technische universität
dortmund

fakultät für informatik
informatik 12

Rechnerarchitektur (RA)

Sommersemester 2020

Jian-Jia Chen
Informatik 12
jian-jia.chen@cs.tu-..
http://ls12-www.cs.tu-dortmund.de/daes/
Tel.: 0231 755 6078

Jens Teubner
Informatik 6
jens.teubner@cs.tu-….
http://dbis.cs.tu-dortmund.de/
Tel.: 0231 755 6481

Willkommen!

2020/04/20

Organisatorisches

Übungen I

	Zeit	Raum	Dozenten
Vorlesung	Mo. 10-12	SRG1, 1001	Prof. Jian-Jia Chen, Prof. Jens Teubner
	Di. 12-14	OH12, E003	
Übung	Mo. 16.15-17.45 (deutsch) - Gruppe RA1	OH16, CILAB (Raum U08)	Christian Hakert
	Mi. 10.15-11.45 (deutsch) - Gruppe RA2		Marcel Ebbrecht
	Do. 10.15-11.45 (deutsch) - Gruppe RA3		Christian Hakert
	Do. 14.15-15.45 (englisch) - Gruppe RA4		Christian Hakert

Anmeldung über ASSESS: https://ess.cs.tu-dortmund.de/ASSESS/

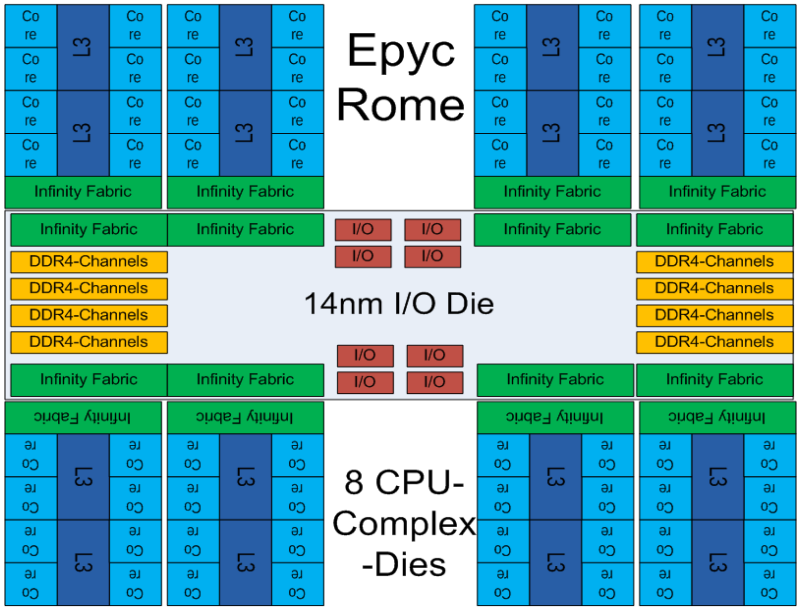
Deadline: 26.04.2020 – 12:00 Uhr
Erste Übung am 27.04.2020

Übungen II

Typischer Übungszettel:

- 20-30% Theorie
- 70-80% Praxis

Aufgaben werden in der Übung gelöst!



AMD EPYC CPU

Themen:

- Parallele Programmierung
  - Pthread, OpenMP und OpenCL
- Simulation (Leistungs-, Energie-, Temperatur-auswertung)
  - Sniper, McPAT und Hotspot
- Many-core Systeme
  - Numa Latenzen, Cache sharing
- Cache benchmarking

Vorlesungen

- Mo 10-12 (SRG1, 1001)
- Di 12-14 (OH12, E003)

Zuerst: BigBlueButton am LS12

https://ls12-bbb.cs.tu-dortmund.de/b/chr-utf-qvt

Materialien zur Vorlesung

Web:

- https://ls12-www.cs.tu-dortmund.de/daes/de/lehre/lehrveranstaltungen/summersemester-2020/rechnerarchitektur-deutsch.html

Bücher

- Hennessy, John L., Patterson, David A.: *Computer Architecture – A Quantitative Approach*, Morgan Kaufman, 4. Auflage 2006 & 5. Auflage, 2011

Prüfung / Leistungsnachweis

- Probeklausur geplant (Information kommt später)
- Prüfung (Bachelor): Klausur über Inhalte von Vorlesung und Übungen (benotet), 8 Credits:
  - 1. Klausur: 06.08.2020, 11.00-13.00,
  - 2. Klausur:15.09.2020, 11.00-13.00
- Sonderregelungen für Nebenfachstudierende: wenn zwingend erforderlich nach Absprache

Inhalte

- die Architektur eines einzelnen Rechners betrachten
- die fortgeschrittenen Konzepte von Rechensystemen erklären
  - Heterogene Konzepte
  - Speicherhierarchie
- die globale Sicht auf Parallelprogrammierung, Rechenzentren und Netze benutzen und optimieren

Bezüge zu anderen Veranstaltungen

Rechnerstrukturen (RS): RA baut auf RS auf

- Stoff aus RS wird als bekannt vorausgesetzt (Gatter, Binäre Logik, Zahlenrepräsentation, ...).
- Stoff aus RS wird vertieft (*multicores, grid, cloud*, Netze, ...)

Betriebssysteme (BS): Ressourcenkonzepte, Virtualisierung, Multitasking

Master:

Betriebssystembau (Master): HW/SW-Schnittstelle, low-level-Programmierung

Software ubiquitärer Systeme: Schichtenübergreifendes Systemverständnis, Energieaspekte, Ressourcenverwaltung

Data processing on modern hardware (Teubner): Ausnutzung von Hardwarestrukturen und Beschleunigern für Datenverarbeitung

Compilerbau: Befehlssatzarchitekturen, Optimierungen (Backend)

Gegenstand des Kurses RA
– Definitionen von „Rechnerarchitektur" –

Def. (nach Stone): *The study of computer architecture is the study of the* ***organization and interconnection of components*** *of computer systems. Computer architects construct computers from* ***basic building blocks*** *such as memories, arithmetic units and buses.*

From these building blocks the computer architect can construct ***any one of a number of different types of computers****, ranging from the smallest hand-held pocket calculator to the largest ultra-fast super computer. The functional behaviour of the components of one computer are similar to that of any other computer, whether it be ultra-small or ultra-fast.*

Gegenstand des Kurses RA
– Definitionen von „Rechnerarchitektur” –

Nach Stone ..

By this we mean that a memory performs the storage function, an adder does addition, and an input/output interface passes data from a processor to the outside world, regardless of the nature of the computer in which they are embedded.

The major differences between computers lie in the way the modules are connected together, and the way the computer system is controlled by the programs. ***In short, computer architecture is the discipline devoted to the design of highly specific and individual computers from a collection of common building blocks.***

Gegenstand des Kurses RA
– Definitionen von „Rechnerarchitektur” –

Def. (nach Amdahl, Blaauw, Brooks):

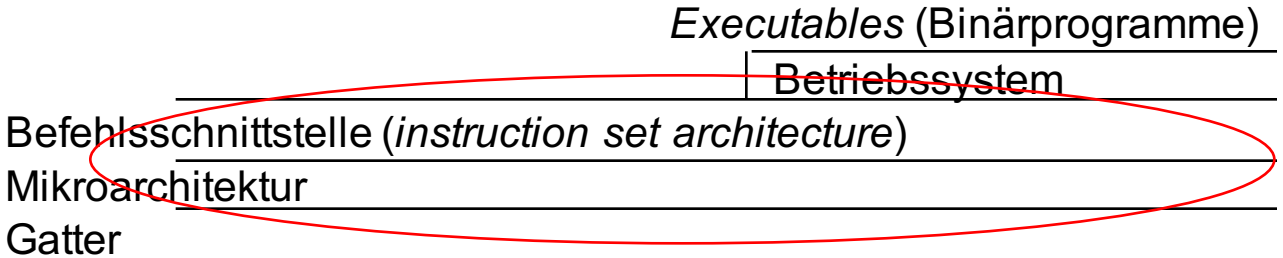
*The term architecture is used here to describe the attributes of a system as seen **by the programmer**, i.e., the conceptual structure and functional behaviour, as distinct from the organization and data flow and control, the logical and the physical implementation.*

Gegenüberstellung der Definitionen

Programmierschnittstelle	Interner Aufbau
Externe Rechnerarchitektur	Interne Rechnerarchitektur
Architektur	Mikroarchitektur
Rechnerarchitektur	Rechnerorganisation

Die externe Rechnerarchitektur definiert

- Programmier- oder Befehlssatzschnittstelle
- engl. *instruction set architecture, (ISA)*
- eine (reale) Rechenmaschine bzw.
- ein *application program interface (API)*.



Themenüberblick

1. Einleitung
2. Programmiermodelle

   Multithreading,Mehrprozessorsysteme, GPU, OpenMP, MPI, Netzwerke
3. Power-/energie-effizient
4. Mikroarchitektur

   instruction scheduling, Zuverlässigkeit, Sprungvorhersage, Superskalarität
5. Speicher
   - Speicherhierarchie, Flashspeicher
   - Transaktionale Speicher
   - Caching

Bei uns etwas breiter & mehr Betonung von Systemen im Vergleich zur Standard-Referenz Hennessy/Patterson*

*(*Kurs sollte mal Rechensysteme heißen→ Verwechselung mit RS)*

5% Überlappung mit „Eingebettete Systeme"

Wieso ist Verständnis von Rechnerarchitektur wichtig?

Zentral ist die Möglichkeiten und Grenzen des „Handwerkszeugs" eines Informatikers einschätzen zu können!

Grundverständnis wird z.B. benötigt:

- bei der Geräteauswahl,
- bei der Fehlersuche,
- bei der Leistungsoptimierung oder Benchmarkentwürfen,
- bei Zuverlässigkeitsanalysen,
- beim Neuentwurf von Systemen,
- bei der Codeoptimierung im Compilerbau,
- bei Sicherheitsfragen.

Wir dürfen keine Wissenslücken in zentralen Bereichen der IT haben!

Schichtenübergreifende Denkweise

Optimierungen erfordern Hardwarewissen

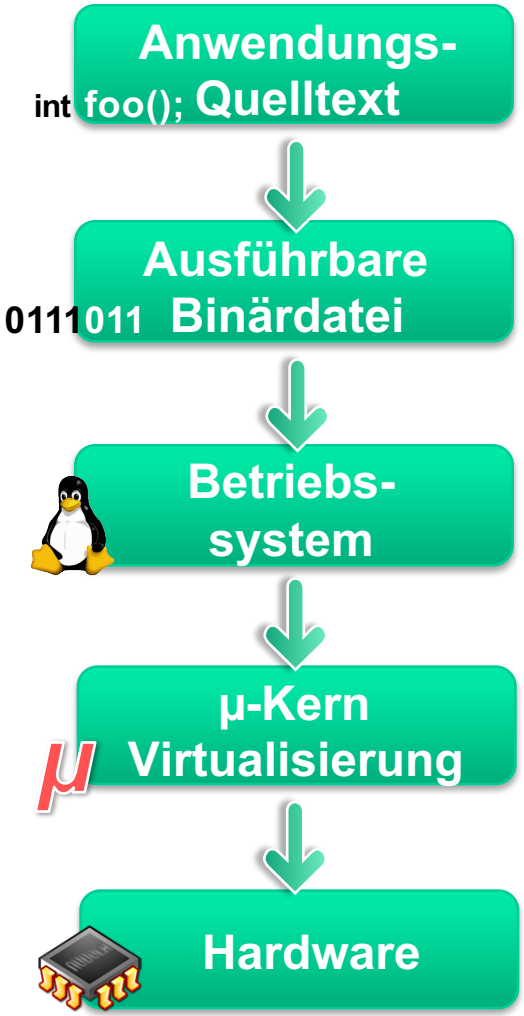
- Auswirkungen von Anwendungsverhalten auf Hardware kennen
- Auswirkungen von Hardwareverhalten auf Anwendungen kennen

(Wir müssen immer die Eigenschaften unserer Hardware bedenken)

Optimierungen erfordern Systemwissen

- Oft werden nur die Hochsprachenebene (z.B. Java) und darüber liegende Systemteile berücksichtigt
- Nichtfunktionale Eigenschaften (z.B. Energie) sind essentiell für den Entwurf effizienter Systeme

(Wir müssen immer die Konfigurationen unserer Software und die Hardwareeigenschaften beachten)



Performance-Optimierung

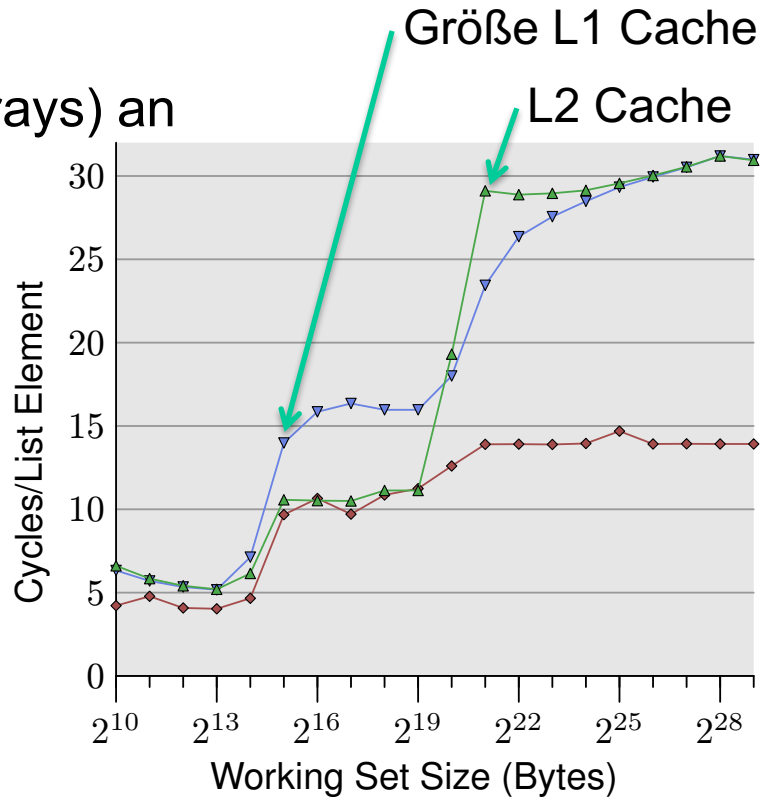
Speicherhierarchien haben großen Einfluß auf Performance

Beispiel: Cacheverhalten

- Operationen von Daten (z.B. Listen, Arrays) an Cacheparametern
- Zugriffszeiten unterscheiden sich um Faktor 6x zwischen Hierarchieebenen

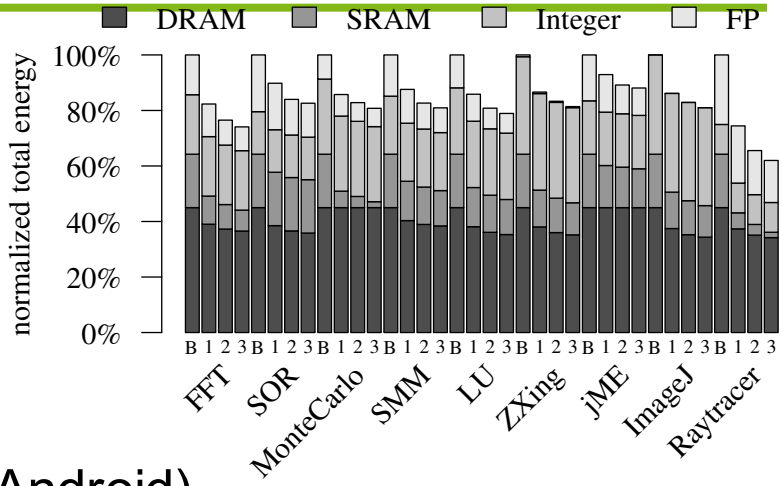
Beispiel: TLB-Verhalten

- Virtueller Speicher ermöglicht transparente Bearbeitung großer Daten
- Optimierungen vermeiden Auslagerung auf Festplatte (Kostenfaktor ~1000)



Ulrich Drepper. *What Every Programmer Should Know About Memory*, RedHat Inc., 2007
H. Kotthaus, I. Korb, M. Engel, P. Marwedel. *Dynamic Page Sharing Optimization for the R Language*, DLS 2014

Optimierung von Leistungsaufnahme und Energieverbrauch



Leistung und Energie in vielen Bereichen der Informatik:

Eingebettete und mobile Systeme
- Energieverbrauch von Mobilgeräten auch für Java-Entwickler wichtig (z.B. Android)

Hochleistungsrechnen (HPC)
- Leistungsaufnahme von Rechenzentren (Kühlung, Energiekosten)

Energieoptimierung mittels holistischer Ansätze
- Compiler-basierte Energieoptimierung
- Ausnutzung des *tradeoff* zwischen Berechnungsgenauigkeit und Energieverbrauch (Approximate computing)

Marwedel et al. Compilation techniques for energy-, code-size-, and run-time-efficient embedded software, IWACT'01
Sampson et al. *EnerJ: Approximate Data Types for Safe and General Low-Power Computation*. PLDI 2011.

Zuverlässigkeit

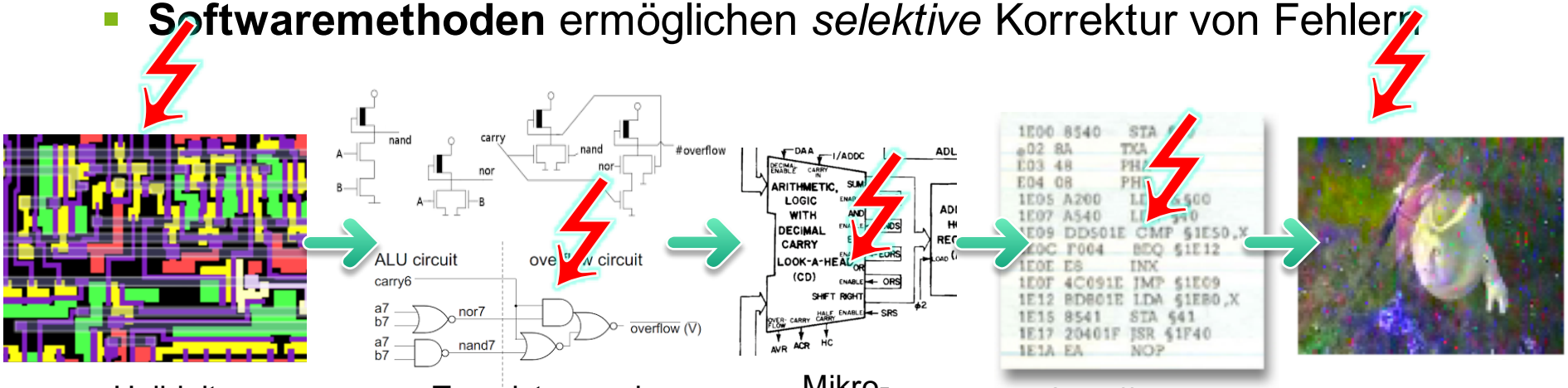
HW-Kosten zuverlässiger Systeme steigen mit sinkenden Halbleiter-Strukturbreiten und Versorgungsspannungen

Effiziente Zuverlässigkeit ermöglicht *profitable* Skalierung

- Reduktion der Fehlerkorrekturkosten oder der zu korrig. Fehler

SW-basierte Zuverlässigkeit erfordert *cross-layer* Wissen

- Anwendungswissen zur Bestimmung der *relevanten* Fehler
- **Softwaremethoden** ermöglichen *selektive* Korrektur von Fehlern



Halbleiter-Layout | Transistor- und Gate-Ebene | Mikro-architektur | ISA-Effekte und Beziehung zum Programmquelltext | Erkennbare (sicht-, hörbare) Effekte

Schmoll, Heinig, Marwedel, Engel. ***Improving the Fault Resilience of an H.264 Decoder using Static Analysis Methods*****. ACM TECS, 2013**

Gliederung (heute)

- Organisatorisches
  Materialien zum Kurs, Übungen,
  Leistungsnachweis
- Gegenstand des Kurses:
  Was ist Rechnerarchitektur?
- Bewertung von Rechnern
- Befehlssätze: RISC und CISC

1.2 Bewertung von Rechnern

Mehrere Kriterien

- Funktional: Befehlssatz, Speichermodell, Interruptmodell
- Preis
- Energieeffizienz (geringe elektrische Leistung)
- „Performanz”: (durchschnittliche) „Rechenleistung“
  zur Abgrenzung von der benötigten elektrischen Leistung hier Bevorzugung von „Performanz“ oder *Performance*
- Realzeitverhalten (*timing predictability*)
- Erweiterbarkeit
- Größe/Gewicht
- Zuverlässigkeit
- Sicherheit, ….

Standardmäßig betont.
Wir wollen die anderen Kriterien nicht ignorieren.

Fotos: P. Marwedel

Funktionale Eigenschaften

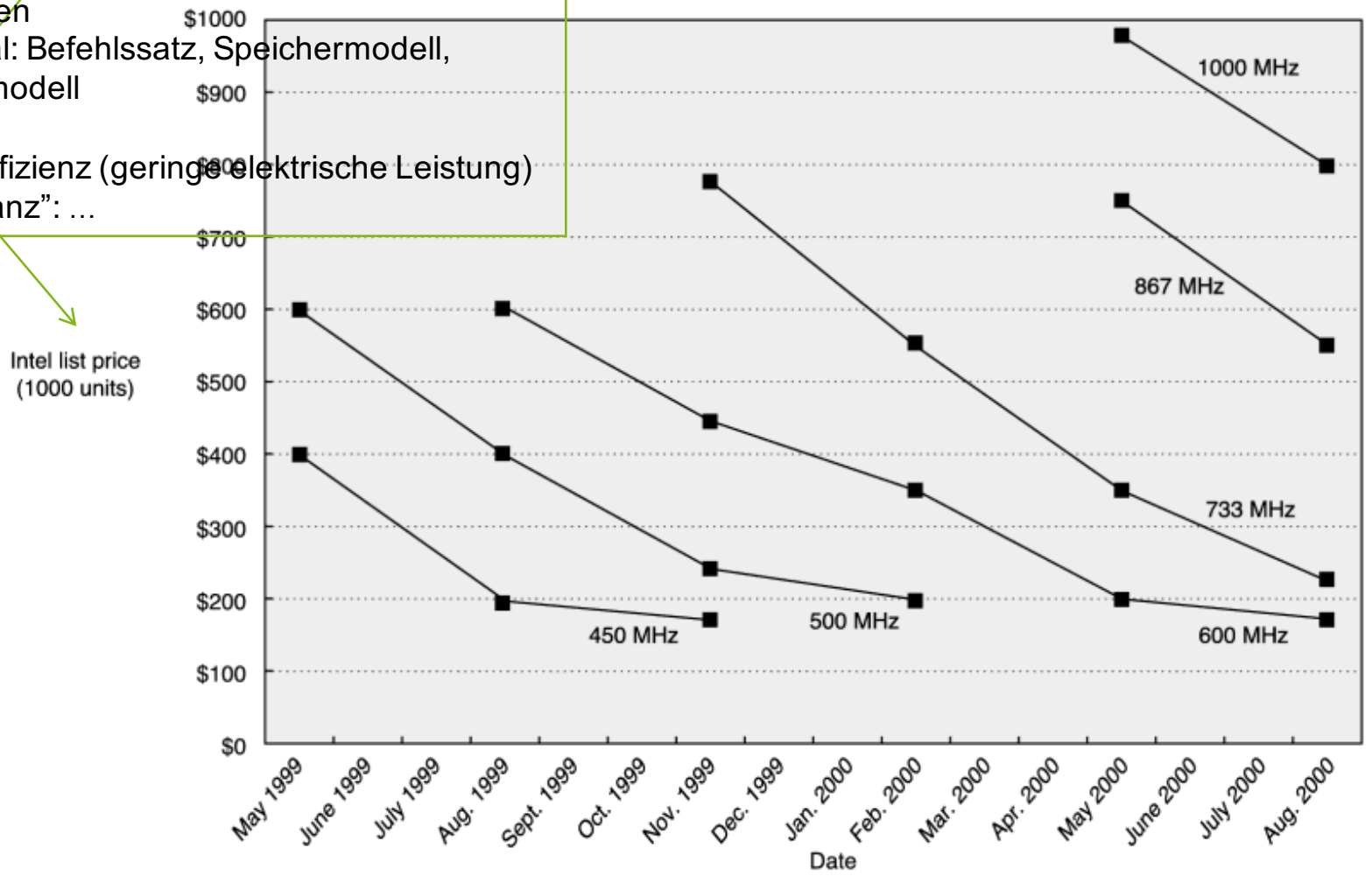
Die Funktion von Befehlssätzen kann wieder nach mehreren Kriterien bewertet werden:

- Operationsprinzip (Von-Neumann, Datenfluss, …)
- Addressbereiche (4 GB, usw.)
- Byte-Addressierbarkeit
- *Endianness*
- Orthogonalität
- n-Adressmaschine
- …

Mehrere Kriterien
- Funktional: Befehlssatz, Speichermodell, Interruptmodell
- Preis
- Energieeffizienz (geringe elektrische Leistung)
- „Performanz”: ...

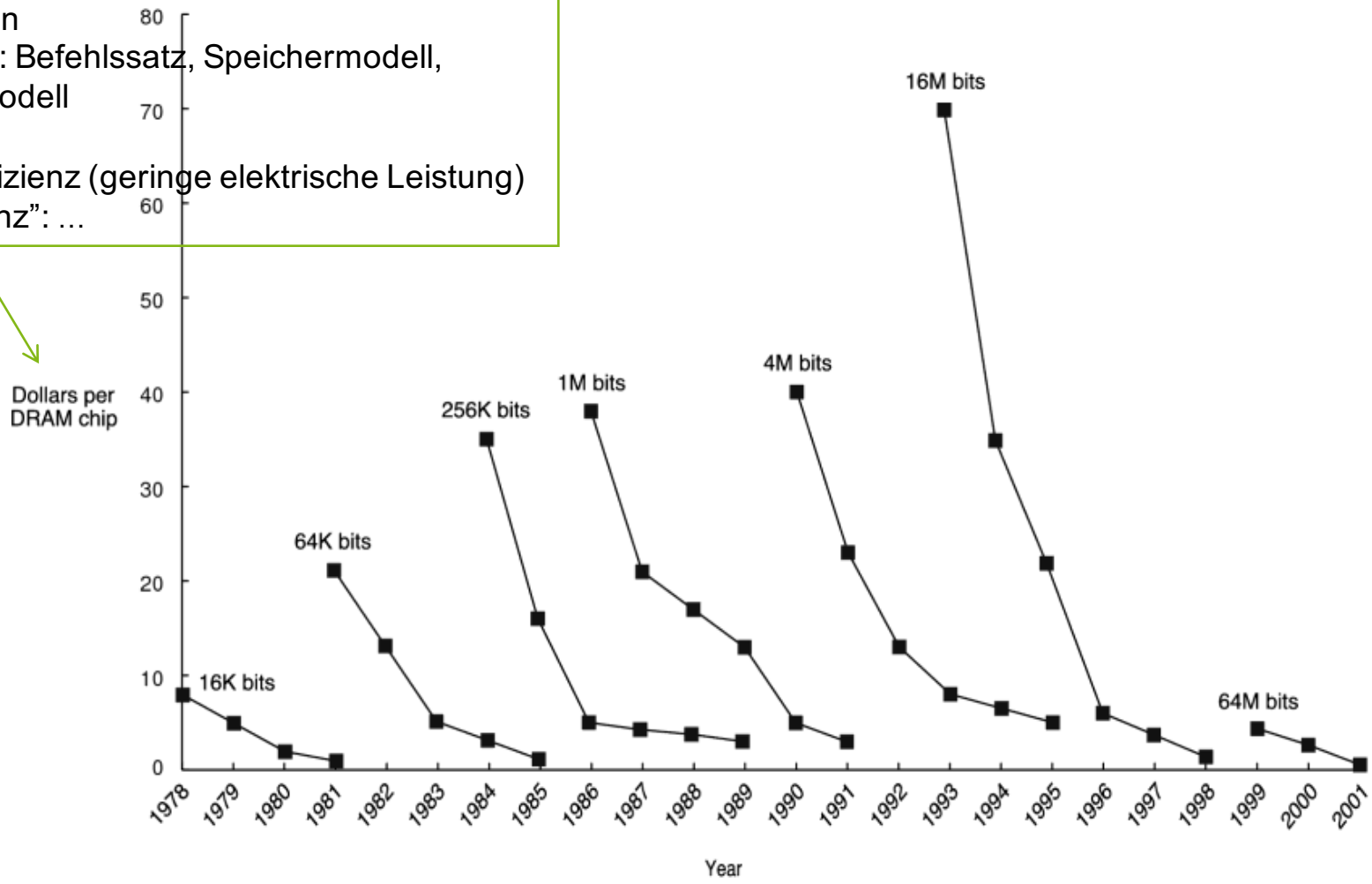
Preisentwicklung bei Mikroprozessoren

Mehrere Kriterien
- Funktional: Befehlssatz, Speichermodell, Interruptmodell
- Preis
- Energieeffizienz (geringe elektrische Leistung)
- „Performanz": ...

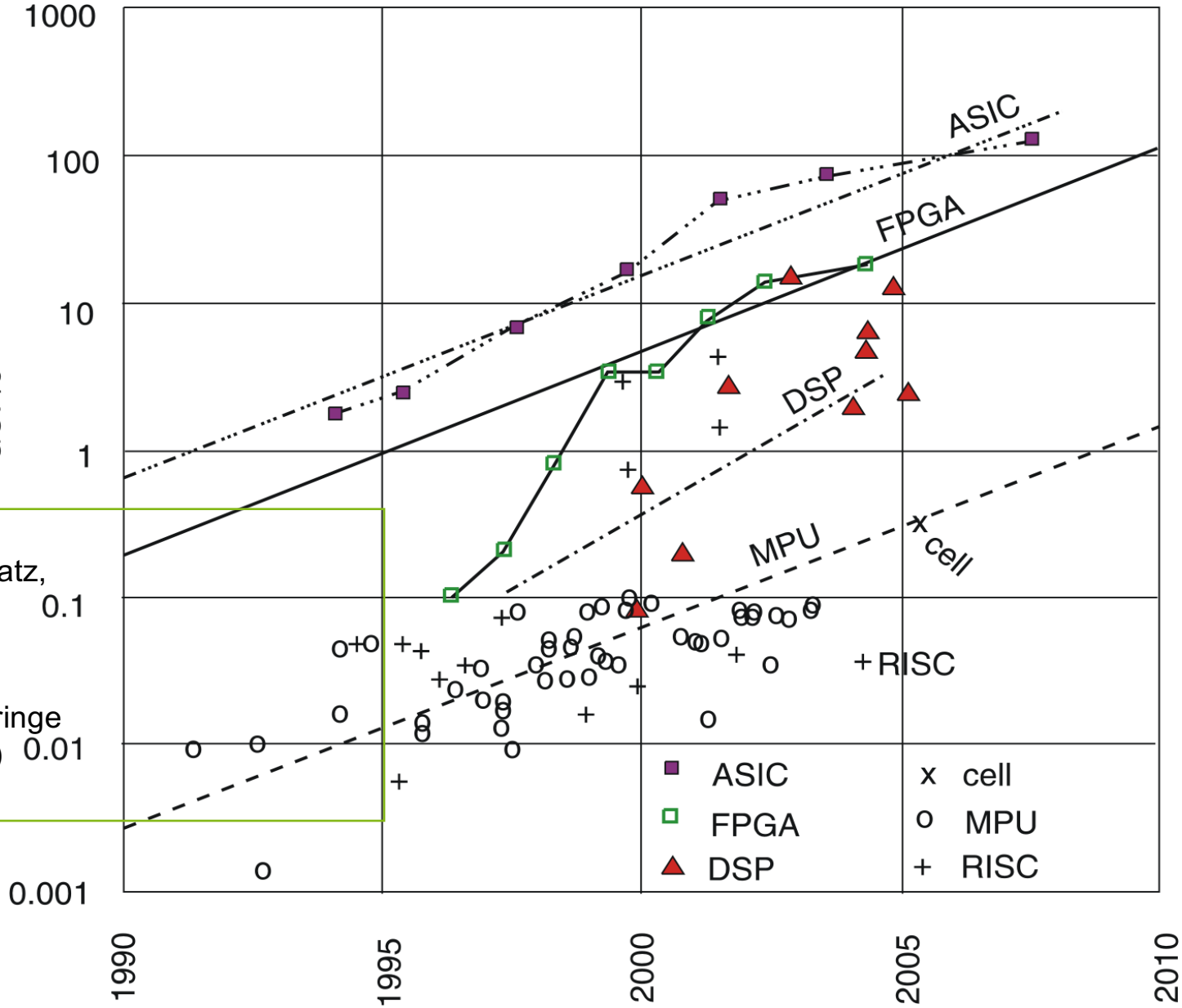


... und bei Speichermodulen

Mehrere Kriterien
- Funktional: Befehlssatz, Speichermodell, Interruptmodell
- Preis
- Energieeffizienz (geringe elektrische Leistung)
- „Performanz": ...



Energie-effizienz



Mehrere Kriterien
- Funktional: Befehlssatz, Speichermodell, Interruptmodell
- Preis
- Energieeffizienz (geringe elektrische Leistung)
- „Performanz": ...

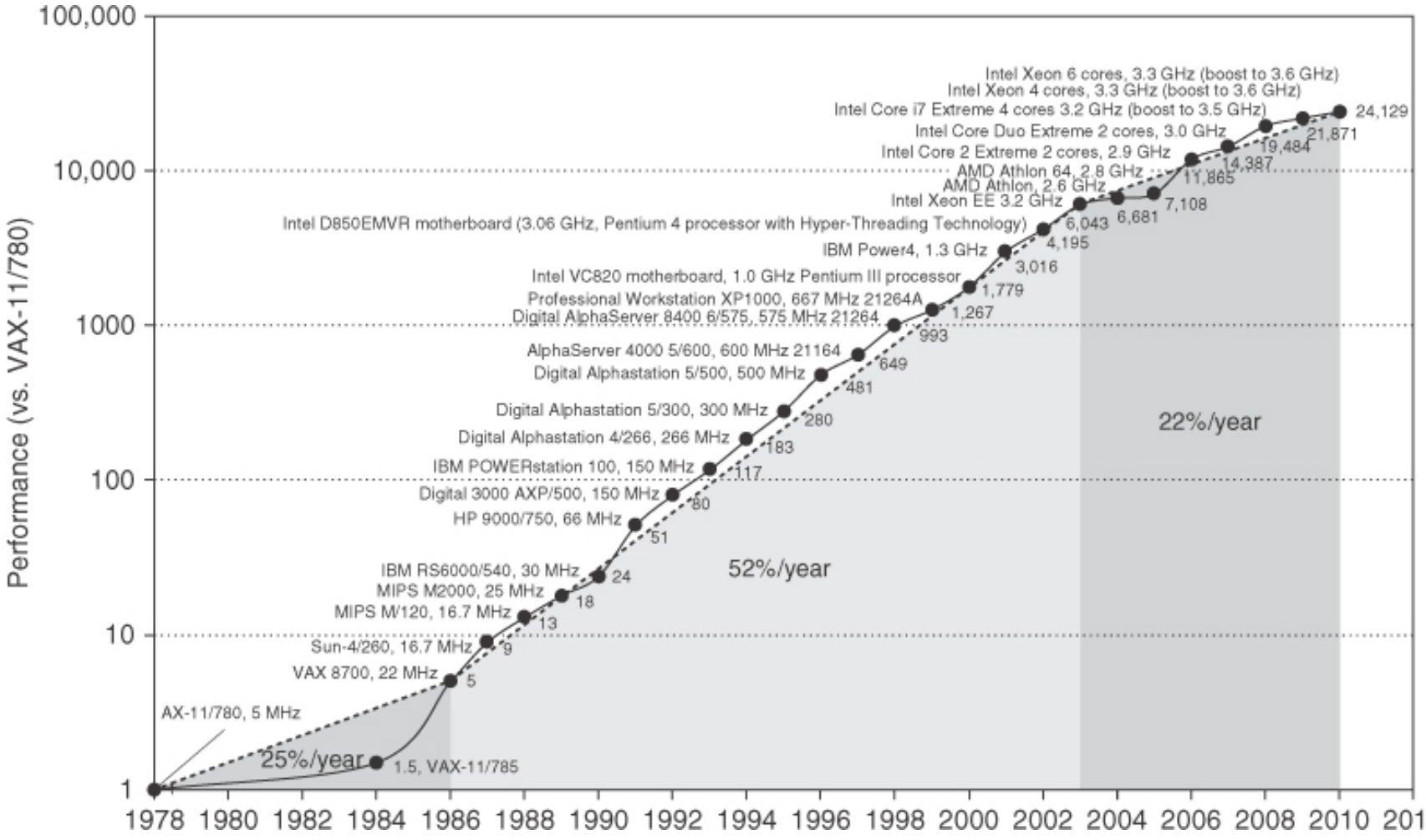
© Hugo De Man, IMEC, Philips, 2007

Performanz

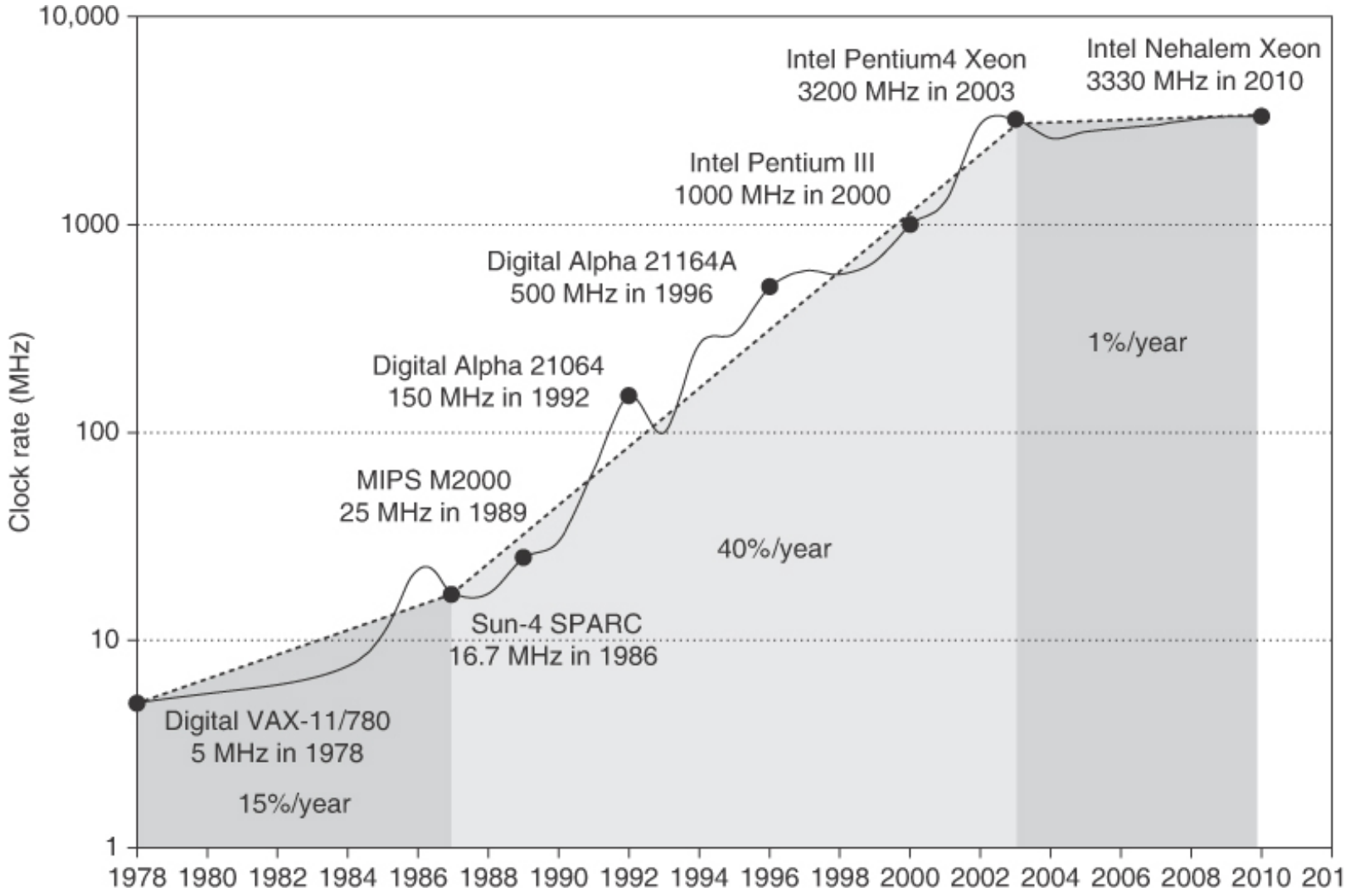
Mehrere Kriterien
- Funktional: Befehlssatz, Speichermodell, Interruptmodell
- Preis
- Energieeffizienz (geringe elektrische Leistung)
- „Performanz": …

- Das Mooresche Gesetz (nach Gordon Moore, Mitbegründer von Intel, 1965)
  „Die Anzahl der auf einem Chip integrierten Transistoren verdoppelt sich alle 18 Monate!"
- Anforderungen aus der Software: Nathans erstes Softwaregesetz (nach Nathan Myhrvold, Microsoft)
  „Software ist ein Gas. Es dehnt sich aus und füllt den Behälter, in dem es sich befindet."
- Anforderungen aus Anwendungen in der Telekommunikations- und Netzwerktechnik, *Video-on-Demand, Multi-Media-Messaging*, mobiles Internet

Performanzentwicklung bei Mikroprozessoren



Entwicklung der Taktrate von Mikroprozessoren



Beurteilung der Performanz bzw. „Rechenleistung“

Was bedeutet „Rechner A ist schneller als Rechner B”?

- Benutzersicht: Antwortzeit
  (bei Bearbeitung einer Aufgabe)
- Serviceanbietersicht: Durchsatz
  (Anzahl Aufgaben, die pro Zeiteinheit bearbeitet werden)

Zentrale Messgröße für ***beide Sichten: Ausführungszeit!***

Performance-Maße

Verschiedene Definitionen von Ausführungszeit:

- **Laufzeit** (*wall-clock time / elapsed time*)
  = Gesamtlaufzeit inkl. I/O, Speicherzugriffen, Betriebssystemoverhead, ggf. weitere Systemlast

  Wartezeiten im Mehrprogrammbetrieb von anderen Prozessen nutzbar

- **CPU-Zeit**
  - Wartezeiten / Bearbeitung anderer Prozesse werden *nicht* berücksichtigt
- **user CPU-Zeit**
  - Nur Programmabarbeitung, *nicht* Betriebssystemdienste
    z.B. unter Unix: `time <Kommando> ...`
    `0.09u 0.07s 0:01.74 9.1%`

Performance-Maße II

Sinnvolle *Performance*-Definitionen:

- System-*Performance*, d.h. Laufzeit in einem unbelasteten (*unloaded*) System (d.h. kein Mehrprogrammbetrieb)
- CPU-*Performance*, d.h. [*user*] CPU-Zeit (Betrachtung unabhängig von I/O [und Betriebssystem])

Wovon Lauf-/CPU-Zeiten messen?

Immer reale Programme!

Es existieren *Performance*-Definitionen, die nicht auf Zeitmessung bzw. Ausführung realer Programme basieren!

Programmauswahl zur *Performance*-Bewertung

Aufgabe von Rechnern selten eindeutig definiert
(i.d.R. nicht: „Ein bestimmtes Programm wird immer ausgeführt")

☞ Performanz im realen Betrieb muss vorhergesagt/ geschätzt werden!

Dhrystone does not use floating point. Typical programs don't …

(R. Richardson, '88)

This program is the result of extensive research to determine the instruction mix of a typical Fortran program. The results … on different machines should give a good indication of which machine performs better under a typical load of Fortran programs. The statements are purposely arranged to defeat optimizations by the compiler.

(H.J.Curnow & B.A.Wichmann, '76)

Instruction set architectures (ISAs)

2.1 RISC und CISC

- *Reduced instruction set computers (RISC)* (1)-

Wenige, einfache Befehle wegen folgender Ziele:

- Hohe Ausführungsgeschwindigkeit
  - durch kleine Anzahl interner Zyklen pro Befehl
  - durch Fließbandverarbeitung (siehe Kap. 3)

Def.: Unter dem **CPI-Wert** (engl. ***cycles per instruction***) einer Menge von Maschinenbefehlen versteht man die mittlere Anzahl interner Bus-Zyklen pro Maschinenbefehl.

RISC-Maschinen: CPI möglichst nicht über 1.
CISC-Maschinen (s.u.): Schwierig, unter CPI = 2 zu kommen.

Programmlaufzeit = Dauer eines Buszyklus * Anzahl der auszuführenden Befehle * CPI-Wert des Programms

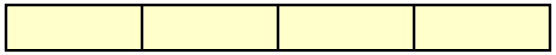
Wiederholung aus RS

Klassifikation von Befehlssätzen

- *Reduced instruction set computers (RISC)* (2)-

Eigenschaften daher:

- feste Befehlswortlänge



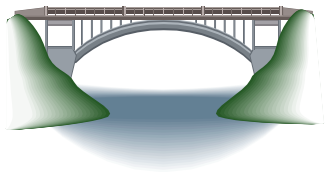
- LOAD/STORE-Architektur!

ld $2,..; ld $3,..; add $3,$3,$2; sw $3

- einfache Adressierungsarten

z.B. keine indirekte Adr.

- „semantische Lücke" zwischen Hochsprachen & Assemblerbefehlen durch Compiler überbrückt.



- Statt aufwändiger Hardware zur Beseitigung von Besonderheiten (z.B. 256 MB-Grenze bei MIPS, 16 Bit Konstanten) wird diese Aufgabe der SW übertragen.
- Rein in HW realisierbar („mit Gattern und Flip-Flops“)

Wiederholung aus RS

Complex instruction set computers (CISC)

Complex instruction set computers (CISC)

Entstanden in Zeiten schlechter Compiler & großer Geschwindigkeitsunterschiede Speicher / Prozessor

- Befehle sollten möglichst nahe an den Hochsprachen sein (keine semantische Lücke)
- Mit jedem geholten Befehl sollte der Prozessor viel tun
  - sehr komplexe Befehle

Wiederholung aus RS

Complex instruction set computers (CISC)
- Eigenschaften -

Relativ kompakte Codierung von Programmen

Für jeden Befehl wurden mehrere interne Zyklen benötigt

☞ Die Anzahl der Zyklen pro Befehl (der ***cpi***-Wert) war groß

- (Mikro-) Programm zur Interpretation der Befehle nötig
- Compiler konnten viele Befehle gar nicht nutzen

Wiederholung aus RS

Zusammenfassung

- Organisatorisches
- Definitionen zur Rechnerarchitektur
- Bewertung von Rechnern
- ISAs
  - RISC, CISC