Projekt: von-Neumann-Architektur

Einleitung

Jeder von uns kommt täglich mit Computern in Berührung. Das geschieht sogar sehr häufig, ohne dass man sich dessen bewusst ist:




All diese Maschinen sind vom Hersteller programmiert und man kann oft Programm-Updates (sogenannte Firmware-Updates) der Hersteller installieren.

Manche Maschinen können jedoch auch von uns (um-)programmiert oder auch konfiguriert werden. Dazu benötigt man eine *Sprache*, in der man sich mit der Maschine sozusagen „unterhalten“ kann.

Möchten wir als Kunde z.B. die Waschmaschine programmieren (beispielsweise 30-Grad-Buntwäsche und schleudern mit 1000 U/min), so haben wir dazu Bedienelemente, Tastaturen, Displays, Drehregler o.ä. an der Maschine, mit denen wir eine Art Programm eingeben können, das dann (nach dem Start) von einem Prozessor abgearbeitet wird.

Eine solche Waschmaschinen-Konfiguration ist also formuliert in einer (hoffentlich anwenderfreundlichen) sehr hohen Programmiersprache.

Jedoch kann der Prozessor (in der Waschmaschine, im Camcorder, aber auch der in einem PC) solche Programme nicht „verstehen“. Sie müssen in eine maschinennahe Sprache übersetzt werden, die für den Prozessor verständlich ist.

Rechnerarchitektur

Die Syntax, aber noch mehr die Semantik einer solchen Sprache muss die Struktur, den inneren Aufbau einer Maschine, berücksichtigen, sie muss also die Architektur der Maschine kennen.

> ***Rechnerarchitektur*** *ist ein Teilgebiet der Technischen Informatik, das sich mit dem Design von Rechnern (Computern) und speziell mit deren Organisation sowie dem externen und internen Aufbau (was ebenfalls mit 'Rechnerarchitektur' bezeichnet wird) beschäftigt.*
> *Wir verstehen darunter*
> - *die interne Struktur des Rechners.*
> - *den Aufbau des Rechners aus verschiedenen Komponenten.*
> - *die Organisation der Arbeitsabläufe .*

Heute gibt es diverse Rechner-Architekturen, die sich zum Teil erheblich voneinander unterscheiden. Jedoch gibt es einige Gemeinsamkeiten:

- Daten werden (in der Regel binär) in adressierbaren Speicherzellen verwaltet.
- Das Programm besteht aus einer Folge von Befehlen, die ebenfalls in den Speicherzellen aufbewahrt werden.
- Es gibt ein oder mehrere Rechenwerke, in denen elementare Rechenoperationen ausgeführt werden können.
- Ein Steuerwerk kontrolliert den Ablauf des Programms.

Neben der Harvard- und vielen weiteren Architekturen basieren sehr viele Computer auf der von-Neumann-Architektur. Sie ist benannt nach dem ungarisch-US-amerikanischen Mathematiker John von-Neumann (1903 – 1957), der grundlegende Prinzipien einer Rechnerarchitektur entwickelt hat.

Diese Architektur wird im Folgenden näher beleuchtet.

Die von-Neumann-Architektur

In der Welt der höheren Programmiersprachen, insbesondere in imperativen, objektorientierten Sprachen wie Java, basieren die Programmabläufe auf Variablen, die gelesen und verändert werden können.

Der Grund dafür ist historischer Natur. Denn solche Programmiersprachen haben ihren Ursprung in dem Modell, das John von-Neumann 1945 entwickelt hat.

Dabei stellt man sich eine Variable als Speicherzelle vor, in der der aktuelle Wert „irgendwie" abgelegt ist. Von-Neumann ging dabei noch einen Schritt weiter, indem er erkannte, dass die Daten sinnvollerweise binär verwaltet werden sollten. Jede Speicherzelle, die für eine solche Variable verantwortlich ist, ist mit Hilfe einer sog. Adresse, ebenfalls binär codiert) ansprechbar, so dass man den Wert einer Variablen über diese Adresse lesen und verändern kann.

Jedoch klafft eine Lücke zwischen dieser Sichtweise und der physikalischen Realität, so wie sie in Computern vorliegt.

Daten wie z.B. natürliche Zahlen[1] werden in fast allen Computern binär codiert und in Speicherzellen[2] verwaltet. Ein Computer wird also eine Reihe von Speicherzellen besitzen:

- Jede Speicherzelle hat eine Nummer, die sog. Adresse der Zelle.
- Die Adressen werden ebenfalls binär angegeben. Je nach Anzahl der zu adressierenden Zellen stehen für die Adresse unterschiedlich viele Bits zur Verfügung.
- Jede Speicherzelle kann einen sog. **`Bitstring`**, also eine Kette von Nullen oder Einsen speichern. Alle Speicherzellen haben dieselbe Breite, die Bitstrings haben also dieselbe Anzahl von Binärzeichen.

Alle Speicherzellen bilden dann den Speicher der von-Neumann-Maschine.

1 Die spezielle Codierung von komplexen Datentypen soll hier nicht weiter thematisiert werden.

2 Bei einer noch feineren Betrachtung besteht der Speicher aus Zellen, die nur ein Bit speichern können. Mehrere solcher Zellen werden zu einer adressierbaren einheit (hier *Speicherzelle* genannt) zusammengefasst.

Der Speicher

Stellen wir uns beispielsweise den gesamten Speicher eines Computers (hier sind nur die ersten 8 Zellen dargestellt; die Adressen sind hier mit 4 Bit-codiert) wie folgt vor:

Adresse	Inhalt
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	

In den Speicherzellen befinden sich Ketten gleicher Länge (Speicherbreite genannt) von Nullen und Einsen. Wenn die Ketten z.B. eine (feste) Länge von 8 Bit[3] haben, könnten die Zellen folgende Belegungen aufweisen[4]:

Adresse	Inhalt
0000	0101 0111
0001	1101 1010
0010	0011 1100
0011	1011 0101
0100	0111 1111
0101	1111 1010
0110	0001 1001
0111	0000 0001

Wie funktioniert jetzt ein von-Neumann-Rechner?

3 Später wird die Speicherbreite auf 16 erweitert.

4 Aus Gründen der Lesbarkeit wurden die 8-Bit-breiten Bitstrings in zwei Vierergruppen aufgeteilt.

Funktionsweise der von-Neumann-Maschine

Die zentrale Idee einer von-Neumann-Architektur besteht nun darin, dass ein Bitstring, der in einer Speicherzellen enthalten ist, als „Datum“, also als ein vom Programm zu verwaltender (Zahlen-)Wert, aber auch als „Befehl“ interpretierbar ist.

Ob es sich um einen Befehl oder ein Datum handelt, ist zunächst nicht zu erkennen. Es hängt ausschließlich davon ab, ob der Inhalt der Speicherzelle von der Maschine als Befehl interpretiert wird (der dann ausgeführt werden soll) oder als Datum behandelt wird.

Dabei kann es natürlich durchaus passieren, dass eine als Datum vorgesehene Zelle als Befehl interpretiert wird. Ist das dann (zufällig) ein zulässiger Befehl, bleibt der Fehler zunächst unbemerkt; doch im weiteren Verlauf des Programms werden sicherlich unerwünschte Effekte auftreten. Handelt es sich um einen unzulässigen Befehl, wird die Programmausführung abbrechen müssen.

Die beschriebene Funktionsweise kann mit Hilfe einer zunächst computerfreien Simulation, mit einem sog. Schachtelcomputer, verdeutlicht werden (siehe dazu **Q2-y.1-Rollenspiel.docx**).

Es handelt sich hierbei um die Simulation einer *1-Adress-Maschine*[5].

Eine 1-Adress-Maschine zeichnet sich dadurch aus, dass der Inhalt einer Speicherzelle als aus zwei Teilen bestehend interpretiert wird, falls die Maschine diese Zelle als Befehl auszuführen hat. Ein Teil des Bitstrings codiert die Art des Befehls, der restliche Teil stellt die Adresse dar, auf den sich der Befehl bezieht.

5 Denkbar sind natürlich auch andere Maschinen (3-Adress-Maschine, 2-Adress-Maschine, Kellermaschine, ...), die als Referatsthemen oder auch als Facharbeitsthemen geeignet sind.

Alle diese Maschinen haben Gemeinsamkeiten, die sie als eine von-Neumann-Architektur charakterisieren, in Abgrenzung z.B. zu Computern, die auf der Harvard-Architektur[6] basieren. Eine von-Neumann-Maschine besteht aus den folgenden vier Teilen:

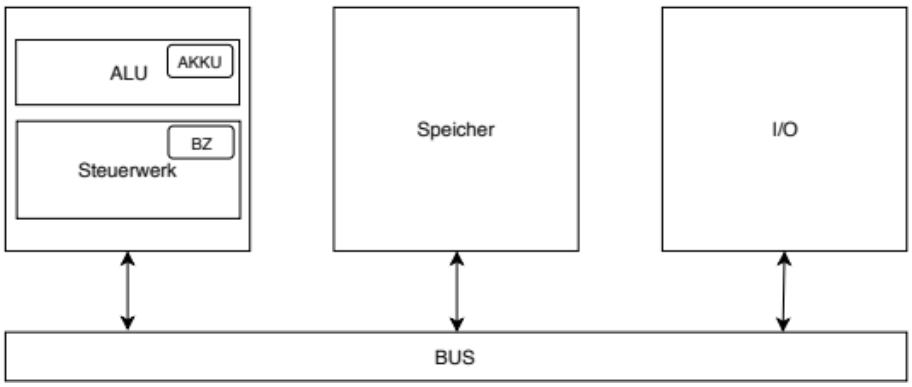
- Das Rechenwerk (die sog. ALU, engl. Arithmetic Logic Unit) führt Rechenoperationen und logische Verknüpfungen durch. Dazu besitzt die ALU eine[7] spezielle Speicherzelle, AKKU genannt, in der z.B. Ergebnisse von Rechenoperationen verwaltet werden.

- Das Steuerwerk interpretiert die Befehle. Es beinhaltet ein Register (der Befehlszähler, mit BZ abgekürzt, engl. *program-counter*), in dem die Adresse des als nächstes als Befehl zu interpretierenden Speichers steht.

- Das Speicherwerk (RAM, engl. *random access memory*), in dem Daten binär gespeichert werden. Diese Daten können als Daten zum „Rechnen“ oder auch als Befehl interpretiert werden.

- Das Ein-Ausgabewerk (I/O), das für den Datenaustausch zwischen dem Anwender (Tastatur, Bildschirm, …) und dem Speicherwerk verantwortlich ist.

Alle diese Teile sind über eine Art Leitung, (Daten-)Bus genannt, miteinander verbunden:



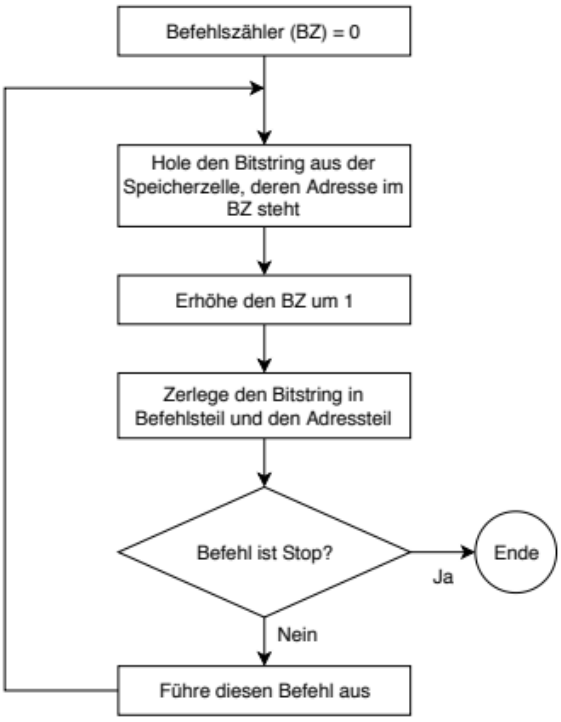
6 Auch das könnte ein Referatsthema oder Thema für eine Facharbeit sein.

7 Ggf. verfügt die Steuereinheit auch über mehrere solcher, Register genannten Speicher.

Die Programmabarbeitung: lineare Programme

Ein Programm wird als eine Folge von (binär codierten) Befehlen in den Speicherzellen abgelegt. Die Daten, auf denen das Programm arbeitet, liegen ebenfalls in den Speicherzellen.

Wird eine von-Neumann-Maschine dann gestartet, wird das Programm abgearbeitet. Die Arbeitsweise kann zunächst mit folgendem Diagramm verdeutlicht werden:



Nachdem der Inhalt der aktuellen Speicherzelle gelesen wurde, wird dieser Bitstring zerlegt in den Befehls- und den Adressteil.

Die zentrale Aktion erfolgt dann in dem letzten Schritt:

```
Führe diesen Befehl aus.
```

Der Befehl wird mit Hilfe der Liste der Maschinenbefehle spezifiziert und eine entsprechende Aktion ausgeführt.

Die Menge der verschiedenen Befehle sowie deren Codierung ist sehr unterschiedlich (Stichworte: CISC, RISC, ...). Referate dazu bieten sich an.

Die Schülerinnen und Schüler können jetzt mit Hilfe des Schachtelcomputers die Arbeitsweise kennenlernen (siehe entsprechende Arbeitsblätter).

Die Programmabarbeitung: nicht-lineare Programme

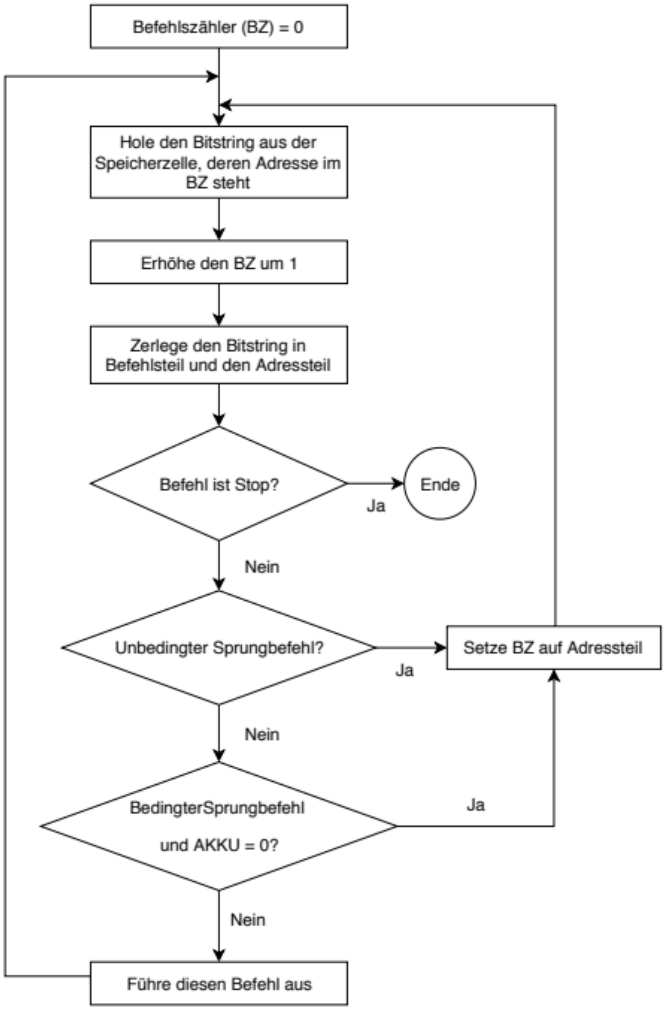
Jedoch wird sehr schnell deutlich, dass die lineare Abarbeitung der Befehle es nicht erlaubt, die aus höheren Programmiersprachen bekannten Fallunterscheidungen und Schleifen zu realisieren.

Die Lösung besteht darin, logische Vergleiche und darauf reagierende sog. Sprungbefehle dem Befehlsvorrat hinzuzufügen. Zu unterscheiden ist dabei zwischen bedingten und unbedingten Sprungbefehlen:

- Ein unbedingter Sprungbefehl veranlasst die Maschine, den Programmzähler auf den Wert zu setzen, der im Adressteil angegeben ist.
- Ein bedingter Sprungbefehl verändert den Programmzähler nur dann, wenn eine Bedingung erfüllt ist. Es gibt dazu mehrere Möglichkeiten (von denen später in der Simulation nur eine Variante realisiert wird).
  Der Sprung wird z.B. nur dann ausgeführt, der PC wird also nur dann neu gesetzt, wenn der Inhalt des Registers AKKU den Wert 0 hat.

Weitere Arten bedingter Sprünge (z.B. Sprung nur dann, wenn AKKU > 0 ist) sind denkbar[8].

Falls die von-Neumann-Maschine unbedingte Sprünge sowie bedingte Sprünge von der Art **`AKKU = 0`** zulässt, wird die Arbeitsweise mit folgendem Diagramm verdeutlicht:



8 Das an anderer Stelle vorgestellte Simulationsprogramm *Johnny* benutzt eine andere Strategie.

Die Maschinenbefehle

Die Speicherzellen beinhalten (in dieser Beispielarchitektur) 16-Bit-breite 0-1-Zeichenketten. Wird ein solcher Bit-String vom Steuerwerk gelesen, wird also als Befehl interpretiert, muss vereinbart werden, welche Aktion mit diesem Befehl verbunden ist.

Hier wollen wir vereinbaren, dass die ersten 4 Bit der Zeichenkette die Funktionalität des Befehls codieren, die restlichen 12 Bit eine Adresse darstellen, auf die sich der Befehl bezieht.

Die hier beispielhaft dargestellte fiktive von-Neumann-Maschine benutzt folgende Befehle:

Codierung des Befehls	Bedeutung	Kurzform
0000	Kein gültiger Befehl.	**data**
0001	Der Adressteil wird als positive Zahl interpretiert und in den AKKU transportiert. Der BZ wird um 1 erhöht.	**lint**
0010	Lese den Inhalt der Speicherzelle mit der angegebenen Adresse und speichere den Wert im AKKU. Der BZ wird um 1 erhöht.	**load**
0011	Speichere den Inhalt des AKKUs in der Speicherzelle mit der angegeben Adresse. Der BZ wird um 1 erhöht.	**store**
0100	Addiere den Inhalt der Speicherzelle mit der angegebenen Adresse zum Inhalt des AKKUs, speichere die Summe im AKKU. Der BZ wird um 1 erhöht.	**add**
0101	Subtrahiere den Inhalt der Speicherzelle mit der angegebenen Adresse vom Inhalt des AKKUs, speichere die Differenz im AKKU. Der BZ wird um 1 erhöht.	**sub**
0110	Multipliziere den Inhalt das AKKUs mit dem Inhalt der Speicherzelle mit der angegebenen Adresse. Der BZ wird um 1 erhöht.	**mul**
0111	Dividiere den Inhalt das AKKUs ganzzahlig durch dem Inhalt der Speicherzelle mit der angegebenen Adresse. Der BZ wird um 1 erhöht.	**div**
1000	Der Inhalt des AKKUs wird zum Rest der ganzzahligen Division von Inhalt von AKKU und Inhalt der Speicherzelle mit der angegebenen Adresse. Der BZ wird um 1 erhöht.	**mod**
1001	Der Befehlszähler wird auf den Wert der Speicherzelle mit der angegebenen Adresse gesetzt.	**jmp**
1010	Der Befehlszähler wird auf den Wert der Speicherzelle mit der angegebenen Adresse gesetzt., falls der Inhalt des AKUs den Wert 0000 hat. Ansonsten wird der befehlszähler um 1 erhöht.	**jeq**
1011	Der Inhalt eines sich öffnenden Eingabefensters wird, als nicht-negative Zahl interpretiert, als Bitstring in den AKKU geschrieben. Der BZ wird um 1 erhöht. (Die Adresse ist hier irrelevant)	**in**
1100	Gib den Inhalt des AKKUs auf einem Ausgabemedium aus. Der BZ wird um 1 erhöht. (Die Adresse ist hier irrelevant)	**out**
1101	Beende das Programm. (Die Adresse ist hier irrelevant)	**stop**
1110	Keine Operation. Der BZ wird um 1 erhöht.	**nop**

Assembler

Die Zuordnung der (in diesem Beispiel) mit 4 Bit codierten Befehle zu deren Funktionalität ist eindeutig. Um die Lesbarkeit (für Menschen) eines von-Neumann-Maschinenprogramms zu vergrößern, können statt der 4-Bit-breiten Codierung die (in der Tabelle angegeben) Kurzformen benutzt werden. Ein in dieser Art geschriebenes Programm kann dann recht einfach von einem geeigneten Algorithmus in das entsprechende Maschinenprogramm übersetzt werden.

Ein solches Programm nennt man *Assembler-Programm*, den zugehörigen Übersetzer nennt man *Assembler*.

Eine typische Assemblerzeile hat dann die Form

```
Befehl Adresse
```

Dabei ist **`Befehl`** ein Element aus der Liste der gültigen Kurzform-Befehle (s.o.). Adresse ist eine Zahl aus dem Bereich [0, 1, …, 4095].

Ein gültiger Befehl ist also z.B.

```
jmp 42
```

und würde in dem Speicher der von-Neumann-Maschine codiert werden als

```
1000 0000 0010 1010
```

Symbolischer Assembler

In einem weiteren Schritt kann man jede Speicherzelle symbolisch benennen. So könnte man der Zelle 42 den symbolischen Namen **`a`** geben. Diese Idee kommt dem Variablenkonzept in höheren Programmiersprachen schon sehr nahe.

Um diese Idee in den Assembler-Programmen zu nutzen, kann eine Zeile eine sog. Marke bekommen.

Für das oben beschriebene Beispiel muss es dazu eine Zeile der Form

```
a: Befehl Adresse
```

geben, so dass dann die Assembler-Zeile

```
jmp a
```

gültig ist.

Gültige Marken sind in diesem Kontext alle Kleinbuchstaben.

Kommentare im symbolischen Assembler

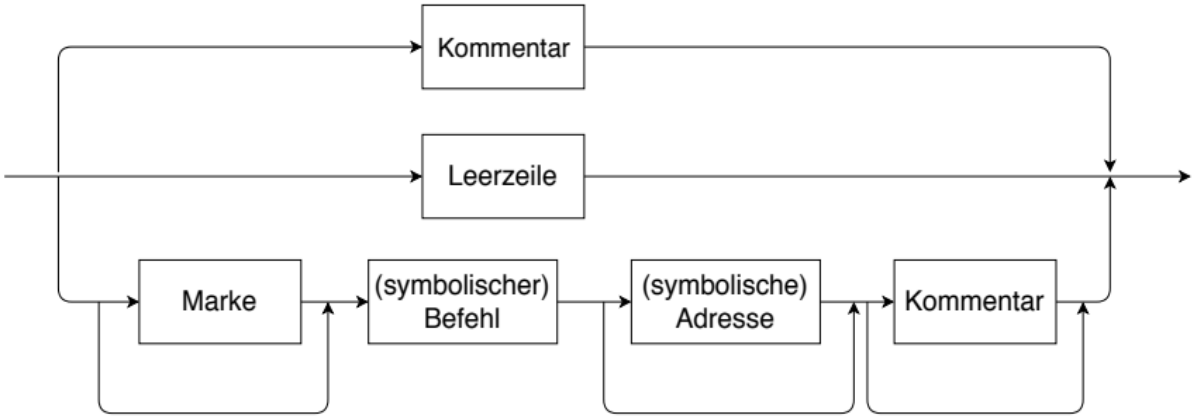
Zur Lesbarkeit eines Assembler-Programms tragen Kommentare bei, so wie man es in höheren Programmiersprachen kennt:

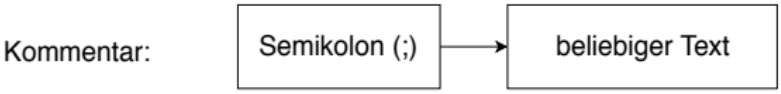
- Leerzeilen im Programmtext werden ignoriert.
- Zeilen, die mit einem Semikolon (`;`) beginnen, werden ignoriert.
- In allen anderen Zeilen wird alle Zeichen nach einem Semikolon (`;`) ignoriert.

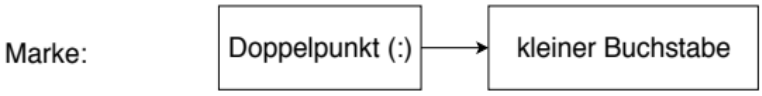
Eine gültige Assembler-Zeile ist dann z.B.

```
x: load n ; Schleifenanfang
```

Insgesamt kann dann die Syntax einer gültigen Assemblerzeile in Form eines Syntaxdiagramms dargestellt werden:







Ein Beispiel

Ein in dieser Art erstelltes Assembler-Programm, das zwei Zahlen a und b multipliziert und das Ergebnis ausgibt. Dabei wird die **`mul`**-Anweisung nicht benutzt; stattdessen wird die Summe `a + a +...+ a` errechnet.

```
;Multipliziere die Werte a und b
;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: out r
stop
a: Data
b: Data
e: Data ; eins
r: Data ; resultat
y: Data ; Kopie von b
```

Höhere Programmiersprachen / Maschinensprachen

Der Zusammenhang zwischen höheren Programmiersprachen wie Java und einer Funktionalität im Sinne der von-Neumann-Architektur ist dann zusammenfassend:

höhere Programmiersprachen	Maschinensprachen, basierend auf der Von-Neumann-Architektur
Variablen	Speicherzellen
Kontrollstrukturen	bedingte und unbedingte Sprunganweisungen
Zuweisungen	Lese- und Schreibanweisungen
arithmetische Ausdrücke	Rechenoperationen

Jedoch ist dieser Zusammenhang beschränkt auf imperative Programmiersprachen. Neben diesen gibt es alternative Konzepte, die sich z.B. in

- funktionalen (wie HASKELL) oder auch in
- deklarativen (wie PROLOG)

Sprachparadigmen finden lassen. Hier müssen ggf. andere Rechnerarchitekturen genutzt werden, oder Emulationen anderer Architekturen auf der von-Neumann-Maschine existieren.

Simulation

Die Funktionsweise eines von-Neumann-Rechners wird verdeutlicht anhand eines sehr eingeschränkten Modells. Das im ersten Abschnitt beschriebene Modell (die Adresse ist 4-Bit, der Speicher ist 8-Bit breit) bietet nur Platz für 16 Speicherzellen, ist zwar für erste Gehversuche sinnvoll, jedoch schon für kleinere Programme ungeeignet. Die Speicherbreite sollte ggf. 16-Bit betragen, wobei die Codierung der Befehle mit 4 Bit ausreichend erscheint. Damit hat man 12 Bit als Adresscodierung zur Verfügung, so dass man $2^{12} = 4096$ Zellen adressieren kann. Gültige Adressen sind also Werte aus dem Bereich [0, 1, …, 4095].

- Ein Java-Programm, **`VNR.jar`**, mit dem man eine solche von-Neumann-Maschine simulieren kann, sowie die zugehörige Dokumentation sind beigefügt.

Kleine Programme wie auf den Arbeitsblättern beschrieben, werden untersucht bzw. erstellt. Dabei können Programme im Binärcode oder Assemblercode angegeben werden.

Eine Computersimulation eines von-Neumann-Rechners bietet auch das Programm ***Johnny***, das über den Link[9]

- johnny-simulator

kostenfrei (nur als als Windows-Anwendung erhältlich) zu beziehen ist. Empfehlenswerte Seiten zu diesem Simulator sind zu finden unter:

- Johnny-Wiki
- Aufbau_Fkt_Rechner_Johnny_V3_0.pdf

Eine Alternative bietet der Modellrechner MOPS, zu beziehen über

- mops

Eine weitere, HTML-basierte Simulation finden Sie unter

- Bonsai-Computer

 Die angegebenen Links wurden am 6.6.2019 verifiziert.