64-040 Modul IP7: Rechnerstrukturen

10 Einführung in die Rechnerarchitektur

Norman Hendrich

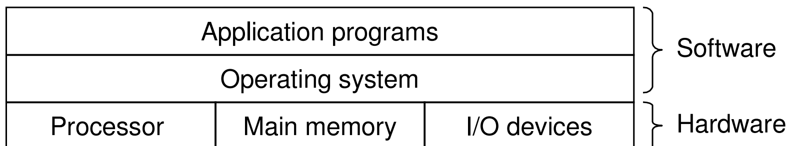
Universität Hamburg
MIN Fakultät, Department Informatik
Vogt-Kölln-Str. 30, D-22527 Hamburg
hendrich@informatik.uni-hamburg.de

WS 2013/2014

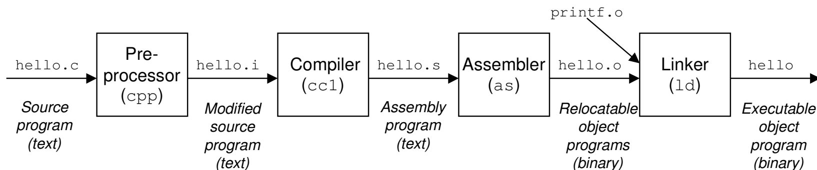
Inhalt

1. Einführung in die Rechnerarchitektur
    - Motivation
    - Beschreibungsebenen
    - Rechnerstrukturen
    - Wie rechnet ein Rechner?
2. Instruction Set Architecture (ISA)
    - Speicherorganisation
    - Befehlssatz
    - Befehlsformate
    - Adressierungsarten
3. Intel x86-Architektur

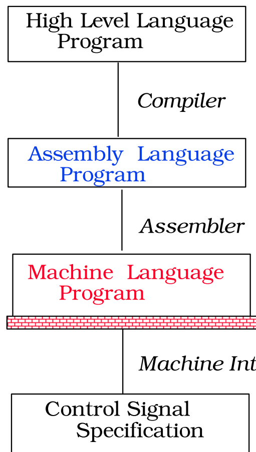
Wiederholung: Abstraktionsebenen



Das Kompilierungssystem



Abstraktionsebenen / Virtuelle Maschinen




```
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)
```

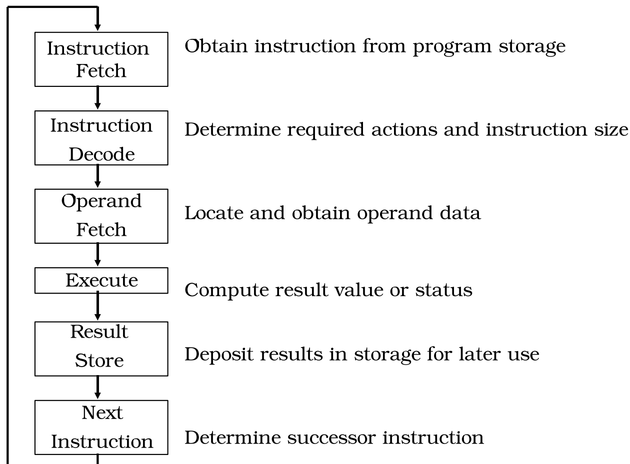
```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

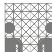
```
ALUOP[0:3] <= InstReg[9:11] & MASK
```

Assemblerebene

- Assemblerprogramme werden nur selten per Hand geschrieben
- aber: Verständnis des Assemblers ist unerlässlich für das Verständnis des Ausführungsmodells auf der Maschinenebene:
  - Programmverhalten bei Fehlern
    - das High-Level Sprachmodell ist dort nicht anwendbar
  - Programmleistung verbessern
    - Ursachen für Programm-Ineffizienz verstehen
  - Systemsoftware implementieren
    - der Compiler hat den Maschinencode als Ziel
    - die Betriebssysteme müssen den Prozesszustand verwalten
    - Gerätetreiber schreiben

Ausführungszyklus





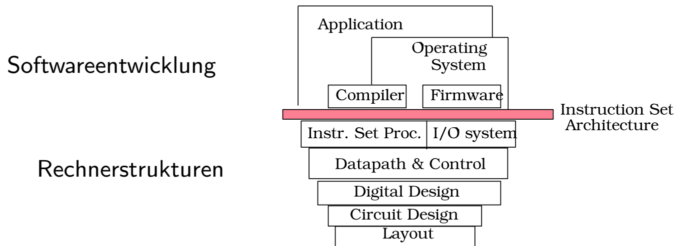
Rechnerstrukturen

- Bestandteile eines Rechners:
  - Prozessor
  - Input (Maus, Keyboard)
  - Output (Bildschirm, Drucker)
  - Speicher (Laufwerke, DRAM, SRAM, CD)
  - Netzwerk
- Unser primärer Schwerpunkt: der Prozessor (Datenpfad und Kontrolle)
  - wird mit Hilfe von Millionen Transistoren implementiert
  - ist unmöglich durch die Untersuchung jedes einzelnen Transistors zu verstehen
  - wir brauchen...

Rechnerstrukturen - Weitere Betrachtung

- Rechnerstrukturen
  - Rechnerarchitektur
  - Implementierung
- Rechnerarchitektur
  - Schnittstelle zwischen Rechner und Benutzer
    - Befehlssatzarchitektur
    - Maschinenorganisation
- Implementierung
  - Hardware-Aufbau von Komponenten,
    die die Rechnerarchitektur realisieren
  - Speichereinheiten, Recheneinheiten, Verbindungssysteme,...

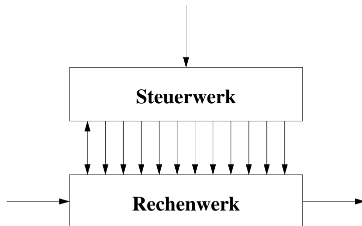
Was umfasst der Begriff Rechnerstrukturen?



- Koordination vieler Abstraktionsebenen der Beschreibung
- Berücksichtigung ständig wechselnder äußerer Einflüsse
- Entwurf, Leistungsmessung, Leistungsbewertung

von-Neumann-Architektur

1. Daten und Befehle liegen im gleichen Schreib-Lese-Speicher
2. Zugriff auf Speicherinhalte erfolgt über die Adresse der Speicherzelle
   - Speicherinhalte sind nicht typisiert
3. Programmausführung erfolgt sequentiell, Befehl für Befehl

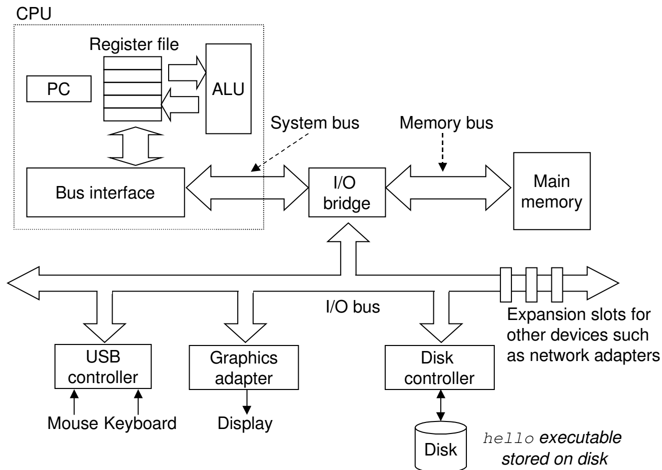


Maschinenorganisation

- Fähigkeiten und Leistung der prinzipiellen Funktionseinheiten
  - z.B., Registers, ALU, Shifters,...
- Verbindungen zwischen diesen Einheiten
- Informationsfluss zwischen den Komponenten
- Logik und Methoden zur Realisierung des Informationsflusses

- „Choreographie" der Funktionseinheiten
- Beschreibung auf Register-Transfer-Ebene
  (Register Transfer Level, RTL)

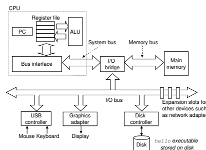
Hardwareorganisation eines typischen Systems



Befehlssatzarchitektur: „ISA“

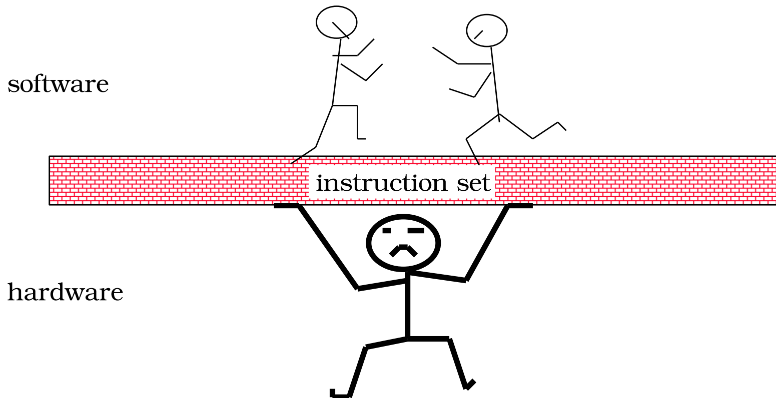
Instruction Set Architecture (ISA): alle für den Programmierer sichtbaren Attribute eines Rechners

- der Struktur



- des Verhaltens:
  - Organisation des programmierbaren Speichers
  - Datentypen und Datenstrukturen: Codierungen und Darstellungen
  - Befehlssatz
  - Befehlsformate
  - Modelle für Befehls- und Datenzugriffe
  - Ausnahmebedingungen

Der Befehlssatz: die zentrale Schnittstelle



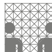
Merkmale der Instruction Set Architecture

- 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)

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



Beispiele für charakteristische ISA

(in dieser Vorlesung bzw. im Praktikum behandelt)

- 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. Microcontroller) werden aus Zeitgründen nicht vorgestellt

Artenvielfalt der Architekturen

Prozessor	4 .. 32 bit	8 bit	–	16 .. 32 bit	32 bit	32 bit	32 bit	8 .. 64 bit	..32 bit
Speicher	1K .. 1M	< 8K	< 1K	1 .. 64M	1 .. 64M	< 512M	8 .. 64M	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

Speicherorganisation

- Wortbreite, Grösse (=Speicherkapazität)
- „little-/big-endian“
- „Alignment“
- „Memory-Map“
- Beispiel: PC mit Windows

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

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

Speicher: Wortbreite

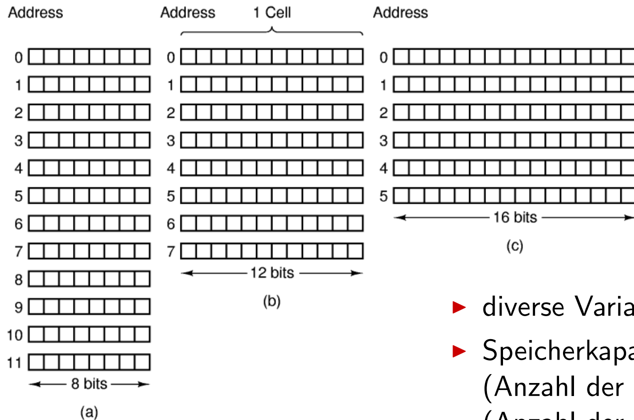
Adressierbare Speicherwortbreiten einiger historisch wichtiger Computer

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

- heute dominieren 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

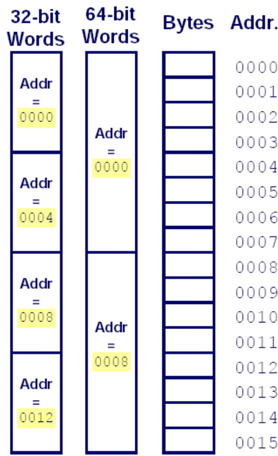
Drei Organisationsformen eines 96-bit Speichers



- diverse Varianten möglich
- Speicherkapazität:
  (Anzahl der Worte) *
  (Anzahl der Bits/Wort)

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



Datentypen auf Maschinenebene

- gängige Prozessoren unterstützen mehrere Datentypen
- entsprechend der elementaren Datentypen in C bzw. 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

Byte-Order

- Wie sollen die Bytes innerhalb eines Wortes angeordnet werden?
- Speicher wort-basiert, Adressierung byte-basiert.
  Zwei Möglichkeiten / Konventionen:

- **Big Endian**: Sun, Mac, usw.
  das MSB (*most significant byte*) hat die kleinste Adresse
  und das LSB (*least significant byte*) die höchste

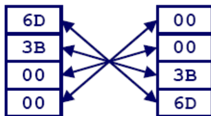
- **Little Endian**: Alpha, x86
  das MSB hat die höchste, das LSB die kleinste Adresse

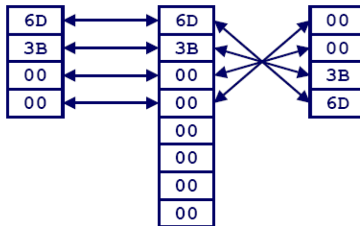
satirische Referenz auf Gulliver's Reisen (Jonathan Swift)

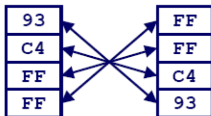
Byte-Order: Beispiel

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

Decimal:	15213			
Binary:	0011	1011	0110	1101
Hex:	3	B	6	D







Two's complement representation

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.
};
```

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-Byteorder

- Byteorder muss bei Datenübertragung zwischen Rechnern berücksichtigt und eingehalten werden

- Internet-Protokoll (IP) nutzt ein big-endian Format
- auf x86-Rechnern müssen alle ausgehenden und ankommenden Datenpakete umgewandelt werden

- zugehörige Hilfsfunktionen / Makros in `netinet/in.h`
  - inaktiv auf big-endian, **byte-swapping** auf little-endian
  - `ntohl(x)`: network-to-host-long
  - `htons(x)`: host-to-network-short
  - ...

Byte-Swapping: Beispiel

Linux: `/usr/include/bits/byteswap.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
...
/* Swap bytes in 32 bit value.  */
#define __bswap_constant_32(x) \
  ((((x) & 0xff000000) >> 24) \
 | (((x) & 0x00ff0000) >>  8) \
 | (((x) & 0x0000ff00) <<  8) \
 | (((x) & 0x000000ff) << 24))
```

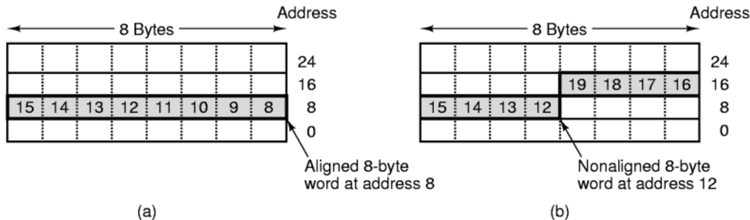
Programm zum Erkennen der Byteorder

- Programm gibt Daten byteweise aus
- C-spezifische Typ- (Pointer-) Konvertierung
- Details: siehe Bryant 2.1.4 (und figures 2.3/2.4)

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

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

Misaligned Memory Access



- Speicher 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
- aber nicht alle Systeme haben voll ausgebauten Speicher

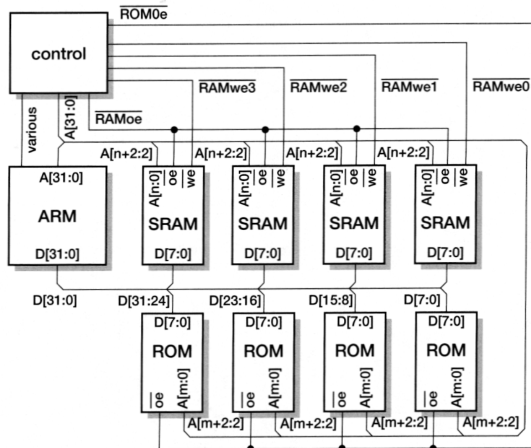
  z.B. 32-bit Adresse entspricht bereits 4GB Hauptspeicher...

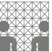
- Aufteilung in RAM und ROM-Bereiche
  - ROM mindestens zum Booten notwendig
- zusätzliche Speicherbereiche für „memory mapped“ I/O

- Memory Map
  - die Zuordnung von Adressen zu realem Speicher
  - aufgeteilt in RAM, ROM, I/O, leere Bereiche
  - “Adress-Dekoder”-Schaltwerk aktiviert die einzelnen Blöcke

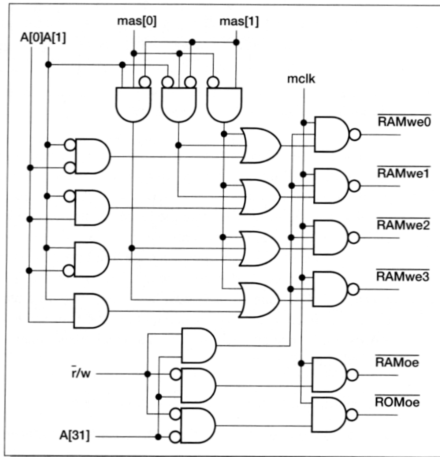
Typisches Speichersystem

32-bit Prozessor, je 4 8-bit SRAMs und ROMs





Typisches Speichersystem: Adressdekodierung



Memory Map: typ. 16-bit System

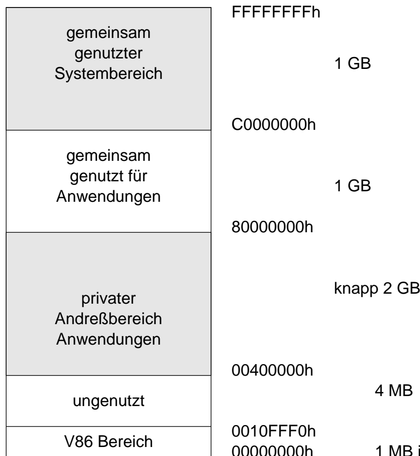
- 16-bit erlaubt 64K Adressen: 0x0000 .. 0xFFFF

- ROM-Bereich für Boot / Betriebssystemkern
- RAM-Bereich für Hauptspeicher (Anwendungsprogramme)
- RAM-Bereich für Betriebssystem (z.B. Interrupt-Tabelle)

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

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

Memory Map: Windows-9x



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

Memory Map: Windows-9x

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

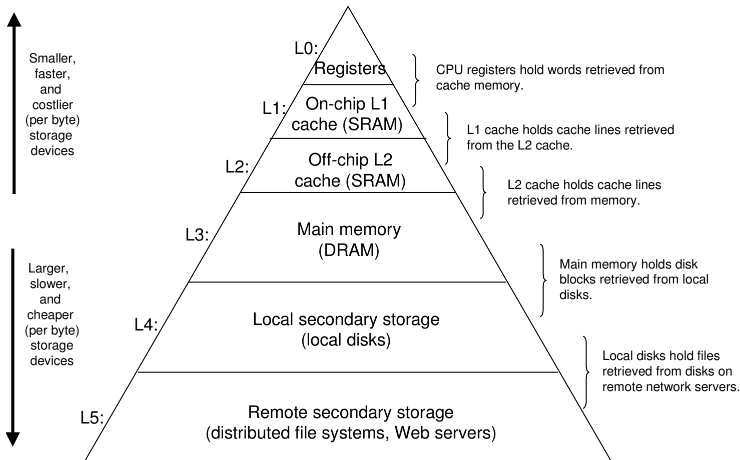
- 32-bit Adressen, 4 GByte Adressraum
- Aufteilung 2 GB für Programme, obere 1+1 GB für Windows
- Beispiel der Zuordnung, diverse Bereiche für I/O reserviert

Memory Map: I/O-Speicherbereiche

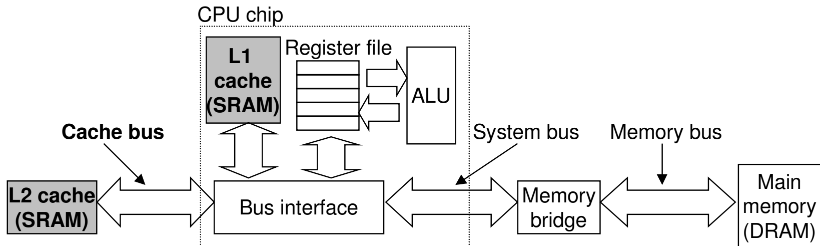
[00000378 - 0000037F] ECP-Druckeranschluss (LPT1)
[00000380 - 000003BB] Hauptplatinenressourcen
[000003B0 - 000003BB] Intel(R) Q963/Q965 PCI Express Root Port – 2991
[000003B0 - 000003BB] Radeon X1300/X1550 Series
[000003C0 - 000003DF] Intel(R) Q963/Q965 PCI Express Root Port – 2991
[000003C0 - 000003DF] Radeon X1300/X1550 Series
[000003C0 - 000003E7] Hauptplatinenressourcen
[000003F0 - 000003F5] Standard-Diskettenlaufwerkcontroller
[000003F6 - 000003F7] Hauptplatinenressourcen
[000003F7 - 000003F7] Standard-Diskettenlaufwerkcontroller
[000003F8 - 000003FF] Kommunikationsanschluss (COM1)
[00000400 - 000004CF] Hauptplatinenressourcen
[000004D0 - 000004D1] Programmierbarer Interruptcontroller

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

Speicherhierarchie



Cache-Speicher



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 addressiert den Speicher
  - gelesener Wert kommt in das Befehlsregister IR
  - Befehl dekodieren
  - 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

- 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:

$$IR = MEM\ [PC]$$
$$PC = PC + 4$$

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

Instruction Decode

„Befehl dekodieren“ - Phase im Befehlszyklus

- Befehl steht im Befehlsregister IR
- Decoder entschlüsselt Opcode und Operanden
- leitet Steuersignale an die Funktionseinheiten
- Programmzähler wird inkrementiert

Instruction Execute

„Befehl ausführen" - Phase im Befehlszyklus

- Befehl steht im Befehlsregister IR
- Decoder hat Opcode und Operanden entschlüsselt
- Steuersignale liegen an Funktionseinheiten

- Ausführung des Befehls durch Aktivierung der Funktionseinheiten

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

- Realisierung über festverdrahtete Hardware
  oder mikroprogrammiert

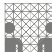
- Demo (bzw. im T3-Praktikum):
  - Realisierung des Mikroprogramms für den D*CORE

Welche Befehle braucht man?

Befehls-„Klassen“:	Beispiele:
• arithmetische Operationen	add, sub, mult, div
logische Operationen	and, or, xor
Schiebe-Operationen	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 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“

- Oberbegriff für moderne Rechnerarchitekturen
  entwickelt ab ca. 1980 bei IBM, Stanford, Berkeley
- auch bekannt unter: „regular instruction set computer“
- typische Merkmale:
  - reguläre Struktur, z.B. 32-bit Wortbreite, 32-bit Befehle
  - nur ein-Wort-Befehle
  - alle Befehle in einem Taktschritt ausführbar
  - „Load-Store“ Architektur, keine Speicheroperanden
  - viele universelle Register, keine Spezialregister
  - optimierende Compiler statt Assemblerprogrammierung
- Beispiele: IBM 801, MIPS, SPARC, DEC Alpha, ARM
- Diskussion und Details CISC vs. RISC später

Befehls-Dekodierung

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

0 1 0 0 1 1 1 0	1 1 1 1 1 1 1 0	0 0 0 0 0 0 0 0	0 0 0 0 1 0 0 0

31 ... 0

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

$\Rightarrow$ „Befehlsformate“

Befehlsformate

Beispiel:

- 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 Befehlssatz

Befehlsformat: drei Beispielarchitekturen

- MIPS: Beispiel für 32-bit RISC Architekturen
  - alle Befehle mit 32-bit kodiert
  - 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 Kodierungen 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
  - Adressierungsarten werden hier mit codiert
- 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 .. R1, jeweils 32-bit
- R1 bis R31 sind Universalregister
- R0 ist konstant Null (ignoriert Schreiboperationen)
  - erlaubt einige Tricks:

$R5 = -R5$	sub R5, R0, R5
$R4 = 0$	add R4, R0, R0
$R3 = 17$	addi R3, R0, 17
$\text{if } (R2 == 0)$	bne R2, R0, label

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

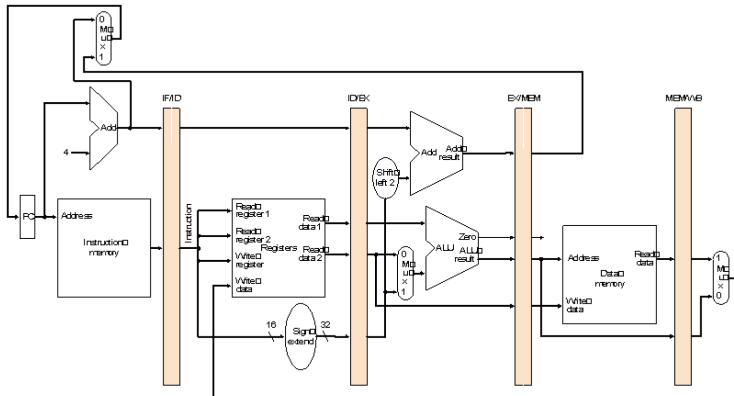
  $R1 = (R2 < R3)$ slt R1, R2, R3

MIPS: Befehlssatz

- Übersicht und Details: siehe Hennessy & Patterson

- dort auch hervorragende Erläuterung der Hardwarestruktur
- klassische fünf-stufige Befehlspipeline (F-D-E-M-W)
  - Instruction-Fetch — Befehl holen
  - Decode — Dekodieren und Operanden holen
  - Execute — ALU-Operation oder Adressberechnung
  - Memory — Speicher lesen oder schreiben
  - Write-back — Resultat in Register speichern

MIPS: Hardwarestruktur



PC
I-Cache

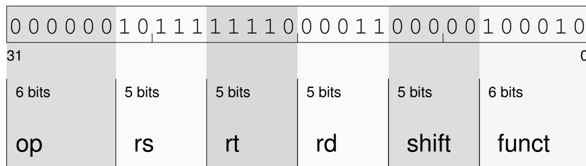
Register
(R0 .. R31)

ALUs

Speicher
D-Cache

Befehlsformat: Beispiel MIPS

Befehl im R-Format



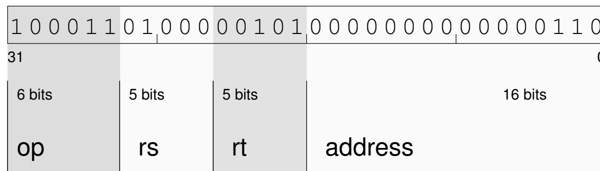
R–Format

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“

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

Befehlsformat: Beispiel MIPS

Befehl im I-Format



I–Format

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

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

Befehlsformat: Beispiel M*CORE

- 32-bit RISC Architektur, Motorola 1998
- besonders einfaches Programmiermodell:

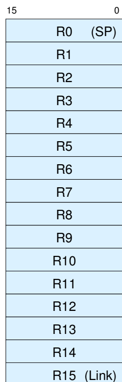
○ Program Counter,	PC
○ 16 Universalregister	R0 .. R15
○ Statusregister	C („carry flag“)
○ 16-bit Befehle	(um Programmspeicher zu sparen)

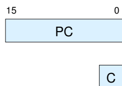
- Verwendung:
  - häufig in Embedded-Systems
  - „smart cards“

D*CORE

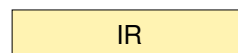
- ähnlich M*CORE
- gleiches Registermodell, aber nur 16-bit Wortbreite
  - Program Counter, PC
  - 16 Universalregister R0 .. R15
  - Statusregister C („carry flag“)
- Subset der Befehle, einfachere Kodierung
- vollständiger Hardwareaufbau in Hades verfügbar
- oder Simulator mit Assembler (winT3asm.exe / t3asm.jar)

D*CORE: Registermodell

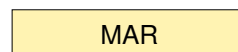


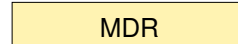


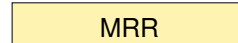
– 16 Universalregister
– Programmzähler
– 1 Carry–Flag



– Befehlsregister







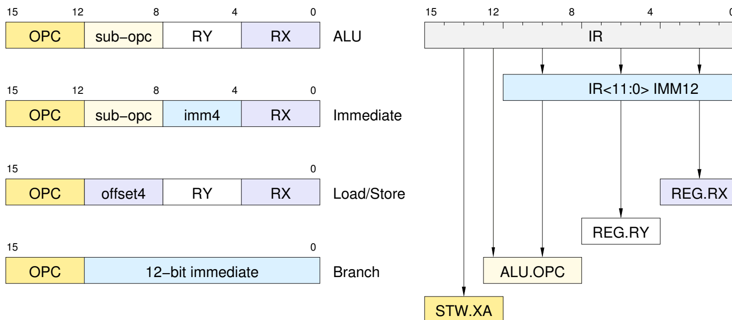
– Bus–Interface

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

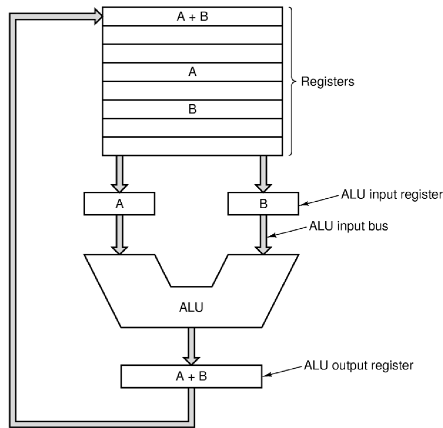
- Rechner soll „rechnen“ können
- typische arithmetische Operation nutzt 3 Variablen
- Resultat, zwei Operanden: $X = Y + Z$

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

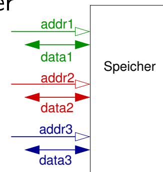
Datenpfad

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



Woher kommen die Operanden?

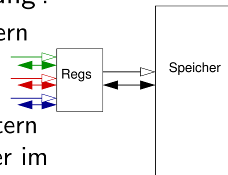
- typische Architektur:
  - von-Neumann Prinzip: alle Daten im Hauptspeicher
  - 3-Adress-Befehle: zwei Operanden, ein Resultat



- „Multiport-Speicher“: mit drei Ports ?
  - sehr aufwendig , 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 $n = \{3 .. 0\}$

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

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

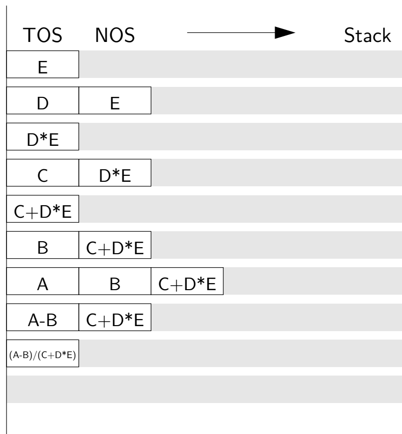
```
push  E
push  D
mul
push  C
add
push  B
push  A
sub
div
pop   Z
```

Stack-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
```



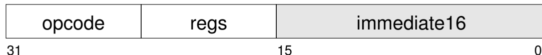
Adressierungsarten

- „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)

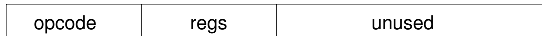
Adressierungsarten (cont.)

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

Immediate-Adressierung



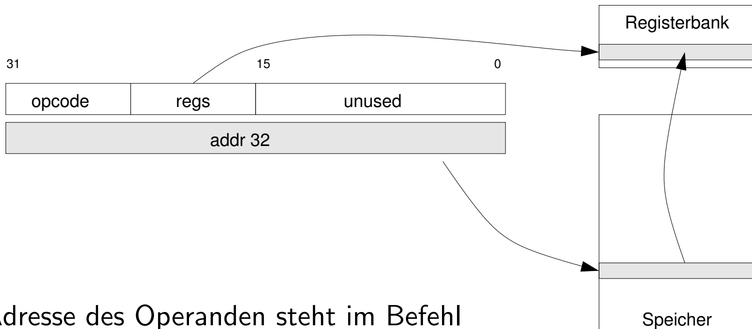
1–Wort Befehl



2–Wort Befehl

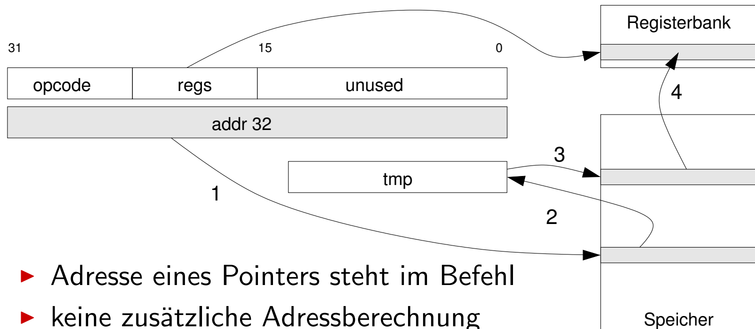
- Operand steht direkt im Befehl, kein zusätzlicher Speicherzugriff
- Länge des Operanden ist kleiner als (Wortbreite - Opcodebreite)
- zur Darstellung grösserer Werte:
  - 2-Wort Befehle (x86)
    (zweites Wort für Immediate-Wert)
  - mehrere Befehle (Mips,SPARC)
    (z.B. obere/untere Hälfte eines Wortes)
  - Immediate-Werte mit zusätzlichem Shift (ARM)

Direkte Adressierung



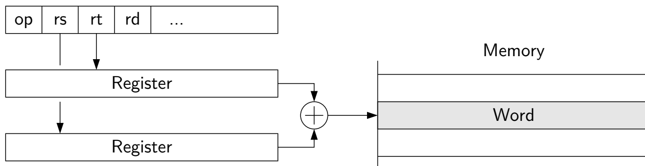
- Adresse des Operanden steht im Befehl
- keine zusätzliche Adressberechnung
- ein zusätzlicher Speicherzugriff: z.B. `R3 = MEM[ addr32 ]`
- Adressbereich beschränkt, oder 2-Wort Befehl (wie Immediate)

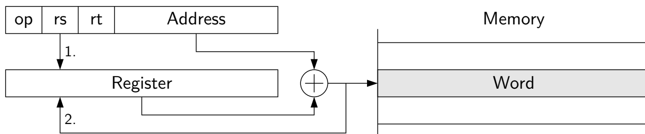
Indirekte Adressierung



- 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





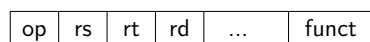
- indizierte Adressierung, z.B. für Arrayzugriffe
  - addr = (Sourceregister) + (Basisregister)
  - addr = (Sourceregister) + offset;
    Sourceregister = addr

Beispiel: MIPS Adressierungsarten

1. Immediate addressing

immediate

2. Register addressing



register

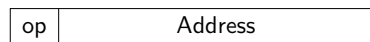
3. Base addressing

index + offset

4. PC-relative addressing

PC + offset

5. Pseudodirect addressing



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

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

Intel x86-Architektur

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

- irreguläre Struktur: CISC
- historisch gewachsen: diverse Erweiterungen (MMX, SSE, ...)
- ab 386 auch wie reguläre 8-Register Maschine verwendbar
- 64-bit Erweiterung “IA-64”, abwärtskompatible mit IA-32

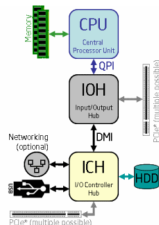
Hinweis: niemand erwartet, dass Sie sich alle Details merken...

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 KB	erster 8-bit Mikroprozessor
8080	4/1974	2	6.000	64 KB	„general-purpose“ CPU auf einem Chip
8086	6/1978	5–10	29.000	1 MB	erste 16-bit CPU auf einem Chip
8088	6/1979	5–8	29.000	1 MB	Einsatz im IBM-PC
80286	2/1982	8–12	134.000	16 MB	„Protectec-Mode“
80386	10/1985	16–33	275.000	4 GB	erste 32-Bit CPU
80486	4/1989	25-100	1.2M	4 GB	integrierter 8K Cache
Pentium	3/1993	60–233	3.1M	4 GB	zwei Pipelines, später MMX
Pentium Pro	3/1995	150–200	5.5M	4 GB	integrierter first und second-level Cache
Pentium II	5/1997	233–400	7.5M	4 GB	Pentimum Pro plus MMX
Pentium III	2/1999	450–1400	9.5–44M	4 GB	SSE-Einheit
Pentium IV	11/2000	1.300–3.600	42–188M	4 GB	Hyperthreading
Core-2	5/2007	1600–3200	143–410M		64-bit Mehrkernprozessoren
Core-i	11/2008	2500–3600	$>$ 700M		Taktanpassung, AVX
...					

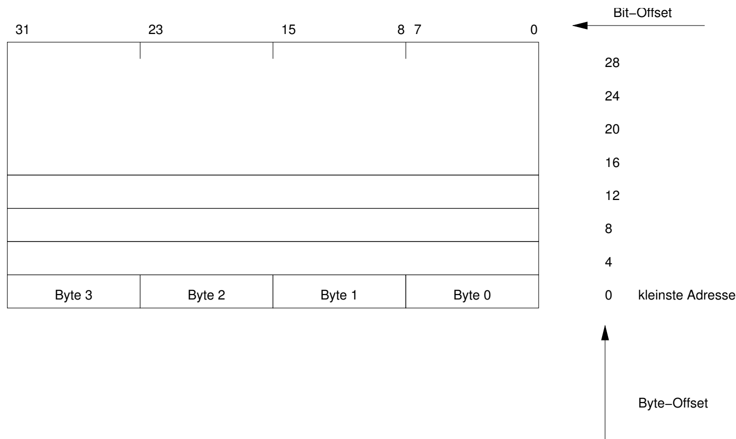
Beispiel: Intel Core i7-960 Processor

Taktfrequenz	bis 3,46 GHz
Anzahl der Cores	4
QPI Durchsatz (quick path interconnect)	4,8 GT/s
Bus Interface	64 Bits
L1 Cache	4x (32 kB I + 32kB D)
L2 Cache	4x 256 kB (I+D)
L3 Cache	8192 kB (I+D)
Prozess	45 nm
Versorgungsspannung	0,8 - 1,375V
Wärmeabgabe	∼ 130 W
Performance (SPECint 2006)	∼ 35



http://ark.intel.com/

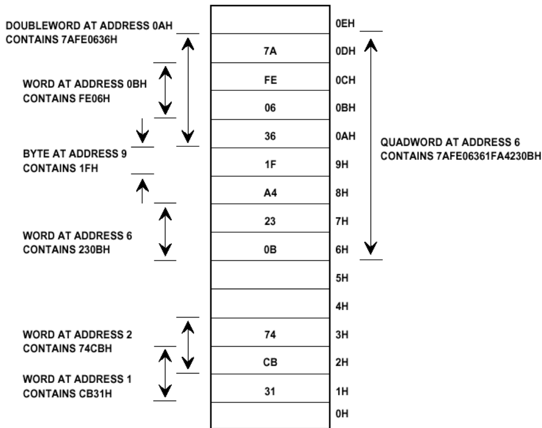
x86: Speichermodell



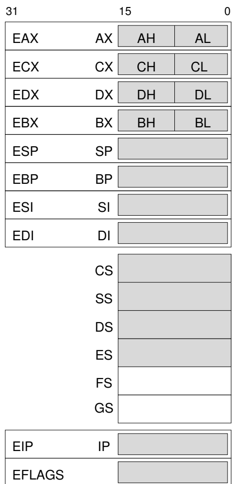
- „little endian“: LSB eines Wortes bei der kleinsten Adresse

x86: Speichermodell

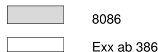
- Speicher voll byte-adressierbar
- mis-aligned Zugriffe langsam
- Beispiel zeigt:
  - Byte
  - Word,
  - Doubleword,
  - Quadword,

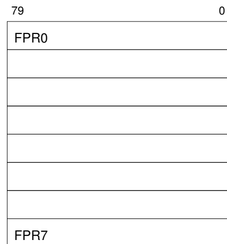


x86: Register



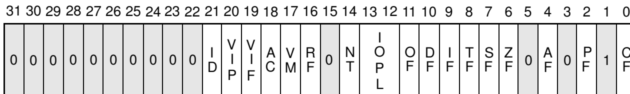
accumulator
count: String, Loop
data, multiply/divide
base addr
stackptr
base of stack segment
index, string src
index, string dst
code segment
stack segment
data segment
extra data segment
PC
status


8086
Exx ab 386



FP Status

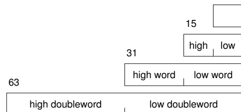
x86: EFLAGS Register



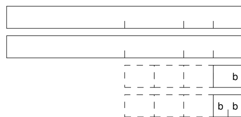
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)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

x86: Datentypen

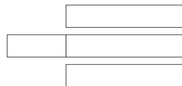
bytes
word
doubleword
quadword



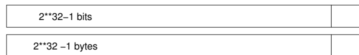
integer (2–complement b/w/dw/qw)
ordinal (unsigned b/w/dw/qw)
BCD (one digit per byte, multiple bytes)
packed BCD (two digits per byte, multiple bytes)



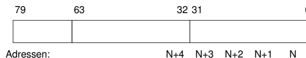
near pointer (32 bit offset)
far pointer (16 bit segment + 32 bit offset)
bit field



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

⇒ extrem aufwendige Dekodierung

⇒ „CISC“

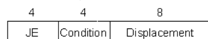
x86: Befehlsformat-Modifier („prefix“)

- alle Befehle können mit „Modifiern“ ergänzt werden:

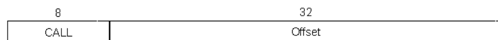
segment override	Adresse aus angewähltem Segmentregister
address size	Umschaltung 16/32-bit Adresse
operand size	Umschaltung 16/32-bit Operanden
repeat	für Stringoperationen: Befehl auf allen Elementen ausführen
lock	Speicherschutz bei Multiprozessorsystemen

x86 Befehlskodierung: Beispiele

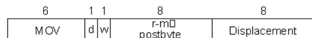
a. JE EIP + displacement



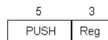
b. CALL



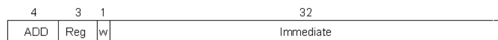
c. MOV EBX, [EDI + 45]



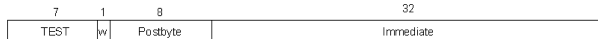
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



- 1 Byte .. 17 Bytes
- vollkommen irregulär
- w: Auswahl 16/32 bit

x86 Befehlssatz: Beispiele

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 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 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 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: Befehle analysieren

Statistiken zeigen: Dominanz der einfachen Instruktionen

- x86-Prozessor

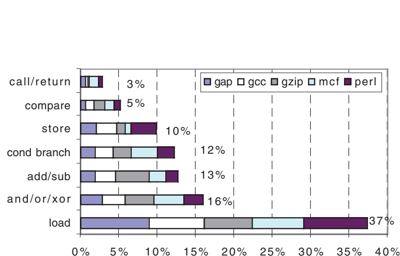
	Anweisung	Ausführungshäufigkeit %
1.	load	22 %
2.	conditional branch	20 %
3.	compare	16 %
4.	store	12 %
5.	add	8 %
6.	and	6 %
7.	sub	5 %
8.	move reg-reg	4 %
9.	call	1 %
10.	return	1 %
Total		96 %

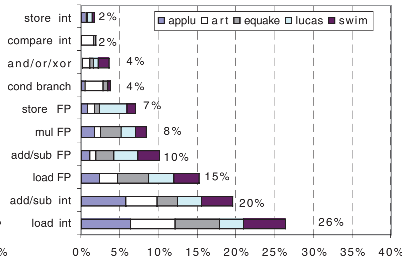
x86: Instruction Usage

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.

MIPS: Instruction Usage





SPECint2000 (96%) SPECfp2000 (97%)

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

⇒ Motivation für RISC