Mikroprozessoren und Mikrorechner

Lehrheft 2

ITT Fachlehrgänge, Pforzheim

ITT

2. Auflage, April 1977

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

4. Hypothetischer Mikrorechner

Wie bereits in der Einleitung zu Abschnitt 3. erwähnt, bildet der Mikroprozessor die Zentraleinheit (CPU) eines Mikrorechners bzw. eines Mikrorechnersystems. Wie der Name Zentraleinheit schon aussagt, bestimmt die CPU die Anwendungs- und Aufbaumöglichkeiten und außerdem letztlich die Arbeitsweise eines Mikrorechners. Um die Funktion eines Mikrorechners verstehen zu können, muß die Funktion der CPU, also des Mikroprozessors, bekannt sein.
Handelsübliche Mikroprozessoren haben viele Eigenschaften, deren Notwendigkeit nicht auf den ersten Blick einzusehen ist. Echte Mikroprozessoren sind meist auch nicht sehr logisch strukturiert und deshalb nicht gut überschaubar. Dieses hängt damit zusammen, daß die Hersteller Eigenschaften wie Leistungsfähigkeit, Arbeitsgeschwindigkeit u.a.m. der Überschaubarkeit vorziehen. Um die Funktionen eines echten Mikroprozessors anschaulicher zu erklären, gehen wir den Weg über einen **hypothetischen Mikroprozessor** und somit über einen **hypothetischen Mikrorechner.** Aufgrund seiner Struktur ist dieser Mikrorechner sehr gut zu verstehen und läßt sich trotzdem zur Erklärung komplizierter Rechenbegriffe anwenden. Außerdem sollen mit ihm die Grundbegriffe des Programmierens vermittelt werden.
Bisher wurde bei allen Abhandlungen eine Beschreibung der Hardware gebracht. Für den Anwender von Mikroprozessoren wäre es aber zu aufwendig, wenn er alle Einzelheiten der integrierten Schaltkreise kennenlernen wollte. Wichtig ist, daß er das Funktionsprinzip so weit versteht, daß er den Mikroprozessor einsetzen kann. Nach den Erklärungen in den vorangegangenen Abschnitten ist Ihnen das grundsätzliche Funktionsprinzip eines Mikroprozessors bereits bekannt.
Nun geht es in erster Linie darum, Sie mit den Systemeigenschaften und Softwarenproblemen vertraut zu machen.

Anmerkung:
Unter **Hardware** versteht man die gerätetechnische Ausstattung eines Rechnersystems.

Als **Software** werden alle zum Betrieb des Systems benötigten Programme bezeichnet.

Der hier besprochene hypothetische Rechner ist ein 8-bit-Rechner mit 4 Akkumulatoren oder Allzweckregistern R0 bis R3.

Anmerkung:
In der Rechnertechnik bezeichnet man häufig Akkumulatoren auch als Register, besonders dann, wenn die ALU-Eigenschaften des Akkumulators nicht ausgenutzt werden. Nachfolgend werden beide Begriffe wechselweise benutzt.

Der Rechner enthält außerdem einen Befehlszähler PC (**P**rogram **C**ounter), der ebenfalls 8 bit hat und somit $2^8 = 256$ Speicherwörter adressieren kann. Als Flags sind vorhanden:
– ein Übertrags- oder Carry-Flag (C-Flag)
– ein Zero-Flag (Z-Flag)
– ein Negativ-Flag (N-Flag)
Zwei der Akkumulatoren können auch für die Adressierung herangezogen werden. Auf diesen Punkt kommen wir später noch zu sprechen. In Bild 4.1 sind die Register und Flags dargestellt.

Arithmetische und logische Verknüpfungen können entweder zwischen dem Inhalt eines Akkumulators und dem eines Speicherplatzes ausgeführt werden, oder aber zwischen den Inhalten zweier Akkumulatoren. Grundsätzlich werden dabei die verschiedenen Operationsabläufe des Mikrorechners durch die ihm eigene Sprache, den **Maschinencode,** eingeleitet und ausgeführt. Der Maschinencode ist durch die Schaltungstechnik des Systems festgelegt. Um einem Rechner sinnvolle Befehle geben zu können, müssen wir also seine Sprache anwenden. Es gibt verschiedene Befehlsgruppen, die nachfolgend erläutert werden. Wesentlich ist zu wissen, daß der Rechner bei einem bestimmten Befehl eine bestimmte Operation ausführt. Beim automatischen Betrieb sind diese Befehle oder eine bestimmte Befehlsfolge im Speicher gespeichert. Anders ausgedrückt, bevor der Rechner arbeiten kann, müssen die **Befehle** bzw. das **Programm** in den Speicher gebracht werden. Dies geschieht über

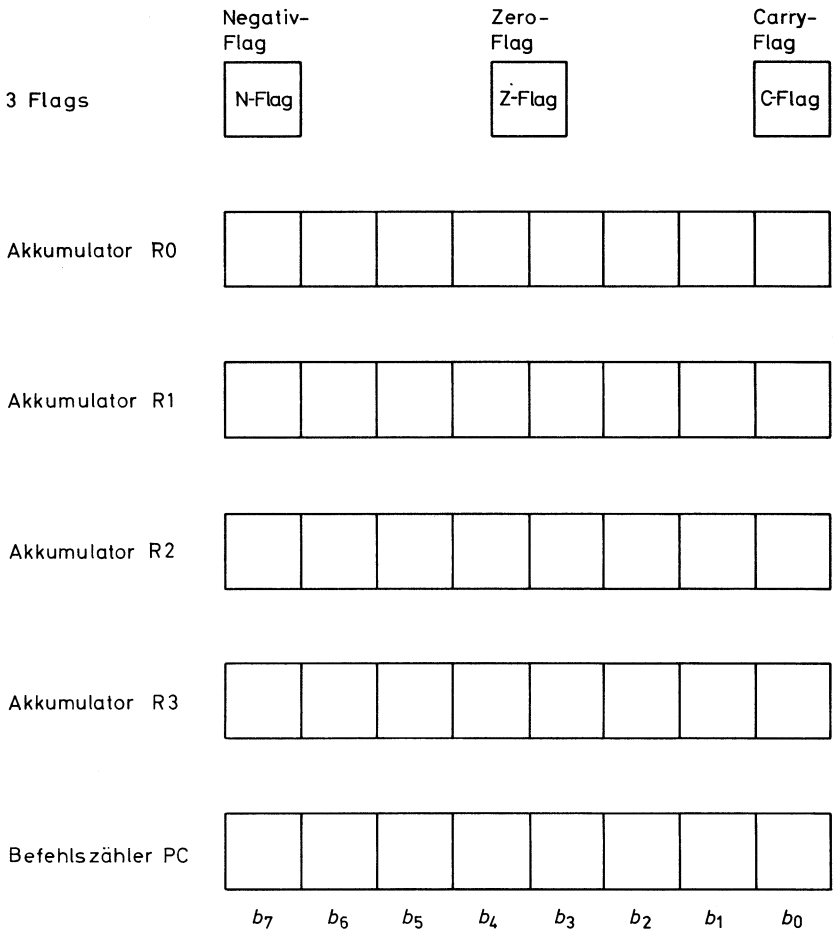
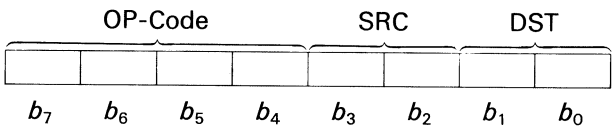


Bild 4.1
Register, Flags und Befehlszähler des hypothetischen Rechners

bestimmte Adressen, über die dann wieder das entsprechende Befehlswort im Programm ausgewählt werden kann. Zuerst wollen wir uns mit den Befehlen befassen, die Operationen zwischen den 4 Akkumulatoren ermöglichen. Diese Befehle werden als **1. Befehlsgruppe** bezeichnet.

4.1 Erste Befehlsgruppe

In dieser Gruppe sind 6 Befehle zusammengefaßt, die folgende Form haben:



Dieser 8-bit-Befehl läßt sich wie folgt aufschlüsseln:
Die bit b_7 bis b_4 stellen den OP-Code (Operationscode) dar, d.h., hier wird festgelegt, welche Operation (Addition, Subtraktion usw.) durchgeführt werden soll.
Mit den bit b_3, b_2 kann einer der 4 Akkumulatoren ausgewählt werden. Dieser Akkumulator wird als SRC-Register (Source-Register = Quellenregister) bezeichnet, weil er in diesem Falle als Datenquelle benutzt wird.
Mit den bit b_1, b_0 kann ein zweiter Akkumulator, das sog. DST-Register (Destination-Register = Zielregister) angesprochen werden. Tab. 4.1.1 zeigt die Zuordnung für die anzusprechenden Register.
Mit dem OP-Code können die in Tab. 4.1.2 aufgeführten Operationen eingeleitet werden.

SRC	b_3	b_2	
	0	0	→ Register R0
	0	1	→ Register R1
	1	0	→ Register R2
	1	1	→ Register R3

DST	b_1	b_0	
	0	0	→ Register R0
	0	1	→ Register R1
	1	0	→ Register R2
	1	1	→ Register R3

Tab. 4.1.1
Zuordnung für die anzusprechenden Register

OP-Code	Befehl	Funktion
0 0 0 0	MOVE	(SRC) → (DST) [Ausnahme HALT]
0 0 0 1	ADDR	(SRC) + (DST) → DST
0 0 1 0	SUBR	(DST) − (SRC) → DST
0 0 1 1	IORR	(SRC) V (DST) → DST
0 1 0 0	XORR	(SRC) ∀ (DST) → DST
0 1 0 1	ANDR	(SRC) Λ (DST) → DST

Tab. 4.1.2
Befehlsliste der 1.Befehlsgruppe

Anmerkung:
Der Klammerausdruck (SRC) bedeutet Inhalt des SRC-Registers. Steht um **eine Zahl** eine Klammer, so bedeutet dies, daß diese Zahl als Adresse zu verstehen ist und somit der Inhalt dieser Adresse für den Operationsablauf zu benutzen ist.

Der OP-Code 0 0 0 0 ist der sog. MOVE-Befehl (MOVE = übertragen), der es ermöglicht, Daten von irgendeinem Akku zu transferieren. Soll z.B. der Inhalt von R1 in R3 kopiert werden, ist der Befehl

0 0 0 0	0 1	1 1
OP-Code	SRC	DST
≙	≙	≙
MOVE	R_1	R_3

erforderlich. An diesem Beispiel können auch deutlich die Bezeichnungen SRC und DST erklärt werden. Die Daten im Quellenregister R1 werden in das Zielregister R3 kopiert, ohne daß der Inhalt von R1 dabei verloren geht.
In der symbolischen Kurzschreibweise wird dieser Befehl

MOVE R3, R1

geschrieben. Die Bewegung der Daten ist bei dieser Scheibweise von rechts nach links zu verstehen. Diese Reihenfolge wurde gewählt, um kompatibel mit der Intel-8080-Assemblersprache zu sein. Unter Assemblersprache versteht man eine maschinenspezifische Programmsprache. Der Befehl

MOVE R3, R3

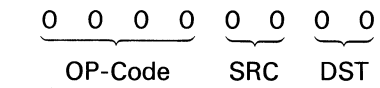
lautet in Maschinensprache

0 0 0 0	1 1	1 1
OP-Code	SRC	DST

d.h., er überträgt Daten von R3 in R3. Dies bedeutet, daß nach außen hin keine Operation erfolgt. Damit kann dieser Befehl als NOP (No Operation = keine Funktion) benutzt werden. Nach diesem Schema sind insgesamt 4 solcher NOP-Befehle möglich, von denen allerdings 3 überflüssig sind. In vielen echten Mikrorechnern werden solche redundanten Befehle (Redundanz = Weitschweifigkeit) für andere Zwecke benutzt. Um den hypothetischen Mikrorechner möglichst übersichtlich zu halten, haben wir als einzige Ausnahme den MOVE-Befehl

MOVE R0, R0

mit dem Maschinencode



als HALT-Befehl benutzt. Auch hierbei gilt, daß die Daten im SRC-Register nicht verlorengehen.

Alle Befehle dieser 1. Gruppe funktionieren ähnlich. Der ADDR-Befehl (Add-Register) mit dem OP-Code 0 0 0 1 addiert den Inhalt des SRC-Akkumulators in den DST-Akkumulator hinein. Dies geht auch aus der Funktionsschreibweise (SRC) + (DST) → DST hervor. Der Pfeil deutet an, daß die Summe im DST-Akkumulator gebildet wird. So bewirkt z.B. der Befehl ADDR R2, R0 (Maschinencode: 0 0 0 1 0 0 1 0), daß der Inhalt von R0 in R2 „hinein" addiert wird. Auch hierbei geht der SRC-Inhalt nicht verloren.
Der Befehl SUBR (Subtract-Register) mit dem OP-Code 0 0 1 0 subtrahiert den SRC-Inhalt vom DST-Inhalt.
Hierzu ist zu bemerken, daß bei diesem Befehl der SRC-Inhalt vom DST-Inhalt abgezogen wird und nicht umgekehrt. Obwohl dies in der Funktionstabelle unsystematisch erscheint, bietet diese Reihenfolge praktische Vorteile und wird deshalb bei den meisten Mikrorechnern angewandt.
Der Befehl IORR (INCLUSIVE-OR-Register) mit dem OP-Code 0 0 1 1 bildet die ODER-Verknüpfung zwischen SRC- und DST-Inhalt. Das Ergebnis erscheint auch wieder im DST-Register.
Die Befehle XORR (EXCLUSIV-OR-Register) und ANDR (AND-Register) bilden in gleicher Weise die EXCLUSIV-ODER bzw. die UND-Verknüpfung.
In einem Programmbeispiel sollen die Inhalte R0 mit R2 in R2 addiert und vom Ergebnis der Inhalt R3 subtrahiert werden. Mit diesem Zwischenergebnis wird dann die ODER-Funktion mit dem Inhalt von R1 gebildet. Das Ergebnis wird im Akkumulator R1 abgespeichert. Das Programm für diese verbale Aufgabenstellung zeigt Tab. 4.1.3.

Exp. 6a

Tab. 4.1.3
Programmbeispiel

Maschinencode			Befehl
OP-Code	SRC	DST	
0 0 0 1	0 0	1 0	ADDR R2, R0
0 0 1 0	1 1	1 0	SUBR R2, R3
0 0 1 1	1 0	0 1	IORR R1, R2
0 0 0 0	0 0	0 0	HALT

Fragen zu Abschnitt 4.1

1. Welche Befehle gehören beim hypothetischen Mikrorechner zur 1. Befehlsgruppe?

2. Erklären Sie die Funktionsschreibweise:

(SRC) V (DST) → DST

3. Geben Sie die Funktionsschreibweise für eine UND-Verknüpfung mit R0 als SRC-Register und R1 als DST-Register an!

4. Welches Register ist beim Befehl

MOVE R3, R2

SRC-Register, und welches Register ist DST-Register?

5. Welche Funktion wird nach dem Befehl

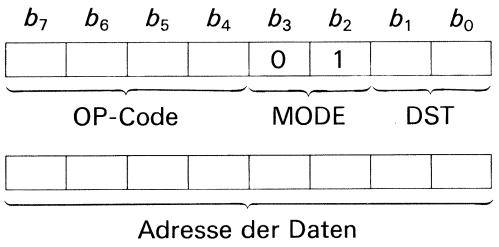
0 0 0 0 0 1 1 1 (Maschinencode)

durchgeführt?

6. Was versteht man unter einem NOP-Befehl?

4.2 Zweite Befehlsgruppe

In dieser Gruppe gibt es auch 6 Befehle, die die gleichen arithmetischen und logischen Funktionen ausführen wie die in der 1. Gruppe. Der einzige Unterschied besteht in der Datenquelle. Anstatt über 2 SRC-bit die Daten aus einem der 4 Akkumulatoren zu holen, werden die Daten jetzt aus dem Speicher bezogen. Dabei werden natürlich jetzt die 2 SRC-bit für die Spezifizierung eines SRC-Registers nicht mehr benötigt. Diese 2 bit werden stattdessen dafür benutzt, die sog. Adressierungsart (Adress-Mode) festzulegen. Um Daten vom Speicher zu holen, muß über eine Adresse die entsprechende Speicherstelle angewählt werden. Diese Adresse könnte wie beim vereinfachten Rechner als Teil des Befehles angegeben werden. Eine solche Adressierungsart wird als **direkte** oder auch **absolute Adressierung** bezeichnet.
Bei Mikroprozessoren ist normalerweise das Befehlswort nicht lang genug, um den OP-Code und die Adresse unterzubringen. Dies wird dann dadurch umgangen, daß mehrere Wörter für einen Befehl benutzt werden. Auch der hypothetische Mikrorechner bietet die Möglichkeit einer direkten Adressierung. Diese wird durch die Adressierungsart-bit 0 1 gekennzeichnet, die anstelle der SRC-bit gesetzt werden. Ein solcher Befehl hat dann folgende Form:



Hieraus ist ersichtlich, daß die Adresse für die Daten in dem **nächsten** 8-bit-Wort enthalten ist. Eingangs wurde bereits erwähnt, daß der Befehlszähler PC 8 bit, d.h. $2^8 = 256$ Speicherwörter, adressieren kann. Aus diesem Grunde sind auch für eine Adresse 8 bit erforderlich.

In Tab. 4.2.1 sind die 6 Befehle der 2. Gruppe aufgeschlüsselt.

OP-Code	Befehl	Funktion
1 0 0 0	LOAD	(EA → DST
1 0 0 1	ADDM	(EA) + (DST) → DST
1 0 1 0	SUBM	(DST) − (EA) → DST
1 0 1 1	IORM	(EA) V (DST) → DST
1 1 0 0	XORM	(EA) ∀ (DST) → DST
1 1 0 1	ANDM	(EA) Λ (DST) → DST

Tab. 4.2.1
Befehlsliste der 2. Befehlsgruppe (EA = effektive Adresse)

Hierzu ein einfaches Beispiel:
Es soll der Inhalt der Adresse 0 1 0 1 1 1 1 1 in den Akkumulator R2 addiert werden. Hierzu wird der ADDM-Befehl (ADD-Memory = Addiere Speicher) mit dem OP-Code 1 0 0 1 benutzt. Bei direkter Adressierung lautet dann der Befehl im Maschinencode:

OP-Code				MODE		DST	
1	0	0	1	0	1	1	0
0	1	0	1	1	1	1	1
Adresse							

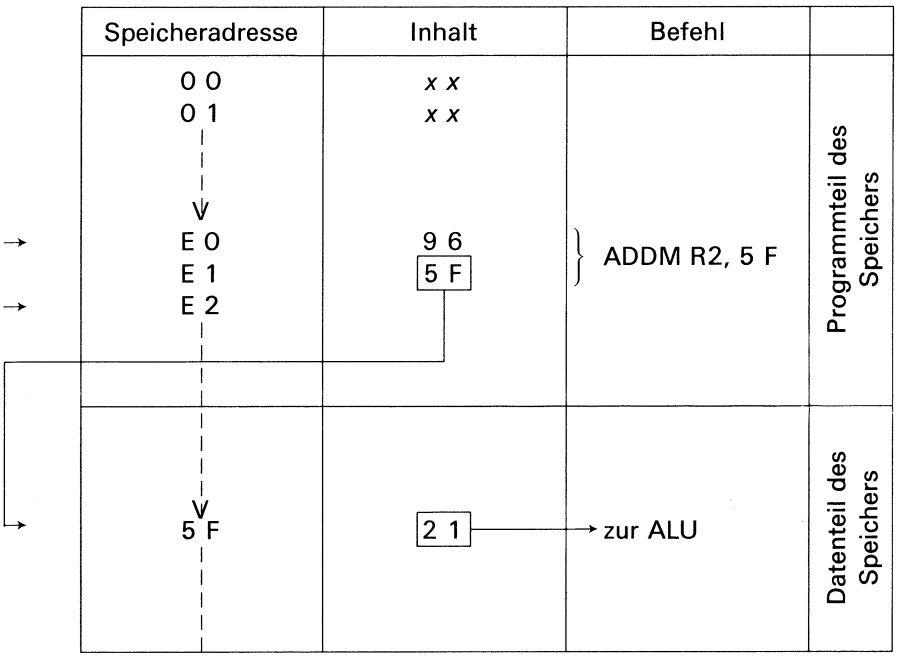
Beachten Sie bitte bei diesem Beispiel, daß das angegebene Adreßwort nur **eine bestimmte Speicherstelle** auswählt. Es darf daher **nicht mit dem Inhalt** dieser Speicherstelle verwechselt werden. Außerdem darf der Mikrorechner bei dieser Befehlsart das Adreßwort nicht als zweiten Befehl interpretieren. Daher muß der Programmzähler automatisch um **2 Stellen** erhöht werden, damit dieses Wort übersprungen wird. Befehle dieser Art werden auch als 2-Wort-Befehle oder 2-Byte-Befehle bezeichnet. Mit 1 Byte wird dabei ein 8-bit-Wort bezeichnet. In Tab. 4.2.2 wird der Ablauf des beschriebenen Additionsbeispieles noch einmal verdeutlicht.

	Speicheradresse	Inhalt	Befehl	
	0 0 0 0 0 0 0 0	x x x x x x x x		Programmteil des Speichers
	0 0 0 0 0 0 0 1	x x x x x x x x		
→	1 1 1 0 0 0 0 0	1 0 0 1 0 1 1 0	ADDM R2,	
	1 1 1 0 0 0 0 1	0 1 0 1 1 1 1 1	0 1 0 1 1 1 1 1	
→	1 1 1 0 0 0 1 0			
→	0 1 0 1 1 1 1 1	0 0 1 0 0 0 0 1	→ zur ALU	Datenteil des Speichers

Tab. 4.2.2
Operationsablauf eines Additionsbefehles nach der 2. Gruppe bei direkter Adressierung

Bei dieser Darstellung wird davon ausgegangen, daß der Befehlszähler PC innerhalb eines Programms die Adresse 1 1 1 0 0 0 0 0 erreicht. Im Programmspeicher steht unter dieser Adresse das Wort 1 0 0 1 0 1 1 0, aus dem OP-Code, MODE und DST hervorgehen. Mit MODE 0 1 wird die direkte Adressierung spezifiziert, so daß unter der Programmadresse 1 1 1 0 0 0 0 1, also der **folgenden** Adresse im Programmteil des Speichers, der Speicherinhalt 0 1 0 1 1 1 1 1 gefunden wird. Der Steuerteil des Rechners wählt jetzt die Speicheradresse 0 1 0 1 1 1 1 1 an, und der Speicherinhalt 0 0 1 0 0 0 0 1 wird im Register R2 zu dessen Inhalt addiert. Nachdem dies geschehen ist, wird der nächste Befehl des Programms, der z.B. unter der Programmadresse 1 1 1 0 0 0 1 0 bei der Programmierung des Systems eingegeben wurde, abgearbeitet. Wesentlich ist die Erkenntnis, daß beim Abarbeiten eines Programms die 2 Byte-Befehle aus 2 aufeinanderfolgenden 8-bit-Wörtern bestehen, wobei das zweite Wort die Speicheradresse angibt, **deren Inhalt** mit dem Inhalt eines der 4 Register arithmetisch oder logisch verknüpft werden soll.
Tab. 4.2.2 zeigt weiterhin, daß bei binärer Darstellung der Wörter der Operationsablauf leicht unübersichtlich wird. Aus diesem Grunde arbeiten wir nachfolgend mit der hexadezimalen

Darstellung. Nur wenn der Maschinencode bit-weise dargestellt werden soll, wird noch die binäre Schreibweise verwendet. In Tab. 4.2.3 haben wir zu Übung denselben Ablauf in hexadezimaler Schreibweise dargestellt.

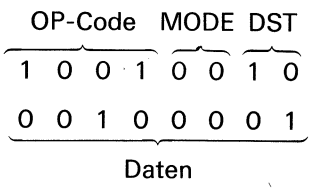


Tab. 4.2.3
Operationsablauf nach Tab. 4.2.2 in hexadezimaler Schreibweise

Die direkte Adressierung wird meistens für variable Daten benutzt. In dem verwendeten Beispiel könnten das Programm im ROM und die variablen Daten im RAM gespeichert sein. In diesem Falle muß natürlich eine klare Trennung zwischen Daten- und Programmteil des Speichers gegeben sein.
Unter dem OP-Code 1 0 0 0 finden Sie in der Befehlsliste dieser Befehlsgruppe (Tab. 4.2.1) den LOAD-Befehl mit der Funktion (EA) → DST. Dieser Befehl stellt eine Art MOVE-Befehl dar. Die Daten kommen hierbei nicht aus einem Akkumulator sondern aus dem Speicher. EA bedeutet bei der Funktionsschreibweise **effektive Adresse.**
Bei direkter Adressierung ist das zweite Byte die effektive Adresse. Bei komplizierten Adressierungsarten kann es sein, daß die eigentliche Adresse oder effektive Adresse, unter der im Speicher Daten zu finden sind, aus verschiedenen Informationen zuerst ermittelt werden muß.
Bei dem LOAD-Befehl wird der Inhalt der effektiven Adresse in das DST-Register geladen. Bei der hier besprochenen direkten Adressierungsart wird der Inhalt der Adresse, die im zweiten Byte steht, in das DST-Register geladen.
Alle anderen Befehle dieser Gruppe sind identisch mit denen der ersten Gruppe. Daß hier die Daten aus dem Speicher geholt werden, geht aus dem M statt dem R in der Kurzschreibweise der Befehle hervor, also ADDM anstatt ADDR.
Im Mikrorechnersystem muß oft mit Konstanten gearbeitet werden. Hier ist die sog. **Immediate-Adressierung** (immediate = unmittelbar) besonders geeignet. Im Gegensatz zur direkten Adressierung steht im zweiten Byte des Befehles keine Adresse, sondern hier sind die Daten der Konstanten selbst angegeben. Abgewandelt auf Immediate-Adressierung lautet der Additionsbefehl nach Tab. 4.2.3 anstatt ADDM R2, 5 F jetzt

ADDM R2, # 2 1

Das Zeichen # gibt an, daß es sich um diese Adressierungsart handelt und die Zahl 2 1 (2 1 ist ab jetzt hexadezimal zu verwerten!) gibt unmittelbar die Daten an, die zum Inhalt von R2 addiert werden sollen.
In Maschinensprache hat der Befehl ADDM R2, # 2 1 die Form:



Hieraus ist zu erkennen, daß in der Maschinensprache Immediate-Addressierung durch MODE 0 0 spezifiziert ist. In Tab. 4.2.4 ist der Operationsablauf des gewählten Befehles bei Immediate-Adressierung dargestellt.

Speicheradresse	Inhalt	Befehl	
0 0	x x		Programmteil des Speichers
0 1	x x		
⋮			
E 0	9 2	ADDM R2, # 2 1	
E 1	2 1 →	zur ALU	
E 2			

Tab. 4.2.4
Beispiel einer Immediate-Adressierung

Dieses Bild verdeutlicht, daß auch die Daten zum Programmbereich des Speichers gehören. Das ist auch der Grund dafür, daß sich diese Adressierungsart besonders für die Verarbeitung von Konstanten eignet. Zu bemerken ist noch, daß hierbei die effektive Adresse EA gleich der nächsten Adresse im Programmablauf entspricht (im Beispiel E 1).
Zur Übung soll jetzt wieder ein kleines Programm für folgende Aufgabenstellung erstellt werden: Der Inhalt der Adresse 9 3 mit 2 5 ist in den Akkumulator R2 zu laden. Davon ist dann die Konstante 7 5 abzuziehen. Das Programm soll bei der Adresse 4 2 anfangen. Tab. 4.2.5 zeigt das hierfür notwendige Programm.

Adresse	Inhalt	Befehl	Kommentar	
4 2	8 6	LOAD R2, 9 3	direkte Adressierung	Programmteil des Speichers
4 3	9 3			
4 4	A 2	SUBM R2, # 7 5	Immediate-Adressierung	
4 5	7 5			
4 6	0 0	HALT		
⋮	⋮			Datenteil des Speichers
9 3	2 5			

Tab. 4.2.5
Programmbeispiel

Beachten Sie bitte, daß auch hier alle Zahlen im Hexadezimalcode angegeben sind. In der Praxis ist es natürlich noch sehr aufwendig, im Hexadezimalsystem zu programmieren. Aus diesem Grunde wird eine sog. Assemblersprache für Mikroprozessoren verwendet. Ausführlich werden wir hierauf später noch eingehen. Einige Begriffe möchten wir jedoch jetzt schon einführen, um die Beispiele zu erleichtern.
Das Programm in Tab. 4.2.5 ist in einer Assemblersprache geschrieben. Die beiden linken Spalten, die Adresse und Inhalt in Hexadezimal enthalten, werden nicht vom Programmierer

sondern vom Rechner selbst hergestellt. Als Eingabe bekommt der Rechner die sog. **mnemonisch** abgekürzten Befehle. Unter mnemonischer Schreibweise versteht man die abgekürzte Schreibweise, die sich bei etwas Übung noch relativ leicht interpretieren läßt. Rechts neben dem Befehl darf noch ein Kommentar angegeben werden, um das Programm leichter verständlich zu machen. Dieser wird vom Befehl durch ein Sonderzeichen, z.B. einen Strichpunkt, abgetrennt. Immediate-Adressierung wird, wie bereits erwähnt, durch das Zeichen # gekennzeichnet. Da normalerweise meistens direkte Adressierung benützt wird, ist immer dann diese Adressierung angesprochen, wenn keine anderen Angaben gemacht werden.

Exp. 6b

Fragen zu Abschnitt 4.2

1. Worin besteht der grundsätzliche Unterschied zwischen Befehlen der 1. und 2. Befehlsgruppe?

2. Was versteht man unter einer direkten oder absoluten Adressierung?

3. Geben Sie für den Befehl

ADDM R0, 9 0

den kompletten Maschinencode an!

4. Welcher Unterschied besteht zwischen direkter Adressierung und Immediate-Adressierung?

5. Was bedeutet effektive Adresse EA bei direkter Adressierung?

6. Erstellen Sie für folgende Aufgabe ein Programm:

$$(x + y - z) \text{ V } a$$

Beginnen Sie das Programm bei der Adresse $4\ 0_{16}$.

4.3 Flags

Der hypothetische Mikroprozessor hat 3 Flags, nämlich das Carry-Flag (Übertrag), das Zero-Flag (Null) und das Negativ-Flag. Die 4 Befehle NOP, HALT, MOVE und LOAD beeinflussen die Flags nicht, d.h., ihr Zustand wird durch diese 4 Befehle nicht geändert. Die anderen Befehle der ersten und der zweiten Gruppe beeinflussen alle 3 Flags. Die arithmetischen Befehle ADDR, SUBR, ADDM und SUBM setzen bzw. löschen die Flags dem Ergebnis entsprechend. Die logischen Befehle IORR, XORR, ANDR, IORM, XORM und ANDM löschen das Carry-Flag und setzen bzw. löschen die anderen 2 Flags dem Ergebnis entsprechend.

4.4 Sprungbefehle

Befehle werden normalerweise von fortlaufenden Adressen abgeholt, da der Befehlszähler nach jedem Befehl um 1 bzw. 2 Stellen erhöht wird. Mit den Sprungbefehlen kann dieser Ablauf geändert werden. Bei einem Sprungbefehl kann der Programmzähler mit einer neuen Adresse geladen werden, d.h., er überspringt eine oder mehrere Adressen vorwärts oder rückwärts. Sprungbefehle können entweder **unbedingt** oder **bedingt** sein:

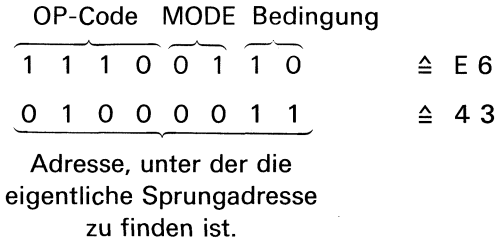
- Unbedingte Sprungbefehle bewirken grundsätzlich einen Sprung des Befehlszählers zu der angegebenen Adresse.
- Bedingte Sprungbefehle lösen nur dann einen Programmsprung aus, wenn die im Befehl spezifizierten Flags einen bestimmten Zustand aufweisen.

Tab. 4.4.1 zeigt die 4 Sprungbefehle.

Bedingung	Befehl	Bedeutung
0 0	JUMP	Springe unbedingt
0 1	JMPZ	Springe, wenn Zero-Flag = 1
1 0	JMPN	Springe, wenn Negativ-Flag = 1
1 1	JMPC	Springe, wenn Carry-Flag = 1

Tab. 4.4.1
Sprungbefehle

Alle Sprungbefehle haben den OP-Code E_{16} = 1 1 1 0_2. Die mnemonische Befehlsschreibweise JUMP bzw. JMP kommt von dem englischen Wort jump = springen. Die Adresse, wohin zu springen ist, wird durch den MODE wie bei den Befehlen der 2. Gruppe bestimmt. Hierzu ist allerdings folgendes zu bemerken: Aus historischen Gründen werden die Adressierungsarten bei Sprungbefehlen etwas anders definiert als bei der 2. Befehlsgruppe. Der LOAD-Befehl mit Immediate-Adressierung lädt bekanntlich eine Konstante, die im nächsten Wort steht, in das DST-Register. Dieses wird durch MODE 0 0 erreicht. Ein Sprungbefehl mit MODE 0 0 lädt die Konstante, die im nächsten Wort steht, in den Befehlszähler, d.h., der Befehlszähler springt auf die Adresse, die im zweiten Befehlswort angegeben ist. Dieser Vorgang wird als **direkte Adressierung** anstatt Immediate-Adressierung bezeichnet. Bei MODE 0 1 handelt es sich dann um eine **indirekte Adressierung.**
Bei dieser Adressierungsart wird das 2. Byte als Adresse benutzt, und mit dem Inhalt dieser Adresse wird der Befehlszähler geladen. Dadurch erfolgt ein Sprung indirekt zu einer neuen Programmadresse.
Wenn bei den Sprungbefehlen kein besonderer Hinweis gegeben wird, handelt es sich immer um direkte Adressierung mit MODE 0 0.
Die indirekte Adressierung wird häufig durch das Zeichen @ gekennzeichnet. So bedeutet z.B. der bedingte Sprungbefehl JMPN @ 4 3 folgendes: Springe zu der Adresse, die unter der Adresse 4 3_{16} im Speicher steht, wenn das Negativ-Flag 1 ist.
Der Maschinencode für diesen Befehl hat folgende Form:



Die Sprungbefehle beeinflussen die Flags nicht. In handelsüblichen Mikroprozessoren gibt es auch Sprungbefehle, die ausgelöst werden müssen, wenn die entsprechenden Flags gleich 0 sind. Damit ist es möglich, Bedingungen umgekehrt nachzuprüfen.
Um dies mit dem hypothetischen Mikrorechner zu tun, werden 2 Sprungbefehle benötigt, wie das Beispiel in Tab. 4.4.2 zeigt.

Adresse	Inhalt	Befehl	Kommentar
5 0	E 3	JMPC 5 4	MODE 0 0 = direkt
5 1	5 4		
5 2	E 0	JUMP 7 F	MODE 0 0 = direkt
5 3	7 F		
5 4	..	.....	

Tab. 4.4.2
Beispiel für das umgekehrte Nachprüfen einer Bedingung mit 2 Sprungbefehlen

Wenn der Befehlszähler die Programmadresse 5 0 anspricht, steht hier der Befehl „Springe auf Adresse 5 4, wenn das Carry-Flag = 1 ist". Da MODE 0 0 vorliegt, ist die Adresse 5 4 im zweiten Befehlswort direkt angegeben.
Dieser Sprung läuft aber nur ab, wenn das Carry-Flag = 1 ist. Ist das nicht der Fall, wird unter der Programmadresse 5 2 der Befehl E 0 gleich „Springe unbedingt auf die Programmadresse 7 F" durchgeführt. Wenn also das Carry-Flag = 0 ist, springt der Befehlszähler zur Adresse 7 F ansonsten wird das Programm bei Adresse 5 4 fortgesetzt.

Exp. 6c

Fragen zu den Abschnitten 4.3 und 4.4

1. Welche Flags enthält der hypothetische Mikrorechner?

2. Mit einem Additionsbefehl ADDM werden folgende Binärzahlen addiert:

```
  1 0 0 0 0 0 0 0
+ 1 0 0 0 0 0 0 0
-----------------
```

Welche Flags sprechen hierbei an?

3. Welche Flags sprechen an, wenn folgende Binärzahlen addiert werden:

```
  0 1 0 0 0 0 0 0
+ 0 1 0 0 0 0 0 0
-----------------
```

4. Nachfolgende Binärzahlen werden über den Befehl SUBM subtrahiert. Geben Sie an, welche Flags ansprechen müssen!

a)
```
  0 1 0 0 0 0 0 0
- 0 0 1 0 0 0 0 0
-----------------
```

b)
```
  0 0 0 0 1 0 0 0
- 0 0 0 1 0 0 0 0
-----------------
```

c)
```
  1 1 0 0 0 0 0 0
- 0 1 0 0 0 1 1 0
-----------------
```

d)
```
  1 0 0 0 0 1 0 0
- 0 0 0 0 0 0 1 1
-----------------
```

5. Welcher Unterschied besteht zwischen einem bedingten und einem unbedingten Sprung?

6. Welcher Befehl wird durch nachfolgenden Maschinencode ausgelöst?

```
1. Byte:   1 1 1 0 0 1 0 1
2. Byte:   0 1 0 0 0 0 0 0
```

7. Erläutern Sie den Befehl:

```
JMPC@ 9 0
```

4.5 Weitere Adressierungsarten des hypothetischen Mikrorechners

Der hypothetische Mikrorechner besitzt noch 2 weitere Adressierungsarten, die zusammen mit den Befehlen der 2. Gruppe und den Sprungbefehlen verwendet werden können. Diese lassen sich am einfachsten anhand eines Beispieles erklären:

Es sei die Aufgabe gestellt, die Speicherinhalte der Adressen $9\ 0_{16}$ bis $D\ 0_{16}$ zu addieren. Man könnte dies über entsprechend viele ADDM-Befehle realisieren (Tab. 4.5.1)

Adresse	Inhalt	Befehl	Kommentar
0 0	2 0	SUBR R0, R0	lösche R0
0 1	9 4	ADDM R0, 9 0	Addiere erstes Wort in R0 hinein
0 2	9 0		
0 3	9 4	ADDM R0, 9 1	Addiere zweites Wort in R0 hinein
0 4	9 1		
0 5	9 4	ADDM R0, 9 2	Addiere drittes Wort in R0 hinein
0 6	9 2		
⋮	⋮ usw.		
8 1	9 4	ADDM R0, 0 D 0	Addiere letztes Wort in R0 hinein
8 2	D 0		
8 3	0 0	HALT	

Tab. 4.5.1
Programm für die Addition der Inhalte der Adressen $9\ 0_{16}$ bis $D\ 0_{16}$

Zu diesem Programm noch folgende Erläuterungen: Als Adressierungsart wird direkte Adressierung verwendet. Mit der Adresse 0 0 im Programmteil des Speichers wird über den Befehl SUBR R0, R0 der Inhalt des Registers R 0 gelöscht. Mit der Programmadresse 0 1 beginnt das eigentliche Additionsprogramm (vergleiche hierzu auch Tab. 4.2.3). Nachdem alle Speicherinhalte addiert sind, folgt als letzter Programmbefehl HALT.
Eine Besonderheit in der Befehlsschreibweise gibt es bei der Adresse 8 1. Anstatt D 0 für die Adresse wird im Befehl 0 D 0 geschrieben. Es ist üblich, Symbole mit einem Buchstaben und Hexadezimalzahlen mit einer Ziffer bei Befehlsangaben zu benutzen. Dementsprechend könnte D 0 als Symbol und nicht als Hexadezimalzahl gewertet werden. Aus diesem Grunde wird 0 D 0 geschrieben.
Aus diesem Beispiel können Sie selbst erkennen, daß hierfür ein relativ langes Programm erforderlich ist. Eine wesentliche Verkürzung ergibt sich durch die sogenannte **Indexed-Adressierung.** Die Idee, die hinter dieser Adressierungsart steht, ist die, mit **einem** ADDM-Befehl auszukommen. Dadurch, daß man diesen in eine **Programmschleife** einbaut, kann er mehrmals benutzt werden.
Bei der Indexed-Adressierung liegt die Adresse in einem der Akkumulatoren, in unserem Falle im Akkumulator R2. Aus diesem Grunde wird dieser Akkumulator auch als **Indexregister** bezeichnet.
Über MODE 1 0 wird R2 als Indexregister betrieben. Es ist zu beachten, daß dies nur bei R2 möglich ist. Betrachten wir als Befehlsbeispiel:

ADDM R0, @ R2

Hierfür lautet der Maschinencode:

OP-Code				MODE		DST	
1	0	0	1	1	0	0	0

Dieser Befehl bewirkt, daß **der Inhalt** der sich im Indexregister befindenden Speicheradresse in R0 „hinein" addiert wird. Das Zeichen @ für indirekt besagt hier, daß der Inhalt von R2 nicht direkt in R0 addiert, sondern als Adresse benutzt wird. Wir wollen jetzt anhand des geschriebenen Programms die Funktion dieser Adressierungsart erläutern.
Als Beispiel hierfür wählen wir wieder die Additionsaufgabe der Adresseninhalte 9 0 bis D 0. Das Programm soll mit der Programmadresse 4 0 beginnen (Tab. 4.5.2).
Diesem Programm liegen folgende Bedingungen zugrunde:
- Als Indexregister dient R2
- Register R1 dient als Zähler für die abzuarbeitenden Datenwörter (Schleifenzähler)
- Als Zielregister DST dient R0

Adresse	Inhalt	Befehl	Kommentar
4 0	8 2	LOAD R2, # 9 0	Adresse der ersten Daten
4 1	9 0		
4 2	8 1	LOAD R1, # 4 1	Anzahl der Datenwörter
4 3	4 1		
4 4	2 0	SUBR R0, R0	lösche laufende Summe in DST
4 5	9 8	ADDM R0,@ R2	addiere Inhalt R2 in DST
4 6	9 2	ADDM R2, # 1	erhöhe Datenadresse
4 7	0 1		
4 8	A 1	SUBM R1, # 1	erniedrige Schleifenzähler
4 9	0 1		
4 A	E 1	JMPZ 4 E	aufhören falls R1 = 0
4 B	4 E		
4 C	E 0	JUMP 4 5	sonst wiederholen
4 D	4 5		
4 E	0 0	HALT	

Tab. 4.5.2
Programm für die Addition nach Tab. 4.5.1 mit Indexed-Adressierung

Die ersten 5 Adressen des Programms dienen zur Vorbereitung (4 0 bis 4 4). Der Hauptteil des Programms beginnt bei Adresse 4 5.
Zuerst wird durch den Befehl LOAD R2, # 9 0 die Adresse (9 0) in das Indexregister R2 geladen. Als nächstes muß vom Programmierer festgestellt werden, wie viele Adressen der Bereich 9 0 bis D 0 ausmacht, es sind 4 1_{16}. Mit dieser Anzahl wird über den Befehl LOAD R1, # 4 1 der Schleifenzähler R1 geladen. Dann wird noch über SUBR R0, R0 das Zählregister gelöscht. Jetzt ist der Befehlszähler bei der Adresse 4 5 angelangt. Entsprechend dem Befehl ADDM R0,@ R2 wird in R0 der **Inhalt** der Adresse 9 0, die im Indexregister steht, zum Inhalt Null des Registers addiert. Jetzt wird das Indexregister um 1 erhöht, steht also dann auf 9 1. Da der Inhalt einer Adresse bereits addiert ist, müssen nur noch die Zahlen der Adressen von 9 1 bis D 0 addiert werden, d.h., der Schleifenzähler muß über den Befehl SUBM R1, # 1 um 1 zurückgestellt werden.
Bei der Adresse 4 A muß jetzt eine Entscheidung getroffen werden. Solange der Schleifenzähler noch nicht bei 0 angelangt ist – er steht im Moment noch bei 40 –, muß das Programm wiederholt werden. Erst bei R1 = 0 darf der HALT-Befehl kommen. Da R1 im Moment noch 40 ist, wird bei der Adresse 4 C mit dem unbedingten Sprungbefehl JUMP 4 5 der Befehlszähler auf die Adresse 4 5 zurückgestellt. Da vorher das Indexregister mit der Adresse 9 1 geladen wurde, wird jetzt zum Inhalt R0 der Inhalt der Adresse 9 1 addiert, das Indexregister mit der Adresse 9 2 geladen, der Schleifenzähler auf 3 F zurückgestellt usw. Dieser Vorgang wiederholt sich so oft, bis der Schleifenzähler auf 0 0 angelangt ist. Jetzt wird über den Befehl JMPZ 4 E (bedingt dadurch, daß das Zero-Flag jetzt gesetzt wird) der bedingte Sprung nach Adresse 4 E gleich HALT durchgeführt. Insgesamt wurde also 4 1_{16}-mal, das sind 65_{10}-mal, die Schleife durchlaufen.
Der Vorteil gegenüber dem ersten Programm mit direkter Adressierung ist offensichtlich. Hierbei wurden nämlich Speicheradressen von 0 0_{16} bis 83_{16} benötigt, d.h. 132_{10} Wörter. Die Indexed-Adressierung benötigt dagegen nur Adressen von 4 0_{16} bis 4 E_{16}, d.h. 15_{10} Wörter.
Das letzte Programm ist ein typisches Beispiel für die Verarbeitung von listenmäßig zusammengefaßten Daten (Listenverarbeitung). Solche Programme kommen in der Praxis sehr häufig vor und damit auch die 2 Befehle

ADDM R0,@ R2

und

ADDM R2, # 1

Diese sehr oft benutzte Befehlsfolge wird in der Praxis oft mit einem automatischen Erhöhen des Indexregisters kombiniert. Eine derartige Adressierung wird dann als **Auto-Increment-**

40
42
44
45
46
48
4A
4C
4E

Exp. 6d

Indexed-Adressierung bezeichnet. Auch der hypothetische Mikrorechner verfügt über diese Adressierungsart, wenn MODE 1 1 programmiert ist. In diesem Falle arbeitet Akkumulator R3 als ein sog. **Autoindexregister.** Prinzipiell arbeitet R3 dann wie R2 im letzten Programm mit dem Unterschied, daß eine automatische Erhöhung der Datenadresse erfolgt. Damit wird der Befehl ADDM R2, # 1 bei Adresse 4 6 nicht mehr benötigt. Die Lösung des genannten Beispieles erfordert demnach das in Tab. 4.5.3 gezeigte Programm.

Adresse	Inhalt	Befehl	Kommentar
4 0	8 3	LOAD R3, # 9 0	Adresse der ersten Daten
4 1	9 0		
4 2	8 1	LOAD R1, # 4 1	Anzahl der Datenwörter
4 3	4 1		
4 4	2 0	SUBR R0, R0	lösche Inhalt im DST R0
4 5	9 C	ADDM R0, @ R3 ↑	
4 6	A 1	SUBM R1, # 1	erniedrige Schleifenzähler
4 7	0 1		
4 8	E 1	JMPZ 4 C	aufhören, falls R1 = 0
4 9	4 C		
4 A	E 0	JUMP 4 5	sonst wiederholen
4 B	4 5		
4 C	0 0	HALT	

Tab. 4.5.3
Programm für die Addition nach Tab. 4.5.1 mit Auto-Increment-Indexed-Adressierung

Daß es sich um Auto-Increment-Indexed-Adressierung handelt, wird in der mnemonischen Befehlsschreibweise hier durch den Pfeil ↑ gekennzeichnet. Pfeil nach R3
Gegenüber der normalen Indexed-Adressierung ist das Programm nur um 2 Programmadressen kürzer geworden. Wesentlich ist aber, daß diese Programmkürzung in der Schleife liegt. Jede Kürzung innerhalb der Schleife beeinflußt die Rechengeschwindigkeit sehr stark, weil die Schleife ja mehrmals durchlaufen werden muß. Beim hypothetischen Mikrorechner kann diese Adressierungsart nur mit Akkumulator R3 durchgeführt werden.

Auto-Increment-Indexed

Exp. 6e

Weitere Beispiele dieser Adressierungsart werden später gebracht, wenn die restlichen Befehle erklärt worden sind.

Fragen zu Abschnitt 4.5

1. Welches Register kann beim hypothetischen Mikrorechner als Indexregister benutzt werden?

2. Erläutern Sie den Befehl:

SUBM R1, @ R2

3. Geben Sie für den Befehl in Frage 2. den Maschinencode an!

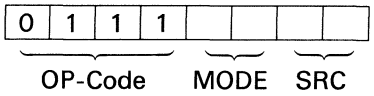
4. Erklären Sie den Befehl:

JUMP @ R2

5. Welcher Unterschied besteht zwischen der Indexed-Adressierung und der Auto-Increment-Indexed-Adressierung?

4.6 STAC-Befehl

Der STAC-Befehl (Store Akkumulator = speichere Akku ab) speichert den Inhalt eines im Befehl spezifizierten Registers im Speicher ab. Der Inhalt des SRC-Registers wird dabei nicht geändert. Dieser Befehl hat die Form:



Bis auf MODE 0 0 entsprechen die Modes denen der 2. Befehlsgruppe. MODE 0 0 = Immediate-Adressierung hat bei einem STAC-Befehl keinen Sinn, da konstante Daten vorhanden sind und diese somit nicht abgespeichert werden müssen. Außerdem ist das Programm normalerweise in einem ROM gespeichert und damit nicht mehr veränderbar. Die restlichen 3 Modes haben folgende Bedeutung:

MODE 0 1 (direkte Adressierung):
Der Befehl STAC R3, 4 7 z.B. speichert den Inhalt von R3 in der Adresse 4 7 ab.

Anmerkung:
Die Schreibweise dieses Befehles wurde so gewählt, daß sie kompatibel mit der Intel-8080-Schreibweise ist. Leider ist dadurch eine Inkompatibilität mit den früher besprochenen Befehlen eingetreten (SRC und DST wie beim Intel 8080 vertauscht).

MODE 1 0 (Indexed-Adressierung):
Der Befehl STAC R0, @ R2 speichert den Inhalt von R0 unter der Speicheradresse ab, die in R2 steht.

MODE 1 1 (Auto-Increment-Indexed-Adressierung):
Der Befehl STAC R2, @ R3 ↑ speichert den Inhalt von R2 unter der Speicheradresse ab, die in R3 steht, und erhöht anschließend R3 automatisch um 1.

In manchen Anwendungsfällen ist es erforderlich, eine Liste von Daten rückwärts zu verarbeiten. Dies bedeutet, daß mit der höheren Adresse begonnen und mit jeder Schleife dann die nächstniedrigere Adresse angewählt wird. Da wir bei unserem hypothetischen Mikrorechner aber nur 2 bit für den MODE haben, sind nur 4 Adressierungsarten möglich. Da aber beim STAC-Befehl die Immediate-Adressierung mit MODE 0 0 sinnlos ist, erhält der STAC-Befehl mit diesem MODE die Funktion der **Auto-Decrement-Indexed-Adressierung.** Diese Adressierungsart ist der Auto-Increment-Indexed-Adressierung sehr ähnlich. Folgende 2 Unterschiede sind zu beachten:

– Das Indexregister R3 wird mit jeder Schleife auf die nächstniedrigere Adresse geschaltet. Dadurch ist es möglich, eine Liste rückwärts abzuarbeiten.

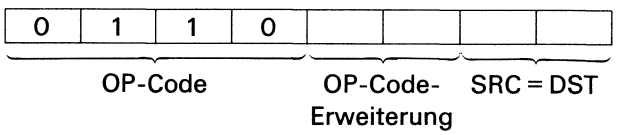
– Das Indexregister wird vom Programmablauf zunächst erniedrigt, **bevor** die dann vorhandene Adresse benützt wird. Der Grund hierfür wird später bei der Erklärung eines „Push-Down-Stacks" genannt.

Um die zweite Eigenschaft zu kennzeichnen, wird ein Pfeil nach unten **vor** R3 gesetzt (z.B. STAC, @ ↓ R3). Bei einer Listenverarbeitung muß diese Eigenschaft dadurch berücksichtigt werden, daß beim Programmbeginn R3 auf eine Adresse höher gebracht wird als die eigentlich gewollte Programmadresse.

Exp. 6f

4.7 INCR-, DECR-, RACL- und RACR-Befehle

Diese 4 Befehle haben den OP-Code 0 1 1 0_2 = 6_{16} und ändern den Inhalt eines der 4 Akkumulatoren. Der veränderte Inhalt wird anschließend zurück in denselben Akkumulator geschrieben.Dies bedeutet, daß SRC-Register und DST-Register gleich sind. Die Befehle haben folgende Form:



Es entsteht quasi eine Erweiterung des OP-Codes um 2 bit, wodurch einer der 4 Befehle nach dem in Tab. 4.7.1 gezeigten Schema spezifiziert werden kann.

OP-Code-Erweiterung	Befehl
0 0	INCR
0 1	DECR
1 0	RACL
1 1	RACR

Tab. 4.7.1
OP-Code-Erweiterung

Der Befehl INCR (Increment Register) **erhöht** den Inhalt des SRC-Registers um 1, der Befehl DECR (Decrement Register) erniedrigt den Inhalt des SRC-Registers um 1. In beiden Fällen werden Zero- und Negativ-Flag beeinflußt, das Carry-Flag aber nicht.
Die Befehle RACL und RACR verschieben (rotieren) die Inhalte des SRC-Registers und des Carry-Flags nach links bzw. nach rechts. Dies bedeutet, daß entsprechend Bild 4.7.1 Carry-Flag und SRC-Register zusammen wie ein 9-bit-Schieberegister funktionieren.

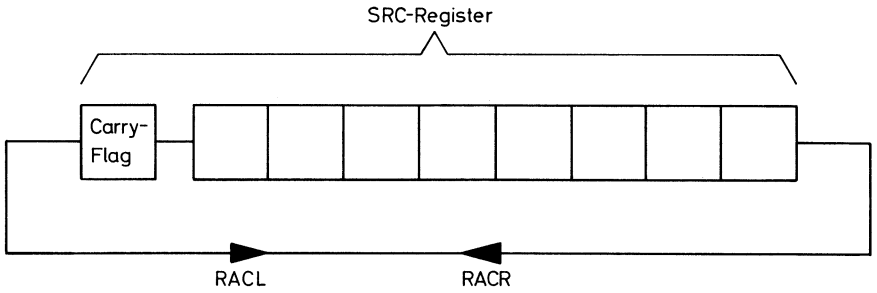


Bild 4.7.1
SRC-Register mit Carry-Flag in der Funktion als Rechts-Links-Schieberegister

Wenn die Funktion eines normalen Schieberegisters verlangt wird (ohne die kreisförmige Rückkopplung nach Bild 4.7.1), muß das Carry-Flag zuerst gelöscht werden. Dieses kann mit dem Befehl ADDM R0, # 0 erzielt werden. Hierbei wird 0 zum Inhalt von R0 addiert. Dadurch wird der Inhalt von R0 nicht verändert und kein Übertrag erzeugt.
Die Befehle RACL und RACR ändern nur den Zustand des Carry-Flags, die beiden anderen Flags werden nicht beeinflußt.

Exp. 6g

Fragen zu den Abschnitten 4.6 und 4.7

1. Welche Aufgabe hat der STAC-Befehl?

2. Erläutern Sie den Befehl:

STAC R1, @ R2

3. Erläutern Sie den Befehl:

STAC R1, @ R3 ↑

4. Was verstehen Sie beim STAC-Befehl unter Auto-Decrement-Indexed-Adressierung?

5. Was ist bei der Auto-Decrement-Indexed-Adressierung besonders zu berücksichtigen?

6. Geben Sie für den Befehl INCR R0 den Maschinencode an!

7. Welche Flags werden durch die Befehle RACL und RACR beeinflußt?

4.8 Zusammenfassung der Befehle des hypothetischen Mikrorechners

Der hypothetische Mikrorechner hat alle Befehle, die man braucht, um jede Art von Problemen lösen zu können. Bis auf die Call-Befehle, die im nächsten Abschnitt beschrieben werden, sind alle Befehle angesprochen. Selbstverständlich können in einem Rechner leistungsfähigere Befehle eingebaut werden, die es ermöglichen, Programme schneller, kürzer und mit weniger Fehlermöglichkeiten zu schreiben. Hierauf wird bei der Besprechung des Intel 8080 Mikroprozessors noch näher eingegangen. Wir haben gesehen, daß die Befehle ihrer Art nach zur Ausführung von arithmetischen und logischen Operationen sowie zur Programmablaufsteuerung dienen können (Sprung- und HALT-Befehle). Alle Befehle sind in Tab. 4.8.1 zusammengefaßt. Tab. 4.8.2 zeigt eine Zusammenstellung der Adressierungsarten des hypothetischen Mikrorechners.

Befehl	Form	Funktion	N	Z	C
HALT	0 0 0 0 0 0 0 0	hält den Rechner an	–	–	–
NOP	0 0 0 0 1 1 1 1	keine Operation	–	–	–
MOVE	0 0 0 0 s s d d	(ss) → dd	–	–	–
ADDR	0 0 0 1 s s d d	(ss) + (dd) → dd	↕	↕	↕
SUBR	0 0 1 0 s s d d	(dd) − (ss) → dd	↕	↕	↕
IORR	0 0 1 1 s s d d	(ss) V (dd) → dd	↕	↕	0
XORR	0 1 0 0 s s d d	(ss) ∀ (dd) → dd	↕	↕	0
ANDR	0 1 0 1 s s d d	(ss) Λ (dd) → dd	↕	↕	0
INCR	0 1 1 0 0 0 s s	(ss) + 1 → ss	↕	↕	–
DECR	0 1 1 0 0 1 s s	(ss) − 1 → ss	↕	↕	–
RACL	0 1 1 0 1 0 s s	Rotiere ss und Carry links	–	–	↕
RACR	0 1 1 0 1 1 s s	Rotiere ss und Carry rechts	–	–	↕
STAC	0 1 1 1 m m s s	(ss) → (mm)	–	–	–
LOAD	1 0 0 0 m m d d	(mm) → dd	–	–	–
ADDM	1 0 0 1 m m d d	(mm) + (dd) → dd	↕	↕	↕
SUBM	1 0 1 0 m m d d	(dd) − (mm) → dd	↕	↕	↕
IORM	1 0 1 1 m m d d	(mm) V (dd) → dd	↕	↕	0
XORM	1 1 0 0 m m d d	(mm) ∀ (dd) → dd	↕	↕	0
ANDM	1 1 0 1 m m d d	(mm) Λ (dd) → dd	↕	↕	0
JUMP	1 1 1 0 m m 0 0	(mm) → PC	–	–	–
JMPZ	1 1 1 0 m m 0 1	Falls Z = 1, (mm) → PC	–	–	–
JMPN	1 1 1 0 m m 1 0	Falls N = 1, (mm) → PC	–	–	–
JMPC	1 1 1 0 m m 1 1	Falls C = 1, (mm) → PC	–	–	–
CALL	1 1 1 1 m m 0 0	Unterprogrammanruf	–	–	–
CALZ	1 1 1 1 m m 0 1	Falls Z = 1, Unterprogrammanruf	–	–	–
CALN	1 1 1 1 m m 1 0	Falls N = 1, Unterprogrammanruf	–	–	–
CALC	1 1 1 1 m m 1 1	Falls C = 1, Unterprogrammanruf	–	–	–

ss = SRC = Source-Register
dd = DST = Destination-Register
mm = MODE
N = Negativ-Flag
Z = Zero-Flag
C = Carry-Flag

Flags: ↕ bedeutet, Flag wird auf 0 oder 1 gesetzt
– bedeutet, Flag wird nicht beeinflußt
0 bedeutet, Flag wird auf 0 gesetzt

Tab. 4.8.1
Zusammenstellung der Befehle des hypothetischen Mikrorechners

Befehl	LOAD ADDM SUBM IORM XORM ANDM		STAC		JUMP JMPZ JMPN JMPC		CALL CALZ CALN CALZ	
Mode 0 0	immediate	#	auto- decrement- indexed über R3	@↓R3	absolut oder direkt		absolut oder direkt	
Mode 0 1	absolut oder direkt		absolut oder direkt		indirekt	@	indirekt	@
Mode 1 0	indexed über R2	@R2	indexed über R2	@R2	indexed über R2	@R2	indexed über R2	@R2
Mode 1 1	auto- increment- indexed über R3	@R3 ↑	auto- increment- indexed über R3	@R3 ↑	auto- increment- indexed über R3	@R3 ↑	nicht benützt	nicht benützt

Tab. 4.8.2
Adressierungsarten des hypothetischen Mikrorechners

4.9 Allgemeine Adressierungsarten

Bis auf die CALL-Befehle wurden alle Adressierungsarten des hypothetischen Mikrorechners besprochen. Nachfolgend werden noch Befehle angesprochen, die in einigen Mikrorechnern zu finden sind.

4.9.1 Relative Adressierung

Bei dieser Adressierungsart enthält der Adreßteil des Befehles nur den Unterschied zwischen der gewünschten Adresse und der Adresse des Befehles selbst (Bild 4.9.1.1).
Der Ablauf ist folgender: Wenn der Befehlszähler die Adresse E 0 spezifiziert, wird über den OP-Code der Befehl LOAD R2, Rel 0 5 ausgelöst. Entscheidend ist hierbei, daß der Inhalt des zweiten Bytes, hier 0 5, zur eigentlichen Befehlsadresse, hier E 0, addiert werden muß (symbolisiert durch das Summenzeichen Σ), um die gewünschte Adresse zu erhalten. Die Summe E 0 + 0 5 = E 5 gibt die Adresse an, deren Inhalt (im Beispiel 2 1) in das Register R2 geladen werden soll. Der Inhalt des zweien Bytes ist also die Differenz zwischen der gewünschten Adresse und der eigentlichen Programmadresse. Diese Differenz wird auch als **Offset** oder **Displacement** bezeichnet. Wenn, wie in der Praxis häufig üblich, die Daten für ein bestimmtes Programmstück in dessen Nähe gespeichert sind, ist diese Differenz relativ klein und benötigt somit wenig Speicherplatz.
In der Befehlsschreibweise ist diese Adressierungsart durch den Vorsatz Rel gekennzeichnet. In der Praxis hätte man hier ein Sonderzeichen verwendet oder einfach LOAD R2, E 5 geschrieben. Wie später noch näher erläutert wird, ist es Aufgabe des Assemblers, eine entsprechende Umwandlung vorzunehmen.
Wenn ein Programm ausschließlich mit relativer Adressierung geschrieben wird, so kann dieses Programm innerhalb des Speicherbereiches beliebig versetzt werden, d.h., das Programm ist relocatable oder bewegbar. Im Gegensatz zur direkten oder indirekten Adressierung, bei denen bei einer Versetzung die Adressen in den Befehlen geändert werden müssen, kann bei ausschließlich relativer Adressierung das Programm ohne Änderung weiterbenutzt werden. In Mikrorechnern findet diese Adressierungsart hauptsächlich bei Sprungbefehlen Anwendung.

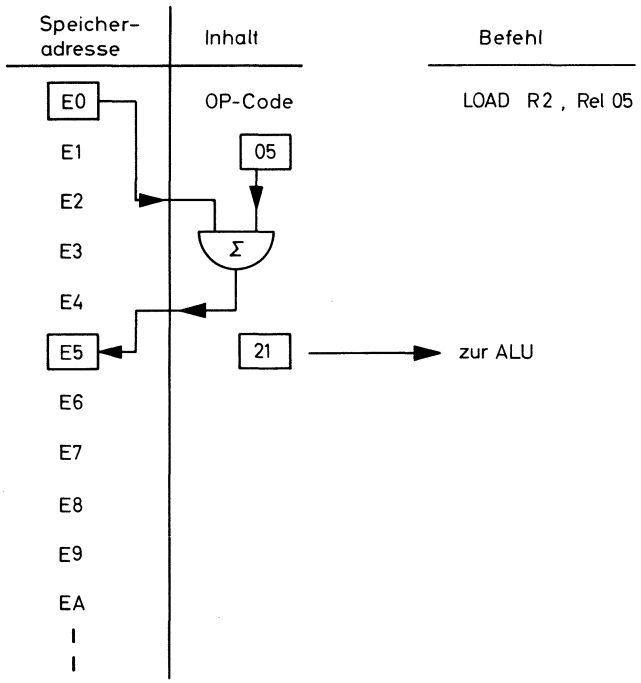


Bild 4.9.1.1
Prinzip der relativen Adressierung

4.9.2 Page-Relative-Adressierung

Ein Nachteil der relativen Adressierung ist, daß für die Summenbildung ein zusätzlicher Aufwand erforderlich ist. Entweder ist ein zweites Addierwerk erforderlich, oder die ALU wird für diese Operation benutzt. Im ersten Falle wird eine zusätzliche Logik gebraucht, im zweiten Falle muß die ALU im Zeitmultiplexverfahren betrieben werden. Zeitmultiplexverfahren bedeutet, daß die ALU wechselweise die beiden Aufgaben durchführen muß. Dies bedeutet aber, daß eine Verlangsamung der Rechengeschwindigkeit eintritt. Als Kompromiß kann die Page-Relative-Adressierung angesehen werden.
Bei diesem Verfahren besitzt der mögliche Offset immer weniger bit als der Befehlszähler. So könnten z.B. bei einem Befehlszähler mit 16 bit Wortlänge für den Offset nur 8 bit benutzt werden (Bild 4.9.2.1)

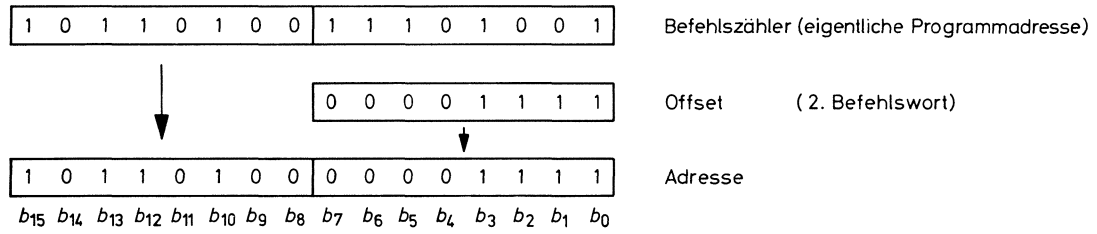


Bild 4.9.2.1
Prinzip der Page-Relative-Adressierung

Der Ablauf ist hier folgender: Wenn der 16-bit-Befehlszähler eine bestimmte Befehlsadresse anwählt, wird die neue Datenadresse aus den Offset-bit b_0 bis b_7 und den bit b_8 bis b_{15} der eigentlichen Programmadresse gebildet. Hierfür wird keine Addierfunktion benötigt.
Programme, die nur diese Adressierungsart benützen, sind Page relocatable. In dem angeführten Beispiel bedeutet dies, daß eine Versetzung immer nur mit Vielfachen von 2^8 erfolgen kann.

4.9.3 Base-Relative-Adressierung

Diese Adressierungsart ist eine Sonderform der Indexed-Adressierung. Auch hierbei wird die Adresse aus der Summe von 2 Teilen gebildet. Diese beiden Teile sind der Offset und der

Inhalt eines Indexregisters, wie z.B. R3 beim hypothetischen Mikrorechner. Das Prinzip zeigt Bild 4.9.3.1.

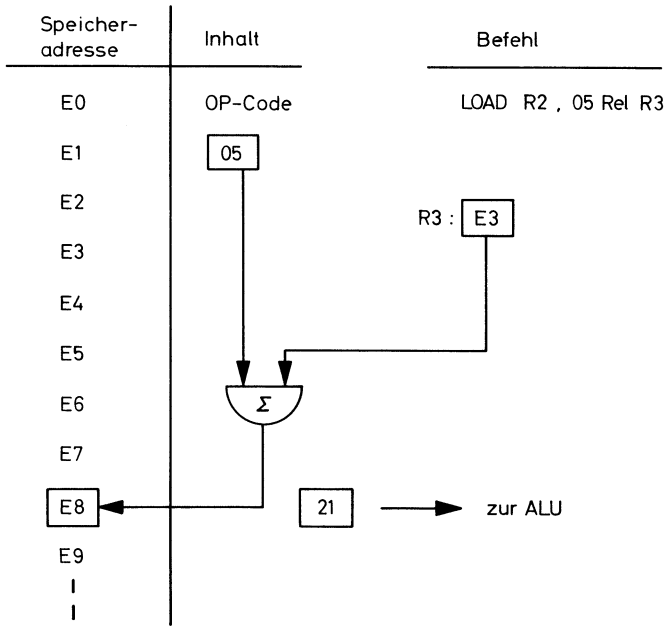


Bild 4.9.3.1
Prinzip der Base-Relative-Adressierung

Die Datenadresse E 8 mit dem Inhalt 2 1 ist hierbei die Summe aus Offset (0 5) und Inhalt des Indexregisters (E 3).
Diese Adressierungsart eignet sich besonders gut für Listenverarbeitung.

4.9.4 Pre-Indexed-Adressierung

Diese Adressierungsart ist eine **indirekte Version** der Base-Relative-Adressierung. Der Inhalt des Indexregisters wird zum Offset addiert. Die so entstandene Summe ergibt eine Adresse, mit der die eigentliche Datenadresse spezifiziert werden kann (Bild 4.9.4.1).

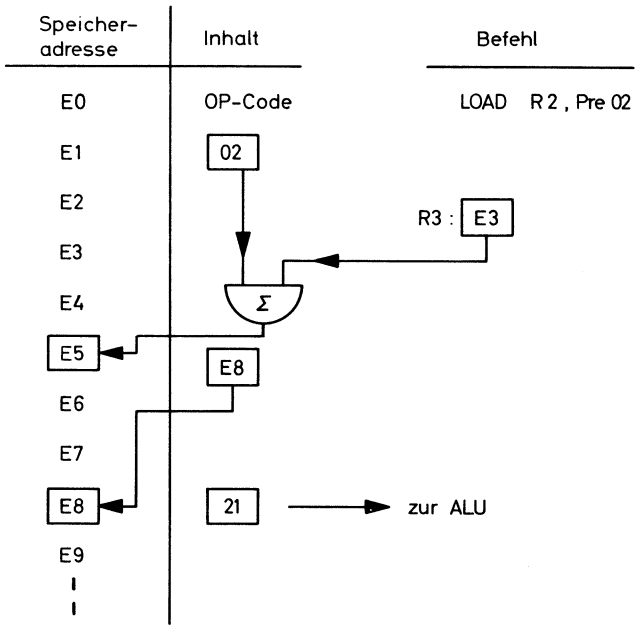


Bild 4.9.1
Prinzip der Pre-Indexed-Adressierung

Zunächst wird aus dem Offset des Befehlswortes (0 2) und dem Inhalt des Indexregisters (E 3) die Summe gebildet (E 5). Diese Summe wird quasi als Zwischenadresse benutzt. Der

Inhalt der Adresse E 5 (E 8) ist die eigentliche Datenadresse. Die Daten der Adresse E 8 (2 1) werden über die ALU in das Register R2 geladen.
Die Adressierungsart erscheint zwar etwas kompliziert, hat aber Vorteile beim Arbeiten mit Unterprogrammen. Hierauf werden wir in einem späteren Abschnitt noch näher eingehen.

4.9.5 Post-Indexed-Adressierung

Diese Adressierungsart ähnelt sehr der Pre-Indexed-Adressierung. Der Unterschied besteht darin, daß die Summe früher gebildet wird (Bild 4.9.5.1).

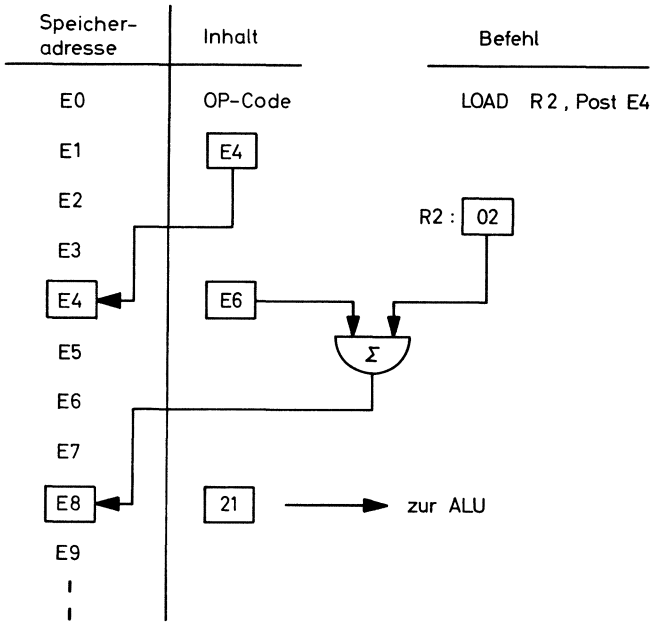


Bild 4.9.5.1
Prinzip der Post-Indexed-Adressierung

Die Anwendung dieser Adressierungsart ist nicht sehr häufig.

4.9.6 Register-Adressierung

Hierbei handelt es sich eigentlich nicht um eine spezielle Adressierungsart, sondern um eine Bezeichnung, die häufig in der Fachliteratur vorkommt. Allgemein spricht man von Register-Adressierung, wenn in einem Befehl ein Operand aus einem Register verwendet wird oder das Ergebnis einer Operation in einem Register abgespeichert werden soll. In diesem Falle muß im Befehl die Nummer des Registers angegeben werden.
Beispiel:

ADD R1, R2

Die Operanden sind die Inhalte der Register R1 und R2. Das Ergebnis wird in R2 abgespeichert.

4.9.7. Weitere Adressierungsarten

Es gibt beliebig viele Möglichkeiten, die einachen Adressierungsarten zu erweitern und zu kombinieren. Dadurch entstehen neue Adressierungsarten wie z.B. Doppel-Indirekt-Adressierung, Indirekt-Relativ-Adressierung usw. In solchen Fällen muß bei der Anwendung die genaue Definition des Mikroprozessor-Herstellers beachtet werden.

Exp. 6h

Fragen zu Abschnitt 4.9

1. Erläutern Sie den Befehl:

ADDM R0, Rel 0 3.

2. Erläutern Sie den Befehl

ADDM R1, 0 4 Rel R3

bei Base-Relativ-Adressierung!

3. Wie wird beim Befehl

LOAD R1, Pre 0 4

die Datenadresse bestimmt? Es wird Pre-Indexed-Adressierung mit R3 als Indexregister angewandt.

Anhang

Antworten auf die Fragen zu Abschnitt 4.1

1. Zur Befehlsgruppe gehören die Befehle:

MOVE
ADDR
SUBR
IORR
XORR
ANDR

2. Bei dieser Funktion wird der Inhalt des SRC-Registers mit dem Inhalt des DST-Registers ODER-verknüpft. Das Ergebnis erscheint im DST-Register.

3. ANDR R1, RO

4. Bei dieser Befehlsschreibweise ist der Datenfluß von rechts nach links. Damit ist R3 das DST-Register und R2 das SRC-Register.

5. Dieser Maschinencode stellt den Befehl

MOVE R3, R1

dar. Dadurch wird der Inhalt des SRC-Registers R1 in das DST-Register R3 transferiert. Der Inhalt von R1 bleibt dabei erhalten, d.h., beide Register haben nach diesem Befehl den gleichen Inhalt. Die Funktionsschreibweise dafür lautet:

(R1) → (R3)

6. Ein NOP-Befehl ist vom Prinzip her gesehen ein MOVE-Befehl, bei dem die Daten eines Registers in das gleiche Register übertragen werden. Dadurch erfolgt nach außen keine Operation. Beim hypothetischen Rechner wird der Befehl MOVE R3, R3 als NOP-Befehl benutzt.

Antworten auf die Fragen zu Abschnitt 4.2

1. Bei der 1. Befehlsgruppe sind die zu verarbeitenden Daten in einem der 4 Register RO bis R3 vorhanden. Bei der 2. Befehlsgruppe sind die Daten unter einer bestimmten Adresse im Speicher gespeichert.

2. Bei der direkten oder absoluten Adressierung wird die Adresse, unter der die Daten im Speicher gespeichert sind, im zweiten Byte des Befehlswortes angegeben.

3.

1. Byte:	1 0 0 1 0 1 0 0
2. Byte:	1 0 0 1 0 0 0 0

4. Während bei der direkten Adressierung im 2. Byte des Befehles die Datenadresse angegeben ist, sind bei der Immediate-Adressierung im 2. Byte die Daten angegeben.

5. Bei direkter Adressierung ist das 2. Byte eines Befehles die effektive Adresse.

6. Mit RO als DST-Register ergibt sich z.B. folgende Möglichkeit:

Adresse	Inhalt	Befehl	Kommentar
4 0	8 4	LOAD R0, 9 0	*x* in R0 laden
4 1	9 0		
4 2	9 4	ADDM R0, 9 1	Addition $x + y$
4 3	9 1		
4 4	A 4	SUBM R0, 9 2	Subtraktion $-z$
4 5	9 2		
4 6	B 4	IORM R0, 9 3	ODER-Verknüpfung bilden
4 7	9 3		
4 8	0 0	HALT	
.	.		
.	.		
.	.		
9 0	*x x*	} Daten	
9 1	*y y*		
9 2	*z z*		
9 3	*a a*		

In den Speicheradressen 9 0 bis 9 3 können für *a, x, y* und *z* beliebige Daten abgespeichert werden.

Antworten auf die Fragen zu den Abschnitten 4.3 und 4.4

1. Der hypothetische Mikrorechner enthält
- ein Carrry-Flag
- ein Negativ-Flag
- ein Zero-Flag

2. Das Ergebnis der Aufgabe lautet:

$$\begin{array}{r} 1\,0\,0\,0\,0\,0\,0\,0 \\ +\,1\,0\,0\,0\,0\,0\,0\,0 \\ \hline 1\,0\,0\,0\,0\,0\,0\,0\,0 \end{array}$$

Da ein Übertrag entsteht, spricht das Carry-Flag an. Außerdem spricht das Zero-Flag an, da das Ergebnis in den 8 bit Null ist.

3. In diesem Falle lautet das Ergebnis:

$$\begin{array}{r} 0\,1\,0\,0\,0\,0\,0\,0 \\ +\,0\,1\,0\,0\,0\,0\,0\,0 \\ \hline 1\,0\,0\,0\,0\,0\,0\,0 \end{array}$$

Da das Negativ-Flag dann anspricht, wenn im werthöchsten bit eine 1 steht (Zweierkomplementarithmetik), spricht dieses Flag jetzt an.

4. a) Die Lösung lautet:

$$\begin{array}{r} 0\,1\,0\,0\,0\,0\,0\,0 \\ -\,0\,0\,1\,0\,0\,0\,0\,0 \\ \hline 0\,0\,1\,0\,0\,0\,0\,0 \end{array}$$

Da von einer größeren positiven Zahl eine kleinere positive Zahl subtrahiert wird, bleibt das Ergebnis positiv. Damit spricht kein Flag an.

b) Die Lösung lautet:

```
  0 0 0 0 1 0 0 0
- 0 0 0 1 0 0 0 0
-----------------
Entlehnung 1 1 1 1 1 1 0 0 0
```

Da hier der Subtrahend größer als der Minuend ist, entsteht ein negatives Ergebnis. Damit spricht das Negativ-Flag an. Außerdem ist eine Entlehnung erforderlich, die vom Rechner als Übertrag gewertet wird, so daß auch das Carry-Flag anspricht. Bei einem SUBM-Befehl spricht das Carry-Flag dann grundsätzlich an, wenn vom **Betrag** her der Subtrahend größer als der Minuend ist.

c) Das Ergebnis lautet:

```
  1 1 0 0 0 0 0 0
- 0 1 0 0 0 1 1 0
-----------------
  0 1 1 1 1 0 1 0
```

Da keine Entlehnung notwendig ist und außerdem ein positives Ergebnis vorliegt, spricht kein Flag an. Auch bei dieser Aufgabe ist vom Betrag her der Minuend größer als der Subtrahend.

d) Das Ergebnis lautet:

```
  1 0 0 0 0 1 0 0
- 0 0 0 0 0 0 1 1
-----------------
  1 0 0 0 0 0 0 1
```

Die 1 im werthöchsten bit bewirkt ein Ansprechen des Negativ-Flags. Eine Entlehnung entsteht nicht, da auch hier vom Betrag her der Minuend größer als der Subtrahend ist.

5. Ein bedingter Sprung wird ausgelöst, wenn über ein Flag eine bestimmte Bedingung erfüllt ist. Ein unbedingter Sprung wird dagegen dann ausgeführt, wenn der Befehlszähler die Adresse mit dem Sprungbefehl angewählt hat.

6. Es handelt sich um einen bedingten Sprungbefehl, der dann ausgelöst wird, wenn das Zero-Flag gesetzt ist. Mode 0 1 besagt, daß sich um eine indirekte Adressierung handelt, d.h., der Befehlszähler springt auf die Adresse, die im Speicher unter der Adresse, die im 2. Byte angegeben ist, enthalten ist.

7. Dieser Befehl hat folgende Bedeutung:
Springe zu der Adresse, die unter der Adresse 9 0 im Speicher steht, wenn das Carry-Flag gesetzt ist.

Antworten auf die Fragen zu Abschnitt 4.5

1. Als Indexregister kann Register R2 verwendet werden.

2. Es handelt sich um einen Subtraktionsbefehl mit R1 als DST-Register. Außerdem wird Indexed-Adressierung benutzt, d.h., es werden vom Inhalt des Registers R1 die Daten subtrahiert, die unter der Adresse, die im Indexregister R2 steht, im Speicher vorhanden sind.

3. Der Maschinencode für den Befehl SUBM R1, @ R2 lautet:

1 0 1 0 1 0 0 1

4. Es handelt sich um einen unbedingten Sprungbefehl in Indexed-Adressierung. In diesem Falle springt der Befehlszähler auf die Adresse, die im Speicher unter der in R2 stehenden

Adresse abgespeichert ist. Hat z.B. R2 den Zustand 8 0 und in Adresse 8 0 ist 9 0 gespeichert, so springt der Befehlszähler auf die Adresse 9 0.

5. Bei der Auto-Increment-Indexed-Adressierung wird das Autoindexregister – beim hypothetischen Rechner Register R3 – automatisch nach jedem Durchlauf um eine Datenadresse erhöht. Hierfür ist bei der Indexed-Adressierung ein besonderer Befehl erforderlich.

Antworten auf die Fragen zu den Abschnitten 4.6 und 4.7

1. Mit dem STAC-Befehl können Daten eines im Befehl spezifizierten SRC-Registers im Speicher abgespeichert werden.

2. Dieser Befehl bewirkt, daß die Daten des Registers R1 unter der Adresse abgespeichert werden, die in R2 angegeben wird.

3. Dieser Befehl bewirkt, daß die Daten des Registers R1 unter der Adresse abgespeichert werden, die in R3 angegeben wird. R3 wird dabei automatisch um 1 erhöht.

4. Das Indexregister R3 wird mit jeder Schleife um 1 erniedrigt.

5. Das Indexregister wird vom Programmablauf zunächst erniedrigt, bevor die dann vorhandene Adresse benutzt wird. Deshalb muß beim Programmbeginn R3 auf eine Adresse höher gebracht werden als die eigentlich gewollte Programmadresse.

6. Der Maschinencode für INCR R0 lautet:

0 1 1 0 0 0 0 0

7. Die Befehle RACL und RACR beeinflussen nur das Carry-Flag.

Antworten auf die Fragen zu Abschnitt 4.9

1. Bei diesem Befehl handelt es sich um relative Adressierung. In diesem Beispiel werden zum Inhalt von R0 die Daten addiert, die 3 Adressen nach dieser Befehlsadresse abgespeichert sind.

2. Die Datenadresse, deren Inhalt zum Inhalt von R1 addiert wird, wird wie folgt bestimmt: Der Offset 0 4 und der Inhalt von R3 werden addiert. Die Summe ist dann die Datenadresse.

3. Zunächst wird die Summe 0 4 + Inhalt R3 gebildet. Damit ergibt sich eine Zwischenadresse, deren Inhalt die eigentliche Datenadresse bildet.