Inhaltsverzeichnis

Kapitel 1

Rechnerarchitektur

Speicherprogrammierung

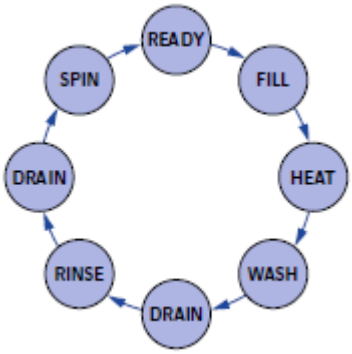
1.1.1 Festverdrahtete "Prozessoren"

DEFINITION:

Im Gegensatz zu frei programmierbaren Prozessoren sind in *festverdrahteten Prozessoren* die Bewegungen des Befehlszählers festgelegt (durch Schaltkreise bestimmt).
Da dies nur dann akzeptable ist, wenn der Prozessor nur eine einzelne festgelegte Aufgabe zu erfüllen hat, ist das in einfachen Automaten (z.B.: Waschmaschinen) der Fall.
Diese durchlaufen eine feste Abfolge von Zuständen, in denen festgelegte Aktionen ausgeführt werden.
Verzweigungen sind zwar Prinzipiell möglich, aber unveränderbar (festes Programm)

Ein gutes Beispiel für festverdrahtete Prozessoren sind Waschmaschinen:

Abbildung 1.1: Zustandsdiagramm einer Waschmaschine

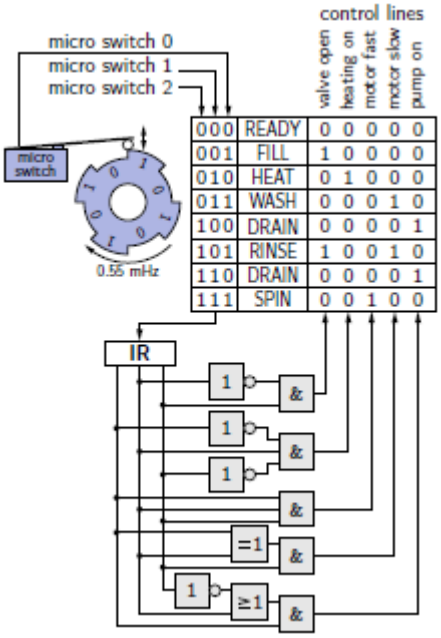


Der Prozessor kann die Waschmaschine anhand gegebener Anweisungen steuern.

- FILL → Ventil AUF - Heizung AUS
- HEAT → Ventil ZU - Heizung AN

Er besitzt z.B. Nockenscheiben, die sich mit einer gewissen frequenz drehen und dabei Mikroschalter betätigen.
Ein Schaltnetz implementiert dann die dekodierung der Ausgabe, die von den Mikroschaltern gesteuert wird.

Abbildung 1.2: Schaltung einer Waschmaschine



DEFINITION: Ablaufsteuerung

Die hier gezeigte Schaltung heißt *Ablaufsteuerung*. Sie läuft schrittweise ab, wobei von einem Schritt auf den nächsten gemäß vorgegebener Übertragungsbedingungen weitergeschaltet wird.
Gegenüber Rechnern mit freier Programmierbarkeit sind Ablaufschaltungen meist eingeschränkt, oft sogar festverdrahtet.

1.1.2 Konzept der Speicherprogrammierung

DEFINITION:

Für eine freie Programmierbarkeit von Automaten oder Rechnern ist das *Konzept der Speicherprogrammierung* entscheident:

- Anweisungen werden nicht festverdrahtet, sondern als kodierte Befehle in einem Speicher abgelegt.
- Programme sind dadurch prinzipiell austauschbar und speicherprogrammierbare Rechner folglich nicht auf eine bestimmte Aufgabe begrenzt
- Dies ermöglicht *universelle Rechenmaschinen*

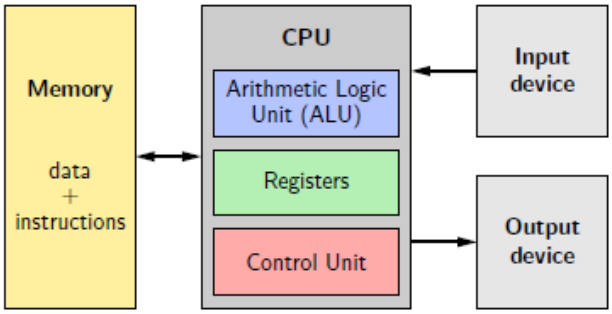
INFO:

Speicherprogrammierung wurde entscheident von John von Neumann (1945) geprägt, der Ideen von Alan Ruting (1936) weiterentwickelte, welcher wiederum mathematische Ideen von Kurt Gödel (1930) aufgegriffen hatte

Ausführen einer Anweisung erfordert einen oder mehrere der folgenden Teilschritte:

- Arithmetisch-Logische Einheit (ALU) berechnet eine funktion $f(registers)$
- Ausgabe der ALU wird in ein Register geschrieben
- Nächste auszuführende Anweisung muss bestimmt werden (Bei Verzweigungsbefehl u.U. nicht die im Speicher folgende)

Abbildung 1.3: Speicherprogrammierung



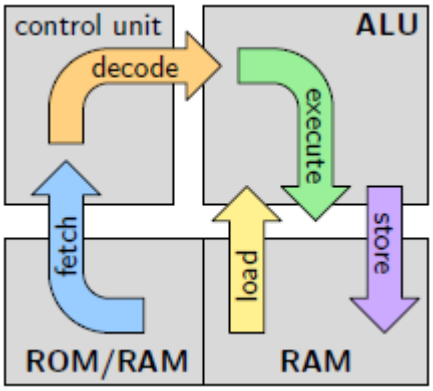
1.1.3 Befehlsaufruf, -dekodierung und -ausführung

Im Konzept der Speicherprogrammierung besteht das Ausführen eines Befehls aus folgenden drei Schritten:

- Befehlsabruf (fetch)
- Befehlsdekodierung (decode)
- Befehlsausführung (execute)

Das ist der *Fetch-Decode-Execute Cycle* (Abb. 1.4), der zur Programmausführung immer wieder durchlaufen wird.

Abbildung 1.4: Fetch-Decode-Execute Cycle



Fetch Übertragen des nächsten auszuführenden Befehls in die Steuereinheit des Prozessors

Decode Dekodieren des befehls durch die Steuereinheit des Prozessors

Execute Anweisung an ALU, die im Programmbefehl kodierte Berechnung f auszuführen

Load, Store Berechnung kann das Laden und Ablegen von Berechnungsergebnissen in den Datenspeicher erfordern

1.1.4 Rechnerarchitekturen (Harvard & von Neumann)

Bei der Harvard-Architektur (Abb. 1.5) liegen Programm und Daten in zwei verschiedenen Speichern, während sie bei der von-Neumann-Architektur (Abb. 1.6) in einem einzigen liegen

Abbildung 1.5: Harvard Architektur

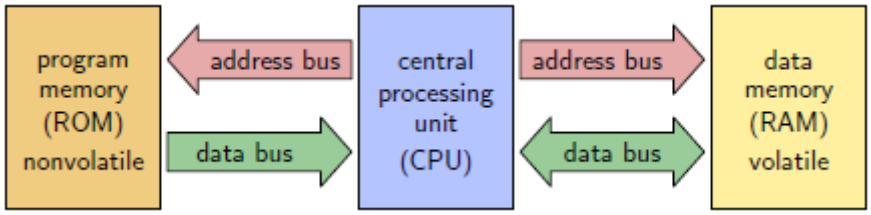
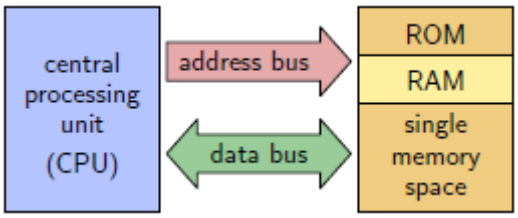


Abbildung 1.6: von Neumann Architektur



Die Hack-Plattform

1.2.1 Überblick: Hack-Rechner

Rahmendaten

- 16-Bit Harvard-Architektur
- Befehls- & Datenspeicher physisch getrennt
- 512×256 Pixel Bildschirm, Standard-tastatur
- führt Hack-Maschinensprache aus

Hauptbestandteile

- Prozessor (Central Processing Unit, CPU)
- Befehlsspeicher (32 kB Read Only Memory, ROM)
- Datenspeicher (16 kB Random Access Memory, RAM)
- "Computer" (übergeordnete Einheit)

1.2.2 Befehls- und Datenspeicher (ROM32K & RAM16K)

1.2.3 Gesamtsystem

1.2.4 Bildschirm & Bildschirmspeicher

1.2.5 Tastatur

1.2.6 hauptspeicherorganisation

1.2.7 Prozessor (Central Processing Unit, CPU)

1.2.8 Gesamtsystem (Computer On A Chip)

1.2.9 TastaRechnerarchitektur Realer Computer

Kapitel 2

Virtuelle Maschine

Hochsprachen und Übersetzung

DEFINITION: Höhere Programmiersprachen

Hochsprachen Programmiersprachen abstrahieren von konkreten Eigenschaften des Rechners. Dadurch sind sie leichter zu verstehen als Sprachen tieferer Ebenen.

Aber:

- können nicht direkt ausgeführt werden
- müssen in Sprachen tieferer Ebenen übersetzt werden

2.1.1 Direkte und zweistufige Übersetzung

Ein großes Problem der direkten Übersetzung ist, dass es viele verschiedene (Hoch-)Sprachen und Hardware-Plattformen gibt:

- Direkte Übersetzung erfordert einen Übersetzer pro Sprache und pro Plattform
- n Sprachen und m Plattformen erfordern $n \times m$ Übersetzer

Abbildung 2.1: Direkte Übersetzung

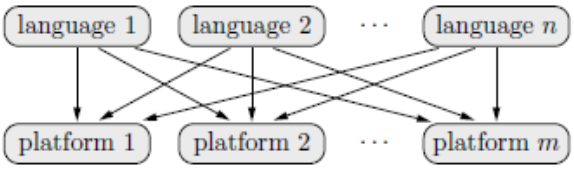
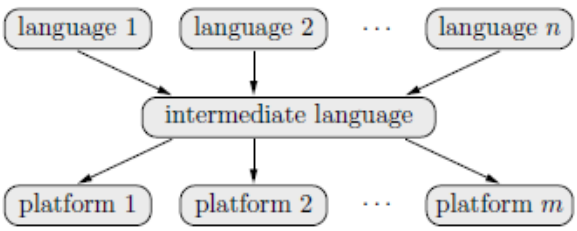


Abbildung 2.2: Zweistufige Übersetzung



Aus diesem Grund werden Hochsprachen zunächst in eine Zwischensprache übersetzt, die unabhängig von der Hardware-Plattform ist

DEFINITION: Zweistufige Übersetzung

Bei der Zweistufigen Übersetzung werden Hochsprachen und Plattformen entkoppelt:
- Erste Stufe hängt nur von Hochsprache ab (Compilation)
- Zweite Stufe hängt nur von Zielmaschine ab (translation/interpretation)

Die Zwischensprache kann als Assembler-/Maschinensprache einer virtuellen oder Pseudo-hardware-Plattform gesehen werden

2.1.2 Vor- und Nachteile Virtueller Maschinen

Vorteile
- Plattform-Unabhängigkeit
- Dynamische Optimierung auf spezielles Zielsystem möglich/einfacher
- Implementierung von Übersetzern wird einfacher

Nachteile
- Effizienzverlust ggü. direkter Übersetzung
- langsamere Ausführung
- Auch kleinere Programme benötigen virtuelle Maschine
- weniger Kontrolle über Zielcode

2.1.3 Systembasierte und prozessbasierte virtuelle Maschinen

Systembasiert
- mehrere Betriebssysteme auf einem Rechner
- so vollständige Nachbildung realer Rechner, dass Betriebssystem ausgefürht werden kann

Prozessbasiert
- Programmausführung unabhängig der Rechnerarchitektur
- Abstrahierung einzelner Programme

2.1.4 Übersetzungspfad

Übersetzung von Jack:

Jede *Klasse* hat:
- Eine Liste statischer Variablen
  → globale Variablen

Jede *Funktion* hat:
- Eine Liste von Argumenten
- Eine Liste lokaler Variablen

Die *Übersetzung* muss den Zugriff auf diese Listen organisierten

Deshalb werden den verschiedenen Listen der Klassen und Funktionen *Speichersegmente* zugewiesen. Für die lokalen Variables werden sie dynamisch bestimmt.

Abbildung 2.3: VM Translator

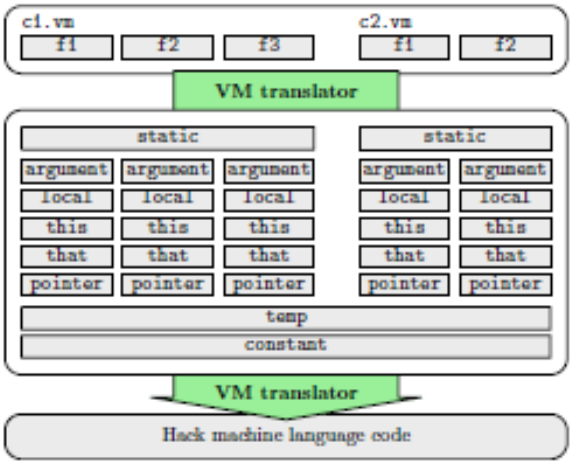


Abbildung 2.4: Jack-Quelltext

```
class c1 {
  static int x1, x2, x3;
  method int f1 (int x) {
    var int a, b;
    …
  }
  method int f2 (int x, int y) {
    var int a, b, c;
    …
  }
  method int f3 (int u) {
    var int x;
    …
  }
}

class c2 {
  static int y1, y2;
  method int f1 (int u, int v) {
    var int a, b;
    …
  }
  method int f2 (int x) {
    var int a1, a2;
    …
  }
}
```

Virtuelle Maschine des Hack-Systems

Die virtuelle Maschine für das Hack-System, die auf einem Stapel als zentrale Datenstruktur und Funktionsaufrufen arbeitet (weitgehend analog zur virtuellen Maschine von Java)
Es wird nur ein einziger 16-Bit-Datentyp verwendet (alle Daten sind 16-Bit)

Wichtig für die Betrachtung der virtuellen Maschine sind:

- Arithmetisch-logische Operationen
- Speicherzugriff
- Programmaublaufsteuerung
- Funktionsaufrufe

2.2.1 Stapel(speicher) und ihre Operationen)

DEFINITION: Stapel(-speicher)

Ein Stapel(-speicher) oder Keller(-speicher) (stack) ist eine häufig verwendete dynamische abstrakte Datenstruktur, die Daten nach dem LIFO-Prinzip speichert

Ein Stapel stellt zwei (bzw. drei) Operationen zur Verfügung:

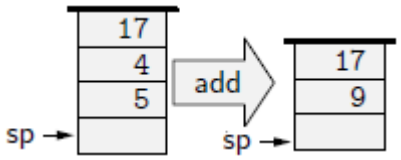
- **push** ("einkellern"): Objekt wird oben auf den Stapel gelegt
- **pop** ("auskellern"): oberstes Objekt wird vom Stapel entfernt und zurückgegeben
- **top / peek** ("nachsehen"): Optionale Option, bei der das oberste Objekt vom Stapel zurückgegeben aber nicht entfernt wird

2.2.2 Stapelarithmetik

Typische arithmetisch-logische Operation mit einem Stapel:

- Hole die beiden obersten Werte x und y vom Stapel (pop)
- Berechne Wert der Funktion $f(x, y)$
- Lege Ergebnis z auf Stapel ab ($push$)

Abbildung 2.5: Stapelarithmetik



Diese Art der Berechnung entspricht der Schreibweise arithmetisch-logischer Operationen in *Postfixnotation* oder *umgekehrter polnischer Notation*(UPN)

2.2.3 Arithmetische und Logische Operationen

Die Operationen der virtuellen Maschine sind gegenüber der Assemblersprache eingeschränkt. Anweisungen, wie *le*, *ge*, etc. ließen sich leicht hinzufügen, können aber auch anders erzeugt werden.

Anweisung	Rückgabewert	Beschreibung
add	$x + y$	Ganzzahladdition (2er-Komplement)
sub	$x - y$	Ganzzahlsubtraktion (2er-Komplement)
neg	$-y$	arithmetische Negation (2er-Komplement)
eq	-1 falls $x = y$, sonst 0	Test auf Gleichheit
gt	-1 falls $x > y$, sonst 0	Test auf größer
lt	-1 falls $x < y$, sonst 0	Test auf kleiner
and	x & y	bitweises Und
or	$x \mid y$	bitweises Oder
not	y	bitweise Negation

Der zu berechnende Ausdruck wird aus Infix- in Postfixnotation umgeschrieben:

$$2\ x - y\ 5 + \cdot = d$$

Dadurch kann er leicht mit Hilfe eines Stapels berechnet werden (siehe Abb. 2.6)

Abbildung 2.6: VM-Ausdruck

$$d = (2 - x) \cdot (y + 5)$$

```
push 2
push x
sub
push y
push 5
add
mult
pop d
```

2.2.4 Speicherzugriff, Speicheraufteilung und Speichersegmente

Speicherzugriff

Die virtuelle Maschine verwaltet vis zu 8 verschiedene (virtuelle) Speichersegmente (vgl. Java) Bisher haben wir immer auf den globalen Speicher zugegriffen. Für den Zugriff auf die virtuellen Speichersegmente gibt es andere Befehle:

- **push** *segment* *index*

  Ablegen des Inhalts von segment[index] auf den Stapel

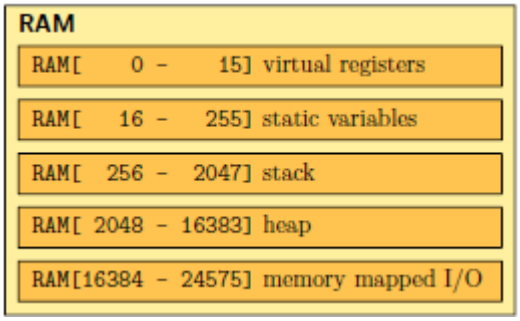
- **pop** *segment* *index*

  Speichern des obersten Stapelelements in segment[index]

Speicheraufteilung

Das Hauptziel der Virtuelle Machine ist es, den Hauptspeicher des Hack-Systems mitzuverwalten.

Abbildung 2.7: RAM im Hack-System



Speichersegmente

- *local*, *argument*, *this*, *that*:

  Direkte Abbildung auf festen Speicherbereich

  Positionen im Speicher werden in $RAM[1..4]$ gehalten (LCL, ARG, $THIS$, $THAT$)

- *pointer*, *temp*:

  pointer wird auf $RAM[3..4]$ abgebildet (*this*, *that*)

  temp wird auf $RAM[5..12]$ abgebildet

- *constant*:

  Tatsächlich virtuell (kein Speicherbereich zugeordnet)

  VM bearbeitet Zugriff von *constanti*, in dem sie die Konstante i liefert

- *static*:

  Verfügbar fpr alle Dateien mit Endung *.vm*

  Statische Variablen werden ab $RAM[16]$ zugeordnet

2.2.5 Programmablauf (bedingte Anweisungen und Schleifen)

Label definieren (ähnlich, zur Assemblersprache) Marken im Programmtext, die z.B. als Sprungziel dienen können:

- *label c*: Definiert Marke im Programmtext, z.B. als Sprungziel
- *goto c*: Springt zu einer Marke im Programmtext (unbedingter Sprung)
- $if - goto\ c$: Springt zu einer Marke im Programmtext, wenn das oberste Stapelelement nicht 0 ist (Element wird von Stapel entfernt)

2.2.6 Objekt- und Arraybehandlung

Objektbehandlung

p31

Arraybehandlung

2.2.7 Funktionsaufrufe, globaler Stapel zur Steuerung

2.2.8 Befehlssatz

2.2.9 Programmstart

Kapitel 3

Hochsprachen und Kompiler

Die Programmiersprache Jack

3.1.1 Allgemeine Syntax

- Jack-Programm ist eine Sammlung von Jack-Klassen
- Jack-Klasse: Sammlung von Jack-Unterprogrammen
- Jack-Unterprogramm:
  - Funktion
  - Methode
  - Konstruktor
- Zwingend: Eine Klasse *Main*, mit Funktion *main*

Abbildung 3.1: Jack-Programmierbare

```
class Main {

   /* Sums up 1 + 2 + 3 + …+ n */
   function int sum(int sum) {
      var int i, sum;
      let sum = 0;
      let i = 1;
      while (~(i > n)) {  // Java: (!(i > n))
         let sum = sum + i;
         let i = i + 1;
      }
      return sum;
   }

   function void main() {
      var int n, sum;
      let n = Keyboard.readInt(Ënter n: ``);
      let sum = Main.sum(n);
      do Output.printString("The result is: ");
      do Output.printInt(sum);
      do Output.println();
   }

} // Main
```

3.1.2 Datentypen und Speicheranforderung

Es gibt drei arten von Datentypen:

Basisdatentypen (primitive types)

- *int* 16-Bit-Ganzzahlen im Zweierkomplement
- *boolean* Boolscher Wert
- *char* Unicode-Zeichen

Abstrakte Datentypen (vom Betriebssystem oder Benutzer definiert)

- *String* (definiert durch Betriebssystem)
- *Fraction* (definiert durch Benutzer)
- *List* (definiert durch Benutzer)

Anwedungsspezifische Datentypen (vom Benutzer definiert)

- *CustomType*
- *AnotherCustomType*
- ...

3.1.3 Speicheranforderung

- Basisdatentypen wird bei Deklaration Speicher zugeordnet
- Objektvariablen wird bei Konstruktion (aufrufen des constructors) Speicher zugeordnet

Compiler (speziell für Jack)

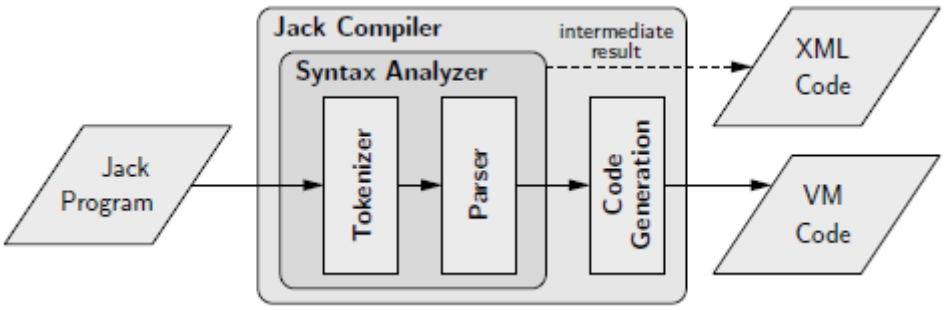
3.2.1 Architektur

Moderne Pbersetzer sind zweistufig

- **1. Stufe** (front-end): von Hochsprache nach Zwischensprache
- **2. Stufe** (back-end): von Zwischensprache nach Maschinensprache

3.2.2 Architektur, Lexikalische Analyse, Parsing

Abbildung 3.2: Jack-Compiler



Syntaxanalyse

Lexikalische Analyze (tokenizing)
Erzeugen eines Stroms von Sprachatomen (token Stream)

- Entferne Leerzeich. & Kommentaren
- Erzeuge Liste von Sprachatomen

Parsen (parsing)
Zuordnen der Sprachatome (token) zu der Sprachgrammatik

Abschließend:
XML-Ausgabe als Nachweis, dass Syntaxanalyse funktioniert

3.2.3 Kontextfreie Grammatiken

3.2.4 Parse-Bäume, Parsen durch rekursiven Abstieg

3.2.5 jack-Grammatik

3.2.6 Jack-Syntaxanalyse

3.2.7 Codeerzeugung

3.2.8 Datenbehandlung, Speicherorganisation

3.2.9 Physische Schicht (Physical Layer)