64-040 Modul InfB-RS: Rechnerstrukturen

https://tams.informatik.uni-hamburg.de/lectures/2015ws/vorlesung/rs

– Kapitel 12 –

Norman Hendrich

Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik

Technische Aspekte Multimodaler Systeme

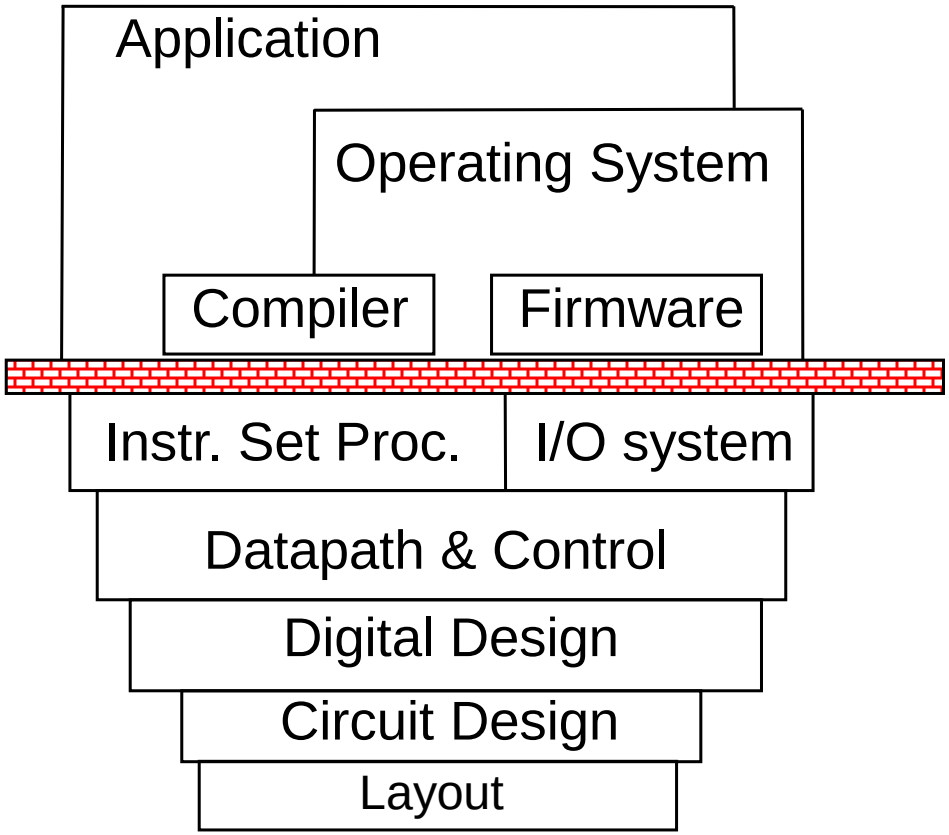
Wintersemester 2015/2016

Kapitel 12

Rechnerarchitektur: Instruction Set Architecture

Softwareentwicklung

Rechnerstrukturen



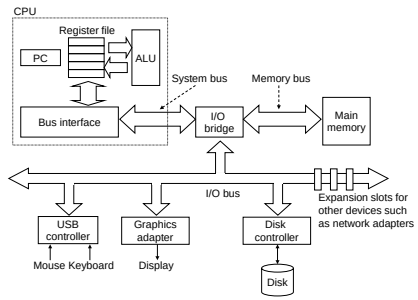
Instruction Set Architecture

[PH14]

Befehlssatzarchitektur – ISA

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

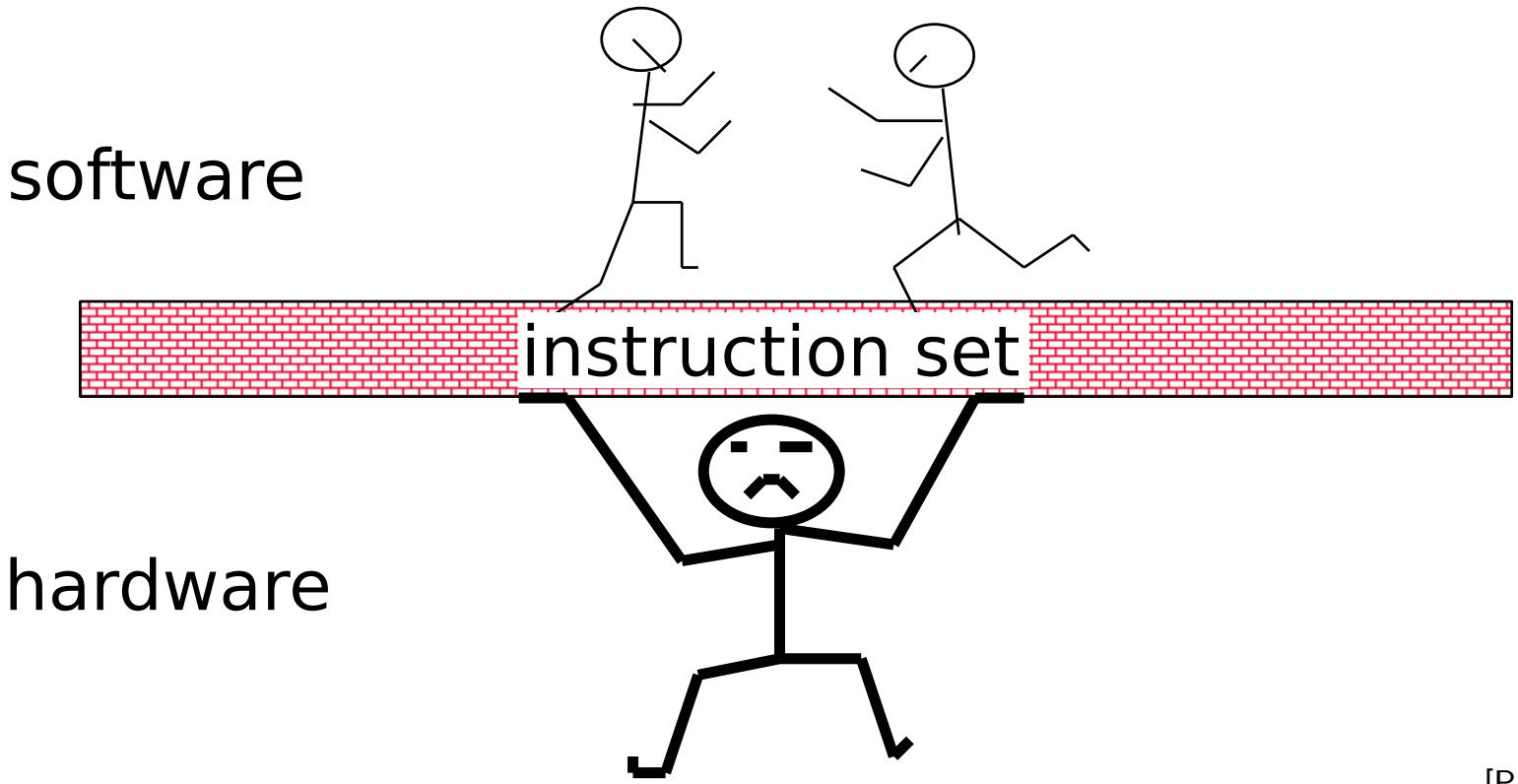
⇒ alle für den Programmierer sichtbaren Attribute eines Rechners



- 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

Befehlssatzarchitektur – ISA (cont.)

- Befehlssatz: die zentrale Schnittstelle



[PH14]

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

Beispiele für charakteristische ISA

in dieser Vorlesung bzw. im Praktikum angesprochen

- MIPS — klassischer 32-bit RISC
- D·CORE — „Demo Rechner“, 16-bit
- x86 — CISC, Verwendung in PCs

- Assemblerprogrammierung, Kontrollstrukturen und Datenstrukturen werden am Beispiel der x86-Architektur vorgestellt

- viele weitere Architekturen (z.B. Mikrocontroller) werden aus Zeitgründen nicht weiter behandelt

Artenvielfalt der Architekturen

									
Prozessor	1 μC	1 μC	1 ASIC	1 μP, ASIP	DSPs	1 μP, 3 DSP	1 μP, DSP	$\approx$100 μC, μP, DSP	1 μP, 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	—	RS232	diverse	GSM	MIDI	V.90	CAN, ...	I^2C, ...
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, ...
- sehr unterschiedliche Anforderungen:
  Echtzeit, Sicherheit, Zuverlässigkeit, Leistungsaufnahme, Abwärme, Temperaturbereich, Störstrahlung, Größe, Kosten etc.

Was ist Rechnerarchitektur?

Definitionen

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

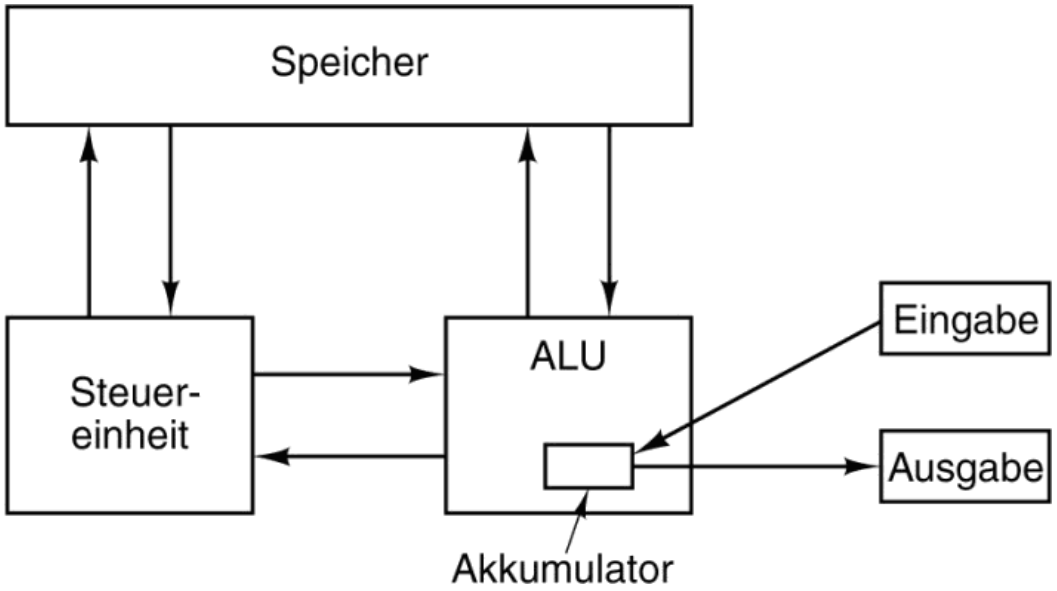
2. *The study of computer architecture is the study of the organization and interconnection of components of computer systems. Computer architects construct computers from basic building blocks such as memories, arithmetic units and buses.*

Was ist Rechnerarchitektur? (cont.)

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

By this we mean that a memory performs the storage function, an adder does addition, and an input/output interface passes data from a processor to the outside world, regardless of the nature of the computer in which they are embedded. The major differences between computers lie in the way of the modules are connected together, and the way the computer system is controlled by the programs. In short, computer architecture is the discipline devoted to the design of highly specific and individual computers from a collection of common building blocks. [Stone]

Wiederholung: von-Neumann Rechner



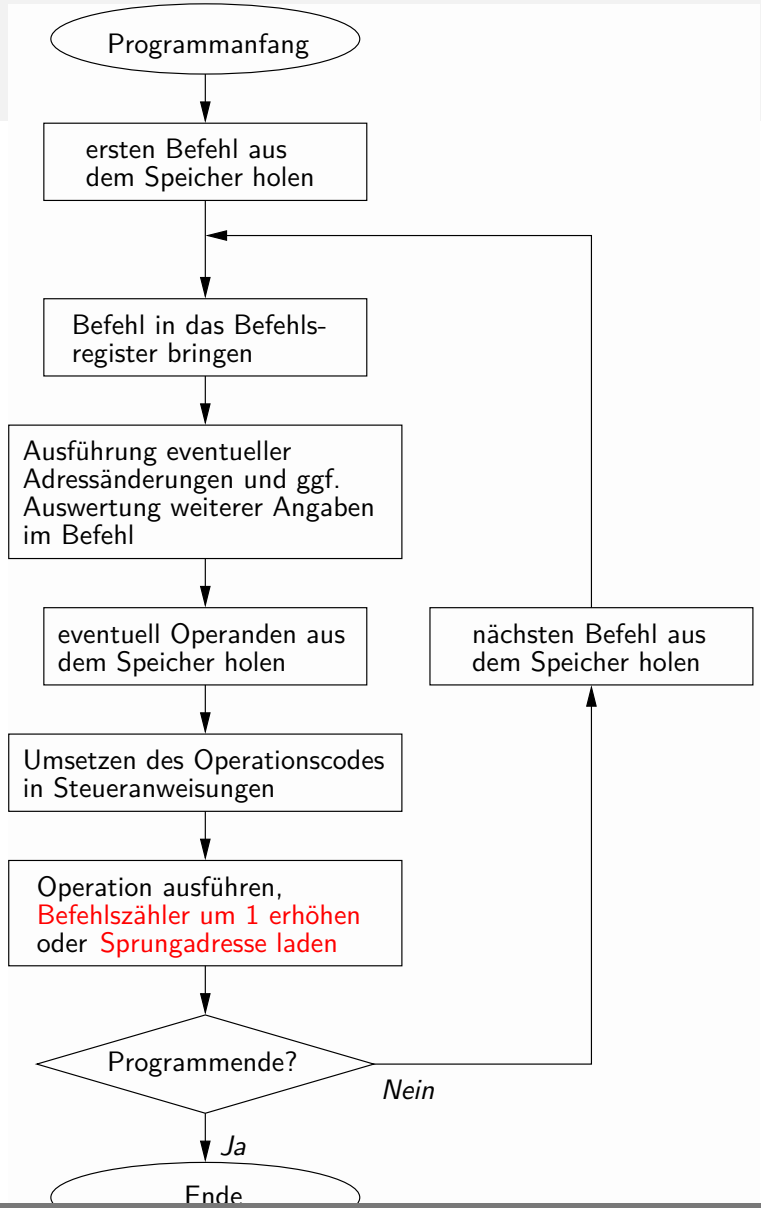
[TA14]

Fünf zentrale Komponenten:

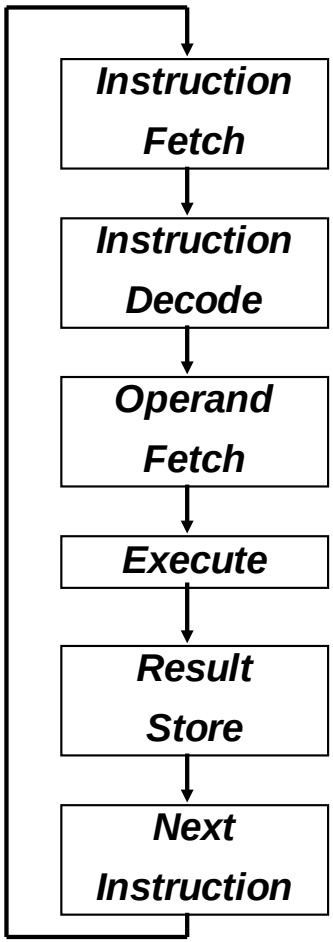
- Prozessor mit **Steuerwerk** und **Rechenwerk** (ALU, Register)
- **Speicher**, gemeinsam genutzt für Programme und Daten
- **Eingabe**- und **Ausgabewerke**
- verbunden durch Bussystem

von-Neumann Konzept

- Programm als Sequenz elementarer Anweisungen (Befehle)
- als Bitvektoren im Speicher codiert
- Interpretation (Operanden, Befehle und Adressen) ergibt sich aus dem Kontext (der Adresse)
- zeitsequenzielle Ausführung der Instruktionen



Befehlszyklus



Befehl aus Programmspeicher holen

auszuführende Aktionen und Länge der Instruktion bestimmen, ggf. Worte nachladen

Operanden ermitteln und laden

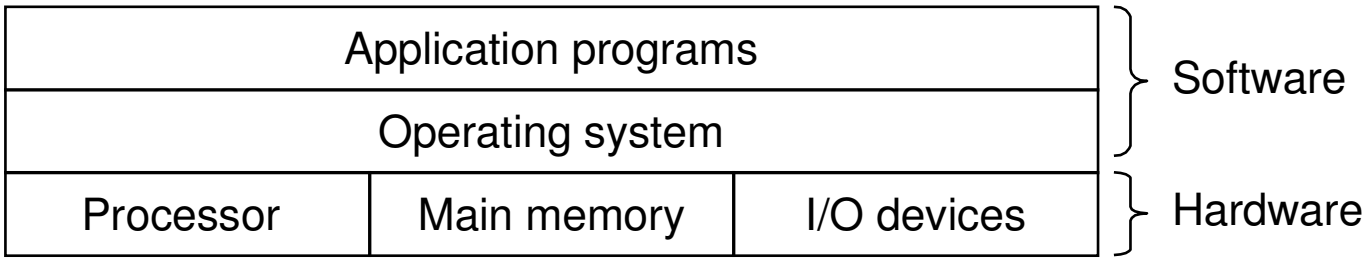
Ergebnis der Operation berechnen bzw. Status ermitteln

Ergebnisse für später abspeichern

Folgeoperation ermitteln

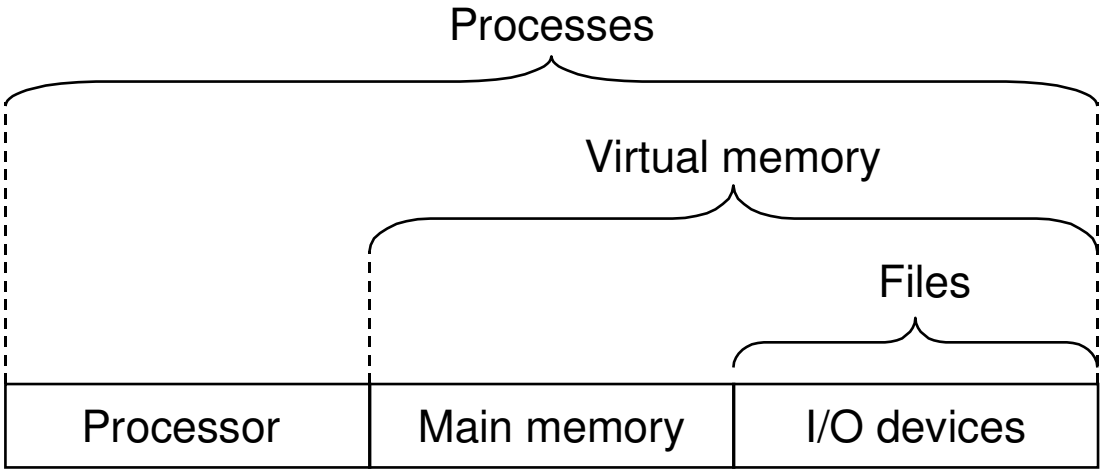
Beschreibungsebenen

- Schichten-Ansicht: Software – Hardware



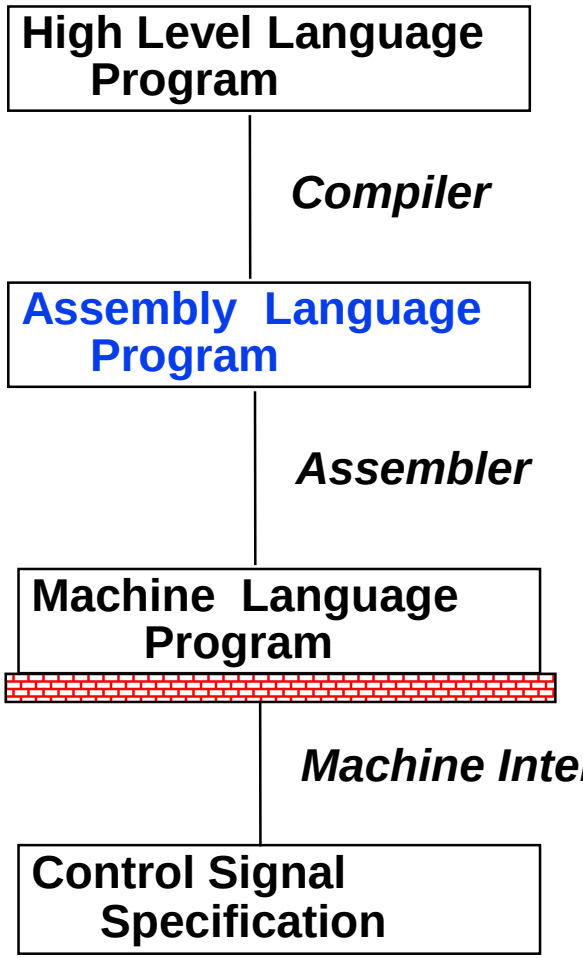
[BO14]

- Abstraktionen durch Betriebssystem



[BO14]

Das Kompilierungssystem




```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw  $15,   0($2)
lw  $16,   4($2)
sw $16,   0($2)
sw $15,   4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

ALUOP[0:3] <= InstReg[9:11] & MASK

[PH14]

Rechnerarchitektur...

- Speicherorganisation
  - Adressierung, Wortbreite, Datentypen
  - Big Endian vs. Little Endian
  - Memory-Map
- Befehlssatz
  - Registermodell
  - Befehlsklassen
  - Befehlsformate und -Codierung
  - Adressierungsarten
- Beispiel: Intel x86
- Analyse und Bewertung einer ISA

Speicherorganisation

- Adressierung
- Wortbreite, Speicherkapazität
- „Big Endian“ / „Little Endian“
- „Alignment“
- „Memory-Map“
- Beispiel: PC mit Windows

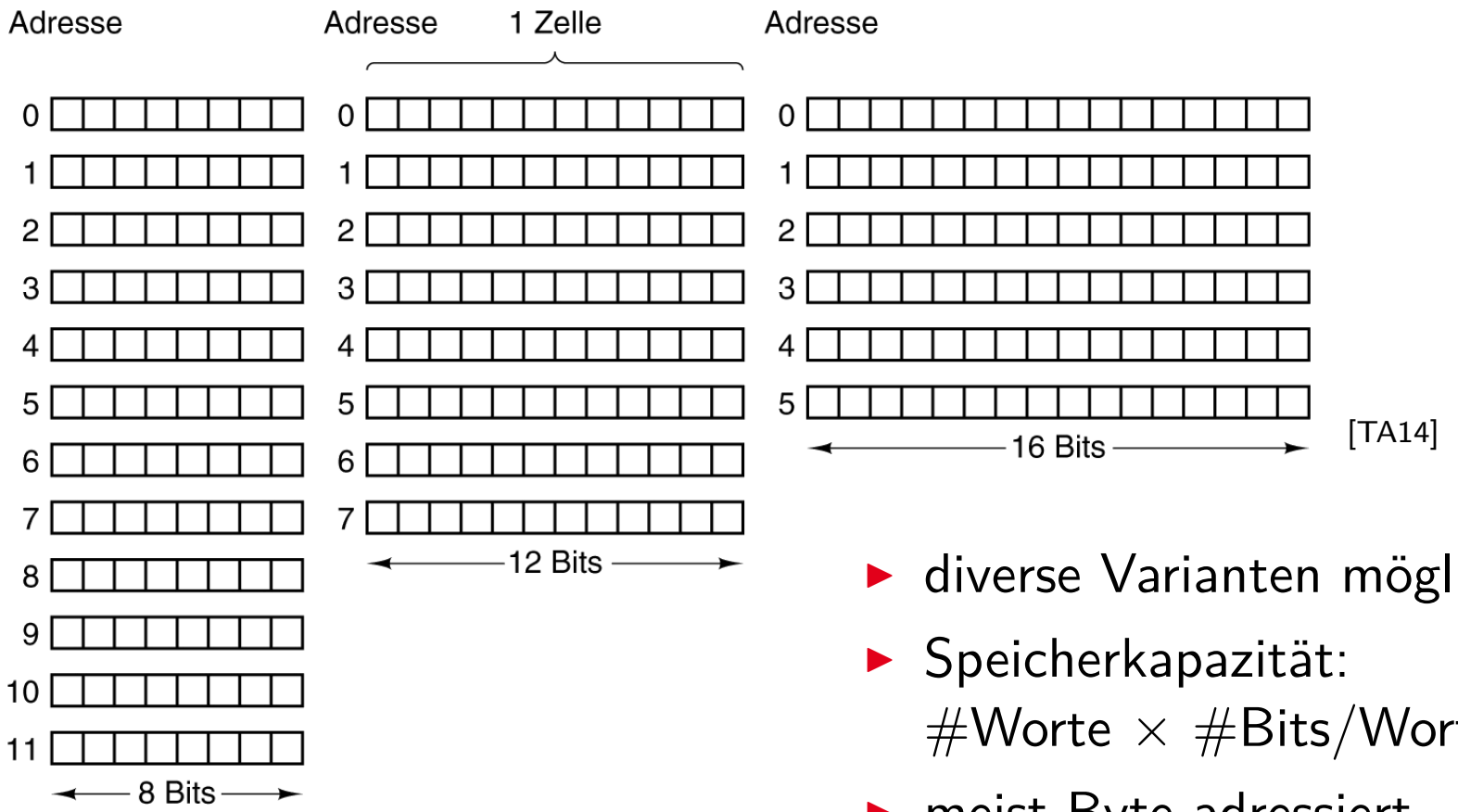
- spätere Themen
  - Cache-Organisation für schnelleren Zugriff
  - Virtueller Speicher für Multitasking
  - Synchronisation in Multiprozessorsystemen (MESI-Protokoll)

Aufbau und Adressierung des Speichers

- 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

Hauptspeicherorganisation

3 Organisationsformen eines 96-bit Speichers: 12×8, 8×12, 6×16 Bits



[TA14]

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

Wortbreite

- Speicherwortbreiten historisch wichtiger Computer

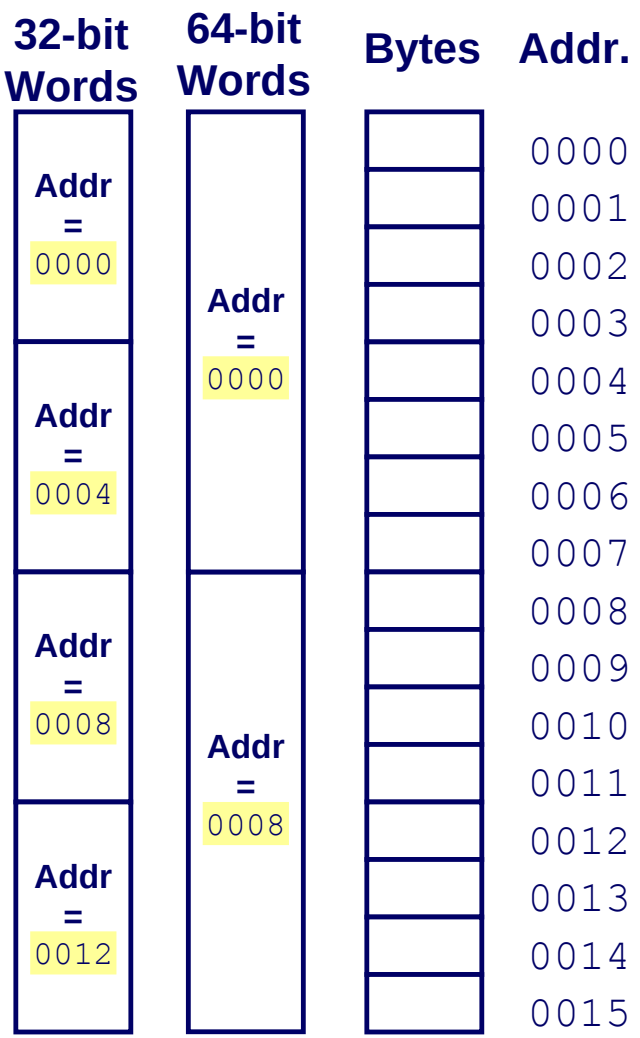
Computer	Bits/Speicherzelle
Burroughs B1700	1
IBM PC	8
DEC PDP-8	12
IBM 1130	16
DEC PDP-15	18
XDS 940	24
Electrologica X8	27
XDS Sigma 9	32
Honeywell 6180	36
CDC 3600	48
CDC Cyber	60

- 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

[BO14]



Datentypen auf Maschinenebene

- 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

Datentypen auf Maschinenebene (cont.)

Abhängigkeiten (!)

- Prozessor
- Betriebssystem
- Compiler

segment word size	16 bit			32 bit						64 bit			
compiler	Microsoft	Borland	Watcom	Microsoft	Intel Windows	Borland	Watcom	Gnu v.3.x	Intel Linux	Microsoft	Intel Windows	Gnu	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				8	8			8	8	8	8	8	8
enum	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
__m128				16	16				16	16	16	16	16
__m256					32				32		32		32
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

Table 1 shows how many bytes of storage various objects use for different compilers.

`www.agner.org/optimize/calling_conventions.pdf`

Byte-Order

- *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

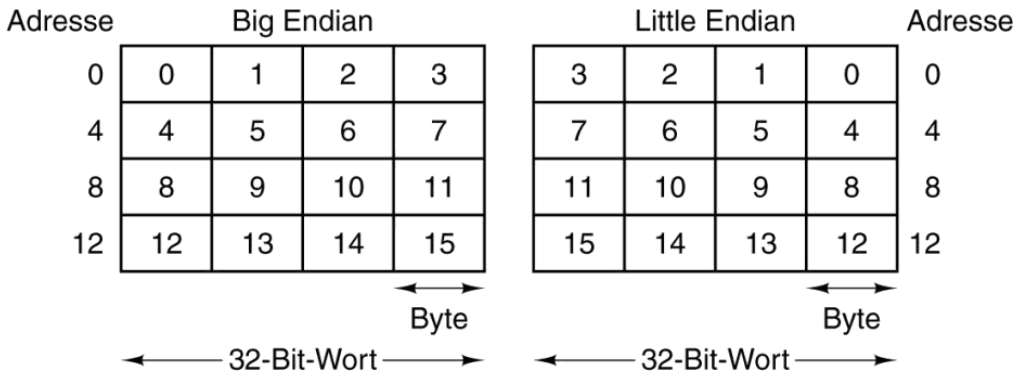
  das LSB (*least significant byte*) hat die höchste –"–

- **Little Endian**: Alpha, x86

  das MSB hat die höchste, das LSB die kleinste Adresse

satirische Referenz auf Gulliver's Reisen (Jonathan Swift)

Big- vs. Little Endian



[TA14]

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

Byte-Order: Beispiel

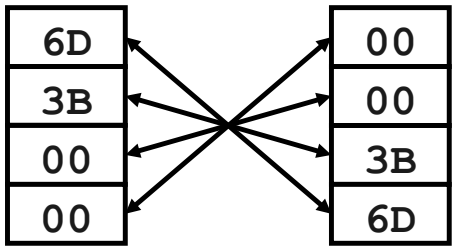
```
      int A =  15213;
      int B = -15213;
long int C =  15213;
```

Dezimal: 15213

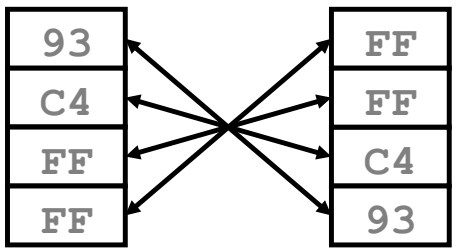
Binär: 0011 1011 0110 1101

Hex: 3 B 6 D

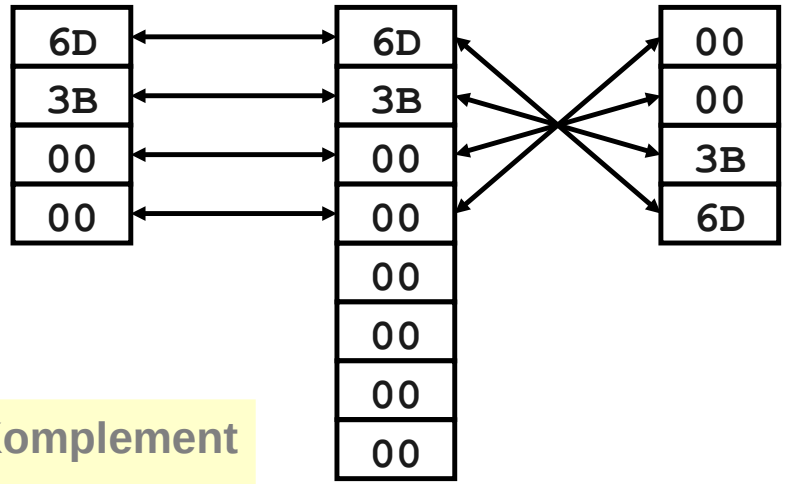
Linux/Alpha A **Sun A**



Linux/Alpha B **Sun B**



Linux C **Alpha C** **Sun C**



2-Komplement

Big Endian

Little Endian

[BO14]

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...
```

Netzwerk Byte-Order

- Byte-Order muss bei Datenübertragung zwischen Rechnern berücksichtigt und eingehalten werden

- Internet-Protokoll (IP) nutzt ein Big Endian Format

$\Rightarrow$ 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
  - ...

Beispiel: Byte-Swapping *network to/from host*

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

```
...
/* 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
...
```

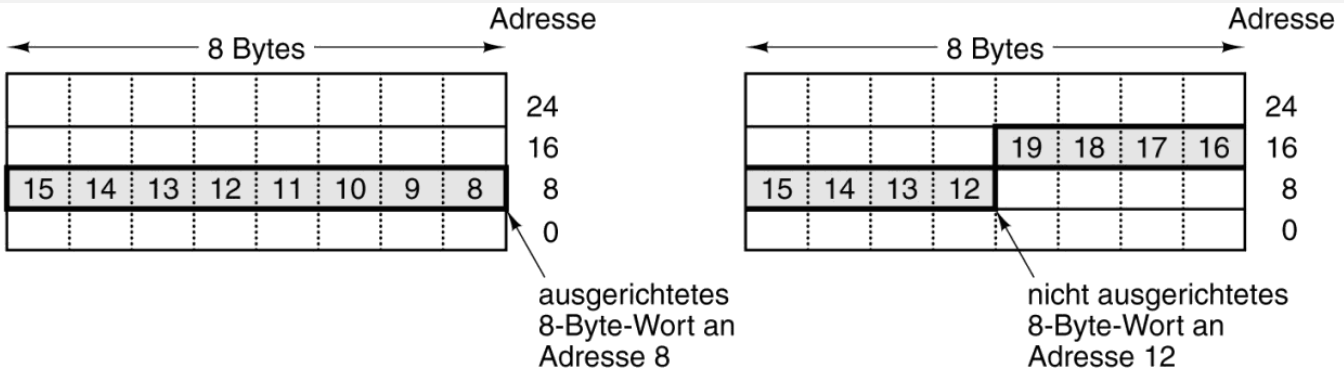
Programm zum Erkennen der Byte-Order

- Programm gibt Daten byteweise aus
- C-spezifische Typ- (Pointer-) Konvertierung
- Details: Bryant, O'Hallaron: 2.1.4 (Abb. 2.3, 2.4) [BO14]

```
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 Wort

⇒ was passiert bei „krummen“ (*misaligned*) Adressen?

- automatische Umsetzung auf mehrere Zugriffe (x86)
- Programmabbruch (SPARC)

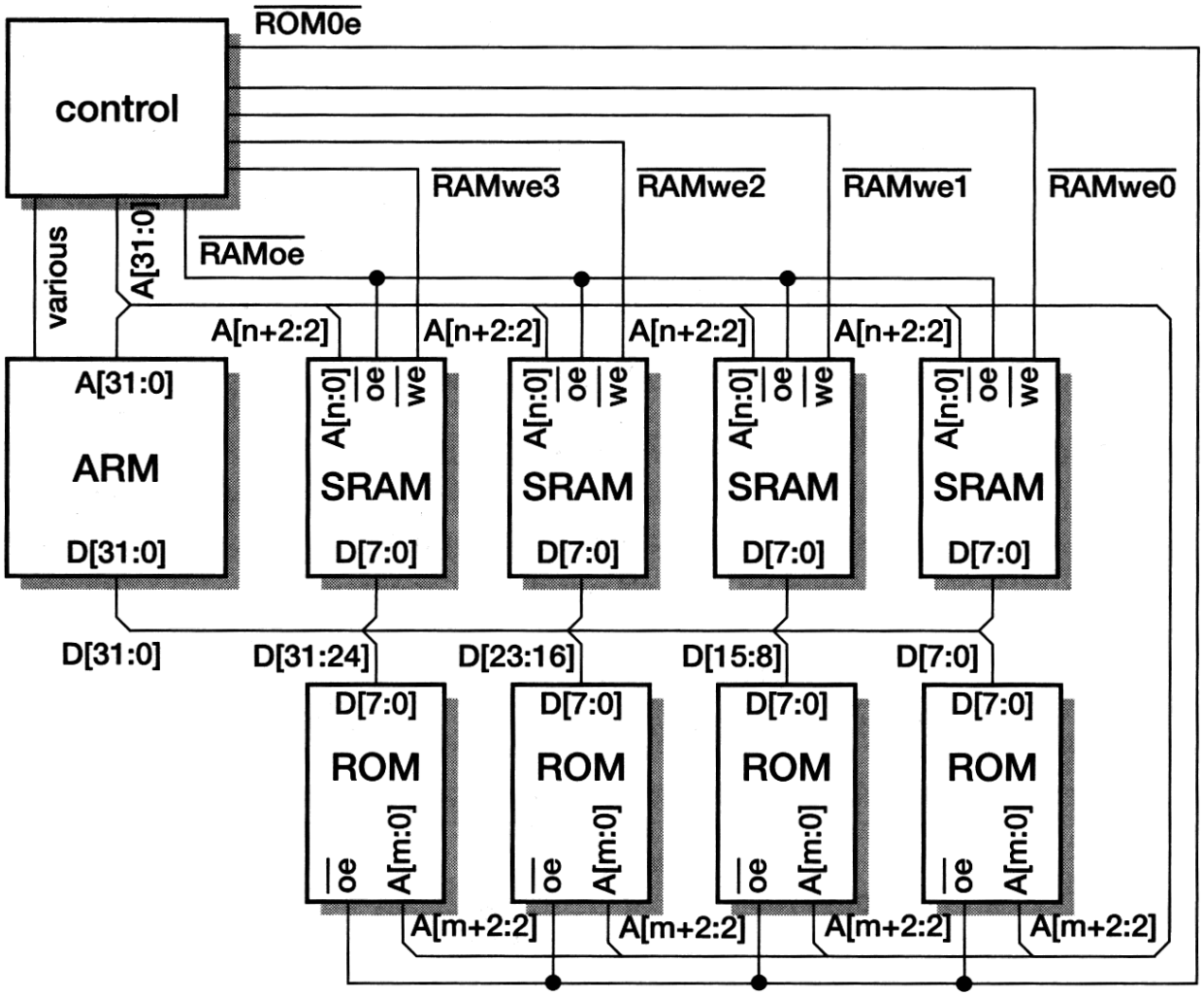
Memory Map

- 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

$\Rightarrow$ „Memory Map“
  - Adressdecoder
  - Hardwareeinheit
  - Zuordnung von Adressen zu „realem“ Speicher

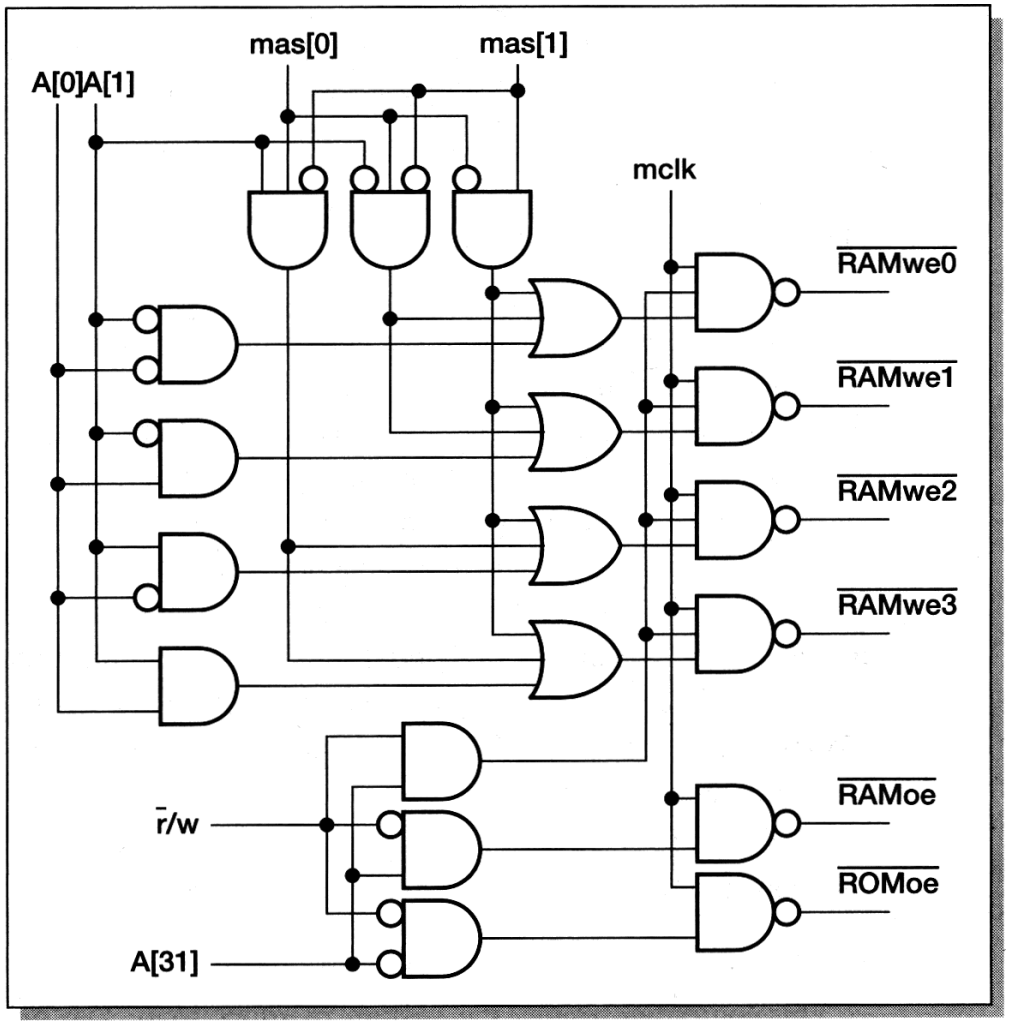
Memory Map: typ. 32-bit System



32-bit Prozessor
4× 8-bit SRAMs
4× 8-bit ROMs

[Fur00]

Memory Map: Adressdecodierung



[Fur00]

Memory Map: typ. 16-bit System

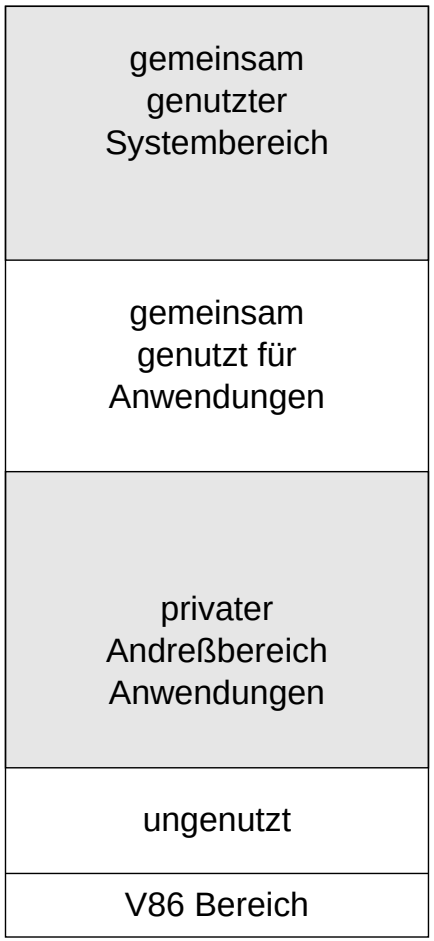
- 16-bit erlaubt 64K Adressen: 0x0000...0xFFFF

- 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)

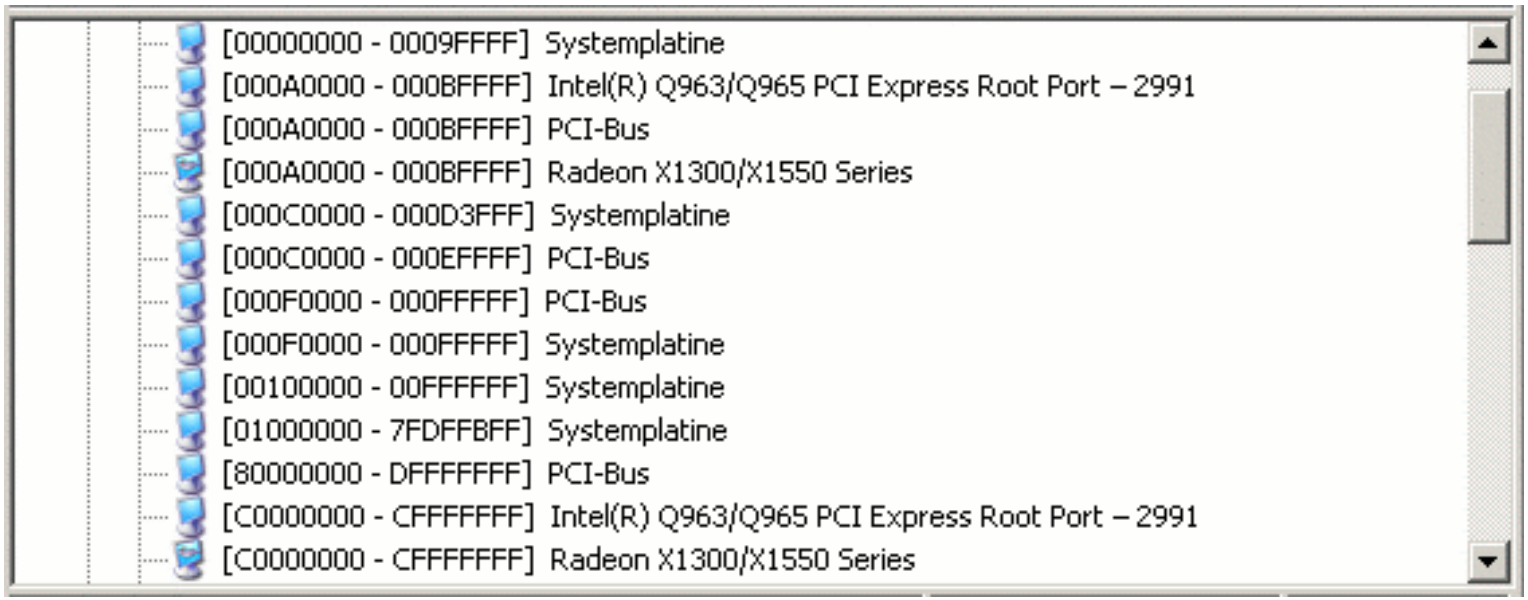
Memory Map: Windows 9x



- Kernel, Treiber, usw. im oberen 1 GiB-Bereich
- 2 GiB für Anwendungen
- DOS-Bereich immer noch für Boot + Geräte

Memory Map: Windows 9x (cont.)

- 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

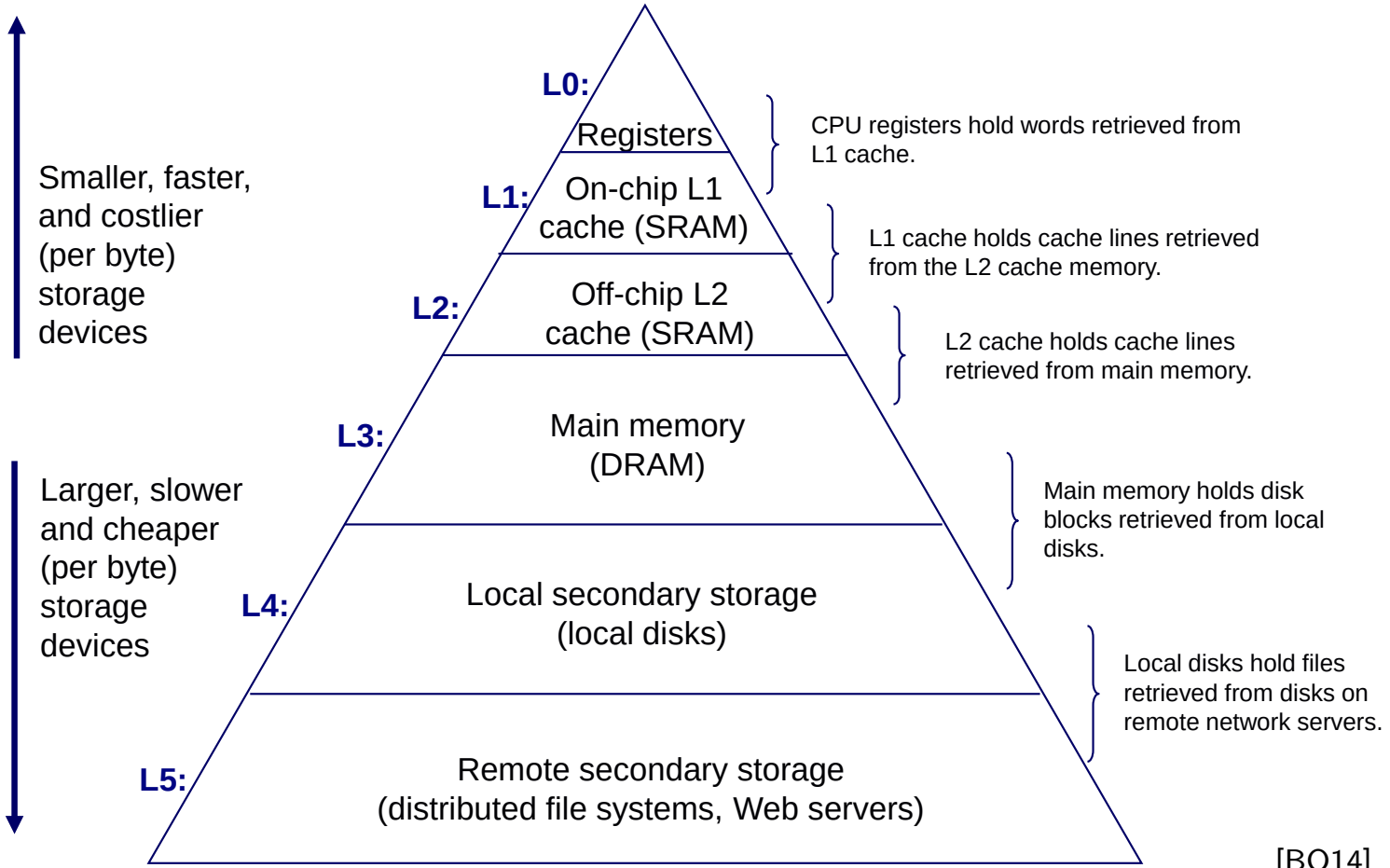


Memory Map: Windows 9x (cont.)

I/O-Speicherbereiche

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

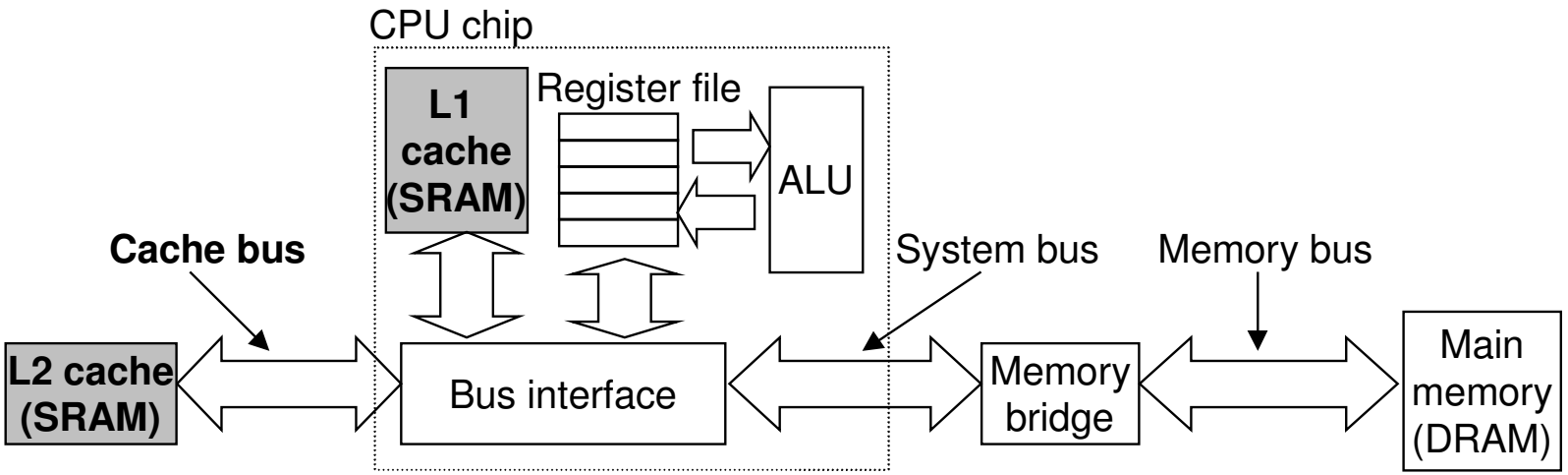
Speicherhierarchie



[BO14]

später mehr. . .

Cache-Speicher



[BO14]

- 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.

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 (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 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

Welche Befehle braucht man?

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, jmpeq, branch, call, return
▶ Maschinensteuerung	trap, halt, (interrupt)

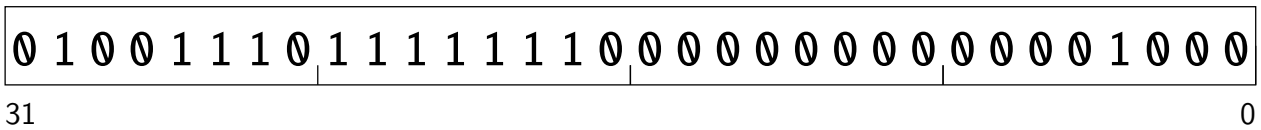
⇒ Befehlssätze und Computerarchitekturen (Details später)

CISC – Complex Instruction Set Computer

RISC – Reduced Instruction Set Computer

Befehls-Decodierung

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



Wie soll die Hardware diesen Wert interpretieren?

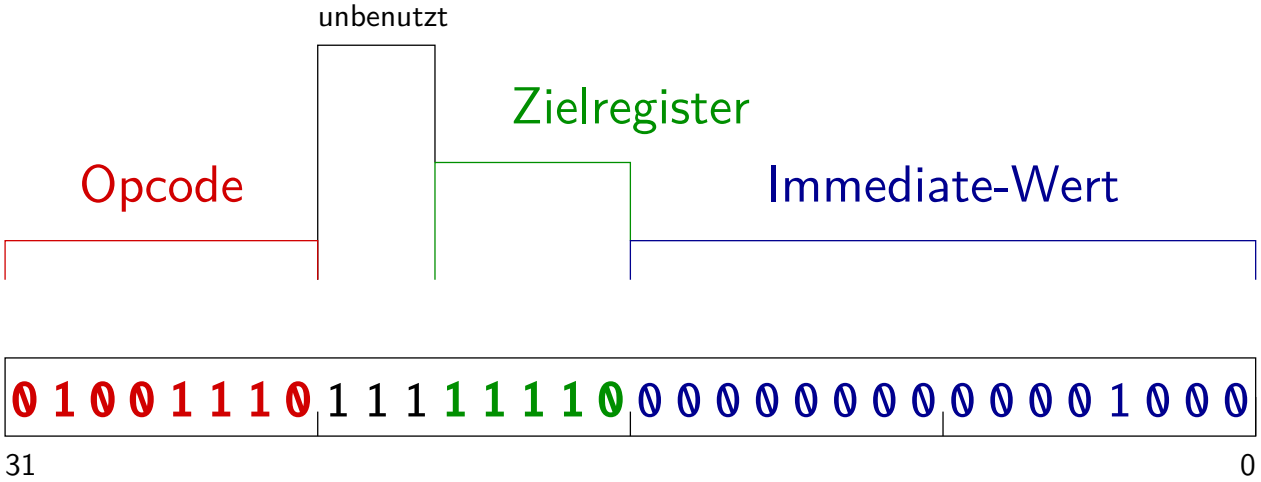
- direkt in einer Tabelle nachschauen (Mikrocode-ROM)
- Problem: Tabelle müsste 2^{32} Einträge haben

$\Rightarrow$ Aufteilung in Felder: Opcode und Operanden

$\Rightarrow$ Decodierung über mehrere, kleine Tabellen

$\Rightarrow$ 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	`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

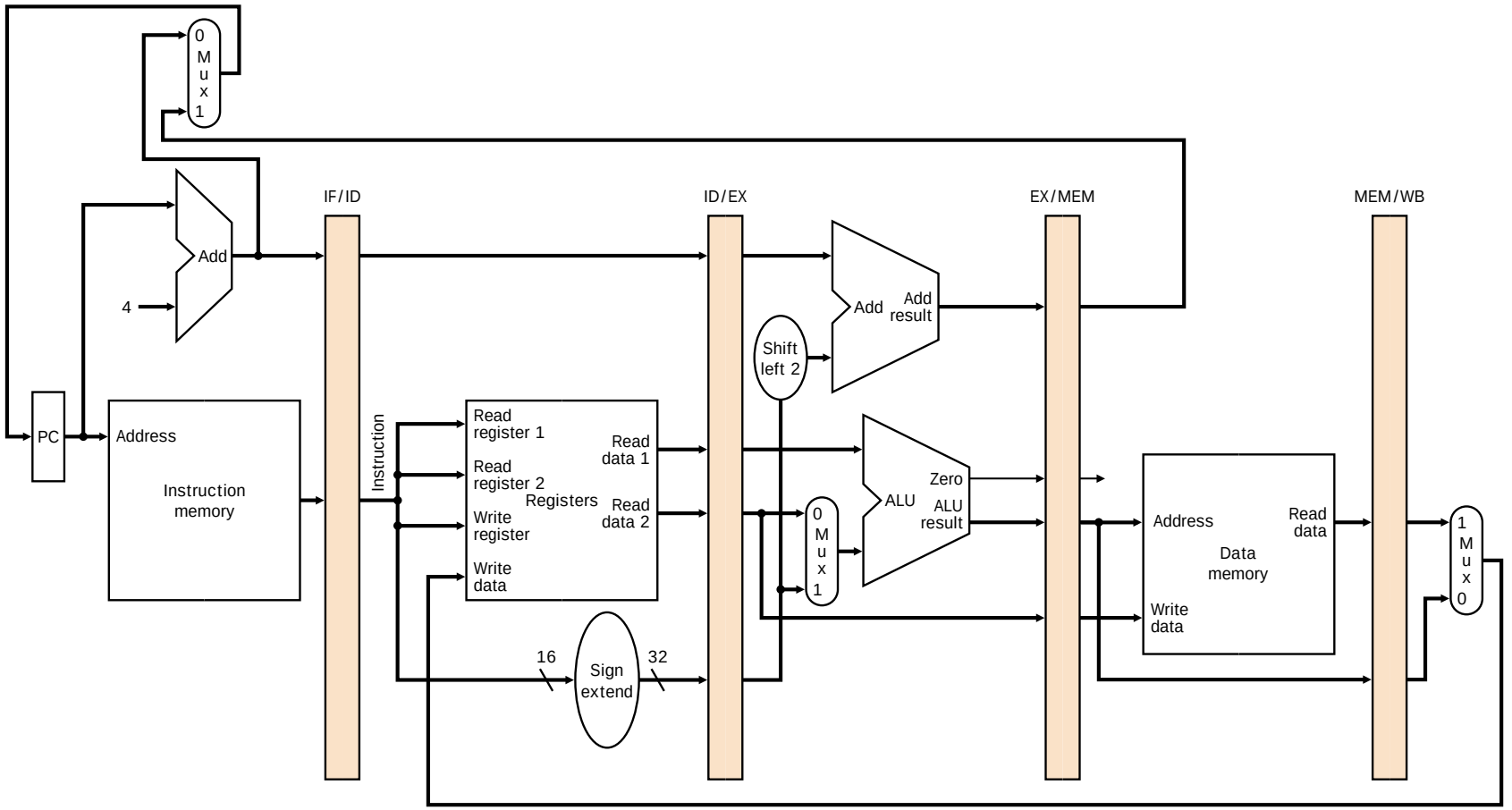
R1 = (R2 < R3)	`slt R1, R2, R3`

MIPS: Befehlssatz

- Übersicht und Details: [PH14, PH11]
  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



[PH14]

PC
I-Cache

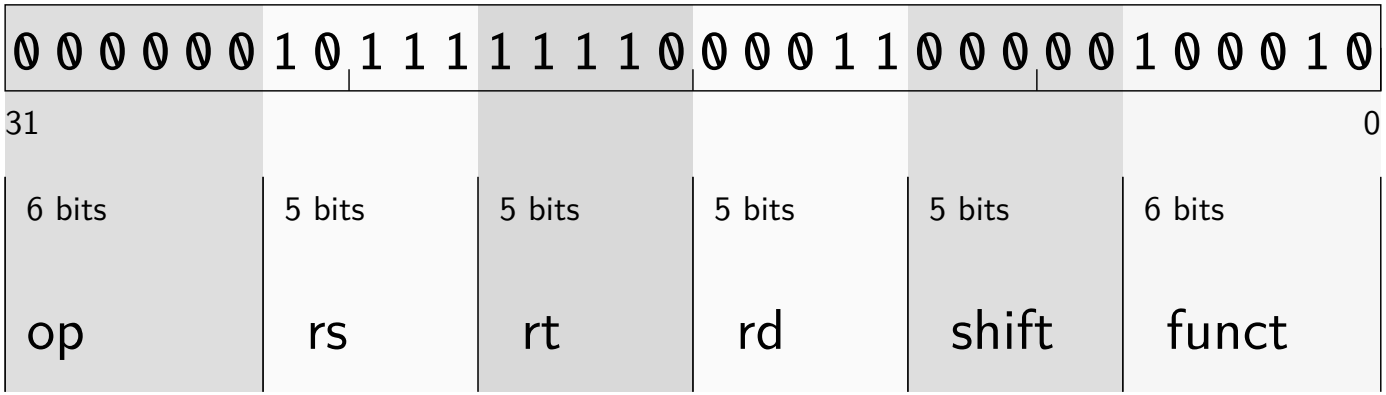
Register
(R0 .. R31)

ALUs

Speicher
D-Cache

MIPS: Befehlsformate

Befehl im R-Format

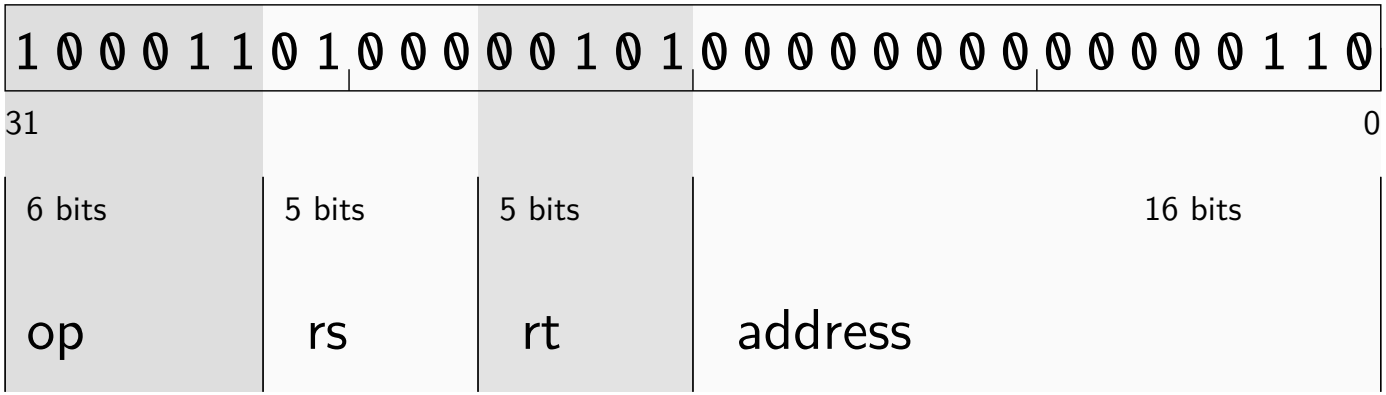


- op: Opcode — Typ des Befehls — 0 = „alu-op“
  rs: source register 1 — erster Operand — 23 = „r23“
  rt: source register 2 — zweiter Operand — 30 = „r30“
  rd: destination register — Zielregister — 3 = „r3“
  shift: shift amount — (optionales Shiften) — 0 = „0“
  funct: ALU function — Rechenoperation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

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
  - Mikrocontroller für eingebettete Systeme
    z.B. *„Smart Cards“*
  - siehe `en.wikipedia.org/wiki/M.CORE`

D·CORE

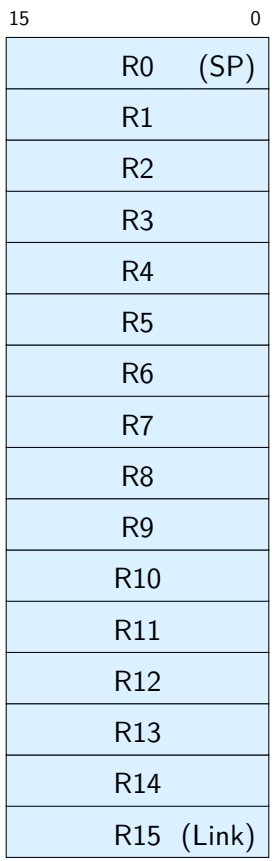
- ähnlich M·CORE
- gleiches Registermodell, aber nur 16-bit Wortbreite
  - Program Counter PC
  - 16 Universalregister R0...R15
  - Statusregister C („carry flag“)

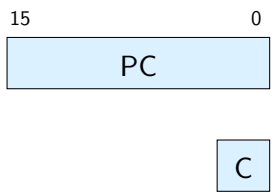
- Subset der Befehle, einfachere Codierung
- vollständiger Hardwareaufbau in Hades verfügbar
  - [HenHA] Hades Demo: `60-dcore/t3/chapter`

oder Simulator mit Assembler (alt)

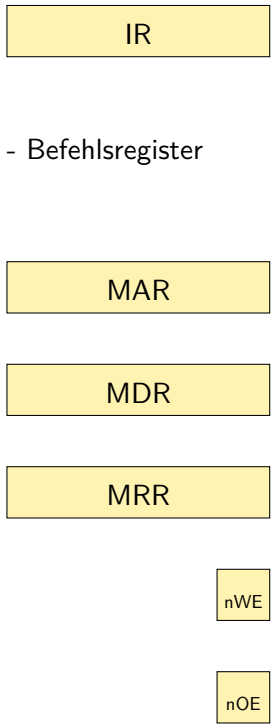
- `tams.informatik.uni-hamburg.de/publications/onlineDoc` (`winT3asm.exe` / `t3asm.jar`)

D·CORE: Registermodell





- 16 Universalregister
- Programmzähler
- 1 Carry-Flag



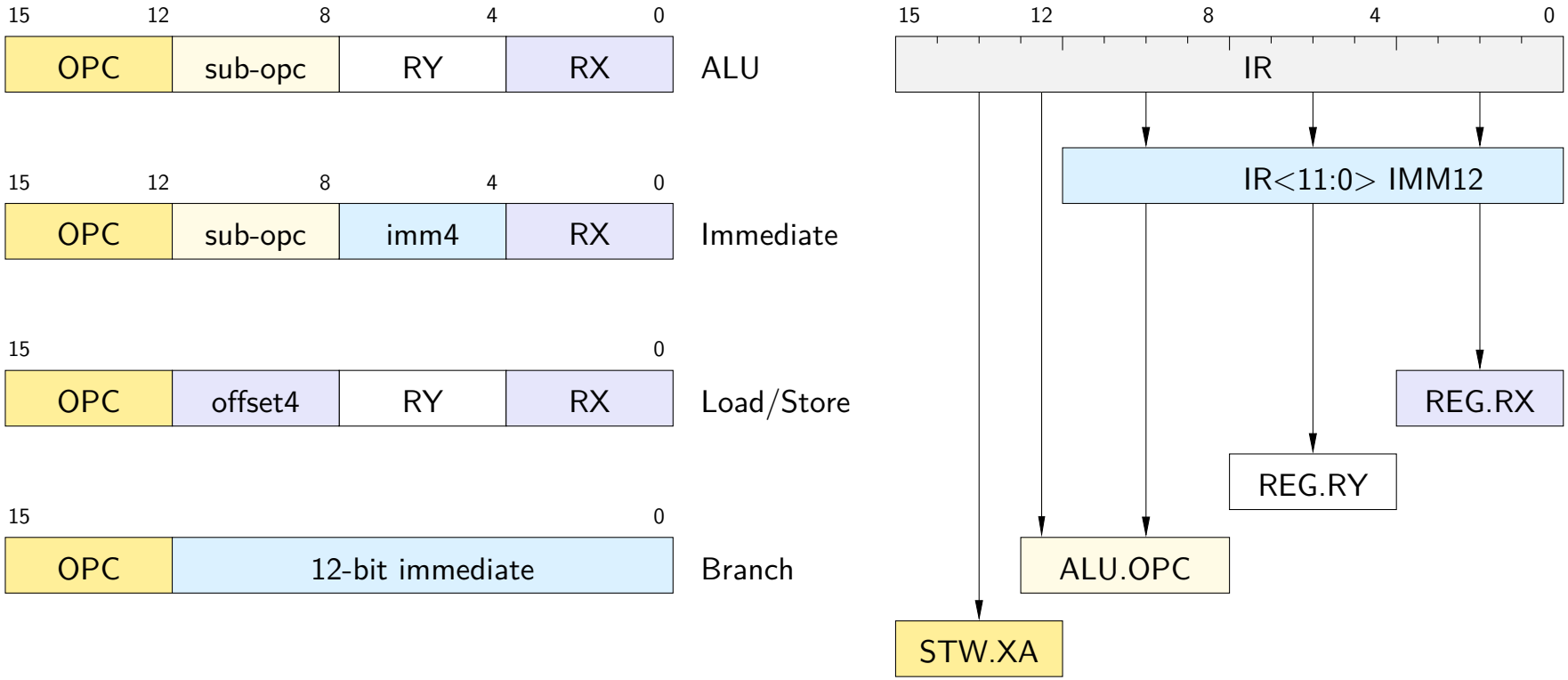
- Befehlsregister
- Bus-Interface

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

D·CORE: Befehlssatz

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

D·CORE: Befehlsformate



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

Adressierungsarten

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

$\Rightarrow$ wichtige Unterscheidungsmerkmale für Rechnerarchitekturen

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

Beispiel: Add-Befehl

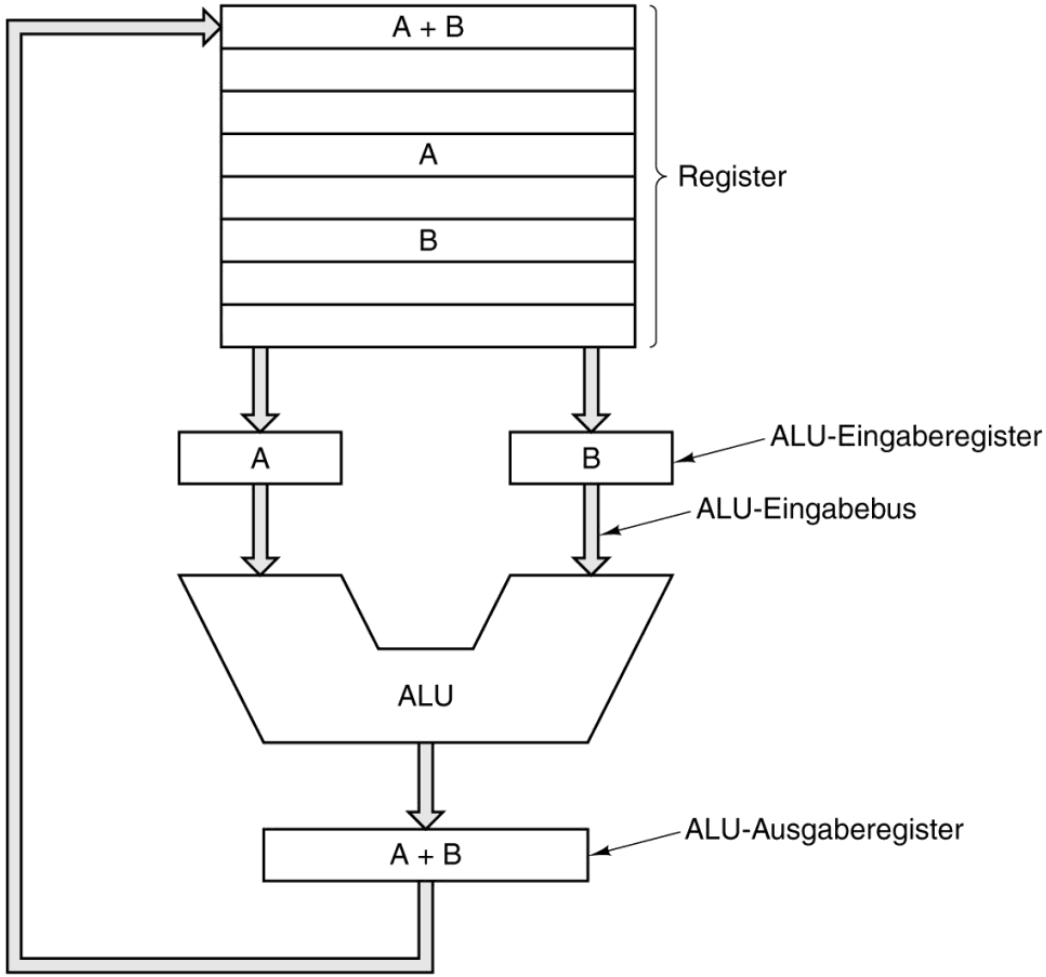
- Rechner soll „rechnen“ können
- typische arithmetische Operation nutzt 3 Variablen
  Resultat, zwei Operanden: $X = Y + Z$

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

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

Beispiel: Datenpfad

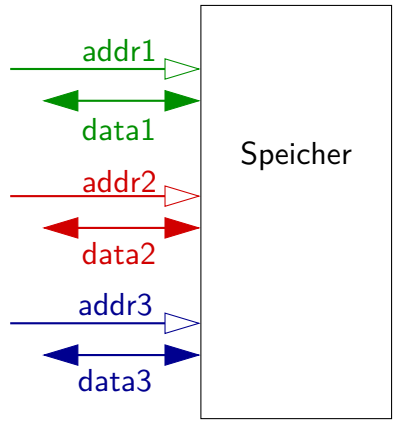
- Register (-bank)
  - liefern Operanden
  - speichern Resultate
- interne Hilfsregister
- ALU, typ. Funktionen:
  - add, add-carry, sub
  - and, or, xor
  - shift, rotate
  - compare
  - (floating point ops.)



[TA14]

Woher kommen die Operanden?

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

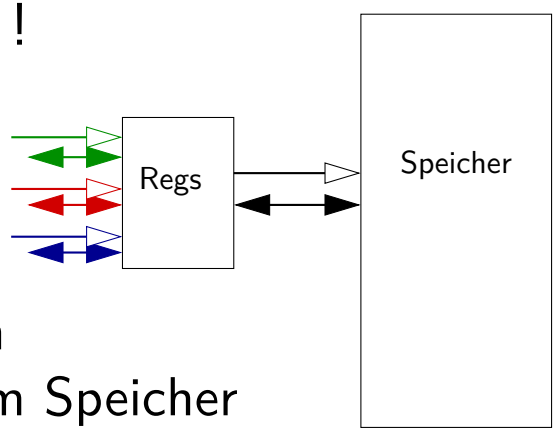


$\Rightarrow$ „Multiport-Speicher“ mit drei Ports ?

- sehr aufwändig, extrem teuer, trotzdem langsam

$\Rightarrow$ Register im Prozessor zur Zwischenspeicherung !

- Datentransfer zwischen Speicher und Registern
  Load reg = MEM[addr]
  Store MEM[addr] = reg



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

n-Adress Maschine $n = \{3 \ldots 0\}$

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, 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

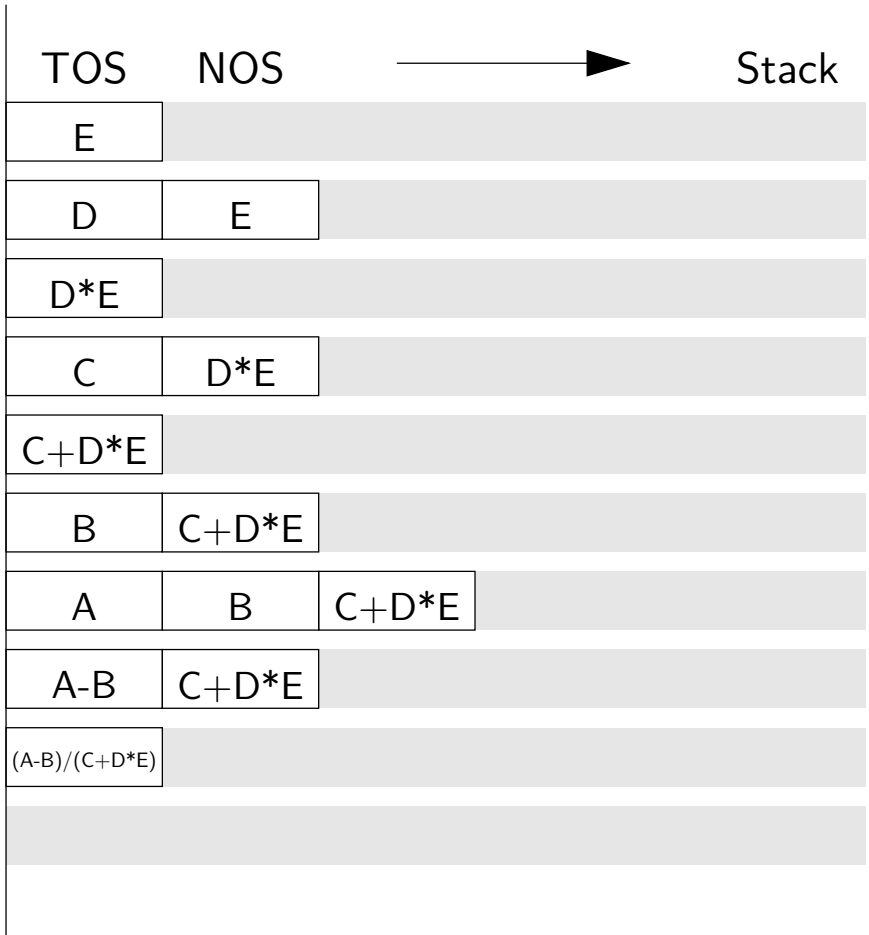
```
push  E
push  D
mul
push  C
add
push  B
push  A
sub
div
pop   Z
```

Beispiel: Stack-Maschine / 0-Adress Maschine

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

0-Adress-Maschine

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



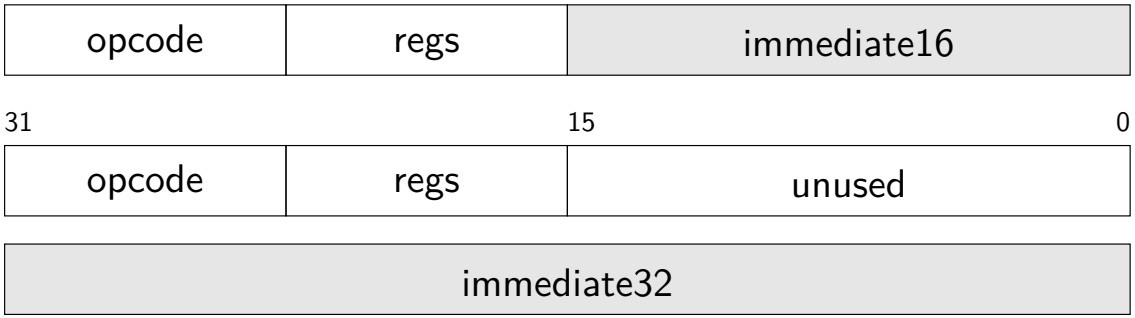
Adressierungsarten

- „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

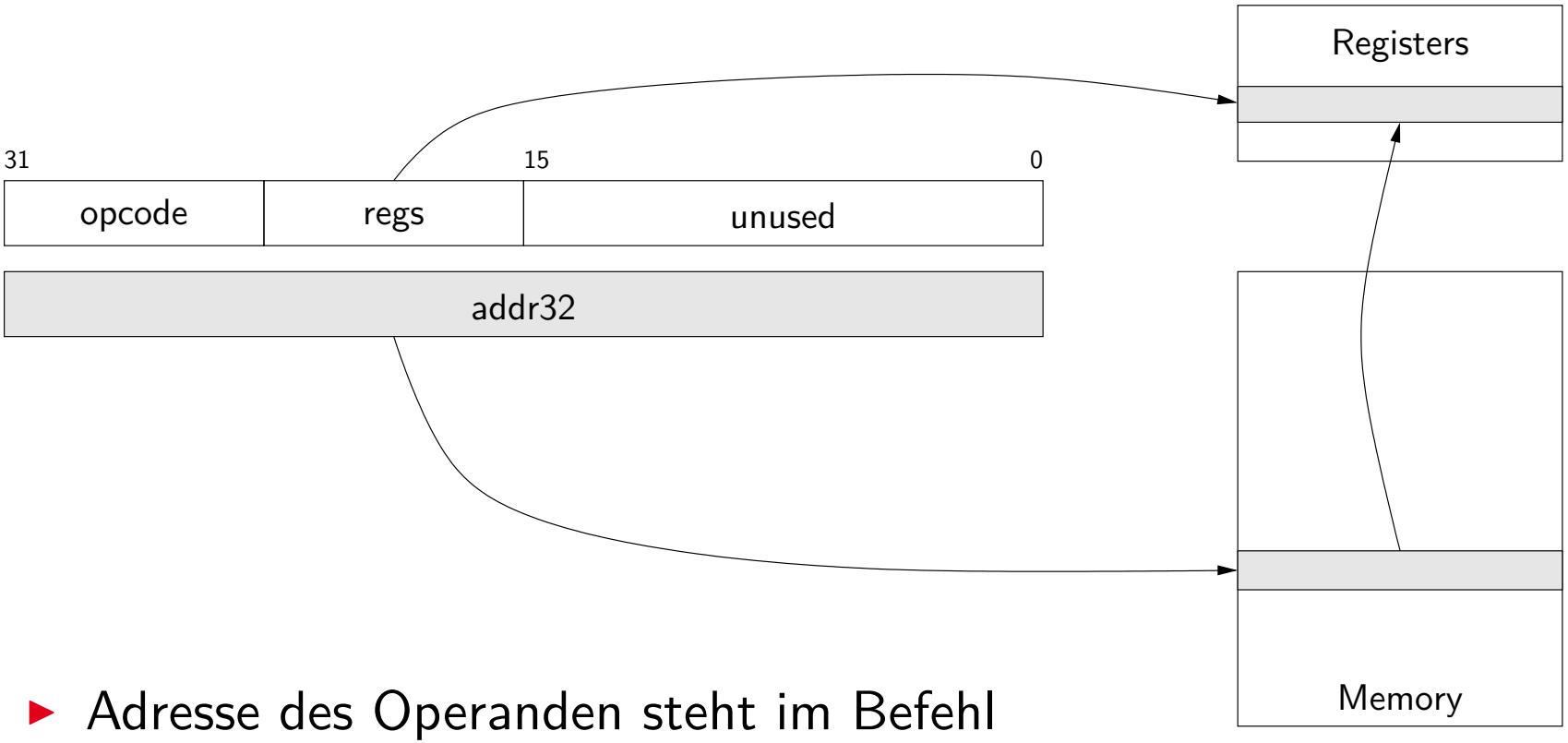


1-Wort Befehl

2-Wort Befehl

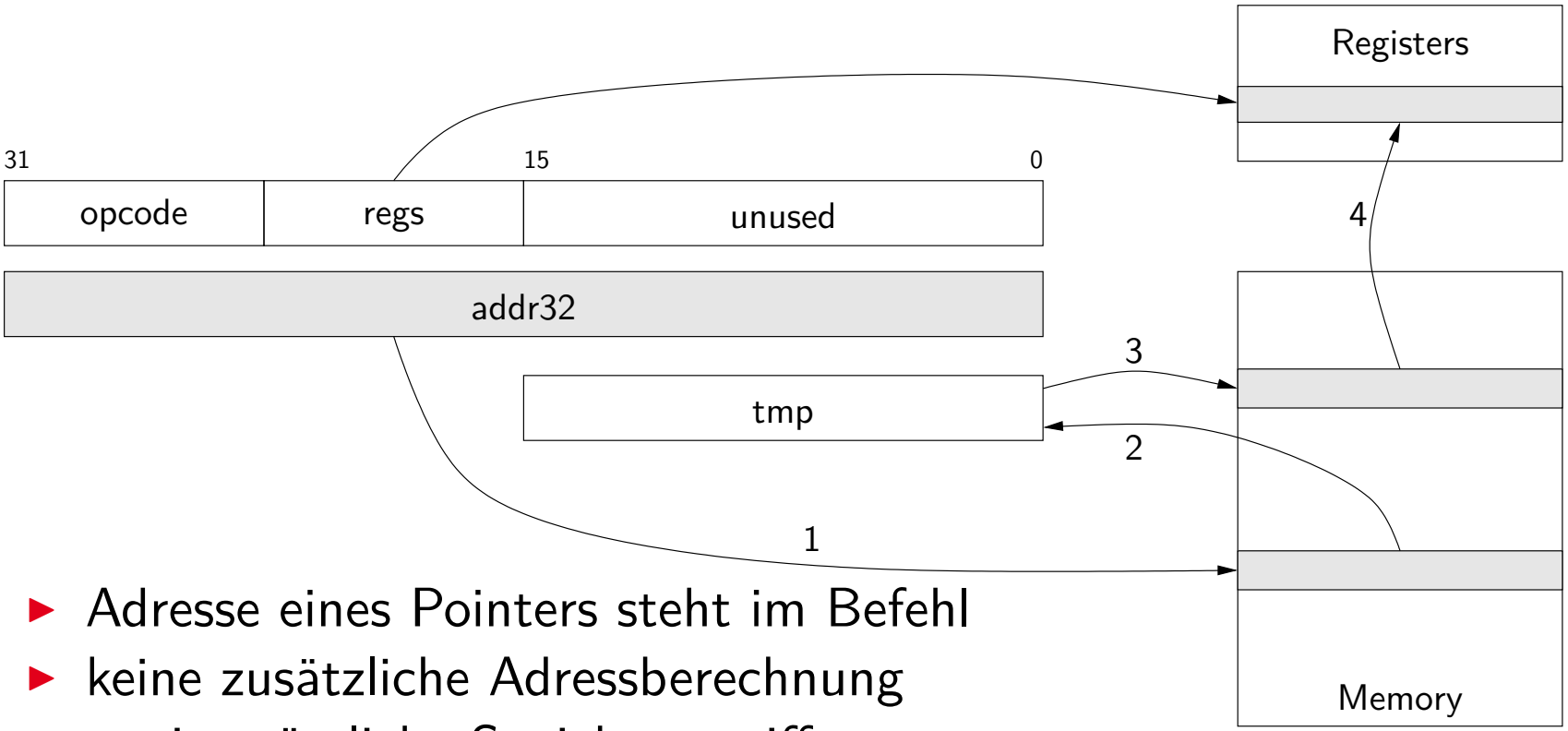
- Operand steht direkt im Befehl, kein zusätzlicher Speicherzugriff
- Länge des Operanden $<$ (Wortbreite - Opcodebreite)
- Darstellung größerer Zahlenwerte
  - 2-Wort Befehle (x86)
    zweites Wort für Immediate-Wert
  - mehrere Befehle (MIPS, SPARC)
    z.B. obere/untere Hälfte eines Wortes
  - Immediate-Werte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

Direkte Adressierung



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

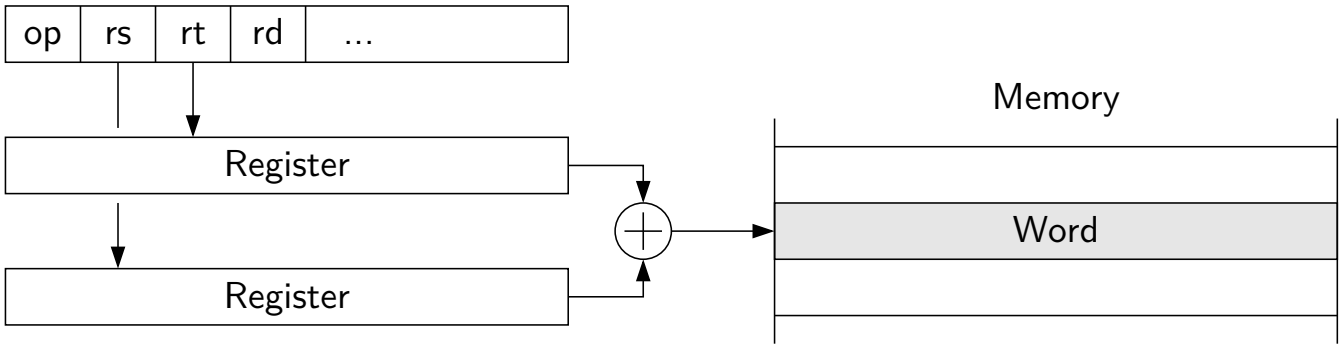
Indirekte Adressierung



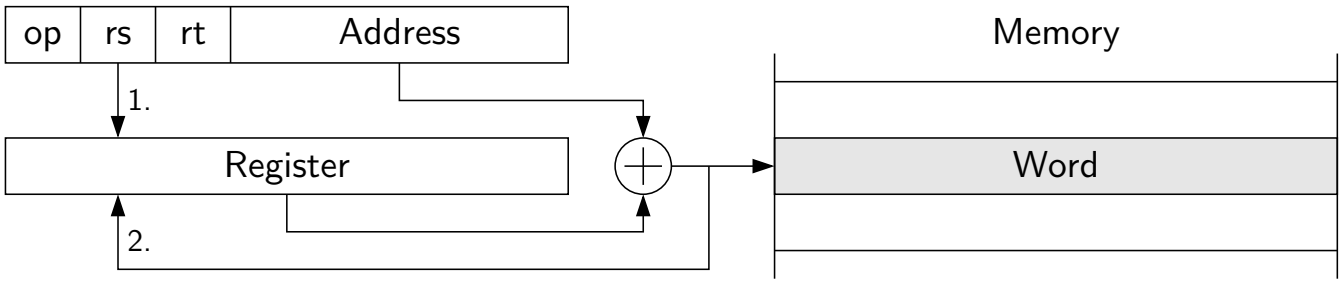
- Adresse eines Pointers steht im Befehl
- keine zusätzliche Adressberechnung
- zwei zusätzliche Speicherzugriffe:
  z.B. tmp = MEM[addr32] R3 = MEM[tmp]
- typische CISC-Adressierungsart, viele Taktzyklen
- kommt bei RISC-Rechnern nicht vor

Indizierte Adressierung

Indexaddressing



Updateaddressing



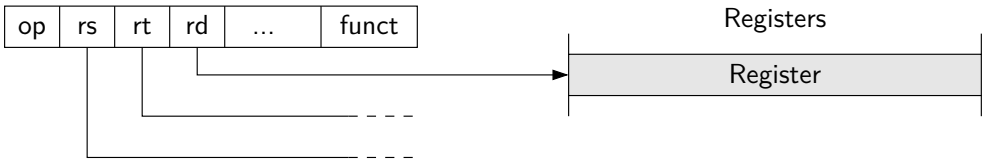
- indizierte Adressierung, z.B. für Arrayzugriffe
  - addr = ⟨*Sourceregister*⟩ + ⟨*Basisregister*⟩
  - addr = ⟨*Sourceregister*⟩ + offset;
    Sourceregister = addr

Beispiel: MIPS Adressierungsarten

1. Immediate addressing

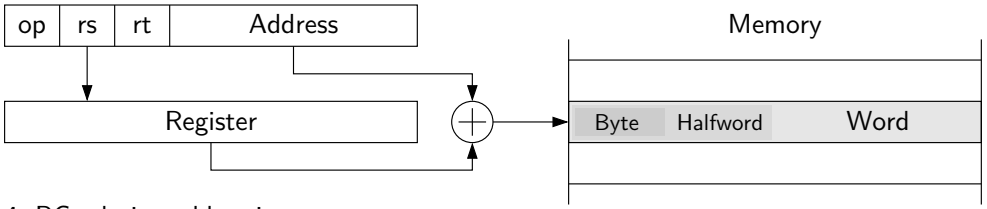
immediate

2. Register addressing



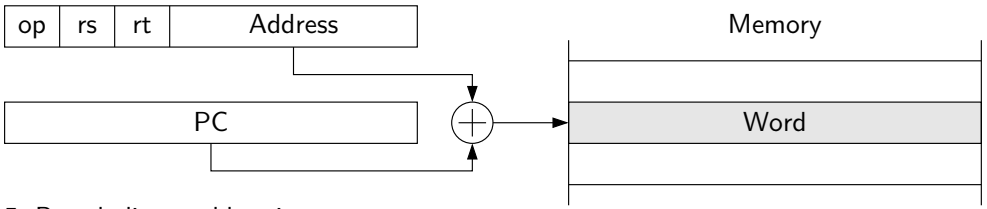
register

3. Base addressing



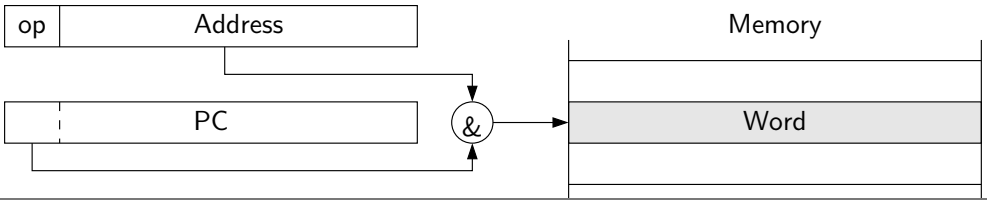
index + offset

4. PC-relative addressing



PC + offset

5. Pseudodirect addressing



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

typische Adressierungsarten

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`

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

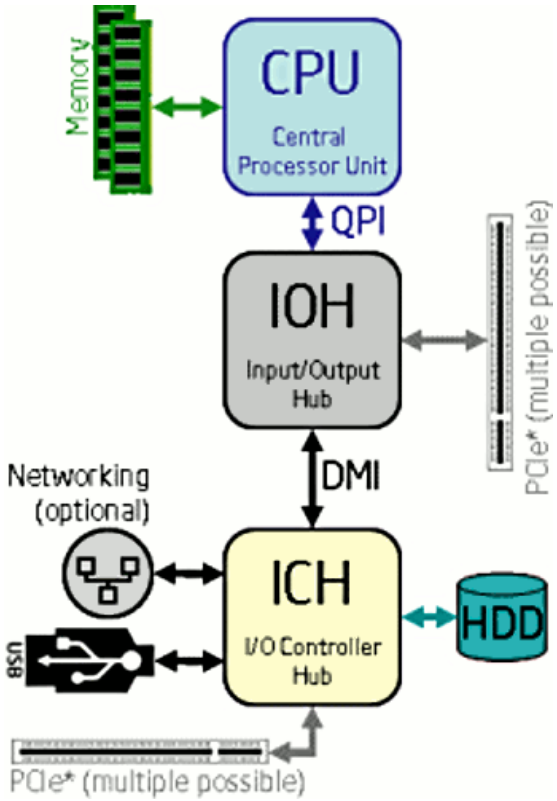
Intel x86: Evolution

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	Speichercontroller, Taktanpassung
...					GPU, I/O-Contr., Spannungsregelung...

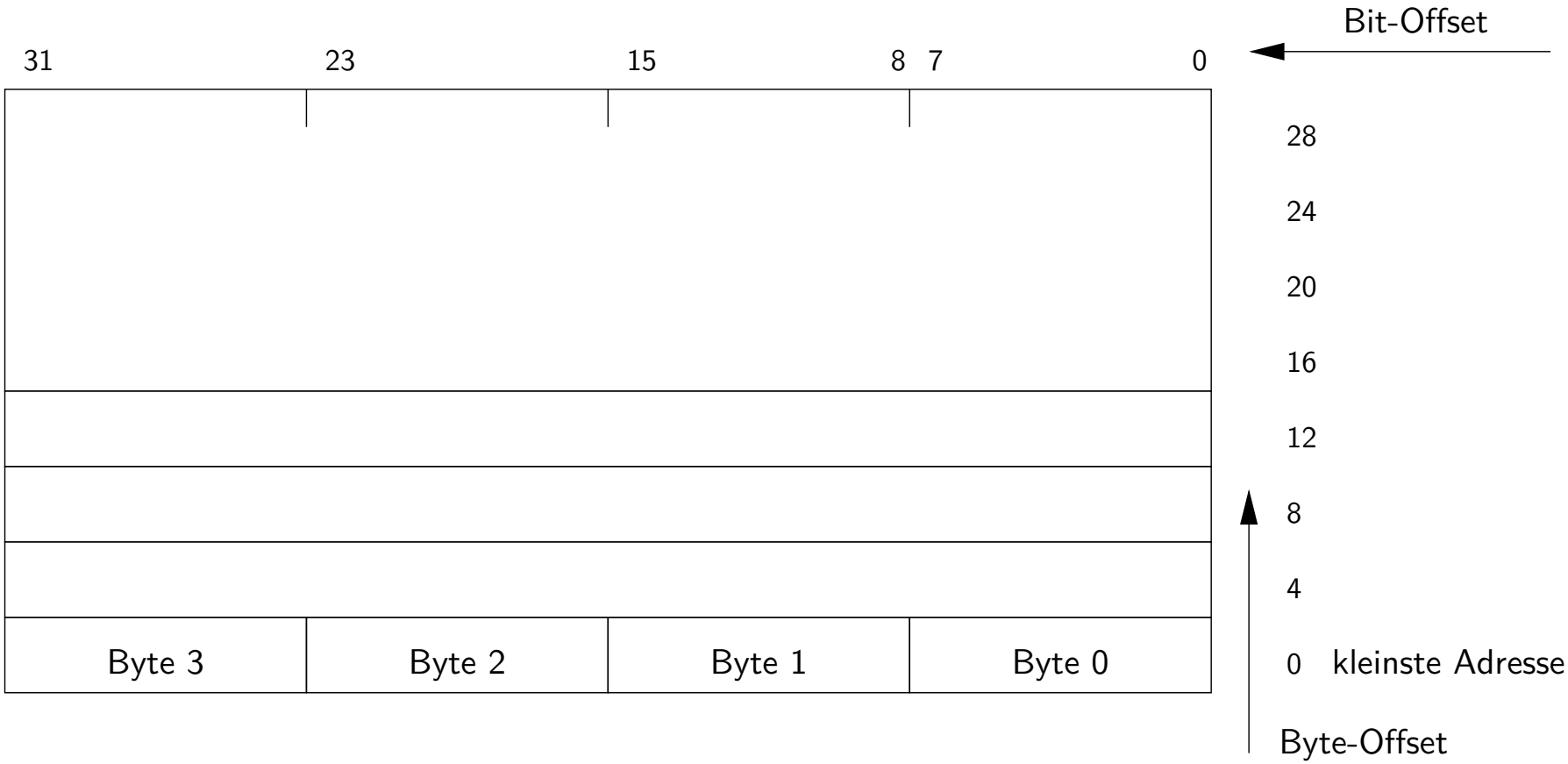
Beispiel: Core i7-960 Prozessor

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

Quellen: `ark.intel.com`, `www.spec.org`



x86: Speichermodell

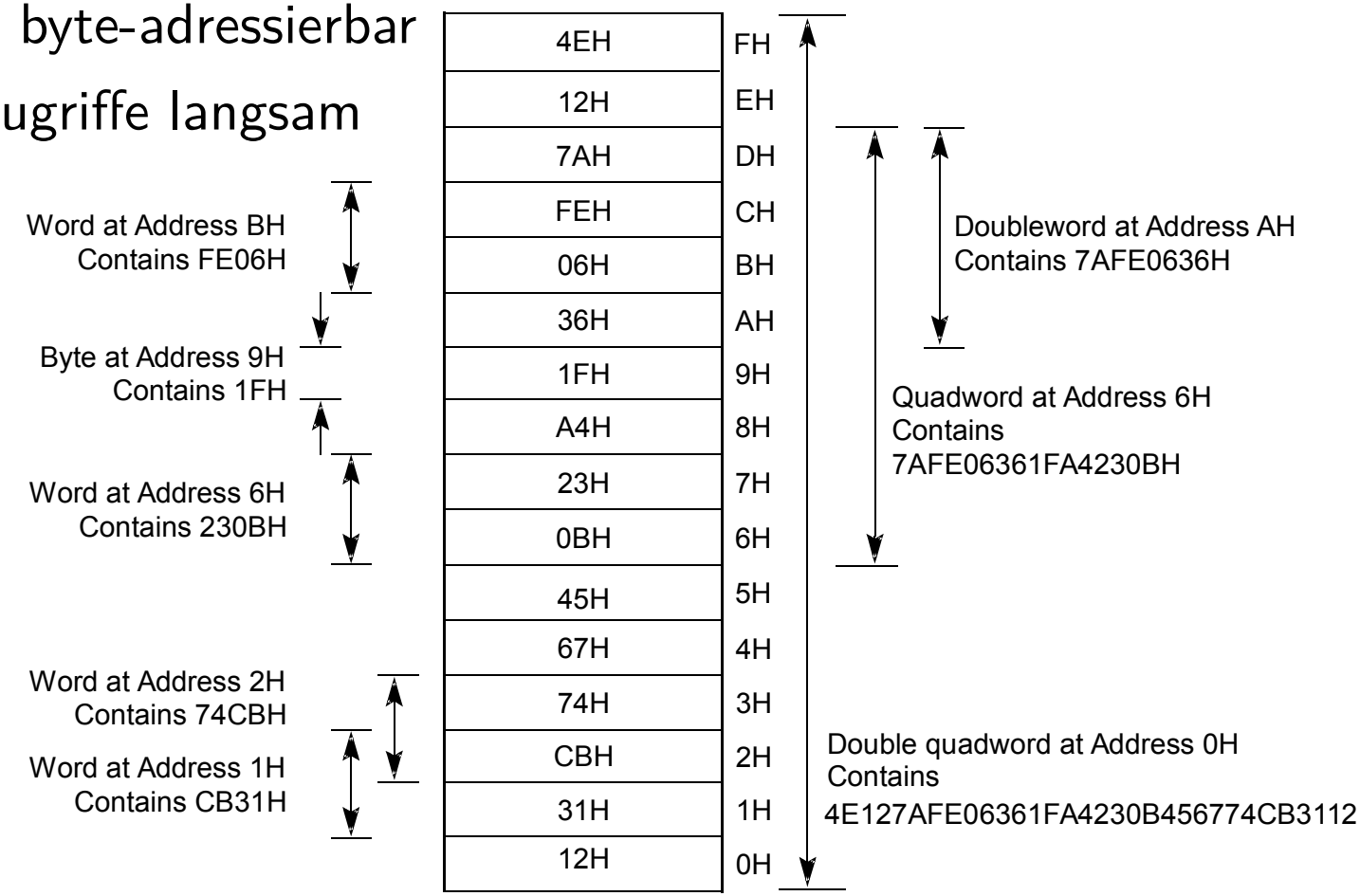


- „little endian“: LSB eines Wortes bei der kleinsten Adresse

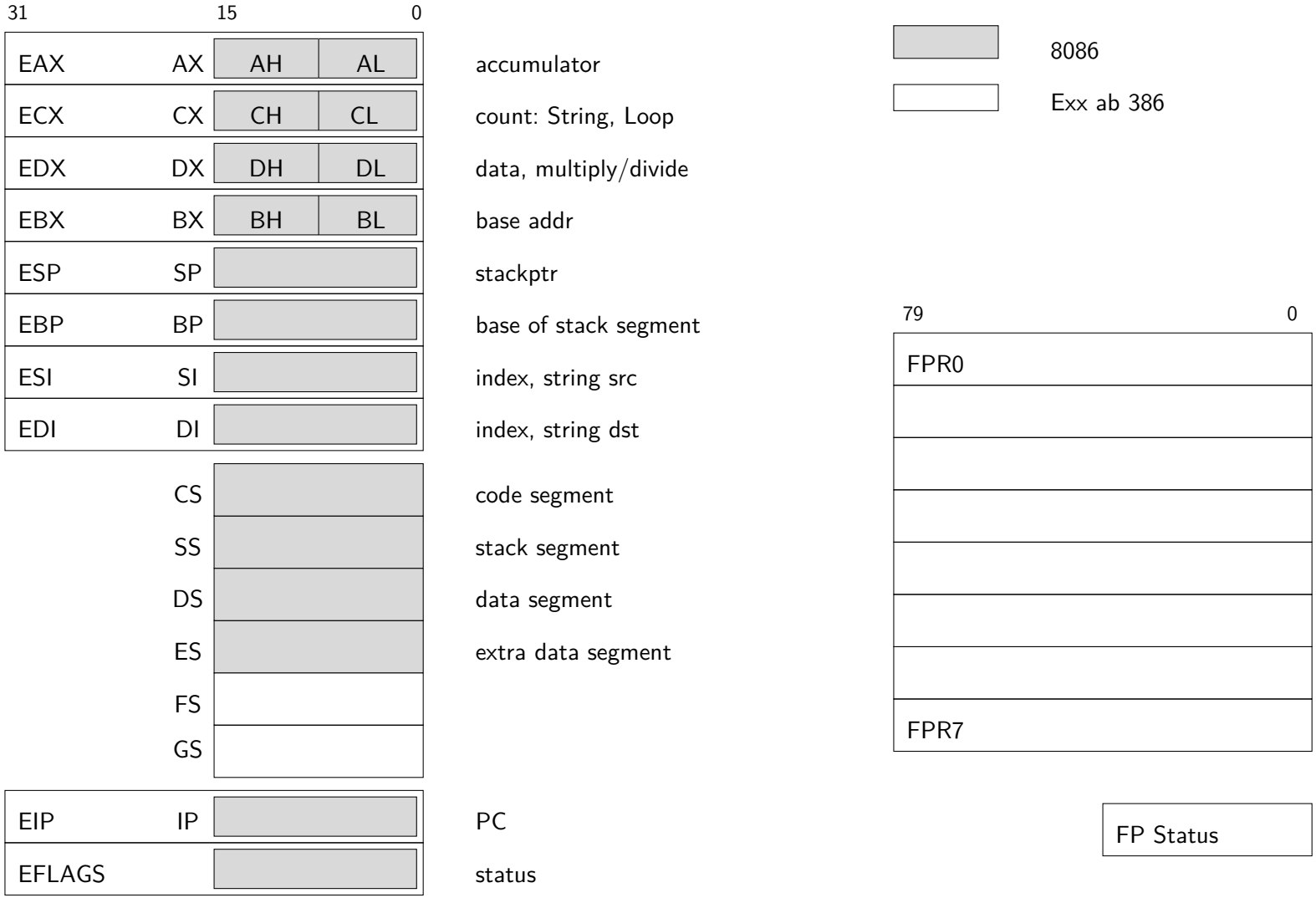
x86: Speichermodell (cont.)

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

[IA64]

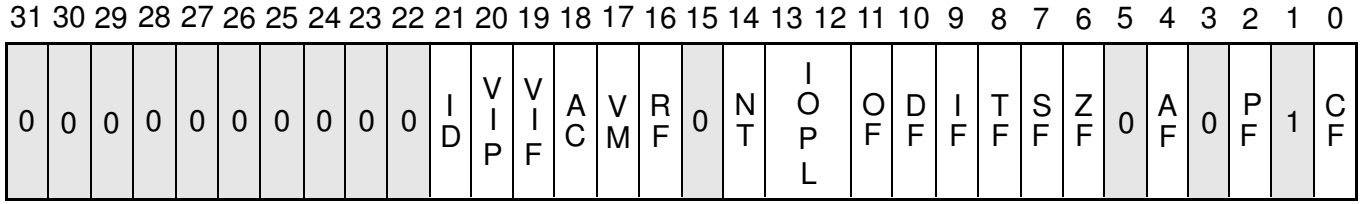


x86: Register



x86: EFLAGS Register

[IA64]



X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)

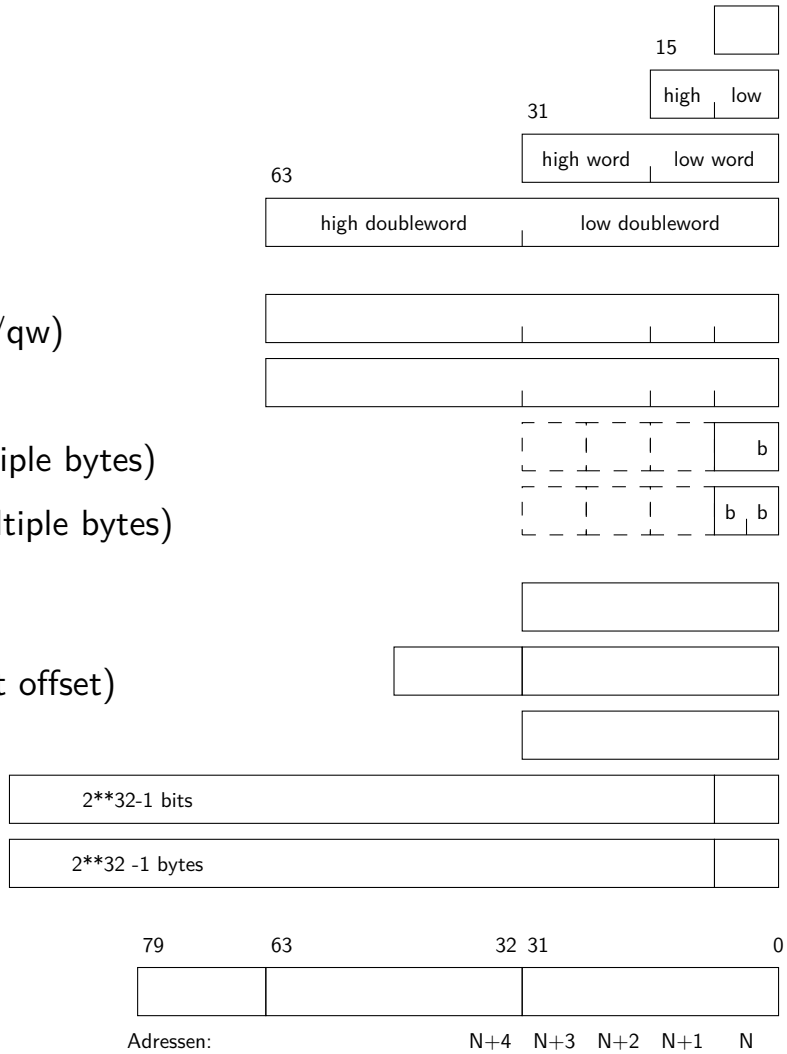
x86: Datentypen

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

- CISC: zusätzlich diverse Ausnahmen/Spezialfälle

x86: Befehlsformate

- 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

$\Rightarrow$ extrem aufwändige Decodierung

$\Rightarrow$ typisches Beispiel für CISC (Complex Instruction Set Computer)

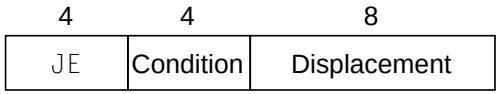
x86: Befehlsformat-Modifier („prefix“)

- 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

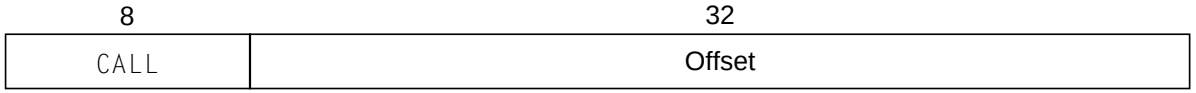
x86 Befehlscodierung: Beispiele

a. JE EIP + displacement

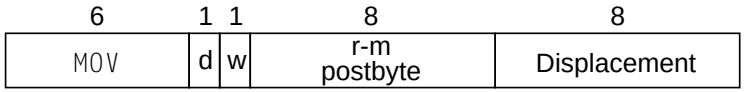


[PH14]

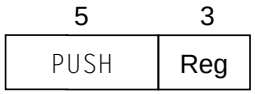
b. CALL



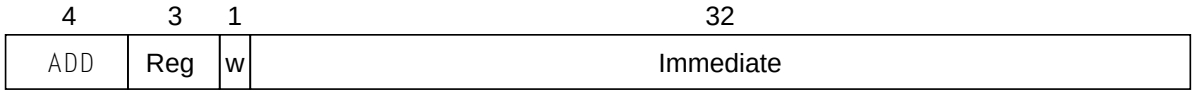
c. MOV EBX, [EDI + 45]



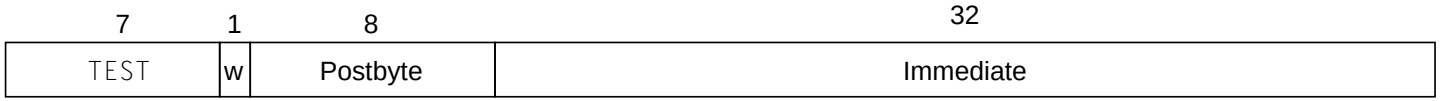
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



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

x86 Befehlscodierung: 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 codea (flags) with EDX & 42
MOVSL	M[EDI] = M[ESI]; EDI = EDI + 4; ESI = ESI + 4

[PH14]

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_putc
0053 43            incl %ebx
0054 83C408        addl $8,%esp
0057 803B00        cmpb $0,(%ebx)
005a 75E8          jne .L24
                .L26:
005c 31C0          xorl %eax,%eax         |   return 0;
005e 8B5DFC        movl -4(%ebp),%ebx     | }
0061 89EC          movl %ebp,%esp
0063 5D            popl %ebp
0064 C3            ret
```

Bewertung der ISA

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

Bewertung der ISA (cont.)

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 %

Bewertung der ISA (cont.)

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%

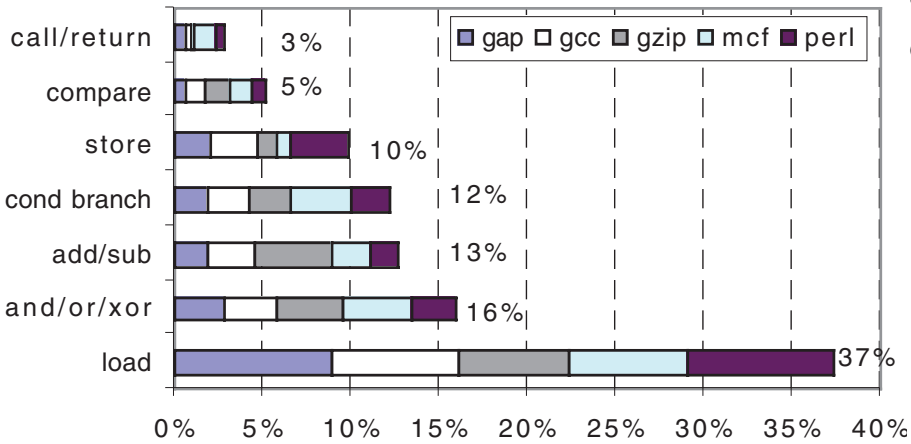
Figure D.15 80x86 instruction mix for five SPECint92 programs.

[HP12]

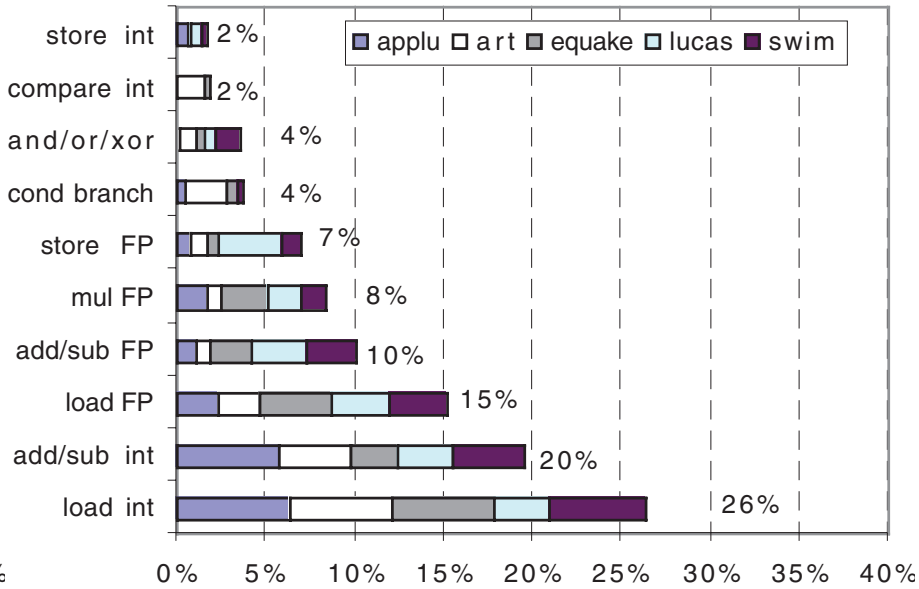
Bewertung der ISA (cont.)

- MIPS-Prozessor

[HP12]



SPECint2000 (96%)



SPECfp2000 (97%)

Bewertung der ISA (cont.)

- ca. 80 % der Berechnungen eines typischen Programms verwenden nur ca. 20 % der Instruktionen einer CPU
- am häufigsten gebrauchten Instruktionen sind einfache Instruktionen: load, store, add...

$\Rightarrow$ Motivation für RISC

CISC: Complex Instruction Set Computer

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 Befehlscodierung
  - mehrere Schreib- und Lesezugriffe pro Befehl
- viele verschiedene Datentypen

CISC: Complex Instruction Set Computer (cont.)

- 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

CISC: Complex Instruction Set Computer (cont.)

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

CISC – Mikroprogrammierung

- ein Befehl kann nicht in einem Takt abgearbeitet werden

$\Rightarrow$ Unterteilung in Mikroinstruktionen ($\varnothing$ 5...7)

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

1. horizontale Mikroprogrammierung
   - langes Mikroprogrammwort (ROM-Zeile)
   - steuert direkt alle Operationen
   - Spalten entsprechen: Kontrollleitungen und Folgeadressen

CISC – Mikroprogrammierung (cont.)

2. vertikale Mikroprogrammierung
   - kurze Mikroprogrammworte
   - Spalten enthalten Mikrooperationscode
   - mehrstufige Decodierung für Kontrollleitungen

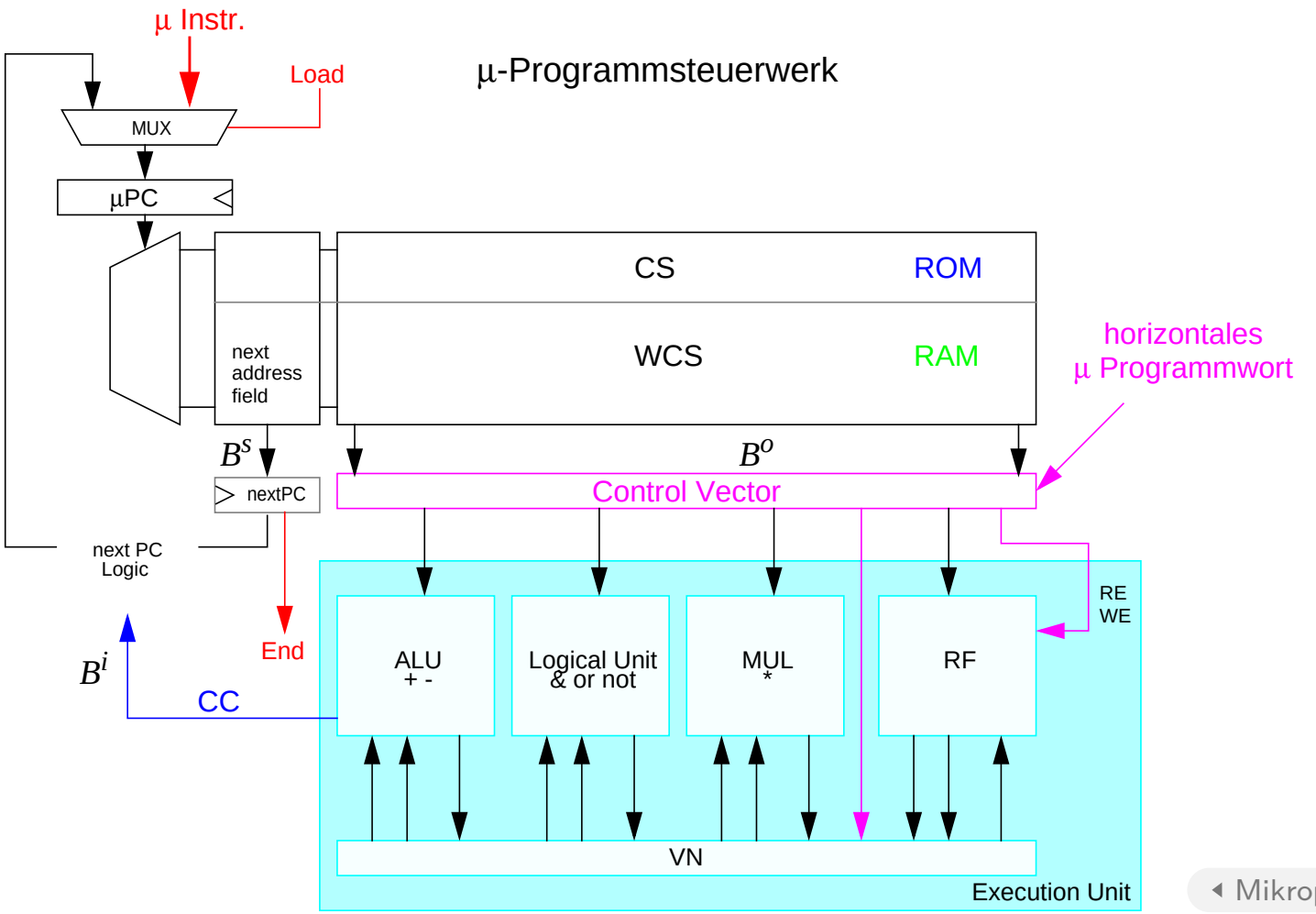
\+ CISC-Befehlssatz mit wenigen Mikrobefehlen realisieren

\+ bei RAM: Mikrobefehlssatz austauschbar

– (mehrstufige) ROM/RAM Zugriffe: zeitaufwändig

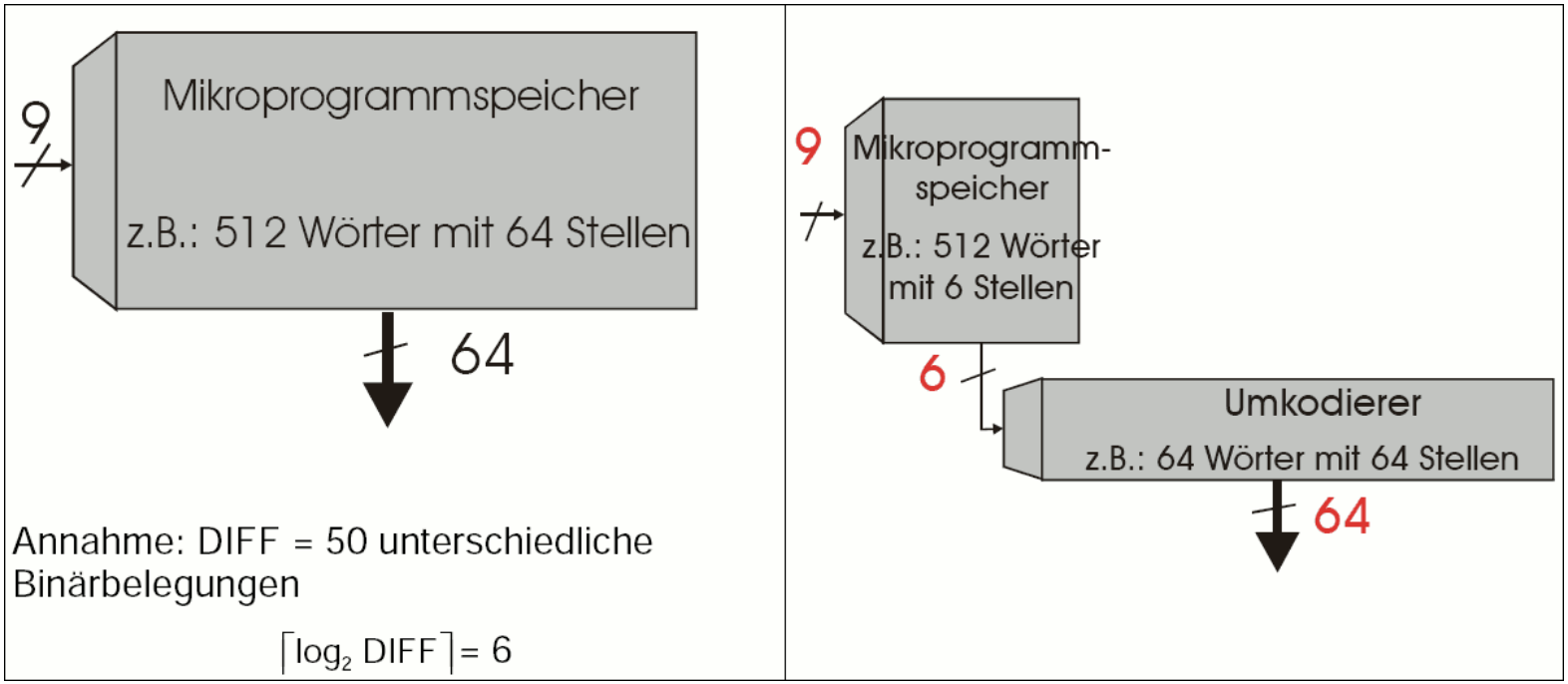
▶ ▸ horizontale Mikroprog. ▸ vertikale Mikroprog.

horizontale Mikroprogrammierung



◂ Mikroprogrammierung

vertikale Mikroprogrammierung



◂ Mikroprogrammierung

RISC: Reduced Instruction Set Computer

oder auch: „regular instruction set computer“

- einfache CPU mit Pipelining statt komplexen Befehlen
- in den 80er Jahren: „RISC-Boom“
  - internes Projekt bei IBM
  - von Hennessy (Stanford) und Patterson (Berkeley) publiziert
- Hochsprachen und optimierende Compiler

  $\Rightarrow$ kein Bedarf mehr für mächtige Assemblerbefehle

  $\Rightarrow$ pro Assemblerbefehl muss nicht mehr „möglichst viel“ lokal in der CPU gerechnet werden (CISC Mikroprogramm)

Beispiele

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

RISC: Reduced Instruction Set Computer (cont.)

typische Merkmale

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

RISC: Reduced Instruction Set Computer (cont.)

- Register-orientierter Befehlssatz
  - viele universelle Register, keine Spezialregister (≥ 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
- optimierende Compiler statt Assemblerprogrammierung

RISC: Reduced Instruction Set Computer (cont.)

Vor- / Nachteile

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

CISC vs. RISC

ursprüngliche Debatte

- 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

ISA Design heute

- 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“)

  $\Rightarrow$ neuer Befehlssatz mit expliziter Parallelität (EPIC)

  $\Rightarrow$ 64-bit Wortgrößen (überwinden Adressraumlimits)

  $\Rightarrow$ benötigt hoch entwickelte Compiler

Literatur

[BO14] R.E. Bryant, D.R. O'Hallaron:
Computer systems – A programmers perspective.
2nd new intl. ed., Pearson Education Ltd., 2014.
ISBN 978–1–292–02584–1. `csapp.cs.cmu.edu`

[TA14] A.S. Tanenbaum, T. Austin: *Rechnerarchitektur – Von der digitalen Logik zum Parallelrechner*.
6. Auflage, Pearson Deutschland GmbH, 2014.
ISBN 978–3–86894–238–5

Literatur (cont.)

[PH14] D.A. Patterson, J.L. Hennessy: *Computer Organization and Design – The Hardware/Software Interface*.
5th edition, Morgan Kaufmann Publishers Inc., 2014.
ISBN 978–0–12–407726–3

[PH11] D.A. Patterson, J.L. Hennessy: *Rechnerorganisation und -entwurf – Die Hardware/Software-Schnittstelle*.
4. Auflage, Oldenbourg, 2011. ISBN 978–3–486–59190–3

[HP12] J.L. Hennessy, D.A. Patterson:
Computer architecture – A quantitative approach.
5th edition, Morgan Kaufmann Publishers Inc., 2012.
ISBN 978–0–12–383872–8

Literatur (cont.)

[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`

[IA64] *Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 1: Basic Architecture*.
Intel Corp.; Santa Clara, CA.
`www.intel.de/content/www/de/de/processors/architectures-software-developer-manuals.html`