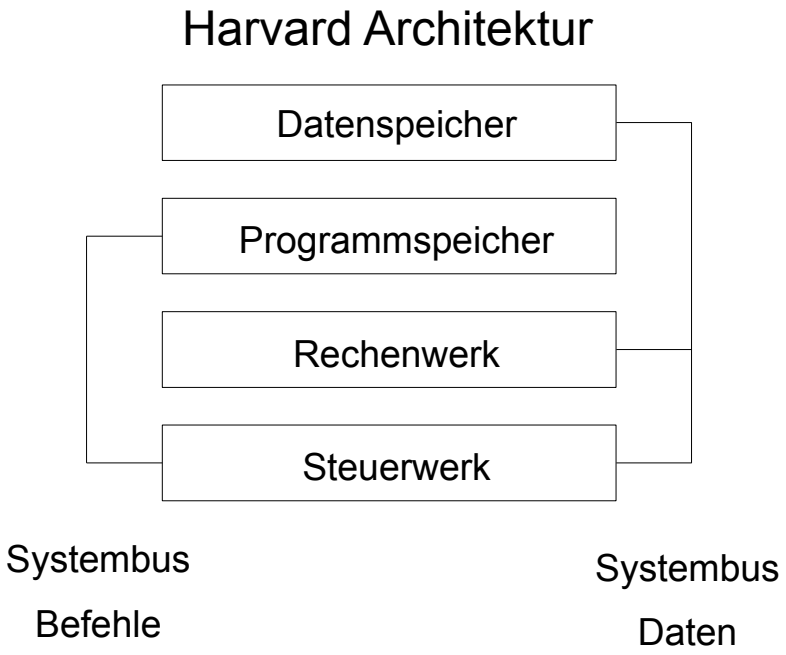
Kapitel 5

Grundlagen der Assemblerprogrammierung

Speicherbenutzung

- **Datenspeicher** muss ein RAM sein.
  - dient zur Speicherung von **variablen** Daten
    - Daten müssen also geschrieben werden können.
    - z.B Ergebnisse von Anweisungen
  - enthält nach dem Einschalten keine gültigen Daten
    - Konstante Werte können also nicht im Datenspeicher stehen.
- **Programmspeicher** kann ein ROM oder ein RAM sein.
  - Enthält die Befehle des Maschinenprogramms.
  - Es existiert keine Notwendigkeit, den Programmspeicher dynamisch zu verändern.
    - Das Maschinenprogramm ist beim Einschalten im Datenspeicher vorhanden (ROM), oder
    - es wird über einen nicht weiter betrachteten Weg in den Datenspeicher geladen und dann nicht mehr verändert.
- **Steuerwerk** liest die Befehle aus dem Programmspeicher und steuert ihre Ausführung.



- Z.B. beinhaltet die **C-Anweisung** *a = b + c* den **Befehl** *ErgOp = QuellOp1 + QuellOp2* und die **Variablen** *a, b und c*.
  - Der **Befehl** steht in codierter Form im Programmspeicher.
  - Die **Werte der Variablen** stehen im Datenspeicher.
  - Im Befehl sind **Adressen** enthalten, die auf die Datenspeicherstellen zeigen, an denen die Variablenwerte stehen.

Maschinenbefehle

- Um ein Programm ausführen zu können, muss es in Form des **Maschinenprogramms** im Programmspeicher stehen.
- Ein Maschinenprogramm besteht aus einer Folge von **Maschinenbefehlen**, die hintereinander im Programmspeicher stehen.
- Ein Maschinenbefehl ist die codierte Form eines Befehls, der den Prozessor anweist, eine bestimmte Operation auszuführen.
- Ein Maschinenbefehl besteht aus dem sog. Operationscode (**OpCode**) und ggf. der Spezifikation von zugehörigen **Operanden**.
  - Semantisch sind OpCodes an den Prozessor gerichtete Kommandos, z.B.: addiere, kopiere, springe, ...
  - Ein OpCode ist ein Binärwert, der bei den meisten Prozessoren 8 Bits lang ist.
- Der OpCode und die Operandenspezifikationen eines Maschinenbefehls stehen in aufeinanderfolgenden Bytes im Programmspeicher.
  - Die Operandenspezifikationen bestimmen die Daten, die bei der durchzuführenden Operation zu verwenden sind.
    - Bezeichnet ein Operand eine **Variable**, steht im Maschinenbefehl die **Datenspeicheradresse**, an der der Wert der Variablen abgespeichert ist.
    - Bezeichnet ein Operand den Wert einer **Konstanten**, steht im Maschinenbefehl direkt der **Wert der Konstanten**.
    - z.B.: C-Anweisung **a = 7** → Fiktiver Maschinenbefehl **44** **35** **7** (35 : Adresse der Variablen a)
      (Semantik des Maschinenbefehls **44** : Kopiere 1 Byte, den konstanten Wert Op2 an die Adresse von Op1).
- Der OpCode definiert also, wie die Bytes hinter dem OpCode zu interpretieren sind, d.h.
  - aus wie vielen Bytes der Maschinenbefehl besteht,
  - ob es überhaupt Operanden gibt, und wenn ja, wie viele und von welchem Typ sie sind.

Maschinenprogrammarchitekturen

- Im allgemeinen Fall benötigt eine arithmetische oder logische Funktion drei Operanden, z.B : **a = b + c**
- Alternative 1 : Drei-Adress-Maschine : **OpCode ErgAdr, QuellOp1, QuellOp2**
  - Konsequenz : lange Befehle
  - Wird praktisch nicht gebraucht und wird deshalb selten verwendet.
- Alternative 2 : Zwei-Adress-Maschine : **OpCode ErgAdr, QuellOp2** ErgAdr ist gleichzeitig Quelladr1
  - Darstellung in C : Op1 = Op1 + Op2
  - Der Ergebniswert überschreibt den Wert eines Operanden.
  - Wenn der alte Wert von Op1 noch gebraucht wird, muss zuvor eine Kopie davon erstellt werden
  - Wird häufig verwendet in mächtigeren Prozessoren (z.B. PC).
- Alternative 3 : Ein-Adress-Maschine : **OpCode QuellOp2**
  - Ein spezielles Register im Steuerwerk, der sog. **Akkumulator**, wird verwendet, um den ersten Operanden (erster Quelloperand und Ergebnisoperand) für die Verknüpfungsoperation temporär aufzunehmen.
  - Eine Sequenz von Maschinenbefehlen ist nötig für die o.g. C-Anweisung : Op1 = Op1 + Op2

```
Move    (Op1) → (Akkumulator)
Add     (Op2)                        (Ergebnis im Akkumulator)
Move    (Akkumulator) → (Op1)
```

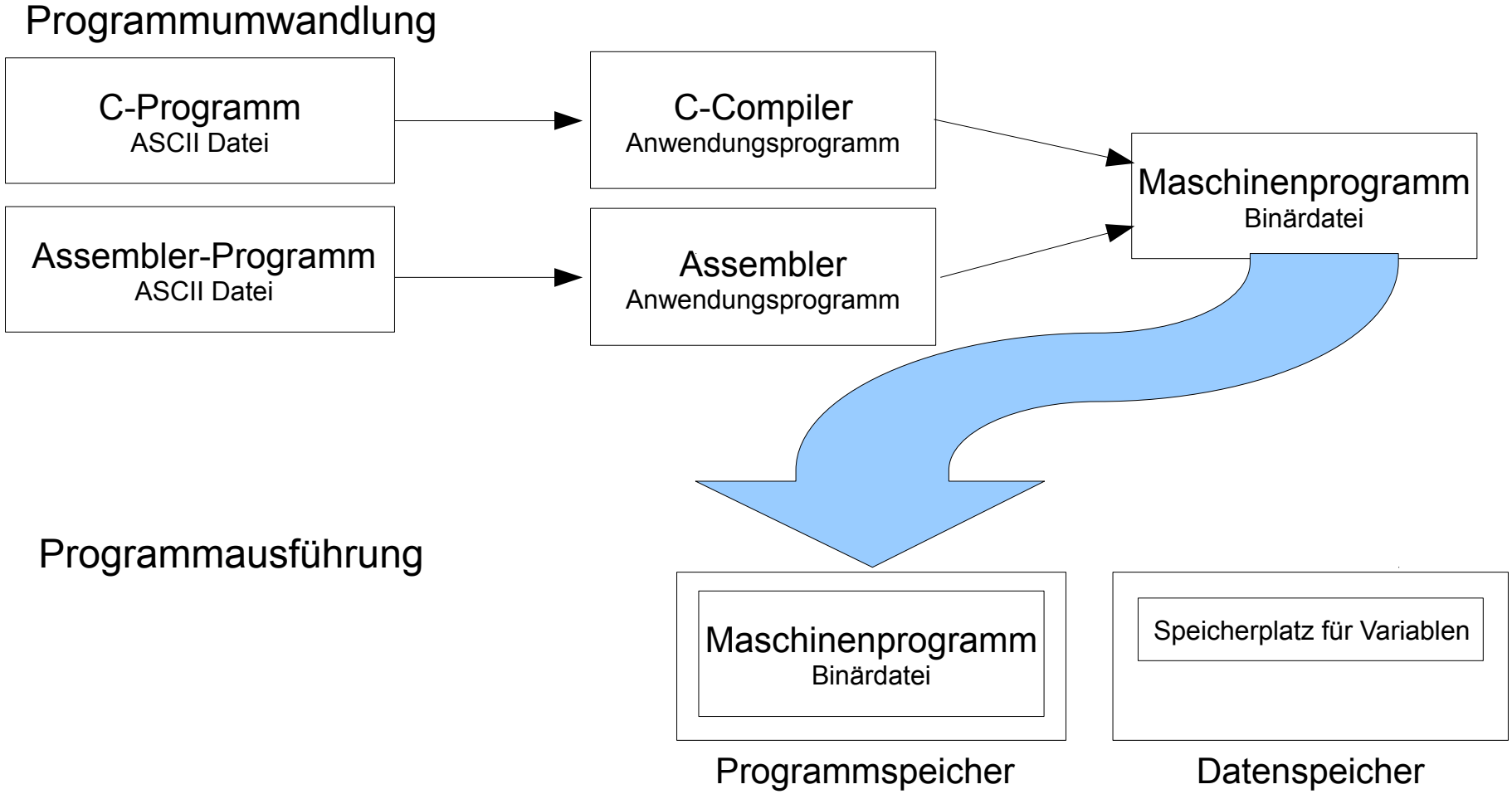
  - Wird häufig in einfacheren Prozessoren verwendet, (z.B Mikrocontroller).
  - 8051 ist eigentlich eine Ein-Adress-Maschine (besitzt einen Akkumulator), stellt aber auch einige Maschinenbefehle mit zwei Adressen zur Verfügung.

Datenspeicher-Realisierungen

- Der in den Architekturen eingeführte Datenspeicher stellt alle Komponenten eines Rechners dar, die variable Daten speichern, die **direkt** vom Programm angesprochen werden können.
- Die wichtigste Datenspeicherkomponente ist der Hauptspeicher - ein separater Baustein oder ein im Prozessor integriertes Speicherarray.
- Vor allen bei großen Systemen ist der Hauptspeicher in DRAM Technologie ausgeführt.
  - DRAMs sind im Vergleich zur Technologie des Prozessors langsam.
- Andererseits beobachtet man, dass Zugriffe zu den selben Daten häufig zeitlich und räumlich begrenzt sind.
  - Z.B. ein Datenelement wird in einem kleinen Programmbereich gleich ein paar Mal angesprochen, im restlichen Programm dann nicht mehr.
- Zur Performanceverbesserung hat man deshalb Komponenten in schnellerer Technologie hinzu gefügt.
  - **Caches**
    - Transparent für das Programm werden kleine Blöcke, sog. Cache Lines (z.B. 64 Bytes) in den Cache kopiert in der Hoffnung, dass noch auf andere Bytes der Cache Line kurzfristig zugegriffen wird.
  - **General Purpose Register**
    - Eine relativ kleine Anzahl (PowerPC : 32, 8051 : 8) von Hardware Registern, die vom Programm über sehr kurze Registeradressen angesprochen werden können.
    - Es ist die Aufgabe des Compilers, die Benutzung der Register zu optimieren.
- Für eine einfachere Struktur gibt es in kleineren Prozessoren den **Akkumulator**
    - Ein Hardware Register im Prozessor.
    - Der OpCode jedes Maschinenbefehls gibt an, wann und wie der Akkumulator zu verwenden ist.

Load/Store-Architektur

- Wird häufig benutzt bei Prozessoren mit langsamem (großem) Hauptspeicher
  - Solche Prozessoren besitzen üblicherweise eine größere Anzahl von Registern (oder einen kleinen, schnellen lokalen Datenspeicher)
- Befehle, die Daten manipulieren, benutzen nur Werte in Registern.
- Für Hauptspeicherzugriffe stehen nur Transportbefehle zwischen dem Hauptspeicher und einem Register zur Verfügung
  - **LOAD** : Daten aus dem Speicher zum Register übertragen
  - **STORE** : Registerinhalt in den Speicher übertragen
- Prozessoren ohne Load/Store Architektur
  - z.B.: x86 Familie, 8051 ohne externen Datenspeicher
- Prozessoren mit Load/Store Architektur
  - z.B.: PowerPC, ARM, 8051 mit externem Datenspeicher (MOVX Befehl zum Transport eines Bytes zwischen Akkumulator und externem Datenspeicher)



- Das Maschinenprogramm
  - wird z.B. durch das Betriebssystem in den Programmspeicher geladen, oder
  - wird in ein ROM gebrannt, das als Programmspeicher eingesetzt wird.
- Der Speicherplatz für Variable hat zum Programmstart einen undefinierten Inhalt.

Beispiel Compilierung (0)

Programm

C-Anweisungen

Symboltabelle

Typ	Name	Adresse/ Wert

Pseudocode fiktiver Befehle

(x) = Inhalt von x

Maschinenprogramm

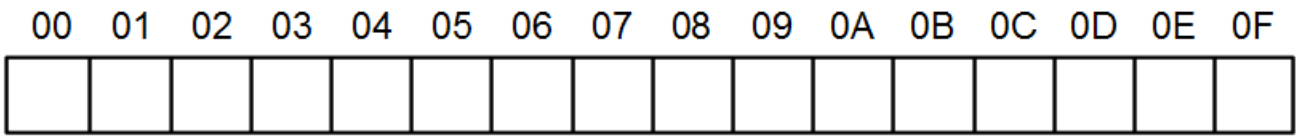
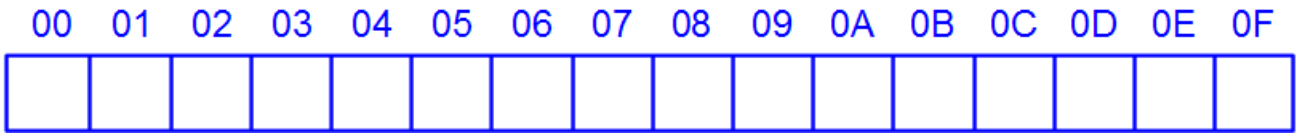


Bild des Datenspeichers (nur zur Veranschaulichung)



Beispiel Compilierung (1)

Programm

C-Anweisungen
int Var1 ;
char Va2 ;
int Var3 ;
char Va4 ;

Symboltabelle

Typ	Name	Adresse/ Wert
V2	Var1	0
V1	Va2	2
V2	Var3	3
V1	Va4	5

Pseudocode fiktiver Befehle

(x) = Inhalt von x

Maschinenprogramm

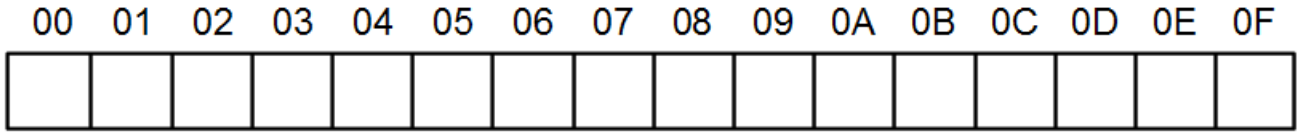
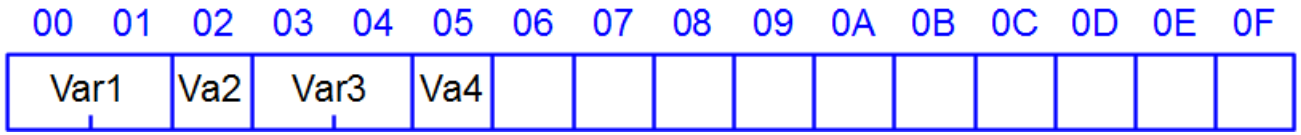


Bild des Datenspeichers (nur zur Veranschaulichung)



Beispiel Compilierung (2)

Programm

C-Anweisungen
int Var1 ;
char Va2 ;
int Var3 ;
char Va4 ;
Va2 = 7;
Va4 = Va2 ;
Var1 = Var3 ;

Symboltabelle

Typ	Name	Adresse/ Wert
V2	Var1	0
V1	Va2	2
V2	Var3	3
V1	Va4	5

Pseudocode fiktiver Befehle

OpCode 75 : (Op1) ← Op2

OpCode A5 : (Op1) ← (Op2)

(x) = Inhalt von x

Maschinenprogramm

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
75	02	07	A5	05	02	A5	00	03	A5	01	04				

Bild des Datenspeichers (nur zur Veranschaulichung)

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
Var1		Va2	Var3		Va4										

Beispiel Compilierung (3)

Programm

C-Anweisungen
:
Var1 = Var3 ;
char i;
for (i=1; i<=3; i++)
{Va2 = Va2 + Va4} ;

Symboltabelle

Typ	Name	Wert/ Adresse
V2	Var1	00
V1	Va2	02
V2	Var3	03
V1	Va4	05
V1	i	06

Pseudocode fiktiver Befehle

OpCode 75 : (Op1) ← Op2

OpCode A5 : (Op1) ← (Op2)

OpCode 66 : springe nach Op3, wenn (Op1) > Op2

OpCode 77 : (Op1) = (Op1) + (Op2)

OpCode 88 : (Op1) = (Op1) + Op2

OpCode 99 : springe nach Op1

Maschinenprogramm

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11
75	02	07	A5	05	02	A5	00	03	A5	01	04	75	06	01	66	06	03

12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	....
1B	77	02	05	88	06	01	99	0F						

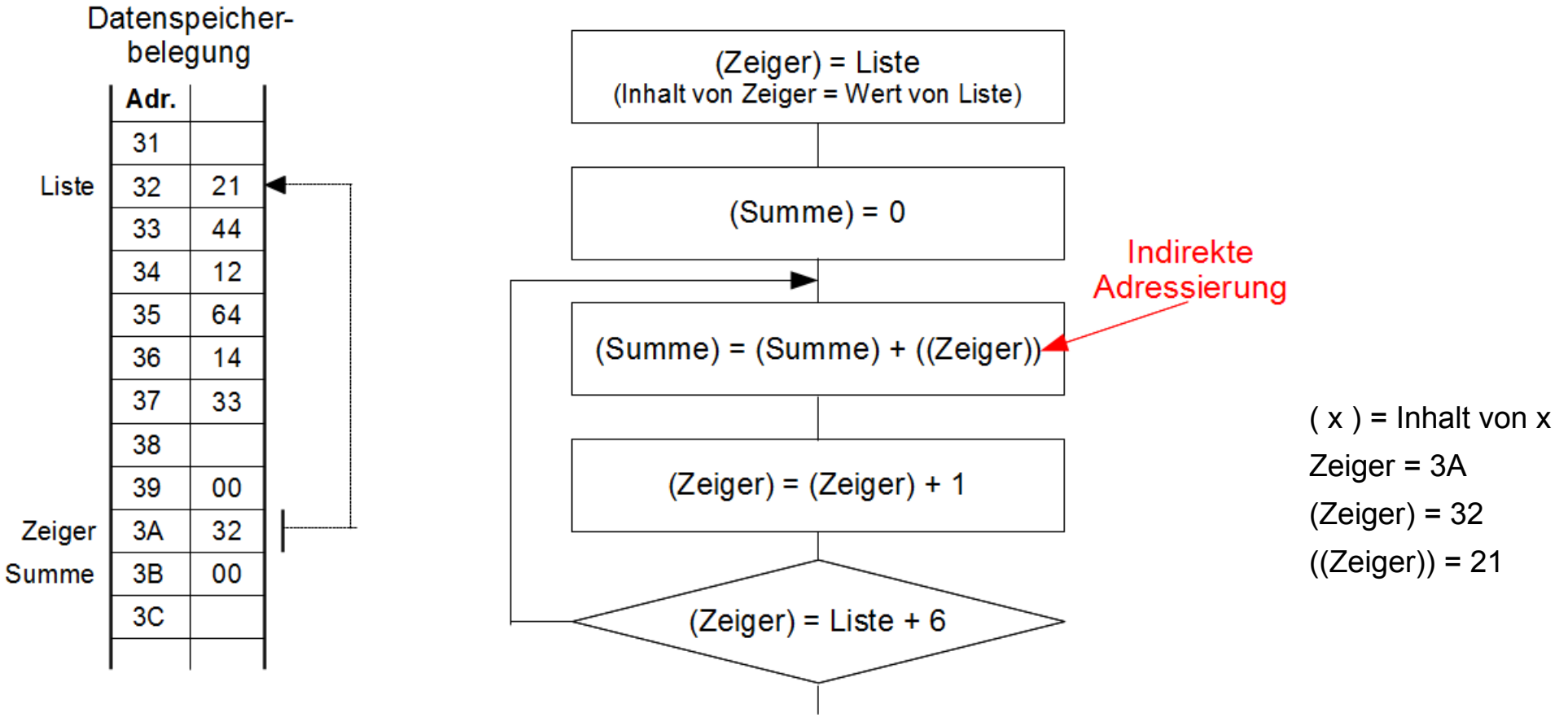
Bild des Datenspeichers (nur zur Veranschaulichung)

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
Var1		Va2	Var3		Va4	i									

Indirekte Adressierung

- **Direkte Adressierung** : Die Adresse in einem Befehl zeigt auf den Wert einer Variablen im Datenspeicher, der bei der Ausführung des Befehls verwendet wird.
- **Indirekte Adressierung** : Die Adresse in einem Befehl zeigt auf einen Wert im Datenspeicher, der eine Adresse darstellt. Diese Adresse zeigt auf den Wert einer Variablen im Datenspeicher, der bei der Ausführung des Befehls verwendet wird.

Beispiel : Summe einer Liste von 1-Byte Werten



Beispiel Compilierung (4)

Programm

C-Anweisungen

```
      :
char Sum ;
char* Ptr;
char List[3] = (10, 20,30);
Ptr = &List;
Sum = *Ptr;
```

Symboltabelle

Typ	Name	Wert/ Adresse
V1	Sum	0
V1	Ptr	1
V3	List	2

Pseudocode fiktiver Befehle

OpCode 75 : (Op1) ← Op2

OpCode AA : (Op1) ← ((Op2))

(x) = Inhalt von x

Maschinenprogramm

20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
75	02	0A	75	03	14	75	04	1E	75	01	02	AA	00	01	

Bild des Datenspeichers (nur zur Veranschaulichung)

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
Sum	Ptr	List													

Benutzung von Operanden

- Im obigen Beispiel 'Summe von Listenwerten' wird
  - der Inhalt der Variablen **Summe** auf den **konstanten Anfangswert 0** gesetzt,
  - der Inhalt der Variablen **Zeiger** auf die Adresse des ersten Wertes von Liste gesetzt.
    - Die **Adresse des ersten Wertes von Liste** ist ebenfalls **konstant**.
  - **Für beide Operationen** ist also der fiktive **OpCode 44** [(Op1) ← Op2] zu verwenden.
    - (Summe) ← 0 : 44 00 00; (Ptr) = Liste : 44 01 02

==> Die Adresse eines Operanden in einem Maschinenbefehl ist eine Konstante.

- In den vorangegangenen Beispielen habe wir drei Typen von Transportbefehlen kennengelernt.
  - OpCode 44 : (Op1) ← Op2 z.B.: 44 01 02 (Ptr) ← Liste
  - OpCode A5 : (Op1) ← (Op2) z.B.: A5 10 00 (Ergebnis) ← (Summe)
  - OpCode AA : (Op1) ← ((Op2)) z.B.: AA 11 01 (Listenwert) ← ((Ptr))
- In allen Fällen sind Op1 und Op2 in den Maschinenbefehlen als Adressen spezifiziert und somit konstant.
- Die Information, ob die Adresse selbst, der direkte Inhalt an dieser Speicheradresse oder der an dieser Adresse indirekt adressierte Wert zu verwenden ist, steckt im OpCode.

==> Die Adressspezifikationen der Operanden in Maschinenbefehlen sind konstant.

- Der Programmspeicher wird bekanntlich nicht verändert.
- Alle variablen Daten befinden sich im Datenspeicher und werden über ihre Adressen angesprochen.

Vom C-Programm zum Maschinenprogramm

- Bei der Compilierungsvorgang erzeugt der Compiler
  - eine Binärdatei (= ein Bytestring), in der das Maschinenprogramm zusammengebaut wird,
  - die **Symboltabelle**, für die Verwaltung der im Programm benutzten Namen.
  - In der Symboltabelle werden registriert :
    - die Variablennamen und ihre zugeordneten Speicheradressen,
    - die Namen der Konstanten und die dafür spezifizierten Werte.
  - Nach Beendigung des Compiliervorgangs wird die Symboltabelle nicht meht benötigt.
    - Die zu den Namen gehörigen Adressen stehen vollständig im Maschinenprogramm.
- Der Compiler prozessiert in der C-Quelldatei eine Anweisung nach der anderen.
- Im Fall der Deklaration einer Variablen
  - bestimmt er den benötigten Speicherplatz aus dem Datentyp,
  - reserviert er diesen Platz im Datenspeicherbereich,
  - registriert er den Variablennamen und die zugehörige Adresse in der Symboltabelle.
- Im Fall der Deklaration einer Konstanten
  - registriert er den Namen, die Größe und den Wert der Konstanten in der Symboltabelle.

Vom C-Programm zum Maschinenprogramm (Forts.)

- Im Fall einer Anweisung
  - fügt er die der Anweisung entsprechenden Sequenz der Maschinenbefehle in die Maschinenprogrammdatei ein.
  - mit der Referenz einer Variablen liest er die für die Variable in der Symboltabelle registrierte **Adresse** aus und fügt diese in den/die Maschinenbefehl/e ein.
  - mit der Referenz einer Konstanten liest er den für die Konstante in der Symboltabelle registrierten **Wert** aus und fügt diesen in den/die Maschinenbefehl/e ein.
- Benötigte Operandentypen :
  - Konstanter Operand
    - Konstanter Wert
    - Adressspezifikation (Adresse im Datenspeicher)
    - Zieladresse eines Sprungbefehls (Adresse im Programmspeicher)
  - Variabler Operand
    - Direkt adressierte Variable
    - Indirekt adressierte Variable

Assemblersprache und -befehle

- Mit Maschinenbefehlen (HEX-Werte) können Menschen schlecht umgehen.
- Deshalb hat man zur einfacheren Bearbeitung die Assemblersprache eingeführt.
- Jeder **Assemblerbefehl** in einem Assemblerprogramm entspricht **genau einem** Maschinenbefehl im zugehörigen Maschinenprogramm.
- Ein Assemblerprogramm wird mit Hilfe des **Assemblers** (ein Anwendungsprogramm wie der Compiler) in ein Maschinenprogramm umgewandelt.
  - Das Quellprogramm ist eine reguläre ASCII-Datei.
  - Das Maschinenprogramm (Objektcode) ist eine Binärdatei.
- Der Bitkombination des OpCodes im Maschinenbefehl entspricht im Assemblerbefehl ein aussagekräftiger **mnemonischer Ausdruck** :

  z.B. : ADD = Additionsoperation
  MOV = Transportoperation
  JMP = Sprungbefehl

  - Speicheradressen und konstante Werte können als symbolische Namen spezifiziert werden.
- Für die Behandlung von symbolischen Namen benutzt der Assembler eine Symboltabelle (wie auch der Compiler).
- Außer den Assemblerbefehlen finden sich im Quellprogramm noch **Pseudobefehle** und **Assemblerdirektiven**
  - Pseudobefehle sind Anweisungen an den Assembler, die zwar keine Maschinenbefehle erzeugen, aber Einfluss auf die Maschinenbefehle haben, z.B. Adress-Spezifikationen.
  - Assemblerdirektiven spezifizieren Optionen des Assembliervorgangs.

Assemblerbefehle

- Für Assemblerbefehle ist ein starres Format vorgeschrieben :

Marke:	***OpCode***	***Operand1,Operand2***	***;Kommentar***
BEGIN:	*MOV*	*27,28*	*;Dieses Beispiel : 8051 Assemblerbefehle*
	AND	*33,#0xFE*	
	JMP	*BEGIN*	

  - Ein Befehl pro Zeile
  - Die Marke (optional) muss in der ersten Position der Zeile beginnen.
  - Jeder Befehl hat einen OpCode.
  - Der OpCode bestimmt, ob und wie viele Operanden vorhanden sind.
  - Im Fall von mehreren Operanden ist der erste Operand der Ergebnisoperand.
  - Semikolon (beim 8051 Assembler) signalisiert den Anfang von Kommentar, der vom Assembler ignoriert wird.

- Neben dem Objektcode erzeugt der Assembler auch eine Programmliste, der die Assemblerbefehle zusammen mit dem dazu gehörigen Maschinenbefehlen zeigt :

Zeilen-Nr.	**Programm-Speicher-Adresse**	**Maschinen-Befehl**	**Marke:**	**OpCode**	**Op1,Op2**	**;Kommentar**
.....						
0123	*03A4*	*A51B1C*	*BEGIN:*	*MOV*	*27,28*	*;Kopiere Inhalt von 28 nach 27*
0124	*03A7*	*5421FE*		*AND*	*33,#0xFE*	
0127	*03AA*	*020123*		*JMP*	*BEGIN*	
.....						

Operandentypen

- Bei Maschinenbefehlen steckt die Verwendung der **Operanden** des Befehls im OpCode.
  - Im 8051 Maschinenbefehl 'A51B1C' spezifiziert der OpCode 'A5', dass sowohl der Operand 1 '1B' als auch der Operand 2 '1C' eine **direkte Adresse** darstellen.
  - Im 8051 Maschinenbefehl '5421FE' spezifiziert der OpCode '54', dass der Operand 1 '21' eine **direkte Adresse** und der Operand 2 'FE' einen **unmittelbaren Wert (Immediate Byte)** darstellen.
  - Im 8051 Maschinenbefehl '020123' spezifiziert der OpCode '02', dass der Operand 1 '0123' eine zwei Byte große Programmspeicheradresse darstellt, nämlich das Ziel eines Sprungbefehls.
- In Assemblersprachen werden häufig generische OpCodes verwendet. Die Verwendung der Operanden wird dann durch ein zusätzliches Zeichen bei der Operandenspezifikation definiert.
  - In 8051 bezeichnet
    - ein vorangestelltes **#-Zeichen** einen unmittelbaren Wert, z.B #0xFE,
    - ein vorangestelltes **@-Zeichen** eine indirekte Adressangabe, z.B. @R0,
    - eine Operandenangabe **ohne zusätzliches Zeichen** eine direkte Adressangabe, z.B. 27.
  - Sprungziele werden praktisch immer als Symbole spezifiziert.

8051 Operandentypen

- **Immediate-Wert:** Der Operand im Befehl ist der unmittelbare Wert (Immediate Byte). Der Wert steht also wie der ganze Befehl im Programmspeicher. (Skript 3.3.1)

  8051 Beispiel : ***MOV 0x27,#0x33*** → *Maschinenbefehl : 0x752733*

  OpCode : Bits 23 .. 16
  Operand1 : Bits 15 ... 8 (Datenspeicheradresse)
  Operand 2 : Bits 7 ... 0 (Immediate-Wert)

  Das **#**-Zeichen spezifiziert einen konstanten Wert.

- **Speicherreferenz** : Der Operand im Befehl ist eine Datenspeicheradresse. Der **Inhalt** dieser Speicherstelle wird zur Ausführung der Operation verwendet.
  (siehe auch Skript 3.3.2)

  8051 Beispiel : ***MOV 0x27,0x28*** → *Maschinenbefehl : 0xA52728*

  OpCode : Bits 23 .. 16
  Operand1 : Bits 15 ... 8 (Datenspeicheradresse)
  Operand 2 : Bits 7 ... 0

  Speicherreferenzen zum internen 8051 Datenspeicher sind 8 Bits groß. Somit lassen sich 2^8 = 256 Bytes adressieren.

- **Registerreferenz** : Der Operand im Befehl ist eine Registeradresse. Der Inhalt dieses Registers wird zur Ausführung der Operation verwendet. (Skript 3.3.3)

  8051 Beispiel : ***INC R3*** → *Maschinenbefehl : 0x2B = 0010 1011B*

  Im OpCode enthalten : Bits 2 ... 0 (Registeradresse)

  Beim 8051 Assembler ist die Registerspezifikation Teil des OpCodes.
  Es gibt 8 Register - deshalb werden drei Bits für die Adressierung benötigt.

8051 Operandentypen (Forts.)

- **Indirekte Speicherreferenz :** Der Operand im Befehl ist eine Datenspeicheradresse. Der Inhalt dieser Datenspeicheradresse ist wieder eine Datenspeicheradresse. Der Inhalt dieser Speicherstelle wird zur Ausführung der Operation verwendet. (Skript 3.3.5)

  Im 8051 Prozessor nicht implementiert.

- **Register-Indirekte Speicherreferenz :** Der Operand im Befehl ist eine Registeradresse. Der Inhalt dieses Registers ist eine Datenspeicheradresse. Der Inhalt dieser Datenspeicherstelle wird zur Ausführung der Operation verwendet. (Skript 3.3.4)

  8051 Beispiel : ***INC @R1*** → *Maschinenbefehl : 0x07 = 0010 0111*B

  Im OpCode enthalten : Bit 0 (Registeradresse)

  Das **@**-Zeichen spezifiziert eine register-indirekte Speicherreferenz.

  Wenn z.B. im Register 1 der Wert 27 steht, wird der Wert in der Datenspeicherstelle 27 inkrementiert.

  Es gibt 2 Register, die zur indirekten Adressierung verwendet werden können - deshalb wird ein Bit für die Adressierung benötigt.

8051 Operandentypen (Forts.)

- **Bitreferenz :** Der Operand adressiert ein einzelnes Bit im internen 8051 Datenspeicher. (Skript 3.3.1)

  8051 Beispiel : ***SETB 0x33*** → *Maschinenbefehl : 0xD233*

  OpCode : Bits 15 ... 8
  Operand1 : Bits 7 ... 0 (Bitadresse)

  Der 8051 Prozessor besitzt 256 direkt adressierbare Bits.

- **Sprungadresse :** Die Sprungadresse ist eine Befehlsadresse im **Programmspeicher**. Der Operand in einem Sprungbefehl ist das Ziel für den Sprungbefehl.

  1. *8051 Beispiel :* ***LJMP*** *0x2728* → *Maschinenbefehl : 0x022728*

     OpCode : Bits 23 ... 16
     Operand : Bits 15 ... 0 (absolute Zieladresse)

  2. 8051 Beispiel : ***SJMP*** *0x27* → *Maschinenbefehl : 0x8027*

     OpCode : Bits 15 ... 8
     Operand : Bits 7 ... 0 (relative Zieladresse) (Details im Kapitel 8051 Befehle)

Verwendung von Symbolen

- Wie zuvor gezeigt, können Datenspeicheradressen, Programmspeicheradressen (Sprungziele), und Werte von Konstanten direkt durch Zahlen ausgedrückt werden.
  - Ändert sich ein solcher Wert, müssen alle Stellen im Programm, an denen dieser Wert verwendet wird, manuell angepasst werden.
  - Deshalb hat man auch im Assembler die Verwendung von Symbolen eingeführt. Bei Änderung der Adresse oder des Wertes wird die Änderung automatisch bei allen Referenzen wirksam.
- Die Verwendung von Symbolen im Compiler und im Assembler ist ziemlich ähnlich.
  - Auch der Assembler benutzt eine Symboltabelle für die Verknüpfung der symbolischen Namen mit ihren tatsächlichen Adressen bzw. Werten.

Symbole als Datenspeicheradressen

- Ein Symbol für eine Speicheradresse muss zunächst deklariert werden (mit Hilfe des Pseudobefehls DS) und kann dann beliebig im Programm referiert werden.
- Der Assembler reserviert bei der Deklaration den notwendigen Speicherplatzund macht wie beim Compiler einen entsprechenden Eintrag in der Symboltabelle.
- Bei der Referenz eines Symbols ermittelt der Assembler in der Symboltabelle die zugewiesene Adresse und setzt diese in den Maschinenbefehl ein.

Symbole als Programmspeicheradressen

- Wenn ein Symbol vor einem Assemblerbefehl steht, weist der Assembler dem Symbol die Programmspeicheradresse des Assemblerbefehls zu und macht einen entsprechenden Eintrag in der Symboltabelle.
- Erscheint ein solches Symbol in einem Sprungbefehl, setzt der Assembler dafür die Adresse des Zielbefehls ein.

Symbole von Datenspeicheradressen in Assembler

Programm

C-Anweisungen
int Var1 ;
char Va2 ;
int Var3 ;
char Va4 ;
Va2 = 7;
Va4 = Va2 ;
Var1 = Var3 ;

Programm

Assembler Befehle	
Var1:	DS 2
Va2:	DS 1
Var3:	DS 2
Va4:	DS 1
	MOV Va2,#7
	MOV Va4,Va2
	MOV Var1,Var3
	MOV Var1+1,Var3+1

Symboltabelle

Typ	Name	Adresse/ Wert
V2	Var1	0
V1	Va2	2
V2	Var3	3
V1	Va4	5

Maschinenprogramm

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
75	02	07	A5	05	02	A5	00	03	A5	01	04				

Bild des Datenspeichers (nur zur Veranschaulichung)

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
Var1		Va2	Var3			Va4									

Symbole von Programmspeicheradressen in Assembler

Programm

C-Anweisungen

```
        :
char i;
for (i=1; i<=3; i++)
{Va2 = Va2 + Va4} ;
```

Programm

Assembler Befehle

```
        :
i:      DS  1
        MOV  i,#1
Loop:   MOV  A,Va2
        ADD  A,Va4
        MOV  Va2,A
        INC  i
        CJNE i,#4,Loop
```

Symboltabelle

Typ	Name	Wert/ Adresse
V2	Var1	00
V1	Va2	02
V2	Var3	03
V1	Va4	05
V1	i	06

Maschinenprogramm

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11
75	02	07	A5	05	02	A5	00	03	A5	01	04	75	06	01	77	02	05

12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	....
88	06	01	67	06	04	0F								

Bild des Datenspeichers (nur zur Veranschaulichung)

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	....
Var1		Va2	Var3		Va4	i										

Indirekte Adressierung in Assembler

Programm

C-Anweisungen

```
      :
char Sum ;
char* Ptr;
char List[3] = (10, 20,30);
Ptr = &List;
Sum = *Ptr;
```

Programm

Assembler Befehle

```
Sum:    DS  1
R0:     DS  1
List:   DS  3
        MOV   List,#10
        MOV   List+1,#20
        MOV   List+2,#30
        MOV   R0,#List
        MOV   Sum,@R0
```

Symboltabelle

Typ	Name	Wert/ Adresse
V1	Sum	0
V1	R0	1
V3	List	2

Maschinenprogramm

20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
75	02	0A	75	03	14	75	04	1E	75	01	02	AA	00	01	

Bild des Datenspeichers (nur zur Veranschaulichung)

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
Sum	R0	List													

Adresse oder Wert einer Variablen

- Angenommen eine Variable im Datenspeicher sei deklariert als : ***Var1 DS 1***
- Der Befehl ***MOV R0,Var1*** lädt den **Wert** der durch Var1 adressierten Speicherstelle in R0.
- Der Befehl ***MOV R0,#Var1*** lädt die **Adresse von Var1** in R0.
  - Bei der Deklaration von Var1 hat der Assembler die zugewiesene Datenspeicheradresse, die konstant ist, in der Symboltabelle abgelegt.
  - Auf diese Weise kann eine Datenspeicheradresse als Variable (für die **indirekte Adressierung)** verfügbar gemacht werden. Im Prinzip handelt es sich um den konstanten Anfangswert einer Variablen.

Symbole für Konstanten

- Wenn ein konstanter Wert mehrfach im Programm gebraucht wird (z.B. der Zinssatz), ist es sinnvoll, dafür ein Symbol einzuführen.
  - Bei einer Änderung muss nur der konstante Wert in der Symboldefinition geändert werden.
  - Bei jedem Aufruf des Symbols wird automatisch der neue Wert verwendet.
- In C kann ein Symbol für eine Konstante deklariert werden mit z.B. *const int ConstSymb = 7 ;*
  - Der Compiler erstellt dafür lediglich einen Eintrag in der Symboltabelle.
  - Bei der Referenz des Symbols ermittelt der Compiler den Wert der Konstanten aus der Symboltabelle und setzt diesen in den Maschinenbefehl ein. Z.B.: *a = b + ConstSymb*.
- In Assembler kann ein Symbol für eine Konstante definiert werden mit z.B. *ConstSymb EQU 7*.
  - Bei jedem Auftreten des Symbols ersetzt es der Assembler durch den zugewiesenen konstanten Wert. z.B. : *MOV R0,#ConstSymb*.

Programmablaufsteuerung (Control Flow) und Sprungbefehle

- Bei der Compilierung platziert der Compiler Maschinenbefehle direkt hinter einander im Programmspeicher.
- Jeder Maschinenbefehl ist über seine **Befehlsadresse** ansprechbar.
  - Die Befehlsadresse ist die Byteadresse des OpCodes im Programmspeicher.
- Bei sequentiellen C-Anweisungen platziert der Compiler die entsprechenden Maschinenbefehle in sequentieller Weise im Programmspeicher.
- Nicht-sequentielle Konstruktionen wie Verzweigungen und Schleifen werden im Maschinenprogramm durch die **Sprungbefehle** realisiert.
- Es gibt **bedingte** und **unbedingte** Sprungbefehle.
  - Der unbedingte Sprungbefehl
    - hat als Operand eine Befehlsadresse,
    - gibt an, dass die Programmausführung mit dem spezifizierten Befehl fortzusetzen ist.
  - Der bedingte Sprungbefehl
    - spezifiziert neben der neuen Befehlsadresse noch einen Bedingung
      - z.B. Wert einer Variablen = 0, Wert1 > Wert2, ….
    - gibt an, dass die Programmausführung mit dem
      - spezifizierten Befehl fortzusetzen ist, **wenn die Bedingung erfüllt ist**.
      - nächsten sequentiellen Befehl fortzusetzen ist, **wenn die Bedingung nicht erfüllt ist**.

Vom Assembler-Programm zum Maschinenprogramm

- Bei der Assemblierung erzeugt der Assembler
  - die Binärdatei, in der das Maschinenprogramm zusammengebaut wird,
  - die **Symboltabelle**, für die Verwaltung der im Programm benutzten Namen.
- Der Assembler prozessiert in der Qelldatei eine Zeile nach der anderen :
- Im Fall der Deklaration einer Variablen
  - ist der benötigte Speicherplatz explizit angegeben,
  - reserviert er diesen Platz hinter dem zuletzt reservierten Bereich,
  - registriert er den Variablennamen und die zugehörige Adresse in der Symboltabelle.
- Im Fall der Deklaration einer Konstanten
  - registriert er den Namen, die Größe und den Wert der Konstanten in der Symboltabelle.
- Im Fall eines Assemblerbefehls
  - fügt er den entsprechenden Maschinenbefehl in die Maschinenprogrammdatei ein,
  - mit der Referenz einer Variablen liest er die für die Variable in der Symboltabelle registrierte **Adresse** aus und fügt diese in den/die Maschinenbefehl/e ein.
  - mit der Referenz einer Konstanten liest er den für die Konstante in der Symboltabelle registrierten **Wert** aus und fügt diesen in den/die Maschinenbefehl/e ein.

Assembliervorgang

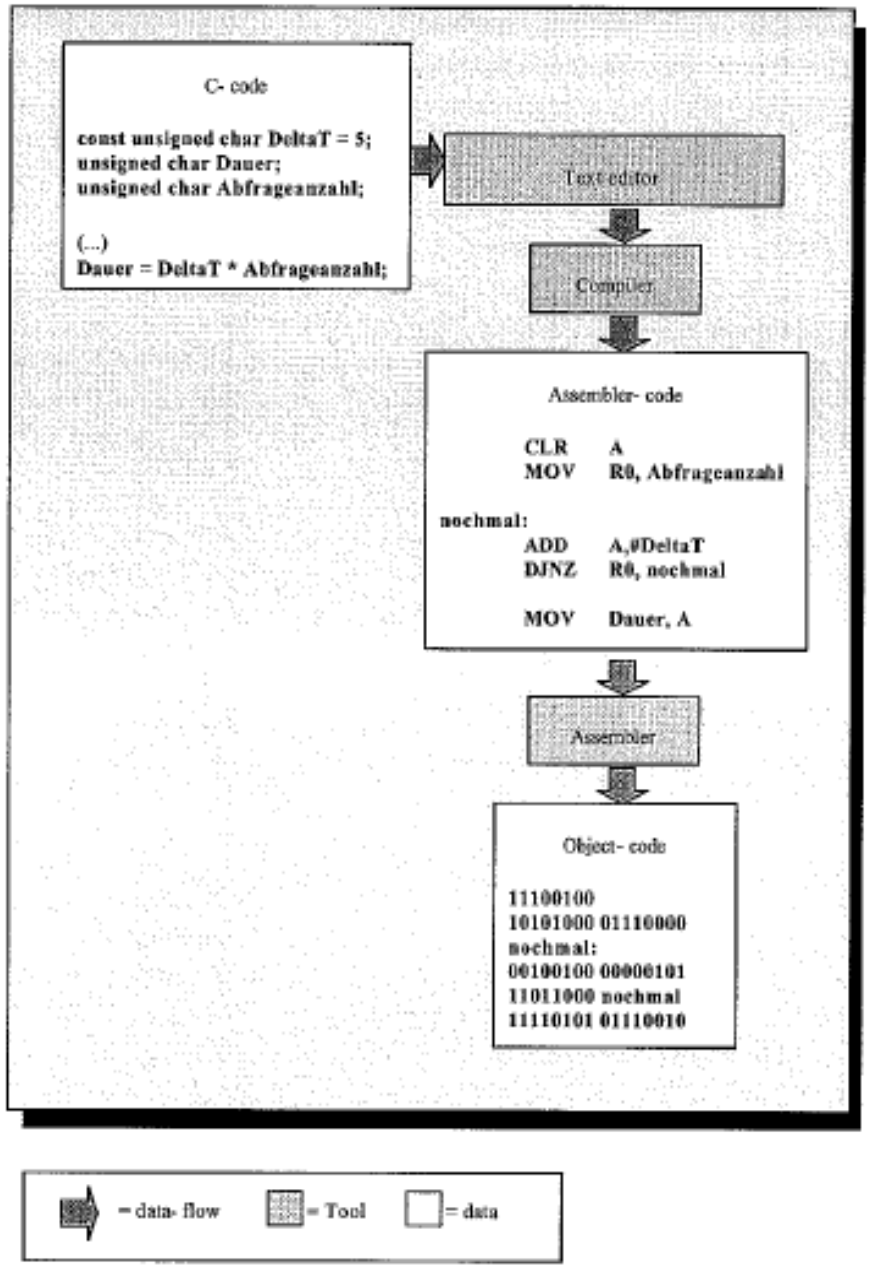


Bild 5: Entwicklungswerkzeuge zur Erzeugung des binären Opcodes

Beispiel : Bild 5 im Skript Seite 11

- C-Anweisungen :

```
const unsigned char DeltaT = 5 ;
unsigned char Dauer ;
unsigned char Abfrageanzahl ;
Dauer = DeltaT * Abfrageanzahl ;
```

- Assemblerprogramm :

```
            CLR     A                   ;Akku = 0
            MOV     R0,Abfrageanzahl    ;Lade Loop Counter
nochmal:
            ADD     A,#DeltaT           ;Addiere konstanten Wert
            DJNZ    R0,nochmal          ;1. dekrementiere R0
                                        ;2. springe, wenn R0 ≠ 0
            MOV     Dauer,A             ;Ergebnis → Variable Dauer
```

- Die Deklarationen von Abfrageanzahl, DeltaT und Dauer

```
Abfrageanzahl:  DS      1               ;Reserviere 1 By
Dauer:          DS      1               ;Reserviere 1 By

DeltaT          EQU     5               ;Weise den konstanten Wert 5 zu
```