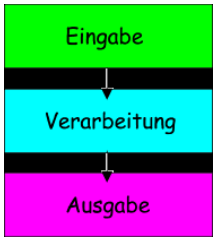
„Von der Programmiersprache zur Maschinensprache“

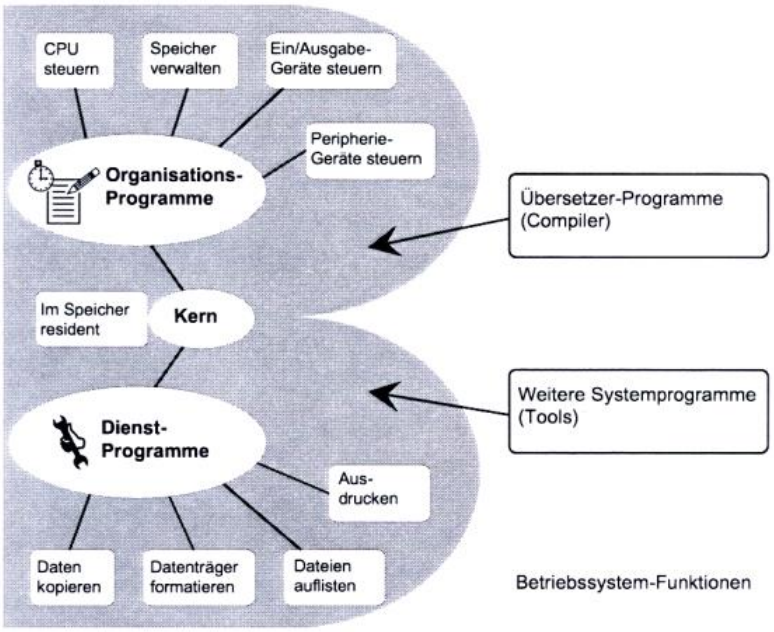
1. Das Betriebssystem



Wie wir alle wissen, ist das „EVA“-Prinzip ein grundlegendes Prinzip bei der Arbeit mit einem Computer – der Anwender hat ein Problem, das er gerne von der Maschine gelöst haben möchte und gibt deshalb Daten über die Tastatur oder andere Eingabegeräte ein. Der PC verarbeitet schließlich diese Daten in einer sehr komplizierten Art und Weise – die wir selbstverständlich nicht bis ins kleinste Detail analysieren werden – bevor er eine (für den Nutzer zufrieden stellende) Ausgabe liefert.

Eine bei der Verarbeitung essentielle Komponente stellt dabei das **Betriebssystem** dar – es ist sozusagen der übergeordnete „Manager“ von dem ganzen Werk. Es hat vor allem 5 wichtige Aufgaben:

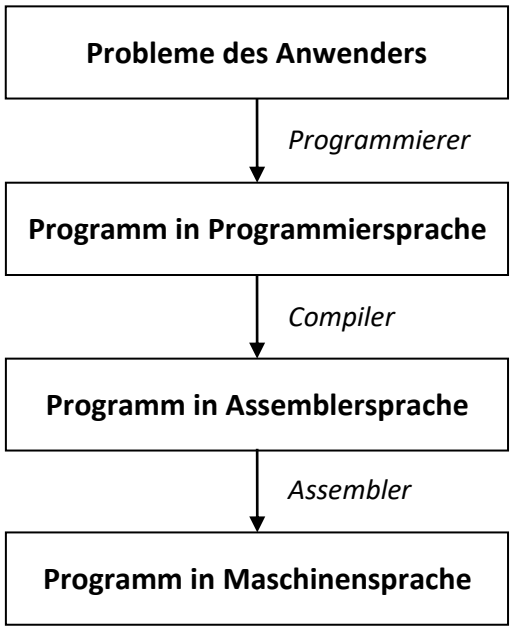
- Steuerung des Rechners
- Datentransfer zum Prozessor
- Speicherverwaltung
- Kontrolle des Datenstromes zwischen den Peripheriegeräten und – nicht zu verachten –
- Ermöglichung des Bootens.



Wie die Graphik rechts schön veranschaulicht, lassen sich die untergeordneten Bestandteile des Betriebssystems vor allem in **Organisationsprogramme** (zur Ablaufsteuerung sämtlicher Vorgänge), **Dienstprogramme** und **Übersetzungsprogramme** unterteilen. Letztere können nochmals grob aufgesplittet werden in „**Compiler**“ und „**Assembler**“, welche im weiteren Verlauf noch intensiver betrachtet werden.

2. Der Weg vom Ausgangsproblem bis zur Ausführung

Man nehme nun einen Anwender mit einem Problem, das er gerne gelöst haben möchte, weshalb er einen erfahrenen Programmierer mit der Umsetzung seiner Idee (in Java) betraut. Hier stellt sich wiederum die Frage: **wie werden die geschriebenen Programme in eine für die Maschine zu interpretierende Form gebracht?** Eine erste grobe Übersicht liefert die rechts abgebildete Grafik.



Die Programmiersprache (Java) an sich kann der PC natürlich so ohne Weiteres nicht direkt verstehen. Sie besteht aus einer endlichen Anzahl von zulässigen Zeichen und Anweisungen, die dem Benutzer das relativ leichte Umsetzen eines Algorithmus ermöglichen soll. Bevor das Betriebssystem aber nun die Befehle der Programmiersprache interpretieren kann, müssen diese in

eine Folge von **elementaren Befehlen** umgewandelt werden, **die für** die **Maschine ausführbar** sind. Dieser Weg, der über eigenständige Übersetzungsprogramme verläuft, soll nun ein wenig beleuchtet werden. Zunächst sorgt ein so genannter „**Compiler**“ u. a. für die Behebung von syntaktischen Fehlern im Quelltext – darunter versteht man einzelne Wörter/ Ausdrücke/ Programmierzeilen, die mit der vorgeschriebenen Grammatik der vorliegenden Sprache nicht zu vereinbaren sind (vom Programmierer gemachte logische Fehler werden natürlich nicht erkannt! Diese muss er selbst durch Testdurchläufe o. Ä. ausschließen). Nun liegt das Programm intern in der „**Assemblersprache**“ vor. Im zweiten Schritt sorgt ein so genannter „**Assembler**“ für die Übersetzung in die **Maschinensprache**, welche aus Codewörtern besteht, die von Speicherzellen aufgenommen und vom Leitwerk interpretiert werden können. Compiler können dementsprechend maschinenunabhängig gebaut werden, Assembler müssen sich an der genauen Verarbeitung des jeweiligen Prozessors orientieren.

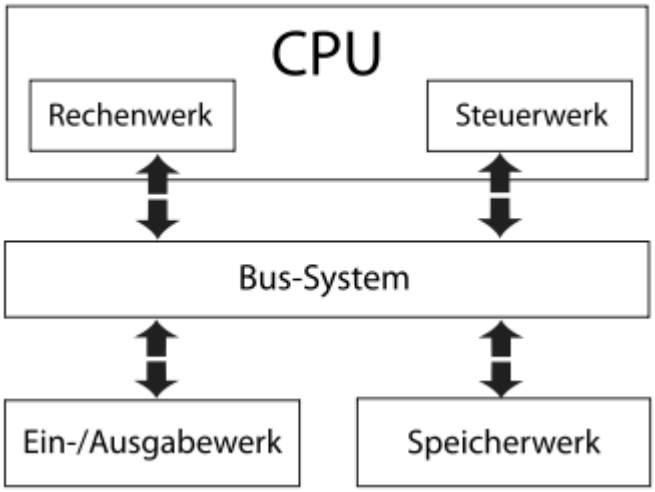
Ziel ⇨ Im Verlauf dieses Skripts werden wir uns nun mit der **Erarbeitung einer Assemblersprache** befassen.

Dieses Ziel kommt dem Verstehen einer Maschinensprache schon recht nahe, da die Assemblersprache eine 1:1-Übersetzung darstellt – allerdings in deutlich leichter lesbarer Form! Die Maschinensprache besteht letzten Endes nur noch aus irgendwelchen Bitmustern, die jeweils spezielle Befehle repräsentieren.

Bevor wir das Ziel direkt in Angriff nehmen, sollten wir uns zum besseren Verständnis also zunächst verdeutlichen, wie ein Rechner der heutigen Generation überhaupt funktioniert.

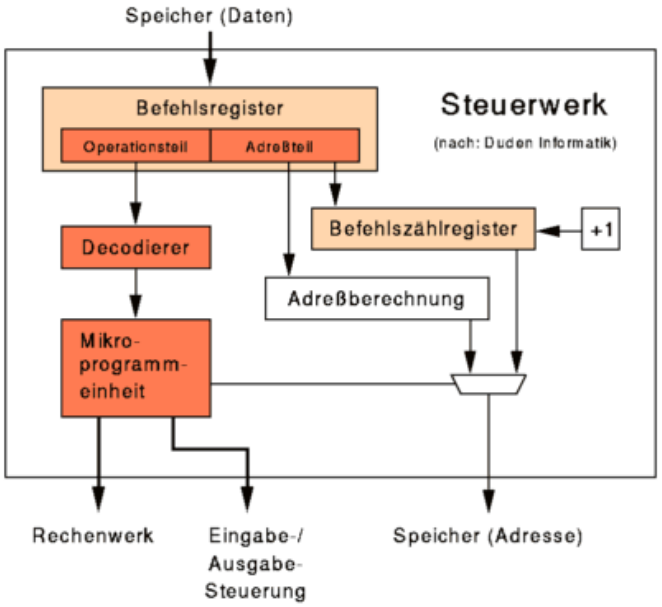
3. Der von-Neumann-Rechner

Das Interessante dabei ist, dass nahezu alle Rechner, die seit ihrer Erfindung gebaut wurden, nach dem **Verarbeitungsprinzip von von-Neumann** arbeiten. Ein so genannter „von-Neumann-Rechner“ kann schematisch wie rechts zu erkennen abgebildet werden.



Das Rechenwerk führt dabei Rechenoperationen und logische Verknüpfungen aus, das Steuerwerk interpretiert die Anweisungen des Programms und steuert die Befehlsabfolge, das Speicherwerk speichert sowohl Programme als auch Daten, welche für das Rechenwerk zugänglich sind, und das Ein-/ Ausgabewerk steuert die Ein- und Ausgabe von Daten zum Anwender (Tastatur, Bildschirm) oder zu anderen Systemen.

Hervorzuheben ist John v. Neumann außerdem als erster Mensch, der die wesentlichen Vorzüge der „internen Programmspeicherung“ erkannte. Zuvor wurden die Daten, mit denen ein Programm arbeitete, und das Programm an sich in zwei verschiedenen Speichermedien eingespeist. Mit der „internen Programmspeicherung“ war es nun möglich, ein Programm selbst als Daten aufzufassen, welche manipuliert werden konnten. Erst dadurch wurde es möglich, ein Programm variabel zu gestalten und Schleifen, Verzweigungen und Sprünge einzubauen. Ein variabler Befehlszähler hält z.B. fest, welcher Befehl als nächstes ausgeführt werden soll, wodurch ein Sprung über mehrere Befehle hinweg kein Problem mehr darstellt. Eine Verzweigung hingegen kann man dadurch realisieren, dass man in Abhängigkeit eines speziellen Datenwertes den Befehlszähler auf zwei unterschiedliche Folgebefehle setzt.



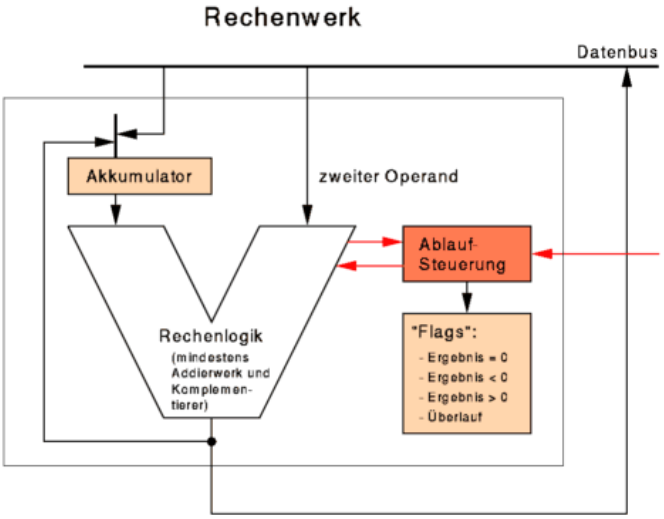
Allgemein kann man das Verarbeitungsprinzip eines von-Neumann-Rechners in vier Teilschritte zerlegen, die man als „**von-Neumann-Zyklus**“ bezeichnet:

1. Befehl holen

Im Befehlszählregister steht die Adresse des Befehls, der bearbeitet werden soll. Das Steuerwerk bewirkt nun, dass dieser Befehl zum Befehlsregister gebracht wird.

2. Befehl dekodieren

Im Befehlsdekodierer wird der Befehl analysiert. Durch Steuersignale wird dafür gesorgt, dass Operanden mit dem Rechenwerk in Verbindung gebracht und die zur Bearbeitung notwendigen Steuersignale erzeugt werden.



3. Befehl ausführen

Die Steuersignale werden nun von anderen Einheiten, die für den entsprechenden Befehl zuständig sind, verarbeitet (Steuerwerk, etc.).

4. Befehlszähler ändern

Am Ende der Ausführung wird der Befehlszähler so verändert, dass er die Adresse des nächsten Befehls enthält. Normalerweise wird der alte Wert nur um die Länge des aktuellen Befehls erhöht.

4. Die Zwischensprache ReJava

Nun aber zurück zu unserem eigentlichen Ziel. Wir wollen die Schritte, die zwischen der Programmiersprache Java und dem gerade vorgestellten von-Neumann-Zyklus liegen, nachvollziehen und verstehen – werden uns aber auf die Assemblersprache konzentrieren und die Maschinensprache nur kurz der Vollständigkeit halber erwähnen.

Das heißt insbesondere, dass wir eine Möglichkeit finden müssen, Java-Programme schrittweise zu „elementarisieren“, was auch u. a. Aufgabe des Compilers in Java ist. Die wesentlichen Zwischenschritte im Übersetzungsvorgang werden wir nun schematisch nachempfinden.

Deshalb ist unser vorläufig erster Schritt, Java-Programme in eine elementare Programmiersprache namens „**ReJava**“ zu übersetzen. Mit Hilfe dieser wird es uns möglich sein, Programmieranweisungen bereits teilweise zu zerlegen – es sollen nur noch Anweisungen mit einer einzigen Operation möglich sein (auch „**Dreiadressbefehle**“ genannt, weil bei einer Operation maximal drei Adressen benötigt werden). Voraussetzung ist natürlich, dass jede noch so komplexe Anweisung in eine Folge von Dreiadressbefehlen zerlegt werden kann. Dem ist aber so, was zunächst anhand eines Beispiels verdeutlicht werden soll. Man schaue sich die folgende Zuweisung an:

$$x = (a + b) * 100 – zahl / 10.$$

Wie wir sehen, sind dafür schon sechs Adressen von Nöten (Konstante werden auch in einer Speicherzelle abgelegt) sowie drei Operationen. Die Übersetzung in einen Dreiadressbefehl muss also über Hilfsvariablen erfolgen:

```
h1 = a + b
h1 = h1 * 100
h2 = zahl / 10
x = h1 - h2.
```

Auch von Java bekannte Abfragen oder Schleifen müssen natürlich umgesetzt werden können, was wiederum kein Problem darstellt, da ein Befehlszähler (vgl. „von-Neumann-Zyklus oben) einen Sprung (bedingt oder unbedingt) ohne Weiteres ermöglicht.

Sprachumfang der Minisprache ReJava

a) Datentypen und Operationen:

Da die interne Speicherung aller Datentypen – genauso wie die Speicherung eines Integer-Datentypen – auf die Codierung in Nullen und Einsen hinausläuft, ist es kein Verlust, sich nur auf den Datentypen Integer zu beschränken.

Die einzigen möglichen Operatoren sind:

+ (Addition), **-** (Subtraktion), ***** (Multiplikation), **/** (ganzzahlige Division)

sowie die Vergleichsoperatoren

<, <=, >, >=, ==, !=

b) Logische Ausdrücke:

Ein logischer Ausdruck kann wegen des Dreiadressformats nur in der folgenden Form durchgeführt werden: **Ausdruck_1 Vergleichsoperator Ausdruck_2**

Wir nennen einen solchen Ausdruck **Bedingung**.

c) Zuweisungen:

Bei Zuweisungen sind die folgenden Einschränkungen zu beachten:

1. Es sind nur die oben genannten Operatoren erlaubt.
2. Auf der rechten Seite einer Zuweisung darf höchstens ein Operator stehen.

(Eine derartige Zuweisung referiert somit offensichtlich auf höchstens drei Adressen.)

d) Anweisungen:

Als Anweisungen sind erlaubt:

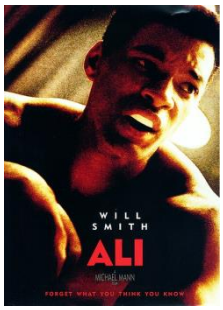
1. Zuweisung: **=**
2. Ein- und Ausgabe: **read, write**
3. unbedingter Sprung zur Marke n: **goto** n
4. bedingter Sprung zur Marke n: **if** Bedingung **then goto** n.

Übungen

Ein erstes Beispiel der Umwandlung einer Java-Anweisung in ReJava-Anweisungen kannst Du nochmals oben nachverfolgen. Probiere dann selbst folgende Java-Anweisungen umzuschreiben. Der Programmkopf und die Variablendeklarationen bleiben dabei erhalten!

1. x = c * (a - b) - (a + d) / (5 * b)
2.
```
public static void Gauss () {
    int n = 0, s = 0;
    n = scanner.nextInt(); s = (n + 1) * n / 2; System.out.print(s)
}
```

Assemblersprache „WinAli“



Nun, wo wir die Zwischensprache ReJava in unser Repertoire aufgenommen haben, können wir uns um eine weitere Zerlegung der erhaltenen Dreiadressbefehle in Einadressbefehle bemühen (da eine Maschinensprache nur solche verwerten kann) – und werden dabei die Assemblersprache **WinAli** kennen lernen. Diese Zerlegung in Befehle mit nur noch einer einzigen Adresse wird durch einen so genannten **Akkumulator** (**ACCU**) möglich, auf den das Leitwerk unmittelbar zugreifen kann; das bedeutet insbesondere, dass der ACCU keine Adresse/ Speicherzelle darstellt. Er nimmt sowohl Operanden als auch Ergebnisse nach Ausführung einer Operation auf.

Allgemein muss man sich zunächst einmal klarmachen, dass jede noch so komplexe Anweisung überhaupt in Einadressbefehle zerlegt werden kann. Da die (vorherige) unproblematische Überführung in Dreiadressbefehle schon veranschaulicht wurde, müssen diese nun nur noch in Einadressbefehle umgewandelt werden, was genauso wenige Schwierigkeiten bereitet. Im Allgemeinen gibt es folgende mögliche elementare Operationen:

1. Der Akku wird geladen (gleich dem Inhalt einer Speicherzelle gesetzt).
2. Der Inhalt des Akkus wird mit dem einer anderen Speicherzelle verrechnet.
3. Der Inhalt des Akkus wird in einer Zelle abgespeichert.

Dies sei zunächst an einem einfachen Beispiel verdeutlicht. Gegeben sei die Dreiadresszuweisung

$$x = a + b,$$

die im Ein-Adress-Format dargestellt werden kann als

```
LADE        ACCU, a
ADDIERE     ACCU, b
SPEICHERE   ACCU, x.
```

Dabei sind die durchzuführenden Aktionen in „Alltagssprache“ formuliert und entsprechen natürlich nicht dem genauen Befehlssatz von WinAli! Der im Moment wichtige Teil wird nun tabellarisch aufgeführt:

Befehlsaufbau	Wirkung	Erläuterung
Transportbefehle:		
LDA R, ADR	R = ADR	Laden
STA R, ADR	ADR = R	Speichern
Ein-/ Ausgabebefehle:		
INI ADR	read(ADR)	(Ein-)Lesen
OUTI ADR	write(ADR)	Schreiben/ Ausgeben
Rechenbefehle:		
ADD R, ADR	R = R + ADR	Addieren
SUB R, ADR	R = R – ADR	Subtrahieren
MUL R, ADR	R = R * ADR	Multiplizieren
DIV R, ADR	R = R div ADR	Dividieren
Pseudobefehle:		
name DS F	Legt einen Speicherplatz an, der im Programm mit „name“ aufgerufen werden kann.	Deklaration einer ganzzahligen Variable
konst DC wert	Legt einen Speicherplatz an, der im Programm mit „konst“ aufgerufen werden kann und weist gleichzeitig diesem den angegebenen „wert“ zu.	Deklaration einer ganzzahligen Konstanten
START 0	Programmkopf	Start
END	Ende des Programms (inkl. Variablen)	Ende
EOJ	Ende der Anweisungen	End of Job

Erläuterungen dazu:

- R ist ein Register im Bereich 0 bis 15, wobei 0 für den ACCU steht.
- ADR ist eine Adresse im Speicherbereich, die auf folgende Arten angegeben werden kann:
    1. Als **konkrete Zahl** (absolute Adressierung), die für einen festen Speicherplatz steht. (LDA 0, 10 bedeutet z. B., dass der Inhalt der Speicherzelle 10 in den ACCU geladen wird.)
    2. Als **Name** (symbolische Adressierung), der als Variable oder Konstante deklariert werden muss und später vom Assembler eine feste Adresse zugeordnet bekommt (z.B. LDA 0, xy).
    3. Als **Zahlenwert in Hochkomma** (unmittelbare Adressierung), der im Programm wie eine Konstante wirkt (Bsp.: LDA 0, ‘121‘).

Somit würde die oben betrachtete Anweisung in WinAli wie folgt aussehen (0 steht für den ACCU):

```
LDA   0, a
ADD   0, b
STA   0, x.
```

Ergänzend sollte noch erwähnt werden, dass man in WinAli **jede Codezeile** in **vier Spalten** unterteilen kann:

„label command params comment“.

Zur Erläuterung: „label“ ist eine Marke oder ein Sprungziel (oft bleibt diese Spalte leer!); „command“ bezeichnet den auszuführenden Befehl; „params“ sind die dem Befehl zugehörigen Parameter (einer oder zwei); „comment“ ist ein Kommentar, der durch „;“ oder „*“ eingeleitet werden muss.

Nun sollten wir uns die schrittweise Übersetzung von Java über ReJava bis hin zu WinAli nochmals an einem ausführlicheren Beispiel klarmachen (, das sogar schon einen weitere Programmierstruktur beinhaltet):

Programm in Java	Programm in ReJava	Programm in WinAli

public static void Beispiel() {
  int a, b, x;
  a = scanner.nextInt();
  b = scanner.nextInt();
 	
  if (a < 0) {
  a = -a;
  }
  x = a + b;
 	
  System.out.print(x);
}
	public static void Beispiel() {
	  int a, b, x;
	  read(a);
	  read(b);
	
	  if (a > 0) then goto n
	  a = -a;
	
	n x = a + b;
	
	  write(x);
	}
		start 0 * eingerückt (s.o.)!
		
		  ini a
		  ini b
		
		  lda 0, a
		  cmp 0, ’0’
		  bh n
		  mul 0, ‘-1’
		n add 0, b
		  sta 0, x
		
		  outi x
		  eoj
		a ds f
		b ds f
		x ds f
		  end

Der (angenäherten) Vollständigkeit halber seien hier alle möglichen Sprungbefehle in WinAli aufgeführt, wobei vorher stets ein Vergleich der Form “**cmp R,A**” durchgeführt werden muss:

b A	goto A;	bnl A	if (R >= O) then goto A;
be A	if (R = O) then goto A;	bnh A	if (R <= O) then goto A;
bne A	if (R <> O) then goto A;	bl A	if (R < O) then goto A;
bh A	if (R > O) then goto A;		