POCKET COMPUTER

FX-785P/FX-790P

EINFÜHRUNG IN DIE ASSEMBLER-SPRACHE

CASIO

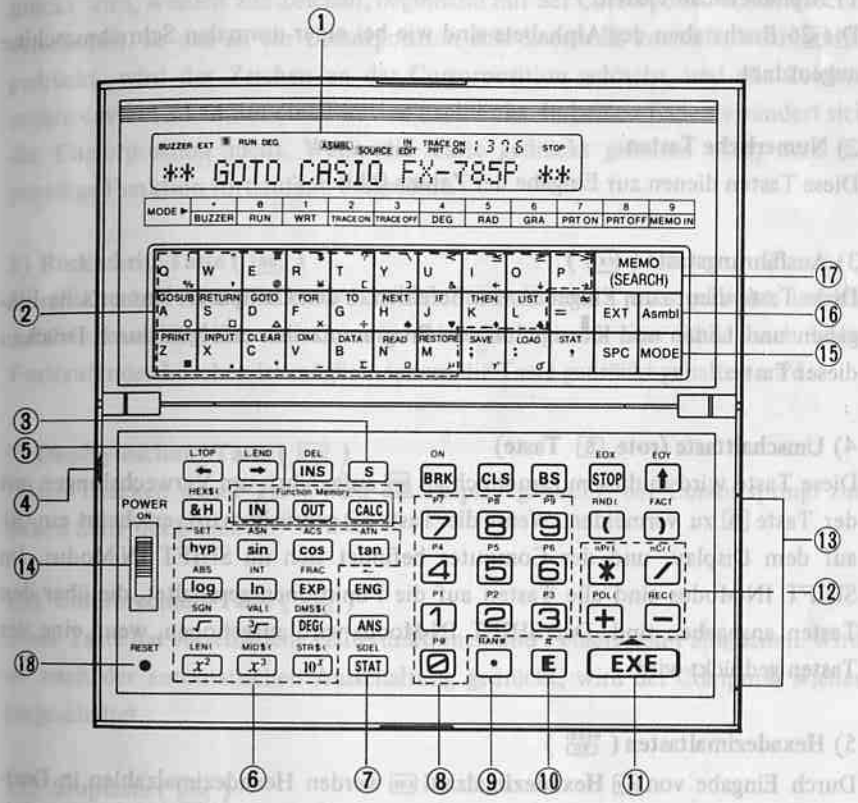
Einleitung

In diesem Handbuch wird das Arbeitsprinzip von Computern behandelt. Für diesen Zweck benutzen wir einen nicht vorhandenen Computer sehr einfachen Aufbaus als Vorstellung (der im folgenden als "hypothetischer Computer" bezeichnet wird). Dieser Computer verfügt über die Funktion, die Arbeitsweise des hypothetischen Computers zu imitieren. Programme können in der Assembler-Sprache eingegeben werden. Dann übersetzt dieser Computer sie in Maschinensprache und führt sie aus. Auf diese Weise können Sie mit diesem Computer die Arbeitsweise von Computern studieren, die allen Computern gemeinsam ist – von handlichen Geräten wie diesem bis hin zu Großrechnern.

Inhalt

1 Allgemeine Einführung

1-1 Bezeichnungen der Teile



① Display
② Alphabetische Tasten
③ Umschalttaste
④ Display-Kontrastregler
⑤ Tasten für Funktionsspeicher
⑥ Funktionstasten
⑦ Eingabetaste für statistische Daten
⑧ Numerische Tasten
⑨ Dezimalpunkttaste
⑩ Exponenttaste
⑪ Ausführungstaste
⑫ Tasten für Grundrechenarten
⑬ Anschluß für Peripheriegeräte
⑭ Ein/Aus-Schalter
⑮ Modus-Taste
⑯ Assemblieren-Taste
⑰ Speicher/Such-Taste
⑱ Initialisierungs-Knopf

*Die Display-Anzeige entspricht dem FX-785P.

1-2 Funktionen der Bedienelemente

1) **Alphabetische Tasten:**
Die 26 Buchstaben des Alphabets sind wie bei einer normalen Schreibmaschine angeordnet.

2) **Numerische Tasten:**
Diese Tasten dienen zur Eingabe der Zahlen 0 bis 9.

3) **Ausführungstaste (EXE)**
Diese Taste dient zum Eingeben von Befehlen in den Computer. Numerische Eingaben und Laden und Korrigieren von Programmzeilen erfolgen durch Drücken dieser Taste.

4) **Umschalttaste (rote S Taste)**
Diese Taste wird in diesem Handbuch als SHIFT aufgeführt, um Verwechslungen mit der Taste S zu vermeiden. Wenn die Taste SHIFT gedrückt wird, erscheint ein S auf dem Display, und der Computer befindet sich im SHIFT IN-Modus. Im SHIFT IN-Modus sind alle Tasten auf die Funktionen geschaltet, die über den Tasten angegeben sind. Der SHIFT IN-Modus wird aufgehoben, wenn eine der Tasten gedrückt wird.

5) **Hexadezimaltasten (HEX$(&H)**
Durch Eingabe von &H **Hexadezimalzahl** EXE werden Hexadezimalzahlen in Dezimalzahlen umgewandelt, die Umwandlung von Dezimalzahlen in Hexadezimalzahlen erfolgt durch Eingabe von SHIFT HEX$(**Dezimalzahl**) EXE .

6) **Cursor-Bewegungstasten (LTOP LEND)**
Diese Tasten dienen zum Bewegen des Cursors nach links und rechts. Wenn SHIFT LTOP gedrückt wird, geht der Cursor zum Zeilenanfang, und wenn SHIFT LEND gedrückt wird, zur Position rechts vom letzten Zeichen in der Zeile.

7) **Einfügung/Löschung-Taste (DEL INS)**
Diese Taste dient zum Korrigieren des angezeigten Zeichenstrings. Wenn INS gedrückt wird, werden alle Zeichen, beginnend mit der Cursorposition, nach rechts verschoben, so daß an der Cursorposition eine Leerstelle entsteht. Wird SHIFT DEL gedrückt, wird das Zeichen an der Cursorposition gelöscht, und alle Zeichen rechts davon rücken um eine Position nach links. In beiden Fällen verändert sich die Cursorposition nicht. Wenn die Taste gedrückt gehalten wird, wird die jeweilige Funktion fortlaufend ausgeführt.

8) **Rückschritt-Taste (BS)**
Durch Drücken dieser Taste wird das Zeichen links vom Cursor gelöscht, und alle Zeichen einschließlich der Cursorposition rücken um eine Stelle nach links. Fortlaufendes Löschen ist möglich, indem die Taste gedrückt gehalten wird.

9) **Displaylöschung-Taste (CLS)**
Durch Drücken dieser Taste wird das Display gelöscht, der Cursor springt zur linken Seite des Displays.

10) **Unterbrechung-Taste (BRK)**
Diese Taste hat verschiedene Haltefunktionen und Fehlerlösch-Fähigkeiten. Wird sie nach der automatischen Ausschaltung gedrückt, wird der Computer wieder eingeschaltet.

11) **Stoptaste (STOP)**
Durch Drücken dieser Taste kann das "scrollende" Display angehalten werden. Die Displayanzeige "rollt" weiter, wenn die Taste EXE gedrückt wird.

12) **Speicher/Such-Taste (MEMO SEARCH)**
Diese Taste dient zum Anzeigen des Quellenprogramms im Quellenbereich.

13) **Assemblierung-Taste (Asmbl)**
Durch Drücken dieser Taste wird das Assemblierung-Menü angezeigt. Zur Verfahrensweise siehe Kapitel 10.

14) **Modus-Taste (MODE)**

Mit dieser Taste wird der Betriebsmodus des Computers spezifiziert, sie wird in Verbindung mit den Tasten [.] , [0]~[9] verwendet. In diesem Handbuch werden die folgenden relativen Modi verwendet.

- MODE [.] . . . . Schaltet den Tasteneingabeton (Summer) ein und aus. Im eingeschalteten Zustand erscheint die Anzeige "BUZZER" oben links auf dem Display.
- MODE [0] . . . . Schaltet in den RUN-Modus.
- MODE [1] . . . . Schaltet in den WRT-Modus.
- MODE [2] . . . . Schaltet in den TRACE-Modus zur Verfolgung der Objektausführung. "TRACE ON" wird angezeigt. (Siehe 10-5 von Kapitel 10.)
- MODE [3] . . . . Schaltet den Trace-Modus aus.
- MODE [7] . . . . Schaltet in den Druckerausgabe-Modus, "PRT ON" wird angezeigt.
- MODE [8] . . . . Schaltet den Druckerausgabe-Modus aus.

2 Zahlendarstellung

Zunächst wollen wir die binäre und hexadezimale Darstellung von Zahlen kurz beschreiben, die später benötigt werden.

- **Binäre Darstellung**

Alle Informationen wie Programme und Daten werden im Computer binär dargestellt. Dazu werden nur die beiden Zahlen 0 und 1 verwendet. Die in binärer Darstellung ausgedrückten Zahlen werden Binärzahlen genannt, die einzelnen Ziffern der Binärzahlen werden als **Bit** bezeichnet. Eine Information, die in einer Binärzahl mit *n* Ziffern ausgedrückt wird, besteht daher aus **n Bits**.

Die Dezimalzahlen von 0 bis 10, mit denen wir täglich umgehen, sind in der folgenden Tabelle in binärer Darstellung aufgeführt. Die Tabelle daneben zeigt die Wertigkeit jeder Ziffer der Binärzahlen.

Dezimalzahl	Binärzahl
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010

Wertigkeit der einzelnen Ziffern von Binärzahlen	Entsprechende Dezimalzahl
2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1024
2^{11}	2048
2^{12}	4096
2^{13}	8192
2^{14}	16384
2^{15}	32768
2^{16}	65536

Beispielsweise ist die Dezimalzahl, die der Binärzahl 10101010 entspricht, gleich $2^7 + 2^5 + 2^3 + 2^1 = 170$.

- **Hexadezimale Darstellung**

Obwohl die Binärdarstellung für die Verwendung in Computern praktisch ist, wird oft die hexadezimale Darstellung benutzt, wenn viele Ziffern zum Ausdrücken einer Zahl benötigt werden, weil diese Darstellung einfacher zu lesen ist. Die 16 Zeichen 0 ~ 9 und A ~ F werden in der hexadezimalen Darstellung eingesetzt, und die in hexadezimaler Darstellung ausgedrückten Zahlen werden Hexadezimalzahlen genannt. Die folgende Tabelle zeigt die Beziehung zwischen Hexadezimalzahlen und Dezimalzahlen.

Hexadezimal-zahlen	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Dezimal-zahlen	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Die Umwandlung zwischen Hexadezimalzahlen und Dezimalzahlen kann mit diesem Computer auf einfache Weise mittels der Taste HEX$ &H durchgeführt werden. Da $2^4 = 16$ gilt, entspricht eine 4-stellige Binärzahl (4 Bits) einer 1-stelligen Hexadezimalzahl. Daher können Binärzahlen mit weniger Stellen in hexadezimaler Darstellung ausgedrückt und trotzdem die binäre Struktur beibehalten werden.

Die hexadezimale Darstellung erfolgt dabei derart, daß die Binärzahl in Gruppen von 4 Ziffern aufgeteilt wird, beginnend mit den niederwertigen Stellen, und jede Gruppe in eine Hexadezimal-Ziffer umgewandelt wird, siehe die Tabelle rechts. Falls die höherwertigen Stellen keine Gruppe zu 4 Ziffern ergeben, werden Nullen hinzugefügt.

4-stellige Binärzahl	Hexadezimal-zahlen
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Beispiel:

	Binärzahlen	**Hexadezimalzahlen**
hinzufügen	1011 0001	B 1
	B 1	
	0101 1111 0011 0111	5F37
	5 F 3 7	

3 Aufbau des Computers und Funktionsweise

Computer bestehen grundsätzlich aus den folgenden vier Teilen:

Die **Zentraleinheit**, abgekürzt als **CPU** der englischen Bezeichnung "Central Processing Unit", ist das Herz des Computers. Die heutigen modernen Computer arbeiten nach dem "**speicherprogrammierten System**". Programme und Daten werden im **Hauptspeicher** gespeichert. Die CPU ruft diese Informationen der Reihe nach ab und verarbeitet sie entsprechend den Anweisungen. In frühen Computern wurden für verschiedene Aufgaben separate Maschinen benötigt. Dagegen sind die CPUs in den heutigen Computern sehr hochentwickelt und können mit den im Hauptspeicher enthaltenen Programmen vielfältige Aufgaben ausführen. Das Gerät, das den Computer selbst bildet, wird als **Hardware** bezeichnet, während die Programme, die dem Computer die Arbeitsanweisungen geben, **Software** genannt werden.

Zur Kommunikation zwischen dem Computer und den ihn benutzenden Menschen sind Eingabe- und Ausgabevorrichtungen erforderlich.

Die **Eingabevorrichtung** dient zum Einlesen von Programmen und Daten in die CPU, ein typisches Beispiel ist die **Tastatur**. Die Eingabevorrichtung erfüllt die gleiche Rolle beim Computer wie die Augen und Ohren beim Menschen.

Die **Ausgabevorrichtung** überträgt die Ergebnisse der Berechnungen in der CPU nach außen auf einen **Bildschirm**, eine **Flüssigkristallanzeige** oder einen Drucker. Bei Menschen könnte man den Mund und die Hände als Ausgabevorrichtung bezeichnen. Die allgemeine Bezeichnung für Eingabe- und Ausgabevorrichtungen ist "**Ein/Ausgabegerät**" oder abgekürzt **I/O** (vom englischen Input/Output device).

Zusätzlich zu diesen Teilen werden **Hilfsspeicher** benötigt, die Informationen aufnehmen, die nicht im Hauptspeicher untergebracht werden können, und

außerdem zum Aufbewahren (Sichern) von Programmen und Daten dienen. Als Hilfsspeicher werden häufig **Magnetplatten** und **Cassetten** eingesetzt. Beim Menschen wären das Notizbücher und Merkhefte.

In vielen der Computer von heute werden quantitative Informationen in digitaler Form verarbeitet. Die kleinste Einheit einer digitalen Information kann durch die Stärke einer Spannung oder durch die Stellung eines Schalter auf Ein oder Aus bestimmt werden. Im Computer werden die digitalen Informationen durch die **Binärzahlen** 0 und 1 ausgedrückt. Diese Informationen werden daher in Einheiten von Binärbits (Bitanzahl) gemessen.

Wenn die CPU eines Computers fähig ist, *n* Bits Informationen gleichzeitig zu verarbeitet, wird dieser Computer als **n-Bit** Computer bezeichnet. Die heutigen Computer sind 4-Bit, 8-Bit, 16-Bit oder 32-Bit Computer.

Der Hauptspeicher ist in kleine Abschnitte, **Zellen** genannt, mit einer festen Bitanzahl unterteilt. Die Anweisungen und Daten von Programmen werden in diesen Zellen in der Form von binären **Bitmustern** gespeichert. Zur Unterscheidung erhält jede Zelle eine Adresse in der Form einer ganzen Zahl, die die Position der Zelle angibt. Normalerweise sind die Adressen in fortlaufender Folge, beginnend mit 0.

Adresse 0	1100010000000011
Adresse 1	0110010000010000
Adresse 2	0000000010000000
Adresse 3	1010101111001101
Adresse 4	0000000000000000
	⋮

(Interne Anordnung des Hauptspeichers mit 16 Bit in jeder Speicherzelle)

Programme bestehen also einfach aus Bitmustern, die in den einzelnen Zellen gespeichert sind, wie in der obigen Abbildung gezeigt. Bei der Entwicklung von CPUs wird präzise festgelegt, wie die CPU bestimmte Typen Bitmuster verarbeitet. Dies ist die einzige Sprache, die die CPU versteht, und wird als "**Maschinensprache**" bezeichnet.

Bitmuster sind jedoch schwierig zu lesen, daher werden zur Darstellung meistens Hexadezimalzahlen entsprechend dem Inhalt der einzelnen Zellen verwendet.

Adresse 0	C 4 0 3
Adresse 1	6 4 1 0
Adresse 2	0 0 8 0
Adresse 3	A B C D
Adresse 4	0 0 0 0
	⋮

(Jede Speicherzelle besteht aus 16 Bits.)

Programme in Maschinensprache sind hauptsächlich dafür erforderlich, dem Computer Anweisungen zu geben und diese Programme im Hauptspeicher zu speichern. Da die Maschinensprache jedoch für Menschen sehr schwierig ist, werden die Maschinensprache-Programme durch mnemonische Codes ausgedrückt. Ein durch mnemonische Codes ausgedrückte Maschinensprache wird **Assemblersprache** genannt.

Die im Hauptspeicher gespeicherten Programme und Daten sehen in Assemblersprache wie folgt aus:

Adresse 0	LD :1:3
Adresse 1	WRITE:1:16
Adresse 2	HJ :0:128
Adresse 3	CONST:ABCD

Anweisungen für den Computer

Daten (hexadezimal ABCD)

Wenn Programme auf diese Weise durch Assemblersprache ausgedrückt werden, kann man die Programme einfacher verstehen (dafür muß man natürlich die Syntax der Assemblersprache kennen), wodurch das Erstellen von Programmen erheblich vereinfacht wird. Da jedoch die CPU die Assemblersprache nicht versteht, ist es erforderlich, Programme in Assemblersprache in Maschinensprache umzuwandeln. Dieser Prozeß wird "**assemblieren**" genannt.

Assemblieren ist eine sehr langweilige Arbeit, die einfach darin besteht, die Assemblersprache Befehl für Befehl mit einer Codierungstabelle in Maschinensprache umzuwandeln. Daher hat man Programme entwickelt, um diese Arbeit vom Computer durchführen zu lassen. Solche Programme zum Umwandeln von Assemblersprache in Maschinensprache werden als **Assembler** bezeichnet.

*Da jede Anweisung in Assemblersprache eine einzelne Operation der CPU bewirkt, muß bei der Erstellung des Programms die Aufgabe in sehr kleine Schritte gegliedert werden. Programmiersprachen, die ähnlich komplex wie menschliche Sprachen sind, können die Programmiergeschwindigkeit wesentlich erhöhen. Bekannte "höhere" Programmiersprachen sind beispielsweise FORTRAN und COBOL.

Auch die von diesem Computer verwendete Programmiersprache BASIC ist eine höhere Programmiersprache. Programme, die in höheren Programmiersprachen geschrieben sind, müssen vor der Ausführung in der CPU in Maschinensprache übersetzt werden. Das Programm, daß diese Übersetzung durchführt, wird Compiler oder Interpretierer genannt. Selbst heute, wo es viele höhere Programmiersprachen gibt, ist es wichtig, die Assemblersprache zu kennen, weil damit effektive Routinen geschrieben werden können und der Speicher wirkungsvoll genutzt werden kann.

4 Ein hypothetischer Computer und Simulator

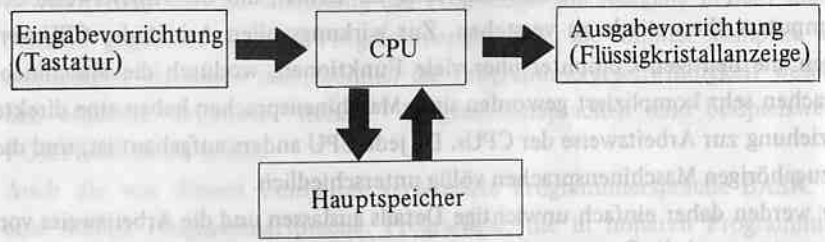
Wie im vorhergehenden Abschnitt erklärt, ist es wichtig, Programmieren in Maschinensprache oder Assemblersprache zu lernen, um die Arbeitsweise des Computers theoretisch zu verstehen. Zur wirkungsvollen Arbeit der CPU verfügen die heutigen Computer über viele Funktionen, wodurch die Maschinensprachen sehr kompliziert geworden sind. Maschinensprachen haben eine direkte Beziehung zur Arbeitsweise der CPUs. Da jede CPU anders aufgebaut ist, sind die dazugehörigen Maschinensprachen völlig unterschiedlich.

Wir werden daher einfach unwichtige Details auslassen und die Arbeitsweise von Computern und die Programmierung in Maschinensprache mittels eines hypothetischen Computers studieren, dessen Maschinensprache nur für die grundsätzlichen Aufgaben konzipiert ist.

Wenn Computer A imstande ist, die Arbeitsweise von Computer B zu imitieren, wird gesagt, daß Computer A ein Simulator von Computer B ist. Und dieser Computer verfügt über eine Simulator-Funktion für unseren hypothetischen Computer. Außerdem hat er eine Assemblerfunktion zum Übersetzen von Programmen, die für den hypothetischen Computer in Assemblersprache geschrieben wurden, in Maschinensprache. Da dieser Computer im folgenden als Simulator des hypothetischen Computers verwendet wird und nicht als BASIC-Computer, werden wir ihn ab jetzt einfach "**Simulator**" nennen. Die Sprache für die Programme des Simulators wird daher als "**Assemblersprache**" bezeichnet.

5 Simulator-Hardware

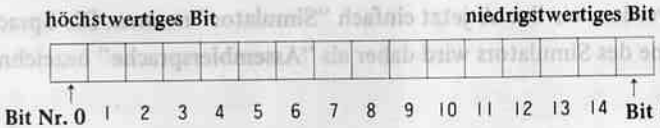
Die folgende Abbildung zeigt den grundsätzlichen Aufbau des Simulators.



*Ein Drucker (Sonderzubehör) kann außerdem als Ausgabevorrichtung und ein Cassettenrecorder als Hilfsspeicher eingesetzt werden.

- **Hauptspeicher**

Der Hauptspeicher des Simulators ist in 16-Bit-Zellen aufgeteilt, wobei jede Zelle als "**Wort**" bezeichnet wird und mit einer "**Adresse**" versehen ist. Die Bits, die ein Wort bilden, sind von links (höchstwertiges Bit) nach rechts (niedrigstwertiges Bit) in numerischer Reihenfolge von 0 bis 15 angeordnet.

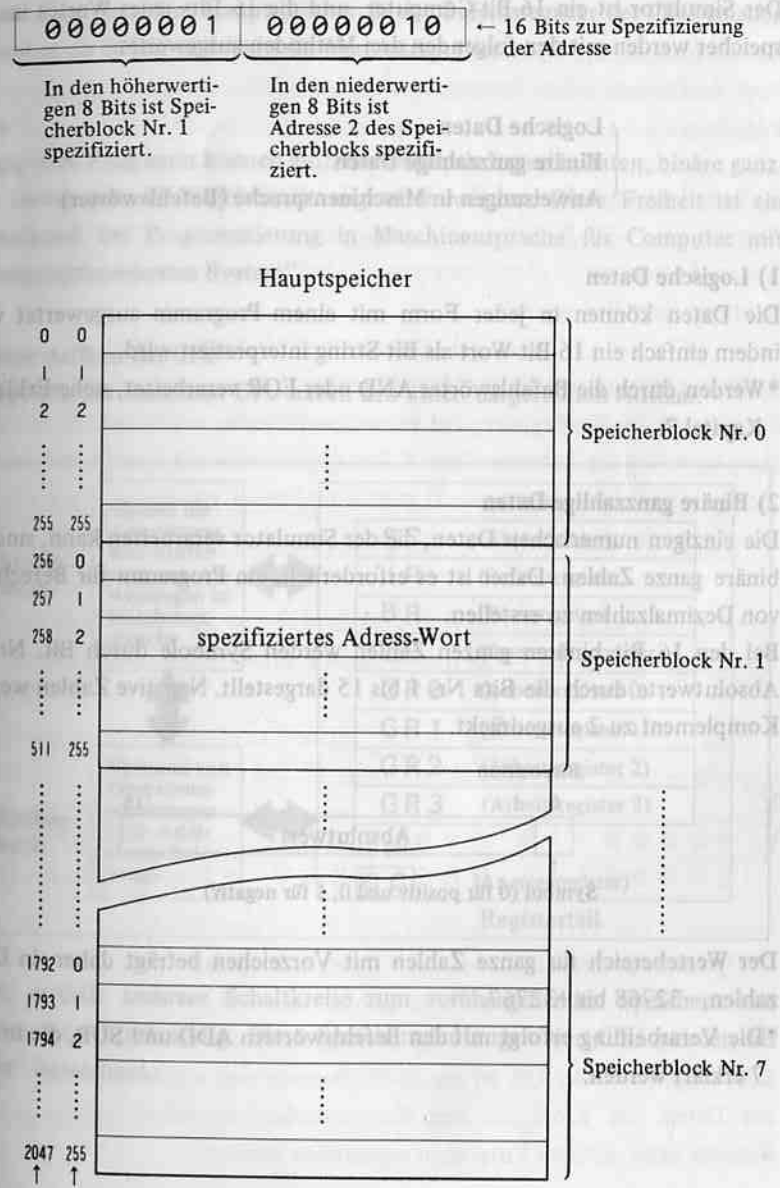


Insgesamt 2048 (2^{11}) Wörter können im Hauptspeicher gespeichert werden, diese Wörter haben die Adressen 0 bis 2047. Obwohl zur Spezifizierung von Adressen 11 Bits ausreichen, werden sie mit 16 Bits spezifiziert.

Die 256 aufeinanderfolgenden Worte, beginnend von der Adresse (einschließlich 0) des ganzzahligen Mehrfachens von 256, werden jeweils als ein Speicherblock bezeichnet. Die 256 Worte, die von 256 × Adresse N beginnen (N = 0 ~ 7), werden als Speicherblock N bezeichnet. Daher stehen bis zu acht Speicherblöcke (Speicherblock Nummer 0 ~ Nummer 7) zur Verfügung. Die höherwertigen 8 Bits der 16 Bits, die eine Adresse spezifizieren, bezeichnen den Speicherblock (0 bis 7) und die niederwertigen 8 Bits die Adresse des Worts im Speicherblock.

*Die Anzahl der Speicherblöcke (1 ~ 8), die bereitgestellt werden kann, ist von der Anzahl der verbleibenden Bytes abhängig.

Spezifiziert Adresse 258 der Gesamtadressen



• **Auswertung der Wörter**

Der Simulator ist ein 16-Bit-Computer, und die 16 Bits jedes Wortes im Hauptspeicher werden mit den folgenden drei Methoden ausgewertet.

Logische Daten
Binäre ganzzahlige Daten
Anweisungen in Maschinensprache (Befehlswörter)

1) Logische Daten

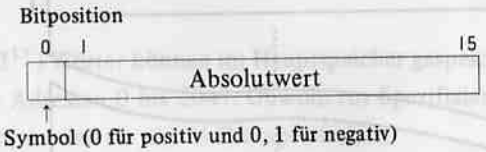
Die Daten können in jeder Form mit einem Programm ausgewertet werden, indem einfach ein 16-Bit-Wort als Bit-String interpretiert wird.

*Werden durch die Befehlswörter AND oder EOR verarbeitet, siehe Erklärung in Kapitel 7.

2) Binäre ganzzahlige Daten

Die einzigen numerischen Daten, die der Simulator verarbeiten kann, sind 16-Bit binäre ganze Zahlen. Daher ist es erforderlich, ein Programm für Berechnungen von Dezimalzahlen zu erstellen.

Bei den 16 Bit binären ganzen Zahlen werden Symbole durch Bit. Nr. 0 und Absolutwerte durch die Bits Nr. 1 bis 15 dargestellt. Negative Zahlen werden als Komplement zu 2 ausgedrückt.



Der Wertebereich für ganze Zahlen mit Vorzeichen beträgt daher, in Dezimalzahlen, -32768 bis +32767.

*Die Verarbeitung erfolgt mit den Befehlswörtern ADD und SUB, die in Kapitel 7 erklärt werden.

3) Anweisungen in Maschinensprache (Befehlswörter)

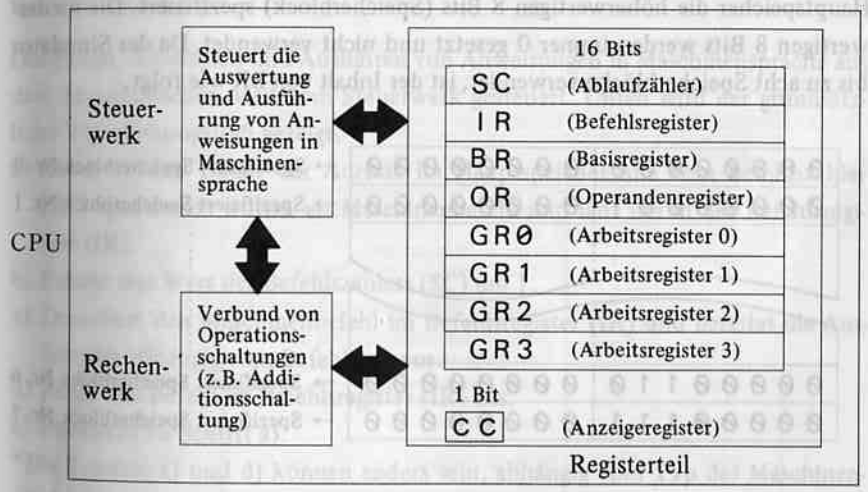
Wenn ein Wort aus dem Hauptspeicher in das Befehlsregister der CPU geladen wird, wird es als Befehl und nicht als Wort interpretiert.

Hinweis:

Abhängig vom Programm können einzelne Daten als logische Daten, binäre ganzzahlige Daten oder als Befehlswort angesehen werden. Diese Freiheit ist ein Hauptmerkmal bei Programmierung in Maschinensprache für Computer mit "**speicherprogrammiertem System**".

• **Interner Aufbau der CPU**

Die Hauptteile der Simulator-CPU haben den unten dargestellten Aufbau.

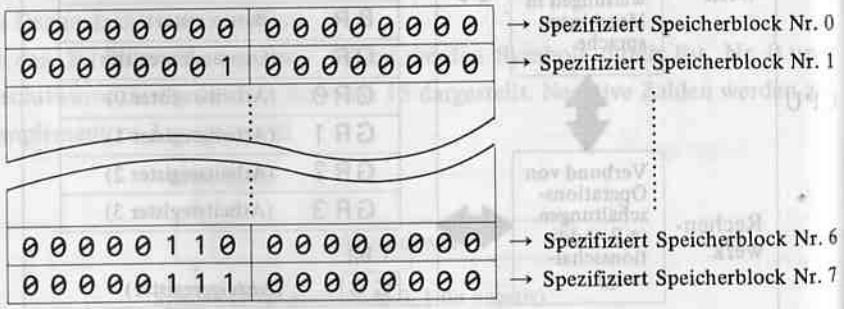


Die CPU enthält mehrere Schaltkreise zum vorübergehenden Speichern von Daten bei der Verarbeitung. Diese Speicherschaltungen der CPU werden als "**Register**" bezeichnet.

1) Ablaufzähler (Symbol SC = Sequence Counter)
Dies ist ein 16-Bit-Register zum Speichern der Adresse im Hauptspeicher, wo die als nächste auszuführende Anweisung in Maschinensprache gespeichert ist. Vor Ausführung eines Programms wird im SC eine Ausführung-Startadresse initialisiert.

2) Befehlsregister (Symbol IR = Instruction Register)
Dies ist ein 16-Bit-Register zur Speicherung der jeweils in Ausgeführung befindlichen Anweisung in Maschinensprache. Auf dieses Register kann nicht im Programm zugegriffen werden.

3) Basisregister (Symbol BR = Base Register)
Dies ist ein 16-Bit-Register, das beim Spezifizieren von Adressen (16 Bits) im Hauptspeicher die höherwertigen 8 Bits (Speicherblock) spezifiziert. Die niederwertigen 8 Bits werden immer 0 gesetzt und nicht verwendet. Da der Simulator bis zu acht Speicherblöcke verwendet, ist der Inhalt von BR wie folgt.



4) Operandenregister (Symbol OR = Operand Register)
Berechnungen wie Addition und Subtraktion werden mit der Operation des Inhalts (Operand) der Wörter im Hauptspeicher für den Inhalt des Arbeitsregisters ausgeführt. Der OR ist ein 16-Bit-Register zum vorübergehenden Speichern der Daten, die dabei aus dem Hauptspeicher abgerufen werden. Auf dieses Register kann nicht im Programm zugegriffen werden.

5) Arbeitsregister (Symbol GR = General Register)
Dies ist ein 16-Bit-Register, das als Indexregister (Symbol **XR**, Erklärung siehe später) zur Adressenmodifikation verwendet werden kann. Hauptsächlich wird es als Operationsregister für arithmetische Operationen wie Addition und Subtraktion verwendet. Es wird als Arbeitsregister (oder als Mehrzweckregister) wegen seiner vielfältigen Verwendungsmöglichkeiten bezeichnet und besteht aus den vier Registern GR0, GR1, GR2 und GR3. Alle vier Register können für arithmetische Operationen benutzt werden, aber nur GR0 kann als Indexregister eingesetzt werden.

6) Anzeigeregister (Symbol CC = Condition Code Register)
Dies ist ein 1-Bit-Register, das Code-Bits (Bit Nr. 0) der Ergebnisse bei Addition oder Subtraktion speichert. Der Inhalt bleibt bis zur nächsten Addition bzw. Subtraktion erhalten.

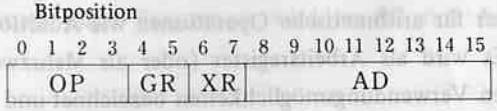
Das Holen, Auswerten und Ausführen von Anweisungen in Maschinensprache aus dem Hauptspeicher wird vom Steuerwerk gesteuert. Unten wird der grundsätzliche Verarbeitungsfluß gezeigt.

a) Bewertet den Inhalt der Adresse im Hauptspeicher, die vom Befehlszähler (SC) spezifiziert wurde, als Maschinenbefehl und lädt ihn in das Befehlsregister (IR).
b) Erhöht den Wert des Befehlszählers (SC) um 1.
c) Decodiert den Maschinenbefehl im Befehlsregister (IR) und bereitet die Ausführung abhängig vom Befehlstyp vor.
d) Führt den Befehl im Befehlsregister (IR) aus.
e) Rückkehr zu Schritt a).

*Die Schritte c) und d) können anders sein, abhängig vom Typ des Maschinenbefehls.

- **Aufbau von Maschinenbefehlen**

Jede Anweisung in Maschinensprache (Befehlswort) hat den folgenden 16-Bit-Aufbau.



Jeder Abschnitt des internen 16-Bit-Aufbaus hat eine semantische Belegung, die "**Feld**" genannt wird. Wie aus der obigen Abbildung ersichtlich, ist ein Befehlswort in vier Felder aufgeteilt, die die folgenden Bezeichnungen und Aufgaben haben.

1) OP-Feld (Befehlscode-Feld)

Spezifiziert einen Befehlscode (Befehlstyp). Befehlscodes werden mit 4 Bits ausgedrückt. Dieser Simulator verfügt über die folgenden 14 Befehlscodes. (Für Einzelheiten siehe Kapitel 7.) Jeder Befehlscode wird durch ein mnemonisches Symbol bezeichnet.

Binäre Anzeige	Hexa-dezimal	Befehl	Mnemonisches Symbol
0000	0	Halt and jump (Halt und Sprung)	HJ
0001	1	Jump if GR is not zero (Sprung wenn GR nicht Null ist)	JNZ
0010	2	Jump on condition (Bedingter Sprung)	JC
0011	3	Jump to subroutine (Sprung zu Unterprogramm)	JSR
0100	4	Shift (Verschiebung)	SFT
0101	5	Read (Lesen)	READ
0110	6	Write (Schreiben)	WRITE
1000	8	Load address immediate (Adresse unmittelbar laden)	LAI
1010	A	Add (Addieren)	ADD
1011	B	Substract (Subtrahieren)	SUB
1100	C	Load (Laden)	LD
1101	D	Store (Speichern)	ST
1110	E	And (Und)	AND
1111	F	Exclusive or (Exklusives Oder)	EOR

2) GR-Feld (Arbeitsregister-Feld)

Spezifiziert die Nummer des Arbeitsregisters, das Objekt des Befehls ist. Im Fall eines "Halt und Sprung"-Befehls wird der Inhalt des GR-Feldes jedoch ignoriert, und eine Bedingung kommt im Fall eines "Bedingten Sprungs" zur Anwendung. (Für Einzelheiten siehe Kapitel 7.)

3) XR-Feld (Indexregister-Feld)

Spezifiziert die Nummer des Indexregisters (GR1, GR2, GR3) zur Modifizierung einer Adresse. Wenn eine Adresse nicht modifiziert wird, wird 0 für das XR-Feld spezifiziert. Dieses Feld dient außerdem zur Kennzeichnung der Verschiebungsrichtung bei Verschiebungs-Befehlen. (Für Einzelheiten siehe Kapitel 7.)

4) AD-Feld (Adress-Feld)

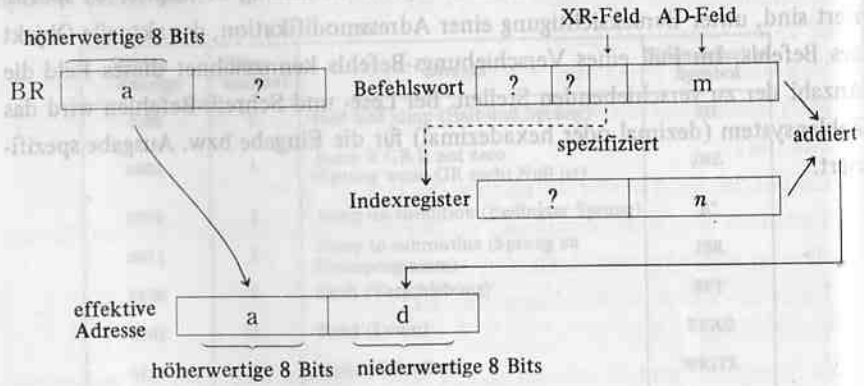
Spezifiziert die niederwertigen 8 Bits der Adresse im Hauptspeicher, die das Objekt des Befehls ist. Jedoch sind die Daten des Worts, die durch das Basisregister (BR) und die "effektive Adresse" (siehe Erklärung in Kapitel 6) spezifiziert sind, unter Berücksichtigung einer Adressmodifikation, das aktuelle Objekt des Befehls. Im Fall eines Verschiebungs-Befehls kennzeichnet dieses Feld die Anzahl der zu verschiebenden Stellen. Bei Lese- und Schreib-Befehlen wird das Zahlensystem (dezimal oder hexadezimal) für die Eingabe bzw. Ausgabe spezifiziert.

6 Adressenmodifikation und effektive Adresse

Die CPU interpretiert das Befehlswort im Hauptspeicher und wechselt die Daten mit Worten der spezifizierten Adresse nach diesen Befehlen aus und steuert auch den Sprung zur spezifizierten Adresse. Im Fall der Objektadresse in diesem Simulator wird die Operation jedoch durch die **"effektive Adresse"** ausgeführt, die mittels einem festen Verfahren erhalten wird.

Die effektive Adresse eines Befehlsworts besteht aus 16 Bits. Von diesen 16 Bits verwenden die höherwertigen 8 Bits immer den Inhalt der höherwertigen 8 Bits des Basisregisters BR. Die niederwertigen 8 Bits der effektiven Adresse sind als Summe des Wertes des AD-Feldes (8 Bits) des Befehlswortes und des Wertes der niederwertigen 8 Bits des im XR-Feld spezifizierten Indexregisters (GR1, GR2, GR3) gesetzt. Wenn die Summe 256 überschreitet, wird der Rest nach Teilung durch 256 verwendet. Wenn der Wert des XR-Feldes 0 ist, wird das AD-Feld die niederwertigen 8 Bits der effektiven Adresse.

Im folgenden Diagramm wird das Verfahren zur Berechnung der effektiven Adresse dargestellt.



Es folgen mehrere Beispiele zur Berechnung der effektiven Adresse.

Beispiel 1:

Der Inhalt von GR1 ist 37846 (dezimal). Die niederwertigen 8 Bits beinhalten 214, was dem Rest nach Teilung durch 256 entspricht.
(37846 ÷ 256 = 147 mit einem Rest von 214)

Wir wollen die effektive Adresse dieses ADD-Befehls mit den Inhalten der betroffenen Register wie in der Abbildung oben berechnen. BR wird Speicherblock Nr. 1, weil sein Inhalt 256 (dezimal) ist. Die 214 der niederwertigen 8 Bits von GR1 werden zur 75 (dezimal) im AD-Feld des Befehlswortes addiert.

214 + 75 = 289 = 256 + 33

Die Summe wird durch 256 dividiert und der Rest von 33 (dezimal) den niederwertigen 8 Bits der effektiven Adresse zugeordnet. Damit ist Adresse 33 in Speicherblock Nr. 1 die effektive Adresse. Die Adresse in Bezug auf die Gesamtadressen ist 289 (256 + 33).

Beispiel 2:

Inhalt des AD-Feldes des Befehlswortes = 43 (dezimal)
Inhalt des XR-Feldes des Befehlswortes = 2 (dezimal)
Inhalt des Indexregisters GR2 = 3 (dezimal)
Inhalt des Basisregisters BR = 0 (dezimal)

Dies bedeutet, daß die effektive Adresse dieses Befehlswortes 46 (= 43 + 3) ist. Da in diesem Beispiel BR gleich 0 ist, ist es Speicherblock Nr. 0, und die Adresse in diesem Block stimmt mit der Adresse in Bezug auf die Gesamtadressen überein.

Beispiel 3:

Inhalt des AD-Feldes des Befehlswortes = 201 (dezimal)
Inhalt des XR-Feldes des Befehlswortes = 3 (dezimal)
Inhalt des Indexregisters GR3 = 739 (dezimal)
Inhalt des Basisregisters BR = 0 (dezimal)

Wir wollen die effektive Adresse des Befehlswortes in diesem Beispiel berechnen. Da der Inhalt des Indexregisters GR3 größer als 256 ist, wird durch 256 geteilt und der Rest zum Holen der niederwertigen 8 Bits verwendet. Die Berechnung ergibt 739 ÷ 256 = 2 mit einem Rest von 227. Dieser Wert wird zu 201 im AD-Feld addiert, was 428 ergibt. Auch dieser Wert überschreitet 256 und wird wieder durch 256 dividiert, was 428 ÷ 256 = 1 mit einem Rest von 172 ergibt. Das ergibt für die niederwertigen 8 Bits der effektiven Adresse einen Wert von 172 (dezimal). Da in diesem Beispiel BR gleich 0 ist, ist es Speicherblock Nr. 0, und die effektive Adresse in Bezug auf die Gesamtadressen ist 172.

Ändern der aktuellen Wirkungen der Adressen-Spezifikation durch das AD-Feld bezüglich des Inhalts des im XR-Feld spezifizierten Indexregisters wird als "**Adressenmodifikation**" bezeichnet. Wenn im XR-Feld eine Nummer 1, 2 oder 3 (dezimal) spezifiziert ist, wird eine Adressenmodifikation durchgeführt. Wenn dagegen der Wert 0 für das XR-Feld spezifiziert ist, bedeutet das, daß "keine Adressenmodifikation durchgeführt wird".

Beispiel 4:

Inhalt des AD-Feldes des Befehlswortes = 38 (dezimal)
Inhalt des XR-Feldes des Befehlswortes = 0 (dezimal)
Inhalt des Basisregisters BR = 0 (dezimal)

In diesem Fall wird keine Adressenmodifikation durchgeführt, und die effektive Adresse ist 38.

7 Anweisungen des Simulators in Maschinensprache

Als Anweisungen des Simulators in Maschinensprache sind 14 Arten Befehlswörter verfügbar. Nach den Funktionen können sie in die folgenden vier Gruppen aufgeteilt werden:

Operationsbefehle
Transferbefehle
Ablaufsteuerbefehle
Eingabe/Ausgabe-Befehle

1) Operationsbefehle

Gibt Anweisungen für arithmetische Operationen wie Addition und Subtraktion und für logische Operationen wie logisches Produkt.

2) Transferbefehle

Gibt Anweisungen für Transfer von Informationen zwischen dem Arbeitsregister und dem Hauptspeicher.

3) Ablaufsteuerbefehle

Steuert die Ablauf-Reihenfolge der Maschinenbefehle. Ein Befehl in Adresse (n + 1) wird normalerweise nach dem Befehl in Adresse n ausgeführt. Dieser Ablauf kann durch einen Ablaufsteuerbefehl geändert werden. Dieser Maschinenbefehl entspricht GOTO, GOSUB und IF ~ THEN in BASIC.

4) Eingabe/Ausgabe- (I/O) Befehle

Gibt Anweisungen für die Eingabe von numerischen Werten von der Eingabevorrichtung (Tastatur) und für die Ausgabe von numerischen Werten zur Ausgabevorrichtung (Flüssigkristallanzeige).

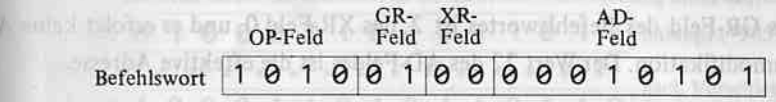
Klassifikation	Bezeichnung	Mnemonischer Code	Befehlscode (hexadezimal	Befehlscode (binär)
Operation	Add (Addition)	ADD	A	1010
	Substract (Subtraktion)	SUB	B	1011
	Shift (Verschiebung)	SFT	4	0100
	And (Und)	AND	E	1110
	Exclusive or (Exklusives Oder)	EOR	F	1111
Transfer	Load (Laden)	LD	C	1100
	Store (Speichern)	ST	D	1101
	Load address immediate (Adresse unmittelbar laden)	LAI	8	1000
Ablaufsteuerung	Jump if GR is not zero (Sprung wenn GR nicht Null)	JNZ	1	0001
	Jump on condition (Bedingter Sprung)	JC	2	0010
	Jump to subroutine (Sprung zu Unterprogramm)	JSR	3	0011
	Halt and Jump (Halt und Sprung)	HJ	0	0000
Eingabe/ Ausgabe (I/O)	Read (Lesen)	READ	5	0101
	Write (Schreiben)	WRITE	6	0110

Im folgenden werden die Funktionen der einzelnen Befehlswörter erklärt.

- **ADD-Befehl (binärer Befehlscode 1010)**

Addiert den Inhalt des im GR-Feld spezifizierten Arbeitsregisters zum Inhalt des durch die effektive Adresse spezifizierten Wortes im Hauptspeicher. Diese werden als binäre ganze Zahlen mit 16-Bit-Code angesehen. Dann setzt dieser Befehl das Ergebnis in das Arbeitsregister. In diesem Fall wird ein Überlauf ignoriert. Dieser Befehl setzt außerdem das Code-Bit (Bit Nr. 0) des Additionsergebnisses in das Anzeigeregister (CC). Der Inhalt des Wortes bleibt unverändert.

Beispiel:
Der folgende ADD-Befehl soll gegeben sein.
(in diesem Fall BR = 0)



Welche Inhalte haben GR1 und CC nach Ausführung dieses ADD-Befehls, wenn der Inhalt von GR1 -124 und der des Wortes in Adresse 21 +55 ist?

Lösung:
Das GR-Feld des Befehlswortes ist 1, das XR-Feld 0, und es erfolgt keine Adressenmodifikation. Daher wird der Wert 21 des AD-Feldes die effektive Adresse.

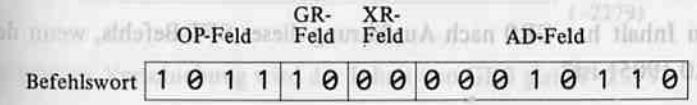
- 124 + 55 = - 69
GR1 Wort von Adresse 21

Das Ergebnis -69 wird in GR1 gesetzt. Da dieser Wert negativ ist und das Code-Bit 1 ist, wird in CC 1 gesetzt.

- **SUB-Befehl (binärer Befehlscode 1011)**

Subtrahiert den Inhalt des durch die effektive Adresse spezifizierten Wortes im Hauptspeicher vom Inhalt des im GR-Feld spezifizierten Arbeitsregisters. Diese werden als binäre ganze Zahlen mit 16-Bit-Code angesehen. Dann setzt dieser Befehl das Ergebnis in das Arbeitsregister. In diesem Fall wird ein Überlauf ignoriert. Code-Bits (Bit Nr. 0) des Subtraktionsergebnisses werden in das Anzeigeregister (CC) gesetzt. Der Inhalt des Wortes bleibt unverändert.

Beispiel:
Der folgende SUB-Befehl soll gegeben sein. (BR = 0)



Welche Inhalte haben GR2 und CC nach Ausführung dieses SUB-Befehls, wenn der Inhalt von GR2 -23456 und der des Wortes in Adresse 22 -32100 ist?

Lösung:
Das GR-Feld des Befehlswortes ist 2, das XR-Feld 0, und es erfolgt keine Adressenmodifikation. Der Wert 22 des AD-Feldes ist die effektive Adresse.

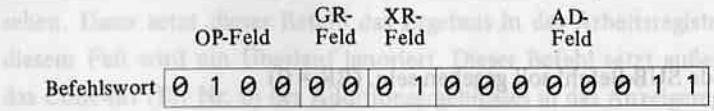
$$\underbrace{(-23456)}_{\text{GR2}} - \underbrace{(-32100)}_{\text{Wort von Adresse 22}} = 8644$$

Das Ergebnis 8644 wird in GR2 gesetzt. In CC wird 0 gesetzt, weil das Ergebnis positiv und das Code-Bit 0 ist.

- **SFT-Befehl (binärer Befehlscode 0100)**

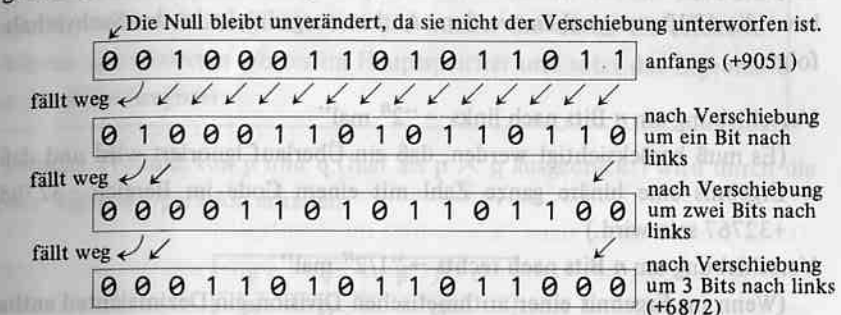
Verschiebt den Inhalt (die Stellen) des im GR-Feld spezifizierten Arbeitsregisters mit Ausnahme des Code-Bits (Bit Nr. 0) um die im AD-Feld spezifizierte Anzahl Bits nach rechts oder links. Die Verschiebungsrichtung wird durch das XR-Feld spezifiziert und ist nach rechts, wenn der Wert des XR-Feldes gleich 0 ist, und nach links, wenn der Wert des XR-Feldes gleich 1 ist. Bei Verschiebung nach links werden in die leeren Bit-Positionen 0 eingefügt, bei Verschiebungen nach rechts wird ein Bit gleich dem Code-Bit eingesetzt. Eine effektive Adresse gibt es nicht.

Beispiel:
Der folgende SFT-Befehl soll gegeben sein.



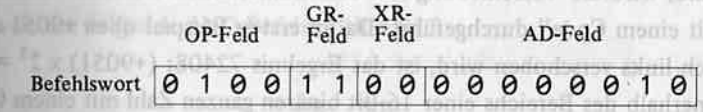
Welchen Inhalt hat GR0 nach Ausführung dieses SFT-Befehls, wenn der Inhalt von GR0 +9051 ist?

Lösung:
Da der Wert des XR-Feldes 1 und der des AD-Feldes 3 ist, wird eine Verschiebung um 3 Bits nach links durchgeführt.

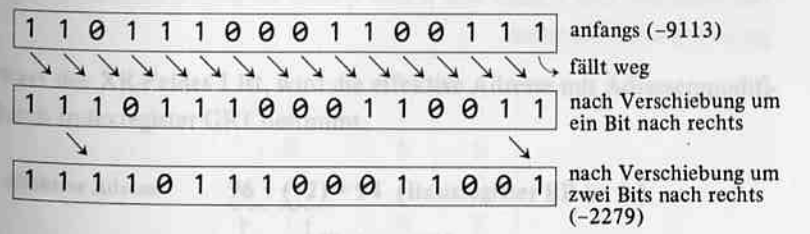


Als Ergebnis der Verschiebung wird der Inhalt von GR0 gleich +6872.

Beispiel:
Welchen Inhalt hat GR3 nach Ausführung des folgenden SFT-Befehls, wenn der Anfangswert -9113 ist?



Lösung:
Das GR-Feld des Befehlswortes ist 3 und das Objekt der Verschiebung ist der Inhalt von GR3. Da der Wert des XR-Feldes 0 und der des AD-Feldes 2 ist, wird eine Verschiebung um zwei Bits nach rechts durchgeführt.
Der Inhalt des Code-Bits (Bit Nr. 0) wird in die leeren Positionen eingesetzt.



Als Ergebnis der Verschiebung wird der Inhalt von GR3 gleich -2279.

Hinweis:
Bei Verschiebungen nach rechts oder links um eine Stellenanzahl, die dezimal ausgedrückt wird, kann die Verschiebung der Bits um binäre Stellen als doppelt bzw. die Hälfte angesehen werden. Anders ausgedrückt ist der Sachverhalt wie folgt:

Verschiebung um n Bits nach links → "2^n mal"
(Es muß berücksichtigt werden, daß ein Überlauf ignoriert wird und daß das Ergebnis eine binäre ganze Zahl mit einem Code im Bereich -32768 bis +32767 sein wird.)

Verschiebung um n Bits nach rechts → "$1/2^n$ mal"
(Wenn im Ergebnis einer arithmetischen Division ein Dezimalanteil enthalten ist, wird dieser als codierte Zahl angesehen, und die Lösung ist die größte ganze Zahl, die nicht größer als das Ergebnis ist.)

Da die Bits Nr. 1 bis Nr. 15, also ohne das Code-Bit, das Objekt der Verschiebung sind, wird die Verschiebung mit dem numerischen Wert als "binäre ganze Zahl mit einem Code" durchgeführt. Da im ersten Beispiel oben +9051 um drei Bits nach links verschoben wird, ist das Ergebnis 72408: $(+9051) \times 2^3 = +9051 \times 8$. Innerhalb des Bereichs einer 16-Bit binären ganzen Zahl mit einem Code ist das Ergebnis +6872 (72408 − 65536).
Im zweiten Beispiel wird -9113 um zwei Bits nach rechts verschoben, daher ist das Ergebnis -2278,25: $(-9113) \times 1/2^2 = (-9113) \times 1/4$. Die größte ganze Zahl, die diesen Wert nicht überschreitet, ist -2279 und damit das Ergebnis.

- **AND-Befehl (binärer Befehlscode 1110)**

Berechnet das logische Produkt pro Bit des Inhalts des im GR-Feld spezifizierten Arbeitsregisters und des Inhalts des durch die effektive Adresse spezifizierten Wortes im Hauptspeicher und setzt das Ergebnis in das Arbeitsregister.

Das logische Produkt von p und q (hier als $p \wedge q$ ausgedrückt) wird durch die folgende logische Operation erhalten.

p	q	$p \wedge q$
0	0	0
0	1	0
1	0	0
1	1	1

Beispiel:
Gegeben ist der folgende AND-Befehl. (BR = 0)

	OP-Feld				GR-Feld		XR-Feld		AD-Feld							
Befehlswort	1	1	1	0	0	0	0	1	0	1	1	0	0	0	0	0

Welchen Inhalt hat GR0 nach Ausführung dieses AND-Befehls, wenn der Inhalt von GR0 in hexadezimaler Darstellung ABCD, der Inhalt von GR1 -2 und der Inhalt des Wortes in Adresse 94 in hexadezimaler Darstellung 00FF ist?

Lösung:
Da der Wert des XR-Feldes 1 ist, wird die effektive Adresse mit Adressenmodifikation durch Indexregister GR1 bestimmt.

effektive Adresse **96 + (-2) = 94** (Basisregister BR ist 0.)
↑ Wert des AD-Feldes; ↑ Wert von GR1

Dann wird das bitweise logische Produkt des Arbeitsregisters GR0 und des Worts in Adresse 94 des Hauptspeichers berechnet.

GR0 (hexadezimal ABCD)	1	0	1	0	1	0	1	1	1	1	0	0	1	1	0	1
Adresse 94 (hexadezimal 00FF)	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
↓ logisches Produkt																
	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	1

Der Inhalt von GR0 ist dann 00CD in hexadezimaler Darstellung.
Durch Bereitstellung eines Datenwortes im Hauptspeicher mit dem AND-Befehl und Berechnung des logischen Produktes dieser Daten und des Arbeitsregisters auf diese Weise ist es möglich, nur die Bitpositionen, in denen 1 geschrieben ist, in das Arbeitsregister zu holen und die anderen Bitpositionen zu maskieren. Dies wird als "Maskierungs-Operation" bezeichnet, und die verwendeten Daten sind "Masken-Daten". Im obigen Beispiel sind die höherwertigen acht Bits von GR0 mit den Maskendaten 00FF maskiert, so daß nur die niederwertigen acht Bits geholt werden.

- **EOR-Befehl (binärer Befehlscode 1111)**

> Berechnet das exklusive OR (exklusive logische Summe) pro Bit des Inhalts des vom GR-Feld spezifizierten Arbeitsregisters und des Inhalts des durch die effektive Adresse spezifizierten Wortes im Hauptspeicher und setzt das Ergebnis in das Arbeitsregister.

Das exklusive OR (Oder) von p und q (hier als $p \oplus q$ ausgedrückt) ist die folgende logische Operation.

p	q	p ⊕ q
0	0	0
0	1	1
1	0	1
1	1	0

Beispiel:
Gegeben sei der folgende EOR-Befehl (BR = 0)

	OP-Feld				GR-Feld		XR-Feld		AD-Feld							
Befehlswort	1	1	1	1	0	0	0	0	0	1	1	0	0	0	0	0

Welchen Inhalt hat GR0 nach Ausführung des EOR-Befehls, wenn GR0 in hexadezimaler Darstellung A50F und der Inhalt des Wortes in Adresse 96 FFFF in hexadezimaler Darstellung ist?

Lösung:
Der Wert des XR-Feldes ist 0, und es erfolgt keine Adressenmodifikation. Daher ist 96 die effektive Adresse. Dann wird das exklusive OR pro Bit von GR0 und des Wortes in Adresse 96 berechnet.

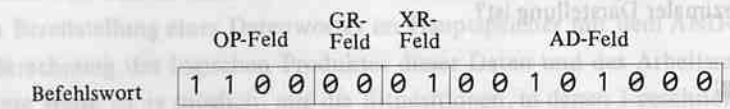
GR0 (hexadezimal A50F)	1	0	1	0	0	1	0	1	0	0	0	0	1	1	1	1
Adresse 96 (hexadezimal FFFF)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
↓ exklusives OR																
	0	1	0	1	1	0	1	0	1	1	1	1	0	0	0	0

Dann ist der Inhalt von GR0 hexadezimal 5AF0.
0 und 1 in jedem Bit von GR0 werden umgedreht. (Für Einzelheiten siehe 11-3 von Kapitel 11.)

- **LD-Befehl (binärer Befehlscode 1100)**

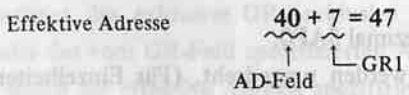
Transferiert des Inhalts des in der effektiven Adresse spezifizierten Wortes im Hauptspeicher in das im GR-Feld spezifizierte Arbeitsregister. Der Inhalt des Wortes bleibt unverändert.

Beispiel:
Gegeben sei der folgende LD-Befehl. (BR = 0)



Welchen Inhalt hat GR0 nach Ausführung dieses LD-Befehls, wenn der Inhalt von GR0 29, der Inhalt von GR1 7 und der Inhalt des Wortes in Adresse 47 112 ist?

Lösung:
Da der Wert des XR-Feldes 1 ist, wird die effektive Adresse durch Adressenmodifikation mit Indexregister GR1 bestimmt.

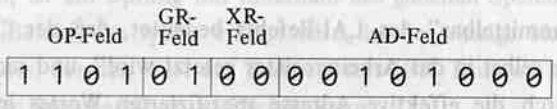


Der Inhalt des Wortes in Adresse 47, 112, wird daher zu GR0 transferiert, und der Inhalt von GR0 wird 112. Der Inhalt des Wortes in Adresse 47 bleibt 112.

- **ST-Befehl (binärer Befehlscode 1101)**

Speichert den Inhalt des im GR-Feld spezifizierten Arbeitsregisters in das durch die effektive Adresse spezifizierte Wort im Hauptspeicher. Der Inhalt des Arbeitsregisters bleibt unverändert.

Beispiel:
Gegeben sei der folgende ST-Befehl. (BR = 0)



Welchen Inhalt hat das Wort in Adresse 40 nach Ausführung dieses ST-Befehls, wenn der Inhalt von GR1 25 und der Inhalt des Wortes in Adresse 40 516 ist?

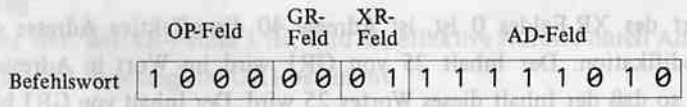
Lösung:
Da der Wert des XR-Feldes 0 ist, ist Adresse 40 die effektive Adresse ohne Adressenmodifikation. Der Inhalt 25 von GR1 wird im Wort in Adresse 40 gespeichert, so daß der Inhalt dieses Wortes 25 wird. Der Inhalt von GR1 bleibt 25.

- **LAI-Befehl (binärer Befehlscode 1000)**

Die niederwertigen 8 Bits der effektiven Adresse werden in die niederwertigen Bits (Bit Nr. 8 bis 15) des im GR-Feld spezifizierten Arbeitsregisters gesetzt, gleichzeitig werden die höherwertigen 8 Bits (Bit Nr. 0 bis 7) alle 0 gesetzt. Dieser Befehl dient dazu, den Inhalt eines spezifizierten Arbeitsregisters zu einem numerischen Wert zwischen 0 und 255 zu machen.

*Das "unmittelbar" des LAI-Befehls bedeutet, daß der "Wert der effektiven Adresse selbst in das Arbeitsregister gesetzt wird", und nicht, daß der "Inhalt des durch die effektive Adresse spezifizierten Wortes im Hauptspeicher in das Arbeitsregister gesetzt wird".

Beispiel:
Gegeben sei der folgende LAI-Befehl. (BR = 0)



Welchen Inhalt hat GR0 nach Ausführung dieses LAI-Befehls, wenn der Inhalt von GR0 84 und der Inhalt von GR1 71 ist?

Lösung:
Da der Wert des XR-Feldes 1 ist, wird die effektive Adresse durch Adressenmodifikation mit Indexregister GR1 bestimmt.

Effektive Adresse $250 + 71 = 321$
(250: AD-Feld; 71: Inhalt von GR1)

Die niederwertigen 8 Bits der effektiven Adresse

1 mit Rest 65: 321 ÷ 256.

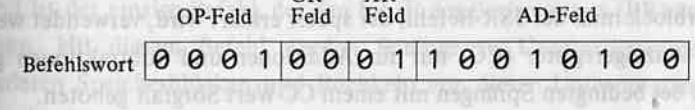
Der Wert 65 in den niederwertigen 8 Bits der effektiven Adresse wird daher in GR0 gesetzt.

- **JNZ-Befehl (binärer Befehlscode 0001)**

Die niederwertigen 8 Bits der effektiven Adresse werden in den Ablaufzähler (SC) gespeichert, und die Ausführung springt zur effektiven Adresse, wenn der Inhalt des im GR-Feld spezifizierten Arbeitsregisters nicht 0 ist. Bei 0 wird der Befehl der nächsten Adresse ausgeführt.

*Da die niederwertigen 8 Bits der effektiven Adresse im Ablaufzähler (SC) gespeichert werden, ist ein Sprung nur innerhalb des gleichen Speicherblocks möglich. Der JSR-Befehl, der später erklärt wird, ist der einzige Maschinenbefehl des Simulators, der den Wert des Basisregisters (BR) ändern und Adressen in einem anderen Speicherblock verarbeiten kann.

Beispiel:
Gegeben sei der folgende Befehl. (BR = 0)



Welche ist die nächste auszuführende Adresse nach Ausführung dieses JNZ-Befehls, wenn der Inhalt von GR0 125 und der Inhalt von GR1 -3 ist?

Lösung:
Da der Wert des XR-Feldes 1 ist, wird die effektive Adresse durch Adressenmodifikation mit Indexregister GR1 bestimmt.

Effektive Adresse $148 + (-3) = 145$
(148: AD-Feld; -3: GR1)

Da der Inhalt von GR0 nicht 0 ist, springt die Ausführung zu Adresse 145, wenn dieser Befehl ausgeführt wird.

- **JC-Befehl (binärer Befehlscode 0010)**

Die folgenden Operationen werden abhängig davon ausgeführt, welcher Inhalt im GR-Feld (Bit Nr. 4 und 5) spezifiziert ist.

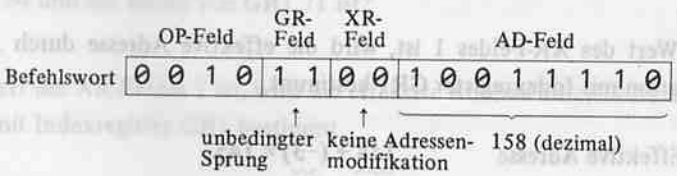
Bit Nr. 4	Bit Nr. 5	Tastenbetätigung
0	0	Geht weiter zum nächsten Befehl.
0	1	Springt zur effektiven Adresse, wenn der Inhalt des Anzeigeregisters (CC) 1 ist.
1	0	Springt zur effektiven Adresse, wenn der Inhalt des Anzeigeregisters (CC) 0 ist.
1	1	Springt unbedingt zur effektiven Adresse.

*Da bei Ausführung eines Sprungs die niederwertigen 8 Bits der effektiven Adresse im Ablaufzähler (SC) gespeichert werden, ist nur ein Sprung zu einer Adresse im gleichen Speicherblock möglich. Zum Sprung in einen anderen Speicherblock muß der JSR-Befehl, der später erklärt wird, verwendet werden.

*Da das Anzeigeregister (CC) nur für Additionen und Subtraktionen gesetzt wird, ist bei bedingten Sprüngen mit einem CC-Wert Sorgfalt geboten.

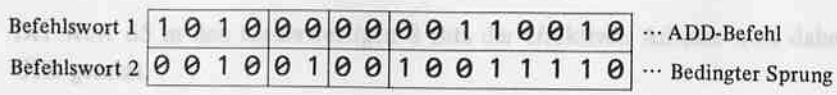
Beispiel:

Der folgende Befehl bewirkt einen unbedingten Sprung zu Adresse 158. (BR = 0)



Beispiel:

Überprüfung, ob der Inhalt von GR0 negativ ist, und wenn negativ, Sprung zur Adresse 158. Wenn die Konstante 0 in Adresse 50 des Hauptspeichers enthalten ist, kann dieser Sprung durch die beiden folgenden Befehle ausgeführt werden.



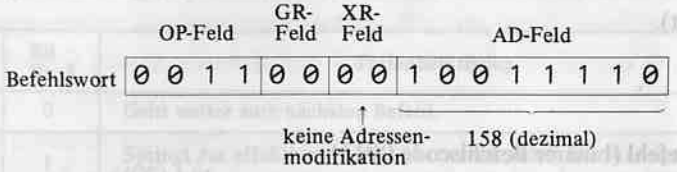
Die Konstante 0 in Adresse 50 wird durch den ADD-Befehl in Befehlswort 1 zu GR0 addiert. Obwohl der Inhalt von GR0 unverändert bleibt, wird das Anzeigeregister (CC) durch das Code-Bit von GR0 gesetzt. Das ermöglicht einen Sprung zu Adresse 158, wenn der Inhalt von CC in Befehlswort 2 gleich 1 ist (d.h. GR0 negativ ist).

- **JSR-Befehl (binärer Befehlscode 0011)**

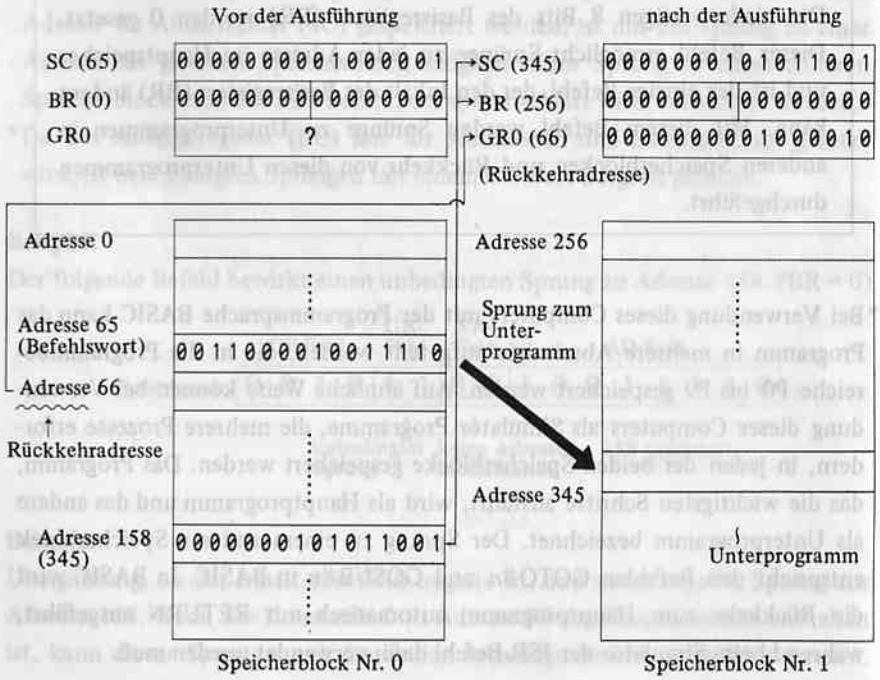
Speichert den Inhalt des durch die effektive Adresse spezifizierten Wortes im Hauptspeicher in den Ablaufzähler (SC) und das Basisregister (BR), nachdem der "Inhalt + 1" des gegenwärtigen Ablaufzählers (SC) in das durch das GR-Feld spezifizierte Arbeitsregister gesetzt wurde. Die niederwertigen 8 Bits des Basisregisters (BR) werden 0 gesetzt. Dieser Befehl ermöglicht Sprünge zu jeder Adresse im Hauptspeicher und ist der einzige Befehl, der den Inhalt des Basisregisters (BR) ändern kann. Mit diesem Befehl werden Sprünge zu Unterprogrammen in anderen Speicherblöcken und Rückkehr von diesen Unterprogrammen durchgeführt.

*Bei Verwendung dieses Computers mit der Programmsprache BASIC kann das Programm in mehrere Abschnitte aufgeteilt werden, die in die Programmbereiche P0 bis P9 gespeichert werden. Auf ähnliche Weise können bei Verwendung dieses Computers als Simulator Programme, die mehrere Prozesse erfordern, in jeden der beiden Speicherblöcke gespeichert werden. Das Programm, das die wichtigsten Schritte ausführt, wird als Hauptprogramm und das andere als Unterprogramm bezeichnet. Der Sprung zu einem anderen Speicherblock entspricht den Befehlen GOTO#*n* und GOSUB#*n* in BASIC. In BASIC wird die Rückkehr zum Hauptprogramm automatisch mit RETURN ausgeführt, während beim Simulator der JSR-Befehl dafür verwendet werden muß.

Beispiel:
Ausführung des folgenden Befehls in Adresse 65, wenn das Hauptprogramm in Speicherblock Nr. 1 ist und der Inhalt von Adresse 158 gleich 345 ist.



Unten ist die Ausführung eines Sprungs von Adresse 345 in Speicherblock Nr. 1 zu einem Unterprogramm abgebildet. Zur Rückkehr vom Unterprogramm muß der Wert von GR0 verwendet werden, da die Rückkehradresse (Adresse 66) in GR0 gespeichert wird. (Für Einzelheiten siehe Kapitel 11-6).



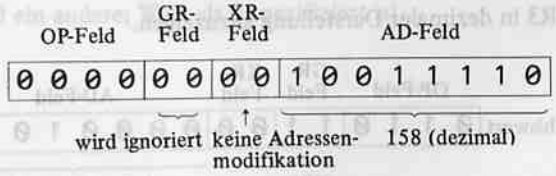
Hinweis:
Da der Hauptspeicher des Simulators aus acht Speicherblöcken (2048 Wörter) besteht, zeigt der Simulator den "Out of address"-Fehler an, wenn die durch den JSR-Befehl spezifizierte effektive Adresse größer als 2047 ist.

- **HJ-Befehl (binärer Befehlscode 0000)**

> Speichert die effektive Adresse in den Ablaufzähler (SC) und stoppt die Ausführung des Maschinensprache-Programms. Wenn ein Neustart-Befehl gegeben wird, startet die Ausführung erneut von der Adresse, die vom Ablaufzähler (SC) angegeben wird. Der Inhalt des GR-Feldes wird ignoriert.

*Wenn der HJ-Befehl ausgeführt wird, stoppt der Simulator die Programmausführung und kehrt zum "Simulator-Menubildschirm" zurück. Die Ausführung wird neugestartet durch "Go" auf dem "Simulator-Menubildschirm". Außer mit dem HJ-Befehl kann die Programmausführung durch Drücken der Taste BRK während der Ausführung des Maschinensprache-Programms gestoppt werden, dann wird der "Simulator-Menublidschirm" angezeigt. Siehe Kapitel 10 "Grundsätzliche Operation des Simulators" für Einzelheiten bezüglich der Ausführung von Maschinensprache-Programmen mit dem Simulator.

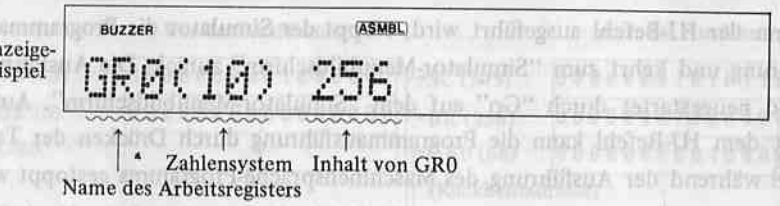
Beispiel:
Bei Ausführung des folgenden HJ-Befehls wird der Inhalt des Ablaufzählers (SC) auf 158 gesetzt und die Programmausführung gestoppt. Wenn ein Neustart-Befehl gegeben wird, wird die Ausführung ab Adresse 158 neugestartet.



- **WRITE-Befehl (binärer Befehlscode 0110)**

Anzeige des Inhalts des im GR-Feld spezifizierten Arbeitsregisters in dem im AD-Feld spezifizierten Zahlensystem. Das spezifizierte Zahlensystem ist entweder dezimal oder hexadezimal. Für das XR-Feld muß immer 0 spezifiziert werden. Es gibt keine effektive Adresse, und der Inhalt des Arbeitsregisters ist nach Ausführung dieses Befehls unverändert.

*Bei Auswertung eines WRITE-Befehls zeigt der Simulator den Namen des im GR-Feld spezifizierten Arbeitsregisters und das im AD-Feld spezifizierte Zahlensystem wie auch den Inhalt des Arbeitsregisters in diesem Zahlensystem an. Der Simulator wartet dann auf Tasteneingabe, bis die Taste EXE gedrückt wird.

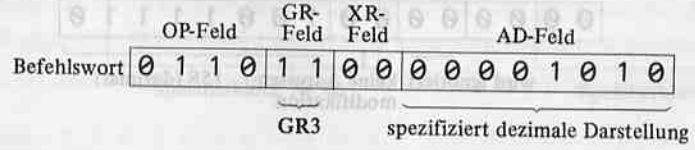


*Wenn die Taste EXE gedrückt wird, addiert der Simulator 1 zum Inhalt des Ablaufzählers (SC) und liest den nächsten Befehl.

*Wenn ein anderer Wert als 10 (dezimal) oder 16 (dezimal) im AD-Feld steht oder wenn ein anderer Wert als 0 für das XR-Feld spezifiziert ist, tritt ein Fehler (Operand Error) auf.

Beispiel:

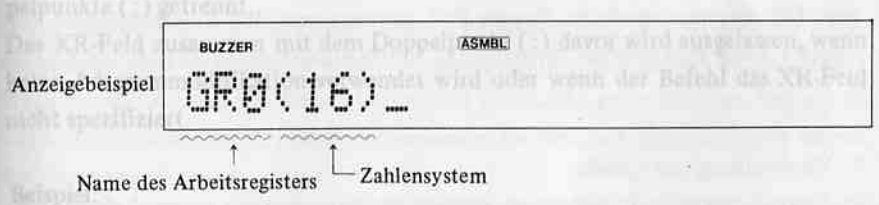
Der folgende Befehl im Programm wird ausgeführt, um den Inhalt des Arbeitsregisters GR3 in dezimaler Darstellung anzuzeigen.



- **READ-Befehl (binärer Befehlscode 0101)**

Speichert einen von der Tastatur eingegebenen numerischen Wert in das im GR-Feld spezifizierte Arbeitsregister. Dieser numerische Wert kann dezimal oder hexadezimal sein, das Zahlensystem wird im AD-Feld spezifiziert. Für das XR-Feld wird immer 0 spezifiziert, und es existiert keine effektive Adresse.

*Bei der Auswertung eines READ-Befehls zeigt der Simulator den im GR-Feld spezifizierten Arbeitsregisternamen und das im AD-Feld spezifizierte Zahlensystem an. Der Simulator wartet dann auf Tasteneingabe, bis die Taste EXE gedrückt wird.



*Numerische Werte entsprechend des angezeigten Zahlensystems eingeben und die Taste EXE drücken. Nach Drücken der Taste EXE interpretiert der Simulator den eingegebenen Wert entsprechend des Zahlensystems im AD-Feld und, wenn es übereinstimmt, speichert diesen Wert in das Arbeitsregister.

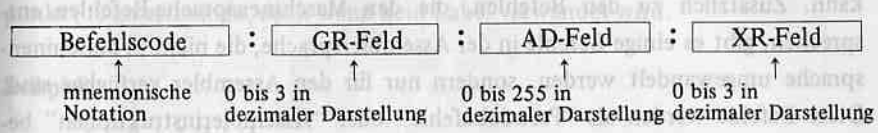
*Wenn das Zahlensystem nicht übereinstimmt oder der Eingabebereich (0000 ~ FFFF in hexadezimaler Darstellung) überschritten wurde, kehrt der Simulator beim Interpretieren des eingegebenen Werts in den Eingabe-Wartezustand zurück und fordert die Eingabe eines neuen Wertes.

*Wenn im AD-Feld ein anderer Wert als 10 (dezimal) oder 16 (dezimal) enthalten ist, tritt ein Operandenfehler auf. Ein Operandenfehler tritt ebenfalls auf, wenn im XR-Feld ein anderer Wert als 0 spezifiziert ist.

Beispiel:
Mit dem folgenden Befehl wird ein numerischer Wert von der Tastatur eingegeben und im Arbeitsregister GR1 gespeichert.

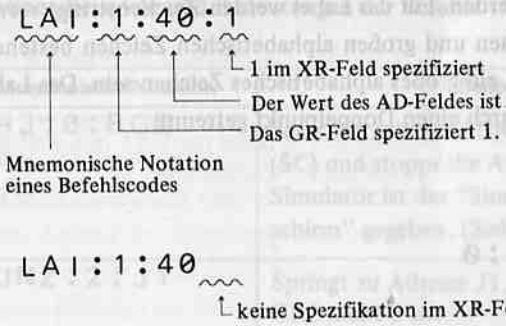
8 Regeln der mnemotechnischen Notation

In der Assemblersprache wird der Befehlscode jedes Maschinenbefehls mnemonisch dargestellt. Die Inhalte des GR-Feldes, XR-Feldes und AD-Feldes werden wie folgt geschrieben.



Es muß beachtet werden, daß sich die obige Darstellung vom internen Aufbau von Befehlsworten unterscheidet: die Information des XR-Feldes kommt zuletzt.
In der Simulator-Assemblersprache werden die Feld-Informationen durch Doppelpunkte (:) getrennt.
Das XR-Feld zusammen mit dem Doppelpunkt (:) davor wird ausgelassen, wenn keine Adressenmodifikation verwendet wird oder wenn der Befehl das XR-Feld nicht spezifiziert.

Beispiel:



- **Pseudobefehl und Label**

Ein in der Assemblersprache geschriebenes Programm wird vom Assembler in ein Maschinensprache-Programm umgewandelt, das von der CPU ausgeführt werden kann. Zusätzlich zu den Befehlen, die den Maschinensprache-Befehlen entsprechen, gibt es einige Befehle in der Assemblersprache, die nicht in Maschinensprache umgewandelt werden, sondern nur für den Assembler verfügbar sind. Diese Befehle werden als "Pseudobefehle" oder "Assemblerinstruktionen" bezeichnet.

In der Assemblersprache können auch spezifische Adressen und Konstante im Programm verwendet werden, in dem ein "Label" angefügt wird. Da diese Labels beim Assemblieren durch den Assembler in Adressen oder Konstante verwandelt werden, können sie in gewisser Hinsicht auch als Pseudobefehle angesehen werden. Sie werden jedoch in diesem Fall getrennt von den Pseudobefehlen verarbeitet. Eine ausführliche Beschreibung der Pseudobefehle finden Sie in Kapitel 9.

- **Verwendung von Labels**

In Assemblersprache-Programmen können Befehlscodes mit einem Label aus bis zu drei Zeichen versehen werden. Für das Label werden Zeichenstrings verwendet, die aus numerischen Zeichen und großen alphabetischen Zeichen bestehen. Das erste Zeichen muß jedoch ein großes alphabetisches Zeichen sein. Das Label und der Befehlscode werden durch einen Doppelpunkt getrennt.

Beispiel:

```
CLR:LAI:1:0
Label

LP:SFT:0:8:1
Label

A:JNZ:0:128
Label

L1:ST:0:64
Label
```


Beim Assemblieren eines Programms, das in Assemblersprache geschrieben ist, geht der Simulator davon aus, daß jede Programmzeile links mit einem Label beginnt. Daher ist es erforderlich, immer vor dem Befehlscode einen Doppelpunkt (:) zu schreiben, auch wenn kein Label verwendet wird.

Beispiel:

```
:LAI:0:0
└ Der Doppelpunkt (:) darf nicht weggelassen werden, auch wenn kein Label
  verwendet wird.
LAI:0:0
└ Dies wird beim Assemblieren als Label angesehen.
```

Wenn ein bestimmtes Befehlslabel verwendet wird (außer bei bestimmten Pseudobefehlen, die in Kapitel 9 erklärt werden), kann dieses Label auf die gleiche Weise wie die Adresse im Hauptspeicher, wo der Befehl gespeichert ist, behandelt werden. Ein definiertes Label kann verwendet werden, um das AD-Feld in einem Befehl zu spezifizieren.

- **Notations-Beispiele und Bedeutungen der Befehle**

Mnemonische Notation	Bedeutung
:HJ:0:BGN	Setzt die Adresse BGN in den Ablaufzähler (SC) und stoppt die Ausführung. In diesem Simulator ist der "Simulator-Menubildschirm" gegeben. (Siehe Kapitel 10-3.)
:JNZ:2:J1	Springt zu Adresse J1, wenn der Inhalt von GR2 nicht 0 ist.
:JC:1:NG	Springt zu Adresse NG, wenn der Inhalt des Anzeigeregisters (CC) gleich 1 ist.
:SFT:1:3:0	Verschiebt den Inhalt von GR1 um 3 Bits nach rechts.
:LAI:0:0	Setzt den Inhalt von GR0 auf 0.
:ADD:1:ONE	Addiert den Inhalt der Adresse ONE zum Inhalt von GR1.

Mnemonische Notation	Bedeutung
:ADD:2:TBL:1	Addiert den Inhalt der Adresse "TBL + (GR1)" zum Inhalt von GR2. *(GR1) bedeutet den Inhalt von GR1. *Eine Adressenmodifikation wird durchgeführt, um GR1 zu einem Indexregister zu machen.
:LD:0:WK	Speichert den Inhalt des Wortes in Adresse WK in GR0.
:ST:1:SAV	Speichert den Inhalt von GR1 in Adresse SAV.
:AND:2:MSK	Berechnet das logische Produkt des Inhalts von GR2 und des Inhalts des Wortes in Adresse MSK und speichert es in GR2.
:EOR:3:AL1	Berechnet das exklusive OR des Inhalts von GR3 und des Inhalts des Wortes in Adresse AL1 und speichert es in GR3.
:READ:0:10	Eingabe eines Wertes in dezimaler Darstellung von der Tastatur und Speicherung in GR0.
:WRITE:1:16	Anzeige des Inhaltes von GR1 in 4-stelliger hexadezimaler Darstellung.

9 Pseudobefehle

Zusätzlich zu den 14 Typen Maschinenbefehlen verfügt die Assemblersprache dieses Simulators über fünf Typen Pseudobefehle. Die Befehlscodes werden durch die folgenden mnemonischen Notationen angegeben.

START
END
RESV
CONST
ADCON

Die Pseudobefehle sind Instruktionen für den Assembler und werden nicht in Maschinensprache übersetzt.

- **Pseudobefehl START** (Programm/Unterprogramm-Start)

Format: Label :START: *n*

Funktion:
Dieser Pseudobefehl muß immer am Anfang eines Assemblersprache-Programms stehen, das Label kann jedoch ausgelassen werden. Das *n* ist eine Dezimalzahl, die die Speicher-Startadresse beim Speichern eines in Maschinensprache assemblierten Programms in den Hauptspeicher spezifiziert.
Das Label des Pseudobefehls START dient als Einsprung von einem anderen Programm bei Verwendung von mehreren Programmen, außerdem wird dadurch das Wort am Anfang des Programms angegeben, das mit diesem Befehl startet. Der Einsprung von einem anderen Programm ist möglich, indem das Label in den Operanden (Objekt) des Pseudobefehls ADCON geschrieben wird.

Beispiel:

```
        :START:32     ....  Spezifiziert den Programm-Speicher-
                            anfang in Adresse 32.
BGN:LD  :1:M          ....  Das Label BGN des nächsten Befehls
                            bedeutet daher 32.
```

- **Pseudobefehl END** (Programm/Unterprogramm-Ende)

Format: `:END:`*n*

Funktion:
Dieser Pseudobefehl muß am Ende jedes Programms in Assemblersprache geschrieben werden. Vor diesem Pseudobefehl darf niemals ein Label stehen, weil sonst beim Assemblieren ein Fehler auftritt. Das *n* ist eine Dezimalzahl oder ein Labelname und spezifiziert eine Adresse für den Start der Programmausführung. Das *n* kann zusammen mit dem davor stehenden Doppelpunkt (:) ausgelassen werden.

Beispiel 1:

`:END:BGN`	Dies bedeutet, daß das Programm hier zu Ende ist und daß die Adresse für Start der Programmausführung BGN ist.

Beispiel 2:

`:END`	Die Adresse für den Ausführungsstart braucht nicht spezifiziert zu werden. In dieser Form wird der Pseudobefehl END am Ende von Unterprogrammen geschrieben.

- **Pseudobefehl RESV** (Speicherplatz reservieren)

Format: Label `:RESV:`*n*

Funktion:
Für *n* wird eine Dezimalzahl geschrieben. Dieser Pseudobefehl dient zum Reservieren von Speicherplatz im Hauptspeicher für *n* fortlaufende Worte. Die Programmspeicherung ändert jedoch nicht den Inhalt des reservierten Speicherplatzes. Das Label kennzeichnet die Adresse des ersten Wortes im reservierten Speicherbereich. Es kann ausgelassen werden.

Beispiel:

`TBL:RESV:12`	Reserviert Speicherplatz im Hauptspeicher für 12 aufeinanderfolgende Worte. Die Adresse dieses Bereichs wird bestimmt durch die Position der Befehle im Programm. Auf die Adresse des ersten Worts im reservierten Bereich kann durch den Labelnamen TBL zugegriffen werden.

- **Pseudobefehl CONST** (Konstantendefinition)

Format: Label `:CONST:`*h*

Funktion:
Für *h* wird eine 4-stellige Hexadezimalzahl geschrieben. Im Hauptspeicher wird ein Wort reserviert, und die für *h* geschriebene Hexadezimalzahl wird im Hauptspeicher als Konstante gespeichert.
Das Label kennzeichnet die Adresse des Wortes, in dem die Konstante *h* gespeichert ist. Es kann ausgelassen werden.

Beispiel:

`AL1:CONST:FFFF`	Reserviert ein Wort im Hauptspeicher und setzt die hexadezimale Konstante FFFF in dieses Wort. Auf die Adresse dieses Wortes kann mit dem Labelnamen AL1 zugegriffen werden.

- **Pseudobefehl ADCON** (Adressenkonstante-Definition)

Format: Label :ADCON:*n*

Funktion:
Für *n* wird eine Dezimalzahl oder ein Labelname geschrieben. Im Hauptspeicher wird ein Wort reserviert, und der für *n* geschriebene Wert wird als Adressenkonstante gespeichert.
Wenn *n* ein Labelname ist und dieser Labelname im gleichen Programm definiert ist, wird der Wert der Adresse, die durch den Labelnamen angegeben ist, die Adressenkonstante. Wenn *n* ein Labelname ist und dieser Labelname nicht im gleichen Programm definiert ist, wird die Adressenkonstante mittels des Labels des Pseudobefehls START eines anderen Programms bestimmt. Das Label vor dem Pseudobefehl ADCON kennzeichnet die Adresse des reservierten Wortes zum Speichern der Adressenkonstante.

Beispiel 1:

```
ADR:ADCON:128 ····
```
Die Zahl 128 wird in Adresse ADR als Adressenkonstante gesetzt.

Beispiel 2:

```
        :START:32
          ⋮
SBR:ADCON:MUL ····
          ⋮
        :END   :BGN
MUL:START:300
          ⋮
        :END
```
Der Labelname MUL repräsentiert die Adresse 300 eines anderen Programms und ist in Adresse SBR als Adressenkonstante gespeichert.

Da in diesem Beispiel die Adresse eines anderen Programms in Adresse SBR gespeichert wird, ist es möglich, die Rückkehradresse in GR0 zu speichern und mit dem Unterprogramm-Sprungbefehl ": JSR : 0 : SBR" zu einem Unterprogramm zu springen, das in Adresse MUL beginnt.

10 Grundsätzliche Operation des Simulators

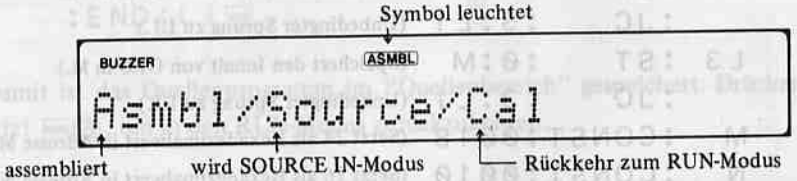
Wir werden jetzt mit diesem Simulator die "grundsätzliche Operation" versuchen, in Assemblersprache geschriebene Programme einzugeben und das Maschinenprogramm nach dem Assemblieren auszuführen.
Für jedes Programmbeispiel werden ausführliche Erklärungen gegeben. Versuchen Sie anhand der Erklärungen, die Programme auszuführen.

10-1 Erzeugung eines Quellenprogramms

Ein in Assemblersprache geschriebenes Programm wird **Quellenprogramm** genannt. Der Assembler übersetzt dann das Quellenprogramm in ein Maschinenprogramm, das die CPU ausführen kann. Dieses Programm in Maschinensprache wird als "**Objektprogramm**" oder einfach nur als "**Objekt**" in Beziehung zum Quellenprogramm bezeichnet.
Wir wollen jetzt ein Quellenprogramm mit dem Simulator erstellen. Dieser Computer ist so aufgebaut, daß ein in Assembler-Sprache geschriebenes Quellenprogramm im "Quellenbereich" gespeichert wird. Daher muß zum Schreiben eines Quellenprogramms zuerst die Taste [Asmbl] gedrückt werden, so daß das Simulator-Startmenü angezeigt wird, danach wird die Taste [S] gedrückt, um den SOURCE IN-Modus einzuschalten. Da es nicht wünschenswert ist, andere Daten im "Quellenbereich" zu haben, sollte anfangs im WRT-Modus ([MODE][1]) der Befehl NEW* ausgeführt werden.

- **Anzeige des Assemblierung-Menüs**

Wenn die Taste [Asmbl] im RUN- oder WRT-Modus gedrückt wird, erscheint die folgende Anzeige.



Wir wollen diese Anzeige als "Assemblierung-Menü" bezeichnen. Bei Anzeige dieses Menüs sind die folgenden Tastenbetätigungen möglich.

Tastenbetätigung	Funktion
[A]	Assembliert ein in Assemblersprache geschriebenes Quellenprogramm.
[S]	Schaltet in den SOURCE IN-Modus (für Eingabe und Korrektur von Quellenprogrammen).
[C]	Rückkehr zum RUN-Modus.
[BRK]	Rückkehr zum RUN-Modus.

Als Beispiel wollen wir ein Programm verwenden, das den größten gemeinsamen Teiler von zwei Werten berechnet (hier verwenden wir die Dezimalzahlen 16 und 24).

Quellenprogrammliste

Label	Befehlscode	Operand	Erklärung
	:START	:32	(Spezifiziert Adresse 32 als Speicherstartadresse.)
L1	:LD	:0:M	(Speichert M in GR0.)
	:SUB	:0:N	(Setzt M – N in GR0.)
	:JNZ	:0:L2	(Springt zu L2, wenn M – N ≠ 0.)
	:HJ	:0:L1	(Stoppt das Programm und setzt SC in L1.)
L2	:JC	:2:L3	(Springt zu L3, wenn M – N > 0.)
	:LD	:0:N	(Speichert N in GR0.)
	:SUB	:0:M	(Setzt N – M in GR0.)
	:ST	:0:N	(Speichert den Inhalt von GR0 in N.)
	:JC	:3:L1	(Unbedingter Sprung zu L1.)
L3	:ST	:0:M	(Speichert den Inhalt von GR0 in M.)
	:JC	:3:L1	(Unbedingter Sprung zu L1.)
M	:CONST	:0018	(Setzt 24 als Hexadezimalwert in Adresse M.)
N	:CONST	:0010	(Setzt 16 als Hexadezimalwert in Adresse N.)
	:END	:L1	(Ende des Programms. Ausführungsadresse ist L1.)

- **Schreiben des Quellenprogramms**

Bei der Eingabe dieses Quellenprogramms darf nicht vergessen werden, die Doppelpunkte (:) als Trennungszeichen einzugeben. Auch am Zeilenanfang muß ein : gesetzt werden, wenn kein Label vorhanden ist. Nach Eingabe jeder Zeile muß die Taste [EXE] gedrückt werden.

Tastenbetätigung

[MODE][1]
NEW[*][EXE] } Löscht die Daten im "Quellenbereich" (WRT-Modul).

[Asmbl] ¦ Anzeige des Assemblierung-Menüs.

[S] ¦ Spezifiziert den SOURCE IN Modus ("[SOURCE] [IN]" auf dem Display leuchtet).

```
:START:32[EXE]
L1:LD:0:M[EXE]
:SUB:0:N[EXE]
:JNZ:0:L2[EXE]
:HJ:0:L1[EXE]
L2:JC:2:L3[EXE]
:LD:0:N[EXE]
:SUB:0:M[EXE]
:ST:0:N[EXE]
:JC:3:L1[EXE]
L3:ST:0:M[EXE]
:JC:3:L1[EXE]
M:CONST:0018[EXE]
N:CONST:0010[EXE]
:END:L1[EXE]
```

Damit ist das Quellenprogramm im "Quellenbereich" gespeichert. Drücken Sie jetzt [MODE][⊘], um in den RUN-Modus zurückzukehren.

- **Anzeige des Quellenprogramms**

Zum Überprüfen wollen wir das Qullenprogramm anzeigen. Durch Ausführung des Befehls LIST∗ 1 im RUN- oder WRT-Modus kann der Inhalt des "Quellenbereichs" im folgenden "Quellenprogramm"-Format angezeigt werden.

Label : Befehlscode : GR-Feld : AD-Feld : XR-Feld

drei Zeichen	fünf Zeichen	ein Zeichen	drei Zeichen	entsprechend des Quellenprogramms

Das soeben eingegebene Quellenprogramm kann durch die Tastenfolge [SHIFT] [LIST] [∗] [1] [EXE] angezeigt werden. Die Anzeige stoppt nach jeder Zeile, durch Drücken von [EXE] wird zur nächsten Zeile weitergegangen.

Tastenbetätigung	Anzeige
[MODE] [⊘]	Ready P0
LIST∗1 [EXE]	:START:32
[EXE]	L1 :LD :0:M
[EXE]	:SUB :0:N
[EXE]	:JNZ :0:L2
[EXE]	:HJ :0:L1
[EXE]	L2 :JC :2:L3
[EXE]	:LD :0:N
[EXE]	:SUB :0:M
[EXE]	:ST :0:N
[EXE]	:JC :3:L1
[EXE]	L3 :ST :0:M
[EXE]	:JC :3:L1
[EXE]	M :CONST:0018
[EXE]	N :CONST:0010
[EXE]	:END :L1
[EXE]	Ready P0

∗Das Quellenprogramm kann mit den vorher beschriebenen Befehlen LIST∗ und LIST∗ (0) angezeigt werden (die Zahl in Klammern kann ausgelassen werden). Im letzteren Fall werden die Zeilen des Quellenprogramms mit Zeilennummer fortlaufend im Abstand von etwa einer Sekunde angezeigt. Zum Unterbrechen der Anzeige und Überprüfen des Inhalts die Taste [STOP] drücken. Durch Drücken von [EXE] wird die Anzeige fortgesetzt.

- **Editieren des Quellenprogramms** (Korrigieren, Einfügen, Löschen)

Siehe Kapitel 7 der "BEDIENUNGSANLETUNG", das Editieren erfolgt auf die gleiche Weise wie für die Memodaten in der Datenbank.

10-2 Assemblieren

Der Simulator verfügt über **Assemblierungsfunktionen**, um das in Assemblersprache geschriebene Quellenprogramm zu einem Objektprogramm zu assemblieren. Zum **Assemblieren** zuerst die Taste [Asmbl] und danach die Taste [A] drücken. Wird die Taste [Asmbl] nach Schreiben des Quellenprogramms in den Quellenbereich gedrückt, leuchtet oben auf dem Display das Symbol [ASMBL]. Dann wird durch Drücken der Taste [A] aus dem Quellenprogramm das Objektprogramm erzeugt. Entsprechend des freien Speicherplatzes (Anzahl der verbleibenden Bytes) zum Zeitpunkt vom Drücken der Taste [A] wird ein Speicherblock im Hauptspeicher zum Erzeugen des Objektprogramms reserviert.

Anzahl der verbleibenden Bytes	Reservierter Speicherblock
0 ~ 512 Bytes	Freier Speicherplatz unzureichend zum Erzeugen des Objektprogramms. "Error 1" wird angezeigt.
513 ~ 1024 Bytes	Reserviert einen Speicherblock (256 Worte) und assembliert.
1025 ~ 1536 Bytes	Reserviert zwei Speicherblöcke (512 Worte) und assembliert.
⋮	
über 4097 Bytes	Reserviert acht Speicherblöcke (2048 Worte) und assembliert.

Wenn Programm-Label vorhanden sind, werden jedoch für jedes Label fünf Bytes benötigt.
Die Adresseneingabe ist im Bereich der reservierten Adressen (0 ~ 2047 im Fall von 8 Speicherblöcken) wirksam.
Der Simulator verfügt außerdem über die Fähigkeit, beim Assemblieren Fehler zu entdecken und anzuzeigen. Die Fehlermeldungen beim Assemblieren unterscheiden sich von den Fehlermeldungen beim Ausführen von BASIC-Programmen, sie sind unten aufgeführt.

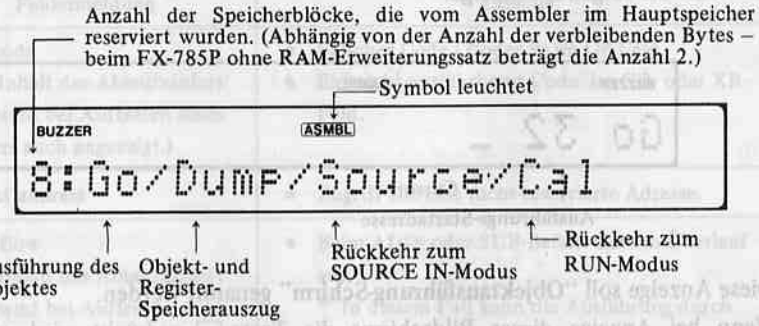
Fehlermeldung	Fehler
Error 1	• Speicherblock zum Erzeugen des Objektprogramms kann nicht reserviert werden. • Unzureichender Platz für Label-Tabelle.
Label error (Protokoll-Nr. der fehlerhaften Zeile wird auch angezeigt)	• Ein Label wurde doppelt deklariert. • Der Pseudobefehl END hat ein Label. • Der Anfangsbuchstabe des Labels ist kein Großbuchstabe. • Das Label überschreitet drei Zeichen.
Op-code error (Protokoll-Nr. der fehlerhaften Zeile wird auch angezeigt)	• Der Befehlscode ist falsch geschrieben. • Der Befehlscode ist nicht vorhanden.
Operand error (Protokoll-Nr. der fehlerhaften Zeile wird auch angezeigt)	• Der Operand (Inhalt des GR-Feldes und AD-Feldes) ist nicht vorhanden. • Der Operand (Inhalt des GR-Feldes und XR-Feldes) ist nicht vorhanden. • Der Operand (Inhalt des AD-Feldes) überschreitet drei Zeichen. • Für Bezug auf den Operanden (Inhalt des AD-Feldes) ist kein Label vorhanden.
Source error	• Keine Pseudobefehle für START oder END. • Andere Fehler

Wenn eine Fehlermeldung angezeigt wird, zuerst den SOURCE IN-Modus (Asmbl S) einschalten und dann das Quellenprogramm entsprechend des Fehlers korrigieren. Wenn die Assemblierung richtig ausgeführt wurde, werden GR0, GR1, GR2 und GR3 zu 0 (gelöscht). Wenn ein Fehler auftritt, die folgenden Maßnahmen durchführen.

1. Wenn Error 1 auftritt
   a) Die Assemblierung abbrechen und mit der Taste BRK in den RUN-Modus schalten.
   b) Mit der Taste EXE in den SOURCE IN-Modus schalten.
2. Wenn ein Label-Fehler, Operandencode-Fehler oder Operandenfehler auftritt.
   a) Die Assemblierung vorübergehend unterbrechen und mit der Taste BRK in den RUN-Modus schalten.
   b) Die Assemblierung mit der Taste EXE fortsetzen und nach Beendigung in den SOURCE IN-Modus schalten.

10-3 Menübildschirm des Simulators

Wenn das Assemblieren richtig durchgeführt wird, erscheint die folgende Anzeige, wobei das Symbol "ASMBL" leuchtet.

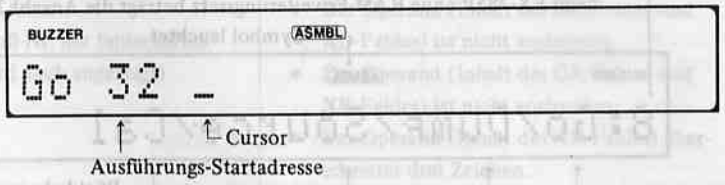


Wir wollen diese Anzeige "Menubildschirm des Simulators" nennen. Bei Anzeige des Menubildschirms des Simulators sind die folgenden Tastenbetätigungen möglich.

Tastenbetätigung	Funktion
[G]	Umschaltung auf den "Objektausführungs-Schirm" (Erklärung in Kapitel 10 - 4).
[D]	Umschaltung auf den "Speicherauszug-Menuschirm" (Erklärung in Kapitel 10 - 6).
[S]	Rückkehr zum SOURCE IN-Modus.
[C]	Rückkehr zum RUN-Modus.
[MODE] [.]	Ein- und Ausschalten des Summertons bei Tasteneingabe. (Bei ON leuchtet das Symbol "BUZZER".)
[MODE] [2]	Aktiviert die Ablaufverfolgung. (Symbol "TRACE ON" leuchtet.)
[MODE] [3]	Schaltet die Ablaufverfolgung aus.
[MODE] [7]	Aktiviert den Drucker-Modus. (Symbol "PRT ON" leuchtet.)
[MODE] [8]	Schaltet den Drucker-Modus aus.

10-4 Ausführung des Objektes

Wenn bei Anzeige des "Simulator-Menuschirms" die Taste [G] gedrückt wird, erscheint die folgende Anzeige.



Diese Anzeige soll "**Objektausführung-Schirm**" genannt werden.

Wenn bei Anzeige dieses Bildschirms die Taste [EXE] gedrückt wird, wird das Objekt ab der Ausführungs-Startadresse ausgeführt.

Zum Ändern der Ausführungs-Startadresse eine neue Adresse oder ein entsprechendes Label eingeben und die Taste [EXE] drücken. Dann wird die Ausführungs-Startadresse geändert, und die Anzeige kehrt zurück zum "Objektausführung-Bildschirm".

Tastenbetätigung	Anzeige	
	Go 32 _	
[4] [0]	Go 32 40_	(Ändert die Adresse in 40.)
[EXE]	Go 40 _	
[L] [1]	Go 40 L1_	(Ändert zu Label L1 Adresse.)
[EXE]	Go 32 _	
[EXE] (Objekt-Ausführung)	8:Go/Dump/Source/Cal	(Rückkehr zum "Simulator-Menubildschirm".)

Wenn das Beispielsprogramm wie oben ausgeführt wird, kehrt die Anzeige zurück zum "Simulator-Menubildschirm", und die Programmausführung wird durch den HJ-Befehl ohne Anzeige gestoppt, weil keine READ- oder WRITE-Befehle vorhanden sind. Zur Beobachtung der Ausführung ist "Objekt-Ablaufverfolgung" erforderlich (Erklärung in Kapitel 10-5).

- **Fehlermeldungen bei der Ausführung von Objekten**

Das Objekt wird unter Kontrolle des Simulators ausgeführt. Wenn während der Ausführung Fehler auftreten, werden die folgenden Fehlermeldungen angezeigt.

Fehlermeldung	Ursache
Bad code (Der Inhalt des Ablaufzählers (SC) wird bei Auftreten eines Fehlers auch angezeigt.)	• Falscher Code (7 oder 9) im OP-Feld. • Ein nicht-ausführbarer Code im GR- oder XR-Feld.
Out of address	• Zugriff auf eine nicht-reservierte Adresse.
Over flow (Der Inhalt des Ablaufzählers (SC) wird bei Auftreten eines Fehlers auch angezeigt.)	• Beim ADD- oder SUB-Befehl trat ein Überlauf auf. * In diesem Fall kann die Ausführung durch Drücken der Taste [EXE] fortgesetzt werden.

Zum Löschen des Fehlers die Taste [BRK] drücken. Dann kehrt die Anzeige zum "Simulator-Menubildschirm" zurück.

10-5 Ablaufverfolgung des Objektes

Wenn das Objekt mit Ablaufverfolgung ausgeführt wird (MODE 2 drücken, "TRACE ON" leuchtet auf dem Display), stoppt die Ausführung nach Anzeige des Inhalts des Objektes und des Arbeitsregisters im folgenden Format für jeden Befehl. Zum Fortsetzen der Ausführung und Anzeige des nächsten Befehls die Taste EXE drücken.

Adresse : Objekt ␣ GR0-Inhalt ␣ GR1-Inhalt ␣ GR2-Inhalt ␣ GR3-Inhalt

(3-stellig dezimal) (4-stellig hexadezimal) (dezimal) (dezimal) (dezimal) (dezimal)

(␣ kennzeichnet eine Leerstelle)

*Wenn die Anzeige 24 Zeichen überschreitet, wird sie nach links verschoben, so daß die Zeichen an der rechten Seite auf das Display kommen. Zum Anhalten der Anzeige STOP drücken, durch Drücken von EXE wird die Anzeige weiter verschoben.

Auf diese Weise kann die Ausführung des Objektes verfolgt werden. Wir wollen die Ausführung unseres Programmbeispiels verfolgen.

Tastenbetätigung	Anzeigebeispiele
	BUZZER ASMBL 8:Go/Dump/Source/Cal
MODE 2	BUZZER ASMBL TRACE ON 8:Go/Dump/Source/Cal
G	BUZZER ASMBL TRACE ON Go 32 _
EXE	BUZZER ASMBL TRACE ON 32:C02B 8 0 0 0
EXE	BUZZER ASMBL TRACE ON 33:B02C 0 0 0 0
EXE	BUZZER ASMBL TRACE ON 34:1024 0 0 0 0

(Die folgenden Tastenbetätigungen werden ausgelassen.)

*Die Anzeige ist abhängig vom Speicherinhalt des Computers und kann vom obigen Beispiel abweichen.

Einfache Verfolgung der Ausführung des Objektprogramms ist nicht sehr ergiebig, da nur die im Arbeitsregister gespeicherten "bedeutungslosen" Werte der Reihe nach angezeigt werden. Änderungen im Register sind gut verständlich, wenn zuerst der Registerinhalt initialisiert und danach durch Drücken von A im Assembler-Menü die Ablaufverfolgung durchgeführt wird. Zum Ausschalten des TRACE-Modus MODE 3 drücken. Dann erlischt das Symbol "TRACE ON".

10-6 Objekt- und Register-Speicherauszug

Die Ausgabe des Wortinhalts im Hauptspeicher oder des Registerinhalts zu einer Ausgabevorrichtung wird "**Speicherauszug**" (Dump) genannt.
Wenn beim "Simulator-Menubildschirm" die Taste D gedrückt wird, erscheint die folgende Anzeige.

BUZZER ASMBL
Dump:Object/Register

Wir bezeichnen diese Anzeige als "**Speicherauszug-Menuschirm**".
Bei diesem "Speicherauszug-Menuschirm" sind die folgenden Tastenbetätigungen möglich.

Tastenbetätigung	Funktion
O	Speicherauszug des Objekt-Inhalts.
R	Speicherauszug des Register-Inhalts.

- **Objekt-Speicherauszug**

Wenn beim "Speicherauszug-Menuschirm" die Taste O gedrückt wird, wird ein Speicherauszug des Objektes ausgegeben. Die Speicherauszug-Startadresse und -Endadresse werden abgefragt, die entsprechenden Adresswerte oder Labels müssen eingegeben werden.

Tastenbetätigung	Anzeige
	Dump:Object/Register
[0]	from _
	↑ Anzeige des Cursors und wartet auf die Eingabe der Speicherauszug-Startadresse.
[L][1] (Eingabe des Labels.)	from L1_
[EXE]	from 32 to _
	↑ Wartet auf die Eingabe der Speicherauszug-Endadresse.
[4][2] (Eingabe des Adresswertes.)	from 32 to 42_

Wenn nach Eingabe der Speicherauszug-Startadresse und -Endadresse die Taste [EXE] gedrückt wird, wird der Speicherauszug des Objektes im folgenden Format im spezifizierten Bereich angezeigt.

```
****  :  ****  *****  **  ␣  ***
Adresse  Objekt  Befehlscode  |  |     |
(dezimal) (hexadezimal) (mnemonisch)  |  |     AD-Feld (dezimal)
                                 |  XR-Feld
                                 GR-Feld
```

Bei einem READ-Befehl wird der Befehlscode an das GR-Feld angehängt und angezeigt, bei einem WRITE-Befehl wird auch das Objekt an das Feld angehängt und angezeigt.

Wir wollen jetzt einen Speicherauszug unseres Beispielprogramms von Adresse 32 bis Adresse 42 durchführen.

Tastenbetätigung	Anzeige
[EXE]	32:C02B LD 00 43
[EXE]	33:B02C SUB 00 44
[EXE]	34:1024 JNZ 00 36
[EXE]	35:0020 HJ 00 32
[EXE]	36:2829 JC 20 41
[EXE]	37:C02C LD 00 44
[EXE]	38:B02B SUB 00 43
[EXE]	39:D02C ST 00 44
[EXE]	40:2C20 JC 30 32
[EXE]	41:D02B ST 00 43
[EXE]	42:2C20 JC 30 32
[EXE]	8:Go/Dump/Source/Cal

Wenn nach dem Speicherauszug die Taste [EXE] gedrückt wird, kehrt die Anzeige zum "Simulator-Menuschirm" zurück.

- **Register-Speicherauszug**

Wenn bei Anzeige des "Speicherauszug-Menuschirms" die Taste [R] gedrückt wird, wird ein Speicherauszug des Registers angezeigt.
Der Speicherauszug der Register erfolgt in der Reihenfolge Basisregister (BR), Arbeitsregister (GR0, GR1, GR2, GR3), Ablaufzähler (SC) und Anzeigeregister (CC) im folgenden Format.

```
***  :  ␣  ****  ␣  *****  ␣  _
Registername  Registerinhalt  Registerinhalt  |  Cursor
              (4-stellig hexadezimal) (dezimal)
```

Tastenbetätigung	Anzeigebeispiele
	Dump:Object/Register
R	BR : 0100 256 _
⌫	BR : 0100 256 0_
EXE	BR : 0000 0 _
EXE	GR0: 0000 0 _
EXE	GR1: FFFF -1 _
⌫	GR1: FFFF -1 0_
EXE	GR1: 0000 0 _
EXE	GR2: 0008 8 _
⌫	GR2: 0008 8 0_
EXE	GR2: 0000 0 _
EXE	GR3: 0000 0 _
EXE	SC : 0020 32 _
EXE	CC : 0000 0 _
EXE	8:Go/Dump/Source/Cal

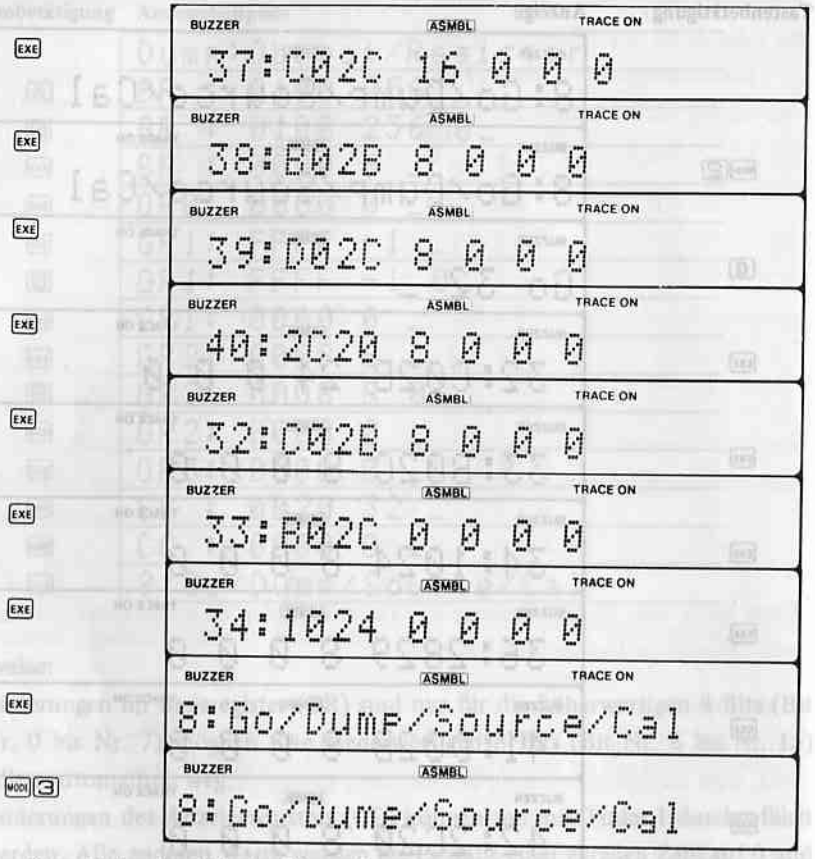
Hinweise:

1) Änderungen im Basisregister (BR) sind nur für die höherwertigen 8 Bits (Bit Nr. 0 bis Nr. 7) möglich. Die niederwertigen 8 Bits (Bit Nr. 8 bis Nr. 15) fallen automatisch weg.
2) Änderungen des Anzeigeregisters (CC) können nur mit 0 oder 1 durchgeführt werden. Alle anderen Werte werden bei Eingabe einer geraden Zahl auf 0 und bei Eingabe einer ungeraden Zahl auf 1 gesetzt.

*Die Inhalte der verschiedenen Register müssen nicht dem obigen Beispiel gleichen, da sie vom jeweiligen Speicherinhalt des Computers abhängig sind. GR0 bis GR3 werden jedoch unmittelbar nach dem Assemblieren 0.

Indem die Taste [A] bei Anzeige des Assemblierungs-Menüs gedrückt und die Assemblierung auf diese Weise ausgeführt wird und wenn nach Initialierung der einzelnen Registerinhalte die "Objekt-Ablaufverfolgung" durchgeführt wird, können Änderungen in den Registern wirkungsvoll verfolgt werden. Das Beispielsprogramm sieht dann wie folgt aus.

Tastenbetätigung	Anzeige
	BUZZER ASMBL 8:Go/Dump/Source/Cal
MODE 2	BUZZER ASMBL TRACE ON 8:Go/Dump/Source/Cal
G	BUZZER ASMBL TRACE ON Go 32 _
EXE	BUZZER ASMBL TRACE ON 32:C02B 24 0 0 0
EXE	BUZZER ASMBL TRACE ON 33:B02C 8 0 0 0
EXE	BUZZER ASMBL TRACE ON 34:1024 8 0 0 0
EXE	BUZZER ASMBL TRACE ON 36:2829 8 0 0 0
EXE	BUZZER ASMBL TRACE ON 41:D02B 8 0 0 0
EXE	BUZZER ASMBL TRACE ON 42:2C20 8 0 0 0
EXE	BUZZER ASMBL TRACE ON 32:C02B 8 0 0 0
EXE	BUZZER ASMBL TRACE ON 33:B02C -8 0 0 0
EXE	BUZZER ASMBL TRACE ON 34:1024 -8 0 0 0
EXE	BUZZER ASMBL TRACE ON 36:2829 -8 0 0 0

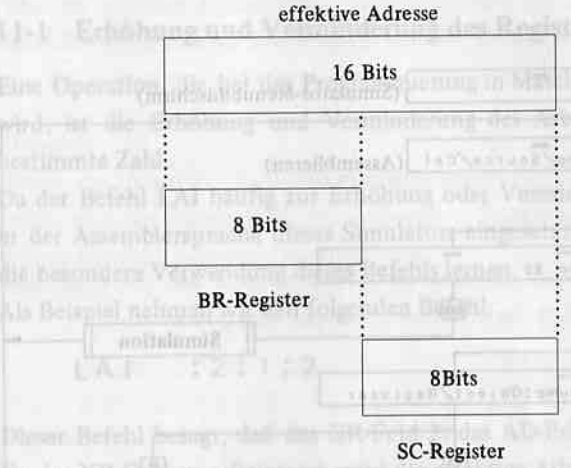


(Ausschalten der Ablaufverfolgung.)

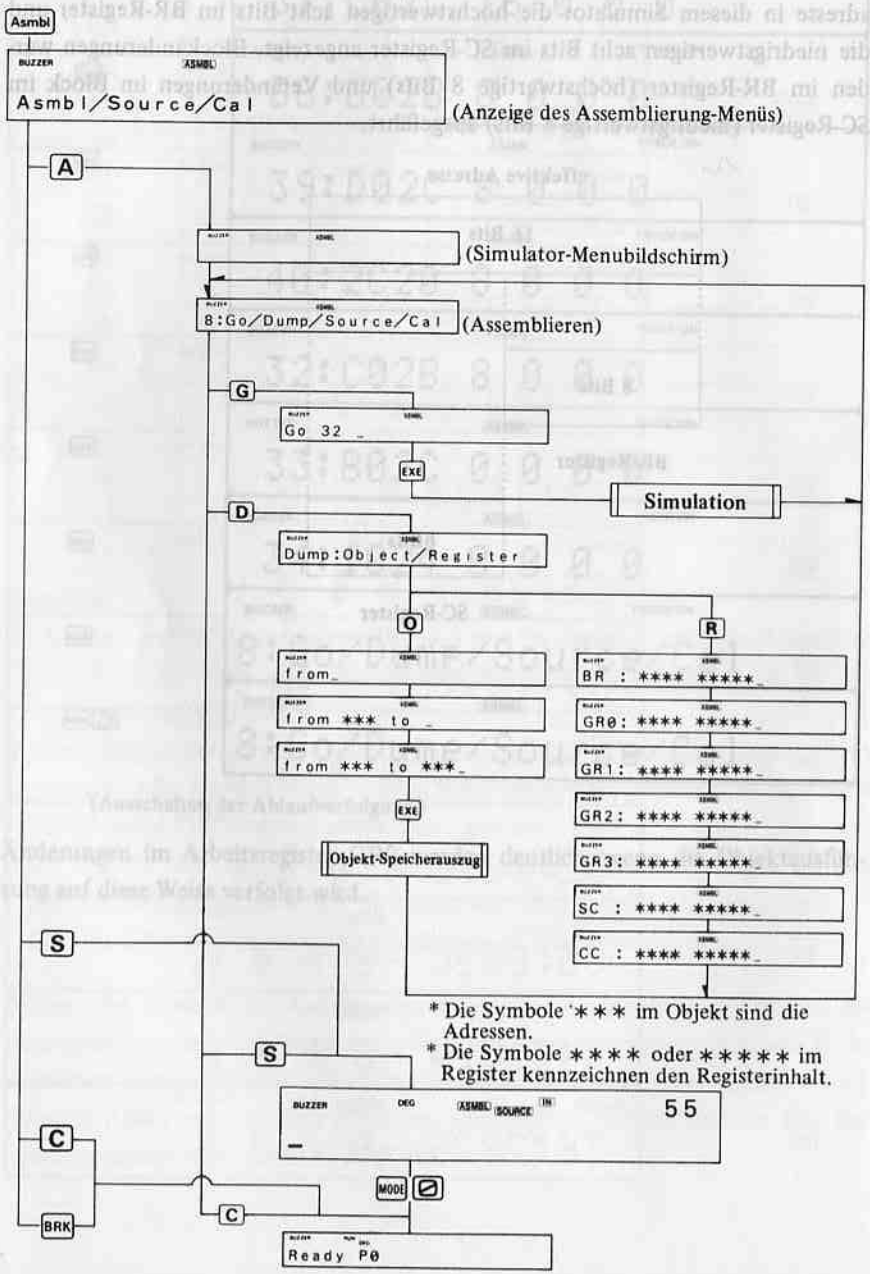
Änderungen im Arbeitsregister GR0 werden deutlich, wenn die Objektausführung auf diese Weise verfolgt wird.

- **Zur Beachtung beim Register-Speicherauszug**

Beim Register-Speicherauszug werden von den 16 Bits der effektiven Befehlsadresse in diesem Simulator die höchstwertigen acht Bits im BR-Register und die niedrigstwertigen acht Bits im SC-Register angezeigt. Blockänderungen werden im BR-Register (höchstwertige 8 Bits) und Veränderungen im Block im SC-Register (niedrigstwertige 8 Bits) ausgeführt.



10-7 Zusammenfassung der grundsätzlichen Operation des Simulators



11 Grundsätzliche Technik für Programmieren in Assembler

In diesem Kapitel wollen wir einige grundsätzliche Techniken zu Programmierung in der Assemblersprache mit diesem Simulator vorstellen. Da jeder Befehl in Assemblersprache nur einen sehr einfachen Prozeß ausführen kann, müssen mehrere Befehle gestapelt werden, wenn die CPU viele Operationen ausführen soll. Die Technik, die Sie in diesem Kapitel lernen, wird als "Baustein" hilfreich sein, wenn Sie ein längeres Programm in Assemblersprache schreiben.

11-1 Erhöhung und Verminderung des Registerinhalts

Eine Operation, die bei der Programmierung in Maschinensprache oft verwendet wird, ist die Erhöhung und Verminderung des Arbeitsregisterinhalts um eine bestimmte Zahl.
Da der Befehl LAI häufig zur Erhöhung oder Verminderung des Registerinhalts in der Assemblersprache dieses Simulators eingesetzt wird, wollen wir zunächst die besondere Verwendung dieses Befehls lernen.
Als Beispiel nehmen wir den folgenden Befehl:

```
LAI    :2:1:2
```

Dieser Befehl besagt, daß das GR-Feld 2, das AD-Feld 1 und das XR-Feld 2 ist. Da das XR-Feld spezifiziert ist, wird die effektive Adresse durch Adressenmodifikation mit dem Indexregister GR2 berechnet.

Effektive Adresse = 1 + (Inhalt von GR2)
(niederwertige 8 Bits) ↑ Wert des AD-Feldes

Durch den LAI-Befehl werden die niederwertigen 8 Bits (Bit Nr. 8 bis 15) dieser effektiven Adresse in das im GR-Feld spezifizierte Arbeitsregister gespeichert. Da in diesem Fall GR2 als Arbeitsregister spezifiziert ist, ist es möglich, mit dem folgenden Befehl den Inhalt von GR2 um 1 zu erhöhen.

GR2 ← niederwertige 8 Bits von [(Inhalt von GR2) + 1)]

Der LAI-Befehl kann jedoch nur Werte im Bereich 0 ~ 255 als die niederwertigen 8 Bits der effektiven Adresse verarbeiten. Werte über 255 werden durch 256 dividiert, der Divisionsrest wird als die niederwertigen 8 Bits eingesetzt.
Wir wollen den folgenden Befehl ausführen und dabei annehmen, daß der Inhalt des Arbeitsregisters GR2 gleich 4 ist.

```
LAI     :2:255:2
```

Die niederwertigen 8 Bits der gegenwärtigen effektiven Adresse können wie folgt berechnet werden:

Effektive Adresse = 255 + 4 = 259

(niederwertige 8 Bits) — Wert des AD-Feldes — Inhalt von GR2

Da 259 jedoch größer als 255 ist, wird der Divisionsrest 3 aus der folgenden Formel als niederwertige 8 Bits der effektiven Adresse verwendet.

259 ÷ 256 = 1 mit Rest 3

Durch Ausführung dieses Befehls in dieser Weise wird der Inhalt des Arbeitsregisters GR2 gleich 3, der ursprüngliche Wert 4 wurde um 1 vermindert.
Daher ist es möglich, diese Methode zu verwenden, um den Inhalt der Arbeitsregister GR1, GR2 und GR3 um 1 zu erhöhen oder zu vermindern.
Außerdem ist es möglich, durch Ändern des Wertes im AD-Feld den Inhalt des Arbeitsregisters entsprechend dem Wert zu ändern. Die folgende Tabelle zeigt einige Beispiele.

Befehl	Bedeutung
LAI :*n*:1:*n*	Erhöht GR*n* um 1
LAI :*n*:2:*n*	Erhöht GR*n* um 2
LAI :*n*:255:*n*	Vermindert GR*n* um 1
LAI :*n*:254:*n*	Vermindert GR*n* um 2

(*n* = 1, 2 oder 3)

Zu Erhöhung des Inhalts von GR*n* um eine bestimmte Zahl können Sie einfach diesen Wert in das AD-Feld schreiben. Wenn Sie den Inhalt von GR*n* vermindern wollen, ziehen Sie den Verminderungswert von 256 ab und schreiben das Ergebnis in das AD-Feld. (Beispiel: Zur Verminderung um 3 muß 253 (256 − 3 = 253) in das AD-Feld geschrieben werden.)
Diese Methode kann jedoch nicht für GR0 eingesetzt werden, da Adressenmodifikation mit dem Indexregister verwendet wird. Da die niederwertigen 8 Bits der effektiven Adresse auch verwendet werden, muß beachtet werden, daß Änderungen des Wertes nur im Bereich 0 ~ 255 möglich sind.

Beispiel:
Wenn das folgende Quellenprogramm assembliert und das Objekt ausgeführt wird, wird der Inhalt 5 des Arbeitsregisters GR1 um 1 vermindert, und die Ausführung stoppt, wenn der Inhalt 0 wird.

```
        :START:32          (Erzeugt das Objekt ab Adresse 32.)
BGN:LAI  :1:5              (Setzt den Anfangswert 5 in GR1.)
LP  :WRITE:1:10            (Anzeige des Inhalts von GR1 in dezimaler Darstellung.)
        :LAI  :1:255:1     (Erhöhung des Inhalts von GR1 um 1.)
        :JNZ  :1:LP        (Sprung zu LP, wenn GR1 ≠ 0.)
        :HJ   :0:BGN       (Setzt den Ablaufzähler auf BGN und stoppt.)
        :END  :BGN         (Ende des Programms)
```

Tastenbetätigung

Asmbl	Asmbl/Source/Cal
A	8:Go/Dump/Source/Cal
G	Go 32_
EXE	GR1 (10) 5
EXE	GR1 (10) 4
EXE	GR1 (10) 3
EXE	GR1 (10) 2
EXE	GR1 (10) 1
EXE	8:Go/Dump/Source/Cal

↳ Da der Inhalt von GR1 gleich 0 ist, wird die Ausführung durch den HJ-Befehl gestoppt, und die Anzeige kehrt zum "Simulator-Menubildschirm" zurück.

Beachten Sie, daß der Befehl "WRITE : 1 : 10" fünfmal ausgeführt wird.
Durch die oben gezeigte Programmierweise ist es im allgemeinen möglich, den Prozeß zwischen zwei LAI-Befehlen durch Verwendung von GR1 als Zähler (Register zum Zählen der Häufigkeit) mit einer spezifizierten Häufigkeit zu wiederholen.

```
      :LAI    :1: [Häufigkeit]
LP    : [Prozeß, der wiederholt wird]
        ⋮
      :LAI    :1:255:1
      :JNZ    :1:LP
```

(Bei Verwendung des LAI-Befehls ist die Häufigkeit jedoch auf den Bereich 1 ~ 256 beschränkt.)

11-2 Benutzung des Arbeitsbereichs

Es wird häufig vorkommen, daß mehrere Worte im Hauptspeicher zum vorübergehenden Speichern von Zwischenergebnissen oder des Inhalts eines Registers reserviert werden. Der Teil des Hauptspeichers, der für diesen Zweck verwendet wird, wird "**Arbeitsbereich**" genannt. Dieser Arbeitsbereich ist ähnlich den "Variablen" in einem BASIC-Programm. Wir wollen uns die Verwendung des Arbeitsbereichs anhand von Beispielen etwas genauer ansehen.

Beispiel: Transfer von Werten zwischen Arbeitsregistern

In der Simulator-Maschinensprache gibt es keinen Befehl zum direkten Übertragen von Inhalten zwischen zwei Arbeitsregistern. Aber die Notwendigkeit, die Inhalte der Arbeitsregister GR0 und GR1 auszutauschen, kann durchaus auftreten. Diese Aufgabe wird dann mithilfe des im Hauptspeicher reservierten Arbeitsbereichs durchgeführt.
Das folgende Programm dient zum Austauschen der Inhalte von GR0 und GR1.


```
      :START:32
BGN:READ  :0:10   } Eingabe eines Dezimalwertes in das Register.
      :READ  :1:10
SWP:ST    :0:WK1  (Speichert den Inhalt von GR0 in WK1.)
      :ST    :1:WK0  (Speichert den Inhalt von GR1 in WK0.)
      :LD    :0:WK0  (Lädt den Inhalt von WK0 in GR0.)
      :LD    :1:WK1  (Lädt den Inhalt von WK1 in GR1.)
OUT:WRITE :0:10   } Ausgabe des Registerinhalts in dezimaler
      :WRITE :1:10   } Darstellung.
      :HJ    :0:BGN  (Stoppt das Programm.)
WK0:RESV  :1      } Arbeitsbereich
WK1:RESV  :1      }
      :END   :BGN
```

In diesem Programm wird das Austauschen in vier Zeilen ab der Zeile mit dem Label SWP durchgeführt. Dieser Austausch erfordert zwei Arbeitsbereiche (Labels WK0 und WK1) und kehrt die Folge von Speichern und Laden um.
Beim Assemblieren dieses Programms und Ausführen des Objektes wird das folgende angezeigt.

Tastenbetätigung

Asmbl	Asmbl/Source/Cal
A	8:Go/Dump/Source/Cal
G	Go 32_
EXE	GR0 (10) _
1	GR0 (10) 1_
EXE	GR1 (10) _
2	GR1 (10) 2_
EXE	GR0 (10) 2
EXE	GR1 (10) 1
EXE	8:Go/Dump/Sorce/Cal

Die obige Tastenbetätigung zeigt, daß bei Eingabe des Wertes 1 in GR0 und des Wertes 2 in GR1 von der Tastatur durch das Austauschen der Wert in GR0 2 und der Wert in GR1 1 wird.

Beispiel: Speichern von Zwischenergebnissen

2-maliges, 4-maliges, 8-maliges, ... (allgemein 2^n-maliges) Erhöhen des Inhalts des Arbeitsregisters kann durch Linksverschieben mit dem SFT-Befehl durchgeführt werden. Was aber, wenn andere Faktoren als diese benötigt werden? Verschiedene Möglichkeiten sind denkbar, eine davon ist die Kombination von 2^n-Faktoren. In unserem Beispiel wollen wir ein Programm erstellen, das den Inhalt des Arbeitsregisters GR0 10 mal erhöht. Dabei nutzen wir aus, daß 10 = 2 + 8 ist.

```
        :START:32
BGN:READ  :0:10    (Eingabe des Dezimalwerts n in GR0.)
    :SFT   :0:1:1  (Verdoppelung (2 x n) des Inhalts von GR0
                    durch Verschiebung um ein Bit nach links.)
    :ST    :0:WK   (Speichern des Zwischenergebnisses (2 x n) in
                    WK.)
    :SFT   :0:2:1  (Erhöhung des Inhalts von GR0 um das acht-
                    fache (8 x n) durch Verschiebung um zwei
                    weitere Bits nach links.)
    :ADD   :0:WK   (Addierung von WK (2 x n) zu GR0 (8 x n)
                    und Speicherung in GR0.)
    :WRITE:0:10    (Ausgabe des Inhalts von GR0 (10 x n) in
                    dezimaler Darstellung.)
    :HJ    :0:BGN  (Stoppt das Programm.)
WK  :RESV  :1      (Arbeitsbereich)
    :END   :BGN
```

Der Inhalt von GR0 wurde durch den SFT-Befehl in der dritten Zeile verdoppelt, vorübergehend in Arbeitsbereich WK gespeichert und dann durch den SFT-Befehl in der fünften Zeile um zwei weitere Bits nach links verschoben. Dadurch betrug der Inhalt von GR0 das achtfache des ursprünglichen Inhalts. Durch Addition des Werts von WK ergab sich dann eine Erhöhung um das 10-fache.
Es folgt ein Beispiel zur Ausführung eines Objektes nach dem Assemblieren.

Tastenbetätigung

Asmbl	Asmbl/Source/Cal
A	8:Go/Dump/Source/Cal
G	Go 32 _
EXE	GR0 (10) _
1 1 2 5	GR0 (10) 1125_
EXE	GR0 (10) 11250
EXE	8:Go/Dump/Source/Cal

Ein andere Methode für Verwendung des Arbeitsbereiches ist, den Inhalt eines Arbeitsregisters vorübergehend im Arbeitsbereich abzuspeichern, um das Register für einen anderen Prozeß zu verwenden, und nach Beendigung des Prozesses den Inhalt wieder zurück ins Arbeitsregister zu bringen.

```
    :
    :ST    :3:WK   (Speichert vorübergehend den Inhalt von
                    GR3 in WK.)
Prozeß, der GR3 verwendet
    :LD    :3:WK   (Zurückbringen des Inhalts von WK in
                    GR3.)
    :
WK  :RESV  :1      (Arbeitsbereich)
```

11-3 Benutzung von logischen Operationen

In diesem Abschnitt wollen wir uns die Verwendung des Befehls für logisches Produkt, AND, und des Befehls für exklusives OR, EOR, ansehen.

- **Maskenoperation**

Eine typische Methode zur Verwendung des AND-Befehls ist, Informationen von einer bestimmten Bitposition eines Arbeitsregisters zu holen. Diese Art Operation wird üblicherweise "Maskenoperation" genannt, weil alle Informationen in nicht benötigten Bitpositionen durch Setzen auf 0 "unsichtbar gemacht werden".
Für die Maskenoperation wird zuerst ein 1-Wort-Datum (Maskendaten genannt) in den Hauptspeicher gespeichert, wobei die gewünschte Bitposition gleich 1 und alle anderen Bitpositionen gleich 0 gesetzt werden. Danach wird das AND des Arbeitsregisters und der Maske berechnet.

Beispiel:

In dieser Operation wollen wir die Bits in geraden Bitpositionen (Bit-Nummern 0, 2, 4, 6, 8, 10, 12 und 14) in Arbeitsregister GR1 erhalten und alle anderen Bits auf 0 setzen. Dann muß die Maske für ein Wort wie folgt aussehen:

```
1010101010101010
```
(AAAA in hexadezimaler Darstellung)

Es folgt ein Beispielsprogramm mit Maskenoperation.

```
      :START:32
BGN:READ  :1:16   (Eingabe eines Hexadezimalwerts in GR1.)
      :AND   :1:MSK  (Maskenoperation)
      :WRITE:1:16   (Ausgabe des Inhalts von GR1 in hexa-
                     dezimaler Darstellung.)
      :HJ    :0:BGN  (Stoppt das Programm.)
MSK:CONST:AAAA   (Maske)
      :END   :BGN
```

Es folgt die Ausführung des Objekts nach dem Assemblieren.

Tastenbetätigung

Asmbl	Asmbl/Source/Cal
A	8:Go/Dump/Source/Cal
G	Go 32 _
EXE	GR1 (16) _
F 5 A C	GR1 (16) F5AC
EXE	GR1 (16) A0A8
EXE	8:Go/Dump/Source/Cal

Der in GR1 gesetzte Hexadezimalwert F5AC sieht ausgedrückt als binärer Bitstring wie folgt aus:

```
1111010110101100
```


Nach Beibehaltung der Bits in den Positionen mit geraden Nummern und Nullsetzen aller anderen Bits sieht der String wie folgt aus:

```
1010000010101000
 A    0    A    8
```
(hexadezimale Darstellung)

- **Komplement zu 2**

In diesem Simulator werden negative ganze Zahlen als "Komplement zu 2" verarbeitet. Das "Komplement zu 2" eines 16-Bit-Wortes bedeutet den Zustand, in dem 0 und 1 umgedreht werden und 1 zu allen Bits addiert wird.

In der Assemblersprache dient der EOR-Befehl zum invertieren der Bits, um das "Komplement zu 2" zu erhalten. Es folgt ein Programm zur Berechnung des "Komplements zu 2" des Inhalts des Arbeitsregisters GR0.

Beispiel:

```
      :START:32
BGN:READ  :0:10   (Eingabe eines Dezimalwerts in GR0.)
      :EOR   :0:AL1  (Invertieren aller Bits von GR0.)
      :ADD   :0:ONE  (Addieren von 1 zu GR0.)
      :WRITE:0:10   (Ausgabe des Inhalts von GR0 in dezimaler
                     Darstellung.)
      :HJ    :0:BGN  (Stoppt das Programm.)
AL1:CONST:FFFF   (Daten zum Invertieren der Bits)
ONE:CONST:0001   (Daten zum Addieren von 1)
      :END   :BGN
```

Es folgt die Ausführung des Objekts nach dem Assemblieren.

Tastenbetätigung

Asmbl	Asmbl/Source/Cal
A	8:Go/Dump/Source/Cal
G	Go 32 _
EXE	GR0 (10) _
1 2 3	GR0 (10) 123_
EXE	GR0 (10) -123
EXE	8:Go/Dump/Source/Cal
G	Go 32 _
EXE	GR0 (10) _
− 4 5	GR0 (10) -45
EXE	GR0 (10) 45
EXE	8:Go/Dump/Source/Cal

Im obigen Beispiel wird das "Komplement zu 2" für **123** und **-45** berechnet. Bei Anzeige in dezimaler Darstellung erfolgt eine entsprechende Vorzeichenumkehrung. Der Grund, aus dem Bitumkehrung durch das exklusive OR der hexadezimalen Daten **FFFF** möglich ist, ist, daß beim exklusiven OR (ausgedrückt durch ⊕), $1 \oplus 1 = 0$ und $0 \oplus 1 = 1$ gilt.

Mit dem ADD-Befehl wird nach der Bitumkehrung 1 zum Inhalt von GR0 addiert. Der vorher erwähnte LAI-Befehl ist nicht fähig, GR0 um 1 zu erhöhen, und dieser Befehl ist nicht geeignet, um 1 zu GR1, GR2 oder GR3 auf diese Weise mit einer ganzen Zahl mit 16 Bit und Vorzeichen zu addieren.

11-4 Vergleichung

Zum Vergleichen der Größe von zwei Zahlen A und B wird normalerweise nach Berechnung von A − B das arithmetische Vorzeichen (plus, 0 oder minus) betrachtet. Mit dem Simulator und Programmierung in Assemblersprache wird dies durch Kombination der Befehle SUB, JC und JNZ durchgeführt.

Beispiel:
Im folgenden Programm wird der in das Arbeitsregister GR0 gesetzte Wert mit der Dezimalzahl 50 (0032 in hexadezimaler Darstellung) verglichen.

Der Inhalt von GR0 ist die Dezimalzahl 100, wenn der Inhalt von GR0 > 50 ist.
Keine Änderung, wenn der Inhalt von GR0 = 50 ist.
Der Inhalt von GR0 ist 1, wenn der Inhalt von GR0 < 50 ist.

Die Verarbeitung ist für die verschiedenen Fälle aufgeteilt.

```
        :START:32
BGN:    :READ  :0:10    (Eingabe eines Dezimalwertes in GR0.)
        :ST    :0:WK    (Vorübergehendes Abspeichern des Inhaltes
                         von GR0.)
        :SUB   :0:FTY   (Vergleich mit 50.)
        :JC    :1:NG    (Sprung zu NG, wenn kleiner als 50.)
        :JNZ   :0:POS   (Sprung zu POS, wenn größer als 50.)
ZER:    :LD    :0:WK    (Zurücksetzen von GR0 auf den ursprüng-
                         lichen Wert, wenn gleich 50.)
        :JC    :3:OUT   (Sprung zu OUT.)
NG      :LAI   :0:1     (Setzt 1 in GR0.)
        :JC    :3:OUT   (Sprung zu OUT.)
POS:    :LAI   :0:100   (Setzt 100 in GR0.)
OUT:    :WRITE :0:10    (Ausgabe des Inhalts von GR0 in dezimaler
                         Darstellung.)
        :HJ    :0:BGN   (Stoppt das Programm.)
WK      :RESV  :1       (Arbeitsbereich zum vorübergehenden
                         Abspeichern)
FTY:    :CONST :0032    (Vergleichswert, in dezimaler Darstellung 50)
        :END   :BGN
```

Wir bringen jetzt die Ausführung des Objektes nach dem Assemblieren. Beachten Sie, daß die Verarbeitung aufgeteilt wird, wenn 51, 50 und 49 in GR0 gesetzt werden.

Tastenbetätigung

Asmbl	Asmbl/Source/Cal
A	8:Go/Dump/Source/Cal
G	Go 32 _
EXE	GR0 (10) _
5 1	GR0 (10) 51_
EXE	GR0 (10) 100
EXE	8:Go/Dump/Source/Cal
G	Go 32 _
EXE	GR0 (10) _
5 0	GR0 (10) 50_
EXE	GR0 (10) 50
EXE	8:Go/Dump/Source/Cal
G	Go 32 _
EXE	GR0 (10) _
4 9	GR0 (10) 49_
EXE	GR0 (10) 1
EXE	8:Go/Dump/Source/Cal

Durch "SUB:0:FTY" wird 50 vom Inhalt von GR0 subtrahiert, das Ergebnis wird im Anzeigeregister (CC) in Übereinstimmung mit dem Vorzeichen gespeichert. (CC wird 0, wenn das Vorzeichen Plus oder 0 ist, und 1, wenn es Minus ist.) Da der Inhalt von GR0 geändert wird, wird in der Zeile unmittelbar davor "ST:0:WK" ausgeführt, um den Inhalt von GR0 vorübergehend im Arbeitsbereich zu speichern.

Wenn CC gleich 1 ist, springt die Ausführung durch den Befehl für bedingten Sprung, "JC:1:NG", zur Adresse NG. Diese Adresse wird übersprungen, wenn der Inhalt von CC gleich 0 ist. Dann entscheidet "JNZ:0:POS", ob das Ergebnis positiv ist, und, wenn ja, springt zur Adresse POS. Da das Ergebnis 0 ist, wird GR0 durch "LD:0:WK" auf den ursprünglichen Wert zurückgesetzt.

Nach Beendigung der Verarbeitung in Adresse ZER und Adresse NG springt die Ausführung zu Adresse OUT durch den Befehl für unbedingten Sprung "JC:3:OUT".

11-5 Tabellenoperation

Wir wollen jetzt ein Programm in Assemblersprache erstellen, das "Tabellenformat-Daten" verarbeitet. Da die "Tabellenformat-Daten" normalerweise in aufeinanderfolgenden Bereichen des Hauptspeichers gespeichert werden, werden diese Daten als **Tabelle** bezeichnet.

Beispiel:

Wir erstellen ein Programm zur Berechnung der Summe aus fünf Daten in einer Tabelle, beginnend von Adresse TBL. Durch Verwendung von Adressenmodifikation mit dem Indexregister bei der Tabellenberechnung kann das Programm kürzer gemacht werden. Im unten dargestellten Programm wird die Summe in Register GR0 gesetzt, GR1 dient als Indexregister.

```
      :START:32
BGN   :LAI   :0:0          (Löscht GR0.)
      :LAI   :1:4          (Setzt 4 in GR1.)
LP    :ADD   :0:TBL:1      (Addiert unter Verwendung des Indexregisters.)
      :LAI   :1:255:1      (Vermindert GR1 um 1.)
      :JNZ   :1:LP         (Durchläuft eine Schleife, wenn GR1 ungleich 0 ist.)
      :ADD   :0:TBL        (Addiert den Inhalt der Adresse TBL am Ende.)
      :WRITE:0:10          (Ausgabe eines Wertes in dezimaler Darstellung.)
      :HJ    :0:BGN        (Stoppt das Programm.)
TBL   :CONST:0008
      :CONST:0002
      :CONST:0006          Tabelle
      :CONST:0001
      :CONST:0003
      :END   :BGN
```

Setzen Sie den Anfangswert 4 in Indexregister GR1. Da der Inhalt von GR1 durch den LAI-Befehl jedesmal, wenn die Schleife beginnend ab Adresse LP durchlaufen wird, um 1 vermindert wird, ändert sich die effektive Adresse von "ADD:0:TBL:1" wie folgt, und die Tabellendaten werden beginnend mit dem letzten Wert nacheinander zu GR0 addiert.

Adresse TBL+4
Adresse TBL+3
Adresse TBL+2
Adresse TBL+1

Die Ausführung springt aus der Schleife, wenn der Inhalt von GR1 gleich 0 wird. Da die Adress-TBL-Daten noch vorhanden sind, wird hier durch "ADD:0:TBL" addiert. Auf diese Weise wird die Gesamtsumme der fünf Daten in der Tabelle, 20, in GR0 gesetzt.
Dieser Art Tabellenoperation entspricht die Verwendung von Feldvariablen in BASIC.

11-6 Erzeugung und Verwendung von Unterprogrammen

Der schwierigste Befehl des Simulators ist der Unterprogramm-Sprungbefehl JSR. Da jedoch JSR der einzigste Befehl ist, der den Inhalt des Basisregisters (BR) ändern kann, ist es für Programmieren in Assemblersprache sehr wichtig, die Verwendung dieses Befehls gut zu verstehen.

Beispiel:
Wir wollen ein Unterprogramm (mit dem Label DIV) erstellen, das den Inhalt des Arbeitsregisters GR2 durch den Inhalt des Arbeitsregisters GR3 teilt, den Quotienten in GR2 und den Divisionsrest in GR3 setzt und dann zum Hauptprogramm zurückkehrt. Dabei wird das Hauptprogramm in Speicherblock Nr. 0 und das Unterprogramm in Speicherblock Nr. 1 gespeichert.


```
      :START:32
BGN   :READ  :2:10    (Eingabe eines Dezimalwerts in GR2.)
      :READ  :3:10    (Eingabe eines Dezimalwerts in GR3.)
      :JSR   :0:SB    (Setzt die Rückkehradresse in GR0 und
                       springt zum Unterprogramm.)
      :WRITE:2:10     (Ausgabe des Quotienten von GR2 in dezi-
                       maler Darstellung.)
      :WRITE:3:10     (Ausgabe des Divisionsrestes von GR3 in
                       dezimaler Darstellung.)
      :HJ    :0:BGN   (Stoppt das Programm.)
SB    :ADCON:DIV      (Definiert eine Unterprogramm-Adresse
      :END   :BGN      für Label SB.)

DIV:START:256
      :ST    :0:SAV   (Speichert die Rückkehradresse in Adresse
                       SAV.)
      :ST    :2:A
      :ST    :3:B
      :LD    :3:A
      :LAI   :2:0
LP    :SUB   :3:B
      :JC    :1:ADJ
      :ADD   :2:ONE
      :JC    :3:LP
ADJ:ADD      :3:B
RET:JSR      :0:SAV
SAV:RESV     :1
A   :RESV    :1
B   :RESV    :1
ONE:CONST:0001
      :END
```

Im Hauptprogramm, das ab Adresse 32 im Speicherblock Nr. 0 gespeichert ist, werden Werte in GR2 und GR3 eingegeben und werden die Werte, die vom Unterprogramm übergeben werden, ausgegeben.
Der Sprung zum Unterprogramm erfolgt durch den Befehl "JSR:0:SB". Die Rückkehradresse (Adresse 35, in der "WRITE:2:10" gespeichert ist, ist die Rückkehradresse) vom Unterprogramm wird durch diesen Befehl in Arbeitsregister GR0 gesetzt, und der durch das Label SB definierte Adressenwert wird im Ablaufzähler gespeichert. Gleichzeitig wird der Inhalt des Basisregisters (BR) in Adresse 0100 (hexadezimale Darstellung) von Speicherblock Nr. 1 gesetzt. Der Sprung wird zur Adresse 256 in Speicherblock Nr. 1 ausgeführt, da 256, die erste Adresse des Unterprogramms, durch den Pseudobefehl ADCON im Label SB definiert ist.
Wir wollen jetzt den Ablauf des Unterprogramms ab Adresse 256 erklären. Obwohl das Arbeitsregister GR0 nicht im Unterprogramm verwendet wird, wird aus Sicherheitsgründen die Rückkehradresse in GR0 durch "ST:0:SAV" in den Arbeitsbereich SAV gesichert. (SAV eines Labels bedeutet Sichern.) Das "Sichern der Rückkehradresse" auf diese Weise sollte beachtet werden, wenn in diesem Simulator ein Unterprogramm in Assemblersprache verwendet wird. Im Unterprogramm wird zuerst der Inhalt von GR2 durch den Arbeitsbereich A in GR3 übertragen, und der ursprüngliche Wert von GR3 wird im Arbeitsbereich B gespeichert. Dieser Prozeß wird ausgeführt, um den Divisionsrest in GR3 bei der Rückkehr in das Hauptprogramm zu speichern.
Die Ausführung wird in der Divisionsschleife (beginnend ab Adresse LP) auf diese Weise mit GR2 als Zähler wiederholt. In der Schleife wird der Wert des Arbeitsbereichs B von dem Wert von GR3 subtrahiert. Die Schleife wird solange durchlaufen, als das Ergebnis positiv oder 0 ist, und die Anzahl der Durchläufe (GR2) wird der Quotient. Wenn der Wert von GR3 - B negativ wird, spring die Ausführung durch den bedingten Sprung "JC:1:ADJ" aus der Schleife und springt zur Adresse ADJ (Einstell-Label). Der Wert von B, vom dem 1 zuviel subtrahiert wurde, wird hier durch "ADD:3:B" korrigiert, und der korrekte Divisionsrest wird in GR3 gesetzt.

"JSR:0:SAV" der Adresse RET dient zur Rückkehr vom Unterprogramm in das Hauptprogramm. Die in den Arbeitsbereich SAV gesicherte Rückkehradresse wird in den Ablaufzähler (SC) gesetzt, und ein Sprung zur Rückkehradresse (Adresse 35) in Speicherblock Nr. 0 wird auf genau die gleiche Weise wie der Sprung zum Unterprogramm durchgeführt.

Hinweis:
Die Erhöhung des Inhalts von GR2 um 1 wird durch "ADD:2:ONE" in der Divisionsschleife des Unterprogramms durchgeführt. Da der Inhalt von GR2 auf 0 bis 255 beschränkt ist, wenn dieser Befehl durch "LAI:2:1:2" ersetzt wird, ist diese Ersetzung in diesem Fall nicht geeignet. (Siehe das Ausführungsbeispiel unten.)

Es folgt die Ausführung des Objektes nach dem Assemblieren.

Tastenbetätigung

Asmbl	Asmbl/Source/Cal
A	8:Go/Dump/Source/Cal
G	Go 32 _
EXE	GR2 (10) _
23456	GR2 (10) 23456_
EXE	GR3 (10) _
78	GR3 (10) 78_
EXE	GR2 (10) 300
EXE	GR3 (10) 56
EXE	8:Go/Dump/Source/Cal

In diesem Beispiel wird 23456 ÷ 78 = 300 mit einem Divisionsrest von 56 berechnet.

Stichwortverzeichnis