Grundlagen der Wirtschaftsinformatik

Bernd Ulmann[a]

01-JAN-2004

Jede kommerzielle Nutzung der folgenden Folien ist untersagt. Die Veröffentlichung, auch auszugsweise, bedarf der Zustimmung des Autors.

[a] ulmann@vaxman.de

Rechnergrundklassen

❶ Analogrechner

- Werte werden als kontinuierliche physikalische Größen dargestellt
- Nahezu ausgestorben

❷ Digitalrechner

- Werte werden in einem Zahlensystem dargestellt
- Werte sind somit nicht kontinuierlich, sondern diskret

Analogrechner

❶ Mechanische Analogrechner

- Rechenschieber: Addiert Längen, durch Tricks (Logarithmierung, spezielle Skalenbeschriftung, etc.) auch Multiplikation, Division und Berechnung besonderer Funktionen möglich
- Planimeter: Bestimmt Flächeninhalt von durch Kurven begrenzten, ebenen Gebieten

❷ Elektronische Analogrechner

- Im 2. Weltkrieg entwickelt (Helmut Hölzers Mischgerät)
- Grundelement ist der sogenannte Operationsverstärker

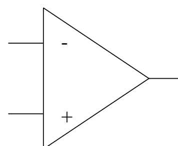
Der ideale Operationsverstärker

❶ Differentieller Eingang

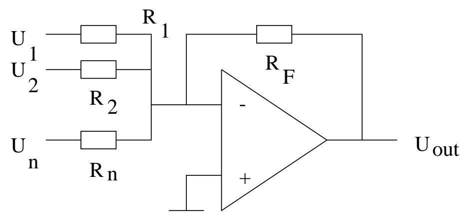
❷ Unendliche hoher Eingangswiderstand

❸ Unendlich hohe Leerlaufverstärkung

❹ Symbol:



Summierer



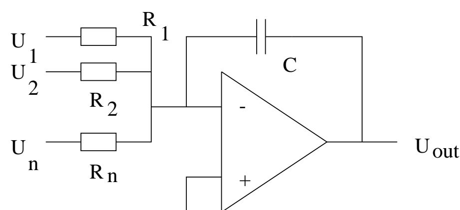
Die Ströme am Knotenpunkt des invertierenden Einganges müssen sich aufheben, da die Leerlaufverstärkung im Idealfall ∞ ist, d.h.

$$\sum_{i=1}^{n} \frac{U_i}{R_i} + \frac{U_{out}}{R_F} = 0$$

Sei $a_i = \frac{R_F}{R_i}$, so ergibt sich

$$-U_{out} = \sum_{i=1}^{n} a_i U_i.$$

Integrierer



Entsprechend ergibt sich

$$-U_{out}(t) = U_0 + \int_0^t \sum_{i=1}^{n} \frac{1}{R_i C} U_i \mathrm{d}t$$

Weitere wichtige Rechenelemente

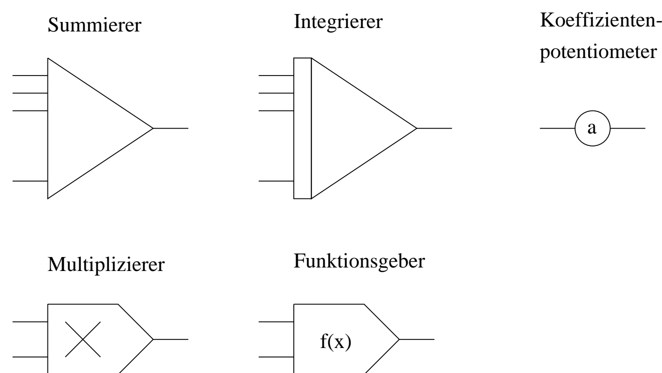
❶ Koeffizientenpotentiometer

❷ freie Verstärker (für implizite Funktionstechnik)

❸ Multiplizierer

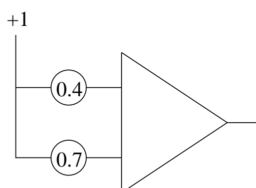
❹ Funktionsgeber

❺ Begrenzer, etc.



Beispiel einer Summation

Berechne -(7+4):



Vorsicht: Variablen können nur Werte innerhalb des Wertebereiches $[-1; +1]$ annehmen, wobei 1 je nach Maschinentyp zumeist 10V oder 100V entspricht, die Rechenschaltung muß also stets *normiert* werden, um zu verhindern, daß Überläufe auftreten.

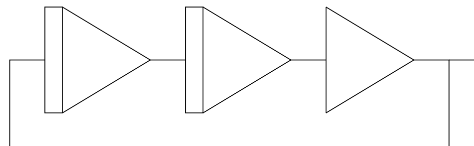
Differentialgleichungsbeispiel

Aufgabe: Erzeuge eine Sinusschwingung.

Idee: Verwende folgende Eigenschaft der Sinusfunktion:

$$\begin{aligned} \sin'(x) &= \cos(x) \\ \cos'(x) &= -\sin(x), \text{ d.h.} \\ \sin''(x) &= -\sin(x) \end{aligned}$$

Rechenschaltung:



Vorteile von Analogrechnern

- Hochgradig parallele Maschine, dadurch extrem schnell, je nach Problemstellung mitunter auch heutigen Digitalrechnern überlegen
- Sehr intuitive, dem Problem angemessene Programmierung
- Der (vermutlich) analogen Welt hervorragend angepaßt
- Ideal für Simulationen, z.B. in Luft- und Raumfahrt, Mechanik (schwingende Systeme), Kernreaktortechnik, etc.

Nachteile von Analogrechnern

- Kleiner Wertebereich von $[-1;+1]$
- Temperatur-, Zeitdrift und andere Ungenauigkeiten der Rechenelemente sind prinzipieller Natur
- Ausgestorben

Zukunft der Analogrechner

- Vielleicht große Zukunftschancen durch analoge FPGAs
- Sehr interessant in den Bereichen Robotik, AI, AL, etc.
- Möglicherweise Turingmaschinen überlegen!

Digitalrechner

- Werte werden nicht analog (kontinuierlich), sondern diskret mit Hilfe von Zahlensystemen dargestellt.
- Hierzu werden in der Regel Stellenwertsysteme verwendet, wobei sich als Basis 2 durchgesetzt hat.
- Die Steuerung geschieht mit Hilfe sogenannter *Programme*, deren Grundlage *Algorithmen* sind.
- Die steuernden Programme werden in einem Speicher, meist zusammen mit ihren Daten, abgelegt.

Zahlensysteme

❶ Nicht-Stellenwertsysteme

Gemeinsamkeit: Werte werden durch Abzählen dargestellt

Beispiele: Ägyptische Zahlen (siehe Papyrus Rhind), römische Zahlen, etc.

Hauptnachteile: Unhandliche Zahldarstellungen, Grundrechenarten sind nur schwer algorithmisch formulierbar, wie berechnet man beispielsweise MDXI·LVII?

❷ Stellenwertsysteme

- Begrenzte Anzahl verschiedener Ziffern
- Die Position einer Ziffer in einer Zahl bestimmt deren Wertigkeit. Beispiel:

$$\begin{aligned} 1234 &= 1 \cdot 1000 + 2 \cdot 100 + 3 \cdot 10 + 4 \\ &= 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0 \end{aligned}$$

Stellenwertsysteme

Allgemein gilt

$$w = \sum_{i=0}^{n-1} z_i b^i$$

w**:** Wert der Zahl

n**:** Anzahl der Ziffern der Zahl

z_i**:** i-te Ziffer

b**:** Basis des Zahlensystems (d.h. Anzahl unterschiedlicher Ziffern)

Das Binärsystem

- Die kleinste mögliche positive Basis für ein Stellenwertsystem ist 2
- Dies ist gleichzeitig die wichtigste Basis im Zusammenhang mit Digitalrechnern
- Das Zahlensystem mit der Basis $b = 2$ wird als *binäres* oder auch als *dyadisches* Zahlensystem bezeichnet
- Entwickelt wurde es (neben anderen) von Leibniz (1646–1716) im Jahre 1696

„Wunderbarer Ursprung aller Zahlen aus 1 und 0, welcher ein schönes Vorbild gibet des Geheimnißes der Schöpfung, da alles von Gott und sonst aus Nichts, entstehet.“

Beispiele für Basisumrechnungen

Binär → Dezimal:

$$\begin{aligned} 1011_2 &= 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= 8 + 0 + 2 + 1 = 11_{10} \end{aligned}$$

Dezimal → Binär:

$11:2$	$=$	5	Rest	1	LSB (Least Significant Bit)
$5:2$	$=$	2	Rest	1	
$2:2$	$=$	1	Rest	0	
$1:2$	$=$	0	Rest	1	MSB (Most Significant Bit)

D.h. $11_{10} = 1011_2$.

Hexadezimal- und Oktaldarstellung

Meist werden Binärzahlen der Einfachheit halber als *Hexadezimal-* (d.h. zur Basis $b = 16$) beziehungsweise (heutzutage selten) als *Oktalzahl* (d.h. zur Basis $b = 8$) dargestellt:

bin	okt	hex	bin	okt	hex	bin	okt	hex	bin	okt	hex
0000	0	0	0100	4	4	1000	10	8	1100	14	C
0001	1	1	0101	5	5	1001	11	9	1101	15	D
0010	2	2	0110	6	6	1010	12	A	1110	16	E
0011	3	3	0111	7	7	1011	13	B	1111	17	F

Beispiel: 0x3F2A$=$0011111100101010$_2$ $=$037452$_8$.

Rechnen im Binärsystem

Wie in jedem anderen Stellenwertsystem auch. Beispiele:

Addition:

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

Multiplikation:

$$\begin{array}{ccccccccc} 0 & 1 & 0 & 1 & \cdot & 0 & 1 & 1 & 0 \\ \hline 0 & 0 & 0 & 0 & & & & & \\ & 0 & 1 & 0 & 1 & & & & \\ & & 0 & 1 & 0 & 1 & & & \\ & & & 0 & 0 & 0 & 0 & & \\ \hline 0 & 0 & 1 & 1 & 1 & 1 & 0 & & \end{array}$$

Warum das Binärsystem?

Heutige Digitalrechner setzen fast ausschließlich das Binärsystem ein:

- Es ist wesentlich einfacher, Rechen- und Speicherelemente zu konstruieren und zu implementieren, die zwischen 2 Zuständen unterscheiden können, als zwischen 10 Zuständen
- Hierfür wird auch in Kauf genommen, daß Eingabewerte vor ihrer Verarbeitung und Ausgabewerte vor ihrer Ausgabe umgerechnet werden müssen.
- In manchen Fällen rechnen auch heutige Maschinen im Dezimalsystem, wobei allerdings jede Ziffer als Folge von vier Bit dargestellt wird (*BCD*).

Addition zweier Bits a und b (Halbaddierer)

Additionstabelle:

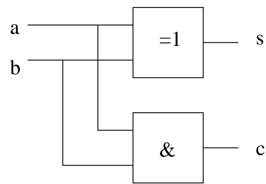
a	b	Σ	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

D.h.

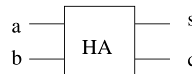
$$\begin{aligned} \Sigma &= (a \wedge \neg b) \vee (\neg a \wedge b) = a \oplus b \quad \text{und} \\ c &= a \wedge b \end{aligned}$$

Erhält ein Addierer zusätzlich zu a und b noch einen Übertrag der vorangehenden Stelle, handelt es sich um einen sogenannten *Volladdierer*.

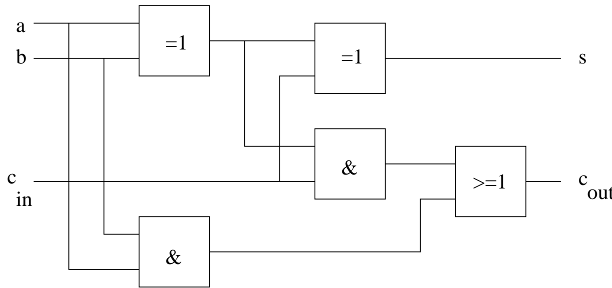
Halb- und Volladdierer



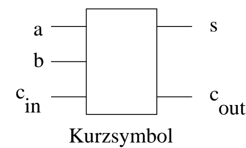
Implementation
eines Halbaddierers



Kurzsymbol

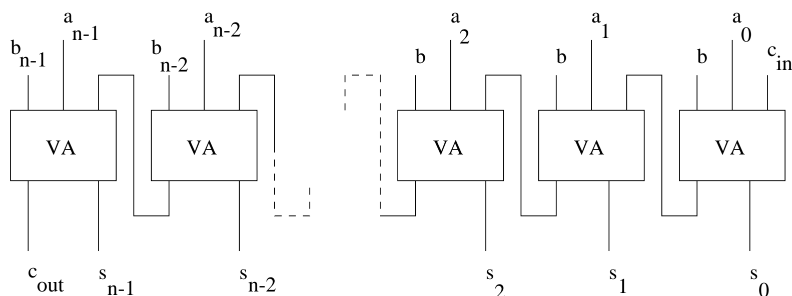


Implementation
eines Volladdierers



Kurzsymbol

Ripple-Carry-Adder



Ein solcher Addierer („to ripple“=„dahinplätschern“) implementiert also den einfachsten möglichen Additionsmechanismus: Beginnend von rechts (LSB) wird Stelle für Stelle verarbeitet, wobei pro Zeitschritt ein Übertrag einer Stelle an die nachfolgende Stelle weitergegeben wird.

Ein Ripple-Carry-Addierer stellt die langsamste Implementation eines Addierers dar und ist für Hochleistungsrechner nicht brauchbar.

Der asynchrone Addierer

- Eine interessante Schaltung ist der sogenannte *asynchrone* Addierer: Hier werden alle Bits zweier Werte parallel zueinander addiert, wobei die bei jeder Stelle möglicherweise auftretenden Überträge in einem eigenen *Register* gespeichert werden, während die Zwischensumme ebenfalls in einem Register abgelegt wird.
- Nach dem ersten Durchlauf nehmen die beiden in den Summen- und Übertragsregistern abgelegten Werte die Stelle der Summanden ein, wobei zuvor die Übertragsbits um eine Stelle nach links verschoben werden.
- Dieser Vorgang der parallelen Summenbildung ohne direkte Übertragsberücksichtigung wird so lange wiederholt, bis das Übertragsregister den Wert 0 enthält.
- Eine solche Schaltung wurde beispielsweise in DIGITALs PDP-6 eingesetzt.

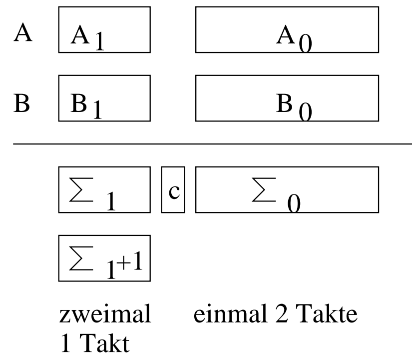
Beispiel einer asynchronen Addition

1. Schritt		2. Schritt		3. Schritt		4. Schritt	
A	0101	A	0110	A	0100	A	0000
B	+ 0011	B	+ 0010	B	+ 0100	B	+ 1000
Σ	0110	Σ	0100	Σ	0000	Σ	1000
C	0001	C	0010	C	0100	C	0000

Im Normalfall sind hierbei deutlich weniger Additionszyklen notwendig, als der Länge von A und B in Bit entspricht.

Carry-Select-Adder

- A und B seien zwei Binärzahlen mit jeweils $3n,\ n \in \mathbb{N}$ Ziffern (ein Ripple-Carry-Addierer würde für die Addition von A und B also $3n$ Takte benötigen).
- A und B werden nun in je zwei Teile mit $2n$ und n Bits unterteilt (im folgenden mit A_0, B_0 beziehungsweise A_1 und B_1 bezeichnet).



Carry-Select-Adder

- Nun werden parallel zueinander die Summen

$$\Sigma_0 = A_0 + B_0 \quad \text{sowie}$$
$$\Sigma_1 = A_1 + B_1 \quad \text{bzw.} \quad \Sigma_1 + 1 = A_1 + B_1 + 1$$

berechnet, wobei hierfür jeweils $1 \cdot 2n$ beziehungsweise $2 \cdot n$ Takte benötigt werden.

- Die drei Teilsummen stehen also nach nur $2n$ Takten und nicht, wie bei einem Ripple-Carry-Addierer nach $3n$ Takten zur Verfügung.
- Falls die Addition $A_0 + B_0$ keinen Übertrag c erzeugte, wird als Ergebnis die Bitfolge $\boxed{\Sigma_0}\boxed{\Sigma_1}$ ausgegeben.
- Entstand hingegen ein Übertrag c, wird als Ergebnis $\boxed{\Sigma_1 + 1}\boxed{\Sigma_0}$ ausgegeben.

Carry-Lookahead-Adder

- In modernen Rechnern finden fast ausschließlich sogenannte *Carry-Lookahead-Adder* Anwendung.
- Die Grundidee hierbei besteht darin, die für die Berechnung der Summe benötigten Übertragsbits möglichst im Voraus zu erkennen, so daß nicht von Stelle zu Stelle ein Übertrag weitergereicht werden muß.
- Angenommen, alle Übertragsbits c_i seien bekannt, so könnte die Summe zweier n Bit langer Zahlen A und B bitweise parallel, d.h. im Idealfall in einem einzigen Takt berechnet werden:

$$\Sigma_i = a_i \oplus b_i \oplus c_i$$

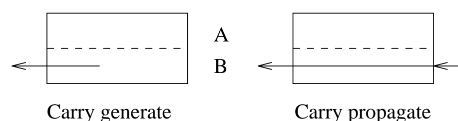
- Leider sind die Übertragsbits c_i im allgemeinen nicht bekannt.

Carry-Lookahead-Adder

Allerdings läßt sich diesbezüglich doch etwas tricksen:

Carry generate: Ein zusammenhängender Block korrespondierender Bits von A und B liefert bei seiner Addition einen Übertrag.

Carry propagate: Ein zusammenhängender Block korrespondierender Bits von A und B liefert bei seiner Addition einen Übertrag, falls die Addition selbst einen Übertrag des vorangehenden Blocks erhielt, d.h. der Übertrag wird durch den Block hindurchgereicht.



Carry-Lookahead-Adder

Für das letzte Übertragsbit c_n gilt also

$$c_n = g(0, n-1) \vee c_0 p(0, n-1),$$

wobei $g()$ und $p()$ die Generate- beziehungsweise Propagatefunktionen darstellen.

Trick: $g()$ und $p()$ lassen sich rekursiv berechnen:

$\forall i \leq k < j$:

$$\begin{aligned} g(i,j) &= g(k+1,j) \vee (g(i,k) \wedge p(k+1,j)) \quad \text{und} \\ p(i,j) &= p(i,k) \wedge p(k+1,j) \end{aligned}$$

$i = j$:

$$g(i,i) = a_i \wedge b_i \quad \text{und}$$

$$p(i,i) = a_i \vee b_i$$

Carry-Lookahead-Adder

- Durch Unterteilung von A und B in beispielsweise 4 Bit große Blöcke können $g()$ und $p()$ für jeden dieser Blöcke verhältnismäßig einfach berechnet werden.
- Jeweils vier solcher Blöcke können nun wieder – mit entsprechender Berechnung von $g()$ und $p()$ auf dieser höheren Stufe – zusammengefaßt werden, usf.
- Insgesamt lassen sich so sehr effizient Summen langer Zahlen A und B berechnen – Aufwände im Bereich weniger Takte für 32 Bit lange Werte sind durchaus üblich.

Subtraktion

Soviel zu Additionstechniken – neben den exemplarisch genannten Verfahren existiert eine Vielzahl anderer Techniken, die aber in der Mehrzahl keine große Bedeutung erlangen konnten.

Was ist mit der nächsten Grundoperation, der Subtraktion?

- Der Bau entsprechender Schaltungen für die Berechnung von $A - B$ ist analog zu Addierern möglich.
- Explizit ausgeführte Subtrahierer und Addierer würden jedoch in der Regel eine schlechte Ausnutzung der zur Verfügung stehenden Hardware bedeuten, da in der Regel nur eine von zwei solchen Einheiten aktiv wäre.
- Besser wäre es also, ohne einen Subtrahierer auszukommen.
- Funktioniert so etwas? Ja, aber – wie so oft – mit einem Trick: Anstelle von $A - B$ wird $A + (-B)$ berechnet.

Die Berechnung von $-B$

- Allgemein gilt zunächst $-B = 0 - B$, was leider genauso schwierig zu berechnen ist wie die Bildung der letztlich angestrebten Differenz $A - B$.
- Vereinfachend kommt nun aber hinzu, daß Digitalrechner aufgrund der prinzipiell beschränkten Länge der von ihnen verarbeiteten Werte stets in einem Ring operieren.
- Mit n Bit langen Zahlen, $n \in \mathbb{N}$, lassen sich 2^n verschiedene Werte $0 \ldots 2^n - 1$ darstellen, die beispielsweise, ausgehend von 0, durch fortgesetzte Addition von 1 erhalten werden können.
- Da nur 2^n verschiedene Werte zur Verfügung stehen, ergibt sich als Ergebnis der Addition von 1 zu $2^n - 1$ der Wert 0, da keine weiteren Werte zur Verfügung stehen.

Der Restklassenring $\mathbb{Z}/m\mathbb{Z}$

- Das Ergebnis der Operation $n \mod m$ ist der Rest, der sich nach ganzzahliger Division von n durch m ergibt – so ist beispielsweise $17 \mod 2 = 1$.
- Zwei Zahlen a und b gehören ein und derselben *Restklasse modulo* m an, wenn gilt:
$$a \mod m = b \mod m.$$
- Gilt beispielsweise $m = 2$, so entsprechen die beiden Restklassen, in die sich die ganzen Zahlen modulo 2 einteilen lassen, einfach den geraden beziehungsweise ungeraden Zahlen.
- Auf der Menge der Restklassen lassen sich nun eine Addition sowie eine Multiplikation definieren, mit denen zusammen man einen sogenannten *Ring*, genauer den *Restklassenring* $\mathbb{Z}/m\mathbb{Z}$ erhält.

Der Restklassenring $\mathbb{Z}/m\mathbb{Z}$

- Der Ring $\mathbb{Z}/m\mathbb{Z}$ enthält also genau m verschiedene Zahlen: $0 \ldots m - 1$
- Ein Beispiel für einen solchen Restklassenring ist die Unterteilung von Tagen in 24 Stunden – die 24 verschiedenen Elemente sind $0 \ldots 23$, nach der dreiundzwanzigsten Stunde eines Tages folgt wieder die nullte Stunde (des nächsten Tages, was aber ohne Belang ist, da die Nummer des Tages quasi ein Übertragszähler ist, der in einem Ring nicht vorkommt).
- Interessanter sind natürlich Ringe mit $m = 2^n$, wobei n der Länge der ganzen Zahlen, welche ein Digitalrechner verarbeitet, entspricht.
- Eine 16-Bit-Maschine, wie beispielsweise ein Exemplar der PDP-11-Serie, kann folglich mit einem 16 Bit Maschinenwort $2^{16} = 65536$ verschiedene Werte darstellen.

Die Berechnung von $-B$

- Denkt man sich nun die 2^n verschiedenen Werte zu einem Kreis zusammengebogen, der wiederspiegelt, daß nach einem Überlauf wieder die Null folgt, wird klar, daß $-B$ innerhalb eines solchen Ringes $\mathbb{Z}/m\mathbb{Z}$ als $-B = 2^m - B$ dargestellt werden kann!
- Der eigentliche Trick besteht nun darin, die Differenz $2^m - B$ zu berechnen, ohne wirklich eine Subtraktion durchzuführen.
- 2^n hat stets die Form $10\ldots0_2$, was für die Berechnung der Differenz eher unangenehm ist, da, falls B nicht gerade 0 ist, viele Überträge zu berücksichtigen sind.
- Macht man sich jedoch klar, daß $2^n = (2^n - 1) + 1$ gilt, wird die Berechnung plötzlich ausgesprochen einfach:

$$-B = 2^n - B = (2^n - 1) + 1 - B = (2^n - 1) - B + 1$$

Die Berechnung von $-B$

- Der große Vorteil ist nun nämlich, daß $2^n - 1$ stets die Form $1\ldots1_2$ besitzt, d.h. bei der Berechnung von $(2^n - 1) - B$ kann an keiner Stelle ein Übertrag auftreten.
- Dies bedeutet jedoch, daß die Differenzbildung stellenparallel in nur einem einzigen Schritt ausgeführt werden kann, wobei letztlich B nur bitweise invertiert wird.
- Im Anschluß hieran ist noch 1 zu addieren, um die Repräsentation von $-B$ im Restklassenring $\mathbb{Z}/m\mathbb{Z}$ zu erhalten.
- Diese Darstellung von $-B$ als $2^n - B$ wird als *Zweierkomplement* bezeichnet.

Die Berechnung von $-B$

- Das *Einerkomplement* einer Zahl B wird durch Berechnung von $(2^n - 1) - B$ berechnet.
- Durch Fortlassen der Addition von 1 im letzten Schritt, kann (mindestens) ein Takt eingespart werden.
- In den 60er-Jahren rechneten fast alle Maschinen von Seymour Cray, d.h. die CDC-1604, CDC-160, CDC-6000, -7000 Baureihe, etc. nicht im Zweier-, sondern im Einerkomplement.
- Der Hauptnachteil des Einerkomplementes ist die Einführung zweier verschiedener Werte für Null, nämlich $+0$ und -0 – ein Effekt, der im Zweierkomplement nicht auftritt.
- Heutzutage kann das Einerkomplement im wesentlichen als ausgestorben angesehen werden.

Beispiel

Das folgende Beispiel soll das Vorgehen anhand der Subtraktion 0x53-0x36 verdeutlichen, die zunächst direkt, und dann über die Bestimmung des Zweierkomplementes von -0x36 durchgeführt wird (in diesem Beispiel wird stets mit 8 Bit langen Werten gerechnet!):

Direkt:

01010011	0x53	83
- 00110110	- 0x36	- 54
00011101	0x1d	29

Beispiel

Über das Zweierkomplement: Berechne zunächst -0x36:

$$\begin{aligned} -00110110 &= (11111111 - 00110110) + 1 \\ &= 11001001 + 1 \\ &= 11001010 \\ &= 0xCA \end{aligned}$$

Nun Berechnen von $0x53 + 0xCA$:

```
    01010011
+   11001010
------------
    00011101
```

Da alle Werte eine Länge von 8 Bit besitzen, wird der bei obiger Addition auftretende höchste Übertrag nicht berücksichtigt!

Zusammenfassung

- Heutige Digitalrechner führen die Berechnung von Differenzen der Form $A - B$ in fast allen Fällen auf die hierzu äquivalente Rechnung $A + (-B)$ zurück, wobei $-B$ durch Bildung des Zweierkomplementes von B erzeugt wird.
- Zur Bildung des Zweierkomplementes einer Binärzahl B gegebener Länge werden zunächst alle Bits von B invertiert, woran sich die Addition des Wertes 1 anschließt.
- Die so erhaltene Zweierkomplementdarstellung von $-B$ wird nun zu A addiert, wobei eventuell auftretende Überträge, die über die Länge der Zahlen A und B hinausgehen, verworfen werden.

Ein kleiner Zahlenstrahl

Die folgende Abbildung zeigt einen Zahlenstrahl für 3 Bit lange Werte unter Berücksichtigung des Zweierkomplementes:

-4	-3	-2	-1	0	1	2	3
100	101	110	111	000	001	010	011

Hierbei fallen drei wesentliche Punkte ins Auge:

- Die betragsmäßig größte negative Zahl besitzt kein positives Gegenstück.
- n Bit lange Zahlen w können also folgende Werte annehmen:

$$-2^{n-1} \leq w \leq 2^n - 1.$$

- Das höchstwertige Bit (das MSB) läßt sich als Vorzeichenbit ansehen.

Gleitkommazahlen

- Die den nun eingeführten ganzen Zahlen (im Zweierkomplement) sind für viele, aber nicht alle in der Praxis auftretenden Berechnungen geeignet.
- Vor allem in naturwissenschaftlichen Berechnungen treten häufig Werte stark unterschiedlicher Größenordnung innerhalb einer einzigen Rechnung auf, so ist es beispielsweise bei der Integration von Funktionen nicht unüblich, zu großen Werten sehr kleine Inkremente zu addieren.
- Um beispielsweise die Rechnung $10^6 + 10^{-6}$ durchführen zu können, wären bei ausschließlicher Verwendung ganzer Zahlen (mit Vorzeichen, d.h. im Zweierkomplement) Längen von 45 Bit notwendig – bei realen Beispielen noch wesentlich mehr.
- Somit ist im Zusammenhang mit derlei Berechnungen eine Verwendung ganzer Zahlen in Binärdarstellung nicht praktikabel.

Gleitkommazahlen

Abhilfe schaffen hier die sogenannten *Gleitkommazahlen*

Idee:

- Alle Werte werden in eine *Mantisse* m sowie einen *Exponenten* e mit Basis b aufgespalten.
- Für die Mantisse m gilt im dezimalen Fall stets

$$1 \leq m < 10,$$

während für binäre Mantissen (d.h. den Normalfall) gilt:

$$1 \leq m < 2.$$

- Gleitkommazahlen, deren Mantisse dieser Bedingung entspricht, heissen *normalisiert*.

Gleitkommazahlen

Im folgenden wird, der Einfachheit- und Gewohnheit halber stets mit dezimalen Werten gearbeitet, für die eine Mantissenlänge von 4 Ziffern bei einer Exponentenlänge von 2 Ziffern und Basis $b = 10$ gilt.

Beispiele:

$$\begin{aligned} 1 &\rightarrow 1.0 \cdot 10^0 \\ 1000000000000 &\rightarrow 1.0 \cdot 10^{12} \\ 0.000000000001 &\rightarrow 1.0 \cdot 10^{-12} \\ 1234 &\rightarrow 1.234 \cdot 10^3 \end{aligned}$$

Gleitkommazahlen

- Die Mantisse m einer Gleitkommazahl enthält also die *signifikanten* Stellen dieser Zahl, während
- der Exponent die Verschiebung dieser Stellen im Wertebereich vollzieht.
- Durch die (naturgemäß) beschränkte Mantissenlänge ist jedoch die Darstellungsgenauigkeit beschränkt, wie folgendes Beispiel zeigt:

$$12345 \rightarrow 1.234 \cdot 10^4 \rightarrow 12340$$

Grundrechenarten bei Gleitkommazahlen

- Bei Addition und Subtraktion müssen zunächst die Exponenten der Summanden einander angepaßt werden:

$$\begin{array}{lr} & 10 \\ + & 1 \\ \hline & 1.0 \cdot 10^1 \\ + & 1.0 \cdot 10^0 \\ \hline & 1.0 \cdot 10^1 \\ + & 0.1 \cdot 10^1 \\ \hline & 1.1 \cdot 10^1 \end{array}$$

Grundrechenarten bei Gleitkommazahlen

Beim Rechnen mit Gleitkommazahlen gilt das Kommutativgesetzt nicht, es ist also (am Beispiel der Addition) $A + B \neq B + A$.

Beispiel: Im folgenden ist die Mantissenlänge wieder 4 – zu berechnen ist $1234 + 0.5 + 0.5$ (Addition von unten nach oben):

$$\begin{array}{lr} & 1.234 \cdot 10^3 \\ + & 5 \cdot 10^{-1} \\ + & 5 \cdot 10^{-1} \\ \hline & 1.234 \cdot 10^3 \\ + & 0.001 \cdot 10^3 \\ \hline & 1.235 \cdot 10^3 \end{array} \qquad \begin{array}{lr} & 5 \cdot 10^{-1} \\ + & 5 \cdot 10^{-1} \\ + & 1.234 \cdot 10^3 \\ \hline & 0.0005 \cdot 10^3 \\ + & 0.0005 \cdot 10^3 \\ + & 1.234 \cdot 10^3 \\ \hline & 1.234 \cdot 10^3 \end{array}$$

Gleitkommazahlen

- Prinzipiell ist beim Rechnen mit Gleichkommazahlen stets mit größter Vorsicht vorzugehen!
- Werden Zahlen addiert oder subtrahiert, ist stets mit den betragsmäßig kleinsten Werten zu beginnen und mit den betragsmäßig größten zu enden, wie das vorangegange Beispiel zeigt.
- Kann eine reelle Zahl nicht als Gleitkommazahl dargestellt werden, muß also zur nächstgelegenen Gleitkommazahl gerundet werden, so spricht man von einem *Rundungsfehler*.

Beispiel: Der dezimale Wert 0.1 läßt sich nicht ohne Rundungsfehler als binäre Gleitkommazahl darstellen (periodischer Bruch):

$$0.1 = \frac{1}{10} = \frac{0}{2^0} + \frac{0}{2^1} + \frac{0}{2^2} + \frac{0}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \ldots$$

IEEE-754 Gleitkommazahlen

Für Gleitkommazahlen, welche der IEEE-754-Norm entsprechen, gilt:

- Die Mantisse m erfüllt stets die Bedingung $1 \leq m < 2$, d.h. das höchstwertige Bit der Mantisse (das MSB) ist stets 1.
- Dieses Bit wird nicht gespeichert, sondern fest als 1 vorausgesetzt (sogenannte *hidden one*).
- Die Mantisse wird nicht als Zahl im Zweierkomplement dargestellt – vielmehr fungiert lediglich das oberste Bit als Vorzeichenbit.

Die beiden häufigsten Formate sind:

Single precision:

Vorzeichen (1 Bit)	Exponent (8 Bit)	Mantisse (23 Bit)

Dezimale Genauigkeit: ≈ 7 Ziffern

Double precision:

Vorzeichen (1 Bit)	Exponent (11 Bit)	Mantisse (52 Bit)

Dezimale Genauigkeit: ≈ 15 Ziffern

Zeichenrepräsentation

Nach den drei grundlegenden numerischen Datentypen

- vorzeichenlosen ganze Zahlen,
- ganze Zahlen mit Vorzeichen (Zweierkomplement) und
- Gleitkommazahlen

fehlt nun eine Möglichkeit zur Repräsentation von Zeichen.

Idee: Jedem darstellbaren Zeichen wird ein ganzzahliger binärer Wert zugeordnet.

Historisch: 6 Bit/Zeichen (z.B. Control Data Corporation) $\rightarrow$ 64 verschiedene Zeichen möglich.

In der Folge: 8 Bit/Zeichen $\rightarrow$ 256 mögliche Zeichen.

ASCII/EBCDIC

Für die Zuordnung der Zeichen müssen verbindliche Regeln gelten, um Daten austauschbar zu halten. Leider haben sich historisch zwei verschiedene Varianten herausgebildet:

ASCII: *American Standard Code for Information Interchange* – offizielle Norm, eigentlich ein 7 Bit-Code $\rightarrow$ 128 verschiedene Zeichen. Fast alle heutigen Maschinen setzen ASCII ein.

EBCDIC: *Extended Binary Coded Decimal Interchange Code* – entwickelt von IBM, eingesetzt beispielsweise bei S/390, z-Serie, AS/400.

ASCII und EBCDIC sind vollständig inkompatibel zueinander!

Unicode

Um diese Inkompatibilitäten zu vermeiden und vor allem auch Sprachen, die mehr als 128 verschiedene Zeichen zu ihrer Darstellung benötigen, zu unterstützen, wurde *Unicode* entwickelt. Ursprünglich waren 16 Bit/Zeichen vorgesehen, mittlerweile haben sich drei Varianten herausgebildet:

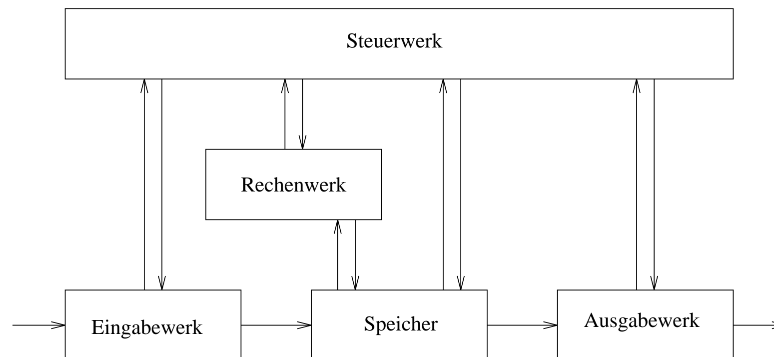
UTF-8: Variable Längenkodierung, ASCII als Subset, maximal 3 Byte/Zeichen, im HTML-Umfeld verbreitet.

UTF-16: 16 Bit/Zeichen $\rightarrow$ 65336 verschiedene Zeichen.

UTF-32: 32 Bit/Zeichen $\rightarrow$ 4294967296 verschiedene Zeichen möglich.

von Neumann-Rechner

- 1946 schlug der US-amerikanische Mathematiker John von Neumann (1903–1957)ein Konzept für einen Universalrechner vor, dem die Mehrzahl aller heutigen Digitalrechner noch immer folgen.
- Rechner dieser Struktur (Architektur) werden als *von Neumann-Rechner* bezeichnet. Ihr Aufbau hat folgende Gestalt:



von Neumann-Rechner

- Der Aufbau eines solchen Rechners ist unabhängig von der Art des mit seiner Hilfe zu lösenden Problems.
- Die Steuerung der Maschine und damit die Lösung eines Problemes geschieht mit Hilfe einer Bearbeitungsvorschrift, eines sogenannten *Programmes*.
- Sowohl das steuernde Programm als auch alle Daten, auf denen dieses operiert, werden in ein und demselben Speicher abgelegt.
- Der Speicher ist in *Speicherzellen* gleicher Größe unterteilt, die durch eindeutige, fortlaufende *Adressen* ausgewählt und angesprochen werden können.
- Ob es sich bei im Speicher abgelegten Werten um *Daten* oder um *Instruktionen* handelt, ist Interpretationssache.

von Neumann-Rechner

- Normalerweise werden die Befehle eines Programmes der Reihe nach abgearbeitet, wobei ein sogenannter *Programmzähler* für die Bereitstellung der jeweils aktuellen Adressen sorgt.
- Muß an einer Stelle von diesem sequentiellen Programmfluß abgewichen werden, wird dies mit Hilfe bedingter oder unbedingter *Sprungbefehle* erzielt.
- von Neumann-Rechner gehören in die Kategorie der *SISD*, *Single Instruction Single Data* Maschinen.
- Neben dieser Architektur gibt es eine Vielzahl anderer Rechnerstrukturen, von denen jedoch keine eine solche Verbreitung wie die von Neumann-Rechner erlangt hat. Ein Beispiel hierfür sind Rechner mit *Harvard Architektur*, bei denen Programm und Daten in voneinander getrennten Speichern abgelegt werden – *digitale Signalprozessoren*, kurz *DSP*s, sind typische Vertreter solcher Maschinen.

Das Steuerwerk

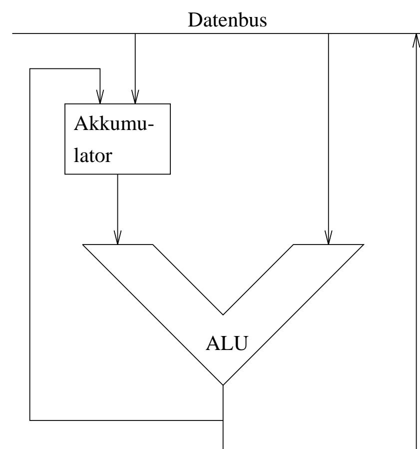
- Das *Steuerwerk* bildet sozusagen das Herz eines Digitalrechners.
- Ihm obliegt die gesamte Steuerung des Rechners.
- Das Steuerwerk beinhaltet normalerweise mindestens den Programmzähler, das *Befehlsregister*, in welchem die aus dem Speicher gelesenen Instruktionen abgelegt werden, den *Befehlsdekoder*, einen *Adressrechner*, sowie eine *Mikroprogrammeinheit* beziehungsweise eine *Ablaufsteuerung*.

Das Rechenwerk

- Das sogenannte *Rechenwerk* setzt sich (meist beziehungsweise mindestens) zusammen aus
  - der arithmetisch logischen Einheit, kurz *ALU*, welche (in der Regel) eine Reihe von Operationen auf zwei Eingabeoperanden vollzieht, deren Ergebnis ein Ausgabeoperand sowie einige *Statusflags* sind, welche spezielle Bedingungen wie *Übertrag*, *Überlauf*, ein Ergebnis des Wertes Null, ein negatives Resultat, etc. wiederspiegeln.
  - Darüberhinaus beinhaltet das Rechenwerk meist einen *Akkumulator* – hierbei handelt es sich um ein *Register*, das Werte aufnimmt und sowohl als Datenquelle als auch als Datensenke für die ALU dient.
  - Die meisten modernen Digitalrechner verfügen über eine Vielzahl solcher Akkumulatoren, welche dann als *Register* bezeichnet werden.

Das Rechenwerk

Die folgende Abbildung zeigt ein stark vereinfachtes Rechenwerk:



Die Ein- und Ausgabewerke

- Die *Ein-* beziehungsweise *Ausgabewerke* stellen die Schnittstellen zwischen dem Rechner und der Außenwelt dar.
- Je nach Ausführung der Maschine sind sie in einer Form implementiert, welche sie wie normale Speicherzellen ansprechbar macht (*memory mapped I/O*).
- Mitunter enthalten die Ein-/Ausgabewerke ihrerseits wieder (kleine) Rechner, um selbständig komplexe Anfragen ausführen zu können – Pionier hier war IBM mit der Schaffung der Channel-Schnittstellen.
- Beispiele für Ein-/Ausgabewerke sind unter (vielem) anderem: Plattencontroller, serielle und parallele Schnittstellen (an welche beispielsweise Terminals, Tastaturen, Mäuse, etc. angeschlossen werden können), Grafikcontroller, etc.

Der Speicher

- Da die Ausführungsgeschwindigkeiten von Speicher auf der einen und Rechen- beziehungsweise Steuerwerk auf der anderen Seite stark unterschiedlich sind, wird anstelle eines einzigen Speicher eine ganze *Speicherhierarchie* eingesetzt.
- Die Idee hierbei ist, in der Nähe des Steuer- und Rechenwerkes einen sehr schnellen, aber damit auch verhältnismäßig kleinen Speicher einzusetzen, um hinreichend schnell Instruktionen und Daten liefern zu können und so Leerlaufzeiten zu vermeiden.
- Da dieser schnelle Speicher meist zu klein ist, um alle notwendigen Daten und Instruktionen eines Programmes zu fassen, wird er wiederum von einem weiteren Speicher, der allerdings etwas langsamer und dafür größer ist, mit Werten versorgt.
- Dieses Schema kann über mehrere Stufen hinweg fortgesetzt werden.

Hauptspeicher und Caches

- Der eigentliche Speicher eines Rechners wird im allgemeinen als *Hauptspeicher* bezeichnet. Er ist verhältnismäßig langsam (mit *Zugriffs*- und *Zykluszeiten* im Bereich einiger 10ns), dafür aber von hoher *Speicherkapazität*, die bis in den Bereich einiger GB reichen kann.
- Die nächste Hierarchiestufe ist zumeist der sogenannte *Second Level Cache* mit kleineren Zugriffs-/Zykluszeiten, dafür aber auch deutlich geringerer Kapazität.
- In der Regel folgt nun noch ein *First Level Cache*, der oftmals auf demselben Chip wie der eigentliche Prozessor oder zumindest in unmittelbarer räumlicher Nähe desselben angeordnet ist.
- Jeder Cache beinhaltet also Kopien von Ausschnitten des ihm vorangehenden Speicher und führt in eigenen Tabellen Buch, welche Adressbereiche als Kopie vorliegen und welche nicht.

Hauptspeicher und Caches

- Fordert der Prozessor (kurz *CPU* für *Central Processing Unit*) Daten vom Speicher an, die sich nicht in dem ihm unmittelbar vorangestellten Cache (meist also dem first level Cache) befinden, resultiert dies in einem sogenannten *Cache Miss*.
- Ein solcher Cache Miss hat zur Folge, daß die Verarbeitung kurz angehalten wird und der angeforderte Speicherbereich in Form einer (oder mehrerer) *Cache Lines* in den betreffenden Cache kopiert werden.
- Unter Umständen müssen hierzu Bereiche des Caches geleert werden, um Platz für die neuen Daten zu schaffen. In diesen Fällen muß entschieden werden, ob die dort enthaltenen Daten einfach verworfen werden können, oder in den eigentlichen Hauptspeicher zurückkopiert werden müssen, was der Fall ist, wenn die betreffenden Daten im Cache modifiziert wurden.

Interleaving

- Um die Geschwindigkeit eines Speichers, mit der auf Anfragen reagiert werden kann, zu erhöhen, wird er meist in *Bänke* gleicher Größe unterteilt.
- Die Größe einer Bank ist stets eine reine Zweierpotenz.
- Wird beispielsweise ein Speicher von 64kB Größe in zwei Bänke zu jeweils 32kB unterteilt, so kann das unterste Bit einer Speicheradresse zur Auswahl der betroffenen Bank herangezogen werden, so daß eine Bank alle geraden und die andere Bank alle ungeraden Adressen enthält.
- Wird nun sequentiell auf Speicherzellen zugegriffen, verteilen sich diese Zugriffe abwechselnd auf jeder der beiden Speicherbänke, so daß ein Zugriff vorbereitet werden kann, während sich der vorangehende (auf der jeweils anderen Bank) noch in der Verarbeitung befindet.

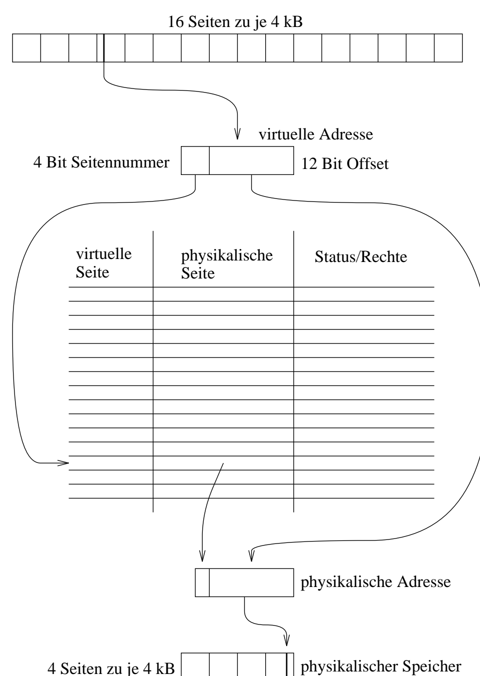
Interleaving

- In der Regel werden wesentlich mehr als nur zwei Speicherbänke eingesetzt – bei Hoch- und Höchstleistungsrechnern sind die Speicher in bis zu 128 Bänke unterteilt.
- Diese Unterteilung in Bänke wird auch als *Interleaving* bezeichnet.
- Der volle Effekt des Interleavings läßt sich nur erreichen, wenn möglichst sequentiell und damit abwechselnd auf die verschiedenen Bänke zugegriffen wird.
- Zugriffe mit festem *Stride* der im schlimmsten Fall der Anzahl zur Verfügung stehender Bänke oder Vielfachen hiervon entspricht, machen diesen Vorteil wieder zunichte, da alle Zugriffe auf eine oder nur wenige Bänke verteilt werden.

Virtueller Speicher

- 1962 implementiert Jack Kilburn in der Atlas-Maschine eine gemeinsame, transparente Verwaltung von Kern- und Trommelspeicher, die nach außen hin als ein einziger Speicher mit der Kapazität des Trommelspeichers erscheint.
- Ziel ist ein möglichst großer sogenannter *virtueller Speicher*, von dem im Normalfall nur ein kleiner Teil auch wirklich physikalisch vorhanden ist.
- Virtueller und physikalischer Speicher werden in sogenannte *Seiten* (oder auch *Pages*) unterteilt, die einander über eine oder mehrere Tabellen (*Seitentafeln* bzw. *Pagetables*) zugeordnet werden.
- Die Abbildung auf der folgenden Seite zeigt eine solche einstufige Umsetzung von einer virtuellen Adresse in eine physikalische Adresse anhand eines nicht sehr realistischen Beispiels mit 64kB virtuellem und 16kB physikalischem Speicher.

Beispiel einer einstufigen Adressumsetzung



Virtueller Speicher

- Durch die tabellengestützte, seitenbasierte Adressumsetzung von virtuellen Adressen in physikalische Adressen kann für jedes Programm der Eindruck erzeugt werden, ausschließlich ihm stünde der gesamte virtuelle Speicher zur Verfügung.
- Da der vorhandene physikalische Speicher in der Regel kleiner als der virtuelle Speicher ist, muß ein geeignetes Medium zur *Auslagerung* von Seiten vorgesehen werden, die augenblicklich keinen Platz im physikalischen Speicher finden können.
- Hierbei gibt es zwei Aus-/Einlagerungsmechanismen:

  Swapping: Alle Seiten eines Prozesses werden verlagert.

  Paging: Nur einzelne Seiten sind betroffen.

Seitenersetzungmechanismen

FIFO: *First In First Out* – die zuerst in den physikalischen Speicher gebrachten Seiten, werden auch als erste wieder ausgelagert, um Platz zu schaffen.

LIFO: *Last In First Out* – unbrauchbar!

LRU: *Least Recently Used* – die Seite, auf welche am längsten nicht zugegriffen wurde, wird ausgelagert.

LFU: *Least Frequently Used* – die Seiten, auf welche am seltensten zugegriffen wurde, wird ausgelagert.

Die LRU- und LFU-Verfahren erweisen sich in der Regel als am effizientesten, sind jedoch auch mit hohem Verwaltungsaufwand verbunden.

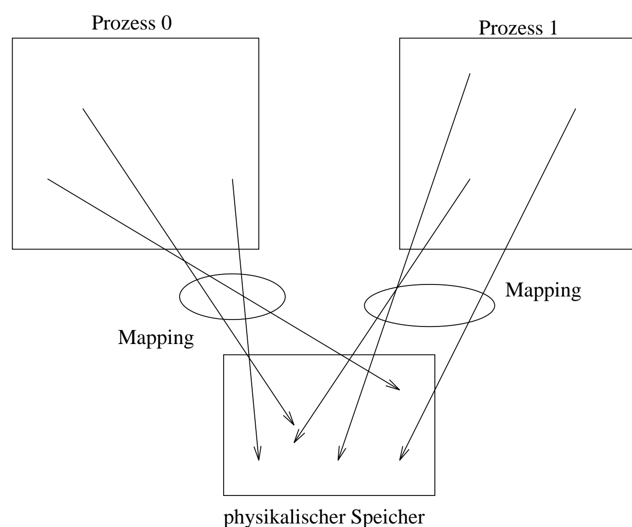
Virtueller Speicher

- Im Normalfall befindet sich zu einem Zeitpunkt also nur eine Teilmenge aller Speicherseiten eines Prozesses im physikalischen Speicher.
- Dies ist möglich, da Programme in der Regel sowohl zeitlich als auch räumlich ein hohes Maß an *Lokalität* besitzen:

  Räumliche Lokalität: Ein Großteil der Rechenzeit wird in wenigen, meist kleinen Programmabschnitten verbracht.

  Zeitliche Lokalität: Wurde auf eine Adresse zugegriffen, sind Folgezugriffe auf diese Adresse sehr wahrscheinlich.
- In der Regel besitzt jeder einzelne Prozess seinen eigenen *Adressraum* und somit seine eigenen Seitentafeln für die Adressumsetzung.

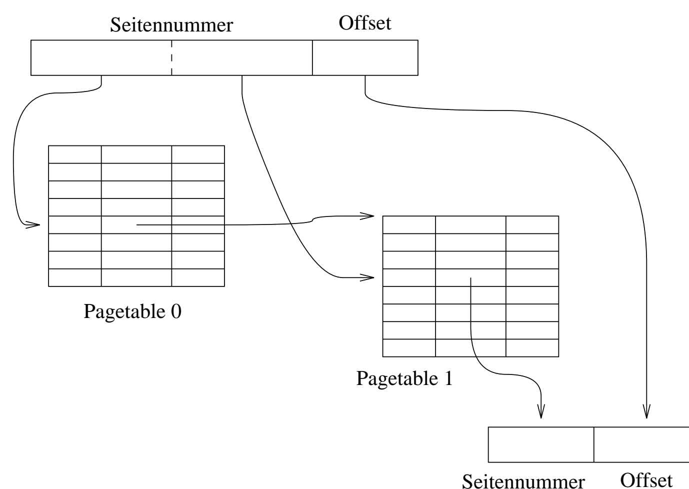
Adressräume



Mehrstufige Adressumsetzung

- Bei großen virtuellen und physikalischen Speichern werden die Seitentafeln bei einer einfachen, einstufigen Adressumsetzung schnell unhandlich groß, so daß in der Praxis meist eine mehrstufige Umsetzung implementiert wird.
- Die folgende Abbildung zeigt diesen Prozeß anhand einer zweistufigen Adressumsetzung:
  - Der linke Teil der virtuellen Seitennummer selektiert eine Zeile aus der obersten Seitentafel, mit deren Hilfe eine von n untergeordneten Seitentafeln ausgewählt wird.
  - Aus dieser zweiten Seitentafel wird mit dem rechten Teil der virtuellen Seitennummer ebenfalls eine Zeile ausgewählt, usf.

Mehrstufige Adressumsetzung



NICE

Im folgenden wird ein einfacher von Neumann-Rechner als Grundlage für einige Beispiele herangezogen. Bei diesem Rechner, *NICE*, handelt es sich um eine Maschine mit einer Wortlänge von 32 Bit, deren grundlegende Eigenschaften im folgenden kurz beschrieben werden[a]:

Register: 16 Register R0 bis R15, die von allen Instruktionen angesprochen werden können. Einige Register haben eine besondere Bedeutung:

- **R0:** Dieses Register liefert, wenn es als Quelle für Operanden verwendet wird, stets den Wert 0 zurück. Es kann jederzeit beschrieben werden.
- **R13:** Interruptregister – wird eine externe Unterbrechnung (ein sogenannter *Interrupt*) ausgelöst, wird hier eine Kopie des Programmzählers abgelegt.
- **R14:** Statusregister
- **R15:** Der Programmzähler – enthält die Adresse der nächsten Instruktion.

[a] Siehe *Designing a NICE processor*, in „Microprocessors and Microsystems“, 23 (1999) 257–264.

Das Statusregister

Die Struktur des Statusregister R14 ist wie folgt:

Die einzelnen Bits haben folgende Bedeutung:

1: Stets 1.

X: 1, wenn die letzte Instruktion 0xFFFFFFFF ergab.

C: 1, falls ein Übertrag auftrag.

Z: 1, wenn die letzte Instruktion 0x00000000 ergab.

N: 1, wenn das letzte Resultat < 0 war.

V: 1, wenn ein Overflow auftrat (d.h. Vorzeichenwechsel).

M: 1, wenn ein maskierbarer Interrupt auftrat.

I: Interruptsteuerbit, ist es 0, sind Interrupts ausgeschlossen.

Instruktionsformat

Alle NICE-Instruktionen weisen eine Länge von 32 Bit (beziehungsweise einem Vielfachen hiervon) auf und besitzen folgende Struktur:

Condition: Steuert die bedingte Ausführung der Instruktion.

Opcode: Code der auszuführenden Instruktion.

Dest: Adresse des Zieloperanden einer Instruktion.

Src: Adresse der Quelloperanden einer Instruktion.

Die folgenden Folien beschreiben die einzelnen Blöcke einer Instruktion detaillierter:

Bedingte Ausführung

Alle NICE-Instruktionen sind *bedingte Instruktionen*, d.h. ihre Ausführung kann vom Zustand eines Bits des Statusregisters abhängig gemacht werden.

Das Condition-Feld hat folgende Struktur:

Mit Hilfe der drei Auswahlbits wird eines der acht obersten Flags des Statusregisters R14 als Steuerbit für die betreffende Instruktion ausgewählt.

Ist das Negiere-Bit gesetzt, wird vor dem Auswerten des selektierten Statusbits sein Inhalt invertiert. Eine Instruktion wird nur ausgeführt, wenn die solchermaßen spezifizierte Bedingung erfüllt ist, d.h. den Wert 1 zurückliefert.

Die Opcodes der ALU

Die Tabelle auf der folgenden Folie enthält alle 16 Instruktionen der NICE-ALU zusammen mit ihren *Mnemonics* und einer kurzen Beschreibung ihrer Wirkung. Hierzu ist noch zu bemerken:

- Alle ALU-Opcodes beginnen mit einem 0-Bit.
- Ist das C-Bit gesetzt, verwendet die Instruktion den Inhalt des Übertragsbits C des Statusregisters R14.
- Ist das M-Bit gesetzt, verändert die Instruktion den Inhalt der Flags des Statusregisters. Beides ist der Normalfall und wird in einem *Assemblerprogramm* nicht extra notiert.

Die Opcodes der ALU

Opcode	Mnemonic	Beschreibung
0CM0000	MOVE	$d = s_0$
0CM0001	SUB	$d = s_0 - s_1$
0CM0010	MDB1	$d = s_0 + (s_0 \wedge s_1)$
0CM0011	ADD	$d = s_0 + s_1$
0CM0100	DBL	$d = 2s_0$
0CM0101	DEC	$d = s_0 - 1$
0CM0110	NOT	$d = \neg s_0$
0CM0111	NOR	$d = \neg(s_0 \vee s_1)$
0CM1000	IAND	$d = \neg s_0 \wedge s_1$
0CM1001	NAND	$d = \neg(s_0 \wedge s_1)$
0CM1010	XOR	$d = s_0 \oplus s_1$
0CM1011	IOR	$d = \neg s_0 \vee s_1$
0CM1100	XNOR	$d = \neg(s_0 \oplus s_1)$
0CM1101	AND	$d = s_0 \wedge s_1$
0CM1110	ONE	$d = 1$
0CM1111	OR	$d = s_0 \vee s_1$

Operandenspezifikation

Eine NICE-Instruktion kann bis zu drei Operanden verarbeiten, von denen zwei Operanden als Datenquellen und ein Operand als Ziel verwendet werden. Jedes der drei Operandenfelder besitzt den im folgenden dargestellten Aufbau:

Modus (3 Bit)	Registernummer (4 Bit)

Mit Hilfe der vier Bit langen Registernummer kann eines der 16 verfügbaren Register als Operand ausgewählt werden, wobei durch die drei Modusbits ein zusätzlicher Freiheitsgrad zur Verfügung steht, Operanden zu modifizieren, etc.

Operandenspezifikation

Mit Hilfe der drei Modusbits können folgende *Adressierungsarten* ausgewählt werden:

Modus	Destination	Source
000	Rxx	Rxx
001	Rxx–	–Rxx
010	Rxx++	Rxx++
011	@#const[Rxx]	#const[Rxx]
100	@Rxx	@Rxx
101	@–Rxx	@–Rxx
110	@Rxx++	@Rxx++
111	@#const[Rxx]	@#const[Rxx]

Operandenspezifikation, Beispiele

Rxx: Register xx wird als Quelle beziehungsweise Ziel einer Operation verwendet. MOVE R1, R0 kopiert den Inhalt von R0 (der stets 0 ist) in Register R1, das somit gelöscht wird.

–Rxx: Vor der Verwendung eines Registers als Datenquelle wird sein Inhalt um 1 verringert (sogenanntes *Postdecrement*).

#const[Rxx]: Als Quelloperand entspricht dies der Summe aus dem Wert #const sowie dem Inhalt des Registers Rxx.

@Rxx: Der Inhalt des Registers Rxx wird als Adresse der Speicherzelle verwendet, deren Inhalt als Quelle beziehungsweise die als Ziel einer Instruktion verwendet wird.

Bedingte Ausführung

Soll beispielsweise der Wert #LOOP in den Programmzähler R15 geschrieben werden (d.h. es wird ein *Sprung* vollzogen), wenn die vorige Instruktion das Z-Bit des Statusregisters R14 nicht gesetzt hatte, so wird die Instruktion wie folgt notiert:

?!Z MOVE R15, #LOOP

Bitweise betrachtet sieht die resultierende Instruktion wie folgt aus, wenn #LOOP gleich 0x12345678 ist:

1011	0000000	0001111	0110000	0000000	00010010001101000101011001111000

Zu beachten ist, daß der Wert #LOOP in einem zweiten 32 Bit-Wort abgelegt wird, die Instruktion also insgesamt 64 Bit umfaßt.

Unbedingte Ausführung

- Die Mehrzahl aller Instruktionen soll allerdings in jedem Fall, d.h. nicht beeinflußt durch den Wert von Statusbits ausgeführt werden.
- Hierzu kann das 1-Bit des Statusregisters R14 eingesetzt werden. Da dieses stets gesetzt ist, kann eine unbedingt auszuführende Instruktion also als ?1 Instruktion... notiert werden. Wird die Bedingung ?1 weggelassen, gilt sie als Defaultwert.
- Die Instruktion MOVE R15, #LOOP entspricht also der ausführlichen Variante ?1 MOVE R15, #LOOP.

Berechne $\sum_{i=1}^{16} i$

Das folgende kleine Assemblerprogramm berechnet die Summe aller ganzen Zahlen zwischen 0 und 16 (d.h. 10_{16}). Der Ablauf hierbei ist wie folgt:

- Zunächst werden einige Register vorbelegt: Das Statusregister R14 wird gelöscht, ebenso das zur Summation verwendete Register R1. Weiterhin wird das Register R2 mit 0x10 belegt.
- Im Anschluß hieran wird in einer Schleife jeweils der Wert von R2 zu dem Inhalt von R1 addiert. Nach der Addition wird der Inhalt von R2 um eins vermindert.
- Ist das Resultat dieser Decrement-Operation ungleich Null, d.h. ist das Z-Bit des Statusregisters nicht gesetzt, wird ein bedingter Sprung zur Additionsinstruktion durchgeführt.

Berechne $\sum_{i=1}^{16} i$

```
I          .EQU 0x10          ; Konstante definieren
           MOVE R14, R0       ; Statusregister loeschen
           MOVE R1, R0        ; Summenregister loeschen
           MOVE R2, #I        ; Laufvariable setzen
LOOP:      ADD  R1, R1, R2    ; R1 = R1 + R2
           DEC  R2            ; R2 = R2 - 1
       ?!Z MOVE R15, #LOOP    ; Falls R2 <> 0 ist
           HALT               ; Prozessor anhalten
```


Maschinen- und Assemblersprachen

- Als *Maschinensprache* werden die direkt von einem Prozessor ausführbaren Bitfolgen bezeichnet. In der Frühzeit der elektronischen Datenverarbeitung mußten Programme, wie das eben gezeigte, wirklich bitweise über Schalter, Lochkarten, Lochstreifen, etc. in den Speicher des Rechners eingelesen werden, bevor sie ausgeführt werden konnten. Als Programmierer mußte man also alle Instruktionen, Adressierungsarten, etc. bitweise kennen, um entsprechende Bitfolgen in Karten o.ä. stanzen zu können.
- Um diesen Prozeß etwas zu vereinfachen, wurden sogenannte *Assembler* entwickelt – dies waren mehr oder minder komplizierte Übersetzungsprogramme, die aus einem symbolisch notierten Programm, wie dem obigen Summationsprogramm, eine Bitfolge generierten, mit welcher der Rechner gesteuert werden kann.

Maschinen- und Assemblersprachen

- Die Instruktionskürzel, die in einem in Assembler notierten Programm verwendet werden, heissen Mnemonics.
- Im Grunde besitzt jede Maschine ihre eigene Maschinen- und damit auch ihre eigene Assemblersprache. Programme, die in Assembler geschrieben wurden, sind also nur auf einer Maschine des Typs (und eventuell auch der Ausstattung) lauffähig, welcher dem des Rechners, auf dem das Programm entwickelt wurde, entspricht.
- Vorteil einer Programmierung in Assembler ist, daß man eine Maschine direkt ansprechen kann und sie somit als guter Programmierer perfekt ausnutzen kann. Eine solche Programmierung ist allerdings sehr aufwendig, benötigt ein großes Wissen über die jeweilige Maschine und in der Regel sind Maschinen-/Assemblerprogramme eher schwer zu *debuggen* beziehungsweise zu erweitern.

Klassifizierung von Programmiersprachen

Eine erste Einteilung von Programmiersprachen läßt sich wie folgt vornehmen:

Maschinensprachen: Sie ermöglichen es, einen bestimmten Rechnertyp direkt anzusprechen – hierdurch sind extrem effiziente Programme möglich. Die Programmierung in Maschinensprachen erfordert aber eine sehr gute Kenntnis des zugrundeliegenden Prozessors sowie ein hohes Maß an Erfahrung. Programme in Maschinensprache sind nicht zwischen Prozessoren unterschiedlichen Typs austauschbar.

Höhere Programmiersprachen: Sie orientieren sich nicht oder nur wenig an der Struktur des zugrundeliegenden Rechners, sondern mehr an der Art der mit ihrer Hilfe zu lösenden Probleme und werden entsprechend auch als *problemorientierte Sprachen* bezeichnet.

Programmierparadigmen

Höhere Programmiersprachen lassen sich ihrerseits verschiedenen *Programmierparadigmen* zuordnen:

Imperative Sprachen: Folgen von Befehlen (C, FORTRAN, PASCAL, etc.)

Objektorientierte Sprachen: Programme basieren auf *Objekten*, die miteinander über Nachrichten kommunizieren und auf denen *Methoden* operieren. (SMALLTALK, C++, etc.)

Funktionale Sprachen: Programme beruhen (ausschließlich) auf der Auswertung mathematischer Funktionen (APL, LISP, ML, Haskell, etc.)

Prädikative Sprachen: Programme bestehen aus Fakten und Regeln (PROLOG)

Regelbasierte Sprachen: Programme bestehen aus Wenn-Dann-Regeln (OPS-5)

Berechne $\sum_{i=1}^{16} i$ in VAX-FORTRAN

```
      PROGRAM SUM
C
      IMPLICIT NONE
      INTEGER*4 SUMME /0/, I, ENDE /16/
C
      DO I = 1, ENDE
        SUMME = SUMME + I
      END DO
C
      WRITE (*, *) SUMME
      END
```

Berechne $\sum_{i=1}^{16} i$ in C

```
#include <stdio.h>

#define ENDE 16

int main ()
{
  int summe = 0, i;

  for (i = 1; i <= ENDE; summe += i++);
  printf ("%d\n", summe);
}
```


Berechne $\sum_{i=1}^{16} i$ in APL

```
⎕←+/⍳16
```

Compiler- und Interpretersprachen

- Um ein in einer Hochsprache formuliertes Programm auf einem Computer ausführen zu können, muß es in der einen oder anderen Weise in eine Folge von semantisch äquivalenten Maschineninstruktionen umgewandelt werden.
- Dies kann entweder mit Hilfe eines *Compilers* geschehen, der ein Hochsprachenprogramm analysiert und ein diesem entsprechendes Programm in der jeweiligen Zielmaschinensprache generiert oder
- mit Hilfe eines *Interpreters*, der das Hochsprachenprogramm Anweisung für Anweisung *zur Laufzeit* interpretiert, d.h. im Gegensatz zu einem Compiler kein explizites Maschinenprogramm erzeugt, sondern das Programm interpretativ abarbeitet.

Compilersprachen

- Eine Vielzahl höherer Programmiersprachen sind Compilersprachen – typische Vertreter sind C, FORTRAN, COBOL, etc.
- Der Hauptvorteil einer Compilersprache ist die in der Regel hohe Ausführungsgeschwindigkeit des erzeugten Maschinenprogrammes, da nur zum Zeitpunkt der eigentlichen Übersetzung Maschinenzeit für den Umsetzungsvorgang aufgewandt werden muß.
- Hierdurch geht jedoch ein gewisses Maß an Interaktivität verloren – eine Anweisung kann nicht direkt ausgeführt werden, sondern muß zunächst übersetzt werden, was nicht zuletzt beim Erlernen einer höheren Programmiersprache (vor allem zu Zeiten verhältnismäßig langsamer Rechner) hinderlich ist.

Interpretersprachen

- Da Interpretersprachen erst zur Laufzeit übersetzt werden, sind sie in der Regel langsamer als vergleichbare Compilersprachen.
- Dieser Nachteil wird jedoch in vielen Fällen durch die wesentlich höhere Interaktivität solcher Sprachen wieder wettgemacht.
- Darüberhinaus sind einige Sprachen prinzipiell nur als Interpretersprachen zu implementieren, da sie beispielsweise Datenstrukturen bereitstellen, die ihrerseits Programme darstellen, jedoch erst während des Ablaufes des Programmes erstellt werden und somit bei einer Compilierung nicht berücksichtigt werden könnten.
- Typische heutige Interpretersprachen sind beispielsweise: Perl, Python, LISP, etc.

Mischformen

- Nicht alle Sprachen lassen sich eindeutig einer der beiden Kategorien *Compilersprachen* oder *Interpretersprachen* zuordnen.
- Ein Beispiel hierfür ist die Sprache JAVA. Im Prinzip handelt es sich hierbei um eine Compilersprache, da ein in JAVA formuliertes Programm zunächst mit Hilfe eines Compilers in einen Maschinencode umgesetzt wird.
- Dieser Maschinencode, in diesem Zusammenhang auch *Bytecode* genannt, entspricht jedoch keiner real existierenden Maschine, sondern einem hypothetischen Prozessor.
- Dies wiederum erfordert, daß der Bytecode mit Hilfe eines Interpreters ausgeführt werden muß.