

Grundlagen Rechnertechnik

Wintersemester 2012/13
Prof. Dr. Peter Gerwinski

Stand: 21. Januar 2013

Soweit nicht anders angegeben:

Copyright © 2012 Peter Gerwinski

Lizenz: CC-by-sa (Version 3.0) oder GNU GPL (Version 3 oder höher)

Sie können dieses Skript einschließlich Quelltext und Beispielprogramme herunterladen unter:

<http://www.peter.gerwinski.de/download/rtech-2012ws.tar.gz>

Inhaltsverzeichnis

1 Einführung	7
1.1 Was ist Rechnertechnik?	7
1.2 Was ist ein Computer?	7
1.3 Rechnertechnik im Verhältnis zu anderen Disziplinen	9
2 Vom Schaltkreis zum Computer	10
2.1 Logik-Schaltkreise	10
2.2 Binärdarstellung von Zahlen	14
2.3 Vom Logik-Schaltkreis zum Addierer	15
2.4 Negative Zahlen	17
2.5 Vom Addierer zum Computer	19
2.6 Computer-Sprachen	23
2.6.1 Maschinensprache	23
2.6.2 Assembler	24
2.6.3 Hochsprachen	25
2.7 Struktur von Assembler-Programmen	26
3 Architekturmerkmale von Prozessoren	28
3.1 Speicherarchitekturen	28
3.2 Registerarchitekturen	29
3.3 Befehlssätze	30
4 Der CPU-Stack	30
4.1 Implementation	30
4.2 Unterprogramme	31
4.3 Registerinhalte sichern	32
5 Hardwarenahe Programmierung	33
5.1 Bit-Operationen	33
5.1.1 Zahlensysteme	33
5.1.2 Bit-Operationen in C	35
5.2 I/O-Ports	36
5.3 Interrupts	37
5.4 volatile-Variable	38
5.5 Software-Interrupts	39
5.6 Byte-Reihenfolge – Endianness	39
5.6.1 Konzept	39
5.6.2 Dateiformate	40
5.6.3 Datenübertragung	40
5.7 Speicherausrichtung – Alignment	41

6 Anwender-Software	42
6.1 Relokation und Linken	42
6.2 Dateiformate	42
6.3 Die Toolchain	43
6.4 Besonderheiten von Mikro-Controllern	44
7 Bus-Systeme	45
7.1 Konzept	45
7.2 Zu berücksichtigen	45
7.3 Beispiele	46
7.4 Beispiel: Benutzung des I ² C-Busses	49
8 Pipelining	50
8.1 Konzept	50
8.2 Arithmetik-Pipelines	51
8.3 Instruktions-Pipelines	54
9 Ausblick	56

Abbildungsverzeichnis

Abb. 1	Zirkel	7
	Quelle: http://commons.wikimedia.org/wiki/File:Drawing-a-circle-with-the-compasses.jpg , abgerufen am 7. 10. 2012 Autor: International Correspondence Schools, Scranton, PA., USA Lizenz: gemeinfrei – <i>public domain</i>	
Abb. 2	Lineal	7
	Quelle: http://commons.wikimedia.org/wiki/File:Holding-a-ruling-pen.jpg , abgerufen am 7. 10. 2012, selbst bearbeitet Autor: International Correspondence Schools, Scranton, PA., USA Lizenz: gemeinfrei – <i>public domain</i>	
Abb. 3	Mechanische Rechenmaschine	8
	Quelle: http://commons.wikimedia.org/wiki/File:1890s_adding_machine.jpg , abgerufen am 7. 10. 2012 Autor: http://commons.wikimedia.org/wiki/User:Trekphiler Lizenz: CC-by-sa (Version 3.0, nicht portiert)	
Abb. 4	Wissenschaftliche Disziplinen mit Bezug zur Informatik, angeordnet nach Abstraktionsgrad ihres jeweiligen Gegenstandes	9
	Quelle/Autor: selbst erstellt Lizenz: CC-by-sa (Version 3.0) oder GNU GPL (Version 3 oder höher)	
Abb. 5	Funktionsprinzip eines Relais	10
	Quelle: http://commons.wikimedia.org/wiki/File:Relay_principle_horizontal.jpg , abgerufen am 7. 10. 2012, selbst bearbeitet Autor: http://commons.wikimedia.org/wiki/User:Bisgaard Lizenz: GNU FDL (Version 1.2 oder höher) oder CC-by-sa (Version 3.0, nicht portiert)	
Abb. 6	Zuse Z3 (Nachbau im Deutschen Museum)	11
	Quelle: http://commons.wikimedia.org/wiki/File:Z3_Deutsches_Museum.JPG , abgerufen am 7. 10. 2012 Autor: http://de.wikipedia.org/wiki/User:Venusianer Lizenz: GNU FDL (Version 1.2 oder höher) oder CC-by-sa (Version 3.0, nicht portiert)	
Abb. 7	Elektronenröhren	11
	Quelle: http://commons.wikimedia.org/wiki/File:Elektronenroehren-auswahl.jpg , abgerufen am 7. 10. 2012 Autor: http://de.wikipedia.org/wiki/Benutzer:Quark48 Lizenz: CC-by-sa 2.0 Deutschland	
Abb. 8	Schemazeichnung einer Elektronenröhre (Triode)	11
	Quelle: http://commons.wikimedia.org/wiki/File:Elektronenroehre_real.png , abgerufen am 7. 10. 2012 Autor: http://de.wikipedia.org/wiki/Benutzer:Quark48 Lizenz: CC-by 2.0 Deutschland	
Abb. 9	ENIAC – der erste Computer auf Basis von Elektronenröhren	12
	Quelle: http://commons.wikimedia.org/wiki/File:Eniac.jpg , abgerufen am 11. 10. 2012 Autor: <i>unbekannt (U. S. Army)</i> Lizenz: gemeinfrei – <i>public domain</i>	
Abb. 10	Transistoren	12
	Quelle: http://commons.wikimedia.org/wiki/File:Transistors-white.jpg , abgerufen am 11. 10. 2012 Autor: http://de.wikipedia.org/wiki/Benutzer:Benedikt.Seidl Lizenz: gemeinfrei – <i>public domain</i>	

Abb. 11 Intel 80486DX2 (Oberseite)	13
Quelle: http://commons.wikimedia.org/wiki/File:Intel_80486DX2_top.jpg , abgerufen am 1. 11. 2012 Autor: http://en.wikipedia.org/wiki/User:Solipsist Lizenz: CC-by-sa (Version 2.0, nicht portiert)	
Abb. 12 Intel 80486DX2 (Unterseite)	13
Quelle: http://commons.wikimedia.org/wiki/File:Intel_80486DX2_bottom.jpg , abgerufen am 1. 11. 2012 Autor: http://en.wikipedia.org/wiki/User:Solipsist Lizenz: CC-by-sa (Version 2.0, nicht portiert)	
Abb. 13 Intel 80486DX2 (geöffnet)	13
Quelle: http://commons.wikimedia.org/wiki/File:80486dx2-large.jpg , abgerufen am 11. 10. 2012 Autor: http://en.wikipedia.org/wiki/User:Überpenguin Lizenz: GNU FDL (Version 1.2 oder höher) oder CC-by-sa (Version 3.0, nicht portiert)	
Abb. 14 Integrierte Schaltung, vergrößert	13
Quelle: http://commons.wikimedia.org/wiki/File:IC_Nanotechnology_2400X.JPG , abgerufen am 11. 10. 2012 Autor: http://commons.wikimedia.org/wiki/User:Angeloleithold Lizenz: GNU FDL (Version 1.2 oder höher) oder CC-by-sa (Version 3.0, nicht portiert)	
Abb. 15 Programm in Maschinensprache und Assembler	23
Quelle/Autor: selbst erstellt Lizenz: CC-by-sa (Version 3.0) oder GNU GPL (Version 3 oder höher) Quelle/Autor: selbst erstellt Lizenz: CC-by-sa (Version 3.0) oder GNU GPL (Version 3 oder höher)	
Abb. 17 Symbolbilder für „Waschen“, „Trocknen“ und „Bügeln“	50
Quelle: http://de.wikipedia.org/wiki/Textilpflegesymbol , abgerufen am 23. 1. 2012 Autor: http://de.wikipedia.org/wiki/Benutzer:Andre_Riemann Lizenz: gemeinfrei – <i>public domain</i>	

1 Einführung

1.1 Was ist Rechnertechnik?

Wikipedia [1] definiert *Rechnertechnik* folgendermaßen:

Die *Rechnertechnik* (auch Computertechnik, englisch *computer engineering*) beschäftigt sich als technisches Fachgebiet mit der Konzeption von informationsverarbeitenden Anlagen, also Computern. Sie baut auf der Digitaltechnik und Mikroelektronik auf.

Die Kernfrage, mit denen wir uns in dieser Lehrveranstaltung befassen werden, lautet also:

Wie funktioniert ein Computer?

Im Gegensatz zu z. B. Lebewesen werden Computer von Menschen entwickelt und gebaut. Daher ist es grundsätzlich möglich, ihre Funktionsweise vollständig zu verstehen.

1.2 Was ist ein Computer?

Ein *Rechner*, englisch *Computer*, ist von der Wortbedeutung her zunächst einmal ein Werkzeug, das dem menschlichen Gehirn beim Rechnen hilft.

Beispiele für derartige Rechenhilfen:

- Finger

Die eigenen Finger werden häufig beim Kopfrechnen als zusätzliches Werkzeug eingesetzt, um sich Zwischenergebnisse (typischerweise im Zahlenbereich von 1 bis 10) zu merken.

- Papier und Bleistift, Sand, Tontafeln etc.

Das schriftliche Addieren, Subtrahieren, Multiplizieren und Dividieren ist aus der Grundschule bekannt. Auch weitere Operationen (z. B. Wurzelziehen) lassen sich auf diese Weise ausführen.

- Zirkel und Lineal

Erstaunlich viele geometrische Konstruktionen lassen sich allein mit Hilfe eines Zirkels und eines Lineals (ohne Maßstab) durchführen. Beispiele:

- Addition, Subtraktion, Multiplikation und Division von Zahlen, dargestellt durch Längenverhältnisse
- Berechnung von $\sqrt{2}$ als Längenverhältnis
- Division eines Winkels durch 2 oder Potenzen von 2
- Konstruktion eines regelmäßigen Fünfecks, Siebzehnecks oder 65537ecks

Wie im 19. Jahrhundert bewiesen wurde (Körper- und GALOIS-Theorie), gibt es aber auch mathematische Probleme, die sich *nicht* allein mit Zirkel und Lineal lösen lassen. Beispiele:

- Berechnung von $\sqrt[3]{2}$ als Längenverhältnis („Würfelverdopplung“)
- Berechnung der Zahl π als Längenverhältnis („Quadratur des Kreises“)
- Division eines Winkels durch 3
- Konstruktion eines regelmäßigen Siebenecks

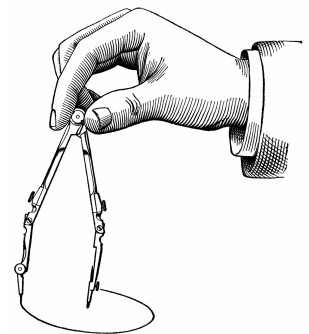


Abbildung 1: Zirkel

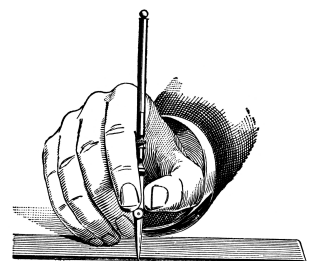


Abbildung 2: Lineal

- Abakus

Bei dieser Rechenhilfe ersetzen verschiebbare Kugeln geschriebene Ziffern. Die eigentliche Rechnung erfolgt genau wie beim schriftlichen Rechnen.

- Mechanische Rechenmaschine

Die Vorgänge, die beim schriftlichen Rechnen durch einen Menschen ausgeführt werden, erfolgen hier durch Zahnräder.

- Rechenschieber

Zwei gegeneinander verschiebbare logarithmische Skalen erlauben es, die Multiplikation auf die Addition zurückzuführen und die Division auf die Subtraktion. Zusätzliche Skalen ermöglichen auch das Ziehen von Wurzeln und weitere Operationen.

- Taschenrechner

Die Vorgänge, die beim schriftlichen Rechnen durch einen Menschen ausgeführt werden, erfolgen hier durch elektronische Schaltungen.

- Computer

Zusätzlich zu den Fähigkeiten eines Taschenrechners erlaubt es ein Computer, Folgen von Rechenvorgängen in Gestalt von *Programmen* vorzubereiten, bei deren Abarbeitung auf bereits vorliegende Ergebnisse und sonstige Ereignisse reagiert werden kann (bedingte Anweisungen und Sprunganweisungen).



Abbildung 3: Mechanische Rechenmaschine

Die oben aufgeführten Rechner lassen sich in zwei Klassen unterteilen:

- *Analogrechner* stellen Zahlenwerte durch kontinuierliche physikalische Größen (Länge, elektrische Spannung, ...) dar. Die Genauigkeit der Ergebnisse entspricht der Genauigkeit, mit der der Rechner gefertigt wurde. Sie unterliegt Qualitätsschwankungen und äußeren Einflüssen und kann sich im Laufe der Zeit allmählich ändern (Qualitätsabnahme durch Verschleiß, aber auch Qualitätsverbesserung durch „Einschleifen“).

Zu dieser Klasse gehören Zirkel und Lineal sowie der Rechenschieber.

- *Digitalrechner* stellen Zahlenwerte durch Ziffern (englisch: *digit*) dar. Sie sind benannt nach den menschlichen Fingern (lateinisch: *digitus*), die als Rechenwerkzeug dieser Klasse angehören. Die erreichbare Genauigkeit hängt allein von der Anzahl der verwendbaren Ziffern ab. Qualitätsverlust, z. B. durch Verschleiß, spielt solange keine Rolle, wie die einzelnen Werte der Ziffern noch unterschieden werden können. Sobald dies nicht mehr der Fall ist, ist der Digitalrechner schlagartig komplett unbrauchbar.

Neben den namensgebenden Fingern gehören zu dieser Klasse auch das schriftliche Rechnen, der Abakus, mechanische Rechenmaschinen und Taschenrechner.

Wenn heutzutage von „Computern“ die Rede ist, so sind damit stets *Digitalcomputer* gemeint. Daneben gibt es auch *Analogcomputer*, die Zahlenwerte durch elektrische Spannungen, Wasserdruck oder sonstige physikalische Größen modellieren.

Die Programmierung elektronischer Analogcomputer erfolgt durch das Umstecken von Schaltungen. Durch geeignete Wahl der Bauelemente können nicht nur die Grundrechenarten, sondern auch Operationen aus der Differential- und Integralrechnung effizient ausgeführt werden. Analogcomputer eignen sich daher besonders gut zum Lösen von Differentialgleichungen. Auf diesem Gebiet hatten Analogcomputer bis in die 1970er Jahre hinein Vorteile gegenüber Digitalcomputern [2], wurden aber mittlerweile durch diese nahezu vollständig verdrängt.

In dieser Lehrveranstaltung geht es ausschließlich um Digitalcomputer.

1.3 Rechnertechnik im Verhältnis zu anderen Disziplinen

Rechnertechnik als wissenschaftliche Disziplin baut auf der Digitaltechnik und der Mikroelektronik auf, bei denen es sich um Teilgebiete der Elektrotechnik handelt.

Die Aufgabe der Rechnertechnik besteht darin, das Gegenständliche – reale Drähte, Halbleiter usw. – so weit zu abstrahieren, daß ein universell einsetzbarer Rechner entsteht. Die Rechenergebnisse hängen also nicht mehr von dem konkreten Rechner ab, auf dem die Programme laufen (wie es z. B. bei einem Rechenschieber der Fall ist), sondern der Rechner bildet eine Abstraktionsschicht für die Programme. Details, wie der Rechner funktioniert, sind für die Programmierung uninteressant; diese Probleme werden innerhalb der Rechnertechnik gelöst.

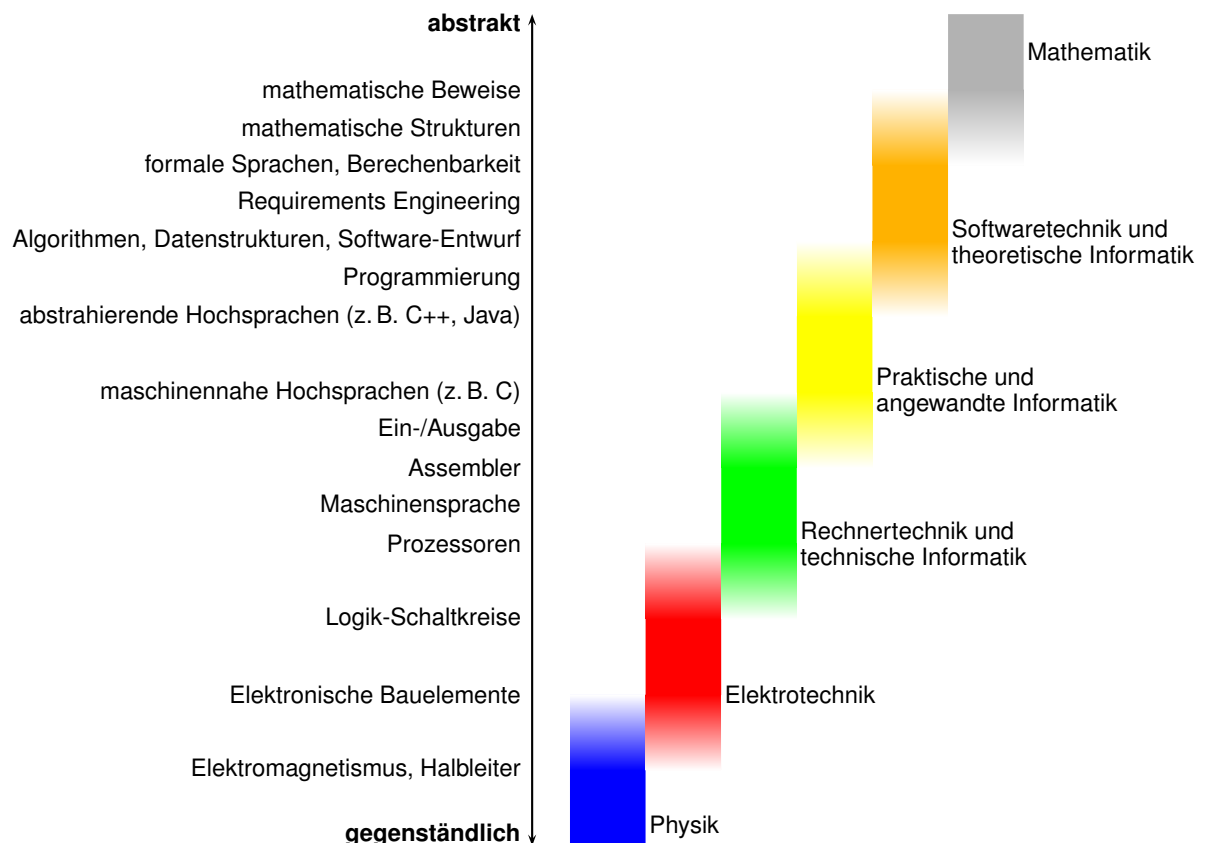


Abbildung 4: Wissenschaftliche Disziplinen mit Bezug zur Informatik, angeordnet nach Abstraktionsgrad ihres jeweiligen Gegenstandes

Umgekehrt verwendet die Rechnertechnik die Ergebnisse der Elektrotechnik (Verhalten elektronischer Logik-Schaltkreise) als Abstraktionsschicht, ohne sich für die Funktionsweise von Transistoren, Relais usw. im Detail zu interessieren. Um diese Abstraktionsschicht zu schaffen, schöpft die Elektrotechnik ihrerseits aus den Ergebnissen der Physik (Elektromagnetismus, Halbleiterphysik, ...).

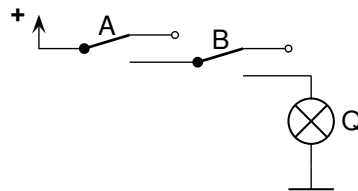
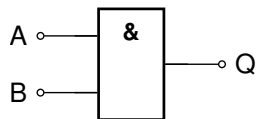
2 Vom Schaltkreis zum Computer

2.1 Logik-Schaltkreise

Bereits mit Lichtschaltern ist es möglich, logische Operationen durchzuführen:

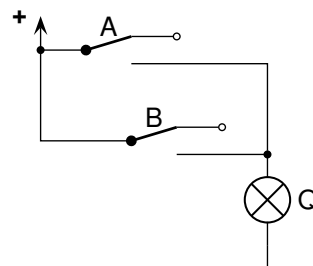
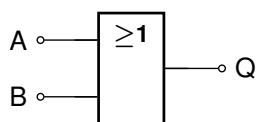
- Und-Verknüpfung: Reihenschaltung

A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1



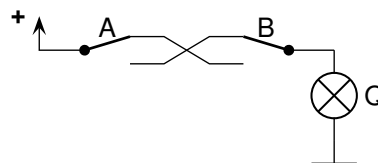
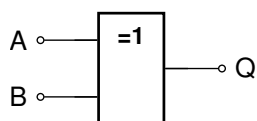
- Oder-Verknüpfung: Parallelschaltung

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1



- Exklusiv-Oder-Verknüpfung
„entweder A oder B, aber nicht beide“

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0



Mehrere dieser Logik-Schaltungen miteinander zu kombinieren, ist im allgemeinen zunächst nicht möglich, da die Eingabe durch eine mechanische Schalterstellung erfolgt und die Ausgabe durch das Leuchten einer Lampe.

Abhilfe schafft ein Schalter, der durch einen Elektromagneten betätigt wird, ein sog. *Relais*.

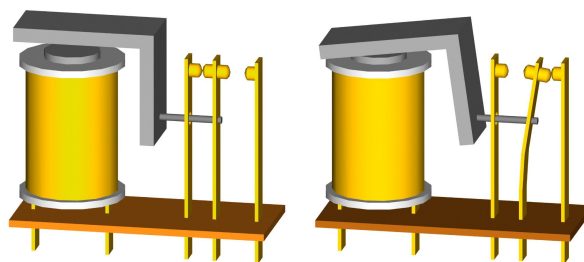
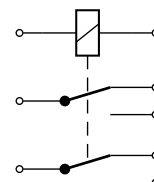
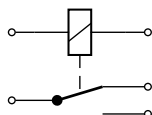


Abbildung 5: Funktionsprinzip eines Relais

Schaltzeichen für Relais mit 1 bzw. 2 Umschaltkontakten:



Bei Relais-Schaltungen ist es möglich, die von einer Logik-Schaltung ausgegebene Spannung als Eingabespannung für die nächste Logik-Schaltung zu verwenden. Jede im folgenden angegebene Logik-Schaltung bis hin zu einem vollständigen Computer kann daher mit Hilfe von Relais realisiert werden.

1941 bauten Konrad Zuse und Helmut Schreyer auf diese Weise den ersten Computer, die Zuse Z3.

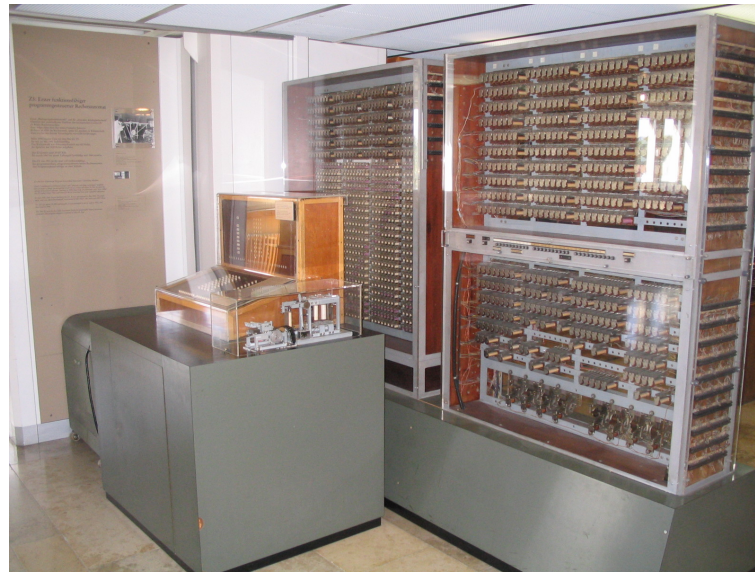


Abbildung 6: Zuse Z3 (Nachbau im Deutschen Museum)

Wesentlich schnellere Schaltzeiten als mit einem Relais erreicht man mit einer **Elektronenröhre**. Hierbei handelt es sich um einen evakuierten Glaskolben mit einer heizbaren **Kathode** in der Mitte, um die herum weitere **Elektroden** coaxial angeordnet sind.



Abbildung 7: Elektronenröhren

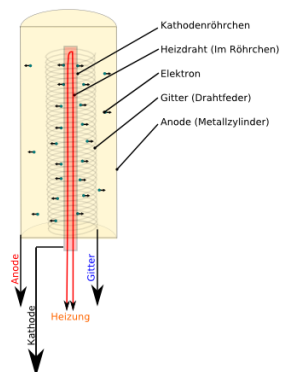


Abbildung 8: Schemazeichnung einer Elektronenröhre (Triode)

Die einfachste Bauform der Elektronenröhre ist die **Diode** (griechisch *dis hodós* – doppelter Weg). Sie hat außer der Kathode in der Mitte nur eine einzige äußere Elektrode, die **Anode**. Eine Diode wirkt als „elektronisches Ventil“ und lässt den Strom nur in einer Richtung durch, nämlich mit dem Minuspol an der Kathode und dem Pluspol an der Anode.

Eine weitere wichtige Bauform der Elektronenröhre ist die **Triode** (griechisch *treis hodós* – dreifacher Weg, siehe Abbildung 8). Bei ihr befindet sich zwischen Anode und Kathode eine weitere Elektrode, das **Gitter**. Durch Anlegen einer Spannung zwischen Kathode und Gitter kann man den Strom, der zwischen Anode und Kathode fließt, steuern. In diesem Sinne wirkt die Triode als „elektronischer Schalter“ und kann die Funktionalität eines Relais übernehmen.

Bis in die 1950er Jahre bildeten Elektronenröhren die Grundlage für Computer.

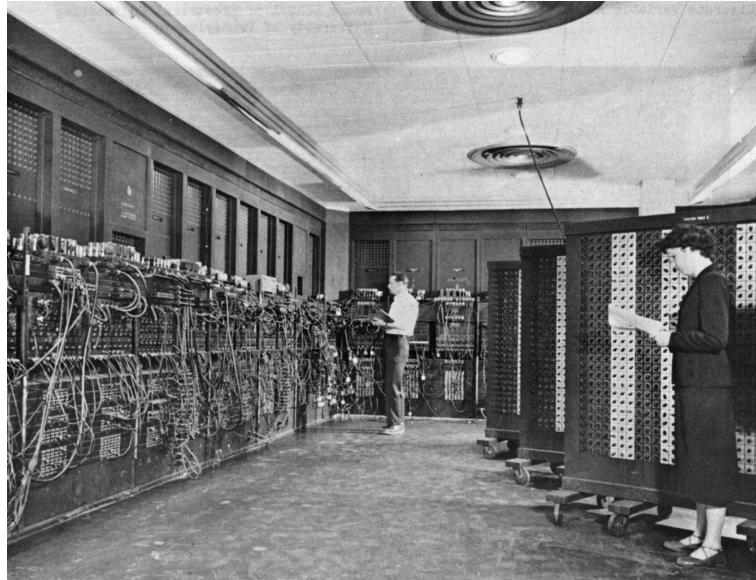


Abbildung 9: ENIAC – der erste Computer auf Basis von Elektronenröhren

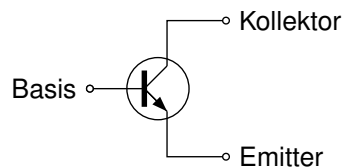
In den 1950er Jahren wurden Elektronenröhren allmählich durch *Halbleiterbauelemente* abgelöst. Auch hier gibt es die Bauformen *Diode* und *Triode*. Letztere wird meistens als *Transistor* bezeichnet (englisch *transfer resistor* – Übertragungswiderstand, d. h. steuerbarer Widerstand).

- Eine *Halbleiterdiode* – im folgenden kurz: *Diode* – wirkt als „elektronisches Ventil“:

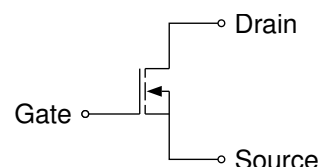
Anode $\circ \rightarrow \text{---} \blacktriangleright \text{---} \circ$ Kathode

Elektrischer Strom in Pfeilrichtung (Pluspol an Anode, Minuspol an Kathode) darf die Diode passieren; elektrischer Strom gegen Pfeilrichtung wird gesperrt.

- *Transistoren* gibt es in verschiedenen Bauformen (*Bipolartransistor*, *Feldeffekttransistor – FET*, ...):



Bipolartransistor



Feldeffekttransistor

Die elektrischen Eigenschaften der verschiedenen Bauformen sind im Detail unterschiedlich; alle können jedoch als „elektronische Schalter“ eingesetzt werden.

Eine typische Anwendung eines Transistors („*Emitterschaltung*“) besteht darin, einen Strom zwischen Basis und Emitter fließen zu lassen und dadurch den Strom zu steuern, der zwischen Kollektor und Emitter fließen darf. Ähnliches gilt für Feldeffekttransistoren: Eine zwischen Gate und Source angelegte Spannung steuert den Stromfluß zwischen Source und Drain.

(Mehr über Transistorschaltungen erfahren Sie in der Lehrveranstaltung *Grundlagen Elektrotechnik*.)

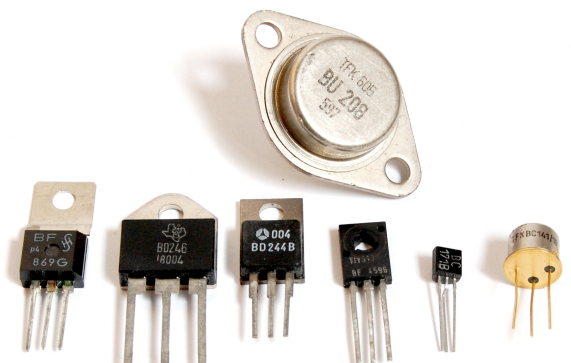


Abbildung 10: Transistoren

In einer *integrierten Schaltung* (Integrated Circuit – IC) werden mikroskopisch kleine Feldeffekttransistoren auf einem Halbleiteruntergrund direkt an denjenigen Stellen erzeugt, an denen sie in der Schaltung benötigt werden.



Abbildung 11: Intel 80486DX2 (Oberseite)

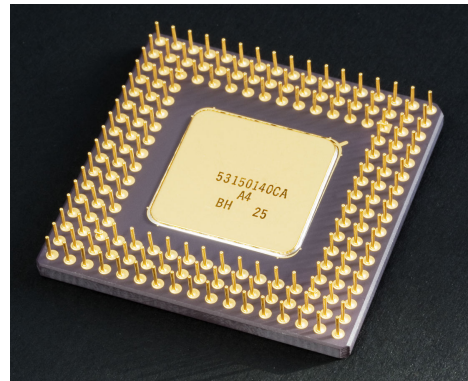


Abbildung 12: Intel 80486DX2 (Unterseite)

Die Abbildungen 11 und 12 zeigen die Ober- bzw. Unterseite eines Intel-80486DX2-Mikroprozessors (1992 erschienen). Das Gehäuse ist ca. 44 mm × 44 mm groß und verfügt auf der Unterseite über 168 Anschlußstifte.

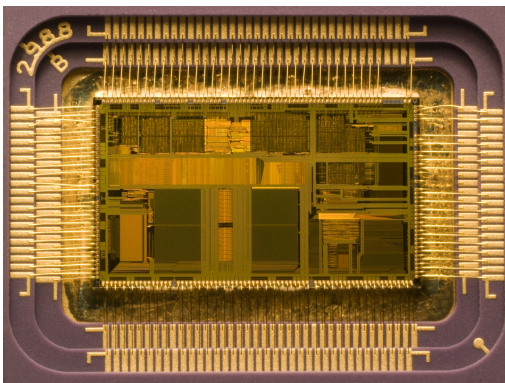


Abbildung 13: Intel 80486DX2 (geöffnet)

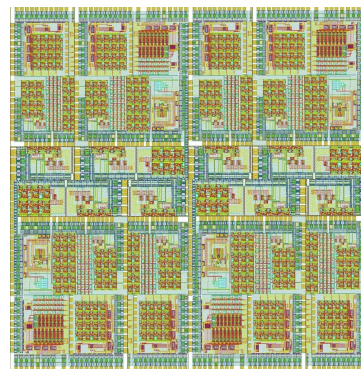


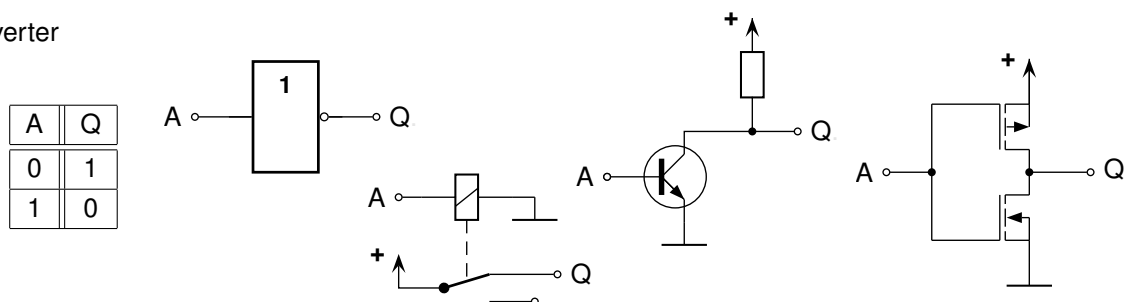
Abbildung 14: Integrierte Schaltung, vergrößert

Abbildung 13 zeigt das Innenleben eines Intel-80486DX2-Mikroprozessors. Die Platine (*Die*) in der Mitte ist 12 mm × 6,75 mm groß. Auf dieser Fläche von 81 mm² befinden sich ca. 1,2 Millionen Transistoren. Aktuelle integrierte Schaltkreise sind noch deutlich stärker miniaturisiert: Ein Intel-Core-i7-Mikroprozessor von 2012 enthält z. B. ca. 1,4 Milliarden Transistoren auf einem Die der Fläche 160 mm², also über 500mal so viele Transistoren pro Fläche.

Abbildung 14 zeigt einen 1200fach vergrößerten Ausschnitt aus einem integrierten Schaltkreis (ca. 0,05 mm × 0,05 mm).

Unter Verwendung von Halbleiterbauelementen und ohmschen Widerständen können wir nun die oben bereits mit mechanischen Schaltern realisierten Logik-Schaltungen reproduzieren und erweitern:

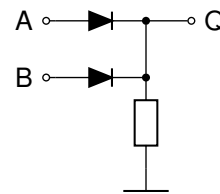
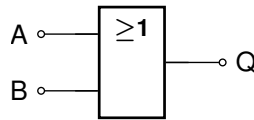
- Inverter



Von links nach rechts: Wahrheitstabelle, Schaltzeichen, Realisierungen mit Relais, mit Bipolartransistor und Widerstand bzw. mit zwei Feldeffekttransistoren (CMOS-Schaltung, Stand der Technik 2012). Anstelle von Transistoren können auch Elektronenröhren verwendet werden.

- Oder-Verknüpfung

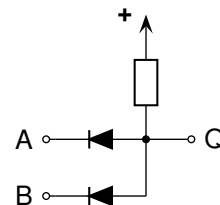
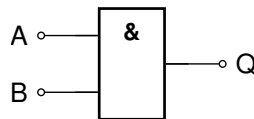
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1



Von links nach rechts: Wahrheitstabelle, Schaltzeichen, Realisierung mit Dioden und Widerstand.

- Und-Verknüpfung

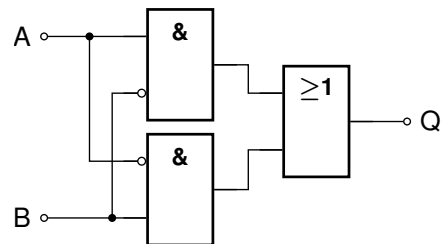
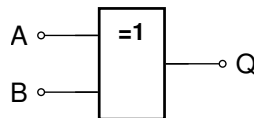
A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1



Von links nach rechts: Wahrheitstabelle, Schaltzeichen, Realisierung mit Dioden und Widerstand.

- Exklusiv-Oder-Verknüpfung
„entweder A oder B, aber nicht beide“

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0



Von links nach rechts: Wahrheitstabelle, Schaltzeichen, Realisierung durch Zurückführen auf bereits bekannte Logik-Schaltungen. Die Kringel an den Eingängen der Und-Schaltungen stehen für vorge-schaltete Inverter.

Bei den oben vorgestellten Schaltungen mit Bipolartransistoren und Dioden handelt es sich um (veraltete) Widerstand-Transistor-Logik (RTL) und Dioden-Logik (DL). In den Schaltungen aktueller Computer (Stand: 2012) kommen ausschließlich Transistoren zum Einsatz, z. B. die o. a. CMOS-Schaltung für einen Inverter.

Entscheidend ist, daß es überhaupt möglich ist, „ideale“ Logik-Schaltungen mit Hilfe realer Bauelemente zu bauen. Bei den weiteren Betrachtungen können wir uns daher von den konkreten Realisierungen lösen und allein mit den abstrahierten Logik-Schaltungen (Schaltzeichen in der mittleren Spalte) arbeiten.

2.2 Binärdarstellung von Zahlen

Es ist möglich, Zahlen allein mit den Ziffern 0 und 1 auszudrücken.

Normalerweise rechnen wir im Dezimalsystem mit den Ziffern 0 bis 9. Ab dem Zahlenwert 10 wird eine Ziffer vorangestellt, die wir mit $10^1 = 10$ multiplizieren. Ab dem Zahlenwert 100 wird eine Ziffer vorangestellt, die wir mit $10^2 = 100$ multiplizieren usw. 137 ist nur eine Abkürzung für $7 \cdot 10^0 + 3 \cdot 10^1 + 1 \cdot 10^2$.

Das Binärsystem funktioniert analog mit Zweier- anstelle von Zehnerpotenzen, also mit $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$ usw.

Wir lesen eine Binärzahl 100101_2 , also eine Folge von Binärziffern, von rechts nach links:

$$\begin{aligned}
 100101_2 &= 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5 \\
 &= 1 + 4 + 32 \\
 &= 37
 \end{aligned}$$

Das Rechnen mit Binärzahlen funktioniert genauso wie das Rechnen mit Dezimalzahlen, nur daß die „Zehner“-Überschreitung bereits bei 2 stattfindet:

$$\begin{array}{rcl} 0 + 0 & = & 0 \\ 0 + 1 & = & 1 \\ 1 + 0 & = & 1 \\ 1 + 1 & = & 10 \end{array}$$

Beispiel: schriftliche Addition der Zahlen $101100_2 = 44$ und $101110_2 = 46$

$$\begin{array}{r} 101100 \\ + 101110 \\ \hline 1011010 \end{array}$$

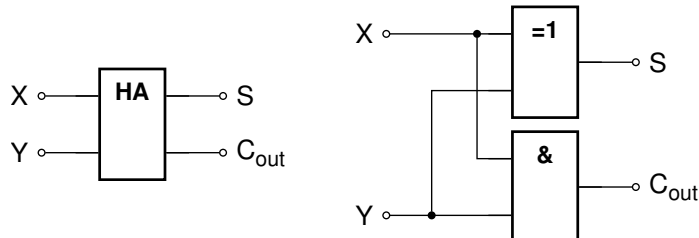
Das Ergebnis lautet:

$$\begin{aligned} 1011010_2 &= 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 + 0 \cdot 2^5 + 1 \cdot 2^6 \\ &= 2 + 8 + 16 + 64 \\ &= 90 \end{aligned}$$

2.3 Vom Logik-Schaltkreis zum Addierer

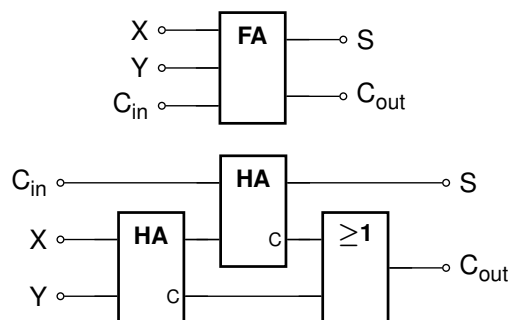
- 1-Bit-Addierer mit 2 Eingängen („Halbaddierer“)
 - Zwei 1-Bit-Zahlen („Bit“ = Binärziffer = 0 oder 1) werden an die Eingänge X und Y gelegt.
 - Das Rechenergebnis kann bis zu 2 Bits umfassen.
 - Am Ausgang soll das Rechenergebnis anliegen:
S soll die Einerziffer ($2^0 = 1$) und C_{out} die Zweierziffer ($2^1 = 2$) anzeigen.

X	Y	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



- 1-Bit-Addierer mit 3 Eingängen („Volladdierer“)
 - Drei 1-Bit-Zahlen werden an die Eingänge X, Y und C_{in} gelegt.
(„C“ steht für „carry“ – Übertrag.)
 - Das Rechenergebnis kann bis zu 2 Bits umfassen.
 - Am Ausgang soll das Rechenergebnis anliegen:
S soll die Einerziffer ($2^0 = 1$) und C_{out} die Zweierziffer ($2^1 = 2$) anzeigen.

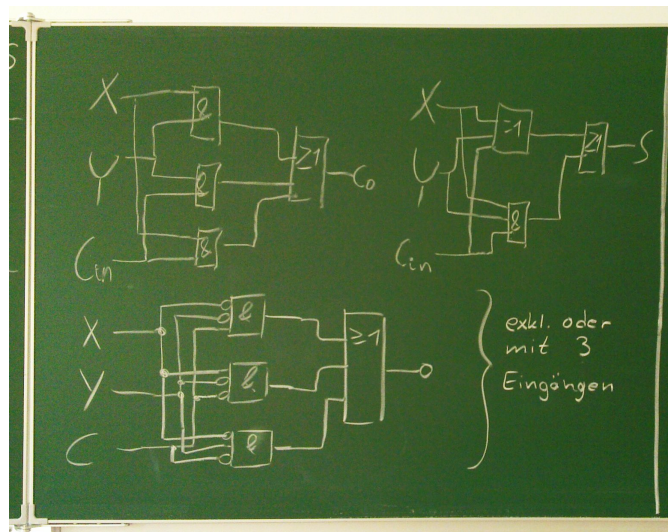
C_{in}	X	Y	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



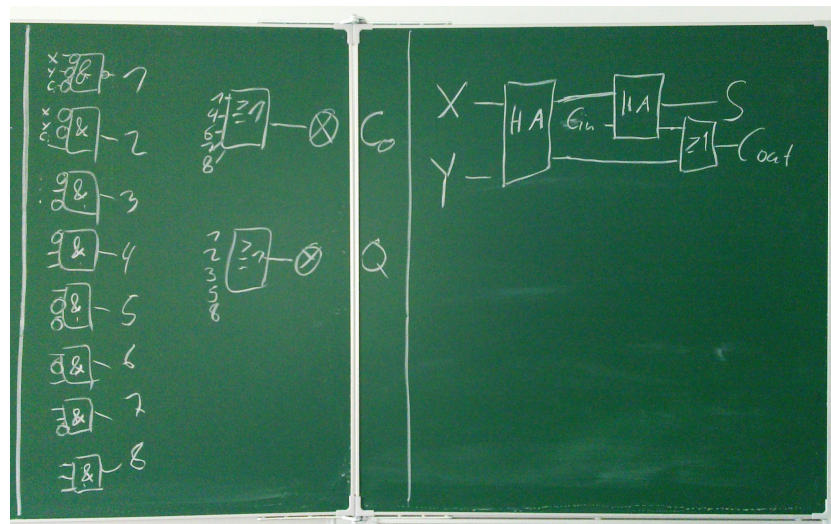
$$X + Y + C_{in} = 2 \cdot C_{out} + 1 \cdot S$$

C_{in}	X	Y		
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Eine andere Form der Wahrheitstabelle des Volladdierers



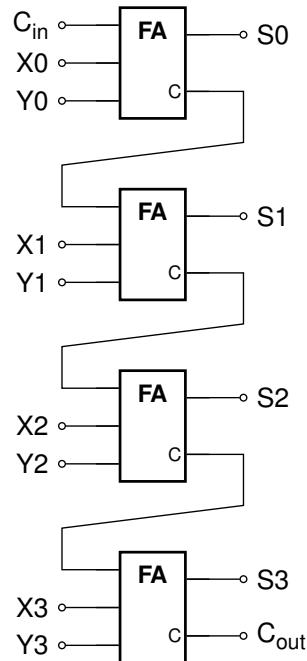
Realisierung eines Volladdierers durch Erkennen von Strukturen in der Wahrheitstabelle



Rechts: Realisierung eines Volladdierers durch Nachvollziehen der Operation (schriftliches Addieren) und Zurückführen auf Halbaddierer

Links: Realisierung eines Volladdierers „mit Gewalt“ (*brute force*): Für jede Zeile der Wahrheitstabelle wird eine Und-Schaltung aufgebaut. Einsen werden direkt an einen Eingang gelegt, Nullen invertiert. Für diejenigen Zeilen, in denen der Ausgang der Gesamtschaltung eine Eins liefern soll, werden die Ausgänge der Und-Schaltungen in einer großen Oder-Schaltung verknüpft. Diese Art der Realisierung ist theoretisch für jede gegebene Wahrheitstabelle möglich.

- 4-Bit-Addierer mit Übertrag (9 Eingänge, 5 Ausgänge)
 - Zwei 4-Bit-Zahlen („Operanden“) werden an jeweils 4 Eingänge gelegt.
 - Ein weiteres Bit („Übertrag“) wird an den Eingang C_{in} gelegt.
Diese 1-Bit-Zahl wird zusätzlich zu den zwei 4-Bit-Zahlen addiert.
 - Das Rechenergebnis kann bis zu 5 Bits umfassen.
Das höchstwertige Bit ist der neue Übertrag (C_{out}).



Problem: Das Signal für den Übertrag wandert von Volladdierer zu Volladdierer;
die Schaltzeiten addieren sich. → langsam

Lösungswege: siehe <http://de.wikipedia.org/wiki/Addierer>

2.4 Negative Zahlen

Was ist das „natürliche“ Format zur Darstellung negativer Zahlen durch Nullen und Einsen?

- Wenn wir zu der Binärzahl 1111_2 den Wert 1 addieren, erhalten wir 10000_2 .
- Wenn dies auf einem 4-Bit-Addierer geschieht, ist die vordere Ziffer 1 der neue Übertrag.
- Wenn dieser Wert dann in einer 4-Bit-Speicherzelle gespeichert wird, fällt der Übertrag unter den Tisch, und der Wert 0 wird gespeichert.
- Fazit: Wenn wir in einem 4-Bit-System zu der Binärzahl 1111_2 den Wert 1 addieren, kommt 0 heraus. Somit ist 1111_2 auf einem 4-Bit-System die „natürliche“ Darstellung der Zahl -1 .

Allgemein gilt: Wenn eine vorzeichenbehaftete Zahl x überall dort Nullen hat, wo eine andere vorzeichenbehaftete Zahl y Einsen hat, setzt man $x = -(y + 1)$.

Diese Darstellung negativer Zahlen nennt man das „Zweierkomplement“.

Left board:

$$\begin{array}{r} 9+6 \\ 1001 \\ 0110 \\ \hline 1111 \end{array}$$

$$\begin{array}{r} 9+7 \\ 1001 \\ 0111 \\ \hline 10000 \\ \uparrow \\ C \quad S=0 \end{array}$$

Right board:

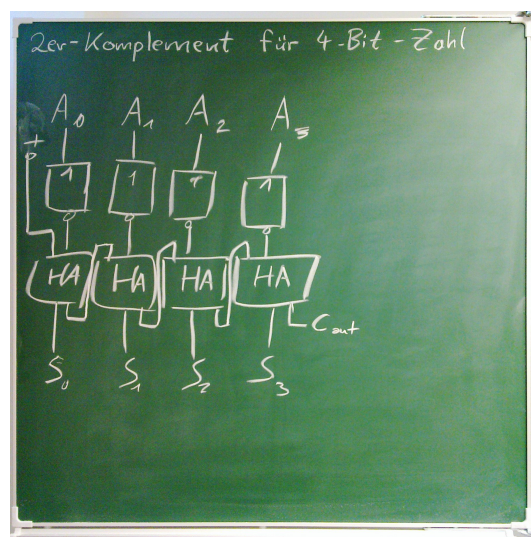
$$\begin{array}{r} 10-7 = 10+9 \\ \quad = 9 \\ 1010 \\ + 1001 \\ \hline 10011 \\ \uparrow \\ C \quad 3 \end{array}$$

2er-Komplement von 7 auf 4-Bit-Rechner

$$\begin{array}{r} 7 \quad 0111 \\ + 1000 \\ \hline 1001 \\ + 1 \\ \hline 1001 \end{array}$$

9 = -7

Auf einem 4-Bit-Addierer gilt: $-7 = 9$.



Schaltung, die das Zweierkomplement berechnet

Konsequenzen:

- Jede Binärzahl läßt sich sowohl als positive als auch als negative Zahl interpretieren.
- Um eine n -Bit-Binärzahl x in ihr Negatives $-x$ umzuwandeln, invertiert man zunächst alle Binärziffern. Danach addiert man, ohne auf Vorzeichen zu achten, den Zahlenwert 1. Falls dabei ein Überlauf auftritt, verwirft man diesen, behält also nur die untersten n Ziffern.
- Diejenige Zahl, die als höchste Ziffer eine 1 und ansonsten nur Nullen hat, hat keine negative Entsprechung außer sich selbst. Per Konvention kann man diese Zahl als größtmögliche positive Zahl, als kleinstmögliche negative Zahl, als alternative Darstellung der Zahl 0 oder als einen ungültigen Wert interpretieren.

Auf PCs lautet die Konvention: kleinstmögliche negative Zahl.

Somit zeigt dort das höchstwertige Bit einer Binärzahl das Vorzeichen an.

- Mit der o. a. Konvention haben

vorzeichenlose 8-Bit-Zahlen	den Wertebereich	0	bis	255,
vorzeichenbehaftete 8-Bit-Zahlen	den Wertebereich	-128	bis	127,
vorzeichenlose 16-Bit-Zahlen	den Wertebereich	0	bis	65535,
vorzeichenbehaftete 16-Bit-Zahlen	den Wertebereich	-32768	bis	32767,
...				
vorzeichenlose n -Bit-Zahlen	den Wertebereich	0	bis	$2^n - 1$,
vorzeichenbehaftete n -Bit-Zahlen	den Wertebereich	-2^{n-1}	bis	$2^{n-1} - 1$.

- Wenn bei einer Addition vorzeichenbehafteter Zahlen der Wertebereich verlassen wird, kann das Ergebnis negativ sein.

Beispiel: Addition der vorzeichenbehafteten 8-Bit-Zahlen 117 und 94

$$0111\ 0101_2 + 0101\ 1110_2 = 1101\ 0011_2 = -45$$

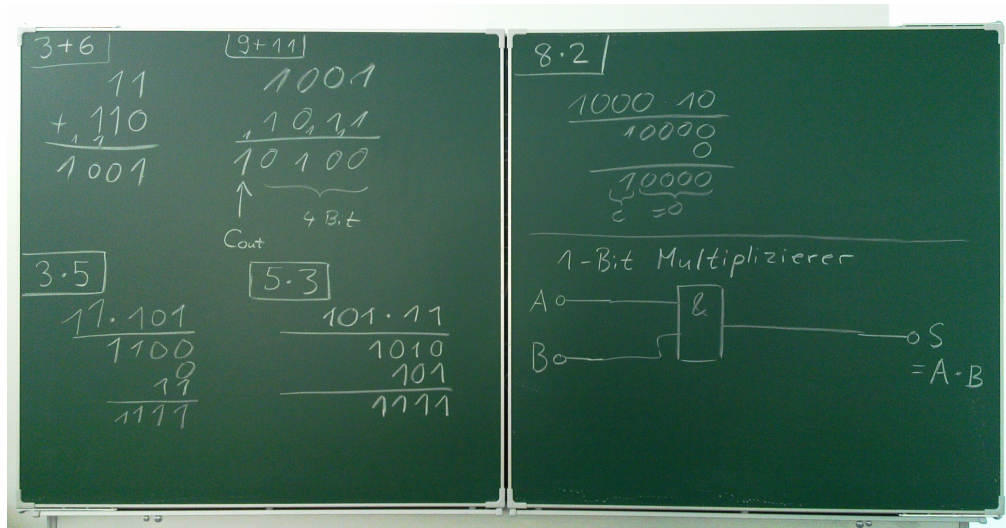
Probe: $45 + (117 + 94) = 256$. Die unteren 8 Bit von 256 sind Null.

Beispiel für 16-Bit-Zahlen: siehe <http://www.xkcd.com/571/>

2.5 Vom Addierer zum Computer

Arithmetisch-logische Einheit

- ALU = Arithmetic Logic Unit
- Schaltkreis, der die Operanden wahlweise auf verschiedene Weisen verknüpft (Oder, Und, Exklusiv-Oder, Addition, ...)
- Die Auswahl der Operation erfolgt über weitere Eingänge, die über zusätzliche Und-Gatter eins der Verknüpfungsergebnisse auf die Ausgänge legen.

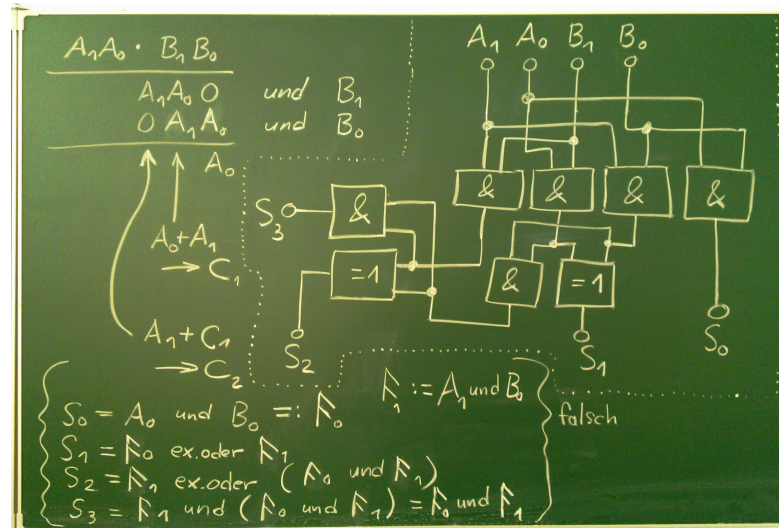


Multiplikation von Binärzahlen

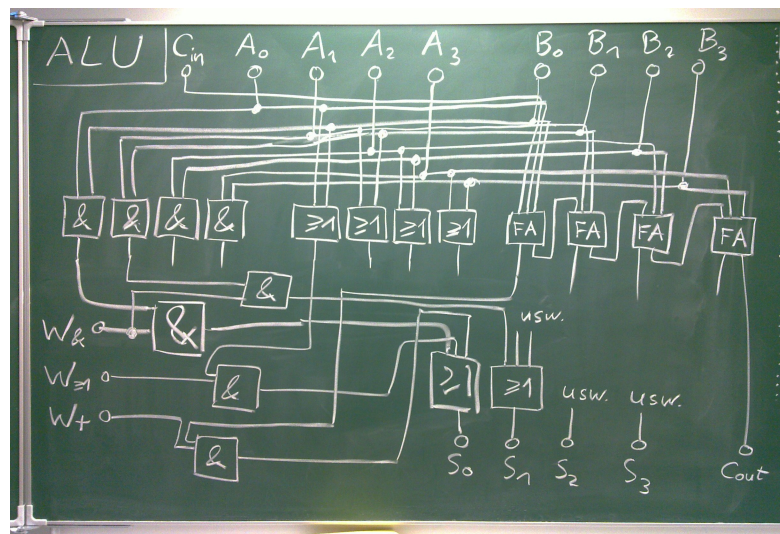
2-Bit-Multiplikierer

A ₁	A ₀	B ₁	B ₀	S ₃	S ₂	S ₁	S ₀
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	0	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0
1	0	1	0	0	0	0	0
1	0	1	1	0	0	0	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	0	0
1	1	1	0	0	0	0	0
1	1	1	1	0	0	0	0

Wahrheitstabelle für einen 2-Bit-Multiplikierer



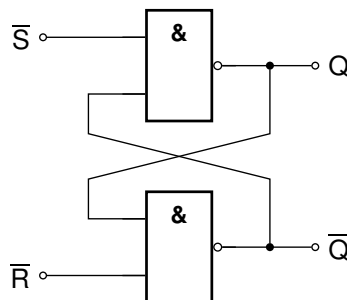
Realisierung eines 2-Bit-Multiplizierers



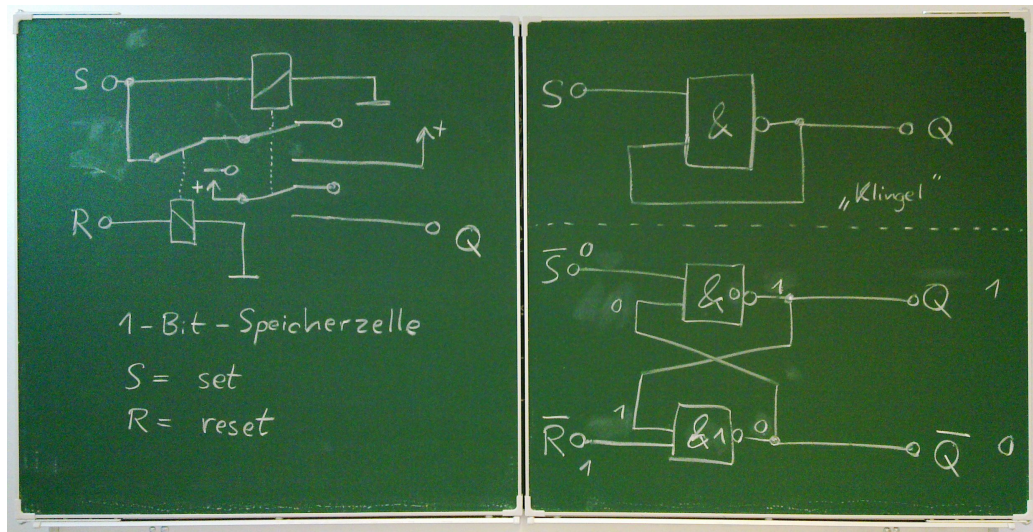
Die hier dargestellte 4-Bit-ALU berechnet auf Wunsch (W-Eingänge) die Und-Verknüpfung, die Oder-Verknüpfung oder die Summe der Operanden A und B.

Speicher

- Ein Flipflop kann sich eine 1 oder 0 (= 1 Bit Information) merken. („statischer Speicher“)



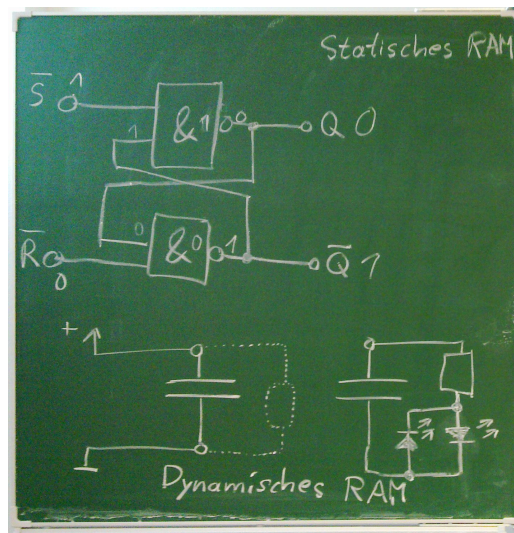
- Durch zusätzliche Und-Gatter kann man aus vielen Flip-Flops eins auswählen
→ Speicher-Adressierung
- Andere Speichermethode: Kondensator
→ muß nachgeladen werden („dynamischer Speicher“)
einfacher und dichter zu packen als statischer Speicher,
aber erfordert „Refresh“ (zusätzliche Schaltung)



Links: Realisierung einer Speicherzelle durch ein Relais: *Selbsthalteschaltung*

Rechts unten: Realisierung einer Speicherzelle durch zwei Und-Schaltungen: *Flip-Flop*

Rechts oben: instabile (und daher sinnlose) Schaltung mit einer Und-Nicht-Schaltung



Statischer Speicher (hier: Flip-Flop, oben) behält seinen Zustand, solange eine Stromversorgung besteht.

Dynamischer Speicher (Kondensator) „vergißt“ durch OHMsche Verluste seinen Zustand allmählich, daher ist ein regelmäßiges Auffrischen erforderlich: *Refresh*

Statischer Speicher ist schneller und leichter handhabbar als dynamischer Speicher. Auf der anderen Seite läßt sich dynamischer Speicher kostengünstiger und in weitaus höheren Dichten herstellen als statischer Speicher. Computer enthalten daher typischerweise geringe Mengen an statischem Speicher (Prozessor-Register, Cache-Speicher) und große Mengen an dynamischem Speicher (Arbeitsspeicher).

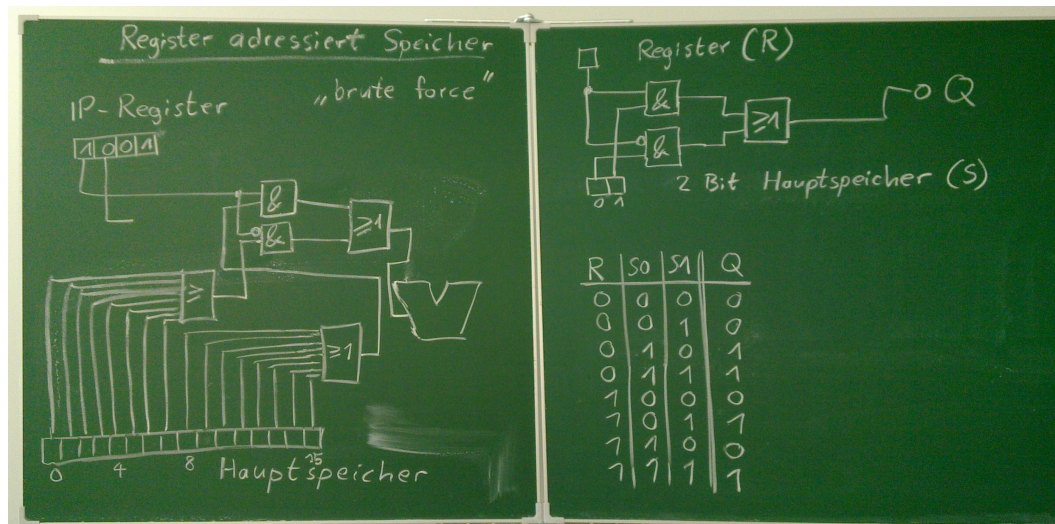
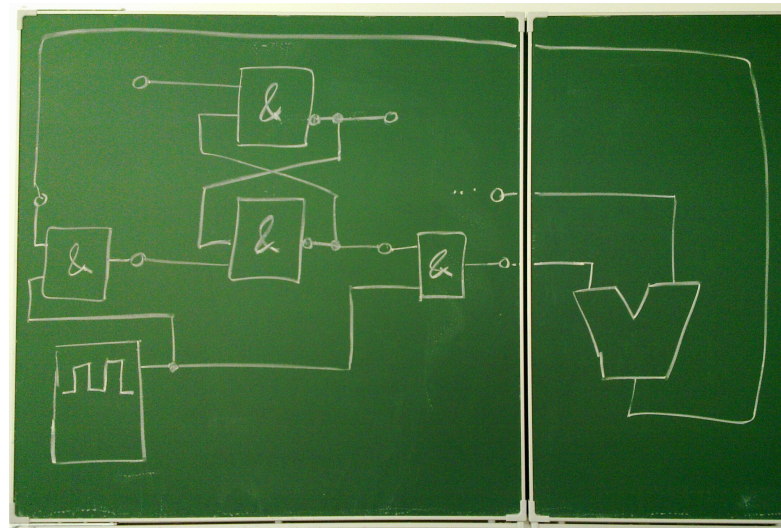
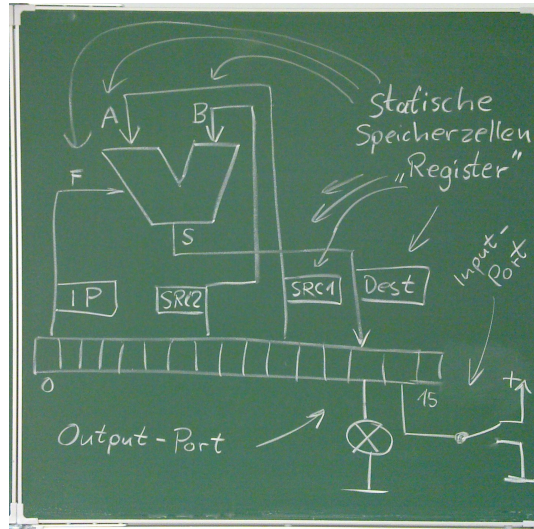
Takt

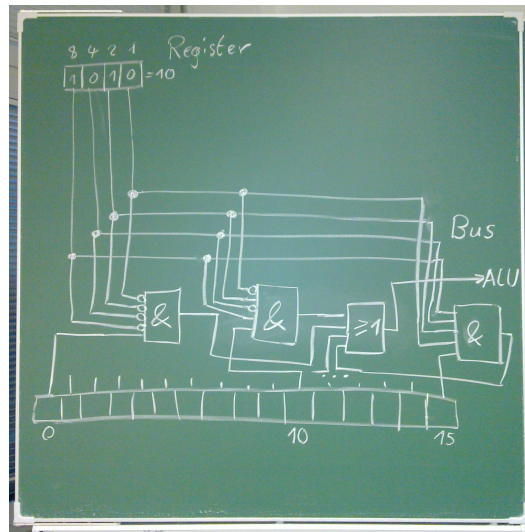
- Ein Taktgeber läßt ein Zählwerk laufen, das in jedem Takt eine andere Operation für die ALU auswählt und die Ein- und Ausgänge der ALU mit anderen Speicherzellen verbindet.
- Welche Operation und welche Speicherzellen jeweils verwendet werden, wird anhand des Inhalts weiterer Speicherzellen entschieden.
→ programmierbare Recheneinheit – „Prozessor“ (oder CPU für „Central Processing Unit“)
- Arbeitsweise eines Computers:
Ein Prozessor bearbeitet taktgesteuert einen Speicher gemäß Instruktionen, die er aus dem Speicher liest.

Ein- und Ausgabe

- Bestimmte Speicherzellen werden mit externen Geräten verknüpft.
- Gerät schreibt, Computer liest → Eingabe (Input)
- Computer schreibt, Gerät liest → Ausgabe (Output)

→ „I/O-Ports“





2.6 Computer-Sprachen

2.6.1 Maschinensprache

Ein auf die oben beschriebene Weise gebauter Computer kann Befehle ausführen, die in Gestalt von Nullen und Einsen vorliegen, bei denen es sich typischerweise um den Inhalt von Speicherzellen handelt (Flip-Flops, Kondensatoren oder Flash-Speicher – früher auch Lochkarten o. ä.).

```

31D6:0111 mov ax, 0e22
31D6:0114 mov [4041], ax
31D6:0117 ret
31D6:0118
-u 100
31D6:0100 B800B8      MOV     AX,B800
31D6:0103 8ED8        MOV     DS,AX
31D6:0105 B8410C      MOV     AX,0C41
31D6:0108 A30004      MOV     [0400],AX
31D6:010B B8620D      MOV     AX,0D62
31D6:010E A30204      MOV     [0402],AX
31D6:0111 B8220E      MOV     AX,0E22
31D6:0114 A30404      MOV     [0404],AX
31D6:0117 C3          RET
31D6:0118 AA          STOSB
31D6:0119 00508B      ADD     [BX+SI-75],DL
31D6:011C 86F8        XCHG   BH,AL
31D6:011E FD          STD
31D6:011F 050300      ADD     AX,0003
-g =100
Program terminated normally (0000)

```

Abbildung 15: Programm in Maschinensprache und Assembler

Abbildung 15 zeigt ein einfaches Programm für einen Intel-x86-kompatiblen 16-Bit-Mikroprozessor, geschrieben und angezeigt mit Hilfe des DOS-Programms [debug](#) (FreeDOS).

Sämtliche Zahlen sind *hexadezimal*, also zur Basis 16 notiert.

Die linke Spalte zeigt die *Speicheradresse*, d. h. die Position der angezeigten Zahlen und Befehle innerhalb des Arbeitsspeichers des Computers. Als Intel-spezifische Besonderheit unterteilt sich die Speicheradresse in einen Segment- und einen Offset-Anteil, die durch einen Doppelpunkt getrennt notiert werden (Details siehe unten).

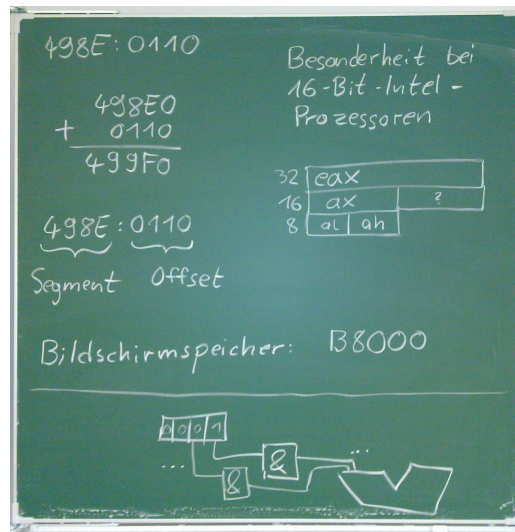
Neben der Speicheradresse wird der Inhalt der Speicherzelle als Zahl angezeigt.

Rechts daneben erscheint eine Übersetzung der Maschinensprache in menschenlesbare Abkürzungen, sog. *Mnemonics*. Diese Darstellung des Programms heißt *Assembler-Sprache* oder kurz *Assembler*.

Ein Programm, das Assembler-Sprache in Maschinensprache übersetzt, heißt ebenfalls *Assembler*.

2.6.2 Assembler

Die einfachsten Assembler-Befehle sind Kopieraktionen zwischen Speicherzellen. In Intel-Syntax steht der Befehl **mov A, B** für: „Kopiere den Inhalt der Speicherzelle B in Speicherzelle A.“ Bei den Speicherzellen kann es sich um adressierte Speicherzellen im Arbeitsspeicher handeln – in Intel-Syntax eine Offset-Adresse in eckigen Klammern – oder um Prozessor-Register – bei 16-Bit-Intel-Prozessoren: **ax, bx, cx, dx, si, di, bp, sp, cs, ds, es** und **ss**.



Die 16-Bit-Prozessoren der Intel-x86-Familie haben einen 20 Bit breiten **Adreßbus**, d. h. es stehen 20 Leitungen zur Verfügung, um eine Speicherzelle auszuwählen. Insgesamt können also $2^{20} = 1\,048\,576$ verschiedene Speicherzellen (ein binäres Megabyte – 1 MiB) ausgewählt werden.

Diese 20 Bit werden nun wie folgt in zweimal 16 Bit aufgeteilt: Die Zahl vor dem Doppelpunkt heißt **Segment-Adresse**, die danach **Offset-Adresse**. Die physikalische Adresse der gewünschten Speicherzelle lautet: $16 \cdot \text{Segment} + \text{Offset}$. Das Segment wird also um eine Hex-Ziffer nach links geschoben und anschließend das Offset addiert, um die physikalische Nummer der gewünschten Speicherzelle zu erhalten. Aus **31D6:0100** wird so beispielsweise die physikalische Adresse **31E60**.

Der Sinn dieser Aufteilung bestand darin, daß man erwartete, daß 16 Adreßleitungen, also maximal 64 kiB Speicher (64 binäre Kilobyte), für Computerprogramme ausreichend seien. In diesem Fall kann man die Segment-Adresse auf einen festen Wert setzen (in Abb. 15: **31D6**) und das Programm in 16-Byte-Schritten an beliebige Stellen im Speicher legen, ohne die im Programm verwendeten Offset-Adressen anpassen zu müssen.

In der Praxis erwiesen sich 64 kiB sehr bald als viel zu wenig, so daß Intel von dem Schema „16 · Segment + Offset“ wieder abrückte. Das Konzept „Segment:Offset“ wurde zwar beibehalten, erhielt jedoch eine andere Bedeutung. (Dazu später mehr.)

Eine weitere Besonderheit der 16-Bit-Prozessoren der Intel-x86-Familie ermöglicht uns, die Auswirkungen eines Programms direkt zu beobachten: Ab Speicherzelle **B8000** – bzw. **B800:0000** in Segment-Offset-Notation – befindet sich der **Bildschirmspeicher**. Werte, die in diesen Speicherzellen gespeichert sind, werden von der **VGA-Grafikkarte** im **Textmodus** direkt in Zeichen umgewandelt, die auf einem angeschlossenen Bildschirm angezeigt werden.

Alle geraden Speicheradressen (**0, 2, 4, 6, 8, A, C, E, 10, 12, ...**) enthalten die Zeichen (Buchstaben, Zahlen, Satzzeichen) in **ASCII**-Kodierung (American Standard Code for Information Interchange); Alle ungeraden Speicheradressen (**1, 3, 5, 7, 9, B, D, F, 11, 13, ...**) enthalten ein Farbschema (**Attribut**):

- 04: Vordergrundfarbe Rot
- 02: Vordergrundfarbe Grün
- 01: Vordergrundfarbe Blau
- 40: Hintergrundfarbe Rot
- 20: Hintergrundfarbe Grün
- 10: Hintergrundfarbe Blau
- 08: Vordergrundfarbe aufhellen
- 80: Blinken

Durch Addition (z. B. $04 + 02 + 08 = 0E$ mit der Bedeutung Rot + Grün + aufhellen = Gelb) ergeben sich insgesamt 16 Vordergrundfarben, 8 Hintergrundfarben und die Möglichkeit, die Zeichen blinken zu lassen.

Um nun von einem Maschinenprogramm aus Zeichen auf dem Bildschirm darzustellen, müssen wir die richtigen Werte an den richtigen Adressen in den Speicher schreiben. Hierzu betrachten wir noch einmal das Programm aus Abb. 15:

```
mov ax, b800
mov ds, ax
mov ax, 0c41
mov [0400], ax
mov ax, 0d62
mov [0402], ax
mov ax, 0e22
mov [0404], ax
ret
```

Der **ret**-Befehl am Ende steht für ein kontrolliertes Beenden des Programms (engl.: *return* – dazu später mehr).

Um auf den Bildschirmspeicher an der physikalischen Adresse **b8000** zugreifen zu können, müssen wir den Segment-Anteil **b800** dieser Adresse in das *Segment-Register ds* laden. Da der Prozessor dies nicht direkt erlaubt, nehmen wir einen Umweg über das *Allzweckregister ax*.

Anschließend schreiben wir an den Offset-Adressen **400**, **402** und **404** die Werte **0c40**, **0d62** bzw. **0e22** in den Arbeitsspeicher. Auch dies ist nicht direkt möglich, sondern erfolgt mit Umweg über das **ax**-Register.

Laut ASCII-Tabelle steht die Hexadezimalzahl **41** für ein großes A. Laut der o. a. Attribut-Tabelle steht **0c** für die Farbe **0c** = **08** + **04** = aufhellen + Rot = Hellrot. Der zusammengesetzte Wert **0c41** steht für ein rotes großes A, das an die Speicheradresse **b8400** geschrieben wird. Dies hat zur Folge, daß ein rotes großes A etwa mittig im oberen Drittel des Bildschirms sichtbar wird (siehe Abb. 15).

Analog sieht man, daß **0d62** für ein hellmagentafarbenes kleines b steht und **0e22** für ein gelbes Anführungszeichen. Diese beiden Zeichen werden in benachbarte Speicherzellen neben das große A geschrieben und werden entsprechend auf dem Bildschirm sichtbar.

2.6.3 Hochsprachen

Die Maschinensprache und die Assembler-Sprache haben den Nachteil, daß sie für jeden Prozessor völlig unterschiedlich aussehen. Jeder Prozessor hat seine eigenen Register und seine eigenen Befehle, die sich nicht wörtlich in Befehle eines anderen Prozessors übersetzen lassen.

Beispiel: Intel-x86-16-Bit-Assembler

- Lade- und Speicher-Befehle mov, ...
arithmetische Befehle add, sub, inc, dec, xor, cmp, ...
unbedingte und bedingte Sprungbefehle jmp, jz, jae, ...
- Register ax, bx, ...

Beispiel: Atmel-AVR-8-Bit-Assembler

- Lade- und Speicher-Befehle ldi, lds, sti, ...
arithmetische Befehle add, sub, subi, eor, cp, ...
unbedingte und bedingte Sprungbefehle rjmp, brsh, brlo, ...
- Register r0, r1, ...

Um Programme nicht für jeden Prozessor neu schreiben zu müssen, wurden Hochsprachen entwickelt. Hierfür gibt es mehrere Konzepte:

Compiler-Sprachen

- Ein *Compiler* übersetzt einen Hochsprachen-*Quelltext* in die Assembler-Sprache.

- Ein *Assembler* übersetzt den Assembler-Quelltext in die Maschinensprache.
- Compiler und Assembler sind Programme, geschrieben in Maschinensprache, Assembler oder einer Hochsprache.
- Beispiele: Fortran, Algol, Pascal, Ada, C, C++, ...

Interpreter- oder Skript-Sprachen

- Ein *Interpreter* liest einen Hochsprachen-*Quelltext* und führt ihn sofort aus.
- Der Interpreter ist ein Programm, geschrieben in Maschinensprache, Assembler oder einer Hochsprache.
- Beispiele: Unix-Shell, BASIC, Perl, Python, ...

Kombinationen

- Ein *Compiler* erzeugt einen *Zwischencode* für eine *virtuelle Maschine*.
- Die virtuelle Maschine ist ein *Interpreter*. Sie liest den Zwischencode und führt ihn sofort aus.
- Beispiele: UCSD-Pascal, Java, ...

Ein wichtiges Beispiel für eine Compiler-Sprache ist die Hochsprache *C*.

Mit dem Programm *gcc* kann man wie folgt C nach Assembler übersetzen:

```
$ gcc -S pruzzel.c
```

Dieser Aufruf erzeugt aus dem C-Quelltext *pruzzel.c* einen Assembler-Quelltext *pruzzel.s* für den Standard-Prozessor, z. B. für die 32-Bit-Intel-Architektur (IA-32).

Mit *avr-gcc* anstelle von *gcc* erzeugt man auf einem beliebigen Prozessor Assembler für 8-Bit-Atmel-AVR-Prozessoren:

```
$ avr-gcc -S pruzzel.c
```

(Die mit diesem Aufruf erzeugte Datei *pruzzel.s* heißt in den Beispielen *pruzzel-avr.s*.)

Wenn man in einer Hochsprache programmiert, ist es normalerweise nicht erforderlich, sich mit dem erzeugten Assembler-Quelltext zu befassen.

Für Fälle, in denen dies doch notwendig wird, bieten viele Fehlersuchprogramme – *Debugger* – die Möglichkeit an, die Ausführung des Programms parallel im C- und im Assembler-Quelltext zu verfolgen. Der GNU-Debugger *gdb* führt z. B. mit dem Befehl *next* eine Zeile C-Quelltext aus und mit dem Befehl *nexti* (*next instruction*) eine Zeile Assembler-Quelltext:

```
$ gcc -g pruzzel.c -o pruzzel
$ gdb -tui ./pruzzel
(gdb) break main
(gdb) run
(gdb) layout split
(gdb) nexti
```

2.7 Struktur von Assembler-Programmen

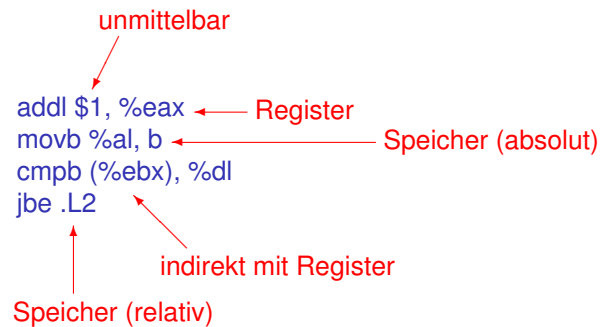
Der grundsätzliche Aufbau von Assembler-Befehlen besteht normalerweise aus dem Schlüsselwort (Mnemonic) für den eigentlichen Befehl, ggf. gefolgt von einem oder mehreren *Operanden*. (Beispiel: IA-32-Assembler)

Befehl Operanden

```
addl $1, %eax
movb %al, b
cmpb (%ebx), %dl
jbe .L2
```

Bei den Operanden kann es sich um Zahlen handeln, die direkt im Befehl stehen (*unmittelbar*) oder sich in einem Prozessorregister oder einer Speicherzelle befinden. Die Nummer der Speicherzelle kann *direkt* im Befehl stehen, oder sie wird *indirekt* einem Prozessorregister entnommen. Außerdem kann die Nummer einer Speicherzelle *absolut* angegeben werden oder *relativ* zu derjenigen Speicherzelle, die den Befehl enthält.

Diese verschiedenen Arten, Operanden auszuwählen, heißen *Adressierungsarten*.



Ein besonders illustratives Beispiel für eine Assembler-Sprache ist der *Redcode*. Diese Sprache wurde für das Spiel *Core Wars* (oder *Core War* – Krieg der Kerne) entwickelt. Sie läuft auf einer virtuellen Maschine namens *Memory Array Redcode Simulator (MARS)*.

In der älteren standardisierten Version (ICWS '88) lautet die vollständige Beschreibung von Redcode:

Instruktionen:

`dat B` – Daten
`mov A, B` – kopiere A nach B
`add A, B` – addiere A zu B
`sub A, B` – subtrahiere A von B
`jmp A` – unbedingter Sprung nach A
`jmz A, B` – Sprung nach A, wenn B = 0
`jmn A, B` – Sprung nach A, wenn B ≠ 0
`djn A, B` – „decrement and jump if not zero“
`cmp A, B` – „compare“: überspringe, falls gleich
`spl A` – „split“: Programm verzweigen

Adressierungsarten:

grundsätzlich: Speicher relativ

`#` – unmittelbar
`$` (oder ohne Symbol) – direkt
`@` – indirekt (B-Feld)
`<` – indirekt mit Prä-Dekrement

- Der Speicher ist ringförmig.
- Anstelle von Prozessorregistern können grundsätzlich alle Speicherzellen für jede Operation verwendet werden.
- Adressierung von Speicherzellen erfolgt grundsätzlich relativ.
- Ein Programm, das einen `dat`-Befehl ausführt, wird beendet.
- Der `spl`-Befehl bewirkt, daß ein Programm *sowohl* an der als Operand angegebenen Speicherzelle *als auch* in der nächsten Speicherzelle ausgeführt wird. (Ausführung abwechselnd)
- In einem „Kampf“ treten zwei (oder mehr) Programme in demselben Speicher gegeneinander an. Ein Programm, das durch `dat`-Befehle vollständig beendet wurde, hat verloren.

Beispiel: Programm *Nothing*

`jmp 0`

Dieses Programm führt eine Endlosschleife aus. Da es nur eine einzige Speicherzelle belegt und noch dazu den numerischen Wert Null hat, wird es von gegnerischen Programmen leicht übersehen, was oft zu einem Unentschieden führt. Gegen Gegner, die dann stattdessen im ringförmigen Speicher sich selbst entdecken und angreifen, kann *Nothing* sogar gewinnen.

Beispiel: Programm *Knirps*

`mov 0, 1`

Dieses ebenfalls nur eine einzige Speicherzelle große Programm kopiert sich selbst vollständig auf die nächste Speicherzelle und führt diese anschließend aus. Es bewegt sich somit mit der höchsten lückenlos erreichbaren Geschwindigkeit („Lichtgeschwindigkeit“) durch den Speicherring. Wenn es dabei auf einen Gegner trifft, überschreibt es diesen, so daß dieser danach ebenfalls `mov 0, 1` ausführt. Da danach keiner der Gegner mehr in der Lage ist, den anderen durch einen `dat`-Befehl zu beenden, führt dies zu einem Unentschieden.

Beispiel: Programm *Mice*

```
ptr    dat #0
start  mov #12, ptr
loop   mov @ptr, <dest
       djn loop, ptr
       spl @dest
       add #653, dest
       jnz start, ptr
dest    dat #833
       end start
```

Dieses Programm erzeugt in einer Schleife (`djn loop, ptr`) eine Kopie von sich selbst und aktiviert diese anschließend mit einem `spl`-Befehl. Danach erzeugen beide „Mäuse“ weitere Kopien von sich selbst.

Wenn auf diese Weise ein Gegner überschrieben wird, findet dieser am Ende des Programms den Befehl `jnz start, ptr` vor, gefolgt von einem `dat`-Befehl. Der Sprung findet also nur dann statt, wenn die – relativ adressierte – Variable `ptr` den Wert Null hat. Dies ist normalerweise nur dann kurz der Fall, wenn der Kopiervorgang abgeschlossen ist und der `spl`-Befehl ausgeführt wird; direkt danach wird `ptr` durch `mov #12, ptr` auf einen Wert ungleich Null gesetzt. Überschriebene Gegner werden daher normalerweise durch den Befehl `dest dat #833` beendet.

Durch seine Kopier-Taktik ist Mice jedem Gegner klar überlegen, der versucht, es mit `dat`-Befehlen zu überschreiben. Durch seine Langsamkeit hingegen erreicht Mice gegen Knirps praktisch immer nur ein Unentschieden.

Als wirksamste Taktik gegen Mice hat sich die *Zeitfalle* erwiesen: Ein in Mice hineingeschriebener Befehl `spl 0` bewirkt, daß sich die betroffene Kopie von Mice bis zur Maximalgrenze aufspaltet und damit die Rechenzeit seiner anderen Kopien herabsetzt. Ein auf diese Weise „gelähmtes“ Mice kann anschließend problemlos mit `dat`-Befehlen überschrieben werden.

3 Architekturmerkmale von Prozessoren

3.1 Speicherarchitekturen

Bezeichnungen

- *Byte* = Zusammenfassung mehrerer Bits zu einer Binärzahl, die ein Zeichen („Character“) darstellen kann, häufig 8 Bits („Oktett“)
- *Speicherwort* = Zusammenfassung mehrerer Bits zu der kleinsten adressierbaren Einheit, häufig 1 Byte
- *RAM* = Random Access Memory = Hauptspeicher
- *ROM* = Read Only Memory = nur lesbarer Speicher

Verschiedene Arten von Speicher

- Prozessor-Register
können direkt mit ALU verbunden werden, besonders schnell (Flipflops), überschaubare Anzahl von Registern
- Hauptspeicher
kann direkt adressiert und mit Prozessor-Registern abgeglichen werden, heute i. d. R. dynamischer Speicher (Kondensatoren)
- I/O-Ports
sind spezielle Speicheradressen, über die mit externen Geräten kommuniziert wird
- Massenspeicher
liegt auf externem Gerät, wird über I/O-Ports angesprochen, Festplatte, Flash-Speicher, ...

Speicherarchitekturen

- *Von-Neumann-Architektur*

Es gibt nur 1 Hauptspeicher, in dem sich sowohl die Befehle als auch die Daten befinden.

Vorteil: Flexibilität in der Speichernutzung

Nachteil: Befehle können überschrieben werden. → Abstürze und Malware möglich

- *Harvard-Architektur*

Es gibt 2 Hauptspeicher. In einem befinden sich die Befehle, im anderen die Daten.

Vorteil: Befehle können nicht überschrieben werden → sicherer als Von-Neumann-Architektur

Nachteile: Leitungen zum Speicher (Bus) müssen doppelt vorhanden sein, freier Befehlsspeicher kann nicht für Daten genutzt werden.

- Weitere Kombinationen:

Hauptspeicher und I/O-Ports gemeinsam oder getrennt,

Hauptspeicher und Prozessorregister gemeinsam oder getrennt

Beispiele:

- Intel IA-32 (i386, Nachfolger und Kompatible):
Von-Neumann-Architektur (plus Speicherschutzmechanismen),
Prozessorregister und I/O-Ports vom Hauptspeicher getrennt
- Atmel AVR (z. B. ATmega32):
Harvard-Architektur (Befehlsspeicher als Flash-Speicher grundsätzlich auch schreibbar),
Prozessorregister und I/O-Ports in gemeinsamem Adressbereich mit Hauptspeicher
- 6502 (heute: Renesas-Mikro-Controller):
Von-Neumann-Architektur,
I/O-Ports in gemeinsamem Adressbereich mit Hauptspeicher,
Prozessorregister und Hauptspeicher getrennt

3.2 Registerarchitekturen

- Mehrere Register, einzeln ansprechbar
- Akkumulator: Nur 1 Register kann rechnen.
- Stack-Architektur: Stapel, „umgekehrte Polnische Notation“

Je nachdem, auf welche dieser Arten die Register eines Prozessors organisiert sind, muß er auf völlig unterschiedliche Weise programmiert werden.

Beispiele:

- Intel IA-32 (i386, Nachfolger und Kompatible):
Mehrere Register, für verschiedene Zwecke spezialisiert (unübersichtlich),
Fließkommaregister: Stack-Architektur
- Atmel AVR (z. B. ATmega32):
32 Register
- 6502 (heute: Renesas-Mikro-Controller):
3 Register: A, X, Y. Nur A kann rechnen → Akkumulator
- Java Virtual Machine (JVM):
Stack-Architektur

3.3 Befehlssätze

Um die manuelle Programmierung in Assembler möglichst komfortabel zu gestalten, wurden in der Vergangenheit viele Prozessoren mit umfangreichen Befehlssätzen ausgestattet. Diese Architektur heißt *Complex Instruction Set Computer (CISC)*.

Realisiert wird CISC durch einen „Prozessor im Prozessor“, der kleine Programme, sog. *Mikroprogramme* ausführte.

Erkauft wird der Komfort durch eine längere Abarbeitungszeit der einzelnen Befehle.

Beispiel: IA-32

Das entgegengesetzte Konzept heißt *Reduced Instruction Set Computer (RISC)*. In einem RISC-Prozessor ist der einzelne Maschinenbefehl weit weniger mächtig als in einem CISC-Prozessor, wird dafür aber wesentlich schneller ausgeführt, idealerweise jeder Befehl in einem einzigen Taktzyklus.

Beispiel: Atmel AVR

Weitere Befehlssatzarchitekturen:

Very Long Instruction Word (VLIW), *Explicitly Parallel Instruction Computing (EPIC)*
(z. B. IA-64)

4 Der CPU-Stack

Achtung: Nicht mit dem Register-Stack verwechseln!

4.1 Implementation

Ein Teil des Speichers wird als „Stack“ reserviert.

Eine Variable, typischerweise ein Prozessorregister, wird als „Stack-Pointer“ (SP) reserviert.

- Um einen Wert auf den Stack zu legen, verschiebt man den SP, so daß er auf einen freien Speicherplatz zeigt. Dorthin speichert man den Wert.
- Um einen Wert vom Stack zurückzuholen, liest man ihn von der Stelle, auf die SP zeigt, und verschiebt anschließend den SP in die entgegengesetzte Richtung.

Da diese Kombinationen von Operationen sehr gebräuchlich sind, enthalten die allermeisten Prozessoren eigene Befehle dafür, z. B.:

- Wert auf den Stack legen: `push`
- Wert vom Stack zurückholen: `pop`
- IP auf den Stack sichern und Unterprogramm aufrufen: `call`
- IP vom Stack zurückholen = Rücksprung aus Unterprogramm: `ret`

Die Richtung, in der der Stack wächst, ist frei wählbar. Beim IA-32 z. B. wächst der Stack von oben nach unten. Dort wird also beim `push` der SP dekrementiert (erniedrigt) und beim `pop` inkrementiert (erhöht).

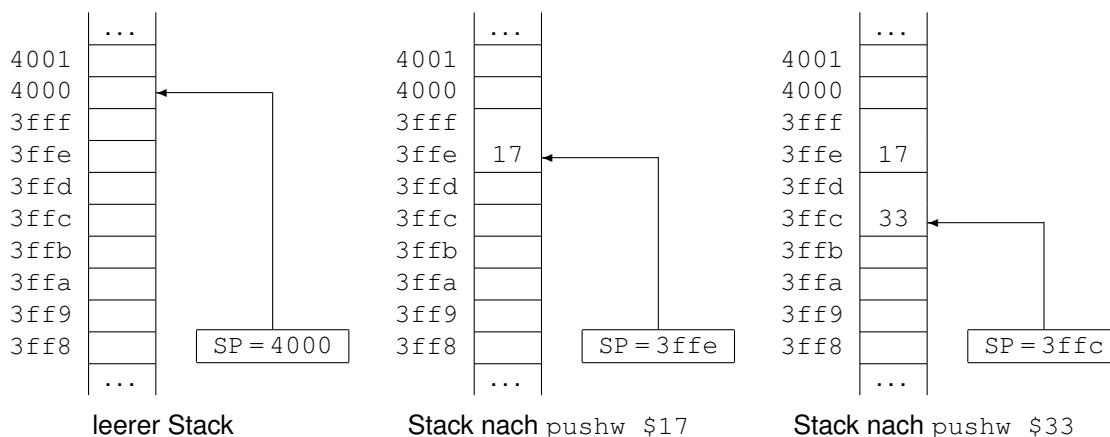
Auch die kleinste Einheit ist frei wählbar. Beim IA-32 können minimal 16 Bit, also immer 2 Bytes zusammen auf den Stack gelegt werden.

Beispiel: Ein für einen IA-32-Prozessor geschriebenes Programm enthält die Befehle:

```
pushw $17
pushw $33
...
popw %ax
```

Bei Programmbeginn werden 16384 Bytes, hexadezimal 4000, für den Stack reserviert, nämlich der Speicherbereich von Adresse 0000 bis Adresse 3fff, und der Stack-Pointer (SP) wird auf den Wert 4000 initialisiert. Er zeigt also am Stack vorbei „ins Leere“.

Wenn nun der Befehl `pushw $17` („push word“) ausgeführt wird, wird SP um 2 auf 3ffe dekrementiert und anschließend der Wert 17 an die beiden Speicheradressen 3ffe und 3fff geschrieben.



Wenn danach der Befehl `pushw $33` ausgeführt wird, wird der Wert 33 an die Speicheradresse 3ffe geschrieben und SP um 2 auf 3ffc dekrementiert.

Wenn anschließend der `popw`-Befehl ausgeführt wird, wird SP um 2 auf 3ffe inkrementiert, und der Stack sieht wieder genauso aus wie nach dem Befehl `pushw $17`.

4.2 Unterprogramme

Um einem Unterprogramm Parameter zu übergeben, müssen diese an einer definierten Stelle gespeichert werden. Mögliche Speicherorte sind:

- Prozessorregister
- Stack

Die Parameterübergabe funktioniert nur, wenn sich Haupt- und Unterprogramm über deren Art und Weise einig sind. Daher existieren mehrere Aufrufkonventionen:

- Das Hauptprogramm speichert die Parameter auf dem Stack. Das Unterprogramm liest sie. Das Hauptprogramm nimmt die Parameter wieder vom Stack. („C-Aufrufkonvention“)
- Das Hauptprogramm speichert die Parameter auf dem Stack. Das Unterprogramm liest sie und nimmt sie wieder vom Stack. („Pascal-Aufrufkonvention“)

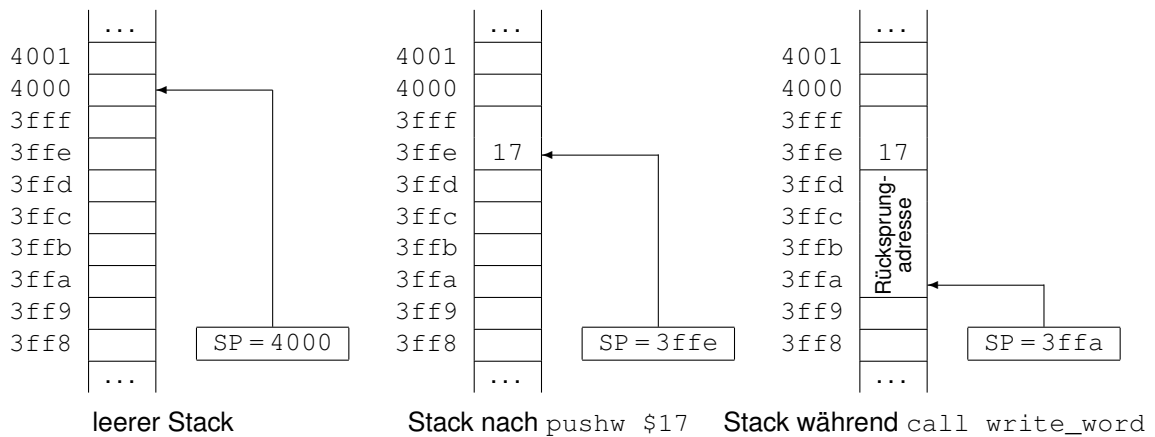
Bemerkung: Trotz der Benennung nach Programmiersprachen sind die Aufrufkonventionen nicht an die jeweiligen Programmiersprachen gebunden. (Zum Beispiel verwendet GNU-Pascal standardmäßig die „C“-Aufrufkonvention.)

Die „stdcall“-Aufrufkonvention unter Microsoft Windows ist eine Variante der Pascal-Aufrufkonvention.

Beispiel: Ein für einen IA-32-Prozessor geschriebenes Programm soll gemäß der C-Aufrufkonvention eine Funktion `write_word()` aufrufen, die als Parameter eine 16-Bit-Ganzzahl erwartet, z. B. den Wert 17. Dies kann folgendermaßen geschehen:

```
pushw $17
call write_word
popw %ax
```

Das `popw` dient dazu, den Stack nach dem Aufruf wieder in den ursprünglichen Zustand zurückzusetzen; der in das `%ax`-Register gelesene Wert wird verworfen. (Stattdessen könnte man auch mit `addw %sp, 2` den Stack-Pointer „von Hand“ inkrementieren.)



Die Rücksprungadresse ist bei IA-32 vier Bytes groß, verbraucht also vier Bytes auf dem Stack.

Während das Unterprogramm läuft, kann es den übergebenen Parameter (Wert: 17) unter der Adresse „Stack-Pointer + 4 Bytes“ ansprechen, in Assembler: `4(%esp)`

4.3 Registerinhalte sichern

Wenn das Hauptprogramm ein Unterprogramm aufruft, können sich dadurch die Inhalte der Register ändern, nämlich dann, wenn das Unterprogramm diese Register benutzt.

Damit dies nicht zu falschen Rechenergebnissen führt, muß entweder das Hauptprogramm diesen Umstand berücksichtigen, oder das Unterprogramm muß Register vor der Verwendung sichern und hinterher wiederherstellen – üblicherweise im CPU-Stack.

Dies ist insbesondere dann wichtig, wenn das Unterprogramm nicht über einen ausdrücklichen Befehl (`call`), sondern auf andere Weise aufgerufen wird.

5 Hardwarenahe Programmierung

5.1 Bit-Operationen

5.1.1 Zahlensysteme

Dezimalsystem

- Basis: 10
- Gültige Ziffern: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

137

Einer: $7 \cdot 10^0$
Zehner: $3 \cdot 10^1$
Hunderter: $1 \cdot 10^2$

$$137_{10} = 1 \cdot 10^2 + 3 \cdot 10^1 + 7 \cdot 10^0 = 100 + 30 + 7 = 137$$

137
+ 42

179

Hexadezimalsystem

- Basis: 16
- Gültige Ziffern: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

137

$7 \cdot 16^0$
 $3 \cdot 16^1$
 $1 \cdot 16^2$

$$137_{16} = 1 \cdot 16^2 + 3 \cdot 16^1 + 7 \cdot 16^0 = 256 + 48 + 7 = 311$$

A380
+ B747
+----
15AC7

- Schreibweise in C: `0x137`

Oktalsystem

- Basis: 8
- Gültige Ziffern: 0, 1, 2, 3, 4, 5, 6, 7

137

$7 \cdot 8^0$
 $3 \cdot 8^1$
 $1 \cdot 8^2$

$$137_8 = 1 \cdot 8^2 + 3 \cdot 8^1 + 7 \cdot 8^0 = 64 + 24 + 7 = 95$$
$$42_8 = 4 \cdot 8^1 + 2 \cdot 8^0 = 32 + 2 = 34$$
$$201_8 = 2 \cdot 8^2 + 0 \cdot 8^1 + 1 \cdot 8^0 = 128 + 1 = 129$$

137
+ 42

111
201

- Schreibweise in C: `0137`

Rechner für beliebige Zahlensysteme: GNU bc

```
$ bc
ibase=8
137      ← Eingabe zur Basis 8
95       ← Ausgabe zur Basis 10
obase=10 ← Eingabe zur Basis 8 (108 = 8)
137 + 42
201      ← Ausgabe zur Basis 8
```

Binärsystem

- Basis: 2
- Gültige Ziffern: 0, 1

110

110
+ 1100

11
10010

$0 \cdot 2^0$
 $1 \cdot 2^1$
 $1 \cdot 2^2$

$$110_2 = 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 4 + 2 + 0 = 6$$

- Binär-Zahlen ermöglichen es, elektronisch zu rechnen ...
- und mehrere „Ja/Nein“ (Bits) zu einer einzigen Zahl zusammenzufassen.
- **Oktal- und Hexadezimal-Zahlen lassen sich ziffernweise in Binär-Zahlen umrechnen:**

000 0	0000 0	1000 8
001 1	0001 1	1001 9
010 2	0010 2	1010 A
011 3	0011 3	1011 B
100 4	0100 4	1100 C
101 5	0101 5	1101 D
110 6	0110 6	1110 E
111 7	0111 7	1111 F

Beispiel: $1101011_2 = 153_8 = 6B_{16}$

Anwendungsbeispiel: Oktal-Schreibweise für Unix-Zugriffsrechte

$-rw-r----- = 0110100000_2 = 640_8$ $\$ chmod 640 file.c$
 $-rwxr-x--- = 0111101000_2 = 750_8$ $\$ chmod 750 subdir$

IP-Adressen (IPv4)

- Basis: 256
- Gültige Ziffern: 0 bis 255, getrennt durch Punkte
- Kompakte Schreibweise für Binärzahlen mit 32 Ziffern (Bits)

192.168.0.1

$1 \cdot 256^0$
 $0 \cdot 256^1$
 $168 \cdot 256^2$
 $192 \cdot 256^3$

$$192.168.0.1_{256} = 11000000101010000000000000000001_2$$

5.1.2 Bit-Operationen in C

<pre> 0110 + 1100 ----- 10010 </pre>	<pre> 0110 1100 ----- 1110 </pre>	<pre> 0110 & 1100 ----- 0100 </pre>	<pre> 0110 ^ 1100 ----- 1010 </pre>	<pre> ~ 1100 ----- 0011 </pre>	<pre> 0110 >> 2 ----- 0001 </pre>
Addition	Oder	Und	Exklusiv-Oder	Negation	Bit-Verschiebung

<pre> 01101100 00000010 ----- 01101110 </pre>	<pre> 01101100 & 11110111 ----- 01100100 </pre>	<pre> 01101100 ^ 00010000 ----- 01111100 </pre>
Bit gezielt setzen	Bit gezielt löschen	Bit gezielt umklappen

- Bits werden häufig von rechts und ab 0 numeriert (hier: 0 bis 7), um die Maskenerzeugung mittels Schiebeoperatoren zu erleichtern.
- Die Bit-Operatoren (z. B. & in C) wirken jeweils auf alle Bits der Zahlen. Die logischen Operatoren (z. B. && in C) prüfen die Zahl insgesamt auf $\neq 0$. Nicht verwechseln!

6 & 12 == 4
6 && 12 == 1

Anwendung: Bit 2 (also das dritte Bit von rechts) in einer 8-Bit-Zahl auf 1 setzen:

<pre> 00000001 << 2 ----- 00000100 </pre>	<pre> 01101100 00000100 ----- 01101100 </pre>
Maske für Bit 2	Bit gezielt setzen

- Schreibweise in C: `a |= 1 << 2;`

Anwendung: Bit 2 in einer 8-Bit-Zahl auf 0 setzen:

<pre>00000001 << 2 ----- 00000100</pre>	<pre>~ 00000100 ----- 11111011</pre>	<pre>01101100 & 11111011 ----- 01101000</pre>
Maske zum Löschen von Bit 2 erzeugen		Bit gezielt löschen

- Schreibweise in C: `a &= ~(1 << 2);`

Anwendung: Bit 2 aus einer 8-Bit-Zahl extrahieren:

<pre> 00000001 << 2 ----- 00000100 </pre>	<pre> 01101100 & 00000100 ----- 00000100 </pre>	<pre> 00000100 >> 2 ----- 00000001 </pre>
Maske für Bit 2	Bit 2 isolieren	in Zahl 0 oder 1 umwandeln

- Schreibweise in C: `x = (a & (1 << 2)) >> 2;`

Beispiel: Netzmaske für 256 IP-Adressen

192.168. 1.123	← IP-Adresse eines Rechners
& 255.255.255. 0	← Netzmaske: 255 = 11111111 ₂
<hr/>	
192.168. 1. 0	← IP-Adresse des Sub-Netzes

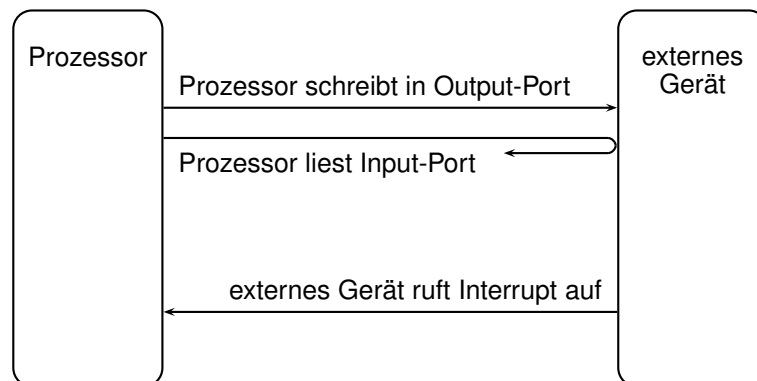
Beispiel: Netzmaske für 8 IP-Adressen

192.168. 1.123	← IP-Adresse eines Rechners	01111011
& 255.255.255.248	← Netzmaske	& 11111000
<hr/>		<hr/>
192.168. 1.120	← IP-Adresse des Sub-Netzes	01111000

5.2 I/O-Ports

Es gibt drei grundlegende Mechanismen für die Kommunikation zwischen dem Prozessor und einem externen Gerät:

- Über Output-Ports kann der Prozessor das Gerät aktiv steuern,
- über Input-Ports kann er es aktiv abfragen,
- und über Interrupts kann das externe Gerät im Prozessor Aktivitäten auslösen.



Input- und Output-Ports, zusammengefaßt: I/O-Ports, sind spezielle Speicherzellen, die mit einem externen Gerät verbunden sind (siehe Seite 22).

- Ein in einen Output-Port geschriebener Wert bewirkt eine Spannungsänderung in einer Leitung, die zu einem externen Gerät führt.
- Wenn ein externes Gerät eine Spannung an eine Leitung anlegt, die zu einer Speicherzelle führt, kann der Prozessor diese als Input-Port lesen.

Um z. B. auf einen Druck auf einen Taster zu warten, kann ein Program periodisch in einer Schleife einen Input-Port lesen und die Schleife erst dann beenden, wenn der Wert für „Taster gedrückt“ gelesen wurde.

Diese Methode heißt „Busy Waiting“: Der Prozessor ist vollständig mit Warten beschäftigt. Wenn gleichzeitig noch andere Aktionen stattfinden sollen, müssen diese in der Schleife mit berücksichtigt werden.

Beispiel für die Verwendung eines Output-Ports: Roboter-Steuerung

Datei: RP6Base/RP6Base_Examples/RP6Examples_20080915/RP6Lib/RP6base/RP6RobotBaseLib.c

Suchbegriff: setMotorDir


```

void setMotorDir(uint8_t left_dir, uint8_t right_dir)
{
    mleft_dir = left_dir;
    mright_dir = right_dir;
    mleft_des_dir = left_dir;
    mright_des_dir = right_dir;
    if(left_dir)
        PORTC |= DIR_L;
    else
        PORTC &= ~DIR_L;
    if(right_dir)
        PORTC |= DIR_R;
    else
        PORTC &= ~DIR_R;
}

```

Die Variable `PORTC` ist ein Output-Port. Durch Manipulation einzelner Bits in dieser Variablen ändert sich die Spannung an den elektrischen „Beinchen“ des Mikro-Controllers. Hierdurch wird die Beschaltung von Elektromotoren umgepolt.

(Die Konstanten `DIR_L` und `DIR_R` sind „Bitmasken“, d. h. Zahlen, die in ihrer Binärdarstellung nur eine einzige 1 und ansonsten Nullen haben. Durch die Oder- und Und-Nicht-Operationen werden einzelne Bits in `PORTC` auf 1 bzw. 0 gesetzt.)

Die direkte Ansteuerung von I/O-Ports ist nur auf Mikro-Controllern üblich. Auf Personal-Computern erfolgt die gesamte Ein- und Ausgabe über Betriebssystem-„Treiber“. Anwenderprogramme greifen dort i. d. R. nicht direkt auf I/O-Ports zu.

5.3 Interrupts

Ein Interrupt ist ein Unterprogramm, das nicht durch einen Befehl (`call`), sondern durch ein externes Gerät (über ein Stromsignal) aufgerufen wird.

Damit dies funktioniert, muß die Adresse, an der sich das Unterprogramm befindet, an einer jederzeit auffindbaren Stelle im Speicher hinterlegt sein. Diese Stelle heißt „Interrupt-Vektor“.

Da ein Interrupt jederzeit erfolgen kann, hat das Hauptprogramm keine Chance, vor dem Aufruf die Registerinhalte zu sichern. Für Interrupt-Unterprogramme, sog. Interrupt-Handler, ist es daher zwingend notwendig, sämtliche Register vor Verwendung zu sichern und hinterher zurückzuholen.

Beispiel für die Verwendung eines Interrupts: Roboter-Steuerung

Datei: `RP6Base/RP6Base_Examples/RP6Examples_20080915/RP6Lib/RP6base/RP6RobotBaseLib.c`

Suchbegriff: ISR

```

ISR (INT0_vect)
{
    mleft_dist++;
    mleft_counter++;
    /* ... */
}

```

- Durch das Schlüsselwort `ISR` anstelle von z. B. `void` teilen wir dem Compiler mit, daß es sich um einen Interrupt-Handler handelt, so daß er entsprechenden Code zum Sichern der Registerinhalte einfügt.
- Durch die Namensgebung `INT0_vect` teilen wir dem Compiler mit, daß er den Interrupt-Vektor Nr. 0 (also den ersten) auf diesen Interrupt-Handler zeigen lassen soll.

(Tatsächlich handelt es sich bei `ISR` und `INT0_vect` um Macros.)

Die Schreibweise ist spezifisch für die Programmierung des Atmel AVR ATmega unter Verwendung der GNU Compiler Collection (GCC). Bei Verwendung anderer Werkzeuge und/oder Prozessoren kann dasselbe Programm völlig anders aussehen. Wie man Interrupt-Handler schreibt und wie man Interrupt-Vektoren setzt, ist ein wichtiger Bestandteil der Dokumentation der Entwicklungswerkzeuge.

Die so geschriebene Funktion wird immer dann aufgerufen, wenn die Hardware den Interrupt Nr. 0 auslöst. Wann das der Fall ist, hängt von der Beschaltung ab. Im Falle des RP6 geschieht es dann, wenn ein Sensor an der linken Raupenkette einen schwarzen Streifen auf der Encoder-Scheibe registriert, also immer dann, wenn sich die linke Raupenkette des Roboters um eine bestimmte Strecke gedreht hat.

Jedesmal wenn sich die Raupenkette um einen Teilstrich weitergedreht hat, werden also zwei Zähler inkrementiert. Wir können dies nutzen, um z. B. durch Auslesen des Zählers `mleft_dist` die zurückgelegte Entfernung zu messen. (Die RP6-Bibliothek selbst stellt nur eine Zeit- und eine Geschwindigkeitsmessung zur Verfügung.) Wie dies konkret geschehen kann, sei im folgenden vorgestellt.

Methode 1: Verändern des Interrupt-Handlers

- Da die Bibliothek `RP6RobotBase` im Quelltext vorhanden ist, können wir sie selbst ändern und in den Interrupt-Handler `ISR (INT0_vect)` einen eigenen Zähler für Sensor-Ticks einbauen.
- Wenn wir diesen zum Zeitpunkt A auf 0 setzen und zum Zeitpunkt B auslesen, erfahren wir, wieviele „Ticks“ der Roboter dazwischen zurückgelegt hat.

Methode 2: Verwenden eines vorhandenen Zählers

- Tatsächlich enthält `ISR (INT0_vect)` bereits zwei Zähler, die bei jedem Sensor-„Tick“ hochgezählt werden: `mleft_dist` und `mleft_counter`.
- Einer davon (`mleft_dist`) wird bei jedem `move()` auf 0 zurückgesetzt. Für diesen Zähler enthält `RP6RobotBaseLib.h` einen „undokumentierten“ Makro `getLeftDistance()`, um ihn auszulesen.
- Bei sorgfältiger Lektüre von `RP6RobotBaseLib.c` erkennt man, daß es unproblematisch ist, den Zähler vom Hauptprogramm aus auf 0 zu setzen. (Dies ist jedoch mit Vorsicht zu genießen: In einer eventuellen Nachfolgeversion der Bibliothek muß dies nicht mehr der Fall sein!)

Methode 3: Abfrage der Sensoren mittels Busy Waiting

- Alternativ zur Verwendung des Interrupt-Handlers kann man auch von der eigenen Hauptschleife aus den Sensor periodisch abfragen und bei jeder Änderung einen Zähler hochzählen.
- Diese Methode heißt „Busy Waiting“. Sie hat den Vorteil der Einfachheit aber den Nachteil, daß der Prozessor „in Vollzeit“ damit beschäftigt ist, einen Sensor abzufragen, und eventuelle andere Aufgaben nur noch „nebenher“ erledigen kann.
- Wenn aus irgendwelchen Gründen der Interrupt-Mechanismus nicht verwendet werden kann (z.B. weil der Prozessor über kein Interrupt-Konzept verfügt), könnten wir die Lichtschranke alternativ auch mit einem Input-Port verdrahten und mittels Busy Waiting abfragen.

Dies funktioniert nur dann, wenn die Schleife wirklich regelmäßig den Sensor abfragt. Sobald der Prozessor längere Zeit mit anderen Dingen beschäftigt ist, können beim Busy Waiting Signale der Lichtschranke verlorengehen. Dieses Problem besteht nicht bei Verwendung von Interrupts.

5.4 volatile-Variable

Im C-Quelltext fällt auf, daß die Zähler-Variablen `mleft_dist` und `mleft_counter` als `volatile uint16_t mleft_counter` bzw. `volatile uint16_t mleft_dist` deklariert sind anstatt einfach nur als `uint16_t mleft_counter` und `uint16_t mleft_dist`.

Das Schlüsselwort `volatile` teilt dem C-Compiler mit, daß eine Variable immer im Speicher (RAM) aufbewahrt werden muß und nicht in einem Prozessorregister zwischengespeichert werden darf.

Dies ist deswegen wichtig, weil jederzeit ein Interrupt erfolgen kann, der den Wert der Variablen im Speicher verändert. Wenn im Hauptprogramm alle „überflüssigen“ Speicherzugriffe wegoptimiert wurden, erfährt es nichts von der Änderung.

Entsprechendes gilt für I/O-Ports: Wenn ein Programm einen Wert in einen Output-Port schreiben oder aus einem Input-Port lesen soll, ist es wichtig, daß der Speicherzugriff auch tatsächlich stattfindet.

5.5 Software-Interrupts

Manche Prozessoren verfügen über einen Befehl, um Interrupts „künstlich“ auszulösen.

Das Betriebssystem MS-DOS verwendet derartige Aufrufe anstelle von „normalen“ Unterprogrammaufrufen, um Programmen Funktionen zur Verfügung zu stellen.

Beispiel: Assembler-Version von `printf ("Hello, world!\n")` unter MS-DOS bzw. Unix

MS-DOS-Version für FASM (gekürzt)

```
hello db 'Hello, world', 10, 13, '$'

mov ah, 09h
mov dx, hello
int 21h
```

Unix-Version für GCC (gekürzt)

```
hello:
    .string "Hello, world!\n"

    pushl $hello
    call printf
```

- Die MS-DOS-Version ruft den Interrupt Nr. 33 (hexadezimal: 21) auf: `int 21h`.
Die Unix-Version verwendet stattdessen einen normalen Unterprogrammaufruf: `call printf`.
- Die MS-DOS-Version übergibt Parameter in Prozessorregistern:
Die Konstante `09h` im `ah`-Register wählt die Funktion „Textausgabe“ aus;
das `dx`-Register enthält einen Zeiger auf den Text.
Die Unix-Version benutzt den Stack zur Übergabe des Parameters: `pushl $hello`.
(`$hello` ist ein Zeiger auf den Text.)
- Obwohl beide Programme auf demselben Prozessor laufen, unterscheiden sich die Sprachdialekte der beiden Assembler FASM und GCC erheblich voneinander.
(Reihenfolge der Operanden umgekehrt, Suffix `l` für „long“, Präfix `$` für Konstanten, ...)

Derartige „Software-Interrupts“ verursachen Probleme, sobald ein Gerät den Interrupt für seinen eigentlichen Zweck verwendet. MS-Windows verwendet – außer zur Emulation von MS-DOS – keine Software-Interrupts mehr.

(Ein sehr ähnlicher Mechanismus wird von modernen Betriebssystemen weiterhin für Systemaufrufe genutzt. Hier geht es darum, den Übergang von potentiell unsicherem Code in Anwenderprogrammen zum vertrauenswürdigen Code des Betriebssystems zu kontrollieren. Für Details siehe:
<http://de.wikipedia.org/wiki/Software-Interrupt>, <http://de.wikipedia.org/wiki/Systemaufruf>)

5.6 Byte-Reihenfolge – Endianness

5.6.1 Konzept

Beim Speichern von Werten, die größer sind als die kleinste adressierbare Einheit (= Speicherzelle oder Speicherwort, häufig 1 Byte), werden mehrere Speicherworte belegt.

Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

$$1027 = 1024 + 2 + 1 = 0000\,0100\,0000\,0011_2 = 0403_{16}$$

Diese 16-Bit-Zahl kann auf zwei verschiedene Weisen in zwei 8-Bit-Speicherzellen gespeichert werden:

04	03	Big-Endian, „großes Ende zuerst“, für Menschen leichter lesbar
03	04	Little-Endian, „kleines Ende zuerst“, bei Additionen effizienter (Schriftliches Addieren beginnt immer beim Einer.)

Welche Konvention man verwendet, ist letztlich Geschmackssache und hängt von der verwendeten Hardware (Prozessor) und Software ab. Man spricht hier von der „Endianness“ (Byte-Reihenfolge) der Hardware bzw. der Software.

Im Kontext des Datenaustausches ist es wichtig, sich auf eine einheitliche Endianness zu verständigen. Dies gilt insbesondere für:

- Dateiformate
- Datenübertragung

5.6.2 Dateiformate

Als Beispiel für Dateiformate, in denen die Reihenfolge der Bytes in 16- und 32-Bit-Zahlen spezifiziert ist, seien hier Audio-Formate genannt:

- RIFF-WAVE-Dateien (`.wav`): Little-Endian
- Au-Dateien (`.au`): Big-Endian
- ältere AIFF-Dateien (`.aiff`): Big-Endian
- neuere AIFF-Dateien (`.aiff`): Little-Endian

Insbesondere ist es bei AIFF-Dateien wichtig, zu prüfen, um welche Variante es sich handelt, anstatt sich auf eine bestimmte Byte-Reihenfolge zu verlassen.

Bei Dateiformaten mit variabler Endianness ist es sinnvoll und üblich, die Endianness durch eine Kennung anzuzeigen. Dies geschieht häufig am Anfang der Datei (im „Vorspann“ – „Header“).

Als weiteres Beispiel seien zwei Monochrom-Grafik-Formate genannt. Hier steht jedes Bit für einen schwarzen bzw. weißen Bildpunkt, daher spielt die Reihenfolge der Bits in den Bytes eine entscheidende Rolle. (Diese Grafik-Formate sind Ihnen bereits aus der Vorlesung „Angewandte Informatik“ bekannt.)

- PBM-Dateien: Big-Endian, MSB first – die höchstwertige Binärziffer ist im Bild links
- XBM-Dateien: Little-Endian, LSB first – die höchstwertige Binärziffer ist im Bild rechts

MSB/LSB = most/least significant bit

Achtung: Die Abkürzungen „MSB/LSB“ werden manchmal auch für „most/least significant *byte*“ verwendet. Im konkreten Fall ist es ratsam, die verwendete Byte- und Bit-Reihenfolge genau zu recherchieren bzw. präzise zu dokumentieren.

5.6.3 Datenübertragung

Bei der Übertragung von Daten über Leitungen spielt sowohl die Reihenfolge der Bits in den Bytes („MSB/LSB first“) als auch die Reihenfolge der Bytes in den übertragenen Zahlen („Big-/Little-Endian“) eine Rolle.

Als Beispiele seien genannt:

- RS-232 (serielle Schnittstelle): MSB first
- I²C: LSB first
- USB: beides
Um Übertragungsfehler erkennen zu können, werden im USB-Protokoll bestimmte Werte einmal gemäß der MSB-first- und einmal gemäß der LSB-first-Konvention übertragen und anschließend auf Gleichheit geprüft.
- Ethernet: LSB first
- TCP/IP (Internet): Big-Endian

Insbesondere gilt für die Übertragung z. B. einer 32-Bit-Zahl über das Internet, daß die vier Bytes von links nach rechts (Big-Endian) übertragen werden, die acht Bits innerhalb jedes Bytes hingegen von rechts nach links (LSB first).

5.7 Speicherausrichtung – Alignment

Ein 32-Bit-Prozessor kann auf eine 32-Bit-Variable effizienter zugreifen, wenn die Speicheradresse der Variablen ein Vielfaches von 32 Bits, also 4 Bytes ist. Eine Variable, auf die dies zutrifft, heißt „korrekt im Speicher ausgerichtet“ („aligned“).

„Effizienter“ kann bedeuten, daß Maschinenbefehle zum Arbeiten mit den Variablen schneller abgearbeitet werden. Es kann aber auch bedeuten, daß der Prozessor gar keine direkte Bearbeitung von inkorrekt ausgerichteten Variablen erlaubt. In diesem Fall bedeutet eine inkorrekte Speicherausrichtung, daß für jede Operation mit der Variablen anstelle eines einzelnen Maschinenbefehls ein kleines Programm aufgerufen werden muß.

Um zu verstehen, welche Konsequenzen dies für die Arbeit mit Rechnern hat, betrachten wir die folgenden Variablen (in C-Schreibweise):

```
uint8_t a;  
uint16_t b;  
uint8_t c;
```

Die Anordnung dieser Variablen im Speicher könnte z. B. folgendermaßen aussehen:

	...
3005	
3004	
3003	c
3002	b
3001	
3000	a
2fff	
	...

Ein optimierender Compiler wird für eine korrekte Ausrichtung der Variablen `b` sorgen, beispielsweise durch Auffüllen mit unbenutzten Speicherzellen:

	...
3005	
3004	c
3003	
3002	b
3001	
3000	a
2fff	
	...

Alternativ ist es dem Compiler auch möglich, die korrekte Ausrichtung durch „Umsortieren“ der Variablen herzustellen und dadurch „Löcher“ zu vermeiden:

	...
3005	
3004	
3003	b
3002	
3001	c
3000	a
2fff	
	...

Fazit: Man kann sich als Programmierer nicht immer darauf verlassen, daß die Variablen im Speicher in einer spezifischen Weise angeordnet sind.

In vielen existierenden Programmen geschieht dies dennoch. Diese Programme sind fehlerhaft. Dort kann es z. B. passieren, daß nach einem Upgrade des Compilers schwer lokalisierbare Fehler auftreten.

Entsprechende Überlegungen gelten für 64-Bit- und 16-Bit-Prozessoren. Die Größe der Variablen, aufgerundet auf die nächste Zweierpotenz, gibt eine Ausrichtung vor. Die Registerbreite des Prozessors markiert die größte Ausrichtung, die noch berücksichtigt werden muß.

Bei 8-Bit-Prozessoren stellt sich die Frage nach der Speicherausrichtung normalerweise nicht, weil die kleinste adressierbare Einheit eines Speichers selten kleiner als 8 Bits ist.

Beispiele:

- Eine 64-Bit-Variable auf einem 64-Bit-Prozessor muß auf 64 Bits ausgerichtet sein.
- Eine 32-Bit-Variable auf einem 64-Bit-Prozessor braucht nur auf 32 Bits ausgerichtet zu sein.
- Eine 64-Bit-Variable auf einem 32-Bit-Prozessor braucht nur auf 32 Bits ausgerichtet zu sein.
- Eine 64-Bit-Variable auf einem 8-Bit-Prozessor braucht nur auf 8 Bits ausgerichtet zu sein.

Bei der Definition von Datenformaten tut man gut daran, die Ausrichtung der Daten von vorneherein zu berücksichtigen, um auf möglichst vielen – auch zukünftigen – Prozessoren eine möglichst effiziente Bearbeitung zu ermöglichen.

Wenn ich beispielsweise ein Dateiformat definiere, in dem 128-Bit-Werte vorkommen, ist es sinnvoll, diese innerhalb der Datei auf 128 Bits (16 Bytes) auszurichten, auch wenn mein eigener Rechner nur über einen 64-Bit-Prozessor verfügt.

6 Anwender-Software

6.1 Relokation und Linken

Wenn eine Software erstellt wird, ist häufig nicht bekannt, an welcher Speicheradresse sie sich zum Zeitpunkt der Ausführung befinden wird.

Ein Betriebssystem, das eine Anwender-Software vom Massenspeicher in den Arbeitsspeicher lädt, muß daher dafür sorgen, daß die in der Software enthaltenen Sprunganweisungen auf die jeweils richtigen Sprungziele verweisen. Dieser Vorgang heißt „Relokation“; der zugehörige Bestandteil des Betriebssystems heißt „Relocator“.

Dasselbe gilt, wenn eine Anwender-Software entwickelt wird und eine Software-Bibliothek mitbenutzt. In diesem Fall müssen die Entwicklungswerkzeuge dafür sorgen, daß die Sprunganweisungen des Hauptprogramms in die Bibliothek auf die jeweils richtigen Sprungziele verweisen. Dieser Vorgang heißt „Linken“ (Verbinden); das zugehörige Entwicklungswerkzeug heißt „Linker“.

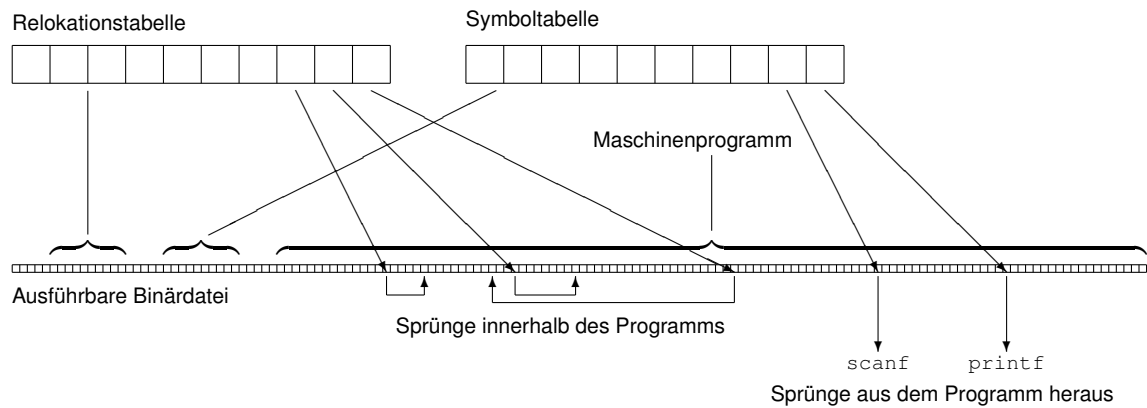
Wenn das Linken erst beim Laden des Programms in den Speicher stattfindet, spricht man von „dynamischem“ Linken, ansonsten von „statischem“ Linken.

Viele Prozessoren (u. a. IA-32) unterstützen eine „Segmentierung“ des Speichers: Das Programm arbeitet mit seinen eigenen Speicheradressen; die Übersetzung in physikalische Speicheradressen erfolgt durch den Prozessor (Hardware). Wenn dies genutzt werden kann, ist keine Relokation notwendig.

6.2 Dateiformate

Damit der Relocator und der Linker ihre Aufgabe erfüllen können, können „fertige“ Maschinenprogramme nicht „einfach so“ auf einem Massenspeicher gespeichert werden, sondern sie müssen zusätzliche Informationen über Sprunganweisungen enthalten:

- Für den Relocator: Relokationstabelle
Tabelle der Sprungziele innerhalb des Programms, die an den tatsächlichen Ort im Speicher angepaßt werden müssen
- Für den Linker: Symboltabelle
Tabelle der Sprungziele außerhalb des Programms, die in das Programm eingetragen werden müssen



Eine Datei, die ein „fertiges“ Maschinenprogramm, eine Relokationstabelle und Symboltabellen für den statischen und dynamischen Linker enthält, heißt „Objekt-Datei“.

Eine Datei, die ein „fertiges“ Maschinenprogramm, eine Relokationstabelle und eine Symboltabelle für den dynamischen Linker enthält, heißt „ausführbare Binärdatei“.

Für Objekt-Dateien und ausführbare Dateien gibt es herstellerspezifische und herstellerübergreifende Standards (z. B. a.out, COFF, ELF).

Übliche Dateiendungen sind

- für Objekt-Dateien: .o (Unix), .obj (MS-Windows)
- für ausführbare Binärdateien: keine Endung (Unix), .com, .exe, .scr (MS-Windows)

Häufig werden mehrere Objekt-Dateien zu einer „Bibliothek“ zusammengefaßt.

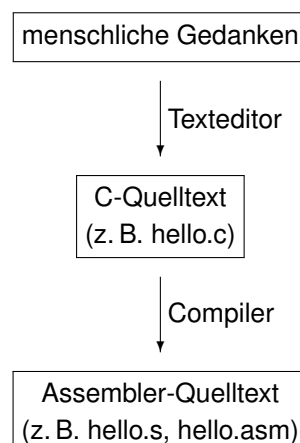
Übliche Dateiendungen sind

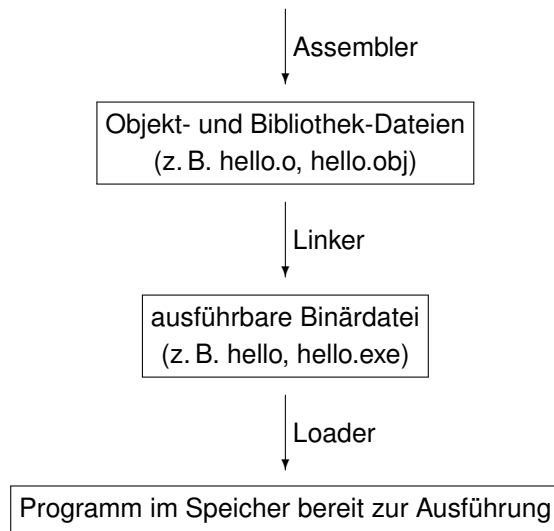
- für statische Bibliotheken: .a (Unix), .lib (MS-Windows)
- für dynamische Bibliotheken: .so (Unix), .dll (MS-Windows)

Derjenige Teil des Betriebssystems, der eine ausführbare Datei in den Arbeitsspeicher lädt und dabei ggf. den Relocator und den Linker aufruft, heißt „Loader“.

6.3 Die Toolchain

Wir können nun die Werkzeuge, die vom Schreiben des Quelltextes bis zur Ausführung des Programms verwendet werden, zu einer Kette, der „Toolchain“, zusammenfassen:





Manche dieser Werkzeuge können zu einer einzigen Software zusammengefaßt sein:

- Das Programm `gcc` faßt Compiler, Assembler und Linker in einem einzigen Aufruf zusammen. Was jeweils aufgerufen wird, entscheidet das Programm anhand der Endungen der übergebenen Dateien.
- Eine „Entwicklungsumgebung“ (z. B. Eclipse) umfaßt typischerweise Texteditor, Compiler, Assembler und Linker plus weitere Werkzeuge.

6.4 Besonderheiten von Mikro-Controllern

Auf einem Personal-Computer läuft ständig eine Software, das Betriebssystem, die in der Lage ist, Anwenderprogramme in den Speicher zu laden und durch den Prozessor ausführen zu lassen.

Ein Mikro-Controller enthält kein Betriebssystem. Damit ein Mikro-Controller sinnvolle Befehle ausführt, müssen wir also auf andere Weise dafür sorgen, daß sie sich im Arbeitsspeicher an der richtigen Stelle befinden.

- ROM: Der Hersteller liefert den Mikro-Controller mit einem nur lesbaren, bereits mit den richtigen Werten beschriebenen Speicher aus. Dies lohnt sich bei großen Stückzahlen.
- In-System Programmer (ISP): Der Mikro-Controller enthält Flash-Speicher, der von außen beschrieben werden kann, z. B. mit Hilfe einer Zusatzschaltung (ISP) von einem Personal-Computer aus.
- Boot-Loader: Der Mikro-Controller enthält Flash-Speicher, der abschnittsweise von der eigenen Software beschrieben werden kann. Ein Teil des Speichers enthält ein Programm (Boot-Loader), das über externe Leitungen (z. B. eine serielle Schnittstelle) Daten entgegennehmen und in den verbleibenden Speicher schreiben kann.

In jedem Fall ist es notwendig, die Relokation bereits vor dem Aufspielen der Software vorzunehmen. Gängige Formate ausführbarer Dateien (z. B. ELF) sind zur direkten Ausführung auf einem Mikro-Controller nicht geeignet. Stattdessen muß – als letzter Schritt der Software-Entwicklung – ein Relocator anhand der ELF-Datei ein Abbild des Speichers des Mikro-Controllers erstellen.

Das Speicherabbild kann unmittelbar als Binärdatei gespeichert werden oder als Text-Darstellung einer Binärdatei, z. B. im Intel-HEX-Format (Dateiendung: `.hex`).

Das so entstandene Speicherabbild kann dann mit Hilfe eines ISP oder eines Boot-Loaders auf den Mikro-Controller aufgespielt werden (Sprechweise: „Download“), oder man läßt es dem Hersteller zukommen, damit dieser eine Serie der Mikro-Controller fertigt, bei denen die Software bereits fertig im ROM liegt.

Die Werkzeuge zur Programmierung von Mikro-Controllern laufen nicht auf dem Mikro-Controller selbst, sondern auf einem Personal-Computer.

Die Werkzeuge erzeugen also Code, der nicht auf demselben Rechner lauffähig ist, sondern auf einem anderen, dem Mikro-Controller. Derartige Werkzeuge heißen „Cross-Werkzeuge“ (Cross-Compiler, Cross-Assembler, Cross-Linker).

7 Bus-Systeme

7.1 Konzept

Die *International Electrotechnical Commission* (IEC) definiert ein Bus-System als

„ein System zur Datenübertragung zwischen mehreren Teilnehmern über einen gemeinsamen Übertragungsweg, bei dem die Teilnehmer nicht an der Datenübertragung zwischen anderen Teilnehmern beteiligt sind.“

<http://www.electropedia.org/iev/iev.nsf/display?openform&ievref=351-32-10>

Übersetzung: Wikipedia – Bus (Datenverarbeitung)

Kurz: Ein Bus-System ist Datenübertragung „ohne Umsteigen“. Sobald Teilnehmer dafür zuständig sind, die Daten an andere Teilnehmer weiterzuleiten (z. B. Router im Internet), handelt es sich nicht mehr um ein Bus-System im engeren Sinne, sondern um eine Kombination mehrerer Bus-Systeme.

In der Rechnertechnik finden sich Bus-Systeme

- von Schaltkreis zu Schaltkreis im Prozessor,
- zwischen Prozessor und Speicher,
- zwischen Prozessor und Gerät,
- zwischen Prozessor und Controller,
- zwischen Controller und Gerät,
- zwischen Netzwerkkarte und Netzwerkkarte

... und in zahlreichen weiteren Anwendungen.

7.2 Zu berücksichtigen

Damit Datenübertragung funktioniert, sind beim Entwurf und Aufbau von Bus-Systemen zahlreiche Aspekte zu berücksichtigen. Hier ein Überblick.

- **Elektromagnetische Störungen**

Elektromagnetische Impulse in der Umgebung der Leitungen können unerwünschte Störsignale induzieren. Umgekehrt können in der Leitung übertragene Signale elektromagnetische Wellen erzeugen, die Funkübertragungen in der Umgebung störend beeinflussen.

Zu den Maßnahmen zur Vermeidung elektromagnetischer Störungen gehören:

- Abschirmung
- symmetrische Signalübertragung
Dasselbe Signal wird in zwei Leitungen mit entgegengesetzter Polarität übertragen. Wenn eine Störung das eine Signal abschwächt, verstärkt sie gleichzeitig das andere.
- unterschiedliche Verdrillung („Twisted Pair“)
Mehrere Leiter in demselben Kabel können sich gegenseitig beeinflussen. Wenn man dabei jeweils die beiden Leitungen einer symmetrischen Signalübertragung mit unterschiedlicher Ganghöhe verdrillt, kommen sich über die Kabellänge verteilt stets unterschiedliche Kabel nahe, so daß sich gegenseitige Störungen herausmitteln.
- ...

- **Reflexion am Kabelende**

Bei hohen Übertragungsgeschwindigkeiten kann die Ausbreitung des Signals in der Leitung nicht mehr als „unendlich schnell“ angenommen werden. In dieser genauen Betrachtung wird nicht die gesamte Leitung gleichzeitig auf z. B. +5 V gezogen, sondern das Signal breitet sich bis zum Kabelende aus, wird dort reflektiert, wandert wieder zurück, wird wieder reflektiert, usw. Will man diesen Einschwingvorgang nicht abwarten, sondern sofort das nächste Signal senden, wird die Reflexion am Kabelende zum Problem. Folgende Gegenmaßnahmen sind üblich:

- Abschlußwiderstand („Terminator“)
Ein ohmscher Widerstand, der dem Wellenwiderstand der Leitung entspricht, dämpft das Signal am Leitungsende herab und verhindert so die Reflexion. (Beispiele: SCSI, BNC-Netz)
- ignorieren
Bei niedrigen Übertragungsgeschwindigkeiten und/oder kurzen Leitungen spielen die o. a. Betrachtungen keine Rolle. Wer hier Kompromisse eingeht, kann auf Terminierung verzichten. (Beispiel: IDE/ATAPI/PATA)
- (Auch hier gibt es weitere Methoden, z. B. aktive Terminierung.)

- **Kabelwege**

Verschiedene Einsatzgebiete erfordern verschiedene Verkabelungen.

- Wenige Leitungen (seriell) vs. viele (parallel)
- Punkt-zu-Punkt-Verbindung vs. Adressierung
Eine Kommunikationsverbindung zwischen genau zwei Teilnehmern heißt Punkt-zu-Punkt-Verbindung.
Wenn man viele Teilnehmer an dieselben Leitungen anschließt und trotzdem mit jedem individuell kommunizieren will, muß aus den Signalen hervorgehen, welcher Teilnehmer gemeint ist. Dieses Konzept heißt Adressierung.
(In manchen Kontexten wird erst dann von einem Bus-System gesprochen, wenn es mehr als zwei Teilnehmer und Adressierung gibt. Im Gegensatz zu einer Punkt-zu-Punkt-Verbindung wird ein solches System dann als „busfähig“ bezeichnet.)
- Topologie: linear, ring-, sternförmig, kaskadierbar
Bei kurzen Wegen und/oder niedrigen Übertragungsgeschwindigkeiten kann die Form der Leitungen beliebig sein. Bei langen Wegen und hohen Übertragungsgeschwindigkeiten müssen Reflexionen an Verzweigungen und an Kabelenden (s. o.) berücksichtigt und ggf. vermieden werden.

- **Weitere Aspekte**

In der Nachfolgeveranstaltung „Vertiefung Rechner- und Netzwerktechnik“ werden u. a. die folgenden Aspekte angesprochen werden:

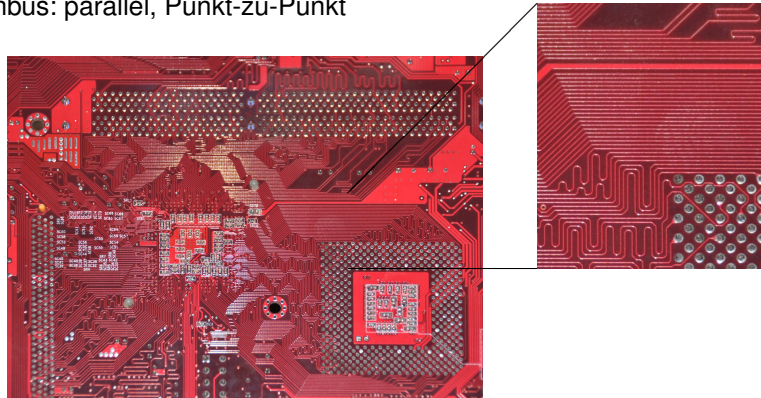
- Kollisionsvermeidung
Wie vermeidet man es, daß mehrere Teilnehmer gleichzeitig senden und sich dadurch gegenseitig stören?
- Sicherheitsaspekte
Wie kann man vermeiden oder es zumindest erkennen, daß sich unberechtigte Teilnehmer an der Kommunikation beteiligen? Wie stellt man die Authentizität einer empfangenen Nachricht sicher?
- u. v. a.

7.3 Beispiele

Im folgenden sollen die Beispiele aus Abschnitt 7.1 hinsichtlich der o. a. Aspekte eingeordnet werden.

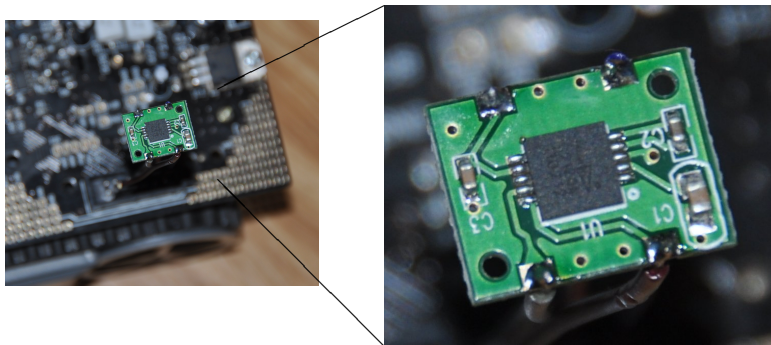
- von Schaltkreis zu Schaltkreis im Prozessor:
parallel, Punkt-zu-Punkt

- zwischen Prozessor und Speicher:
Adreß- und Datenbus: parallel, Punkt-zu-Punkt

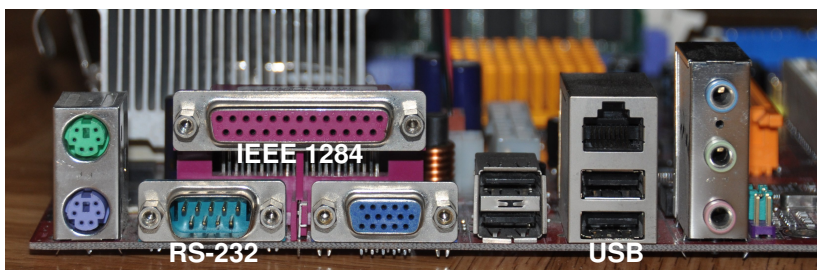


Auf den Hauptplatinen (Motherboards) aktueller Computer erkennt man zahlreiche parallel verlaufende Leitungen (von möglichst gleicher Länge).

- zwischen Prozessor und Gerät:
I²C: seriell, mit Adressierung
Beispiel: Kompaßmodul an RP6



- zwischen Prozessor und Controller (auf dem Motherboard):
parallel, mit Adressierung
- zwischen Controller und Gerät:
 - IEEE 1284 (Centronics): parallel, Punkt-zu-Punkt
 - RS-232: seriell, Punkt-zu-Punkt
 - RS-485, USB, CAN: seriell, mit Adressierung



Anschlüsse an der Rückseite eines PC-Motherboards

- zwischen Controller und Festplatte:
 - SCSI: parallel, terminiert, mit Adressierung, linear

- PATA (ältere Bezeichnungen: IDE, ATAPI):
parallel, nicht terminiert,
mit Adressierung (Master/Slave), linear
- SATA: seriell, Punkt-zu-Punkt



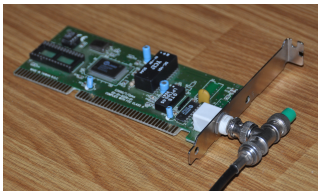
Festplatte an PATA-Datenkabel



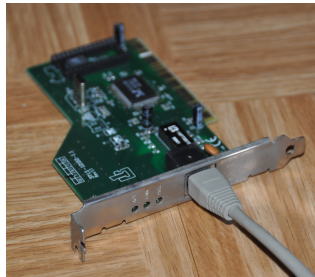
Festplatte an SATA-Datenkabel

Obwohl bei PATA 16 Leitungen zur Datenübertragung zur Verfügung stehen und bei SATA nur eine, hat SATA höhere Übertragungsraten. Der Grund dafür liegt darin, daß die Anforderung des *exakt gleichzeitigen* Bereitstellens der Datenbits an den nicht terminierten parallelen Leitungen die Taktfrequenz dermaßen begrenzt, daß eine serielle Übertragung der Daten mit wesentlich höherer Taktfrequenz letztlich effizienter ist.

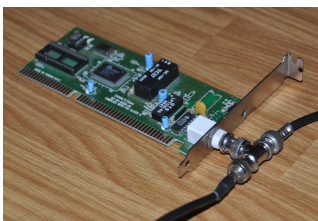
- zwischen Netzwerkkarte und Netzwerkkarte:
seriell, mit Adressierung
 - Token Ring: ringförmig
 - BNC-Ethernet: linear, terminiert
 - Twisted-Pair-Ethernet: Punkt-zu-Punkt,
mit Hubs/Switches: sternförmig
 - WLAN: sternförmig (Access-Point),
im Ad-Hoc-Modus: beliebige Topologie



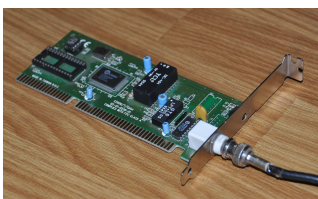
BNC-Netz: Endpunkt (terminiert)



Twisted-Pair-Netz



BNC-Netz: Mitte (vorbeigeführt)



BNC-Netz: **falsch** angeschlossen



WLAN-Access-Point

7.4 Beispiel: Benutzung des I²C-Busses

Als konkretes Beispiel für die Anwendung eines Bus-Systems sei hier das Auslesen des bereits erwähnten Kompaß-Moduls für den RP6-Roboter näher ausgeführt.

Eine C-Funktion zum Auslesen der Kompaßwerte lautet:

```
void read_compass (uint16_t *x, uint16_t *y)
{
    I2CTWI_transmit2Bytes (0x60, 0x00, 0x02); // set coil
    mSleep (1);
    I2CTWI_transmit2Bytes (0x60, 0x00, 0x04); // reset coil
    mSleep (5);
    uint8_t result[5];
    I2CTWI_transmit2Bytes (0x60, 0x00, 0x01); // Messung starten
    mSleep (5);                               // 5ms warten, bis Sensor fertig gemessen hat
    I2CTWI_transmitByte (0x60, 0x01);         // Leseindex setzen
    I2CTWI_readBytes (0x61, result, 4);       // lesen: msb x, lsb x, msb y, lsb y
    result[0] &= 0b00001111;                  // Unwichtige Bits vom msb abschneiden
    result[2] &= 0b00001111;
    *x = (result[0] << 8) + result[1];        // Wert berechnen aus msb und lsb
    *y = (result[2] << 8) + result[3];
}
```

Das Senden von Befehlen an das Kompaßmodul erfolgt immer nach demselben Schema.

Beispiel: `I2CTWI_transmit2Bytes (0x60, 0x00, 0x01); // Messung starten`

- **Sende das Byte 0x60**
Hiermit wird das Gerät „Kompaßmodul“ zum Schreiben adressiert.
Nachfolgende Bytes werden vom Kompaßmodul verarbeitet
und von eventuellen weiteren an denselben I²C-Bus angeschlossenen Geräten ignoriert.
- **Sende das Byte 0x00**
Hiermit wird das „Register Nr. 0“ innerhalb des Kompaßmoduls zum Schreiben adressiert.
- **Sende das Byte 0x01**
Dies sind die eigentlichen Nutzdaten – in diesem Fall der Befehl Nr. 1 „Messung starten“.

Das Auslesen von Daten erfolgt analog mit einer um 1 erhöhten Adresse:

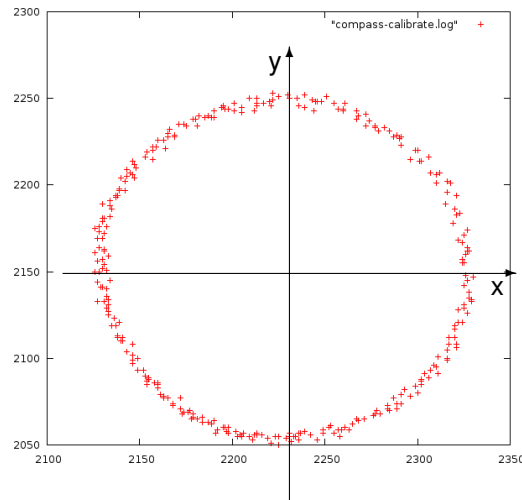
Zum Lesen wird das Kompaßmodul über die Adresse 0x61 angesprochen
anstatt, wie zum Schreiben, 0x60.

Die Verwendung der Funktion aus einem Programm heraus erfolgt wie folgt:

```
uint16_t compass_x, compass_y;
read_compass (&compass_x, &compass_y);
```

Als konkrete Anwendung können wir die Funktion einsetzen, um das Kompaßmodul zu kalibrieren und den Roboter in eine Himmelsrichtung auszurichten.

Wir lassen den Roboter auf der Stelle rotieren und rufen dabei wiederholt die Funktion `read_compass()` auf. Wenn wir die *x*- und *y*-Werte aufzeichnen und gegeneinander auftragen, erhalten wir näherungsweise eine Ellipse:



Um nun den Roboter in eine Himmelsrichtung auszurichten, können wir z. B. folgendermaßen vorgehen:

- Achsen normieren:
Wir berechnen den Mittelpunkt (Mitte zwischen Maximal- und Minimalwerten) und subtrahieren ihn von den Meßwerten.
- In den derart normierten Meßwerten markiert der Nulldurchgang einer Achse eine Himmelsrichtung.
Beispiel: $x = 0$ markiert die Nord-Süd-Richtung.
- Das Vorzeichen der jeweils anderen Achse sagt aus, welche der beiden Himmelsrichtungen es ist.
Beispiel: $x = 0$ zusammen mit $y > 0$ markiert Ausrichtung nach Norden.

Um also z. B. den Roboter nach Norden auszurichten, können wir ihn zunächst grob drehen, bis y positiv ist, und anschließend fein drehen, bis $x = 0$ ist.

Hinweis für die Praxis: Der Funktionsaufruf `I2CTWI_readBytes (0x61, result, 4);` überschreibt in der Variablen `result[]` nicht nur die vier angeforderten Bytes, sondern noch ein fünftes Byte für interne Zwecke. (Daher: `uint8_t result[5];` und nicht `uint8_t result[4];`)




8 Pipelining

8.1 Konzept

Wenn eine Aufgabe unter Verwendung mehrerer Ressourcen (z. B. Werkzeuge) wiederholt ausgeführt werden soll, ist es sinnvoll, die Aufgabe so in Teilaufgaben zu zerlegen, daß die Teilaufgaben parallel unter Verwendung der jeweiligen Ressourcen ausgeführt werden. Dieses Konzept heißt „Pipelining“.

Zur Illustration verwenden wir ein Beispiel aus dem Alltag.

- Es sollen drei Ladungen Wäsche gewaschen werden.
- Der Waschvorgang besteht aus den Teilaufgaben

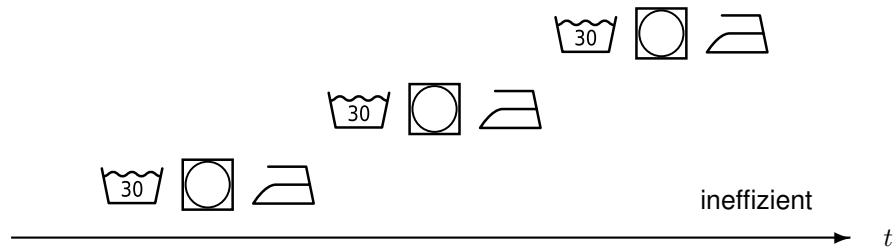
„Waschen“ , „Trocknen“  und „Bügeln“ .

- Die Teilaufgaben müssen in der richtigen Reihenfolge ausgeführt werden:
Die Wäsche („Daten“) „fließt“ von einer Ressource zur nächsten – „Datenfluß“.
- Jede Teilaufgabe belegt jeweils eine Ressource (Waschmaschine, Trockner, Bügeleisen).



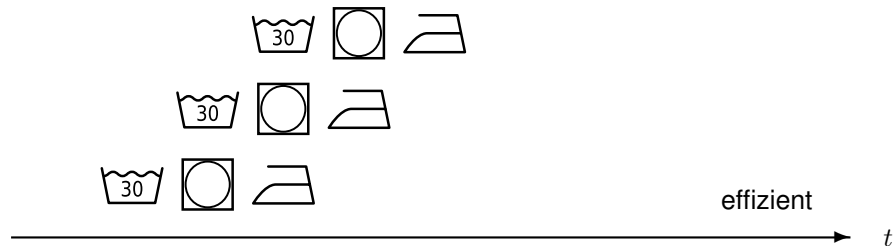
Abbildung 16: Symbolbilder für „Waschen“, „Trocknen“ und „Bügeln“

Ohne Pipelining sieht der Waschvorgang folgendermaßen aus:



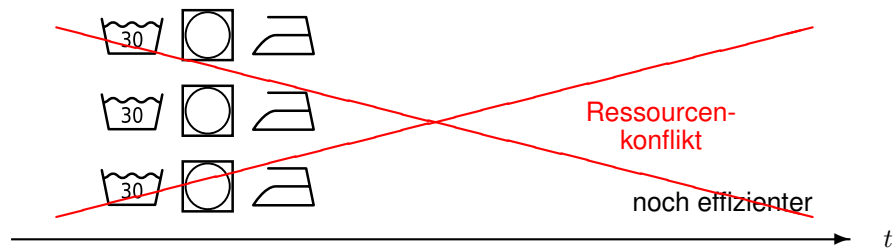
Um den Waschvorgang effizienter zu gestalten, können wir, während die erste Waschladung im Trockner ist, bereits die zweite in der Waschmaschine waschen usw.

Die Zerlegung der Aufgabe in drei parallel ausführbare Teilaufgaben nennen wir eine „dreistufige Pipeline“.



Wenn nur eine Waschmaschine vorhanden ist, kann die zweite Ladung Wäsche solange nicht gewaschen werden, bis die Waschmaschine wieder frei ist.

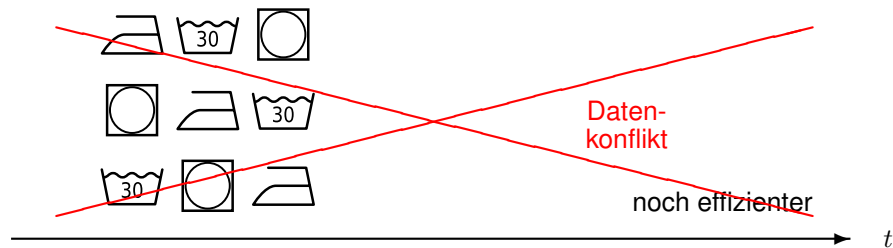
Dies nennt man einen „Ressourcenkonflikt“.



Eine andere Randbedingung ist die Reihenfolge der Teilaufgaben: Es ist nicht zielführend, eine Ladung Wäsche zu trocknen, bevor sie gewaschen wurde.

Entsprechendes gilt im Prozessor: Es ist nicht möglich, Rechenergebnisse weiterzuverarbeiten, bevor diese vorliegen.

Dies nennt man einen „Datenkonflikt“.



8.2 Arithmetik-Pipelines

In Kapitel 3 haben wir bereits einen Register-Stack kennengelernt.

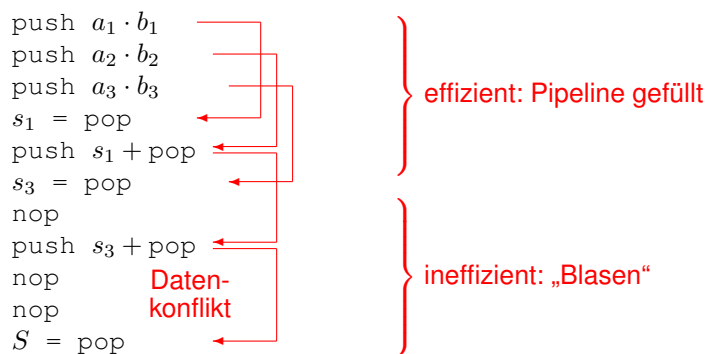
Eine Arithmetik-Pipeline ist gewissermaßen ein „Register-FIFO“: In einem Assemblerbefehl wird eine Rechenaufgabe abgeschickt; eine genau definierte Anzahl Befehle später („Länge der Pipeline“) kann das Ergebnis abgerufen werden.

Beispiel: Berechnung des Skalarprodukts zweier Vektoren der Länge 3

$$S = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$

mit einer Pipeline der Länge 3.

Zur Vereinfachung verwenden wir anstelle von echten Assembler-Befehlen einen Pseudo-Code: Mit dem Befehl `push` wird eine Rechenaufgabe in den „FIFO“ geschoben. Drei Befehle später liegt das Rechenergebnis vor und kann mit der „Funktion“ `pop` abgerufen und weiterbearbeitet werden. Der – häufig tatsächlich vorhandene – Assemblerbefehl `nop` steht für „no operation“. Der Prozessor macht in diesem Fall nichts Neues, rechnet aber im Hintergrund weiter.



An mehreren Stellen ist die Pipeline nicht komplett gefüllt, und manche Ressourcen bleiben ungenutzt. Man nennt dieses Phänomen „Blasen“ und erkennt es an den `nop`-Befehlen. Blasen entstehen als Folge von Konflikten – hier: Datenkonflikten.

Durch die Blasen wirkt der Code eher ineffizient gegenüber einer kompakteren Schreibweise ohne Pipeline:

```

s1 = a1 · b1
s2 = a2 · b2
s3 = a3 · b3
S = s1 + s2
S = S + s3
  
```

Tatsächlich jedoch müssen die Nicht-Pipeline-Befehle bei jeder Rechenaufgabe warten, bis das Ergebnis vorliegt. In diesem Beispiel wären das drei Taktzyklen; wir können uns also unter jeden Nicht-Pipeline-Befehl zwei `nops` denken.

Mit Pipeline benötigt die Rechnung 11 Taktzyklen; ohne Pipeline sind es 15. Bei längeren Rechnungen ist der Unterschied deutlicher.

Als reales Beispiel wird im folgenden die Addition zweier beliebig langer Vektoren mittels einer dreistufigen Pipeline auf einem i860-Prozessor gezeigt.

Der erste Teil dient der Vorbereitung. Hier werden Index-Variablen und Zeiger auf die beiden Arrays initialisiert. Dies soll hier nicht näher vertieft werden und wird nur der Vollständigkeit halber aufgelistet.

```

.align 8
.globl _vadd
nop
_vadd:
shr 1, r19, r19
bte r19, r0, exitadd
addu 0x000F, r16, r16
andnot 0x000F, r16, r16
adds -16, r16, r16
addu 0x000F, r17, r17
andnot 0x000F, r17, r17
  
```

```

adds -16,r17,r17
addu 0x000F,r18,r18
andnot 0x000F,r18,r18
adds -16,r18,r18
mov -1,r20

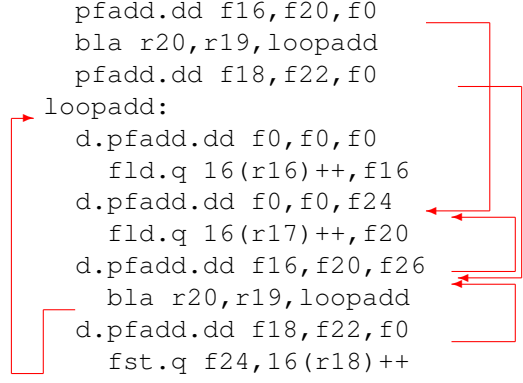
```

Im zweiten Teil erfolgt die eigentliche Addition:

```

fld.q 16(r16)++,f16
fld.q 16(r17)++,f20
pfadd.dd f16,f20,f0
bla r20,r19,loopadd
pfadd.dd f18,f22,f0
loopadd:
d.pfadd.dd f0,f0,f0
fld.q 16(r16)++,f16
d.pfadd.dd f0,f0,f24
fld.q 16(r17)++,f20
d.pfadd.dd f16,f20,f26
bla r20,r19,loopadd
d.pfadd.dd f18,f22,f0
fst.q f24,16(r18)++
nop
nop
nop
exitadd:
bri r1
nop

```



sechsmal f0 = 2 Blasen

r16 und r17 sind ganzzahlige Register, die hier als Zeiger auf die beiden Arrays verwendet werden.

f16 bis f26 sind Fließkommaregister. Um doppelt genaue Operationen auszuführen (8-Byte- statt 4-Byte-Fließkommazahlen), werden jeweils zwei aufeinanderfolgende Fließkommaregister zusammengelegt. „f16“ bezeichnet in diesem Programm also das Registerpaar f16/f17.

Das Fließkommaregister f0 ist ein Pseudo-Register. Das Lesen von f0 ergibt grundsätzlich Null, und in f0 geschriebene Werte werden verworfen.

Der Befehl fld bedeutet: Lade das rechts stehende Fließkommaregister. Der Zusatz .q bewirkt, daß zwei Registerpaare, also insgesamt vier Fließkommaregister gleichzeitig geladen werden. Der Speicherort, von dem geladen werden soll, wird in der ersten Zeile mit 16(r16)++ angegeben. 16(r16) bedeutet: Lade die Fließkommaregister von der Speicherzelle, auf die das Ganzzahlregister r16, erhöht um 16, zeigt. Das ++ bedeutet, daß r16 nach dem Ladevorgang um die Zahl vor der Klammer (hier also 16 Bytes – die Größe von zwei Registerpaaren) erhöht wird.

Der Befehl pfadd.dd f16,f20,f0 addiert zwei doppelt genaue Fließkommaregisterpaare in einer Pipeline. Die Rechnung f16+f20 wird begonnen; das Rechenergebnis, das in diesem Moment aus der Pipeline kommt (hier: undefiniert), wird im Fließkommaregisterpaar f0 gespeichert (hier also: verworfen).

Der Befehl bla r20,r19,loopadd („Branch on LCC and Add“) ist ein bedingter Sprungbefehl. Er addiert r20 (enthält die Konstante -1) zu r19 (Schleifenzähler) und verzweigt dann zu loopadd, wenn *beim vorherigen bla-Aufruf* das Ergebnis ≥ 0 war, wobei vor der Verzweigung noch eine Instruktion ausgeführt wird. Für den bla-Befehl oberhalb der Schleife befindet sich das Verzweigungsziel loopadd genau dort, wo es ohnehin weiterginge. Dieses bla verzweigt also gar nicht, sondern es sorgt dafür, daß das bla am Ende der Schleife einen sinnvollen „vorherigen bla-Aufruf“ vorfindet.

Für das bla am Ende der Schleife ist die zusätzlich ausgeführte Instruktion in dem roten Pfeil auf der linken Seite mit angedeutet.

Die den Befehlen vorangestellten d. innerhalb der Schleife bedeuten, daß der jeweils darunterstehende, eingerückte Befehl *gleichzeitig* mit ausgeführt wird.

Innerhalb der Schleife werden vier Pipeline-Fließkommabefehle ausgeführt. Die Pfeile zeigen, wann jeweils das Ergebnis einer Rechnung vorliegt. Die gleichzeitig ausgeführten Lade- und Speicherbefehle laden jeweils zwei neue Operanden bzw. speichern zwei Ergebnisse.

Obwohl parallel zur eigentlichen Addition pro Rechenbefehl jeweils zwei Lade- oder Speicherbefehle ausgeführt werden, bleibt das Laden und Speichern der „Flaschenhals“ der Rechnung. Dort, wo nicht schnell genug neue Operanden geladen werden können, wird die Pipeline-Rechnung durch `d.pfadd f0, f0, f0` lediglich fortgesetzt, ohne daß gleichzeitig eine neue Rechnung angeworfen wird. Durch das Zählen der „leeren“ `f0`-Operationen können wir die Effizienz der Pipeline-Rechnung sofort ablesen: Jeweils drei `f0` stehen zusammen für einen unproduktiven Taktzyklus – eine Blase in der Pipeline.

Obwohl also der i860 theoretisch in jedem Taktzyklus eine Fließkommaaddition vollenden könnte, erreichen wir in der Praxis „nur“ 2 Additionen (und 2 Blasen) in 4 Taktzyklen. (Ohne Pipeline: 3 Taktzyklen pro Addition)

Zum Abschluß dieses Beispiels sei noch bemerkt, daß es sich hier um ein ausgesprochen *einfaches* Beispiel handelt. Über die hier kurz angerissenen Befehle hinaus kennt der i860-Prozessor z. B. noch Befehle, bei denen gleichzeitig mit einer doppelt genauen Pipeline-Fließkommaaddition noch jeweils *eine halbe* doppelt genaue Pipeline-Fließkommamultiplikation ausgeführt wird. Die Multiplikations-Pipeline steht nur in jedem zweiten Prozessortakt zur Verfügung, hat dafür aber nur zwei Stufen (also vier Takte) anstatt, wie die Additions-Pipeline, drei.

Bei optimaler Auslastung kann der i860 also pro Takt das Ergebnis einer Fließkommaaddition und einer halben Fließkommamultiplikation abliefern und gleichzeitig *entweder* bis zu zwei Lade- *oder* bis zu zwei Speicher-Operationen *oder* eine Ganzzahl-Operation *oder* einen Sprung ausführen, wobei das Erhöhen oder Vergleichen eines Index *keine* Ganzzahl-Operation erfordert, sondern „nebenher“ erfolgt.

Um diese „Peak Performance“ tatsächlich zu erreichen, ist zum einen eine „passende“ Aufgabenstellung erforderlich, zum anderen aufwendiges Optimieren des Assembler-Codes – entweder durch den Compiler oder manuell durch den Programmierer.

8.3 Instruktions-Pipelines

Ein Prozessor benötigt Zeit, um einen Befehl zu verstehen. Gemäß dem Pipelining-Konzept ist es sinnvoll, während der Ausführung eines Befehls bereits die nächsten Befehle vor auszulesen.

Dies kann in manchen Situationen zu Problemen führen, insbesondere bei bedingten Sprüngen.

Zur Illustration betrachten wir einen bedingten Sprung in einer beliebigen Assemblersprache.

```
.L3:
→ movw r30,r20
  add r30,r18
  adc r31,r19
  mov r24,r18
  subi r24,lo8(-(1))
  st Z,r24
  subi r18,lo8(-(1))
  sbci r19,hi8(-(1))
  cp r22,r18
  cpc r23,r19
→ brge .L3
→ ret
```

bedingter Sprung: Welche Befehle vorauslesen?
Kontrollflußkonflikt

`brge` („branch if greater or equal“) ist ein bedingter Sprungbefehl. Vor dessen Ausführung will der Prozessor bereits Befehle vorauslesen – aber welche? Wenn er das `ret` unterhalb von `brge` vorausliest, ist diese Information wertlos, wenn nach oben gesprungen wird. Wenn er das `movw` am Ziel des Sprunges vorausliest, ist diese Information wertlos, wenn *nicht* nach oben gesprungen wird.

Immer wenn die vorausgelesene Information verworfen und neu gelesen werden muß, verliert der Prozessor Zeit. Bei langen Befehls-Pipelines – mehr als 10 Stufen in modernen Prozessoren – kann dies eine erhebliche Einbuße an Effizienz bedeuten. Dies nennt man einen Kontrollflußkonflikt.

Zur Vermeidung von Kontrollflußkonflikten versucht man, Verzweigungen möglichst zuverlässig vorausszusagen. Hierzu seien mehrere Vorgehensweisen skizziert:

- Wir nehmen an, daß es sich bei Sprüngen nach oben um Schleifen und bei Sprüngen nach unten um Auswahl-Verzweigungen handelt. In diesem Fall ist es sinnvoll, Sprünge nach oben grundsätzlich mit „ja“ vorherzusagen und Sprünge nach unten grundsätzlich mit „nein“.

Wenn z. B. eine Schleife 99mal ausgeführt und beim 100sten Mal verlassen wird, ergibt die Zweigvorhersage 99 Treffer und 1 Fehltreffer, also nur 1 Blase als Folge eines Kontrollflußkonfliktes gegenüber 99 korrekt vorausgelesenen Befehlen. Entsprechendes gilt für eine Auswahl von 100 Varianten, von denen nur eine gewählt wird.


Diese primitive Art der Zweigvorhersage schlägt fehl, wenn das Programm z. B. eine Schleife enthält, die normalerweise direkt wieder verlassen wird, oder wenn in einer Reihe von Befehlen jedem einzelnen eine Bedingung („if“) voransteht, die normalerweise erfüllt ist.

- Ein anderes Konzept der Zweigvorhersage nutzt das Wissen aus, das der Programmierer oder der Compiler über das Programm hat: verzögerte Sprünge – „Delayed Branches“.

```

loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++

```



Der auf den Sprungbefehl folgende Befehl wird noch vor dem Sprung ausgeführt. Der Programmierer entscheidet, wie dieser Befehl lautet, und hat somit die Möglichkeit, die wahrscheinlichere Aktion – mit bzw. ohne Sprung – vorzubereiten.

Im o. a. Beispiel für den i860 lautet der Befehl, der vor dem Sprung noch ausgeführt wird:

```

d.pfadd.dd f18,f22,f0
fst.q f24,16(r18)++

```

Der Fließkommateil `d.pfadd.dd f18,f22,f0` leitet eine neue Rechnung ein und setzt laufende Rechnungen fort, ist also nur dann sinnvoll, wenn die Schleife fortgesetzt wird. Der Ganzzahlanteil `fst.q f24,16(r18)++` speichert das Ergebnis einer vergangenen Rechnung, ist also in beiden Fällen sinnvoll. Dieses Programm ist also daraufhin optimiert, daß die Schleife oft ausgeführt wird, verliert aber auch im ungünstigen Fall nur einen halben Taktzyklus (den Fließkommaanteil).

- Eine aus Hardware-Sicht aufwendigere Lösung ist das Konzept des „Branch History Tables“: Der Prozessor merkt sich für bedingte Sprünge, ob sie beim letzten Mal ausgeführt wurden oder nicht. Wenn Code mehrfach ausgeführt wird, *kann* dies eine sinnvolle Vorhersage sein.

Diese Liste ist bei weitem nicht vollständig, sondern soll nur einen kleinen Einblick vermitteln, was alles nötig ist, um die Leistungsfähigkeit aktueller Prozessoren zu ermöglichen.

Die Sprung- oder Zweigvorhersage ist ein wichtiges Konzept moderner Prozessoren und Gegenstand aktueller Forschung.

9 Ausblick

- Aktuelle Prozessoren implementieren *explicitly Parallel Instruction Computing (EPIC)*.
- *Field Programmable Gate Arrays (FPGAs)* sind Bausteine, auf denen man per Software Logik-Schaltungen entstehen läßt (Hardwarebeschreibungssprachen: VHDL, Verilog). Auf diese Weise erzeugt man spezialisierte Computer-Hardware, die ihre Aufgaben selbst bei vergleichsweise niedrigen Taktfrequenzen effizienter erledigen als Allzweck-Computer.
- *C-to-Hardware-Synthese*: Es ist auch möglich, spezialisierte Hardware direkt anhand von C-Code (anstelle von VHDL, Verilog) zu generieren.

Literatur

- [1] <http://de.wikipedia.org/wiki/Rechnertechnik>, abgerufen am 7. 10. 2012
- [2] <http://www.robotrontechnik.de/html/computer/analogrechner.htm>,
<http://www.heise.de/tp/artikel/36/36877/1.html>, beides abgerufen am 7. 10. 2012