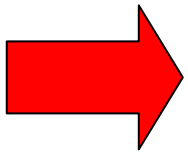
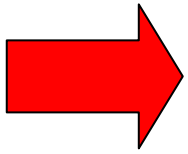
Prozessornahe Programiertechniken

Otto-von-Guericke-Universität Magdeburg

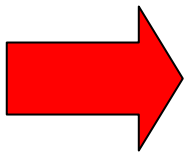
Prozessornahe Programmiertechniken



Kontrollstruktur-Muster



Positionsunabhängiger Code



Unterprogrammtechniken

Prozessornahe Programmiertechniken

Kontrollstruktur-Muster (Templates):

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

Prozessornahe Programmiertechniken

Positionsunabhängiger Code:

- **Relative Adressierung**
  - **Sprünge**
  - **Programmzähler-relative Adressierung**
- **Dynamische Zuordnung von temporärem Speicher durch Nutzung des System Stacks**

Prozessornahe Programmiertechniken

Unterprogrammtechniken:

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

Prozessornahe Programmiertechniken

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:

```
            LDB     #I
REPEAT
              •
              •             Ausführung
                            von aktion

            INCB
            CMPB    #(J+1)
            BNE     REPEAT
 EXIT
```

Muster für Kontrollstrukturen

FOR K = I TO J DO aktion

Assembler Umsetzung:

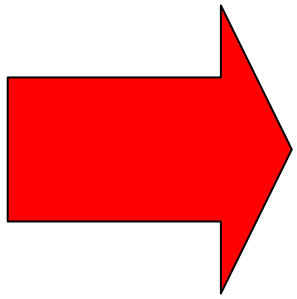
```
          LDB    #I
REPEAT
            •
            •            Ausführung
            •            von aktion
          INCB
          CMPB   #(J+1)
          BNE    REPEAT
EXIT
```

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
        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
        begin
        v:= pred(v);            /*  v :=v-1
        body
        end
     end
end
```

Assembler-Realisierung der Schleife: FOR k= i TO n 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 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

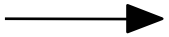
```
          LDB     #i
REPEAT
            •
            •              Ausführung
            •              von aktion
          INCB
          CMPB    #(j+1)
          BNE     REPEAT
EXIT
```

REPEAT aktion UNTIL a=b

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

Kontrollstrukturen: Case (Switch) Statement

```
CASE  k (k ≤ N)

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

I=N         aktionN
```



Assembler-Repräsentation

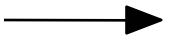
```
aktion0  FDB    hh
                ll
aktion1  FDB    hh
                ll
aktion2  FDB    hh
                ll
aktion3  FDB    hh
                ll
```

B enthält den CASE Index : x

```
                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 “aktion3“
EXEPTION
```

Kontrollstrukturen: Case (Switch) Statement

```
CASE  k (k ≤ N)

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

I =N        aktionN
```



Assembler-Repräsentation

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

B enthält den CASE Index : x

```
                CMPB    #N              gültige Eingabe ?
                BHI     EXEPTION
                ASLB                    Berechng .d. Offsets in die Tabelle (ein Eintrag:2Byte)
                LEAX    aktion0 ,PCR    Lade Indexregister mit Anfang d. Tabelle relativ zum PC
                JMP     [B  ,X]         Springe indirekt zum Anfang des Progr.-Teils "aktion3"
                                        (B wird Acc.Offset  automatisch zu X addiert)
EXEPTION
```

Hier wird die Tabelle relativ zum Programmzähler adressiert. Außerdem wird beim Sprung der automatische Accumulator Index ausgenutzt.

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**
- **Adressierung von Speicher durch “Konstante Distanz“ vom Programmzähler**
- **Nutzung des Hardware-Stacks als temporären Speicher**

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

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

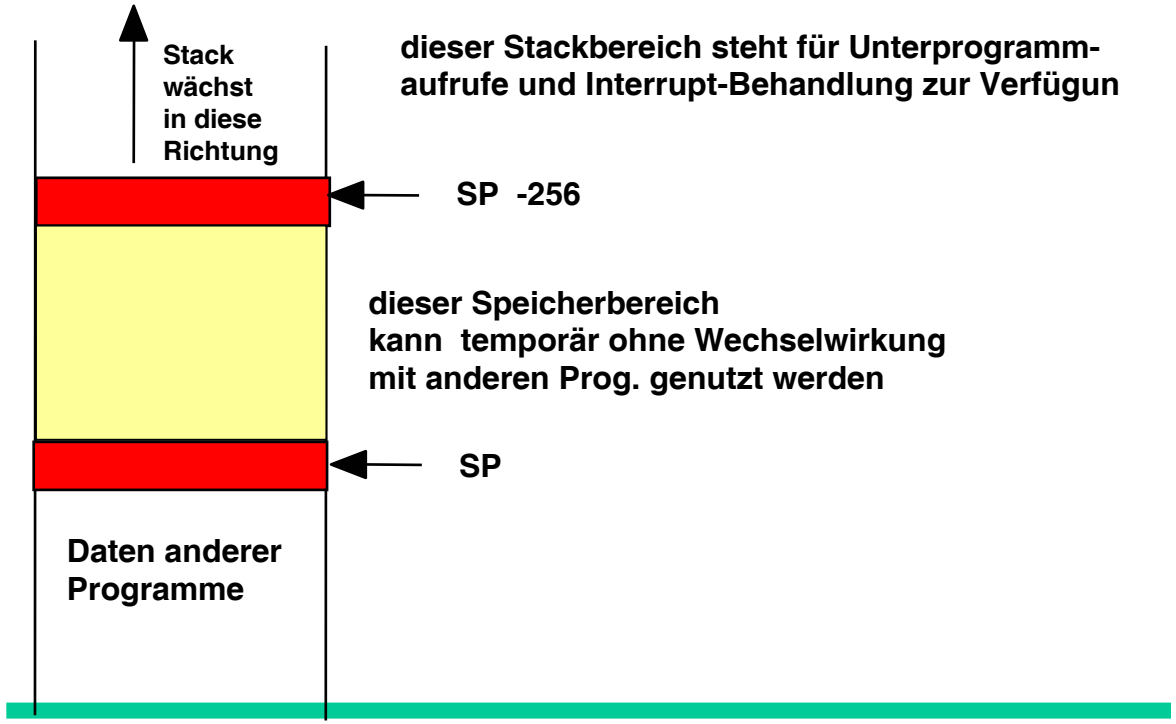
Der Anfang von “Table“ hat einen Versatz von $10D vom START.
Der Assembler berechnet die Distanz $109, da er den Versatz vom um 4 incrementierten PC berücksichtigt. Durch die Asemblerdirektive “PCR“ wird der Befehl “LEAX TABLE,PCR“ vom Assembler in “LEAX offset,PC “d.h. 30 8D 01 09 übersetzt.

Nutzung des System Stacks als temporärer Speicher

Statische Zuordnung: (static allocation)	**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)	**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
```



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

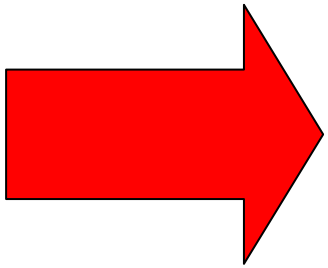
```
FUNCT   PSHS        Save Registers
        RegLst      specified in the Postbyte of PSHS Instruction
          •
          •
        PULS        Restore Registers
        RegLst      specified in the Postbyte of PULS Instruction
        RTS         Return from Subroutine to next_Instr
```

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)

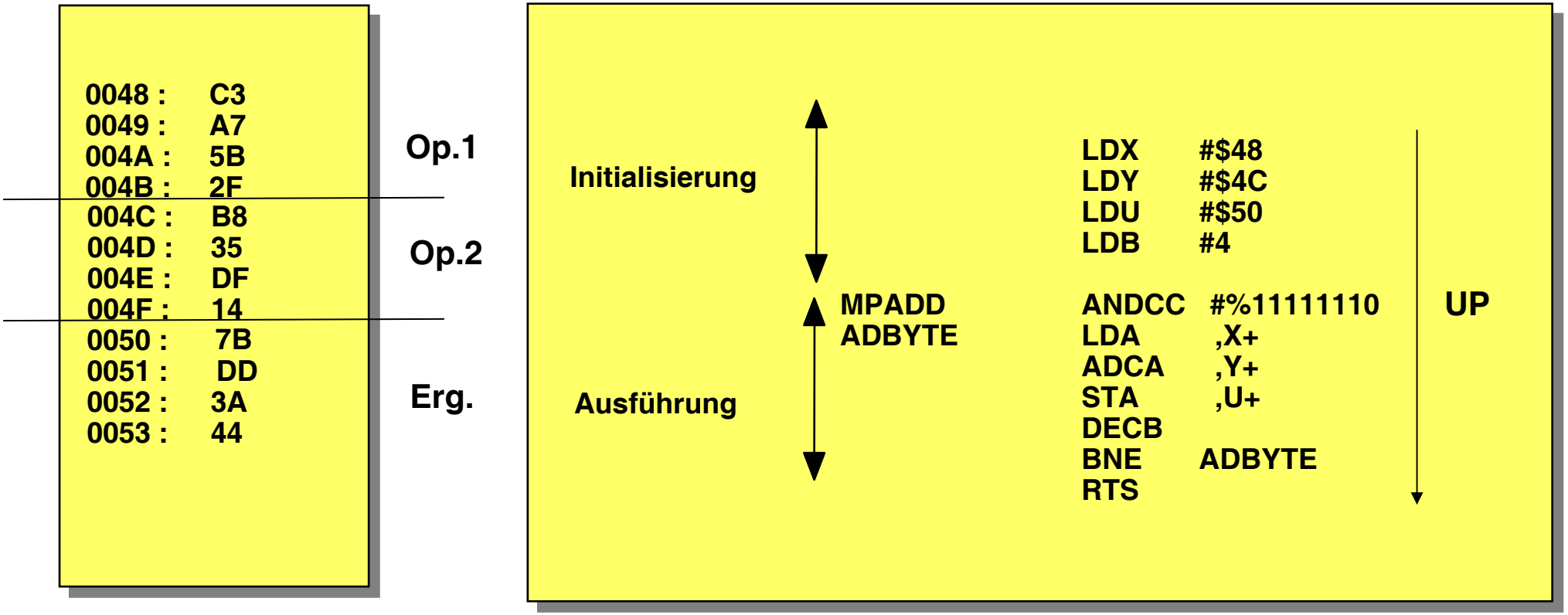
- **Grundsätzliche Möglichkeiten der Parameter Übertragung**
  - Call-by-value
  - Call-by-name

- **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)

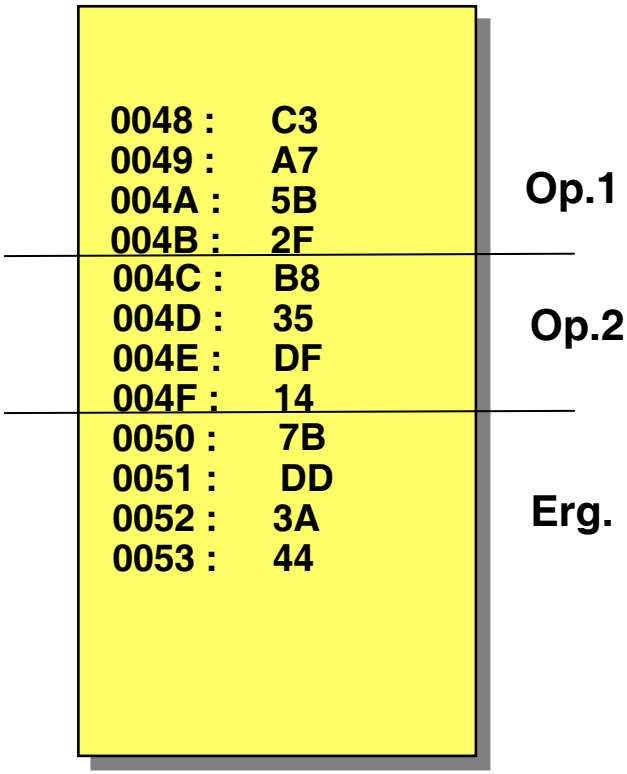
Unterprogrammtechniken

Ausführung einer Mehrbyte-Addition in einem Unterprogramm:
Parameterübergabe: Zeiger in Registern, statisch, implizites Wissen über gemeinsam benutzten Speicherbereich



Organisation der Daten im Speicher

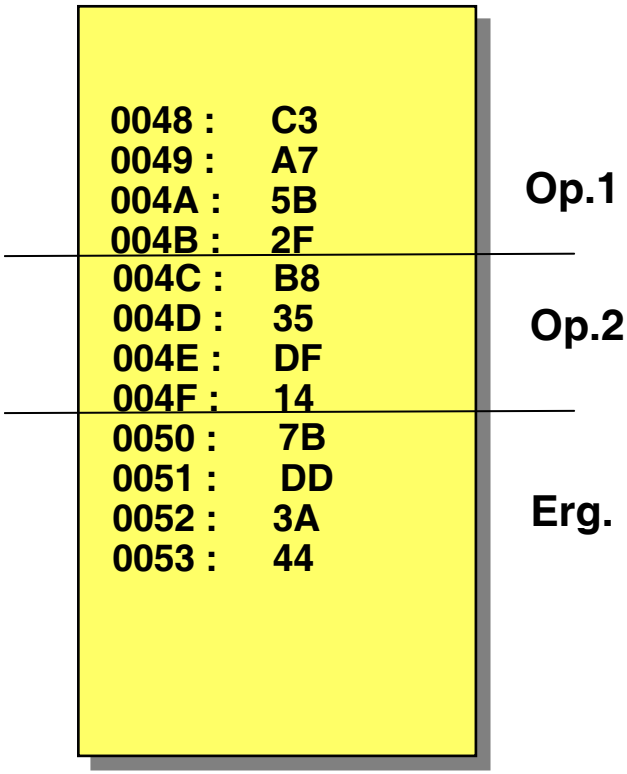
Ausführung einer Mehrbyte-Addition in einem Unterprogramm: Parameterübergabe: Zeiger in Registern (statisch)



Organisation der Daten im Speicher

```
Summand_1   EQU $48
Summand_2   EQU $4C
Summe       EQU $50
Anzahl_Bytes EQU  4
            •
            •
Initialisierung
            LDX     Summand_1
            LDY     Summand_2
            LDU     Summe
            LDB     Anzahl_Bytes
            JSR     MPADD
            •
            •
MPADD       ANDCC   #%11111110
ADBYTE      LDA     ,X+
Ausführung  ADCA    ,Y+
            STA     ,U+
            DECB
            BNE     ADBYTE
            RTS
                                  UP
```

Ausführung einer Mehrbyte-Addition in einem Unterprogramm: Parameterübergabe: Zeiger in Registern (dynamisch)



Organisation der Daten im Speicher

```
Initialisierung
                LDX     Summand_1
                LDY     Summand_2
                LDU     Summe
                LDB     Anzahl_Bytes
                JSR     MPADD
                 •
                 •
Summand_1       RMB     2
Summand_2       RMB     2
Summe           RMB     2

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

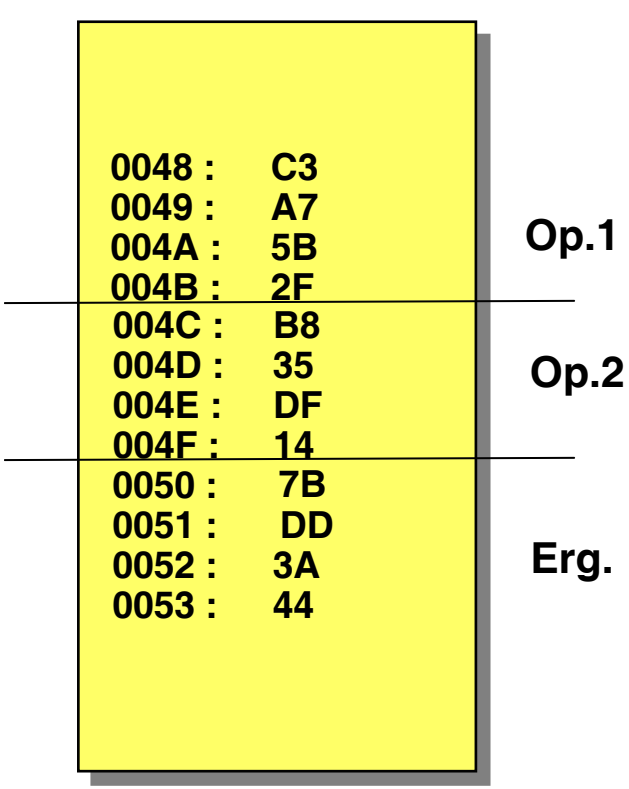
UP

Ausführung einer Mehrbyte-Addition in einem Unterprogramm:

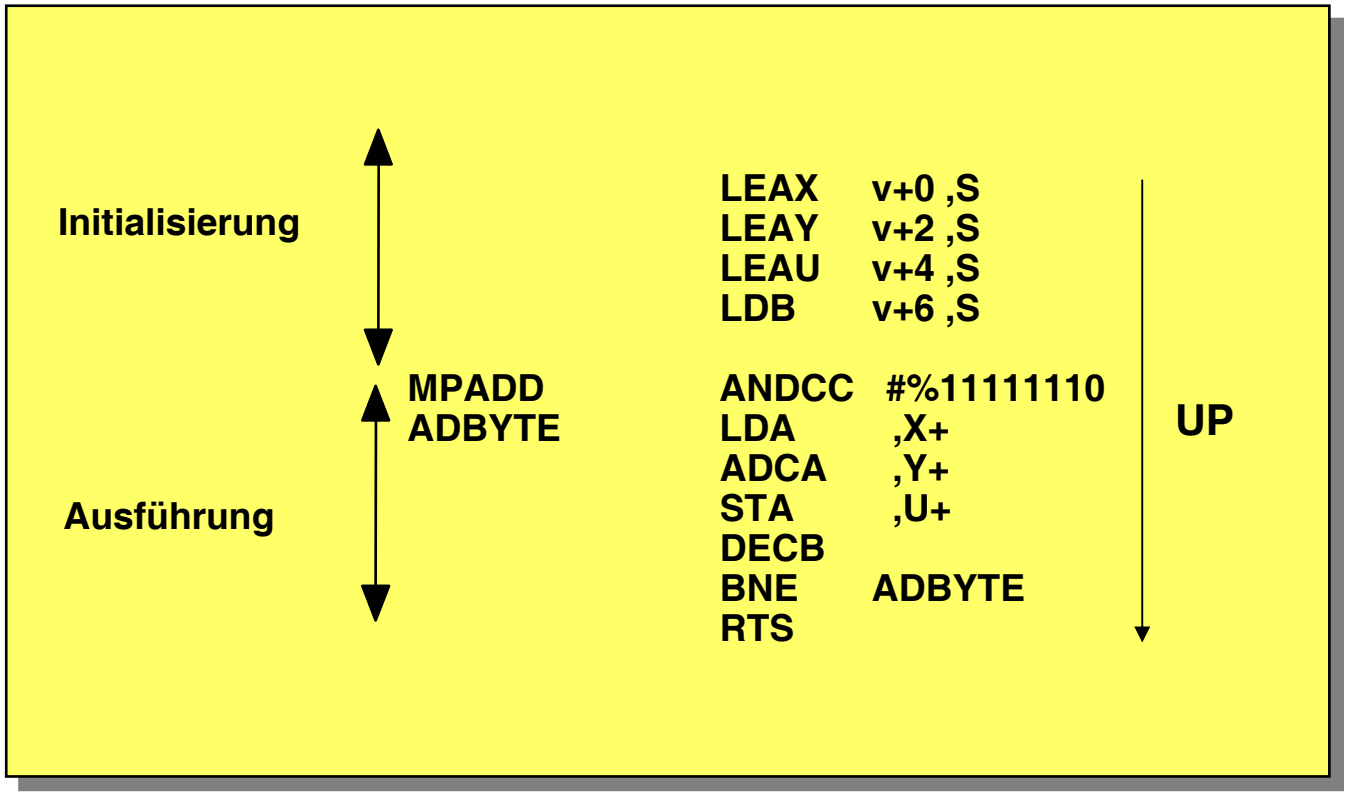
Parameterübergabe auf dem Stack (dynamisch)

Stack:

0048
004C
0050
0004



Organisation der Daten im Speicher



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**

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

Funktion FACT ist rekursiv definiert:

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

Unterprogrammtechniken

Rekursives C - Programm zur Berechnung der Fakultät

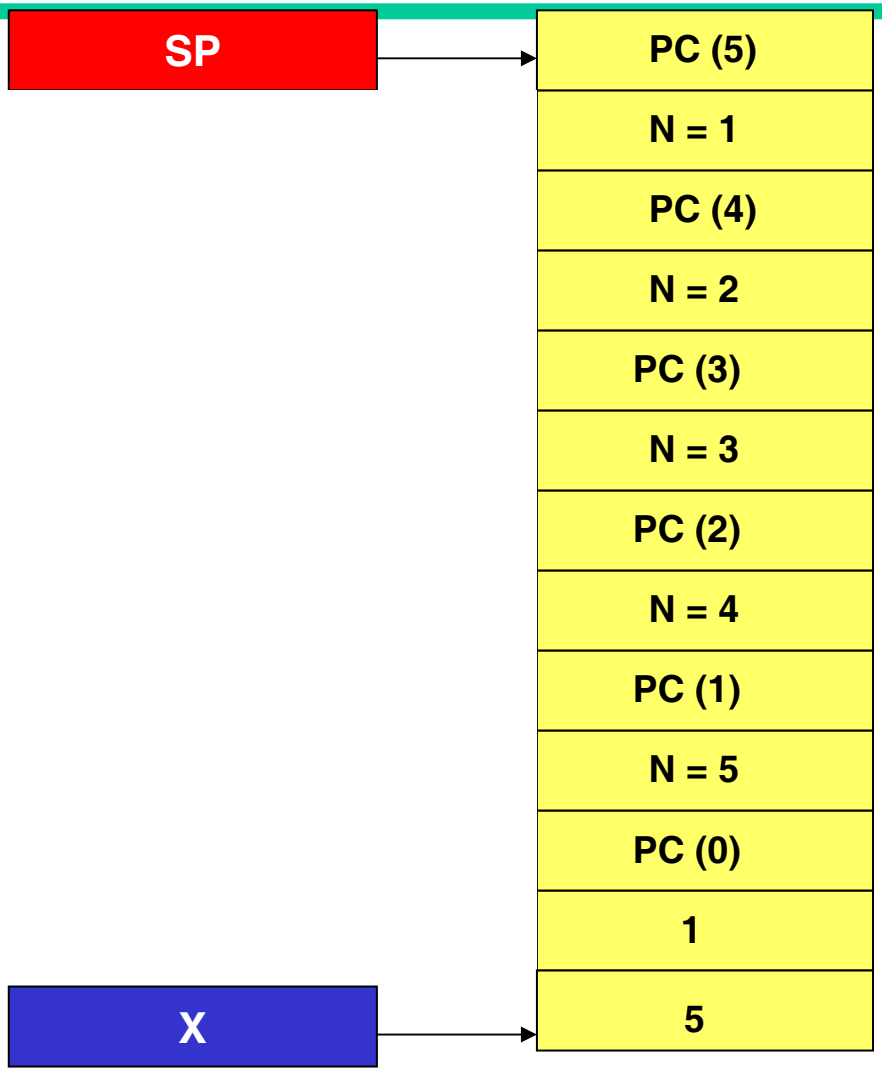
```
main ()
{
    printf ("die Fakultät von 5 ist: %d\n", fact (5));
}

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
```

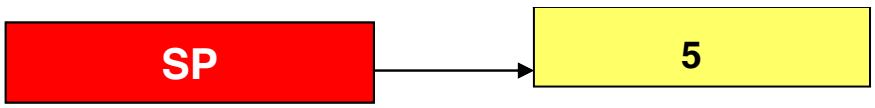
Unterprogrammtechniken




```
        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
```

Unterprogrammtechniken

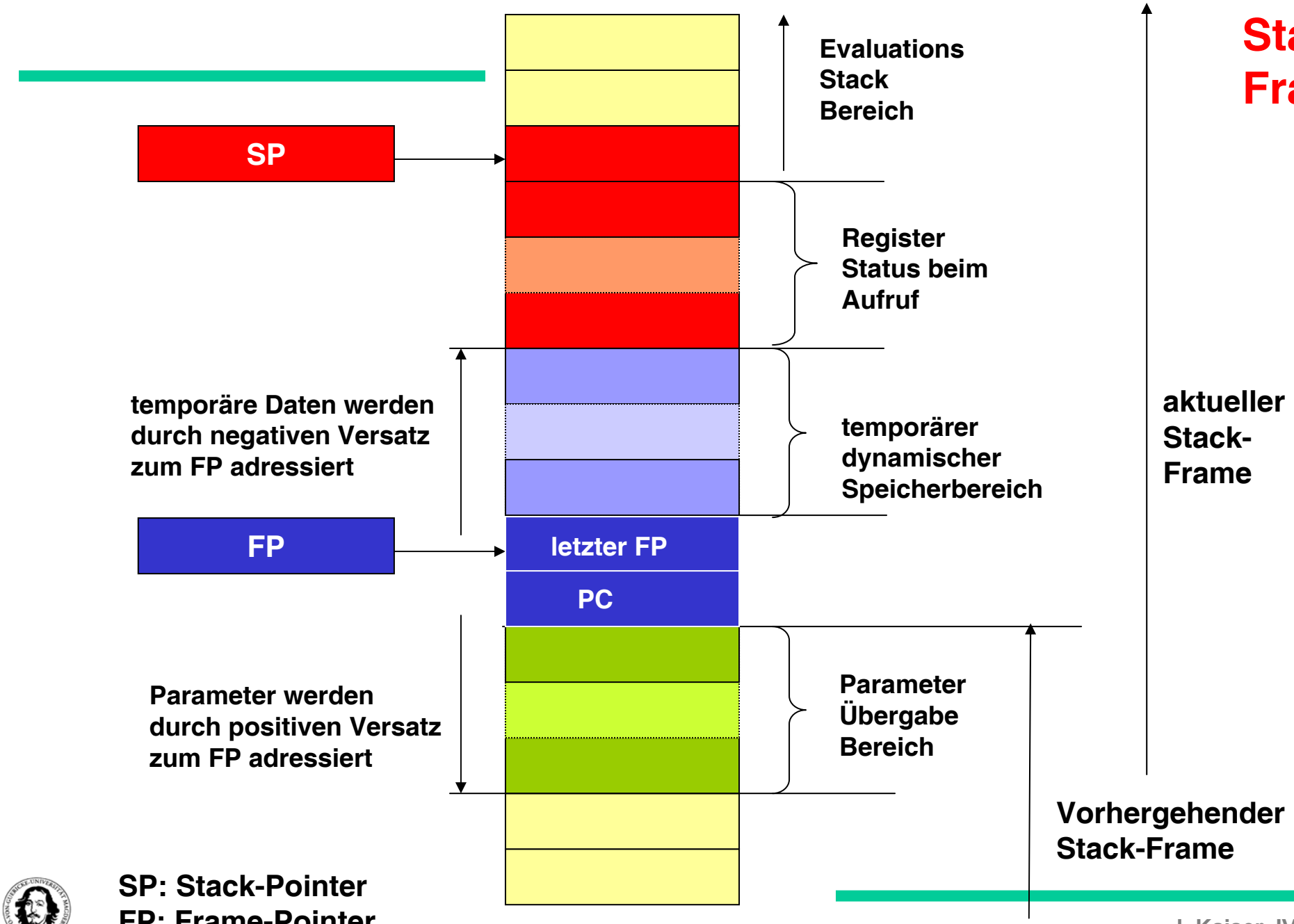
```
        LEAX    ,S
        LEAS    1,S
        LDA     #1
        STA     1,X
        LDA     ,X
        INCA
        BSR     FACT
        LDA     1 ,X
        LEAS    ,X
END     •••

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



TEMP

Parameter N

Stack-Frame



SP: Stack-Pointer
FP: Frame-Pointer

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.

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
```

Lernziele

- Verständnis für die Unterstützung der Systemprogrammierung durch die Prozessorarchitektur.

- Grundlegende Kontrollstrukturen und ihre Abbildung auf Assemblerebene:
  - if..then..else
  - for k=i to n do
  - while ... do
  - repeat... until
  - switch case

- Unterstützung positionsunabhängiger Adressierungsmethoden

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

Unterprogrammtechniken

Iteratives C - Programm zur Berechnung der Fakultät

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

int fact (int n)
{
    f = 1;
    i = 1;
    while (i++ < n)
            f = f * i;
    return (f);
}
```

Iteratives Assemblerprogramm zur Berechnung der Fakultät

N: Parameter der Funktion FACT (N) S
Z: Schleifenzähler S - 1
F: Funktionswert S - 2

```
                •
                •
                TFR     S, X        Nutze X als Basisregister zur Adressierung von N
                LEAS    -2, S       Zuordnung von 2 temp. Speicherplätzen für F auf dem Stack
                BSR     FACT
                PULS    A           Funktionswert in Register A retten
                LEAS    2 ,S        Stack "aufräumen"

FACT            LDA     #1          Initialisieren
                STA     -1 , X      des Schleifenzählers Z und
                STA     -2, X       (temporären) Funktionswertes F

REPEAT          LDB     -1, X       Laden des aktuellen Wertes des Schleifenzählers Z
                CMPB    ,X          Vergleich mit dem Funktionsparameter
                BGE     OUT         wenn  Z <  N ist, dann beenden mit Funktionswert = 1
                INCB                Erhöhen des Schleifenzählers
                STB     -1, X       und abspeichern
                LDA     -2 ,X       Laden des (temporären) Funktionswertes
                MUL                 Multiplikation A * B ; erhöhter Schleifenzähler * temp. Funktionswert
                STB     -2 ,X       Abspeichern des neuen (temporären) Funktionswertes
                BRA     REPEAT      Rücksprung zum Anfang der Schleife
OUT             RTS
```