Vorlesung: Rechnerstrukturen, Teil 2 (Modul IP7)

J. Zhang

zhang@informatik.uni-hamburg.de

Universität Hamburg

Fachbereich Informatik

AB Technische Aspekte Multimodaler Systeme

Inhaltsverzeichnis

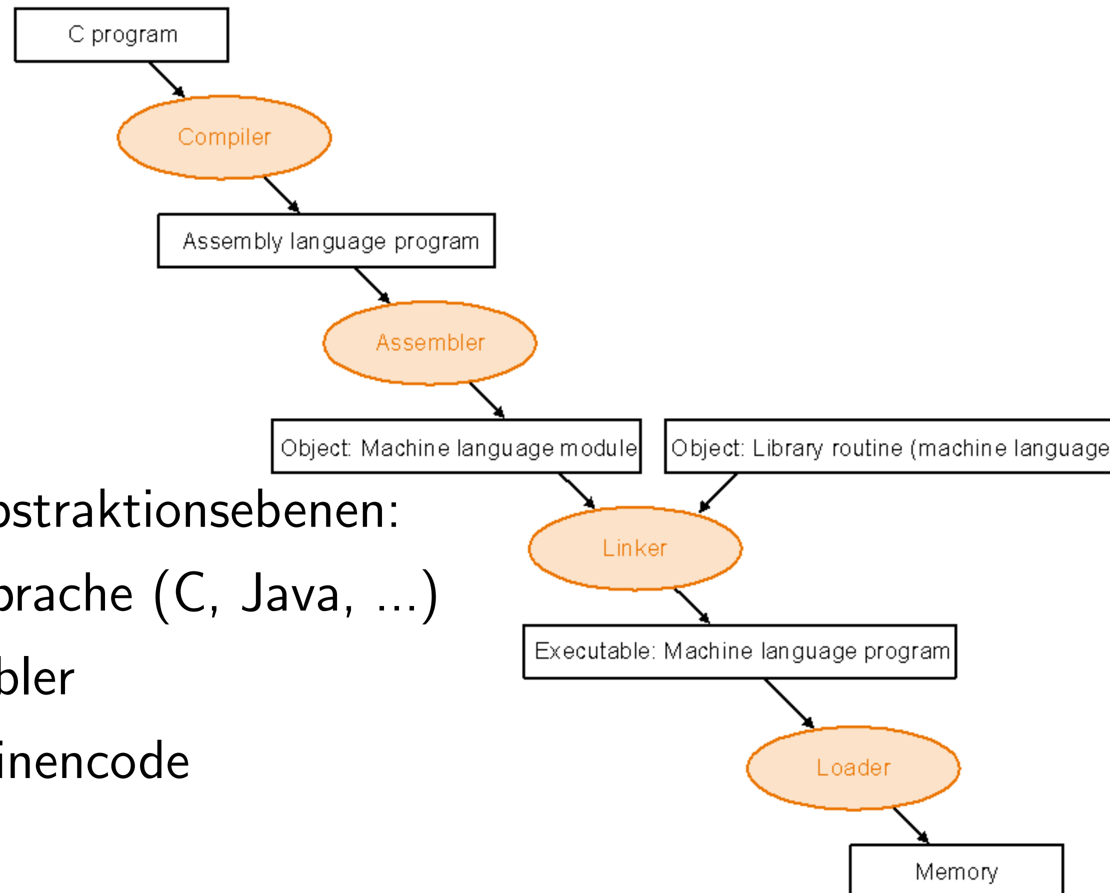
0. Wiederholung: Software-Schichten 40

3. Instruction Set Architecture 42

- Speicherorganisation 44
- Befehlszyklus: 54
- Befehlsformat: Einteilung in mehrere Felder 62
- Adressierungsarten 73

4. x86-Architektur 88

Wiederholung: Software-Schichten



mehrere Abstraktionsebenen:

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

Artenvielfalt

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

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

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

=> Echtzeit–, Sicherheits–, Zuverlässigkeitsanforderungen

Instruction Set Architecture

ISA = „instruction set architecture“
= HW/SW-Schnittstelle einer Prozessorfamilie

charakteristische Merkmale:

- Rechnerklasse (Stack-/Akku-/Registermaschine)
- Registersatz (Anzahl und Art der Rechenregister)
- Speichermodell (Wortbreite, Adressierung, ...)

- Befehlssatz (Definition aller Befehle)
- Art, Zahl der Operanden (Anzahl/Wortbreite/Reg./Speicher)
- Ausrichtung der Daten (Alignment/Endianness)

- I/O-, Unterbrechungsstruktur (Ein- und Ausgabe, Interrupts)
- Systemsoftware (Loader/Assembler/Compiler/Debugger)

Instruction Set Architecture Fortsetzung

Beispiele: (in dieser Vorlesung bzw. im Praktikum)

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

Speicherorganisation

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

spätere Themen:

- ◦ Cache-Organisation für schnelleren Zugriff
- ◦ Virtueller Speicher für Multitasking
- ◦ MESI-Protokoll für Multiprozessorsysteme
- ◦ Synchronisation in Multiprozessorsystemen

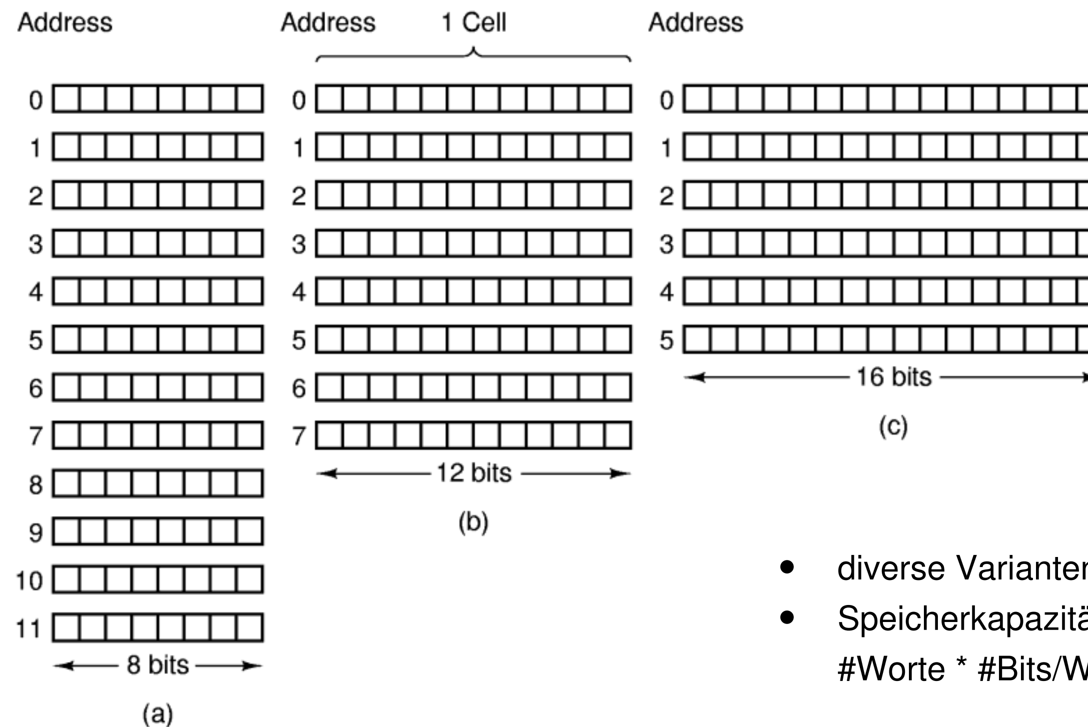
Speicher: Wortbreite

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

- Wortbreiten einiger historisch wichtiger Computer

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

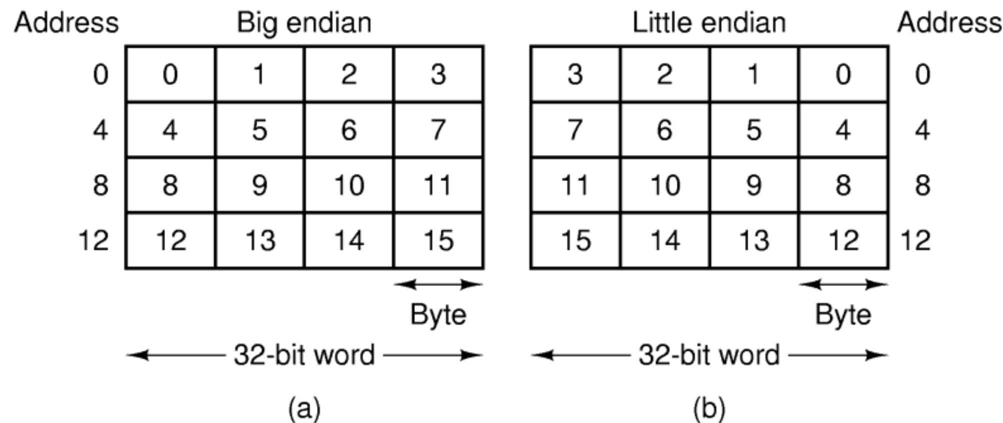
Hauptspeicher: Organisation



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

Drei Möglichkeiten, einen 96–Bit Speicher zu organisieren

Big- vs. Little Endian



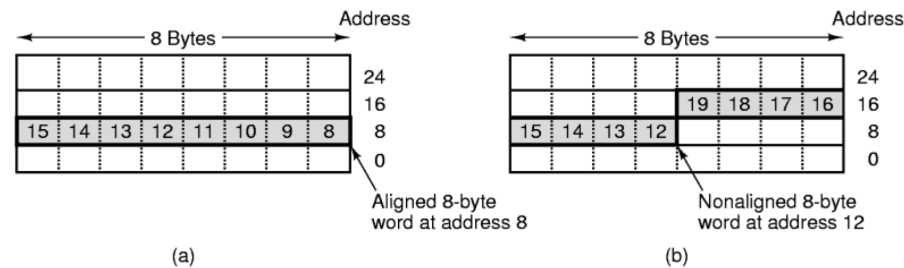
(a) Big endian memory. (b) Little endian memory.

Anordnung einzelner Bytes in einem Wort (hier 32-bit)

- big-endian LSB ... MSB Anordnung, gut für Strings
- little-endian MSB ... LSB Anordnung, gut für Zahlen

- beide Varianten haben Vor- und Nachteile
- komplizierte Umrechnung

Misaligned Access



Ein 8–Byte Word in einem little–endian Speicher.
(a) Aligned, (b) not aligned. Bei manchen Maschinen müssen die Wörter im Speicher aligned werden.

- Speicher wird (meistens) Byte-weise adressiert
- aber Zugriffe lesen/schreiben jeweils ein ganzes Wort

was passiert bei „krummen“ (misaligned) Adressen?

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

„Memory Map“

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

⇒ „Memory Map“

- Zuordnung von Adressen zu „realen“ Speichern
- Aufgabe des Adress-„Dekoders“
- Beispiel: Windows

Memory Map: typ. 16-bit System

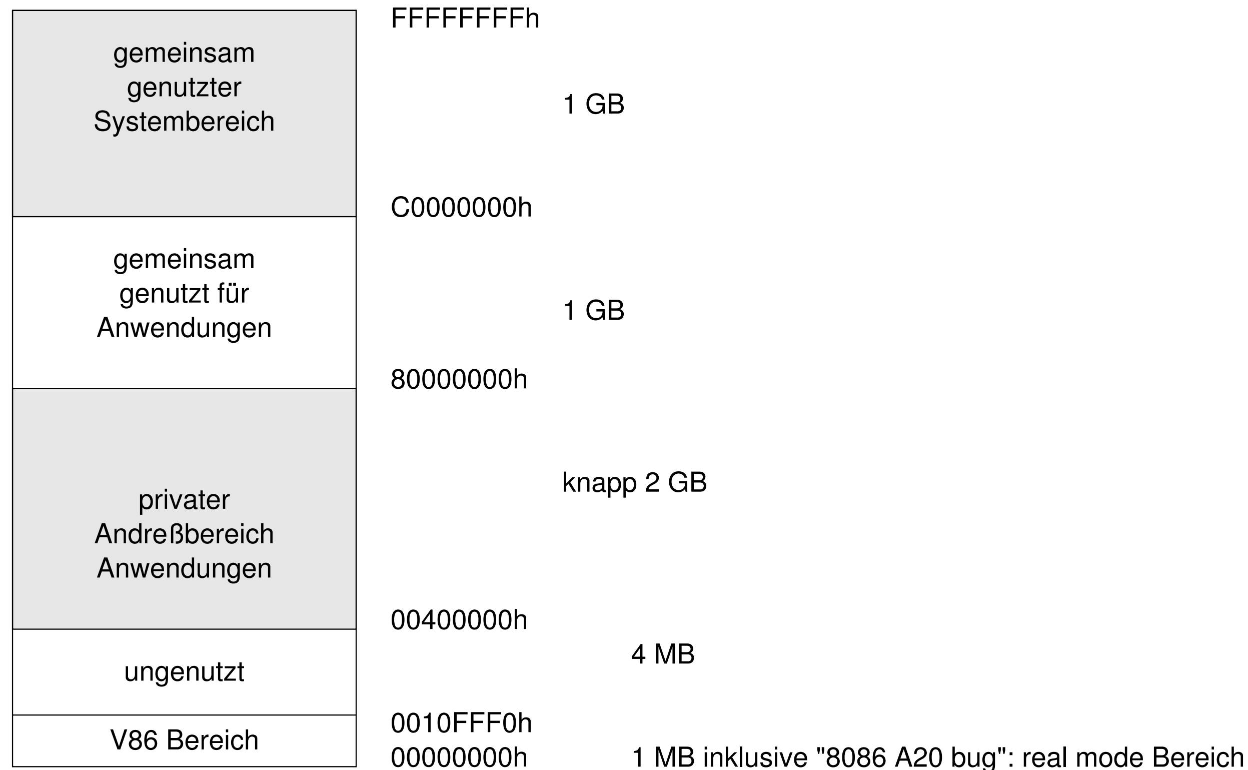
- 16-bit erlaubt 64K Adressen: 0x0000 .. 0xFFFF

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

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

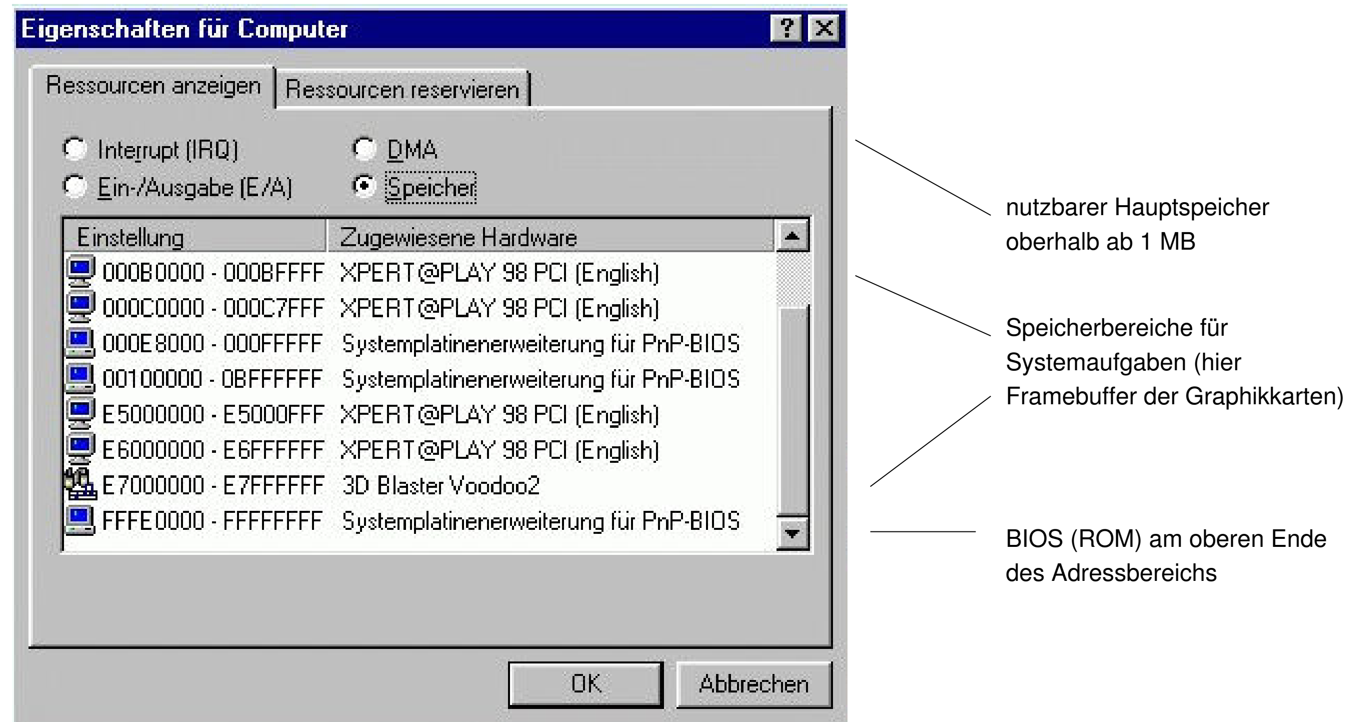
- Demo und Beispiele später

PC: Windows 9x Speicherbereiche



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

PC: Speicherbereiche, Beispiel



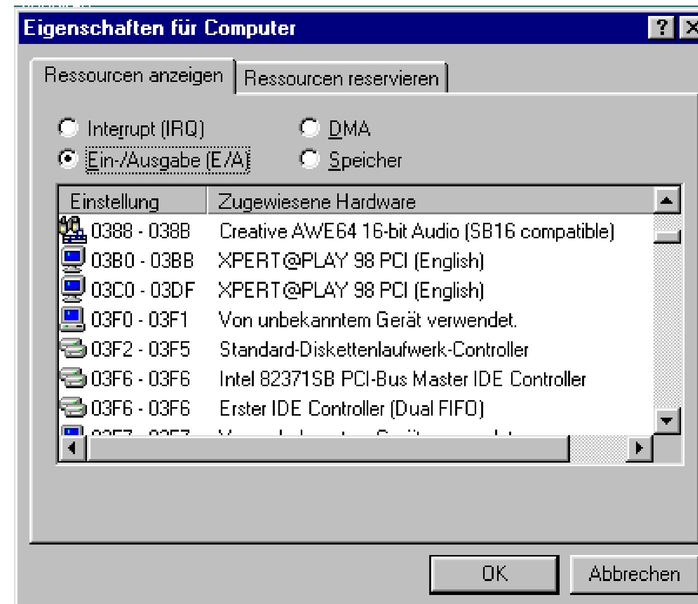
nutzbarer Hauptspeicher oberhalb ab 1 MB

Speicherbereiche für Systemaufgaben (hier Framebuffer der Graphikkarten)

BIOS (ROM) am oberen Ende des Adressbereichs

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

PC: IO-Adressen, Beispiel



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

Befehlszyklus:

- von-Neumann Prinzip: Daten und Befehle im Hauptspeicher

Befehlszyklus: (in Endlosschleife):

- Programmzähler PC addressiert den Speicher
- gelesener Wert kommt in das Befehlsregister IR
- Befehl dekodieren
- Befehl ausführen
- nächsten Befehl auswählen

⇒ man braucht mindestens diese Register:

PC	program counter	Adresse des Befehls
IR	instruction register	aktueller Befehl
ACC	accumulator	ein oder mehrere
R0..R31	registerbank	Rechenregister (Operanden)

Beispiel: Boot-Prozess

Was passiert beim Einschalten des Rechners?

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

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

Instruction Fetch

„Befehl holen“ Phase im Befehlszyklus:

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

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

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

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

Instruction Execute

- Befehl steht im Befehlsregister IR
- Decoder hat Opcode und Operanden entschlüsselt

- Ausführung des Befehls durch Ansteuerung

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

- Realisierung über festverdrahtete Hardware
- oder mikroprogrammiert

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

Welche Befehle braucht man?

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

CISC

„Complex instruction set computer“:

Bezeichnung für (ältere) Computer-Architekturen
mit irregulärem, komplexem Befehlssatz
typische Merkmale:

- sehr viele Befehle, viele Datentypen
- komplexe Befehlskodierung, Befehle variabler Länge
- viele Adressierungsarten
- Mischung von Register- und Speicheroperanden
- komplexe Befehle mit langer Ausführungszeit
- Problem: Compiler benutzen solche Befehle gar nicht

- Beispiele: Intel 80x86, Motorola 68K, DEC Vax

RISC

„reduced instruction set computer“
„regular instruction set computer“

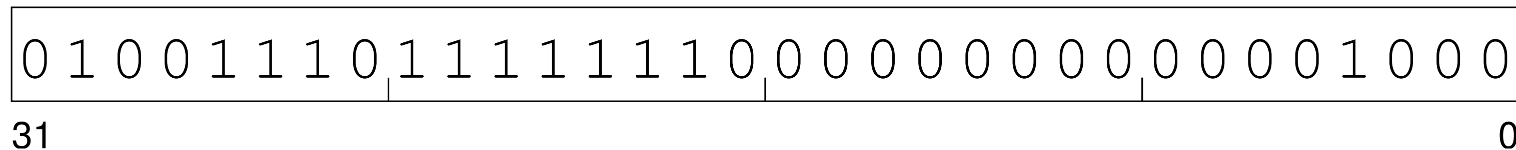
Oberbegriff für moderne Rechnerarchitekturen
entwickelt ab ca. 1980 bei IBM, Stanford, Berkeley

- reguläre Struktur, z.B. 32-bit Wortbreite, 32-bit Befehle
- Load-Store Architektur, keine Speicheroperanden
- viele Register, keine Spezialaufgaben
- optimierende Compiler statt Assemblerprogrammierung

- IBM 801, MIPS, SPARC, DEC Alpha, ARM
- Diskussion und Details CISC vs. RISC später

Befehls-Dekodierung

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

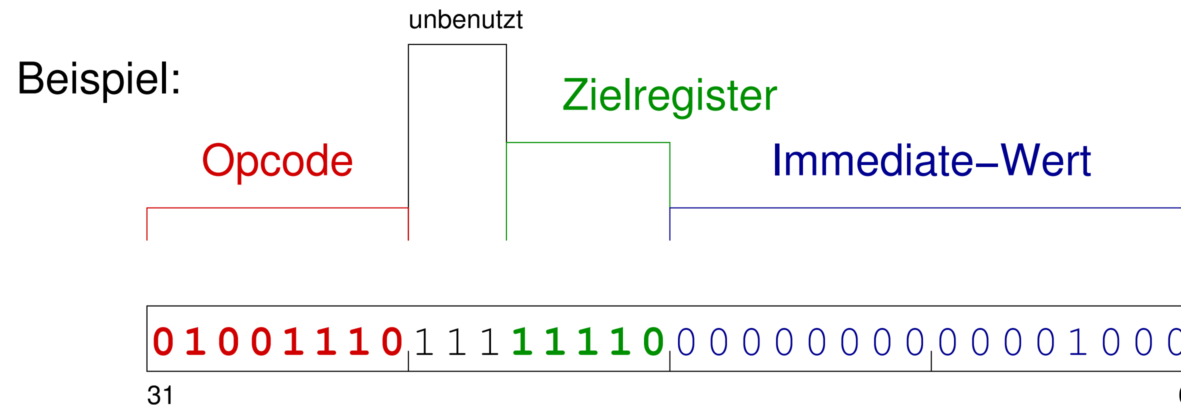


Wie soll die Hardware diesen Wert interpretieren?

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

⇒ „Befehlsformate“

Befehlsformat: Einteilung in mehrere Felder

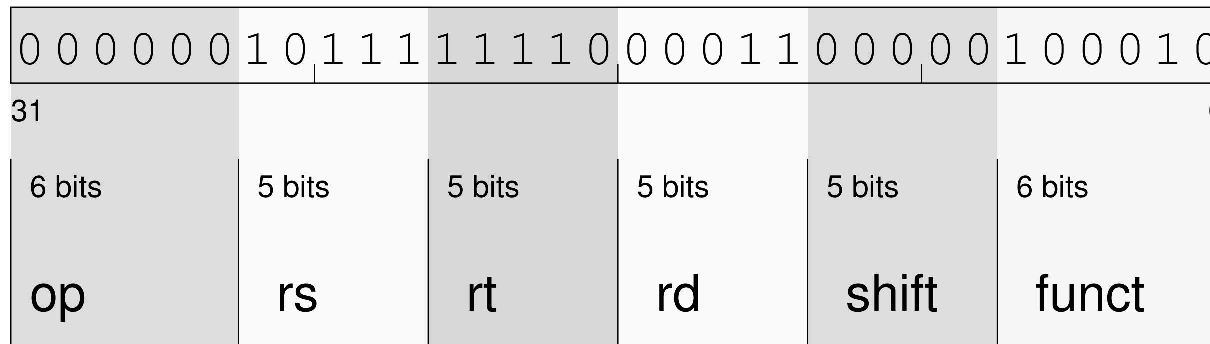


Befehls„format“: Aufteilung in mehrere Felder:

- Opcode — eigentlicher Befehl
- ALU-Operation — add/sub/incr/shift/usw.
- Register-Indizes — Operanden / Resultat
- Speicher-Adressen — für Speicherzugriffe
- Immediate-Operanden — Werte direkt im Befehl

- Lage und Anzahl der Felder abhängig vom jeweiligen Befehlssatz

Befehlsformat: Beispiel MIPS

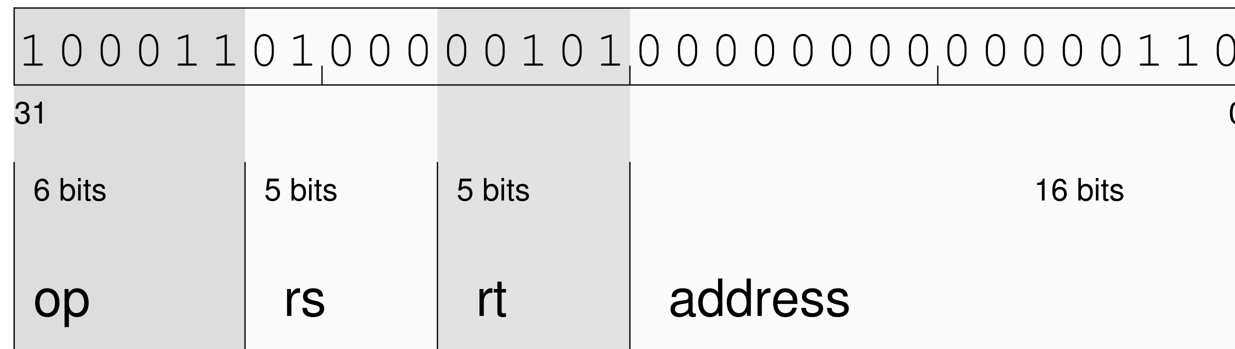


R–Format

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

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

Befehlsformat: Beispiel MIPS



I–Format

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

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

MIPS

- „Microprocessor without interlocked pipeline stages“
- entwickelt an der Univ. Stanford, seit 1982
- Einsatz: eingebettete Systeme, SGI Workstations/Server

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

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

MIPS: Register

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

$$
\begin{array}{ll}
R5 = -R5 & subR5, R0, R5 \\
R4 = 0 & addR4, R0, R0 \\
R3 = 17 & addiR3, R0, 17 \\
if(R2 == 0) & bneR2, R0, label
\end{array}
$$

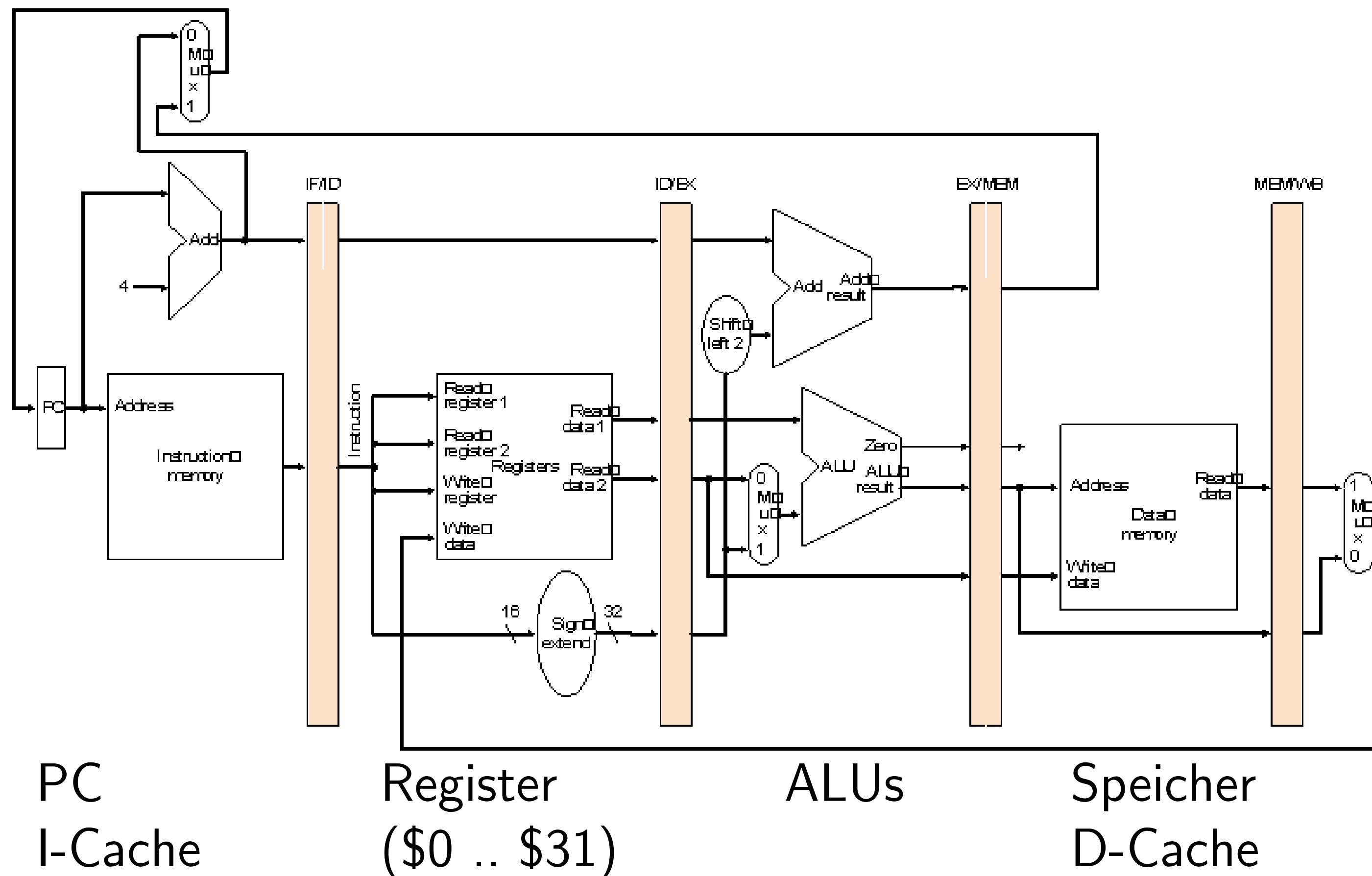
- keine separaten Statusflags
- Vergleichsoperationen setzen Zielregister auf 0 bzw. 1:

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

MIPS: Befehlssatz

- Übersicht und Details: siehe Hennessy & Patterson
- 3 Befehlsformate: ALU, ALU Immediate, Load/Store, Branch

MIPS: Hardwarestruktur



PC
I-Cache

Register
($0 .. $31)

ALUs

Speicher
D-Cache

M*CORE

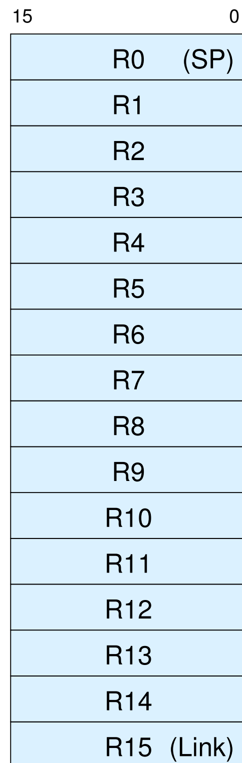
- 32-bit RISC Architektur, Motorola 1998
- besonders einfaches Programmiermodell:
  - Program Counter, PC
  - 16 Universalregister R0 .. R15
  - Statusregister C („carry flag“)

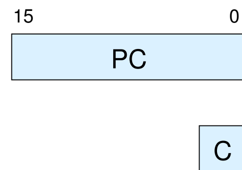
  - 16-bit Befehle (um Programmspeicher zu sparen)

D*CORE:

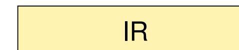
- gleiches Registermodell, aber nur 16-bit Wortbreite
- Subset der Befehle, einfachere Kodierung
- vollständiger Hardwareaufbau in Hades verfügbar
- oder Simulator mit Assembler (winT3asm.exe / t3asm.jar)

D*CORE: Register

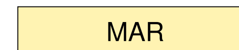


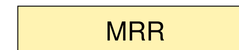


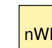
– 16 Universalregister
– Programmzähler
– 1 Carry–Flag

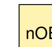


– Befehlsregister









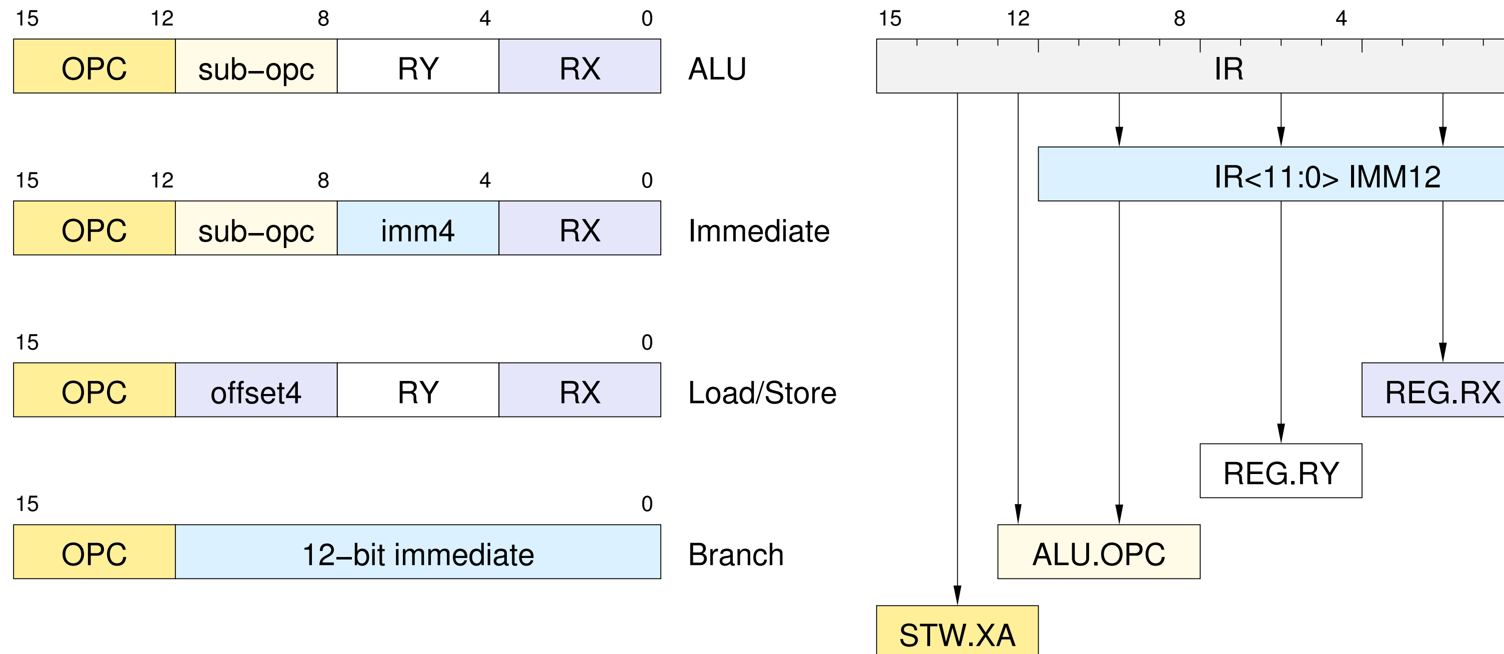
– Bus–Interface

- Programmierer sieht nur R0..R15, PC, C (carry flag)

D*CORE: Befehlssatz

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

D*CORE: Befehlsformate



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

Adressierungsarten

- Woher kommen die Operanden / Daten für die Befehle?
- Hauptspeicher, Universalregister, Spezialregister

- Wieviele Operanden pro Befehl?
- 0- / 1- / 2- / 3-Adress-Maschinen

- Wie werden die Operanden adressiert?
- immediate / direkt / indirekt / indiziert / autoinkrement / usw.

⇒ wichtige Unterscheidungsmerkmale für Rechnerarchitekturen

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

Beispiel: Add-Befehl

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

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

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

Datenpfad

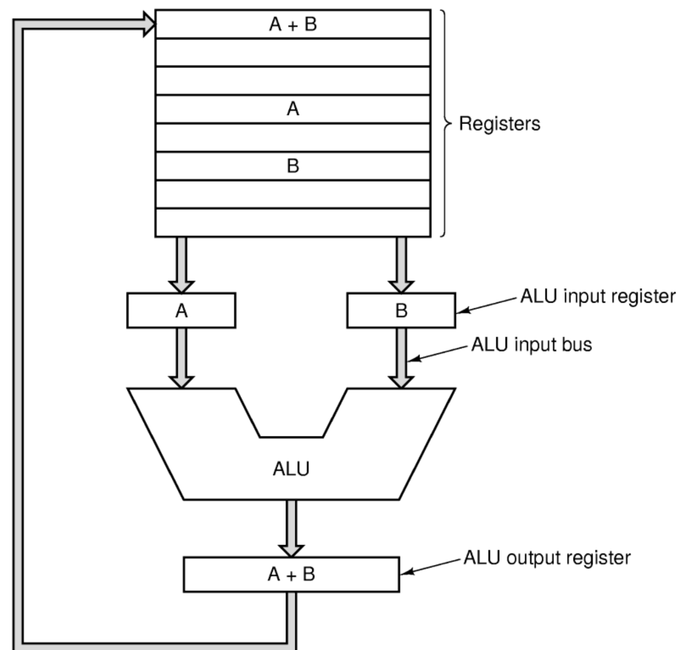


Figure 2-2. The data path of a typical von Neumann machine.

Register (-bank)

- liefern Operanden
- speichern Resultate

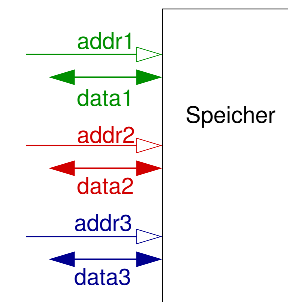
- interne Hilfsregister

ALU: typ. Funktionen:

- add, add-carry, sub
- and, or, xor
- shift, rotate
- compare
- (floating point ops.)

Woher kommen die Operanden?

- von-Neumann Prinzip: alle Daten im Hauptspeicher
- typ. Operation: zwei Operanden, ein Resultat
- „Multiport-Speicher“: mit drei Ports ?!
- sehr aufwendig , extrem teuer, trotzdem langsam

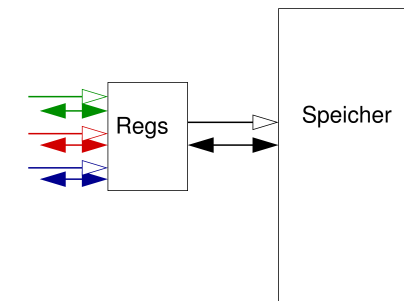


stattdessen:

- 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

3-Adress-Format	X = Y + Z
	sehr flexibel, leicht zu programmieren
	Befehl muss 3 Adressen speichern
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-Reg.
	häufig in älteren / 8-bit Rechnern
0-Adress-Format	TOS = TOS + NOS
	Stapelspeicher (top of stack, next of stack)
	Adressverwaltung entfällt
	im Compilerbau beliebt

n-Adress-Maschine

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

```
3-Adress-Maschine
sub Z, A, B
mul T, D, E
add T, T, C
div Z, Z, T
```

```
2-Adress-Maschine
mov   Z, A
sub   Z, B
mov   T, D
mul   T, E
add   T, C
div   Z, T
```

```
1-Adress-Maschine
load   D
mul    E
add    C
stor   Z
load   A
sub    B
div    Z
stor   Z
```

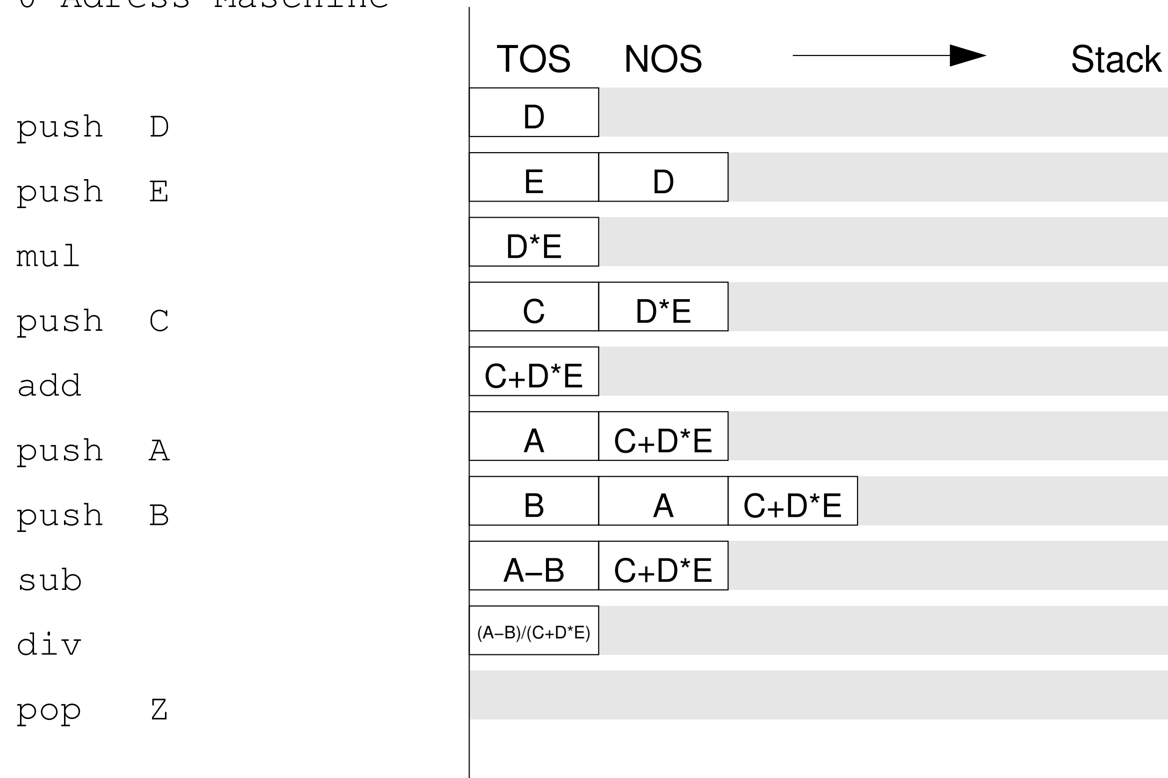
```
0-Adress-Maschine
push   D
push   E
mul
push   C
add
push   A
push   B
sub
div
pop    Z
```

Stack-Maschine

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

0–Adress–Maschine

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



Adressierungsarten (1)

„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)

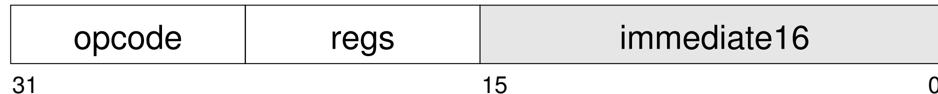
Adressierungsarten (2)

„register“ wie Direktmodus, aber Register statt Speicher
32 Register: benötigt 5 bit im Befehl
genug Platz für 2- oder 3-Adress Formate

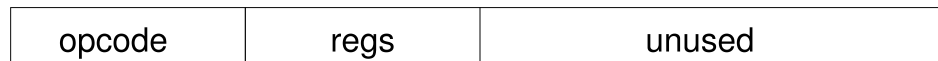
„register-indirekt“ Befehl spezifiziert ein Register
mit der Speicheradresse des Operanden
ein zusätzlicher Speicherzugriff

„indiziert“ Angabe mit Register und Offset
Inhalt des Registers liefert Basisadresse
Speicherzugriff auf (Basisaddresse+offset)
ideal für Array- und Objektzugriffe
Hauptmodus in RISC-Rechnern
(andere Bezeichnung: „Versatz-Modus“)

Immediate-Adressierung



1–Wort Befehl



2–Wort Befehl

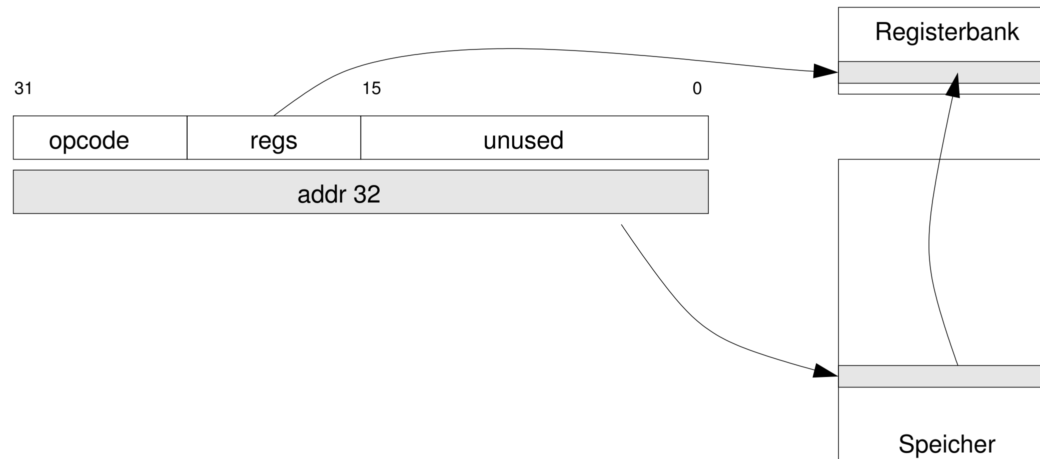
„immediate"

- Operand steht direkt im Befehl, kein zusätzlicher Speicherzugriff
- z.B. R3 = Immediate16
- Länge des Operanden ist kleiner als (Wortbreite - Opcodebreite)

  zur Darstellung grösserer Werte:

- 2-Wort Befehle (zweites Wort für Immediate-Wert) (x86)
- mehrere Befehle, z.B. obere/untere Hälfte eines Wortes (Mips,SPARC)
- Immediate-Werte mit zusätzlichem Shift (ARM)

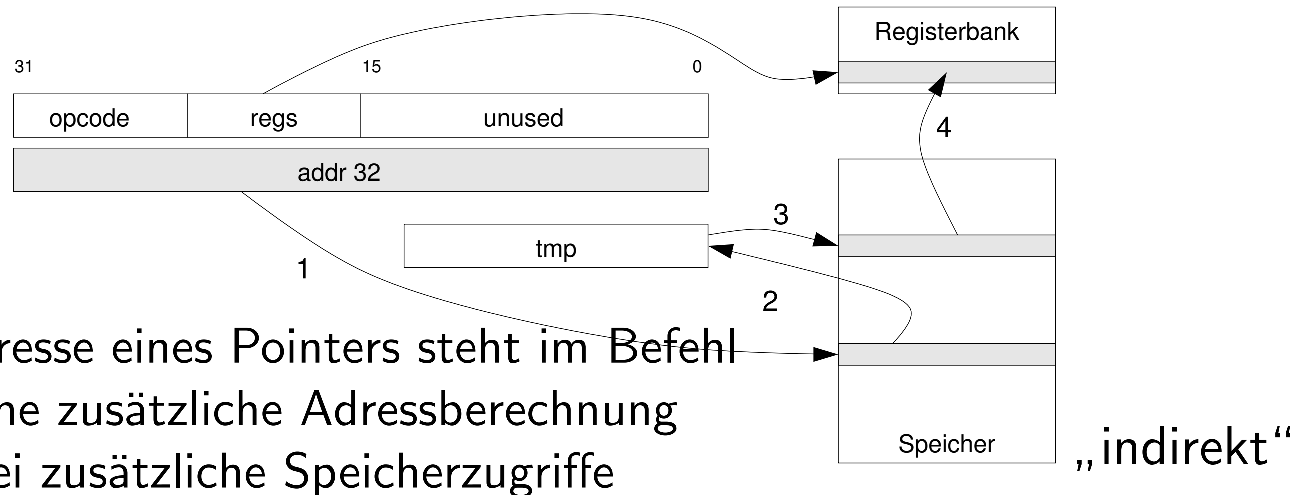
direkte Adressierung



„direkt“

- 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



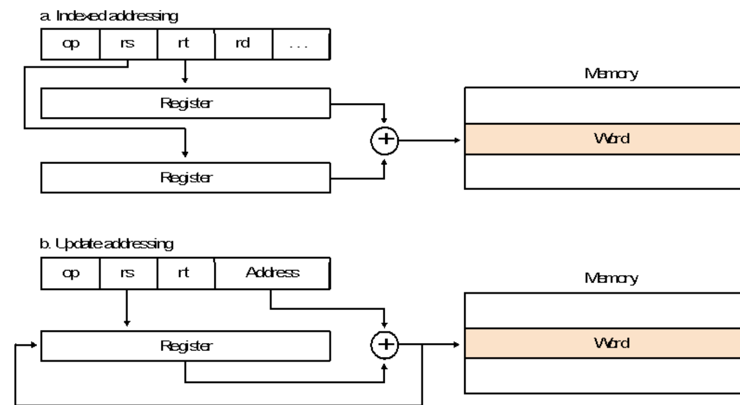
„indirekt“

- Adresse eines Pointers steht im Befehl
- keine zusätzliche Adressberechnung
- zwei zusätzliche Speicherzugriffe

z.B. tmp = MEM[addr32]; R3 = MEM[tmp]

- typische CISC-Adressierungsart, viele Taktzyklen
- kommt bei RISC-Rechnern nicht vor

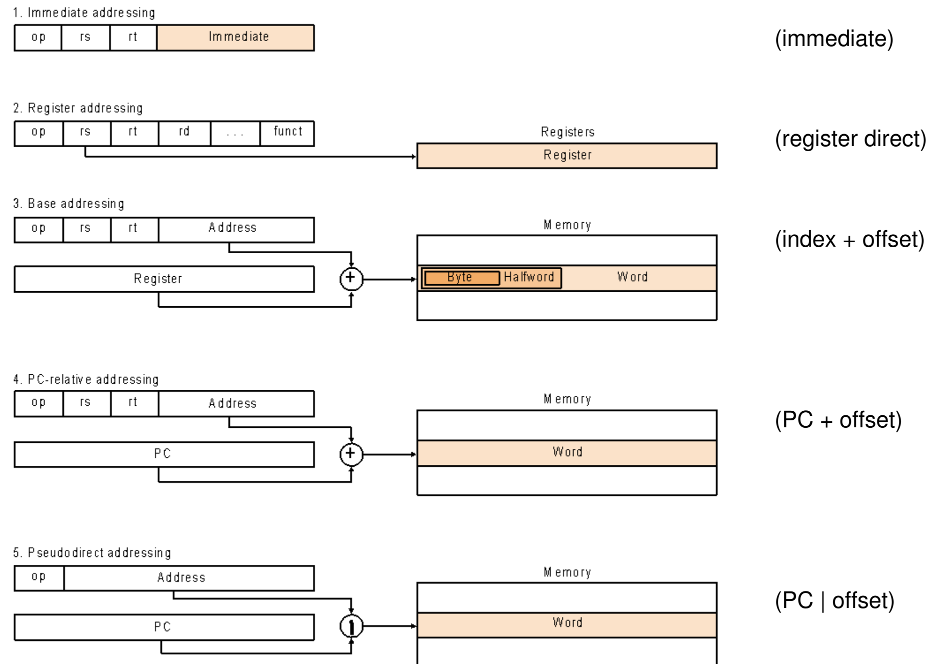
Indizierte Adressierung



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

- addr = (Sourceregister + offset)
  sourceregister = addr

Weitere Addressierungsarten



(immediate)

(register direct)

(index + offset)

(PC + offset)

(PC | offset)

Adressierung: Varianten

welche Adressierungsarten / Varianten sind üblich?

0-Adress (Stack-) Maschine:	Java virtuelle Maschine
1-Adress (Akkumulator) Maschine.	8-bit Microcontroller
	einige x86 Befehle
2-Adress Maschine:	einige x86 Befehle,
	16-bit Rechner
3-Adress Maschine:	32-bit RISC

- CISC-Rechner unterstützen diverse Adressierungsarten
- RISC meistens nur indiziert mit offset

- später noch genügend Beispiele (und Übungen)

x86-Architektur

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

x86:

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

Hinweis:

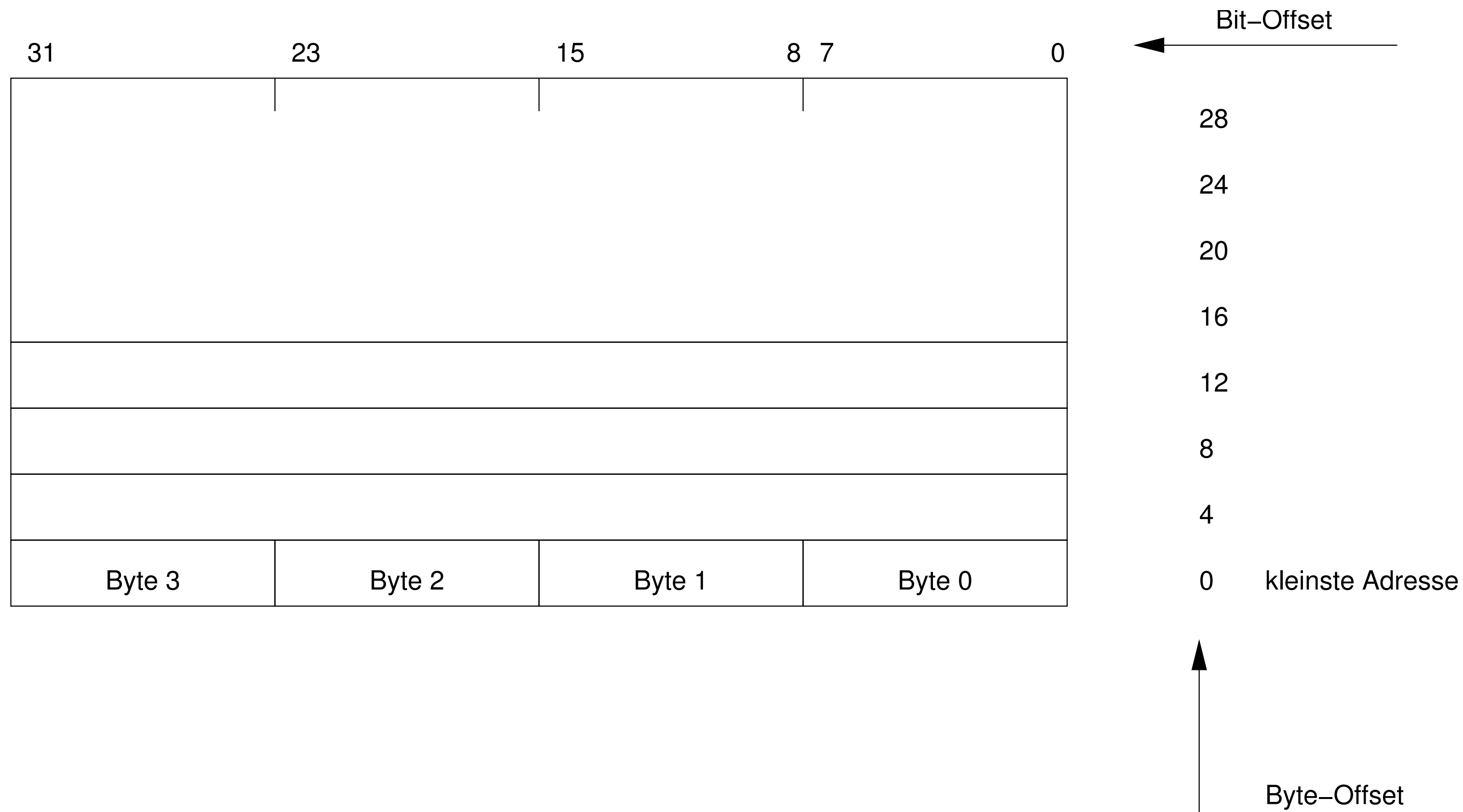
- die folgenden Folien zeigen eine *vereinfachte* Version
- niemand erwartet, dass Sie sich die Details merken
- x86-Assemblerprogrammierung ist „grausam“

Beispiel: Evolution des Intel x86

Chip	Date	MHz	Transistors	Memory	Notes
4004	4/1971	0.108	2,300	640	First microprocessor on a chip
8008	4/1972	0.108	3,500	16 KB	First 8-bit microprocessor
8080	4/1974	2	6,000	64 KB	First general-purpose CPU on a chip
8086	6/1978	5-10	29,000	1 MB	First 16-bit CPU on a chip
8088	6/1979	5-8	29,000	1 MB	Used in IBM PC
80286	2/1982	8-12	134,000	16 MB	Memory protection present
80386	10/1985	16-33	275,000	4 GB	First 32-bit CPU
80486	4/1989	25-100	1.2M	4 GB	Built-in 8K cache memory
Pentium	3/1993	60-233	3.1M	4 GB	Two pipelines; later models had MMX
Pentium Pro	3/1995	150-200	5.5M	4 GB	Two levels of cache built in
Pentium II	5/1997	233-400	7.5M	4 GB	Pentium Pro plus MMX

Figure 1-10. The Intel CPU family. Clock speeds are measured in MHz (megahertz) where 1 MHz is 1 million cycles/sec.

x86: Speichermodell



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

x86: Speichermodell

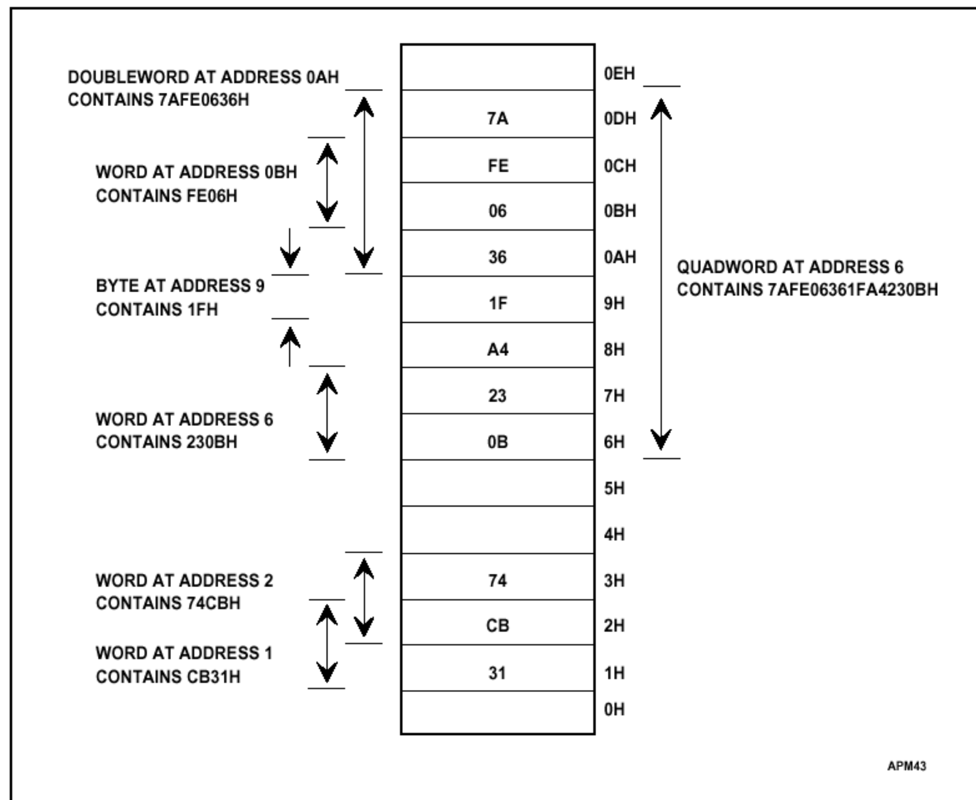


Figure 3-3. Bytes, Words, Doublewords and Quadwords in Memory

- Speicher voll byte-adressierbar
- mis-aligned Zugriffe langsam

x86: Register

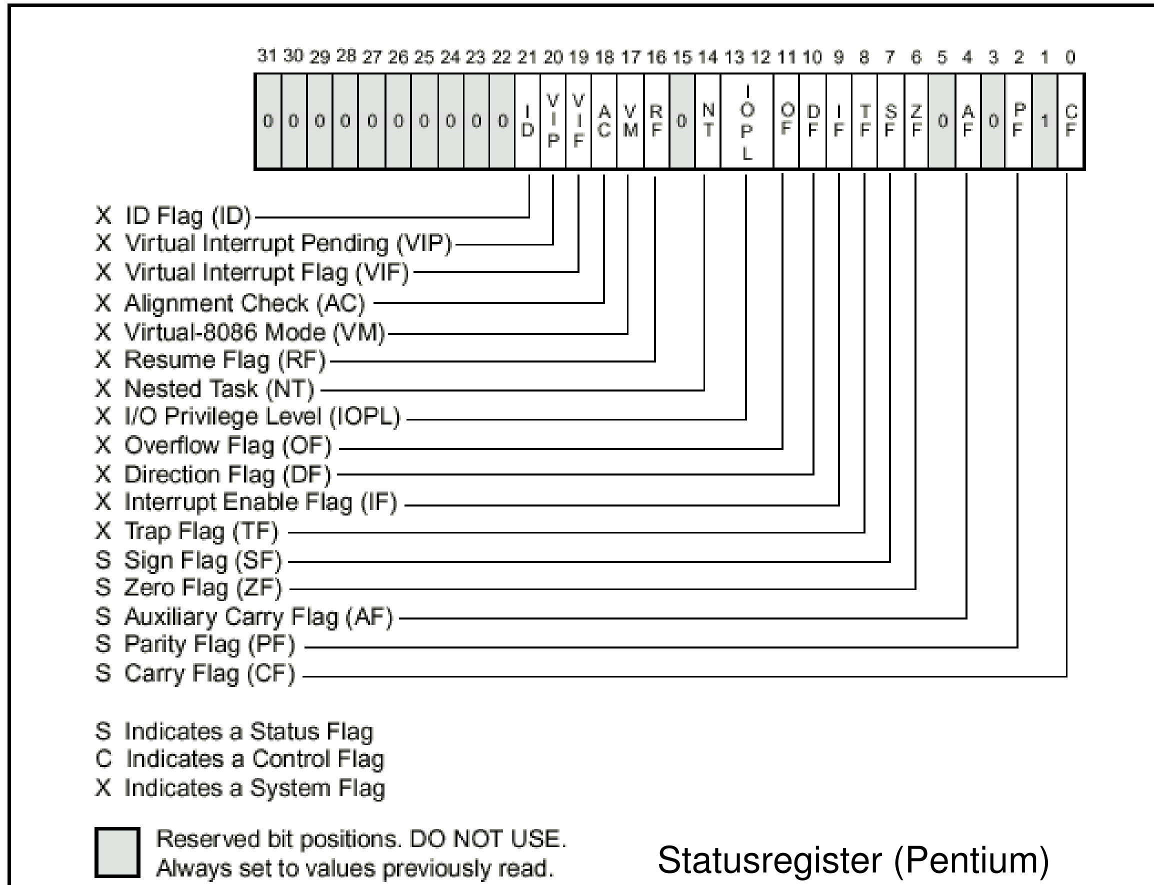
31		15	0	
EAX	AX	AH	AL	accumulator
ECX	CX	CH	CL	count: String, Loop
EDX	DX	DH	DL	data, multiply/divide
EBX	BX	BH	BL	base addr
ESP	SP			stackptr
EBP	BP			base of stack segment
ESI	SI			index, string src
EDI	DI			index, string dst
	CS			code segment
	SS			stack segment
	DS			data segment
	ES			extra data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

Legende	
grau	8086
weiß	Exx ab 386

79 … 0
FPR0
FPR7

FP Status

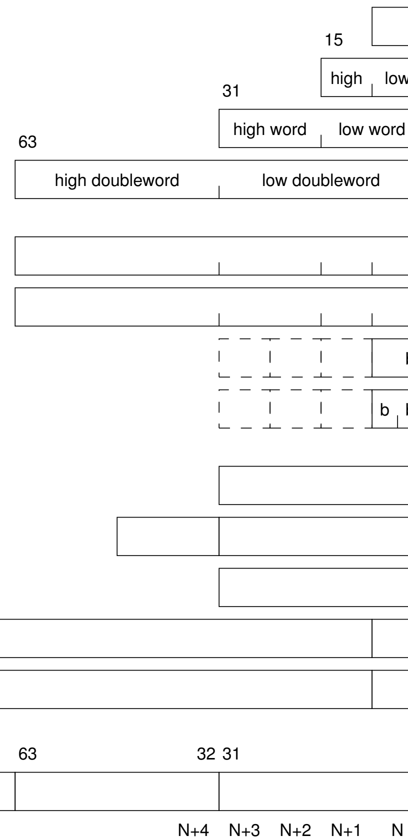
x86: EFLAGS Register



Statusregister (Pentium)

Datentypen: CISC...

bytes		
word		
doubleword		
quadword		
integer	(2–complement b/w/dw/qw)	
ordinal	(unsigned b/w/dw/qw)	
BCD	(one digit per byte, multiple bytes)	
packed BCD	(two digits per byte, multiple bytes)	
near pointer	(32 bit offset)	
far pointer	(16 bit segment + 32 bit offset)	
bit field		
bit string		
byte string		
float / double / extended		



x86: Befehlssatz

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

x86: Befehlsformate: CISC . . .

außergewöhnlich komplexes Befehlsformat:

1	prefix	(repeat / segment override / etc.)
2	opcode	(eigentlicher Befehl)
3	register specifier	(Ziel / Quellregister)
4	address mode specifier	(diverse Varianten)
5	scale-index-base	(Speicheradressierung)
6	displacement	(Offset)
7	immediate operand	

- ausser dem Opcode alle Bestandteile optional
- unterschiedliche Länge der Befehle, von 1 .. 37 Byte

⇒ extrem aufwendige Dekodierung

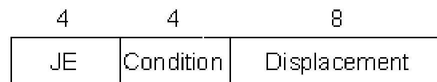
x86: Modifier

alle Befehle können mit „Modifiern" ergänzt werden:

• segment override	Addr. aus angewähltem Segmentregister
• address size	Umschaltung 16/32-bit
• operand size	Umschaltung 16/32-bit
• repeat	für Stringoperationen Operation auf allen Elementen ausführen
• lock	Speicherschutz für Multiprozessoren

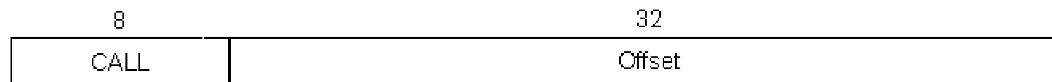
Beispiel: x86 Befehlskodierung

a. JE EIP + displacement

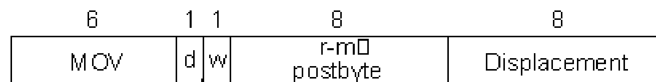


- 1 Byte .. 36 Bytes
- vollkommen irregulär

b. CALL

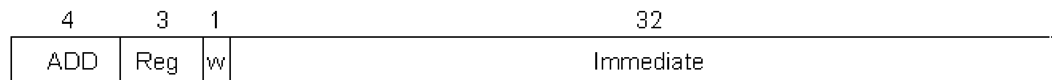


c. MOV EBX, [EDI + 45]

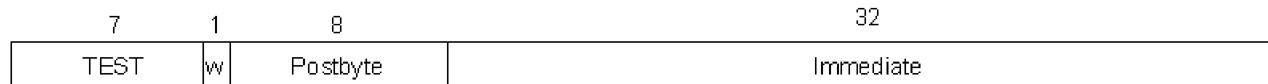


d. PUSH ESI

e. ADD EAX, #6765



f. TEST EDX, #42



Beispiel: x86 Befehle

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

x86: Assembler-Beispiel

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

x86: Assembler-Beispiel (2)

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

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

Ergänzende Literatur

Zur Rechnerarchitektur (2. Termin):

Literatur

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

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

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