64-040 Modul IP7: Rechnerstrukturen (RS)

http://tams-www.informatik.uni-hamburg.de/lectures/2009ws/vorlesung/rs

Norman Hendrich, Jianwei Zhang

Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Department Informatik
Technische Aspekte Multimodaler Systeme

Wintersemester 2009/2010

Gliederung

1. Wiederholung: Software-Schichten
2. Instruction Set Architecture (ISA)
   - Speicherorganisation
   - Befehlszyklus:
   - Befehlsformat: Einteilung in mehrere Felder
   - Adressierungsarten
3. x86-Architektur
4. Assembler-Programmierung
   - Assembler und Disassembler
   - Einfache Addressierungsmodi (Speicherreferenzen)
   - Arithmetische Operationen
   - Kontrollfluss
5. Assembler-Programmierung

Gliederung (cont.)

Gliederung (cont.)

Festplatten
Cache Speicher
Virtuelle Speicher
DRAM als Cache
Virtueller Speicher: Speicherverwaltung
Virtuelle Speicher: Schutzmechanismen
Multi-Ebenen Seiten-Tabellen
Zusammenfassung der virtuellen Speicher
Das Speichersystem von Pentium und Linux
Zusammenfassung Speichersystem
RAID
Optisches Speichermedium CD-ROM

9. I/O: Ein- und Ausgabe

Gliederung (cont.)

- Busse
- Unterbrechungen
- DMA (“Direct Memory Access”)

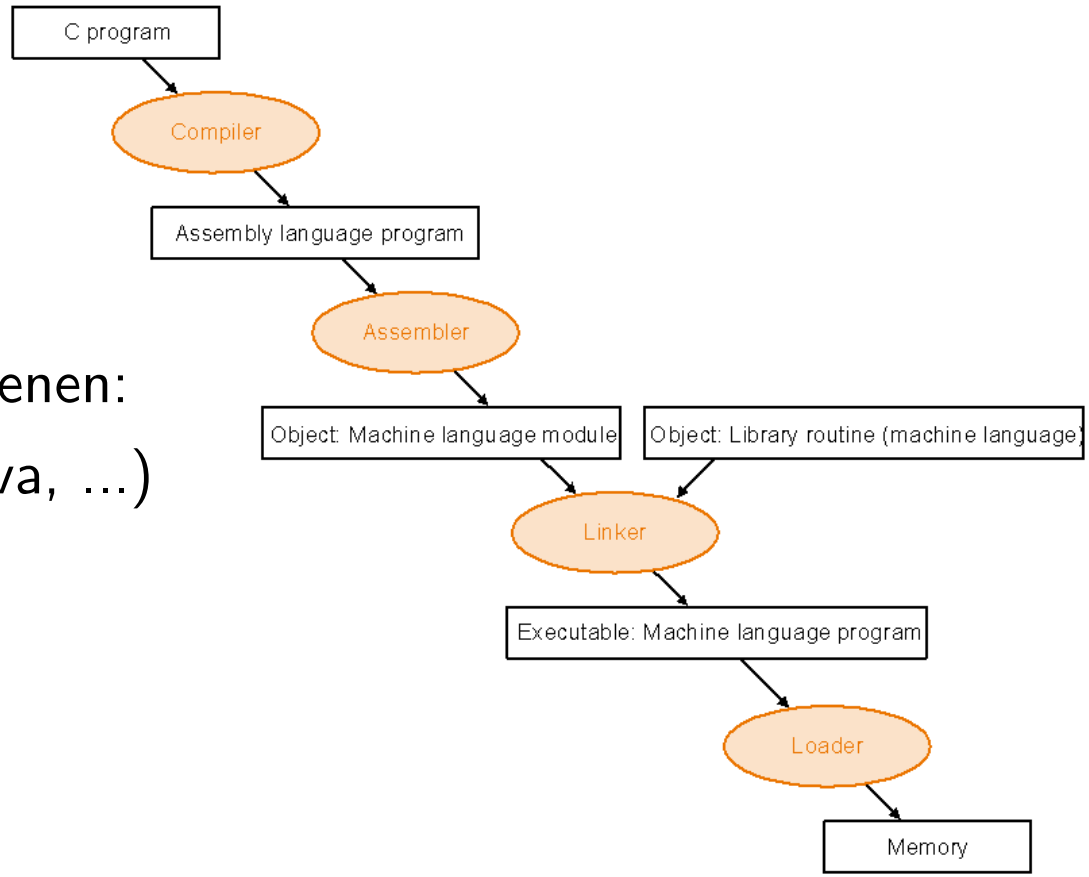
10. Ausnahmebehandlungen und Prozesse
    - Kontrollfluss
    - Exceptions
    - Synchrone Exceptions
    - Prozesse
11. Parallelrechner

Gliederung

Wiederholung: Software-Schichten

mehrere Abstraktionsebenen:

- Hochsprache (C, Java, ...)
- Assembler
- Maschinencode



Gliederung

Gliederung (cont.)

Artenvielfalt der Architekturen









Prozessor	4 .. 32 bit	8 bit	–	16 .. 32 bit	32 bit	32 bit	32 bit	8 .. 64 bit	..32 bit
Speicher	1K .. 1M	< 8K	< 1K	1 .. 64M	1 .. 64M	< 512M	8 .. 64M	1 K .. 10 M	< 64 M
ASICs	1 uC	1 uC	1 ASIC	1 uP ASIP	DSPs	1 uP, 3 DSP	1 uP, DSP	~ 100 uC, uP, DSP	uP, ASIP
Netzwerk	cardIO	–	RS232	diverse	GSM	MIDI	V.90	CAN,...	I2C,...
Echtzeit	nein	nein	soft	soft	hard	soft	hard	hard	hard
Safety	keine	mittel	keine	gering	gering	gering	gering	hoch	hoch

=> riesiges Spektrum: 4 bit .. 64 bit Prozessoren, DSPs, digitale/analoge ASICs, ...

=> Sensoren/Aktoren: Tasten, Displays, Druck, Temperatur, Antennen, CCD, ...

=> Echtzeit−, Sicherheits−, Zuverlässigkeitsanforderungen

Instruction Set Architecture (ISA)

- HW/SW-Schnittstelle einer Prozessorfamilie
- charakteristische Merkmale:
  - Rechnerklasse (Stack-/Akku-/Registermaschine)
  - Registersatz (Anzahl und Art der Rechenregister)
  - Speichermodell (Wortbreite, Adressierung, ...)
  - Befehlssatz (Definition aller Befehle)
  - Art, Zahl der Operanden (Anzahl/Wortbreite/Reg./Speicher)
  - Ausrichtung der Daten (Alignment/Endianness)
  - I/O-, Unterbrechungsstruktur (Ein- und Ausgabe, Interrupts)
  - Systemsoftware (Loader/Assembler/Compiler/Debugger)

Beispiele für charakteristische ISA

(in dieser Vorlesung bzw. im Praktikum behandelt)

- MIPS (klassischer 32-bit RISC)
- x86 (CISC, Verwendung in PCs)
- D*CORE („Demo Rechner“, 16-bit)

Speicherorganisation

- Wortbreite, Grösse (=Speicherkapazität)
- „little-/big-endian“
- „Alignment“
- „Memory-Map“
- Beispiel PC

- spätere Themen:
  - Cache-Organisation für schnelleren Zugriff
  - Virtueller Speicher für Multitasking
  - MESI-Protokoll für Multiprozessorsysteme
  - Synchronisation in Multiprozessorsystemen

Speicher: Wortbreite

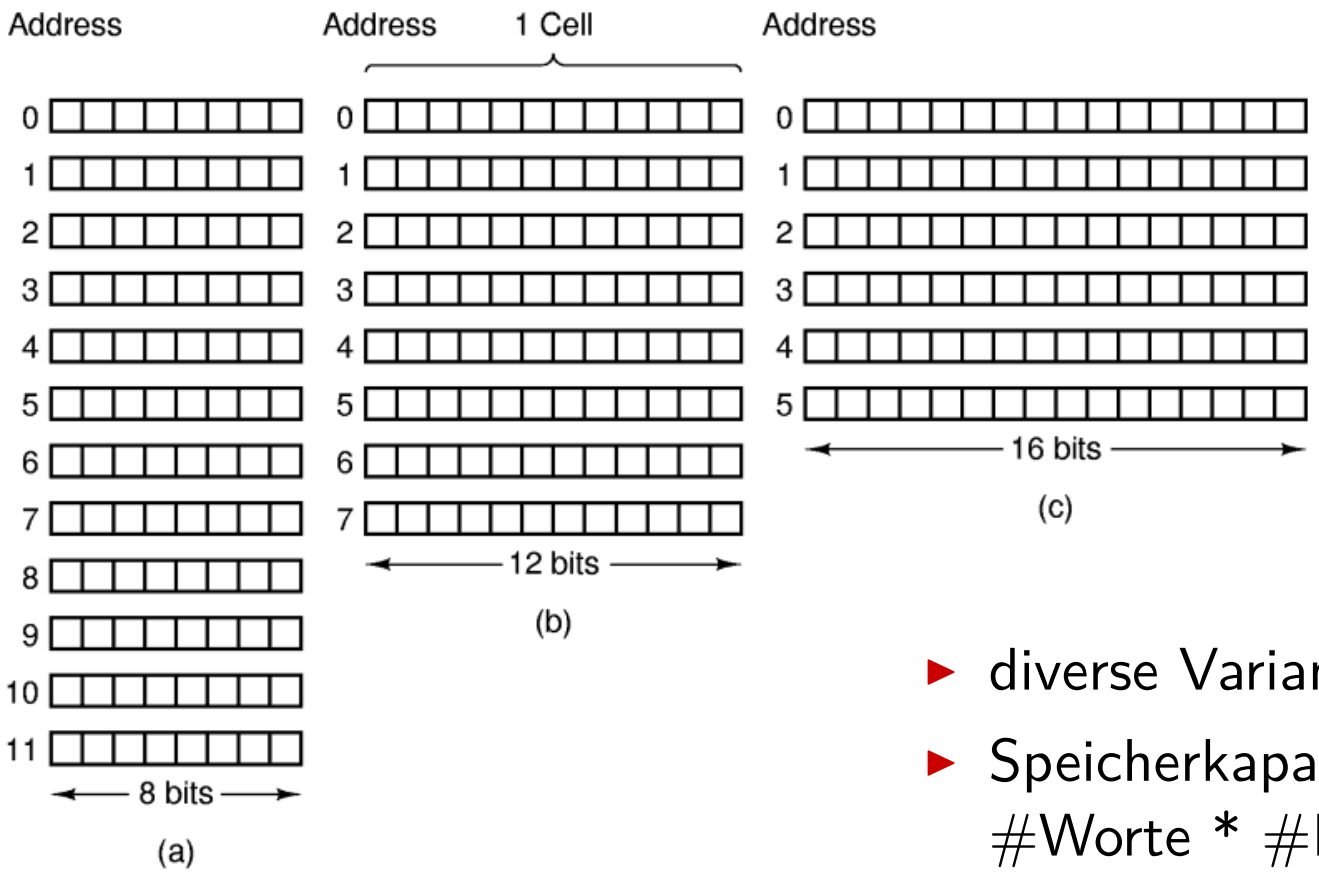
Adressierbare Speicherwortbreiten einiger historisch wichtiger Computer

Computer	Bits/cell
Burroughs B1700	1
IBM PC	8
DEC PDP-8	12
IBM 1130	16
DEC PDP-15	18
XDS 940	24
Electrologica X8	27
XDS Sigma 9	32
Honeywell 6180	36
CDC 3600	48
CDC Cyber	60

- heute vor allem 8/16/32/64-bit Systeme
- erlaubt 8-bit ASCII, 16-bit Unicode, 32-/64-bit Floating-Point
- Beispiel Intel x86: „byte“, „word“, „double word“, „quad word“

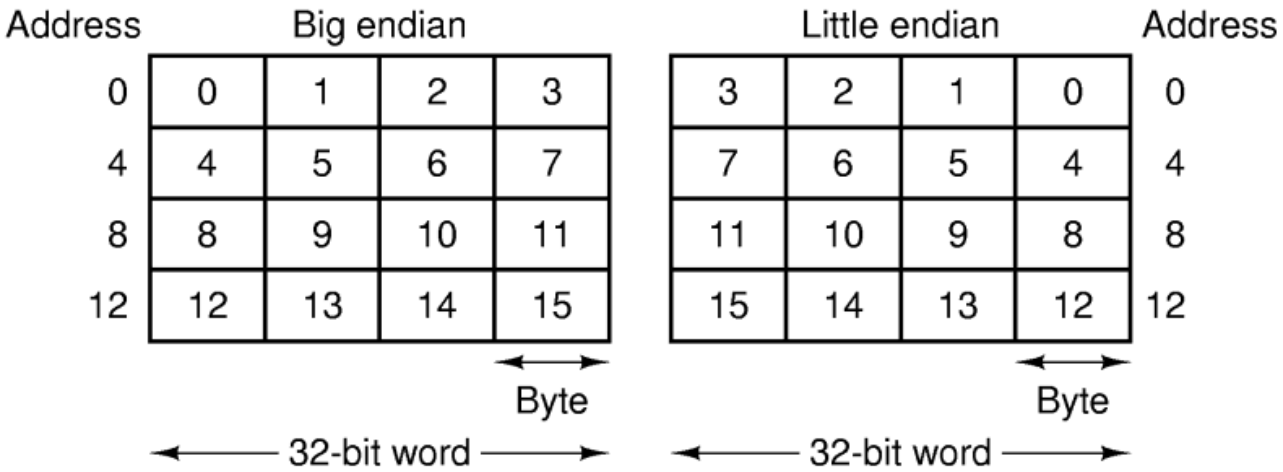
Hauptspeicher: Organisation

Drei Organisationsformen eines 96-bit Speichers



- diverse Varianten möglich
- Speicherkapazität:
  #Worte * #Bits/Wort

Big- vs. Little Endian

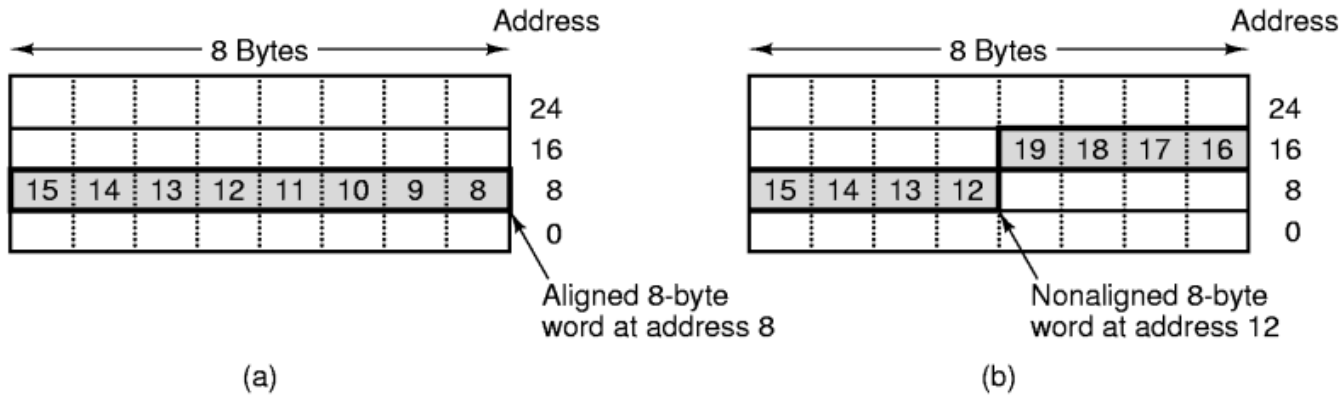


- Anordnung einzelner Bytes in einem Wort (hier 32-bit)
  - big-endian LSB ... MSB Anordnung, gut für Strings
  - little-endian MSB ... LSB Anordnung, gut für Zahlen

- beide Varianten haben Vor- und Nachteile
- komplizierte Umrechnung

Misaligned Access

- Beispiel: 8-Byte-Wort in einem little-Endian Speicher
  (a) „aligned“ bezüglich Speicherwort
  (b) „nonaligned“ an Byte-Adresse 12



- Speicher wird (meistens) Byte-weise adressiert
- aber Zugriffe lesen/schreiben jeweils ein ganzes Wort
- was passiert bei „krummen“ (misaligned) Adressen?
  - automatische Umsetzung auf mehrere Zugriffe (x86)
  - Programmabbruch (MIPS)

„Memory Map“

- CPU kann im Prinzip alle möglichen Adressen ansprechen
- aber nicht alle Systeme haben voll ausgebauten Speiche (32-bit Adresse entspricht bereits 4GB Hauptspeicher...)
- Aufteilung in RAM und ROM-Bereiche
- ROM mindestens zum Booten notwendig
- zusätzliche Speicherbereiche für „memory mapped“ I/O

- „Memory Map“
  - Zuordnung von Adressen zu „realem“ Speicher
  - Aufgabe des Adress-„Dekoders"
  - Beispiel: Windows

Memory Map: typ. 16-bit System

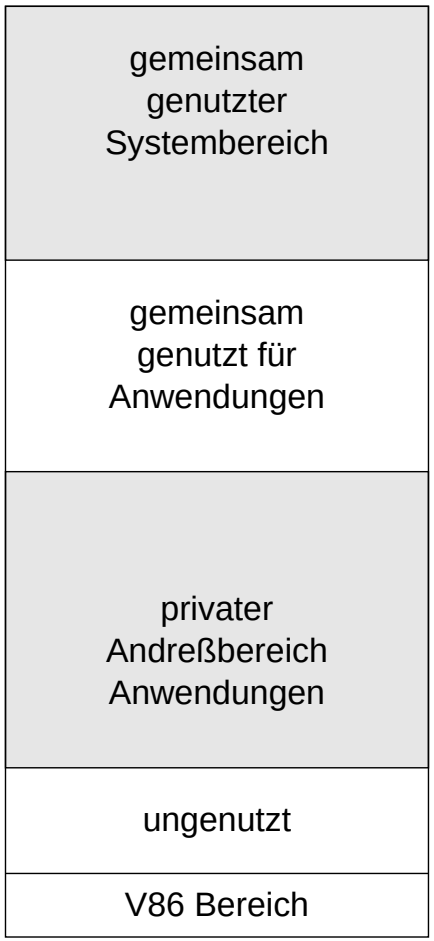
- 16-bit erlaubt 64K Adressen: 0x0000 .. 0xFFFF

- ROM-Bereich für Boot / Betriebssystemkern
- RAM-Bereich für Hauptspeicher
- RAM-Bereich für Interrupt-Tabelle

- I/O-Bereiche für serielle / parallel Schnittstellen
- I/O-Bereiche für weitere Schnittstellen

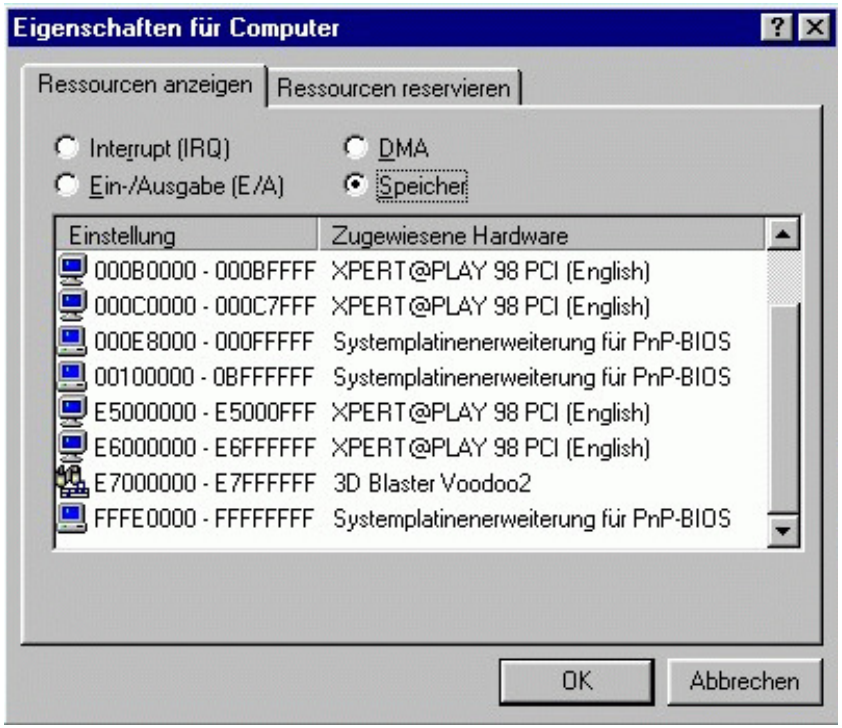
- Demo und Beispiele später

PC: Windows 9x Speicherbereiche



- DOS-Bereich immer noch für Boot / Geräte (VGA) notwendig
- Kernel, Treiber, usw. im oberen 1 GB-Bereich

PC: Speicherbereiche, Beispiel



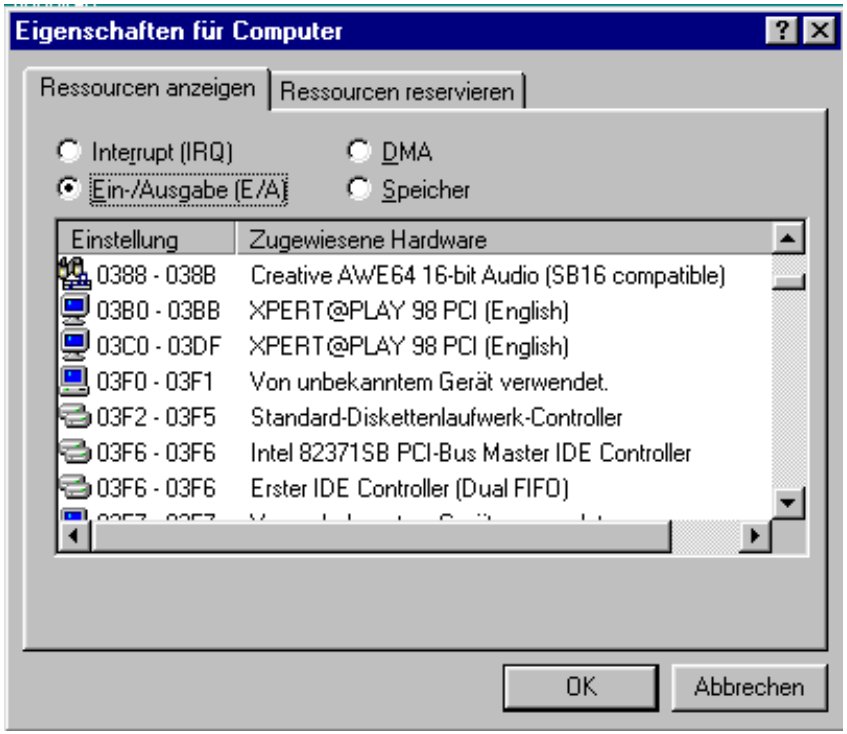
nutzbarer Hauptspeicher oberhalb 1 MB

Speicherbereiche für Systemaufgaben (hier Framebuffer der Graphikkarten)

BIOS (ROM) am oberen Ende des Adressbereichs

- Windows 9x erlaubt bis 4 GByte Adressraum
- Adressen 00000000h bis ffffffffh
- Aufteilung 1 GB / 1 GB / 2 GB

PC: IO-Adressen, Beispiel



- I/O-Adressraum gesamt nur 64 KByte
- je nach Zahl der I/O-Geräte evtl. fast voll ausgenutzt
- eingeschränkte Autokonfiguration über PnP-BIOS

Befehlszyklus:

- Prämisse: von-Neumann Prinz
  - Daten und Befehle im gemeinsamen Hauptspeicher
- Abarbeitung des Befehlszyklus in Endlosschleife
  - Programmzähler PC addressiert den Speicher
  - gelesener Wert kommt in das Befehlsregister IR
  - Befehl dekodieren
  - Befehl ausführen
  - nächsten Befehl auswählen
- minimal benötigte Register:

PC	program counter	Adresse des Befehls
IR	instruction register	aktueller Befehl
R0..R31	registerbank	Rechenregister (Operanden)

Beispiel: Boot-Prozess

Was passiert beim Einschalten des Rechners?

- Chipsatz erzeugt Reset-Signale für alle ICs
- Reset für die zentralen Prozessor-Register (PC, ...)
- PC wird auf Startwert initialisiert (z.B. 0xFFFF FFEF)
- Befehlszyklus wird gestartet

- Prozessor greift auf die Startadresse zu
- dort liegt ein ROM mit dem Boot-Programm
- Initialisierung und Selbsttest des Prozessors
- Löschen und Initialisieren der Caches
- Konfiguration des Chipsatzes
- Erkennung und Initialisierung von I/O-Komponenten

- Laden des Betriebssystems

Instruction Fetch

„Befehl holen“ - Phase im Befehlszyklus

- Programmzähler (PC) liefert Adresse für den Speicher
- Lesezugriff auf den Speicher
- Resultat wird im Befehlsregister (IR) abgelegt
- Programmzähler wird inkrementiert

- Beispiel für 32-bit RISC mit 32-bit Befehlen:

$$\begin{aligned} IR &= MEM[PC] \\ PC &= PC + 4 \end{aligned}$$

- bei CISC-Maschinen evtl. weitere Zugriffe notwendig, abhängig von der Art (und Länge) des Befehls

Instruction Decode

„Befehl dekodieren“ - Phase im Befehlszyklus

- Befehl steht im Befehlsregister IR
- Decoder entschlüsselt Opcode und Operanden
- leitet Steuersignale an die Funktionseinheiten
- Programmzähler wird inkrementiert

Instruction Execute

„Befehl ausführen“ - Phase im Befehlszyklus

- Befehl steht im Befehlsregister IR
- Decoder hat Opcode und Operanden entschlüsselt
- Steuersignale liegen an Funktionseinheiten

- Ausführung des Befehls durch Aktivierung der Funktionseinheiten

- Details abhängig von der Art des Befehls
- Ausführungszeit abhängig vom Befehl

- Realisierung über festverdrahtete Hardware
  oder mikroprogrammiert

- Demo (bzw. im T3-Praktikum):
  - Realisierung des Mikroprogramms für den D*CORE

Welche Befehle braucht man?

Befehls-„Klassen“:	Beispiele:
• arithmetische Operationen logische Operationen Schiebe-Operationen	add, sub, mult, div and, or, xor shift-left, rotate
• Vergleichsoperationen	cmpeq, cmpgt, cmplt
• Datentransfers	load, store, I/O
• Programm-Kontrollfluß	jump, branch call, return
• Maschinensteuerung	trap, halt, (interrupt)

CISC

„Complex instruction set computer“

- Bezeichnung für Computer-Architekturen mit irregulärem, komplexem Befehlssatz
- typische Merkmale:
  - sehr viele Befehle, viele Datentypen
  - komplexe Befehlskodierung, Befehle variabler Länge
  - viele Adressierungsarten
  - Mischung von Register- und Speicheroperanden
  - komplexe Befehle mit langer Ausführungszeit
  - Problem: Compiler benutzen solche Befehle gar nicht
- Beispiele: Intel 80x86, Motorola 68K, DEC Vax

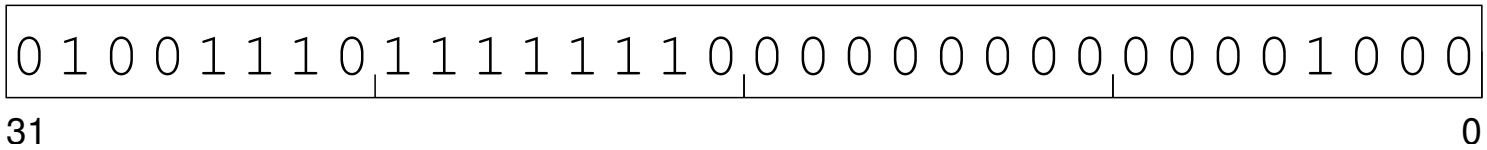
RISC

„reduced instruction set computer“

- Oberbegriff für moderne Rechnerarchitekturen entwickelt ab ca. 1980 bei IBM, Stanford, Berkeley
- auch bekannt unter: „regular instruction set computer“
- typische Merkmale:
  - reguläre Struktur, z.B. 32-bit Wortbreite, 32-bit Befehle
  - nur ein-Wort-Befehle
  - alle Befehle in einem Taktschritt ausführbar
  - „Load-Store“ Architektur, keine Speicheroperanden
  - viele universielle Register, keine Spezialregister
  - optimierende Compiler statt Assemblerprogrammierung
- Beispiele: IBM 801, MIPS, SPARC, DEC Alpha, ARM
- Diskussion und Details CISC vs. RISC später

Befehls-Dekodierung

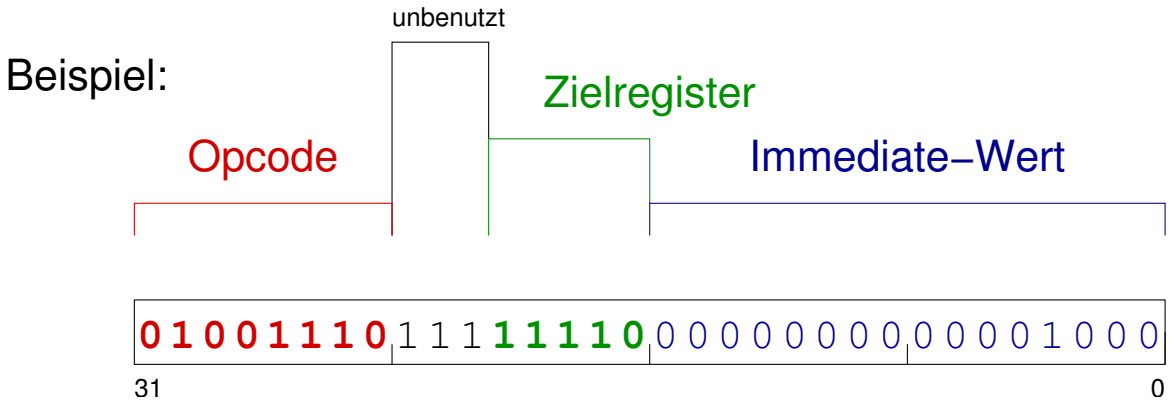
- Befehlsregister IR enthält den aktuellen Befehl
- z.B. einen 32-bit Wert



- Wie soll die Hardware diesen Wert interpretieren?
  - direkt in einer Tabelle nachschauen (Mikrocode-ROM)
  - Problem: Tabelle müsste 2^{32} Einträge haben
    - deshalb Aufteilung in Felder: Opcode und Operanden
    - Dekodierung über mehrere, kleine Tabellen
    - unterschiedliche Aufteilung für unterschiedliche Befehle:

  $\Rightarrow$ „Befehlsformate“

Befehlsformat: Einteilung in mehrere Felder



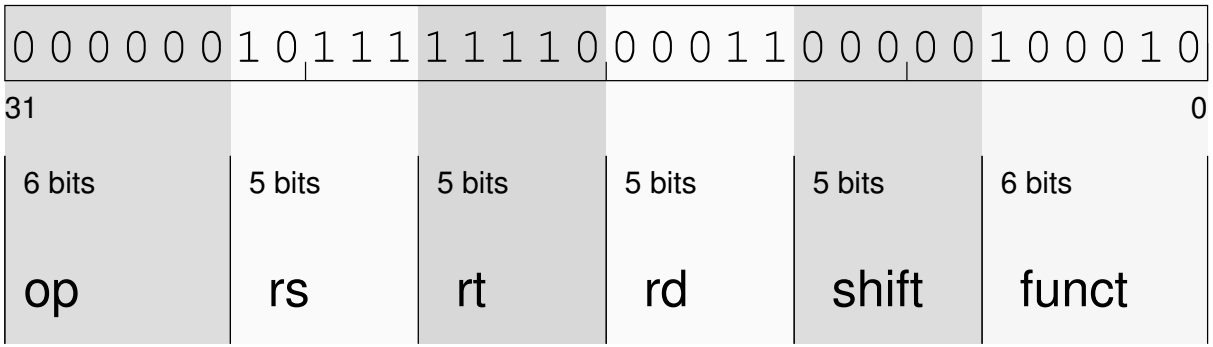
- Befehls„format“: Aufteilung in mehrere Felder:
  - Opcode — eigentlicher Befehl
  - ALU-Operation — add/sub/incr/shift/usw.
  - Register-Indizes — Operanden / Resultat
  - Speicher-Adressen — für Speicherzugriffe
  - Immediate-Operanden — Werte direkt im Befehl
- Lage und Anzahl der Felder abhängig vom Befehlssatz

Befehlsformat: Beispiel MIPS

- festes Befehlsformat
  - alle Befehle sind 32 Bit lang
- Opcode-Feld ist 6-bit breit
  - Adressierungsarten werden hier mit codiert
- wenige Befehlsformate
  - R-Format:
    - Register-Register ALU-Operationen
  - I-Format:
    - Lade- und Speicheroperationen
    - alle Operationen mit unmittelbaren Operanden
    - Jump-Register
    - Jump-and-Link-Register

Befehlsformat: Beispiel MIPS

Befehl im R-Format



R–Format

op:	Opcode	Typ des Befehls	0=„alu-op“
rs:	source register 1	erster Operand	„r23“
rt:	source register 2	zweiter Operand	„r30“
rd:	destination register	Zielregister	„r3“
shift:	shift amount	(optionales Shiften)	„0“
funct:	ALU function	Rechenoperation	34=„sub“

$\Rightarrow sub\, r3, r23, r30 \qquad r3 = r23 - r30$

Befehlsformat: Beispiel MIPS

Befehl im I-Format

I–Format

op:	Opcode	Typ des Befehls	35=„lw“
rs:	destination register	Zielregister	„r8“
rt:	base register	Basisadresse	„r5“
addr:	address offset	Offset	6

$\Rightarrow lw\ r8, addr(r5) \qquad r8 = MEM[r5 + addr]$

MIPS

„Microprocessor without interlocked pipeline stages“

- entwickelt an der Univ. Stanford, seit 1982
- Einsatz: eingebettete Systeme, SGI Workstations/Server

- klassische 32-bit RISC Architektur
- 32-bit Wortbreite, 32-bit Speicher, 32-bit Befehle
- 32Register: R0 ist konstant Null, R1 .. R31 Universalregister
- Load-Store Architektur, nur base+offset Adressierung

- sehr einfacher Befehlssatz, 3-Adress-Befehle
- keinerlei HW-Unterstützung für „komplexe“ SW-Konstrukte
- SW muß sogar HW-Konflikte („Hazards“) vermeiden
- Koprozessor-Konzept zur Erweiterung

MIPS: Register

- 32 Register, R0 .. R1, jeweils 32-bit
- R1 bis R31 sind Universalregister
- R0 ist konstant Null (ignoriert Schreiboperationen)
  - erlaubt einige Tricks:

$R5 = -R5$	$subR5, R0, R5$
$R4 = 0$	$addR4, R0, R0$
$R3 = 17$	$addiR3, R0, 17$
$if\,(R2 == 0)$	$bneR2, R0, label$

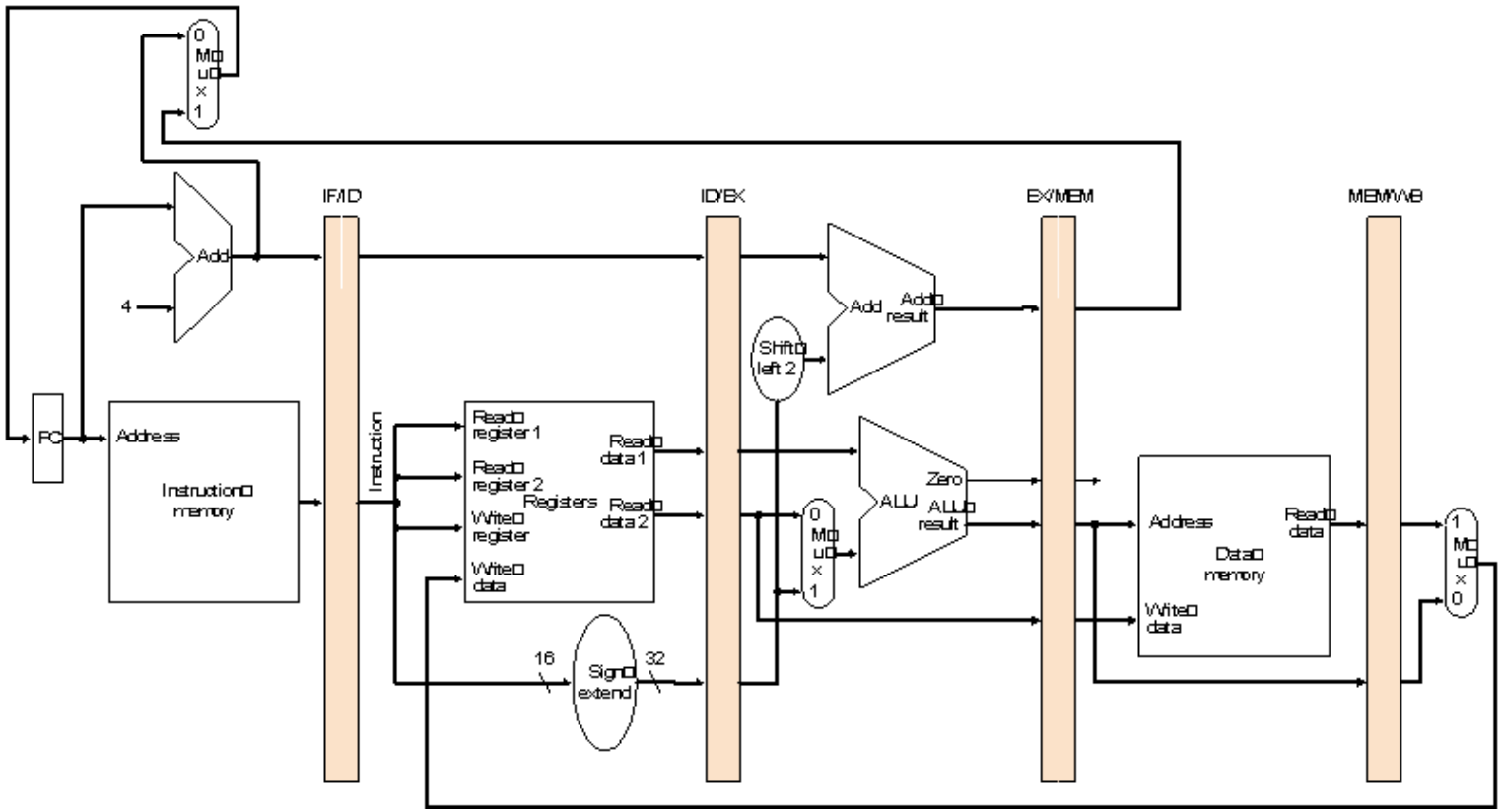
- keine separaten Statusflags
- Vergleichsoperationen setzen Zielregister auf 0 bzw. 1:

$R1 = (R2 < R3)$ $\quad sltR1, R2, R3$

MIPS: Befehlssatz

- Übersicht und Details: siehe Hennessy & Patterson

MIPS: Hardwarestruktur



PC
I-Cache

Register
($0 .. $31)

ALUs

Speicher
D-Cache

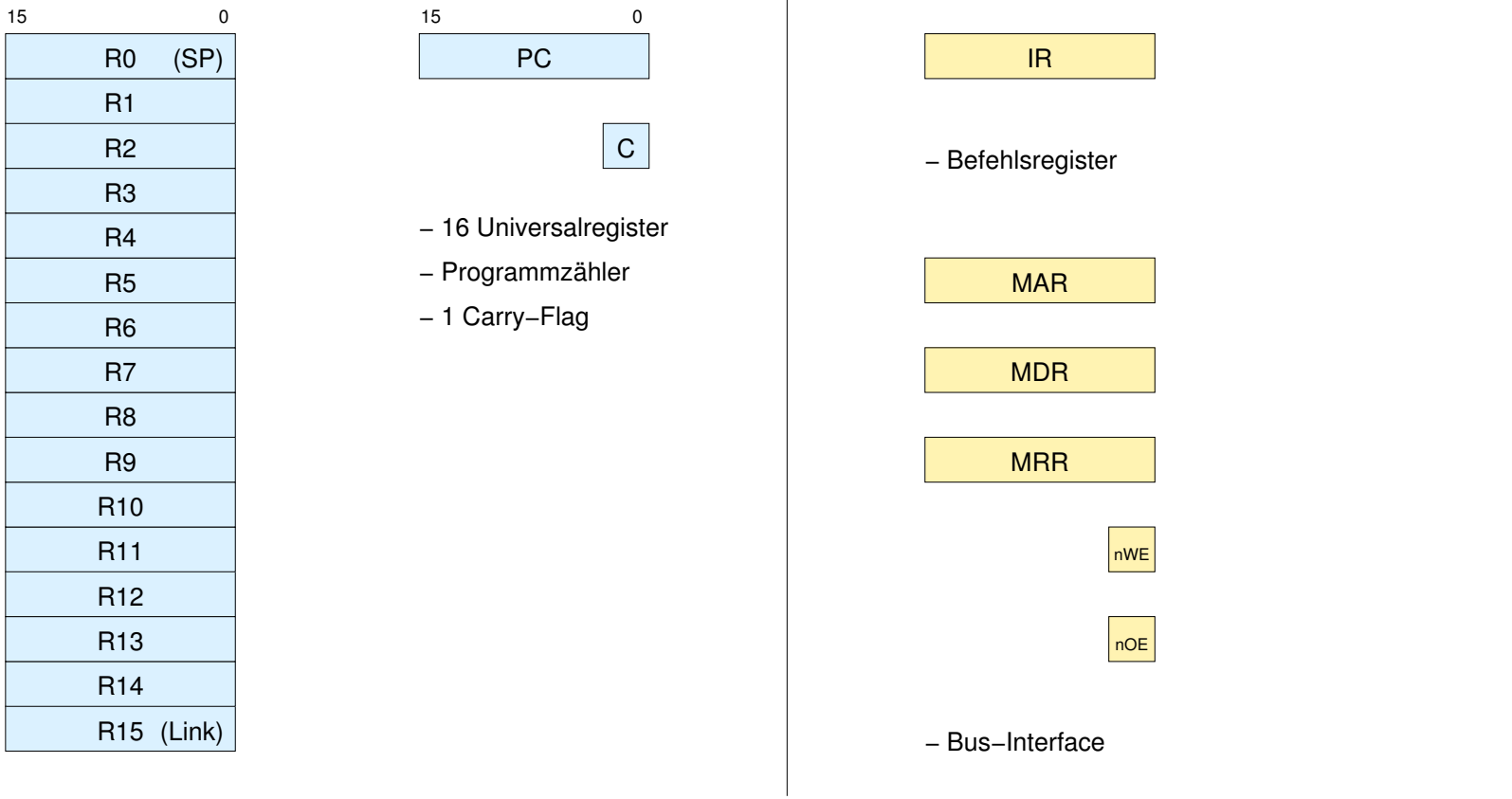
M*CORE

- 32-bit RISC Architektur, Motorola 1998
- besonders einfaches Programmiermodell:
  - Program Counter, PC
  - 16 Universalregister R0 .. R15
  - Statusregister C („carry flag“)
  - 16-bit Befehle (um Programmspeicher zu sparen)
- Verwendung:
  - häufig in Embedded-Systems
  - „smart cards“

D*CORE

- ähnlich M*CORE
- gleiches Registermodell, aber nur 16-bit Wortbreite
  - Program Counter, PC
  - 16 Universalregister R0 .. R15
  - Statusregister C („carry flag“)
- Subset der Befehle, einfachere Kodierung
- vollständiger Hardwareaufbau in Hades verfügbar
- oder Simulator mit Assembler (winT3asm.exe / t3asm.jar)

D*CORE: Register

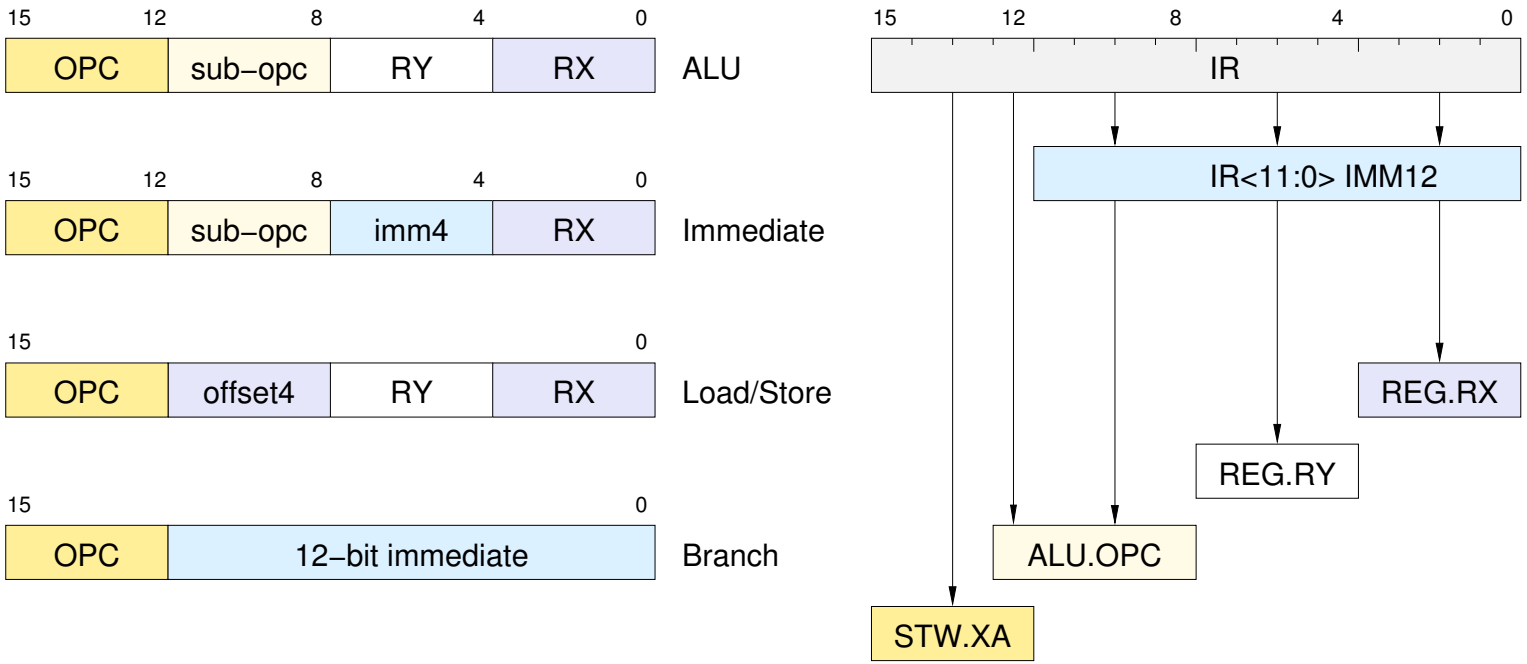


- sichtbar für Programmierer: R0..R15, PC und C (carry flag)

D*CORE: Befehlssatz

mov	move register
addu, addc	Addition (ohne, mit Carry)
subu	Subtraktion
and, or xor	logische Operationen
lsl, lsr, asr	logische, arithmetische Shifts
cmpe, cmpne, ...	Vergleichsoperationen
movi, addi, ...	Operationen mit Immediate-Operanden
ldw, stw	Speicherzugriffe, load/store
br, jmp	unbedingte Sprünge
bt, bf	bedingte Sprünge
jsr	Unterprogrammaufruf
trap	Software interrupt
rfi	return from interrupt

D*CORE: Befehlsformate



- 4-bit Opcode, 4-bit Registeradressen
- einfaches Zerlegen des Befehls in die einzelnen Felder

Adressierungsarten

- woher kommen die Operanden / Daten für die Befehle?
  - Hauptspeicher, Universalregister, Spezialregister
- Wieviele Operanden pro Befehl?
  - 0- / 1- / 2- / 3-Adress-Maschinen
- Wie werden die Operanden adressiert?
  - immediate / direkt / indirekt / indiziert / autoinkrement / usw.

$\Rightarrow$ wichtige Unterscheidungsmerkmale für Rechnerarchitekturen

- Zugriff auf Hauptspeicher ist 100x langsamer als Registerzugriff
  - möglichst Register statt Hauptspeicher verwenden (!)
  - „load/store“-Architekturen

Beispiel: Add-Befehl

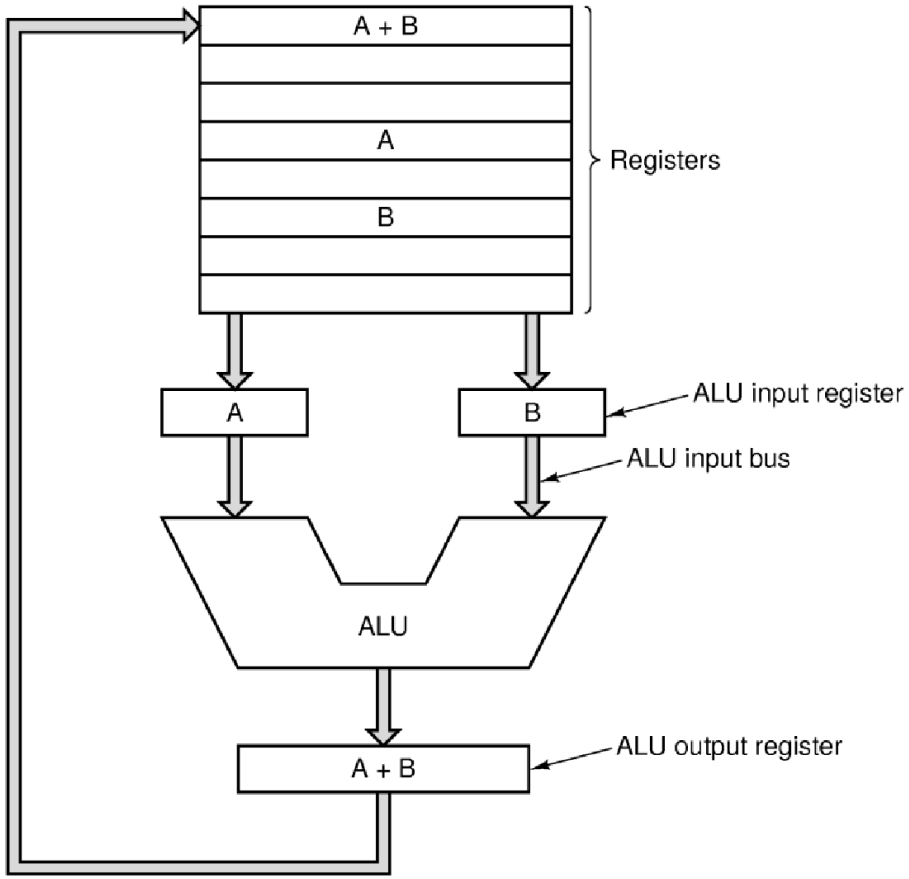
- Rechner soll „rechnen“ können
- typische arithmetische Operation nutzt 3 Variablen
- Resultat, zwei Operanden: $X = Y + Z$

add r2, r4, r5	reg2 = reg4 + reg5 „addiere den Inhalt von R4 und R5 und speichere das Resultat in R2“

- woher kommen die Operanden?
- wo soll das Resultat hin?
  - Speicher
  - Register
- entsprechende Klassifikation der Architektur

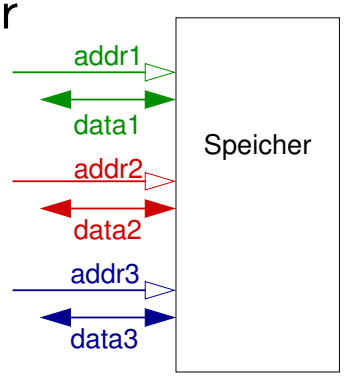
Datenpfad

- Register (-bank)
  - liefern Operanden
  - speichern Resultate
- interne Hilfsregister
- ALU, typ. Funktionen:
  - add, add-carry, sub
  - and, or, xor
  - shift, rotate
  - compare
  - (floating point ops.)



Woher kommen die Operanden?

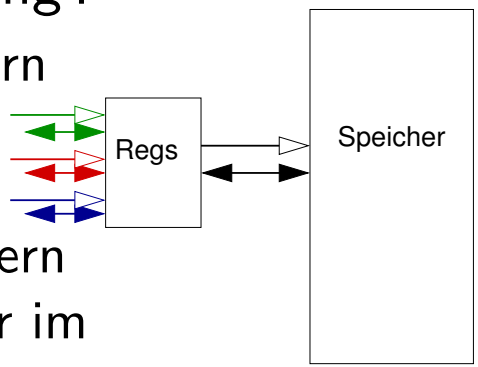
- typische Architektur:
  - von-Neumann Prinzip: alle Daten im Hauptspeicher
  - 3-Adress-Befehle: zwei Operanden, ein Resultat



- „Multiport-Speicher“: mit drei Ports ?
  - sehr aufwendig , extrem teuer, trotzdem langsam
- Register im Prozessor zur Zwischenspeicherung !
  - Datentransfer zwischen Speicher und Registern

    Load reg = MEM[addr]

    Store MEM[addr] = reg
  - RISC: Rechenbefehle arbeiten nur mit Registern
  - CISC: gemischt, Operanden in Registern oder im Speicher



n-Adress-Maschine (n = {3..0})

- 3-Adress-Format
  - X = Y + Z
  - sehr flexibel, leicht zu programmieren
  - Befehl muss 3 Adressen kodieren
- 2-Adress-Format
  - X = X + Z
  - eine Adresse doppelt verwendet (für Resultat und einen Operanden)
  - Format wird häufig verwendet
- 1-Adress-Format
  - ACC = ACC + Z
  - alle Befehle nutzen das Akkumulator-Register
  - häufig in älteren / 8-bit Rechnern
- 0-Adress-Format
  - TOS = TOS + NOS
  - Stapelspeicher (top of stack, next of stack)
  - Adressverwaltung entfällt
  - im Compilerbau beliebt

n-Adress-Maschine

Beispiel: Z = (A-B) / (C + D*E) T = Hilfsregister

3-Adress-Maschine

```
sub Z, A, B
mul T, D, E
add T, T, C
div Z, Z, T
```

2-Adress-Maschine

```
mov  Z, A
sub  Z, B
mov  T, D
mul  T, E
add  T, C
div  Z, T
```

1-Adress-Maschine

```
load  D
mul   E
add   C
stor  Z
load  A
sub   B
div   Z
stor  Z
```

0-Adress-Maschine

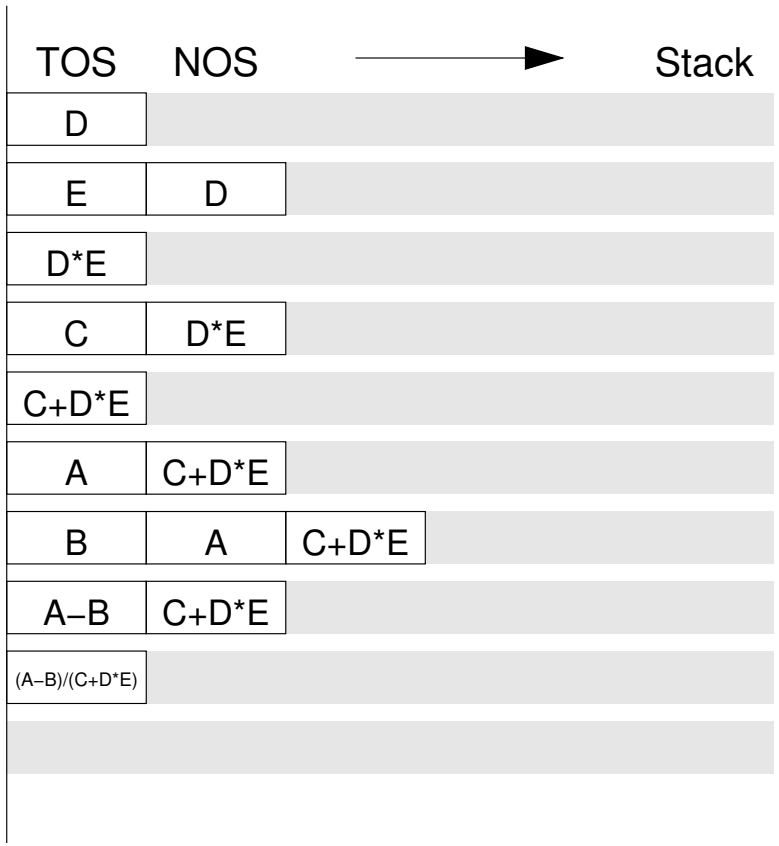
```
push  D
push  E
mul
push  C
add
push  A
push  B
sub
div
pop   Z
```

Stack-Maschine

```
Beispiel:    Z = (A–B) / (C + D*E)          T = Hilfsregister
```

0-Adress-Maschine

```
push   D
push   E
mul
push   C
add
push   A
push   B
sub
div
pop    Z
```



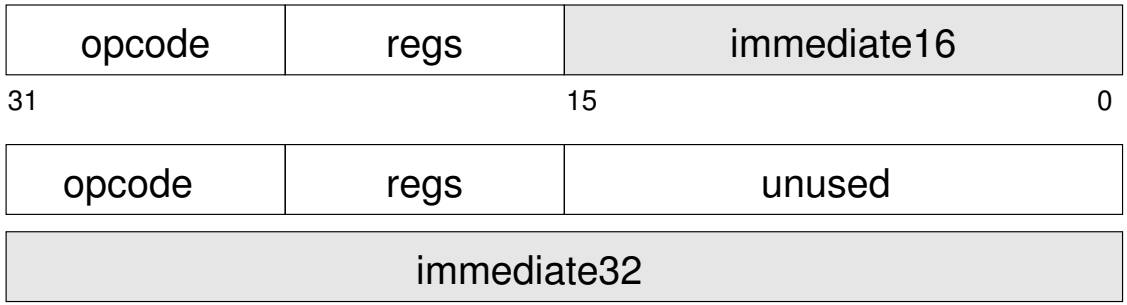
Adressierungsarten

- „immediate“
  - Operand steht direkt im Befehl
  - kein zusätzlicher Speicherzugriff
  - aber Länge des Operanden beschränkt
- „direkt“
  - Adresse des Operanden steht im Befehl
  - keine zusätzliche Adressberechnung
  - ein zusätzlicher Speicherzugriff
  - Adressbereich beschränkt
- „indirekt“
  - Adresse eines Pointers steht im Befehl
  - erster Speicherzugriff liest Wert des Pointers
  - zweiter Speicherzugriff liefert Operanden
  - sehr flexibel (aber langsam)

Adressierungsarten (cont.)

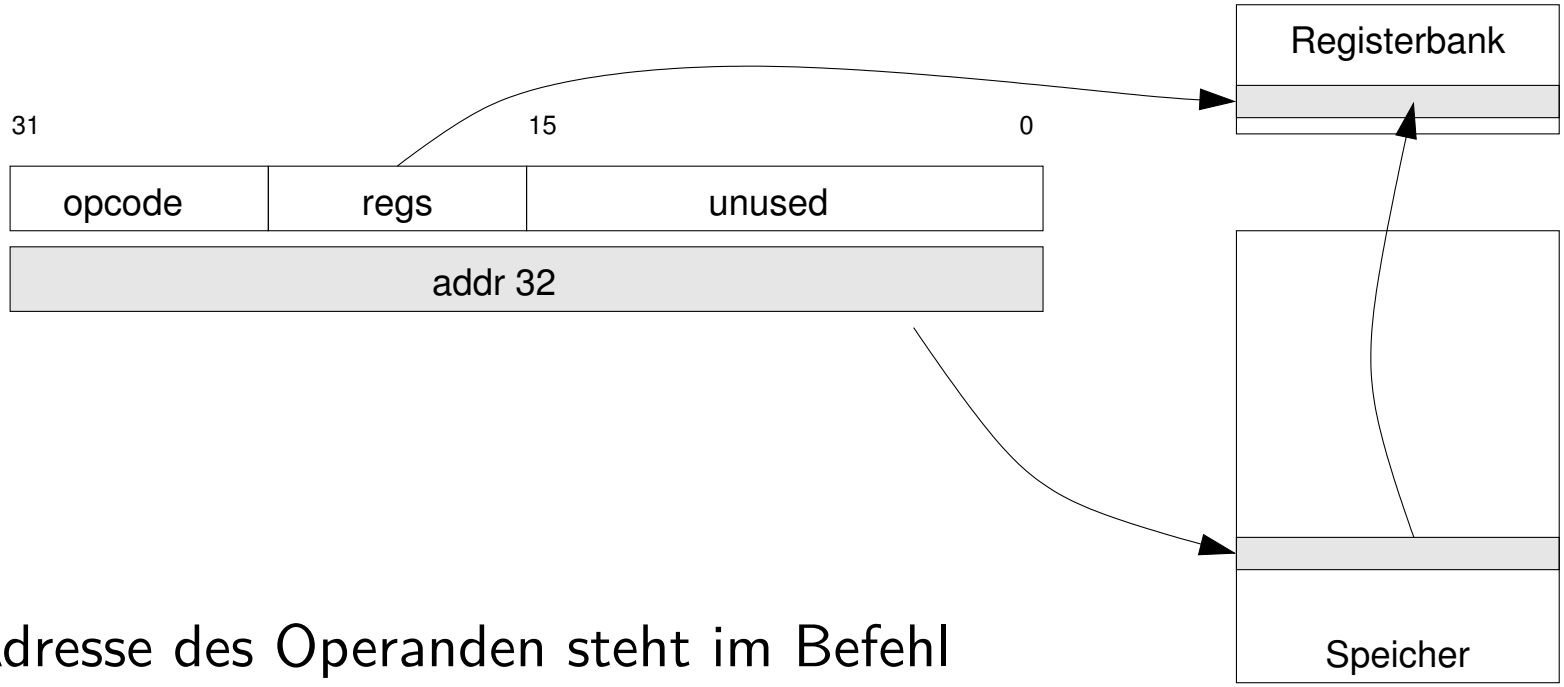
- „register“
  - wie Direktmodus, aber Register statt Speicher
  - 32 Register: benötigen 5 bit im Befehl
  - genug Platz für 2- oder 3-Adress Formate
- „register-indirekt“
  - Befehl spezifiziert ein Register
  - mit der Speicheradresse des Operanden
  - ein zusätzlicher Speicherzugriff
- „indiziert“
  - Angabe mit Register und Offset
  - Inhalt des Registers liefert Basisadresse
  - Speicherzugriff auf (Basisaddresse+offset)
  - ideal für Array- und Objektzugriffe
  - Hauptmodus in RISC-Rechnern (auch: „Versatz-Modus“)

Immediate-Adressierung



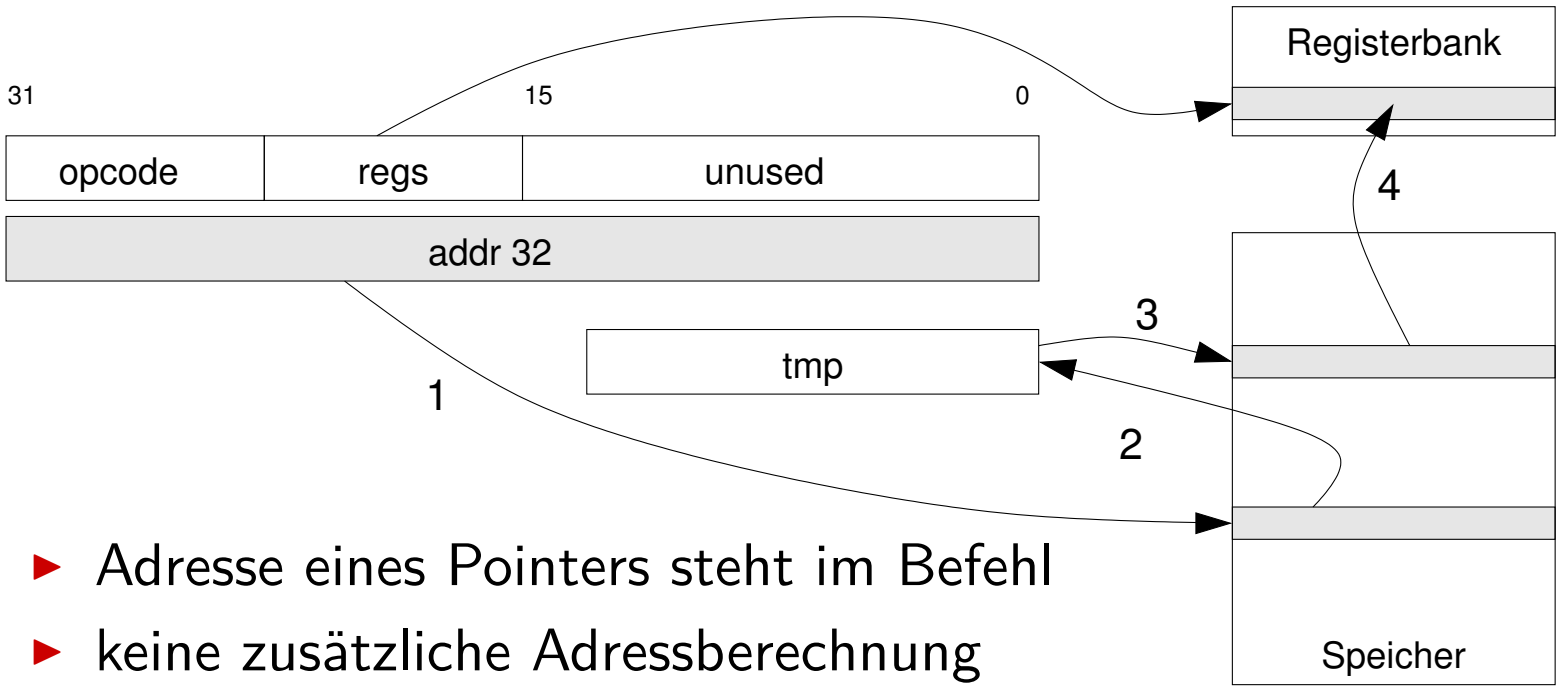
- Operand steht direkt im Befehl, kein zusätzlicher Speicherzugriff
- Länge des Operanden ist kleiner als (Wortbreite - Opcodebreite)
- zur Darstellung grösserer Werte:
  - 2-Wort Befehle (x86)
    (zweites Wort für Immediate-Wert)
  - mehrere Befehle (Mips,SPARC)
    (z.B. obere/untere Hälfte eines Wortes)
  - Immediate-Werte mit zusätzlichem Shift (ARM)

direkte Adressierung



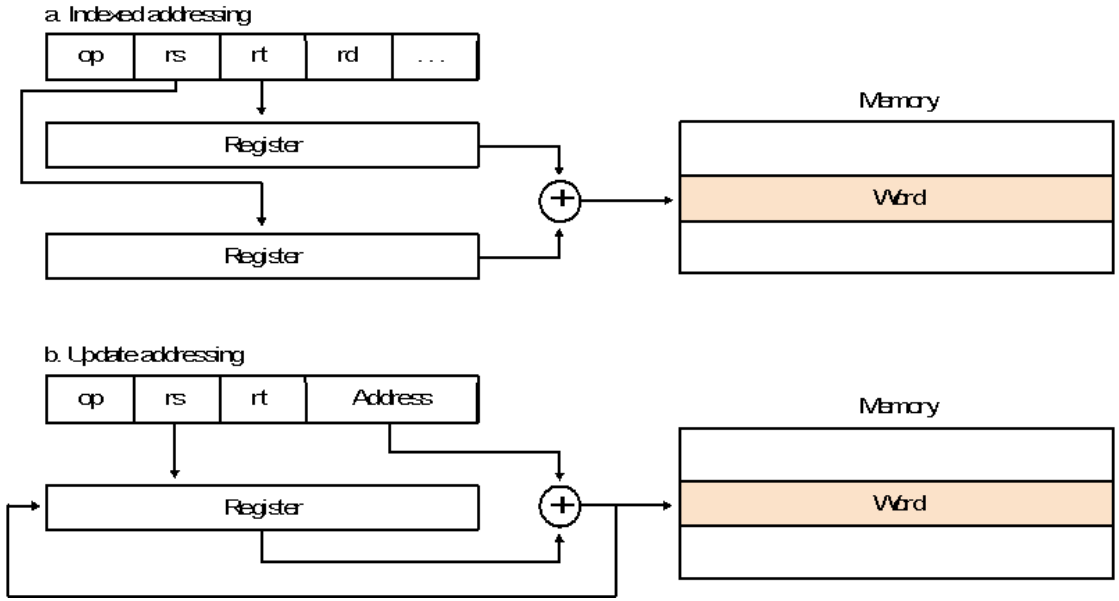
- Adresse des Operanden steht im Befehl
- keine zusätzliche Adressberechnung
- ein zusätzlicher Speicherzugriff: z.B. `R3 = MEM[ addr32 ]`
- Adressbereich beschränkt, oder 2-Wort Befehl (wie Immediate)

indirekte Adressierung



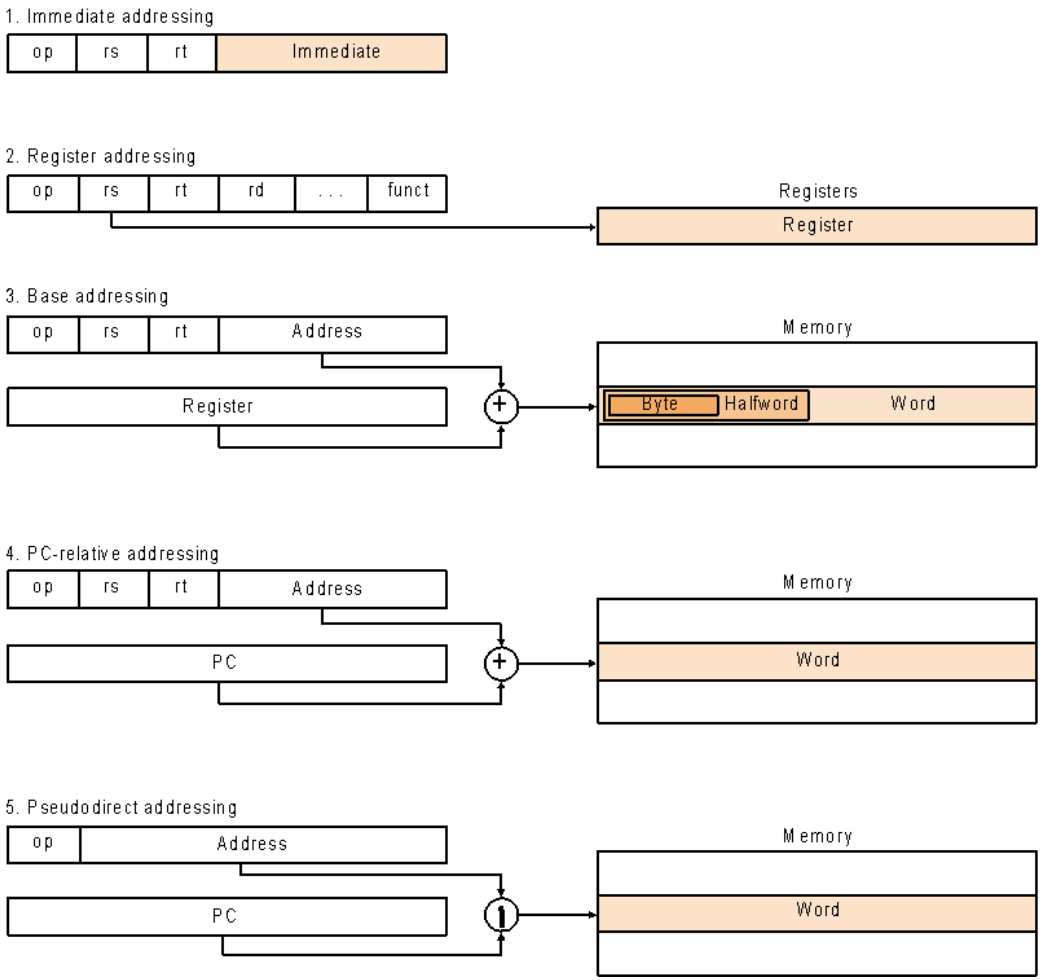
- Adresse eines Pointers steht im Befehl
- keine zusätzliche Adressberechnung
- zwei zusätzliche Speicherzugriffe:
  z.B. `tmp = MEM[ addr32 ]; R3 = MEM[ tmp ]`
- typische CISC-Adressierungsart, viele Taktzyklen
- kommt bei RISC-Rechnern nicht vor

Indizierte Adressierung



- indizierte Adressierung, z.B. für Arrayzugriffe:
  - addr = (Basisregister) + (Sourceregister)
  - addr = (Sourceregister) + offset;
    sourceregister = addr

Weitere Addressierungsarten



(immediate)

(register direct)

(index + offset)

(PC + offset)

(PC | offset)

Adressierung: Varianten

- welche Adressierungsarten / Varianten sind üblich?
  - 0-Adress (Stack-) Maschine: Java virtuelle Maschine
  - 1-Adress (Akkumulator) Maschine. 8-bit Microcontroller, einige x86 Befehle
  - 2-Adress Maschine: einige x86 Befehle, 16-bit Rechner
  - 3-Adress Maschine: 32-bit RISC
- CISC-Rechner unterstützen diverse Adressierungsarten
- RISC meistens nur indiziert mit offset

Gliederung

x86-Architektur[1]

- übliche Bezeichnung für die Intel-Prozessorfamilie
- von 8086, 80286, 80386, 80486, Pentium ... Pentium-IV
- oder „IA-32“: Intel architecture, 32-bit

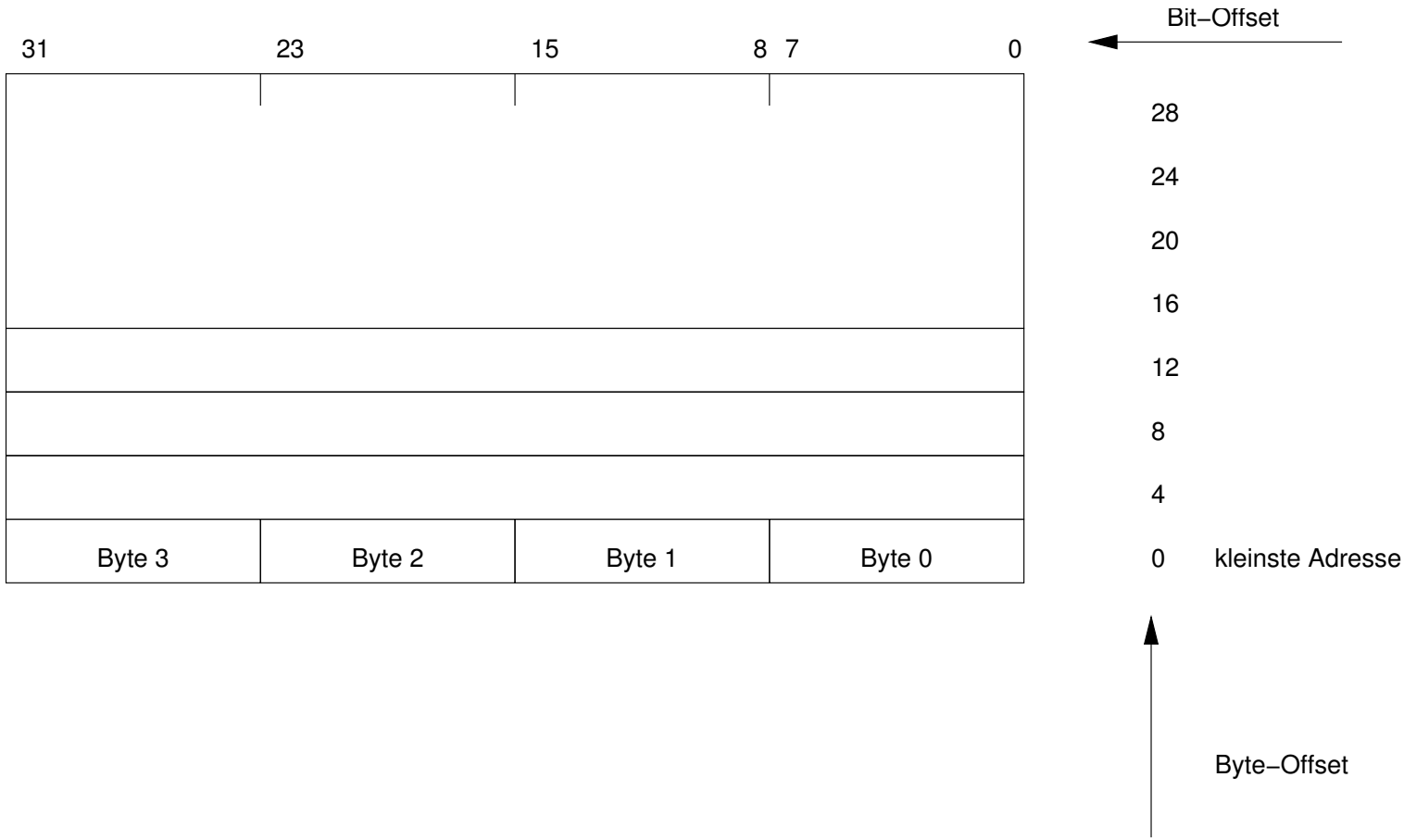
- vollständig irreguläre Struktur: CISC
- historisch gewachsen: diverse Erweiterungen (MMX, SSE, ...)
- ab 386 auch wie reguläre 8-Register Maschine verwendbar

[1] Hinweis:
- die folgenden Folien zeigen eine „vereinfachte“ Version
- niemand erwartet, dass Sie sich die Details merken
- x86-Assemblerprogrammierung ist „grausam“

Beispiel: Evolution des Intel x86

Chip	Datum	MHz	Transistoren	Speicher	
4004	4/1971	0,108	2.300	640	erster Mikropro
8008	4/1972	0.108	3.500	16 KB	erster 8-bit Mik
8080	4/1974	2	6.000	64 KB	Erste „general-p
8086	6/1978	5–10	29.000	1 MB	erste 16-bit CP
8088	6/1979	5–8	29.000	1 MB	Einsatz im IBM
80286	2/1982	8–12	134.000	16 MB	„Protectec-Mod
80386	10/1985	16–33	275.000	4 GB	erste 32-Bit CP
80486	4/1989	25-100	1.2M	4 GB	integrierter 8K
Pentium	3/1993	60–233	3.1M	4 GB	zwei Pipelines,
Pentium Pro	3/1995	150–200	5.5M	4 GB	integrierter first
Pentium II	5/1997	233–400	7.5M	4 GB	Pentimum Pro
Pentium III	2/1999	450–1400	9.5–44M	4 GB	SSE-Einheit
Pentium IV	11/2000	1.300–3.600	42–188M	4 GB	hohe Taktfrequ
CORE-2	5/2007	1.600–3.200	143–410M		64-bit Architekt

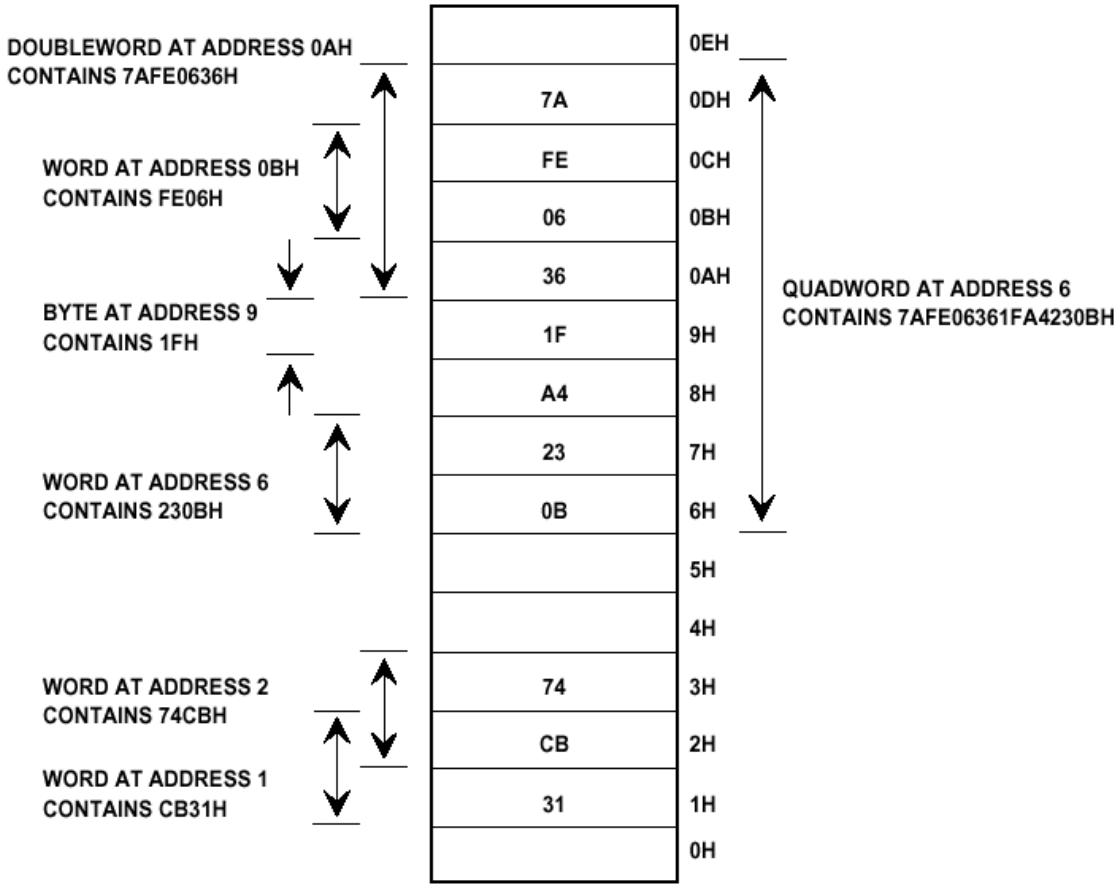
x86: Speichermodell



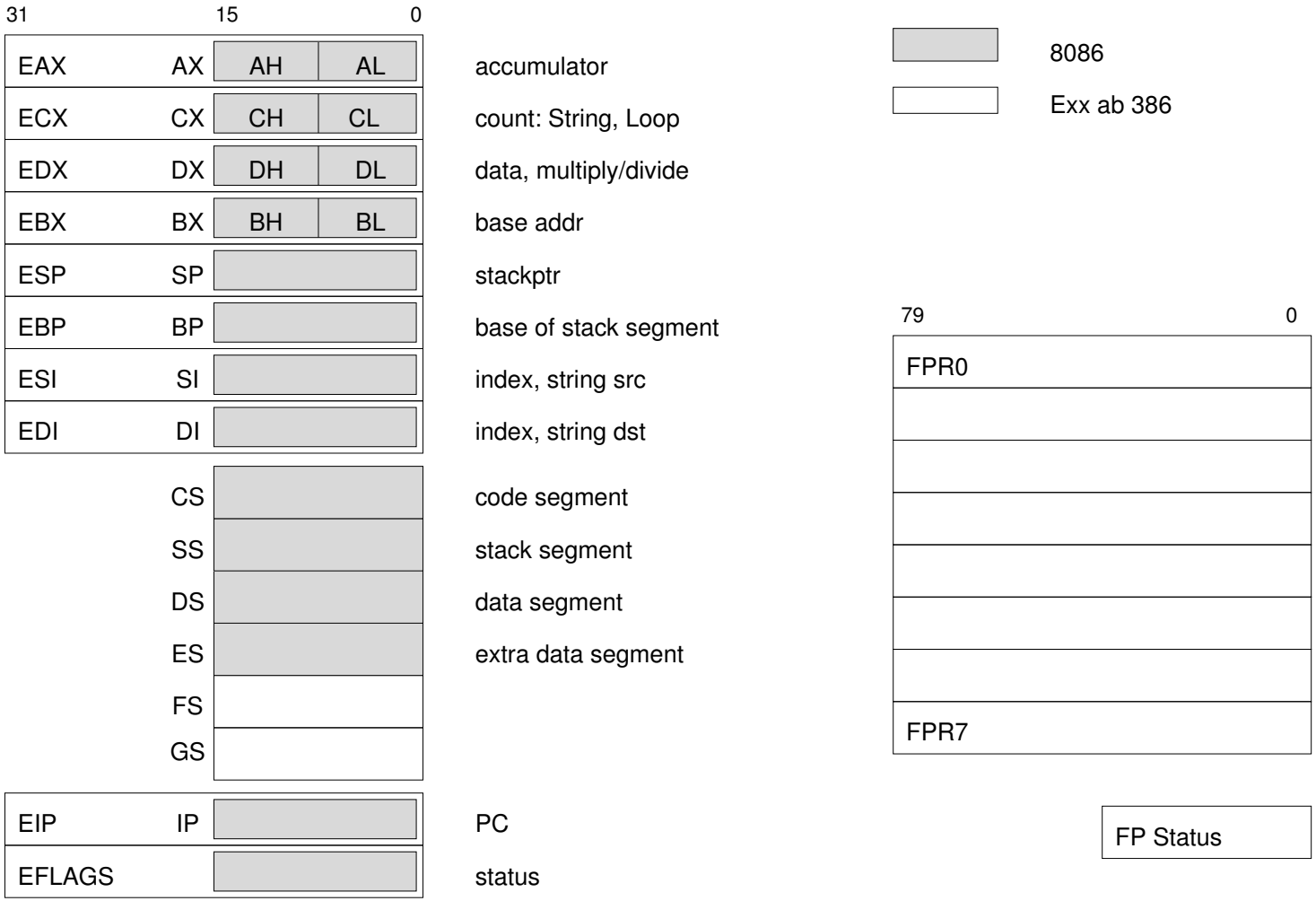
- „little endian“: LSB eines Wortes bei der kleinsten Adresse

x86: Speichermodell

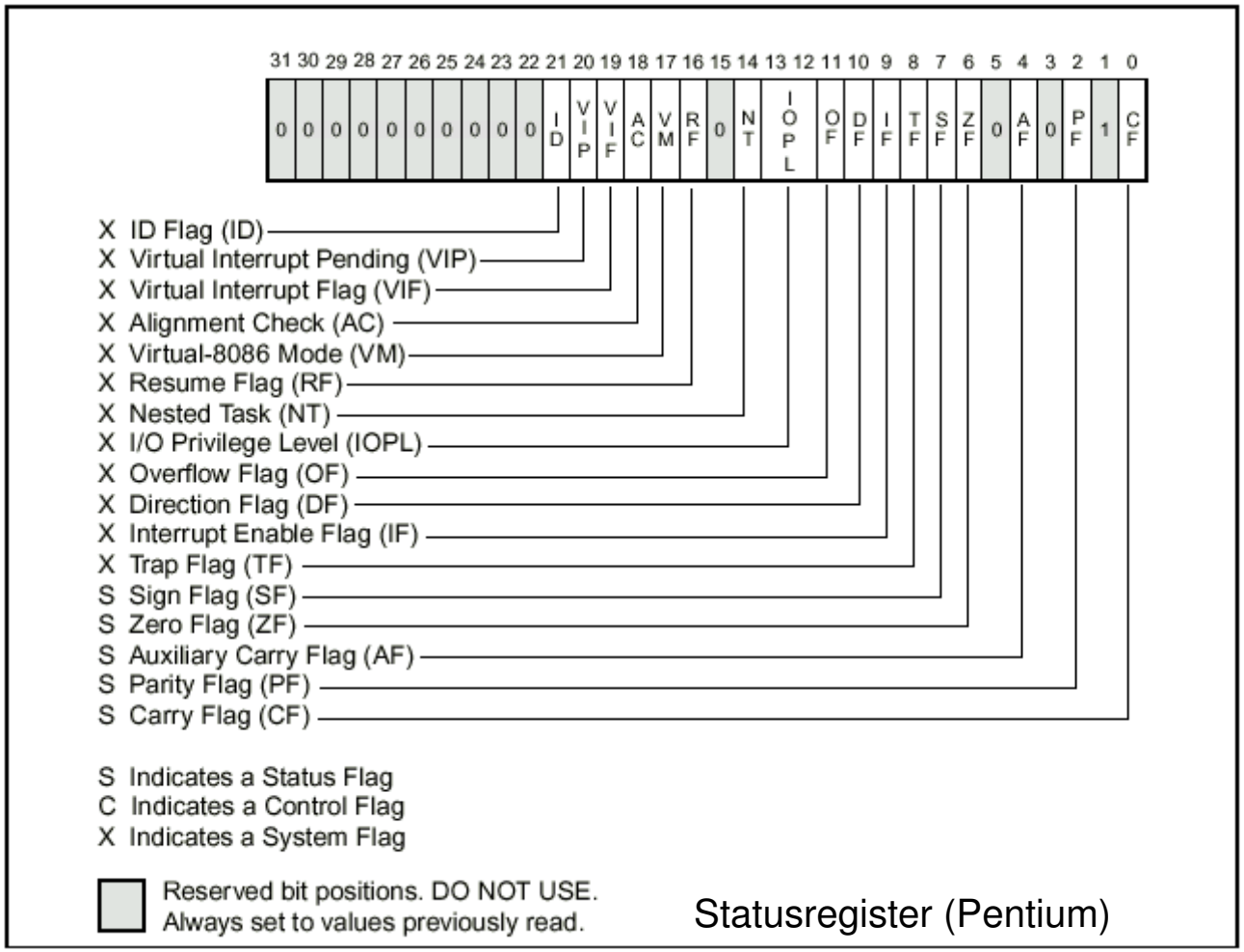
- Speicher voll byte-adressierbar
- mis-aligned Zugriffe langsam
- Beispiel zeigt:
  - Byte
  - Word,
  - Doubleword,
  - Quadword,



x86: Register



x86: EFLAGS Register



Statusregister (Pentium)

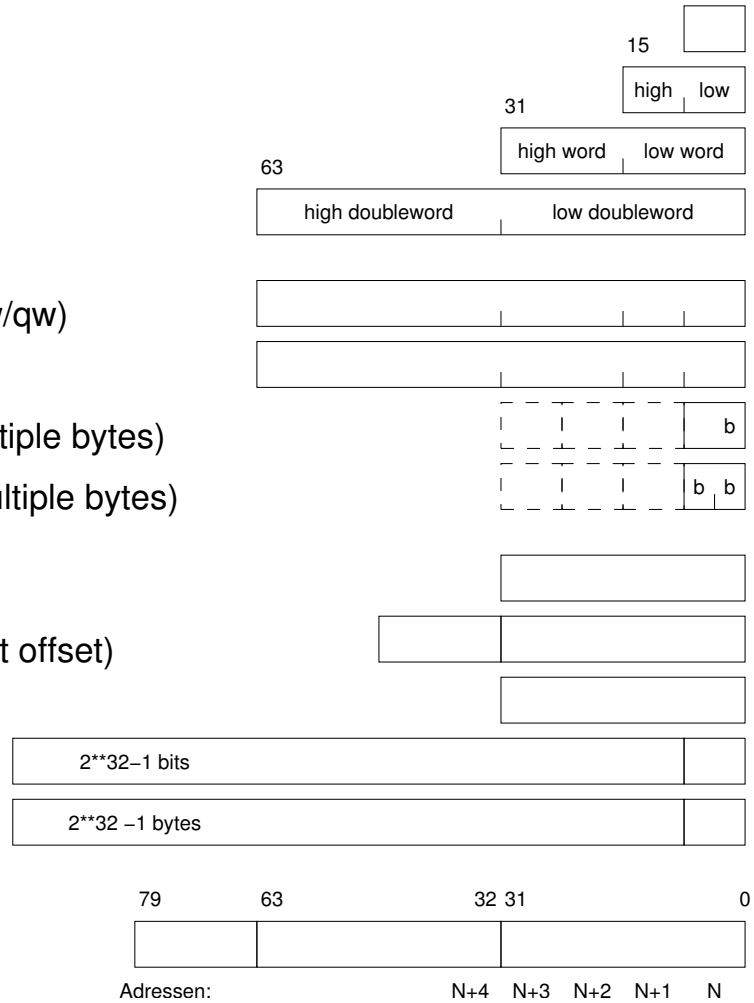
Datentypen: CISC...

bytes
word
doubleword
quadword

integer (2–complement b/w/dw/qw)
ordinal (unsigned b/w/dw/qw)
BCD (one digit per byte, multiple bytes)
packed BCD (two digits per byte, multiple bytes)

near pointer (32 bit offset)
far pointer (16 bit segment + 32 bit offset)
bit field
bit string
byte string

float / double / extended



x86: Befehlssatz

▶ Datenzugriff	mov, xchg
▶ Stack-Befehle	push, pusha, pop, popa
▶ Typumwandlung	cwd, cdq, cbw (byte->word), movsx, . . .
▶ Binärarithmetik	add, adc, inc, sub, sbb, dec, cmp, neg, . . .
	mul, imul, div, idiv,
▶ Dezimalarithmetik	packed / unpacked BCD: daa, das, aaa, aas, .
▶ Logikoperationen	and, or, xor, not, sal, shr, shr, . . .
▶ Sprungbefehle	jmp, call, ret, int, iret, loop, loopne, . . .
▶ String-Operationen	ovs, cmps, scas, load, stos, . . .
▶ „high-level“	enter (create stack frame), . . .
▶ diverses	lahf (load AH from flags), . . .
▶ Segment-Register	far call, far ret, lds (load data pointer)
⇒ CISC	zusätzlich diverse Ausnahmen/Spezialfälle

x86: Befehlsformate: CISC . . .

- außergewöhnlich komplexes Befehlsformat:
  1. prefix (repeat / segment override / etc.)
  2. opcode (eigentlicher Befehl)
  3. register specifier (Ziel / Quellregister)
  4. address mode specifier (diverse Varianten)
  5. scale-index-base (Speicheradressierung)
  6. displacement (Offset)
  7. immediate operand
- ausser dem Opcode alle Bestandteile optional
- unterschiedliche Länge der Befehle, von 1 .. 37 Byte

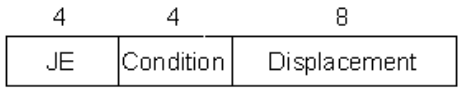
$\Rightarrow$ extrem aufwendige Dekodierung

x86: Modifier

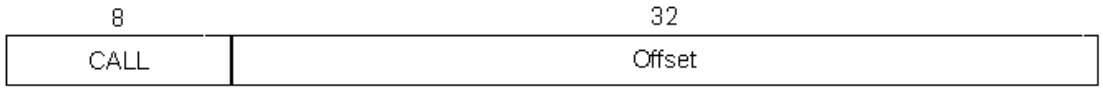
- alle Befehle können mit „Modifiern“ ergänzt werden:
  - segment override — Addr. aus angewähltem Segmentregister
  - address size — Umschaltung 16/32-bit
  - operand size — Umschaltung 16/32-bit
  - repeat — für Stringoperationen
Operation auf allen Elementen ausführen
  - lock — Speicherschutz für Multiprozessoren

Beispiel: x86 Befehlskodierung

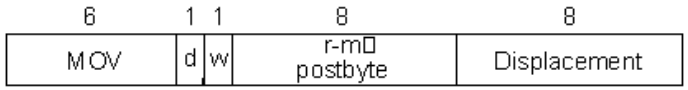
a. JE EIP + displacement



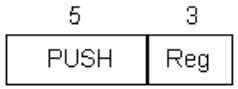
b. CALL



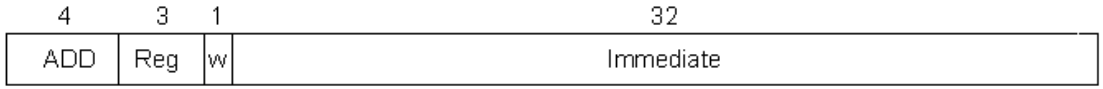
c. MOV EBX, [EDI + 45]



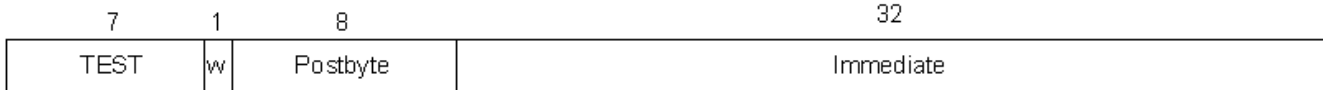
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



- 1 Byte .. 36 Bytes
- vollkommen irregulär

Beispiel: x86 Befehle

Instruction	Function
JE name	If equal (CC) EIP = name};□ EIP – 128 ≤ name < EIP + 128
JMP name	{EIP = NAME};
CALL name	SP = SP – 4; M[SP] = EIP + 5; EIP = name;
MOVW EBX,[EDI + 45]	EBX = M [EDI + 45]
PUSH ESI	SP = SP – 4; M[SP] = ESI
POP EDI	EDI = M[SP]; SP = SP + 4
ADD EAX,#6765	EAX = EAX + 6765
TEST EDX,#42	Set condition codea (flags) with EDX & 42
MOVSL	M[EDI] = M[ESI];□ EDI = EDI + 4; ESI = ESI + 4

x86: Assembler-Beispiel

```
addr opcode    assembler                c quellcode
-------------------------------------------------------------------------------
               .file     "hello.c"
               .text
0000 48656C6C  .string  "Hello x86!\n"
     6F207838
     36210A00
               .text
               print:
0000 55           pushl %ebp             | void print( char* s ) {
0001 89E5         movl %esp,%ebp
0003 53           pushl %ebx
0004 8B5D08       movl 8(%ebp),%ebx
0007 803B00       cmpb $0,(%ebx)         |   while( *s != 0 ) {
000a 7418         je .L18
                 .align 4
               .L19:
000c A100000000   movl stdout,%eax       |     putc( *s, stdout );
0011 50           pushl %eax
0012 0FBE03       movsbl (%ebx),%eax
0015 50           pushl %eax
0016 E8FCFFFF     call _IO_putc
     FF
001b 43           incl %ebx              |     s++;
001c 83C408       addl $8,%esp           |   }
001f 803B00       cmpb $0,(%ebx)
0022 75E8         jne .L19
               .L18:
0024 8B5DFC       movl -4(%ebp),%ebx     | }
0027 89EC         movl %ebp,%esp
0029 5D           popl %ebp
002a C3           ret
```

x86: Assembler-Beispiel (2)

```
addr opcode    assembler                c quellcode
-------------------------------------------------------------------------------
               .Lfe1:
               .Lscope0:
002b 908D7426  .align 16
     00
               main:
0030 55           pushl %ebp            | int main( int argc, char** argv )
0031 89E5         movl %esp,%ebp
0033 53           pushl %ebx

0034 BB00000000   movl $.LC0,%ebx       |   print( "Hello x86!\n" );
0039 803D0000     cmpb $0,.LC0
     000000
0040 741A         je .L26
0042 89F6         .align 4
               .L24:
0044 A100000000   movl stdout,%eax
0049 50           pushl %eax
004a 0FBE03       movsbl (%ebx),%eax
004d 50           pushl %eax
004e E8FCFFFFFF   call _IO_putc
0053 43           incl %ebx
0054 83C408       addl $8,%esp
0057 803B00       cmpb $0,(%ebx)
005a 75E8         jne .L24
               .L26:
005c 31C0         xorl %eax,%eax        |   return 0;
005e 8B5DFC       movl -4(%ebp),%ebx    | }
0061 89EC         movl %ebp,%esp
0063 5D           popl %ebp
0064 C3           ret
```

Ergänzende Literatur

Zur Rechnerarchitektur (2. Termin):

[1] Randal E. Bryant and David O'Hallaron.
Computer Systems.
Pearson Education, Inc., New Jersey, 2003.

[2] David A. Patterson and John L. Hennessy.
Computer Organization and Design. The Hardware / Software Interface.
Morgan Kaufmann Publishers, Inc., San Francisco, 1998.

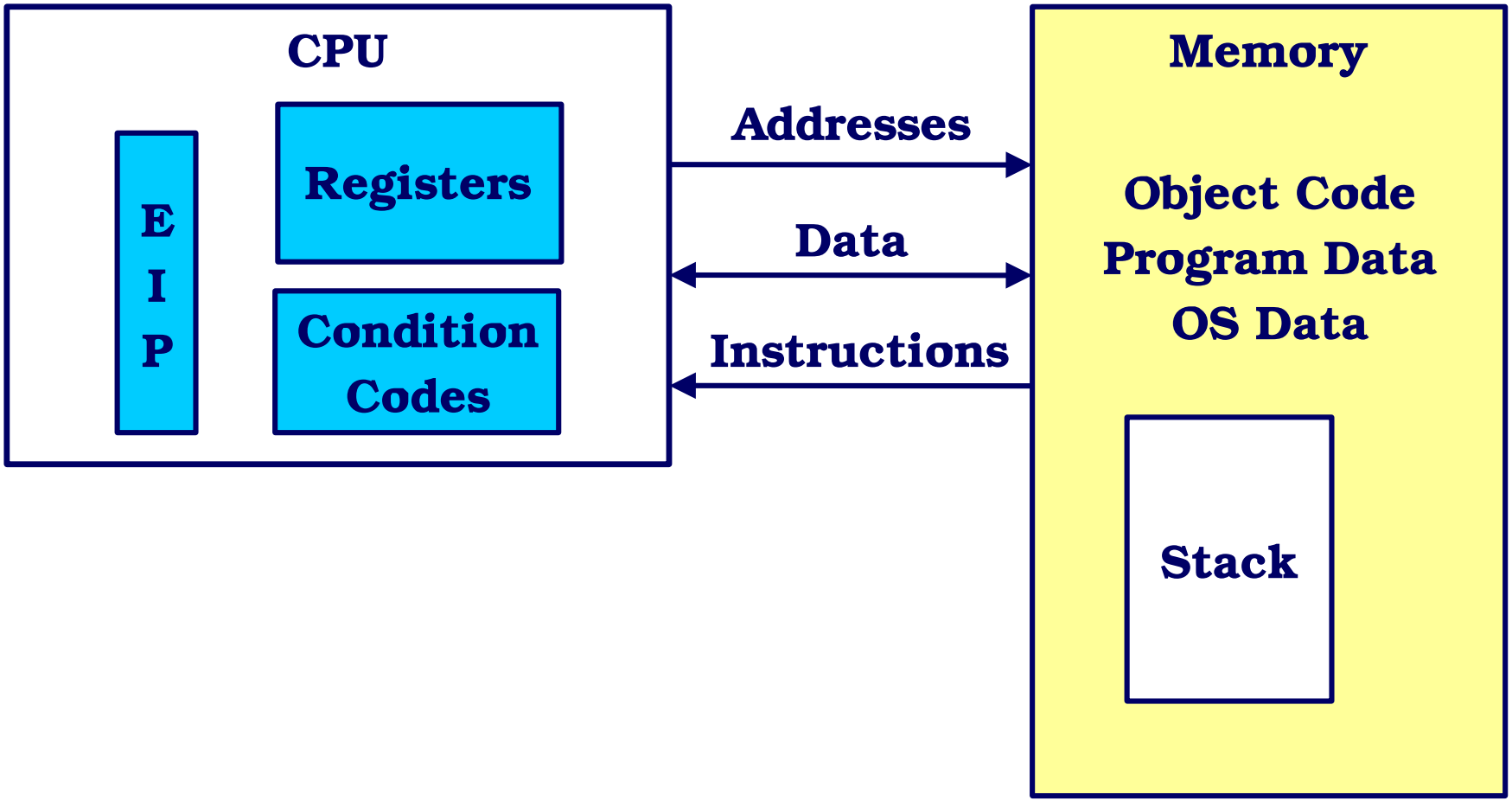
[3] Andrew S. Tanenbaum and James Goodman.
Computerarchitektur.
Pearson Studium München, 2001.

Gliederung

Gliederung (cont.)

Assembler-Programmierung

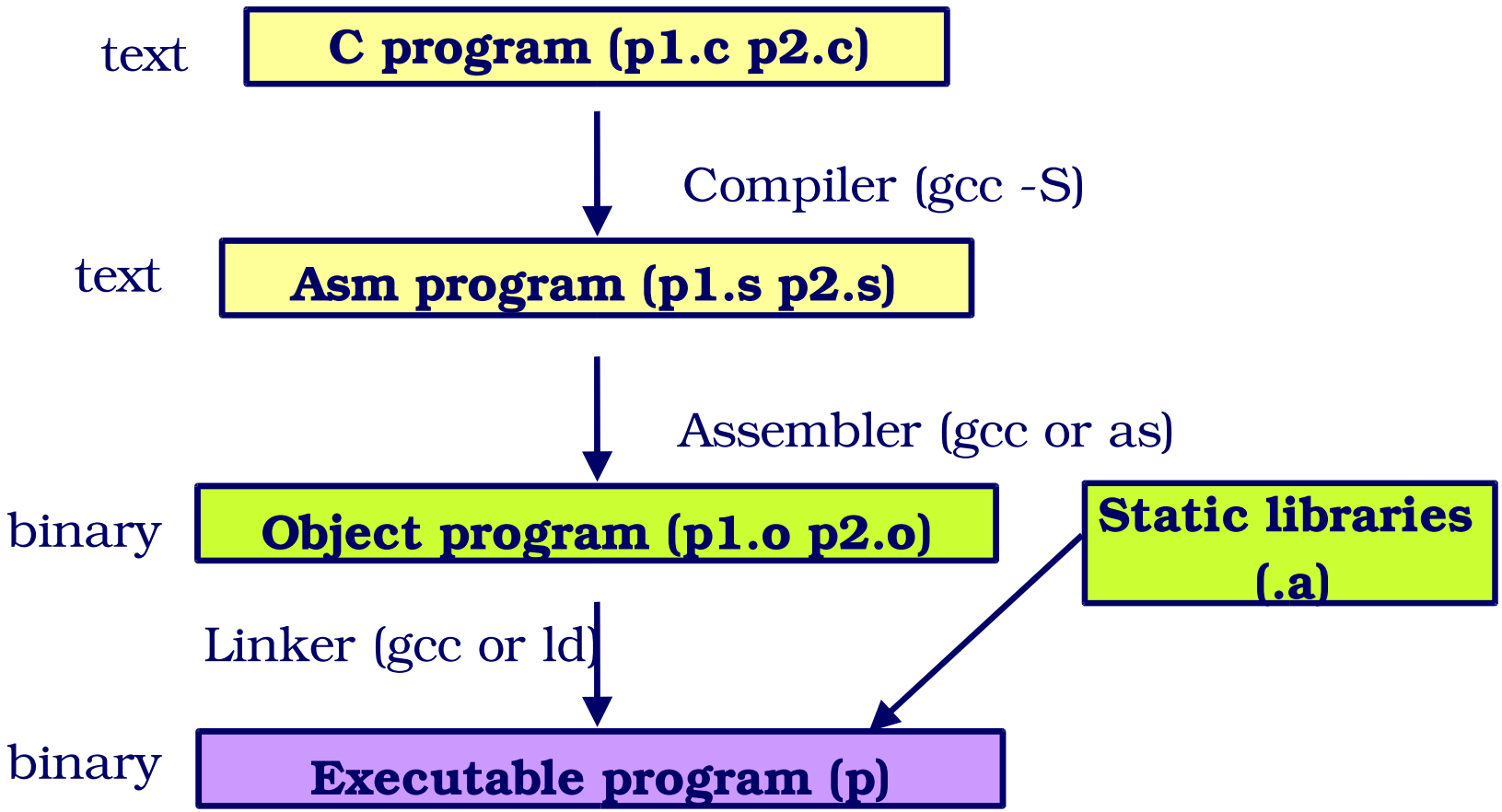
Assembler aus der Sicht des Programmierers



Beobachtbare Zustände (Assemblersicht)

- EIP — Programmzähler
  - Adresse der nächsten Anweisung
- Register-Bank
  - häufig benutzte Programmdaten
- Zustandscodes
  - gespeicherte Statusinformationen über die letzte arithmetische Operation
  - werden für Conditional Branching benötigt
- Speicher
  - durch Bytes addressierbarer Array
  - Code, Nutzerdaten, (einige) OS Daten
  - beinhaltet Kellerspeicher zur Unterstützung von Abläufen

Umwandlung von C in Objektcode



Kompilieren in Assembler

C Code

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Generated Assembly

```
_sum:
        pushl %ebp
        movl %esp,%ebp
        movl 12(%ebp),%e
        addl 8(%ebp),%ea
        movl %ebp,%esp
        popl %ebp
        ret
```

Obtain with command

```
gcc -O -S code.c
```

Produces file `code.s`

Assembler Charakteristika

Minimale Datentypen

- Ganzzahl- Daten mit 1,2 oder 4 Bytes
  - Datenwerte
  - Adressen
- Gleitkomma-Daten mit 4,8, oder 10 Bytes
- keine Aggregattypen wie Arrays oder Strukturen
  - nur fortlaufend zugeteilte Bytes im Speicher

Assembler Charakteristika

Primitive Operationen

- arithmetische Funktionen auf Registern und Speicher
- Transfer von Daten zwischen Speicher und Registern
  - laden Daten aus dem Speicher ins Register
  - legen Registerdaten im Speicher ab
- transferieren Kontrolle
  - „Unconditional Jumps“ zu/von Prozeduren
  - „Conditional Branches“

Objektcode

Code for `sum`

```
0x401040 <sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

- Total of 13 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address `0x401040`

Assembler und Binder

Assembler

- übersetzt .s zu .o
- binäre Kodierung jeder Anweisung
- fast vollständiges Bild des ausführbaren Codes
- fehlende Verknüpfungen zwischen Codes in verschiedenen Dateien

Binder (Linker)

- löst Referenzen zwischen Dateien auf
- kombiniert mit statischen Laufzeit-Bibliotheken
  - Z.B. Code für malloc, printf
- manche Bibliotheken sind *dynamisch* verknüpft
  - Verknüpfung wird zur Laufzeit erstellt

Beispiel eines Maschinenbefehls

- C-Code: `int t = x+y;`
  - addiert zwei Ganzzahlen mit Vorzeichen
- Assembler: `addl 8(%ebp), %eax`
  - fügt 2 4-Byte-Ganzzahlen hinzu
    - „Lange“ Wörter in GCC Sprache
    - gleicher Befehl für mit oder ohne Vorzeichen
  - Operanden:

    x: Register `%eax`
    y: Speicher `M[%ebp+8]`
    t: Register `%eax`
    Resultatwert in `%eax`
- Objektcode: `0x401046: 03 45 08`
  - 3-Byte Befehl, Gespeichert unter der Adresse `0x401046`

```
int t = x+y;
```

```
    addl 8(%ebp),%eax
```

Similar to expression
`x += y`

```
0x401046:      03 45 08
```

Objektcode Disassembler

```
00401040 <_sum>
 0:   55                    push   %ebp
 1:   89 e5                 mov    %esp,%ebp
 3:   8b 45 0c              mov    0xc(%ebp),%eax
 6:   03 45 08              add    0x8(%ebp),%eax
 9:   89 ec                 mov    %ebp,%esp
 b:   5d                    pop    %ebp
 c:   c3                    ret
 d:   8d 76 00              lea    0x0(%esi),%esi
```

objdump -d p

- Werkzeug zur Untersuchung des Objektcodes
- rekonstuiert aus Binärcode den Assemblercode
- kann entweder auf einer a.out (vollständiges, ausführbares Programm) oder einer .o Datei ausgeführt werden

Alternativer Disassembler (gdb)

gdb p
disassemble sum
Disassemble procedure
x/13b sum
Examine the 13 bytes starting at sum

```
0x401040  <sum>:      push  %ebp
0x401041  <sum+1>:    mov   %esp,%ebp
0x401043  <sum+3>:    mov   0xc(%ebp),%eax
0x401046  <sum+6>:    add   0x8(%ebp),%eax
0x401049  <sum+9>:    mov   %ebp,%esp
0x40104b  <sum+11>:   pop   %ebp
0x40104c  <sum+12>:   ret
0x40104d  <sum+13>:   lea   0x0(%esi),%esi
```

Was kann „disassembliert“ werden?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:     file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:
 30001000 <.text>:
 30001000:        55                   push  %ebp
 30001001:        8b ec                mov   %esp,%ebp
 30001003:        6a ff                push  0xffffffff
 30001005:        68 90 10 00 30       push  0x30001090
 3000100a:        68 91 dc 4c 30       push  0x304cdc91
```

- alles, was als ausführbarer Code interpretiert werden kann
- Disassembler untersucht Bytes und rekonstruiert Assemblerquelle

Datentransfer („move“)

- Fromat: `movl` *`Source, Dest`*
- transferiert ein 4-Byte („long“) Wort
- sehr häufige Instruktion
- Typ der Operanden
  - Immediate: Konstante, ganzzahlig
    - wie C-Konstante, aber mit dem Präfix ‘$’
    - z.B., $0x400, $-533
    - kodiert mit 1,2 oder 4 Bytes
  - Register: eins von 8 ganzzahl Registern
    - aber %esp und und %ebp für spezielle Aufgaben reserviert
    - bei anderen spezielle Verwendungen in bestimmten Anweisungen
  - Speicher: 4 konsekutive Speicherbytes
    - Zahlreiche „Adressmodi“

movl Operanden-Kombinationen

	Source	Destination		C Analogon
movl	*Imm*	*Reg*	`movl $0x4, %eax`	`temp = 0x4;`
		Mem	`movl $-147, (%eax)`	`*p = 147`
	Reg	*Reg*	`movl %eax, %edx`	`temp2 = temp1;`
		Mem	`movl %eax, (%edx)`	`*P = temp;`
	Mem	*Reg*	`movl (%eax), %edx`	`temp = *p;`

Einfache Addressierungsmodi (Speicherreferenzen)

- Normal: `(R)` → `Mem[Reg[R]]`
  - Register `R` spezifiziert die Speicheradresse
  - Beispiel: `movl (%ecx), %eax`
- Displacement: `D(R)` → `Mem[Reg[R]+D]`
  - Register R
  - Konstantes „Displacement“ D spezifiziert den „offset“
  - Beispiel: `movl 8 (%ebp), %edx`

Benutzen einfacher Adressierungsmodi

```
void swap(int *xp, int *yp)
{
        int t0 = *xp;
        int t1 = *yp;
        *xp = t1;
        *yp = t0;
}
```

```
swap:
        pushl   %ebp
        movl    %esp,       %ebp
        pushl   %ebx

        movl    12(%ebp), %ecx
        movl    8(%ebp),    %edx
        movl    (%ecx),     %eax
        movl    (%edx),     %ebx
        movl    %eax,       (%edx)
        movl    %ebx,       (%ecx)

        movl    -4(%ebp), %ebx
        movl    %ebp,       %esp
        popl    %ebp
        ret
```

Indizierte Adressierungmodi

- gebräuchlichste Form:
  - `Imm(Rb, Ri, S)` → `Mem[Reg[Rb]+S*Reg[Ri]+Imm]`
    - *Imm* : Offset
    - *Rb* : Basisregister: eins der 8 ganzzahligen Registern
    - *Ri* : Indexregister: jedes außer %esp
      (%ebp grundsätzlich möglich, jedoch unwarscheinlich)
    - *S* : „Scaling“ Faktor, 1,2,4 oder 8
- Spezielle Fälle:
  - `(Rb,Ri)` → `Mem[Reg[Rb] + Reg[Ri]]`
  - `Imm(Rb,Ri)` → `Mem[Reg[Rb] + Reg[Ri] + Imm]`
  - `(Rb,Ri,S)` → `Mem[Reg[Rb] + S*Reg[Ri]]`

Beispiel für Adressenberechnung

%edx	0xf000
%ecx	0x100

Expression	Computation	Address
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx,%ecx)	0xf000 + 0x100	0xf100
(%edx,%ecx,4)	0xf000 + 4*0x100	0xf400
0x80(,%edx,2)	2*0xf000 + 0x80	0x1e080

Arithmetische Operationen

Instruktionen mit binären Operatoren

Format		Operation					
addl	Src,Dest	Dest	=	Dest	+	Src	
subl	Src,Dest	Dest	=	Dest	-	Src	
imull	Src,Dest	Dest	=	Dest	*	Src	
sall	Src,Dest	Dest	=	Dest	«	Src	Also called shll
sarl	Src,Dest	Dest	=	Dest	»	Src	Arithmetic
shrl	Src,Dest	Dest	=	Dest	»	Src	Logical
xorl	Src,Dest	Dest	=	Dest	$\hat{}$	Src	also: $\oplus$
andl	Src,Dest	Dest	=	Dest	&	Src	
orl	Src,Dest	Dest	=	Dest	|	Src	

Arithmetische Operationen

Instruktionen mit unären Operatoren

Format		Operation
incl	Dest	Dest = Dest + 1
decl	Dest	Dest = Dest - 1
negl	Dest	Dest = – Dest
notl	Dest	Dest = ∼ Dest

Ein Beispiel

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
movl 8(%ebp),%eax        eax = x
xorl 12(%ebp),%eax       eax = x^y          (t1)
sarl $17,%eax            eax = t1>>17       (t2)
andl $8185,%eax          eax = t2 & 8185
```

$(2^{13} = 8192,\ 2^{13} - 7 = 8185]$

Kontrollfluss

Themen

- Zustandscodes
  - Setzen
  - Testen
- Ablaufsteuerung
  - „If-then-else“ (Verzweigungen)
  - „Loop“-Variationen
  - „Switch Statements“

Zustandscodes

Implizite Aktualisierung

- Einzel-Bit Register: `CF` Carry Flag `SF` Sign Flag
  `ZF` Zero Flag `OF` Overflow Flag
- implizit gesetzt durch arithmetische Operationen, z.B.:
  `addl Src, Dest` C-Analogon: `t = a + b`
  - CF setzen, wenn höchstwertigstes Bit Übertrag generiert (Überlauf bei Vorzeichenlosen Zahlen)
  - ZF setzen wenn $t = 0$
  - SF setzen wenn $t < 0$
  - OF setzen wenn das Zweierkomplement überläuft:
    $(a > 0 \;\&\&\; b > 0 \;\&\&\; t < 0) \;||\; (a < 0 \;\&\&\; b < 0 \;\&\&\; t >= 0)$

Zustandscodes setzen

Explizite Aktualisierung

- Einzel-Bit Register: `CF` Carry Flag `SF` Sign Flag
  `ZF` Zero Flag `OF` Overflow Flag
- explizites Setzen bei Vergleichsanweisung: `cmpl Src2, Src1`
  - wie Berechnung von `Src1-Src2` (`subl Src2,Src1`), jedoch ohne Abspeichern des Resultats
    - CF setzen, falls höchstwertigstes Bit Übertrag generiert (für Vergleiche von vorzeichenlosen Zahlen)
    - ZF setzen wenn $Src1 = Src2$
    - SF setzen wenn $(Src1 - Src2) < 0$
    - OF setzen wenn das Zweierkomplement überläuft:
      $(a > 0 \&\& b > 0 \&\& (a - b) < 0) || (a < 0 \&\& b > 0 \&\& (a - b) > 0)$
      mit $a = Src1$ und $b = Src2$

Zustandscodes setzen

Explizite Aktualisierung

- Einzel-Bit Register: `CF` `Carry Flag` `SF` `Sign Flag`
  `ZF` `Zero Flag` `OF` `Overflow Flag`
- explizites Setzen mit Testanweisung: `testl Src2, Src1`
  - hilfreich, wenn einer der Operanden eine Bitmaske ist
  - setzt Zustandscodes basierend auf dem Wert von Src1 & Src2
  - wie Berechnung von Src1 & Src2 (`andl Src2,Src1`), jedoch ohne Abspeichern des Resultats
    - ZF setzen wenn $Src1 \& Src2 = 0$
    - Sf setzen wenn $Src1 \& Src2 < 0$

Zustandscodes Lesen

setCC Anweisungen

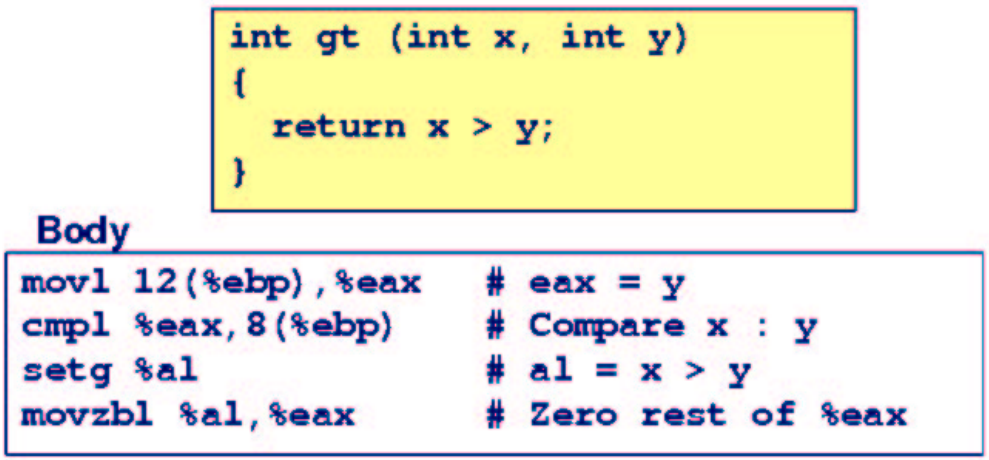
- setzt einzelnes Byte (Ein-Byte Registerelemente) basierend auf Kombinationen von Zustandscodes

setCC	Zustand	Beschreibung
sete	ZF	gleich Null
setne	¬ZF	nicht gleich/nicht Null
sets	SF	negativ
setns	¬SF	nicht-negativ
setg	¬(SF^OF) & ¬ZF	größer (mit Vorzeichen)
setge	¬(SF^OF)	größer oder gleich (mit Vorzeichen)
setl	(SF^OF)	kleiner (mit Vorzeichen)
setle	(SF\^OF)| ZF	kleiner oder gleich (mit Vorzeichen)
seta	¬CF&¬ZF	darüber (ohne Vorzeichen)
setb	CF	darunter (ohne Vorzeichen)

Zustandscodes Lesen

setCC Anweisungen

- ein-Byte-Zieloperand (Register, Speicher)
- meist kombiniert mit `movzbl`
  (um hochwertige Bits zu löschen)



Sprungbefehle („Jump“)

- `Jcc` Anweisungen: Sprung („Jump“) abhängig von Zustandscode „Condition Code (cc)“

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signe
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

„Conditional Branch“ Beispiel

```
int max(int x, int y)
{
  if (x > y)
    return x;
  else
    return y;
}
```

```
_max:
        pushl %ebp
        movl %esp,%ebp

        movl 8(%ebp),%edx
        movl 12(%ebp),%eax
        cmpl %eax,%edx
        jle L9
        movl %edx,%eax
L9:

        movl %ebp,%esp
        popl %ebp
        ret
```

Set Up

Body

Finish

„Do-While“ Loop Beispiel

Goto Version

```
int fact_goto
  (int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

Assembly

```
_fact_goto:
        pushl %ebp    # Setup
        movl %esp,%ebp       # S
        movl $1,%eax  # eax = 1
        movl 8(%ebp),%edx    # x

L11:
        imull %edx,%eax      # r
x
        decl %edx     # x--
        cmpl $1,%edx  # Compare
        jg L11 # if > goto loop

        movl %ebp,%esp       # R
        popl %ebp     # Finish
        ret    # Finish
```

- C erlaubt „goto“ zur Kontrollübertragung
  - Maschinen-Level Programmierungsstil
  - „goto“ als schlechter Programmierstil geächtet!
- benutzt „Backward branch“ um „Looping“ fortzusetzen
- nimmt „Branch“ nur wenn „while“ Bedingung erfüllt

Allgemeine „Do-While“ Übersetzung

C Code

```
int fact_do
  (int x)
{
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);
  return result;
}
```

Goto Version

```
int fact_goto(int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

- Schleifenkörper kann beliebige Folge von C „Statements“ sein
- Abbruchbedingung ist zurückgelieferte Ganzzahl („integer“)
  - $= 0$ als falsch interpretiert; $\neq 0$ als wahr interpretiert

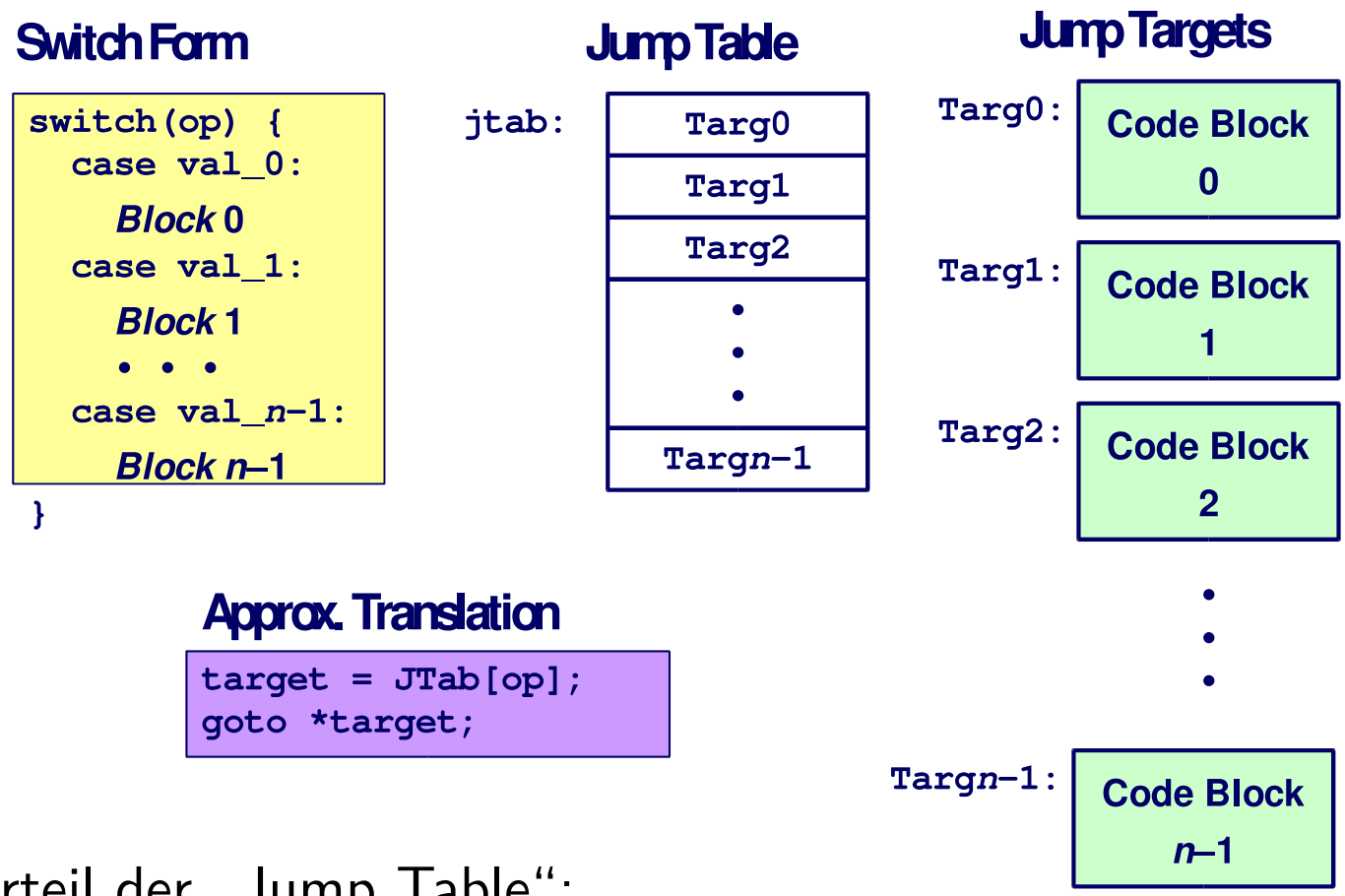
„Switch Statements“

- Implementierungsoptionen
  - Serie von „Conditionals“
    - gut bei wenigen Alternativen
    - langsam bei vielen Fällen
  - „Jump Table“
    - „Lookup Branch Target“
    - Vermeidet „Conditionals“
    - möglich falls Alternativen kleine ganzzahligen Konstanten sind
  - GCC
    - wählt eine der beiden Varianten basierend auf der Fallstruktur
- Bug im Beispielcode
  - keine Vorgabe gegeben

```
typedef enum
 {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
  case ADD :
    return '+';
  case MULT:
    return '*';
  case MINUS:
    return '-';
  case DIV:
    return '/';
  case MOD:
    return '%';
  case BAD:
    return '?';
  }
}
```

„Jump Table“ Struktur



- Vorteil der „Jump Table“:
  - k-way branch in $\mathcal{O}(1)$ Operationen

„Switch Statement“ Beispiel

Branching Possibilities

```
typedef enum
  {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
    • • •
  }
}
```

Enumerated Values

ADD	0
MULT	1
MINUS	2
DIV	3
MOD	4
BAD	5

Setup:

```
unparse_symbol:
        pushl %ebp      # Setup
        movl %esp,%ebp          # Setup
        movl 8(%ebp),%eax       # eax = op
        cmpl $5,%eax    # Compare op : 5
        ja .L49         # If > goto done
        jmp *.L57(,%eax,4)      # goto Table
[op]
```

„Switch Statement“ Beispiel

Assembler-Setup Erkärung

- Symbolische Labels
  - Assembler übersetzt Labels der Form .LXX in Adressen
- „Table“ Struktur
  - Jedes Ziel erfordert 4 Bytes
  - Basisadresse bei .L57
- Jumping:
  - `jmp .L49`
    - „Jump Target“ wird mit Label .L49 gekennzeichnet
  - `jmp * :L57(,%eax, 4)`
    - Start des „Jump Table“ wird mit Label .L57 gekennzeichnet
    - Register %eax hält op
    - mit dem Faktor 4 skaliert, um „offset“ ins „ Table“ zu bekommen
    - holt „Target“ von der effektiven Adresse .L57 + op*4

„Jump Table“ aus „Binary“ Extrahieren

- „Jump Table“ in „Read Only“ Datensegment gespeichert (.rodata)
  - zahlreiche feste Werte von Ihrem Code benötigt
- kann mit obdjump untersucht werden:
  `obdjump code-examples -s --section=.rodata`
  - zeigt alles im angegebenen Segment.
  - schwer zu lesen
    - „Jump Table“ Einträge mit umgekehrter Byte-Anordnung gezeigt

Zusammenfassung – Assembler

- C Kontrollstrukturen
  - „if-then-else“
  - „do-while“
  - „while“
  - „switch“
- Assembler Kontrolle
  - „Jump“
  - „Conditional Jump“
- Compiler
  - generiert Assembler Code, um komplexere Kontrolle zu implementieren
- Standard Techniken
  - Alle „Loops“ konvertiert in „do-while“ Form
  - große „Switch Statements“ verwenden „Jump Tables“

Zusammenfassung – Assembler (cont.)

- CISC Bedingungen
  - CISC Maschinen haben üblicherweise Zustandscode-Register (wie die x86-Architektur)
- RISC Bedingungen
  - keine speziellen Zustandscode-Register
  - benutzen Universalregister um Zustandsinformationen zu speichern
  - spezielle Vergleichs-Anweisungen
  - Z.B. auf Alpha
    `cmple $16, 1, $1`
    - Stellt Register $1 auf 1 wenn Register $16 $<=$ 1

Ergänzende Literatur

Zur Rechnerarchitektur (3. Termin – Assemblerprogrammierung):

[1] Randal E. Bryant and David O'Hallaron.
Computer Systems.
Pearson Education, Inc., New Jersey, 2003.

[2] David A. Patterson and John L. Hennessy.
Computer Organization and Design. The Hardware / Software Interface.
Morgan Kaufmann Publishers, Inc., San Francisco, 1998.

[3] Andrew S. Tanenbaum.
Computerarchitektur.
Pearson Studium München, 2006.

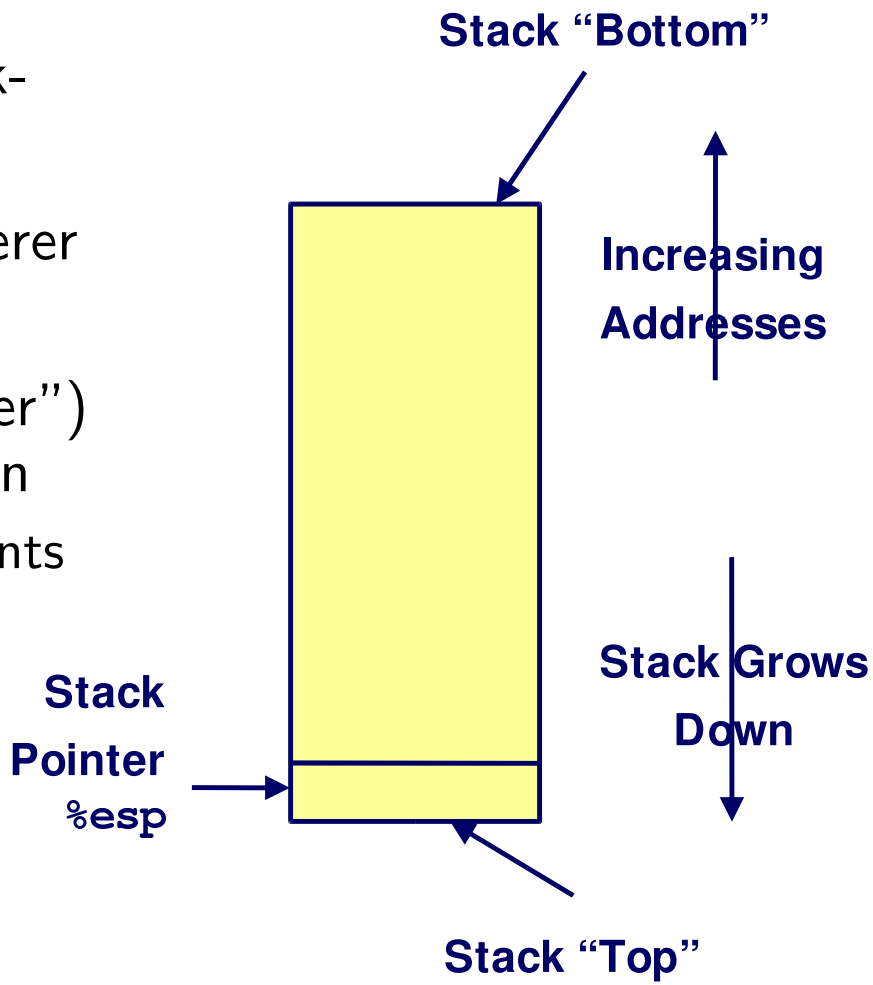
Gliederung

Gliederung (cont.)

11. Parallelrechner

IA32 Kellerspeicher

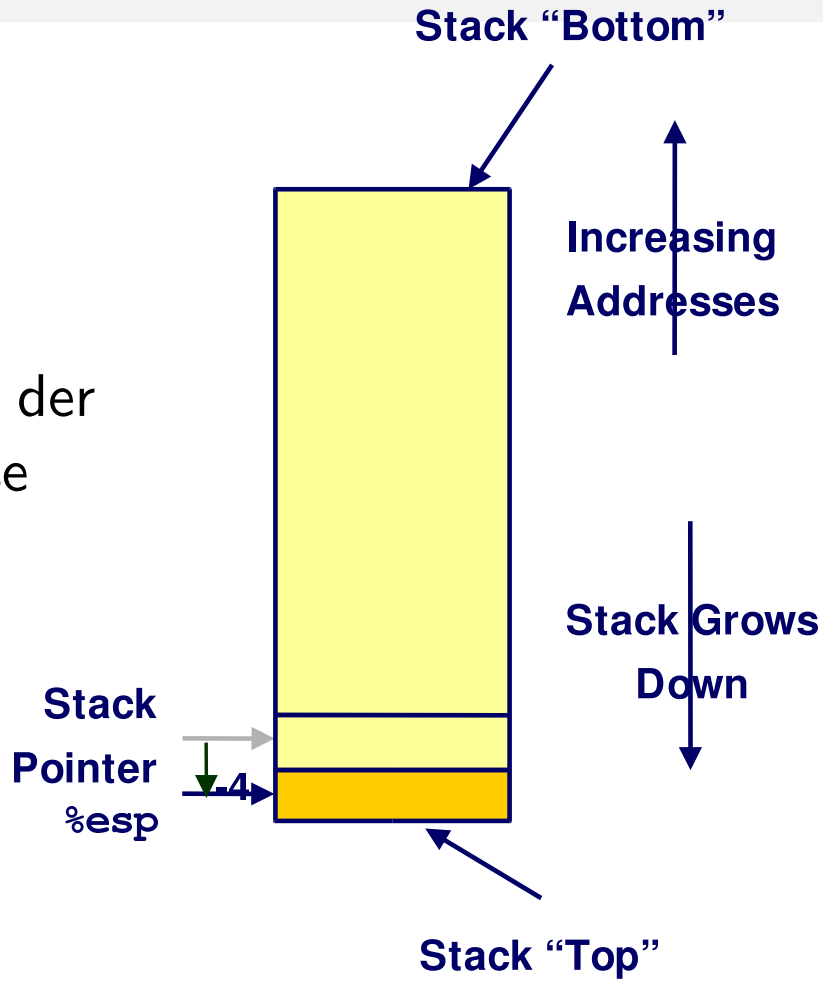
- Speicherregion, die mit Stack-operationen verwaltet wird
- Wächst in Richtung niederigerer Adressen
- Register %esp ("Stack-Pointer") gibt aktuelle Stack-Adresse an
  - Adresse des obersten Elements



IA32 Stack

"Pushing"

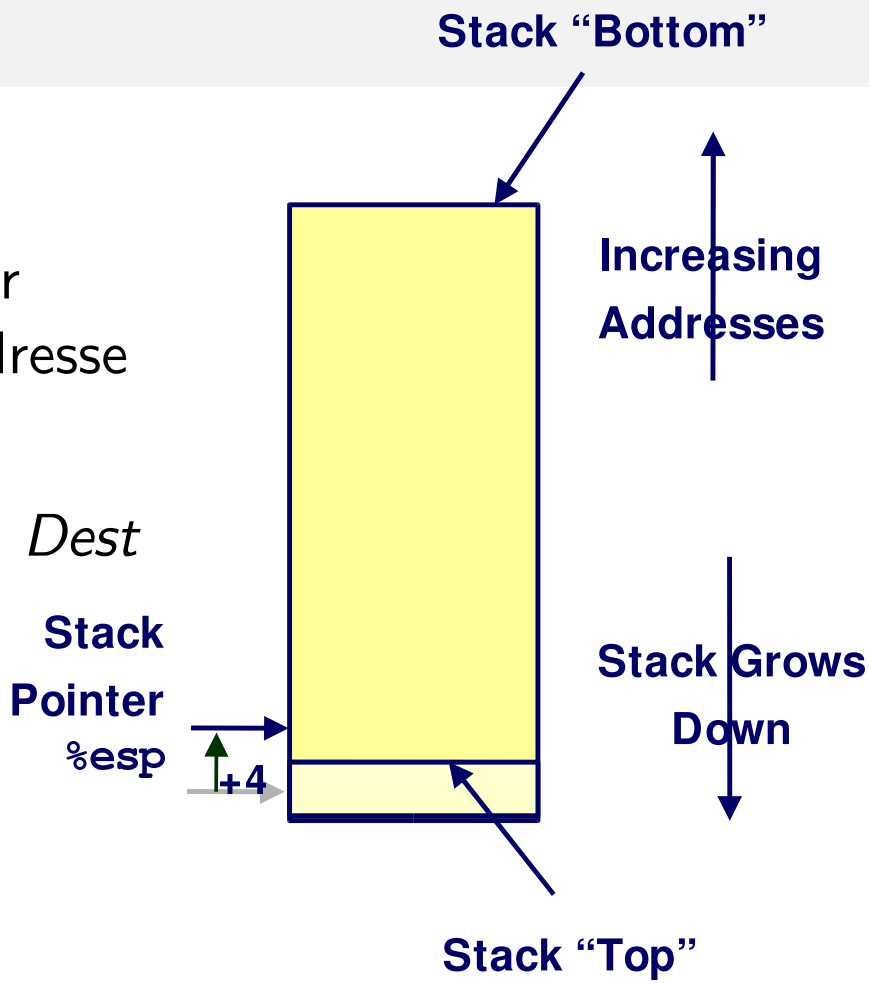
- pushl Src
- holt Operanden aus Src
- dekrementiert %esp um 4
- speichert den Operanden unter der von %esp vorgegebenen Adresse



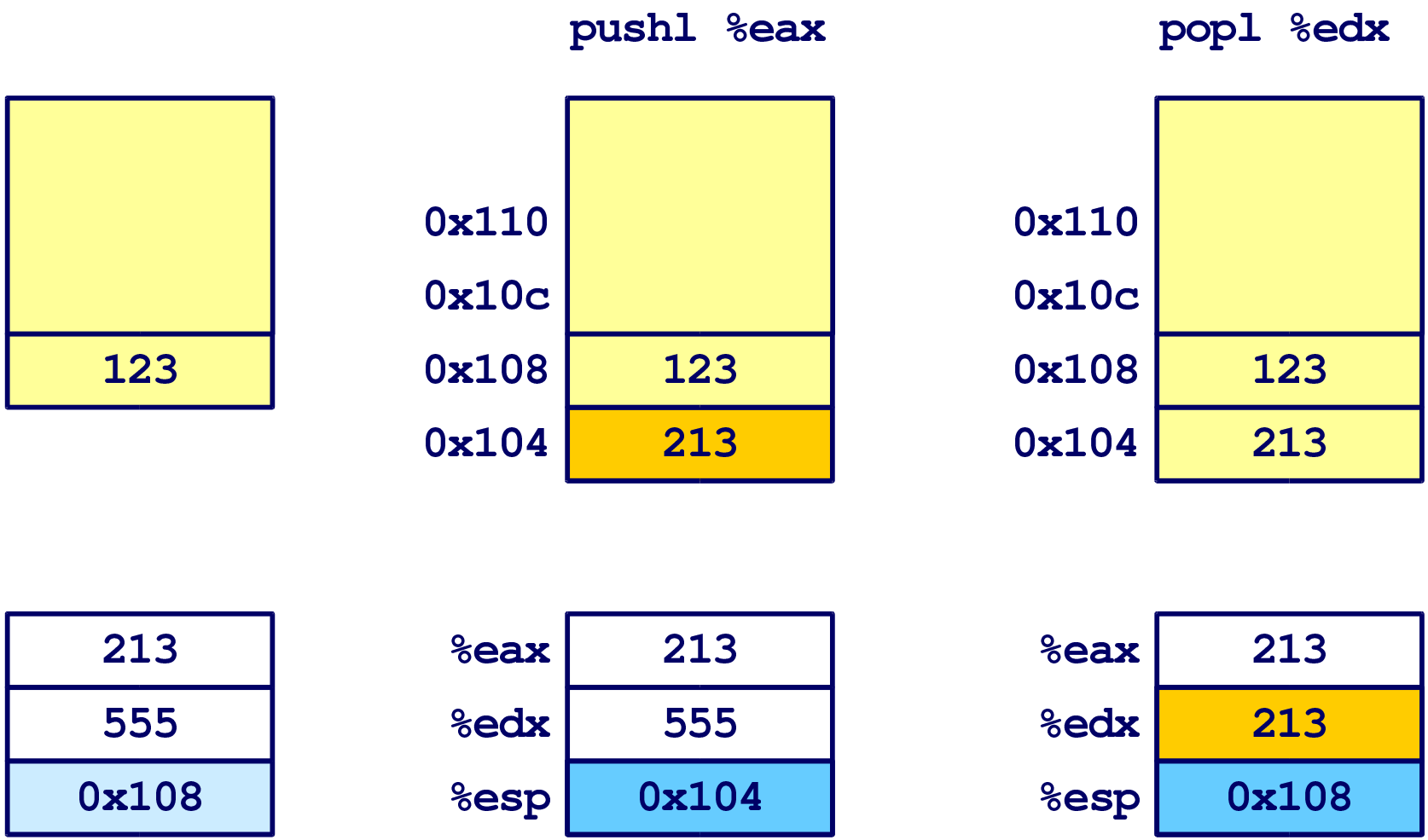
IA32 Stack

"Popping"

- popl Dest
- liest den Operanden unter der durch %esp vorgegebenen Adresse
- inkrementiert %esp um 4
- schreibt gelesenen Wert nach *Dest*



Beispiele für Stack-Operationen



Stack Ablaufsteuerung und Konventionen

Beispiel: Prozedur-Aufruf

- Prozedur-Aufruf ("Call")
  - Benutzt den Stack zur Unterstützung von "Call" und "Return"
    call <label> →"Push" "Return" Adresse auf Stack,
    →"Jump" to <label>
- Wert der "Return" Adresse
  - Adresse der auf den "call" folgenden Anweisung
  - Beispiel:

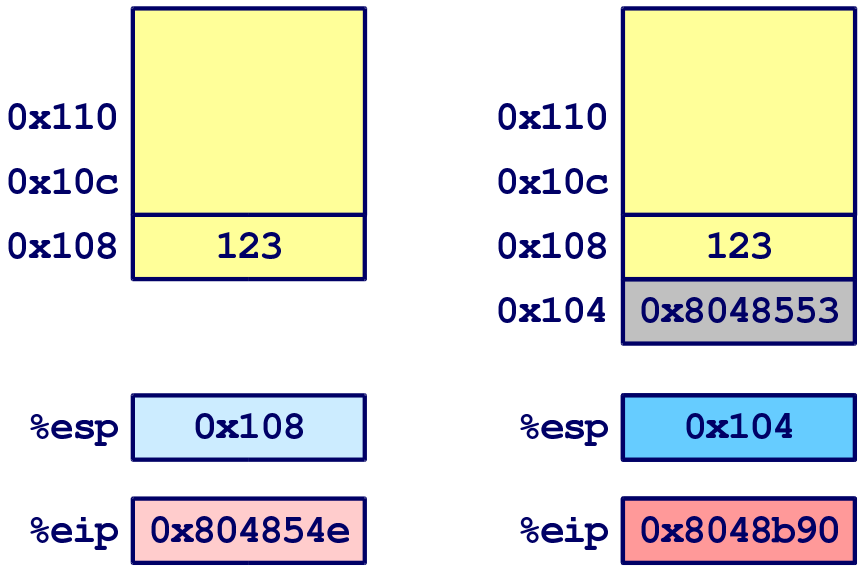
```
804854e:   e8 3d 06 00 00   ;call 8048b90
8048553:   50               ;pushl %eax
< main >   . . .            ;. . .
8048b90:                    ;Prozedureinsprung
. . .      . . .            ;. . .
ret        . . .            ;Rücksprung
```

  - Rücksprung- ("Return") Adresse = 0x8048553
- Prozedur "Return":
  - ret →Pop-Adresse vom Stack,
    →"Jump" zur Adresse

Beispiel Prozeduraufruf (“Call”)

```
804854e:   e8 3d 06 00 00        call   8048b90 <main>
8048553:   50                    pushl  %eax
```

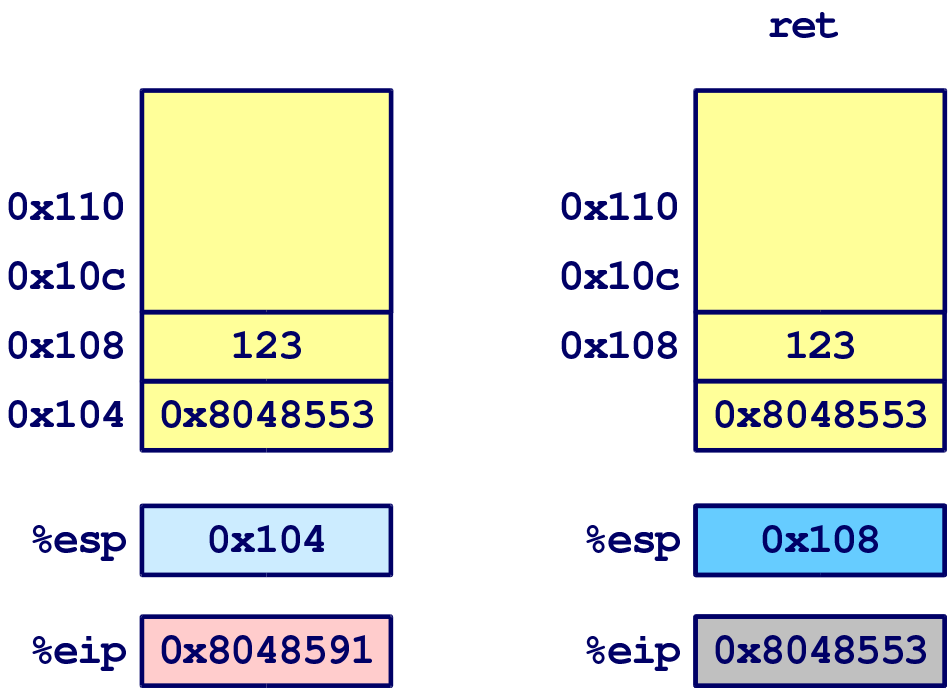
`call 8048b90`



`%eip` is program counter

Beispiel Prozedurrücksprung (“Return”)

```
8048591:   c3                  ret
```



`%eip` is program counter

Stack-basierende Sprachen

- Sprachen, die Rekursion unterstützen
  - z.B., C, Pascal, Java
- Code muss “Reentrant” sein
  - multiple, simultane Instantiierungen einer Prozedur
- braucht Ort, um den Zustand jeder Instantiierung zu speichern
  - Argumente
  - lokale Variable
  - Rücksprungadresse
- Stack-Disziplin
  - Zustand einer Prozedur, der für einen beschränkten Zeitraum benötigt wird
    - von “when called” zu “when return”
  - der “Callee” kommt vor dem “Caller” zurück
- Stack “Frame”
  - Zustand für eine einzelne Prozedur-Instantiierung

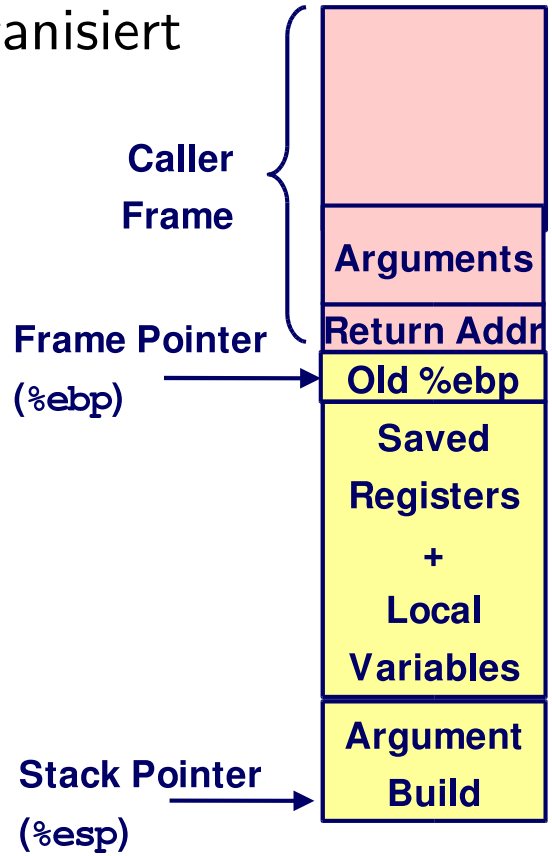
Stack “Frame”

- Inhalt
  - Parameter
  - Lokale Variablen
  - “Return” Adresse
  - Temporäre Daten
- Verwaltung
  - Speicherbereich, der bei “Enter” Prozedur zugeteilt wird
    - “Set-up” Code
  - freigegeben bei “Return”
    - “Finish” Code
- Adressenverweise (“Pointer”)
  - Stack Adressenverweis %esp gibt das obere Ende des Stacks an
  - “Frame” Adressenverweis %ebp gibt den Anfang des aktuellen “Frame” an

IA32/Linux Stack "Frame"

Aktueller Stack "Frame"

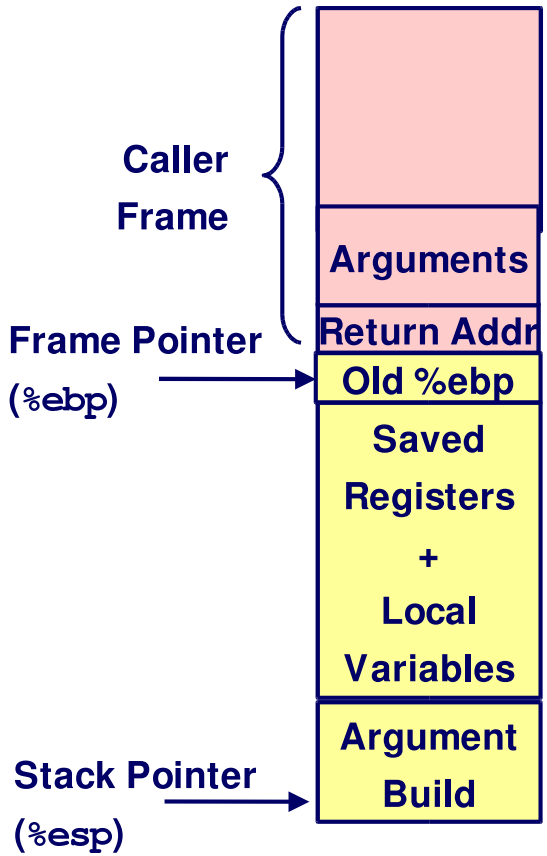
- von oben (Top) nach unten (Bottom) organisiert
- Parameter für eine weitere Funktion, die gleich aufgerufen wird ("call")
- lokale Variablen
  - wenn sie nicht in Registern gehalten werden können
- gespeicherter Register Kontext
- alter "Frame" Adressenverweis



IA32/Linux Stack "Frame"

"Caller" Stack "Frame"

- "Return" Adresse
  - wird von "Call"-Anweisung "gepusht"
- Argumente für aktuellen "Call"



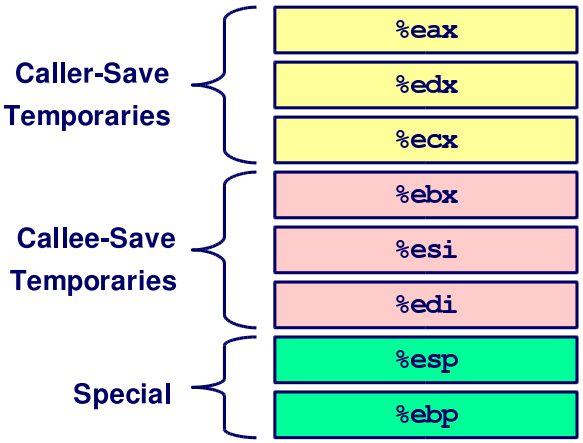
Register Sicherungs-Konventionen

- *yoo* ruft Prozedur *who* auf
  - ist *yoo* der "Caller", *who* ist der "Callee"
- kann "Callee" ein Register für vorübergehende Speicherung benutzen?
  - Inhalt des Registers %edx wird von *who* überschrieben
- Konventionen
  - "Caller" speichert ("Caller Save")
    - "Caller" speichert vor dem "Calling" vorübergehend in seinem Frame
  - "Callee" speichert ("Callee Save")
    - "Callee" speichert vor Verwendung vorübergehend in seinem Frame

IA32/Linux Register Verwendung

Ganzzahlige ("Integer") Register

- zwei werden speziell verwendet
  - %ebp, %esp
- drei werden als "Callee save" verwaltet
  - %ebx, %esi, %edi
  - alte Werte werden vor Verwendung auf dem Stack gesichert
- drei werden als "Caller-save" verwaltet
  - %eax, %edx, %ecx
  - "Caller" sichert diese Register
- Register %eax speichert auch den zurückgelieferten Wert



Rekursive Fakultät

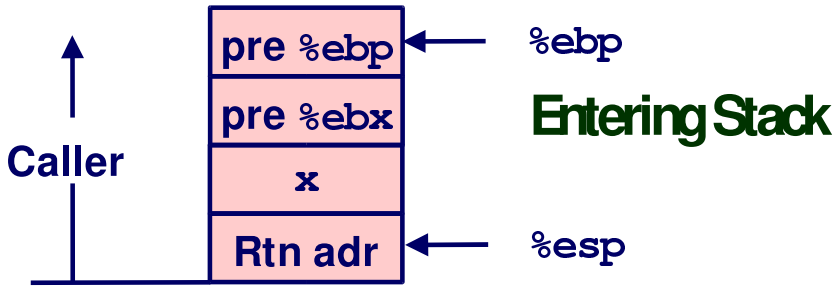
- %eax benutzt ohne vorheriges Speichern

```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

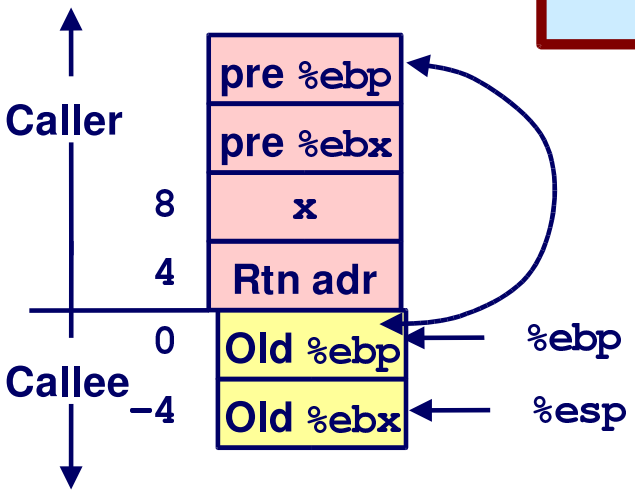
- %ebx benutzt, aber am Anfang gespeichert und am Ende zurück-geschrieben

```
.globl rfact
        .type
rfact,@function
rfact:
        pushl %ebp
        movl %esp,%ebp
        pushl %ebx
        movl 8(%ebp),%ebx
        cmpl $1,%ebx
        jle .L78
        leal -1(%ebx),%eax
        pushl %eax
        call rfact
        imull %ebx,%eax
        jmp .L79
        .align 4
.L78:
        movl $1,%eax
.L79:
        movl -4(%ebp),%ebx
        movl %ebp,%esp
        popl %ebp
        ret
```

Rfact Stack-Aufbau




```
rfact:
        pushl %ebp
        movl %esp,%ebp
        pushl %ebx
```



Rfact Stack-Körper

Recursion

```
  movl 8(%ebp),%ebx # ebx = x
  cmpl $1,%ebx       # Compare x : 1
  jle .L78           # If <= goto Term
  leal -1(%ebx),%eax # eax = x-1
  pushl %eax         # Push x-1
  call rfact         # rfact(x-1)
  imull %ebx,%eax    # rval * x
  jmp .L79           # Goto done
.L78:              # Term:
  movl $1,%eax       # return val = 1
.L79:              # Done:
```

```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1) ;
  return rval * x;
}
```

Registers

%ebx **Stored value of x**

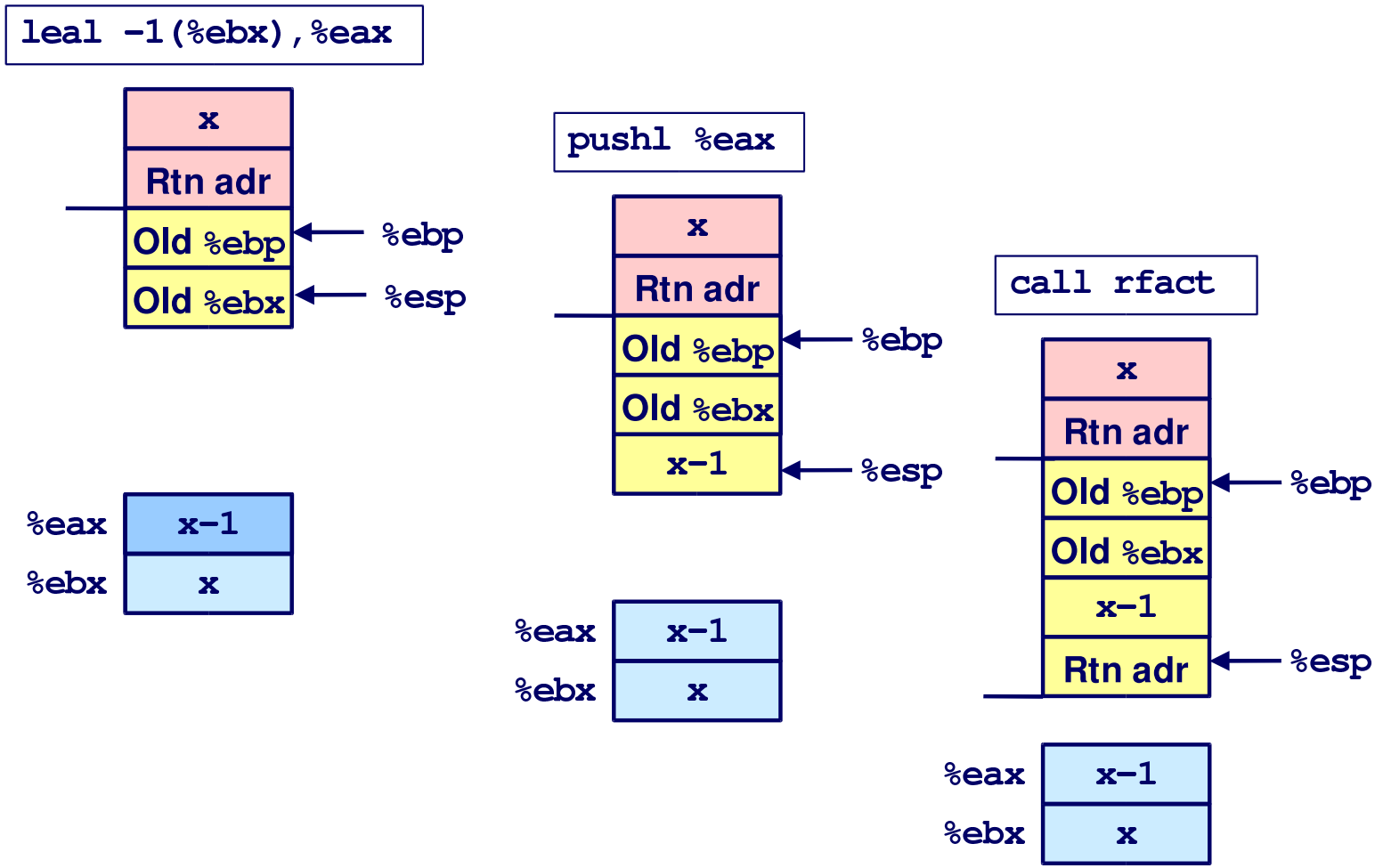
%eax

Temporary value of x-1

Returned value from rfact(x-1)

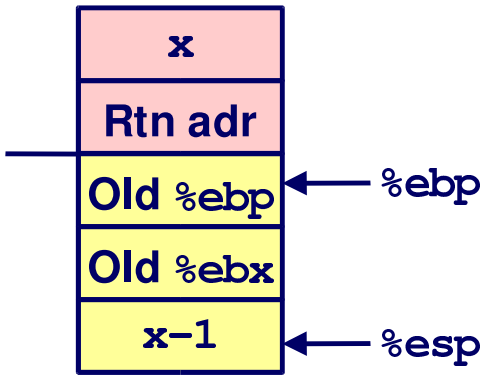
Returned value from this call

Rfact Rekursion

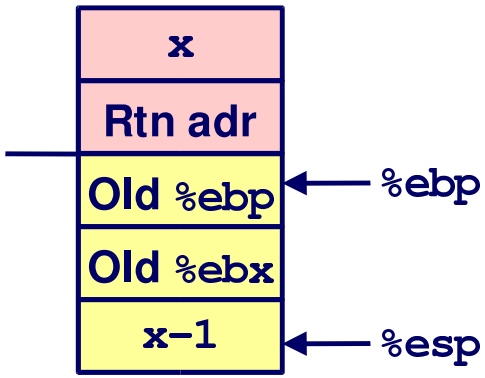


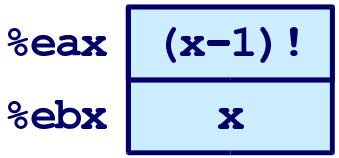
Rfact Ergebnis

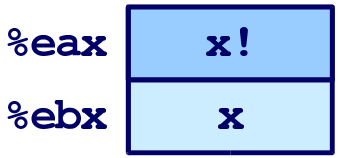
Return from Call




```
imull %ebx,%eax
```



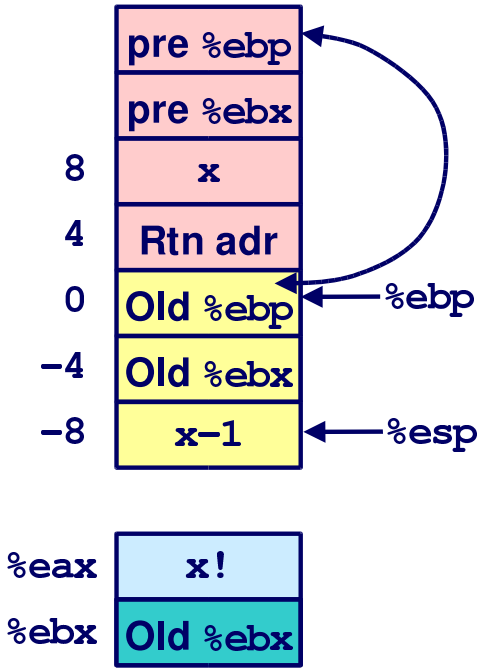


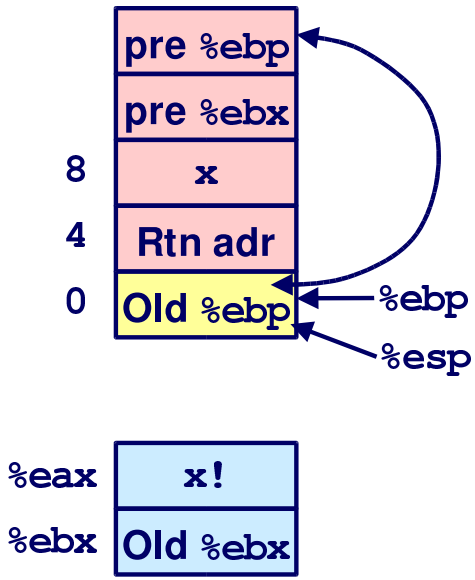


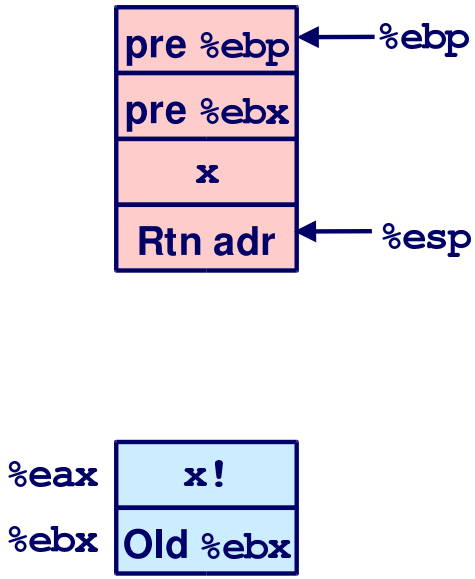
Assume that `rfact(x-1)` returns `(x-1)!` in register `%eax`

Rfact Durchführung

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```







Adressenverweis

- Adressenverweis übergeben ("call by reference"), um Wert einer Variablen zurückschreiben zu können

Recursive Procedure

```
void s_helper
   (int x, int *accum)
{
  if (x <= 1)
    return;
  else {
    int z = *accum * x;
    *accum = z;
    s_helper (x-1,accum);
  }
}
```

Top-Level Call

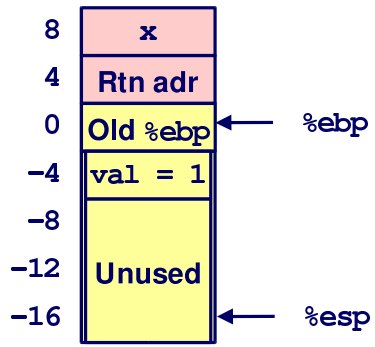
```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

Adressenverweis schaffen und initialisieren

- benutzen des Stacks für Lokale Variable (hier: Variable val)
  - Adressenverweis dorthin muss geschaffen werden
  - berechne Adressenverweis als -4(%ebp)
- Var val auf Stack "pushen": `movl $1, -4(%ebp)`

Initial part of `sfact`

```
_sfact:
        pushl %ebp      # Save %ebp
        movl %esp,%ebp          # Set %ebp
        subl $16,%esp # Add 16 bytes
        movl 8(%ebp),%edx       # edx = x
        movl $1,-4(%ebp)        # val = 1
```




```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

Zusammenfassung: Stack

- Stack ermöglicht Rekursion
  - private Speicherung für jeden Prozedur-“Call”
    - Instantiierungen kommen sich nicht ins Gehege
    - Adressierung von lokalen Variablen und Argumenten kann relativ zu den Positionen des Stacks (“Frame Pointer”sein
  - kann mittels Stack-Disziplin verwaltet werden
    - Prozeduren terminieren in umgekehrter Reihenfolge der “Calls”
- IA32 Prozeduren sind Kombination von Anweisungen + Konventionen
  - “Call” / “Return” Anweisungen
  - Register Verwendungs-Konventionen
    - “Caller/Callee save”
    - %ebp und %esp
  - Organisations-Konventionen des Stack “Frame”

Grundlegende Datentypen

- Ganzzahl (Integer)
  - wird in allgemeinen Registern gespeichert
  - “mit/ohne Vorzeichen”: abhängig von den Anweisungen

Intel	GAS	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	l	4	[unsigned] int

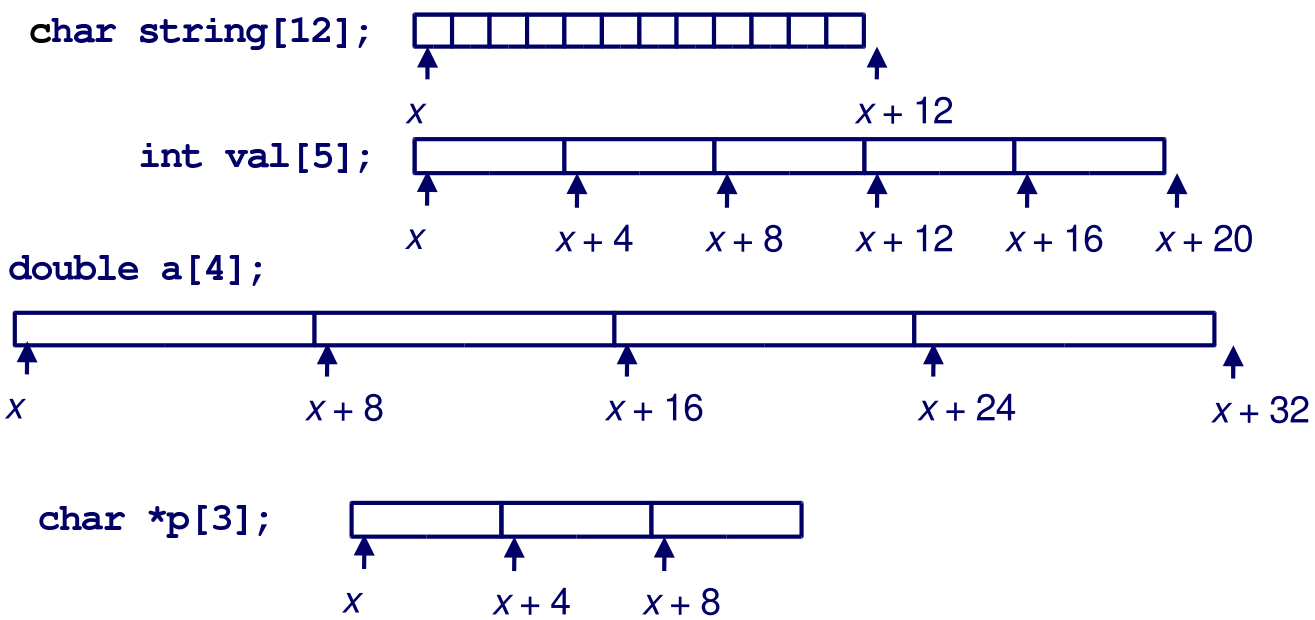
- Gleitkomma (Floating Point)
  - wird in Gleitkomma- Registern gespeichert

Intel	GAS	Bytes	C
Single	s	4	float
Double	l	8	double
Extended	t	10/12	long double

"Array" Zuordnung

Grundlegendes Prinzip

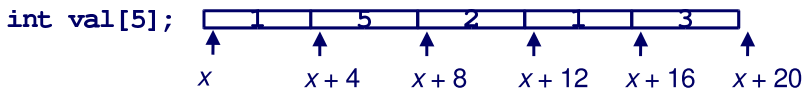
- `T A[l];`
  - "Array"`A` mit Daten des Types `T` und der Länge `l`
  - fortlaufend zugeteilte Region von `l`* sizeof(`T`) Bytes



“Array” Zugang

Konvention

- `T A[l];`
  - “Array”`A` mit Daten des Types `T` und der Länge `l`
  - Bezeichner `A` kann als Adressenverweis verwendet werden, um Element 0 anzuordnen (“to array”)

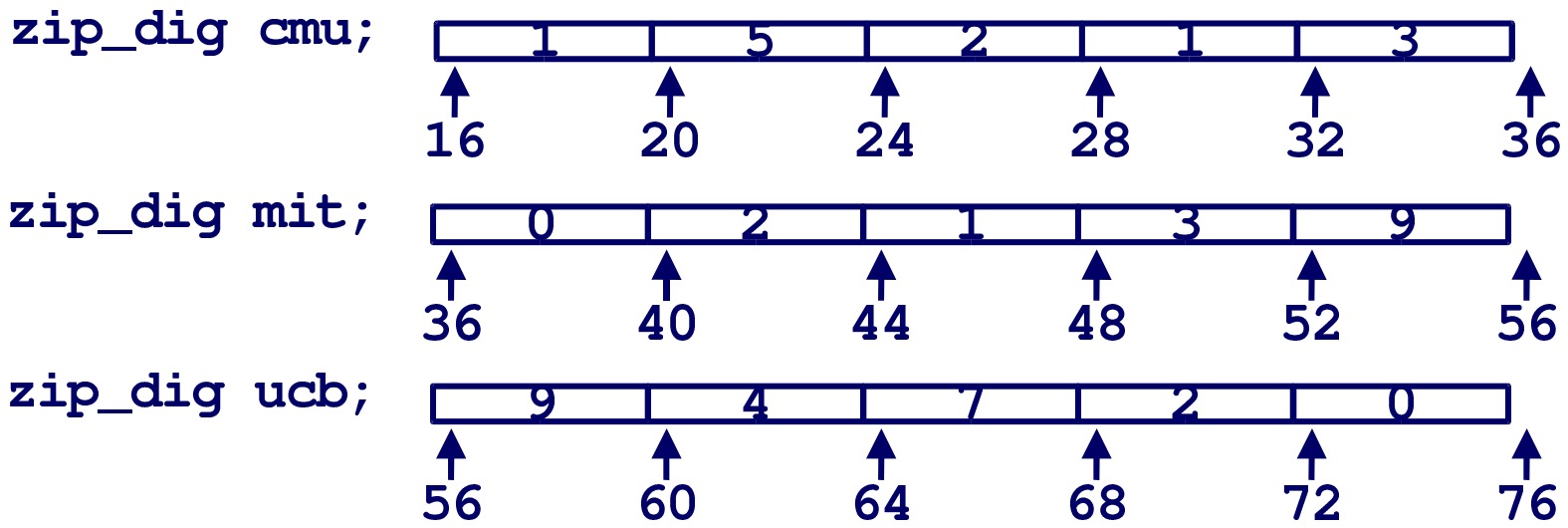


Reference	Type	Value
`val[4]`	`int`	3
`val`	`int *`	x
`val+1`	`int *`	x + 4
`&val[2]`	`int *`	x + 8
`val[5]`	`int`	??
`*(val+1)`	`int`	5
`val + i`	`int *`	x + 4 i

"Array" Beispiel

```
typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



Zugriff auf "Array"

- Referenzierung
  - Code führt keine Bereichsüberprüfung ("bounds checking") durch
  - Verhalten außerhalb des Indexbereichs ist Implementierungs-abhängig
    - keine garantierte relative Zuteilung verschiedener "Arrays"
- Umsetzung durch gcc
  - eliminert "Loop"-Variable *i*
  - konvertiert "Array"-Code zu Adressenverweis-Code
  - wird in "do-while" Form ausgedrückt
    - Test bei Eintritt unnötig

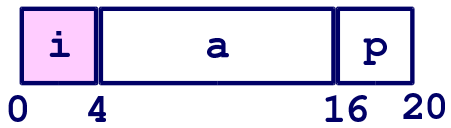
Strukturen

Konzept

- fortlaufend zugeteilte Speicherregion
- bezieht sich mit Namen auf Teile in der Struktur
- Teile können verschiedenen Typs sein

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

Memory Layout




```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

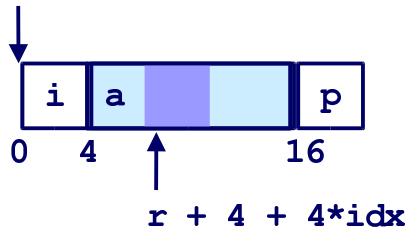
Assembly

```
# %eax = val
# %edx = r
movl %eax,(%edx)     # Mem[r] = val
```

Generieren eines Adressenverweises auf ein Strukturteil

- generieren einer Referenz auf ein "Array"-Element der Struktur
  - "Offset" jedes Strukturteils wird durch Compiler festgelegt

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```



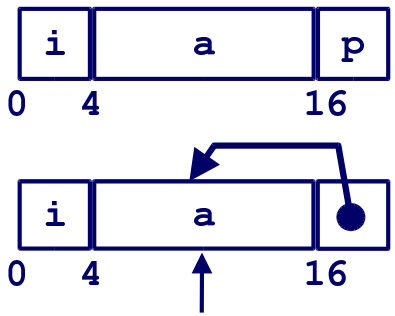

```
int *
find_a
 (struct rec *r, int idx)
{
  return &r->a[idx];
}
```

```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax     # 4*idx
leal 4(%eax,%edx),%eax   # r+4*idx+4
```

Strukturreferenzierung

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

```
void
set_p(struct rec *r)
{
  r->p =
   &r->a[r->i];
}
```



Element `i`

```
# %edx = r
        movl (%edx),%ecx      # r->i
        leal 0(,%ecx,4),%eax  # 4*(r->i)
        leal 4(%edx,%eax),%eax      # r+4+4*(r->i)
        movl %eax,16(%edx)    # Update r->p
```

Zusammenfassung: Datentypen

- "Arrays" in C
  - fortlaufend zugeteilter Speicher
  - Adressverweis auf das erste Element
  - keine Bereichsüberprüfung ("Bounds Checking")
- Compiler Optimierungen
  - Compiler wandelt Array Code oft in Adressverweise ("pointer code") um
  - verwendet Adressierungsmodi um "Array"-Indizes zu skalieren
  - viele Tricks, um die "Array"-Indizierung in Schleifen zu verbessern
- Strukturen
  - Bytes werden in der ausgewiesenen Reihenfolge zugeteilt
  - in der Mitte und am Ende polstern, um die richtige Ausrichtung zu erreichen

Ergänzende Literatur

Zur Rechnerarchitektur (Teil 2, 4. Termin):

[1] Randal E. Bryant and David O'Hallaron.
Computer systems.
pages 1000–2000. Pearson Education, Inc., New Jersey, 2003.

Gliederung

Gliederung (cont.)

10. Ausnahmebehandlungen und Prozesse
11. Parallelrechner

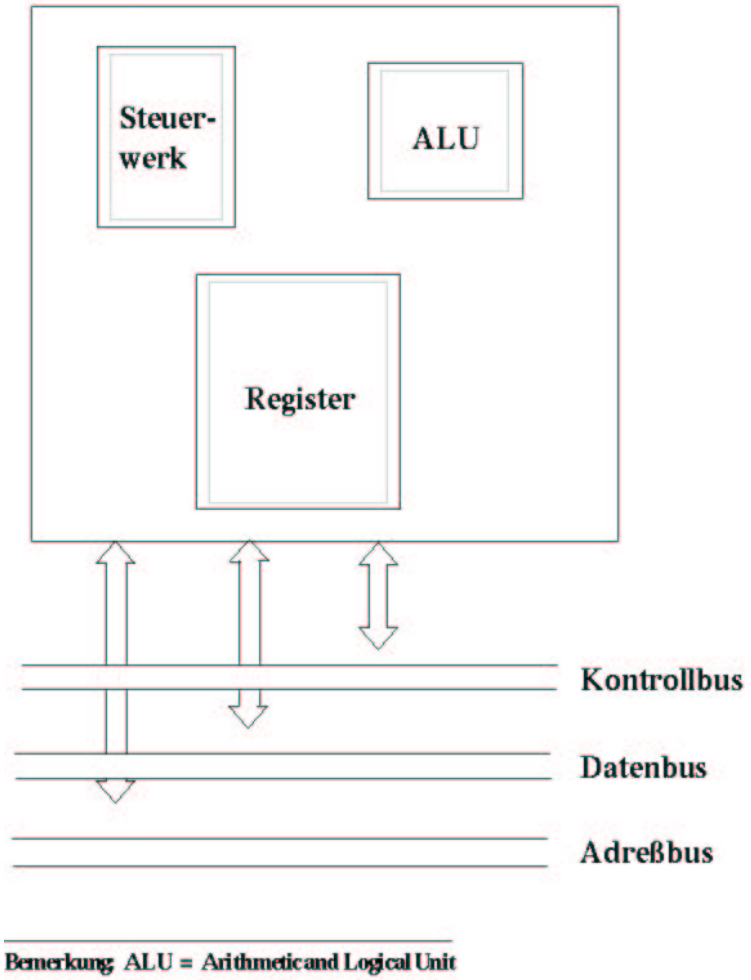
Computerarchitektur

Überblick

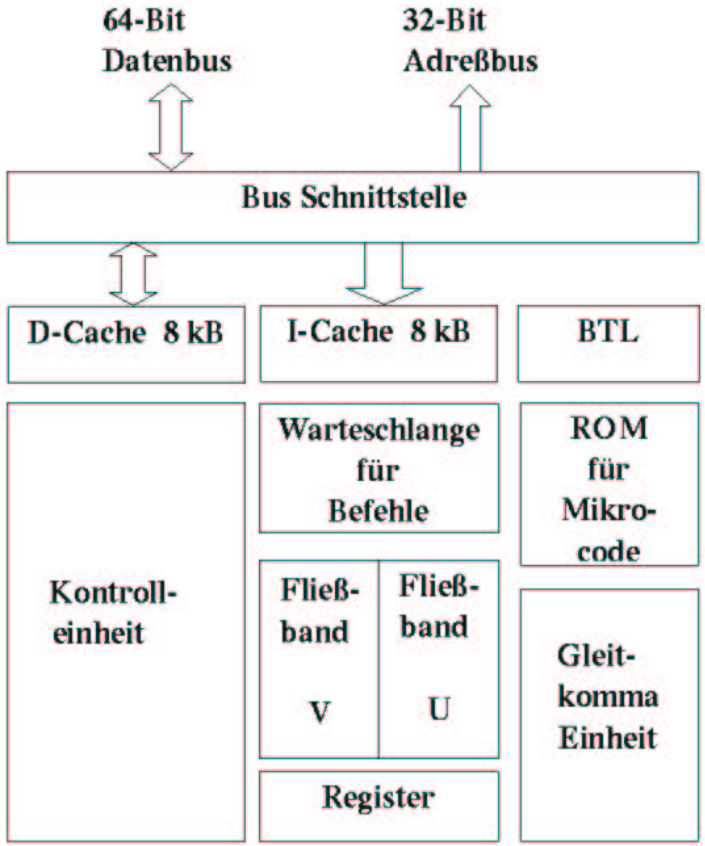
- Hintergrund
  - Befehlssätze
  - Logikdesign
- Sequenzielle Implementierung
  - einfaches, aber langsames Prozessordesign
- "Pipelined Implementation"
  - Prozessordesign unter Einsatz des "Pipelining"-Prinzips
    - lässt mehr Vorgänge gleichzeitig ablaufen
- Fortgeschrittene Themen
  - Leistungsanalyse
  - Hochleistungsprozessor Design

Grundlagen

Bild einer Zentraleinheit:



Pentium Blockdiagramm

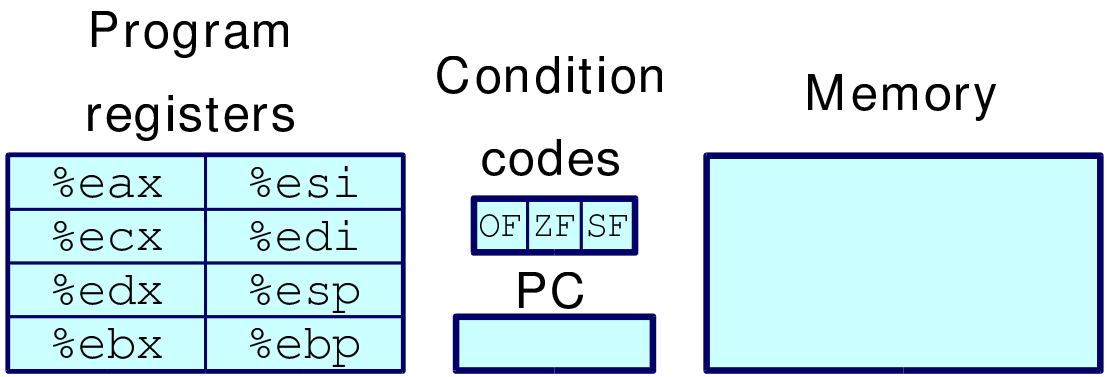


Bemerkung: BTL = Branch Trace Logic;
der Datencache nutzt ein MESI-Protokoll.

Vorgehensweise

- mit Design eines speziellen Befehlssatzes arbeiten
  - Y86 - eine vereinfachte Version des Intel IA32 (a.k.a. x86)
  - ist einer bekannt, sind mehr oder weniger alle beaknnt
- auf “Mikroarchitekturebene” arbeiten
  - grundlegende Hardwareblöcke per Assembler in die allgemeine Prozessorstruktur integrieren
    - Speicher, funtionale Einheiten, etc.
  - mit Kontrolllogik sorgt für den richtigen Ablauf jeder Anweisung
- einfache Hardware-Beschreibungssprache zur Beschreibung der Kontrolllogik benutzen
  - lässt sich erweitern und modifizieren
  - mit Simulationen testen

Y86 Prozessor Zustand



- Programmregister:
  - 8 Register, wie IA32
  - jeweils 32 bit breit
- Zustandscodes:
  - 3-Bit Flagregister, gesetzt durch ALU
  - OF: Overflow, ZF: Zero, SF: Negativ
- Programmzähler:
  - Adresse der nächsten auszuführenden Anweisung
- Speicher:
  - Byte-adressierbarer Speicher
  - Wörter werden im “little-endian”-Format gespeichert

Befehlssätze

Format der Y86 Anweisungen

- variable Befehlslänge
  - Befehle 1 – 6 Byte lang
  - Länge der Anweisung kann aus dem ersten Byte bestimmt werden
  - weniger Anweisungstypen als bei IA32
  - einfachere Kodierung als bei IA32
- Anweisung greift auf einen Teil des Programmzustands zu und modifiziert ihn

Registerkodierung

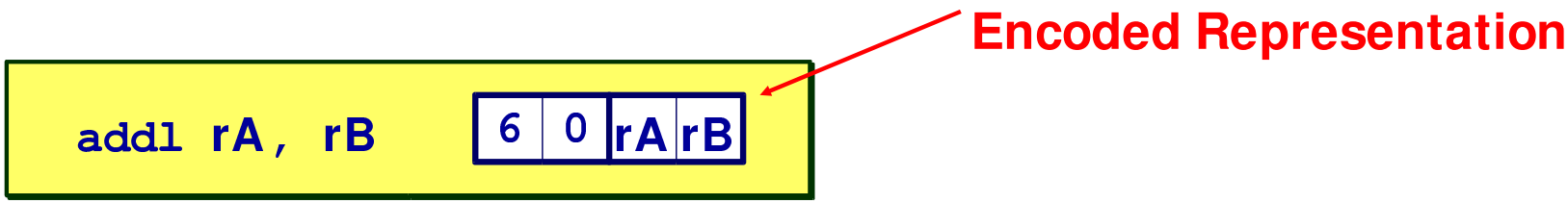
- Registeri-ID 4-Bit breit

%eax	0
%ecx	1
%edx	2
%ebx	3

%esi	6
%edi	7
%esp	4
%ebp	5

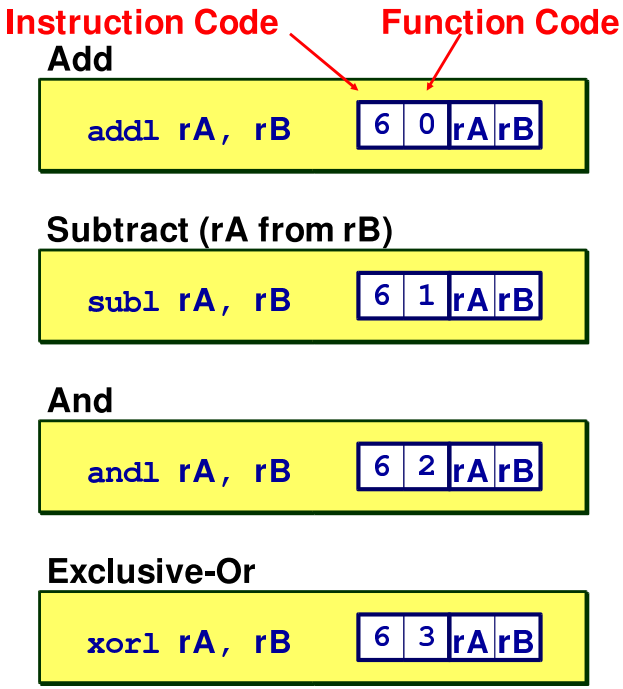
- Kodierung wie bei IA32
- Register ID 8 gibt “kein Register” an
  - wird an zahlreichen Stellen im Hardwaredesign einsetzt

Anweisungsbeispiel – Addition



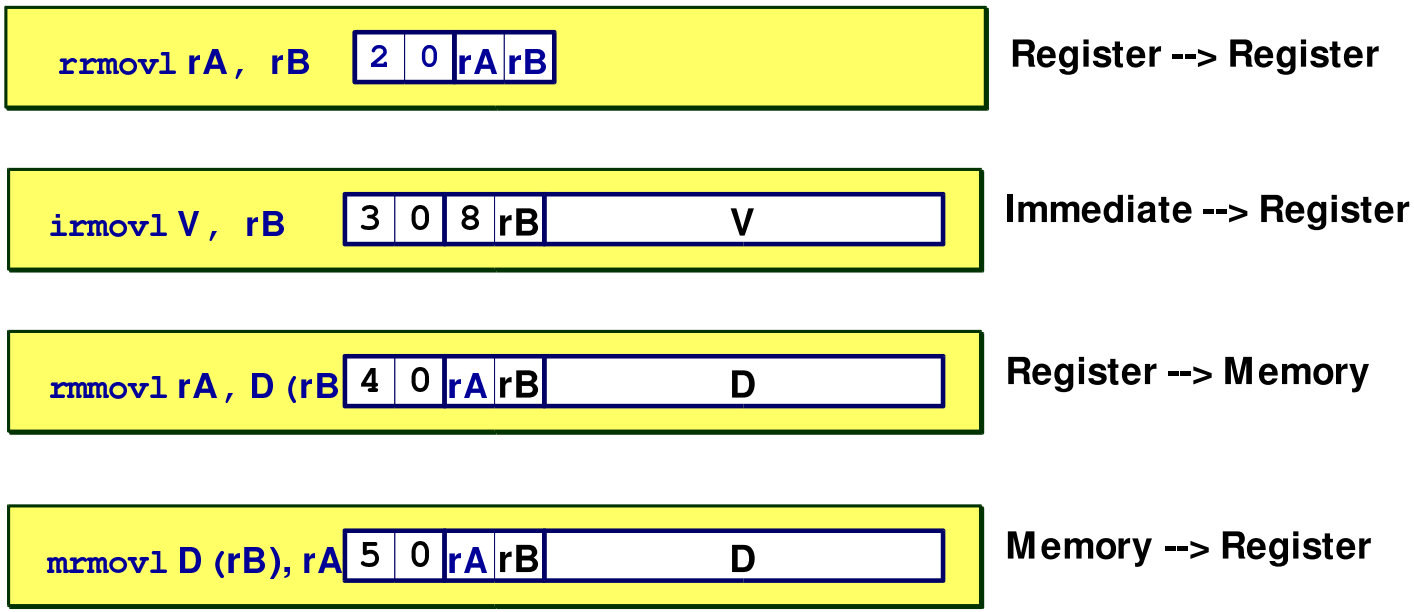
- addiert Wert im Register rA zu dem im Register rB
  - speichert Resultate im Register rB
  - merke: Y86 erlaubt Addition nur auf Registern
- setzt Zustandscodes je nach Ergebnis
- z.B., addl %eax, %esi Kodierung: 60 06
- zwei-Byte Kodierung
  - erstes Byte gibt den Anweisungstyp an
  - zweites Byte gibt die Quell- und Zielregister an

Arithmetische und logische Operationen



- Kodierungen unterscheiden sich nur durch “Funktionscode”
  - niederwertiges Nibble (4-bit) des ersten Anweisungswortes
- setzt Zustandscodes als Nebeneffekt

“Move” Operationen



- wie die IA32 movl Anweisung
- einfacheres Format für Speicheradressen
- verschiedene Namen, um sie auseinander zu halten

CISC Befehlssätze

- "Complex Instruction Set Computer"
- "Stack"-orientierter Befehlssatz
  - Übergabe von Argumenten
  - Speichern des Programmzählers
  - explizite "Push" und "Pop" Anweisungen
- arithmetische Anweisungen können auf Speicher zugreifen
  - addl %eax, 12(%ebx, %ecx, 4)
    - Erfordert Schreib- und Lesezugriff auf den Speicher
    - Komplexe Adressberechnung
- Zustandscodes ("Flags")
  - gesetzt durch arithmetische und logische Anweisungen
- Philosophie
  - Hinzufügen von Maschinen-Anweisungen, um "typische" Programmieraufgaben durchzuführen

RISC Befehlssätze

- "Reduced Instruction Set Computer"
- Internes Projekt bei IBM
  - von Hennessy (Stanford) und Patterson (Berkeley) publiziert
- wenige und einfache Anweisungen
  - benötigen i. d. Regel mehr Anweisungen für eine Aufgabe
  - werden aber mit kleiner, schneller Hardware ausgeführt
- Register-orientierter Befehlssatz
  - viele (üblicherweise 32) Register
  - Register für Argumente, "Return"-Adressen, Zwischenergebnisse
- nur "load" und "store" Anweisungen können auf Speicher zugreifen
  - wie bei Y86 mrmovl und rmmovl
- Keine Zustandscodes
  - Testanweisungen hinterlassen Resultat (0/1) direkt im Register

CISC gegen RISC

Ursprüngliche Debatte

- streng geteilte Lager
- CISC Anhänger - einfach für den Compiler; weniger Code Bytes
- RISC Anhänger - besser für optimierende Compiler; schnelle Abarbeitung auf einfacher Hardware

Aktueller Stand

- Grenzen verwischen
  - RISC-Prozessoren werden komplexer
  - CISC-Prozessoren weisen RISC-Konzepte oder gar RISC-Kern auf
- für Desktop Prozessoren ist die Wahl der ISA kein Thema:
  - Code-Kompatibilität ist sehr wichig!
  - mit genügend Hardware wird alles schnell ausgeführt
- eingebettete Prozessoren: eindeutige RISC-Orientierung
  - kleiner, billiger, weniger Leistung

Zusammenfassung

Y86 Befehlssatz Architektur

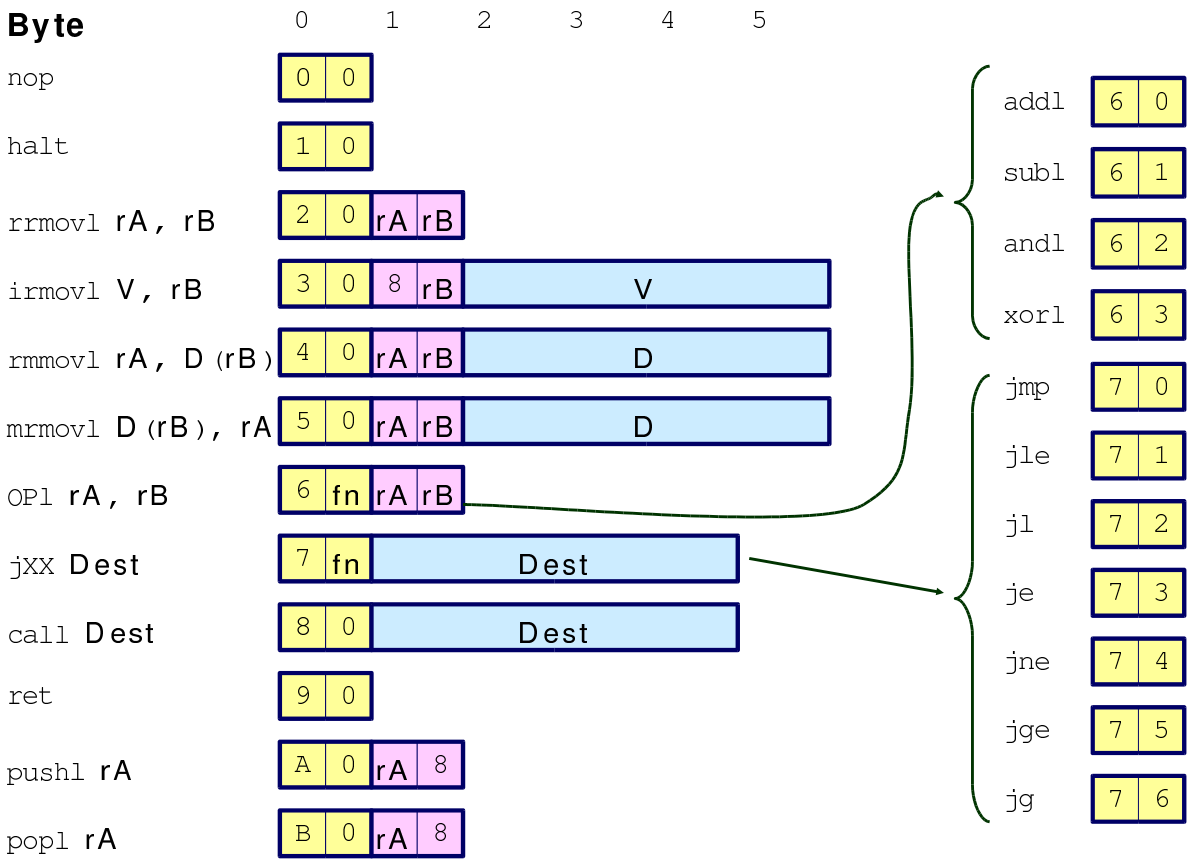
- gleiche Zustände und Anweisungen wie IA32
- simplere Kodierungen
- irgendwo zwischen CISC und RISC

ISA Design heute

- Restriktionen durch Hardware abgeschwächt
- Code-Kompatibilität leichter zu erfüllen
  - Emulation in Firm- und Hardware
- Intel bewegt sich weg von IA32
  - erlaubt nicht genug Parallelität
- hat IA64 eingeführt ("Intel Architecture 64-Bit")
  - → neuer Befehlssatz mit expliziter Parallelität (EPIC)
  - → 64-Bit Wortgrößen (überwinden Adressraumlimits)
  - → Benötigt hoch entwickelte Compiler

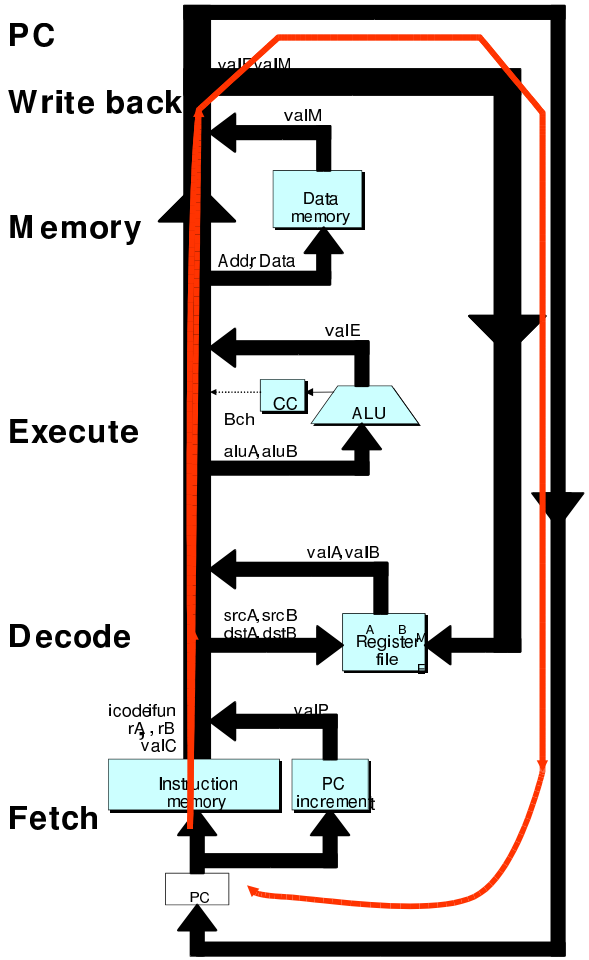
Sequenzielle Implementierung

Y86 Befehlssatz



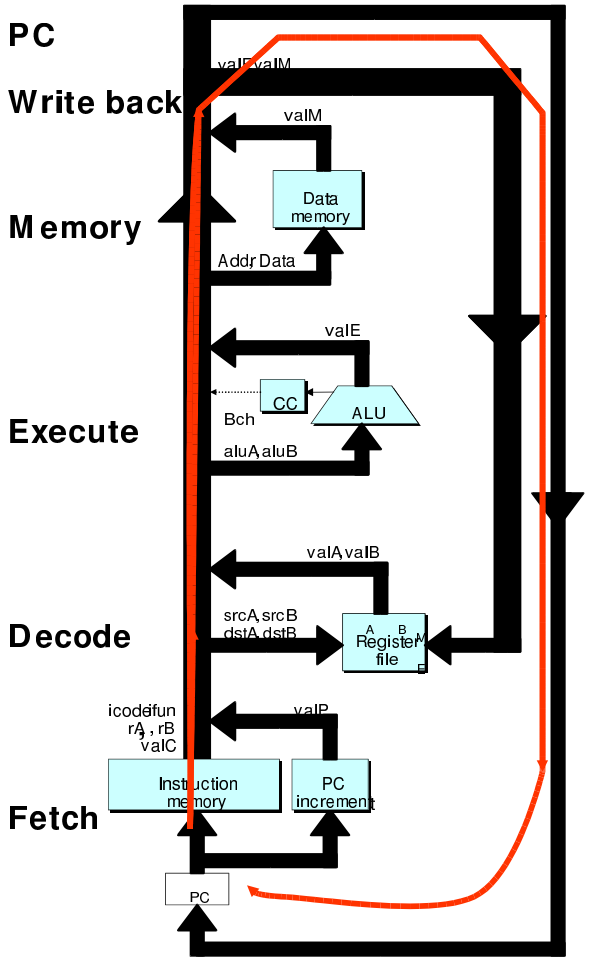
SEQ Hardware Struktur

- Zustand
  - Programmzähler Register (PC)
  - Zustandscode Register (CC)
  - Registerbank
  - Speicher
    - Gemeinsamer Speicherraum für Daten und Anweisungen
- Abarbeitung von Anweisungen
  - Lesen der Anweisung unter der vom PC angegebenen Adresse
  - Verarbeitung durch die Stufen
  - Aktualisierung des Programmzählers

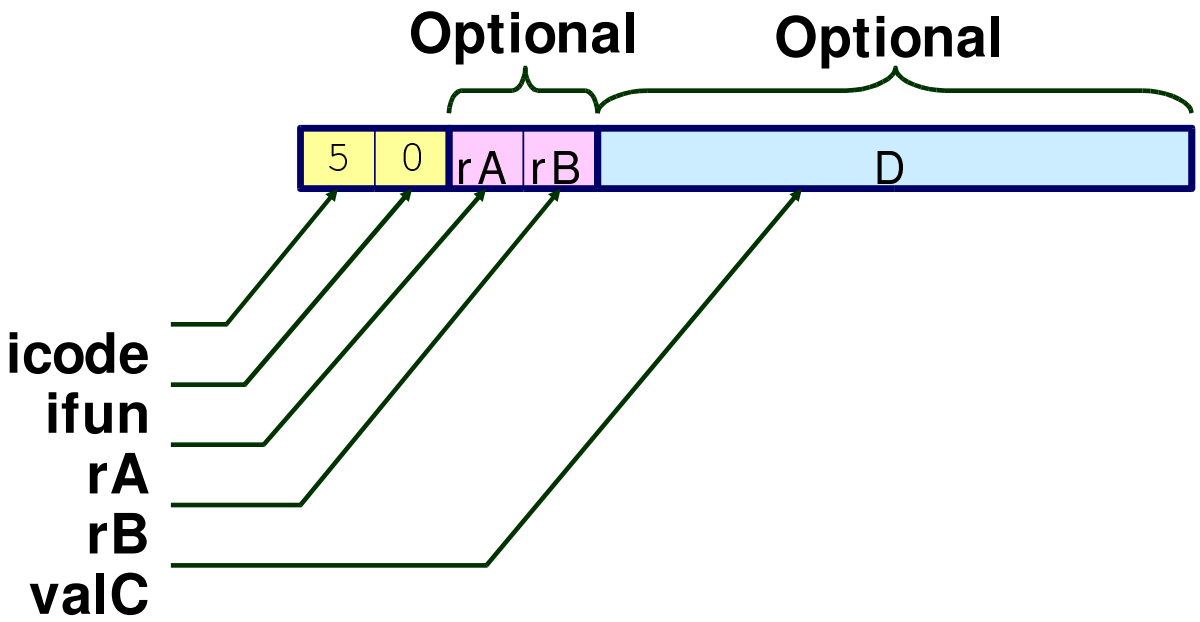


SEQ Stufe

- Holen (Fetch)
  - Anweisung aus Speicher lesen
- Dekodieren (Decode)
  - lese Programmregister
- Ausführen (Execute)
  - berechne Wert oder Adresse
- Speichern (Memory)
  - lese oder schreibe Daten
- Zurückschreiben (Write Back)
  - schreibe in Registerbank
- PC Schreiben
  - aktualisiere Programmzähler



Anweisungsdekodierung



Anweisungsformat:	Anweisungs-Byte	icode:ifun
	optionales Register Byte	rA:rB
	optionale Konstante	valC

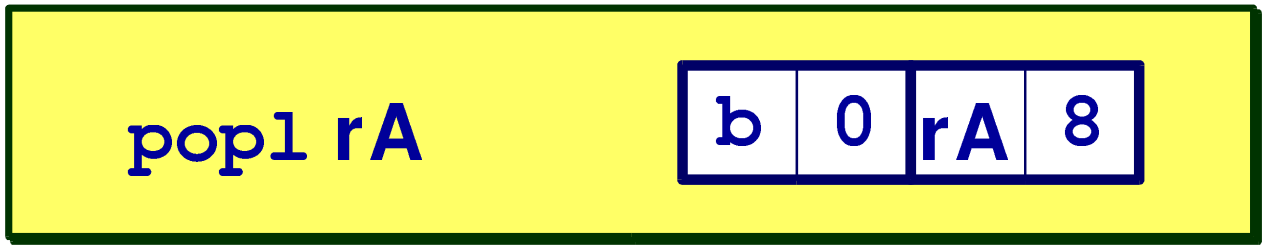
Ausführen einer Arith./Logischen Operation

Holen	lese 2 Bytes, berechne PC+2
Dekodieren	lese Operanden Register
Ausführen	Alu Operation, setze Zustandscodes
Speichern	tue nichts
Zurückschreiben	aktualisiere Register
PC Schreiben	erhöhe PC um 2

Ausführen von `rmmovl`

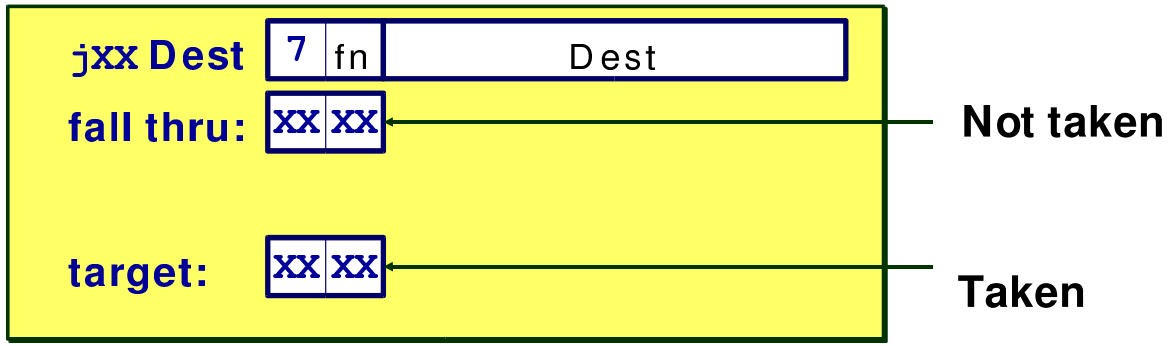
Holen	lese 6 Bytes, berechne PC+6
Dekodieren	lese Operanden Register
Ausführen	berechne geltende Adresse
Speichern	schreibe in Speicher
Zurückschreiben	tue nichts
PC Screiben	erhöhe PC um 6

Ausführen von popl



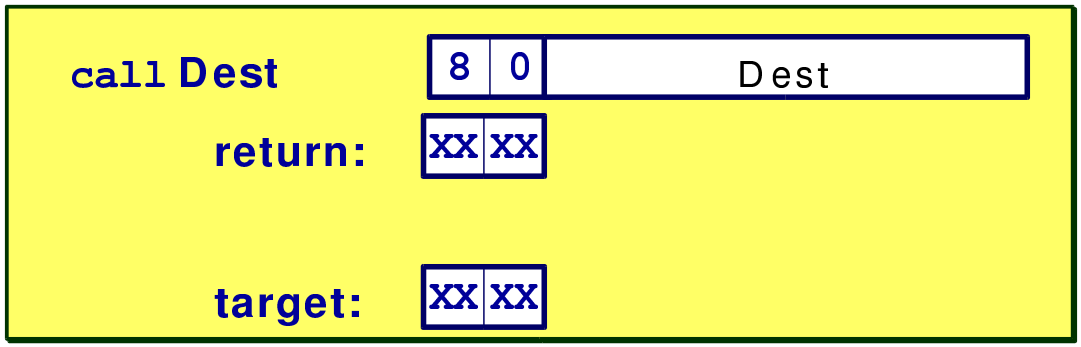
Holen	lese 2 Bytes, berechne PC+2
Dekodieren	lese Stack Adressverweis (Stack Pointer)
Ausführen	erhöhe Stack Adressverweis um 4
Speichern	lese aus altem Stack Adressverweis
Zurückschreiben	aktualisere Stack Adressverweis, schreibe Resultat ins Register
PC Schreiben	erhöhe PC um 2

Ausführen von `Jump`



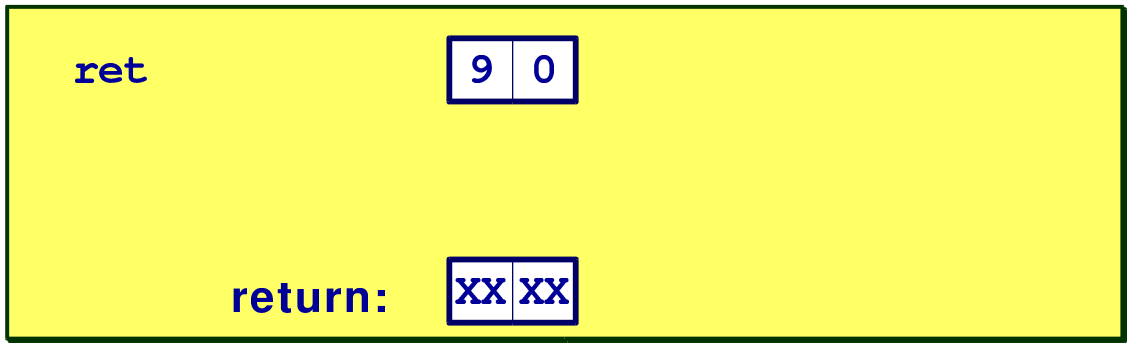
Holen	lese 5 Bytes, berechne PC+5
Dekodieren	tue nichts
Ausführen	entscheide, ob abhängig vom Sprungbefehl (Jump) und Zustandscode gesprungen wird
Speichern	tue nichts
Zurückschreiben	tue nichts
PC Schreiben	setze PC auf $\begin{cases} \texttt{Dest} & \text{wenn Sprung} \\ PC+5 & \text{sonst} \end{cases}$

Ausführen von `call`



Holen	lese 5 Bytes, berechne PC+5
Dekodieren	lese Stack Adressverweis
Ausführen	verringere Stack Adressverweis um 4
Speichern	speichere erhöhten PC unter neuen Wert des Stack Adressverweises
Zurückschreiben	aktualisiere Stack Adressverweis
PC Schreiben	setze PC auf `Dest`

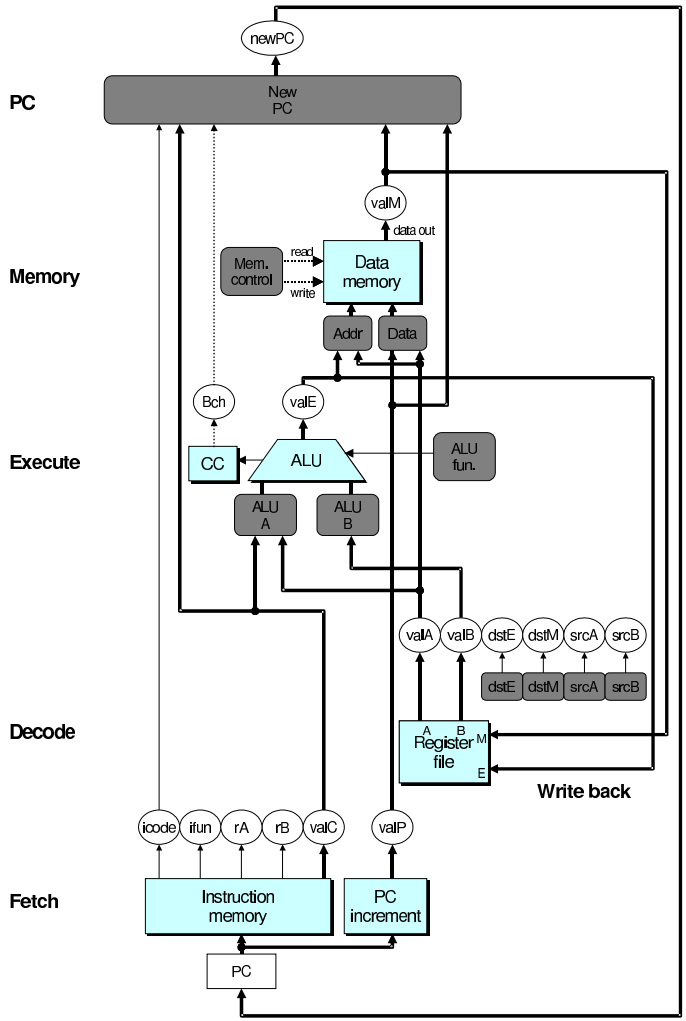
Ausführen von `ret`



Holen	lese 1 Byte
Dekodieren	lese Stack Adressverweis
Ausführen	erhöhe Stack Adressverweis um 4
Speichern	lese “Return”-Adresse unter alten Stack Adressverweises
Zurückschreiben	aktualisiere Stack Adressverweis
PC Schreiben	setze PC auf “Return”-Adresse

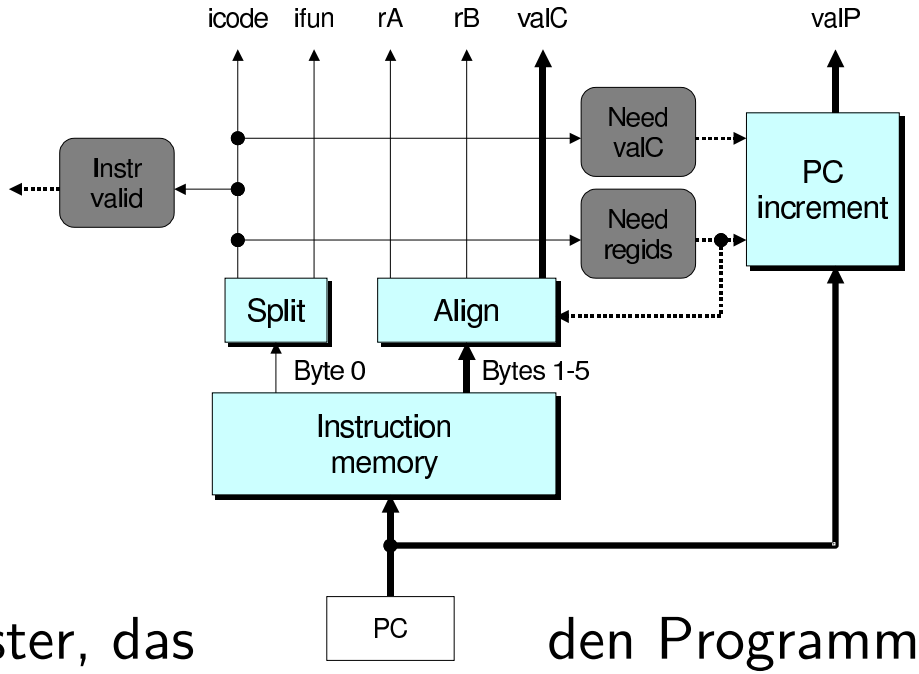
SEQ Hardware

- Blaue Kästchen: vorher entworfene Hardwareblöcke
  - z.B. Speicher, ALU
- Graue Kästchen: Kontrolllogik
  - Wird in HCL beschrieben
- Weiße Ovale: Label für Signale
- Fette Linien: 32-Bit Wortwerte
- Schmale Linien: 4-8 Bit Werte
- Gestrichelte Linien: 1-Bit Werte



Holen ("Fetch") Logik

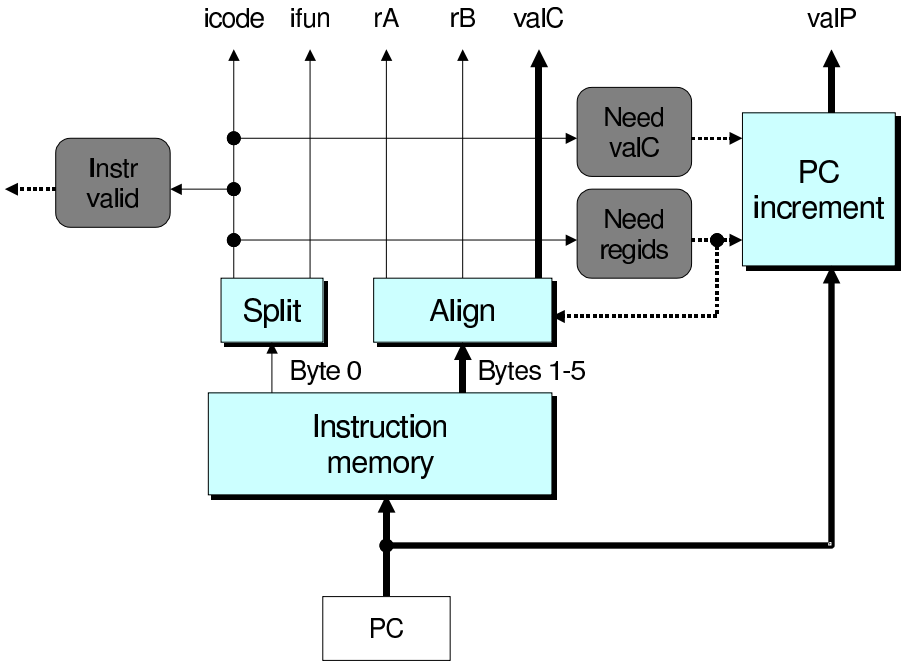
Vordefinierte Blöck



- "PC": Register, das den Programmzähler enthält
- "Instruction memory": Liest 6 Bytes (PC nach PC+5)
- "Split": Teilt Anweisungsbyte in icode und ifun auf
- "Align": Holt Felder für rA, rB, und valC

Holen ("Fetch") Logik

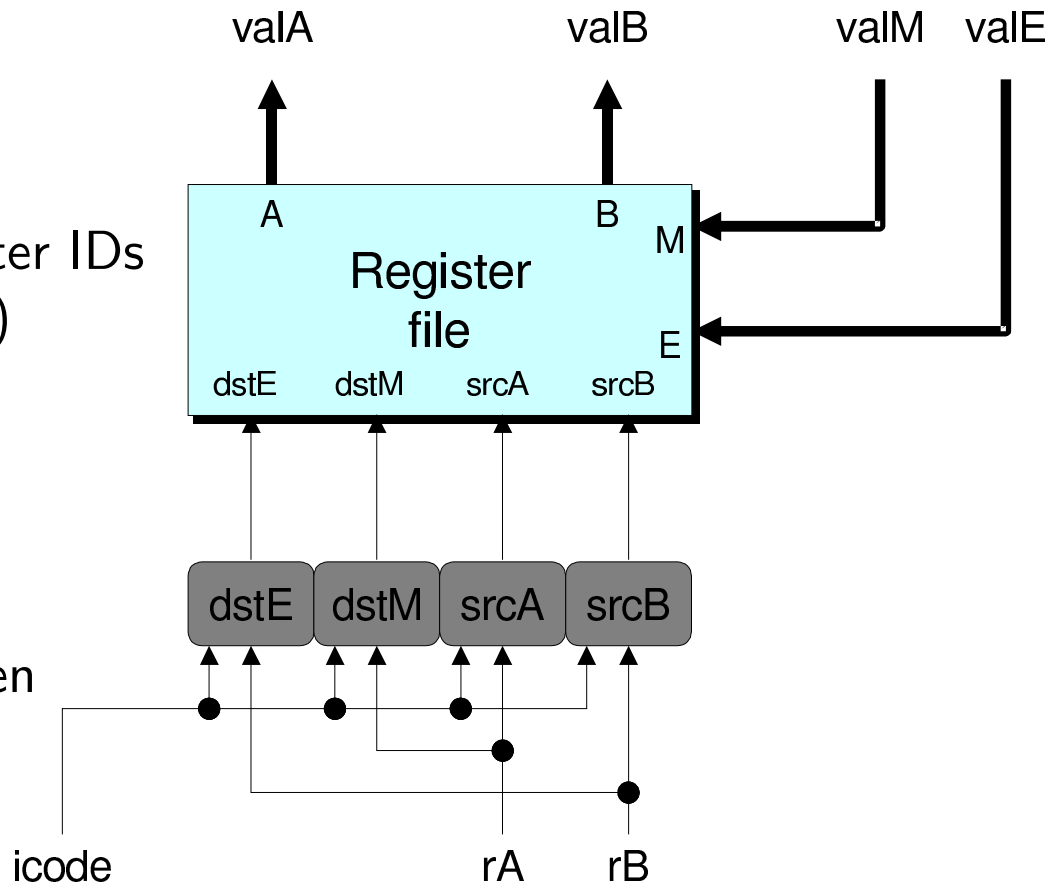
Kontrolllogik



- "Instr. Valid": Ist diese Anweisung gültig?
- "Need regids": Enthält diese Anweisung Registeradressen?
- "Need valC": Enthält diese Anweisung eine Konstante?

Dekodierungslogik

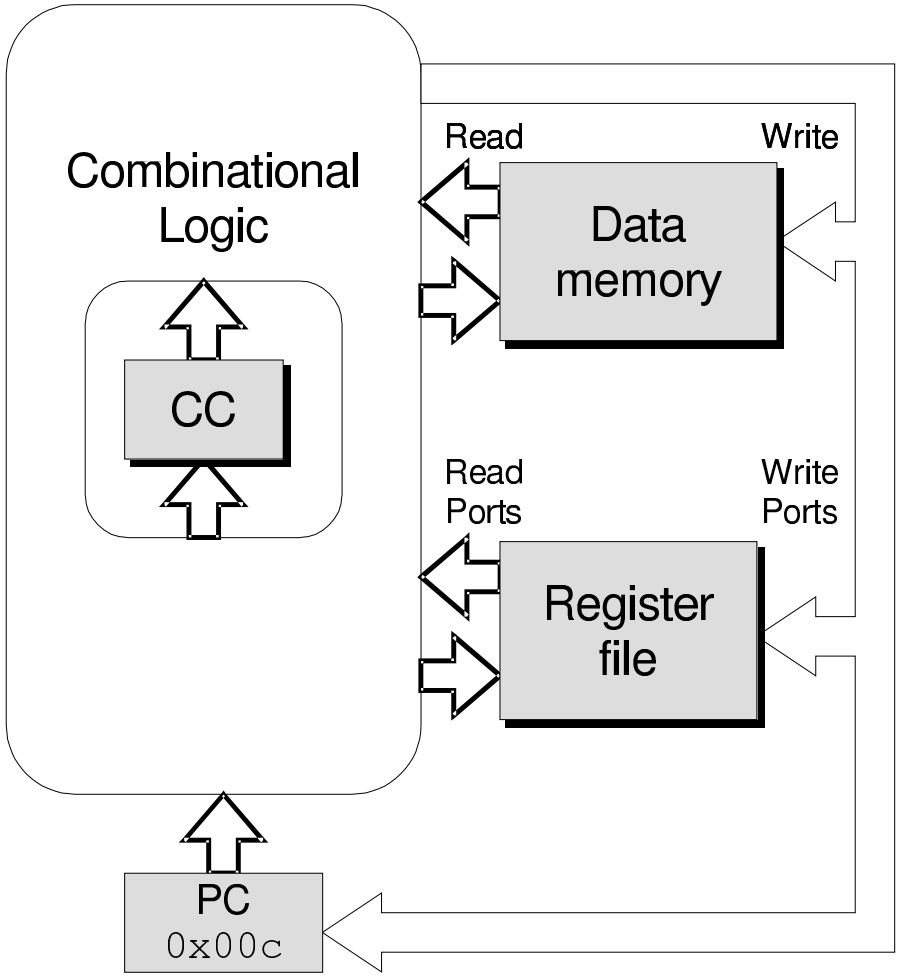
- Registerbank
  - Liest Ports A, B
  - Schreibt Ports E, M
  - Adressen sind Register IDs oder 8 (kein Zugriff)
- Kontrolllogik
  - srcA, srcB: Lese-Adressen
  - dstA, dstB: Schreib-Adressen



SEQ Operation

- Zustand
  - PC Register
  - Zustandscode Register
  - Datenspeicher
  - Registerbank

  alle vorderflankengesteuert
- Kombinationslogik
  - ALU
  - Kontrolllogik
  - Leseoperationen
    - Anweisungen
    - Registerbänke
    - Datenspeicher



SEQ Zusammenfassung

Implementierung

- jede Anweisung wird als Serie simpler Schritte formuliert
- gleichartiger Ablauf für jeden Anweisungstyp
- zusammengesetzt aus Registern, Speicher, vorher entworfene kombinatorische Blöcke
- verbunden mit Kontrolllogik

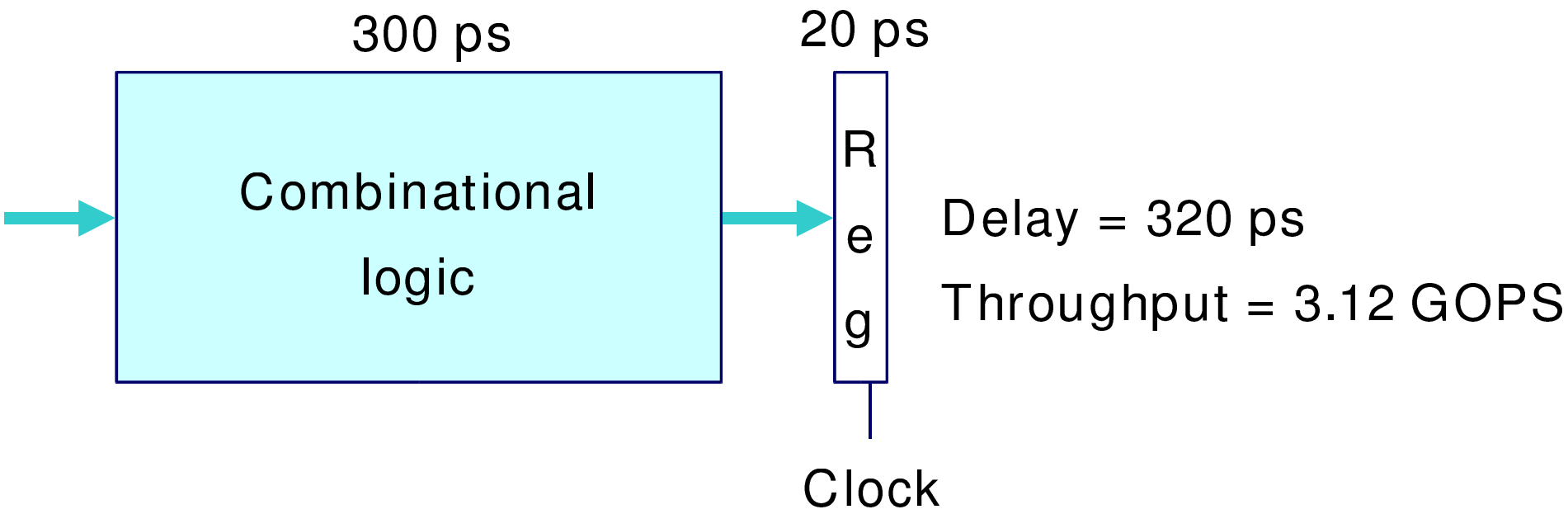
Limitierungen

- zu langsam für den praktischen Einsatz
- in einem Zyklus wird auf Anweisungsspeicher, Registerbank, ALU und Datenspeicher zugegriffen
- Takt müsste sehr langsam laufen
- Hardwareeinheiten sind nur für Bruchteile eines Taktzyklus aktiv

Pipelining

- lässt Vorgänge gleichzeitig ablaufen
- “Real-World Pipelines”: Autowaschanlagen
- Konzept
  - Prozess in unabhängige Abschnitte aufteilen
  - Objekt sequentiell durch diese Abschnitte laufen lassen
  - zu jedem gegebenen Zeitpunkt werden zahlreiche Objekte bearbeitet

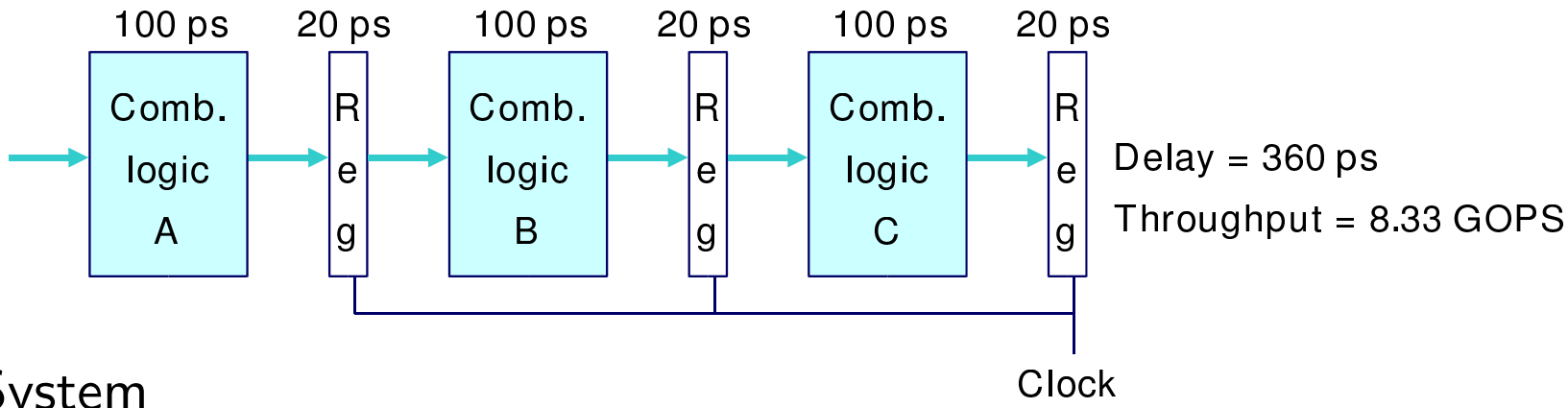
Berechnungsbeispiel



System

- Verarbeitung erfordert 300 Pikosekunden [ps]
- weitere 20 ps um das Resultat im Register zu speichern
- Zykluszeit: mindestens 320 ps

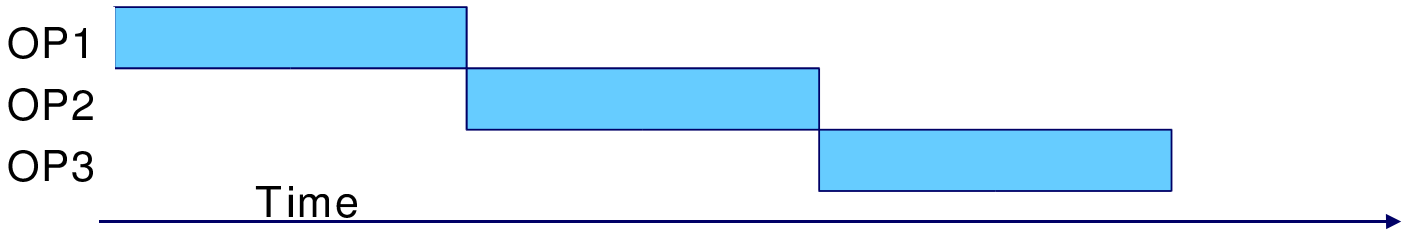
3-Wege "Pipelined" Version



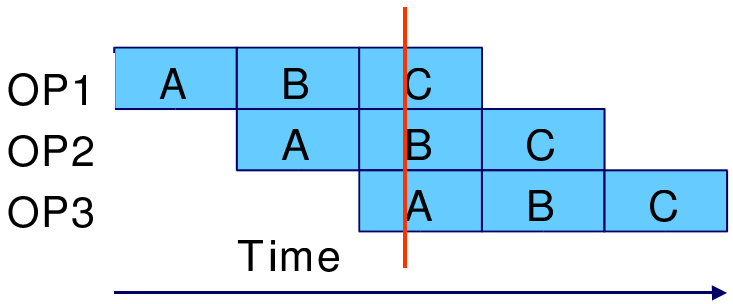
- teilt Kombinationslogik in 3 Blöcke zu je 100 ps auf
- neue Operation kann beginnen, sobald vorherige Abschnitt A durchgelaufen hat – beginnt alle 120 ps neue Operation
- allgemeine Latenzzunahme
  - 360 ps von Start bis Ende

Pipeline-Diagramme

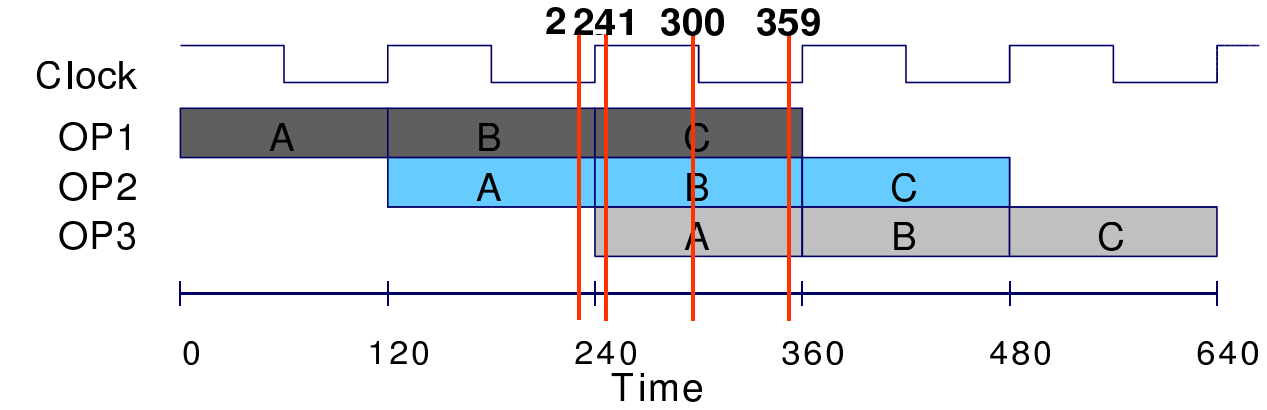
Unpipelined

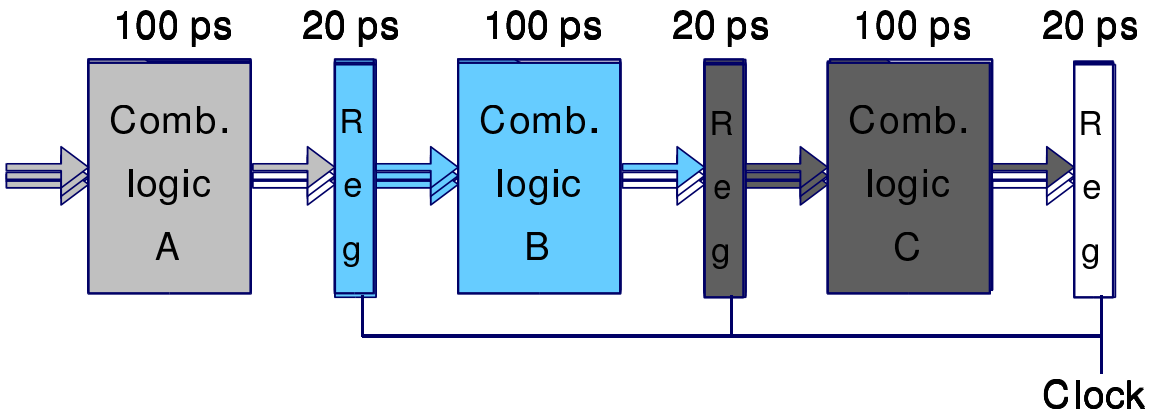


3-Way Pipelined

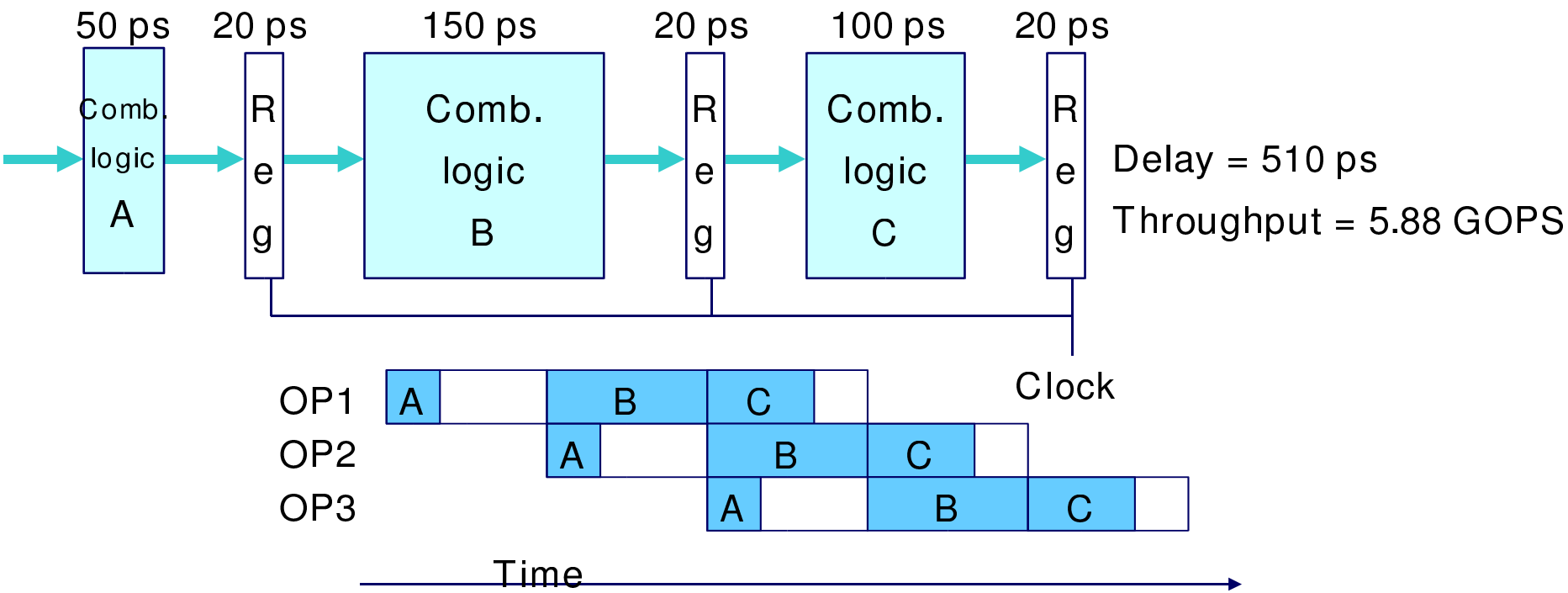


Funktionsweise einer Pipeline

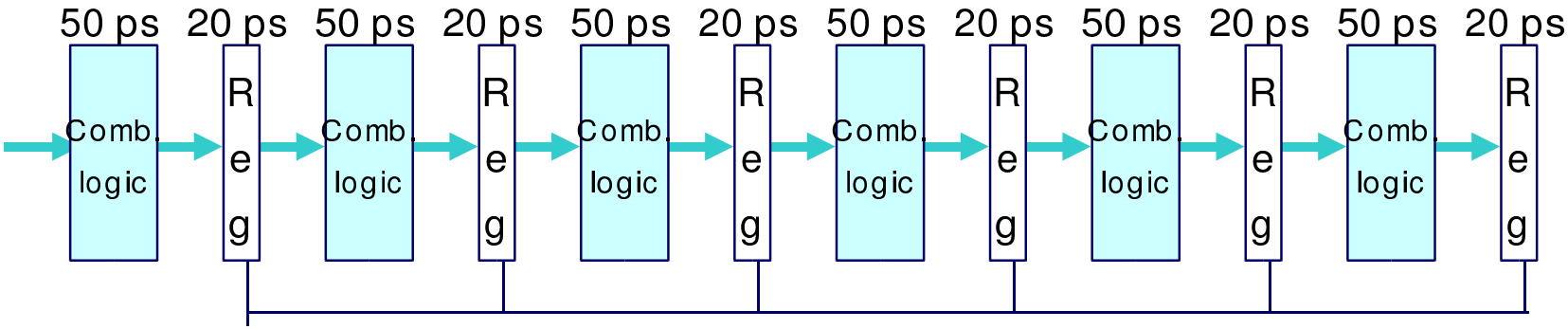




Limitierungen: Nichtuniforme Verzögerungen



Limitierungen: Register "Overhead"



- mit steigender Länge der Pipeline gewinnt Anteil des (registerbedingten) Overhead an Bedeutung
- Taktzeit, die für das Laden der Register verbraucht wird:
  - 1-Register "Pipeline": 6,25%
  - 3-Register "Pipeline": 16,67%
  - 6-Register "Pipeline": 28,57%
- Geschwindigkeitsgewinne moderner Prozessordesigns durch sehr große "Pipelinelängen"

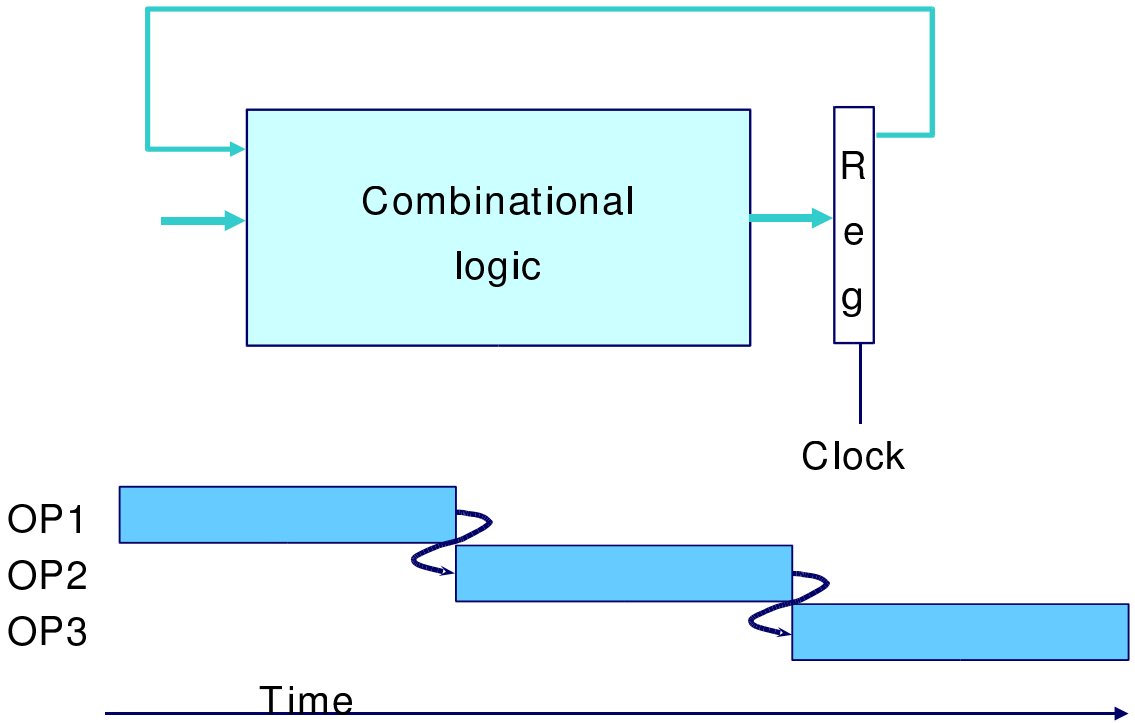
Ergänzende Literatur

Zur Rechnerarchitektur (Teil 2, 5. Termin):

[1] Randal E. Bryant and David O'Hallaron.
Computer systems.
pages 258–317. Pearson Education, Inc., New Jersey, 2003.

Gliederung

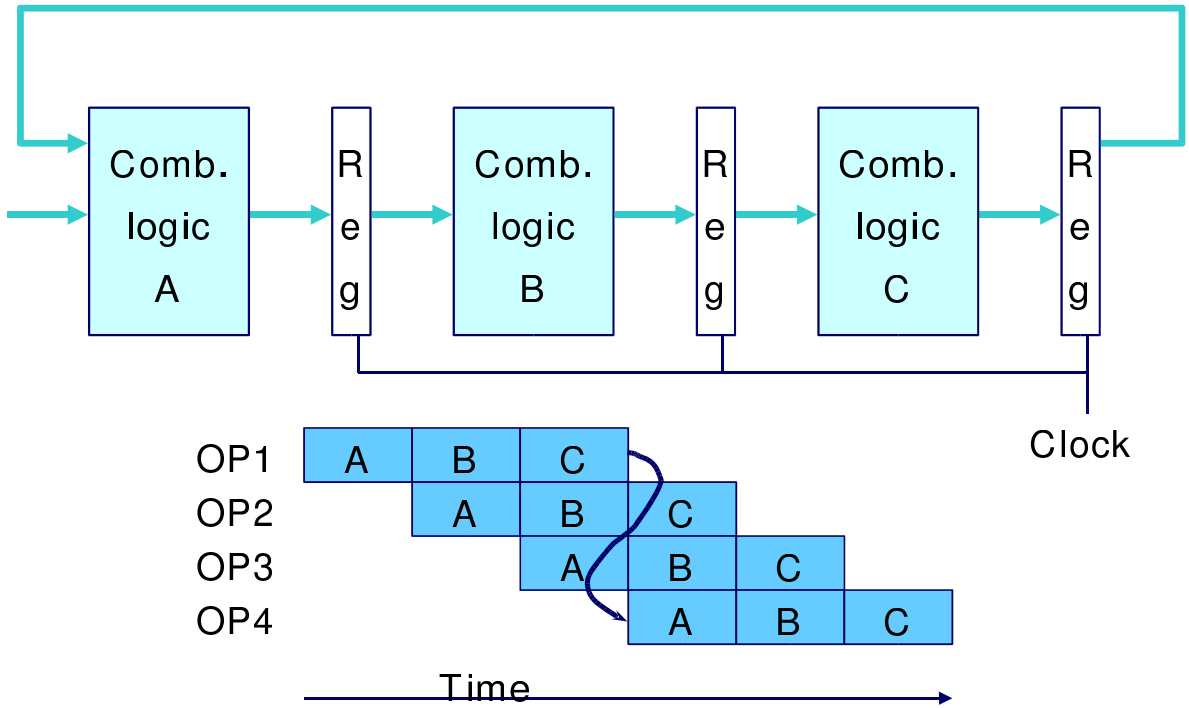
Datenabhängigkeiten



Systematik:

- Jede Operation hängt vom Erebnis der vorhergehenden ab

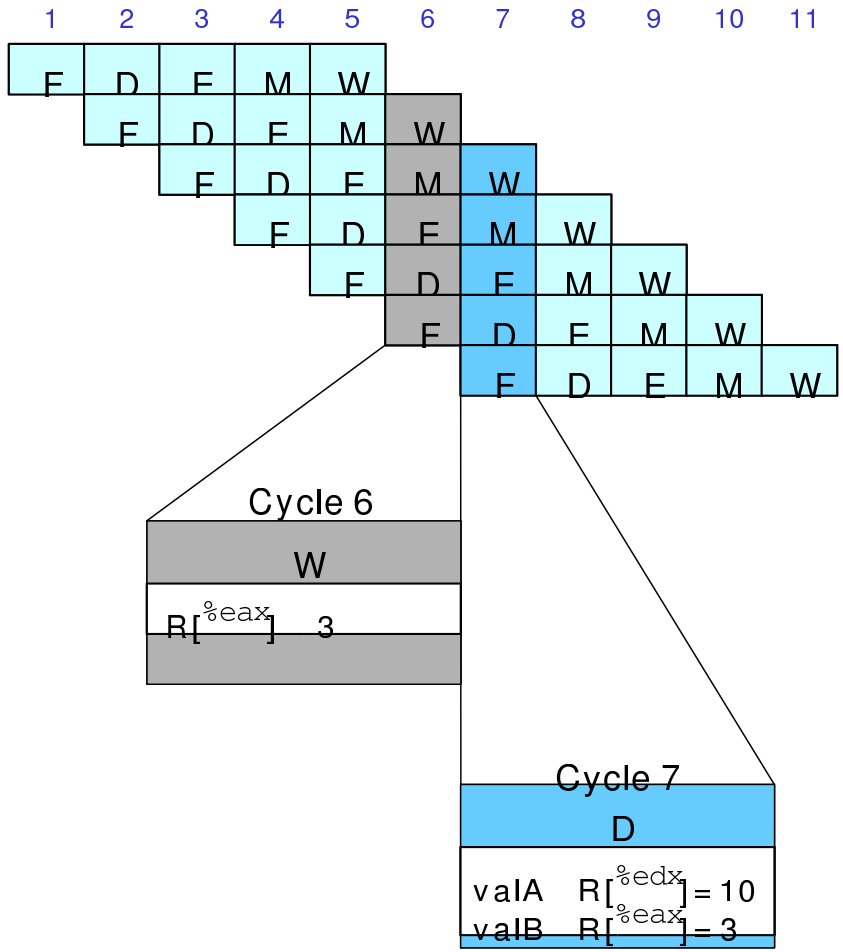
Daten Hazards



- Resultat-Feedback kommt zu spät für die nächste Operation
- Pipelining ändert Verhalten des gesamten Systems

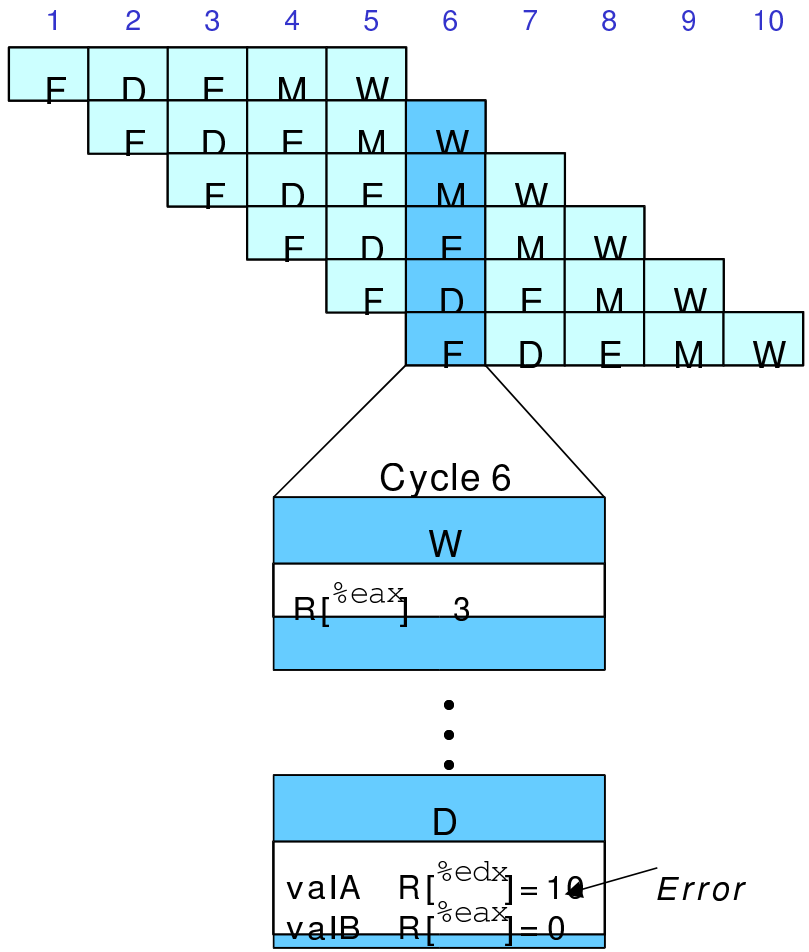
Datenabhängigkeiten: 3 Nops (No Operation)

```
# demo-h3.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
0x00e: nop
0x00f: addl %edx,%eax
0x011: halt
```



Datenabhängigkeiten: 2 Nops

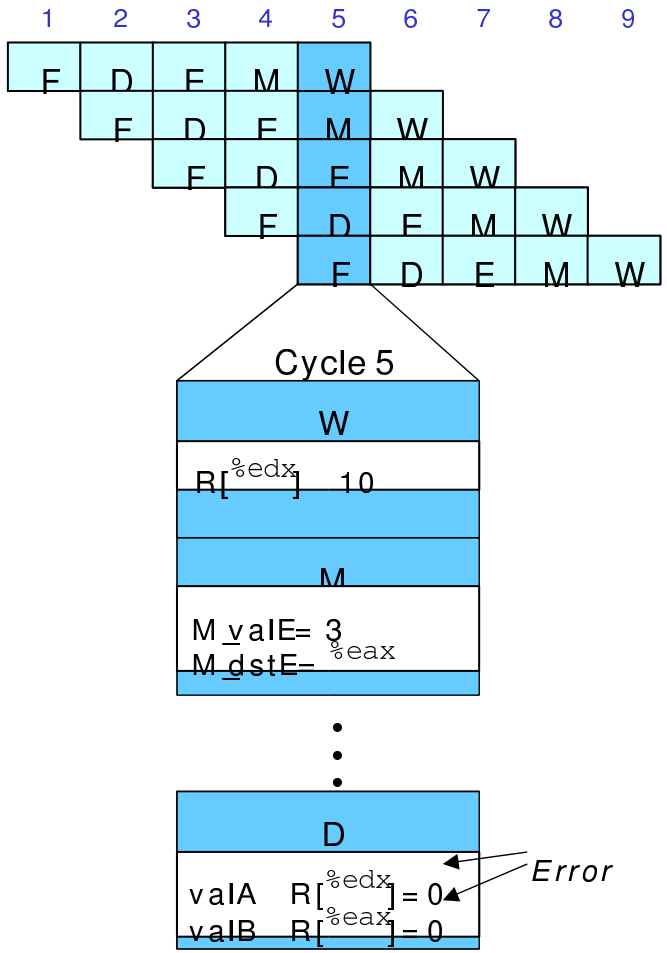
```
# demo-h2.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
0x00e: addl %edx,%eax
0x010: halt
```



Datenabhängigkeiten: 1 Nops

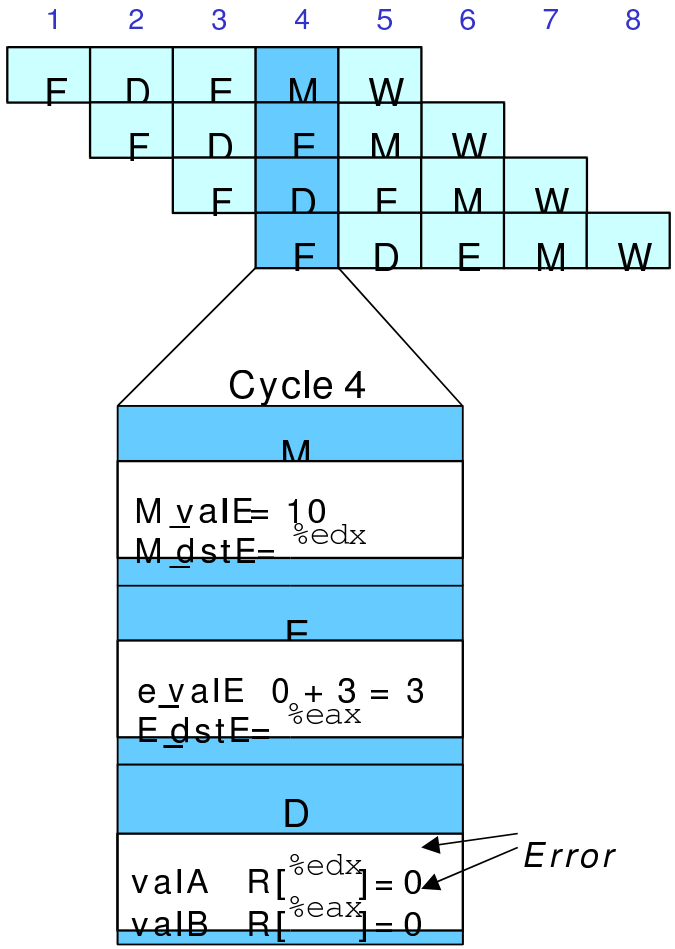
```
# demo-h1.ys

0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: addl %edx,%eax
0x00f: halt
```



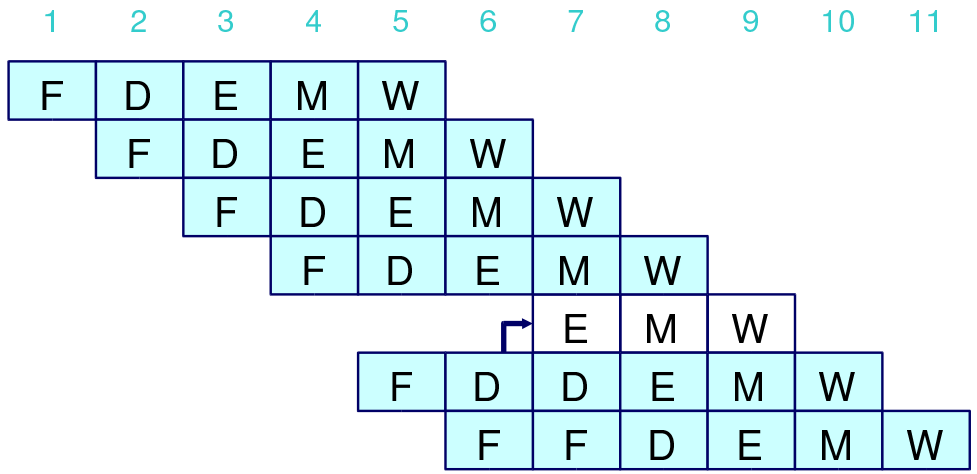
Datenabhängigkeiten: No Nop

```
# demo-h0.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: addl %edx,%eax
0x00e: halt
```



“Stalling” bei Datenabhängigkeiten

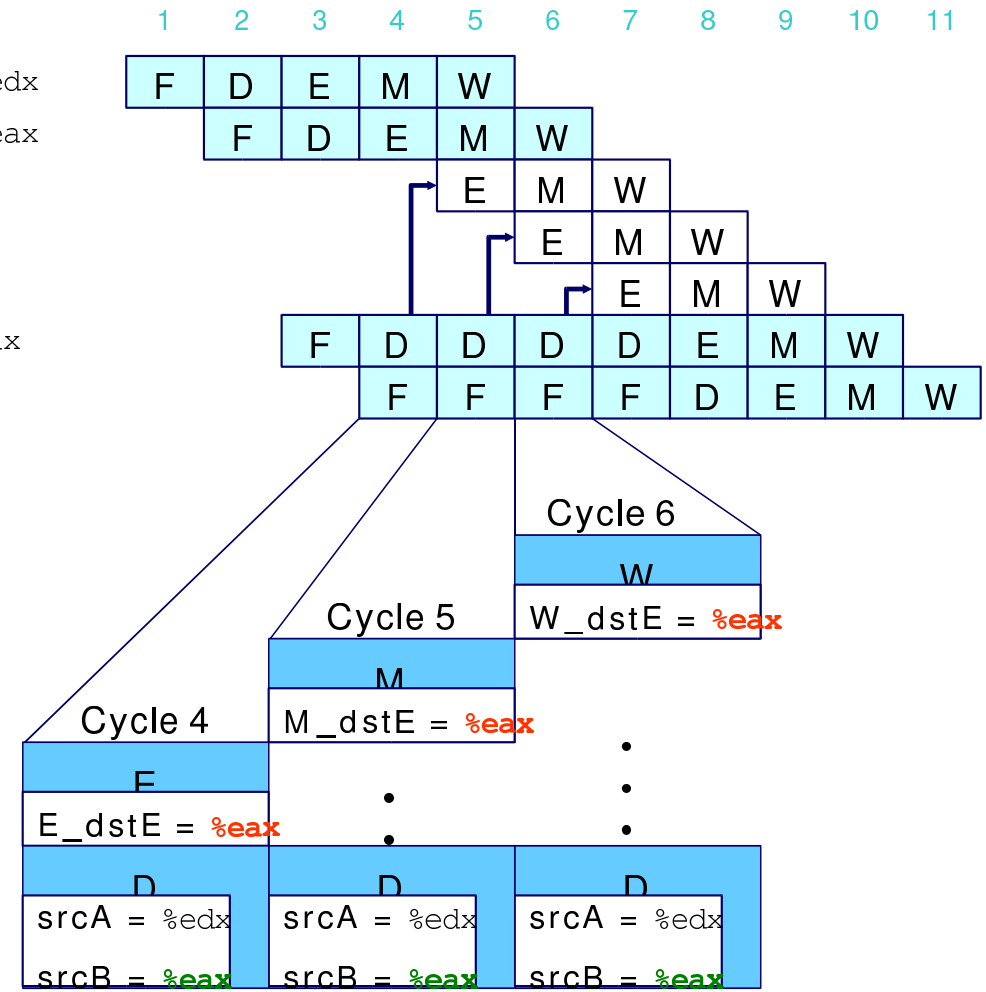
```
# demo-h2.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
       bubble
0x00e: addl %edx,%eax
0x010: halt
```



- falls die Anweisung zu dicht auf eine andere folgt, die in das von der aktuellen Anweisung benutzte Register schreibt, dann verzögere Anweisung
- halte Anweisung im Decode-Status
- füge nop dynamisch in “Execute Stage” ein

Stalling x3

```
# demo-h0.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
       bubble
       bubble
       bubble
0x00c: addl %edx,%eax
0x00e: halt
```



Was passiert beim Stalling?

```
# demo-h0.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: addl %edx,%eax
0x00e: halt
```

Cycle 8

Write Back	***bubble***
Memory	***bubble***
Execute	`0x00c: addl %edx,%eax`
Decode	`0x00e: halt`
Fetch	

- verzögerte Instruktion wird im “Decode”-Status gehalten
- folgende Anweisung verbleibt im “Fetch” Status (Befehl-holen-Status)
- Bubbles werden in “Execute Stage” eingefügt
  - wie dynamisch generierte nops
  - bewegen sich durch spätere Stufen (stages)

“Data Forwarding”

Naive Pipeline

- Register werden erst nach Vollendung der “Write-back” Phase (stage) zurückgeschrieben
- Source-Operanden werden aus der Registerbank während der Dekodierungsphase gelesen
  - müssen zu Beginn der Phase in der Registerbank sein

Beobachtung

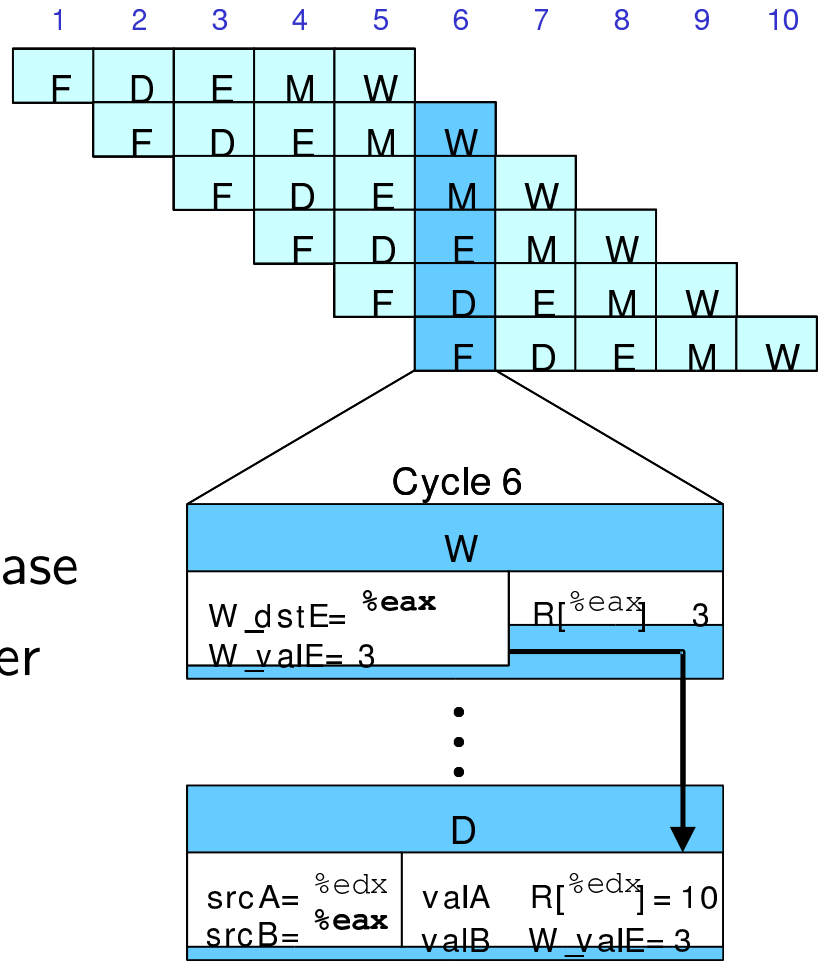
- Ergebnis wird während der “Execute Stage” oder der “Memory Stage” generiert

Trick (“Data Forwarding”)

- Resultate direkt in die Dekodierstufe leiten
- muss erst zum Ende der Dekodierphase verfügbar sein

Beispiel für "Data Forwarding"

```
# demo-h2.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
0x00e: addl %edx,%eax
0x010: halt
```



- irmovl in der "Write-back" Phase
- Zielwert im W Pipeline Register
- Weiterleiten als valB in die Dekodierungsphase

Sprungvorhersage

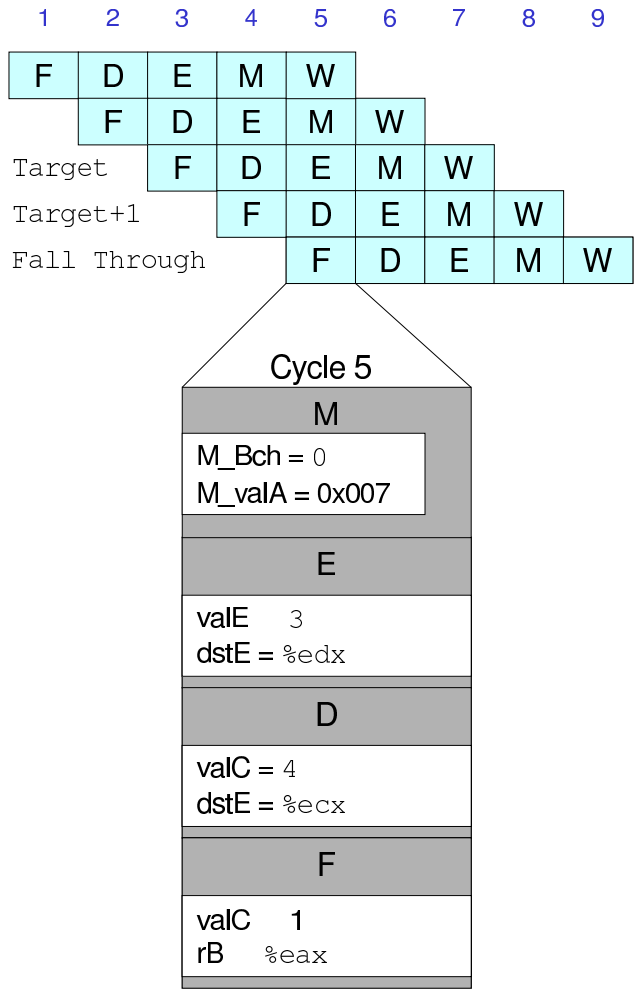
Beispiel für falsche Sprungvorhersage

```
0x000:      xorl %eax,%eax
0x002:      jne  t                  # Not taken
0x007:      irmovl $1, %eax         # Fall through
0x00d:      nop
0x00e:      nop
0x00f:      nop
0x010:      halt
0x011: t:   irmovl $3, %edx         # Target (Should not execute)
0x017:      irmovl $4, %ecx         # Should not execute
0x01d:      irmovl $5, %edx         # Should not execute
```

- sollte nur die ersten 8 Anweisungen ausführen

Trace einer falschen Sprungvorhersage

```
# demo-j
0x000:     xorl %eax,%eax
0x002:     jne t # Not taken
0x011: t:  irmovl $3, %edx # Target
0x017:     irmovl $4, %ecx # Target+1
0x007:     irmovl $1, %eax # Fall Through
```

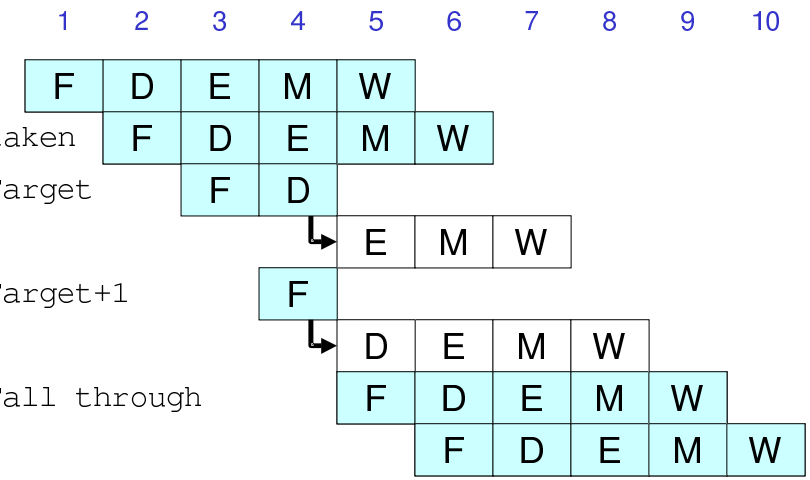


- fälschliche Ausführung zweier Anweisungen am Sprungziel

Behandlung einer falschen Sprungvorhersage (Branch Misprediction)

- Sprung wird vorhergesagt
  - holt die nächsten 2 Anweisungen am Sprungziel

```
# demo-j.ys
0x000:     xorl %eax,%eax
0x002:     jne target # Not taken
0x011: t:  irmovl $2,%edx # Target
           bubble
0x017:     irmovl $3,%ebx # Target+1
           bubble
0x007:     irmovl $1,%eax # Fall through
0x00d:     nop
```



- Abbruch bei falscher Vorhersage
  - erkenne nicht ausgeführten Sprung in der “Execute”-Phase
  - ersetze im folgenden Zyklus die Anweisung der “Execute”-Phase
  - fülle Dekodierungsphase mit Bubbles auf
  - es sind noch keine Seiteneffekte aufgetreten

Pipeline Zusammenfassung

Data Hazards

- Meist durch Forwarding umgangen
  - Kein Performance Verlust
- Load/Use Hazards erfordern “Stalling” und “Forwarding”
  - Verzögerung um einen Takt

Control Hazards

- bei falscher Sprungvorhersage breche Anweisungen ab
  - zwei Taktzyklen verschwendet
- bei “ret”-Anweisung verzögere “Fetch Stage” während “ret” durch die Pipeline wandert
  - drei Taktzyklen verschwendet

Pipeline Zusammenfassung II

Control Kombinationen

- Müssen sorgfältig analysiert werden
- schwer zu handhaben für Pipeline-Kontroll-Mechanismen
- Beispiele:
  - “not taken jump” gefolgt von “ret”-Anweisung
  - “load/use Hazard” mit Beteiligung von %ESP gefolgt von “ret”-Anweisung

Ausnahmebehandlung ("Exception Handling")

- "Pipeline" kann normalen Ablauf nicht fortsetzen
- Ursachen
  - Halt Anweisung
  - ungültige Adresse für Anweisung oder Daten
  - ungültige Anweisung
  - "Pipeline" Kontrollfehler
- erforderliches Vorgehen
  - einige Anweisungen vollenden
    - Entweder aktuelle oder vorherige (hängt von Ausnahmetyp ab)
  - andere verwerfen
  - "Exception handler" aufrufen
    - wie unerwarteter Prozeduraufruf

PentiumPro Operationen & Befehlspipeline

- übersetzt Anweisungen dynamisch in "μOPs"
  - 118 Bit breite interne Befehsworte
  - enthält Operation, zwei Quellen und Ziel
- führt μOPs mit "Out of Order" Maschine aus (superskalare Archtektur)
  - μOP ausgeführt wenn
    - Operanden verfügbar sind
    - Funktionelle Einheit verfügbar ist
  - Ausführung wird durch "Reservation Stations" kontrolliert
    - beobachtet die Datenabhängigkeiten zwischen μOPs
    - teilt Ressourcen zu

Pentium 4 Eigenschaften

- "Trace" Cache
  - ersetzt traditionellen Anweisungscache
  - nimmt Anweisungen in dekodierter Form in Cache auf
  - reduziert benötigte Rate für den Anweisungsdecoder
- "Double pumped" ALUs (2 Operationen pro Taktzyklus)
  - simple Anweisungen (hinzufügen)
- sehr große "Pipeline"
  - 20+Zyklus "Branch Penalty"
  - ermöglicht sehr hohe Taktfrequenzen
  - langsamer als Pentium III mit gleicher Taktfrequenz

Prozessor Zusammenfassung

- Prozessordesign
  - einheitlichens “Frameworks” für alle Anweisungen
    - Anweisungen können Hardwareressourcen untereinander zu teilen
  - Kontrolllogik steuert Hardwareressourcen
- Operation
  - Zustand wird in Speichern und Registern gehalten
  - Berechnung wird von kombinatorischer Logik durchgeführt
  - Taktung der Register/Speicher kontrolliert das allgemeine Verhalten
- Leistungssteigerung
  - “Pipelining” erhöht den Datendurchsatz und verbessert die Ressourcennutzung
  - ISA bleibt unverändert!

Gliederung

Gliederung (cont.)

Die Speicherhierarchie

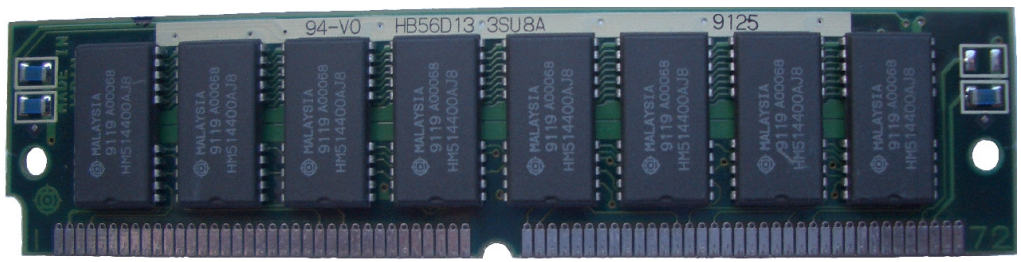
Themen:

- Speichertechnologien und Trends
- Lokalität der Referenzen
- “Caching” in der Speicherhierarchie

Random-Access Memory (RAM)

Hauptmerkmale

- RAM ist als Chip gepackt
- Grundspeichereinheit ist eine Zelle (ein Bit pro Zelle)
- Viele RAM Chips bilden einen Speicher



Random-Access Memory (RAM)

Statischer RAM (SRAM)

- Jede Zelle speichert Bits mit einer 6-Transistor Schaltung
- Speichert Wert solange er mit Energie versorgt wird
- Relativ unanfällig für Störungen wie elektrische Brummspannungen
- Schneller und teurer als DRAM

Dynamischer RAM (DRAM)

- Jede Zelle speichert Bits mit einem Kondensator und einem Transistor
- Der Wert muss alle 10-100 ms aufgefrischt werden
- Anfällig für Störungen
- Langsamer und billiger als SRAM

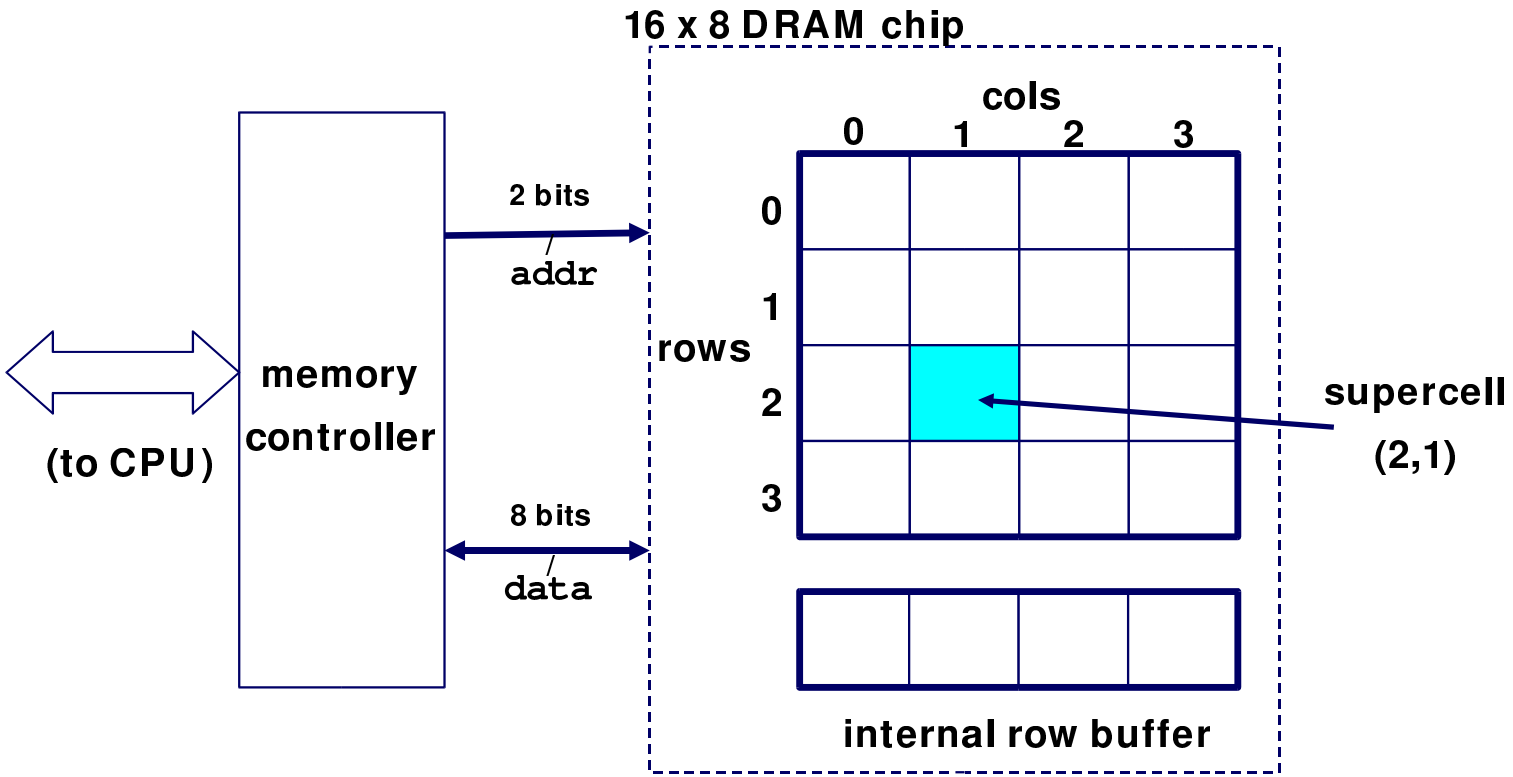
SRAM gegen DRAM Zusammenfassung

	Tran. per bit	Access time	Persist?	Sensitive?	Cost	Applications
SRAM	6	1X	Yes	No	100x	cache memories
DRAM	1	10X	No	Yes	1X	Main memories, frame buffers

Konventionelle DRAM Organisation

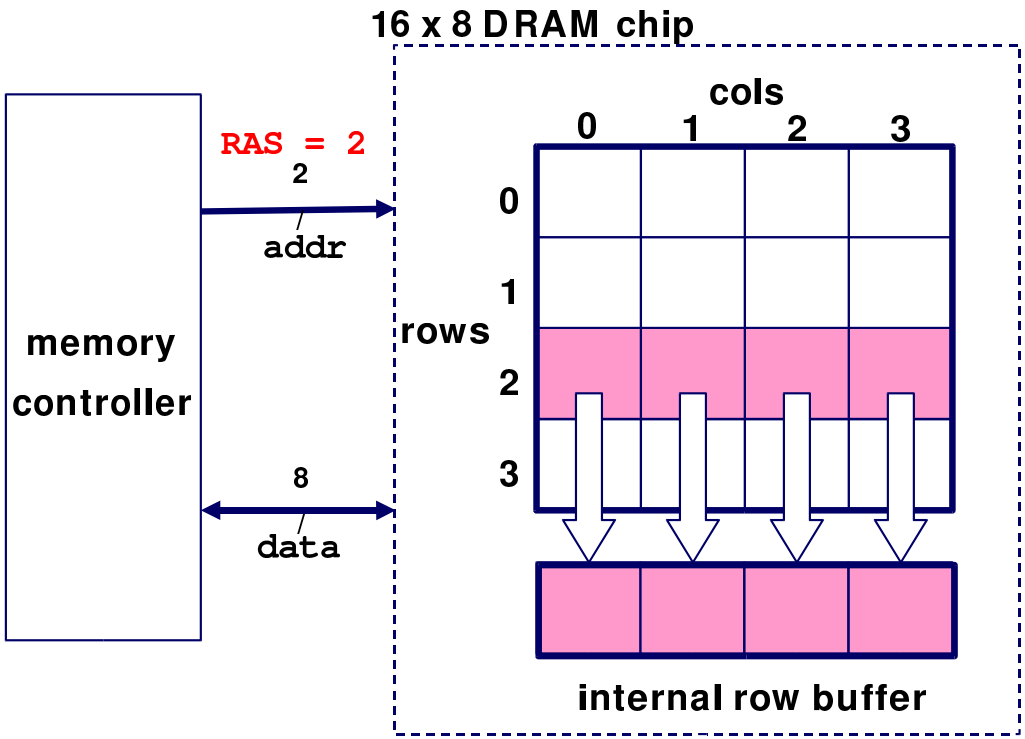
(d x w) DRAM:

- dw Summe aller Bits organisiert als d Superzellen mit w Bits



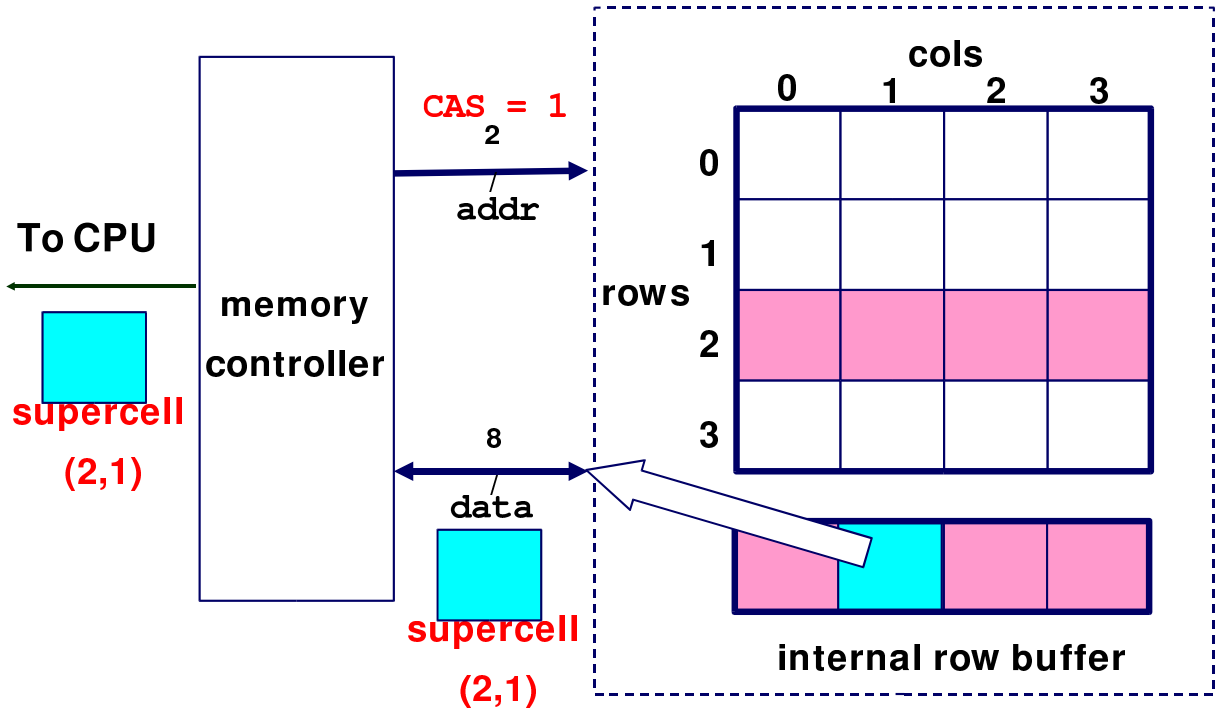
Lesen der DRAM Superzelle (2,1), Schritt 1

- Schritt 1(a): "Row Access Strobe" (RAS) wählt Zeile 2.
- Schritt 1(b): Zeile 2 wird aus dem DRAM Array in den Zeilenpuffer ("Row Buffer") kopiert.



Lesen der DRAM Superzelle (2,1), Schritt 2

- Schritt 2(a): "Column Access Strobe" (CAS) wählt Spalte 1.
- Schritt 2(b): Superzelle (2,1) wird aus dem Buffer ausgelesen und auf die Datenleitungen gelegt



Erweiterte DRAMS

Alle erweiterten DRAMS sind um den konventionellen DRAM Kern herum aufgebaut:

- "Fast Page Mode DRAM" (FPM DRAM)
  - greift auf den Inhalt einer Speicherzeile zu mit z. B.:
    - RAS, CAS, CAS, CAS, CAS
    - sonst:(RAS, CAS), (RAS, CAS), (RAS, CAS), (RAS, CAS)
- "Extended Data Out DRAM" (EDO DRAM)
  - erweiterter FPM DRAM mit schneller aufeinanderfolgenden CAS Signalen
- synchroner (taktgesteuerter) DRAM (SDRAM)
  - betrieben mit steigender Taktflanke statt mit asynchronen Kontrollsignalen

Erweiterte DRAMS (cont.)

- "Double Data-Rate Synchronous DRAM" (DDR SDRAM)
  - SDRAM-Erweiterung die beide Taktflanken als Kontrollsignal benutzt
- Video RAM (VRAM)
  - wie FPM DRAM, aber Output wird durch Verschieben des "Row Buffer" hergestellt
  - zwei Ports (erlauben gleichzeitiges Lesen und Schreiben)

Nichtflüchtige Speicher

- DRAM und SRAM sind flüchtige Speicher.
  - Informationen gehen beim Abschalten verloren
- nichtflüchtige Speicher speichern Werte selbst wenn sie spannungslos sind
  - allgemeiner Name ist “Read-Only-Memory” (ROM)
  - irreführend, da einige ROMs auch verändert werden können
- Arten von ROMs
  - Programmmierbarer ROM (PROM)
  - “Erasable Programmable ROM” (EPROM) (Löschbarer, programmierbarer ROM)
  - “Electrically Eraseable PROM” (EEPROM) (Elektrisch löschbarer PROM)
  - Flash Speicher

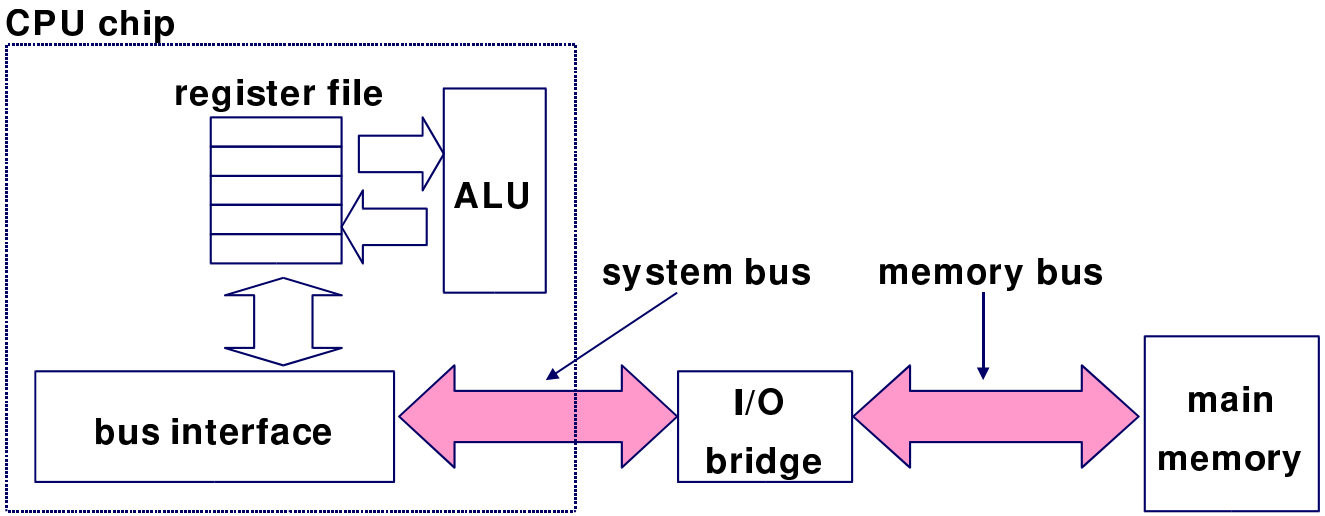
Anwendungsbeispiel nichtflüchtige Speicher

Firmware

- Programm wird in einem ROM gespeichert
  - Boot Zeitcode, BIOS (basic input/output system)
  - Grafikkarten, Festplattencontroller

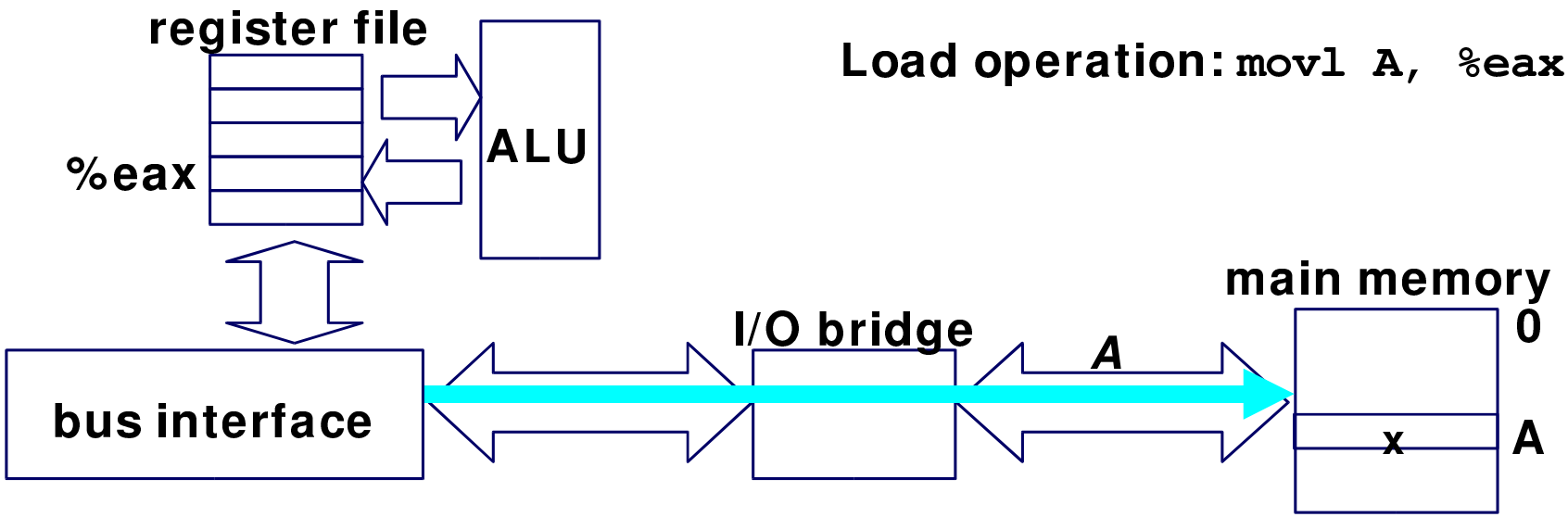
Typische Busstrukturen zur Verbindung von CPU und Speicher

- Busse (allg.):
  - Bündel paralleler Leitungen
  - es gibt mehr als einen Treiber ($\rightarrow$ Tristate-Treiber)
- Busse im Rechner:
  - zur Übertragung von Adressen, Daten und Kontrollsignalen
  - werden üblicherweise von mehreren Geräten genutzt



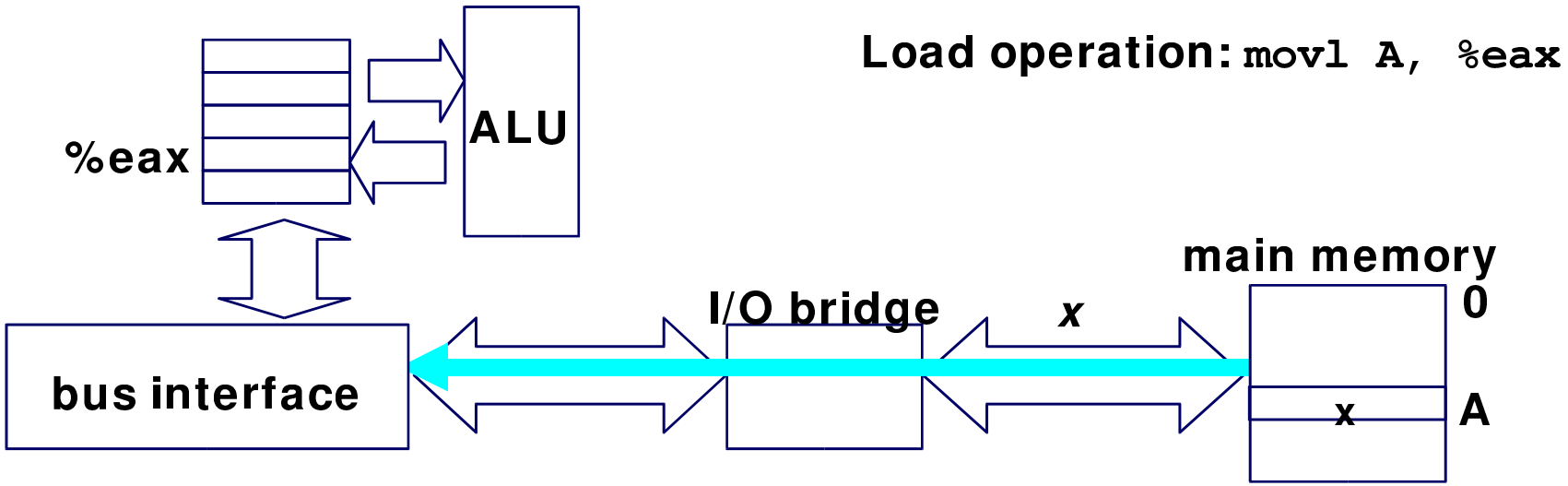
Lesender Speicherzugriff (1)

- CPU legt Adresse A auf den Speicherbus



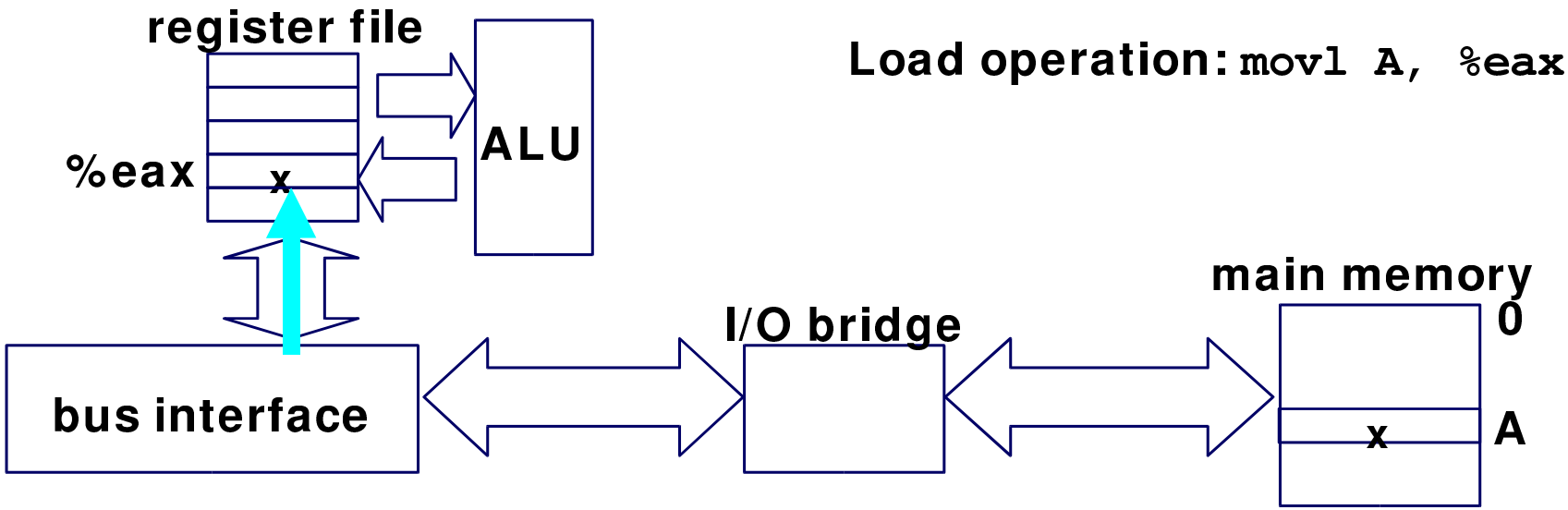
Lesender Speicherzugriff (2)

- Hauptspeicher liest Adresse A vom Speicherbus
- ruft das Wort x unter der Adressa A ab
- legt das Wort x auf den Bus



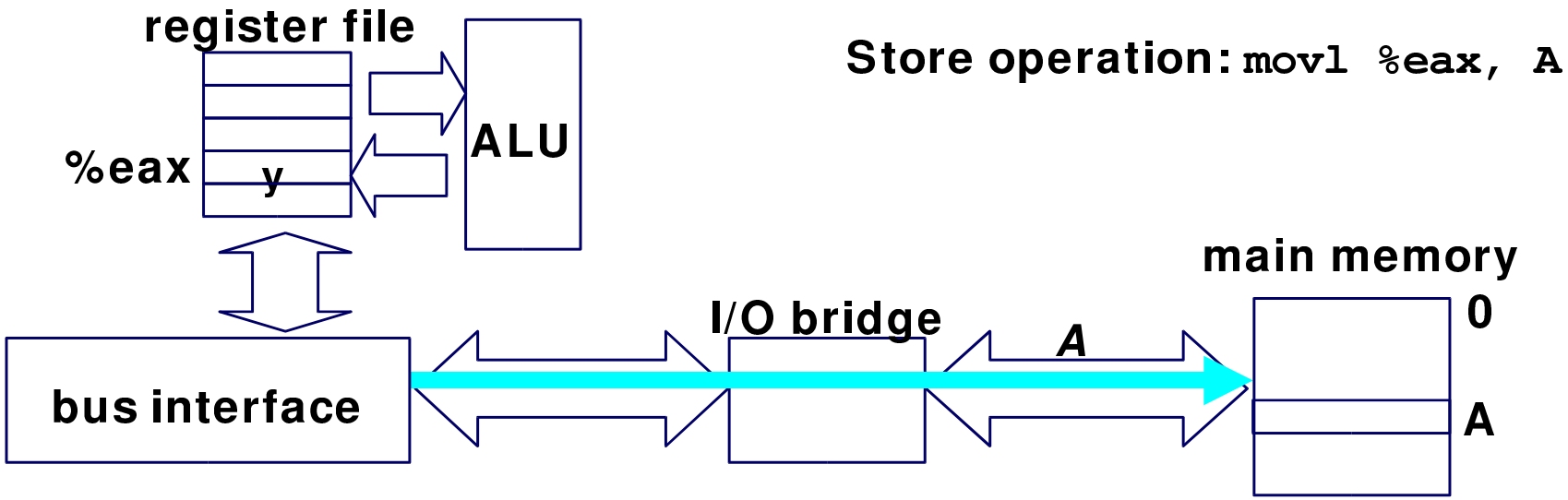
Lesender Speicherzugriff (3)

- CPU liest Wort x vom Bus
- CPU kopiert Wert x ins Register %eax



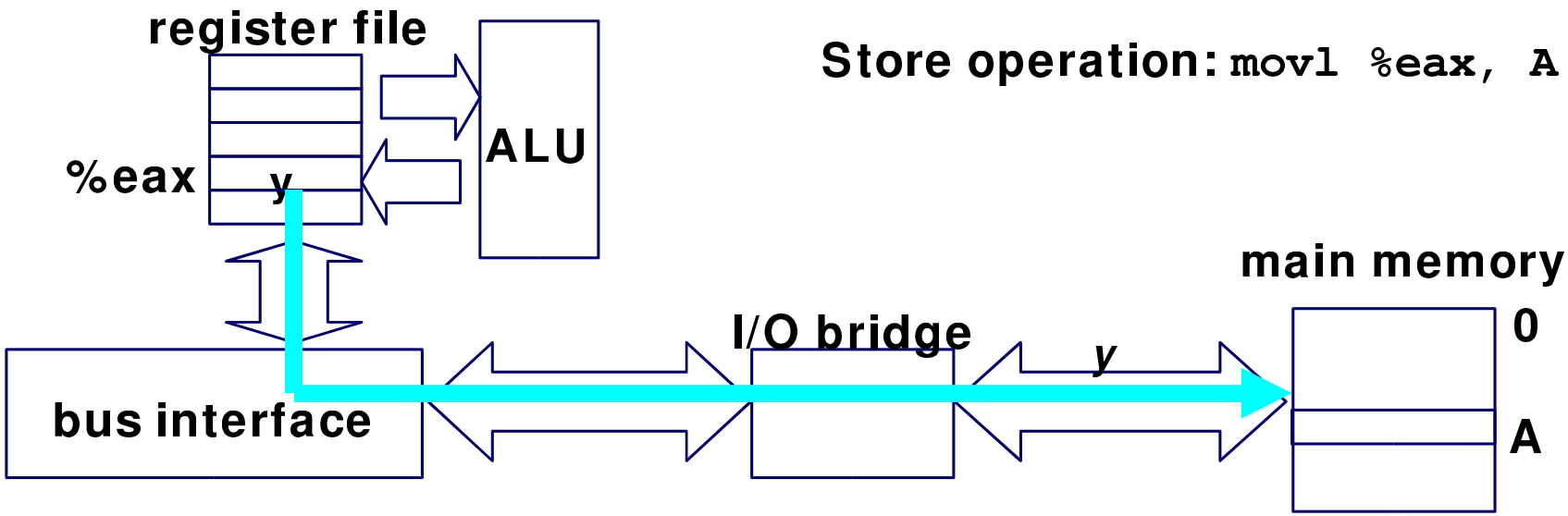
schreibender Speicherzugriff (1)

- CPU legt die Adresse A auf den Bus
- Hauptspeicher liest Adresse
- Hauptspeicher wartet auf Ankunft des Datenwortes



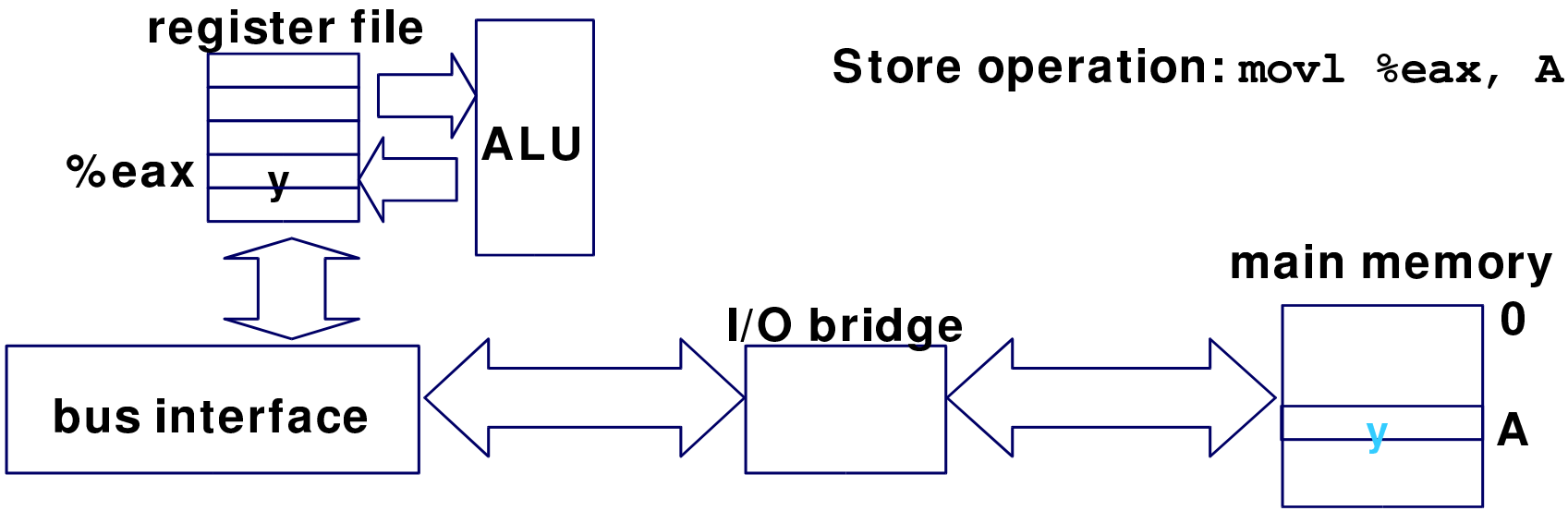
schreibender Speicherzugriff (2)

- CPU legt Datenwort y auf den Bus



schreibender Speicherzugriff (3)

- Hauptspeicher liest Datenwort y vom Bus
- Hauptspeicher speichert Datenwort y unter Adresse A ab



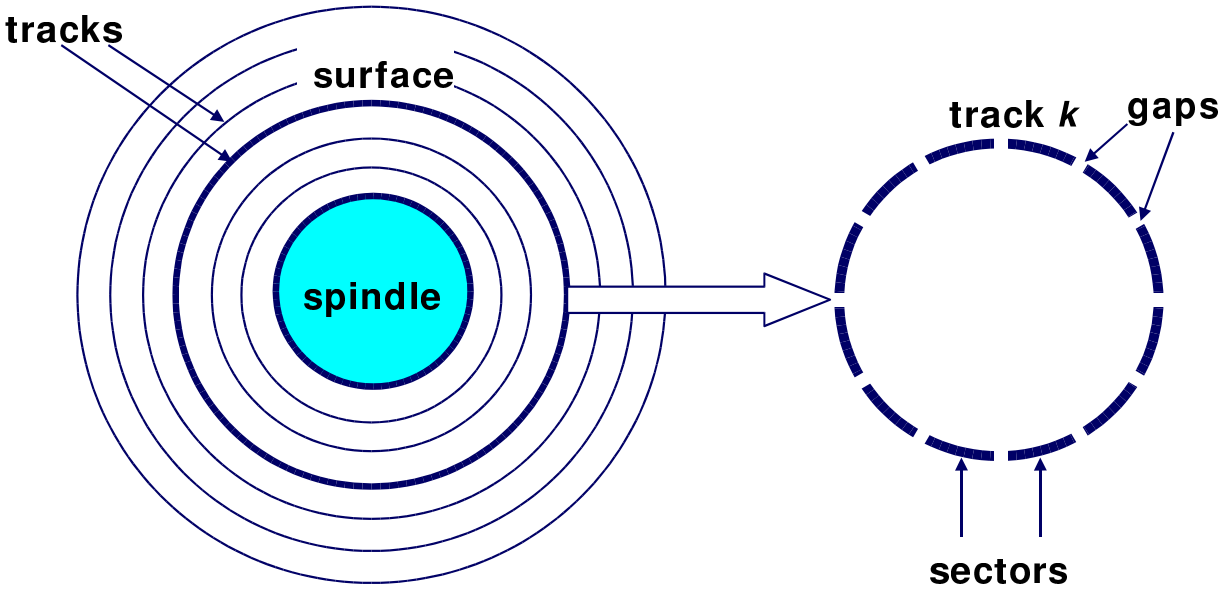
Ergänzende Literatur

Zur Rechnerarchitektur (Teil 2, 6. Termin):

[1] Randal E. Bryant and David O'Hallaron.
Computer systems.
pages 317–360, 455–466. Pearson Education, Inc., New Jersey, 2003.

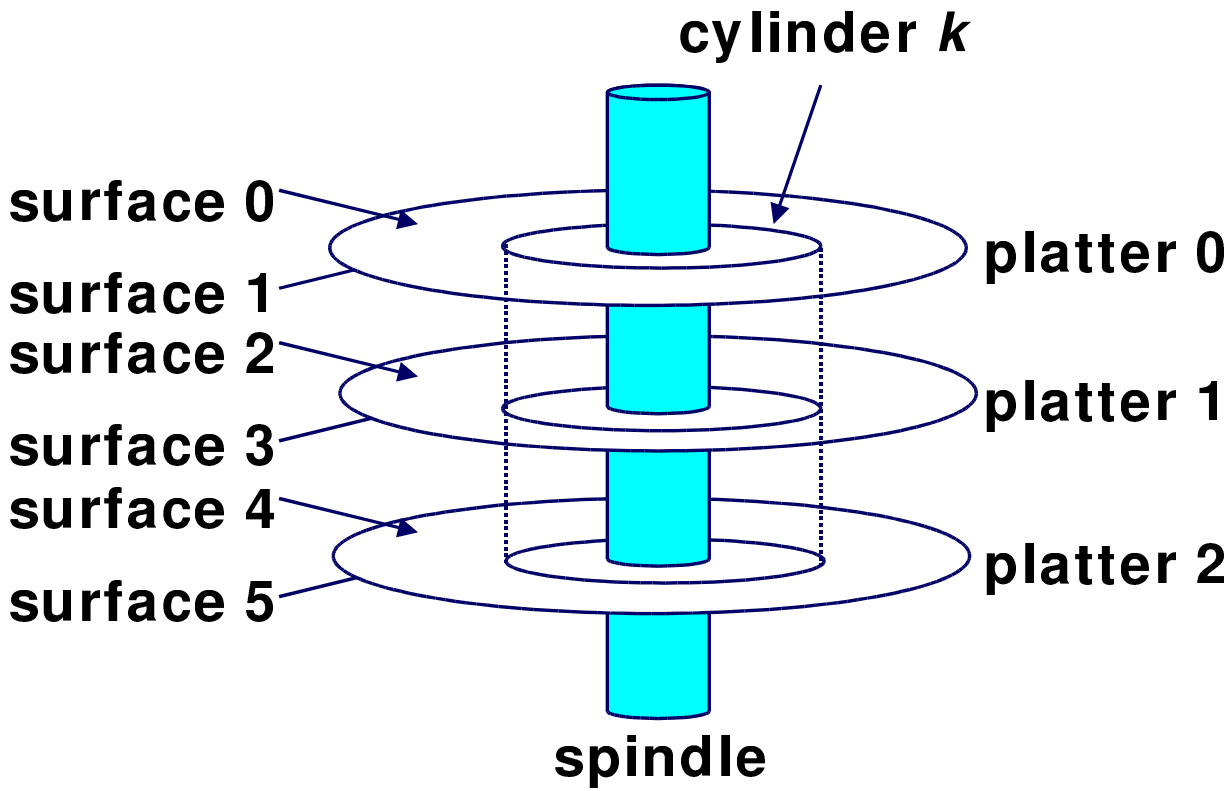
Festplattengeometrie

- Platten mit jeweils zwei Oberflächen (“surfaces”).
- Oberfläche trägt konzentrischen Ringen, die Spuren (“tracks”)
- Spur besteht aus Sektoren (“sectors”)
  - durch Lücken (“gaps”) getrennt



Festplattengeometrie (Ansicht mehrerer Platten)

- Untereinander liegende Spuren bilden einen Zylinder



Festplattenkapazität

- Kapazität: Höchstzahl speicherbarer Bits
  - Angabe der Kapazität in Gigabyte [GB] (hier: $1\,GB = 10^9\,Byte$)
- Kapazität wird von folgenden technologischen Faktoren bestimmt:
  - Aufnahmedichte [bits/in]:
    Anzahl der Bits / 1-Inch Segment einer Spur
  - Spurdichte [tracks/in]:
    Anzahl der Spuren / radiales 1-Inch Segment
  - Flächendichte [bits/in^2]:
    Resultat aus Aufnahme- und Spurdichte
- Spuren in getrennte Aufnahmezonen ("Recording Zones") unterteilt
  - jede Spur einer Zone hat gleichviele Sektoren (festgelegt durch die Ausdehnung der innersten Spur)
  - jede Zone hat unterschiedlich viele Sektoren/Spuren

Berechnen der Festplattenkapazität

Kapazität = (#Bytes/Sektor) x (Durchschn. # Sektoren/Spur)x (Spuren/Oberfläche) x (# Oberflächen/Platten) x (# Platten/Festplatte)

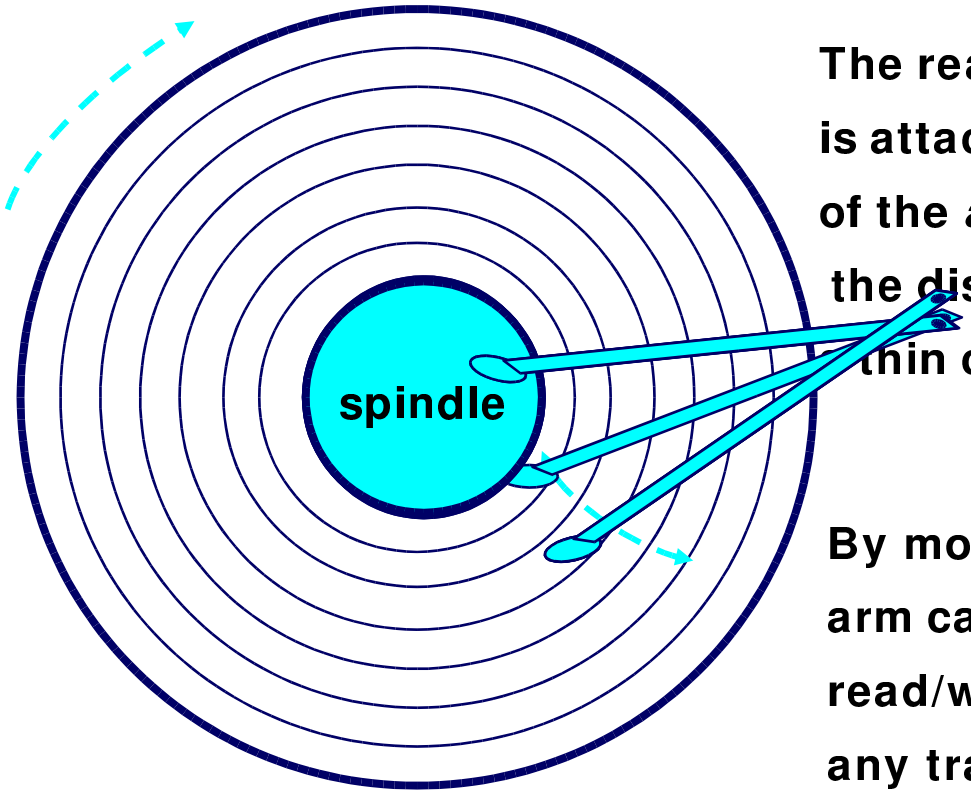
Beispiel:

- 512 Bytes/Sektor
- 300 Sektoren/Spuren (im Durchschnitt)
- 20.000 Spuren/Oberfläche
- 2 Oberflächen/Platten
- 5 Platten/Festplatte

Kapazität = 512 x 300 x 20000 x 2 x 5
= 30.720.000.000
= 30.72 GB

Festplatten-Operation (Ansicht einer Platte)

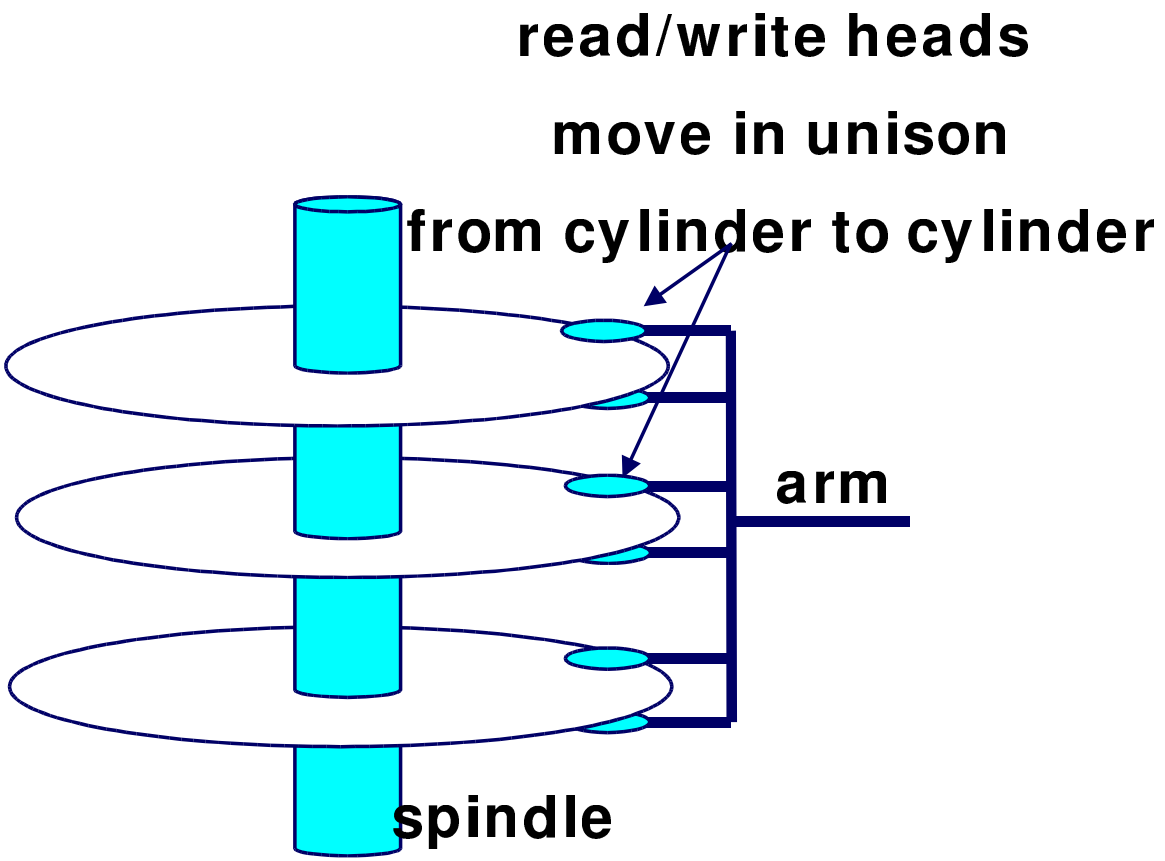
The disk surface spins at a fixed rotational rate



The read/write *head* is attached to the end of the *arm* and flies over the disk surface on a thin cushion of air.

By moving radially, the arm can position the read/write head over any track.

Festplatten-Operation (Ansicht mehrerer Platten)



Festplatten-Zugriffszeit

Durchschnittliche (avg) Zugriffszeit auf einen Zielsektor wird angenähert durch:

- $T_{Zugriff} = T_{avgSuche} + T_{avgRotationslatenz} + T_{avgTransfer}$

Suchzeit ($T_{avgSuche}$)

- Zeit in der Schreib-Lese-Köpfe (“heads”) über den Zylinder mit dem Targetsektor positioniert werden
- üblicherweise $T_{avgSuche} = 9$ ms

Festplatten-Zugriffszeit (cont.)

Rotationslatenzzeit ($T_{avgRotationslatenz}$)

- Wartezeit, bis das erste Bit des Targetsektors unter dem r/w Schreib-Lese-Kopf durchrotiert.
- $T_{avgRotationslatenz}$ = 1/2 x 1/RPMs x 60 Sek/1 Min

Transferzeit ($T_{avgTransfer}$)

- Zeit, in der die Bits des Targetsektors gelesen werden
- $T_{avgTransfer}$ = 1/RPM x 1/(Durchschn # Sektoren/Spur) x 60 Sek/ 1 Min

Beispiel für Festplatten-Zugriffszeit

Gegeben:

- Umdrehungszahl = 7.200 RPM ("Rotations per Minute")
- Durchschnittliche Suchzeit = 9 ms.
- Avg # Sektoren/Spur = 400

Abgeleitet:

- $T_{avgRotationslatenz}$
  = 1/2 x (60 Sek/7200 RPM) x 1000 ms/Sek = 4 ms.
- $T_{avgTransfer}$
  = 60/7200 RPM x 1/400 Sek/Spur x 1000 ms/Sek = 0.02 ms
- $T_{avgZugriff}$
  = 9 ms + 4 ms + 0.02 ms

Beispiel für Festplatten-Zugriffszeit (cont.)

Wichtige Punkte:

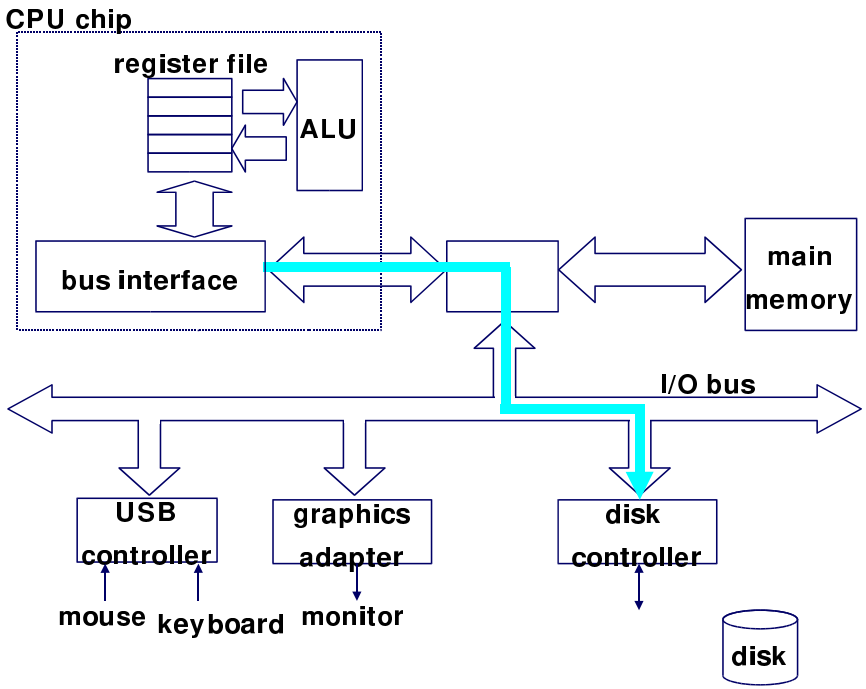
- Zugriffszeit wird von Suchzeit und Rotationslatenzzeit dominiert
- erstes Bit eines Sektors ist das “teuerste”, der Rest ist quasi umsonst
- SRAM Zugriffszeit ist ca. 4ns/Doppelwort, DRAM ca. 60 ns
  - Festplatte ist ca. 40.000 mal langsamer als SRAM
  - 2.500 mal langsamer als DRAM

Logische Festplattenblöcke

- simple, abstrakte Ansicht der komplexen Sektorengeometrie:
  - Bereich verfügbarer Sektoren wird als eine Sequenz logischer Blöcke der Größe b modelliert (0,1,2,..,n)
- Abbildung zwischen logischen Blöcken und tatsächlichen (physikalischen) Sektoren
  - ausgeführt durch eine Hard-/Firmware Einheit (Festplattencontroller)
  - konvertiert logische Blöcke zu Tripeln (Oberfläche, Spur, Sektor)
- erlauben Controller, für jede Zone Ersatzzylinder bereit zu stellen
  - Unterschied zwischen “formatierter Kapazität” und “Maximaler Kapazität”

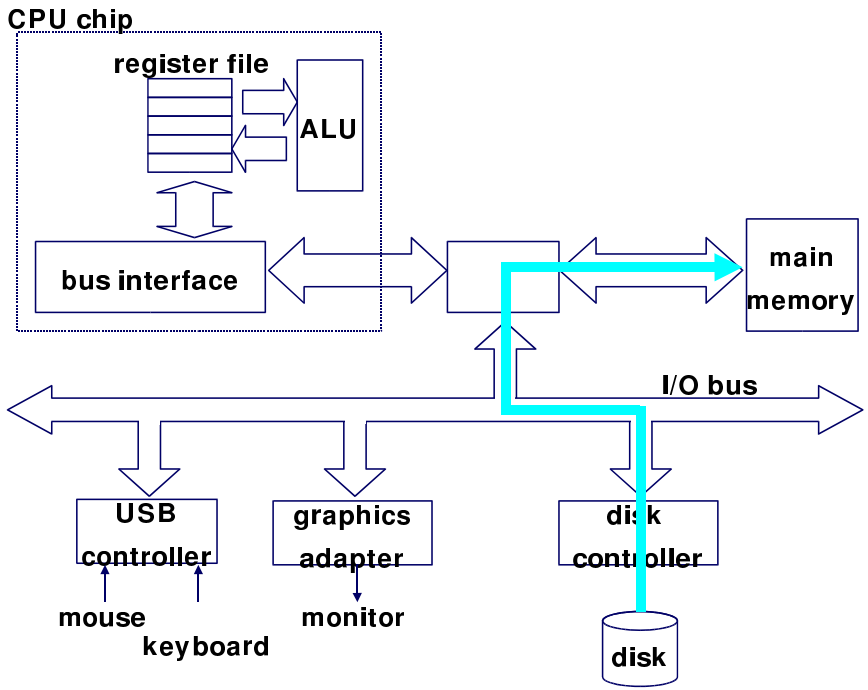
Lesen eines Festplattensektors

- CPU initiiert Lesevorgang von Festplatte
- auf Port (Adresse) des Festplattencontrollers wird geschrieben:
  - Befehl, logische Blocknummer, Zielspeicheradresse

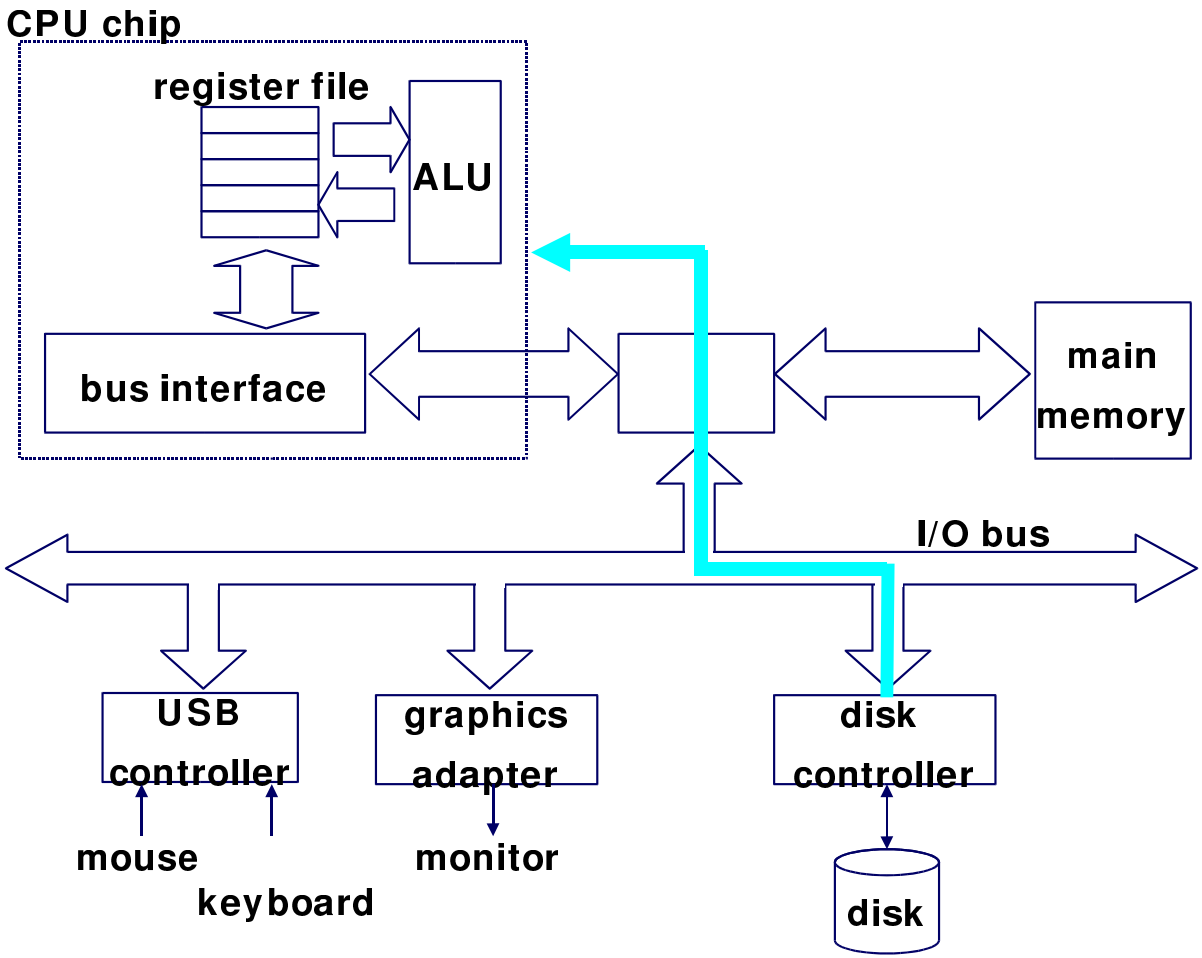


Lesen eines Festplattensektors (cont.)

- Festplattencontroller liest den Sektor aus
- führt DMA-Zugriff auf Hauptspeicher



Lesen eines Festplattensektors (cont.)



Speicherhierarchien

- Fundamentale, bleibende Eigenschaften von Hard- und Software:
  - schnelle vs. langsame Speichertechnologie
    - schnell: hohe Kosten pro Byte langsam: geringe Kosten/Byte
    - schnell: geringer Kapazität langsam: hohe Kapazität
  - Abstand zwischen CPU und Hauptspeichergeschwindigkeit vergrößert sich
  - gut geschriebene Programme haben meist eine gute Lokalität
- Motivation für spezielle Organisation von Speichersystemen:
  $\longrightarrow$ Speicherhierarchie

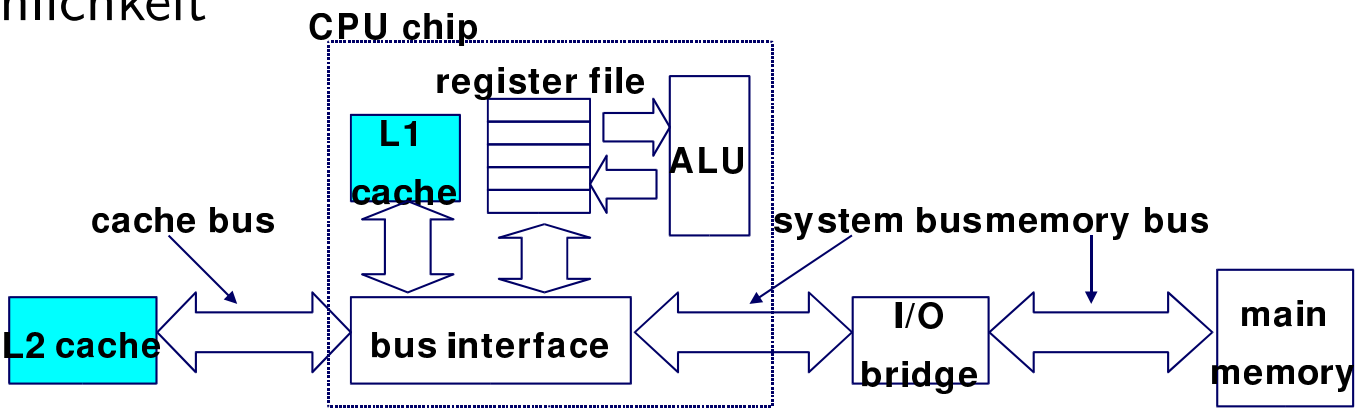
Cache Speicher

Themen

- Allgemeine Cache Speicher Organisation
- Direkt abgebildete Caches (“direct mapped”)
- Bereichsassoziative Caches (“set associative”)
- Einfluss der Caches auf die Leistung

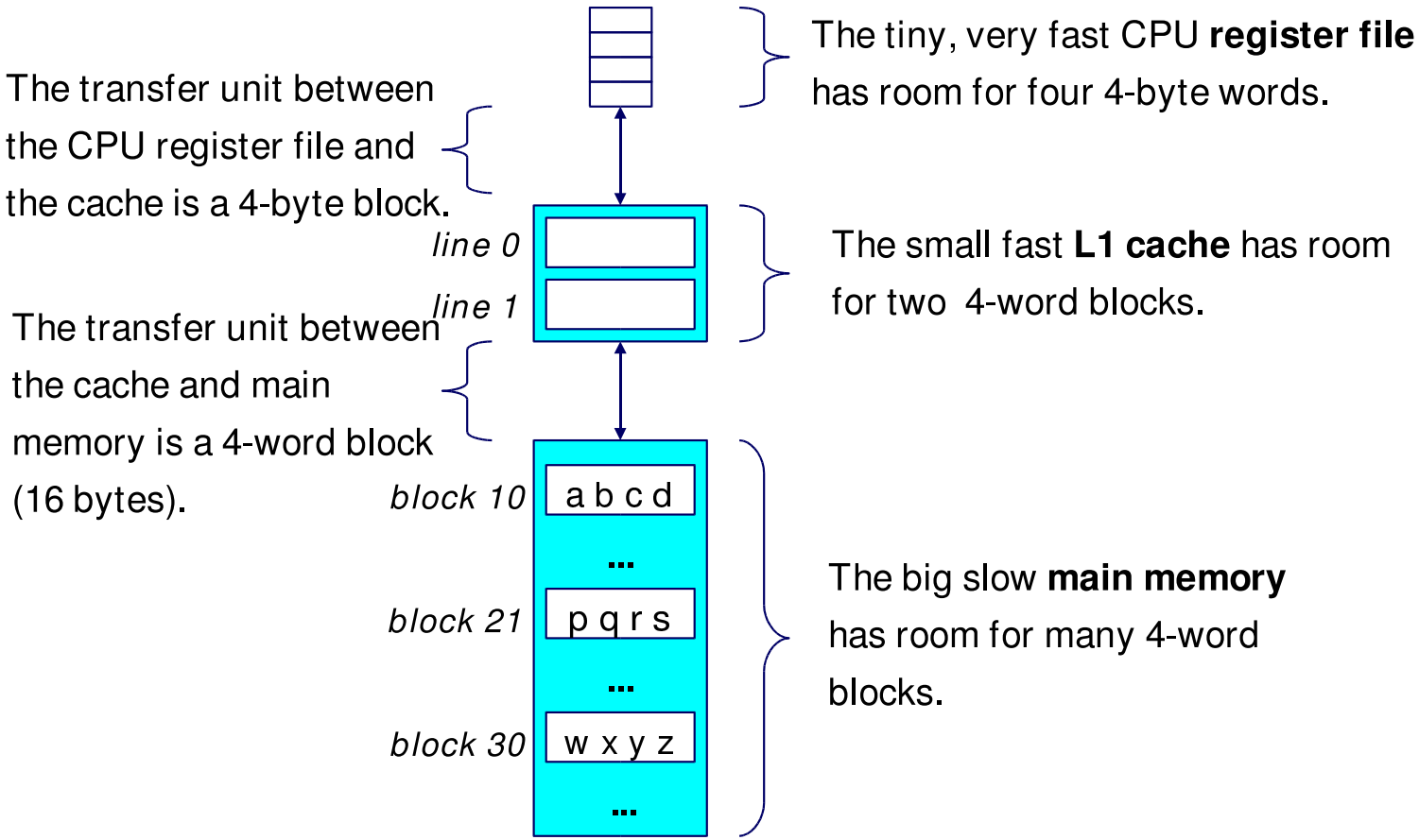
Caches Begriffsbestimmung

- kleine, schnelle Speicher
- werden durch Hardware verwaltet
- enthalten Blöcke des Hauptspeichers mit erhöhter Zugriffswahrscheinlichkeit
- Typische Struktur:



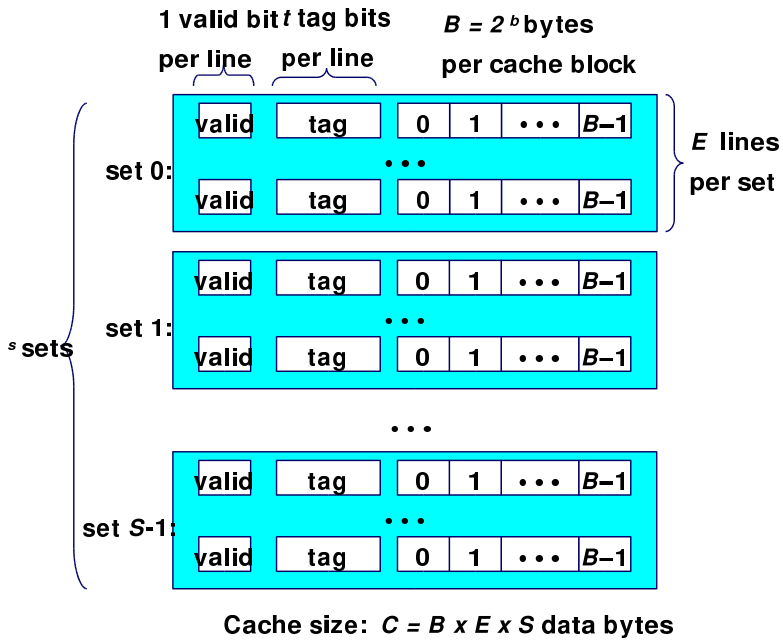
- CPU referenziert Adresse:
  - parallele Suche in L1 (level 1), L2 (level 2) und Hauptspeicher
  - erfolgreiche Suche liefert Datum
  - ggf. noch laufende Suchen werden abgebrochen

Einschieben eines L1 Cache zwischen CPU und Hauptspeicher

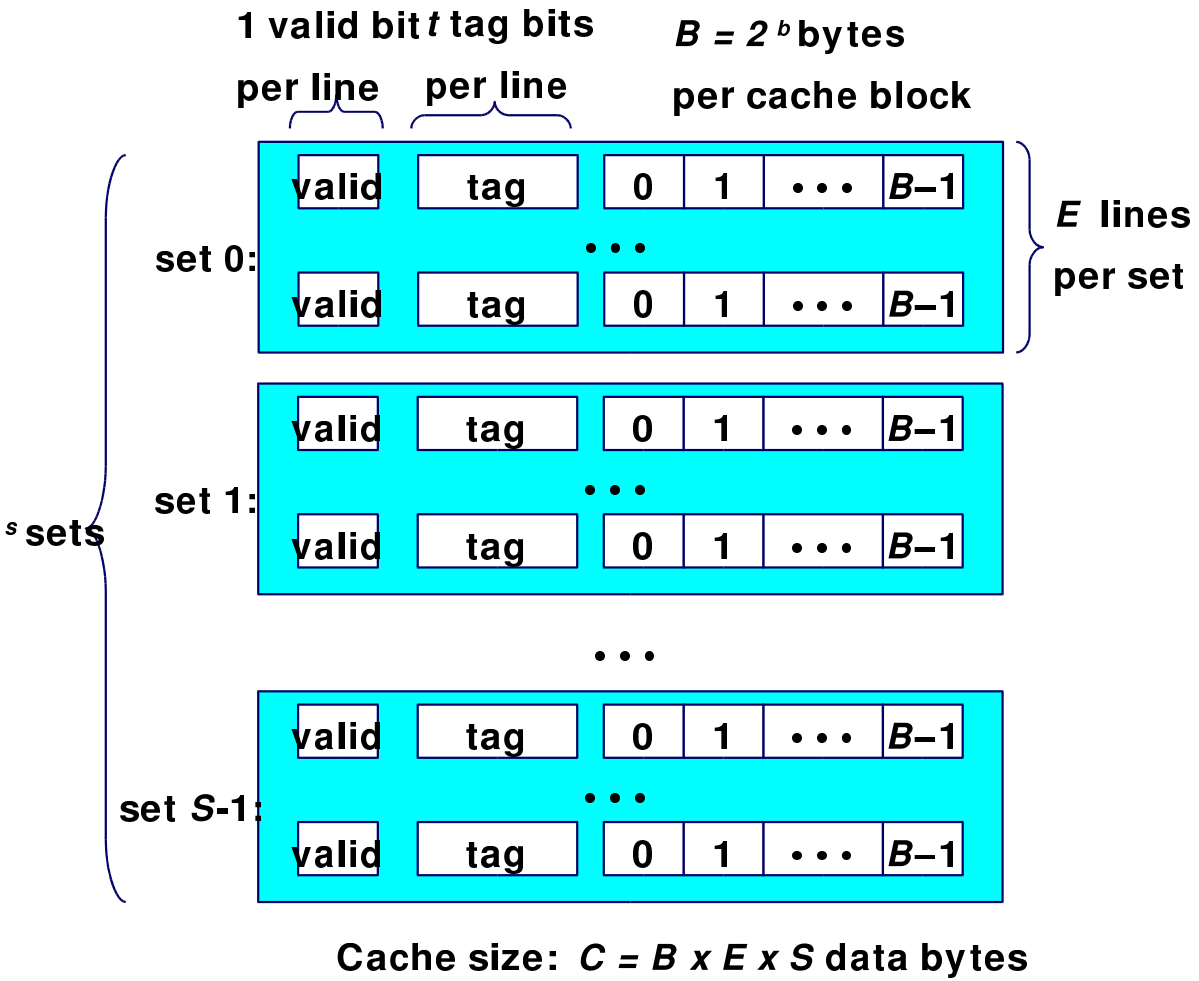


Allgemeine Organisation eines Cache Speichers

- Cache ist ein Array von Speicher-Bereichen (“sets”)
- jeder Bereich enthält eine oder mehrere Zeilen
- jede Zeile enthält einen Datenblock

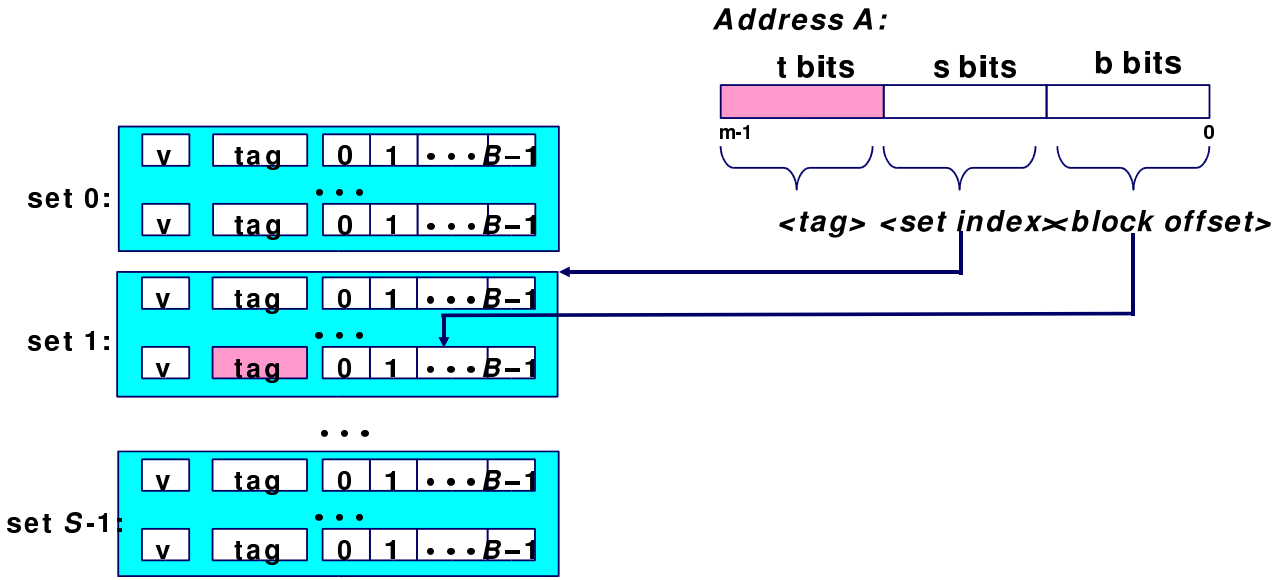


Allgemeine Organisation eines Cache Speichers (cont.)



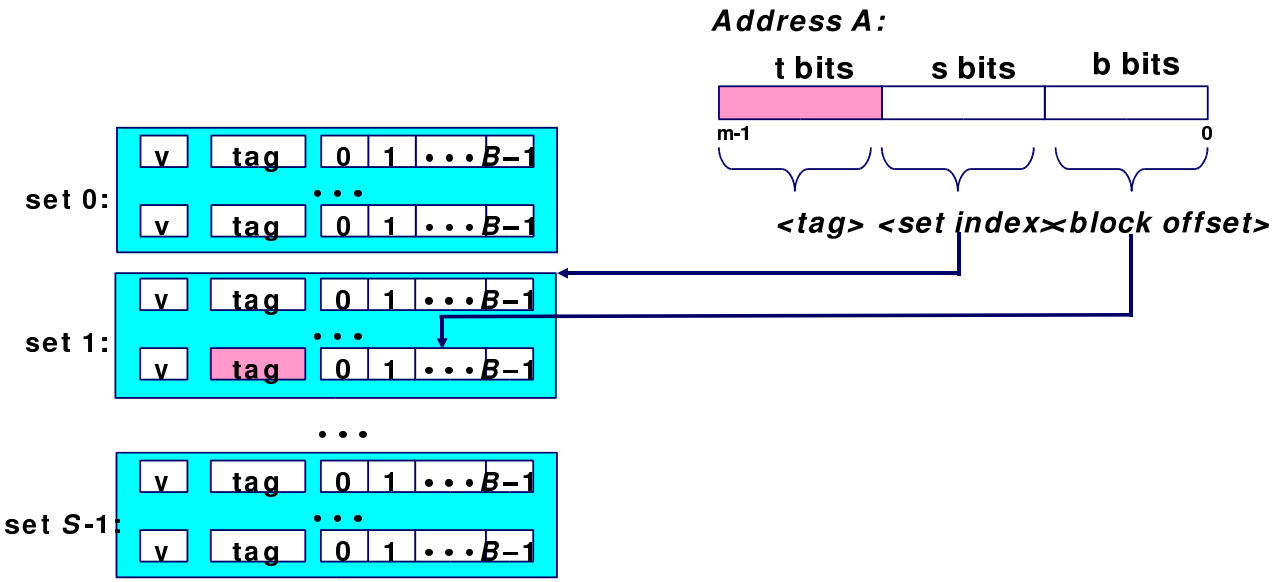
Adressierung von Caches

- Adresse A im Cache, falls:
  - Adressteil <"set index"> von A bestimmt Bereich ("set")
  - stimmen "tag" Bits einer Cache-Zeile im ("set") mit Adressteil <"tag"> von A überein?
  - ist die Zeile als gültig <"valid"> markiert?



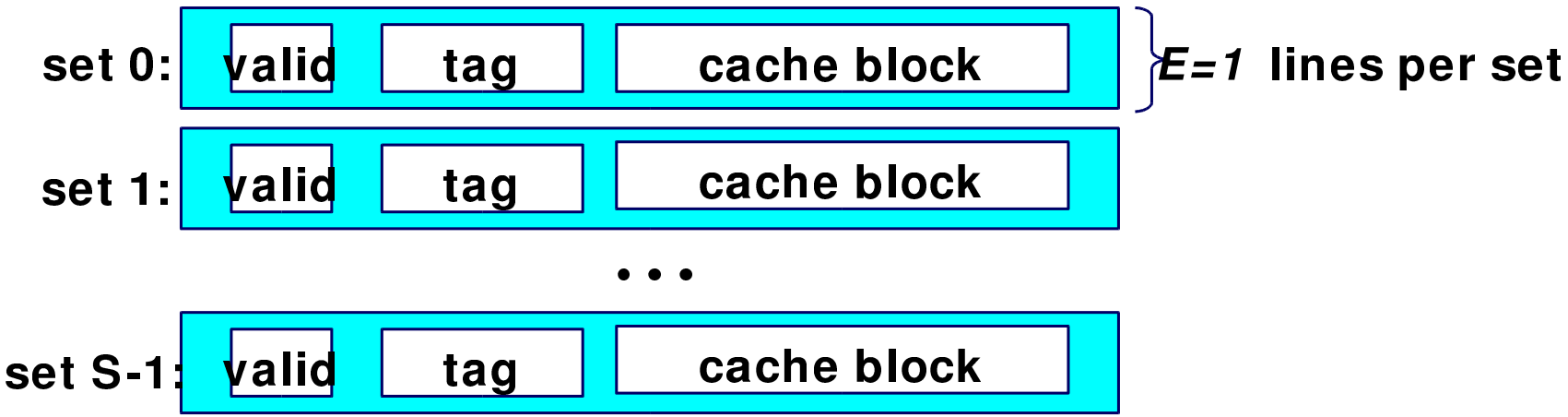
Adressierung von Caches (cont.)

- Cache-Zeile ("cache line") enhält einen Datenbereich der Größe 2^b Byte
- gesuchtes Wort ab Offset <"block offset bytes">



Direkt abgebildeter Cache (direct mapped)

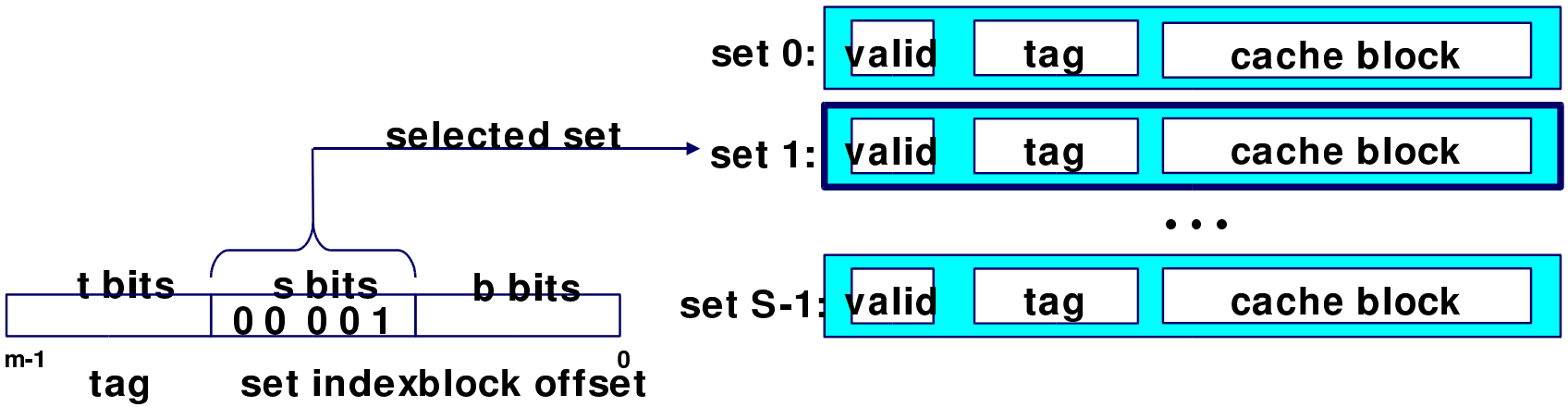
- einfachste Cache-Art
- genau 1 Zeile pro Bereich



Zugriff auf direkt abgebildete Caches

Bereichsauswahl

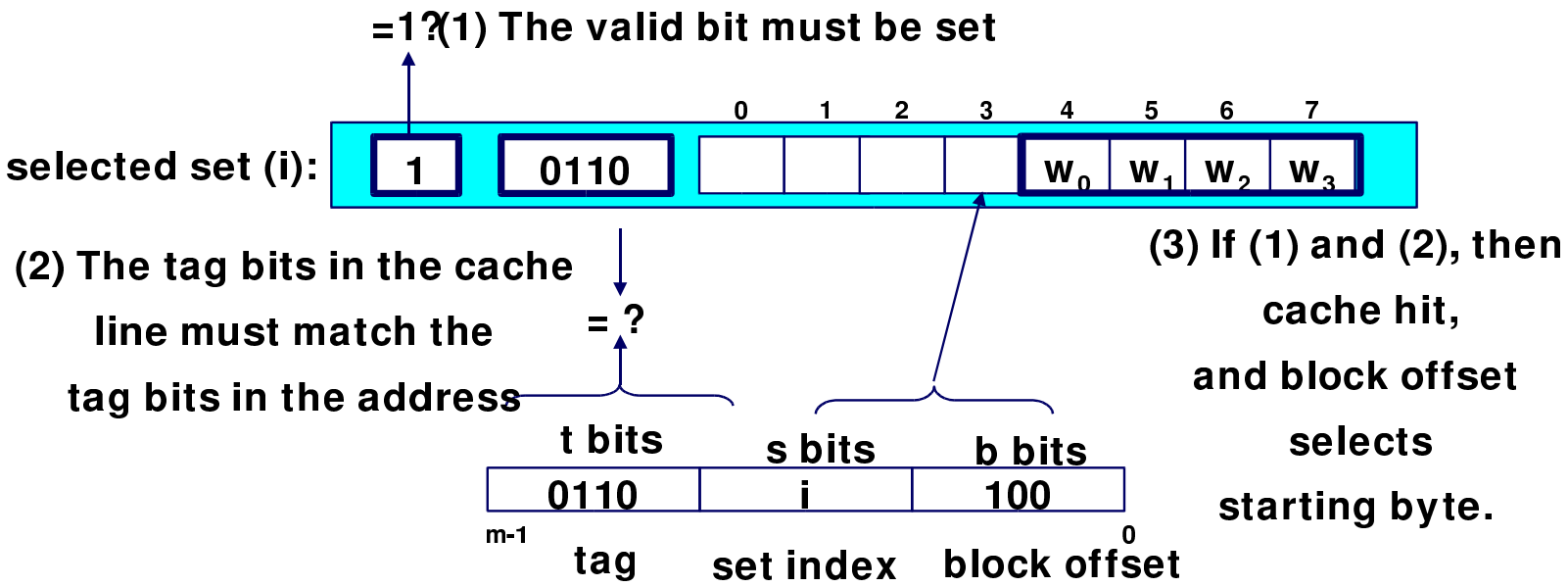
- Benutze die Bereichsindex-Bits (“set index”) um den fraglichen Bereich zu bestimmen



Zugriff auf direkt abgebildete Caches

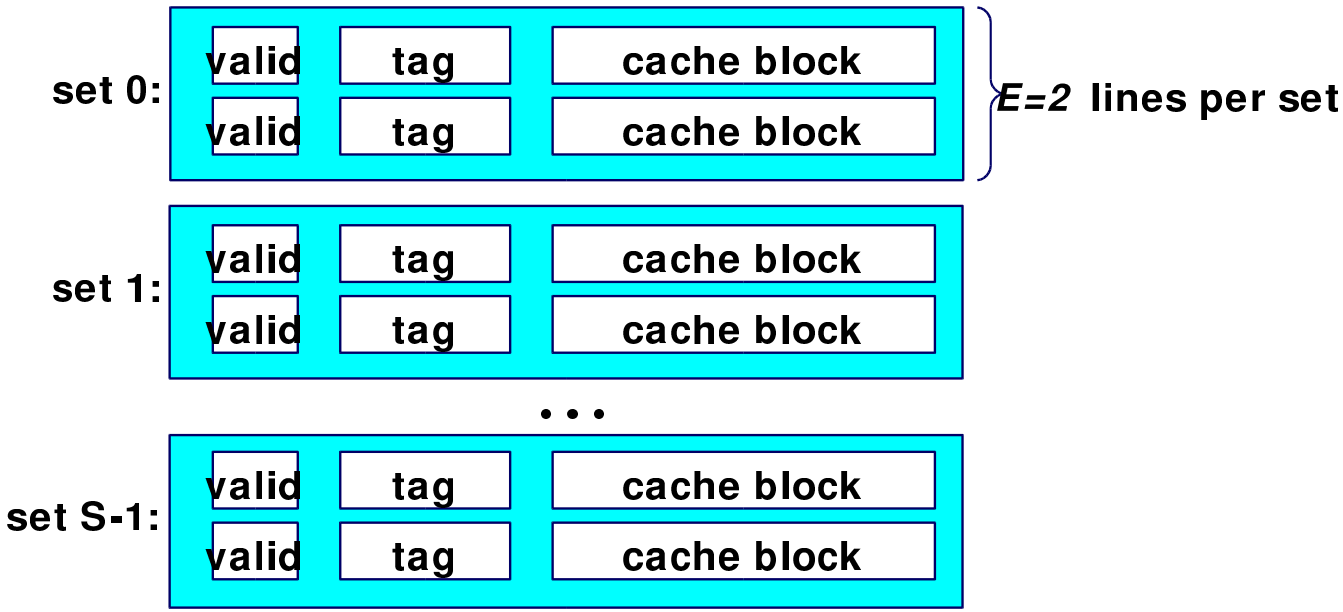
"Line matching" und Wortselektion

- "Line matching": stimmt "tag" überein?
  - Wortselektion: extrahiere Wort unter angegebenem Offset



Bereichsassoziative Caches ("set associative")

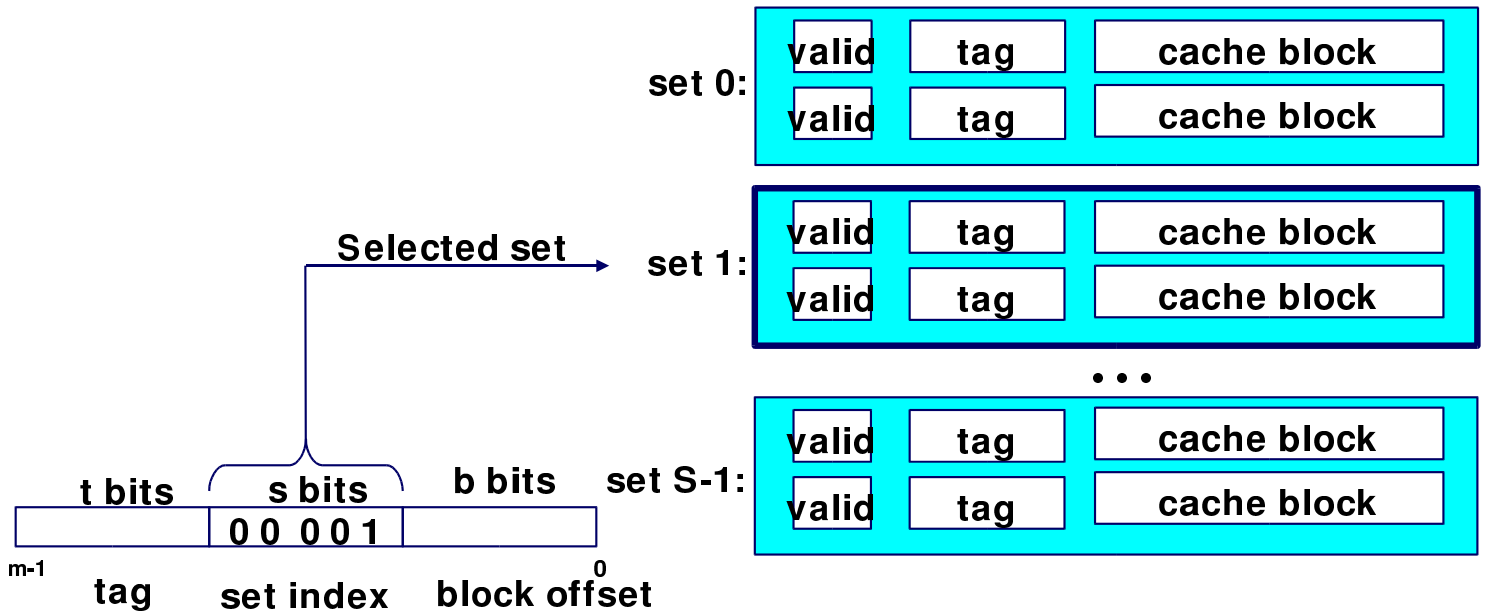
- verfügen über eine Anzahl (E) Zeilen pro Bereich
- Beispiel mit E=2 ("2-way set associative cache"):



Zugriff auf Bereichsassoziative Caches

Bereichsauswahl:

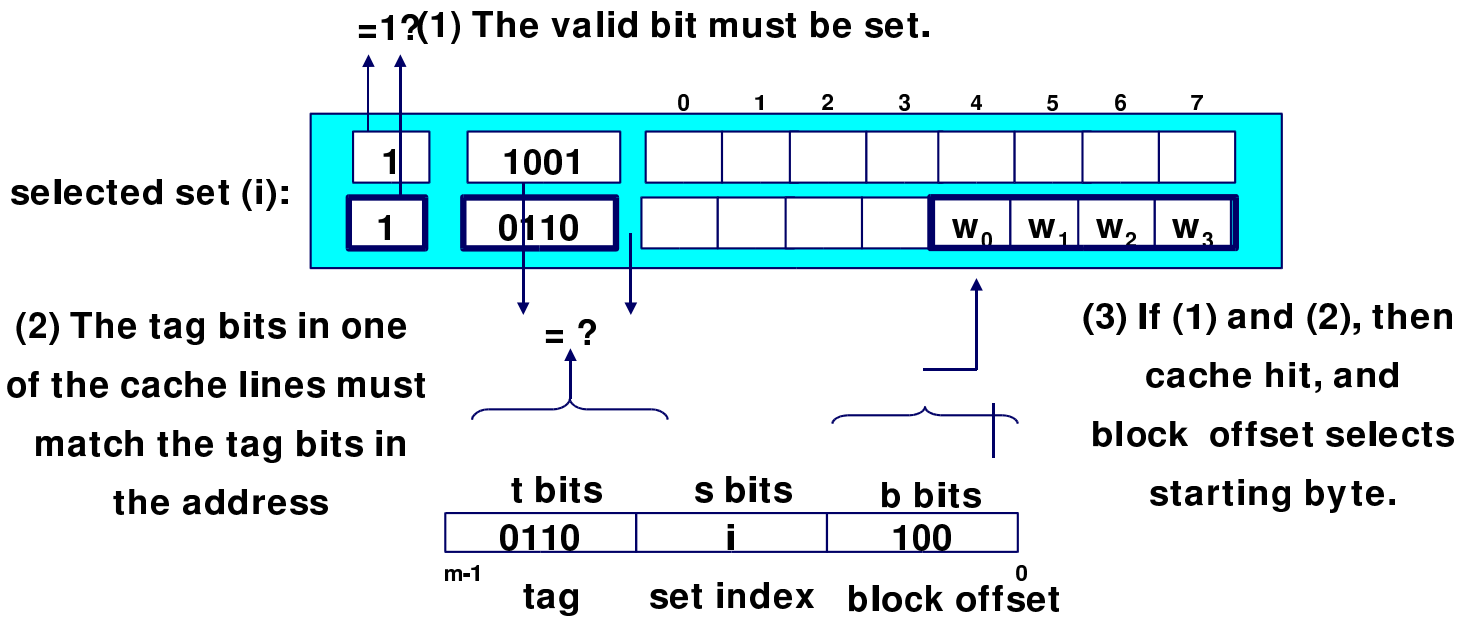
- identisch mit direkt abgebildetem Cache



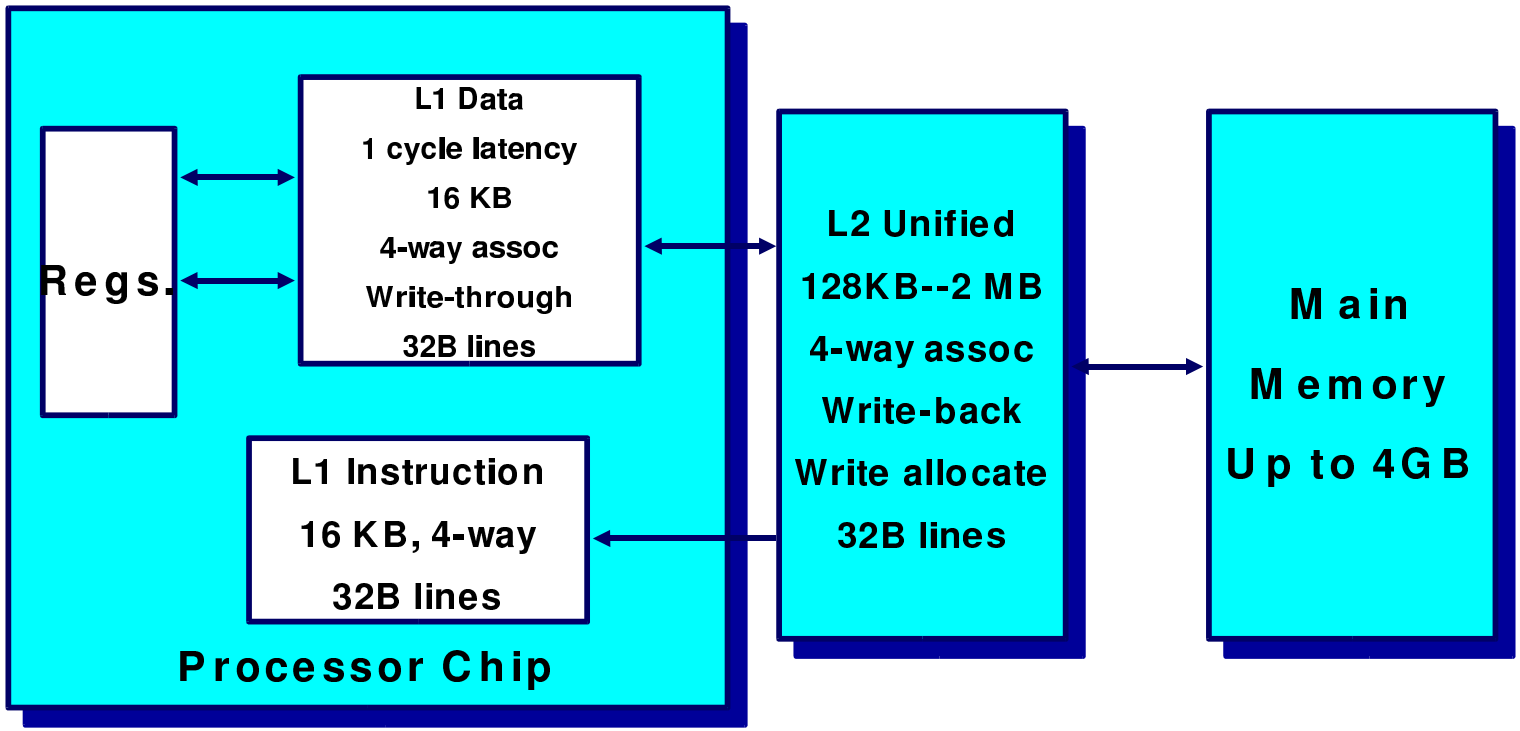
Zugriff auf Bereichsassoziative Caches

"Line matching" und Wortselektion

- "Line matching": finde gültige Zeile im ausgewählten Bereich ("set") mit übereinstimmendem "tag"
  - Wortselektion: extrahiere Wort unter angegebenem Offset



Intel Pentium Cache Hierarchie



Abschließende Bemerkungen

Der Programmierer kann sein Programm für maximale Cacheleistung optimieren:

- durch entsprechende Organisation der Datenstrukturen
- durch Steuerung des Zugriffs auf die Daten
  - Geschachtelte Schleifenstruktur
  - Blockbildung ist eine übliche Technik

Abschließende Bemerkungen (Forts.)

Alle Systeme bevorzugen einen "Cache-freundlichen Code":

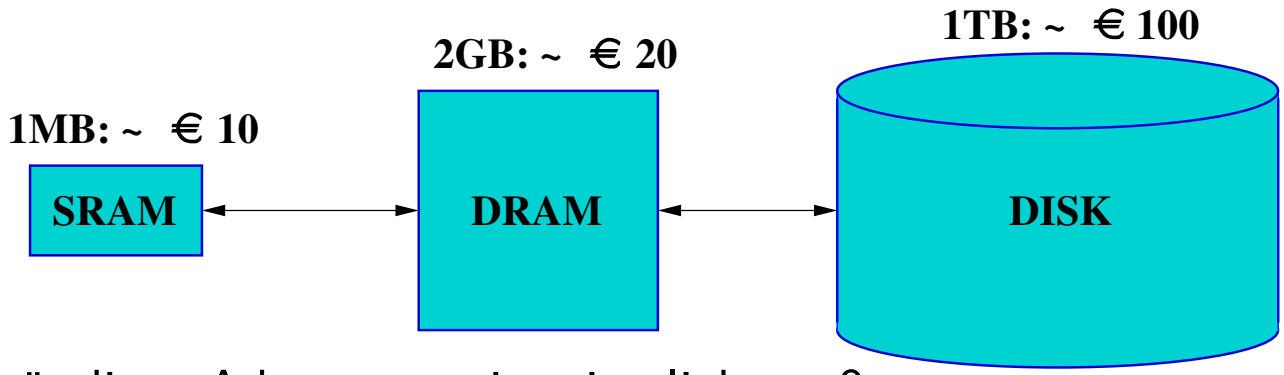
- Erreichen der optimalen Leistung ist sehr plattformspezifisch
  - Cachegrößen, Zeilengrößen, Assoziativität etc.
  - aber: generelle Empfehlungen für "Cache-freundlichen Code":
    - "working set" klein $\longrightarrow$ zeitliche Lokalität
    - kleine Adressfortschaltungen ("strides") $\longrightarrow$ räumliche Lokalität

Virtuelle Speicher

Motivation für einen virtuellen Speicher

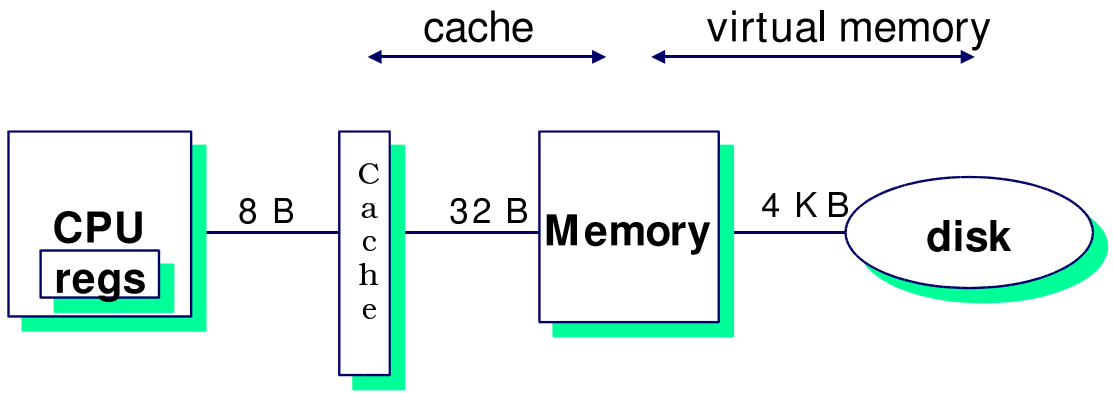
1. Benutzung von Hauptspeicher als "*Cache*" für die Festplatte
   - Prozessesaressraum kann physikalische Speichergröße übersteigen
   - Summe der Adressräume mehrerer kleiner Prozesse kann physikalischen Speicher übersteigen
2. Vereinfachung der Speicherverwaltung
   - viele Prozesse liegen im Hauptspeicher
     - jeder Prozess mit seinem eigenen Adressraum
   - nur "aktiver" Code und Daten sind tatsächlich im Speicher
     - Teile dem Prozess bei Bedarf mehr Speicher zu
3. Bereitstellung von Schutzmechanismen
   - ein Prozess kann einem anderen nicht ins Gehege kommen
     - sie operieren in verschiedenen Adressräumen
   - Benutzerprozess hat keinen Zugriff auf privilegierte Informationen
     - jeder virt. Adressraum hat eigene Zugriffsrechte

Motivation Nr.1: DRAM ist ein "Cache" für die Festplatte



- Vollständiger Adressraum ist ziemlich groß:
  - 32-Bit Adressen: $\approx 4x10^9$ (4 Milliarden) Bytes
  - 64-Bit Adressen: $\approx 16x10^{16}$ (Quintillionen) Bytes
- Speichern auf der Festplatte ist $\approx$ 180x billiger als im DRAM
  - 1 TB DRAM: $\approx$ €18.000
  - 1 TB Festplatte: $\approx$ €100
- kostengünstig Zugriff auf große Datenmengen
  - Hauptanteil der Daten auf Festplatte speichern

Ebenen in der Speicherhierarchie

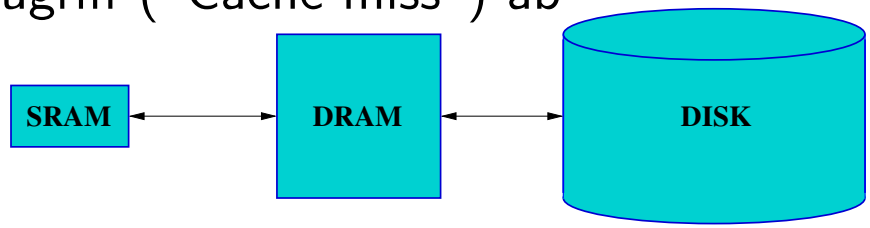


	Register	Cache	Memory	Disk Memory
size:	32 B	32 KB-4MB	1024 MB	100 GB
speed:	1 ns	2 ns	30 ns	8 ms
$/Mbyte:		$125/MB	$0.20/MB	$0.001/MB
line size:	8 B	32 B	4 KB	

larger, slower, cheaper

DRAM versus SRAM als "Cache"

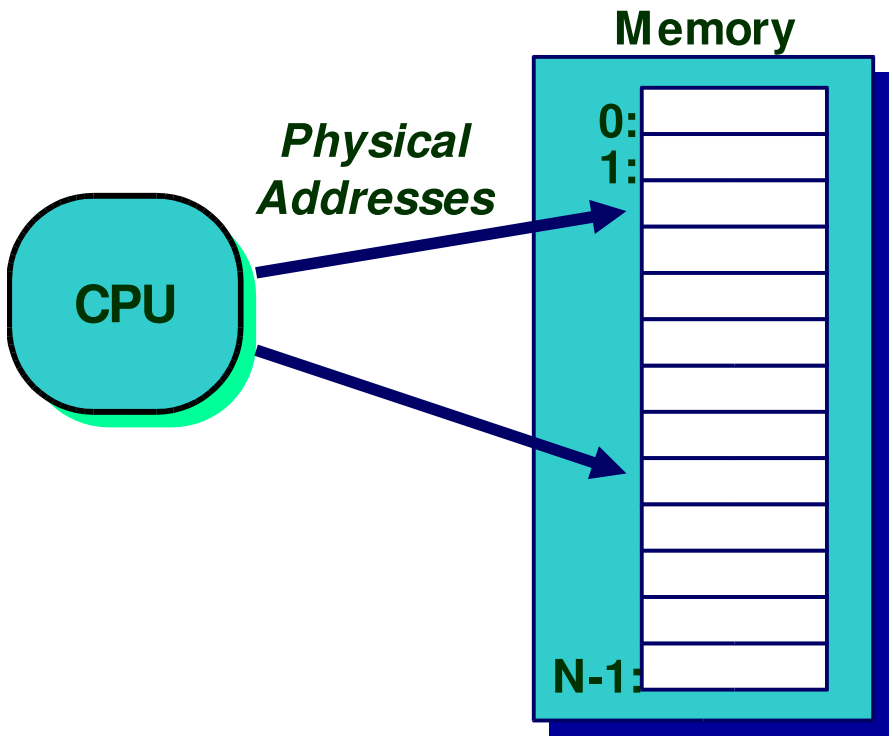
- DRAM vs. Festplatte ist extremer als SRAM vs. DRAM
- Zugriffswartezeiten
  - DRAM $\approx$ 10x langsamer als SRAM
  - Festplatte $\approx$ 100.000x langsamer als DRAM
    - Wichtigkeit der Nutzung der räumlichen Lokalität:
    - erstes Byte ist $\approx$ 100.000x langsamer als nachfolgende Bytes $\rightarrow$ deutlich signifikanter als z. B. 4fache Verbesserung durch Seitenmodus ("Page-Mode") gegenüber regulärem RAM-Zugriff
- Schlussfolgerung
  - Designentscheidung für DRAM als Cache hängt von den enormen Kosten bei Fehlzugriff ("Cache miss") ab



Ein System mit ausschließlich physikalischem Speicher

Beispiele:

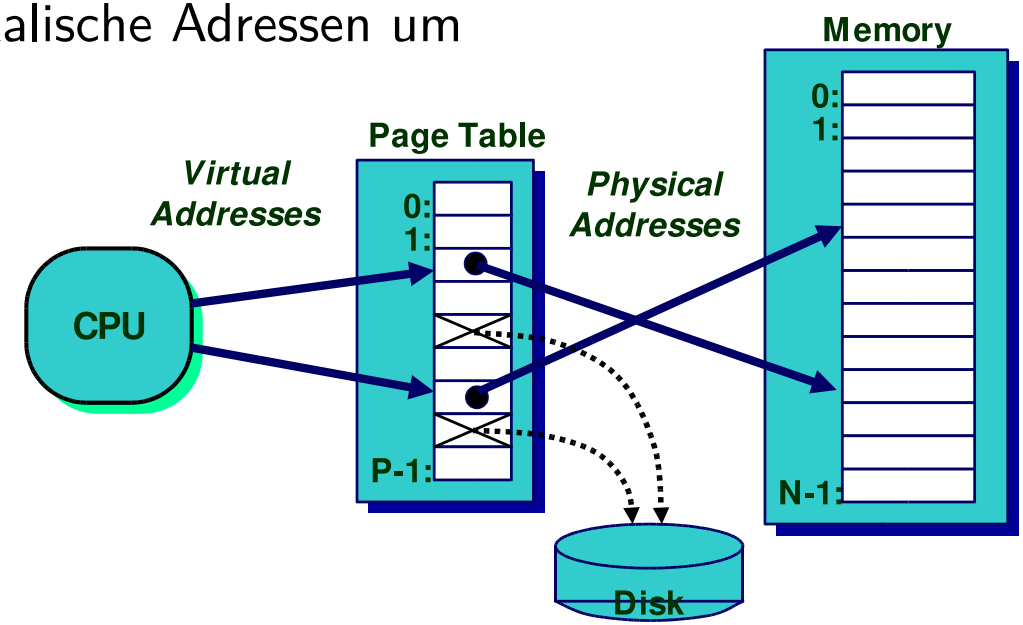
- die meisten Cray Maschinen, frühe PCs, fast alle eingebetteten Systeme etc.



Ein System mit virtuellem Speicher

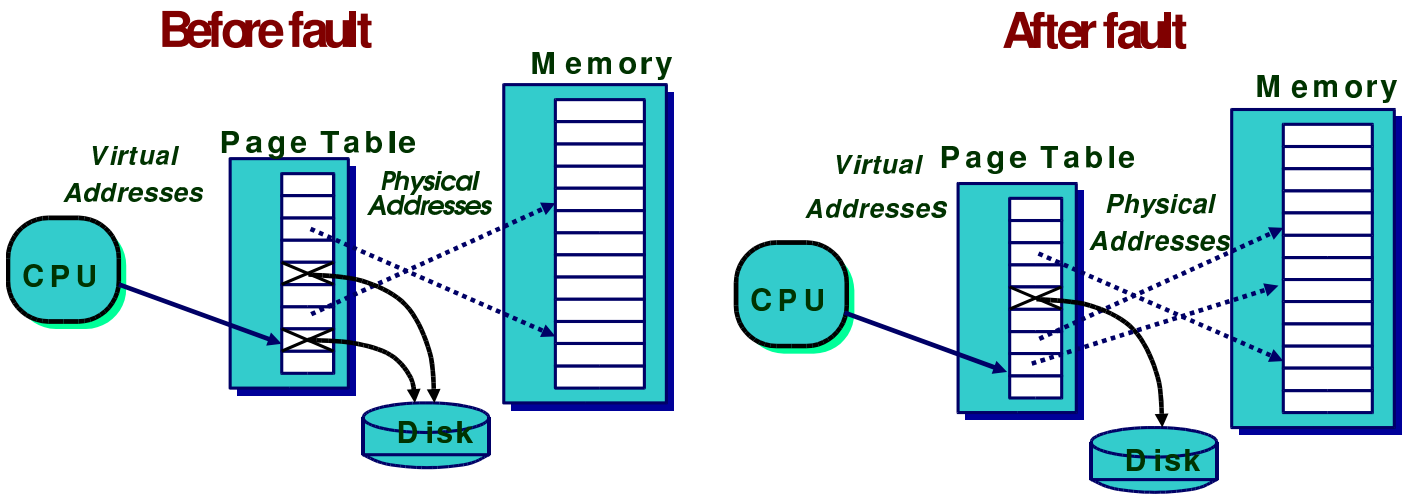
Beispiele:

- Workstations, Server, moderne PCs etc.
- Adressübersetzung:
  - Betriebssystem (OS) verwalteter Seitentabelle ("page table")
  - Hardware wandelt via Seitentabelle virtuelle Adressen in physikalische Adressen um



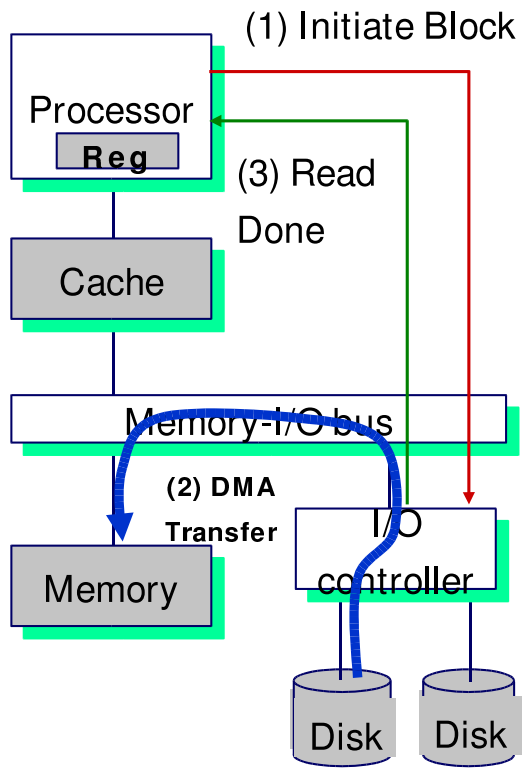
Seitenfehler

- Seiten-Tabelleneintrag zeigt auf Startadresse der virt. Seite auf Platte
- "exception handler" des OS wird aufgerufen, um Daten von der Festplatte in den Speicher zu laden
  - laufender Prozess wird unterbrochen, andere können weiterlaufen
  - OS hat die Kontrolle über die Plazierung der neuen Seite im Hauptspeicher (Ersetzungsstrategien) etc.



Einen Seitenfehler behandeln

- Prozessor meldet dem Controller
  - Lies Block der Länge P ab der Festplattenadresse X und speichere sie ab Speicheradresse Y ab.
- Lesezugriff erfolgt in Form von:
  - Direct Memory Access (DMA)
  - Kontrolle durch I/O Controller
- I/O Controller meldet Abschluss
  - Gibt Interrupt an den Prozessor
  - OS lässt unterbrochenen Prozess weiterlaufen



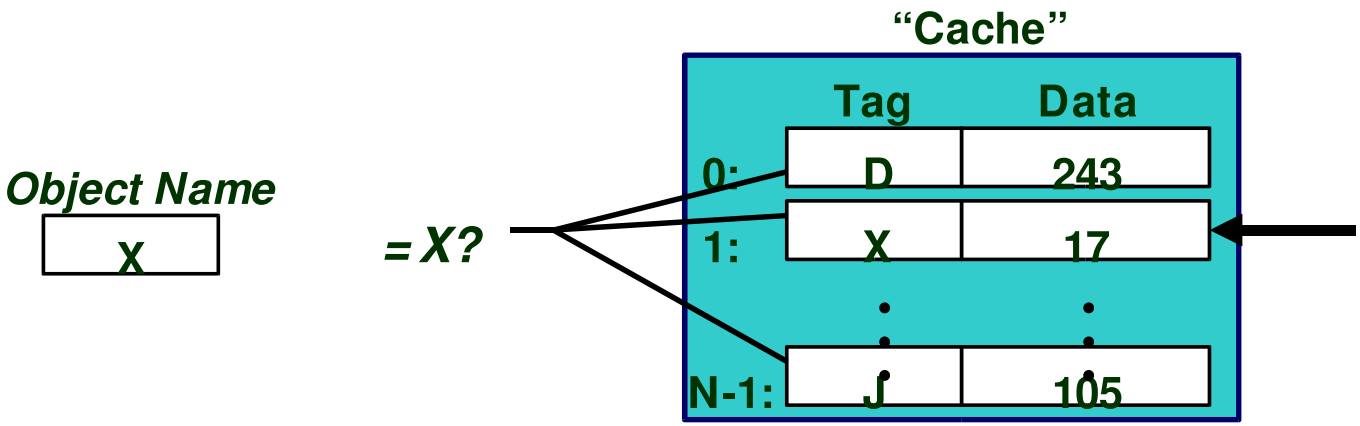
Ergänzende Literatur

Zur Rechnerarchitektur (Teil 2, 7. Termin):

[1] Randal E. Bryant and David O'Hallaron.
Computer systems.
pages 454–531, 690–700. Pearson Education, Inc., New Jersey, 2003.

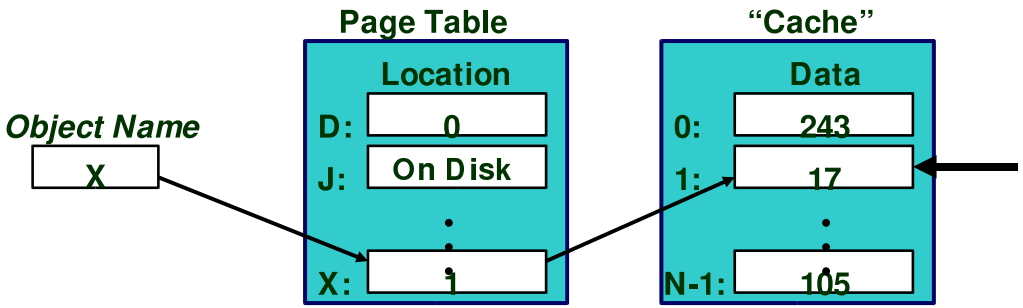
DRAM als Cache

Auffinden eines Objekts in einem Cache



- *Tag* wird zusammen mit den Daten in der Cachezeile gespeichert
- Bildet vom Cacheblock auf Speicherblöcke ab
  - Spart einige Bits, da nur *Tag* gespeichert wird
- Kein *Tag* für nicht im Cache befindlichen Block
- Auslesen der Informationen mit spezieller Hardware
  - kann schnell mit mehreren *Tags* vergleichen

Auffinden eines Objekts in einem Cache bei virtueller Adressierung



DRAM Cache

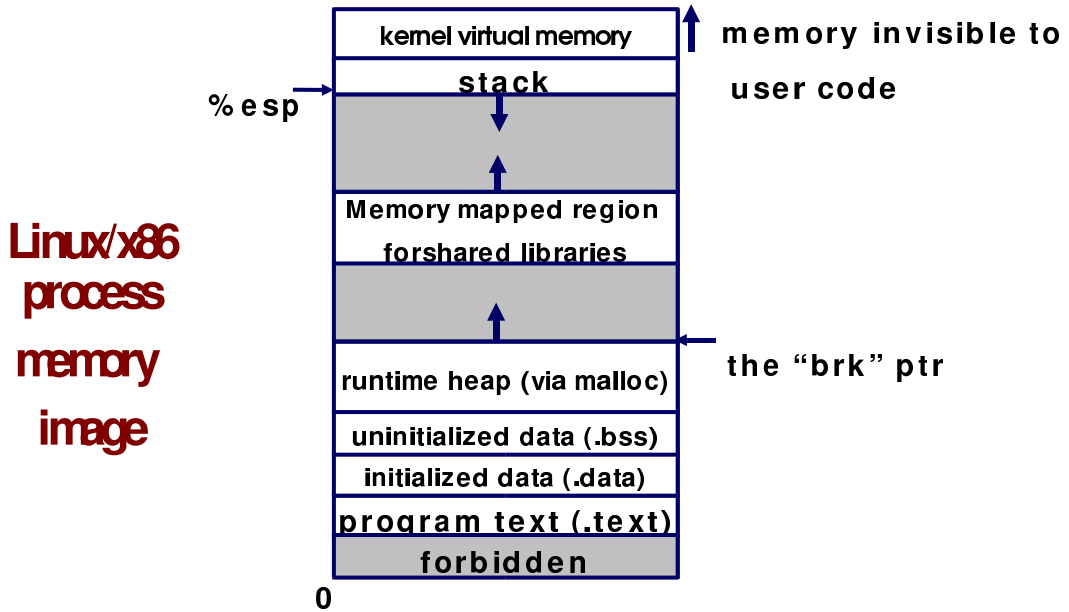
- Jede zugeteilte Seite des virtuellen Speichers hat einen Eintrag in der Seiten-Tabelle
- Abbilden von virtuellen Seiten auf physikalische
- Seiten-Tabelleneintrag existiert, selbst wenn die Seite nicht im Speicher liegt
  - Gibt Festplattenadresse an
  - Einzige Methode, um Seite zu finden
- OS liest Informationen aus

Virtueller Speicher: Speicherverwaltung

Motivation für virt. Speicher: Speicherverwaltung

Mehrere Prozesse können im physikalischen Speicher liegen

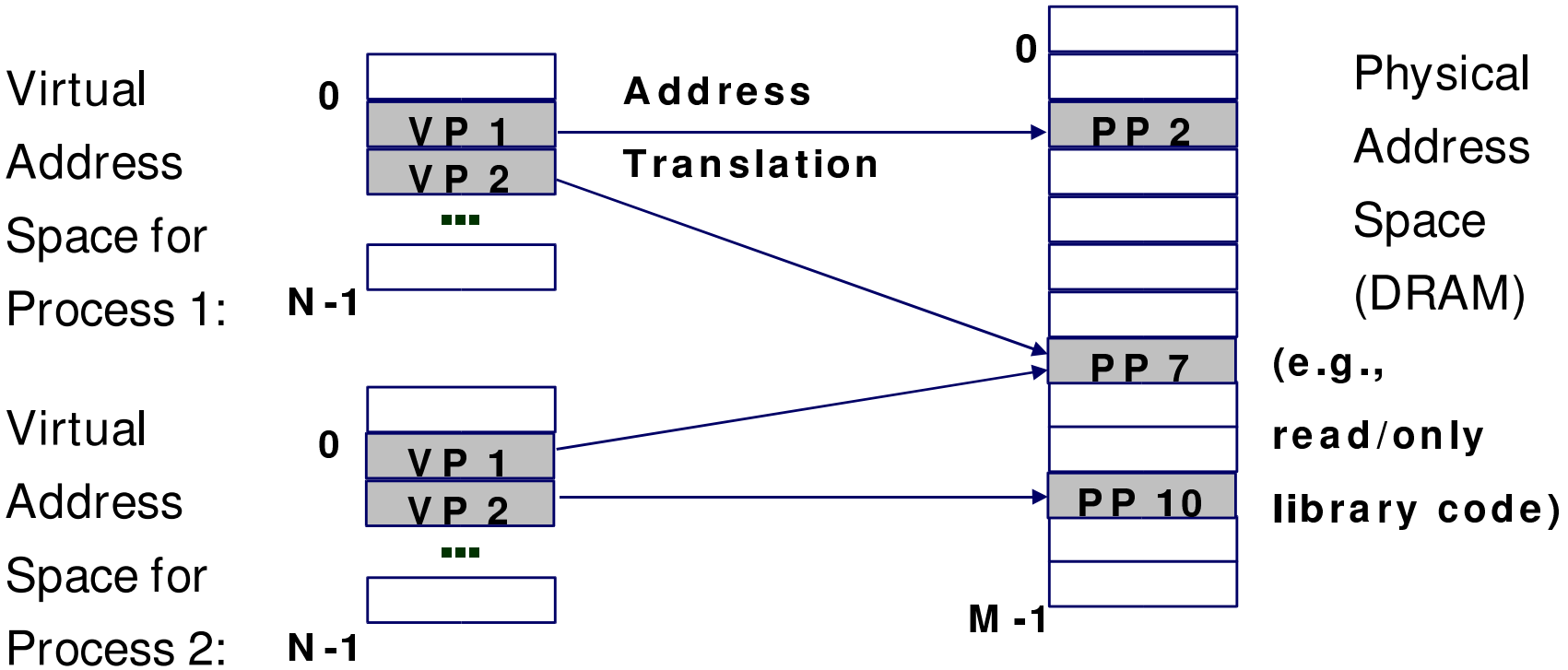
- Wie werden Adresskonflikte gelöst?
- Was passiert, wenn zwei Prozesse auf dieselbe Adresse zugreifen?



Auflösung der Adresskonflikte: Separate Virt. Adr. Räume

- Virtuelle und physikalische Adressräume sind in gleichgroße Blöcke unterteilt
  - Blöcke werden Seiten (“Pages”) genannt (sowohl virtuelle als auch physikalische)
- Jeder Prozess hat seinen eigenen virtuellen Adressraum
  - Das Betriebssystem kontrolliert wie virtuelle Seiten auf den physikalischen Speicher abgebildet werden

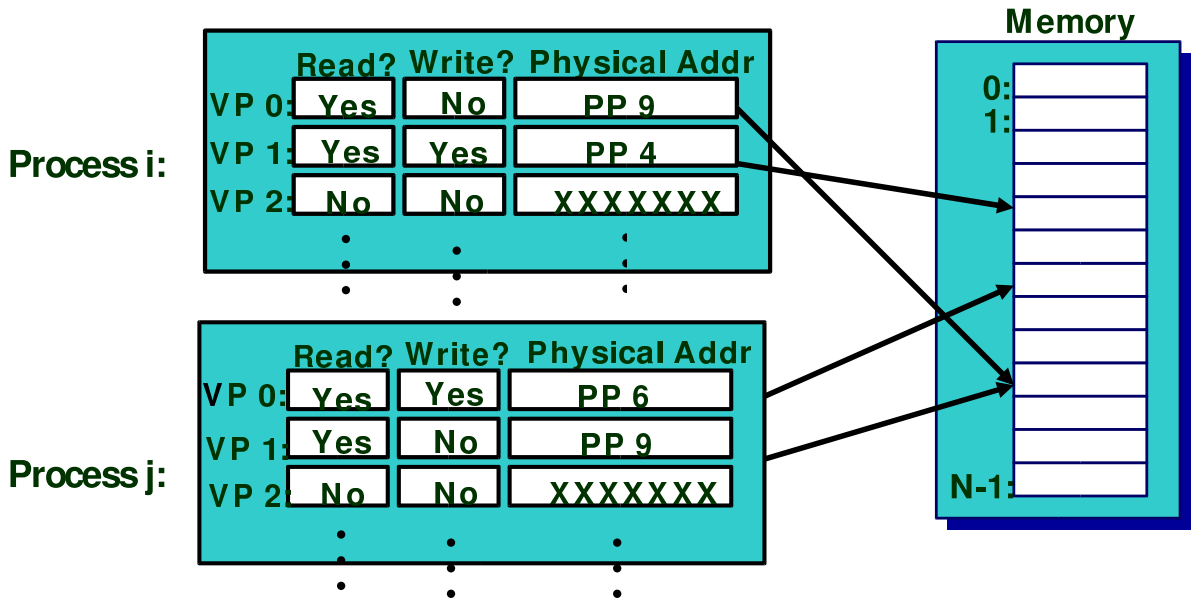
Auflösung der Adresskonflikte: Separate Virt. Adr. Räume (cont.)



Virtuelle Speicher: Schutzmechanismen

Seiten-Tabelleneintrag enthält Informationen über Zugriffsrechte

- Hardware erzwingt den Schutz (“Trap” (“Exception”) in OS bei Verstoß)



VM ("Virtual Memory"): Adressumsetzung

Virtueller Adressraum

- V = {0, 1, ..., N-1}

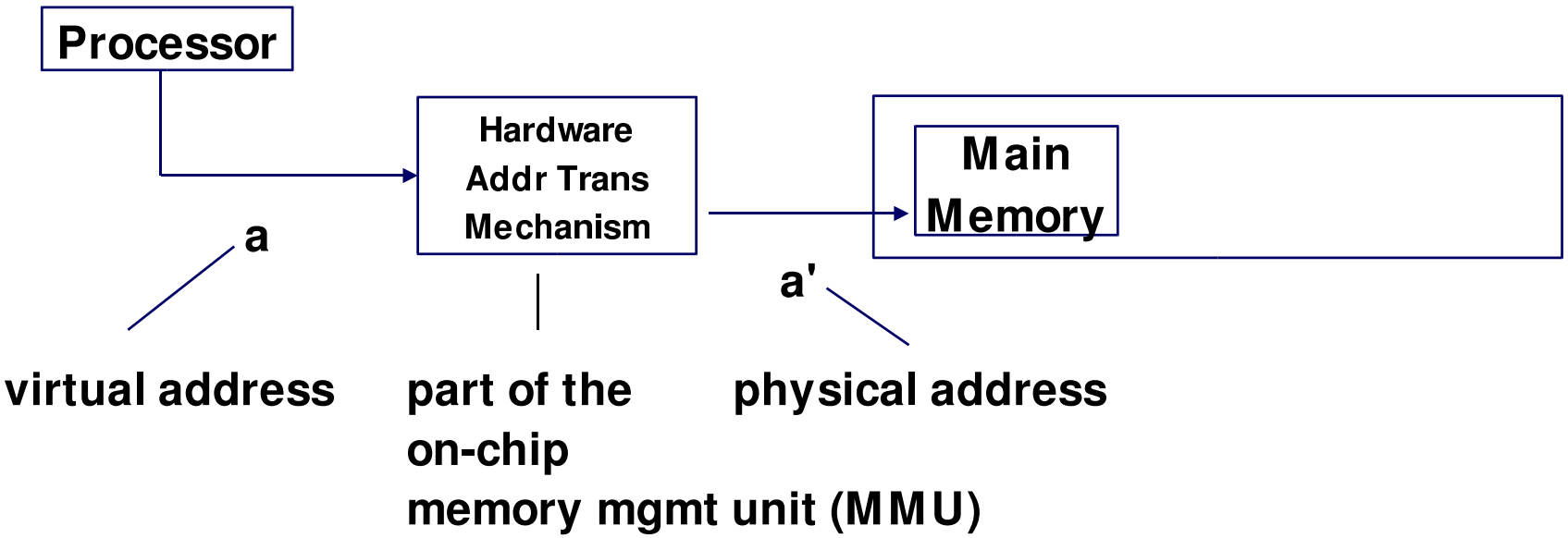
Physikalischer Adresssraum

- P = {0, 1, ..., M-1}
- M<N

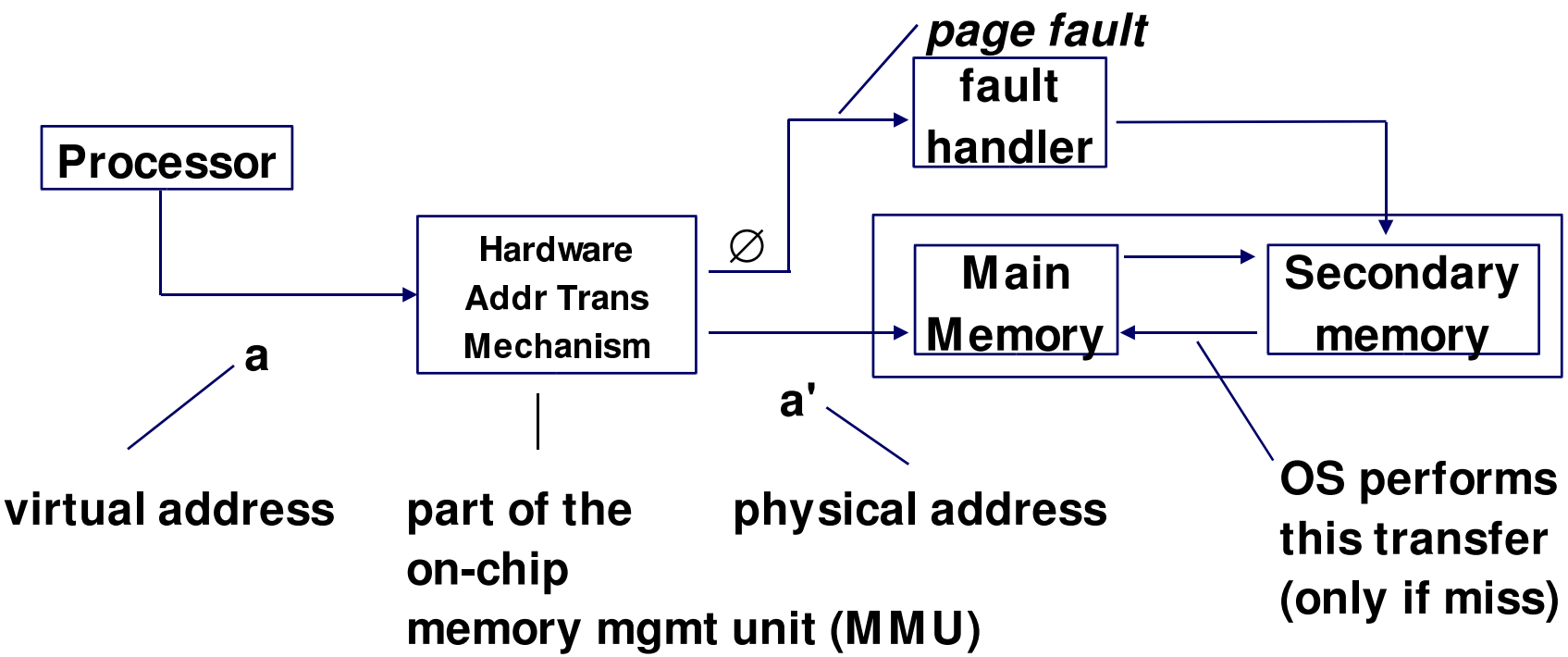
Adressumsetzung

- MAP: $V \rightarrow P \cup \{\varnothing\}$
- Für eine virtuelle Adresse a:
  - MAP(a) = a', wenn Daten bei virtueller Adresse a und physikalischer Adresse a' in P sind
  - MAP(a) = ∅, wenn Daten bei virtueller Adresse a nicht im physikalischen Speicher sind, d. h. entweder ungültig oder nur auf Festplatte gespeichert sind.

VM Adressumsetzung: Hit



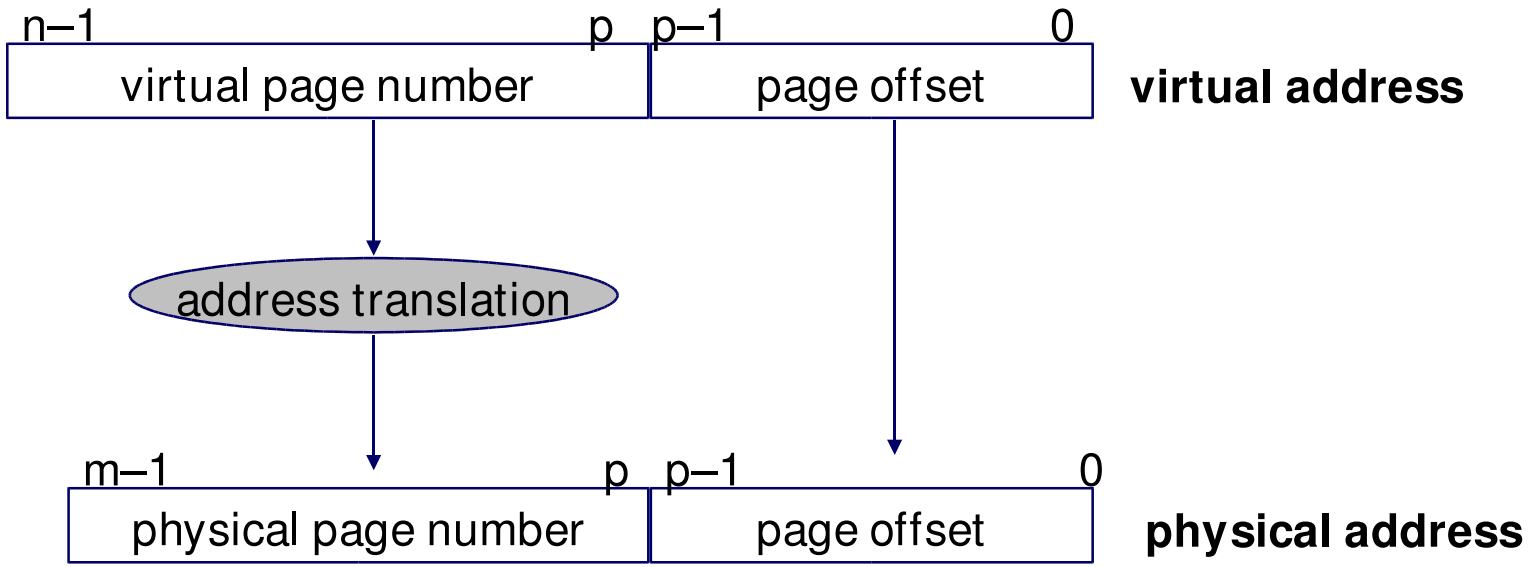
VM Adressumsetzung: Miss



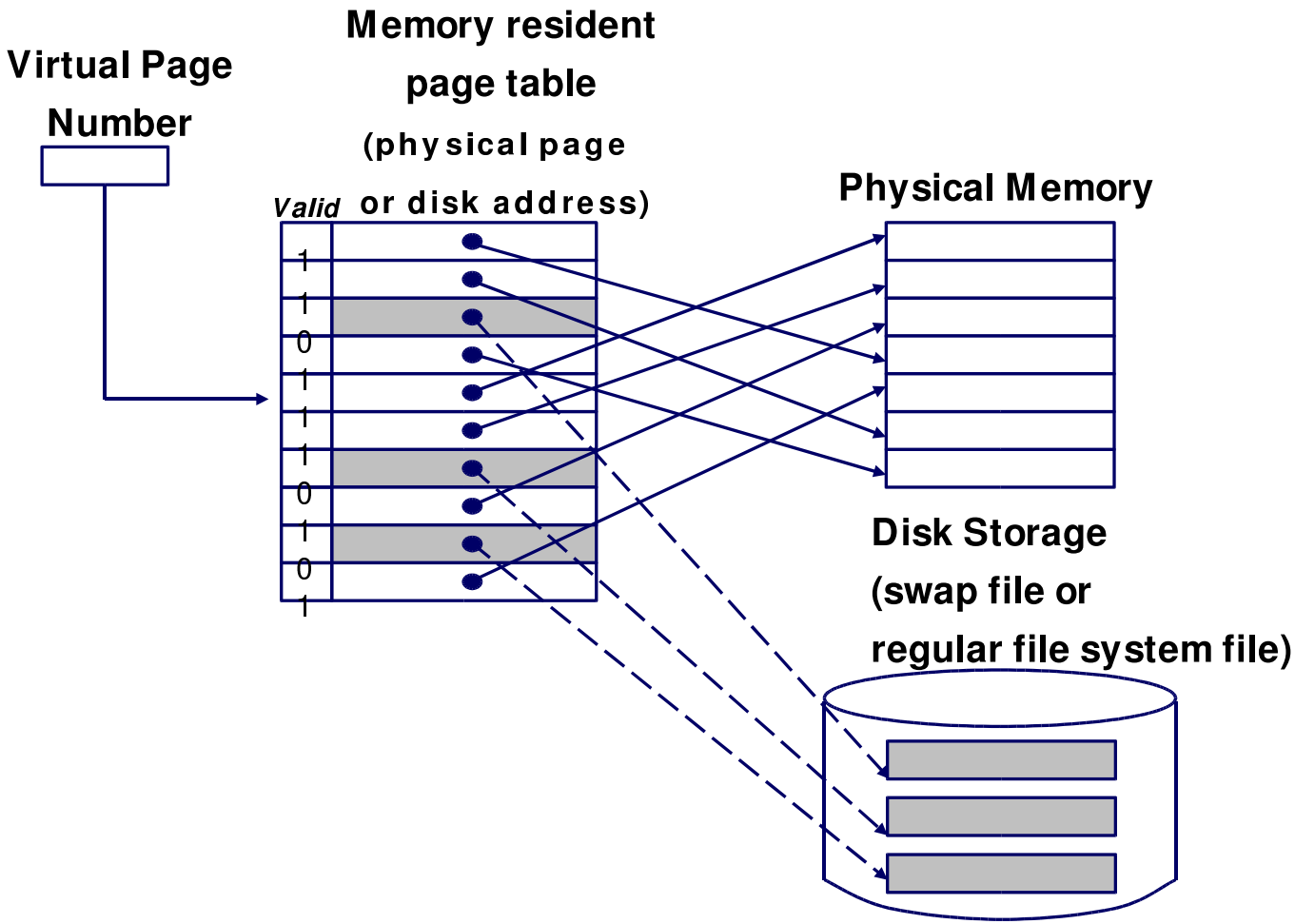
VM Adressumsetzung

Parameter

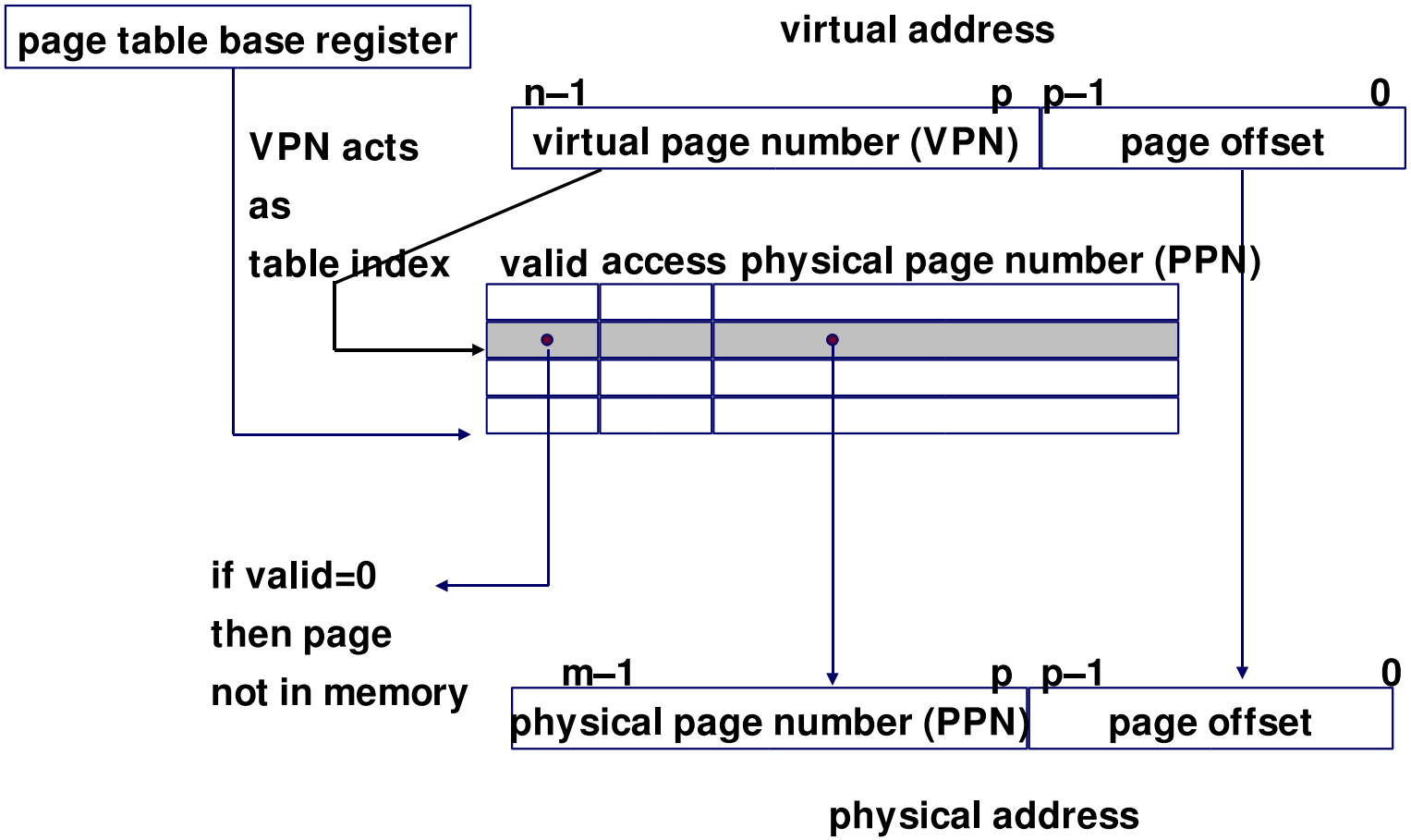
- P = 2^p = Seitengröße (Bytes)
- N = 2^n = Limit der virtuellen Adresse
- M = 2^m = Limit der physikalischen Adresse



Seiten-Tabellen



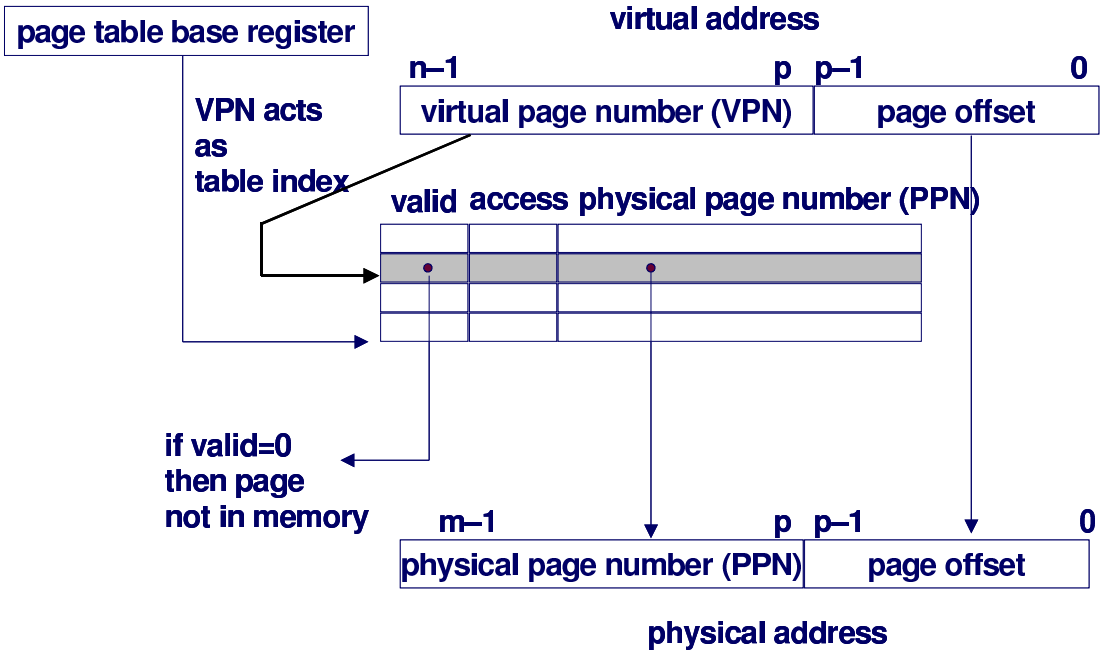
Adressumsetzung via Seiten-Tabellen



Seiten-Tabellenoperation

Umsetzung:

- Eigene Menge von Seiten-Tabellen für jeden Prozess
- VPN ("Virtual Page Number") bildet den Index in der Seiten-Tabelle (zeigt auf einen Seiten-Tabelleneintrag)



Seiten-Tabellenoperation

Berechnung der physikalischen Adresse:

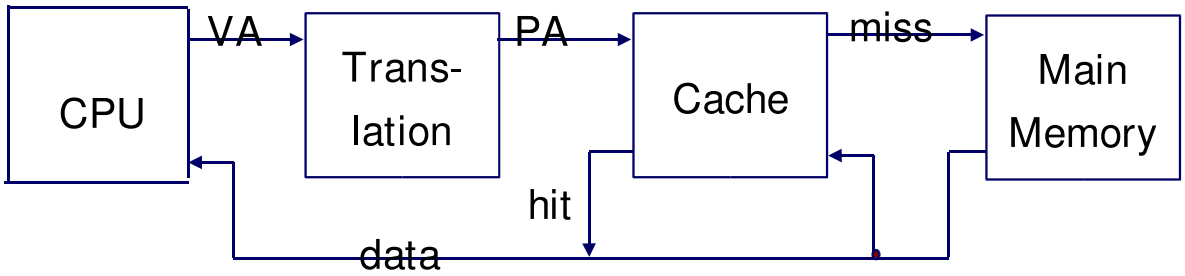
- Seiten-Tabelleneintrag liefert Informationen über die Seite.
  - wenn (valid Bit = 1), dann ist die Seite im Speicher, d.h. benutze physikalische Seitennummer (PPN "Physical Page Number"), um Adresse zu konstruieren
  - wenn (valid Bit = 0), dann ist die Seite auf der Festplatte, d.h. Seitenfehler

Seiten-Tabellenoperation

Schutzüberprüfung:

- Zugriffsrechtefeld gibt Zugriffserlaubnis an
  - z.B. read-only, read-write, execute-only
  - typischerweise werden zahlreiche Schutzmodi unterstützt (z.B. Kernel gegen User)
- Schutzrechteverletzung: wenn Benutzer nicht die nötigen Rechte hat

Integration von VM und Cache



Die meisten Caches werden “physikalisch adressiert”

- Zugriff über physikalische Adressen
- gestattet mehreren Prozessen, gleichzeitig Blöcke im Cache zu haben
- gesttatet mehreren Prozessen, die Seiten zu teilen
- Cache muss sich nicht mit Schutzproblemen befassen
  - Zugriffsrechte werden als Teil der Adressumsetzung überprüft

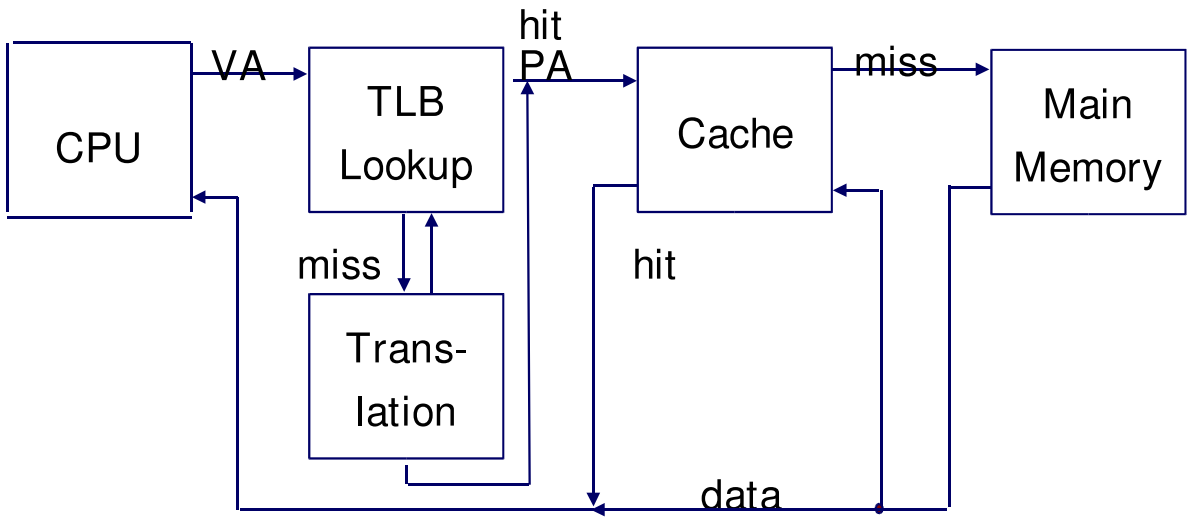
Die Adressumsetzung wird vor dem Cache “Lookup” durchgeführt

- könnte selbst einen Speicherzugriff (auf den PTE) beinhalten
- Seiten-Tabelleneinträge auch “gecacht” werden

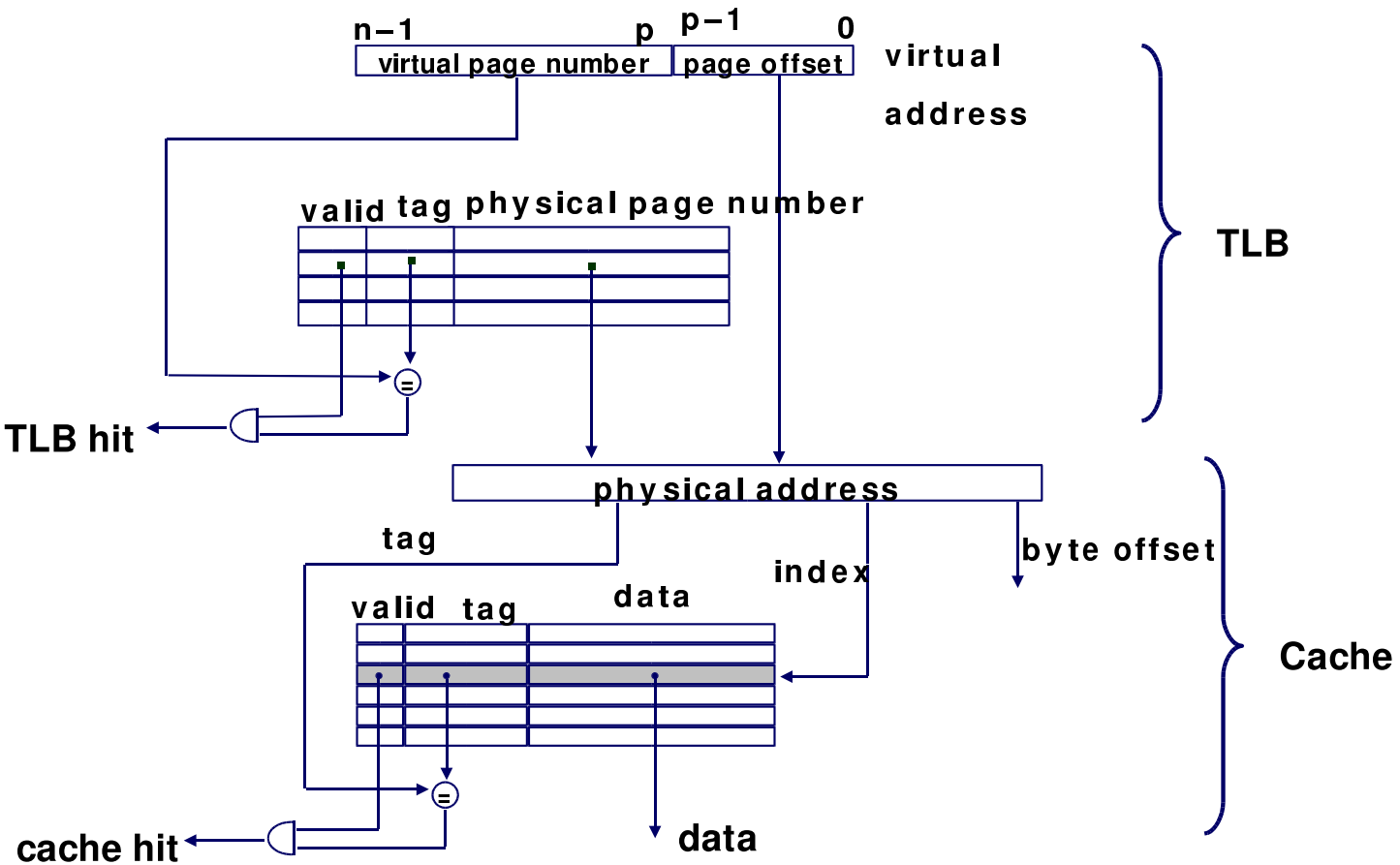
Beschleunigung der Umsetzung mit einem TLB

"Translation Lookaside Buffer" (TLB)

- Kleiner Hardware Cache in der MMU (Memory Management Unit)
- Bildet virtuelle Seitenzahlen auf physikalische ab
- Enthält die kompletten Seiten-Tabelleneinträge für wenige Seiten



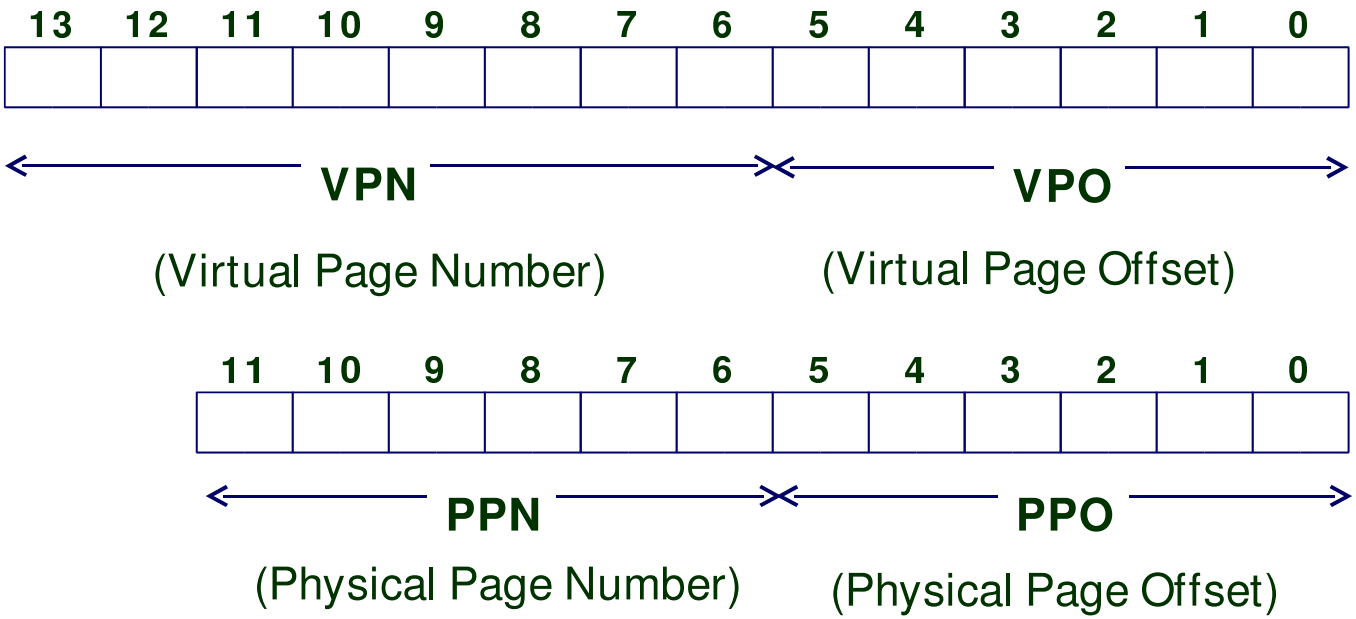
Adressumsetzung mit einem TLB



Beispiel für einfaches Speichersystem

Adressierung

- 14-Bit virtuelle Adresse
- 12-Bit physikalische Adresse
- Seitengröße = 64 Bytes



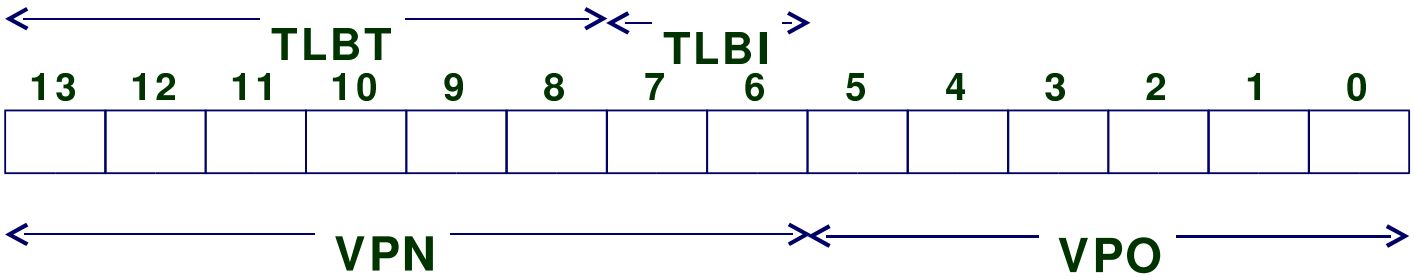
Seiten-Tabelle des einfachen Speichersystems

- Nur die ersten 16 Einträge werden gezeigt

VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	–	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	–	0
04	–	0	0C	–	0
05	16	1	0D	2D	1
06	–	0	0E	11	1
07	–	0	0F	0D	1

Einfaches Speichersystem: TLB

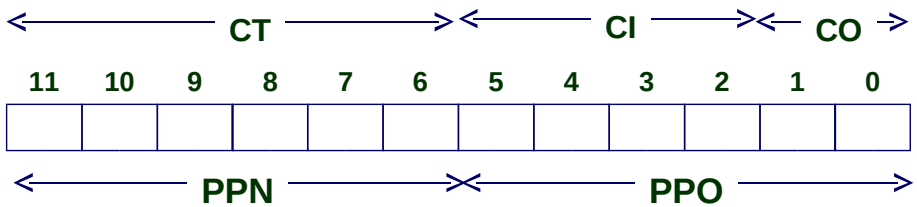
- 16 Einträge
- 4-fach assoziativ



Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Einfaches Speichersystem: Cache

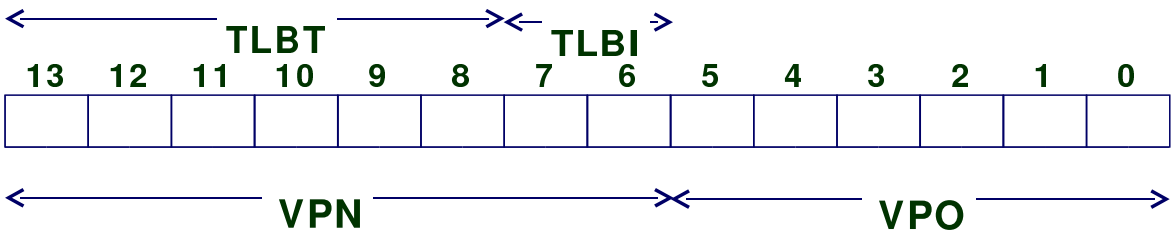
- 16 Zeilen
- 4-byte pro Zeile (line size)
- Direkt abgebildet



Idx	Tag	Valid	B0	B1	B2	B3	Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11	8	24	1	3A	00	51	89
1	15	0	–	–	–	–	9	2D	0	–	–	–	–
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	–	–	–	–	B	0B	0	–	–	–	–
4	32	1	43	6D	8F	09	C	12	0	–	–	–	–
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	–	–	–	–	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	–	–	–	–

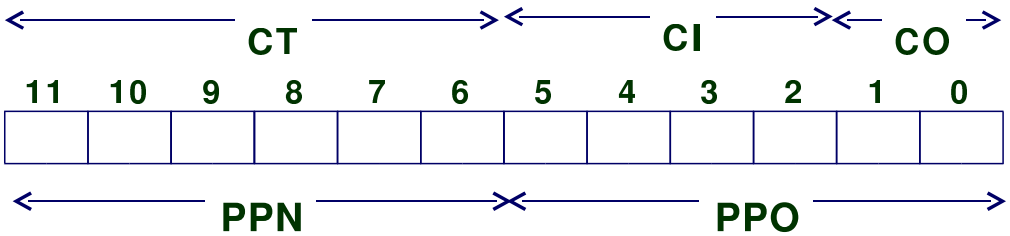
Adressumsetzungsbeispiel Nr. 1

Virtual Address `0x03D4`



VPN ___ TLBI ___ TLBT ____ TLB Hit? __ Page Fault? __ PPN: ____

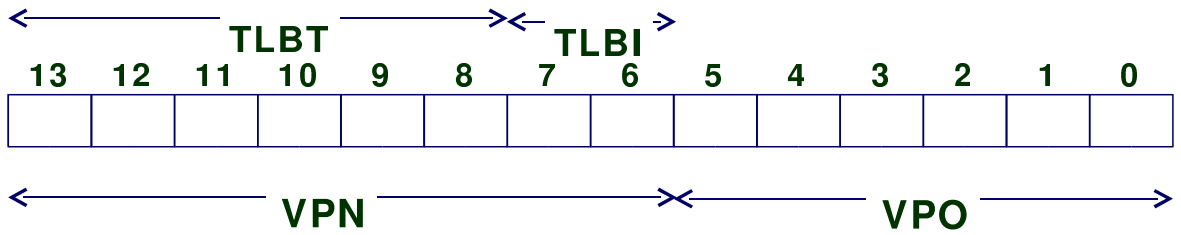
Physical Address



Offset ___ CI___ CT ____ Hit? __ Byte: ____

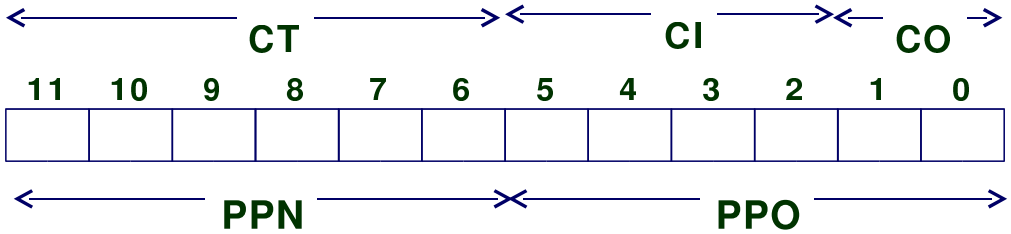
Adressumsetzungsbeispiel Nr. 2

Virtual Address `0x0B8F`



VPN ___ TLBI ___ TLBT ____ TLB Hit? __ Page Fault? __ PPN: ____

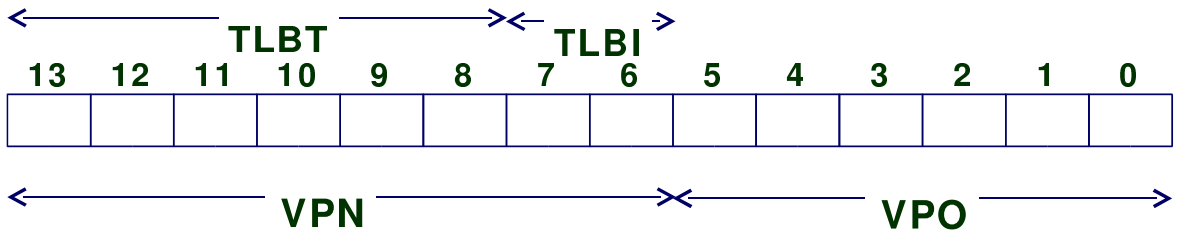
Physical Address



Offset ___ CI___ CT ____ Hit? __ Byte: ____

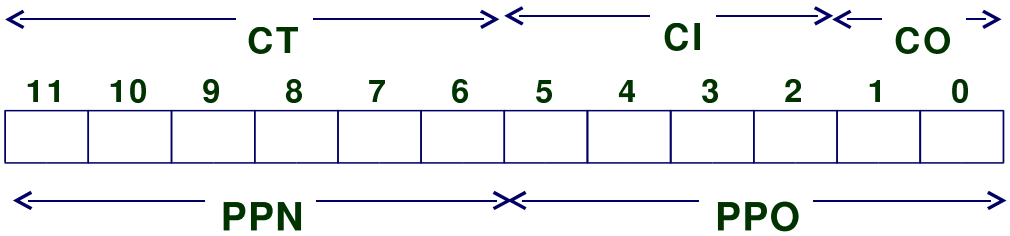
Adressumsetzungsbeispiel Nr. 3

Virtual Address 0x0040



VPN ___ TLBI ___ TLBT ____ TLB Hit? __ Page Fault? __ PPN: ____

Physical Address

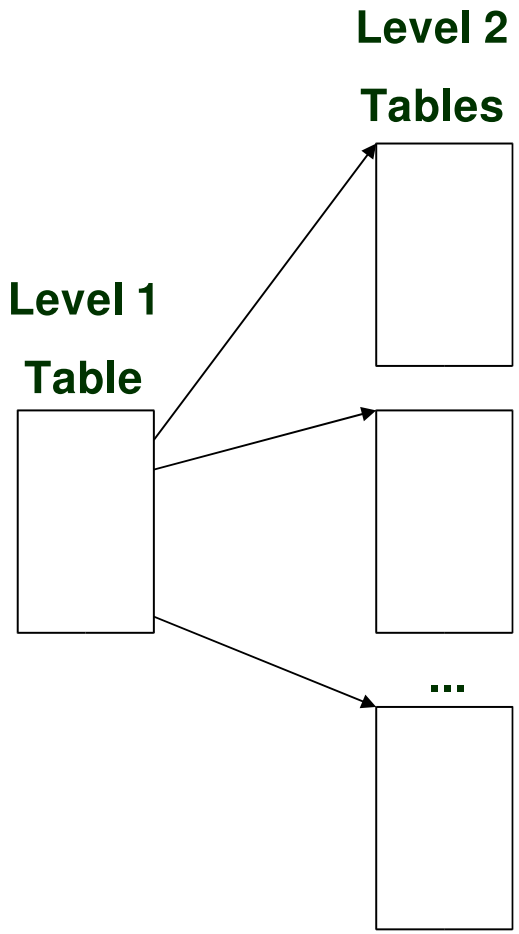


Offset ___ CI___ CT ____ Hit? __ Byte: ____

Multi-Ebenen Seiten-Tabellen

Gegeben:

- 4KB (2^{12}) Seitengröße
- 32-Bit Adressraum
- 4-Byte PTE ("Page Table Entry" = Seitentabelleneintrag)



Multi-Ebenen Seiten-Tabellen (cont.)

Problem:

- Würde eine 4 MB Seiten-Tabelle benötigen!
  - 2^{20} Bytes

Übliche Lösung

- Multi-Ebenen Seiten-Tabellen
- z.B. zweistufige Tabelle (Pentium P6)
  - Ebene-1 Tabelle: 1024 Einträge, wovon jeder auf eine Ebene-2 Seiten-Tabelle zeigt
  - Ebene-2 Tabelle: 1024 Einträge, wovon jeder auf eine Seite zeigt

Zusammenfassung der virtuellen Speicher

Aus Sicht des Programmierers:

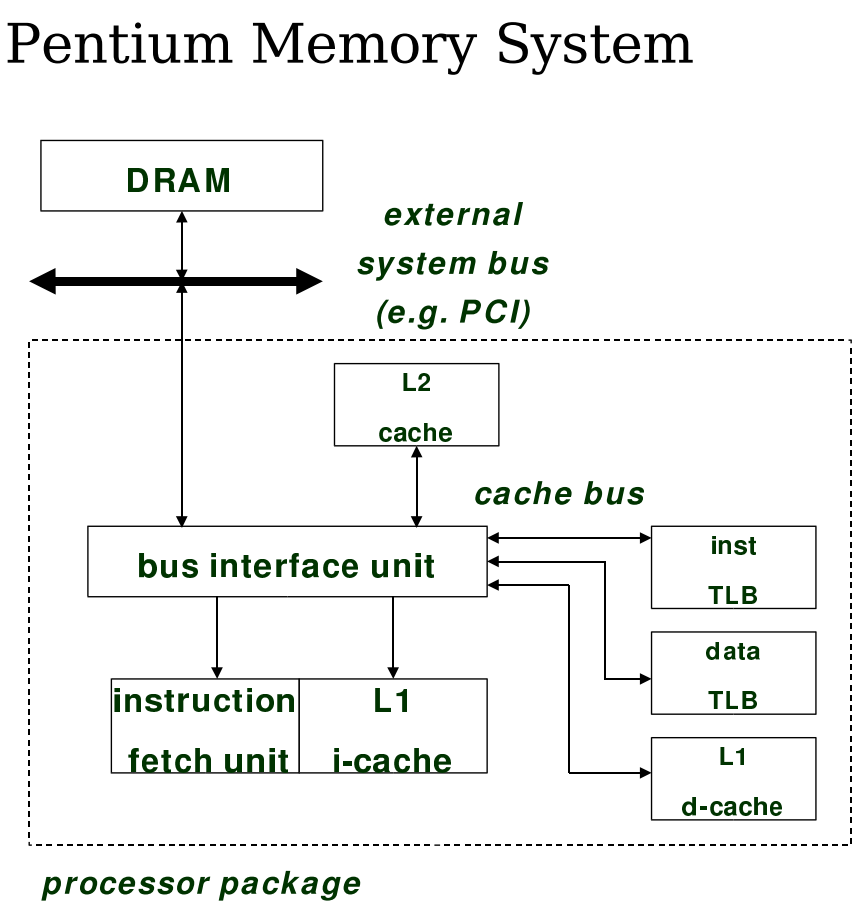
- großer “flacher” Adressraum
  - kann große Blöcke benachbarter Adressen zuteilen
- Prozessor “besitzt” die gesamte Maschine
  - hat privaten Adressraum
  - bleibt unberührt vom Verhalten anderer Prozesse

Virtuellen Speicher: Hauptthemen (Forts.)

Aus Sicht des Systems:

- Virtueller Adressraum des Benutzers wird durch Abbildung auf Seitenbereich erzeugt
  - muss nicht fortlaufend sein
  - wird dynamisch zugeteilt
  - erzwingt Schutz bei Adressumsetzung
- OS kann viele Prozesse gleichzeitig verwalten
  - ständiges Wechseln zwischen Prozessen
  - vor allem wenn auf Ressourcen gewartet werden muss, z.B. Festplatten I/O um Seitenfehler zu beheben

Das Speichersystem von Pentium und Linux



32 bit address space

4 KB page size

L1, L2, and TLBs

- 4-way set associative

inst TLB

- 32 entries
- 8 sets

data TLB

- 64 entries
- 16 sets

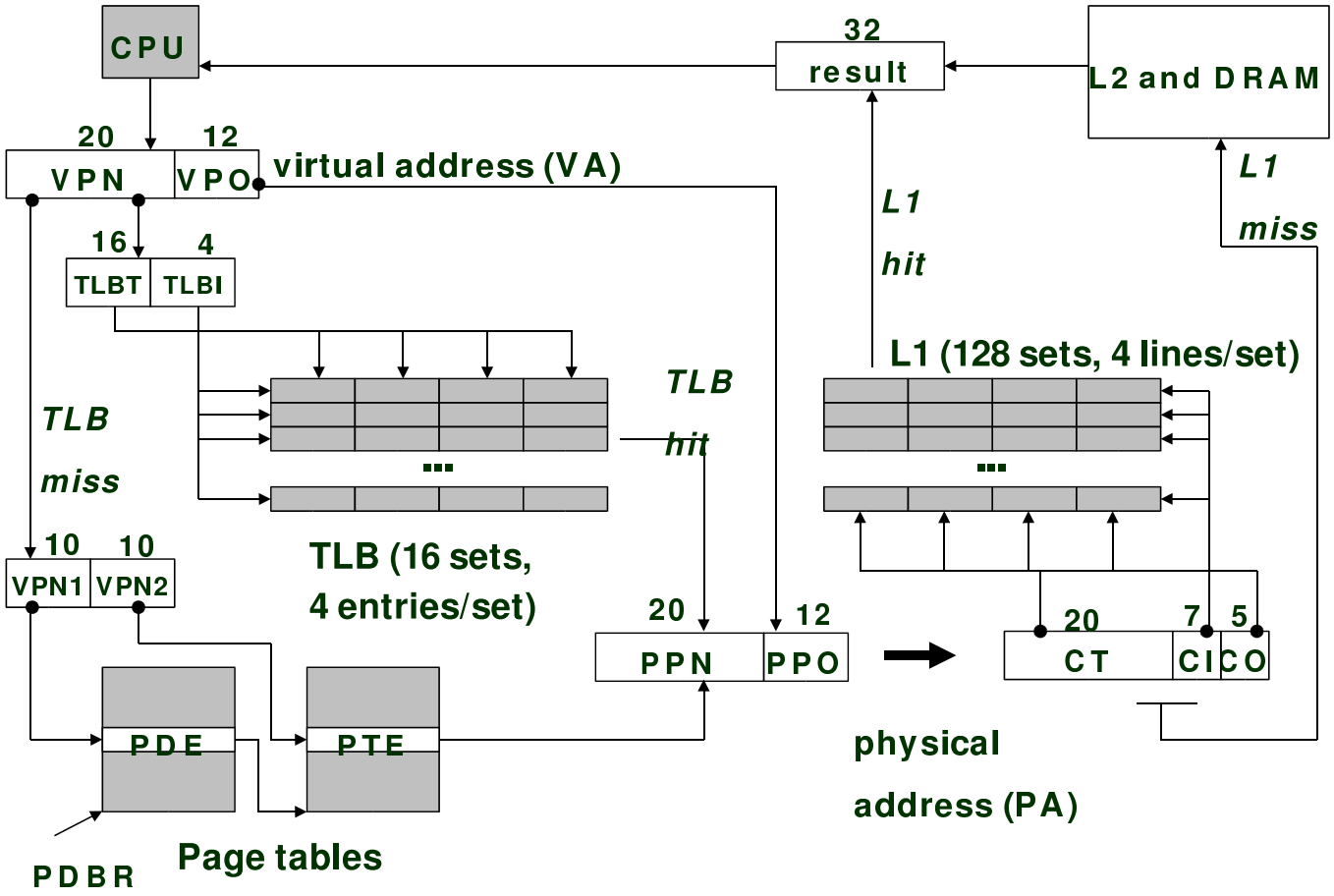
L1 i-cache and d-cache

- 16 KB
- 32 B line size
- 128 sets

L2 cache

- unified
- 128 KB -- 2 MB

Adressumsetzung beim Pentium



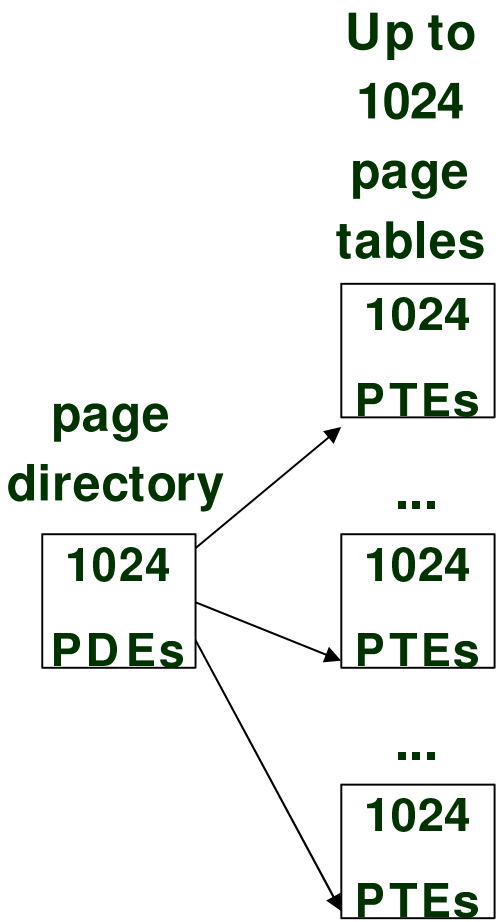
Zweistufige Seiten-Tabellenstruktur

Seiten-Verzeichnis (Page Directory):

- 1024 4-Byte Einträge ("Page Directory Entries PDEs"), die auf Seitentabellen verweisen
- Ein Seiten-Verzeichnis pro Prozess
- Seiten-Verzeichnis muss im Speicher liegen, während der zugehörige Prozess läuft
- immer über PDBR (Page Directory Base Register) zugreifbar

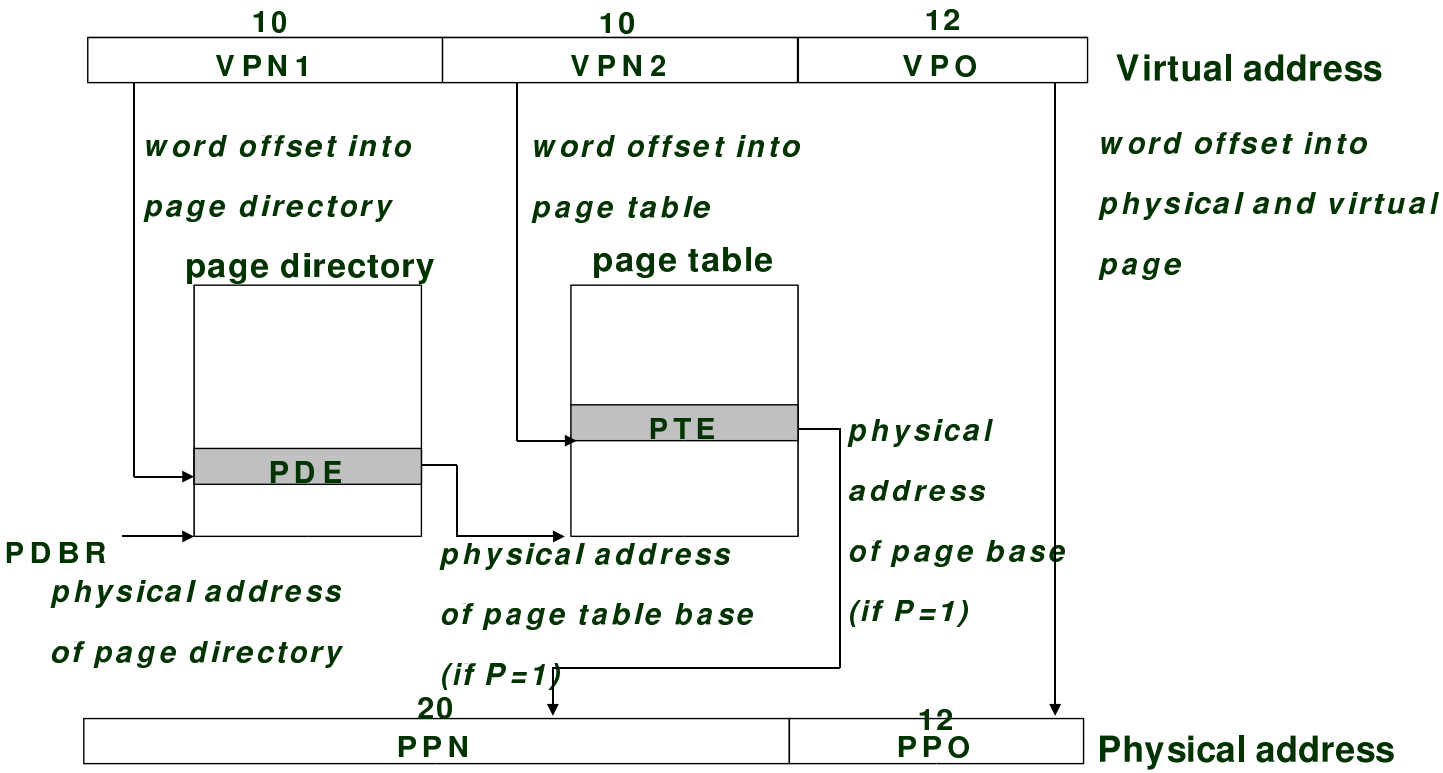
Seiten-Tabellen:

- 1024 4-byte Einträge ("Page Table Entries PTEs"), die auf Seiten verweisen
- Seiten-Tabellen können ein- und ausgelagert werden ("page in, page out")

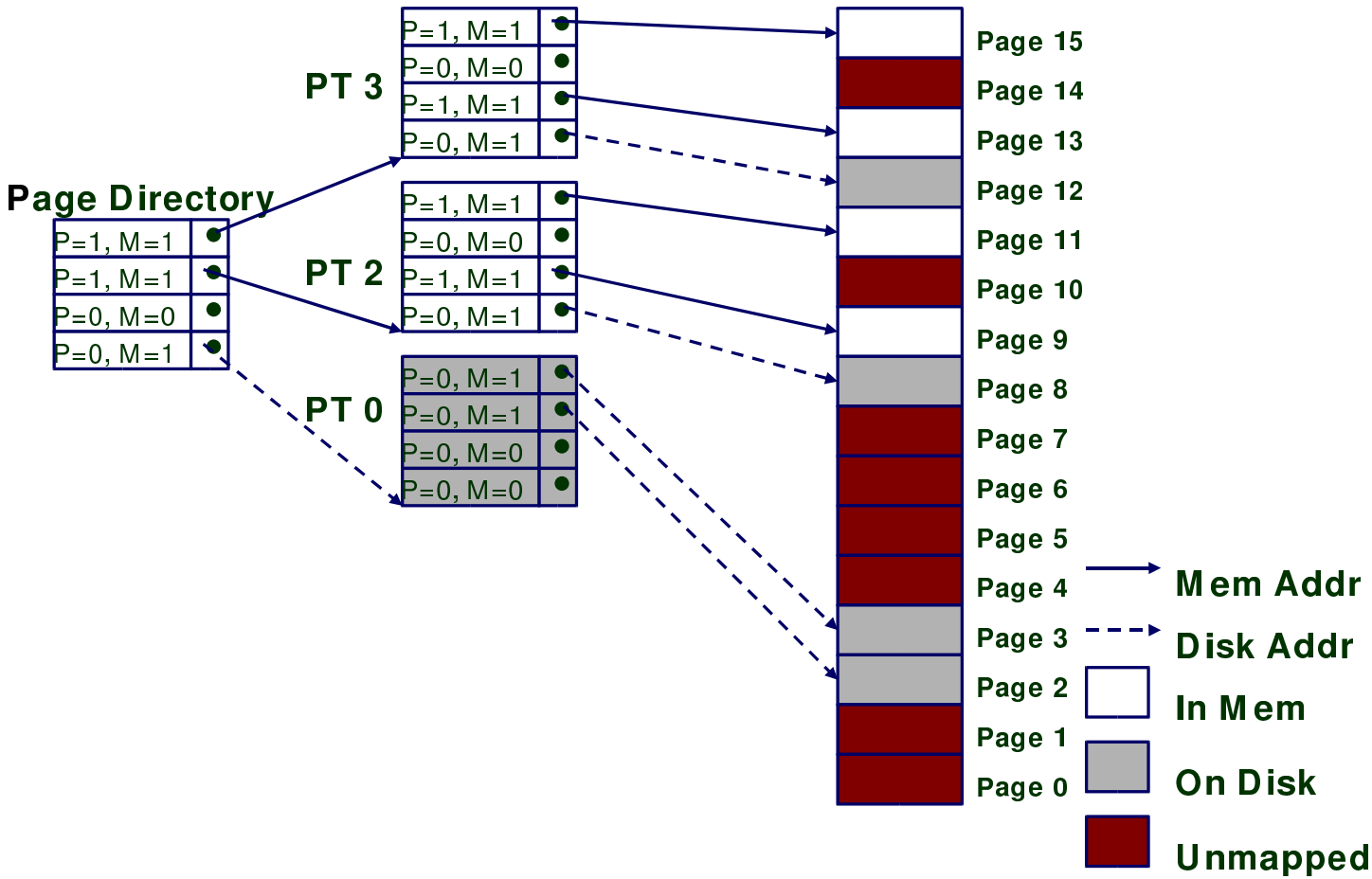


Umsetzung von virtuellen auf physikalische Adressen

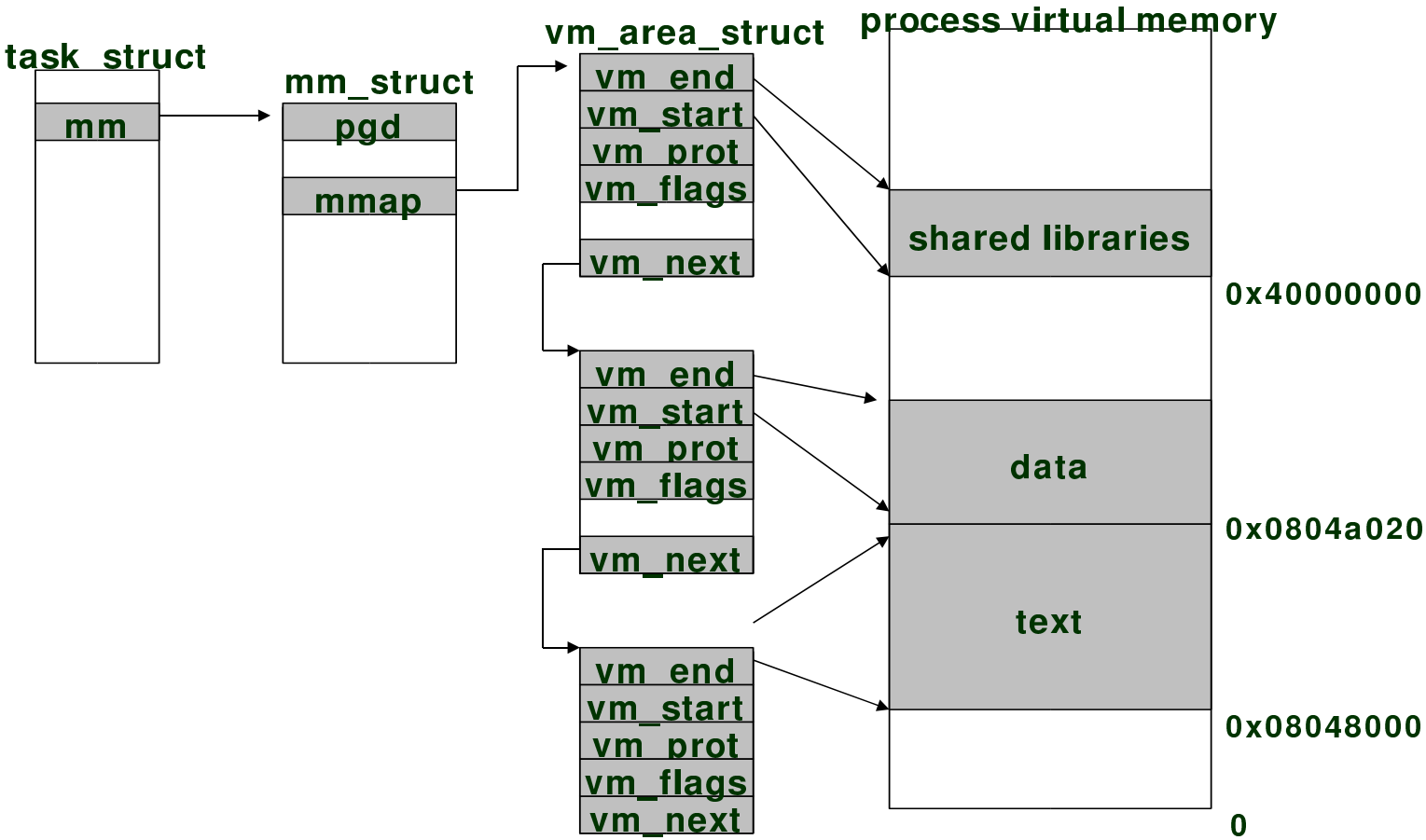
mit den Seiten-Tabellen des Pentiums



Darstellung des virtuellen Adressraums



Linux organisiert VM als Menge von Bereichen



Zusammenfassung Speichersystem

Cache Speicher

- Dient nur zur Beschleunigung
- Verhalten unsichtbar für Anwendungsprogrammierer und OS
- Komplett in Hardware implementiert

Virtueller Speicher

- Ermöglicht viele Funktionen des OS
  - Prozesse erzeugen (neue und abgeleitete ("forking"))
  - Taskwechsel
  - Schutzmechanismen
- Implementierung mit Hardware und Software
  - Software verwaltet die Tabellen und Zuteilungen
  - Hardwarezugriff auf die Tabellen
  - Hardware-Caching der Einträge (TLB)

Ergänzende Literatur

Zur Rechnerarchitektur (Teil 2, 8. Termin):

[1] Randal E. Bryant and David O'Hallaron.
Computer systems.
pages 690–721. Pearson Education, Inc., New Jersey, 2003.

RAID: Motivation

Amdahl's Gesetz:

langsamste Komponente behindert Leistungssteigerungen

→ ausgewogenes Verhältnis CPU–Speicher–I/O nötig
→ CPU und Speicher skalieren mit der Halbleitertechnologie
→ aber wie kann die I/O-Leistung gesteigert werden?

RAID, "redundant array of inexpensive disks":

- Grundidee: viele kleine PC-Festplatten statt einer großen
- bedingt durch damalige (1985) Festplattentechnologie: Großrechner-Festplatten vs. PC-Festplatten
- Zuverlässigkeit durch redundante Platten
- Wiederherstellung der Daten nach Plattenausfall
- ursprünglich: "independent disks"

Disks: RAID

"redundant array of inexpensive disks"

- bahnbrechende Untersuchung von Festplatten-Perormance
- ursprüngliche Analyse von Großrechner- und PC-Festplatten
- Ersetzen weniger großer durch viele kleine Festplatten
- Zuverlässigkeit des Gesamtsystems?

- diverse RAID-Varianten (Level)
- unterschiedliche Anzahl von Platten
- Strategien zur Verwendung von Nutz- und Reserveplatten
- Ausfallsicherheit, Hot-Plugging
- Optimierung auf Schreib- und/oder Leseperformance
- vielfache Anwendungen

- möglichst das Original lesen!

(Patterson, Gibson, Katz: UCB report CS-98-391)

RAID: Ausgangsbasis

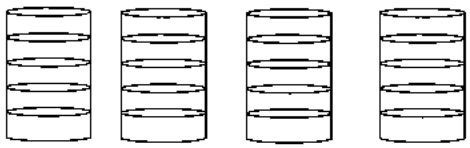
Characteristics	*IBM 3380*	*Fujitsu M2361A*	*Conners CP3100*	*3380 v. CP3100*	*2361 v. CP3100*
				(>1 means 3100 better)	
Disk diameter (inches)	14	10.5	3.5	4	3
Formatted Data Capacity (MB)	7500	600	100	.01	.2
Price/MB(controller incl.)	$18-$10	$20-$17	$10-$7	1-2.5	1.7-3
MTTF Rated (hours)	30,000	20,000	30,000	1	1.5
MTTF in practice (hours)	100,000	?	?	?	?
No. Actuators	4	1	1	.2	1
Maximum I/O's/second/Actuator	50	40	30	.6	.8
Typical I/O's/second/Actuator	30	24	20	.7	.8
Maximum I/O's/second/box	200	40	30	.2	.8
Typical I/O's/second/box	120	24	20	.2	.8
Transfer Rate (MB/sec)	3	2.5	1	.3	.4
Power/box (W)	6,600	640	10†	660	64
Volume (cu. ft.)	24	3.4	.03	800	11

Table I. *Comparison of IBM 3380 disk model AK4 for mainframe computers, the Fujitsu M2361A "Super Eagle" disk for minicomputers, and the Conners Peripherals CP 3100 disk for personal computers. By "Maximum I/O's/second" we mean the maximum number of average seeks and average rotates for a single sector access. Cost and reliability information on the 3380 comes from widespread experience [IBM 87] [Gawlick87] and the information on the Fujitsu from the manual [Fujitsu 87], while some numbers on the new CP3100 are based on speculation. The price per megabyte is given as a range to allow for different prices for volume discount and different mark-up practices of the vendors.*

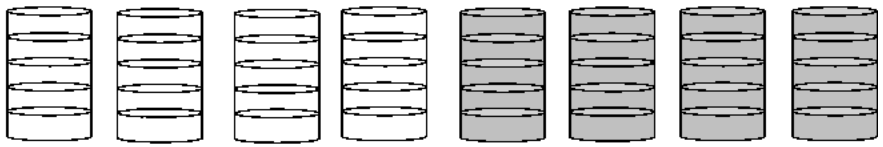
RAID: 7 Typen

Man unterscheidet mehrere Ebenen:

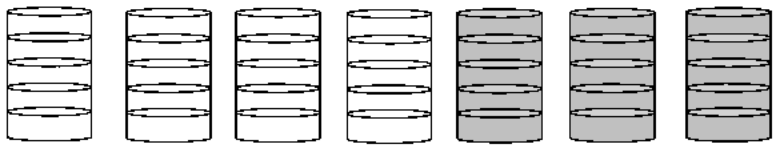
RAID 0: Keine Redundanz



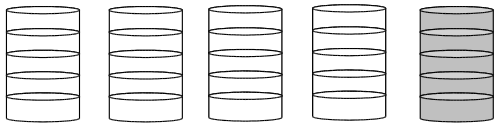
RAID 1: Spiegelung von Einzelplatten



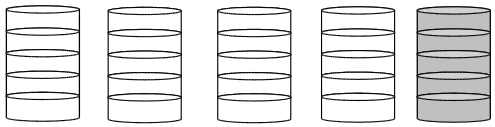
RAID 2: Einsatz fehlerkorrigierender Bitcodes



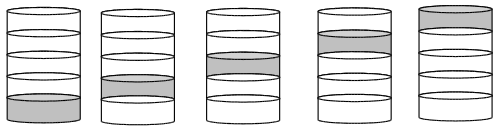
RAID 3: Bit-Parität



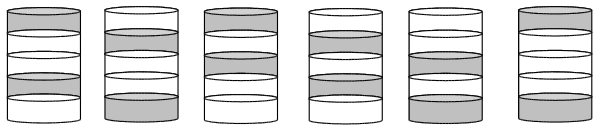
RAID 4: Block-Parität



RAID 5: Rotierende Block-Parität



RAID 6: Doppelte Redundanz



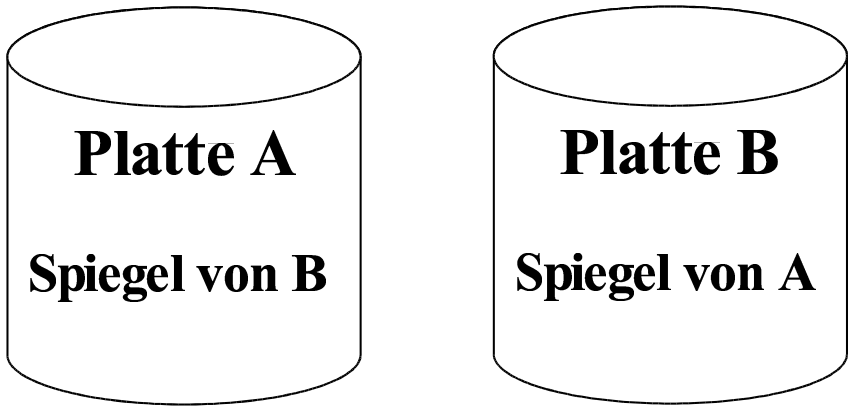
RAID 0

- Die Datenblöcke werden gleichmäßig auf mehrere unabhängige Platten verteilt.
- Hierdurch erreicht man Lastglättung.

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

RAID 1

- Jeder Platteninhalt ist zweimal vorhanden.
- Bei Ausfall einer Platte nutzt man das Duplikat und erstellt eine neue Kopie auf einer Reserve-platte.
- Hierbei muß jeder Schreibzugriff doppelt ausgeführt werden.
- Lesezugriffe werden nur einmal ausgeführt.



RAID 2

- Verwendung eines fehlerkorrigierenden Bitcodes
- Ein einfacher ist der (7, 4) Hamming-Code.
  - Die Originalbit und die Redundanzbit werden auf verschiedene Platten verteilt.
  - Ein Datenzugriff erfordert viele synchronisierte Plattenoperationen.
  - Zu einem Original-Quadrupel gehören drei Korrekturbits.
  - Es werden also also sieben Platten benötigt .
- Insgesamt erscheint der Aufwand nicht gerechtfertigt.

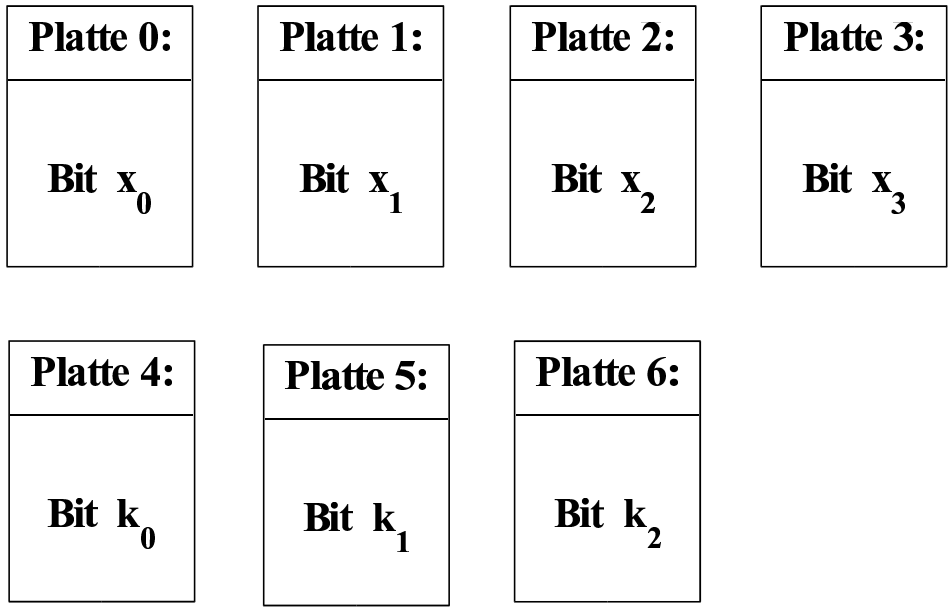
RAID 2 (cont.)

Gleichungssystem:

$k2 = x2 + x1 + x0 \pmod 2$

$k1 = x3 + x1 + x0 \pmod 2$

$k0 = x3 + x2 + x0 \pmod 2$



RAID 3

- Schutz gegen Totalausfall einer Platte
- Einsatz Paritätsplatte
- Beispiel:
  - Verteilung der Originaldaten auf vier Platten
  - zusätzliche Paritätsplatte
    - Bildung der Modulo-2-Summe über entsprechende Bits der Originaldaten Formel: $p = x_0 \; xor \; x_1 \; xor \; x_2 \; xor \; x_3$
  - Ausfall einer der Platten 0 bis 4 (hier: Platte 1)
    - Restaurierung des Inhalts der ausgefallenen Platte mittels Modulo-2-Addition über verbleibende Bits und Paritätsplatte
      Formel: $x_1 = x_0 \; xor \; x_2 \; xor \; x_3 \; xor \; p$

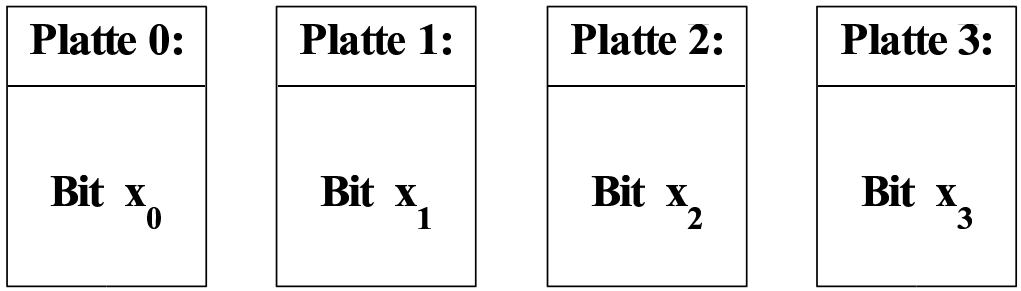
RAID 3 (cont.)

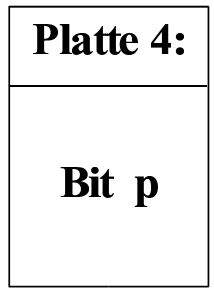
$$X_0 \oplus X_1 \oplus X_2 \oplus X_3 = p$$

X_0	X_1	X_2	X_3	p
1	1	0	1	1
0	1	1	0	0
0	0	0	0	0
1	1	0	0	0

$$X_1 = X_0 \oplus X_2 \oplus X_3 \oplus p$$

X_1	X_0	X_2	X_3	p
1	1	0	1	1
1	0	1	0	0
0	0	0	0	0
1	1	0	0	0





RAID 4, 5, 6

RAID 4:
Dies ist ähnlich zu RAID 3. Die Paritätsinformation nutzt man nur bei Totalausfall einer Platte. Daher führen nur Schreibzugriffe zu einer erhöhten Last. Jeder Schreibzugriff erfordert zwei Lese- und zwei Schreibzugriffe.

RAID 5:
Um den Flaschenhals der Paritätsplatte zu mildern, verteilt man die Redundanzinformation zyklisch über alle Platten.

RAID 6:
Hier erhöht man die Datensicherheit gegenüber RAID 5 durch Bildung mehrerer unabhängiger Schutzinformationen.

Optisches Speichermedium CD-ROM

(CD-ROM = Compact Disc Read Only Memory)

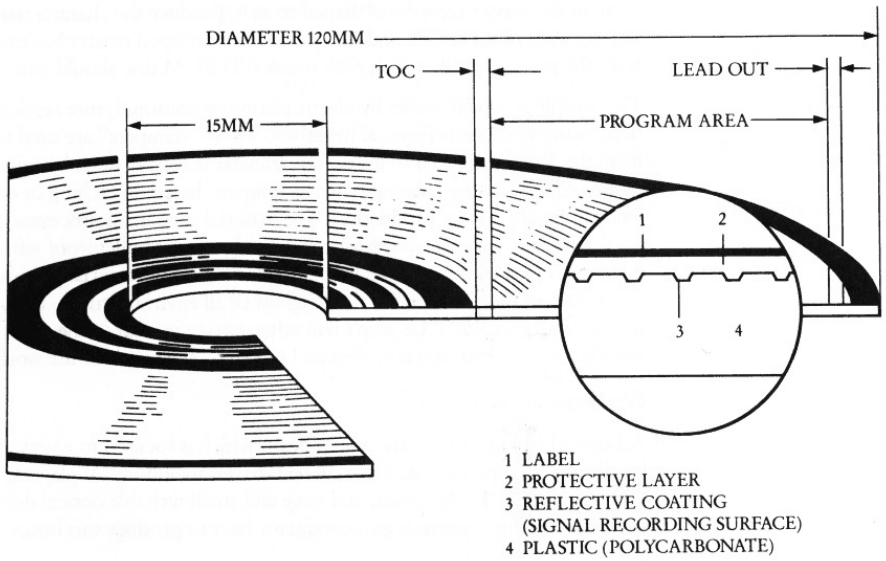
- Weiterentwicklung der Audio-CD (1982)
- Aufzeichnungspirale der Daten ca. 5 km Länge
- Speicherkapazität:
  - 74-Minuten-CD-DA: ca. 783 Millionen Nutzbyte
  - CD-ROM: ca. 650 MB
- Standards:
  - Audio-CD und CD-ROM sind IEC 908 (“red book”)
  - ISO/IEC 10149 (“yellow book”)

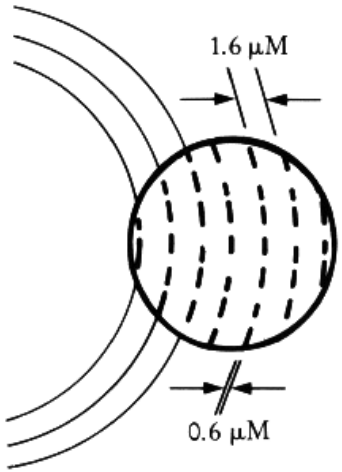
Aufzeichnungsverfahren für die Rohdaten

- 24 8-Bit-Datenbyte werden hinzugefügt:
  - 4 Q-Paritäts-Byte
  - 4 P-Paritäts-Byte
  - ein Subcode-Byte
- jedes Byte wird durch 17 Kanalbit codiert
- geschickte Verschachtelung der Datenbyte
- Wahrscheinlichkeit von 10^{-8} für nicht korrigierbare Lesefehler auf der Audio-Ebene

- fur Daten ist dies zu gering
  - 276 Byte Fehlerkorrekturcode auf 2.048 Datenbyte zusätzlich $\longrightarrow$ Wahrscheinlichkeit von 10^{-12} für nicht korrigierbare Lesefehler auf Daten-Ebene

CD: Prinzip

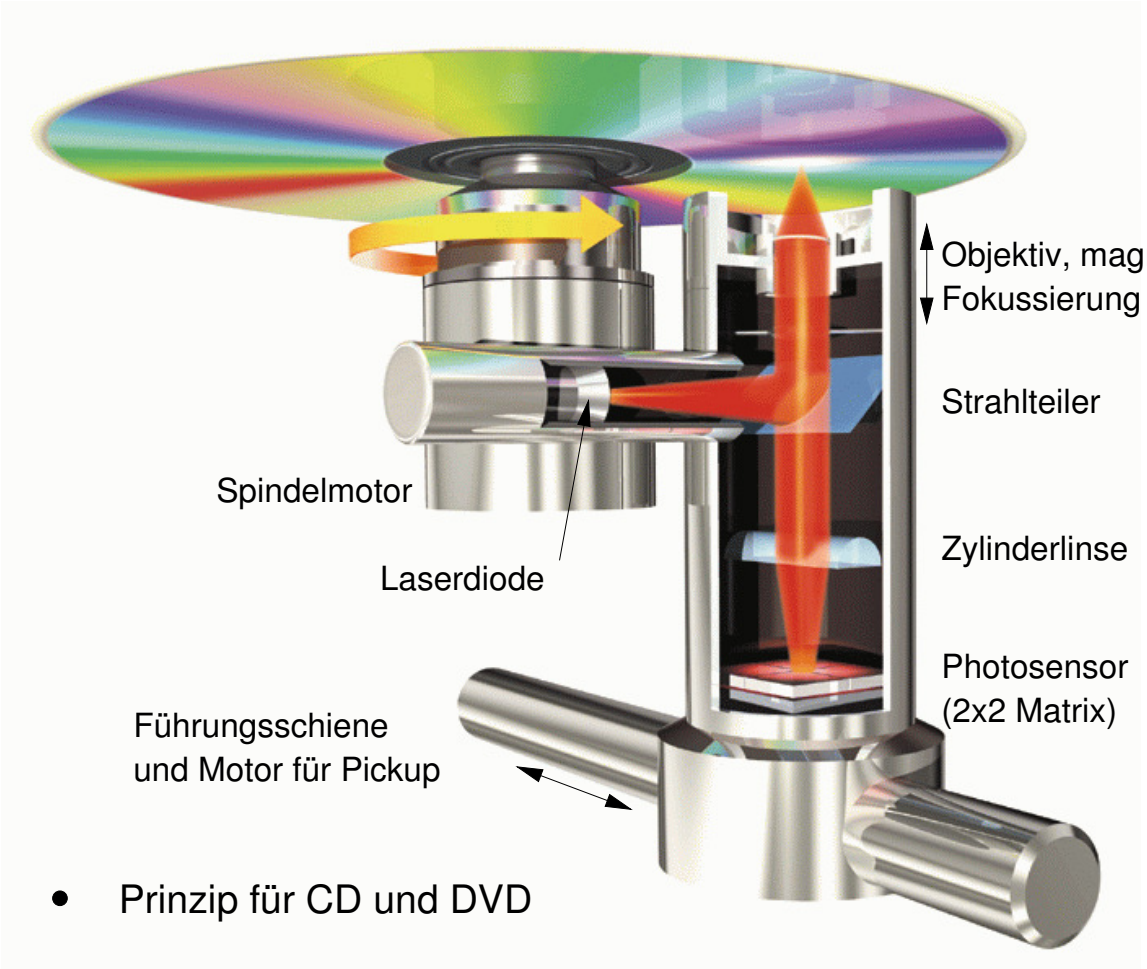




- Polycarbonatträger, 12cm Durchmesser
- eingeprägte Vertiefungen ("pits") bilden die Daten
- spiralförmige Datenspur, 1.6µm Abstand, ca. 16000 Windungen
- Fertigungsmängel fest eingeplant => leistungsfähige Fehlerkorrektur

(CD–ROM – the new papyrus)

CD: Aufbau eines Players



- Prinzip für CD und DVD

Einige Merkmale der Audio-CD:

maximale Spieldauer:	74 Minuten, 33 Sekunden
Lesegeschwindigkeit:	1,2 m/s - 1,4 m/s
Spurabstand:	1,6 μm
Durchmesser:	120 mm
Dicke:	1,2 mm
Zentrumsdurchmesser:	15 mm
Aufzeichnungsbereich:	46 mm - 117 mm
Datenbereich:	50 mm - 116 mm
Minimale "PitLänge:	0,833 μm (1,2 m/s) bis 0,972 μm (1,4 m/s)
Maximale "PitLänge:	3,05 μm (1,2 m/s) bis 3,56 μm (1,4 m/s)
"PitTiefe:	0,11 μm
"PitBreite:	0,5 μm

Einige Merkmale der Audio-CD: (cont.)

Standard Wellenlänge:	780 nm
Brechungsindex des Materials:	1,55
Zahl der Kanäle:	2 (4 zulässig)
Quantelung:	16 Bit linear
Abtastfrequenz:	44.100 Hz
Kanal-Bit-Rate:	4,3218 Mbit/s
Daten-Bit-Rate:	2,0338 Mbit/s
Verhältnis Datenbit zu Kanalbit:	8 : 17
Fehlerkorrekturcode:	Verschachtelter Reed-Solomon Code (25% Redundanz)
Modulationssystem:	Eight to fourteen Modulation (EFM)

Ein Audio-Rahmen:

Synchronisation	24	Bit
Subcode	14	Bit
6 * 2 * 2 * 14 Datenbit	336	Bit
8 * 14 Paritätsbit	112	Bit
34 * 3 Pufferbit	102	Bit
Summe:	588	Bit

Ein Audio-Rahmen: Bemerkungen

- 192 Datenbit werden in 588 Kanalbit codiert.
- Das Subcode-Byte enthält 8 Bit, die den Subkanälen P, Q, R, S, T, U, V, W zugeordnet sind. Nur die Subkanäle P und Q werden von der Audio-disc genutzt.
- Die Fehlerbehandlung erfolgt in drei Schritten:
  - Zunächst wird versucht, die fehlerhaften Byte zu korrigieren.
  - Unkorrigierbare Bytewerte versucht man durch Interpolation zu gewinnen.
  - Läßt sich auch die Interpolation nicht durchführen, dann wird der entsprechende Musikteil durch Stille ersetzt.

Beispiel zur Codeverschränkung:

- Länge der Nachricht: 103 Zeichen
- ohne Codeverschränkung gespeichert:
  - Auslesen der teilweise zerstörten Nachricht:
    Ein Buch mit dem Titel “Das Geheimnis meiner XXXXXXXXX”
    kann nichts anderes als leere Seiten enthalten.
- Nachricht gespeichert mit Verschränkungszahl 11:
  - Gespeicherte Zeichenfolge:
    E Itnsmi aeialet etSnhnlilhai ene so ann di iBXXXXXXXXXmul
    eDtermeincenaer ncihe” sneMT hs r .siiet mekG
  - Auslesen der teilweise zerstörten Nachricht:
    Ein Buch mit dXm Titel XDas Geheimnis me inXr MillioXen”
    kann Xichts anXeres alsX leere SeiXen enthaXten.
- ursprüngliche Nachricht jetzt rekonstruierbar.

Bemerkung: Mittels Codeverschränkung werden Bündelfehler in Einzelfehler transformiert.

Konversion von 8-Bit-Byte zu “pits and lands”

- Daten:
  11101000 11100010 10111010 11101011
- Umsetzung in 14-Bit-Darstellung (Eight-to-Fourteen-Modulation):
  10000100000010 10000100010010 10010000001001 00001001000010
- Einfügen von Pufferbits:
  10000100000010 000 10000100010010 000 10010000001001 001
  00001001000010
- Bitfolge:
  10000100000010000100010010000100100000100100100001001000010
- “Pits and Lands”:

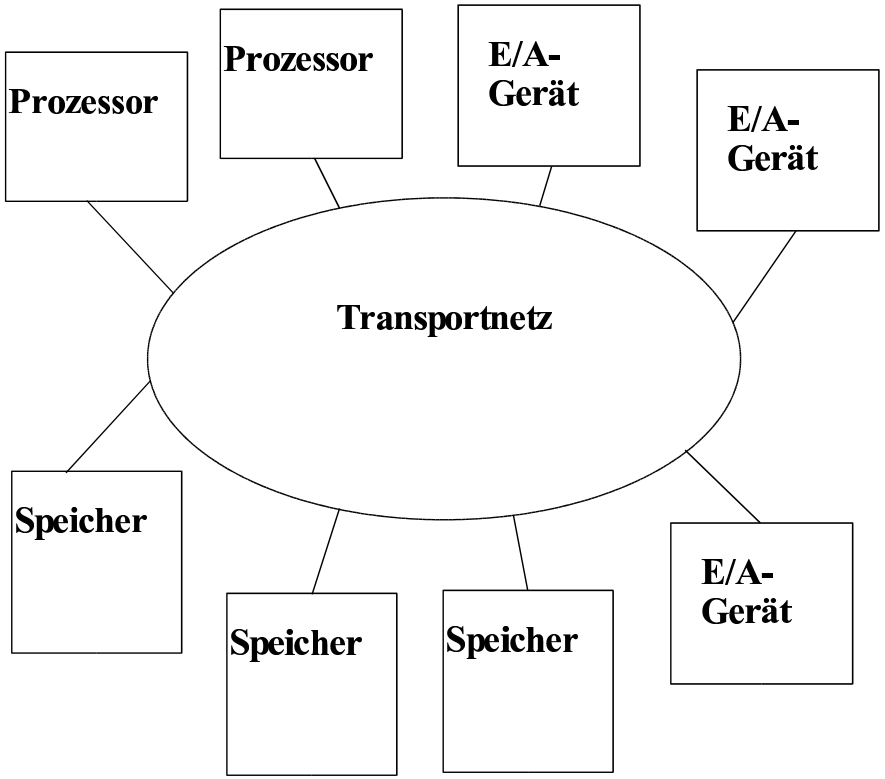
Bemerkung: Der Wechsel zwischen “pit” und “land” wird durch eine 1 codiert.

Gliederung

Gliederung (cont.)

I/O: Ein- und Ausgabe

Rechenanlage als Ansammlung von Einzelgeräten:

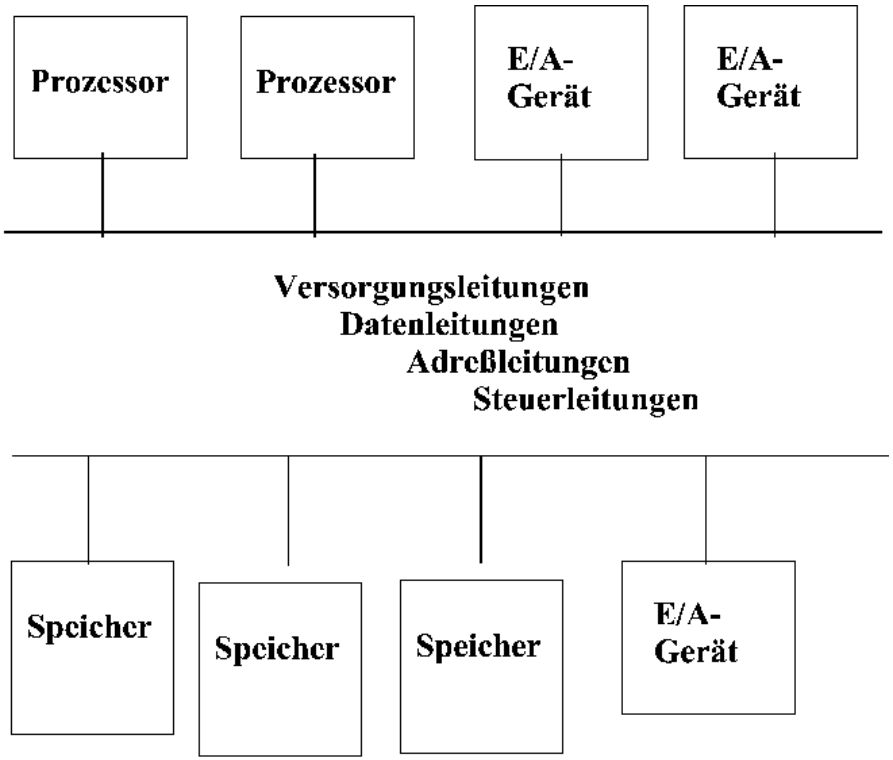


Bemerkung:

Der Datenaustausch zwischen zwei Geräten ist durch Protokolle zu regeln.

Busse

Ein einfaches Transportsystem ist der Bus.



Bemerkung:
Das Wort Bus ist eine Verkürzung des lateinischen Wortes omnibus. Es existieren viele Busstandards, zwei bekannte sind der PCI-Bus und der SCSI-Bus.

Kenngrößen eines Busses

- Verbindungsvielfalt:
  - Verbindung zwischen zwei Geräten,
  - Verbindung zwischen mehreren (≥ 3) Geräten.
- Nutzungsart:
  - Mehrfachnutzung von Leitungen,
  - dedizierte Leitungen.
- Bestimmung des temporären Busherrschers:
  - zentralisiertes Verfahren,
  - verteiltes Verfahren.
- Zeitsteuerung:
  - Taktsteuerung,
  - kein Zentraltakt (“asynchron”).

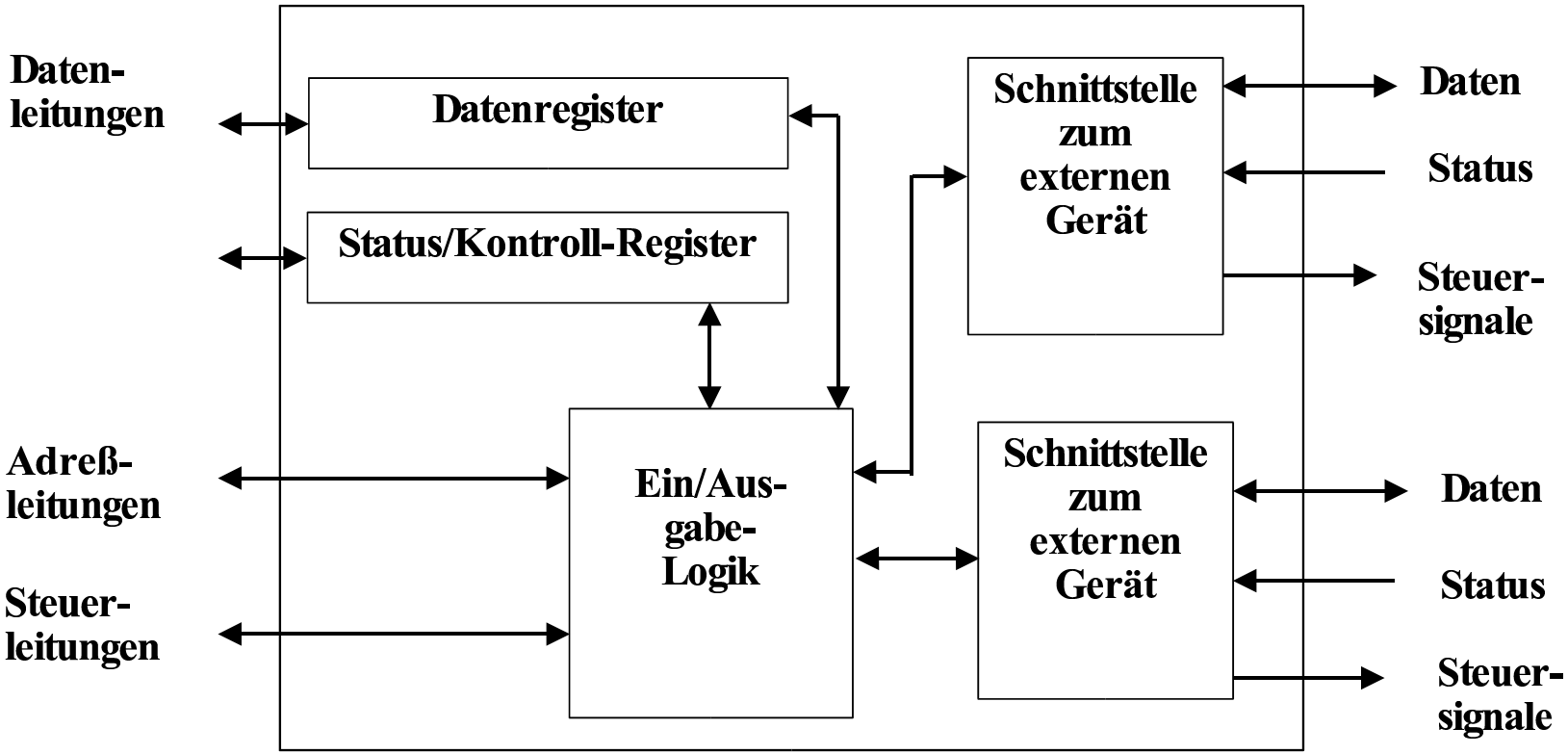
Kenngrößen eines Busses (cont.)

- Busbreite (Zahl der Leitungen für Teilbusse):
  - Daten,
  - Adresse.
- Daten-Transfer-Art:
  - “read”,
  - “write”,
  - “read-modify-write”,
  - “read-after-write”,
  - “block-transfer”.

Bemerkung: Ein Bus, der viele Einheiten verbindet, stellt einen Systemengpaß dar. Daher finden sich in einer Rechenanlage häufig mehrere Bussysteme.

Generisches E/A-Modul

Schnittstelle zum Systembus



Einige typische Datentransferraten:

Gigabit Ethernet	10^9	Bit/s
Bildschirm	10^9	Bit/s
640 x 480, 8 Bit Farbtiefe, 70 fps:	$1{,}7*10^8$	Bit/s
1024 x 768, 24 Bit Farbtiefe, 100 fps:	$1{,}9*10^9$	Bit/s
1600 x 1200, 24 Bit Farbtiefe, 100 fps:	$4{,}6*10^9$	Bit/s
Ultra DMA 133	10^9	Bit/s
Fast Ethernet	10^8	Bit/s
Festplatte	10^8	Bit/s
Ethernet	10^7	Bit/s
Laser Drucker	10^6	Bit/s
Modem	$5*10^4$	Bit/s
Maus	10^2	Bit/s
Tastatur	10^2	Bit/s

SCSI: Übersicht

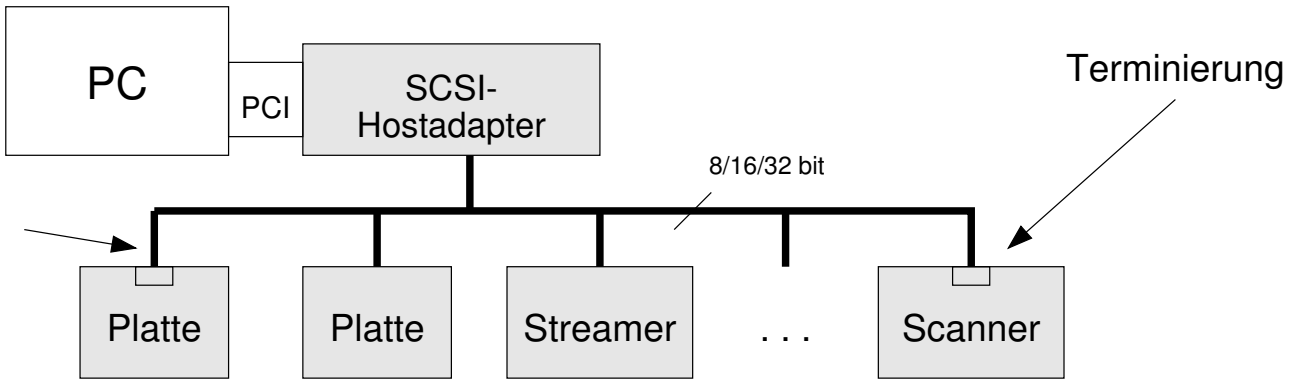
SCSI := Small Computer Systems Interface

- hervorgegangen aus "Shugart Associates SI"
- standardisiert als SCSI-I, SCSI-II, SCSI-III
- Einsatz in PCs (Server), Mac, Workstations

- universieller Bus für Peripheriegeräte ("Targets")
- Z. B. Bandlaufwerke, Scanner, Musiksynthesizer, ...
- 8-bi parallel (wide-SCSI mit 16-/32-bit)

- "Hostadapter" steuert Bus
- komplexe Befehle und Arbitrierung
- flexibler, aber auch teurer und komplexer als EIDE/ATAPI

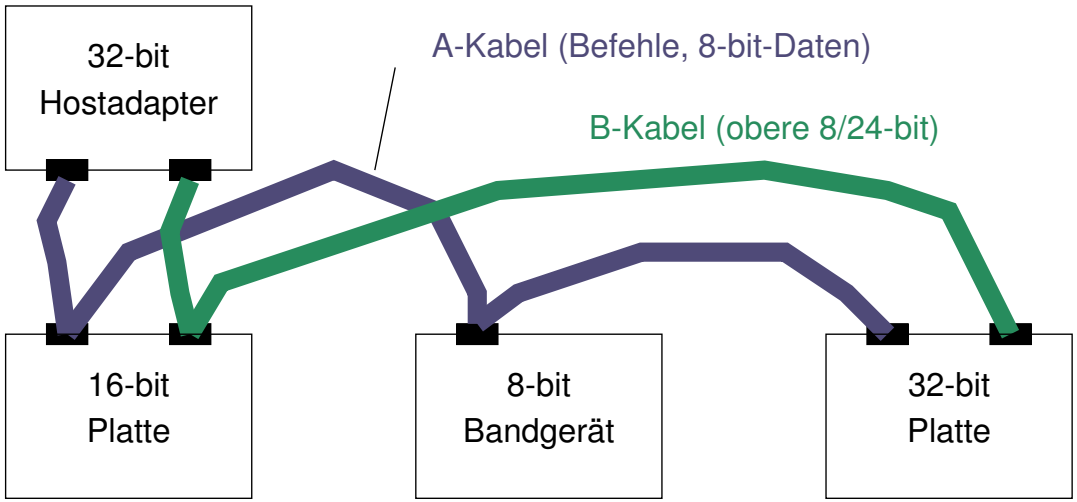
SCSI: Grundlagen



- Bus mit 8 Geräten (LUN 0..7), inklusive Controller
- Gerätenummer per Schalter eingestellt (nicht automatisch)
- komplexe Regeln zur Verkabelung (Abstände, Terminierung)
- parallele Datenübertragung, 8-bit oder (wide) 16/32-bit
- aufwendiges Busprotokoll mit Arbitrierung und split-transaktions
- Geräte handeln die jeweils bestmögliche Geschwindigkeit aus
- langsame Geräte stören schnelle Geräte nicht

SCSI: Varianten

- Befehlssätze: SCSI-1, SCSI-2, SCSI-3
- Busbreite: normal 8-bit, wide-SCSI 16-bit und 32-bit
- Bustiming: SCSI-1 bis 5 MB/s, Fast 10 MB/s Ultra 20 MB/s
- alle Kombinationen, z. B. U2W = Ultra-Wide SCSI-2
- alle Gerätevarianten miteinander kombinierbar
- insbesondere auch normale und wide-SCSI Geräte



Unterbrechungen

Durch eine Unterbrechung wird der Prozessor dem laufenden Programm entzogen und einem anderen Programm zugeteilt. Auslöser für einen derartigen Programmwechsel ist im allgemeinen ein Hardware-Signal.

Bemerkungen:

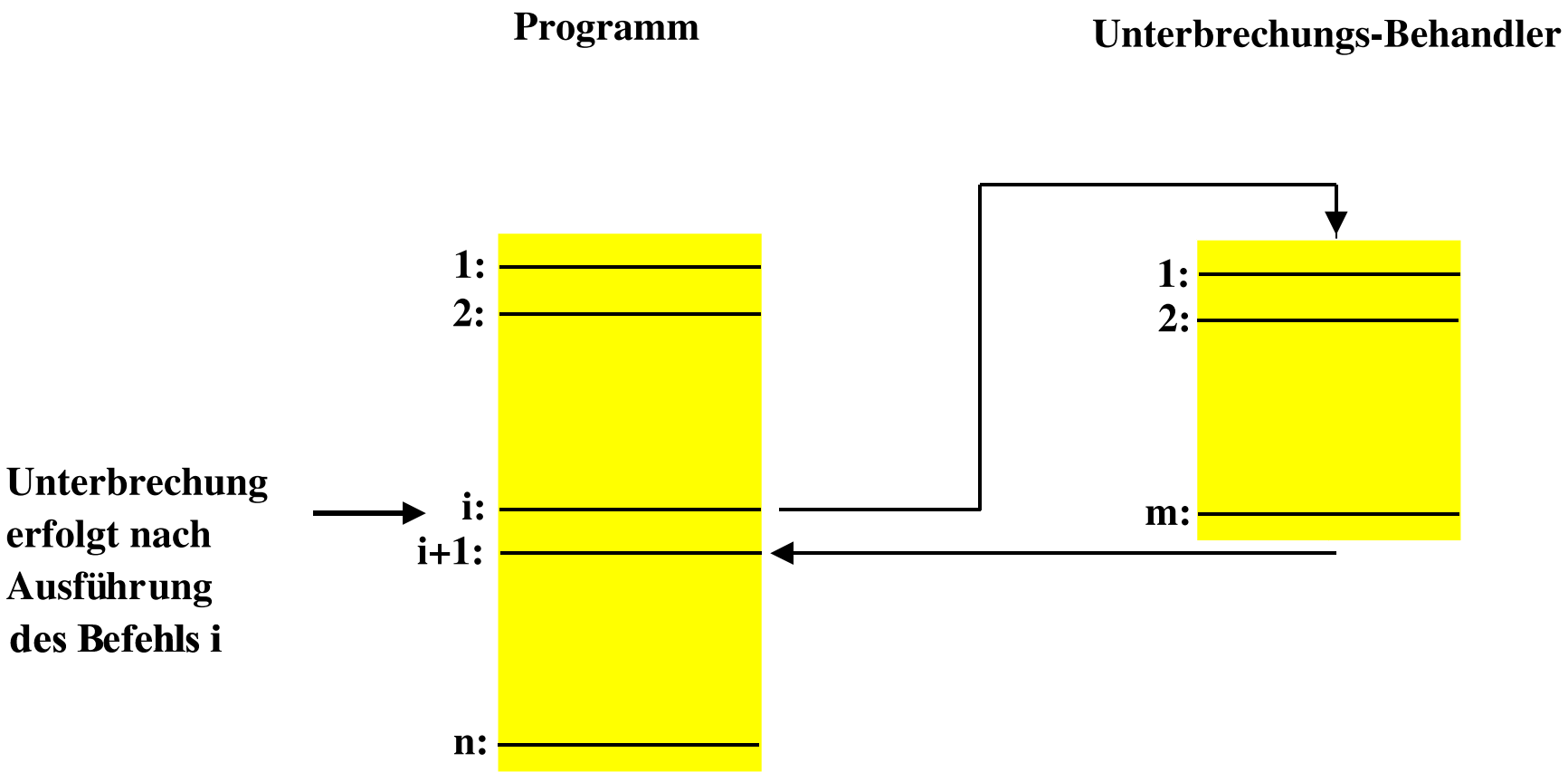
- Im allgemeinen wird eine Unterbrechung nur nach Ausführung eines Maschinenbefehls behandelt
- Ausser den von der Hardware ausgelösten Unterbrechungen gibt es *synchrone Exceptions*, die von der Software selbst ausgelöst werden (*traps*, *faults*, *aborts*).

Unterbrechungen (cont.)

Grober Ablauf einer Unterbrechung:

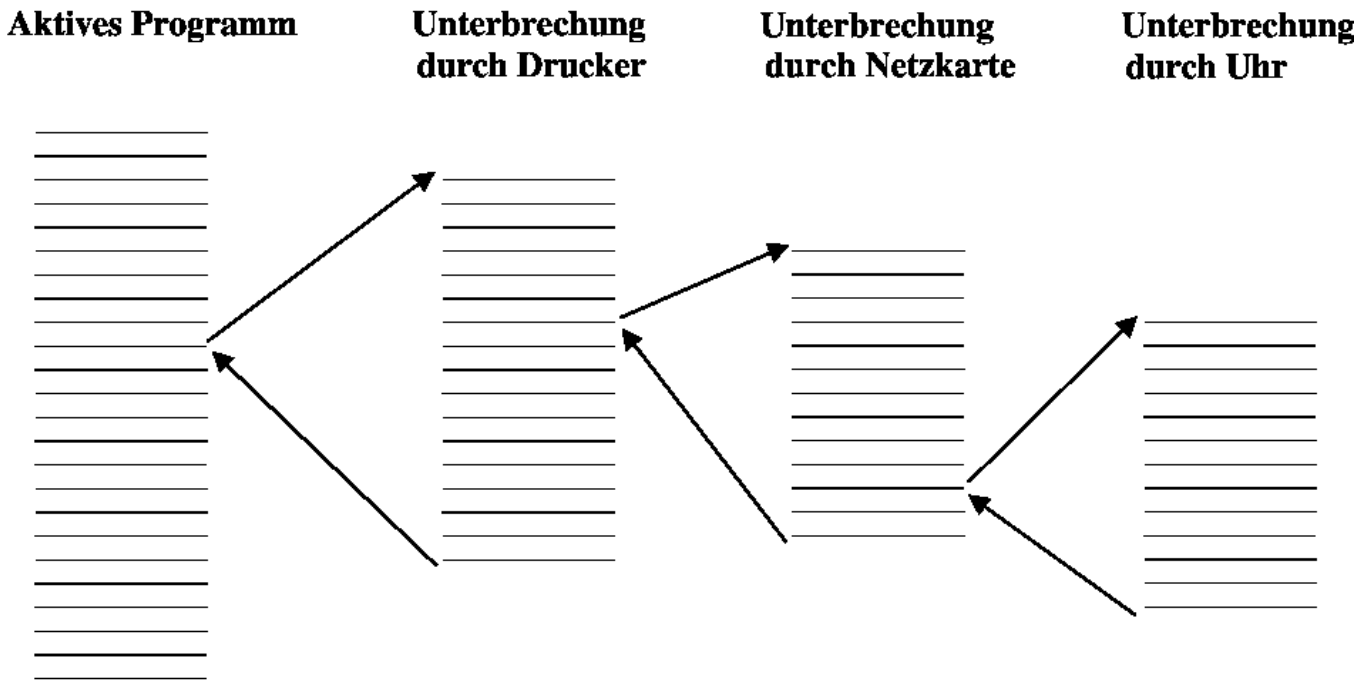
1. Ein Unterbrechungssignal S tritt auf und wird erkannt.
2. Es wird geprüft, ob das Signal S das aktive Programm AP unterbrechen darf.
3. Wenn nein, wird das Signal S zur späteren Behandlung in eine Warteschlange eingereiht.
4. Wenn ja, wird der Programmkontext von AP an vorbestimmten Orten gerettet.
5. Das dem Signal S zugeordnete Programm SP wird gestartet.
6. Hat das Programm SP seine Aufgabe erledigt, dann wird der Programmkontext von AP restauriert und das Programm AP setzt seine Arbeit fort. Im Idealfall hat eine Unterbrechung keine Auswirkungen auf das unterbrochene Programm.

Übergabe der Kontrolle mittels Unterbrechungen:



Wiederunterbrechung

Eine Unterbrechungsbehandlungsroutine kann auch wieder unterbrochen werden.



Bemerkung: Mittels Unterbrechungsmaskierung verhindert man die Unterbrechung einer Unterbrechungsbehandlungsroutine.

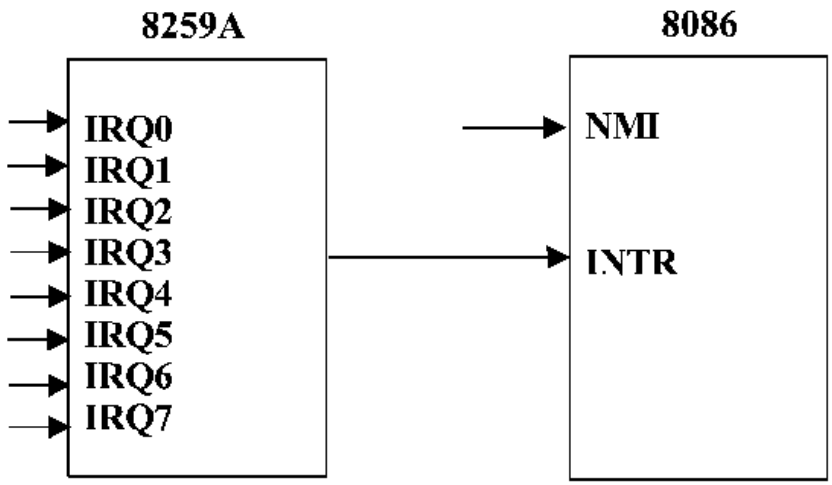
Bemerkungen zum Unterbrechungsbegriff

1. integriert in Behandlung von Ausnahmesituationen
   - Division durch 0, Schutzverletzungen, Ausrichtungsverletzungen, Seitenalarme
2. schneller Kontextwechsel $\longrightarrow$ Hardwarerealisierung
3. Priorisierung der Ausnahmeursachen
   - im Konfliktfall wird nach Priorität entschieden
4. Unterbrechungsanforderungen können überschreiben werden
5. Numerierung der möglichen Unterbrechungen
   - Startadresse der Unterbrechungsbehandlungsroutine aus Unterbrechungsnummer ableitbar
   - niederen Hauptspeicheradressen für Unterbrechungsvektoren

8086-Systeme

Bemerkungen:

1. 80x86 Prozessoren besitzen zwei Unterbrechungseingänge, einen normalen und einen NMI-Eingang (NMI = "Non-Maskable Interrupt")



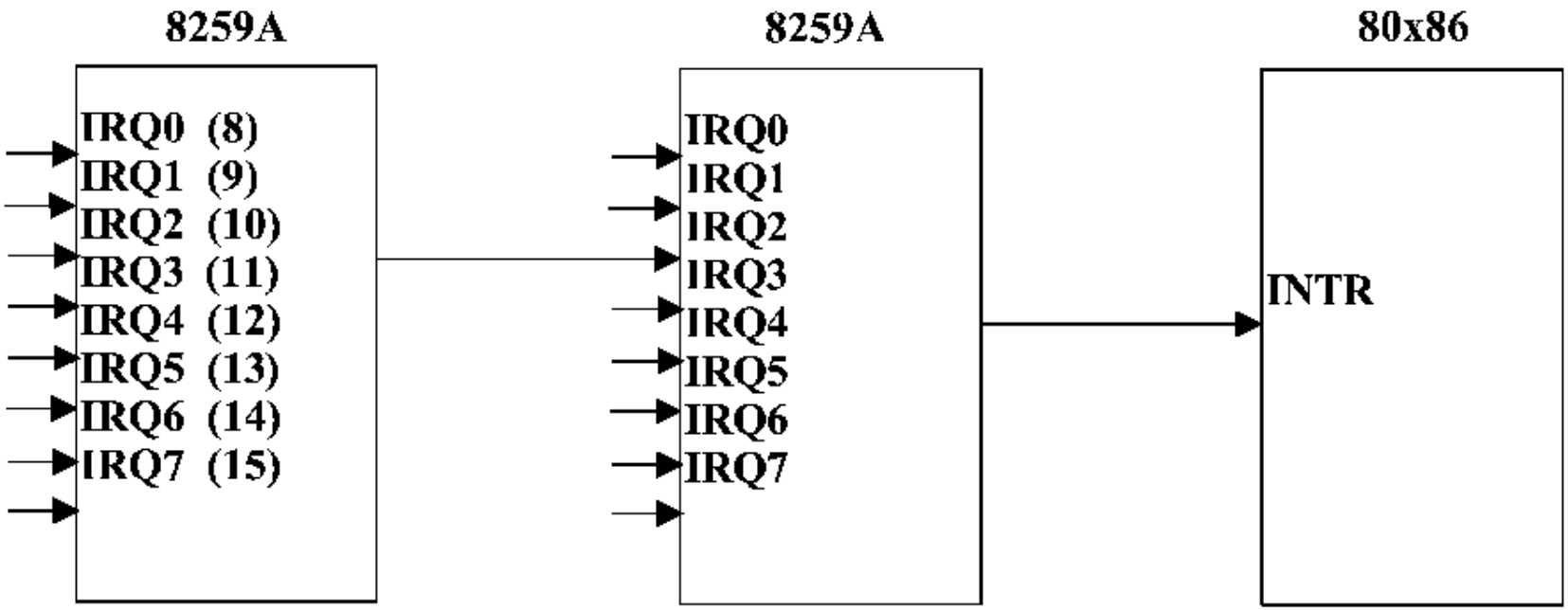
2. Ein 8086-System besitzt acht Unterbrechungsleitungen, die von einem 8259A-Baustein verwaltet werden. Der Unterbrechungs-Controller 8259A ist über die Ports 20h und 21h programmierbar.
3. Die nicht maskierbare Unterbrechung kann nicht über den CPU-Befehl CLI unterdrückt werden, wohl aber über Port 70h. Der "NMI" dient dazu, schwerwiegende Hardware-Fehler, wie z. B. Paritätsfehler und Stromausfall, zu erkennen.

Struktur einer Unterbrechungs-Behandlungs-Routine

Beispiel: einzelner 8259A PIC (PIC = Programmable Interrupt Controller)

1. Rette alle Register.
2. Führe nicht unterbrechbaren Code aus.
3. Ermögliche Unterbrechungen.
4. Führe unterbrechbaren Code aus.
5. Unterbinde Unterbrechungen
6. Sende nichtspezifischen End-Of-Interrupt an PIC.
7. Restauriere Register.
8. Rückkehr aus Unterbrechungsbehandler.

Kaskadierung von Unterbrechungen



Die normale Prioritätsanordnung der Unterbrechungen ist:
IRQ0, IRQ1, IRQ8 - IRQ15, IRQ3 - IRQ7

Struktur einer Unterbrechungsbehandlungsroutine

bei Kaskadierung von 8259A Bausteinen:

B = “Slave” und A = “Master”:

1. Rette alle Register
2. Führe nicht unterbrechbaren Code aus
3. Ermögliche Unterbrechungen
4. Führe unterbrechbaren Code aus
5. Unterbinde Unterbrechungen
6. Sende nichtspezifischen End-Of-Interrupt an PIC B
7. Lese ISR des PIC B

Struktur einer Unterbrechungsbehandlungsroutine (cont.)

bei Kaskadierung von 8259A Bausteinen:

8. Ist niedriger priorisierte Unterbrechung von B in Bearbeitung?

   Nein: Sende nichtspezifischen End-Of-Interrupt an A
   Restauriere Register
   Rückkehr vom Interrupt

   Ja: Restauriere Register
   Rückkehr vom Interrupt

Unterbrechungen im PC

und Zuordnung von Unterbrechungsleitungen und Unterbrechungsvektoren:

- Beispiel 80x86:
  - bis zu 256 Unterbrechungen
  - Adressen der Unterbrechungsroutinen: 0:0 bis 0:03fch
  - Unterbrechung hier Oberbegriff für:
    Ausnahme, Software-Interrupt und Hardware-Interrupt
- Beispiele für Ausnahmen:

  INT 0: Divisionsfehler
  INT 1: Einzelschrittmodus
  INT 5: Grenzverletzungen bei Array-Zugriff
  INT 6: Falscher Befehlscode

Geräte-Unterbrechungen im AT

Leitung	U-Vektor	Gerät
IRQ 0	8	Zeitunterbrechung
IRQ 1	9	Tastatur
IRQ 2	0ah	Kaskade für Controller 2
IRQ 3	0bh	Serielle Schnittstelle 2
IRQ 4	0ch	Serielle Schnittstelle 1
IRQ 5	0dh	Parallele Schnittstelle 2
IRQ 6	0eh	Diskettenlaufwerk
IRQ 7	0fh	Parallele Schnittstelle 1
IRQ 8	70h	Echtzeituhr
IRQ 9	71h	umgeleitet auf IRQ 2
IRQ 10–12	72h	reserviert
IRQ 13	75h	FPU Unterbrechung
IRQ 14	76h	Festplattenlaufwerk
IRQ 15	77h	reserviert

Unterbrechungsbehandlung für PC-Bus im Überblick

Schritte der Unterbrechungsbehandlung im System:

8259A	akzeptiert Unterbrechung
8259A	bestimmt Priorität
8259A	signalisiert Unterbrechung
Prozessor	bestätigt Unterbrechung
8259A	legt Unterbrechungsvektor auf Unterbrechung

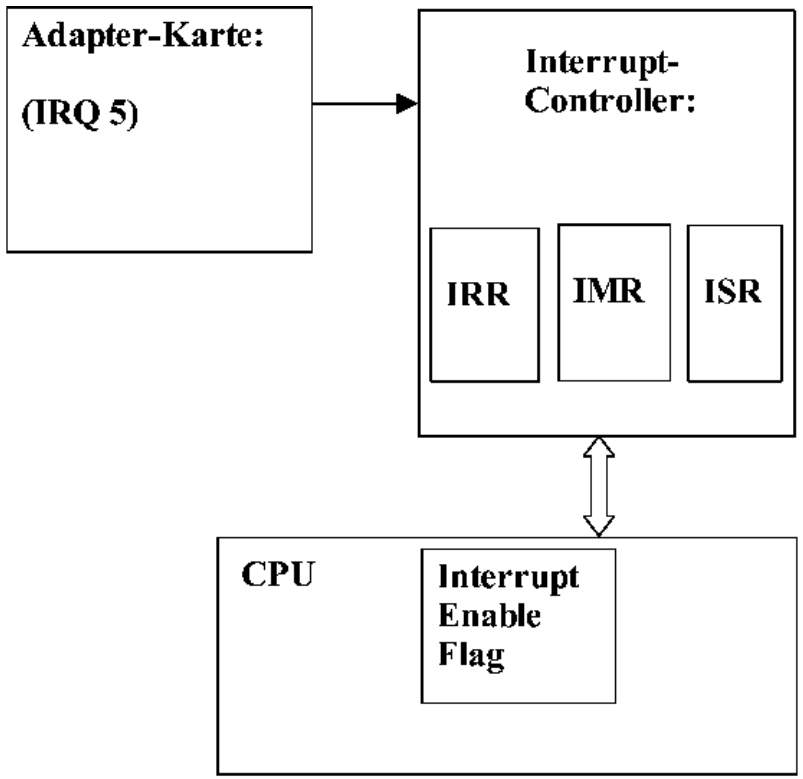
Unterbrechungsbehandlung für PC-Bus

Schritte der Unterbrechungsbehandlung durch Prozessor:

- Prozessor erhält Unterbrechungsaufforderung
- Prozessor beendet gegenwärtigen Befehl
- Prozessor bestätigt Unterbrechungssignal
- Prozessor rettet Statuswort und Befehlszähler
- Prozessor lädt neues Statuswort und neue Befehlsadresse
  - volle Maschinenzustand wird von Unterbrechungsroutine gerettet
  - Unterbrechung wird bearbeitet
  - Maschinenzustand wird restauriert
- altes Statuswort und alter Befehlszähler werden restauriert

Ein Beispiel zum Ablauf einer Unterbrechung im PC

Beispiel mit Unterbrechung 5



Ein Beispiel zum Ablauf einer Unterbrechung im PC (cont.)

Beispiel mit Unterbrechung 5

Ablauf einer Unterbrechung:

1. Ein Gerät meldet eine Unterbrechung über Eingang IR 5 des Interrupt-Controllers an.
2. Im IRR des Controllers wird das Bit 5 gesetzt.
3. Es wird geprüft, ob Unterbrechung 5 zulässig ist, bei Zulässigkeit wird zu Schritt 4 übergegangen.
4. Es wird überprüft, ob eine höher priorisierte Unterbrechung ansteht.
5. Falls nein, wird die CPU über Leitung INT be nachrichtigt.
6. Falls Unterbrechungen zulässig sind, sendet die CPU in kurzem Abstand zweimal ein INTA-Signal.

Ein Beispiel zum Ablauf einer Unterbrechung im PC (cont.)

Beispiel mit Unterbrechung 5

7. Bei Empfang des ersten INTA-Signals wird das Bit 5 im ISR gesetzt und Bit 5 im IRR gelöscht. Bei Empfang des zweiten INTA-Signals sendet der Controller die Adresse der Unterbrechungs-Behandlungs-Routine an die CPU.
8. Die CPU rettet den Zustand des gegenwärtigen Prozesses auf den Prozeßkeller und beginnt die Ausführung der Unterbrechungs-Routine.
9. Die Unterbrechungsbehandlung wird ausgeführt, in ihrem Verlauf wird das Unterbrechungssignal zurückgenommen.

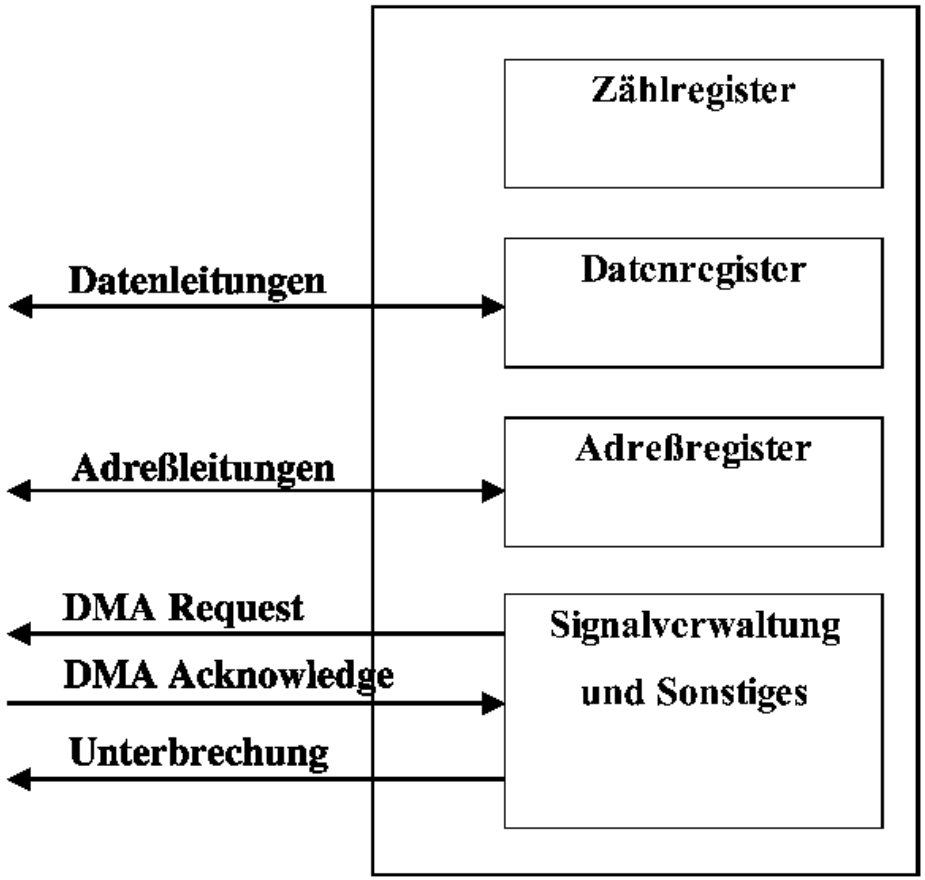
Ein Beispiel zum Ablauf einer Unterbrechung im PC (cont.)

Beispiel mit Unterbrechung 5

10. Die Unterbrechungs-Routine sendet End-of-Interrupt an den Controller, Datenwert 20h an Port 20h.
11. Der End-of-Interrupt Befehl löscht das Bit 5 im ISR, nun kann wieder eine neue Unterbrechung über IR 5 behandelt werden.
12. Mit dem Befehl IRET ("Return from Interrupt") wird die Unterbrechungsbehandlung beendet und der unterbrochene Prozeß wieder aufgenommen.

DMA ("Direct Memory Access")

Ein DMA-Modul ist ein einfacher E/A-Modul:

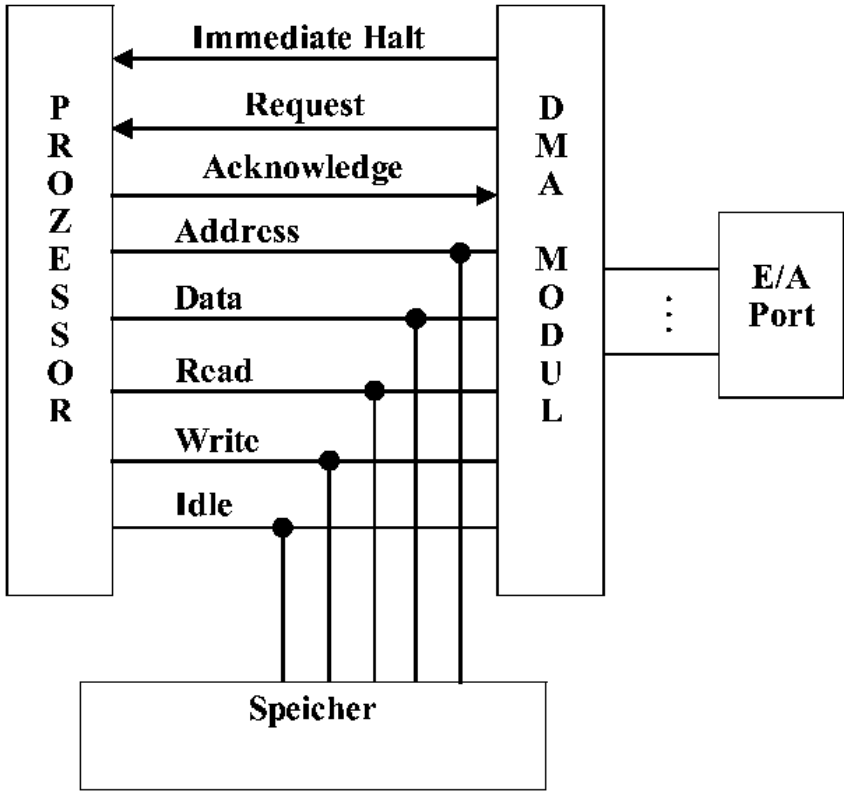


Bemerkung:
Anbindung des DMA-Moduls an die Ein- und Ausgabegeräte wird nicht gezeigt.

Zur Kommunikation Prozessor - DMA-Modul

Bemerkung:
Im Bild sind drei Möglichkeiten der exklusiven Nutzung des Systembusses durch den DMA-Modul vorgesehenen:

1. über Immediate Halt,
2. über Abfragen der Idle-Leitung,
3. ein Request - Acknowledge - Protokoll.



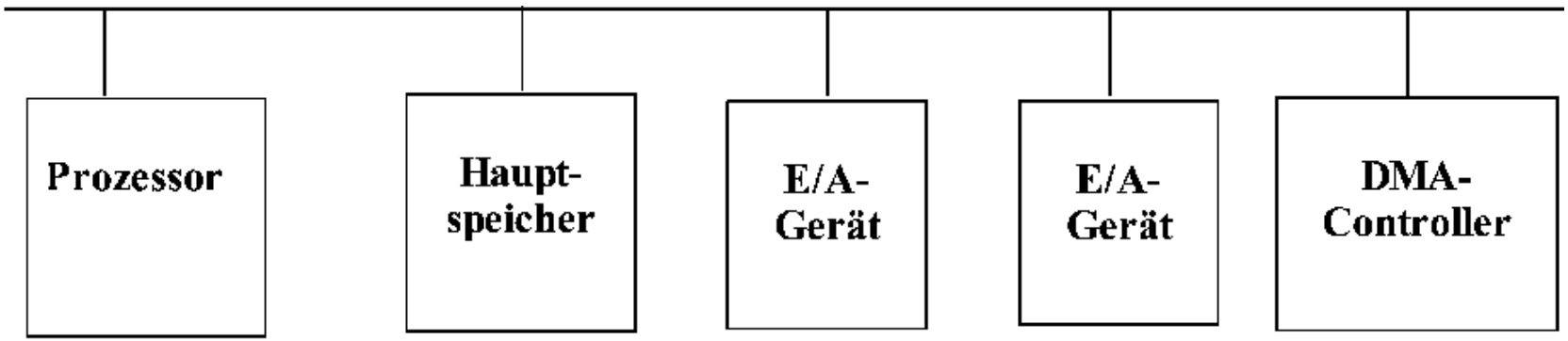
Ablauf eines DMA-Transfers:

1. Prozessor lädt Zählregister und Adreßregister des DMA-Moduls mit Startwerten
   - Adreßregister enthält Speicheradresse
   - Zählregister die Wortzahl
2. DMA-Modul bereit für Datentransfer?
   - DMA Request Signal wird gesetzt
   - Prozessor quittiert Busanforderung mit DMA Acknowledge
3. Transferierung eins Datenworts zwischen Speicher und DMA-Modul; Zähler und Adresse werden aktualisiert
4. DMA-Modul gibt Bus frei
   - Deaktivierung des DMA Request Signals
   - Prozessor quittiert mit Deaktivierung von DMA Acknowledge
5. Falls der Wortzähler <> 0?
   - ja: $\longrightarrow$ Wiederholung ab Schritt 2
   - nein: $\longrightarrow$ Prozessor durch Unterbrechungssignal benachrichtigt

Bemerkung: Die zeitweilige, kurzfristige Benutzung des Systembusses zur

DMA-Konfiguration (1)

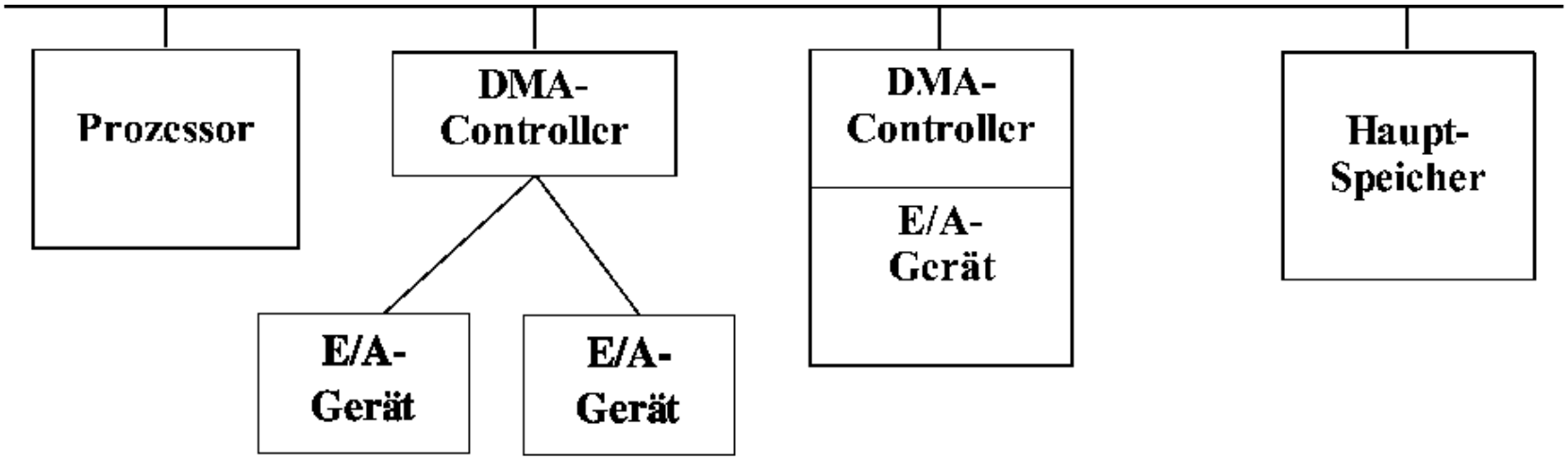
Ein-Bus-System:



Bemerkung: In dieser Konfiguration ist es denkbar, daß jeder Datentransfer den Bus zweimal beansprucht, ein Datum wird zwischen E/A-Gerät und DMA-Controller und zwischen DMA-Controller und Hauptpeicher übertragen; der Prozessor wird zweimal gestört.

DMA-Konfiguration (2)

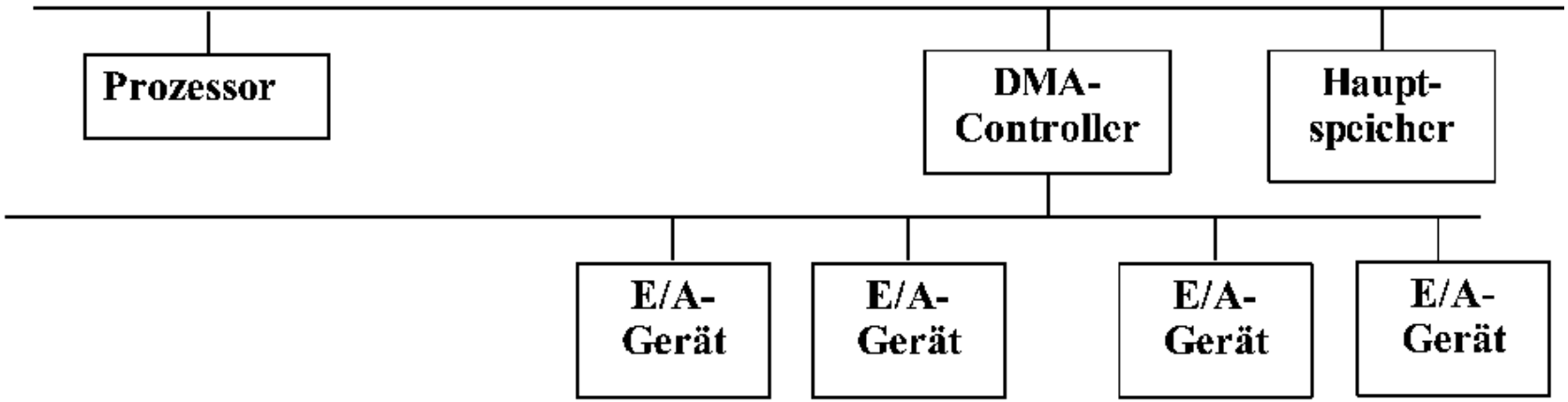
Ein-Bus-System mit integrierten DMA-Controllern:



Bemerkung: Jeder Datentransfer beansprucht den Bus nur einmal.

DMA-Konfiguration (3)

Separater E/A-Bus:



Bemerkung: Jeder Datentransfer beansprucht den Bus nur einmal, daher wird der Prozessor nur einmal pro Datentransfer gestört.

Ergänzende Literatur

[1] Randal E. Bryant and David O'Hallaron.
Computer Systems.
Pearson Education, Inc., New Jersey, 2003.

[2] N. Hendrich.
Pc technologie: Cd-rom und dvd.
http://tams-www.informatik.uni-hamburg.de/lehre/ss2002/pc-technologie/09-cdrom.pdf.

[3] M. Lehmann.
T3: Kapitel 4 (externspeicher).
http://www.informatik.uni-hamburg.de/TKRN/world/abro/T3.WS0102/t3k4ws02.pdf.

Ergänzende Literatur (cont.)

[4] Andrew S. Tanenbaum and James Goodman.
Computerarchitektur.
Pearson Studium München, 2001.

Gliederung

Gliederung (cont.)

Exceptional Control Flow

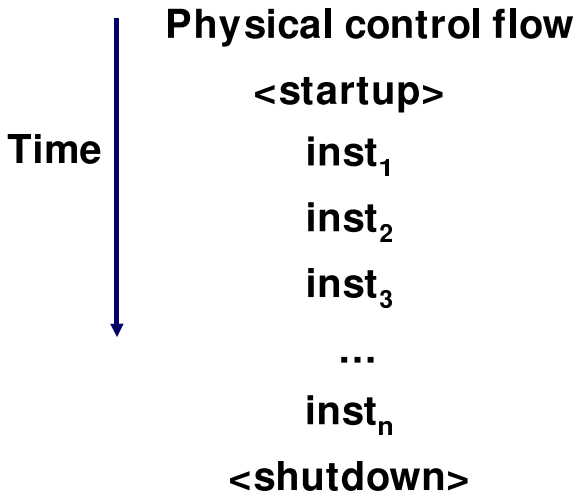
Themen:

- Ausnahmen
- Process Context Switch
- Erzeugen und Löschen von Prozessen

Kontrollfluss

Computer machen nur eines:

- Vom Hoch- bis zum Runterfahren wird jeweils nur eine Anweisungssequenz von der CPU gelesen und ausgeführt - und zwar Anweisung für Anweisung ("Befehlszyklus").
- Diese Sequenz stellt den physikalischen Kontrollfluss des Systems dar.



Änderung des Kontrollflusses

Bisher lediglich zwei Mechanismen, um den Kontrollfluss zu ändern

- Sprünge und Verzweigungen (Jumps and Branches)
- Aufrufe und Rücksprünge unter Verwendung des Stacks
- Beide reagieren auf Änderungen im Programmzustand

Unzureichend für ein sinnvolles System

- Schwierig für die CPU, auf Änderungen im Systemzustand zu reagieren
  - Daten kommen von einer Festplatte oder einer Netzwerkkarte an
  - Division durch Null
  - Benutzereingabe von ctrl-c über die Tastatur
  - Systemuhr läuft ab

System braucht Mechanismen für Ausnahmebehandlungen (Exceptional Control Flow)

Ausnahmebehandlungen

(Exceptional Control Flow)

Auf allen Ebenen eines Computersystems existieren Mechanismen für Ausnahmebehandlungen.

Mechanismus auf niederen Ebenen

- Exceptions
  - Änderung des Kontrollflusses als Reaktion auf ein Systemereignis (also Änderung des Systemzustands)
- Kombination von Hardware und OS-Software

Ausnahmebehandlungen (Forts.)

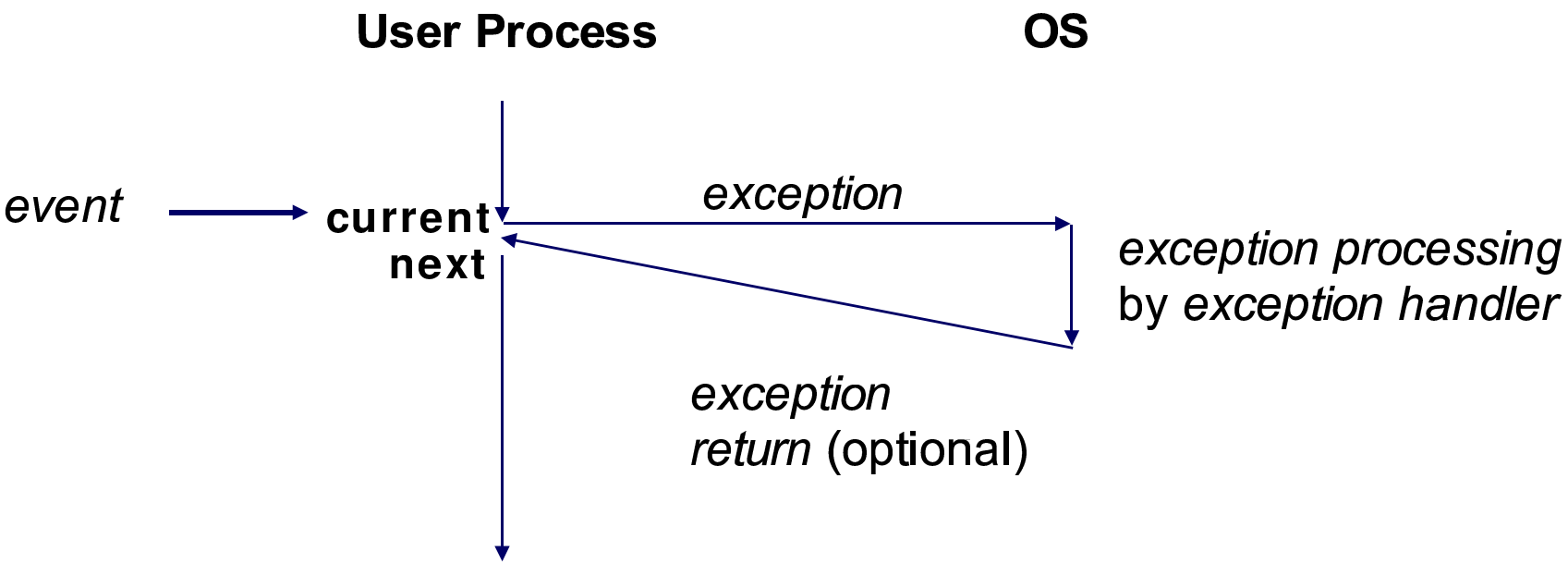
(Exceptional Control Flow)

Mechanismen auf höheren Ebenen

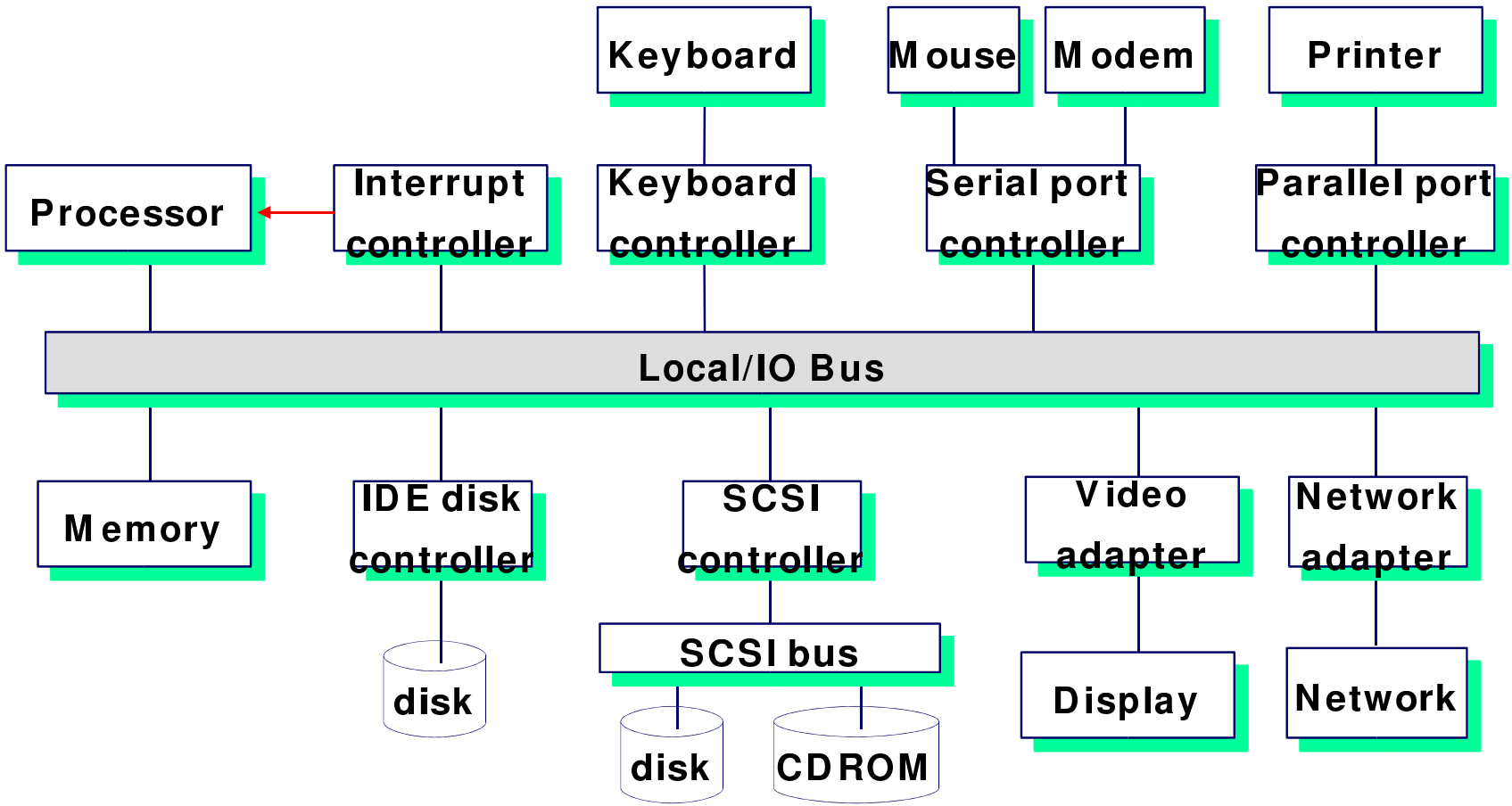
- Prozessumschaltung
- Signale
- Nichtlokale Sprünge (nonlocal Jumps)
- Implementiert durch
  - OS-Software (Context Switch und Signals)
  - oder C Runtime Library: nichtlokale Sprünge

Exceptions

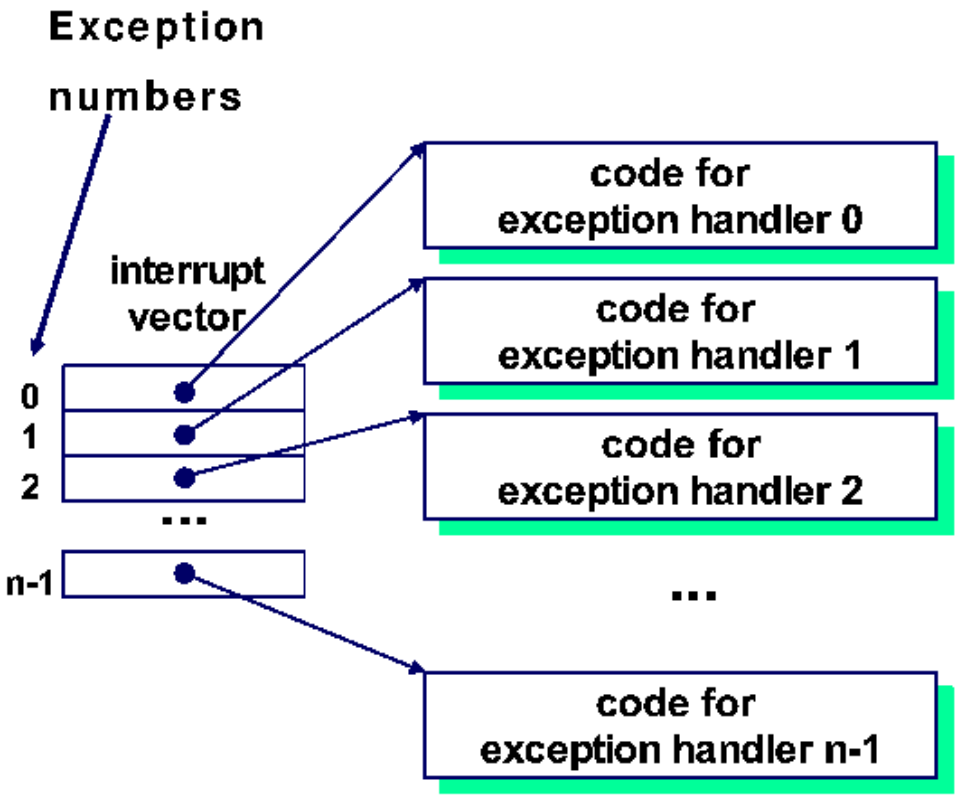
Eine Exception ist die übergabe der Kontrolle an das OS als Reaktion auf ein Ereignis (d.h. Änderung des Prozessorzustandes)



Systemkontext für Exceptions



Interrupt Vectors



- Each type of event has a unique exception number k
- Index into jump table (a.k.a., interrupt vector)
- Jump table entry k points to a function (exception handler).
- Handler k is called each time exception k occurs.

Asynchrone Exceptions (Interrupts)

Verursacht durch Ereignisse außerhalb des Prozessors

- werden durch Setzen der Prozessor-Interruptleitung angezeigt
- Interruptroutine lässt Prozessor bei der nächsten Anweisung der unterbrochenen Befehlsfolge weitermachen.

Beispiele

- I/O Interrupts
  - Eingabe von ctrl-c an der Tastatur
  - Eintreffen eines Pakets über das Netzwerk
  - Eintreffen eines Datensektors von der Festplatte
- Hard Reset Interrupt
  - Drücken des Resetknopfes
- Soft Reset Interrupt
  - Eingabe von ctrl-alt-del an der Tastatur

Synchrone Exceptions

Verursacht durch Ereignisse, die als Ergebnis der Ausführung einer Anweisung auftreten

- Traps
  - Beabsichtigt;
    Beispiele: Systemaufrufe, Breakpoint Traps und spezielle Instruktionen
  - Danach wird an der nächsten Anweisung der unterbrochenen Befehlsfolge weitergearbeitet
- Faults
  - Unbeabsichtig, aber möglicherweise behebbar;
    Beispiele: Seitenfehler (behebbar), Schutzverletzung (nicht behebbar)
  - Entweder neue Ausführung der fehlerverursachenden (aktuellen) Anweisung oder Abbruch.
- Aborts
  - Unbeabsichtigt und nicht behebbar;
    Beispiele: Parity Fehler, Hardwarefehler
  - Bricht aktuelles Programm ab

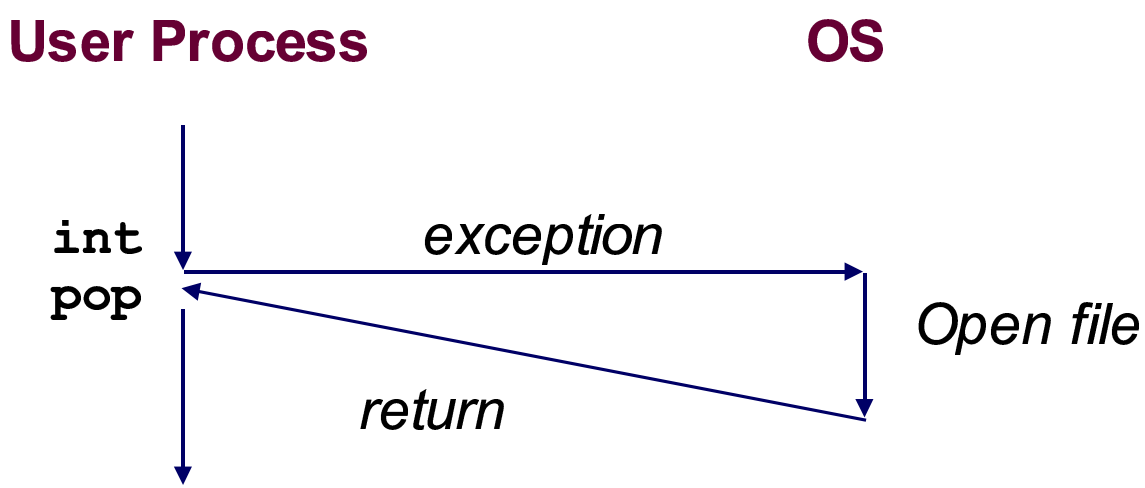
Trap Beispiel

Öffnen einer Datei:

- Benutzer ruft *open(filename, options)*
  - Funktion open führt den Assemblerbefehl *int* aus, der zu einem Systemaufruf führt.
- OS muss Datei finden oder erzeugen und für Schreiben und Lesen vorbereiten
- Gibt Integer File Descriptor zurück

Trap Beispiel (Forts.)

```
0804d070 <__libc_open>:
 . . .
 804d082:      cd 80                    int    $0x80
 804d084:      5b                       pop    %ebx
 . . .
```



Fault Beispiel 1

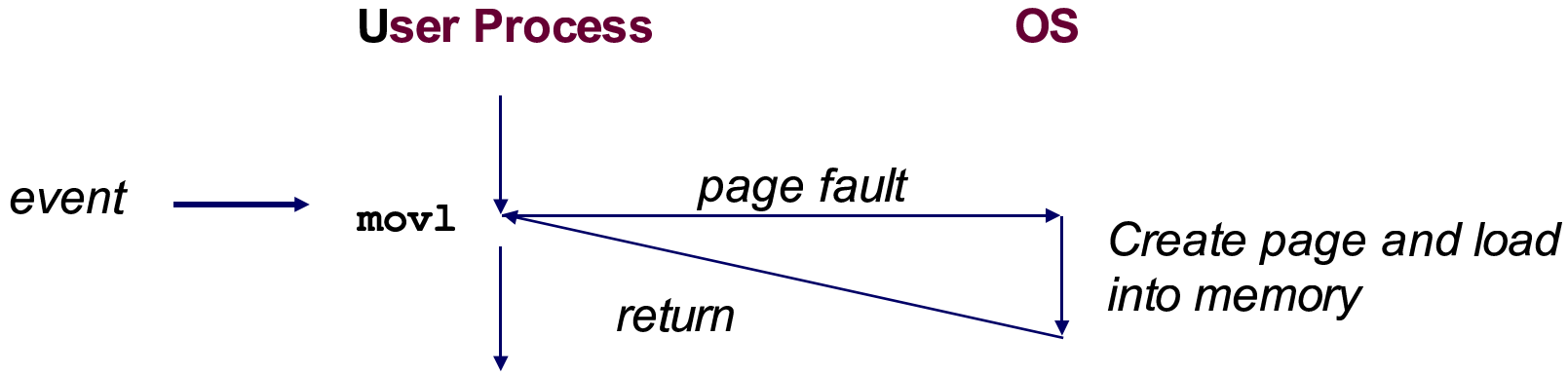
Speicherzugriff:

- Benutzer schreibt in eine Speicheradresse
- Diese Seite des Benutzerspeichers ist auf die Festplatte ausgelagert
- Seiten-Handler muss die Seite in den physikalischen Speicher laden
- Kehrt zur fehlerverursachenden Anweisung zurück
- Erfolgreicher zweiter Versuch

Fault Beispiel 1 (Forts.)

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

```
80483b7:    c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```



Fault Beispiel 2

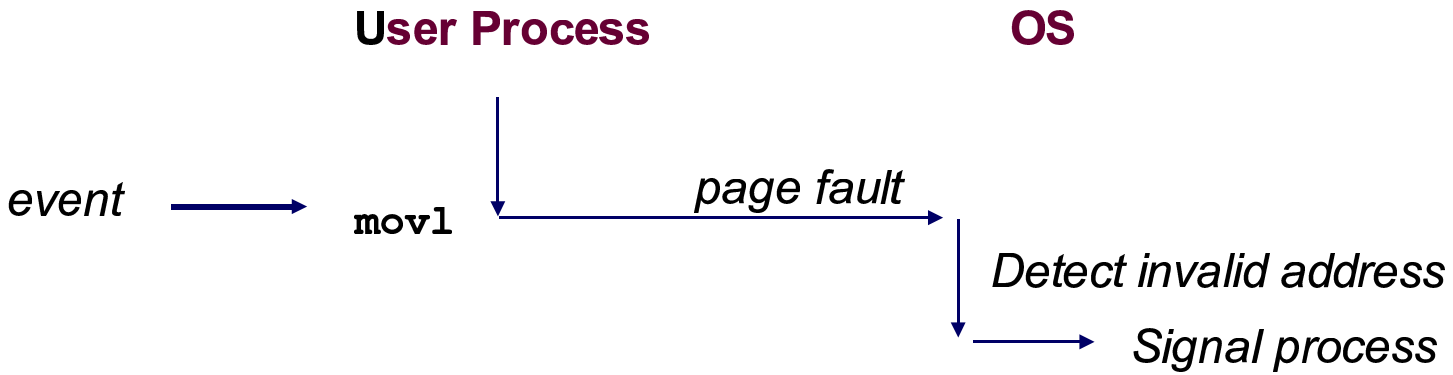
Speicherzugriff:

- Benutzer schreibt in eine Speicheradresse
- Adresse ist ungültig
- Seiten-Handler entdeckt ungültige Adresse
- Sendet *SIGSEG*-Signal zum Benutzerprozess
- Benutzerprozess wird mit “Segmentation Fault” beendet

Fault Beispiel 2 (Forts.)

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

```
80483b7:    c7 05 60 e3 04 08 0d   movl   $0xd,0x804e360
```



Prozesse

Definition

Ein Prozess ist die Instanz eines laufenden Programms

- eine der Grundideen der Informatik
- nicht das Gleiche wie “Programm” oder “Prozessor”

Prozessbegriff

- Prozessbegriff stellt zwei wesentliche Abstraktionen zur Verfügung
  - Logischer Kontrollfluss
    - jedes Programm hat scheinbar exklusiven Zugriff auf die CPU
  - privater Adressraum
    - jedes Programm hat scheinbar exklusiven Zugriff auf den Hauptspeicher
- Wie werden diese Illusionen aufrechterhalten?
  - Prozessausführungen werden verschränkt (Multitasking)
  - Adressraum wird durch das virtuelle Speichersystem verwaltet

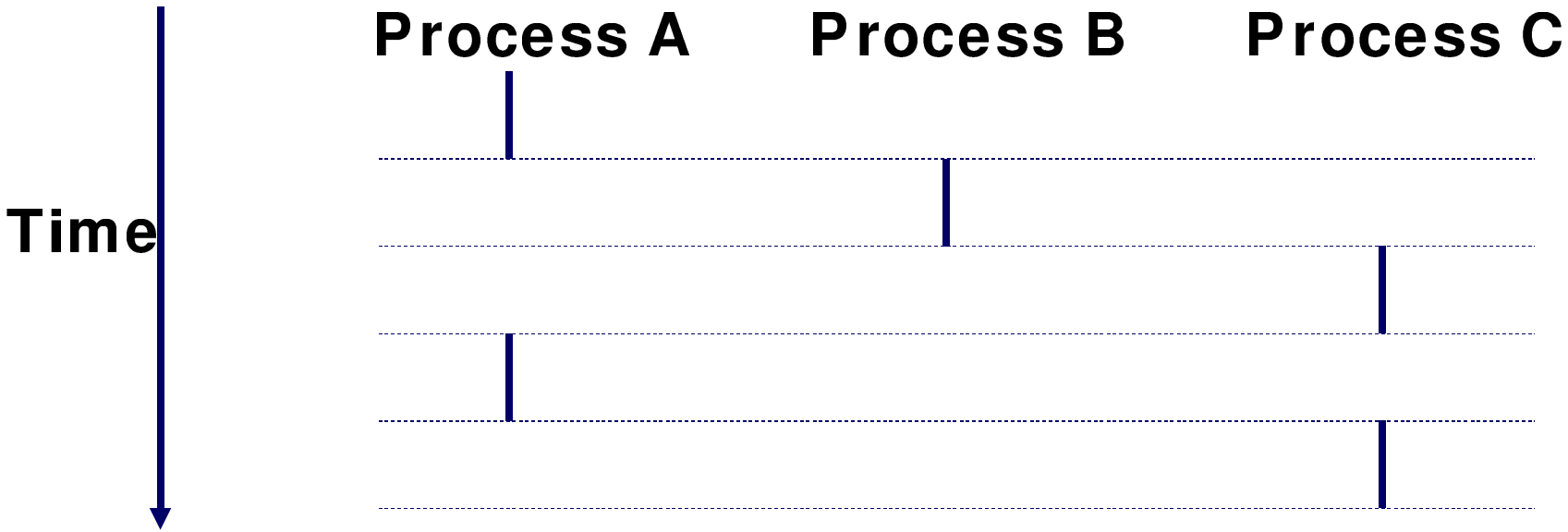
Multitasking

Mehrere Prozesse laufen konkurrent auf dem System

- Prozess: Ausführen eines Programms
  - Zustand besteht aus Speicherbelegung, Registerwerten und Programmzähler
- Wechselt ständig von einem Prozess zum anderen
  - Prozess wird suspendiert, wenn er auf I/O wartet oder Timer Event auftritt
  - Prozess wird entsprechend der Scheduling Strategie wiederaufgesetzt wenn I/O Operation abgeschlossen ist
- Stellt sich dem Benutzer dar, als würden alle Prozesse gleichzeitig ausgeführt
  - obwohl die meisten Syteme nur einen Prozess zur Zeit ausführen können
  - allerdings ist die Verarbeitungsleistung bedingt durch den Overhead geringer als würden sie wirklich alleine laufen

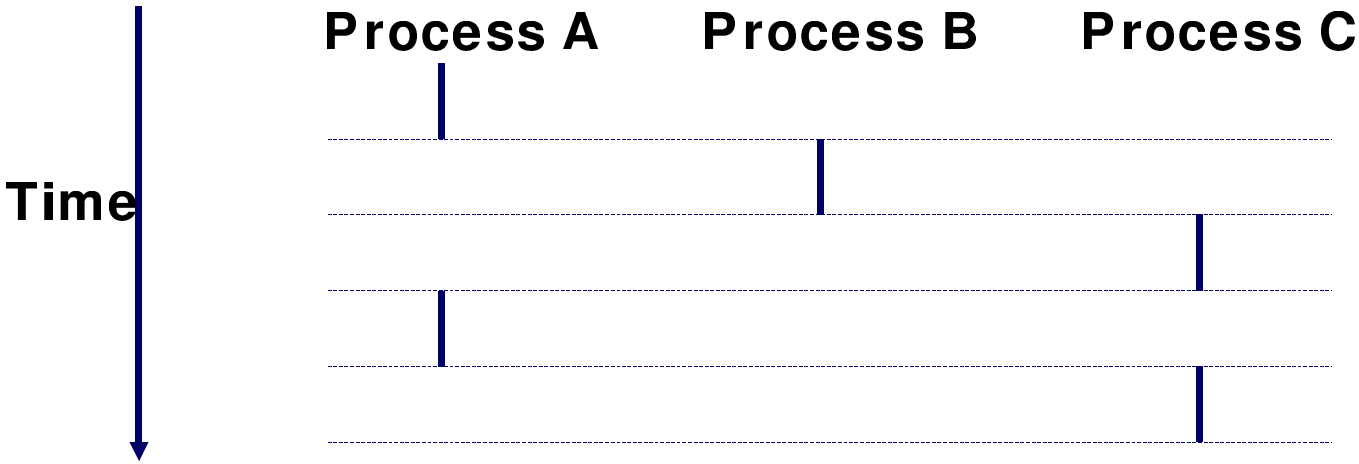
Logische Kontrollflüsse

Jeder Prozess hat seinen eigenen logischen Kontrollfluss.



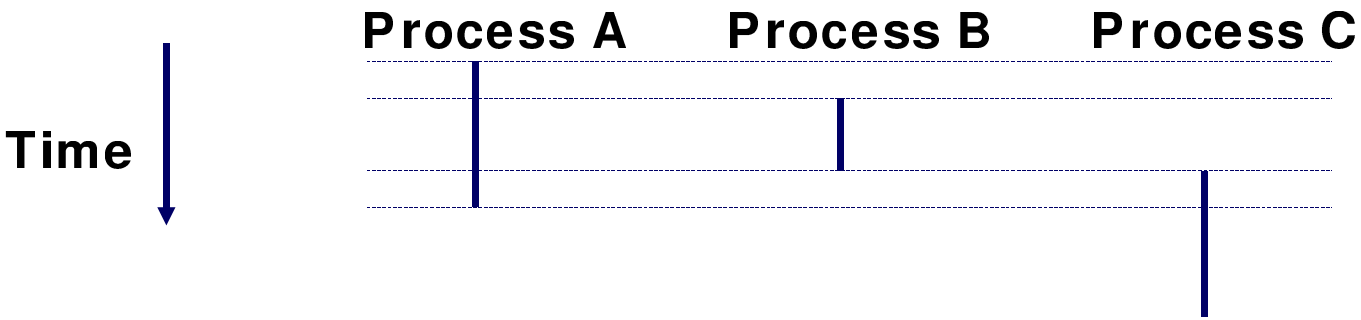
Konkurrente Prozesse

- Zwei Prozesse laufen konkurrent, wenn ihre Kontrollflüsse sich zeitlich überlappen.
- Ansonsten sind sie sequentiell.
- Beispiele:
  - Konkurrent: A & B, A & C
  - Sequenziell: B & C



Benutzersicht auf konkurrente Prozesse

- Kontrollflüsse für konkurrente Prozesse sind physikalisch getrennt in der Zeit.
- Trotzdem kann man konkurrente Prozesse als parallel laufend bezeichnen.

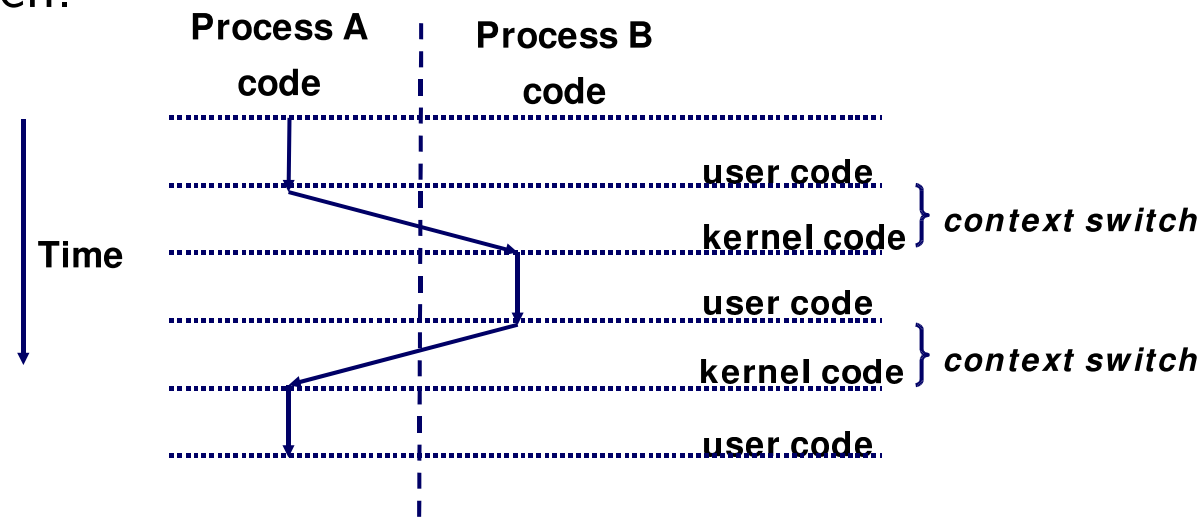


Context Switching

Prozesse werden vom *Kernel* verwaltet, einem gemeinsam genutzten OS Code-Block.

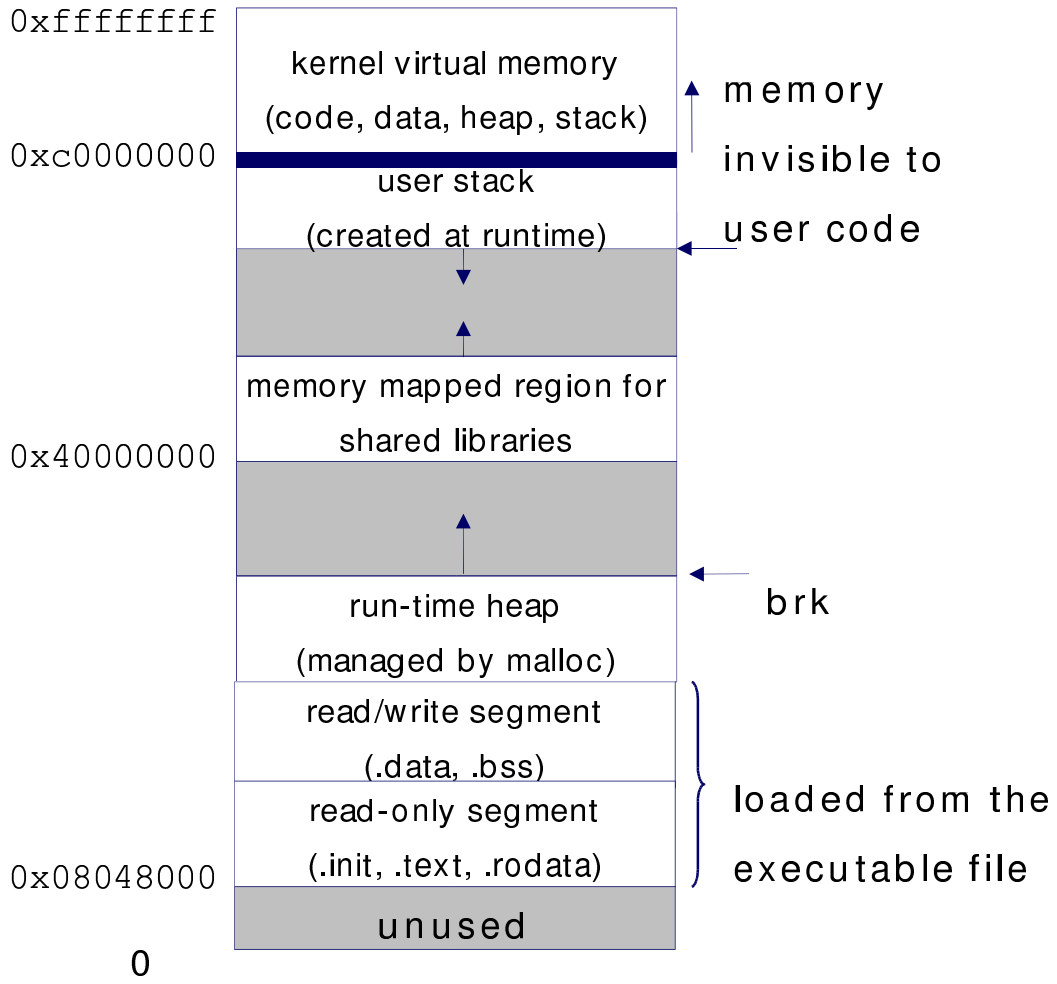
- Wichtig: Der Kernel ist kein eigenständiger Prozess, sondern läuft als Teil eines Benutzerprozesses.

Der Kontrollfluss wechselt über einen Context Switch von einem Prozess zum anderen.



Private Adressräume

Jeder Prozess
hat seinen eigenen
privaten Adressraum:



Zusammenfassung

Exceptions

- Ereignisse, die einen abweichenden Kontrollfluss erfordern
- Generierung extern (Interrupts) oder intern (Traps und Faults)

Prozesse

- Zu jeder gegebenen Zeit sind mehrere Prozesse im System aktiv
- Nur jeweils einer kann zur Zeit ausgeführt werden

Gliederung

Parallelrechner

Ständig steigende Anforderungen an die Rechenleistung, u.a. für:

- Wettervorhersage, Geologie
- Astronomie, Kernphysik, Gentechnologie
- Datenbanken, Transaktionssysteme

Problem/Konsequenz:

- Performance eines einzelnen Rechners ist begrenzt
- Verteilen eines Programms auf mehrere Prozessoren

Herausforderungen bei der Parallelverarbeitung:

- Wie viele und welche Prozessoren?
- Kommunikation zwischen den Prozessoren?
- Programmierung und Hilfssoftware?

Performance

- Antwortzeit ("wall clock time", "response time", "execution time"):
  Gesamtzeit zwischen Programmstart und -ende, inkl. I/O
- Ausführungszeit (reine CPU-Zeit)

user-time	CPU-Zeit für Benutzerprogramm
system-time	CPU-Zeit für OS-Aktivitäten
`Unix: time make`	`7.950u 2.390s 0:22.98 44.9%`

- Durchsatz: Anzahl bearbeiteter Programme / Zeit
- performance $= \frac{1}{execution\ time}$
- speedup $= \frac{performance\ x}{performance\ y} = \frac{execution\ time\ y}{execution\ time\ x}$

Performancegewinn

Ausführungszeit: Anzahl der Befehle * Zeit pro Befehl

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

Amdahl's Gesetz:

"Speedup" durch Parallelisierung

System 1: berechnet Funktion X, zeitlicher Anteil $0 < F < 1$

System 2: Funktion X' ist schneller als X mit "speedup" SX:
SX = Zeitbedarf(X) / Zeitbedarf(X')

Amdahl's Gesetz: $S_{gesamt} = \frac{1}{(1-F)+\frac{F}{SX}}$

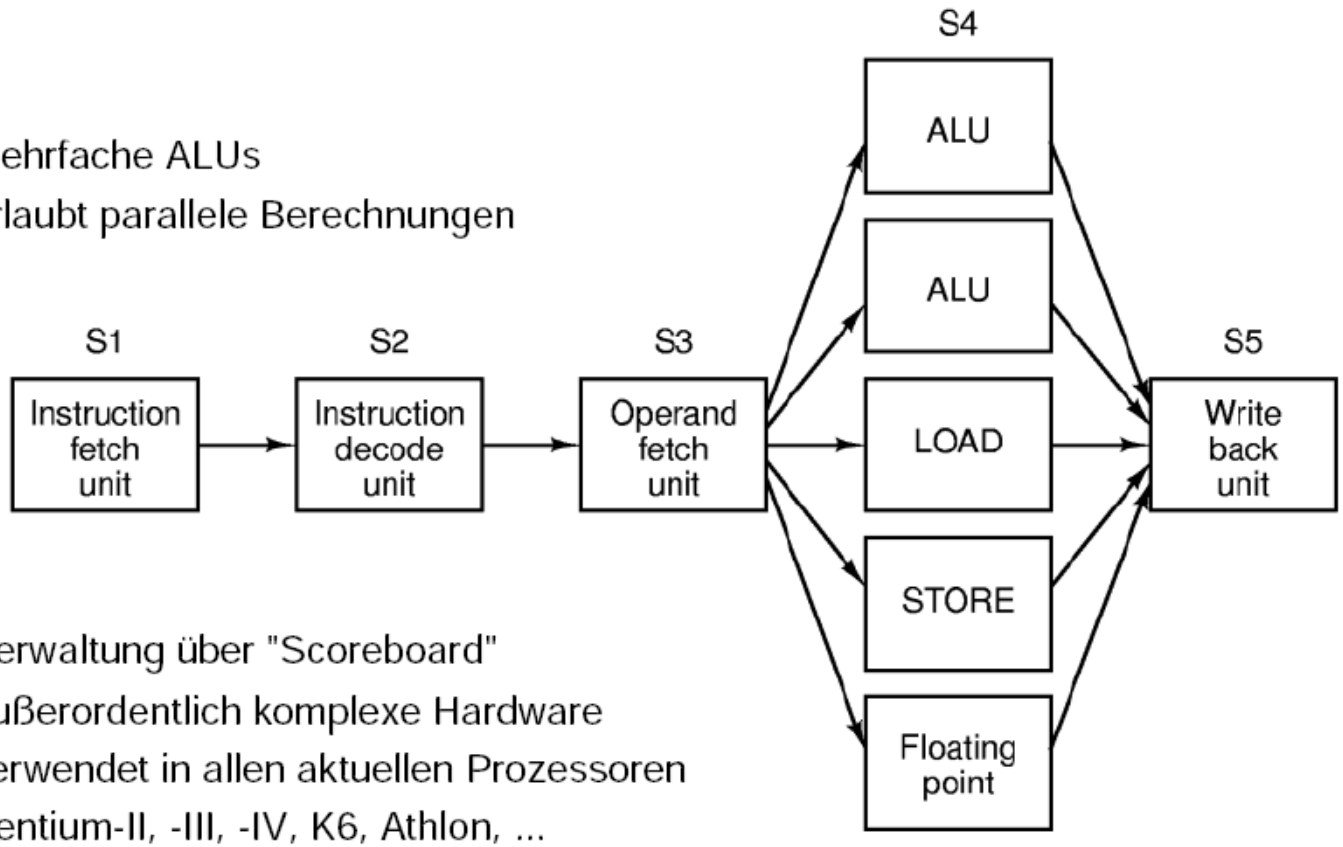
$\longrightarrow$ Optimierung lohnt nur für häufige Operationen !!

Beispiele:

SX = 1.1,	F = 0.98,	Sgesamt = 1/(0.02 + 0.89)	= 1.10
SX = 2,	F = 0.9,	Sgesamt = 1/(0.1 + 0.45)	= 1.82
SX = 2,	F = 0.5,	Sgesamt = 1/(0.5 + 0.25)	= 1.33
SX = 10,	F = 0.1,	Sgesamt = 1/(0.9 + 0.01)	= 1.09

Parallele Ausführung: Superskalarer Prozessor

- mehrfache ALUs
- erlaubt parallele Berechnungen



- Verwaltung über "Scoreboard"
- außerordentlich komplexe Hardware
- verwendet in allen aktuellen Prozessoren
- Pentium-II, -III, -IV, K6, Athlon, ...

Parallelrechner: Motivation

Leistungssteigerung durch schnellere Einzelprozessoren
Grenze: Takte oberhalb von 10 GHz derzeit unrealistisch

$\Rightarrow$ mehrere Prozessoren

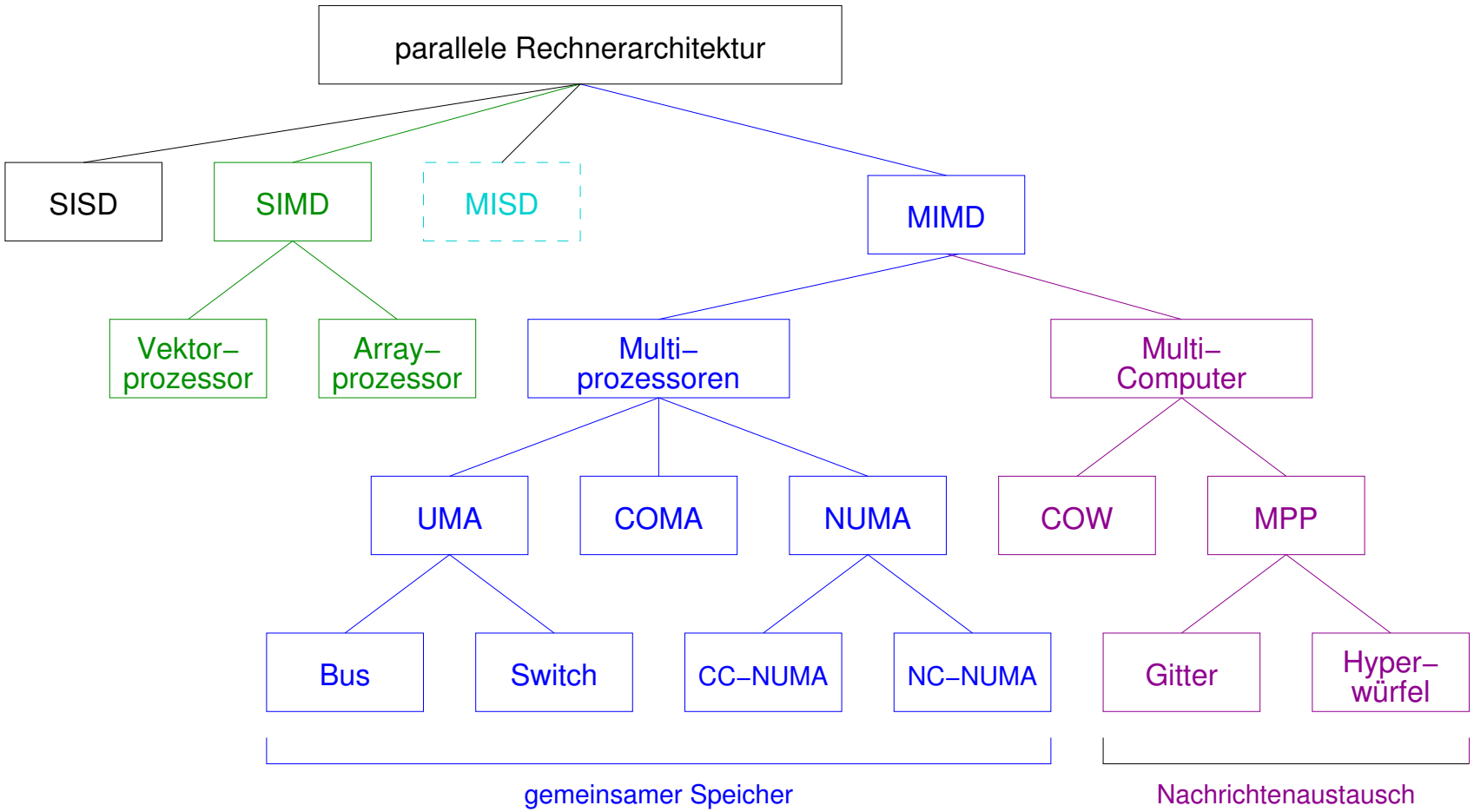
- diverse Architekturkonzepte
- shared-memory vs. message-passing
- Overhead durch Kommunikation
- Programmierung ist ungelöstes Problem
- derzeit beliebter Kompromiss:
  - bus-basierte SMPs mit 2..16 Prozessoren

SIMD: Flynn-Klassifikation

SIMD	"single instruction, single data" ⇒ jeder klassische PC
SIMD	"single instruction, multiple data" ⇒ Feldrechner/Parallelrechner ⇒ z.B. Connection-Machine 2: 64K Prozessoren ⇒ eingeschränkt: MMX/SSE: 2-8 fach parallel
MIMD	"multiple instruction, multiple data" ⇒ Multiprozessormaschinen ⇒ z.B. vierfach PentiumPro-Server
MISD	. . .

Parallelrechner: Klassifikation

(Tanenbaum)



Ergänzende Literatur

[1] Randal E. Bryant and David O'Hallaron.
Computer Systems.
Pearson Education, Inc., New Jersey, 2003.

[2] David A. Patterson and John L. Hennessy.
Computer Organization and Design. The Hardware / Software Interface.
Morgan Kaufmann Publishers, Inc., San Francisco, 1998.

[3] Andrew S. Tanenbaum and James Goodman.
Computerarchitektur.
Pearson Studium München, 2001.