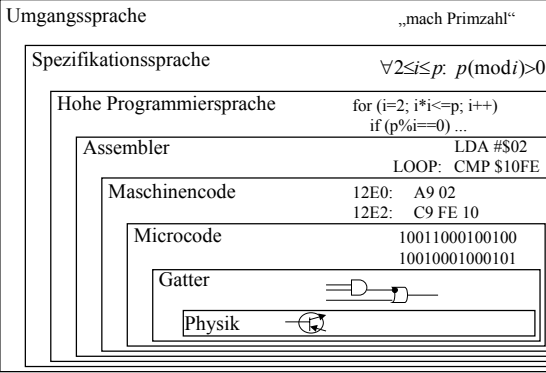


# Die Software-/Hardware-Hierarchie



Vom Programm zur Maschine

# Die Software-/Hardware-Hierarchie

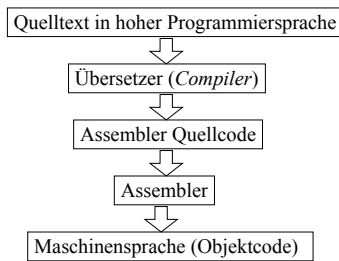
geschichtliche Entwicklung:

- Programmierung durch manuelle **Verdrahtung** der Bauteile
- Definition der Reihenfolge der Gatter (**Microcode**)
- Zusammenfassen komplexer Operationen (z.B. Addition) => Maschinensprache (**Hexcode**)
- Werkzeug für bessere Lesbarkeit und leichteren Entwurf von Maschinenspracheprogrammen (**Assembler**)
- Wunsch nach hardwareunabhängiger Programmierung => höhere **Programmiersprachen** (ALGOL, Lisp, ..., Java)
- Deklarative Programmiersprachen (Prolog), **Problemspezifikationssprachen**

Vom Programm zur Maschine

# Die Software-/Hardware-Hierarchie

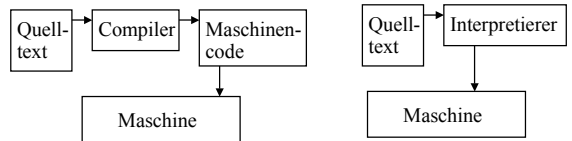
Zusätzliche Hierarchieebene => neuer Abbildungsmechanismus



Vom Programm zur Maschine

# Die Software-/Hardware-Hierarchie

Compiler vs. Interpretierer



- Compiler erzeugt eigenständiges Programm
- Interpretierer führt einzelne Kommandos unmittelbar aus

Auch Mischformen: Compiler+Interpretierer, z.B. Lisp, Java

Vom Programm zur Maschine

# Die Software-/Hardware-Hierarchie

Compiler vs. Interpretierer

Vorteile Compiler:

- erzeugt eigenständiges ausführbares Programm
- Programm kann relativ schnell abgearbeitet werden
- Quellcode kann geheim gehalten werden

Vorteile Interpretierer:

- leichte interaktive Programmerstellung
- kann auf verschiedenen Rechnertypen interpretiert werden
- Änderung/Erstellung von Programmcode zur Laufzeit

Vom Programm zur Maschine

# Die Software-/Hardware-Hierarchie

Compiler vs. Interpretierer

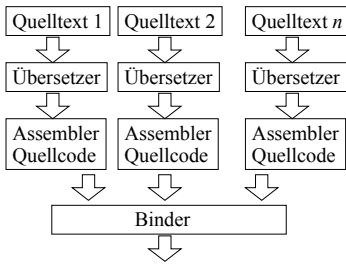
Es gibt auch Crosscompiler/Crossassembler:

Programm, das auf Maschine A ausgeführt wird und Programme für Maschine B erzeugt

- z.B. für:
- Rechner, die zu klein sind für eigene Compiler
  - Rechner, die (noch) keinen eigenen Compiler haben

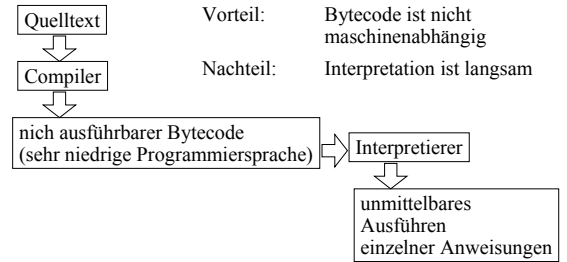
# Die Software-/Hardware-Hierarchie

Binder: Programme können aus mehreren Quelltexten (sogar verschiedener Hochsprachen) bestehen.



# Die Software-/Hardware-Hierarchie

Bytecode (Java), P-Code (UCSD-Pascal)



# Die Software-/Hardware-Hierarchie

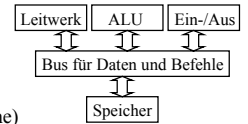
## Warum Maschinenprogrammierung?

- Verständnis für die Funktionsweise von Computern
- Konstruktion von Übersetzern (Compilern) und Interpretieren
- Wartung von immer noch verwendeten großen Programmsystemen, die in Maschinensprache geschrieben sind
- Entwicklung neuer Maschinen/Prozessoren
- Programmierung von Kleinstgeräten, die keine Hochsprache verarbeiten können
- Befriedigung sehr hoher Geschwindigkeitsanforderungen

# Von-Neumann-Rechner

Definition durch J. von Neumann 1946

Ein Rechner besteht aus

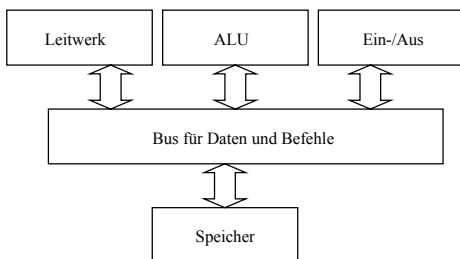


- Speicher (für Daten *und* Programme)
- Steuer- oder Leitwerk (Steuert Vorgänge auf Datenbus)
- Rechenwerk (Arithmetic Logic Unit, ALU)
- Ein-/Ausgabeprozessor (Tastatur, Bildschirm, etc.)
- Datenbus (für Daten und Befehle)

Für damalige Zeit gar nicht selbstverständlich:

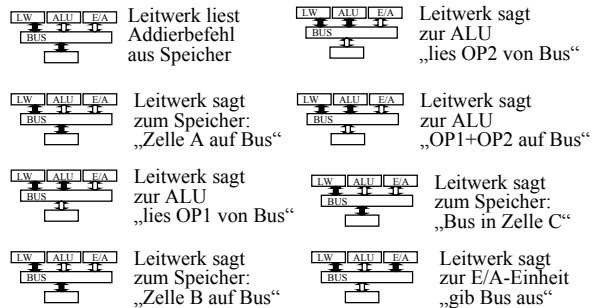
- Daten und Programme im selben Speicher
- Ein zentraler Bus über den alle Teile kommunizieren

# Von-Neumann-Rechner



# Von-Neumann-Rechner

Beispielablauf einer Addition  $C=A+B$



Vom Programm zur Maschine  
**Von-Neumann-Rechner**

Allgemeiner Ablauf einer Rechenoperation:

- Rechenbefehl wird aus Speicher gelesen.
- Die Operanden werden der Reihe nach aus dem Speicher auf den Bus gelegt.
- Die ALU liest die Operanden der Reihe nach vom Bus und speichert sie intern.
- Die ALU wird angewiesen eine Operation mit allen gespeicherten Operanden durchzuführen.
- Die ALU legt das Ergebnis auf den Bus.
- Das Ergebnis wird vom Bus in den Hauptspeicher übertragen oder vom Ein-/Ausgabeprozessor verarbeitet.

Vom Programm zur Maschine  
**Von-Neumann-Rechner**

Arbeit eines Von-Neumann-Rechners als Algorithmus:

```

PC = 0;           // Befehlszähler initialisieren
do
{
    B = Speicher(PC); // Hole nächsten Maschinenbefehl
    führe B aus;
    PC++;
}
while (B!=STOP);
    
```

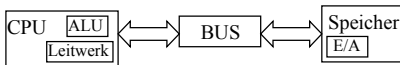
Vom Programm zur Maschine  
**Von-Neumann-Rechner**

In der Praxis:

Mehrere Bestandteile eines Von-Neumann-Rechners in einem Chip (manchmal sogar alle).

Meist: Leitwerk und Rechenwerk sind eine Einheit, die CPU (central processing unit).

Oft: Ein-/Ausgabeprozessor existiert aus der Sicht des Leitwerks nicht, ist Bestandteil des Speichers (Speicherzellen mit besonderer Funktion).



Vom Programm zur Maschine  
**Von-Neumann-Rechner**

Typische Eigenschaften:

- Datenbus transportiert mehrere Bits gleichzeitig ( $2^n$ ).
- Neben den **Datenleitungen** gibt es zusätzliche **Steuerleitungen**, die die gerade transportierten Daten beschreiben.
- Die Zellen des Speichers sind durchnummeriert, beginnend mit 0. Die Nummer einer Zelle wird als **Adresse** bezeichnet
- Es gibt einen **Programmzähler** bzw. Befehlszähler, der die Nummer der Speicherzelle mit dem nächsten Befehl enthält.
- In einer Speicherzelle sind mehrere Bits gespeichert, der Inhalt heißt **Wort**. Verschiedene Wortlängen sind üblich ( $2^n$ ).

Vom Programm zur Maschine  
**Von-Neumann-Rechner**

Ein Typisches Maschinenprogramm:

1000 A9 02  
 1002 C7 1005  
 1004 F0  
 1005 0B

```

CPU: 00 = ...
     ...
     0B = illegal
     ...
     A9 = lade nachfolgendes Byte in ALU
     ...
     C7 = addiere den Inhalt der Speicher-
           zelle deren Nummer folgt
     ...
     F0 = beende Programm
     ...
    
```

Vom Programm zur Maschine  
**Maschinenbefehle**

- **Sprungbefehle:**
  - unbedingte (Befehlszähler wird immer verändert)
  - bedingte (Sprung nur, wenn eine Bedingung erfüllt ist)
- **Rechenoperationen:**
  - arithmetische (+, -, \*, /)
  - logische (¬, ∧, ∨)
  - shifts (<<, >>)  $x \ll n = x \cdot 2^n$
- **Datentransport:**
  - Kopiere Speicherzelle von nach (auch Blöcke)
- **Sonstige:**
  - Flags setzen/löschen (z.B. Übertrag)
  - Direktiven (Trap, Unterbrechung auslösen)

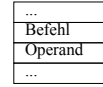
# Typische CPU

- Befehlszähler (Adresse des nächsten Befehls)
- Register (feste Zahl in der CPU integrierter Speicherzellen) für sehr schnelle Verarbeitung, evtl. unterschiedliche Größen
- Stapelzeiger (Spezielles Register mit eigenen Befehlen zur Implementierung eines Kellers (*stack pointer*))
- Flags (einzelne Bits, die bestimmte Zustände anzeigen)
- Unterbrechungsmechanismus (Hardwaresignal erzwingt unbedingten Sprung an bestimmte Adresse)

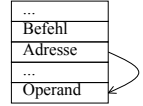
# Speicherzugriffsarten

Verschiedene Operanden von Maschinenbefehlen:

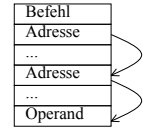
- direkt (Operand folgt Befehl)



- absolut (Adresse des Operanden folgt Befehl)



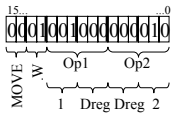
- indirekt (Adresse der Adresse folgt)



- indiziert (Adresse folgt, addiere Register)

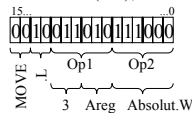
# Speicherzugriffsarten

MOVE.W D1,D2



Bewege ein Wort von Datenregister 1 ins Datenregister 2 (Operanden implizit im 16-bit-Opcode enthalten)

MOVE.L (A3),\$F1E0

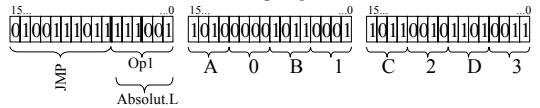


Bewege ein Langwort von dort, wo Adressregister 3 hinzeigt zur explizit angegebenen Adresse \$F1E0.

# Speicherzugriffsarten

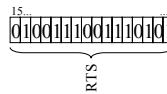
JMP \$A0B1C2D3

Springe sofort zur Adresse A0B1C2D3



RTS

Springe zurück aus Unterprogramm

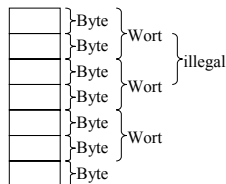


# Speicherzugriffsarten

Verschiedene Ausrichtungen von Operanden

- Operanden können verschieden groß sein (Bytes = 8 Bits, Worte = z.B. 16 Bits, Langworte = 32 Bits)

- Speicherarchitektur kann so angelegt sein, daß längere Operanden nur an bestimmten Speicheradressen stehen dürfen

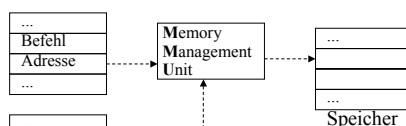


# Speicherzugriffsarten

Virtueller Speicher

oft Unterscheidung **virtueller** Speicher und **physikalischer** Speicher,

Umgebungsvariable definiert Abbildung von virtuell nach physikalisch, (eine Umgebung je Benutzer oder je Prozeß)



Vorteile:

- Positionsunabhängigkeit
- Konfliktvermeidung

# Vom Programm zur Maschine Codierung von Zahlen

Was muß alles codiert werden?

- natürliche Zahlen (klar 2er-System)
- ganze Zahlen: 1-er Komplement,  $-n = 11...1 - n$   
2-er Komplement  $-n = 11...1 - n + 1$
- Wahrheitswerte z.B. true = 11..1, false = 00..0
- Zeichen (Character) ASCII, ISO, und andere
- Fixpunkt-Zahlen  $10010.11 + 00110.01 = 11001.00$
- Gleitkommazahlen  $\text{Mantisse} \cdot 2^{\text{Exponent}}$  z.B.  $0.\underbrace{1101}_{4\text{-bit-Mantisse}} \cdot 10^{\underbrace{10}_{3\text{-bit-Exponent}}}$

Kommt alles nochmal im Detail in Vorlesung TI.

# Vom Programm zur Maschine Codierung von Zahlen

Darstellung ganzer Zahlen im Format:

Vorzeichen/Betrag	Einer-Komplement	Zweier-Komplement
...	...	...
0011 3	0011 3	0011 3
0010 2	0010 2	0010 2
0001 1	0001 1	0001 1
0000 0	0000 0	0000 0
1001 -1	1110 -1	1111 -1
1010 -2	1101 -2	1110 -2
1011 -3	1100 -3	1101 -3
...	...	...

# Vom Programm zur Maschine Codierung von Zahlen

Überlauf: Jede begrenzte Zahlendarstellung hat Problem wenn größte darstellbare Zahl überschritten wird.

z.B. 4-bit-natürliche Zahlen:  $1110 + 1 = 1111 + 1 = 0000$

$$\begin{array}{r} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \\ + \boxed{0} \boxed{0} \boxed{0} \boxed{1} \\ \hline = 1 \boxed{0} \boxed{0} \boxed{0} \boxed{0} \end{array}$$

oder 1-er Komplement:

$$0110 + 0011 = 1001$$

(6) (3) (-6)

oder 2-er Komplement:

$$0110 + 0011 = 1001$$

(6) (3) (-7)

nicht mehr darstellbar

# Vom Programm zur Maschine Codierung von Zahlen

Rechnen mit verschiedenen Codes:

Vorzeichen/Betrag: Normale Operation mit Fallunterscheidung

$$\begin{array}{r} \boxed{v} \boxed{0} \boxed{1} \boxed{1} \pm 3(\text{dec}) \\ + \boxed{w} \boxed{0} \boxed{0} \boxed{1} \pm 1(\text{dec}) \end{array}$$

$$= \boxed{0} \boxed{1} \boxed{0} \boxed{0} \quad 4(\text{dec}) \quad v=0, w=0 \quad (\text{alles Positiv, Addition})$$

$$= \boxed{0} \boxed{0} \boxed{1} \boxed{0} \quad 2(\text{dec}) \quad v=0, w=1 \quad (\text{Subtraktion})$$

$$= \boxed{1} \boxed{1} \boxed{0} \boxed{0} \quad -4(\text{dec}) \quad v=1, w=1 \quad (\text{alles Negativ, Addition})$$

$$= \boxed{1} \boxed{0} \boxed{1} \boxed{0} \quad -2(\text{dec}) \quad v=1, w=0 \quad (\text{Subtraktion})$$

# Vom Programm zur Maschine Codierung von Zahlen

Rechnen mit verschiedenen Codes: 1-er-Komplement

$$\begin{array}{r} \boxed{0} \boxed{0} \boxed{1} \boxed{0} +2(\text{dec}) \\ + \boxed{0} \boxed{0} \boxed{1} \boxed{1} +3(\text{dec}) \\ \hline = \boxed{0} \boxed{1} \boxed{0} \boxed{1} \end{array} \quad \begin{array}{r} \boxed{0} \boxed{0} \boxed{1} \boxed{0} +2(\text{dec}) \\ - \boxed{0} \boxed{0} \boxed{1} \boxed{1} +3(\text{dec}) \\ \hline = \boxed{1} \boxed{1} \boxed{1} \boxed{1} \quad 0(\text{dec}) \neq -1(\text{dec}) \end{array}$$

Macht Subtraktion in 1-er Komplement unnötig aufwendig

$$\left\{ \begin{array}{l} - \boxed{0} \boxed{0} \boxed{0} \boxed{1} \\ = \boxed{1} \boxed{1} \boxed{1} \boxed{0} \quad -1(\text{dec}) \end{array} \right.$$

# Vom Programm zur Maschine Codierung von Zahlen

Rechnen mit verschiedenen Codes: 2-er-Komplement

$$\begin{array}{r} \boxed{0} \boxed{0} \boxed{1} \boxed{0} +2(\text{dec}) \\ + \boxed{0} \boxed{0} \boxed{1} \boxed{1} +3(\text{dec}) \\ \hline = \boxed{0} \boxed{1} \boxed{0} \boxed{1} \end{array} \quad \begin{array}{r} \boxed{0} \boxed{0} \boxed{1} \boxed{0} +2(\text{dec}) \\ - \boxed{0} \boxed{0} \boxed{1} \boxed{1} +3(\text{dec}) \\ \hline = \boxed{1} \boxed{1} \boxed{1} \boxed{1} \quad -1(\text{dec}) \end{array}$$

Gewöhnliche Subtraktion im 2-er System liefert korrektes Ergebnis im 2-er-Komplement.

=> Standard-Code für negative Zahlen ist 2-er-Komplement

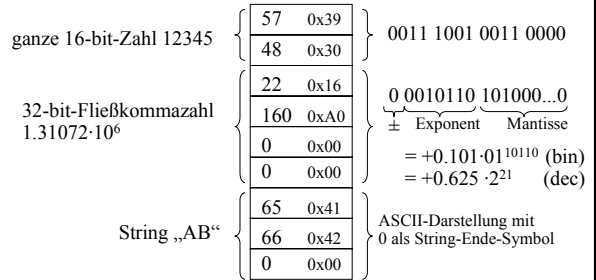
# Arithmetische Operationen

Typische Operationen aus dem Repertoire einer CPU:

- Addition (fast immer vorhanden)
- Subtraktion auch (sonst Komplementierung + Addition)
- Shift-Operationen (1 Bit nach links =  $\cdot 2$ , 1 Bit nach rechts =  $/2$ )
- Multiplikation (oft vorhanden, sonst Shifts + Additionen)
- Division (wenn nicht, dann Shifts + Subtraktionen)
- Rechnen mit Fließkommazahlen mit externem Chip (floating point processor) oder „zu Fuß“.

# Datentypen im Speicher

Elementare Datentypen stehen als Code direkt im Speicher:



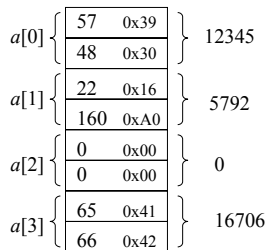
# Datentypen im Speicher

Felder (Arrays)

typischerweise alle Elemente hintereinander im Speicher

Zugriff auf Element  $a[i]$ :

hole Größe eines Elements  
multipliziere mit  $i$   
addiere Adresse von  $a[0]$   
hole Element von dort

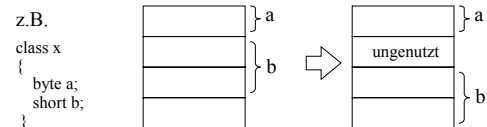


# Datentypen im Speicher

Verbunde (Structs) Zusammenfassungen mehrerer Daten

z.B. die Instanzvariablen eines Objektes bilden zusammengenommen einen Verbund.

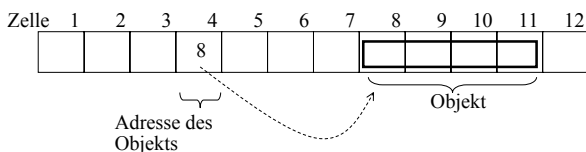
- einzelne Verbundelemente stehen hintereinander im Speicher
- zu Beachten: vom Prozessor geforderte Ausrichtung muß stimmen:



# Datentypen im Speicher

Referenzen (*pointer*)

- jedes Objekt hat eine Adresse (die Adresse der Speicherzelle, ab der es gespeichert ist)
- eine Referenz auf ein Objekt ist einfach dessen Adresse



# Assemblerprogrammierung

Maschinenprogramme werden i.d.R. nicht als Hex-Code eingegeben.

- Maschinenbefehle haben Namen (**Mnemonic**):  
z.B. „goto“ statt 167 bzw. 0xA7
- Adressen können Namen (**Labels**) gegeben werden:  
z.B. „goto Ende“ statt „goto 0x00FE77C“
- Numerische Werte können auf verschiedene Arten angegeben werden:  
z.B. „goto 1234“ oder „goto 1000+0xA\*n“
- Assembler übernimmt **Codierung**:  
z.B. „data 'HELLO WORLD'“ statt „data 72 69 76 76 ...“
- Immer wiederkehrende Konstruktionen können zu einem **Makro** zusammengefaßt werden

# Assemblerprogrammierung

Beispiele für Assemblerprogramme:

Java Bytecode

68000

6502

```

bipush 10
istore_1
goto End
Start:
iinc 1 -1
End:
iload_1
ifgt Start
    
```

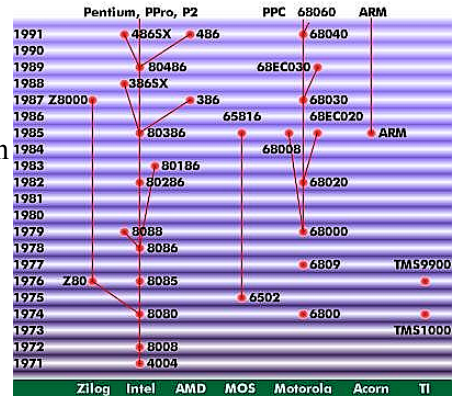
```

move.w #10,D1
Start:
jmp End
add.w #-1,D1
End:
bgt Start
    
```

```

LDA #$0A
Start:
JMP End
ADC #-1
End:
BPL Start
    
```

## Geschichte der Prozessoren



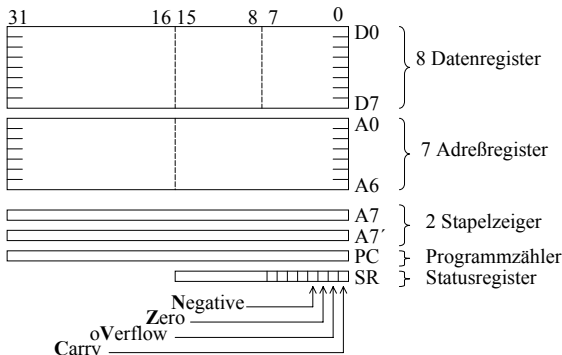
## Einsatzgebiete des 68000

- Apple Macintosh Computer  
Erster weit verbreiteter Computer mit graphischer Oberfläche
- Atari ST/TT/Falcon, Commodore Amiga  
Verbreitete „Home-Computer“ von 1985-1990
- Palm Pilot  
Verbreitetster „Handflächen-Computer“
- Kleingerätesteuerungen  
TV/SAT/Video, Telefone, etc.

## Bestandteile des 68000

- \* CISC - Complex Instruction Set Computer Architektur (nicht RISC)
- \* Acht 32-bit Allzweck-Datenregister (D0-D7).
- \* Acht 32-bit Allzweck-Adressregister (A0-A7).
- \* 32-bit Befehlszähler - 4 Gigabyte linear adressierbarer Speicher
- \* 16-bit Datenbus, 23+1-bit Adressbus (ab 68020 32-bit)
- \* 56 Klassen von Maschinenbefehlen
- \* Ein-/Ausgabe wird in den Speicher abgebildet
- \* 14 verschiedene Adressierungsarten
- \* 5 verschiedene Datentypen (Bit, Byte, BCD, Wort und Langwort)
- \* 2 Betriebsarten: Supervisormodus und Benutzermodus
- \* Unterbrechungsmechanismus (7 Unterbrechungsebenen)
- \* Spätere Versionen beinhalten Memory Management Unit

## Interner Aufbau des Prozessors



## Ein Eindruck vom Befehlssatz des 68000

MOVE von, nach für Datentransport

ADD, SUB, DIVU, DIVS, MULU, MULS,  
für arithmetische Operationen

AND, EOR, NOT, OR, ROL, ROR  
für logische Operationen für Bitrotationen

Bcc, DBcc für bedingte Sprünge  
(mit Registerdekrement)

BRA, JMP, JSR, RTS für unbedingte Sprünge

# Adressierungsarten des 68000

(nicht jeder Befehl versteht alle Adressierungsarten für jeden Operanden)

- move #123,x ; Direkt (die Zahl 123 kommt sofort nach x)
- move x,\$50000 ; Absolut (Langwort)
- move x,\$500.w ; Absolut (Wort)
- move x,D0 ; Datenregister direkt
- move x,A0 ; Adressregister direkt
- move x,(A0) ; Adressregister indirekt
- move x,(A0)+ ; Adressregister indirekt mit Post-Inkrement
- move x,-(A0) ; Adressregister indirekt mit Pre-Dekrement
- move x,\$123(A0) ; Adressregister indirekt mit Offset
- move x,\$12(a0,d0.w) ; Adressregister indirekt mit Offset und Index
- move Offset(PC),x ; Relativ zum PC mit Offset
- move Offset(PC,d0.w),x ; Relativ zum PC mit Offset und Index

# Ein kleines Programmbeispiel

Aufgabe: gegeben: Byte in der Speicheradresse \$1234  
 gesucht: Byte in der Speicheradresse \$3456  
 deren Produkt als Wort in \$5678

Programm: START MOVE.L #0,D0  
 MOVE.B \$1234,D0  
 MOVE.L #0,D1  
 MOVE.B \$3456,D1  
 MULU D0,D1  
 MOVE.W D1,\$5678

# Ein weiteres Programmbeispiel

Aufgabe: gegeben: Wort *n* in Datenregister D6  
 gesucht: Wort *n!* in Datenregister D0

```
START  MOVE.W  #1,D0
        MOVE.W  D6,D1
LOOP   MULU   D1,D0
        CMP    #2,D1
        DBMI   D1,LOOP
```

# Vom Programm zur Maschine Assemblerprogrammierung

Beispiele für Assemblerprogramme:

Java Bytecode	68000	6502
bipush 10 istore_1 goto End	move.w #10,D1  jmp End	LDA #\$0A  JMP End
Start: iinc 1 -1	Start: add.w #-1,D1	Start: ADC #-1
End: iload_1 ifgt Start	End: bgt Start	End: BPL Start

# Vom Programm zur Maschine Assemblerprogrammierung

Zeit in „Nanosekunden“

Laufzeit von Maschinenprogrammen ist sehr genau bestimmbar.

Ausschnitt aus 68000-Dokumentation:

Befehl	Zyklen	Prozessortakt z.B.	50 MHz
add.w	4	=> Ein Taktzyklus =	20 ns
divs	122		
mulu	40	Division mit Vorz.	122 Zyklen
jmp	16		
...		=> Zeit für eine Division	2440 ns

# Vom Programm zur Maschine Ablaufsteuerung

Programmabläufe werden gesteuert durch

- Schleifen (for, while, do-while)
- bedingte Sprünge (if, if-else, switch-case, x ? y : z)
- unbedingte Sprünge (goto, break, continue)
- Aufrufe von Prozeduren / Funktionen
- Ausnahmeunterbrechungen

Diese Mechanismen müssen auch auf der Ebene der Maschinensprache realisiert werden können.

## Vom Programm zur Maschine Ablaufsteuerung

Realisierung von Schleifen

Beispiel in Hochsprache: `c=10; while (c>0) c--;`

Schleife in Assembler (Java Bytecode):

```

bipush 10
istore_1      ; setze Variable 1 auf 10
Loop:
  iload_1     ; hole Wert der Variablen 1
  ifle Done  ; falls der Wert <= 0 springe zu „Done“
  iinc 1 -1  ; addiere zur Variablen 1 den Wert -1
  goto Loop  ; springe zur Marke „Loop“
  
```

Done:

## Vom Programm zur Maschine Ablaufsteuerung

Realisierung von Schleifen

Allgemeine while-Schleife in Hochsprache:

`vorher ; while (Bedingung) {Anweisungen} ; nachher`

Schleife in Assembler:

```

... vorher ...
Schleife:
  ... Werte Bedingung aus
  ... falls Bedingung nicht erfüllt, gehe zu „Ende“
  ... Anweisungen ...
  ... gehe zu „Schleife“
... nachher ...
  
```

## Vom Programm zur Maschine Ablaufsteuerung

Realisierung von Schleifen

Allgemeine while-Schleife in Hochsprache:

`vorher ; while (Bedingung) {Anweisungen} ; nachher`

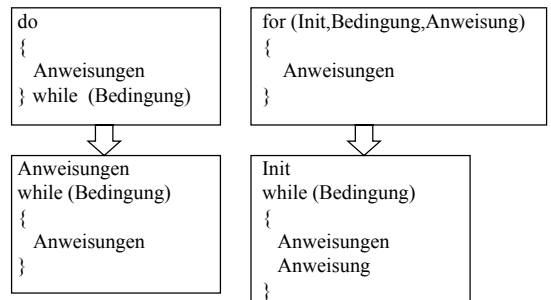
Alternative Schleife in Assembler:

```

... vorher ...
Schleife:
  ... gehe zu „Ende“
  ... Anweisungen ...
Ende:
  ... Werte Bedingung aus
  ... falls Bedingung erfüllt, gehe zu „Schleife“
  ... nachher ...
  
```

## Vom Programm zur Maschine Ablaufsteuerung

i.a. gilt: Jede Schleife läßt sich als while-Schleife darstellen:



## Vom Programm zur Maschine Ablaufsteuerung

Realisierung von Schleifen

Eine Möglichkeit: gegeben Schleife in Hochsprache transformiere Schleife in while-Schleife verwende Standard-Schablone

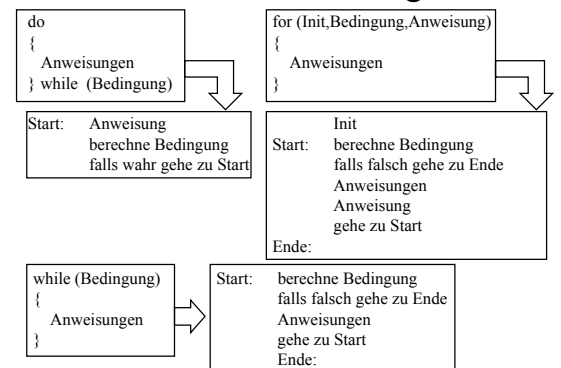
```

      gehe zu „Ende“
Start: ... Anweisungen ...
Ende:  Werte Bedingung aus
      falls Bedingung erfüllt, gehe zu „Start“
  
```

Nachteil: ineffizient für `do ... while`

Andere Möglichkeit: verwende für jede Schleifenform eine spezielle Schablone

## Ablaufsteuerung



## Vom Programm zur Maschine Ablaufsteuerung

### Prozeduren / Funktionen

- Funktionen sind Prozeduren, die einen Rückgabewert haben.
- Manche Programmiersprachen unterscheiden nicht (C: alles Funktionen, Java: alles Methoden)
- Prozeduren haben verschiedene Lokaliätsaspekte:
  - Compilezeit: Sichtbarkeit von Variablen
  - Laufzeit: Lebensdauer lokaler Variablen
- Prozeduren können **geschachtelt** werden
- Prozeduren können sich selbst rekursiv aufrufen

=> optimaler Mechanismus für Prozeduren ist ein **Laufzeitkeller (stack)**

## Vom Programm zur Maschine Ablaufsteuerung

### Prozeduren / Funktionen

- der **Laufzeitkeller** bzw. Stapel (*stack*) ist standardmäßig ein Ausschnitt aus dem Speicher (in dem auch Daten und Befehle sind)
- der Keller kann im Speicher nach oben oder nach unten „wachsen“
- vor Aufruf von Unterprogrammen muß der Aktuelle Befehlszähler gemerkt werden (auf dem Stapel)

Viele Prozessoren haben Befehle JSR (*jump to subroutine*) und RTS (*return from subroutine*)

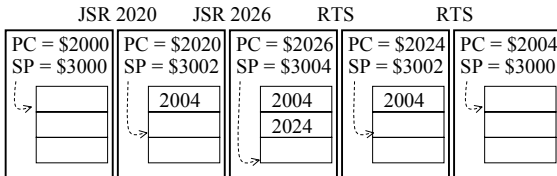
oder ähnliche mit der Funktionalität:

JSR <Adresse> = lege Befehlszähler auf Stapel, gehe zu Adresse  
RTS = hole Adresse von Stapel und setze Befehlszähler auf diesen Wert

## Vom Programm zur Maschine Ablaufsteuerung

Auszug aus Speicher:

2000	JSR 2020
...	
2020	JSR 2026
2024	RTS
2026	RTS



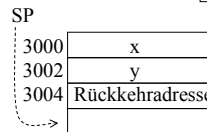
## Vom Programm zur Maschine Ablaufsteuerung

### Prozeduren mit Parametern

Funktions-/Prozedurparameter werden auch auf dem Stapel abgelegt.

Hochsprache:  $f(x,y)$

Maschinensprache: lege x auf Stapel  
lege y auf Stapel  
springe ins Unterprogramm f  
nimm x vom Stapel weg



Im Unterprogramm gilt immer:

- Rückkehradresse steht gleich hinter SP
- Parameter stehen gleich hinter Rückkehradresse

## Vom Programm zur Maschine Ablaufsteuerung

### Funktionen mit Parametern

Auch Ergebnisse von Funktionen werden auf dem Stapel abgelegt.

Hochsprache:  $y=f(x)$

Maschinensprache: lege x auf Stapel  
lege „Dummy“ auf Stapel  
springe in Unterprogramm f  
in f: berechne f(x)  
schreibe Ergebnis in Dummy  
hole Dummy von Stapel nach y  
hole x von Stapel

Hierbei genutzt: Funktionsaufruf  $y=f(x)$  schreibbar als Prozeduraufruf  $f(x,y)$

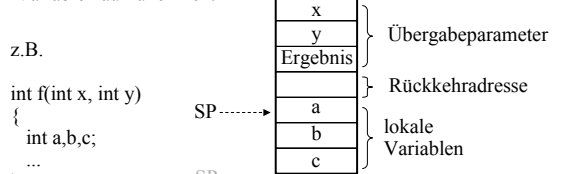
Bemerkung: Statt Platzhalter kann auch eine Referenz (Zeiger auf Speicherzelle für Ergebnis) abgelegt werden.

## Vom Programm zur Maschine Ablaufsteuerung

Verschiedene Parameter-/Funktionswertübergaben sind üblich.

Allgemein: Aufruf eine Prozedur => Anlegen einer „Schachtel“ auf dem Stapel

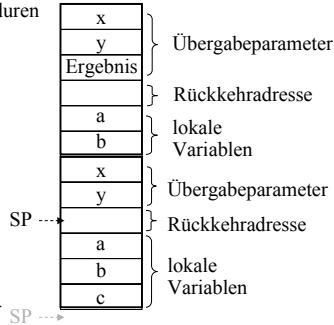
Eine **Prozedurschachtel** ist auch geeignet, lokale (nicht-statische) Variablen aufzunehmen:



## Vom Programm zur Maschine Ablaufsteuerung

Verschachtelung von Prozeduren

Prozeduren/Funktionen  
geschachtelt aufgerufen  
=>  
mehrere Schachteln  
hintereinander auf Stapel



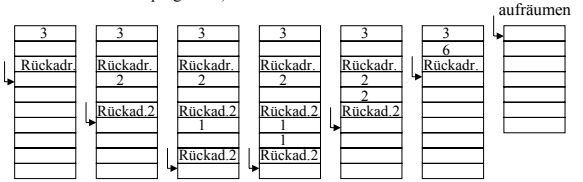
Beachte:  
Daten auf Stapel, die nicht  
Rückkehradressen sind,  
müssen **aufgeräumt** werden.

## Vom Programm zur Maschine Ablaufsteuerung

Ein Beispiel: Berechnung von  $n!$

falls  $n=1$ ,  
dann schreibe 1 in Ergebniszelle  
Rücksprung  
sonst  $n-1$  auf Stapel  
Platz für Ergebnis auf Stapel  
Rückkehradresse auf Stapel  
rekursiver Aufruf

(Rückadr. von Hauptprogramm,  
Rückadr.2 von Unterprogramm)



## Vom Programm zur Maschine Ablaufsteuerung

Ein Beispiel: Berechnung von  $n!$

2000	START	MOVE.W	#3,-(SP)	wir wollen 3!, also 3 auf Stapel
2004		MOVE.W	#0,-(SP)	Platz für 2 Byte auf Stapel
2008		JSR	FACT	
...				
3000	FACT	MOVE.W	6(SP),D1	hole aktuelles $n$ vom Stapel
3004		CMP.W	D1,#1	vergleiche mit 1
3008		BEQ	END	falls ==, dann springe zu END
300C		ADD.W	#-1,D1	berechne $n-1$
3010		MOVE.W	D1,-(SP)	schiebe $n-1$ auf Stapel
3012		MOVE.W	#0,-(SP)	Platz für 2 Byte auf Stapel
3016		JSR	FACT	rekursiver Aufruf mit $n-1$
301A		MOVE.W	(SP)+,D2	hole Ergebnis $(n-1)!$ vom Stapel
301C		MOVE.W	(SP)+,D0	räume Parameter weg
301E		MOVE.W	6(SP),D1	hole nochmal aktuelles $n$
3022		MULU	D2,D1	Multipliziere $(n-1)!$ mit $n$
3024	END	MOVE.W	D1,4(SP)	Ergebnis auf Stapelplatz
3028		RTS		

## Vom Programm zur Maschine Ablaufsteuerung

8000						
7FFC	0003	0003	0003	0003	0003	0003
7FF8	0000	0000	0000	0000	0000	0006
7FF4	200E	200E	200E	200E	200E	200E
7FF0	0000	0000	0000	0000	0000	0000
7FEF	0002	0002	0002	0002	0002	0002
7FEF	0000	0000	0000	0000	0002	0002
7FEF	301A	301A	301A	301A	301A	301A
7FEF	0000	0000	0000	0000	0000	0000
7FEF	0000	0001	0001	0001	0001	0001
7FEF	0000	0001	0001	0001	0001	0001
7FEA	0000	301A	301A	301A	301A	301A
7FE8	0000	0000	0000	0000	0000	0000
7FE6						

1. Mal bei FACT      2. Mal bei FACT      3. Mal bei FACT      1. Mal bei END      2. Mal bei END      3. Mal bei END      zurück aus FACT

## Vom Programm zur Maschine Auswertung von Ausdrücken

Programmablaufsteuerung ist relativ einfach,  
Komplizierter ist Auswertung von Ausdrücken:

z.B.  $a[7] = b - (d - f(3 + c[1][d/2] - g, 2) + ((1 - b++) * (d < b ? 0 : 1 + h, q)))$

- zu beachten:
- Punkt-vor-Strich-Regel (u.a. Prioritätsregeln)
  - Klammerung (evtl. geschachtelt)
  - Aufrufe von Funktionen (evtl. mehrere Argumente)
  - Zugriffe auf Felder (evtl. mehrdimensional)
  - unäre Operatoren (-x)
  - ternäre Operatoren (x?y:z)
  - Zugriffe auf Strukturelemente (x.y)
  - In einigen Sprachen auch Zeiger (x→y)

## Vom Programm zur Maschine Auswertung von Ausdrücken

Eine andere Ausdrucksschreibweise

Wenn wir statt  $x$  Operator  $y$   
schreiben  $x$  y Operator

ist das die sog. **umgekehrte polnische Notation (UPN)**,  
oder auch **Postfix**-Schreibweise (Operator  $x$   $y$  = Präfix-Schreibweise).  
Benannt nach dem polnischen Logiker Jan Lukasiewicz (1929).

Beispiele

Infix:	$a+b+c$	$a+b*c-d/e$	$2*-(a+b)$
Postfix:	$a b + c +$	$a b c * + d e / -$	$2 a b + \sim *$
Präfix:	$+ a + b c$	$- + * b c a / d e$	$* 2 \sim + a b$

Vom Programm zur Maschine

# Auswertung von Ausdrücken

Eine andere Ausdruckschreibweise

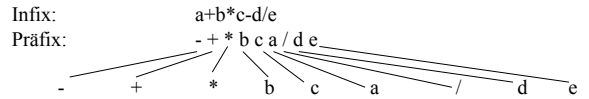
Es gibt unterschiedlichstellige Operatoren:

	Infix	Präfix	Postfix
unäre:	-x	~ x	x ~
binäre:	x+y	+ x y	x y +
ternäre:	x?y:z	? x y z	x y z ?

Vom Programm zur Maschine

# Auswertung von Ausdrücken

Interpretation der Präfix-Schreibweise



Differenz ( Summe ( Produkt ( b und c ) und a ) und Quotient ( d und e ) )

Vom Programm zur Maschine

# Auswertung von Ausdrücken

Interpretation der Postfix-Schreibweise (UPN)

Infix:  $a+b*c-d/e$   
 Postfix:  $a b c * + d e / -$

von b und c das Produkt  
 von a und (von b und c das Produkt) die Summe  
 von (von a und (von b und c das Produkt) die Summe) und (von d ...

# Auswertung von Ausdrücken

Auswertung eines Ausdrucks in Postfix-Schreibweise (UPN)

Ausdruck:  $a b c * + d e / -$

- Vorgehen:
- a a = 1. Operand
  - b b = 1. Operand, a = 2. Operand
  - c c = 1. Operand, b = 2. Operand, a = 3. Operand
  - \* berechne Produkte der beiden neuesten Operanden  $b*c = 1$ . Operand, a = 2. Operand
  - + berechne Summe der beiden neuesten Operanden  $a+b*c = 1$ . Operand
  - d d = 1. Operand,  $a+b*c = 2$ . Operand
  - e e = 1. Operand, d = 2. Operand,  $a+b*c = 3$ . Operand
  - / berechne Quotient der beiden neuesten Operanden  $d/e = 1$ . Operand,  $a+b*c = 2$ . Operand
  - berechne Differenz der beiden neuesten Operanden  $a+b*c-d/e = 1$ . Operand

Vom Programm zur Maschine

# Auswertung von Ausdrücken

Auswertung eines Ausdrucks in Postfix-Schreibweise (UPN)

Ausdruck:  $a b c * + d e / -$

Auswertung mit Keller:

a	b	c	*	+	d	e	/	-
a	b	c	$b*c$	$a+b*c$	d	e	d/e	$a+b*c-d/e$
	a	b	a		$a+b*c$	d	$a+b*c$	
		a				$a+b*c$		

Vom Programm zur Maschine

# Auswertung von Ausdrücken

Auswertung eines Ausdrucks in Postfix-Schreibweise (UPN)

Allgemeiner Algorithmus für einfache Ausdrücke:

Solange Ausdruck nicht abgearbeitet  
 hole nächstes Element E des Ausdrucks  
 falls E eine Variable oder Konstante  
 schiebe Wert von E auf Keller  
 falls E ein n-stelliger Operator  
 ersetze die Kellerelemente  $k_1, \dots, k_n$  durch  $E(k_1, \dots, k_n)$

## Auswertung von Ausdrücken

Beachtung von Prioritäten

Umwandlung von Infix nach Postfix erfolgt in der Reihenfolge der Prioritäten der Operatoren; z.B.:  $x+2y^3+z-4$

Infix:  $x + 2 * y ^ 3 + z * 4$   
 $\Rightarrow x + 2 * \{y 3 ^\} + z * 4$   
 $\Rightarrow x + \{2 \{y 3 ^\} *\} + z * 4$   
 $\Rightarrow x + \{2 \{y 3 ^\} *\} + \{z 4 *\}$   
 $\Rightarrow x + \{\{2 \{y 3 ^\} *\} \{z 4 *\} +\}$   
 $\Rightarrow \{x \{\{2 \{y 3 ^\} *\} \{z 4 *\} +\} +\}$

Also Postfixdarstellung =  $x 2 y 3 ^ * z 4 * + +$

## Auswertung von Ausdrücken

Bearbeitung von Klammern in Ausdrücken

z.B.:  $2*(x+3*(y+z))$

Vorgehensweise: Auflösung von innen nach außen

Solange Klammern vorhanden

- finde innersten geklammerten Ausdruck, hier „(y+z)“
- ersetze ihn durch seine Postfixdarstellung

Im Beispiel:  $2*(x+3*(y+z))$   
 $\Rightarrow 2*(x+3*\{y z +\})$   
 $\Rightarrow 2*\{x 3 \{y z +\} * +\}$   
 $\Rightarrow 2 \{x 3 \{y z +\} * +\} *$

also

$2 x 3 y z + * + *$

## Auswertung von Ausdrücken

Bearbeitung von Funktionen/Feldern in Ausdrücken

z.B.:  $x+2*f(y,z)$

Wir betrachten f als 2-stelligen Operator höchster Priorität, dann haben wir

Infix:  $x+2* y f z$

Postfix:  $x 2 y z f * +$

$n$ -dimensionale Felder können als Funktionen mit  $n$  Parametern betrachtet werden.

## Auswertung von Ausdrücken

Bearbeitung von Verbundelementen

z.B.  $3*x.im+y.re$  (evtl. komplexe Zahlen als Objekte)

Wir betrachten  $x$  und  $y$  als einstellige Funktionen die nur die Argumente im oder re erlauben. Dann ergibt sich:

Infix:  $3*x.im+y.re$   
 $\Rightarrow 3*\{im x\} + \{re y\}$   
 $\Rightarrow \{3 \{im x\} *\} + \{re y\}$   
 $\Rightarrow \{3 \{im x\} *\} \{re y\} +$

In der Praxis wird statt der Namen der Verbundelemente („im“ oder „re“) deren Index (1 oder 2) benutzt.

Postfix:  $3 im x * re y +$

## Funktionsweise von Compilern

Ein Compiler besteht aus verschiedenen Teilen:

- lexikalische Analyse: Erkennen von definierten Zeichenfolgen (z.B.: „class“, „for“, „while“, ...)
- syntaktische Analyse: Erkennen von Fehlern in der Syntax (z.B.: „for (i=1:);“, „while { a=2; }“ ...)
- semantische Analyse: Verstehen, was das Programm tun soll.

Syntax = Regeln wie Symbolfolgen aussehen dürfen

Semantik = Bedeutung von Symbolfolgen

## Funktionsweise von Compilern

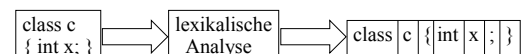
Lexikalische Analyse:

class	16
for	17
...	
while	39

Compiler hat ein Lexikon mit Einträgen, z.B.:

benutzerdefinierte Symbolfolgen

(z.B. Variablenamen) werden dynamisch dazugenommen.



# Funktionsweise von Compilern

Syntaktische (und teilweise semantische) Analyse wird von einem Zerteiler (*parser*) durchgeführt

Regeln für Parser:

IF = if (<Bedingung>) <Anweisung>  
 oder if (<Bedingung>) <Anweisung> else <Anweisung>

WHILE = while (<Bedingung>) <Anweisung>

<Anweisung> = <einfache Anweisung>  
 oder {<einfache Anweisung>...}

Ein Parser versucht alle Regeln seines Regelkatalogs anzuwenden und ersetzt schrittweise Symbolfolgen.

# Funktionsweise von Compilern

Beispiel für einen Parse-Vorgang:

Regeln: A = A+A    Regel 1  
 A = (A)    Regel 2    A = Nichtterminal  
 A = 1    Regel 3    1 = Terminal

Programm: (1+(1+1))

Parser macht: (1+(1+1))  
 Regel 3 => (A+(1+1))  
 Regel 3 => (A+(A+1))  
 Regel 3 => (A+(A+A))  
 Regel 1 => (A+(A))  
 Regel 2 => (A+A)  
 Regel 1 => (A)  
 Regel 2 => A

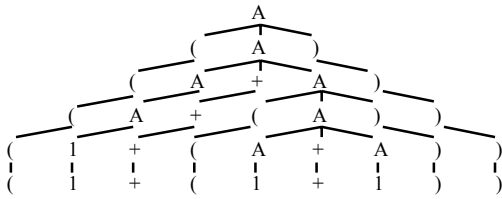
Die Liste der angewendeten Regeln spiegelt die Semantik eines Programmes wider.

# Funktionsweise von Compilern

Ein Parsvorgang ist erfolgreich abgeschlossen, wenn er in einem „akzeptierenden“ Terminal endet.

Dann ist die Syntax des Programmes korrekt.

Die Angewendeten Regeln eines Parsevorganges können als Parsebaum angegeben werden:



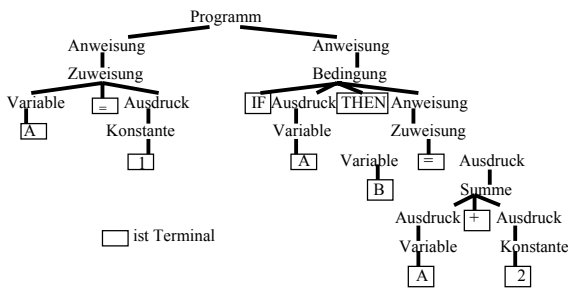
# Funktionsweise von Compilern

- Programm : Anweisung Anweisung ...
- Anweisung : Zuweisung | Bedingung
- Zuweisung : Variable = Ausdruck
- Bedingung : IF Ausdruck THEN Anweisung
- Variable : A | B | C | ...
- Ausdruck : Summe | Variable | Konstante
- Konstante : 0 | 1 | 2 | ...
- Summe : Ausdruck + Ausdruck

Ein Beispielprogramm: A=1  
 IF A THEN B=A+2

# Funktionsweise von Compilern

Parsebaum für: A=1  
 IF A THEN B=A+2



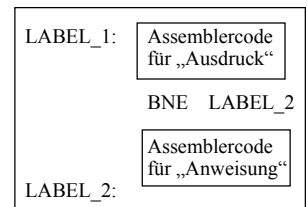
□ ist Terminal

# Funktionsweise von Compilern

Compiler-Aktionen

z.B.  $\overbrace{\text{IF}[\text{Ausdruck}][\text{THEN}]\text{Anweisung}}^{\text{Bedingung}}$

erzeugt Assemblercode



# Funktionsweise von Compilern

Compiler-Aktionen z.B. für



erzeugt Assemblercode

```
LABEL_1: Assemblercode
           für 1. Ausdruck
           Assemblercode
           für 2. Ausdruck
           POP R1
           POP R2
           ADD R1 R2
           PUSH R2
```

# Funktionsweise von Compilern

Compiler-Aktionen z.B. für

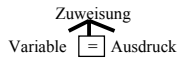


erzeugt Assemblercode

```
LABEL_1: MOVE #2,R1
           PUSH R1
```

# Funktionsweise von Compilern

Compiler-Aktionen z.B. für



erzeugt Assemblercode

```
LABEL_1: Assemblercode
           für „Ausdruck“
           POP R1
           MOVE R1,Variable
           ...
Variable: Platz für Variable
```