

Vorlesungsgliederung

□ Einführung

- Motivation, Historische Anmerkungen

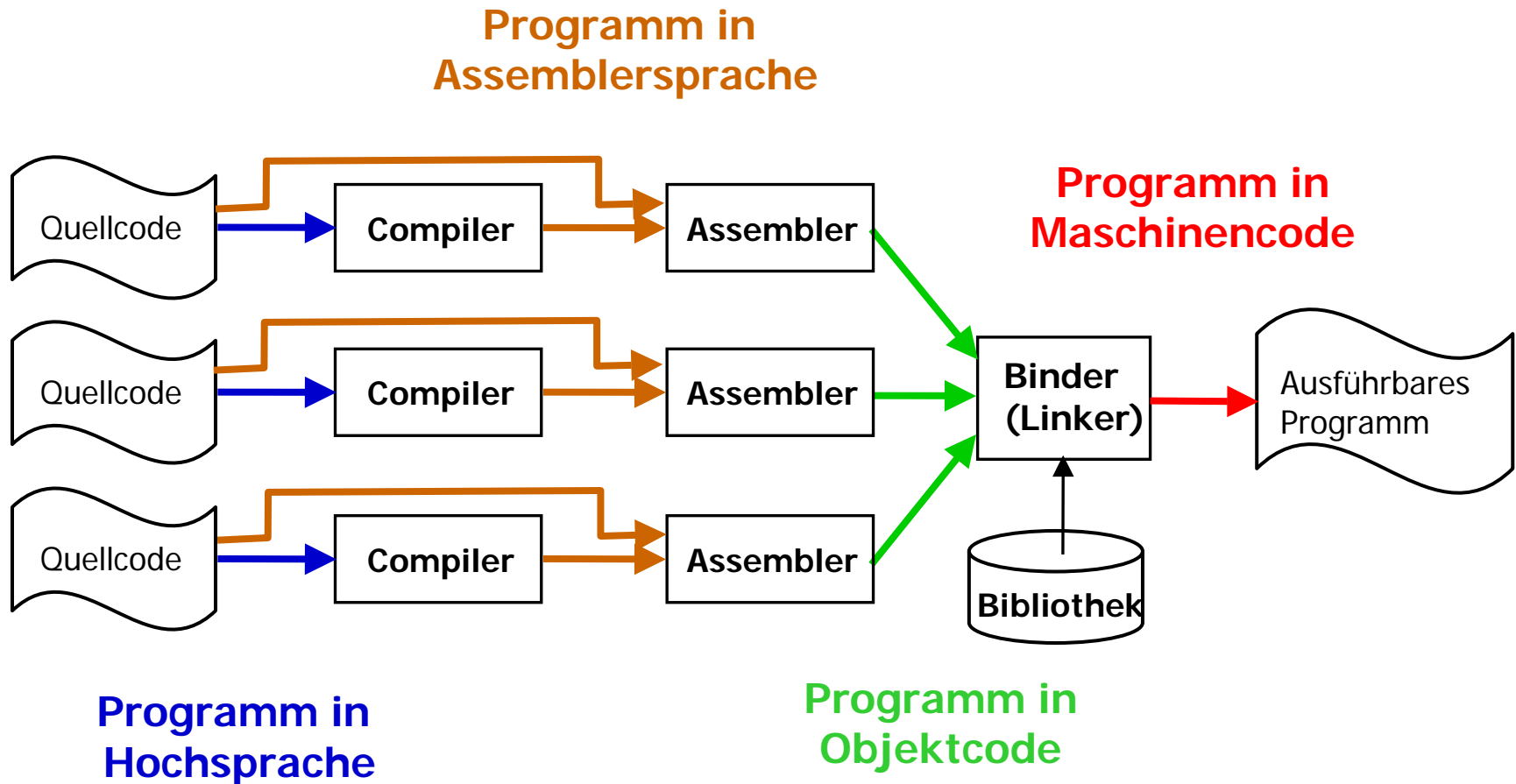
□ Anforderungen höherer Programmiersprachen

- Programmkonstrukte
- Variable und Konstante

□ Ein grundlegendes Rechnermodell

- Leitwerk, Rechenwerk
- Speicherwerk
- Ein-Ausgabewerk
- Verbindungsstrukturen
- Maschinenbefehlszyklus

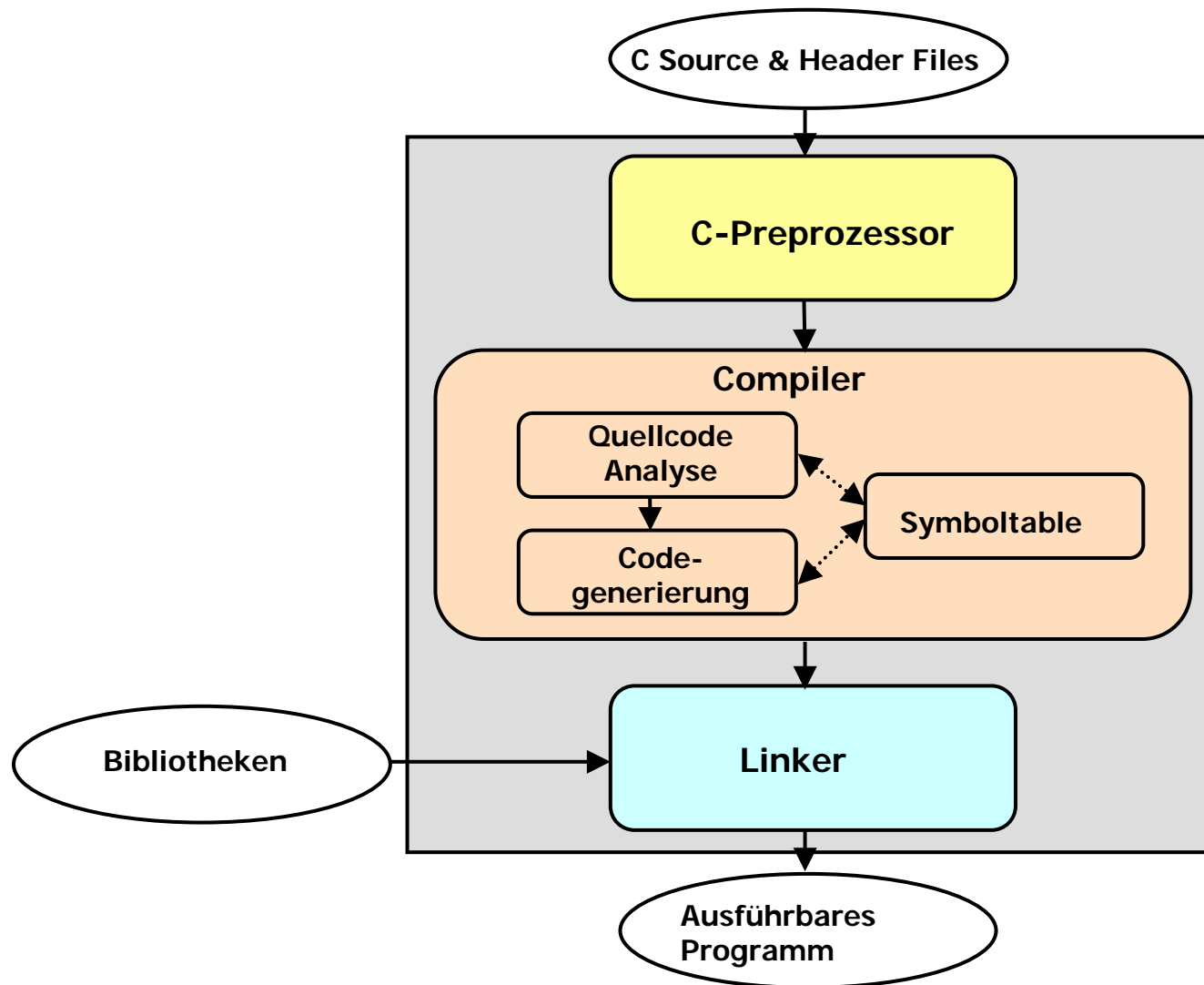
Wdh. Vom Quellcode zum ausführbaren Programm



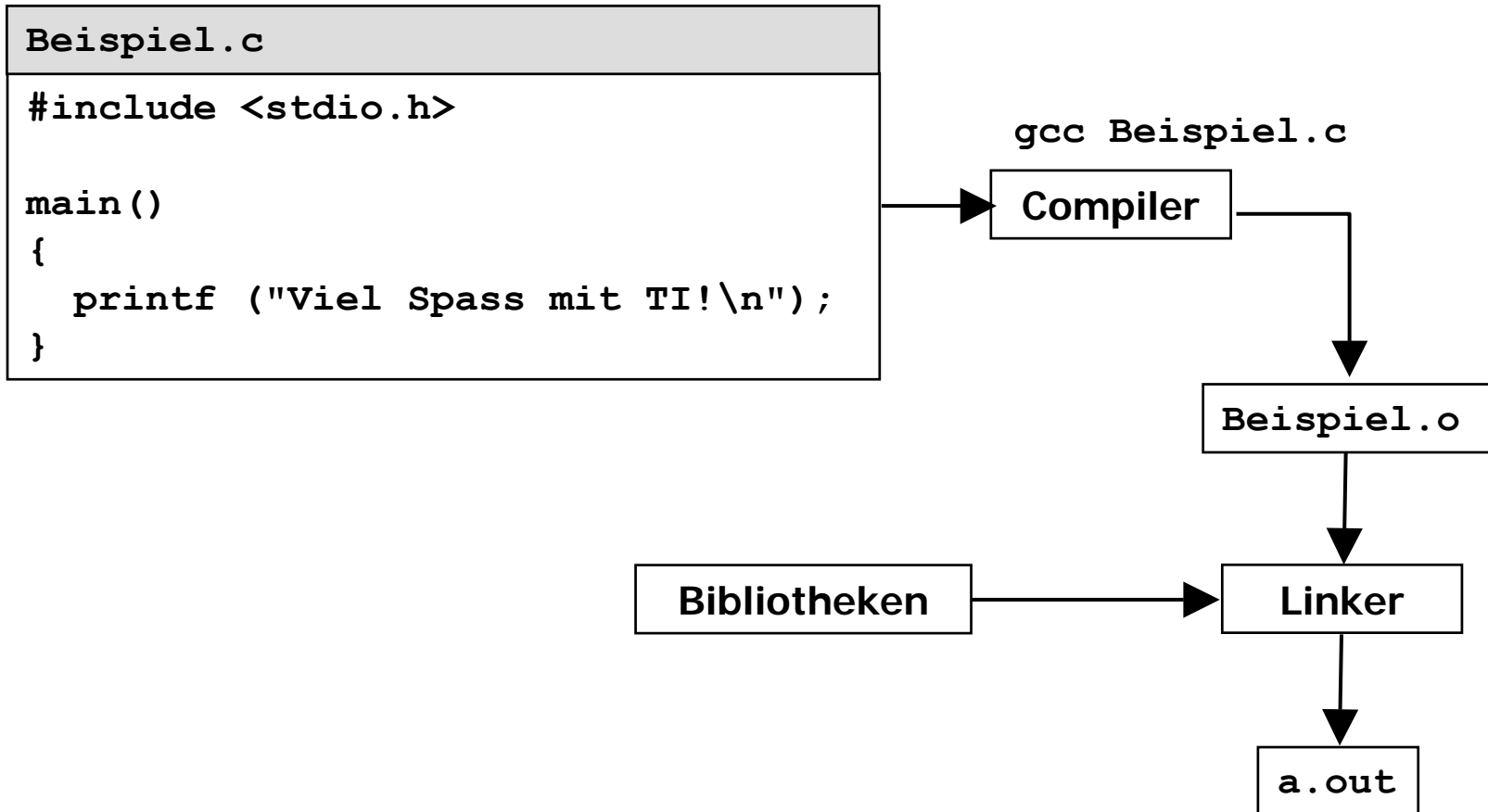
Wdh. Sprache C

- ❑ C kennt nur 4 Datentypen: **char**, **int**, **float** und **double**.
- ❑ Es gibt einfache Kontrollstrukturen: Entscheidungen, Schleifen, Zusammenfassungen von Anweisungen (Blöcke) und Unterprogramme.
- ❑ Ein-/Ausgabe ist nicht Teil der Sprache C sondern wird über Bibliotheksfunktionen erledigt. Keine Operationen auf zusammengesetzten Objekten als Ganzes (z. B. Zeichenketten).
- ❑ Parameter werden als Werte an Funktionen übergeben, daher kann eine Funktion diese nicht ändern.
- ❑ Möglichkeit von Zeigern (Adressen) als Parametern, deren Zielobjekt geändert werden kann.

Programmerstellung und -ausführung



Einfaches Beispiel



2.3 Grundlagen: Datentypen

□ 4 Grunddatentypen

- **char**: einzelnes Zeichen (Charakter); meist 1 Byte
- **int**: Integerzahl; 2 oder 4 Byte
- **float**: Gleitkommazahl; meist 4 Byte
- **double**: Gleitkommazahl in doppelter Genauigkeit; meist 8 Byte
- Wertebereiche dieser Grunddatentypen ist **rechnerabhängig**
- Beispiele:
 - `char buchstabe = 'T';`
 - `int i = 4;`
 - `double pi = 3.1415;`

2.3 Grundlagen: Operatoren

□ Arithmetische Operatoren:

Operator Symbol	Operation	Beispiel
*	Multiplikation	$x * y$
/	Division	x / y
%	Modulo	$x \% y$
+	Addition	$x + y$
-	Subtraktion	$x - y$

2.3 Grundlagen: Operatoren

□ Bit-Operatoren:

Operator Symbol	Operation	Beispiel
~	Bitweise NOT	$\sim x$
<<	links Schieben	$x \ll y$
>>	rechts Schieben	$x \gg y$
&	Bitweise AND	$x \& y$
^	Bitweise XOR	$x \wedge y$
	Bitweise OR	$x \mid y$

2.3 Grundlagen: Operatoren

□ Vergleichsoperatoren:

Operator Symbol	Operation	Beispiel
>	größer als	$x > y$
>=	größer gleich	$x \geq y$
<	kleiner als	$x < y$
<=	kleiner gleich	$x \leq y$
==	gleich	$x == y$
!=	ungleich	$x != y$

2.3 Grundlagen: Operatoren

□ Spezial-Operatoren:

Operator Symbol	Operation	Beispiel
++	Inkrement (postfix)	x ++
--	Dekrement (postfix)	x --
++	Inkrement (präfix)	++x
--	Dekrement (präfix)	--x
+=	add and assign	x += y
-=	subtract and assign	x -= y
*=	multiply and assign	x *= y
/=	divide and assign	x /= y
%=	modulus and assign	x %= y
&=	and and assign	x &= y
=	or and assign	x = y
^=	xor and assign	x ^= y
<<=	left-shift and assign	x <<= y
>>=	right-shift and assign	x >>= y

2.3 Grundlagen: Operatoren

□ Spezial-Operatoren:

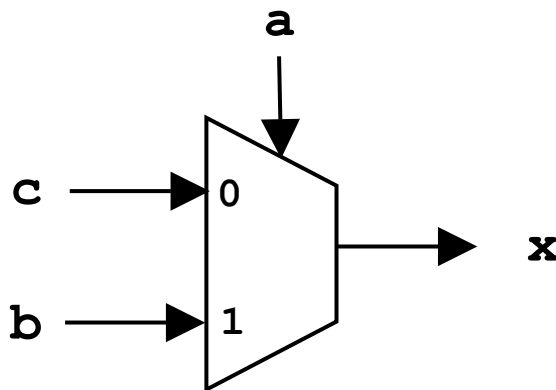
➤ Operatorpaar „?:“ `expr1 ? expr2 : expr3`

➤ **Beispiel 1:**

`z = (a > b) ? a : b;` `/* z = max(a,b) */`

➤ **Beispiel 2:**

`x = a ? b : c` `/* a true dann x = b */`
 `/* sonst x = c */`



Bedingter Ausdruck entspricht der logischen Funktion eines Multiplexers

Operatoren nach Priorität

Priorität	Symbol	Assoziativität	Bedeutung
15	()	L-R	Funktionsaufruf
	[]		Indizierung
	->	R-L	Elementzugriff
	.		Elementzugriff
14	+(Vorzeichen)	R-L	Vorzeichen
	-(Vorzeichen)		Vorzeichen
	!		
	~		Bitkomplement
	++(Präfix)		Präfix-Inkrement
	--(Präfix)		Präfix-Dekrement
	++(Postfix)		Postfix-Inkrement
	--(Postfix)		Postfix-Dekrement
	&		Adresse
	*		Zeigerdereferenzierung
	(Typ)		Typumwandlung
	sizeof		Größe

Operatoren nach Priorität

Priorität	Symbol	Assoziativität	Bedeutung
13	*	L-R	Multiplikation
	/		Division
	%		Modulo
12	+		Addition
	-		Subtraktion
11	<<	L-R	Links-Shift
	>>		Rechts-Shift
10	<	L-R	Kleiner
	<=		Kleiner gleich
	>	L-R	größer
	>=		größer gleich
9	==	L-R	gleich
	!=		ungleich

Operatoren nach Priorität

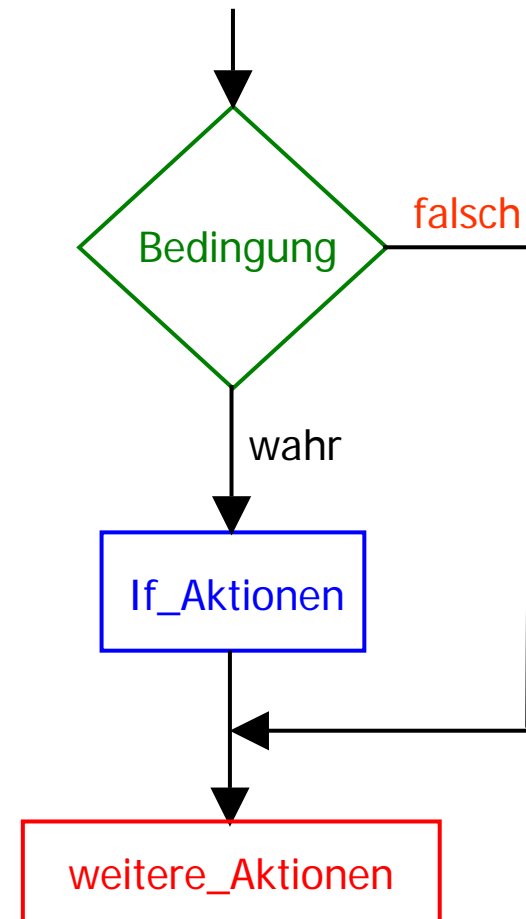
Priorität	Symbol	Assoziativität	Bedeutung
8	&	L-R	Bitweise UND
7	^	L-R	Bitweise exklusives ODER
6		L-R	Bitweise ODER
5	&&	L-R	logisches UND
4		L-R	Bitweise ODER
3	:?	L-R	Bedingung
2	=	L-R	Zuweisung
	=, /=, %, +=, -=, &=, ^=, =, <<=, >>=		Zusammengesetzte Zuweisung
1	,		Komma-Operator

2.4 Kontrollstrukturen

Kontrollstrukturen definieren die Reihenfolge von Berechnungen

□ **if Anweisung:**

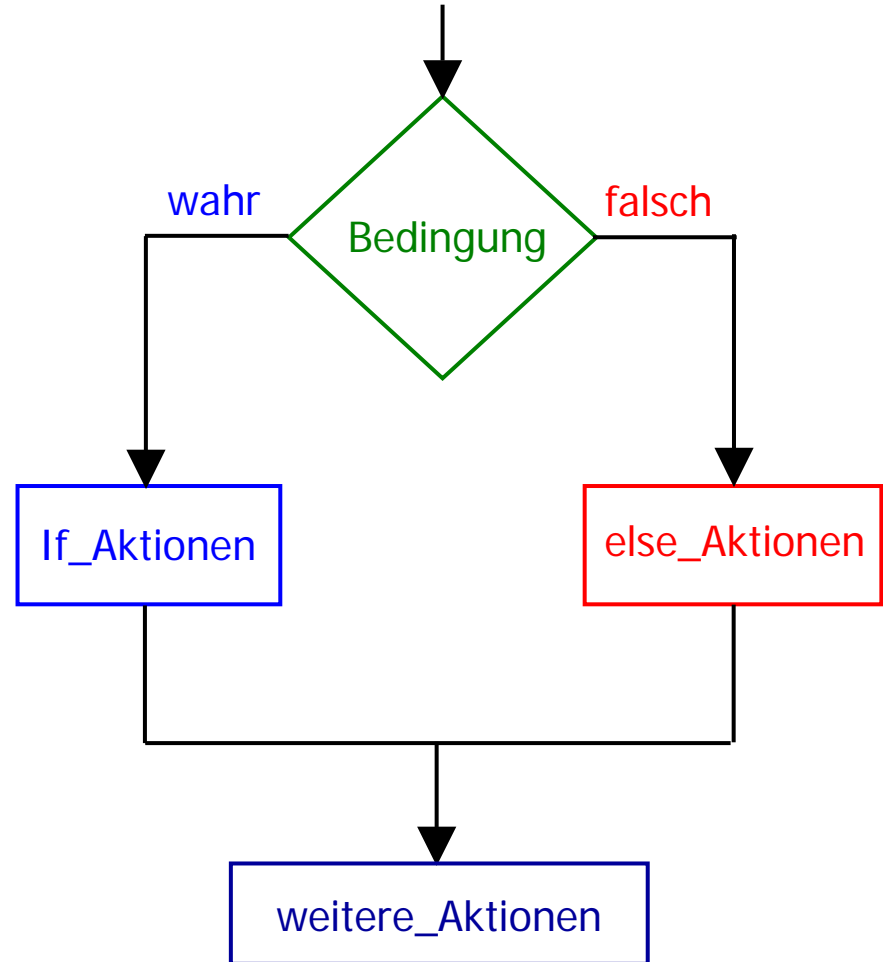
```
if (Bedingung_ist_wahr)
{
    if_Aktionen
}
weitere_Aktionen;
```



2.4 Kontrollstrukturen

□ if-else Anweisung:

```
if (Bedingung_ist_wahr)
{
    if_Aktionen
}
else
{
    else_Aktionen
}
weitere_Aktionen;
```



2.4 Kontrollstrukturen

□ switch Anweisung:

```
switch(ausdruck)
{
    case a: a Anweisungen    /* ausdruck = a */
        break;

    case b: b Anweisungen    /* ausdruck = b */
        break;

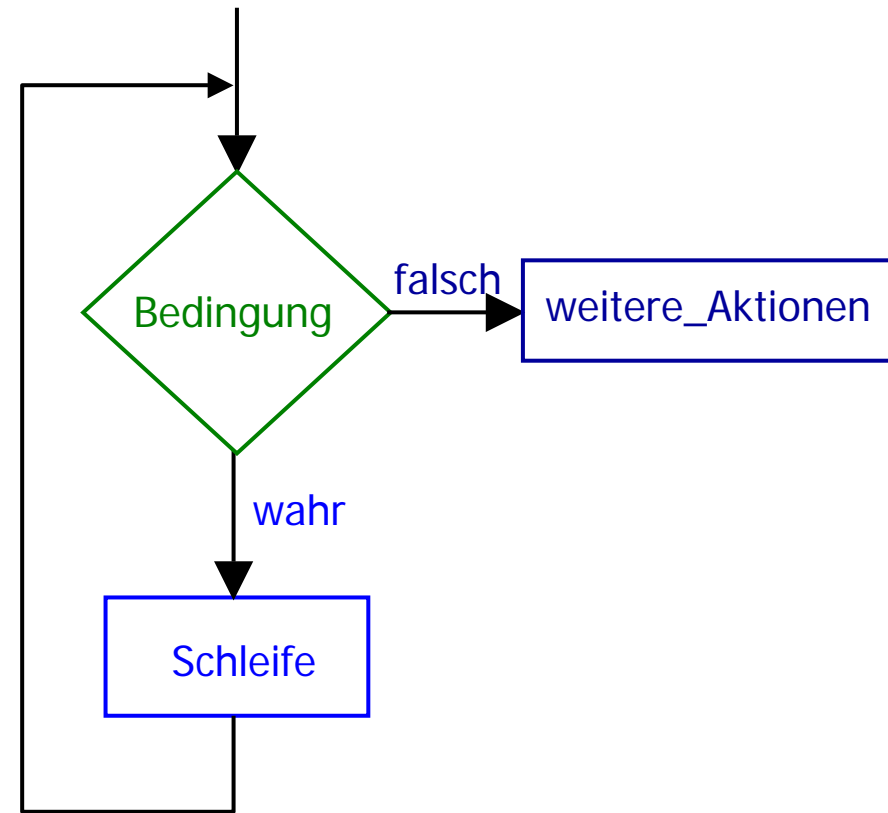
    case c: c Anweisungen    /* ausdruck = c */
        break;

    default: default Anweisungen
        break;
}
```

2.4 Kontrollstrukturen

□ while Schleifen:

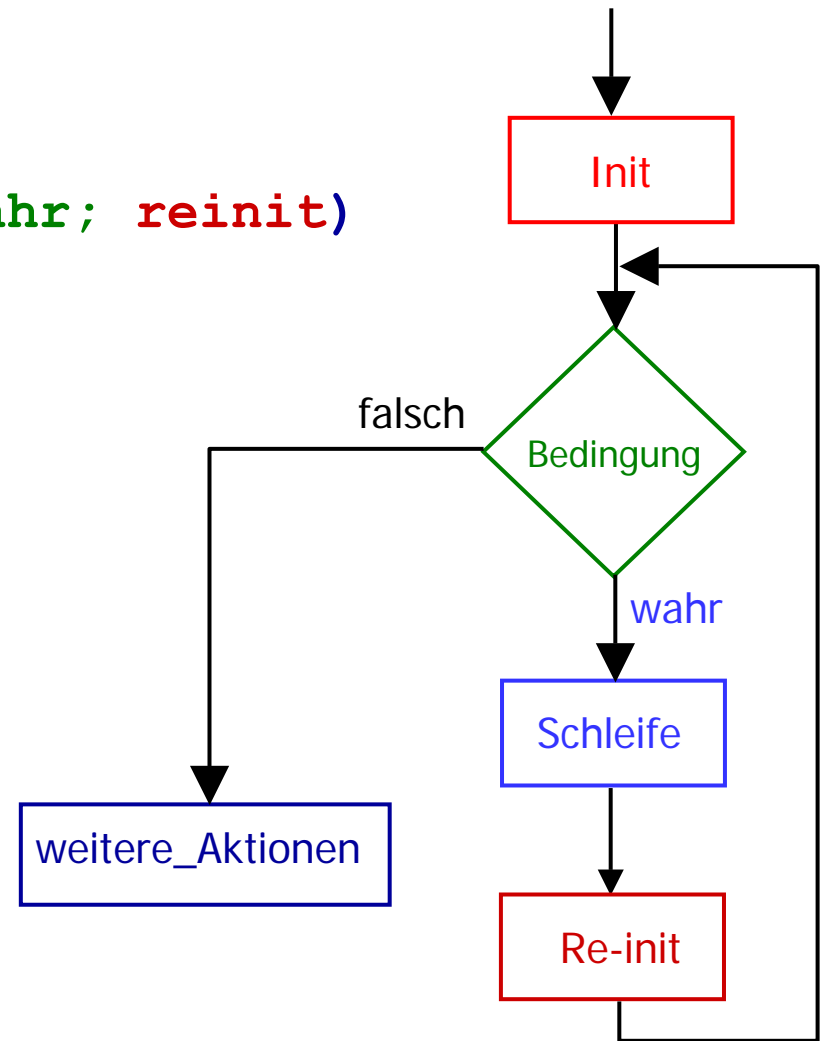
```
while (Bedingung_ist_wahr)
{
    while_Aktionen
}
weitere_Aktionen;
```



2.4 Kontrollstrukturen

□ for Schleifen:

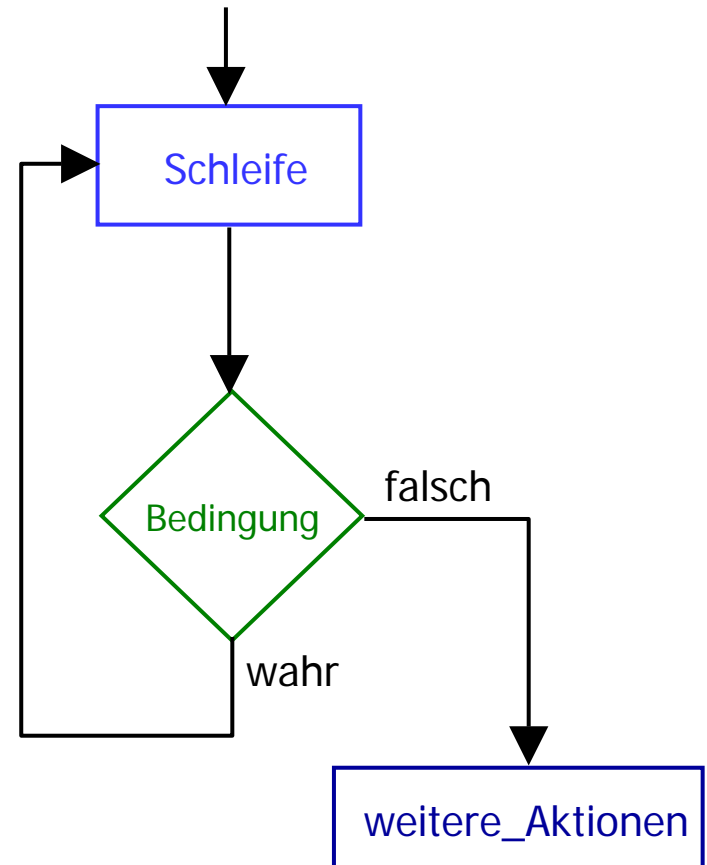
```
for (init; Bedingung_ist_wahr; reinit)
{
    for_Aktionen
}
weitere_Aktionen;
```



2.4 Kontrollstrukturen

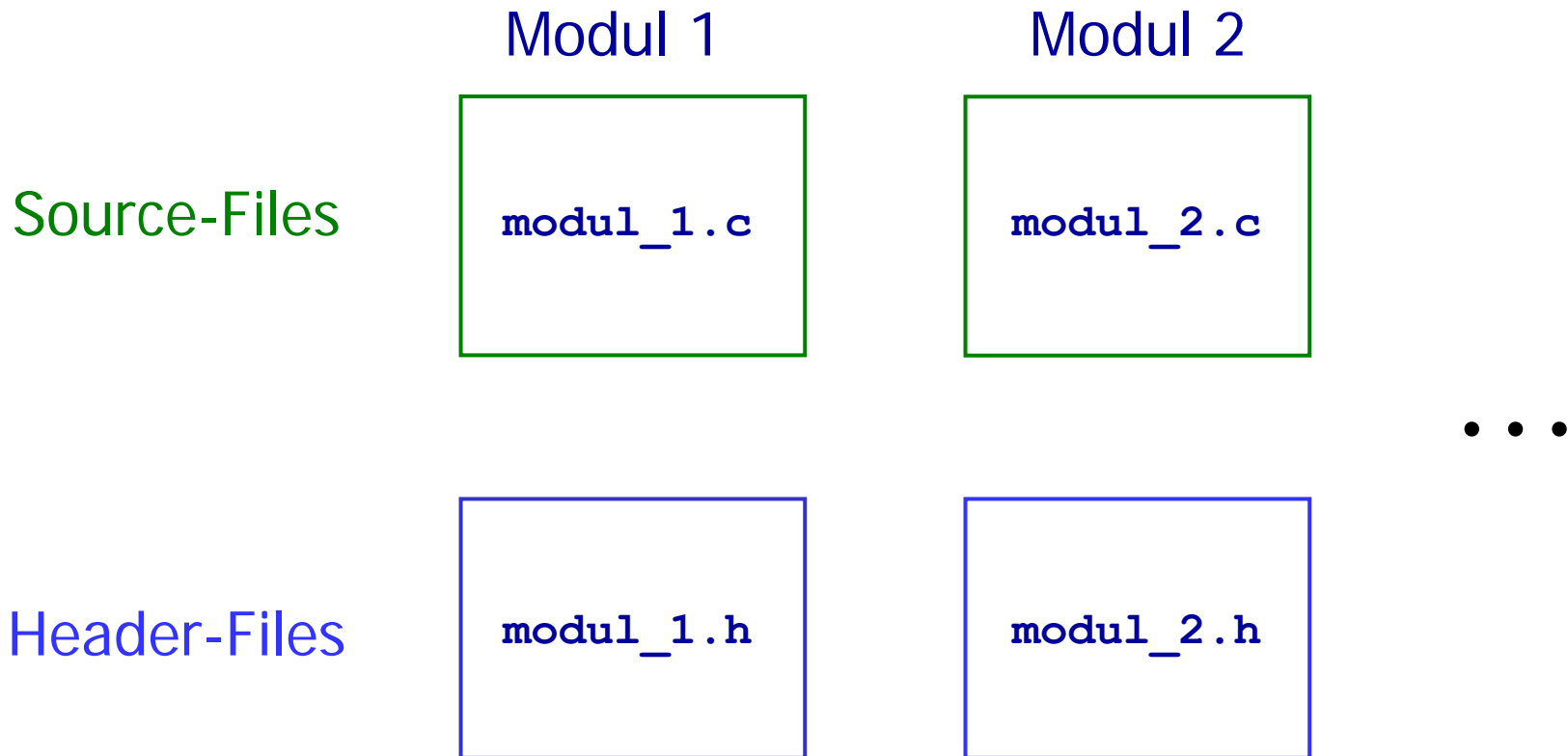
□ do-while Schleife:

```
do {  
    do_Aktionen  
}  
while (Bedingung_ist_wahr)  
  
weitere_Aktionen;
```



2.5 Funktionen und Programmstruktur

Aufbau eines Programms



2.5 Funktionen und Programmstruktur

Aufbau eines Programms

Preprozessor Anweisungen

```
#include <stdio.h>          /* Einbinden einer Bibliothek (I/O-Funktionen) */
#include "modul_1.h"         /* Einbinden eines Moduls „modul_1“ */

#define COLOR_OF_EYES  blau /* Globale Textersetzung im Programmtext,
                             Makrodefinition */
```

Globale Deklarationen und Definitionen:

```
int i;                      /* Deklaration */
int j=13;                   /* Definition */
int fakultaet (int n);      /* Funktionsprototyp */
```

2.5 Funktionen und Programmstruktur

```
int fakultaet (int n) {    /* Funktionsdefinition */  
    ...  
}
```

```
void main () {  
    printf("Hallo \n");  
}
```

- Jedes Programm enthält eine Funktion „main“ von Typ void
- Unterprogramme werden in C in Form von **Funktionen** definiert, die **Parameterlisten** und **Ergebnistypen** haben.
- Die Abarbeitung eines Programms beginnt stets mit der Ausführung von „main“. Innerhalb von „main“ können die weiteren definierten Funktionen aufgerufen werden.

2.5 Funktionen und Programmstruktur

- ❑ Funktionen werden global definiert
- ❑ rekursive Funktionsaufrufe sind zulässig: eine Funktion darf sich selbst aufrufen
- ❑ Beispiel (Fakultätsberechnung):

```
fakultaet(int n) {  
    if ( n == 1 )  
        return(1);  
    else  
        return( n * fakultaet(n-1) );  
}
```

Parameterübergabe an Funktionen

❑ **call by value**

- Normalfall in C
- Es wird eine Kopie des aktuellen Parameters an die Funktion übergeben
- Funktionen können den Wert der Kopien ändern, ohne Auswirkungen auf den Wert des aktuellen Parameters beim Aufrufer.

❑ **call by reference**

- In C nur indirekt mit Hilfe von Zeigern realisierbar
- Der Übergabeparameter ist eine Variable und die aufgerufene Funktion erhält die Speicheradresse dieser Variablen.
- Wenn die Funktion den Wert des Parameters verändert, ändert sie den Inhalt der zugehörigen Speicherzelle → der Wert der Variablen beim Aufrufer der Funktion ändert sich auch

Globale und Lokale Variablen

- ❑ **Globale Variablen** sind im gesamten Programm einschließlich aller Unterprogramme bekannt. (Sie sollten vermieden werden)
- ❑ **Lokale Variablen** sind innerhalb einer Funktion oder eines „Blockes“ deklarierte Variable.

```
#include <stdio.h>
```

```
int global = 0;          /* Das ist eine globale Variable */
```

```
main () {  
    int lokal = 1;       /* Das ist eine Lokale Variable in main */  
    printf("Global %d, Lokal %d\n", global, lokal);  
    {  
        int lokal = 2;   /* Das ist eine lokale Variable im Block */  
        printf("Global %d, Lokal %d\n", global, lokal);  
    }  
    printf("Global %d, Lokal %d\n", global, lokal);  
}
```

Speicherklassen

Definieren die Lebensdauer und den Gültigkeitsbereich (Sichtbarkeit) von Variablen und/oder Funktionen.

In C gibt es die Speicherklassen:

- **auto** wird für lokale Variablen eingesetzt. Objekte, die als auto gelten nur solange wie der Bereich, in dem sie definiert wurden.
- **register** wird verwendet, um eine lokale Variable in einem Register der CPU zu speichern. „register“ sollte nur für Variablen verwendet werden, auf die schnell zugegriffen werden muss, z. B. Zähler
- **static** wird bei der Definition globaler Variablen verwendet.
- **extern** definiert eine globale Variable, die in allen Programm-Modulen sichtbar ist.

2.6 Zeiger und Vektoren

- ❑ Mächtigstes Werkzeug der Sprache C zur maschinennahen Programmierung
- ❑ Effiziente Programmierung durch Vermeidung von Kopieroperationen
- ❑ Pointerarithmetik → Direkter Zugriff auf Elemente strukturierter Daten
- ❑ Fortgeschrittene Verwendung von Pointern kann zu kryptischem Code und Unleserlichkeit führen ☹
- ❑ Macht das Programmieren mit C fehleranfällig.

2.6 Zeiger und Vektoren

- Ein **Zeiger „pointer“** enthält eine Adresse, die auf Daten verweist.

```
int *p;          /* *ptr ist vom Typ „int“  
                  ptr ist vom Typ „Zeiger auf int“ */  
  
int *q;  
int a = 3;  
int b;  
  
p = &a;          /* der unäre Operator „&“ liefert die  
                  Adresse von */  
  
b = *p + 1;      /* der unäre Operator „*“ liefert den  
                  Wert, auf den p zeigt */  
  
q = (int*) 0x8010 /* Typumwandlung einer Adresse in  
                  Hexadezimalschreibweise auf  
                  „Zeiger auf int“ */
```

2.6 Zeiger und Vektoren

```
int *p;  
int *q;  
int a = 3;  
int b;  
  
p = &a;  
b = *p + 1;  
q = (int*) 0x8010
```

	Adresse	Inhalt
p	...	0x8004
a	0x8004	3
b	...	4
q	...	0x8010
	0x8010	

2.6 Zeiger und Vektoren

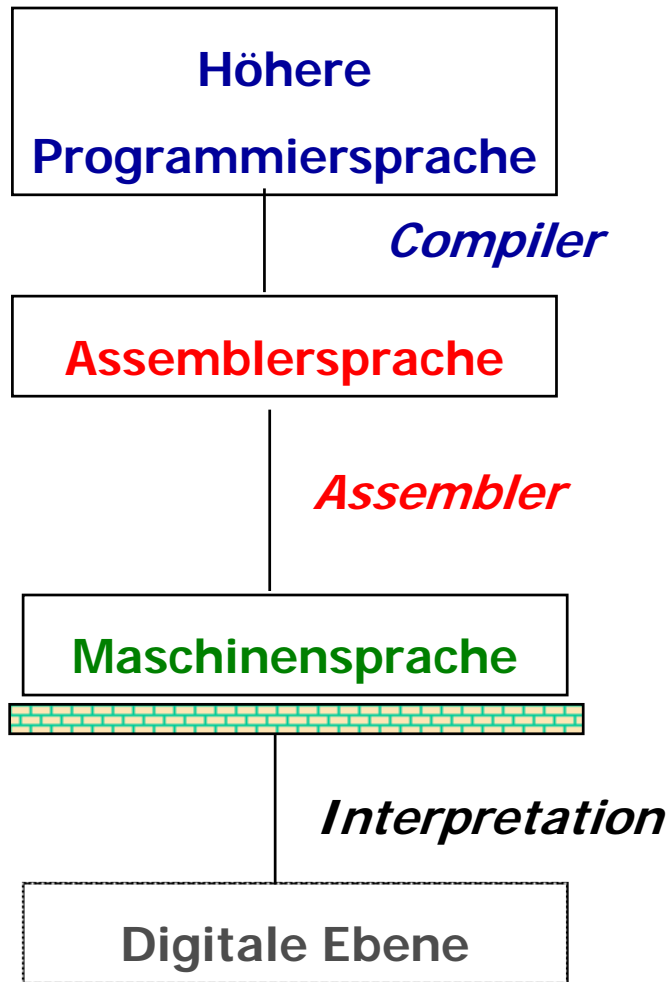
```
int n[] = {1, 2, 3};  
char c[] = {'a', 'b', 'c'};  
char s[] = "de";  
s[1] = 'f';
```

	Adresse	Inhalt
n[0]	0x8004	1
n[1]	0x8008	2
n[2]	0x800C	3

	Adresse	Inhalt
c[0]	0x8014	'a'
c[1]	0x8018	'b'
c[2]	0x801C	'c'

	Adresse	Inhalt
s[0]	0x8024	'd'
s[1]	0x8028	'f'
s[2]	0x802C	'\0'

Hierarchie



```
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
```

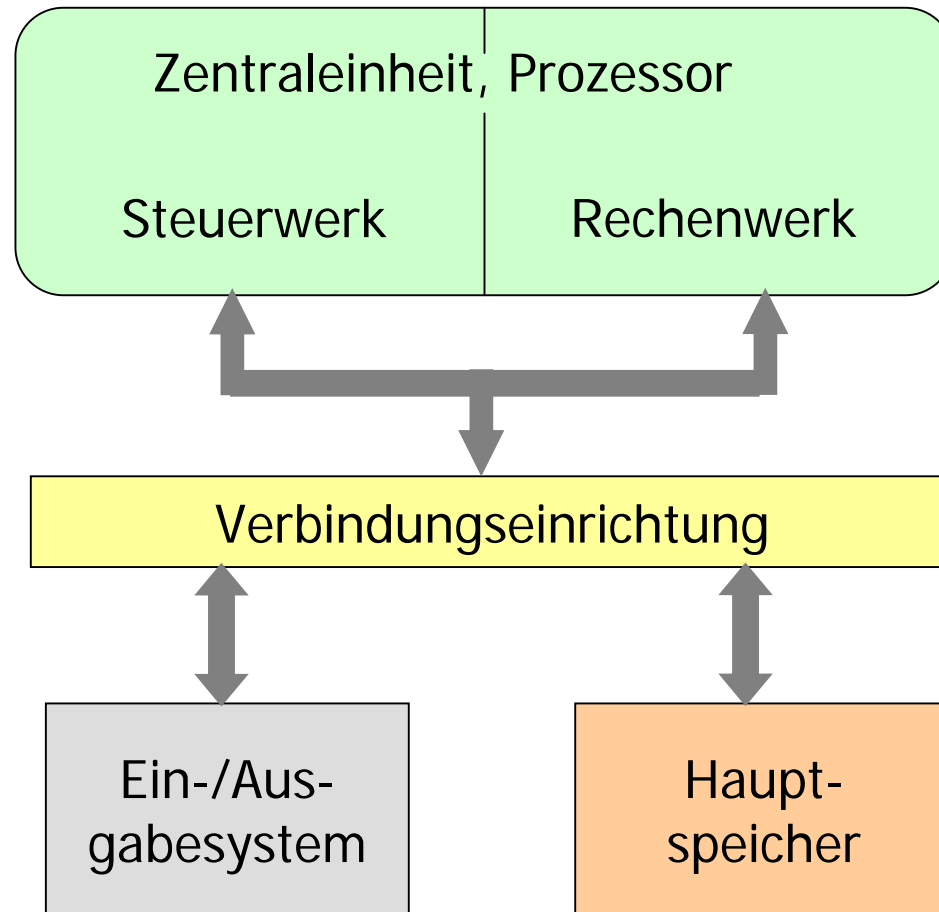
```
ALUOP[0:3] ← InstReg[9:11] & MASK
```

Kapitel 3

Ein grundlegendes Rechnermodell

- Organisationsprinzip des von Neumann Rechners
- Aufbau eines einfachen Mikroprozessors
 - Steuerwerk (Leitwerk)
 - Rechenwerk
 - Speicherwerk
 - Ein-Ausgabewerk
 - Verbindungsstrukturen
- Maschinenbefehlszyklus

3.1 Organisationsprinzip des von Neumann Rechners



Komponenten des von Neumann Rechners

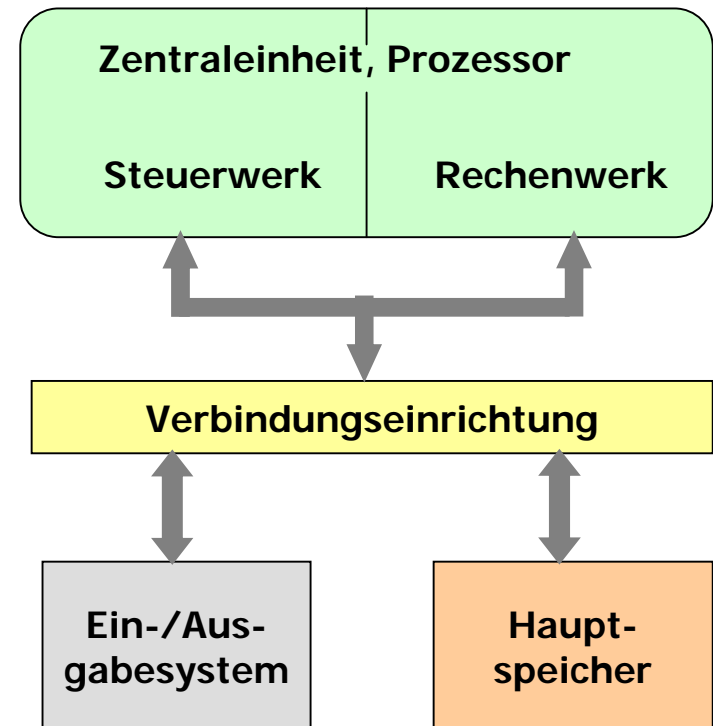
□ Zentraleinheit

(central processing unit,
CPU, Prozessor)

Verarbeitet Daten gemäß eines
Programms. Sie besteht aus
Leitwerk und Rechenwerk:

➤ Leitwerk (Steuerwerk, control unit, CU)

- Holt die Befehle eines Programms
aus dem Speicher
- entschlüsselt sie und
- steuert ihre Ausführung in der verlangten
Reihenfolge durch
Steuer- und Synchronisier-Signale.

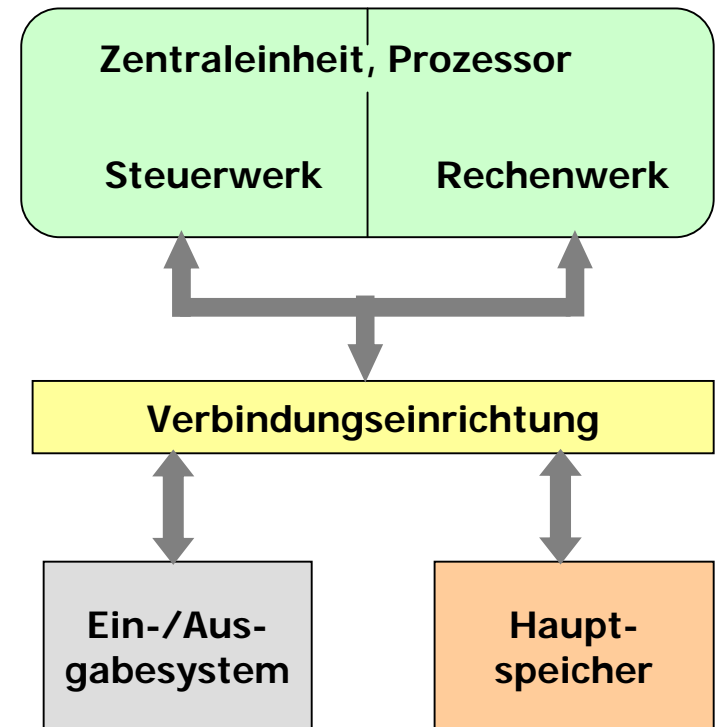


Komponenten des von Neumann Rechners

□ Zentraleinheit

Verarbeitet Daten gemäß eines Programms. Sie besteht aus Leitwerk und Rechenwerk:

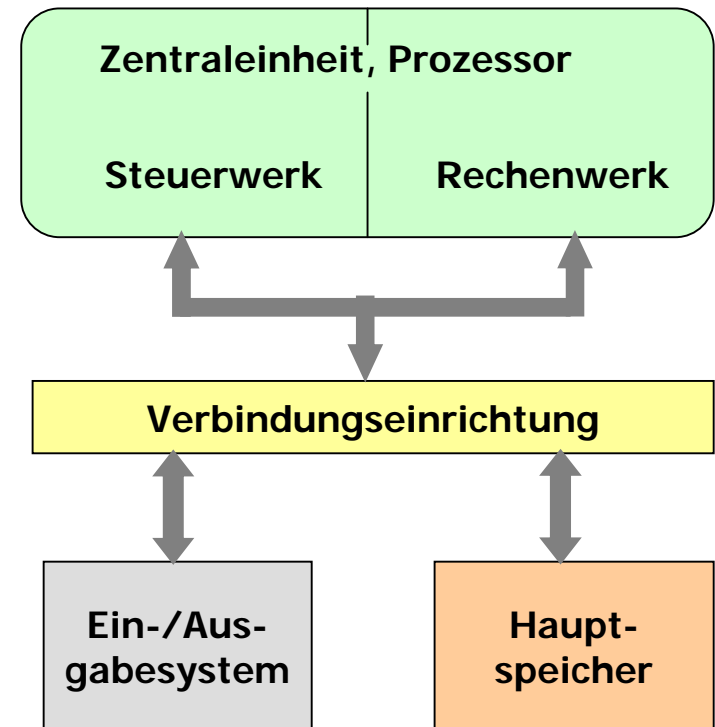
- **Rechenwerk (Operationswerk, Ausführungseinheit, ALU)**
 - Führt arithmetisch/logische Operationen aus.
 - Wird durch Steuersignale des Leitwerks beeinflusst und
 - liefert seinerseits Meldesignale an das Leitwerk zurück.



Komponenten des von Neumann Rechners

□ Hauptspeicher

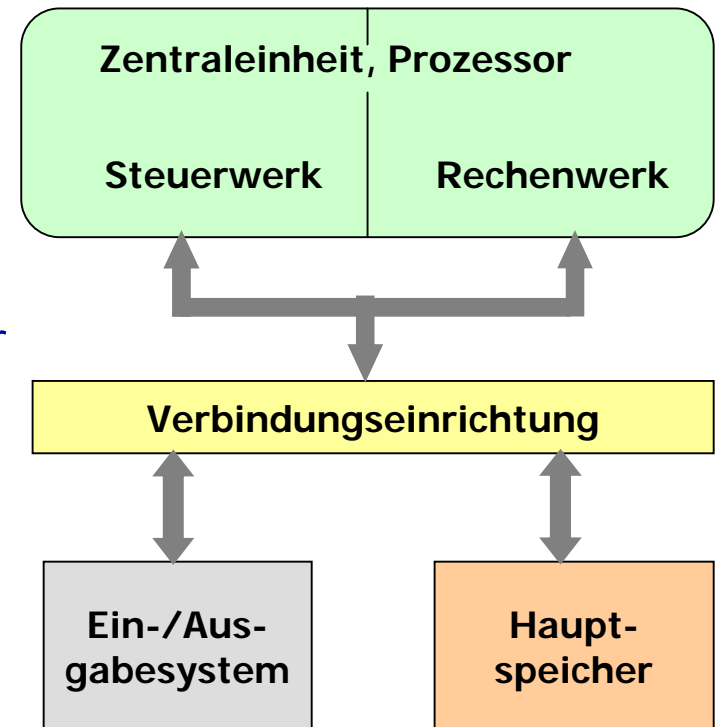
- Jede Speicherzelle ist eindeutig durch ihre Nummer (Adresse) identifizierbar.
- Dort werden Programme und Daten aufbewahrt (von-Neumann-Konzept).
- Den einzelnen Speicherzellen ist nicht anzusehen, welchen Typ von Information sie enthält
- Alternativ: **Harvard-Architektur** mit getrenntem Programm- und Datenspeicher.
- Inhalt nach Abschaltung des Rechners flüchtig.



Komponenten des von Neumann Rechners

□ Verbindungsstruktur (BUS)

- **Adreßleitungen:** Leitungen, auf denen die Adressinformation transportiert wird (unidirektional).
- **Datenleitungen:** Transportieren Daten und Befehle von/zum Prozessor (bidirektional).
- **Steuerleitungen:** Geben Steuerinformationen von/zum Prozessor (uni- oder bidirektional).



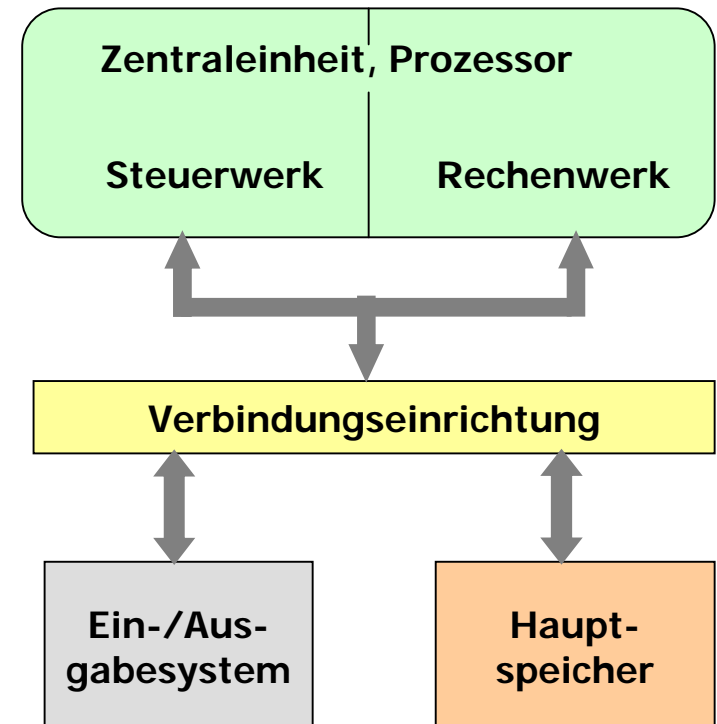
□ Bus (Sammelschiene)

Systembus = Adreßbus + Datenbus + Steuerbus

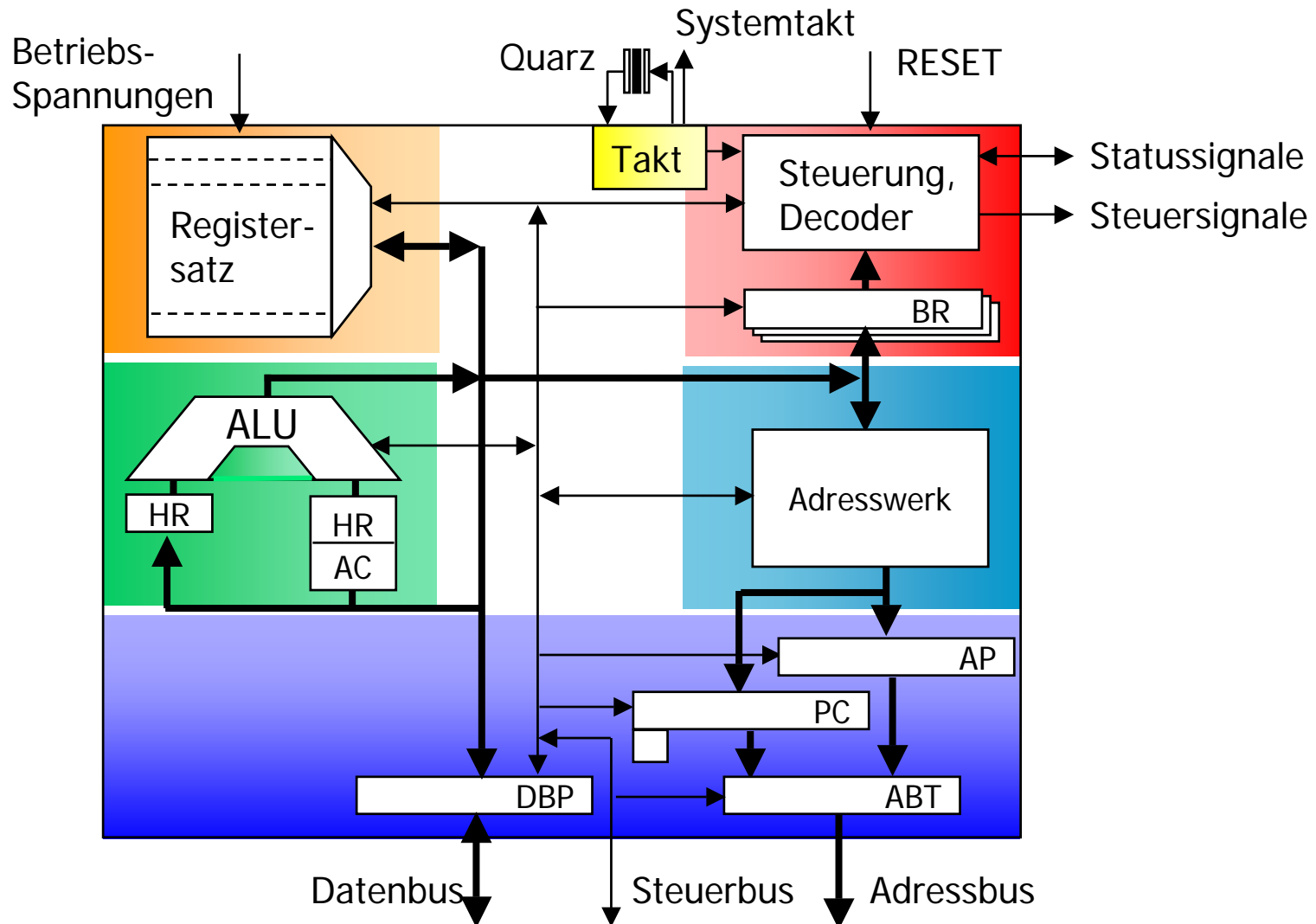
Komponenten des von Neumann Rechners

□ Ein-/Ausgabesystem (Peripheriegeräte)

- Geräte zur Eingabe von Daten und Programmen und zur Ausgabe der verarbeiteten Daten (Bildschirme, Drucker, Terminals, ...)
- Diese Geräte sind über Ein-/Ausgabe-Schnittstellen mit dem Rechner verbunden.
- Die Verbindung der Schnittstellen mit dem Prozessor (und zu den Peripheriegeräten) geschieht durch Adreß-, Daten- und Steuerleitungen.



3.2 Aufbau eines einfachen μP



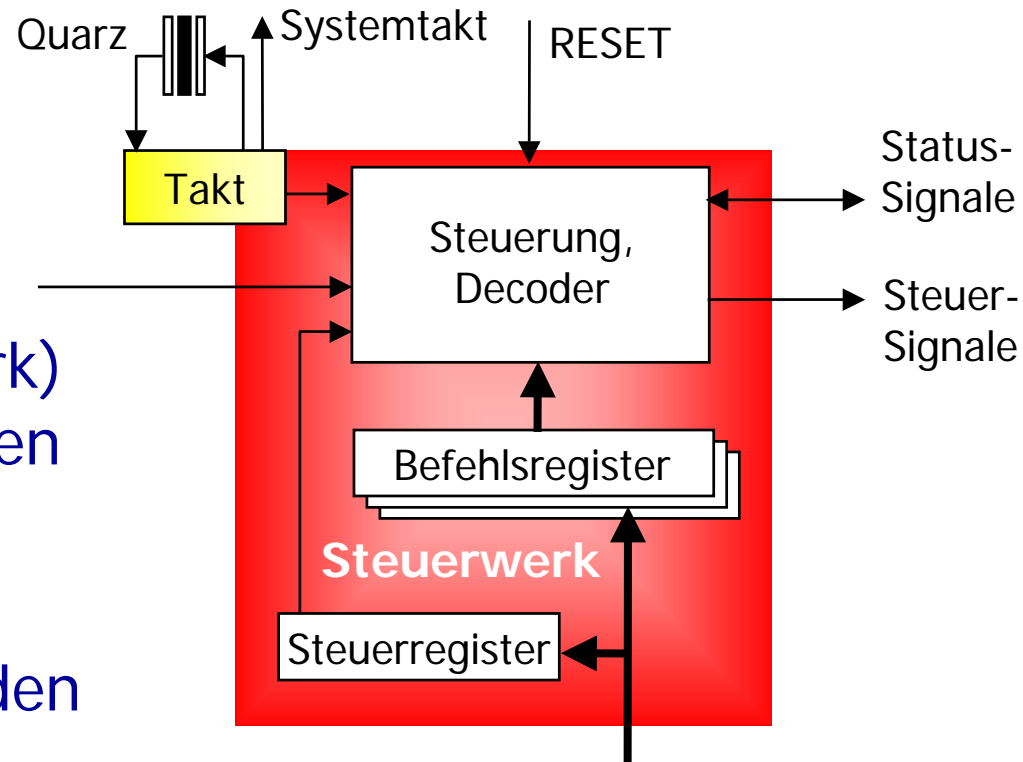
Aufbau eines einfachen μP

- ❑ Steuerwerk
- ❑ Rechenwerk
- ❑ Registersatz
- ❑ Adresswerk
- ❑ Systembusschnittstelle
- ❑ Interne Busse

3.2.1 Steuerwerk

Steuert die Systemkomponenten

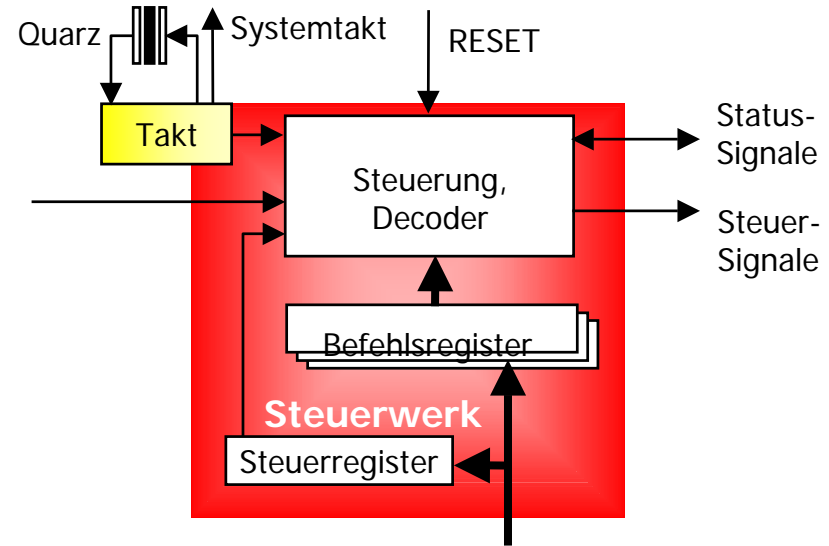
- **Befehlsregister**
enthält den gerade ausgeführten Befehl
- **Dekoder** (mikro-programmiertes Schaltwerk)
wird von den Statussignalen beeinflusst und erzeugt die Steuersignale
- **Taktgenerator** erzeugt den vom externen Quarz festgelegten Systemtakt



3.2.1 Steuerwerk

Synchrones Schaltwerk

Meist liegt ein sog. **dynamisches Schaltwerk** vor, d. h. die Zustandsinformation ist nicht in Flipflops, sondern in Kondensatoren gespeichert



→ Mindesttaktfrequenz ist erforderlich

Unterhalb dieser Taktfrequenz gehen die Inhalte durch Leckströme bereits vor dem nächsten Taktzyklus verloren.

Taktgenerator ist bei modernen Mikroprozessoren on Chip (mit externem Quarz verbunden)