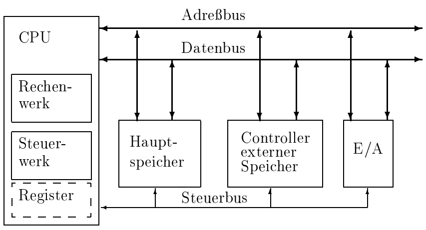
Kapitel 1

Rechnerarchitekturen

1.1 Das von-Neumann-Konzept

Abbildung 1.1: Universalrechner

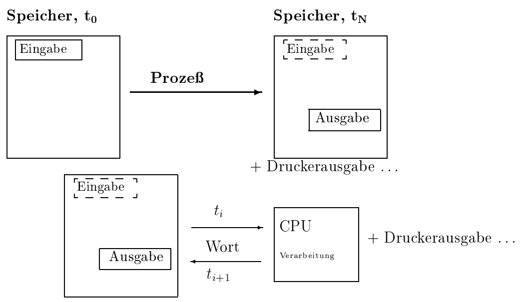


Prinzipien:

1. **Computerbestandteile:** CPU (= Rechenwerk und Steuerwerk), Speicher, Ein- und Ausgabewerk.
2. **Universalität:** Spezialisierung nur durch von außen eingegebenes Programm.
3. **Programme als Daten:** Eingabedaten, Zwischen- und Endergebnisse sowie Programme werden im selben Speicher abgelegt.
4. **linearer Speicher:** Der Speicher ist in gleichgroße Zellen, die fortlaufend numeriert sind, eingeteilt.

5. **sequentielle Ausführung:** Aufeinanderfolgende Befehle eines Programms werden in aufeinanderfolgenden Speicherzellen abgelegt. Beim Programmablauf wird im allgemeinen der Befehlszähler fortlaufend inkrementiert.
6. **Sprünge:** Sprungbefehle ermöglichen ein Durchbrechen der linearen Ausführungsreihenfolge.
7. **zusätzliche Primitiva:** Datentransferbefehle, arithmetische Operationen, logische und Vergleichsoperationen, *heute auch — jedoch nicht in klassischen CPUs :* Multimedia-Befehle (insbesondere Echtzeit-Komprimierung/-Dekomprimierung; vgl. MPEG = Standard der `Moving Picture Experts Group` und JPEG = Standard der `Joint Photographic Experts Group`); indizierte, relative, … Adressierung.
8. **binäre Codierung:** Daten (Befehle, Adressen, Operanden) werden binär codiert. Geeignete Schaltwerke (Decodierer) sorgen für die richtige Entschlüsselung.

Abbildung 1.2: Dekomposition von Prozessen in Sequenzen von Primitiva

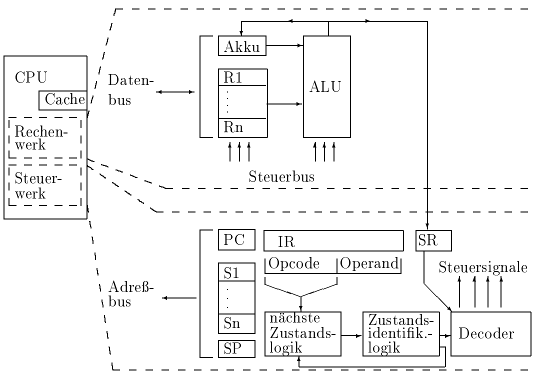


1.2 CISC/RISC

Traditionelle CISC's (= complex instruction set computers) sind:

- IBM360
- DEC VAX
- Intel 80386 (80x86)
- Motorola 68030 (680x0)
- …

Abbildung 1.3: CISC-CPU-Aufbau



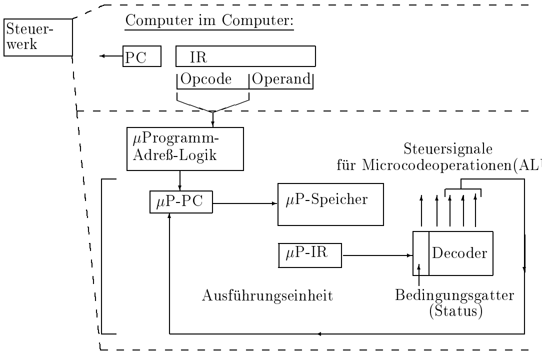
In Zeiten, als der Zeitraum zur Übertragung eines Datenwortes vom Speicher in ein Register oder umgekehrt Größenordnungen länger dauerte als ein einfacher logischer oder arithmetischer Primitiv-Befehl, mußte zunächst die Programmabarbeitung durch *wait-states* künstlich verlangsamt werden.
Man versuchte dann aus Gründen der Ökonomie, eine CPU immer komplexere Primitiv-Befehle (deren Ausführungszeit lediglich anfangs immer noch höchstens

Abbildung 1.4: Programmausführung

```
PC := StartingAddress;
RunBit := 1;
while RunBit = 1 do begin
  IR := memory(PC);
  PC := PC + Befehlslänge(IR);
               häufig=1, fest
  InstrType := DetermineInstrType(IR);
  DataAddress := LocateData(IR,InstrType,PC);
  if DataAddress <> nil then
    data := memory(DataAddress)
  else
    data := non_valid;
  Execute(IR,InstrType,data,memory,registers,PC,RunBit);
end;
```

eine „Datentransferzeit Register in Speicher“ lang war) ausführen zu lassen. In der damaligen Zeit waren dafür jedoch nicht genügend Schaltoperationen auf einem Chip realisierbar, weshalb man das Rechenwerk der CPU wiederum als Computer aufbaute:

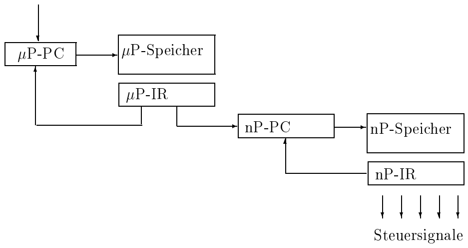
Abbildung 1.5: Microprogrammierte CISC-CPU



Software („SW“) reduziert den Schaltungsaufwand (= Anzahl der Gatter, Anzahl der Leitungen,...) und „interpretiert“ den Maschinencode der CPU (Computer im Computer).

Eventuell werden sogar „noch“ Nanoprogramme benutzt (68000er,...):

Abbildung 1.6: Nanoprogrammierte CISC-CPU



Das CISC-Designprinzip:

Reduziere die „semantic gap“ zwischen Maschinensprache und Hochsprache durch

- viele komplexe Maschinenbefehle ($\gtrsim$ 200), etwa „case“, „while“, ...
- viele Adressierungsmodi, etwa für „Felder“, „Verbunde“, ...
- Unterprogramm-Management im Maschinencode

unter besonderer Beachtung der mindestens um den Faktor 10 langsameren Transferbefehle zum/vom Speicher.

„Überalterung“ von CISC:

Ungefähr 1970 wird

- die Komplexität des Microprogramms immer schlechter zu managen
- der Speicherzugriff schneller
- lohnt sich ein Redesign aufgrund von Messungen „typischer“ Anwendungen:

Tabelle 1.1: Analyse typischer Computeranwendungen auf ihre Instruktionstypen hin

%-Verteilung	SAL Operating System Tanenbaum (1978)	XPL Systempgm. Wortman (1972)	FORTRAN num. Pgm. Knuth (1971)	C Systempgm. Patterson (1982)	Pascal Systempgm. Patterson (1982)
:=	47[1)]	55	51	38	45
if	17	17	10	43	29
call	25	17	5	12	15
loop	6	5	9	3	5
goto	0	1	9	3	0
sonstiges	5	5	16	1	6

Dabei sind in [1)] 80% der Wertzuweisungen von dem einfachen Typ „Variable := Wert“.

Konsequenz: Der Overhead durch das komplexe Microprogramm und die vielen Adressierungsmodi wird nur sehr selten auch ausgenutzt.

RISC (= reduced instruction set computer)-Designprinzip:

- Analysiere Anwendungen, um die Schlüsseloperationen zu finden.
- Reduziere unter allen Umständen die „Datenpfadzykluszeit“ (Register → ALU → Register) für diese Operationen, also: *keine* Microcode Interpretation (Maschinencode = Microcode)
- Jedes „neue“ Feature ist nur zuzulassen, wenn es häufig benutzt wird und die Mehrzahl der existierenden nicht verlangsamt.

(Wiederhole diesen Design-Zyklus für die anderen CPU-Ressourcen: Cache, Speichermanagement, Gleitkomma-Koprozessoren, ...)

Tabelle 1.2: Von CISC zu RISC

	CISC			RISC		
	IBM 370/168	VAX 11/780	Xerox Dorado	IBM 801	Berkeley RISC I	Stanford MIPS
Jahr	1973	1978	1978	1980	1981	1983
# Operationen	208	303	270	120	39	55
Microcode-Größe [byte]	54K	61K	17K	—	—	—
Befehlslänge [byte]	2–6	2–57	1–3	4	4	4
Ausführungsmodell	reg-reg reg-men men-men	reg- reg reg-men men-men	stack	reg-reg	reg-reg	reg-reg

CISC- und RISC-Architekturen lassen sich durch einige charakteristische Merkmale voneinander unterscheiden:

Tabelle 1.3: Unterscheidungsmerkmale CISC/RISC

	Bei RISC üblich	Bei CISC üblich
1.	einfache Befehle, 1 Befehl/Zyklus	komplexe Befehle, 1 Befehl/viele Zyklen
2.	LOAD/STORE-Architektur (nur LOAD/STORE greift auf Speicher zu)	mem-reg, mem-mem Befehle
3.	pipelined	kaum pipelined
4.	Hardware, kein Microcode	Microcode
5.	feste Befehlslänge	variable Befehlslänge (1...17 byte beim 80386)
6.	Wenige Befehle, wenige Adressierungsmodi	viele Befehle und Adressierungsmodi
7.	Komplexität im Compiler	Komplexität im Microprogramm
8.	Registerstack	feste Registermenge

Bemerkungen:

zu 1. Deswegen evtl. sogar keine Multiplikation/Division, die dann in Runtime-Bibliotheken angeboten werden muß.

zu 2. Vorhandene RISC-Varianten: (load|store) (signed|unsigned) (byte|halfword|word)

zu 3.
- delayed load/store: Codereorganisation bzw. NOP's
- Pipeline:

Zyklus	1	2	3	4	5	6	7 ...
instr1	fetch	decode	exec	write			
instr2		fetch	decode	exec	write		
instr3			fetch	decode	exec	write	
⋮							

zu 4. in RISC viel Chip-Platz frei: mehr Register realisierbar

zu 5. Die variable Länge benötigte Microcode (oder aufwendige Decoder).

zu 6.
- Befehlsdecoder-Länge wächst exponentiell mit der Anzahl der Befehle.
- Geschwindigkeit/Komplexität: wenige Adressierungsmodi

zu 7. In RISC-Systemen sind gute Compiler unumgänglich.

zu 8. Registerstack:

- ∼ 500 Register, von denen 32 gleichzeitig verfügbar sind
- überlappende Register-Fenster (s.u.)
- Register-Mehrfachbenutzung („Lebenszeit“ von Variablen)

Der Registerstack ist auch bei CISC möglich, wenn genügend Chip-Platz verfügbar ist.

Abbildung 1.7: Registerstack am Beispiel SPARC