FACHHOCHSCHULE DES LANDES RHEINLAND-PFALZ
ABTEILUNG KAISERSLAUTERN FB ELEKTROTECHNIK

Referat aus der Vorlesung *Prozessrechner*

über das Thema

REENTRANTE PROGRAMME

Bearbeiter: Thomas Höh Matr.Nr 2925

Betreuer: Prof. Schmiedel

WS 81/82

Inhaltsverzeichnis

0. Vorbemerkung

Im ersten Abschnitt wird versucht, den Begriff der reentranten Schreibweise von Programmen im Hinblick auf Compilerbau bzw. Betriebssystem-Programmen zu erläutern. Im zweiten Abschnitt wird speziell auf den Mikroprozessor 8080/8085 eingegangen. Der dritte Abschnitt ergänzt das vorher Beschriebene anhand von Beispielen.

Die in eckigen Klammern [ ] stehenden Zahlen verweisen auf Angaben im Literaturverzeichnis auf den letzten Seiten des Referates.

Dieses Dokument ist eine Abschrift des Originals aus dem Jahre 1981 und wurde von Verfasser selbst im Jahr 2009 erstellt.

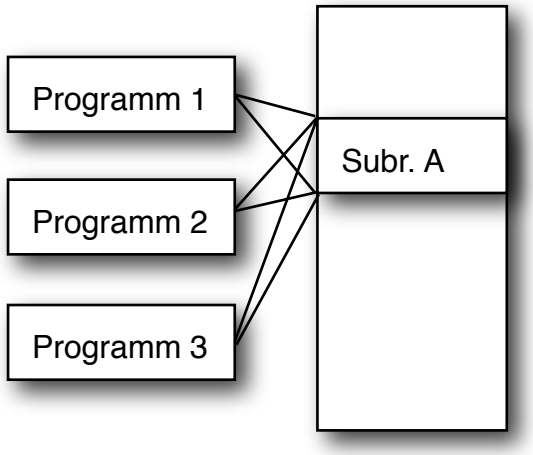
1. Reentrante Programme bei großen Computersystemen

1.1 Programme die sich selbst verändern

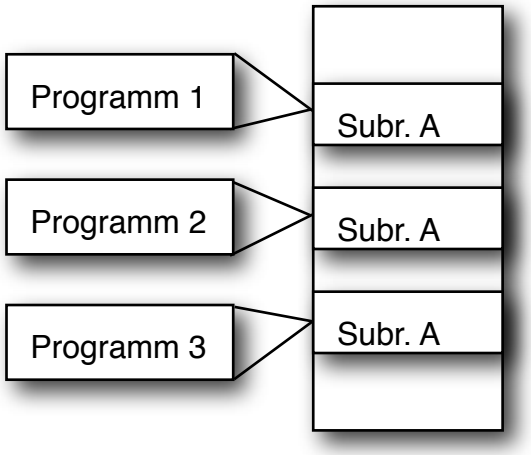
Bei Computersystemen besteht oft das Problem, daß ein Anwendungsprogramm von mehreren verschiedenen Prozessen (Hauptprogrammen) verwendet werden soll. Um Speicherplatz zu sparen, wird eine einzige Kopie des Programmes in den Arbeitsspeicher gebracht und von allen Prozessen benutzt (siehe Bild). [9], [15]

Hierfür ist es erforderlich, daß das Anwendungsprogramm während der Prozeßausführung nicht geändert wird, bzw. sich nicht selbst verändert, d.h. die Instruktionen des Programmes müssen zu jeder Zeit dieselben sein. [5], [10]

Dies ist eine Bedingung, die ein Programm erfüllen muß, um reentrant zu sein.



reentranter Zugriff



herkömmlicher Zugriff

1.2 Trennung zwischen Programm und Daten

Eine weitere Forderung an ein reentrantes Programm ist eine strikte Trennung zwischen Daten (temporärer Teil), die ein Programm benutzt, und dem Programm selbst (permanenter Teil).
Diese Trennung ermöglicht es, das Programm (procedure) mehr als einmal (gleichzeitig) auszuführen. [1], [11]
Dies ist gleichbedeutend, wie wenn zwei Köche gleichzeitig dasselbe Kochbuch (Programm) benutzen, aber unterschiedliche Töpfe und Zutaten (Daten). [11]

1.3 Definition

Ein Programm, das sich nicht selbst verändern kann, bzw. das für jeden Prozeß (Ablauf) ein separates Datenfeld benutzt, nennt man reentrant.

Im Englischen ist auch die Bezeichnung pure (rein) procedure üblich. [10], [11], [3]
Andere Bezeichnungen für reentrant sind auch wiedereintrittsfähig und ablaufinvariant. [13], [21], [3]

1.4. Anwendung

Anwendung finden reentrante Programme in größeren Computersystemen z.B. bei „time sharing“ - Betrieb, in Betriebssystem-Programmen, Compiler und Bibliotheksprogrammen. [12]

1.5. Vergleich reentrant - seriell wiederverwendbar

Manche Prozeduren sind seriell wiederverwendbar (reusable), aber nicht reentrant. Das bedeutet, daß die Prozedur nicht von zwei Prozessen gleichzeitig benutzt werden kann, aber nacheinander (zweiter Prozeß nach der vollständigen Ausführung des ersten Prozesses). [9], [11]
Seriell wiederverwendbare Programme sind also nicht bei Interupt- bzw. time sharing - Systemen zu verwenden.

1.6 Vergleich reentrant - rekursiv

Reentrant organisierte Programmteile, die sich während der Ausführung selbst aufrufen können, werden auch rekursiv oder selbst-reentrant genannt (self-reentrancy).

Die Ausführung von rekursiven Programmteilen bedingt immer eine reentrante Organisation derselben. Jedoch reentrant organisierte Programme müssen nicht unbedingt auch rekursiv sein. [1], [5], [6], [15]

Beispiel eines rekursiven Programmes (Prozedur) zur Berechnung der Fakultät einer Zahl geschrieben in der höheren Programmiersprache Algol: [14], [16]

```
INTEGER PROCEDURE FAK(N); INTEGER N;
FAK:=IF N=0 THEN 1 ELSE N*FAK(N-1);
```

Fortran-Unterprogramme können im allgemeinen nicht rekursiv aufgerufen werden. [6]

2. Reentrante Schreibweise von Programmen in Assembler 8080/8085

2.1 Definition

> Ein Unterprogramm ist reentrant, wenn es durch ein Unterbrechungsprogramm (ISR) unterbrochen werden kann und erneut in des Unterprogramm eingetreten werden kann. [13], [19], [6]

Man kann also sagen, ein reentrantes Unterprogramm ist ein Programm, das sowohl von einem Hauptprogramm, als such von einer Interrupt-Service-Routine (ISR) benutzt werden kann.

Ist das Unterprogrmm gerade von einem Hauptprogramm aufgerufen und es tritt ein Interupt auf, dann muß ausgeschlossen werden, daß Daten, die im Unterprogramm benötigt werden, zerstört werden, wenn die ISR dasselbe Unterprogramm mit anderen Parametern wieder aufruft (siehe Bsp. 3.1 u. 3.2). [3], [6], [16]

Wenn ein Unterprogramm nicht reentrant geschrieben werden kann, müssen für die verschiedenen aufrufenden Programme auch verschiedene Versionen des Unterprogrammes zur Verfügung stehen. Wenn dies nicht berücksichtigt wird, können Fehler auftreten (siehe Beispiele).

2.2 Anwendung

Anwendung finden reentrant geschriebene Programme vor allem als Standardprogramme in Mikroprozessorsystemen, die auf Unterbrechungen basieren. [13]

Beispiele hierfür sind:
Codeumwandlung, Zeichenmanipulation, Fehler-, Prüf-Programme, I/O-Routinen. [19]

2.3 Methoden, um reentrante Programme zu erhalten

Mikroprozessor-Unterprogramme sind relativ leicht reentrant zu schreiben, da der Aufruf-Befehl den Stack verwendet, was automatisch der Wiedereintritt ergibt. [13]

Allerdings sind folgende Anwendungsfälle zu beachten, da bei ihnen bestimmte Regeln anzuwenden sind:

- Parameterübergabe in ein Unterprogramm
- Abspeichern von Zwischenergebnissen in einem Unterprogramm
- Dynamische Programmierung

2.3.1 Regeln

Ein reentrantes Programm darf nur auf Daten zugreifen, die

a) in Zwischenregistern der CPU stehen,
b) indirekt adressiert sind über einen Zeiger, dessen absoluter Wert erzeugt wurde, bevor das Unterprogramm aufgerufen wurde,
c) zwar direkt adressiert in ROM oder RAM stehen, aber nur gelesen und zu keiner Zeit geändert werden (Konstante).
d) Ein Programm darf sich nicht selbst verändern, indem es selbst den Code für Instruktionen in einen RAM-Bereich schreibt und diese dann ausführt.

Mit anderen Worten:

> Ein Unterprogramm ist nicht reentrant, wenn es direkt Zugriff zu einem RAM-Speicher hat, oder wenn es einen RAM-Bereich mittels eines Zeigers absolut adressiert. [3]

2.3.2 Methoden

a) Parameter bzw. Zwischenergebnisse sollten im allgemeinen in CPU-Registern abgelegt werden.
b) Komplexe Unterprogramme erfordern u.U. zu viele Parameter (bzw. Zwischenergebnisse), mehr als CPU-Register vorhanden sind. Solche Programme können einen zugewiesenen Speicherbereich benutzen, wobei jedes aufrufende Programm eine unterschiedlichen Bereich haben muß. Die Anfangsadresse des jeweiligen Variablenbereiches (Block) wird in einem CPU-Register dem Unterprogramm übergeben (indexindirekte Adressierung). Die Anzahl der Blöcke bestimmt die Schachtelungstiefe.
c) Die eleganteste Methode ist die Benutzung des Stacks zum Ablegen von Zwischenergebnissen. Diese Methode ist u.U. mühsam und umständlich bei der Programmierung; der Gebrauch des Stacks ist allerdings einfacher, wenn es sich um ein System mit verschiedenen Ebenen von Unterprogrammen und Interupts handelt.

Bei älteren Mikroprozessortypen (z.B. PDP 8) war eine reentrante Schreibweise nicht möglich, da hier die erste Zeile eines Unterprogrammes für die Rücksprungadresse vorgesehen war. Bei Unterbrechung und erneutem Aufruf würde hier die Rücksprungadresse überschrieben. Das Unterprogramm würde also nicht mehr zum Hauptprogramm zurückfinden.

2.4 Rekursive Programme [5], [6], [13]

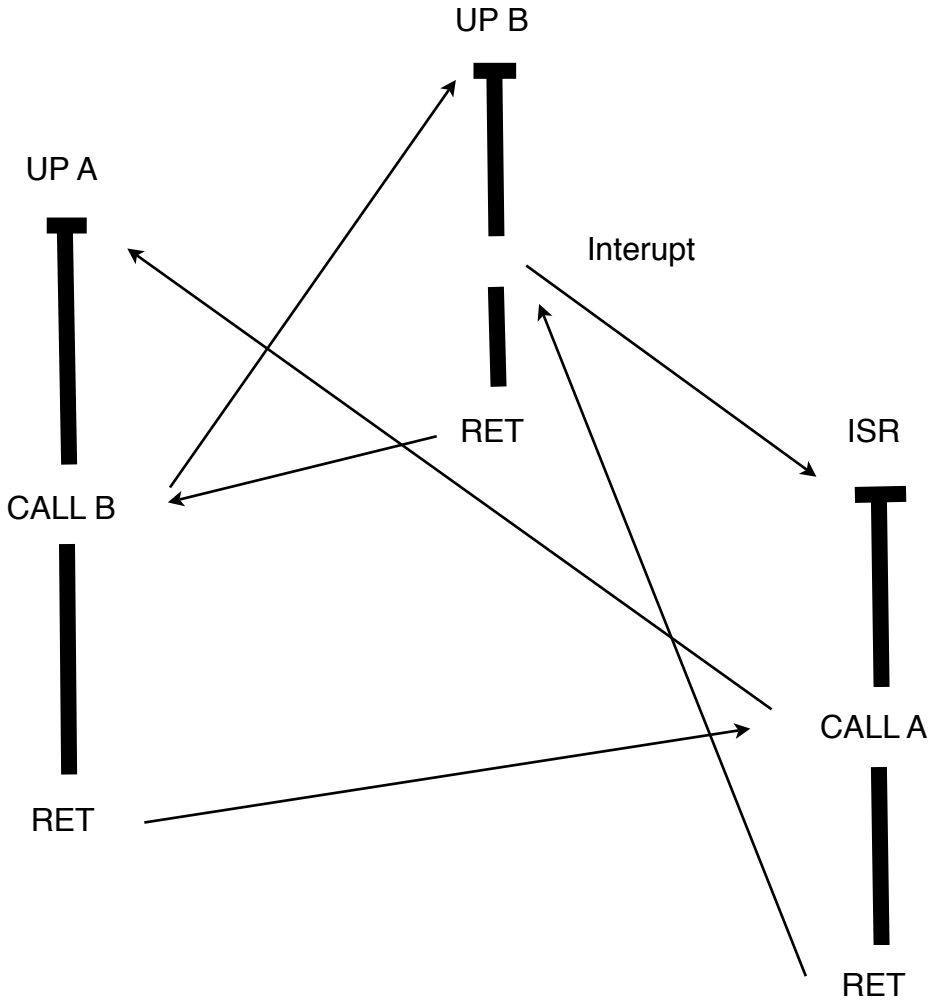
In rekursive Programme, d.h. Programme, die sich selbst aufrufen, muß natürlich auch wieder eingetreten werden können.

Bei Mikroprozessoranwendungen sind rekursive Programme allerdings nicht sehr gebräuchlich.

2.5 Grenzen der reentranten Schreibweise

Es können Fälle auftreten, wo selbst eine reentrante schreibweise von Unterprogrammen zu Fehlern führen kann. Dies soll anhand des folgenden Beispieles erörtert werden:

UP A ruft UP B auf, dieses wird unterbrochen und die ISR ruft UP A auf.



Wenn UP A so geschrieben ist, daß es dem reentrant geschriebenen UP B einen Zeiger auf einen Datenbereich übergibt, werden beim zweiten Aufruf die früheren Daten überschrieben.

Eine solche verschachtelte Anordnung führt dann zu einem fehlerfreien Ablauf, wenn ein übergeordnetes Programm (Betriebssystem) es ermöglicht, daß z.B. durch einen Zeiger angezeigt wird, zum wievielten Mal das UP A aufgerufen ist, damit für jeden Aufruf von UP B ein anderes Datenfeld bereitgestellt werden kann.

Solche Betriebssysteme bedingen zwar längere Programmlaufzeiten, ermöglichen aber, daß solche Anordnungen wie oben beschrieben, auch mit noch größerer Verschachtelungstiefe erlaubt sind.

Wenn allerdings die Programmteile so geschrieben werden, daß sie den Stack zur Datenspeicherung (PUSH) gebrauchen (3.1.3 und 3.3.c), würde die gezeigte Anordnung funktionieren. Hierbei müßte allerdings auf einen ausreichend großen Stack-Bereich geachtet werden, da der Stack-Speicherbedarf mit zunehmender Schachtelungstiefe ansteigt.

3. Beispiele

3.1 Addition von 8-bit-Werten

Unterprogramm zur Addition von zehn 8-bit-Werten.
Ergebnis im Akkumulator.
Überlauf wird nicht berücksichtigt.
Verwendete Register: A, B, H, L

3.1.1 Datenfeld mit absoluten Adressen

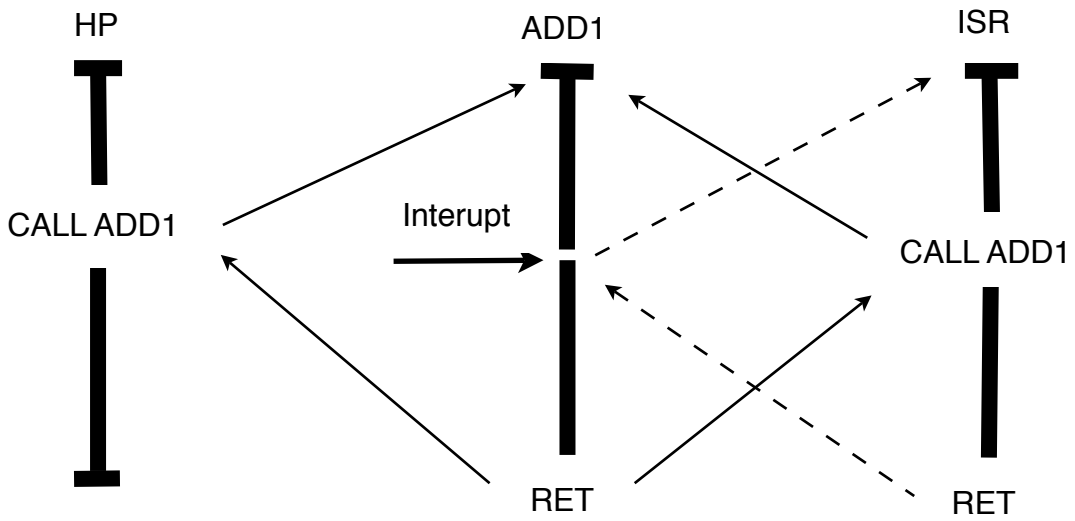
Die Werte stehen in externen Speicherplätzen (RAM) unter folgenden Adressen:

```
          ZAHL 0    in   0F000H
             ...
          ZAHL 9    in   0F009H


ADD1:     MVI  B,9        ; SCHLEIFENZÄHLER
          LXI  H,0F000H   ; ANFANGSADRESSE DES ZAHLENFELDES
          MOV  A,M        ; ERSTER WERT
LOOP:     INX  H          ; SPEICHERADRESSE ERHÖHEN
          ADD  M          ; ADDITION
          DCR  B          ; SCHLEIFENZÄHLER ERNIEDRIGEN
          JNZ  LOOP
          RET
```

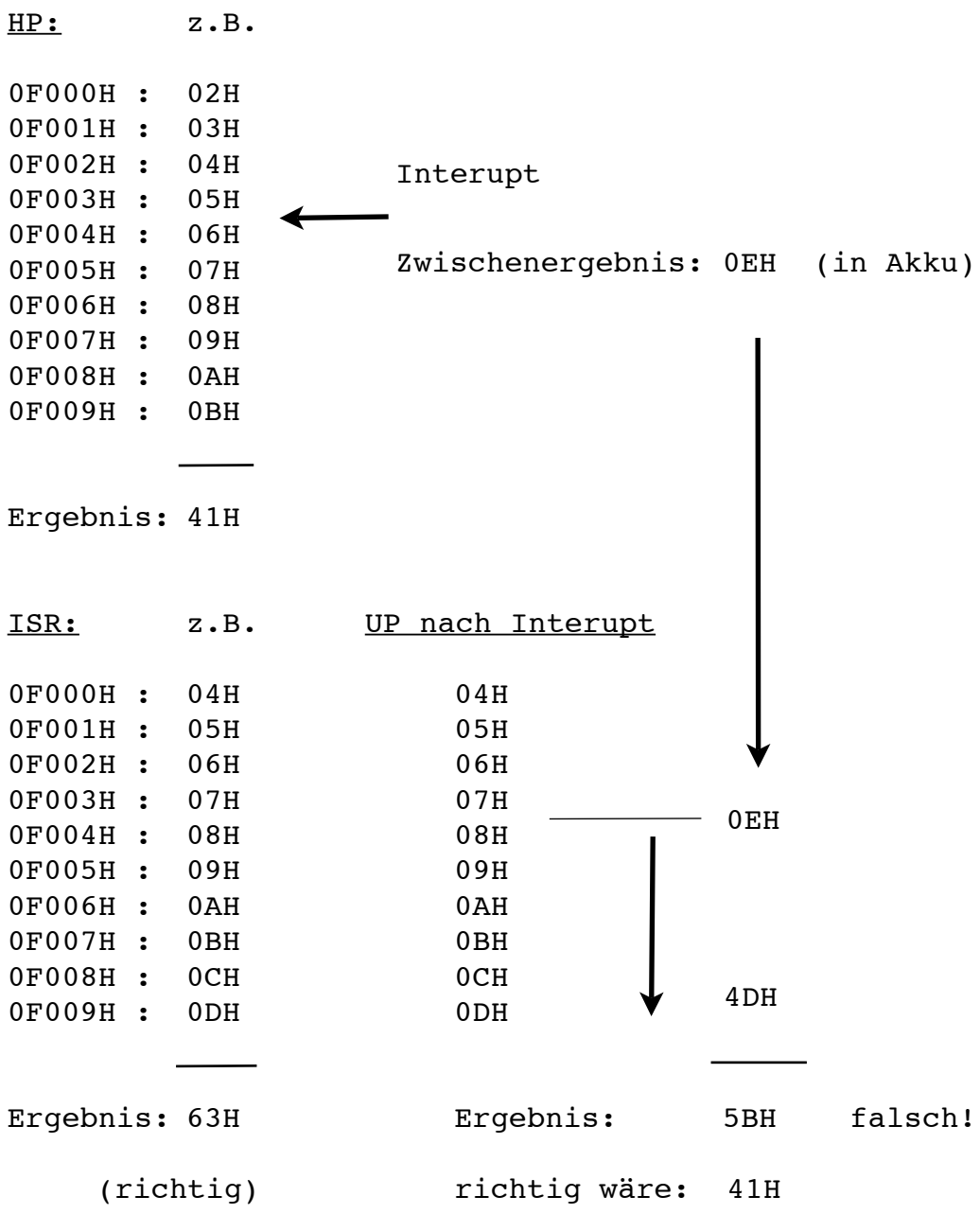
nicht reentrant

Angenommen, während der Abarbeitung des Unterprogrammes ADD1 würde ein Interupt auftreten. Der Programmablauf würde mit der entsprechenden Interupt-Service-Routine (ISR) fortgeführt werden und diese würde nun ihrerseits dieses Unterprogramm ADD1 benötigen.



Es werden (von der ISR) neue zu addierende Werte in den Speicherbereich geschrieben, das Unterprogramm ADD1 aufgerufen und, angenommen es erfolgten keine weiteren Unterbrechungen mehr, die Additionen durchgeführt und zur ISR zurückgesprungen. Nach Beendigung der ISR wird nun das Unterprogramm ADD1 seinen ersten Durchlauf beenden wollen. Es stehen zwar die alten (richtigen) Registerinhalte wieder zur Verfügung (angenommen, die ISR hatte sie gerettet), die Zahlen im externen Speicher sind aber nicht mehr die richtigen, da diese beim vorherigen Durchlauf überschrieben wurden. Somit liefert das Unterprogramm zwangsweise ein falsches Ergebnis an das Hauptprogramm. [6]

Speicherinhalte während der Bearbeitung:



Um solche Fehler zu vermeiden gibt es zwei Möglichkeiten. Erstens könnte man während des Unterprogrammlaufes alle Interupts sperren (Disable Interupt), was aber in den meisten Fällen nicht möglich sein wird (Echtzeitbetrieb).

Die zweite Möglichkeit besteht darin, jedem aufrufenden Programm seinen eigenen „privaten“ Datenbereich zu schaffen, den jeweils auch das Unterprogramm benutzt. Anfangsadresse (u.U. auch Größe) dieses Variablenblockes müssen dem Unterprogramm in einem Register übergeben werden.

3.1.2 Datenfeld mit relativen Adressen

Das nach obigen Gesichtspunkten gestaltete Programm könnte folgendermaßen aussehen, wobei die Anfangsadresse des Datenblocks im Registerpaar H&L stehen muß:

```
ADD2:     MVI  B,9        ; SCHLEIFENZÄHLER
          MOV  A,M        ; ERSTER WERT IN AKKU
LOOP:     INX  H          ; SPEICHERADRESSE ERHÖHEN
          ADD  M          ; ADDITION
          DCR  B          ; SCHLEIFENZÄHLER ERNIEDRIGEN
          JNZ  LOOP
          RET
```

reentrant

Der Datenbereich des Hauptprogrammes wird durch die Interupt-Service-Routine nicht verändert. Eine Erweiterung der Möglichkeiten des Unterprogrammes wäre noch gewesen, die Größe des Datenblockes mit in das Unterprogramm zu übergeben und damit die Anzahl der zu addierenden Werte variabel zu gestalten.

möglicher Aufruf:

HP:

```
.
.
LXI  H,0F000H
CALL ADD2
.
.
```

ISR:

```
.
.
LXI  H,0F0F0H
CALL ADD2
.
.
RET
```

3.1.3 Übergabe der Daten im Stack

1. Die Daten müssen vor Aufruf des Unterprogrammes ADD3 mittels der PUSH-Anweisung in den Stack geschrieben werden.
2. Es ist aber auch möglich, den Stackpointer mit der Anfangsadresse eines schon vorhandenen Datenbereiches zu besetzen bevor das Unterprogramm aufgerufen wird. Hierbei ist allerdings zu beachten, daß schon im Stack befindliche Daten verlorengehen (z.B. Rücksprungadresse). Diese Methode ist also für ISR nicht zu empfehlen. Es wäre zwar möglich, den alten Stackpointer-Inhalt zwischenzuspeichern und nach dem Aufruf von Unterprogramm ADD3 wieder einzusetzen, was allerdings negative Folgen in Bezug auf Rechenzeit und Speicherplatzbedarf hat.

Verwendete Register: A, B, D, E, H, L

reentrant

```
ADD3:   POP  H          ; RÜCKSPRUNGADRESSE
        XCHG            ; IN D&E BRINGEN
        MVI  B,5        ; SCHLEIFENZÄHLER
        XRA  A          ; AKKU LÖSCHEN
LOOP:   POP  H          ; ZWEI WEERTE AUS STACK IN H&L
        ADD  L          ; ADDITION REG L + AKKU
        ADD  H          ; ADDITION REG H + AKKU
        DCR  B          ; SCHLEIFENZÄHLER ERNIEDRIGEN
        JNZ  LOOP
        XCHG            ; ALTER SP-INHALT IN H&L
        PCHL            ; ALTER SP-INHALT IN PC = RET
```

möglicher Aufruf:

zu 1.)

```
.
.
PUSH ZAHL0
...
PUSH ZAHL9
CALL ADD3
.
.
```

(HP bzw. ISR)

zu 2.)

```
Datenblock:
  ZAHL9:  9000H
  ...
  ZAHL0:  9009H
```

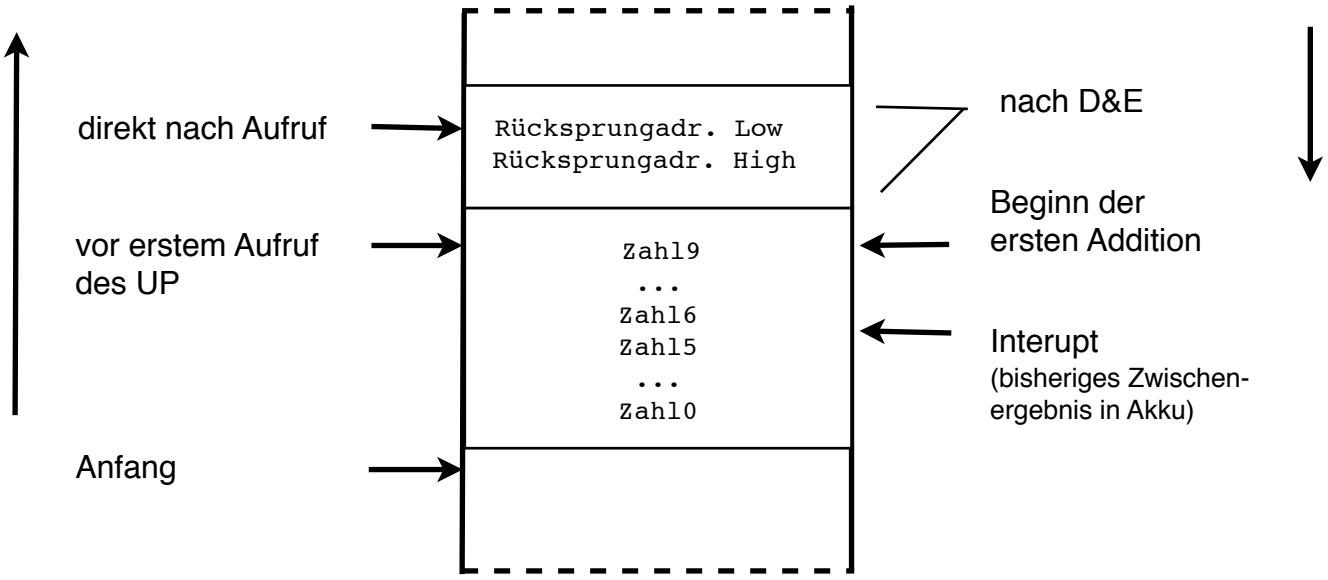
```
.
.
LXI  SP,9000H
CALL ADD3
.
.
```

(NUR HP)

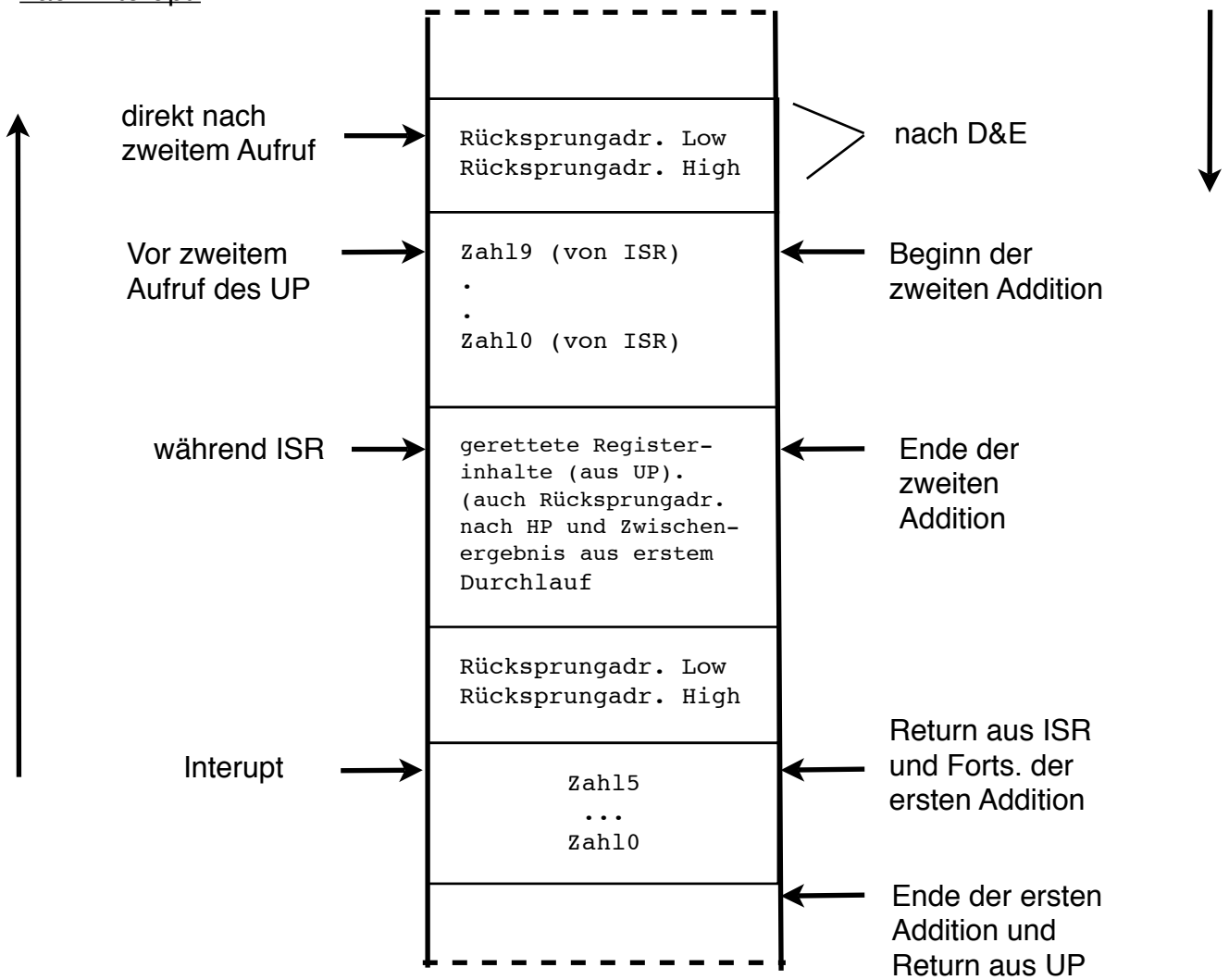
Bei der Übergabe der Daten mittels Stack muß auf eine ausreichende Reservierung für den Stackbereich im RAM geachtet werden.

Stackinhalte während der Bearbeitung (bei Methode 1):

vor Interupt:



nach Interupt:



3.1.4 Vergleich in Bezug auf Rechenzeit und Speicherbedarf [13], [19]

	3.1.1 nicht reentrant	3.1.2 reentrant	3.1.3 reentrant
Bytes	13	10	14
States	61	51	70

Methode 3.1.2 ist offensichtlich am günstigsten. Allerdings hat Methode 3.1.3 den Vorteil, daß sich der Programm-Ersteller nicht um die Speicherorganisation der verschiedenen Datenblöcke (wie z.B. bei Methode 3.1.2) zu kümmern braucht, was eventuell eine Erleichterung in der Programmerstellung bringt.

3.2 ASCII - Dezimal - Umwandlung

Das Unterprogramm ASDEC wandelt ein Ziffer, die in ASCII-Code dargestellt ist (´0´... ´9´), in einen entsprechenden dezimalen Wert um.
(z.B.: `5`= 35H = 53D -> 5D)
Wenn das vorgegebene Zeichen keine ASCII-Ziffer ist, ist das Ergebnis 0FFH (Fehlermeldung).
Benutzte Register: A

a) Nicht-reentrante Schreibweise

Benutzung eines externen Speichers zur Parameterübergabe. Das zu prüfende Zeichen steht im Speicher unter der Adresse 40H. Die Antwort unter Adresse 41H.

```
ASDEC:    LDA  40H       ; HOLE ASCII
          SUI  ´0´       ; SUBTRAHIERE ASCII ´0´
          CPI  10        ; IST ERGEBNIS KLEINER 10
          JC   JA        ; JA: SPEICHERE ERGEBNIS
          MVI  A,0FFH    ; NEIN: ERGEBNIS = FF
JA:       STA  41H       ;
          RET
```

b) Reentrante Schreibweise

Benutzung des Akkumulators zur Parameterübergabe. Das zu prüfende Zeichen sowie die Antwort stehen im Akku.

```
ASDEC:    SUI  ´0´
          CPI  10
          JC   JA
          MVI  A,0FFH
JA:       RET
```

möglicher Aufruf:

```
        LDA  ASDAT     ; LADE AKKU MIT ASCII-ZEICHEN
        CALL ASDEC     ; UMWANDLUNG
        CPI  0FFH      ; FEHLER?
        JZ   ERROR     ; JA: FEHLER-ROUTINE
```

c) Bemerkung

Wenn, wie bei diesem Beispiel, ein oder nur wenige Parameter zu übergeben sind, so ist es auf jeden Fall vorteilhaft, die CPU-Register zu benutzen.
Die Anzahl der Parameter ist hier natürlich begrenzt durch die Anzahl der vorhandenen Register (A,B,C,D,E,H,L).

d) Vergleich beider Methoden in Bezug auf Rechenzeit und Speicherbedarf

	a)	**b)**
Bytes	16	10
States	67	41

3.3 Größter Wert aus einem Feld von 8-bit-Zahlen [19]

Das Unterprogramm MAX findet den größten Wert aus einem Feld von 8bit-Zahlen und schreibt diese Wert in den Akkumulator.
Die Länge des Feldes muß vor Aufruf des Unterprogrammes in Register B geschrieben werden.
Verwendete Register: A,B,H,L

a) Feld in Datenbereich mit festgelegter (konstanter) Anfangsadresse

```
MAX:    LXI  H,8000H   ; ANF.ADR. DES FELDES
ZIEL1:  MOV  A,M       ; ZAHL IN AKKU
ZIEL2:  DCR  B         ; SCHLEIFENZÄHLER ERNIEDRIGEN
        JZ   ZIEL3     ; FELD ABGEARBEITET
        INX  H         ; ADRESSE ERHÖHEN
        CMP  M         ; AKKU GRÖßER (IND. H&L)
        JNC  ZIEL2     ; JA
        JMP  ZIEL1     ; NEIN
ZIEL3:  RET
```

nicht reentrant

Das Unterprogramm hat praktisch seinen eigenen Datenbereich, daher ist es nicht reentrant.

b) Anfangsadresse des Feldes in H&L

```
MAX:    MOV  A,M
ZIEL1:  DCR  B
        JNZ  ZIEL2
        INX  H
        CMP  M
        JNC  ZIEL1
        JMP  MAX
ZIEL2:  RET
```

reentrant

möglicher Aufruf:

```
        ...
        LDA  MENGE
        MOV  B,A
        LXI  H,ADR
        CALL MAX
        ...
```

Hier kann im Gegensatz zu a) auf die erste Instruktion `LXI H,8000H` verzichtet werden. Reentrant ist das Unterprogramm deshalb, weil es keinen eigenen Variablenbereich hat, sondern es benutzt jeweils den vom aufrufenden Programm übergebenen Bereich. Somit kann es von mehreren übergeordneten Programmen mit verschiedenen Datenfeldern aufgerufen bzw. auch unterbrochen (Interupt) werden, ohne daß bei einer Unterbrechung noch gebauchte Daten durch Überschreiben verloren gehen können.

c) Übergabe der Daten im Stack

Benutzte Register: A,B,D,E,H,L

```
MAX:    POP  H          ; RÜCKSPRUNGADRESSE
        XCHG            ; IN D&E
        POP  H          ; AUS STACK IN H&L
ZIEL1:  MOV  A,L        ; REGISTER L IN AKKU
ZIEL2:  DCR  B          ; SCHLEIFENZÄHLER ERNIEDRIGEN
        JZ   ZIEL3      ; SPRUNG WENN FERTIG
        POP  H
        CMP  L          ; VERGLEICH AKKU MIT REG. L
        JNC  ZIEL2      ; SPRUNG WENN AKKU GRÖSSER L
        JMP  ZIEL1      ; SPRUNG WENN AKKU KLEiNER L
ZIEL3:  XCHG            ; RÜCKSPRUNGADRESSE IN H&L
        PCHL            ; RÜCKSPRUNGADRESSE IN PC = RET
```

reentrant

möglicher Aufruf:

```
...
LDA  ZAHL1
PUSH PSW
...
LDA  ZAHLN
PUSH PSW
CALL MAX
...
```

Es ist zu beachten, daß lediglich das Low-Order-Byte (LOB) des jeweiligen Registerpaares im Unterprogramm benutzt wird. Das High-Order-Byte (HOB) wird im Unterprogramm ignoriert. Dies ist zwar eine Verschwendung von Stack-Speicherplatz, wurde aber gewählt, damit das Unterprogramm übersichtlicher bleibt und mit den vorherigen Beipielen besser zu vergleichen ist.

d) Stackpointer mit Anfangsadressen des Feldes besetzen

Wie bei Methode c) wird der Stack zur Datenübergabe benutzt. Allerdings werden die Daten nicht in den Stack gebracht (PUSH), sondern einfach der Stackpointer an die Anfangsadresse des jeweiligen Datenfeldes gesetzt, bevor das Unterprogramm aufgerufen wird.
In Gegensatz zu c) wird hier vorausgesetzt, daß das LOB und das HOB Daten enthalten, sodaß das Unterprogramm zwangsläufig umfangreicher wird.

möglicher Aufruf: [17], [19]

```
...
LXI  H,0
DAD  SP
SHLD SAVSP     ; SP RETTEN
LXI  SP,DATA   ; ANF.ADR. DES FELDES IN SP
CALL MAX1
LHLD SAVSP     ; SP ZURÜCK
SPHL
...
```

Vor dem Aufruf muß die Adresse des „normalen“ Stacks zwischengespeichert werden und danach wieder in den Stackpointer eingesetzt werden, da der Stack noch wichtige Daten enthalten kann.
Da nach einer Unterbrechung die ISR ihren eigenen Datenbereich benutzt, d.h. den Stackpointer mit der Anfangsadresse ihres Datenbereiches besetzt, werden keine Daten des Hauptprogrammes überschrieben.

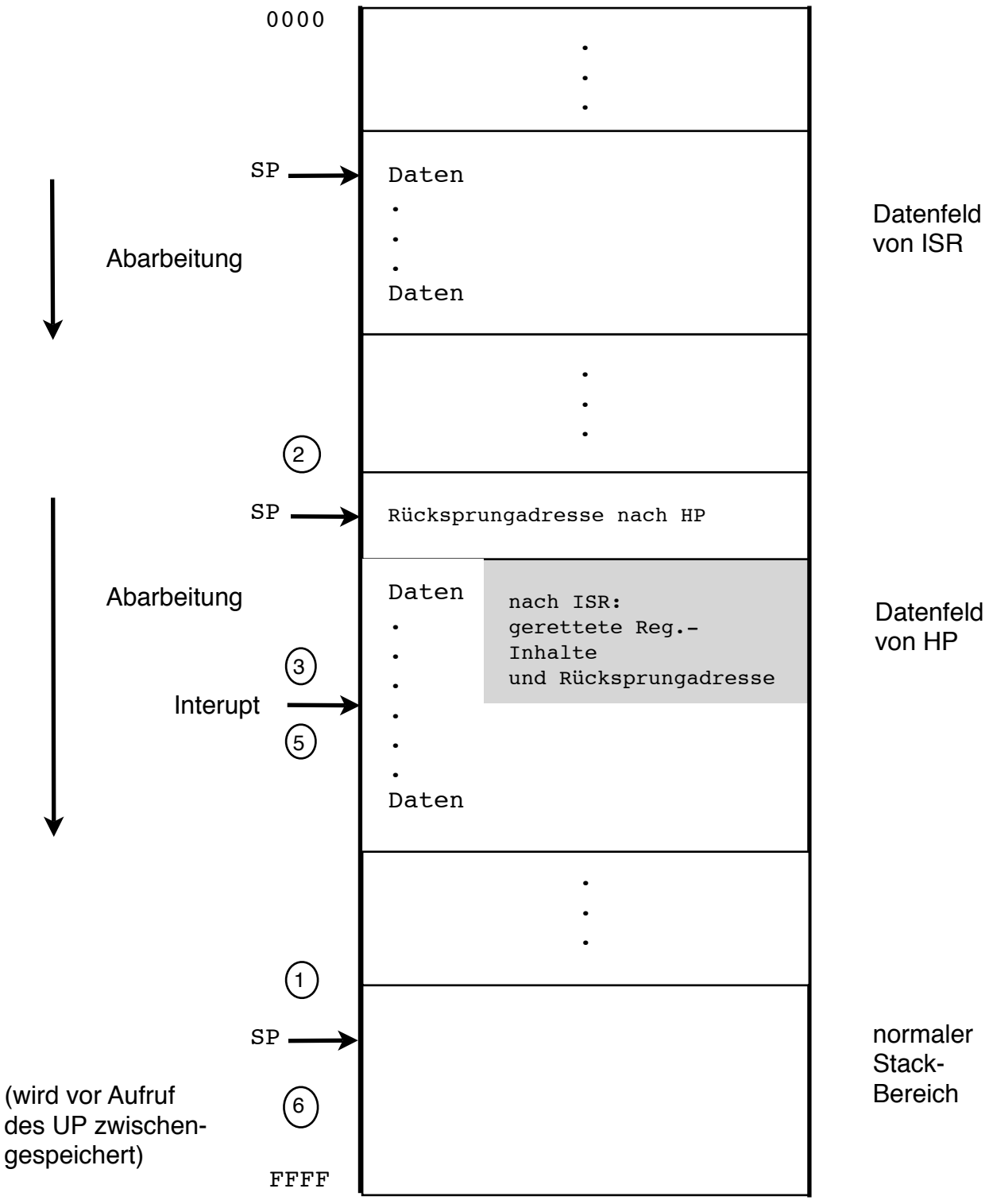
```
MAX1:     POP  H          ; RÜCKSPRUNGSADRESSE
          XCHG            ; IN D&E
          INR  B
          XRA  A
ZIEL1:    DCR  B
          JZ   ZIEL3
          POP  H
          CMP  L          ; VERGLEICHE AKKU MIT REG L
          JNC  ZIEL2      ; SPRUNG WENN AKKU GRÖßER L
          MOV  A,L
ZIEL2:    DCR  B
          JZ   ZIEL3
          CMP  H
          JNC  ZIEL1      ; SPRUNG WENN AKKU GRÖ?ER H
          MOV  A,H
          JMP  ZIEL1
ZIEL3:    XCHG
          PCHL            ; RÜCKSPRUNG
```

Stackinhalte während der Bearbeitung:

Es wird vorausgesetzt, daß das Hauptprogramm vor dem Aufruf die Adresse des Stacks zwischenspeichert (1), und den Stackpointer mit der Anfangsadresse des Datenfeldes besetzt (2).
Tritt nun während der Bearbeitung des Datenfeldes im Unterprogramm ein Interupt auf (3), erfolgt indirekt ein Aufruf der ISR. Diese speichert wiederum den Inhalt des Stackpointers, allerdings in einem anderen Speicherplatz als das Hauptprogramm, und setzt dafür die Anfangsadresse ihres eigenen Datenbereiches ein (4).
Nach Ablauf des Unterprogrammes und der ISR wird die erste Bearbeitung weitergeführt (5), und mit dem Ergebnis zum Hauptprogramm zurückgesprungen, wo der alte SP-Inhalt wieder eingesetzt wird (6).

Wichtig ist, daß nach Ablauf der ISR die Daten, die schon benutzt wurden, gelöscht sind (sie wurden überschrieben mit Rücksprungadresse und den geretteten Registerinhalten von der ISR).
Dies muß im Hauptprogramm berücksichtigt werden; möglicherweise müßte vor Aufruf des Unterprogrammes eine Kopie des Datenfeldes angefertigt werden.

Darum ist die Methode nicht zu verwenden, wenn im Unterprogramm ein Datenbereich mehrmals in einer Schleife bearbeitet wird.


0000
SP
Daten
Daten
Datenfeld von ISR
Abarbeitung
2
SP
Rücksprungadresse nach HP
Abarbeitung
Daten
nach ISR:
gerettete Reg.-
Inhalte
und Rücksprungadresse
Datenfeld von HP
3
Interupt
5
Daten
1
SP
normaler Stack-Bereich
(wird vor Aufruf des UP zwischen-gespeichert)
6
FFFF

e) Ablegen der Daten unmittelbar hinter dem Unterprogrammaufruf [17]

```
MAX:      POP  H          ; ADR. DES ERSTEN WERTES IN H&L
ZIEL1:    MOV  A,M
ZIEL2:    INX  H
          DCR  B                        reentrant
          JZ   ZIEL3
          CMP  M
          JNC  ZIEL2
          JMP  ZIEL1
ZIEL3:    PUSH H          ; NEUE RÜCKPRUNGADRESSE IN STACK
          RET
```

Die beiden letzten Instruktionen können ersetzt werden durch: `PCHL`

Die Rücksprungadresse, die vom CALL-Befehl in den Stack gebracht wird, adressiert den ersten Wert des Datenfeldes und nicht den nächsten Befehl. Diese Methode ist nur anzuwenden, wenn das aufrufende Programmteil, und somit das Datenfeld in einem RAM-Speicher steht.

möglicher Aufruf mit angenommenen Werten:

```
          .
          .
          .
8010H :   MVI  B,7
8012H :   CALL MAX
8015H :   (Zahl1)
8016H :   (Zahl2)
.                          \
.                           >  Datenfeld
.                          /
801AH :   (Zahl6)
801BH :   (Zahl7)
801CH :   .
          .
          .
```

Das Unterprogramm addiert praktisch zum niederwertigen Byte der Rücksprungadresse den Inhalt des Registers B (Größe des Feldes) und interpretiert das Ergebnis als neue Rücksprungadresse.

3.4 Möglichkeiten der Abspeicherung von Zwischenergebnissen in Unterprogammen

Wie in Anschnitt 2.3.2 schon erwähnt, gibt es verschiedene Möglichkeiten, Zwischenergebnisse abzuspeichern, auch wenn die CPU-Register nicht verwendet werden.

Beispiel:

Addition mit mehrfacher Genauigkeit [13], [19]

Addition von zwei Mehrwort-Binärzahlen. Die Länge der Zahlen (in Worten) befindet sich im Akkumulator, die Startadresse der Zahlen in den Registerpaaren D&E und H&L und die Startadresse des Ergebnisses in B&C. Alle Zahlen beginnen mit den niederwertigen Bits.

a) nichtreentrante Schreibweise

```
MADD:     ANA  A          ; ANFÄNGLICHER ÜBERTRAG = 0
ZIEL:     STA  20H        ; ZAHLENLÄNGE ZWISCHENSPEICHERN
          LDAX D          ; HOLE WORT VON ERSTER ZAHL
          ADC  M          ; ADDIERE WORT VON ZWEITER ZAHL
          STAX B          ; SPEICHERE WORT DES ERGEBNISSES
          INX  B          ; BRINGE ZEIGER
          INX  D          ; AUf NEUESTEN
          INX  H          ; STAND
          LDA  20H        ; ZAHLENLÄNGE IN AKKU
          DCR  A          ; ZÄHLER AUF NEUEN STAND
          JNZ  ZIEL       ; ZURÜCK
          RET
```

Da das Programm den Akkumulator zur Addition benötigt, wird der verlorene Inhalt (Zahlenlänge) im Speicherplatz 20H zwischengespeichert.
Tritt während des Bearbeitung ein Interupt auf, der das Unterprogramm wieder zu einem Ablauf veranlasst, so wird die neue Zahlenlänge in denselben Speicherplatz gebracht, wodurch der alte Wert überschrieben wird.
Nach Wiederaufnahme der ersten Bearbeitung steht im Speicher ein falscher Schleifenzähler. Es werden daher je nach Situation zu viele oder zu wenige Bytes addiert, was zu einem falschen Ergebnis führt.

b) reentrante Schreibweise

```
MADD:     PUSH B          ; ADRESSE DES ERGEBNISSES IN STACK
          MOV  B,A
          ANA  A
ZIEL:     LDAX D
          ADC  M
          XTHL            ; HOLE ADRESSE DES ERGEBNISSES
          MOV  M,A        ; ERGEBNIS ABSPEICHERN
          INX  H
          XTHL
          INX  D                    reentrant
          INX  H
          DCR  B
          JNZ  ZIEL
          POP  B          ; ELEMINIEREN DER ADRESSE
          RET
```

Bei dieser Methode wird mit dem Befehl `XTHL` praktisch die Spitze des Stacks als ein zusätzliches Registerpaar verwendet.
Da sich bei einer Unterbrechung und einem Wiedereintritt alle vorherigen Zwischenergebnisse im Stack befinden und nicht überschrieben, sondern lediglich „überdeckt“ worden sind, ist diese Methode reentrant.

Für reentrante Unterprogramme, die viele Zwischenergebnisse abspeichern müssen, gibt es neben der Benutzung des Stacks eine andere Möglichkeit.
Wie bei der Parameterübergabe (siehe 3.1 und 3.3) wird dem Unterprogramm die Anfangsadresse eines Datenfeldes übergeben, welches das Unterprogramm frei benutzen kann. Jedes aufrufende Programm muß dann natürlich ein separates Feld bereitstellen.

3.5 Sich selbst verändernder Code

Wenn bei einem Unterprogramm die Forderung besteht, anhand verschiedener Bedingungen an bestimmte, zu berechnende Stellen zu springen, kann dies folgendermaßen realisiert werden:

Ausschnitt eines Unterprogrammes mit angenommenen Adressen:

```
          .
          .
0400H     LDA  LOB       ; HOLE LOB DES SPRUNGZIELES
0403H     STA  0DH       ;
0406H     LDA  HOB       ; HOLE HOB DES SPRUNGZIELES
0409H     STA  0EH       ;
040CH     JMP  ...       ; SPRINGE AN VORGEGEBENES ZIEL
040DH     ...
040EH     ...
          .
          .
```

nicht reentrant

Vor dieser Befehlssequenz muß das Sprungziel aufgrund der vorgegebenen Bedingung errechnet und in die Speicherplätze LOB und HOB geschrieben werden.
Diese Methode ist nur möglich, wenn das Programm in einem RAM-Speicher steht.
Angenommen, während der Abarbeitung des Befehles von Adresse 0409H tritt ein Interupt auf. Bei Wiedereintritt in das Unterprogramm durch eine ISR mit anderen Parametern (und Bedingungen) wird eine andere Sprungadresse eingesetzt und der Sprungbefehl richtig ausgeführt.
Nach Beendigung des zweiten Durchlaufes des Unterprogrammes erfolgt ein Rücksprung zur ISR. Nach Beendigung der ISR wird der erste Durchlauf des Unterprogrammes fortgesetzt mit dem JMP-Befehl, in dessen zweitem und dritten Byte jetzt aber das Sprungziel steht, das aus den Bedingungen der ISR resultierte und nicht mit dem zuerst errechneten übereinstimmt (Übereinstimmung wäre zufällig).
Eine reentrante Schreibweise wäre z.B. dadurch zu erreichen, indem man die errechnete Sprungadresse in das Registerpaar H&L bringt und den inhalt von H&L in den Programm-Counter (PC) transferiert:

```
          .
          .
          LDA  LOB
          MOV  L,A
          LDA  H,A
          PCHL
          .
          .
```

reentrant

Diese Methode hat zusätzlich den großen Vorteil, daß das so geschriebene Programm auch in einem ROM stehen kann.

Literaturverzeichnis

[1] Programming Languages, Information Structures and Machine Organization
Peter Wegner
Mc GRAW-HILL BOOK COMPANY

[2] Messen, Steuern, Regeln mit Prozeßrechnern
Max Syrbe
Akademische Verlagsgesellschaft Frankfurt

[3] Microcomputer-based Design
John B. Peatman
Mc GRAW-HILL BOOK COMPANY

[4] Maschinennahes Programmieren von Mikrocomputern
Günter Koch
BI-Verlag

[5] Programming for Microcomputers
J. C. Cluley
Edward Arnold Ltd.

[6] Microprocessors and small digital Computer Systems for Engineers and Scientists
Granino A. Korn
Mc GRAW-HILL BOOK COMPANY

[7] Minicomputers for Engineers and Scientists
Granino A. Korn
Mc GRAW-HILL BOOK COMPANY

[8] Microcomputer Experimentation with the INTEL SDK-85
Lance A. Leventhal, Colin Walsh
PRENTICE-HALL INC.

[9] Interactive Systeme Vol. 2
W. G. Spruth
Science Research Associates GmbH

[10] System Programming
John J. Donovan
Mc GRAW-HILL BOOK COMPANY

[11] Structured concurrent programming with operating systems applications
Holt/Graham/Lazonska/Scott
Addison-Wesley Publishing Company

[12] Introduction to operating System Design
A.N. Habermann
Science Research Associates

[13] 8080A/8085 Programmieren in Assembler
Lance A. Leventhal
te-wi Verlag

[14] Emulations- und Testadapter ETA 80
Bedienungsanleitung
Siemens

[15] Minicomputer Systems
R.H. Eckhouse Jr.
L.R. Morris

[16] Assembler Language Programming
Struble

[17] Mikrocomputer Bausteine
Microcomputer System SAB 8085
Siemens Datenbuch 1980/81

[18] 8080/8085 Assembly Language Programming Manual
Order Number 9800301C
Intel Corporation

[19] Introduction to Microprocessors. Software, Hardware, Programming
Lance A. Leventhal
Prentice-Hall-INC

[20] 8080/8085 Software Design. Book 2
C.A. Titus, D.G. Larsen, J.A. Titus
Howard W. Sams & Co., INC

[21] Informatik II
Steinbuch/Weber
Springer Verlag