RSB Zusammenfassung

Inhaltsverzeichnis

Einführung

Von-Neumann Architektur

Ein Rechnerarchitekturkonzept, welches 1945 von J. Mauchly, J. P. Eckert und J. von-Neumann vorgestellt wurde. Das Modell beschreibt einen Rechner, der aus mehreren Komponenten besteht und mit Hilfe dieser Befehle und Programme abarbeiten kann.

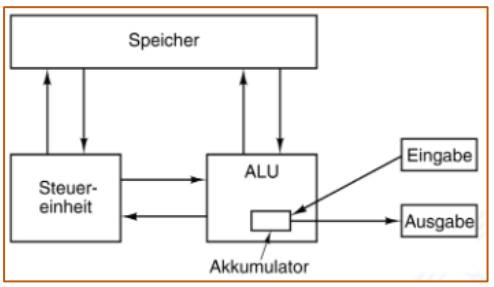
5 Hauptkomponenten:

Rechenwerk:

- Ausführung von Rechenoperationen
- Ausführung von Logikoperationen

Steuerwerk:

- Koordiniert Systemkomponenten basierend auf Befehl
- Regelt Programmablauf



Darstellung eines Von-Neumann Rechners

Speicherwerk:

- Speichert Daten und Befehlen gleichartig
  - Daten können als Befehle interpretiert werden und umgekehrt
- Fortläufig Adressiert

Ein-/Ausgabewerk:

- Steuert Dateneingabe/-ausgabe zwischen Komponenten und Peripherie
  - Peripherie: Extern angeschlossene Geräte (Monitor/Tastatur/…)

Bussystem:

- Ermöglicht Datenübertragung zwischen Komponenten

Abarbeitung eines Befehls:

1. **(FETCH)** Befehl von aktueller Speicheradresse holen
2. **(DECODE)** Befehl dekodieren
3. **(EXECUTE)** Befehl ausführen, Speicheradresse erhöhen/ändern

Moores Law

Regel die besagt, dass sich die **Anzahl von Transistoren** von Chips alle 3 Jahre vervierfacht.

Formel: $L(t) = L(0) * 2^{t/18}$ => Leistung an Zeitpunkt t (in Jahren)

$L(0)$ => Leistung an Zeitpunkt 0

Informationsverarbeitung

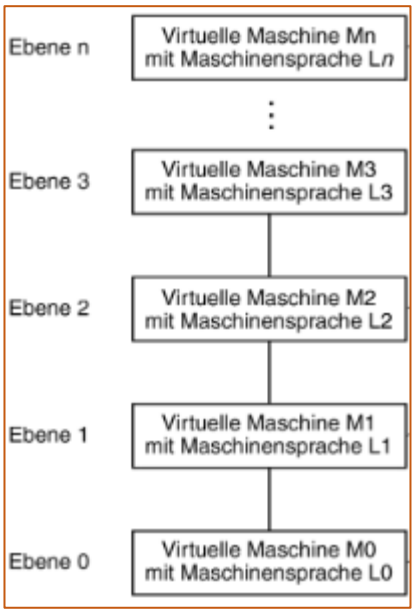
Semantic Gap

Computer kennt (auf der untersten Ebene) nur **Maschinensprache**. Menschen wollen gerne aber abstraktere Befehle als nur Maschinenbefehle geben (bsp. Online Shopping).

Semantic Gap: Überbrücken der Lücke zwischen Mensch und Maschine

Aus diesem Grund werden **Abstraktionsebenen** (Mn) geschaffen, die aufeinander Aufbauen und immer mächtigere Befehle/Sprachen (Ln) bieten, indem sie einen/mehrere Befehle der Ebenen unter sich ausführen. Diese Ebenen werden auch **Virtuelle Maschinen** genannt.

Der User greift (i.d.R.) auf die **oberste Schicht** zu, die **unterste Schicht** führt direkt Befehle auf der Maschinenebene aus.



Compiler

Erzeugen eines neuen Programms, in dem jeder L1 Befehl durch eine zugehörige Folge von L0 Befehlen ersetzt wird.

Interpreter

Direkte Ausführung der L0 Befehlsfolgen zu jedem L1 Befehl.

Information

Abstrakter Gehalt einer Aussage.

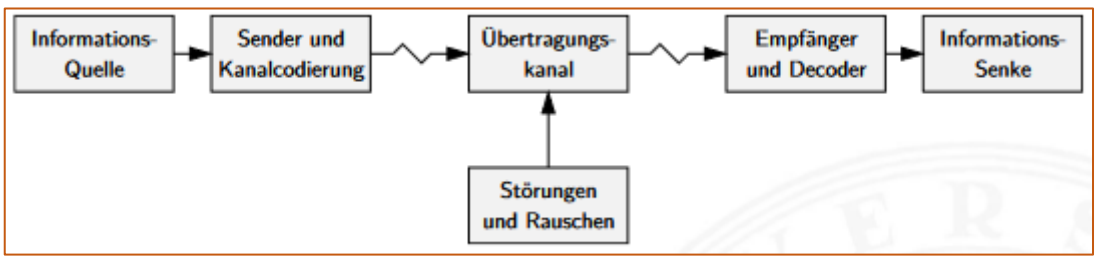
Information hat immer eine **Repräsentation**: die Darstellungsart, in der die Information codiert ist.

Bsp. Repräsentation:

Text (Deutsch)	Zehn
Text (Englisch)	Ten
Zahl (Dezimal)	10
Zahl (Binär)	1010

Informationsübertragung

Nachricht: Zeichen oder Funktionen, die Informationen zum Zweck der Weitergabe aufgrund bekannter oder unterstellter Abmachungen darstellen.



Nachrichtentechnisches Modell der Informationsübertragung

Zahlensysteme

Binär

Zahlensystem, in denen Zahlen mit nur 2 Zeichen repräsentiert werden, i.d.R. **0** und **1**.

Dezimal	Binär	Dezimal	Binär
0	0	**16** (= 2^4)	1 0000
1	1	**32** (= 2^5)	10 0000
2 (= 2^1)	10	**64** (= 2^6)	100 0000
3	11	**128** (= 2^7)	1000 0000
4 (= 2^2)	100	**256** (= 2^8)	1 0000 0000
8 (= 2^3)	1000	**512** (= 2^9)	10 0000 0000

Hexadezimal

Zahlensystem, in denen Zahlen mit 16 Zeichen repräsentiert werden, i.d.R. **{0, 1, …, 9, A, B, C, D, E, F}**.
4 Stellen einer Binärzahl lassen sich direkt in 1 Stelle einer Hexadezimalzahl umwandeln.

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

Bsp. 108 = 64 + 32 + 8 + 4 (Decimal)
= $0110\ 1100_2$ (Binary)
= $6C_{16}$ (Hex)

Oktal

Zahlensystem, in denen Zahlen mit 8 Zeichen repräsentiert werden, i.d.R. **{0, 1, …, 8}**. 3 Stellen einer Binärzahl lassen sich direkt in 1 Stelle einer Oktalzahl umwandeln.

Dezimal	Binär	Oktal
0	000	0
1	001	1
2	010	2
3	011	3
4	100	4
5	101	5
6	110	6
7	111	7

Bsp. 101 100 111 000 (Binary) = 5470 (Oktal)

Umwandlung zwischen Zahlensystemen

Es gibt 3 Möglichkeiten, Zahlen zwischen Zahlensystemen umzuwandeln.

Dabei wird eine Zahl **n** aus der Basis **A** in eine Zahl der Basis **B** umgewandelt.

1. **Potenztabelle**
   - Solange n > 0:
     - Subtraktion der größten Potenz von **B** von **n**.
     - Notieren dieser Potenz
   - Aufaddieren aller notierten Potenzen

   Bsp.: 26 (Dezimal) in Binär

Subtraktion	Potenzen
26 – 16 = 10	16 -> 1 0000
10 – 8 = 2	8 -> 1000
2 – 2 = 0	2 -> 10

   Ergebnis: 10000 + 1000 + 10 = **11010**

2. **Divisionsrestverfahren**
   - Solange n > 0:
     - Division von **n** durch **B**, jeweils abrunden
     - Notieren des Rests an der zugehörigen Position
   - Spiegelverkehrtes Aneinanderreihen der Reste

   Bsp.: 26 (Dezimal) in Binär

Division	Reste
26 / 2 = 13	0
13 / 2 = 6	1
6 / 2 = 3	0
3 / 2 = 1	1
1 / 2 = 1	1

   Ergebnis: **11010**

3. **Hornerschema**
   - Darstellung der **Potenzsumme** von **n** durch ineinander verschachtelte Faktoren
   - Umwandlung der Faktoren nach **B**
   - Ausrechnen des Terms

   Bsp.: 26 (Dezimal) in Binär

   26 = 2 * 10 + 6 (dezimal)
   = 10 * 1010 + 110 (binär)
   = 10100 + 110 (binär)
   = 11010 (binär)

Darstellung negativer Zahlen

3 Varianten:

1. **Betrag und Vorzeichen**
   - Vorderstes Bit speichert Vorzeichen
   - 1 = negative Zahl, 0 = positive Zahl

2. **Exzess-Codierung**
   - Einfache Um-Interpretation der Binärcodierung
   - **z = c – offset**
   - Offset ist durch Namen der Codierung angegeben: Exzess-8 heißt **offset = 8**

3. **b-Komplement**
   - Invertieren aller Zeichen und Addieren von 1
     - 1230 => 8769 + 1 = 8770 (decimal)
     - 0111 => 1000 + 1 = 1001 (binary)

$$K_b(z) = b^n - z, \quad \text{für} \quad z \neq 0$$
$$= 0, \quad \text{für} \quad z = 0$$

   - **(b-1)-Komplement**: Invertieren aller Zeichen
   - Erlaubt Subtraktion bei fester Länge durch alleinige Anwendung von Addition
     - $A - B = A + \overline{B} + 1$

Alle 1- und 2- Komplementwerte für Binärzahlen der Länge 3

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

Gleitkommazahlen

3 Bereiche: **Vorzeichen** s (Codiert wie „Betrag und Vorzeichen“)
Mantisse m (Auf Bereich 1,0 - 2,0 normalisiert)
Exponent e (Float: Exzess-127, Double: Exzess-1023)

$$\textbf{Zahl} = (-1)^{s} * m * B^{(e - offset)} \quad (\textbf{B} = \text{Basis})$$

Die Formate Float und Double haben unterschiedlich viele Bits:

	Float (32 bits)	Double (64 bits)
Vorzeichen	1	1
Mantisse	8	11
Exponent	23	52

Bsp.: -4,75 in als Float

$-4{,}75 = -(2^2 + 2^{-1} + 2^{-2}) \Rightarrow -100{,}11$ (in binär umwandeln)

$-100{,}11 * 2^0$ (normalisieren)
$-10{,}011 * 2^1$
$-1{,}0011 * 2^2$

Vorzeichen: 1
Exponent: 2, als Exzess-127: 129 => 10000001
Mantisse: 0011 => 00110000000000000000000

Ergebnis: 1 10000001 00110000000000000000000

Sonderwerte:

	Vorzeichen	Exponent	Mantisse
0	V	00…00	00…00
Inf	V	11…11	00…00
NaN	V	11…11	Alles außer 00…00

Mathematische Eigenschaften:

	Addition	Multiplikation
Abgeschlossen	Ja	Ja
Kommutativ	Ja	Ja
Assoziativ	Nein: Überlauf, Rundung	Nein: Überlauf, Rundung
Distributiv	-	Nein
Null/Eins ist neutrales Element?	Ja	Ja
Monotonie	Fast	Fast

Zeichen und Text

Zeichen: Element z aus einer zur Darstellung von Information vereinbarten, einer Abmachung unterliegenden, endlichen Menge Z von Elementen

Zeichensatz : Menge Z von Elementen

Binärzeichen: Jedes der Zeichen aus einem Vorrat / aus einer Menge von zwei Symbolen

Alphabet: Ein in vereinbarter Reihenfolge geordneter Zeichenvorrat

Zeichenkette: Eine Folge von Zeichen

Wort: Eine Folge von Zeichen, die in einem gegebenen Zusammenhang als Einheit bezeichnet wird

Stelle: Die Lage/Position eines Zeichens innerhalb einer Zeichenkette

Ascii

- 1967 eingeführte Codierung für Textdateien
- **7**-bit pro Zeichen, **128** Zeichen insgesamt
- 95 druckbare Zeichen, 33 Steuerzeichen

Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1...	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2...	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6...	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

- SP = Leerzeichen, CR = carriage-return, LF = line-feed
- ESC = escape, DEL = delete, BEL = bell usw.

Die ASCII Codetabelle

ISO-8859 Familie

- Erweiterung von ASCII um Sonderzeichen und Umlaute
- **8**-bit pro Zeichen, **256** Zeichen darstellbar
- Verschiedene Bereiche für verschiedene Sprachen/Sprachregionen
  - Tabelle für Baltisch, Kyrillisch, Arabisch, Griechisch, …

Unicode

- System zur Codierung aller Zeichen **aller bekannten Schriftsysteme**
- **16**-bit für jedes Zeichen, bis zu **65.536** Zeichen
- ab Unicode 3.0 mehrere *Planes* mit **je** 65.356 Zeichen
- Codierung als UTF-8 oder UTF-16

UTF-8 Codierung:

- Mehrere Bytes zur Darstellung möglich (1..n)
- Anzahl führende **1** gibt an, wie viele Bytegruppen zum Zeichen gehören, gefolgt von **0**
- Alle folgenden Bytegruppen beginnen mit **10**

Binär-Darstellung	Anzahl Wörter
`0*** ****`	128
`110* **** 10** ****`	1920
`1110 **** 10** **** 10** ****`	63.488
`1111 0*** 10** **** 10** **** 10** ****`	bis 2^{21}

Logische Operationen

	Schreibweise	X = 0 Y = 0	X = 0 Y = 1	X = 1 Y = 0	X = 1 Y = 1
NOT X	**¬ X**	**1**	**1**	0	0
X AND Y	**X ∧ Y**	0	0	0	**1**
X OR Y	**X ∨ Y**	0	**1**	**1**	**1**
X XOR Y	**X ⊕ Y**	0	**1**	**1**	0
X NAND Y	**X NAND Y**	**1**	**1**	**1**	0
X NOR Y	**X NOR Y**	**1**	0	0	0

DeMorgansche Regeln

Ersetzen von AND durch OR und umgekehrt

$$\neg(a \vee b) = \neg a \wedge \neg b$$

$$\neg(a \wedge b) = \neg a \vee \neg b$$

Bitweise Operationen

Die bitweisen Operationen führen logische Operationen auf **allen** Bits einer Zahl aus.

- Negation **~**
- AND **&**
- OR **|**
- XOR **^**

Bsp.: ~123 (dec)
=> ~1111011 (bin)
=> 0000100

Bsp.: 55 & 47 (dec)
=> 110111 & 101111 (bin)
=> 100111

Schiebeoperationen

5 Varianten:

- Shift-Left **shl** (0 nachschieben)
- Logical Shift-Right **srl** (0 nachschieben)
- Arithmetic Shift-Right **sra** (Vorzeichenbit nachschieben)
- Rotate-Left **rol**
- Rotate-Right **ror**

Arithmetic Shift entspricht Division/Multiplikation mit $\mathbf{2^n}$, kann genutzt werden um bestimmte Multiplikationen/Divisionen Hardwareeffizient zu implementieren.

Codierung

Begriffe

Codewörter: die Wörter der Repräsentation B aus einem Zeichenvorrat Z
Code: die Menge aller Codewörter
Blockcode: alle Codewörter haben dieselbe Länge
Binärzeichen: der Zeichenvorrat z enthält genau zwei Zeichen
Binärwörter: Codewörter aus Binärzeichen
Binärcode: alle Codewörter sind Binärwörter

Begriffe für Binärcodes:

Minimalcode: alle $N = 2^n$ Codewörter bei Wortlänge n werden benutzt
Redundanter Code: nicht alle möglichen Codewörter werden benutzt
Gewicht: Anzahl der Einsen in einem Codewort
komplementär: zu jedem Codewort c existiert ein gültiges Codewort $\bar{c}$
einschrittig: aufeinanderfolgende Codewörter unterscheiden sich nur an einer Stelle
zyklisch (einschr.): bei n geordneten Codewörtern ist $c_0 = c_n$
Dualcode: Name für Codierung der Integerzahlen

Einschrittige Codes

Code, bei denen sich benachbarte Codewörter nur um eine Stelle unterscheiden.

0000
0001
0011
0010
0110
0100
1100
1000

*Ein Binärcode. Der Code ist redundant, **einschrittig** und zyklisch*

Maximaler Ablesefehler:

- 2^{n-1} beim **Dualcode**
- 1 beim **einschrittigen Code**

Es existieren zwei Möglichkeiten einschrittige Codes zu generieren.

[1] Rekursive Konstruktion

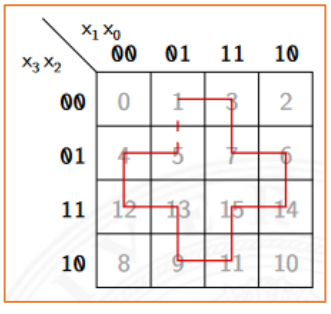
1. Starte mit zwei Codewörtern: **0** und **1**
2. Hänge eine führende **0** vor alle Codewörter
3. Hänge eine führende **1** vor alle Codewörter in umgekehrter Reihenfolge
4. Wiederhole ab 2.

Bsp.: {**0**, **1**} (neue Zeichen sind fett markiert)
{**0**0, **0**1, **1**1, **1**0}
{**0**00, **0**01, **0**11, **0**10, **1**10, **1**11, **1**01, **1**00

[2] KV-Diagramm

1. Stelle 2D-KV-Diagramm auf (für Gewöhnlich 2x2, 2x4 oder 4x4)
2. Finde zyklischen Pfad durch KV-Diagramm (darf auch **über den Rand** gehen)
3. Lese Codewörter vom Pfad ab

Bsp.:



Wir können direkt alle Codewörter ablesen

0001	**1110**	**1101**
0011	**1111**	**1100**
0111	**1011**	**0100**
0110	**1001**	**0101**

Fano Bedingung

Kein Wort aus einem Code bildet den **Anfang** eines anderen Codeworts.
-> Auch Präfix Eigenschaft genannt

- **Präfix-Codes** sind eindeutig decodierbar
- Jeder **Blockcode** ist ein Präfix-Code

Shannon-Fano Codierung

- Art, Elemente effizient basierend auf ihrer Auftrittswahrscheinlichkeit zu codieren
- Code erfüllt die **Fano Bedingung**

Schritte	Beispiel
Gegeben sind die Urwörter $\mathbf{a_i}$ und deren Auftrittswahrscheinlichkeiten $\mathbf{p(a_i)}$.	a_i = {A, B, C, D, E} $p(a_i)$ = {0,2; 0,15; 0,25; 0,1; 0,3}
Ordnung der Urwörter anhand ihrer Wahrscheinlichkeiten.	a_i = {E, C, A, B, D} $p(a_i)$ = {0,3; 0,25; 0,2; 0,15; 0,1}
Einteilung der geordneten Wörter in Zwei Gruppen mit möglichst gleichen Gesamtwahrscheinlichkeiten	$p(a_1, a_2) = 0,55$ $p(a_3, a_4, a_5) = 0,45$
Die erste Gruppe erhält als erste Stelle eine 0, die andere eine 1.	$p(a_1, a_2) = 0,55$ -> 0 $p(a_3, a_4, a_5) = 0,45$ -> 1
Dasselbe Verfahren auf die Teilgruppen anwenden.	…
Stopp, wenn jede Teilgruppe nur noch ein Urwort enthält.	A => 10 B => 110 C => 01 D => 111 E => 00

Huffman Codierung

- Ähnlich zur Shannon-Fano Codierung, nur kleine Abweichungen
- ergibt kleinstmögliche mittlere Codewortlängen

Schritte	Beispiel
Gegeben sind die Urwörter $\mathbf{a_i}$ und deren Auftrittswahrscheinlichkeiten $\mathbf{p(a_i)}$.	a_i = {A, B, C, D, E} $p(a_i)$ = {0,2; 0,15; 0,25; 0,1; 0,3}
Ordnung der Urwörter anhand ihrer Wahrscheinlichkeiten.	a_i = {E, C, A, B, D} $p(a_i)$ = {0,3; 0,25; 0,2; 0,15; 0,1}
Fasse die beiden Wörter mit der niedrigsten Wahrscheinlichkeit zusammen und ersetze sie durch ein neues Wort	B, D => F p(F) = p(B) + p(D) = 0,25
Wiederhole den Schritt, bis nur noch zwei Wörter bleiben	A, F => G p(G) = 0,45 E, C => H p(H) = 0,55
Rekursiv als Wort codieren (Bsp. links = 0, rechts = 1)	H => 0 **E => 00** **C => 01** G => 1 **A => 10** F => 11 **B => 110** **D => 111**

Informationsbegriff

- **n** mögliche, sich gegenseitig ausschließende Ereignisse $\mathbf{A_i}$ die zufällig nacheinander mit Wahrscheinlichkeiten $\mathbf{p_i}$ eintreten
- Informationsgehalt eines Ereignisses: $\mathrm{I}(A_i) = \log_2(\frac{1}{p_1})$
- Wert von **I** ist eine reelle Größe, wird in **Bit** gemessen

Entropie

- gibt **durchschnittliche Information** bei Empfang eines Symbols an
- = der **Erwartungswert des Informationsgehalts**
- Entropie ist am **höchsten**, wenn alle Wahrscheinlichkeiten **gleich groß** sind

$$H = -\sum_i p_i * \log_2(p_i)$$

Bsp.: A_i = {A, B, C}
p_i = {0,5; 0,25; 0,25}

$$\begin{aligned} H &= -\left(0{,}5 * \log_2(0{,}5) + 0{,}25 * \log_2(0{,}25) + 0{,}25 * \log_2(0{,}25)\right) \\ &= -\left(0{,}5 * (-1) + 0{,}25 * (-2) + 0{,}25 * (-2)\right) \\ &= -(-0{,}5 - 0{,}5 - 0{,}5) \\ &= 1{,}5 \end{aligned}$$

Möglicher Informationsgehalt

- wird genutzt, um **größtmöglichen Informationsgehalt** eines Codes zu berechnen
- = **mittlere Codewortlänge**

$$H_0 = \sum_i p_i * \log_2(q^{l_i})$$

q = Basis
l_i = Länge des Wortes

Für Binärcodes gilt:

$$H_0 = \sum_i p_i * l_i$$

Für binäre Blockcodes gilt:

$$H_0 = N$$

Redundanz

- Differenz zwischen dem **möglichen** und **tatsächlich genutzten** Informationsgehalt

$$R = H_0 - H$$

Relative Redundanz: $r = \frac{H_0 - H}{H_0}$

Kanalkapazität

- Übertragungskanäle sind nicht perfekt
- maximal pro Binärstelle übertragbare Informationsgehalt

$$C = 1 - H(F)$$

- Bei binären, symmetrischen (=Wahrscheinlichkeit 0, 1 gleich groß) Kanälen ist die Wahrscheinlichkeit, dass eine 0 zu einer 1 geflippt wird genau so groß, wie die Wahrscheinlichkeit, dass eine 1 zu einer 0 geflippt wird
- => Wahrscheinlichkeit von Übertragungsfehler **P**

$$H(F) = P * \log_2\left(\frac{1}{p}\right) + (1 - P) * \log_2(\frac{1}{1 - P})$$

Fehlererkennende Codes

Begriffe:

Block-Code: k-Informationsbits werden in n-Bits codiert
Faltungscodes: ein Bitstrom wird in einen Codebitstrom höherer Bitrate codiert
linearer (n; k)-Code: ein k-dimensionaler Unterraum des GF(2)^n

Hamming-Abstand: die Anzahl der Stellen, an denen sich zwei Binärcodewörter der Länge w unterscheiden
Hamming-Gewicht: Hamming-Abstand eines Codeworts vom Null-Wort (= Anzahl Einsen)

Bsp.: a = 1100, b = 0111
Hamming-Abstand a zu b = 3

- Zur Fehlererkennung und Fehlerkorrektur ist eine Codierung mit **Redundanz** erforderlich
- Repräsentation enthält extra Bits, die keine neuen Informationen enthalten
- Codewörter so wählen, dass sie **alle** paarweise den Hamming-Abstand **d** haben
  - **Fehlererkennung** bis zu $(d - 1)$ Stellen
  - **Fehlerkorrektur** bis zu $\frac{(d-1)}{2}$ Stellen

Hamming-Codes

- Jeweils geschrieben **(N, n)-Hamming Code**
  - n = Anzahl Datenbits
  - (k = Anzahl Prüfbits)
  - N = n + k (Codewortlänge)

Bsp.: (7, 4)-Hamming Code

  - Codewörter sind 7 bits lang
  - haben 4 Datenbits, 3 Prüfbits

Konstruktion:

1. Bestimme kleinstes **k** mit $\mathbf{n \leq 2^k - k - 1}$
2. Plaziere Prüfbits an den Positionen 2^0, 2^1, …, 2^{k-1}, alle anderen Positionen sind Datenbits
3. Berechne Werte der Prüfbits
   - Prüfbit **i** ist XOR der Bits, deren **Positionsnummer** ein gesetztes Bit an Position **i** haben
   - Bsp.: Prüfbit p_2
     -> XOR aller Bits, die an Position stehen die eine 1 an Bitstelle 2 hat.

     Positionen: 0**0**0**0**, 0**0**0**1**, 0**0**1**0**, 0**0**1**1**, 0**1**0**0**, 0**1**0**1**, 0**1**1**0**, 0**1**1**1**, …
     Werte: 0, 0, 0, 1, 1, 1, 1, 1, …
     p_2 = 0 XOR 1 XOR 1 XOR 1 = **1**

Schaltfunktionen

Begriffe

Schaltfunktion: eine eindeutige Funktion **f**, die jeder Wertekombination $(b_1, b_2, \dots, b_n)$ von Schaltvariablen einen (i.d.R. binären) Wert zuweist:

$$y = f(b_1, b_2, \dots, b_n) \in \{0, 1\}$$

Schaltvariable: eine Variable, die nur endlich viele Werte annehmen kann – typisch sind binäre Schaltvariablen

Ausgangsvariable: die Schaltvariable am Ausgang der Funktion, die den Wert y annimmt

Normalformen

Disjunktive Normalform (DNF)

ODER Verknüpfung aller **Minterme** (**UND**-Verknüpfung aller Schaltvariablen der Funktion, wobei die Variablen entweder negiert oder nicht negiert auftreten dürfen) mit dem **Funktionswert 1**

Bsp.:

x	y	z	f (x, y, z)
0	0	0	0
0	0	1	0
0	**1**	**0**	**1**
0	1	1	0
1	**0**	**0**	**1**
1	**0**	**1**	**1**
1	1	0	0
1	**1**	**1**	**1**

=> $f(x, y, z) = (\overline{x} \wedge y \wedge \overline{z}) \vee (x \wedge \overline{y} \wedge \overline{z}) \vee (x \wedge \overline{y} \wedge z) \vee (x \wedge y \wedge z)$

Konjunktive Normalform (DNF)

UND Verknüpfung aller **Maxterme** (**ODER**-Verknüpfung aller Schaltvariablen der Funktion, wobei die Variablen entweder negiert oder nicht negiert auftreten dürfen) mit dem **Funktionswert 0.**

Achtung! Variablen in den Maxtermen werden nochmal negiert!

Bsp.:

x	y	z	f (x, y, z)
0	**0**	**0**	**0**
0	**0**	**1**	**0**
0	1	0	1
0	**1**	**1**	**0**
1	0	0	1
1	0	1	1
1	**1**	**0**	**0**
1	1	1	1

=> $f(x,y,z) = (x \vee y \vee z) \wedge (x \vee y \vee \overline{z}) \wedge (x \vee \overline{y} \vee \overline{z}) \wedge (\overline{x} \vee \overline{y} \vee z)$

Allgemein disjunktive/konjunktive Form

Wie DNF/KNF, aber es müssen nicht immer alle Schaltvariablen in den Min-/Maxtermen vorhanden sein

Bsp.: $f(x,y,z) = (\overline{x} \wedge y \wedge \overline{z}) \vee (x \wedge \overline{y} \wedge \overline{z}) \vee (x \wedge \overline{y} \wedge z) \vee (x \wedge y \wedge z)$

$= (\overline{x} \wedge y \wedge \overline{z}) \vee (x \wedge \overline{y} \wedge \overline{z}) \vee (x \wedge z)$

Reed-Muller Form

XOR-Verknüpfung von **UND-Termen** (die 1 ist auch als ganzer Term erlaubt).

Substitutionsregeln:

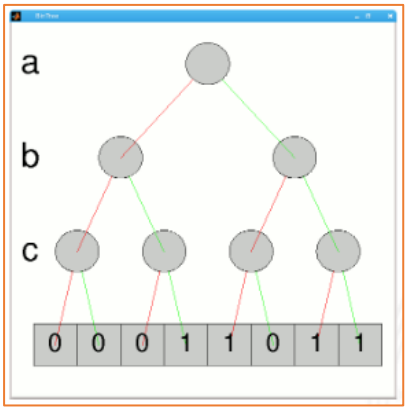
$\overline{a} = a \oplus 1$
$a \vee b = a \oplus b \oplus ab$
$a \oplus a = 0$

Bsp.:
$$
\begin{aligned}
f(x,y,z) &= (\overline{x} \vee y)\, z \\
&= (\overline{x} \oplus y \oplus \overline{x}y)\, z \\
&= ((\mathrm{x} \oplus 1) \oplus y \oplus (\mathrm{x} \oplus 1)\, y)\, z \\
&= (\mathrm{x} \oplus 1 \oplus y \oplus \mathrm{xy} \oplus y)\, z \\
&= (\mathrm{x} \oplus 1 \oplus \mathrm{xy})\, z \\
&= \mathrm{xz} \oplus z \oplus \mathrm{xyz}
\end{aligned}
$$

Grafische Darstellung

Entscheidungsbaum

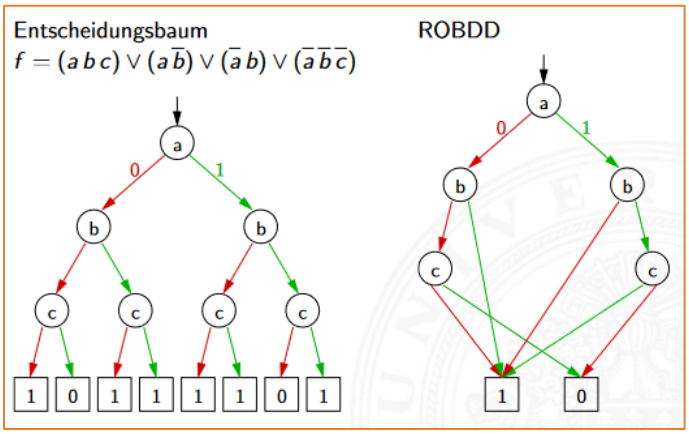
- Darstellung einer Schaltfunktion als Baum
- Jeder Knoten = Variable, jede Verzweigung = if/else Entscheidung



Bsp.: f (a, b, c) = (a ∧ c) ∨ (b ∧ c)

ROBDD

- **= Reduced Ordered Binary-Decision Diagrams**
- Entscheidungsbaum, bei denen Verzweigungen, die zum selben Ergebnis führen, vereinfacht werden



KV-Diagramme

- grafisches Verfahren, um Schaltfunktionen zu **minimieren**

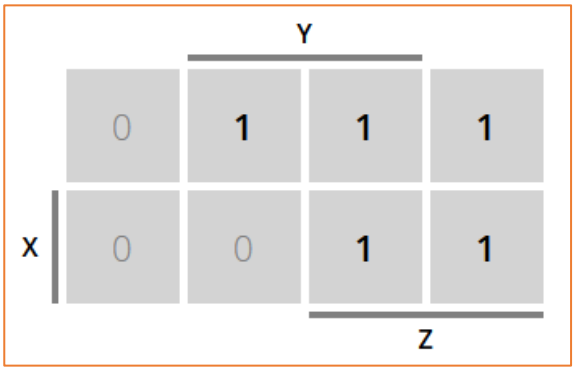
Ablauf:

1. KV-Diagramm für Anzahl Variablen aufstellen
2. KV-Diagramm mit Funktionswerten füllen
3. Zusammenfassen benachbarter **0**/**1**-Terme (Dont-Care (*) in beiden Fällen erlaubt) durch Schleifen
   - möglichst große Schleifen, möglichst wenig Schleifen
   - 1 führt zu DNF
   - 0 führt zu KNF
4. Ablesen von Funktionstermen aus den Schleifen, 1 Schleife = 1 Term

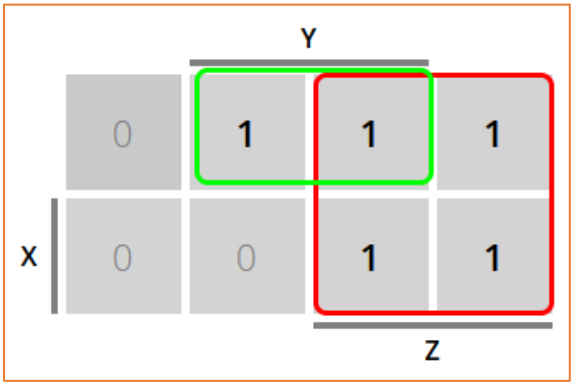
Bsp.: gegeben: Funktionswerte

x	y	z	f (x, y, z)
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

KV-Diagramm aufstellen und füllen

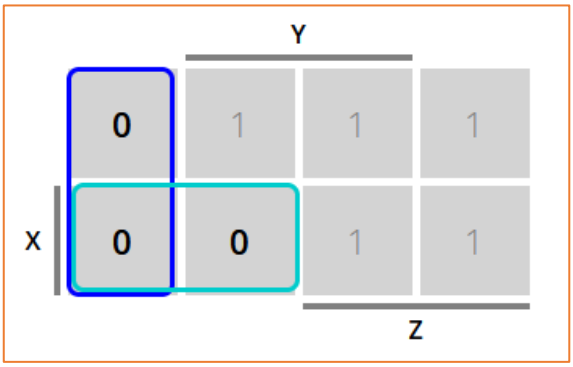


Schleifen ziehen (DNF)



Funktion aufstellen $f(x,y,z) = z \vee (\overline{x} \wedge y)$

Schleifen ziehen (KNF)



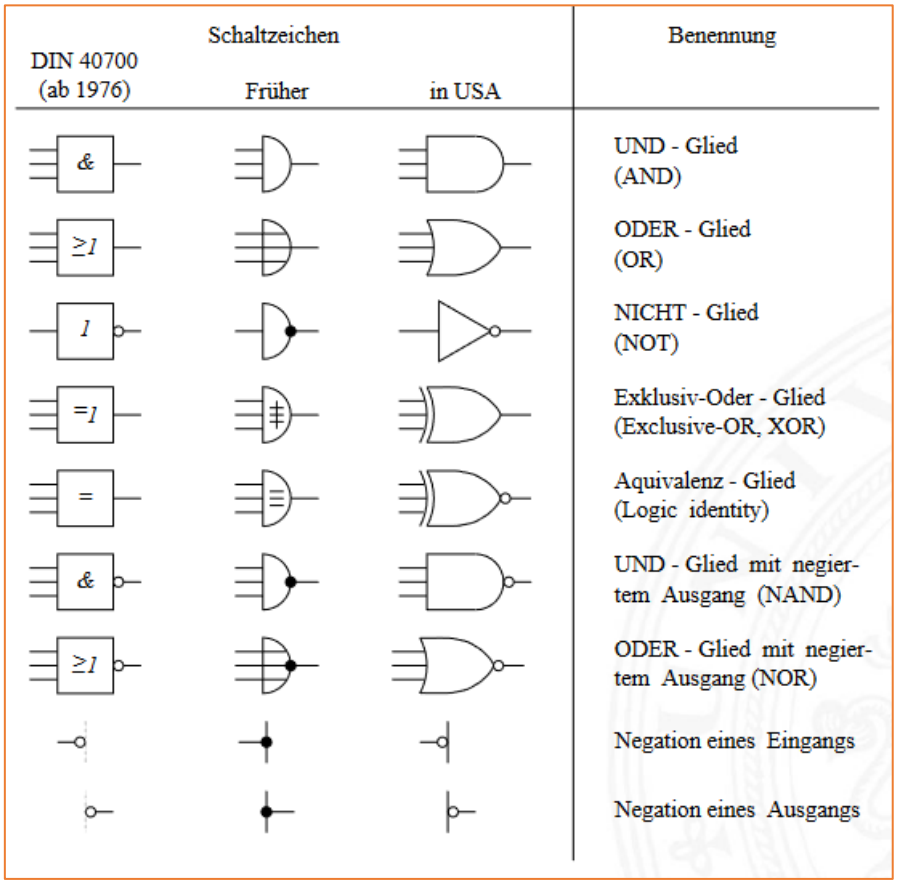
Funktion aufstellen $f(x,y,z) = (\overline{x} \vee z) \wedge (y \vee z)$

Schaltnetze

Schaltnetz ein digitales System mit **n** Eingängen (b_1, b_2, … ,b_n) und **m** Ausgängen (y_1, y_2, …, y_m), dessen Ausgangsvariablen zu jedem Zeitpunkt nur von den aktuellen Werten der Eingangsvariablen abhängen

=> Bündel von Schaltfunktionen

Schaltsymbole



Multiplexer

- Umschalter zwischen mehreren Dateneingängen
  - **$\log_2(n)$** Steuereingänge $s_1, \dots, s_{\log_2 n}$
  - **n** Dateneingänge $a_1, \dots, a_n$
  - **ein** Datenausgang y
- **Demultiplexer**: Umschalter zwischen mehreren Datenausgängen (Gegenteil von Multiplexer)

Bsp.: ein Multiplexer zwischen 2 Dateneingängen
=> n = 2
=> 1 Steuereingang

s	a_1	a_0	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Schaltnetze für logische/arithmetische Operationen

Halbaddierer

- berechnet **1-bit Summe s** und Übertrag c_o (carry-out) von zwei Eingangsbits **a** und **b**
  - $c_o = a \wedge b$
  - $s = a \oplus b$

a	b	c_o	s
0	0	**0**	**0**
0	1	**0**	**1**
1	0	**0**	**1**
1	1	**1**	**0**

Volladdierer

- berechnet **1-bit Summe s** und Übertrag c_o (carry-out) von zwei Eingangsbits **a** und **b** sowie Eingangsübertrag c_i (carry-in)
  - $c_o = ab \vee ac_i \vee bc_i = (ab) \vee (a \vee b)ci$
  - $s = a \oplus b \oplus c_i$

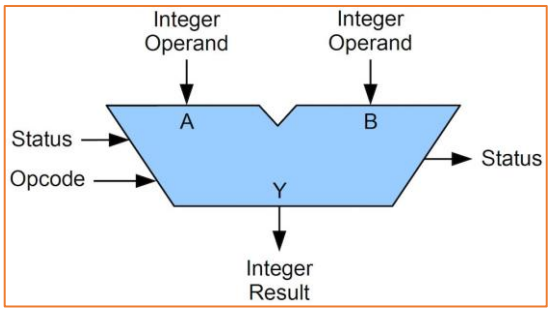
a	b	c_i	c_o	s
0	0	0	**0**	**0**
0	0	1	**0**	**1**
0	1	0	**0**	**1**
0	1	1	**1**	**0**
1	0	0	**0**	**1**
1	0	1	**1**	**0**
1	1	0	**1**	**0**
1	1	1	**1**	**1**

- zum Berechnen von Summen mit mehr bits schaltet man die Volladdierer in einer Reihe und verbindet jeweils das **carry-out** des hinteren in das **carry-in** des vorderen Addierers

- **Subtrahieren**:
  - $A - B = A + \overline{B} + 1$
  - B invertieren, carry-in des ersten Addierers auf 1 setzen

ALU

- das zentrale Rechenwerk in Prozessoren
- kombiniertes Schaltnetz für arithmetische und logische Operationen
  - Addition/Subtraktion
  - bitweise logische Operationen
  - Schiebeoperationen
  - evtl. Multiplikationen

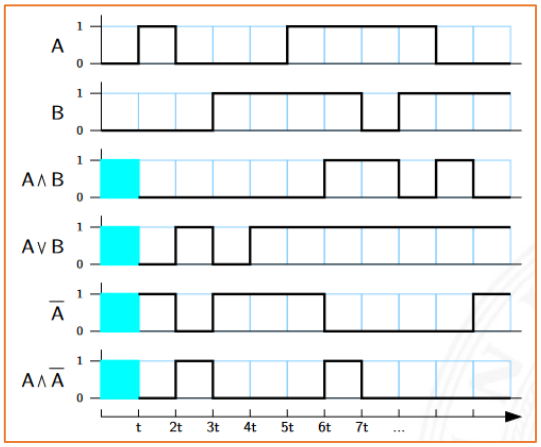


Zeitverhalten von Schaltungen

- mehrere Abstraktionsebenen mit zunehmendem Detail (Verzögerung um festes τ / individuelle Gatterverzögerungen / …)
  - Wir betrachten Gatterlaufzeiten als **Vielfache einer Grundverzögerung τ**
  - Jedes Gatter hat eine Laufzeit von **1 τ**
  - Leitungslaufzeiten werden ignoriert

Impulsdiagramme

- Darstellung der logischen Werte einer Schaltfunktion als Funktion der Zeit
- Gatterlaufzeiten werden berücksichtigt
- Jedes Gatter erhöht die Verzögerung um **1 τ**



Impulsdiagramme für verschiedene Schaltfunktionen. Blauer Bereich = undefiniert

Hazards

- die Eigenschaft einer Schaltfunktion, bei bestimmten Kombinationen der individuellen Verzögerungen ihrer Verknüpfungsglieder ein **Fehlverhalten** zu zeigen

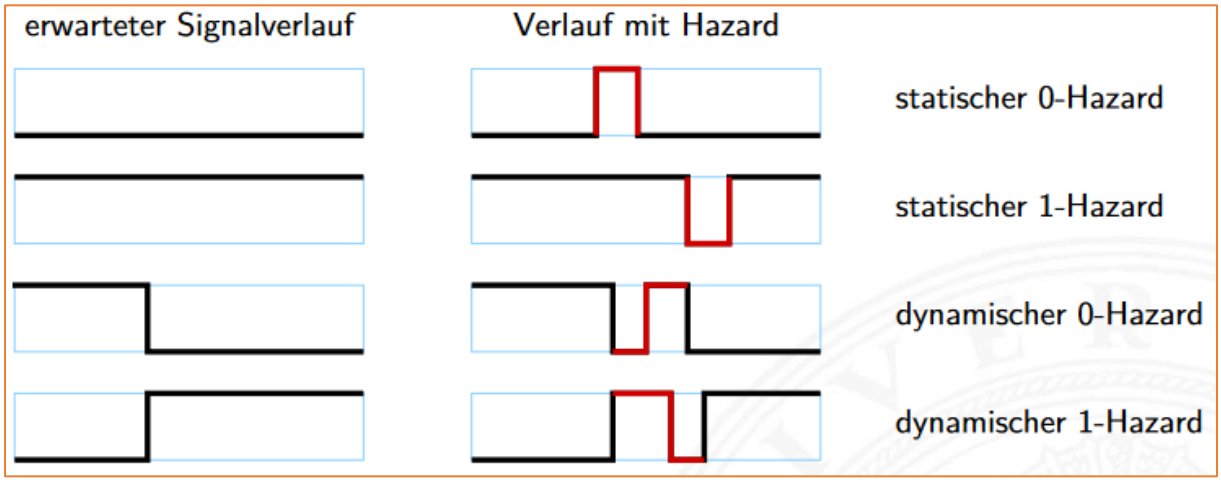
Klassifikation:

Nach der Erscheinungsform am Ausgang

statisch	der Ausgangswert soll unverändert sein, es tritt aber ein Wechsel auf
dynamisch	der Ausgangswert soll (einmal) wechseln, es tritt aber ein mehrfacher Wechsel auf

Nach den Eingangsbedingungen:

Strukturhazard	bedingt durch die Struktur der Schaltung, auch bei Umschalten eines einzigen Eingangswertes
Funktionshazard	bedingt durch die Funktion der Schaltung



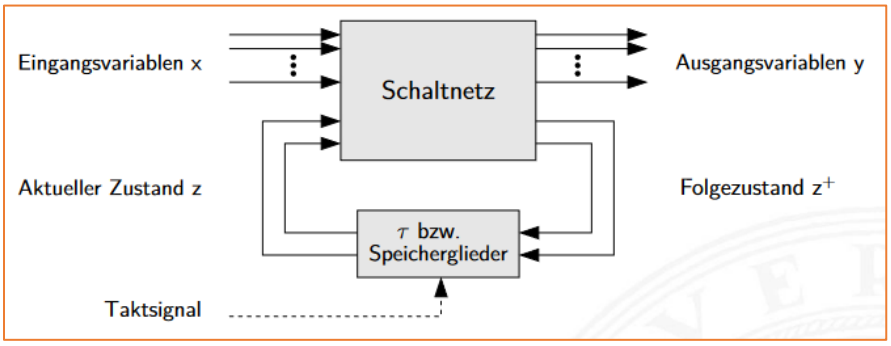
Schaltwerke

Schaltwerk Schaltung mit Rückkopplungen und Verzögerungen

- Ausgangswerte nicht nur von Eingangswerten abhängig, sondern auch von der Vorgeschichte
- interner Zustand repräsentiert „Vorgeschichte“, i.d.R. mehrere Zustandsbits

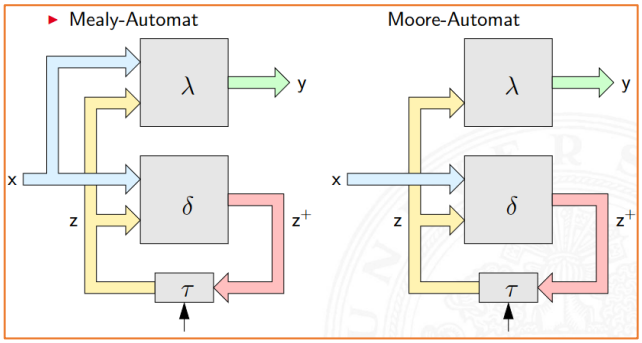
Aufbau

- Eingangsvariablen **x**
- Ausgangsvariablen **y**
- Folgezustand **z^+**
- Rückkopplung läuft über Verzögerung **τ**



Endliche Automaten

- Automatenmodell, auf dem Schaltwerke basieren
- Nächster Zustand **z^+** hängt von vorherigem Zustand **z** und Eingangsvariablen **x** ab
- Ausgabe **y** hängt ab von…
  - Zustand **(Moore)**
  - Zustand und Eingabe **(Mealy)**



Taktung von Schaltnetzen

- für gewöhnlich werden Schaltnetze **synchron** getaktet
- Schaltnetz braucht Zeit, um Output **(z^+, y)** für Input **(z, x)** zu berechnen
- Speicherglieder (Flip-Flops) speichern **neuen Zustand** des Schaltnetzes, nachdem dieser Berechnet wurde

Ablauf:

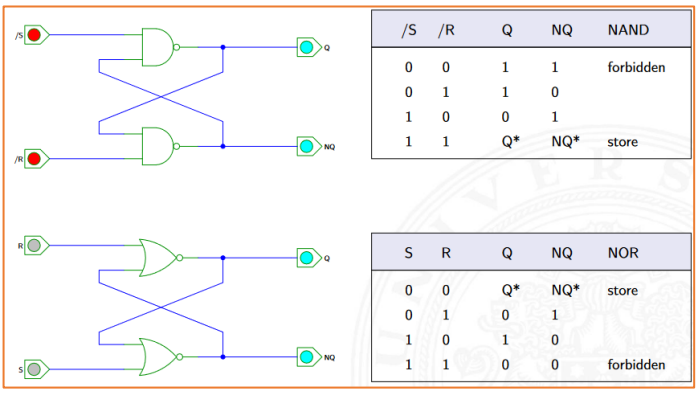
1. Speicherglieder speichern Zustand A
2. Speicherglieder leiten Zustand **z** an Schaltnetz weiter
3. Schaltnetz berechnet Folgezustand **z^+** und leitet diesen an Speicherglieder weiter
4. Speicherglieder übernehmen **z^+** beim nächsten Taktsignal

Flip-Flops

- Bauelemente, die jeweils 1 Bit speichern können
- Verschiedene Typen mit verschiedenen Funktionalitäten
- i.d.R. zwei Ausgänge Q (eigentlicher Output) und NQ ($= \overline{Q}$)

RS-Flipflop

- zwei Eingänge, **S** (set) und **R** (reset)
- bei S wird der Zustand gesetzt, bei R zurückgesetzt
- Verschiedene Realisierungen möglich



/S	/R	Q	NQ	NAND
0	0	1	1	forbidden
0	1	1	0	
1	0	0	1	
1	1	Q*	NQ*	store

S	R	Q	NQ	NOR
0	0	Q*	NQ*	store
0	1	0	1	
1	0	1	0	
1	1	0	0	forbidden

RS-Flipflop mit Takt

- drei Eingänge, **S** (set), **R** (reset) und **C** (clock)
- wie RS-Flipflop, aber Änderungen sind nur wirksam während **C** „aktiv“ sind
- mehrere Varianten, über **C** einen neuen Wert zu übernehmen
  - Pegelgesteuert
    - high-aktiv: Neuer Wert wird bei C = 1 übernommen
    - low-aktiv: Neuer Wert wird bei C = 0 übernommen
  - Flankengesteuert
    - Vorderflankensteuerung: Neuer Wert, wenn C von 0 auf 1 ändert
    - Rückflankensteuerung: Neuer Wert, wenn C von 1 auf 0 ändert

Taktung von Flip-Flops

- Flipflops werden entwickelt, um Schaltwerke einfacher entwerfen und betreiben zu können
- Aber: jedes Flipflop selbst ist ein asynchrones Schaltwerk mit kompliziertem internem Zeitverhalten
  - Daten- und Takteingänge dürfen sich **nicht gleichzeitig** ändern
  - Vorlauf- und Haltezeiten

Vorlaufzeit t_s Zeitintervall, innerhalb dessen das Datensignal vor dem nächsten Takt stabil anliegen muss

Haltezeit t_h Zeitintervall, innerhalb dessen das Datensignal nach einem Takt noch stabil anliegen muss

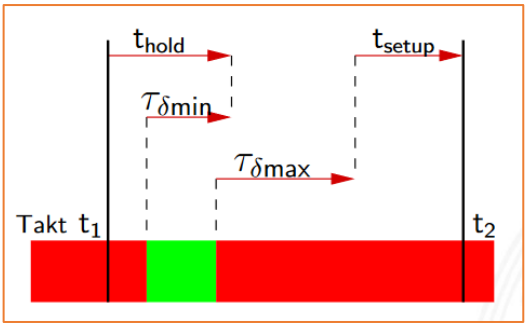
Durchlaufverzögerung τ_δ Dauer die ein Wert braucht, um Schaltfunktion zu durchlaufen, zwischen $\tau_{\delta_{min}}$ und $\tau_{\delta_{max}}$

Ausgangsverzögerung t_{FF} Dauer die ein Wert braucht, um Flipflop zu durchlaufen

Wertänderung zwischen Takten $\mathbf{t_1}$ und $\mathbf{t_2}$:

- Wert darf sich **nicht** in t_{hold} und t_{setup} ändern
- Verzögerungen $\boldsymbol{\tau_{\delta_{min}}}$ und $\boldsymbol{\tau_{\delta_{max}}}$ sind zu beachten
- Früheste Änderung ist ab $t_1 + t_{hold} - \tau_{\delta_{min}}$ möglich
- Späteste Änderung ist bis $t_2 - t_{setup} - \tau_{\delta_{max}}$ möglich

$$\{\boldsymbol{t_1 + t_{hold} - \tau_{\delta_{min}};\ t_2 - t_{setup} - \tau_{\delta_{max}}}\}$$



Wert des Flipflops darf sich nur im grünen Bereich ändern

Berechnung der maximalen Taktfrequenz:

- Die Taktfrequenz **Δt** lässt sich aus den oberen Werten berechnen
- Es gelten die Regeln:

$$\Delta \boldsymbol{t} \geq (\boldsymbol{t_{FF} + \tau_{\delta_{max}} + t_{setup}})$$

$$\Delta \boldsymbol{t} \geq (\boldsymbol{t_{hold} + t_{setup}})$$

Beschreibung von Schaltwerken

Flusstafel

- Tabelle für die **Folgezustände** als Funktion des aktuellen Zustands und der Eingabewerte
- beschreibt das δ-Schaltnetz

Ausgangstafel

- Tabelle für die **Ausgabewerte** als Funktion des aktuellen Zustands (bzw. des Zustands und der Eingabewerte)
- beschreibt das λ-Schaltnetz

Bsp.: Ampel

Zustand	Codierung		Folgezustand	
	z_1	z_0	z_1^+	z_0^+
rot	0	0	0	1
rot-gelb	0	1	1	0
grün	1	0	1	1
gelb	1	1	0	0

Flusstafel einer Ampel

Zustand	Codierung		Ausgänge		
	z_1	z_0	rt	ge	gr
rot	0	0	1	0	0
rot-gelb	0	1	1	1	0
grün	1	0	0	0	1
gelb	1	1	0	1	0

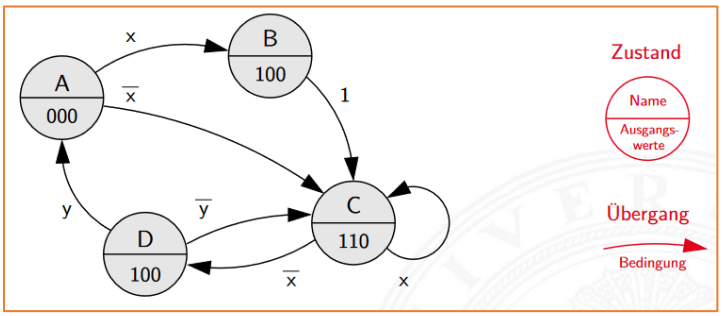
Ausgangstafel einer Ampel

Zustandsdiagramm

- Grafische Darstellung eines Schaltwerks
  - 1 Knoten = 1 Zustand
  - 1 Kante = 1 möglicher Übergang

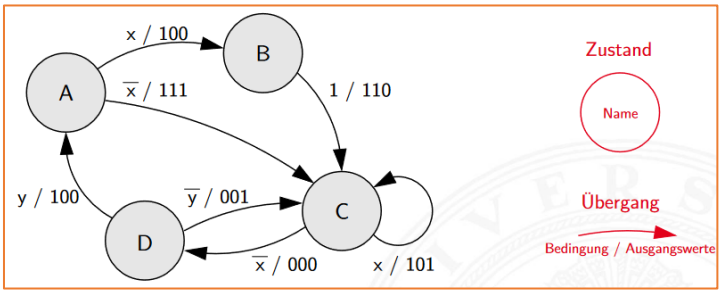
Moore Automat

- Ausgangswerte hängen nur vom Zustand ab
- Werden direkt am Zustand/Knoten notiert



Mealy Automat

- Ausgangswerte hängen von Zustand und Eingabe ab
- Werden an Kanten notiert notiert



Rechnerarchitektur 1

Rechnerarchitektur hat zwei wichtige Aspekte:

1. **Operationsprinzip**:
    - das funktionelle Verhalten der Architektur (Befehlssatz)
    - = Programmierschnittstelle
    - = ISA - Instruction Set Architecture
    - = „Software“

2. **Hardwarearchitektur**
    - der strukturelle Aufbaue des Rechnersystems (Mikroarchitektur)
    - = Hardware

Speicher

- System zur Speicherung von Information
- als Feld von N Adressen mit je m-bit Speicherworten
    - typischerweise mit n-bit Adressen und $N = 2^n$
    - Kapazität also $2^n * m$ Bits

ROM

- Read-Only Memory
- kann nur gelesen werden, nicht beschrieben

RAM

- Random-Access Memory
- Speicher, der im Betrieb gelesen und beschrieben werden kann
- 2 Varianten: **SRAM** und **DRAM**

SRAM	DRAM
= static RAM	*= dynamic Ram*
Inhalt bleibt gespeichert, solange Betriebsspannung anliegt	Inhalt muss nach lesen, schreiben und in regelmäßigen Zeitabständen **refresht** werden
Schnell	10x langsamer als SRAM
Hoher Platzbedarf	Niedriger Platzbedarf

Bussysteme

- elektrische (und logische) Verbindung
- mehrere Geräte können gleichzeitig angeschlossen sein
- beliebig viele dürfen gleichzeitig vom Bus lesen, nur einer darf gleichzeitig auf den Bus senden

Da nur eine Komponente gleichzeitig auf den Bus schreiben darf, muss dieser **arbitriert** werden.

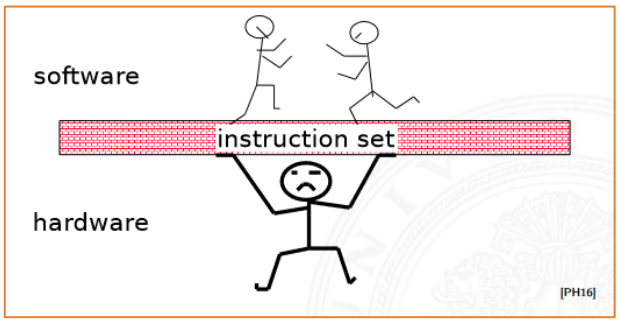
2 Varianten:

1. **Zentrale Arbitrierung**
   - *Arbiter* gewährt Bus-Requests
   - Verschiedene Strategien der Vergabe

2. **Dezentrale Arbitrierung**
   - Protokollbasiert
   - Komponenten bestimmen selbst, ob sie zu einem Zeitpunkt senden können/wollen

Instruction Set Architecture

Alle für den **Programmierer** sichtbaren Attribute eines Rechners.



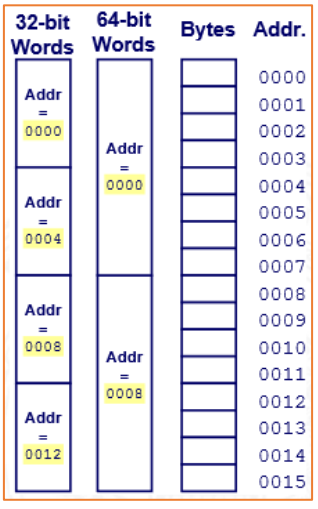
Speicherorganisation

- i.d.R. sind Datentypen mehrfache von 8-bit
  - 16-bit, 32-bit, 64-bit, …
- Speicher ist i.d.R. Wortweise aufgebaut, einzelne Speichersegmente (Adressen) sind ein Mehrfaches von 8-bit
  - typischerweise 32-bit oder 64-bit

<u>Wort-basierte Organisation des Speichers</u>

- Speicher ist Wort-orientiert
- Addressierung ist Byte-orientiert
  - um Adresse anzusprechen, wird erstes Byte der Adresse angegeben

<u>Bsp.:</u> bei 32-bit Speicherworten beginnt alle 4 Bytes ein neues Wort



<u>Byte-Order</u>

- wie sollen die Bytes innerhalb eines Wortes angeordnet werden?
- 2 Konventionen: **Big Endian** und **Little Endian**
  - **Big Endian** Vordestes Byte hat die kleinste Adresse
  - **Little Endian** Vorderstes Byte hat die größte Adresse

Bsp.: 32-bit Speicherworte = 4 Byte
Wir wollen das Wort an Adresse 0008 lesen.

Adresse	0008	0009	0010	0011
Bytes	2F	FF	11	00

Big Endian: 2F FF 11 00 = 805.245.184
Little Endian: 00 11 FF 2F = 1.179.439

Memory Map

- CPU kann *im Prinzip* alle möglichen Adressen ansprechen
- Memory Map ist eine Datenstruktur, die angibt, wie Speicher angeordnet ist
  - Aufteilung in *read-write*- und *read-only*-Bereiche
  - *Read-only* für eingebettete Systeme
  - Treiberverwaltung
- Ist selbst auch im Speicher gespeichert

Speicherhierarchie

- obere Ebenen sind schneller, aber teurer pro Byte
- untere Ebenen sind langsamer, aber billiger

Cache schneller Zwischenspeicher, überbrückt Geschwindigkeits-unterschied zwischen CPU und Hauptspeicher

Ebene	Ebenenname
L0	Register
L1	L1 Cache (SRAM)
L2	L2 Cache (SRAM)
L3	L3 Cache (SRAM)
L4	Main Memory (DRAM)
L5	Local Secondary Storage
L6	Remote Secondary Storage

Befehlszyklus

1. **FETCH**
   - Programmzähler (PC) liefert Adresse für Speicher
   - Lesezugriff an der Adresse
   - Resultat wird in Befehlsregister (IR) abgelegt
   - Programmzähler wird inkrementiert

2. **DECODE**
   - Befehlsregister leitet Befehl an Decoder weiter
   - Decoder entschlüsselt Opcode und Operanden
     - für jeden Opcode ex. klare Vorgaben an welchen Stellen welche Daten stehen
     - kann von Opcode zu Opcode abweichen
   - leitet Steuersignale an die Funktionseinheiten

3. **EXECUTE**
    - Ausführung des Befehls durch Aktivierung der Funktionseinheiten
    - ggf. Programmzähler ändern (Jump/Branch Befehl)

Adressierungsarten

n-Adress Maschinen

Adress Format	Beschreibung
3-Adress Format	X = Y + Z
2-Adress Format	X = X + Z
1-Adress Format	ACC = ACC + Z - Befehle nutzen das Akkumulator-Register als Zwischenspeicher
0-Adress Format	TOS = TOS + NOS - Stapelspeicher: *top of stack, next of stack* - arithm. Befehle verarbeiten den TOS mit dem NOS - Adressverwaltung entfällt

Beispiel: Z = (A-B) / (C + D*E) Hilfsregister: T

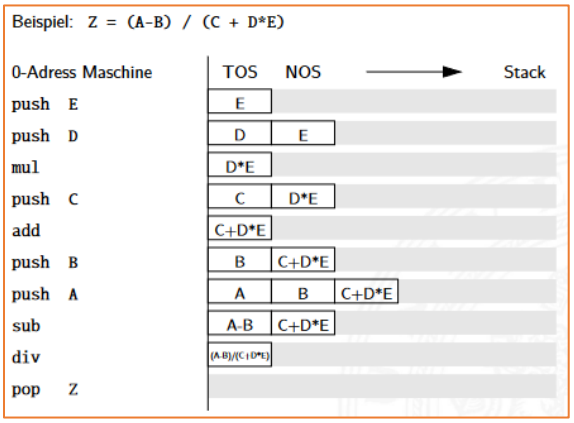
3-Adress Maschine
```
sub Z, A, B
mul T, D, E
add T, C, T
div Z, Z, T
```

2-Adress Maschine
```
mov  Z, A
sub  Z, B
mov  T, D
mul  T, E
add  T, C
div  Z, T
```

1-Adress Maschine
```
load D
mul  E
add  C
stor Z
load A
sub  B
div  Z
stor Z
```

0-Adress Maschine
```
push E
push D
mul
push C
add
push B
push A
sub
div
pop  Z
```

Mögliche Befehlslisten für die verschiedenen Adressformate



Abarbeitung eines Programms auf einer 0-Adress Maschine

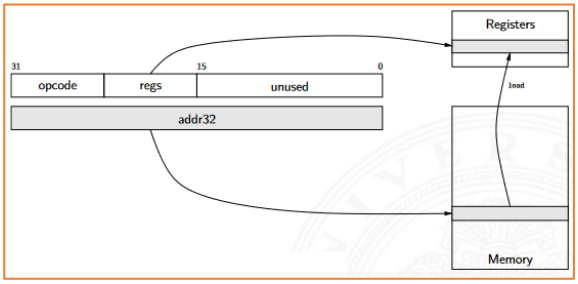
Operandenadressierung

immediate

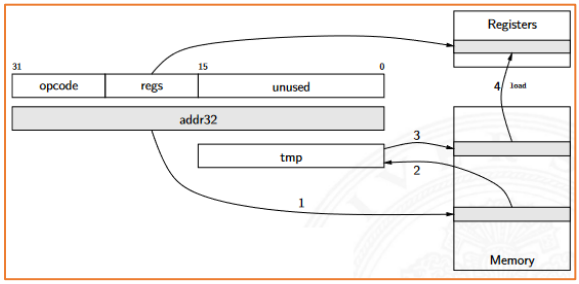
- Operand steht direkt im Befehl
- kein zusätzlicher Speicherzugriff
- aber Länge des Operanden beschränkt

direkt

- Adresse des Operanden steht im Befehl
- keine zusätzliche Adressberechnung
- ein Speicherzugriff



indirekt



- Adresse eines **Pointers** steht im Befehl
- erster Speicherzugriff liest Wert des Pointers
- zweiter Speicherzugriff liefert Operanden
- flexibel, aber langsam

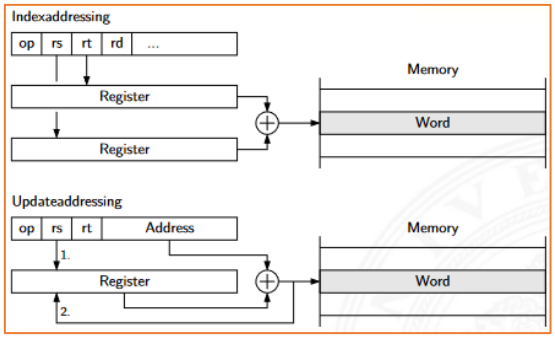
register

- wie Direktmodus, aber Register statt Speicher

register-indirekt

- Befehl spezifiziert ein Register
- Register enthält Speicheradresse auf die zugegriffen wird
- ein Speicherzugriff

indiziert



- Angabe mit Register und Offset
- Inhalt des Registers liefert Basisadresse
- Speicherzugriff auf (Basisadresse + Offset)
- ideal für Array- und Objektzugriffe
- Hauptmodus in RISC-Rechnern (auch: „Versatz-Modus“)

CISC

= „Complex Instruction Set Computer“

Rechnerarchitekturen mit irregulärem, komplexem Befehlssatz und (unterschiedlich) langer Ausführungszeit

- große Instruktionssätze (> 300 Befehle)
- unterschiedlich lange Befehlsformate
- mehrere Schreib- und Lesezugriffe pro Befehl
- viele verschiedene Datentypen
- heutzutage eher selten genutzt

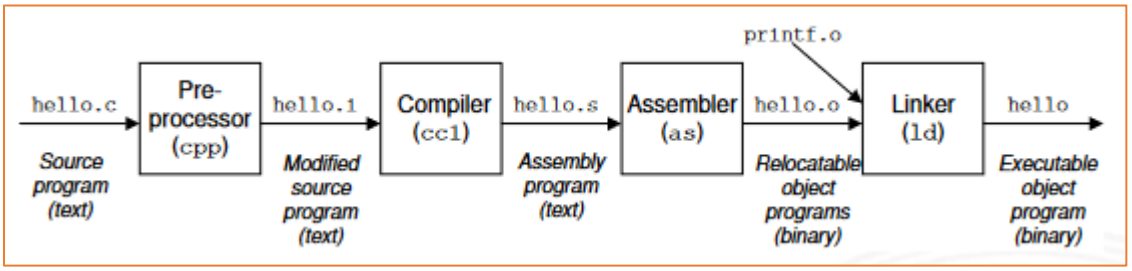
RISC

= „Regular Instruction Set Computer“

Rechnerarchitekturen mit regulären Befehlssätzen und ähnlich langen Ausführungszeiten

- reduzierte Anzahl, einfache Instruktionen
- reguläre Struktur, z.B. 32-bit Wortbreite, 32-bit Befehle
- nur **ein-Wort** Befehle
- alle Befehle in gleicher Zeit ausführbar ⇒ Pipeline-Verarbeitung
- Speicherzugriff nur durch „Load“ und „Store“ Anweisungen

Assembler-Programmierung



- Programme haben verschiedene Repräsentationen
  - Hochsprache
  - Assembler
  - Maschinensprache

Assemblercode

- Sehr hardwarenahe Programmiersprache (≠ Maschinensprache)
- Zugriff auf kompletten Befehlssatz und alle Register einer Maschine
- Ein Befehl pro Zeile
- Label für Sprünge (`goto Label`)

Befehle:

- arithmetische/logische Funktionen auf Register und Speicher
- Datentransfer zwischen Speicher und Registern
- Kontrolltransfer (Jumps/Calls/Interrupts)

Format	Code	Beschreibung
C-Code	`*dest = t;`	Speichert Wert `t` nach Adresse aus `dest`
Assemblercode	`movq %rax, (%rbx)`	Kopiere einen 8-Byte Wert (Quad Word) in den Hauptspeicher
Objektcode (x86)	`0x40059e: 48 89 03`	3-Byte Befehl an Adresse `0x40059e`

Assembler

- übersetzt `.s` zu `.o`
- (fast) vollständig Maschinenlesbar
- keine Verknüpfung zu Code aus anderen Dateien/Bibliotheken
  - kommt erst mit Linker/Binder

Linker / Binder

- löst Referenzen zwischen Dateien auf
- kombiniert Code mit statischen Laufzeit-Bibliotheken
- dynamische Bibliotheken werden erst **zur Laufzeit** verknüpft

Adressierungsarten

- Als Beispielsbefehl wird hier `movq` genommen
  - transferiert ein 8-Byte „long“ Wort
  - Kann direkt Wert annehmen, oder Wert aus Register oder Speicher lesen
- es wird jeweils ein Wert in Register `%rax` gespeichert

Datenquelle	Befehl	Beschreibung
Immediate	`movq $0x4, %rax`	Speichert eine 4 in `%rax`
Register	`movq %rdx, %rax`	Speichert den Inhalt von `%rdx` in `%rax`
Memory	`movq (%rdx), %rax`	Speichert den Inhalt von Speicheradresse in `%rdx` in `%rax`
Displacement	`movq 4(%rdx), %rax`	Speichert den Inhalt von Speicheradresse **+ 4** in `%rdx` in `%rax`

Indizierte Adressierung

- „erweitertes Displacement“
- allgemeine Form: `Imm(Rb, Ri, S) → Mem[Reg[Rb] + S * Reg[Ri] + Imm]`
  - **`Imm`** Offset
  - **`Rb`** Basisregister
  - **`Ri`** Indexregister
  - **`S`** Skalierungsfaktor (1, 2, 4 oder 8)

Bsp.: gegeben:
```
%rax = 2000
%rdx = 8
```

Zugriff auf `16(%rax, %rdx, 2)` liefert folgenden Wert:

```
Mem[2000 + 2 * 8 + 16] = Mem[2032]
```

Zustandscodes (Flags)

- Zeigen bestimmte Informationen nach Operationen an, bspw. ob Addition zu Überlauf führte
  - **CF** Carry Flag
  - **ZF** Zero Flag
  - **SF** Sign Flag
  - **OF** Overflow Flag
- können genutzt werden, um Programmfluss zu steuern
- mehrere Möglichkeiten diese zu setzen

1. **Implizite Aktualisierung durch arithmetische Operation**
   - Beispiel: `addq (src), (dst)` (in C: t = a + b)
   - **CF** setzen, wenn höchstwertiges Bit Übertrag generiert
   - **ZF** setzen, wenn t = 0
   - **SF** setzen, wenn t < 0
   - **OF** setzen, wenn Zweierkomplement überläuft

2. **Explizites Setzen durch Vergleichsoperation**
   - Beispiel: `compq (src2), (src1)`
     - wie Berechnung von `(src1) – (src2)`, aber ohne Abspeichern des Resultats
   - **CF** setzen, wenn höchstwertiges Bit Übertrag generiert
   - **ZF** setzen, wenn `(src1) = (src2)`
   - **SF** setzen, wenn `(src1 – src2) < 0`
   - **OF** setzen, wenn Zweierkomplement überläuft

3. **Explizites Setzen durch Testanweisung**
   - Beispiel: `testq (src2), (src1)`
     - wie Berechnung von `(src1) & (src2)`, aber ohne Abspeichern des Resultats
     - praktisch, wenn eine der Operanden eine Bitmaske ist
   - **ZF** setzen, wenn `(src1) & (src2) = 0`
   - **SF** setzen, wenn `(src1) & (src2) < 0`

- Flags können genutzt werden, um bedingte Sprünge zu implementieren
  - `if/else` Anweisung

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

Sprungbefehle und deren Abhängigkeiten

Label

- symbolischer Name für bestimmte Adressen
- können für Sprünge genutzt werden
- werden vom Programmierer / Compiler vergeben

```
loop:
        movq    %rdi, %rax
        compq   %rsi, %rdi
        je      loop
        ret
```

Falls `%rsi` und `%rdi` gleich sind, springt `je` zum Label `Loop`

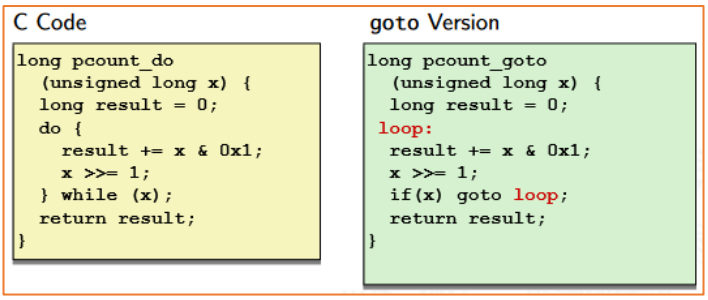
Übersetzen von bekannten Programmierkonzepten

`if/else`

- lässt sich direkt über konditionale Anweisungen implementieren
- Bsp.: `cmovle` anstatt `cmov`

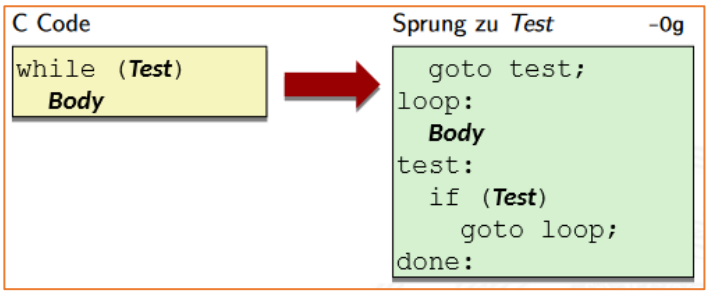
do … while

- Label an Anfang von Schleifenkörper
- konditionalen Sprungbefehl an Ende von Schleifenkörper



while

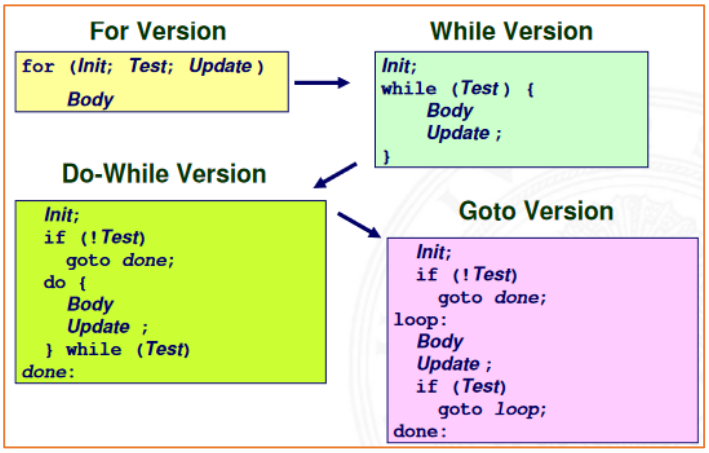
- Label an Anfang von Schleifenkörper
- eigenen Testblock am Ende von Schleifenkörper mit konditionalen Sprungbefehl zum Anfang
- zu Beginn wird zuerst in den Testblock gesprungen



for

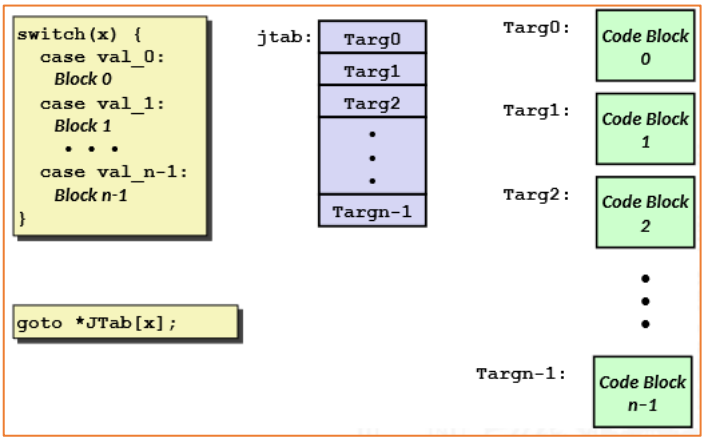
- 4 Blöcke: Init, Test, Update und Body
- Init wird zuerst und einmalig ausgeführt
- danach Test, der bestimmt, ob Schleife überhaupt ausgeführt wird

- Body und Update werden jede Schleife nacheinander ausgeführt
- danach Test, der bestimmt, ob Schleife wiederholt wird



switch (Sprungtabelle)

- Compiler erzeugt Codesegment für jeden case Zweig
- Sprungtabelle wird für alle Adresse der Codesegmente erstellt
- bei Ausführung:
  - Bestimmung des Falls
  - Indizierter Zugriff auf Adresse des Codesegments
  - Ausführen des Codesegments



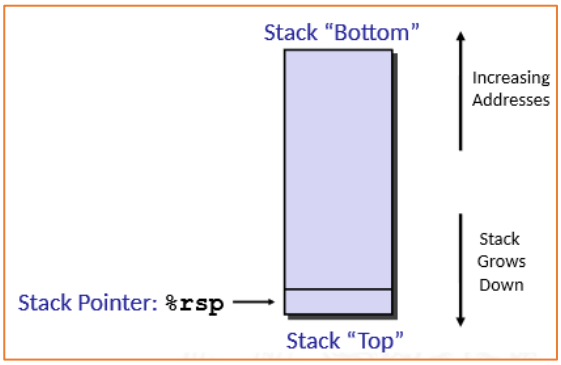
Links: C-Code
Mitte: Sprungtabelle
Rechts: Codesegmente

Stack

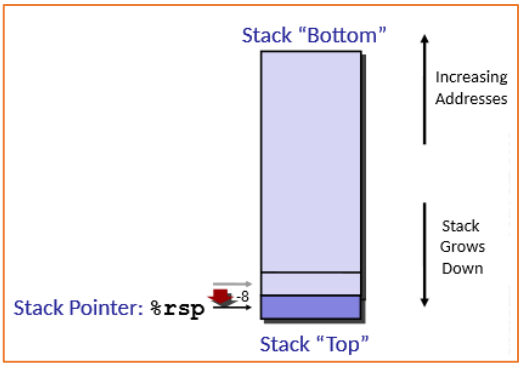
- Speicherregion
- Wächst nach unten
- Register **`%rsp`** zeigt immer auf die aktuelle Stack-Adresse/oberstes Element
- Speichert verschiedene Daten, die für Funktionen wichtig sind
  - Aufruf-Parameter von Funktionen
  - Lokale Variablen
  - Rücksprungadresse zu Hauptfunktion
  - Rückgabewerte

Speichern eines Werts im Stack (Push)

- Befehl: `pushq (src)`
  1. Operanden aus `(src)` hole
  2. `%rsp` um 8 dekrementieren
  3. Operanden bei `%rsp` speichern



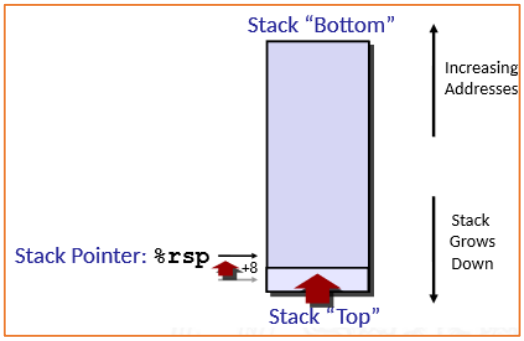
Stack vor Push



Stack nach Push

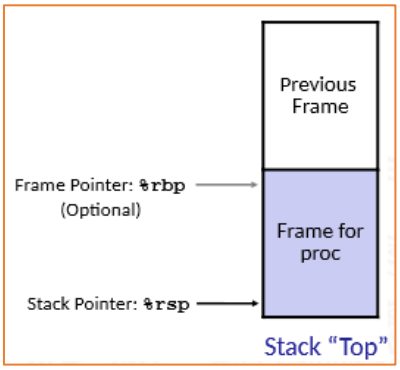
Holen eines Werts aus dem Stack (Pop)

- Befehl: `popq (dst)`
  1. Operanden von `%rsp` lesen
  2. `%rsp` um 8 inkrementieren
  3. Wert in `(dst)`speichern



Stack nach Pop

Stack-Frame



- Bereich im Stack, der alle Daten für einen Funktionsaufruf enthält
  - Rücksprungadresse zu Hauptfunktion
  - ggf. lokale Variablen
  - ggf. temporäre Daten
- Bei Funktionsaufruf wird ein neuer Stack-Frame erstellt
- Beispiel verschachtelter Funktionsaufruf: **Folie 981 ff.**

2 Konventionen zur Frame-Speicherung:
Bsp.: Funktion **yoo** ruft Unterfunktion **who** auf

- **Caller-Saved**
  - **yoo** speichert in seinen Frame vor Prozeduraufruf

- **Callee-Saved**
  - **who** speichert in seinen Frame vor Ausführung

Arrays

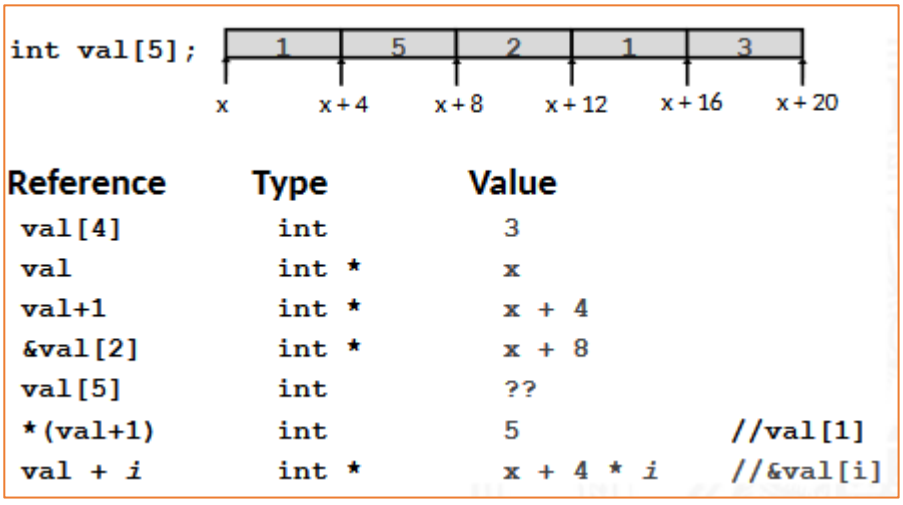
- Datenstruktur zur Speicherung einer festen Anzahl von Elementen vom gleichen Datentyp
- fortlaufender Speicherbereich von fester Länge
- Länge ist abhängig von Speicherplätzen und Bytegröße des Datentyps

Intel x86	as	Bytes	C (Architektur-, Compiler-, OS-abhängig)
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
doubleword	l	4	[unsigned] int / long
quadword	q	8	[unsigned] long / long long

Intel x86	as	Bytes	C (as: *Gnu ASsembler*)
Single	s	4	float
Double	l	8	double
Extended	t	10/12	long double

Bsp.: int-Array mit 5 Speicherplätzen (`int val[5];`)

- Bytegröße int: 4 Byte
- 5 Speicherplätze
- 4 * 5 = **20 Bytes** im Speicher für Array



Verschiedene Zugriffsarten auf Array val

Linker und Loader

- Programm ist in mehrere Tile aufgeteilt
  - Bsp. Funktionen main und sum, main ruft sum auf
- Teilprogramme werden einzeln von Compiler umgewandelt
  - main.c -> main.o
  - sum.c -> sum.o
- Compilte Teilprogramme werden von Linker zu einem Programm zusammengefügt

- Wenn System-/Programmierspracheneigene Funktionen aufgerufen werden. müssen diese beim Kompilieren mit eingebunden werden
- 2 Arten:
  - **Static Linking**
    - Funktionen aus Bibliotheksarchiven (.a) werden in ausführbares Programm eingebaut
    - nicht genutzte Funktionen werden entfernt
    - Linken **während Compilierung**
  - **Dynamic Linking**
    - Bibliotheken werden erst beim Laden in Speicher oder womöglich. erst zur Laufzeit dazugelinkt
    - erlaubt gemeinsame Nutzung von mehreren Prozessen, Bibliotheksfunktionen liegen aber nur ein Mal im Speicher

Rechnerarchitektur 2

Pipelining

- Erhöhung der Effizienz von Operationen, in dem mehrere Elemente gleichzeitig abgearbeitet werden können
- Aufteilen von Operationen in (möglichst gleich lange) Teilschritte (Stages)
- Taktfrequenz muss mindestens die Latenzzeit der langsamsten Stage sein
- in jeder Stage kann gleichzeitig ein Element abgearbeitet werden
- nach durchlaufen einer Stage geht ein Element in die nächste Stage und ein anderes rückt nach
- => wie ein Fließband

Pipelinestage

- einzelner Abschnitt der Pipeline
- besteht aus Logikteil und Register
- kann nur ein Element gleichzeitig enthalten
- alle Pipelinestages sollten möglichst die gleiche Durchlaufdauer haben

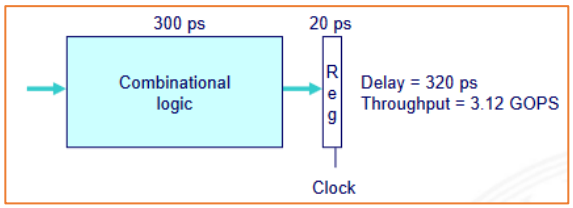
Ablauf

1. Element ist in Register i
2. Element durchläuft Logikteil der nächsten Stage
3. Element erreicht Register i+1
4. Bei nächstem Takt speichert Register i+1 das Element

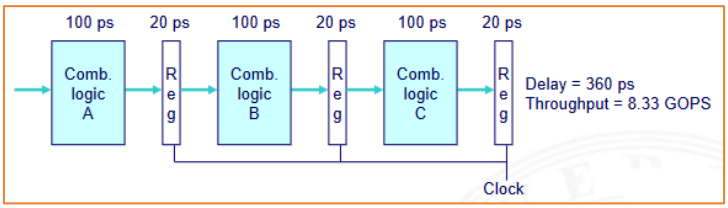
Analyse

- **Latenz**: Zeit, die eine Instruktion braucht um Pipeline zu durchlaufen
  i.d.R. in Sekunden angegeben
- **Durchsatz**: Anzahl Instruktionen, die in einem Zeitraum abgeschlossen werden
  i.d.R. in OPS angegeben, *Operations per Second*
  $= \frac{1\,s}{Latenz}$

- Bei K Pipelinestufen mit je N Instruktionen:
  - ohne Pipeline: N * K Taktzyklen
  - mit Pipeline: K + N – 1 Taktzyklen
  - **Speedup:** $S = \frac{N * K}{K + N - 1}, \quad \lim_{N \to \infty} S = K$



Nur eine Stage,
weniger Latenz aber auch weniger Durchsatz



Mehrere Stages,
leicht mehr Latenz aber auch höherer Durchsatz

- **Schlecht** für Pipelining:
  - gleichzeitiger Zugriff auf eine Ressource durch mehrere Pipelinestufen
  - Ergebnisse von Instruktionen werden innerhalb der Pipeline benötigt
  - Sprungbefehle in der Pipelinesequenz

- **Sehr schlecht** für Pipelining
  - Unterbrechung des Programmkontexts
    - Interrupt, System-Call, Exceptions, Prozesswechsel, …

Parallelität

Begriffe:

- **Antwortzeit** Gesamtzeit zwischen Programmstart und -ende, inklusive Unterbrechungen
  - $performance = \frac{1}{excution\ time}$
- **Ausführungszeit** reine CPU-Zeit
  - Anzahl der Befehle * Zeit pro Befehl
- **Durchsatz** Anzahl bearbeitete Programme / Zeit
- **Speedup** $s = \frac{performance\ x}{performance\ y} = \frac{execution\ time\ y}{execution\ time\ x}$

Amdahls Gesetz

- wird genutzt, um Einfluss der Beschleunigung einer Teilfunktion auf Programm zu berechnen
- dabei besitzt Programm P die Funktion X mit Anteil 0 < f < 1
- X wird ersetzt mit schnellerer Funktion X‘, Speedup S_X

Speedup: $S_{gesamt} = \frac{1}{(1-f) + f/S_X}$

Speedup bei Parallelrechner mit n-Prozessoren: $S_{gesamt} = \frac{1}{(1-f) + k(n) + f/n}$

n = Anzahl Prozessoren
f = Anteil parallelisierbarer Berechnung
(1 – f) = Anteil nicht parallelisierbarer Berechnung
k() = Kommunikationsoverhead zwischen den Prozessoren

Superskalar

- Superskalare CPUs besitzen mehrere Recheneinheiten: 4 … 12
- in jedem Takt werden (dynamisch) mehrere Instruktionen eines konventionell linearen Instruktionsstroms abgearbeitet
- pro Takt kann mehr als eine Instruktion gestartet werden

- Mehrere Arten von Datenabhängigkeiten
  - **RAW** – Read after Write
    Instruktion I_x darf Daten erst lesen, wenn I_{x-n} geschrieben hat
  - **WAR** – Write after Read
    Instruktion I_x darf Daten erst lesen, wenn I_{x-n} gelesen hat
  - **WAW** – Write after Write
    Instruktion I_x darf Daten erst überschreiben, wenn I_{x-n} geschrieben hat
- **WAR** und **WAW** können durch „Register Renaming“ gelöst werden
- **RAW** ist eine „*echte*“ Abhängigkeit, Forwarding ist kaum möglich und in superskalaren Pipelines extrem aufwändig

Cache

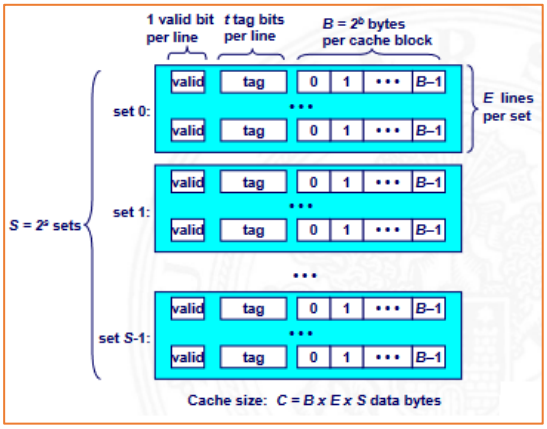
Begriffe

Treffer (Hit) Zugriff auf Daten die bereits im Cache sind
Fehler (Miss) Zugriff auf Daten die nicht im Cache sind
Treffer-Rate R_{Hit} Wahrscheinlichkeit, dass Daten im Cache sind
Fehler-Rate R_{Miss} **$1 - R_{Hit}$**
Hit-Time T_{Hit} Zeit, bis Daten bei Treffer geliefert werden
Miss-Penalty T_{Miss} zusätzlich benötigte Zeit bei Miss

- Mittlere Speicherzugriffszeit = **$T_{Hit} + T_{Miss} * R_{Miss}$**

Aufbau

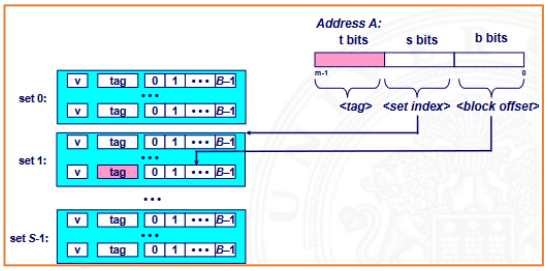
- Cache ist ein Array von Speicher-Bereichen („sets“)
- jeder Bereich enthält eine oder mehrere Zeilen
- jede Zeile enthält ein „valid“-Bit
  - zeigt an, ob Zeile genutzt
- jede Zeile enthält einen Datenblock
- jeder Block enthält mehrere Bytes



Adressierung von Caches:

- Adresse **A** ist im Speicher, ist **m** Bits lang
- **A** wird in 3 Teile aufgeteilt:
  - tag {m-1; k}
  - set index {k+1; l}
  - block offset {l+1; 0}

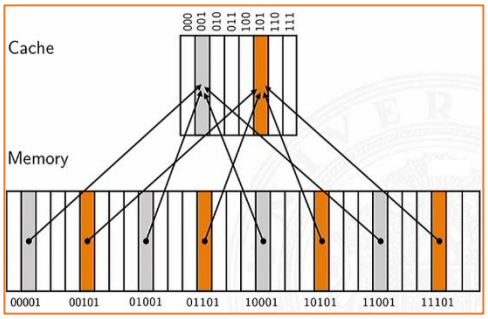
Bsp.: A = FF AD 10



- gegeben: Aufteilungsschema
  - bspw. tag = FF
  - set index = AD
  - block offset = 10
- Wert von A wird in Set **AD**, Offset **10** und Tag **FF** gespeichert

Abbildungsarten:

- **Direkt abgebildet**
  - jede Adresse ist genau einer Speicherzelle im Cache zugeordnet
  - nur eine Zeile pro Cachezelle
  - wie *Hashing*
  - Problem: Cache Thrashing
    - Zugriff auf Adressen mit gleichen Cacheabbildungen wird ineffizient



Direkte Abbildung.
Die Adressen werden anhand ihrer letzten 3 Bits dem Cache zugeordnet

- **Bereichsassoziativ**
  - jeder Speicheradresse ist ein Bereich S mit mehreren (2, 4, 8, …) Cachezeilen zugeordnet
  - Cachezeilen können sich (im Vergleich zu direkter Abbildung) um ihre *tags* unterscheiden

- **Voll-assoziativ**
  - jeder Speicheradresse des Speichers kann jede beliebige Cachezeile zugeordnet werden
  - nur für sehr kleine Caches realisierbar

Cache-Misses

Cold Miss	Cache ist noch leer
Conflict Miss	Kapazität reicht aus, aber Daten werden immer auf selben Block abgebildet
Capacity Miss	Anzahl aktiver Blöcke ist größer als Kapazität des Caches

Ersetzungsstrategie

- was soll ersetzt werden, wenn der Cache voll ist?
- **LRU** der älteste, nicht benutzte Cache Eintrag
- **LFU** der am wenigsten benutzte Cache Eintrag

Betriebssysteme

Aufgaben eines Betriebssystems

- Prozessverwaltung
- Speicherverwaltung
- Ein-/Ausgabeverwaltung
- Interruptverarbeitung

Interrupts

- sequenzieller Ablauf des Programms wird unterbrochen
  - Programm wartet bspw. auf Daten aus Speicher
- in der Zeit soll der Prozessor nicht stillstehen, sondern kann weiterarbeiten
- ⇒ Wechsel zu anderem Prozess

Interruptquellen:

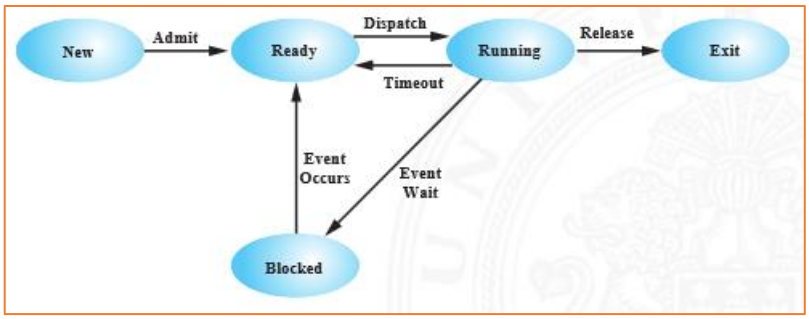
- I/O
- Exception
- Speicher
- Regelmäßiges Interrupt durch Betriebssystem, um sicherzustellen dass alle Prozesse abgearbeitet werden

Prozesse

- zentral verwaltete Einheit im Betriebssystem
- Prozess = Programm während der Ausführung
- besteht aus mehreren Komponenten
    - das ausführbare Programm
    - die zugehörigen Daten
    - der Programmkontext
        - prozessspezifische Daten des OS
        - Inhalt der Prozessorregister
        - Wartet der Prozess grade auf Ereignisse?
        - Prioritäten/Rechte/…

Context-Switchting

Ablauf:



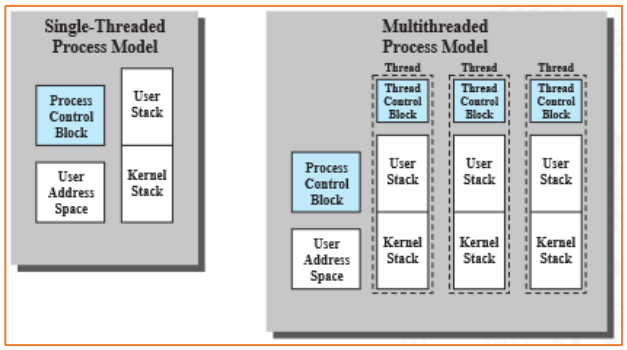
- Prozesse werden regelmäßig gestartet und beendet
    - Startet in *New*
    - Endet in *Exit*
- Wartet ein Programm auf Ereignis, wird es interrupted und in *Blocked* verschoben
    - bei Ereignis geht dieses in *Ready*
- Wird ein Programm vom BS „getimoutet“, wird es interrupted und in *Ready* verschoben
- Nachdem Prozess unterbrochen wird, nimmt sich der CPU den nächsten von *Ready*

Modelle:

- einzelne Phasen (Ready, Blocked, Running) können Queues sein
- Queues können Prioritäten haben
- Scheduling Algorithmen, die auf unterschiedliche Arten den nächsten auszuführenden Prozess bestimmen

Threads

- eigene Ausführungseinheit in Prozess
- ein Prozess kann mehrere Threads haben
- eigener Zustand, Kontext, Stack
- Zugriff auf Datenstrukturen und Ressourcen des Prozesses



Multithreading mehrere parallele Ausführungen innerhalb eines Prozesses, d.h. nicht ein „Ablauf“ von Code, sondern mehrere

Nebenläufigkeit

- abwechselndes und überlapptes Rechnen
- Timing der Abarbeitung nicht vorhersehbar

Begriffe

atomare Operation	Funktion oder Aktion die entweder komplett oder gar nicht gemacht werden kann. Kann nicht unterbrochen werden
Critical Section	Codebereiche mehrerer Prozesse, in denen auf gemeinsame Ressourcen (z.B. Speicher) zugegriffen wird
Deadlock	zwei oder mehr Prozesse können nicht weiterarbeiten, da sie gegenseitig aufeinander warten
Mutual Exclusion	wenn ein Prozess in seiner **Critical Section** ist, kann kein zweiter Prozess in einer **Critical Section** sein, der die gleichen Ressourcen nutzt. - notwendig, um **Race Conditions** zu vermeiden - kann zu Deadlock und Starvation führen
Race Condition	mehrere Threads/Prozesse lesen und schreiben Daten, wobei das Ergebnis von deren zeitlicher Reihenfolge abhängig ist
Starvation	ein lauffähiger Prozess könnte (weiter-) arbeiten, wird aber nie bedient.

Mutual Exclusion

Es gibt mehrere Möglichkeiten, Mutex zu implementieren:

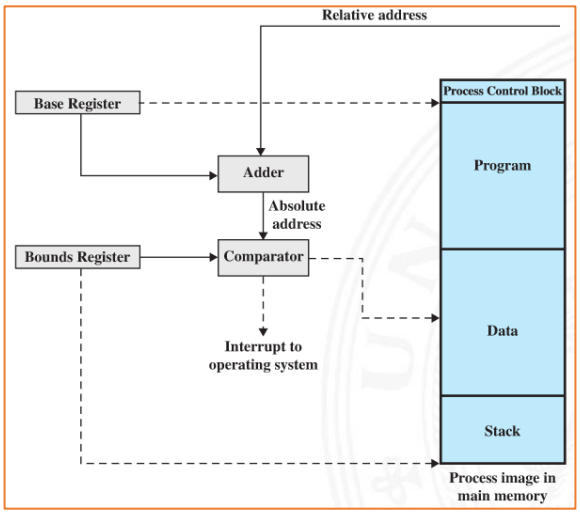
- Implementierung in Software (don’t do this)

- **Semaphor**
  - Ressource hat Integer der angibt, wie viele gleichzeitig zugreifen können
  - bei Zugriffsstart: Integer = Integer – 1
  - bei Zugriffsende: Integer = Integer + 1

- **Monitor**
  - In Programmiersprachen häufig implementiert
  - nur ein Prozess darf Monitor sein
  - mehr steht da nicht, ich schätze mal das ist nicht Klausurrelevant

Virtueller Speicher

- beim Programmieren immer die physikalischen Adressen zu wissen wäre zu aufwendig
- daher: logische Adressen in Programmen sind unabhängig von physikalischen Adressen
- Adressen im Code werden zu **virtuellen Adressen:**
  - virtuelle Adressen können über Abbildung (Maps) zu physikalischen Adressen übersetzt werden
  - bspw. Addierung von Offset



Begriffe

Frame Block fester Größe im Hauptspeicher
Page Block fester Größe im Sekundärspeicher
Segment Block variable Größe im Sekundärspeicher

- Pages können temporär in Hauptspeicher kopiert werden (=> Paging)
- Pages können an verschiedenen Stellen im Sekundärspeicher liegen
- Segmente können in Hauptspeicher geladen werden (=> Segmentierung)
- Segmente können in Seiten unterteilt werden, die dann jeweils in den Hauptspeicher geladen werden (=> Segmentierung + Paging)

Swapping Prozess auslagern, kann u.U. auch an anderer Stelle im Hauptspeicher fortgesetzt werden