

# Zusammenfassung Informatik 2

Thomas Marc Jähnel  
jaehnel@in.tum.de

1. September 2002

## Inhaltsverzeichnis

<b>1</b>	<b>Rekursive &amp; dynamische Sorten</b>	<b>2</b>
1.1	Von Stacks und Queues . . . . .	2
1.2	Baumartige Strukturen . . . . .	2
1.2.1	Beispiele für Anwendungen von Bäumen . . . . .	2
1.2.2	LISP-Bäume . . . . .	2
1.3	Geflechte . . . . .	3
1.3.1	Verkettete Listen . . . . .	3
1.3.2	Zyklische Geflechte . . . . .	3
<b>2</b>	<b>Objektorientierte Programmierung</b>	<b>3</b>
2.1	Klassen & Objekte . . . . .	3
2.1.1	Klassen . . . . .	3
2.1.2	Kapselung und Persistenz . . . . .	4
2.2	Vererbung . . . . .	4
2.2.1	Polymorphie . . . . .	4
2.3	Erweiterungen der Objektorientierung . . . . .	5
<b>3</b>	<b>Rechnerstrukturen</b>	<b>5</b>
3.1	Codierungstheorie . . . . .	5
3.1.1	Codes einheitlicher Länge . . . . .	5
3.1.2	Codes variierender Länge . . . . .	6
3.1.3	Serien- und Parallelwortcodierung . . . . .	6
3.1.4	Shannonsches Codierungstheorem . . . . .	6
3.1.5	Huffmann-Algorithmus . . . . .	7
3.1.6	Codesicherung . . . . .	7
3.2	Binäre Schaltnetze und Schaltwerke . . . . .	7
3.2.1	Boolsche Funktionen . . . . .	7
3.2.2	Schaltnetze . . . . .	8
3.2.3	Der Halbaddierer . . . . .	8
3.2.4	Konstruktion von Schaltnetzen . . . . .	8
3.2.5	Der Volladdierer . . . . .	8
3.2.6	Zahlendarstellung . . . . .	8
3.2.7	Transistoren & Schaltwerke . . . . .	9
<b>4</b>	<b>Maschinennahe Programmierung</b>	<b>10</b>
4.1	Addressierungsarten . . . . .	10

## 1 Rekursive & dynamische Sorten

Im nun folgenden Abschnitt geht es um Sorten, die *nicht* mehr statisch in ihrer Größe beschränkt sind, sondern aus zusammengesetzten Datenelementen bestehen, die strukturiert aufgebaut sind.<sup>1</sup>

### 1.1 Von Stacks und Queues

Stacks<sup>2</sup> arbeiten bekanntermassen nach dem LIFO (Last In First Out) Prinzip. Wohingegen eine Queue oder Warteschlange nachdem FIFO Prinzip arbeitet.

Beispielhalber realisiert Broy einen Stack mit einer Pointer-Variable und eine Queue mit zwei Pointern.

### 1.2 Baumartige Strukturen

*Bäume* sind Strukturen die aus einer einfachen Informationskomponente sowie einer Menge von unabhängigen, gleichartigen Teilstrukturen bestehen. Sie spielen in fast allen Bereichen der Informatik eine wichtige Rolle.

Folgende Darstellungsarten eignen sich für Bäume:

- Graphen
- Euler-Venn-Diagramme<sup>3</sup>
- Klammerstrukturen

#### 1.2.1 Beispiele für Anwendungen von Bäumen

1. Umwandlung von Bäumen in Sequenzen: (*preorder*, *inorder*, *postorder* bezieht sich auf die Stellung des root-elements)
2. Testen auf *Auswahlbaum*: Ein Auswahlbaum  $t$  über  $N$  ist entweder leer, oder beide seiner Teilbäume sind leer, oder es muss das root Element das Maximum des linken und rechten Teilbaumes darstellen.
3. *heapsort* setzt auf einem Auswahlbaum auf und benutzt diesen zum effizienten sortieren in  $O(n \log n)$  Schritten.
4. Auswertung arithmetischer Ausdrücke in Postfixnotation: Die root-elemente enthalten die Rechenoperationen (+, -, \*, /) und nicht-leere Teilbäume. Die Operanden werden als Bäume mit leeren Teilbäumen eingeordnet.

Somit lassen sich die Ausdrücke aus infix-Notation in Postfix umwandeln (durch postorder, s.O.)

#### 1.2.2 LISP-Bäume

In LISP-Bäumen tragen nur die Endknoten (=Blätter) Informationen. Binärbäume lassen sich leicht LISP-Bäume umwandeln, sowie auch das Umgekehrte z.B. durch Einführung einer markierung für Leer funktioniert.

---

<sup>1</sup>Broy S. 207 ff

<sup>2</sup>i. Dt. auch Stapel oder Keller genannt

<sup>3</sup>kreisförmige Mengendarstellung, s.S. 219

### 1.3 Geflechte

*Geflechte* sind Tupelarten die u.A. auch Referenzen einschliessen, also verkettete Listen. Mit Hilfe von Geflechtem läßt sich das Duplizieren gewisser Informationen im Speicher verhindern.

#### 1.3.1 Verkettete Listen

Beim Speichern von Sequenzen in linearem Speicher tritt beim Einfügen das Problem auf, daß u.U. große Teile im Arbeitsspeicher verschoben werden müssen.

Dieses Problem läßt sich durch *einfach verkettete Listen* umgehen. Dabei wird an das Listenelement noch ein Pointer auf das nächste Element angehängt. Somit muss beim Einfügen nur ein Pointer verändert werden.

Soll auf die Liste von beiden Seiten zugegriffen werden, ist es geschickter eine *zweifach verkettete Liste* zu benutzen, bei denen es einen Pointer auf das vorhergehende Element und einen auf das nächste gibt. Hierbei erweist es sich häufig als günstig einen Header einzuführen, der auf das erste und letzte Element in der Liste verweist.

#### 1.3.2 Zyklische Geflechte

Eine Geflechtsstruktur heißt *zyklisch*, wenn sie eine Referenz  $x$  enthält, von der ausgehend durch Anwendung von Dereferenzieren und durch die Anwendung von Selektoren die Referenz  $x$  wieder erreicht werden kann.

Bei *zyklischen Geflechtem* ist besondere Vorsicht geboten, weil bei naiver Vorgehensweise *nichtterminierende* Funktionen entstehen können!

Bei einem *verallgemeinerten Baum* kann jeder Knoten beliebig viele Teilbäume enthalten.

Von *gemeinsamen Teilstrukturen* spricht man, wenn man innerhalb eines Geflechts die selbe Programmvariable auf unterschiedlichen Zugriffspfaden erreichen kann.

Besonders komplexe, allerdings auf effiziente Strukturen entstehen, wenn man gemeinsame Teilstrukturen und zyklische Verweise kombiniert. Als Beispiel seien hier die *gefädeltten Bäume* erwähnt.

## 2 Objektorientierte Programmierung

Objektorientierte Programmierung ist eine Weiterentwicklung der prozeduralen Programmierung unter Berücksichtigung methodischer Gesichtspunkte.

### 2.1 Klassen & Objekte

Bei objektorientierten Programmen entstehen zur Laufzeit aus *Klassen Objekte* auf die mit Hilfe von *Methoden* zugegriffen werden kann.

#### 2.1.1 Klassen

Eine Klasse *kapselt* Funktionen und Prozeduren (in der obj. orientierten Programmierung *Methoden* genannt), sowie Konstanten und Variable (hier *Attribute*). Eine Klasse trägt natürlich auch einen Namen. Mit Hilfe dieses Namens deklariert die Klasse auch einen neuen Datentyp (Sorte) der die Objekte dieser Klasse bestimmt.

Ein neues Objekt aus einer Klasse wird mit Hilfe eines *Konstruktors*<sup>4</sup> erzeugt, der eine Art Pointer auf das neue Objekt zurückliefert. Die somit entstandene Referenz auf ein Objekt nennt man *Objektidentifikator*. Allerdings ist ein Dereferenzieren hier nicht möglich.

Die Klassendefinition führt nun dazu, dass jedem dieser Objektidentifikatoren die zum Export freigegebenen Methoden zur Bearbeitung der Objekte zur Verfügung stehen. Man spricht dann von einer *syntaktischen Schnittstelle*.

Attribute werden hier nicht als Teil der syntaktischen Schnittstelle angesehen, da auf sie nicht direkt zugegriffen werden kann. Anzumerken ist auch die Ähnlichkeit der Konzepte von Signatur und syntaktischer Schnittstelle. Die Kenntnis über die syntaktische Schnittstelle einer Klasse genügt zwar um *syntaktisch* korrekt auf die Klasse zuzugreifen, sie sagt aber nichts über die Wirkung der Methoden und somit über die *Semantik* der Klasse aus.

Eine Klasse A heißt zu einer Klasse B *funktional ersetzbar*, wenn A die syntaktische Schnittstelle von B umfaßt und in beliebigen Programmen die eine durch die andere ersetzbar ist, ohne dass sich das Ergebnis des Programms ändert. Man nennt die Klassen dann auch *wirkungsgleich*.

### 2.1.2 Kapselung und Persistenz

Das Besondere der Objekte und Klassen ist die Kapselung (=Zusammenfassung) von Daten und Algorithmen in einer Einheit. Somit werden Differenzen in Lebensdauer und Gültigkeit von Variablen vermieden. Die Lebensdauer der Attribute und Methoden eines Objekts besteht über die Dauer eines Aufrufs einer Methode fort.

Wir sprechen von *Persistenz* wenn Objekte über die Laufzeit eines Programmes weiterexistieren und somit eine *persistente* Datenkapsel bilden.

## 2.2 Vererbung

Die *Vererbung* definiert Relationen zwischen Klassen und erlaubt somit auszudrücken, dass eine Klasse ein Spezialfall einer anderen Klasse ist. Desweiteren wird damit die Übernahme von Attribut- und Methodendeklarationen ausgedrückt (=codereuse). Die Vererbung führt damit zum Aufbau von Beziehungshierarchien. Insbesondere erbt natürlich jede *Unterklasse* die *erweiterte syntaktische Schnittstelle*<sup>5</sup> ihrer Oberklasse.

Wir unterscheiden nun zwischen einfacher (*singulärer*) und mehrfacher (*multipler*) Vererbung. Bei multipler Vererbung erbt die Unterklasse von mehreren anderen Klassen. Dies kann u.A. zu Komplikationen durch Namenskonflikte führen.

In den meisten objektorientierten Sprachen lassen sich geerbte Methoden in einer Unterklasse auch durch Neue (in der Regel syntaktisch gleiche) überschreiben.

### 2.2.1 Polymorphie

Wenn wir eine Methode der Oberklasse in der Unterklasse überschreiben hängt es davon ab, von welchem Typ das aufrufende Objekt ist, welchen Methode beim Aufruf zum Zug kommt. Hier spricht man dann also von *Polymorphie* und *late Binding*.

---

<sup>4</sup>Im Broy create-Methode genannt

<sup>5</sup>syntaktische Schnittstelle + Attribute

### 2.3 Erweiterungen der Objektorientierung

Das Konzept der *abstrakten* Klasse hat sich weit verbreitet. Hier geht es darum, eine Klasse zu definieren, die nie direkt als Objekt erzeugt werden kann, sondern nur der Vererbung dient. Sie darf Methoden enthalten bei denen *kein* Rumpf angegeben ist. Methoden die nicht überschrieben werden dürfen, bezeichnet man als *final*.

Eine andere Erweiterung betrifft die Attribute. Durch das Schlüsselwort *public* gekennzeichnete Attribute lassen sich direkt über die Objekte ansprechen, ohne dass dafür eigens Zugriffsfunktionen definiert werden müssen.

## 3 Rechnerstrukturen

### 3.1 Codierungstheorie

Für die maschinelle Informationsverarbeitung sind bestimmte Repräsentationssysteme besser geeignet als andere. Bei der Auswahl eines geeigneten Codierungsverfahrens stehen zwei Ziele im Vordergrund:

- Ökonomie der Darstellung (mögl. kurze Codes)
- Fehlersicherheit

Im Folgenden wird eine endliche Menge von Zeichen, auch *Zeichenvorrat* genannt, vorausgesetzt. Ein linear geordneter Zeichenvorrat wird auch *Alphabet* genannt. Die Menge der endlichen Zeichenfolgen über dem Zeichenvorrat heißen auch *Wörter*.

#### 3.1.1 Codes einheitlicher Länge

Bei einer Binärcodierung, bei der jedes Zeichen auf ein Binärwort gleicher Länge abgebildet wird, sprechen wir auch von *Codes einheitlicher Länge*. Aus technischen Gründen (Registerlänge) ist diese Art der Repräsentation sehr gebräuchlich.

Der *Hammingabstand* ist die Anzahl der "Stellen" an denen sich zwei gleichlange Binärwörter unterscheiden. Der Hammingabstand einer Codierungsabbildung ist der *kleinste* Hammingabstand zwischen zwei *verschiedenen* Codewörtern. Eine Codierung mit einem großen Hammingabstand ist somit Fehlertoleranter (u.U. erlaubt sie sogar die Korrektur von Fehlern), allerdings werden auch mehr Bits benötigt.

Ein Gray-Code ist ein Code einheitlicher Länge, bei dem der die Codierung zweier aufeinanderfolgender Zeichen sich in genau einem bit unterscheidet und somit der Hammingabstand genau 1 beträgt. Von einem *zyklischen Gray-Code* spricht man, wenn sich zusätzlich noch die Codierung des ersten und letzten Zeichens einer Ordnung nur in einem bit unterscheiden.

Größere Hammingabstände können z.B. durch die Verlängerung einer bestehenden Codierung erreicht werden. Als Beispiel sei hier die Einführung eines Paritybitsbit<sup>6</sup> genannt.

Verwendet man dagegen die lexikographische Ordnung des Zeichenvorrates, so spricht man hier von einer *direkten* Codierung. In diesem Zusammenhang werden auch oft sog. Gewichte verwendet, die der jeweiligen Stelle im Code einen Faktor zuordnen<sup>7</sup>.

<sup>6</sup>= 1 falls Quersumme gerade, sonst 0

<sup>7</sup>s.a. Broy S. 273

Ein *Kettencode* ist ein Binärcode, der durch schrittweises Vorbeiwandern einer Zeichenfolge an einem Ablesefenster entsteht. Bei einem 4-Bit Kettencode ist also das Ablesefenster 4 Stellen breit.

### 3.1.2 Codes variirender Länge

Codes variirender Länge sind maschinell schwieriger zu handhaben und finden sich, bedingt durch heutige Rechnerarchitekturen nur sehr selten.

Sie haben den Vorteil, dass man bei Kenntnis der relativen Häufigkeit des Auftretens von bestimmten Zeichen für die häufigeren auch kurze Codewörter verwenden kann, und somit die durchschnittliche Länge der Codewörter klein halten kann. Da in der Regel aber Zeichenfolgen codiert werden, entsteht bei Codes mit variierender Länge das Problem, dass man schwieriger erkennt wann ein Codewort aufhört und ein neues beginnt.

### 3.1.3 Serien- und Parallelwortcodierung

Diese beiden Begriffe sind insbesondere wichtig, wenn es darum geht, die codierte Information über Leitungen zu Übertragen. Hier unterscheiden wir dann zwischen der *Parallelen-* oder der *Serienübertragung*.

Bei der Serienübertragung benötigt man für ein Wort der Länge  $n$  auch  $n$  Takte zur Übertragung, wohingegen man bei der Parallelübertragung  $n$  Leitungen aber nur einen Takt benötigt.

Bei der Serienübertragung von Wörtern läßt sich die Menge der in Frage kommenden Dekodierungsergebnisse mit jedem übertragenen bit weiter einengen. Bei Codes mit variirender Länge muss das Codierungsverfahren allerdings der *Fano-Bedingung* genügen, damit die *Injektivität* und somit die Dekodierbarkeit der Codierung gewährt ist<sup>8</sup>. Die Fano-Bedingung lautet: *Kein Codewort darf der Anfang eines anderen sein.*

### 3.1.4 Shannonsches Codierungstheorem

Wir sprechen von einer *stochastischen* oder *Shannonschen Nachrichtenquelle*, wenn wir die mittlere Häufigkeit für das Auftreten eines jeden Zeichens genau kennen<sup>9</sup>.

Der *Informationsgehalt* eines Zeichens entspricht dem Kehrwert der Häufigkeit des Auftretens dieses Zeichens<sup>10</sup>.

Mit *Entropie* bezeichnen wir den mittleren Entscheidungsgehalt pro Zeichen der sich wie folgt errechnet:

$$H = \sum_{a \in A} p_a * \text{ld}\left(\frac{1}{p_a}\right) \quad (1)$$

wobei  $\text{ld}(\dots)$  der Entscheidungsgehalt eines bestimmten Zeichens  $a$  ist. Die Entropie ist ein Maß für die Schwankung der mittleren Häufigkeit der Zeichen. Ist die Wahrscheinlichkeit der Zeichen sehr unterschiedlich, so ist die Entropie sehr klein und umgekehrt<sup>11</sup>.

Die *mittlere Wortlänge* einer Codierung erhalten wir mit folgender Formel:

$$L = \sum_{a \in A} p_a |c(a)| \quad (2)$$

<sup>8</sup>dass dies genügt wird auf S. 279 begründet

<sup>9</sup>Natürliche Sprache fällt nicht in diese Kategorie, da die Wahrscheinlichkeit für das Auftreten eines bestimmten Zeichens auch von den vorhergehenden abhängt

<sup>10</sup>Erläuterung S. 280

<sup>11</sup>d.h. die Entropie ist am größten, wenn alle Zeichen gleich wahrscheinlich sind

dabei beschreibt  $|c(a)|$  die Wortlänge eines binären Codeowortes.  
 283 - 285 ausgelassen!

### 3.1.5 Huffman-Algorithmus

Der Huffman-Algorithmus dient dazu eine optimale Codierung für ein gegebenes Alphabet mit einer gegebenen Wahrscheinlichkeitsverteilung zu finden. Man geht zur Anwendung folgendermaßen vor:

1. Man schreibt sich oben alle Zeichen mit ihren Wahrscheinlichkeiten der Reihe nach hin.
2. Dann verbindet man immer die beiden mit der geringsten Wahrscheinlichkeit zu einem neuen Knoten der dann die Summe der beiden Wahrscheinlichkeiten und eine neues Zeichen, bestehend aus den zwei kombinierten, als Wert trägt. Das macht man so lange bis ein fertiger Baum entsteht.
3. Nun geht man von der Wurzel nach oben und schreibt jeweils dem einen Ast ein  $O$  zu und dem anderen ein  $L$  und hängt dabei immer das vorne an, was der Knoten unterhalb schon als Codierung hatte.

Dass das Verfahren funktioniert, läßt sich durch vollständige Induktion beweisen<sup>12</sup>. Im Prinzip geht der Beweis davon aus, dass jede 1-Bit-Codierung für zwei Zeichen optimal ist. und deshalb reduziert man bei jedem Schritt die Menge der Zeichen um eins, da man ja immer zwei alte durch ein neues ersetzt.

### 3.1.6 Codesicherung

Zur *Codesicherung* lassen sich beispielweise bei Codes einheitlicher Länge, die evtl. nicht benötigten Codewörter dazu nutzen, um

- die Wahrscheinlichkeit, ein falsches Zeichen zu erhalten, möglichst gering zu halten
- die Wahrscheinlichkeit, dass Fehler erkannt werden, möglichst groß zu machen
- die Wahrscheinlichkeit, dass aufgetretene Fehler korrigierbar sind, möglichst groß zu machen

Desweiteren läßt sich aus ein *vergrößerter Hammingabstand* dazu nutzen, um Fehler zu vermeiden.

## 3.2 Binäre Schaltnetze und Schaltwerke

### 3.2.1 Boolesche Funktionen

Eine *boolesche Funktion* ist eine  $n$ -stellige Abbildung auf den Wahrheitswerten:

$$f : B^n \rightarrow B \quad (3)$$

Zur Darstellung boolescher Abbildungen mit *freien Identifikatoren* eignet sich die *vollständige disjunktive Normalform* oder auch *DNF* genannt<sup>13</sup>.

<sup>12</sup>siehe Übungsblatt 5

<sup>13</sup>Beispiel, s.S. 301

### 3.2.2 Schaltnetze

Eine *Schaltfunktion* ist nun eine Abbildung folgender Art:

$$f : B^n \rightarrow B^m \quad (4)$$

*Schaltnetze* lassen sich nun sowohl *parallel* als auch *sequentiell* kombinieren.

### 3.2.3 Der Halbaddierer

Der Halbaddierer hat zwei Eingänge (a und b) und zwei Ausgänge (s und ü) und realisiert eine binäre Addition "ohne Übertrag":

$$s = (\neg a \wedge b) \vee (a \wedge \neg b) \quad (5)$$

$$\ddot{u} = a \wedge b \quad (6)$$

Eine Funktionstermdarstellung des Schaltbildes auf S. 313 sieht folgendermaßen aus:

$$HA = (V \parallel V) \cdot (I \parallel P \parallel I) \cdot (AND \parallel OR) \cdot (V \parallel NOT) \cdot (I \parallel OR) \cdot (I \parallel NOT) \quad (7)$$

Die in 7 verwendeten Symbole entsprechen den gängigen Bezeichnungen: V = Verzweigung, I = Identität, P = Permutation . . .

### 3.2.4 Konstruktion von Schaltnetzen

Schaltnetze werden nach folgendem Schema konstruiert:

1. Ableiten von Termen für alle Ausgänge mit Hilfe der DNF
2. Umformen der Terme in funktional äquivalente mit Hilfe der Gesetze der Schaltungs algebra
3. Parallele und sequentielle Komposition ergibt das gewünschte Schaltnetz

### 3.2.5 Der Volladdierer

Ein *Volladdierer*<sup>14</sup> berücksichtigt nun auch einen Übertrag bei der Addition und läßt sich mit Hilfe von zwei Halbaddierern realisieren:

$$VA = (HA \parallel I) \cdot (I \parallel HA) \cdot (OR \parallel I) \quad (8)$$

Durch die Kombination von *n* Volladdierern läßt sich somit ein *n*-bit Additionsnetz realisieren, wobei der letzte Übertrag dann als Overflow Signal weitergeleitet werden kann, um zu signalisieren, dass bei der Berechnung ein Überlauf stattgefunden hat.

### 3.2.6 Zahlendarstellung

Für die Codierung der *ganzen Zahlen* eignet sich unsere bisherige direkte Codierung in Binärzahlen nicht. Deshalb existieren verschiedene andere Codierungen. Wir gehen im folgenden von *n*-stelligen Zahlen aus, womit sich eine Codewortlänge von *n*+1 Bit ergibt.

Bei der *1-Komplement-Darstellung* auf dem Intervall  $[-2^n, 2^n]$  geht man wie folgt vor:

<sup>14</sup>Wertetabelle und Schaltbild, s.S. 316 ff

1. Bei positiven Zahlen:  $b_{n+1} = 0$  alle anderen entsprechen dem Standard
2. Bei negativen Zahlen: genau das bitweise Inverse zu 1.

Nachteilig daran ist, dass somit zwei verschiedene Codierungen für die Null existieren, nämlich: LLL ... LLL und OOO ... OOO.

Dieses Problem läßt sich mit der *2-Komplement-Darstellung* umgehen. Hier wird zu dem bitweisen Inversen noch 1 addiert um das negative einer Zahl zu erhalten. Somit ist  $5 = \text{OLOL}$  und  $-5 = \text{LOLL}$ . Dadurch ergibt sich der asymmetrische Wertebereich von  $[-2^n, 2^n - 1]$ . Dies ist die Darstellungsart die in heutigen Rechnerarchitekturen ausschließlich verwendet wird.

### 3.2.7 Transistoren & Schaltwerke

Funktionsweise von Transistoren ist auf S. 337 ff beschrieben. Kurz: Liegt an B Strom an so fließt Strom von K nach E.

*Schaltwerke* sind Schaltnetze bei denen auch Rückkopplungen erlaubt sind. Mit Hilfe von Rückkopplungen lassen sich Zustände speichern.

Als einfaches Beispiel soll hier das *R-S Flip-Flop*<sup>15</sup> dienen. Solange kein Signal an den Eingängen anliegt, wird der vorherige Zustand der Ausgänge gehalten. Wenn der Set Eingang auf L gesetzt wird, wird Q auf L und  $\bar{Q}$  auf O gesetzt. Beim Reset passiert genau das Umgekehrte. Beide Eingänge auf L zu setzen ist nicht erlaubt und führt zu einem *unbestimmten* Zustand.

---

<sup>15</sup>Reset-Set Flip-Flop, Schaltbild: s.S. 356

## 4 Maschinennahe Programmierung

### 4.1 Adressierungsarten

**Absolute Adressierung** Der Operand wird durch eine absolute Speicheradresse angegeben.

**Direkter Operand** Hier wird der Wert *direkt* im Befehl übergeben. Dies kennzeichnet ein *I* vor dem Wert.

**Registeradressierung** Hier wird ein Register angesprochen und dient dann als Operand. Zum Beispiel *Rxx* oder *SP*.

**Relative Adressierung** Hier wird der *Wert* des Registers als Adresse verwendet. Zum Beispiel:

```
R5 := 1234h
MOVE W 10, !R5    -- S[0x1234] := S[0x000A]
MOVE W I 10, !R5  -- S[0x1234] := 0x000A
```

Relative Adressierung existiert auch in einer *indizierten* Form. Der so angegebene Wert wird automatisch mit der angegebenen Operandenlänge multipliziert:

```
MOVE W I 5, 2+!R2/2/  -- S[R2+2+2*4] := 0x0005
```

**Indirekte Adressierung** Hier wird eine Speicherzelle mit relativer Adressierung ermittelt, deren Inhalt dann die endgültige Adresse enthält. Beispiel:

```
MOVE W I 5, !!R6      -- S[S[R6]] := 0x0005
MOVE W I 5, !(2 + !R6) -- S[S[2+R6]] :=...
```

Und das ganze nochmal in *indizierter* Form:

```
MOVE W I 5, !(2+!R6)/2/  -- S[S[2+R6+2*4]] :=...
```

**Kelleradressierung** Entspricht so ungefähr *push* und *pop*.

```
MOVE W I H'10000', SP  -- SP init
MOVE W I 7, -!SP      -- Wert 7 auf den Stack
MOVE W 1, !SP+        -- S[0x0001] := 7
```

danach ist der Stack wieder leer und SP zeigt wieder auf 0x10000. *Diese Adressierungsart ist nicht auf SP beschränkt.*