Rechnerstrukturen und Betriebssysteme

64-040 Vorlesung RSB
Kapitel 12: Rechnerarchitektur (2)
Instruction Set Architecture

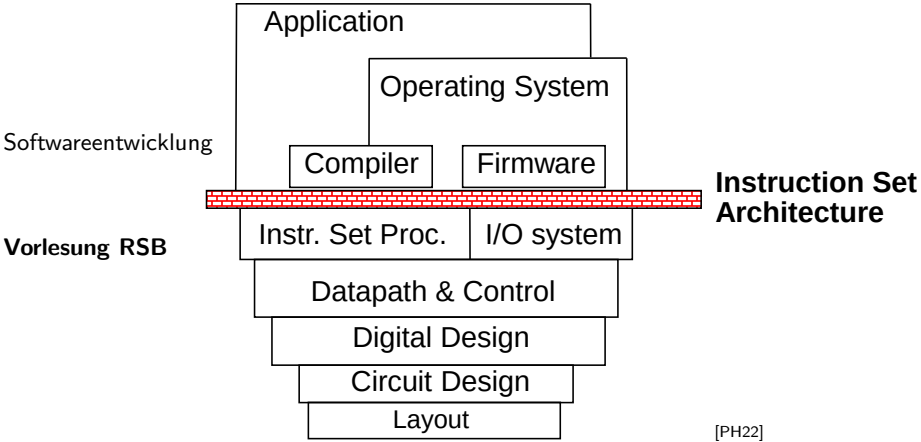
Norman Hendrich

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

Wintersemester 2025/2026

Gliederung

Rechnerstrukturen und Betriebssysteme – Einordnung



[PH22]

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.*

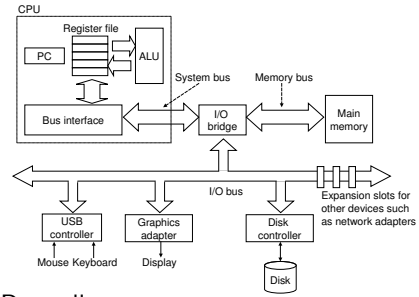
   [Amdahl, Blaauw, Brooks]

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. 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.*

Befehlssatzarchitektur – ISA

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

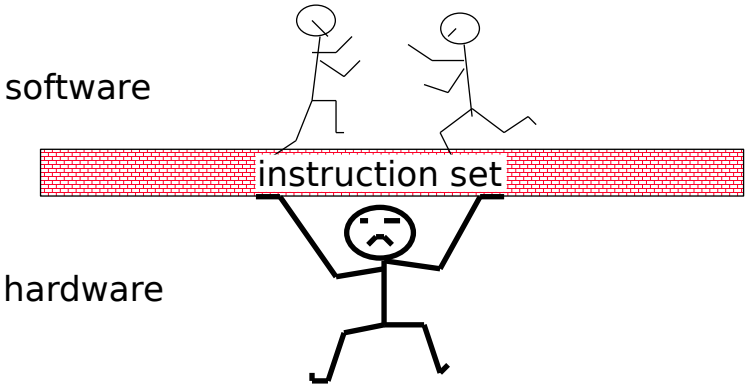
⇒ alle für den Programmierer sichtbaren Attribute einer Rechnerarchitektur



- 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

Befehlssatzarchitektur – ISA

- Befehlssatz: die zentrale Schnittstelle



[PH22]

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

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
- 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

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

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

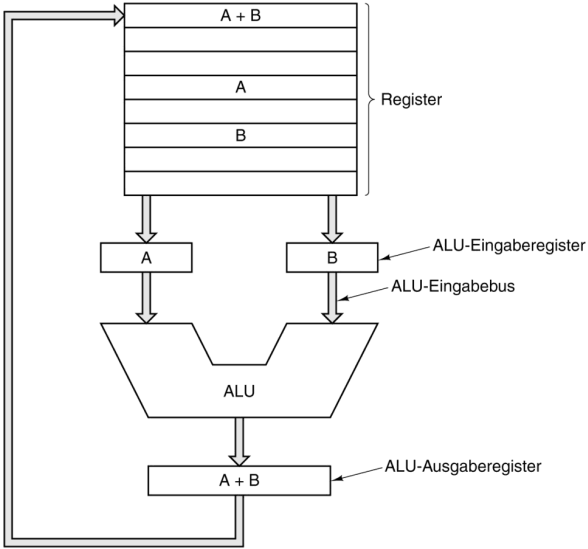
- Rechner soll „rechnen“ können
- typische arithmetische Operation nutzt 3 Variablen
  Resultat, zwei Operanden: $X = Y + Z$

  `add r2, r4, r5` 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

Beispiel: Datenpfad

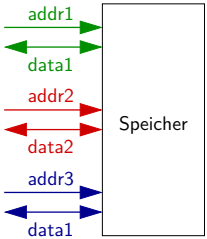
- Register (-bank)
  - liefern Operanden
  - speichern Resultate
- interne Hilfsregister
- ALU, typ. Funktionen:
  - add, add-carry, sub
  - and, or, xor
  - shift, rotate
  - compare
  - (floating point ops.)



[TA14]

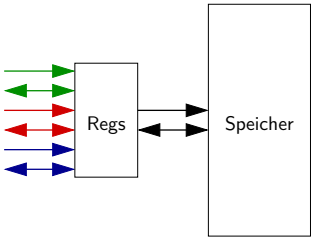
Woher kommen die Operanden?

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



⇒ „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



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: 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	TOS	NOS	Stack
`push E`	E		
`push D`	D	E	
`mul`	D*E		
`push C`	C	D*E	
`add`	C+D*E		
`push B`	B	C+D*E	
`push A`	A	B	C+D*E
`sub`	A-B	C+D*E	
`div`	(A-B)/(C+D*E)		
`pop Z`			

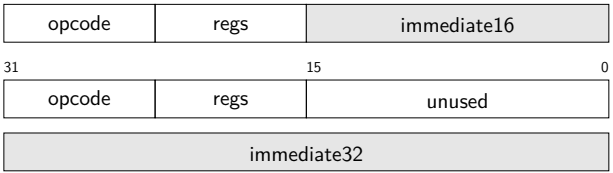
Adressierungsarten

- „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.)

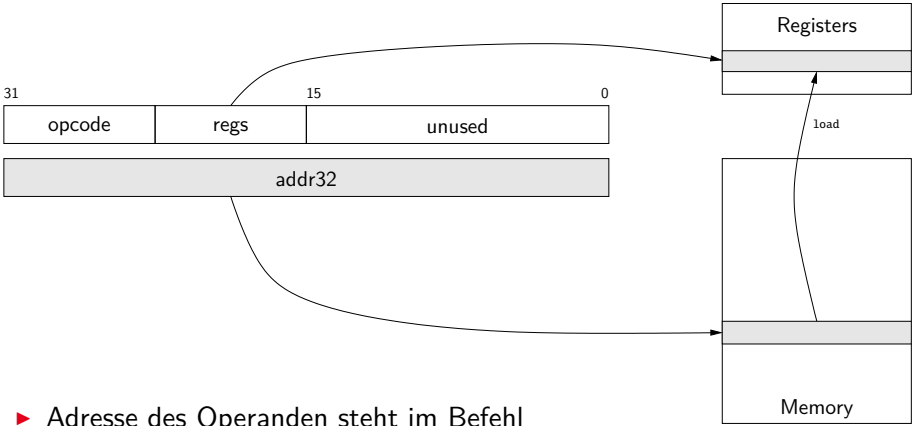
- „register“
  - wie Direktmodus, aber Register statt Speicher
  - 32 Register: benötigen 5 bit im Befehl
  - genug Platz für 2- oder 3-Adress Formate
- „register-indirekt“
  - Befehl spezifiziert ein Register
  - mit der Speicheradresse des Operanden
  - ein zusätzlicher Speicherzugriff
- „indiziert“
  - 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



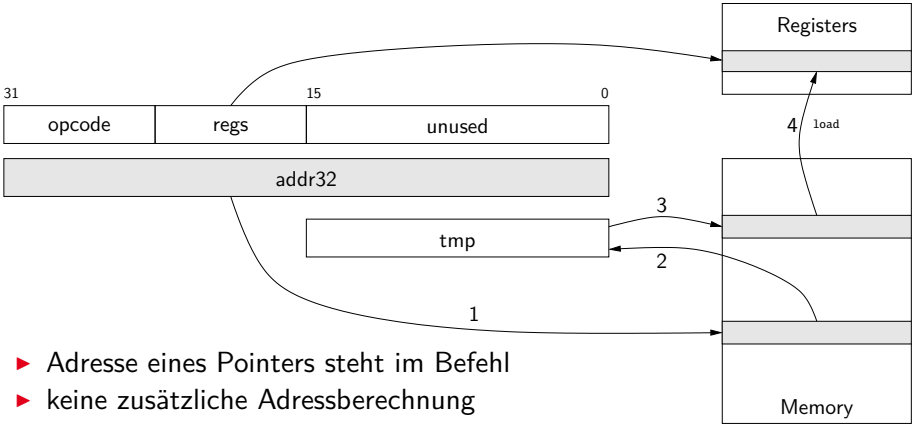
- Operand steht direkt im Befehl, kein zusätzlicher Speicherzugriff
- Länge des Operanden $<$ (Wortbreite - Opcodebreite)
- Darstellung größerer Zahlenwerte
  - 2-Wort Befehle (x86)
    zweites Wort für Immediate-Wert
  - mehrere Befehle (MIPS, SPARC)
    z.B. obere/untere Hälfte eines Wortes
  - Immediate-Werte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

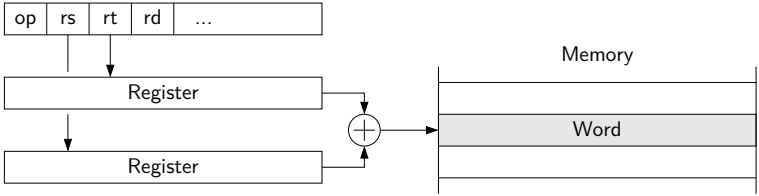
Indirekte Adressierung



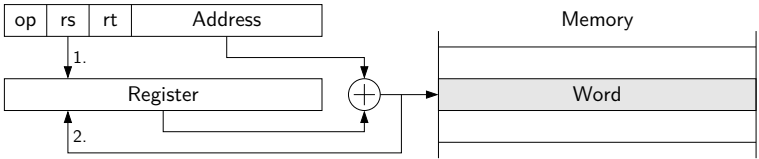
- Adresse eines Pointers steht im Befehl
- keine zusätzliche Adressberechnung
- zwei zusätzliche Speicherzugriffe:
  z.B. tmp = MEM[addr32] R3 = MEM[tmp]
- typische CISC-Adressierungsart, viele Taktzyklen
- kommt bei RISC-Rechnern nicht vor

Indizierte Adressierung

Indexaddressing

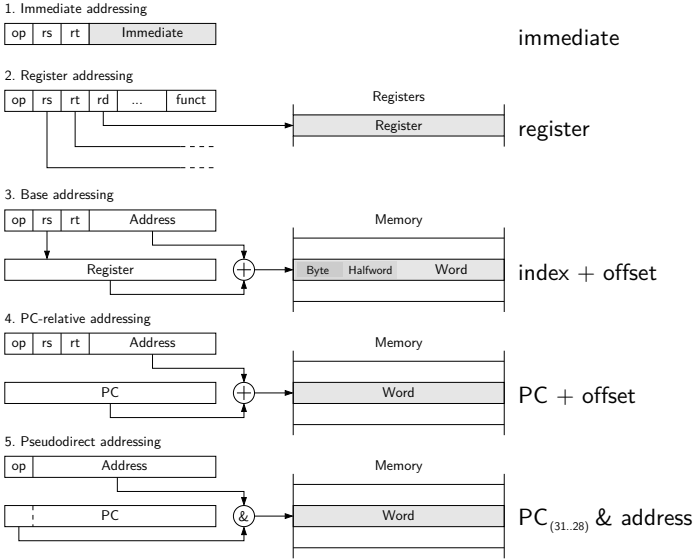


Updateaddressing



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

Beispiel: MIPS Adressierungsarten



typische Adressierungsarten

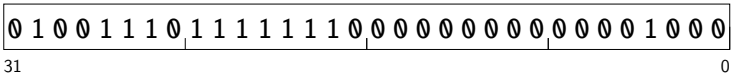
welche Adressierungsarten / -Varianten sind üblich?

- 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`

Befehls-Decodierung

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



Wie soll die Hardware diesen Wert interpretieren?

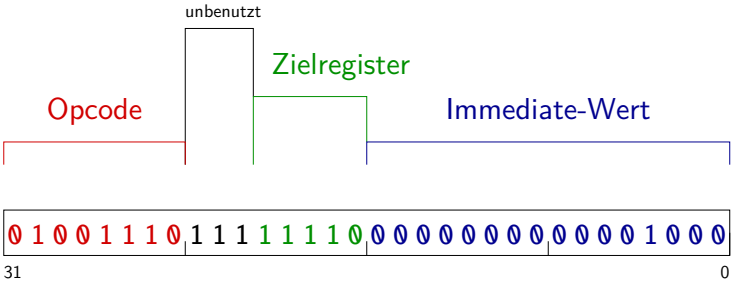
- direkt in einer Tabelle nachschauen (Mikrocode-ROM)
- Problem: Tabelle müsste 2^{32} Einträge haben

$\Rightarrow$ 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 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

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 (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

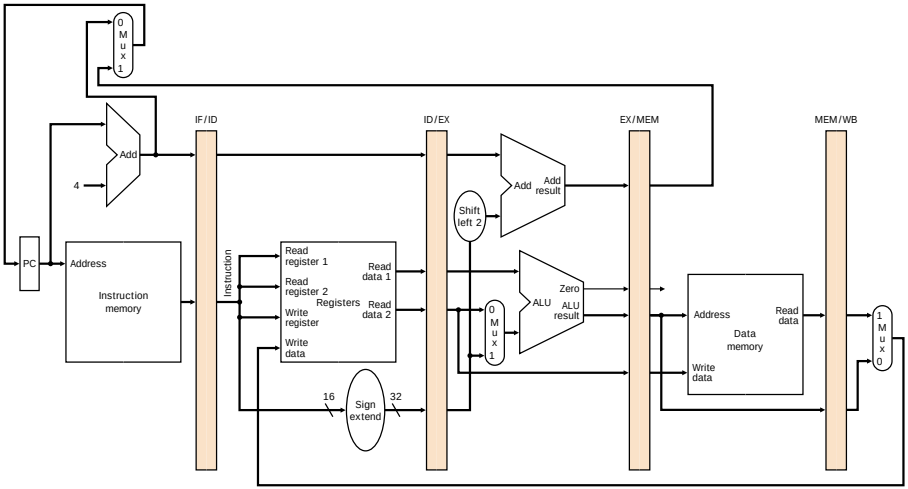
R1 = (R2 < R3) `slt R1, R2, R3`

MIPS: Befehlssatz

- 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



[PH22]

PC
I-Cache

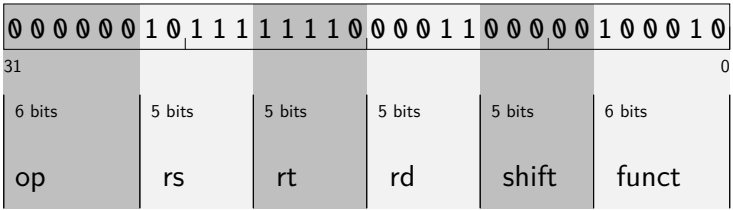
Register
(R0 . . . R31)

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)`

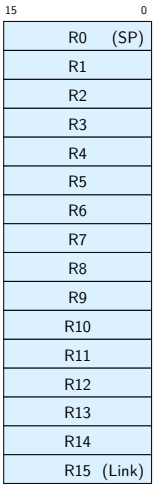
Befehlsformat: Beispiel M·CORE

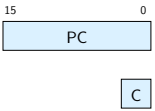
- 32-bit RISC Architektur, Motorola 1998
- besonders einfaches Programmiermodell
  - Program Counter PC
  - 16 Universalregister R0 . . . R15
  - Statusregister C („carry flag“)
  - 16-bit Befehle (um Programmspeicher zu sparen)
- Verwendung
  - Mikrocontroller für eingebettete Systeme, z.B. *„Smart Cards“*
  - siehe `en.wikipedia.org/wiki/M.CORE`

- ähnlich M·CORE
- gleiches Registermodell, aber nur 16-bit Wortbreite
  - Program Counter PC
  - 16 Universalregister R0 . . . R15
  - Statusregister C („carry flag“)
- Subset der Befehle, einfachere Codierung
- vollständiger Hardwareaufbau in Hades verfügbar
  - [Hen] Hades Demo: `60-dcore/t3/chapter`

  oder Simulator mit Assembler aus den Praktikumsunterlagen
  - 64-042: Rechnerstrukturen und Betriebssysteme (`t3asm`)

D·CORE: Registermodell





- 16 Universalregister
- Programmzähler
- 1 Carry-Flag

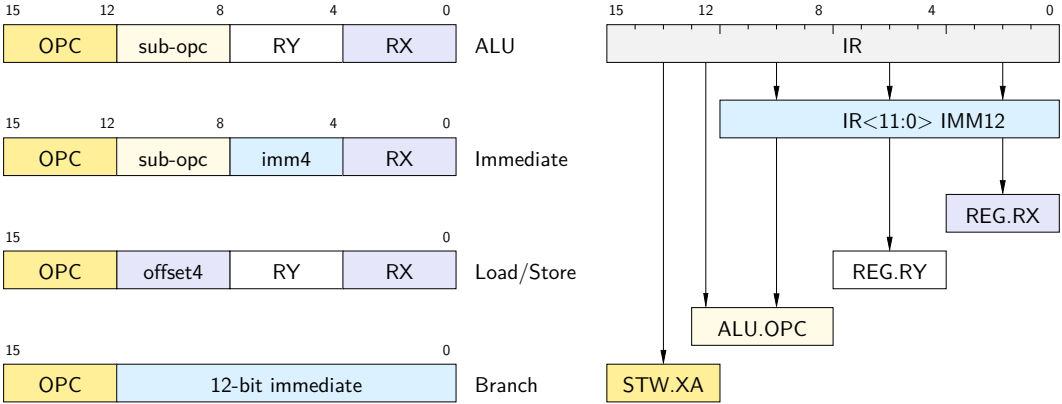
- Befehlsregister
- Bus-Interface

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

D·CORE: Befehlssatz

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

Intel x86-Architektur

- übliche Bezeichnung für die Intel-Prozessorfamilie
  8086, 80286, 80386, 80486, Pentium . . . Pentium 4, Core 2, Core-i, Core Ultra . . .
- eigentlich „IA-32“ (Intel architecture, 32-bit) . . . „IA-64“

- irreguläre Struktur: CISC
- historisch gewachsen: diverse Erweiterungen (MMX, SSE, AVX . . .)
- 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

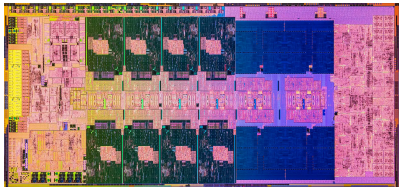
Intel x86: Evolution

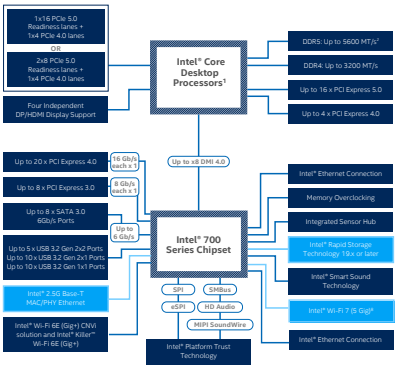
Chip	Datum	MHz	Transistoren	Speicher	Anmerkungen
4004	4/1971	0,108	2 300	640 B	erster Mikroprozessor auf einem Chip
8008	4/1972	0,108	3 500	16 KiB	erster 8-bit Mikroprozessor
8080	4/1974	2	6 000	64 KiB	„general-purpose“ CPU auf einem Chip
8086	6/1978	5–10	29 000	1 MiB	erste 16-bit CPU auf einem Chip
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 ... Befehlssatz: AVX ...
...	11/2021				Performance- + Efficiency-Cores ...

Beispiel: Core i9-14900K Prozessor

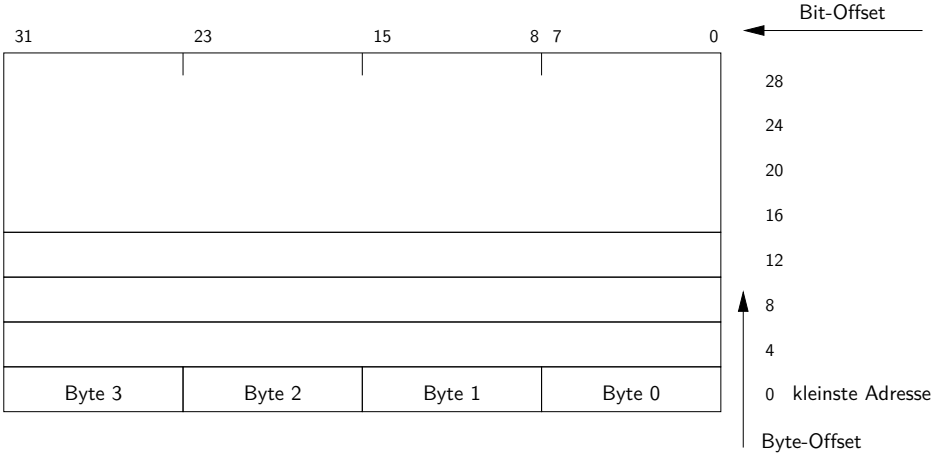
Performance Cores	8 ($\times$ 2 Hyperthreading)
Taktfrequenz	3,2 GHz (max. 6,0 GHz)
L1 Cache	8×32 KiB I + 48KiB D
L2 Cache	$8 \times 2{,}0$ MiB (I+D)
Efficiency Cores	16
Taktfrequenz	2,4 GHz (max. 4,4 GHz)
L1 Cache	16×32 KiB I + 64KiB D
L2 Cache	4×2 MiB (I+D)
L3 Cache	36 MiB (I+D)
Memory Controller	$2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

Quellen: `ark.intel.com`, `www.intel.de`
`en.wikichip.org`





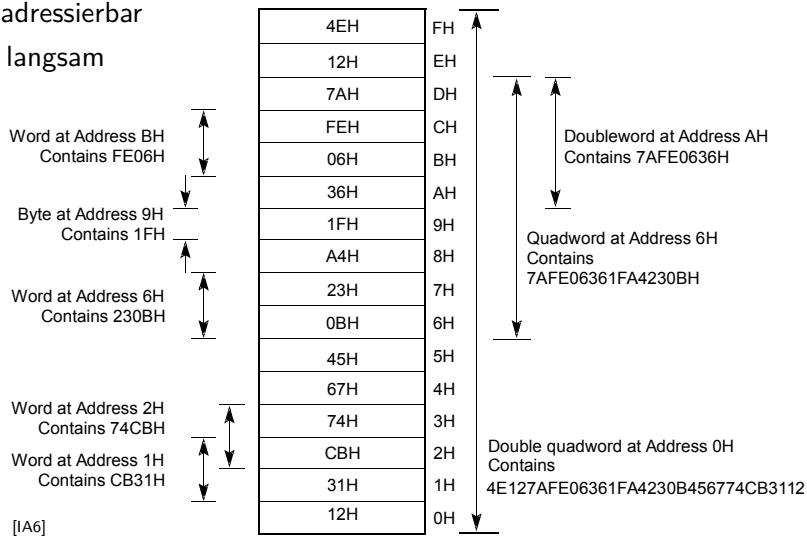
x86: Speichermodell



- „Little Endian“: LSB eines Wortes bei der kleinsten Adresse

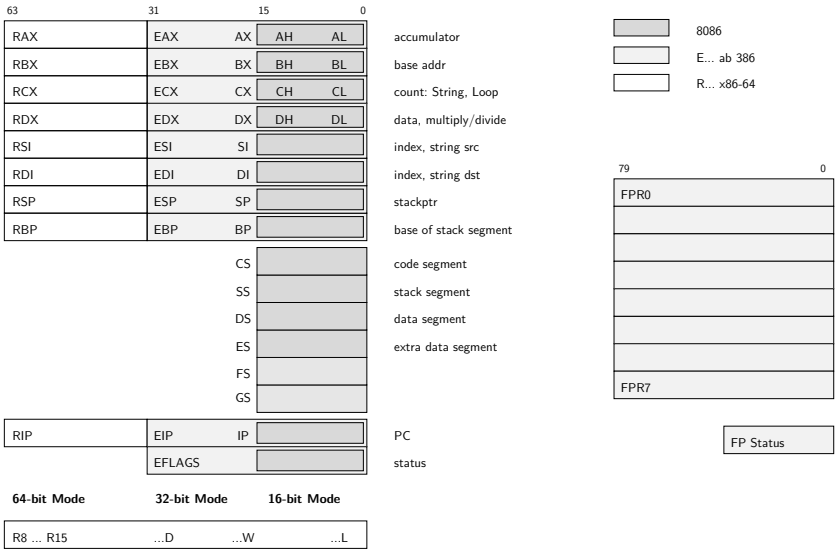
x86: Speichermodell (cont.)

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

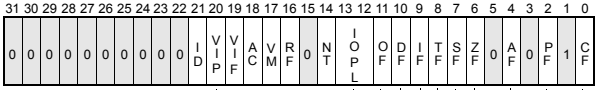


[IA6]

x86: Register



x86: EFLAGS Register



X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check / Access Control (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)

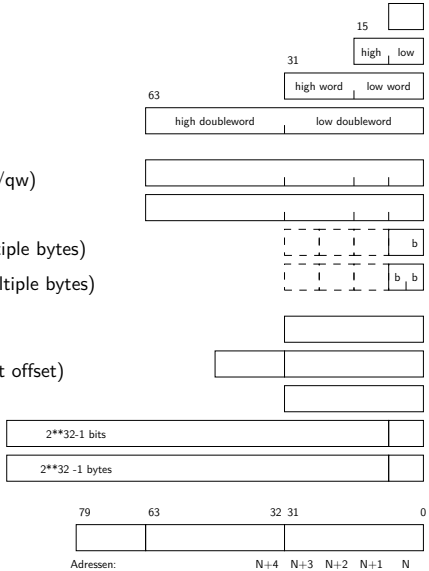
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag

Reserved bit positions. DO NOT USE.
Always set to values previously read.

[IA6]

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

Funktionalität	
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

$\Rightarrow$ CISC – **C**omplex **I**nstruction **S**et **C**omputer

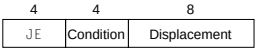
x86: Befehlsformat-Modifier („prefix“)

- alle Befehle können mit Modifiern ergänzt werden

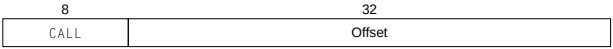
segment override	Adresse aus angewähltem Segmentregister
address size	Umschaltung 16/32/64-bit Adresse
operand size	Umschaltung 16/32/64-bit Operanden
repeat	Stringoperationen: für alle Elemente
lock	Speicherschutz bei Multiprozessorsystemen

x86 Befehlscodierung: Beispiele

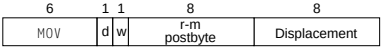
a. JE EIP + displacement



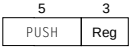
b. CALL



c. MOV EBX, [EDI + 45]

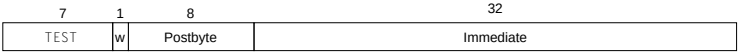


d. PUSH ESI



e. ADD EAX, #6765

f. TEST EDX, #42



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

[PH22]

x86 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 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_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.)

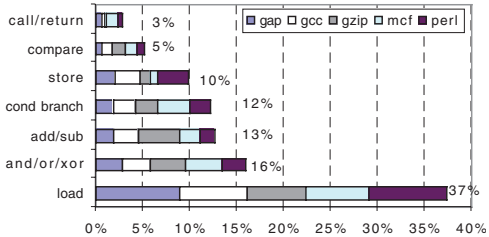
- MIPS-Prozessor

[HP17]

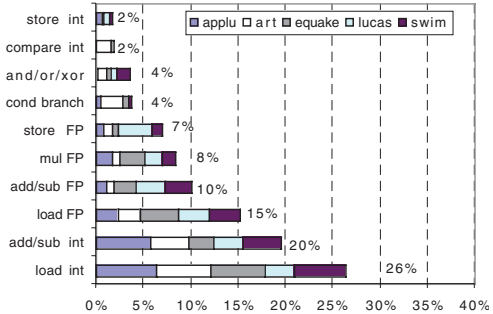
Instruction	compress	eqntott	espresso	gcc (cc1)	li	Int. average
load	20.8%	18.5%	21.9%	24.9%	23.3%	22%
store	13.8%	3.2%	8.3%	16.6%	18.7%	12%
add	10.3%	8.8%	8.15%	7.6%	6.1%	8%
sub	7.0%	10.6%	3.5%	2.9%	3.6%	5%
mul				0.1%		0%
div						0%
compare	8.2%	27.7%	15.3%	13.5%	7.7%	16%
mov reg-reg	7.9%	0.6%	5.0%	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.

Bewertung der ISA (cont.)



SPECint2000 (96%)



SPECfp2000 (97%)

- 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 ...

$\Rightarrow$ Motivation für RISC

CISC – **C**omplex **I**nstruction **S**et **C**omputer

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: $1 \ldots n$-Wort Befehle
  - komplexe Befehlscodierung
  - mehrere Schreib- und Lesezugriffe pro Befehl
- viele verschiedene Datentypen

CISC – **C**omplex **I**nstruction **S**et **C**omputer (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 – **C**omplex **I**nstruction **S**et **C**omputer (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

▸ horizontale Mikroprog.

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

CISC – Mikroprogrammierung (cont.)

2. vertikale Mikroprogrammierung

▸ vertikale Mikroprog.

   - kurze Mikroprogrammworte
   - Spalten enthalten Mikrooperationscode
   - mehrstufige Decodierung für Kontrollleitungen

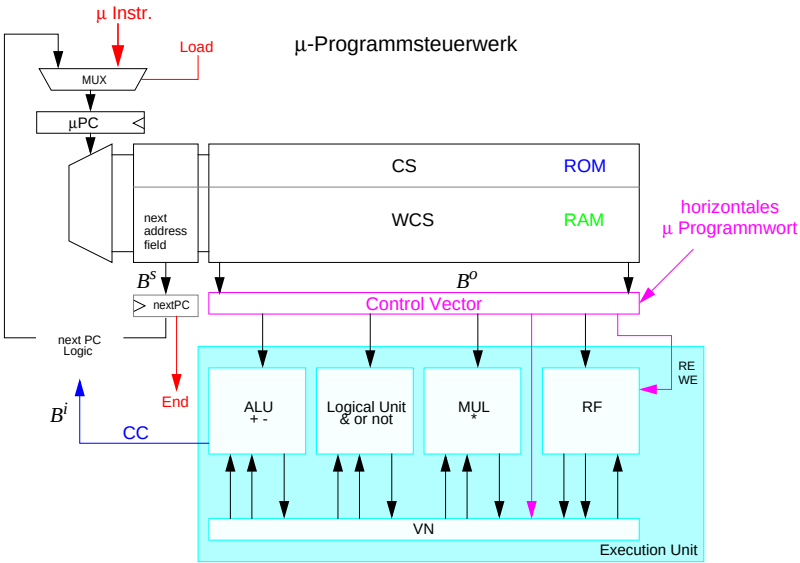
\+ CISC-Befehlssatz mit wenigen Mikrobefehlen realisieren

\+ μ-Programm im RAM: Mikrobefehlssatz austauschbar

\− (mehrstufige) ROM/RAM Zugriffe: zeitaufwändig

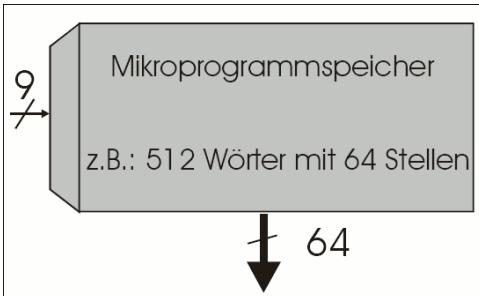
$\Rightarrow$ wird inzwischen nur noch benutzt, um CISC Befehle in RISC-artige Sequenzen umzusetzen (x86)

horizontale Mikroprogrammierung



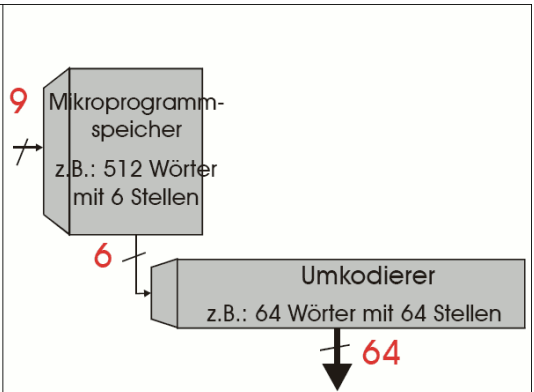
◂ Mikroprogrammierung

vertikale Mikroprogrammierung



Annahme: DIFF = 50 unterschiedliche Binärbelegungen

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



◂ Mikroprogrammierung

RISC – **R**educed **I**nstruction **S**et **C**omputer

oft auch: „**R**egular **I**nstruction **S**et **C**omputer“

- 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
  - $\Rightarrow$ statt Mikroprogramm in CISC: mehrere, einfache Assemblerbefehle
  - $\Rightarrow$ 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

RISC – **R**educed **I**nstruction **S**et **C**omputer (cont.)

- 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 (≥ 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

RISC – **R**educed **I**nstruction **S**et **C**omputer (cont.)

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

CISC vs. RISC

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

Literatur

[BO15] Randal E. Bryant and David R. O'Hallaron.
Computer systems – A programmers perspective.
Pearson Education Limited, Harlow, third, global ed. edition, 2015.

[Fur00] Steve Furber.
ARM System-on-Chip Architecture.
Pearson Education Limited, Harlow, second edition, 2000.

[Hen] Norman Hendrich.
Hades — hamburg design system.
Lehrmaterial, Universität Hamburg, Fachbereich Informatik, AB TAMS.

[HP17] John L. Hennessy and David A. Patterson.
Computer architecture – A quantitative approach.
Morgan Kaufmann Publishers Inc., San Mateo, CA, sixth edition, 2017.

Literatur (cont.)

[IA6] Intel Corp., Santa Clara, CA.
Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 1: Basic Architecture.

[PH20] David A. Patterson and John L. Hennessy.
Computer Organization and Design – The Hardware Software Interface – RISC-V Edition.
Morgan Kaufmann Publishers Inc., San Mateo, CA, second edition, 2020.

[PH22] David A. Patterson and John L. Hennessy.
Rechnerorganisation und Rechnerentwurf – Die Hardware/Software-Schnittstelle – MIPS Edition.
De Gruyter Oldenbourg, Berlin, sixth edition, 2022.

Literatur (cont.)

[TA14] Andrew S. Tanenbaum and Todd Austin.
Rechnerarchitektur – Von der digitalen Logik zum Parallelrechner.
Pearson Deutschland GmbH, Hallbergmoos, sixth edition, 2014.