Mikroprozessoren und Mikrorechner

Lehrheft 3

ITT Fachlehrgänge, Pforzheim

ITT

Druck: Druckerei Seiter, 7535 Königsbach-Stein

Mikroprozessoren und Mikrorechner

Verfasser:
Dr. Jürgen Gerlach
C. D. Nabavi, B. Sc.
Forschungszentrum der
Standard Elektrik Lorenz AG
Stuttgart

5. Programmierung

Für Anfänger auf dem Gebiet der Rechnertechnik ist es häufig schwierig, mit dem Schreiben eigener Programme in Assemblersprache zu beginnen, obwohl sie solche Programme ohne weiteres lesen und auch verstehen können. Für diese Art der Programme ist eine bestimmte Denkungsweise erforderlich. Grundsätzlich gilt, daß ein gegebenes Problem in primitive, nacheinander ablaufende Einzeloperationen aufgelöst werden muß, die der Mikrorechner ausführen kann. Außerdem müssen die Einzeloperationen so ausgewählt sein, daß die Eigenschaften des verwendeten Computers möglichst vorteilhaft zur Lösung des Problems eingesetzt werden. Dazu kommen vor allem bei etwas größeren Programmen noch die Fragen einer geeigneten Strukturierung, d.h. einer sinnvollen Aufgliederung des Gesamtprogramms in einzelne möglichst abgeschlossene Unterfunktionen oder Programm-Moduln. Diese Strukturierung ist ein entscheidender Punkt bei der Programmplanung, da sie die Verständlichkeit des Programms erheblich beeinflußt. Ein gut strukturiertes Programm ist leichter verständlich, mit geringerem Aufwand zu testen und leichter zu modifizieren, wenn sich die Anforderungen geändert haben.

Gute Programmierung erfordert Erfahrung. In diesem Abschnitt sollen einige grundlegende Programmiertechniken erläutert und geübt werden. Damit wird eine Basis geschaffen für eigene Arbeiten auf diesem Gebiet.

Ein Programm für ein einfaches Problem, wie z.B. die Addition von 3 Zahlen, ist ohne nennenswerte Überlegungen hinzuschreiben. Die Struktur des Programms ist offensichtlich und muß deshalb nicht erst erarbeitet werden. Anders ist es dagegen bei Programmen zur Lösung komplexer Problemstellungen. Hier muß der Programmierer viele Fälle unterscheiden und unterschiedliche Programmpfade vorsehen. Die Folge der Operationen ist kompliziert, so daß das Schreiben des Codes, also des Programms, eine sehr komplexe und oft verwirrende Aufgabe ist.

Eine große Hilfe bietet hierbei ein **Programmablaufplan,** auch **Flußdiagramm** genannt. Ein Flußdiagramm ist eine grafische Darstellung der Problemlösung, es gibt den logischen Zusammenhang und den Ablauf der Operationen an, die der Rechner ausführen soll. Eine solche logische, formalisierende Darstellung des Programmablaufes erleichtert zunächst schon die Suche nach der optimalen Lösung, und dann auch die Aufgabe des Umsetzens in die Rechnerbefehle, also die eigentliche Codierung. Auch beim Testen eines Programms ist das Flußdiagramm eine große Erleichterung.

Ein Flußdiagramm besteht lediglich aus einer Reihe von einzelnen Kästchen, die durch Linien miteinander verbunden sind. Diese Kästchen haben unterschiedliches Aussehen, je nach Art der Operation, die sie symbolisieren. In Bild 5.1 sind die wichtigsten Symbole eines Flußdiagramms zusammengestellt.

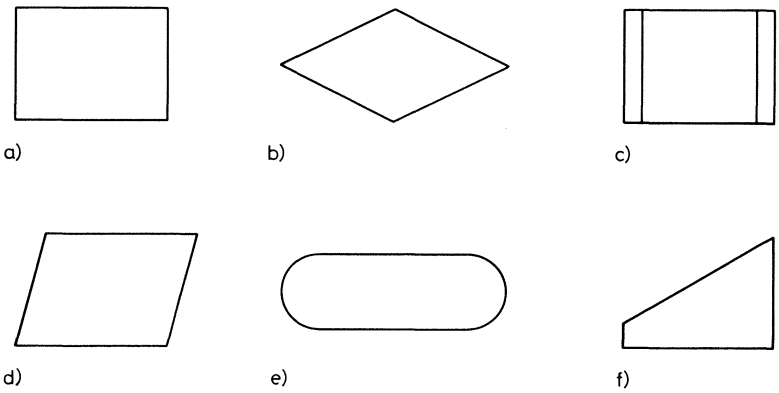


Bild 5.1
Wichtige Symbole eines Flußdiagramms bzw. Programmablaufplanes
a) Kennzeichnung einer allgemeinen Operation
b) Kennzeichnung einer Verzweigung
c) Anruf eines Unterprogramms
d) Kennzeichnung von Ein-Ausgabe-Operationen
e) Grenzstellen eines Flußdiagramms, z. B. START, HALT
f) Dateneingabe von Hand

Bild 5.2 zeigt ein Flußdiagramm für eine Addition von 3 Zahlen.

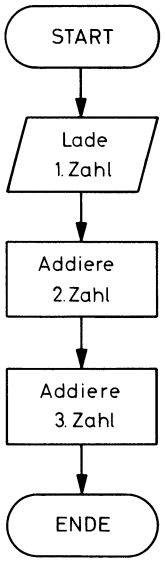


Bild 5.2
Flußdiagramm für eine Addition von 3 Zahlen

Die Operationen sind in der richtigen Reihenfolge aufgeführt, das Programm läuft von START bis ENDE ab.
Ein weiterer Punkt bei der Programmierung ist die Programmverzweigung. Hier wird ein Programmpfad in 2 Programmpfade aufgespalten. Je nachdem, ob eine bestimmte Bedingung erfüllt ist oder nicht, wird der eine oder andere Pfad eingeschlagen.
So können z.B. vom Vorzeichen einer Zahl verschiedene Aktionen abgeleitet werden. Wenn beispielsweise der Kontostand ein „Minus" aufweist, soll der Kunde gemahnt werden, ist er positiv, wird ein normaler Kontoauszug erstellt (Bild 5.3).

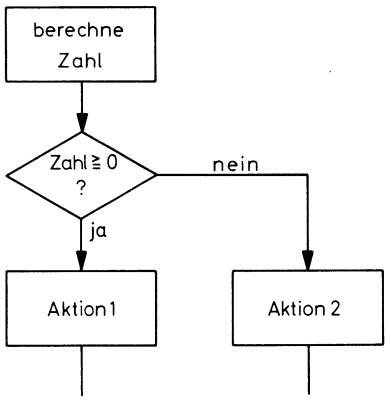


Bild 5.3
Beispiel einer Programmverzweigung

Mit solchen Verzweigungen können beliebig komplexe Programmabläufe realisiert werden.

5.1 Programmbeispiele

5.1.1 Schleifen

Das Prinzip der Schleife wurde schon mehrmals benutzt. Eine Schleife besteht aus bis zu 3 Teilen (Bild 5.1.1.1).
Den ersten Teil der Schleife bildet die **Initialisierung.** Hier werden der Schleifenzähler und die Register in den Anfangszustand gebracht. Jetzt setzt die eigentliche Schleife selbst ein (2. Teil) und endet mit einem bedingten Sprungbefehl. Ist die Endbedingung erreicht, kann dieser Programmteil abgeschlossen werden (3. Teil).

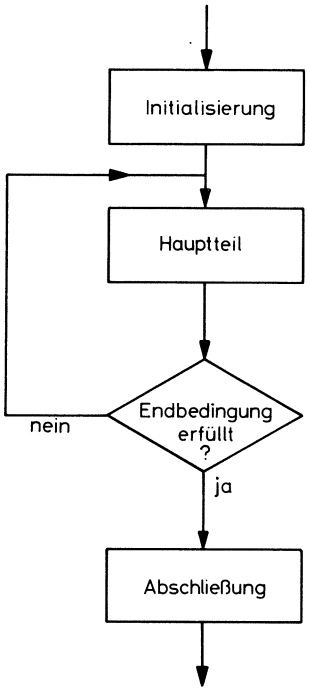


Bild 5.1.1.1
Flußdiagramm einer Schleife

Exp. 7

5.1.2 Computed-JUMP

Der Computed-JUMP ist ein wichtiger Begriff der Programmierung. Wir wollen ihn an einem Beispiel erläutern:
Eine Zahl zwischen 0_{16} und 7_{16} wird von den *A*-Schaltern in das Experimentiersystem einge-

Adresse	Inhalt	Befehl	Kommentar
4 0	8 4	LOAD R0, F F	*A*-Schalter einlesen
4 1	F F		
4 2	3 0	IORR R0, R0	LOAD setzt Z-Flag nicht
4 3	E 1	JMPZ 07	erster Sprung
4 4	0 7		
4 5	6 4	DECR R0	
4 6	E 1	JMPZ 7 0	zweiter Sprung
4 7	7 0		
4 8	6 4	DECR R0	
4 9	E 1	JMPZ D 8	dritter Sprung
4 A	D 8		
4 B	6 4	DECR R0	
4 C	E 1	JMPZ A 7	vierter Sprung
4 D	A 7		
4 E	6 4	DECR R0	
4 F	E 1	JMPZ 2 8	fünfter Sprung
5 0	2 8		
5 1	6 4	DECR R0	
5 2	E 1	JMPZ F 3	sechster Sprung
5 3	F 3		
5 4	6 4	DECR R0	
5 5	E 1	JMPZ 0 1	siebter Sprung
5 6	0 1		
5 7	E 0	JUMP 2 A	achter Sprung
5 8	2 A		

Tab. 5.1.2.1
Programm zum Flußdiagramm in Bild 5.1.2.1

geben. Je nach eingegebener Zahl muß der Mikrorechner eines von 8 Programmen auswählen, die bei den Adressen 0 7, 7 0, D 8, A 7, 2 8, F 3, 0 1, 2 A anfangen. Bei Eingabe der Zahl 0 muß er zu Programm 0 7 springen, bei 1 auf Programm 7 0, bei 2 auf Programm D 8 usw. Ein Flußdiagramm hierfür ist in Bild 5.1.2.1 dargestellt.

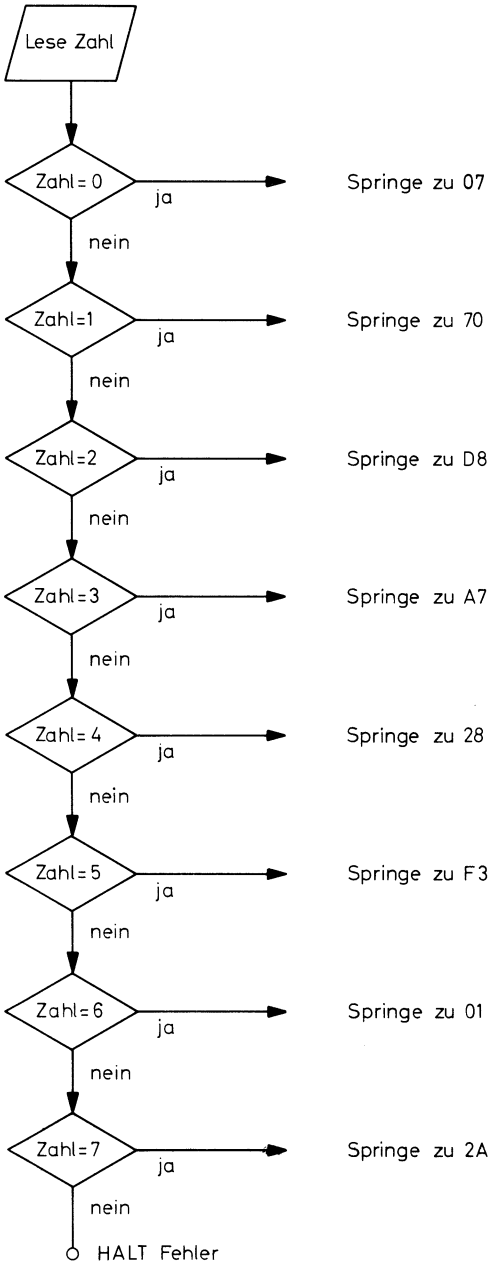


Bild 5.1.2.1
Mehrfachverzweigung abhängig vom Wert einer Variablen

Tab. 5.1.2.1 zeigt hierfür ein Programm.
In diesem Programm wird 1 ständig von der eingegebenen Zahl abgezogen. Wenn das Ergebnis 0 ist, erfolgt ein Sprung zu der angegebenen Adresse. Der Befehl IORR R0, R0 dient dazu, das Z-Flag beim ersten Schritt zu setzen bzw. zu löschen. Der letzte JUMP-Befehl ist unbedingt, weil hier nur noch die Zahl 7 übrigbleibt.
Dieses Programm ist relativ lang und würde bei entsprechend größeren Zahlen noch umfangreicher.
Betrachten wir das Programm in Tab. 5.1.2.2.
Dieses Programm hat nur 3 Befehle und eine Tabelle der Sprungadressen. Wenn z. B. an den *A*-Schaltern eine 3 steht und mit dem ersten Befehl in R2 eingelesen wird, wird im zweiten Befehl hierzu 5 0 addiert. Damit ist der Inhalt von R2 jetzt 5 3. Der Befehl JUMP @ R2 be-

Adresse	Inhalt	Befehl	Kommentar
4 0	8 6	LOAD R2, F F	lese *A*-Schalter ein
4 1	F F		
4 2	9 2	ADDM R2, #5 0	
4 3	5 0		
4 4	E 8	JUMP@ R2	
.	.		
.	.		
.	.		
5 0	0 7		
5 1	7 0		
5 2	D 8		
5 3	A 7		
5 4	2 8		
5 5	F 3		
5 6	0 1		
5 7	2 A		

Tab. 5.1.2.2
Verkürztes Programm zum Flußdiagramm in Bild 5.1.2.1

sagt, daß der Befehlszähler auf die Adresse springen soll, die durch den Inhalt der Adresse 5 3 festgelegt ist.
Dieser sogenannte Computed-JUMP-Befehl (Sprungbefehl mit ausgerechneter Adresse) braucht nur 13 Wörter gegenüber 25 der ersten Version. Außerdem ist er viel übersichtlicher und schneller.

Exp. 8

5.1.3 Programm zur Multiplikation binärer Zahlen

In Lehrheft 1 wurde bereits die Multiplikation zweier Binärzahlen besprochen. Dabei wurde deutlich, daß die Regeln der binären Multiplikation denen der dezimalen Multiplikation entsprechen. In diesem Beispiel soll ein Algorithmus entwickelt werden, nach dem dann ein Programm für den hypothetischen Mikrorechner erstellt werden kann.
Gegeben sind zwei 8-bit-Zahlen, die zu multiplizieren sind:

```
1 0 0 1 1 0 1 1 · 1 1 0 0 1 1 0 1
1 0 0 1 1 0 1 1
  1 0 0 1 1 0 1 1
    0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0
        1 0 0 1 1 0 1 1
          1 0 0 1 1 0 1 1
            0 0 0 0 0 0 0 0
              1 0 0 1 1 0 1 1
```

Das Beispiel zeigt zunächst, daß das Produkt zweier 8-bit-Zahlen nicht mehr in ein 8-bit-Register paßt, es benötigt 16 bit, also doppelte Genauigkeit. Ferner ist zu erkennen, daß die Multiplikation in mehreren Schritten ausgeführt werden muß. Für jede Stelle des Multiplikators ist ein Rechenschritt erforderlich. In diesem Beispiel sind es also 8 Schritte, plus die Gesamtaddition als letzten Schritt.
Die einzelnen Schritte sind:

1. Schritt
Prüfe das linksstehende bit (bit 2^7) des Multiplikators; ist es eine 1, schreibe den Multiplikanden in die Additionstabelle, ist es eine 0, schreibe 8 mal 0 in die Additionstabelle.
2. Schritt
Prüfe bit 2^6 und verfahre wie in Schritt 1 aber um eine Stelle nach rechts verschoben.

Die Schritte 3 bis 8 laufen entsprechend ab.
Letzter Schritt:
Summiere alle 8 Summanden auf. Die Summe ist das gesuchte Produkt.

Dieser Algorithmus läßt sich umstellen und dadurch „rechnergeeignet" machen. Zunächst haben wir in unserem hypothetischen Rechner (dies gilt auch für reale Mikroprozessoren) keinen Befehl, um 8 Summanden in einem Schritt zu addieren. Daher muß die Summenbildung in mehreren Schritten ausgeführt werden.
Wenn es nun gelingt, die 8 Einzelschritte so zu spezifizieren, daß sie identisch werden, kann eine Programmschleife eingesetzt werden, die 8mal zu durchlaufen ist. So würde sich ein sehr kompaktes Multiplikationsprogramm ergeben.
Wir wollen jetzt das Multiplikationsprogramm analysieren und so formulieren, daß die angesprochenen Punkte berücksichtigt werden. Als Ergebnis der Analyse erhalten wir dann das Flußdiagramm des Multiplikationsprogramms.
In jedem Einzelschritt ist zunächst zu prüfen, ob ein bestimmtes bit einer Binärzahl 0 oder 1 ist. Abhängig hiervon sind unterschiedliche Additionen vom Programm durchzuführen, es ist also eine Verzweigung erforderlich.
Eine solche Verzweigung kann mit einem beliebigen Sprungbefehl realisiert werden. Dafür ist allerdings erforderlich, daß das zu prüfende bit in eines der Flags gebracht wird. Nur dann kann vom Zustand des Flags abgeleitet ein bedingter Sprung durchgeführt werden.
Wenn der Multiplikator bei jedem Schritt um eine Stelle nach links verschoben wird, so kommt im ersten Schritt bit 2^7, beim zweiten Schritt bit 2^6 usw. in das Carry-Flag. Mit einem bedingten Sprungbefehl, der den Zustand des Carry-Flags berücksichtigt, kann dann die Verzweigung ausgeführt werden. Das Teilflußdiagramm hierfür zeigt Bild 5.1.3.1.

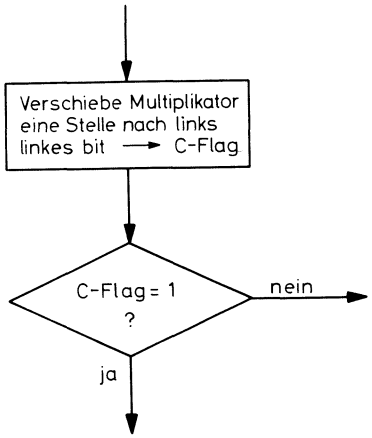


Bild 5.1.3.1
Prüfung der Stellen des Multiplikators auf 0 oder 1

Dieser Programmteil ist bei jedem Schritt durchzuführen.
Die nächste Aufgabe im Multiplikationsalgorithmus ist die stellenrichtige Addition des Multiplikanden bzw. der Zahl 0. Dies wird am besten dadurch erreicht, daß eine laufende Zwischensumme verwendet wird, in der bei jedem Einzelschritt der Multiplikand bzw. die 0 stellenrichtig hineinaddiert wird. Diese Zwischensumme muß am Programmanfang 0 betragen. Im 1. Schritt wird der Multiplikand bzw. 0 ganz links hineinaddiert, im nächsten Schritt um eine Stelle nach rechts verschoben usw. Die Zwischensumme benötigt doppelte Genauigkeit, d.h., die Addition ist in jedem Schritt mit doppelter Genauigkeit auszuführen, ebenso wie die Verschiebung des Multiplikanden nach rechts. In Bild 5.1.3.2 ist dieser Vorgang dargestellt.

Bild 5.1.3.2 zeigt, wo der Betrag von bit 2^7 des Multiplikanden im 1. Schritt schließlich im Endergebnis auftaucht. Das *x* markiert diese Stelle im Verlauf der Einzelschritte. Das *x* besagt allerdings nicht, daß das betreffende bit während der gesamten Rechnung unverändert bleibt. Bei den Additionen in den einzelnen Schritten können immer Überträge entstehen, die das markierte bit verändern. Das *x* deutet an, wo der Beitrag von bit 2^7 des Multiplikanden im 1. Schritt zu finden ist. Auch dieser Beitrag kann sich selbstverständlich durch einen Übertrag weiter links im Ergebnis auswirken, er kann aber nicht nach rechts wirken.

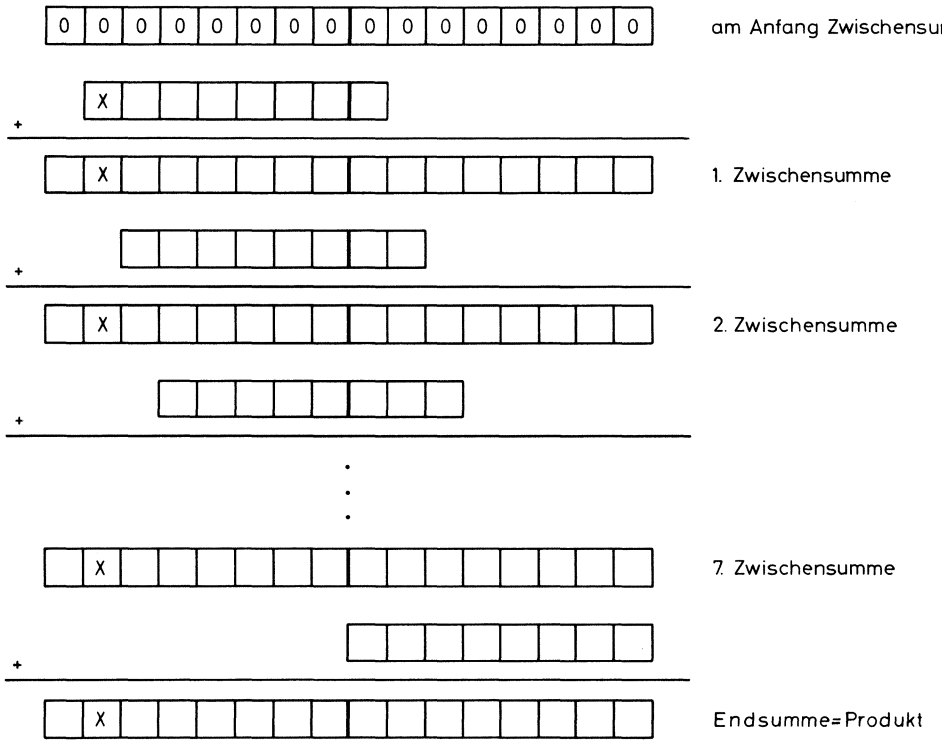


Bild 5.1.3.2
Multiplikationsalgorithmus bei Verschiebung des Multiplikanden nach rechts

Anstatt die Zwischensumme festzuhalten und den Multiplikanden nach rechts zu bewegen, kann man auch den Multiplikanden festhalten und die Zwischensumme nach links verschieben. Dies ist in Bild 5.1.3.3 dargestellt.

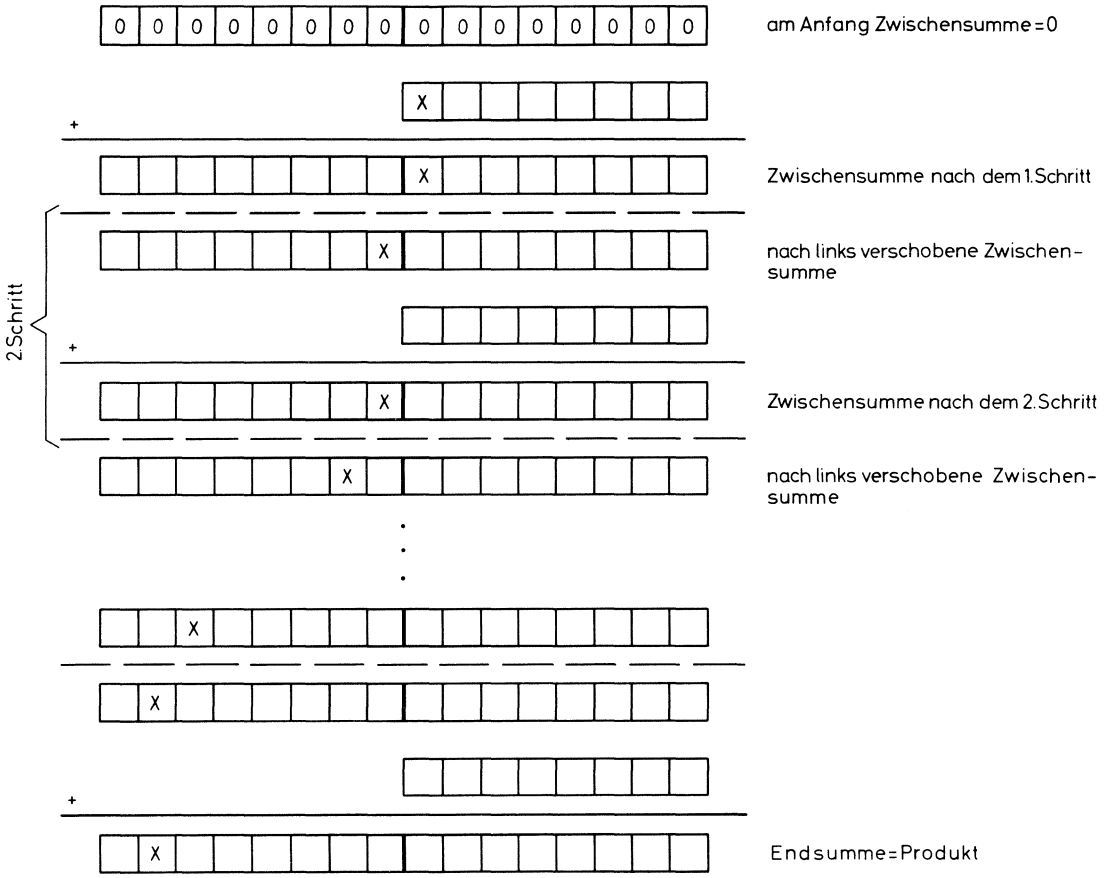


Bild 5.1.3.3
Multiplikationsalgorithmus bei Verschiebung der Zwischensumme nach links

Der Vergleich der Bilder 5.1.3.2 und 5.1.3.3 zeigt, daß beide Verfahren identische Ergebnisse liefern. Das Verfahren nach Bild 5.1.3.3 hat jedoch programmtechnisch verschiedene Vorteile. Die eigentliche Addition erfolgt hierbei an einer festen Stelle. Addiert werden nämlich stets die beiden Rechnerwörter, die die rechte Hälfte der momentanen Zwischensumme und den Multiplikanden bzw. 0 enthalten. Ergibt sich bei dieser Addition ein Übertrag muß zur linken Hälfte der momentanen Zwischensumme eine 1 addiert werden.
Damit ergibt sich ein Flußdiagramm entsprechend Bild 5.1.3.4.

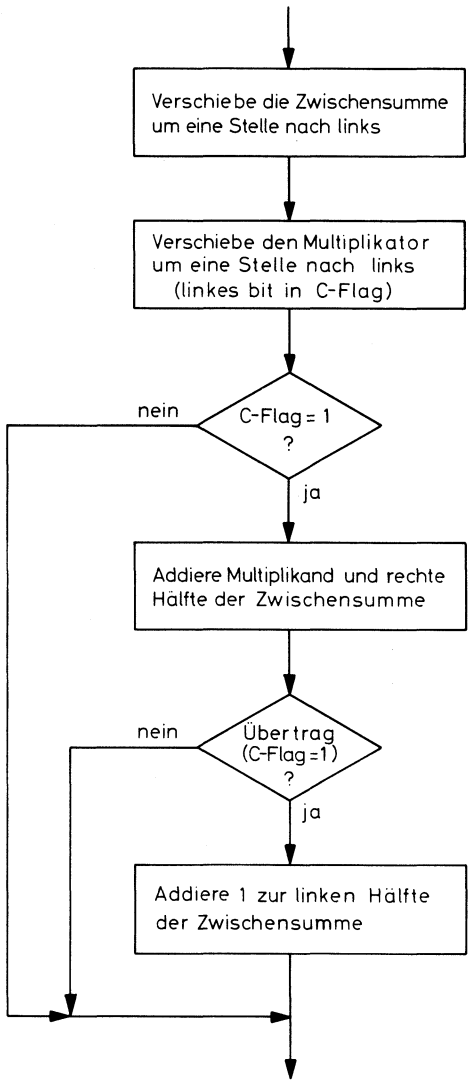


Bild 5.1.3.4
Flußdiagramm des Multiplikationsprogramms

Diese Operationen sind für die Schritte 2 bis 8 identisch auszuführen, beim 1. Schritt ist keine Verschiebung erforderlich. Der 1. Schritt beginnt mit der Zwischensumme 0. Daher können wir auch beim 1. Schritt die Verschiebung ausführen, da sich ja nichts ändern kann. Damit lassen sich die beschriebenen Operationen in eine Schleife einbauen, die 8mal durchlaufen werden muß. Hierzu wird ein Schleifenzähler benötigt, der zu Beginn des Programms, also in der Initialisierungsphase, auf 8 bit gesetzt wird. Mit jeder Schleife wird dieser Zähler decrementiert und über ein Flag auf den Zustand 0 überwacht. In der Initialisierung wird auch die Zwischensumme auf 0 gesetzt. Damit ergibt sich ein komplettes Flußdiagramm entsprechend Bild 5.1.3.5.
Dieses Flußdiagramm ist bereits nicht mehr ganz unabhängig vom Rechnertyp und seinen Instruktionen, da bereits Rechnereigenschaften wie z. B. das Verschieben ins Carry-Flag und die spezielle Art der Addition in 2 Stufen mit berücksichtigt werden. Man kann ein solches Flußdiagramm auch allgemeiner formulieren, es ist dann aber zwangsläufig weniger detailliert und somit für unseren Zweck weniger instruktiv.

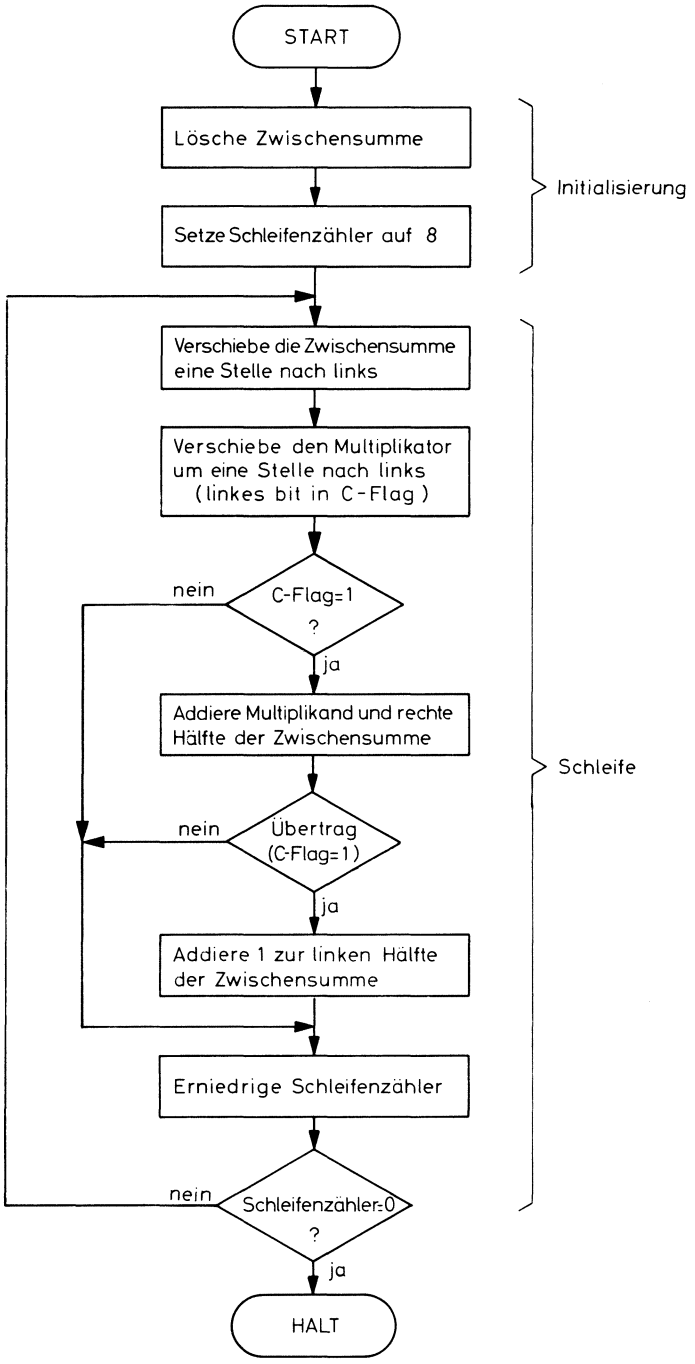


Bild 5.1.3.5
Vollständiges Flußdiagramm des Multiplikationsprogramms

Dieses Multiplikationsprogramm kann weiter vereinfacht und verkürzt werden, wenn folgendes berücksichtigt wird: Aus Bild 5.1.3.3 ist zu ersehen, daß beim Verschieben der Zwischensumme nach links auf der linken Seite immer Nullen herausgeschoben werden. Da insgesamt 8mal eine Verschiebung erfolgt, werden also alle Nullen, die zu Beginn in der linken Hälfte der Zwischensumme stehen, im Verlauf der Operation unverändert links herausgeschoben. Diese „verschwendeten" bit können wir besser nutzen, indem wir am Programmanfang nicht Nullen in die linke Hälfte der Zwischensumme laden sondern den Multiplikator. Die einzelnen bit des Multiplikators werden dann ebenso unverändert nach links in das Carry-Flag geschoben und können so geprüft und verarbeitet werden wie vorher. Der Multiplikator macht dabei durch das Verschieben nach links genau in dem Maße Platz wie es für die Zwischensumme erforderlich ist. Das so vereinfachte Flußdiagramm zeigt Bild 5.1.3.6.
Nachdem das Flußdiagramm aufgestellt ist, kann mit dem Schreiben des eigentlichen Programms begonnen werden. Die erste Aufgabe des Programmierers ist die Zuordnung der Re-

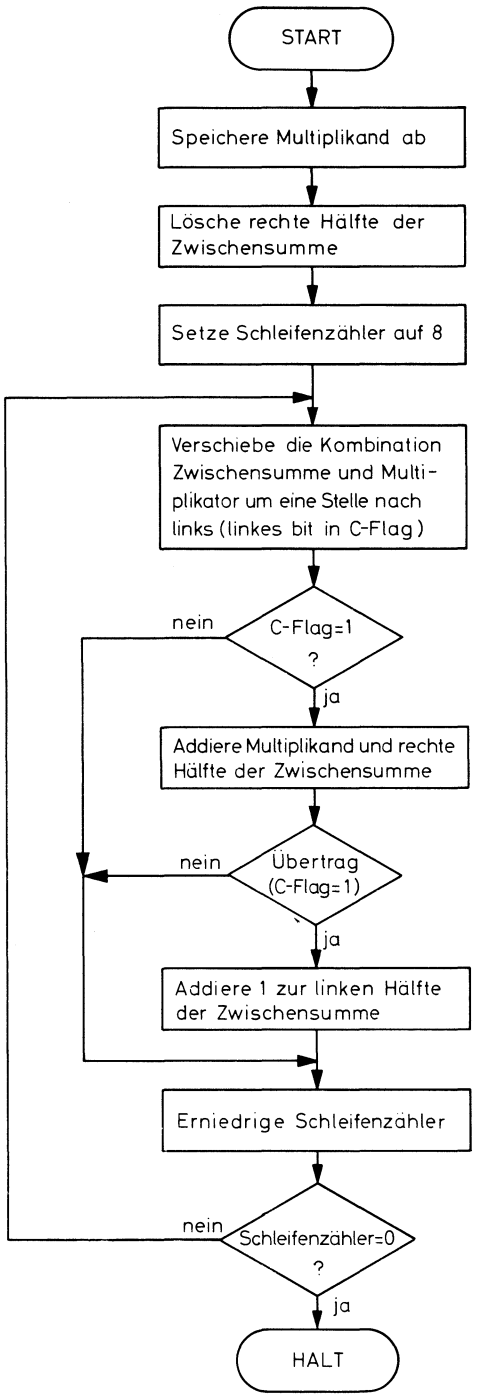


Bild 5.1.3.6
Flußdiagramm des verkürzten Multiplikationsprogramms

gister zu den Variablen im Programm. Im hypothetischen Mikrorechner stehen 4 Register zur Verfügung. Da die Verschiebebefehle nur auf Register, nicht aber auf Speicherzellen wirken, ist es von vornherein zweckmäßig, die Variablen, die verschoben werden müssen, in Registern zu halten. Das gleiche Argument gilt für den Schleifenzähler, er muß ja in der Schleife erniedrigt werden. Dafür eignet sich der DECR-Befehl, der ebenfalls nur auf Register wirkt. Wir können also beispielsweise folgende Zuordnung treffen:

R0 → linke Hälfte der Zwischensumme (Multiplikator)
R1 → rechte Hälfte der Zwischensumme
R2 → Schleifenzähler

Weiter werden folgende Festlegungen getroffen:
Die zu multiplizierenden Zahlen stehen vor Programmbeginn in den Registern R0 und R1.

Das Ergebnis soll am Schluß in den Adressen 52 und 53 gespeichert werden. Die Startadresse des Programms soll Adresse 20 sein. Jetzt kann das Flußdiagramm in ein Programm umgesetzt werden (Tab. 5.1.3.1).

Adresse	Inhalt	Befehl	Kommentar	
2 0	7 5	STAC R1, F 0	speichere Multiplikand	1)
2 1	F 0			
2 2	4 5	XORR R1, R1	lösche Zwischensumme und Carry-Flag	
2 3	8 2	LOAD R2,# 0 8	setze Schleifenzähler	
2 4	0 8			
2 5	6 9	RACL R1	verschiebe rechte Hälfte Zwischensumme	2)
2 6	6 8	RACL R0	verschiebe linke Hälfte Zwischensumme	
2 7	E 3	JMPC 2 B	prüfe Carry-Flag	3)
2 8	2 B			
2 9	E 0	JUMP 3 3		
2 A	3 3			
2 B	9 5	ADDM R1, F 0	C-Flag war 1, addiere	
2 C	F 0			
2 D	E 3	JMPC 31		4)
2 E	3 1			
2 F	E 0	JUMP 33		
3 0	3 3			
3 1	9 0	ADDM R0, #1	C-Flag war 1, addiere	5)
3 2	0 1			
3 3	6 6	DECR R2	erniedrige Schleifenzähler	
3 4	E 1	JMPZ 3 8	Schleifenende?	
3 5	3 8			
3 6	E 0	JUMP 2 5		
3 7	2 5			
3 8	7 4	STAC R0, 5 2	speichere Ergebnis	6)
3 9	5 2			
3 A	7 5	STAC R1, 5 3		
3 B	5 3			
3 C	0 0	HALT		

Tab. 5.1.3.1
Programm zum Flußdiagramm in Bild 5.1.3.6

Programmerläuterung:

1. Der Multiplikand steht in R1. Dieses Register wird später für die rechte Hälfte der Zwischensumme benötigt. Aus diesem Grunde muß der Multiplikand abgespeichert werden. Dies geschieht in unserem Beispiel unter der Adresse F 0.
2. Die Verschiebung der Zwischensumme erfolgt in 2 Schritten. Zuerst wird die rechte Hälfte (R1) verschoben. Das herausgeschobene bit kommt in das Carry-Flag. Danach wird die linke Hälfte (R0) verschoben. Der Inhalt des Carry-Flags wird dabei von rechts in R0 hineingeschoben, so daß insgesamt eine Verschiebung mit doppelter Wortlänge durchgeführt wird (Bild 5.1.3.7).

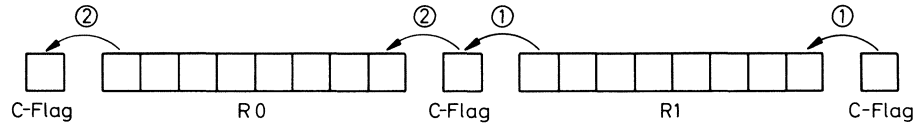


Bild 5.1.3.7
Verschieben der rechten und linken Hälfte der Zwischensumme.

Dabei ist noch zu beachten, daß durch den Befehl RACL 2 1 der Inhalt des Carry-Flags von rechts in R1 hineingeschoben wird. Über das Programm muß deshalb sichergestellt werden,

daß vor dieser Instruktion das Carry-Flag 0 enthält. Dies wird beim ersten Programmdurchlauf durch die Instruktion XORR R1, R1 garantiert. Dieser Befehl bewirkt nämlich ein Löschen des Carry-Flags. Auch in der Schleife muß diese Bedingung erfüllt sein.
3. und 4. Aus dem Flußdiagramm ist zu sehen, daß die folgenden Programmschritte (Addition) übersprungen werden müssen, wenn das Carry-Flag Null ist. Ein solcher Befehl ist im hypothetischen Rechner nicht vorhanden. Er muß daher durch einen bedingten und unbedingten Sprung realisiert werden, was natürlich zu einem längeren Programm führt.
5. Dieser Befehl addiert 1 zur linken Hälfte der Zwischensumme und löscht gleichzeitig das Carry-Flag, da sich bei dieser Addition kein Übertrag ergeben kann. Damit ist sichergestellt, daß im nächsten Schleifendurchlauf mit dem Befehl RACL R1 eine Null von rechts in die Zwischensumme geschoben wird. Zum Erhöhen von R0 könnte auf den ersten Blick auch der Befehl INCR R0 verwendet werden. Dies ist nicht möglich, da dieser Befehl das Carry-Flag nicht beeinflußt.

Exp. 9

6. Die Akkumulatorinhalte sind unter festen Adressen abzuspeichern. Hierfür verwenden wir direkte Adressierung.

Fragen zu Abschnitten 5. und 5.1

1. Erklären Sie den Begriff Flußdiagramm!

2. Was verstehen Sie unter einer Programmverzweigung?

3. Kann eine Programmschleife mit einem **unbedingten** Sprung gebildet werden?

4. Auf welche Adresse springt der Befehlszähler, wenn bei dem Computed-JUMP-Programm in Abschnitt 5.1.2 das Indexregister R2 über Adresse F F mit 5 geladen wird?

5. Das Multiplikationsprogramm in Abschnitt 5.1.3 liefert ein Ergebnis mit doppelter Genauigkeit.
a) Wie groß ist die Anzeigekapazität bei 16 bit Ergebniswortlänge?
b) Reicht diese Kapazität für die Multiplikation von F F · F F aus?

5.2. Unterprogramme

Eine gute Planung eines Systems setzt voraus, daß zunächst die Anforderungen an das Gesamtsystem genau analysiert werden. Danach wird dann dieser Gesamtkomplex in definierte, möglichst abgeschlossene Teilfunktionen gegliedert. Dies gilt sowohl für die Planung eines größeren Schaltungskomplexes wie auch für die Planung eines größeren Programms. Im ersten Fall werden die Teilfunktionen dann als Baugruppen realisiert, im zweiten Fall durch Programm-Moduln. Ein wichtiges Hilfsmittel bei dieser Programmgliederung in einzelne Moduln ist das **Unterprogramm** oder die **Subrontine.** Ein Unterprogramm ist ein Programm, das eine bestimmte Funktion ausführt, wenn es von einem anderen Programm, dem **Hauptprogramm,** angerufen wird. Nachdem die Funktion ausgeführt ist, übergibt das Unterprogramm die Kontrolle wieder zurück an das Hauptprogramm.
Nehmen wir an, in einem Programm ist 2mal dieselbe Funktion auszuführen, d.h. es enthält 2mal identische Programmteile. Diese Programmteile werden dann zweckmäßigerweise aus dem Programm herausgenommen und nur einmal getrennt in einen anderen Speicherbereich geschrieben. Wenn im Hauptprogramm dieses Programmstück gebraucht wird, dann springt der Rechner dorthin, arbeitet das Unterprogramm ab und springt anschließend zum Hauptprogramm zurück. Er kann auch zu einem späteren Zeitpunkt das Unterprogramm noch einmal anrufen, wobei der Rücksprung dann allerdings zu einer anderen Adresse im Hauptprogramm erfolgen muß. Dieses Prinzip ist in Bild 5.2.1 dargestellt.
Am Ende des Unterprogramms muß also eindeutig festgelegt sein, zu welcher Adresse der Rücksprung erfolgen muß. Diese Rücksprungadresse ist variabel und kann deshalb nicht fest im Programm programmiert werden. Sie muß daher beim Sprung zum Unterprogramm

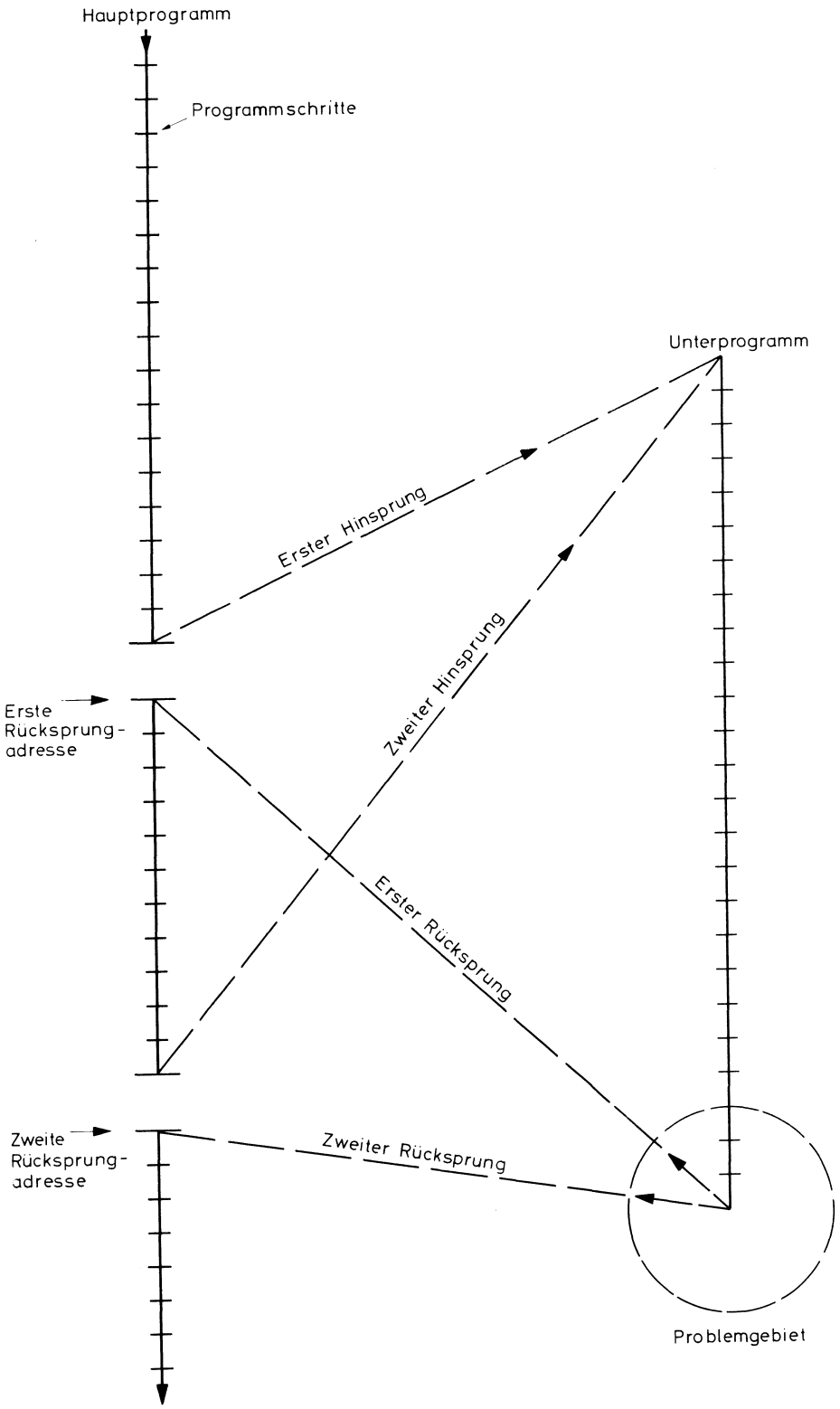


Bild 5.2.1
Prinzip eines Unterprogramms

gespeichert werden. Dieses wäre zu erreichen, indem man dem Hauptprogramm die Aufgabe gibt, die Rücksprungadresse in irgendeine vorgesehene Adresse, z.B. F 0, zu laden. Angenommen, das Unterprogramm beginnt bei Adresse 9 0. Seine Funktion soll sein, das Einerkomplement der Doppelgenauigkeitszahl in R0 und R1 zu bilden.
Dieses Programm hat die in Tab. 5.2.1 gezeigte Form.
Die ersten beiden Befehle komplementieren R0 und R1, der dritte Befehl ist der Rücksprung. Der Rechner springt zu der Adresse, die in der Adresse F 0 steht (indirekte Adressierung).

Adresse	Inhalt	Befehl	Kommentar
9 0	C 0	XORM R0, # F F	
9 1	F F		
9 2	C 1	XORM R1, # F F	
9 3	F F		
9 4	E 4	JUMP @ F 0	
9 5	F 0		

Tab. 5.2.1
Beispiel eines Unterprogramms

Bevor dieses Unterprogramm angerufen wird, muß zuerst die Rücksprungadresse in F 0 gespeichert werden. Dann muß mit einem normalen Sprungbefehl der Sprung zur Adresse 9 0 erfolgen. Tab. 5.2.2 zeigt ein Hauptprogramm, das das Unterprogramm 2mal anruft, d.h., die Daten in R0 und R1 werden komplementiert und zurückkomplementiert.

Adresse	Inhalt	Befehl	Kommentar
4 0	8 2	LOAD R2, # 4 6	Rücksprungadresse 1
4 1	4 6		
4 2	7 6	STAC R2, F 0	Speichere in F 0
4 3	F 0		
4 4	E 0	JUMP 9 0	Anruf 1
4 5	9 0		
4 6	8 2	LOAD R2, # 4 C	Rücksprungadresse 2
4 7	4 C		
4 8	7 6	STAC R2, F 0	Speichere in R 0
4 9	F 0		
4 A	E 0	JUMP 9 0	Anruf 2
4 B	9 0		
4 C	0 0	HALT	

Tab. 5.2.2.
Hauptprogramm mit 2 Unterprogrammanrufen

Die ersten beiden Befehle laden die 1. Rücksprungsadresse in F 0. Der dritte Befehl führt den Sprung zum Unterprogramm aus. Hier wird jetzt der Inhalt von R 0 und R1 komplementiert. Der Befehl in Adresse 9 4 bewirkt einen Sprung zu der Adresse, die in F 0 steht. Dies ist in unserem Falle Adresse 4 6 im Hauptprogramm. Bei Adresse 4 8 wird eine neue Rücksprungadresse in F 0 geladen und dann das Unterprogramm erneut angerufen. Bei Adresse 9 4 erfolgt wieder ein Rücksprung ins Hauptprogramm, diesmal zur Adresse 4 C. Auf diese Art und Weise kann das Unterprogramm beliebig oft angerufen werden. Bei diesem kleinen Beispiel besteht das Unterprogramm nur aus 2 Befehlen und dem Rücksprung. Es lohnt sich daher eigentlich nicht, ein Unterprogramm für diese Funktion zu verwenden, es sollte ja auch lediglich das Prinzip erläutert werden. Besteht jedoch ein Unterprogramm aus mehreren hundert Befehlen, so kann das Gesamtprogramm durch ein Unterprogramm doch wesentlich verkürzt werden. Außerdem wird es daduch übersichtlicher und ist somit leichter zu testen.

Exp. 10

5.2.1 Unterprogramme mit CALL-Befehlen

Unterprogramme werden in der Praxis sehr häufig benutzt. Das Programm des Experimentiersystems enthält z.B. etwa ein Dutzend Unterprogramme; eines davon wird über 50mal angerufen. Damit ein Unterprogrammanruf einfacher wird, muß die Rücksprungadresse gespeichert werden und nicht erst vom Programmierer in eine bestimmte Adresse geladen werden. Deshalb haben praktisch alle Mikroprozessoren einen besonderen Sprungbefehl, den

sog. **Subrontine-Call-** oder **Jump-to-Subrontine-Befehl,** kurz CALL-Befehl genannt. Dieser Befehl hat eine doppelte Aufgabe: Er muß einmal den Sprung zum Unterprogramm ausführen und zum anderen sicherstellen, daß nach Ablauf des Unterprogramms ein Rücksprung zu der dem CALL-Befehl folgenden Befehlsadresse im Hauptprogramm erfolgt. Dazu wird die Rücksprungadresse abgespeichert, bevor der eigentliche Sprung ausgeführt wird. Dieser Vorgang ist in Bild 5.2.1.1 dargestellt.

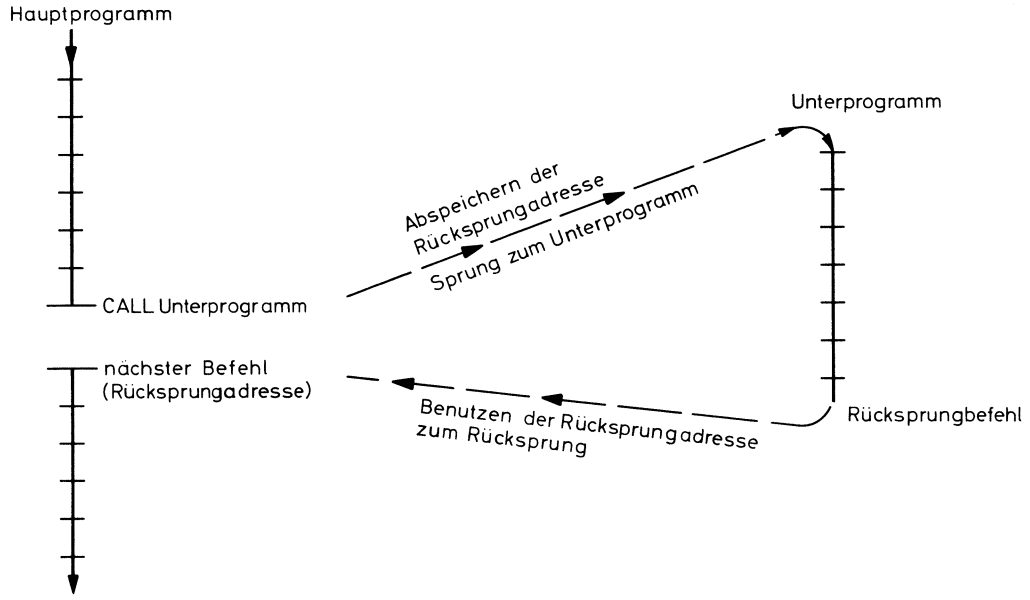


Bild 5.2.1.1
Anruf eines Unterprogramms mit dem CALL-Befehl

Wird der CALL-Befehl aus dem Speicher geholt (Fetch-Zyklus) enthält der Programmzähler natürlich die Adresse des CALL-Befehles. Im Laufe der Befehlsausführungen wird dann, wie bei anderen Befehlen auch, der Programmzähler erhöht. Er wählt also die nächste Instruktion bzw. den nächsten Befehl an. Bei einer normalen Instruktion wäre damit der Programmzähler für den nächsten Fetch-Zyklus bereit. Im Falle des CALL-Befehles ist die nächste Instruktion die Rücksprungadresse, also die Adresse des ersten Befehles, der nach Ablauf des Unterprogramms ausgeführt werden muß. Der CALL-Befehl wird also in 2 Schritten ausgeführt:
– Abspeichern des Inhaltes des inkrementierten Programmzählers, also der Rücksprungadresse
– Laden des Programmzählers mit der Sprungadresse, also Ausführung des Sprunges zum Unterprogramm
In der Praxis ist häufig eine Verschachtelung von Unterprogrammen notwendig. D.h., in einem Unterprogramm wird ein Unterprogramm angerufen, das u.U. wieder ein Unterprogramm anruft (Bild 5.2.1.2).
Aus Bild 5.2.1.2 geht hervor, daß für jeden Unterprogrammanruf eine Rücksprungadresse gespeichert werden muß. Diese muß so lange aufbewahrt werden, bis der entsprechende Rücksprung ausgeführt ist. In Mikroprozessoren werden die Rücksprungadressen auf verschiedene Art abgespeichert. Dies kann in speziell zugeordneten Registern, im Hauptspeicher usw. erfolgen. Die Art, wie die Rücksprungadressen gespeichert werden, beeinflußt die Möglichkeit der Verschachtelung von Unterprogrammen. Am flexibelsten ist die Speicherung der Rücksprungadressen im Hauptspeicher.
Aus diesem Grunde wird diese Art in der Praxis am häufigsten verwendet. Sie soll deshalb genauer behandelt werden.
Bei dem Beispiel in Bild 5.2.1.2 muß beim ersten Unterprogrammanruf die Rücksprungadresse Ad 1, beim zweiten Anruf die Adresse Ad 2, dann Ad 3 usw. gespeichert werden. Diese Rücksprungadressen müssen in entgegengesetzter Reihenfolge zum Rücksprung verwendet werden, in der sie anfallen. D.h., die zuerst abgespeicherte Rücksprungadresse wird als letzte benutzt usw. Man braucht daher zum Speichern der Rücksprungadressen von Unterprogrammen eine sog. **First-In-Lost-Out**-Speicherorganisation, auch **Push-Down-Stack** genannt.

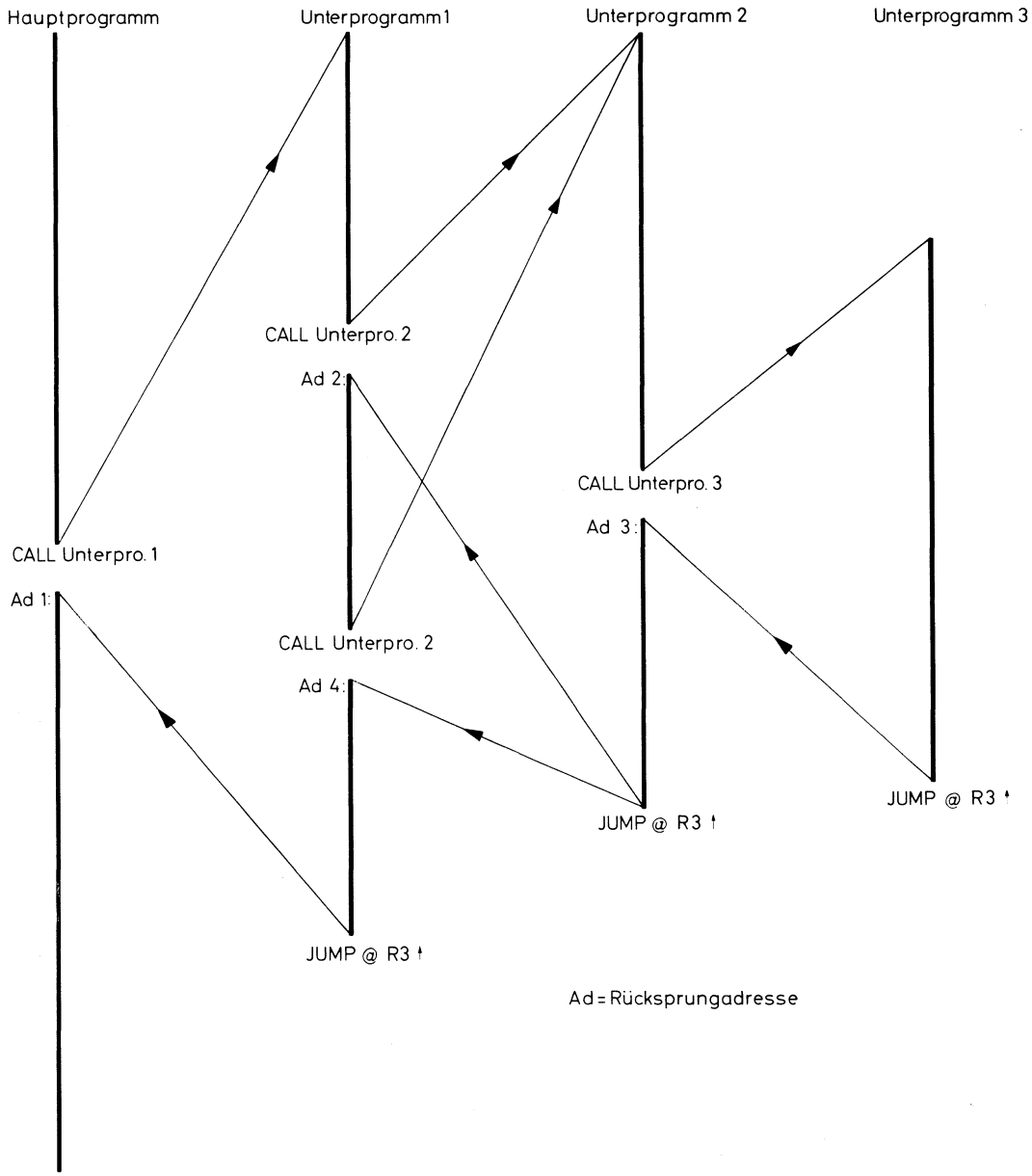
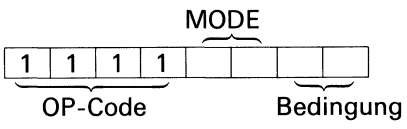


Bild 5.2.1.2
Verschachtelte Unterprogramme

Durch Anwendung einer geeigneten Adressierung bei CALL- und Rücksprungbefehl entsteht im Rechner automatisch eine solche Push-Down-Stack-Organisation. Der Befehl „CALL Unterprogramm" verwendet Auto-Decrement-Indexed-Adressierung über R3 zur Abspeicherung der Rücksprungadressen. Dadurch wird der Inhalt von R3 zunächst um 1 erniedrigt, so daß der neue Inhalt von R3 nun die Adresse ist, in der die Rücksprungadresse im Hauptspeicher abgespeichert wird. Dann wird der Sprung zum Unterprogramm ausgeführt. Beim nächsten CALL-Befehl läuft der gleiche Vorgang ab, so daß die Rücksprungadressen im Speicher „untereinander" abgelegt werden. Der Rücksprung von allen Unterprogrammen wird mit dem Befehl JUMP @ R3 ↑ ausgeführt. Dieser Befehl verwendet den Inhalt von R3 als Adresse. Der Inhalt dieser Adresse ist die Sprungadresse. Anschließend wird R3 um 1 erhöht.
Beim hypothetischen Mikrorechner hat der CALL-Befehl den OP-Code F. Wie beim JUMP-Befehl kann der CALL-Befehl auch unbedingt oder vom Zustand eines der 3 Flags abhängig sein. Das Format des CALL-Befehles ist:



Tab. 5.2.1.1 zeigt diese 4 Bedingungen.

Bedingung	Befehl	Bedeutung
0 0	CALL	CALL unbedingt
0 1	CALZ	CALL, wenn Zero-Flag = 1
1 0	CALN	CALL, wenn Negativ-Flag = 1
1 1	CALC	CALL, wenn Carry-Flag = 1

Tab. 5.2.1.1
Bedingungen der CALL-Befehle

Mit den MODE-bit können die gleichen Adressierungsarten wie beim Sprungbefehl gewählt werden. Allerdings hat MODE 1 1 (Auto-Increment-Indexed-Adressierung über R3) keinen Sinn, weil diese Adressierungsart R3 als Quelle für die Sprungadresse und dann zur Speicherung der Rücksprungadresse benützen würde. Es ist schwierig sich einen Fall vorzustellen, bei dem dies nützlich sein könnte. Daher ist MODE 1 1 beim CALL-Befehl zu vermeiden.

Die Anwendung des CALL-Befehles und die Verschachtelung von Unterprogrammen soll nun am Beispiel von Bild 5.2.1.2 näher erläutert werden.
Wir nehmen an, daß zur Speicherung der Rücksprungadressen der Speicherbereich von Adresse 9 5 bis zur Adresse 9 9 zur Verfügung steht. Das Register R3 soll hierbei als Stack-Pointer (SP) benutzt werden. Der Ausdruck Stack-Pointer kann mit „Stapel-Zeiger" übersetzt werden, d.h., der Stack-Pointer zeigt an, welche der gestapelten Rücksprungadressen zu einem bestimmten Zeitpunkt benutzt werden muß. Am Anfang des Programms wird R3 mit der Zahl 9 A geladen, also mit einer Adresse, die um 1 höher ist, als die obere Adresse des vorgesehenen Speicherbereiches:

9 9	
9 8	
9 7	
9 6	
9 5	

(SP) = 9 A

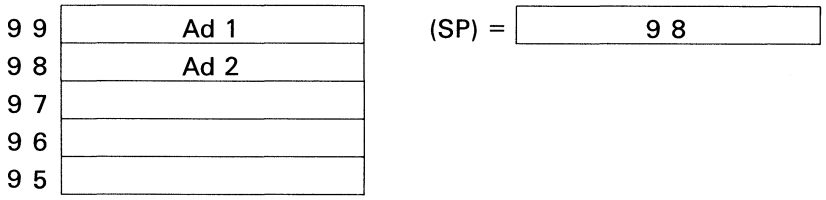
Dies ist der Zustand vor dem ersten CALL-Befehl. Wird nun im Hauptprogramm der Befehl CALL Unterpro 1 ausgeführt, so wird die Rücksprungadresse in Auto-Decrement-Indexed-Adressierung über den Stack-Pointer SP abgespeichert. Zuerst wird also der Stack-Pointer R3 um 1 auf 9 9 erniedrigt, so daß er auf die obere Adresse des Speicherbereiches zeigt.

Dann wird die Rücksprungadresse, also der Inhalt des inkrementierten Programmzählers, unter der Adresse abgespeichert, die im Stack-Pointer enthalten ist. Die Rücksprungadresse Ad 1 gelangt also in Adresse 9 9. Dann wird der eigentliche Sprung ausgeführt, d.h., die Adresse des Unterprogramms 1 wird in den Programmzähler geladen. Nach dem ersten CALL-Befehl ergibt sich folgender Zustand:

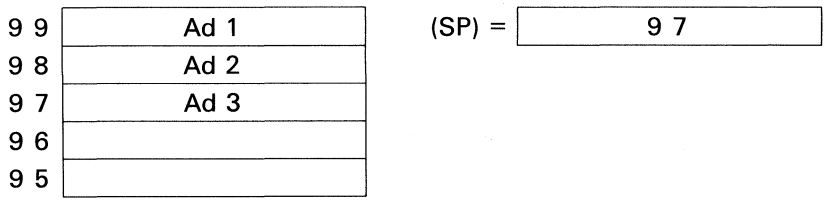
9 9	Ad 1
9 8	
9 7	
9 6	
9 5	

(SP) = 9 9

Nachdem der erste Teil des Unterprogramms 1 ausgeführt ist, folgt der Befehl CALL Unterpro 2. Die zugehörige Rücksprungadresse ist Adresse Ad 2. Zunächst wird wieder der Inhalt des Stack-Pointers um 1 erniedrigt. Dann wird die Rücksprungadresse Ad 2 unter der Adresse abgespeichert, die im Stack-Pointer enthalten ist, also in Adresse 9 8:



In Unterprogramm 2 wird ein Unterprogramm 3 angerufen, dessen Rücksprungadresse Ad 3 ist. Sie wird entsprechend in Adresse 9 7 abgespeichert. Der Stack-Pointer R3 enthält dann 9 7:



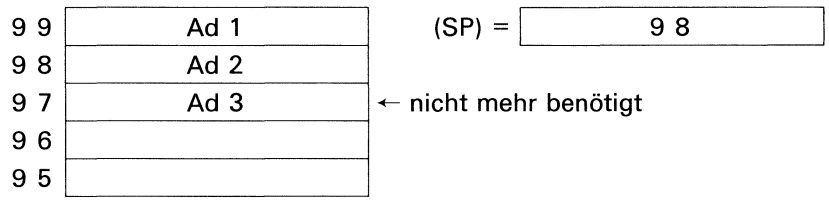
Das Unterprogramm 3 enthält keinen weiteren CALL-Befehl mehr. Ist dieses Unterprogramm ausgeführt, muß der Rücksprung zu dem anrufenden Programm erfolgen. Die betreffende Rücksprungadresse Ad 3 im Unterprogramm 2 steht an der letzten benutzten Stelle des Speicherbereiches (sie steht oben auf dem Stack). Diese Adresse wird vom Stack-Pointer markiert, der ja momentan die Adresse 9 7 enthält.
Wird nun zum Rücksprung der Befehl

JUMP @SP ↑

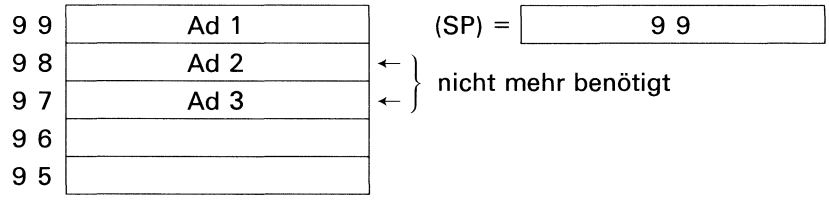
benutzt, so wird der Inhalt der Adresse, die im Stack-Pointer steht, in den Programmzähler geladen, also in diesem Fall Ad 3. Damit wird der Rücksprung zu Ad 3 im Unterprogramm 2 ausgeführt. Damit ist aber nur ein Teil der Aufgabe erledigt. Nach diesem Befehl enthält der Stack-Pointer immer noch die Rücksprungadresse 9 7. Die nächste Rücksprungadresse ist aber die Adresse 9 8, nämlich der Rücksprung in das Unterprogramm 1. Diese Erhöhung des Stack-Pointers ist durch den Rücksprungbefehl

JUMP @ SP ↑

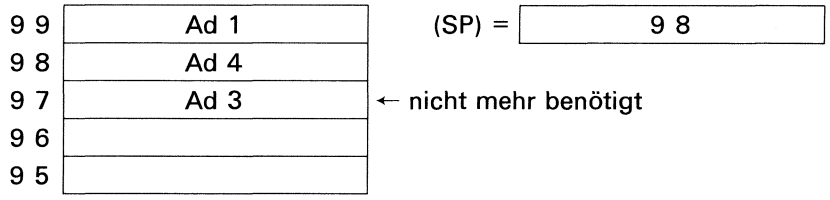
zu erreichen. Bei diesem Befehl wird der erste Teil der Aufgabe wie vorher beschrieben ausgeführt und anschließend der Stack-Pointer um 1 erhöht. Dieser Befehl wird deshalb häufig auch als Return oder Return from Subroutine bezeichnet. Nach dem ersten Rücksprung liegen also folgende Verhältnisse vor:



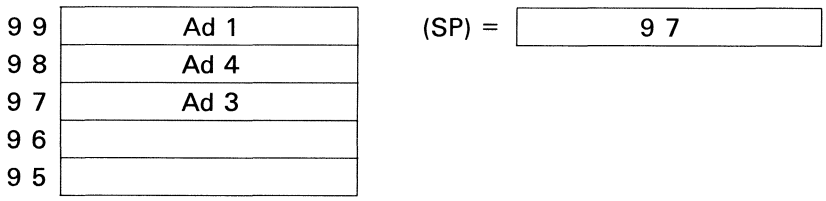
Nun werden die letzten Instruktionen von Unterprogramm 2 ausgeführt bis zum Rücksprungbefehl zu Ad 2 durch den Befehl JUMP @ SP ↑. Zunächst erfolgt der Sprung indirekt über den Stack-Pointer, also zu Ad 2 im Unterprogramm 1, dann erfolgt die Erhöhung des Stack-Pointers auf 9 9. Damit ergeben sich nach dem 2. Rücksprung folgende Verhältnisse:



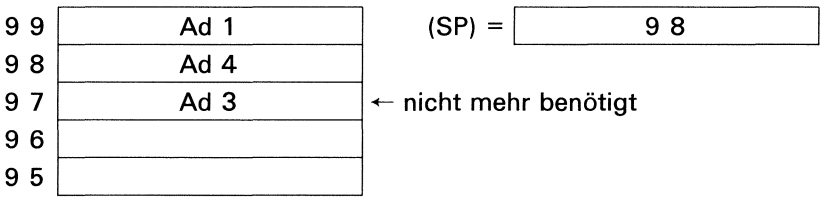
Nun läuft Unterprogramm 1 weiter bis zum Befehl CALL Unterpro 2. Dieser Befehl erniedrigt zunächst den Stack-Pointer um 1 auf 9 8 und speichert dann die Rücksprungadresse Ad 4 an der Stelle 9 8 ab. Dann erfolgt der Sprung zum Unterprogramm 2. Es ergeben sich folgende Verhältnisse:



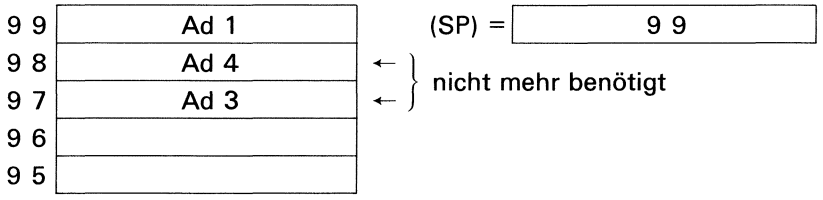
Die Rücksprungadresse Ad 2 wird hierbei also überschrieben. Dies ist jedoch bedeutungslos, da diese Adresse bereits zum Rücksprung verwendet wurde und deshalb nicht mehr benötigt wird.
Nun läuft dieses Unterprogramm 2 zum zweiten Mal ab. Im Laufe dieses Programms erfolgt wieder der Befehl CALL Unterpro 3. Die Rücksprungadresse, also Ad 3, wird unter der Adresse abgespeichert, die nach Erniedrigung im Stack-Pointer steht, also in Adresse 9 7. Zufällig steht hier vom vorherigen Anruf bereits Ad 3. Das hat keine Bedeutung, es ist bedingt durch die Anordnung der Anrufe im Beispiel. In einem anderen Fall würde der Inhalt von 9 7 durch Ad 3 überschrieben. Nach Abspeicherung der Rücksprungadresse erfolgt der Sprung zum Unterprogramm. Der Zustand des Stack ist jetzt folgender:



Der letzte Befehl im Unterprogramm 3 ist JUMP @ SP↑. Damit erfolgt ein Rücksprung zur Adresse Ad 3 und dann eine Erhöhung des Stack-Pointers:



Am Ende des Unterprogramms 2 steht ebenfalls der Befehl JUMP @ SP↑, d.h., es wird der Rücksprung zur Adresse Ad 4 durchgeführt und der Stack-Pointer auf 9 9 erhöht:



Nun wird bis zum Befehl JUMP @ SP↑ der Rest vom Unterprogramm 1 ausgeführt. Es erfolgt dann der Rücksprung zu Ad 1 im Hauptprogramm, der Stack-Pointer wird auf 9 A erhöht. Damit ist der Stack wieder leer, eine neue Rücksprungadresse könnte wieder in 9 9 abgespeichert werden.
Nach diesen ausführlichen Abhandlungen dürfte Ihnen die Funktion des CALL-Befehles klar geworden sein. Folgende zusätzliche Punkte sind noch zu berücksichtigen:

– Das Schreiben der Rücksprungadressen in den Stack wird in der Fachsprache auch als **Push** bezeichnet, das Zurückholen dieser Adressen auch als **Pop** oder **Pull.**
– Das Konzept des Push-Down-Stack kann dadurch erweitert werden, daß nicht nur die Rücksprungadressen, sondern auch Daten im Stack zwischengespeichert werden. Der Programmierer muß in diesem Falle aber unbedingt dafür sorgen, daß die Daten **vor** einem Rücksprung wieder vom Stack zurückgeholt werden. Wenn dies nicht erfolgt, geht die Synchronisation der Rücksprungadressen verloren, was dann natürlich zu völlig falschen Rücksprüngen führt. In der Praxis wird dieses Verfahren sehr häufig zum Abspeichern des **Prozessorstatus** angewandt. Gemeint ist hiermit das Abspeichern der momentanen Registerinhalte, das Abspeichern der Flag-Zustände usw. am Anfang eines Unterprogramms. Vor dem Rücksprung können dann die Register- und Flag-Zustände wieder zurückgelesen werden, so daß bei der Fortsetzung des Hauptprogramms der ursprüngliche Prozessorstatus wieder zur Verfügung steht. Durch diese Programmierungsart wird also erreicht, daß die Inhalte der Register und Flags durch das Unterprogramm nicht verändert werden, **obwohl** die Register und Flags im Unterprogramm benutzt werden. Wir werden auf diesen Punkt später noch näher eingehen.
– Eine zweite, sehr wichtige Anwendung von Push-Down-Stack ist das Auflösen arithmetischer Klammerausdrücke und das Lösen von Operationen mit unterschiedlichen Prioritäten. Dieses Prinzip wird daher in manchen Taschenrechnern sowie in Compilern, das sind Übersetzungsprogramme für höhere Programmiersprachen, verwendet.
– Das Abspeichern der Rücksprungadressen und die Rücksprünge zu den anrufenden Programmen bei verschiedenen Unterprogrammen werden über eine geeignete Adressierung gesteuert. Wie am Beispiel gezeigt, wird im CALL-Befehl die Rücksprungadresse in Auto-Decrement-Indexed-Adressierung indirekt über den Stack-Pointer (R3) abgespeichert. Der Rücksprung erfolgt mit Auto-Increment-Indexed-Adressierung über den Stack-Pointer. In manchen Mikroprozessoren ist die Zuordnung dieser Adressierungsarten vertauscht. An der Stack-Operation ändert sich dabei grundsätzlich nichts. Der einzige Unterschied ist der, daß eine Richtungsumkehr im Stack entsteht, d.h., der Stack-Bereich beginnt jetzt bei den niedrigeren Adressen.

Der Umfang des Stack-Bereiches wird bestimmt durch die Anzahl der Hierarchieebenen bei der Verschachtelung von Unterprogrammen. Im behandelten Beispiel wurden 3 Niveaus von Unterprogrammen angerufen. Damit ergaben sich im Stack 3 Einträge.
Bei vielen Mikroprozessoren wird, wie im Beispiel beschrieben, der Stack-Bereich im Hauptspeicher untergebracht. Der Programmierer muß deshalb im Speicher einen bestimmten Bereich hierfür reservieren. Wie viele Niveaus von Unterprogrammen miteinander verschachtelt werden sollen, ist bei dieser Art praktisch nur begrenzt durch den im Hauptspeicher zur Verfügung stehenden Platz. Neben diesen Mikroprozessoen gibt es auch Typen, die einen getrennten Stack-Bereich (z.B. in der CPU) aufweisen. Naturgemäß ist somit eine bestimmte Größe vorgegeben. Damit ist dann die Zahl der Niveaus der Verschachtelung begrenzt sowie auch die Möglichkeit, Daten im Stack zwischenzuspeichern. Wenn hierbei der vorgesehene Stack-Bereich zu klein ist, kommt man bei komplexeren Programmen schnell an eine Grenze, bei der eine vernünftige Aufgliederung in Unterprogramme schwer möglich ist. Bei praktischen Systemen ist dieser Punkt unbedingt zu berücksichtigen. Nur durch eine gute Strukturierung des Gesamtprogramms in einzelne Unterprogramme ist zu erreichen, daß das Programm übersichtlich bleibt, vernünftig zu testen ist und später auch einmal mit wenig Aufwand geändert werden kann.
Ein kleiner Nachteil des Aufgliederns eines Programms in einzelne Unterprogramme soll hier nicht unerwähnt bleiben. Der Anruf von Unterprogrammen kostet Zeit, ebenso wie der Transfer von Informationen in und aus Unterprogrammen, da hierzu zusätzliche Instruktionen ausgeführt werden müssen. Bei sehr zeitkritischen Programmen, z.B. Programmen, die sehr schnell Prozeßdaten in Echtzeit verarbeiten müssen, muß u.U. jede Möglichkeit zur Verringerung der Programmlaufzeit ausgenutzt werden. Hierzu gehört auch das Vermeiden jedes nicht wirklich unvermeidbaren CALL- oder JUMP-Befehles. Es sei jedoch nochmals wiederholt, daß die Übersichtlichkeit des Programms ein entscheidendes Entwurfskriterium ist. Programmiertricks machen häufig das Programm unübersichtlich. Man sollte deshalb, wenn solche Tricks unvermeidlich sind, zunächst durch eine besonders klare und detaillierte Dokumentation für einen Ausgleich sorgen.

Exp. 11

5.3 Argumente von Unterprogrammen

Der zuvor angesprochene Transfer von Informationen aus dem anrufenden Programm ins Unterprogramm sowie des Ergebnisses des Unterprogramms zurück ins anrufende Programm soll nachfolgend genau betrachtet werden.

Nehmen wir als Beispiel das früher erwähnte Unterprogramm zur Multiplikation zweier Zahlen. Dieses Unterprogramm muß vom Hauptprogramm den Multiplikanden und den Multiplikator erhalten. Am Ende des Unterprogramms muß das Produkt an das Hauptprogramm zurückgegeben werden. Diese Eingangs- und Ausgangsdaten eines Unterprogramms werden auch als **Argumente** des Unterprogramms bezeichnet. Zwischen dem Programmierer, der das Unterprogramm schreibt, und dem Programmierer der es in seinem Hauptprogramm verwendet, muß nun irgendeine Vereinbarung getroffen werden, wie diese Argumente von einem Programm zum anderen gegeben werden. Am Rande sei hierzu erwähnt, daß Mängel in diesem Kommunikationsprozeß schon häufig eine Menge Zeit und Geld gekostet haben, weil erst nach Beendigung der Programmierung bei der Systemintegration festgestellt wurde, daß die Programme nicht zusammenpassen.

Für die Übergabe von Daten zwischen Haupt- und Unterprogramm gibt es eine Reihe von Möglichkeiten. Man kann z.B. im Hauptprogramm die Daten in die verfügbaren Register laden und dann das Unterprogramm anrufen, das dann die Daten in den Registern findet. Auf die gleiche Art kann das Unterprogramm seine Ergebnisse zum Hauptprogramm transferieren. Diese Art des Argumenttransfers wird häufig angewendet. Sie ist allerdings auf solche Fälle begrenzt, bei denen nicht zu viele Daten übergeben werden müssen und freie Register zur Verfügung stehen.

Wenn wir z.B. aus dem Multiplikationsprogramm für den hypothetischen Mikrorechner ein Unterprogramm machen wollen, läßt sich diese Methode verwenden. Aus dem vorher aufgestellten Multiplikationsprogramm wird z.B. dadurch ein Unterprogramm (MULTR = Multiplikation von Registerinhalten), daß wir den in Adresse 3 C gespeicherten HALT-Befehl durch den Befehl JUMP @ R↑ ersetzen, der dann über den Stack-Pointer (R3) den Rücksprung ausführt. Das Unterprogramm MULTR führt die Funktion (R0) · (R1) = (R0R1) aus (die Inhalte von R0 und R1 werden multipliziert, das Ergebnis steht im Registerpaar R0R1).

Bevor wir nun mit den theoretischen Betrachtungen weiterfahren, müssen zunächst, damit Sie die hier entwickelten Programme auch auf dem MP-Experimenter testen bzw. nachvollziehen können, einige Punkte abgeklärt werden. Bereits in Lehrheft 2 wurde darauf hingewiesen, daß die Adressen von 0 0 bis 1 8 belegt sind. In die Adressen von 0 1 bis 1 8 ist über das ROM das behandelte Multiplikationsprogramm fest gespeichert, da dieses Programm sehr häufig benötigt wird. Tab. 5.3.1 zeigt deshalb das Multiplikationsprogramm unter Berücksichtigung der Adressen von 0 1 bis 1 8 noch einmal.

Die Adresse 1 9 gehört nicht mehr zum ROM-Bereich. In diese Adresse muß jetzt der JUMP @ R3 ↑-Befehl geladen werden.

Unter der Voraussetzung, daß, wie bereits früher besprochen, die beiden zu multiplizierenden Zahlen unter den Adressen 5 0 und 5 1 abgespeichert sind und das Ergebnis in 5 2 und 5 3 abgespeichert werden soll, benötigen wir das Hauptprogramm, das Tab. 5.3.2 zeigt.

Wenn mehr Argumente zu übergeben sind, als freie Register zur Verfügung stehen, so muß eine andere Art der Argumentübergabe verwendet werden. Man könnte z.B. die Vereinbarung treffen, daß das Hauptprogramm die Daten in bestimmte, festgelegte Adressen schreibt, aus denen sie dann das Unterprogramm abholt. Das Unterprogramm seinerseits verwendet ebenfalls vereinbarte Adressen zum Abspeichern der Ergebnisse. Auch dieses Verfahren ist in der Praxis üblich, es führt aber zu recht unübersichtlichen Programmabläufen. Eine Verbesserung in dieser Hinsicht wird erreicht, wenn die Argumente der Unterprogramme nicht in beliebig vereinbarte Stellen im Speicher, sondern unmittelbar nach dem CALL-Befehl gespeichert werden. Zur Lösung der vorigen Aufgabe würde man dann folgenden Unterprogrammanruf verwenden

RAM:

```
CALL Unterprogramm
Faktor 1
Faktor 2
PR          ; Platz für rechte Produkthälfte
PL          ; Platz für linke Produkthälfte
.
.
.
```

Adresse	Inhalt	Befehl	Kommentar
0 1	7 5	STAC R1, F 0	
0 2	F 0		
0 3	4 5	XORR R1, R1	
0 4	8 2	LOAD R2, # 0 8	
0 5	0 8		
0 6	6 9	RACL R1	
0 7	6 8	RACL R0	
0 8	E 3	JMPC 0 C	
0 9	0 C		
0 A	E 0	JUMP 1 4	
0 B	1 4		
0 C	9 5	ADDM R1, F 0	
0 D	F 0		
0 E	E 3	JMPC 1 2	
0 F	1 2		
1 0	E 0	JUMP 1 4	
1 1	1 4		
1 2	9 0	ADDM R0, # 0 1	
1 3	0 1		
1 4	6 6	DECR R2	
1 5	E 1	JMPZ 1 9	
1 6	1 9		
1 7	E 0	JUMP 0 6	
1 8	0 6		
1 9	E C	JUMP @R3↑	

Tab. 5.3.1
Fest eingespeichertes Multiplikationsprogramm

Adresse	Inhalt	Befehl	Kommentar
2 0	8 3	LOAD R3, # F F	Initialisiere Stack-Pointer
2 1	F F		
2 2	8 4	LOAD R0, 5 0	Lade Multiplikator
2 3	5 0		
2 4	8 5	LOAD R1, 5 1	Lade Multiplikand
2 5	5 1		
2 6	F 0	CALL 0 1	MULTR beginnt bei Adresse 0 1
2 7	0 1		
2 8	7 5	STAC R1, 5 2	Speichere Ergebnis
2 9	5 2		
2 A	7 4	STAC R0, 5 3	
2 B	5 3		
2 C	0 0	HALT	

Tab. 5.3.2
Hauptprogramm für die Multiplikation

Die Adressen der Faktoren und des Produktes sind nun nicht mehr beliebig abgespeichert, ihre Lage ist eindeutig festgelegt. Dieses Verfahren wird bei Prozeßrechnern, deren Programme im Kernspeicher stehen, häufig verwendet. Bei Mikrorechnern sind die Programme üblicherweise in einem Festwertspeicher (ROM) abgespeichert. Bei dieser Art der Argumentübergabe könnten dann also nur konstante Daten an das Unterprogramm übergeben werden, da vom Prinzip her in einem ROM keine variablen Daten gespeichert werden können.
Bei Mikrorechnern müssen die Speicherbereiche für das Programm und für Konstanten vom Bereich für variable Daten getrennt werden, d.h. eine Unterteilung in einen ROM- und einen RAM-Bereich. Um diese Trennung der Speicherbereiche beim Transfer von variablen Daten

zwischen Haupt- und Unterprogramm beizubehalten, verwendet man häufig als Argumente des Unterprogramms nicht die Daten selbst, sondern deren Adressen. Im Falle des Multiplikations-Unterprogramms (MULTA = Multiplikation von Adreßargumenten), könnte also z.B. folgende Programmgliederung verwendet werden

Rom-Bereich:

```
Hauptprogramm      .
                   .
                   CALL + MULTA
                   ADR F 1      ; Faktor 1 steht in Adresse ADR F 1
                   ADR F 2      ; Faktor 2 steht in Adresse ADR F 2
                   ADR P        ; das Produkt wird in die Adressen
                                  ADR P und ADR P + 1 abgespeichert
                   .
                   .
                   .
Unterprogramm      .  )
                   .  |
                   .  }          Multiplikationsprogramm
                   .  |
                   .  )
                   JUMP @ R3 ↑  ; Rücksprung
```

RAM-Bereich:

Daten	ADR F 1	Faktor 1
	ADR F 2	Faktor 2
	ADR P	PR
	ADR P + 1	PL

Im folgenden wollen wir das Multiplikationsbeispiel für die Übergabe von Adreßargumenten erweitern. Dabei läßt sich gleichzeitig die vorhergehende Methode demonstrieren. Sie ist eine Vorstufe zur Übergabe von Adreßargumenten.

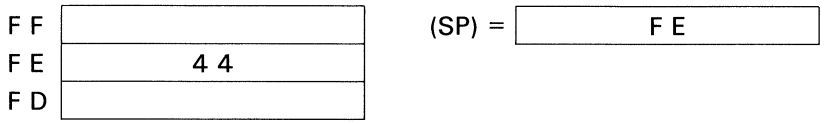
Anmerkung:
Bei dem MP-Lehrsystem verwenden wir natürlich das RAM für die Programmspeicherung.

Folgende Voraussetzungen sollen für dieses Beispiel gelten:
- Es sollen die Daten, die in den Adressen 5 0 und 5 1 stehen, miteinander multipliziert werden
- Das Ergebnis soll in den Adressen 5 2 und 5 3 abgespeichert werden
- Das Hauptpogramm soll mit der Adresse 4 0 beginnen
- Ausgangsstellung des Stack-Pointers ist Adresse F F

Bevor die einzelnen Programmbereiche und Programmschritte näher erläutert werden, geben wir zunächst das Gesamtprogramm an (Tab. 5.3.3). Sie können dieses Programm laden, damit Sie die nachfolgenden Erläuterungen besser nachvollziehen können.
Das eigentliche Hauptprogramm umfaßt den Adreßbereich von 4 0 bis 4 7. Hierzu ist zu bemerken, daß in den Adressen 4 4 und 4 5 die Adressen enthalten sind, unter denen die beiden Faktoren abgespeichert sind. In diesem Beispiel sind es die Adressen 5 0 und 5 1. In Adresse 4 6 ist die Adresse für das Produkt abgespeichert. Wie Sie noch sehen werden, wird die zweite Adresse für das Produkt, das ja 16 bit umfaßt bzw. umfassen kann, im Unterprogramm festgelegt.
Das Hauptprogramm beginnt mit der Initialisierung des Stack-Pointers, d.h., R3 wird mit F F geladen. Wird danach dann der Befehl CALL MULTA ausgeführt, wird die Rücksprungadresse 4 4 in die Adresse F F geladen (beachten Sie die Adressierungsart: Auto-Decrement-Indexed über R3!). Nachfolgend ist der Zustand des Stacks nach dem CALL-Befehl dargestellt:

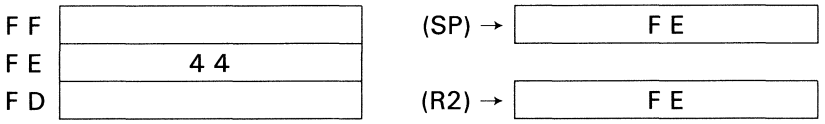
Adresse	Inhalt	Befehl	Kommentar
4 0	8 3	LOAD R3, # F F	Initialisiere Stack-Pointer
4 1	F F		
4 2	F 0	CALL MULTA	
4 3	2 6		
4 4	5 0	ADR F 1	Adresse von Faktor 1
4 5	5 1	ADR F 2	Adresse von Faktor 2
4 6	5 2	ADR P	Produktadresse
4 7	0 0	HALT	
2 6	0 E	MOVE R2, R3	Lade R2 mit Adr von R3
2 7	8 A	LOAD R2, @R2	Lade R2 mit Inh. der Adr von R2
2 8	8 A	LOAD R2, @R2	Lade R2 mit Inhalt der zweiten Adr von R3
2 9	8 8	LOAD R0, @ R2	Lade R0 mit Inh. der Adr aus R2
2 A	0 E	MOVE R2, R3	Lade R2 mit Adr von R3
2 B	8 A	LOAD R2, @R2	Lade R2 mit Inh. der Adresse von R2
2 C	6 2	INCR R2	Erhöhe Adr in R2 um 1
2 D	8 A	LOAD R2, @ R2	Lade R2 mit Inh. der um 1 erhöhten Adr.
2 E	8 9	LOAD R1, @ R2	Lade R1 mit Inh. der Adr aus R2
2 F	E 0	JUMP 0 1	Beginn Unterp.
3 0	0 1		
0 1		Multiplikations-programm im ROM-Bereich	Unterprogramm
0 2			
.			
.			
.			
.			
.			
1 7			
1 8			
1 9	0 E	MOVE R2, R3	Lade R2 mit Adr von R3
1 A	8 A	LOAD R2, @ R2	Lade R2 mit Inh. der Adr. von R2
1 B	9 2	ADDM R2, # 0 2	Addiere zum Inh. der Adr. in R2 +2
1 C	0 2		
1 D	8 A	LOAD R2, @ R2	Lade R2 mit dem Inh. der um 2 erhöhten Adr.
1 E	7 9	STAC R1, @ R2	Speichere R1 unter der Adr in R2
1 F	6 2	INCR R2	Erhöhe R2 um 1
2 0	7 8	STAC R0, @ R2	Speichere R0 unter der Adr. in R2
2 1	8 E	LOAD R2, @ R3 ↑	Lade R2 mit Inh von R3 und erhöhe R3 um 1
2 2	9 2	ADDM R2, # 0 3	Addiere zum Inh von R2 +2
2 3	0 3		
2 4	7 2	STAC R2, @↓ R3	Speichere neues Ziel von R2 auf vorherige Adr in R3 zurück
2 5	E C	JUMP @ R3 ↑	Springe zur Adresse in R3

Tab. 5.3.3
Gesamtprogramm für das Multiplikationsbeispiel

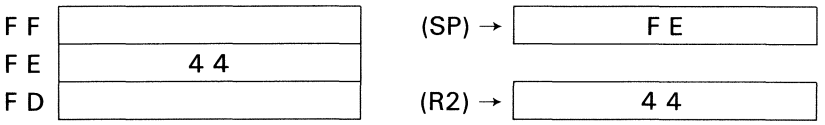


In diesem Falle ist aber 4 4 nicht die eigentliche Rücksprungadresse, sondern die Adresse des ersten Arguments F 1.
Das Unterprogramm beginnt bei der Adresse 2 6. Mit dem Befehl MOVE R2, R3 wird der Inhalt von R3, also F E in R2 geladen. Der nachfolgende Befehl LOAD R2, @ R2 bewirkt, daß R2 mit dem Inhalt der Adresse F E, also 4 4, geladen wird. Mit dem nachfolgenden Befehl

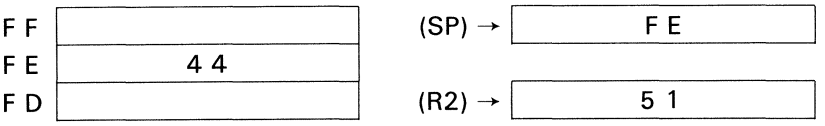
LOAD R2, @ R2 wird R2 mit dem Inhalt der Adresse 4 4, also mit 5 0, geladen. Dies ist **die Adresse** von Faktor F 1. Dieser Faktor soll in Register R0 geladen werden, was mit dem Befehl LOAD R0, @ R2 geschieht.
Mit den nachfolgenden Befehlen von Adresse 2 A bis zur Adresse 2 E wird nach dem gleichen Prinzip der Faktor F 2 in das Register R1 geladen. Hierbei muß allerdings der Inhalt von R2 incrementiert werden.
Mit dem Befehl JUMP 0 1 erfolgt dann ein Sprung zum Multiplikationsprogramm im ROM-Bereich des Rechners. Hier wird nun, wie bereits früher beschrieben, die eigentliche Multiplikation vorgenommen (bis Adresse 1 8).
Im nächsten Programmteil (ab Adresse 1 9) muß das Ergebnis, also der Inhalt von R0 und R1, abgespeichert werden. Zunächst muß hierzu das Indexregister R2 mit dem Inhalt des Stack-Pointers R3 geladen werden. Nach dem Befehl MOVE R2, R3 ergeben sich folgende Verhältnisse:



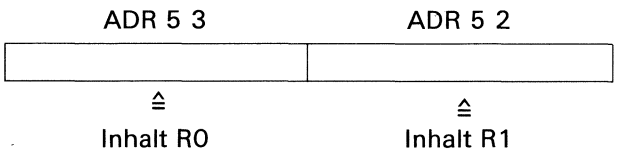
Danach erfolgt der Befehl LOAD R2, @ R2, so daß jetzt R2 mit dem Inhalt der Adresse F E geladen wird:



Der Befehl ADDM R2, # 0 2 erhöht den Inhalt von R2 um 0 2, so daß in R2 jetzt die Adresse 4 6 steht. Mit dem Befehl LOAD R2, @ R2 wird die Abspeicheradresse ADR P (5 1) in R2 geladen:

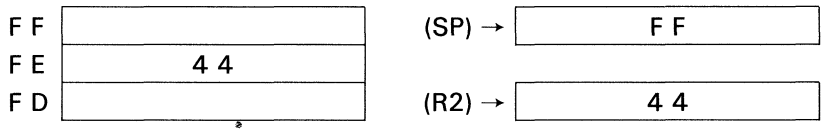


Danach wird mit dem Befehl STAC R1, @ R2 jetzt die erste Ergebnishälfte, die in R1 steht, in Adresse 5 2 abgespeichert. Dann wird R2 um 1 erhöht, so daß R2 jetzt die Abspeicheradresse ADR P + 1 enthält. Mit dem Befehl STAC R0, @ R2 wird die zweite Ergebnishälfte in Adresse 5 3 abgespeichert.
Bei der hier gewählten Registerordnung steht in Adresse 5 3 die linke, in Adresse 5 2 die rechte Ergebnishälfte:

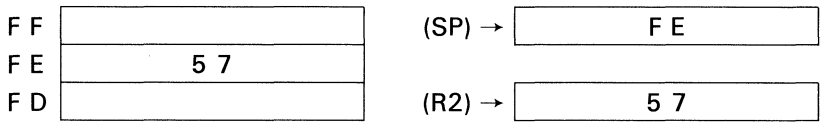


Wenn nun im Programmablauf direkt nach dem Abspeichern des Ergebnisses der Rücksprung ausgeführt würde, so würde ein Sprung zur Adresse 4 4 erfolgen, da oben im Stack immer noch die Adresse 4 4 steht. In diesem Falle würden dann die Argumentadressen als Befehl ausgeführt, so daß ein fehlerhafter Programmablauf zustande käme. Im Falle unseres Hauptprogramms ist die Rücksprungadresse 4 7, also der HALT-Befehl. Aus diesem Grunde muß eine Korrektur der Rücksprungadresse im Stack erfolgen, so daß die Argumente des Unterprogramms übersprungen werden. Mit dem Befehl LOAD R2, @R3 ↑ wird der Inhalt der Adresse,

auf die der Stack-Pointer zeigt (in unserem Falle F E), in R2 geladen. Damit ergeben sich folgende Verhältnisse:



Mit dem Befehl ADDM R2, # 0 3 wird jetzt zunächst in R2 die richtige Rücksprungadresse 4 7 hergestellt. Diese Rücksprungadresse wird anschließend mit dem Befehl STAC R2, @↓ R3 abgespeichert. Dieser Befehl erniedrigt zunächst R3 um 1 und speichert dann die korrigierte Rücksprungadresse im Stack ab. Vor dem JUMP-Befehl ergeben sich also nachfolgende Verhältnisse:



Jetzt kann mit dem Befehl JUMP @ R3 ↑ der Rücksprung zur richtigen Adresse des Hauptprogramms erfolgen. Gleichzeitig wird der Stack-Pointer wieder in seine Ausgangsadresse F F gebracht.
Dieses Beispiel hat deutlich gezeigt, daß durch eine solche Methode zur Argumentübergabe das Unterprogramm länger und auch komplizierter wird. Um wieviel länger, hängt von den Adressierungsarten ab, die zur Verfügung stehen. Dagegen wird das Hauptprogramm kürzer und übersichtlicher. Dies ist bei der Integration von vielen Programmoduln ein eintscheidender Vorteil. Man sollte deshalb, wenn man einen Mikroprozessor für eine bestimmte Aufgabe auswählt, immer auch mit berücksichtigen, wie leicht oder schwierig es ist, z.B. Adreßargumente an ein Unterprogramm zu übergeben.

Exp. 12

Fragen zu den Abschnitten 5.2 und 5.3

1. Welche Anforderungen ~~müssen~~ muß an die Rücksprungadresse am Ende eines Unterprogramms gestellt werden?

2. In den Adressen E 0 bis E C soll ein bestimmtes Unterprogramm enthalten sein. Im Hauptprogramm soll nach der Adresse 5 0 dieses Unterprogramm mit JUMP abgerufen werden. Geben Sie die dazu notwendigen Befehle im Hauptprogramm an, wenn der Rücksprungbefehl im Unterprogramm

JUMP @ A 7

lautet!

3. Geben Sie an, welche Vorgänge beim Anruf eines Unterprogramms mit dem CALL-Befehl ablaufen!

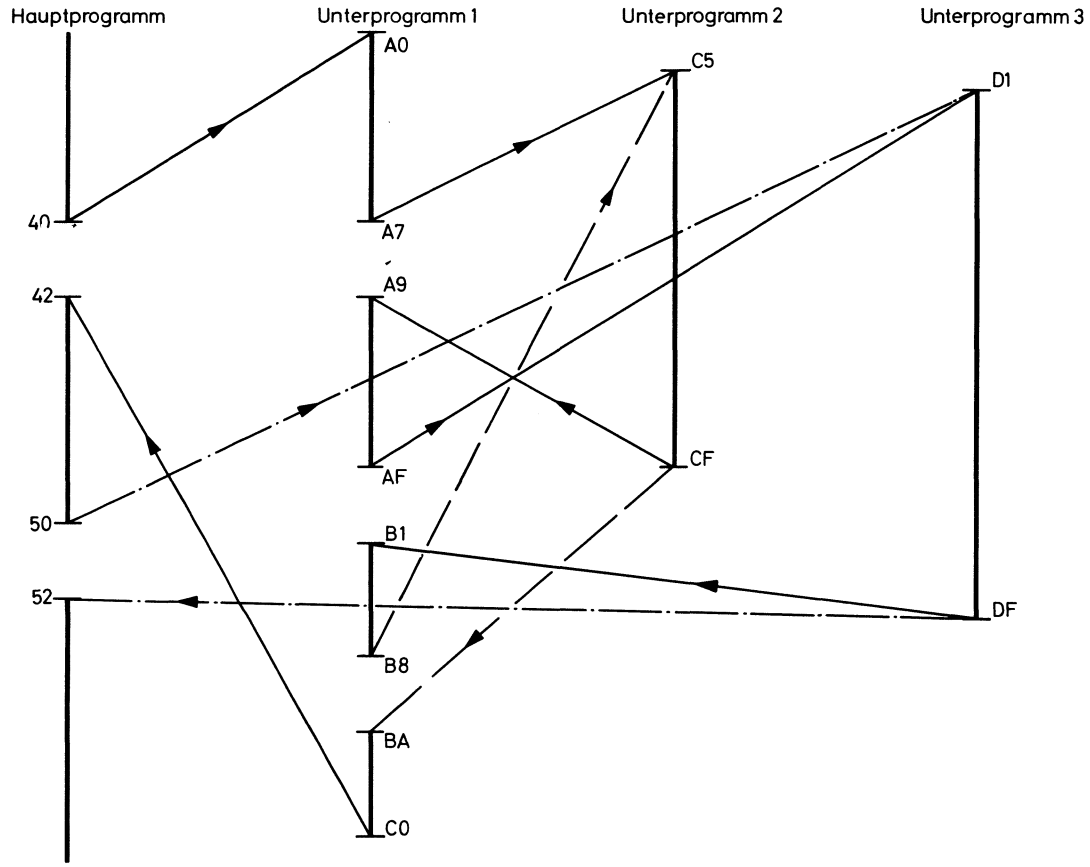
4. Wie lautet die Rücksprungadresse, wenn in einem Hauptprogramm bei Adresse 4 0 mit dem Befehl

CALL @ B 0

ein Unterprogramm angerufen wird?

5. Wie muß grundsätzlich der letzte Befehl in einem Unterprogramm lauten, wenn der Anruf mit CALL erfolgt?

6. Die Abbildung zeigt ein Programm mit verschachtelten Unterprogrammen.



a) Erläutern Sie den Programmablauf mit Hilfe der angegebenen Adressen!
b) Wie viele Adressen sind für den Stack-Bereich unbedingt zu reservieren?

7. Durch welche einfache Programmänderung kann das Multiplikationsprogramm MULTA in Abschnitt 5.3 mit Übergabe von Adreßargumenten zum Quadrierprogramm umgewandelt werden?

Anhang

Antworten auf die Fragen zu den Abschnitten 5. und 5.1

1. Ein Flußdiagramm ist die grafische Darstellung einer Problemlösung, die die Programmerstellung erleichtert.

2. In Abhängigkeit von einer bestimmten Bedingung kann der Programmablauf über 2 verschiedene Programmpfade erfolgen.

3. Nein, da dann ein Verlassen der Schleife nicht mehr möglich ist.

4. Durch ADDM R2, # 5 0 wird R2 auf 5 5 gesetzt. Der Befehl JUMP @ R2 besagt, daß das Sprungziel der Inhalt der Adresse 5 5, also F 3, ist.

5. a) Mit 16 bit kann als größte Zahl $2^{16} - 1 = 65\,535$ dargestellt werden (Anzeige F F, F F).
b) Bei der Eingabe F F · F F muß als Ergebnis

$$(2^8 - 1) \cdot (2^8 - 1) = 255 \cdot 255 = 65\,025$$

dargestellt werden. Damit ist die Anzeigekapazität nicht voll ausgeschöpft.

Antworten auf die Fragen zu den Abschnitten 5.2 und 5.3

1. Die Rücksprungadresse muß variabel sein, da ein Unterprogramm mehrmals vom Hauptprogramm angerufen werden kann und dabei in der Regel unterschiedliche Rücksprungadressen erforderlich sind.

2.

Adresse	Inhalt	Befehl
5 0	*x x*	*x x x x*
5 1	8 0	LOAD RO, # 5 7
5 2	5 7	
5 3	7 4	STAC RO, A 7
5 4	A 7	
5 5	E 0	JUMP E 0
5 6	E 0	
5 7	*x x*	*x x x x*

3. Zunächst wird die Rücksprungadresse abgespeichert. Anschließend erfolgt der Sprung ins Unterprogramm und dessen Abarbeitung. Mit der zuvor gespeicherten Adresse erfolgt dann der Rücksprung ins Hauptprogramm.

4. Der Befehl CALL @ B 0 ist ein 2-Byte-Befehl. Damit ergibt sich als Rücksprungadresse 4 2.

5. Beim hypothetischen Rechner wird grundsätzlich R3 als Stack-Pointer für die Rücksprungadressen verwendet, also kann der Rücksprung nun mit JUMP @ R3 ↑ erfolgen.

6. a) Bei Adresse 4 0 im Hauptprogramm wird Unterprogramm 1 angerufen. Bei Adresse A 7 wird im Unterprogramm 1 das Unterprogramm 2 angerufen und abgearbeitet (Rücksprung nach A 9). Bei A F wird dann Unterprogramm 3 angerufen und abgearbeitet (Rücksprung nach B 1), bei Adresse B 8 Unterprogramm 2 nochmals abgearbeitet (Rücksprung nach B A). Bei C0 erfolgt der Rücksprung ins Hauptprogramm nach Adresse 4 2. Bei Adresse 5 0 wird dann vom Hauptprogramm das Unterprogramm 3 nochmals direkt angerufen und abgearbeitet (Rücksprung nach 5 2). Jetzt kann das Hauptprogramm weiter abgearbeitet werden.

b) In diesem Beispiel werden nur 2 Stack-Adressen benötigt. Der Ablauf ist nachfolgend tabellarisch dargestellt. Dabei wird der Anfang des Stack-Bereiches durch das Laden der Adresse 3 0 in R3 festgelegt:

Sprung	SP	Inhalt SP
	3 0	*x x*
4 0→A 0	2 F	4 2
A 7→C 5	2 E	A 9
A 9←C F	2 F	4 2
A F→D 1	2 E	B 1
B 1←D F	2 F	4 2
B 8→C 5	2 E	B A
B A←C F	2 F	4 2
4 2←C 0	3 0	*x x*
5 0→D 1	2 F	5 2
5 2←D F	3 0	*x x*

Die Tabelle zeigt, daß als Stack-Adressen nur 2 F und 2 E benötigt werden. Adresse 3 0 kann mit beliebigen Inhalten belegt sein, da für den eigentlichen Stack-Betrieb diese Adresse nicht benötigt wird.

7. Die Argumentadressen in den Speicherplätzen 4 4 und 4 5 müssen gleich gewählt werden (z.B. 5 0 in beiden Speicherplätzen). Dadurch wird der Inhalt von Adresse 5 0 als Faktor F 1 **und** F 2 verwendet.