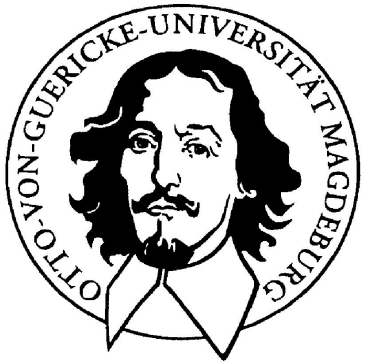
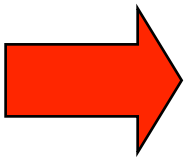
Rechnersysteme



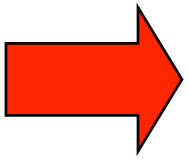
Prozessornahe Programmiertechniken

Otto-von-Guericke-Universität Magdeburg

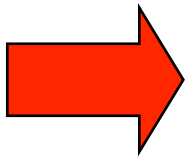
Prozessornahe Programmiertechniken



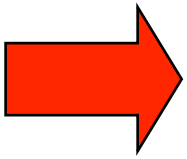
Variablen und Zuweisungen



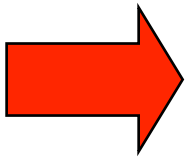
Datenstrukturen und Zeiger



Kontrollstruktur-Muster



Positionsunabhängiger Code



Unterprogrammtechniken

Variablen und Zuweisungen

```
C Programmsegment:
..
uint8_t a, b, c;          /*Deklaration des Bedarfs an Speicherplatz  */
                          /*vom Datentyp z.B. unsigned integer, 8 Bit */
..
..
a = b + c;                /* addiere b und c und speichere das Resultat in a */
```

In imperativen Sprachen ist "a = b + c" *kein mathematischer Ausdruck*, sondern eine Vorschrift, wie mit Inhalten von Speicherplätzen umzugehen ist!

Fragen:

- wo befinden sich die Operanden ?

- welcher Operator wird angewendet?

- wo wird das Ergebnis gespeichert?

```
C Programmsegment:
..
uint8_t a, b, c;          /*Deklaration des Bedarfs an Speicherplatz   */
                          /*vom Datentyp z.B. unsigned integer, 8 Bit  */
..
..
a = b + c;                /* addiere b und c und speichere das Resultat in a */
```

a, b, und c sind *Variablen*, die im Rechner durch *Speicherplätze* repräsentiert sind, die entsprechende Werte enthalten.

Dieser Bedarf an Speicherplätzen wird durch die *Deklaration* "int a, b, c;" spezifiziert. Der Compiler wird dieses Konstrukt in Anweisungen übersetzen, die:

1. Für die Variablen a, b,und c entsprechende Speicherplätze oder Register reservieren. Durch die Deklaration der Datentypen ist die Größe der Speicherworte festgelegt.

2. Die entsprechende Operation (gemäß den Datentypen) durchführen.

3. Das Resultat auf einem spezifizierten Speicherplatz ablegen. Daher bezeichnet "=" nicht Gleichheit sondern ist die Zuweisung "←".

Variablen und Zuweisungen

a ← b + c

Ein Compiler würde für die 6809 folgende Assemblersequenz erzeugen und Reservierungen vornehmen.

```
..
..
Start       LDA var-b      /*lade den Wert in var-b in den Accumulator
            ADD var-c      /*addiere den Wert var-c (aus dem Speicher)
            STA var-a      /* speichere das Resultat auf Speicherplatz var-c
..
..
..
var-a       RMB
var-b       RMB
var-c       RMB
```

Falls man einen Operanden nach der Addition nicht mehr benötigt, kann er mit dem Resultat überschrieben werden, z.B. b ← b+c. Das spart Speicherplatz und zeigt nochmal deutlich, dass "=" ein Zuweisung bezeichnet.

Variablen und Zuweisungen

2-Adress-Befehle
z.B. 68K oder Pentium

Code Sequenz:

```
ld r2, b
ld r3, c
mv r1, r2
add r1, r3
st a, r1
```

3 Adress-Befehle
RISC-CPUs wie Sparc, MIPS, Alpha, ARM, .

Code Sequenz:

```
ld r2, b
ld r3, c
add r1, r2, r3
st a, r1
```

r2 und/oder r3 die Quellregister und r1 das Zielregister. a, b, und c sind Speicherplätze.

Fazit: Das Programmstatement "a = b + c" ist kein mathematisch oder logischer Ausdruck, sondern a, b, und c sind Container für Daten und "=" ist die Zuweisung des Resultat-Containers.Die Architektur des Rechners beeinflusst in erheblichem Maß die Umsetzung.

Variablen und Zuweisungen

```
C-Programm
"simple_add_char.c"

static char a, b, c;
int main()
{
a=b+c;
}
```

(globale) Variablen werden im Speicher angelegt. Für ein "char" wird 1 Byte reserviert.

GAS LISTING simple_add_char.s

```
 1                              .file       "simple_add_char.c"
 2                              .version    "01.01"
 3              gcc2_compiled.:
 4              .text
 5                              .align 16
 6              .globl main
 7                              .type        main,@function
 8              main:
 9 0000 55                                   pushl %ebp
10 0001 89E5                                 movl %esp,%ebp
11 0003 A0010000                             movb b,%al
11      00
12 0008 8A150200                             movb c,%dl
12      0000
13 000e 88C1                                 movb %al,%cl
14 0010 00D1                                 addb %dl,%cl
15 0012 880D0000                             movb %cl,a
15      0000
16              .L2:
17 0018 89EC                                 movl %ebp,%esp
18 001a 5D                                   popl %ebp
19 001b C3                                   ret
20              .Lfe1:
21                             .size        main,.Lfe1-main
22                             .local       a
23                             .comm        a,1,1
24                             .local       b
25                             .comm        b,1,1
26                             .local       c
27                             .comm        c,1,1
28 001c 8D742600               .ident       "GCC: (GNU) 2.95.3 20010315 (SuSE)"

DEFINED SYMBOLS
                     *ABS*:00000000 simple_add_char.c
  simple_add_char.s:3      .text:00000000 gcc2_compiled.
  simple_add_char.s:8      .text:00000000 main
  simple_add_char.s:23      .bss:00000001 b
  simple_add_char.s:25      .bss:00000002 c
  simple_add_char.s:21      .bss:00000000 a

NO UNDEFINED SYMBOLS
```

Variablen und Zuweisungen

```
C-Programm
"simple_add_int.c"

static int a, b, c;
int main()
{
a=b+c;
}
```

(globale) Variablen werden im Speicher angelegt. Für ein "int" werden 4 Byte reserviert.

GAS LISTING simple_add_int.s

```
 1                              .file   "simple_add_int.c"
 2                              .version        "01.01"
 3                      gcc2_compiled.:
 4                      .text
 5                              .align 16
 6                      .globl main
 7                              .type    main,@function
 8                      main:
 9 0000 55                              pushl %ebp
10 0001 89E5                            movl %esp,%ebp
11 0003 A1040000                        movl b,%eax
11      00
12 0008 8B150800                        movl c,%edx
12      0000
13 000e 8D0C02                          leal (%edx,%eax),%ecx
14 0011 890D0000                        movl %ecx,a
14      0000
15                      .L2:
16 0017 89EC                            movl %ebp,%esp
17 0019 5D                              popl %ebp
18 001a C3                              ret
19                      .Lfe1:
20                              .size    main,.Lfe1-main
21                              .local  a
22                              .comm   a,4,4
23                              .local  b
24                              .comm   b,4,4
25                              .local  c
26                              .comm   c,4,4
27 001b 908D7426                .ident  "GCC: (GNU) 2.95.3 20010315 (SuSE)"
27      00

DEFINED SYMBOLS
                     *ABS*:00000000 simple_add_int.c
  simple_add_int.s:3      .text:00000000 gcc2_compiled.
  simple_add_int.s:8      .text:00000000 main
  simple_add_int.s:24     .bss:00000004 b
  simple_add_int.s:26     .bss:00000008 c
  simple_add_int.s:22     .bss:00000000 a

NO UNDEFINED SYMBOLS
```

Zeiger und Vektoren (Pointer und Arrays)

Motivierendes Beispiel: Vertauschen von Werten in einer Liste.

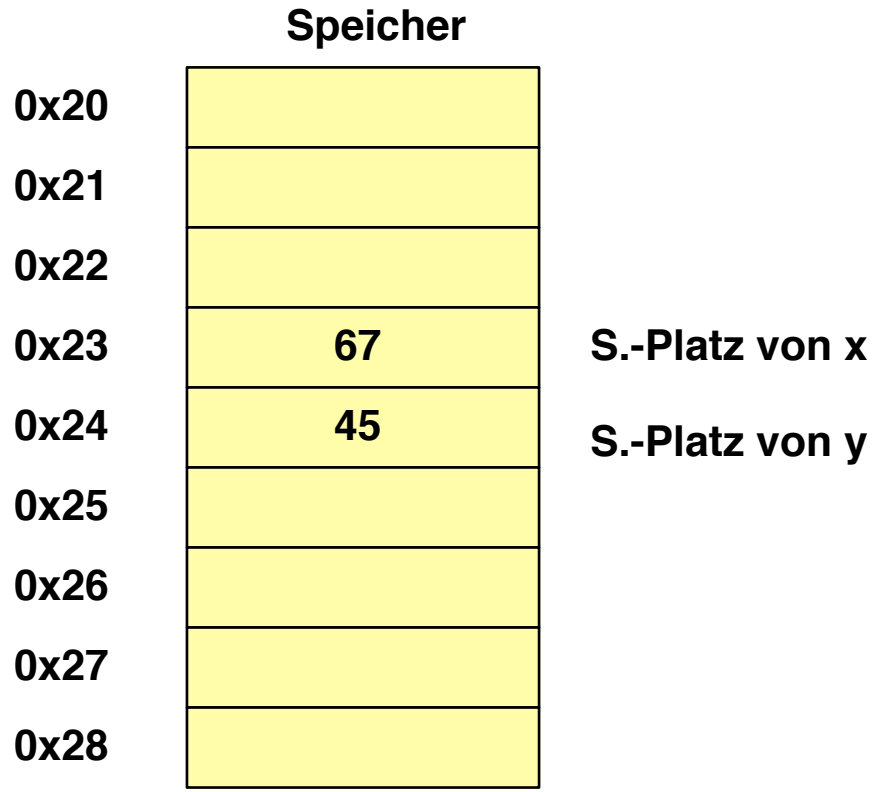
Swap-Funktion soll Inhalte beliebiger Listenplätze vertauschen.

swap(x,y); ?

Was sollen die Übergabe-Parameter x und y spezifizieren? Eine Funktion kann nur Werte (keine Adressen) übergeben.

Wie kann die Funktion die Argumente des aufrufenden Programms ändern?

1. **Lösung x und y sind globale Variablen.**
2. **Lösung das aufrufende Programm übergibt die Zeiger auf x und y.**



Zeiger und Vektoren (Pointer und Arrays)

Das Konzept des Pointers

C-Notation: `pointer_auf_x = &x /*Deklaration einer Zeigervariablen */`

Der Adress-Operator "&" liefert die Adresse einer Variablen. &x liefert daher die Speicheradresse der Variablen x und NICHT den Wert der Variablen x!

Das Statement weist diese Adresse der Zeigervariablen "pointer_auf_x" zu.

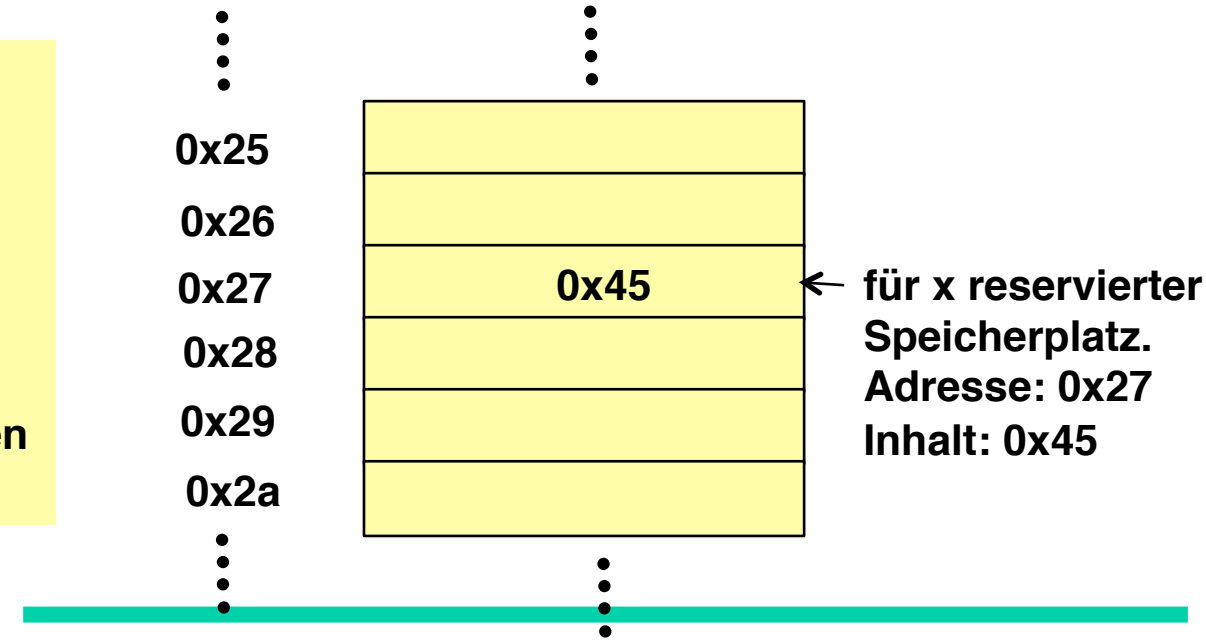
Beispiel:

Das C-Statement:

wert_von_x = x weist der Variablen wert_von_x den Wert "0x45" zu.

Das Statement:

pointer_auf_x = &x weist der Variablen pointer_auf_x den Wert 0x27 zu.



Zeiger und Vektoren (Pointer und Arrays)

Frage: Wenn ich die Adresse &x einer Variablen habe, wie greife ich dann auf ihren Wert zu?

Verweis-Operator "`*`":

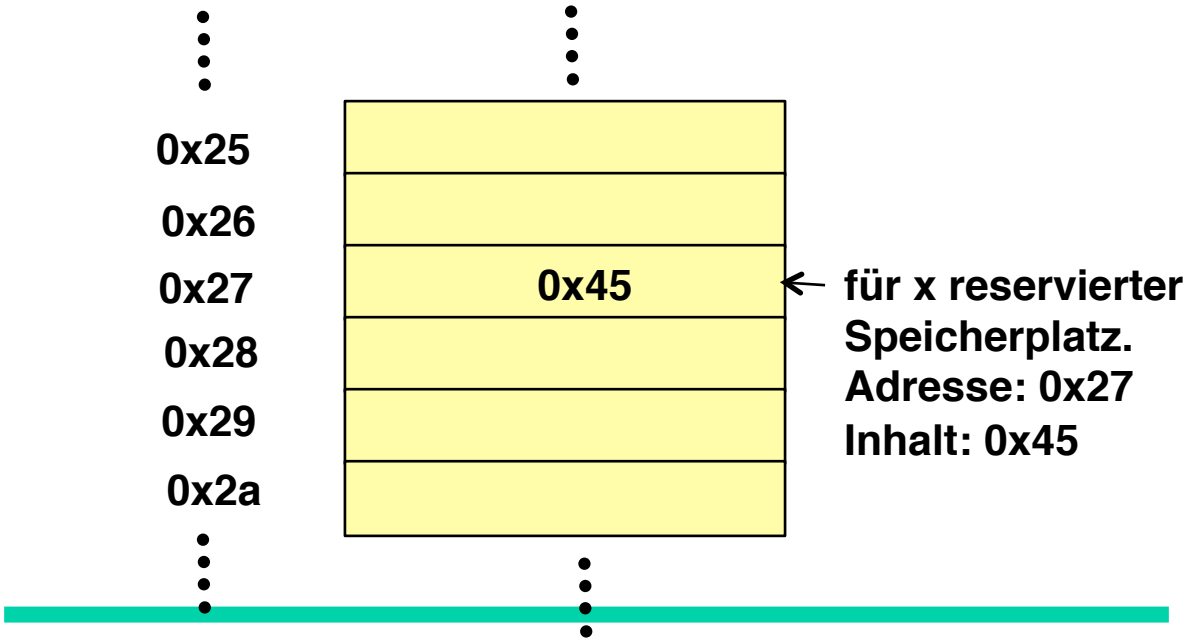
Mit `y= *pointer_auf_x` wird y der Wert auf den "`pointer_auf_x`"zeigt, zugewiesen.

Beispiel:

Das C-Statement:

```
pointer_auf_x   = &x
y = *pointer_auf_x
```

ist äquivalent zu `y = x`

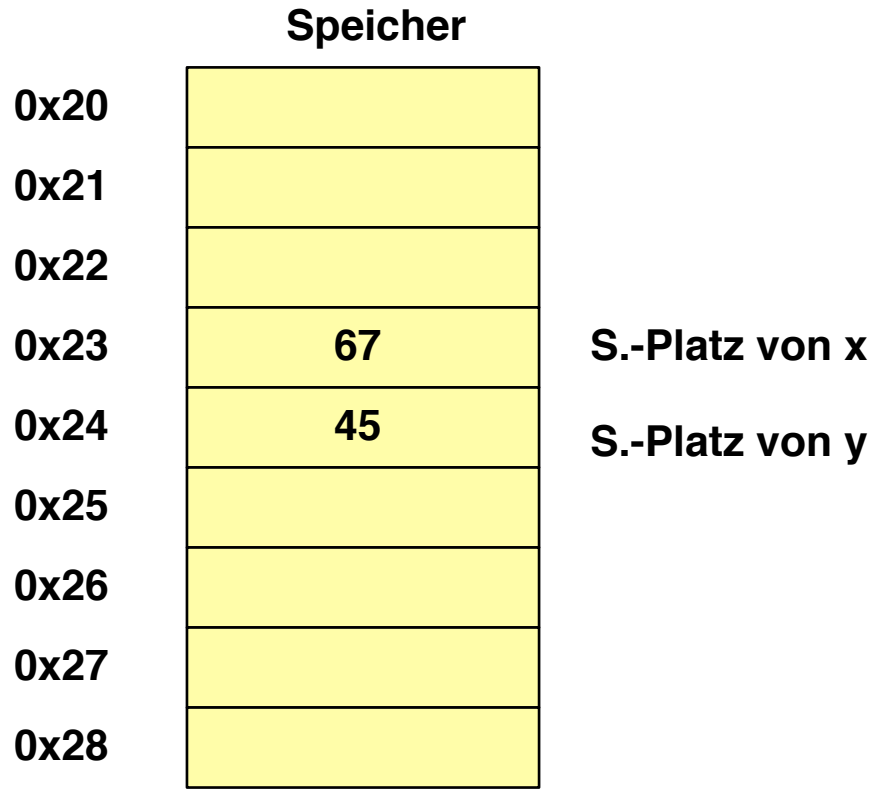


Zeiger und Vektoren (Pointer und Arrays)

Motivierendes Beispiel: Vertauschen von Werten in einer Liste.

```
swap (&a, &b);

.
.
.

swap(px, py)
int *px, *py;
{
     int temp;
     temp = *px;
     *px = *py;
     *py = temp;
}
```

als Pointer definiert



Zeiger und Vektoren (Pointer und Arrays)

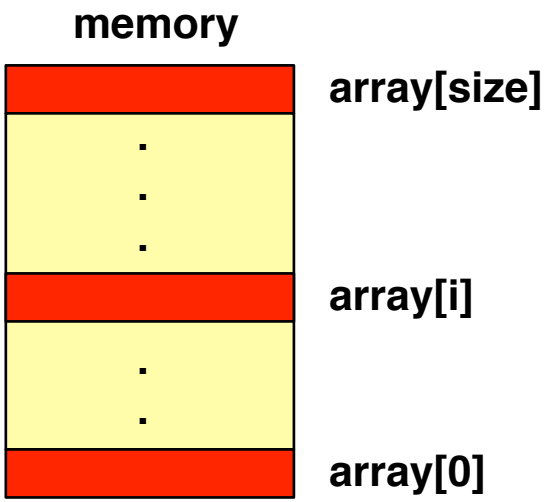
int a[10]

(Intel IA-32!)

	0	1	2	3	
x+0	a[1]	a[1]	a[1]	a[1]	x+3
x+4	a[2]	a[2]	a[2]	a[2]	x+7
x+8	a[3]	a[3]	a[3]	a[3]	x+11
x+12	a[4]	a[4]	a[4]	a[4]	x+15
x+16	a[5]	a[5]	a[5]	a[5]	x+19
x+20	a[6]	a[6]	a[6]	a[6]	x+23
x+24	a[7]	a[7]	a[7]	a[7]	x+27
x+28	a[8]	a[8]	a[8]	a[8]	x+31
x+32	a[9]	a[9]	a[9]	a[9]	x+35
x+36	a[10]	a[10]	a[10]	a[10]	x+39

Zeiger und Vektoren (Pointer vs. Arrays)

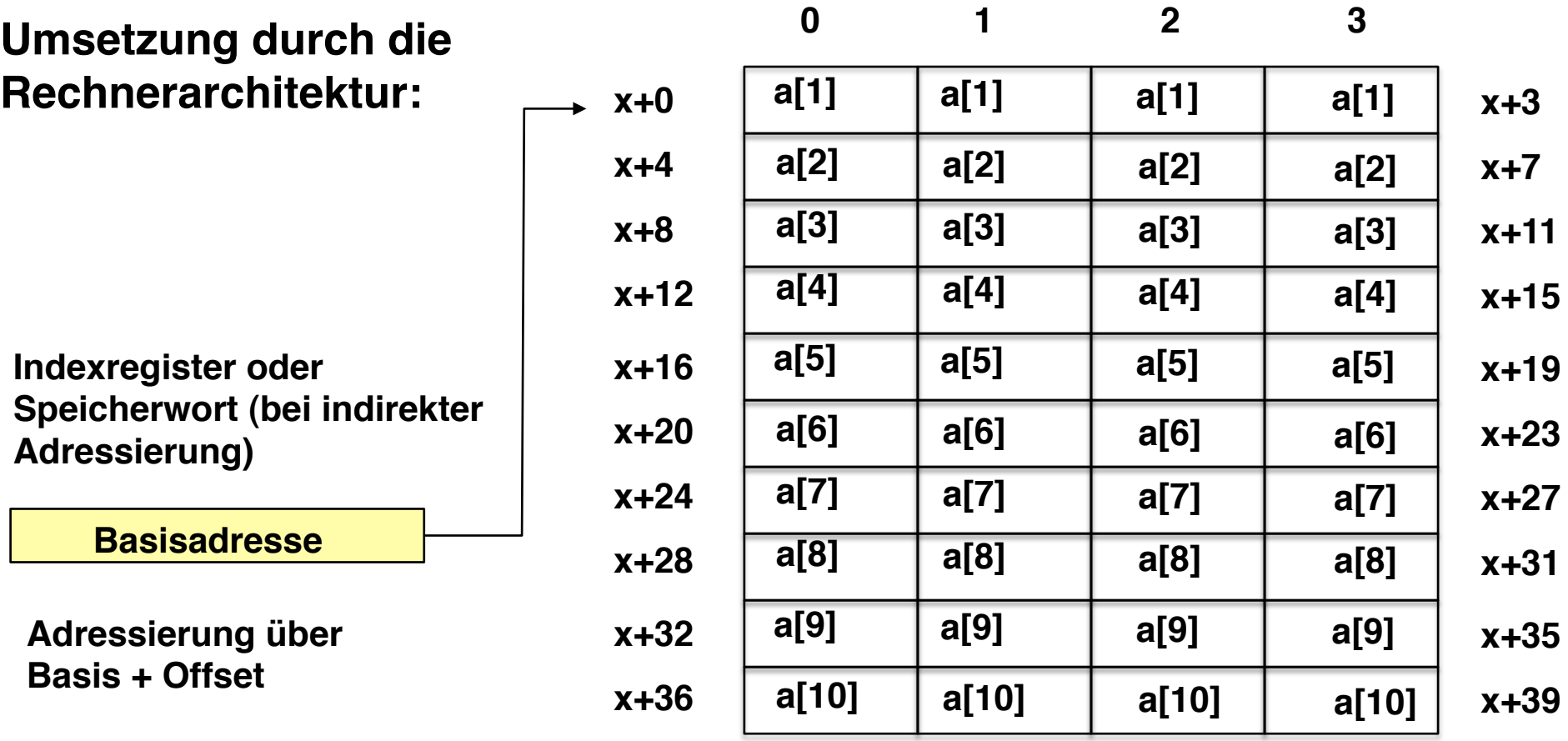
Programm zum Beschreiben des Arrays mit "0".




```
clear(int array[], int size;
{
    int i;
    for (i=0; i < size; i += 1)
        array[i] = 0;
}
```

Zeiger und Vektoren (Pointer und Arrays)

Umsetzung durch die Rechnerarchitektur:



Bei mehrdimensionalen Arrays kann die Adressierung über mehrere Indexregister, feste Offsets usw. erfolgen.

Structuren (Structures)

```
struct my_date{
int day;
int hoj;
long ssjcb;
char month;
int year;
long dsbb;
}
```

(Intel IA-32!)

Speicherlayout

	0	1	2	3
x+0	day	day	day	day
x+4	hoj	hoj	hoj	hoj
x+8	ssjcb	ssjcb	ssjcb	ssjcb
x+12	ssjcb	ssjcb	ssjcb	ssjcb
x+16				month
x+20	year	year	year	year
x+24	dsbb	dsbb	dsbb	dsbb
x+28	dsbb	dsbb	dsbb	dsbb

- Strukturen werden im Speicher angelegt
- Die Elemente werden meist auf Wortgrenzen angeordnet (alignment)
- Der Compiler kann mit festem Offset die Elemente der Struktur adressieren

Kontrollstrukturen

Kontrollstruktur-Muster:

- **IF . . . THEN . . .ELSE**
- **FOR K= I TO N DO**
- **WHILE . . . DO**
- **REPEAT . . . UNTIL**
- **CASE OF . . .**

Muster für Kontrollstrukturen

IF <Bedingung> THEN <Aktion>

Assembler Umsetzung:

```
CMP    a, b          a: Register A,B,D,S,U,X,Y    b:Speicheradresse
Bxx    EXIT          Branch on Condition
       •
       •             Aktion
       •
EXIT   Continue
```

Wahl der Sprungbedingung

```
IF <Bedingung> THEN <Aktion>

CMP   a, b        a: Register A,B,D,S,U,X,Y   b:Speicheradresse
Bxx   EXIT        Branch on Condition FALSE
        •
        •         Aktion
        •
EXIT  Continue
```

Bedingung	mit Vorz.	ohne Vorz.	Branch Mnemonic Bxx	
a = b	x	x	BNE	
a ≠ b	x	x	BEQ	
a > b	x		BLE	signed values
a ≥ b	x		BLT	
a < b	x		BGE	
a ≤ b	x		BGT	
a > b		x	BLS	
a ≥ b		x	BLO	unsigned values
a < b		x	BHS	
a ≤ b		x	BHI	

Muster für Kontrollstrukturen

```
IF  a<b THEN  aktion1  ELSE  aktion2
```

Assembler Umsetzung:

```
            CMP     a,b
            BGE     aktion2
              •
              •               Ausführung
              •               von aktion1
            BRA     EXIT
aktion2       •
              •               Ausführung
              •               von aktion2
 EXIT
```

Muster für Kontrollstrukturen

WHILE a>b DO aktion

Assembler Umsetzung:

```
REPEAT   CMP   a,b
         BLE   EXIT
           •
           •          Ausführung
           •          von aktion
         BRA   REPEAT
EXIT
```

Alternative Umsetzung:

```
           •
           •
         BRA   COMP
REPEAT     •
           •          Ausführung
           •          von aktion
COMP     CMP   a,b
         BGT   REPEAT
```

Muster für Kontrollstrukturen

REPEAT aktion UNTIL a=b

Assembler Umsetzung:

```
REPEAT
          .
          .                 Ausführung
                            von aktion
        LDB    a

        CMPB   b
        BNE    REPEAT
EXIT
```

```
          LDB    a
REPEAT
          .
          .                 Ausführung
                            von aktion

        CMPB   b
        BNE    REPEAT
EXIT
```

Muster für Kontrollstrukturen

```
FOR K = I TO J  DO aktion
```

Assembler Umsetzung:

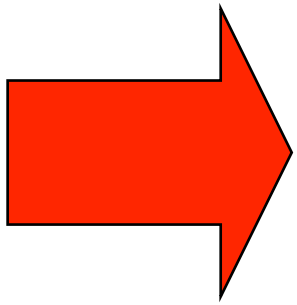
```
          LDB    #I
REPEAT
            .
            .              Ausführung
            .              von aktion
          INCB
          CMPB   #(J+1)
          BNE    REPEAT
EXIT
```

FOR k=i TO j DO body

Fragen??

darf i=j sein? wird die Schleife dann ein Mal oder kein Mal durchlaufen?
darf i>j sein? wird ein Wert mit Vorzeichen richtig behandelt?
darf i=j=0 sein?

Darf jeder Programmierer seine eigene FOR-Schleifen-Semantik programmieren?
Wie ist eine FOR-Schleife in einer Hochsprache definiert?



PASCAL
ISO/IEC 7185, Second Edition vom 15.10.90
Revision of ISO 7185 von 1983

PASCAL Definition der FOR-Schleife

for v:= e1 to e2 do body

shall be equivalent to

```
begin
temp1 := e1;
temp2 := e2;
if temp1 <= temp2
   begin
   v := temp1;
   body;
   while v <> temp2 do        /* v≠ temp2
      begin
      v:= succ(v);            /* v :=v+1
      body
      end
    end
end
```

for v:= e1 downto e2 do body

shall be equivalent to

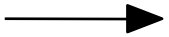
```
begin
temp1 := e1;
temp2 := e2;
if temp1 >= temp2
   begin
   v := temp1;
   body;
   while v <> temp2 do        /* v≠ temp2
      begin
      v:= pred(v);            /*  v :=v-1
      body
      end
    end
end
```

Assembler-Realisierung der Schleife: FOR k= i TO k DO body

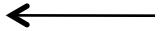
```
Test    LDA     tmp_1           (tmp_1 := i) Überprüfen der Gültigkeit der Schleifenparam.
        CMPA    tmp_2           (tmp_2 := k)
        BHI     invalid         Unsigned
//      (BGT    invalid)        Signed

//      (LDA    tmp_1)          im vorliegenden Fall liegt tmp_1 schon in A

loop    CMPA    tmp_2
        BEQ     exit
        ADDA    incv            bzw. INCA, wenn um „1“ erhöht wird
        •
        •
        •
        BRA     loop
exit    •
        •

tmp_1   RMB     1               Schleifenparam. werden dyn. berechnet
tmp_2   RMB     1
incv    RMB     1

// tmp_1 FCB    i               Schleifenparam. werden statisch festgelegt
// tmp_2 FCB    k
// incv  FCB    increment constant
```

Kontrollstrukturen: Case (Switch) Statement

```
CASE  k (k ≤ N)

I =0        aktion0
I =1        aktion1
I =2        aktion2
I =3        aktion3
            •
            •

I=N         aktionN
```



```
Assemblernotation:

aktion0  FDB    hh
                ll
aktion1  FDB    hh
                ll
aktion2  FDB    hh
                ll
aktion3  FDB    hh
                ll
```



Beginn der Einsprungtabelle

```
B enthält den CASE Index : k

                CMPB    #N              gültige Eingabe ?
                BHI     EXEPTION
                ASLB                    Berechng .d. Offsets in die Tabelle (ein Eintrag:2Byte)
                LDX     #aktion0
                ABX                     Addiere Offset in B zu X
                JMP     [  ,X]          Springe indirekt zum Anfang des Progr.-Teils “aktion k“
EXEPTION
```

Kontrollstrukturen Zusammenfassung:

IF a<b THEN aktion1 ELSE aktion2

```
          CMP     a,b
          BGE     aktion2
            •
            •                 Ausführung
            •                 von aktion1
          BRA     EXIT
aktion2     •
            •                 Ausführung
            •                 von aktion2
EXIT
```

WHILE a>b DO aktion

```
REPEAT          CMP     a,b
          BLE   EXIT
            •
            •                 Ausführung
            •                 von aktion
          BRA   REPEAT
EXIT
```

FOR K = I TO J DO aktion

```
          LDB     #i
REPEAT
            •
            •                 Ausführung
            •                 von aktion
          INCB
          CMPB    #(j+1)
          BNE     REPEAT
EXIT
```

REPEAT aktion UNTIL a=b

```
REPEAT
            •
            •                 Ausführung
            •                 von aktion
          CMP     a,b
          BNE     REPEAT
EXIT
```

Positionsunabhängiger Code

Positionsunabhängiger Code:
- **Ziele**
- **Mechanismen**

Ziele und Vorteile positionsunabhängigen Codes:

- ★ **Freie Verschiebbarkeit im Speicher**
- ★ **Nutzung von Objekt-Code Bibliotheken**
- ★ **Unabhängige Assemblierung**

Positionsunabhängiger Code

Positionsunabhängiger Code:
- **Ziele**
- **Mechanismen**

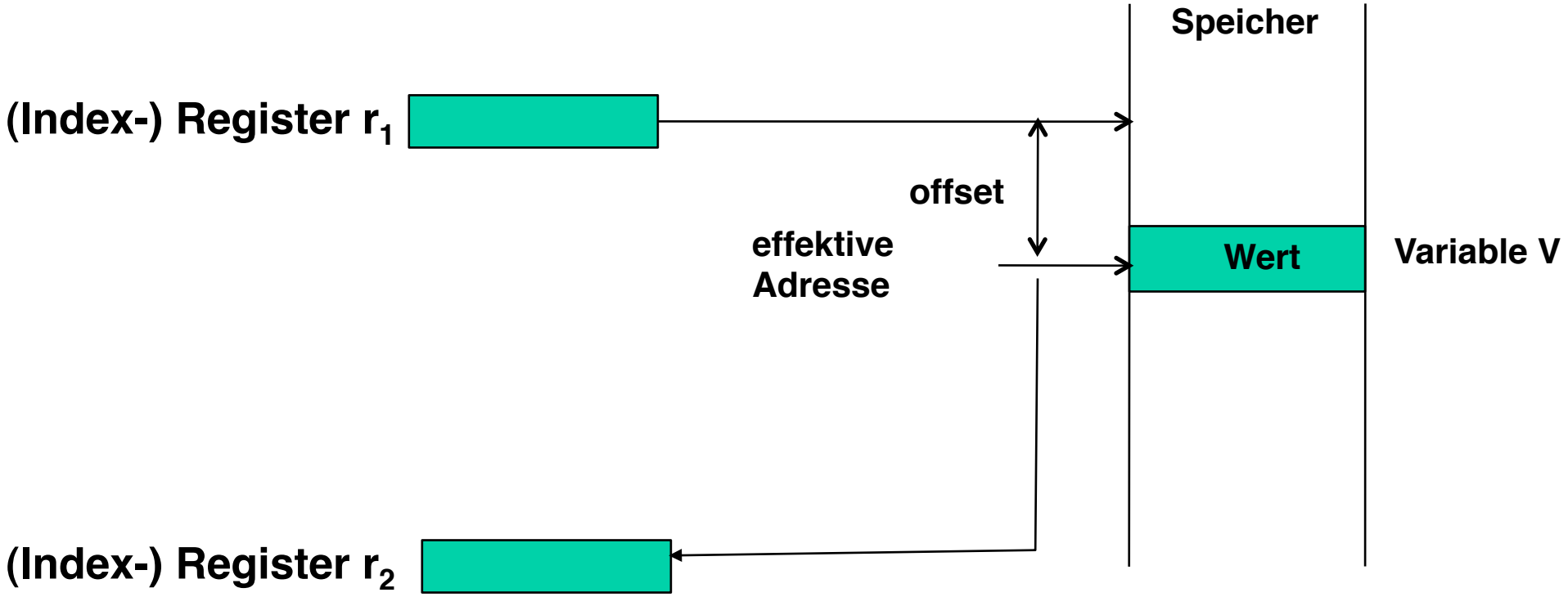
Mechanismen zur Unterstützung positionsunabhängigen Codes:

- **Relative Sprünge**
- **Ausnutzung der indizierten Adressierung**
- **Adressierung von Speicher durch “Konstante Distanz“ vom Programmzähler**
- **Nutzung des Hardware-Stacks als temporären Speicher**

Die Instruktion "Load Effective Address" LEA

LEA DST ,SRC (DST: Zielregister, SRC: Quellregister, Adr - Modus)

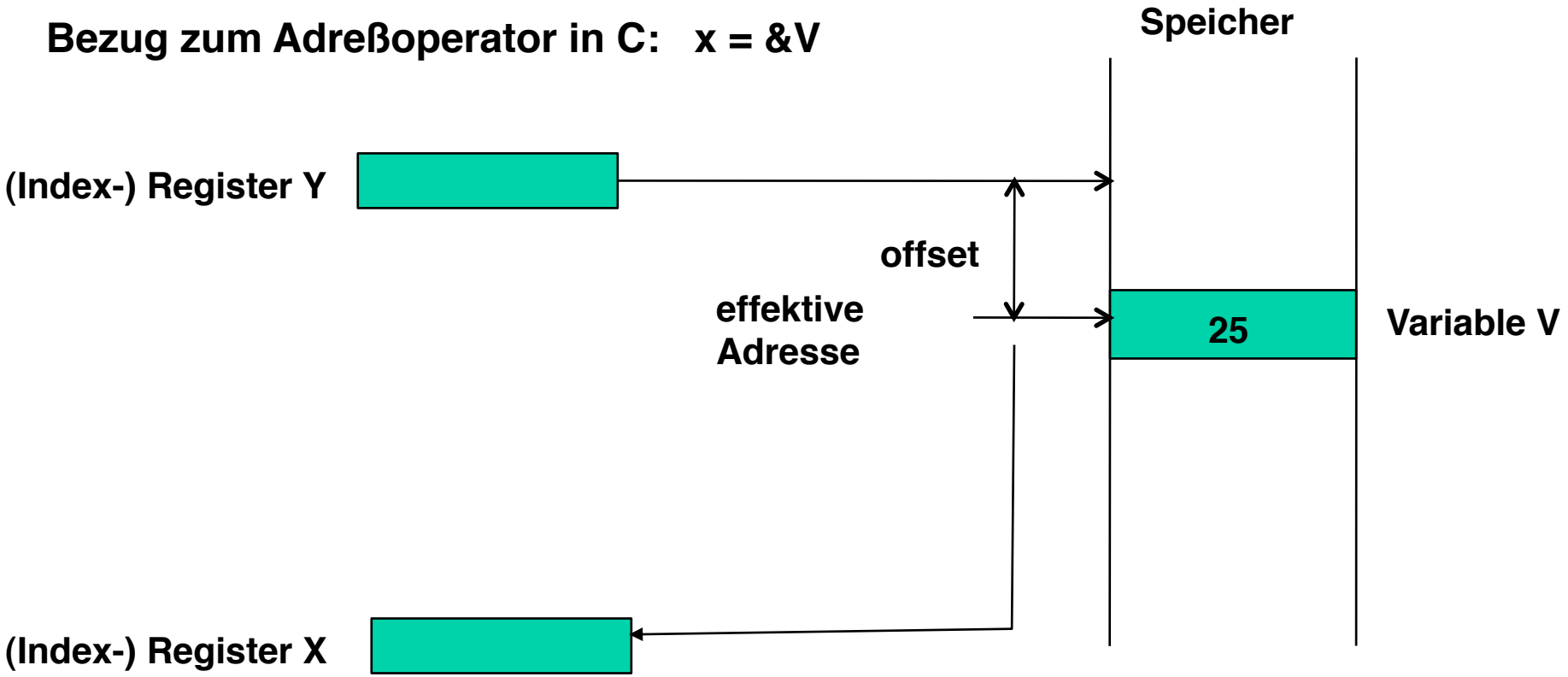
LEA(r_2) offset ,r_1 (Offset, Skalierung etc. wird durch Adr-Modus bestimmt)



Bezug zum Pointerkonzept

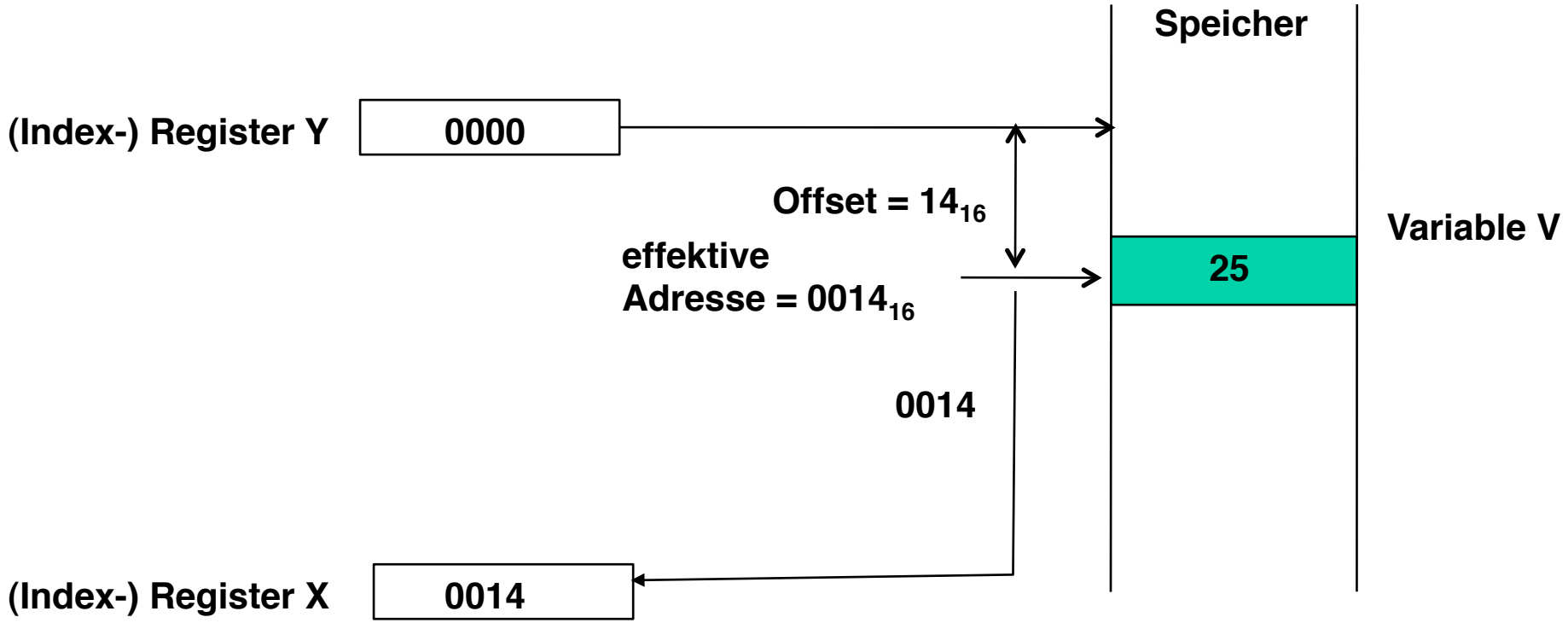
LEAX (offset, scale) ,Y

Bezug zum Adreßoperator in C: x = &V



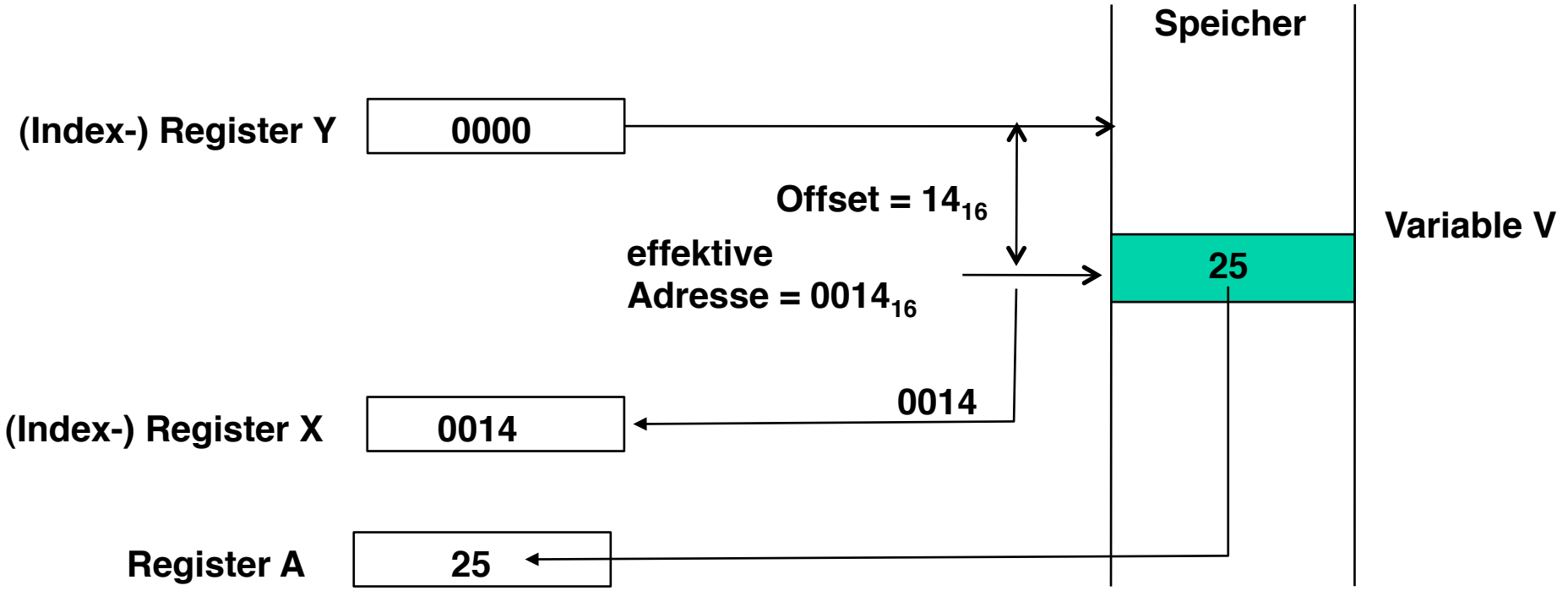
Bezug zum Pointerkonzept (Beispiel)

```
int V[10];          deklariert ein integer array (int ist ein 32-Bit Wert)
...
...
x = &V[5];                    LEAX    (5•4) ,Y    (offset•scale),Y
.....
```



Bezug zum Pointerkonzept (Beispiel)

```
int V[10];          deklariert ein integer array (int ist ein 32-Bit Wert)
...
x = &V[5];
a = *x;                          LEAX   (5•4) ,Y    (offset•scale),Y
.....                            LDA            ,X
```



Die Instruktion : LEA (Register) - Load Effective Address

- unterstützt arithmetische Operationen auf den Adreßregistern X,Y,S,U

LEAX, LEAY, LEAS, LEAU lädt nicht den Operanden, auf den die Adresse zeigt, sondern die Adresse selbst !

Beisp.:

```
•    LEAX    1,X       Increment X
•    LEAY    -1,Y      Decrement Y
•    LEAU    $ABCD,U   Addiere $ABCD zum U-SP
•    LEAX    0,PC      äqu. zu TFR  PC,X
```

- unterstützt positionsunabhängigen Code

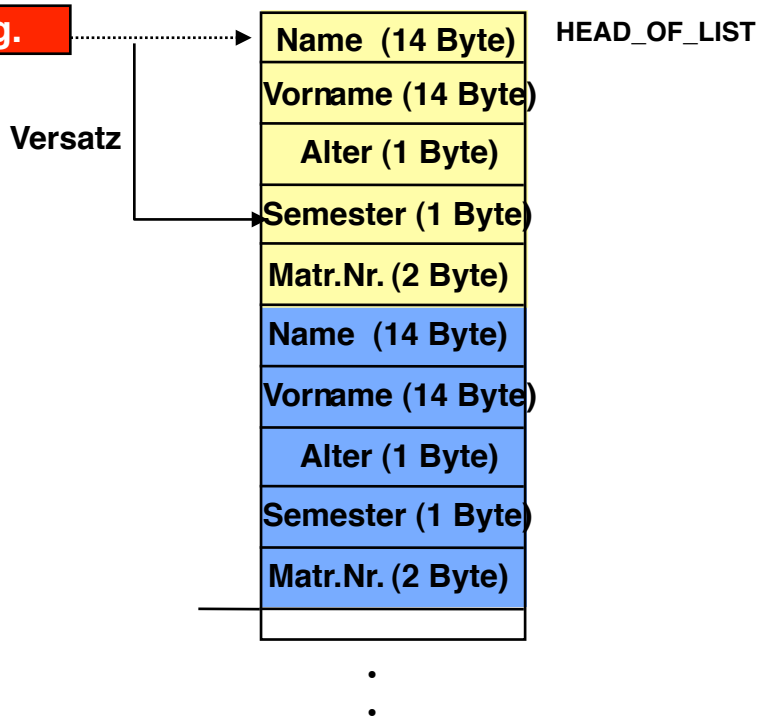
Beisp.:

```
          Addr.  OPCODE     OP-Addr.  Mnemonic
0100      30     8D         0109      START  LEAX   TABLE ,PCR
0104      A6     80                   LOOP   LDA    ,X+
0106                                          .
                                              .
020D                                  Table  FCC    /table of whatever/
```

Der Anfang von “Table“ hat einen Versatz von $10D vom START.
Der Assembler berechnet die Distanz $109, da er den Versatz vom um 4 incrementierten PC berücksichtigt. Durch die Asemblerdirektive “PCR“ wird der Befehl “LEAX TABLE,PCR“ vom Assembler in “LEAX offset,PC “d.h. 30 8D 01 09 übersetzt.

Beispiel für LEA

```
            LDX     HEAD_OF_LIST
            LDA     Sem_O
COMP        LDB     A ,X        Indiziert mit Register A Offset
            CMPB    S_zahl      Falls Semesterzahl >12:
            BHI     OUTPUT      Ausgabe: Name, Vorname
            LEAX    RL, X       Addieren der Record-Länge*
            CMPX    End_lst     Ende der Liste erreicht?
            BHI     Fin
            BRA     COMP
            .
            .
            .
RL          FCB     32          Record Länge
Sem_O       FCB     29          tatsächiiche Semester Anzahl
S_zahl      FCB     12          Regelstudienzeit
End_lst     FDB     $ABCD
```



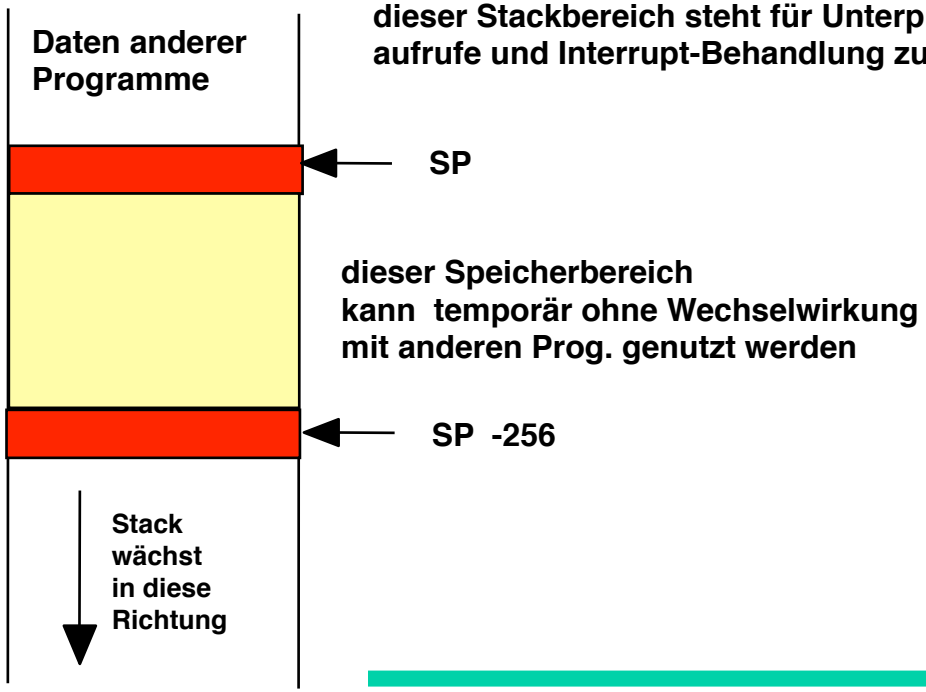
Nutzung des System Stacks als temporärer Speicher

Statische Zuordnung: (static allocation, Globale Variablen)	**Speicher wird fest reserviert, z.B. durch die Assemblerdirektiven : RMB, FCC, FCB, etc. Dieser Speicher kann während der Programmlaufzeit nicht (ohne Gefahr) anderweitig genutzt werden. Auch wenn das entpr. Programm nicht aktiv ist, ist es gefährlich, diesen Speicherbereich zu benutzen, da ein Aktivieren des Programms die dort abgelegten Daten anderer Programme überschreibt.**
Dyn. Zuordnung: (dynamic allocation. Lokale Varibalen)	**Speicher wird während der Laufzeit genutzt und dann wieder freigegeben. Die Nutzung zertört keine Daten von anderen Programmen.**

Stack als temporärer, dyn. zugeordneter Speicher:

LEAS -256,S reserviert den Speicher

LEAS 256,S gibt den Speicher frei



dieser Stackbereich steht für Unterprogramm-aufrufe und Interrupt-Behandlung zur Verfügun

dieser Speicherbereich kann temporär ohne Wechselwirkung mit anderen Prog. genutzt werden

Prozessornahe Programmiertechniken

Unterprogrammtechniken:

- **Unterprogrammaufruf**
- **Abspeichern des aktuellen Programmstatus**
- **Parameter Übertragung**
- **Rekursive Aufrufe (re-entrant Code)**

Unterprogrammtechniken

- Generell
- Unterprogrammaufruf
- Abspeichern des aktuellen Programmstatus
- Parameter Übertragung
- Rekursive Aufrufe (reentrant Code)

Unterprogramme, Subroutinen, Funktionen

Ziele:

- Wiederverwendung von Programmen oder Programmstücken
- Reduzierung des Speicherbedarfs durch Code-Sharing
- Modulare Strukturierung nach funktionalen Gesichtspunkten
- Verbesserung der Testbarkeit
- Unterprogrammbibliotheken

Architektur- Unterstützung beim Unterprogramm-Aufruf:

- Automatische Speicherung der Rücksprungadresse
- Mögliche Speicherung von Teilen oder des gesamten Prozessorstatus (Register)
- Unterstützung bei der Isolierung von aufrufendem und aufgerufenen Programm
- Unterstützung bei der Parameterübergabe
- Automatische Rückkehr an die entprechende Stelle im aufrufenden Programm
- Unterstützung rekursiver Unterprogrammaufrufe

Generelle Form:

Haupt-Programm

```
                .
                .
                .
JSR      FUNCT      Unterprogrammaufruf (PC wird automatisch gesp.)
 next_Instr         Rücksprungadresse
                .
                .
```

Unter-Programm

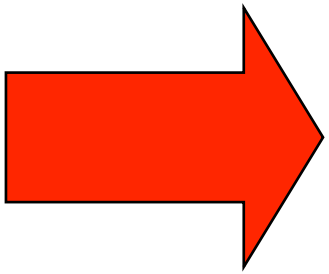
```
FUNCT   PSHS        Save Registers
        RegLst      specified in the Postbyte of PSHS Instruction
          .
          .
        PULS        Restore Registers
        RegLst      specified in the Postbyte of PULS Instruction
        RTS         Return from Subroutine to next_Instr
```

Unterprogrammtechniken

- Unterprogrammaufruf
- Abspeichern des aktuellen Programmstatus
- Parameter Übertragung (Parameter Passing)
- Rekursive Aufrufe (reentrant Code)

Abspeichern des Prozessorzustands

Abspeichern lokaler Variablen



Der Stack ist die bevorzugte Speicherstruktur zur "Rettung" des Programmstatus, da er Positionsunabhängigkeit und Schachtelung unterstützt

Unterprogrammtechniken

- Unterprogrammaufruf
- Abspeichern des aktuellen Programmstatus
- Parameter Übertragung (Parameter Passing)
- Rekursive Aufrufe (reentrant Code)

- **Grundsätzliche Möglichkeiten der Parameter Übertragung**
  - Call-by-value (einzige Option für C)
  - Call-by-name (Referenz)

- **Speicherstruktur für die Übertragung von Parametern:**
  - Register
  - Parameter in einem gemeinsamen Speicherbereich
  - Parameter Liste nach dem Unterprogrammaufruf
  - Stack

Problem: Positionsunabhängigkeit, Schachtelung von Unterprogrammen (rekursive UP)

Ausführung einer Mehrbyte-Addition in einem Unterprogramm: Parameterübergabe: Zeiger in Registern auf globale Variablen

Organisation der Daten im Speicher

S_1	0048	C3
	0049	A7
	004A	5B
	004B	2F
S_2	004C	B8
	004D	35
	004E	DF
	004F	14
SUM	0050	7B
	0051	DD
	0052	3A
	0053	44
N_Bytes	0054	04

Initialisierung

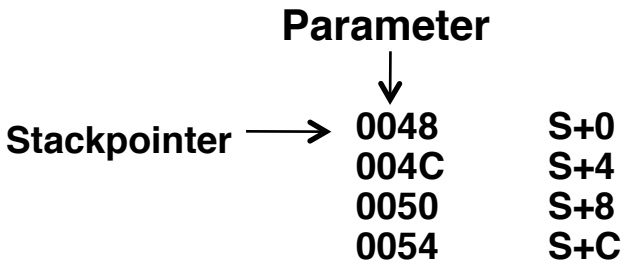
```
        LEAX    S_1
        LEAY    S_2
        LEAU    SUM
        LDB     N_Bytes
        JSR     MPADD
        .
        .
S_1     RMB     4
S_2     RMB     4
SUM     RMB     4
N_Bytes RMB     1
```

Ausführung

```
MPADD   ANDCC   #%11111110
ADBYTE  LDA     ,X+
        ADCA    ,Y+
        STA     ,U+
        DECB
        BNE     ADBYTE
        RTS
```

UP

Ausführung einer Mehrbyte-Addition in einem Unterprogramm: Parameterübergabe: lokale Variablen auf dem Stack



S_1	0048	C3
	0049	A7
	004A	5B
	004B	2F
S_2	004C	B8
	004D	35
	004E	DF
	004F	14
SUM	0050	7B
	0051	DD
	0052	3A
	0053	44
N_Bytes	0054	04

Initialisierung

```
LEAX   0 ,S
LEAY   2 ,S
LEAU   4 ,S
LDB    6 ,S
```

Ausführung

```
MPADD  ANDCC  #%11111110
ADBYTE LDA    ,X+
       ADCA   ,Y+
       STA    ,U+
       DECB
       BNE    ADBYTE
       RTS
```

UP

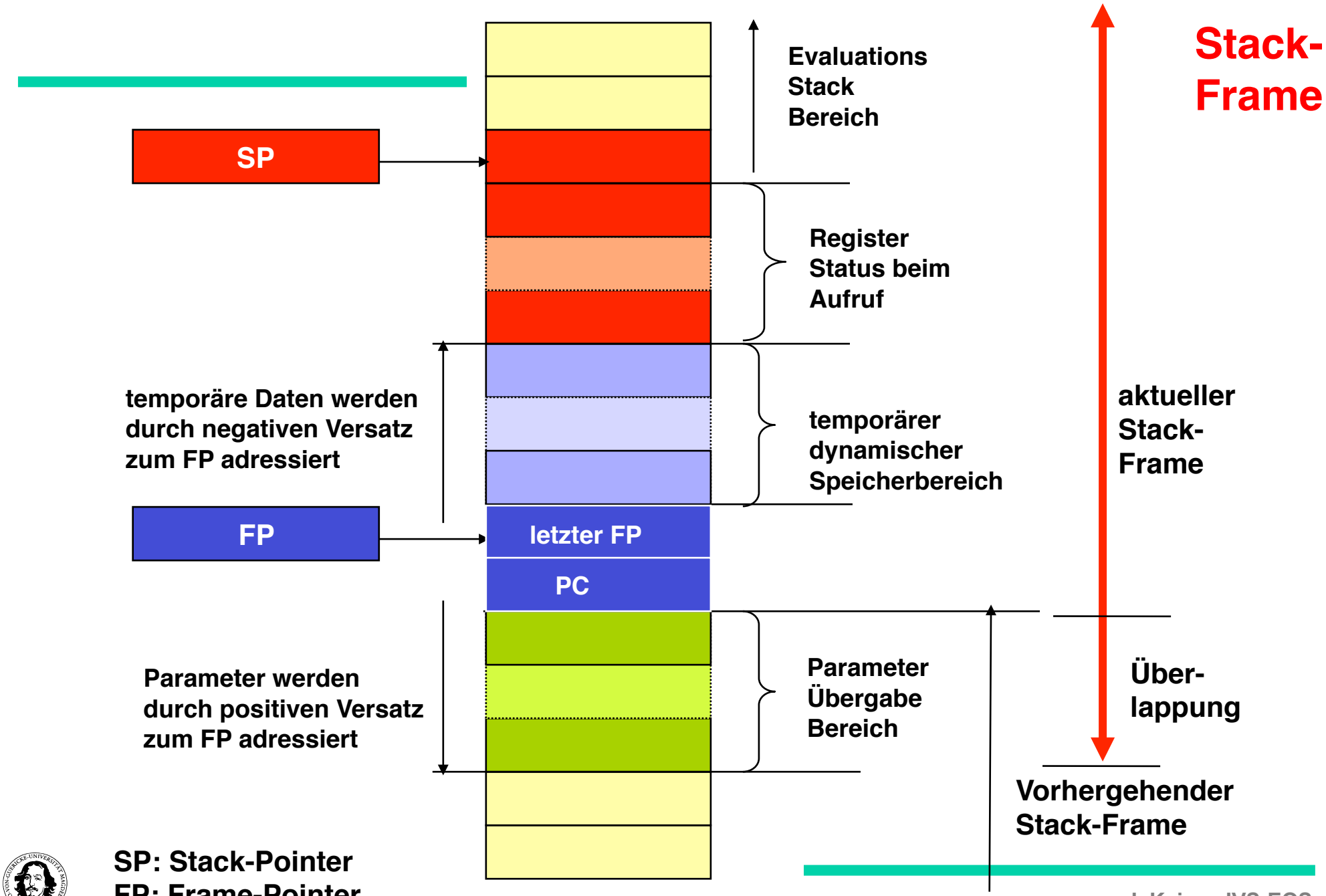
v: Versatz zum Stackpointer

Unterprogrammtechniken

- Unterprogrammaufruf
- Abspeichern des aktuellen Programmstatus
- Parameter Übertragung (Parameter Passing)
- Rekursive Aufrufe (re-entrant Code)

Schachtelung von Unterprogrammen erfordert:

- **Abspeichern des Programm-Status und der Rücksprungadressen in geordneter Weise**
- **Dynamische Zuordnung von temporärem Speicherplatz**
- **Übergabe der Parameter auf dem Stack**


Stack-
Frame
Evaluations
Stack
Bereich
SP
Register
Status beim
Aufruf
temporäre Daten werden
durch negativen Versatz
zum FP adressiert
temporärer
dynamischer
Speicherbereich
aktueller
Stack-
Frame
FP
letzter FP
PC
Parameter werden
durch positiven Versatz
zum FP adressiert
Parameter
Übergabe
Bereich
Über-
lappung
Vorhergehender
Stack-Frame
SP: Stack-Pointer
FP: Frame-Pointer

Beispiel für den Aufbau eines Prozedur-Rahmens (Stack-Frame)

```
#include "stdio.h"

int main()
{
int a, b, c;
b=3;
c=5;
a=b+c;
printf("%d\n",a);
}
```

```
GAS LISTING simple_add_int.s
...
...
  11                            main:
Vorbereiten des Stack-frames:
  12 0000 55                            pushl %ebp
  13 0001 89E5                          movl %esp,%ebp
  14 0003 83EC18                        subl $24,%esp

Bereitstellen der Parameter und Ausführen der Operation:
  15 0006 C745F803 000000       movl $3,-8(%ebp)
  16 000d C745F405 000000       movl $5,-12(%ebp)
  17 0014 8B45F8                movl -8(%ebp),%eax
  18 0017 8B55F4                movl -12(%ebp),%edx
  19 001a 8D0C02                leal (%edx,%eax),%ecx

Bereitstellen der Parameter für "printf":
  20 001d 894DFC                movl %ecx,-4(%ebp)
  21 0020 83C4F8                addl $-8,%esp
  22 0023 8B45FC                movl -4(%ebp),%eax
  23 0026 50                    pushl %eax
  24 0027 68000000 00           pushl $.LC0
  25 002c E8FCFFFF FF           call printf
  26 0031 83C410                addl $16,%esp

Wiederherstellen des ursprünglichen Stack-frames:
  27                            .L2:
  28 0034 89EC                  movl %ebp,%esp
  29 0036 5D                    popl %ebp
  30 0037 C3                    ret
```

Unterprogrammtechniken

Beispiel: Berechnung der Fakultätsfunktion: n ! für n ≤ 5

Diese Einschränkung wurde gemacht, weil bei größeren n eine aufwendige Mehrbyte-Multiplikation erforderlich wäre, was hier nicht Gegenstand des Problems ist.

Def. : $0! = 1,$
$n! = n \cdot (n-1)!$

Funktion FACT ist rekursiv definiert:

$FACT(0) = 1,$
$FACT(N) = N\ FACT(\ N-1\ )$

Unterprogrammtechniken

Rekursives C - Programm zur Berechnung der Fakultät

```
main ()
{
    printf ("die Fakultät von 5 ist: %d\n", fact (5));
}

int fact (int n)
{
    if (n < 1)
      return (1);

    else
      return (n * fact (n-1));
}
```

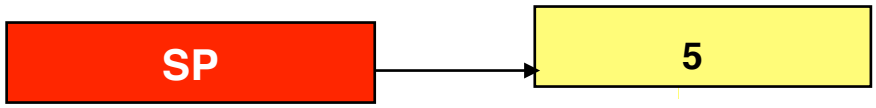
Rekursives Assembler - Programm zur Berechnung der Fakultätsfunktion

N: Parameter der Funktion FACT (N) S
F: Funktionswert S-1

```
        LEAX    ,S          Speichern des Stackpointers in Register X ( äquivalent zu : TFR   S,X )
        LEAS    1,S         Zuordnung eines temporären Speicherplatzes F  auf dem Stack
        LDA     #1          Initialisiere F mit 1
        STA     1 ,X
        LDA     ,X          Lade den Parameter N vom Stack
        INCA                Erhöhe um 1, um korrekte Abbruchbedingung zu erzeugen
        BSR     FACT        Springe zum Unterprogramm FACT
        LDA     1 ,X        "Retten" des Ergebnisses in A
        LEAS    ,X          Freigeben des temporären Speicherplatzes auf dem Stack
END     • • •

FACT    DECA                Decrementiere N
        BLE     RET         IF (N-1) ≤ 0 THEN RETURN
        PSHS    A           Speichere das aktuelle N des Unterprogramms auf dem Stack
        BSR     FACT        Rufe FACT zur Berechnung des nächsten N auf
        PULS    B           Lade ein N vom Stack nach B
        LDA     1 ,X        Lade das bisherige Produkt von F nach A
        MUL                 A•B
        STB     1 ,X        Speichere das Produkt (LSByte von Acc D) in F
RET     RTS
```

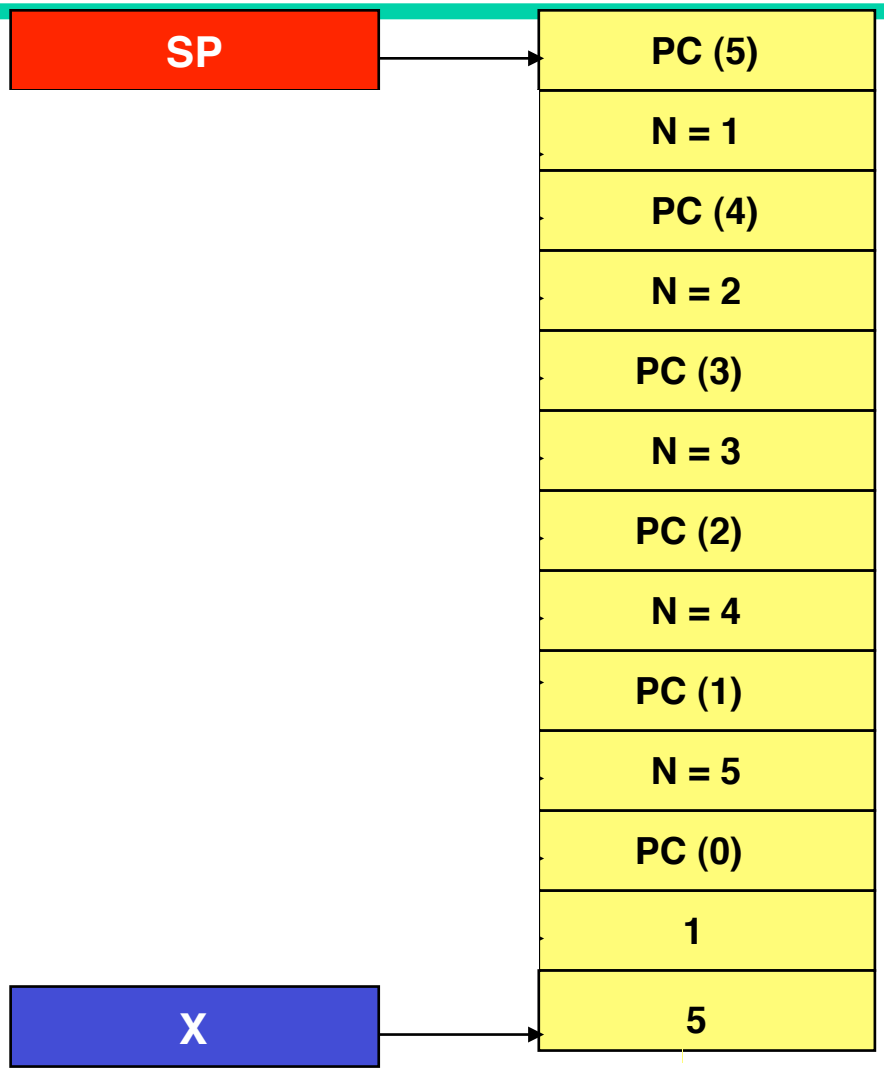
Unterprogrammtechniken

```
        LEAX    ,S
        LEAS    1,S
        LDA     #1
        STA     1,X
        LDA     ,X
        INCA
        BSR     FACT
        LDA     1 ,X
        LEAS    ,X
END     • • •


FACT    DECA
        BLE     RET
        PSHS    A
        BSR     FACT
        PULS    B
        LDA     1 ,X
        MUL
        STB     1 ,X
RET     RTS
```



Parameter N

Unterprogrammtechniken




```
        LEAX    ,S
        LEAS    1,S
        LDA     #1
        STA     1,X
        LDA     ,X
        INCA
        BSR     FACT
        LDA     1 ,X
        LEAS    ,X
END     • • •

FACT    DECA
        BLE     RET
        PSHS    A
        BSR     FACT
        PULS    B
        LDA     1 ,X
        MUL
        STB     1 ,X
RET     RTS
```

link: Link and Allocate

SP ← SP- 4	Dekrementieren des SP
SP ← An	Speichern des alten FP
An ← SP	Inhalt des SP wird neuer FP
SP ← SP + d	Allokation von Speicher d: negativer Versatz

MC 6809 Realisierung

```
PSHS     X
LEAX     ,S
LEAS   -d, S

PSHS   Register Liste
```

ulnk: Unlink

SP ← An	FP wird neuer SP
An ← [SP]	alter FP wird neuer FP
SP ← SP + 4	SP zeigt auf Adr. vor Link

MC 6809 Realisierung

```
PULS     Register Liste


TFR      X, S
LDX        , S
LEAS     2 , S
RTS
```

LINK

Link and Allocate
(M68000 Family)

LINK

Operation: SP – 4 → SP; An → (SP); SP → An; SP + d_n → SP

Assembler Syntax: LINK An, # < displacement >

Attributes: Size = (Word, Long*)

*MC68020, MC68030, MC68040 and CPU32 only.

Description: Pushes the contents of the specified address register onto the stack. Then loads the updated stack pointer into the address register. Finally, adds the displacement value to the stack pointer. For word-size operation, the displacement is the sign-extended word following the operation word. For long size operation, the displacement is the long word following the operation word. The address register occupies one long word on the stack. The user should specify a negative displacement in order to allocate stack area.

Condition Codes:

Not affected.

Instruction Format:

WORD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	0	REGISTER		
WORD DISPLACEMENT															

Instruction Format:

LONG

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	0	0	1	REGISTER		
HIGH-ORDER DISPLACEMENT															
LOW-ORDER DISPLACEMENT															

UNLK

Unlink
(M68000 Family)

UNLK

Operation: An → SP; (SP) → An; SP + 4 → SP

Assembler Syntax: UNLK An

Attributes: Unsized

Description: Loads the stack pointer from the specified address register, then loads the address register with the long word pulled from the top of the stack.

Condition Codes:

Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	1	REGISTER		

Instruction Field:

Register field—Specifies the address register for the instruction.

Lernziele

- Verständnis für die Unterstützung der Systemprogrammierung durch die Prozessorarchitektur.

- Grundlegende Daten- und Kontrollstrukturen und ihre Abbildung auf Assemblerebene:
  - Zuweisung, Arrays und Structures
  - Kontrollstrukturen

- Unterstützung positionsunabhängiger Adressierungsmethoden

- Unterprogrammtechniken:
  - Prozedurstatus
  - Parameterübergabe
  - Aufbau eines Prozedurrahmens auf dem Stack
  - Rekursive (und wiedereintrittsfähige (re-entrant)) Programme