



Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG

MIN-Fakultät  
Fachbereich Informatik



# 64-040 Modul InfB-RSB

## Rechnerstrukturen und Betriebssysteme

[https://tams.informatik.uni-hamburg.de/  
lectures/2023ws/vorlesung/rsb](https://tams.informatik.uni-hamburg.de/lectures/2023ws/vorlesung/rsb)

– Kapitel 12 –

Andreas Mäder



Universität Hamburg  
Fakultät für Mathematik, Informatik und Naturwissenschaften  
Fachbereich Informatik

**Technische Aspekte Multimodaler Systeme**

Wintersemester 2023/2024



## Instruction Set Architecture

Speicherorganisation

Befehlssatz

Befehlsformate

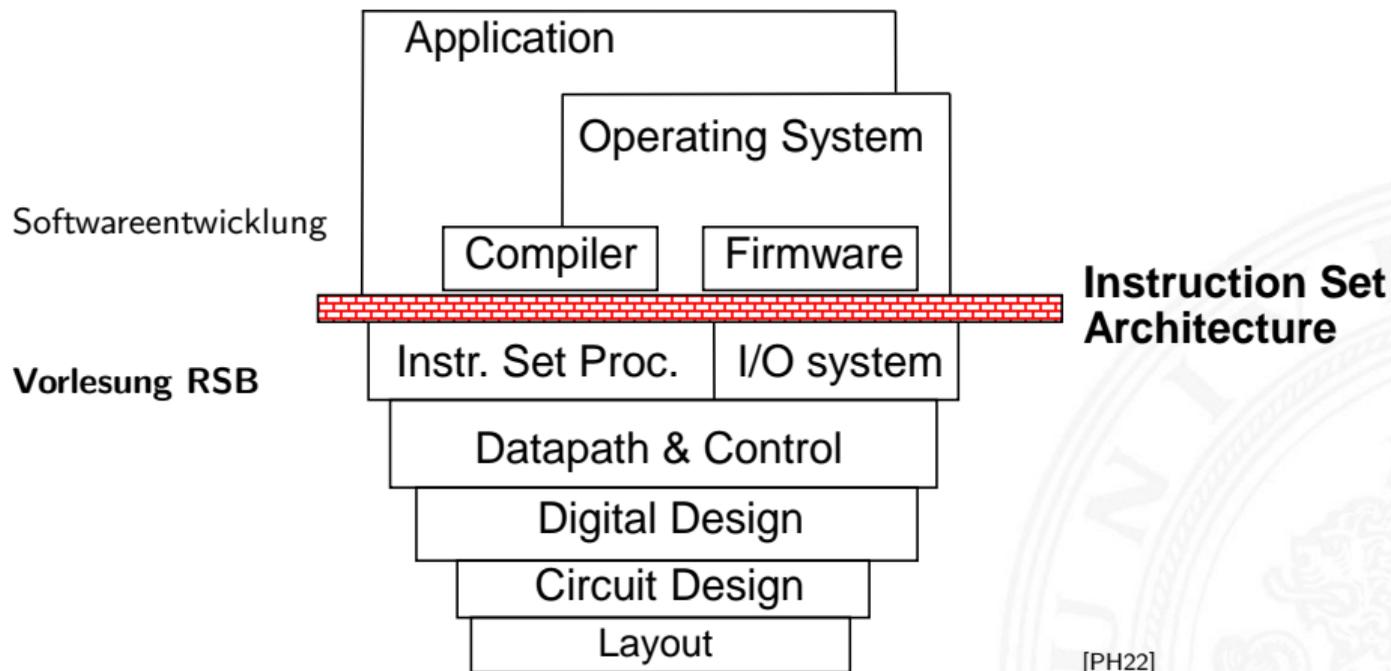
Adressierungsarten

Intel x86-Architektur

Befehlssätze

Literatur



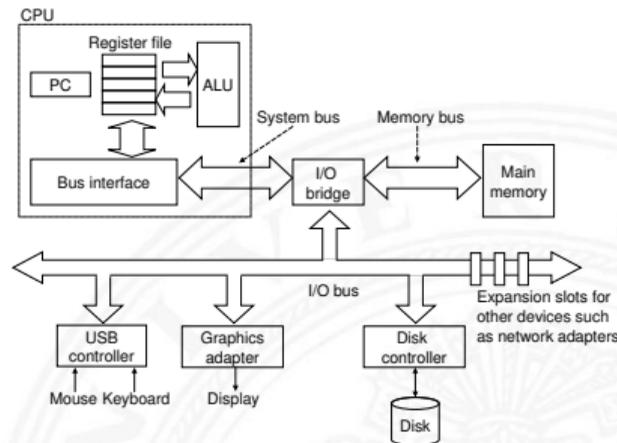


[PH22]

## ISA – Instruction **S**et **A**rchitecture

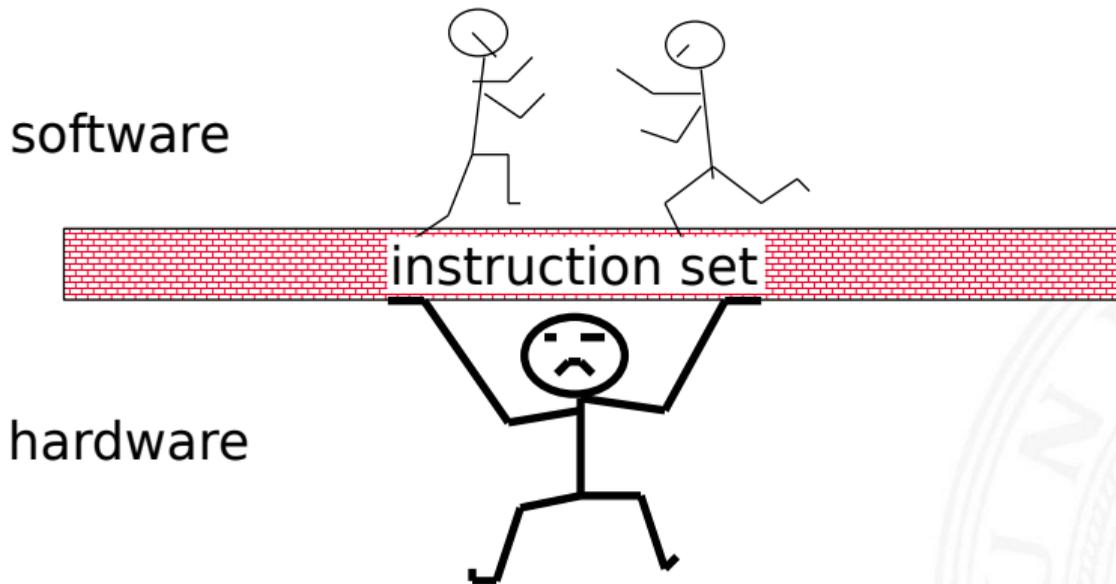
⇒ alle für den Programmierer sichtbaren Attribute einer Rechnerarchitektur

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





- ▶ Befehlssatz: die zentrale Schnittstelle



[PH22]



- ▶ 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	1 $\mu$ C	1 $\mu$ C	1 ASIC	1 $\mu$ P, ASIP	DSPs	1 $\mu$ P, 3 DSP	1 $\mu$ P, DSP	$\approx$ 100 $\mu$ C, $\mu$ P, DSP	1 $\mu$ P, ASIP
[bit]	4 ... 32	8	—	16 ... 32	32	32	32	8 ... 64	16 ... 32
Speicher	1 K ... 1 M	< 8 K	< 1 K	1 ... 64 M	1 ... 64 M	< 512 M	8 ... 64 M	1 K ... 10 M	< 64 M
Netzwerk	cardIO	—	RS-232	diverse	GSM	MIDI	V.90	CAN ...	I <sup>2</sup> C ...
Echtzeit	—	—	soft	soft	hard	soft	hard	hard	hard
Sicherheit	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 ...
- ▶ viele, sehr unterschiedliche Anforderungen: Durchsatz, Geschwindigkeit, Echtzeit, Sicherheit, Zuverlässigkeit, Leistungsaufnahme, Abwärme, Störstrahlung, Temperaturbereich, Größe, Kosten etc.



- ▶ Adressierung
- ▶ Wortbreite, 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 (z.B. MESI-Protokoll)

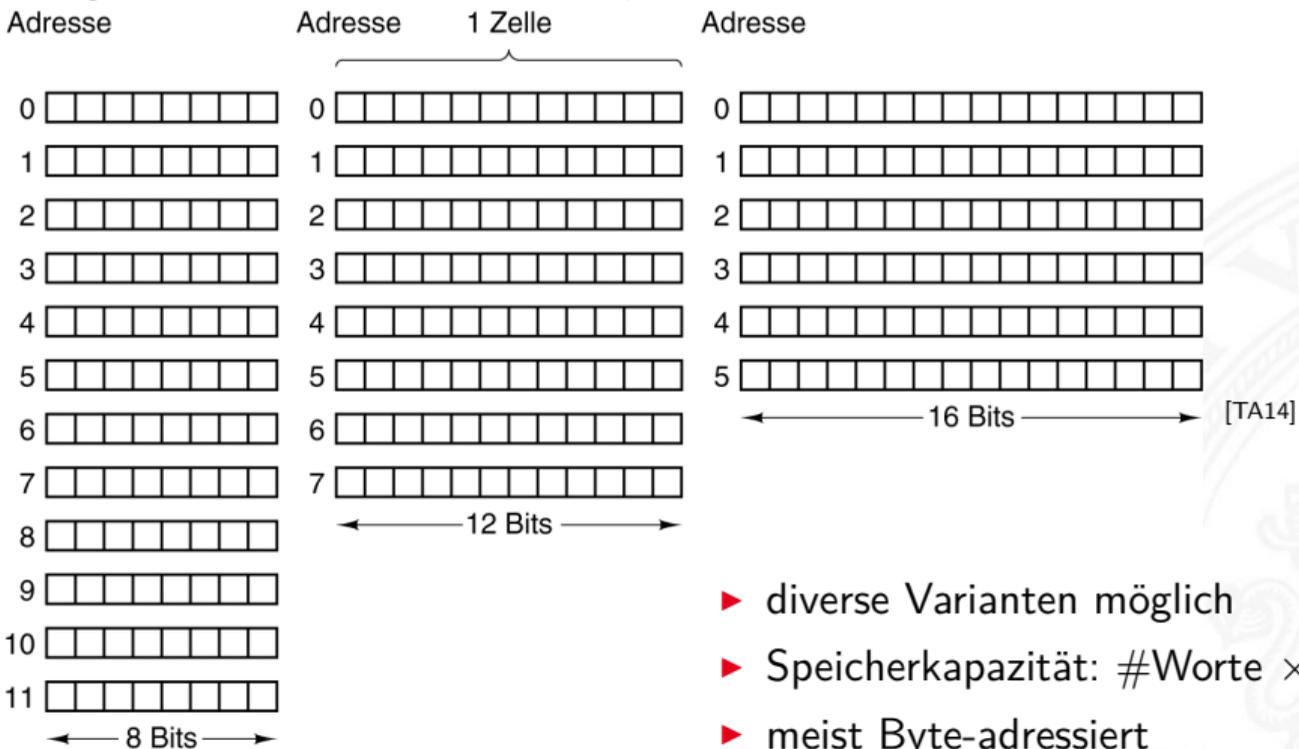


- ▶ Abspeichern von Zahlen, Zeichen, Strings?
  - ▶ kleinster Datentyp üblicherweise ein Byte (8-bit)
  - ▶ andere Daten als Vielfache: 16-bit, 32-bit, 64-bit ...
  
- ▶ Organisation und Adressierung des Speichers?
  - ▶ Adressen typisch in Bytes angegeben
  - ▶ erlaubt Adressierung einzelner ASCII-Zeichen usw.
  
- ▶ aber Maschine/Prozessor arbeitet wortweise
- ▶ Speicher daher ebenfalls wortweise aufgebaut
- ▶ typischerweise 32-bit oder 64-bit





## 3 Organisationsformen eines 96-bit Speichers: $12 \times 8$ , $8 \times 12$ , $6 \times 16$ Bits



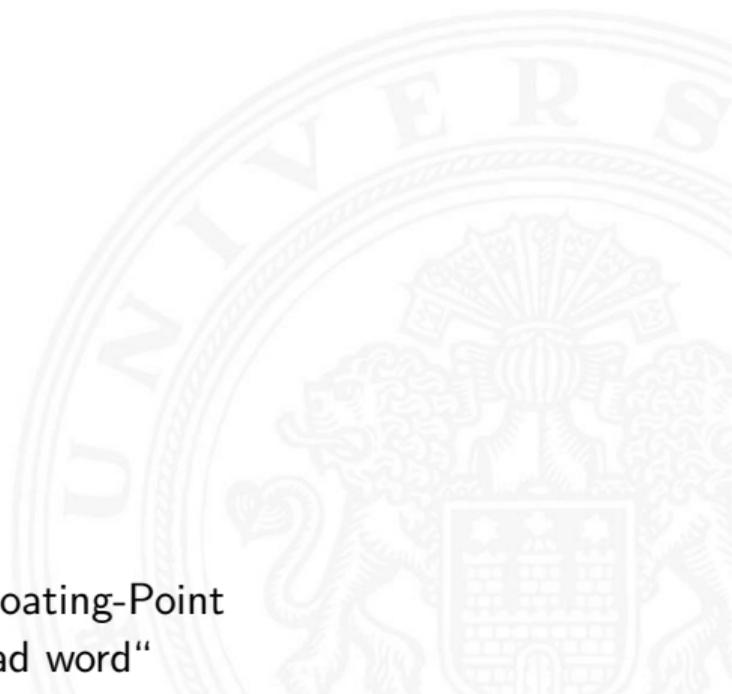
- ▶ diverse Varianten möglich
- ▶ Speicherkapazität:  $\#Worte \times \#Bits/Wort$
- ▶ meist Byte-adressiert



## ► Speicherwortbreiten historisch wichtiger Computer

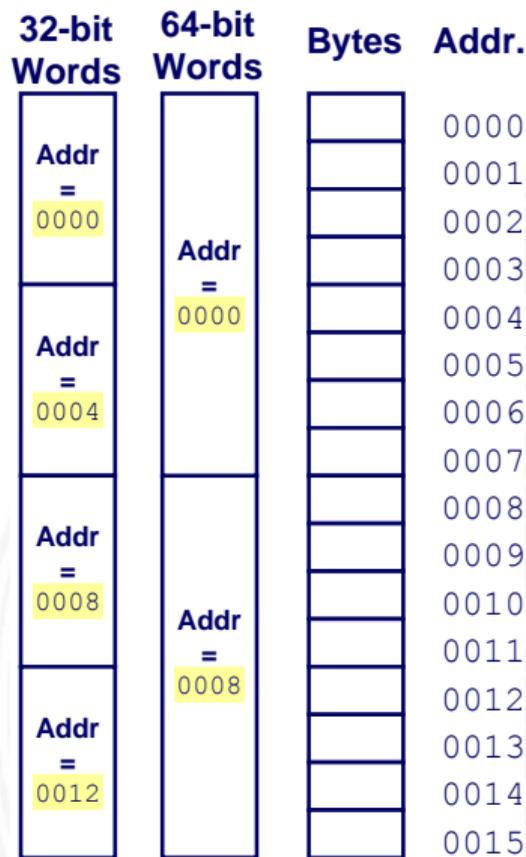
Computer	Bits/Speicherzelle
Burroughs B1700	1
IBM PC	8
DEC PDP-8	12
IBM 1130	16
DEC PDP-15	18
XDS 940	24
Electrologica X8	27
XDS Sigma 9	32
Honeywell 6180	36
CDC 3600	48
CDC Cyber	60

- heute 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“



# Wort-basierte Organisation des Speichers

- ▶ Speicher Wort-orientiert
- ▶ Adressierung Byte-orientiert
  - ▶ die Adresse des ersten Bytes im Wort
  - ▶ Adressen aufeinanderfolgender Worte unterscheiden sich um 4 (32-bit Wort) oder 8 (64-bit)
- ▶ Adressen normalerweise Vielfache der Wortlänge
- ▶ verschobene Adressen „in der Mitte“ eines Worts oft unzulässig



- ▶ gängige Prozessoren unterstützen mehrere Datentypen
- ▶ entsprechend der elementaren Datentypen in C, Java ...
- ▶ `void*` ist ein **Pointer** (Referenz, Speicheradresse)
- ▶ Beispiel für die Anzahl der Bytes:

C Datentyp	DEC Alpha	typ. 32-bit	Intel IA-32 (x86)
int	4	4	4
long int	8	4	4
char	1	1	1
short	2	2	2
float	4	4	4
double	8	8	8
long double	8	8	10/12
void *	8	4	4

# Wdh. Datentypen auf Maschinenebene (cont.)

## Abhängigkeiten (!)

- ▶ Prozessor
- ▶ Betriebssystem
- ▶ Compiler

segment word size compiler	16 bit			32 bit					64 bit				
	Microsoft	Borland	Watcom	Microsoft	Intel Windows	Borland	Watcom	Gnu, Clang	Intel Linux	Microsoft	Intel Windows	Gnu, Clang	Intel Linux
bool	2	1	1	1	1	1	1	1	1	1	1	1	1
char	1	1	1	1	1	1	1	1	1	1	1	1	1
wchar_t		2		2	2	2	2	2	2	2	2	4	4
short int	2	2	2	2	2	2	2	2	2	2	2	2	2
int	2	2	2	4	4	4	4	4	4	4	4	4	4
long int	4	4	4	4	4	4	4	4	4	4	4	8	8
int64_t				8	8			8	8	8	8	8	8
enum (typical)	2	2	1	4	4	4	4	4	4	4	4	4	4
float	4	4	4	4	4	4	4	4	4	4	4	4	4
double	8	8	8	8	8	8	8	8	8	8	8	8	8
long double	10	10	8	8	16	10	8	12	12	8	16	16	16
__m64				8	8			8	8		8	8	8
__m128				16	16			16	16	16	16	16	16
__m256				32	32			32	32	32	32	32	32
__m512				64	64			64	64	64	64	64	64
pointer	2	2	2	4	4	4	4	4	4	8	8	8	8
far pointer	4	4	4										
function pointer	2	2	2	4	4	4	4	4	4	8	8	8	8
data member pointer (min)	2	4	6	4	4	8	4	4	4	4	4	8	8
data member pointer (max)		4	6	12	12	8	12	4	4	12	12	8	8
member function pointer (min)	2	12	6	4	4	12	4	8	8	8	8	16	16
member function pointer (max)		12	6	16	16	12	16	8	8	24	24	16	16

[www.agner.org/optimize/calling\\_conventions.pdf](http://www.agner.org/optimize/calling_conventions.pdf)



- ▶ *Wie sollen die Bytes innerhalb eines Wortes angeordnet werden?*
- ▶ Speicher wort-basiert  $\Leftrightarrow$  Adressierung byte-basiert

Zwei Möglichkeiten / Konventionen:

- ▶ **Big Endian:** Sun, Mac usw.  
das MSB (*most significant byte*) hat die kleinste Adresse  
das LSB (*least significant byte*) hat die höchste –"
- ▶ **Little Endian:** Alpha, x86  
das MSB hat die höchste, das LSB die kleinste Adresse

satirische Referenz auf Gulliver's Reisen (Jonathan Swift)

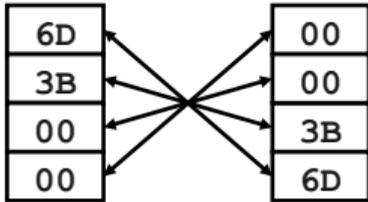


# Byte-Order: Beispiel

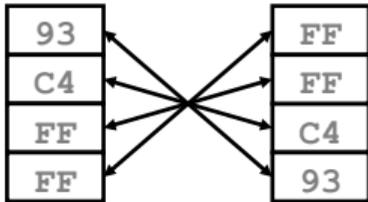
```
int A = 15213;  
int B = -15213;  
long int C = 15213;
```

Dezimal: 15213  
Binär: 0011 1011 0110 1101  
Hex: 3 B 6 D

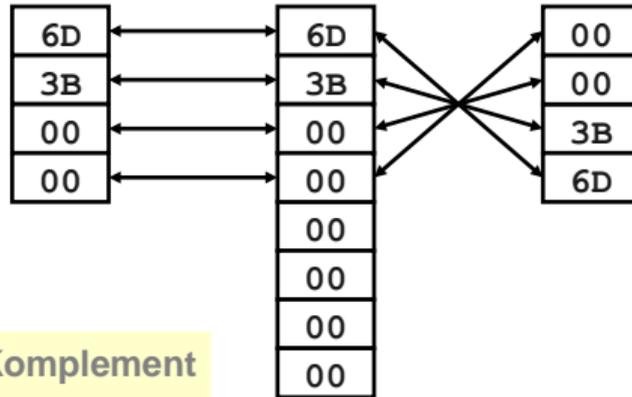
Linux/Alpha A Sun A



Linux/Alpha B Sun B



Linux c Alpha c Sun c



2-Komplement  
Big Endian  
Little Endian



[BO15]

# Byte-Order: Beispiel Datenstruktur

```
/* JimSmith.c - example record for byte-order demo */  
  
typedef struct employee {  
    int    age;  
    int    salary;  
    char   name[12];  
} employee_t;  
  
static employee_t jimmy = {  
    23,                // 0x0017  
    50000,             // 0xc350  
    "Jim Smith",      // J=0x4a i=0x69 usw.  
};
```



# Byte-Order: Beispiel x86 und SPARC

```
tams12> objdump -s JimSmith.x86.o
JimSmith.x86.o:      file format elf32-i386

Contents of section .data:
 0000 17000000 50c30000 4a696d20 536d6974  ....P...Jim Smit
 0010 68000000                                     h...

tams12> objdump -s JimSmith.sparc.o
JimSmith.sparc.o:   file format elf32-sparc

Contents of section .data:
 0000 00000017 0000c350 4a696d20 536d6974  ....PJim Smit
 0010 68000000                                     h...
```



- ▶ Byte-Order muss bei Datenübertragung zwischen Rechnern berücksichtigt und eingehalten werden
  
- ▶ Internet-Protokoll (IP) nutzt ein Big Endian Format
- ⇒ auf x86-Rechnern müssen alle ausgehenden und ankommenden Datenpakete umgewandelt werden
  
- ▶ zugehörige Hilfsfunktionen / Makros in `netinet/in.h`
  - ▶ inaktiv auf Big Endian, **byte-swapping** auf Little Endian
  - ▶ `ntohl(x)`: network-to-host-long
  - ▶ `htons(x)`: host-to-network-short
  - ▶ ...





Linux: `/usr/include/bits/byteswap.h`

(distributionsabhängig)

```
...
/* Swap bytes in 32 bit value.  */
#define __bswap_32(x) \
    (((x) & 0xff000000) >> 24) | (((x) & 0x00ff0000) >> 8) |\
    (((x) & 0x0000ff00) << 8) | (((x) & 0x000000ff) << 24))
...
```

Linux: `/usr/include/netinet/in.h`

```
...
# if __BYTE_ORDER == __LITTLE_ENDIAN
#   define ntohl(x) __bswap_32 (x)
#   define ntohs(x) __bswap_16 (x)
#   define htonl(x) __bswap_32 (x)
#   define htons(x) __bswap_16 (x)
# endif
...
```



- ▶ Programm gibt Daten byteweise aus
- ▶ C-spezifische Typ- (Pointer-) Konvertierung
- ▶ Details: Bryant, O'Hallaron [BO15], 2.1.4 (Abb. 2.3, 2.4)

```
void show_bytes( byte_pointer start, int len ) {
    int i;
    for( i=0; i < len; i++ ) {
        printf( " %.2x", start[i] );
    }
    printf( "\n" );
}

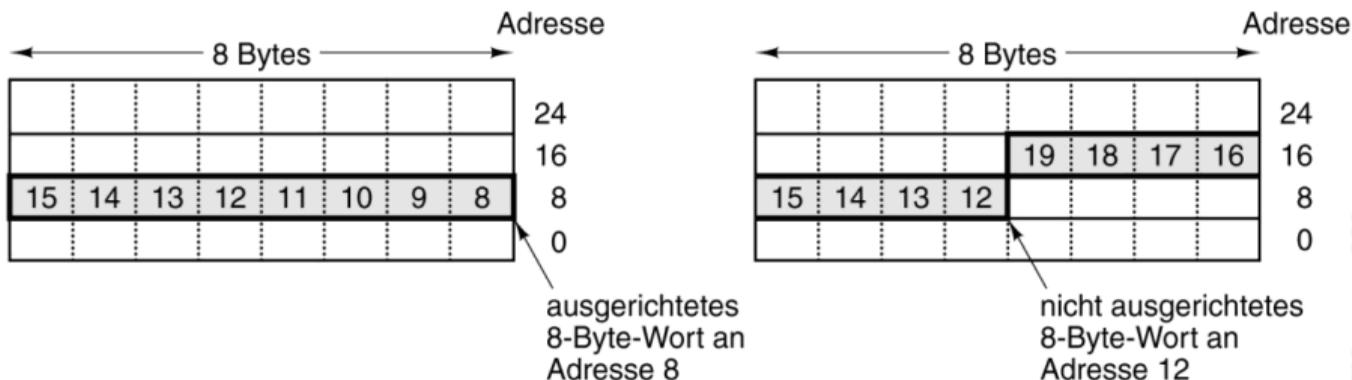
void show_double( double x ) {
    show_bytes( (byte_pointer) &x, sizeof( double ) );
}

...

```



# „Misaligned“ Zugriff



[TA14]

- ▶ Beispiel: 8-Byte-Wort in Little Endian Speicher
  - ▶ „aligned“ bezüglich Speicherwort
  - ▶ „non aligned“ an Byte-Adresse 12
- ▶ Speicher wird (meistens) Byte-weise adressiert, aber Zugriffe lesen/schreiben jeweils ein ganzes Speicherwort
- ⇒ was passiert bei „krummen“ (*misaligned*) Adressen?
  - ▶ automatische Umsetzung auf mehrere Zugriffe (x86)
  - ▶ Programmabbruch (SPARC)

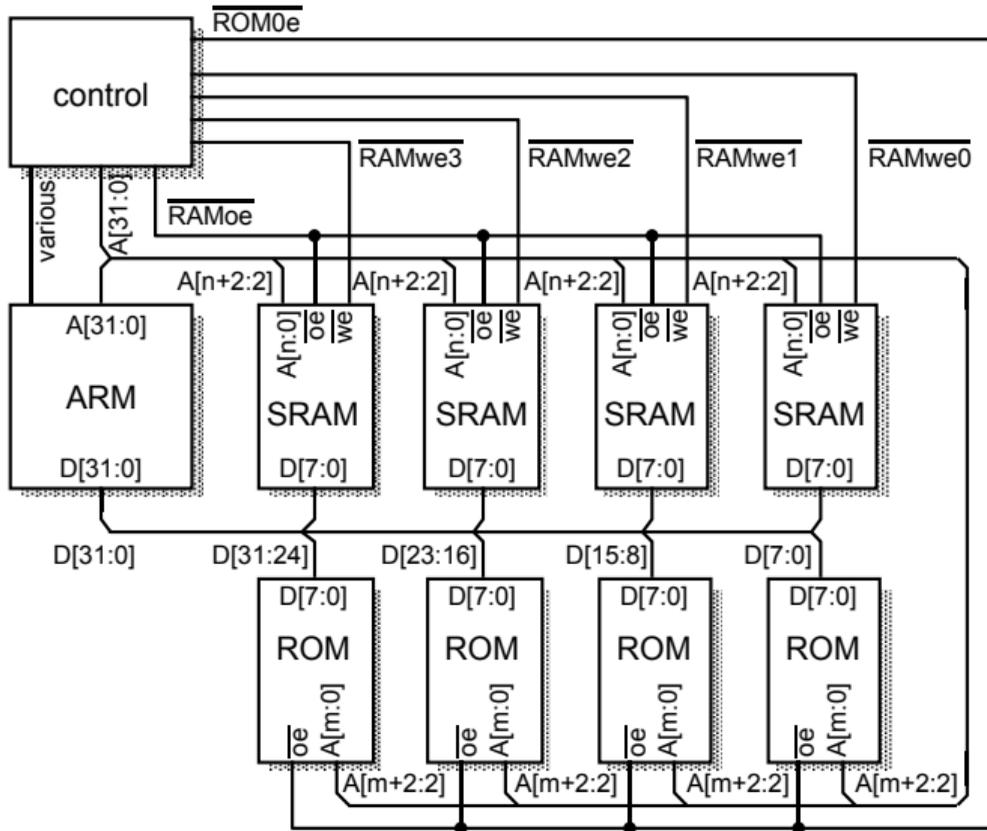


- ▶ CPU kann im Prinzip alle möglichen Adressen ansprechen
- ▶ in der Regel: **kein voll ausgebauter Speicher**  
32 bit Adresse entsprechen 4 GiB Hauptspeicher, 64 bit ...

⇒ „Memory Map“

- ▶ Adresdecoder als Hardwareeinheit
  - ▶ Aufteilung in *read-write*- und *read-only*-Bereiche
  - ▶ ROM zum Booten notwendig
  - ▶ Read-only in *eingebetteten Systemen*: Firmware, OS, Programme
  - ▶ zusätzliche Speicherbereiche für „memory mapped“ I/O
- ▶ Adressabbildung in Betriebssystemen (Windows, Linux etc.)
  - ▶ Zuordnung von Adressen zu „realem“ Speicher
  - ▶ alle Hardwarekomponenten (+ Erweiterungskarten)  
Ein-/Ausgabekanäle  
Interrupts
  - ▶ Verwaltung über *Treiber*

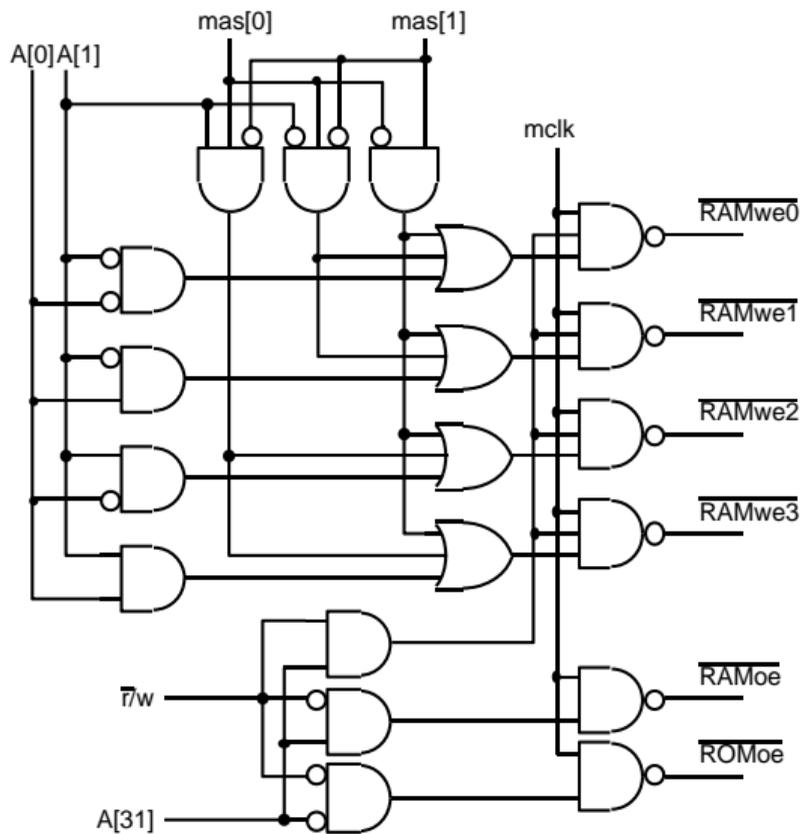
# Adressabbildung Hardware: ARM



32-bit ARM Proz.  
4 × 8-bit SRAMs  
4 × 8-bit ROMs

[Fur00]

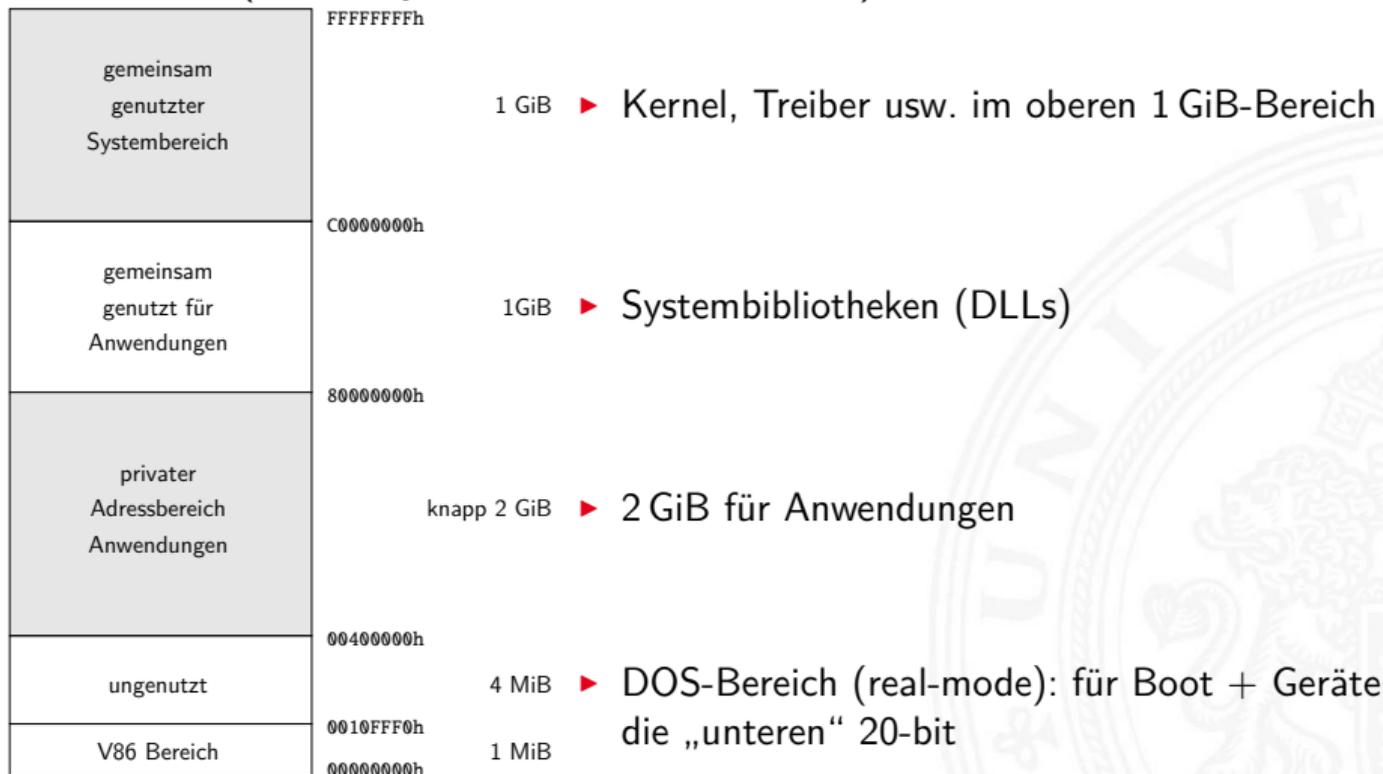
# Adressabbildung Hardware: ARM (cont.)



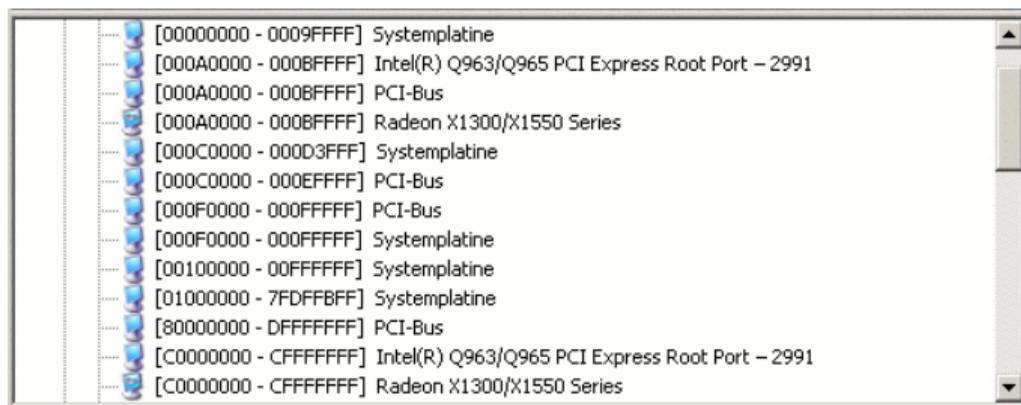
Adresdecoder Hardware

[Fur00]

## ► Windows 95 (32-bit System: 4 GiB Adressraum)



- ▶ Windows 95 (32-bit System)
  - ▶ 32-bit Adressen, 4 GiByte Adressraum
  - ▶ Aufteilung 2 GiB für Programme, obere 1+1 GiB für Windows
  - ▶ unabhängig von physikalischem Speicher
  - ▶ Beispiel der Zuordnung, diverse Bereiche für I/O reserviert

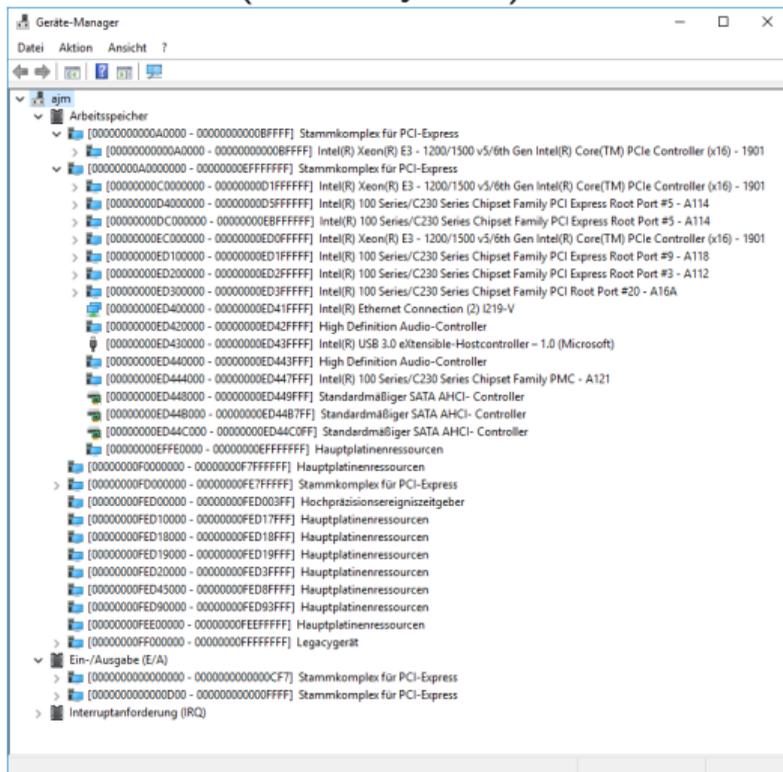


The screenshot shows a list of memory address ranges and their corresponding hardware components. The ranges are listed in hexadecimal format, and the components are listed in German. The components include Systemplatine, Intel(R) Q963/Q965 PCI Express Root Port - 2991, PCI-Bus, and Radeon X1300/X1550 Series.

[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 - 000FFFFFF]	PCI-Bus
[000F0000 - 000FFFFFF]	Systemplatine
[00100000 - 00FFFFFF]	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

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

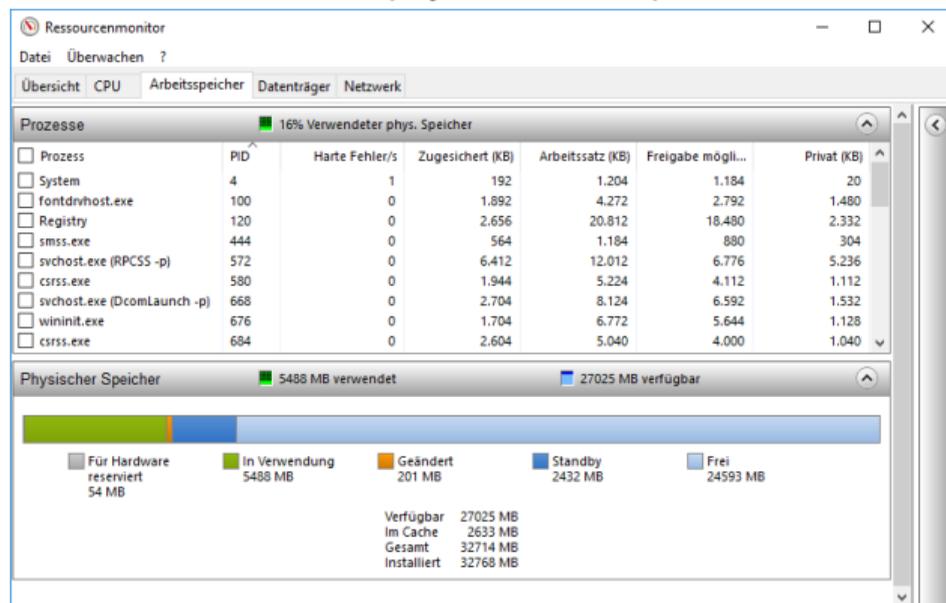
## ► Windows 10 (64-bit System)



⇒ Adressbereiche

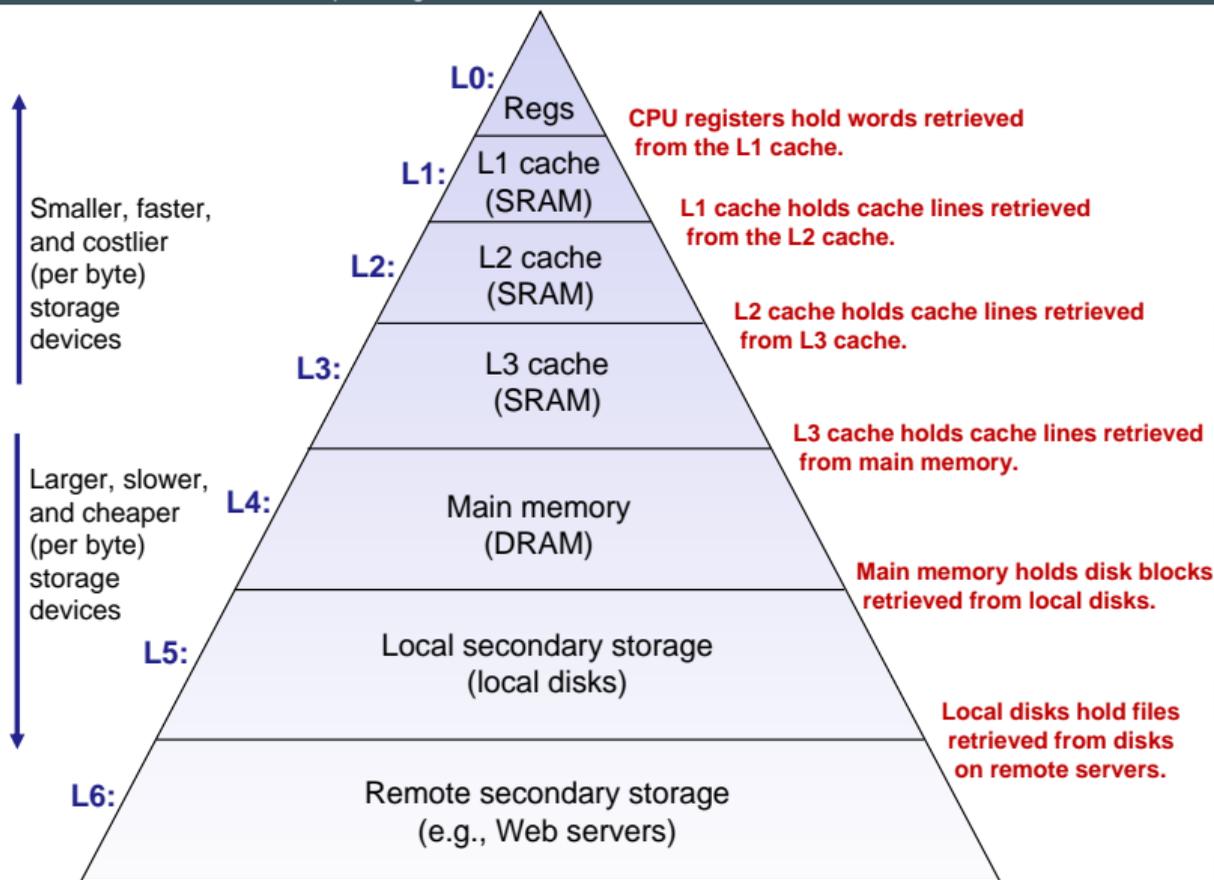
mehrstufige Abbildung:

1. alle Hardwarekomponenten, Ein-/Ausgabeeinheiten und Interrupts  $\Rightarrow$  Adressbereiche
2. Adressbereiche  $\Rightarrow$  physikalischer Speicher



► Linux siehe: free, vmstat, dmidecode, hwinfo, top, lshw, ...

# Speicherhierarchie







- ▶ Speicher ist nicht unbegrenzt
  - ▶ muss zugeteilt und verwaltet werden
  - ▶ viele Anwendungen werden vom Speicher dominiert
- ▶ besondere Sorgfalt beim Umgang mit Speicher
  - ▶ Fehler sind besonders gefährlich und schwer zu Debuggen
  - ▶ Auswirkungen sind sowohl zeitlich als auch räumlich entfernt
- ▶ Speicherleistung ist nicht konstant  
von Größe des Speichers, Datenstrukturen und -zugriffen abhängig
- ▶ Wechselwirkungen: Speichersystem  $\Leftrightarrow$  Programme
  - ▶ Cacheorganisation und Virtual-Memory beeinflussen Performanz/Programmleistung
  - ▶ Anpassung des Programms an das Speichersystem kann Laufzeitverhalten verbessern

→ siehe *14 Rechnerarchitektur II – Speicherhierarchie*

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





- ▶ 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
- ▶ benötigte Register

## Steuerwerk

PC	Program Counter	Adresse des Befehls
IR	Instruction Register	aktueller Befehl

## Rechenwerk

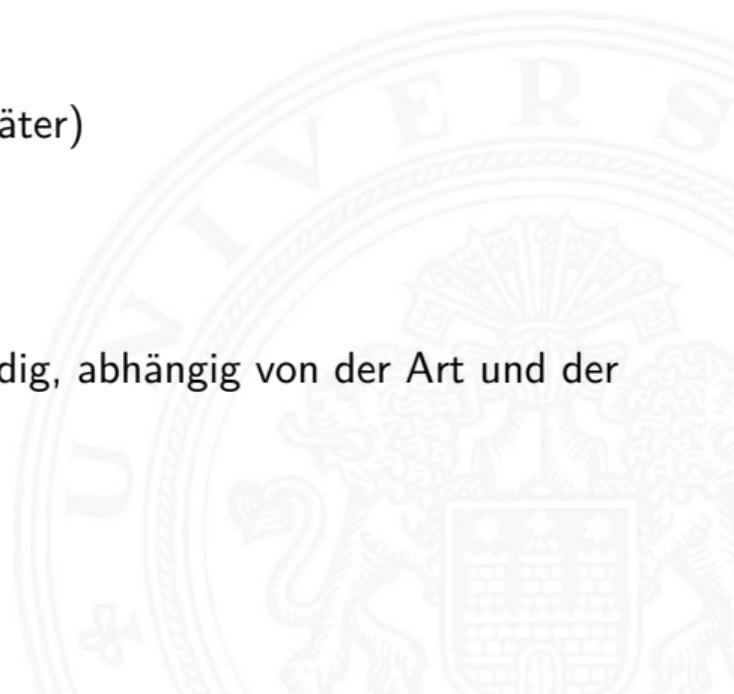
R0...R31	Registerbank	Rechenregister (Operanden)
ACC	Akkumulator	= Minimalanforderung



# 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 (ggf. auch später)
  - ▶ 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 der 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

## Operand Fetch

- ▶ wird meist zu anderen Phasen hinzugezählt
- RISC: Teil von *Instruction Decode*
- CISC: –"– *Instruction Execute*
1. Operanden holen

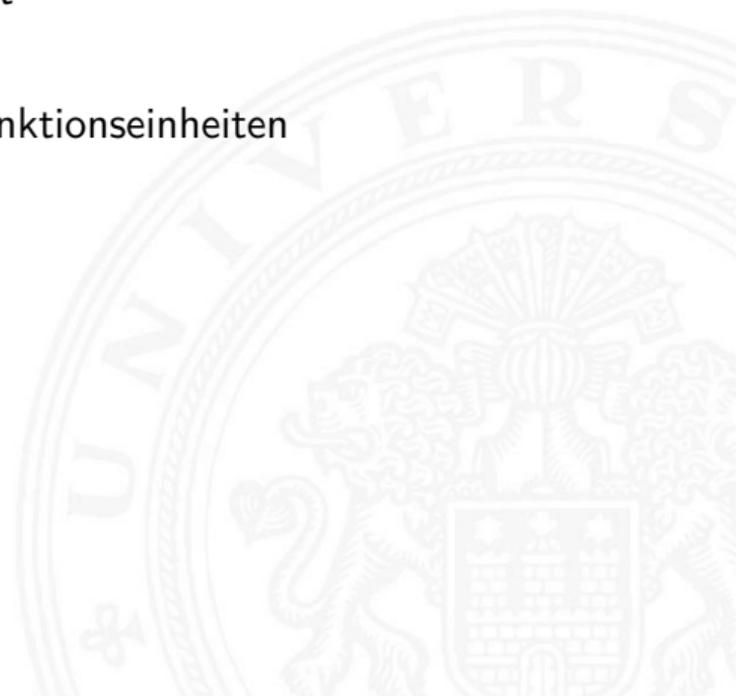




# Instruction Execute

## „Befehl ausführen“ Phase im Befehlszyklus

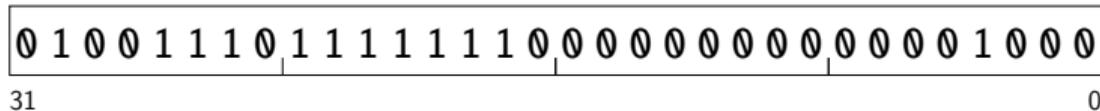
- ▷ Befehl steht im Befehlsregister IR
  - ▷ Decoder hat Opcode und Operanden entschlüsselt
  - ▷ Steuersignale liegen an Funktionseinheiten
  - 1. Ausführung des Befehls durch Aktivierung der Funktionseinheiten
  - 2. ggf. Programmzähler setzen/inkrementieren
- 
- ▶ Details abhängig von der Art des Befehls
  - ▶ Ausführungszeit            –"–
  - ▶ Realisierung
    - ▶ fest verdrahtete Hardware
    - ▶ mikroprogrammiert



Befehlsklassen	Beispiele
▶ arithmetische Operationen	add, sub, inc, dec, mult, div
logische Operationen	and, or, xor
schiebe Operationen	shl, sra, srl, ror
▶ Vergleichsoperationen	cmpeq, cmpgt, cmplt
▶ Datentransfers	load, store, I/O
▶ Programm-Kontrollfluss	jump, jmq, branch, call, return
▶ Maschinensteuerung	trap, halt, (interrupt)
⇒ Befehlssätze und Computerarchitekturen (Details später)	
CISC – Complex Instruction Set Computer	
RISC – Reduced Instruction Set Computer	

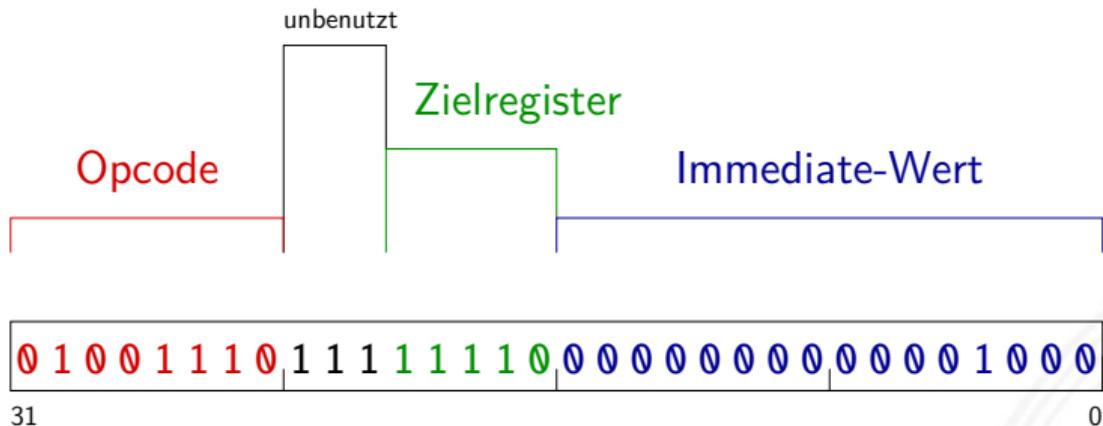


- ▷ Befehlsregister IR enthält den aktuellen Befehl
- ▷ z.B. einen 32-bit Wert



Wie soll die Hardware diesen Wert interpretieren?

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



- ▶ 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 Praktikum RSB (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





- ▶ 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

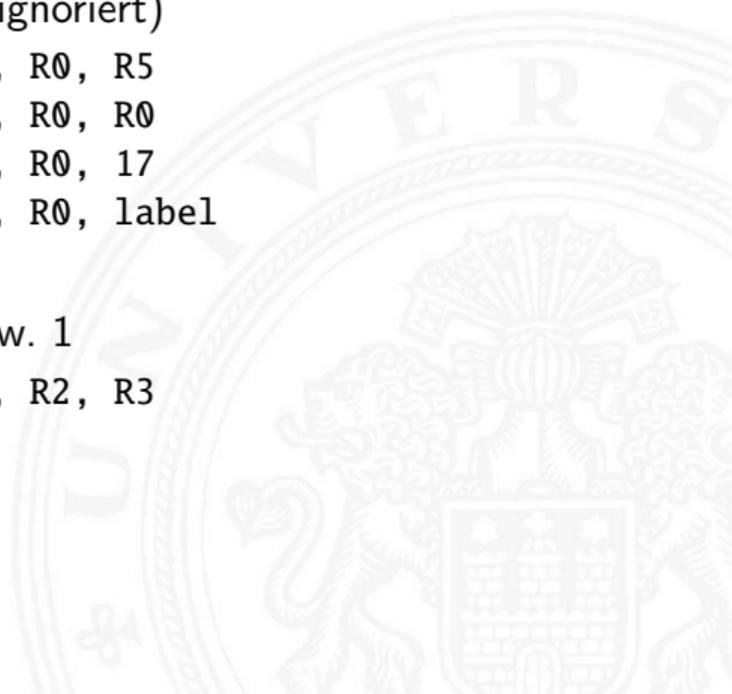


- ▶ 32 Register, R0 ... R31, jeweils 32-bit
- ▶ R1 bis R31 sind Universalregister
- ▶ R0 ist konstant Null (Schreiboperationen werden ignoriert)

- ▶ R0 Tricks
  - R5 = -R5           sub  R5, R0, R5
  - R4 = 0             add  R4, R0, R0
  - R3 = 17            addi R3, R0, 17
  - if (R2 != 0)       bne  R2, R0, label

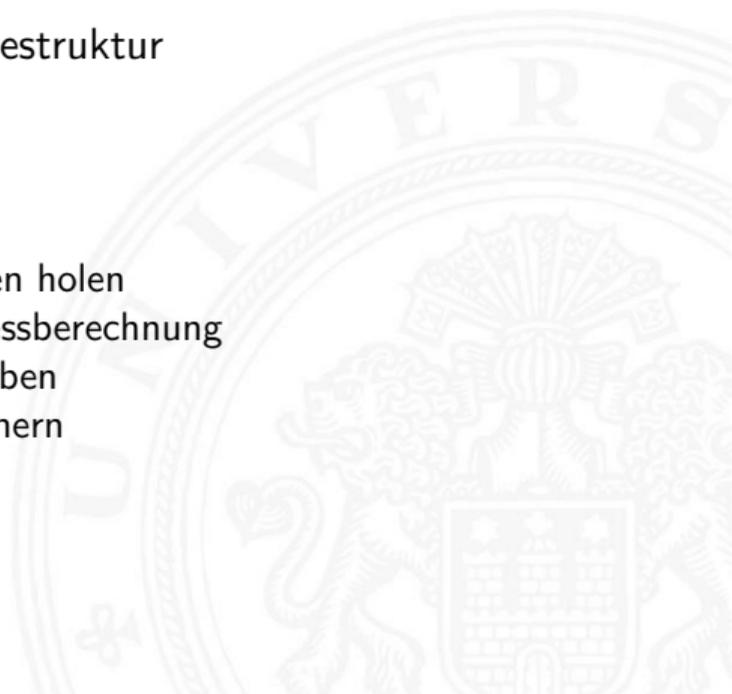
- ▶ keine separaten Statusflags
- ▶ Vergleichsoperationen setzen Zielregister auf 0 bzw. 1

- R1 = (R2 < R3)     slt  R1, R2, R3



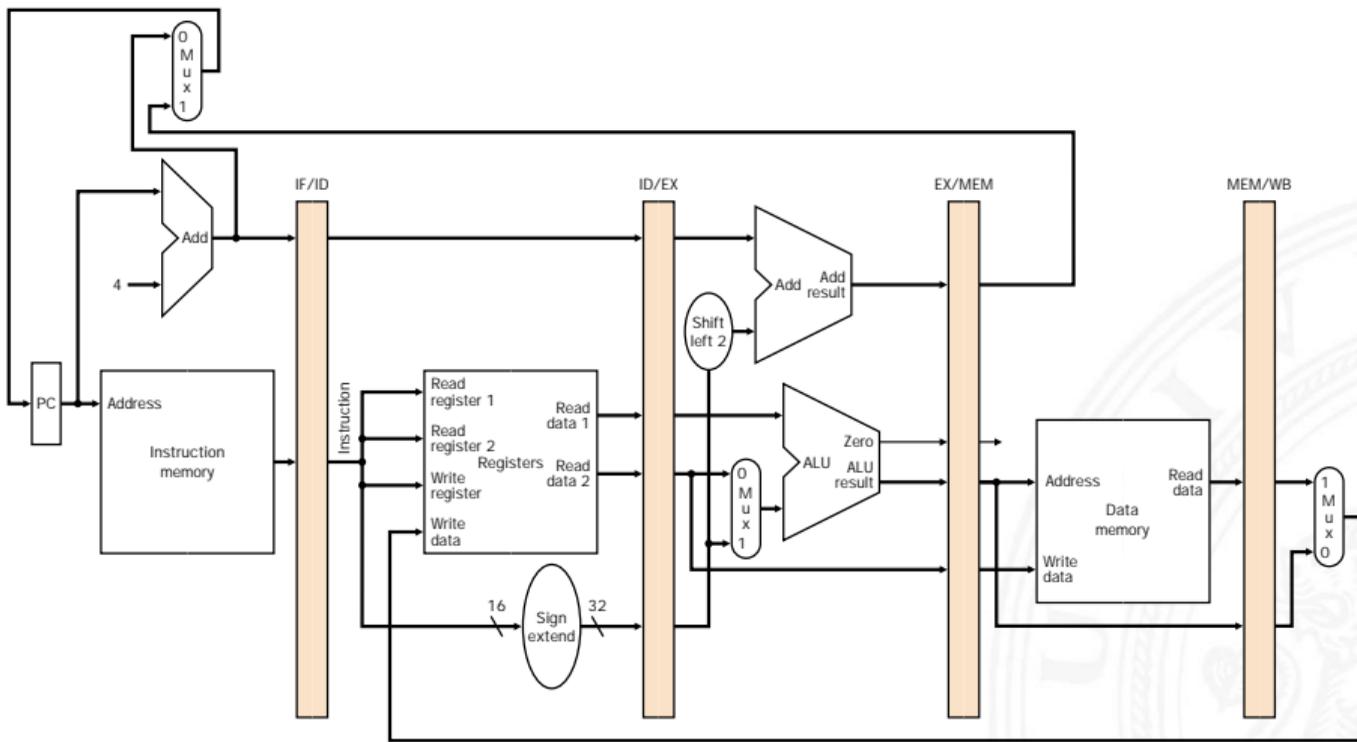


- ▶ Übersicht und Details: [PH22, PH20]  
David A. Patterson, John L. Hennessy: *Computer Organization and Design*  
– *The Hardware/Software Interface*
- ▶ dort auch hervorragende Erläuterung der Hardwarestruktur
- ▶ klassische fünf-stufige Befehlspipeline
  - ▶ Instruction-Fetch      Befehl holen
  - ▶ Decode                    Decodieren und Operanden holen
  - ▶ Execute                  ALU-Operation oder Adressberechnung
  - ▶ Memory                  Speicher lesen oder schreiben
  - ▶ Write-Back                Resultat in Register speichern





# MIPS: Hardwarestruktur



PC  
I-Cache

Register  
(R0...R31)

ALUs

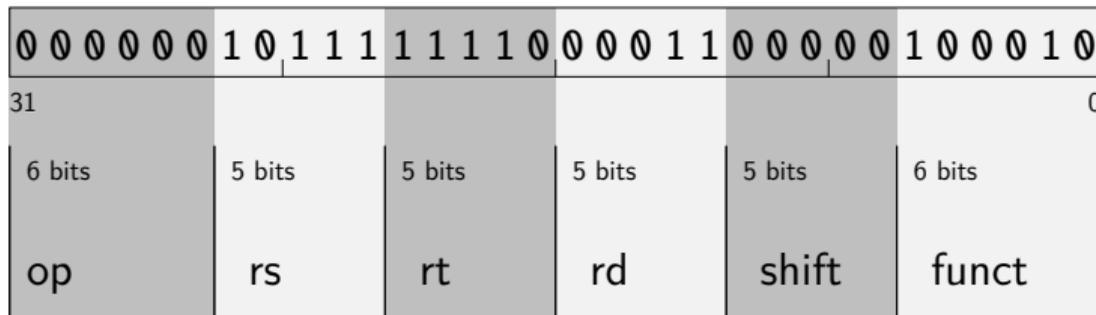
Speicher  
D-Cache

[PH22]



# MIPS: Befehlsformate

## Befehl im R-Format

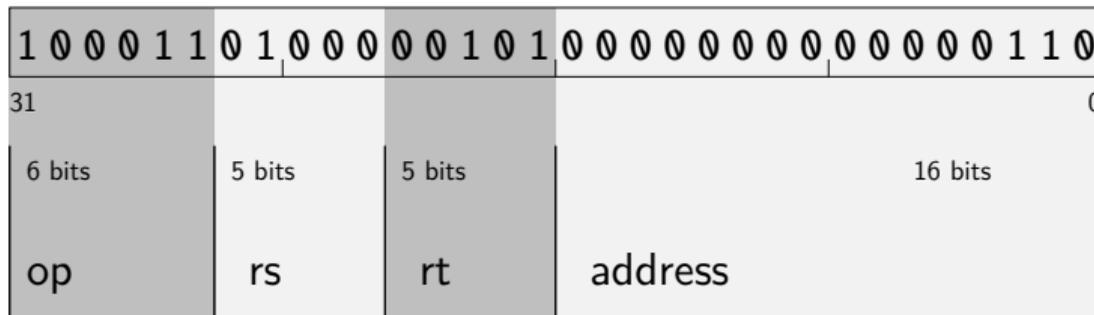


- ▶ op: Opcode      Typ des Befehls      0 = „alu-op“
  - rs: source register 1      erster Operand      23 = „r23“
  - rt: source register 2      zweiter Operand      30 = „r30“
  - rd: destination register      Zielregister      3 = „r3“
  - shift: shift amount      (optionales Shiften)      0 = „0“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



# MIPS: Befehlsformate

## Befehl im I-Format

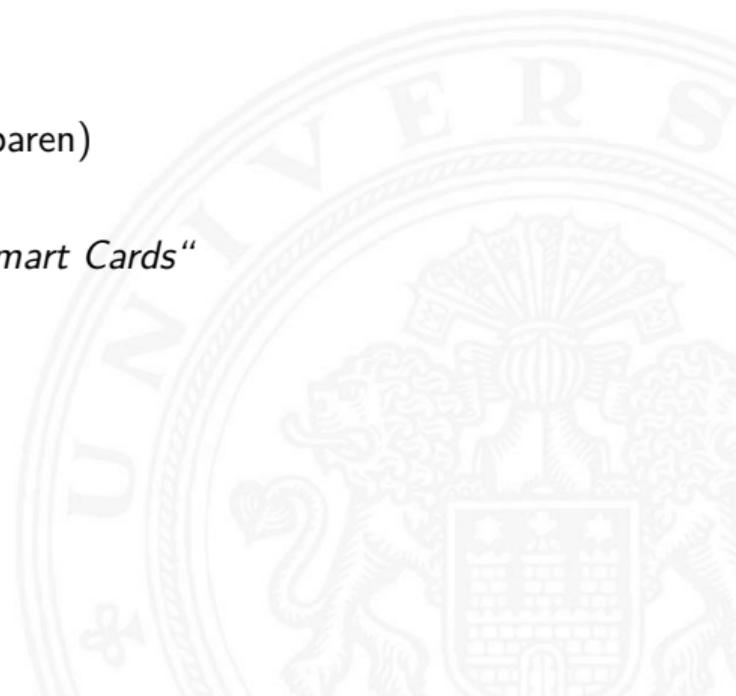


- ▶ op: Opcode                      Typ des Befehls    35 = „lw“
- rs: base register                Basisadresse        8 = „r8“
- rt: destination register        Zielregister         5 = „r5“
- addr: address offset            Offset                6 = „6“

⇒ r5 = MEM[r8+6]                      lw r5, 6(r8)



- ▶ 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
  - ▶ Mikrocontroller für eingebettete Systeme, z.B. „Smart Cards“
  - ▶ siehe [en.wikipedia.org/wiki/M.CORE](http://en.wikipedia.org/wiki/M.CORE)



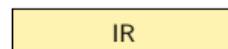
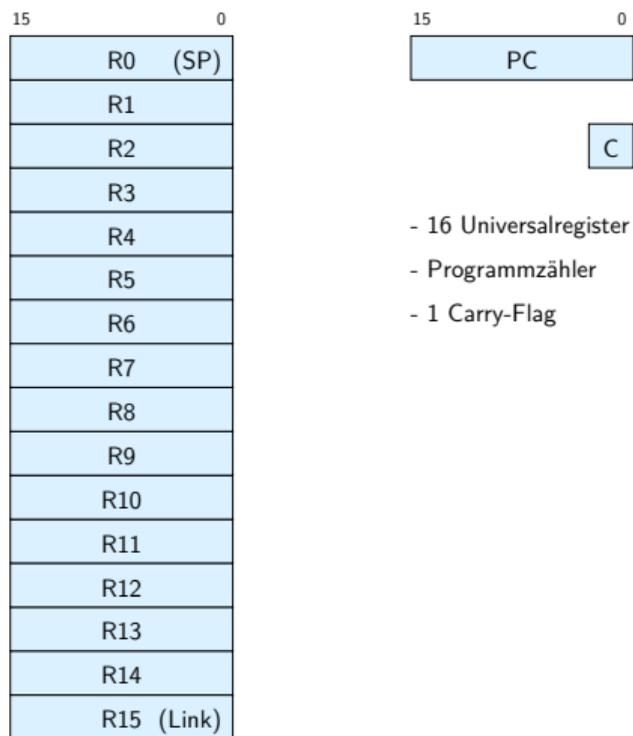


- ▶ ähnlich M·CORE
- ▶ gleiches Registermodell, aber nur 16-bit Wortbreite
  - ▶ Program Counter PC
  - ▶ 16 Universalregister R0...R15
  - ▶ Statusregister C („carry flag“)
- ▶ Subset der Befehle, einfachere Codierung
- ▶ vollständiger Hardwareaufbau in Hades verfügbar
  - ▶ [HenHA] Hades Demo: [60-dcore/t3/chapter](#)  
oder Simulator mit Assembler aus den Praktikumsunterlagen
  - ▶ 64-042: Rechnerstrukturen und Betriebssysteme (t3asm)

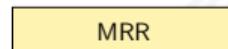
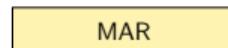




# D-CORE: Registermodell



- Befehlsregister



- Bus-Interface

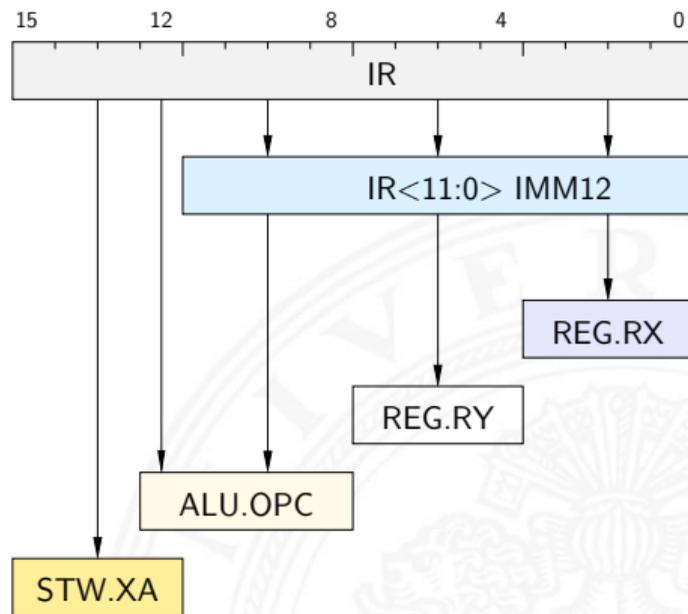
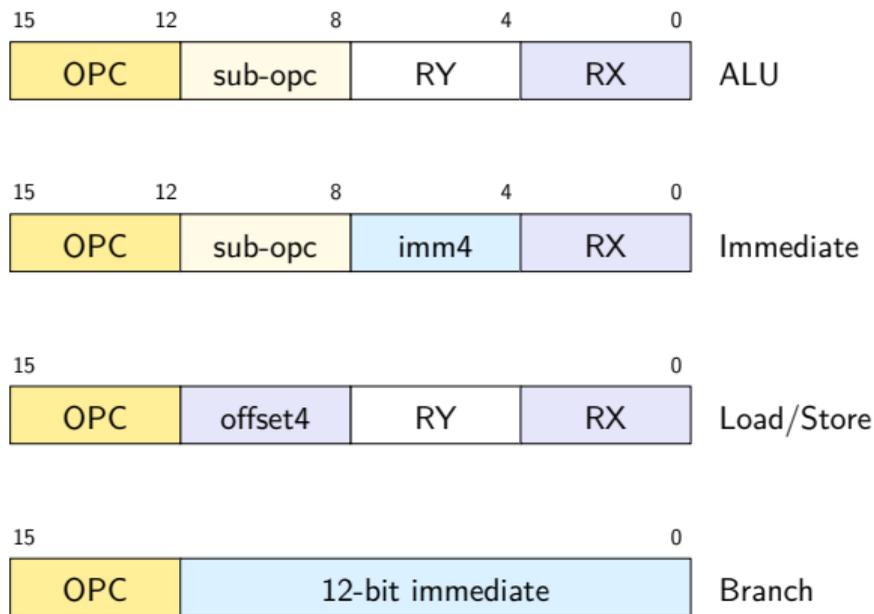
► sichtbar für Programmierer: R0...R15, PC und C



Befehl	Funktion
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 und 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



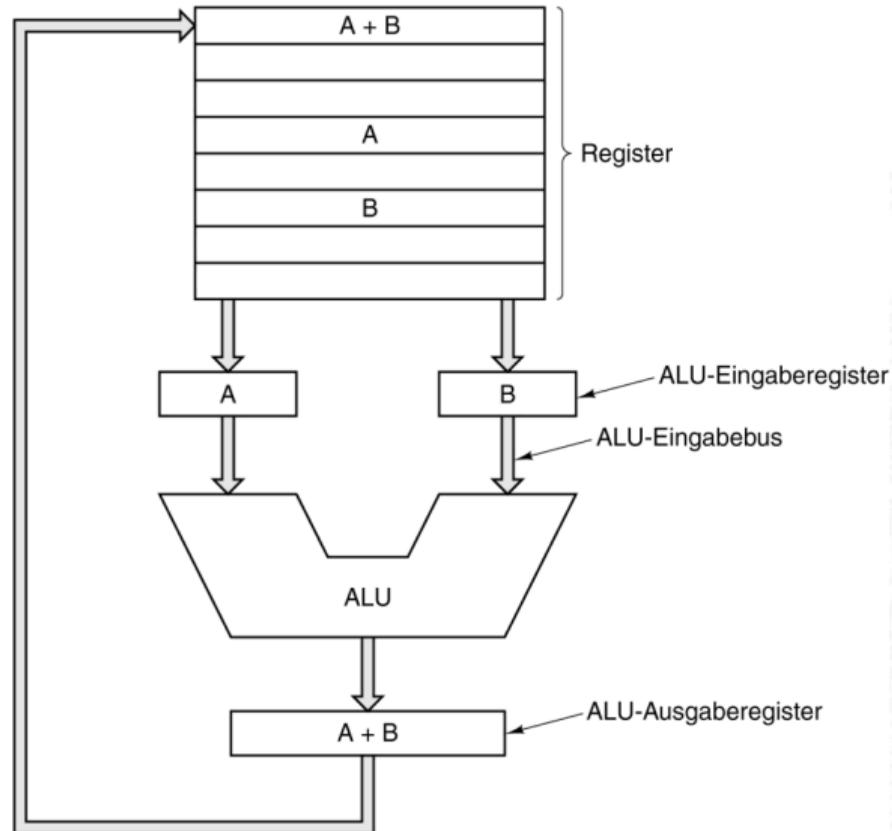
- ▶ Woher kommen die Operanden / Daten für die Befehle?
    - ▶ Hauptspeicher, Universalregister, Spezialregister
  - ▶ Wie viele Operanden pro Befehl?
    - ▶ 0- / 1- / 2- / 3-Adress Maschinen
  - ▶ Wie werden die Operanden adressiert?
    - ▶ immediate / direkt / indirekt / indiziert / autoinkrement / usw.
- ⇒ wichtige Unterscheidungsmerkmale für Rechnerarchitekturen
- ▶ Zugriff auf Hauptspeicher:  $\approx 100 \times$  langsamer als Registerzugriff
    - ▶ möglichst Register statt Hauptspeicher verwenden!
    - ▶ „load/store“-Architekturen



- ▷ Rechner soll „rechnen“ können
- ▷ typische arithmetische Operation nutzt 3 Variablen  
Resultat, zwei Operanden:  $X = Y + Z$   
add r2, r4, r5 Inhalt von r4 und r5 addieren, Resultat in r2 speichern
- ▶ woher kommen die Operanden?
- ▶ wo soll das Resultat hin?
  - ▶ Speicher
  - ▶ Register
- ▶ entsprechende Klassifikation der Architektur



- ▶ 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?

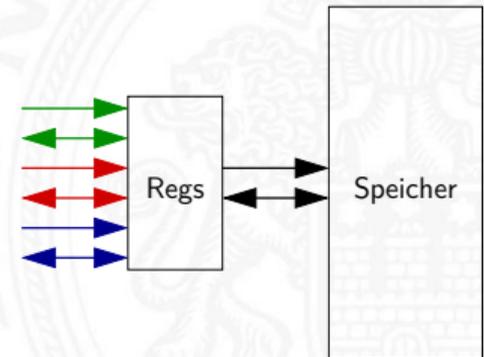
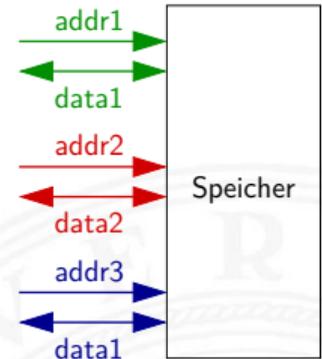
- ▶ typische Architektur
  - ▶ von-Neumann Prinzip: alle Daten im Hauptspeicher
  - ▶ 3-Adress Befehle: zwei Operanden, ein Resultat

⇒ „Multiport-Speicher“ mit drei Ports ?

- ▶ sehr aufwändig, 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





- 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: Resultat und ein Operand
  - ▶ 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, C, T
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
```

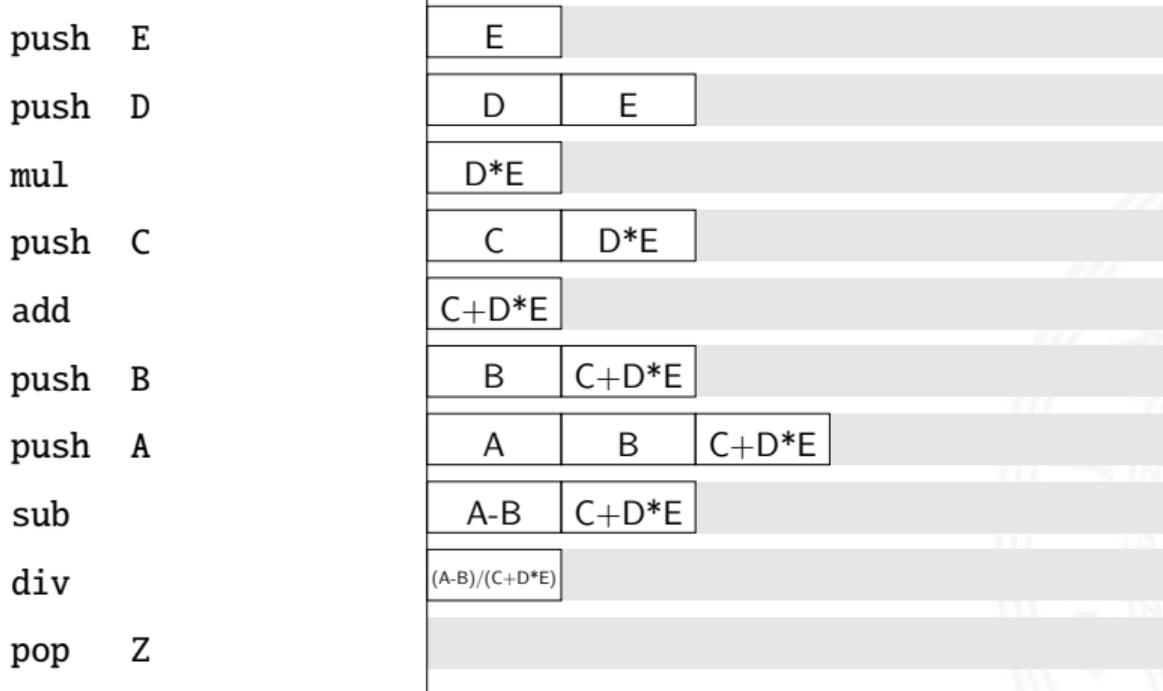
## 0-Adress Maschine

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

# Beispiel: Stack-Maschine / 0-Adress Maschine

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

0-Adress Maschine



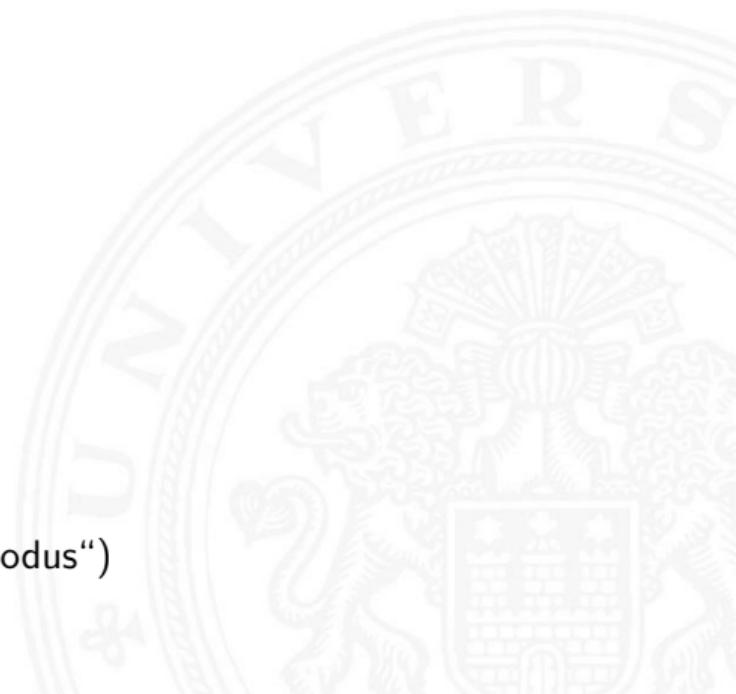


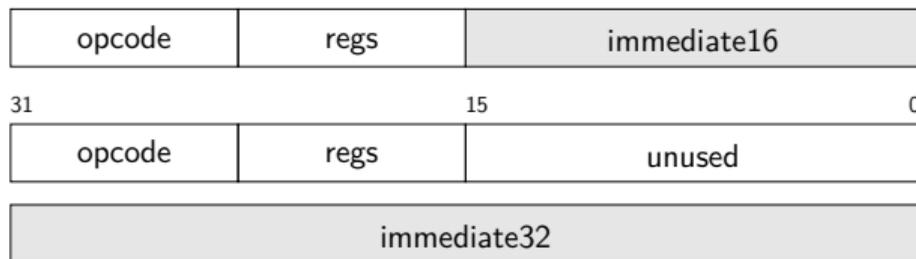
- ▶ „immediate“
  - ▶ Operand steht direkt im Befehl
  - ▶ kein zusätzlicher Speicherzugriff
  - ▶ aber Länge des Operanden beschränkt
- ▶ „direkt“
  - ▶ Adresse des Operanden steht im Befehl
  - ▶ keine zusätzliche Adressberechnung
  - ▶ ein zusätzlicher Speicherzugriff
  - ▶ Adressbereich beschränkt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





- ▶ „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“)

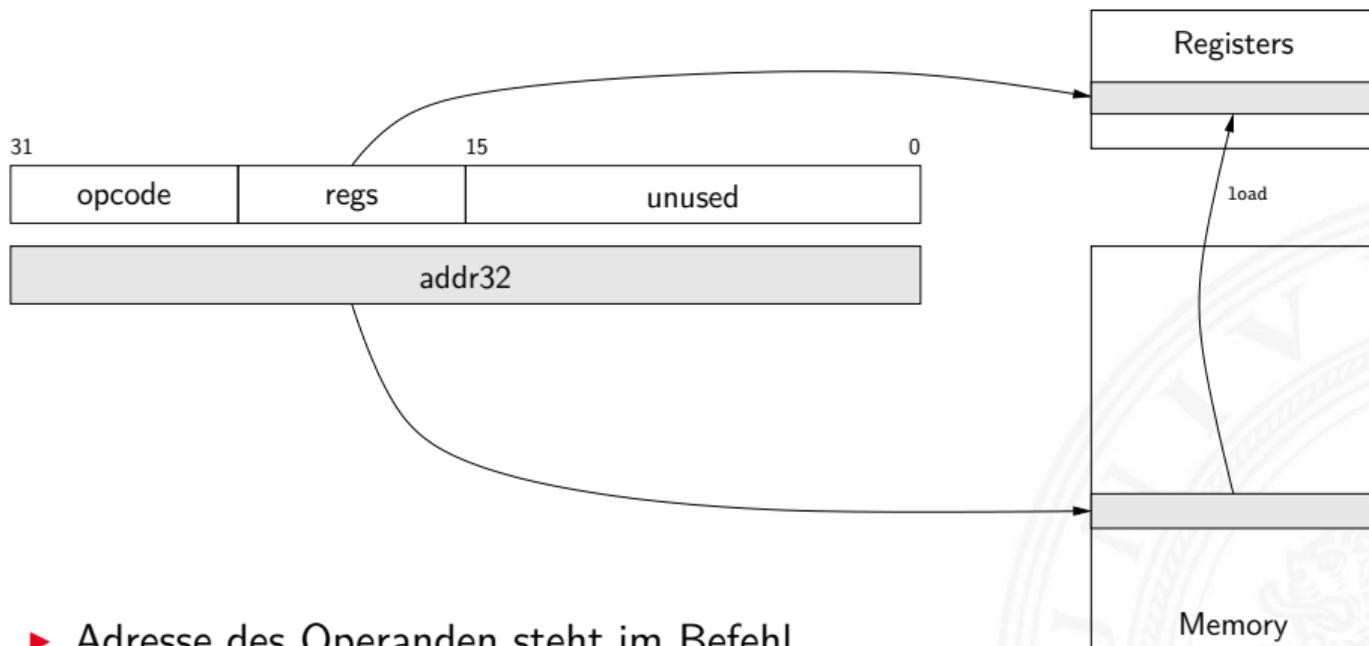




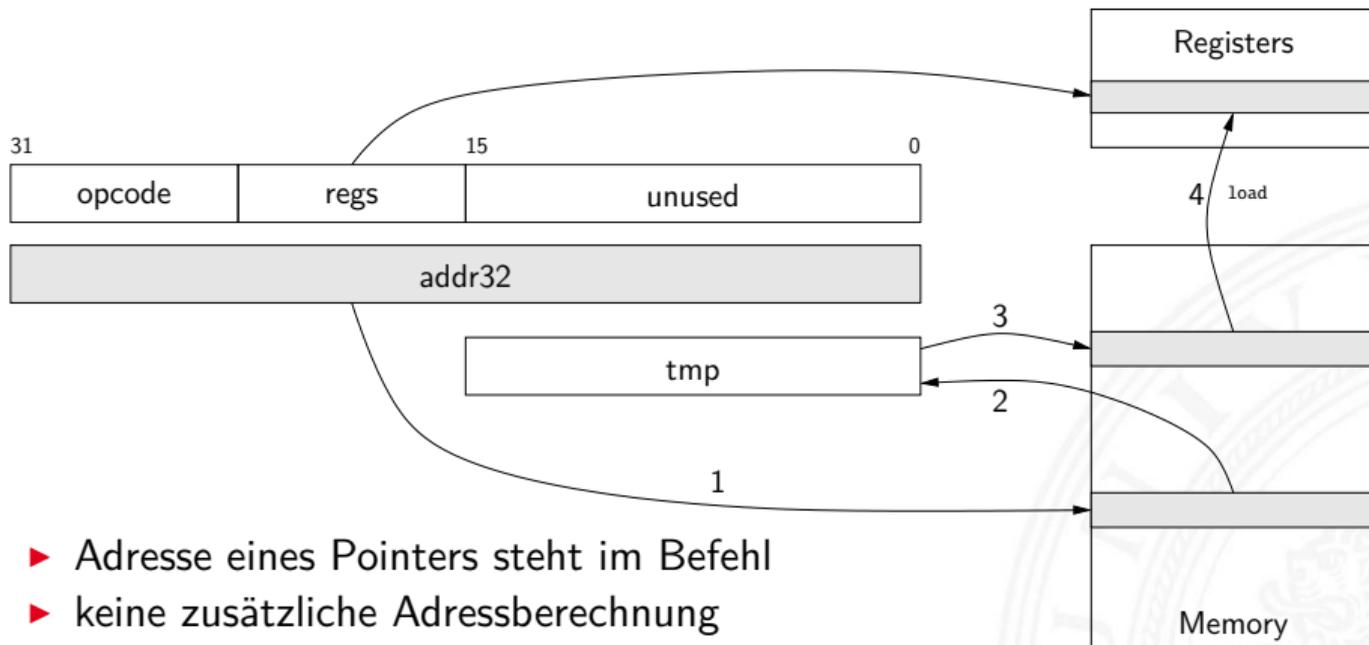
1-Wort Befehl

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)

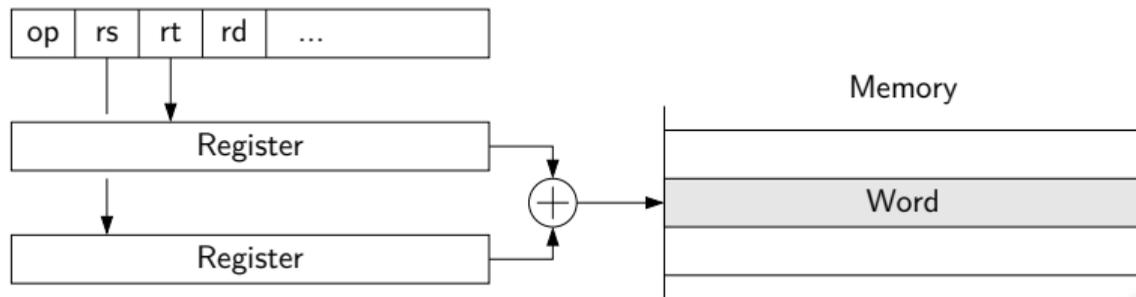


- ▶ Adresse des Operanden steht im Befehl
- ▶ keine zusätzliche Adressberechnung
- ▶ ein zusätzlicher Speicherzugriff: z.B.  $R3 = \text{MEM}[\text{addr32}]$
- ▶ Adressbereich beschränkt oder 2-Wort Befehl (wie Immediate)

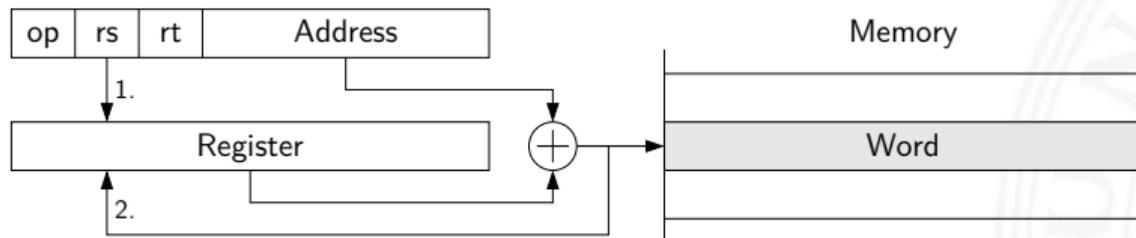


- ▶ Adresse eines Pointers steht im Befehl
- ▶ keine zusätzliche Adressberechnung
- ▶ zwei zusätzliche Speicherzugriffe:  
z.B.  $tmp = MEM[addr32]$   $R3 = MEM[tmp]$
- ▶ typische CISC-Adressierungsart, viele Taktzyklen
- ▶ kommt bei RISC-Rechnern nicht vor

## Indexaddressing



## Updateaddressing



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



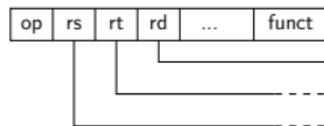
# Beispiel: MIPS Adressierungsarten

## 1. Immediate addressing



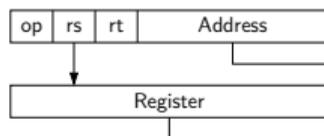
immediate

## 2. Register addressing

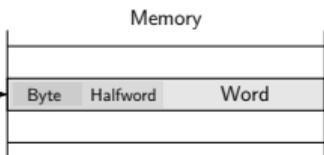


register

## 3. Base addressing

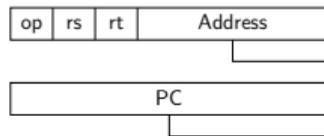


+

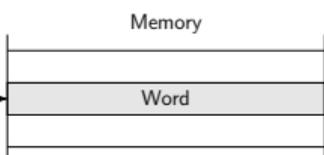


index + offset

## 4. PC-relative addressing

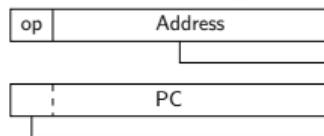


+

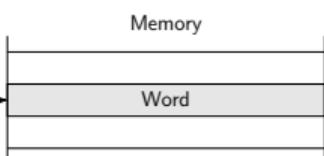


PC + offset

## 5. Pseudodirect addressing



&



PC<sub>(31..28)</sub> & address





welche Adressierungsarten / -Varianten sind üblich?

- ▶ 0-Adress (Stack-) Maschine      Java virtuelle Maschine
  - ▶ 1-Adress (Akkumulator) Maschine      8-bit Mikrocontroller  
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)





- ▶ übliche Bezeichnung für die Intel-Prozessorfamilie
- ▶ von 8086, 80286, 80386, 80486, Pentium ... Pentium 4, 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

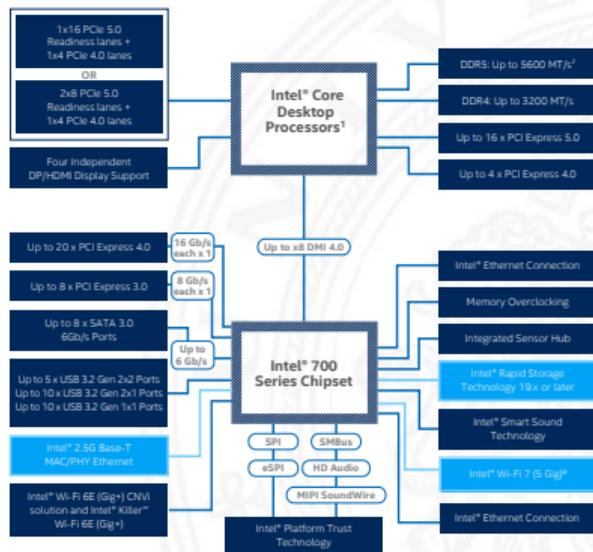
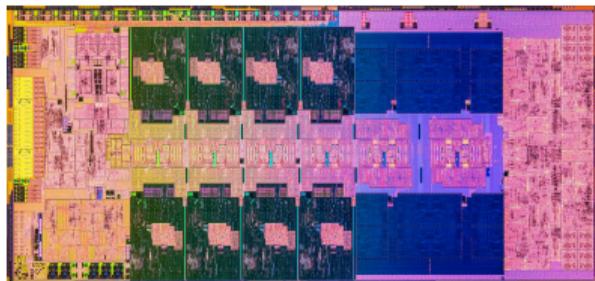
Hinweis: niemand erwartet, dass Sie sich alle Details merken

Chip	Datum	MHz	Transistoren	Speicher	Anmerkungen
4004	4/1971	0,108	2 300	640 B	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 4	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	Speichercontroller, Taktanpassung
...					GPU, I/O-Contr., Spannungsregelung ...
...	11/2021				Befehlssatz: AVX ... Performance- / Efficiency-Core ...

# Beispiel: Core i9-14900K Prozessor

Performance Cores	8 (× 2 Hyperthreading)
Taktfrequenz	3,2 GHz (max. 6,0 GHz)
L1 Cache	8 × 32 KiB I + 48KiB D
L2 Cache	8 × 2,0 MiB (I+D)
Efficiency Cores	16
Taktfrequenz	2,4 GHz (max. 4,4 GHz)
L1 Cache	16 × 32 KiB I + 64KiB D
L2 Cache	4 × 2 MiB (I+D)
L3 Cache	36 MiB (I+D)
Memory Controller	2 × 44,8 GiB/s
DMI Durchsatz	16 GT/s ↔ Chipsatz
Prozess	7 nm
Größe	257 mm <sup>2</sup>
Leistungsaufnahme	125 W (< 253 W)

Quellen: [ark.intel.com](http://ark.intel.com), [www.intel.de](http://www.intel.de)  
[en.wikichip.org](http://en.wikichip.org)

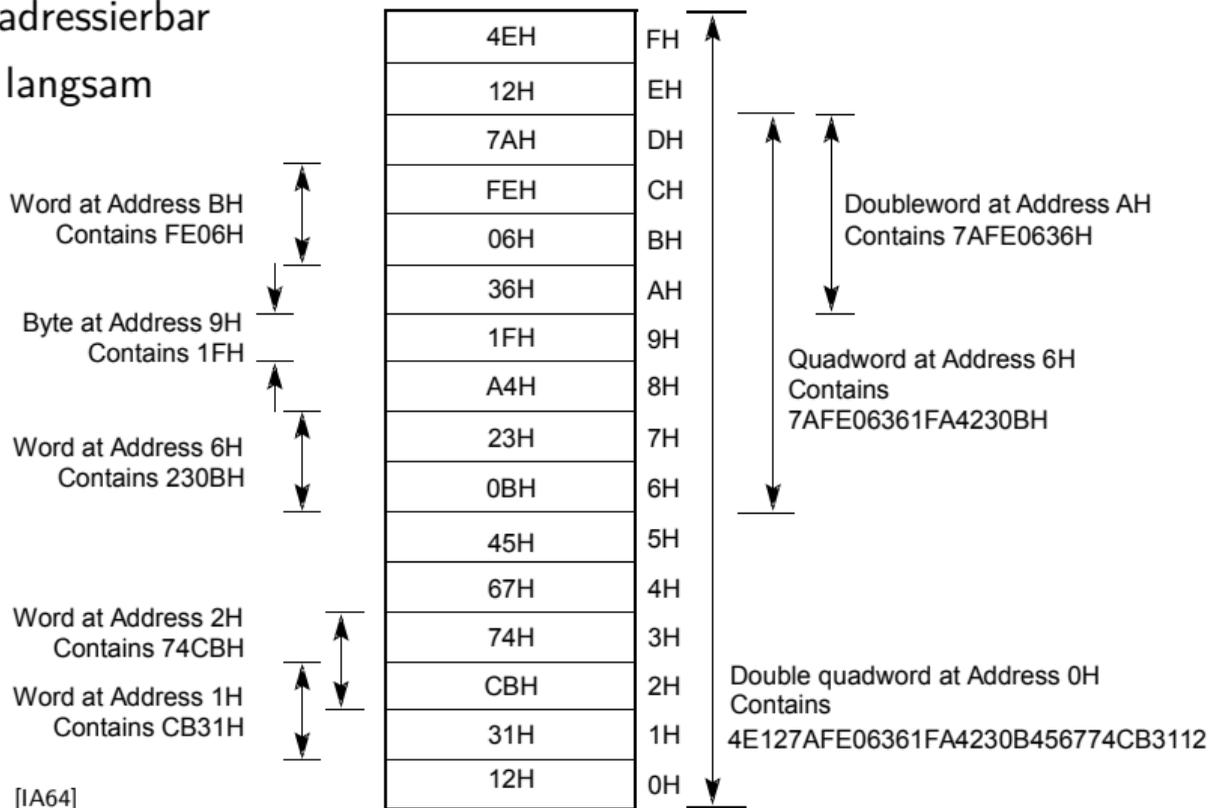




# x86: Speichermodell (cont.)

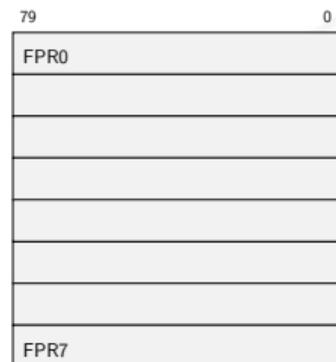
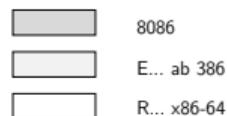
- ▶ Speicher voll byte-adressierbar
- ▶ misaligned Zugriffe langsam

## ▶ Beispiel



# x86: Register

63	31	15	0	
RAX	EAX	AX	AH AL	accumulator
RBX	EBX	BX	BH BL	base addr
RCX	ECX	CX	CH CL	count: String, Loop
RDX	EDX	DX	DH DL	data, multiply/divide
RSI	ESI	SI		index, string src
RDI	EDI	DI		index, string dst
RSP	ESP	SP		stackptr
RBP	EBP	BP		base of stack segment
		CS		code segment
		SS		stack segment
		DS		data segment
		ES		extra data segment
		FS		
		GS		
RIP	EIP	IP		PC
	EFLAGS			status



**64-bit Mode**      **32-bit Mode**      **16-bit Mode**

R8 ... R15	...D	...W	...L
------------	------	------	------



# 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

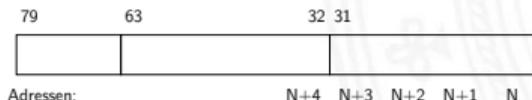
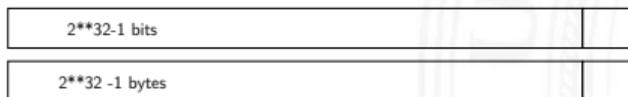
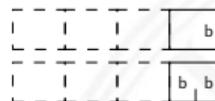
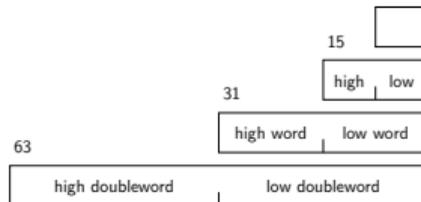
(16 bit segment + 32 bit offset)

bit field

bit string

byte string

float / double / extended



## Funktionalität

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	(packed/unpacked BCD) <code>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

- ▶ 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 aufwändige 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 16/32/64-bit Adresse

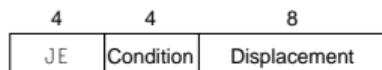
operand size        Umschaltung 16/32/64-bit Operanden

repeat                Stringoperationen: für alle Elemente

lock                  Speicherschutz bei Multiprozessorsystemen

# x86 Befehlscodierung: Beispiele

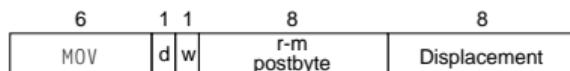
a. JE EIP + displacement



b. CALL



c. MOV EBX, [EDI + 45]



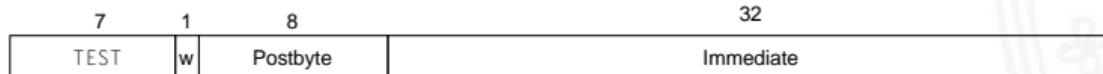
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



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

[PH22]

# x86 Befehlskodierung: Beispiele (cont.)

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 codes (flags) with EDX & 42
MOVSL	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

[PH22]

# 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        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 E8FCFFFFFF    call _IO_putc
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 main(...)

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_puts	
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	



## Kriterien für einen *guten* Befehlssatz

- ▶ vollständig: alle notwendigen Instruktionen verfügbar
- ▶ orthogonal: keine zwei Instruktionen leisten das Gleiche
- ▶ symmetrisch: z.B. Addition  $\Leftrightarrow$  Subtraktion
- ▶ adäquat: technischer Aufwand entsprechend zum Nutzen
- ▶ effizient: kurze Ausführungszeiten





Statistiken zeigen: Dominanz der einfachen Instruktionen

► x86-Prozessor

	Anweisung	Ausführungshäufigkeit %
1.	load	22 %
2.	conditional branch	20 %
3.	compare	16 %
4.	store	12 %
5.	add	8 %
6.	and	6 %
7.	sub	5 %
8.	move reg-reg	4 %
9.	call	1 %
10.	return	1 %
	Total	96 %

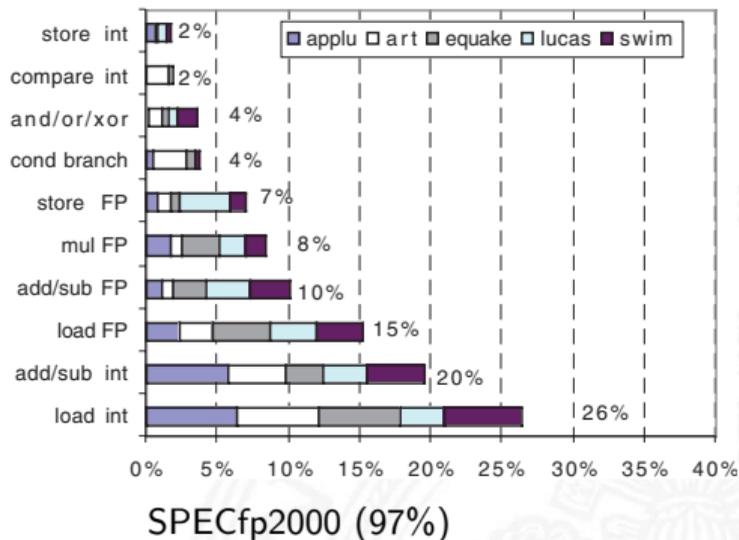
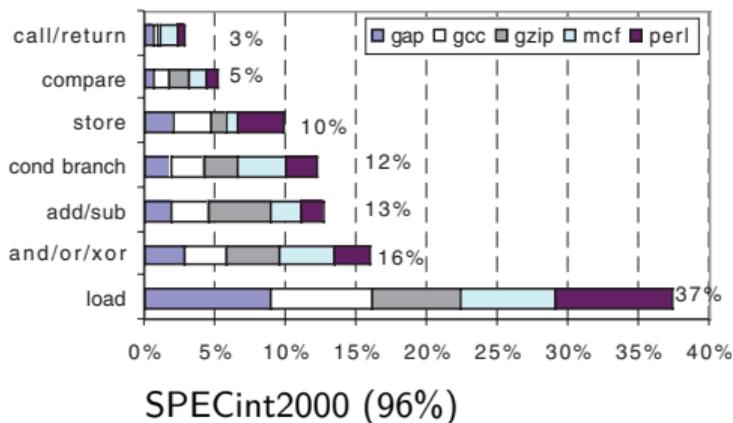
## ► MIPS-Prozessor

Instruction	compress	eqntott	espresso	gcc (cc1)	li	Int. average
load	20.8%	18.5%	21.9%	24.9%	23.3%	22%
store	13.8%	3.2%	8.3%	16.6%	18.7%	12%
add	10.3%	8.8%	8.15%	7.6%	6.1%	8%
sub	7.0%	10.6%	3.5%	2.9%	3.6%	5%
mul				0.1%		0%
div						0%
compare	8.2%	27.7%	15.3%	13.5%	7.7%	16%
mov reg-reg	7.9%	0.6%	5.0%	4.2%	7.8%	4%
load imm	0.5%	0.2%	0.6%	0.4%		0%
cond. branch	15.5%	28.6%	18.9%	17.4%	15.4%	20%
uncond. branch	1.2%	0.2%	0.9%	2.2%	2.2%	1%
call	0.5%	0.4%	0.7%	1.5%	3.2%	1%
return, jmp indirect	0.5%	0.4%	0.7%	1.5%	3.2%	1%
shift	3.8%		2.5%	1.7%		1%
and	8.4%	1.0%	8.7%	4.5%	8.4%	6%
or	0.6%		2.7%	0.4%	0.4%	1%
other (xor, not, . . .)	0.9%		2.2%	0.1%		1%
load FP						0%
store FP						0%
add FP						0%
sub FP						0%
mul FP						0%
div FP						0%
compare FP						0%
mov reg-reg FP						0%
other (abs, sqrt, . . .)						0%

[HP17]

**Figure D.15** 80x86 instruction mix for five SPECint92 programs.

# Bewertung der ISA (cont.)



- ▶ über 80 % der Berechnungen eines typischen Programms verwenden nur ca. 20 % der Instruktionen einer CPU
  - ▶ sehr einfache Instruktionen werden am häufigsten gebraucht: load, store, add ...
- ⇒ Motivation für RISC



Rechnerarchitekturen mit irregulärem, komplexem Befehlssatz und (unterschiedlich) langer Ausführungszeit

- ▶ aus der Zeit der ersten Großrechner, 60er Jahre
- ▶ Programmierung auf Assemblerebene
- ▶ Komplexität durch sehr viele (mächtige) Befehle umgehen

typische Merkmale

- ▶ Instruktionssätze mit mehreren hundert Befehlen ( $> 300$ )
- ▶ unterschiedlich lange Instruktionsformate: 1...n-Wort Befehle
  - ▶ komplexe Befehlskodierung
  - ▶ mehrere Schreib- und Lesezugriffe pro Befehl
- ▶ viele verschiedene Datentypen



- ▶ sehr viele Adressierungsarten, -Kombinationen
  - ▶ fast alle Befehle können auf Speicher zugreifen
  - ▶ Mischung von Register- und Speicheroperanden
  - ▶ komplexe Adressberechnung
- ▶ Unterprogrammaufrufe: über Stack
  - ▶ Übergabe von Argumenten
  - ▶ Speichern des Programmzählers
  - ▶ explizite „Push“ und „Pop“ Anweisungen
- ▶ Zustandscodes („*Flags*“)
  - ▶ implizit gesetzt durch arithmetische und logische Anweisungen





## Vor- / Nachteile

- + nah an der Programmiersprache, einfacher Assembler
- + kompakter Code: weniger Befehle holen, kleiner I-Cache
- Befehlssatz vom Compiler schwer auszunutzen
- Ausführungszeit abhängig von: Befehl, Adressmodi ...
- Instruktion holen schwierig, da variables Instruktionsformat
- Speicherhierarchie schwer handhabbar: Adressmodi
- Pipelining schwierig

## Beispiele

- ▶ Intel x86 / IA-64, Motorola 68 000, DEC Vax





- ▶ ein Befehl kann nicht in einem Takt abgearbeitet werden
- ⇒ Unterteilung in Mikroinstruktionen ( $\varnothing$  5...7)

- ▶ Ablaufsteuerung durch endlichen Automaten
- ▶ meist als ROM (RAM) implementiert, das *Mikroprogramm*worte beinhaltet

## 1. horizontale Mikroprogrammierung

- ▶ langes Mikroprogrammwort (ROM-Zeile)
- ▶ steuert direkt alle Operationen
- ▶ Spalten entsprechen: Kontrollleitungen und Folgeadressen

▶ horizontale Mikroprog.



## 2. vertikale Mikroprogrammierung

▶ vertikale Mikroprog.

- ▶ kurze Mikroprogrammworter
- ▶ Spalten enthalten Mikrooperationscode
- ▶ mehrstufige Decodierung für Kontrollleitungen

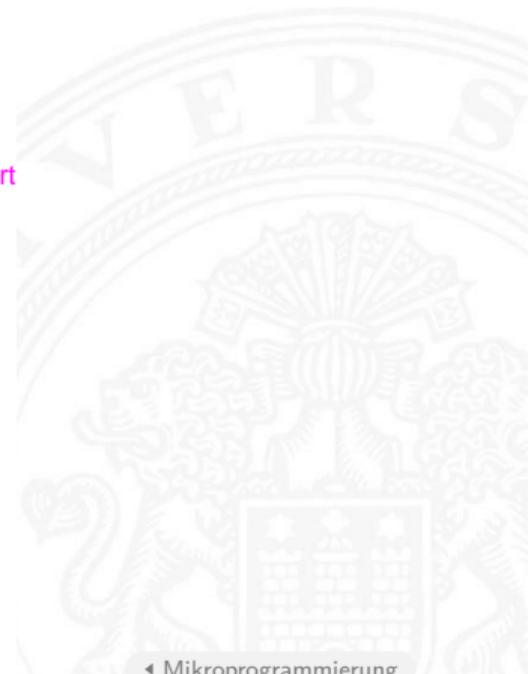
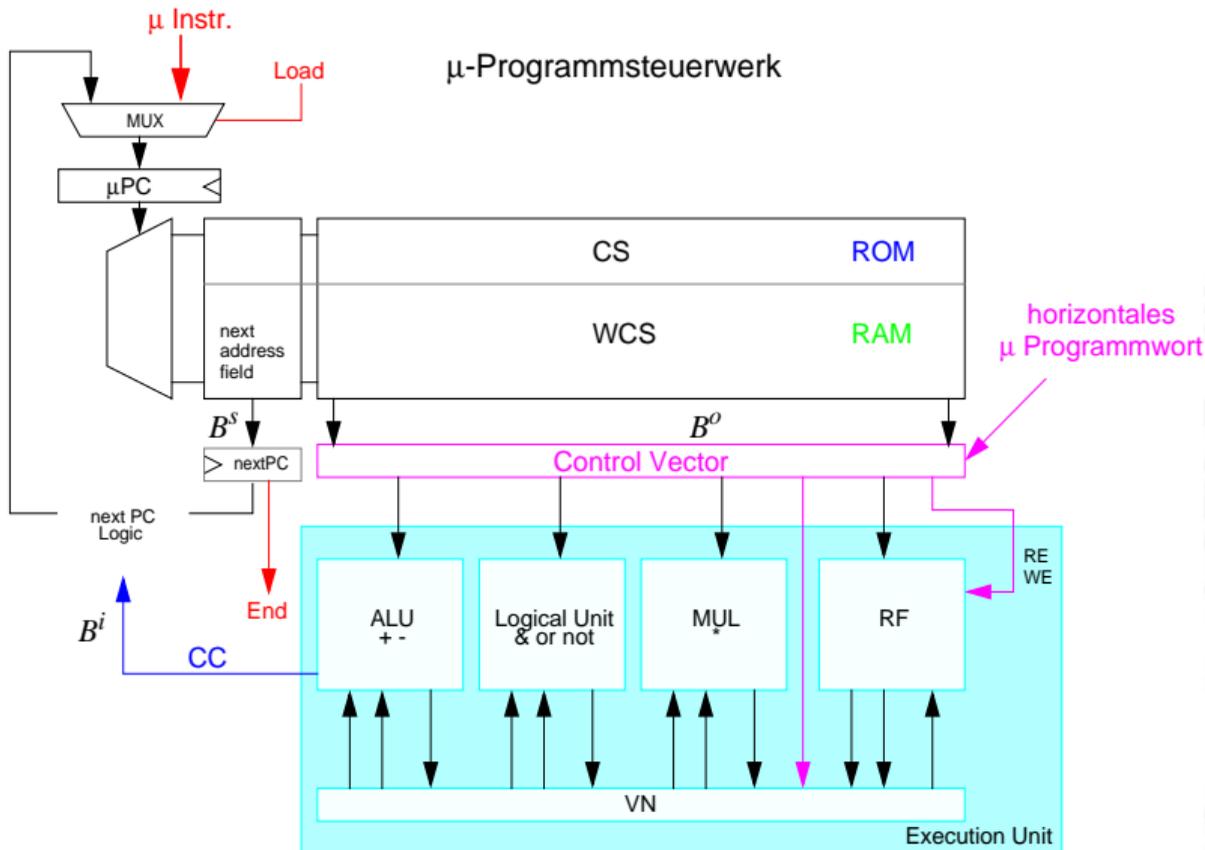
+ CISC-Befehlssatz mit wenigen Mikrobefehlen realisieren

+  $\mu$ -Programm im RAM: Mikrobefehlssatz austauschbar

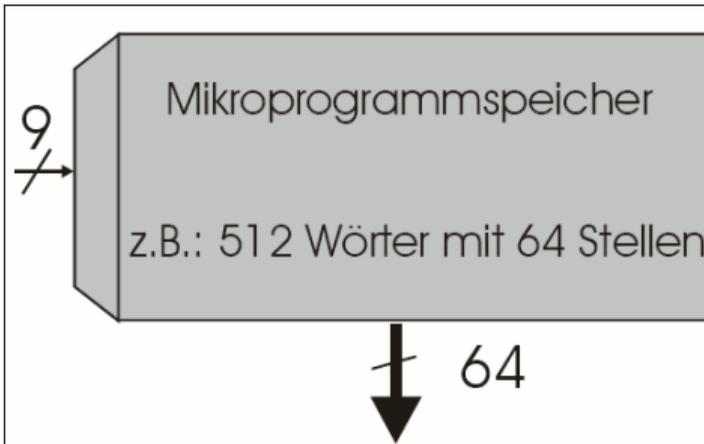
– (mehrstufige) ROM/RAM Zugriffe: zeitaufwändig

⇒ wird inzwischen nur noch benutzt, um CISC Befehle in RISC-artige Sequenzen umzusetzen (x86)

# horizontale Mikroprogrammierung

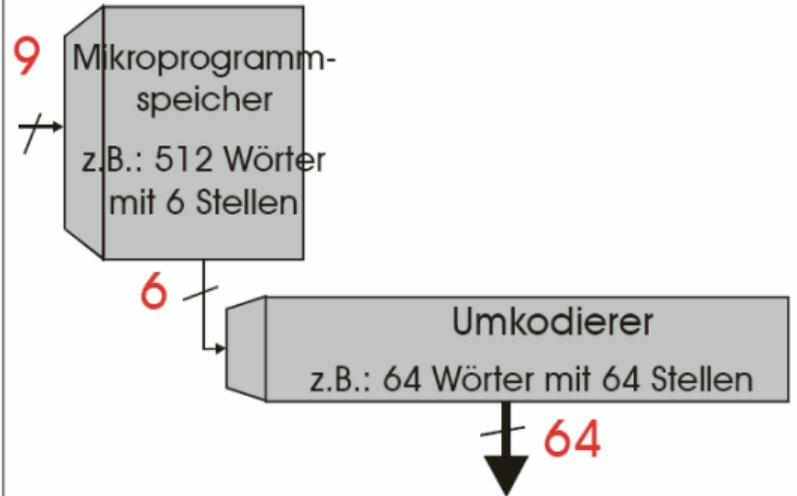


# vertikale Mikroprogrammierung



Annahme: DIFF = 50 unterschiedliche  
Binärbelegungen

$$\lceil \log_2 \text{DIFF} \rceil = 6$$





oft auch: „**Regular Instruction Set Computer**“

- ▶ seit den 80er Jahren: „RISC-Boom“
  - ▶ internes Projekt bei IBM
  - ▶ von Hennessy (Stanford) und Patterson (Berkeley) publiziert
- ▶ Grundidee: Komplexitätsreduktion in der CPU
- ▶ optimierende Compiler statt Assemblerprogrammierung
  - ⇒ statt Mikroprogramm in CISC: mehrere, einfache Assemblerbefehle
  - ⇒ CISC Komplexität der Befehle wird durch Compiler abgelöst

Beispiele

- ▶ IBM 801, MIPS, SPARC, DEC Alpha, ARM

typische Merkmale

- ▶ reduzierte Anzahl einfacher Instruktionen (z.B. 128)
  - ▶ benötigen in der Regel mehr Anweisungen für eine Aufgabe
  - ▶ werden aber mit kleiner, sehr schneller Hardware ausgeführt

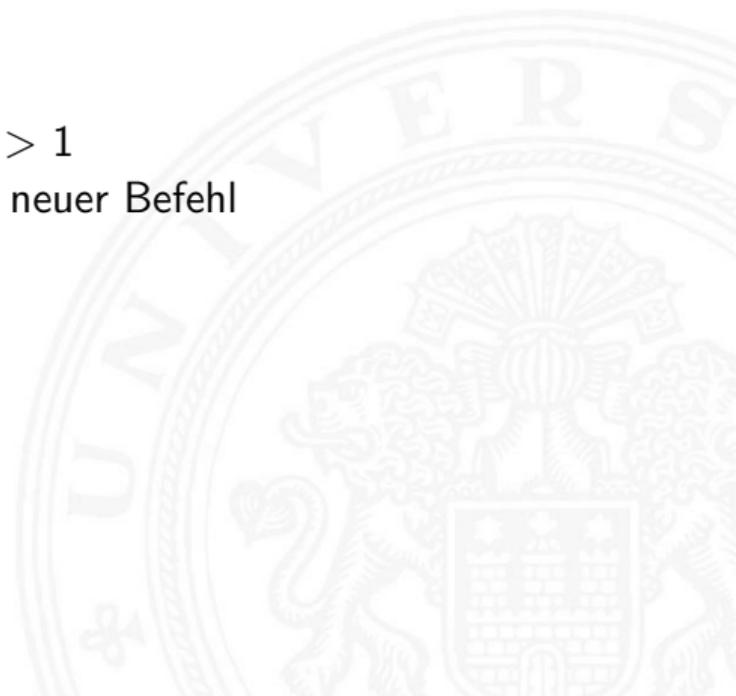


- ▶ reguläre Struktur, z.B. 32-bit Wortbreite, 32-bit Befehle
- ▶ nur ein-Wort Befehle
- ▶ alle Befehle in gleicher Zeit ausführbar  $\Rightarrow$  Pipeline-Verarbeitung
- ▶ Speicherzugriff *nur* durch „Load“ und „Store“ Anweisungen
  - ▶ alle anderen Operationen arbeiten auf Registern
  - ▶ keine Speicheroperanden
- ▶ Register-orientierter Befehlssatz
  - ▶ viele universelle Register, keine Spezialregister ( $\geq 32$ )
  - ▶ oft mehrere (logische) *Registersätze*: Zuordnung zu Unterprogrammen, Tasks etc.
- ▶ Unterprogrammaufrufe: über Register
  - ▶ Register für Argumente, „Return“-Adressen, Zwischenergebnisse
- ▶ keine Zustandscodes („*Flags*“)
  - ▶ spezielle Testanweisungen
  - ▶ speichern Resultat direkt im Register



## Vor- / Nachteile

- + fest-verdrahtete Logik, kein Mikroprogramm
- + einfache Instruktionen, wenige Adressierungsarten
- + Pipelining gut möglich
- + Cycles per Instruction = 1, bei Superskalarität  $> 1$   
in Verbindung mit Pipelining: je Takt (mind.) ein neuer Befehl
- längerer Maschinencode
- viele Register notwendig
- ▶ optimierende Compiler nötig / möglich
- ▶ High-performance Speicherhierarchie notwendig





## ursprüngliche Debatte

- ▶ streng geteilte Lager
- ▶ pro CISC: einfach für den Compiler; weniger Code Bytes
- ▶ pro RISC: 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 wichtig!
  - ▶ mit genügend Hardware wird alles schnell ausgeführt
- ▶ eingebettete Prozessoren: eindeutige RISC-Orientierung
  - + kleiner, billiger, weniger Leistungsverbrauch



- ▶ Restriktionen durch Hardware abgeschwächt
- ▶ Code-Kompatibilität leichter zu erfüllen
  - ▶ Emulation in Firm- und Hardware
- ▶ Intel bewegt sich weg von IA-32
  - ▶ erlaubt nicht genug Parallelität

hat IA-64 eingeführt („Intel Architecture 64-bit“)

- ⇒ neuer Befehlssatz mit expliziter Parallelität (EPIC)
- ⇒ 64-bit Wortgrößen (überwinden Adressraumlimits)
- ⇒ benötigt hoch entwickelte Compiler





- [BO15] R.E. Bryant, D.R. O'Hallaron:  
*Computer systems – A programmers perspective.*  
3rd global ed., Pearson Education Ltd., 2015. ISBN 978-1-292-10176-7  
csapp.cs.cmu.edu
- [TA14] A.S. Tanenbaum, T. Austin:  
*Rechnerarchitektur – Von der digitalen Logik zum Parallelrechner.*  
6. Auflage, Pearson Deutschland GmbH, 2014. ISBN 978-3-8689-4238-5
- [PH22] D.A. Patterson, J.L. Hennessy: *Rechnerorganisation und Rechnerentwurf – Die Hardware/Software-Schnittstelle – MIPS Edition.*  
6. Auflage, De Gruyter Oldenbourg, 2022. ISBN 978-3-11-075598-5
- [PH20] D.A. Patterson, J.L. Hennessy: *Computer Organization and Design – The Hardware Software Interface – RISC-V Edition.*  
2nd edition, Morgan Kaufmann Publishers Inc., 2020. ISBN 978-0-12-820331-6



- [HP17] J.L. Hennessy, D.A. Patterson:  
*Computer architecture – A quantitative approach.*  
6th edition, Morgan Kaufmann Publishers Inc., 2017. ISBN 978-0-12-811905-1
- [Fur00] S. Furber: *ARM System-on-Chip Architecture.*  
2nd edition, Pearson Education Limited, 2000. ISBN 978-0-201-67519-1
- [HenHA] N. Hendrich: *HADES — HAMBURG DEsign System.*  
Universität Hamburg, FB Informatik, Lehrmaterial.  
[tams.informatik.uni-hamburg.de/applets/hades/webdemos](http://tams.informatik.uni-hamburg.de/applets/hades/webdemos)
- [IA64] *Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 1: Basic Architecture.*  
Intel Corp.; Santa Clara, CA.  
[software.intel.com/en-us/articles/intel-sdm](http://software.intel.com/en-us/articles/intel-sdm)