Vorlesung Rechnerarchitektur WS 07/08

Prof. Dr.-Ing. Eike Jessen (em.)
jessen@in.tum.de
Lehrstuhl für Netzwerkarchitekturen
Lehrstuhl für Rechnertechnik und
Rechnerorganisation/Parallelrechnerarchitektur

Zur Vorlesung

- Zeit und Ort: Dienstag, 9.30 bis 12.00 Uhr (2 Pausen!), Raum MI 00.08.038
- Information, Folien: http://www.net.informatik.tu-muenchen.de//teaching/WS07/comparch/
- Ziel: Folien werden als PDF spätestens am Tag der Vorlesung bereitgestellt
- Ältere Unterlagen zur Vorlesung Rechnerarchitektur:

- Gerndt 2006/2007: www.lrr.in.tum.de/~gerndt/home/Teaching/WS2006/Rechnerarchitektur/Rechnerarchitektur.htm

- Jessen 2000/2001: www.net.in.tum.de/teaching/former/jessenscripts.html

Keine Übungen zur Vorlesung!

Studienbegleitende Prüfung (wer muss eine machen? Bitte Mail bis 21.1.2008 an jessen@in.tum.de) ab 18.2.2008; Prüfer: Jessen

Herr Jessen ist erreichbar:

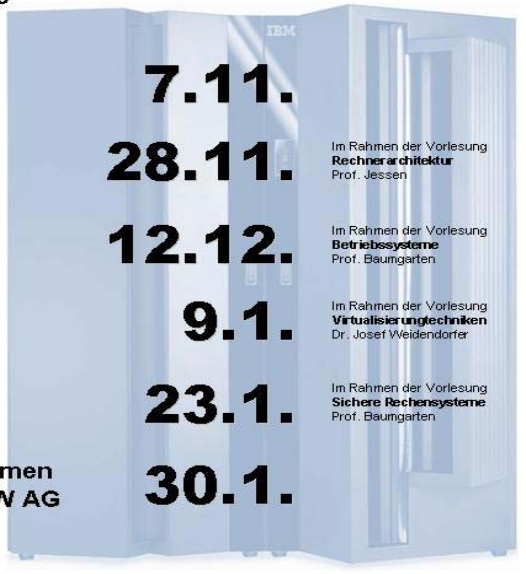
- in den Vorlesungspausen
- nach Verabredung (jessen@in.tum.de, Tel: 089-289-17076) im Raum 02.13.041

Ringvorlesung

Mainframe Technologie Heute

IBM System z

Jeweils
Mittwoch, 16:15 im HS 3



Eröffnungsvortrag Anschließend Buffet	**7.11.**	
Architektur Klaus-Dieter Müller Technology Development, IBM	**28.11.**	Im Rahmen der Vorlesung **Rechnerarchitektur** Prof. Jessen
Betriebssysteme Matthias Bangert System z IT Specialist, IBM	**12.12.**	Im Rahmen der Vorlesung **Betriebssysteme** Prof. Baumgarten
Virtualisierung Michael Störchle System z Specialist, IBM	**9.1.**	Im Rahmen der Vorlesung **Virtualisierungtechniken** Dr. Josef Weidendorfer
IT-Sicherheit Thomas Hanicke Systems Architect, IBM	**23.1.**	Im Rahmen der Vorlesung **Sichere Rechensysteme** Prof. Baumgarten
Einsatz im Unternehmen am Beispiel der BMW AG Wolfgang Pielmeier Leiter Server Betrieb, BMW AG	**30.1.**	

Die Vorlesungsreihe wird im Sommersemester fortgesetzt mit einem Bachelorpraktikum zum Thema
Industrieeinsatz des System z

Technische Universität München

Vortragende: Experten von IBM / BMW
Veranstalter: Prof. Dr. Uwe Baumgarten
Prof. Dr. Michael Gerndt

Eike Jessen

- Studium Nachrichtentechnik TU Berlin bis 1960
- Forschung Computer/Radar-Integration
- Promotion (Assoziative Speicher/Prozessoren) 1964
- Entwicklungsleiter Rechner/Großrechner AEG-Telefunken Konstanz, bis 1972
- Professor für Informatik Universität Hamburg / Universität der Bundeswehr / TU München (1983-2002)
- Vorsitzender Deutsches Forschungsnetz (bis 2005)
- Vizepräsident der TU München 1994 – 1996
- Schwerpunkte: Leistungsanalyse, Netze, Computer

Übersicht, Gliederung

Einführung: Ort, Zeit, Dozent, Zweck, Übersicht

1. Grundlagen

   Architektur, von Neumann, Interpretation und Ausschnitte, Funktionseinheiten und Aufträge, Beispiele, Leistungsbewertung, Geschichte und Trends

2. Zentralprozessoren

   Abgrenzung und Grunderscheinungen, Schichtung, Beispiele (Pentium, UltraSparc, Itanium), Befehlssatzarchitektur (Datentypen, Befehlsformate/-typen, EPIC, Adressierung, Unterbrechungen), Mikroarchitektur (Caches, Pipelining, Superskalarität, Out-of-Order Processing, Threading), Sonderformen (Vektor-/ Feldprozessoren, Signalprozessoren, andere)

3. Speicher

   Speicherhierarchie, Caches, Hauptspeicher, Periphere Speicher (Magnetik, Optik, Halbleiter)

4. Bussysteme

5. Ein-/Ausgabe

6. MIMD

   Multiprozessoren, Multirechner, Parallelrechner, Grids

Ziele der Vorlesung

- Grundkenntnisse in Rechnerarchitektur
- Wichtige Vorkenntnisse für Kurse zu Zuverlässigkeit und Fehlertoleranz, Hochleistungsarchitekturen, Parallel Programming, Grid Computing, Betriebssysteme, Rechnernetze und Verteilte Systeme, Rechensysteme in Einzeldarstellungen, Compiler
- Verständnis der Wechselwirkung zwischen Technologie, Rechnerkonzepten und Anwendung
- Grundlagen für Rechnerauswahl und -entwurf

Voraussetzungen

- Einführung in die Technische Informatik

Literatur

- Hennessy, J.L., Patterson, D.A.: Computer Architecture – A Quantitative Approach, 4. Auflage, Morgan Kaufmann, 2007
- Tanenbaum, A.S.: Structured Computer Organisation, 5. Auflage, Pearson Prentice Hall, 2006

Über beide Bücher ist sehr viel weitere Literatur erreichbar

1. Grundlagen

1.1 Architektur

„architekton“ ist der Oberhandwerker.

Der Begriff Rechnerarchitektur (computer architecture) soll bei uns verstanden werden wie bei (Bode A 90)

- allgemeine Strukturlehre mit ihren Hilfsmitteln

- Ingenieurwissenschaftliche Disziplin, die bestehende und künftige Rechenanlagen beschreibt, vergleicht, beurteilt, verbessert und entwirft

- Betrachtet dabei Aufbau und Eigenschaften des Ganzen (der Rechenanlage), seiner Teile (Komponenten) und der Verbindungen (Globalstruktur, Infrastruktur)

Wichtiger anderer Begriff, vor allem im Amerikanischen meist als architecture bezeichnet (Blaauw 1964):

„attributes of a system as seen by the programmer, i.e. the conceptual structure and functional behaviour, as distinct from the organisation and data flow and control, the logical, and the physical implementation“

Hierfür wird in der Vorlesung Funktionalität gesagt. Die Abstraktion/ Virtualisierung durch den Blaauwschen Architekturbegriff ist in der Informatik äußerst wichtig und auf allen Ebenen einsetzbar: Sprache, Prozessor, Rechenwerk, Addierer, Schalter, …; vor allem auch hierarchisch!

Architektur (Funktionalität) kann extrem langlebig sein, z.B. IBM /360-Architektur (1965) als Teilfunktionalität in verschiedenen Implementationen seit über 40 Jahren aufrecht erhalten, Intel 8080-Architektur (1974) seit über 30 Jahren, jeweils in sehr verschiedenen Implementationen:

- Zeitliche Generationen (z.B. /360, /370, /390, zSeries)

- Familien mit Mitgliedern verschiedener Leistung und Anwendung (z.B. PC/embedded/Server/Mobil-Prozessoren)

Architekturbegriff in Systemen oft:

Menge von (Klassen von) Funktionseinheiten, jeweils durch Funktionalität und Interoperabilität beschrieben, aus denen Systeme einer bestimmten Klasse gebildet werden können, z.B.

- Netzarchitekturen
- Managementarchitekturen
- Datenbankarchitekturen

Rechnerarchitektur ist -mehr als andere Informatikgebiete- ein etwas naives Gebiet:

- Die unglaubliche Entwicklungsgeschwindigkeit der Computer ist weit überwiegend der Technologie zu danken
- Es gibt wenige Strukturprinzipien der Rechnerarchitektur, die lediglich variiert und neu kombiniert werden
- Hersteller und Wissenschaftler produzieren einen Strom von neuen Bezeichnern, die ängstlich nachgesprochen werden, hinter denen aber oft nur Wiederbelebungen und Pseudoinnovationen stehen
- Mit missionarischem Eifer werden in der Wissenschaft Konzepte verfolgt, für die es nur schwache Motive gibt, z.B. non-von-Maschinen, Konnektionismus

1.2 Funktionseinheiten und Aufträge

Funktionseinheit FE (functional unit): Durch Aufgabe oder Wirkung abgrenzbares Gebilde (DIN 44300).

Bei uns meist durch Funktionalität definierbar

- Absichtlich black box: exakte Spezifikation ohne Einsicht in die Realisierung anzustreben!
- Rechner, Transistor, Compiler (Wirkung nur bei Ausführung!)

System (system): Funktionseinheit, die als aus Funktionseinheiten bestehend aufgefasst wird.

- Also keine black-box-Perspektive!
- Rechner, …, Rechensystem (d.h. Rechner mit Betriebssoftware)

Auftrag (task (!)): Verpflichtung einer FE zu einer Handlung.

- Rechenauftrag (job), Speicherauftrag, …

Verweilzeit y (latency, delay, sojourn time) eines Auftrags in einer FE: Zeitdauer von Auftragsübergabe /-übernahme bis Ende der durch den Auftrag bestimmten Handlung in der ausführenden FE.

- Zugriffszeit (Lesen, Schreiben, Löschen), Programmlaufzeit (Rechenauftrag an Rechensystem)
- Verweilzeit hängt offenbar ab von Auftrag und Funktionseinheit (Art, Zustand); besonders wichtig: beschäftigt sich die FE sofort und nur mit diesem Auftrag?

Auftragsbestand, Füllung f (population): Zahl der Aufträge, die aktuell ihre Verweilzeit in der betrachteten FE zubringen.

- Keineswegs nur 0 oder 1! Rechensysteme können viele Rechenaufträge aktuell übernommen, aber nicht erledigt haben. Rechnernetze können Tausende von Paketen gleichzeitig transportieren. Ein superskalarer Pipeline-Prozessor kann 50 Operationen zu einem Zeitpunkt in Arbeit haben.
- Es ist für unsere Definition gleichgültig, ob die Aufträge gleichzeitig oder im zeitlichen Wechsel verarbeitet werden.
- Abgeleitete Begriffe:

  f=0 → FE ist frei (idle), f>0 FE ist beschäftigt (busy). Ist stets f≤1, dann liegt serieller Betrieb (serial operation) vor.

Der Verweilzeitbegriff ist brutto (black box), er sagt nichts über einen Netto-Zeitbedarf aus. Um trotzdem einen Begriff zu verwenden, der die Konkurrenz durch Bearbeitung anderer Aufträge heraushält, definiert man:

Bedienzeit b (service time): Verweilzeit eines Auftrags bei Füllung 1

- In Rechnern ist typisch, dass ein einzelner Auftrag ohne Verzug erledigt wird (vgl. aber Autos vor Verkehrsampel an leerer Kreuzung).
- Also Bedienzeiten: Zugriffszeit zu Register oder Speicher (bei nur einem Zugreifer), Additionszeit, Laufzeit über einen Telefonie-kanal.
- Also keine Bedienzeit: Befehlsausführungszeit in einem Pipeline-Prozessor, Paketlaufzeit im Internet

Kapazität k (capacity) einer FE ist die (von der Art der Aufträge abhängige) maximale Füllung mit Aufträgen

- Speicherkapazität (Speicherauftrag), Kapazität einer CPU (z.B. höchstens 50 Befehle übernommen und noch nicht abgeschlossen)
- Eine FE mit k=1 heißt exklusiv oder einfach
- Eine FE, deren Füllung gleich ihrer Kapazität ist (f=k), heißt belegt (occupied)

Durchsatz d (throughput) einer FE: Zahl der von FE in einem Zeitintervall erledigten Aufträge, geteilt durch die Dauer des Zeitintervalls.

- Durchsatz hängt von der Art der Aufträge, der FE und dem Auftragszugangsprozess ab.
- Durchsatz der CPU: Befehle/s, Durchsatz eines Netzes: Pakete/s, Durchsatz einer Platte: Blöcke/s (oder: Bytes/s)

Grenzdurchsatz c (maximun throughput, bandwidth): durch Art der Aufträge bestimmter maximaler Durchsatz

- es wird vorausgesetzt, dass genügend viele Aufträge angeboten werden
- wie Bedienzeit und Kapazität wichtige Leistungsgröße einer FE!
- bei CPUs oft in Mips/Gips/Tflops angegeben (10^6 / 10^9 Operationen je Sekunde / 10^{12} Gleitpunktoperationen je Sekunde).

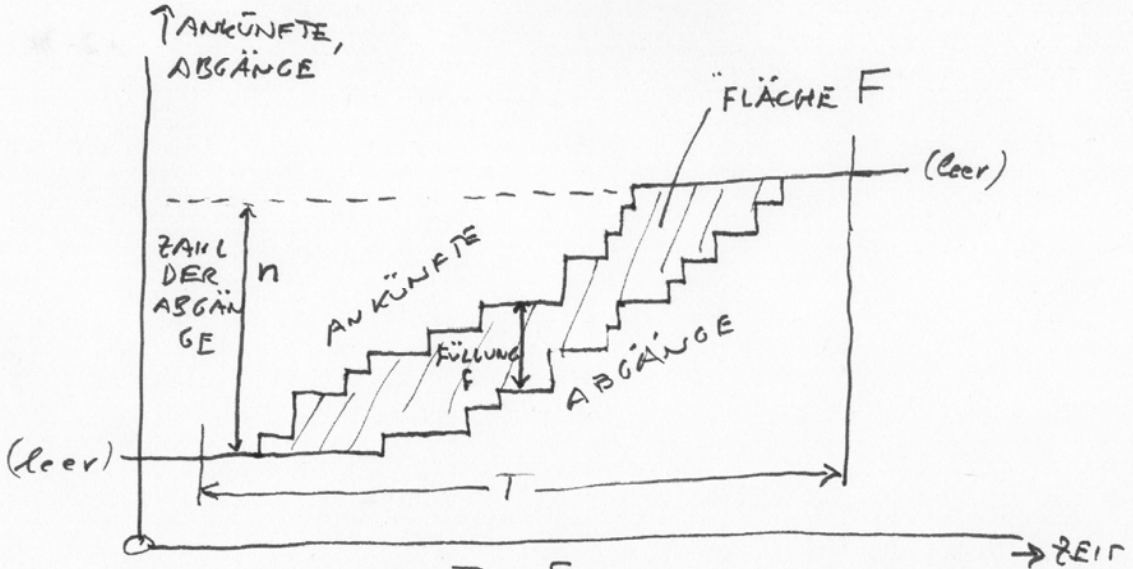
Auslastung ρ (rho) (utilisation) einer FE ist der Quotient von Durchsatz zu Grenzdurchsatz

- Bei exklusiven FE ist die Auslastung zugleich der relative Zeitanteil, in dem die FE belegt ist bzw. die Wahrscheinlichkeit, dass ein unabhängiger Beobachter die FE belegt findet.

Little‘s Formel: Wenn in einem Zeitintervall ebenso viele Aufträge zugehen wie abgehen (Flussgleichgewicht, flow balance), dann gilt zwischen den Mittelwerten der Füllung f und der Verweilzeit y der abgehenden Aufträge und dem Durchsatz d im Intervall f=y*d

Beispiel: Ein Netz transportiere 2000 Pakete/s. Die mittlere Laufzeit ist 10 ms. Also sind im Mittel 10 ms * 2000/s = 20 Pakete im Netz.

Beispiel: Die mittlere Ausführungszeit einer CPU-Operation sei 20ns, es soll ein Durchsatz von 2 Gips ($2*10^9$ Operations/s) erzielt werden. Also muss die CPU 20 ns x $2*10^9$ Op/s = 40 Op gleichzeitig in Ausführung haben. Verlängert sich die Ausführung durch die Komplexität der CPU um 20%, so ist der Durchsatz nur mit 20% mehr Operationen zu halten!



MITTLERE FÜLLUNG $\bar{f} = \frac{F}{T}$

DURCHSATZ $d = \frac{n}{T}$

MITTLERE VERWEILZEIT $\bar{y} = \frac{F}{n}$

LITTLE: $\bar{f} = \bar{y} \cdot d \cong \frac{F}{n} \cdot \frac{n}{T} = \frac{F}{T}$

Funktionseinheiten (und ihr Betrieb) kosten Geld. Die Kosten werden meist als „Kosten/Zeitraum“: Kostenrate κ (Kappa) aufgefasst:

- Miete (Kaufpreis evtl. auf Lebenszeit beziehen)
- Wartung
- Operating
- Energie
- Umgebung (Gebäude, Kühlung, …)

Die mittleren Kosten K je Auftrag hängen von Kostenrate κ und Auslastung ab:

$$K = \frac{\text{Kosten im Intervall T}}{\text{Zahl der erledigten Aufträge im Intervall T}}$$

$$= \frac{\kappa * T}{d * T} = \frac{\kappa}{c * \rho}$$

Also: bei gegebenem Grenzdurchsatz c werden die Kosten je Auftrag, K, durch Steigerung der Auslastung minimiert! Wenn wir fair sind, geben wir die durch die Steigerung der Auslastung eingetretene Kostensenkung an den Auftraggeber weiter.

Ein Tarif legt fest, wie die Kosten auf die Auftraggeber umgelegt werden, z.B. billiger Nachttarif trotz niedriger Auslastung!

Für den Auftraggeber ist aber die Verweilzeit seines Auftrags meist wichtiger als die Auftragskosten! Er hat Verweilzeitkosten, und Verweilzeit wächst bei großer Füllung i.a. mit der Auslastung! Wettbewerb des Auftragsbestandes um die Betriebsmittel!

Der Auftraggeber wird die hohe Auslastung nur dann akzeptieren, wenn die Senkung der Auftragskosten größer als seine zusätzlichen Wartekosten (der „Nutzenverlust“ seines Auftrags) ist. Realzeitforderung! (real time requirement). Also: Selektive Bedienstrategie (service strategy) erforderlich!

Aufträge bedingen sich häufig derart, dass erst die Fertigstellung von Auftrag A die Übergabe / Übernahme von Auftrag B erlaubt.

Man sagt dazu:

A ist präzedent (precedent) zu B, A < B.

Es darf nicht zugleich A < B und B < A sein, sonst Verklemmung (deadlock).

Ist A < B oder B < A, dann heißen A und B sequentiell zueinander (sequential), sonst nebenläufig (concurrent). Achtung! Andere Bedeutung bei Bode!

Gibt es für zwei Aufträge keinen Zeitpunkt, zu dem beide übernommen / übergeben, aber noch nicht erledigt sind, d.h. ihre Verweilzeiten überlappen sich, so heißen die Aufträge seriell (serial), sonst heißen sie kollateral (collateral).

- nur nebenläufige Aufträge können kollateral bearbeitet werden
- sequentielle und nebenläufige Aufträge können seriell bearbeitet werden.

Wir sind dem Bezeichner „parallel“ (parallel) zunächst ausgewichen, da er für nebenläufig und kollateral und Parallelrechner (das ist einer, der nebenläufige Teile eines Auftrages kollateral verarbeitet, zwecks Bedienzeitreduktion) und Konfigurationen verwendet wird (z.B. „parallele Kanäle“). Aber wenn keine Unklarheit besteht, sagt die Vorlesung parallel für kollateral.

Schließlich noch ein paar Begriffe, um die (Un)vollkommenheit technischer Funktionseinheiten zu beschreiben:

Eine FE ist

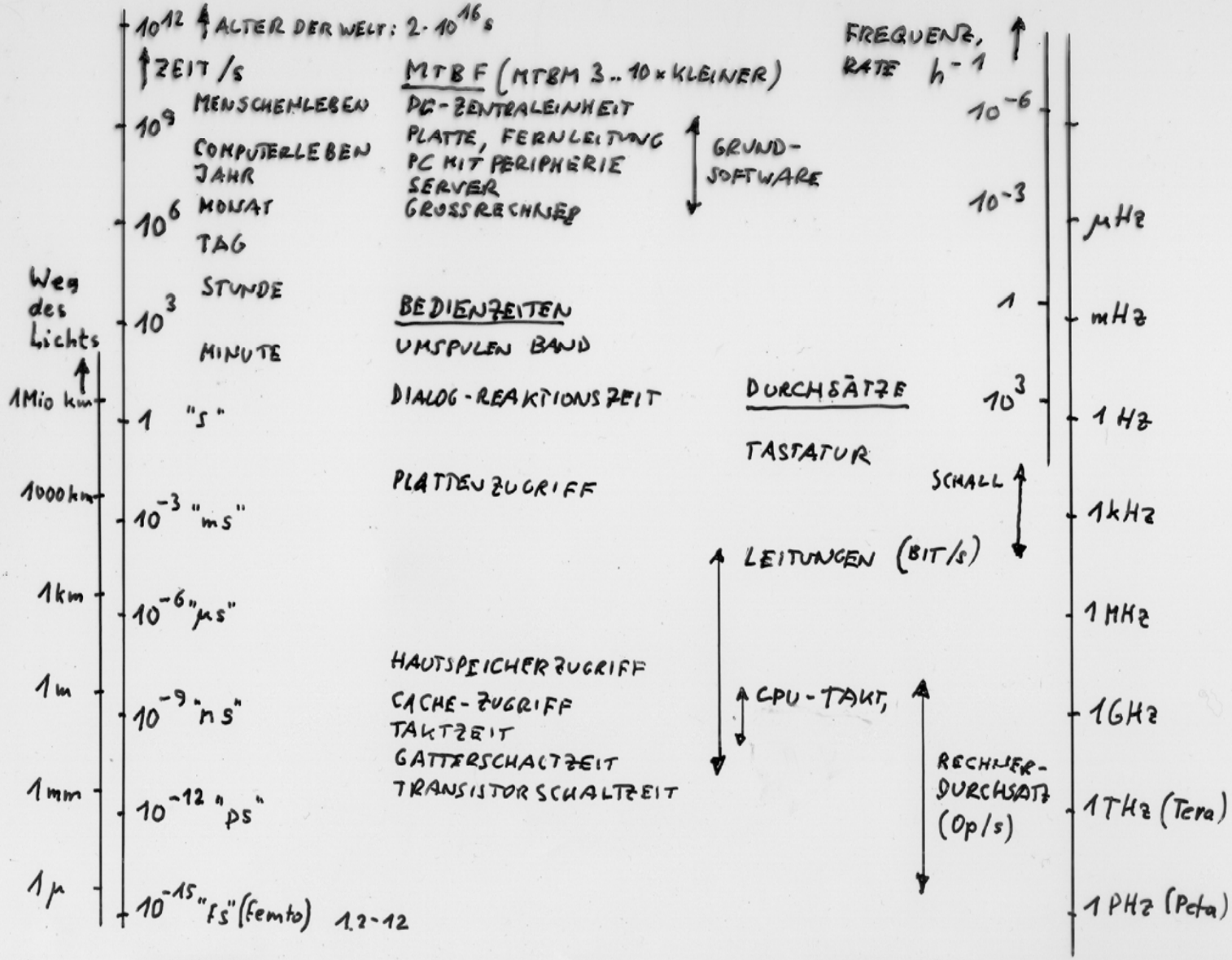
- zuverlässig (reliable), wenn sie sich im Betrieb spezifikationsgemäß verhält. Sonst hat sie (mindestens) einen Fehler (fault). Fehler führen zu fehlerhaften Auftragsausführungen: Störungen (malfunctions). Der mittlere zeitliche Abstand von Störungen heißt MTBM (meantime between malfunctions). Verhindern die Fehler den Betrieb oder sind sonst nicht akzeptabel, liegt ein Ausfall (failure) vor. MTBF: meantime between failures. Verfügbarkeit (availability) ist der relative Anteil der planmäßigen Betriebszeit, in dem die FE Aufträge spezifikationsgemäß erledigt.
- fehlertolerant, wenn sie trotz Anwesenheit von Fehlern spezifikationsgemäß arbeitet.

Eine FE ist

- sicher (secure), wenn sie nicht missbraucht werden kann (z.B. unerlaubte Einsichtnahme, Manipulation, Sabotage)
- sicher (unschädlich, safe), wenn sie im Betrieb, auch unter Störungen oder Missbrauch, nicht Schaden für ihre Umgebung anrichten kann.

In Prozessoren gibt es privilegierte Funktionen (z.B. Speicher-schutz (memory protection)), die

- Zuverlässigkeit verbessern (verhindern Fehlerausbreitung)
- Sicherheit gegen Missbrauch verbessern (verhindern Übergriffe der Prozesse aufeinander)
- Unschädlichkeit stützen (scheiternder Prozess kann nicht andere schädigen)


10^{12} ALTER DER WELT: $2 \cdot 10^{16}$ s
ZEIT / s
MENSCHENLEBEN
10^9
COMPUTERLEBEN
JAHR
10^6 MONAT
TAG
STUNDE
10^3
MINUTE
1 "s"
10^{-3} "ms"
10^{-6} "µs"
10^{-9} "ns"
10^{-12} "ps"
10^{-15} "fs" (femto)
Weg des Lichts
1Mio km
1000km
1km
1m
1mm
1µ
MTBF (MTBM 3..10 x KLEINER)
PC-ZENTRALEINHEIT
PLATTE, FERNLEITUNG
PC MIT PERIPHERIE
SERVER
GROSSRECHNER
GRUND-SOFTWARE
BEDIENZEITEN
UMSPULEN BAND
DIALOG-REAKTIONSZEIT
PLATTENZUGRIFF
HAUPTSPEICHERZUGRIFF
CACHE-ZUGRIFF
TAKTZEIT
GATTERSCHALTZEIT
TRANSISTORSCHALTZEIT
FREQUENZ, RATE h^{-1}
10^{-6}
10^{-3}
1
10^3
µHz
mHz
1 Hz
1kHz
1MHz
1GHz
1THz (Tera)
1PHz (Peta)
DURCHSÄTZE
TASTATUR
SCHALL
LEITUNGEN (BIT/s)
CPU-TAKT,
RECHNER-DURCHSATZ (Op/s)
1.2-12

1.3 von Neumann Revisited

Wirklich mal lesen: Die Überlegungen der großen Erfinder:

Zuse, K.: Der Computer – mein Lebenswerk, Springer 1993

http://www.mediaculture-online.de

Burks, Goldstine, von Neumann: Preliminary Discussion of the Logical Design of an Electronic Computing Instrument, 1946; abgedruckt z.B. in Bell, G. et al.: Computer Structures, Readings and Examples, McGraw Hill, 1971

http://research.microsoft.com/~gbell/computer_structures_Readings_and_Examples/00000112.htm

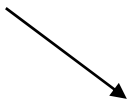
Von Neumann kannte Zuses Arbeiten 1946 nicht.

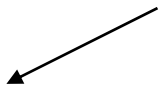
Ideengeschichte der programmgesteuerten Rechenmaschine: Mechanisierung des Rechnens

1623 Schickard: Addier/Subtrahiermaschine
1645 Pascal: Addier/Subtrahiermaschine
1674 Leibniz: Vierspeziesmaschine

(industrielle Fertigung)

18. Jahrhundert: Musik/Spielautomate
1805 Jacquard: Lochkartengesteuerte Webstuhl





1823 Babbage: Konzept und Laboraufbau eines lochkartengesteuerten mechanischen Rechners
1941 Zuse „Z3“: Funktionsfähiger lochstreifengesteuerter elektromechanischer Rechner
1945 von Neumann: Konzept eines Rechners mit gespeichertem Programm, der Verhalten von berechneten Werten abhängig machen und Programm ändern kann
2007 etwa 10^{11} Rechner im Gebrauch; schnellster Rechner etwa 10^{15} mal schneller als Zuses Rechner

Von Neumann Konzept

1. Rechner besteht aus Speicher (storage, memory), Leitwerk (control unit), Rechenwerk (arithmetic unit, data path), Ein-/ Ausgabegeräten (i/o units)

- schon bei Zuse! Heute: Speicherhierarchie, Leitwerk und Rechenwerk bilden Zentralprozessor (central processing unit, cpu): erweitert um Cache Memory wichtiger Baustein (Multiprozessorsysteme, Chipfabrikation); mehrere Rechenwerke in „superskalaren“ Prozessoren und in Feldrechnern (array processors); interne Verbindungsstruktur (z.B. Bus-System) ist kritischer Bestandteil des Rechners

2. Die Struktur ist unabhängig vom bearbeiteten Problem

- Schon bei Zuse, nicht aber bei Analogrechnern und bei anderen früheren Rechnern (z.B. Eckert/Mauchly ENIAC (1945)

3. Programm und Werte stehen in demselben Speicher und können durch den Rechner verändert werden.

Heute:

- Bei „Mikrokontrollern“ („eingebettete“ Rechner für invariante Aufgaben) aus Zuverlässigkeits-/ Sicherheits- (reliability/security) Gründen oft getrennter, nicht beschreibbarer Programm/Konstanten-Speicher und beschreibbarer Speicher für veränderliche Werte.
- Veränderung des Programms während der Ausführung: nein! Sonst Probleme:
  -- Zuverlässigkeit, Sicherheit
  -- „Eintrittsinvarianz“: bei Rekursion und Mehrfachnutzung eines Programms durch mehrere Prozesse darf Programm nicht verändert werden!
  -- Bei Vorgriff der CPU auf „nachfolgende“ Befehle
  -- Bei Programmverdrängung (geändertes Programm müsste auf Platte o.ä. rückkopiert werden).
- Die von von Neumann gewünschte Flexibilität wird durch bedingte Sprünge und Adressmodifikationen erreicht, ohne Änderung des Programms

4. Der Speicher ist in Zellen gleicher Größe geteilt, die durch fortlaufende Nummern (Adressen) bezeichnet werden

- schon bei Zuse
- im Laufe der Rechnergeschichte vielfältige Variationen, ohne anhaltende Bedeutung:
  - -- Datenzugriff nur indirekt über (maschinengestützte) Deskriptoren (-> objektorientierte Rechner)
  - -- Datentyp als Bestandteil des Zelleninhaltes, verwendbar
    - -- zur automatischen Prüfung (Befehlszähler darf nicht auf Gleitpunktzahl zugreifen) oder
    - -- zur automatischen Anpassung (Addition passt sich an Zahlentyp an).

„Tagged architectures“

Und noch grundsätzlicher:

Von Neumanns wiederbeschreibbare Zellen sind das Urbild der Programmvariablen (considered harmful). Es sind Berechnungs-schemen entwickelt worden, die mit einmaliger Wertzuweisung an eine Variable auskommen (single assignment-variable, „Formulare“ bei [Broy M 98, S. 156]):

- Datenflussmaschinen (bedingte Ausdrücke und Iteration)
- Reduktionsmaschinen (bedingte Ausdrücke und Rekursion)

Beide kommen ohne serielles Programm aus; beide haben technisch keine Bedeutung erlangt.

Ein wichtiges Prinzip der Datenflussmaschine („data driven evaluation“, berechnet wird, sobald die Eingangswerte verfügbar sind) wird zur internen Steuerung in vielen Prozessoren eingesetzt.

5. Das Programm besteht aus einer Folge von Befehlen (instructions), die sequentiell zu verstehen sind und in der Aufzeichnungsreihenfolge auszuführen sind. Die Befehle enthalten i.a. nicht Werte, mit denen zu rechnen ist, sondern die Adressen der sie beherbergenden Zellen.

- schon bei Zuse
- die Entwicklung der Struktur „schneller“ Rechner (d.h. hoher Grenzdurchsatz (Befehle/s)) ist ein einziger listenreicher Kampf gegen von Neumanns Serialität. Aus Leistungsgründen sind viele Befehle desselben Programms kollateral/parallel in Bearbeitung (Little‘s Formel!), trotzdem muss das Ergebnis dem bei serieller Verarbeitung entsprechen:

  Pipelines, Superskalarität, EPIC (explicitly parallel instruction computer), spekulative Auswertung; Parallelrechner.

Der Umstand, dass die Operanden und Ergebnisse durch Adressen repräsentiert werden, ist direkte Folge der freien Parametrisierbarkeit, d.h. der Flexibilität der Programme, aber auch Ursache umständlicher Abläufe, z.B. A:=B+C:

Befehlsadresse an Hauptspeicher
Befehl vom Hauptspeicher
Adresse von B an Hauptspeicher
B vom Hauptspeicher
Adresse von C an Hauptspeicher
C vom Hauptspeicher
Addition B+C
Adresse von A an Hauptspeicher
A an Hauptspeicher

„von Neumann Flaschenhals“ (bottleneck) Prozessor/ Hauptspeicher

In der Tat wird der Grenzdurchsatz der Zentraleinheit wesentlich durch die Zugriffe auf den Hauptspeicher begrenzt, mit ungünstigem Trend: Vom Intel 80286 (1982) bis zum Intel Pentium 4 (2005) ist die Taktfrequenz um einen Faktor 120, der Grenzdurchsatz (Op/s) um einen Faktor 2250 gewachsen, die Hauptspeicherzugriffszeit aber nur um den Faktor 4,3 kleiner geworden. Der Leistungszuwachs wurde durch „Cache Memories“, Pipelining usw. ermöglicht.

6. Von der Befehlsfolge kann durch Sprungbefehle (jump/branch instructions) abgewichen werden. Die Ausführung eines Sprungs kann von gespeicherten, d.h. auch: errechneten, Werten abhängig gemacht werden.

   - Neu! Das war nicht so bei Zuse! Entscheidende Bedeutung!
   - Ähnlich wirken vom Rechner vorgenommene Adressmodifikationen.
   - Sprünge (insbesondere bedingte) sind ein wesentliches Handicap bei kollateraler Programmabarbeitung in der CPU. Bedingte Befehle können günstiger sein.

7. Die Maschine benutzt Binärcodes; Zahlen werden dual dargestellt.

- Schon bei Zuse (der sogar Gleitpunktzahlen in der Z3 verarbeitet).

Jahrzehntelang wurde die Parole „Befreiung von von Neumann“ gläubig aufgenommen.

Wichtigste Erweiterungen des von Neumann-Konzeptes:

- Adressmodifikation: Die im Befehl enthaltene(n) „Programm“adresse(n) werden i.a. in 2 Stufen modifiziert:
  -- durch den Programmablauf in eine Prozessadresse (effektive Adresse)
  -- durch Betriebssystem und Maschinenfunktionen in eine Maschinenadresse (physical address)

  Erlaubt Modifikation des Programmverhaltens ohne Modifikation des Programms!
- Kollaterale/parallele Verarbeitung innerhalb des ganzen Rechners (z.B. E/A), insbesondere in der CPU.
- Speicherhierarchie: Scheinbar schnellster Speicher mit der Kapazität des größten vorhanden.

(Erweiterung des von Neumann-Konzeptes)

- Steuerung des Prozessors in einer Hierarchie
  -- von Neumann-Schicht (instruction set architecture ISA)
  -- Mikromaschine
  (--evtl. weitere Schicht Nanomaschine)

- Unterbrechung der Programmausführung und Umsteuerung durch interne und externe Signale

- Systeme aus Prozessoren bzw. Rechnern: Multiprozessoren, Multirechner

Und von größter Bedeutung: Erweiterung der Funktionalität durch Software:

- Höhere (benutzergerechte) Schnittstellen
- Langfristige Datenhaltung
- Fehlertoleranz, Sicherheit
- Effektive Ablaufsteuerung für Rechenaufträge

1.4 Schichtung, Interpretation und Übersetzung

Aufträge an Rechensysteme durchlaufen mehrere Sprachebenen, die verschiedenen Anforderungen an die Benutzerkanäle, Abstraktion, Universalität und Bezug auf die elementare (binäre) Technik erfüllen müssen.

Der Übergang von einer Sprachebene zur nächst „tieferen“ kann durch Übersetzung oder Interpretation geschehen.

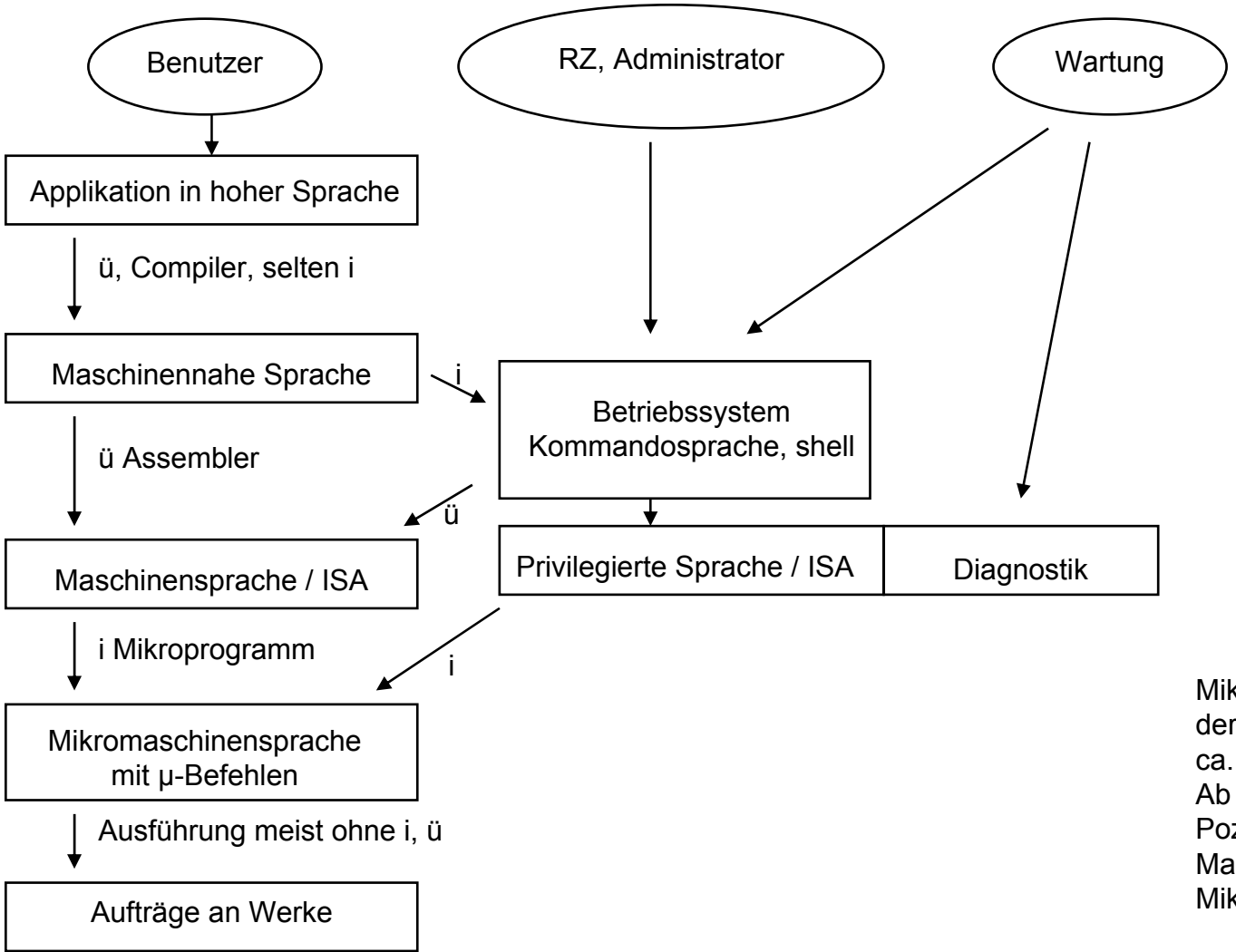
Übersetzung (translation, compilation, assembling) erzeugt aus einer Auftragsbeschreibung in der „höheren“ Sprache A eine semantisch äquivalente in Sprache B.

Interpretation (interpretation, aber nicht in der Bedeutung „Dolmetschen“) ist in der Informatik die schrittweise Ableitung und Durchführung einer Auftragsfolge aus einer komplexen Anweisung (Programm), bei der in jedem Ableitungsschnitt der durch die Ausführung erreichte Zustand herangezogen werden kann.

Ein interpretierendes Werk heißt Prozessor (z.B. von Neumanns Leitwerk (leitet Aufträge aus Programm ab) und Rechenwerk (führt aus). Ein interpretierendes Programm heißt Interpreter.

Beide Techniken sind hierarchisch einsetzbar!

Rechensystem (computing system): Sprachschichten (levels) und Auftrags-übergänge (transitions): Übersetzung ü und Interpretation i



Rückmeldungen laufen von unten nach oben!

ISA Instruction Set Architecture

Mikroprogrammierung dien
der Mikrointerpretation. Vor
ca. 1960 bis 1978 Standard
Ab 1978 in RISC-
Pozessoren Verschmelzun
Maschinensprache/
Mikromaschinensprache

Interpretation	Übersetzung
Liefert nur die Folge relevanter Aufträge	Liefert vollständiges Programm
Erlaubt Selbstreferenzierung und Veränderung des zu interpretierenden Programms	Erlaubt keine Selbstreferenzierung oder Veränderung des zu übersetzenden Programms
Ist daher mächtiger	Ist daher schwächer
In Sonderfällen effizienter (z.B. kurzer Weg durch das Programm, einmalig)	Meist effizienter (z.B. bei Mehrfachausführung)
	Erlaubt Optimierung in größerem Kontext
Interpretation benutzt kleine Übergabeeinheiten , dann sogleich Ausführung/Zustandsänderung	Wenn man die Ausführung des Programms einbezieht, dann hat Übersetzung eine große Übergabeeinheit: Gesamtprogramm

1.5 Aktuelle Ausschnitte

Für den Interpretationsprozess muss das Programm und der aktuelle Zustand (z.B. Variablenwerte) in Speichern bereitgestellt werden.

Speicher mit einer kleinen Zugriffszeit sind teurer je Bit, Speicher mit einer großen Zugriffszeit sind billiger je Bit.

Speicher großer Kapazitäten haben stets große Zugriffszeit, können aber billiger je Bit sein.

Interpretationen brauchen in kleinen Zeitintervallen nur kleine Teilmengen des Speicherinhalts

- diese ändern sich nur langsam (zeitliche Lokalität, temporal locality
- diese sind weitgehend kompakt im Adressraum (räumliche Lokalität, spatial locality)

Gründe: Befehlszähler, Datenstrukturen, Bindung der Variablen an Moduln.

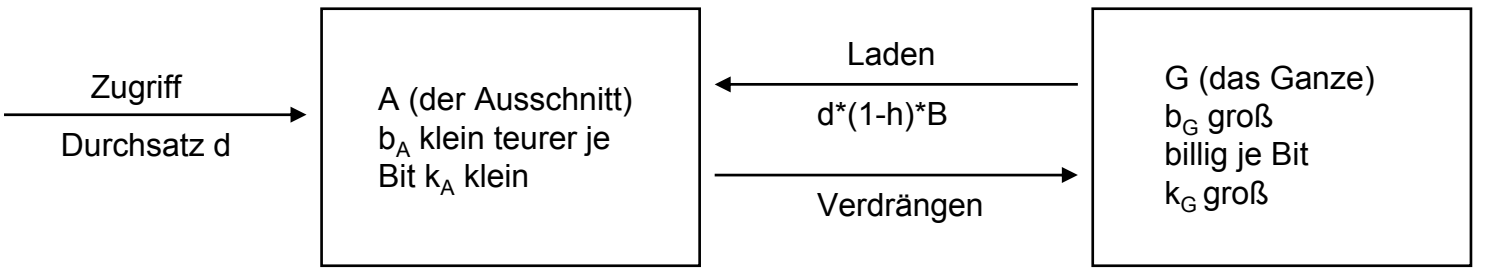
Idee des aktuellen Ausschnitts (current environment):

Erstmals F.R. Güntsch 1957, erste Maschine Ferranti „Atlas“ 1963.
Virtualisierung des Speichers: Ganze Umgebung des Interpretationsprozesses liegt im großen, billigen Speicher.
Automatisch wird ein aktueller Ausschnitt in einem kleinen Speicher geringer Zugriffszeit organisiert:
„So billig wie der große, so schnell wie der kleine Speicher“.

Und hierarchisch eingesetzt:

Register	Cache (1-2-3)	Hauptspeicher	Platten	Bandarchiv
1 kB	1 MB	1 GB	100 GB	1 TB- 1PB
1 ns	3 ns	30 ns	10 ms	min

Grundstruktur des Ausschnitts:



Zum Beispiel	Cache	Hauptspeicher
Bedien„Zugriffs“zeit	b_A	b_G
Kapazität	k_A	k_G

Wichtigstes Gütekriterium: Trefferhäufigkeit h (gesuchtes Datum ist im Ausschnitt) (hit ratio). Wächst mit Lokalität des Zugriffs-prozesses und Kapazität des Ausschnitts und Qualität der Lade/Verdrängungsstrategie. Hohes h senkt mittlere Zugriffszeit $b = b_A + (1-h)\, b_G$ und Ladedurchsatz: d*(1-h)*B (B Blockgröße).

Wer führt die Lade/Verdrängungsstrategie aus?

Register/Hauptspeicher:
Programmierer (Assembler), Compiler (höhere Sprache)

Cache/Hauptspeicher:
Prozessor (Hardware)

Hauptspeicher/peripherer Speicher:
Betriebssystem (Seitenaustausch), Programmierer

Peripherer Speicher/Archivspeicher:
Rechenzentrum/Administrator

Strategischer Freiraum liegt im Austauschverfahren:

Ladestrategie: on demand, selten vorhersehend. Wie kann man trotz prinzipiell unbekanntem künftigen Zugriffsverlauf mit hoher Trefferhäufigkeit h das Gesuchte A bereitstellen? Z.B. durch zusätzliches Laden der adressräumlichen Umgebung (Block) zu jedem benötigtem Datum: Nutzung der räumlichen Lokalität!

Verdrängungsstrategie: z.B. least recently used (das am längsten Unbenötigte). Die Vorhersage der unbekannten Zukunft wird durch Rückblick in die Vergangenheit ersetzt: Nutzung der zeitlichen Lokalität!

(Nicht veränderte Inhalte werden nicht zurücktransportiert, sondern überschrieben).

Durch Ausschnitt und Lade/Verdrängungsverfahren wird eine Einheit mit gleicher Funktionalität, aber verbesserten Eigenschaften geschaffen: Virtualisierung.

Überaus wichtiges Prinzip der technischen Informatik (virtuelle Prozessoren, Maschinen, Kanäle, Peripheriegeräte, ..). Abbildung einer Einheit auf die nächsthöhere Schicht der Hierarchie!

Und noch:

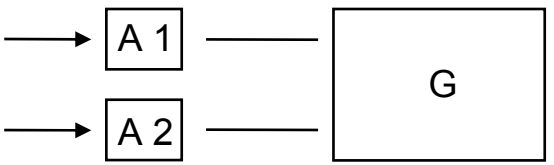
Wie findet der Zugreifer das Datum im Ausschnitt?

- Adresse neben Datum vermerken, Ausschnitt durchsuchen (parallel!)

- Adresse bestimmt Ort im Ausschnitt (kann aber wegen $k_A < k_G$ nicht umkehrbar eindeutig sein! Erleichtert aber Suche beträchtlich).

Siehe die kommende Darstellung zu Caches.

Und dann noch:



Sobald es mehr als einen Ausschnitt gibt und wenigstens ein Zugreifer schreibt in seinem Ausschnitt, geht die Kohärenz (alle sehen immer das Gleiche oder gar nichts) verloren. Lösungen: immer auch in G schreiben, in fremdem A löschen

Aber in einen verteilten System gibt es i.a. kein G, sondern die Teilinhalte (Block, Seite, …) können eigene Heimatorte haben und diese auch verlieren; i.a. werden von ihnen viele Instanzierungen bestehen.

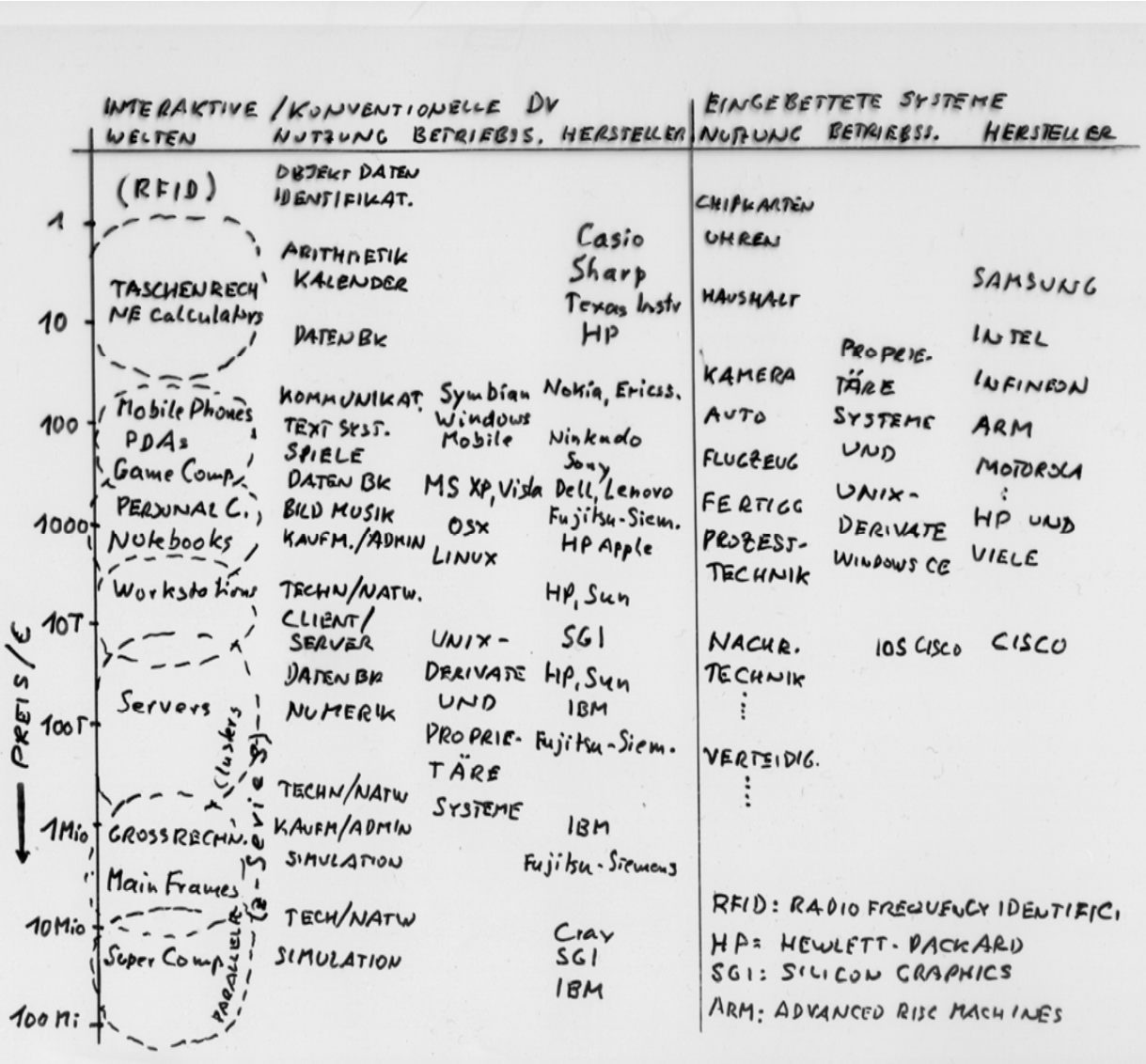
„n readers/1writer“: Jeder Teilinhalt ist jederzeit für alle lesbar, aber nicht schreibbar, oder er hat einen einzigen Schreibeigentümer. Dann ist er für diesen beschreibbar und auch nur für diesen lesbar. Will ein anderer Prozess den Teilinhalt lesen oder schreiben, dann muss der Schreibeigentümer enteignet werden. Damit steht der Teilinhalt allen zur Herstellung einer eigenen Instanzierung zur Verfügung. Will der neue Eigentümer aber schreiben, so müssen alle anderen Instanzierungen des Teilinhaltes zuvor ungültig gemacht werden.

1.6 Rechnertypen und Markt

Der EDV-Markt beträgt 2007 etwa 1000 Mrd. € (das Bruttosozial-produkt Deutschland beträgt 2000 Mrd. €), wächst etwa 10% jährlich, bei Verbesserung des Leistungs-/Preisverhältnisses von etwa 40%/Jahr.

Ein rückläufiger Anteil betrifft Rechner (30%?).

Wir verschaffen uns einen Gesamtüberblick und sehen dann ein paar uns nahestehende Beispiele an, sozusagen äußerlich.


INTERAKTIVE / KONVENTIONELLE DV
WELTEN
NUTZUNG BETRIEBSS. HERSTELLER
EINGEBETTETE SYSTEME
NUTZUNG BETRIEBSS. HERSTELLER
(RFID)
OBJEKT DATEN IDENTIFIKAT.
CHIPKARTEN
1
UHREN
ARITHMETIK
KALENDER
Casio
Sharp
Texas Instr
HP
TASCHENRECHNE Calculators
HAUSHALT
SAMSUNG
10
DATENBK
INTEL
PROPRIE-
KAMERA
TÄRE
INFINEON
KOMMUNIKAT.
Symbian
Nokia, Ericss.
Mobile Phones
TEXT SYST.
Windows Mobile
AUTO
SYSTEME
ARM
100
PDAs
SPIELE
Nintendo
Sony
FLUGZEUG
UND
MOTOROLA
Game Comp.
DATEN BK
MS XP, Vista
Dell, Lenovo
FERTIGG
UNIX-
PERSONAL C.
BILD MUSIK
Fujitsu-Siem.
1000
HP UND
Notebooks
KAUFM./ADMIN
OSX
LINUX
HP Apple
PROZESS-
DERIVATE
Workstations
TECHN/NATW.
HP, Sun
TECHNIK
WINDOWS CE
VIELE
10T
CLIENT/
SERVER
SGI
NACHR.
IOS CISCO
CISCO
UNIX-
DATENBK
DERIVATE
HP, Sun
TECHNIK
Servers
NUMERIK
UND
IBM
100T
PROPRIE-
Fujitsu-Siem.
TÄRE
VERTEIDIG.
TECHN/NATW
SYSTEME
1Mio
GROSSRECHN.
KAUFM/ADMIN
IBM
SIMULATION
Fujitsu-Siemens
Main Frames
10Mio
TECH/NATW
Cray
SIMULATION
SGI
Super Comp.
IBM
100Mio
PREIS/€
(Clusters
Service Sj
PARALLEL &
RFID: RADIO FREQUENCY IDENTIFIC.
HP: HEWLETT-PACKARD
SGI: SILICON GRAPHICS
ARM: ADVANCED RISC MACHINES

Nahestehende Beispiele zur Anschauung:

- Rechnerbetrieb an der Fakultät für Informatik, TUM und Studentenrechner

- Bundeshöchstleistungsrechner im Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften

Rechnerbetrieb an der Fakultät für Informatik, TUM

Etwa 350 Bedienstete, davon 33 Professoren

Etwa 1900 Studierende

Rechnerbetriebsgruppe mit 20 Mitarbeitern

- Vernetzung (1 Gb/s Ethernet)
- Beratung und Beschaffung: Hardware und Standardsoftware
- Betrieb Rechnerhalle (100 Plätze), Bibliothek, …
- Wartung

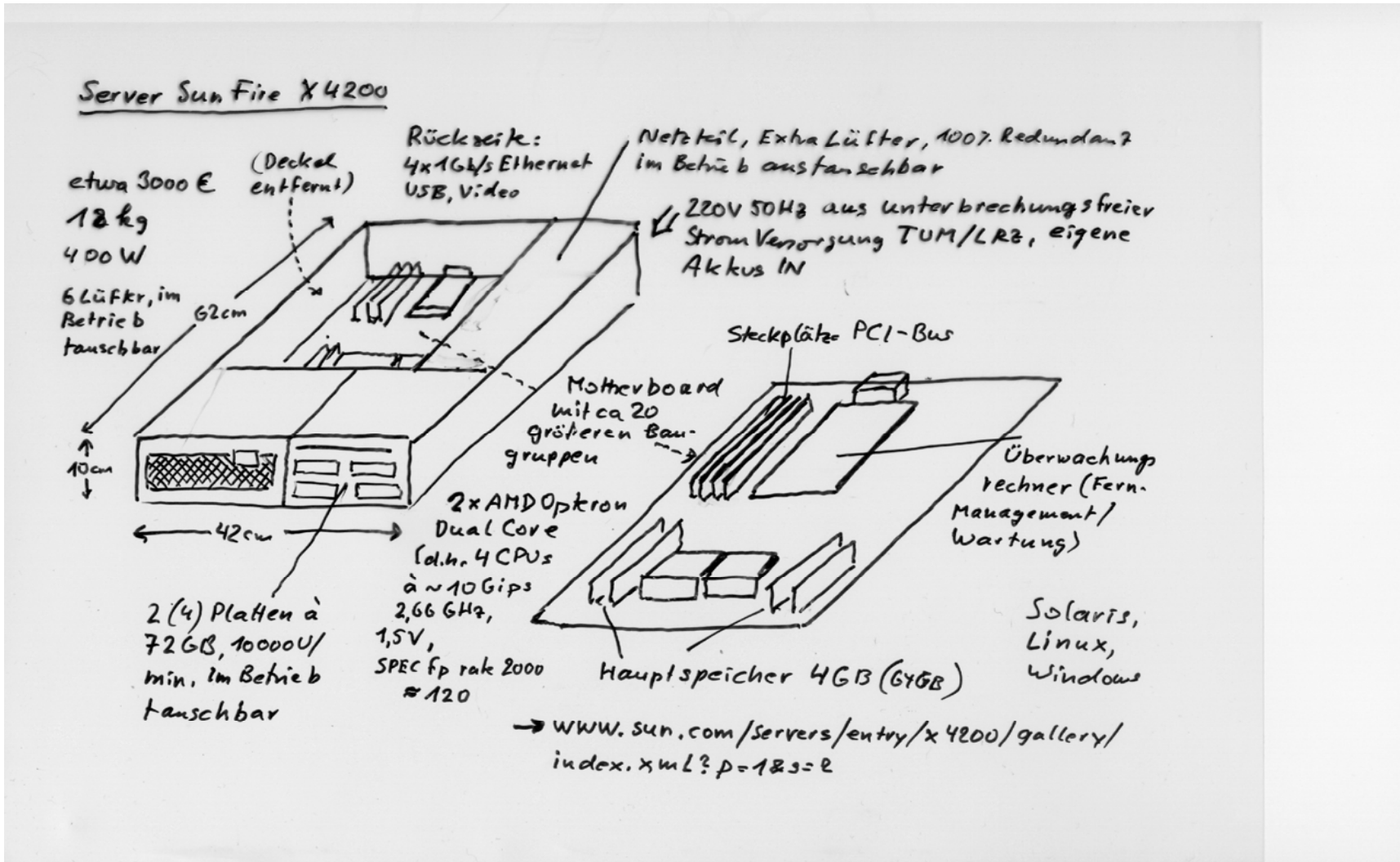
Rechner werden mit erheblichem Hochschulrabatt gekauft, mit fünfjähriger Garantie (Ersatzteile).

Etwa 900 Rechner an den Lehrstühlen (Bedienstete, Praktika).

Unbekannte Zahl externer Notebooks, mit eingeschränktem Service.

Beispiel: Rechner in der Rechnerhalle, Bibliothek, …

4	Server Sun Sunfire X 4200
100	„Thin“ Clients Sun Ray 16


Server Sun Fire X4200
etwa 3000 €
18 kg
400 W
6 Lüfter, im Betrieb tauschbar
(Deckel entfernt)
62cm
10cm
42cm
Rückseite:
4x1Gb/s Ethernet
USB, Video
Netzteil, Extra Lüfter, 100% Redundanz im Betrieb austauschbar
220V 50Hz aus unterbrechungsfreier Stromversorgung TUM/LRZ, eigene Akkus IN
Steckplätze PCI-Bus
Motherboard mit ca 20 größeren Baugruppen
Überwachungsrechner (Fern-Management/Wartung)
2x AMD Opteron Dual Core (d.h. 4 CPUs à ~10 Gips 2,66 GHz, 1,5V, SPEC fp rak 2000 ≈ 120
2 (4) Platten à 72 GB, 10000U/min, im Betrieb tauschbar
Hauptspeicher 4GB (64GB)
Solaris, Linux, Windows
→ www.sun.com/servers/entry/x4200/gallery/index.xml?p=1&s=2

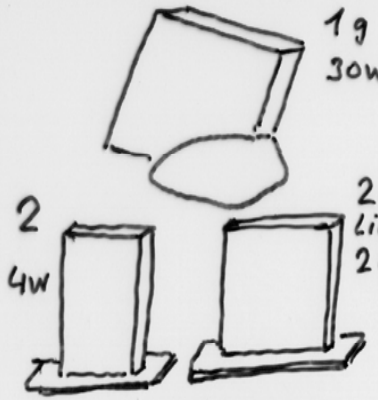
Sun Ray Thin Clients 1g, 2, 2FS

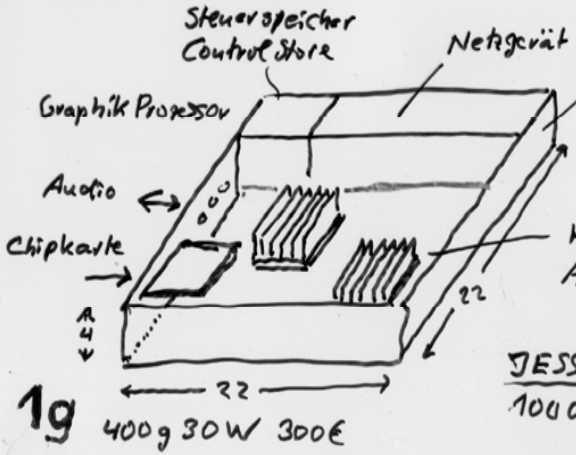
Graphik- und Kommunikationsbaustein für Rechnerarbeitsplatz, dazu Monitor, Tastatur, Maus

"Stateless": Anwendungsprozesse hinterlassen keine Spuren; rudimentäres Betriebssystem; keine Platte: ProzessStates im Server. Prozess wandert mit Chipkarte.

Keine Wartung / Reparatur / kein Ein/Ausschalten

Keine Belüftung





In der Seitenwand: Anschlüsse für Ethernet, Monitor (DVI Digital Video Interface), USB (Universal Serial Bus)

Haupt-Prozessor: Sun Sparc bzw. Alchemy 1550 (32b, 500 MHz, 0,5 W) Mips Core

JESSENS GESETZ: 1 kg HARDWARE KOSTET IMMER 1000 DM, ABER SEIT 2001 500 €.

Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften

Zentrale Rechenkapazität, Ausbildung, Kommunikationsnetz (Münchener Hochschulnetz), Datenhaltung, Sicherheit, … für die Münchener Hochschulen / für Bayern / für Deutschland.

Betreibt Bundes-Höchstleistungsrechner HLRB II.

SGI (Silicon Graphics) Altix 4700

Zehntschnellster Rechner der Erde nach TOP 500 Sommer 2007

c = 62,3 TFlops, aus 9728 CPUs (Intel Itanium)

39 TB Hauptspeicher

600 TB Plattenspeicher

100 t

1 MW

Altix 4700

Hochleistungsrechner sind heute immer Parallelrechner: Verarbeitung von einem oder mehreren intern nebenläufigen Benutzeraufträgen, dazu funktionelle „Zerlegung“ der Maschine.

SGI Altix 4700: Zweistufige Hierarchie: Partitionen, deren Prozessoren intern auf Hauptspeicher direkt und gleich zugreifen (Shared Memory Processor). Die Gesamtmaschine im LRZ setzt sich aus 19 Partitionen zusammen; es gibt eindeutige Adressen, aber Zugriffe auf andere Partitionen verlaufen langsamer (NUMA: Non-Uniform Memory Access). Innerhalb jeder Partition Kohärenz der Cache-Inhalte! (cc-NUMA cache coherent NUMA). LINUX-Betriebssystem je Partition, darüber Altair PPS „Batch-System“ für Betriebsführung.

Konstruktiv ist jede Partition aus 128 oder 256 „Blades“ aufgebaut, 40 je Schrank.

- 8 Bladetypen (Rechner, Speicher, E/A, Sondertypen)
- ein Computing Blade enthält 2 oder 4 Intel Itanium Montecito CPUs („Cores“ in Intel-Terminologie), und 8 oder 16 GB Hauptspeicher
- Speicherzugriff in der Partition über NUMAlink 4-Netz (2x 6,4 GB/s je Blade), nicht blockierende „Crossbar“-Schalter.

Verbindung der Partitionen untereinander in derselben Technik, aber indirekt.

Intel Itanium ist 64 b EPIC/VLIW Prozessor: mit jedem Befehlswort erhält der Prozessor drei vom Compiler als nebenläufig bestimmte Befehle zur parallelen Ausführung: Explicit Parallel Instruction Computer / Very Long Instruction Word.

Itanium Montecito: 220 Mio. Transistoren, 100 W, etwa 1000 bis 3000 €; 1,6 GHz, c= 9,6 Gips

1.7 Kommunikationsstrukturen

1.7.1 Bedeutung und Begriffe

Kommunikationsstrukturen: Strukturen und Verfahren zur Verbindung von Nachrichtenquellen/senken (message sources/sinks).

Wachsende Bedeutung von Kommunikation in Rechensystemen:

- Asynchron arbeitende, lose gekoppelte Einheiten wichtig wegen
  -- Modularisierungsvorteilen, lokaler Takt, Fehlertoleranz
  -- Rechnernetzen: Nutzer, Daten, Prozessoren, Geräte notwendig verteilt
- Großintegration erzwingt gestufte Kommunikationsschichten:
  -- auf den Chip (Prozessorbus, Mehrprozessorsysteme (multicore))
  -- Pin-Limit (pin limitation)
  -- auf Processor Board
  -- im Rechner (vgl. Altix)

Begriffe:

Schnittstelle (interface): Menge der Festlegungen, die ordnungsgemäßes Verhalten eines Kommunikationspartners gegenüber einem anderen beschreiben, besteht aus:

- physischer Schnittstelle (physical interface): Leitungen, Stecker, Potentiale, Feldstärke, Frequenzband, Zeiten, Widerstände, Spannungen, …

- Kommunikationsprotokoll (communication protocol): Syntax, Semantik und zeitliche Ordnung des Nachrichtenaustausches zwischen Partnern

  (und Nachrichten (messages): Das sind Daten, die gerade transportiert werden)

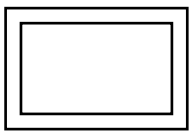
1.7.2 Elemente

Nachrichtenstation (Quelle, Senke),
z.B. Rechner, Speicher

- sendet/empfängt Nutzdaten

- baut Nachrichtentransport auf / ab

Nach Einsatz bei Auf/Abbau:



aktiv (master), z.B. Steuerwerk

passiv (slave), z.B. Speicher

Leitungen

Unterscheidbar nach Einsatz (Steuer/Datentransport), nach Richtung und Gleichzeitigkeit des Nachrichtentransports,

- (voll)duplex (bidirectional): beide Richtungen gleichzeitig nutzbar, z.B. klassische Telephonie (POTS, plain old telephon service)
- halbduplex: beide Richtungen im zeitlichen Wechsel nutzbar, z.B. Bus
- simplex: nur eine Richtung

Nach Bit-Füllung im Leitungsquerschnitt:

- seriell
- parallel

Und in die Niederungen der Übertragungstechnik:

Auf einer langen „seriellen Leitung“ können viele Bits gleichzeitig unterwegs sein, ein Bit von

- 1µs Dauer misst ca. 200m
- 1ms Dauer misst ca. 200 km

„Leitungen“ können sein:

- Drahtleitungen (verdrillt) bis ca. 1 Gb/s
- Streifenleitungen bis ca. 1 Gb/s
- Koaxialkabel bis ca. 1 Gb/s
- Glasfaserkabel bis ca. n*40 Gb/s (n<30)
- Funkverbindungen bis ca. 100 Mb/s (simplex: Richtfunk)

Werte ohne Zwischenverstärkung, entfernungsabhängig

Netze aus Nachrichtenstationen und Leitungen enthalten für die Nutzdaten

- Zugeordnete (dedicated) Leitungen: diese dienen genau 2 Nachrichtenstationen
- Gemeinsame (shared) Leitungen: Aufteilung unter den Kommunikationspartnern
  - Frequenzteilung (-multiplex): nicht in Rechnern, aber auf Lichtleitern (sozusagen: Kanäle verschiedener „Farbe“)
  - Zeitteilung (-multiplex)
    - starre Scheiben (slots), fest zugeteilt, z.B. Synchronbus
    - starre Scheiben, nach Anforderung, z.B. Synchronbus
    - variable Scheiben, z.B. Asynchronbus

Frequenzteilung und fest zugeteilte starre Scheiben: Einfach, gute Auslastung möglich, aber keine Anpassung an wechselnde Last.

• Steuerung (control)

baut Leitungszugang (line/medium access) bzw. Verbindung (connection) auf und ab. Realisierbar:

- zentral: oft billig, gute Auslastung der Leitungen möglich, eindeutige Steuerung, Ausfallanfälligkeit („single point of failure“)
- dezentral: leicht erweiterbar, Potential für Fehlertoleranz (aber Diagnose und Recovery meist schwierig)

1.7.3 Vermittlung (switching)

O Nachrichtenvermittler (switch, router)

Wählt den Weg für die Verbindung bzw. für ankommende Nachrichten, kann oft auch Nachrichten speichern und umsetzen. Damit indirekte Übertragung (indirect transmission).

Zwei wichtige Fälle:

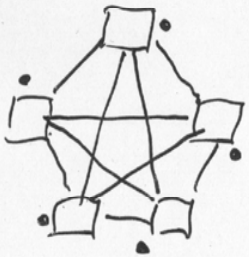
- Leitungsschaltung/-vermittlung
- Teilstreckenschaltung/-vermittlung

Beide Bezeichner schlecht.

1.7.4 Beispiele: Vermittlungsfreie (Direkte) Netze

VERTEILTE STEUERUNG, ZUGEORDNETE WEGE:

VOLLER VERBUND:



SCHLECHT ERWEITERBAR

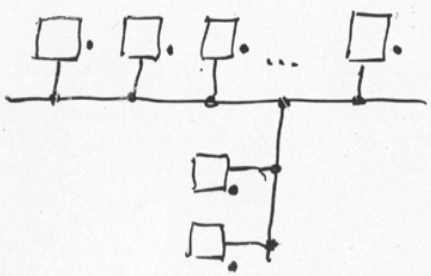
$\binom{n}{2}$ WEGE NOTWENDIG

DURCHSATZSTARK

KEINE BÜNDELUNGSVORTEILE

IN DIESER FORM NICHT AUSFALLSICHER

VERTEILTE STEUERUNG, GEMEINSAMER WEG:



CSMA/CD - ETHERNET

GUT ERWEITERBAR ("skalierbar")

DURCHSATZ SCHWACH

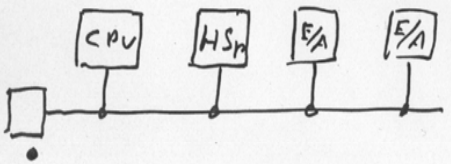
DEZENTRALER ZUGANG

UNGEEIGNET BEI STARKER LAST

NICHT AUSFALLSICHER

NICHT SICHER (secure)

DIREKT, ZENTRALE STEUERUNG, GEMEINSAMER WEG

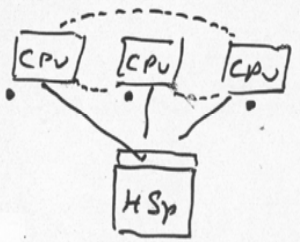


ZENTRALER BUS IN KLEINEN RECHNERN

(DEC PDP 11 1970)

- DURCHSATZ SCHWACH (ABER BESSER ALS BEI VERTEILTER STEUERUNG)
- NOCH WENIGER AUSFALLSICHER ALS BEI VERTEILTER STEUERUNG
- AUCH HIERARCHISCH VERWENDET: PROZESSOR-BUS, SYSTEMBUS, PERIPHERIE-BUS

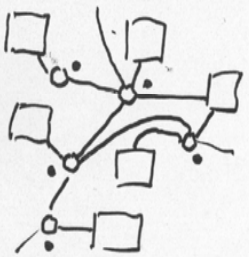
ÄHNLICH WIRKT EIN ZENTRALER SPEICHER ALS KOMMUNIKATIONS MEDIUM IN EINER MULTIPROZESSER- (multicore) ANLAGE



OHNE INTERPROZESSOR-ALARME NICHT BRAUCHBAR

1.7.5 Beispiele: Vermittlung (Indirekte) Netze

INDIREKT, VERTEILTE STEUERUNG, GEMEINSAME WEGE

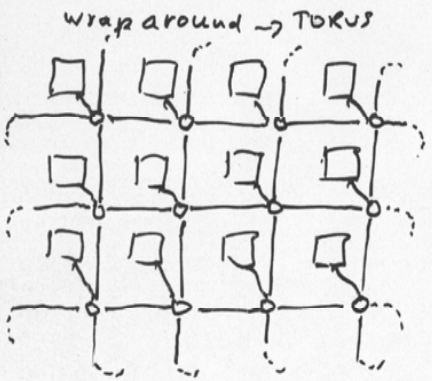


FERNNETZ (wide area network).

TYPISCH CHAOTISCHE STRUKTUR, GEPRÄGT DURCH VERTEILUNG DER NACHRICHTEN-STATIONEN (HIER "hosts") UND DER VERFÜGBAREN LEITUNGEN UND AUF-STELLUNGSORTE DER VERMITTLUNGS-RECHNER. IM INTERNET VIELE MANAGE-MENTDOMÄNEN, ÜBERGANGSPUNKTE (Internet Exchanges). GUTE ERWEI-TERBARKEIT. FEHLERTOLERANZ BEI REDUNDANTEN WEGEN (DANN WEGFINDUNGS (routing)-PROBLEM.

INDIREKT, VERTEILTE STEUERUNG, GEMEINSAME WEGE

ABER REGULÄR STATT CHAOTISCH:

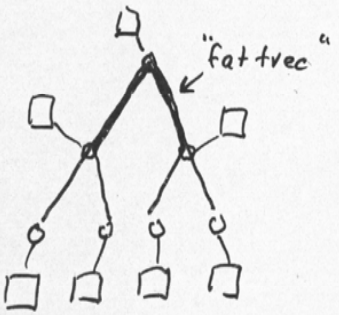


VERBINDUNGSWERKE IN MULTIRECHNERN

PARALLELRECHNER: TEIL DER PROGRAMMFUNKTIONALITÄT (WENIGSTENS IN LEISTUNGS PERSPEKTIVE

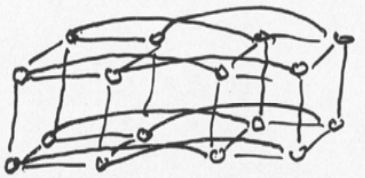
BEISPIEL GITTER/TORUS

KOSTEN $\sim n$, WEGLÄNGE $\sim \sqrt{n}$



BEISPIEL BAUM

KOSTEN $\sim n$ WEGLÄNGEN $\log n$



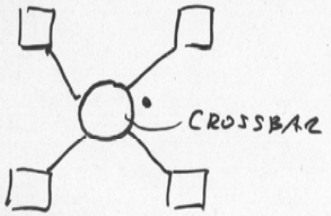
BEISPIEL HYPERCUBE

KOSTEN $\sim n.\log n$

WEGLÄNGEN $\sim \log n$

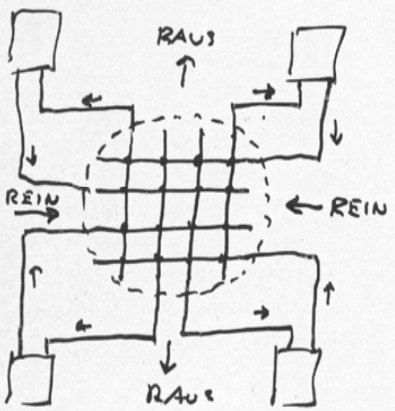
SCHLECHTE ERWEITERBARKEIT

INDIREKT, ZENTRALE STEUERUNG, ZUGEORDNETE WEGE



CROSSBAR (KREUZSCHIENENVERTEILER)
SCHALTET JEDE NACHRICHTENSTATION ZU JEDER ANDEREN DURCH, WENN DEREN ZUGANG FREI IST.

VERGRÖSSERT:

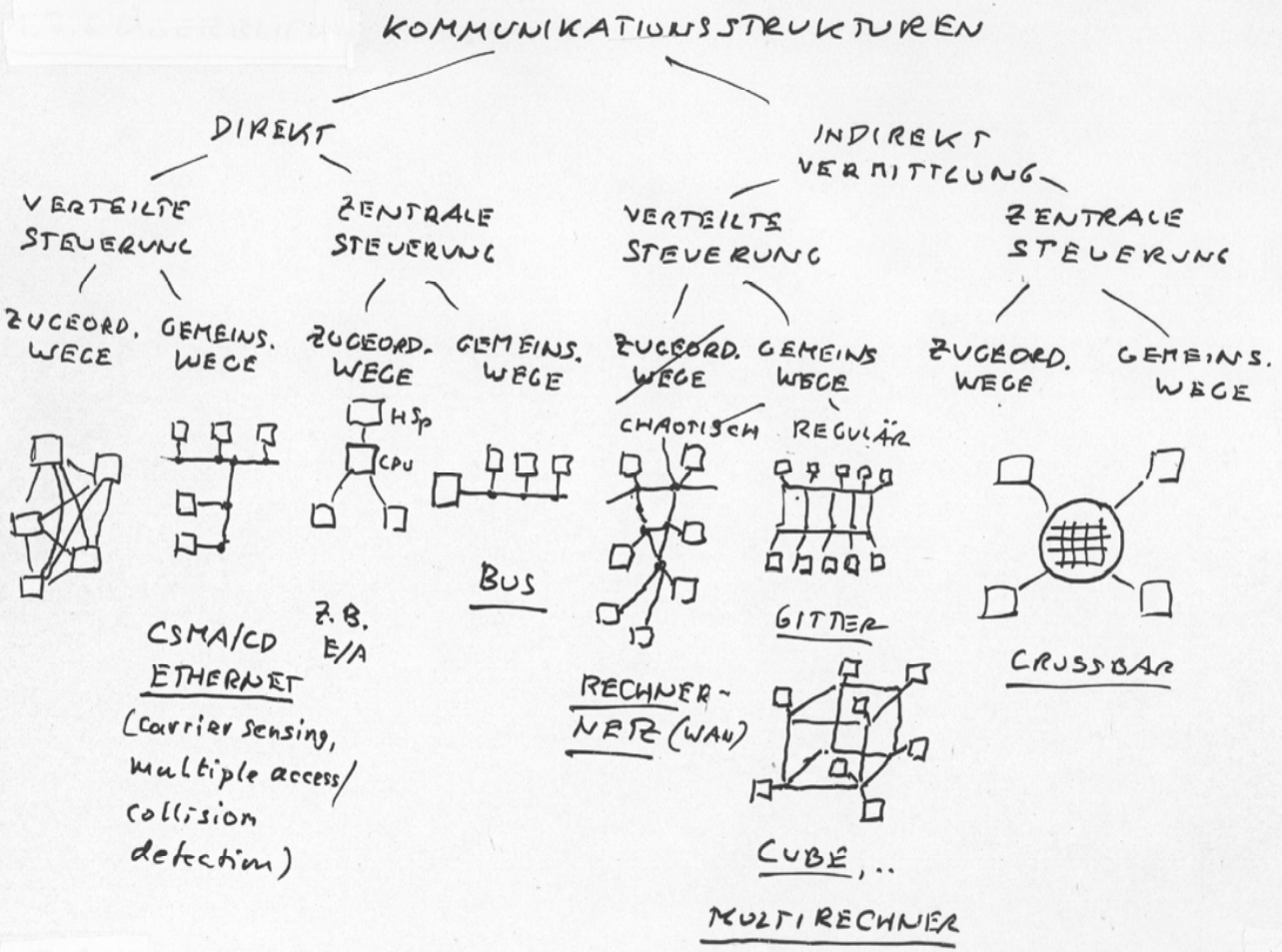


UM 1920 TATSÄCHLICH DURCH GEKREUZTE METALLSTÄBE GEBAUT (FERNSPRECH-VERMITTLUNGSTECHNIK)

KOSTEN $\sim n^2$ WEGLÄNGE $\mathcal{O}(1)$!

OFT HIERARCHISCH EINGESETZT.

1.7.6 Überblick: Kommunikationsstrukturen



1.8 Leistungsbewertung

1.8.1 Leistung, Zweck der Bewertung

Unter Leistungsbewertung (performance evaluation) versteht man die Ermittlung und Bewertung der Leistung von Rechensystemen (computing systems), d.h. Rechner und Betriebssoftware, oftmals einschließlich Anwendungsprozess. Leistungen sind:

- Zeitverhalten bei der Auftragsbearbeitung: Bedienzeiten: Ausführungszeiten von Befehlen, Speicherzugriffen, Programmen; Grenzdurchsätze (hier praktisch: Durchsatz bei hohem Auftragsangebot): Operationsdurchsatz der CPU bei Abarbeitung eines/vieler Programme; Busdurchsatz, Speicher-durchsatz, Leitungsdurchsatz, … und damit beschäftigt sich dieses Kapitel!

- aber auch: Funktionalität, Zuverlässigkeit, Benutzbarkeit (useability), Konfigurierbarkeit, elektrische (Verlust-)Leistung.

Zweck der Leistungsbewertung

- Entwurf von Rechensystemen bzw. Konfigurationen: Vorhersage der Leistung unter einer gegebenen Last, Entdeckung von Engpässen. Engpass (bottleneck) in einem System ist die Funktionseinheit mit der höchsten Auslastung; der Engpass bestimmt den Grenzdurchsatz des Gesamtsystems; meist bilden sich an ihm erhebliche Warteschlangen

- Tuning von Rechensystemen (Anpassung von Funktionseinheiten und Bedienstrategien an die Last)

- Auswahl von Rechensystemen

Kriterien hängen vom Einsatz ab! Vgl. Perspektive des Betriebes/ Benutzers Kap. 1.2!

1.8.2 Verfahren

E = Entwurf, T = Tuning, A = Auswahl. Aufwand steigt nach unten

Auswertung statischer Hardwareparameter	Laufzeitmessung bestehender oder künstlicher Programme	Messung des Betriebs bestehender Anlagen	Modellbasierte Verfahren
A, E Ausführungszeit von Operationen, Leitungsbandbreite A Kernprogramme („Kernels“)	A, T Benchmarks A, T Synthetische Programme	T Hardware-Monitore T Software-Monitore	E Analytische Modelle (algebraisch/ numerisch), für einfache Systeme E Simulations-Modelle
z.B. Additions/Multi-plikationszeit, Speicherkapazität, Speichergrenzdurch satz, „Mips“ aus Befehlsmixen	Benchmark: Messung der Gesamtausführungs-zeit einer Menge von „natürlichen“ oder synthetischen“ Programmen	Periodisch/ ereignisbezogen messen. Auswertung schritthaltend/ später. Sorgfältige Experimentplanung nötig!	Stochastisch oder „operational“. Simulation evtl. mit Anlagenteilen. Sorgfältige Experimentplanung wichtig!

Typische Sprechweisen, und noch ein wichtiger Begriff:

Elapsed Time, Wall Clock Time sind Verweilzeiten, sie können durch den Betriebsmittelwettbewerb (Multiprogramming / Multi threading) mit anderen Aufträgen bestimmt sein.

Ausführungszeit (execution time) von Elementaraufträgen in der Hardware sind als Bedienzeiten gemeint (Additions / Zugriffszeit / …)

Betriebsmittelzeiten (work) (CPU-Zeit, Plattenzeit, Denkzeit, aber auch Wartezeit) werden oft über die Verweilzeit eines Auftrags summiert angegeben. Z.B. UNIX Time Command: „90.7 s 12.9 s 1:59 65%“: User CPU __ System CPU time __ Elapsed time __ ratio (user CPU time/elapsed time)

1.8.3 Statische Hardwareparameter

Überwiegend bei der Entwicklung festgelegte Größen, die aber nur einen groben Anhalt für die Leistung geben:

- Befehlssatz, Datenformate
- Ausführungszeit der Befehle
- Taktfrequenz / Taktzeit der CPU
- Grenzdurchsatz der Bus-Systeme, Peripheriegeräte, Datenleitungen
- Kapazität und Zugriffszeit der Speicher (Cache 1/2/3, Hauptspeicher, periphere Speicher
- usf.

Diese Größen erfassen nicht den Einfluss (des Algorithmus, der Programmiersprache,) des Übersetzers, des Betriebssystems (Strategien, Overhead), der Parallelarbeit (positiv für „unseren“ Job: Parallelausführung eigener nebenläufiger Teile; negativ: Zeitmultiplex mit anderen Aufträgen, Wettbewerb um exklusive Funktionseinheiten).

Abgeleitete Größen:

Mixe: Gewichtete mittlere Ausführungszeit, z.B. Befehlsmix

$$T := \sum_{i=1}^{n} h_i * t_i$$

Befehle 1 .. i .. n

Befehlshäufigkeiten h_i: sind CPU/Compiler/anwendungsspezifisch

Befehlsausführungszeiten t_i

Formel berücksichtigt Parallelarbeit in CPU nicht!

Nicht mehr geeignet!

Cycles per Instruction CPI

Eine CPU habe einen Befehlsdurchsatz d. Dann werden Befehle mit einem mittleren Abstand a=1/d fertig, z.B. d=5 Gips = $5*10^9$ Befehle/s, also a=0,2 ns. Messen wir a in Taktzeiten, dann heißt es CPI (cycles per instruction). Keine Bedien(Ausführungs-)zeit ! Hat die CPU eine Taktfrequenz f von 1 GHz, dann ist die Taktzeit t=1/1 GHz=1 ns und CPI=a/t= 0,2ns/1ns = 0,2!

Die CPU-Zeit für die Ausführung eines Programms von n ausgeführten Befehlen („Instruction Count IC“) ist folglich

$T_{CPU}=n*a=n*CPI*t=n*CPI/f =IC*CPI/f$

T_{CPU} = IC * CPI/f

Instruction Count IC: hängt ab von Problem, Algorithmus, Sprache, Compiler, Befehlssatz

Cycles per Instruction CPI: hängt ab von Nebenläufigkeit, Parallelisierung (Kollateralität), Befehlssatz, Speicher/Cache-Zugriffszeit, Folge der Befehle

Clock frequency f: hängt ab von Technologie (Halbleiter/Schaltungstechnik, Kühlung, Packungstechnik)

CPI ist kein statischer Hardwareparameter!

1.8.4 Laufzeitmessungen an Programmen

Benchmark: Vergleichende Leistungsbewertung mehrerer Rechner (CPUs, …) durch Messung der Laufzeit von Programmen, oft zusätzlich Größen wie CPU/Speicherauslastung; dazu

- Programme im Quellcode, auf alle Zielmaschinen übersetzbar

- Rechner muss verfügbar sein

- für detailliertere Einsichten ist Monitoring notwendig:
  -- Hardware-Monitoring: eingeschränkte Sicht, keine Verfälschung des gemessenen Prozesses (falls Datenreduktion/-speicherung nicht auf demselben Rechner).
  -- Software-Monitoring: breite, auch anwendungsnahe Sicht, Verfälschung des gemessenen Prozesses.

Synthetische Benchmarks:

- Verwendung von Programmbausteinen, die jeweils eine gewünschte Teilbelastung (z.B. Gleitpunkt, Adressrechnung, Prozeduraufrufe, Plattenzugriffe) erzeugen und in Hinblick auf die gewünschte Gesamtlast zu einem Benchmark komponiert werden.
- Optimierende Compiler erkennen aber „toten“ Code und streichen ihn, also besser in Assembler.

Natürliche Benchmarks:

- „natürliche“ Programme/Programmbausteine
- Ergebnis wird (i.a. absichtlich!) auch durch Compiler und Betriebssystem beeinflusst: Güte des Codes/der Optimierung, Güte der Betriebsmittelzuteilung, Betriebssystem-“Overhead“

Beispiele von Benchmarks:

Whetstone: synthetisch, 9 numerikintensive Fortran-Moduln
Dhrystone: viele kleine Moduln, kein Gleitpunkt, Ada, C, Pascal
Linpack: Lösung eines Systems linearer Gleichungen, Grundlage der TOP 500 Liste (http://www.top500.org)
Lawrence Livermore Loops: Vektorisierung

Heute vor allem wichtig: Standardisierte Benchmarks:
SPEC-Benchmarks (Standard Performance Evaluation Corporation)
TPC-Benchmark (Transaction Processing Performance Council)
EEMBC-Benchmarks (Embedded Microprocessor Benchmark Consortium)

1.8.5 SPEC

Konsortium von Herstellern und Wissenschaftseinrichtungen (z.B. LRZ) mit dem Ziel produktunabhängige Standards zur Leistungs-bewertung zu definieren und die Ergebnisse zu verbreiten.

Untergruppen:

Open Systems Group (viele nennen sich so!): SPEC CPU (int/fp), Java Virtual Machine BM, SPEC MAIL, File Servers, Web Server.

High Performance Group: große industrielle Anwendungen, Parallelrechner (MPI, OMP)

Java Client/Server

Power (elektrische Leistung)

u.a.

SPEC – Philosophie von SPEC:

„The goal of SPEC is to ensure that the marketplace has a fair and useful set of metrics to differentiate candidate systems. The path chosen is an attempt to balance between requiring strict compliance and allowing vendors to demonstrate their advantages. The belief is that a good test that is reasonable to utilize will lead to a greater availability of results in the marketplace.

The basic SPEC methodology is to provide the benchmarker with a standardized suite of source code based upon existing applications that has already been ported to a wide variety of platforms by its membership. The benchmarker then takes this source code, compiles it for the system in question and then can tune the system for the best results. The use of already accepted and ported source code greatly reduces the problem of making apples-to-oranges comparisons.“

SPEC CPU Benchmarks (genauer genommen BM für Zentral-einheiten, da Hauptspeicher mit in die Bewertung eingeht; ohne ihn kein Programmlauf!)

SPEC CPU 2006: Weiterentwicklung des bisherigen SPEC CPU 2000:

- größerer BM (CPUs sind schneller)
- größere Datensätze (Caches sind schneller und größer)
- schlechtere Datenlokalität (ebenso)

SPEC CPU 2006: Satz von Programmen:

- Satz von Messungen der Bearbeitungszeit: SPECint2006, SPECfp2006

  Satz von Messungen des Durchsatzes: SPECint_rate2006, SPECfp_rate2006

  Diese Werte werden mit „aggressiver“ Optimierung gewonnen. Mit einfacher Optimierung heißen sie:

  SPECint_base2006, SPEChfp_base2006

  SPECint_rate_base2006, SPECfp_rate_base2006

- Die Messergebnisse werden reziprok relativ zur Bearbeitungszeit auf der Sun Ultra 10 angegeben (verbreitete Sun Workstation, 1998, 64 b, Ultra SPARCII CPU, 296 MHz, Solaris Betriebssystem), also

$$SPECint_x = \frac{\text{Bearbeitungszeit Sun Ultra 10}}{\text{Bearbeitungszeit auf Computer x}}$$

Um ein übergreifendes Maß für die Leistung nach SPEC zu haben, wurde das geometrische Mittel der so relativierten Werte über dem Satz von Programmen gebildet und SPECmark genannt. Noch unklar, ob auch Bestandteil von SPEC2006.

1.8.6 Modellbasierte Verfahren

Ein Modell eines Systems/eines Prozesses ist eine zweckmäßige Abstraktion zur Gewinnung von Erkenntnissen, typisch

- Erkenntnisse am realen System/Prozess nicht (einfach) gewinnbar
- es gibt ein Verfahren zur Gewinnung der Einsicht am Modell (z.B. Rechnung, Simulation, Betrieb (eines physischen Modells)).

Technisch brauchbare Modelle müssen kalibriert werden (die wichtigen technischen Parameter müssen in der Realität gemessen oder geschätzt werden) und validiert werden (die Gültigkeit muss durch Vergleich der „Antworten“ des Modells und der Realität gezeigt werden).

In der Leistungsbewertung 2 wesentliche Kategorien:

- operationales Modell: nur im Betrieb erfassbare Größen
- stochastisches Modell: (praktisch nicht streng nachweisbare Annahmen: Wahrscheinlichkeit, Unabhängigkeit, …)

Und drei Modellklassen:

- Einfachmodelle, z.B. Little‘s Formel, (operational/stochastisch)
- Stochastische Prozesse (meist vom Markov-Typ)
- Simulationsmodelle (operational/stochastisch)

Die beiden ersten haben den Vorteil, dass sie auch algebraische Resultate liefern (Einflussgrößen ablesbar), nicht nur numerische. Für komplexe Modelle sind nur Simulationen geeignet; ihre Planung und Auswertung ist schwierig, wenn die Ergebnisse aussagekräftig sein sollen.

Stoff für Spezialvorlesung.

1.8.7 Parametereinfluss im System, und ein wichtiges Einfachmodell

Eine grundlegende Frage beim Tuning eines Systems ist, wie die Systemleistung L (z.B. Grenzdurchsatz c oder Bedienzeit b durch Änderung von Werten von Systemparametern $p_1, \ldots p_i \ldots p_n$ beeinflusst wird. Falls L differenzierbar bezüglich der Parameter ist, gilt natürlich:

$$\Delta L = \sum_{i=1}^{n} \frac{\partial L}{\partial p_i} \cdot \Delta p_i \quad \text{für "kleine" } \Delta p_i$$

D.h.: Die Parameteränderungen summieren sich, aber jeweils gewichtet mit der Abhängigkeit der Systemleistung vom Parameter: $\frac{\partial L}{\partial p_i}$

(Wertvolle Einsicht, obwohl Differenzierbarkeit z.B. bei Rechnern nicht gegeben! 512 oder 1024 KB, 2 oder 4 Platten...)

(Parametereinfluss im System)

Für den Fall $L = c$ (Grenzdurchsatz des Systems) ist i.a. $\frac{\partial L}{\partial c_E}$ maximal (c_E Grenzdurchsatz des Engpasses). Tuning muss aber bei dem Parameter eingreifen, wo die gewünschte Verbesserung der Systemleistung mit kleinsten Kosten K herbeigeführt werden kann: Das bedeutet auch:

<u>Optimal</u> ist ein System dann, wenn für alle Parameter p das Verhältnis $\left(\frac{\Delta L}{\Delta K}\right)_p$ gleich ist! (Wäre das nicht so, dann müsste man „Geld" ΔK von einem Parameter wegnehmen und in den investieren, wo es mehr ΔL bringt!)

Klar, dass das nicht bloss für Computer gilt!

Wichtiges Einfachmodell: Amdahls Speedup-Formel für Parallelrechner (Gene Amdahl, 1967)

Für die Bedienzeit gilt oft: $b = \sum_{i=1}^{n} b_i$. Dabei ist b_i die gesamt in der Funktionseinheit i aufgewendete Bedienzeit des Auftrags ("work", vgl. UNIX-Time-Command).

Als Parameter p_i benutzen wir einen Beschleunigungsfaktor

$s_i = \frac{b_{io}}{b_i}$ (Speedup s_i: Verhältnis von Ausgangswert b_{io} zu beschleunigtem Wert b_i).

Also insgesamt (mit $b_0 = \sum_{i=1}^{n} b_{io}$ als Ausgangswert der Gesamtbedienzeit

$$b = \sum_{i=1}^{n} \frac{b_{io}}{s_i},$$

und der erzielte Speedup insgesamt ist

$$s = \frac{b_0}{b} = \frac{b_0}{\sum_{i=1}^{n} \frac{b_{io}}{s_i}} = \frac{1}{\sum_{i=1}^{n} \frac{1}{s_i} \frac{b_{io}}{b}} = \frac{1}{\sum_{i=1}^{n} \frac{1}{s_i} \cdot \beta_i}$$

mit $\beta_i = \frac{b_{io}}{b_0}$: ursprünglicher Bedienzeitanteil von i

Gene Amdahl (einer der IBM /360-Väter) wollte 1967 vor einer MIMD/SIMD-Euphorie warnen. Er rechnet vor, dass ein Job mit $b = b_1 + b_2$,

b_1 ideal parallelisierbar, d.h. $b_1 = \frac{b_{10}}{s_1} = \frac{b_{10}}{p}$ $(s_1 = p;\ p$ Werke$)$

b_2 nicht parallelisierbar, d.h. $b_2 = b_{20}$ $(s_2 = 1)$. $\beta_2 = G_2$ serieller Anteil

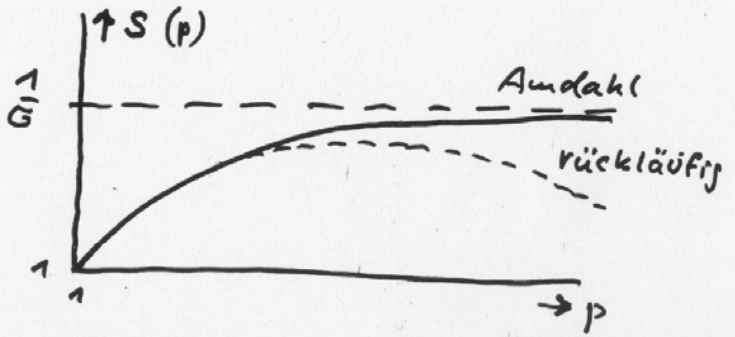
Also

$$s = \frac{1}{\frac{1}{s_1}(1-G) + \frac{1}{s_2}G} = \frac{1}{\frac{1}{p}(1-G) + G} = \frac{p}{1-G+p\cdot G} = \frac{p}{1+(p-1)\cdot G} \quad \|$$

Also $\lim_{p\to\infty} s(p) = \frac{1}{G}$

z.B. $G = 5\%$ nicht parallelisierbar:

$\lim_{p\to\infty} s(p) = 20$



Tatsächlich wachsen b_1 und b_2 typisch mit p, was das Ergebnis verschlechtert, bis zu rücklaufigem Speedup $s(p)$

1.9 Kurze Geschichte

Jahr	Rechnerentwicklung [Technologie]	Restinformatik
1936	[Fernsehen, Magnetband]	Church: Lambda-Kalkül Turing: Turing-Maschine Churchsche These
1941	Zuse Z3: programmgesteuerter, elektromechanischer Digitalrechner, Gleitpunkt	
1943	Colossus: Elektronischer Spezialrechner (Dechiffrierung)	
1946	v. Neumann u.a.: Konzept für speicherprogrammierbare Rechner; ENIAC (nicht programmierbar)	Zuse: Plankalkül, höhere Programmiersprache
1947	[Bipolarer Transistor]	
1949	Wilkes: EDSAC, v. Neumannrechner, Indexregister	
1951	Eckert, Mauchly: UNIVAC 1, industrielle Rechnerproduktion Wilkes: Konzept der Mikroprogrammierung	

Jahr	Rechnerentwicklung [Technologie]	Restinformatik
1954	[Kernspeicher]; Whirlwind	Assembler, einfaches Betriebssystem
1956	UNIVAC: RR 1103: Unterbrechung	Backus: Fortran
1957	MIT TX2: überlappte E/A, Güntsch: Konzept des virtuellen Speichers, Zuse: Konzept für Feldrechner. Sage-Rechnernetz	Mehrprogramm-Betriebssystem
1958	[Vorformen alphanumerischer Displays]	Algol
1959	Transistorrechner (z.B. Siemens S2002); von hier an rechnet „Zweite Generation“	
1960	RW 400: Fehlertolerantes Multirechnersystem	Lisp
1961	IBM-Stretch: Pipeline, [ECL] , IBM 1401	Cobol, Quicksort
1962	IBM 7090: Verdeckte Basisregister, Schriftleser, Wilkes: Konzept Cache-Memory	Teilnehmer-Betriebs-system, Petri: Petrinetze

Jahr	Rechnerentwicklung [Technologie]	Restinformatik
1963	Burroughs B 5000: Highlevel-Language-Machine, Mehrprozessor, segmentweise Adressierung. Ferranti Atlas: Seiten-adressierung, Traps. [Integrierte Schaltungen in der Raumfahrt]	
1964	Control Data 6600: asymmetrischer Multirechner, Hochleistungscomputer, parallele spezielle Rechenwerke, [Flüssigkeitskühlung], Operateur-Displays	BASIC
1965	IBM/360: Familie (1:300 Leistung), Großanwendung Mikroprogrammierung, „Solid Logic Technology“, von hier an rechnet „Dritte Generation“. DEC PDP8 „Minicomputer“	
1966	Texas Instruments: Taschenrechner. [Feldeffekt-Transistor gewinnt Bedeutung; Prinzip: Lilienthal 1930], [Integrierte Schaltungen in Computern]	Simula (1967), objektorientierte Programmiersprache

Jahr	Rechnerentwicklung [Technologie]	Restinformatik
1969	ILLIAC IV Feldrechner, [Halbleiterspeicher], IBM /360 Mod. 85: Cache-Memory	Relationale Datenbank
1971	[Telefax] , Intel 4004 Mikroprozessor, IBM /370, DEC PDP11: Unibus, ARPA-Netz	Pascal, Unix
1972	Diskette	C
1973	Xerox Alto: Workstation Syre: LAU-Datenflussrechner	Backus kritisiert v. Neumann-(Programmier)Stil
1974	[etwa 1974 Maus], Cray 1 (Vektorrechner)	Prolog, Capabilities
1975	IBM: Laserdrucker, Ethernet	Expertensysteme
1977	DEC VAX-11, Commodore PET1: Personal Computer (595 $)	Modula, TCP/IP
1978	Berkling: Reduktionsmaschine	Public-Key Kryptographie, OSI-Referenzmodell
1979		ADA

Jahr	Rechnerentwicklung [Technologie]	Restinformatik
1980	Symbolics Lisp-Rechner, IBM 801 RISC	
1981	Apple Lisa, IBM PC (mit Intel 8088), Osborne-1 (tragbarer Rechner, 11 kg)	Fensterorientierte graphisch/interaktive Benutzeroberflächen (Xerox-Entwicklung) MS-DOS
1982	Patterson: RISC, Hennessy: MIPS	
1983	Apple Macintosh (mit Motorola 68k)	C++
1984	Hillis: Connection-Machine: 64000 Rechenwerke, Sun-Workstations, Apollo-Workstations	
1985	Inmos Transputer, MIPS (RISC)	Verteilte Betriebssysteme
1986	Virtuell globaler Speicher	Viren
1987	Sun SPARC (RISC II)	

Jahr	Rechnerentwicklung [Technologie]	Restinformatik
1988	Laptop, Pasrsytec Supercluster (64 Rechenwerke)	OS/2
1989	Intel i860: Hochleistungs-Mikroprozessor mit RISC Basisprozessor, Nintendo Game Boy	
1990	FDDI-1: 100 Mbit/s Stadtnetz, ATM	
1991	Palmtop mit Schrifteingabe	Linux, Corba
1992	DEC Alpha: superskalarer Mikroprozessor, 150 Mips, 64 bit	MPI, PVM
1993	Intel Pentium: superskalarer Mikroprozessor 60 Mips Apple Newton (Palmtop, Schrifteingabe)	WWW, Windows NT
1994	Sony PlayStation 1 (100 Mio. verkauft)	Java
1995		Windows 95
1996	Pentium MMX (Multimedia Zusatz)	SETI@home

Jahr	Rechnerentwicklung [Technologie]	Restinformatik
1997	Intel/Sandia ASCI Red überschreitet 1 TFlops/s (Multirechner)	
1999	10^9 Computer, AMD Athlon (1 GHz)	
2000	Mobile Notebooks über Funk, Internet 10^6 Rechner, Intel Pentium 4 (Netburst: Superpipeline)	
2001	Intel Itanium 64b, EPIC/VLIW IBM p690 2 CPU je Chip	Web Services
2002	Intel Itanium II NEC Earth Simulator (Multivektorrechner, 36 TFlops) bleibt bis 2004 schnellster Rechner	
2003	AMD 64 erweitert Intels x86-Architektur auf 64b: AMD Opteron Intel schwenkt auf Multiprozessor („Multicore-“) Chips um, da für Einzel-prozessoren eine Leistungssteigerung an thermische Grenzen stößt	

Jahr	Rechnerentwicklung [Technologie]	Restinformatik
2004	Intel übernimmt AMDs 64b-Architektur IBM verkauft PC-Geschäft an Lenovo	„Web 2.0“
2005	AMD Athlon x2	
2006	Intel Core Duo; IBM Cell Processor (8 CPUs)	
2007	Sun Ultra Sparc (8 CPUs je Chip)	

1.10 Trends

Technologische Trends bei Computern sind über viele Jahrzehnte stabil und haben sich mit konstanten Faktoren (d.h. exponentiell) verändert. Sie werden i.a. auch derart über die nächsten 5 bis 10 Jahre so extrapoliert, allerdings ist bei (Ein-)Prozessoren seit 2003 eine deutliche Schwächung der Entwicklung zu beobachten. Strukturelle Trends und Anwendungstrends sind deutlich schwerer zu beurteilen. Also:

1.10.1 Technologische Trends

Wichtigste Trendgröße seit 1970:

Zahl der Transistorfunktionen je Chip: Wächst mit Faktor 1,6/a; d.h. Verdopplung in 1,5 Jahren (Mooresches „Gesetz“). Beruht weit überwiegend auf Miniaturisierung der elementaren Abmessungen: 10µ (Intel 4004, 1971) auf 65nm heute (Faktor 165 linear, 24.000 in der Fläche); in geringem Maße auf der Vergrößerung der Chipfläche.

Nutzbarkeit für CPUs begrenzt durch

- Mangel an strukturellen Konzepten, mit denen man mit dem Mehraufwand eine (möglichst überproportionale) Leistungs-steigerung erzielt
- Erwärmung

Taktfrequenz: Bei Ein-Chip-Prozessoren von 1971 von 0,4 MHz bis 2004 auf etwa 4000 MHz gestiegen: Faktor 10.000 (1,32/a). Aber seitdem nur noch langsamer Anstieg (Erwärmung!)

Grenzdurchsatz der Zentraleinheiten (also CPU + Cache + Hauptspeicher): Faktor 10**15 seit 1940 (10**5 durch Parallelisierung), also 1,67/a. Der Unterschied zur Taktfrequenz zeigt den Anteil der strukturellen Verbesserungen (trotz Hauptspeicher!)

Kosten je MIPS: Etwa um den Faktor 1,4 jährlich gesunken.

Hauptspeicherkapazität je Rechner: Etwa Faktor 1,44/a.

Hauptspeicherzugriffszeiten: 1,08/a (d.h. ohne Cache wären die Grenzdurchsätze der Zentraleinheiten nicht erreichbar.)

Plattenspeicher: Zugriffszeiten haben sich in Jahrzehnten kaum verändert, aber Kapazität und Durchsatz sind stark angestiegen: Seit 1970:

Kapazität von 25 MB auf 100 GB: Faktor 4000; 1,25/a

Grenzdurchsatz von 250 KB/s auf 100 MB/s: Faktor 400; 1,18/a

Zugriffszeit von 50 ms auf 1ms: Faktor 50; 1,11/a

Der relative Abstand zwischen den Zugriffszeiten der beiden wichtigsten Rechnerspeichermedien hat sich nicht wesentlich geändert (und ist viel zu groß!).

Anzahl der Computer: 1,48/a (wahrscheinlich höher, durch unzureichende Erfassung der eingebetteten Computer).

Durchsatz des Internets (heute ca. 2,5 · 10**20 b/a): Faktor 2,2/a.

Grenzdurchsatz eines Lichtleiters (heute 10**12 b/s)

aber während die technologischen und wirtschaftlichen Parameter jährlich etwa 1,3 bis 1,5 mal leistungsfähiger werden, ist die Größe größter noch einsetzbarer Softwaresysteme nur um 1,25/a angewachsen.

1.10.2 Qualitative Prognosen

Wir diskutieren den Stand von Prognosen von 1999 (Experten-tagung des Münchener Kreises):

Vorhersagen für 2010:

- Multimedia in mobilen Geräten
- 1 Tb/s auf einem Lichtleiter
- Internet überträgt in gesicherter Qualität (d.h. ohne Verzerrungen und Verluste im Paketfluss), „Quality of Service“
- Computerbedienung drastisch einfacher
- 1 Gb Speicherchip
- Online Sprachübersetzung
- Fachzeitschriften verschwunden
- 50% der Europäer haben einen Personal Digital Assistent
- 100 Computer je Haushalt in Europa
- Programmierung in natürlicher Sprache

Prognose von 1999 für 2015:

- Papierloses Büro
- Software Agenten weit verbreitet

Und erst später:

- Quantencomputer
- Computersteuerung durch Gestik, Blicke
- vollautomatische Fabriken
- Suchmaschine mit qualitativer Selektion

1.11 Strukturelle Freiheitsgrade

Wir stellen hier zusammen, welche grundsätzlichen strukturellen Freiheitsgrade für den Rechnerentwurf bestehen.

1.11.1 Ausführungsmodell

An der Hardware/Software-Schnittstelle sichtbare Funktionalität: Datenstrukturen, Operationen, Ablaufsteuerung. Bekannt geworden sind:

- Deklaratives Prinzip: Der Maschine sind Eigenschaften des gewünschten Ergebnisses beschrieben, z.B. in Prädikatenlogik. Über KI-Maschinen der 70er/80er Jahre hinaus (PROLOG-Maschinen) ohne Bedeutung geblieben. Potential für fehlerfreie Software, aber kein Schutz gegen Spezifikationsfehler; Maschine erreicht Ergebnis in sehr aufwendigen Suchprozessen.

- Funktionales Prinzip: Die Maschine berechnet Ausdrücke (bedingt und rekursiv, mit eingelagerten Funktionsdefinitionen). „Demanddriven“-Berechnung. Keine wiederverwendbaren Variablen an der HW/SW-Schnittstelle. Praktisch untauglich für Arbeiten mit großen Datenstrukturen. – Damit verwandt Datenfluss-Prinzip: „data driven“-Berechnung (sobald Daten vorliegen). Vgl. Kap. 1.3. Ebenso wenig für große Datenstrukturen geeignet, aber in „Mikroarchitektur“ eingesetzt (vgl. 1.4).
- Prozedurales Prinzip: Bis heute einziges realistisches Prinzip. An der HW/SW-Schnittstelle wird das Verfahren der Erzeugung des Ergebnisses beschrieben. Bleibt aber: auf welcher Höhe?
  - High Level Language Machines: Höhere Sprache mit HW/SW-Schnittstelle: ausgestorben, da zu speziell auf eine Sprache ausgerichtet. Aber Vorteil Faktor 2..3 an Speicherplatz und Zeit.
  - Objektorientierte Maschine: ausgestorben, da zu ineffizient.

1.11.2 Bindungszeitpunkt (eher ein Aspekt der Software-Architektur)

Der Algorithmus liegt primär in einer Nicht-Maschinensprache („höheren“ Sprache) vor. Die „semantische Lücke“ zu den elementaren Operationen (Transporte, Einfachverknüpfungen, …), Datenstrukturen und Ablaufsteuerung sowie zur Handhabung in der Mikromaschine (Caches, Parallelität) kann durch Abbildung zu verschiedenen Bindungszeitpunkten vollzogen werden:

- Programmierzeit
- Übersetzungszeit (Präprozessor, Compiler)
- Montage- / Ladezeit
- Laufzeit (Interpreter)

Meist werden mehrere Bindungszeitpunkte benutzt.

Beispiel: High-Level-Language Machine
- Makroersetzungen und/oder Bereitstellen von Bibliotheksprozeduren
- Compilation, z.B. auf stack-orientierte Zwischensprache
- Interpretation der Zwischensprache durch Mikroprogramm

Beispiel: Prüfung und Ersetzung der virtuellen Adressen
- zur Laufzeit nach Vorgaben des Betriebssystems (die i.a. auch erst zur Laufzeit gebildet werden)

Beispiel: Festlegung von Registeradressen
- durch Programmierer
- durch Compiler
- durch Prozessor (Renaming)

Beispiel: Abbildung z.B. eines Arrays auf Speicherzellen
- durch Compiler
- zusätzlich Überprüfung der Abbildung interpretativ zur Sicherung korrekter Zugriffe

1.11.3 Nebenläufigkeit und Parallelität (Kollateralität)

Grundlegender Freiheitsgrad auf allen Ebenen!
Schaltwerke, Mikroprogramm, von Neumann-Ebene (ISA Instruction Set Architecture: ILP Instruction Level Parallelism), Threads (in gemeinsamen Adressraum), Prozesse (getrennte Adressräume), virtuelle Maschinen.

Zweck:

- Bearbeitungszeit (Bedien-/Verweilzeit) des Auftrags
  - dazu Aufbrechen des Auftrags in kollateral ausgeführte Teilaufträge, z.B. Parallelrechner und/oder
  - Erhöhung des CPU-Durchsatzes (Little: d=f/y Vergrößerung von f, wenn y nicht verkleinerbar)
- um die Auslastung zu verbessern, z.B. CPU-Auslastung durch Einsatz von nebenläufigen Prozessen (Multiprogramming) oder Threads (Multithreading)
- um gewisse Bedienstrategien zu realisieren (z.B. Bevorzugung von Kurzaufträgen ohne Vorkenntnisse: viele nebenläufige Aufträge starten)

Zwei Stufen:

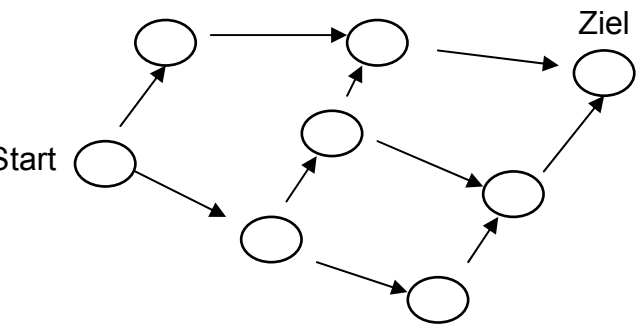
A Nebenläufigkeit bereitstellen:
Das kann der Programmierer, Compiler oder Prozessor tun!

B Nebenläufigkeit durch Parallelausführung nutzen:
Das kann Programmierer, Compiler, Betriebssystem, Prozessor tun. Voraussetzung: für Parallelität ausreichende Betriebsmittel!

A Bereitstellung von Nebenläufigkeit

Ersatzbild für komplexen (nebenläufigen) Auftrag:

Auftragssystem: (Menge der Aufträge, Präzedenzrelation) „kein Auftrag kann begonnen werden, wenn sein Vorgänger nicht fertig ist.“ Gründe:



Vorgänger muss Betriebsmittel freigeben (oft durch Bereitstellung ausreichend vieler Betriebsmittel lösbar

Auftrag darf einen Wert erst lesen, wenn alle präzedenten Schreiboperationen (read after write RAW)/ einen Wert erst schreiben, wenn alle präzedenten Schreib- und Leseoperationen (write afte write WAW, write after read WAR) auf dieser Variable abgeschlossen sind (auch in letzteren Fällen helfen zusätzliche Betriebsmittel (Speicher)).

Wenn aber keiner der Gründe vorliegt, kann die Präzedenz entfallen!

D.h. es wird Nebenläufigkeit erzeugt!

Beispiel: Ein typisches Maschinenprogramm: Es sieht sequentiell aus, enthält aber viel Nebenläufigkeit (überall wo keiner der Gründe (Betriebsmittelkonflikt, Lese-/Schreib-Präzedenzen) vorliegt.

Instruction Level Parallelism: Wichtiger Ansatz, anwendbar auf konventionelle Programme, liefert mäßige Nebenläufigkeit (mittleres f, etwa bis 30)

Beispiel: Operationen auf Arrays, z.B. Addition zweier Vektoren der Dimension k liefert k nebenläufige Additionen (wenn nicht hierfür ein Befehl verwendet werden kann). Kann hohe Nebenläufigkeit liefern (großes f).

Oft ist die Nebenläufigkeit schon durch den Programmierer bereitgestellt (bzw. durch den Erfinder des Algorithmus): Typisch bei Parallelrechnerprogrammen: Menge von Moduln (oft gleichartig), z.B. unabhängige Suche, Messwertverarbeitung, räumliche Aufteilung), Gesamtstruktur oft durch gegenseitige Ergebnisübernahme definiert, d.h. Inter-Modul-Kommunikation.

Wichtig! Hohe Nebenläufigkeit (denken Sie an die 9728 Itaniums der Altix 4700, die allerdings überwiegend partitioniert betrieben wird).

B Nutzung der Nebenläufigkeit durch Parallelität

Auf der Parallelität baut eine alte Rechnerklassifikation (Flynn 1971) auf: S (single), M (multi, parallel), I (instruction stream), D (data stream)

SISD	konventionelle Rechner: zu einem Zeitpunkt ein Befehlsstrom, der einen Datenstrom verursacht.
SIMD	Feldrechner, Vektorrechner: zu einem Zeitpunkt ein Befehlsstrom, der mehrere Datenströme auslöst, vgl. das Beispiel der Vektoraddition.
MIMD	Multiprozessor, Multirechner: zu einem Zeitpunkt viele Befehlsströme mit ihren Datenströmen (eigentlich M(SISD)).
MISD	tritt bei Graphik- und Signalprozessoren auf (eine Prozessorkette, durch die ein Datenstrom läuft)

Parallelität (Kollateralität: Mehr als ein Auftrag übernommen, aber nicht fertiggestellt) liegt unabhängig davon vor, ob die Aufträge im Zeitmultiplex (auch auf nur einer Funktionseinheit) ausgeführt werden oder auf vielen Einheiten (möglicherweise jeweils auf „ihren eigenen“: devoted resources). Die Nebenläufigkeit muss in beiden Fällen gegeben sein, damit das Ergebnis unabhängig vom zeitlichen Vorgehen ist.

Parallelität bedeutet auch, dass der „Status“ der parallel abgearbeiteten Aufträge bereitgehalten, evtl. kurzfristig ausgetauscht werden muss:

- Multiprogramming: Programme im Hauptspeicher oder „Swap“
- Multithreading: Zusätzlich mehrfache Registersätze
- Multirechner: Mehrfache Software oder Ineffizienz
- Pipelining / mehrfache Werke: Befehlszählerstand der Befehle und Operandenwerte, für Rekonstruktion; notwendig für präzise Unterbrechungen und für Rücksetzen bei gescheiterter Sprungvorhersage / spekulativer Ausführung.

Realisierungsformen der Parallelität, vgl. Bild!

Pipeline: Gut bei gleich strukturierten Aufträgen (RISC-CPUs).

Mehrfache Werke: Geeignet auch für wesentlich verschieden strukturierte Aufträge; Auslastung hängt stark von Anteil und Folgegesetzmäßigkeit der speziellen Aufträge ab (gemischtes Auftreten ist günstig!). Redundanz ergibt sich natürlich).

Vektorpipeline: Vektorrechner (waren?) sehr geeignet für array-orientierte Probleme (z.B. numerische Lösung partieller Differentialgleichungen). Operanden laufen im Strom aus Vektorregistern oder Hauptspeicher in die Pipeline.

Parallelrechner: Hier ist die Bedienzeit sehr großer Aufträge das Ziel. Die „Arbeit“ (work) ist die Summe aller CPU-Bedienzeiten des Auftrags. I.a. steigt w monoton mit p: w=w(p). Die Bedienzeit b=w(p)/p sinkt nur, so lange w(p) schwächer als p steigt! Ähnlich wirken Synchronisation und Kommunikation innerhalb des Auftrags. Es gibt im Allgemeinen eine optimale Parallelität $p_{opt} < k$, so dass b minimal wird: Partitionierung des Parallelrechners.

p: Parallelität („Füllung“)

k: Kapazität der Maschine

Zeitmultiplex von virtuellen Maschinen und Prozessen und Threads auf einer CPU: Es wird damit die Auslastung (und also Durchsatz, Kosten) verbessert, wobei die Verweilzeit steigt (bei Multiplex mehrerer Prozesse) bzw. die Bedienzeit sinkt (bei Multiplex „eigener“ Threads).

Verteilung von Prozessen auf mehrere CPUs/Rechner: Typische Strategie in großen „Mainframe“-Konfigurationen.

A ERZEUGUNG VON NEBENLÄUFIGKEIT: Streichung/Spreizung von Präzedenz

Pseudosequentielles Programm

Datenflussanalyse
Betr. Mittelkonflikte auflösen
Arrays, Schleifen nutzen
Spreizung: Code Motion, out of order-issue,
Auf CPUs: Sprungvorhersage, Unterbrechungsrekonstruktion

SIMD-Struktur

Vektorisierender Compiler
SIMD-Befehle
Vektorrechner-Programm

Explizit parallele Algorithmen

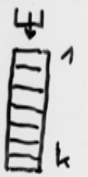
Modulen mit Kommunikations/Synchronisations-Abhängigkeiten
Parallelrechner-Programm

Prozesse, Threads

Prozesse vom Betriebssystem
Threads vom Programmierer (mit Kommunikations/Synchronisationsabhängigkeiten)

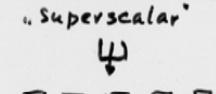
B NUTZUNG VON NEBENLÄUFIGKEIT: PARALLELITÄT DER KAPAZITÄT k

PIPELINE



$c = \frac{k}{b}$

MEHRFACHE WERKE „Superscalar"



u.U. jedes Werk eine Pipeline

$c = \frac{k}{b}$ BZW $c = \frac{k \cdot k'}{b}$

BEI k' PIPELINE STUFEN

VEKTOR-PIPELINE(S)

$c = \frac{k}{b}$

PARALLEL-RECHNER

$b = \frac{W}{k}$

W: SUMME DER MODULBEDIENZEITEN, „Work"

ZEITMULTIPLEX AUF EINER CPU: $g_{max} = 1$, BEI VERTEILUNG AUF k CPUs/RECHNER ZUSÄTZLICH

$C = \frac{k}{b}$

b BEDIENZEIT JE PROZESS/THREAD.

c GRENZDURCHSATZ
b (MITTLERE) BEDIENZEIT JE OPERATION

1.11.4 Ausschnitte

Dieser wichtige Freiheitsgrad, Handhabung der Speicherhierarchie durch das Ausschnittsprinzip wurde bereits in Kap. 1.5 behandelt.

2. Zentralprozessoren

2.0 Überblick: Instruction Set Architecture und Microarchitecture

vgl. 1.4.3: Sprachschichten und Auftragsübergänge!

Im Folgenden:

Instruction Set Architecture bzw. HW/Software-Schnittstelle:

Funktionalität, die über die Maschinensprache sichtbar wird: Formate, Register, maschinengestützte Datentypen, Befehle, Privilegierung, (nicht privilegierte) Adressierung, Unterbrechungen: Abschnitt 2.2 bis 2.7.

Tatsächlich ist das eine virtuelle Maschine, realisiert durch die Mikroarchitektur(-Maschine).

(hat nichts mit „Mikro“prozessor zu tun, auch mit „Mikro“sekunde nicht, wohl aber mit „Mikro“programm): Abschnitt 2.8 bis 2.10.

Mikromaschine benutzt von der ISA nicht sichtbare Techniken wie

- Parallelverarbeitung von Befehlen (Pipeline(s), parallele Werke/Superskalarität, out-of-order start/issue und commit der Befehlsausführung, spekulative Ausführung)
- Cache Memories
- Registerpool mit dynamischer Abbildung „sichtbarer“ Register
- Abbildung der vom Programmlauf erzeugten Adressen (Prozessadressen) auf Maschinenadressen
- Traps: nicht zulässige oder nicht direkt ausführbare Befehle werden in Unterbrechungsanforderungen überführt
- Kontextwechsel für Multithreading
- Steuerung der Taktfrequenz und Einschläfern von Prozessorteilen abhängig von der Auslastung

Die Trennung von ISA und Microarchitecture ist besonders klar bei den Intel- und IBM (360…zSeries)-Prozessoren, die eine ISA benutzen, die für hohen Prozessordurchsatz ungeeignet ist und eine komplexe Umsetzung auf die Mikromaschine erfordert.

Eine ältere Form dieser Umsetzung ist Mikroprogrammierung (vgl. 2.8), bei der die Operationcodes der Befehle Einsprünge in Mikroprogramme definieren: Ausführung durch Mikrointerpretation.

Aber heute

- dezentrale Verarbeitung vieler Befehle, asynchron
- dauert der „Speicher“ (d.h. real: Cache)-Zugriff nur wenige Takte, so dass eine zeitliche Auflösung durch einen Interpretationsvorgang nicht mehr möglich ist.

Daher (siehe Pentium 4): Übersetzung in Mikrocode!

2.1 Unsere Beispiele: Intel Pentium, die CISC/RISC-Wende, Sun Ultra Sparc, Intel Itanium

2.1.1 Intel Pentium

Intel wurde 1968 unter Beteiligung von IBM als Halbleiterhersteller gegründet. Unternehmensziel u.a. Second Source für IBM. Aber:

1971	Erster Mikroprozessor (Intel 4004)
1981	wählt IBM den Intel-Prozessor 8088 (16b-Architektur, später IA-16 genannt) als Basis für seinen PC, begründet damit Intels überragende Stellung im Prozessormarkt
1985	Intel 80386: IA-32
2001	Intel Itanium: 64b, nicht kompatibel zu IA 16/32, Architektur mit HP entwickelt, „IA-64“
2004	Intel übernimmt AMDs 64b-Architektur AMD64 als Intel „EM-64T“, später Intel 64 genannt
2006	Intel Core 2 Duo, erster Intel Multicore

Intel ist primär Bauelementehersteller.

Intel vermarktet Varianten der Prozessoren unter den Namen

Xeon: Hohe Leistung (hier im Wettbewerb mit Intel Itanium) durch mehr Cache Memory, schnelleren Bus und bessere Multiprocessing-Unterstützung

Celeron: Geringe und mittlere Leistung: Desktop-Systeme (PCs)

M bzw. Centrino-Chipsatz: Notebooks: Weniger Leistungsaufnahme

NAME	J/HR	TECHNOLOGIE	TRANSISTORF.	TAKT MHz	PINS	LEISTUNG W	WORTLÄNG. INTERN	DATEN BUS BREITE	REGISTER	BEFEHLE	CPU-DURCHS.	ADRESSIERG	
4004	71	p-MOS 10µ	2000										TASCHEN RECHNER CHIP
8008	72	p-MOS	3000	0,8	18	0,4	8	8	7x8b UNSYMM.	66	INTEL: „1"	16K	DISPLAY-CHIP. NICHT 4004-KOMPATIBEL
8080	74	n-MOS Depletion Load	6000	3	40	1	8	8	7x8b UNSYMM	111	INTEL: „10"	64K	UNIVERSAL-CHIP
8085	76	n-MOS Depl. L.	6500	5	40		8	8	7x8b UNSYMM	113		64K	
8086	78	High Density n-MOS Depl. Load	29000	8	40		16	16	8x16b UNSYMM	133	INTEL: „100"	4x64K in 1MB	ADRESSMODIFIKATIONEN. SEGMENTIERUNG NICHT GANZ 8080-KOMPATIBEL
8088	80						16	8	"	133		"	→ IBM PC (1981)
80186	82						16	16					ZUSÄTZL. E/A-STEUERG AUF CHIP
80286	82	µMOS	130 $\cdot 10^3$	10	68	3,6	16	16	8x16b UNSYMM	140	INTEL: „300"	REAL 4x 64K IN 1MB PROTECTED 1GB	PROTECTED MODE MIT SCHUTZMECHANISMEN REAL MODE → PC-AT
80386	85	CHMOS III 1,2µ	275. 10^3	16.. 33			32	32	8x32b SYMM	151	3,6 Mips	REAL, PROT. VIRTUAL 64TB	SEITENADRESSIERUNG MMU AUF CHIP
80386SX	86	"	"	16.. 20			32	16	"	"		"	AUSTAUSCH FÜR 80286
80486	89	CHMOS III 1,2µ IV 0,8µ	1,2 · 10^6	16.. 66	168		32	32	"	157	25 Mips (25MHz)	"	GLEITPUNKT-RW, MMU 8KB CACHE AUF CHIP
PENTIUM	93/95	BiCMOS 0,8/0,35µ	3.1 · 10^6	66/200	272	13	2x32	32	8x32b SYMM	163 (220?)	60/120?	"	2x8KB CACHE, 2 FESTP., 1 GLEIT PKT. PIPE, SPRUNGVORHERSAGE
PENT. MMX	96	BiCM. 0,35	$4,5 \cdot 10^6$	200	272	16	32,64	64	+MMX	277	140?	"	MMX MULTIMEDIA BEFEHLE
PENTIUM (PRO) II	96/98	BiCMOS 0,28µ	$7,5 \cdot 10^6$	200/450	528	30	32,64	64	8x32b RENAMING	277	300/500?	"	15 BEFEHLE Out of order MÖGLICH, 2x32KB CACHE
PENT. III	99/00	BiCMOS 0,25/0,18	8,2/24 · 10^6	450/1000	370	25	32,64	64	"	347	550/?/1100	"	40 BEFEHLE OOO, WEITERE MM-BEFEHLE

NAME	JAHR	TECHNOLOGIE	TRANSISTOR FKTN	TAKT GHz	PINS	LEISTG W	WORTLÄNGE b	BUS b	REGISTER	BEFEHLE	CPU-DURCHSATZ	ADRESSIERUNG	
PENTIUM 4	2000 – 2006	180–65 nm	42 – 376 · 10^6	1,3 .. 3,8	478	65 .. 115	32, 64	64	8×32 GP 8×80 FP	347 –		REAL, PROTECTED-ENHANC.	SLANGE PIPES, TRACE CACHE, WEITERE SSE, AB 2004 EM64T: 16×64, 16×80 FP "ENHANCED" ADRESSIM
PENTIUM M	2003	130 nm	55 · 10^6	0,9 .. 1,7			32	64	8×32 GP 8×80 FP				REDUZIERUNG DER ABWÄRME FEINGRANULARE ABSCHALTUNG
CORE DUO	2006	65 nm – 45 nm	151 · 10^6	1,5 .. 2,3			32, 64		16×64 GP 16×80 FP				PENTIUM 4 REDUZIERT E µARCH.: CORE µARCH, 2 CPU / CHIP
(WOODCREST)	2007	65 – 45 nm		1,7 .. 3		40 .. 80	32, 64		16×64 GP 16×80 FP				PENTIUM M + CORE µ ARCH.
ITANIUM 1	2001	180 nm	25	0,8		130	64		128×64 GP 128×80 FP				EPIC, BEDINGTE OPERATIONEN
ITANIUM 2	2002	130–90 nm	220 · 10^6	1,6	1296	100	64		128×64 GP 127×80 FP		9,6 Gips		

INTEL-VORHERSAGEN: MOORE SCHES GESETZ GILT FÜR NÄCHSTE 15 JAHRE
BIS 2010 10× LEISTUNG BEI GLEICHER ABWÄRME
MULTICORE WIRD STANDARD-STRUKTUR

GP: GENERAL PURPOSE, FP: FLOATING POINT, EPIC: EXPLICIT PARALLEL INSTRUCTIONS COMPUTER SSE: STREAMING SIMD EXTENSION

Pentium:

Datentypen:
Festpunktzahlen (8, 16, 32, 64b)
Gleitpunktzahlen (32, 64, 80b)
Multimediadaten (128b als 8/16/32b-Einheiten)
Dezimalstrings (1 oder 2 Ziffern je B)
Bitfelder (bis 32b)
Bitstrings, Bytestrings (bis $2^{32} - 1$ Elemente)
Adressen (32, 46b)

Register: (virtuell! Dynamisch auf physische abgebildet)
8 Allzweckregister zu 32b, auch als 8 Register zu 8, 16b verwendbar (Rücksicht auf ältere Prozessoren); seit EM-64T: 16 Register zu 64b.
8 Gleitpunktregister zu 80b (seit EM-64T: 16), Multimedia-Register, weitere Register

Pentium:

Befehle:

Wegen der vielen Varianten verschiedene Zählungen

Systembefehle:		32
Transporte:		25
Sprünge, Aufrufe, Programm. Unterbrechung:		9
Vergleiche, Tests, bedingte Sprünge und Transporte:		18
Schleifen:		4
Adressierung:		5
Andere:		8
Arithmetik:	Festpunkt	12
	Dezimal	6
	Gleitpunkt	97
		216

Pentium:

Befehle (Fortsetzung):

Übertrag:		216
Multimedia:	MMX	57
	SSE	70
	SSE 2	144
	SSE 3	?
	SSE 4	47
Flags, Conditions:		9
Konversion:		5
„Logik“ (und, oder, …):		4
Shift, Rotation:		16
Strings:		7
Bits:		6
INSGESAMT:		ca. 600

Mit dem „Woodcrest“-Prozessor werden typische Befehlsfolgen zu einem Befehl verschmolzen: Abermalige Erweiterung der Liste.

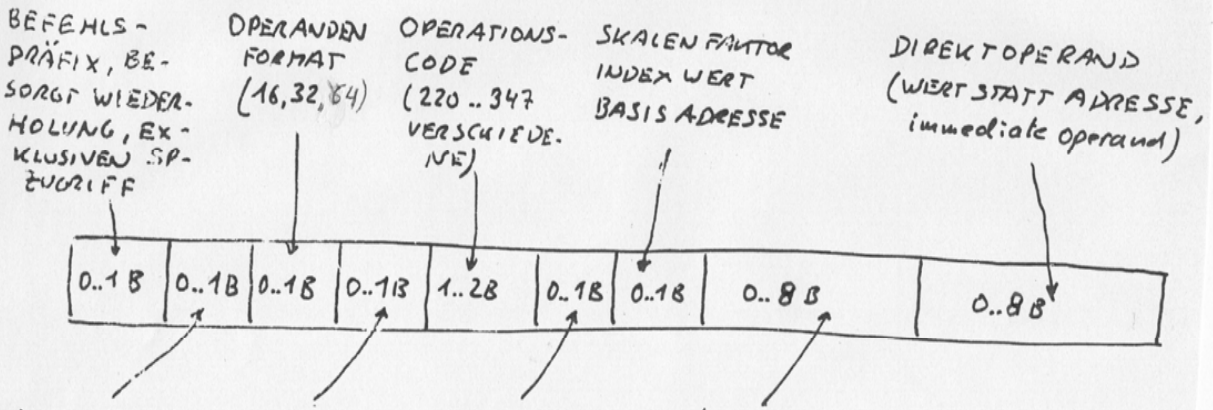
Pentium:

Prozessormodi

Protected: Benutzermodus mit 16- oder 32- oder 64-Bitadressen,
Prozessweise einstellbar, 8086-, 8086ff-virtuelle Maschinen.
Später als „Enhanced Mode“ erweitert.

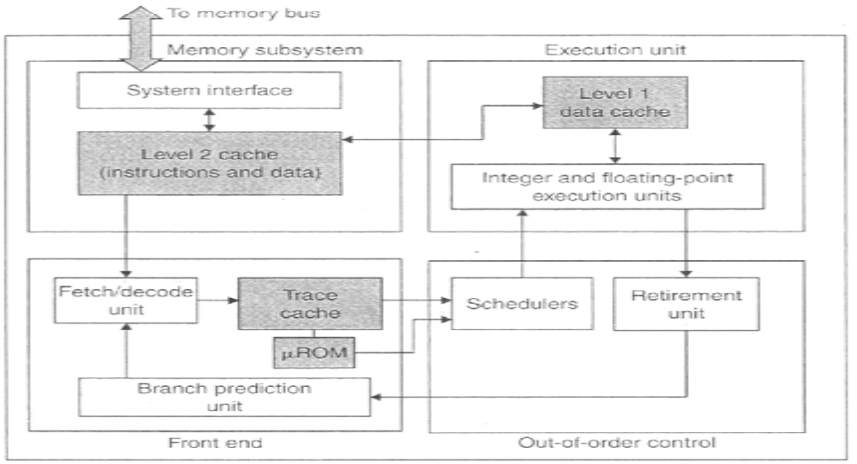
Real Address: Prozessor simuliert einen Intel 8086, 80286, 80386.

BEFEHLSFORMAT DES PENTIUM (AUSGESPROCHEN KOMPLEX)



ORT DES ERSTEN OPERANDEN VIELFACH IMPLIZIT IM OPERATIONSCODE.

Blockdiagramm Pentium 4 (Copyright: Tanenbaum)



Pentium:

Probleme der aufsteigenden Kompatibilität (ähnlich wie IBM/360, /370, /390, ESA, zSeries, aber besserer Ausgangspunkt von 1965): Im Pentium wird ISA und damit auch die Mikromaschine belastet durch das Gebot der Verträglichkeit („legacy software“). 8080/8086 bereits durch 8008 (1972) gebunden. In Intel xx86-Prozessoren steckt viel Aufwand und Ingenieurskunst, um von diesem Startpunkt 35 Jahre ein marktfähiges Produkt anzubieten. Im Einzelnen:

- Befehle variabler Länge mit komplexen Varianten: Aufwändige Decodierung; der beabsichtigte Vorteil (kompaktes Programm) ist wegen der stark gesunkenen Hauptspeicherkosten nicht mehr attraktiv.

(Weiter mit Pentium-Schwächen)

- Befehle verschiedener Ablaufstruktur und Dauer: Ungeeignet für Pipelining. Abhilfe: Pentium übersetzt (nicht: interpretiert!) Operationscodes, in Folgen von Mikrobefehlen, die RISC-artig sind.
- Hauptspeicherzugriff in sehr vielen Befehlen, statt Beschränkung auf Registerzugriffe (mit extra Lade/Speicherbefehlen, wie RISC): Bedeutet lange, schwankende (Speicherhierarchie!) Ausführungszeiten.
- Kleiner Registersatz, mit individuellen Rollen der Register: Kompliziert Codeerzeugung, macht Räumung von Registern zwecks Wiederverwendung notwendig. Kleiner Registersatz begrenzt Parallelität. Abhilfe: „Schattenregister“, auf die die vom Compiler vorgegebenen Registeradressen vom Prozessor abgebildet werden.

(Weiter mit Pentium-Schwächen)

- Unterbrechungen müssen „präzise“ sein, d.h. vor Eintritt in die Unterbrechungsbehandlung müssen alle „logisch“ vorherigen Befehle ausgeführt sein, die nachfolgenden müssen noch wirkungslos geblieben sein (oder zurückgesetzt).
- Die Komplexität verlangt Pipelines mit vielen Stufen (bis 21 Pentium 4, „Netburst“-Mikroarchitektur). Damit wird der Verlust bei Abbruch der Verarbeitungsfolge, bei Eintritt eines (falsch vorhergesagten) Sprunges entsprechend groß.

Weitere Behandlung des Pentiums:

Instruction Set Architecture in Kapitel 2.2 bis 2.6: Pentium als Beispiel.

Mikroarchitektur in Kapitel 2.9 und 2.10 (Cache und Parallelität)

PENTIUM/XEON HEUTE, IM KREIS SEINER KOLLEGEN:

July 30, 2007

PDF Version

Chart Watch: Server Processors

COPYRIGHT:
MICROPROCESSOR REPORT

Processor	AMD Opteron 2222SE	AMD Opteron 8222SE	Intel Pentium Extrm Ed 965	Intel Core 2 X6800	Intel Xeon 5160	Intel Xeon 5355
Bit-width	32/64-bit	32/64-bit	32/64-bit	32/64-bit	32/64-bit	32/64-bit
Cores/chip x Threads/core	2 x 1	2 x 1	2 x 2	2 x 1	2 x 1	4 x 1
Clock Rate	3.00GHz	3.00GHz	3.73GHz	2.93GHz	3.00GHz	2.66GHz
Cache: L1-L2-L3 - I/D or Unified	2 x 64K/64K - 2 x 1M - N/A	2 x 64K/64K - 2 x 1M - N/A	2 x 16K/12K - 2 x 2M - NA	2 x 32K/32K - 2 x 4M - NA	2 x 32K/32K - 4M - NA	4 x 32K/32K - 8M - NA
Execution Rate/Core	3 x86 instr	3 x86 instr	3 uOPs	4 uOPs	5 uOPs	5 uOPs
Pipeline Stages	12/17 stages	12/17 stages	31 stages	14 stages	14 stages	14 stages
Out of Order	72ROPs	72ROPs	128 uOPs	96 uOPs	126 uOPs	126 uOPs
Memory B/W	8.5GB/s	8.5GB/s	8.5GB/s	8.5GB/s	10.5GB/s	10.5GB/s
Package	LGA-1207	LGA-1207	LGA-775	LGA-775	LGA-771	LGA-771
IC Process	90nm 9M SOI	90nm 9M SOI	65nm 8M	65nm 8M	65nm 8M	65nm 8M
Die Size	230mm2	230mm2	162mm2	143mm2	143mm2	286mm2
Transistors	227 million	227 million	376 million	291 million	291 million	582 million
List Price (intro)	$873	$2,149	$1,207	$999	$851	$1,172
Power (Max)	120W(MTP)	120W(MTP)	130W(TDP)	75W	80W	120W
Availability	2Q07	2Q07	2Q06	3Q06	2Q06	4Q06
Scalability	1–2 chips	2–8 chips	1 chip	1 chip	1–2 chips	1–2 chips
SPECint/fp2006 [cores]	13.5 / 14.3 [2]	11.2 / 13.0 [8]	11.7 / 12.7 [2]	18.5 / 16.8 [2]	18.9 / 17.1 [4]	17.3 / 16.2 [8]
SPECint/fp2006_rate [cores]	50.8 / 49.4 [4]	96.1 / 93.0 [8]	23.1 / 21.7 [2]	31.1 / 26.8 [2]	60.0 / 44.1 [4]	92.6 / 88.9 [8]
Architecture Status	Active	Active	Inactive	Active	Active	Active

Processor	Intel Itanium 2 9050	IBM Power6	IBM Power5+	Fujitsu SPARC64 VI	Sun UltraSPARC IV+	Sun UltraSPARC T1
Bit-width	64-bit	64-bit	64-bit	64-bit	64-bit	64-bit
Cores/chip x Threads/core	2 x 2	2 x 2	2 x 2	2 x 2	2 x 1	8 x 4
Clock Rate	1.60GHz	4.70GHz	2.20GHz	2.40GHz	1.95GHz	1.20GHz
Cache: L1-L2-L3 I/D or Unified	2 x 16K/16K - 1M/256K - 12M(on)	2 x 64K/64K - 2 x 4M - 32(off)	2 x 64K/32K - 1.92M - 36M(off)	2 x 128K/128K - 5M - NA	2 x 64K/64K - 2M - 32M(off)	8 x 16K/8K - 3M - N/A
Execution Rate/Core	6 issue	7 issue	5 issue	4 issue	4 issue	1 issue
Pipeline Stages	8 stages	13 stages	15 stages	15 stages	14 stages	6 stages
Out of Order	None	"limited"	200 instr	64	None	None
Memory B/W	8.5GB/s	75GB/s	12.8GB/s	8GB/s	4.8GB/s	25.6GB/s
Package	mPGA-700	N/A	MCM-5370 pins	412 I/O pins	FC-LGA 1368	1933 pins
IC Process	90nm 7M	65nm 10m	90nm 10m	90nm 10M	90nm 9M	90nm 9M
Die Size	596mm2	341mm2	245mm2	421mm2	335mm2	378mm2
Transistors	1.72 billion	790 million	276 million	540 million	295 million	279 million
List Price (intro)	$3,692	N/A	N/A	N/A	N/A	N/A
Power (Max)	104W	~100W	100W	120W	90W	70W
Availability	3Q06	2Q07	4Q05	2Q07	3Q06	4Q05
Scalability	1–64 chips	2–32 chips	1–32 chips	4–64 chips	1–72 chips	1 chip
SPECint/fp2006 [cores]	14.5 / 17.3 [2]	17.8 / 18.7 [1]	10.5 / 12.9 [1]	9.7 / 11.1 [32]	N/A	N/A
SPECint/fp2006_rate [cores]	1534 / 1671 [128]	410 / 379 [16]	197 / 229 [16]	1111 / 1160 [128]	1120 / N/A [144]	N/A
Architecture Status	Active	Active	Inactive	Active	Inactive	Active

All SPEC scores are base.

The table above gives the vital statistics for the key high-end processors available.

2.1.2 Die CISC-RISC-Wende

Geschah zwar schon 1980, ist aber noch immer wichtig zu verstehen, weil sie zeigt, wie es zu Umbrüchen in Prozessor-konzepten kommen konnte und kann.

Hauptlinie der Rechnerfunktionalität war von 1960-1980 gekennzeichnet durch:

a. Aufstieg von wenigen zu vielen Datentypen/formaten und Befehlen
b. Bildung von Komplexbefehlen für häufige Operationsfolgen, z.B. Schleifen, Prozeduraufrufen, Prozesswechsel
c. Annäherung an Semantik höherer Sprachen
d. Bildung komplexer Privilegierungsstrukturen zugunsten des Betriebssystems
e. Vielschichtige Ausschnittsbildung
f. Bitsparende Befehlscodierung

Die Folgen von a-c, f waren:

- Vielfältig formatierte Befehle unterschiedlicher Ablaufstrukturen.
- Compiler können die komplexe Funktionalität nicht wirkungsvoll ausnutzen in der Codeerzeugung.
- Für den technologischen Stand von 1980 (40.000 Transistoren je Chip) waren die Prozessoren zu komplex (Intel 16b-CPUs kein Widerspruch!).

Um 1980 schwächten sich wichtige Voraussetzungen der bisherigen Entwicklung ab:

- Hauptspeicher war nicht mehr teuer: lieber einfache Decodierbarkeit als Biteinsparung bei den Befehlen.
- Verbreitung von UNIX und höheren Sprachen: Kompatibilität mit Altprozessoren weniger wichtig.
- In den aufkommenden Workstations weniger Kontextwechsel als auf Großrechnern: große Registersätze zulässig.
- Wettbewerb verlangt kürzere Entwicklungszeiten.

Neuer Ansatz RISC (Reduced Instruction Set Computer), (eine irreführende Bezeichnung).
IBM 801 (Cocke), RISC (Patterson), MIPS (Hennessy).
Schmähwort gegen die alten Architekturen (z.B. IBM/370, Intel, DEC VAX) CISC (Complex Instruction Set).

RISC: Tatsächlich Übergang von den heterogenen, teils komplexen, vielformatigen Befehlen zu schlichten Befehlen gleichen Formats und gleicher Ablaufstruktur, die sich auf einer Pipeline aber mit hohem Durchsatz verarbeiten lassen, und

- Operanden und Ergebnisse auf großem uniformen Register-Block,
- Speicher-Register-Verkehr durch vor/nachlaufende Lade/Speicher-Befehle (load/store), und
- die einfachen Befehle können ohne Mikroprogrammsteuerung realisiert werden.

Heutiger Stand:

Seit 1980 sind alle neuen Prozessoren als RISC-Prozessoren implementiert. Entweder

- Ist ihre ISA bereits als RISC konzipiert (unser Beispiel Sparc und fast alle seine Konkurrenten, einschließlich des auf den ersten Blick unähnlichen Itaniums (VLIW))
- Oder ein RISC simuliert die CISC-Funktionalität (ab Pentium Pro sind die Intelprozessoren superskalare Prozessoren, auf denen die überkommenen Befehle einzeln in den Code einer unterliegenden RISC-Mikromaschine übersetzt werden

Kann sich eine Wende wie CISC-RISC wiederholen?

Ja!

Dringend brauchen wir ein Konzept, mit dem wir aus k mal mehr Transistorfunktionen k-fache Leistung (oder noch mehr) rausholen. Die Multicores bleiben i.a. unter dem Faktor k! Und k wächst in den nächsten 10 Jahren mit Moore‘s Gesetz weiter, und 1,6**10 ist 110!

Wir kommen darauf zurück.

2.1.3 Sun Ultra SPARC III

Um 1980 wurden an amerikanischen Hochschulen Prototypen leistungsfähiger Maschinen gebaut, die als Wissenschaftler-Einzelarbeitsplatz bestimmt waren (Workstations), den Ansätzen für Personal Computer an Leistung weit voraus. Ausstattung mit UNIX und Internet-Protokollen.

Wichtiger Offspring:

Sun (Stanford University Network) Microsystems

(Gründer u.a. Andreas von Bechtolsheim, studiert an der TUM Elektrotechnik/Datentechnik und Carnegie-Wellon University, später bei CISCO; einer der ersten Google-Investoren).

Erstes Produkt 1981 Sun-1 (auf Motorola 68020)

1982 ff:	Sun-2, Sun-3, große geschäftliche Erfolge
1987:	eigene Prozessorarchitektur: SPARC (Scalable Processor Architecture): 32b, 69 Befehle, 128 Register (!), von denen 32 direkt zugreifbar und 24 weitere (als „Register Window“) variabel zuschaltbar sind (schneller Kontextwechsel!). Sun stellt die Chips nicht selbst her! Prozessor-Architektur ist „offen“, d.h. von anderen lizenzfrei nutzbar.
1995:	UltraSPARC 64b
2001:	UltraSPARC III: unser Beispiel

UltraSPARC III

Zielsetzung multimediale Workstation und Server-Prozessor
(zentraler Teil des Sun-Geschäftes)

- 2001 (Weiterentwicklungen IIIi und IV (Dual Core) in 2003…2005
- 130/90nm, 29/90 Mio. Transistoren, 53/108 W, 1368 pins
- 128 Register (32 direkt und 24 als verschiebbares Window), 64b
- zu den anfänglichen 69 Befehlen zusätzlich: VIS (visual instruction set, 23 weitere Befehle, für Multimedia)
- 0,6 .. 2,1 GHz Takt
- getrennter Cache 1 für Befehle und Daten, Cache 2 (8 bis 16 MB), groß, aber nicht auf dem Chip!
- 4 Befehle gleichzeitig startbar, 8 Pipelines (2x Integer, 2x Floating Point, 1 Load/Store, 1 Verzweigung)

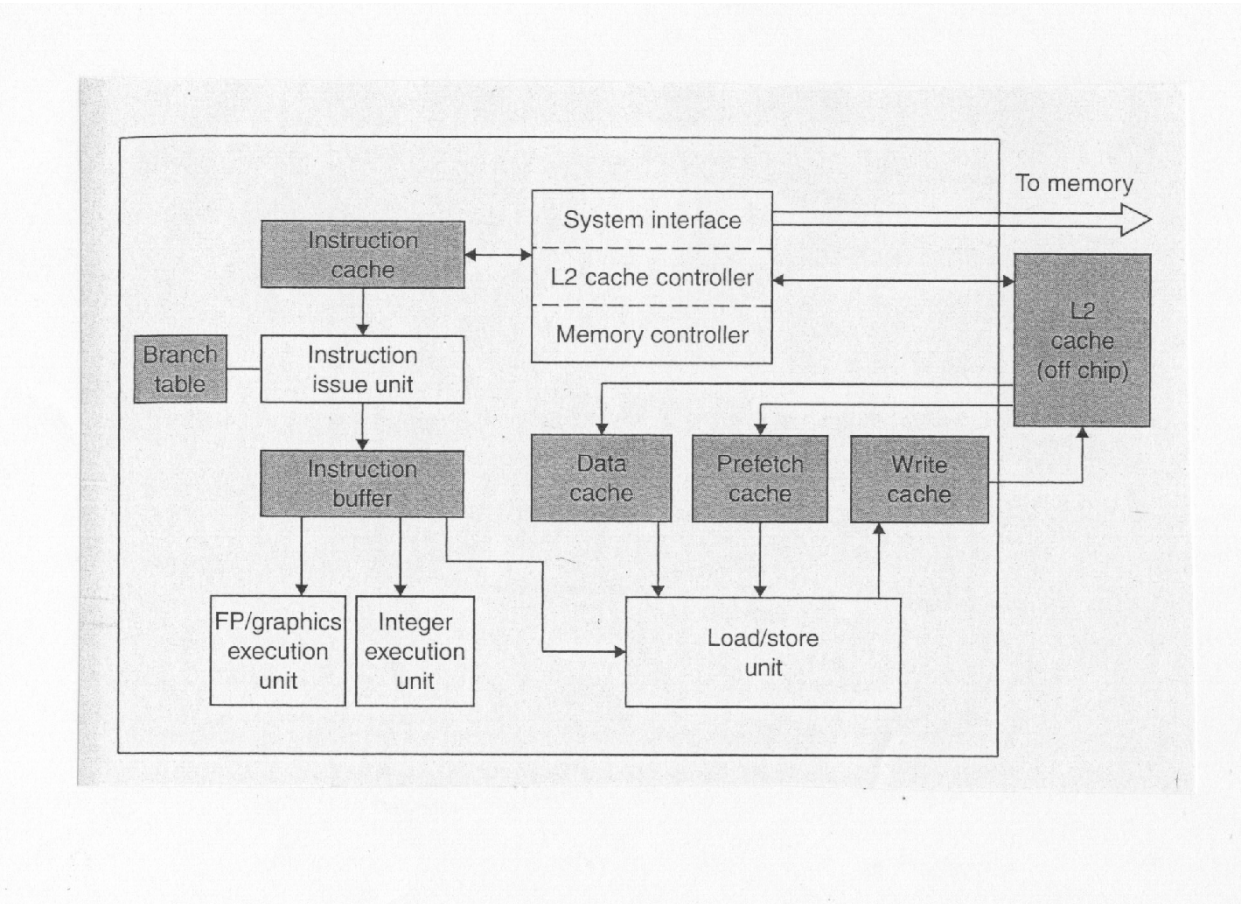
Weiterentwicklung zu Multicore –Strukturen:

2005/2006:

UltraSPARC T1/T2, „System on a Chip“ (8 Cores, 32/64 Threads, Ethernet, PCI (Peripheral Component Interconnect), Chiffrierung auf dem Chip). Grenzdurchsatz T2 nch SPEC int rate 14x/35x UltraSPARC III. T1-Chip hieß „Niagara“.

Dem hohen Thread-Grenzdurchsatz steht allerdings keine entsprechend kurze Bedienzeit für einen einzelnen Thread gegenüber!

Sun UltraSPARC III Prozessorstruktur



2.1.4 Intel Itanium (IA-64)

Angesichts der 64b-RISC-Prozessoren (DEC Alpha, IBM Power, MIPS, SPARC) erarbeiteten Hewlett Packard und Intel ab ca. 1995 eine neue Prozessor-Architektur, die langfristig mehr Parallelität bei weniger Komplexität bieten sollte: IA-64, mit Migrationshilfen bzw. teilweiser Kompatibilität mit HP PA-RISC (Precision Architecture) und Intel IA-32.

Der Anlauf des Produkts (Intel „Merced“, 2001) wurde durch geringe Leistung und Compilerprobleme erschwert. Danach „McKinley“, …, „Madison“, Montecito“ (Dual Core, siehe Altix 4700), „Montvale“. Intel wurde durch AMD gezwungen, eine 64b-Fortsetzung der x86-Linie (IA-32) anzubieten; die IA-32-Emulation auf Itanium war ineffizient und der Prozessor zu teuer.

Itanium: Hauptziele (1)

Nebenläufigkeit im Compiler erzeugen! Steuer- und Datenflussanalyse (control/data flow analysis) und Betriebsmittelplanung erlaubt es, „Gruppen“ (groups) von (im Wesentlichen) nebenläufigen Befehlen zu definieren:

EPIC (Explicit Parallel Instruction Processor).

Die Gruppen werden hintereinander ausgeführt, wenn der Prozessor nicht erkennt, dass er die nächste Gruppe vorzeitig angreifen darf.

Die Zuweisung von 32 Registern erfolgt durch den Compiler, weiterer dynamisch.

Vorverlegung des Bindungszeitpunktes!

Itanium: Hauptziele (2)

Vorverlegung des Bindungszeitpunktes bedeutet grundsätzlich:

- weitergreifende Optimierung am Programm möglich
- aber keine Information über die aktuelle Lage bei der Ausführung (z.B. Cache/Speicherzugriffszeiten)
- Effizienzvorteil bei Mehrfachausführung (wie Übersetzer/ Interpreter).

Itanium: Hauptziele (3)

(Nebenläufigkeit vom Compiler!)

- Der Compiler gibt dem Prozessor Empfehlungen (Hints) zum voraussichtlichen Ablauf, z.B. Vorziehen von Ladeoperationen oder ob ein bedingter Sprung ausgeführt wird oder nicht, und welches Sprungziel genommen wird. Weitere Unterstützung bei bedingten Sprüngen durch bedingte Operationen.

Aber die Hoffnung, dass durch Verlagerung der Erzeugung von Nebenläufigkeit in den Compiler der Prozessor stark entlastet würde, erfüllt sich nur teilweise, z.B. muss der Itanium-Prozessor trotzdem eine dynamische, d.h. empirische Sprungvorhersage benutzen.

Itanium: Hauptziele (4)

(Nebenläufigkeit vom Compiler!)

- Compiler setzt optimistisch auf Nebenläufigkeit in gewissen unklaren Situationen: vermutlich notwendige Ladeoperationen aus dem Hauptspeicher werden früh gestartet, notfalls wirkungslos gemacht.
- An der ALAT (Advance Load Address Table) wird überprüft, ob auf die gelesene Größe präzedente Schreiboperationen ausgeführt werden.
- Wenn eine vorgezogene Ladeoperation einen Alarm auslöst (z.B. unzulässige Adresse), wird sie abgebrochen, im Zielregister wird der Wert „NaT“ (Not a Thing) vermerkt, und die Alarmbehandlung, die ja schwerwiegende Seiteneffekte haben könnte, erst ausgeführt, wenn die Ladeoperation wirklich vollzogen werden muss.

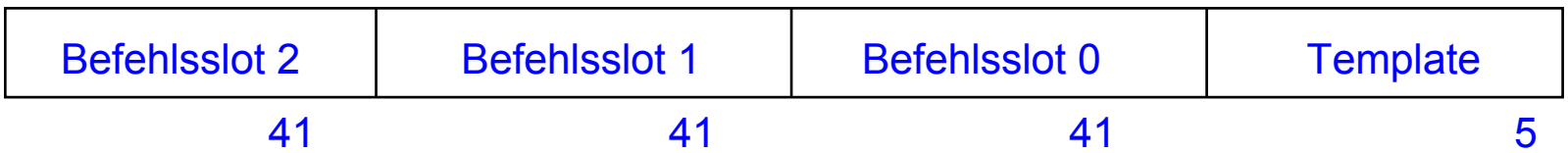
Itanium: Hauptziele (5)

Mehr Parallelität im Prozessor bieten!

- Befehle in Bündeln (bundles) zu dritt gespeichert (VLIW: Very Long Instruction Word), bis zu zwei Bündel gleichzeitig gestartet
- 3 mal 128 Register und Sonderregister
- 11 Pipelines
- breite externe Schnittstellen, insbesondere zum Hauptspeicher

Itanium: Befehle in Bündeln

Befehlsformat: 128b-Bündel



Die Befehle eines Bündels sind nebenläufig und werden gleichzeitig gestartet, dank Planung des Compilers.
Das Template erklärt:

- Welcher Befehl braucht welche Ausführungseinheit?
- Gibt es im Bundle eine Gruppengrenze („Stop“)?
- Belegen Befehle mehr als einen Slot?

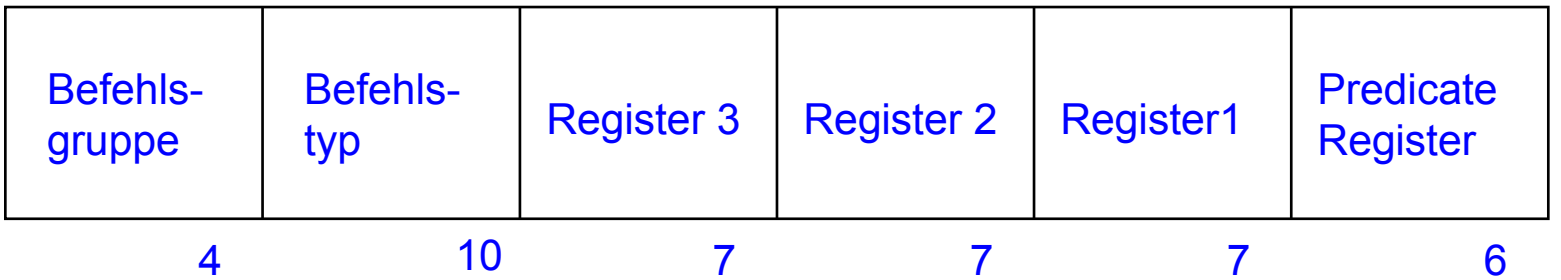
Itanium: Ausführungseinheiten, Art und Anzahl (Itanium II)

I-Einheit (Anzahl 2)	für A-Befehle (Festpunkt-Arithmetik)
	für I-Befehle (Festpunkt, nicht Arithmetik)
M-Einheit (Anzahl 4)	für A-Befehle
	für M-Befehle (Speicherzugriff)
F-Einheit (Anzahl 2)	für F-Befehle (Gleitpunkt)
B-Einheit (Anzahl 3)	für B-Befehle (Sprünge)
L+X-Einheit	für Befehle mit 64b-Direkt-Operand und Sonderfälle

Die Pipelines sind nur 8-stufig.

Itanium: Befehlsformat

Beispiel: Verarbeitungsbefehl



Register: eines aus 128 allgemeinen Registern (GR)

Predicate Register: eines aus 64 Bedingungsregistern (Predicate Register, je 1b), bestimmt, ob der Befehl ausgeführt wird oder wirkungslos bleibt

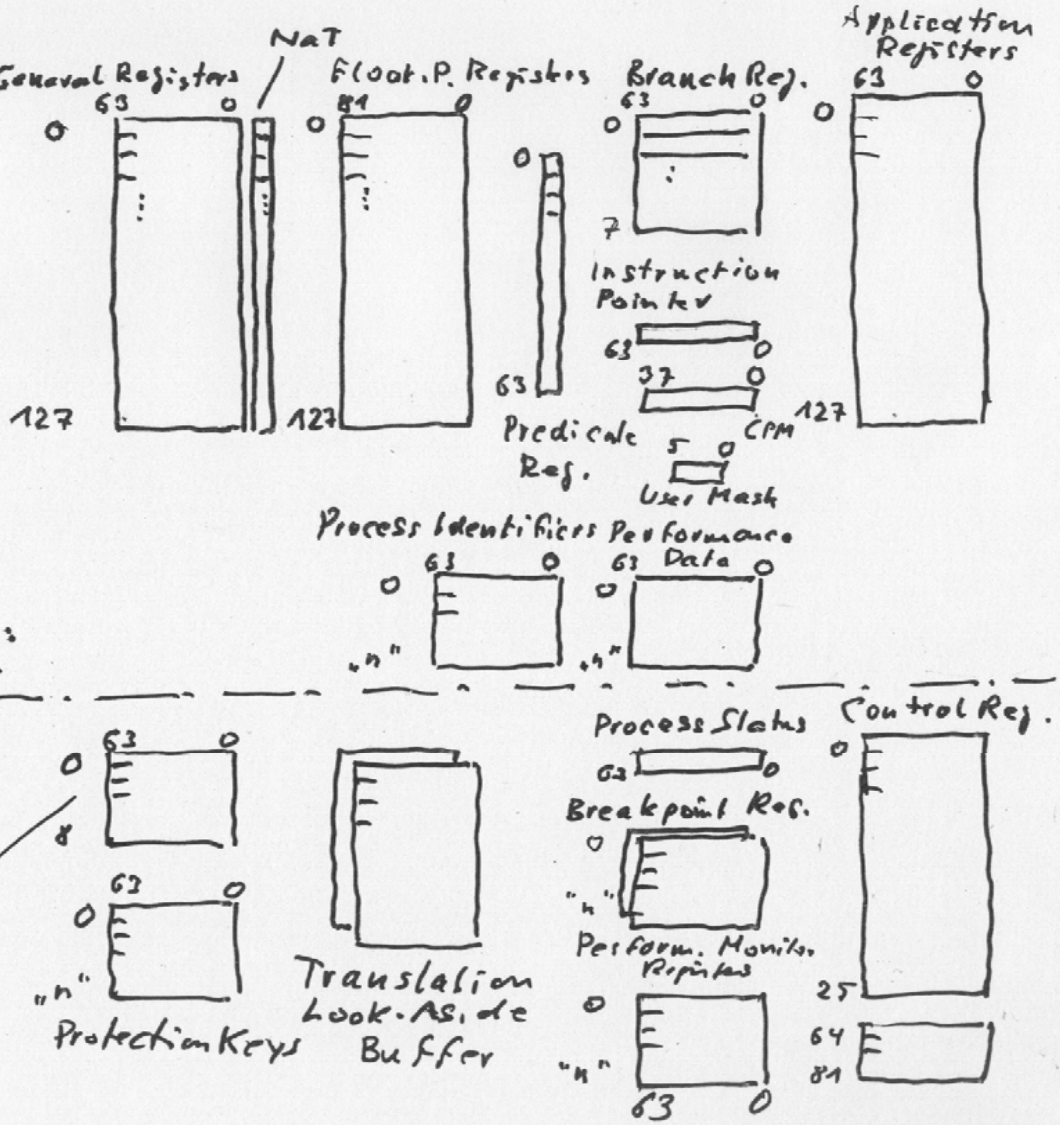
Itanium: Register (1)

Die wichtigsten Register sind

- 128 allgemeine Register (general purpose registers) zu 64b
- 128 Gleitpunktregister (floating point registers) zu 80b
- 128 Anwendungsregister (application registers) zu 64b, für den IA-32-Modus und als Schnittstelle zwischen Anwendungs-programm und Betriebssystem, u.a.
- 64 Bedingungsregister (predicate registers)
- 8 Sprungzielregister (branch registers), für die Zieladresse indirekter Sprünge

Daneben Register für Betriebssystemfunktionen.

REGISTER DES ITANIUM



Itanium: Register (2)

Verwendung der allgemeinen Register (GR):

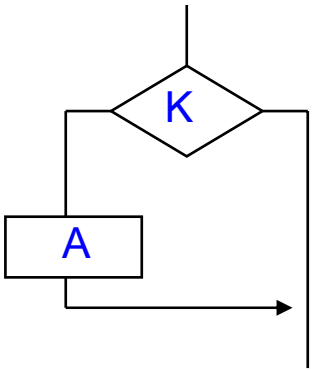
GR 0 .. GR 31 („static“) werden wie gewöhnlich genutzt, u.a. aber auch für häufig gebrauchte Werte des main program. Wenn eine Prozedur diese Register benutzt, muss sie den Inhalt i.a. zunächst retten und später wiederherstellen.

GR 32 .. GR 127 („dynamic“) enthalten den jüngeren Teil des Laufzeitstapels (run-time stack), mit den Eingabe/Ausgabe-Parametern der Prozedur und (während der Aktivierung der Prozedur) ihren lokalen Variablen, d.h. den Aufrufsatz der Prozedur (activation record). Diese Register werden dynamisch, d.h. bei Betreten und Verlassen der Prozedur, vergeben. Erforderlichenfalls Überlauf in den Hauptspeicher.

Itanium: Bedingungsregister (1)

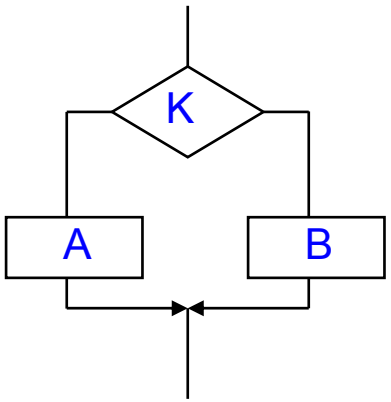
Die Bedingungsregister (predicate registers) entscheiden, ob das Ergebnis eines ausgeführten Befehls für gültig erklärt oder verworfen wird (Entscheidung in der retirement phase). Dabei dienen die Bedingungsregister 0 .. 15 der Steuerung von Operationen, die bedingten Sprüngen folgen, und 16 .. 63 dem „Software Pipelining“ von gewissen Schleifen.

Itanium: Bedingungsregister (2)



Bedingter Sprung mit leerer Alternative:

Der Compiler bedingt die Ausführung von A über ein Bedingungsregister P. Die in A enthaltenen Anweisungen können ausgeführt werden, noch bevor K berechnet ist, werden aber erst, wenn K ihr Prädikat P auf „wahr“ setzt, gültig (committed).



Bedingter Sprung mit nicht leerer Alternative:

A und B können ausgeführt werden, noch bevor K berechnet ist. Sie sind durch komplementäre Prädikate P_A und P_B bedingt und erst, wenn K berechnet ist, wird P_A oder P_B auf „wahr“ gesetzt.

Itanium: Bedingungsregister (3)

Abwicklung von Schleifen (Loop Unrolling) und Software Pipelining

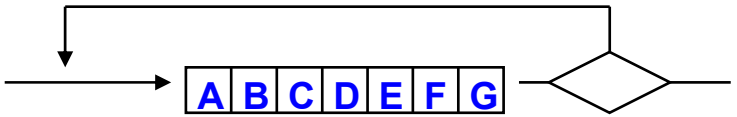
Wenn eine Schleife so gesteuert ist, dass schon der Compiler die Zahl der Durchläufe sehen kann (z.B. Zählschleife mit festen Grenzen), dann kann die Abwicklung (Unrolling) günstig sein: Vermeidung der Sprungbefehle, aber dafür u.U. wesentlich längerer Programmcode. Wenn zudem zwischen den Ausführungen des Schleifenrumpfes keine kritischen Datenabhängigkeiten bestehen, dann kann die Ausführung parallel erfolgen, z.B. in einem Pipeline-Schema (nächster Rumpf eine Operation später): Software-Pipelining.

Itanium: Software-Pipelining

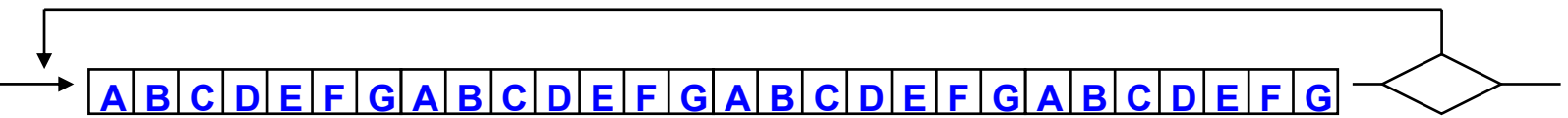
Software-Pipelining parallelisiert die Ausführung aufeinander-folgender Schleifendurchläufe. Sie erfordert, wenn die Operationen A, B .. komplex sind, eigene Prozessoren, sonst auf CPU-Pipeline durchführbar. Da die Register im Schleifenrumpf vom Compiler fest gewählt werden, muss der Prozessor ein Renaming durchführen: Register Rotation modulo n (n: Zahl der parallelen Schleifen-rümpfe).

Loop Unrolling und Software Pipelining

Loop:

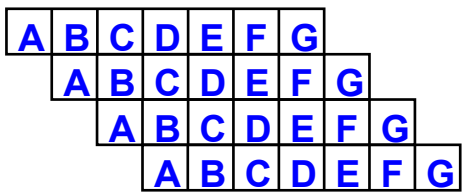


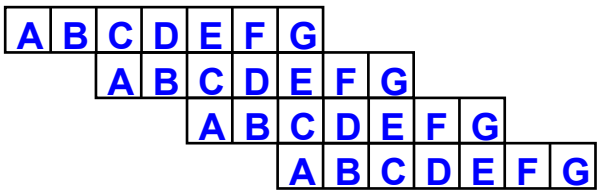
Loop Unrolling:



Ist Zahl de Durchläufe durch 4 teilbar?

Software Pipelining:

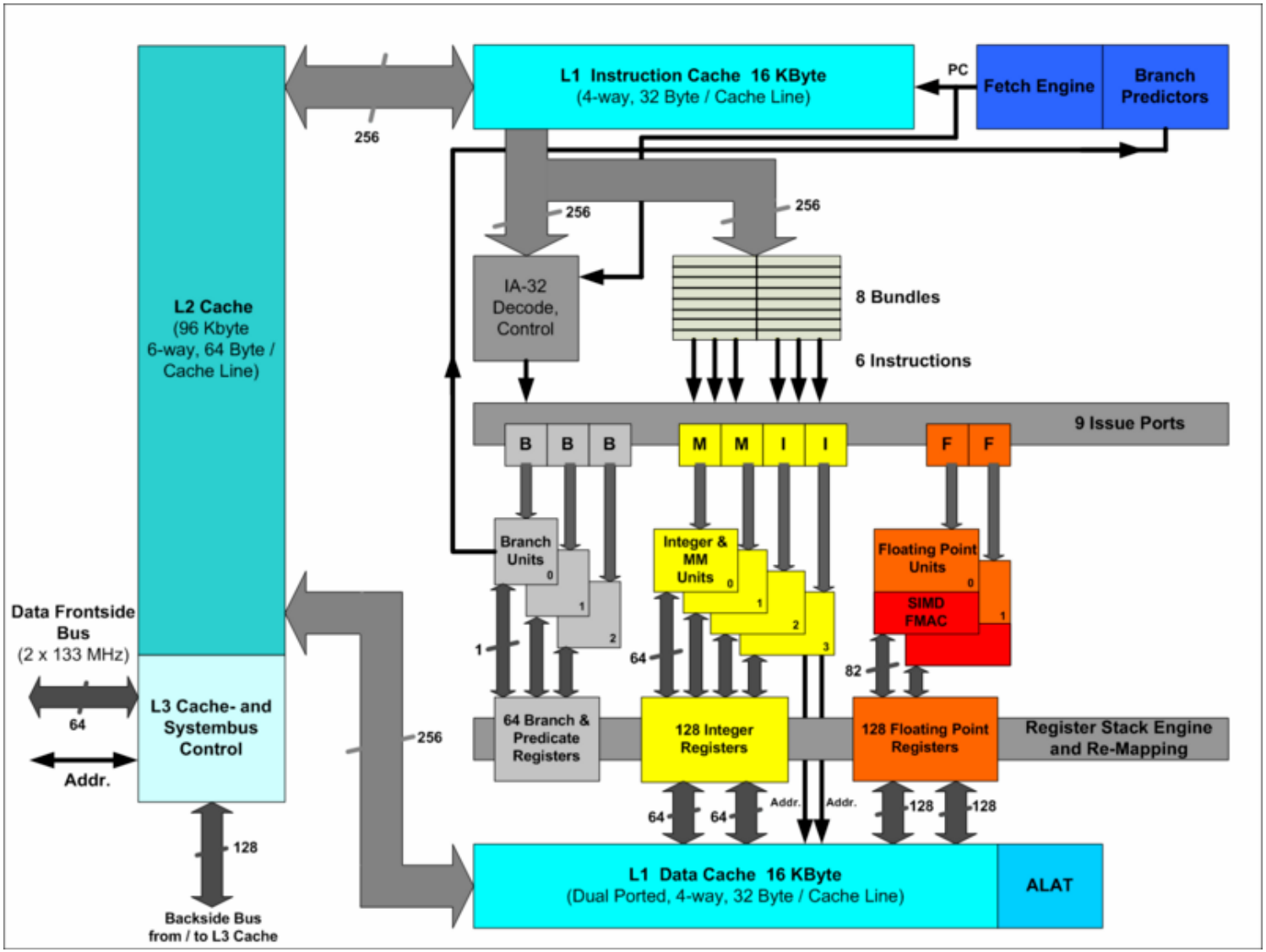




Ausführung auf CPU-Pipeline oder Multi-prozessor. Rechte Version erleichtert Ergebnisüber-gabe zwischen Schleifen

Itanium unterstützt Software Pipelining durch dynamische Bereitstellung von Registern („Register Rotation“) und Predication.

Itanium:


L1 Instruction Cache 16 KByte
(4-way, 32 Byte / Cache Line)
PC
Fetch Engine
Branch Predictors
256
256
256
L2 Cache
(96 Kbyte
6-way, 64 Byte /
Cache Line)
IA-32
Decode,
Control
8 Bundles
6 Instructions
9 Issue Ports
B
B
B
M
M
I
I
F
F
Branch Units
Integer & MM Units
Floating Point Units
SIMD FMAC
Data Frontside Bus
(2 x 133 MHz)
64
Addr.
L3 Cache- and Systembus Control
1
64
82
256
64 Branch & Predicate Registers
128 Integer Registers
128 Floating Point Registers
Register Stack Engine and Re-Mapping
64
64
Addr.
Addr.
128
128
128
Backside Bus
from / to L3 Cache
L1 Data Cache 16 KByte
(Dual Ported, 4-way, 32 Byte / Cache Line)
ALAT

2.2. Grundstrukturen

2.2.1 Elementarfunktionen und Schichtung

Wir kennen längst: Speicher, Zellen, Adressen, Befehlszähler (?)/-adressregister, Sprünge, Unterbrechungen, Problem der Speicherschnittstelle, Register, Zahldarstellungen, …

Und viele Einzelheiten von Pentium, SPARC, Itanium.

Und die Schichtung (ISA, Mikroarchitektur), die allerdings bei RISC (also auch Itanium) weniger überzeugend ist.

2.2.2 Status des Prozessors

Grundsätzliche Frage bei Funktionseinheiten:

Zustandbehaftet (stateful) oder zustandsfrei (stateless)?

Seit langer Zeit die Tendenz, dem Prozessor immer mehr Zustand zu geben:

Cache(s), Register aller Arten, Mikrozustand (Belegung welcher Einheiten mit welcher Phase der Ausführung welchen Befehls?)

Zustandsbehaftete Funktionseinheiten bieten i.a.

- mehr Effizienz durch kurze Zugriffszeiten auf die benötigten Daten
- weniger Außenverkehr und mehr autonome Funktionalität, dank ihres Gedächtnisses

Zustandsfreie Funktionseinheiten sind darauf angewiesen, dass der Status des Prozesses in einem Speicher außerhalb von ihnen Liegt oder mit der Beauftragung übermittelt wird:
Also z.B. Prozessor weiß lediglich, wo „seine“ Zustandsgrößen im Speicher liegen (z.B. unter welcher konstanten Speicheradresse er den Befehlszählerstand findet, usf.)

Der Charme zustandsfreier Funktionseinheiten ist

- bei Ausfall ist nichts verloren, einfache Recovery
- unverzüglicher Kontextwechsel ohne vorherige Rettung des Zustandes

Aber: auch nur zustandsarme Prozessoren sind längst aus dem Felde geschlagen,

- Kontextwechsel dauern zwar viele Takte, aber die Takte sind so kurz, dass die externen Anforderungen (Realzeitbetrieb) leicht gehalten werden.
- Effizienznachteil der zustandsfreien Maschine, die einen hochgradig zustandsbehafteten Prozess ausführt, ist inakzeptabel.
- Prozessorhardware wird immer billiger, so dass zeitkritische Kontextwechsel (Thread-Umschaltungen) durch mehrfachen Zustand im Prozessor verkürzt werden können.

2.2.3 Privilegierung und Modi

Nur ein Benutzer, dem das Rechensystem (das ist Rechner und Grundsoftware) und alle enthaltenen Daten vollständig und allein gehören, darf auch alles anstellen, was programmierbar ist!

Aber bereits wenn die Grundsoftware nach ihm anderen dienen soll, umso mehr falls etwa

- Daten und Programme anderer Benutzer im System sind (z.B. bei Mehrprogrammbetrieb) oder
- fremde externe Prozesse vom Rechner gesteuert werden können,

muss man den Benutzer einschränken!

Notwendige Einschränkungen des Benutzerprogrammlaufs:

- gewisse Unterbrechungswünsche sind vom Benutzer nicht unterdrückbar und führen zur Behandlung ins Betriebssystem; damit wird Prozessorentzug möglich, bei Zeitüberschreitung oder Verstößen (z.B. Zugriff auf nicht zugeteilte Betriebsmittel).

  Auf den meisten Prozessoren kann der Benutzerprogrammlauf nur wenige Unterbrechungswünsche unterdrücken, z.B. arithmetische Alarme, Messpunkte, Testpunkte, die er selbst benutzt

- der Benutzerprogrammlauf kann nicht alle Befehle benutzen, bzw. die Befehle haben eingeschränkte Wirkung. Dadurch bleiben kritische Teile des Rechners unzugänglich (Ein/Ausgabe, Uhr, Wecker (timer), Speicherbereiche), -> Modi

(Einschränkung des Benutzerprogrammlaufs)

- Die Maschine prüft die aktuellen Rechte des Benutzerprogrammlaufs bei Ausführung seiner Befehle. Die Rechte sind vom Benutzer nicht änderbar.

Beispiel Pentium: Raffinierte Adressraumstrukturierung für jedes Benutzerprogramm in „Segmenten“. – Mehrstufige Schutzfunktionen – 32 Systembefehle

Modi

Prozessoren benutzen wenigstens 2 Interpretationsmodi für Programme

- Benutzerprogramm: Benutzermodus (user mode)
- Betriebssystem: Systemmodus (system / executive mode)

Das Benutzerprogramm wird nur im Benutzermodus interpretiert. Es kann Dienstleistungen des Betriebssystems anfordern, die im Systemmodus ausgeführt werden; Rückkehr mit Kontrolle, dass keine privilegierte Information übertragen wird.

Das Betriebssystem wird

- Systemkern: im Systemmodus
- Systemprozesse (verzögerbare Dienste): im Benutzermodus oder in anderen Modi ausgeführt.

(Modi)

Beispiel Pentium: 4 Modi („Privilege Levels“)

0: Betriebssystemkern

1: Betriebssystemprozesse

2: Global benutzte Prozesse

3: Benutzerprogrammläufe

Daneben gibt es die IA-16, IA-32, Intel 64 Modi!

Wunsch nach mehr als 2 Modi rührt her von Strukturierungs- und Schutzbedürfnissen bei sehr großen Betriebssystemen.
Gelegentlich Fortsetzung in die Benutzerprozesse hinein: Bei Gruppen kooperierender Benutzerprozesse Einführung von Privilegierungsniveaus für Zugriffe auf Daten und Programme.
Dann 16 .. 64 Niveaus.

Auswirkung des aktuellen Maschinenmodus auf die Befehls-ausführung:

Ausführung kann

- möglich sein, unabhängig vom Modus in gleicher Weise (z.B. Arithmetik, Transporte in allgemeine Register)
- möglich, aber modusabhängig verschieden sein (z.B. Adressierung)
- unmöglich sein; dann wird ein Unterbrechungswunsch erzeugt, über den der Prozessor die Bearbeitung des Betriebssystems an definierter Stelle aufnimmt, womit
  - der den Befehl benutzende Prozess unterbrochen wird und evtl. das Betriebssystem die mit dem Befehl gewünschte Wirkung sinngemäß herstellt, z.B. „stop“: Ende des Prozesses (nicht als Maschinenbetrieb), „E/A“: Ein/Ausgabe über das Betriebssystem (statt direkter Betätigung des Geräts)

Bedingungen für „Virtuelle Maschinen“

Ein Basisbetriebssystem (VM Monitor, Hypervisor o.ä.) führt im Zeitmultiplex mehrere (auch verschiedene) Betriebssysteme aus, auf denen jeweils (u.U. viele) Anwenderprogramme laufen. Dabei:

- Basisbetriebssystem: im Systemmodus
- Betriebssysteme: im Benutzermodus (!)
- Anwenderprozesse: im Benutzermodus

Voraussetzungen an den Prozessor:

- Prozessor ist entziehbar (Wecker, Verstöße, nicht unterdrückbar)
- Virtuelle Adressierung (zur privilegierten Hauptspeicherverwaltung, Schutzmechanismen)
- Nicht privilegierte Operationen in beiden Modi gleich wirkend
- Alle Befehle mit privilegierter Wirkung (Organisation des Virtuellen Speichers, E/A, Timer, Unterbrechungsmasken, Systemregister) sind privilegiert und nur im Systemmodus ausführbar, im Benutzermodus lösen sie einen Systemaufruf aus.

Übergang in den privilegierten Modus

- Durch gewisse (oder alle) Unterbrechungen; solche können sich vor allem auch aus Verstößen ergeben (Befehle, Adressen); kann sehr sinnvolle Konstruktion sein! Geht mit der Rettung des Prozessstatus einher (Registerinhalte).
- Durch besondere Befehle („Supervisor Call“).

  Beispiel: Itanium: „Enter Privileged Code“, schlanker als Unterbrechungsbehandlung: kein Abspeichern des Status. – Übergang in den jeweils höher privilegierten Modus.

  Beispiel Pentium: Systemaufruf durch Call in den Betriebssystemadressraum, wird über „Call Gate“ geleitet, führt in Unterbrechung.

Übergang in den weniger privilegierten Modus

Beispiel: Itanium: „Return from Interrupt“, Pentium: „Return“ (über Call Gate), „Return from Interrupt“

2.2.4 Datentypen

Von großem Einfluss auf Struktur und Aufwand des Prozessors ist die Zahl und Art der Datentypen und die Zahl und Art der Operationen, die der Prozessor auf ihnen ausführen kann.

Müssen Datentypen und Operationen durch den Prozessor fixiert sein?

> Burroughs B 1700 (1971): Bitadressierbarer Speicher, beliebige Datenformate, Interpretation erfolgt durch (evtl. vom Benutzer selbst verfasste) Interpreter, i.a. je an einer Familie höherer Sprachen orientiert.

Aus Effizienzgründen und quasi-Standardisierung der 8/16/32/64-Bits-Formate unüblich.

Statt Einzelbit-Adressierung prozessorfixierte Datentypen mit zugehörigen Operationen

Bits: (selten!)
Bitfeld: Einzelbitoperation auf z.B. 16/32 Bits, ^, v, =, ≠ wichtig, allgemein üblich
Bitstring: Direkt mit Anfangsposition/Länge oder indirekt über „Deskriptor“ adressiert: Gelegentlich, z.B. VAX
Dezimalzahl (Festpunkt): Gelegentlich
Bytes: Typisch 8 Bits (EBCDIC extended binary coded decimals interchange code) oder ASCII (American standard code for information interchange, 7 Bits). Weitverbreitet
Bytestring: Wie Bitstring, aber weit verbreitet; manchmal auch Darstellungen mit Endmarke. IBM, ca. 1955.
Dezimalzifferstring: (packed decimals) verbreitet als zwei 4 Bits-Ziffern je Byte. Zugang wie Bitstring, selten über Schlusszeichen (Vorzeichen).

Festpunktzahlen (dual): Auf allen Rechnern, oft mehrere Formate, z.B. 8/16/32/64 Bits. Operationen an Langworten oft als Unterprogramm, aber maschinenunterstützt (carry bit als Unterbrechungswunsch).

Gleitpunktzahlen (dual): Auf vielen Rechnern, bei Mikroprozessoren früher über Coprozessor. Gelegentlich mehrere Formate. Erstmals Zuse 1936.

Komplexe Zahlen: Selten

Felder: Auf Vektorrechnern, sonst selten. Aber Hilfsoperationen (Indexmodifikation, Indexprüfung).

Listen, File, Queue: Selten

Stack: Als Hardwarestruktur selten (English Electric KDF 9, 1958; dann Burroughs-Maschinen). Für Ausdrucksberechnung und Laufzeit-Stack.

Häufig: Stützung durch Adressierung im Hauptspeicher (autoincrement/autodecrement-Technik). Lade- bzw. Speicher-operationen holen einen Wert aus dem Stack (mit Heraufsetzen der Zugriffsadresse) bzw. speichern einen Wert im Stack (mit Herab-setzen der Zugriffsadresse). Unter der Konvention, dass der Stack nach unten wächst, entsprechen die Operationen „POP“ bzw. „PUSH“. Die Modifikation der Adresse entspricht gerade dem Platzbedarf des Wertes.

Weitere Datentypen im Prozessor als Folge des von Neumann-Prinzips, mit ebenfalls erheblichem Einfluss auf Prozessorleistung und -komplexität:

(Befehle, vgl. Abschnitt 2.3)

Adressen: Heute meist mit einem Festpunktformat übereinstimmend, mit Festpunktoperationen manipulierbar; vgl. die sorgfältige Trennung von (typgebundenen!) Operationen in höheren Sprachen.

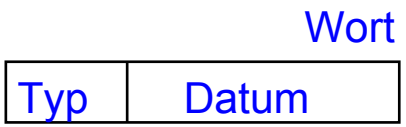
Bei 64b-Prozessoren oft nur Teilwort (46, 48b) für Adresse.

Selten, aber konzeptuell interessant:

Deskriptoren: Größen, über die (oft obligatorisch) komplexe Datenstrukturen (und oft: Programme) zugegriffen werden. Enthalten Adresse, Typ, Zugriffsrechte, Strukturinformation, evtl. Zugang zu Operationen auf der Datenstruktur.

Zugriffsausweise (capabilities): Schlüssel, die auf (wenigen) Maschinen zugreifende Prozesse benötigen, um auf Daten/Programme zuzugreifen. Vgl. Zertifikate in verteilten Systemen!

Üblicherweise ist Datentyp nicht explizit notiert, sondern durch Interpretation implizit definiert.
Anders: tagged architectures, Typenkennungsmaschinen



Typ dient:

- Zur Operationsanpassung (generische Operationen), z.B. ein Additionsbefehl auf alle Gleit/Festpunkt/Byte/Dezimal-Formate anwendbar; Algorithmuswahl und Konversion automatisch

oder

- Zur Erkennung fehlerhafter Zugriffe:
  -- Addition eines Befehls
  -- Änderung eines Zugriffsausweises oder eines Deskriptors
  -- Ausführung eines Bitfeldes

Rudimentär im SPARC Processor

2.2.5 Abbildung der Wortformate auf Byte- bzw. Speicherzellen-Raster

Jonathan Swift: Gullivers Reisen (1726):

Gulliver gelangt in das Land Liliput, dessen Einwohner nur 6 Zoll groß sind. Dort war es verboten, ein Ei an der stumpfen Seite zu öffnen, weil sich der Prinz des Landes dabei in den Finger geschnitten hatte. Ein Teil der Bevölkerung, die BIG ENDIANS, wollte das nicht hinnehmen, es gab 6 Rebellionen, ein Kaiser starb, einer musste gehen, die BIG ENDIANS wurden aus dem öffentlichen Dienst ausgeschlossen, 100 BIG ENDIAN-Bücher verboten. Viele BIG ENDIANS retteten sich in das benachbarte Blefuscu. Es kam zum Krieg, der 3 Jahre dauerte und Liliput 30000 Soldaten, vierzig große und noch mehr kleine Schiffe kostete. Die Invasion von Liliput konnte nur dadurch verhindert werden, dass Gulliver („der Menschenberg“) die feindliche Flotte raubte.

BIG ENDIANS

$\downarrow 2^{31}$ $\downarrow 2^{0}$

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Motorola, IBM/360 (!)
Itanium (umschaltbar)
SPARC
Blefuscu

LITTLE ENDIANS

$\downarrow 2^{31}$ $\downarrow 2^{0}$

3	2	1	0
7	6	5	4
11	10	9	8
15	14	13	12

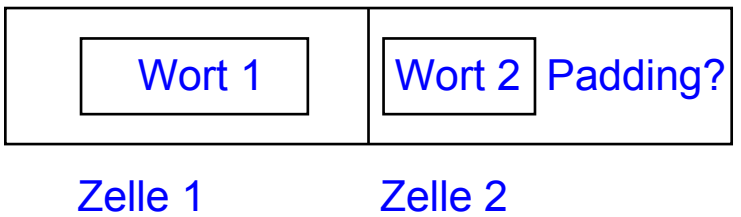
Intel Pentium, IBM PC (!)
Itanium (umschaltbar)
VAX
Liliput

Beispiel: 32 b vorzeichenlose ganze Zahl, abgebildet auf Byte-Adressen-Raster. Big Endians beginnen Byte-Adressen auf der höchstwertigen Position, Little Endians auf der niedrigstwertigen. Bei Datenaustausch zwischen Big und Little Endians

- Bytes wortweise: müssen die Wörter am Ziel zerlegt, die Byteanordnung gespiegelt werden
- Wörter bnyteweise: muss die Bytefolge in Vierergruppen am Ziel gespiegelt werden

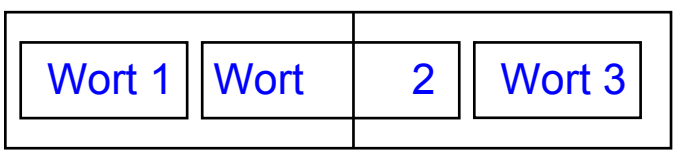
Bytefelder und Zahlen müssen als solche bekannt (evtl. bezeichnet) sein!

Ein anderes Abbildungsproblem ist dadurch gegeben, dass u.U. Dateneinheiten nicht auf Grenzen von Speicherzellen beginnen. Der Prozessor beschafft mit jedem Speicherzugriff den Inhalt von einer oder mehreren (2/4/8 …) Speicherzellen. Die gesuchte Daten-/Befehlseinheit kann auf einer Zellgrenze beginnen,



„aligned“: Vorzug, wenn Speicher billig RISC-normal, notfalls Alarm und programmierte Behandlung

oder nicht:



„mis-aligned“: bedeutet Mehrfachzugriff für einzelne Daten- und Befehlseinheit, dann Extraktion. Pentium (überwiegend)

Mis-Alignment bedeutet wesentliche Verlängerung des Zugriffs, aber erleichtert Codeerzeugung, Portabilität, Hauptspeicherausnutzung

2.3 Befehle

2.3.1 Formate

Klassische Lösung (von Neumann 1946, alle RISCs)

Einheitliches Format:
Zwar einfaches Leitwerk, Befehlszähler generiert nächste Adresse, aber redundant im Speicher / Cache und auf dem Bus

- weil Befehle sehr verschieden häufig
- weil Befehle nach Art verschieden viele und verschiedenformatige Adressen und Werte enthalten müssen.

Daher lange Zeit (Burroughs 1963) auch verschiedene Formate benutzt: Meist beschreiben die ersten Bits, welches Format vorliegt.

Befehlszähler	→	Befehlsadressregister
Instruction counter	→	program pointer reg. instruction address reg.

Also im Vergleich:

	konstantes Format	variables Format
Platzbedarf in Speicher und Cache (Einfluss auf Trefferratio)	- (ca. 140%)	+
Transportbedarf	-	+
Alignment	+	-
Decodierung	+	-
Folgebefehlsadresse	+	-
Codeerzeugung	+	-
Namhafte Vertreter	SPARC Itanium (fast)	IBM/360.. Pentium

2.3.2 Register und Befehlssatz

Der Befehl muss Ort von Operanden und Ergebnissen bzw. Sprungziel angeben.

Bevorzugte Orte in der Maschine sind Register. Vorteile:

- Zugriffszeit, z.B. 1 Takt gegen 5 Takte (Cache 1) bzw. 500 Takte (Hauptspeicher)
- Größte Platzersparnis im Befehl: z.B. Multiplikation benutzt stets das Multiplikandenregister, das Multiplikatorregister und den Akkumulator der Maschine. Nach der Operation steht das doppelt lange Resultat im Doppelregister Akkumulator/Multiplikatorregister. Aber damit müssen wir die Register erst räumen und neu laden vor der nächsten Multiplikation! Intel-Pentium-Verfahren!
- Große Platzersparnis im Befehl: Satz von allgemeinen Registern, z.B. 16 oder 256, adressierbar mit 4 bzw. 8 Bits. (Ferranti Pegasus, 1957; IBM/360, 1965; Standard bei RISCs). Notwendigkeit des Rettens/ Wiederherstellen bei Kontextwechsel (Multiprogramming) oder von mehrfachen Registersätzen (Multithreading).

Darüber hinaus 2 Ansätze, um neben der besseren Zugriffszeit auch noch Platzersparnis zu erzielen:

- Implizierung (Orte ergeben sich unausgesprochen aus Befehl)
- Kurzadressen.

Lösungen mit teils Registeradressen, teils Speicheradressen sind wichtig

- immer bei load/store-Befehlen
- ein Operand oder das Ergebnis steht im Speicher (sehr häufig bei IBM/360 ff. und Intel x86). Wegen der schwankenden und u.U. extrem langen Zugriffszeiten im System Cache/Hauptspeicher für Pipelining nicht geeignet! Oder vorgezogenes Laden; zu den Problemen siehe Itanium.

2.3.3 Drei/Zwei/Eins/Null-Adress-Prinzip

Eigentlich müsste man einen Satz von Musterprogrammen (z.B. Benchmark) definieren und dann berechnen, wie bei gegebenen Kosten die durchsatzstärkste (ganzer Satz ist fertig gerechnet) Prozessorfunktionalität und Befehlsliste aussieht. Aber das kann die Informatik bisher nicht.

Also Stand der Technik:

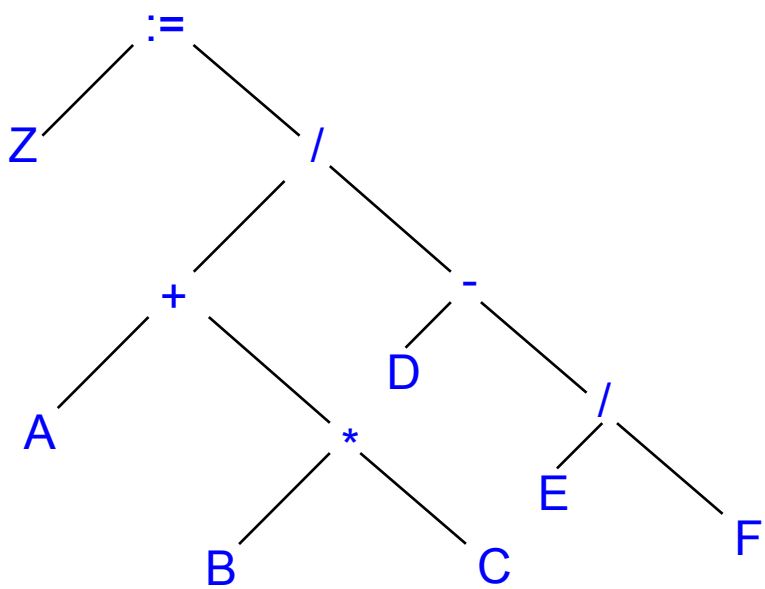
- Respektiere Kompatibilitätszwänge!
- Mache für alternative Konzepte Probecodierungen und simuliere! (Platzbedarf, Laufzeit, Zahl der Speicherzugriffe)

Wir nehmen einen einfachen Fall vor:

Bewertung von Drei/Zwei/Eins/Null-Adress-Prinzip an einem arithmetischen Ausdruck.

Unser Beispiel:

Z := (A+B*C)/(D-E/F)
Von uns und dem Compiler
lieber als Baum gesehen:



Für die Befehle des Programms
erfinden wir eine symbolische
Schreibung:
MULT >B< >C< >G<

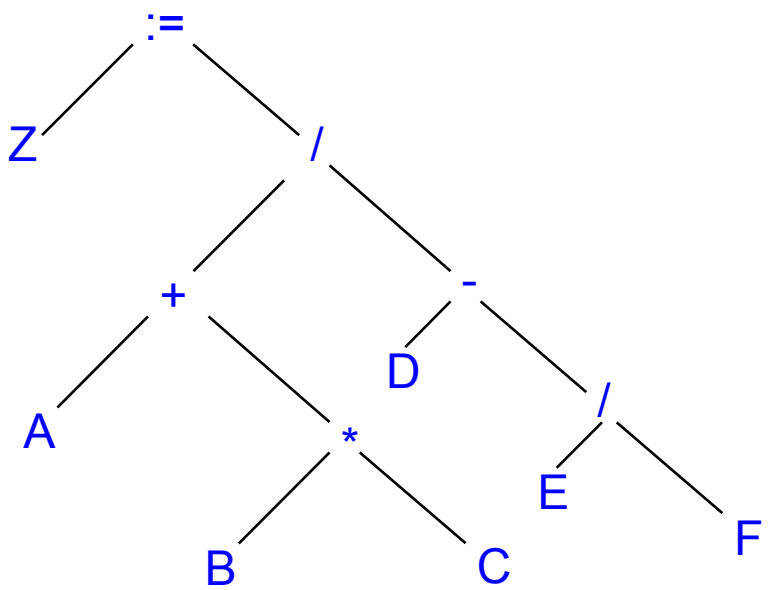
>B< ist „Adresse von B“

Drei-Adress-Prinzip

Das folgende Programm tut‘s:

MULT	>B< >C< >G<
ADD	>A< >G< >G<
DIV	>E< >F< >H<
SUB	>D< >H< >H<
DIV	>G< >H< >Z<

G, H sind Hilfszellen für unsere Zwischenwerte. Noch einfache Code-Erzeugung für Compiler. Platzbedarf 5 Opcodes (MULT, ADD, DIV, SUB, ..) 15 Adressen, Operationen: 5



```
MULT    >B< >C< >G<
ADD     >A< >G< >G<
DIV     >E< >F< >H>
SUB     >D< >H> >H>
DIV     >g> >H< >Z<
```

Z := (A+B*C) / (D-E/F)

Verallgemeinert:
Ausdruck mit v Variablen, o=v-1 Operatoren, Adressformat a Bits, Opcode-Format c Bits

Platzbedarf (3 ADRM): o*(3a+c)
Befehlszahl: o
Hauptspeicherzugriffe: 4*o
Zwischenergebniszellen in von Baumstruktur abhängiger Anzahl (Optimierungssache, Compiler!). Übereinstimmende Unterbäume einfach Behandelbar. Zuweisung braucht keinen Befehl.

Drei-Adress-Prinzip

- Mit vollen Speicheradressen unerträglich platzaufwendig; kann verbessert werden durch Basisadresse und Versatz (displacement), → 2.4
- Mit Hauptspeicherzugriffen sehr anspruchsvoll bezüglich Hauptspeicherdurchsatz; lange Befehlsausführungszeiten, Überlappung notwendig
- Hat bei nur Hauptspeicherzugriffen den Vorteil, dass nach jedem Befehl der Prozessor operandenfrei: Kleiner Status, unterbrechungsgünstig.

Aber günstig sind Drei-Adress-Befehle bei Registermaschinen (und das ist ja der heute übliche Fall!).

Übergang zu Befehlsformaten mit kleinerer Anzahl von Adressen je Operation durch zwei Prinzipien:

- Überdeckung: Ein Operand wird durch Ergebnis überschrieben (Achtung! Verlust, oder erst kopieren)

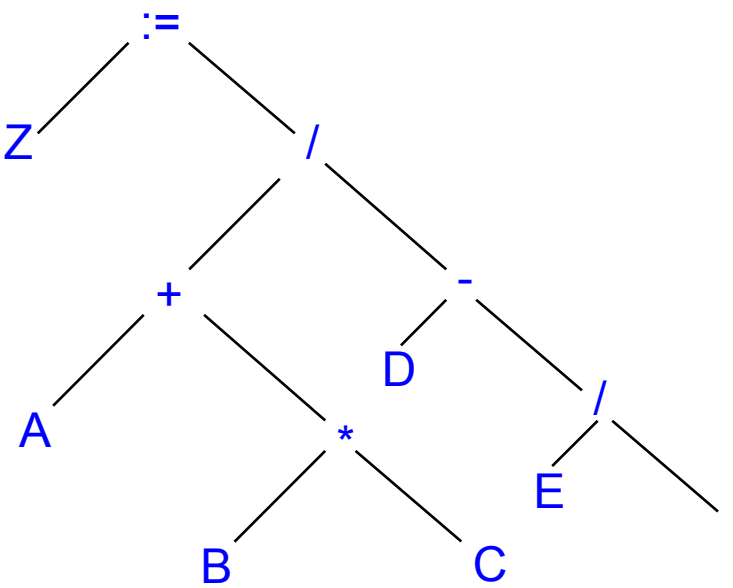
- Implizierung, eine Größe ist stets an einem stillschweigend bekannten Ort (z.B. Akkumulator“-Register. Stack-Spitze).

Beide können kombiniert werden.

Zwei-Adress-Prinzip

Ausnutzung der Überdeckung

MULT	>B< >C<	B:=B*C
ADD	>B< >A<	B:=B+A
DIV	>E< >F<	E:=E/F
SUB	>D< >E<	D:=D-E
DIV	>B< >D<	B:=B/D
Transp	>Z< >B>	Z:=B



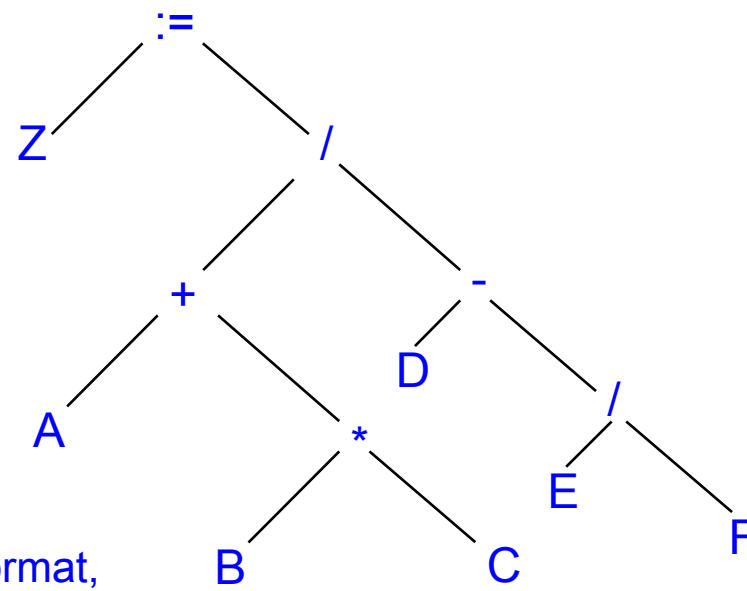
Platzbedarf: 6 Operanden, 12 Adressen
Operationen: 6

Wir haben die Argumente überschrieben!
Zuweisung braucht nun Extra-Befehl!

Zwei-Adress-Prinzip

Ausnutzung der Überdeckung

MULT	>B< >C<	B:=B*C
ADD	>B< >A<	B:=B+A
DIV	>E< >F<	E:=E/F
SUB	>D< >E<	D:=D-E
DIV	>B< >D<	B:=B/D
Transp	>Z< >B>	Z:=B



Verallgemeinert:
o Zahl der Operationen des Ausdrucks, a Adressformat,
c Opcode-Format
Platzbedarf (2ADRM):
o*(2a+c) + 1*(2a+c) + [Laden, Speichern], wegen 1)
Befehlszahl: o+1, Hauptspeicherzugriffe: 3 (o+1)
Zuweisung braucht Befehl
1) Soll Minuend oder Subtrahend überschrieben werden? Zwei Befehle?

Nun wieder: Zweiadress-Registermaschine

Registermaschine mit Zweiadress-Befehlen:

Wichtig in Mischform mit Dreiadressregistertyp.

„Anderthalbadressmaschine“: Eine Registeradresse, eine Hauptspeicheradresse. Typisch 1 Operand aus Hauptspeicher.

(Minuend, Subtrahend? Also zwei x zwei Subtraktionsbefehle oder erst Transportieren?)

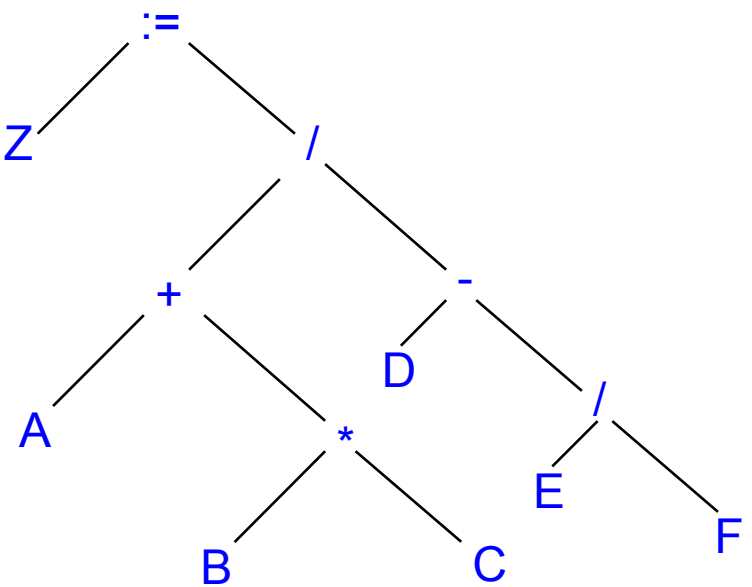
Ergebnis wird mit Speichere-Befehl in den Hauptspeicher gebracht.

Häufig in IBM/360 ff und Pentium gebraucht.

Ein-Adress-Prinzip (von Neumann):

Ausnutzung von Überdeckung und Implizierung:
„Akkumulator“ ist stets beteiligtes Register (AC)

HOLE	>B>	AC:=B
MULT	>C<	AC:=AC*C
ADD	>A<	AC:=AC+A
Speich	>G<	G:=AC
HOLE	>E<	AC:=E
DIV	>F<	AC:=AC/F
Speich	>H<	H:=AC
HOLE	>D<	AC:=D
SUB	>H<	AC:=AC-H
Speich	>H<	H:=AC
HOLE	>G<	AC:=G
DIV	>H<	AC:=AC/H
Speich	>Z<	Z:=AC



Platzbedarf: 13 Operations Codes, 13 Adressen, Operationen: 13
Wenn man aber zunächst den Divisor berechnet und noch einen Befehl

SUB2 >X< AC:=X-AC

erfindet, kommt man auf 9 (!) Operationen hinunter (falls der Compiler das schafft).

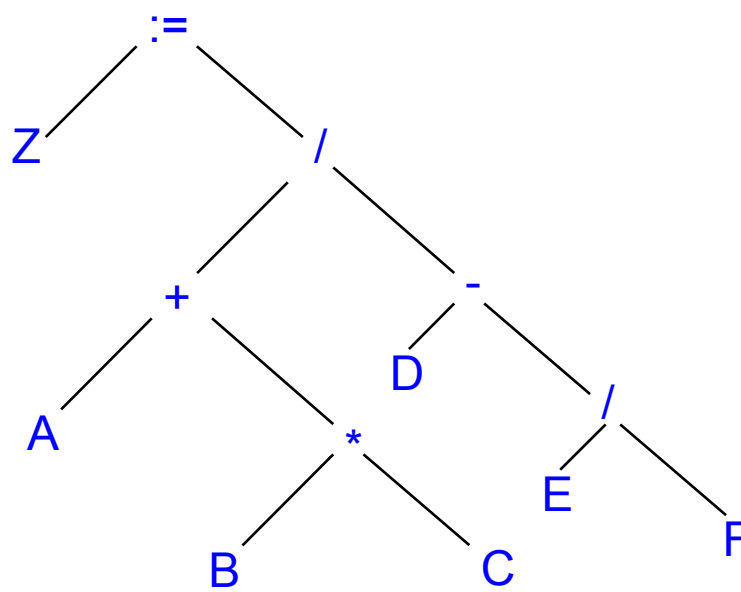
Ein-Adress-Prinzip:

Verallgemeinert:

o Zahl der Operanden des Ausdrucks
Adressformat a Bits, Opcode-Format c Bits
Platzbedarf (1ADRM):
o*(a+c) + 2(a+c) + [Laden, Speichern]
↑
Ergebnis, 1. Laden

Anteil Laden und Speichern hängt von Baumstruktur und Compiler ab!
Befehlszahl: o+2+[Laden, Speichern]
Hauptspeicherzugriffe: 2(o+1+[Laden, Speichern])
Befehlsvarianten wie AC:=D-AC, Zusatzbefehle wie AC:=-AC, AC:=1/AC sind sehr nützlich.
Nutzung übereinstimmender Unterbäume braucht besondere Speicher- und Ladeoperation.
Im Wesentlichen werden keine Argumente überschrieben!

Null-Adress-Prinzip:
(English Electric KDF9, 1959; Burroughs 1962)



Arithmetischer Ausdruck wird in Postfixnotation gebracht:
Z:=(A+B*C) / (D-E/F) >Z< ABC*+DEF/-/:=
Und das ist die Maschinensprache: Also 7 Adressen, 6 Operationscodes.
Die Maschine arbeitet mit einem Operanden-Keller. Abarbeitung von links: Jeder Operand wird gekellert, ebenso die Adresse von Z, jede Operation wirkt auf die beiden oberen Kellereinträge, Ergebnis ersetzt diese. Situationen:

1	2	3	4	5	6	7	8
C			F				
B	B*C		E	E/F			
A	A	Zähler	D	D	Nenner		
>Z<	>Z<	>Z<	Zähler	Zähler	Zähler	Quotient	
			>Z<	>Z<	>Z<	>Z<	(Stack leer)

Zeit →

Null-Adress-Prinzip:

- Überdeckung (Operand auf Stack wird überschrieben) und Implizierung (beide Operanden und Ergebnis auf Stack) → Operation ohne Angabe von Adressen.
- Alle Variablen des Ausdrucks sind auf den Stack zu laden; aber dann kein Laden/Speichern, da Operanden stets korrekt bereitgestellt!
- Abspeicherung am Ende. Zusätzliches Abspeichern (copy, nicht pop!) zur Behandlung übereinstimmender Unterbäume.

Verallgemeinert:

o Zahl der Operanden des Ausdrucks
Adressformat a Bits, Operandenformat c Bits
Platzbedarf (0ADRM): (o+1)*a + o*c + (a+c)

↑ so viele Variable ↑ Abspeichern

= o*(a+c) + 2a+c

Befehlszahl: o+1 (anfängl. Laden!)
Hauptspeicherzugriffe: o+2 (Befehle)

Bewertung der Ergebnisse

Platzbedarf:

Verallgemeinert:

3ADRM:	o*(3a+c)
2ADRM:	o*(2a+c) + (2a+c) + [Laden, Speichern]
1ADRM:	o*(a+c) + 2*(a+c) + [Laden, Speichern]
0ADRM:	o*(a+c) + (2a+c)

[Laden, Speichern] ist Compiler-sensibel.
3ADRM und 0ADRM sind Compiler-günstig.
3ADRM nutzt übereinstimmende Unterbäume ohne Mehraufwand.
3ADRM nur bei unkomplexen Ausdrücken erträglich.
Für komplexe Ausdrücke Null-Adress-Prinzip günstig (diese sind selten!!)
Tatsächlich weitgehende Verschiebung bei Benutzung von Registern bei 3ADRM und 2ADRM! a ist klein, z.B. nur 8 Bits statt z.B. 32 Bits.

Zahlenbeispiel: Unser Beispielausdruck

Platzbedarf (Bits)

	Zahl Operations-codes (8Bits)	Zahl Adressen (8 oder 32 Bits)	Platzbedarf 8 Bits Adressen	Platzbedarf 32 Bits Adressen
3 ADRM	5	15	160	520
2 ADRM	6	12	144	432
1 ADRM	13	13	208	520
Verbessert, vgl. 2.3.3-10	9	9	144	360
0 ADRM	7	7	112	280

Und nochmals Verallgemeinert:

	Befehlszahl	Register- bzw. Hauptspeicherzugriff
3 ADRM	O	4 o
2 ADRM	o+1 + [L, Sp]	3(o+1 + [L, Sp])
1 ADRM	o+2 + [L, Sp]	2(o+2 + [L, Sp])
0 ADRM	o+1	o+2

Bei Registermaschinen (3ADRM, 2ADRM) bleiben aber Hauptspeicherzugriffe für Laden und Abspeichern.

Nachbemerkungen:

- Die Null-Adress-Maschine schneidet nur in komplexen Ausdrücken so gut ab. Der normale Programmierer hasst komplexe Ausdrücke.
- Arithmetik macht nur 5-10% eines Programms aus.
- Bei Maschinen mit einheitlichem Befehlsformat müssen auch andere Operationen im selben Format codierbar sein (also z.B. Transporte, Operationen mit großen Konstanten (evtl. im Register bereithalten)).
- Seit den RISC-Prozessoren 2- und 3-Adress-Registerbefehle typisch.

2.3.4 Befehlstypen

Die Semantik der Befehle wird bestimmt durch

- das gewählte Format (zu kleines Format zwingt zu Vorbefehlen, z.B. compare-branch; zu großes Format führt zu aufgeblähter Befehlswirkung mit Varianten; Vorteil von uneinheitlichen Formaten)
- Semantik der Benutzersprache
- Maschinenstruktur
- Ablauferfordernisse (z.B. Pipelining)

Nach Häufigkeit in Programmen:

- Transporte (ca. 50%)
- Sprünge (ca. 25%)
- Arithmetik (ca. 10%)
- Shifts (ca. 3%)
- „Logische“ Befehle (ca. 3%)
- Rest (ca. 9%)

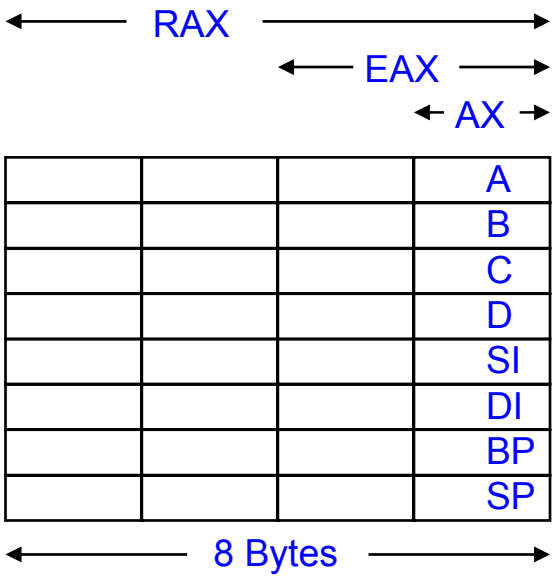
Die Angaben schwanken mit Maschinenarchitektur/Sprache/Algorithmus.

Beispiel: Befehlslisten:

Leitwerkbefehle (betreffen ALU/arith. Pipes nicht)		Pentium	SPARC
Systembefehle (mit S/A), meist privil.		32	18
Transporte		25	50
Sprünge, Aufrufe, programmierte Unterbrechungen		9	10
Vergleiche, Tests, bedingte Sprünge und Transporte		18	8
Schleifen		4	-
Adressierung		5	-
Andere		8	-
Rechenwerkbefehle			
Arithmetik	Festpunkt	18	25
	Gleitpunkt	97	50
Multimedia		ca. 350	16
Flags, Conditions		9	-
Konversion		5	-
„Logik“ (and, or, ..)		4	12
Shift, Rotation		16	3
Strings		7	-
Bits		6	-
		ca. 600	192

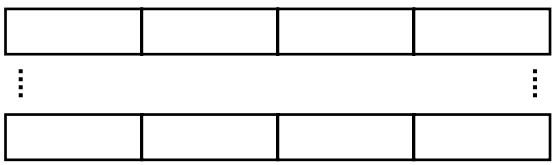
Beispiel: Transporte im Pentium (Core 2, d.h. „Intel 64“)

Register:



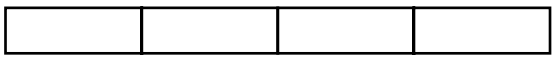
64b Intel 64
32b IA-32
16b IA-16

A Hauptregister
B Basisadresse
C Schleifen
D Multiplikation, Division
SI Source Index
DI Destination Index
BP Base Pointer (Activation Record)
SP Stack Pointer

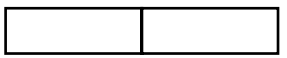


R 8 General Purpose Register (nur diese sind befehlsunspezifisch)

R 15



Instruction Pointer



Flags (Steuerbits, Condition Codes)

Register im Pentium: und dann noch:

CS	Code Segment
SS	Stack Segment
DS	Data Segment
ES	Extra Segment
FS	weiteres Segment
GS	weiteres Segment

Adressen in der Local/Global Segment Descriptor Table, darüber Zugang zum Speicher

und Gleitpunkt (80b)- und Multimedia-Register (64b)

Pentium-Transport-Formate (gilt ähnlich auch für andere Operationen): abhängig vom Ausführungsmodus:

REAL MODE: Der Prozessor simuliert bitgenau einen 8086: 16b

PROTECTED MODE: 4 Privilegierungsstufen. Im Segment-deskriptor (oder im Befehl) ist das Format festgelegt: 16/32/64b

VIRTUAL MODE: Der Prozessor führt mehrere Virtuelle Maschinen aus, mit jeweils eigener Festlegung des Formates.

(Transporte im Pentium)

Quelle und Senke (Register/Speicher) und Wert statt Adresse aus Operationscode ersichtlich. Also (r Register, m Speicherzelle):

MOV (20 Varianten): r→m, m →r, r →r (wichtig! da Register nicht gleichwertig!), Wert →r, Wert →m
Verkürzte Form, arbeitet nur mit A-Register.

MOVS (Stringbefehl) transportiert m →m

XCHG exchange r↔R, m ↔m; letztere Form unter Lock, d.h. die beiden Zellen werden für andere erst wieder zugänglich, wenn beide ihren endgültigen Wert haben (Synchronisationsmittel in Mehrprozessoranlagen).

Außerdem Lade/Speicherbefehle für die 80b-Gleitpunktregister.

Wichtiger Sonderfall von Transporten:

Registerinhalt(e) vom Stack (pop) oder auf den Stack (push). Stack üblicherweise zu kleineren Adressen wachsend

Beispiel Pentium II: Verwendung des Stack Pointers SP/ESP:

POP	Einzelwort/Doppelwort
POPA	Pop 8 Allgemeine Register 16b
POPAD	Pop 8 Allgemeine Register 32b

PUSH, PUSHA, PUSHAD: entsprechend.

Daneben POP und PUSH für die „Gleitpunkt“-Register mit Status-information:

FRSTOR	Restore FPU State
FSAVE	Store FPU State

Diese beiden Operationen benutzen nicht den Stack Pointer.

Vergleiche aber die Lösung mit Autoincrement/decrement!

Noch ein wichtiger Sonderfall:

Schreiben und Lesen der privilegierten Register im Prozessor. Es überwiegen Codierungen der Register im Operationscode (statt Adressierung!).
Mindestens die Schreiboperationen sind privilegiert.

Beispiel Pentium II:
Unter den 32 “System“-Befehlen sind 7 Schreiboperationen und 5 Leseoperationen an privilegierten Registern.

(Leitwerksbefehle)
(Transporte)

Transporte von Zeichenketten

- Bytestrings (auf Byte-adressierten Maschinen zugleich „Worttransport“)
- Dezimalstrings
- Wortblöcke (wichtig für gewisse Speicherverwaltungstechniken, Puffer umkopieren)

Strings beschrieben durch Deskriptor
  Komponententyp
  Länge
  Anfangsadresse
  .
  .

Oder Strukturinformation im zugreifenden Befehl (Adressteil).

(Transporte von Zeichenketten)

Die Länge der Kette kann groß sein.
Folge: Die Befehle müssen während (und nicht erst nach) der Ausführung unterbrechbar sein, d.h. der Intrabefehlsstatus muss rettbar sein:

- Laufende Adresse
- Restlänge

Oft durch Benutzung der allgemeinen Register!

Variante: Ketten mit expliziter Schlussmarke statt Längenangabe.
Schlecht, da nicht vorweg übersehbar, welche Zellen beschrieben.

(Transporte)
Beispiel: Zeichenkettenbefehle in Pentium II:

Die Befehle werden mit Wiederholungspräfix rep realisiert (Ende bei Nulldurchgang im ECX-Register) oder mit anderen Wiederholungspräfixen (z.B. Übereinstimmung mit Wort in Register). Mitwirkende Register:

ESI	laufende Quelladresse
EDI	laufende Zieladresse

MOVS	transportiert String aus Bytes/Wörtern/Doppelwörtnern im Speicher
CMPS	vergleicht zwei Strings
SCAS	vergleicht einen String mit Inhalt von AL/AX/EAX, hält z.B. bei erstem Treffer
LODS	lädt die Elemente eines Strings in AL/AX/EAX (sinnvoll in Verarbeitungsschleife)
STOS	speichert den Inhalt von AL/AX/EAX als String (sinnvoll in Schleife)

Letzter Fall, gehört halb zu Transporten: Beschaffung der effektiven („Prozess“)-Adresse; wird oft gebraucht, um die vom Prozess erarbeitete Adresse (z.B. nach Basis/indizierter/indirekter Adressierung) festzuhalten oder weiter zu bearbeiten; liefert nicht die Maschinenadresse (d.h. Adresse nach z.B.: Seiten-adressierung)

Beispiel Pentium II: LEA load effective address

(Leitwerksbefehle)

Sprünge: Mit ca. 25% Anteil nicht etwa wegen des go to-Trauma schlimm: Sprünge gestalten die Befehlsadressfolge unvorhersehbar, ruinieren vorsorgliches Befehlsholen, Pipelining und Superskalarität.

Unbedingter Sprung: Einadressbefehl; Modifikation der Adresse ist wichtig für Vielfachverzweigungen (case, computed go to), daher auch Anderthalbadressformat häufig (Steuergröße im Register).

Beispiel: Sprünge im Pentium II

JMP Jump

Schlichter Einadressbefehl in 4 Hauptvarianten, weiteren durch Adress-modifikationen.
Der Pentium benutzt für jeden Prozess einen (virtuellen) Adressraum aus maximal 16k Segmenten zu je maximal 4 GB Adressraum (also höchstens $2^{14}*2^{32}=2^{48}$ B adressierbar je Prozess).

Als Sprungziel kommt in Frage:

- eigenes Segment (32b Adresse oder 8b relativ; über Register oder Speicher ebenfalls möglich)
- fremdes Segment über Call Gate und Berechtigungskontrolle; Call Gate ist Datenstruktur wie Segmentdeskriptor
- andere Task über Task Gate.

Erst die Gates enthalten Zieladressen (Zugriffsschutz!).

(Leitwerksbefehle)

Bedingte Sprünge

- Einadressbefehle, wenn Bedingung im Operationscode („Springe, wenn Inhalt des Akkumulators Null“ oder „Springe, wenn Überlauf“). Besondere Rolle: Condition Codes, geben Aufschluss über Ausgang letztvollzogener Operationen (z.B. „null“, „negativ“, …). Klassische Lösung! Aber unbrauchbar, wenn Reihenfolge der Befehle modifiziert wird (Condition Code schon bereitgestellt? Noch aufzuheben?)
- Zweiadressbefehle, wenn ein beliebiges Register die Bedingung stellt (z.B. „R12 positiv“)
- Dreiadressbefehl, wenn zwei beliebige Register verglichen werden.

Die Lösungen über Register bleiben geeignet, wenn von der Ausführungsreihenfolge abgegangen wird! Bedingter Sprung muss Schreiben in „sein“ Register abwarten, Register bleibt reserviert bis zu „seinem“ bedingten Sprung.

(Leitwerksbefehle)

Unterprogrammsprünge/Rücksprünge:

Aufgaben:

Sprung mit
- Sicherung von bisherigem Kontext, z.B. Registerinhalte (nicht Prozessstatus im Betriebssystem-Sinn!)
- Rücksprung vorbereiten
- Zugang zu Parametern schaffen

Rücksprung mit
- Abbau von lokalem Status
- Manchmal: Rückübertragung von Ergebnissen (call by result)

Verschiedene Techniken, unterscheidbar vor allem nach Bereitstellung von Rücksprungadresse und Parametern.

Beispiel Pentium II

Call:

- Gewöhnlicher Unterprogrammsprung im eigenen Segment (absolut, relativ zu Befehlsadressregister oder indirekt über Register oder Speicherzelle)
- Mit Segmentwechsel bei gleicher Privilegierung; dann Kellerung der bisherigen Segmentkennung (segment selector)
- Mit Segmentwechsel zu größerer Privilegierung (geringeres protection level): nur indirekt über einen call gate genannten Deskriptor (Aufruffilter).

Genaueres im Adressierungskapitel.

Call benutzt ESP als Stackpointer, EBP als Basispointer (frame pointer, Basisadresse des Aufrufsatzes).

Ret:

- Rücksprung mit Freigabe des Stacks (stack pointer ← base pointer). Eventuell Segmentkennung wieder herstellen.

Dazu zwei weitere Befehle für Auf/Abbau des Aufrufsatzes (activation records):

Enter:

1. Parameter: Größe des Aufrufsatzes,

2. Parameter: Zahl der textuell umgebenden Blöcke (statische Schachtelungstiefe). Für Display (current environment vector, Vektor von Base Pointern der aktuell globalen Aufrufsätze).

Leave: Baut Aufrufsatz wieder ab

Also Folge:

Call

Enter

<Prozedurrumpf>

Leave

Ret

Systemaufrufe:

Sprünge mit Übergang in privilegierten Modus/Niveau, mit Angabe der gewünschten Systemdienstleistung, eventuell Pointer auf Versorgungs-block.

Ablauf:

- Übergang auf Stack des Zielmodus
- Status des bisherigen Prozesses kellern (nicht nur Register, sondern auch Programmstatuswort, Betriebssystemdaten dieses Prozesses (Priorität etc. → Prozesskontrollblock)
- Übergang auf Zielmodus
- Verzweigung zum gewünschten Dienst, Prüfung der Berechtigung
- Prüfung der mitgebrachten Parameter (sind das vielleicht Adressen, zu denen der Aufrufer gar nicht zugreifen (lassen) darf?

Rückkehr aus dem System:

- Setzt Prozess wieder ein (privilegiert!)

Beispiel Pentium II

Kein Systemaufruf, keine Systemrückkehr!
Konsequenterweise durch Call, Ret erledigt, über Call Gate – Deskriptor, vgl. Kapitel Adressierung. Dabei wird Funktion des Prozess-Rettens/Restaurierens teils in Mikrocode, teils in angesprungenes Segment verlagert.

Ähnliche Struktur: Übergang in Unterbrechungsbehandlung

Typisch durch Aufnehmen eines Unterbrechungssignals (interrupt request), aber auch durch Befehl (programmierte Unterbrechung). Rückkehr durch (i.a. privilegierten!) Befehl.

Beispiel Pentium II
INT, INTO, BOUND, IRET

Und der letzte der Leitwerksbefehle:

Aufruf eines Mikroprogramms:

Motiv:

- Der Kunde hat sein eigenes Mikroprogramm für eine Operation(sfolge) geschrieben (wehe ihm, er arbeitet ohne Netz im ungeschützten Prozessor, den er nicht versteht).
- Der Hersteller will die Konkurrenz abschütteln, in dem er Schlüssel-befehlsfolgen ins dunkle Innere des Prozessors verlagert.
- Zeitkritische Befehlsfolgen werden schneller (Abrufphasen entfallen).
- Natürlich fast ausgestorben!

Adresse im Befehl liefert Sprungziel im Mikroprogrammspeicher. Dort muss korrekte Funktion und Rückkehr realisiert sein (Befehlsadresse setzen, Abrufphase (Mikroprogramm) aufrufen.

Rechenwerkbefehle

Exkurs:
4 Blätter aus Rechnerarchitekturvorlesung 2006/7 von Herrn Prof. Gerndt:

Gleitpunktformate und ihre Bedeutung.

Die Formate, Bedeutung und die Algorithmen sind in IEEE Standard 754 (1985) festgelegt.

ISA – Gebräuchliche Datentypen

einfache Genauigkeit

Gleitkommazahlen ...

- Wert einer Gleitpunktzahl (nach IEEE 754):

e	m	Wert	Anmerkung
$e = 0$	$m = 0$	0	Null
$e = 0$	$m > 0$	$(-1)^s \cdot 0.m \cdot 2^{-b+1}$	denormalisiert
$0 < e < e_{max}$		$(-1)^s \cdot 1.m \cdot 2^{e-b}$	normalisiert
$e = e_{max}$	$m = 0$	$\pm\infty$	Unendlich
$e = e_{max}$	$m > 0$	NaN	*Not a Number*

- wobei
  - $e_{max} = 255, b = 127$ für einfache Genauigkeit (b: *bias*)
  - $e_{max} = 2047, b = 1023$ für doppelte Genauigkeit

ISA – Gleitkommazahlen

- Normalisierung
  - Eindeutige Darstellung
  - Mantisse wird unter Anpassung des Exponenten solange verschoben, bis eine 1 vor dem Komma steht
  - Diese 1 wird nicht explizit abgespeichert
- Ausnahmen:
  - Die Null hat zwei Darstellungen (+0, -0)
  - Denormalisierte Darstellung für Zahlen nahe bei Null
  - Spezielle Werte: Unendlich und NaN
    - Unendliche kennzeichnet Überlauf des Ergebnisses
    - NaN repräsentiert undefinierte Ergebnisse, z.B. bei 0/0

Gleitkommazahlen: Beispiele

- $00\ 80\ 00\ 00_h$ in Hexadezimaler Schreibweise
  - S=0
  - E=1
  - M=0
  - Wert: 2^{1-127} x $1.000..0_b$
- $3F\ 80\ 00\ 01_h$
  - S=0
  - E=127
  - M=1
  - Wert: 2^0 x $1.00\ldots01_b$ =$1+2^{-23}$

ISA - Gleitkommazahlen

- Zahlbereich für normalisierte Zahlen:
  - einfache Genauigkeit:
    $\pm\, 2^{-126} \ldots (2 - 2^{-23}) \cdot 2^{127} \approx \pm\, 1.2 \cdot 10^{-38} \ldots 3.4 \cdot 10^{38}$
  - doppelte Genauigkeit:
    $\pm\, 2^{-1022} \ldots (2 - 2^{-52}) \cdot 2^{1023} \approx \pm\, 2.2 \cdot 10^{-308} \ldots 2 \cdot 10^{308}$
- Zahlbereich für denormalisierte Zahlen:
  - einfache Genauigkeit:
    $\pm\, 2^{-149} \ldots (2 - 2^{-23}) \cdot 2^{-126} \approx \pm\, 1.4 \cdot 10^{-45} \ldots 1.2 \cdot 10^{-38}$
  - doppelte Genauigkeit:
    $\pm\, 2^{-1074} \ldots (2 - 2^{-52}) \cdot 2^{-1022} \approx \pm\, 5 \cdot 10^{-324} \ldots 2.2 \cdot 10^{-308}$
  - Verlust an Genauigkeit bei Annäherung an die Null!

Und nun zu den Rechenwerkoperationen:

Einstellige Operationen:

- Datentypkonversionen: kurz/lang, (Festpunkt/Gleitpunkt/ Bytestring/Dezimalstring)
- Normalisierung einer Gleitpunktzahl
- Kehrwert, Vorzeichenumkehr, Betragsbildung, Wurzel, Zählen
- Komplementierung eines Binärwortes (NOT)

Zweistellige Rechenwerkoperationen:

- Grundrechenarten: +, -, *, / für kurze/lange Formate, Festpunkt/ Gleitpunkt/Bytestring/Dezimalstring

- Boolesche Operationen: bitweises AND, OR, XOR, NAND, NOR

- Shifts:
  -- mit/ohne Vorzeichenberücksichtigung: arithmetische/logische Shifts, rechts/links
  -- zirkulär („Rotation“), oder mit Null/Eins/Vorzeichenauffüllung
  -- Stellenzahl

Ein/Ausgabebefehle (siehe auch Kap. 5!)

Zwei grundverschiedene Ansätze:

memory-mapped i/o: Die E/A-Steuerwerke (i/o controllers) besitzen adressierbare Zellen, die vom Prozessor wie Hauptspeicherzellen lesbar/beschreibbar sind

→ es gibt keine besonderen E/A-Befehle, sondern nur Transporte von/zu bestimmten Adressbereichen; heute überwiegend gebraucht.

Dabei kann das E/A-Steuerwerk nach Start weitgehend autonom Blocktransporte mit dem Hauptspeicher ausführen (DMA, direct memory access), dazu erhält es vom Prozessor einzelne Aufträge. Im Benutzermodus meist über privilegierte Adressraumüber-wachung abgefangen und an Betriebssystem übergeben.

Älterer Ansatz:

Besondere E/A-Befehle, die das Gerät über Kanal/Unterkanal-Nummer kennzeichnen; Befehl wird von Parameterblock begleitet, der Betriebsart beschreibt (ein/aus, positionieren, Vorschub, …). Im Benutzermodus als privilegierter Befehl vom Betriebssystem abgefangen. Betriebssystem setzt den E/A-Befehl in eine Kette von Anweisungen für den Kanal/Controller um, die dieser interpretiert (E/A-“Prozessor“). Natürlich wird hier ebenso DMA-Technik benutzt.

Daneben sind Einzelworttransporte an CPU-Schnittstellen in Gebrauch (z.B. Byte-out, Byte-in), für elementarste Ein/Ausgabe.

2.4 Adressierung

In der Transformation der in den Befehlen enthaltenen Adressangaben („Programmadressen“) liegen wichtige Freiheitsgrade der Programmier- und Betriebstechniken!

2.4.1 Wert statt Adresse

Oft ist es unnötig und unpraktisch, einen Wert erst mittels einer im Befehl enthaltenen Adresse aus dem Speicher zu holen. Man kann den Wert („immediate operand“) im Befehl unterbringen. Zwei wichtige Einschränkungen:

- Wert muss in das Adressformat im Befehl passen, oder Befehl erhält angepasstes Format
- Wert darf sich nicht ändern, sonst Änderung des Programms, mit schlimmen Folgen:
  -- unsicher
  -- Probleme bei Ausschnittsbildung, Sharing, Rekursion
  -- Probleme bei Parallelisierung (bereits in Verarbeitung befindliche Befehle könnten verändert werden)

Also: brauchbar für (meist kurze) Konstanten.

2.4.2 Programm-, Prozess-, Maschinen-adressen

Programmadressen: In Befehlen und als Adress-Werte im Programm vorliegende Adressen

Ebenso, aber Adressen in mehreren Adressräumen, → 2.4.5

Nach Vorgaben des Programms erzeugt der Prozessor aus Programm-adressen Prozessadressen, mittels: „Index“-Modifikation, Substitution, (Befehlszähler)relativer Adressierung, „offene“ Basisadressierung.

Ebenso, zusätzlich evtl. Abbildung vieler Programmadressräume in einen Prozessadressraum; mittels „offener“ Basisadressierung.

Hauptziel: Operanden/Ergebnis/Sprungadressen erst zur Laufzeit festlegen; Nebenziele: kurze Programmadresse, Erleichterung des Bindens

Falls nicht, dann:

Prozessadressen (effektive Adressen): im Prozessor benutzt

Ebenso, aber Adressen in mehreren Adressräumen. → 2.4.5

Prozessadressen (effektive Adressen): im Prozessor benutzt

Ebenso, aber Adressen in mehreren Adressräumen. → 2.4.5

Nach Vorgaben des Betriebssystems erzeugt der Prozessor aus Prozessadressen Maschinenadressen, mittels „verdeckter“ Basisadressierung, Seitenadressierung.

Hauptziele: beliebige Lage des Programms und seiner Werte/partielle Lagerung im Speicher

Ebenso, zusätzlich Abbildung vieler Prozessadressräume auf einen Maschinenadressraum; mittels verdeckter Basis-adressierung oder Seiten-adressierung

Maschinenadressen: vom Prozessor gegenüber dem Hauptspeicher benutzt (oft dann noch zerlegt in Moduladresse (vorzugsweise hintere Bits) und Adresse im Modul.

Nun Vorgehen:

Einführung der grundlegenden Adressierungstechniken als Lösungen zu programmiertechnischen, betrieblichen und anderen Forderungen.

Zunächst einzeln, mit Beispielen in Notation von adhoc erfundenen, meist graphischen Notationen.

Vormerkungen zur Transformation von Programmadressen in Prozessadressen:

Die „nicht privilegierten“ Modifikationen treten situationsabhängig auf (z.B. Adressierung auf dem Stack anders als im übrigen Speicher). Wenn sie in den Befehlsablauf integriert werden (wie meist im Folgenden angenommen), ergeben sich daher stark schwankende Ablaufstrukturen und Zeitbedarfe (auch gleichartiger!) Befehle. Das führt zu schlechtem Pipelining. RISC-Prozessoren ersetzen daher diese Modifikationen durch Vorbefehle („brauchen ja nur je einen Takt“), die die Adresse im Register aufbauen, über das dann indirekt auf den Speicher zugegriffen wird.

Die Funktion der Modifikation bleibt dieselbe! Sie wird nur auf mehrere Befehle verteilt. Speicheraufwendig! Durchsatzgünstig!

2.4.3 Transformation von Programmadressen in Prozessadressen:

Substitution, Indexmodifikation, Adressierung relativ zu Basisadresse/ Befehlszähler

Forderung I:
Der Rechenprozess muss die Programmadressen für Operanden, Ergebnisse, Sprungziele zur Laufzeit erzeugen/ändern können

- Unterprogrammparameter/Rücksprünge
- Mehrfachinkarnationen (Rekursion, dynamische Datenstrukturen)
- Datenstrukturen unbekannten Platzbedarfs (Dynamische Felder)
- Berechnete Indizes in Feldern

d.h. in Prozessadressen transformieren und zwar ohne Änderung der Befehle!

Lösung:
- Substitution (Ersetzung)
- Indexmodifikation (hier: fallweise angewendete additive Änderung der Adresse, vorgegeben durch Interpretationsprozess; anders: Basisadressierung (additive Korrektur, aber stets angewendet)

Notwendige Trennung der Adressinformation:

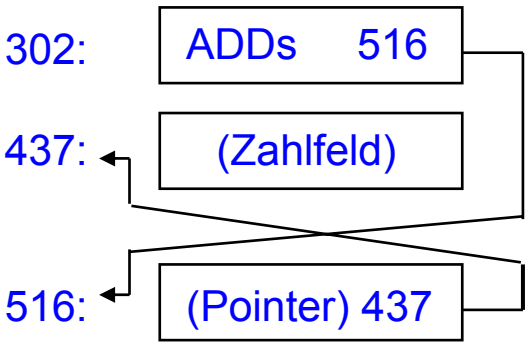
- im Programm (Befehl, Adresswort) steht, was zur Übersetzungszeit bekannt ist und dann ungeändert bleibt

- zur Modifikation wird verwendet, was erst zur Laufzeit bekannt wird; dabei berechnete Adressen werden im Datenbereich nicht im Programm geführt

Substitution:

(Adressersetzung, indirekte Adressierung; als Maschinenfunktion von H. Schecher (TUM) 1956 eingeführt)

Programmadresse bezeichnet nicht die Zelle des Wertes/Sprungziels, sondern Zelle/Register der Adresse des Wertes/Sprungziels (dies ist also „Objekt 2. Referenzstufe“!)



Beispiel: ADDs 516 maschinentechnisches Abbild von (Pascal) q+p↑Zahl wobei p in Zelle 516, Zahlfeld in Zelle 437 (1. Feld im durch p↑ bezeichneten Record).

Substitution ist bei früheren Maschinen auch iterierbar gewesen (DEC 10/20).

Substitution über Register besonders wichtig:

(„register indirect“)

- spart Bits im Befehl (wichtig bei langen Adressen)
- RISC-Prozessoren erlauben keine gelegentliche (Pipeline!) komplexe Adress-modifikation. Also Adresse durch Vorbefehlsfolge in Register aufbauen!

Indexmodifikation

Indexmodifikation ist eine additive Korrektur der Programmadresse um den „Index“, fallweise angewendet

Prozessadresse := Programmadresse + Index

Zwei Varianten:

Index ist Relativgröße, invariant gegen Lage im Programmadressraum (eigentlicher Index), Programmadresse im Befehlswort ist lageabhängig

- Index in einem Feld (array); typabhängige Schrittweite (skalierter Index) günstig!

Index ist lageabhängig, im Befehlswort steht eine Relativgröße (!), (Index wie Basisadresse)

- Parameterzugang, Rücksprungsadressen
- Zugang zu Variablen im Stack

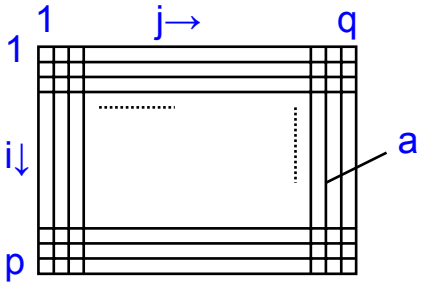
Indexmodifikation verlangt im Befehlsformat:

- Angabe, dass zu modifizieren ist: Extrabit, am besten der Adresse zuzuordnen, evtl. im Operator

- Angabe, welches Register zu benutzen ist

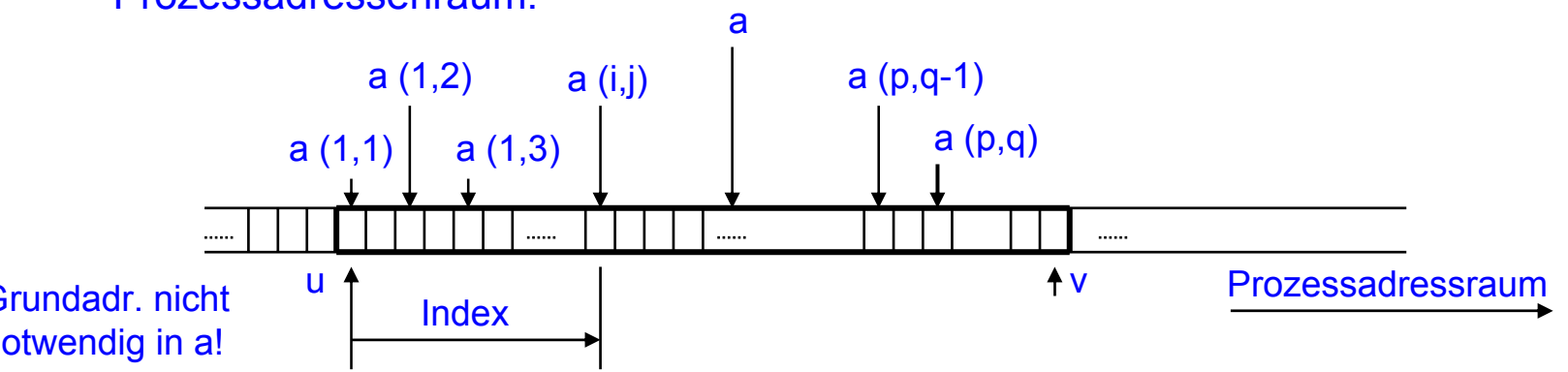
Index ist Relativgröße:

Beispiel:



Indizierte Variable a [i,j] adressieren
a: array [1..p, 1..q] of t
t brauche platzbedarf(t) Zellen

Prozessadressenraum:



Das rechnet i.a. das Programm

Index:= ((i-1)*q+j-1)*platzbedarf(t)
Prozessadresse := Grundadresse + (das tut der Prozessor) Index
- Vorsicht, dass i ε [1..p] ^ j ε [1..q]!
- Platzbedarf erscheint eine Zeile tiefer im Falle des skalierten Index!

Bemerkungen zur indizierten Variablen:

- Skalierter Index: Adressierung in Vielfachen des Platzbedarfs, ist günstig vor allem bei Vielformat-Maschinen
- Prüfung auf Einhaltung der Adressgrenzen ist wichtig!
- Arithmetische und Test-Operationen am Index (ist Festpunktgröße), z.B. auch falls Index zugleich Laufvariable in Schleife
- Informationen über Platzbedarf (t), Grundadresse, Stufigkeit, Indexgrenzen, Adressgrenzen (u, v) bildet Felddeskriptor, kann maschinengestützte Datenstruktur sein; dann Auswertung bei Feldzugriff durch Mikroprogramm.

Index ist lageabhängig

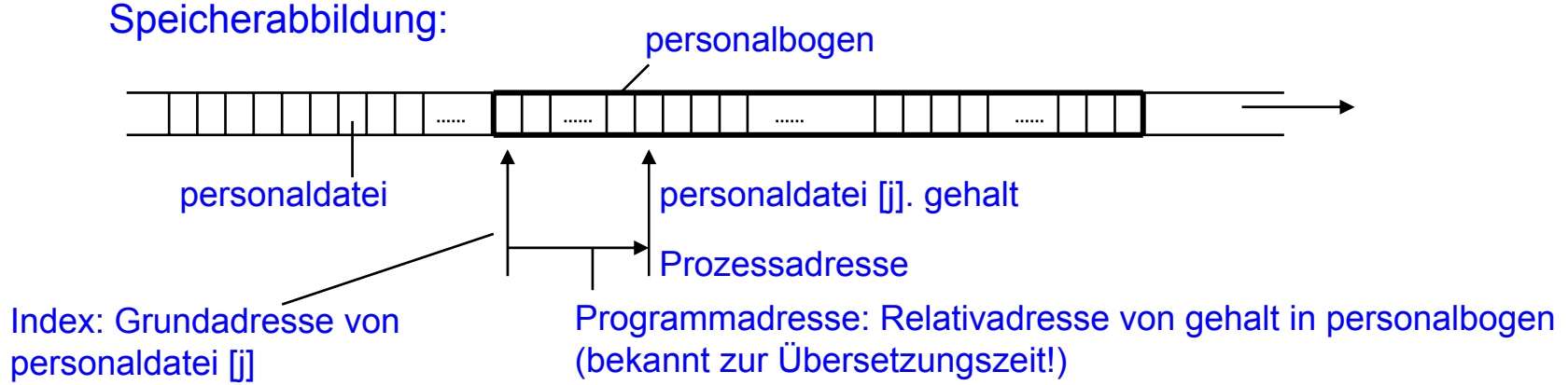
Beispiel 1: type personalbogen = record name: array [1..20] of character;

gehalt: real;

end;

var personaldatei: array [1..1250] of personalbogen

Speicherabbildung:

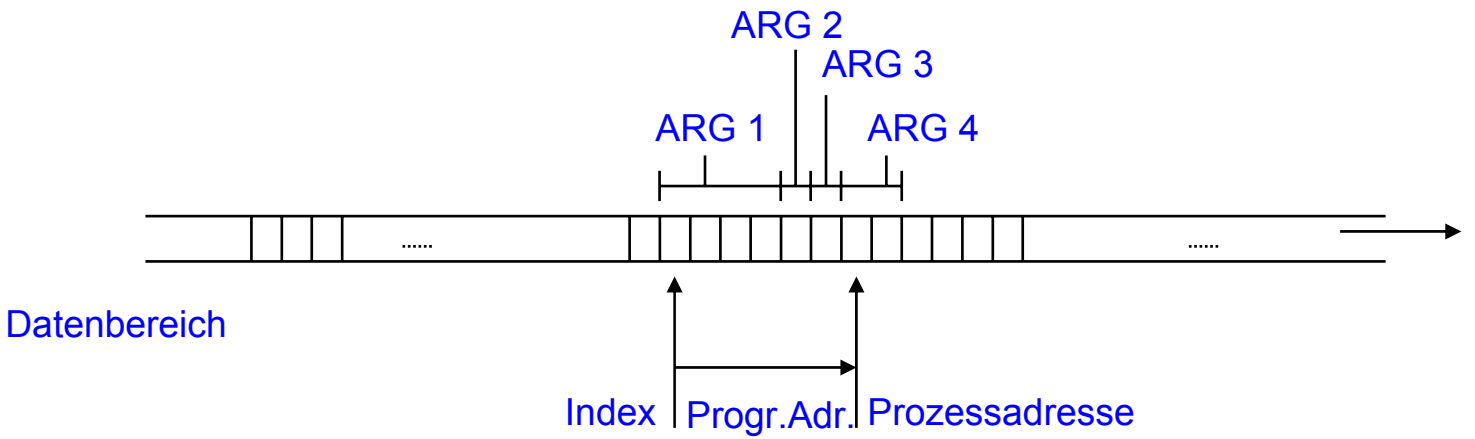


Hier wäre auch ein doppelter Index denkbar:
Index 1:= Grundadresse (personaldatei), Index 2:= j*platzbedarf (personalbogen)
Prozessadresse:= Programmadresse + Index 1 + Index 2

Index ist lageabhängig

Beispiel 2:

Index als Grundadresse der Argumentliste zu einem Prozeduraufruf



Index: Grundadresse, ist aufrufabhängig, u.U. zur Programmier/Übersetzungszeit unbekannt

Programmadresse: Relativgröße, ist wegen spezifizierter Argumentliste/Typen zur Programmier/Übersetzungszeit bekannt: Unveränderlicher Adressteil im Befehl!

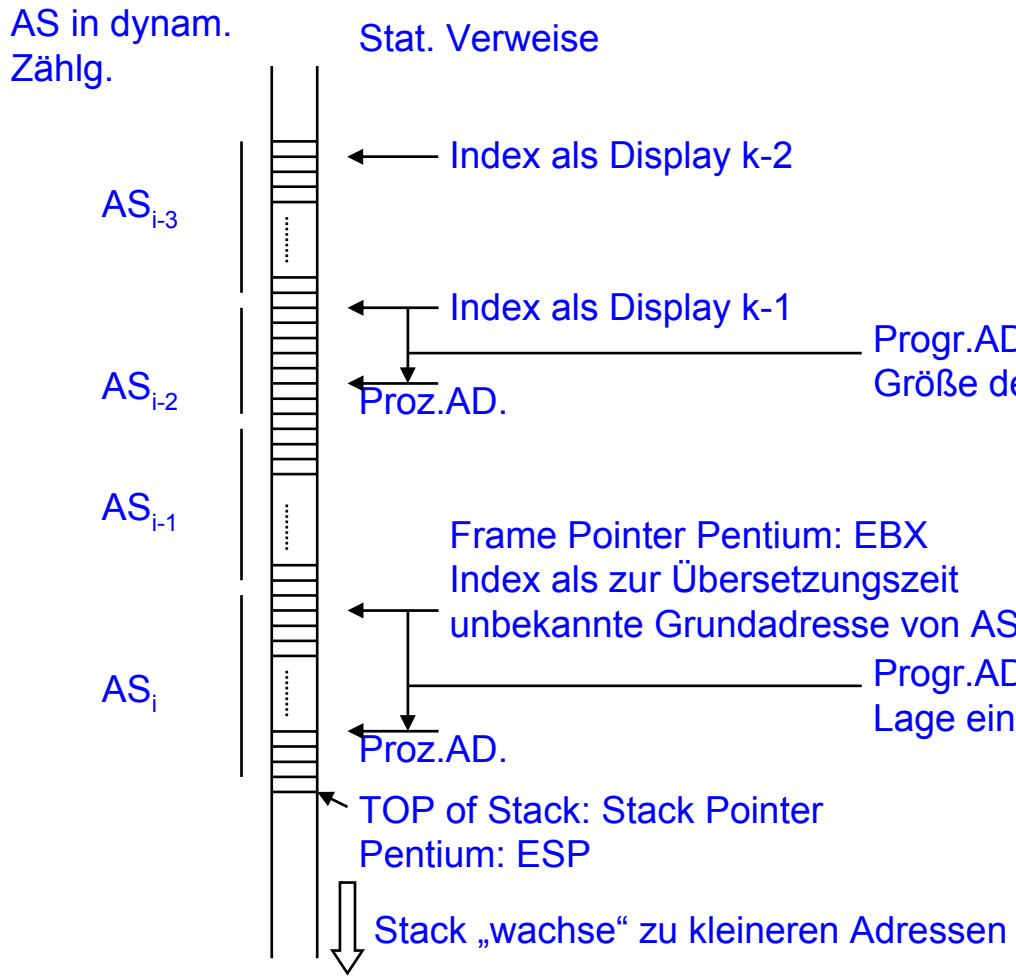
Index ist lageabhängig

Beispiel 3: Laufzeitstapel (-Keller, runtime stack). Für Programme mit (Block) Prozeduraufrufschachtelung (inkl. Rekursion), wie aus Einführung in die Informatik bekannt, wird für jeden Block/ Prozedur-Eintritt/Aufruf eine Datenstruktur.

Aufrufsatz (AS, activation record), bestehend aus:

- Organisationsgrößen (Grundadressen des Vorgänger-AS und des umfassenden nach der Ordnung der textuellen Schachtelung (d.h. die globalen Größen definierenden))
- lokalen Variablen (soweit Platzbedarf zur Übersetzungszeit bekannt)
- Parameter (Versorgungsblock)
- Registerinhalte einschließlich Bef.Zähler (zu retten bei Folgeaufruf)
- Platzbedarf für lokale Variable, soweit erst vor Eintritt bekannt
- Weiteres (z.B. Ausdrucksberechnungskeller)

Index ist lageabhängig
Laufzeitstapel aus Aufrufsätzen (AS)

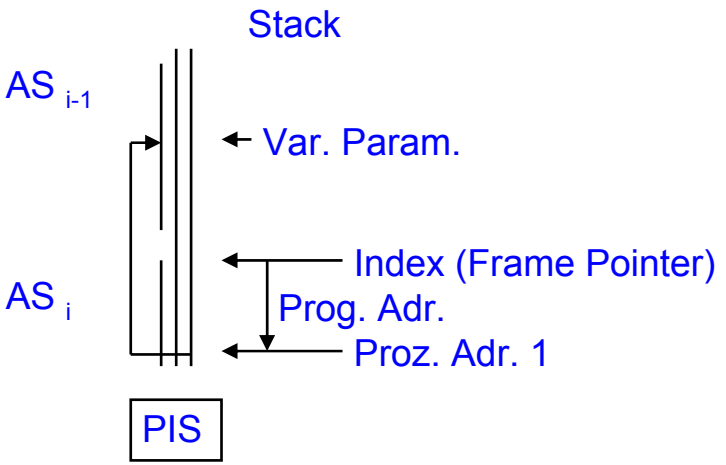


AS_{i-3}, AS_{i-1} seien Teile der Umgebung (d.h. die globalen Größen definierend)

Sun UltraSPARC und Itanium führen den jüngsten Teil des Laufzeitstapels automatisch im Registerblock

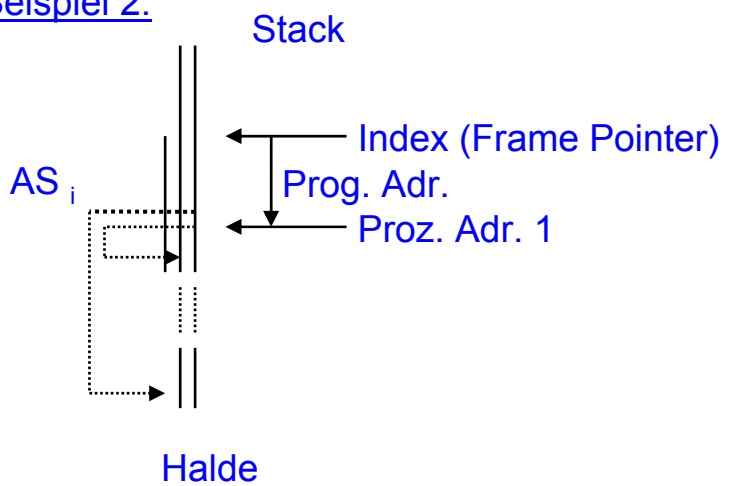
Konsekutive Anwendung von Index-Modifikation und Substitution:

Beispiel 1:



Zugriff auf Variablenparameter aus dem AS_{i-1}:
Programmadresse: Relativ-Position
Index: Grundadresse (AS_i)
Proz.Adr.1: Zeigt auf Pointer (Adresse), substituiert
→
Proz.Adr.2: Dort Variablenparameter

Beispiel 2:

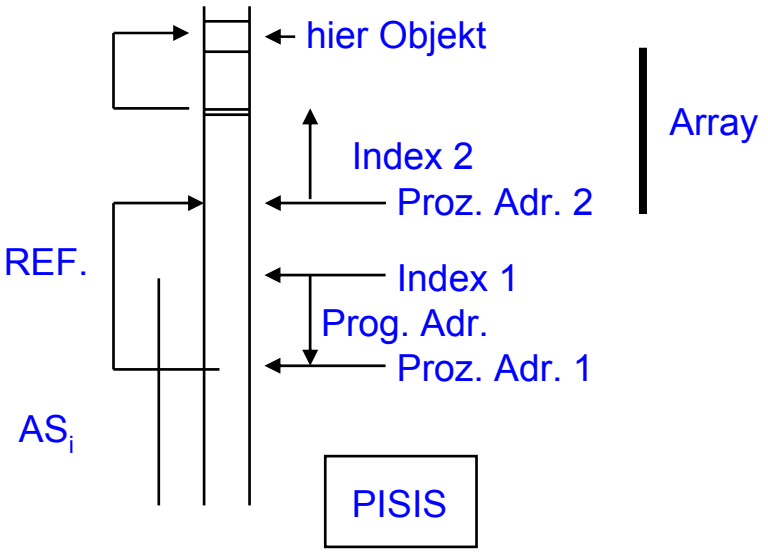


Zugriff auf Variable unbekannten Platzbedarfs im Stack, oder auf Halde

PIS d.h. Programmadresse,
dann Indexmodifikation
dann Substitution

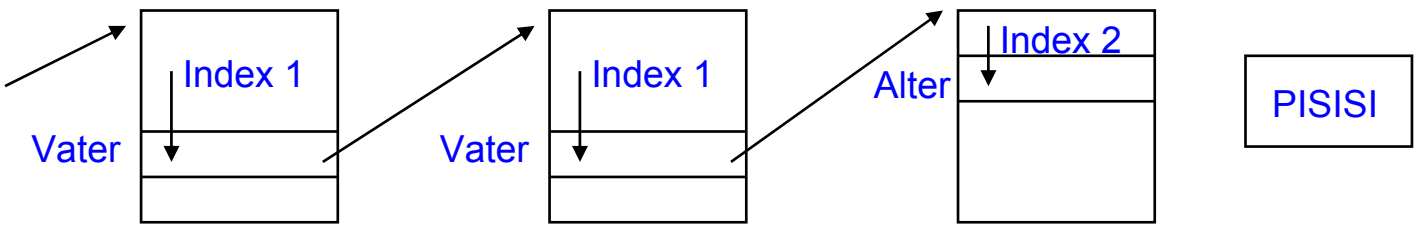
Konsekutive Anwendung von Index-Modifikation und Substitution:

Beispiel 3:



Zugriff auf ein Objekt über indizierte Pointervariable in einem Array, das als Referenceparameter übergeben wurde

Beispiel 4:



Es gab Maschinen mit Index/Substitutionsfolge beliebiger Tiefe: DEC 10/20

Aber heute ist das was für eine Berechnung in allgemeinen Registern, mit Ersetzungen.

Forderung II:
Kurzadressen innerhalb der aktuell zugegriffenen Teilmenge der Hauptspeicherzellen (Nutzung der Lokalität).

(Keine allgemeine Lösung):
Substitution über Register. Wegen Ladens der Register und wegen begrenzter Registerzahl nur für geringe Zahl von Adressen akzeptabel.

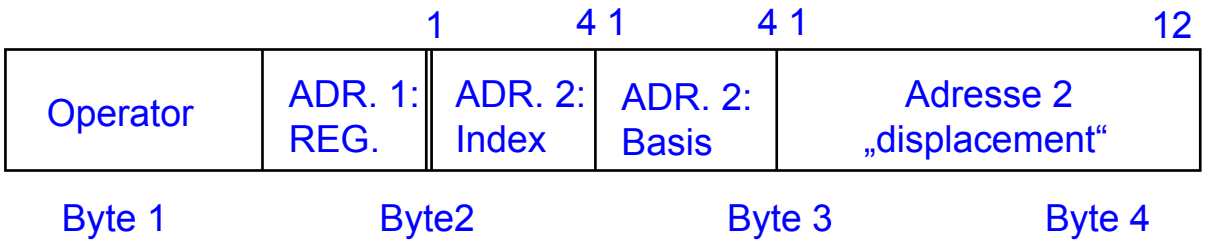
Lösung 3:
Befehlszählerrelative Adressierung. Alle (oder einige) Adressen sind als Relativangaben zum Befehlszählerstand zu verstehen

- oft bei (kurzen) Sprüngen
- auch für Variablenadressierung geeignet, aber dann Vermischung von pure und impure part schwer vermeidbar. Variablenadressierung für Compiler aufwendig!

Lösung 4:
Jeder Zugriffstyp greift in einen Adress-Unterraum, z.B. für Programm, Stapel, Halde (z.B. PDP11, Intel 8086). Notbehelf. Vgl. auch 2.4.5 („Segmente“).

(Forderung: Kurzadressen)

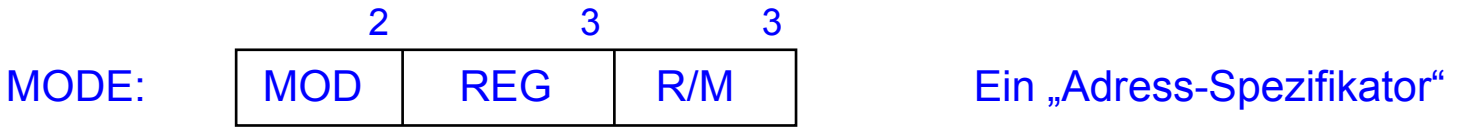
Lösung 5:
Offene Basisadressen (relative Adressierung). Zu jeder Wert- oder Sprungadresse wird eine Basisadresse angegeben, ihr Wert wird addiert.
Eigentliche Adresse ist lageunabhängiger Relativwert, z.B. nur 12 Bits bei 24 Bits Basisadresse.
Klassisches Beispiel: IBM/360 und Nachfolger (1965). Eine Maschine mit verschiedenen Befehlsformaten, z.B. RX-Format:



In 16 allgemeinen Registern stehen aktuelle Operanden, Ergebnisse, Indizes, Basisadressen.
Prozessadr. 2 := Adresse 2 + <ADR. 2 Index> + <ADR. 2 Basis>
Da die Basisadresse 24 oder 32b lang ist, werden je Adressangabe 8 oder 16b gespart, aber dafür Aufwand für Laden der Basisadresse, Belegung eines Registers!

Beispiel Intel Pentium II:

Hier nur der 32b-Protected Mode!
Viele (nicht alle!) Befehle erlauben Werte/Sprungziele im Speicher, dann typisch 1½-Adressformat.
Steuerung der Modifikation über Mode-Byte im Befehl:



R/M: Gibt an, ob für die Adresse EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI benutzt wird oder ob ein SIB-Byte folgt

MOD: Bestimmt mit R/M folgende Adressierungstypen:
- Register
- Register indirekt (aber nicht SP, BP!)
- Register mit Displacement 8b oder 32b (aber nicht SP!)
- Absolute Adresse

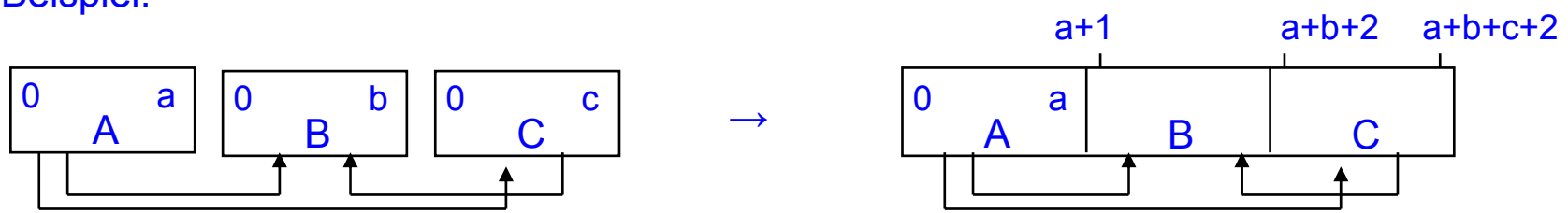
SIB: Skalenfaktor, je ein Register als Basis und als Index: Skalierter Index mit und ohne Displacement

Forderung III:
Die Adressierung unterstütze das Binden

Binden (linking, Montage): Aufbau eines zusammenhängend linear adressierten Programms (mit Werten) aus Moduln, die je von Null bis zu einer individuellen Obergrenze adressieren.
Dazu sind:

- alle lageabhängigen Adressen additiv zu korrigieren nach endgültiger Anfangsadresse des Moduls
- alle Querbezüge (aus dem Modul auf externals) endgültig zu adressieren

Beispiel:



Binden findet vor dem Programmlauf statt, anhand von Compilerprotokollen über lageunabhängige Größen und externals.

Lösungen für die Unterstützung des Bindens:

Befehlszählerrelative Adressierung erübrigt die additive Korrektur der Adressen, nicht das Einsetzen der externen Referenzen; (position-independent code, PIC).

Adressierung über offene Basisadressen reduziert die additive Korrektur auf die Korrektur der Basisadressen.

Wiederaufnahme der Überlegungen: Segmentweise Adressierung (Binden auf die Laufzeit verschieben) → 2.4.5

Ähnliches Problem: Laden (Fixieren) → 2.4.4

2.4.4 Transformation von Prozessadressen in Maschinenadressen: Verdeckte Basisadresse, Seitenadressierung

Zunächst: Keine Transformation durch Prozessor

Im einfachsten Fall sind die Prozessadressen bereits die Maschinenadressen. Zwei unangenehme Folgen:

- Wenn das Programm lageabhängige Größen enthält (nicht zu modifizierende Adressen, offene Basisadressen, oder (!) lageabhängige Indizes, Substitutionsadressen), dann ist das Programm auf den Ausführungsort festgelegt. Nicht akzeptabel bei Mehrprogrammbetrieb im Hauptspeicher! Abhilfe programmiert durch „Lader“. (Kein Probem bei befehlszähler-relativer Adressierung!)
- Kein (veränderlicher) aktueller Ausschnitt des Prozess-adressraums im Hauptspeicher. Abhilfe programmiert durch „Overlay“. (Kein Problem, wenn Hauptspeicher ausreichend groß!)

Lader (Fixierer, Loader) benutzt wie Binder, mit dem er oft ein Programm ist, vom Assembler/Compiler hinterlassene Tabellen, die angeben, wo im Programm und seinen Werten lageabhängige Größen (nicht zu modifizierende Adressen, offene Basisadressen oder (!) lageabhängige Indizes, Substitutionsadressen) stehen. Typisch erzeugen Assembler/Compiler Programme, die Prozess-adressen von Null aufsteigend generieren; dann genügt Addition der vorgesehenen Anfangsadresse des Programms (im Maschinenadressraum) auf alle lageabhängigen Größen.

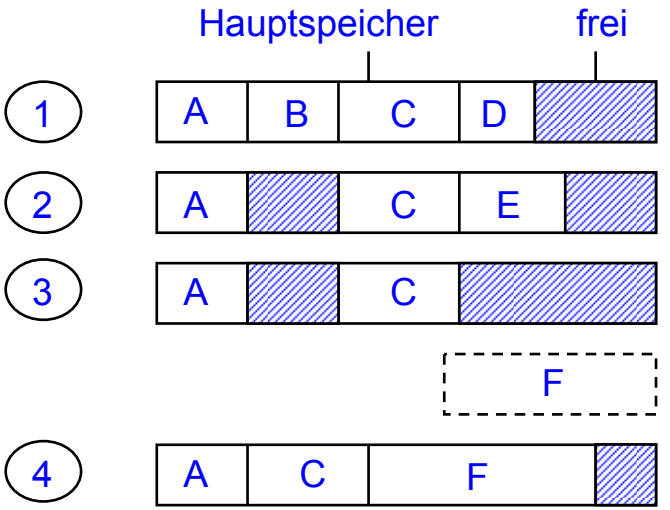
Forderung III:

Unterstützung des Laders

Bereits bekannt: Offene Basisadressierung (vgl. das zum Binden gesagte).

Und übrigens: Bei befehlszählerrelativer Adressierung entfällt das Laden.

Noch schöner wäre es, wenn die Abbildung der Prozessadressen auf die Maschinenadressen veränderlich wäre:

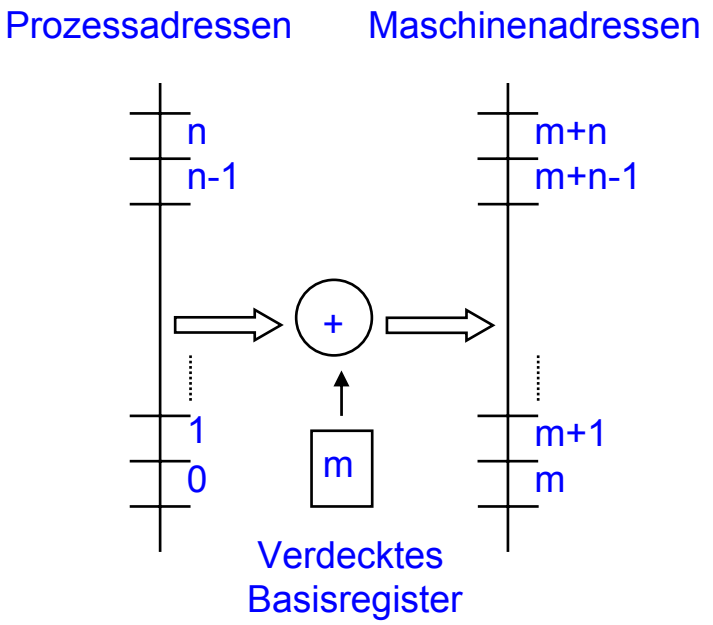


B, D fertig, E eingelagert

E fertig, F wäre einlagerbar, wenn C verschieblich wäre

Das ist aber mit dem Lader nur möglich, wenn bekannt ist, wo C lageabhängige Größen führt. Da das Programm diese aber beliebig umspeichern darf, sind die vom Assembler/Compiler angelegten Tafeln wertlos, sobald das Programm angelaufen ist.

Forderung IV:
Lageinvariantes Programm
Schon bekannte Lösung:
Befehlszählerrelative Adressierung.
Andere Lösung: Verdeckte Basisadresse (1962, IBM 7094).
Der Prozess benutzt einen Prozessadressenraum von 0 bis n. Der Prozessor erhöht vor jedem Zugriff auf den Hauptspeicher die Prozessadresse um den Wert m der Anfangsadresse des zugeteilten Bereichs im Hauptspeicher aus einem „verdeckten“ Basisregister. Dieses ist privilegiert, Inhalt nur im Systemmodus setzbar. Typisch kontrolliert der Prozessor an einem zweiten Register, ob die Prozessadresse höchstens n ist.



Verdeckte Basisadressierung

- Verschiebung jetzt möglich: Transport des Programms mit seinen Werten, dann Umsetzen der Basisadresse
- Addition meist dadurch erleichtert, dass z.B. letzte 5 oder 10 Bits der Basisadresse null sind, d.h. Programme beginnen auf 32/1K Adressen-Raster. Folge: „Verschnitt“ durch u.U. nicht nutzbare Hauptspeicherzonen („fragmentation“).

Verschiebetechniken

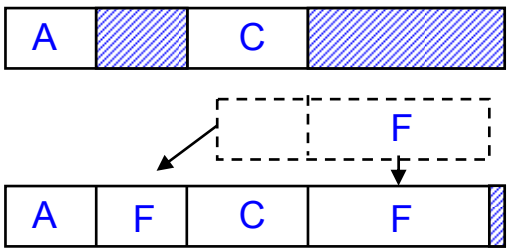
sind problematisch, weil

- Programme nicht immer verschiebbar sind (bei Ein/Ausgabe, z.B.)
- Verschiebezeitanteile lang (!) sein können.
  --Schlecht: wenn Prozessor das Programm mit Blocktransport verschiebt.
  --Besser: Verschiebung durch swap (Ausgabe/Eingabe auf/von Platte).
  --Noch besser: Verschiebung durch blocktransportierende Steuerwerke oder Kanäle.

Forderung V:

Zerteilte Unterbringung von Objekten im Maschinenadressraum (Hauptspeicher) hat die Gründe

- Verschieben soll unnötig sein, weshalb nicht so:



- Gemeinsame Nutzung von Speicherbereichen soll für beliebige Prozesse (nicht nur zwei im Speicher benachbarte) möglich sein.

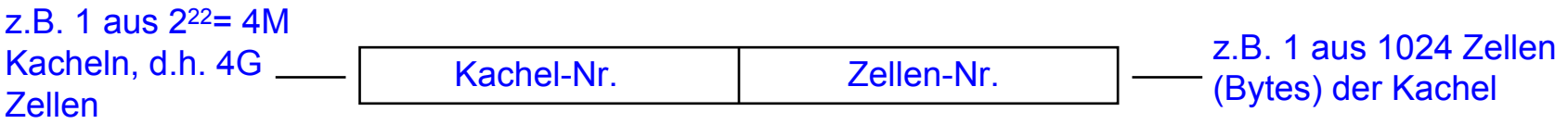
Lösung 7:

Seitenadressierung

(für den zweiten Grund ist u.u. segmentweise Adressierung die bessere Lösung)

Seitenadressierung

Der Maschinenadressraum wird in Abschnitte fester Größe geteilt:
Kacheln, Seitenrahmen (page frames, physical pages), z.B. 2^{10} = 1024 Wörter (4KB). Maschinenadresse zerfällt:



Ebenso wird der Prozessadressraum in Seiten (pages) gleicher Größe geteilt. Die Prozessadresse zerfällt, z.B.

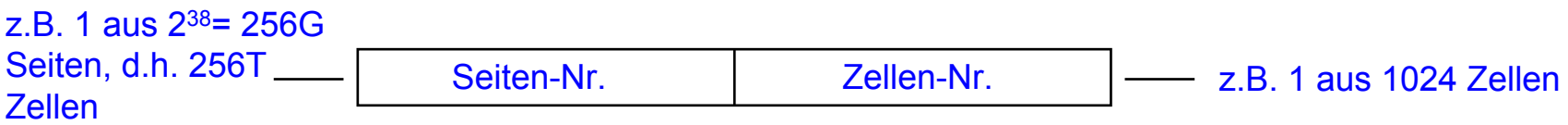
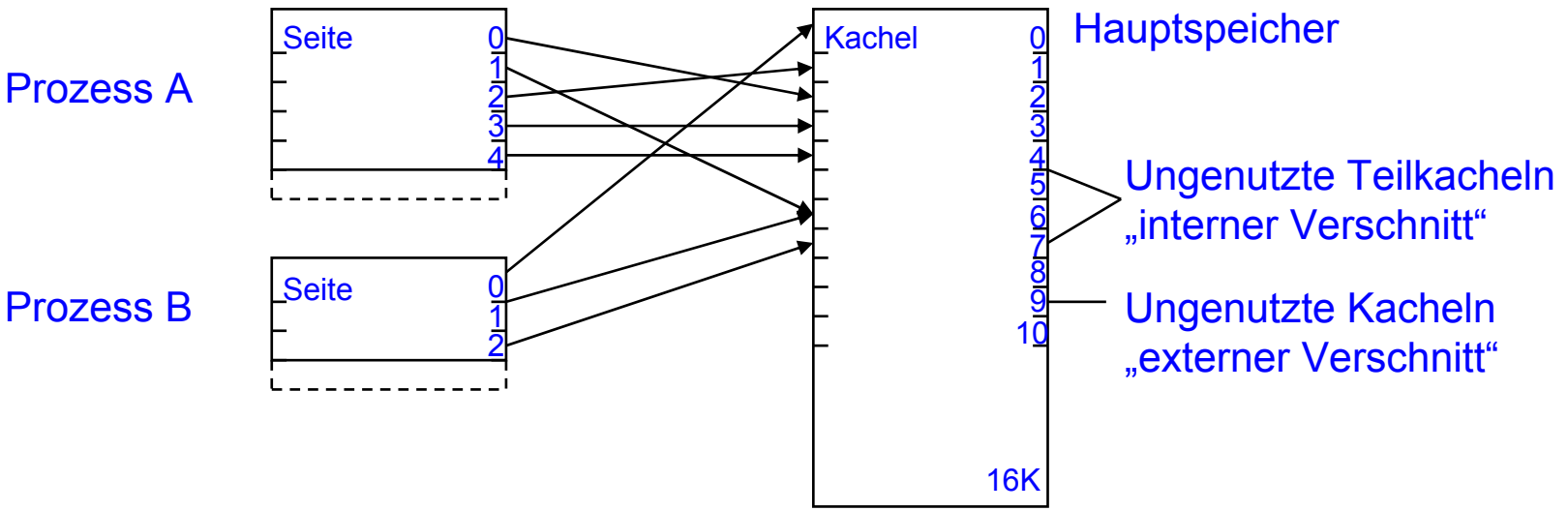


Abbildung von Seiten-Nummern auf Kachel-Nummern durch den Prozessor anhand von privilegierten Tafeln.

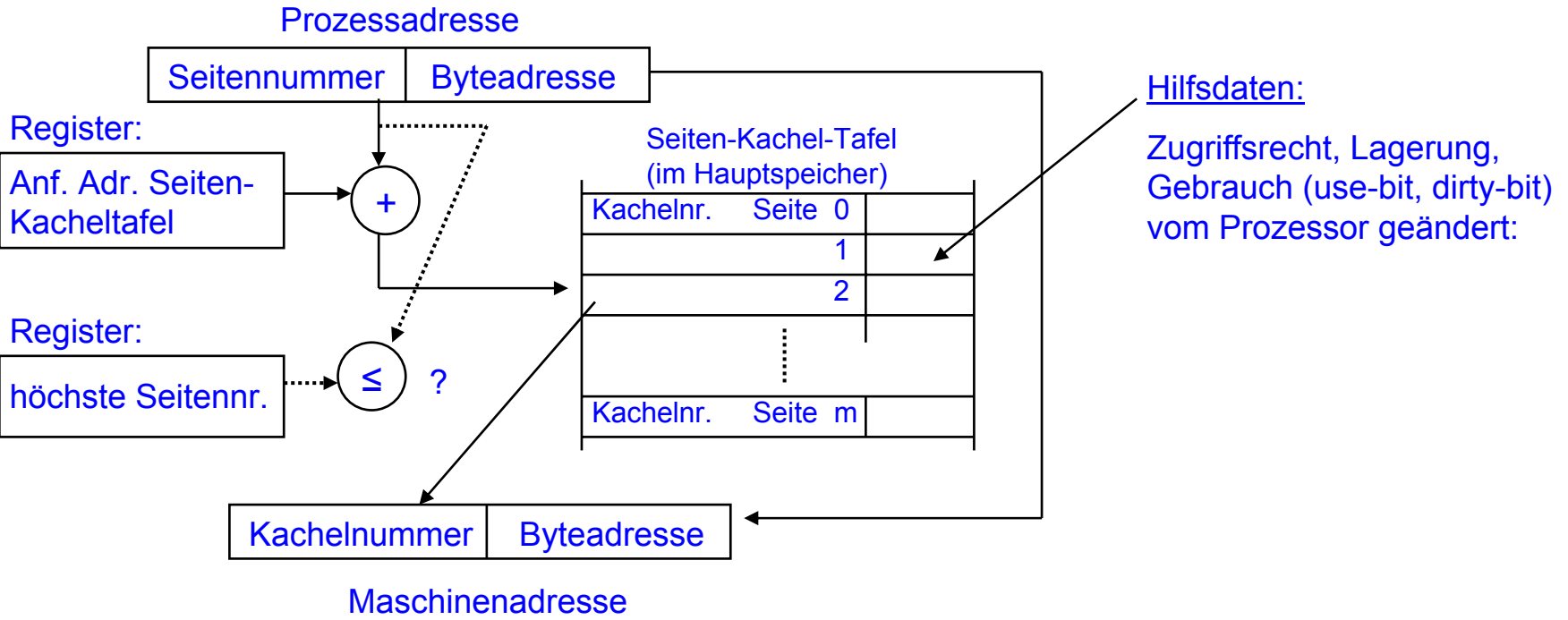
Abbildung Seiten → Kacheln erlaubt die gewünschte Zerteilung im Maschinenadressraum:



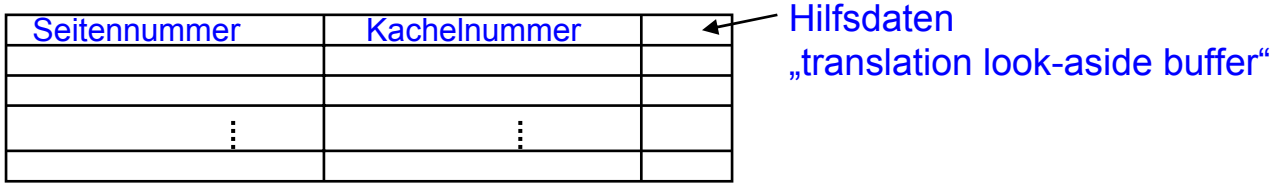
Seiten/Kachelgröße ist ein Kompromiss zwischen

- Abwägen: Kleine Tafeln für Abbildung (→ lieber große Seiten) und geringer interner Verschnitt (lieber kleine Seiten), spricht für ca. 64..256B
- Forderung, dass Seite auch Transportformat im Verkehr mit peripherem Speicher (lieber große Seiten)

Abbildung Seitennummer → Kachelnummer wird über dem Prozess eigene, aber ihm unzugreifbare Seiten-Kacheltafel realisiert, vom Betriebssystem aufgestellt



Jeder Griff in den Prozessadressenraum führt über die Tafel (Verlangsamung!!) oder (meist) 4 .. 256 „assoziative“ Register, mit den letztgebrauchten Seitennummer-Kachelnummer-Paaren



Aktueller Ausschnitt aus Seiten-Kachel-Tafel!

Das Register für die höchste Seitennummer

- schützt den Prozess vor vermutlich schweren Fehlern (Zugriff auf nicht existente Objekte außerhalb des ihm zugeteilten Prozessadressenraums)

- schützt das System vor schwerwiegenden Fehlzugriffen in den Maschinenadressraum (indem beliebige Zelleninhalte aus Kachel-Nummern aufgefasst werden)

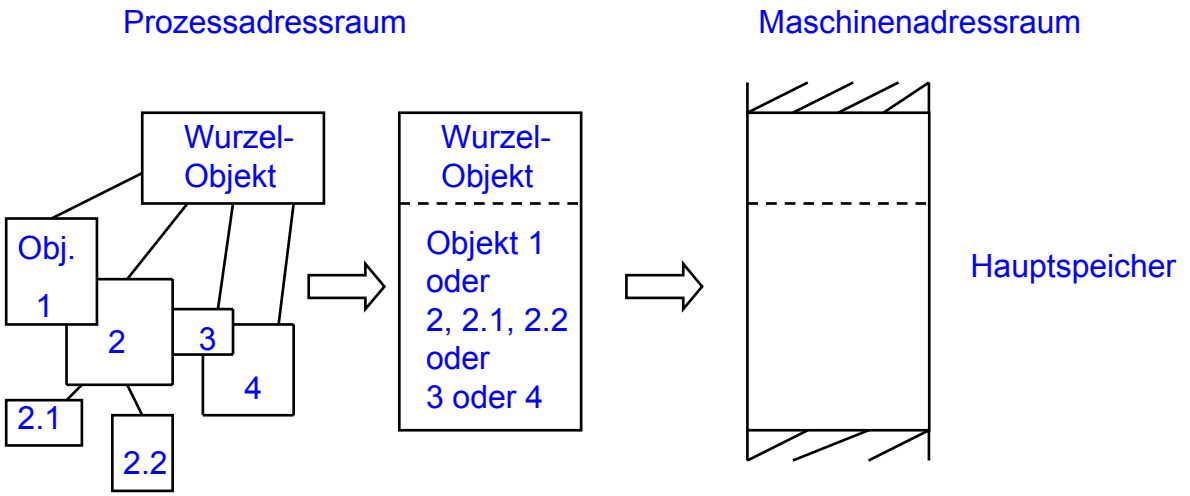
Seitenadressierung als Lösung des Zerteilungsproblems ist eine anerkannt schöne Sache, und sie erfüllt auch noch die Forderung V!

Forderung V:

Nur ein wechselnder aktueller Ausschnitt des Prozessadressenraums wird aus den Maschinenadressraum abgebildet. Dabei soll der Prozessadressenraum im Verhältnis zum Hauptspeicher

- größer (das ist die Regel)
- kleiner (bei Epigonen von Rechnern mit ungenügendem Adressraum) sein.

Lösung 8 (keine!):
Overlay



Der Prozessadressraum wird durch Überlagerung mehrfach belegt: Eindeutigkeit nur für ein Objekt und seine Vorfahren im Baum. Kommunikation nur über gemeinsame Vorfahren. Programmaufbereitung durch Binder.

Heutige Lösungen für den aktuellen Ausschnitt im Hauptspeicher:

Großer Adressraum 2^{32} .. 2^{64}B, gegliedert in

Seiten	Segmente
(Die Regel, starres Format)	(Progr./Datenstruktur bestimmt Format)
Güntsch 1957	Barton 1963
Kilburn 1962 (Ferranti Atlas)	(Burroughs B 5000)

Von diesen wird eine aktuelle Auswahl im Hauptspeicher gehalten, der als virtueller Speicher wirkt.

Lösung 9:
Seitentausch
Seiten-Kachel-Tafel enthält Lagerklasse (Hauptspeicher, Platte). Versuch, eine im Hauptspeicher nicht vorhandene Seite zu erreichen, führt über Alarm in das Betriebssystem, das nachlädt (demand fetching). Evtl. Verdrängung einer Seite: dazu Gebrauchsinformation in Tafel, z.B. LRU (least recently used).
Gleiches Verfahren bei (selten!) Segmenten möglich.

Beispiel zur Seitenadressierung bei sehr großem Adressraum: Sun Ultra SPARC III

Prozessadressen (virtual addresses): von 64b 44b genutzt.
Maschinenadressen (physical addresses): 41b
4 Seitengrößen einstellbar im Bereich 8 KB bis 4 MB, also z.B.
8KB: 2^{31} Seiten, 2^{28} Kacheln (maximal)
4MB: 2^{22} Seiten, 2^{19} Kacheln (maximal)

Je ein Translation Look-Aside Buffer TLB für Befehle und Daten, à 64 Einträge, für jeden mit Prozess-Nummer (TLB muss bei Prozesswechsel nicht gelöscht werden). TLB ist Spitze einer Hierarchie von Seiten-Kacheltabellen. Nächst tiefere Schicht ist ein Cache (Translation Storage Buffer) mit direkter Abbildung, d.h. ein Teil der Seitennummer identifiziert genau eine Zelle, wo die volle Seiten-Kachel-Information liegen kann (aber möglicherweise auch eine andere!). Das Betriebssystem führt im Speicher/auf Platte ein vollständiges Verzeichnis der genutzten Seiten-Kachel-Entsprechungen.

2.4.5 Unabhängige Adressräume = Segmente

Forderung VII: Unabhängige Teiladressräume

Bereits kennen gelernt: Binden

Die genannte Lösung durch Adresstransformation ist nur brauchbar, wenn

- vor Lauf alle Moduln nach Anzahl und Platzbedarf (bis auf einen, den letzten) bekannt sind
- die einzelnen Moduln nicht verschiedene Zugriffsart und Privilegierung untereinander haben.

Schön wäre es, wenn

- kein Binden erforderlich, Adressanpassung erst zur Laufzeit

und

- Überwachung des Zugriffs oder Einsprungs auf fremde Segmente automatisch zur Laufzeit erfolgte.

D.h. der Prozessor soll Adressen verschiedener Prozessadressenräume auf Maschinenadressraum abbilden!

Segmentweise Adressierung

Der Prozessadressenraum besteht zur Laufzeit aus n unabhängigen Teilräumen, relativ zu Null adressiert, entsprechend n Moduln, mit kontrolliertem Übergang: Segmente. Größe, Zugriffsart/Rechte, Gebrauchshäufigkeit i.a. segmentindividuell. Befehle enthalten nur segmentrelative Adressen, Übergang in anderes Segment, i.a. explizit notwendig:
Abbildung auf Maschinenadressen:

- Segmenttafel enthält zu jeder Segmentnummer (verdeckt) Basisadresse, Lagerklasse, Zugriffsart, Größe oder
- Segmenttafel enthält statt Basisadresse Verweis auf segmenteigene Seitenkacheltafel.

Erinnerung: Aktueller Ausschnitt kann auch (und besser) Segmentuntermenge sein!

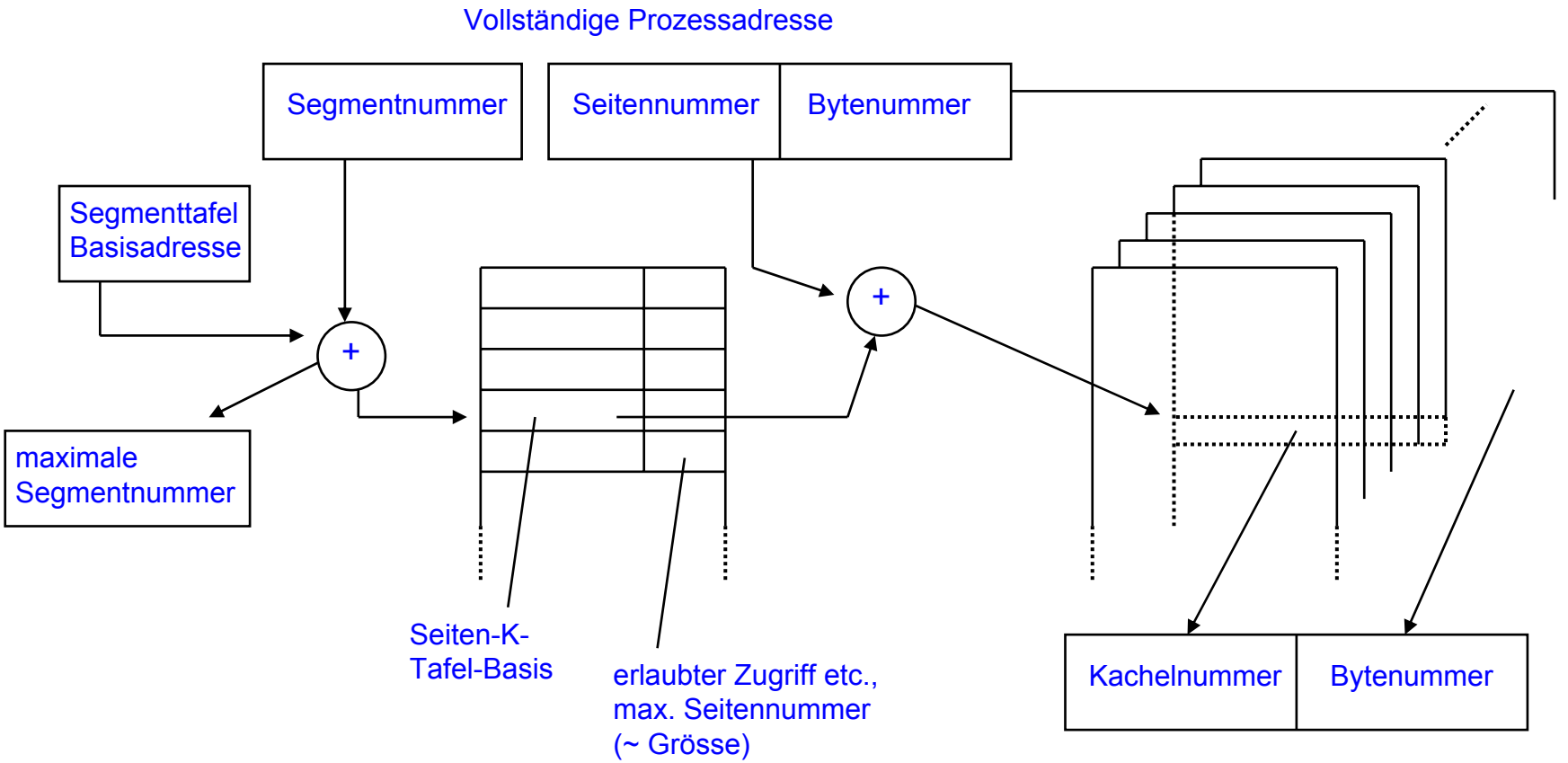
Gleichzeitig können mehrere Segmente implizit vereinbart sein:
z.B. Code Segment, Stack Segment, Data Segment.

Der Wechsel solcher Segmente muss dann explizit genannt werden, z.B. mit Sonderbefehl, Nachladen Segmentregister (→ Pentium).

Die Bezeichnung Segment ist geduldig und geliebt. Beispiele für anderen Gebrauch:

- Die Objekte im Overlay-Baum.
- „Lineare“ Segmentierung (IBM/360 usf. …): Die ersten Adressbits sind Segmentnummer. Durch Zählen (Befehlsadresse), Modifikation kann das Segment unkontrolliert gewechselt werden.
- „Segmente“, die auf offener Basisadressierung beruhen oder auf anderen Basisadressen, ohne dass Nicht-Überdeckung der Segmente gesichert ist (z.B. Intel 8086).

Segmentweise Adressierung mit segmentindividueller Seitenadressierung:



Auch hierzu assoziative Register mit den aktuellen Segmentnummer-Seitennummer-Kachelnummer-Entsprechungen; Tafeln u.U. teilweise im Hauptspeicher, teilweise auf Platte.

Vorteil der Segment/Seiten-Adressierung:

- gestreute Anordnung

- Ausschnitte aus großen Segmenten

Beispiel: Segmentierung und Schutzmechanismen in Pentium

Vorgeschichte:

8080-CPU (1974) hatte 16b (64KB) Adressraum. Mit Übergang auf 8086 (1978) Erweiterung auf 20b (1MB) Maschinenadressraum, bei Aufrechterhaltung der 16b-Programmadressen. Benutzung von 3 implizit bei Speicherzugriff verwendeten Segmenten (Code, Stack, Data), deren Basisadressen vom Prozess frei wählbar in Segment-register speicherbar waren (damit Ausweitung auf 20b Adresse). Dazu ein weiteres Segment (Extra Data). Segmentgrößen 64KB.
Aber keine Schutzmechanismen, kein Demand Fetching!
Bei Übergang in 80286 (1982) Schutzmechanismen (etwa wie noch heute im Pentium).
Bei Übergang in 80386 (1985) zusätzlich Seitenadressierung, Segmentgröße bis 4GB. Ist im Wesentlichen so im Pentium erhalten.

Pentium III:

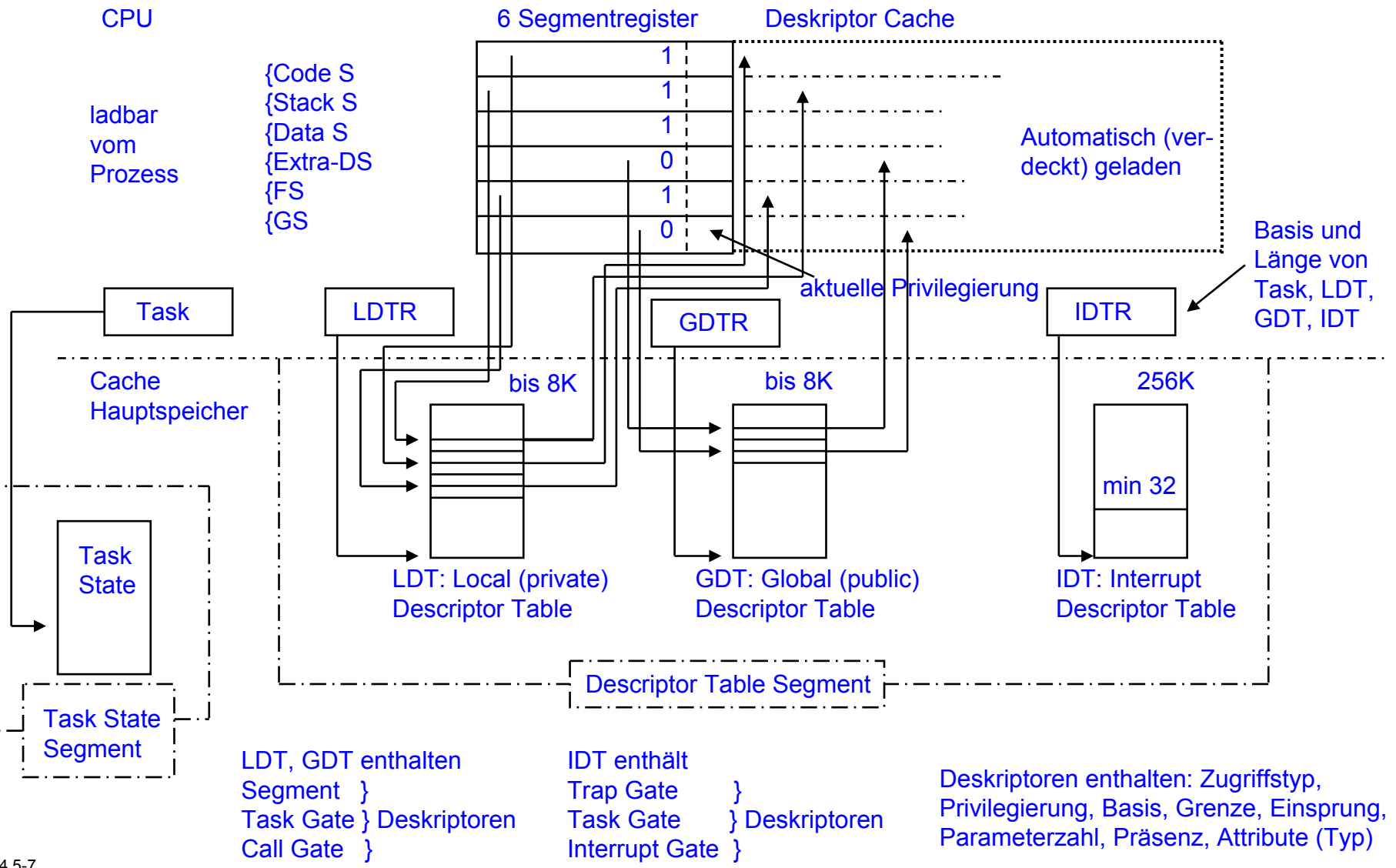
Ausweitung auf 6 aktuell verfügbare Segmente (Segmentregister).
Zugehörige Segmentdeskriptoren aus
- Local Descriptor Table (≤ 8K Deskriptoren), privates Segmentverzeichnis des Prozesses
- Global Descriptor Table (≤ 8K Deskriptoren), öffentliches Segmentverzeichnis, hierin auch Betriebssystem!

Segmentdeskriptoren enthalten Attribute, Zugriffstyp, Privilegierung (Zugriff nur auf gleich oder minder privilegierten Code (ausgenommen: Programmbibliothek) oder minder privilegierte Daten), Basis, Grenze, Präsenz im Speicher.
Daneben über dieselbe Tafel Task Gate/Call Gate Deskriptoren als Übergang in andere Task/in höher privilegierten Code; dabei Schutzmaßnahmen:

- Einsprung nur an erlaubter Stelle
- Mitgebrachte Adressen werden überprüft, ob im Zugriffsbereich des Aufrufers liegend.

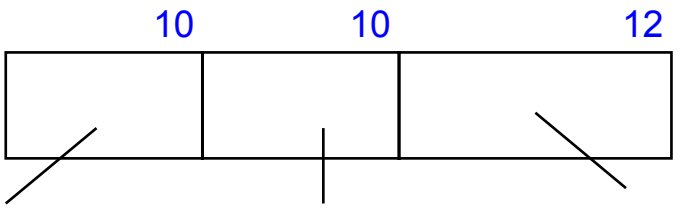
Ähnlicher Mechanismus für Unterbrechungen.

Zugang zu den Segment/Task Gate/CallGate/Trap Gate/Interrupt Gate Deskriptoren:



Seitenadressierung:

Über die Basisadresse wird die Prozessadresse in eine 32b (wenig!) „lineare“ Adresse umgesetzt. Diese wird zweistufig umgesetzt:



Index im Page Directory, ≤ 1024 Verweise auf Seitentafeln

Index in Seiten-tafel (≤ 1024 Kachelnummern)

Byte in der 4KB-Seite

Durch mehrfache Page Directories sind größere Adress-räume als 2^{32} B beherrschbar!

Damit soll (angesichts der Unternutzung des 2^{32} B-Adressraums) verhindert werden, dass eine Tafel aus $2^{10+10} \approx 10^6$ Einträgen zu führen ist.

Vereinfachte Lösung: Alle Segmente im selben linearen Adress-raum, Segment hat eine 1024-Seitentafel.

2.4.6 Zugriffsschutz im Hauptspeicher

Manchmal „Speicherschutz“ genannt. Zu regeln ist: Welcher Prozess? darf worauf? (Segment, Seite, Datenstruktur, Code) wie? (lesend, ausführend, schreibend) zugreifen?

Lösungen:

- Bei einfacher verdeckter Basisadresse: Zugriffsschutz durch Vergleich mit Höchstadresse (Was? Nicht: Wie?)
- Bei Seitenadressierung: „Wer? Worauf?“ Geregelt durch Kachelnummer in Seitenkacheltafel, „wie?“ durch Hilfsdaten. Vgl. 2.4.4-9
- Bei Segmentierung: Segment ist natürliche Schutzeinheit! „Wer? Worauf?“ geregelt durch Segmenttafel und Seitenkacheltafeln oder verdeckte Basisadressen/Höchstadressen, „wie?“ geregelt durch Hilfsdaten in Segmenttafel.

2.5 Unterbrechungen (Interrupts, Exceptions)

Es ist erforderlich, den Interpretationsprozess des Prozessors mit anderen Prozessen zu koordinieren und zu synchronisieren, die mehr oder weniger asynchron verlaufen. Das Mittel dazu ist Unterbrechung.

Grundsätzlicher Ablauf der Unterbrechung:

Unterbrechungswunsch → akzeptiert → Unterbrechung → Prozess-Status definiert retten (wegen Verschachtelung und Last In - First Out-Folge, typisch unter Benutzung eines Stacks) → Unterbrechungsbehandlung anspringen, abarbeiten (oft geteilt: unaufschiebbare Anfangsbehandlung, aufschiebbare eigentliche Behandlung) → Wiedereinsetzen eines Prozess-Status

Beispiele für Prozesse, die asynchron zum Interpretationsprozess sind:

a. Prozess hardwaretechnischer Fehler (Klima, Spannung, ..)
b. Uhrprozess
c. Interpretationsprozesse anderer Prozessoren
d. Prozesse in peripheren Geräten
e. Benutzer-/Operateurverhalten
f. Externe technische Prozesse

Ein Unterbrechungswunsch, der auf a ... f zurückgeht, betrifft i.a. nicht den laufenden Prozess! Oft Interrupt Request im engeren Sinne genannt. Die Behandlung der durch diese ausgelösten Unterbrechungen kann nicht vom aktuellen Prozess, sondern nur vom Betriebssystem erfolgen, Nachricht an etwa betroffenen Prozess.

Daneben gibt es Wünsche, die auf die Erledigung des aktuellen Befehls zurückgehen und eine Ausnahmebehandlung erfordern, die dasselbe Verfahren benutzt (Status retten, Behandlung durch Programm …). Sie sind also nicht eigentlich asynchron; oft exceptions genannt:

g. Adressierungsfehler
h. Operandenfehler (z.B. durch Division durch Null, Typfehler)
i. Operatorfehler (z.B. Stop verboten)
j. Operationsfehler (z.B. Überlauf)
k. Moduswechsel
   hier liegt natürlich gar keine Ausnahme vor, da der Befehl den Wechsel verlangt! Aber Ablauf wie Unterbrechung gewünscht.

Die Behandlung kann in einigen Fällen (h, j) vom Prozess selbst gemacht werden (z.B. Laufzeitsystem des Übersetzers).

Varianten des Ablaufs:

- Abbruch des laufenden Programms
  -- nach einem Befehl (die Regel)
  -- in einem Befehl (notwendig bei Nichtbeendbarkeit, z.B. Seitenalarm; empfehlenswert bei lang laufender Operation, z.B. Wortgruppen-transport).
  Lösung:
  * Rücksetzung vor den Befehl (kann man Status restaurieren? Wirkung zurücksetzen?)
  * Wiederaufnahme in der Operation (dann vermehrter (Mikro-)Status!)

- Verzweigung
  -- auf eine einzige Zieladresse für alle Unterbrechungsursachen, dort folgt Analyse (billige Hardware, langsam)
  -- Unterbrechungstyp bestimmt Zieladresse z.B. über Tafel im Speicher (vector table).

Schließlich gibt es noch weitere Unterbrechungseinleitungen, die vom aktuellen Programm ausgelöst werden:

l. Software-Interrupts (programmierte Unterbrechungen): Das sind Befehle, die bestimmte Unterbrechungswünsche setzen können (z.B. bei ausgebliebenem „Hardware“-Unterbrechungswunsch, oder zu Testzwecken)

m. Debugging-Traps (Entwanzungs-Fallen): Ist für eine Befehlsfolge ein Merkbit im Prozessor gesetzt, dann wird nach jedem Befehl in eine Analyseroutine verzweigt: Überwachung ohne Änderung der Befehlsfolge.

Sind nur Unterbrechungen an bestimmten Stellen gewünscht, kann man spezielle Unterbrechungen durch Befehle auslösen (break-Befehle, auch bedingbar, debug mode).

Erste Komplizierung:

Durch unabhängiges Auftreten der Wünsche a .. f und (bei nicht überlappender CPU nur einem) der Klasse g .. m: Es gibt i.a. viele anstehende Unterbrechungswünsche zu einem Zeitpunkt.

Zwei Grundtechniken:

- Priorisierung der Unterbrechungswünsche, statisch nach Herkunft; im Prozessor ablaufender Prozess erhält ebenfalls eine Prioritätsstufe. Höchstpriorisierter (Prozess, Unterbrechungswunsch) erhält Prozessor.
- Absichtliche „Maskierung“ von Unterbrechungswünschen (global: Unterbrechungssperre).

Oft kombiniert.

Zweite Komplizierung:

Wenn der Prozessor die Befehle nicht seriell abarbeitet, dann kann ein Befehl einen Unterbrechungswunsch liefern, während der nach der Befehlsfolge präzedente Befehl, der seinerseits einen Unterbrechungswunsch liefern wird, noch nicht abgearbeitet ist (-> Kap. 2.10)

Beispiel: Intel Pentium

Interrupt: Extern und asynchron

- Maskable: (Pauschal) unterdrückbar durch Löschen des Interrupt Enable-Flipflops
- Non-Maskable: (NMI): Wird unbedingt nach nächstem Operationsende aufgenommen; während seiner Behandlung schiebt der Prozessor weitere NMI auf.

Exception: Intern und synchron

- Fault: Auslösende Operation wird rückgesetzt (!), z.B. Seitenfehler
- Trap: Auslösende Operation wird vollendet, z.B. programmierte Unterbrechung
- Abort: Programm wird abgesetzt, z.B. Doppel-Exception (gewisse Fälle)

Übersicht über Unterbrechungswünsche:
Non-Maskable Interrupts und Exceptions:

Index	Anlass	Behandlung
0	Division	Fault
1	Debug	Trap oder Fault
2	NMI	
3	Breakpoint	Trap
4	Overflow	Trap
5	Bound	Fault
6	Invalid Opcode	Fault
7	Device not available	Fault
8	Double fault	Abort
9	(Reserved)	
10	Invalid Task State	Fault
11	Segment not present	Fault
12	Stack fault	Fault
13	General protection	Fault oder Trap
14	Page not present	Fault
15	(Reserved)	
16	Floating Point	Fault
17	Alignment	Fault
18	Machine check	
19-31	(Reserved)	
Maskable Interrupts		Trap
32-255	(Maskable Interrupts)	
	(Int n-Befehl)	

Interrupt Descriptor Table IDT

(vgl. 2.4.5-7)

Der Interrupt-Vektor (Index) (hier im allgemeinen Sinn: D.h. interrupt oder exception) bezeichnet einen 8B-Descriptor in der Interrupt Descriptor Table:

- Segment Selector (16b)
- Offset (32b)
- Privilege Level
- Segment Present

Die IDT kann irgendwo im Speicher liegen (Basis-Adresse und Länge steht im IDTR-Register).

2.6 Rechenwerk und arithmetische Operationen

Hier nur ein ganz roher Überblick. Zahldarstellung und arithmetische Operationen sind ein Teilgebiet der Numerik/ Rechnerarchitektur, das seit Jahrzehnten gut stabilisiert und standardisiert ist! Vgl. etwa Goldberg, D.: Computer Arithmetic, in [Hennessy Patterson 2007].

2.6.1 Rechenwerk

Rechenwerk (arithmetic, data path) ist der Teil des Prozessors, der die durch das Programm explizit bestimmten Berechnungen ausführt (d.h. Adressrechnung z.B. oft nicht im Rechenwerk).

Besteht aus:

- Registern für Operanden/Ergebnisse, meist nach Festpunkt/ Gleitpunkt getrennt
- Rechen-Schaltnetzen mit Hilfsregistern (Shiftzähler, Multiplikator/ Quotientenregister, Addenden/ Multiplikanden Registern, …). Diese arithmetisch/logische Einheit (ALU) kann als Pipeline und/oder funktionsspezifische Werke ausgebildet sein
- Einrichtungen für die Operationensteuerung

Bereits bekannt: Registerstruktur, Abarbeitung zweistelliger Verknüpfungen, Rechenwerksbefehle

2.6.2 Festpunkt- und Gleitpunktzahlen

Radix-(Polyadische) Darstellung:

$a_m\ a_{m-1} \ldots a_i \ldots a_1$

ist eine Zahl mit Wert

$$a = \sum_{i=1}^{m} a_i * B^{i-1} * B^k$$

dabei

a_i ε [0 .. B-1] Ziffern

B ganz, ≥ 2, Basis

B^k Skalenfaktor, üblich

k=0: B-al-Punkt rechts neben a_1: a ist ganze Zahl

k=-m: B-al-Punkt links neben a_m: a ist echter B-al-Bruch

$a_m\ a_{m-1}$ .. a_i .. a_1 mit festem k ist Festpunktzahl. In Rechnern typisch k=0 oder (seltener) k=-m. Punkt nicht notiert. Für Addition/Subtraktion k ohne Bedeutung.

Mängel der Festpunktdarstellung:

- Bereich: Differenz zwischen betragsgrößter und betragskleinster Zahl ist zu klein: B^m-1 bei Festpunkt
- Genauigkeit (Zahl der korrekten Ziffern (einschließlich korrekt gerundeter) ohne führende oder nachfolgende Nullen): ist im Bereich kleiner Beträge für eine gute Annäherung an beliebige reelle Zahlen zu klein.

Daher Gleitpunkt (integrierter logarithmischer „Skalenfaktor“).

Typische Gleitpunktdarstellung:
v c p
v: Vorzeichen: 1 Bit, übliche Konvention
c: Charakteristik: $c_s\ c_{s-1} \ldots c_1$, aus dieser ist Exponent x berechenbar
p: Mantisse: $p_r\ p_{r-1} \ldots p_1$

Und hat z.B. den Wert

$$a = \underbrace{\left(\sum_{i=1}^{r} p_i * B^{i-1}\right)}_{\text{Mantisse}} * B^{-r} * G^x$$

G ist Basis der Gleitpunktdarstellung.

$$a = \left(\sum_{i=1}^{r} p_i * B^{i-1}\right) * B^{-r} * G^{x}$$

Vorteil der Gleitpunktdarstellung:

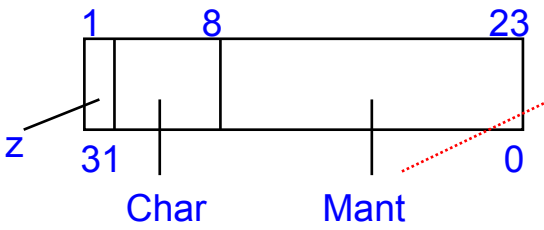
- Bereich kann wesentlich größer als B^m-1 (hier m=r+s, Zahl der Ziffern) gemacht werden
- Genauigkeit im ganzen Bereich annähernd konstant.

Abbild in höheren Sprachen
irreführend: real

In Rechnern aus technischen Gründen meist B=2, selten B=10 (nur Festpunkt, dann a_i trotzdem binär codiert), G=2 oder G=16.

IEEE 754 Gleitpunkt-Standard mit Definition von Rundung und exceptions:

Einfaches Format:



Mantisse

Exponent

d.h. Implizite führende 1

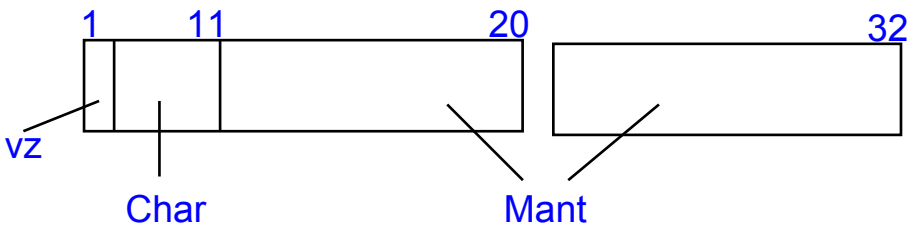
Wert = $(-1)^{vz} * 2^{Char-127} * (1. Mant)$

dabei

(Char=255) und
- Mant ≠ 0: Not a Number
- Mant = 0: ∞

Alle Stellen Null: 0

Doppeltes Format:



2.6.3 Komplementdarstellung und verallgemeinerte Addition von Festpunktzahlen

Kurzfassung:

Festpunktzahlen: Positiv: m-stellige Dualzahl mit Vorzeichen (0). Negativ: Vorzeichen (1) und Komplement zu

2^m-1: Stellen-(Einer-)Komplement, einfach bildbar, 2 Nullen, nach Addition Korrektur, falls Übertrag in Stelle m+2 („Einerrücklauf“)

2^m: Bereichs-(Zweier-)Komplement, unsymmetrischer Darstellungsbereich -2^m .. 2^m-1. Heute üblich.

Verallgemeinerte Addition wird korrekt durch Rechnung modulo 2^{m+1} ausgeführt, dabei

- Überlauf falls (++) → - oder (--) → +
- Einerrücklauf bei Stellenkomplement.

Die Bit-parallele Addition verlangt Grundschritt und im Mittel ld m Übertragsschritte; Spezialnetze für schnellen Übertragsabbau unbedingt erforderlich!

2.6.4 Multiplikation und Division von Festpunktzahlen

Algorithmus etwa wie auf der Schule gelernt. Für die Beschleunigung wichtig: Multiplikationstafel (z.B. für Bereich [1 .. 15] x [1 .. 15])

Multiplikation:

Liefert doppelt langes Ergebnis! Zur Beschleunigung noch wichtig: bei Aufbau des Produktes als Folge von Additionsschritten und Shifts werden eventuell nicht abgebaute Übertragsreste im nächsten Schritt mitverarbeitet.

Division:

Zur Prüfung, wie oft der Divisor im Rest des Dividenden enthalten ist, dient ebenso Multiplikationstafel, außerdem versuchsweise Subtraktion; auch hier kann man aufeinanderfolgende Schritte teilweise verschmelzen. Auch Division p:q liefert doppelt langes Ergebnis: p div q, p mod q. Wurzelziehen folgt einem divisionsartigen Algorithmus.

2.6.5 Gleitpunktoperationen

Gleitpunktzahl aus Vorzeichen Charakteristik Mantisse!

Um

- die Genauigkeit voll zu nutzen
- eine eindeutige Darstellung zu sichern

werden Gleitpunktzahlen in der Maschine normalisiert geführt, d.h.

- bei Gleitpunktbasis 2 ohne führende Null in der Mantisse; d.h. das Bitfeld wird linksbündig verschoben; bei IEEE 754 ohne die implizite 1 vor dem Dualpunkt darzustellen
- bei Gleitpunktbasis 16 mit höchstens 3 führenden Nullen.

Addition /Subtraktion:

Sind die Charakteristiken ungleich, so wird die kleinere Charakteristik angepasst (=> führende Nullen in Mantisse), dann Addition/Subtraktion, erforderlichenfalls Normalisierung

Multiplikation/Division:

Die Charakteristiken werden addiert/subtrahiert und um Betrag des kleinsten Exponenten korrigiert; die Mantissen werden multipliziert/ dividiert, erforderlichenfalls Normalisierung

Bei Gleitpunktoperationen kann das assoziative und das distributive Gesetz verletzt werden!

2.7 Spezialprozessoren

2.7.1 Eingebettete Rechner (Microcontrollers)

Vergleiche auch 1.6!

Für Realzeitaufgaben in technischer Umgebung (Sensoren, Aktoren) spezialisierte Rechner, meist CPU und Speicher auf einem Chip.

Speicher oft aufgeteilt („Harvard-Modell“) in nicht beschreibbaren Programmspeicher und Datenspeicher.

Nach Anzahl viel wichtiger als konventionelle Prozessoren. Bei großer Stückzahl sehr billig (bis ca. 1 Euro).

Einfachste Betriebsformen; Anwendungssoftware (unveränderlich) und rudimentäres Betriebssystem.

Wichtige Funktionen: Signalwandler, Timer, Kommunikation. Einfacher Befehlssatz.

2.7.2 Vektorprozessoren

Viele Berechnungen fußen auf Arrays als wesentliche Datenstruktur, mit nebenläufigen Operationen auf deren Komponenten, insbesondere bei der numerischen Lösung von partiellen Differentialgleichungen (verbreitet „Simulation“ genannt). Größte Bedeutung bei der Berechnung von raum/zeitlichen Vorgängen im Kontinuum, darstellbar als Funktion mehrerer Veränderlicher; etwa:

- Strömungen, Diffusion, Wärmeausbreitung
- Chemische Prozesse (Industrie, Biologie), Verbrennung, Explosion, astrophysikalische Vorgänge
- Mehrdimensionale Elastizität und Schwingungen
- Wellenausbreitung.

Wesentliches Werkzeug der Natur- und Ingenieurwissenschaften.

Zwei Paradigmen der Berechnung:

Das mehrdimensionale Kontinuum wird diskretisiert und liefert damit ein mehrdimensionales Gitter, das

- entweder auf die Rechner/Prozessoren eines Parallelrechners abgebildet wird, die den Funktionswert unter (möglichst geringer) Kommunikation mit den Nachbarzellen berechnen: Ausführung auf einen Parallelrechner; dabei oft das gleiche Programm auf vielen gleichartigen Daten: SPMD (Single Program-Multiple Data)

- oder das Gitter wird als Array dargestellt, und die Komponenten werden in vielen Durchgängen über das Array seriell berechnet: SPMD oder eher SIMD (Single Instruction-Multiple Data): Operationen auf den Vektoren des Arrays: Vektorrechner.

Grundelement eines Vektorrechners ist ein (oder mehrere) Pipeline-Rechenwerk. Da die Operationen auf den Vektorkomponenten (fast) nebenläufig sind und die Dimension der Vektoren oft sehr hoch (10^2 .. 10^4) ist, erreicht man ausgezeichnete Wirkungsgrade der Pipeline. Die Operanden werden dabei aus Vektorregistern entnommen, und die Ergebnisse dort abgelegt; wegen der Gleichmäßigkeit und Vorhersagbarkeit der Operanden/Ergebnisströme ist auch direkte Kommunikation mit dem Hauptspeicher (hoch verschränkt, vgl. Kapitel 3) möglich.

Vektorrechner gehen besonders elegant mit dem Problem der fallabhängigen Bearbeitung einer Array-Komponente um: nicht mit datenabhängigem Sprung (wie bei SISD-Prozessoren), sondern unter Benutzung von Maskenvektoren.

Das gesamte Array wird komponentenweise getestet, welche von zwei Operationen (abhängig vom Datenwert) anzuwenden ist. Das Ergebnis gibt ein binäres Masken-Array. Im nächsten Durchgang werden alle Komponenten, gesteuert durch das Maskenarray, der einen, danach -weiterer Durchgang- der alternativen Operation unterzogen.

Neben den genannten Aufgaben gibt es weitere, bei denen viele Datensätze gleichartig zu verarbeiten sind (z.B. stochastisch unabhängige Simulationsläufe).

Vektorrechner waren bis etwa 1990 das Paradigma für Hochleistungsrechner. Ihre Parallelität ist durch die Gitterstruktur begrenzt. Parallelrechner sind flexibler einsetzbar (vgl. die Partitionen der SGI Altix) und nicht auf die SIMD/SPMD-Parallelität begrenzt.

2.7.3 Graphikprozessoren

Einige wichtige Verarbeitungsstrukturen sind dadurch gekennzeichnet, dass ein langer Strom von Daten durch eine Folge von Verarbeitungsschritten geführt wird, die programmiert sein können (dann als „MISD“ Multiple Instruction-Single Data auffassbar).

Beispiel: Graphikprozessor

Aufgabe ist die Generierung eines realistischen Bildes (bei Animation: einer Bildfolge) aus einer symbolischen Beschreibung. Schrittweise wird über eine Konfiguration von Komponenten (auch z.B. menschlicher Gliedmaßen in bestimmten Stellungen) ein abstraktes räumliches Modell (als Menge der Eckpunkte von Dreiecken, die Oberflächen darstellen) hergestellt, ggf. geglättet, der Blickwinkel festgelegt, damit verdeckte Bildteile entfernt, die Oberflächenstruktur und Farbe eingebracht, Beleuchtung und Reflexion realisiert. Am Ende ergibt sich das „Frame“ bzw. eine Folge von „Frames“. Große Bedeutung u.a. in Spielen und Filmen.

Die Verarbeitungsstruktur besteht in einem Satz von spezialisierten, oft programmierbaren Werken, durch die das entstehende Bild als Strom von Elementen (am Ende von Bildpunkten, Pixels) geleitet wird. Die Bildelemente sind weitgehend unabhängig bearbeitbar, so dass in den Werken SIMD-Parallelität nutzbar ist.

(Fast) kein Problem mit Hauptspeicherzugriffen, Cachelokalität, bedingten Sprüngen usw. Die Werke werden auf der Chipoberfläche bereitgestellt und programmgesteuert miteinander verschaltet, um dem Strom von Elementen den Weg zu definieren.

2.7.4 Digitale Signalprozessoren

Ein Signal ist eine physikalische Größe, die dem Datentransport dient, z.B. Strom/Spannung, Licht, Luftschall. Signalprozessoren gibt es schon 80 Jahre, z.B. in der Telefonie, nur waren sie „analog“, d.h. aufgebaut aus (oft rückgekoppelten) Schaltungen aus Verstärkern, Widerständen, Kondensatoren, Spulen sorgten sie für die erwünschte Form von Signalverläufen. Aber diese Komponenten erlauben nur einfache Funktionen, und Spulen sind nicht in Halbleiterschaltungen darstellbar.

Digitale Signalprozessoren modifizieren das durchlaufende digitalisierte Signal in Realzeit. Strukturell große Ähnlichkeit mit Graphikprozessoren, aber andere Werke.

Anwendungen von Signalprozessoren (Vorlesung Prof. Gerndt):

- Mobiltelefonie, Modems, Audioeffekte, Sprachanalyse
- Medizin (EEG, EKG, Ultraschall), ABS, Radar, Sonar.

Signalanalyse und –Modifikation ist „im Zeitbereich“ (Zeitverlauf) und „im Frequenzbereich“ äquivalent möglich, oftmals sehr verschieden effizient. Daher Übergang zwischen beiden Bereichen (durch „Fourier-Transformation“) wichtige Operation.

2.8 Steuerung der Mikromaschine

Die Mikromaschine, die die Mikroarchitektur realisiert und damit

- der ISA-Maschine (Instruction Set Architecture) einen leistungsfähigen und zuverlässigen Unterbau liefert und
- die ISA-Operationen und –Datenstrukturen auf elementare Schaltnetze und Schaltwerke abbildet,

ist schon in 2.0 charakterisiert worden. Dort ist auch Mikroprogrammierung als eine (augenblicklich wenig aktuelle) Steuerungstechnik für die Mikromaschine genannt worden.
Bei RISC-Prozessoren liegen die Operationen und Datenstrukturen der ISA bereits nahe an denen der Mikroarchitektur.

Die Abbildung der Elemente der ISA auf Elemente der Mikroarchitektur ist wiederum ein Beispiel für 1.4.3 (Sprachschichten und Auftragsübergänge) und 1.11.2 (Bindungszeitpunkt). Der Prozessor benutzt Interpretation, Übersetzung (im Kontext eines Befehls oder mehrerer Befehle) und (Fast-)Direktausführung. Interpretation (unglücklicherweise Mikroprogrammierung genannt) lohnt (immer noch) einen aufmerksamen Blick:

- „vertikale“ Mikroprogrammierung: ähnlich ISA-Ebene; Befehlszähler, Sprünge, elementarer (aber funktional vollständiger) Befehlssatz. Mikroprogramm aus vielen kurzen Befehlen. Langsam, billig
- „horizontale“ Mikroprogrammierung: Mikrobefehle sind Folgezustand und Ausgabe eines Mealy-Automaten; volle Flexibilität beim Einsatz elementarer Hardwarefunktionen. Mikroprogramm aus wenigen, langen Befehlen. Schnell, teuer.

Übergang zur Mikromaschine: Direktausführung, Interpretation, Übersetzung

CPU	Mikro-programmiert	RISC	Pentium, Athlon	Transmeta Crusoe
Befehlssatz	Komplex, z.B. x86 oder IBM Mainframes	gleichartig struk-turierte Befehls-abläufe, z.B. SPARC, Power	x86	x86
ISA-Programm wird zur Laufzeit	befehlsweise interpretiert durch Mikroprogramm	direkt ausgeführt durch verteilte Steuerungen	befehlsweise über-setzt in 1..3 „R“ops, diese werden auf RISC ausgeführt	basisblockweise übersetzt und optimiert in „Atome“, EPIC-Ausführung
mehrere Pipelines	schwierig, da Zentralsteuerung	gut beherrsch-bar	gut beherrsch-bar	gut beherrsch-bar, aber (1 Atom je Pipe-line), starr
Dynamisches Scheduling der Operationen	nein	nach Präzedenz und Datenfluss, bis etwa 40 Operationen	nach Präzedenz und Datenfluss, bis etwa 40 Operationen	„Atome“ an „ihre“ Pipeline gebunden
Besonderheiten	Funktionale Ände-rungen u. Ausführung fremder Befehlssätze leicht möglich. Leistungsschwach	unkomplex, leistungsstark	bereits durch-laufene Befehls-folgen werden im Cache 1 übersetzt bereit gehalten	Optimierung. Transmeta Crusoe Pionier der Software/ aktivitätsgesteuerten Energieeinsparung

2.9 Cache Memories

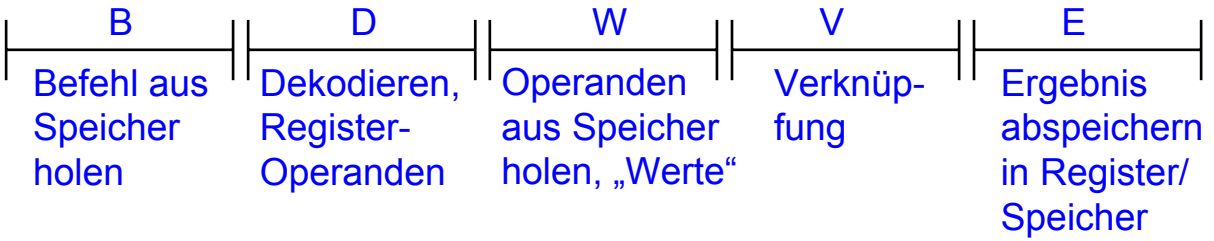
2.9.1 Noch mal grundsätzliches: Grenzdurchsatz der Zentraleinheit

Zentraleinheit: Prozessor und Hauptspeicher

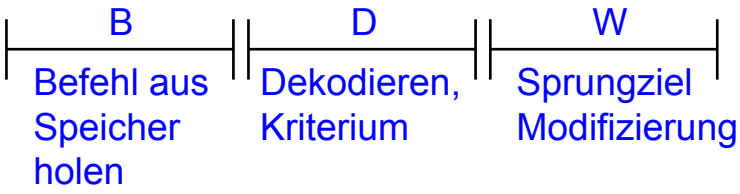
Im einfachsten Fall arbeitet ein Prozessor die Befehle seriell ab. $\overline{b}$ sei die mittlere Bearbeitungszeit je Befehl. Der Grenzdurchsatz der Zentraleinheit ist:

$$c = \frac{1}{\overline{b}} \quad \text{Befehle / Zeiteinheit} \qquad \text{z.B. „Mips“}$$

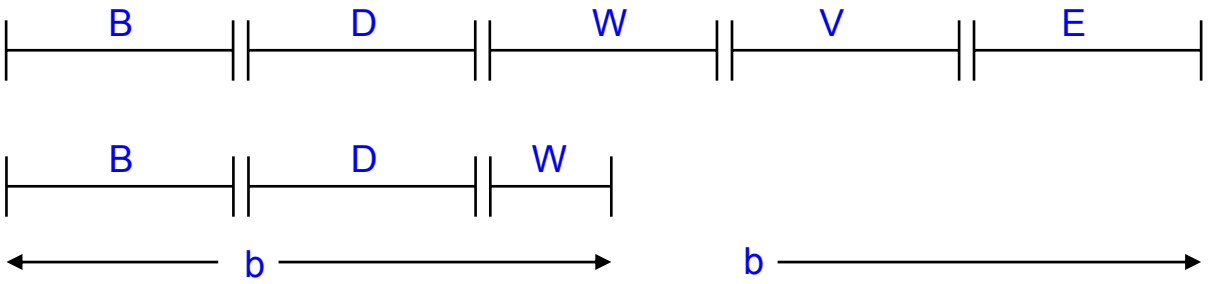
Zeitliche Gliederung von Operationen (etwas grob):



Verarbeitungsbefehl



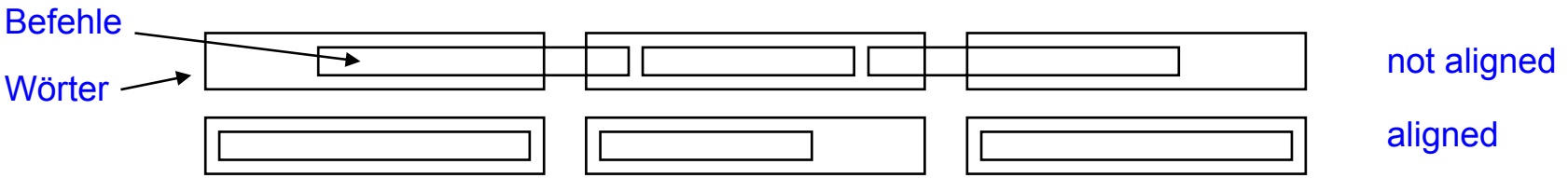
Sprungbefehl



Zahl der Zugriffe:

Phase B:

Zahl hängt ab von Verhältnis Befehlslänge: Wortlänge (im Speicher/Bus), z.B. ½ … 4 /8 Byte-Wortlänge), außerdem von „alignment“, d.h. Ausrichtung der Befehlsanfänge auf Wortanfänge



und noch davon, ob es für die Seitenadressierung zu einem extra Speicherzugriff kommt.

Phase W und E:

Zahl hängt von Zahl und Länge der Operanden ab, außerdem von Adressersetzungen und den für Phase B genannten Einflussgrößen.

Typisch sind $\Sigma \in \{1; 2\}$ Speicherzugriffe erforderlich, dank der Register.

Σ z.B. 1,5 Speicherzugriffe/Befehl
Das heißt: machen wir den Prozessor intern auch noch so schnell, nie kommen wir über:

$$C \leq \frac{1}{\Sigma * T_{ZugriffHSP}}$$

weg.

Dabei ist: $T_{ZugriffHSP} = T_{Bus} + T_{HS} \approx 20ns + 40ns$

d.h

$$C \leq \frac{1}{1{,}5 * 60ns} \approx 11 \text{ Mips}$$

Faktisch viel weniger, weil D und V eben auch Zeit brauchen!
Aber es gibt zwei Hilfsmittel:

Cache Memory und Parallelisierung, z.B. Pipelining

Beide kombinierbar, beide unterbrechungssensibel, beide verlangen meist, dass der Hauptspeicher viele Lese/Schreibvorgänge gleichzeitig abwickeln kann.

2.9.2 Cache Memories: Konfiguration, Zugriffszeit und Durchsatz

Cache Memories (vgl. 1.6) bieten dem Prozessor einen Ausschnitt aus dem Hauptspeicherinhalt. Heute werden meistens drei Stufen realisiert:

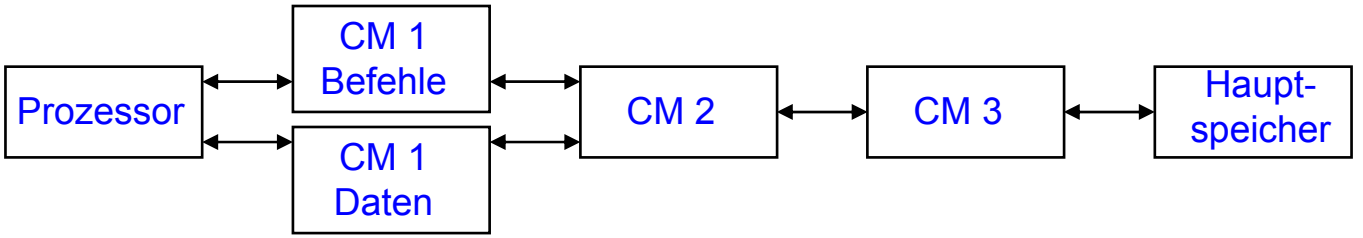
Schicht 1:	$T_{Cache\ 1}$	etwa 3 ns	k_{Cache1}	z.B. 2x 64 KB
Schicht 2:	$T_{Cache\ 2}$	etwa 10 ns	k_{Cache2}	z.B. 2 MB
Schicht 3:	$T_{Cache\ 3}$	etwa 20 ns	k_{Cache3}	z.B. 32 MB
(Hauptspeicher):	T_{HS}	etwa 40 ns	k_{HS}	z.B. 2 GB

Schicht 1 wird immer, Schicht 2 oft auf dem Prozessorchip implementiert. Für die Cache Memories und den Hauptspeicher mit Inhalten I_1, I_2, I_3, I_{HS} gilt eine Inklusionseigenschaft:

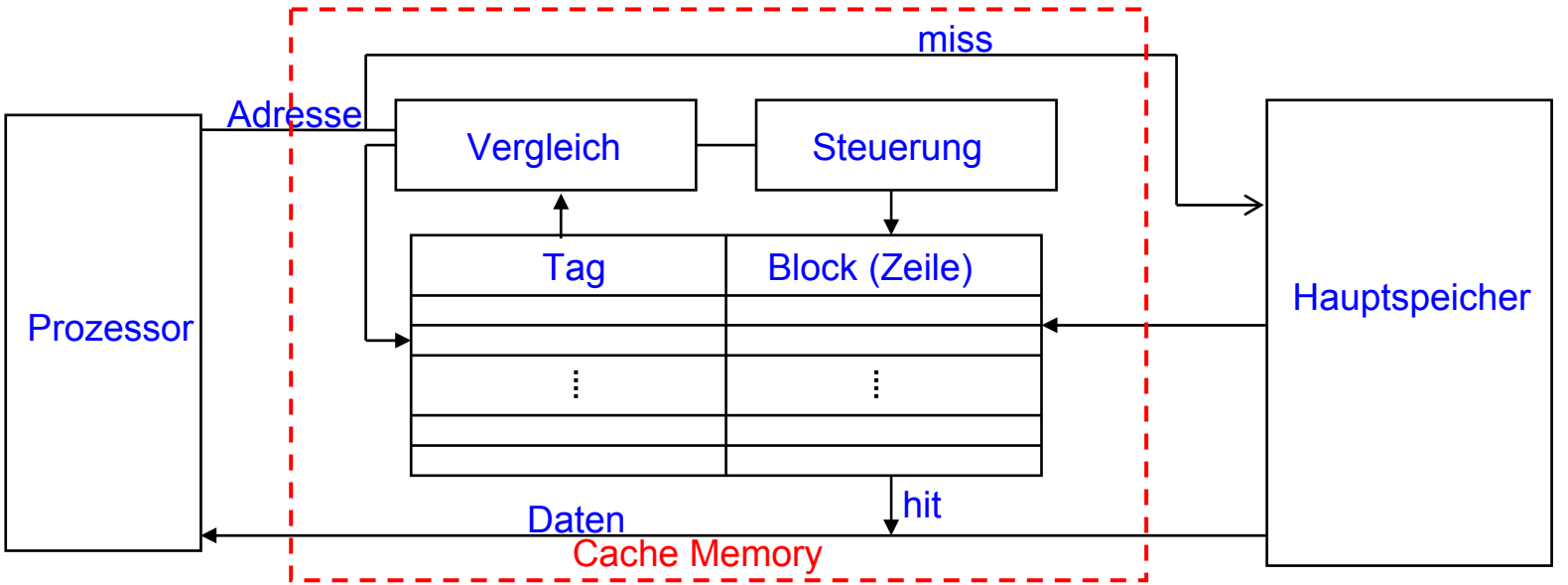
I_1 enthalten in I_2 enthalten in I_3 enthalten in I_{HS}

Cache 1 wird meistens für Befehle und Daten getrennt realisiert.

Also:



und Prozessor und CM 1 (nur eines) allein:



„Tag“ (Marke) ist ganze Adresse: vollassoziatives CM
nur Teil-Adresse, ein Block: direkt abbildendes CM
nur Teil-Adresse, mehrere Blöcke: mengenassoziatives CM

und Tag enthält „Gültigbit“: „Zelle ist nicht leer“.

Ein Cache Memory (CM) ist „transparent“, d.h. es bietet dem Zugreifer dieselbe funktionale Schnittstelle wie der Hauptspeicher, aber mit reduzierter Zugriffszeit T_{SP} zum „Speichersystem“ (aus Cache (s) und Hauptspeicher). An CM1 erreicht man Trefferverhältnisse h (Hit Ratio) von 0,7 .. 0,98. Die resultierende Zugriffszeit ist (vgl. 1.6 mit h=0,95)

$$T_{SP} = 1*T_{CM} + (1-h) * T_{HS,B} = 1 * 3ns + 0,05 * 60ns = 6ns$$

1-h heißt auch Miss Ratio m.
Damit steigt die Schranke für „unseren“ ZE-Grenzdurchsatz auf

$$C_{ZE} \leq \frac{1}{1,5*T_{SP}} = 110 \text{ Mips}$$

(das langt ja noch nicht für unser Produkt!)

Die Durchsatzforderung des Prozessors an Bus und Hauptspeicher ist bei einer Blockgröße BG und einer Wortgröße W, h = 0,95

$d_{HS} = 1{,}5 * C_{ZE} * (1\text{-}h) * BG = 1{,}5 * C_{ZE} * 0{,}05 * 32B = 2{,}4B * C_{ZE}$

Ohne CM wäre

$d_{HS} = 1{,}5 * C_{ZE} * W = 1{,}5 * C_{ZE} * 8B = 12B * C_{ZE}$

Dabei

- haben wir (wie schon früher) angenommen, dass der Prozessor im Mittel 1,5 mal je Operation auf den „Speicher“ zugreift
- haben wir als Wortlänge 8B und als Blockgröße (bei CM sagt man oft „Zeile“ (line) statt Block) 32B angenommen.

Die Ergebnisse sind noch besser, wenn man CM 2 und CM 3 in das schlichte Modell einbezieht (wirkt wie Verkleinerung von T_{SP}).

Unsere Resultate:

- Zugriffszeit von 60 auf 6ns
- obere Schranke des ZE-Grenzdurchsatzes von 11 Mips auf 110 Mips
- Durchsatz am Hauptspeicher von C_{ZE} * 12B auf C_{ZE} * 2,4B

sind unter Berücksichtigung von CM 2 und CM 3 noch etwas besser.

Aber unsere (einfachen) Formeln zeigen auch, dass ein schlechter Wert von h das CM schädlich macht: Wird

$h < \frac{T_{CM}}{T_{HS}}$ (bei unserem Beispiel 0,05), dann ist $T_{SP} > T_{HS,B}$

Ebenso wird

$h < \frac{BG - W}{BG}$ (bei unserem Beispiel $\frac{32-8}{32} = 0{,}75$), dann vergrößert das CM die Last auf Bus und Hauptspeicher.

Kritisch bei Aufbau des „Working Sets“ im Cache (nach Prozess-Start) und bei schlechter Lokalität oder zu kleinem Cache.

Die Anforderungen verschärfen sich durch Parallelverarbeitung im Prozessor:

- in einem Takt kann bereits eine Pipeline einen Befehl und einen Operanden verlangen (daher geteiltes CM 1)

- im Miss-Fall darf das CM nicht abwarten bis der Speicher (oder das „höhere“ CM) den Block liefert, sondern muss sofort weitere Aufträge abzuwickeln bereit sein: „Non-blocking Cache“

- bei Stau werden Schreibaufträge zurückgestellt und zwischen-gepuffert; bei Lesezugriffen muss (Konsistenz!) geprüft werden, ob ein Wert mit der verlangten Adresse im Schreibpuffer steht, also nicht aus CM oder Hauptspeicher geholt werden darf.

Cache Memories sind eine feine Sache:

- sie verbessern (meist) Zugriffszeit und Grenzdurchsatz des Prozessors und Last am Speicher stark
- sie stellen keine Ansprüche an den Befehlssatz (wie es Pipelines und EPICs tun)
- trotzdem lassen sie etwas Raum für Optimierung durch den Programmierer/Compiler: geschickte Adresswahl erhöht h bei direkt abbildenden und mengenassoziativen Caches; Prefetch-Befehle holen einen Block in das Cache, bevor der Prozessor ihn braucht

Und haben ihre Schattenseiten:

- bei schlechtem h
- wegen der Konsistenzprobleme, sobald mehr als ein Zugreifer am gemeinsamen System (Cache, Speicher) ist, und ein Schreiber schreibt nicht auf alle Exemplare derselben Zelle, vgl. 1.6 „Inkonsistenz“)

2.9.3 Platzierungsstrategie

Das CM hat weniger Kapazität als der Hauptspeicher. Eine Adresse hat im CM nicht „ihre“ exklusive Zelle. Wohin mit dem Block? Wie ihn wiederfinden?

Voll assoziativ (fully associative): auf einen beliebigen freien Platz (wie Seitenadressierung)

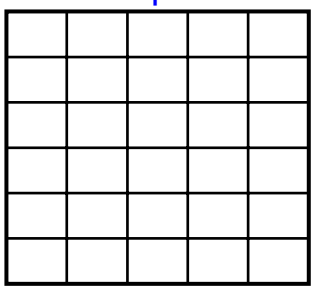
Direkt abbildend (direct mapping): auf genau einen Platz (möglicherweise belegt!)

Z-fach-Gruppen-assoziativ (z-way set associative): auf einen von z Plätzen (möglicherweise alle belegt!)

Auffinden eines Worts/Blocks im CM

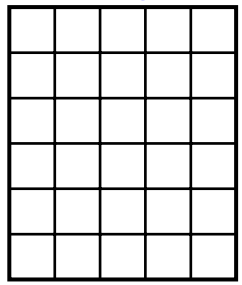
Durch Prüfung der Blockkennung (tag) zu einem (direkt abbildend), z (gruppenassoziativ) oder allen (voll assoziativ) Blöcken im CM.
Typische Struktur im CM:

Datenspeicher



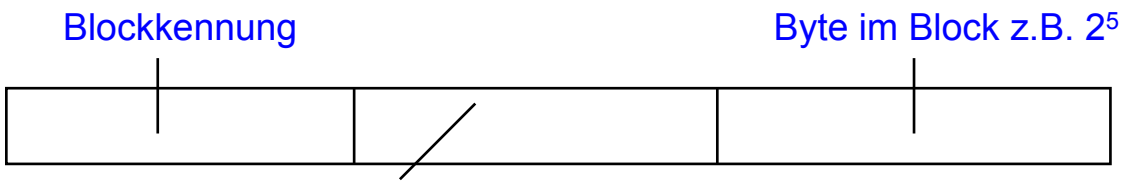
k Blockrahmen für max. k Blöcke

Kennungsspeicher (tag memory)



k Felder für je ein Gültigkeitsbit (belegt?) und eine Blockkennung

Ableitung der Blockkennung aus der Adresse:



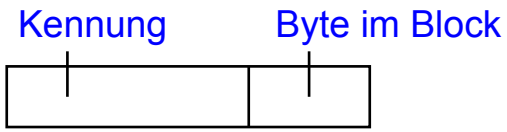
Bei z-fach gruppenassoziativem CM Nummer der Gruppe, in der Block im CM liegen muss, z Blöcke/ Gruppe.

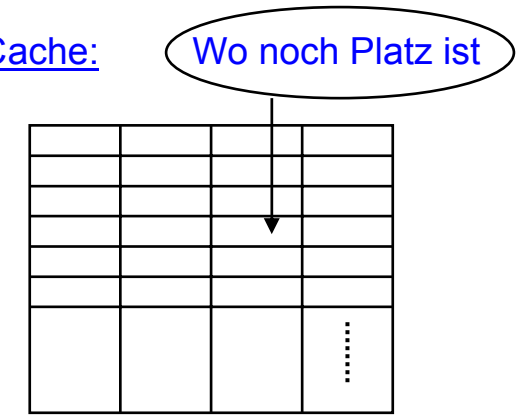
Bei direkter Abbildung: Adresse des Blocks im CM.

Im vollassoziativen Fall ist dieses Feld nicht existent; die Blockkennung ist entsprechend länger!

Damit ergeben sich folgende Varianten:

vollassoziativ



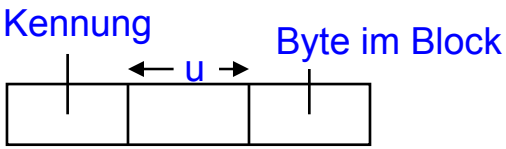


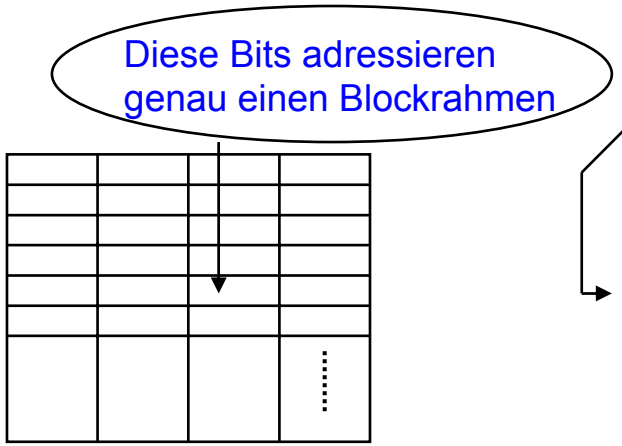
Beste Nutzung der Kapazität

Cache-Kapazität k

Unabhängig von Aufteilung der Adresse

direkt abbildend

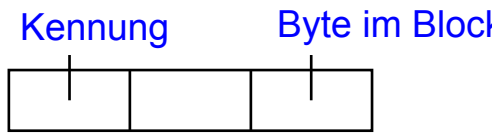


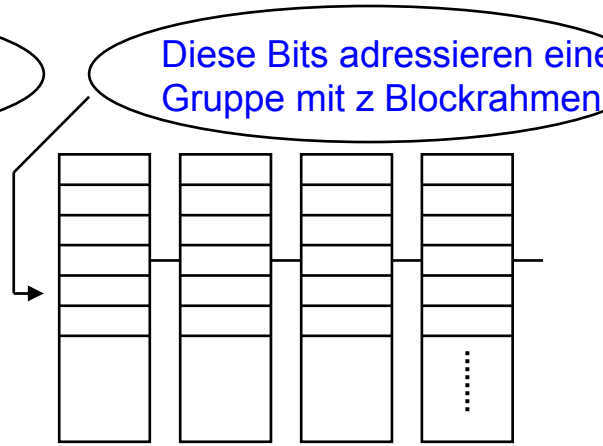


Schnellstes Lesen und Schreiben

$2^u * b$, dabei u: Zahl der Adressbits für Blockrahmen, b Blockgröße

z-fach gruppenassoziativ





$z * 2^u * b$, dabei u: Zahl der Adressbits für Blockrahmen, b Blockgröße

Das Auffinden eines Wortes bei den (z- oder voll-) assoziativen Techniken ist langsamer, obwohl alle Blockkennungen (der Gruppe) gleichzeitig geprüft werden. Direkt abbildende CM nutzen die Kapazität schlechter (unnötige Verdrängungen!)

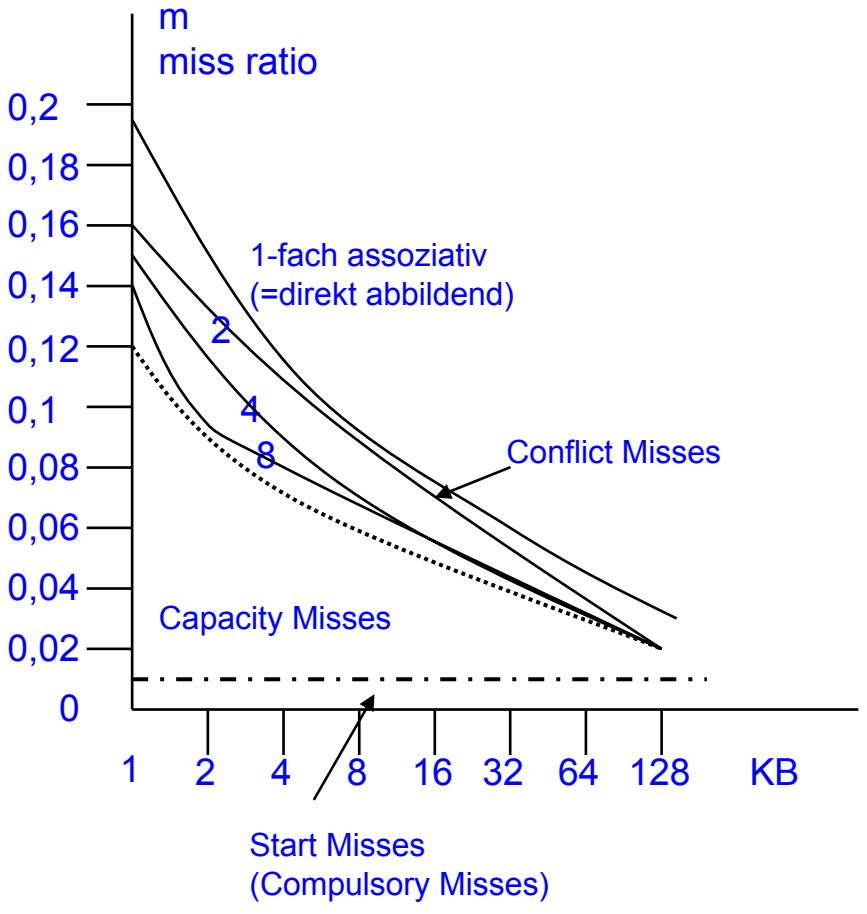
Die Fehlzugriffe (misses) sind unterscheidbar:

- Start (compulsory) Misses: jeder zugegriffene Block muss einmal in das CM geladen werden und erzeugt davor einen Miss

- Capacity Misses: ist die Kapazität eines CM kleiner als die Menge der zugegriffenen Blöcke, dann kommt es (auch bei vollassoziativem Cache) zu Verdrängungen und damit zu Miss bei erneutem Zugriff

- Conflict Misses: sind diejenigen, die nicht in einem voll assoziativen, wohl aber in direkt abbildenden oder z-fach assoziativen CM auftreten, da verdrängt wird, obwohl noch Blockrahmen frei sind.

Typisches Beispiel: Miss ratio als Funktion der Kapazität des CM bei Blockgröße k_B = 32B, Verdrängung LRU, VAX unter Ultrix, Mehrprogrammbetrieb, ca. 10^6 Zugriffe (Simulation!)



VAX: weit verbreiteter mittlerer Rechner der 80er, DEC

ULTRIX: eine UNIX-Spielart

Noch mal genauer:

Mittlere Zugriffszeit bei Lesen (90%):

$T_{SP} = T_{CM} + (1-h) * T_{BlockHS}$

Dabei T_{CM} Zeit für Lesen der Gruppe und Prüfung auf Identität mit Blockkennung. Im hit-Fall ist damit das Lesen abgeschlossen. Sonst (Häufigkeit 1-h) muss die Blockadresse an die Speicher-steuerung geschickt werden, z.B. 4 Speichermoduln (adress-verschränkt!) geben je 8B eines 32B-Blocks ab; muss an Cache geliefert werden. Gesamt Zeitbedarf $T_{BlockHS}$ kann beträchtlich über der Hauptspeicherzugriffszeit T_{HS} (z.B. 40ns) liegen.
Abhilfe: Aus dem Block wird das angeforderte Wort sofort an den Prozessor geliefert.

Mittlere Zugriffszeit bei Schreiben (10%):

Der ZP schreibt als Ergebnis typisch 1 .. 8 B, d.h. es kann mehr als ein Block betroffen sein.
Bei Treffer schreibt der ZP blockweise in das CM: erst lesen, dann ändern, schreiben. Bei Nichttreffer (Miss) gibt es zwei Möglichkeiten

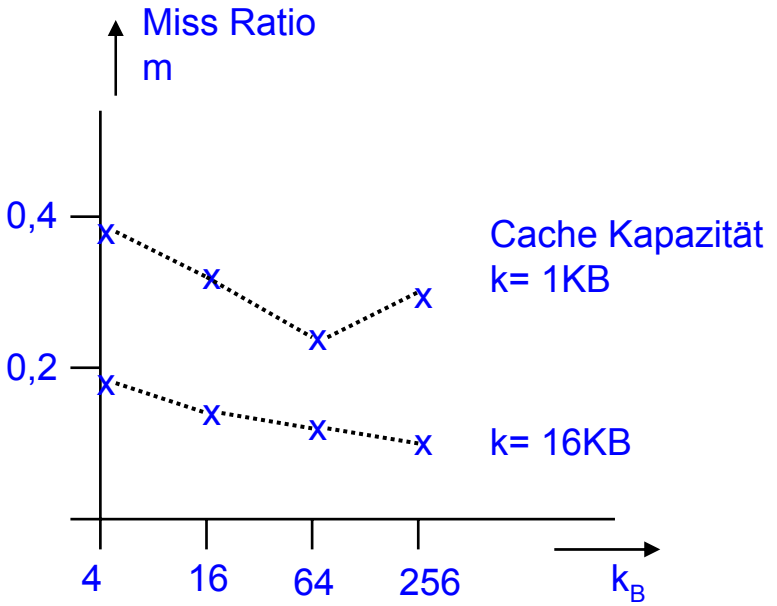
- zu beschreibender Block wird aus Hauptspeicher in CM geladen (write allocate), dann wie oben
- der ZP schreibt nur in den Hauptspeicher. Typisch wartet der ZP nicht das Ende des Schreibens ab, sondern schreibt lediglich in einen Schreibpuffer, der ein oder mehrere Wörter fasst (write around).

Bei Treffer oder bei write allocate muss der Prozessor die Konsistenz wiederherstellen. Zwei Techniken:

- write through (store through): schreibt „sofort“ auch in den Speicher
- copy back: schreibt bei Verdrängung des Blocks im Cache (und vor Verdrängung aus Hauptspeicher!) zurück.

2.9.4 Lade und Ersetzungsstrategie

Einfluss der Blockgröße k_B aus Miss Ratio m: Wenn die Blockgröße klein ist, fehlt der look-ahead-Effekt. Ist sie aber nicht klein im Verhältnis zur Kapazität, führen breit gestreute Zugriffe zu Verdrängungen!
Große Blöcke sind außerdem weniger schnell nachladbar, verlangen mehr Durchsatz an CM, Bus, Haupt-speicher!



Also: Ladestrategie: blockweise on demand. Prefetching (voraus-schauendes Laden) bisher wenig üblich, aber Prefetch-Befehle!

Ersetzungsstrategie: LRU (least recently used, d.h. Verdrängung des am längsten nicht zugegriffenen Blocks) oder Random (Zufall). Einfluss auf h ist nicht allzu groß. – Bei direkt abbildenden Caches gibt es keine Ersetzungsstrategie (keine Wahl zu treffen).

2.9.5 Maschinen- oder Prozessadressen im Cache Memory

Adressierung des CM in Maschinenadressen hat folgende Vorteile gegenüber Adressierung in Prozessadressen:

- gemeinsame Zellen mehrerer Prozesse sind im CM eindeutig. Das gilt ebenso, wenn ein Prozess eine Zelle unter verschiedenen Prozessadressen erreicht (Abbildung auf dieselbe Kachel)

- bei Prozesswechsel muss CM nicht gelöscht werden (vorteilhaft, falls Prozess bald wieder aufgenommen wird, z.B. bei Unterbrechungsbehandlung). Nach Löschung arbeitet CM zunächst mit schlechtem hit ratio! 1000 Unterbrechungen/sek. => 1.000.000 Operationen Abstand, d.h. CM-Inhalt. Inhalt bei großem Cache gerade aufgebaut bis zur nächsten Unterbrechung.

Adressierung in Prozessadressen spart die Zeit für Adressumsetzung (interessant in CM), aber

- diese ist mit Gruppenselektion parallelisierbar

- Konsistenz verlangt Umsetzung Prozessadresse (Prozess 1) → Maschinenadresse → Prozessadresse (Prozess 2) zur Erkennung von „alias“-Zellen

- Löschung bei Prozesswechsel umgehbar, falls Tag um Prozesskennung erweitert wird (bei einigen neuen Prozessoren)

2.10 Parallelität in der Mikromaschine

2.10.1 Übersicht

Das Cache Memory kürzt die effektive Zugriffszeit des Prozessors auf den Hauptspeicher und senkt die Durchsatzansprüche an Bus und Hauptspeicher. Es trägt auch zur Erhöhung des Grenzdurchsatzes der Zentraleinheit (das ist das wesentliche Ziel!) bei, aber nicht ausreichend!

Zurück zu Little:

$$d = \frac{f}{y}$$

(d Durchsatz, f mittlere Füllung (Zahl der Operationen im Prozessor), y mittlere Verweilzeit dieser Operationen, d.h. Bedienzeit- und Wartezeitanteile)

Können wir y nicht verkleinern, müssen wir f erhöhen, aber leider durch Nebenläufigkeit begrenzt (RAW, WAR, WAW-Konflikte) sowie durch Betriebsmittel-(„strukturelle“)Konflikte.

2 Freiheitsgrade, vgl. 1.11.3:

- Pipelines
- mehrfache Werke und Superskalarität
- meist heute kombiniert: mehrere Pipelines/Pipelines mit mehrfachen Werken

Übersicht: Erhöhung des ZE-Grenzdurchsatzes:

Erhöhung der Nebenläufigkeit in einem „Fenster von 10 .. 50 aktuell dekodierten/ ausgeführten Befehlen:

- Code Motion, z.B. Delay-Slot-Befehle: Compiler (Spreizung von präzedenten Folgen durch Einsetzen nebenläufiger Operationen)
- Beseitigung von WAR, WAW-Problemen durch „Schattenregister“
- Verzicht auf bedingte Sprünge durch bedingte Befehle
- Verzicht auf synchrone Unterbrechungen (Poison-Bits)
- Codespreizung durch Multithreading (Einsetzen von Befehlen aus fremden Threads)
- Bildung von „Gruppen“ nebenläufiger Befehle (EPIC, Compiler)

Beseitigung von Betriebsmittelengpässen:

- Mehr (Schatten-)Register, Abbau von Spezialisierung (z.B. Register), mehr Cache/Speicherkapazität, mehr Werke

Verbesserung wichtiger Zeitparameter:

- Cache statt Hauptspeicher: aber Trefferverhältnis (Compiler!), Konsistenz in der Hierarchie
- Register Bypass/Forward
- Branch Target Buffer

Noch Übersicht:

Mehr Parallelität (Nutzung der erhöhten Nebenläufigkeit und der Beseitigung von Betriebsmittelengpässen, und spekulative Ausführung)

- Pipeline
- mehrfache Werke /out-of-order Issue
- spekulative Ausführung: bei bedingten Sprüngen (nach Vorhersage oder immer), bezüglich synchroner Unterbrechungen („nicht beendete Operationen werden schon keine liefern“): erfordert Rücksetzen! Schattenregister!

2.10.2 Behandlung von Sprüngen

Probleme durch Sprünge bei nicht-serieller Ausführung der Befehlsfolge (gleich ob in Pipeline(s) oder mehrfachen Werken):

- Unbedingter Sprung (jump): erst bei Dekodierung des Befehls ist klar, dass die Nachfolger nicht ausgeführt werden dürfen (also: umsonst geholt, etwaige Wirkungen zurücksetzen!). Fortsetzung erst, wenn der am Sprungziel stehende Befehl eingetroffen ist.
- Bedingter Sprung (branch): erst bei Dekodierung des Befehls ist klar, dass vielleicht die Nachfolger nicht ausgeführt werden dürfen (…). Fortsetzung erst, wenn die (u.U. noch nicht berechnete) Sprungbedingung vorliegt. Bei Sprung geht es erst weiter, wenn der am Sprungziel stehende Befehl eingetroffen ist.
- Modifizierter Sprung (unbedingt/bedingt): noch ärger, da die Sprungzieladresse durch Modifikation mit einem (eventuell noch nicht berechneten) Wert gewonnen wird.

Der mit Sprungbefehlen auftretende Aufschub der Befehlsabarbeitung bedeutet i.a. den gleichgroßen Aufschub aller Folgebefehle (d.h. der Durchsatz wird durch jeden Sprung gesenkt!). Abhilfe durch Delay-Slot-Operationen und spekulative Ausführung (aber komplex).

Am besten, es gäbe keine Sprünge! Aber das wäre ein vernichtender Schlag gegen wohlstrukturierte Programme und gegen die (Pseudo-)Turingberechenbarkeit unserer Computer.

Am besten, es gäbe keine bedingten Sprünge! Teilweise durch bedingte Befehle vermeidbar, vgl. Kap. 2.1.4, Predicate Register im Itanium.

Wir studieren zunächst Abhilfen für Probleme, die allen Sprüngen gemeinsam sind, später besondere für bedingte Sprünge:

Delay-Slot-Befehle: Der Prozessor führt n (z.B. 1 .. 4) Befehle, die in der Adressfolge auf den Sprung folgen, stets aus. Damit wird der durch den Sprung entstehende Aufschub genutzt. Voraussetzung: Programmierer bzw. Compiler finden n zum Sprungbefehl nebenläufige Befehle, sonst müssen die „Delay-Slots“ in der Befehlsfolge mit Nullbefehlen gefüllt werden (no ops).

Reduzierung der Zugriffszeit zum Sprungziel, Beschaffung des Nachfolgebefehls. Hoffentlich steht der im Cache Memory 1! Noch schneller geht es mit einem „Sprungzielcache“, wo man unter dem aktuellen Befehlszählerstand den Nachfolgebefehl findet (also noch vor der Decodierung den Sprungbefehl!), außerdem aber dessen Adresse. Organisation wie ein direktabbildendes Cache Memory.

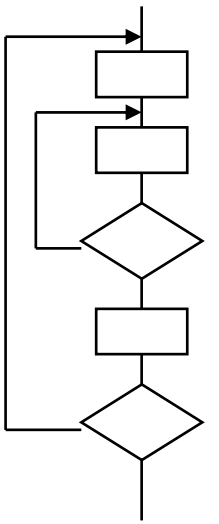
Nun Abhilfen speziell für bedingte Sprünge:

Schneller Zugriff auf die Sprungbedingung:
Durch Resultat eines Vorbefehls (besondere Berechnung, oder Hinterlassung solcher Nebenresultate wie Ergebnis null/negativ/positiv, arithmetischer Überlauf) wird die Sprungbedingung in einem Register abgelegt, auf das der bedingte Sprung zugreift. Typischer RAW-Konflikt! Am besten hat der Compiler zeitlich die Berechnung deutlich vor den Sprungbefehl vorgezogen (geht nicht über sogenannte Flag/Condition Code-Register, das immer über die letzte Operation berichtet!). Weitere Verbesserung durch Register Bypass/Result Forwarding: Das berechnete Kriterium wird sofort für den Sprungbefehl angeboten; dieser muss nicht auf das Ablegen in einem Register und den nachfolgenden eigenen Zugriff warten. Für RAW allgemein eingesetzt!

Schließlich benutzen heute alle Prozessoren Vorhersagetechniken für bedingte Sprünge (wo wird fortgesetzt?)

- Statische Vorhersage (d.h. Information, die vor Start des Programmlaufs vorliegt):
  -- das kann der Prozessor selbst wissen: Rücksprünge werden meist vollzogen, Vorwärtssprünge eher nicht (?)
  -- der Compiler kann es wissen: typischer Durchlauf durch Programm/Datenstruktur. Der Compiler setzt einen Hinweis („Hint“) für den Prozessor in den Befehl
  -- durch „Profiling“, das Programm läuft „probe“, die Sprunghäufigkeit wird gemessen; auch verfeinerte Aufschlüsse möglich, z.B. Sprunghistorien, Abhängigkeiten zwischen Sprüngen. Ebenfalls als „Hint“ in den Befehl.

Die wichtigsten Vorhersagetechniken für bedingte Sprünge sind aber dynamisch:



- Vorhersage aufgrund der letzten Entscheidung oder besser aufgrund der beiden letzten Entscheidungen (verhindert doppelte Falschaussagen am Ende des inneren Schleifenrumpfes), wo die Folge rück/…/rück/vor/rück/…/rück auftritt; 2 Fehler je Abarbeitung der inneren Schleife bei Vorhersage an der letzten Entscheidung; 1 Fehler je Abarbeitung bei Berücksichtigung der beiden letzten Entscheidungen, implementierbar wie/mit Sprungzielcache
- oft noch besser bei Mitberücksichtigung aller k letztdurchlaufener bedingter Sprünge. Ein Verfahren dazu ist, in einem k-stelligen Binärwort diese „Branch History“ festzuhalten und dieses Wort als Zugang zu einem direktabbildenden oder assoziativen Cache zu benutzen, wo aufgezeichnet ist, ob der aktuell vorliegende Sprung beim letzten Auftreten mit gleicher k-Vorgeschichte vollzogen wurde oder nicht.

Schließlich kann man bei bedingten Sprüngen beide Alternativen ausführen (ein Fall für spekulative Ausführung). Diese liegt auch bereits vor, wenn der Prozessor aufgrund einer falschen Vorhersage den falschen Weg verfolgt. Die Ausführung muss rücksetzbar sein:

- Man schreibt auf „Schattenregister“ statt auf die „eigentlichen“ Register: aber Präzedenzverletzung für die weiteren Operationen nicht zu erkennen und Wertübertragungen fraglich. Mit Vollzug des Sprungs werden die Schattenregister gültige Register.
- Man rettet die zu überschreibenden Werte in Schattenregister und holt sie zurück, wenn der Sprung nicht vollzogen wird.

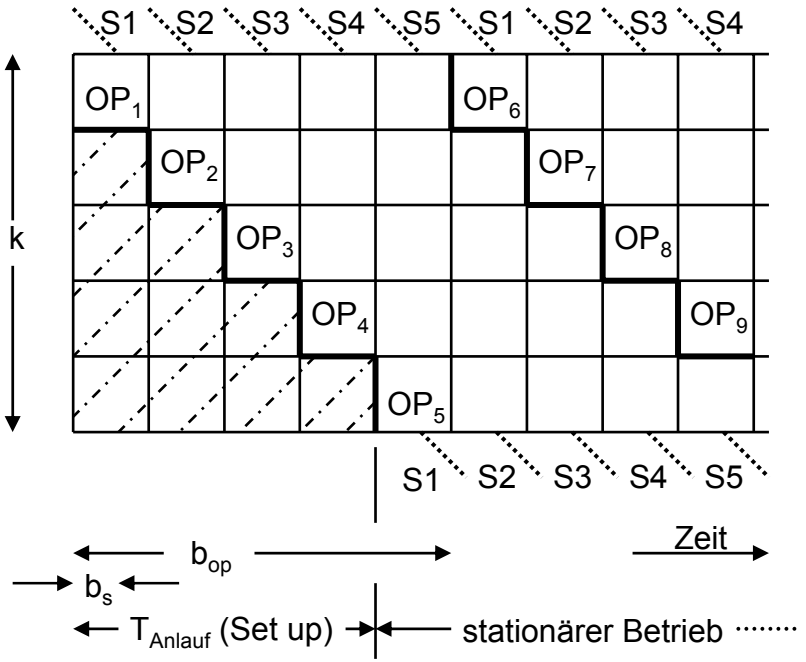
Schattenregister sind ein wichtiges Element in der Parallelarbeit im Prozessor. Der Prozessor besitzt sehr viel mehr Register als sichtbar sind und definiert dynamisch an einer Tafel, welches Register gerade die Rolle welchen sichtbaren Registers spielt.

2.10.3 Pipeline

Pipeline, bekannt seit Abschnitt 1.11.3:

Jede Operation wird in eine Folge von Phasen aufgeteilt, vgl. 2.9.1, und in einer Kette von Werken bearbeitet, deren jedes eine Phase durchführt. Die Operationen Op_i folgen einander dicht, so dass bei Durchführung von Phase p in Stufe s der Pipeline an Operationen Op_i gerade Phase p-1 an Operation Op_{i-1} in Stufe s-1 bearbeitet wird. Jede Stufe hat eine gleich große Bedienzeit b_s. Dauert eine Phase länger als b_s so wird sie auf n konsekutive Stufen aufgeteilt: n= entier (b_p/b_s) + 1 (b_p Dauer der Phase p). Die Bedienzeit für jede Operation Op dauert b_{op}; die Pipeline hat k Stufen, und der Grenzdurchsatz ist ideal $c= 1/b_s = 1/(b_{op}/k) = k/b_{op}$, d.h. mit der Stufigkeit k beliebig steigerbar! Auch wenn b_{op} groß ist (z.B. Speicherzugriff)!

„Raum“/Zeit/Operationen-Diagramm einer k=5-stufigen Pipeline



z.B.:

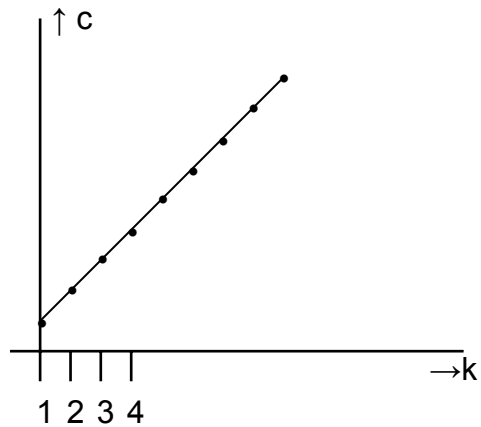
S1: Befehl holen (instruction fetch)
S2: Befehl entschlüsseln (decode)
S3: Operand holen (operand fetch)
S4: Operation ausführen (execute)
S5: Ergebnis speichern (result store)

Allgemein: c = c(k) (stationär)

Takt/Stufenzeit $b_s = b_{op}/k$

also: $c = \frac{1}{b_s} = k/b_{op}$

aber Anlauf kostet k-1 Operationen im Grenzdurchsatz!



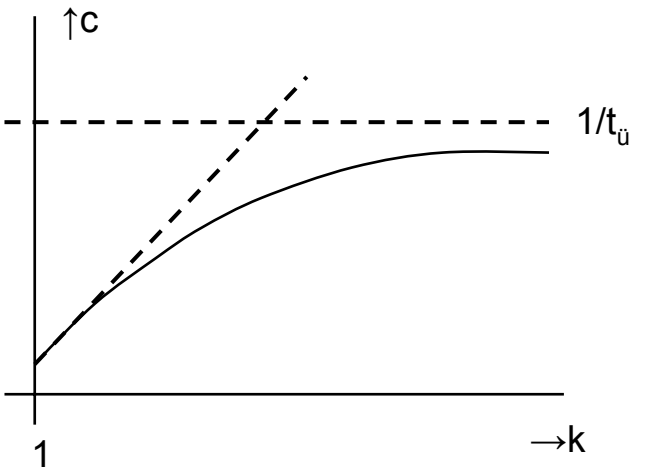
Kommentare zu unserer Erfindung (1):

1. Es ist nicht möglich, k beliebig groß zu machen: $b_s = b_{op}/k$ ist die Taktzeit der Pipeline und (fast immer) auch des Prozessors. Schaltungstechnische und thermische Grenzen, heute bei ca. 4 GHz.
2. Es ist nicht nützlich, k beliebig groß zu machen. In der Taktzeit b_s muss die Übergabe von Stufe zu Stufe (über ein Register) untergebracht werden, $t_ü$. Also ist von $b_s = b_s' + t_ü$ nur b_s' produktiv und wird mit k verkleinert:

$c = 1/b_s = 1/(b_{op}/k + t_ü)$

und für

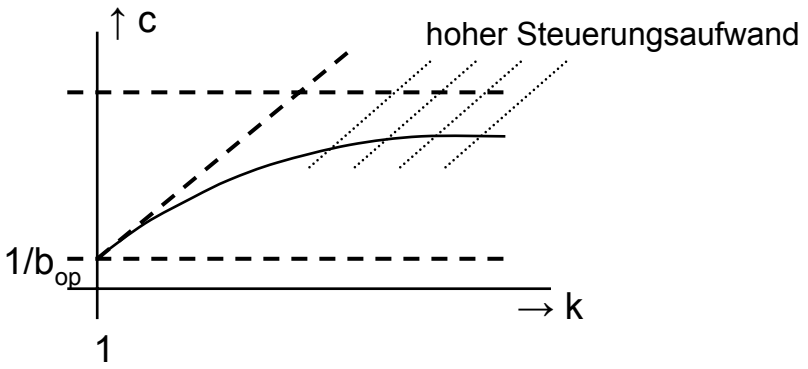
$k \rightarrow \infty \quad c \rightarrow 1/t_ü$



Kommentare zu unserer Erfindung (2):

3. Es ist schädlich, k sehr groß zu machen. k ist auch die Zahl der aktuell nicht vollendeten Operationen! Bei Start einer Phase p müssen alle erforderlichen Werte (Operanden, Sprungbedingungen, Modifikationsgrößen, …) vorliegen und alle benötigten Betriebsmittel müssen frei sein (Register, Cache, Addierer, …). Bei einem „synchronen“ Unterbrechungswunsch müssen alle präzedenten Operationen zunächst fertig gestellt werden („in-order retire“) (oder wenigstens so weit gebracht werden, dass sie keine Unterbrechungswünsche mehr liefern). In letzterem Fall wächst die Menge der potentiellen Konfliktoperationen mit O(k). Der Overhead für die korrekte Steuerung steigt quadratisch mit k!

Also sieht die Charakteristik qualitativ so aus:



Die uns bekannten Hilfsmittel:

- Register Bypass / Result Forwarding
- Schattenregister (gegen WAR- und WAW-Konflikte)
- Sprungvorhersagen
- Sprungziel-Cache
- Delay-Slot-Operationen

lindern die Leiden, sie beseitigen sie nicht

Pipelines haben 5 bis 15 Stufen.

Berechnung zu unserer Erfindung:

Wenn eine Operation mit Wahrscheinlichkeit P_{Halt} den Fluss für im Mittel m Schnitte aufhält, dann dauert die Abarbeitung von n Operationen nicht

$T_n = n*b_s$, sondern

$T_n = n*b_s + n*P_{Halt}*m*b_s = n*b_s(1+P_{Halt}*m)$

Der Durchsatz beträgt:

$d = n/(n*b_s*(1+P_{Halt}*m)) = 1/b_s(1+P_{Halt}*m)$

Und die mittlere Ausführungszeit einer Operation (Little!)

$y = k/d = k*b_s*(1+P_{Halt}*m)$

Der Pipeline-Halt heißt „stall“.

In Kalifornien heißt $1+P_{Halt}*m$ „CPI“ (Cycles per Instruction).

Noch ein paar Bemerkungen:

- Im Idealfall ist die Verlängerung von b_{op} (etwa Zugriff auf langsamen Speicher CM2/3 oder gar Hauptspeicher, oder lange Ausführungsphasen (execute)) ohne Einfluss auf den Grenzdurchsatz, aber wohl auf k ($k = c*b_{op}$!), also im Realfall wieder schädlich (Zahl der Konfliktpartner).

- Man wird daher Operationen mit langer Ausführungszeit (Beispiele: Festpunktaddition/Subtraktion, Shift, boolesche Operation: 1 Takt, Gleitpunktaddition/Subtraktion: 2 Takte, Multiplikation: 4 Takte, aber Division, Wurzel = 24 Takte) außerhalb der Pipeline seriell ausführen. Aber das Divisionswerk hat dann einen Grenzdurchsatz von nur 1/24 Takten, und wehe die Division endet mit Abspeicherung im Hauptspeicher und liefert einen Seitenfehler!

Noch eine Bemerkung:

- Wenn eine Operation nicht fortsetzbar ist (wartet auf Operand, z.B. Quotient), dann ist es wünschenswert (aber komplex), dass die Operation von anderen, nebenläufigen überholt wird. Aber: Operation rücksetzen, später wieder einspeisen, eventuelle verschachtelt, bei mehreren wartenden Operationen. „out-of-order issue“. Wichtig für mehrfache Werke, nächster Abschnitt.

Und Compiler und unsere Erfindung:

Der Compiler kann uns helfen:

- RAW Konfliktpartner durch Einschieben nebenläufiger Operationen auseinanderzerren, wenn er keine findet, „no op“s einsetzen.
- Ebenso mit den Delay Slots.
- Und die Betriebsmittelbelegung bei der Codeerzeugung simulieren, selbe Medizin.

Immerhin nimmt er der Pipeline ihre Steuerungskomplexität (was die RISC-Erfinder liebten!), aber wenn wir einen Prozessor für Intel oder IBM bauen, und 20 Jahre alter Binärcode soll laufen, hilft das nichts (und die „no op“s nehmen noch Platz im Speicher weg).

Letztes Pipeline-Wort:

Die SIMD-Pipelines in Vektor/Graphik/Signalprozessoren unterliegen lediglich unseren Kommentaren 1 und 2. Für sie ist vor allem die Anlaufzeit (set-up time) kritisch (k-1 Takte!). Also bei Vektor-Länge n

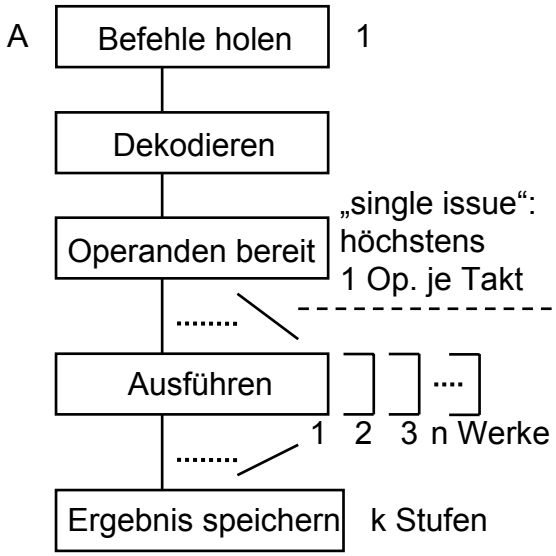
c nicht $1/b_s$, sondern

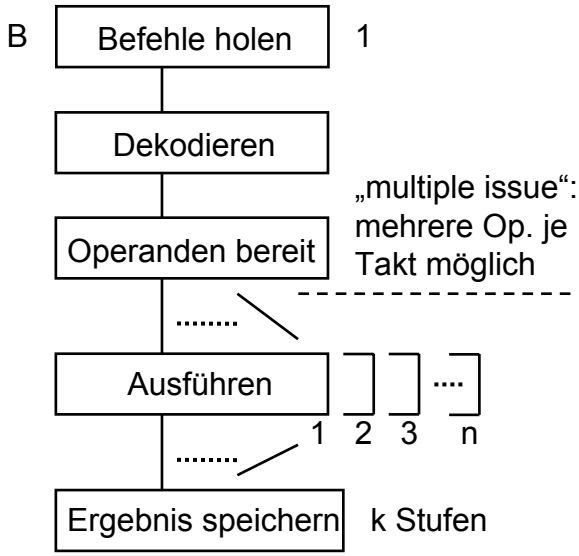
$$c = \frac{n}{n + k-1} * \frac{1}{b_s}$$ (k-1 Takte Anlaufzeit),

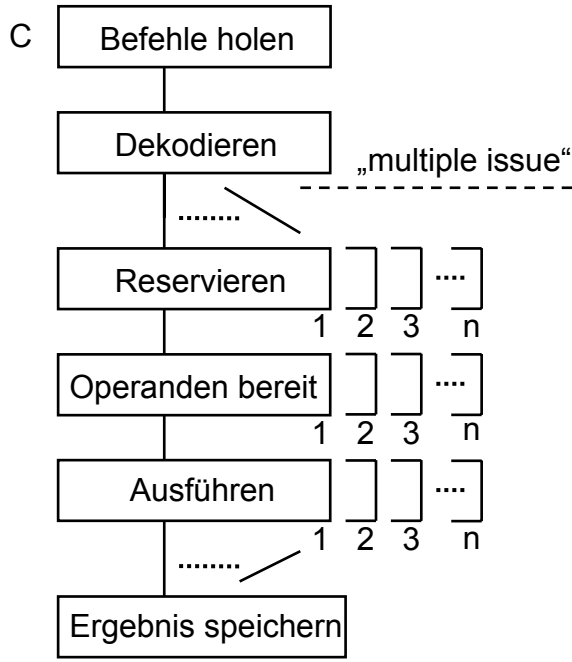
also lange Vektoren gesucht!

2.10.4 Mehrfache Werke und Superskalarität

Nur die Ausführungsphase oder ein großer Teil des Befehlsablaufs wird autonomen Werken übertragen:







erreichbar:

$c = \frac{k}{b_{op}}$ (b_{op} Ausführungszeit einer Op.),

wenn der Gesamtdurchsatz der n Werke so groß ist wie der jeder anderen einzelnen Stufe

erreichbar:

$c = \frac{k * n}{b_{op}}$ n issue

„Reservation Station“: dezentrale Koordination und Synchronisation

mehrere Werke und Multiple Issue: Superskalarität

Bestenfalls erreichbarer Durchsatz, d.h. ohne irgendwelche Konflikte, Vollauslastung der n Werke, Taktzeit b_o, Operationsausführungszeit b_{op}:

Fall A: $c = k/b_{op} = \frac{1}{b_o}$

Voraussetzung: die n Werke haben einen Gesamtgrenzdurchsatz wie der Rest der Pipeline, d.h. $1/b_o$, und die Operationen treten in günstiger Folge auf (also nicht: aufeinanderfolgende Operationen gehen an dasselbe Werk), sonst: jedes Werk hat Grenzdurchsatz $1/b_o$!

Fall B/C: $c = k*n/b_{op} = n/b_o$

Voraussetzung: jedes Werk hat einen Grenzdurchsatz $1/b_o$ (Pipelining!) und die Operationen treten in günstiger Folge auf; in einem „issue“ können höchstens so viele gleichartige Operationen gestartet werden, wie (freie) geeignete Werke vorhanden sind.

In der 1. Phase werden i.a. (Extrapolation des Befehlszählerstandes) viele Bytes des Programms geholt, in der 2. Phase dekodiert und erforderlichenfalls (Pentium) in Befehle gegliedert.

Im Fall A und B wird geprüft, ob die Operanden in Registern verfügbar sind und ob ein passendes Werk frei ist und dann
Fall A: höchstens eine Operation an die Ausführung übergeben (issue)
Fall B: mehrere (bis 2 .. 6) Operationen an die Ausführung übergeben (multi-issue, „Superskalarität).

Die Bearbeitung der Operationen in den Werken dauert verschieden lang, dadurch sind in A und B Überholungen während der Ausführung möglich. Eine wichtige Unterscheidung ist noch, ob sich Operationen beim „issue“ überholen dürfen (out-of-order issue) und ob Operationen beim Schreiben ihres Ergebnisses in ihr Zielregister von der vom Programm vorgegebenen Folge abweichen dürfen (out-of-order retire).

Fall C ist superskalar wie B, unterscheidet sich darin, dass ein Teil der Koordination/Synchronisation an „Reservierungsstationen“ delegiert ist, die jeweils einem Werk zugeteilt sind und die Bereitstellung der Operanden prüfen. B und C geht meist mit höheren Durchsatzansprüchen einher, dazu organisiert man die Werke auch als Pipeline.

Die Bezeichnung „superskalar“ ist töricht; Deutungen:

- Der Prozessor macht „skalare“ Operationen (d.h. nicht-vektorielle, nicht-SIMD) leistet trotzdem super was.
- Durch den Übergang zu „multiple issue“ wird eine Leistungssteigerung erzielt, die mehr als skalar zur Erhöhung der Taktfrequenz ist.

Das Schema A wurde mit der Control Data CDC 6600 (1964) geboren (Cray, Thornton). Die CDC 6600 benutzte dabei single in-order issue, 10 exklusive Werke (höchstens eine Operation, kein Pipelining), out-of-order retire (unschädlich, da es keine synchronen Unterbrechungen, sondern nur „ungültige Ergebnisse“ gab). Zur Steuerung wurde das Scoreboard benutzt (Punkte-(Tor-) Anzeige), ähnlich unserem nachfolgenden 1. Beispiel aus [Tannenbaum A 05].

Das Schema C kam mit dem Modell 91 der IBM/360 (1967). Auf ihm wird ein Verfahren benutzt, das nach seinem Erfinder Tomasulo-Algorithmus heißt.

Beispiel 1 aus [Tannenbaum A 05]:

- 2 issue
- in-order issue (nicht ausgebbare Operation hält die Nachfolger auf)
- in-order retire (zunächst müssen alle Vorgänger geschrieben haben)
- Register werden mehrfach gelesen (Zählung offener Lesezugriffe)
- Register werden einmal beschrieben (kein/ein offener Schreibzugriff)
- Addition/Subtraktion braucht 2 Takte im Werk, Multiplikation 3 Takte, Resultat verfügbar am Ende des letzten Takts
- Zuteilung der Werke wird im Beispiel nicht berücksichtigt, also: “jederzeit ist gesuchtes Werk frei“

Beispiel 1:

					Registers being read								Registers being written							
Cy	#	Decoded	Iss	Ret	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
1	1	R3=R0*R1	1		1	1										1				
	2	R4=R0+R2	2		2	1	1									1	1			
2	3	R5=R0+R1	3		3	2	1									1	1	1		
	4	R6=R1+R4	—		3	2	1									1	1	1		
3					3	2	1									1	1	1		
4				1	2	1	1										1	1		
				2	1	1												1		
				3																
5			4			1			1										1	
	5	R7=R1*R2	5			2	1		1										1	1
6	6	R1=R0−R2	—			2	1		1										1	1
7				4		1	1													1
8				5																
9			6		1		1							1						
	7	R3=R3*R1	—		1		1							1						
10					1		1							1						
11				6																
12			7			1		1								1				
	8	R1=R4+R4	—			1		1								1				
13						1		1								1				
14						1		1								1				
15				7																
16			8						2					1						
17									2					1						
18				8																

Figure 4-43. A superscalar CPU with in-order issue and in-order completion.

Beispiel 2 aus [Tannenbaum A 05]:

Wie Beispiel 1:

- 2 issue
- Register werden mehrfach gelesen
- Register werden einmal beschrieben
- Addition/Subtraktion 2 Takte, Multiplikation 3 Takte,
- Zuteilung der Werke nicht berücksichtigt

aber:

- out-of-order issue
- out-of-order retire (im Beispiel tritt keine präzedente Unterbrechung auf)
- Schattenregister S1 und S2

und damit Kürzung des Zeitbedarfs von 18 auf 9 Takte!

Beispiel 2:

					Registers being read								Registers being written							
Cy	#	Decoded	Iss	Ret	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
1	1	R3=R0*R1	1		1	1										1				
	2	R4=R0+R2	2		2	1	1									1	1			
2	3	R5=R0+R1	3		3	2	1									1	1	1		
	4	R6=R1+R4	–		3	2	1									1	1	1		
3	5	R7=R1*R2	5		3	3	2									1	1	1		1
	6	S1=R0–R2	6		4	3	3									1	1	1		1
				2	3	3	2									1		1		1
4			4		3	4	2		1							1		1	1	1
	7	R3=R3*S1	–		3	4	2		1							1		1	1	1
	8	S2=R4+R4	8		3	4	2		3							1		1	1	1
				1	2	3	2		3									1	1	1
				3	1	2	2		3										1	1
5				6		2	1		3					1					1	1
6			7			2	1	1	3					1		1			1	1
				4		1	1	1	2					1		1				1
				5				1	2					1		1				
				8				1								1				
7								1								1				
8								1								1				
9				7																

Figure 4-44. Operation of a superscalar CPU with out-of-order issue and out-of-order completion.

2.10.5 Vergleich EPIC/VLIW

EPIC Explicitly Parallel Instruction Computing und VLIW Very Long Instruction Word (eine der Manifestationen von EPIC) ist in 2.1.4 Intel Itanium (IA-64) beschrieben worden.

EPIC:

- Compiler soll möglichst viel Nebenläufigkeit herstellen (vgl. 2.10.1): Code Motion, Schattenregister bzw. Einsatz von vielen Registern, Einsatz bedingter Befehle (Erfindung von DEC Alpha, nicht von Intel Itanium), Verzicht auf synchrone Unterbrechungen, spekulative Ausführung, … (das gilt für alle Prozessoren! Aber ungeeignet wenn alte Objekte (Binaries) auszuführen sind (Legacy Systems))

- Compiler soll Parallelität festlegen (hier liegt die Stärke von EPIC!): Befehlsbündel, Zuweisung der Befehle an Werke (noch viel weniger für Legacy Systems).

Hauptvorteil bei Einsatz des Compilers für Nebenläufigkeit und Parallelität: Befehlsfolge gestaltbar, großer Kontext bekannt, Prozessor muss Nebenläufigkeit und Parallelität nicht selbst planen und realisieren

(allerdings: auch der Itanium braucht Sprungvorhersage und spekulative Ausführung)

damit Entlastung der Prozessor-Komplexität. Platz für mehr Register, mehr Werke.

Aber Begrenzung: Compiler kennt den realen Ablauf nur vage (wirkliche Zugriffszeiten, Ausführungszeiten, Unterbrechungen, Sprungentscheidungen), und darin bewähren sich die Pipeline/ Superskalar-Prozessoren dieses Kapitels und bringen eine Leistung von gleicher Größenordnung wie Itanium!

2.10.6 Multithreading

Ein Thread (Faden) ist eine Operationsfolge (Ausführung einer Befehlsfolge), die zu anderen Threads nebenläufig ist. Mehrere Threads gehören zu einem Prozess (Einheit der Betriebsmittel/ Ablaufplanung des Betriebssystems). Sie teilen sich dem Prozess zugeteilte Betriebsmittel (vor allem CPU-Zeit, Hauptspeicher) und werden in demselben Prozessadressraum des Prozesses ausgeführt. Threads werden im Zeitmultiplex und „Raum“multiplex auf einem Prozessor ausgeführt (oder auch auf „Multicore-Prozessoren“). Wie (eine Ebene „höher“) der Mehrprogrammbetrieb es erlaubt, E/A-Zeiten eines Prozesses durch einen anderen Prozess zu nutzen, nutzt Multithreading kurzzeitige Auslastungslücken von Werken des Prozessors. Macht mehrfache Registersätze erforderlich!

Nutzung von Auslastungslücken des Prozessors: mehr Durchsatz (Operationen/Zeiteinheit) und (da die Threads zu demselben Prozess gehören) kürzere Ausführungszeit!

Formen von Multithreading:

Feingranulares Multithreading: Der Prozessor wechselt von Operation zu Operation den Thread (Honeywell H 800, 1960!!)

Grobgranulares Multithreading: Der Prozessor wechselt den Thread, sobald der aktuelle Thread wartet, z.B. Cache Miss.

Simultanes Multithreading: Der Prozessor unterhält gleichzeitig mehrere Threads (d.h er holt Befehle mehrerer Threads und führt sie parallel in allen Stufen der Pipeline aus).
Besondere Unterstützung erforderlich („Hyperthreading“).

3. Speicher

3.1 Hauptspeicher

Hauptspeicher sind Speicher mit adressierbarem Einzelzellen-zugriff (und meist auch Blockzugriff: Cache-Transport). Heute sind alle Hauptspeicher Halbleiterspeicher.

Zu Erhöhung des Grenzdurchsatzes der Hauptspeicher:

- technische Wortlänge oft viel größer als Prozessorwortlänge: Einzelzugriff liefert 64 .. 256 Bits

- Speicherverschränkung: Der Hauptspeicher ist aus vielen (n= 4 .. 32) Moduln (Bänken) aufgebaut, die parallel arbeitsfähig sind. Bei Zugriff auf eine Folge von Wörtern wird ausgenutzt, dass konsekutive Adressen modulo 2^n auf die 2^n Moduln verteilt werden: die letzten n Adressbits werden als Moduladresse benutzt, der Rest der Adresse adressiert im Modul („Speicherverschränkung“, memory interleaving).

Hauptspeichertechniken:

Static RAMs (SRAMs):

RAM (Random Access Memory) für Hauptspeicher unspezifische Bezeichnung, die aufkam, als Trommeln als Hauptspeicher durch die zellenweise mit gleicher Zugriffszeit les/schreibbaren Kernspeicher ersetzt wurden. SRAMs benutzen 6-Transistor Flipflopschaltung je Bit, Speicherung erlischt mit Betriebsspannung (volatile). Zugriffszeit wenige ns. Technologie der Cache Memories!

Dynamic RAMs (DRAMs):

Speicherung in einer integrierten Kapazität, 1 Transistor je Bit. 20 .. 60 ns Zugriffszeit. Durch Leckströme verschwindet Ladung, macht „Refresh“ (Lesen/Schreiben) etwa alle 20 ms notwendig. Variante DDR (Double Data Rate) liefert 2 Wörter je Lesezyklus aus. Die Technologie der Hauptspeicher!

ROM (Read-only memory):
Schreiben einmalig (bei Produktion (Maske) oder durch „Programmierung“, PROM), für unveränderliche Inhalte, z.B. Programmspeicher in Micro-controllern.

EPROM (Erasable Programmable ROM):
Durch Bestrahlung mit UV-Licht kann PROM-Inhalt wieder gelöscht werden; also nicht im Betrieb löschbar!

Und außerhalb des Hauptspeichers, aber wichtig (Kamera, Mobiltelefon, …) vielleicht Ersatz für Plattenspeicher (!!!):

Flash Memory:
Bausteine bis etwa 32 GB, 50 ns, aber Lebensdauer nur etwa 100.000 Löschungen.

3.2 Periphere Speicher und Datenträger (alle rotierend, nur Übersicht)

Bauform	Speicher-prinzip	Goemetrie	Zugriff	Zugriffs-zeit	Durchsatz	Kapazität	Verwendung
Platten-speicher	magnetisch	1..4 Scheiben, beidseitig, konzen-trische Kreise	Radial verschiebli cher Arm mit S/L-Kopf	2 ..6 ms	bis 320 MB/s	bis 200 GB	Rechner
CD-ROM Compakt Disk	Vertiefungen in Ober-fläche	1 Scheibe, ein/beid-seitige Spirale	Laser	Sekunden	160 KB/s	0,7 GB je Seite	Daten, Musik
DVD Digitale Versatile Disk	Vertiefungen bis 2 Schichten	1 Scheibe, ein/beid-seitige Spirale	Laser	Sekunden	1,4 MB/s	4,7 GB je Seite und Schicht	Daten, Video
Blu Ray	Vertiefungen auch mehrere Schichten	1 Scheibe, einseitig	Laser	Sekunden	4,5 MB/s	25 GB je Schicht	Daten, Video

4. Bussysteme und rechnerinterne Vernetzung

Ein Bus ist ein exklusives leitungsgebundenes Kommunikations-medium, das n (>2) Kommunikationspartner im Zeitmultiplex zur Verfügung steht.

Wichtige Dimensionen:

- seriell/parallel (und dann: wie breit?)
- Adressformat (Adresse dient der Identifizierung der Partner bzw. einzelner Zellen bei ihnen)
- Datenformat
- synchron: der Bus gewährt auf Aufforderung feste Zeitquanten zur Übertragung; asynchron: Bus passt sich geforderten Übertragungszeiten an, geringerer Durchsatz
- Master: kann einen Datentransport starten
- Arbiter: entscheidet auf einem „multi-master-fähigen“ Bus, wer ihn einsetzen darf
- Brücken (bridges) zwischen Bussen.
- Local Bus, Memory Bus, Peripheral Bus (hier Standards wichtig!)

Nochmals [Tanenbaum A 06] :

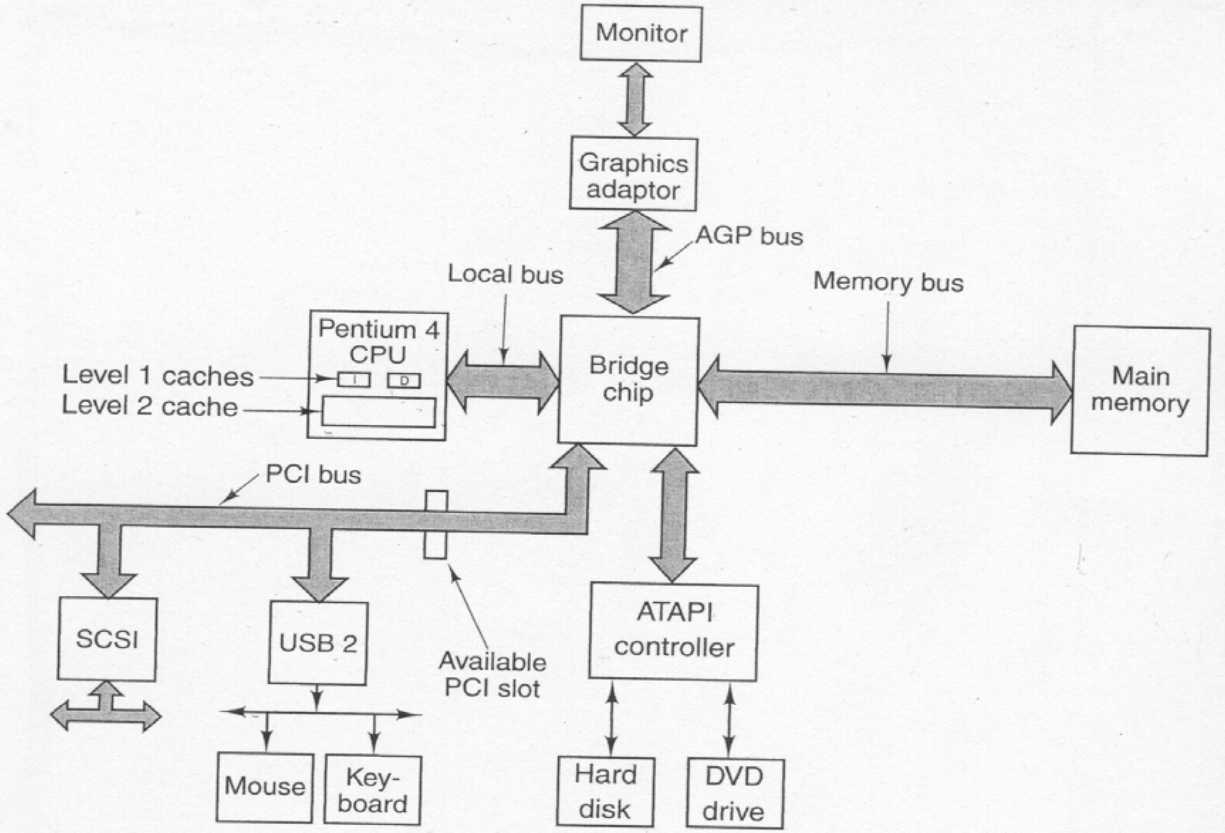


Figure 3-53. The bus structure of a modern Pentium 4.

Beispiele:

Name	Jahr	Adresse	Daten	Frequenz	Datendurchsatz	Einsatz
ISA Industry Standard Architecture	1985	20b	16b	8,33 MHz	16,7 MB/s	Peripherie
EISA extended ISA	1988		32b	8,33 MHz	33,3 MB/s	Peripherie
PCI Peripheral Component Interconnect	1990		64b		528 MB/s	Peripherie, auch mobile Rechner
SCSI Small Computer System Interface	1986		16b	160 MHz	320 MB/s	periphere Speicher
AGP Accelerated Graphics Port	2000				2,1 GB/s	Graphik
USB Universal Serial Bus	1993			0,06 MHz	1,5 GB/s	langsame Peripherie plug-and-play, kaskadierbar

PCI Express

Kein Bus, sondern eine Punkt-zu-Multipunkt-Verbindung mit Paketprotokoll (also ähnlich strukturierter Ethernet-Verkabelung). Serielle Übertragung, 2,5 .. 10 Gb/s (0,3 .. 0,8 GB/s), Cyclic Redundancy Check, Quittungen, Flusskontrolle, virtuelle Kanäle.

5. (statt Ein/Ausgabe:) Thermik

Computer brauchen inakzeptabel viel Energie (tragen dabei zur CO_2-Produktion bei (also SGI Altix LRZ 1+1 MW, ganze Fakultät für Informatik mehr als 0,1 MW)). 3% des Weltverbrauchs, wie Luftverkehr; Rechnerverbrauch wächst schneller!

Die Energiepreise steigen. Betriebskosten in 5 Jahren erreichen Anschaffungskosten.

Die Erwärmung verhindert die Leistungssteigerung (Taktfrequenz steigt seit einigen Jahren kaum. 150 W-Grenze für CPUs).

Maßnahmen:

- Abschalten wann immer möglich! (Ruheverluste sind viel zu hoch!)
- Ungenutzte Kapazitäten nutzen, auch extern (Grids, Clouds)
- Übergang in rationellere Konfigurationen („Serverzentralisierung“)
- Optimierung von Software und Konfigurationen
- Effizientere Stromversorgung und Kühlung
- Serverfarmen an Orte billiger Primärenergie legen (mit Wind, Flut und Sonnenschein wandern?)
- Höhere Integration (z.B. Multicore) liefert Beispiele von Einsparung (effizientere Kommunikation), aber Wärmeabfuhr erschwert!
- Neue Halbleitertechnik (Indium-Antimonid)
- Effizientere Bussysteme und Peripherie (Flash Memory statt Platte?)
- Selektive lastgesteuerte Schlaf/Ab-Schaltung (Takt, Spannung)

Intel hofft bis 2010 zehnfache Leistung bei 10-facher Energieeffizienz zu bieten (d.h. Abwärme wie heute)

6. MIMD: Multiprozessoren, Multirechner, Parallelrechner, Grids

6.1 Übersicht

Multiprozessorsystem: Rechensystem mit mehreren Prozessoren, dessen Hauptspeicher allen Prozessoren adressiert zugreifbar ist (z.B. Load, Store). Die Prozessoren sind in ihrer Funktionalität und Rolle im System nicht spezialisiert, dann:

Symmetrisches Multiprozessorsystem SMP (Vorsicht, heißt auch Shared Memory Multiprocessor): Vorteile für Programmierung, Betriebsmittelzuteilung, Fehlertoleranz.
Asymmetrische Multiprozessorsysteme: z.B. Prozessoren mit Peripherie-Zugang, Spezialprozessoren.

Ein Betriebssystem!
Faktisch sehen nicht alle Prozessoren alle Speicherinhalte jederzeit gleich (Kompromiss für mehr Parallelität): Problem der „Ablauf-Konsistenz“ im Hauptspeicher.
Daneben Cache-Kohärenzproblem (vgl. 1 writer, n readers, siehe 1.5)

Wesentliches Handicap für die Entwicklung „großer“ Multiprozessor-systeme ist die gemeinsame Hauptspeicherschnittstelle (durch Caches gemildert, verläuft hierüber die Programminterpretation von n Prozessoren!) Daher neben
UMA (uniform memory access) verbreitet NUMA (non-UMA): Zu den Prozessoren gehören „lokale“ Speicher, zu denen sie bevorzugt zugreifen, dabei
cc NUMA: Cache Consistent NUMA: Hardware sorgt für Kohärenz.
ncc NUMA: non cc NUMA: Software muss Kohärenz besorgen.

Anderer Ansatz: Der Speicher ist zwar verteilt, aber die Seiten des virtuellen Gesamtspeichers wandern zum Zugreifer (COMA Cache-only memory access).

Multiprozessoren haben durch die Multicore-Entwicklung zusätzliche Bedeutung erhalten! Beispiel Sun T2 (6.4)
Multithread-Prozessoren sind funktional den Multiprozessoren äquivalent.

Multirechnersystem (distributed memory multiprocessor (unglückliche Bezeichnung!)): Rechensysteme aus n Rechnern ohne gemeinsamen Adressraum. Kommunikation über Botschaften. n Betriebssysteme, oft n mal dieselbe Anwendungssoftware: Speicheraufwand! Kräftige Entlastung des Speicherverkehrs. Multirechner sind daher für große n geeignet, z.B. 10^5. Meist werden allerdings hierarchische Lösungen gewählt: bei denen die „Rechner“ SMP-Multicores sind. Damit in 2008 1 Petaflop/s voraussichtlich erreicht (10^6 x 1 GFlop/s). Multirechner schlechter zu programmieren (MPI Message Passing Interface).

Formen: Konzentrierter Multirechner (wie SGI Altix, siehe 1.6), die eigentlich eine ccNUMA-Maschine ist, mit einem Adressraum, aber wie ein Multirechner gefahren wird.
Cluster aus PCs oder Workstations
Grids aus i.a. heterogenen, geographisch verteilten Rechnern und Instrumenten, dynamische Konfiguration.

Parallelrechner: Rechensystem, das Aufträge ausführt, die (intern, meist explizit) nebenläufig sind und parallel ausgeführt werden.

Wichtige Bauformen: Multiprozessorsysteme, Multirechnersysteme, (Feld/Vektorrechner).
Ziel des Parallelrechnens: Bedienzeit großer Aufträge kürzen.
Problem: Algorithmusentwicklung, automatische Erzeugung nebenläufiger Programme aus sequentiellen Programmen.

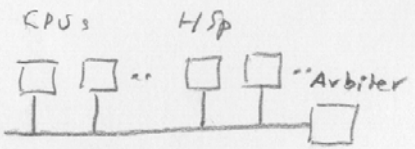
Multiprozessoren/Multirechner haben noch andere Zwecke:
- Leistungssteigerung im Multiprogramming-Betrieb
- große Transaktionssysteme
- Fehlertoleranz

6.2 Verbindungswerke

Wir vertiefen Abschnitt 1.7, insbes. 1.7.5: Gitter, Baum, Hypercube.
Heute werden die meisten Multiprozessoren/Multirechner aus handelsüblichen Prozessor/Speicher-Rechnerbausteinen entwickelt (COTS components off the shelf). Das wichtigste strukturgebende Merkmal ist damit das Verbindungswerk (interconnection network).

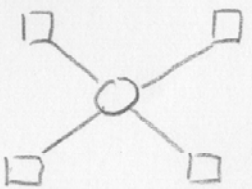
Anforderungen:
Grenzdurchsatz (Netz, einzelne Verbindung), mittlere Laufzeit einer Nachricht (Bedien/Verweilzeit, bei vermittelnden Netzen: Stufigkeit), Blockierung (Endteilnehmer frei, aber kein freier Weg durch das Netz), Erweiterbarkeit, Fehlertoleranz, Kosten, Eignung für nxn-Kommunikation (z.B. n Rechner) oder nxm-Kommunikation (z.B. n Prozessoren, m Speicher(moduln).

Bus



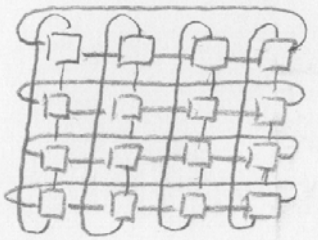
Kleine MPz,
„Snooping" erlaubt
Cache-Kohärenz

Crossbar vgl. 1.7.5

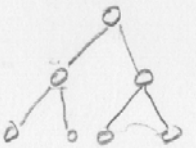


blockierungsfrei

Gitter/Torus

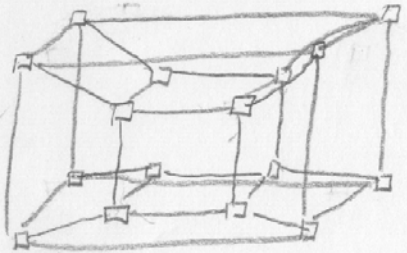


Baum

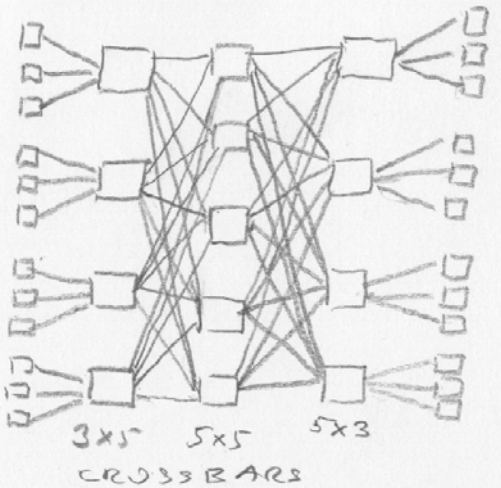


fetter Baum: Grenzdurchsatz wächst zur Wurzel

Hyperwürfel

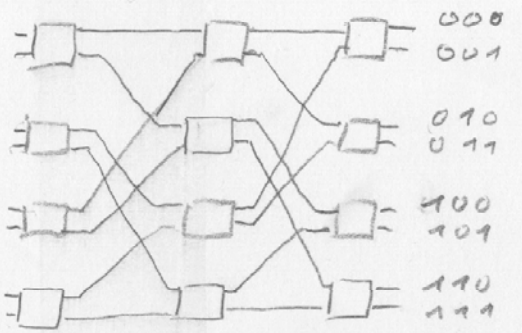


Clos-Netz



blockierungsfrei, Kosten
geringer als Crossbar

Delta-Netz (ähnlich Omega, Banyan)



nicht blockierungsfrei

Bewertung (bei k Partnern): c Grenzdurchsatz, b Bedienzeit, FT Fehlertoleranz, E Erweiterbarkeit, P Ports je Partner, c_o Grenzdurchsatz einer Leitung

Name	Stufigkeit	c (k)	b (k)	blockierd.	FT	E	P	Kosten
Bus	1	c_o	0 (1)	Ja	Nein	+	1	0 (k)
Crossbar	1	$k * c_o$	0 (1)	Nein	Nein	-	k	0 (k^2)
Gitter	0 ($\sqrt{k}$)	$< k * c_o$	0 ($\sqrt{k}$)	Ja	Ja	+	4	0 (k)
Baum	0 (log k)	$< k * c_o$	0 (log k)	Ja	Nein	+	3	0 (k)
Hyper-würfel	0 (log k)	$< k * c_o$	0 (log k)	Ja	Ja	-	log k	0 (k*log k)
Clos-Netz	3	$k * c_o$	0 (3)	Nein	Ja	-	1	0 ($k^{3/2}$)
Delta-Netz	0 (log k)	$< k * c_o$	0 (log k)	ja	Nein	-	1	0 (k*log k)

Das Clos-Netz ist nur bei geeigneter Dimensionierung nicht-blokierend.

6.3 Leistung

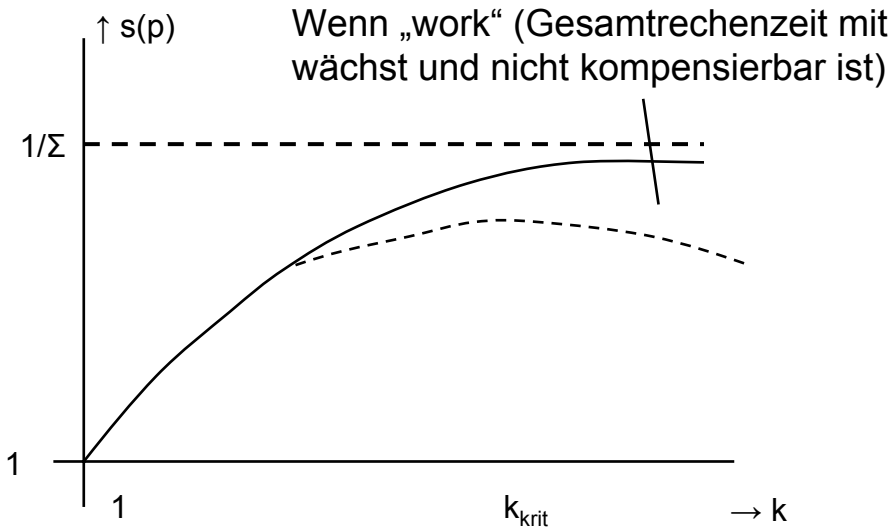
Typische Leistungsmerkmale für Parallelrechner sind Speedup und Effizienz. Speedups kennzeichnet die relative Kürzung der Bedienzeit abhängig von der Zahl k der einsetzbaren parallelen Werke (Prozessoren, Rechner). Also:

$$s = \frac{b(1)}{b(k)}$$ ideal: s=k „linearer Speedup“

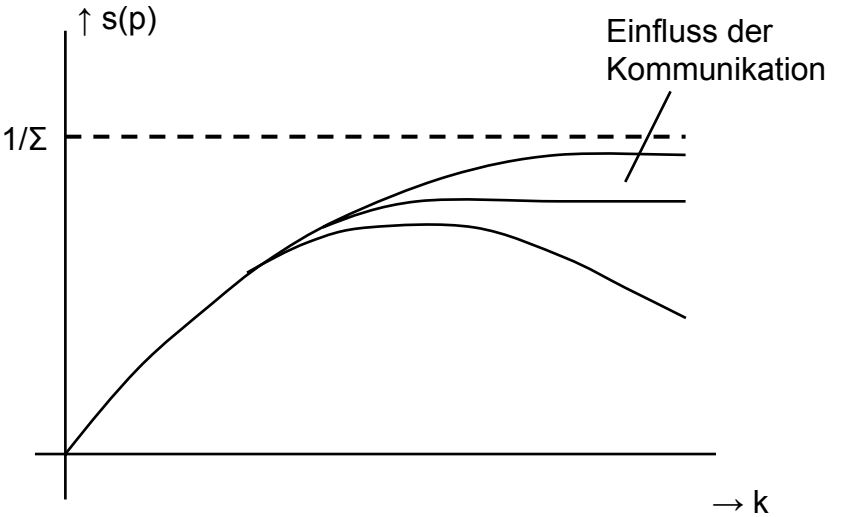
Wir kennen schon Amdahls Speedup-Formel (Anteil Σ der Rechenzeit nicht, Anteil 1-Σ ideal parallelisierbar

$$s(k) = \frac{k}{1+(1-p) * \Sigma}$$

Wächst die Gesamtrechenzeit mit k^2, dann ist sie durch Vergrößerung von k nicht mehr zu begrenzen.



Weitere Schwächung des Speedups durch Präzedenz auch im parallelisierbaren Teil und durch Kommunikation.



Effizienz kennzeichnet die Kürzung der Bedienzeit relativ zur Zahl der eingesetzten Werke, k: e=s/k.

Idealer Speedup: Effizienz 1

Begrenzter und rückläufiger Sppedup: Effizienz sinkt mit k auf Null.

Gib keinem Job mehr als k_{krit} Werke: er vergeudet sie, zu seinem eigenen Schaden!

Unterauslastung wegen Präzedenz oder Kommunikation lässt sich (grundsätzlich) durch Multiprogramming/Multithreading nutzen!

6.4 Multiprozessorsysteme

Wir nehmen als besonders aktuelles Beispiel Multicore-Multi-prozessorsysteme auf.

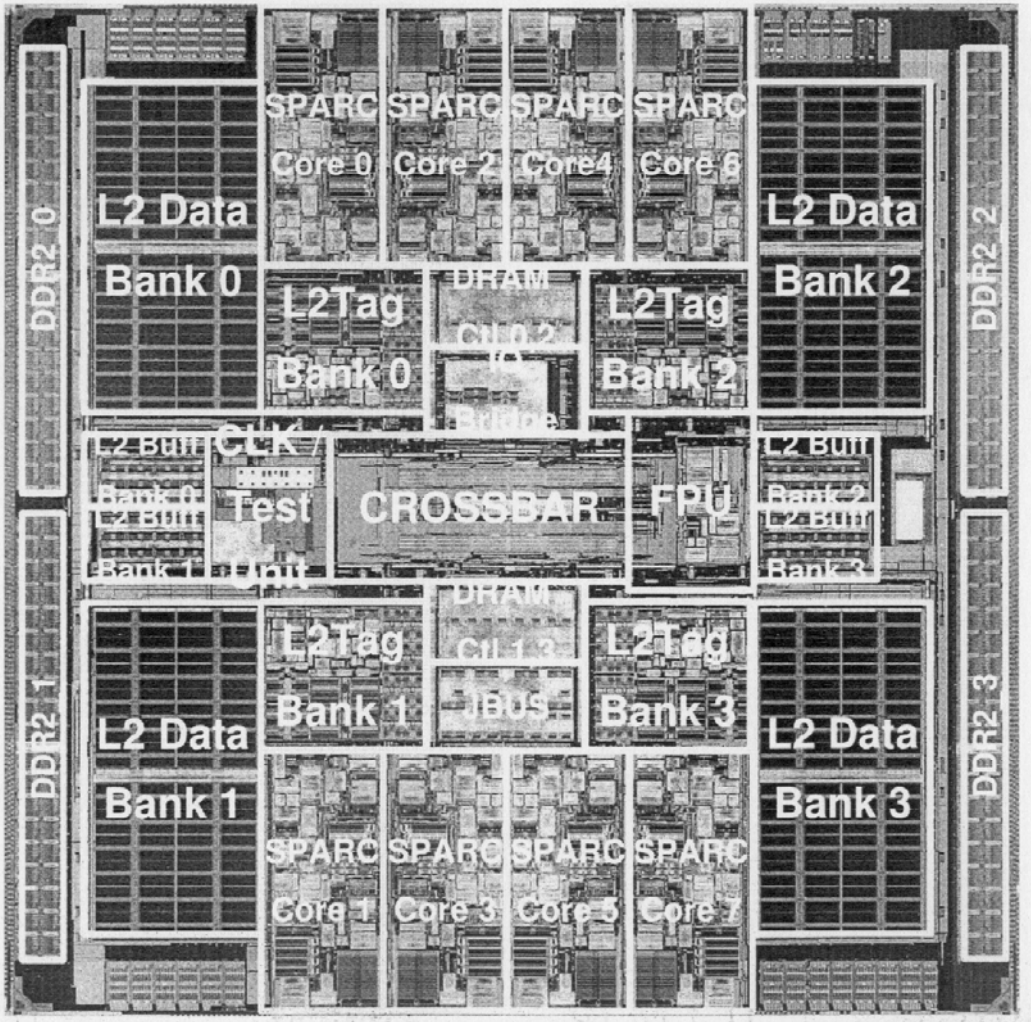
Multicore-Chips sind Chips mittlerer bis großer Transistorzahl (200.0 bis 500.0), auf denen mehrere „Cores“ realisiert sind, i.a. Prozessoren. Der Chip enthält außerdem Level 1-Cache Memories und ein gemeinsames Level 2-Cache Memory. Vorteil dieses Weges:

- Platzbedarf und Leistungsverbrauch je Prozessor sehr viel günstiger als bei Einzelprozessoren mit Level 2-Cache.
- Einfacher Entwurf (statt komplexem Hochleistungsprozessor 2/4/8 Einfachprozessoren).
- Realisierung auf dem Chip mit gemeinsamem Cache 2 erlaubt sehr verzögerungsarme, breitbandige Kommunikation der Prozessoren.

Sun Ultra SPARC T1 („Niagara“) (2005) war ein 8-Prozessor-Multicore, mit einfacher Struktur (6-stufige Pipeline) und vierfachem feinkörnigem Multithreading: in jedem Takt wird zugleich zum nächsten rechenfähigen Thread übergegangen; Wartezeiten (Cache, Sprünge, RAW-Konflikte) werden von jeweils anderen Threads genutzt. Dem Umschalten dient eine der sechs Pipeline-Stufen. Obwohl es sich um single-issue-CPUs handelt, liefern die 8 CPUs zusammen einen Befehlsdurchsatz von etwa 5/Takt (1,2 GHz Takt), vergleichbar einem komplexen Hochleistungs-prozessor.

Haupthandicap des Multicore-Ansatzes: Nebenläufige Software!

The UltraSPARC T1

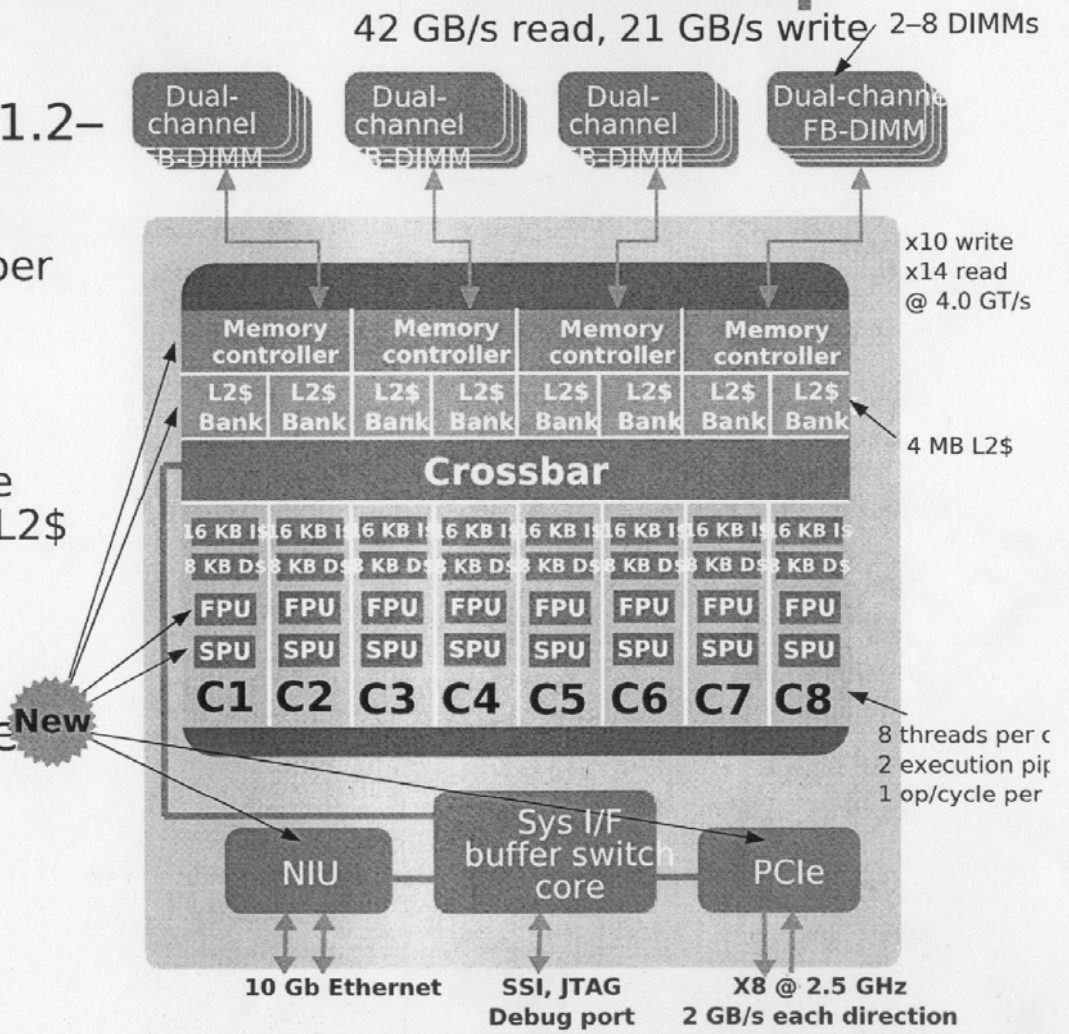


- Features:
  - 8 Cores, 4 threads/Core
  - 16KB I-Cache/Core
  - 8KB D-Cache/Core
  - Shared 3MB L2 Cache
  - 4 144-bit DDR2 channels
  - (25 GB/sec)
  - High BW Interconnect
  - Crossbar to Memory/IO
  - 3.2 GB/sec JBUS I/O
- Technology:
  - TI's 90nm CMOS
  - 9LM Cu

Sun Ultra SPARC T2 (2007) ist ein 8-Prozesor Multicore mit komplexeren Prozessorkernen (je 2 Pipelines, double issue), ebenfalls für je 4 Threads. „Server on a Chip“.

UltraSPARC T2: Server on a Chip

- 8 SPARC V9 cores @ 1.2–1.4GHz
  - > 8 threads per core
  - > 2 execution pipelines per core
  - > 1 instruction/cycle per pipeline
  - > 1 FPU per core
  - > 1 SPU (crypto) per core
  - > 4 MB, 16-way, 8-bank L2$
- 4 FB-DIMM DRAM controllers
- 2.5 GHz x 8 PCI-Expre interface
- 2 x 10 Gb on-chip Ethernet
- Technology: TI 65nm
- Die size: 342mm^2



Speicherkonsistenz

Die Prozessoren eines Multiprozessors arbeiten auf einem gemeinsamen Speicher. Die Zugriffe durchlaufen ein Verbindungswerk, wo es zu nicht vorhersehbaren Verzögerungen kommen kann. Der Zugreifer kann weder ein aktuelles Bild erwarten noch einen „Schnappschuss“ der Zugreifer-Aufträge. Aber die Hardware kann den Prozessoren bestimmte „Consistency Models“ anbieten, die wenigstens eine gewisse semantische Verlässlichkeit bieten und Grundlage brauchbarer Algorithmen sein können. Je geringer die Anforderungen sind, desto weniger Sychronisationsaufwand und desto mehr Nebenläufigkeit besteht.

Strict Consistency: Alle Zugriffe kommen in eine FCFS-Warteschlange und werden an einem Speicher normal (keine Doppel, auch nicht in Caches) abgefertigt. Nicht im 21. Jahrhundert!

Sequential Consistency: Die Hardware erledigt die Zugriffe nach ihren Möglichkeiten, es kann zu Überholungen kommen, aber die entstehende Reihenfolge ist für alle Beobachter (lesende Prozessoren z.B.) dieselbe.

Processor Consistency: Schreiboperationen jeden Prozessors werden von jedem in derselben Reihenfolge gesehen, in der sie beauftragt wurden. An jedem Speicherwort sehen alle dieselbe Schreibreihenfolge (wer auch immer dazu beigetragen hat).

Weak Consistency: Keine Garantie für eine bestimmte Reihenfolge der Ausführung von Zugriffen oder gleicher Bilder für alle Zugreifer. Aber es gibt Synchronisationsoperationen, die alle hängenden Schreiboperationen erzwingen, bevor weitere ausgeführt werden. Und die Synchronisations-operationen sind in Sequential Consistency.

Release Consistency: Das Modell der Weak Consistency wird eingeengt auf Variable für die ein kritischer Abschnitt vereinbart ist. Er kann erst betreten werden, wenn alle hängenden Schreib-operationen dieser Variablen erledigt sind. Dann kann der eingetretene Prozess beliebig lesen und schreiben; seine Schreiboperationen müssen erst beendet sein, bevor wieder ein Prozess in den kritischen Abschnitt eintreten darf.

6.5 Multirechner

Auch „Distributed Memory System“ genannt (gemeint wie „Verteiltes System“, d.h. kein eindeutiger übergreifender Zustand) oder „Messages Passing Multicomputer“.

Schon gehabt:

- n Betriebssysteme
- für große n geeignet (10^4, bald 10^5): Petaflop-Computer
- MPI, PVM als Programmierschicht
- konzentrierte Multirechner: Hochleistungsmaschinen, vgl. SGI Altix 4700 (→ 1.6): 9728 CPUs, 62,3 TFlops, Intel Itanium, 39 TBytes Hauptspeicher, 1 MW, 100t, 38 M€, Nr. 18 unter den Top 500 (Nov. 2007). Ganzheitlicher Adressraum (d.h. Multiprozessor), gegliedert aber in 19 SMP-Partitionen, verbunden über Fat Tree, Linux
- Cluster (typisch Linux, wesentliches Material für Grids)
- Grids

6.6 Grids

Grids sind Verteilte Systeme aus Rechnern und Instrumenten (z.B. Beschleuniger) und Speichern, für

- gemeinschaftliche Nutzung von Geräten, Daten und Software
- Unterstützung der Zusammenarbeit
- Unterstützung Virtueller Organisationen (organisationsübergreifende Zweckverbindungen für Gridnutzung, z.B. ein Projekt).

Die Betriebsmittel, Benutzerschaft, Virtuellen Organisationen sind dauerndem Wechsel unterworfen.

Typisch für Grids sind

- heterogene Betriebsmittel
- geographische Verteilung
- Betriebsmittel und Nutzer aus verschiedenen Einrichtungen und Managementdomänen
- Orientierung an Service-Orientierter Architektur (SOA).

„Grid“ ist Assoziation zu „Power Grid“ (Elektrizitätsnetz).

Zweck der Grids

- Betriebsmittel des Benutzers nach Bedarf aufweiten (quantitativ und funktional), Back-up
- Heterogenität überwinden: Grid-Middleware virtualisiert
- Zusammenarbeit fördern
- Vorteile von SOA (einfache Implementierung von Anwendungen aus vorgefertigten Diensten verschiedenster Herkunft) nutzen
- Einrichtungsübergreifende Sicherheit

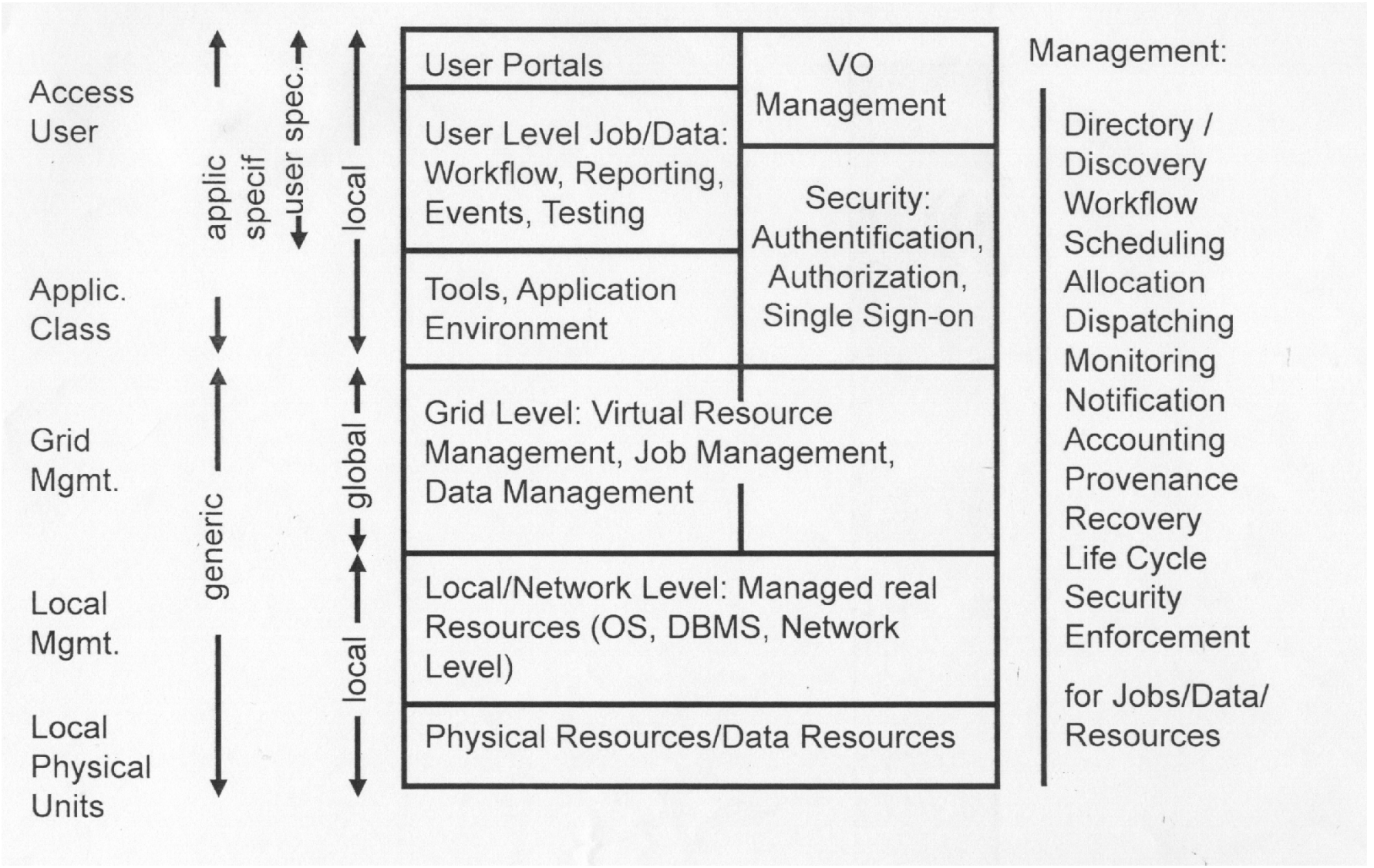
Seit einiger Zeit Einfachformen:

- on demand Computing (IBM, Sun, andere)
- Cloud Computing (Amazon, Google und andere)

Probleme:

Standards, Abrechnungsformen, Sicherheit, Anreize Betriebsmittel abzugeben.

Schichtung der Grid-Architekturen:



Rückblick auf den 16.10.2007

- Architektur (Strukturlehre, Funktionalität)
- Funktionseinheiten, Aufträge
- Verweilzeit, Füllung, Bedienzeit, Kapazität, Durchsatz, Auslastung, Little‘s Formel
- Kostenrate, mittlere Auftragskosten, Tarifierung
- Präzedenz, sequentielle/nebenläufige, seriell/kollateral (parallel)
- Zuverlässigkeit, Fehler, Störung, Ausfall, MTBM/F, Verfügbarkeit
- Fehlertoleranz
- Sicher (kein Missbrauch möglich), Sicher (unschädlich)

Rückblick auf den 23.10.2007

von Neumann: Wirklich neu war nur die Beeinflussung des Rechenprozesses durch sich selbst (aber erstrangig!).

Aber heute: Speicherhierarchie, Interpretationshierarchie, Unterbrechung, Multiprozessor/Multirechnersysteme, Programm wird nicht mehr seriell ausgeführt, Programm ist invariant (dafür bedingte Sprünge und Adressmodifikationen), Alternativen (Datenfluss/Reduktionsmaschine) haben sich nicht durchgesetzt, wohl aber Parallelrechner.

Interpretation und Übersetzung.

Aktuelle Ausschnitte: Kosten/Zugriffszeit/Kapazitäts-Gesetze.

Räumliche/zeitliche Lokalität: Lade/Verdrängungsstrategie.

Trefferhäufigkeit h.

Wie findet man etwas im Ausschnitt?

Kohärenzproblem bei mehreren Zugreifern mit eigenem Ausschnitt.

Rückblick auf den 30.10.2007

Rechnertypen und Markt: Allgemeiner Markt (Taschenrechner, Mobiltelefon, Kameras, PDAs, PCs, Notebooks, Workstations, Server, Mainframes, Supercomputer), Embedded Computers.

Sun Sunfire Servers mit Sun Ray „Thin“ Client.

SGI Altix 4700 Superrechner: 9728 mal Intel Itanium 2.

Kommunikationsstrukturen: Duplex/simplex, seriell/parallel, Leitungstechnologie, zugeordnet/gemeinsam, Frequenz/Zeitmultiplex

Vermittlung (Nachrichten-, Leitungs-)

Ethernet, Bus, Fernnetz, Gitter (Torus), Baum, Würfel

Rückblick auf den 6.11.2007

Kommunikationsinfrastrukturen: Vermittelt, rechnerintern z.B. Gitter, Baum, Würfel; Crossbar

Leistungsbewertung: Ziele

Statische Hardwareparameter

Laufzeitmessung (Benchmarks), SPEC

Monitoring

Modelle (analytisch/simulativ, stochastisch/operational

Geschichte der Rechner: bis 1955

Rückblick auf den 13.11.2007

Entwicklung der Rechner:

Bis 1941: (Turing, Church), (Fernsehen, Magnetband), Zuse Z 3

Bis 1951: von Neumann, elektronische Rechner, Univac

Bis 1961: Systemsoftware, Netz, Transistorrechner, Industrie

Bis 1971: Mehrprogramm/Teilnehmerbetrieb, priveligierte Funktionen, Mehrprozessor/rechner Systeme, Mikroprozessor, Cache, UNIX

Bis 1981: Workstation, Feldrechner, PC, Internet, lokale Netze, Vektorrechner, Apple Lise, IBM PC, RISC

Bis 1991: Connection Machine, Viren, Laptop, Palmtop

Bis 2001: WWW, Parallelrechner, Java, 10^9 Computer

Danach: Einchipprozessor stößt an Leistungsgrenzen; „Multicores“

Wachstumsfaktoren: Technologie, Anzahl: 1,4 .. 1,6/ Jahr (aber Zugriffszeiten nur 1,1/Jahr), Softwarekomplexität 1,25/Jahr

Rückblick auf den 27.11.2007

Freiheitsgrade:

Ausführungsmodell (deklarativ, funktional, prozedural)

Bindungszeitpunkt (Programmierer, Compiler, Betriebssystem, Prozessor)

Nebenläufigkeit und Parallelität

- Zweck: Bearbeitungszeit, Auslastung, selektive Strategie
- Betriebsmittelkonflikte, Respektierung von Schreib-/Lese-Präzedenz (Regel!)
- Beispiele: Pseudosequentielles Programm: ILP, SIMD, explizit parallele Algorithmen, Parallelität von Therads/Prozessen/Virtuellen Maschinen

Rückblick auf den 29.11.2007

Zentralprozessoren:

Pentium: Geht auf 1972 zurück, Weg über IA-16, IA-32, Intel 64.

Ca. 600 Befehle, komplexes Format, Umsetzung auf RISC-Basismaschine, 1-3 GHz, 100W, 16+16 Register, 5 Pipelines, Multicore

CISC-RISC-Wende: Auslöser: Hohe ISA-Komplexität, vom Compiler nicht genutzt, unzureichende Integrationsfähigkeit, verbilligte Hauptspeicher, einfaches Betriebsbild auf Workstations. Resultat: Gleichstrukturierte Befehle (Pipeline!), Operanden und Ergebnisse nur in Registern (load/store Befehle notwendig)

Sun Ultra SPARCIII: 64b, 128 Register, Fenster-Mechanismus, 92 Befehle, 8 Pipelines; Multicoreformen T1/T2 mit 8 Cores

Rückblick auf den 7.12.2007

Itanium (3. Prozessorbeispiel):

Compiler erzeugt Nebenläufigkeit (Gruppe) und legt Parallelverarbeitung explizit fest („Bündel“, EPIC“). Empfehlungen des Compilers zu Sprungkriterium, Sprungziel, vorsorglichem Laden. Bedingungsbits (Predicates) für bedingte Sprünge. Laufzeitstapel in Registern. 11 Ausführungseinheiten, mäßig tiefe (8) Pipelines. Software Pipelining durch automatische Registerzuteilung und Bedingungsbits unterstützt.

Grundstrukturen:

Anwachsender Prozessorstatus, Privilegierung, Einschränkung der Benutzerprozesse (Befehle, Befehlswirkungen, Unterbrechungen)

Rückblick auf den 11.12.2007

Interpretationsmodi des Prozessors: Modi, Übergänge, virtuelle Maschinen und deren Anforderungen.

Datentypen: Bits, Bitfeld, Byte, Bit/Bytestring, Dezimalzifferstring, Festpunkt/Gleitpunktzahlen, Stack(unterstützung), (Befehle), Adressen, Deskriptoren, Zugriffsausweise, Typenkennung. Abbildung auf den Speicher: Little/Big Endians, Alignment. Und die Akademie von Laputa.

Befehle: Festes/variables Format, Register, Drei-Adress-Prinzip

Rückblick auf den unglücklichen 18.12.2007

Befehle: Format, Häufigkeit, Befehlslisten Pentium, UltraSPARC.

Untersuchung der Befehlsstruktur zweistelliger Operationen (d.h. 3 Orte):
- compilergünstig: 3 Adr.M., 0 Adr.M.
- Register unabdingbar (Durchsatz, Platzbedarf)
- heute überwiegen 3/2-Registerbefehle

Allgemeine Register:
- Pentium 8 (gewidmet) + 8 (frei)
- UltraSPARC III (128, aber Fenster (Stack) Mechanismus, 32 eingeschränkt frei)
- Itanium (128, Fenstermechanismus, Fenster von Prozedur einstellbar)

Transporte (50%)
Sprünge (25%)
- unbedingte/bedingte (Condition Code oder Register)
- Sprungzielmodifikation
- Hin/Rücksprung in Prozeduren/in Betriebssystem

Rückblick auf den 8.01.2008

Befehle:

Unterprogramm/System/Unterbrechungs-Aufruf und -Rückkehr

E/A: Sonderbefehle oder (häufiger) „memory mapped i/o“

Adressierung:

Werte statt Adressen (immediate operand, direct operand). Von Programmadressen zu Prozessadressen (effektive Adresse):

- Substitution
- Additive Korrekturen:
  -- fallweise: „Index“ (indizierte Variable, Unterprogrammparameter (mit Substitution für Referenzparameter), …
  -- immer: relativ zu Basisadressen oder zum Befehlszähler (problematisch: beides erlaubt Kurzadressen und unterstützt/ erübrigt das Binden (linking).

Rückblick auf den 15.1.2008

Von Prozessadressen zu Maschinenadressen:
Verschiebbarkeit im Speicher (anfänglich, später), gestreute Speicherung (Speichernutzung, Kommunikation), partielle Speicherung (aktueller Ausschnitt, „Virtueller Speicher“). Lader, Basisadressen/befehlszählerrelative Adressierung, verdeckte Basisadressierung, Seitenadressierung.

Mehrfache Adressräume je Prozess:
Segmente (Rechte, Zugriffstyp, evtl. auch Anordnung und partielle Speicherung; auch in Kombination mit Seitenadressierung).

Unterbrechungen:
Unterbrechungswunsch, akzeptiert?, Prozessstatus retten, Anfangsbehandlung, Einsetzung eines Prozesses.
Quellen: aktueller Prozess, andere Vorgänge. Wo wieder aufsetzen nach Unterbrechung?

Rückblick auf den 22.1.2008

(noch) Unterbrechungen: Priorisierung, Maskierung

Rechenwerk: Festpunkt/Gleitpunkt, Bereich/Genauigkeit, positive/negative Zahlen, Grundrechenarten

Spezialprozessoren „Microcontroller“, Vektorrechner, Graphikprozessoren, Signalprozessoren

Übergang ISA → Mikromaschine:
- Mikroprogrammierung
- befehlsweise Übersetzung auf RISC (Pentium, Athlon)
- basisblockweise Übersetzung auf EPIC (Transmeta Crusoe)
- „ISA ist Mikromaschine“ (RISC)

Cache Memory:
- ohne Cache und Parallelverarbeitung Zentraleinheits-Durchsatz ungenügend
- 3-stufige Cache-Hierarchie, Cache 1 doppelt

Rückblick auf den 29.1.2008

Cache Memory: Zugriffszeit und Speicher/Bus-Durchsatz! 3-stufig, Platzierung: voll-assoziativ, direktabbildend, Z-Gruppen assoziativ, non-blocking Cache, Schreibpuffer; Start/Capacity/Conflict Misses. Schreiben: write allocate, write around; Kohärenz: write through/copy back.

Parallelität in der Mikromaschine:
Nebenläufigkeit: Code Motion, Schattenregister, bedingte Befehle, Poison-Bits, Multithreading.
Mehr Betriebsmittel! Bessere Zeitparameter (Cache, Sprungzielcache)

Sprünge: undbedingt/bedingt/modifiziert. Sprungvorhersage: stabil, dynamisch
Pipeline: ideal $c=k/b_{op}$, aber Sättigung durch Stufenübergabe; k= 5 .. 20.

Alle obigen Maßnahmen sind wichtig!