

# 64-040 Modul IP7: Rechnerstrukturen

[http://tams.informatik.uni-hamburg.de/  
lectures/2012ws/vorlesung/rs](http://tams.informatik.uni-hamburg.de/lectures/2012ws/vorlesung/rs)

– Kapitel 15 –

Andreas Mäder



Universität Hamburg  
Fakultät für Mathematik, Informatik und Naturwissenschaften  
Fachbereich Informatik

**Technische Aspekte Multimodaler Systeme**

Wintersemester 2012/2013

# Kapitel 15

## Instruction Set Architecture

Speicherorganisation

Befehlssatz

Befehlsformate

Adressierungsarten

Intel x86-Architektur

Literatur



# Befehlssatzarchitektur – ISA

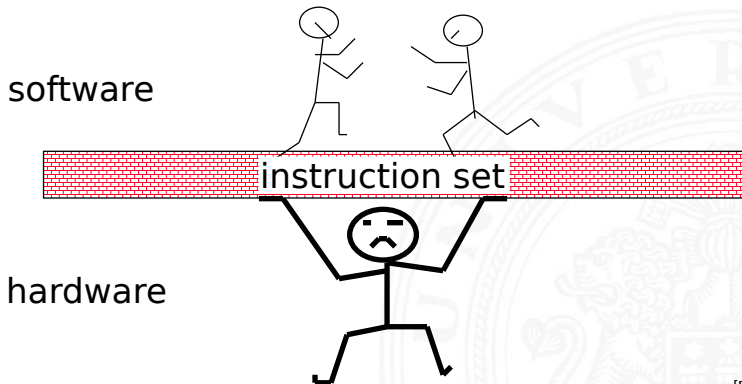
## ISA – **I**nstruction **S**et **A**rchitecture

⇒ alle für den Programmierer sichtbaren Attribute eines Rechners

- ▶ der (konzeptionellen) Struktur
  - ▶ Funktionseinheiten der Hardware:  
Recheneinheiten, Speichereinheiten, Verbindungssysteme, ...
- ▶ des Verhaltens
  - ▶ Organisation des programmierbaren Speichers
  - ▶ Datentypen und Datenstrukturen: Codierungen und Darstellungen
  - ▶ Befehlssatz
  - ▶ Befehlsformate
  - ▶ Modelle für Befehls- und Datenzugriffe
  - ▶ Ausnahmebedingungen

## Befehlssatzarchitektur – ISA (cont.)

- Befehlssatz: die zentrale Schnittstelle



[PH12]

# Merkmale der Instruction Set Architecture

- ▶ Speichermodell                      Wortbreite, Adressierung, . . .
- ▶ Rechnerklasse                      Stack-/Akku-/Registermaschine
- ▶ Registersatz                      Anzahl und Art der Rechenregister
- ▶ Befehlssatz                      Definition aller Befehle
- ▶ Art, Zahl der Operanden              Anzahl/Wortbreite/Reg./Speicher
- ▶ Ausrichtung der Daten              Alignment/Endianness
- ▶ Ein- und Ausgabe, Unterbrechungsstruktur (Interrupts)
- ▶ Systemsoftware                      Loader, Assembler,  
   Compiler, Debugger



# Artenvielfalt vom „Embedded 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	1K..10M	< 64M
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..64 bit Prozessoren, DSPs, digitale/analoge ASICs, ...
- ▶ Sensoren/Aktoren: Tasten, Displays, Druck, Temperatur, Antennen, CCD, ...
- ▶ Echtzeit-, Sicherheits-, Zuverlässigkeitsanforderungen

# Speicherorganisation

- ▶ Wortbreite, Größe / Speicherkapazität
- ▶ „Big Endian“ / „Little Endian“
- ▶ „Alignment“
- ▶ „Memory-Map“
- ▶ Beispiel: PC mit Windows
  
- ▶ spätere Themen
  - ▶ Cache-Organisation für schnelleren Zugriff
  - ▶ Virtueller Speicher für Multitasking
  - ▶ Synchronisation in Multiprozessorsystemen (MESI-Protokoll)



# Wortbreite

## ► Speicherwortbreiten historisch wichtiger Computer

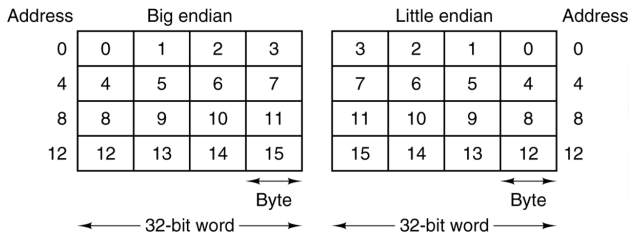
Computer	Bits/cell
Burroughs B1700	1
IBM PC	8
DEC PDP-8	12
IBM 1130	16
DEC PDP-15	18
XDS 940	24
Electrologica X8	27
XDS Sigma 9	32
Honeywell 6180	36
CDC 3600	48
CDC Cyber	60

[Tan09]

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



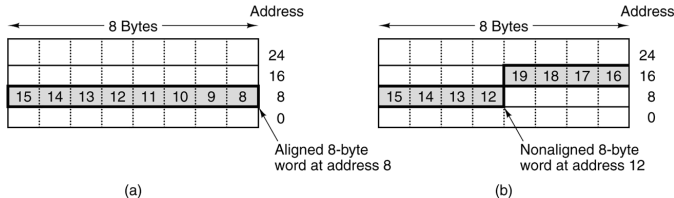
# Big- vs. Little Endian



[Tan09]

- ▶ Anordnung einzelner Bytes in einem Wort (hier 32 bit)
  - ▶ Big Endian ( $n \dots n + 3$ ): MSB...LSB „String“-Reihenfolge
  - ▶ Little Endian ( $n \dots n + 3$ ): LSB ...MSB „Zahlen“-Reihenfolge
- ▶ beide Varianten haben Vor- und Nachteile
- ▶ ggf. Umrechnung zwischen beiden Systemen notwendig
- siehe „8.6 Logische Operationen – Speicher-Organisation“

# „Misaligned“ Zugriff



[Tan09]

- ▶ Beispiel: 8-Byte-Wort in 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
  - ▶ Programmabbruch

(x86)  
(MIPS)

# Memory Map

- ▶ CPU kann im Prinzip alle möglichen Adressen ansprechen
- ▶ in der Regel: **kein voll ausgebauter Speicher**  
32 bit Adresse entsprechen 4 GiB Hauptspeicher, 64 bit ...

- ▶ Aufteilung in RAM und ROM-Bereiche
- ▶ ROM mindestens zum Booten notwendig
- ▶ zusätzliche Speicherbereiche für „memory mapped“ I/O

⇒ „Memory Map“

- ▶ Adressdecoder
- ▶ Hardwareeinheit
- ▶ Zuordnung von Adressen zu „realem“ Speicher

## 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

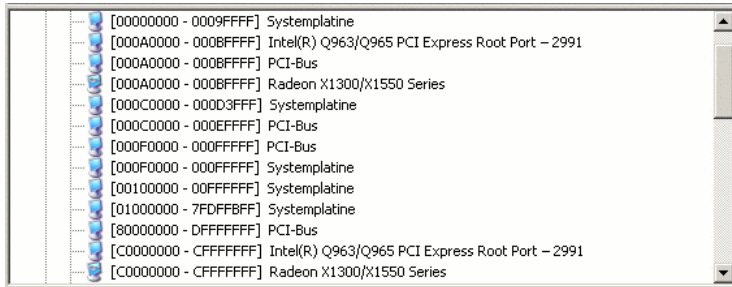
Demo und Beispiele: im RS-Praktikum (64-042)

# Memory Map: Windows 9x

gemeinsam genutzter Systembereich	FFFFFFFh	1 GB
gemeinsam genutzt für Anwendungen	C0000000h	1 GB
privater Adreßbereich Anwendungen	80000000h	knapp 2 GB
ungenutzt	00400000h	4 MB
V86 Bereich	0010FFF0h 00000000h	1 MB inklusive "8086 A20 bug": real mode Bereich

- ▶ DOS-Bereich immer noch für Boot / Geräte (VGA) reserviert
- ▶ Kernel, Treiber, usw. im oberen 1 GiB-Bereich
- ▶ 2 GiB für Anwendungen

## Memory Map: Windows 9x (cont.)



[00000000 - 0009FFFF]	Systemplatine
[000A0000 - 000BFFFF]	Intel(R) Q963/Q965 PCI Express Root Port - 2991
[000A0000 - 000BFFFF]	PCI-Bus
[000A0000 - 000BFFFF]	Radeon X1300/X1550 Series
[000C0000 - 000D3FFF]	Systemplatine
[000C0000 - 000EFFFF]	PCI-Bus
[000F0000 - 000FFFFFFF]	PCI-Bus
[000F0000 - 000FFFFFFF]	Systemplatine
[00100000 - 00FFFFFFF]	Systemplatine
[01000000 - 7FDFFBFF]	Systemplatine
[80000000 - DFFFFFFF]	PCI-Bus
[C0000000 - CFFFFFFF]	Intel(R) Q963/Q965 PCI Express Root Port - 2991
[C0000000 - CFFFFFFF]	Radeon X1300/X1550 Series

- ▶ 32-bit Adressen, 4 GiByte Adressraum
- ▶ Aufteilung 2 GiB für Programme, obere 1+1 GiB für Windows
- ▶ Beispiel der Zuordnung, diverse Bereiche für I/O reserviert



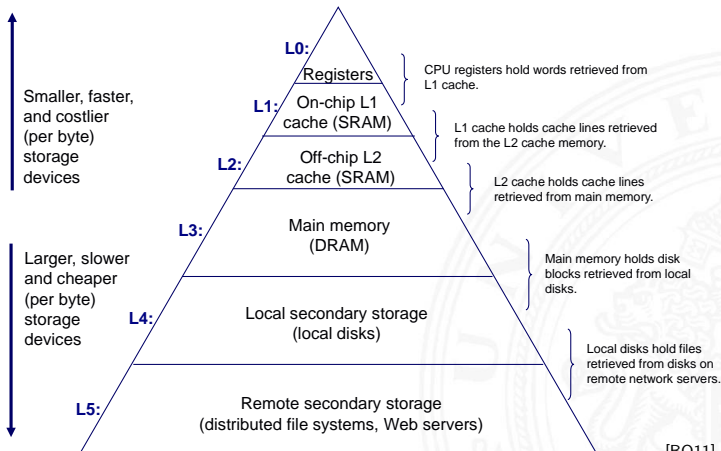
## Memory Map: Windows 9x (cont.)

### I/O-Speicherbereiche

[00000378 - 0000037F]	ECP-Druckeranschluss (LPT1)
[00000380 - 000003BB]	Hauptplatinenressourcen
[00000380 - 000003BB]	Intel(R) Q963/Q965 PCI Express Root Port - 2991
[00000380 - 000003BB]	Radeon X1300/X1550 Series
[000003C0 - 000003DF]	Intel(R) Q963/Q965 PCI Express Root Port - 2991
[000003C0 - 000003DF]	Radeon X1300/X1550 Series
[000003C0 - 000003E7]	Hauptplatinenressourcen
[000003F0 - 000003F5]	Standard-Diskettenlaufwerkcontroller
[000003F6 - 000003F7]	Hauptplatinenressourcen
[000003F7 - 000003F7]	Standard-Diskettenlaufwerkcontroller
[000003F8 - 000003FF]	Kommunikationsanschluss (COM1)
[00000400 - 000004CF]	Hauptplatinenressourcen
[000004D0 - 000004D1]	Programmierbarer Interruptcontroller

- ▶ x86 I/O-Adressraum gesamt nur 64 KiByte
- ▶ je nach Zahl der I/O-Geräte evtl. fast voll ausgenutzt
- ▶ Adressen vom BIOS zugeteilt

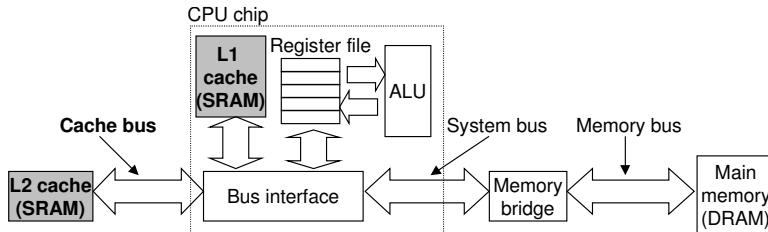
# Speicherhierarchie



[BO11]

später mehr...

# Cache-Speicher



[BO11]

- ▶ Cache Strategien
  - ▶ Welche Daten sollen in Cache?
  - ▶ Welche werden aus Cache entfernt?
- ▶ Cache Abbildung: direct-mapped, n-fach assoz., voll assoziativ
- ▶ Cache Organisation: Größe, Wortbreite, etc.

# Speicher ist wichtig!

- ▶ Speicher ist nicht unbegrenzt
    - ▶ muss zugeteilt und verwaltet werden
    - ▶ viele Anwendungen werden vom Speicher dominiert
  - ▶ Fehler, die auf Speicher verweisen, sind besonders gefährlich
    - ▶ Auswirkungen sind sowohl zeitlich als auch räumlich entfernt
  - ▶ Speicherleistung ist nicht gleichbleibend
- Wechselwirkungen: Speichersystem  $\Leftrightarrow$  Programme
- ▶ „Cache“- und „Virtual“-Memory Auswirkungen können Performance/Programmleistung stark beeinflussen
  - ▶ Anpassung des Programms an das Speichersystem kann Geschwindigkeit bedeutend verbessern

→ siehe „18 Speicherhierarchie“

# ISA-Merkmale des Prozessors

- ▶ Befehlszyklus
- ▶ Befehlsklassen
- ▶ Registermodell
- ▶ n-Adress Maschine
- ▶ Adressierungsarten



# Befehlszyklus

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

PC	Program Counter	Adresse des Befehls
IR	Instruction Register	aktueller Befehl
R0...R31	Registerbank	Rechenregister (Operanden)

# Instruction Fetch

## „Befehl holen“ Phase im Befehlszyklus

1. Programmzähler (PC) liefert Adresse für den Speicher
2. Lesezugriff auf den Speicher
3. Resultat wird im Befehlsregister (IR) abgelegt
4. Programmzähler wird inkrementiert
  - ▶ Beispiel für 32 bit RISC mit 32 bit Befehlen
    - ▶  $IR = MEM[PC]$
    - ▶  $PC = PC + 4$
  - ▶ bei CISC-Maschinen evtl. weitere Zugriffe notwendig, abhängig von der Art (und Länge) des Befehls

# Instruction Decode

## „Befehl decodieren“ Phase im Befehlszyklus

- ▷ Befehl steht im Befehlsregister IR
- 1. Decoder entschlüsselt Opcode und Operanden
- 2. leitet Steuersignale an die Funktionseinheiten
- 3. 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
- 1. Ausführung des Befehls durch Aktivierung der Funktionseinheiten
  - ▶ Details abhängig von der Art des Befehls
  - ▶ Ausführungszeit —"
  - ▶ Realisierung
    - ▶ fest verdrahtete Hardware
    - ▶ mikroprogrammiert

## Welche Befehle braucht man?

Befehlsklassen	Beispiele
<ul style="list-style-type: none"> <li>▶ arithmetische Operationen</li> <li>    logische Operationen</li> <li>    schiebe Operationen</li> <li>▶ Vergleichsoperationen</li> <li>▶ Datentransfers</li> <li>▶ Programm-Kontrollfluss</li> <li>▶ Maschinensteuerung</li> </ul>	<p>add, sub, inc, dec, mult, div</p> <p>and, or, xor</p> <p>shl, sra, srl, ror</p> <p>cmpeq, cmpgt, cmplt</p> <p>load, store, I/O</p> <p>jump, jmqeq, branch, call, return</p> <p>trap, halt, (interrupt)</p>

# CISC – Complex Instruction Set Computer

- ▶ 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
- ▶ Motivation
  - ▶ aus der Zeit der ersten Großrechner, 60er Jahre
  - ▶ Assemblerprogrammierung: Komplexität durch viele (mächtige) Befehle umgehen
- ▶ 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 universelle Register, keine Spezialregister
    - ▶ optimierende Compiler statt Assemblerprogrammierung
  - ▶ Beispiele: IBM 801, MIPS, SPARC, DEC Alpha, ARM
- Diskussion und Details später: „17 Computerarchitektur“

## Befehls-Decodierung

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

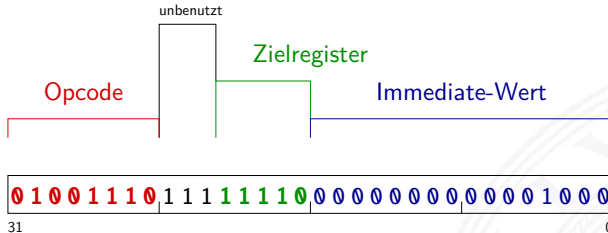
0	1	0	0	1	1	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

31
0

Wie soll die Hardware diesen Wert interpretieren?

- ▶ direkt in einer Tabelle nachschauen (Mikrocode-ROM)
  - ▶ Problem: Tabelle müsste  $2^{32}$  Einträge haben
  - ⇒ Aufteilung in Felder: Opcode und Operanden
  - ⇒ Decodierung über mehrere, kleine Tabellen
  - ⇒ unterschiedliche Aufteilung für unterschiedliche Befehle:
- Befehlsformate

# Befehlsformate



- ▶ Befehlsformat: 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: drei Beispielarchitekturen

- ▶ MIPS: Beispiel für 32-bit RISC Architekturen
  - ▶ alle Befehle mit 32-bit codiert
  - ▶ nur 3 Befehlsformate (R, I, J)
- ▶ D\*CORE: Beispiel für 16-bit Architektur
  - ▶ siehe RS-Praktikum (64-042) für Details
- ▶ Intel x86: Beispiel für CISC-Architekturen
  - ▶ irreguläre Struktur, viele Formate
  - ▶ mehrere Codierungen für einen Befehl
  - ▶ 1-Byte. . . 36-Bytes pro Befehl

## Befehlsformat: Beispiel MIPS

- ▶ festes Befehlsformat
  - ▶ alle Befehle sind 32 Bit lang
- ▶ Opcode-Feld ist immer 6-bit breit
  - ▶ codiert auch verschiedene Adressierungsmodi

wenige Befehlsformate

- ▶ R-Format
  - ▶ Register-Register ALU-Operationen
- ▶ I-/J-Format
  - ▶ Lade- und Speicheroperationen
  - ▶ alle Operationen mit unmittelbaren Operanden
  - ▶ Jump-Register
  - ▶ Jump-and-Link-Register



# MIPS: Übersicht

## „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
- ▶ 32 Register: 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: Registermodell

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

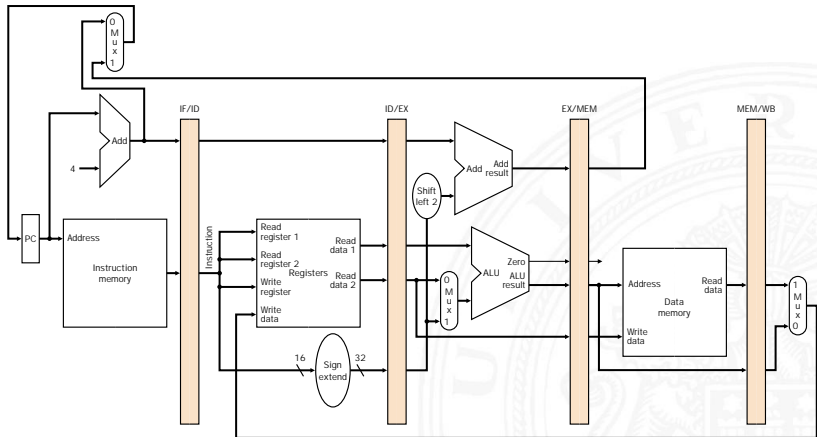
$R5 = -R5$	<code>sub R5, R0, R5</code>
$R4 = 0$	<code>add R4, R0, R0</code>
$R3 = 17$	<code>addi R3, R0, 17</code>
$\text{if } (R2 == 0)$	<code>bne R2, R0, label</code>
- ▶ keine separaten Statusflags
- ▶ Vergleichsoperationen setzen Zielregister auf 0 bzw. 1
 

$R1 = (R2 < R3)$	<code>slt R1, R2, R3</code>
------------------	-----------------------------

# MIPS: Befehlssatz

- ▶ Übersicht und Details: [PH11, PH12]  
David A. Patterson, John L. Hennessy: *Computer Organization and Design: The Hardware/Software Interface*
- ▶ dort auch hervorragende Erläuterung der Hardwarestruktur
- ▶ klassische fünf-stufige Befehlspipeline
  - ▶ Instruction-Fetch      Befehl holen
  - ▶ Decode                  Decodieren und Operanden holen
  - ▶ Execute                ALU-Operation oder Adressberechnung
  - ▶ Memory                Speicher lesen oder schreiben
  - ▶ Write-Back             Resultat in Register speichern

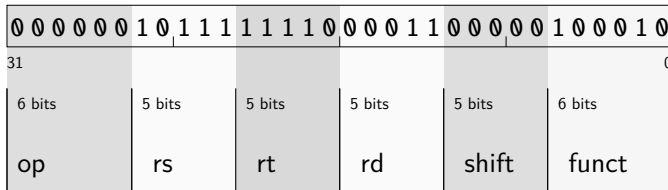
# MIPS: Hardwarestruktur



[PH12]

# MIPS: Befehlsformate

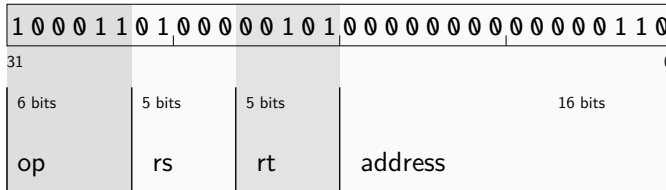
## Befehl im R-Format



- op: Opcode      Typ des Befehls      0 = „alu-op“
  - rs: source register 1      erster Operand      23 = „r23“
  - rt: source register 2      zweiter Operand      30 = „r30“
  - rd: destination register      Zielregister      3 = „r3“
  - shift: shift amount      (optionales Shiften)      0 = „0“
  - funct: ALU function      Rechenoperation      34 = „sub“
- ⇒ r3 = r23 - r30      sub r3, r23, r30

# MIPS: Befehlsformate

## Befehl im I-Format



- |       |                      |                 |           |
|-------|----------------------|-----------------|-----------|
| ► op: | Opcode               | Typ des Befehls | 35 = „lw“ |
| rs:   | base register        | Basisadresse    | 8 = „r8“  |
| rt:   | destination register | Zielregister    | 5 = „r5“  |
| addr: | address offset       | Offset          | 6 = „6“   |
- ⇒  $r5 = \text{MEM}[r8+6]$  lw r5, 6(r8)

## Befehlsformat: Beispiel M\*CORE

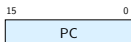
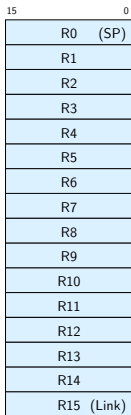
- ▶ 32-bit RISC Architektur, Motorola 1998
- ▶ besonders einfaches Programmiermodell
  - ▶ Program Counter     PC
  - ▶ 16 Universalregister   R0...R15
  - ▶ Statusregister        C („carry flag“)
  - ▶ 16-bit Befehle        (um Programmspeicher zu sparen)
- ▶ Verwendung
  - ▶ häufig in Embedded-Systems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

# D\*CORE

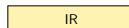
- ▶ ähnlich M\*CORE
- ▶ gleiches Registermodell, aber nur 16-bit Wortbreite
  - ▶ Program Counter     PC
  - ▶ 16 Universalregister   R0...R15
  - ▶ Statusregister         C („carry flag“)
- ▶ Subset der Befehle, einfachere Codierung
- ▶ vollständiger Hardwareaufbau in Hades verfügbar
  - ▶ [Hen] Hades Webdemo: [60-dcore/t3/chapter](http://60-dcore/t3/chapter)  
oder Simulator mit Assembler (alt)
    - ▶ [tams.informatik.uni-hamburg.de/publications/onlineDoc](http://tams.informatik.uni-hamburg.de/publications/onlineDoc)  
(winT3asm.exe / t3asm.jar)



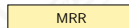
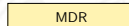
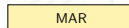
# D\*CORE: Registermodell



- 16 Universalregister
- Programmzähler
- 1 Carry-Flag



- Befehlsregister



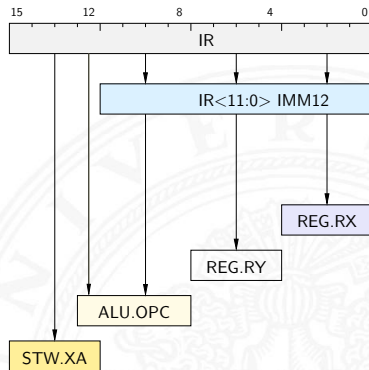
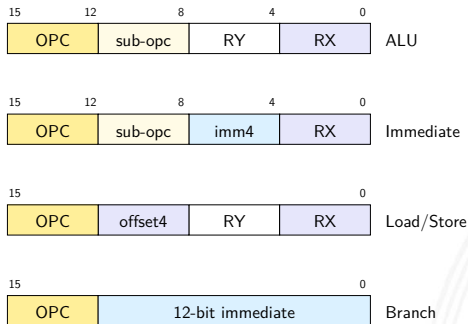
- Bus-Interface

► sichtbar für Programmierer: R0...R15, PC und C

## D\*CORE: Befehlssatz

<code>mov</code>	move register
<code>addu, addc</code>	Addition (ohne, mit Carry)
<code>subu</code>	Subtraktion
<code>and, or xor</code>	logische Operationen
<code>lsl, lsr, asr</code>	logische, arithmetische Shifts
<code>cmpe, cmpne, ...</code>	Vergleichsoperationen
<code>movi, addi, ...</code>	Operationen mit Immediate-Operanden
<code>ldw, stw</code>	Speicherzugriffe, load/store
<code>br, jmp</code>	unbedingte Sprünge
<code>bt, bf</code>	bedingte Sprünge
<code>jsr</code>	Unterprogrammaufruf
<code>trap</code>	Software interrupt
<code>rfi</code>	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
- ▶ Wie viele Operanden pro Befehl?
  - ▶ 0- / 1- / 2- / 3-Adress Maschinen
- ▶ Wie werden die Operanden adressiert?
  - ▶ immediate / direkt / indirekt / indiziert / autoinkrement / usw.

⇒ wichtige Unterscheidungsmerkmale für Rechnerarchitekturen

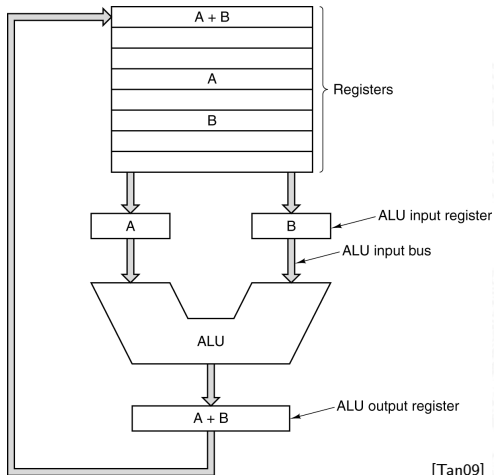
- ▶ Zugriff auf Hauptspeicher:  $\approx 100 \times$  langsamer als Registerzugriff
  - ▶ möglichst Register statt Hauptspeicher verwenden (!)
  - ▶ „load/store“-Architekturen

## Beispiel: Add-Befehl

- ▷ Rechner soll „rechnen“ können
- ▷ typische arithmetische Operation nutzt 3 Variablen  
Resultat, zwei Operanden:  $X = Y + Z$   
`add r2, r4, r5`     $\text{reg2} = \text{reg4} + \text{reg5}$   
 „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

## Beispiel: 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.)

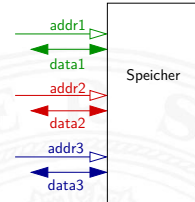


[Tan09]

# 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 ?

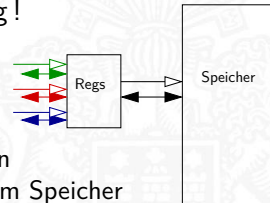
- sehr aufwendig, extrem teuer, trotzdem langsam

## ⇒ Register im Prozessor zur Zwischenspeicherung !

- Datentransfer zwischen Speicher und Registern

*Load*                       $\text{reg} = \text{MEM}[\text{addr}]$

*Store*     $\text{MEM}[\text{addr}] = \text{reg}$



- RISC: Rechenbefehle arbeiten *nur* mit Registern
- CISC: gemischt, Operanden in Registern oder im Speicher

## n-Adress Maschine $n = \{3 \dots 0\}$

- 3-Adress Format
  - ▶  $X = Y + Z$
  - ▶ sehr flexibel, leicht zu programmieren
  - ▶ Befehl muss 3 Adressen codieren
- 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



## Beispiel: n-Adress Maschine

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

Hilfsregister: T

### 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
```

## Beispiel: Stack-Maschine / 0-Adress Maschine

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

0-Adress-Maschine

push D

push E

mul

push C

add

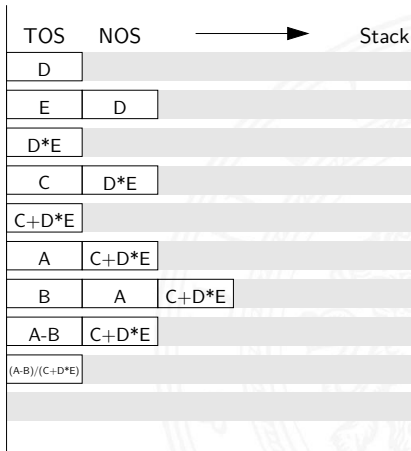
push A

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 (Basisadresse+offset)
  - ▶ ideal für Array- und Objektzugriffe
  - ▶ Hauptmodus in RISC-Rechnern (auch: „Versatz-Modus“)

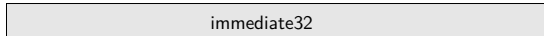
# Immediate Adressierung



1-Wort Befehl

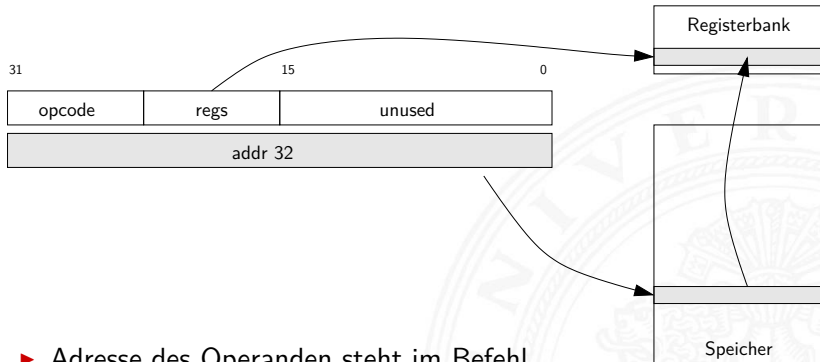


2-Wort Befehl



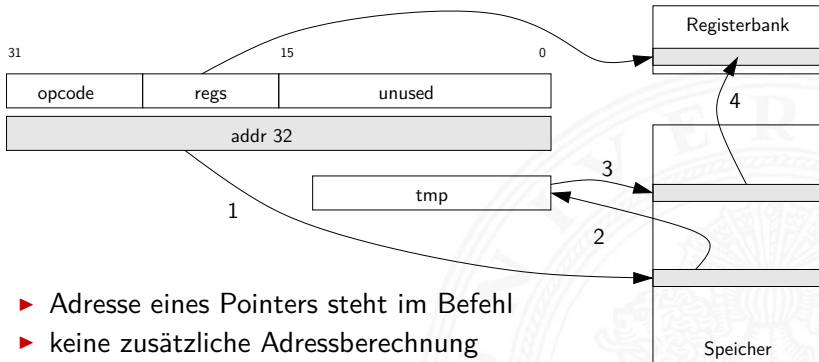
- ▶ Operand steht direkt im Befehl, kein zusätzlicher Speicherzugriff
- ▶ Länge des Operanden < (Wortbreite - Opcodebreite)
- ▶ Darstellung größerer Zahlenwerte
  - ▶ 2-Wort Befehle (x86)  
zweites Wort für Immediate-Wert
  - ▶ mehrere Befehle (MIPS, SPARC)  
z.B. obere/untere Hälfte eines Wortes
  - ▶ Immediate-Werte mit zusätzlichem Shift (ARM)

## Direkte Adressierung



- ▶ Adresse des Operanden steht im Befehl
- ▶ keine zusätzliche Adressberechnung
- ▶ ein zusätzlicher Speicherzugriff: z.B.  $R3 = \text{MEM}[\text{addr}32]$
- ▶ Adressbereich beschränkt, oder 2-Wort Befehl (wie Immediate)

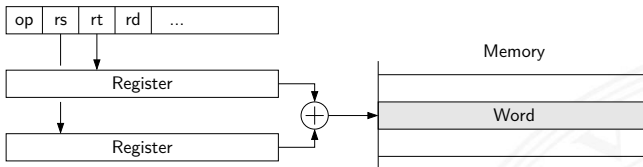
## Indirekte Adressierung



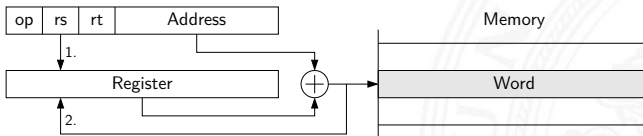
- ▶ Adresse eines Pointers steht im Befehl
- ▶ keine zusätzliche Adressberechnung
- ▶ zwei zusätzliche Speicherzugriffe:  
z.B.  $\text{tmp} = \text{MEM}[\text{addr32}] \quad \text{R3} = \text{MEM}[\text{tmp}]$
- ▶ typische CISC-Adressierungsart, viele Taktzyklen
- ▶ kommt bei RISC-Rechnern nicht vor

# Indizierte Adressierung

## Indexaddressing



## Updateaddressing



- ▶ indizierte Adressierung, z.B. für Arrayzugriffe
  - ▶  $\text{addr} = \langle \text{Sourceregister} \rangle + \langle \text{Basisregister} \rangle$
  - ▶  $\text{addr} = \langle \text{Sourceregister} \rangle + \text{offset};$   
 $\text{Sourceregister} = \text{addr}$



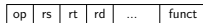
# Beispiel: MIPS Adressierungsarten

## 1. Immediate addressing



immediate

## 2. Register addressing



Registers

Register

register

## 3. Base addressing



Memory

Register

+

Byte Halfword Word

index + offset

## 4. PC-relative addressing



Memory

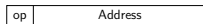
PC

+

Word

PC + offset

## 5. Pseudodirect addressing



Memory

PC

&

Word

$PC_{(31..28)} \& \text{address}$

## typische Adressierungsarten

welche Adressierungsarten / Varianten sind üblich?

- ▶ 0-Adress (Stack-) Maschine      Java virtuelle Maschine
- ▶ 1-Adress (Akkumulator) Maschine      8-bit Mikrokontroller  
einige x86 Befehle
- ▶ 2-Adress Maschine      16-bit Rechner  
einige x86 Befehle
- ▶ 3-Adress Maschine      32-bit RISC
- ▶ CISC Rechner unterstützen diverse Adressierungsarten
- ▶ RISC meistens nur indiziert mit Offset
- ▶ siehe [en.wikipedia.org/wiki/Addressing\\_mode](http://en.wikipedia.org/wiki/Addressing_mode)

# Intel x86-Architektur

- ▶ übliche Bezeichnung für die Intel-Prozessorfamilie
- ▶ von 8086, 80286, 80386, 80486, Pentium... Pentium-IV, Core 2, Core-i...
- ▶ eigentlich „IA-32“ (Intel architecture, 32-bit)... „IA-64“
- ▶ irreguläre Struktur: CISC
- ▶ historisch gewachsen: diverse Erweiterungen (MMX, SSE, ...)
- ▶ Abwärtskompatibilität: IA-64 mit IA-32 Emulation
- ▶ ab 386 auch wie reguläre 8-Register Maschine verwendbar

Hinweis: niemand erwartet, dass Sie sich alle Details merken

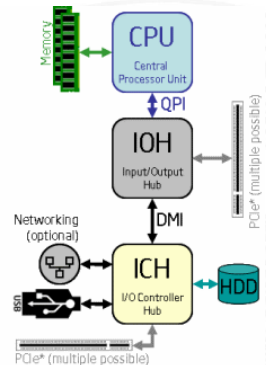
# Intel x86: Evolution

Chip	Datum	MHz	Transistoren	Speicher	Anmerkungen
4004	4/1971	0,108	2 300	640	erster Mikroprozessor auf einem Chip
8008	4/1972	0,108	3 500	16 KiB	erster 8-bit Mikroprozessor
8080	4/1974	2	6 000	64 KiB	„general-purpose“ CPU auf einem Chip
8086	6/1978	5–10	29 000	1 MiB	erste 16-bit CPU auf einem Chip
8088	6/1979	5–8	29 000	1 MiB	Einsatz im IBM-PC
80286	2/1982	8–12	134 000	16 MiB	„Protected-Mode“
80386	10/1985	16–33	275 000	4 GiB	erste 32-Bit CPU
80486	4/1989	25–100	1,2M	4 GiB	integrierter 8K Cache
Pentium	3/1993	60–233	3,1M	4 GiB	zwei Pipelines, später MMX
Pentium Pro	3/1995	150–200	5,5M	4 GiB	integrierter first und second-level Cache
Pentium II	5/1997	233–400	7,5M	4 GiB	Pentium Pro plus MMX
Pentium III	2/1999	450–1 400	9,5–44M	4 GiB	SSE-Einheit
Pentium IV	11/2000	1 300–3 600	42–188M	4 GiB	Hyperthreading
Core-2	5/2007	1 600–3 200	143–410M	4 GiB	64-bit Architektur, Mehrkernprozessoren
Core-i...	11/2008	2,500–3,600	> 700M	64 GiB	Taktanpassung (Turbo Boost)
...					

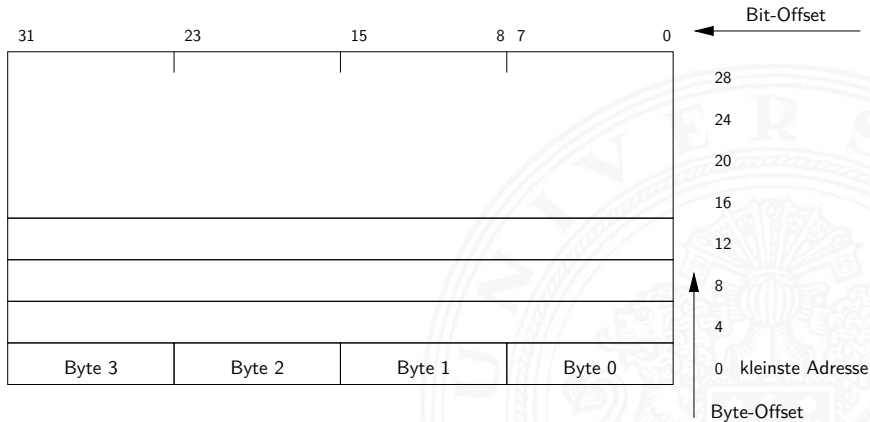
## Beispiel: Core i7-960 Prozessor

Taktfrequenz	bis 3,46 GHz
Anzahl der Cores	4 (× 2 Hyperthreading)
QPI Durchsatz (quick path interconnect)	4,8 GT/s
Bus Interface	64 Bits
L1 Cache	4 × 32 kB I + 32 kB D
L2 Cache	4 × 256 kB (I+D)
L3 Cache	8192 kB (I+D)
Prozess	45 nm
Versorgungsspannung	0,8 - 1,375V
Wärmeabgabe	~ 130 W
Performance (SPECint 2006)	~ 38

Quellen: [ark.intel.com](http://ark.intel.com), [www.spec.org](http://www.spec.org)



# x86: Speichermodell



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

## x86: Speichermodell (cont.)

- Speicher voll byte-adressierbar
- misaligned Zugriffe langsam
- Beispiel

Word at Address BH  
Contains FE06H

Byte at Address 9H  
Contains 1FH

Word at Address 6H  
Contains 230BH

Word at Address 2H  
Contains 74CBH

Word at Address 1H  
Contains CB31H

	EH
7AH	DH
FEH	CH
06H	BH
36H	AH
1FH	9H
A4H	8H
23H	7H
0BH	6H
	5H
	4H
74H	3H
CBH	2H
31H	1H
	0H

Doubleword at Address AH  
Contains 7AFE0636H

Quadword at Address 6H  
Contains 7AFE06361FA4230BH

# x86: Register

31	15	0
EAX	AX	AH AL
ECX	CX	CH CL
EDX	DX	DH DL
EBX	BX	BH BL
ESP	SP	
EBP	BP	
ESI	SI	
EDI	DI	
	CS	
	SS	
	DS	
	ES	
	FS	
	GS	
EIP	IP	
EFLAGS		

accumulator

count: String, Loop

data, multiply/divide

base addr

stackptr

base of stack segment

index, string src

index, string dst

code segment

stack segment

data segment

extra data segment

PC

status

8086

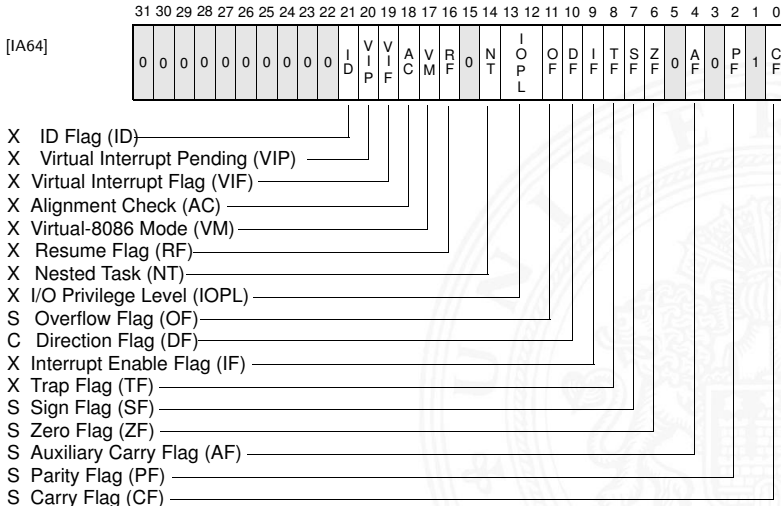
Exx ab 386

79	0
FPR0	
FPR7	

FP Status



# x86: EFLAGS Register



# x86: Datentypen

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

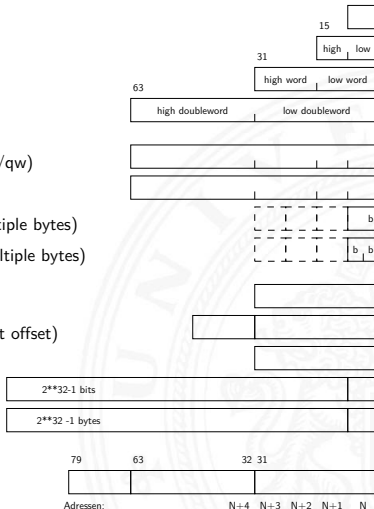
(16 bit segment + 32 bit offset)

bit field

bit string

byte string

float / double / extended



## x86: Befehlssatz

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

- CISC: zusätzlich diverse Ausnahmen/Spezialfälle

## x86: Befehlsformate

- ▶ 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...36 Bytes
- ⇒ extrem aufwendige Decodierung
- ⇒ CISC – **C**omplex **I**nstruction **S**et **C**omputer

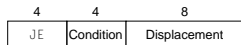
## x86: Befehlsformat-Modifizier („prefix“)

- ▶ alle Befehle können mit Modifiern ergänzt werden

segment override	Adresse aus angewähltem Segmentregister
address size	Umschaltung 16/32-bit Adresse
operand size	Umschaltung 16/32-bit Operanden
repeat	Stringoperationen: für alle Elemente
lock	Speicherschutz bei Multiprozessorsystemen

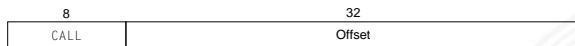
# x86 Befehlskodierung: Beispiele

a. JE EIP + displacement

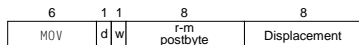


[PH12]

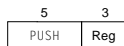
b. CALL



c. MOV EBX, [EDI + 45]



d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



- ▶ 1 Byte... 36 Bytes
- ▶ vollkommen irregulär
- ▶ w: Auswahl 16/32 bit

## x86 Befehlskodierung: Beispiele (cont.)

Instruction	Function
JE name	If equal (CC) EIP = name; $EIP - 128 \leq \text{name} < EIP + 128$
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

[PH12]

# x86: Assembler-Beispiel print(...)

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	cmpl \$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	cmpl \$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 main(...)

addr	opcode	assembler	c quellcode
-----			
		.Lfel:	
		.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	cmpl \$0, .LC0	
	00000000		
0040	741A	je .L26	
0042	89F6	.align 4	
		.L24:	
0044	A100000000	movl stdout,%eax	
0049	50	pushl %eax	
004a	0FBEO3	movsbl (%ebx),%eax	
004d	50	pushl %eax	
004e	E8FCFFFFFF	call _IO_putc	
0053	43	incl %ebx	
0054	83C408	addl \$8,%esp	
0057	803B00	cmpl \$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	

# Literatur

[BO11] R.E. Bryant, D.R. O'Hallaron:

*Computer systems – A programmers perspective.*

2nd edition, Pearson, 2011. ISBN 0–13–713336–7

Grafiken z.T. aus: Foliensatz

[Tan06] A.S. Tanenbaum: *Computerarchitektur – Strukturen, Konzepte, Grundlagen.* 5. Auflage, Pearson Studium, 2006.  
ISBN 3–8273–7151–1

[Tan09] A.S. Tanenbaum: *Structured Computer Organization.*  
5th rev. edition, Pearson International, 2009.  
ISBN 0–13–509405–4

## Literatur (cont.)

[PH11] D.A. Patterson, J.L. Hennessy: *Rechnerorganisation und -entwurf – Die Hardware/Software-Schnittstelle*.  
4. Auflage, Oldenbourg Verlag, 2011.  
ISBN 978-3-486-58190-3

[PH12] D.A. Patterson, J.L. Hennessy: *Computer Organization and Design – The Hardware/Software Interface*.  
4th rev. edition, Morgan Kaufmann Publishers, 2012.  
ISBN 978-0-12-374750-1  
Grafiken z.T. aus: Foliensatz

## Literatur (cont.)

[Hen] N. Hendrich: *HADES — HAmBurg DEsign System*.  
Universität Hamburg, FB Informatik, Lehrmaterial.  
[tams.informatik.uni-hamburg.de/applets/hades](http://tams.informatik.uni-hamburg.de/applets/hades)

[IA64] *Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 1: Basic Architecture*.  
Intel Corp.; Santa Clara, CA.  
[www.intel.com](http://www.intel.com)