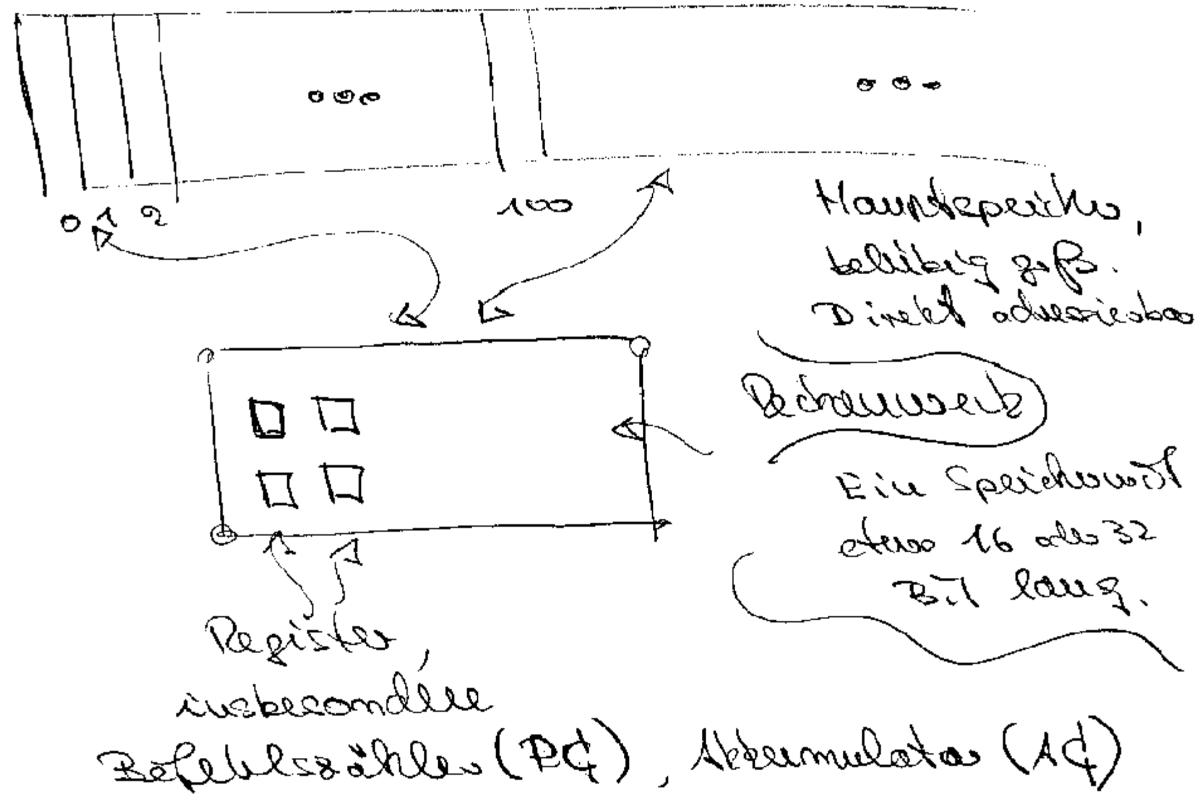
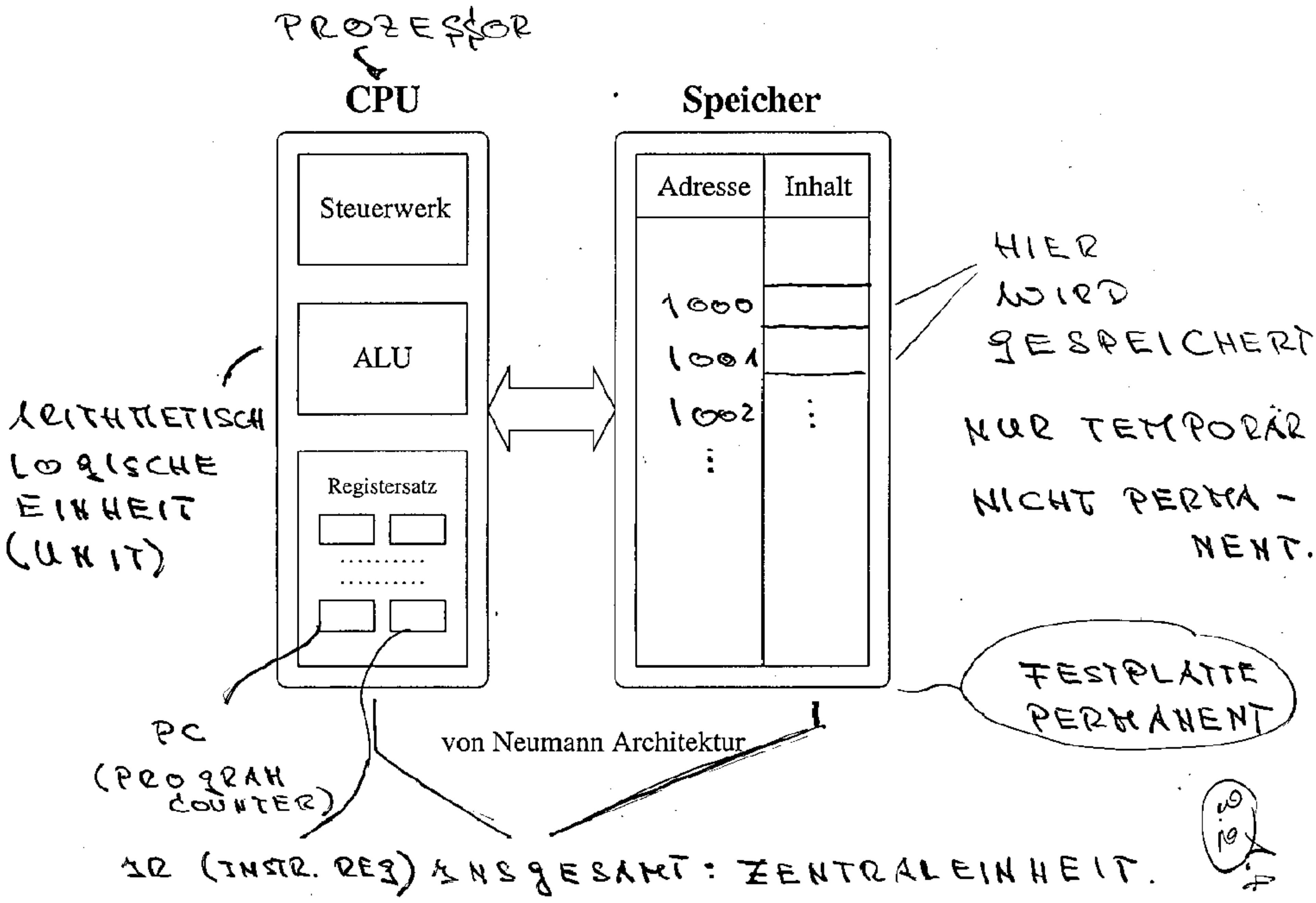
3. Zeit und Platz

Unsere Programme laufen letztlich immer auf einem von Neumann Rechner wie auf Seite 3.2. ab.

Man spricht auch von einer random access Machine (RAM), die dann vereinfacht so aussieht:





von Neumann Architektur

Typische Befehle des b.f

Load 5000 // Inhalt von Speicherplatz
// 5000 in Akkumulator

Add 1005 // Inhalt von 1005
// hinzuaddieren

Store 2000 // Inhalt des Akkumulators
// in Platz 2000
// speichern.

Programme im Hauptspeicher.
Simulation von Schleifen durch
Sprungbefehle; etwa:

Jp E // unbedingter Sprung
// nach E.

Jpz E // Ist ein bestimmtes
// Register = 0, dann
// Sprung nach E (Jump zero)

Zur Realisierung von pointern werden Speicherinhalte als Adressen interpretiert. Deshalb auch folgende Befehle:

Load ↑x // Inhalt des Platzes
// dessen Adresse
// in Platz x steht
// wird geladen

Store ↑x, Add ↑x, ...

Weitere Konventionen sind etwa:

- Eingabe in einem speziell reservierten Bereich des Hauptspeichers.
- Ebenso Ausgabe.

```
import Prog1Tools.IOTools;
public class PRIM  //Name muss gleich Dateinamen sein!!
                    // Hier einmal ein Primzahltest.
                     // Durch "return" wird das Program beendet -
                      // "Sprung ans Ende".
{
  public static void main(String[] args)
  {
   long c ,d;
   c =  IOTools.readLong("Einlesen eines langen c >= 2 zum Test:" );
   if (c == 2)
   {
   System.out.print(c + " ist PRIM");
   return;
   }
   d = 2;
   while (d * d <= c)  // Warum reicht es, bei d*d > c aufzuhoeren??
   {if (c% d == 0)
       { System.out.println( c +  "ist nicht PRIM, denn " + d + " teilt " + c);
         return;  // Hier ist das Programm zu Ende, Erlaeuterung siehe oben.
        }
   d++;  //Die Schleife hoert auf, nach dem Lauf wo hier d so gesetzt
          // wird, dass dann d * d > c ist!! Das ist ein Punkt, wo man immer
          // aufpassen muss.
   }
   System.out.println("Die Schleife ist fertig ohne ordentlichen"+
                  " Teiler, also ist "+ c + " PRIM");

  }
 }
```

while Schleife

Java Programm zum Primzahltest.

Dabei PRIM.java bei Vorlesung

AuP.

Das obige Programm wird

etwa folgendermaßen für die

RAM übersetzt. Zunächst die

while-Schleife:

```
Load d    // Sp-Platz von d
          // in Register (Akkumulator)

Mult d    // Mult. von Sp-Platz
          // d mit Akk.

Store e   // Ergebnis in e.
```

Analog in f   e-c berechnen

```
Load f    // Holen d·d-c
          // in Register

Jgz  E    // Sprung aus Ende
          // wenn d·d - c >≠ 0,
          // d.h. d·d >≠ c
```

Das hier ist nur
der Kopf der while Schleife.

Jetzt den Rumpf

$f(i\%d) \{ \ldots \}$

- d % d in Speicherplatz m ausrechnen.
- Load m
- Jgz F // Sprung wenn > 0
- Übersetzung von System.out.println(…)
- Jp E // Unbedingt aus // Ende.

d++

- F: Load d
- Inc // Register erhöhen
- Jp A // Zum Anfang

E: Stop

Als Flußdiagramm: bzw S. 14 folgende.

wichtige Beobachtung: Jede einzelne Java-Zeile führt zur Abarbeitung einer konstanten Anzahl von Maschinenbefehlen. (Konstant ⇒ unabhängig von der Eingabe c, wohl abhängig von der eigentlichen Programmzeile.)

Für das Programm PRIM gilt:

\# ausgeführte Java Zeilen bei Eingabe c

$$\leq \underbrace{a \cdot \sqrt{c}}_{\text{Schleife (Kopf und Rumpf)}} + \underbrace{b}_{\text{Anfang + Ende}}$$

a, b konstant, d.h. unabhängig von c!

Also auch

\# ausgeführte Maschinenbefehle

bei Eingabe c

$\leq$

$a' \cdot \sqrt{c} + b'$

Ausführung eines Maschinenbefehls:

im Nanosekundenbereich

\#Nanosekunden bei Eingabe c

$\leq$

$a'' \sqrt{c} + b''$.

In jedem Falle gibt es eine

Konstante k, so daß der „Verbrauch"

$\leq k \cdot \sqrt{c}$ ist ($k = a+1, a'+1, a''+1$).

Fazit: Laufzeit unterscheidet sich nur um einen konstanten Faktor von der # ausgeführten Programmzeilen.

Schnellere Rechner ⇒ Maschinenbefehle schneller ⇒ Laufzeit nur um einen konstanten Faktor schneller.

Bestimmen Laufzeit nur bis auf einen konstanten Faktor.

<u>Definition</u> ($\mathcal{O}$ - Notation)

Sei $f, g : \mathbb{N} \to \mathbb{R}$. Dann

$f(n)$ ist $\mathcal{O}(g(n))$ falls es gibt eine Konstante $c > 0$ mit $|f(n)| \leq c \cdot g(n)$ für alle hinreichend großen n. □

Das Programm PRIM hat

Zeit $O(\sqrt{q})$ bei Eingabe q.

Platzverbrauch (= # belegte Plätze)

etwa 3 also $O(1)$.

Eine Java-Zeile $\Leftrightarrow$ endliche

Anzahl Maschinenbefehle

trifft nur bedingt zu:

Solange Operanden nicht

zu groß (d.h. etwa in einem oder

einer endlichen Zahl von Speicher-

plätzen). D.h. wir verwenden

das uniforme Kostenmaß.

Größe des Operanden ist uniform 1
oder $O(1)$.

(3.12)

Zur Laufzeit unseres Algorithmus

BFS(G, s), von S. 1.22. Sei

$G = (V, E)$ mit $|V| = n$, $|E| = m$.

Datenstrukturen initialisieren: $O(n)$.

Speicherplatz
reservieren usf.

1. Array dol setzen: $O(n)$.

2. $O(1)$

3. Kopf der while-Schleife

Einmal $O(1)$

4. Einmal $O(1)$.

5. Kopf einmal $O(1)$
(gehen Adj[u] durch)
Rumpf einmal $O(1)$

3.13

In einem Lauf der while-Schleife wird 5. bis zu $n-1$-mal durchlaufen. Also 5. in $O(n)$.

Rest $O(1)$

Rest $O(1)$.

Also: while Schleife einmal:

$$\underbrace{O(1)}_{\text{„}O(1)+O(1)+O(1)\text{"}} + \underbrace{O(n)}_{O(n) \text{ für 5.}}$$

Durchläufe von 3. $\leq n$, denn schwarz bleibt schwarz. $n \cdot O(n)$

Also Zeit $O(1) + O(n^2) = O(n^2)$.

Aber das ist nicht gut genug.

Bei $E = \emptyset$, also keine Kante,

haben wir nur eine Zeit von $O(n)$,

da 5. jedesmal nur $O(1)$ braucht.

Wie oft wird 5. insgesamt

(über das ganze Programm hinweg)

betreten? Für jede Kante

genau einmal. Also das

Programm hat in 5. die Zeit

$O(m + n)$, n für die Betrachtung

von $Adj[u]$ an sich, auch wenn $m = 0$.

Der Rest des Programms hat

Zeit von $O(n)$ und nur

bekommen $O(m+n) \le O(n^2)$.

Beachte: $O(m+n)$ ist bestmöglich, da allein das Lesen des ganzen Graphen $O(m+n)$ erfordert.

Regel: Die Multiplikationsregel für geschachtelte Schleifen:

Schleife 1 $\le n$ Läufe

Schleife 2 $\le m$ Läufe,

also Gesamtzahl Läufe von Schleife 1 und Schleife 2 ist $m \cdot n$ gilt nur bedingt. Besser ist es (oft) Schleife 2 insgesamt zu zählen. Globales Zählen.

(3.16)

Topologisches Sortieren nach §.1.41

$G = (V, E)$, $|V| = n$, $|E| = m$

Initialisierung: $O(n)$ — Einmal durch Adjazenzlisten.

1. Array E-Grad bestimmen: $O(m)$

   Knoten mit Grad 0 suchen: $O(n)$

   Knoten in top. Sortierung tun

   und Adjazenzlisten anpassen: $O(1)$

2. Wieder genauso: $O(m) + O(n) + O(1)$

   ⋮

Läufe n. Also $O(m \cdot n) + O(n \cdot n)$

Also $O(m \cdot n)$ sofern $m \geq n/2$.

bis n^3, sehr groß.

S. 17

Nach S. 1.44 bekommen wir aber Linearzeit heraus:

1. und 2.: E-Grade ermitteln und Q setzen: $O(m)$

3. Ein Lauf durch 3. $O(1)$! Insgesamt n Läufe.

Also: $O(n)$ keine Schleifen.

Zeit $O(m+n)$. Zeitersparnis durch geschicktes Merken.