

Kapitel 11

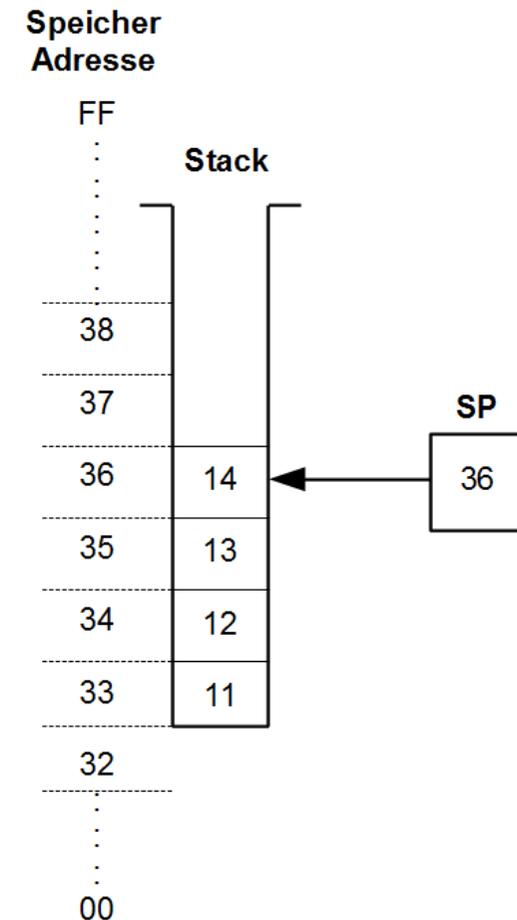
Unterprogramme

Unterprogrammtechnik

- **Funktionen** in der C-Programmierung dienen dem Zweck,
 1. ein größeres Programm übersichtlicher zu strukturieren,
 2. eine mehrfach gebrauchte Programmfunktion nur einmal im Programm zu haben, sie aber an mehreren Stellen im Programm zu verwenden.
- Die vergleichbare Komponente in der Assemblerprogrammierung ist das **Unterprogramm**.
- Das Unterprogramm ist eine abgeschlossene Folge von Befehlen innerhalb des Programms.
 - Um ein Unterprogramm an einer bestimmten Stelle im Hauptprogramm zu verwenden, wird es mit einem CALL Befehl aufgerufen.
 - Der Prozessor führt dann die Befehle des Unterprogramms aus.
 - Sobald das Ende des Unterprogramms erreicht ist, macht der Prozessor im Hauptprogramm mit dem Befehl weiter, der auf den CALL Befehl folgt.
 - Der Prozessor speichert bei Ausführung des CALL Befehls die Rückkehradresse auf dem Stack ab.

Stack

- Definition : **Stack** (= Stapel) : **LIFO Queue** (Last In First Out Warteschlange)
 - Beispiel eines Stacks : ein Stapel Blätter
 - Es gibt nur zwei mögliche Operationen :
 - Ein neues Blatt oben auf den Stapel legen.
 - Das oberste Blatt vom Stapel entfernen.
- Der **8051 Stack** ist ein Bereich im (ganzen) internen Datenspeicher, den der Programmierer fest legt.
 - Stack Operationen :
 1. Ein Byte an der ersten freien Stelle im Stackbereich ablegen.
 2. Das als letztes abgelegte Byte wegnehmen.
- Zum Betrieb des Stacks stellt der 8051 Prozessor das SFR **Stack Pointer (SP)** an der Adresse 0x81 zur Verfügung.
 - Der Stack Pointer zeigt immer auf das zuletzt auf dem Stack abgelegte Byte.
 - Beim Speichern eines Bytes in den Stack wird zuerst der Wert im StackPointer um 1 erhöht und dann das Byte an diese Adresse geschrieben.
 - Beim Auslesen wird das Byte von der Adresse gelesen, auf die der Stackpointer gerade zeigt, und anschließend der Stackpointer um 1 vermindert.



Stack (Forts.)

- Der Programmierer legt den **Anfang des Stackbereichs** fest (im Beispiel oben Adresse 0x33),
 - indem er am Anfang des Programms den Wert des Stack Pointers auf die gewünschte Adresse im internen Datenspeicher setzt (im Beispiel oben auf 0x32).
 - Der Stackbereich wird zunächst im Datenspeicher reserviert (z.B.: `STACK DS 40`),
 - und dann der Stackpointer auf den Anfang des Bereiches gesetzt (z.B.: `MOV SP,#STACK-1`).
- Im Reset-Vorgang wird der Stackpointer auf den **Anfangswert 7** initialisiert.
 - d.h. er zeigt auf das letzte Byte der Registerbank 0;
 - Solange das Programm den StackPointer nicht selbst initialisiert, beginnt der Stackbereich also direkt hinter der Registerbank 0.
- Die Größe des benutzten Stackbereichs ändert sich während der Laufzeit.
 - Der Stackbereich muss groß genug sein, dass immer alle zu speichernden Daten hinein passen.
 - Es liegt in der Verantwortung des Programmierers, sicher zu stellen, dass
 - beim Schreiben in den Stackbereich keine anderen Daten überschrieben werden,
 - der Stackbereich nicht das Ende des internen Datenspeichers überschreitet,
 - nicht mehr aber auch nicht weniger Daten aus dem Stack ausgelesen werden, als vorher hinein geschrieben wurden.
- Verwendung des Stacks
 - Als kurzzeitiger Zwischenspeicher.
 - Zur Sicherung der Rücksprungadresse bei Unterprogrammaufrufen und Interrupts.
 - Zur Sicherung essentieller Daten beim Aufruf von Unterroutinen und bei Interrupts.

Stack-Befehle

- **PUSH *dadr***

1. inkrementiert den StackPointer $SP \leftarrow SP + 1$
2. kopiert das von ***dadr*** adressierte Byte im internen Datenspeicher an die Speicherstelle im Stackbereich, die der StackPointer jetzt adressiert. $(SP) \leftarrow (dadr)$

- **POP *dadr***

1. kopiert das von StackPointer adressierte Byte im Stackbereich an die von ***dadr*** adressierte Speicherstelle im internen Datenspeicher. $(dadr) \leftarrow (SP)$
2. dekrementiert den StackPointer $SP \leftarrow SP - 1$

- Da die Befehle PUSH und POP eine **interne Datenspeicheradresse** als Operand verlangen,
 - kann der Akkumulator nur über seine SFR Adresse 0xE0 bzw. sein SFR Symbol **ACC** adressiert werden
 - also **PUSH ACC**, nicht PUSH A.
 - kann ein Register nur über seine Datenspeicheradresse adressiert werden :
 - also **PUSH AR0** , **PUSH AR7**, nicht PUSH R0, ... PUSH R7.
 - Adressen im PUSH/POP Befehl :
 - 0x00 ... 0x07 für ein Register in Register Bank 0,
 - 0x08 ... 0x0F für ein Register in Register Bank 1, usw.

Zur Erinnerung :

- Die Registerangabe in einem Maschinenbefehl ist Teil des OpCpde-Bytes (Bits 2 ... 0)
z.B. Assemblerbefehl MOV A,R2 → Maschinenbefehl 1110 1010

Stack-Befehle (Forts.)

- In alten Zeiten, als der 8051 nur eine Registerbank hatte, wurden als interne Datenspeicheradressen für die Register R0 bis R7 die Symbole **AR0 ... AR7** (absolute Adresse von Rx) eingeführt.
 - Aus Kompatibilitätsgründen soll das weiterhin gelten.
- Da es jedoch jetzt vier Registerbänke gibt, muss der Assembler wissen, welche Registerbank gerade aktiv ist.

- Die nachfolgenden Befehle aktivieren **während der Laufzeit** die Registerbank 1 und sprechen dann R2 an.

```

MOV   PSW,#0x08      ;aktiviere Register Bank 1      PSW :      0000 1000
MOV   A,R2           ;Maschinenbefehl : 1110 1010    R2  :              010
PUSH  AR2            ;Maschinenbefehl : 0xC00A       Speicheradr.: 0000 1010

```

- Der Assembler muss jedoch **bereits zur Assemblierzeit** wissen, dass Bank 1 aktiv ist, um die Adresse im PUSH Befehl richtig setzen zu können.
- Der Programmierer benutzt den Pseudobefehl USING, um dem Assembler anzugeben, an welchen Stellen im Programm die Benutzung einer bestimmten Registerbank beginnt.

```

USING 1           ;ab hier wird Registerbank 1 verwendet
MOV   PSW,#0x08      ;aktiviere Register Bank 1
MOV   A,R2           ;Maschinenbefehl : 1110 1010
PUSH  AR2            ;Maschinenbefehl : 0xC00A
.....
.....
USING 2           ;ab hier wird Registerbank 2 verwendet

```

- Der Pseudobefehl **USING** (USING 0, USING 1, USING 2, oder USING 3) spezifiziert die Registerbank, die für die nachfolgenden Befehle gültig ist.
 - Diese Spezifikation dient nur dazu, dass der Assembler beim Auftreten eines Symbols AR0 ... AR7 die interne Datenspeicheradresse für das Register der spezifizierten Registerbank einsetzt.
 - Ein USING Pseudobefehl ist für alle Befehle bis zum nächsten USING Pseudobefehl bzw. bis zum Ende des Programms gültig.
- Gibt es keine USING Spezifikation in einem Programm, verwendet der Assembler ausschließlich die Registerbank 0.

Übungsaufgabe AR3 :

Addieren Sie die beiden 4 Byte langen Binärzahlen bei Op1 und Op2 und legen Sie das Ergebnis bei Erg ab !

Hinweis : Speichern Sie den Wert von Op1 zuerst in den Stack und benutzen Sie R0 und R1 zur indirekten Adressierung von Op2 und Erg während der byteweisen Addition !

Unterprogramm-Befehle

● LCALL adr16

1. Rettet die Adresse des auf den LCALL folgenden Befehls (2 Bytes) auf den Stack.
2. Führt einen Sprung zur absoluten Programmspeicheradresse adr16, der Anfangsadresse des Unterprogramms, aus.

● ACALL adr11

- Wird äußerst selten benutzt. Analog zum AJMP Befehl.

● RET

- Holt zwei Bytes vom Stack und lädt sie in den Befehlszähler (PC).
 - Dies entspricht einem Sprung zu der Adresse, die bei der Ausführung des Call Befehls auf den Stack gerettet wurde.
 - Das Programm wird also nach Ausführung des Unterprogramms mit dem auf den Call Befehl folgenden Befehl fortgesetzt.

LCALL adr16

$$\begin{aligned} (PC) &\leftarrow (PC) + 3 \\ (SP) &\leftarrow (SP) + 1 \\ ((SP)) &\leftarrow (PC \text{ LOW}) \\ (SP) &\leftarrow (SP) + 1 \\ ((SP)) &\leftarrow (PC \text{ HIGH}) \\ (PC) &\leftarrow \text{adr16} \end{aligned}$$

RET

$$\begin{aligned} (PC) &\leftarrow (PC) + 1 \\ (PC \text{ HIGH}) &\leftarrow ((SP)) \\ (SP) &\leftarrow (SP) - 1 \\ (PC \text{ LOW}) &\leftarrow ((SP)) \\ (SP) &\leftarrow (SP) - 1 \end{aligned}$$

- Das Unterprogramm schiebt sich sozusagen an der Stelle des Call Befehls in das Hauptprogramm.
- Die Speicherung der Returnadresse im Stack, erlaubt den **rekursiven Aufruf** von Unterprogrammen.

- Das Unterprogramm muss eventuell Speicherelemente verwenden, in denen für das Hauptprogramm relevante Informationen stehen.
 - Insbesondere Akkumulator, PSW und Register.
 - Damit diese Informationen nicht verloren gehen, sind im Unterprogramm zusätzliche Befehle einzufügen, die diese Informationen
 - direkt nach dem Aufruf des Unterprogramms auf den Stack retten,
 - direkt vor dem Ende des Unterprogramms wieder zurück stellen.
- Um Fehler möglichst zu vermeiden, wird dringend empfohlen, **die Schnittstelle zu Unterprogrammen akribisch zu dokumentieren**.
Anzugeben sind :
 - Symbol zum Aufruf des Unterprogramms (Name des Unterprogramms)
 - Funktionsbeschreibung
 - Input-Parameter (Typ und ggf. Adresse)
 - Output-Parameter (Typ und ggf. Adresse)
 - Die vom Unterprogramm zerstörten Elemente

Siehe auch das folgende Beispiel

Beispiel 1 : Addition von zwei 4 Byte Werten als Unterprogramm

Hauptprogramm

```

:      :
:      :
MOV    Op1,....    ;Lade Werte in Op1 ....
:      :
:      :
LCALL  Add4By    ;Rufe Unterprogramm
:      :
:      :
MOV    ...,Erg     ;Verwerte das Ergebnis
:      :
:      :

```

Unterprogramm Add4By

Funktion : **Addiert zwei 4 Byte lange Binärzahlen**

Input : 4 Bytes an der Adresse Op1,
4 Bytes an der Adresse Op2

Output : 4 Bytes an der Adresse Erg

Zerstört : Akkumulator

Add4By:

```

PUSH  PSW
PUSH  AR0           ;Rette zu erhaltende Werte
PUSH  AR1
PUSH  Op1
PUSH  Op1+1         ;Op1 auf den Stack
PUSH  Op1+2         ;MSB zuerst, LSB zuletzt
PUSH  Op1+3

MOV   R0,#Op2+3    ;R0 auf das LSB von Op2
MOV   R1,#Erg+3    ;R1 auf das LSB von Erg
CLR   CY
Loop: POP   ACC           ;Hole Op1 Byte,
      ADDC  A,@R0
      MOV   @R1,A
      DEC  R0
      DEC  R1
      CJNE R1,#Erg-1,Loop ;Durchlaufe Schleife 4 mal

POP   AR1
POP   AR0           ;Stelle zu erhaltende Werte zurück
POP   PSW
RET                ;Kehre zum Hauptprog. zurück

```

Unterprogramm-Parameter

- Beim Aufruf eines Unterprogramms kann das Hauptprogramm Werte zur Verfügung stellen, die das Unterprogramm benutzt, und das Unterprogramm kann Ergebniswerte für das Hauptprogramm liefern.
 - Vergleiche Argumentübergabe bei C-Funktionen.
 - Diese Werte können von Aufruf zu Aufruf unterschiedlich sein.
- Argumentübergabe in 8051 Programmen kann erfolgen
 - in gemeinsam bekannten Speicherstellen (siehe Beispiel oben)
 - in Registern (siehe folgendes Beispiel)
 - auch gemischte Argumentübergabe möglich

Beispiel 2 : Addition von zwei 4 Byte Werten als Unterprogramm

Hauptprogramm

```

:
:
MOV    R0,#Operand1 ;Lade Adresse von Op1
MOV    R1,#Operand2 ;Lade Adresse von Op2
MOV    R2,#Ergebnis ;Lade Adresse von Erg
LCALL  Add4By       ;Rufe Unterprogramm auf
MOV    ...,Erg      ;Verwerte das Ergebnis
:
:
```

Unterprogramm Add4By

Funktion : Addiert zwei 4 Byte lange Binärzahlen

Input : R0 = Adresse des ersten Operanden, Wert des Operanden
R1 = Adresse des zweiten Operanden, Wert des Operanden
R2 = Adresse des Ergebnisses

Output : An der Adresse in R2 = Ergebnis

Zerstört : Akkumulator, PSW

```

Add4By:  PUSH   AR0           ;Rette zu erhaltende Werte
            PUSH   AR1
            PUSH   AR3

            MOV    R3,#4
PushOp1:   MOV    A,@R0
            PUSH   ACC           ;Op1 auf den Stack
            INC    R0
            DJNZ   R3,PushOp1

            MOV    A,R1
            ADD    A,#4           ;+4, weil DEC vor ADD
            MOV    R1,A           ;R1 zeigt hinter Operand 2
            MOV    A,R2
            ADD    A,#4
            MOV    R0,A           ;R0 zeigt hinter Ergebnis

Loop:      DEC    R0
            DEC    R1
            POP    ACC           ;Hole Op1 Byte, (LSB zuerst, MSB zuletzt)
            ADDC   A,@R1
            MOV    @R0,A
            CJNE   R0,2,Loop     ;Durchlaufe die Schleife 4 mal
                                           ;was bedeutet die '2' in diesem Befehl ?

            POP    AR3           ;Stelle zu erhaltende Werte zurück
            POP    AR1
            POP    AR0

            RET                   ;Kehre zum Hauptprogramm zurück

```

Interrupt Routinen

Ein Interrupt ist ein von Hardware initiiertes Unterprogrammaufruf.

(An dieser Stelle wird nur der Mechanismus beschrieben, wie eine Routine ausgeführt wird, die auf einen Interrupt reagiert. Die eigentliche Besprechung von Interrupts erfolgt später in diesem Dokument.)

- In einem Computersystem gibt es zuweilen außergewöhnliche Situationen, in denen bestimmte Routinen ausgeführt werden müssen. Diese Routinen kümmern sich dann um diese speziellen Situationen.

Beispiele :

- Eine E/A-Operation in einer peripheren Einheit wird beendet. Eine Routine sollte das Programm, das die Operation ursprünglich angestoßen hat, darüber informieren.
- Es passiert ein Speicherzugriff zu einer ungültigen Adresse. Eine Fehlerbehandlungsroutine sollte sich darum kümmern.
- All diese außergewöhnlichen Situationen im laufenden Programm permanent abzufragen, würde eine Menge Performance kosten.
 - (Die Methode, in einer Programmschleife das Auftreten eines Ereignisses abzuwarten, nennt man **Polling**. Polling ist z. B. bei Testprogrammen für E/A-Geräte eine häufig verwendete Methode).
- Die Lösung für diese Problematik ist der **Interrupt-Mechanismus**.
 - Das Auftreten des außergewöhnlichen Ereignisses aktiviert eine Interrupt-Anforderung (**Interrupt Request**).
 - Wenn die Interrupt-Logik im Prozessor entsprechend konditioniert ist (wenn Interrupts **enabled** sind), wird die eigentliche Unterbrechung (**der Interrupt**) durchgeführt.
 - Zu diesem Zweck rettet das Steuerwerk am Ende des gerade ausgeführten Befehls den Befehlszähler auf den Stack und verzweigt zur **Interrupt Service Routine (ISR)**.
 - Dieser Vorgang ist analog zum Aufruf eines Unterprogramms.

- Interrupt Service Routine

- Im Fall eines **Unterprogrammaufrufs** ist die Anfangsadresse der aufgerufenen Routine der Operand im **CALL** Befehl.
- Im Falle eines **Interrupts** ist die Anfangsadresse der aufgerufenen Routine eine **fest verdrahtete absolute** Programmspeicheradresse.
- Am Ende der Interrupt Service Routine erfolgt die Rückkehr ins unterbrochene Programm mit dem Befehl **RETI**.
 - Analog zum RET Befehl bei einem Unterprogrammaufruf.
 - Zusätzlich zum Laden der Returnadresse in den PC setzt der RETI Befehl die Indikatoren im Steuerwerk zurück, die angezeigt haben, dass die gerade ausgeführten Befehle Teil einer ISR sind.
 - Dadurch kann ein neuer (während der Ausführung der ISR blockierter) Interrupt prozessiert werden
- Da nicht voraussehbar ist, an welcher Stelle das Programm unterbrochen wird, muss die Interrupt Service Routine die Inhalte aller Ressourcen, die sie selbst verwendet, zu Beginn auf den Stack retten und am Ende wieder zurückstellen.

Makros

- Ein Makro dient dazu, häufiger auftretende Befehlssequenzen an allen notwendigen Stellen ins Programm einzufügen.
 - Die Befehlssequenz in der **Makrodefinition**, wird nur einmal (wie ein Unterprogramm) im Programmcode aufgeführt.
 - An jeder Stelle im Programm, an der diese Befehlssequenz gebraucht wird, erfolgt ein **Makroaufruf**.
 - D.h. der Assembler fügt die Befehlssequenz an diesen Stellen ins Programm ein.
 - Im Unterschied zum Unterprogramm sind die Befehle der Befehlssequenz dann mehrfach vorhanden.
- Im Prinzip erspart ein Makro lediglich Schreibarbeit.
- Die Makrobefehle MACRO und ENDM identifizieren den Beginn und das Ende einer Makrodefinition.
- Das Symbol vor dem MACRO Pseudobefehl ist der Name, mit dem das Makro aufgerufen wird.

- **Beispiel** : Löschen eines 10 Byte langen Anzeigefeldes

- Makrodefinition :

```

LoeFeld: MACRO
LOCAL Loop
  MOV R0,#Feld
Loop:  MOV @R0,#0x20      ;Leerzeichen einsetzen
      INC R0
      CJNE R0,#Feld+10,Loop
ENDM

```

- Makroaufruf :

```

:
LoeFeld
:

```

- Makroexpansion :

```

:
MOV R0,#Feld
??0000: MOV @R0,#20H      ;Leerzeichen einsetzen
      INC R0
      CJNE R0,#Feld+10,??0000
:

```

- Wenn der Assembler im obigen Beispiel den Makrocode unverändert expandieren würde, wäre bei mehrfachem Aufruf das Symbol Loop mehrfach vorhanden, was einen Fehler zur Folge hätte.
 - Der Macrobefehl LOCAL identifiziert das darauf folgende Symbol als lokales Symbol.
 - Der Makroassembler ersetzt dieses Symbol bei jedem Makroaufruf durch ein anderes selbst erfundenes Symbol (??0000, ??0001,).

Parameter

- In einem Makro können auch Parameter verwendet werden.
 - Die Parameter in der Makrodefinition werden bei der Expansion durch die Argumente des Makroaufrufs ersetzt.
 - Positionelle Ersetzung : Parameter 1 wird durch Argument 1 ersetzt, Parameter 2 durch Argument 2, usw.
- **Beispiel** : Kopieren von Zeichenketten mit variablen Längen

- Makrodefinition :

CopyStr:

```

MACRO   Src,Dst,Cnt
LOCAL   Label
MOV      R2,#Cnt   ;Länge
MOV      R1,#Src   ;Quelle
MOV      R0,#Dst   ;Ziel
Label:
MOV      A,@R1
MOV      @R0,A     ;kopiere Zeichen
INC      R0
INC      R1
DJNZ    R2,Label
ENDM

```

- Makroexpansion :

```

:
:
MOV      R2,#8     ;Länge
MOV      R1,#Orig  ;Quelle
MOV      R0,#Ziel  ;Ziel
??0000:
MOV      A,@R1
MOV      @R0,A     ;kopiere Zeichen
INC      R0
INC      R1
DJNZ    R2,??0000
:

```

- Makroaufruf :

```

:
CopyStr  Orig, Ziel, 8
:

```

Gemeinsame Benutzung von Makro und Unterprogramm

- **Ausgangssituation** : An vielen Stellen im Hauptprogramm soll eine Gleitkomma-Addition durchgeführt werden.

Hauptprogramm

```
.....
AddGKZ  G1, G2,Erg1      ;Makroaufruf
```

```
.....
AddGKZ  G3, G4,Erg3      ;Makroaufruf
```

```
.....
```

Makrodefinition

AddGKZ:

```
MACRO  Src1,Src2,Dest
PUSH  AR0
PUSH  AR1
PUSH  AR2

MOV   R0,#Src1
MOV   R1,#Src2
MOV   R2,#Dest
LCALL UPAddGKZ

POP   AR2
POP   AR1
POP   AR0
ENDM
```

Unterprogramm *UPAddGKZ*

Funktion : **Addiert zwei Gleitkommazahlen**

Input : R0 = Adresse des ersten Operanden,
R1 = Adresse des zweiten Operanden,
R2 = Adresse des Ergebnisses

Output : An der Adresse in R2 = Ergebnis

Zerstört : R0, R1, R2

UPAddGKZ:

Übungsaufgabe UP1:

Schreiben Sie ein Makro + Unterprogramm, das zwei 16 Bit Binärzahlen addiert (AddInt ErgOp1,Op2)

Lösung zu Übungsaufgabe

Übungsaufgabe UP1 :

;Hauptprogramm

```
.....  
AddInt   Zähler1,Zähler2   ;Makroaufruf  
JC       Fehler  
.....
```

;Makrodefinition

```
AddInt:  MACRO  ErgOp1,Op2  
    PUSH   AR0  
    PUSH   AR1  
    MOV    R0,#ErgOp1  
    MOV    R1,#Op2  
    LCALL  UPAddInt  
    POP    AR1  
    POP    AR0  
ENDM
```

;Unterprogramm *UPAddInt***;Funktion : Addiert zwei Integerzahlen**

;Input : R0 = Adresse des ersten Operanden,
; R1 = Adresse des zweiten Operanden,
;Output : Ergebnis der Addition an der Adresse in R0
; Carry Flag = Carry für Integer-Addition
;Zerstört : R0, R1, PSW

UPAddInt:

```
PUSH    ACC
INC     R0      ;Zeige auf LSB
INC     R1
MOV     A,@R0
ADD     A,@R1   Addiere LSB Bytes
MOV     @R0,A
DEC     R0      ;Zeige auf MSB
DEC     R1
MOV     A,@R0
ADDC    A,@R1   Addiere MSB Bytes
MOV     @R0,A
POP     ACC
RET
```