Klaus Bovermann

2. September 2019

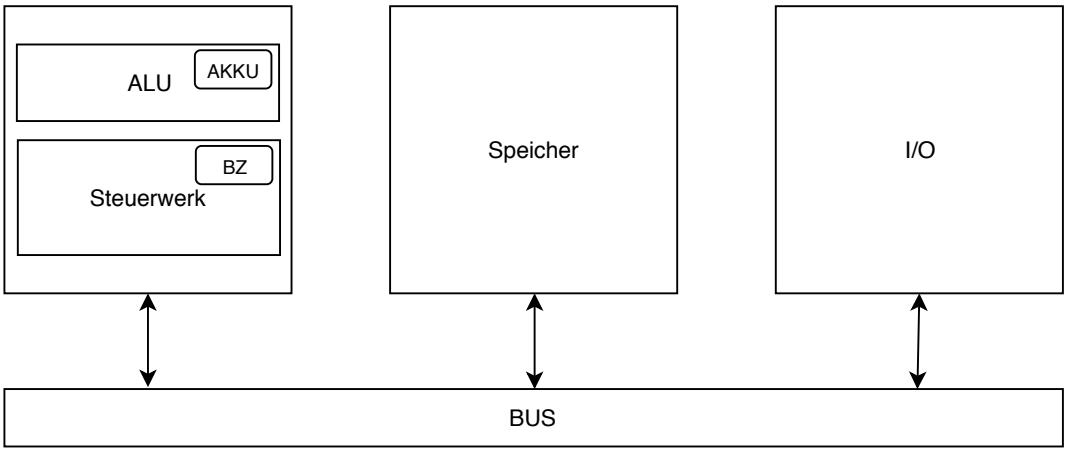
Von-Neumann-Rechner

Simulationssoftware einer einfachen Maschine

Eine Unterrichtseinheit in der Qualifikationsstufe im Fach Informatik beschäftigt sich mit den grundlegenden Prinzipien der von-Neumann-Architektur. Wesentliche Bestandteile einer auf diesen Prinzipien basierenden Hardware sind:

- Speicher
- Recheneinheit (ALU) mit dem Akkumulator (AKKU)
- Steuerwerk mit einem Befehlszähler (BZ)
- Ein- und Ausgabeeinheit (I/O) als Verbindung zur Außenwelt

Diese Bauteile sind über einen sog. Bus miteinander verbunden:



Der Speicher

besteht aus einer Anzahl von Speicherzellen. Jede Speicherzelle besteht aus einer für die Maschine charakteristischen Anzahl (im Folgenden Breite genannt) von Binärzellen, die jeweils eine Null (0) oder eine Eins (1) verwalten können.

Jede Speicherzelle ist über eine Adresse ansprechbar. Man kann über diese Adresse den Inhalt der Zelle (also ein Binärwort) lesen oder ein Binärwort hineinschreiben.

In der hier vorgestellten Simulation gibt es $2^{12} = 4096$ Speicherzellen, die alle eine Breite von 16 Bit haben. Die kleinste zulässige Adresse ist 0, die größtmögliche Adresse also 4095.

Die Speicherbreite von 16 hat zur Folge, dass die kleinste zu verwaltende Binärzahl also 0000000000000000 (dezimal 0) ist; die größte Zahl ist 1111111111111111, dezimal umgerechnet also $2^{16} - 1 = 65535$.

Die ersten 4 Bit werden als Befehlscode genutzt, die letzten 12 Bit werden als Zahl interpretiert. Diese Zahl kann als Adresse oder als Datum interpretiert werden (s.u.).

Der Akkumulator (AKKU)

ist ebenfalls eine Speicherzelle, die sich jedoch nicht im Speicher, sondern in der Recheneinheit (ALU) befindet. Der Akkumulator hat also insbesondere dieselbe Breite (hier also 16) wie die Speicherzellen im Speicher.

Das Besondere dieser Speicherzelle besteht darin, dass Rechenoperationen dort ausgeführt werden können. So kann man z.B. den Inhalt des Akkus um den Inhalt einer Speicherzelle erhöhen oder verringern. Dabei werden die Inhalte als (binär dargestellte) natürliche Zahlen interpretiert. Die dabei ggf. auftauchenden binären Über- oder Unterläufe werden ignoriert.

Der Befehlszähler (BZ)

ist eine Adresse, die vom Steuerwerk verwaltet wird. Wird eine von-Neumann-Maschine gestartet, hat der Befehlszähler den Wert 0. Der Befehlszähler wird vom Steuerwerk nach bestimmten Regeln verändert.

Das Steuerwerk

ist dafür verantwortlich, diejenige Speicherzelle zu betrachten, deren Adresse gerade in dem Befehlszähler steht. Der Inhalt dieser Speicherzelle ist eine 16-stellige Binärzahl, die von der Steuereinheit in zwei Teile zerlegt wird:

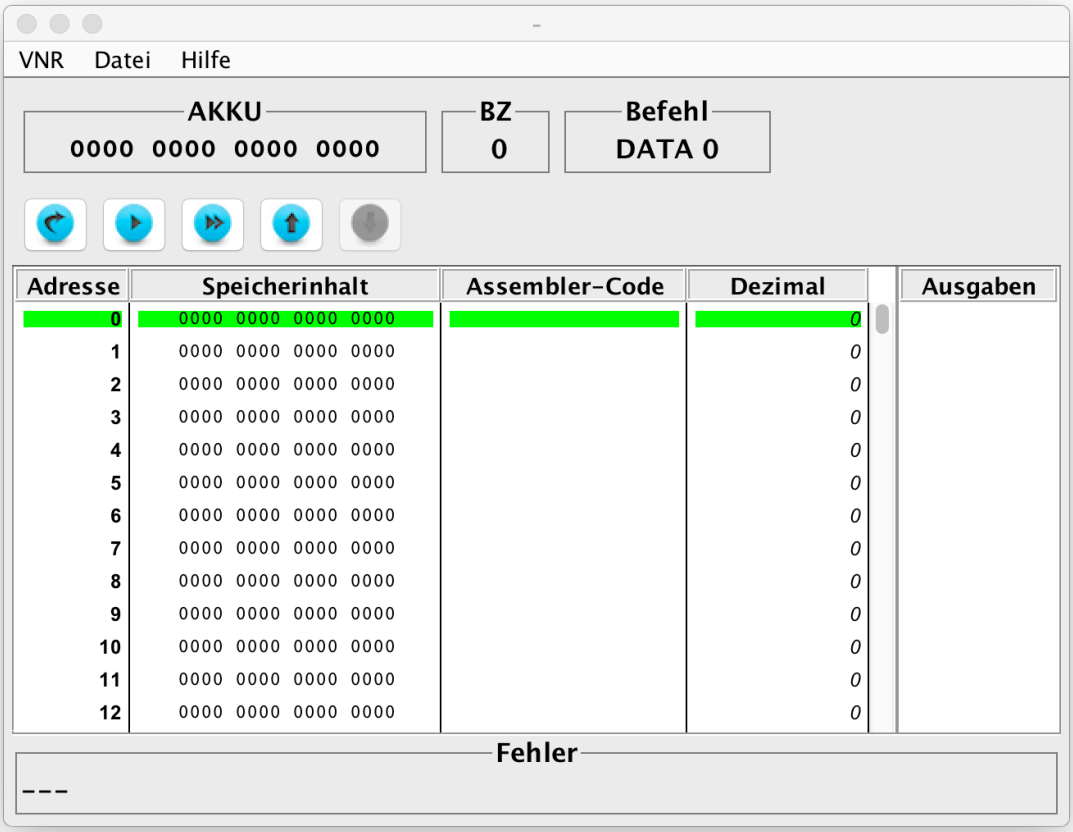
- Die ersten vier Binärstellen codieren einen Befehl, dessen Semantik noch genauer erläutert wird.
- Die restlichen 12 Binärstellen werden als Adresse einer Speicherzelle interpretiert, auf die sich der Befehl bezieht.

Dieser Befehl wird dann vom Steuerwerk ausgeführt.

Das Simulationsprogramm VNR

Um die Funktionalität einer von-Neumann-Maschine testen zu können, wurde eine Java-Anwendung entwickelt, die sich auf die notwendigste Funktionalität beschränkt. Bewusst wurde im Sinne einer didaktischen Reduktion auf viele Features (indirekte Adressierung, Micro-Schritte, …) verzichtet. Wenn dennoch weitere Funktionalitäten gewünscht werden, ist es sinnvoll, einen der im Anhang beschriebenen Simulatoren zu nutzen.

Die Oberfläche der Software (der Speicher enthält noch kein Programm):



Optionen

Das Verhalten des Programms kann mit Hilfe einer Konfigurationsdatei (`VNR.ini`) angepasst werden:

- Die Sprache der Oberfläche ist wahlweise Deutsch (de) oder Englisch (en). Weitere Sprachanpassungen könnten folgen.
- Die dritte (Assembler-Code) und auch die vierte Spalte (dezimale Darstellung des Speicherinhalts) kann zum Editieren freigegeben werden.
- Jede Zelle eines leeren Speichers kann wahlweise mit NOP (s.u.) belegt werden.

Dazu müssen die entsprechenden Einträge in der o.g. Konfigurationsdatei geändert und das Programm erneut gestartet werden.

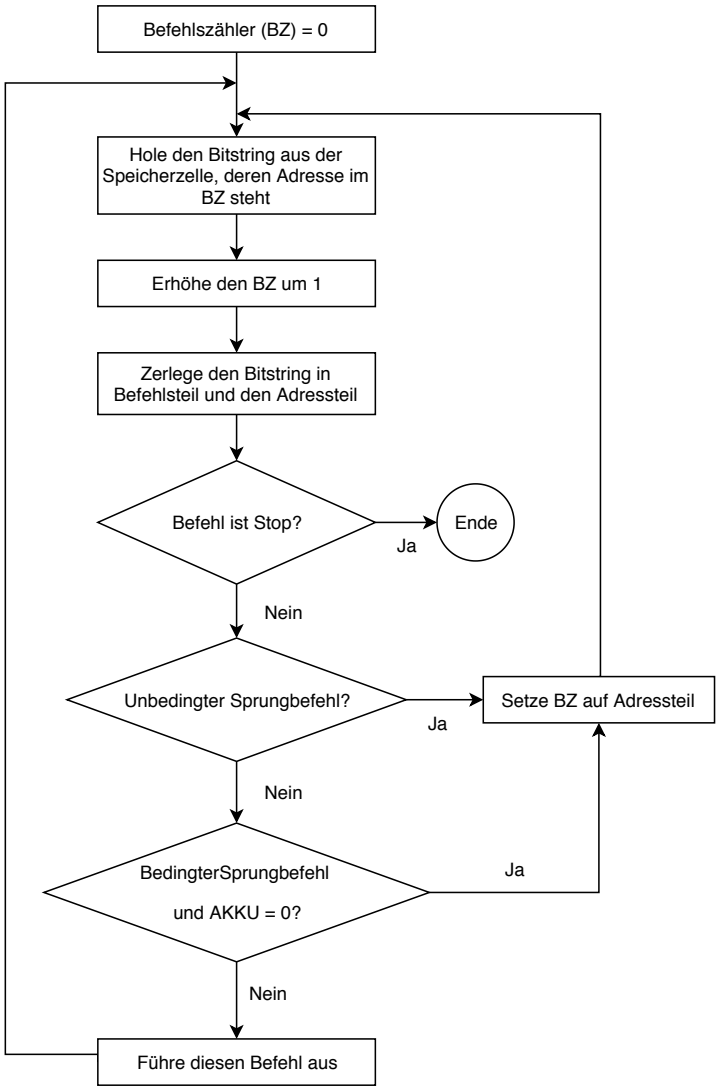
Die Semantik der Befehle

wird aus der folgenden Tabelle ersichtlich. Der in der linken Spalte angegebene Binärcode stellt die ersten 4 Stellen des Speicherinhalts dar.

Bin-Code	Assembler	Beschreibung
0000	DATA	DATA ungültiger Befehl; wird benutzt, um Speicherzellen mit einem Symbol zu markieren
0001	LINT	LINT xx Der Wert xx wird als positive ganze Zahl in den AKKU geschrieben. BZ wird um 1 erhöht.
0010	LOAD	LOAD xx Inhalt von Zelle xx wird in den AKKU transportiert. BZ wird um 1 erhöht.
0011	STORE	STORE xx Inhalt des AKKUs wird in die Zelle xx transportiert. BZ wird um 1 erhöht.
0100	ADD	ADD xx Inhalt der Zelle xx wird zum AKKU addiert. BZ wird um 1 erhöht.
0101	SUB	SUB xx Inhalt der Zelle xx wird vom AKKU subtrahiert. BZ wird um 1 erhöht.
0110	MUL	MUL xx Akku wird mit dem Inhalt der Zelle xx multipliziert. BZ wird um 1 erhöht.
0111	DIV	DIV xx Akku wird durch den Inhalt der Zelle xx dividiert. BZ wird um 1 erhöht.
1000	MOD	MOD xx Akku erhält den Rest der Division von AKKU durch den Inhalt der Zelle xx. BZ wird um 1 erhöht.
1001	JMP	JUMP xx Der Befehlszähler wird auf xx gesetzt.
1010	JEQ	JEQ xx Der Befehlszähler wird auf xx gesetzt, falls AKKU == 0 ist. Ansonsten wird BZ um 1 erhöht.
1011	IN	IN Der Inhalt des Eingabefensters wird in den AKKU transportiert. BZ wird um 1 erhöht.
1100	OUT	OUT Der Inhalt des AKKUs wird ausgegeben BZ wird um 1 erhöht.
1101	STOP	STOP Das Programm hält an.
1110	NOP	NOP Keine Operation. BZ wird um 1 erhöht.

Der Programmablauf

vollzieht sich in dem sogenannten von-Neumann-Zyklus:



von-Neumann-Zyklus

1. Die Steuereinheit lädt den binär-codierten Inhalt der Speicherzelle, die vom BZ adressiert ist und zerlegt sie, wie oben dargestellt.
2. Der BZ wird um 1 erhöht.
3. Der durch die ersten 4 Bit codierte Befehl wird - sofern gültig - ausgeführt.

Diese drei Schritte werden solange wiederholt, bis ein Stop-Befehl erreicht ist.

Befehlsausführung

Die Steuereinheit hat in jedem Schritt den Inhalt der aktuellen Speicherzelle in die beiden Teile (4 Bit als Codierung des Befehls, 12 Bit als zugehörige Adresse) zerlegt. Die (willkürliche) Zuordnung der 4-bit-breiten Codierung zu den in der obigen Tabelle angegebenen Befehlen veranlasst die Steuereinheit zu der entsprechenden Reaktion.

Dabei kann es vorkommen, dass der 4-bit-breite Code zu einem ungültigen Befehl gehört, denn nicht alle 16 Möglichkeiten codieren einen gültigen Befehl. Auch wenn beim Start eines Programms alle Zellen gültige Inhalte haben, kann es vorkommen, dass das Programm selber eine Zelle verändert, die die Steuereinheit später als Befehl interpretieren soll.

Erzeugen, Speichern und Laden von binär-codierten Programmen

Nach dem Öffnen des Simulators hat man schreibenden Zugriff auf die Speicherzellen der von-Neumann-Maschine (Spalte 2). Die Syntax einer Speicherzelle verlangt die Eingabe einer 16-stelligen 0-1-Zeichenkette.

Den Inhalt des Speichers kann man als Textdatei mit der Endung **`mem`** abspeichern. Dabei werden die 16-bit-breiten 0-1-Zeichenfolgen aller 4096 Speicherzellen zeilenweise in die Datei geschrieben.

Eine Textdatei mit dieser Endung kann in den Speicher geladen werden. Die Zeilen dieser Datei bestehen aus 16-bit-breiten 0-1-Zeichenfolgen. Leerzeichen in den Zeilen werden ignoriert. Es müssen nicht unbedingt 4096 Zeilen vorhanden sein; restliche Speicherzellen werden initialisiert mit 0000 0000 0000 0000.

Assembler; Erzeugen, Speichern und Laden

Die Zuordnung der 4-bit-breiten Codierungen zu den in der obigen Tabelle angegebenen Befehlen ist (natürlich) eindeutig. Um Programme lesbarer schreiben zu können, werden Speicherinhalte auch in lesbarer Form angegeben und können in dieser Form eingelesen und gespeichert werden.

Ein auf diese Art codiertes Programm ist ein sog. *Assembler-Programm*; den Übersetzer, der ein solches Programm in eine Folge von 0-1-Zeichenfolgen übersetzt, nennt man einen *Assembler*.

Speichert man den Speicherinhalt des Simulators als Assembler-Programm ab, wird eine Textdatei mit der Endung **`ass`** erzeugt. Die Zeilen haben die Form

`Assemblercode Adresse`

Die Syntax der Assemblerbefehle ist weiter unten angegeben.

Jede Zelle des Speichers wird als Assemblerzeile in der Datei abgelegt, obwohl nicht alle 4096 Zellen im Programm angegeben werden müssen. Das hat zur Folge, dass ein Assemblerprogramm mit wenigen Zeilen, das geladen wurde, beim Speichern in einem 4096-zeiligen Assemblerprogramm ausgegeben wird.

Nach dem Starten des Programms hat man optional (siehe Optionen) schreibenden Zugriff auf den Assemblercode, so dass man statt der Eingabe von Bitstrings in der zweiten Spalte auch Assemblerbefehle in der dritten Zeile eingeben kann.

Es ist möglich, mit einem Texteditor ein Programm zu erzeugen, das - mit der Endung **`ass`** versehen - Assembler-Code enthält und dann in den Speicher zu laden. Hier müssen nicht notwendigerweise alle 4096 Zeilen geschrieben werden; restliche Speicherzellen werden initialisiert mit 0000 0000 0000 0000.

Symbolische Assembler-Adressen

Imperative Programme bestehen aus einer Folge von Befehlen, die Daten manipulieren. Höhere Programmiersprachen kennen das Konzept der Variablen, die dort mit Namen versehen sind, über den man lesend und schreibend auf den Wert zugreifen kann.

In der von-Neumann-Architektur sind die Befehle codiert in den Speicherzellen abgelegt. Wie bereits erwähnt, codieren die ersten 4 Bit den Befehl, die restlichen 12 Bit verwalten die Adresse, auf die sich der Befehl bezieht.

Erstellt man ein Assembler-Programm, müssen die Befehle die Form

`Assemblercode Adresse`

haben. Die Adressen müssen dabei Zahlen aus dem Bereich [0; 4095] sein. Jedoch ist es benutzerfreundlicher, wenn man sog. *symbolische Adressen* verwenden kann. Fügt man z.B. in ein Assemblerfile eine Zeile ein, müssten ansonsten alle Adressen angepasst werden.

So kann z.B. ein Programm die Speicherzelle mit der Adresse 42 nutzen, um eine Zahl zu verwalten. Sinnvoll ist es nun, Speicherzellen mit einem Namen zu versehen. Statt also z.B. **`Load 42`** zu schreiben, verabredet man, dass die Speicherzelle 42 den (symbolischen) Namen **`n`** erhält, so dass jetzt der Befehl **`Load n`** erlaubt ist.

In den Assembler-Programmen sind solche symbolischen Adressen nutzbar; die Kleinbuchstaben a, b, … z stehen dafür zur Verfügung.

In der dritten Spalte des Simulators sind solche symbolischen Adressen natürlich nicht(!) nutzbar, da den Speicherzellen dort keine symbolischen Adressen zugeordnet sind.

Um in einem Assembler-Programm Speicherzellen mit einem Namen zu versehen, muss die entsprechende Zeile eine sog. Marke erhalten. Dazu beginnt eine solche Zeile mit dem gewünschten Kleinbuchstaben, gefolgt von einem Doppelpunkt (`:`).

Kommentare

Um die Lesbarkeit eines Assembler-Programms noch weiter zu erhöhen, ist es erlaubt Kommentarzeilen und Leerzeilen zu nutzen. Alle leeren Zeilen, sowie Zeilen, die mit einem Semikolon (`;`) beginnen, werden beim Einlesen ignoriert. Darüber hinaus kann am Ende einer jeden Zeile ein Kommentar stehen, der mit einem Semikolon abgetrennt ist.

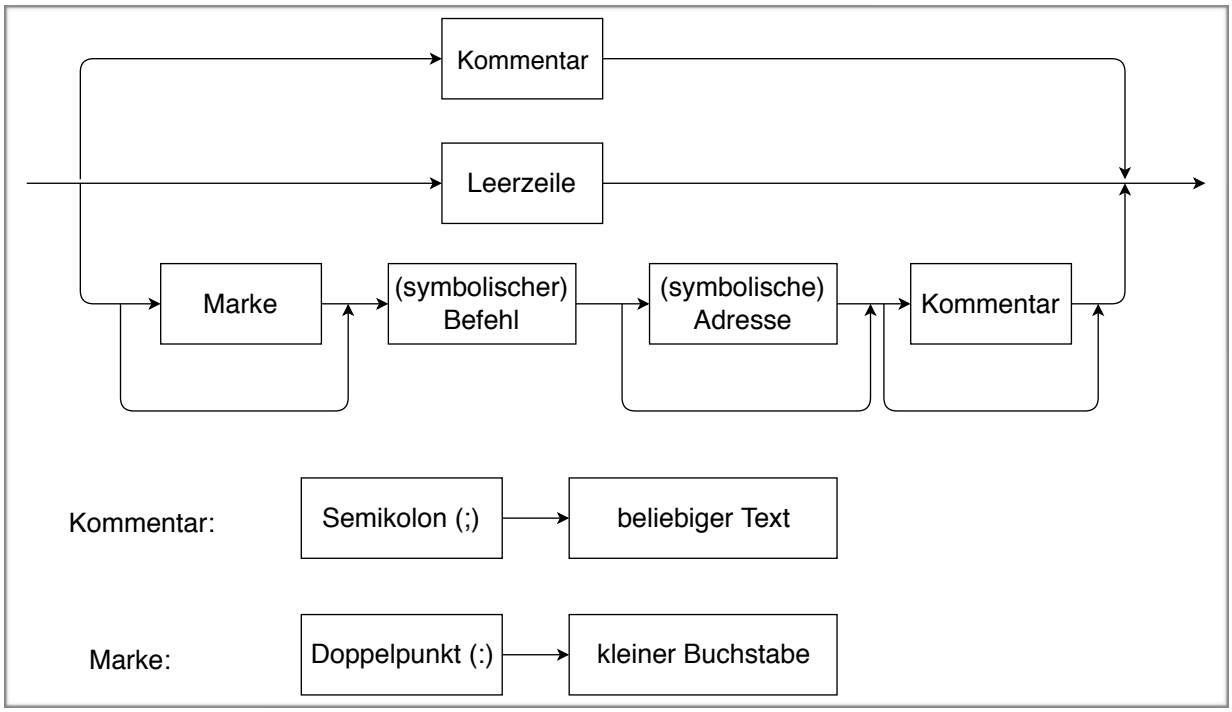
Assembler-Syntax

Insgesamt hat eine gültige Assembler-Zeile die Form

Marke: **Befehl** ***Adresse*** ***;Kommentar***

wobei die kursiv dargestellten Bestandteile optional sind. Ein Leerzeichen ist nur zwischen Befehl und Adresse notwendig. Eine ggf. fehlende Adresse wird durch 0 ergänzt.

Die Syntax ist ersichtlich aus dem folgenden Diagramm:



Ausführen eines Programms

Nachdem ein von-Neumann-Programm in die Maschine geladen ist, kann das Programm gestartet werden.

- Für das Programm benötigte Parameter bzw. Konstanten können direkt in den Speicher eingegeben, als Konstanten geladen oder von der Konsole gelesen werden:
  - Soll das Programm z.B. das Quadrat einer Zahl z berechnen, und hat der Programmierer als Speicherzelle für z die 7-te Zelle vorgesehen, so kann man "per Hand" die zu quadrierende Zahl in die 7-te Zelle eintragen.
  - Alternativ kann ein Integer-Wert mit dem Assembler-Befehl **`LINT xxx`** gesetzt werden. Dabei wird **`xxx`** als Integer interpretiert und in den Akku geschrieben.
    Im Beispiel findet sich also in dem Assembler die Zeile **`LINT 7`**.
  - Eine weitere Alternative bietet der Assembler-Befehl **`IN`**. Trifft der Simulator auf diesen Befehl, hält die Abarbeitung an und es öffnet sich ein Eingabe-Feld, in das man eine Zahl schreiben kann. Setzt man die Abarbeitung fort (im schrittweisen Modus durch Klick auf den entspr. Button, im automatischen Modus durch RETURN), wird der Wert in den Akku geschrieben.
- Der Reset-Button  setzt den Befehlszähler auf 0 und löscht das Ausgabefenster.
- Das Programm kann schrittweise mit  ausgeführt werden. Dabei werden im oberen Teil des Programmfensters der Inhalt des Akkus, der Index des als Nächstes auszuführenden Befehls sowie die binäre Darstellung des als Nächstes auszuführenden Befehls angezeigt. Der ToolTip des AKKU-Labels zeigt den Inhalt dezimal an.
- Das Programm kann mit  ab der aktuellen Zelle auch automatisch ablaufen. Die Abarbeitung läuft solange, bis der **`STOP`**-Befehl erreicht ist. Eine Endlosschleife kann durch RESET abgebrochen werden.
- Man kann die momentane Speicherbelegung jederzeit mit  (intern) speichern.
- Diesen Speicherzustand kann man dann mit dem Button  wiederherstellen.

Ein Beispiel

Das folgende Programm soll zwei Zahlen (in diesem Fall 7 und 8) miteinander multiplizieren und das Ergebnis ausgeben, wobei nur Addition und Subtraktion erlaubt sind.

Ein mögliches Assembler-Programm dazu:

```
        ;Multipliziere die Werte a=7 und b=8
        ;und gib das Produkt aus

        ;a=7
        lint 7
        store a

        ;b=8
        lint 8
        store b

        ;e=1
        lint 1
        store e

        load b
        store y
        load r;

        resultat = 0
        sub r
        store r

        s:load y; schleifenanfang
        jeq z
        load r; r = r + a
        add a
        store r
        load y; y = y - 1
        sub e
        store y
        jmp s; rücksprung

        z: load r
        out
        stop

        a: Data
        b: Data
        e: Data ; eins
        r: Data ; resultat
        y: Data ; Kopie von b
```