

Inhaltsverzeichnis

1	Schaltungstechnik I	4
1.1	Boolesche Algebra / Schaltalgebra	4
1.1.1	Axiomensystem	4
1.1.2	Boolesche Funktionen	5
1.2	Gatterlogik	6
1.2.1	Implementierung	6
2	Schaltungstechnik II	7
2.1	Minimierung Boolescher Formeln	7
2.1.1	Äquivalenzumformungen	7
2.1.2	Systematische Vereinfachungsverfahren	7
2.1.3	Petricks Algorithmus	10
2.2	Programmierbare Logikarrays	11
2.2.1	Hardware-Implementation	12
3	Binärarithmetik & Ihre Implementierung	13
3.1	Erinnerung: Arithmetik	13
3.1.1	Zahlensysteme	13
3.1.2	Addition und Subtraktion	13
3.1.3	Multiplikation und Division	14
3.2	Binärearithmetik und Implementierung	14
3.3	Addition	14
3.3.1	Halbaddierer	14
3.3.2	Volladdierer	15
3.3.3	n-Bit-Addierer	15
3.3.4	Darstellung negativer Zahlen	17
3.4	Multiplikation	17
3.4.1	Multiplikation mit Potenzen der Basis	17
3.4.2	Standardalgorithmus	18
3.4.3	Negative Zahlen	18
3.4.4	Booths Algorithmus	19
3.5	Die Arithmetisch-Logische-Einheit (ALU)	20
3.5.1	Die ALU in der Hack-Architektur	20
4	Sequentielle Logik	21
4.1	Sequentielle Logik	21
4.1.1	Logikschaltungen: Kombinatorische & Sequentielle Logik	21
4.1.2	Prinzip der Rückkopplung	22

4.1.3	Asynchrone und synchrone Schaltwerke	22
4.1.4	Taktsignal (Clock Signal)	22
4.2	Bistabile Kippstufen	23
4.2.1	Implementierung mit Gattern	24
4.2.2	SR-Riegel	24
4.2.3	Direkt gesteuerte FlipFlops (Riegel)	26
4.2.4	Taktpegelsteuerung	27
4.2.5	Taktpegelsteuerung mit Rückkopplung	27
4.2.6	Taktflankensteuerung	28
4.2.7	Master-Slave-Prinzip	29
4.2.8	Schaltzeichen	29
4.2.9	Schaltverhalten	30
4.3	Register, Zähler und Speicher	30
4.3.1	Schieberegister und Zähler	30
5	Rechnerarchitektur	32
5.1	Speicherprogrammierung	32
5.1.1	Festverdrahtete "Prozessoren"	32
5.1.2	Konzept der Speicherprogrammierung	33
5.1.3	Befehlsaufruf, -dekodierung und -ausführung	34
5.1.4	Rechnerarchitekturen (Harvard & von Neumann)	34
6	Maschinensprache und Assembler	36
6.1	Die Hack-Maschinensprache	36
6.1.1	Einführung in die Maschinensprache	36
6.1.2	A-Anweisungen (Address Instructions)	37
6.1.3	C-Anweisungen (Compute Instructions)	38
6.2	Assembler und Assemblersprache	39
6.2.1	Physikalische und symbolische Programmierung	39
6.2.2	Maschinensprache und Assemblersprache	39
6.2.3	Die Hack-Assemblersprache	39
6.2.4	Symbole und Symbolverwaltung	41
6.2.5	Programmübersetzung und Assemblerimplementierung	41
7	Virtuelle Maschine	42
7.1	Hochsprachen und Übersetzung	42
7.1.1	Direkte und zweistufige Übersetzung	42
7.1.2	Vor- und Nachteile Virtueller Maschinen	43
7.1.3	Systembasierte und prozessbasierte virtuelle Maschinen	43
7.1.4	Übersetzungspfad	44
7.2	Virtuelle Maschine des Hack-Systems	44
7.2.1	Stapel(speicher) und ihre Operationen)	45
7.2.2	Stapelarithmetik	45
7.2.3	Arithmetische und Logische Operationen	45
7.2.4	Speicherzugriff, Speicheraufteilung und Speichersegmente	46
7.2.5	Programmablauf (bedingte Anweisungen und Schleifen)	47
8	Hochsprachen und Compiler	48
8.1	Die Programmiersprache Jack	48

8.1.1	Allgemeine Syntax	48
8.1.2	Datentypen und Speicheranforderung	48
8.1.3	Speicheranforderung	49
8.2	Compiler (speziell für Jack)	49
8.2.1	Architektur	49
8.2.2	Architektur, Lexikalische Analyse, Parsing	49
8.2.3	Kontextfreie Grammatiken	50
8.2.4	Parse-Bäume, Parsen durch rekursiven Abstieg	50
8.2.5	Jack-Syntaxanalyse	50
8.2.6	Codeerzeugung	50
9	Rechnernetze	51
9.1	Das Internet	51
9.1.1	Struktur des Internet	51
9.1.2	Leitungsvermittlung	51
9.1.3	Paketvermittlung	52
9.1.4	Warteschlangen	52
9.1.5	Übertragungskapazität	53
9.1.6	Verzögerungen	53
9.2	Abstraktionsschichten & Protokolle	54
9.2.1	Anwendungsschicht (Application Layer)	54
9.2.2	Transportschicht (Transport Layer)	55
9.2.3	Netzwerkschicht (Network Layer)	56
9.2.4	Verbindungsschicht (Data Link Layer)	56
9.2.5	Physische Schicht (Physical Layer)	56

Kapitel 1

Schaltungstechnik I

1.1 Boolesche Algebra / Schaltalgebra

DEFINITION: Boolesche Algebra

Eine *Boolesche Algebra* ist eine Menge B mit dem Nullelement 0 und dem Einselement 1 (d.h., $0, 1 \in B$), auf der die Operationen **Konjunktion** \wedge und **Disjunktion** \vee sowie **Negation** \neg definiert sind.

DEFINITION: Schaltalgebra

Eine *Schaltalgebra* ist eine Boolesche Algebra mit der Trägermenge $B = \{0, 1\}$.

1.1.1 Axiomensystem

George Boole (1847)

Kommutativität	$a \wedge b = b \wedge a$	$a \vee b = b \vee a$
Assoziativität	$(a \wedge b) \wedge c = a \wedge (b \wedge c)$	$(a \vee b) \vee c = a \vee (b \vee c)$
Idempotenz	$a \wedge a = a$	$a \vee a = a$
Distributivität	$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$	$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$
Neutralität	$a \wedge 1 = a$	$a \vee 0 = a$
Extremalität	$a \wedge 0 = 0$	$a \vee 1 = 1$
Involution	$\neg \neg a = a$	

De Morgan (1860)

De Morgan	$\neg(a \wedge b) = \neg a \vee \neg b$	$\neg(a \vee b) = \neg a \wedge \neg b$
Komplementarität	$a \wedge \neg a = 0$	$a \vee \neg a = 1$
Dualität	$\neg 0 = 1$	$\neg 1 = 0$
Absorption	$a \vee (a \wedge b) = a$	$a \wedge (a \vee b) = a$

1.1.2 Boolesche Funktionen

DEFINITION: Boolesche Funktionen

Eine Boolesche Funktion ist eine Funktion $f : \{0, 1\}^n \rightarrow \{0, 1\}$, $n \geq 0$. Die Anzahl n der Argumente der Funktion f heißt ihre Stelligkeit (arity).

Für kleine Stelligkeiten gibt es Spezielle Ausdrücke:

- $n = 1$: **unär** (unary)
- $n = 2$: **binär** (binary)
- $n = 3$: **ternär** (ternary)

Boolesche Funktionen werden durch **Schaltnetze** implementiert.

Jede Boolesche Funktion kann durch **Wahrheitstafeln** und **Formeln der Schaltalgebra** dargestellt werden.

Darstellung durch Wahrheitstafeln

- eine Spalte pro Funktionsargument,
- eine Zeile pro mögliche Wertekombination der Funktionsargumente,
- zusätzliche Spalte für den Funktionswert.

Abbildung 1.1: Wahrheitstafel einer ternären Booleschen Funktion

x_1	x_2	x_3	y
0	0	0	0
1	0	0	1
0	1	0	1
0	0	1	0
1	0	1	1
0	1	1	0
1	1	1	1

Darstellung durch Formeln der Schaltalgebra

Disjunktive Normalform

- Darstellung der Minterme
- Sum of Products (SOP)

Bilde *Disjunktion der Konjunktionen* aus Literalen aus jeder Zeile in der der Funktionswert 1 ist

Konjunktive Normalform

- Darstellung der Maxterme
- Product of Sums (POS)

Bilde *Konjunktion der Disjunktionen* aus Literalen aus jeder Zeile in der der Funktionswert 0 ist

Konjunktive und Disjunktive Normalform im Vergleich: Bevorzuge die disjunktive bei weniger Einsen, die konjunktive bei weniger Nullen in den Funktionswerten.

Vollständige Operationenmengen

DEFINITION: Vollständige Operationenmengen

Eine vollständige Operationenmenge ist eine Menge von Booleschen Operationen, die ausreicht, um alle Booleschen Funktionen darzustellen

Da sowohl die disjunktive als auch die konjunktive Normalform nur die Operationen Konjunktion, Disjunktion und Negation benutzt, ist die Menge $O = \{\vee, \wedge, \neg\}$ eine vollständige Operationenmenge.

Von einer anderen Operation O' kann man zeigen, dass sie vollständig ist, indem man die drei Operationen von O nur mit Hilfe der Operationen aus dieser Menge O' darstellt.

1.2 Gatterlogik

DEFINITION: (Logik-)Gatter

Ein Logikgatter ist eine Anordnung zur Realisierung einer booleschen Funktion, die binäre Eingangssignale zu einem binären Ausgangssignal verarbeitet. Die Eingangssignale werden durch Implementierung logischer Operatoren zu einem einzigen logischen Ergebnis umgewandelt.

1.2.1 Implementierung

Boolesche Funktionen werden mit Hilfe der Gatterlogik implementiert. Das heißt, mehrere elementare Gatter werden zusammengeschaltet, um komplexe Boolesche Funktionen zu berechnen:

Durch elektronische Bauteile, speziell (Feldeffekt-)Transistoren, werden spannungsgesteuerte Schalter dargestellt. aus mehreren Schaltern werden dann Gatter zusammengesetzt, die einfache Boolesche Funktionen darstellen. Die Gatter bilden jeweils eine vollständige Operationenmenge.

Kapitel 2

Schaltungstechnik II

2.1 Minimierung Boolescher Formeln

DEFINITION: Semantische & Syntaktische Äquivalenz

Seien φ und ψ Boolesche Formeln, dann gilt:

- Wenn beide Formeln für alle Belegungen den gleichen Wahrheitswert haben, dann sind die Formeln

Semantische äquivalent: $\varphi \equiv \psi$

Wenn φ durch Äquivalenzumformungen in ψ umgeformt werden kann, dann sind die Formeln

Syntaktische äquivalent: $\varphi = \psi$

Für die Syntaktische Äquivalenz ist der Nachweis oft wesentlich kürzer. Ein Abschluss der Äquivalenzprüfung ist allerdings nicht garantiert (kein Weg gefunden $\nrightarrow \varphi \equiv \psi$). Für die Semantische Äquivalenz kann der Nachweis sehr aufwendig sein. Zum Falsifizieren wird jedoch lediglich eine Wertkombination benötigt, die nicht äquivalent ist.

2.1.1 Äquivalenzumformungen

Die Axiome der Booleschen Algebra erlauben es, alle semantische geltenden Äquivalenten auf syntaktischem Wege abzuleiten, denn es gilt:

Zwei Boolesche Formeln sind **genau dann** semantisch äquivalent, wenn sie syntaktisch äquivalent sind.

2.1.2 Systematische Vereinfachungsverfahren

Das Problem Äquivalenzumformungen ist, dass es keine klare Strategie zur Vereinfachung gibt. Besser wäre ein systematisches Vereinfachungsverfahren.

Karnaugh-Veitch-Diagramme

DEFINITION: Karnaugh-Veitch-Diagramme

Tabellarische Darstellungen Boolescher Funktion (wie Wahrheitstafeln, nur andere Auflistung der Funktionswerte).

- 2^n Felder für n Argumente.
- Anordnung, dass ein Übergang zu einem Nachbarfeld den Wert nur genau einer der Variablen ändert (ein sog. Gray-Code)

Sind zwei benachbarte Felder eines Karnaugh-Veitch-Diagramms beide 1, so zeigt dies an, dass eine bestimmte Variable in der ursprünglichen Funktion irrelevant ist. Beim Zusammenfassen von Feldern darf auch der Rand überschritten werden, da sich auch in diesem Fall der Wert nur einer Variable ändert: Wie oben gezeigt können auch mehr als 2 Felder Zusammengefasst werden, jedoch nur, wenn die Felderzahlen Zweierpotenzen sind.

1. **Schritt:** Finde alle maximalen Zusammenfassungen von Feldern

(dürfen überlappen, aber nicht in größeren Zusammenfassungen vorkommen)

	AB			
	00	01	11	10
8	0	0	1	1
9	0	0	1	1
10	0	0	0	1
11	0	1	1	1

	AB			
	00	01	11	10
8	0	0	1	1
9	0	0	1	1
10	0	0	0	1
11	0	1	1	1

	AB			
	00	01	11	10
8	0	0	1	1
9	0	0	1	1
10	0	0	0	1
11	0	1	1	1

	AB			
	00	01	11	10
8	0	0	1	1
9	0	0	1	1
10	0	0	0	1
11	0	1	1	1

2. **Schritt:** Wähle möglichst wenige Zusammenfassungen, die alle Einsen abdecken

	AB			
	00	01	11	10
8	0	0	1	1
9	0	0	1	1
10	0	0	0	1
11	0	1	1	1

	AB			
	00	01	11	10
8	0	0	1	1
9	0	0	1	1
10	0	0	0	1
11	0	1	1	1

	AB			
	00	01	11	10
8	0	0	1	1
9	0	0	1	1
10	0	0	0	1
11	0	1	1	1

3. **Schritt:** Sammele die benötigten Ausdrücke:

	AB			
	00	01	11	10
8	0	0	1	1
9	0	0	1	1
10	0	0	0	1
11	0	1	1	1

$$(A \wedge \bar{C}) \vee (A \wedge B) \vee (B \wedge C \wedge \bar{D})$$

Quine-McCuskey-Algorithmus

Eine Minimierung logischer Funktionen mit bis zu sechs Argumenten ist zwar mit erweiterten Karnaugh-Veitch-Diagrammen im Prinzip möglich, aber unhandlich. Ein besserer Weg besteht in der Verwendung eines anderen Verfahrens des **Quine-McClunskey-Algorithmus**.

1. **Schritt:** Bilde die disjunktive (analog auch konjunktive) Normalform der zu minimierenden Funktion.
2. **Schritt:** Finden der **Primimplikanten** (Abb. 2.1).
 - Vereinige Terme, in denen eine einzelne Variable in dem einen Term als positives, im anderen als negatives Literal auftritt. (Der gleiche Term darf für mehrere Vereinigungen verwendet werden.)

$$(A_1 \dots A_i \dots A_n) \vee (A_1 \dots \overline{A_i} \dots A_n) = A_1 \dots A_{i-1} A_{i+1} \dots A_n$$

- Wiederhole dies Rekursiv mit den Vereinigungsergebnissen, bis keine weiteren Vereinigungen mehr möglich sind.
- Vernachlässige alle Terme, die mit anderen Vereinigt wurden.
- Übrig bleiben die sogenannten **Primimplikanten**

Abbildung 2.1: Tabelle zur bestimmung von Primimplikanten

1er Implikanten			
1	$\overline{A}\overline{B}\overline{C}\overline{D}$	0100	×
2	$\overline{A}\overline{B}\overline{C}D$	1000	×
3	$\overline{A}\overline{B}C\overline{D}$	1001	×
4	$\overline{A}\overline{B}CD$	1010	×
5	$\overline{A}B\overline{C}\overline{D}$	1011	×
6	$\overline{A}B\overline{C}D$	1100	×
7	$\overline{A}BC\overline{D}$	1110	×
8	$\overline{A}BCD$	1111	×

Die 1er Implikanten
sind die Minterme.

2er Implikanten			
1+6 → 9	$\overline{B}\overline{C}\overline{D}$	-100	P_1
2+3 → 10	$\overline{A}\overline{B}\overline{C}$	100-	×
2+4 → 11	$\overline{A}\overline{B}D$	10-0	×
2+6 → 12	$\overline{A}\overline{C}\overline{D}$	1-00	×
3+5 → 13	$\overline{A}\overline{B}D$	10-1	×
4+5 → 14	$\overline{A}\overline{B}C$	101-	×
4+7 → 15	$\overline{A}C\overline{D}$	1-10	×
5+8 → 16	$\overline{A}CD$	1-11	×
6+7 → 17	$\overline{A}B\overline{D}$	11-0	×
7+8 → 18	$\overline{A}BC$	111-	×

4er Implikanten			
10+14, 11+13 → 19	$\overline{A}\overline{B}$	10--	P_2
11+17, 12+15 → 20	$\overline{A}\overline{D}$	1--0	P_3
14+18, 15+16 → 21	$\overline{A}C$	1-1-	P_4

Primimplikanten
(unvereinigte Implikanten)
sind mit P_i bezeichnet.

3. **Schritt:** Aufstellen und Auswerten der Primimplikantentabelle

- Spalten: Minterme, Zeilen: Primimplikanten
- Finde wesentliche Primimplikanten (aufsuchen der Spalten mit nur einer Markierung)

Eine systematische Methode für diese Auswahl von Primimplikanten ist **Petricks Algorithmus**

Primimplikanten			Minterme (Konjunktionen der disjunktiven NF)							
			0100	1000	1001	1010	1011	1100	1110	1111
P_1	-100	*	•					•		
P_2	10--	*		•	•	•	•			
P_3	1--0			•		•		•	•	
P_4	1-1-	*				•	•		•	•

- Abdeckung nur durch einen Primimplikanten → wesentlich
- Abdeckung auch durch wesentliche Primimplikanten.
- Nicht benötigte Abdeckungen

INFO:

Primimplikanten des Quine-McCluskey-Algorithmus entsprechen Feld-Zusammenfassungen in Karnaugh-Veitch-Diagrammen. Dementsprechend gibt es auch wesentliche Zusammenfassungen in Karnaugh-Veitch-Diagrammen. (Feldergruppen, die von keiner der anderen Gruppen abgedeckt werden)

Grün entspricht dem 4er Primimplikanten 10–
Rot entspricht dem 4er Primimplikanten -110
Blau entspricht dem 2er Primimplikanten -110
Gelb entspricht dem 4er Primimplikanten 1–0

		AB			
CD		00	01	11	10
	00	0	0	1	1
	01	0	0	1	1
	11	0	0	0	1
	10	0	1	1	1

2.1.3 Petricks Algorithmus

Nicht immer decken die wesentlichen Primimplikanten alle Minterme ab. In diesem Fall müssen aus den restlichen Primimplikanten geeignete ausgewählt werden, um die verbleibenden Minterme abzudecken.

1. **Schritt:** Bilde eine reduzierte Primimplikantentabelle

Diese enthält nur noch die noch nicht abgedeckten Minterme und die nicht wesentlichen Primimplikanten.

2. **Schritt:** Ordne jedem Primimplikanten eine Auswahlvariable P_i zu.

3. **Schritt:** Bilde für jeden Minterm (Spalte) die Disjunktion der Auswahlvariablen.

Alle Primimplikanten, die diesen Minterm abdecken werden mit einer Disjunktion verknüpft.

4. **Schritt:** Verknüpfe alle Disjunktionen zu einer Konjunktion C .

5. **Schritt:** Wandle Konjunktion C durch die Distributivgesetze in eine Disjunktion D' um

Nun ergibt sich eine Disjunktion aus Konjunktionen, die jeweils alle Minterme abdecken.

$$(P_1 \wedge P_2 \wedge P_3) \vee (P_2 \wedge P_3 \wedge P_4) \vee (P_3 \wedge P_5)$$

6. **Schritt:** Wähle die Konjunktionen aus D' , mit den wenigsten Primimplikanten

$$P_3 \wedge P_5$$

		Minterme (Konjunktionen)					
Primimplikanten		000	001	010	101	110	111
P_1	00-	•	•				
P_2	0-0	•		•			
P_3	-01		•		•		
P_4	-10			•		•	
P_5	1-1				•		•
P_6	11-					•	•

$$\begin{aligned} &(P_1 \vee P_2) \text{ (001)} \\ &\wedge (P_1 \vee P_3) \text{ (001)} \\ &\wedge (P_2 \vee P_4) \text{ (010)} \\ &\wedge (P_3 \vee P_5) \text{ (101)} \\ &\wedge (P_4 \vee P_6) \text{ (110)} \\ &\wedge (P_5 \vee P_6) \text{ (111)} \end{aligned}$$

2.2 Programmierbare Logikarrays

Alle betrachteten Minimierungsergebnisse haben die folgende allgemeine Form:

$$o = (i_1 \wedge \bar{i}_2 \wedge \bar{i}_3 \wedge \dots) \vee (\bar{i}_1 \wedge i_2 \wedge \bar{i}_3 \wedge \dots) \vee (\bar{i}_1 \wedge \bar{i}_2 \wedge i_3 \wedge \dots) \vee \dots$$

Alle Boolesche Formeln können in diese Form gebracht werden, denn es handelt sich um eine Disjunktion von Konjunktionen von Literalen (sum of products, SOP)

DEFINITION: Programmierbare Logikarrays

Programmierbare Logikarrays (programmable logic array, PLAs) sind solche Formeln realisiert in Hardware".

- Alle Eingaben i_k und ihre Negation \bar{i}_k sind verfügbar
- Die Eingaben werden über AND-Gatter verknüpft
- Die Ausgaben der AND-Gatter werden durch OR-Gatter verknüpft
- Ein PLA wird durch Entfernen von Verbindungen "konfiguriert" (programmiert)

Gatterimplementierung von Originalfunktion und disjunkt. Normalform

Abbildung 2.2: Mehr Gatter, aber standardisierte Struktur

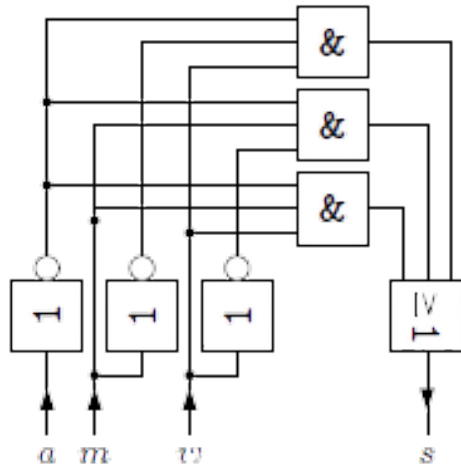
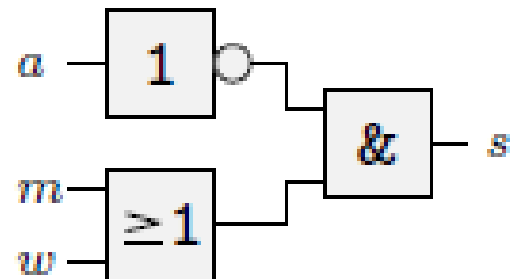
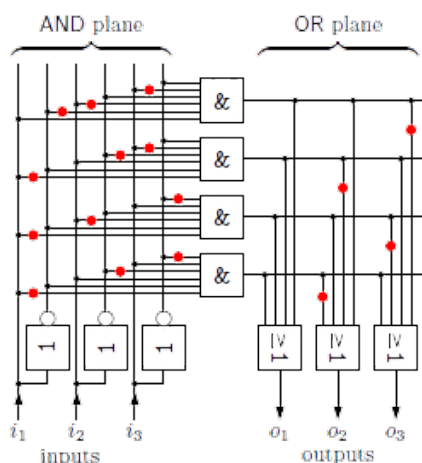


Abbildung 2.3: Weniger Gatter, Struktur abhängig von Funktion



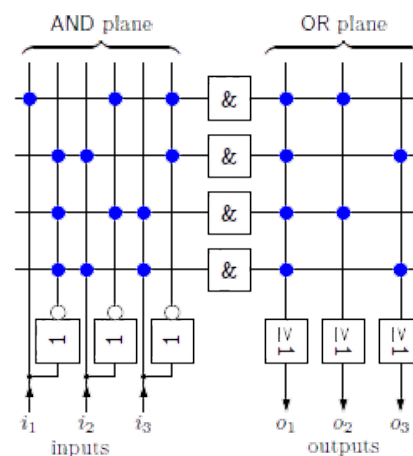
2.2.1 Hardware-Implementation

Abbildung 2.4: Originale Darstellung von Schaltungen



Bei Auslieferung sind Logikarrays komplett verbunden (an jedem UND-Gatter liegen alle negierte und unnegierte Eingaben an). Beim programmieren des Logikarrays werden die rot markierten Verbindungen getrennt

Abbildung 2.5: Vereinfachte Darstellung von Schaltungen



Zur Vereinfachung stellt man nur eine Linie je Gatter dar und markiert die **Verbindungen**

Für eine elektronische Implementierung (durch Transistoren) sind AND- und OR-Gatter nicht besonders gut geeignet. Besser geeignet sind NAND- und NOR-Gatter (und ggf. NOT-Gatter).

Kapitel 3

Binärarithmetik & Ihre Implementierung

3.1 Erinnerung: Arithmetik

3.1.1 Zahlensysteme

Im prinzip kann jede beliebige Zahl als Basis eines Zahlensystems gewählt werden. Das in der Rechnertechnik verwendete *Binärsystem* (Basis 2) hat den Vorteil der kleinstmöglichen Zahl an Ziffernzeichen, nämlich nur zwei:

0, 1

Das auch häufig verwendete *Hexadezimalsystem* (Basis 16):

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

3.1.2 Addition und Subtraktion

Addition und Subtraktion kann sehr leicht stellenweise in einem Stellensystem ausgeführt werden.

Einziges Problem ist die Behandlung von *Stellenüberlauf* und *Stellenunterlauf*. In diesen Fällen entsteht ein *Übertrag*.

Diese Rechenschemata sind nicht nur im Dezimalsystem, sondern im Prinzip in jedem Zahlensystem anwendbar.

$$\begin{array}{r} 4 \ 6 \ 7 \ 8 \ 5 \ 3 \ 9 \ 9 \\ + \ 2_{\text{1}} \ 8 \ 0_{\text{1}} \ 7 \ 1 \ 2_{\text{1}} \ 3_{\text{1}} \ 4 \\ \hline = \ 7 \ 4 \ 8 \ 5 \ 6 \ 6 \ 3 \ 3 \\ 4 \ 6 \ 7 \ 8 \ 5 \ 3 \ 9 \ 9 \\ - \ 2_{\text{1}} \ 8 \ 0_{\text{1}} \ 7 \ 1 \ 2_{\text{1}} \ 3_{\text{1}} \ 4 \\ \hline = \ 4 \ 6 \ 7 \ 8 \ 5 \ 3 \ 9 \ 9 \end{array}$$

3.1.3 Multiplikation und Division

Die Multiplikation wird auf eine Summe von Stellenprodukten zurückgeführt.

Hierin besteht das Hauptproblem darin, den richtigen Stellenfaktor zu bestimmen.

Meist wird er geschätzt, anschließend ausprobiert und gegebenenfalls die Schätzung korrigiert.

Abbildung 3.1: Beispiel von Multiplikation

					4	6	7	8	5	3	9	9	
×									9	6	4	3	1
					4	6	7	8	5	3	9	9	
+			1	4	0	3	5	6	1	9	7		3
+		1	8	7	1	4	1	5	9	6			4
+	2	8	0	7	1	2	3	9	4				6
+	4	2	1	0	6	8	4	9	1				9
=	4	5	1	1	5	6	2	8	1	0	9	6	9

3.2 Binärearithmetik und Implementierung

Die beiden Ziffern des Binärsystems können direkt durch Schalter Implementiert werden (0: offen (falsch), 1: geschlossen (wahr))

3.3 Addition

Die Addition ist die einfachste und am häufigsten verwendete Operation in der arithmetisch-logischen Einheit, da sie sich direkt in Wahrheitstafeln übersetzen lassen:

An der Addition von zwei Bits mit Wert 1 zeigt, warum wir für den Ein-Bit-Addierer zwei Ausgänge benötigen:

- Die *Summe* (sum, s)
- Den *Übertrag* (carry, c)

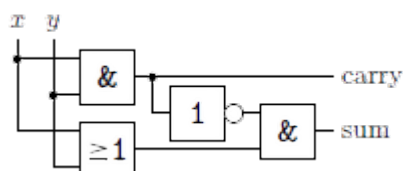
Die bitweise Addition kann offenbar auch durch (Logik-)gatter implementiert werden.

3.3.1 Halbaddierer

Wegen der möglichkeit eines Übertrags benötigt man für die Addition nicht nur eine Boolesche Funktion, sondern zwei (wobei die zweite den Übertrag bestimmt)

$$c = x \wedge y \quad s = (x \wedge \bar{y}) \vee (y \wedge \bar{x})$$

Abbildung 3.2: Halbaddierer



Verknüpfung der beiden Funktionen erlaubt eine Implementierung mit weniger Gattern, weiteres Ausnutzen der Booleschen Gesetze zur Vereinfachung und durch Nutzung anderer, zusammengesetzter Gatter ergibt sich dann eine deutlich kleinere Schaltung:

$$s = (x \wedge \bar{y}) \vee (\bar{x} \wedge y) = x \oplus y$$

3.3.2 Volladdierer

Um Addition nicht nur für ein Bit, sondern für n Bits, $n \geq 2$, zu implementieren, braucht man einen *Addierer mit drei Eingängen*, um den Übertrag der vorangehenden Addition zu berücksichtigen. Ein Volladdierer berücksichtigt den Übertrag einer vorangehenden Addition

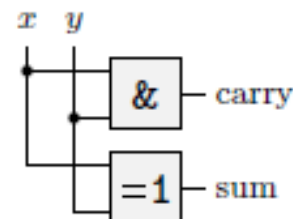


Abbildung 3.3: Optimierter Halbaddierer

x	y	c_{in}	s	c_{out}
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

Ein Volladdierer berechnet die Funktion

$$s = (x + y) + x_{in}$$

Aus diesem Grund wird er am einfachsten aus zwei Halbaddierern zusammengesetzt (mit $z = x_{in}$)

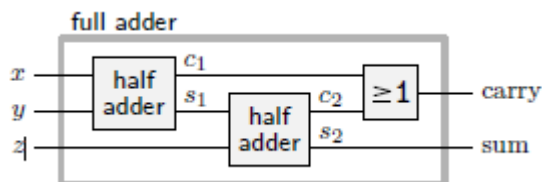


Abbildung 3.4: Volladdierer

3.3.3 n-Bit-Addierer

n-Bit-Übertragungskette-Addierer

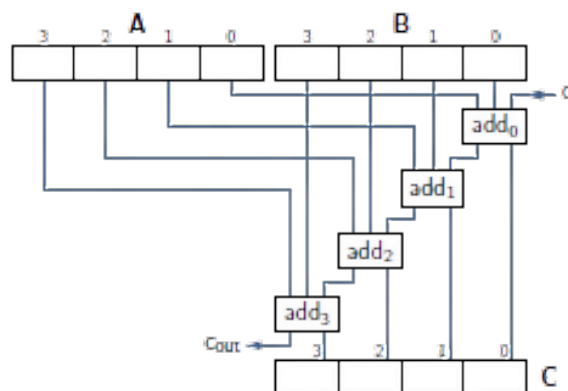
DEFINITION: Übertragungskette-Addierer

In einem n -Bit-Übertragungskette-Addierer wird mithilfe von n Volladdierern Addition durchgeführt

Probleme:

- Übertrag breitet sich wellenartig durch die Addiererkette aus
- Volladdierer add_k kann erst anfangen, wenn add_{k-1} seine Berechnung abgeschlossen hat

Abbildung 3.5: Übertragungskette-Addierer



n -Bit-Übertragsauswahl-Addierer

DEFINITION: Übertragsauswahl-Addierer

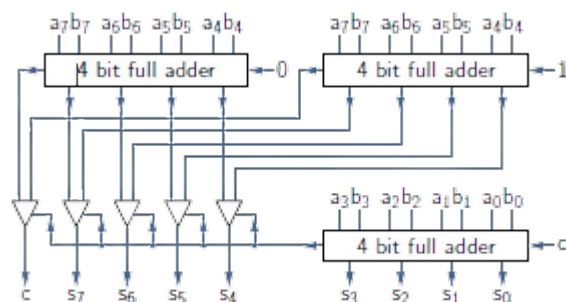
In einem n -Bit-Übertragsauswahl-Addierer werden die Summen des unteren Halbwortes (Bits 0 bis $\frac{n}{2} - 1$) und des oberen Halbwortes (Bits $\frac{n}{2} - 1$ bis n) parallel berechnet. Dadurch umgeht man die Problematik des wellenartigen Übertrags.

Da der Wert des Übertrags aus dem unteren Halbwort noch nicht bekannt ist, wenn die Summenbildung für das obere Halbwort beginnt, wird diese Summe zweimal, in getrennten Schaltungen berechnet:

- Schaltung 1: $c_{in} = 0$
- Schaltung 2: $c_{in} = 1$

Wenn der Übertrag des unteren Halbworts berechnet ist, wird er über einen Multiplexer zur Auswahl des richtigen oberen Halbworts benutzt

Abbildung 3.6: Übertragsauswahl-Addierer



INFO:

Bei längeren Binärzahlen wird das Prinzip des Übertragsauswahl-Addierers rekursiv angewandt (d.h., die Halbwörter werden ihrerseits zerlegt)

3.3.4 Darstellung negativer Zahlen

Betrag & Vorzeichen

Höchstwertiges Bit gibt
Vorzeichen an

Problem: Zahlen sollten
gleiche Stellenzahl aufweisen

Einerkomplement

Negation durch Inversion der
Zahl

Problem: Übertrag, Zwei
Darstellungen für Null

Zweierkomplement

Negation durch Inversion und
Addition von 1

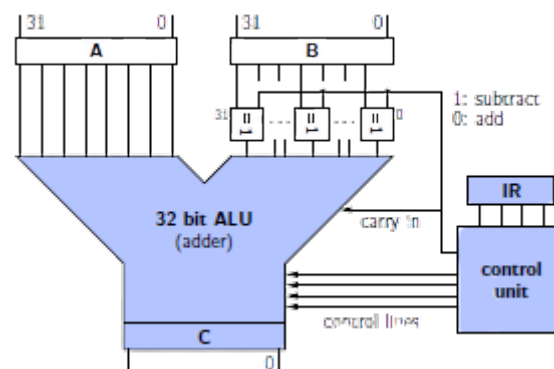
n-Bit-Addierer & -Subtrahierer im Zweierkomplement

DEFINITION: Addierer & Subtrahierer im 2er-Komplement

n -Bit-Subtrahierer bestehen aus eine, n -Bit-Addierer und Gattern, die die Negationsregel implementieren.

Idee: $A - B = A + (-B)$

Abbildung 3.7: Beispiel eines n -Bit-Subtrahierers



3.4 Multiplikation

3.4.1 Multiplikation mit Potenzen der Basis

Multiplikation mit einer Potenz der Basis b des Zahlensystems ist einfach: Eine Multiplikation b^k verschiebt die Ziffern um k Stellen nach links. Die freiwerdenden niederwertigen Stellen werden auf 0 gesetzt.

Division durch eine Potenz der Basis b ist analog zur Multiplikation: Eine Division durch b^k verschiebt die Ziffern um k Stellen nach rechts. Dies liefert allerdings nur den ganzzahligen Teil der Division

$$\begin{aligned} [d_{n-1} \dots d_0] \cdot b^k &= [d_{n-1} \dots d_0 \ 0_1 \dots 0_k]_b \\ 482_{10} \cdot 10^2 &= 48200_{10} \\ 10101_2 \cdot 2^3 &= 10101000_2 \end{aligned}$$

$$\begin{aligned} [d_{n-1} \dots d_0] \div b^k &= [d_{n-1} \dots d_k]_b \\ 482_{10} \div 10^2 &= 4_{10} \\ 10101_2 \div 2^3 &= 10_2 \end{aligned}$$

Der Rest einer Division (modulo) durch b^k sind die letzten k Stellen der Zahl:

$$\begin{aligned} [d_{n-1} \dots d_0] \bmod b^k &= [d_{k-1} \dots d_0]_b \\ 482_{10} \bmod 10^2 &= 82_{10} \\ 10101_2 \bmod 2^3 &= 101_2 \end{aligned}$$

INFO:

Man beachte, dass die Division durch $b1k$ äquivalent zur Multiplikation mit b^{-k} ist.

3.4.2 Standardalgorithmus

Für die allgemeine binäre Multiplikation wird der Grundschulalgorithmus des Addierens von Stellenprodukten im Binärsystem angewandt. Die Stellenprodukte werden durch Bit-Schieben berechnet

Ist die k -te Stelle des Factors A eins ($A_k = 1$), so wird der Factor B , multipliziert mit dem Stellenwert 2^k auf das Ergebnis auf-addiert.

Die Multiplikation mit 2^k wird durch Bit-Schieben erreicht.

Somit wird Faktor A in eine Summe von Zweierpotenzen zerlegt:

$$A_2 = a_3 \cdot 2^3 + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0$$

		1	1	0	1				$(13)_{10}$
\times		1	0	1	0				$(10)_{10}$
		1	0	1	0			1	$(10)_{10}$
+		0	0	0	0			0	$(0)_{10}$
+		1	0	1	0			1	$(40)_{10}$
+		1	0	1	0			1	$(80)_{10}$
=	1	0	0	0	0	0	1	0	$(130)_{10}$

Beachte: Allgemein liefert die Multiplikation zweier n -Bit-Uahlen ein $2n$ -Bit-Ergebnis

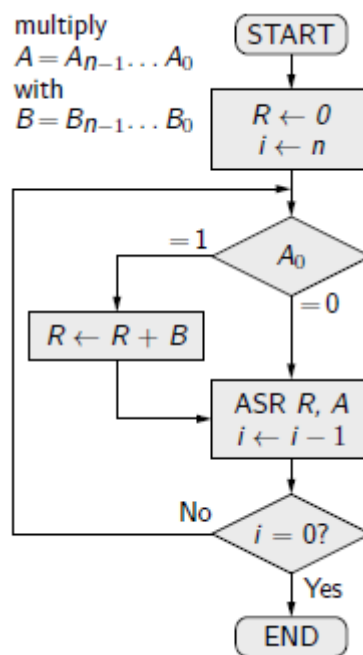
3.4.3 Negative Zahlen

Um mit den Standardalgorithmus auch Negative Zahlen multiplizieren zu können, müssen die Faktoren auf 8-Bit erweitert werden (aktuell 4-Bit).

Um eine im Zweierkomplement interpretierte Zahl von n auf m Bit ($m > n$) zu erweitern, müssen höherwertige Stellen passend aufgefüllt werden:

Die neuen $m - n$ Stellen bekommen den Wert des Vorzeichenbits:

Abbildung 3.8: Standardalgorithmus



$$-4_{10} = [1\ 011]_{2k} = [1111\ 1011]_{2k} = -5_{10}$$

$$[a_{n-1}a_{n-2}a_{n-3}a_{n-4}]_{2k} = [a_{n-1}a_{n-1}a_{n-1}a_{n-1}\ a_{n-1}\dots a_{n-4}]_{2k}$$

Leider reicht die Stellenerweiterung alleine nicht immer aus, daher ist es angebracht eine andere Methode zu suchen, die mit negativen Zahlen besser umgehen kann: *Booths Algorithmus*

3.4.4 Booths Algorithmus

DEFINITION: Booths Algorithmus

Die Kernidee von Booths Algorithmus ist es, den ersten Faktor im Produkt $a \cdot b$ in der Form:

$$a = (a_1 - a_2) + (a_3 - a_4) + \dots + (a_{k-1} - a_k)$$

zu zerlegen, so dass die Multiplikation $a \cdot b$ zu:

$$a \cdot b = a_1b - a_2b + a_3b - a_4b + \dots + a_{k-1}b - a_kb$$

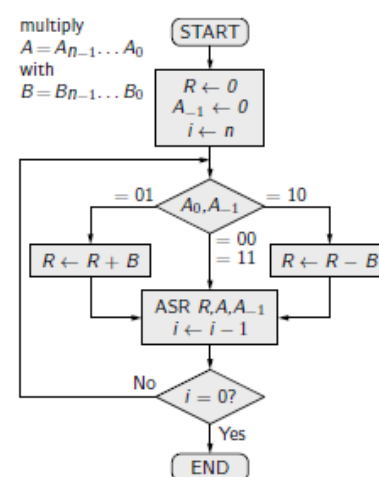
Dies ist nützlich, da eine Differenz von Zweierpotenzen einer Binärzahl mit genau einer Kette von Einsen entspricht. Dementsprechend benötigt der Algorithmus nur so viele Additionen, wie es Wechsel zwischen 1 und 0 (und umgekehrt 0 und 1) im Faktor A gibt.

$$7_{10} = 00000111_2 = 1000_2 - 0001_2 = 2^3_{10} - 2^0_{10}$$

	0	0	0	0	0	1	1	1		$(+7)_{10}$	B
\times	1	1	1	1	0	0	0	0		$(-5)_{10}$	A
+	0	0	0	0	0	1	0	1	1	$(5)_{10}$	$-2^0 \cdot B$
+	0	0	0	0	0	0	0	0	1	$(0)_{10}$	
+	0	0		0	0	0	0	0	1	$(0)_{10}$	
+	1	1	0	1	1	0	0	0	1	$(-40)_{10}$	$2^3 \cdot B$
=	1	1	0	1	1	1	0	1		$(-35)_{10}$	$2^3 \cdot B - 2^0 \cdot B$

Wir haben hier 7_{10} in $7_{10} = 8_{10} - 1_{10} = 2^3_{10} - 2^0_{10} = 1000_2 - 0001_2$ zerlegt

Abbildung 3.9: Booths Algorithmus



Negativer erster Faktor: Im Fall, dass der erste Faktor negativ ist, es also eine 1 als Vorzeichen gibt, dann können die führenden einsen ignoriert werden:

$$\begin{aligned}
 11111011_2 &= 11111111_2 + (10000000_2 - 1000_2) + (0100_2 - 0001_2) \\
 &= -2_{10}^7 + (2_{10}^7 - 2_{10}^3) + (2_{10}^2 - 2_{10}^0) \\
 &= \cancel{-2_{10}^7} + (\cancel{2_{10}^7} - 2_{10}^3) + (2_{10}^2 - 2_{10}^0) \\
 &= -2_{10}^3 + 2_{10}^2 - 2_{10}^0
 \end{aligned}$$

3.5 Die Arithmetisch-Logische-Einheit (ALU)

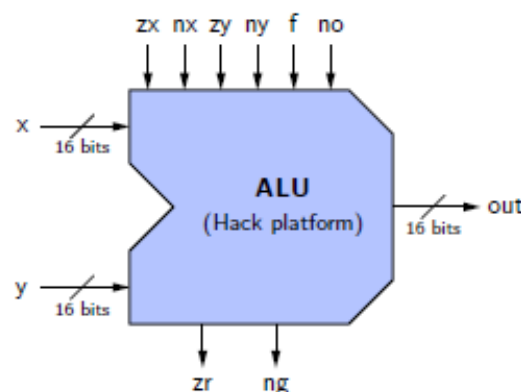
3.5.1 Die ALU in der Hack-Architektur

Die Eingaben zx , nx , ny , f und no kodieren die auszuführende Operation:

- zx Setze Eingabe $x = 0$
- zy Setze Eingabe $y = 0$
- nx Bilde Einerkomplement der Eingabe x
- ny Bilde Einerkomplement der Eingabe y
- f Wählt Operation: Addition / bitweises Und
- no Bilde Einerkomplement der Ausgabe out
- zr Zero Flag ($zr = 1 \rightarrow out = 0$)
- ng Negative Flag ($ng = 1 \rightarrow out < 0$)

Über-/Unterlauf wird ignoriert

Abbildung 3.10: ALU - Hack (16 Bit)



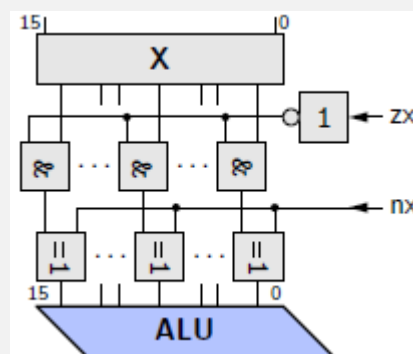
Durch die 6 Steuerbits (Abb. 3.10) kann im Prinzip zwischen $2^6 = 62$ Wahrheitstafeln ausgewählt werden, von diesen sind allerdings nur 18 relevant.

DEFINITION:

Die Steuerbits dienen als Eingaben für Steuergatter, die zugeordneten Operationen bewirken.

Diese Steuergatter können, wie in Abb. 3.11 die Eingabe oder, wie im Fall von no die Ausgabe verändern.

Abbildung 3.11: CAPTION

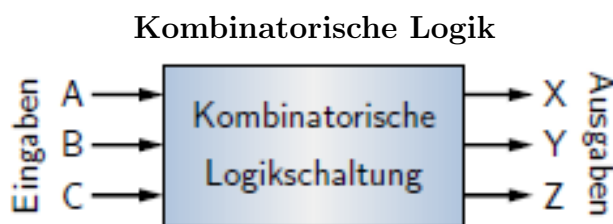


Kapitel 4

Sequentielle Logik

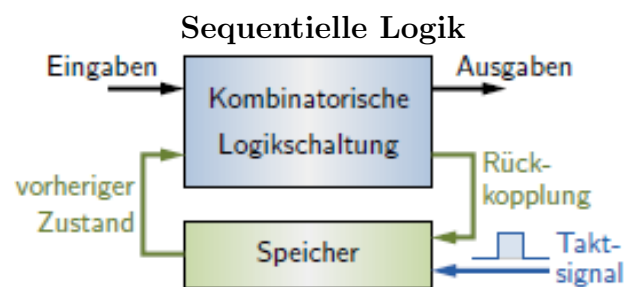
4.1 Sequentielle Logik

4.1.1 Logikschaltungen: Kombinatorische & Sequentielle Logik



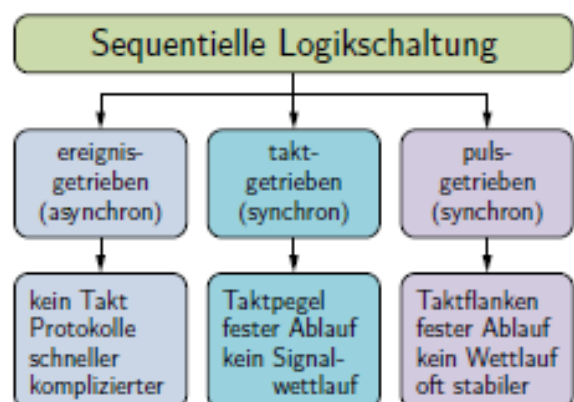
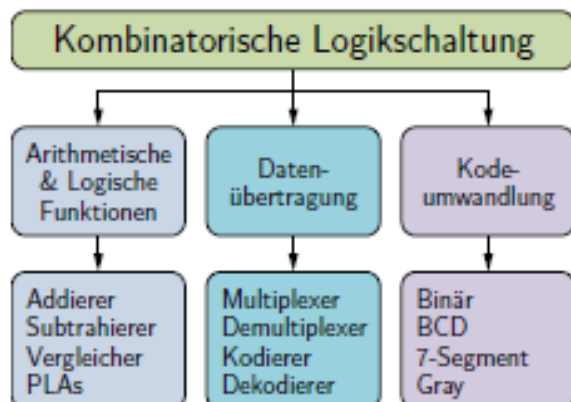
Implementierung: Schaltnetze

- Einfaches Berechnen von Ein- & Ausgaben
- Keine Informationsspeicherung
- Zustandslosigkeit
- verzögerungsfreie Berechnung



Implementierung: Schaltwerke

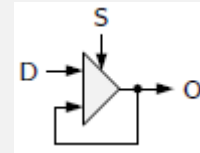
- Rückkopplung von Ausgaben auf Eingaben
- Explizite Informationsspeicherung
- Unterscheidung von Zuständen
- Gatterlaufzeiten explizit berücksichtigt



4.1.2 Prinzip der Rückkopplung

DEFINITION: Rückkopplung

Durch Rückkopplung werden Ausgaben als Eingabesignal verwendet. Hierdurch kann ein (Ausgabe-) Zustand festgehalten werden, bis ein Ereignis ihn wieder ändert.



Probleme der Rückkopplung Durch Rückkopplung kann es zu (unerwünschten) *Schwingungen* kommen. Diese sind ein Beispiel für *Signallaufzeitprobleme*, die durch Rückkopplungen auftreten können).

Ein weiteres Beispiel sind sogenannte *Wettlaufsituationen* (*Race Conditions*), die auftreten, wenn sich zwei Signale auf zwei oder mehr Wegen ausbreiten, und die Ausgabe davon abhängen kann, welches Signal schneller weitergegeben wird

Signallaufzeitprobleme lassen sich am leichtesten durch ein zentral erzeugtes Taktsignal vermeiden, das bestimmt, wann Eingaben übernommen werden.

4.1.3 Asynchrone und synchrone Schaltwerke

Asynchrone Schaltwerke

- Direkte Steuerung durch Änderung der Eingangssignale
- Wann und ob ein stabiler Zustand erreicht wird von Gatterlaufzeit abhängig
- Oft komplizierter, aufwendiger Entwurf
- Sehr schnelle Schaltwerke möglich

Synchrone Schaltwerke

- Steuerung durch Taktsignal
- Eingangssignale nur zu von Takt festgelegten Zeitpunkten übernommen
- Meist einfacher, systematischer Entwurf.
- Langsamstes Bauteil bestimmt maximal mögliche Taktfrequenz

4.1.4 Taktsignal (Clock Signal)

DEFINITION:

Ein *Taktsignal* (Clock Signal) wird meist durch einen *Quarzoszillator* erzeugt. Durch elektromagnetische Resonanz eines piezoelektrischen Quarzkristalls (Schwingquarz) entsteht ein Taktsignal mit fester Frequenz.

Taktzyklus / -periode (Cycle / Period) T :

Zeit zwischen steigenden/fallenden Flanke und der nächsten

Taktfrequenz (Frequency) f :

Reziprokwert der Taktperiode T

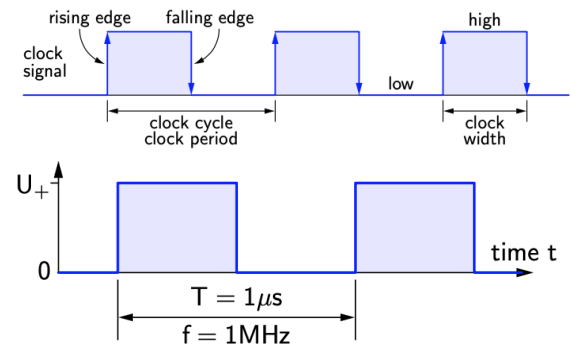
Taktbreite (Width) :

Die Zeit zwischen einer steigenden und fallenden Flanke (kann aber muss nicht die Hälfte der Taktperiode sein)

Taktpegel / Taktzustand :

1-Pegel (Versorgungsspannung, high) und 0-Pegel (Masse, low) - Unterscheidung der Phasen im Taktzyklus

Abbildung 4.1: Idealisierend: Taktsignal als Folge von Rechteckimpulsen



INFO:

- Oft ist die Reihenfolge von Berechnungen wichtig (manche können parallel, andere müssen strikt sequentiell abgearbeitet werden)
→ Das Taktsignal gilt als Zeitgeber für solche Berechnungen.
- Typische Taktfrequenzen liegen im zwischen einigen KiloHertz (kHz) und einigen GigaHertz (GHz)
- Manchmal werden asymmetrische Taktsignale benötigt, die durch Verzögerung (delay) und logische Und-Verknüpfung mit dem Original erzeugt werden

Abbildung 4.2: Erzeugung eines verzögerten Taktsignals

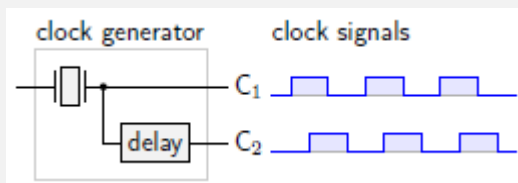
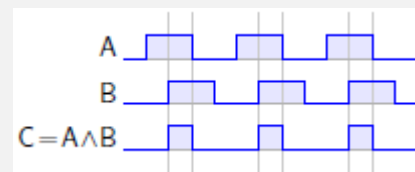


Abbildung 4.3: Asym. Taktsignal durch Und-Verknüpfung



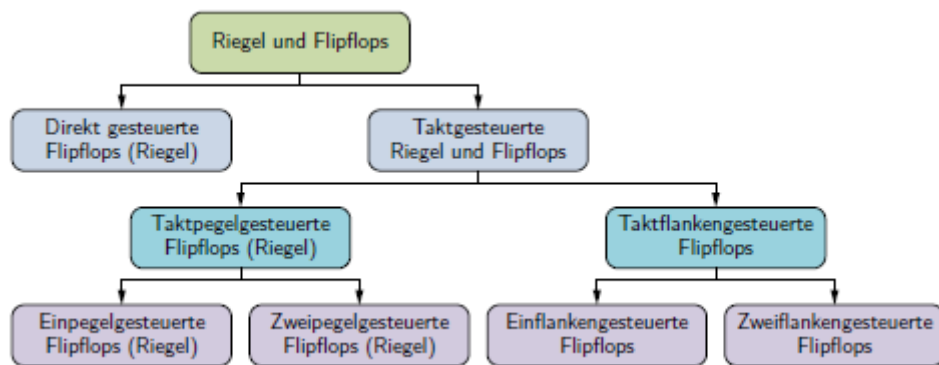
4.2 Bistabile Kippstufen

DEFINITION:

Eine *bistabile Kippstufe*, auch *bistabiles Kippglied* oder *Riegel*, ist eine rückgekoppelte Schaltung aus zwei NOR- oder zwei NAND-Gattern mit zwei Ein- und Ausgängen (auch speziell *SR-Riegel* genannt), die zwei Stabile (daher bistabile) und einen metastabilen Zustand hat.

Oft wird einfach (aber ungenau) von *FlipFlop* gesprochen

Abbildung 4.4: Übersicht: Bistabile Kippstufen



4.2.1 Implementierung mit Gattern

Abbildung 4.5: Riegel aus NOR-Gattern

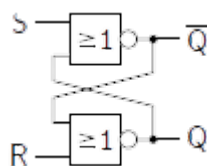


Abbildung 4.6: Riegel aus NAND-Gattern

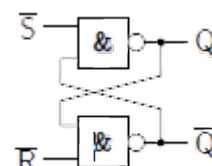
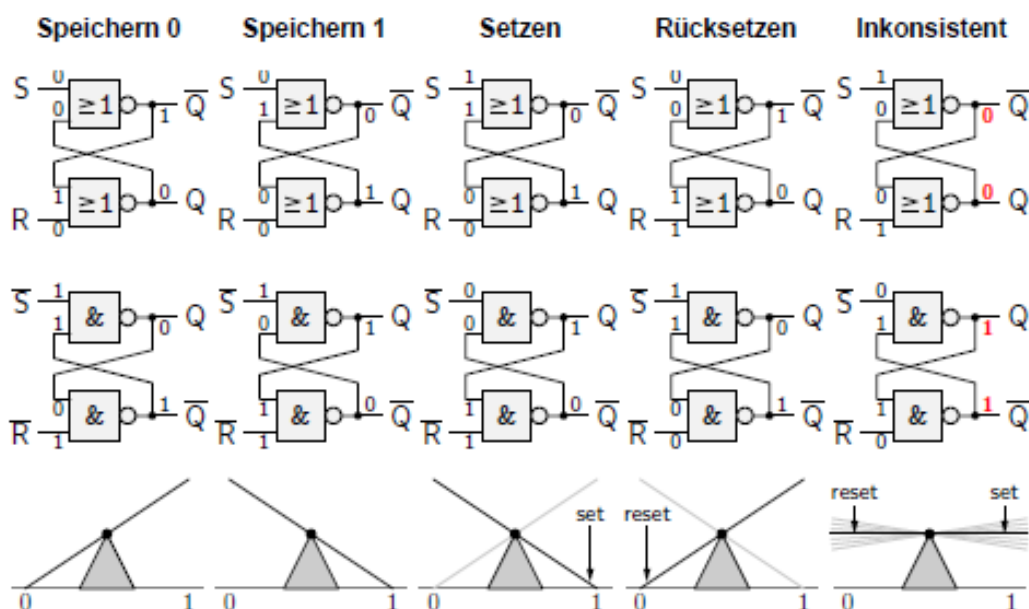


Abbildung 4.7: Zusammenfassung aller Zustände bistabiler Kippstufen



4.2.2 SR-Riegel

Stabile Zustände

Wenn die Eingabe des Riegels $S = R = 0$ (bzw. $\bar{S} = \bar{R} = 1$ im NAND-Gatter) ist, bleibt der Riegel stabil 0 oder 1:

Abbildung 4.8: NOR-Riegel mit $S = R = 0$

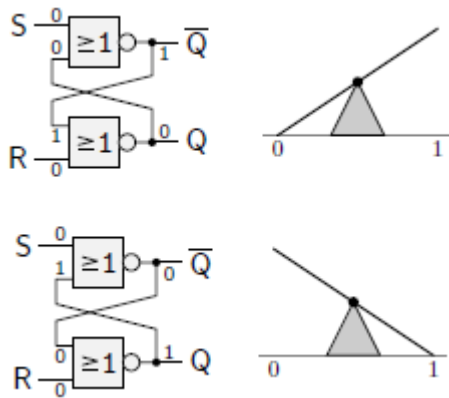
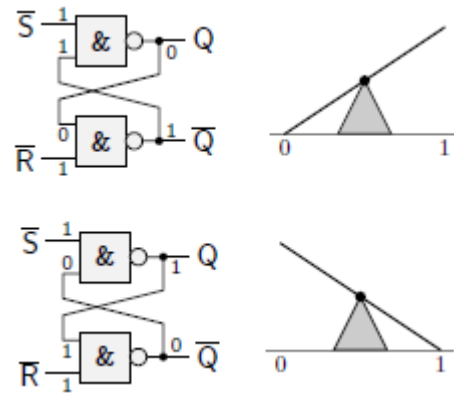


Abbildung 4.9: NAND-Riegel mit $\bar{S} = \bar{R} = 1$



Die Eingabe kann somit als *Speicherzustand* interpretiert werden, in dem die Kippstufe ihren Zustand beibehält

Setzen & Rücksetzen (set & reset)

Die Eingabe
 $S = 1, R = 0$ (NAND: $\bar{S} = 0, \bar{R} = 1$)
kann als Setzen (set) interpretiert werden

Die Eingabe
 $S = 0, R = 1$ (NAND: $\bar{S} = 1, \bar{R} = 0$)
kann als Rücksetzen (reset) interpretiert werden

Abbildung 4.10: NOR-Riegel

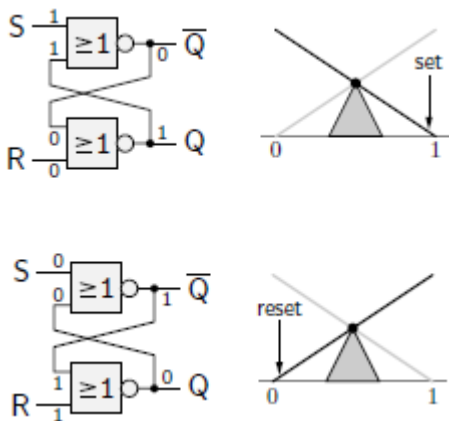
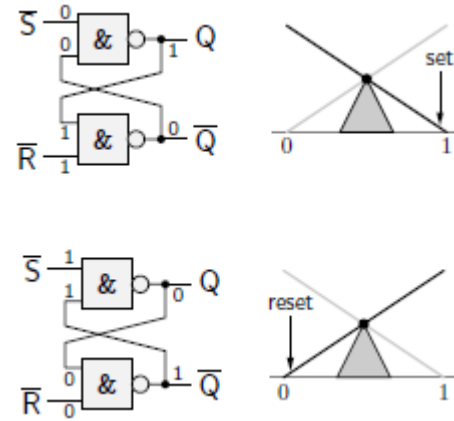


Abbildung 4.11: NAND-Riegel



Diese Zustände bleiben erhalten, wenn die Eingaben auf $S = R = 0$ (NAND: $\bar{S} = \bar{R} = 1$) wechseln.

Inkonsistenter (metastabiler) Zustand

Die Eingaben $S = R = 1$ (NAND: $\bar{S} = \bar{R} = 0$) führen zu einem inkonsistenten Zustand, da in diesem Fall $Q = \bar{Q} = 0$ (NAND: $Q = \bar{Q} = 1$) gilt, also die Ausgänge nicht komplementär sind, ausserdem der Übergang in den Speicherzustand $S = \bar{R} = 0$ (NAND: $\bar{S} = \bar{R} = 1$) undefiniert ist.

Abbildung 4.12: NOR-Riegel

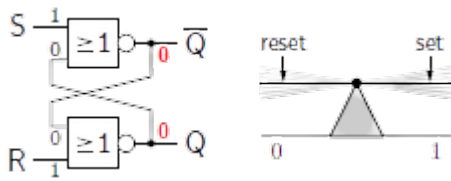
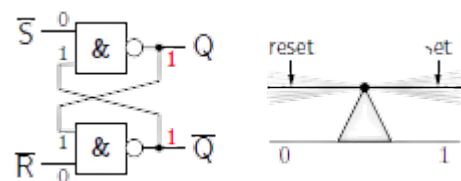


Abbildung 4.13: NAND-Riegel



Dieser Zustand wird metastabil genannt, da der Speicherzustand undefiniert wird. Die Kippstufe bleibt in diesem Zustand, bis eine Eingabe die Oberhand gewinnt und sie in den zugehörigen Zustand bringt.

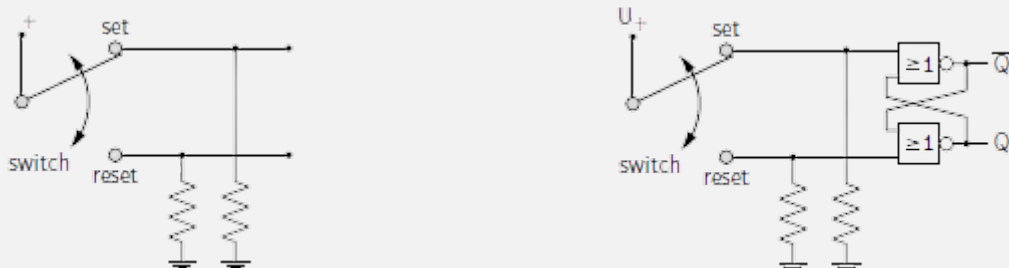
INFO:

Bistabile Kippstufen können durch ihre beiden stabilen Zustände ein Bit speichern, aber der metastabile Zustand ist problematisch.

BEISPIEL: Anwendung - Prellfreier Schalter/Taster

Bei elektromagnetischen Schaltern kommt es durch mechanische Störeffekte oft zu einem sogenannten Prellen des Schalters. Statt eines sofortigen Schaltsch, kommt es kurzzeitig zu mehrfachem.

Wird an die Schalterkontakte eine bistabile Kippstufe angeschlossen, so wird das Prellen unterdrückt, da ein mehrfaches Eingangssignal nur einmalig zu einer Zustandsänderung führt.



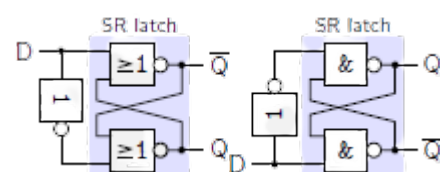
4.2.3 Direkt gesteuerte FlipFlops (Riegel)

Durch vorgeschaltete Zusatzgitter kann die problematische Eingabe $S = R = 1$ bzw. $\bar{S} = \bar{R} = 0$ ausgeschlossen werden.

D-Riegel (D latch)

- Stellt sicher, dass $R = \neg S$
- Vorteil: Schließt problematische Eingaben aus
- Nachteil: $S = R = 0$ geht ebenfalls verloren

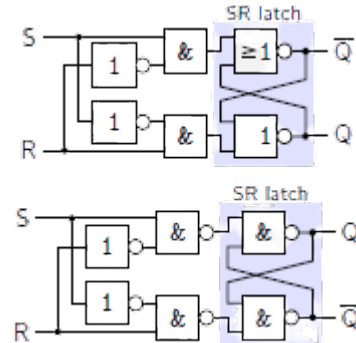
Abbildung 4.14: D-Riegel



E-Riegel (E latch)

- Stellt sicher, dass $R = \neg S$
- Vorteil: Schließt problematische Eingaben aus
- Nachteil: $S = R = 0$ geht ebenfalls verloren
- Man beachte, dass die NAND-Form keine negierten Eingaben mehr hat!

Abbildung 4.15: E-Riegel



4.2.4 Taktpegelsteuerung

Durch anlegen eines Taktsignals, können Riegel ihre Eingaben bedingt sperren, so dass nur bei 1-Taktpegel Eingaben übernommen werden.

- Sperrt Eingaben bedingt, so dass $S = R = 1$ im Sperrzustand keinen Schaden mehr anrichten kann
- Man beachte, dass die NAND-Form keine negierten Eingaben mehr hat!

Zusätzliches Sicherstellen von $R = \neg S$ ergibt den *sperrbaren D-Riegel*:

Abbildung 4.16: Sperrbarer D-Riegel

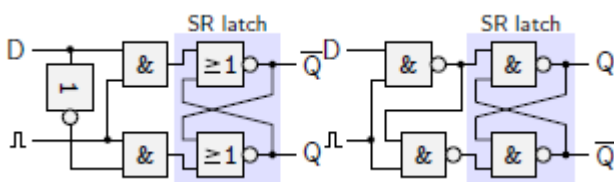
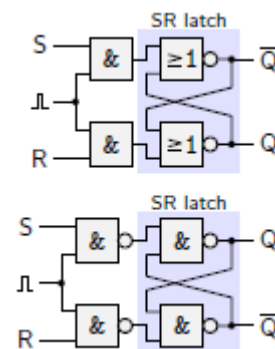


Abbildung 4.17: Sperrbarer SR-Riegel

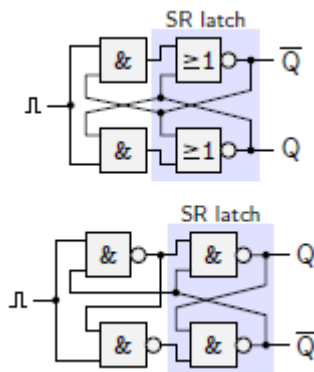


4.2.5 Taktpegelsteuerung mit Rückkopplung

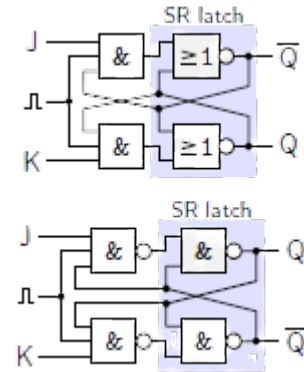
Durch eine Rückkopplung der Ausgaben des SR-Riegels auf die Sperrgatter lässt sich erreichen, dass die Eingabe $S = R = 1$ eine neue Funktion erhält:

Das Umkehren (toggle) des bestehenden Speicherzustands

T-Riegel (T latch)



JK-Riegel (JK latch)



BEISPIEL: Laufzeitprobleme (Schwingungen)

Man beachte aber: Das durch die Eingabe $S = R = 1$ bewirkte Umkehren führt zu Laufzeitproblemen (Schwingungen), da sich kein stabiler Zustand einstellt

4.2.6 Taktflankensteuerung

Problem der Taktpegelsteuerung ist, dass die Eingaben während der gesamten 1-Phase des Taktes eine Wirkung auf den Zustand des Riegels ausüben.

Besser wäre eine Übernahme der Eingaben an der steigenden Flanke: Die Taktflankensteuerung

DEFINITION: Taktflankensteuerung

Taktflankensteuerung löst das Problem der Taktpegelsteuerung, bei dem die Eingaben während der gesamten 1-Phase des Taktes eine Wirkung auf den Zustand des Riegels ausüben.

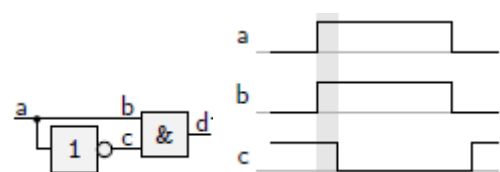
Bei Taktflankensteuerung spricht man nicht mehr von Riegel (dieser ist direkt oder taktpegelgesteuert), sondern von *FlipFlop*

Durch die Nutzung von NOT-Gattern (Invertieren) können Signale verzögert werden (Verzögerungsglieder).

Verbindet man diese mit einem AND-Gatter, bilden sie ein *Impulsglied*

Dadurch wird aus einer Taktflanke ein kurzer *Taktimpuls* erzeugt

Abbildung 4.18: Schaltung zur Erzeugung eines Taktimpulses



Zur Implementierung der Taktflankensteuerung schaltet man eine Schaltung (wie oben) vor den Eingang für das Taktsignal.

Abbildung 4.19: D-FlipFlop (D-Riegel mit Taktflankensteuerung)

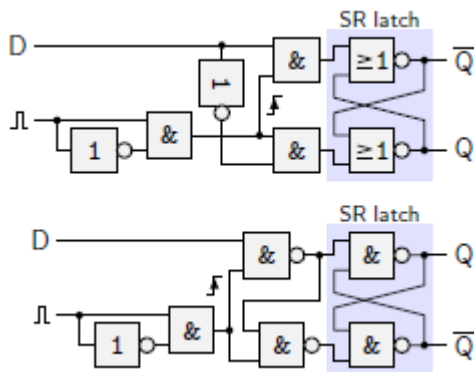
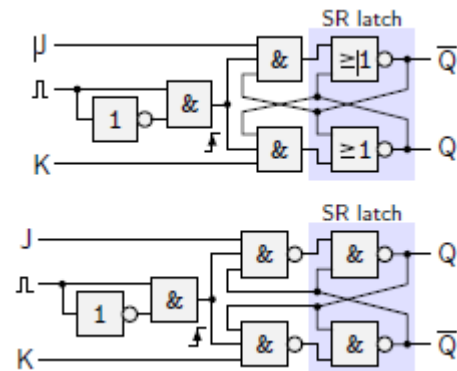


Abbildung 4.20: JK-FlipFlop (JK-Riegel mit Taktflankensteuerung)



INFO: Taktflankensteuerung & Laufzeitprobleme

Taktflankensteuerung mindert Laufzeitprobleme und macht Schaltungen robuster, da Gatterlaufzeiten von den Umgebungsbedingungen abhängen können, kann es aber dennoch zu Laufzeitproblemen kommen.

4.2.7 Master-Slave-Prinzip

Gerade hintereinandergeschaltete FlipFlops können noch zu Problemen führen, wenn eine vorangehende Stufe schon ihre Ausgaben ändert, bevor die nachfolgende Stufe die Eingabeauswertung abgeschaltet hat. (Gatterlaufzeiten!)

DEFINITION:

Beim *Master-Slave-Prinzip* werden zwei Riegel hintereinandergeschaltet. Der erste (vordere) Riegel (Master) wertet seine Eingaben während des 1-Taktpegels, der zweite (hintere) Riegel (Slave) wertet seine Eingaben während des 0-Taktpegels aus. Während der eine Riegel seine Ausgaben auswertet, sind die Ausgaben des jeweils anderen stabil, da jeweils nur einer der Riegel aktiv ist. So kann man (fast) alle Laufzeitprobleme vermeiden

Implementiert wird das Prinzip mit *Zweipegelsteuerung*. Hierbei bekommt der Slave-Riegel ein Taktsignal, das gegenüber dem Master-Riegel invertiert ist.

4.2.8 Schaltzeichen

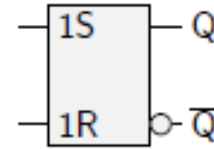
Da Riegel und Flipflops sehr häufig verwendete Schaltungsbausteine sind, werden sie durch eigene Schaltzeichen dargestellt.

Diese Schaltzeichen sind einfache Rechtecke:

Ein- und Ausgänge:

- Eingänge links (z.B.: S, D, J)
- Ausgänge rechts (z.B.: R, K)
- Setzen-Eingänge oben, Rücksetzen-Eingang unten
- Ausgang Q oben, \overline{Q} unten.
- Negationszeichen am Ausgang \overline{Q}
- Setzen- und Rücksetzen-Eingänge: 1 links der Bezeichnung versehen (z.B.: 1S, 1R, 1D, 1J, 1K)

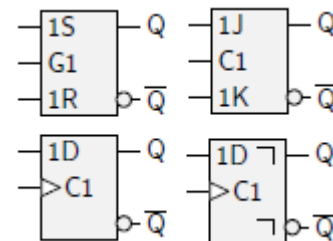
Abbildung 4.21: Schaltzeichen - SR-Riegel



Gated- / Clock-Eingänge:

- Gated (G) und Clock (C) Eingänge: links in der Mitte
- Eingänge, die Setzen-/Rücksetzen-Eingänge modifizieren: 1 links der Bezeichnung (z.B.: G1, C1)
- Negationsblase: Takteingänge mit reagieren auf 0-Taktpegel, ohne auf 1-Taktpegel (Flipflops: fallende, ohne steigende Flanke)
- weißes Dreieck an einem Takteingang: *Taktflankensteuerung*
- Winkel an Ausgang: Master-Slave-Flipflops

Abbildung 4.22: Riegel mit Gate & Clock & Master-Slave



4.2.9 Schaltverhalten

4.3 Register, Zähler und Speicher

4.3.1 Schieberegister und Zähler

DEFINITION:

Schieberegister können durch Verkettung von D-Flipflops erstellt werden. Mit jedem Taktzyklus wird der Speicherinhalt der Flipflops um ein Flipflop weitergeschoben. Schieberegister haben eine feste Anzahl von Speicherplätzen (Anzahl der Flipflops) und arbeiten nach dem FIFO-Prinzip.

Abbildung 4.23: 4-Bit seriell zu parallel

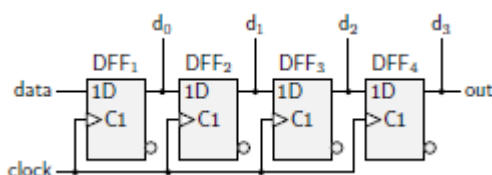
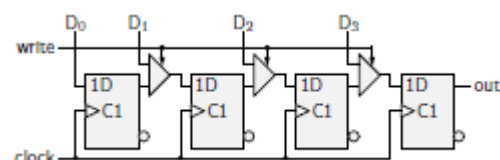


Abbildung 4.24: 4-Bit parallel zu seriell



Schieberegister können auch zur Wandlung eines seriellen Datenstroms in einen parallelen (4.23) oder eines parallelen Datenstroms in einen seriellen (4.24) benutzt werden.

Einfache Rückgekoppelte Schieberegister: Ringzähler

Abbildung 4.25: Einf. 4-Bit-Ringzähler

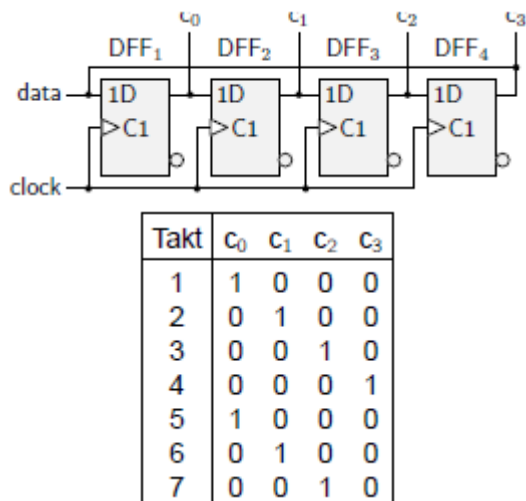
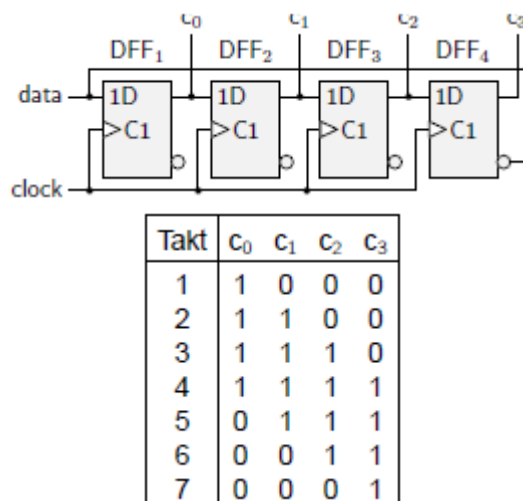


Abbildung 4.26: 4-Bit-Johnson-Ringzähler



Linear Rückgekoppelte Schieberegister: Ringzähler

Abbildung 4.27: Linear Rückgekoppeltes Schieberegister (LFSR)

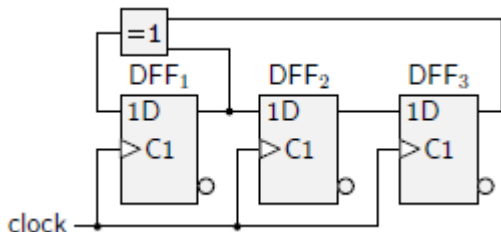


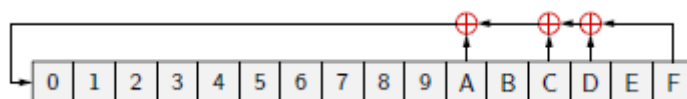
Abbildung 4.28: Linear Rückgekoppeltes Schieberegister (LFSR)



Durch komplizierte Rückkopplungen, die nicht nur vom letzten Flipflop, sondern auch von dazwischenliegenden rückkoppeln (Verknüpfung der Rückkopplungen über exklusives Oder) kann man komplizierte Bitfolgen erzeugen

Solche *linear rückgekoppelten Schieberegister* werden vereinfacht wie Abb. 4.28 dargestellt

Abbildung 4.29: Linear Rückgekoppeltes Schieberegister



Das LFSR in Abb. 4.29 hat Rückkopplungen von den Positionen 10, 12, 13 und 15. Solche Schieberegister werden z.B. zur Erzeugung von Pseudozufallszahlen oder zur Zyklischen Redundanzprüfung verwendet

Kapitel 5

Rechnerarchitektur

5.1 Speicherprogrammierung

5.1.1 Festverdrahtete "Prozessoren"

DEFINITION:

Im Gegensatz zu frei programmierbaren Prozessoren sind in *festverdrahteten Prozessoren* die Bewegungen des Befehlszählers festgelegt (durch Schaltkreise bestimmt).

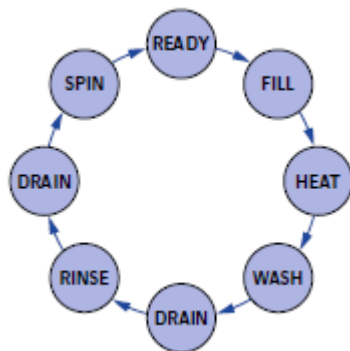
Da dies nur dann akzeptable ist, wenn der Prozessor nur eine einzelne festgelegte Aufgabe zu erfüllen hat, ist das in einfachen Automaten (z.B.: Waschmaschinen) der Fall.

Diese durchlaufen eine feste Abfolge von Zuständen, in denen festgelegte Aktionen ausgeführt werden.

Verzweigungen sind zwar Prinzipiell möglich, aber unveränderbar (festes Programm)

Ein gutes Beispiel für festverdrahtete Prozessoren sind Waschmaschinen:

Abbildung 5.1: Zustandsdiagramm einer Waschmaschine



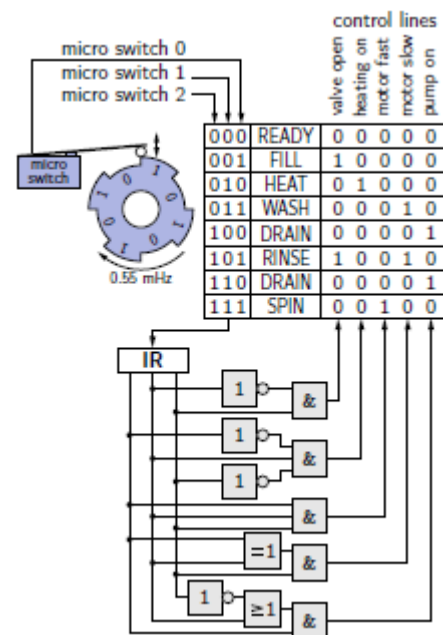
Der Prozessor kann die Waschmaschine anhand gegebener Anweisungen steuern.

- FILL → Ventil AUF - Heizung AUS
- HEAT → Ventil ZU - Heizung AN

Er besitzt z.B. Nockenscheiben, die sich mit einer gewissen frequenz drehen und dabei Mikroschalter betätigen.

Ein Schaltnetz implementiert dann die dekodierung der Ausgabe, die von den Mikroschaltern gesteuert wird.

Abbildung 5.2: Schaltung einer Waschmaschine



DEFINITION: Ablaufsteuerung

Die hier gezeigte Schaltung heißt *Ablaufsteuerung*. Sie läuft schrittweise ab, wobei von einem Schritt auf den nächsten gemäß vorgegebener Übertragungsbedingungen weitergeschaltet wird.

Gegenüber Rechnern mit freier Programmierbarkeit sind Ablaufschaltungen meist eingeschränkt, oft sogar festverdrahtet.

5.1.2 Konzept der Speicherprogrammierung

DEFINITION:

Für eine freie Programmierbarkeit von Automaten oder Rechnern ist das *Konzept der Speicherprogrammierung* entscheidend:

- Anweisungen werden nicht festverdrahtet, sondern als kodierte Befehle in einem Speicher abgelegt.
- Programme sind dadurch prinzipiell austauschbar und speicherprogrammierbare Rechner folglich nicht auf eine bestimmte Aufgabe begrenzt
- Dies ermöglicht *universelle Rechenmaschinen*

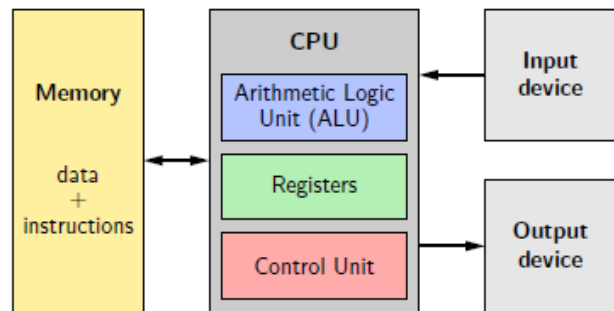
INFO:

Speicherprogrammierung wurde entscheidend von John von Neumann (1945) geprägt, der Ideen von Alan Turing (1936) weiterentwickelte, welcher wiederum mathematische Ideen von Kurt Gödel (1930) aufgegriffen hatte

Ausführen einer Anweisung erfordert einen oder mehrere der folgenden Teilschritte:

- Arithmetisch-Logische Einheit (ALU) berechnet eine Funktion $f(\text{registers})$
- Ausgabe der ALU wird in ein Register geschrieben
- Nächste auszuführende Anweisung muss bestimmt werden (Bei Verzweigungsbehl u.U. nicht die im Speicher folgende)

Abbildung 5.3: Speicherprogrammierung



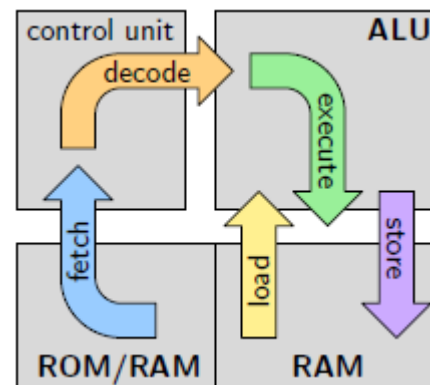
5.1.3 Befehlsaufruf, -dekodierung und -ausführung

Im Konzept der Speicherprogrammierung besteht das Ausführen eines Befehls aus folgenden drei Schritten:

- Befehlsabruf (fetch)
- Befehlsdekodierung (decode)
- Befehlsausführung (execute)

Das ist der *Fetch-Decode-Execute Cycle* (Abb. 5.4), der zur Programmausführung immer wieder durchlaufen wird.

Abbildung 5.4: Fetch-Decode-Execute Cycle



Fetch Übertragen des nächsten auszuführenden Befehls in die Steuereinheit des Prozessors

Decode Dekodieren des Befehls durch die Steuereinheit des Prozessors

Execute Anweisung an ALU, die im Programmbefehl kodierte Berechnung f auszuführen

Load, Store Berechnung kann das Laden und Ablegen von Berechnungsergebnissen in den Datenspeicher erfordern

5.1.4 Rechnerarchitekturen (Harvard & von Neumann)

Bei der Harvard-Architektur (Abb. 5.5) liegen Programm und Daten in zwei verschiedenen Speichern, während sie bei der von-Neumann-Architektur (Abb. 5.6) in einem einzigen liegen

Abbildung 5.5: Harvard Architektur

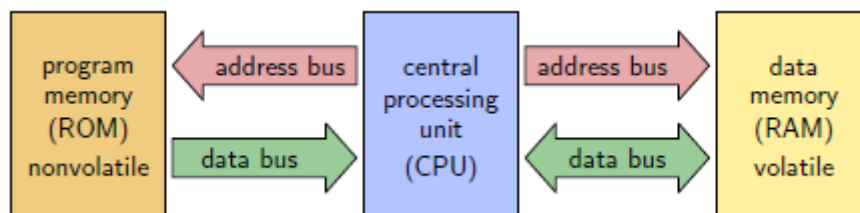
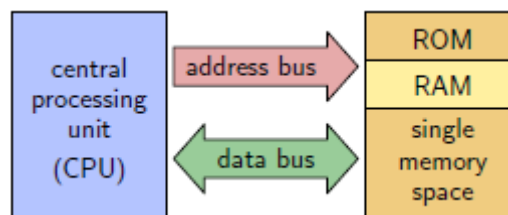


Abbildung 5.6: von Neumann Architektur



Kapitel 6

Maschinensprache und Assembler

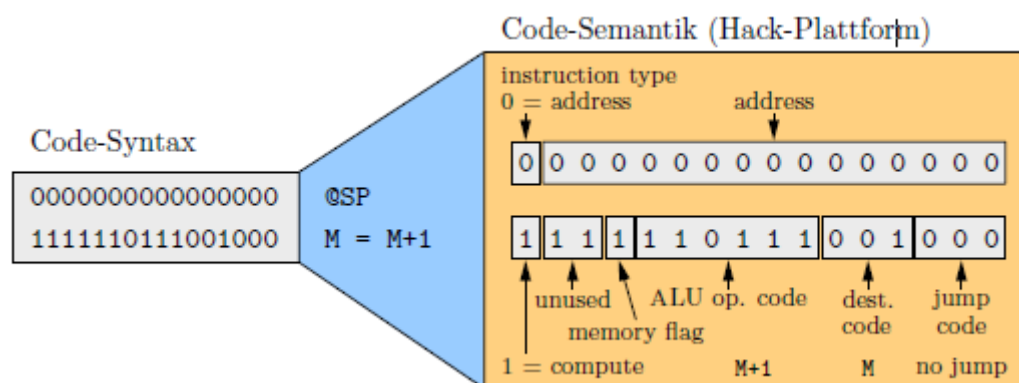
6.1 Die Hack-Maschinensprache

6.1.1 Einführung in die Maschinensprache

DEFINITION: Maschinensprache

Maschinensprache kann von Rechnern direkt ausgeführt werden und ist deshalb auch abhängig von der Hardware-Plattform, welche ihre Semantik realisiert. Auch besteht die Maschinensprache nicht mehr aus Symbolen, sondern aus Binärzahlen, weshalb sie von Menschen nur mit großen Schwierigkeiten zu lesen ist

Abbildung 6.1: Semantik der Maschinensprache



Da die Hardware-Plattform für die Realisierung der Semantik zuständig ist, sollte diese in der Lage sein,

- die Befehle zu interpretieren (gemäß Semantik Abb. 6.1)
- und auszuführen (Operationen durchführen)

Anweisungen

Die Hack-Maschinensprache besteht nur aus zwei Arten von Anweisungen:

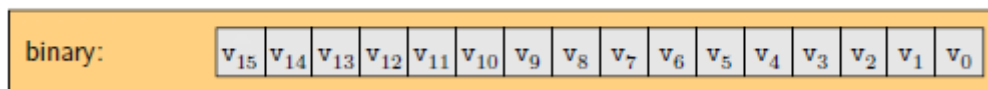
A-Anweisungen (address)

- address instructions
- A instructions

C-Anweisungen (compute)

- compute instructions
- C instructions

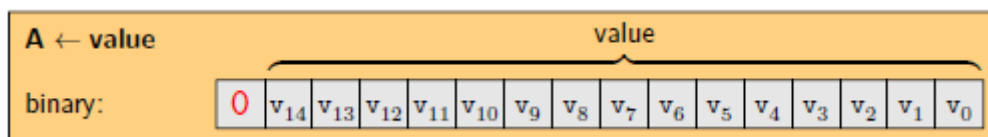
Abbildung 6.2: Anweisung



Das 15-te Bit (Abb. 6.2, rot markiert) bestimmt die Art der Anweisung: Eine 0 steht für A-Anweisung, während eine 1 für C-Anweisung steht.

6.1.2 A-Anweisungen (Address Instructions)

Abbildung 6.3: A-Anweisung

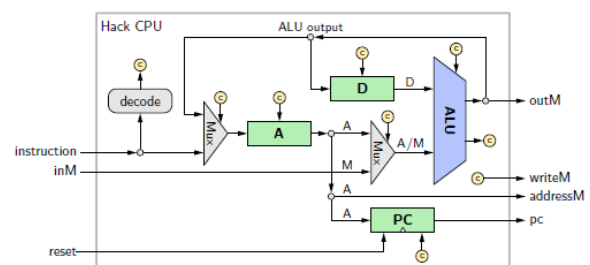


Eine A-Anweisung lädt einen (konstanten) 15-Bit-Wert in das A-Register. Dieser Wert kann später verwendet werden als:

- Datenspeicheradresse
- neuer Wert des Befehlszählers
- Wert der in eine Berechnung der ALU eingeht

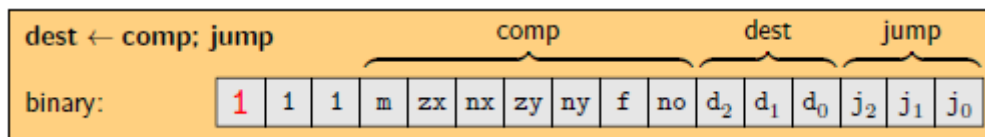
Das oberste Bit der A-Anweisung beeinflusst den Multiplexer vor dem A-Register und das Steuerbit des A-Registers

Abbildung 6.4: Hack-CPU



6.1.3 C-Anweisungen (Compute Instructions)

Abbildung 6.5: A-Anweisung



Die C-Anweisung führt eine Berechnung mit Hilfe der Arithmetisch-Logischen Einheit (ALU) durch:

- Die auszuführende Berechnung (computation, comp) ist in den Bits m , zx , nx , zy , ny , f und no kodiert.

→ ALU-Steuerbits

- Die Bits d_2 , d_1 und d_0 (destination, dest) bestimmen den Speicherort des Ergebnisses

→ Wählt zwischen D-Register, A-Register, und Memory[A] (= durch A-Register adressierte Speicherzelle) (bzw. Kombination)

- Die Bits j_2 , j_1 und j_0 (jump) bestimmen, ob und unter welchen Bedingungen gesprungen (verzweigt) werden soll

→ Wählt zwischen $out = 0$, $out < 0$, $out > 0$ (bzw Kombination, wie $out \leq 0$)

Man beachte, dass das höchstwertige Bit einer C-Anweisung (Abb. 6.5) nun den Wert 1 (für C-Anweisung) hat.

Abbildung 6.6: Destination

d_2 d_1 d_0	Mnemonic	Speicherziel
0 0 0	—	Wert wird nicht gespeichert.
0 0 1	M	Memory [A]
0 1 0	D	D-Register
0 1 1	MD	Memory [A] und D-Register
1 0 0	A	A-Register
1 0 1	AM	Memory [A] und A-Register
1 1 0	AD	A- und D-Register
1 1 1	AMD	Memory [A] und A- und D-Register

Abbildung 6.7: Jump

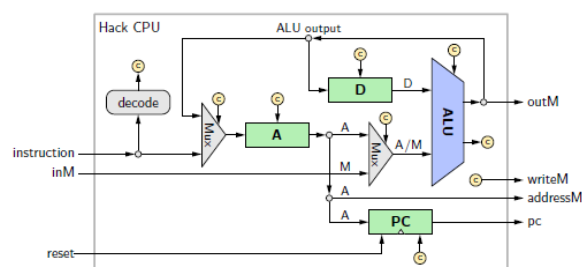
j_2 (out < 0)	j_1 (out = 0)	j_0 (out > 0)	Mnemonic	Sprung
0	0	0	—	niemals
0	0	1	JGT	falls out > 0
0	1	0	JEQ	falls out = 0
0	1	1	JGE	falls out ≥ 0
1	0	0	JLT	falls out < 0
1	0	1	JNE	falls out ≠ 0
1	1	0	JLE	falls out ≤ 0
1	1	1	JMP	immer

- Die *dest-Bits* beeinflussen die Steuerbits der D- und A-Registers, den Multiplexer vor dem A-Register und die Ausgabe writeM

→ Je nach Steuerbits kann in einige oder alle dieser Ziele geschrieben werden

- Die *jump-Bits* beeinflussen zusammen mit den Ausgaben zr und ng der ALU die Steuerbits des PC-Registers (program counter)

Abbildung 6.8: Hack-CPU



6.2 Assembler und Assemblersprache

6.2.1 Physikalische und symbolische Programmierung

Dualität von Hardware (Rechner) und Software (Programm)

- Maschinensprache kann als abstrakte Beschreibung (der Fähigkeiten) der Hardware-Plattform gesehen werden
- Hardware kann als physikalisches Mittel gesehen werden, um eine abstrakte Maschinensprache zu realisieren

6.2.2 Maschinensprache und Assemblersprache

DEFINITION: Maschinensprache

ist nah am physikalischen Rechner
Programme als Folgen von *Binärzahlen*,
die als Befehle interpretiert werden

DEFINITION: Assemblersprache

symbolische Form der Masch.sprache
Symbole (für menschen verständlich) zur
Darstellung von Programmen

Jede Binärzahl der Maschinensprache entspricht ein einfacher symbolischer Ausdruck in der Assemblersprache. Diese Symbolischen Ausdrücke müssen mithilfe eines Assemblers aus der Assemblersprache in die Maschinensprache übersetzt werden.

DEFINITION:

Ein Assembler ist ein Programm zur Übersetzung der Assemblersprache in die, von der Hardware-Plattform direkt ausführbare, Maschinensprache.

BEISPIEL: Beispiel C-Anweisung in Hack-Maschinensprache

Assemblersprache	Maschinensprache (binär)	Hexadezimal
$A = A - 1$	1110 1100 1001 0111	0xEC97

6.2.3 Die Hack-Assemblersprache

C-Anweisungen

C-Anweisungen bestehen aus drei Komponenten:

DEST	=	COMP	;	JUMP
Speicherort des Ergebnisses		Auszuführende Berechnung		Sprungbedingung

Computation Symboltabelle (ALU)

zx	nx	zy	ny	f	no	$m = 0$	$m = 1$
1	0	1	0	1	0	0	
1	1	1	1	1	1	1	
1	1	1	0	1	0	-1	
0	0	1	1	0	0	D	
1	1	0	0	0	0	A	M
0	0	1	1	0	1	$D/!D$	
1	1	0	0	0	1	$A/!A$	$M/!M$
0	0	1	1	1	1	$-D$	
1	1	0	0	1	1	$-A$	$-M$
0	1	1	1	1	1	$D + 1$	
1	1	0	1	1	1	$A + 1$	$M + 1$
0	0	1	1	1	0	$D - 1$	
1	1	0	0	1	0	$A - 1$	
0	0	0	0	1	0	$D + A$	$D + M$
0	1	0	0	1	1	$D - A$	$D - M$
0	0	0	1	1	1	$A - D$	$M - D$
0	0	0	0	0	0	$D \& A$	$D \& M$
0	1	0	1	0	1	$D A$	$D M$

Destination Symboltabelle

Symbol	Speicherort
-	Keine Speicherung
M	Memory[A]
D	D-Register
MD	Memory[A] & D-Register
A	A-Register
AM	A-Register & Memory[A]
AD	A-Register und D-Register
AMD	A & D-Reg., Memory[A]

Jump Symboltabelle

Symbol	Sprungbedingung
-	Spring niemals
JGT	Spring falls <i>out</i> > 0
JEQ	Spring falls <i>out</i> = 0
JGE	Spring falls <i>out</i> ≥ 0
JLT	Spring falls <i>out</i> < 0
JNE	Spring falls <i>out</i> ≠ 0
JLE	Spring falls <i>out</i> ≤ 0
JMP	Spring immer

A-Anweisungen

Verwendung der A-Anweisung

@value	Vorgang	Beschreibung
$D = A$	$D \leftarrow value$	Laden einer Konstante
$D = M$	$D \leftarrow RAM[value]$	Auswahl Datenspeicherzelle
JMP	$fetch\ ROM[value]$	Auswahl Befehlsspeicherzelle

@value
SOMETHING

L-Anweisungen

Die Anweisung (LABEL) deklariert ein neues Label mit dem Name 'Label'. Der Assembler übersetzt diese dann in die Adresse der nächsten Anweisung (nachfolgende Zeile)

(LABEL)
// Instructions
@LABEL
0; JMP

6.2.4 Symbole und Symbolverwaltung

Symbol	Beschreibung
A	A-Register (Adressenregister)
D	D-Register (Datenregister)
M	Hauptspeicher-Register (Adresse A)
SP	RAM-Adresse 0
LCL	RAM-Adresse 1
ARG	RAM-Adresse 2
THIS	RAM-Adresse 3
THAT	RAM-Adresse 4
R0-R15	RAM-Register (16)
SCREEN	16384 Adresse des Bildschirmspeichers
KBD	24576 Adresse des Tastaturregister

6.2.5 Programmübersetzung und Assemblerimplementierung

Initialisierung der Symboltabelle:

- Leere Symboltabelle wird erzeugt
- vordefinierte Symbole eingefügt

Erster Durchlauf:

- Eintragung von benutzerdefinierten Marken in Symboltabelle
- Für jede Markendef. ein paar (*LABEL*, *n*) (*n* Anzahl der bereits durchlaufenden Zeilen)

Zweiter Durchlauf:

- Markendefinitionen übersprungen
- C-Anweisungen:
 - Aufsuchen und Zusammensetzen der zugehörigen Binärcodes
 - gib erhaltene Binärzahl aus
- A-Anweisungen (@*xxx*):
 - Falls *xxx* Zahl ist: Gib Binärzahl aus
 - Falls *xxx* Marke ist: Suche Symbol in Tabelle
 - * Vorhanden: lies in Zahl *k* aus Tabelle aus
 - * Nicht Vorhanden: füge Symbolpaar (*xxx*, *k*) hinzu (*k* ist Adresse der nächsten freien Datenspeicherzelle, ab 16)

Gibt *k* als Binärzahl aus

Kapitel 7

Virtuelle Maschine

7.1 Hochsprachen und Übersetzung

DEFINITION: Höhere Programmiersprachen

Hochsprachen Programmiersprachen abstrahieren von konkreten Eigenschaften des Rechners. Dadurch sind sie leichter zu verstehen als Sprachen tieferer Ebenen.

Aber:

- können nicht direkt ausgeführt werden
- müssen in Sprachen tieferer Ebenen übersetzt werden

7.1.1 Direkte und zweistufige Übersetzung

Ein großes Problem der direkten Übersetzung ist, dass es viele verschiedene (Hoch-)Sprachen und Hardware-Plattformen gibt:

- Direkte Übersetzung erfordert einen Übersetzer pro Sprache und pro Plattform
- n Sprachen und m Plattformen erfordern $n \times m$ Übersetzer

Abbildung 7.1: Direkte Übersetzung

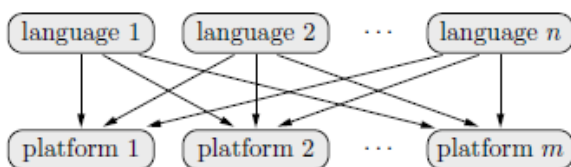
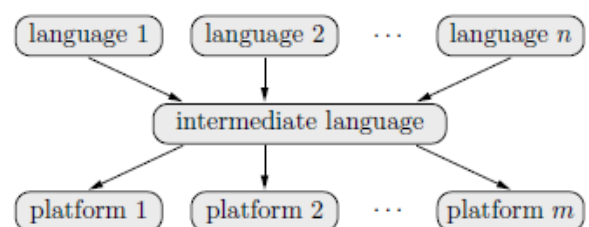


Abbildung 7.2: Zweistufige Übersetzung



Aus diesem Grund werden Hochsprachen zunächst in eine Zwischensprache übersetzt, die unabhängig von der Hardware-Plattform ist

DEFINITION: Zweistufige Übersetzung

Bei der Zweistufigen Übersetzung werden Hochsprachen und Plattformen entkoppelt:

- Erste Stufe hängt nur von Hochsprache ab (Compilation)
- Zweite Stufe hängt nur von Zielmaschine ab (translation/interpretation)

Die Zwischensprache kann als Assembler-/Maschinensprache einer virtuellen oder Pseudo-hardware-Plattform gesehen werden

7.1.2 Vor- und Nachteile Virtueller Maschinen

Vorteile

- Plattform-Unabhängigkeit
- Dynamische Optimierung auf spezielles Zielsystem möglich/einfacher
- Implementierung von Übersetzern wird einfacher

Nachteile

- Effizienzverlust ggü. direkter Übersetzung
- langsamere Ausführung
- Auch kleinere Programme benötigen virtuelle Maschine
- weniger Kontrolle über Zielcode

7.1.3 Systembasierte und prozessbasierte virtuelle Maschinen

Systembasiert

- mehrere Betriebssysteme auf einem Rechner
- so vollständige Nachbildung realer Rechner, dass Betriebssystem ausgeführt werden kann

Prozessbasiert

- Programmausführung unabhängig der Rechnerarchitektur
- Abstrahierung einzelner Programme

7.1.4 Übersetzungspfad

Übersetzung von Jack:

Jede *Klasse* hat:

- Eine Liste statischer Variablen
→ globale Variablen

Jede *Funktion* hat:

- Eine Liste von Argumenten
- Eine Liste lokaler Variablen

Die *Übersetzung* muss den Zugriff auf diese Listen organisierten

Deshalb werden den verschiedenen Listen der Klassen und Funktionen *Speicher-segmente* zugewiesen. Für die lokalen Variablen werden sie dynamisch bestimmt.

Abbildung 7.3: VM Translator

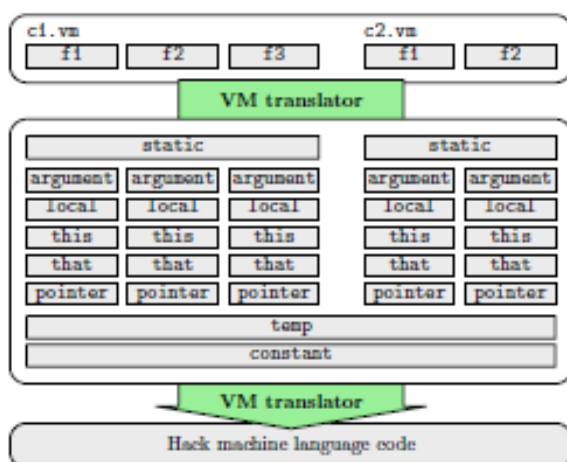


Abbildung 7.4: Jack-Quelltext

```
class c1 {
    static int x1, x2, x3;
    method int f1 (int x) {
        var int a, b;
        ...
    }
    method int f2 (int x, int y) {
        var int a, b, c;
        ...
    }
    method int f3 (int u) {
        var int x;
        ...
    }
}

class c2 {
    static int y1, y2;
    method int f1 (int u, int v) {
        var int a, b;
        ...
    }
    method int f2 (int x) {
        var int a1, a2;
        ...
    }
}
```

7.2 Virtuelle Maschine des Hack-Systems

Die virtuelle Maschine für das Hack-System, die auf einem Stapel als zentrale Datenstruktur und Funktionsaufrufen arbeitet (weitgehend analog zur virtuellen Maschine von Java)

Es wird nur ein einziger 16-Bit-Datentyp verwendet (alle Daten sind 16-Bit)

Wichtig für die Betrachtung der virtuellen Maschine sind:

- Arithmetisch-logische Operationen
- Speicherzugriff
- Programmablaufsteuerung
- Funktionsaufrufe

7.2.1 Stapel(speicher) und ihre Operationen)

DEFINITION: Stapel(-speicher)

Ein Stapel(-speicher) oder Keller(-speicher) (stack) ist eine häufig verwendete dynamische abstrakte Datenstruktur, die Daten nach dem LIFO-Prinzip speichert

Ein Stapel stellt zwei (bzw. drei) Operationen zur Verfügung:

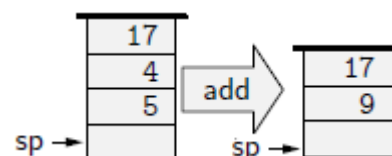
- **push** ("einkellern"): Objekt wird oben auf den Stapel gelegt
- **pop** ("auskellern"): oberstes Objekt wird vom Stapel entfernt und zurückgegeben
- **top / peek** ("nachsehen"): Optionale Option, bei der das oberste Objekt vom Stapel zurückgegeben aber nicht entfernt wird

7.2.2 Stapelarithmetik

Typische arithmetisch-logische Operation mit einem Stapel:

- Hole die beiden obersten Werte x und y vom Stapel (pop)
- Berechne Wert der Funktion $f(x, y)$
- Lege Ergebnis z auf Stapel ab (push)

Abbildung 7.5: Stapelarithmetik



Diese Art der Berechnung entspricht der Schreibweise arithmetisch-logischer Operationen in *Postfixnotation* oder *umgekehrter polnischer Notation* (UPN)

7.2.3 Arithmetische und Logische Operationen

Die Operationen der virtuellen Maschine sind gegenüber der Assemblersprache eingeschränkt. Anweisungen, wie *le*, *ge*, etc. ließen sich leicht hinzufügen, können aber auch anders erzeugt werden.

Anweisung	Rückgabewert	Beschreibung
<i>add</i>	$x + y$	Ganzzahladdition (2er-Komplement)
<i>sub</i>	$x - y$	Ganzzahlsabtraktion (2er-Komplement)
<i>neg</i>	$-y$	arithmetische Negation (2er-Komplement)
<i>eq</i>	-1 falls $x = y$, sonst 0	Test auf Gleichheit
<i>gt</i>	-1 falls $x > y$, sonst 0	Test auf größer
<i>lt</i>	-1 falls $x < y$, sonst 0	Test auf kleiner
<i>and</i>	$x \& y$	bitweises Und
<i>or</i>	$x y$	bitweises Oder
<i>not</i>	y	bitweise Negation

Der zu berechnende Ausdruck wird aus Infix- in Postfixnotation umgeschrieben:

$$2 \ x - y \ 5 + \cdot = d$$

Dadurch kann er leicht mit Hilfe eines Stapels berechnet werden (siehe Abb. 7.6)

Abbildung 7.6: VM-Ausdruck

$$d = (2 - x) \cdot (y + 5)$$

```
push 2
push x
sub
push y
push 5
add
mult
pop d
```

7.2.4 Speicherzugriff, Speicheraufteilung und Speichersegmente

Speicherzugriff

Die virtuelle Maschine verwaltet bis zu 8 verschiedene (virtuelle) Speichersegmente (vgl. Java). Bisher haben wir immer auf den globalen Speicher zugegriffen. Für den Zugriff auf die virtuellen Speichersegmente gibt es andere Befehle:

- **push** segment index

Ablegen des Inhalts von `segment[index]` auf den Stapel

- **pop** segment index

Speichern des obersten Stapелеlements in `segment[index]`

Speicheraufteilung

Das Hauptziel der Virtuellen Maschine ist es, den Hauptspeicher des Hack-Systems mitzuverwalten.

Abbildung 7.7: RAM im Hack-System

RAM		
RAM[0 - 15]	virtual registers
RAM[16 - 255]	static variables
RAM[256 - 2047]	stack
RAM[2048 - 16383]	heap
RAM[16384 - 24575]	memory mapped I/O

Speichersegmente

- *local, argument, this, that*:

Direkte Abbildung auf festen Speicherbereich

Positionen im Speicher werden in `RAM[1..4]` gehalten (*LCL, ARG, THIS, THAT*)

- *pointer, temp*:

pointer wird auf $RAM[3..4]$ abgebildet (*this, that*)

temp wird auf $RAM[5..12]$ abgebildet

- *constant*:

Tatsächlich virtuell (kein Speicherbereich zugeordnet)

VM bearbeitet Zugriff von *constant_i*, in dem sie die Konstante *i* liefert

- *static*:

Verfügbar für alle Dateien mit Endung *.vm*

Statische Variablen werden ab $RAM[16]$ zugeordnet

7.2.5 Programmablauf (bedingte Anweisungen und Schleifen)

Label definieren (ähnlich, zur Assemblersprache) Marken im Programmtext, die z.B. als Sprungziel dienen können:

- *label c*: Definiert Marke im Programmtext, z.B. als Sprungziel
- *goto c*: Springt zu einer Marke im Programmtext (unbedingter Sprung)
- *if – goto c*: Springt zu einer Marke im Programmtext, wenn das oberste Stapелеlement nicht 0 ist (Element wird von Stapel entfernt)

Kapitel 8

Hochsprachen und Kompiler

8.1 Die Programmiersprache Jack

8.1.1 Allgemeine Syntax

- Jack-Programm ist eine Sammlung von Jack-Klassen
- Jack-Klasse: Sammlung von Jack-Unterprogrammen
- Jack-Unterprogramm:
 - Funktion
 - Methode
 - Konstruktor
- Zwingend: Eine Klasse *Main*, mit Funktion *main*

Abbildung 8.1: Jack-Programmierbare

```
class Main {  
  
    /* Sums up 1 + 2 + 3 + ... + n */  
    function int sum(int sum) {  
        var int i, sum;  
        let sum = 0;  
        let i = 1;  
        while (~(i > n)) { // Java: (!(i > n))  
            let sum = sum + i;  
            let i = i + 1;  
        }  
        return sum;  
    }  
  
    function void main() {  
        var int n, sum;  
        let n = Keyboard.readInt(Enter n: ``);  
        let sum = Main.sum(n);  
        do Output.printString("The result is: ");  
        do Output.printInt(sum);  
        do Output.println();  
    }  
  
} // Main
```

8.1.2 Datentypen und Speichieranforderung

Es gibt drei arten von Datentypen:

Basisdatentypen (primitive types)

- *int* 16-Bit-Ganzzahlen im Zweierkomplement

- *boolean* Boolescher Wert
- *char* Unicode-Zeichen

Abstrakte Datentypen (vom Betriebssystem oder Benutzer definiert)

- *String* (definiert durch Betriebssystem)
- *Fraction* (definiert durch Benutzer)
- *List* (definiert durch Benutzer)

Anwendungsspezifische Datentypen (vom Benutzer definiert)

- *CustomType*
- *AnotherCustomType*
- ...

8.1.3 Speichieranforderung

- Basisdatentypen wird bei Deklaration Speicher zugeordnet
- Objektvariablen wird bei Konstruktion (aufrufen des constructors) Speicher zugeordnet

8.2 Compiler (speziell für Jack)

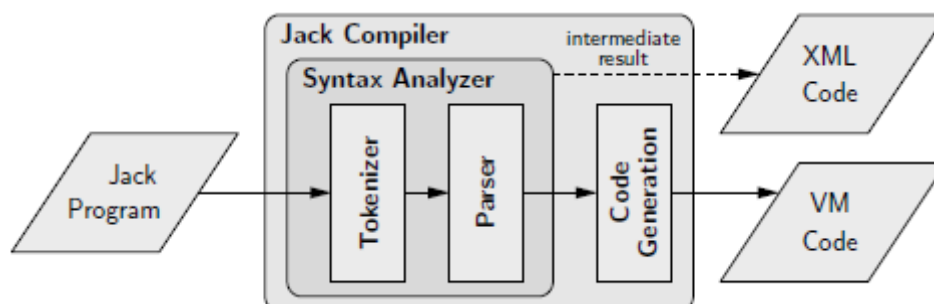
8.2.1 Architektur

Moderne Pbersetzer sind zweistufig

- 1. Stufe (front-end): von Hochsprache nach Zwischensprache
- 2. Stufe (back-end): von Zwischensprache nach Maschinensprache

8.2.2 Architektur, Lexikalische Analyse, Parsing

Abbildung 8.2: Jack-Compiler



Syntaxanalyse

Lexikalische Analyze (tokenizing)

Erzeugen eines Stroms von Sprachatomen
(token Stream)

- Entferne Leerzeich. & Kommentaren
- Erzeuge Liste von Sprachatomen

Parsen (parsing)

Zuordnen der Sprachatome (token) zu der
Sprachgrammatik

Abschließend:

XML-Ausgabe als Nachweis, dass Syntaxanalyse funktioniert

8.2.3 Kontextfreie Grammatiken

Jede (formale) Sprache kann durch eine *Grammatik* beschrieben werden. Eine solche Grammatik besteht aus *Produktionsregeln*, die angeben, wie aus Wörtern neue Wörter produziert werden.

In Programmiersprachen verwendet man *kontextfreie Sprachen*, die von *Stapel-/Kellerautomaten* erkannt (geparsed) werden können.

8.2.4 Parse-Bäume, Parsen durch rekursiven Abstieg

Zum Parsen eines Ausdruck kann ein sogenannter Parse-Baum erstellt werden. Er visualisiert, wie man aus einfachen Termen komplexe Formen erstellen kann

8.2.5 Jack-Syntaxanalyse

Auf der Grammatik aufbauend kann ein Syntax-Analysierer implementiert werden, der einen Quelltext lesen und mit der Grammatik vergleichen kann.

8.2.6 Codeerzeugung

Bei der Codeerzeugung geht es ums Nachbilden der Semantik des Quelltextes in der Zielsprache. Man kann diesen Prozess als Erweiterung des Syntax-Analysierers betrachten, nur dass statt passiven XML-Codes VM-Code erzeugt wird.

Kapitel 9

Rechnernetze

9.1 Das Internet

9.1.1 Struktur des Internet

DEFINITION: Internet - Wirte & Vermittlungsstellen

Das Internet ist ein Rechnernetzwerk, das etwa eine Milliarde Rechner rund um die Erde miteinander verbindet.

Die Rechner nennt man *Wirte (Hosts)* oder *Endsysteme (End Systems)*

Sie sind durch ein Netz von *Kommunikationsverbindungen (Communication Links)* und *Paketvermittlungen (packet switches)* verbunden

Netze, die Endsysteme mit der ersten Vermittlungsstelle (router) verbinden heißen Zugriffsnetze.

DEFINITION: Vermittlungsstellen

Die Kernstruktur des Internets ist ein Netz von miteinander verbundenen *Vermittlungsstellen (router)*, über die Nachrichten in Form von Datenpaketen übertragen werden.

9.1.2 Leitungsvermittlung

DEFINITION:

Bei der *Leitungsvermittlung* sind die Vermittlungsstellen durch Leitungen verbunden, von denen jeweils eine der zu vermittelnden Kommunikation zugeordnet wird.

Probleme

- Volle Übertragungskapazität der Leitung für eine Kommunikation reserviert, auch wenn sie nicht genutzt wird (Pause in der Übertragung).
- Zahl der gleichzeitig möglichen Übertragungen durch Anzahl der Leitungen beschränkt

Diese Probleme lassen sich durch Multiplexverfahren lösen

9.1.5 Übertragungskapazität

9.1.6 Verzögerungen

Übertragungsverzögerung

Entsteht, da Datenpaket in Vermittlungsstelle vor Weiterleiten erst vollständig empfangen werden müssen.

(→ Speichern und Weiterleiten)

- Zeit die vergeht, zwischen:
 - dem Senden/Empfangen des ersten und
 - dem Senden/Empfangen des letzten Bits eines Pakets
- Abhängig von
 - Übertragungskapazität
 - Größe der Datenpakete
- Entsteht in jeder Vermittlungsstelle erneut

Abbildung 9.4: Übertragungsverzögerung

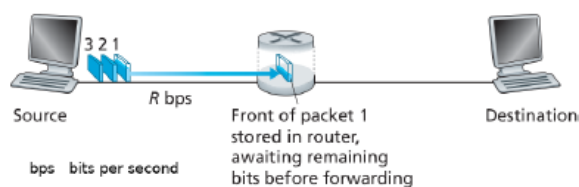
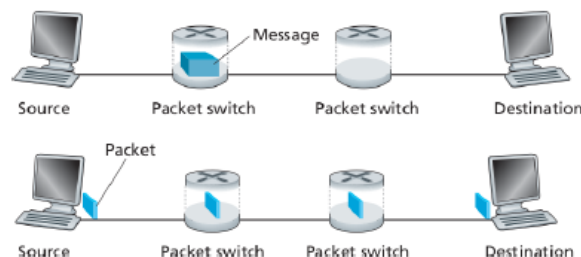


Abbildung 9.5: Vergleich - Paketgrößen

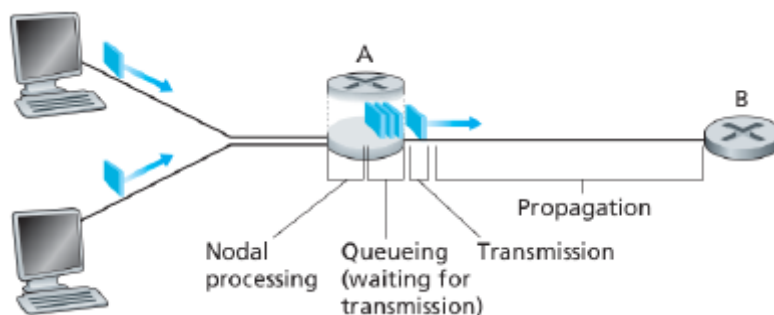


Wartezeit in Warteschlange

Entsteht, wenn ein empfangenes Paket nicht sofort weitergeleitet werden kann, weil die Ausgangsleitung belegt ist.

- Verzögerung:
 - wächst, wenn mehr Pakete empfangen werden, als übertragen werden können
 - schrumpft, wenn weniger Pakete empfangen werden, als übertragen werden können
- Länge ist endlich maximal, bei Überläufen kommt es zu *Paketverlust*

Abbildung 9.6: Verzögerungen



Verarbeitungszeit

Entsteht durch rechentechnische Verarbeitung eines Paketes in der Vermittlungsstelle

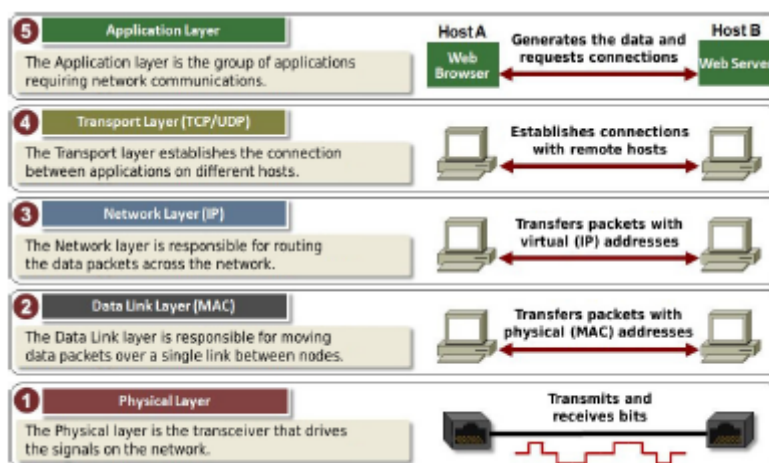
(→ entsteht in jeder Vermittlungsstelle ist aber meist gering)

9.2 Abstraktionsschichten & Protokolle

Die Funktionalität eines Rechnernetzes wird in (abstrakte) Schichten geteilt (Abb. 9.7). Das vereinfacht Organisation, Wartung, etc, wobei jede Schicht einen bestimmten Dienst mit Hilfe:

- eigener, schicht-interner Aktionen und
- von tieferen Schichten bereitgestellten Dienste

Abbildung 9.7: Abstraktionsschichten



9.2.1 Anwendungsschicht (Application Layer)

Client-Server vs Peer-to-Peer

Client-Server-Architektur

- Ein, immer verfügbarer Netzbediener, Server, der Anfragen von vielen anderen Wirten, den Clients erhält
- Typisches Beispiel: Webserver
- Bei stark nachgefragten Bedienern (z.B.: Google, Facebook) werden mehrere Bediener in Datenzentren zu einem virtuellen Bediener zusammengeschaltet

Peer-to-Peer-Architektur (P2P)

- Anwendung nutzt direkte Kommunikation zwischen Wirten, die temporär verbunden, ggf. auch nur temporär verfügbar sind
- Typisches Beispiel: BitTorrent
- Angenehme Eigenschaft: Selbstskalierbarkeit da jeder Empfänger auch wieder als Sender auftreten kann

HyperText Transfer Protocol - HTTP

DEFINITION: HyperText Transfer Protocol - HTTP

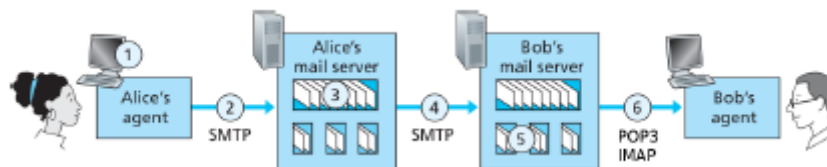
HTTP ist das Herzstück des sogenannten *World Wide Web (WWW)*, ein über das Internet aufrufbares System von elektronischen Hypertext-Dokumenten (Websites).

Simple Mail Transfer Protocol - SMTP

DEFINITION: Simple Mail Transfer Protocol - SMTP

SMTP dient der zustellung von elektronischer Post. Es überträgt nachrichten vom Email-Server des Senders zum Email-Server des Empfängers. (Abb. 9.8)

Abbildung 9.8: SMTP Diagramm



POP3 und IMAP

POP3

- ASCII-Protokoll, bei dem die Steuerung der Datenübertragung durch Kommandos geschieht
- Sehr eingeschränkt und erlaubt nur Auflisten, Abholen und Löschen von Nachrichten am Email-Server

IMAP

- Netzwerkprotokoll, das Netzwerkdateisystem für Emails bereitstellt
- Erweitert Funktionen von POP3, so dass Benutzer Emails, Ordnerstrukturen und Einstellungen auf den Email-Servern belassen können.

INFO: Simple Mail Access Protocol

SMAP ist ein Ansatz die Funktionalität von SMTP und IMAP zu vereinen.

File Transfer Protocol - FTP

DEFINITION: File Transfer Protocol

FTP dient zum Übertragen von Dateien von einem Wirt auf einen anderen.

9.2.2 Transportschicht (Transport Layer)

Die Hauptfunktionen der Transportschicht sind:

Logische Kommunikation Details der physischen Infrastruktur werden verborgen. Für Anwendungen ist es, als wären Wirte direkt verbunden

Verlässliche Zustellung Garantieren der Zustellung, z.B.: Erkennen von Paketverlusten und neue Übertragung

Integritätsprüfung Durch Prüfsummen werden Bitfehler erkannt und u.U. sogar korrigiert

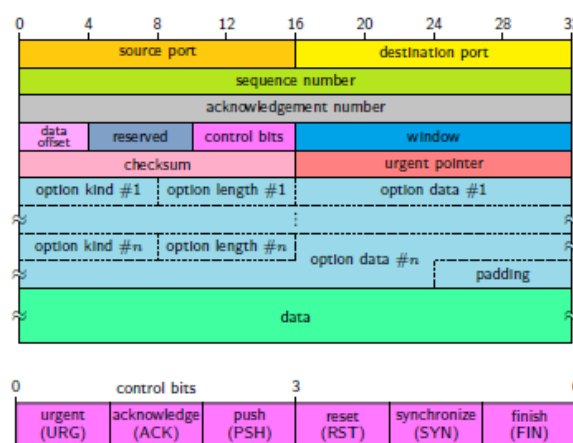
Blockierungskontrolle Verhindert überforderung der Leitung & Vermittlungsstellen, durch übermäßige Datenübertragung

Multiplex und Demultiplex Erweiterung der Kommunikation von Wirten auf Prozesse (über sog. Sockets)

Transmission Control Protocol - TCP

Die Protokolle der Transportschicht sind in den Wirten und Endsystemen, aber nicht in den Vermittlungsstellen implementiert. Nachrichten in der Anwendungsschicht werden in *Segmente* zerlegt (z.B.: TCP-Segmente, Abb. 9.9)

Abbildung 9.9: Struktur - TCP-Segment



9.2.3 Netzwerkschicht (Network Layer)

Die Hauptfunktionen der Netzwerkschicht sind:

Weiterleiten (forwarding)

Vermittlungsstellen müssen Datenpakete nach Ankunft an die passende Ausgangsleitung weiterleiten.

Wegeplanung (routing)

Die Netzwerkschicht bestimmt den Weg oder Pfad, den Pakete durch das Netzwerk nehmen, während sie vom Empfänger übertragen werden.

Um diese Funktionen zu implementieren, verwalten die Vermittlungsstellen *Weiterleitungstabellen* (Abb. 9.10)

Abbildung 9.10: Weiterleitungstabelle

destination	subnet mask	gateway /router	interface	metric (hops)
192.168.0.0	255.255.255.0	192.168.1.1	192.168.1.2	2
192.168.1.0	255.255.255.0	192.168.1.2	192.168.1.2	1
192.168.2.0	255.255.255.0	192.168.2.1	192.168.2.1	1
0.0.0.0	0.0.0.0	192.168.1.1	192.168.1.2	3

Internet Protocol - IP

9.2.4 Verbindungsschicht (Data Link Layer)

Ethernet IEEE 802.3

9.2.5 Physische Schicht (Physical Layer)