

Grundlagen der Informatik

Simon Krenger, Gilles Bertholet

December 3, 2011

Chapter 1

Einführung

1.1 Lizenzen und Patente

Unter einer Lizenz versteht man im rechtlichen Sinne ein Gebrauchsrecht. Wenn Software gekauft wird, so wird das Gebrauchsrecht erworben. In der Informatik gibt es verschiedene Lizenzen, unter anderem folgende:

1. Open Source Lizenzen, z.B. die EULA: End User License Agreement
2. Closed Source Lizenzen, z.B. die GPL: GNU Public License (GNU = GNU Is Not Unix, → Richard Stallman)

Konkret ist die Situation in der Schweiz irregulär.

Legal, Illegal, Scheissegal

Lizenzen und Patente werden von der sogenannten "Intellectual Property (IP)" Lobby gefördert. Verfechter dieser Lobby werden mit den folgenden Begriffen in Verbindung gebracht:

1. Lizenzen
2. Patente
3. Trademarks (TM, entspricht in etwa ® [= Registrierte Marke])

Auf der anderen Seite haben wir die Bewegung der freien Software, die mit der GPL eine Lizenz erschaffen haben, welche die folgenden Eigenschaften besitzt (unvollständige Liste):

1. Enthält das Recht zum Kopieren
2. Enthält das Recht zu Reproduzieren
3. Neue Programme, die auf GPL Code basieren, müssen ebenfalls unter der GPL veröffentlicht werden.

Als Konsequenz kann gesagt werden, dass es einen Gegensatz zwischen der "Open Source" Bewegung und der "Intellectual Property" Bewegung gibt.

$$\text{Intellectual Property} \iff \text{Open Source}$$

In der Schweiz gibt es keine Softwarepatente.

Definition 1. *Ein Patent ist ein hoheitlich erteiltes gewerbliches Schutzrecht für eine Erfindung. Der Inhaber des Patents ist berechtigt, anderen die Benutzung der Erfindung zu untersagen.*

Ein Patent ist damit ein vom Staat erteiltes Monopol auf Bewirtschaftung eines Objektes (zeitlich begrenzt).

$$\text{"Patente fördern Innovation"} \iff \text{"Patente verhindern Innovation"}$$

Die Überlegungsweise bezüglich Lizenzierung hängt stark vom vertretenen Standpunkt aus. Unter einem Paradigmawechsel versteht man einen Bruch in der Überlegungsweise.

Beispiel 1. *Wir zeigen einen Paradigmawechsel anhand des Transportwesens vor und nach dem ersten Weltkrieg.*

<i>Vor 1914</i>	<i>Nach 1918</i>	
<i>Pferde</i>	<i>Autos</i>	
<i>Stroh</i>	<i>Tankstellennetzwerk</i>	
<i>Stallknecht</i>	<i>Automechaniker</i>	(1.1)
<i>Sattler</i>	<i>Strassennetz</i>	
<i>Schmied</i>	<i>Automechaniker</i>	
<i>Kutscher</i>	<i>Fahrer</i>	

In der Vergangenheit hat in der Informatik ein Paradigmawechsel vom Softwareverkauf hin zum Lizenzverkauf stattgefunden.

→ Es werden keine Lösungen, sondern Lizenzen verkauft

1.2 Begriffe und Definitionen

1.2.1 Funktion / Struktur

Beispiel 2. *Wir erklären den Unterschied zwischen Funktion und Struktur anhand von zwei Beispielen.*

1. *Auto*

Funktion: Mobilität

Struktur: Vier Räder, ein Motor, ein Steuerrad

2. *Buch*

Funktion: Informationsübermittlung

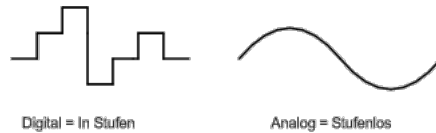
Struktur: Inhaltsverzeichnis, Prolog, Kapitel, Glossar

D.h. man erkennt auch bei einem chinesischen Buch eine Struktur, ohne den Inhalt zu verstehen.

Durch eine sogenannte Strukturanalyse bei einem Codebeispiel wird nicht die Funktion ermittelt, sondern die Gesamtstruktur, die einzelnen Komponenten.

→ Aufzählen der Komponenten

1.2.2 Analog / Digital

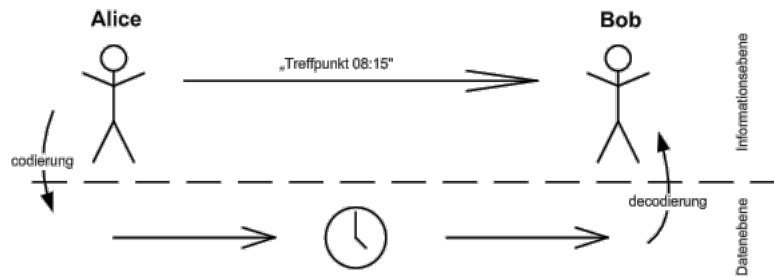


Analog = Stufenlos

Digital = In Stufen / Zuständen

Wir haben die gleichen Konzepte in der Algebra: \mathbb{R} ist nicht abzählbar, entspricht in etwa dem analogen Konzept. \mathbb{N} und \mathbb{Q} sind abzählbar, entsprechen daher eher dem digitalen Konzept.

1.2.3 Daten / Informationen



Daten sind losse vom Kontext, Informationen haben einen Kontext. Man spricht davon, dass die Informationsebene kontextbehaftet ist. Reine Daten ohne Kontext können kaum interpretiert oder verarbeitet werden.

1.2.4 Syntax / Semantik

Syntax = Form \iff Sinn = Semantik

Syntax = Struktur \iff Funktion = Semantik

Ein Computer kann zwar die Syntax überprüfen, bei der Semantik (Sinn und Funktion des Programms) kann der Computer keine Überprüfung durchführen.

1.3 Definition Computer

Stichworte: "Rechner", "Hirn", "Input → Datenverarbeitung → Output"

Ein Computer ist ein abstrakter Begriff, wobei die konkrete Implementation auf verschiedene Arten passieren kann. Ein Computer kann analog oder digital sein, mechanisch, elektrisch oder biochemisch realisiert sein.

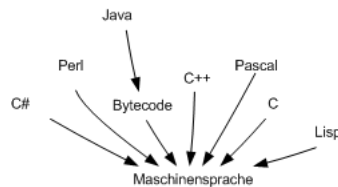
Um den Computer zu definieren müssen wir eine Metabetrachtung (von aussen) durchführen. Bei einer Metabetrachtung wird ein formales System (System mit Regeln) extern und nicht von innen heraus betrachtet.

Alan Turing (1912 - 1954) definiert einen Computer folgendermassen:

Eine Maschine zur Manipulation von Zeichen

In der theoretischen Informatik unterscheidet man bei Programmiersprachen zwischen "turing-vollständigen" Sprachen und "nicht-turing-vollständigen" Sprachen. "Turing-vollständige" Sprachen können eine Turing-Maschine nachbilden. Wenn ein Problem in einer "turing-vollständigen" Sprache gelöst werden kann, so kann es in jeder anderen beliebig "turing-vollständigen" Sprache ebenfalls gelöst werden. Lediglich die Konzepte (z.B. Objektorientierung, Logische Programmierung, ...) ändern sich.

1.4 Programiersprachen & Algorithmen



Hochsprachen nehmen dem Programmierer die trivialen Probleme ab (Speicher-allozierung, Schleifen). Als Informatiker müssen wir die Konzepte hinter den Programmiersprachen verstehen. Dazu gehören neben der objektorientierten Programmierung beispielsweise die prozedurale oder logische Programmierung.

Es kann gesagt werden:

Ein Algorithmus beschreibt einen Lösungsweg

Ein Programm ist die Implementation eines Algorithmus

Bei der Implementation eines Algorithmus beschreibt die Programmiersprache die Regeln des formalen Systems.

Bei der Programmierung wird im Grunde wie folgt vorgegangen:

Problem → Metabetrachtung → Algorithmus → Lösung

Eine Metabetrachtung führen wir durch, in dem wir ausserhalb des Systems denken, am Besten mit einem Bleistift und einem Blatt Papier. Erst nach der Definition des Algorithmus können wir diesen in einem Programm am Computer umsetzen (Programmieren).

Ein Programm sollte sein:

1. Fehlerfrei
2. Lesbar
3. Kurz
4. Simpel
5. (Effizient)

Laufzeiten lassen sich in verschiedene Komplexitätsklassen (vgl. Landau-Symbole) einteilen:

$$O(c) = \text{konstant} \quad (1.2)$$

$$O(n) = \text{linear} \quad (1.3)$$

$$O(n^2) = \text{quadratisch} \quad (1.4)$$

$$O(2^n) = \text{exponentiell} \quad (1.5)$$

$$O(n \log(n)) = \text{logarithmisch} \quad (1.6)$$

1.5 Darstellung von Werten

Es gibt verschiedene Möglichkeiten, Werte zu repräsentieren, so zum Beispiel:

MDCVI \rightarrow römisch 1606

1.5.1 Dezimales Zahlensystem

Das dezimale Zahlensystem ist ein System mit der Basis 10 und ist ein so genanntes positionelles System, d.h. die Position der Ziffer beeinflusst die repräsentierte Zahl.

Wir geben jeder Position einen Namen. Die Zahl 13408 besteht aus den folgenden Ziffern:

Ziffer	Position	Gewichtung	Exponent	
1	Zehntausender	10000	10^4	(1.7)
3	Tausender	1000	10^3	
4	Hunderter	100	10^2	
0	Zehner	10	10^1	
8	Einer	1	10^0	

Wir sehen, dass die Zahl 13408 eigentlich die Summe aller Ziffern multipliziert mit ihrer Gewichtung ist. Konkret rechnen wir:

$$1 \cdot 10000 + 3 \cdot 1000 + 4 \cdot 100 + 0 \cdot 10 + 8 \cdot 1 \quad (1.8)$$

oder mit Exponentenschreibweise:

$$1 \cdot 10^4 + 3 \cdot 10^3 + 4 \cdot 10^2 + 0 \cdot 10^1 + 8 \cdot 10^0 \quad (1.9)$$

Der Exponent zeigt dabei die Position an (Position 0 bis Position 4). Allgemein formuliert:

$$\begin{aligned} & a_n \cdot a_{n-1} \cdot a_{n-2} \cdot \dots \cdot a_2 \cdot a_1 \cdot a_0 \\ = & a_n \cdot 10^n + a_{n-1} \cdot 10^{n-1} + \dots + a_1 \cdot 10^1 + a_0 \cdot 10^0 \\ = & \sum_{i=0}^n a_i \cdot 10^i \quad \text{wobei} \quad a_i \in \{0, 1, 2, 3, \dots, 8, 9\} \end{aligned} \quad (1.10)$$

Der Exponent i entspricht der Position im positionellen System, wird auch als Index bezeichnet. Weiter halten wir fest:

a_n ist der Koeffizient

10 ist die Basis

n ist der Index

Die einzelnen Ziffern der Ziffernmenge werden als Digits bezeichnet.

$$\text{Ziffernmenge} \quad a_i \in \{0, 1, 2, 3, \dots, 8, 9\} \quad (1.11)$$

1.5.2 Binäres Zahlensystem

Im Gegensatz zum dezimalen Zahlensystem mit Basis 10 ändern wir beim binären Zahlensystem die Basis zu 2.

$$\begin{aligned}\text{Basis :} & \quad 2 \\ \text{Ziffernmenge :} & \quad a_i \in \{0, 1\}\end{aligned}\tag{1.12}$$

Die einzelnen Ziffern der Ziffernmenge werden als Bits bezeichnet.

$$\begin{aligned}& a_n \cdot a_{n-1} \cdot a_{n-2} \cdot \dots \cdot a_2 \cdot a_1 \cdot a_0 \\ = & a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \dots + a_1 \cdot 2^1 + a_0 \cdot 2^0 \\ = & \sum_{i=0}^n a_i \cdot 2^i \quad \text{wobei} \quad a_i \in \{0, 1\}\end{aligned}\tag{1.13}$$

Beispiel 3.

$$\begin{aligned}1101_2 &= 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 8 + 4 + 0 + 1 \\ &= 13\end{aligned}\tag{1.14}$$

Zusätzlich definieren wir:

8 Bit \rightarrow 1 Byte

4 Bit \rightarrow 1 Nibble

1.5.3 Hexadezimal System

$$\begin{aligned}\text{Basis :} & \quad 16 \\ \text{Ziffernmenge :} & \quad a_i \in \{0, 1, 2, \dots, 8, 9, A, B, C, D, E, F\}\end{aligned}\tag{1.15}$$

$$\begin{aligned}& a_n \cdot a_{n-1} \cdot a_{n-2} \cdot \dots \cdot a_2 \cdot a_1 \cdot a_0 \\ = & a_n \cdot 16^n + a_{n-1} \cdot 16^{n-1} + \dots + a_1 \cdot 16^1 + a_0 \cdot 16^0 \\ = & \sum_{i=0}^n a_i \cdot 16^i \quad \text{wobei} \quad a_i \in \{0, 1, 2, \dots, D, E, F\}\end{aligned}\tag{1.16}$$

Beispiel 4.

$$\begin{aligned}A5_{16} = 0xA5 &= 10 \cdot 16^1 + 5 \cdot 16^0 \\ &= 165\end{aligned}\tag{1.17}$$

1.5.4 Umwandlung Hexadezimal und Binär

Binäre Werte und hexadezimale Werte formen wir direkt um, analog folgender Tabelle:

Dezimal	Binär	Hexadezimal
0	0000	0x00
1	0001	0x01
2	0010	0x02
3	0011	0x03
4	0100	0x04
5	0101	0x05
6	0110	0x06
7	0111	0x07
8	1000	0x08
9	1001	0x09
10	1010	0x0A
11	1011	0x0B
12	1100	0x0C
13	1101	0x0D
14	1110	0x0E
15	1111	0x0F

(1.18)

1.5.5 Umwandlung Basis n zu Basis 10

Um eine Zahl mit einer beliebigen Basis n umzurechnen, nutzen wir die Eigenschaften von positionellen Zahlensystemen.

$$\begin{aligned}
 \text{Basis :} & \quad n \\
 \text{Ziffernmenge :} & \quad a_i \in \mathbb{Z}_n
 \end{aligned}
 \tag{1.19}$$

$$\begin{aligned}
 & a_m \cdot a_{m-1} \cdot a_{m-2} \cdot \dots \cdot a_2 \cdot a_1 \cdot a_0 \\
 = & a_m \cdot n^m + a_{m-1} \cdot n^{m-1} + \dots + a_1 \cdot n^1 + a_0 \cdot n^0 \\
 = & \sum_{i=0}^m a_i \cdot n^i \quad \text{wobei} \quad a_i \in \mathbb{Z}_n
 \end{aligned}
 \tag{1.20}$$

1.5.6 Umwandlung Basis 10 zu Basis n

Um eine Zahl von Basis 10 zu Basis N umzuwandeln, nutzen wir die Divisionsmethode (auch Modulo-Methode genannt). Dabei wird die umzuwandelnde Zahl im Dezimalsystem kontinuierlich mit der Basis dividiert. Wir zeigen dies anhand des folgenden Beispiels.

Beispiel 5. *Wir wollen die Zahl 33_{10} als eine Zahl mit Basis 3 schreiben.*

$$\begin{array}{rcll} 33 : 3 & = & 11 & \text{Rest } 0 \\ 11 : 3 & = & 3 & \text{Rest } 2 \\ 3 : 3 & = & 1 & \text{Rest } 0 \\ 1 : 3 & = & 0 & \text{Rest } 1 \end{array} \quad (1.21)$$

Wir hören auf, sobald das Resultat 0 ergibt. Das Resultat lesen wir aus den Modulo-Werten von unten nach oben (in diesem Beispiel $33_{10} = 1020_3$).

Wir dividieren die umzuwandelnde Zahl durch die zu erreichende Basis und schreiben den Rest in eine separate Spalte. Anschliessend wiederholen diese Schritte, bis wir als Resultat 0 erhalten.

1.5.7 Umwandlung von Basis m zu Basis n

Wir berechnen immer zuerst den Wert im Dezimalsystem und rechnen anschliessend mit der Divisionsmethode in die neue Basis um.

Basis n \rightarrow Basis 10 \rightarrow Basis m

1.6 Darstellung von negativen Werten

Im dezimalen System können wir eine Zahl mit dem Zeichen '-' negativ ausdrücken. Im binären System haben wir diese Möglichkeit nicht. Bis jetzt können wir nur die natürlichen Zahlen (\mathbb{N}) darstellen.

1.6.1 Signed magnitude representation

Wir stellen das Vorzeichen mit dem höchsten Bit (auch MSB, "most significant bit" genannt) dar und definieren:

$$a_n \cdot a_{n-1} \cdot a_{n-2} \cdot \dots \cdot a_1 \cdot a_0 \quad (1.22)$$

wobei

$$\begin{aligned} a_n &= \text{Vorzeichen} \\ a_{n-1} \cdot a_{n-2} \cdot \dots \cdot a_1 \cdot a_0 &= \text{Wert} \end{aligned} \quad (1.23)$$

Dies führt uns allerdings zum Problem, dass einige Zahlen keine Eindeutige Zuordnung haben ($1000_2 = 0000_2 = 0$).

1.6.2 Signed representation

Ein Kilometerzähler mit 4 Stellen kann Zahlen von 0 bis 9999 darstellen, bei einem Überlauf (overflow) muss also sichergestellt werden, dass keine Information verloren geht. Um zu überprüfen, ob wir eine Zahl mit n Stellen darstellen können, brauchen wir den Modulo-Operator.

Definition 2. Wir definieren den Modulo wie folgt:

$$x \bmod y := \text{Rest von } \frac{x}{y} \quad (1.24)$$

Wir stellen dabei fest, dass der Modulo einer Zahl immer $< y$ ist.

Beispiel 6.

$$\begin{aligned} 5 \bmod 3 &= 2 \\ 4 \bmod 3 &= 1 \\ 2 \bmod 3 &= 2 \\ 1 \bmod 3 &= 1 \\ 0 \bmod 3 &= 0 \end{aligned} \quad (1.25)$$

Für das folgende Kapitel benötigen wir ausserdem die binäre Addition, welche wiederum den Modulo-Operator benötigt:

Definition 3.

$$a \oplus b := (a + b) \bmod 2^n \quad (1.26)$$

Wenn wir die Werte der Beispiele anschauen, bemerken wir, dass wir mit dem Zweier-Komplement negative Zahlen darstellen können, denn wenn wir den Algorithmus des Zweier-Komplements anwenden, so ergibt sich aus einer positiven Zahl deren negativer Wert und umgekehrt. Wir sehen dies auch anhand folgender Tabelle:

Unsigned representation		Signed representation	
	0	0000	0
	1	0001	1
	2	0010	2
	3	0011	3
	4	0100	4
	5	0101	5
	6	0110	6
	7	0111	7
	8	1000	-8
	9	1001	-7
	10	1010	-6
	11	1011	-5
	12	1100	-4
	13	1101	-3
	14	1110	-2
	15	1111	-1

(1.27)

Den Schritt von 7 zu -8 nennen wir overflow. Bei der unsigned representation nennen wir den Übergang von 1111_2 zu 0000_2 (Addition +1) carry, da bei dieser Addition das Carry-Bit im Prozessor gesetzt wird.

1.6.3 Vergleich signed, unsigned

Wir sehen uns den Wertebereich der verschiedenen Darstellungen an, zuerst allgemein:

	signed representation	unsigned representation
Maximum	$2^{n-1} - 1$	$2^n - 1$
Minimum	$-(2^{n-1})$	0

(1.28)

Bei 4 Bits, bzw. bei 8 Bits:

4 Bits	signed representation	unsigned representation
Maximum	7	15
Minimum	-8	0

(1.29)

8 Bits	signed representation	unsigned representation
Maximum	127	255
Minimum	-128	0

(1.30)

1.7 Binäre Operationen

1.7.1 Das Einer-Komplement K1 (Inversion)

Das Einer-Komplement beschreibt die Inversion, also die Umkehrung der einzelnen Bits in einem binären Wert.

Definition 4. $a_1 \in \{0, 1\}$

$$\begin{aligned} & K1(a_n, \cdot a_{n-1}, \cdot a_{n-2} \cdot \dots \cdot a_2 \cdot a_1 \cdot a_0) \\ & := \neg a_n, \cdot \neg a_{n-1}, \cdot \neg a_{n-2} \cdot \dots \cdot \neg a_2 \cdot \neg a_1 \cdot \neg a_0 \end{aligned} \quad (1.31)$$

Beispiel 7. $K1(1100) = 0011$

1.7.2 Das Zweier-Komplement K2 (Negation)

Wir brauchen eine Methode, wie wir eine binäre Zahl darstellen können und diese einfach in eine negative Zahl konvertieren können. Dies machen wir mit dem Zweier-Komplement.

Definition 5. $a_1 \in \{0, 1\}$

$$\begin{aligned} & K2(a_n, \cdot a_{n-1}, \cdot a_{n-2} \cdot \dots \cdot a_2 \cdot a_1 \cdot a_0) \\ & = K1(a_n, \cdot a_{n-1}, \cdot a_{n-2} \cdot \dots \cdot a_2 \cdot a_1 \cdot a_0) \oplus 1 \end{aligned} \quad (1.32)$$

Beispiel 8.

$$\begin{aligned} K2(0110) &= K1(0110) \oplus 1 \\ &= 1001 \oplus 1 = (1001 + 0001) \mod 2^4 \\ &= 1010_2 = 10_{10} \end{aligned} \quad (1.33)$$

$$\begin{aligned} K2(0000) &= K1(0000) \oplus 1 \\ &= 1111 \oplus 1 = (1111 + 0001) \mod 2^4 \\ &= 10000 \mod 2^4 = 0000_2 \end{aligned} \quad (1.34)$$

$$\begin{aligned} K2(1111) &= K1(1111) \oplus 1 \\ &= 0000 \oplus 1 = (0000 + 0001) \mod 2^4 \\ &= 10001_2 = 1_{10} \end{aligned} \quad (1.35)$$

$$\begin{aligned} K2(0001) &= K1(0001) \oplus 1 \\ &= 1110 \oplus 1 = (1110 + 0001) \mod 2^4 \\ &= 1111_2 = 15_{10} \end{aligned} \quad (1.36)$$

1.7.3 Addition

Die binäre Addition beruht auf der folgenden Wahrheitstabelle, wenn wir zwei Bits a_n und b_n addieren:

a_n	b_n	Summe	c_{n+1}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Das Carry-Bit (c_n) wird dann zur nächsten Bit-Addition "mitgenommen".

Beispiel 9. Wir zeigen die Addition an den folgenden Beispielen:

$$1. \ 6_{10} + 10_{10} = 16_{10}$$

	0	0	0	1	1	0
+	0	0 ₁	1 ₁	0 ₁	1	0
	0	1	0	0	0	0

1.7.4 Subtraktion

Definition 6. $a - b := a \oplus K2(b)$

	<u>unsigned</u>		<u>computer code</u>		<u>signed</u>
			1000		-4
Beispiel 10.	$\frac{12,15}{27}$	←	<u>+1111</u>	$\xrightarrow{\text{overflow}}$	<u>+(-1)</u>
			0111		-5
	≠ 11	←	<u>0111</u>	→	-5

1.8 Flags

1.8.1 Carry- und Overflowflag

Um zu wissen, ob das angezeigte Resultat aus der vorhergegangenen Rechnung korrekt war, setzt der Prozessor sogenannte Flags. Hier soll nun das Carry-Flag und das Overflow-Flag besprochen werden.

Das Carry-Flag wird gesetzt, wenn das Resultat einer unsigned Rechnung nicht stimmt, d.h. wenn das korrekte Resultat grösser der maximal darstellbaren Zahl ist ($2^n - 1$, wobei n Anzahl der Bits ist) oder unter 0 fällt.

Das Overflow-Flag wird gesetzt, wenn bei einer arithmetischen Operation mit signed-Zahlen das Resultat entweder kleiner der minimal darstellbaren Zahl ($< -(2^{n-1})$) ist oder grösser der maximal darstellbaren Zahl ($> 2^{n-1} - 1$) ist.

Allgemein stellen wir eine Addition wie folgt dar:

$$\begin{array}{ccccccc}
 & a_n & a_{n-1} & \dots & a_2 & a_1 & a_0 \\
 + & b_n & b_{n-1} & \dots & b_2 & b_1 & b_0 \\
 \hline
 c_{n+1} & c_n & c_{n-1} & \dots & c_2 & c_1 & c_0
 \end{array}$$

Nun finden wir für das Carry-Flag, bzw. das Overflow-Flag folgende Regeln:

	Addition	Subtraktion
Carry-Flag	C_{n+1}	$\neg C_{n+1}$
Overflow-Flag	$(a_n \wedge b_n \wedge \neg c_n) \vee (\neg a \wedge \neg b_n \wedge c_n)$	

Es ist wichtig zu verstehen, dass der Prozessor keinen Unterschied von unsigned und signed representation macht. Die Flags werden bei jeder Berechnung des Prozessors gesetzt. Das ausgeführte Programm muss dann die Flags beachten und entsprechend handhaben.

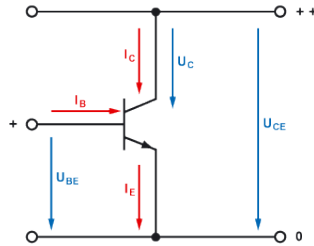
Chapter 2

Digitaltechnik

Das Moorsche Gesetz besagt, dass sich die Komplexität (= Anzahl Transistoren) integrierter Schaltkreise regelmässig verdoppelt. Je nach Quelle werden 18 oder 24 Monate als Zeitraum genannt.

2.1 Transistor

Ein Transistor ist ein elektronisches Bauelement, bei welchem mittels einem Steuerstrom ein grösserer Strom gesteuert werden kann. Ein Transistor wird in der Elektronik mit folgendem Symbol dargestellt (Ströme und Spannungen eingezeichnet):



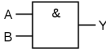


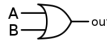

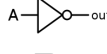

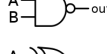

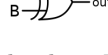
Grundlegend funktioniert der Transistor folgendermassen:

Ohne angelegte Spannung ist die Strecke Kollektor-Emitter (oben nach unten) elektrisch isolierend (Widerstand $R_{CE} = \infty\Omega$), d.h. es kann kein Strom fliessen ($I_C = 0$).

Wird nun an der Basis (links) eine Spannung von (ungefähr) $U_{BE} = 0.7V$ angelegt, so fliesst ein Basisstrom (I_B) und die Kollektor-Emitter-Strecke wird elektrisch leitend ($R_{CE} < \infty\Omega$). Somit lässt sich über einen kleinen Basisstrom ein grosser Strom steuern.

2.2 Gatter

Unter einem Gatter verstehen wir in der Digitaltechnik ein Element, dass eine bestimmte logische Operation durchführen kann. Diese Elemente können alle mit Transistoren realisiert werden. Die Operationen kennen wir aus der Logik, die Symbole entnehmen wir der folgenden Tabelle:

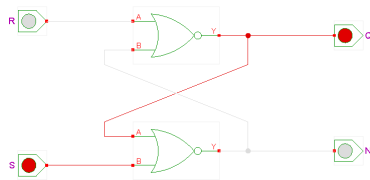
Funktion	IEC-Symbol	ANSI-Symbol	Boolsche Formel
AND			$Y = A \wedge B$
OR			$Y = A \vee B$
NOT			$Y = \neg A$
NAND			$Y = \neg(A \wedge B)$
XOR			$Y = A \underline{\vee} B$

Für jedes Gatter kennen wir drei verschiedene Repräsentationen:

- Schaltbild
- Boolsche Formel
- Wahrheitstabelle

2.3 RS-Flipflop

Ein RS-Flipflop (auch Latch genannt) ist eine logische Schaltung, mit welcher sich 1 Bit speichern lässt. Der Teil "RS" des Namens bezieht sich auf die Eingänge S ("Set") und R ("Reset"). Das die folgende Schaltung zeigt den Aufbau eines RS-Flipflops mit NOR-Gattern:



Wie oben bereits definiert, speichert ein RS-Flipflop ein Bit, dabei gibt es gemäss der Logik bei zwei Eingängen vier verschiedene Zustände des Flipflops:

S	R	Q	$\neg Q$
0	0	Q^*	$\neg Q^*$
0	1	0	1
1	0	1	0
1	1	-	-

Wenn wir das RS-Flipflop genauer untersuchen, so erkennen wir vier verschiedene Zustände:

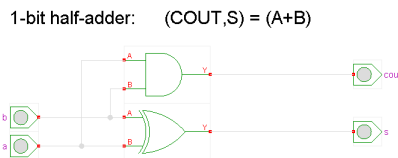
- $S = 0$ und $R = 0$: In diesem Zustand zeigen die Ausgänge den gespeicherten Wert des Flipflops an, es gibt keine Veränderung der Ausgangswerte.
- $S = 1$ und $R = 0$: In diesem Zustand wird das Flipflop, bzw. der Ausgang Q des Flipflops auf 1 gesetzt. Der Ausgang $\neg Q$ entspricht auf 0.
- $S = 0$ und $R = 1$: In diesem Zustand wird das Flipflop auf 0 gesetzt. Das bedeutet, dass der Ausgang Q auf 0 gesetzt wird.
- $S = 1$ und $R = 1$: Illegal (nicht definierter) Zustand, da es hier eine Race-Condition gibt.

RS-Flipflops werden beispielsweise bei statischen RAM (SRAM) eingesetzt, da sie sehr schnell sind. Bei einer CPU werden solche Latches beispielsweise für den internen Cache oder die Register eingesetzt.

2.4 Arithmetische Schaltungen

2.4.1 Halbaddierer

Ein Halbaddierer ist eine einfache logische Schaltung, mit welcher sich zwei Bits addieren lassen. Neben dem Resultat wird das Carry-Bit ausgegeben. Das Schaltbild für den Halbaddierer sieht folgendermassen aus:



Für die zwei Eingänge und zwei Ausgänge erstellen wir eine Wahrheitswertta-
belle:

a_n	b_n	Summe	c_{n+1}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Aus dieser Wahrheitswerttabelle können wir folgende boolsche Formeln her-
leiten:

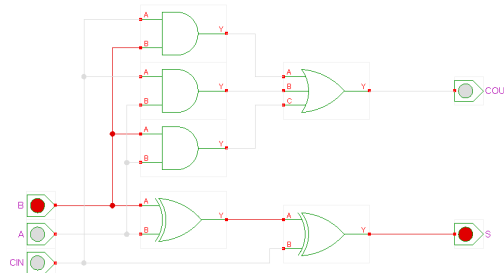
$$S = (a_n \wedge \neg b_n) \vee (\neg a_n \wedge b_n) = a \vee b \quad (2.1)$$

$$COUT = a_n \wedge b_n \quad (2.2)$$

2.4.2 Volladdierer

Grundsätzlich handelt es sich beim Volladdierer um einen Halbaddierer, welcher das Carry-Bit der vorhergegangenen Addition ebenfalls in die Rechnung mit einbezieht. Statt zwei Eingänge haben wir nun drei Eingänge (nämlich a , b und das Carry-Bit c).

1-bit full-adder: $(COUT, S) = (A+B+Cin)$



Wie sehen dank dieser Implementation sofort, dass der Summen-Ausgang (S) nur gesetzt wird, wenn lediglich einer der Eingänge auf 1 gesetzt ist. Das Carry-Bit wird wiederum gesetzt, sobald mindestens zwei Eingänge auf 1 gesetzt sind.

Die Schaltung lässt sich zu folgenden booleschen Formeln umwandeln:

$$S = (A \vee B) \wedge \neg CIN \quad (2.3)$$

$$COUT = (A \wedge B) \vee (B \wedge CIN) \vee (A \wedge CIN) \quad (2.4)$$

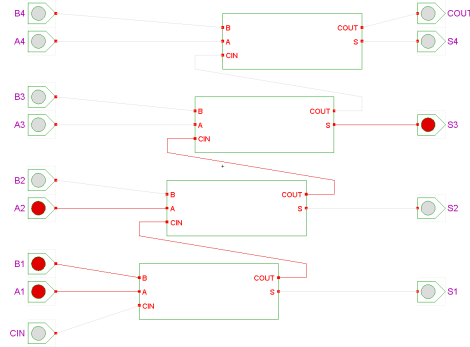
Daraus können wir nun die Wahrheitswerttabelle für den Volladdierer erstellen:

a_n	b_n	c_n	Summe	c_{n+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

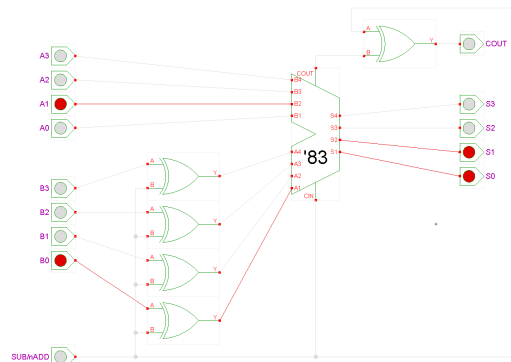
Bei einer ALU (Arithmetic Logic Unit) werden mehrere Volladdierer hintereinander geschaltet, um zwei binäre Zahlen zusammenzurechnen.

2.4.3 Implementation Einfache ALU

Um einen 4-Bit Addierer/Subtrahierer zu realisieren, kaskadieren wir vier Volladdierer:



Diese Schaltung abstrahieren wir dann zu einem ALU-Baustein, welcher im folgenden Schema als "83" dargestellt ist:



Analysieren wir die Schaltung, so erkennen wir die zwei Betriebsmodi:

- Addition, wenn das "SUB/nADD"-Bit = 0 ist. Dann werden beide 4-Bit-Werte unverändert an die ALU weitergegeben. Das Carry-Flag der gesamten Schaltung (COUT) wird analog dem Carry-Bit der ALU (COUT) gesetzt.
- Subtraktion, wenn das "SUB/nADD"-Bit = 1 ist. Sobald das Bit gesetzt ist, wird die zweite Zahl (B_n) invertiert und das CIN-Bit der ALU auf 1 gesetzt. Zusätzlich wird das COUT-Flag invertiert gesetzt, d.h. wenn ein Carry-Bit gesetzt ist, so wird das Carry-Flag nicht gesetzt und umgekehrt.

Diese ALU lässt sich um beliebig viele Bits erweitern. Dazu werden intern lediglich zusätzliche Volladdierer kaskadiert.

2.5 Memory (Speicher)

2.5.1 Multiplexer

Ein Multiplexer dient dazu, von einer beliebigen Anzahl Eingänge genau einen an den Ausgang weiter zu schalten. D.h. mittels der Adresseingänge können wir bestimmen, welcher Eingang an den Ausgang weitergegeben werden kann.

2.5.2 Adress-Decoder

Mit einem Adress-Decoder kann eine bestimmte Leitung am Adress-Decoder aktiv geschaltet werden. Über die Adresseingänge ($A_1 \dots A_n$) können wir bestimmen, welcher Ausgang (bei 4 Adressbits $2^4 = 16$ Ausgänge) auf aktiv geschaltet wird.

Im Folgenden die Wahrheitswerttabelle für einen 2:4 Adress-Decoder:

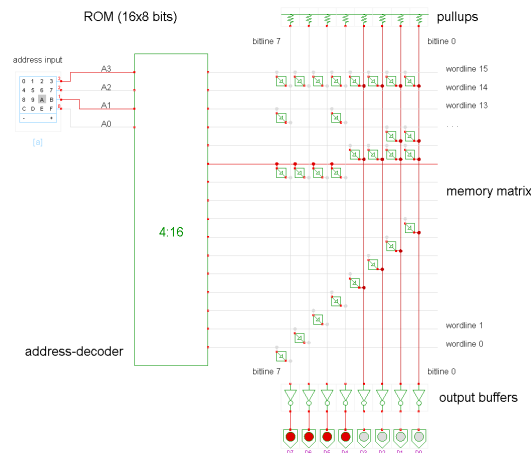
A_1	A_0	O_3	O_2	O_1	O_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Adress-Decoder setzen wir beispielsweise bei Speicherbausteinen ein, damit wir eine Zeile oder Spalte des Speichers auswählen können.

2.5.3 Read Only Memory (ROM)

Ein ROM ist ein "nur-lesbarer" Speicher. Dies bedeutet, dass wir von Aussen lediglich bestimmen können, welche Stelle des Speichers wir abfragen wollen.

Folgende Schaltung zeigt ein ROM, welches 16 8-Bit-Werte speichern kann (also 16 Byte).



Um die Funktion zu verstehen, analysieren wir die Elemente des ROMs:

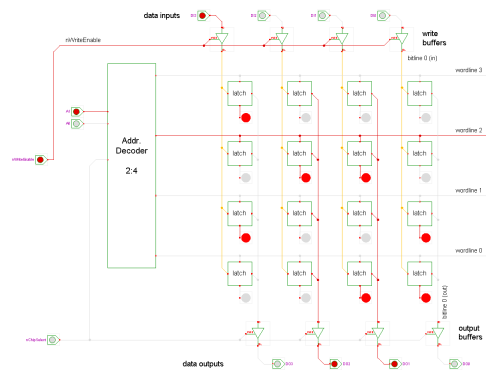
- Der Adress-Decoder decodiert die Adresse, welche an den Eingängen A_0 bis A_3 angelegt ist und setzt die entsprechende Zeile der Speicher-Matrix auf 1.
- Die Speicher-Matrix besteht aus 16 Zeilen mit jeweils 8 Stellen. Eine Stelle kann entweder leer sein (keine Sicherung, "fuse") oder mit einer Sicherung belegt sein. Wenn an einer Stelle eine Sicherung ist, wird die dazugehörige Spalte auf 0 gesetzt sobald die Zeile auf 1 gesetzt wird. Intern wird dies durch einen Transistor realisiert, welcher die Spalte auf Null-Potential zieht.
- Die Ausgabe-Gatter invertieren nun den Wert der Zeilen und geben diese am Ausgang des ROMs aus.

Die Sicherungen werden bei der Herstellung des ROMs gesetzt und können nicht nachträglich geändert werden. Solche ROMs werden heute nur noch selten eingesetzt und wurden durch PROM (Programmable ROM), EPROM (Erasable Programmable ROM) oder EEPROM (Electrically Erasable Programmable ROM) ersetzt.

2.5.4 Random Access Memory (RAM)

Generell können wir bei RAM zwischen zwei verschiedenen RAM-Arten unterscheiden:

- Statisches RAM (SRAM): Statisches RAM wird mit Transistoren (Latches) realisiert. Dies bedeutet, statisches RAM ist sehr schnell, allerdings auch relativ teuer und benötigt im Gegensatz zum dynamischen RAM auch relativ viel Platz.
- Dynamisches RAM (DRAM): Dieser Typ von RAM ist mit Kondensatoren und wenigen Transistoren aufgebaut. Die Information wird als elektrische Ladung im Kondensator gespeichert. Jeder Kondensator speichert ein Bit. Vorteilig bei diesem RAM-Typ sind die tiefen Produktionskosten und die grosse Dichte die erreicht werden kann. Allerdings ist das dynamische RAM im Vergleich zum statischen RAM relativ langsam und erfordert wegen der Leckströme einen regelmässigen Refresh der Speicherzellen. Bei modernen DRAM ist diese Refresh-Funktionalität allerdings im DRAM eingebaut.



2.6 CPU

2.6.1 Architektur

Von Neumann

2.6.2 Blockschaltbild

2.6.3 Ausführungszyklus

Übersicht

In jedem Ausführungszyklus führt die CPU verschiedene Schritte aus, um einen Maschinenbefehl auszuführen.

Fetch

Adress

Execute

Chapter 3

Assembler

3.1 Grundlagen, Syntax

Assembler (auch Maschinensprache genannt) ist die direkteste Programmiersprache für Programme. Programme, wie in Assembler geschrieben sind, werden vom Assembler direkt in so genannte OP-Codes (Befehlswerte) übersetzt, welche dann direkt von der CPU verarbeitet werden.

Durch den direkten Zugriff auf die Hardware ist Assembler sehr schnell (im Vergleich zu Hochsprachen) und generiert sehr kleinen Code.

3.2 Adressierungsarten

3.2.1 Implicit

3.2.2 Immediate

3.2.3 Absolute

3.2.4 Indirect

3.2.5 Zero Page

3.2.6 Relative

Im Zweier-Komplement

3.3 Conditional branching

3.4 Subroutines

3.4.1 Calling Conventions

Chapter 4

C