprungverhalten
  - Sprungverlaufstabelle
    - liefert Vorhersage, ob gesprungen wird
    - indexiert durch Teil der Befehlsadresse
  - Sprungzieladresscache
    - speichert Sprungziele
    - meist als satz-assoziativer Cache realisiert
- i.d.R. höhere Trefferquote als statische Verfahren

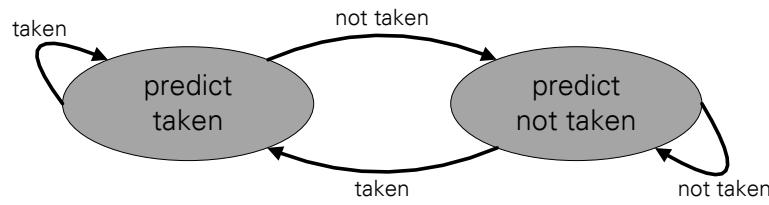
## Sprungverlaufstabelle (PHT: Pattern History Table)



- Ein oder mehrere Bits je Eintrag
  - Geben Wahrscheinlichkeit an mit der gesprungen wird
  - Einträge werden nach jeder Ausführung eines Sprungbefehls aktualisiert
- mehrere Befehle können auf den gleichen Eintrag abgebildet werden
  - Interferenz reduziert Qualität der Vorhersage

## Ein-Bit-Prädiktor

---

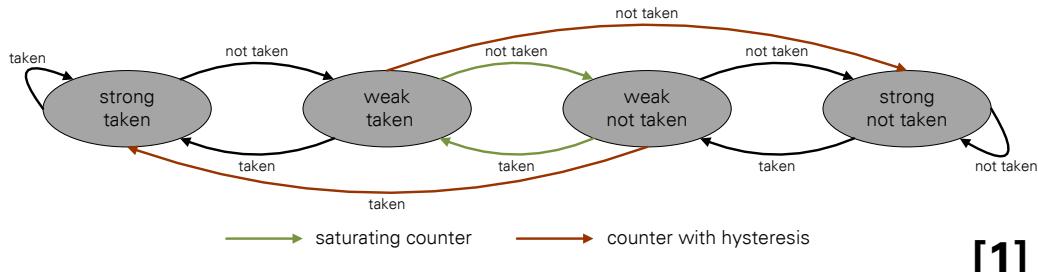


[1]

- 1-Bit-Zustandsinformation pro Sprungbefehl (in PHT)
- Bit bei Fehlspukulation invertieren

## Zwei-Bit-Prädiktor

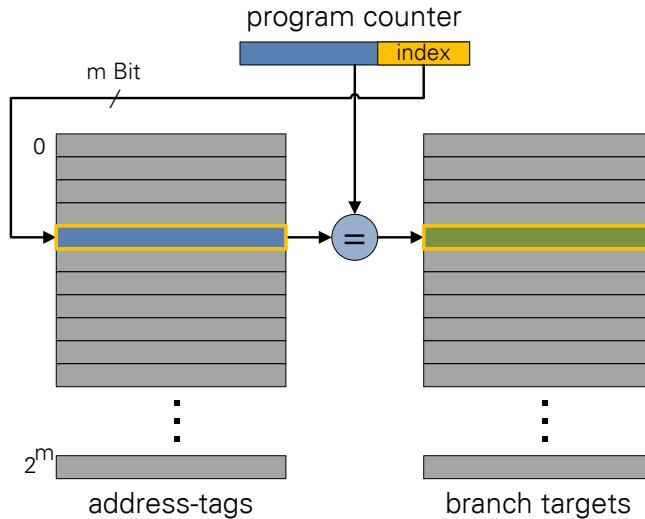
---



[1]

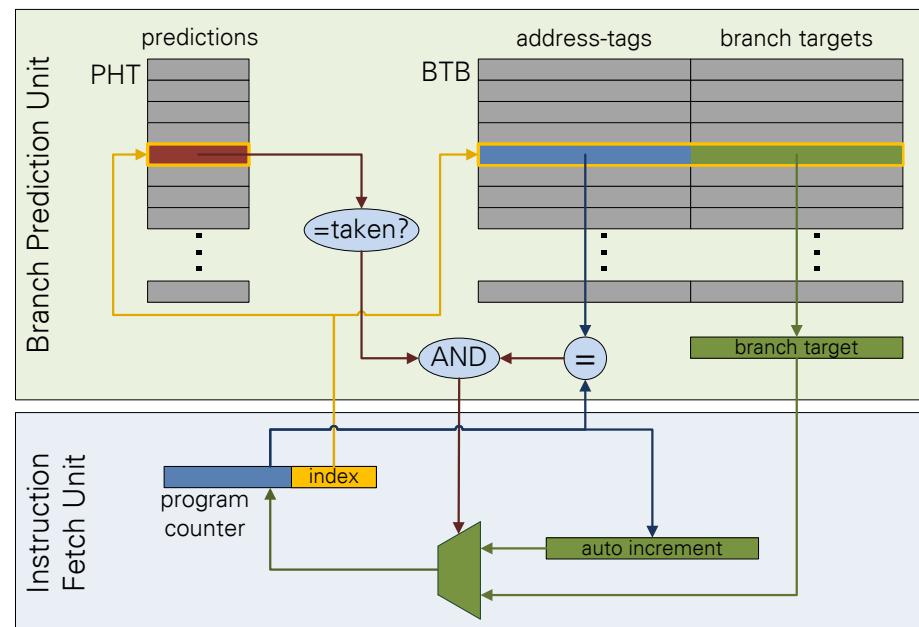
- Vorteil: erster Sprung bei wiederholtem Schleifeneintritt wird richtig vorhergesagt
- Erweiterung auf mehr als 2 Bits bringt keinen Vorteil

## Sprungzieladresscache (BTB: Branch Target Buffer)



- Eindeutige Adress-Tags → Keine Interferenz
- begrenzter Speicherplatz → Verdrängung möglich

## Zusammenspiel mit Instruction Fetch



## Gründe für falsche Vorhersagen

---

- Warmlaufphasen nach Programmstart oder Kontextwechsel
- Änderung des Sprungverhalten
  - z.B. Schleifenaustritt
- irreguläres Sprungverhalten
  - z.B. abhängig von Eingabedaten
- Interferenz in Sprungverlaufstabelle

## Bewertung n-Bit-Prädiktoren

---

- geringe Fehlerquote bei (großen) Schleifen
- schlechte Leistung bei if-then-, if-then-else Anweisungen
- Problem: Abhängigkeiten zwischen Sprüngen
  - Beispiel:

```
if (x==0) y=0;
...
if (y==0)
...
```
- Sprünge werden von n-Bit-Prädiktoren unabhängig vorhergesagt
  - Bei gleichem Verhalten auch gleiche Fehlerrate
  - zweiter Sprung könnte aber mit höherer Wahrscheinlichkeit richtig vorhergesagt werden

## Korrelationsanalyse

### Grundlage der Vorhersage:

- Bisheriges Verhalten des Sprungbefehls selbst
- Zusätzlich: Verhalten vorheriger (anderer) Sprungbefehle

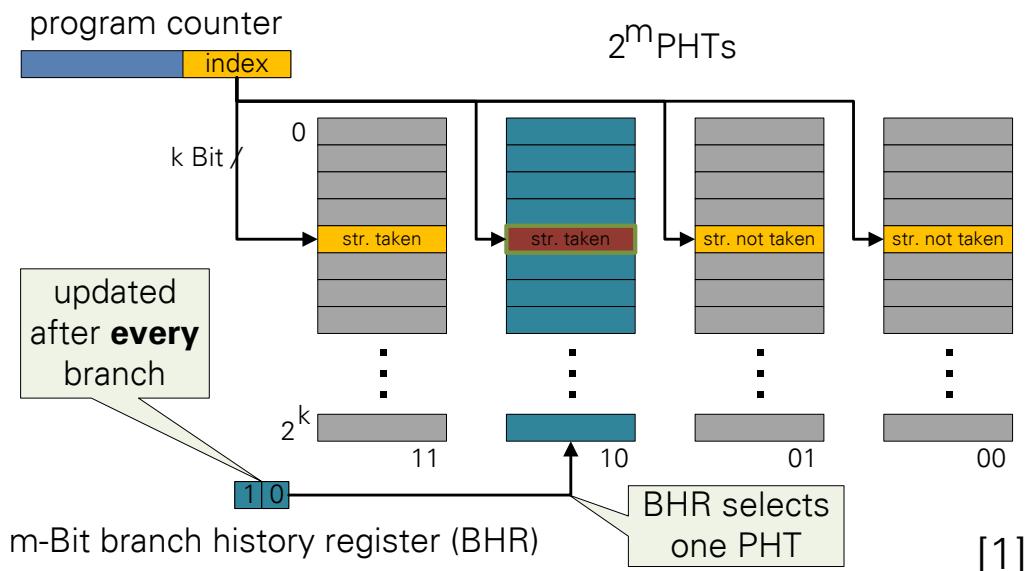
### ( $m, n$ )-Prädiktor

- Branch History Register (BHR)
  - $m$ -Bit-Schieberegister
  - Speichert Verhalten der letzten  $m$  Sprungbefehle  
(1=taken, 0=not taken)
- $2^m$  Sprungverlaufstabellen ( $n$ -Bit-Prädiktoren)
  - Auswahl einer Sprungverlaufstabelle durch BHR

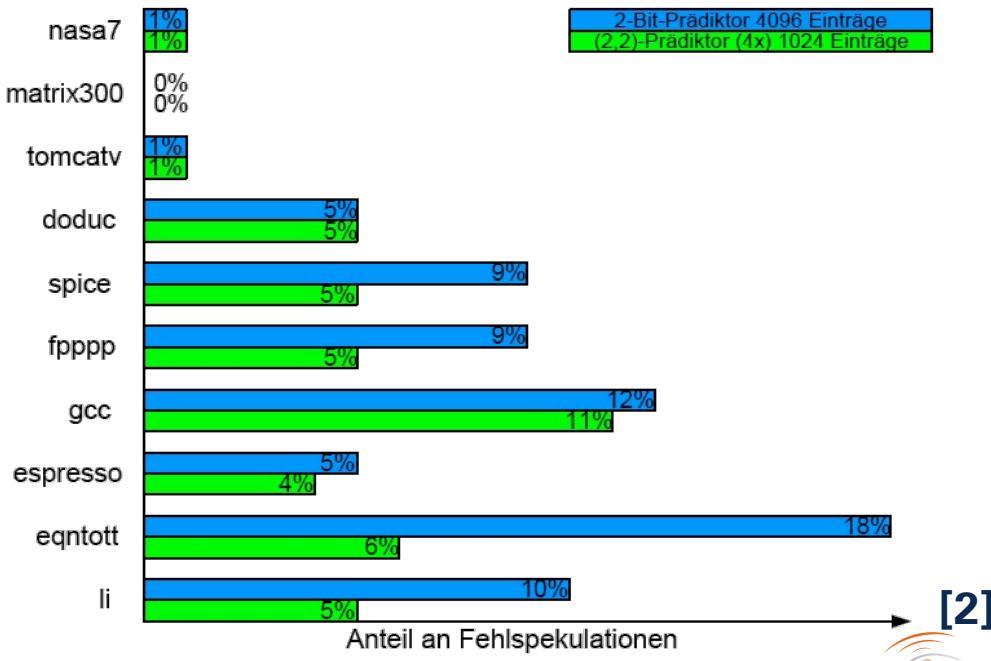
### Berücksichtigen Beziehungen zwischen Sprüngen

- wenig zusätzliche Hardware (BHR)
- i.d.R. höhere Trefferquote als  $n$ -Bit-Prädiktoren

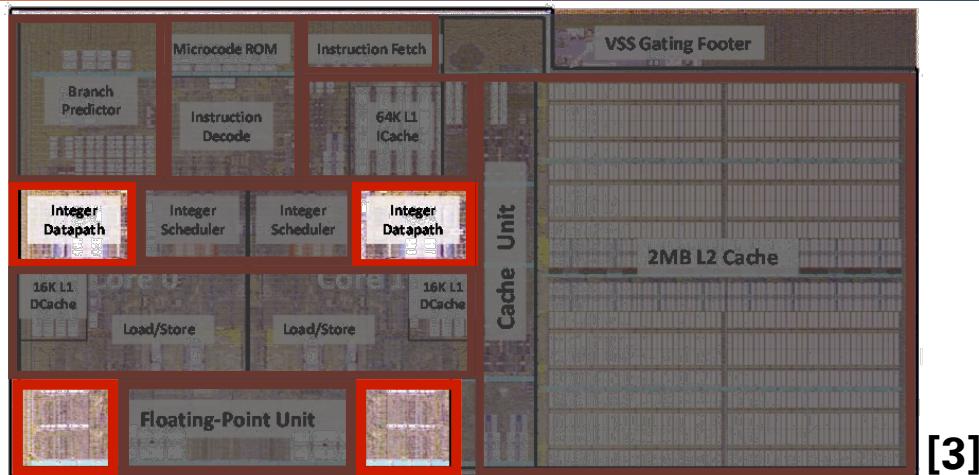
## (2,2)-Prädiktor



## Leistungsvergleich - SPECint89 und SPECfp89



## Hardwareaufwand: Aufteilung der Chipfläche



- Beispiel: dualcore AMD Bulldozer Modul
  - > 50% Cache und Speicherlogik
  - Sprungvorhersage nimmt etwa die Hälfte vom Front-end ein
  - Ca. 50% der verbleibenden Chipfläche für Out-of-Order Execution
  - Eigentliche Rechenwerke sehr klein (<10%)

## Zusammenfassung SISD

---

- Erhöhung der sequenziellen Verarbeitungsgeschwindigkeit
  - Mehr Instruktionen pro Sekunde (IPS) → Anwendungen laufen schneller
  - Keine Anpassung der Programme erforderlich
- Aber hohe Leistung muss teuer erkauft werden
  - Große Caches
  - Zusätzliche Logik und Puffer für Out-of-Order Execution
  - Tabellen der Sprungvorhersage benötigen viel Chip-Fläche
  - geringe Rechenleistung/Transistor
- Weitere Verbesserung der IPS schwierig
  - Nur geringe Steigerungen mit jeder neuen Prozessorgeneration

⇒ **Parallelverarbeitung zur weiteren Steigerung der Rechenleistung**

## Inhalt

---

- Steigerung der Rechenleistung bei sequenzieller Verarbeitung (SISD)
  - Pipelining und Superskalare Ausführung
  - Cache
  - Out-of-Order Execution
  - Sprungvorhersage
- Parallelverarbeitung
  - Befehlssatz-Erweiterungen (SIMD)
  - Multicore Prozessoren (MIMD)
  - Energie-Effizienz

## Parallelverarbeitung

---

### • SIMD Befehlssätze

- Verbreiterung der Datenpfade, Register und Rechenwerke
- Mehr Rechenleistung mit der Selben Anzahl von Befehlen
- Programme müssen an Befehlssätze angepasst werden

### • Multicore Prozessoren

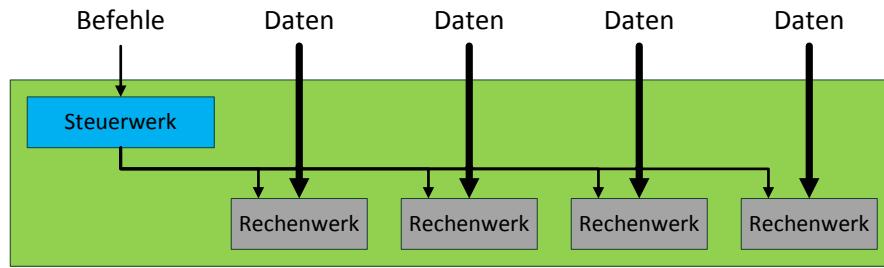
- Mehrere CPUs pro Prozessor
  - Steuerwerk und Rechenwerke sowie Teile der Speicherhierarchie werden dupliziert
  - Betriebssystem weist den Prozessoren unterschiedliche Prozesse zu (Scheduling)
  - Programme müssen mehrere Prozesse nutzen um von Multicore Prozessoren zu profitieren
- Keine Änderungen am Befehlssatz

---

# SIMD

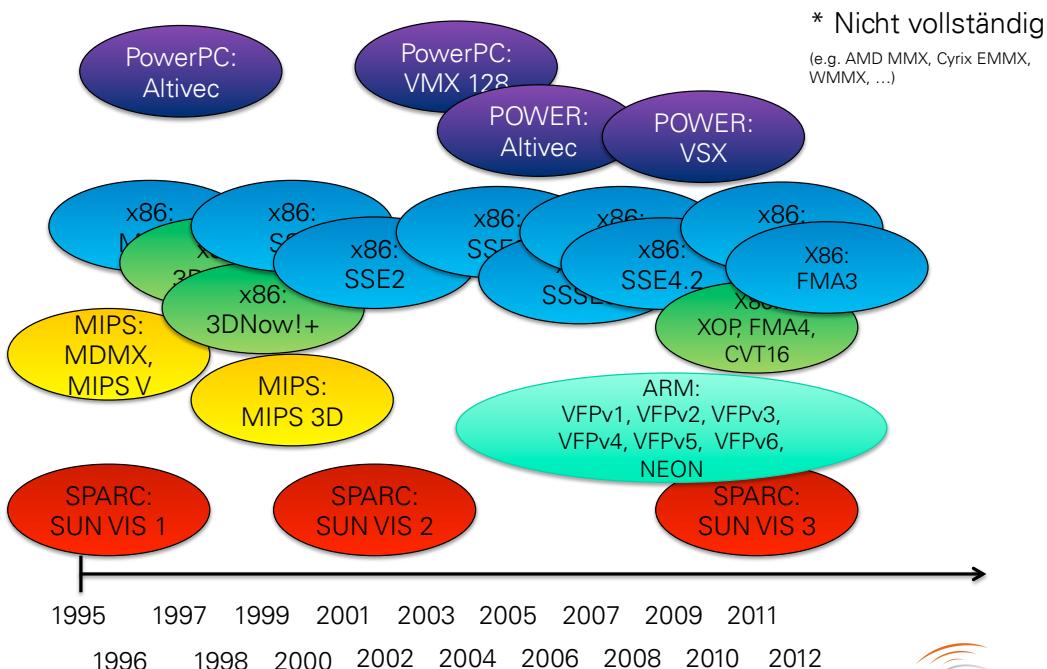
## Single Instruction Multiple Data

- Ein Befehlsstrom regelt mehrere Funktionseinheiten, die jeweils einen eigenen Datenstrom verarbeiten



- Explizite Parallelität statt aufwändiger Extraktion zur Laufzeit
  - Manuell durch Programmierer oder automatisch durch Compiler
  - Bei Erhöhung der Anzahl paralleler Operationen ist Anpassung der Software bzw. erneutes Compilieren erforderlich

## SIMD Entwicklungen seit 1995



## SIMD Befehlssätze im Vergleich

---

- Manche Erweiterungen arbeiten mit zusätzlichen Registern, manche benutzen vorhandene Register
- Unterschiedliche Vektorlängen (z.B. 64, 128 oder 256 Bit)
- Operationen mit unterschiedlichen Datentypen
  - Integer mit unterschiedlicher Breite der Operanden
  - Floating Point mit einfacher oder doppelter Genauigkeit
  - Floating Point Berechnungen nicht immer IEEE konform
- Nicht alle unterstützen Scatter- und Gather-Speicheroperationen zur Bearbeitung unabhängiger Datenströme
  - ggf. umständliches zusammenstellen der Vektorregister aus einzelnen Lade-Operationen erforderlich

## SIMD Erweiterungen in x86 Prozessoren

---

- 1996 MMX – Intel (Pentium MMX)
  - Verschiedene Derivate und MMX Erweiterungen von unterschiedlichen Prozessorherstellern (Intel, AMD, Cyrix)
  - Nutzt x87 FPU Register für Integer SIMD
- 1997 3DNow! – AMD (AMD K6-2)
  - Nutzt x87 FPU Register für Floating Point SIMD
  - Wurde von anderen Prozessorherstellern nicht übernommen und wird auch in aktuellen AMD Prozessoren nicht mehr unterstützt
- 1998 SSE – Intel (Pentium 3)
  - 8 neue 128 Bit Register (bei x86\_64 auf 16 Register erweitert)
  - Fließkommaberechnungen mit einfacher Genauigkeit (32 Bit)
  - Von anderen Prozessorherstellern (AMD, VIA) übernommen

## SIMD Erweiterungen in x86 Prozessoren

---

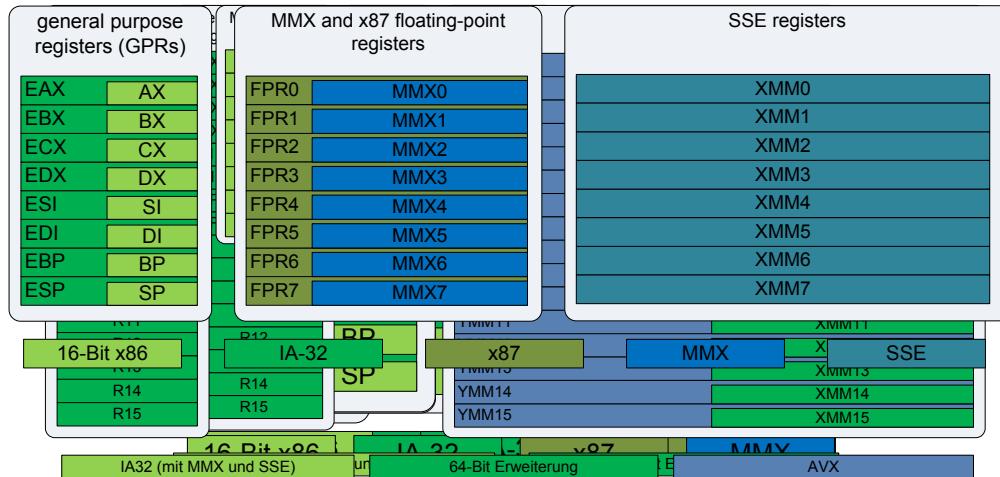
- 2001 SSE2 – Intel (Pentium 4)
  - Unterstützung für Integer und Floating Point mit doppelter Genauigkeit
- 2004 SSE3 – Intel (Pentium 4, Prescott)
  - Horizontale Berechnungen (innerhalb eines Vektorregisters)
- 2006 SSSE3 – Intel (Core2, Conroe)
- 2007 SSE4.1 – Intel (Core2, Penryn)
- 2007 SSE4a – AMD (Phenom, Barcelona)
- 2008 SSE4.2 – Intel (Core i\*, Nehalem)
- 2011 AVX – Intel (2te Generation Core i\*, Sandy Bridge)
  - Erweitert SIMD Register auf 256 Bit Breite
  - Ausschließlich Fließkommaberechnungen
  - Auch in aktuellen AMD Prozessoren verfügbar (AMD FX, Bulldozer)

## SIMD Erweiterungen in x86 Prozessoren

---

- 2011 FMA4 – AMD (AMD FX, Bulldozer)
  - Fused Multiply Accumulate
  - Vier-Adress-Format:  $D=A \cdot B + C$
  - Einfache und Doppelte Genauigkeit
- 2012 FMA3 – AMD (AMD FX, Piledriver)
  - Drei-Adress-Format:  $A=A \cdot B + C$
  - Kompatibel mit Intel FMA3 Spezifikation
- 2013 AVX2 – Intel (Haswell)
  - Integer SIMD Operationen auf 256 Bit AVX Registern
  - Fused Multiply Add, 3-Address-Format

## Architekturelle Registers

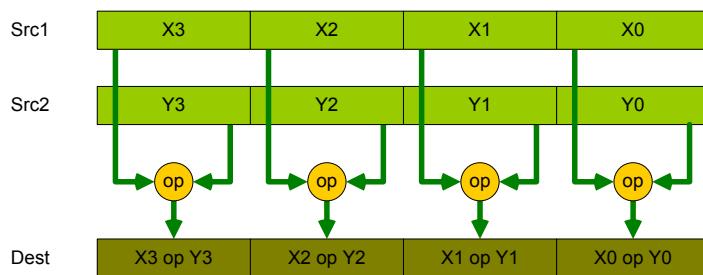


- Original x86 hat nur sehr wenige Register

– Häufige Speicherzugriffe erforderlich

- Zusätzliche Register durch Befehlssatzerweiterungen

## Befehlsbreite

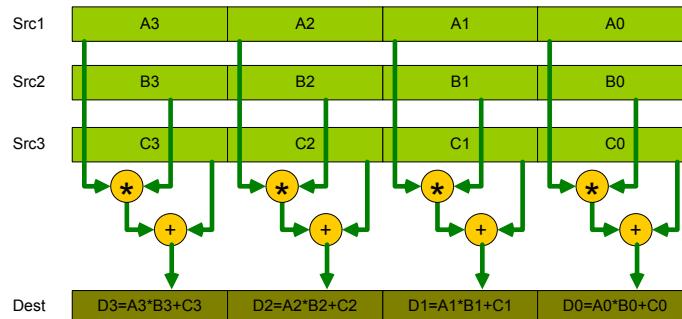


- Mehrere unabhängige Operationen auf SIMD Registern
  - SSE: 4 single precision oder 2 double precision Operationen
  - AVX: 8 single precision oder 4 double precision Operationen
- Lade- und Speicherbefehle für komplett Register
  - Um volle Cache- und Speicher-Bandbreite zu erzielen müssen Operanden an aufeinander folgenden Speicheradressen liegen

## Verarbeitungsbreite

- Verarbeitungsbreite ist nicht notwendigerweise identisch zur Befehlsbreite
  - SIMD-Befehle können von Decodern in mehrere Microops mit geringerer Verarbeitungsbreite zerlegt werden
  - Ermöglicht Unterstützung neuer Befehlssätze ohne Anpassungen an den Ausführungseinheiten
  - Die maximale Rechenleistung wird durch die Anzahl und die Verarbeitungsbreite der Rechenwerke bestimmt
- Beispiele:
  - Intel Pentium 3 und Pentium M unterstützen 128 Bit SSE-Befehle, aber die FPU verarbeitet nur 64 Bit pro Takt
  - AMD FX Prozessoren unterstützen 256 Bit AVX- und FMA-Befehle, aber die beiden FPUs verarbeiten jeweils nur 128 Bit

## Fused Multiply-Add (FMA)



- Kombination von Multiplikation und Addition in einem Befehl
- Genauereres Ergebnis als getrennte Multiplikation und Addition
  - Nur eine Rundung statt zwei bei getrennter Berechnung
  - Implementierung in Hardware erforderlich
  - Kompatibilität kann nicht durch Decoder hergestellt werden

## Double Precision Floating-Point Performance

Prozessor	Jahr	Befehls-satz	Befehls-breite	FPU (pro Kern)	Operationen /Takt
Pentium Pro (P6)	1995	x87	64 Bit	1x 64 Bit	1
Pentium 2 (Klamath)	1997	x87	64 Bit	1x 64 Bit	1
Pentium 3 (Katmai)	1999	SSE	64 Bit*	1x 64 Bit	1
Pentium 4 (Willamette)	2000	SSE2	128 Bit	1x 128 Bit	2
Pentium M (Banias)	2003	SSE2	128 Bit	1x 64 Bit	1
Core2 Duo (Conroe)	2006	S-SSE3	128 Bit	2x 128 Bit	4
1st gen Core i7 (Nehalem)	2008	SSE4.2	128 Bit	2x 128 Bit	4
2nd gen Core i7 (Sandy Br.)	2011	AVX	256 Bit	2x 256 Bit	8
*: SSE unterstützt nur einfache Genauigkeit → für doppelte Genauigkeit weiterhin x87					
4th gen Core i7 (Haswell) 2013 AVX2/ 256 Bit FMA 2x 256 Bit 16					
Bis zur Einführung der Core Mikroarchitektur Leistungssteigerung im Wesentlichen über höheren Takt					

- Danach Leistungssteigerung über Erhöhung der IPC

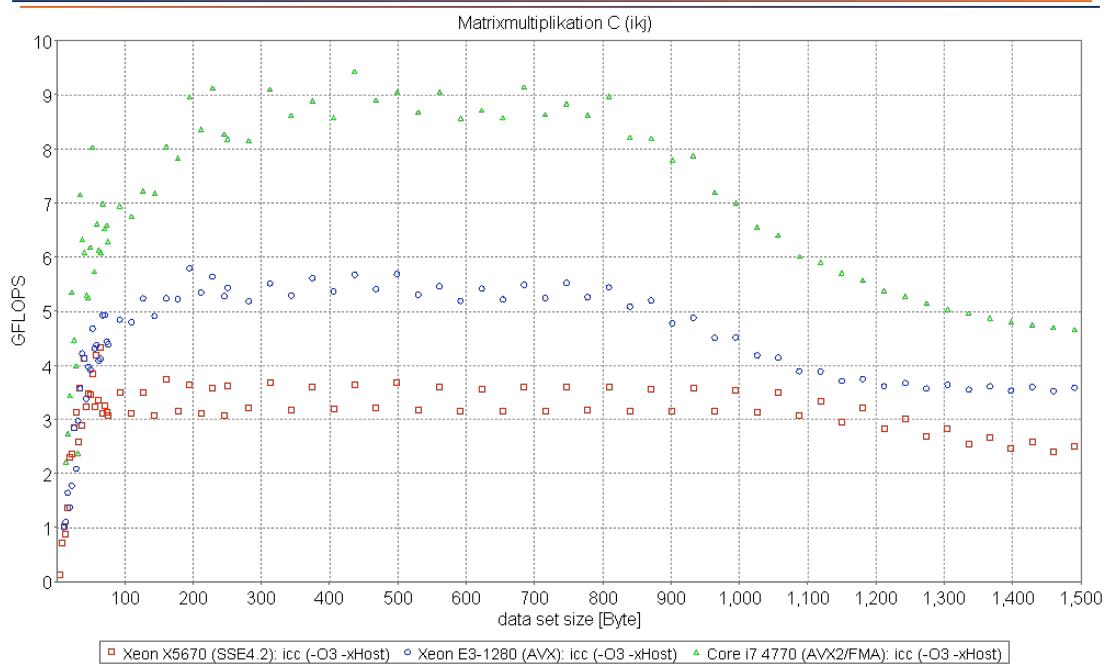
## SIMD über Compiler

- Compiler versuchen Code automatisch zu vektorisieren
  - Dazu müssen ggf. Compiler-Flags angegeben werden (z.B. -O3, -xAVX)
  - Zusätzliche Flags zur Kontrolle (z.B. -vec-report [level])
    - Gibt Hinweise was (warum) nicht vektorisiert werden konnte
- Zwei wesentliche Arten:
  - Block-Vektorisierung
    - Fasst im Quellcode aufeinander folgende Befehle zu einen SIMD-Befehl zusammen
  - Schleifen-Vektorisierung
    - Fasst mehrere Schleifeniterationen zu einer Iteration mit SIMD-Befehlen zusammen
      - Klappt nur für einfache Schleifen
        - „for(i=0;i<100;i++) a[i]=b[i]\*c[i]“  
→ 25 Iterationen mit SSE (single precision)
        - „for(;;){...}“ oder „for(i=0;i<n; i++,j++){...}“  
→ keine Vektorisierung (unexpected loop form)

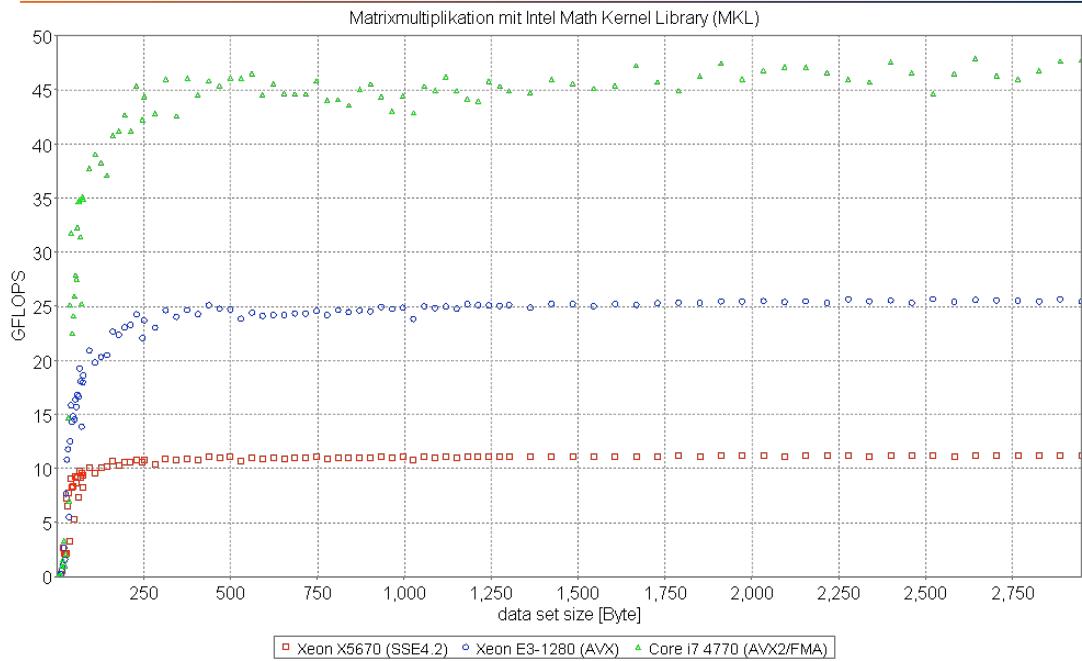
## SIMD über Compiler: Performance

Prozessor	Core i7 4770	Xeon E3-1280	Xeon X5670
Mikroarchitektur	Haswell	Sandy Bridge	Westmere
Befehlssatz	AVX2	AVX	SSE4.2
Takt	3,4 GHz	3,5 GHz	2,93 GHz
FPU	2x 256 Bit FMA	2x 256 Bit	2x 128 Bit
Operationen/ Takt	16 dp (8 fma)	8 dp (4 mul, 4 add)	4 dp (2 mul, 2 add)
L1 / L2 pro Kern	2x 32 KiB / 256 KiB	2x 32KiB / 256 KiB	2x 32KiB / 256 KiB
L3 pro Chip	8 MiB	8 MiB	12 MiB
Speicher	2x DDR3 1600	2x DDR3 1333	3x DDR3 1333
Compiler	Intel 14.0	Intel 12.0	Intel 11.1
<b>GFLOPS (1 Kern)</b>	<b>54,4</b>	<b>28</b>	<b>11,7</b>

## SIMD über Compiler: Performance



## Nutzung optimierter Bibliotheken



## Explizite Nutzung von SIMD Befehlen

### ● Intrinsics

- Können direkt in C Code genutzt werden
- `#include <*intrin.h>` (\* definiert MMX/SSE/AVX subset)
- Zusätzliche Datentypen für SIMD-Register
- Compiler kümmert sich um Registerallokation
- SIMD-Befehle: `_mm_<operation>_<datatype>(operands)`
- z.B. `_mm_add_ps()` == add packed single (4\*32 bit, SSE)

### ● Inline Assembler

- Programmierer für Nutzung der Register selbst verantwortlich
  - Mehr Kontrolle aber auch mehr Aufwand
  - Ggf. getrennte Implementierung für 32 und 64 Bit, wegen Registeranzahl
- `__asm` Regionen schwierig zu nutzen
  - Parameterübergabe
  - manuelle Sicherung von Register-Inhalten

## SIMD x86 – MMX: Beispiel – toUpperCase()

---

Standardimplementierung "Wandle Kleinbuchstaben eines Strings in Großbuchstaben um":

```
for (i=0;i<string_length;i++)  
    if ((string[i]>='a') && (string[i]<='z'))  
        target_string[i]=string[i]-(‘z’-‘z’);  
    else  
        target_string[i]=string[i];
```

1.) Optimierung von "target\_string[i]=string[i]-(‘z’-‘z’);" mit Bit Magic!

'A'-'Z' ASCII-Code 0x41-0x5A  
'a'-'z' ASCII-Code 0x61-0x7A  
→ lösche 0x20 um Kleinbuchstaben in Großbuchstaben umzuwandeln  
→ target\_string[i]=string[i]&0xDF (löscht Bit an Position 0x20)

2.) Erlaube nur "größer-als" Vergleiche

```
for (i=0;i<string_length;i++)  
    if ((string[i]>'a'-1) && (!(string[i]>'z'))) {  
        target_string[i]=string[i]&0xDF;  
    } else  
        target_string[i]=string[i];
```



TECHNISCHE  
UNIVERSITÄT  
DRESDEN

---

Wolfgang E. Nagel



## SIMD x86 – MMX: Beispiel – toUpperCase()

---

3.) Kürzere Version:

```
for (i=0;i<string_length;i++)  
    target_string[i]=  
        ((string[i]>('a'-1)) && (!(string[i]>'z'))) ?  
            string[i]&0xDF:  
            string[i];
```

4.) Zusätzliche Operation (ohne Auswirkung):

```
for (i=0;i<string_length;i++)  
    target_string[i]=  
        ((string[i]>('a'-1)) && (!(string[i]>'z'))) ?  
            string[i]&0xDF:  
            string[i] &0xFF;
```

5.) Umwandeln von "and" in "or"

```
for (i=0;i<string_length;i++)  
    target_string[i]=  
        ( ('a' > string[i] ) || ( string[i] > 'z' ) ?  
            string[i]&0xFF:  
            string[i]&0xDF;
```

6.) Entwicklung einer Version, die die 0xDF/0xFF Masken in MMX-Register erstellt



TECHNISCHE  
UNIVERSITÄT  
DRESDEN

---

Wolfgang E. Nagel



## SIMD x86 – MMX: Beispiel – toUpperCase()

```
emms                                // switch to mmx mode
movq mm1, 0x6161616161616161      //Lower bound is a (0x61)
movq mm2, 0x7A7A7A7A7A7A7A7A      //Upper bound is z (0x7A)
mov esi, 0                           // offset
mov ecx, (string_length/8)
L1:
    movdq mm0, string[esi]          // load 8 characters (8*XX) to mm0
    movq mm4, mm0                  // mm4=XX
    movq mm5, mm1                  // mm5='a'
    pcmppgtb mm4, mm2             // if XX > z, then mm4 = 0xFF else mm4 = 0x00
    pcmppgtb mm5, mm0             // if a > XX, then mm5 = 0xFF else mm5 = 0x00
    por mm5, mm4                  // if (XX >= a) AND (XX <= z) then mm5 = 0x00, else 0xFF
    por mm5, 0xDFDFDFDFDFDFDFD // set 'no-lowercase-bit' for all 0x00
    pand mm0, mm5                 //apply it to char array
    movdq target_string[esi] mm0//store it
    add esi, 8
loop L1
emms                                // switch back
```



## SIMD x86 – MMX: Beispiel – toUpperCase()

H	I	~	W	E	L	T	!	0x00	0x00	0xFF	0x00	0x00	0x00	0x00	0x00
a	a	a	a	a	a	a	a	0xFF	0x00						
z	z	z	z	z	z	z	z								

```
    movdq mm0, string[esi]          // load 8 characters (8*XX) to mm0
    movq mm4, mm0                  // mm4=XX
    movq mm5, mm1                  // mm5='a'
    pcmppgtb mm4, mm2             // if XX > z, then mm4 = 0xFF else mm4 = 0x00
    pcmppgtb mm5, mm0             // if a > XX, then mm5 = 0xFF else mm5 = 0x00
    por mm5, mm4                  // if (XX >= a) AND (XX <= z) then mm5 = 0x00, else 0xFF
    por mm5, 0xDFDFDFDFDFDFDFD // set 'no-lowercase-bit' for all 0x00
    pand mm0, mm5                 //apply it to char array
    movdq target_string[esi] mm0//store it
```



## SIMD x86 – SSE: Beispiel – Matrix-Vektor-Multiplikation

---

Standardimplementierung :

```
for (i=0;i<N;i++)  
  for (j=0;j<M;j++)  
    y[i]=y[i]+A[i][j]*x[j];
```

Schleifenentrollen: (Annahme: M ist Vielfaches von 4)

```
for (i=0;i<N;i++)  
  for (j=0;j<M;j+=4)  
    y[i]=y[i]  
      +A[i][j]*x[j]  
      +A[i][j+1]*x[j+1]  
      +A[i][j+2]*x[j+2]  
      +A[i][j+3]*x[j+3];
```

## SIMD x86 – SSE: Beispiel – Matrix-Vektor-Multiplikation

---

Einführen eines neuen Vektors für innere Schleife

```
for (i=0;i<N;i++){  
  float[4] tmp={0,0,0,0};  
  for (j=0;j<M;j+=4){  
    tmp[0]+=A[i][j]*x[j];  
    tmp[1]+=A[i][j+1]*x[j+1];  
    tmp[2]+=A[i][j+2]*x[j+2];  
    tmp[3]+=A[i][j+3]*x[j+3];  
  }  
  y[i]+=tmp[0]+tmp[1]+tmp[2]+tmp[3];  
}
```

## SIMD x86 – SSE: Beispiel – Matrix-Vektor-Multiplikation

```
for (i=0;i<N;i++) {
    y_vec=_mm_load_ss (&y[i]);
    tmp_vec=_mm_setzero_ps();
    for (j=0;j<M;j+=4) {
        A_vec=_mm_loadu_ps(&(A[i][j]));      // 4 Elemente der Matrix laden
        x_vec=_mm_loadu_ps(&x[j]);           // 4 Elemente des Vektors laden
        A_vec=_mm_mul_ps(A_vec,x_vec);       // 4 Multiplikationen
        tmp_vec=_mm_add_ps(tmp_vec,A_vec);    // Aufsummieren in tmp_vec
    } // done inner loop
    // add tmp_vec horizontally
    tmp2_vec=_mm_shuffle_ps(tmp_vec, tmp_vec, _MM_SHUFFLE(2,3,0,1))
    tmp_vec=_mm_add_ps(tmp_vec,tmp2_vec)
    tmp2_vec=_mm_shuffle_ps(tmp_vec, tmp_vec, _MM_SHUFFLE(0,1,2,3))
    tmp_vec=_mm_add_ps(tmp_vec,tmp2_vec)
    // add to y and store new y value
    y_vec=_mm_add_ss(tmp_vec,y_vec)
    _mm_store_ss(&y[i],y_vec);
}
```



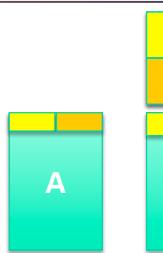
TECHNISCHE  
UNIVERSITÄT  
DRESDEN

Wolfgang E. Nagel



## SIMD x86 – SSE: Beispiel – Matrix-Vektor-Multiplikation

-	-	-	0.0
1.12	9.75	3.65	1.5
0.2	1.08	2.66	0.07
1.0	1.8	1.9	0.1



```
→ y_vec=_mm_load_ss (&y[i]);
→ tmp_vec=_mm_setzero_ps();
for (j=0;j<M;j+=4) {
    → A_vec=_mm_loadu_ps(&(A[i][j]));
    → x_vec=_mm_loadu_ps(&x[j]);
    → A_vec=_mm_mul_ps(A_vec,x_vec);
    → tmp_vec=_mm_add_ps(tmp_vec,A_vec);
} // done inner loop
```



TECHNISCHE  
UNIVERSITÄT  
DRESDEN

Wolfgang E. Nagel



## SIMD x86 – SSE: Beispiel – Matrix-Vektor-Multiplikation

---

-	-	-	<b>16.02</b>
<b>16.02</b>	<b>16.02</b>	<b>16.02</b>	<b>16.02</b>
<b>5.15</b>	<b>5.15</b>	<b>10.87</b>	<b>10.87</b>

```
// add tmp_vec horizontally
→ tmp2_vec=_mm_shuffle_ps(tmp_vec, tmp_vec,_MM_SHUFFLE(2,3,0,1))
→ tmp_vec=_mm_add_ps(tmp_vec,tmp2_vec)
→ tmp2_vec=_mm_shuffle_ps(tmp_vec, tmp_vec,_MM_SHUFFLE(0,1,2,3))
→ tmp_vec=_mm_add_ps(tmp_vec,tmp2_vec)
// add to y and store new y value
→ y_vec=_mm_add_ss(tmp_vec,y_vec)
_mm_store_ss(&y[i],y_vec);
}
```

---

## MULTICORE PROZESSOREN

## Stetig steigendes Transistor Budget

---

- Exponentielles Wachstum der Transistorzahl zu schnell für evolutionäre Weiterentwicklung der Mikroarchitekturen
  - Caches vergrößern alleine führt nicht zur gewünschten Leistungssteigerung
  - Weiterer Ausbau von Superskalarität, Out-of-Order Execution und Sprungvorhersage ebenfalls mit abnehmendem Leistungszuwachs
    - Daten-Abhängigkeiten zwischen Befehlen
    - Häufige Verzweigungen
  - Breite von SIMD Befehlen kann nicht beliebig gesteigert werden

### ● Neuer Ansatz:

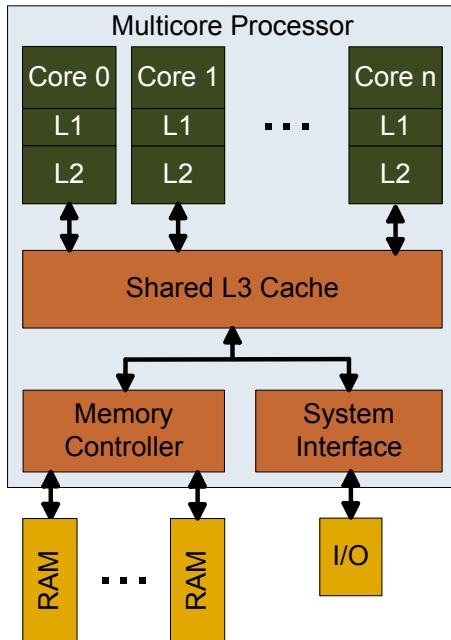
- **Mehr Prozessoren statt schnellerer Prozessoren**

## Arten von Multicore Prozessoren

---

- Symmetrische Multicore Prozessoren (SMP)
  - Identische Prozessor-Kerne
  - Programme können von allen Kernen gleichermaßen ausgeführt werden
- Asymmetrische Multicore Prozessoren (AMP)
  - Einheitlicher Befehlssatz (z.B. ARM big.LITTLE)
    - Kerne mit unterschiedlicher Rechenleistung und Energie-Effizienz
    - Programme können prinzipiell von allen Kernen ausgeführt werden
    - Aber Auswahl des geeigneten Kern-Typs anhand der Anforderungen der Anwendung notwendig
  - Verschiedene Befehlssätze (z.B. CPU+GPU)
    - Ebenfalls unterschiedliche Rechenleistung und Energie-Effizienz
    - Programme an bestimmte Kerne gebunden

## Symmetrische Multicore Prozessoren



- Identische Prozessor-Kerne
  - Gleicher Befehlssatz
  - Gleiche Rechenleistung
- Typischerweise mehrere Cache Level
  - L1 Cache i.d.R. pro Kern
  - Höhere Cache Level häufig von mehreren Kernen gemeinsam genutzt
- Speichercontroller und System-Anbindung von allen Kernen genutzt
  - Häufig mehrere Speicherkanäle
  - Kerne teilen sich die verfügbare Bandbreite

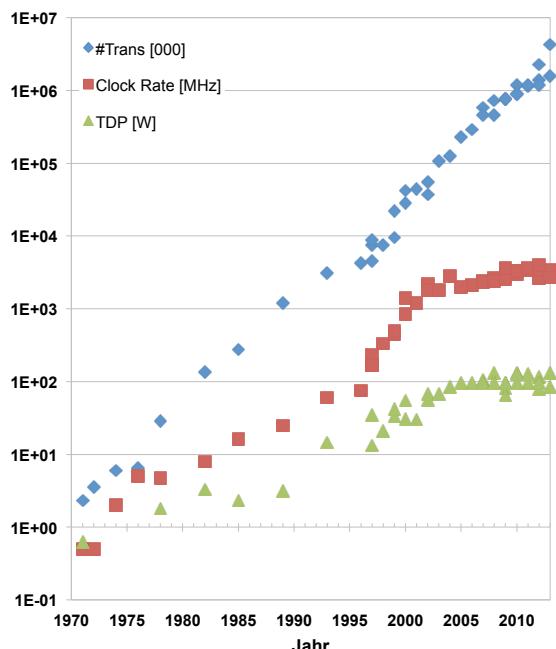
## Double Precision Floating-Point Performance

Prozessor	Jahr	Frequenz [MHz]	Operationen/Takt	Kerne	GFLOPS
Pentium Pro (P6)	1995	200	1	1	0,2
Pentium 2 (Klamath)	1997	300	1	1	0,3
Pentium 3 (Katmai)	1999	600	1	1	0,6
Pentium 4 (Willamette)	2000	1500	2	1	3,0
Pentium 4 (Northwood)	2002	3067	2	1	6,1
Pentium D (Smithfield)	2005	3200	2	2	12,8
Core2 Duo (Conroe)	2006	2667	4	2	21,3
1st gen Core i7 (Nehalem)	2008	3200	4	4	51,2
2nd gen Core i7 (Sandy Br.)	2011	3400	8	4	108,8
4th gen Core i7 (Haswell)	2013	3400	16	4	217,6

• Bis 2005 Leistungsaugmentation im Wesentlichen über höhere Taktfrequenz

- Danach Wechsel zu mehr Kernen

## Entwicklung des Energieverbrauchs



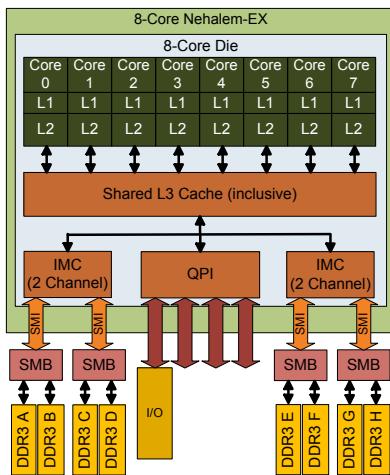
- (Luft-)Kühlung limitiert Verbrauch auf 100 – 150 Watt pro Prozessor
- Hohe Frequenz erfordert hohe Spannung
  - $E \sim V^2$
  - $E \sim f$
  - $E \sim$  transistor count
- Mehr Kerne mit geringerer Frequenz sind effizienter für parallele Anwendungen
  - z.B. Trade-off 65W Core2:  
2x 3.33 oder 4x 2.83 GHz

## Nutzung von Multicore Prozessoren

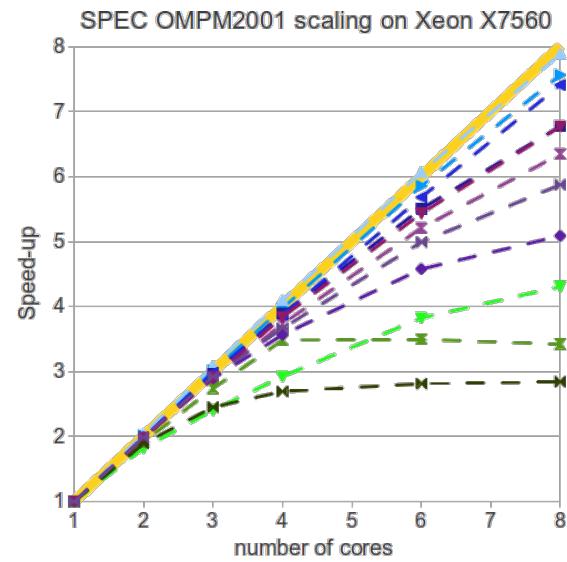
- Programme müssen explizit mehreren Prozessoren nutzen
  - OpenMP
    - Gemeinsamer Adressraum
    - Definition paralleler Regionen im Quellcode
      - Threads werden zur Laufzeit dynamisch angelegt
      - Basierend auf pthreads
    - Angabe der Threadanzahl über Umgebungsvariable OMP\_NUM\_THREADS
  - MPI
    - Mehrere Prozesse mit getrennten Adressräumen
    - Expliziter Nachrichtenaustausch
    - Angabe der Prozessanzahl beim Start der Anwendung
      - mpirun –np <n> ./executable

## Skalierbarkeit paralleler Anwendungen

Intel Xeon X7560

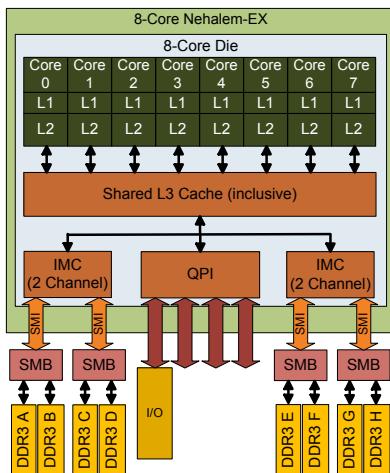


- 24 MiB L3
- 4 Speicherkanäle

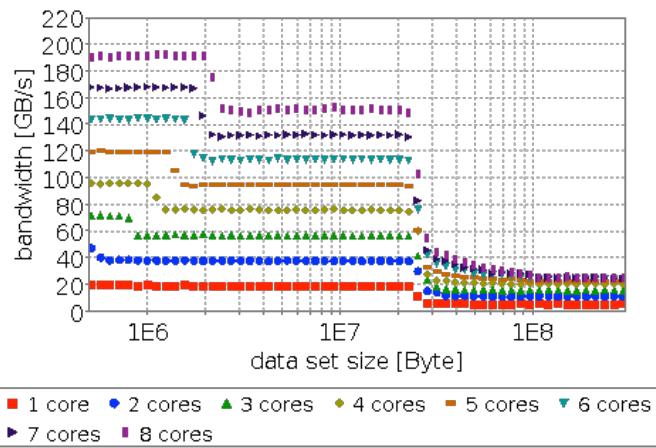


## Skalierung der Bandbreite mit der Kernanzahl

Intel Xeon X7560



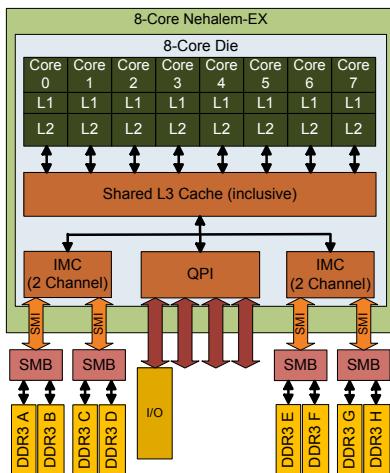
- 24 MiB L3
- 4 Speicherkanäle



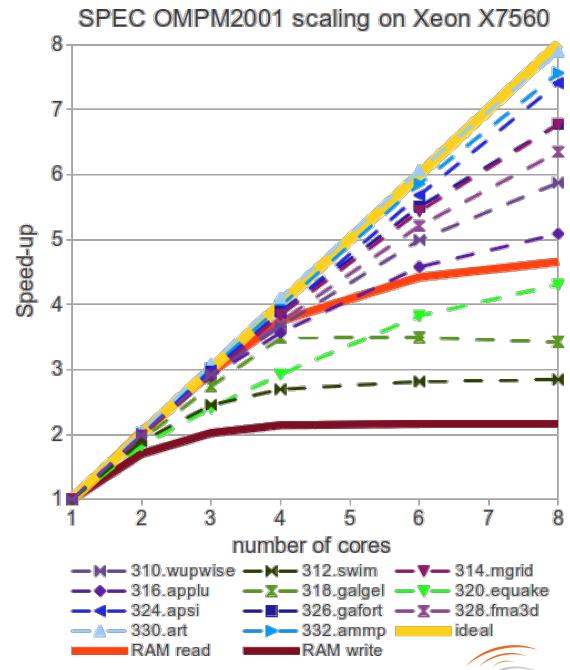
- Bandbreite des L3 Caches skaliert linear mit Kernanzahl
- Speicherbandbreite mit 4 Kernen nahezu ausgeschöpft

# Skalierbarkeit paralleler Anwendungen

Intel Xeon X7560

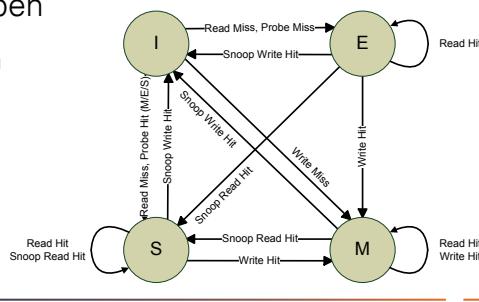


- 24 MiB L3
- 4 Speicherkanäle

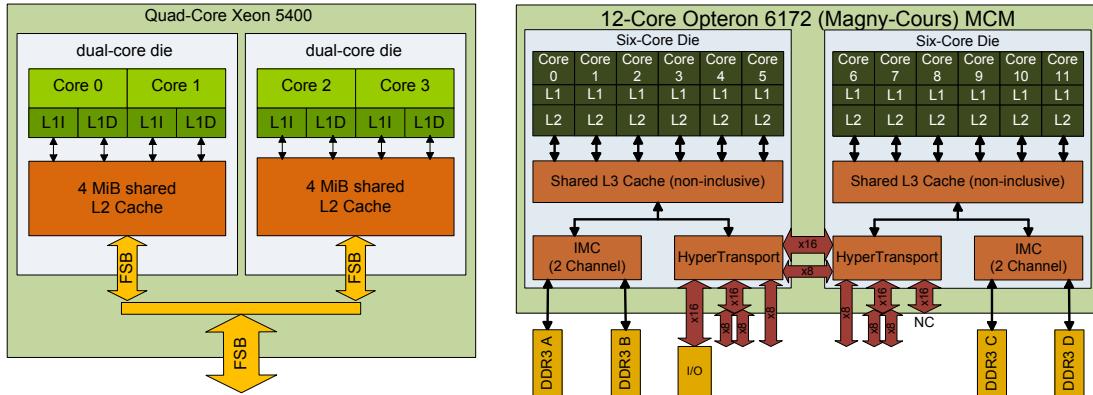


## Cache Kohärenz

- Lokale und gemeinsame Caches
  - Speichern Daten, die von den einzelnen Kernen angefordert werden
  - Enthalten auch Daten, die von mehreren Kernen genutzt werden
- Cache Kohärenz Problem
  - Caches sind transparent für die Anwendung
    - parallele Programme müssen das gleiche Resultat liefern das sie ohne Caches liefern würden
    - Änderungen müssen für alle Kerne sichtbar sein
- Kohärenz Protokolle stellen sicher, dass alle Kerne eine konsistente Sicht auf den Speicher haben
  - Invalidierung aller Kopien bevor Änderungen vorgenommen werden
  - Beispiel: MESI-Protokoll

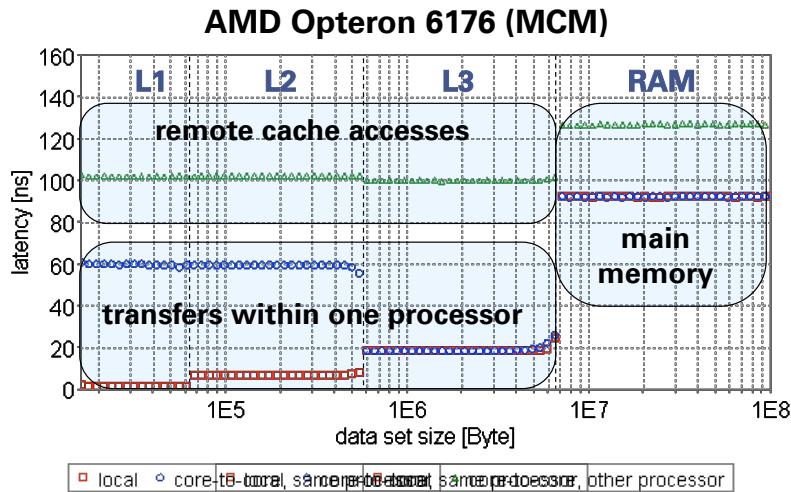


## Multi-Chip-Module (MCM)



- Einfache Methode um Kernanzahl zu erhöhen
- Verwendung des selben Chips in unterschiedlichen Produktkategorien
  - 1 Chip für Mainstream
  - 2 Chips für High-End oder Server Prozessoren

## Cache- und Speicherlatenz in Multicore Prozessoren



- Ansteigende Latenz innerhalb der Cache-Hierarchie eines Kerns
- Hohe Latenz beim Zugriff auf dedizierte Caches anderer Kerne
- Non Uniform Memory Access (NUMA) in Multi-Chip-Modulen

# Asymmetrische Multicore Prozessoren

## Single ISA heterogeneous core

- ARM big.LITTLE: Kombination von Cortex A15 und Cortex A7 CPUs
- Nvidia Tegra 4:
  - 4+1 ARM Cortex A15 Prozessor-Kerne
  - 4x high performance und 1x low power
- Jeweils selber Befehlssatz aber unterschiedliche Leistungsniveaus
  - Keine Anpassung der Anwendungen nötig
  - Kompliziertes Scheduling

## Heterogeneous ISA

- AMD APUs: x86 + Radeon Kerne
- Nvidia Tegra: ARM + GPU Kerne
- Unterschiedliche Befehlssätze
- Programmierung mit CUDA, OpenCL, OpenGL, etc.

# Beispiel: AMD Accelerated Processing Units



Nathan Brookwood:

AMD Fusion™ Family of APUs

- Asymmetrischer Multicore mit unterschiedlichen Befehlssätzen
  - Bis zu 4 x86 Kerne
  - Bis zu 512 GPU Kerne (Kaveri)
- GPU Kerne können mit Hilfe von OpenCL und OpenACC auch für Berechnungen genutzt werden
- Heterogeneous uniform memory access (hUMA)
  - Gemeinsamer Adressraum für CPU und GPU Kerne

# Zusammenfassung Parallelverarbeitung

---

## • SIMD Befehlssätze

- Theoretische Rechenleistung steigt linear mit Verarbeitungsbreite
- Anpassung der Software erforderlich
  - Automatische Vektorisierung durch Compiler nur für einfache Fälle
  - Manuelle Nutzung nicht trivial
- Speed-up in Anwendungen i.d.R. nicht proportional zur Verarbeitungsbreite

## • Multicore Prozessoren

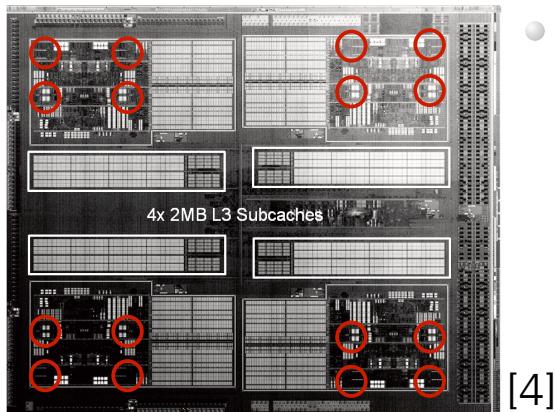
- Rechenleistung steigt linear mit Kernanzahl
- Benötigt parallele Programme → OpenMP, MPI, PGAS
- Rechenleistung häufig durch gemeinsame Ressourcen (z.B. Speicherbandbreite) beschränkt

---

# ENERGIE-EFFIZIENZ

## Aufteilung der Chipfläche in General Purpose Prozessoren

- Fokus auf hohe Rechenleistung für einzelne Threads
  - Hohe IPC → superskalare out-of-order Execution, Sprungvorhersage
  - Latenz-optimierte Speicherhierarchie → große Caches
- Beispiel: 8-Kern AMD FX
  - 4 Module mit je 2 Kernen und 2 MiB L2
  - Ca. 50% vom Chip sind gemeinsame Ressourcen
    - 8 MiB L3 Cache
    - Crossbar
    - Speichercontroller
  - Rechenwerke < 5% der Chipfläche



[4]

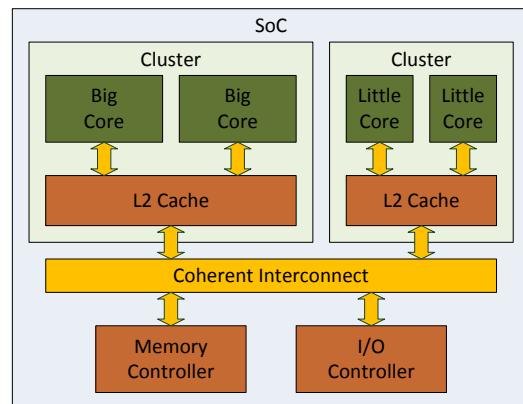
→ geringe Rechenleistung pro Transistor (Chipfläche)  
schlechte Energie-Effizienz

→

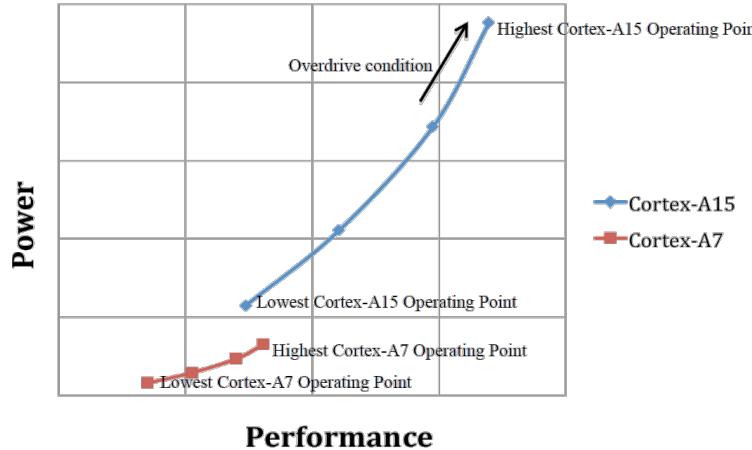


## ARM big.LITTLE

- Big Core: ARM Cortex A15
  - ARM v7 ISA (32 Bit)
  - Out-of-order Architektur
  - 3-fach superskalar
- Little Core: ARM Cortex A7
  - ARM v7 ISA (32 Bit)
  - In-order Architektur
  - 2-fach superskalar
- Identischer Befehlssatz erlaubt Verschiebung von Anwendungen und Betriebssystem zwischen Kern-Typen
- Verschiedene Betriebsarten
  - Migrationsmodel: Umschaltung zwischen Clustern
  - MP Model: alle Kerne gleichzeitig aktiv



## ARM big.LITTLE



[5]

- Cortex A15 liefert doppelte bis dreifache Rechenleistung des Cortex A7 (beide bei ihrer jeweils höchsten Taktfrequenz)
- Cortex A7 hat drei- bis vierfache Rechenleistung pro Watt
  - z.B. Hälfte der Rechenleistung bei einem Sechstel der Leistungsaufnahme

## Throughput Computing

- Hohe Single-Thread Performance nicht in allen Bereichen erforderlich
- Mikro-Server
  - Vielzahl von dicht gepackten, stromsparenden Prozessoren
    - Geringe Taktfrequenz
    - In-order oder moderate out-of-order Verarbeitung
  - Höhere Rechenleistung pro Watt als klassische Server-Prozessoren
- Hardware-Beschleuniger / Many-Core Prozessoren
  - Viele einfache Kerne auf einem Chip
  - Auf Bandbreite optimierte Speicher-Hierarchie (wenig Cache, viele Speicherkanäle)
  - Deutlich bessere Rechenleistung pro Transistor und pro Watt
  - Schwieriger zu Nutzen (Amdahl's Law)

## Zusammenfassung

---

### ● Hardware-Sicht

- Einfache Methoden mehr Transistoren zu nutzen:
  - Kernanzahl erhöhen
  - Cache vergrößern
- Aufwändige Weiterentwicklungen:
  - Befehlssatzerweiterungen
  - Steigerung der Instruktionen pro Sekunde (IPS) durch Verbesserungen an der Mikroarchitektur (out-of-order, Sprungvorhersage)

### ● Software-Sicht

- Gesteigerte Single Thread Leistung zu nutzen ist trivial
- SIMD zum Teil automatisiert über Compiler, manuell aufwändig
- Mehr Cache hilft nur wenn das Problem nicht schon in den Cache passt
- Skalierung auf hohe Kern-Anzahl ggf. schwierig (Amdahl's Law)

### ⇒ Gestiegen Anforderungen an Softwareentwicklung

## Quellen

---

- [1] Bringschulte, U.; Ungerer, T., *Mikrocontroller und Mikroprozessoren.*, Springer, 2002. - S. 272-300
- [2] Hennessy, J. L.; Patterson, D. A., *Computer Architecture – A Quantitative Approach*, Morgan Kaufmann Publishers, 2003 - 3rd ed. - pp. 196- 214
- [3] Fischer T. et al., *Design Solutions for the Bulldozer 32nm SOI 2-Core Processor Module in an 8-Core CPU*, ISSCC 2011
- [4] Weiss D. et al., *An 8MB Level-3 Cache in 32nm SOI with Column-Select Aliasing*, ISSCC 2011
- [5] Peter Greenhalgh, *Big.LITTLE Processing with ARM Cortex™-A15 & Cortex-A7*, whitepaper ARM, 2011