1 Hardwareentwurf

1.0 Prolog

Lernziele:

- Wie werden Rechner aufgebaut, warum werden sie so aufgebaut und wie funktionieren sie?
- Rechner werden heutzutage durch höchstintegrierte Schaltungen realisiert
- Stelle sicher, dass die erzeugten Maskendaten von Anfang an ein korrekt arbeitendes System liefern
- Schaltplaneingabe - Beschreibung mittels Hardwarebeschreibungssprache (HDL)
- Korrektheitsprüfung durch Simulation
- Logiksynthese und -minimierung
- Testmustererzeugung
- Formale Verifikation (Testbench)

1.1 Grundlegende Begriffe

1.1.1 Relationen und Funktionen

Relation $R \subseteq A \times B := \{(a,b) \mid a \in A \text{ und } b \in B\}$
$X \subseteq A$: R(X) Bild von X unter R
R ist partielle Funktion $\forall a \in Q(R) : \#R(a) \leq 1$

R heißt **totale Funktion** oder Abbildung $\Leftrightarrow$ D(R) = Q(R)
injektiv $f(a) = f(b) \Rightarrow a = b$ ($f(x)$ hat höchstens ein Urbild)
surjektiv $f(A) = B$ ($f(x)$ hat min ein Urbild)
bijektiv injektiv, surjektiv, total

Relationen auf Menge M
$R \subseteq M \times M$

- reflexiv $\forall a \in M\ aRa$
- symmetrisch $\forall a, b \in M\ aRb \Rightarrow bRa$
- antisymmetrisch $\forall a, b \in M\ aRb$ und $bRa \Rightarrow a = b$
- transitiv $\forall a, b, c \in M : aRb$ und $bRc \Rightarrow aRc$

Äquivalenzrelation: reflexiv, symmetrisch, transitiv
Ordnungsrelation: relfexiv, antisymmetrisch, transitiv

Operationen auf Relationen $\{(a,c) \mid (a,b) \in R \wedge (b,c) \in S\} = R \circ S \Leftrightarrow$ Z(R) = Q(S)

1.1.2 Zeichen und Worte

- Konkatentation: Hintereinanderreihung
- Substring: Teilwort

$v, w \in A^+ = A^* \backslash \{\epsilon\}$
v Präfix von w $\Leftrightarrow \exists u \in A^* \quad vu = w$
v Suffix von w $\Leftrightarrow \exists u \in A^* \quad uv = w$
v Infix von w $\Leftrightarrow \exists u \in A^* \quad avb = w$
XFix ist echt $\Leftrightarrow v \neq w$

Fortsetzung
$\varphi : A \to B^*$, dann $\varphi : A^* \to B^*$ mit $\varphi(\epsilon) := \epsilon$
Fortsetzung ist Homomorphismus (lineare Abbildung)

Kodierung
$\varphi : A \to B^*$ heißt Kodierung nur dann, wenn $\varphi : A^* \to B^*$ injektiv ist

1. $\exists n \in \mathbb{N} : \forall a \in A : |\varphi(a)| = n$
2. $\varphi(a)$ präfixfrei: $\forall a, b \in \varphi(a)$: a ist nicht Präfix von b

1.1.3 Sprachen

Syntax: welche Worte erlaubt sind
Semantik: Bedeutung der Worte
Sprache mit der Bedeutung $\varphi : A^* \to M$

Beispiel bin : $B^* \to \mathbb{N}_0$

$$\text{bin(u)} = \left\{ \begin{array}{ll} 0 & u = \epsilon \\ 2bin(u[1 : |u| - 1]) + u(|u|) & sonst \end{array} \right\}$$

$$\begin{aligned} bin(0100101) &= 2 * bin(010010) + 1 \\ &= 2^2 bin(01001) + 1 \\ &= 2^3 bin(0100) + 2^2 + 1 \\ &= 2^5 + 2^2 + 1 = 37 \end{aligned}$$

bin ist surjektiv (Induktion), aber nicht injektiv (bin(u) = bin(0u))

BNF(Backus Naur Form)
$< IntegerKonstante >::= [+ \mid -]\{0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\}+$
Bedingung:
Es gibt zu jedem verwendeten Wortmengebezeichner genau eine Definition

Exakte Definition bei komplexeren Sprachen sehr viel schwieriger und umfangreicher

$< typedeclaration >::= TYPE < identifier > IS\{< scalartypedef > \mid ...\}$
TYPE BIT IS $('0','1')$;

1.1.5 Was ist Hardware?

Hardware besteht aus Menge B parallel arbeitender **Bausteininstanzen** und einer Menge S von **Signalen**, die zur Kommunikation zwischen den Bausteininstanzen mit der Aussenwelt dienen
$b \in B$ wirkt auf Signale über ihre **Ports** P(b)
$s \in S$ verbindet eine Menge P(s) von Ports

Man wird natürlich viele Bausteininstanzen benutzen, die das gleiche Verhalten haben

entity:	Definition einer Bausteinschnittstelle (Ports)
component:	Definition eines Bausteins zur Benutzung
Instanzierungen:	Erzeugung von Instanzen von Komponenten und deren Verdrahtung durch Anweisungen

Jeder Port $p \in P$ ist einem Signal sig(p) und einer Bausteininstanz b(p) eindeutig zugeordnet. ($\rightarrow$ Verbindungen)
Signalwert (Waveform): $\beta(s) : T \rightarrow V$ (Zeit $\rightarrow$ Werte)

1.2 Grundbausteine der Digitaltechnik

1.2.1 Darstellung von Zeichen

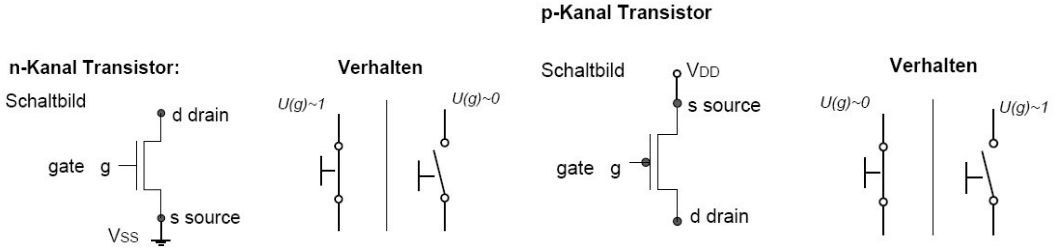
Wir müssen Zeichen durch Zustände physikalischer Systeme realisieren
Diese Binärkodierung ist notwendig und hinreichend um alles zu kodieren
Gute Störsicherheit und Zuverlässigkeit

1.2.2 CMOS Technologie

C: complementary: sich gegenseitig ergänzend
MOS (Metal Oxide Semiconductor)
Transistoren als spannungsgesteuerte Schalter
Interpretiere Spannungen U(x) zwischen x und dem Bezugspol V_{SS} als Werte aus B (Versorgungspol V_{DD})

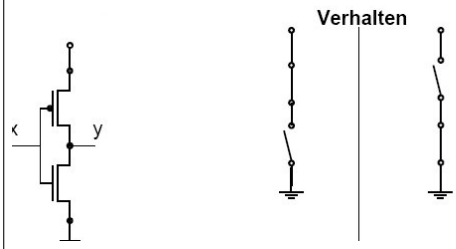
- U(x) $V_{SS}(= 0V)$ interpretiere als 0
- U(x) V_{DD} interpretiere als 1

MOS FETs als Schalter (FET: Feldeffekttransistoren)



n-Kanal und p-Kanal Transistor verhalten sich als Schalter komplementär zueinander

Der CMOS Inverter (Negation)



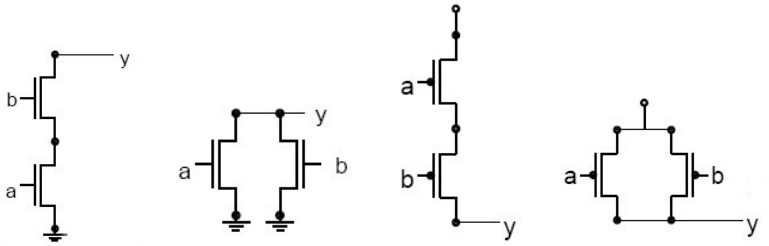
Für diese Schaltung nutzt man dieses Zeichen 

Schaltzeiten eines Inverters
Es gibt keine abrupte Veränderung
• Widerstand (Ohmsches Gesetz) $U(t) = R \cdot I(t)$
• Kapazität (Kondensatorgleichung) $Q(t) = C \cdot U(t)$
negativ exponentieller Verlauf der Spannung
Abfalls/Anstiegszeit: t_{dhl} und t_{dlh} (dhl - delay-high-low)
mit $U_y(t) = A \cdot e^{-\frac{t}{RC}}$ gilt $t_{dhl} = 2RC$, $t_{dlh} = RC$

1.2.4 Einfache Verknüpfungen

Die leitenden Verbindungen bestehen unter gegenseitigem Ausschließ
Die Verschaltungen im n-Kanal und p-Kanal Teil sind dual zueinander
(Serien $\leftrightarrow$ Paralllel)

- Serienschaltung gegen V_{SS} nicht y $\Leftrightarrow$ a und b
- Parallelschaltung gegen V_{SS} nicht y $\Leftrightarrow$ a oder b
- Serienschaltung gegen V_{DD} y $\Leftrightarrow$ nicht (a oder b)
- Parallelschaltung gegen V_{DD} y $\Leftrightarrow$ nicht (a und b)



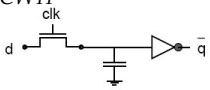
NAND und NOR

nand(a,b) = nicht (a und b) 
nor(a,b) = nicht (a oder b) 

Diese CMOS-Komplexgatter kann man beliebig (theoretisch) weitertreiben

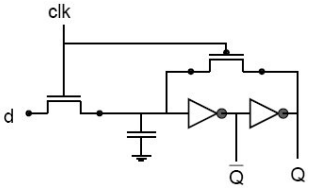
1.2.5 Speicherelemente (0,1)

Dynamisches Daten-Latch (D-Latch)
Durch Menge der partiellen Funktionen F([0:k-1],O) gibt es genau zwei definierte Zuständen
Man kann CMOS-Schaltung zum Speichern nutzen.
Einfügen von **clk** zum Ändern von Zuständen
t_{CWH} auf 1: Übernahmetick

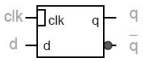


clk = 1: d wird mit dem Eingang des Inverters verbunden und lädt Kapazität auf
clk = 0: d und die Kapazität entkoppelt und Wert bleibt unabhängig von d erhalten
Problem: Leckströme am Eingang des Inverters (Zeitbedingung ⇒ dynamisch)

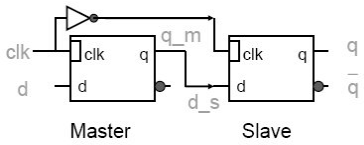
⇒ **statisches Daten-Latch** (mit Halteschaltung)



Schaltbild:



Master/Slave Latch Gefahr, dass d sich in der 1 Phase von clk ändern kann (d transparent mit q verbunden)
⇒ Übernahmezeitpunkt eng um die 1/0 Flanke des Taktes clk:

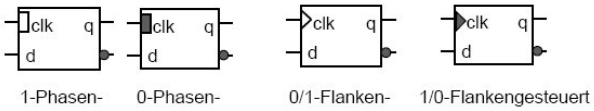


clk = 1: Master Latch schaltet d transparent nach q_m
clk = 0: Master verriegelt, Slave schaltet q_m nach d_m

Master-Slave-Latch-Timing

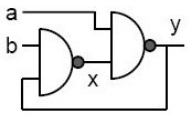
- delay t_d bis Ausgang übernimmt
- setup t_s Zeit, die d vor Flanke stabil sein muss
- hold t_h Zeit, die d nach Flanke stabil sein muss

Latches können zustands- oder flankengesteuert sein



Schieberegister mit Latches: flankengesteuertes Latch ist ein Element des SR
(Kontrolle, dass in 1. Takt nur um 1. Stelle verschoben wird)

1.2.6 Bistabile Schaltungen (mit Logikgattern)



y = nicht (a und nicht(b und y))

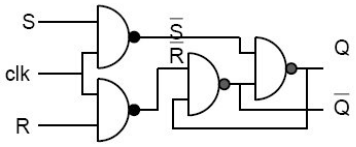
a=b=1: y = nicht (1 und nicht(1 und y)) = y **hält y**

a=0, b=1: y = nicht (0 und nicht(1 und y)) = 1 **setzen**

a=1, b=0: y = nicht (1 und nicht(0 und y)) = 0 **rücksetzen**

a=b=0: y = nicht (0 und nicht(0 und y)) = 1 **unbenutzt**

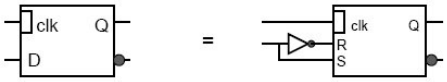
Wir nennen diese Schaltung **Basis-R/S FlipFlop**



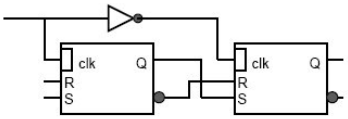
Für clk = 0 werden SZ und RZ auf nicht(0) = 1 gezwungen
Schaltsymbol:

Schaltsymbol:

Offenbar kann man aus dem RS FLipFlop leicht ein
D-Latch bauen:

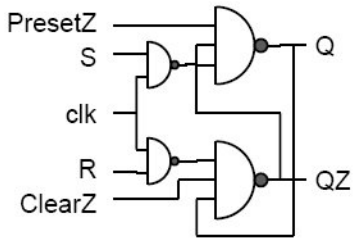


Und nach dem Master/Slave Prinzip ein flankengesteuertes
RS-FlipFlop:



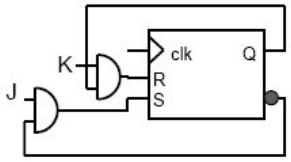
Synchrone und asynchrone Kontrolle

Ausgang Q kann synchron (Takt) oder asynchron (Kontrollleitungen) manipuliert wrdn



JK-Flipflop kippt für (J,K) = (1,1) den aktuellen Zustand

Beispiel: das JK FlipFlop



0	0	Q	0	0	Q
0	1	Q	Q	0	0
1	0	Q	0	QZ	1
1	1	Q	Q	QZ	QZ

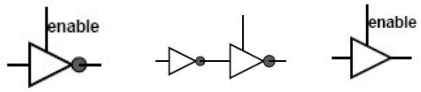
Man gibt RS-FlipFlop durch vorschalten zusätzliche Bedeutung

JK-FlipFlop:
z.B. zur Halbierung des Taktes, da nur an einer Flanke (0/1) signal invertiert werden kann

1.2.7 Tristate Treiber und Busse

bis jetzt strenge Unterscheidung zwischen Eingängen (Transistor gates) und Ausgängen (getrieben durch leitende Pfade nach V_{SS} oder V_{DD})
⇒ Schaffe Möglichkeit, dass bei n Komponenten, jede mit jeder direkt Kontakt aufnehmen kann
(2n(n-1) Verbindungen)
⇒ Ports die Eingang oder Ausgang sein können

Tristate Treiber verhält sich neutral gegen beide Versorgungspole
(mittels hohem Widerstand ⇒ Abkopplung vom Bus)
neutrales Verhalten: Inverter mit einem Schalter (enable)
Weiterer Inverter liefert nichtinvertierenden Treiber



⇒ Verhalten: 0,1 oder neutral
⇒ Schreiben auf Bus wird durch Tristate Treiber geregelt **Busse - Eigenschaften**

- Möglichkeit viele Komponenten direkt zu verbinden
- bidirektionale Datenleitungen
- unidirektionale Kontrollleitungen
- Taktleitungen (synchrone Busse)
- Protokoll (wann darf wer lesen oder schreiben?)

Es muss Signale geben, die von mehreren Prozessen beschrieben werden dürfen
⇒ mehrere Treiber weisen Signal einen Wert zu
⇒ dieser Wert muss aus allen Werten berechnet werden (aufgelöster Wert)

(Resolved) Type Utri_state_logic IS ('Z','0','1','X')

		–|Z	|0	|1	|X	
Z	– hochohmiger Zustand	((Z,	0,	1,	X),	–Z
0	– Logikwert 0	(0,	0,	X,	X),	–0
1	– Logikwert 1	(1,	X,	1,	X),	–1
X	–Fehlerbedingung	(X,	X,	X,	X)),	–X

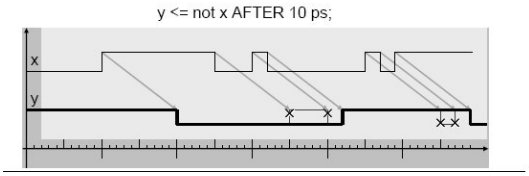
Treiber
Jede Signalinstanz eines unaufgelösten Typs wird von höchstens einer Prozessinstanz zugewiesen
⇒ ein Signal darf in einem Prozess höchstens eine Zuweisung erhalten
⇒ Treiber des Signals

Verzögerungsarten bei Signalen

- Einfügen **vor** schon bestehende Ereignisse
  $\Rightarrow$ führt zum Löschen aller nachfolgenden Ereignisse
- Einfügen **hinter** schon bestehende Ereignisse
  - träge Verzögerung (intertial delay), Default
  - nichtträge Verzögerung (transport delay)

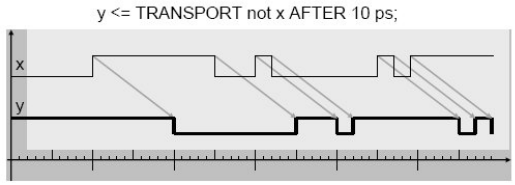
Intertial

bei Einfügen eines Ereignisses werden alle vorhergehenden Ereignisse zwischen und jetzt und dem Einfügezeitpunkt gelöscht
$\Rightarrow$ keine Signalpulse die kürzer sind als Verzögerung



Transport

alle vorhergehenden Ereignisse zwischen aktuellem Zeitpunkt und Einfügezeitpunkt bleiben erhalten
$\Rightarrow$ kurze Signalpulse, die ggf. physikalisch gar nicht mehr entstehen



Fazit: Variablen versus Signale

Deklaration

- Variablen werden in Prozessen / Unterprogrammen deklariert
- Signale können nicht in Prozessen / Unterprog. deklariert werden

Verwendung

- Variablen: Abspeichern temporärer Werte
- Signale: Verbindungen in der Hardware

Wertzuweisung

- Variablen Wertzuweisung erfolgt sofort
- Signale: nachdem der zuweisende Prozess *terminiert* ist

1.3 Verbände und boolsche Algebra

Partielle Ordnung
Relfexivität, Antisymmetrie, Transitivität

Poset (partially ordered set)
Menge M, partielle Ordnung $\leq$, (M,$\leq$) ist Poset

Halbverbände
Poset(M,$\leq$) heißt oberer (unterer) Halbverband
$\Leftrightarrow$
$\forall\{a,b\} \subseteq M \; \exists\, c \in M : c = \sup\{a,b\}$
$(\forall\{a,b\} \subseteq M \; \exists\, c \in M : c = \inf\{a,b\})$

vollständiger Halbverband
(M,$\leq$) heißt vollständig $\Leftrightarrow$
Zu jeder Teilmenge $U \subseteq M$ existiert das Supremum (Infimum)

Verband
Poset(M,$\leq$) heißt Verband $\Leftrightarrow$
(M,$\leq$) ist (vollständiger) oberer und unterer Halbverband

Rechenregeln in Verbänden

(V1)	$a \vee b = b \vee a$	
	$a \cdot b = b \cdot a$	(Kommutativität)
(V2)	$a \vee (b \vee c) = (a \vee b) \vee c$	
	$a \cdot (b \cdot c) = (a \cdot b) \cdot c$	(Assoziativität)
(V3)	$a \vee (a \cdot b) = a$	
	$a \cdot (a \vee b) = a$	(Absorption)

Dualität: jeweils Vertauschung von "und" und "oder" möglich

Distributiver Verband

(V4)	$a \vee (b \cdot c) = (a \vee b) \cdot (a \vee c)$	
	$a \cdot (b \vee c) = (a \cdot b) \vee (a \cdot c)$	(Distributivität)

$\cdot$ bindet stärker als $\vee$

Sätze
$(M_1, \leq_1)$ und $(M_2, \leq_2)$ (vollst.) Verbände. Dann auch:

- $(M_1 \times M_2, \leq_{1,2})$ $(a,b) \leq_{1,2} (c,d) \Leftrightarrow a \leq_1 c$ und $b \leq_2 d$
- $(Abb(X, M_1), \leq)$ mit $f \leq g \Leftrightarrow \forall x \in X : f(x) \leq_1 g(x)$

Boolesche Algebra
$(M, \vee, \cdot, \neg)$ heißt boolsche Algebra, dann und nur dann, wenn

(A1)	$(M, \vee, \cdot)$ ist distributiver Verband
(A2)	$\forall a,b \in M : a \vee b \cdot \neg b = a$
	$\forall a,b \in M : a \cdot (b \vee \neg b) = a$

Eigenschaften boolscher Algebren

Idempotenz
$a \vee a = a$ und $a \cdot a = a$

Es existiert genau ein "1-Element" und ein "0-Element" die zueinander dual sind
$a \vee 0 = a \qquad a \cdot 1 = a \qquad a \cdot 0 = 0 \qquad a \vee 1 = 1$

Komplement
$a \vee b = 1$, $a \cdot b = 0$ nur für b = $\overline{a}$ ist Komplement von a

DE MORGAN

$\overline{a \vee b} = \overline{a} \cdot \overline{b}$	$\overline{a \cdot b} = \overline{a} \vee \overline{b}$

$\overline{0} = 1 \qquad \overline{1} = 0$

Sätze
M_1, M_2 b.A., A Menge

- $(2^A, \cup, \cap, \neg)$
- $(M_1 \times M_2, \vee, \cdot, \neg)$
- $Abb(A, M_1)$

sind wieder b.A.

Digitaltechnik
$\{0,1\}$ kleinste boolsche Algebra B
$\Rightarrow B^n = B \times \cdots \times B$ wieder b.A.
$S_{n,k} = Abb(B^n, B^k)$ b.A. (k-stellige Schaltfunktion)

Atome

(At1)	$a \neq 0$
(At2)	$\forall b \in M : a \cdot b \in \{a, 0\}$

$\Rightarrow$ a Atom
$a \in M$ ist Atom $\Leftrightarrow \forall b \in M : b \leq a \Rightarrow$ (b = 0 oder b = a)
a,b Atome, dann $ab = 0$
$f \in M \Rightarrow f = \bigvee a$ mit $a \in At(M)$ und $a \cdot f \neq 0$

Minterm zu p oder Punkt p
Für Punkt $p \in B^n$ sei x^p die Funktion, die nur auf p den Wert 1 hat
Dann ist x^p Atom mit $x^p(q) = \left\{ \begin{array}{ll} 1 & \text{für p = q} \\ 0 & \text{für } p \neq q \end{array} \right\}$

ON-Set
$On(f) := \{a \in At(M) \mid af \neq 0\} \subseteq At(M)$
$\Rightarrow$ alle Elemente als Veroderung ihres ON-Sets darstellbar
$f = \bigvee_{p, f(p)=1} x^p$ (disjunktive Normalform von f - eindeutige Darstellung der Fkt.)

M b.A., Dann gibt es $n \in \mathbb{N}$ mit $\#M = 2^n$

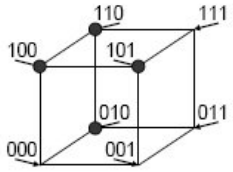
1.4 Schaltfunktionen und boolsche Ausdrücke

$f : B^n \to B^k$ heißt Schaltfunktion
$S_n := Abb(B^n, B)$ ist boolsche Algebra
Jede Schaltfunktion aus S_n kann mittels $\{x_1, ..., x_n\}$ und den Operatoren $\cdot, \vee, \neg$ dargestellt werden

Disjunktive Normalform

$On(f) := \{p \mid f(p) = 1\} = \{p \mid f \cdot x^p = x^p\}$	ON-Set der Funktion
$f = \bigvee_{p \in On(f)} x_1^{p_1} \dots x_n^{p_n}$	disjunktive Normalform

Anschauung
Man kann die Vektoren $p \in B^n$ als Eckpunkte eines n-dimensionalen Würfels auffassen



x1	x2	x3	f
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	1
1	1	1	0

ON(f) = $\{010, 100, 101, 110\}$
$f = x^{010} \vee x^{100} \vee y^{101} \vee x^{110} = x_1^0 x_2^1 x_3^0 \vee x_1^1 x_2^0 x_3^0 \vee x_1^1 x_2^0 x_3^1 \vee x_1^1 x_2^1 x_3^0 =$
$\overline{x_1} x_2 \overline{x_3} \vee x_1 \overline{x_2} \overline{x_3} \vee x_1 \overline{x_2} x_3 \vee x_1 x_2 \overline{x_3}$

Literale
$Y = \{y_1, ..., y_n\}$, $y_i, \overline{y_i}$ sind Literale

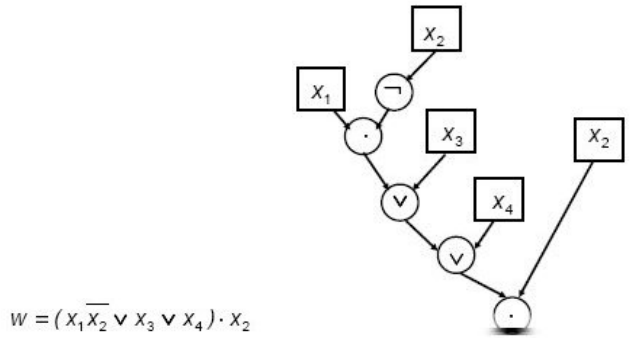
Boolsche Ausdrücke
mit impliziter Klammerung von links nach rechts

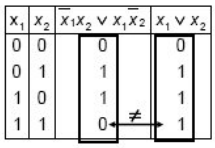
Ausdrücke:	eindeutig zerlegbarin Ausdruck und Term ($\vee$)
Terme:	eindeutig zerlegbar in Produkt aus Term und Faktor ($\cdot$)
Faktoren:	Konstakte, Literale, geklammerte Ausdrücke

Normalformen

Disjunktive Normalform	Konjunktive Normalform
$w_1 \vee ... \vee w_n$, w_i ist Produkt	$w_1 \dots w_n$, w_i ist Klausel: $w_i = c_1 \vee ... \vee c_n$

Syntaxbäume





x_1	x_2	$\overline{x}_1x_2 \vee x_1\overline{x}_2$	$x_1 \vee x_2$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0 ≠	1

$\overline{a}b \vee a\overline{b} = a \oplus b$ EXOR (exklusives oder)
$a \equiv b = \overline{a \oplus b}$ Äquivalenz ($ab \vee \overline{a}\overline{b}$)

Syntheseproblem
Finde kürzesten Ausdruck für boolschen Ausdruck
$\Rightarrow$ bis heute keine (bis auf erschöpfende) Lösungsmtehode

Meist interpretieren wir als die Variablen x_i als Projektion auf die i-te Komponente
$x_i(p) = p_i$

Substitution	Beispiel
$w \in BA(Y), a_1, ..., a_k \in BA(Y)$	$w = y_1\overline{y_2} \vee y_3$
$w(y_1 = a_1, ...y_k = a_k)$	$w(y_1 = y_1y_3, y_2 = \overline{y_3}) = (y_1y_3)\overline{\overline{y_3}} \vee y_3$

$\Rightarrow$ Substitutionen sind von der Reihenfolge abhängig

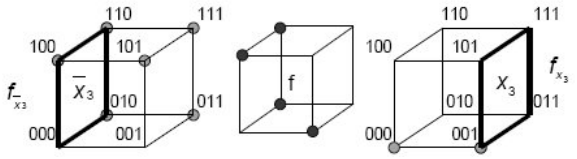
Kofaktoren
$f_{x_i}(x_1, ..., x_n) = f(x_1, ..., x_{i-1}, 1, x_{i+1}, ..., x_n)$
$f_{\overline{x_i}}(x_1, ..., x_n) = f(x_1, ..., x_{i-1}, 0, x_{i+1}, ..., x_n)$
heißen Kofaktoren von f nach x_i bzw. $\overline{x_i}$
$\Rightarrow$ Hängen nicht mehr vom Wert der i-ten Komponenten ab
f ist unabhängig von $x_i \Leftrightarrow f(x_i) = f(\overline{x_i})$

Support
Variablen von denen f abhängt $supp(f) = \{x_i \mid f_{x_i} \neq f_{\overline{x_i}}\}$

Syntaktischer Support
Menge der Variablen, die in w vorkommen: $ssupp(w) \supseteq supp(I(w))$

Shannon Expansion
$f = \overline{x_i}f_{\overline{x_i}} \vee x_if_{x_i}$
Funktion kann durch ihre Kofaktoren dargestellt werden

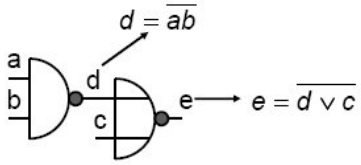
Anschauung
Kofaktor: Restriktion auf Seitenfläche



2 Hardwareentwurf - die Disziplinen

2.1 Schaltkreise (SK)

Wir können nun digitale Schaltkreise mit boolschen Gleichungen beschreiben



Schaltkreis über **Bausteinsystem** A besteht aus einer Menge von

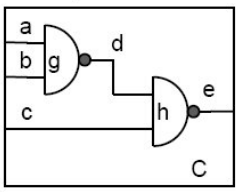
G	**Bausteininstanzen** (Gates)
$P(C) = \{C.a_1, ..., C.a_n\}$	von äußeren Anschlüssen (**Ports**)
N	**Netzen**

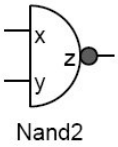
Zu jeder Bausteininstanz gehören eine Menge
P(g) = $\{g.a_1, ..., g.a_k\}$ von Anschlüssen (Konnektoren, Ports) und ein Baustein comp(g) in A den g instanziert.

- $g \neq h \Rightarrow P(g) \cap P(h) = \{\}$
- $s \in N : s \subseteq P(C) \cup \bigcup_{g \in G} P(g)$

  Dabei liegt jeder Anschluss g.x auf genau einem Netz, dem Netz zum Anschluss. Wir nennen dies net(g.x)

Beispiel:





Schaltkreis C
Gatter G = $\{g, h\}$, comp(g) = Nand2 = comp(h)
Netze N = $\{a, b, c, d, e\}$
Ports P(C) = $\{C.a, C.b, C.c, C.e\}$
P(g) = $\{g.x, g.y, g.z\}$ P(h) = $\{h.x, h.y, h.z\}$
d=$\{g.z, h.x\}$ e = $\{h.z, C.e\}$
net(g.y) = b

Orientierte Schaltkreise
orientiert: Unterscheidung zwischen Ein- und Ausgängen
Sind alle Bausteinsysteme orientiert ⇒ orientierter SK
P(g) = In(g) ∪ Out(g)
In(g) ∩ Out(g) = {}
In(g) ≠ {} ≠ Out(g)

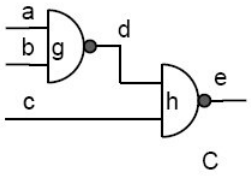
Es gibt auch nützliche nicht-orientierte Bausteine (Schalter)

Orientierte wohlgeformte Schaltkreise

(OC 1) $\forall\, s \in N\#(s \cap (In(C) \cup \bigcup_{g \in G} Out(g))) = 1$
⇒ höchstens ein Ausgang einer Bausteininstanz gehört zu Netz s
⇒ Dieser Ausgang ist der Treiber von s: treiber(s)
⇒ Hat Baustein nur einen Ausgang kann auch $g \in G$ der Treiber von s sein

(OC 2) $\forall\, s \cap \bigcup_{g \in G} Out(g) = \{\} \Rightarrow In(C) \cap s \neq \{\}$
Netze, die nicht von Ausgangsports getrieben werden, müssen von außen zugänglich sein
⇒ Primäreingang s zu C.x

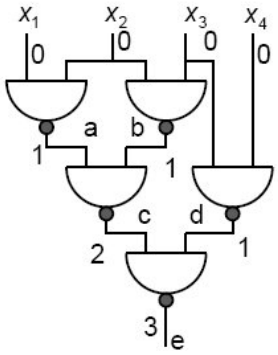
Pfad
Netze $s_1, ..., s_k$
$\forall\, 1 \leq i < k\ \exists\, g:\ In(g_i) \cap s_i \neq \{\} \neq Out(g_i) \cap s_{i+1}$



Pfade: d,e
a,d,e
b,d,e
c,e

Rückkopplungsfreie Schaltkreise
Für alle Pfad $s_1, ..., s_k$ gilt: $s_i = s_j \Rightarrow i = j$
$\Rightarrow$ alle Pfade haben Länge kleiner gleich #N

Tiefe, Länge des längsten Pfades
tiefe(s) = $\left\{ \begin{array}{ll} 0 & s \cap In(c) \neq \{\} \\ 1 + max\{tiefe(r) \mid r \cap In(gate(treiber(s))) \neq \{\}\} & sonst \end{array} \right\}$
Primäreingang: Tiefe 0

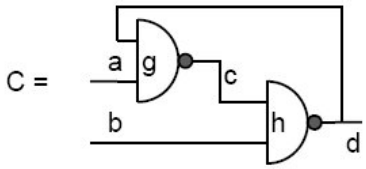


Netz	Tiefe
x_1	0
x_2	0
x_3	0
x_4	0
a	1
b	1
c	2
d	1
e	3

Statisches Verhalten
Bausteinsystem A ist kombinatorisch $\Leftrightarrow$ orientierte Anschlüsse, Ausgang y kann boolsche Funktion zugeordnet werden

Lokale Funktion
Jedem Netz s wird seine lokale Funktion zugeordnet
C[s]=C[s]$(a_1, ..., a_k) = y_i(..., x_i = net(g.x_i), ...)$

Charakteristische Funktion
$\chi_c = \prod_{s \in N} (s \equiv C[s])$
stabil, Belegung jedes Signals $\widehat{=}$ Auswertung der lokalen Funktion



$\chi_c = (c \equiv \overline{ad})(d \equiv \overline{bc})$
(a,b,c,d) = (1,0,1,0)
Instabil, da $\chi_c(1,0,1,0) = (1 \equiv \overline{10})(0 \equiv \overline{01}) = (1 \equiv 1)(0 \equiv 1) = 1 \cdot 0 = 0$
(a,b,c,d) = (0,1,1,0)
Stabil, da $\chi_c(0,1,1,0) = (1 \equiv \overline{00})(0 \equiv \overline{11}) = (1 \equiv 1)(0 \equiv 0) = 1 \cdot 1 = 1$
(a,b,c,d) = (1,1,$\overline{y}$,y)
Stabil, da $\chi_c(1,0,\overline{y},y) = (\overline{y} \equiv \overline{1y})(y \equiv \overline{1\overline{y}}) = (\overline{y} \equiv \overline{y})(y \equiv y) = 1 \cdot 1 = 1$

2.2 Entwurf kombinatorischer Schaltkreise

2.2.1 Kombinatorische Schaltkreise

Kombinatorischer Schaltkreise C

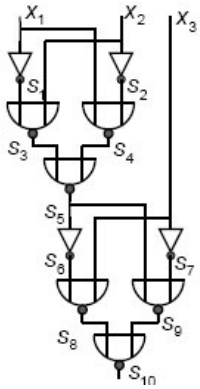
- Baustein in C kombinatorisch
  (orientiert & Zuordnung einer boolschen Funktion möglich)
- C wohlgeformt
- C rückkopplungsfrei

Globale Funktion zu Signal s
Jedes Signal als Funktion über den Eingängen auffassen
$F[s] = C[s](s_{i_1} = F[s_{i_1}], ..., s_{i_k} = F[s_{i_k}])$

topologische Sortierung der Signale
$s_1, ..., s_m$: erst $s_1, ..., s_n$ Primäreingänge, dann nach Tiefe

stabile Belegung
C komb. SK. Dann gibt es zu jeder Belegung der Primäreingänge $p_1, ..., p_n$ eine stabile Belegung $p_1, ..., p_m$ mit $p_i = F[s](p_1, ..., p_n)$
(Erst: Tiefe 0, dann Tiefe > 0)

Zusammenfassung
Jede Schaltfunktion ist durch einen boolschen Ausdruck über den Primäreingängen darstellbar
$\Rightarrow$ sukzessive Substitution und Vereinfachung der Ausdrücke
$\Rightarrow$ Ausdruck wächst sonst exponentiell mit Größe der Schaltung
Jede Schaltfunktion $f \in S_n$ ist eindeutig darstellbar durch ihre disjunktive NF:
$f = \bigvee_{p \in B^n, f(p)=1} x_1^{p_1} \dots x_n^{p_n}$



$F[s_1] = C[s_1](x_1 = F[x_1]) = \overline{x_1},\ F[s_2] = \overline{x_2}$
$F[s_3] = C[s_3](s_1 = F[s_1], x_2 = F[x_2])$
$= \overline{F[s_1] \vee F[x_2]} = \overline{\overline{x}_1 \vee x_2}$
$F[s_4] = \overline{x_1 \vee \overline{x}_2}$
$F[s_5] = \overline{\overline{\overline{x}_1 \vee x_2} \vee \overline{x_1 \vee \overline{x}_2}}$
$F[s_6] = \overline{\overline{\overline{\overline{x}_1 \vee x_2} \vee \overline{x_1 \vee \overline{x}_2}}},\ F[s_7] = \overline{x_3}$
$F[s_8] = \overline{\overline{\overline{\overline{\overline{x}_1 \vee x_2} \vee \overline{x_1 \vee \overline{x}_2}}} \vee x_3}$
$F[s_9] = \overline{\overline{\overline{\overline{\overline{x}_1 \vee x_2} \vee \overline{x_1 \vee \overline{x}_2}}} \vee \overline{x_3}}$

2.2.2 Konstruierbarkeit

A : kombinatorisches Bausteinsystem mit kombinatorischen Schaltkreisen Cand, Cnot bzw. Cor, Cnot gibt mit Eingängen a,b und Ausgang y, so dass

$$\text{F[y]} = \left\{ \begin{array}{cc} a \cdot b & \text{in Cand} \\ a \vee b & \text{in Cor} \\ \overline{a} & \text{in Cnot} \end{array} \right\}$$

Dann ist jede Schaltfunktion auf einem Ausgang eines entsprechenden Schaltkreises über A realisierbar

(es genügt sogar nur ein Nand2 Baustein)

Syntheseproblem für kombinatorische Schaltkreise

Geg: Schaltfunktion und komb. Bausteinsystem
Ges: Kombinatorischer Schaltkreis C, der Schaltfunktion berechnet mit minimaler Laufzeit
$\Rightarrow$ bis heute kein exaktes Lösungsverfahren

2.2.3 Disjunktive Formen

Disjunktive Form: $f = f_1 \vee ... \vee f_m$, wobei f_i Produkte

Minimierungsverfahren: Quine & McCluskey
mit Und- und Oder-Stufe $\Rightarrow$ 2-stufiger Schaltkreis
$\Rightarrow$ durch Vereinfachen der einzelnen Schaltfunktionen

$$\begin{aligned} f_1 &= x_1\overline{x_2}\overline{x_3} \vee x_1\overline{x_2}x_3 \vee x_1x_2\overline{x_3} \vee x_1x_2x_3 \\ &= x_1\overline{x_2}(\overline{x_3} \vee x_3) \vee x_1x_2(\overline{x_3} \vee x_3) \\ &= x_1\overline{x_2} \vee x_1x_2 = x_1 \end{aligned}$$

Syntheseproblem für 2-stufige Schaltungen

Geg: Schaltfunktion $f \in S^D_{n,k}$
Ges: disjunktive Form, Darstellung zu f die minimale Kosten hat

Lösungsskizze:

- Minimiere die Formel der rechten Seite einer Signalzuweisung
- Erzeuge zu einer Funktion von bitvector(n) nach bitvector(m) eine disjunktive Form

Redundanz
I_D Interpretation über S_n^D und r Ausdruck mit ON(I(r)) = $B^n \backslash D$, g,h boolsche Ausdrücke
$g \equiv_{I_D} h \Leftrightarrow g \vee r \equiv_I h \vee r$
Wir nennen eine Funktion r, deren ON-Set genau aus den Elementen außerhalb des Definitionsbereichs D einer partiellen Funktion f besteht auch Redundanz von f

Implikanten

Geg: disjunktive Formen r,f
Ges: g minimaler Kosten mit $g \vee r = f \vee r$

Ein Produkt p heißt **Implikant** von $f \in S_n^D \Leftrightarrow p(f \vee r) = p$
($\Leftrightarrow p \leq f \vee r \Leftrightarrow p \vee f \vee r = f \vee r$)

Sei $g \vee r = f \vee r$ für eine disjunktive Form g und eine partielle Funktion f, dann ist jedes g_i Implikant von f

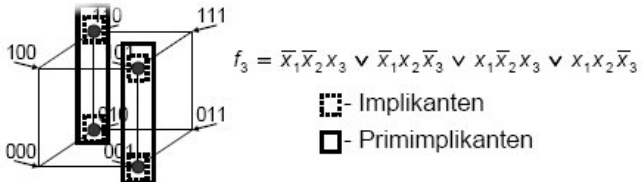
Primimplikanten
$\Rightarrow$ nur disjunktive Formen untersuchen, die ausschließlich aus Implikanten bestehen
Ein Implikant p einer partiellen Funktion f heißt **Primimplikant** von f, genau dann, wenn es keinen Implikanten $p' \neq p$ von f gibt, mit
$p \cdot p' = p$ ($\Leftrightarrow p \leq p' \Leftrightarrow p \vee p' = p'$) (Es gibt keinen "größeren" Implikanten)

Beispiel:
$f_3 = \overline{x_1}\overline{x_2}x_3 \vee \overline{x_1}x_2\overline{x_3} \vee x_1\overline{x_2}x_3 \vee x_1x_2\overline{x_3} = \overline{x_2}x_3 \vee x_2\overline{x_3}$
Minterme sind Implikanten. Aber auch $\overline{x_2}x_3$, denn
$\overline{x_2}x_3 \cdot f_3 = \overline{x_1}\overline{x_2}x_3 \vee 0 \vee x_1\overline{x_2}x_3 \vee 0 = \overline{x_2}x_3$
Prim? $p \in \{\overline{x_2}, x_3, 1\}$

- 1 fällt weg, da f nicht konstant
- $\overline{x_2}f_3 = \overline{x_2}x_3$
- $x_3 \cdot f_3 = \overline{x_2}x_3$

$\Rightarrow \overline{x_2}x_3$ Primimplikant

Geometrische Anschauung (Würfel)
Implikant von f ist ein Unterwürfel, dessen Ecken nur aus dem On-Set und der Redundaz der Fkt. sind
Ein Primimplikant ist ein maximaler Unterwürfel mit dieser Eigenschaft

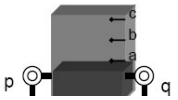


Partielle Fkt auf totale erweitern? $\Rightarrow$ Primimplikanten (egal ob Redundanz)

Primimplikantentheorem
Sei g eine disjunktive Form für $f \in S_n^D$ unter Redundanz r mit minimalen Kosten. Dann ist jedes Produkt g_i von g ein Primimplikant

Man braucht nur noch Darstellungen zu betrachten, die ausschließlich aus Primimplikanten bestehen

Beispiel:



a=1: mindestens 1/3 voll
b=1: mindestens 2/3 voll
c=1: voll

- schwächere Pumpe p läuft, wenn Behälter min 2/3 voll
- weniger als 2/3, mehr als 1/3, stärkere Pumpe q läuft
- weniger als 1/3: beide Pumpen laufen

a	b	c	p	q	r
0	0	0	1	1	0
0	0	1	*	*	1
0	1	0	*	*	1
0	1	1	*	*	1
1	0	0	0	1	0
1	0	1	*	*	1
1	1	0	1	0	0
1	1	1	0	0	0

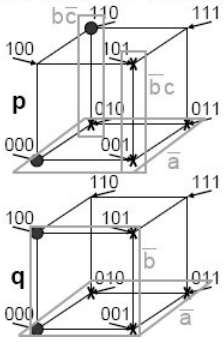
Redundant!

Nicht alle Eingangskombinationen kommen vor, da für die Sensorsignale einbaubedingt stets $a \geq b \geq c$ gelten muß.

Redundanz: $r = \overline{a}\overline{b}c \vee \overline{a}b\overline{c} \vee \overline{a}bc \vee a\overline{b}c$

a	b	c	p	q	r
0	0	0	1	1	0
0	0	1	*	*	1
0	1	0	*	*	1
0	1	1	*	*	1
1	0	0	0	1	0
1	0	1	*	*	1
1	1	0	1	0	0
1	1	1	0	0	0

Ziel: Finde die Primimplikanten für *p,q*



Primimplikanten zu p:
$\overline{a}$, $b\overline{c}$, $\overline{b}c$

Primimplikanten zu q:
$\overline{a}$, $\overline{b}$

$p = \overline{a} \vee b\overline{c}$, da
$\overline{b}c$ total redundant
$\overline{a}$ und $b\overline{c}$ enthalten Punkte des On-Sets

$q = \overline{b}$, da
auf $\overline{b}$ kann man nicht verzichten
$\overline{b}$ man braucht $\overline{a}$ nicht mehr: $\overline{b}$ dominiert $\overline{a}$

Karnaugh-Diagramme

Funktionstafel als 2 dimensionale Tabelle

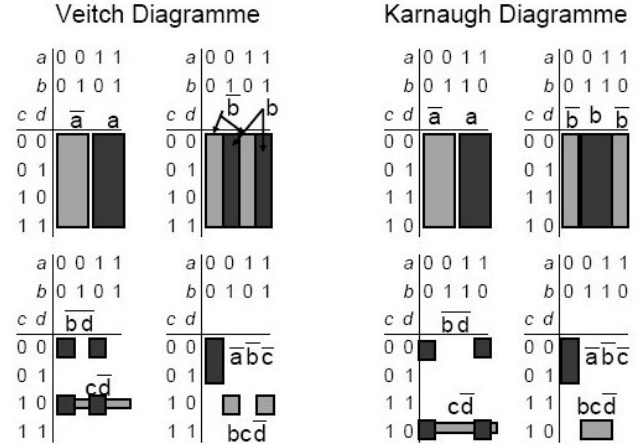
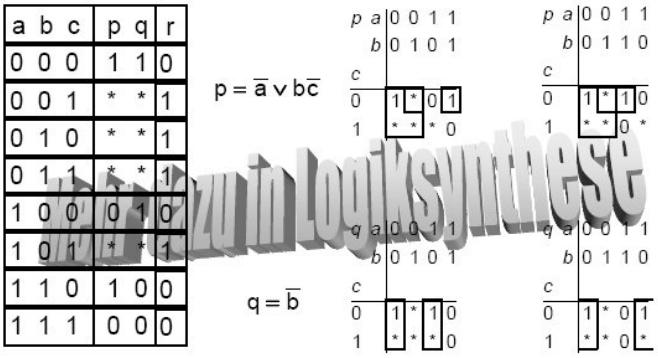


Diagramme zur Pumpensteuerung

a	b	c	p	q	r
0	0	0	1	1	0
0	0	1	*	*	1
0	1	0	*	*	1
0	1	1	*	*	1
1	0	0	0	1	0
1	0	1	*	*	1
1	1	0	1	0	0
1	1	1	0	0	0



Problem: Bis *4, 5* Variablen sieht man noch was. Was tut man aber für *n>5*? Benutze Synthesesoftware!

2.2.4 Mehrstufige Schaltkreise

Im Falle zweistufiger Schaltkreise kann man das Syntehseproblem also mit sehr hohem Aufwand lösen.
Bei zweistufigen Schaltkreisen kommt man allerdings schnell an Grenzen

PGPC (Parity Generator Parity Checker)

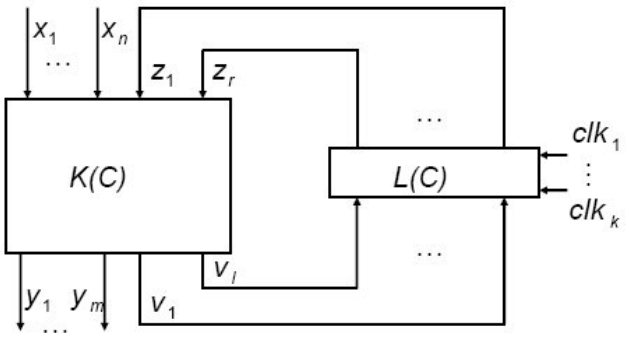
party: $B^n \to B$ mit $parity(x) = (\sum_{i=1}^{n} x_i) mod 2$

Hier ist jeder MinTerm Primimplikant und wesentlich, da jeder Minterm auf einen Punkt =1 und sind somit paarweise verschieden.
$\Rightarrow$ Mehrstufiger Schaltkreis: (da sonst $> 2^{n-1}$ Gatter)
Parity eines Vektors ergibt sich aus Summe modulo 2 der Parity von zwei Teilvektoren
GENERIC Parameter in der Definition von ENTITIES (mehr in VHDL)

2.3 Synchrone Schaltkreise und endliche Automaten

Schaltkreis synchron genau dann wenn:

- alle Signale s mit $g.clk \in s$ für ein Latch g sind Primäreingänge von C
- Nach Entfernung aller Latches und Hinzunahme der Ausgangssignale der Latches zu den Primäreingängen ist C kombinatorisch



Huffman Normalform
L(C) besteht nur aus Latches, die von den Takten $clk_1, ..., clk_n$ kontrolliert werden
K(C) ist kombinatorischer Schaltkreis mit $\{z_1, ..., z_r\} \cap \{x_1, ..., x_n\} = \varnothing$

Schaltwerke

- *Zustandgesteuerte* Schaltwerke: L(C) enthält nur zustandsgesteuerte Latches
- *Flankengesteuerte* Schaltwerke: L(C) enthält nur flankengesteuerte Latches

kombinatorischer Schalktkreis liefert Funktion f, die auf ein neues Element der Eingabefolge und einen aktuellen Zustand der Latches einen neuen Zustand und ein Element der Ausgabefolge berechnet

Taktzustandsgesteuertes Schaltwerk
In aktiver Phase sind Latches transparent
$\Rightarrow$ Berechnung des Folgezustands im Schaltkreis darf sich in der aktiven Phase nicht mehr an den Ausgängen bemerkbar machen.

Zeitbedingungen

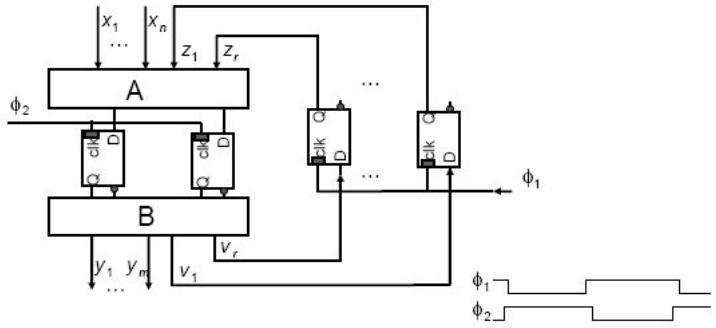
aktive Phase (0)
Berechnungen des Folgezustandes darf sich nicht mehr an Ausgängen bemerkbar machen

Minimale 0 Weite des Taktes
t_d Verzögerung des Latches
t_{min} Minimale Reaktionszeit des Schaltkreises
$\boxed{t_{CWL} \leq t_d(Latch) + t_{min}(C)}$

Taktperiode oder Zykluszeit
v muss spätestens 'Setup-Zeit vor Ende der nächsten aktiven Phase' den Folgezustand von z bereitstellen
$\boxed{T(clk) = t_{CWH} + t_{CWL} \geq t_d(Latch) + t_{max}(C)}$
Somit ist sichergestellt, dass der neu berechnete Zustand in der nächsten aktiven Phase noch übernommen wird

Taktfrequenz
$\boxed{f_{clk} = \frac{1}{T(clk)}}$
maximale Verzögerung des Schaltkreises
+ Verzögerung des Latches bestimmen die Taktperiode
UND: Maximalweitenbedingung der aktiven Phase

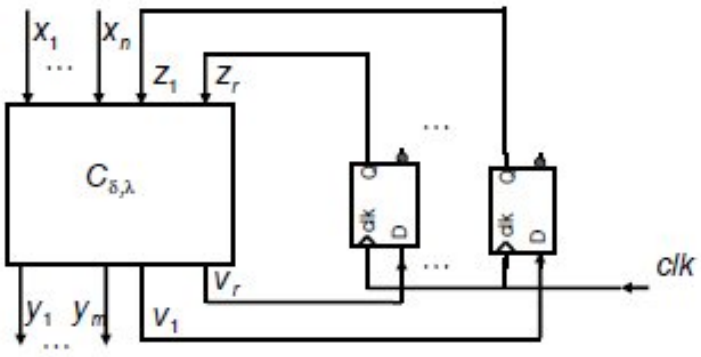
Schaltwerk mit zwei parallel arbeitenden Schaltkreisen A,B und Latches mit zwei unterschiedlich getakteten Gruppen:
$t_{CWL}(\phi_1) \geq t_{max}(A) + t_d(Latch)$
$t_{CWL}(\phi_2) \geq t_{max}(B) + t_d(Latch)$



$t_{CWH}(\phi_1) \geq t_{CWL}(\phi_2) + t_{skew}$
$t_{CWH}(\phi_2) \geq t_{CWL}(\phi_1) + t_{skew}$

Skew (t_{skew})
ist eine Zeitschranke, um sicherzustellen, dass die Takte überall dort, wo sie über Taktleitungen hinpropagiert werden unabhängig von den Leitungslaufzeiten sicher nicht überlappend sind

Taktflankengesteuerte Schaltwerke



$\Rightarrow$ Schieberegister
Es muss lediglich sicher sein, dass v t_{setup} vor der nächsten Flanke gültig ist.
$T(clk) \geq t_d(Latch) + t_{max}(C_{\delta,\lambda})$

MEALY-Automat

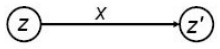
Tupel $M = (Z, X, Y, \delta, \lambda)$ heißt Mealy(Moore) Automat
Z: Zustandsmenge
X: Eingabemenge
Y: Ausgabemenge
δ: Übergangsfunktion: $\delta : Z \times X \rightarrow Z$
λ: Ausgabefunktion: $\lambda : Z \times X \rightarrow Y$ (Moore: $\lambda : Z \rightarrow Y$)

$\Rightarrow$ Steuerung zeitlicher Abläufe in synchronem Raster bei endl Gedächtnis
$\delta \in S^D_{n+r,r}$, $\lambda \in S^D_{n+r,m}$
$\Rightarrow$ Konstruktion von Schaltkreis $C_{\delta,\lambda}$ mit $f_{\delta,\lambda} \in S^D_{n+r,m+r}$

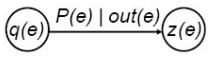
$$f_{\delta\lambda}(z,x) = (\lambda(z,x), \delta(z,x))$$

Zustandsmenge	$\Leftrightarrow$	Zustände in den Latches
Eingabe	$\Leftrightarrow$	Eingabevektoren
Ausgabe	$\Leftrightarrow$	Ausgabevektoren

Übergangsdiagramme



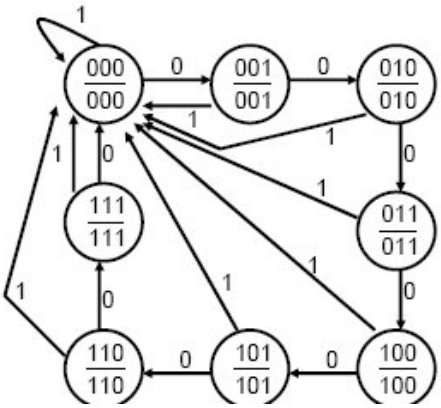
$\Leftrightarrow \delta(z,x) = z'$



, wobei q Quelle, z Ziel, P Eingabe und out Ausgabe
Es muss für $e \neq e' \Rightarrow P(e) \cdot P(e') = 0$ gelten sonst ist der Automat nichtdeterministisch

Beispiel: Zähler - Eingabe reset (0 oder 1)

z_0	z_1	z_2	reset	z'_0	z'_1	z'_2	y_0	y_1	y_2
0	0	0	0	0	0	1	0	0	1
0	0	1	0	0	1	0	0	1	0
0	1	0	0	0	1	1	0	1	1
0	1	1	0	1	0	0	1	0	0
1	0	0	0	1	0	1	1	0	1
1	0	1	0	1	1	0	1	1	0
1	1	0	0	1	1	1	1	1	1
1	1	1	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮



Beispiel: Fotoappartat

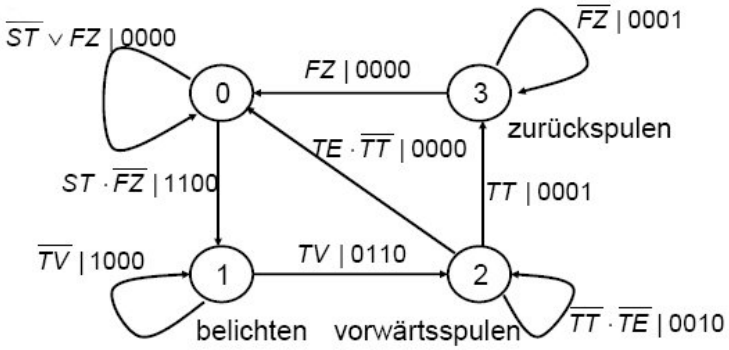
Timer (T)
- TV Ablauf Verschlusszeit
- TT Ablauf Transportzeitschranke

Signale
- FZ Film in Apparat (low active)
- ST Auslösertaste gedrückt
- TE Vorwärtstransport

Steuersignale
- V Verschluss öfnen
- MV Motor vorwärts
- MR Motor rückwärts

4-Tupel(V,T,MV,MR)

z_0z_1 Codierung des Zustands



$$f_V = \underbrace{\overline{z}_0\overline{z}_1 ST \cdot \overline{FZ}}_{(a)} \vee \underbrace{\overline{z}_0 z_1 \overline{TV}}_{(b)}$$
$$f_{MV} = \underbrace{\overline{z}_0 z_1 TV}_{(c)} \vee \underbrace{z_0\overline{z}_1\overline{TT} \cdot \overline{TE}}_{(d)}$$
$$f_{MR} = \underbrace{z_0\overline{z}_1 TT}_{(e)} \vee \underbrace{z_0 z_1 \overline{FZ}}_{(f)}$$
$$f_T = \underbrace{\overline{z}_0 z_1 TV}_{(c)} \vee \underbrace{\overline{z}_0\overline{z}_1 ST \cdot \overline{FZ}}_{(a)}$$
$$f_{z_0} = \underbrace{\overline{z}_0 z_1 TV}_{(c)} \vee \underbrace{z_0\overline{z}_1\overline{TT} \cdot \overline{TE}}_{(d)} \vee \underbrace{z_0\overline{z}_1 TT}_{(e)} \vee \underbrace{z_0 z_1 \overline{FZ}}_{(f)}$$
$$f_{z_1} = \underbrace{z_0\overline{z}_1 TT}_{(e)} \vee \underbrace{\overline{z}_0\overline{z}_1 ST \cdot \overline{FZ}}_{(a)} \vee \underbrace{\overline{z}_0 z_1 \overline{TV}}_{(b)} \vee \underbrace{z_0 z_1 \overline{FZ}}_{(f)}$$

Konstruktion durch Ausdrücke aus Mealy-Automaten

f_{y_i} Funktion der Ausgabeleitung
f_{z_i} Funktion für Zustandsbit z_i
$E_{y_i} = \{e \mid$ e hat Beschriftung P(e)$\mid r$ und $r_i = 1\}$
$E_{z_i} = \{e \mid z(e) = (..., q_{i-1}, 1, q_{i+1}, ...)\}$
$f_{y_i} = \bigvee_{e \in E_{y_i}} z^{q(e)} \cdot P(e)$
$f_{z_i} = \bigvee_{e \in E_{z_i}} z^{q(x)} \cdot P(e)$

2.4 Asynchrone Schaltkreise

Schaltkreise über kombinatorischen Bausteinen, die weder rückkopplungsfrei sein, noch auf jedem Zyklus einen Latch haben müssen

Delayunabhängige Analyse

($\rightarrow$ welche Belegungszustände erreicht werden können)

- **Bausteinverzögerungsmodell** (Muller model)
  Man darf jedem Bausteinausgang (Signaltreiber bzw. Signal) eine bel. Verzögerung größer 0 zuordnen
- **Leitungsverzögerungsmodell** (Huffman model)
  Man darf jeder Leitung (jedem Zweig eines Signals) eine bel. Verzögerung größer 0 zuordnen

Leitungsverzögerungsmodell ist allgemeiner
Wir beschränken uns auf das Bausteinverzögerungsmodell

Schaltvorgang
Belegung p' geht aus p durch Schaltvorgang hervor,
wenn $p'_i = p_i$ für $i \neq s$ und $p'_s = C[s](p) = \overline{p_s}$
Wir schreiben p kann nach p' schalten $p \succ p'$

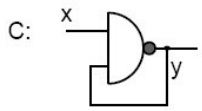
Stabile Belegung

Belegung ist stabil $\Leftrightarrow \forall p' B^m : \neg(p \succ p')$

Instabile Belegung

Folge $p = p^{(0)} \succ p^{(1)} \succ \cdots \succ p^{(n)} \succ \ldots$ (können auch endlos sein)
endlos $\Leftrightarrow$ prüfe alle Folgen der Länge $\leq 2^{\#s}$, denn spätestens nach $2^{\#s}$ Schritten muss sich eine Belegung wiederholen
Wiederholt sich erstmals eine Belegung, kann man diese Teilfolge immer wieder anhängen: "der Schaltkreis schwingt"

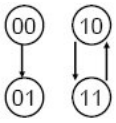
Beispiel 1



$C[y] = \overline{xy}$
$\chi_C = (y \equiv \overline{xy})$
(x,y) = (0,0) instabil, da $0 \neq 1$
(x,y) = (0,1) stabil, da $1 = \overline{0}$
(x,y) = (1,0) instabil, da $0 \neq 1$
(x,y) = (1,1) instabil, da $1 \neq 0$

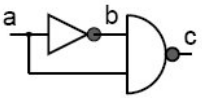
Belegungsdiagramm $\Rightarrow$ Schaltvorgänge analysieren

- Punkte(Knoten): alle Belegungen
- Pfeile(Kanten): von p nach q: $\Leftrightarrow p \succ q$



- Punkte, die keinen ausgehenden Pfeil haben: stabile Belegungen
- Zyklen: endlose Schaltfolgen

Beispiel 2



$$F[c] = C[c](b = F[b], a) = \overline{F[b] \cdot a} = \overline{\bar{a}a} = \overline{0} = 1$$

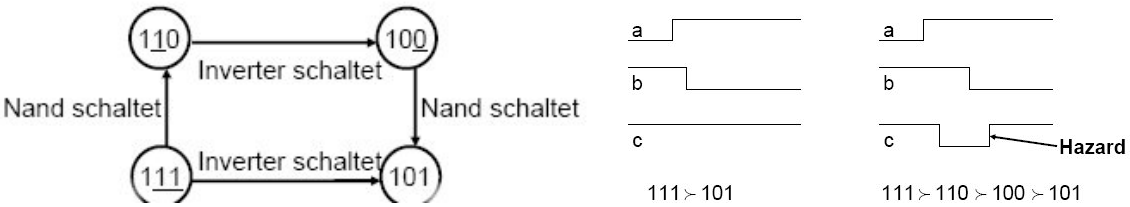
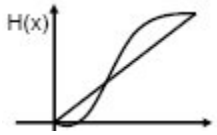


Diagramm zeigt uns alle möglichen Einschwingvorgänge
schalten von 011 nach 111

Metastabilität

In bistabilen Schaltungen kann es bei Einschwingvorgängen zu metastbilen Zuständen kommen, d.h. längeranhaltende Belegungen, die weder als 0 noch als 1 interpretiert werden können (nicht vermeidbar)



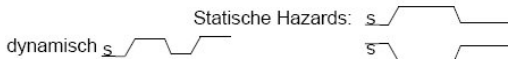
Übertragungssystem H mit x = H(x) (0 oder 1) und H'(x) = 0
beginnend mit Steigung H'(x) $<<$ 1 in (0,0) und $<<$ 1 in (1,1)
Schnittpunkt in (y,y) ist unvermeidbarer metastabiler Zustand

Metastabilität ist v.a. dann nicht vermeidbar, wenn man asynchrone externe Signale hat.

Maßnahmen

- Bauteile, die längeranhaltenden metastabilen Zustand unwahrscheinlich machen
- Zeitpunkt metastabiler Zustand weit weg von Zeitpunkt der Systemverhalten beeinflusst

Hazards
C kombinatorischer Schaltkreis, s Signal, p,q Eingabemuster (aufeinanderfolgend)
s hat **(statischen) Hazard** unter p,q genau dann wenn F[s](p) = F[s](q), die Schaltung aber beim Einschwingen instabil an s ist.
s hat **dynamischen Hazard** unter p,q genau dann, wenn F[s](p) $\neq$ F[s](q), aber s mehr als einer Änderung unterliegt



Smallest Cube Containing
$p, q \in B^n$ $scc(p,q) = \bigcap_{a\ Produkt\ \ a(p)=1=a(q)} a$
Beispiele:

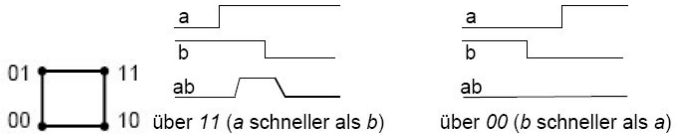
- scc(0000,1011) = $\overline{x_2}$
- scc(0101,0111) = $\overline{x_1}x_2x_4$
- scc($x_1\overline{x_2} \vee x_1\overline{x_3}$) = x_1

$scc(f) = \bigcap_{a\ Produkt\ \ ON(f)\subseteq ON(a)} a$

unvermeidbare Hazards
$F[s]_{scc(p,q)} \notin \{0,1\}$
$\Rightarrow$ globale Schaltfunktion $F[s]_{scc(p,q)} \in \{0,1\}$ ist konstant (0 oder 1)
Andernfalls gibt es q' im scc(p,q) mit $F[s](q') \neq F[s](q)$

Funktionale Hazards
Hazard unabhängig davon, wie F[s] berechnet wird

Beispiel: AND, scc(01,10)



Intervallabschätzung
Zeitintervalle abschätzen. $d_{min}(x,y), d_{max}(x,y)$ (Reaktionszeit Baustein)
$t_{min}(s) = 0$, s Primäreingang
$t_{min}(s) = min\{t_{min}(sig(g.x)) + d_{min}(x,y) \mid g.y = treiber(s)\}$
$t_{max}(s) = 0$, s Primäreingang
$t_{max}(s) = max\{t_{max}(sig(g.x)) + d_{max}(x,y) \mid g.y = treiber(s)\}$

Primäreingänge ändern sich: s ist höchstens in $[t + t_{min}(s), t + t_{max}(s)]$ instabil

3 Aufbau und Strukur eines Prozessors

3.1 Maschinen, Instruktionen und Programme

3.1.1 Abstrakte Maschinen

IS	Befehlssatz (instruction set)
P	Programmemenge P = IS^* (Folge von Befehlen)
O	Menge der Objekte mit denen gerechnet wird
mem $\in F(\mathbb{N}_0, O)$	gibt Speicherzustand an
mem(i)	welches Objekt im Zustand mem an Stelle i

Abstrakte Maschine M

M = (O,P,S,K,K_a,K_e,Δ)

O	Menge der elementaren Objekte
P	Menge der Programme
S = F($\mathbb{N}_O$,O)	Speicher
K = $P \times S$	Konfigurationsmenge
$K_a \subseteq K$	Startkonfiguration
$K_e \subseteq K$	Endkonfiguration
$\Delta : K \to K$	Übergangsfunktion mit D(Δ) $\subseteq K \backslash K_e$

Konfiguration

$k \in K$ "Schnappschuss" der Maschine bei der Arbeit

$\left\{ \begin{array}{l} \text{Programmzustand p} \\ \text{Speicherzustand s} \end{array} \right\}$ k = (p,s)

Berechnung

Folge $k_0, ..., k_n$ heißt eine Berechnung der Maschine M, wenn (nur dann)
$k_0 \in K_a$ und $\forall i < n$: $k_{i+1} = \Delta(k_i)$

$k' = \Delta(k_i)$	$k \to k'$ (direkt berechenbar)
$k' = \Delta^n(k_i)$	$k \to^n k'$ (direkt aus n Schritten berechenbar)
$k \to^* k'$	$\exists n > 0 : k \to^n k'$ (in endlich vielen Schritten)

Länge der Berechnung: n, falls $k_0, ..., k_n$ Berechnung
$k_n \notin D(\Delta)$: Berechnung ist terminiert (kann nicht mehr fortgeführt werden)
$k_n \in K_e$: Berechnung ist regulär

3.1.2 Instruktionssätze - CISC oder RISC?

Schnittstelle zwischen und Hardware und Programmiersprache

Complete Instruction Set Computer (CISC)
z.B.: 80x86 (2 Operanden; einer immer gleichzeitig Ziel), IBM370,...

- großer Befehlssatz
- viele Addressierungsarten
- kurze Programme
- unterschiedl. Bearbeitungszeit
- niedriger Durchsatz von Befehlen

Reduced Instruction Set Computer (RISC)
z.B.: SPARC, HPPA, Alpha,...

- magerer Befehlssatz
- geringe Bearbeitungszeit
- hoher Durchsatz von Befehlen
- ABER: längere Programme

3.1.3 Die Maschine SL (Straigt Line- Befehle werden nacheinander abgearbeitet)

Befehlssatz implementiert als Operation auf Operanden

Arithmetische Operationen

$$\left\{\begin{array}{l}\text{ADD i,j,k}\\ \text{MUL i,j,k}\\ \text{SUB i,j,k}\\ \text{DIV i,j,k}\end{array}\right\} A_{arith} \text{ für i,j,k aus } \mathbb{N}_0$$

im Speicher transportieren

$A_{trans} = \{\text{L i,j} \mid \text{i,j aus } \mathbb{N}_0\}$

IS := $A_{arith} \cup A_{trans}$
P := IS^*
O := $\mathbb{Z}$

Beispiel: Polynomauswertung auf SL

$$F(n+2) = \sum_{i=0}^{n} a_i x^i = ((...(a_n x + a_{n-1})x + a_{n-2})x + ... + a_1)x + a_0$$

```
Initialisierung { MUL n+2, 0 , n+1;      //x * a_n
                  ADD n+2,n+2,n;         //aufaddieren
                  MUL n+2,n+2,0;         //x ranmultiplizieren
                  ADD n+2,n+2,n-1;       //aufaddieren von a_{n-2}
                  MUL n+2,n+2,0;         //x ranmultiplizieren
                  .
n-1 mal           .
                  .
                  ADD n+2,n+2,2;         //aufaddieren von a_1
                  MUL n+2,n+2,0;         //x ranmultiplizieren
                  ADD n+2,n+2,1;         //aufaddieren von a_0
```

Bsp: $a_2x^2 + a_1x + a_0 = (a_2x + a_1)x + a_0$

mem(0)	x
mem(1)	a_0
mem(2)	a_1
mem(3)	a_2
mem(4)	a_2*x

```
MUL 4,0,3;
ADD 4,4,2;
MUL 4,4,0;
ADD 4,4,1;
```

Problem

Es ex. kein Programm, dass einen beliebigen Speicherzustand mem(0)=1 setzt
Bew: Falls alle S(i) = 0 sind gibt es keine Instruktion die Speicher ändert

⇒ Erweiterung LOAD IMMEDIATLY

$A_{trans} = A_{trans} \cup \{\text{LDI i,r} \mid i \in \mathbb{N}_0, r \in \mathbb{Z}\}$mit mem(i) = r

Der Operand r wird unmmittelbar (immedtialy) in den Speicher übernommen

3.1.4 Verzweigungen und Sprünge

Wir nennen die erweiterte SL: **WüRC0 (Würzburger RISC 0)**

Speicherplatz **pc (program counter)**: zeigt auf den aktuell zu bearbeitenden Befehl und werde selbst durch Befehle verändert
Speicherzustand s: (pc,mem), $pc \in \mathbb{N}_0, mem \in F(\mathbb{N}_0, \mathbb{Z})$
s(0) = pc;
s(i+1) = mem(i);
$\Delta(p, (pc, mem)) = (p, (pc', mem'))$

Verzweigung (Branches) und Sprünge(Jumps)

JMP (Sprung) - - pc = i
BEQZ, BNEZ(Verzweigungen) - - if s(i) = 0 pc = k else pc = pc + 1

SXX (Set condition XX) SLT, SLE, SEQ, SNE, SGE, SGT (Vergleiche)
- - falls erfüllt s(i) = 1 sonst s(i) = 0

J: Jump, B: Branch
EQ: =, NE: $\neq$; LT: $<$, LE: $\leq$, GT: $>$, GE: $\geq$

WüRC0LC(Low Cost): lediglich: SEQ, SLT, JMP
Man kann WüRC0 auf WüRC0LC simulieren, aber:
Erhöhung der Programmlänge

Marken (labels)

- Markiere Instruktion mit Label
- Sprungziel ebenfalls Labelname
- Jede Marke hat im Programm genau ein definiertes Auftreten (Wert i, falls p(i))
- alle anderen Auftreten werden durch den Wert einer Marke ersetzt

```
BNEZ 4,loop
.
loop: ...
```

Verarbeitungszeitanalyse bei ggT-Programm

Verlust von WüRC0LC ggü. WüRC0 ca. $\frac{4}{7}$
Dies führt zu Gewinn falls wir WüRC0LC etwa 36% schneller bauen können

3.1.5 Adressierungsarten

Aufgabe für bel Speicherzustand S mit S(0) = n einen Zustand F mit F(10+i) = i zu erzeugen scheitert mit aktuellen Befehlssatz

Bew: höchstens p Speicherzellen gesetzt, da jede Instruktion max eine verändert
Stets die Zahl #D(F) der in F von p veränderten Speicherzellen $\leq |p|$
Wähle Startbelegung S mit S(0) $> |p|$

Lösen mittels:

Indirekte Addressierung
mit LD (Load) und ST (Store)

LD (Load): s(i) = s(s(j)+k)
ST (Store) : s(s(i)+k) = s(j)

- Laden und speichern in eine Zelle, deren Adresse durch den Inhalt des Quell- bzw. Zieloperanden indirekt gegeben ist
- Gegensatz zur bisher benutzen direkten Adressierung
- Den immediate Operanden sc2 nennen wir Displacement

Adresse verweist auf ein Reg. in dem Adresse des eigentlichen Ziels liegt
(für Unterprogrammaufruf oder wenn zur Entwicklungszeit noch nicht feststeht wieviele Speicherzellen angesprochen werden)

Immediate Erweiterungen - Operanden Bestandteil des Befehls

ADDI i,j,k: mem[i] = mem[j] + k
analog: SUBI, MULI, DIVI
INC, DEC als inkrement, dekrement

Direkte Adressierung

Adresse verweist auf Register

Implizit

JMP nutzt pc implizit als Zielregister

3.1.6 Unterprogramme

Indirekter Sprung

JAL i,j: mem[i] = pc + 1, pc′ = j RET i: pc′ = s[i]

Unterprogrammregeln

(U1) mem(0) trägt Adresse von Speicherzelle k, so dass in mem(k),...,mem(k+l-1) die Argumente stehen

(U2) Keine Speicherzelle j > k+l wird vom aufrufenden Programm schon benutzt

(U3) a) Register mem(0),...,mem(r-1) dürfen vom Unterprogramm verändert werden
(aufrufende Programm trägt Vorsorge - caller saves Strategie)

b) Register mem(0),...,mem(r-1) müssen unverändernt zurückgeliefert werden
(Unterprogrmm trägt Vorsorge - callee saves Strategie)

(U4) mem(0) hat nach Rückkehraus Unterprog. den gleichen Wert wie beim Aufruf

(U5) mem(1) hält stets die Rücksprungadresse
man könnte auch andere nehmen, aber: Callee muss wissen wo sie steht

- mem(0) zeigt an, ab welchem Speicherbereich nur noch Argumente und freie Plätze kommen

- mem(0) ist Stackpointer (oberes Ende des Stapels)

Register

- **special purpose** (pc, mem(0), mem(1))
- **general purpose** allgemeine Aufgaben

3.1.7 Ausblick

Primitive Instruktionssätze

Oft ausgezeichnetes Register acc(Akkumulator)
⇒ sämtliche Berechnungen über dieses ausgezeichnete Register

- Akkumulatormaschine
- oft nur einen (Einadresscode) oder zwei (Zweiadressecode) weitere Operanden außer dem Akkumulator
- unrealistische Befehlssätze

3.2 Die Objekte der Maschine

Zahlendartellung

d: $B^k \to \mathbb{N}/\mathbb{Z}/\mathbb{R}$
$\Rightarrow$ alles auf Bitstrings zurückführen

Wortbreite n

einer Maschine legt die maximale Länge von Objekten als Bitstrings fest, die in einem Regsiter oder Speicherplatz gehalten werden können
$a = a_0, ..., a_{n-1}$ Maschinenwort (heute üblich: n=64,n=32)

(unsigned) ganze Zahl

$u_n : B^n \mapsto \mathbb{N}_0$ mit $\boxed{u_n(a_0, ..., a_{n-1}) := 2^{n-1} \sum_{i=0}^{n-1} a_i \cdot 2^{-i}}$

a_0 hat höchstes Gewicht: signifikantes Bit
$u_n(B^n) = [0 : 2^n - 1]$

Signed Numbers (Betrag und Vorzeichen)

$bv_n : B^n \mapsto \mathbb{Z}$ mit $\boxed{bv_n(a_0, ..., a_{n-1}) := (-1)^{a_0} \cdot 2^{n-1} \sum_{i=1}^{n-1} a_i \cdot 2^{-i}}$

$bv_n(B^n) = [-(2^{n-1} - 1) : 2^{n-1} - 1]$

Signed Numbers (Zweierkomplement)

$b_{2,n} : B^n \mapsto \mathbb{Z}$ mit $\boxed{b_{2,n}(a_0, ..., a_{n-1}) := -2^{n-1}a_0 + 2^{n-1} \sum_{i=1}^{n-1} a_i \cdot 2^{-i}}$

$b_{2,n}(B^n) = [-2^{n-1} : 2^{n-1} - 1]$
Injektiv, leicht mit unsigned adder zu addieren

Signed Numbers (Einerkomplement)

$b_{1,n} : B^n \mapsto \mathbb{Z}$ mit $\boxed{b_{1,n}(a_0, ..., a_{n-1}) := -(2^{n-1} - 1)a_0 + 2^{n-1} \sum_{i=1}^{n-1} a_i \cdot 2^{-i}}$

$b_{1,n}(B^n) = [-(2^{n-1} - 1) : 2^{n-1} - 1]$

$$b_2(\bar{a}) + b_2(a) + 1 = 0 \Leftrightarrow b_{2,n}(1, ..., 1) + 1 = 0$$

$$\begin{aligned}
b_{2,n}(1, ..., 1) + 1 &= -2^{n-1} + 2^{n-1} \sum_{i=1}^{n-1} 2^{-i} + 1 = \\
&= -2^{n-1} + 2^{n-1} \sum_{i=0}^{n-1} (\frac{1}{2})^{-i} - 2^{n-1} + 1 = \\
&= 2 * (-2^{n-1}) + 2^{n-1} \left(\frac{(\frac{1}{2})^n - 1}{\frac{1}{2} - 1} \right) + 1 = \\
&= -2^n + 2^{n-1}(-\frac{1}{2}^{n-1} + 2) + 1 = \\
&= -2^n - 1 + 2^n + 1 = \\
&= 0
\end{aligned}$$

$$\begin{aligned}
b_2(a) &= -a_0 \cdot 2^{n-1} + 2^{n-1} \cdot \sum_{i=1}^{n-1} a_0 2^{-i} = \\
&= -a_0 \cdot 2^{n-1}(2 - 1) + 2^{n-1} \cdot \sum_{i=1}^{n-1} a_0 2^{-i} = \\
&= -a_0 \cdot 2^n + a_0 \cdot 2^{n-1} + 2^{n-1} \cdot \sum_{i=1}^{n-1} a_0 2^{-i} = \\
&= -a_0 \cdot 2^n + 2^{n-1} \cdot \sum_{i=0}^{n-1} a_0 2^{-i} = \\
&= -a_0 \cdot 2^n + u(n)
\end{aligned}$$

allgemeine Zahlendarstellung

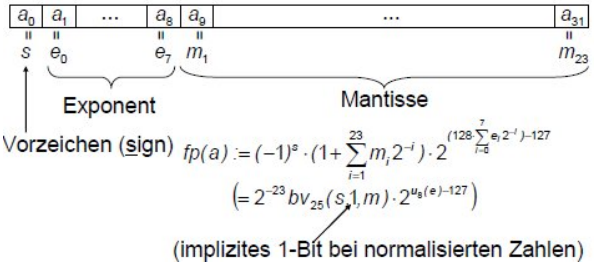
$d_n : B^n \mapsto \mathbb{Z}$ (Bitstrings als Zahlen)

1. $d_n(B^n)$ Intervall
2. $0 \in d_n(B^n)$
3. Symmetrie: $d_n(B^n) = [-a; a]$ (nur b_v und b_1)
4. d_n injektiv und total (nur u und b_2 - negative 0)

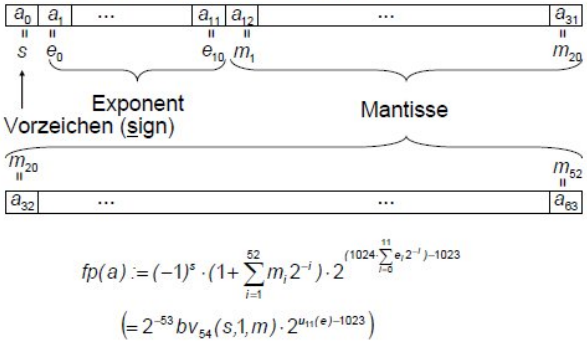
Gleitkommzahlen

Mantisse $\rightarrow a \cdot 10^e$ bzw $a \cdot 10^{-e} \leftarrow$ Exponent

Single precision floating point numbers



double precision floating point numbers



$$fp(a) := (-1)^s \cdot (1 + \sum_{i=1}^{52} m_i 2^{-i}) \cdot 2^{(1024 \cdot \sum_{i=6}^{11} e_i 2^{-i}) - 1023}$$

$$\left(= 2^{-53} bv_{54}(s,1,m) \cdot 2^{u_{11}(e)-1023}\right)$$

Zeichen

injektive Abbildung $c : A \mapsto B^k$
ordnet Zeichen im Zeichnsatz A einen Code zu
sinnvoll: lexikographische Anordnung

$u_k(c(0)) < ... < a_k(c(9)) < u_k(c(A)) < ... < u_k(c(Z)) < u_k(c(a)) < ...$

Krieg der Endians

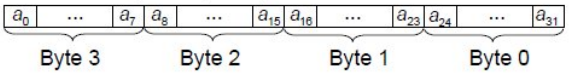
⇒ in welcher Reihenfolge legt man Bytes in einem Teilwort ab
größere Teilworte stellen Zahlen dar:
Bits von signifikant(0) nach weniger signifikant(n-1)

Bytes in kleinerer Adresse ordnet man:

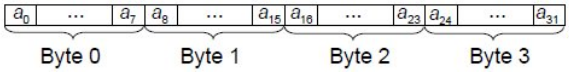
- die weniger signifikante Position des Teilworts zu (Little Endian)
- die mehr signifikante Position des Teilworts zu (Big Endian)

Beispiel: 32-bit Maschine

Little Endian (I80x86, VAX)

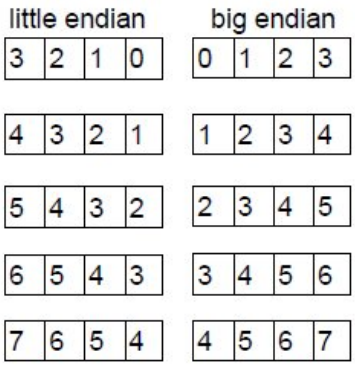


Big Endian (IBM370, Motorola 680x0)



Alignment

- **Aligned Access**
  Teilwortadressen sind vielfaches der Länge in Bytes
  (Halbwort, Wort, Doppelwort)
- **Misaligned Access**
  Teilworte dürfen auf bel. Positionen beginnen



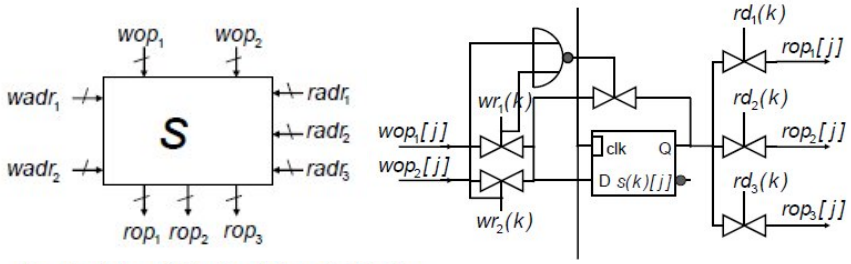
Ausblick

- Adressen: unsigned Zahlen
- Befehle: Maschinenworte (oder Vielfache), Objekte + Arithmetik

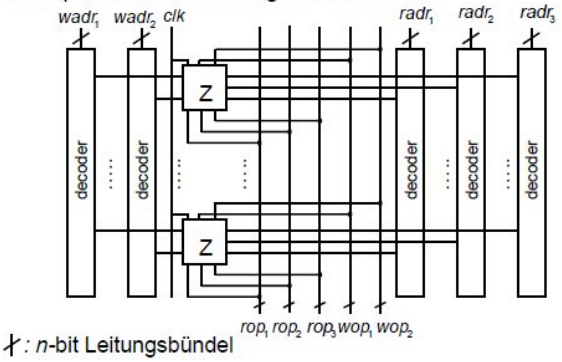
3.3 Speicherstrukturen

Multiportspeicher

"3-Lese-2-Schreib-Speicher" (D-Latch)
aus WüRC0: 2 Speicherworte geändert, 3 gelesen



Der Speicher hätte also folgende Struktur:



: n-bit Leitungsbündel

- wadr: Adressen der Schreiboperanden wop
- radr: Adressen der Leseoperanden rop

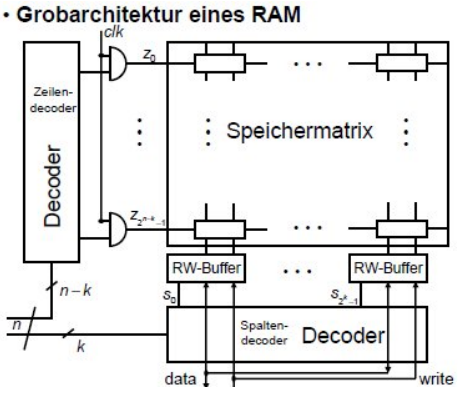
Schreibleitungen: $wop_i[j]$, i=1,2 $\quad s[u_n(wadr_i)] = wop_i$
Leseleitungen: $rop_i[j]$ $\quad rop_i = s[u_n(radr_i)]$

Schreibzugriffe an Port 1 und 2 müssen stets auf verschiedenen Zellen stattfinden, sonst ggf. Kurzschluss
$wr_i(k) = 1 \Leftrightarrow u_n(wadr_i) = k$
$rd_i(k) = 1 \Leftrightarrow u_n(radr_i) = k$
$\Rightarrow$ sehr teuer $\Rightarrow$ nicht als wirklich große Speicher

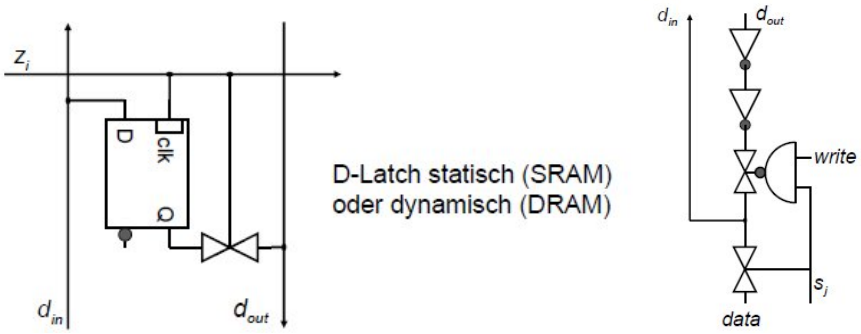
Speicherkonzept der Maschine

- **GP**-Register (General Purpose)
  Zellen des kleinen Multiportspeicher
- **SP**-Register (Special Purpose)
  Zellen mit Sonderaufgaben (Programmzähler)

Hauptteil des Speichers: Single-Port, der in einem Zyklus ausgelesen oder beschrieben werden kann
RAM (Random Access Memory)



Read / Write-Buffer



Lesezugriff (write = 0) über Doppelinverter verstärkt, zurückgeschickt und ausgegeben
Schreibzugriff (write = 1) Doppelinverter verstärken ⇒ ausgewählte Spalte überschreiben

Zeitbedingungen

- t_{seta} Adressen müssen t_{seta} vor dem clock stabil sein, damit Spalten- und Zeilenauswahl vor Zugriff stabil
- t_{setup} write und data t_{setup} stabil vor clock
- t_{access} data wird erst t_{access} nach aktiver clock stabil

Speicher

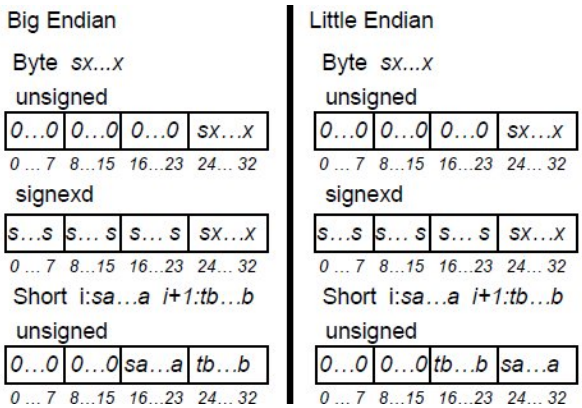
Annahme: Wir haben Speicherbausteine mit 1-bit Wortbreite
⇒ Konstruktion des Hauptspeichers einer Maschine
Byteorientierten Speicherblock aus 8 parallelgeschalteten bitorientierten Speicherblöcken
(enable-Leitung: Speicher zum Arbeiten ermächtigt)

Speicherzelle ohne Kenntnis der Technologie

FlipFlop und Verhalten (stabile Zustände mittels χ_C

Teilwortzugriffe (rechtsbündig)

- unsigned: signifikante Stellen mit 0 auffüllen
- sign extended: sig. Stellen mit Vorzeichen auffüllen



Zeitbedingung und Arbeitsweise

Modell generisch: Zeitfaktor, Adressbreite

Takt (clk)
aktiv(0) $\Rightarrow$ Zugriff auf Speichermatrix
$\Rightarrow$ Kontrollsignale (Adressen, write, enable) müssen während dieser Phase + delay_faktor davor stabil sein

Aktive Phase $>$ addresswidth*delayfaktor

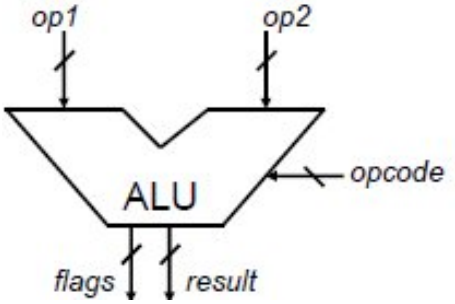
enable = 0
soll bidirektionale Port data hochohmig gehalten werden
(Speicher arbeitet nicht)

write = 1
data für ganze 1-Phase hochohmig $\Rightarrow$ Eingang zum Schreiben

3.4 Arithmetische und logische Operationen - - die ALU

ALU (arithmetical logical unit) - die Einheit, die diese Operationen zur Verfügung stellt

- **op 1,2** Operanden der auszuführenden Operation
- **result** Resultatwort zur Operation
- **opcode** Auswählen der auszuführenden Operation
- **flags** Statusanzeigen (Fehler, Vergleichsergebnisse)



Funktionsscheiben

Zu jeder Operation ein Schaltkreis $\widehat{=}$ Funktionsscheibe
$\Rightarrow$ nur dann rechnen, wenn Ergebnis auch benutzt wird
(unnötiger Energieverbrauch, Kühlprobleme)
$\Rightarrow$ Schalter mit dem die Operanden angelegt werden mittels Decoder (one-hot-code)

Addition und Subtraktion

addu: $B^{2n} \mapsto B^{n+1}$
addu(a,b) = (ov,s) $\Leftrightarrow 2^n ov + u(s) = u(a) + u(b)$
ov: Überlaufanzeige: Summe größer als Maschinenwort

1. $b_2(a) = b_1(a) - a_0$
2. $b_1(\overline{a}) = -b_1(a)$ und $-b_2(x) = b_2(\overline{x}) + 1$
3. $b_1(a_0, a_0 \oplus a_1, ..., a_0 \oplus a_{n-1}) = b_v(a)$
   $b_v(a_0, a_0 \oplus a_1, ..., a_0 \oplus a_{n-1}) = b_1(a)$

Subtraktion : aus (2): nur bits des Subtrahenden invertieren

Zurückführen auf unsigned Adder

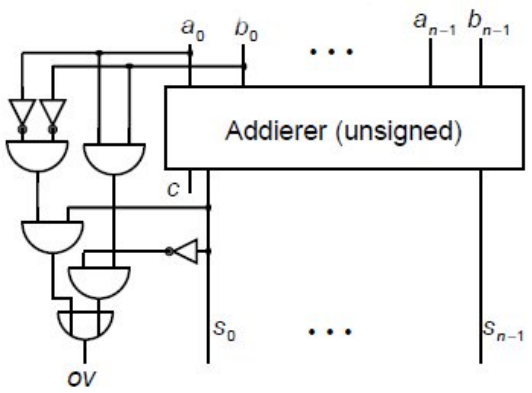
- $b_2(a) = -a_0 2^n + u(a)$
- $2^n c + u(s) = u(x) + u(y)$

$\Rightarrow 2^n c + b_2(s) + s_0 2^n = b_2(x) + x_0 2^n + b_2(y) + y_0 2^n$
$\Rightarrow b_2(s) = b_2(x) + b_2(y) + (x_0 + y_0 - s_0 - c)2^n$
Den hinteren Teil der Summe betrachten wir genauer

- $x_0 + y_0 = 0$, beide Operanden nicht negativ
  $u(x) + u(y) \leq 2^{n-1} - 1 + 2^{n-1} - 1 = 2^n - 2 \Rightarrow c = 0$
  $(x_0 + y_0 - s_0 - c)2^n = -s_0 2^n \neq 0 \Rightarrow$ Überlauf
  $\Rightarrow$ Ergebnis korrekt wenn kein Überlauf ($s_0 = 0$ und Ergebnis damit positiv)

- $x_0 + y_0 = 1$, Operanden haben verschiedenes Vorzeichen
  Für einlaufenden Übertrag c_1 im Addierer gilt:
  $2c + s_0 = x_0 + y_0 + c_1$ mit $c_1 \in \{0, 1\}$
  $1 = x_0 + y_0 \leq 2c + s_0 \leq x_0 + y_0 + 1 = 2$ Also ist $c + s_0 = 1 \Rightarrow$ Ergebnis immer korrekt

- $x_0 + y_0 = 2$, beide Operanden negativ
  $u(x) + u(y) \geq 2^{n-1} + 2^{n-1} = 2^n \Rightarrow c = 1$
  $b_2(x) + b_2(y) \leq 2^n + 2^{n-1} - 1 - 2^{n+1} = -2^{n-1} - 1 \Rightarrow$ Überlauf
  $\Rightarrow$ Ergebnis korrekt wenn kein Überlauf ($s_0 = 1$ und Ergebnis damit negativ)



Überlaufbedingungen

$x, y, s \in B^n, c \in B$ mit $2^n c + u_n(s) = u_n(x) + u_n(y)$
Dann gilt:
$b_{2,n}(s) = b_{2,n}(x) + b_{2,n}(y)$ und $b_{2,n}(s) \in [-2^{n-1} : 2^{n-1} - 1]$
$\Leftrightarrow$
$\overline{x_0 y_0} s_0 \vee x_0 y_0 \overline{s_0} = 0$

Multiplikation

n Partialprodukte aufaddieren: $p_i = b_i 2^{n-1-i} u(a)$

$u(a) \cdot u(b) = u(a) \cdot 2^{n-1} \sum_{i=0}^{n-1} b_i 2^i$

Multiplikation mit Zweierpotenzen $\Leftrightarrow$ Linksshift
(sofern keine führenden Stellen hinaus geschoben werden)

prod: $B^n \times B^n \mapsto B^{2n}$
$u_{2n}(prod(a,b)) = u_n(a) \cdot u_n(b)$

Paritalprodukte in 2n-bit breiten Registern (2n bit ALU) iterativ aufsummiert
$\Rightarrow$ Shift & Add Multiplizierer (dauert n Taktperioden)

Besser: Carry Save Addierer $\Rightarrow$ CSA Multiplizierer

Wallace Tree: in logartihmischer Tiefe multiplizieren
(Partialprodukte parallel erzeugen)

3.5 Realisierung eines einfachen Prozessors

Arithmetik
OP $\in$ {ADD, SUB, MUL,DIV, SXX}
d $\in \{b_v, b_1, b_2\}$ (Zahlendarstellung)
OP i,j,k, OPU i,j,k OPI i,j,k OPUI i,j,k
Suffix U: unsigned, Suffix I: Immediate

OP i,j,k bzw. OPI i,j,k
reg(i) = $d_n^{-1}(d_n(reg(j))op\ d_n(reg(k)))$ bzw.
reg(i) = $d_n^{-1}(d_n(reg(j))op\ k)$

Verzweigungen
BEQ i,j; BNEQ i,j, BOV j BNOV j (OV: Verzweigungen nach dem Overflow flag)

Transporte
LD i,j,k, ST i,j,k
LD: reg s(i) = mem($u_n(reg\ s(j)) + k$)
ST: mem($u_n(reg\ s(i)) + k$) = reg s(j)
$\Rightarrow$ Transporte zwischen Registern und Speichern

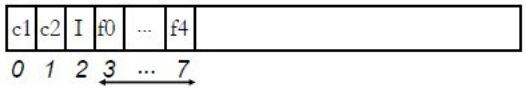
Sprünge
JMP k, RET i, JAL i,k

Nullregister
REG(0) nicht überschreibar mit Wert 0

Kodierung

- Arithmetik auf Registern (A)
- Verzweigungen (B)
- Transporte (T)
- Sprünge (J)

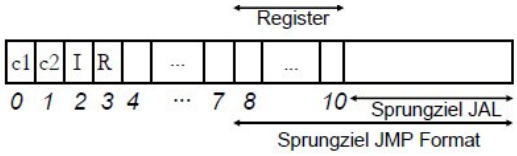
Function Codes - Kodiere Instruktionen in Maschinenworten



$(c_1 c_2)$: (00) Arithmetik
(10) Transport
(01) Branch
(11) Jump

I: Immediate bit
5 bit: $f_0, ..., f_4$ für Arithmetikbefehle, Sprünge
1 Byte: bei 3 Adressen noch je 1 Byte
für Registeradressen (256 Reg)

Sprünge



24 bit Sprungziele: am besten relativ angeben

1-Zyklus WüRC

in jedem Zyklus ein Befehl $\Rightarrow$ 2 RAM Speicher(Programmcode und Speicher)
(**Harvard Architektur**)
Kontrolle erzeugt:

- Steuerleitungen für Multiplexer (Kontrolle des Datenflusses)
- Operationcodes, Adressen und Kontrollsignale ...

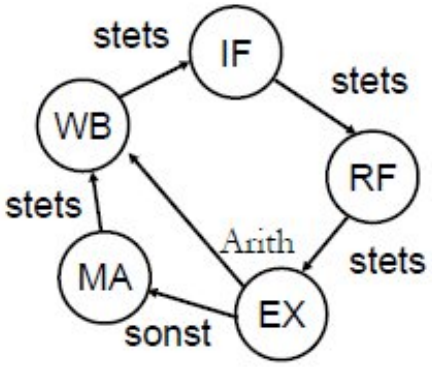
Kritik:
pro Taktzyklus einen Befehl, aber Zykluszeit
$r_{clk} \geq 2 \cdot t_{access\_cache} + 2 \cdot t_{acces\_reg} + t_{ALU} + t_0$

Mehr-Zyklus-Implementierung

Vorteile:

- Speicherzugriffe für Daten und Befehle in unterschiedlichen Zyklen
  $\Rightarrow$ nur ein Speicher (**Von Neumann Architektur**)
- Befehle, die nicht alle Resourcen nutzen können in wenigen Zyklen laufen
- Befehle in verschiedenen Zyklen parallel laufen lassen (Pipelining)

5-Zyklus WüRC



- Befehl holen (instruction fetch - IF)
- Befehl dekodieren und Register auslesen (instruction decode and register fetch - RF)
- Berechnung ausführen (execute - EX)
- Speicherzugriff (memory access - MA)
- Ergebnis wegschreiben (write back - WB)

Taktperiode:
$r_{clk} \geq max\{t_{access\_cache}, t_{acces\_reg}, t_{ALU}\} + t_0$

Pipeline WüRC

Es muss

- Befehls- und Datenchaces benutzt werden
- Jeder Befehl in 5 Zyklen bearbeitbar

$\Rightarrow$ Befehle parallel im Fließbandprinzip

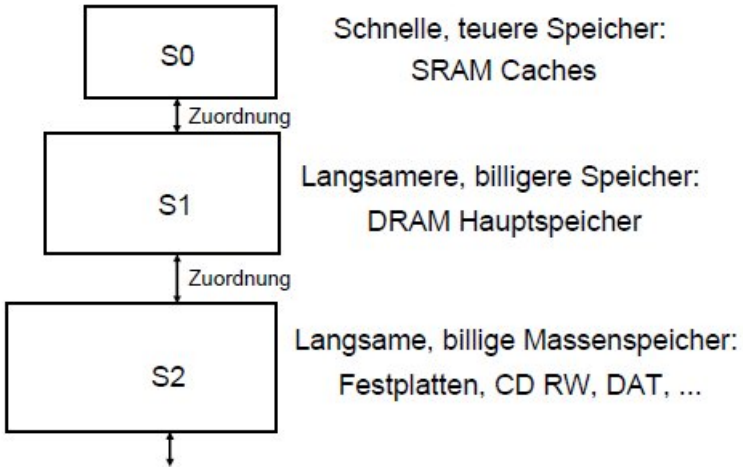
Fehler: Adresse des Zieloperanden darf nicht direkt angelegt werden, sonst würde Zieloperand des Befehls 4 mit dem Ergebnis des Befehls 1 überschrieben werden

Ausblick

- Pipelining nicht so leicht umzusetzen wie hier skizziert
- Rechenwerke mit unterschiedl tiefen PipelineStufen

3.6 Die Speicherhierarchie

Realisiere schnelle Befehlsausführung (kleine Speicher) und unterstütze viele Anwendungen (großer Speicher)



Mit zunehmender Tiefe in der Hierarchie, wächst die Zugriffszeit und sinken die Kosten pro Bit Speicherplatz

Zeitliche und räumliche Lokalität (Prinzip)

Annahme: Daten und Befehle, die in einem kurzen Zeitintervall der Maschine bearbeitet werden, liegen im Speicher meistens nahe benachbart beieinander

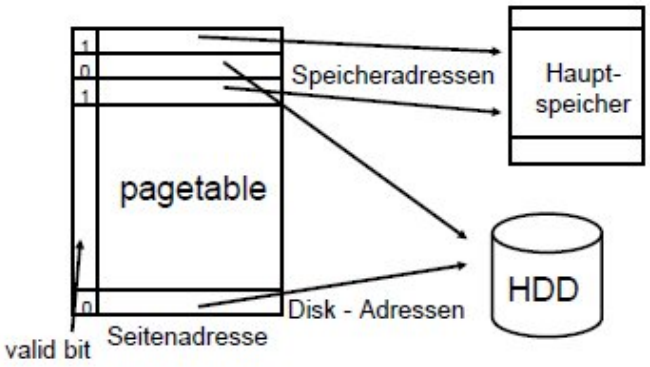
Cache (direkte Zuordnung)

weitverbreitete Technik bei Zusammenwirken zwischen Caches und Hauptspeicher: direkte Zuordnung
neben Datenwort, auch die führenden n-b bit seiner Adresse im cache.
Solange zwei Zellen den Abstand $< 2^b$ haben können sie gleichzeitig im Cache aufbewahrt werden (*cache_miss*)
Lokalität ist also gewahrt

Virtuelle Adressierung

- Man kann selten ganzen Adressraum als RAM Speicher realisieren
- Unterstützung von multi tasking / multiuser Systemen
  Maschinenworte = virtuelle Adressen $\Leftrightarrow$ physikalische Adressen von [0;M-1]
  Umsetzung: Betriebssystem
- Zur Umsetzung: Pagetable
  führt Buch ob Seite im Hauptspeicher liegt oder auf langsamere Massenspeicher ausgelagert wurde

pagetable



Teile der Pagetable in kleinen, schnellen Speicher: Translation Look Aside Buffer (TLB) ist Umsetzung zwischen Cache und Hauptspeicher

Speicherschutz - Exceptions

- Programm beenden
- nach Bereinigung der Situation wieder fortzsetzen
- Unterprogramme zur Ausnahmenbehandlung
- exception program counter (Spezialregister, da Befehl nicht im Programm)

Interrupts

Unterbrechung von langsamen Peripheriegeräten zur Berechnung