

Grundlagen der Informatik

Martin Schmidli

30. Januar 2014

Inhaltsverzeichnis

1	Einführung	4
1.1	Lizenzen und Patente	4
1.2	Syntax und Semantik	4
1.3	Analog vs Digital	4
1.4	Daten und Informationen	5
1.5	Definition Computer	5
1.6	Programmiersprachen & Algorithmen	5
2	Zahlensysteme	6
2.1	Dezimalsystem	6
2.1.1	Umwandlung Basis n zu Basis 10	7
2.1.2	Umwandlung Basis 10 zu Basis n	7
2.1.3	Umwandlung von Basis m zu Basis n	7
2.2	Hexadezimals System	8
2.3	Binärsystem	8
2.4	Umwandlung Hexadezimal und Binär	9
2.5	Negative Zahlen im Binärsystem	9
2.6	Speichereinheiten	10
2.7	Binäraddition	11
2.7.1	Flags	11
2.8	Zeichenspeicherung	13
3	Digitaltechnik	14
3.1	Transistoren	14
3.2	Logische Gatter	15
3.3	only-NAND	15
3.4	RS-Flipflop	16
3.5	Multiplexer	16
3.6	Decoder	17
3.7	Arithmetische Schaltungen	17
3.7.1	Halbaddierer	17
3.7.2	Volladdierer	18
3.7.3	n-Bit Addierer	18
3.8	Read-only memory	19
3.9	Random-access memory	20
3.10	Speichergrösse	21

4	Computerarchitektur	23
4.1	Von-Neumann Architektur	23
4.2	CPU	23
4.2.1	ALU (Arithmetic Logic Unit)	24
4.2.2	CU (Control Unit)	24
4.2.3	Clock	24
4.3	Register	24
4.3.1	Instruction Pointer (IP)	24
4.3.2	Adress-Register (AR)	24
4.3.3	Instruktionsregister (IR)	24
4.3.4	Accumulator / Prozessor Register	25
4.4	Memory (Programm-/Datenspeicher)	25
4.5	Bussystem	25
4.6	Instruction Set	25
4.7	Instruction Execution Cycle	25
5	6502 Assemblersprache	28
5.1	Die 6502 Architektur	28
5.1.1	Register des 6502	28
5.2	Logische / Arithmetische Operationen	29
5.3	Memory Transfers	29
5.4	Addressing Modes	30
5.4.1	Accumulator	30
5.4.2	Immediate	30
5.4.3	Absolute Addressing	30
5.4.4	Relative	31
5.4.5	Zero-Page	32
5.4.6	Indirect	32
5.4.7	Absolute Indexed Addressing	32
5.4.8	Indexed Zero Page Addressing	32
5.4.9	Indexed Indirect Addressing (Indirect,X)	32
5.4.10	Indirect Indexed Addressing (Indirect),Y	33
5.5	Conditional Jumps	33
5.5.1	Conditional Jump x86 Spezifisch	33
5.6	Stack	34
5.6.1	PHA: Push Akku Value on Stack	34
5.6.2	PHP: Push Programcounter Value on Stack	35
5.6.3	PLA: Pop Data from Stack, store it in Akku	36
5.6.4	Endian	36
5.7	Subroutines	37
5.7.1	JSR	37
5.7.2	RTS	37
6	Die Programmiersprache C	38
6.1	Variablen	38
6.2	Pointers	39
6.2.1	Einfaches Dereferenzieren	39
6.2.2	Doppeltes Dereferenzieren	41
6.3	Nutzen von Pointern	42

6.4	Arrays	42
6.5	Dynamische Speicherallozierung/ Dynamic Memory Allocation	43
6.6	Speichersegmente	45
6.7	Listen	46
6.8	Operator Precedence	48
6.8.1	Beispiele	49
7	Tables	50

Kapitel 1

Einführung

1.1 Lizenzen und Patente

Unter einer Lizenz versteht man ein Gebrauchsrecht. Wenn Software gekauft wird, so wird ein Gebrauchsrecht für diese Software erworben. Es gibt verschiedene Arten von Lizenzen, z.B:

1. Open Source Lizenzen, z.B. GNU GPL
2. Closed Source Lizenzen, z.B. EULA (End User License Agreement)

Während die Closed Source Lizenzen in der Regel Verbote enthalten (Kopieren verboten, abändern verboten etc.), machen die Open Source Lizenzen auf die Rechte aufmerksam (Kopieren, abändern, verteilen erlaubt etc.).

Beide Arten von Lizenzen verfolgen unterschiedliche Businessmodelle. Bei Closed Source Lizenzen will man die Kunden mittels proprietären Formaten längerfristig an ihre Produkte binden. Das Closed Source Businessmodell ermöglicht einigen wenigen Anbietern einen grossen Reichtum zu erlangen. Bei Open Source kann sich hingegen nur der Beste behaupten.

1.2 Syntax und Semantik

Die Syntax steht für die Struktur, die Semantik für die Funktion. Die Struktur eines Buches beispielsweise, besteht aus dem Inhaltsverzeichnis, dem Prolog, den Kapiteln, einer bestimmten Sprache etc. Die Funktion ist die Informationsübermittlung. Bei einem chinesischen Buch erkennt man zwar die Struktur, allerdings kann man den Inhalt nicht verstehen.

Eine sogenannte Strukturanalyse in einem Code ist daher immer einfach - eine Funktionsanalyse hingegen benötigt sehr viel Zeit.

1.3 Analog vs Digital

Analog bedeutet soviel wie ein zustandsloses (stufenloses) System. Im Gegensatz dazu besitzt ein digitales Signal Zustände/Stufen. Beispielsweise gibt es zwischen den Zuständen 0 und 1 keine weitere Abgrenzung, während es in der analogen Welt unendlich viele Zwischenstufen gibt, z.B. 0.1, 0.00123 etc.

1.4 Daten und Informationen

Im Gegensatz zu Daten benötigen Informationen einen Kontext. Reine Daten ohne Kontext können kaum interpretiert werden.

1.5 Definition Computer

Ein Computer ist ein abstrakter Begriff. Die Implementation kann auf verschiedene Arten passieren, beispielsweise analog, digital, elektrisch, mechanisch.

Alan Turing (1912 - 1954) definierte einen Computer als

„Maschine zur Manipulation von Zeichen“

1.6 Programmiersprachen & Algorithmen

Dem Computer sind lediglich die Zustände 0 und 1 bekannt. Die Sprache eines Computers - auch Maschinencode - genannt, besteht somit ebenfalls nur aus 0 und 1 und ist für Menschen nicht lesbar. Jeder Prozessor besitzt daher eine eigene Sprache - genannt Assemblersprache - welche bestimmten Binärcodes einen Befehl zuordnet.

Hochsprachen sind abstrahierte Sprachen, welche dem Programmierer viele Probleme abnehmen, beispielsweise die Speicherallokation, Schleifen etc.

Ein Algorithmus beschreibt einen Lösungsweg, ein Programm ist die Implementation eines Algorithmus.

Kapitel 2

Zahlensysteme

Zahlen repräsentieren Werte.

2.1 Dezimalsystem

Das Dezimalsystem besitzt zur Darstellung einer Zahl 10 verschiedene Ziffern; man spricht dabei auch von der Ziffermenge:

$$A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Unser Zahlensystem ist ein sogenanntes Stellenwertsystem - auch bekannt als ein positionelles System - das heisst, die Position einer Ziffer bestimmt deren Wert. Beispielsweise besteht die Zahl 5072 aus den folgenden Ziffern:

Ziffer	Position	Stellenwert	Exponent
5	Tausender	1000	10^3
0	Hunderter	100	10^2
7	Zehner	10	10^1
2	Einer	1	10^0

Wir sehen, dass die Zahl 5072 eigentlich die Summe aller Ziffern multipliziert mit dem Stellenwert ist:

$$5 \cdot 1000 + 0 \cdot 100 + 7 \cdot 10 + 2 \cdot 1$$

oder in Exponentenschreibweise:

$$5 \cdot 10^3 + 0 \cdot 10^2 + 7 \cdot 10^1 + 2 \cdot 10^0$$

Der Exponent zeigt dabei die Position an (Position 0 bis Position 3). Allgemein formuliert:

$$a_n \cdot 10^n + a_{n-1} \cdot 10^{n-1} + \dots + a_1 \cdot 10^1 + a_0 \cdot 10^0$$

oder mit dem Summenzeichen (Sigma):

$$\sum_{i=0}^n a_i \cdot 10^i \quad \text{wobei} \quad a_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

2.1.1 Umwandlung Basis n zu Basis 10

Um eine Zahl einer beliebigen Basis n in das Dezimalsystem umzurechnen, nutzen wir die Eigenschaften von positionellen Zahlensystemen, sprich, wir berechnen die Summe aller Produkte aus der jeweiligen Ziffer mit dem Wert ihrer Stelle (Basis^{index}):

$$\begin{array}{ll} \text{Basis :} & n \\ \text{Ziffernmenge :} & a_i \in \mathbb{Z}_n \end{array}$$

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

Beispiel 1. Wir möchten die Zahl 325_7 ($_7$ kennzeichnet das Zahlensystem der Zahl, in diesem Beispiel das 7er System) in das Dezimalsystem umwandeln:

$$\begin{aligned} 325_7 &= 3 \cdot 7^2 + 2 \cdot 7^1 + 5 \cdot 7^0 \\ &= 3 \cdot 49 + 2 \cdot 7 + 5 \cdot 1 \\ &= 147 + 14 + 5 \\ &= 166 \end{aligned}$$

2.1.2 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 2. Wir wollen die Zahl 33_{10} als eine Zahl mit Basis 3 schreiben.

$$\begin{array}{lll} 33 : 3 &= & 11 \quad \text{Rest } 0 \\ 11 : 3 &= & 3 \quad \text{Rest } 2 \\ 3 : 3 &= & 1 \quad \text{Rest } 0 \\ 1 : 3 &= & 0 \quad \text{Rest } 1 \end{array}$$

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.

2.1.3 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.

$$\text{Basis } n \rightarrow \text{Basis } 10 \rightarrow \text{Basis } m$$

2.2 Hexadezimals System

Das Hexadezimal System besteht aus 16 Ziffern und gehört in der Informatik zusammen mit dem Binärsystem zu den wichtigsten Zahlensystemen.

$$\begin{aligned} \text{Basis :} & \quad 16 \\ \text{Ziffernmenge :} & \quad a_i \in \{0, 1, 2, \dots, 8, 9, A, B, C, D, E, F\} \end{aligned} \quad (2.1)$$

$$\begin{aligned} & a_n a_{n-1} a_{n-2} \dots a_2 a_1 a_0 \\ = & a_n \cdot 16^n + a_{n-1} \cdot 16^{n-1} + \dots + a_1 \cdot 16^1 + a_0 \cdot 16^0 \\ = & \sum_{i=0}^n a_i \cdot 16^i \quad \text{wobei} \quad a_i \in \{0, 1, 2, \dots, D, E, F\} \end{aligned} \quad (2.2)$$

Beispiel 3.

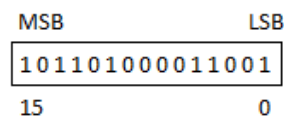
$$\begin{aligned} A5_{16} = 0xA5 &= 10 \cdot 16^1 + 5 \cdot 16^0 \\ &= 165 \end{aligned} \quad (2.3)$$

2.3 Binärsystem

Der Computer speichert Daten als eine Sammlung von elektronischen Ladungen (on oder off, true oder false). Das Binärsystem besitzt somit lediglich zwei Ziffern zur Darstellung dieser zwei Zustände:

$$A = \{0, 1\}$$

Jede Ziffer entspricht damit einem **Bit** als einen Platzhalter für eine 0 oder eine 1. Das erste Bit besitzt den höchsten Wert und wird daher auch als „Most Significant Bit“ bezeichnet, während das letzte Bit das „Least Significant Bit“ genannt wird.



2.4 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

(2.4)

2.5 Negative Zahlen im Binärsystem

Während wir zur Darstellung von negativen Zahlen im Dezimalsystem die Ziffermenge um das Minus-Symbol erweitern können, ist dies im Binärsystem so nicht möglich. Stattdessen kann das MSB einer binären Zahl das Vorzeichen (engl. „Sign“) der Zahl bestimmen: 1 steht für negative, 0 für positive Zahlen. Wir nutzen ausserdem das sogenannte Zweierkomplement um die Zahlen umzuwandeln. Das Zweierkomplement wird aus einer beliebigen binären Zahl berechnet, indem man jede Ziffer invertiert und anschliessend 1 dazu addiert:

Beispiel: Wir möchten die Zahl 0010_2 bzw. 2_{10} in eine negative Binärzahl verwandeln:

$$\begin{aligned}\text{Zweierkomplement}(0010) \\ &= 1101\end{aligned}$$

Nun addieren wir 1 zum Resultat:

$$\begin{aligned}1101 + 1 \\ &= 1110\end{aligned}$$

Wird im Speicher ein Bereich von 4-Bit betrachtet, beispielsweise 1110, so entsprechen diese Daten je nach Interpretation/Kontext einer anderen Information:

Unsigned-Interpretation

$$\begin{aligned}1110 \\ &= 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\ &= 14_{10}\end{aligned}$$

Signed-Interpretation

$$\begin{aligned}
 & \text{Zweierkomplement}(1110) = 0010 \\
 = & 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\
 = & -2
 \end{aligned}$$

Weitere Zahlen:

	Unsigned interpretation		Signed interpretation
	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

Dadurch, dass für das Vorzeichen ein Bit benötigt wird, reduziert sich der Wertebereich (Range) einer Zahl entsprechend:

$$-2^{n-1}, \dots, 0, \dots, 2^{n-1} - 1$$

Der Wertebereich reicht somit für eine positive Zahl weniger als für negative.

2.6 Speichereinheiten

Die Standard Speichereinheit in einem Computer ist ein Byte. Ein Byte enthält 8 Bit. Diese Zahl ist besonders historisch bedingt. Mit der Entwicklung der Computerarchitekturen, wurden immer grössere Einheiten eingeführt:

$$\begin{aligned}
 1 \text{ byte} &= 8 \text{ bit} \\
 1 \text{ word} &= 16 \text{ bit} \\
 1 \text{ doubleword} &= 32 \text{ bit} \\
 1 \text{ quadword} &= 64 \text{ bit}
 \end{aligned}$$

Datentyp	Range	Datentyp	Range
Unsigned byte	0 bis 255	Signed byte	-128 bis +127
Unsigned word	0 bis 65'535	Signed word	-32'768 bis +32'767
Unsigned doubleword	0 bis 4'294'967'295	Signed doubleword	-2'147'483'648 bis +2'147'483'647

2.7 Binäraddition

Wenn zwei binäre Zahlen addiert werden sollen, geschieht dies Bit für Bit, beginnend mit dem LSB. Für die Addition zweier Bits gibt es 4 Möglichkeiten:

$$\begin{array}{c|c} 0 + 0 = 0 & 0 + 1 = 1 \\ \hline 1 + 0 = 1 & 1 + 1 = 10 \end{array}$$

Der Wert von $1 + 1 = 10$ übersteigt offensichtlich den Wertebereich eines Bits und wird Carry (Übertrag) genannt und wird zur Addition der nächsthöheren Stelle mitgenommen.

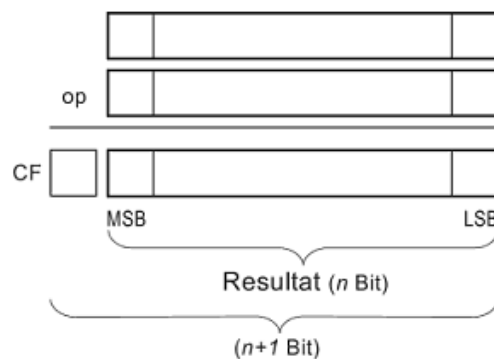
Beispiel: $6 + 10 = 16$

$$\begin{array}{cccccc} & 0 & 0 & 0 & 1 & 1 & 0 \\ + & 0 & 0_1 & 1_1 & 0_1 & 1 & 0 \\ \hline 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{array}$$

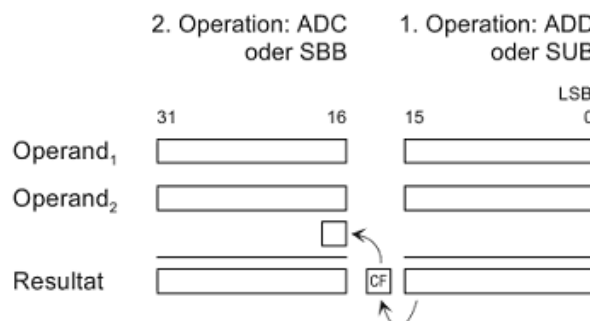
2.7.1 Flags

Um herauszufinden, ob das angezeigte Resultat einer Rechnung stimmt, setzt der Prozessor sogenannte Flags.

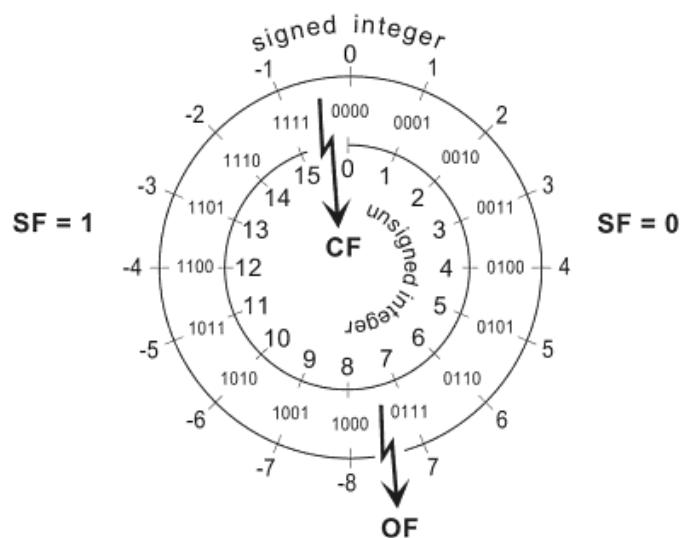
Das **Carry-Flag** wird gesetzt, wenn das Resultat einer unsigned Rechnung nicht stimmt, d.h. wenn das korrekte Resultat grösser der maximal darstellbaren Zahl ist.



Das Carryflag wird auch genutzt, um Additionen grosser Binärzahlen in mehrere Teilschritte aufzuteilen.



Das **Overflow-Flag** wird gesetzt, wenn bei einer arithmetischen Operation mit signed-Zahlen das Resultat entweder kleiner der minimal darstellbaren Zahl ist oder grösser der maximal darstellbaren Zahl.



Allgemein stellen wir eine Addition wie folgt dar:

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

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

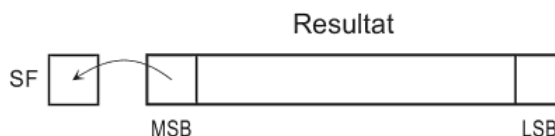
	Addition	Subtraktion
Carry-Flag	C_{n+1}	$\neg C_{n+1}$
Overflow-Flag	$(a_n \wedge b_n \wedge \neg c_n) \vee (\neg a_n \wedge \neg b_n \wedge c_n)$	

Eine einfach zu merkende Regel für das Overflow-Flag:

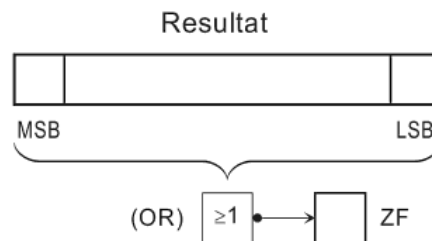
Bei Signed-Zahlen: Das Flag wird gesetzt, wenn aus einer Addition zweier positiver Zahlen eine negative folgt oder umgekehrt.

Bei Unsigned-Zahlen: Das Flag wird gesetzt, wenn das Carry des MSB 1 ist.

Das **Sign-Flag** entspricht dem höchstwertigen Bit des Resultats. Es wird zwar immer berechnet, macht jedoch nur bei vorzeichenbehafteten Operanden einen Sinn.



Das **Zero-Flag** zeigt an, ob das Resultat einer Operation null ist. Da die Null bei Zahlen mit und ohne Vorzeichen gleich dargestellt wird, ist keine Unterscheidung nötig. Das Zero-Flag wird aus der invertierten Oder-Verknüpfung (NOR) aller Bits des Resultates gebildet.



2.8 Zeichenspeicherung

Man mag sich nun fragen, wie z.B. Buchstaben gespeichert werden können. Dazu werden sogenannte Character Sets benutzt, welche ein Zeichen einer Zahl zuordnen. Bis vor wenigen Jahren wurden dafür nur 8 Bit benutzt. Wegen der grossen Vielfalt an Sprachen wurde dann der 16-Bit Unicode Set eingeführt.

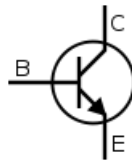
Wenn wir im Speicher nun ein Byte mit dem dezimalen Wert 65 finden, so entspricht dieser in Binärschreibweise 01000001. Ein bestimmtes Programm würde dies nun vermutlich in hexadezimaler Schreibweise als 41 ausgeben. Der Grafikkartentreiber hingegen würde stattdessen den Buchstaben A anzeigen, denn 65 entspricht dem ASCII Code für den Buchstaben A.

Kapitel 3

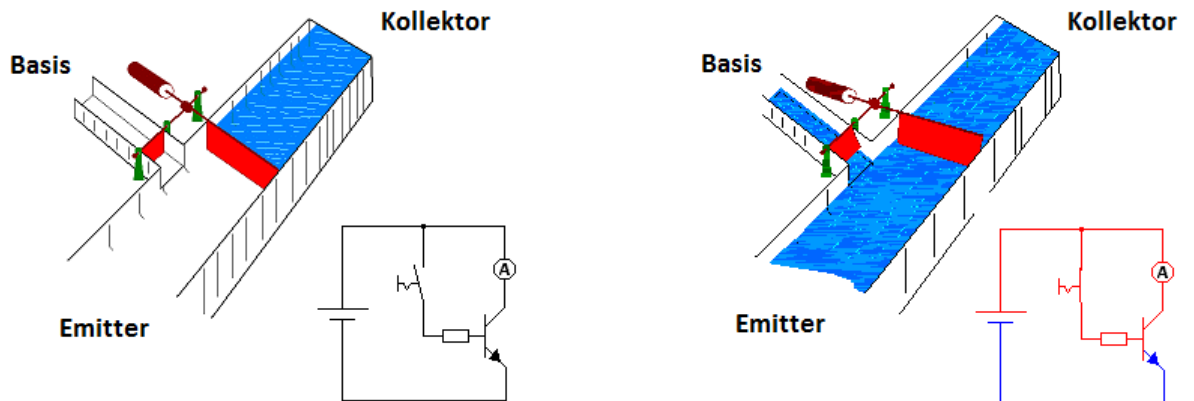
Digitaltechnik

3.1 Transistoren

Ein Transistor (kurz für **transfer resistor**) ist ein nicht lineares Schaltelement (=Schaltung) und ermöglicht es, mit Strom einen weiteren Strom zu steuern, ohne dass dabei mechanische Bewegungen ausgeführt werden müssen.



Durch eine kleine Kontrollspannung wird dabei der Stromfluss de- bzw. aktiviert.



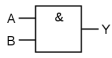
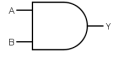

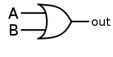

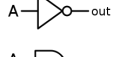



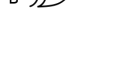
Die Zustände lassen sich mit einer Wahrheitstabelle darstellen:

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

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. Daraus ergibt sich ein schneller technischer Fortschritt. Nach der mechanischen und elektrischen Steuerung von Strömen, fehlt uns heute die Möglichkeit, einen Strom mit Licht steuern zu können. Ein solcher Licht-Transistor würde es ermöglichen, extrem schnelle Supercomputer zu bauen. Die ETH hat in diesem Bereich erste Fortschritte erzielt, allerdings sind diese Transistoren noch weit von einer praktischen Nutzung entfernt.

3.2 Logische Gatter

Unter einem logischen Gatter versteht man ein aus Transistoren realisiertes Element, mit welchem logische Operationen implementiert werden.

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

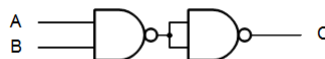
3.3 only-NAND

Es ist möglich, sämtliche logischen Gatter ausschliesslich mit NAND Gattern zu realisieren.

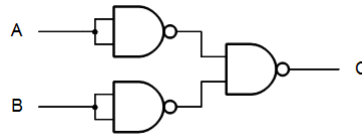
NOT Implementierung mit NAND-Gattern:



AND Implementierung mit NAND-Gattern:

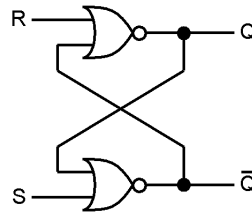


OR Implementierung mit NAND-Gattern:



3.4 RS-Flipflop

Ein RS-Flipflop (auch „Latch“ genannt) ist eine logische Schaltung, mit welcher sich 1 Bit speichern lässt.



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

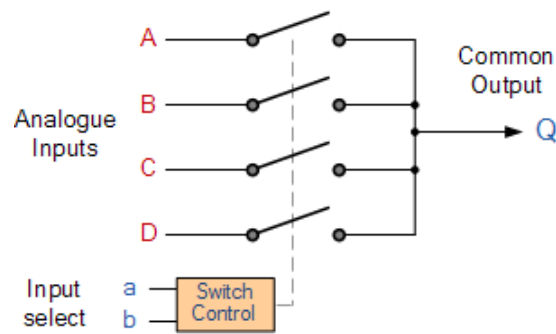
Wenn wir das RS-Flipflop genauer untersuchen, so erkennen wir vier verschiedene Zustände:

- $S = 0$ 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 $\neg Q$ entsprechend 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.

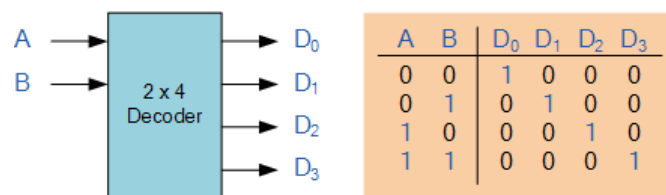
RS-Flipflops werden beispielsweise bei statischen RAM (SRAM) eingesetzt, da sie sehr schnell sind. Bei einer CPU werden solche Latches beispielsweise für den internen Cache oder die Register eingesetzt.

3.5 Multiplexer

Ein Multiplexer (kurz MUX) ist eine Selektionsschaltung, mit der aus einer Anzahl von Eingangssignalen eines ausgewählt und an den Ausgang durchgeschaltet werden kann. Sie sind vergleichbar mit den früheren Telefonschaltzentralen.

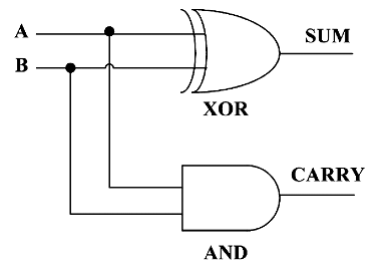


3.6 Decoder



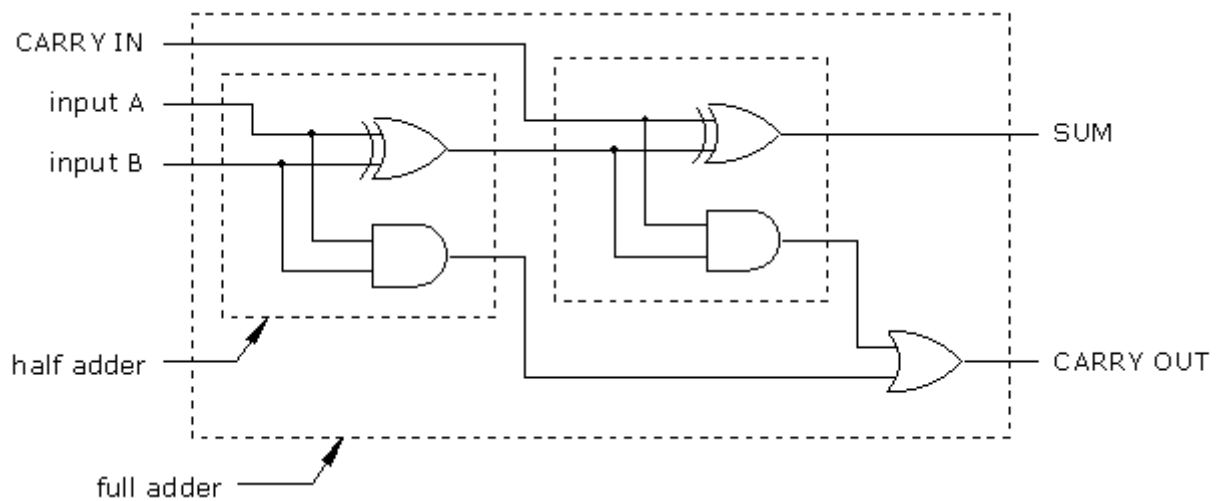
3.7 Arithmetische Schaltungen

3.7.1 Halbaddierer



Ein Halbaddierer ist eine aus logischen Gattern gebaute logische Schaltung, mit welcher sich zwei Bits addieren lassen. Zusätzlich zum Resultat in Form des Summenbits (0 oder 1) , wird das Carry-Bit ausgegeben, falls die Addition zu einem Übertrag führt.

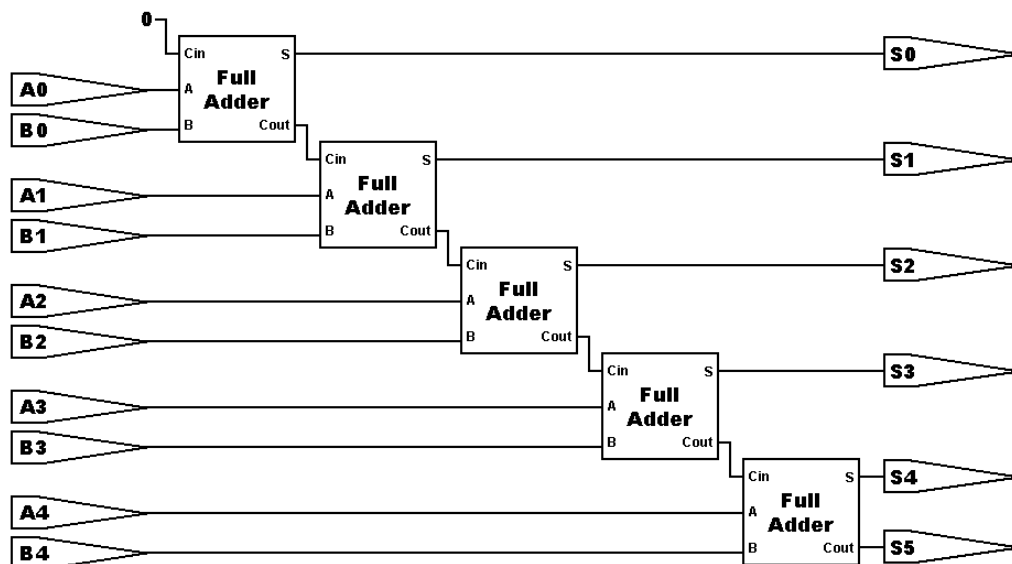
3.7.2 Volladdierer



Beim Volladdierer handelt es sich um einen Halbaddierer, bei dem zusätzlich das Carry-Bit der vorhergegangenen Addition mitberücksichtigt wird. Zur Implementierung werden meist zwei Halbaddierer verbunden. Mit Volladdierern lassen sich also auch mehrstellige Binärzahlen addieren. Bei der ALU (Arithmetic Logic Unit), einem wichtigen Bestandteil des Prozessors, werden mehrere Volladdierer hintereinander geschaltet, um genau dies zu erreichen.

3.7.3 n-Bit Addierer

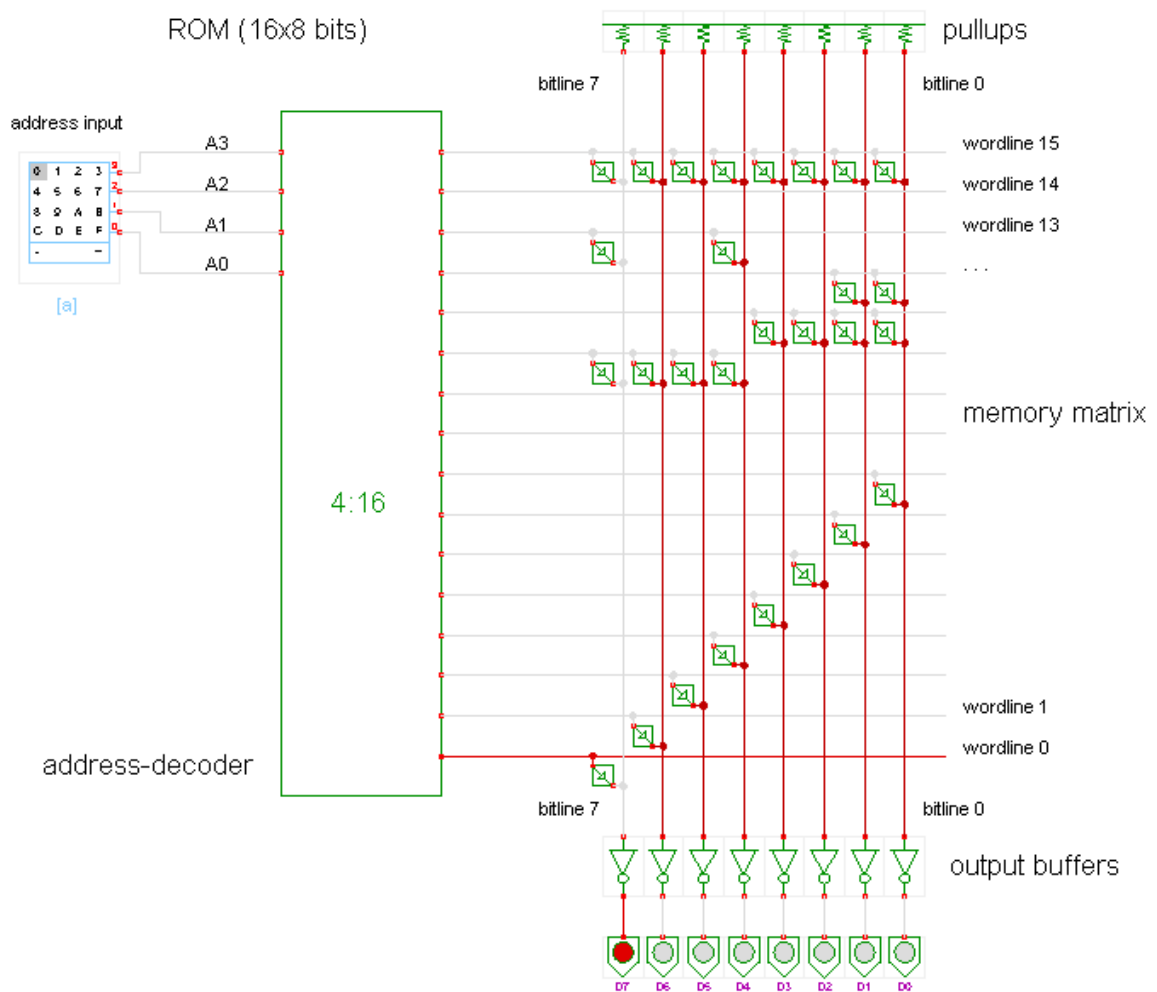
Durch Hintereinanderreihung mehrerer Volladdierer, können wir auch mehrstellige Bit-Zahlen addieren:



3.8 Read-only memory

Read-only memory (ROM) ist ein nicht beschreibbarer Speicher, der aus folgenden Komponenten besteht:

- Address-Decoder: Durch einen Decoder können mit einer 4-bit Adresse 16 Zeilen angesprochen werden.
- Memory-Matrix: In jeder der 16 Zeilen stehen 8-Bit zur Speicherung von Werten zur Verfügung.



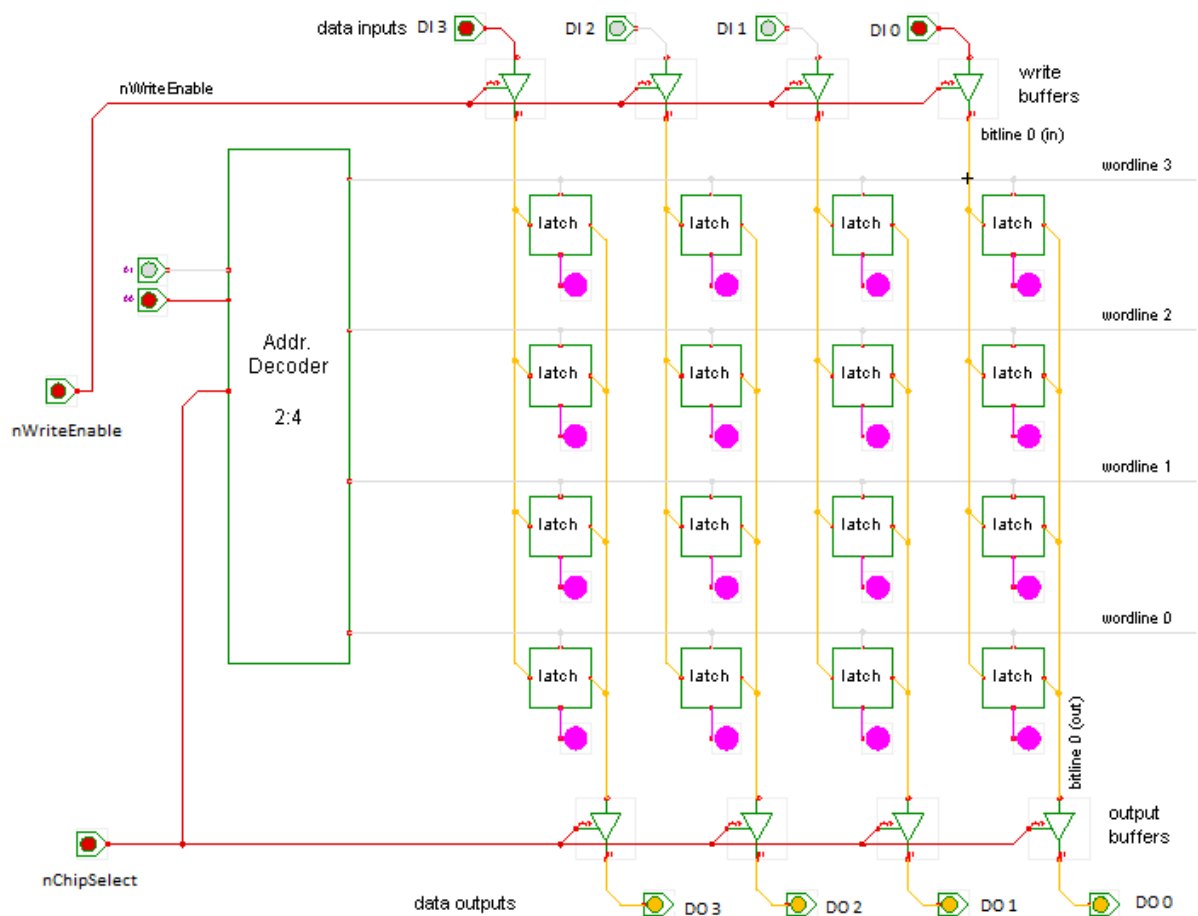
Die vertikale Dimension bzw. die Anzahl Zeilen kann erhöht werden, erfordert dann aber eine Erweiterung des Addressdecoders. Alternativ kann auch die horizontale Dimension, die Wortbreite, unabhängig davon erhöht werden. Damit können pro Zeile mehr Bits gespeichert werden.

Die einzelnen Bits werden beim ROM durch Sicherungen realisiert, welche bei der Produktion

wahlweise durchgebrannt werden, um damit die Werte 0 bzw. 1 zu repräsentieren. Ein nachträgliches Ändern des Speicherinhaltes ist somit nicht möglich. ROMs werden heute nur noch selten eingesetzt und wurden grösstenteils durch PROM (Programmable ROM), EPROM (Erasable Programmable ROM) oder EEPROM (Electrically Erasable Programmable ROM) ersetzt.

3.9 Random-access memory

Im Gegensatz zum ROM, ist der RAM Speicher beschreibbar. RAM besitzt also zusätzlich zu der Memory-Matrix und dem Address-Decoder einen Output Puffer. Mittels eines $nWriteEnable$ Schalters wird bestimmt, ob wir lesen oder schreiben wollen. Mit dem Chipselect Schalter kann der gesamte Chip deaktiviert werden (weder schreiben noch lesen ist mehr möglich).



Generell können wir bei RAM zwischen zwei verschiedenen Arten unterscheiden:

- **Statisches RAM (SRAM):** Statisches RAM wird mit Transistoren (Latches) realisiert. Dadurch ist SRAM sehr schnell, aber auch sehr teuer und benötigt im Gegensatz zum dynamischen RAM relativ viel Platz.

- **Dynamisches RAM (DRAM):** Dynamisches RAM ist mit Kondensatoren aufgebaut. Die Information wird als elektrische Ladung im Kondensator gespeichert. Von Vorteil sind hier die tiefen Produktionskosten und die grosse Dichte, die erreicht werden kann. Dafür ist DRAM im Vergleich mit SRAM relativ langsam und erfordert einen regelmässigen Refresh der Speicherzellen.

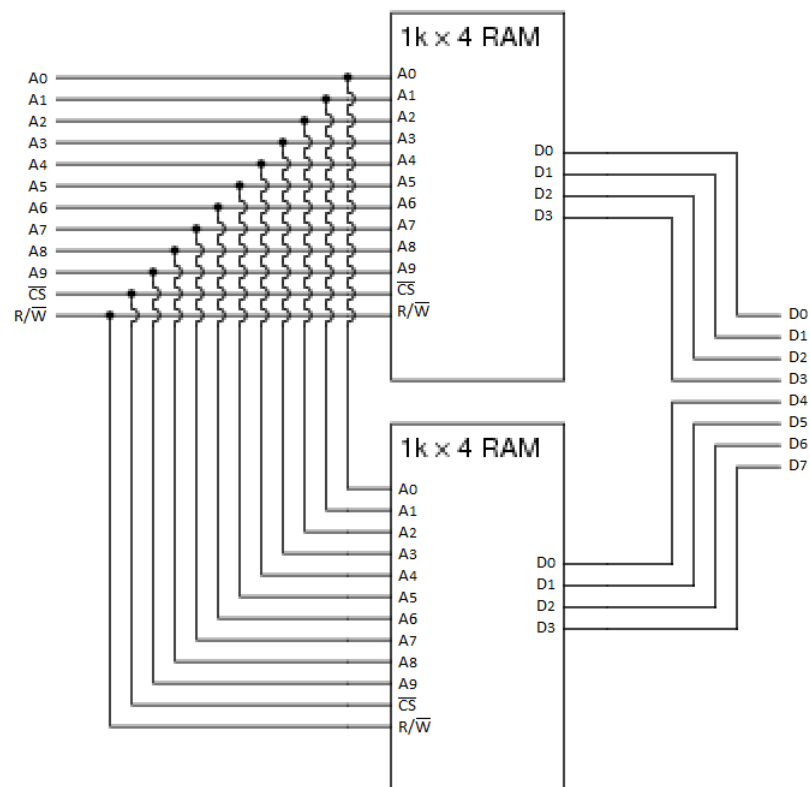
3.10 Speichergrösse

Aus der Bezeichnung eines Memorychips kann üblicherweise die Datenmenge bestimmt werden. Ein 16x8 RAM besitzt 16 Adressen (die mit einem 4 Bit Adressbus und einem 4:16 Decoder angesprochen werden können) und einen Datenbus (Wortlänge) von 8 Bit. Daraus ergibt sich eine Speichergrösse von 128 Bit.

Ein 1k x4 RAM hat 1024 Adressen mit je 4 Bit. Um diese 1024 Adresse anzusprechen, wird ein 10 Bit Adressbus benötigt.

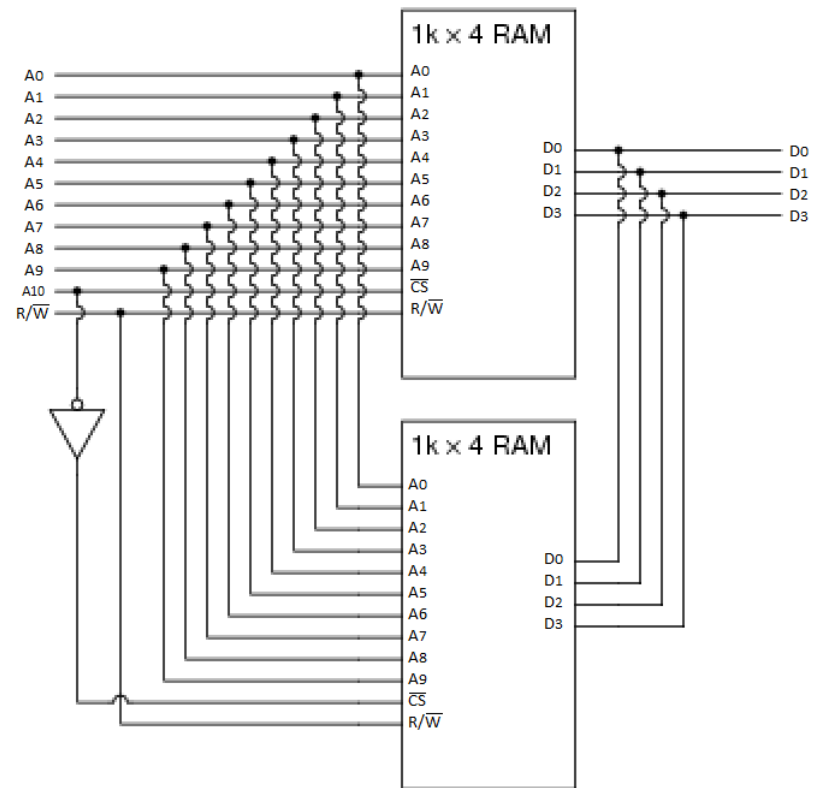
Einzelne RAM Chips können auf zwei verschiedene Wege miteinander verbunden werden, um die Speichergrösse zu erhöhen:

1. Erhöhung der Wortlänge



Durch Verdoppelung des Adressbuses, verdoppelt sich die Wortlänge (Die Datenmenge pro Adresse).

2. Erhöhung der Adressbreite

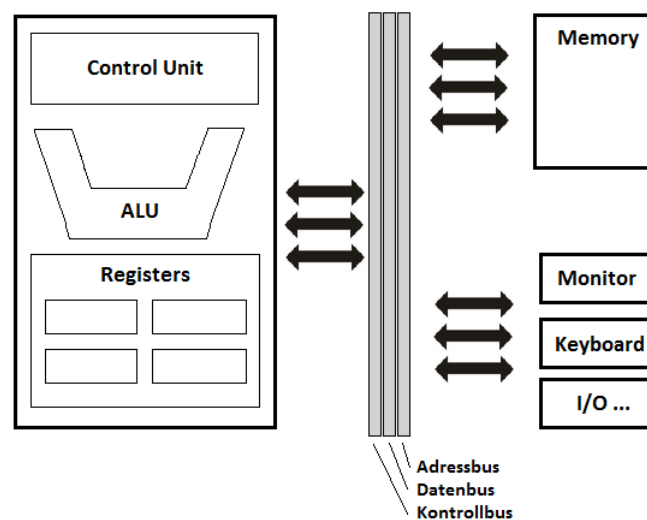


Bei dieser Variante wird der Adressbus verdoppelt, indem ein Bit hinzukommt, welches lediglich den Chipselect beider Memorychips steuert.

Kapitel 4

Computerarchitektur

4.1 Von-Neumann Architektur



Die Von-Neumann Architektur ist ein Referenzmodell für den Aufbau eines Computers, wonach ein Rechner aus folgenden Komponenten besteht:

- CPU - Die zentrale Einheit bestehend aus der ALU, welche Rechenoperationen und logische Verknüpfungen ausführt und der Control Unit, welche die Maschinenbefehle interpretiert und ausführt.
- Memory - Speichert sowohl Programme als auch Daten
- I/O Units - Steuern Ein- und Ausgabe von Daten zum Anwender (Tastatur, Maus, Bildschirm) und anderen Systemen.

4.2 CPU

Die CPU (Central processor unit) besteht aus folgenden Komponenten:

4.2.1 ALU (Arithmetic Logic Unit)

Die ALU führt arithmetische und logische Operationen wie die Addition, die Subtraktion und logische Operationen wie AND, OR und NOT durch. Die ALU ist mit den Registern und über den Datenbus mit dem Memory verbunden. Das Resultat wird an das Ziel-Register bzw. an den Speicher weitergeleitet. Die ALU hält für die Control-Unit einige Flags (Indikatoren) bereit, an denen die CU erkennen kann, wie die Operation verlaufen ist (Überlauf, Gleichheit usw.)

4.2.2 CU (Control Unit)

Die CU koordiniert den Ablauf der Schritte der Befehlsausführung. Sie holt, interpretiert und exekutiert eine Instruktion nach dem anderen.

4.2.3 Clock

Jede Operation der CPU und des System Bus wird durch die interne Clock synchronisiert, die zu einer konstanten Rate einen Takt vorgibt. Die grundlegendste Zeiteinheit im Computer ist somit ein Machine Cycle. Die Taktrate der CPU gibt an, wie viele Machine Cycles sie pro Sekunde vorgibt: 1 GHz entspricht 1 Mrd. Cycles pro Sekunde. Ein Machine Cycle dauert somit 1 Nanosekunde. Ein Maschinenbefehl benötigt mindestens einen Cycle, einige jedoch bis zu 50 (z.B. multiply). Befehle, welche Zugriff auf den Arbeitsspeicher verlangen, haben oft leere Cycles (sogenannte Wait states), weil sich die Geschwindigkeit der CPU von der des System Bus und des Speichers unterscheidet.

4.3 Register

Register sind sehr schnelle RAM (SRAM) Bausteine, die bedeutend schneller als das normale Memory benutzt werden können, da sie sich direkt in der CPU befinden.

4.3.1 Instruction Pointer (IP)

Der IP enthält die Speicher-Adresse der Instruktion, die als nächstes zu holen (fetch) und auszuführen (execute) ist. Er wird jeweils nachgeführt (inkrementiert) um so viele Bytes, wie soeben herbeigeschafft wurden. Es ist auch möglich, mit Sprunganweisungen eine komplett andere Speicheradresse zu setzen, die als nächstes zu holen und auszuführen ist.

4.3.2 Adress-Register (AR)

Das AR enthält die Adresse der zu holenden Instruktion oder die Adresse der zu lesenden oder zu schreibenden Daten.

4.3.3 Instruktionsregister (IR)

Das IR enthält in der Fetch Phase die als nächstes auszuführende Instruktion und behält diese bis ans Ende der Ausführung. Das IR hat eine Kapazität, die der längsten aller Instruktionen entspricht. Bei kürzeren Instruktionen werden überzählige Bytes ignoriert.

4.3.4 Accumulator / Prozessor Register

Register wie EAX, BAX etc. werden zur temporären Speicherung von Daten / Adressen genutzt. Sie sind oft Quelle bzw. Ziel von arithmetischen oder logischen Operationen. Der Zugriff ist bedeutend schneller als auf die Daten im Speicher, da sie direkt im CPU Chip implementiert sind, dafür ist deren Anzahl und Speicherplatz beschränkt.

4.4 Memory (Programm-/Datenspeicher)

Der Programm-/Datenspeicher wird zwar oft als Teil des Prozessorsystems betrachtet und ist intensiv daran beteiligt, allerdings ist er nicht Bestandteil der CPU. Sowohl Programme als auch Daten werden im gleichen Baustein gespeichert. Der Programm-Speicher enthält die Programm-instruktionen, welche von der CPU fortlaufend geholt und ausgeführt werden. Der Datenspeicher enthält die Plätze für die Variablen, auf welche die Programme zugreifen.

4.5 Bussystem

Die Komponenten sind über ein Bussystem verbunden, welches aus folgenden Komponenten besteht:

- Adressbus - Beinhaltet 8, 16, 32 oder 64 Leitungen (Bit) zur Adressierung des zu lesenden bzw. schreibenden Speichers. Bei einem 32-Bit Datenbus können 2^{32} Speicherzellen (bei 8 Bit pro Zelle entspricht dies ungefähr 4 Gigabyte), die maximal direkt adressiert werden können.
- Datenbus - Transportiert Daten zwischen den einzelnen Komponenten. Die Bezeichnung „32-Bit“ oder „64 Bit“ CPU bezieht sich üblicherweise auf die Breite des Datenbuses.
- Steuerbus - Übernimmt die Kontrolle des Bussystems. Unter anderem wird die Lese-/Schreibsteuerung (Richtung des Databuses) auf dem Steuerbus übertragen.

4.6 Instruction Set

Jede CPU besitzt ein sogenanntes Instruction Set - eine Sammlung verschiedener Instruktionen zu deren Programmierung, beispielsweise das Laden eines Wertes aus dem Speicher in ein Register oder das Addieren zweier Zahlen. Da der Prozessor jedoch nur die 0 und die 1 versteht, müssten Programmierer die CPU theoretisch durch sinnvolle Zeichenfolgen wie 1010100111111111 programmieren. Damit die Programmierer stattdessen mit symbolischen Bezeichnungen arbeiten können, hat die Control Unit Zugriff auf eine „Zuordnungstabelle“, in welcher z.B. dem Code 10101001 bzw. seiner hexadezimalen Entsprechung (Opcode) A9 ein sogenannter Mnemonic Code zugeordnet, hier beispielsweise LDA, welcher die CPU zum Laden eines Wertes anweist.

Abstrakt	Mnemonic	Opcode	Maschinencode
Laden des Wertes 255 in ein Register	LDA #\$FF	A9 FF	10101001 11111111

4.7 Instruction Execution Cycle

Ein Maschinencode Befehl kann in verschiedene Unter-Operationen unterteilt werden, sogenannte Instruction Execution Cycles. Wenn die CPU z.B. eine Operation zweier Zahlen im Speicher durchführen soll, muss sie zuerst die Adresse der beiden Zahlen/Operanden berechnen, die Adressen auf

den Addressbus legen, auf den Speicher warten und so weiter.

Eine Addition kann in einer Hochsprache wie C/C++ mit einer einzigen, atomaren Instruktion in der Form $\mathbf{a} + \mathbf{b}$ durchgeführt werden. Ein sogenannter Compiler übersetzt solche Befehle in für den Prozessor verständliche Maschineninstruktionen. Für die genannte Addition sieht dies beispielsweise so ähnlich aus:

```
mov Reg ← [a]
add Reg ← [b]
mov [c] ← Reg
```

Die Schreibweise ist sehr allgemein gehalten und syntaktisch nicht ganz korrekt. Der Begriff „Reg“ müsste durch ein in der jeweiligen CPU Architektur existierendes Register (z.B. `eax` bei Intel Prozessoren) ersetzt werden. Der Pfeil ist ausserdem durch ein Komma zu ersetzen.

Bei der Ausführung einer Anwendung wird der Code in den Arbeitsspeicher geladen. Anschliessend wird das **Instruction Pointer Register** (Befehlszähler, Programmcounter) auf die Instruktion des Programmes gesetzt, die als erstes auszuführen ist.

Unter Anweisungen der Control Unit, werden nun folgende Schritte ausgeführt:

1. Fetch-Instruktion/Phase:

- Der Inhalt des Instructionpointers wird in das Adressregister übertragen.
- Über das Bussystem wird der Wert dieser Adresse (die Instruktion `mov eax,[a]` „ge-fetched“, d.h., aus dem Speicher gelesen und ins Instruktionsregister geschrieben.
- Als nächstes wird der Instructionpointer um so viele Bytes inkrementiert, wie die Instruktion besitzt; er zeigt nun auf die nächste Instruktion.

2. Decode-Instruktion/Phase (Oft als Bestandteil der Fetch-Phase aufgeführt)

- Die Control-Unit analysiert den erhaltenen Befehl.

3. Execute-Phase:

- Die Control-Unit verlangt, dass wir zuerst den Inhalt der Variable `a` im Speicher auslesen müssen. Dafür wird die Adresse von `a` ins Adressregister gelegt (die vorhandene wird dabei überschrieben).
- Die adressierte Speicherstelle `[a]` wird nun ausgelesen und in das angegebene Register gelegt. Die eckigen Klammern zeigen dabei, dass nicht als Konstante, sondern als Speicheradresse zu interpretieren ist. Es wird also der Wert an der Speicheradresse von `a` gelesen.
- Der Inhalt des Programmzählers wird wieder ins Adressregister übertragen; die nächste Instruktion wird gelesen: `add eax,[b]`. Der Instructionpointer wird wieder inkrementiert.
- Nach der Interpretation wird der Wert der Speicherstelle `b` angefordert
- Mit dem Befehl `add` wird die ALU angewiesen, den gerade ausgelesenen Wert von `b` mit dem angegebenen Register (in welchem wir den Wert von `a` abgelegt haben) zu addieren. Das Resultat wird wieder in das Register gespeichert, in welchem der Wert `a` gespeichert war.

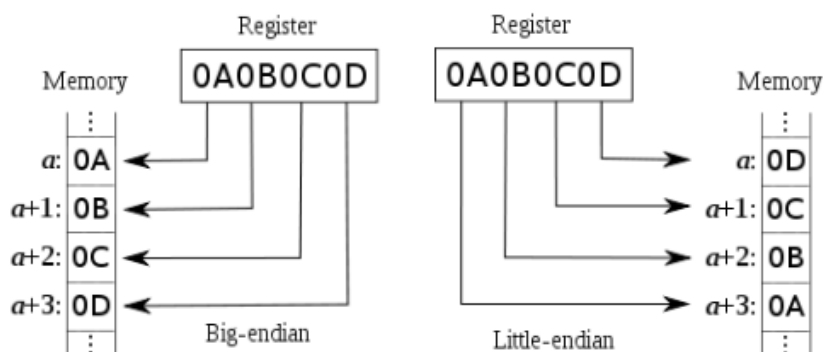
- Wieder wird die nächste Instruktion aus dem Speicher gelesen und ins Instruktionsregister übertragen: **mov [c],eax**. Der Instructionpointer wird inkrementiert.
- Nach Analyse der CU wird die Adresse der Variable c ins Adress-Register gelegt, der Wert des Registers auf den Datenbus gelegt und mit dem Schreibbefehl auf dem Control bus werden diese Daten (Das Resultat der Rechnung) in die adressierte Speicherstelle c geschrieben.

Kapitel 5

6502 Assemblersprache

5.1 Die 6502 Architektur

Der 6502 Microprocessor gehört zu der 8-Bit CPU Kategorie. Er hat lediglich einige wenige interne Register, 64 kb Memory, einen 16-Bit Addressbus und einen 8-Bit Datenbus. Der 6502 funktioniert nach der „Little Endian“ Bytereihenfolge, das heisst, das Byte mit den niederstwertigen Bits (d. h. die am wenigsten signifikanten Stellen), wird als erstes genannt.

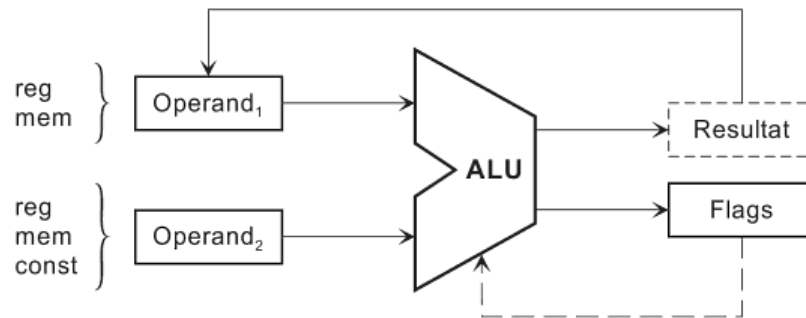


5.1.1 Register des 6502

1. Ein 8-bit Accumulator Register (A)
2. Zwei 8-bit Index Register (X und Y)
3. Ein 8-bit Prozessor Status Register (SR)
4. Ein 8-bit Stack Pointer (SP)
5. Ein 16-bit Program Counter (PC) bestehnd aus zwei 8-Bit Register PC LowByte (PCL) und PC HighByte (PCH)

5.2 Logische / Arithmetische Operationen

Logische und arithmetische Operationen werden in der ALU durchgeführt. Sie benötigen jeweils zwei Operanden, die die selbe Grösse haben müssen und aus einem Register, Speicher oder aus einer konstanten Angabe nach dem OP Code stammen.



5.3 Memory Transfers

Der Grossteil der Maschinen-Instruktionen sind für den Transfer von Daten zwischen dem Speicher und den Registern zuständig. Beispielsweise ermöglicht der Befehl LDA (Load Accumulator) das Laden eines Wertes in das Akku Register (beispielsweise um anschliessend eine Addition oder Subtraktion durchzuführen).

Addressing Mode	Bsp.	Beschreibung	OP-Code
Immediate	LDA #10	Lädt \$10 (dec. 16) in den Akku	A9 10
Zero Page	LDA \$00	Lädt den Akku mit dem Wert an der Zero Page Adresse \$00	A5 00
Zero Page,X	LDA \$10,X	Lädt den Akku mit dem Wert an der Zero Page Adresse berechnet aus \$10 addiert mit dem Inhalt des Index Registers X	B5 10
Absolute	LDA \$1234	Lädt den Akku mit dem Wert an Adresse \$1234	AD 34 12
Absolute,X	LDA \$1234,X	Lädt den Akku mit dem Wert an der Adresse berechnet aus \$1234 und dem Wert des Index Registers X	BD 34 12
Absolute,Y	LDA \$1234,Y	Lädt den Akku mit dem Wert an der Adresse berechnet aus \$1234 und dem Wert des Index Registers Y	B9 34 12
(Indirect,X)	LDA (\$20,X)	Lädt den Akku mit dem Wert an der Adresse, welche an der Adresse berechnet aus \$20 addiert mit dem Inhalt des Index Registers X gespeichert ist.	A1 20
(Indirect),Y	LDA (\$20),Y	Lädt den Akku mit dem Wert an der Adresse, berechnet aus dem Wert an Adresse \$20 addiert mit dem Inhalt des Index Registers Y.	B1 20

5.4 Addressing Modes

Grundsätzlich wird zwischen zwei Gruppen von Adressierungsmethoden unterschieden: Den indexierten und den nicht-indexierten.

5.4.1 Accumulator

Einige Befehle erfordern die Angabe eines Registers, z.B. INC A, oder DEC A

5.4.2 Immediate

Bei einigen Befehlen werden die Daten direkt als zweites Byte nach dem OP-Code angegeben. Dazu wird das Raute Symbol # genutzt, z.B.

1 LDA #\$B2

Lädt ein Byte (den hexadezimalen Wert B2) in den Akkumulator. Das Symbol \$ zeigt, dass der folgende Wert im Hexadezimalsystem betrachtet wird.

Fetch/ Execute	Operation	Beschreibung
Fetch	ABL <- PCL	Programmcouter ins Adr. Register schreiben
	ABH <- PCH	
	R \overline{W} <- 1	Read-mode
	Memory Access	Anweisung ins DBR schreiben
	IR <- DBR	
Execute	ABL <- PCL + 1	Wert PC + 1 in ABR schreiben
	ABH <- PCH + 1	
	R \overline{W} <- 1	Read-mode
	Memory Access	Wert "Erstes Byte" an Memory[PC+1] lesen
	Akku <- DBR	Wert in Akku schreiben
	PC <- PC + 2	PC um 2 inkrementieren

5.4.3 Absolute Addressing

Bei der absoluten Adressierung wird die Memory Adresse des zu ladenden Wertes direkt angegeben. Diese Adressierungsart ermöglicht somit, die ganzen 65k Bytes des Speichers zu adressieren. Beispiele:

Wert an Speicheradresse \$ABCD in den Akku schreiben:

1 LDA \$ABCD

Der CPU übersetzt LDA zu AD gem. OP Code Liste, die Speicheradresse wird vertauscht. Hex-dump:

AD CD AB

Der Befehl wird nun wie folgt abgearbeitet:

Fetch/ Execute	Operation	Beschreibung
Fetch	$ABL <- PCL$ $ABH <- PCH$ $R \overline{W} <- 1$ Memory Access $IR <- DBR$ $Instruction == AD$	Programmcounter ins Adr. Register schreiben Read-mode Nächste Anweisung ins DBR schreiben Im IR Register steht nun "AD" Der CPU erkennt die Funktion AD. Er weiss nun dass die 2 folgenden Bytes die Teile der eigentlichen Adresse sind. Es beginnt der Execute Teil.
Execute	$ABL <- PCL + 1$ $ABH <- PCH + 1$ $R \overline{W} <- 1$ Memory Access "Dummy Register" <- DBR $ABH L <- PC + 2$ $R \overline{W} <- 1$ Memory Acces $ABH <- DBR$ $ABL <- "Dummy Register"$ $R \overline{W} <- 1$ Memmory Access Akku <- DBR $PC <- PC + 3$	Wert PC + 1 in ABR schreiben Read-mode Wert "CD" an Memory[PC+1] lesen Ablageort leider nicht bekannt, deshalb nennen wir das Register Dummy Register Wert PC + 2 in ABR schreiben Auf Read schalten Wert "AB" an Memory[PC+2] lesen Wert "AB" in ADB High Byte schreiben Wert "CD" in ADB Low Byte schreiben Auf Read schalten Wert an Speicherstelle \$ABCD lesen Wert in Akku schreiben PC um 3 inkrementieren, "ADCDAB" = 3 Byte

5.4.4 Relative

Die relative Adressierung wird beim 6502 nur für Branch-Operationen (Verzweigungen) genutzt. Das Byte nach dem OP-Code ist der Verzweigungs-Offset.

Beispiel:

Wenn wir Zwei Zahlen vergleichen wollen, können wir den BNE Befehl benutzen. In unserem Beispiel vergleichen wir 6 mit 5. Das Zero Flag bleibt auf 0. BNE lässt den Programmcounter zur Adresse von notequal: jumpen. Diese Adresse ist vom jetzigen Programmcounter aus gesehen, also relativ zur aktuellen Position, 3 Bytes entfernt.

```

1 LDA #$06
2 CMP #$05
3 BNE notequal
4 LDA #$09
5 BRK
6 notequal:
7 LDA #$0A

```

Im Hexdump wird folgendes ersichtlich.

d0 03

Der BNE Befehl hat als Paramter 03. Der Programmcounter wird um 3 erhöht und das Programm wird an der Position Programmcounter + 4 fortgesetzt.

5.4.5 Zero-Page

Der Speicher wird beim 6502 mit 16 Bit adressiert und kann somit aufgeteilt werden in zwei 8 Bit (= 2 Byte) Werte, die jeweils 256 Bytes adressieren können. Beim ersten Address-Byte spricht man dabei auch von der Seitenzahl (Pages). Wir haben also insgesamt 256 Pages mit jeweils 256 Adressen. Sämtliche Adressen auf Page 0, also die Adressen \$0000 bis \$00FF können somit mit nur einem Byte angesprochen werden, während das High-Byte immer 0 ist, was der Prozessor bedeutend schneller durchführen kann. Diese Adressen werden auch Zero-Page Adressen genannt. Bsp. LDA \$35 lädt den Wert an der Adresse \$0035 in den Akku.

5.4.6 Indirect

Diese Adressierung wird nur vom JMP Befehl verwendet. JMP (\$1000) führt dazu, dass der Instruction Pointer auf die Adresse gesetzt wird, die an Adresse 0x1000 gespeichert ist.

5.4.7 Absolute Indexed Addressing

Bei dieser Adressierung wird ein Indexregister (X oder Y) zu einer im 2. und 3. Byte angegebenen absoluten Adresse addiert. Dies ist nützlich, um zum Beispiel 10 Bytes zu füllen, die von der Adresse 0x1009 bis 0x1000 runtergezählt werden.

5.4.8 Indexed Zero Page Addressing

Funktioniert genau wie bei absolute indexed, allerdings ist die Zieladresse auf die ersten 0xFF Bytes limitiert. Falls die Zieladresse über 0xFF rausschaut, wird sie abgeschnitten: LDA \$C0,X und X ist \$60, dann wird die Adresse \$120 (\$C0 + \$60 = \$120) durch abschneiden des Carrys auf \$20 gesetzt.

5.4.9 Indexed Indirect Addressing (Indirect,X)

Bei dieser Adressierung wird der Wert im zweiten Byte zum Wert im X Register hinzuaddiert. Bsp: LDA (\$20,X). Falls X den Wert \$04 enthält, so wird zuerst X zu \$20 addiert (=\$24) und anschließend der Wert an der Zieladresse aus dem Speicher an Stelle \$24 ins low-Byte und der Wert an Stelle \$25+1 in das high-Byte geladen, welche zusammen dann die effektive Zieladresse ergeben. Diese muss allerdings in der Zeropage liegen.

Beispiel:

```
1 LDX #02
2 LDA ($15, X)
```

Speicher:

\$0017	EF
\$0018	CD
...	
\$CDEF	09

Der Wert 02 im X Register wird zu \$15 addiert. Es ergibt sich die Adresse \$17.

Nun werden die Werte an der Adresse \$17 und \$18 ausgelesen. Die Werte sind \$17 = EF und \$18

= CD. Die neue Adresse ist \$CDEF. Der Wert 09 von der Adresse \$CDEF wird in den Akku geladen.

5.4.10 Indirect Indexed Addressing (Indirect),Y

Dieser Modus nutzt nur das Y-Register. Das zweite Byte der Instruktion zeigt auf eine Memory Adresse in der Zeropage. Der Inhalt dieser Speicheradresse wird zum Inhalt des Y Registers addiert, das Resultat ist das Low Byte der effektiven Adresse. Der Übertrag dieser Addition wird zum Inhalt der nächsten Zeropage Adresse addiert, was dann das High Byte der effektiven Adresse ergibt.

Beispiel:

Speicher:

\$0015	CD
\$0016	AB
...	
\$ABCF	08

Beispiel:

- ```
1 LDY #02
2 LDA ($15),Y
```

Zuerst werden die Werte an der Speicheradresse \$15 und \$16 ausgelesen. Dies ergibt die Adresse \$ABCD. Nun wird der Wert aus dem Y-Register (\$2) hinzuaddiert. Dies ergibt nun die effektive Zieladresse \$ABCF, an welcher der Wert \$08 in den Akku geladen wird.

## 5.5 Conditional Jumps

Um If-Else Abfragen mit Assembler zu realisieren, werden sogenannte Conditional Jumps benutzt, um in bestimmten Fällen (wenn bestimmte Flags gesetzt sind) an einen neuen Ort im Programm zu springen. Um die Flags zu setzen, ohne dabei die Register zu überschreiben, gibt es den Befehl CMP, welcher eine Subtraktion des einen mit dem anderen Wert durchführt. Falls das Resultat dieser Subtraktion 0 ist (das Zeroflag wird gesetzt), so stimmen die beiden zu vergleichenden Werte überein. Nun können wir mit dem Befehl BEQ (Branch Equal) zu einer Speicheradresse springen, in welchem dieser Fall abgearbeitet wird. Mit BNE (Branch Not Equals) können wir somit die Else Verzweigung, als den Fall, dass die beiden Werte nicht übereinstimmen, abfangen.

### 5.5.1 Conditional Jump x86 Spezifisch

unsigned

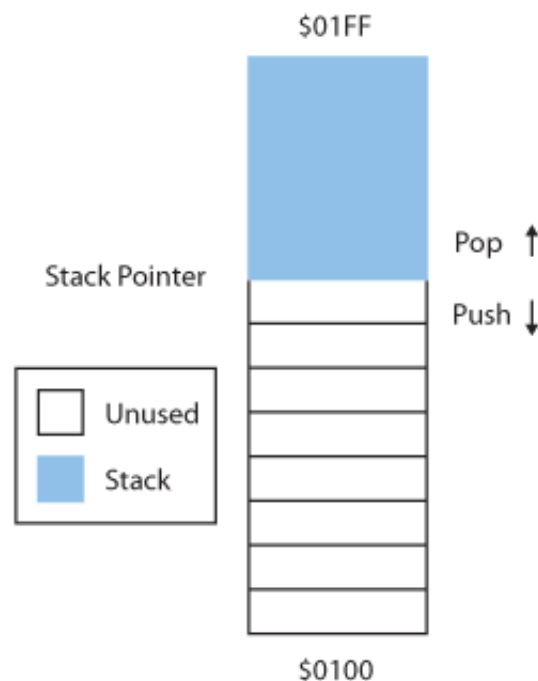
| Relation   | Differenz      | Flags                        | Memonic                |
|------------|----------------|------------------------------|------------------------|
| $A = B$    | $A - B = 0$    | z                            | jeq <adr> (equal)      |
| $A \neq B$ | $A - B \neq 0$ | $\bar{z}$                    | jne <adr> (notequal)   |
| $A < B$    | $A - B < 0$    | c                            | jb <adr> (below)       |
| $A \leq B$ | $A - B \leq 0$ | $c \vee z$                   | jbe <adr> (belowequal) |
| $A > B$    | $A - B > 0$    | $\overline{c \vee z}$        | ja <adr> (above)       |
| $A \geq B$ | $A - B \geq 0$ | $\overline{c \vee z} \vee z$ | jae <adr> (aboveequal) |

### signed

| Relation   | Differenz      | Flags                                   | Memonic                                     |
|------------|----------------|-----------------------------------------|---------------------------------------------|
| $A = B$    | $A - B = 0$    | $z$                                     | <code>jeq &lt;adr&gt;</code> (equal)        |
| $A \neq B$ | $A - B \neq 0$ | $\bar{z}$                               | <code>jne &lt;adr&gt;</code> (notequal)     |
| $A < B$    | $A - B < 0$    | $s \text{ xor } of$                     | <code>jl &lt;adr&gt;</code> (less)          |
| $A \leq B$ | $A - B \leq 0$ | $(s \text{ xor } of) \vee z$            | <code>jle &lt;adr&gt;</code> (lessequal)    |
| $A > B$    | $A - B > 0$    | $\overline{(s \text{ xor } of)} \vee z$ | <code>jg &lt;adr&gt;</code> (greater)       |
| $A \geq B$ | $A - B \geq 0$ | $\overline{s \text{ xor } of}$          | <code>jge &lt;adr&gt;</code> (greaterequal) |

## 5.6 Stack

Der Stack ist ein spezieller Bereich des Memory. Den Stack kann man sich wie ein Stapel Bücher vorstellen, auf welchen man die Bücher eines nach dem anderen stappelt und später, beim zuletzt hingelegten Buch, anfängt wieder zu entfernen (auch LIFO „Last in, first out“-Prinzip genannt).



Wird in Hochsprachen wie C, C++ oder Java in einer Funktion eine lokale Variable erstellt, so wird diese im Stack gespeichert. In Assembler gibt es diverse Instruktionen, die mit dem Stack arbeiten.

### 5.6.1 PHA: Push Akku Value on Stack

Unser Beispiel:

Code:

### PHA

Wert des Akkus wird auf den Stack gelegt

| Fetch/ Execute | Operation             | Beschreibung                                                  |
|----------------|-----------------------|---------------------------------------------------------------|
| Fetch          | ABL <- PCL            | PC in Adressbusregister schreiben                             |
|                | ABH <- PCH            |                                                               |
|                | R $\overline{W}$ <- 1 | Auf Read schalten                                             |
|                | Memory Access         | Nächste Anweisung lesen                                       |
|                | IR <- DBR             | Anweisung in IR Register schreiben                            |
|                | IR = 48               | Der CPU erkennt die Funktion 48. Es beginnt der Execute Teil. |
| Execute        | ABL <- SP             | Stackpointer Wert in AB Low Byte schreiben                    |
|                | ABH <- 0x01           | Wert 0x01 in AB High Byte schreiben, Standard bei Stackpoint  |
|                | DBR <- A              | Wert des Akkumulators in DBR schreiben                        |
|                | R $\overline{W}$ <- 0 | Auf Write schalten                                            |
|                | Memory Access         | Wert des Akkus auf Stack legen                                |
|                | SP <- SP -1           | Stackpointer dekrementieren                                   |
|                | PC <- PC + 1          | PC um 1 erhöhen                                               |

### 5.6.2 PHP: Push Programcounter Value on Stack

Unser Beispiel:

Code:

#### PHP

Wert des Programmcounters wird auf den Stack gelegt

| Fetch/ Execute | Operation             | Beschreibung                                                  |
|----------------|-----------------------|---------------------------------------------------------------|
| Fetch          | ABL <- PCL            | PC in Adressbusregister schreiben                             |
|                | ABH <- PCH            |                                                               |
|                | R $\overline{W}$ <- 1 | Auf Read schalten                                             |
|                | Memory Access         | Nächste Anweisung lesen                                       |
|                | IR <- DBR             | Anweisung in IR Register schreiben                            |
|                | IR = 08               | Der CPU erkennt die Funktion 08. Es beginnt der Execute Teil. |
| Execute        | ABL <- SP             | Stackpointer Wert in AB Low Byte schreiben                    |
|                | ABH <- 0x01           | Wert 0x01 in AB High Byte schreiben, Standard bei Stackpoint  |
|                | DBR <- PC             | Wert des Programcounters in DBR schreiben                     |
|                | R $\overline{W}$ <- 0 | Auf Write schalten                                            |
|                | Memory Access         | Wert des Akkus auf Stack legen                                |
|                | SP <- SP -1           | Stackpointer dekrementieren                                   |
|                | PC <- PC + 1          | PC um 1 erhöhen                                               |

### 5.6.3 PLA: Pop Data from Stack, store it in Akku

Unser Beispiel:

Code:

**PLA**

"Oberster" Wert der auf dem Stack wird in den Akku geschrieben.

| Fetch/ Execute            | Operation             | Beschreibung                                           |
|---------------------------|-----------------------|--------------------------------------------------------|
| Fetch                     | ABR <- PC             | PC in Adressbusregister schreiben                      |
|                           | R $\overline{W}$ <- 1 | Auf Read schalten                                      |
|                           | Memmmory Access       | ZUgriff auf Memory, nächste Anweisung lesen            |
|                           | IR <- DBR             |                                                        |
|                           | IR = 68               | Der CPU weiss, nun muss er einen PLA Befehl abarbeiten |
| Execute<br>A <- Mem[SP+1] | ABL <- SP+1           | Stackpointer + 1 in ABL schreiben                      |
|                           | ABH <- 0x01           | ABH auf 0x01 setzen, Standard bei Stack                |
|                           | R $\overline{W}$ <- 1 | Auf Read schalten                                      |
|                           | Memory Acces          | Lesen n Speicherzelle Mem[SP+1]                        |
|                           | A <- DBR              | Wert in Akku schreiben                                 |
|                           | SP <- SP+1            | Stackpointer um 1 inkrementieren                       |
|                           | PC <- PC + 1          | PC um 1 inkrementieren                                 |

### 5.6.4 Endian

| Little Endian                                                                                                                              | Big Endian                                                                                                                                            |
|--------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| Bei Little-Endian (wörtlich „Klein-Ender“) wird das Byte mit den niederwertigen Bits (Rechts) an der kleinsten Speicheradresse gespeichert | Bei Big-Endian (wörtlich „Groß-Ender“) wird das Byte mit den höchwertigen Bits (Links) zuerst gespeichert, das heißt an der kleinsten Speicheradresse |

**Beispiel 4.** Im folgenden Beispiel wird die Ganzzahl 439.041.101 (Vierhundertneununddreißig Millionen...) als 32-Bit-Integer-Wert gespeichert (Binär: 00011010 00101011 00111100 01001101, hexadezimal: 1A 2B 3C 4D). Die Speicherung erfolgt in vier Bytes ab der hypothetischen Speicheradresse 10000.

|         | Big Endian |     |          | Little Endian |     |          |
|---------|------------|-----|----------|---------------|-----|----------|
| Adresse | Hex        | Dez | Binär    | Hex           | Dez | Binär    |
| 10000   | 1A         | 26  | 00011010 | 4D            | 77  | 01001101 |
| 10001   | 2B         | 43  | 00101011 | 3C            | 60  | 00111100 |
| 10002   | 3C         | 60  | 00111100 | 2B            | 43  | 00101011 |
| 10003   | 4D         | 77  | 01001101 | 1A            | 26  | 00011010 |

## 5.7 Subroutines

Subroutinen ermöglichen, oft benutzte Codeprozeduren auszulagern und aufrufen zu können. Zu einer Subroutine gehört beispielsweise deren Bezeichnung, Übergabewerte (Parameter) um dynamischen Einfluss auf deren Funktion zu haben und der Rückgabewert. Funktionen mit Hilfe eines speziellen Bereichs im RAM ermöglicht, dem Stack. Um in eine Subroutine zu springen, nutzen wir den Befehl JSR (Jump to Subroutine).

Wollen wir von der Subroutine zum normalen Programmfluss zurückkehren, nutzen wir die RTS (Return from Subroutine).

### 5.7.1 JSR

Der JSR Befehl hat 2 Aufgaben

- Aktueller PC + 2 auf Stack legen. Dient als Returnadresse für den RTS Befehl  
ST <- PCL  
ST+1 <- PCH
- Zur Angegebenen Adresse jumpen

### 5.7.2 RTS

Mit dem RTS Befehl springen wir von aus einer Subroutine zurück in den normalen Programmfluss. Als Zieladresse dient hier der PC, der während dem JSR Befehl auf den Stack gelegt wurde. Diese Adresse wird vom Stack gepopt und in den PC geschrieben.

## Kapitel 6

# Die Programmiersprache C

C ist eine Hochsprache, das heisst, sie abstrahiert die Assemblersprachen. Trotzdem ist C sehr systemnahe, weshalb Betriebssysteme normalerweise in C geschrieben werden. Während Assembler quasi 1 zu 1 in den Bytecode der Maschinenbefehle umgewandelt wird, ist C Code für die CPU unverständlich und muss zuerst mit Hilfe eines Compilers in die Assemblersprache der jeweiligen CPU übersetzt werden.

### 6.1 Variablen

Der Compiler/Linker reserviert im Speicher eine bestimmte Anzahl Bytes in Form eines zusammenhängenden Blocks, welchem ein Namen zugeordnet wird, über welchen der Programmierer zugreifen und mit einem Wert versehen kann. Die Grösse dieses Speicherblocks und der Interpretationskontext wird durch einen Datentyp definiert. Für ganze Zahlen gibt es beispielsweise den Typ „Integer“, welcher auf den meisten Computern 4-Byte an Speicher benötigt.

Beispiel: Mit

```
1 int x = 1;
```

können wir eine Variable vom Typ Integer definieren. Es werden 4-Byte im Speicher reserviert, auf die wir fortan über den Namen i zugreifen können.

| Adresse | Inhalt |
|---------|--------|
| 282001  | ...    |
| 282002  | ...    |
| 282003  | ...    |
| 282004  | 01     |
| 282005  | 00     |
| 282006  | 00     |
| 282007  | 00     |
| 282008  | ...    |
| 282009  | ...    |

## 6.2 Pointers

Ein Pointer ist eine Variable, die als Wert eine Speicheradresse besitzt und somit auf eine andere Variable „zeigt“.

### 6.2.1 Einfaches Dereferenzieren

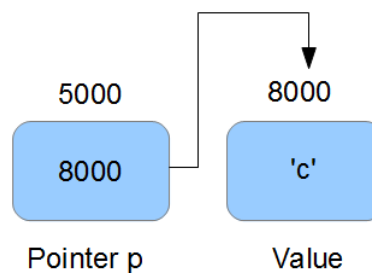
Beispiel:

```
1 char value = 'c';
2 char *p = &value;
```

Zeile 1 erzeugt eine Variable namens „value“, die ein einzelnes Zeichen (Char) speichert und sich bsw. an Adresse 8000 befindet. In Zeile 2 definieren wir mit Hilfe des \*-Zeichens einen Pointer vom Typ char, welchem wir mit dem Referenzoperator (&) die Adresse der Variable „value“ zuordnen.

Um nun auf den Wert zuzugreifen, auf den unser Pointer p2 zeigt, nutzen wir den Dereferenzierungsoperator (\*):

```
1 *p // Wert des Ziels des Pointers p: 'c'
2 p // Wert des Pointers p: 0x8000
3 &p // Adresse des Pointers p: 0x5000
```



#### Weitere Beispiele:

Dereferenzieren vom Pointer ermöglicht uns auf i zuzugreifen. \*ip leitet uns zur Variable i weiter.

```
1 int i;
2 /* Define a Pointer for Variable i */
3 int *ip;
4 i = 5;
5 /* Set ip Value to Address of i */
6 ip = &i;
7 /* Dereference of *ip = Acces to Variable i/ Systemaddress of i,
8 set value of i */
9 *ip = 7;
```

Der Pointer jp soll nun auf i zeigen. Wie kann das realisiert werden?



```
1 int j ,
2 int *jp ,
3 int i ,
4 int ip ,
5 ip = &i;
6
7 /* Solution */
8 jp = ip
```

Wir haben zwei Variablen und wollen deren Wert tauschen /swappen.

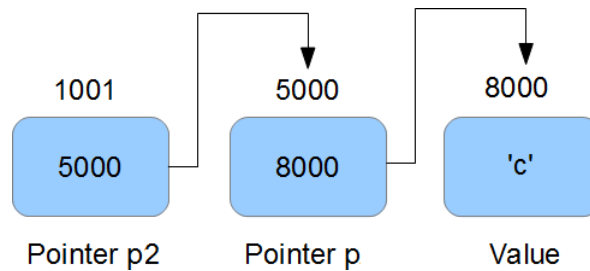
```
1 int a = 5;
2 int b = 7;
3 /* Target a = 7, b = 5 */
4
5 /*Solution:
6 We define a function for swapping, we use pointers as parameter*/
7 swap(int *ap, int *bp)
8 {
9 int dummy = *ap;
10 *ap = *bp;
11 *bp = dummy;
12 }
13
14 /* We call the function this way */
15 swap(&a,&b);
```

## 6.2.2 Doppeltes Dereferenzieren

Es ist ausserdem möglich, Zeiger auf Zeiger zu erstellen.

Doppeltes Auflösen des Pointers. Pointer beinhaltet Pointer.

```
1 char value = 'c';
2 char *p = &value;
3 char **p2 = &p;
```



```
1 *p2 // Wert auf den p2 zeigt: Die Speicheradresse 8000
2 **p2 // Wert auf den p1 zeigt, auf welchen p2 zeigt: 'c'
3 p2 // Wert des Pointers p2: Die Speicheradresse 5000
4 &p // Adresse des Pointers p: Die Speicheradresse 1001
```

### weiteres Beispiel

```
1 int i;
2 int *ip;
3 ip = &i;
4 /* Define a double Pointer */
5 int **ipp;
6 /* Set ipp value to Address of ip */
7 ipp = &ip;
8 /* Set value of i = 9 */
9 **ipp = 9;
10 /* **ipp -> *ip -> i/
```

## 6.3 Nutzen von Pointern

In C geschieht die Werteübergabe an Subroutinen standardmässig nach dem „**Call By Value**“ Prinzip. Es gibt in C (erst in C++) kein „**Call By Reference**“ wie in anderen Sprachen. Bsp:

```
1 void doubleValueOf(int i){
2 i = 2*i;
3 }
4
5 void main(){
6 int zahl = 5;
7 doubleValueOf(zahl);
8 }
```

Wenn wir die Funktion „doubleValueOf“ aufrufen, wird der Wert der Variable „zahl“, auf den Stack gelegt und innerhalb der Funktion darauf zugegriffen. Die ursprüngliche Variable „zahl“ liegt jedoch an einer Stelle im Speicher, die der Funktion nicht bekannt ist. Es wird somit der Wert einer Kopie der Variable „zahl“ verdoppelt, die nach Ende der Funktion verloren geht. Zurück in unserem Hauptprogramm, wird die Variable „zahl“ immer noch 5 als Wert haben.

Pointer ermöglichen nun, statt einer Kopie einen Referenz auf die richtige Variable „zahl“ zu übergeben.

```
1 void doubleValueOf(int *pi){
2 *pi = 2>(*pi); // Der Wert der Adresse, auf die pi zeigt, wird auf 2*
 // den Wert von i (*i) gesetzt.
3 }
4
5 void main(){
6 int zahl = 5;
7 doubleValueOf(&zahl);
8 }
```

## 6.4 Arrays

Arrays sind Felder/Reihungen von Variablen eines bestimmten Datentypes. Beispiel:

```
1 int ar[5] = {1, 2, 3, 4, 5}
```

Erstellt 5 aufeinanderfolgende Integer Variablen mit den Werten 1,2,3,4 und 5. Wir können direkt auf ein bestimmten Element zugreifen:

```
1 ar[2] = 7;
```

Wenn ein Integer 4 Byte beansprucht, ist die Grösse des gesamten Arrays  $5 \cdot 4 \text{ Byte} = 20 \text{ Byte}$ .

C besitzt standardmässig keinen Datentyp „String“, um Zeichenfolgen darzustellen. Stattdessen können wir ein Array von Chars benutzen:

```
1 char str[6] = { 'H', 'a', 'l', 'l', 'o', '\0' };
```

Das Zeichen '\0' wird benötigt, um das Ende eines Strings zu markieren. Wir können auch

```
1 char str[] = "Hallo";
```

schreiben. Dies erstellt ein Array mit 6 Char Elementen, 5 für „Hallo“ und ein weiteres für '\0'. Eine weitere Möglichkeit wäre es, einen Zeiger auf ein Char Array zu erstellen:

```
1 char *str = "Hallo";
```

Dabei wird sowohl ein (verstecktes) Char Array mit dem Inhalt {'H','a','l','l','o'} und einen Char Pointer namens str erstellt, der auf das erste Element des versteckten Arrays zeigt. str zeigt nun also auf das erste Char-Element, den Buchstaben 'H':

```
1 *str // 'H', oder auch str[0]
2 *(str+1) // 'a', oder auch str[1]
3 *(str+5) // 'a'
```

Achtung: Es ist undefiniert was passiert, wenn nun \*str ein neuer Wert zugeordnet wird!

## 6.5 Dynamische Speicherallokation/ Dynamic Memory Allocation

Lokale und globale Variablen werden normalerweise statisch, d.h. beim Starten des Programms reserviert. Manchmal ist es notwendig, Speicher dynamisch, zur Laufzeit, anzufordern, beispielsweise weil man auf Benutzereingaben reagieren will oder weil man eine Variable einer bestimmten Grösse benötigt.

Die Programmiersprache C bietet dafür einige Funktionen zur Verfügung.

**Definition 1.** *Statisch bedeutet, dass der benötigte Speicher zu Beginn des Programms reserviert wird.*

Ein Beispiel:

```
1 int a;
2 int *ip;
3 int array[10];
```

**Definition 2.** *Dynamisch bedeutet, dass der benötigte Speicher zur Laufzeit des Programms reserviert wird.*

Um den Speicher zur Laufzeit zu reservieren schreiben wir:

```
1 int *ip;
2 ip = malloc(sizeof(int)*10);
```

Damit reservieren wir den Platz für 10 int Werte, also ein Array der Grösse 10. Ein int ist 32Bit gross, dementsprechend werden 320Bit oder 40Byte reserviert.

Der Befehl malloc (Memory allocation) gibt die Speicheradresse des ersten int Blocks zurück. Wir speichern diesen in einem Pointer.

Aufgabe:

Wert 3 an erste Speicheradresse im Array schreiben

```
1 *ip = 3;
```

\*ip wird dereferenziert wir erhalten Zugriff zur ersten Speicheradresse

Aufgabe2:

Wert 2 an zweite Speicheradresse im Array schreiben

```
1 *(ip+1) = 2;
```

Dadurch dass wir int \*ip schreiben, weiss der Compiler wie weit er mit \*ip+1 springen muss. Da es sich um int Werte handelt, rechnet er zur ersten Speicheradresse 4 Byte hinzu. \*ip+2 rechnet er 8Byte hinzu. usw.

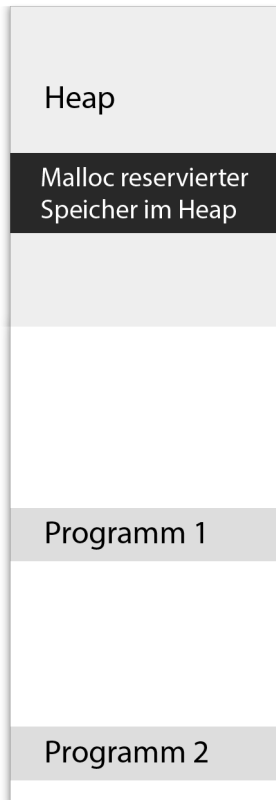
Achtung!

```
1 *(ip+100) = 1
```

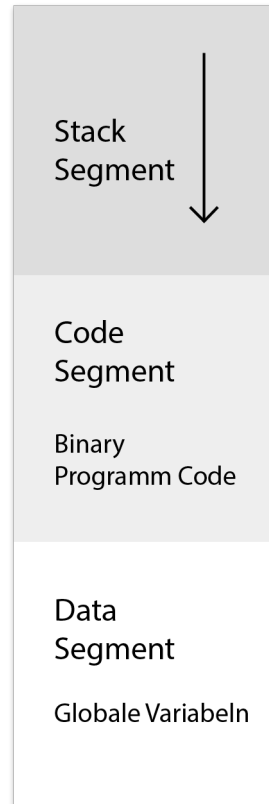
Ist nicht möglich, schliesslich haben wir nur 10 int Blöcke reserviert. Die Adresse liegt ausserhalb des Speicherbereichs.

## 6.6 Speichersegmente

RAM



Programm 1



Der Heap Bereich ist der freie Speicher

## 6.7 Listen

Eine unbekannte Anzahl Elemente verwalten.

```
1 struct element
2 {
3 int data; // The Data of the element
4 struct element *next; // the next element
5 }
```

Wir definieren das "Objekt" element, es besitzt einen Wert und einen Pointer auf das nächste element Objekt.

Um eine Verlinkte Liste von Elementen mit den werten 1,2,3... zu erstellen, gehen wir wie folgt vor:

```
1 struct element
2 {
3 int data; // The Data of the element
4 struct element *next; // the next element
5 }
6
7 struct element *head;
8 struct element *np; // next Pointer
9
10 /* 1 */
11 // reservate space for the first element
12 head = malloc(sizeof (struct element));
13
14 /* 2*/
15 (*head.data) = 1; // Set data
16
17 /* 3 */
18 // reservate space for the next Element
19 np = malloc(sizeof(struct node));
20
21 /* 4 */
22 (*head).next = np;
23
24 /* 5 */
25 // Set data
26 ((*head).next).data = 2;
27
28 /* 6 */
29 // Shorter Version
30 np->next = malloc(sizeof(struct node));
31 np->next->data = 3;
```

Kurze Version:

```
1 struct element
2 {
3 int data; // The Data of the element
4 struct element *next; // the next element
5 }
6 struct element *np; // next Element Pointer
7
8 /* 1. Element */
9 np = malloc(sizeof(struct element));
10 np->data = 1;
11 np->next = malloc(sizeof(struct element));
12
13 /* 2. Element */
14 np = np->next;
15 np->data = 2;
16 np->next = malloc(sizeof(struct element));
17
18 //...
```



## 6.8 Operator Precedence

### 6.8.1 Beispiele

| C-Code                              | Aussage                                                                    |
|-------------------------------------|----------------------------------------------------------------------------|
| <code>int *x(char)</code>           | x as function (char) returning pointer to int                              |
| <code>int (*x)(char)</code>         | x as pointer to function (char) returning int                              |
| <code>int *(*x[5])</code>           | dx as array 5 of pointer to pointer to int                                 |
| <code>int *x(int, char *)</code>    | x as function (int, pointer to char) returning pointer to int              |
| <code>int *(*x)(int, char *)</code> | x as pointer to function (int, pointer to char) returning pointer to int   |
| <code>int (**x)(int, char *)</code> | x as pointer to pointer to function (int, pointer to char) returning int   |
| <code>int **x(int *)</code>         | declare x as function (pointer to int) returning pointer to pointer to int |
| <code>int (*x[5])(int)</code>       | declare x as array 5 of pointer to function (int) returning int            |

## Kapitel 7

## Tables

# C Operator Precedence Table

This page lists C operators in order of *precedence* (highest to lowest). Their *associativity* indicates in what order operators of equal precedence in an expression are applied.

| Operator                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | Description                                                                                                                                                                                                                               | Associativity |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|
| ( )<br>[ ]<br>.<br>-><br>++ --                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | Parentheses (function call) (see Note 1)<br>Brackets (array subscript)<br>Member selection via object name<br>Member selection via pointer<br>Postfix increment/decrement (see Note 2)                                                    | left-to-right |
| ++ --<br>+ -<br>! ~<br>(type)<br>*<br>&<br>sizeof                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | Prefix increment/decrement<br>Unary plus/minus<br>Logical negation/bitwise complement<br>Cast (convert value to temporary value of <i>type</i> )<br>Dereference<br>Address (of operand)<br>Determine size in bytes on this implementation | right-to-left |
| * / %                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | Multiplication/division/modulus                                                                                                                                                                                                           | left-to-right |
| + -                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | Addition/subtraction                                                                                                                                                                                                                      | left-to-right |
| << >>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | Bitwise shift left, Bitwise shift right                                                                                                                                                                                                   | left-to-right |
| < <=<br>> >=                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | Relational less than/less than or equal to<br>Relational greater than/greater than or equal to                                                                                                                                            | left-to-right |
| == !=                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | Relational is equal to/is not equal to                                                                                                                                                                                                    | left-to-right |
| &                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | Bitwise AND                                                                                                                                                                                                                               | left-to-right |
| ^                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | Bitwise exclusive OR                                                                                                                                                                                                                      | left-to-right |
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                | Bitwise inclusive OR                                                                                                                                                                                                                      | left-to-right |
| &&                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | Logical AND                                                                                                                                                                                                                               | left-to-right |
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                | Logical OR                                                                                                                                                                                                                                | left-to-right |
| ? :                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | Ternary conditional                                                                                                                                                                                                                       | right-to-left |
| =<br>+= -=<br>*= /=<br>%= &=<br>^=  =<br><<= >>=                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               | Assignment<br>Addition/subtraction assignment<br>Multiplication/division assignment<br>Modulus/bitwise AND assignment<br>Bitwise exclusive/inclusive OR assignment<br>Bitwise shift left/right assignment                                 | right-to-left |
| ,                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | Comma (separate expressions)                                                                                                                                                                                                              | left-to-right |
| <p><b>Note 1:</b><br/>Parentheses are also used to group sub-expressions to force a different precedence; such parenthetical expressions can be nested and are evaluated from inner to outer.</p> <p><b>Note 2:</b><br/>Postfix increment/decrement have high precedence, but the actual increment or decrement of the operand is delayed (to be accomplished sometime before the statement completes execution). So in the statement <code>y = x * z++</code>; the current value of <code>z</code> is used to evaluate the expression (i.e., <code>z++</code> evaluates to <code>z</code>) and <code>z</code> only incremented after all else is done. See <a href="#">postinc.c</a> for another example.</p> |                                                                                                                                                                                                                                           |               |

## Instruction Set Summary

| Instr                    | Addressing Mode | Assembler Format | Operation                                    | Op Code Hex | Dec | Bytes | Clock Cycles | Status Register - P                     | Instr                                                |
|--------------------------|-----------------|------------------|----------------------------------------------|-------------|-----|-------|--------------|-----------------------------------------|------------------------------------------------------|
| ADC                      | Immediate       | ADC #oper        | A + # + C → .A, C                            | 69          | 105 | 2     | 2            | N V D I Z C                             | ADC                                                  |
|                          | Zero Page       | ADC addr         | A + [addr] + C → .A, C                       | 65          | 101 | 2     | 3            | ✓ ✓ - - - ✓ ✓                           |                                                      |
|                          | Zero Page, X    | ADC addr, X      | A + [addr + .X] + C → .A, C                  | 75          | 117 | 2     | 4            |                                         |                                                      |
|                          | Absolute        | ADC ADDR         | A + [ADDR] + C → .A, C                       | 6D          | 109 | 3     | 4            |                                         |                                                      |
|                          | Absolute, X     | ADC ADDR, X      | A + [ADDR + .X] + C → .A, C                  | 7D          | 125 | 3     | 4*           |                                         |                                                      |
|                          | Absolute, Y     | ADC ADDR, Y      | A + [ADDR + .Y] + C → .A, C                  | 79          | 121 | 3     | 4*           |                                         |                                                      |
|                          | (Indirect, X)   | ADC (addr, X)    | A + [[addr + .X + 1, addr + .X]] + C → .A, C | 61          | 97  | 2     | 6            |                                         |                                                      |
|                          | (Indirect),Y    | ADC (addr),Y     | A + [[addr + 1, addr] + .Y] + C → .A, C      | 71          | 113 | 2     | 5*           |                                         |                                                      |
| AND                      | Immediate       | AND #oper        | A ∧ # → .A                                   | 29          | 47  | 2     | 2            | N V D I Z C                             | AND                                                  |
|                          | Zero Page       | AND addr         | A ∧ [addr] → .A                              | 25          | 37  | 2     | 3            | ✓ - - - - ✓ -                           |                                                      |
|                          | Zero Page, X    | AND addr, X      | A ∧ [addr + .X] → .A                         | 35          | 53  | 2     | 4            |                                         |                                                      |
|                          | Absolute        | AND ADDR         | A ∧ [ADDR] → .A                              | 2D          | 45  | 3     | 4            |                                         |                                                      |
|                          | Absolute, X     | AND ADDR, X      | A ∧ [ADDR + .X] → .A                         | 3D          | 61  | 3     | 4*           |                                         |                                                      |
|                          | Absolute, Y     | AND ADDR, Y      | A ∧ [ADDR + .Y] → .A                         | 39          | 57  | 3     | 4*           |                                         |                                                      |
|                          | (Indirect, X)   | AND (addr, X)    | A ∧ [[addr + .X + 1, addr + .X]] → .A        | 21          | 33  | 2     | 6            |                                         |                                                      |
|                          | (Indirect),Y    | AND (addr),Y     | A ∧ [[addr + 1, addr] + .Y] → .A             | 31          | 49  | 2     | 5*           |                                         |                                                      |
| ASL                      | Accumulator     | ASL A            | A (←) → A : 0 → bit 0, bit 7 → C             | 0A          | 10  | 1     | 2            | N V D I Z C                             | ASL                                                  |
|                          | Zero Page       | ASL addr         | [addr] (←) → [addr]                          | 06          | 6   | 2     | 5            | ✓ - - - - ✓ ✓                           |                                                      |
|                          | Zero Page, X    | ASL addr, X      | [addr + .X] (←) → [addr + .X]                | 16          | 22  | 2     | 6            |                                         |                                                      |
|                          | Absolute        | ASL ADDR         | [ADDR] (←) → [ADDR]                          | 0E          | 14  | 3     | 6            |                                         |                                                      |
|                          | Absolute, X     | ASL ADDR, X      | [ADDR + .X] (←) → [ADDR + .X]                | 1E          | 30  | 3     | 7            |                                         |                                                      |
| BCC                      | Relative        | BCC oper         | Branch on C = 0                              | 90          | 144 | 2     | 2*           | N V D I Z C                             | BCC<br>BCS<br>BEQ<br>BNE<br>BMI<br>BPL<br>BVS<br>BVC |
| BCS                      | Relative        | BCS oper         | Branch on C = 1                              | B0          | 176 | 2     | 2*           | - - - - -                               |                                                      |
| BEQ                      | Relative        | BEQ oper         | Branch on Z = 1                              | F0          | 240 | 2     | 2*           |                                         |                                                      |
| BNE                      | Relative        | BNE oper         | Branch on Z = 0                              | D0          | 208 | 2     | 2*           | All Branches                            |                                                      |
| BMI                      | Relative        | BMI oper         | Branch on N = 1                              | 30          | 48  | 2     | 2*           | * - Add 1 if branch to same page        |                                                      |
| BPL                      | Relative        | BPL oper         | Branch on N = 0                              | 10          | 16  | 2     | 2*           | * - Add 2 if branch to diff page        |                                                      |
| BVS                      | Relative        | BVS oper         | Branch on V = 1                              | 70          | 112 | 2     | 2*           |                                         |                                                      |
| BVC                      | Relative        | BVC oper         | Branch on V = 0                              | 50          | 80  | 2     | 2*           |                                         |                                                      |
| BIT                      | Zero Page       | BIT addr         | A ∧ [addr] : bit 7 → N, bit 6 → V            | 24          | 36  | 2     | 3            | N V D I Z C                             | BIT                                                  |
|                          | Absolute        | BIT ADDR         | A ∧ [ADDR]                                   | 2C          | 44  | 3     | 4            | b <sub>7</sub> b <sub>6</sub> - - - ✓ - |                                                      |
| BRK                      | Implied         | BRK 1→B flag     | PC + 2 ↓ P ↓, [FFFE]→PCL, [FFFF]→PCH         | 00          | 0   | 1     | 7            | - - - - 1 - - -                         | BRK                                                  |
| CLC<br>CLD<br>CLI<br>CLV | Implied         | CLC              | 0→C                                          | 18          | 24  | 1     | 2            | N V D I Z C                             | CLC<br>CLD<br>CLI<br>CLV                             |
|                          | Implied         | CLD              | 0→D                                          | D8          | 216 | 1     | 2            | - - - - 0 - - -                         |                                                      |
|                          | Implied         | CLI              | 0→I                                          | 58          | 88  | 1     | 2            | - - - - 0 - - -                         |                                                      |
|                          | Implied         | CLV              | 0→V                                          | B8          | 184 | 1     | 2            | - 0 - - - - -                           |                                                      |
| CMP                      | Immediate       | CMP #oper        | A - #                                        | C9          | 201 | 2     | 2            | N V D I Z C                             | CMP                                                  |
|                          | Zero Page       | CMP addr         | A - [addr]                                   | C5          | 197 | 2     | 3            | ✓ - - - - ✓ ✓                           |                                                      |
|                          | Zero Page, X    | CMP addr, X      | A - [addr + .X]                              | D5          | 213 | 2     | 4            |                                         |                                                      |
|                          | Absolute        | CMP ADDR         | A - [ADDR]                                   | CD          | 205 | 3     | 4            |                                         |                                                      |
|                          | Absolute, X     | CMP ADDR, X      | A - [ADDR + .X]                              | DD          | 221 | 3     | 4*           |                                         |                                                      |
|                          | Absolute, Y     | CMP ADDR, Y      | A - [ADDR + .Y]                              | D9          | 217 | 3     | 4*           |                                         |                                                      |
|                          | (Indirect, X)   | CMP (addr, X)    | A - [[addr + .X + 1, addr + .X]]             | C1          | 193 | 2     | 6            |                                         |                                                      |
|                          | (Indirect),Y    | CMP (addr),Y     | A - [[addr + 1, addr] + .Y]                  | D1          | 209 | 2     | 5*           |                                         |                                                      |
| CPX                      | Immediate       | CPX #oper        | X - #                                        | E0          | 224 | 2     | 2            | N V D I Z C                             | CPX                                                  |
|                          | Zero Page       | CPX addr         | X - [addr]                                   | E4          | 228 | 2     | 3            | ✓ - - - - ✓ ✓                           |                                                      |
|                          | Absolute        | CPX ADDR         | X - [ADDR]                                   | EC          | 236 | 3     | 4            |                                         |                                                      |
| CPY                      | Immediate       | CPY #oper        | Y - #                                        | C0          | 192 | 2     | 2            | N V D I Z C                             | CPY                                                  |
|                          | Zero Page       | CPY addr         | Y - [addr]                                   | C4          | 196 | 2     | 3            | ✓ - - - - ✓ ✓                           |                                                      |
|                          | Absolute        | CPY ADDR         | Y - [ADDR]                                   | CC          | 204 | 3     | 4            |                                         |                                                      |
| DEC                      | Zero Page       | DEC addr         | [addr] - 1 → [addr]                          | C6          | 198 | 2     | 5            | N V D I Z C                             | DEC                                                  |
|                          | Zero Page, X    | DEC addr, X      | [addr + .X] - 1 → [addr + .X]                | D6          | 214 | 2     | 6            | ✓ - - - - ✓ -                           |                                                      |
|                          | Absolute        | DEC ADDR         | [ADDR] - 1 → [ADDR]                          | CE          | 206 | 3     | 6            |                                         |                                                      |
|                          | Absolute, X     | DEC ADDR, X      | [ADDR + .X] - 1 → [ADDR + .X]                | DE          | 222 | 3     | 7            |                                         |                                                      |
| DEX<br>DEY               | Implied         | DEX              | X - 1 → X                                    | CA          | 202 | 1     | 2            | N V D I Z C                             | DEX<br>DEY                                           |
|                          | Implied         | DEY              | Y - 1 → Y                                    | 88          | 136 | 1     | 2            | ✓ - - - - ✓ -                           |                                                      |
| EOR                      | Immediate       | EOR #oper        | A ⊕ # → A                                    | 49          | 73  | 2     | 2            | N V D I Z C                             | EOR                                                  |
|                          | Zero Page       | EOR addr         | A ⊕ [addr] → A                               | 45          | 69  | 2     | 3            | ✓ - - - - ✓ -                           |                                                      |
|                          | Zero Page, X    | EOR addr, X      | A ⊕ [addr + .X] → A                          | 55          | 85  | 2     | 4            |                                         |                                                      |
|                          | Absolute        | EOR ADDR         | A ⊕ [ADDR] → A                               | 4D          | 77  | 3     | 4            |                                         |                                                      |
|                          | Absolute, X     | EOR ADDR, X      | A ⊕ [ADDR + .X] → A                          | 5D          | 93  | 3     | 4*           |                                         |                                                      |
|                          | Absolute, Y     | EOR ADDR, Y      | A ⊕ [ADDR + .Y] → A                          | 59          | 89  | 3     | 4*           |                                         |                                                      |
|                          | (Indirect, X)   | EOR (addr, X)    | A ⊕ [[addr + .X + 1, addr + .X]] → A         | 41          | 65  | 2     | 6            |                                         |                                                      |
|                          | (Indirect),Y    | EOR (addr),Y     | A ⊕ [[addr + 1, addr] + .Y] → A              | 51          | 81  | 2     | 5*           |                                         |                                                      |
| INC                      | Zero Page       | INC addr         | [addr] + 1 → [addr]                          | E6          | 230 | 2     | 5            | N V D I Z C                             | INC                                                  |
|                          | Zero Page, X    | INC addr, X      | [addr + .X] + 1 → [addr + .X]                | F6          | 246 | 2     | 6            | ✓ - - - - ✓ -                           |                                                      |
|                          | Absolute        | INC ADDR         | [ADDR] + 1 → [ADDR]                          | EE          | 238 | 3     | 6            |                                         |                                                      |
|                          | Absolute, X     | INC ADDR, X      | [ADDR + .X] + 1 → [ADDR + .X]                | FE          | 254 | 3     | 7            |                                         |                                                      |
| INX<br>INY               | Implied         | INX              | X + 1 → X                                    | E8          | 232 | 1     | 2            | N V D I Z C                             | INX<br>INY                                           |
|                          | Implied         | INY              | Y + 1 → Y                                    | C8          | 200 | 1     | 2            | ✓ - - - - ✓ -                           |                                                      |
| JMP                      | Absolute        | JMP ADDR         | [PC + 1] → PCL, [PC + 2] → PCH               | 4C          | 76  | 3     | 3            | N V D I Z C                             | JMP                                                  |
|                          | Indirect        | JMP (ADDR)       | [ADDR] → PCL, [ADDR + 1] → PCH               | 6C          | 108 | 3     | 5            | - - - - -                               |                                                      |
| JSR                      | Absolute        | JSR ADDR         | PC + 2 ↓, [PC + 1] → PCL, [PC + 2] → PCH     | 20          | 32  | 3     | 6            |                                         | JSR                                                  |

| Instr                                              | Addressing Mode | Assembler Format | Operation                                   | Op Code Hex | Dec | Bytes | Clock Cycles | Status Register - P                | Instr                                              |
|----------------------------------------------------|-----------------|------------------|---------------------------------------------|-------------|-----|-------|--------------|------------------------------------|----------------------------------------------------|
| <b>LDA</b>                                         | Immediate       | LDA #oper        | # → A                                       | A9          | 169 | 2     | 2            | N V D I Z C                        | <b>LDA</b>                                         |
|                                                    | Zero Page       | LDA addr         | [addr] → A                                  | A5          | 165 | 2     | 3            | ✓ - - - ✓ -                        |                                                    |
|                                                    | Zero Page, X    | LDA addr, X      | [addr + X] → A                              | B5          | 181 | 2     | 4            |                                    |                                                    |
|                                                    | Absolute        | LDA ADDR         | [ADDR] → A                                  | AD          | 173 | 3     | 4            |                                    |                                                    |
|                                                    | Absolute, X     | LDA ADDR, X      | [ADDR + X] → A                              | BD          | 189 | 3     | 4*           |                                    |                                                    |
|                                                    | Absolute, Y     | LDA ADDR, Y      | [ADDR + Y] → A                              | B9          | 185 | 3     | 4*           |                                    |                                                    |
|                                                    | (Indirect, X)   | LDA (addr, X)    | [[addr + X + 1, addr + X]] → A              | A1          | 161 | 2     | 6            |                                    |                                                    |
|                                                    | (Indirect), Y   | LDA (addr), Y    | [[addr + 1, addr] + Y] → A                  | B1          | 177 | 2     | 5*           |                                    |                                                    |
| <b>LDX</b>                                         | Immediate       | LDX #oper        | # → X                                       | A2          | 162 | 2     | 2            | N V D I Z C                        | <b>LDX</b>                                         |
|                                                    | Zero Page       | LDX addr         | [addr] → X                                  | A6          | 166 | 2     | 3            | ✓ - - - ✓ -                        |                                                    |
|                                                    | Zero Page, Y    | LDX addr, Y      | [addr + Y] → X                              | B6          | 182 | 2     | 4            |                                    |                                                    |
|                                                    | Absolute        | LDX ADDR         | [ADDR] → X                                  | AE          | 174 | 3     | 4            |                                    |                                                    |
|                                                    | Absolute, Y     | LDX ADDR, Y      | [ADDR + Y] → X                              | BE          | 190 | 3     | 4*           |                                    |                                                    |
| <b>LDY</b>                                         | Immediate       | LDY #oper        | # → Y                                       | A0          | 160 | 2     | 2            | N V D I Z C                        | <b>LDY</b>                                         |
|                                                    | Zero Page       | LDY addr         | [addr] → Y                                  | A4          | 164 | 2     | 3            | ✓ - - - ✓ -                        |                                                    |
|                                                    | Zero Page, X    | LDY addr, X      | [addr + X] → Y                              | B4          | 180 | 2     | 4            |                                    |                                                    |
|                                                    | Absolute        | LDY ADDR         | [ADDR] → Y                                  | AC          | 172 | 3     | 4            |                                    |                                                    |
|                                                    | Absolute, X     | LDY ADDR, X      | [ADDR + X] → Y                              | BC          | 188 | 3     | 4*           |                                    |                                                    |
| <b>LSR</b>                                         | Accumulator     | LSR A            | A (←) → A ; 0 → bit7, bit0 → C              | 4A          | 74  | 1     | 2            | N V D I Z C                        | <b>LSR</b>                                         |
|                                                    | Zero Page       | LSR addr         | [addr] (←) → [addr]                         | 46          | 70  | 2     | 5            | 0 - - - ✓ ✓                        |                                                    |
|                                                    | Zero Page, X    | LSR addr, Y      | [addr + X] (←) → [addr + X]                 | 56          | 86  | 2     | 6            |                                    |                                                    |
|                                                    | Absolute        | LSR ADDR         | [ADDR] (←) → [ADDR]                         | 4E          | 78  | 3     | 6            |                                    |                                                    |
|                                                    | Absolute, X     | LSR ADDR, X      | [ADDR + X] (←) → [ADDR + X]                 | 5E          | 94  | 3     | 7            |                                    |                                                    |
| <b>NOP</b>                                         | Implied         | NOP              | No Operation                                | EA          | 234 | 1     | 2            | - - - - -                          | <b>NOP</b>                                         |
| <b>ORA</b>                                         | Immediate       | ORA #oper        | A U # → A                                   | 09          | 9   | 2     | 2            | N V D I Z C                        | <b>ORA</b>                                         |
|                                                    | Zero Page       | ORA addr         | A U [addr] → A                              | 05          | 5   | 2     | 3            | ✓ - - - ✓ -                        |                                                    |
|                                                    | Zero Page, X    | ORA addr, X      | A U [addr + X] → A                          | 15          | 21  | 2     | 4            |                                    |                                                    |
|                                                    | Absolute        | ORA ADDR         | A U [ADDR] → A                              | 0D          | 13  | 3     | 4            |                                    |                                                    |
|                                                    | Absolute, X     | ORA ADDR, X      | A U [ADDR + X] → A                          | 1D          | 29  | 3     | 4*           |                                    |                                                    |
|                                                    | Absolute, Y     | ORA ADDR, Y      | A U [ADDR + Y] → A                          | 19          | 25  | 3     | 4*           |                                    |                                                    |
|                                                    | (Indirect, X)   | ORA (addr, X)    | A U [[addr + X + 1, addr + X]] → A          | 01          | 1   | 2     | 6            |                                    |                                                    |
|                                                    | (Indirect), Y   | ORA (addr), Y    | A U [[addr + 1, addr] + Y] → A              | 11          | 17  | 2     | 5*           |                                    |                                                    |
| <b>PHA<br/>PLA<br/>PHP<br/>PLP</b>                 | Implied         | PHA              | A ↑, SP - 1 → SP                            | 48          | 72  | 1     | 3            | N V D I Z C                        | <b>PHA<br/>PLA<br/>PHP<br/>PLP</b>                 |
|                                                    | Implied         | PLA              | A ↑, SP + 1 → SP                            | 68          | 104 | 1     | 4            | - - - - -                          |                                                    |
|                                                    | Implied         | PHP              | P ↑, SP - 1 → SP                            | 08          | 8   | 1     | 3            | - - - - -                          |                                                    |
|                                                    | Implied         | PLP              | P ↑, SP + 1 → SP                            | 28          | 40  | 1     | 4            | All Push/Pulls xcpt P/P from stack |                                                    |
| <b>ROL</b>                                         | Accumulator     | ROL A            | A (←) → A ; C → bit0, bit7 → C              | 2A          | 42  | 1     | 2            | N V D I Z C                        | <b>ROL</b>                                         |
|                                                    | Zero Page       | ROL addr         | [addr] (←) → [addr]                         | 26          | 38  | 2     | 5            | ✓ - - - ✓ ✓                        |                                                    |
|                                                    | Zero Page, X    | ROL addr, X      | [addr + X] (←) → [addr + X]                 | 36          | 54  | 2     | 6            |                                    |                                                    |
|                                                    | Absolute        | ROL ADDR         | [ADDR] (←) → [ADDR]                         | 2E          | 46  | 3     | 6            |                                    |                                                    |
|                                                    | Absolute, X     | ROL ADDR, X      | [ADDR + X] (←) → [ADDR + X]                 | 3E          | 62  | 3     | 7            |                                    |                                                    |
| <b>ROR</b>                                         | Accumulator     | ROR A            | A (←) → A ; C → bit7, bit0 → C              | 6A          | 106 | 1     | 2            | N V D I Z C                        | <b>ROR</b>                                         |
|                                                    | Zero Page       | ROR addr         | [addr] (←) → [addr]                         | 66          | 102 | 2     | 5            | ✓ - - - ✓ ✓                        |                                                    |
|                                                    | Zero Page, X    | ROR addr, Y      | [addr + X] (←) → [addr + X]                 | 76          | 118 | 2     | 6            |                                    |                                                    |
|                                                    | Absolute        | ROR ADDR         | [ADDR] (←) → [ADDR]                         | 6E          | 110 | 3     | 6            |                                    |                                                    |
|                                                    | Absolute, X     | ROR ADDR, X      | [ADDR + X] (←) → [ADDR + X]                 | 7E          | 126 | 3     | 7            |                                    |                                                    |
| <b>RTI</b>                                         | Implied         | RTI              | P ↑, PC ↑, SP + 3 → SP, PC + 1 → PC         | 40          | 64  | 1     | 6            | from stack                         | <b>RTI</b>                                         |
| <b>RTS</b>                                         | Implied         | RTS              | PC ↑, SP + 2 → SP, PC + 1 → PC              | 60          | 96  | 1     | 6            | - - - - -                          | <b>RTS</b>                                         |
| <b>SBC</b>                                         | Immediate       | SBC #oper        | A - # - C̄ → A, C̄ = Borrow                 | E9          | 233 | 2     | 2            | N V D I Z C                        | <b>SBC</b>                                         |
|                                                    | Zero Page       | SBC addr         | A - [addr] - C̄ → A, C̄                     | E5          | 229 | 2     | 3            | ✓ ✓ - - ✓ ✓                        |                                                    |
|                                                    | Zero Page, X    | SBC addr, X      | A - [addr + X] - C̄ → A, C̄                 | F5          | 245 | 2     | 4            |                                    |                                                    |
|                                                    | Absolute        | SBC ADDR         | A - [ADDR] - C̄ → A, C̄                     | ED          | 237 | 3     | 4            |                                    |                                                    |
|                                                    | Absolute, X     | SBC ADDR, X      | A - [ADDR + X] - C̄ → A, C̄                 | FD          | 253 | 3     | 4*           |                                    |                                                    |
|                                                    | Absolute, Y     | SBC ADDR, Y      | A - [ADDR + Y] - C̄ → A, C̄                 | F9          | 249 | 3     | 4*           |                                    |                                                    |
|                                                    | (Indirect, X)   | SBC (addr, X)    | A - [[addr + X + 1, addr + X]] - C̄ → A, C̄ | E1          | 225 | 2     | 6            |                                    |                                                    |
|                                                    | (Indirect), Y   | SBC (addr), Y    | A - [[addr + 1, addr] + Y] - C̄ → A, C̄     | F1          | 241 | 2     | 5*           |                                    |                                                    |
| <b>SEC<br/>SED<br/>SEI</b>                         | Implied         | SEC              | 1 → C                                       | 38          | 56  | 1     | 2            | N V D I Z C                        | <b>SEC<br/>SED<br/>SEI</b>                         |
|                                                    | Implied         | SED              | 1 → D                                       | F8          | 248 | 1     | 2            | - - - 1 - -                        |                                                    |
|                                                    | Implied         | SEI              | 1 → I                                       | 78          | 120 | 1     | 2            | - - - 1 - -                        |                                                    |
| <b>STA</b>                                         | Zero Page       | STA addr         | A → [addr]                                  | 85          | 133 | 2     | 3            | N V D I Z C                        | <b>STA</b>                                         |
|                                                    | Zero Page, X    | STA addr, X      | A → [addr + X]                              | 95          | 149 | 2     | 4            | - - - - -                          |                                                    |
|                                                    | Absolute        | STA ADDR         | A → [ADDR]                                  | 8D          | 141 | 3     | 4            |                                    |                                                    |
|                                                    | Absolute, X     | STA ADDR, X      | A → [ADDR + X]                              | 9D          | 157 | 3     | 5            |                                    |                                                    |
|                                                    | Absolute, Y     | STA ADDR, Y      | A → [ADDR + Y]                              | 99          | 153 | 3     | 5            |                                    |                                                    |
|                                                    | (Indirect, X)   | STA (addr, X)    | A → [[addr + X + 1, addr + X]]              | 81          | 129 | 2     | 6            |                                    |                                                    |
|                                                    | (Indirect), Y   | STA (addr), Y    | A → [[addr + 1, addr] + Y]                  | 91          | 145 | 2     | 6            |                                    |                                                    |
| <b>STX</b>                                         | Zero Page       | STX addr         | X → [addr]                                  | 86          | 134 | 2     | 3            | N V D I Z C                        | <b>STX</b>                                         |
|                                                    | Zero Page, Y    | STX addr, Y      | X → [addr + Y]                              | 96          | 150 | 2     | 4            | - - - - -                          |                                                    |
|                                                    | Absolute        | STX ADDR         | X → [ADDR]                                  | 8E          | 142 | 3     | 4            |                                    |                                                    |
| <b>STY</b>                                         | Zero Page       | STY addr         | Y → [addr]                                  | 84          | 132 | 2     | 3            | N V D I Z C                        | <b>STY</b>                                         |
|                                                    | Zero Page, X    | STY addr, X      | Y → [addr + X]                              | 94          | 148 | 2     | 4            | - - - - -                          |                                                    |
|                                                    | Absolute        | STY ADDR         | Y → [ADDR]                                  | 8C          | 140 | 3     | 4            |                                    |                                                    |
| <b>TAX<br/>TXA<br/>TAY<br/>TYA<br/>TSX<br/>TXS</b> | Implied         | TAX              | A → X                                       | AA          | 170 | 1     | 2            | N V D I Z C                        | <b>TAX<br/>TXA<br/>TAY<br/>TYA<br/>TSX<br/>TXS</b> |
|                                                    | Implied         | TXA              | X → A                                       | 8A          | 138 | 1     | 2            | ✓ - - - ✓ -                        |                                                    |
|                                                    | Implied         | TAY              | A → Y                                       | A8          | 168 | 1     | 2            | - - - - -                          |                                                    |
|                                                    | Implied         | TYA              | Y → A                                       | 98          | 152 | 1     | 2            | All Transfers xcpt TXS             |                                                    |
|                                                    | Implied         | TSX              | SP → X                                      | BA          | 186 | 1     | 2            | - - - - -                          |                                                    |
|                                                    | Implied         | TXS              | X → SP                                      | 9A          | 154 | 1     | 2            | - - - - -                          |                                                    |

# REGULAR ASCII CHART (character codes 0 – 127)

|              |             |   |       |              |             |   |       |              |             |    |              |             |   |              |             |   |              |             |   |              |             |   |              |             |   |
|--------------|-------------|---|-------|--------------|-------------|---|-------|--------------|-------------|----|--------------|-------------|---|--------------|-------------|---|--------------|-------------|---|--------------|-------------|---|--------------|-------------|---|
| 000 <i>d</i> | 00 <i>h</i> | ☐ | (nul) | 016 <i>d</i> | 10 <i>h</i> | ► | (dle) | 032 <i>d</i> | 20 <i>h</i> | □  | 048 <i>d</i> | 30 <i>h</i> | 0 | 064 <i>d</i> | 40 <i>h</i> | @ | 080 <i>d</i> | 50 <i>h</i> | P | 096 <i>d</i> | 60 <i>h</i> | ‘ | 112 <i>d</i> | 70 <i>h</i> | p |
| 001 <i>d</i> | 01 <i>h</i> | ☺ | (soh) | 017 <i>d</i> | 11 <i>h</i> | ◄ | (dc1) | 033 <i>d</i> | 21 <i>h</i> | !  | 049 <i>d</i> | 31 <i>h</i> | 1 | 065 <i>d</i> | 41 <i>h</i> | A | 081 <i>d</i> | 51 <i>h</i> | Q | 097 <i>d</i> | 61 <i>h</i> | a | 113 <i>d</i> | 71 <i>h</i> | q |
| 002 <i>d</i> | 02 <i>h</i> | ☎ | (stx) | 018 <i>d</i> | 12 <i>h</i> | ‡ | (dc2) | 034 <i>d</i> | 22 <i>h</i> | "  | 050 <i>d</i> | 32 <i>h</i> | 2 | 066 <i>d</i> | 42 <i>h</i> | B | 082 <i>d</i> | 52 <i>h</i> | R | 098 <i>d</i> | 62 <i>h</i> | b | 114 <i>d</i> | 72 <i>h</i> | r |
| 003 <i>d</i> | 03 <i>h</i> | ♥ | (etx) | 019 <i>d</i> | 13 <i>h</i> | ‡ | (dc3) | 035 <i>d</i> | 23 <i>h</i> | #  | 051 <i>d</i> | 33 <i>h</i> | 3 | 067 <i>d</i> | 43 <i>h</i> | C | 083 <i>d</i> | 53 <i>h</i> | S | 099 <i>d</i> | 63 <i>h</i> | c | 115 <i>d</i> | 73 <i>h</i> | s |
| 004 <i>d</i> | 04 <i>h</i> | ♦ | (eot) | 020 <i>d</i> | 14 <i>h</i> | ⌋ | (dc4) | 036 <i>d</i> | 24 <i>h</i> | \$ | 052 <i>d</i> | 34 <i>h</i> | 4 | 068 <i>d</i> | 44 <i>h</i> | D | 084 <i>d</i> | 54 <i>h</i> | T | 100 <i>d</i> | 64 <i>h</i> | d | 116 <i>d</i> | 74 <i>h</i> | t |
| 005 <i>d</i> | 05 <i>h</i> | ♣ | (enq) | 021 <i>d</i> | 15 <i>h</i> | § | (nak) | 037 <i>d</i> | 25 <i>h</i> | %  | 053 <i>d</i> | 35 <i>h</i> | 5 | 069 <i>d</i> | 45 <i>h</i> | E | 085 <i>d</i> | 55 <i>h</i> | U | 101 <i>d</i> | 65 <i>h</i> | e | 117 <i>d</i> | 75 <i>h</i> | u |
| 006 <i>d</i> | 06 <i>h</i> | ♠ | (ack) | 022 <i>d</i> | 16 <i>h</i> | — | (syn) | 038 <i>d</i> | 26 <i>h</i> | &  | 054 <i>d</i> | 36 <i>h</i> | 6 | 070 <i>d</i> | 46 <i>h</i> | F | 086 <i>d</i> | 56 <i>h</i> | V | 102 <i>d</i> | 66 <i>h</i> | f | 118 <i>d</i> | 76 <i>h</i> | v |
| 007 <i>d</i> | 07 <i>h</i> | • | (bel) | 023 <i>d</i> | 17 <i>h</i> | ‡ | (etb) | 039 <i>d</i> | 27 <i>h</i> | '  | 055 <i>d</i> | 37 <i>h</i> | 7 | 071 <i>d</i> | 47 <i>h</i> | G | 087 <i>d</i> | 57 <i>h</i> | W | 103 <i>d</i> | 67 <i>h</i> | g | 119 <i>d</i> | 77 <i>h</i> | w |
| 008 <i>d</i> | 08 <i>h</i> | ■ | (bs)  | 024 <i>d</i> | 18 <i>h</i> | ↑ | (can) | 040 <i>d</i> | 28 <i>h</i> | (  | 056 <i>d</i> | 38 <i>h</i> | 8 | 072 <i>d</i> | 48 <i>h</i> | H | 088 <i>d</i> | 58 <i>h</i> | X | 104 <i>d</i> | 68 <i>h</i> | h | 120 <i>d</i> | 78 <i>h</i> | x |
| 009 <i>d</i> | 09 <i>h</i> |   | (tab) | 025 <i>d</i> | 19 <i>h</i> | ↓ | (em)  | 041 <i>d</i> | 29 <i>h</i> | )  | 057 <i>d</i> | 39 <i>h</i> | 9 | 073 <i>d</i> | 49 <i>h</i> | I | 089 <i>d</i> | 59 <i>h</i> | Y | 105 <i>d</i> | 69 <i>h</i> | i | 121 <i>d</i> | 79 <i>h</i> | y |
| 010 <i>d</i> | 0A <i>h</i> | ▣ | (lf)  | 026 <i>d</i> | 1A <i>h</i> |   | (eof) | 042 <i>d</i> | 2A <i>h</i> | *  | 058 <i>d</i> | 3A <i>h</i> | : | 074 <i>d</i> | 4A <i>h</i> | J | 090 <i>d</i> | 5A <i>h</i> | Z | 106 <i>d</i> | 6A <i>h</i> | j | 122 <i>d</i> | 7A <i>h</i> | z |
| 011 <i>d</i> | 0B <i>h</i> | ♂ | (vt)  | 027 <i>d</i> | 1B <i>h</i> | ← | (esc) | 043 <i>d</i> | 2B <i>h</i> | +  | 059 <i>d</i> | 3B <i>h</i> | ; | 075 <i>d</i> | 4B <i>h</i> | K | 091 <i>d</i> | 5B <i>h</i> | [ | 107 <i>d</i> | 6B <i>h</i> | k | 123 <i>d</i> | 7B <i>h</i> | { |
| 012 <i>d</i> | 0C <i>h</i> |   | (np)  | 028 <i>d</i> | 1C <i>h</i> | ⌞ | (fs)  | 044 <i>d</i> | 2C <i>h</i> | ,  | 060 <i>d</i> | 3C <i>h</i> | < | 076 <i>d</i> | 4C <i>h</i> | L | 092 <i>d</i> | 5C <i>h</i> | \ | 108 <i>d</i> | 6C <i>h</i> | l | 124 <i>d</i> | 7C <i>h</i> |   |
| 013 <i>d</i> | 0D <i>h</i> | ♪ | (cr)  | 029 <i>d</i> | 1D <i>h</i> | ↔ | (gs)  | 045 <i>d</i> | 2D <i>h</i> | –  | 061 <i>d</i> | 3D <i>h</i> | = | 077 <i>d</i> | 4D <i>h</i> | M | 093 <i>d</i> | 5D <i>h</i> | ] | 109 <i>d</i> | 6D <i>h</i> | m | 125 <i>d</i> | 7D <i>h</i> | } |
| 014 <i>d</i> | 0E <i>h</i> | ♫ | (so)  | 030 <i>d</i> | 1E <i>h</i> | ▲ | (rs)  | 046 <i>d</i> | 2E <i>h</i> | .  | 062 <i>d</i> | 3E <i>h</i> | > | 078 <i>d</i> | 4E <i>h</i> | N | 094 <i>d</i> | 5E <i>h</i> | ^ | 110 <i>d</i> | 6E <i>h</i> | n | 126 <i>d</i> | 7E <i>h</i> | ~ |
| 015 <i>d</i> | 0F <i>h</i> | ✱ | (si)  | 031 <i>d</i> | 1F <i>h</i> | ▼ | (us)  | 047 <i>d</i> | 2F <i>h</i> | /  | 063 <i>d</i> | 3F <i>h</i> | ? | 079 <i>d</i> | 4F <i>h</i> | O | 095 <i>d</i> | 5F <i>h</i> | _ | 111 <i>d</i> | 6F <i>h</i> | o | 127 <i>d</i> | 7F <i>h</i> | △ |

## EXTENDED ASCII CHART (character codes 128 – 255) LATIN1/CP1252

|              |             |     |              |             |    |              |             |   |              |             |   |              |             |   |              |             |   |              |             |   |              |             |   |
|--------------|-------------|-----|--------------|-------------|----|--------------|-------------|---|--------------|-------------|---|--------------|-------------|---|--------------|-------------|---|--------------|-------------|---|--------------|-------------|---|
| 128 <i>d</i> | 80 <i>h</i> | €   | 144 <i>d</i> | 90 <i>h</i> |    | 160 <i>d</i> | A0 <i>h</i> | ↻ | 176 <i>d</i> | B0 <i>h</i> | ° | 192 <i>d</i> | C0 <i>h</i> | À | 208 <i>d</i> | D0 <i>h</i> | Ð | 224 <i>d</i> | E0 <i>h</i> | à | 240 <i>d</i> | F0 <i>h</i> | ð |
| 129 <i>d</i> | 81 <i>h</i> |     | 145 <i>d</i> | 91 <i>h</i> | ‘  | 161 <i>d</i> | A1 <i>h</i> | ¡ | 177 <i>d</i> | B1 <i>h</i> | ± | 193 <i>d</i> | C1 <i>h</i> | Á | 209 <i>d</i> | D1 <i>h</i> | Ñ | 225 <i>d</i> | E1 <i>h</i> | á | 241 <i>d</i> | F1 <i>h</i> | ñ |
| 130 <i>d</i> | 82 <i>h</i> | ,   | 146 <i>d</i> | 92 <i>h</i> | ,’ | 162 <i>d</i> | A2 <i>h</i> | ¢ | 178 <i>d</i> | B2 <i>h</i> | ² | 194 <i>d</i> | C2 <i>h</i> | Â | 210 <i>d</i> | D2 <i>h</i> | Ò | 226 <i>d</i> | E2 <i>h</i> | â | 242 <i>d</i> | F2 <i>h</i> | ò |
| 131 <i>d</i> | 83 <i>h</i> | ƒ   | 147 <i>d</i> | 93 <i>h</i> | “  | 163 <i>d</i> | A3 <i>h</i> | £ | 179 <i>d</i> | B3 <i>h</i> | ³ | 195 <i>d</i> | C3 <i>h</i> | Ã | 211 <i>d</i> | D3 <i>h</i> | Ó | 227 <i>d</i> | E3 <i>h</i> | ã | 243 <i>d</i> | F3 <i>h</i> | ó |
| 132 <i>d</i> | 84 <i>h</i> | „   | 148 <i>d</i> | 94 <i>h</i> | ”  | 164 <i>d</i> | A4 <i>h</i> | ¤ | 180 <i>d</i> | B4 <i>h</i> | ´ | 196 <i>d</i> | C4 <i>h</i> | Ä | 212 <i>d</i> | D4 <i>h</i> | Ô | 228 <i>d</i> | E4 <i>h</i> | ä | 244 <i>d</i> | F4 <i>h</i> | ô |
| 133 <i>d</i> | 85 <i>h</i> | ... | 149 <i>d</i> | 95 <i>h</i> | ●  | 165 <i>d</i> | A5 <i>h</i> | ¥ | 181 <i>d</i> | B5 <i>h</i> | µ | 197 <i>d</i> | C5 <i>h</i> | Å | 213 <i>d</i> | D5 <i>h</i> | Õ | 229 <i>d</i> | E5 <i>h</i> | å | 245 <i>d</i> | F5 <i>h</i> | ö |
| 134 <i>d</i> | 86 <i>h</i> | †   | 150 <i>d</i> | 96 <i>h</i> | –  | 166 <i>d</i> | A6 <i>h</i> | ¦ | 182 <i>d</i> | B6 <i>h</i> | ¶ | 198 <i>d</i> | C6 <i>h</i> | Æ | 214 <i>d</i> | D6 <i>h</i> | Ö | 230 <i>d</i> | E6 <i>h</i> | æ | 246 <i>d</i> | F6 <i>h</i> | ö |
| 135 <i>d</i> | 87 <i>h</i> | ‡   | 151 <i>d</i> | 97 <i>h</i> | -- | 167 <i>d</i> | A7 <i>h</i> | § | 183 <i>d</i> | B7 <i>h</i> | · | 199 <i>d</i> | C7 <i>h</i> | Ç | 215 <i>d</i> | D7 <i>h</i> | × | 231 <i>d</i> | E7 <i>h</i> | ç | 247 <i>d</i> | F7 <i>h</i> | ÷ |
| 136 <i>d</i> | 88 <i>h</i> | ˆ   | 152 <i>d</i> | 98 <i>h</i> | ˜  | 168 <i>d</i> | A8 <i>h</i> | ¨ | 184 <i>d</i> | B8 <i>h</i> | ¸ | 200 <i>d</i> | C8 <i>h</i> | È | 216 <i>d</i> | D8 <i>h</i> | Ø | 232 <i>d</i> | E8 <i>h</i> | è | 248 <i>d</i> | F8 <i>h</i> | ø |
| 137 <i>d</i> | 89 <i>h</i> | ‰   | 153 <i>d</i> | 99 <i>h</i> | ™  | 169 <i>d</i> | A9 <i>h</i> | © | 185 <i>d</i> | B9 <i>h</i> | ¹ | 201 <i>d</i> | C9 <i>h</i> | É | 217 <i>d</i> | D9 <i>h</i> | Ù | 233 <i>d</i> | E9 <i>h</i> | é | 249 <i>d</i> | F9 <i>h</i> | ù |
| 138 <i>d</i> | 8A <i>h</i> | Š   | 154 <i>d</i> | 9A <i>h</i> | š  | 170 <i>d</i> | AA <i>h</i> | ª | 186 <i>d</i> | BA <i>h</i> | º | 202 <i>d</i> | CA <i>h</i> | Ê | 218 <i>d</i> | DA <i>h</i> | Ú | 234 <i>d</i> | EA <i>h</i> | ê | 250 <i>d</i> | FA <i>h</i> | ú |
| 139 <i>d</i> | 8B <i>h</i> | <   | 155 <i>d</i> | 9B <i>h</i> | >  | 171 <i>d</i> | AB <i>h</i> | « | 187 <i>d</i> | BB <i>h</i> | » | 203 <i>d</i> | CB <i>h</i> | Ë | 219 <i>d</i> | DB <i>h</i> | Û | 235 <i>d</i> | EB <i>h</i> | ë | 251 <i>d</i> | FB <i>h</i> | û |
| 140 <i>d</i> | 8C <i>h</i> | ƒ   | 156 <i>d</i> | 9C <i>h</i> | œ  | 172 <i>d</i> | AC <i>h</i> | ¬ | 188 <i>d</i> | BC <i>h</i> | ¼ | 204 <i>d</i> | CC <i>h</i> | Ï | 220 <i>d</i> | DC <i>h</i> | Ü | 236 <i>d</i> | EC <i>h</i> | ì | 252 <i>d</i> | FC <i>h</i> | ü |
| 141 <i>d</i> | 8D <i>h</i> |     | 157 <i>d</i> | 9D <i>h</i> |    | 173 <i>d</i> | AD <i>h</i> |   | 189 <i>d</i> | BD <i>h</i> | ½ | 205 <i>d</i> | CD <i>h</i> | Î | 221 <i>d</i> | DD <i>h</i> | Ý | 237 <i>d</i> | ED <i>h</i> | í | 253 <i>d</i> | FD <i>h</i> | ý |
| 142 <i>d</i> | 8E <i>h</i> | Ž   | 158 <i>d</i> | 9E <i>h</i> | ž  | 174 <i>d</i> | AE <i>h</i> | ® | 190 <i>d</i> | BE <i>h</i> | ¾ | 206 <i>d</i> | CE <i>h</i> | Ï | 222 <i>d</i> | DE <i>h</i> | Þ | 238 <i>d</i> | EE <i>h</i> | î | 254 <i>d</i> | FE <i>h</i> | þ |
| 143 <i>d</i> | 8F <i>h</i> |     | 159 <i>d</i> | 9F <i>h</i> | ÿ  | 175 <i>d</i> | AF <i>h</i> | – | 191 <i>d</i> | BF <i>h</i> | ¿ | 207 <i>d</i> | CF <i>h</i> | İ | 223 <i>d</i> | DF <i>h</i> | ƒ | 239 <i>d</i> | EF <i>h</i> | ï | 255 <i>d</i> | FF <i>h</i> | ÿ |

### Hexadecimal to Binary

|   |      |   |      |   |      |   |      |
|---|------|---|------|---|------|---|------|
| 0 | 0000 | 4 | 0100 | 8 | 1000 | C | 1100 |
| 1 | 0001 | 5 | 0101 | 9 | 1001 | D | 1101 |
| 2 | 0010 | 6 | 0110 | A | 1010 | E | 1110 |
| 3 | 0011 | 7 | 0111 | B | 1011 | F | 1111 |

### Groups of ASCII-Code in Binary

| Bit 6 | Bit 5 | Group                  |
|-------|-------|------------------------|
| 0     | 0     | Control Characters     |
| 0     | 1     | Digits and Punctuation |
| 1     | 0     | Upper Case and Special |
| 1     | 1     | Lower Case and Special |

© 2009 Michael Goerz

This work is licensed under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 License.

To view a copy of this license, visit

<http://creativecommons.org/licenses/by-nc-sa/>