3. Aufbau und Struktur eines Prozessors

Zur Vorlesung Rechenanlagen

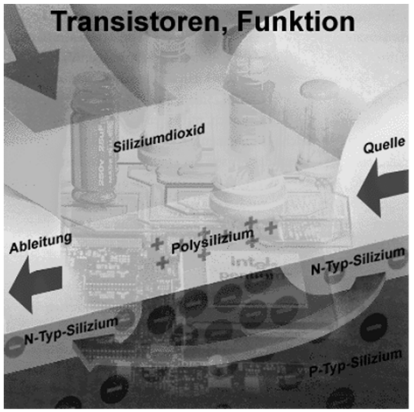
SS 2019

3.1 Maschinen, Instruktionen und Programme

Zur Vorlesung Rechenanlagen

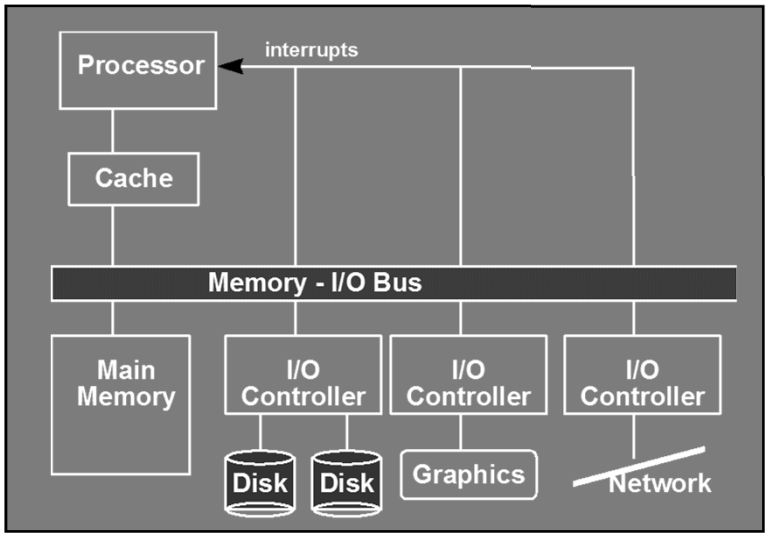
SS 2019

Was ist ein Rechner ?

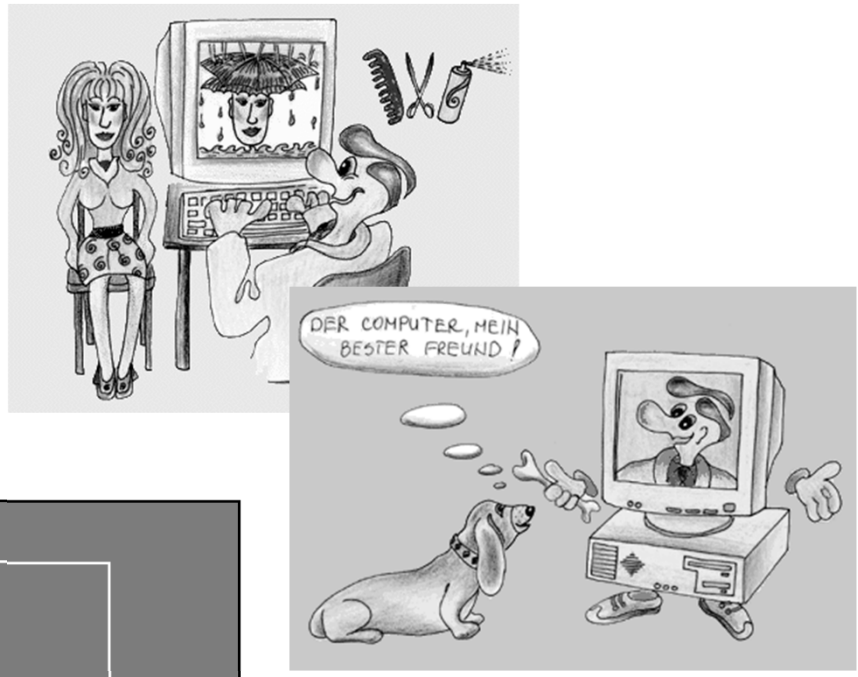


Transistor-Ebene

Layout Ebene

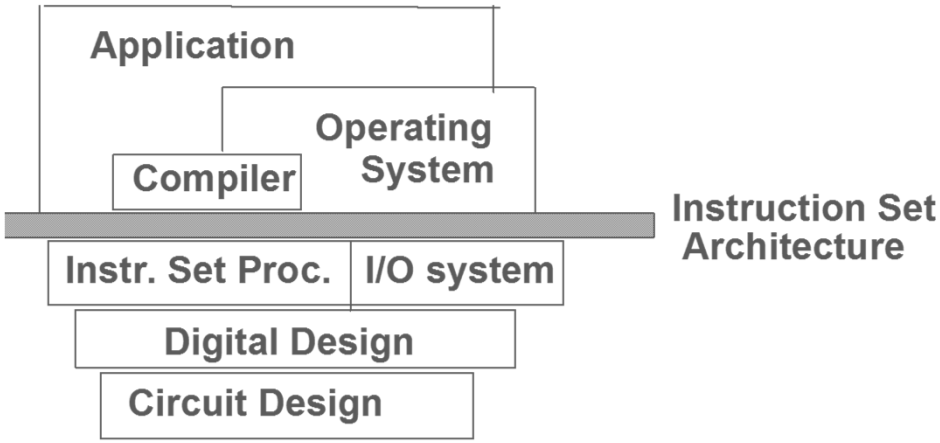


Register-Transfer-Ebene

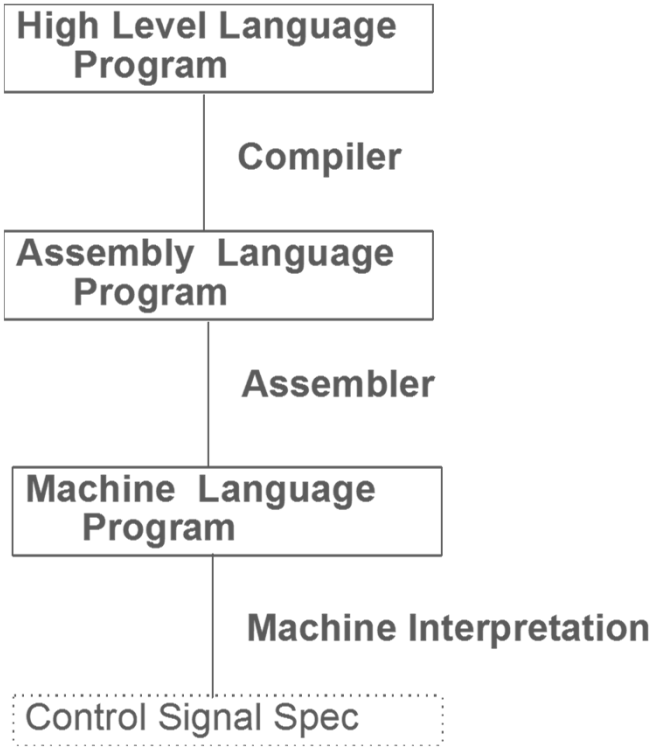


Anwendungsebene

Rechner = Hierarchie von virtuellen Maschinen

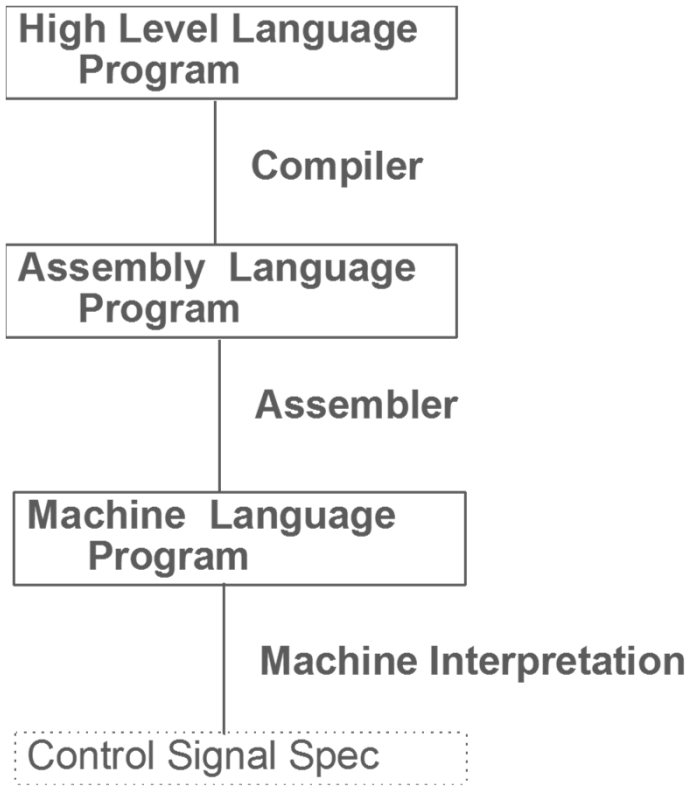


Realität



Logische Sicht

Hierarchie virtueller Rechner: Illustration




```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

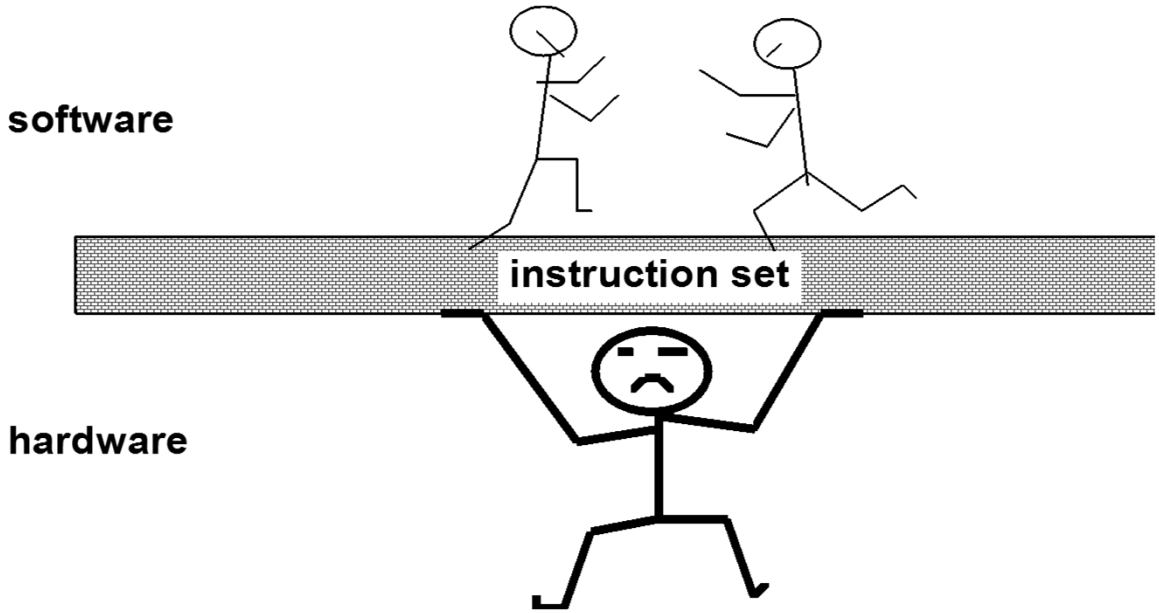
```
LOAD R5,  R2, #0
LOAD R6,  R2, #4
STORE R6, R2, #0
STORE R5,  R2, #4
```

```
1001 0000 0000 0000 0010 0000 1111 0000
1001 0000 0100 0000 0010 0001 0000 0000
0001 0000 0000 0000 0010 0001 0000 0000
0001 0000 0100 0000 0010 0000 1111 0000
```

... Interpretation auf der Hardware

Im Mittelpunkt steht die Maschinensprache

..., die unterste Ebene, auf der der Benutzer mit dem Rechner reden kann !



Virtueller Rechner

- Jede Ebene (mit Ausnahme der untersten) ist ein "virtueller Rechner" M_i, der durch eine Sprache L_i definiert ist.

- Programme der Sprache L_{i+1} werden
  - in Programme der Sprache L_j $(j \leq i)$ übersetzt oder
  - durch ein Programm der Sprache L_j $(j \leq i)$ interpretiert.

- Die virtuelle Maschine M_i gaukelt dem Benutzer vor, seine in L_i geschriebenen Programme würden direkt auf Hardware ausgeführt.

3.1.1 Instruktionssätze

Der Befehlssatz einer Maschine bildet die eigentliche Schnittstelle zur Programmierung.

Fragen:

- **Welche Instruktionen braucht man unbedingt?**
- **Welche Eigenschaften sind notwendig?**
- **Was ist nur programmiertechnischer Komfort?**

In der Praxis gibt es viele Instruktionssätze, sehr reichhaltige (CISC) und eher magere (RISC). Sie bilden stets einen Kompromiß zwischen Anforderungen von

- höheren Programmiersprachen
- Betriebssystemen
- technischer Realisierbarkeit
- Geschwindigkeit

CISC oder RISC?

Man unterscheidet heute zwischen sogenannten

Complete **I**nstruction **S**et **C**omputer (CISC)

Beispiele: 80x86, VAX, IBM370, 68xxx,...

Sehr großer Befehlssatz, viele Befehle und Adressierungsarten. Dadurch recht kurze Programme aber sehr unterschiedliche Bearbeitungszeit pro Befehl und niedriger Durchsatz von Befehlen.

Reduced **I**nstruction **S**et **C**omputer (RISC)

Beispiele: SPARC, HPPA, Alpha, PowerPC,...

Magerer Befehlssatz, nur das Notwendigste. Sehr geringe Bearbeitungszeit pro Befehl und hoher Durchsatz von Befehlen, aber längere Programme.

Maschinenbefehle und Maschinensprache

Wir wollen nun versuchen, nach und nach den Befehlssatz *IS* einer einfachen Maschine, der **WüRC** (Würzburger RISC), und damit deren Sprache schrittweise aus Notwendigkeiten heraus zu entwickeln.

Beobachtung: Arithmetik wird stets über 2 Operanden ausgeführt und liefert ein Resultat. Es scheint natürlich, den Befehlen gleichwertig 3 Operanden zu geben, einen für das Resultat und 2 für die Operanden. Für ganze Zahlen wäre dies

```
ADD Rd,Rs,Rt        MUL Rd,Rs,Rt
SUB Rd,Rs,Rt        DIV Rd,Rs,Rt
```

Bemerkung: Wenn man unterschiedliche Zahlen und Zahlenformate hat, braucht man diese Befehle auch für jedes Zahlenformat, etwa

`ADDU Rd,Rs,Rt ...` für unsigned numbers oder

`ADDD Fd,Fs,Ft ...` für doppelt genaue Gleitkommazahlen

Wir lassen Unsigned- und Gleitkommaoperationen zunächst aussen vor, um den Rechner einfach zu halten!

Definition der Wirkung der Befehle

Die Befehle, die wir bisher angeschaut haben arbeiten alle auf einem sehr kleinen und schnellen Speicher, den **Registern**. Wir werden im Verlauf der Vorlesung noch sehen, welche Vorteile es hat, Rechnern Register zu geben. Wir definieren die Wirkung der Befehle, indem wir uns die Register als ein Array

REG[*0:#Register-1*] mit dem Bereich *0…#Register-1,*

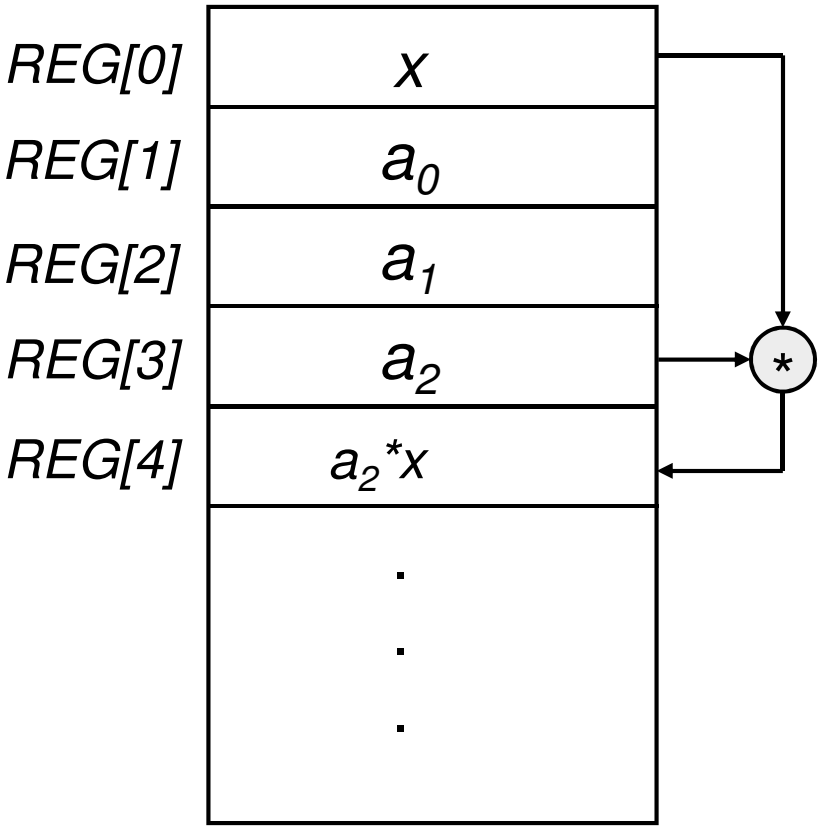
und einen Programmzähler als Variable pc vorstellen, die stets auf die auszuführende Instruktion zeigt.

Ein **Programm** sei stets eine Folge Prog[] von *0* bis *Programmlänge -1*

Prog[pc]	Wirkung
ADD Rd,Rs,Rt	**REG[d]=REG[s]+REG[t], pc=pc+1**
MUL Rd,Rs,Rt	**REG[d]=REG[s]*REG[t], pc=pc+1**
SUB Rd,Rs,Rt	**REG[d]=REG[s]-REG[t], pc=pc+1**
DIV Rd,Rs,Rt	**REG[d]=REG[s]/REG[t], pc=pc+1**

Beispiel: $a_2x^2 + a_1x + a_0$

Registerfile

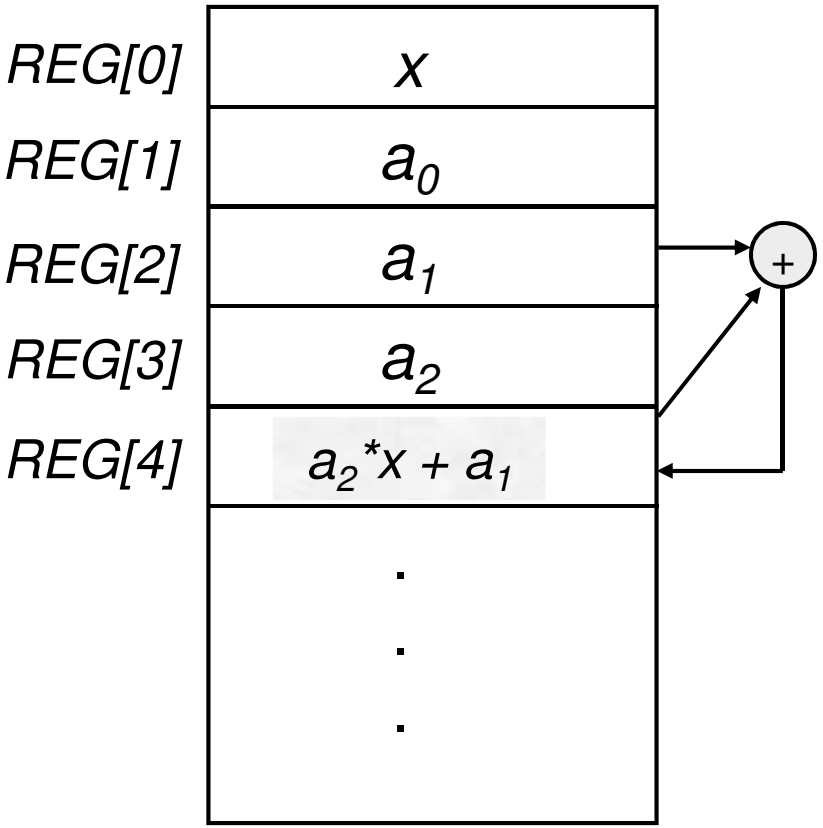


Programm

```
MUL R4,R0,R3;
ADD R4,R4,R2;
MUL R4,R4,R0;
ADD R4,R4,R1;
```

Ablauf für $a_2x^2 + a_1x + a_0$

Registerfile

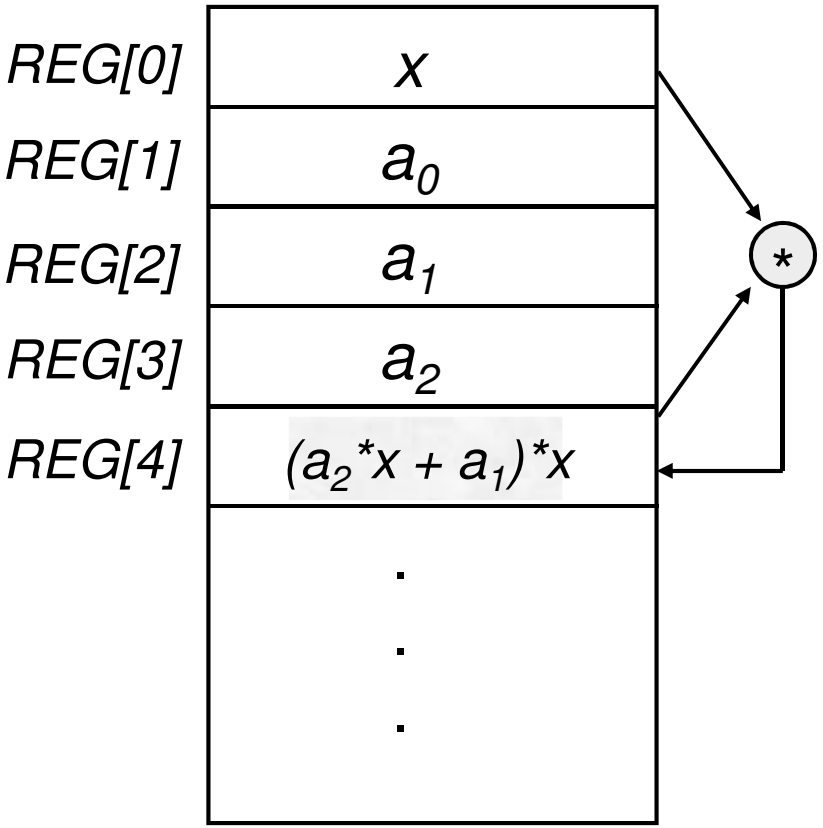


Programm

```
MUL R4,R0,R3;
ADD R4,R4,R2;
MUL R4,R4,R0;
ADD R4,R4,R1;
```

Ablauf für $a_2x^2 + a_1x + a_0$

Registerfile

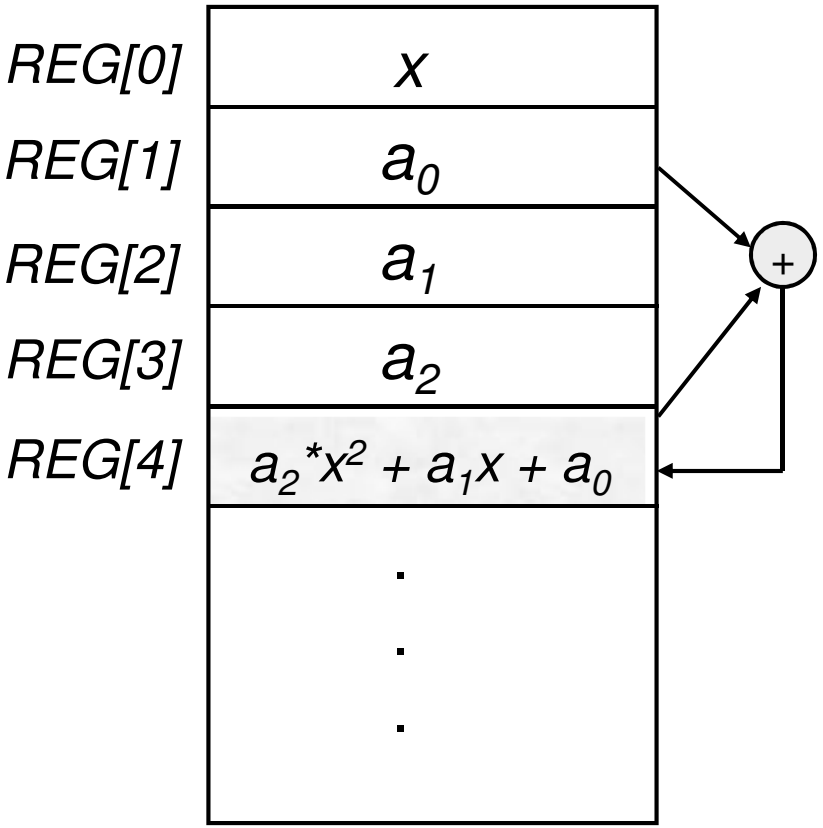


Programm

```
MUL R4,R0,R3;
ADD R4,R4,R2;
MUL R4,R4,R0;
ADD R4,R4,R1;
```

Ablauf für $a_2x^2 + a_1x + a_0$

Registerfile



Programm

```
MUL R4,R0,R3;
ADD R4,R4,R2;
MUL R4,R4,R0;
ADD R4,R4,R1;
```

Verzweigungen und Sprünge

Wir können nun zwar rechnen, das war‘s aber auch schon!

Aufgabe: Schreibe ein Programm, das in einem beliebigen Zustand der Register *REG[0] = 5* setzt, wenn *REG[1]* kleiner ist als *REG[2]*, sonst *REG[0] = -5*.

Dieses Programm existiert nicht, weil wir keine Fallunterscheidung machen können. Befehle werden stets nacheinander abgearbeitet. Auf Benutzung eines Programmzählers hätte man bisher auch verzichten können und Programme von

Prog[0] bis Prog[Programmlänge -1]

einfach der Reihe nach abarbeiten können.

Wie kann man die Reihenfolge der Abarbeitung von Befehlen beeinflussen?

Verzweigungen und Sprünge ff

Wir führen Befehle zur eigentlichen Kontrolle des Programmablaufs ein:
Verzweigungen (**B**ranches) und Sprünge (**J**umps).

Prog[pc]	Wirkung	
`JMP #t`	**`pc=pc+t+1`**	
`BEQZ Rs,#t`	**`pc=pc+1,`**	falls REG[s] != 0
	`pc=pc+t+1,`	falls REG[s] == 0
`BNEZ Rs,#t`	**`pc=pc+1,`**	falls REG[s] == 0
	`pc=pc+t+1,`	falls REG[s] != 0

Set Condition Befehle

Man erkennt, dass diese Verzweigungsbefehle nur ausreichen, um = oder ≠ zu entscheiden, und danach zu verzweigen. Um auch andere Bedingungen prüfen zu können, erweitern wir die Maschine noch um Vergleichsbefehle **`SXX Ri,Rj,Rk`** (Set Condition XX):

Prog[pc]	Wirkung
`SLT Rd,Rs,Rt`	`REG[d]=(REG[s]<REG[t]), pc=pc+1`
`SLE Rd,Rs,Rt`	`REG[d]=(REG[s]<=REG[t]), pc=pc+1`
`SEQ Rd,Rs,Rt`	`REG[d]=(REG[s]== REG[t]), pc=pc+1`
`SNE Rd,Rs,Rt`	`REG[d]=(REG[s]!= REG[t]), pc=pc+1`
`SGE Rd,Rs,Rt`	`REG[d]=(REG[s]>=REG[t]), pc=pc+1`
`SGT Rd,Rs,Rt`	`REG[d]=(REG[s]>REG[t]), pc=pc+1`

Load/Store Befehle

Bisher können wir mit den Befehlen nur Register ansprechen, aber keine Plätze im Hauptspeicher. Dies ist bei RISC Prozessoren in der Tat auch so. Sie sehen für die Kommunikation mit dem Speicher Transportbefehle (LOAD , STORE) vor:
Wir fassen dazu den Hauptspeicher als Array MEM[0:#Plätze-1] auf.

Prog[pc]	Wirkung
`LOAD Rd,Rs,#k`	`REG[d]=MEM[k+REG[s]], pc=pc+1`
`STORE Rs,Rd,#k`	`MEM[k+REG[d]]= REG[s], pc=pc+1`

Man erkennt bei diesen Befehlen noch eine weitere Besonderheit:
die Adresse des Quell-(Load) bzw. Ziel-(Store) Operanden wird **indirekt**, durch den Inhalt eines Registers plus ein **Displacement** angegeben.
Man spricht hier auch von **indirekter Adressierung** im Gegensatz zur bisher benutzten **direkten Adressierung**.
CISC Maschinen sehen indirekte Adressierung bei jedem Operanden vor, bei dem es Sinn macht (siehe Ausblick).

Adressierungsarten

Neben der indirekten Adressierung kann es auch Sinn machen, unmittelbar im Befehl den Wert des Operanden anzugeben. Dies findet man bei RISC Maschinen meist für den zweiten Quelloperanden. Bei CISC Maschinen für jeden Operanden bei dem es Sinn macht:

Prog[pc]	Wirkung
`ADDI Rd,Rs,#k`	`REG[d]=REG[s]+k, pc=pc+1`
`SUBI Rd,Rs,#k`	`REG[d]=REG[s]-k, pc=pc+1`

Man spricht hier auch von **unmittelbarer Adressierung** oder **immediate Adressierung**. Wir erlauben diese auch bei SXX Befehlen.

Adressierungsarten -- ff

Man könnte auch versuchen, die Speicherzellen bei LOAD und STORE Befehlen nur direkt anzusprechen, etwa wie folgt (absolute Adressierung):

Prog[pc]	Wirkung
`LOAD Rd,#k`	`REG[d]=MEM[k], pc=pc+1`
`STORE Rs,#k`	`MEM[k]= REG[s], pc=pc+1`

Dies hätte folgende entscheidende Nachteile:

- Man könnte mit einem Programm nur die Speicherzellen ansprechen, die explizit drin stehen. Damit könnte man keine Aufgaben lösen, die ein Berechnen von Adressen erfordern, wie zum Beispiel das Verschieben eines Speicherbereiches vorgegebener Länge.
- Man müsste die Adressen immer explizit im Befehlscode ablegen, was sehr unausgewogene Befehlsformate zur Folge hätte.

Nullregister

Man kann nun einige Befehle und Mechanismen leicht auf bisher eingeführte Befehle zurückführen, wenn man ein Register als Nullkonstante benutzt. Wir vereinbaren ab sofort:

- Das Register REG[0] hat stets den Inhalt 0.
- Schreibzugriffe auf REG[0] sind wirkungslos.

Mit dieser Modifikation kann man einige nützliche "Befehle" als Aliase auf vorhandene Befehle zurückführen.

Beispiele:

Lade eine Konstante

`LDI Rd, #k` alias **`ADDI Rd,R0,#k`**

Transport eines Registers

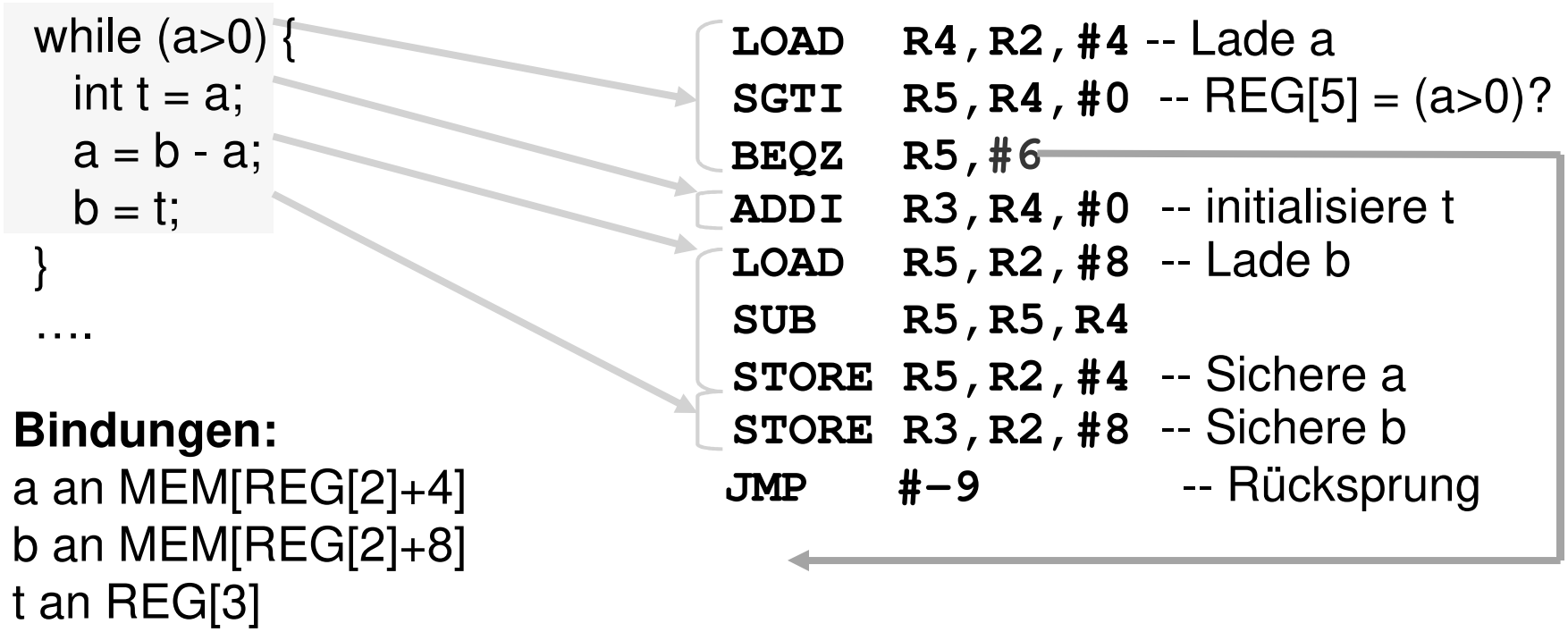
`MOV Rs, Rd` alias **`ADD Rd,Rs,R0`**

Zugriff auf absolute Speicheradresse ("Absolute Adressierung")

`LOAD Rd,#source` alias **`LOAD Rd,R0,#source`**

Hochsprache nach Maschine

Wir illustrieren, wie man ein Programmstück in Maschinensprache umsetzen könnte:



Bindungen:
a an MEM[REG[2]+4]
b an MEM[REG[2]+8]
t an REG[3]

Beispiel:

Wir wollen unsere Definitionen nun an einem einfachen Programm erproben:

Aufgabe: Schreibe ein Programm, das zu zwei positiven Zahlen ihren größten, gemeinsamen Teiler ausgibt.

Wir benutzen dazu den **Euklidschen Algorithmus**, der auf folgender Beobachtung beruht:
Für $x \leq y$ gilt:

$$ggT(x,y) = \begin{cases} y & \text{falls } x = 0 \\ ggT(y \bmod x, x) & \text{falls } x > 0 \end{cases}$$

Da das Minimum der Operanden stets abnimmt, wenn man (x, y) durch (y mod x, x) ersetzt, terminiert das Verfahren.

Das ggT Programm der WüRC

Wir entwickeln nun ein WüRC Programm, das bei der Registerbelegung REG[2] = x und REG[3] = y den ggT in REG[3] zurückliefert:

```
init:
   0:  SLE   R4,R2,R3
   1:  BNEZ  R4,#3          // für x>y muß man noch tauschen
   2:  MOV   R2,R4          // Tausch von x,y
   3:  MOV   R3,R2
   4:  MOV   R4,R3          // Tausch beendet
loop:
   5:  MOV   R2,R5          // rette x in R5
   6:  DIV   R4,R3,R2       // y mod x = y- (y/x)*x
   7:  MUL   R4,R4,R2
   8:  SUB   R2,R3,R4       // R2 = y mod x
   9:  MOV   R5,R3          // also muß nun x nach R3
  10:  SGT   R4,R2,R0
  11:  BNEZ  R4,#-7         // für x>0 das ganze nochmal!
```

Anmerkungen zum ggT-Programm

Wir haben zur besseren Lesbarkeit die Instruktion *6* im Programm mit loop markiert und die Sprungziele zu dieser Anweisung ebenfalls loop genannt. Solche Markierungen nennt man **Marken** (labels). Zu ihrer Verwendung gelten folgenden Regeln:

- **Jede Marke hat im Programm genau ein definierendes Auftreten. Sie erhält den Wert *i* falls sie vor der Anweisung *Prog[i]* auftritt.**
- **Alle anderen Auftreten werden durch den Wert <Markenwert - Platz des Auftretens - 1> ersetzt.**

3.1.2 Unterprogramme

Zur Lösung großer Programmieraufgaben fehlt uns immer noch Komfort. Eine weitere Programmieraufgabe soll dies demonstrieren:

Aufgabe:
Schreibe ein Programmstück, das das Skalarprodukt zweier *n*-stelliger Vektoren *x,y* unter folgender Speicherbelegung berechnet:

- *MEM[k]* der Wert von *n*
- *MEM[MEM[k+1]],…, MEM[MEM[k+1]+n-1]* der Vektor *x*
- *MEM[MEM[k+2]],…, MEM[MEM[k+2]+n-1]* der Vektor *y*
- *MEM[k+3]* Adresse des Resultats

Folgende Sequenz löst die Aufgabe:

```
        LOAD R2,R0,#k         // R2 halte n
        STORE R0,R0,#k+4      // Resultat temporär in MEM[k+4]
        LDI  R3,#1            // R3 dient als Zähler i
        LOAD R4,R0,#k+1       // R4 zeigt auf xi
        LOAD R5,R0,#k+2       // R5 zeigt auf yi
loop:SGT  R6,R3,R2            // Zähler > n?
        BNEZ R6,#weiter       // fertig, falls ja!
        LOAD R6,R4,#0         // hole xi
        LOAD R7,R5,#0         // hole yi
        MUL  R6,R6,R7
        LOAD R7,R0,#k+4       // hole Resultat
        ADD  R7,R7,R6         // aufaddieren auf Resultat
        STORE R7,R0,#k+4      // sichern des Resultats
        INC  R3               // Zähler und Adressen
        INC  R4               // inkrementieren
        INC  R5
        JMP  loop
weiter:LOAD R3,R0,#k+3        // Holen der Resultatsadresse
         STORE R7,R3,#0       // und Rückgabe
```

Solche Aufgaben kommen in einem größeren Programm möglicherweise sehr oft vor. Man möchte aber nicht immer die Sequenz hinschreiben müssen um sie wiederzuverwenden.

Probleme

- **Wie kann man die Sequenz so darstellen, dass sie für jedes *k* benutzt werden kann, ohne explizit dafür neu hingeschrieben zu werden? *k* ist offenbar die Adresse relativ zu der**
  - **die Argumente abzuholen, und**
  - **das Resultat abzulegen ist**
- **Wie findet man weiter an der Stelle „weiter“? Wenn wir die Sequenz abgearbeitet haben, müssen wir an die Stelle des Programmes gehen, an der die Sequenz aufgerufen wurde. Dies ist i.A. bei jedem Aufruf eine andere Stelle.**

Bemerkung: Vergleiche mit Prozeduren und Funktionen in höheren Programmiersprachen!

Beobachtungen:

- ***k*** **kommt bei fast allen Adressargumenten vor. Hier könnten sich die Displacements, die wir bei Transporten eingeführt haben, auszahlen, d.h. man könnte als Adresse *REG[j]+i* nehmen, wobei in *REG[j]* der Wert *k* steht.**
- **Wir haben einfach die Register *R2,R3,…* als Hilfszellen benutzt. Es scheint nützlich zu sein, wenn Unterprogramme stets auf die Register als Hilfszellen zugreifen könnten, sie quasi als „Freiwild“ benutzen können.**
- **Will man das Ziel bei „weiter“ bestimmen, kommt man mit relativen Sprüngen nicht aus. Wir brauchen dazu Sprünge, die indirekt über Speicherinhalte laufen.**

Indirekter Sprung

Wir erweitern die Maschine noch um die Befehle

```
JREG Rs     // Jump Register
```

mit der Wirkung

pc = REG[s];

sowie

```
JAL Rd,#f   // Jump and link
```

mit der Wirkung

pc = pc + f + 1, REG[d] = pc + 1;

Eine Aufrufsequenz hätte dann etwa folgendes Aussehen:

```
   JAL R2,#Skalarprodukt
weiter:...
```

Und die anschliessende Rückkehr einfach

```
JREG R2
```

Programmierstil

Mit diesen neuen Befehlen kann man nun die Sequenz durch **`JAL Ri,#skalarprodukt`** aufrufen, und durch **`JREG Ri`** korrekt beenden. Das Register *i* muss sowohl dem aufrufenden als auch dem aufgerufenen Programm bekannt sein. Es sichert den Rücksprung.

Wir müssen darüberhinaus auch dafür sorgen, dass

- der Speicher korrekt benutzt wird,
- die Argumente und Resultate korrekt ausgetauscht werden,
- die Register korrekt behandelt werden.

Dazu braucht man keine neuen Befehle mehr, sondern lediglich eine Vereinbarung, wie Unterprogramme zu schreiben sind und wie sie auf den Speicher wirken dürfen. Eine solche programmiertechnische Vereinbarung nennt man auch einfach einen **Programmierstil**.

Unterprogrammregeln

(U1) Das Register *SP* trägt stets die Adresse einer Speicherzelle *k*, so dass in *MEM[k+1], … , MEM[k+l]* die Argumente stehen. Wir wählen z.B. ***SP = R1*.**

(U2) Keine Speicherzelle *j <= k* wird vom aufrufenden Programm schon benutzt.

(U3) (a) Alle Register (außer SP) dürfen vom Unterprogramm verändert werden (das aufrufende Programm trägt Vorsorge). Oder

(b) Alle Register müssen unverändert zurückgeliefert werden (das Unterprogramm trägt Vorsorge).

(U4) *SP* enthält nach Rückkehr aus dem Unterprogramm den gleichen Wert wie beim Aufruf.

(U5) *R2* hält beim Aufruf stets die Rücksprungadresse.

Unterprogrammregeln ff

Man nennt U3 (a) die **caller saves** Strategie,
U3 (b) die **callee saves** Strategie.

Das Register *SP* spielt nun auch eine Sonderrolle:
Es zeigt stets den Bereich des Speichers an, ab dem nur noch freie Plätze kommen. Man kann dies auch als oberes Ende eines Stapels betrachten. Wir nennen daher *SP* auch **Stackpointer**, weil es auf das obere Ende eines Stacks zeigt.
Die Rücksprungadresse könnte man auch an einem anderen Platz aufbewahren. Allerdings muss ein Unterprogramm wissen, wo sie gesichert ist, unabhängig davon, von welchem Programm es aufgerufen wurde. Deswegen muss man sich auch auf ein Rücksprungregister einigen (U5).

Skalarprodukt Unterprogramm

Wir sind nun in der Lage, das Unterprogramm zu unserem Beispiel, "Skalarprodukt", nach (U3)(b) (callee saves) anzugeben:

Vereinbarungen:

in *MEM[SP+1]* steht *n*
in *MEM[SP+2]* steht die Adresse des Anfangs von *x*
in *MEM[SP+3]* steht die Adresse des Anfangs von *y*
in *MEM[SP+4]* steht das Resultat

Das Unterprogramm für 8 Register lautet nun wie folgt:

Eintritt ins Unterprogramm

```
Skalarprodukt: SUBI R1,R1,#7 //Platz zum Arbeiten schaffen
   STORE R2,R1,#2 // Sichere alle Register
   STORE R3,R1,#3    // die benutzt werden
   STORE R4,R1,#4
   STORE R5,R1,#5
   STORE R6,R1,#6
   STORE R7,R1,#7
   LOAD R2,R1,#8     // R2 halte nun n
   STORE R0,R1,#1 // Resultat temporär in MEM[SP+1]
   LDI  R3,#1     // R3 dient als Zähler i
   LOAD R4,R1,#9  // R4 zeigt auf xi
   LOAD R5,R1,#10 // R5 zeigt auf yi
```

Eigentliche Berechnung

```
loop:SGT  R6,R3,R2
   BNEZ R6,#return // noch zu tun?
   LOAD R6,R4,#0   // hole xi
   LOAD R7,R5,#0   // hole yi
   MUL  R6,R6,R7
   LOAD R7,R1,#1   // hole Resultat
   ADD  R7,R6,R7      // aufaddieren auf Resultat
   STORE R7,R1,#1   // sichern des Resultats
   INC  R3         // Zähler und Adressen
   INC  R4         // inkrementieren
   INC  R5
   JMP  loop
```

Rückkehr ins Hauptprogramm

```
return: LOAD R3,R1,#11 // Holen des Resultatsadresse
   STORE R7,R3,#0      // und Wegschreiben des Resultats
   LOAD R2,R1,#2      // Wiederherstellung der Register
   LOAD R3,R1,#3      // die benutzt wurden
   LOAD R4,R1,#4
   LOAD R5,R1,#5
   LOAD R6,R1,#6
   LOAD R7,R1,#7
   ADDI R1,R1,#7     //Stapel auf Aufrufniveau
   JREG R2      // und Tschüss
```

Aufruf des Unterprogramms

Annahmen:
Skalarprodukt wird selbst von einem Unterprogramm aufgerufen, das den Speicher von *MEM[REG[1]+1]* bis *MEM[REG[1]+m]* benutzt, d.h. zum Aufrufzeitpunkt sei *MEM[REG[1]]* und alle kleineren Adressen frei.
Der Wert von *n* stehe in *MEM[n_adr],* die Anfangsadressen von *x* und *y* in *MEM[x_anf]* sowie *MEM[y_anf]* und die Adresse des Ergebnisses in *MEM[e_ad].*

```
// Beginn der Aufrufsequenz
  SUBI R1,R1,#4      // Anpassen von SP um Zahl der Args.
  LOAD R7,R0,#n_adr // Hole n
  STORE R7,R1,#1     // Übergabe von n in MEM[SP+m+1]
  LDI  R7,#x_anf // Hole Anfangsadresse von x
  STORE R7,R1,#2 // Übergabe Anfangsadresse von x
  LDI  R7,#y_anf // Hole Anfangsadresse von y
  STORE R7,R1,#3 // Übergabe Anfangsadresse von y
  LDI  R7,#e_ad // Hole Ergebnisadresse
  STORE R7,R1,#4 // Übergabe Ergebnisadresse
  JAL  R2,#skalarprodukt // Aufruf
  ADDI R1,R1,#4      // Anpassen von SP auf altes Niveau
```

Kritik

Wir sehen, dass *R7* eigentlich nicht gerettet werden muss, weil es zur Parameterübergabe verwendet wird. Da wir keine Befehle haben, die nur auf dem Hauptspeicher operieren, brauchen wir stets mindestens ein Register zur Parameterübergabe.
Es liegt also auf der Hand, einige Register stets zur Parameterübergabe zu benutzen, z.B. eins für den Rückgabewert und noch ein paar weitere für die Parameter.
Dies ist dann aber fest im Programmierstil zu vereinbaren!

3.1.3 Ausblick:

Wir wollen es zunächst bei unserer ad hoc entwickelten Maschine belassen. Viele Befehle hätte man auch anders strukturieren und definieren können. Manche Dinge finden erst eine Erklärung, wenn wir wissen, welche technischen Möglichkeiten wir zum Bau einer Maschine haben. Wir schliessen daher den Abschnitt mit ein paar Anmerkungen ab:

Register:
Wir haben Register als ein programmiertechnisches Hilfsmittel im Zusammenhang mit

- Spezialaufgaben (*pc*,*REG[0],REG[1],REG[2]*)
  → **special purpose**
- allgemeine Aufgaben
  →**general purpose**

eingeführt. Für allgemeine Aufgaben deshalb, weil fast alle Befehle nur direkt ihre Operanden adressieren können, und wir dafür einen kleinen, schnellen Speicher vorhalten wollten.

Autoinkrement, Autodekrement:

Häufig werden Datenstrukturen mit aufsteigender Adresse durchmustert (z.B. Vektoren). Dazu benötigten wir im Skalarproduktprogramm die Sequenz:

```
LOAD R6,R4,#0 ...  INC R4
```

Dies könnte man auch in einen Befehl packen, indem man den Operanden des Ladebefehls automatisch inkrementiert. Man spricht dann von **Autoinkrement.** Entsprechendes gilt fürs Dekrementieren. Folgende Befehle (nicht im WüRC Befehlssatz vorhanden!) könnten dies leisten:

Prog[pc]	Wirkung
`LOAD Rd,Rs+,#1`	`REG[d]=MEM[1+REG[s]], pc=pc+1` `REG[s]=REG[s]+1`
`STORE Rs,Rd+,#1`	`MEM[1+REG[d]]= REG[s], pc=pc+1` `REG[d] = REG[d]+1`

Indirekte Operanden:

Arithmetikbefehle können nur direkt adressieren.
Deswegen haben wir bei Unterprogrammen auch Register eingeführt. Häufig ist dann die Sequenz:

```
LOAD Ri,R1,#offset ...  ADD Rr,Rs,Ri
```

Dies könnte man auch in einen Befehl packen, indem man einen oder gar alle Operanden des Befehls indirekt angeben kann. Es gibt Maschinen (CISC), bei denen jeder Operand in jeder dafür vernünftigen Adressierungsart (unmittelbar, direkt, indirekt, indirekt mit Offset...) angegeben werden kann.

Orthogonaler Befehlssatz (vgl. VAX)

Dies kann zu einem wahren Wildwuchs von Befehlen führen, so dass eine andere Notation für die Maschinensprache notwendig würde. Man könnte dann Befehle folgender Art notieren:

`ADD Ri,@Rj,@Rk+`

Addiere mit Ziel direkt adressiert, Quelle 1 indirekt und Quelle 2 indirekt mit Autoinkrement.

d.h. REG[i] = MEM[REG[j]] +MEM[REG[k]],

REG[k] = REG[k]+1 , PC = PC + 1

oder

`ADD @Ri+,@Rj-,#k`

Addiere mit Ziel indirekt mit Autoinkrement, Quelle 1 indirekt mit Autodekrement und Quelle 2 unmittelbar.

d.h. MEM[REG[i]] = MEM[REG[j]] + k,

REG[i] = REG[i]+1, REG[j] = REG[j]-1, PC = PC + 1

Primitive Instruktionssätze

In manchen Lehrbüchern wird aus didaktischen Gründen häufig eine extrem vereinfachte Maschine eingeführt, bei der sämtliche Berechnungen und Transporte über ein ausgezeichnetes Register *acc,* dem **Akkumulator** laufen:

Die Akkumulatormaschine

Arithmetikbefehle haben oft nur einen (Einadresscode) oder zwei (Zweiadresscode) weitere Operanden ausser dem Akkumulator. Folgende Tabelle gibt einen Einblick über solche (unrealistischen) Befehlssätze.

Akkumulatormaschinen

Prog[pc]	Wirkung

`ADD Rs`
`  analog SUB,`
`MUL`
	`acc = acc + REG[s], pc=pc+1`

`BEQZ #k`	`pc=pc+k, falls acc = 0` `pc=pc+1, sonst`
`LOAD Rs,#l`	`acc = MEM[l+REG[s]], pc=pc+1`
`STORE Rd,#l`	`MEM[l+REG[d]]= acc, pc=pc+1`