

Rechenanlagen

SS 2019

- Vorlesung:** Mo 12:00-13:30, Fr 10:15-11:45
- Dozent:** Prof. Dr. Reiner Kolla
- Assistent:** M.Sc. Johannes Mühr



Organisatorische Anmerkungen

- Material zu Vorlesung und Übungen erhältlich über uniweite eLearning-Plattform **Wuecampus**
- Zum Login ist ein Novell-Account des Rechenzentrums notwendig. Informationen dazu und zur Anmeldung erhältlich über
<https://www.rz.uni-wuerzburg.de>
- Übungsanmeldung nach ***Losverfahren*** bis einschließlich **1.5.2019** über **Wuestudy** möglich
- Adresse um zum Kurs Rechenanlagen zu gelangen
<https://wuecampus2.uni-wuerzburg.de>

*Zugangsschlüssel für den Kurs: **FlipFlop***

Organisatorische Anmerkungen

- Über eLearning-Plattform Wuecampus finden Sie z.B. folgende Materialien:
 - **Skript zur Vorlesung (Powerpoint, PDF)**
Achtung: Skript wird ggf. nachträglich aktualisiert !
 - **Aufgabenblätter**
 - **FAQs, HOWTOs, Weblinks u.a.**
- **Übungsbetrieb** (1. Übung am 16.04.2018 - Organisatorisches)
- **Aufgabenblätter** (immer Montags)
 - wöchentlich, Bearbeitungszeit: 1 Woche (i.d.R.)
 - Zahl der Bearbeiter pro Lösung: min: 2 / max: 3
 - mindestens 2 mal „vorrechnen“ in der Übung und
 - 50% der Aufgabenpunktesind hinreichend für das Bestehen des **Tutoriums Rechenanlagen** bzw. das Erreichen des **Bonus**.
Das **Modul Rechenanlagen** wird durch eine Klausur geprüft.

Organisatorische Anmerkungen

➤ Programmieraufgaben

Wir empfehlen den Public Domain Simulator **GHDL** zur Simulation.

Download und Installation siehe Download- und HOWTO-Bereich des Kurses Rechenanlagen bei der eLearning-Plattform Moodle.

Aufgaben sind aber prinzipiell auch ohne Rechner lösbar, d.h. nur anhand des VL-Skripts und der Beispielprogramme. Erwartet werden keine zu 100% syntaktisch korrekten Quellcodes.

Abgabe von Programmieraufgaben: *in schriftlicher Form*

Bücherliste

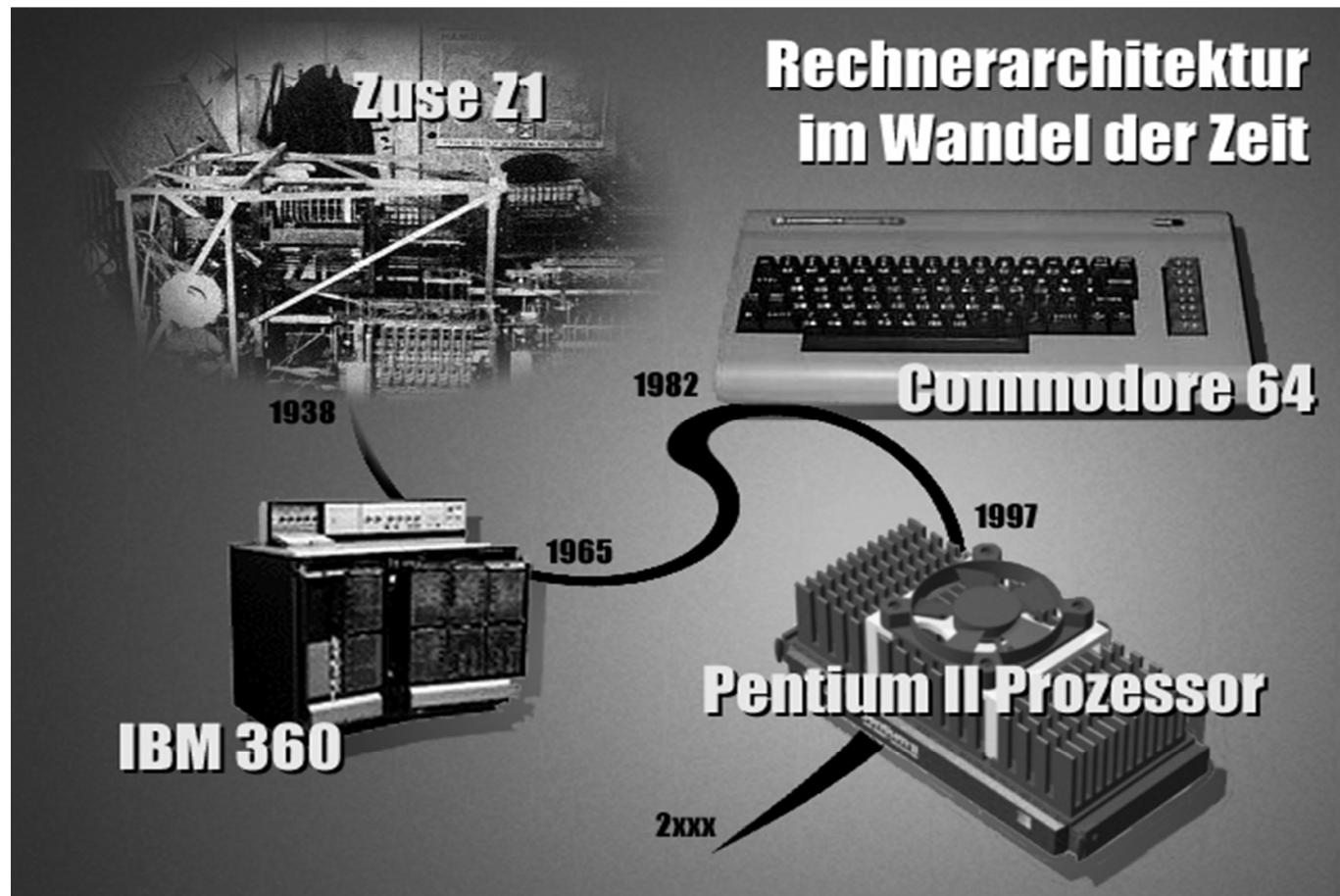
- I. **Structured Computer Organization**, 4th ed
A.S. Tanenbaum & J.R. Goodman, Prentice Hall
- II. **Computer Architecture, a Quantitative Approach**,
D. Patterson & J.L. Hennessy, Morgan Kaufmann Publishers Inc
- III. **Technische Informatik Band 2: Entwurf digitaler Schaltungen**, *Günter Kemnitz, Springer*
(hat VHDL als begleitende HDL)

Buch zu VHDL

Kompaktkurs VHDL,
P. Molitor und J. Ritter,
Oldenbourg Verlag 2013



1.0 Prolog



Eniac (1946)



- 30 Tonnen, 3m hoch, 24m breit
- 18 000 Elektronenröhren
- Multiplikationszeit: 3ms

1943-50: Was man sich damals so dachte!

- **Thomas Watson (IBM)**

„I think there is a world market for maybe five computers.”

- **Popular Mechanics:**

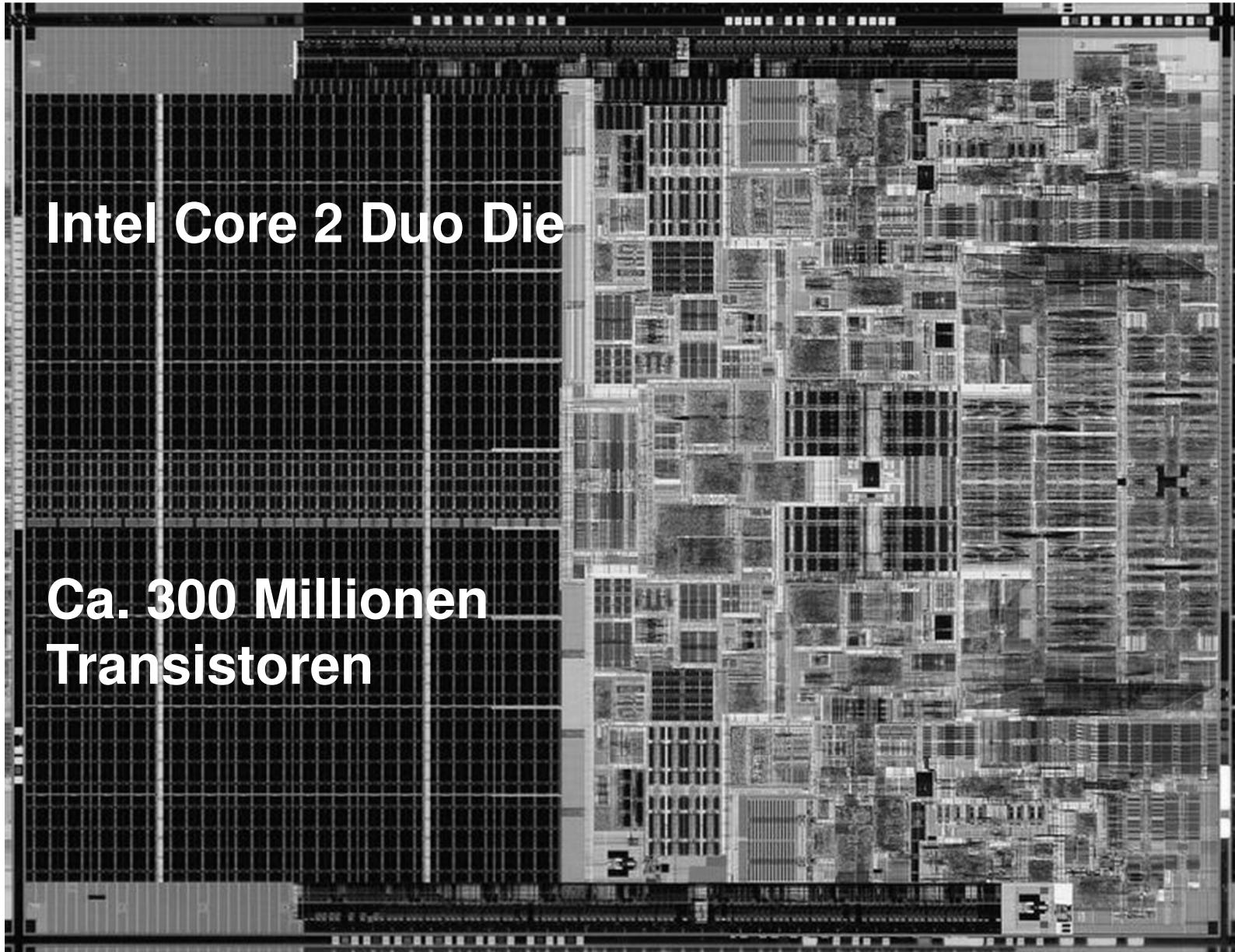
„Computers in the future may weigh no more than 1.5 tons.”

2000 -- und heute!

- ENIAC on a chip
- Laptop = alle Rechner der Welt von 1950

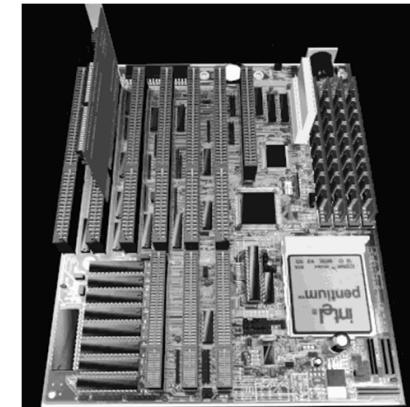
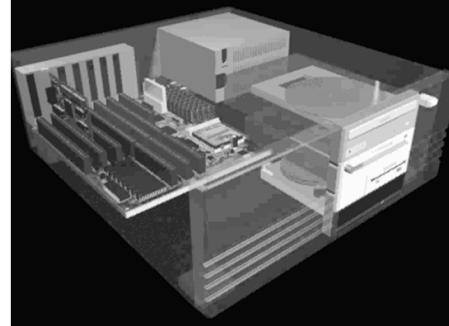
Intel Core 2 Duo Die

**Ca. 300 Millionen
Transistoren**



Lernziel in dieser Vorlesung

"Wie werden Rechner aufgebaut, warum werden sie so aufgebaut und wie funktionieren sie?"



Lernziel in dieser Vorlesung -- ff

Rechner werden heutzutage durch **höchstintegrierte Schaltungen** realisiert.

Frage



Prozessor =

Maskendaten für einen Halbleiterfertigungsprozess

+ Gehäusespezifikation

+ Bondplan

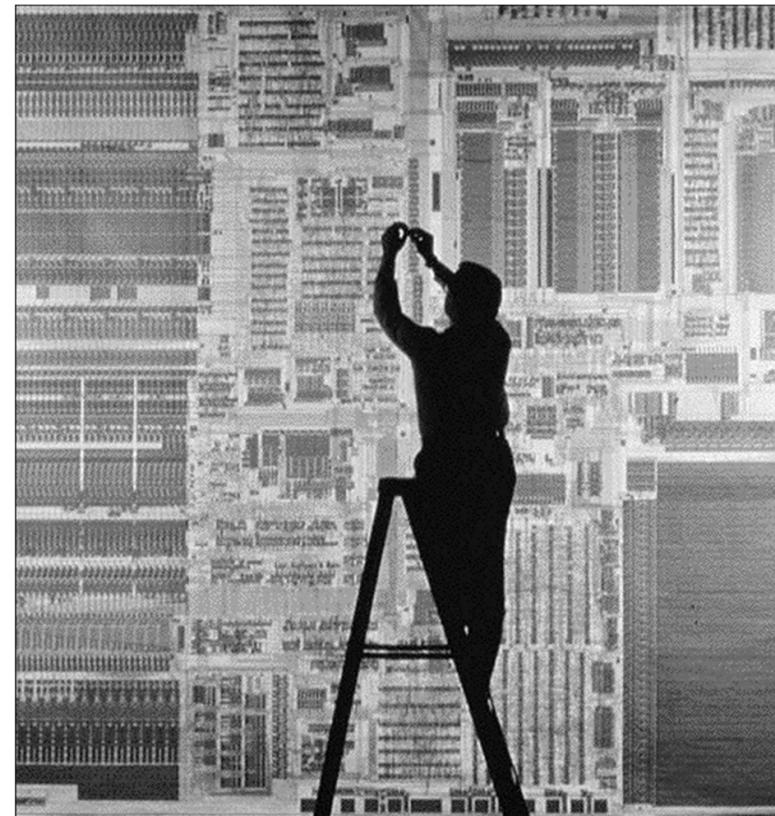


Der Entwurf von Rechnern erfordert die Benutzung vieler Softwarewerkzeuge. Aufbau und Arbeitsweise von Rechenanlagen sind demnach nur noch im Zusammenhang mit diesen Werkzeugen zu verstehen. Wir tragen dieser Tatsache in dieser Vorlesung Rechnung.

Problem: First time success

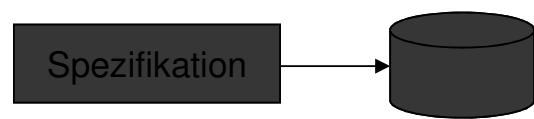
Stelle sicher, dass die erzeugten Maskendaten von Anfang an ein korrekt arbeitendes System liefern.

Hohe Anforderungen an die Überprüfung und Sicherstellung der Korrektheit vor der Anfertigung des ersten physischen Exemplars.



(so wird das nichts!)

Vorgehensweise beim Entwurf -- Designflow

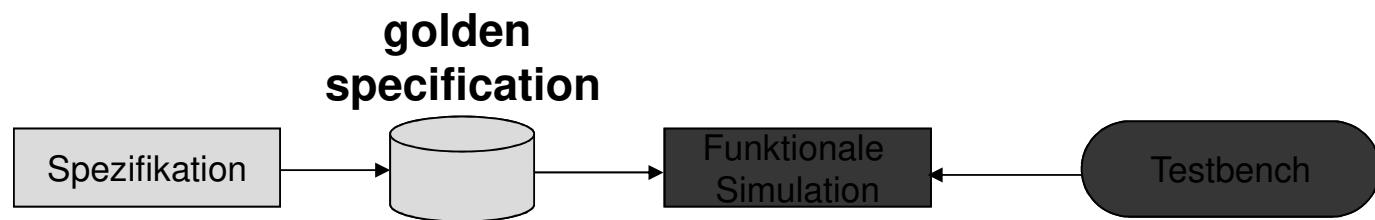


Spezifikation

- **Schaltplaneingabe (Schematic Entry)**
 - Graphikeditor zur Eingabe einer Entwurfsskizze der Schaltung
 - Herstellung der Verbindungen von Blöcken
 - Strukturbeschreibung
- **Beschreibung mittels einer Hardwarebeschreibungssprache (HDL)**
 - Hardware wird wie Software durch Programme spezifiziert
 - Struktur- oder Verhaltensbeschreibung

In dieser Vorlesung werden wir begleitend zum Stoff eine Hardwarebeschreibungssprache (Hardware Description Language = HDL) einführen. Sie ist eine echte Teilmenge des IEEE Standards **VHDL93** (VHSIC (= very high speed integrated circuit) Hardware Description Language). Wir nennen sie **WüHDL**.

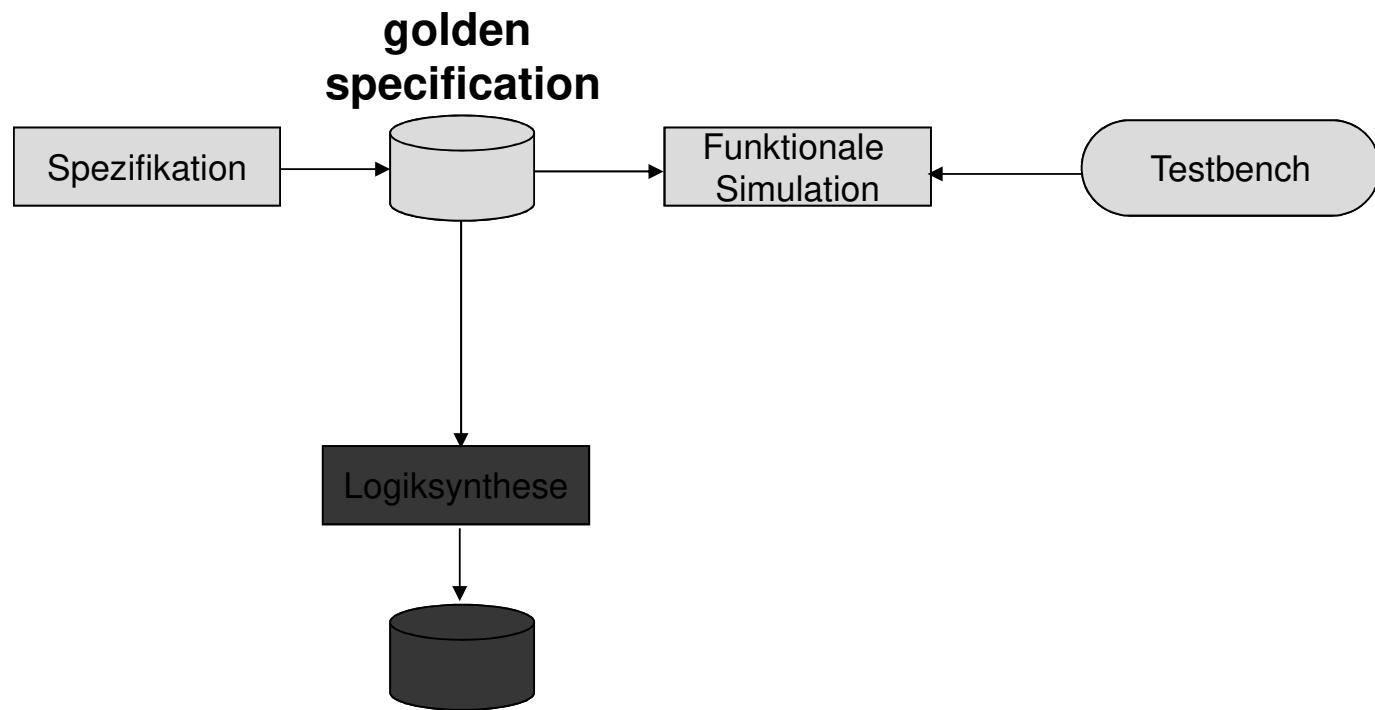
Designflow



Korrektheitsprüfung durch Simulation

- **Funktionale Simulation der Spezifikation**
= Überprüfung der logischen Korrektheit der Spezifikation
- **Vorgehensweise:**
 - Anlegen ausgewählter Eingabemuster, den **Stimuli**
 - Ist / Soll-Vergleich
 - Die Simulation wird üblicherweise durch ein Programm, die **Testbench**, gesteuert

Designflow



Logiksynthese und -minimierung

Versuch, die Logik der Schaltung

- zu generieren, bzw.
- zu minimieren.

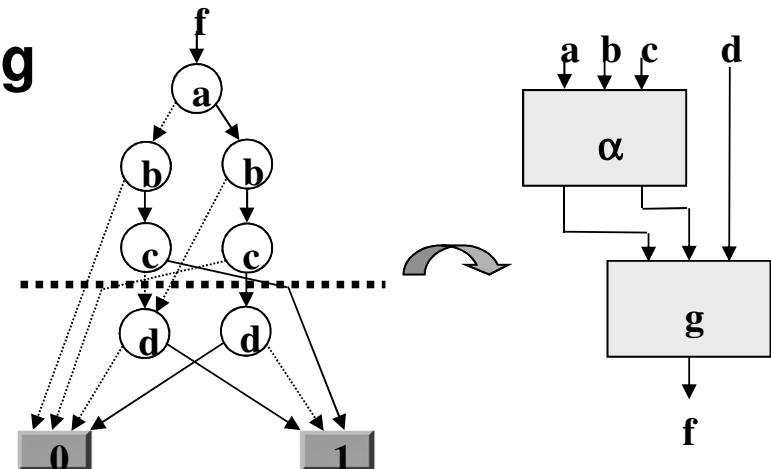
Unterscheidung zwischen

- **kombinatorischer Synthese**

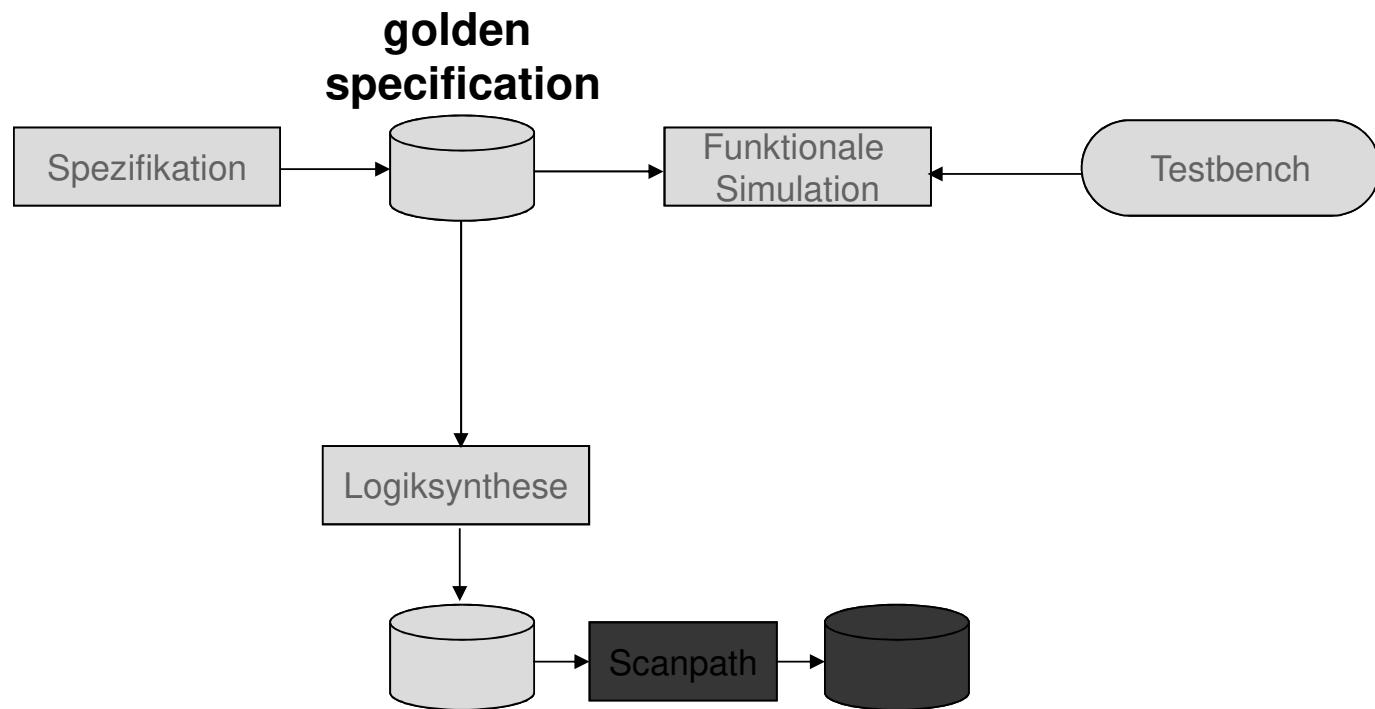
= Realisierung einer Booleschen Funktion

- **sequentieller Synthese**

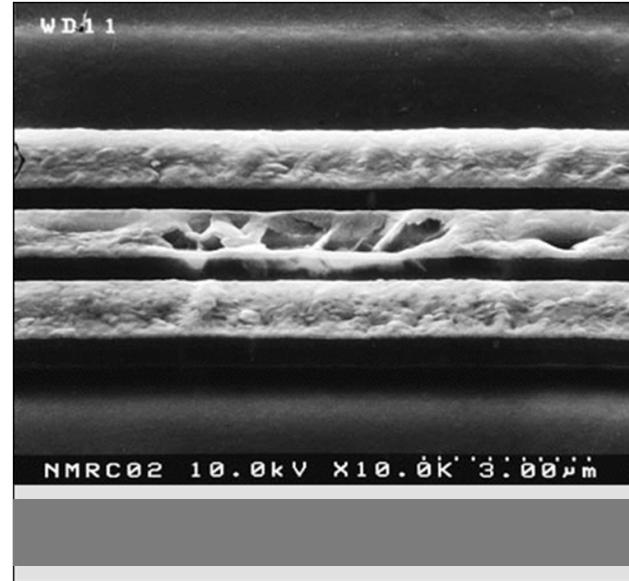
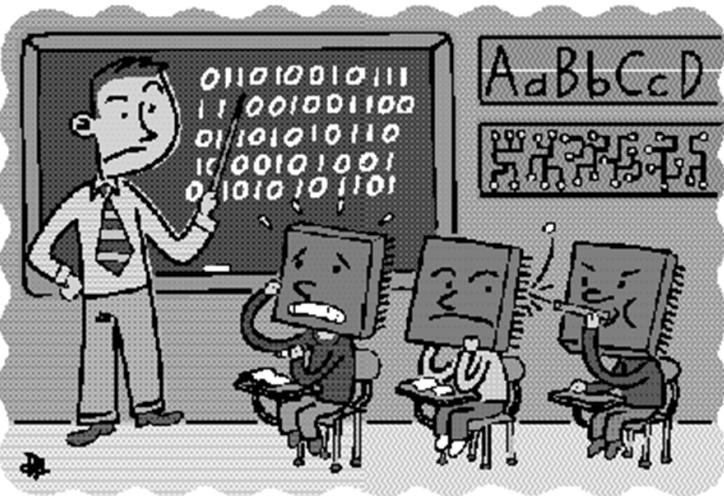
= Realisierung eines Endlichen Automaten



Designflow

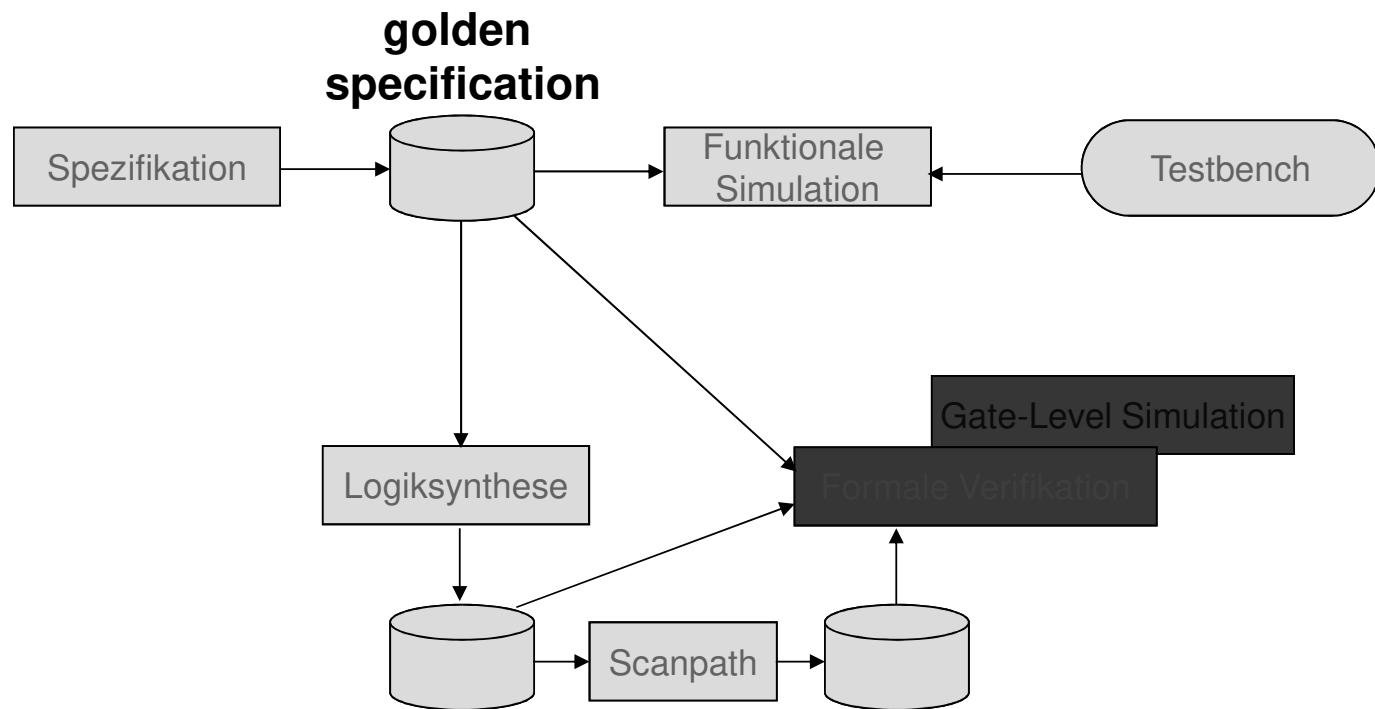


Testmustererzeugung

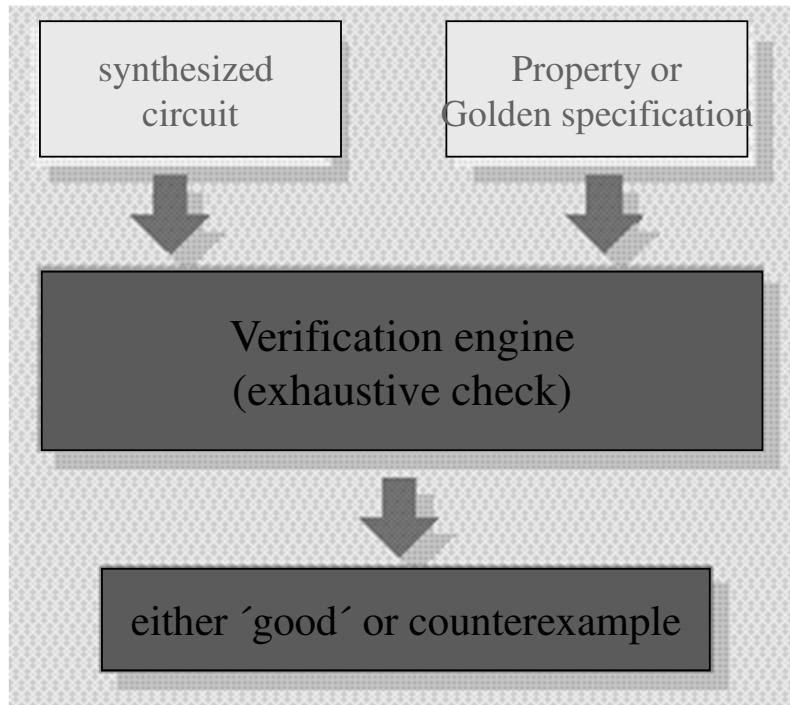


- Überprüfung, ob Spezifikation effizienten Test auf Fabrikationsfehler erlaubt
- Berechnung eines (fast) vollständigen Tests,
 - Menge von Eingabemuster, die alle Fabrikationsfehler (Fehler eines gegebenen Fehlermodells) entdeckt.

Designflow



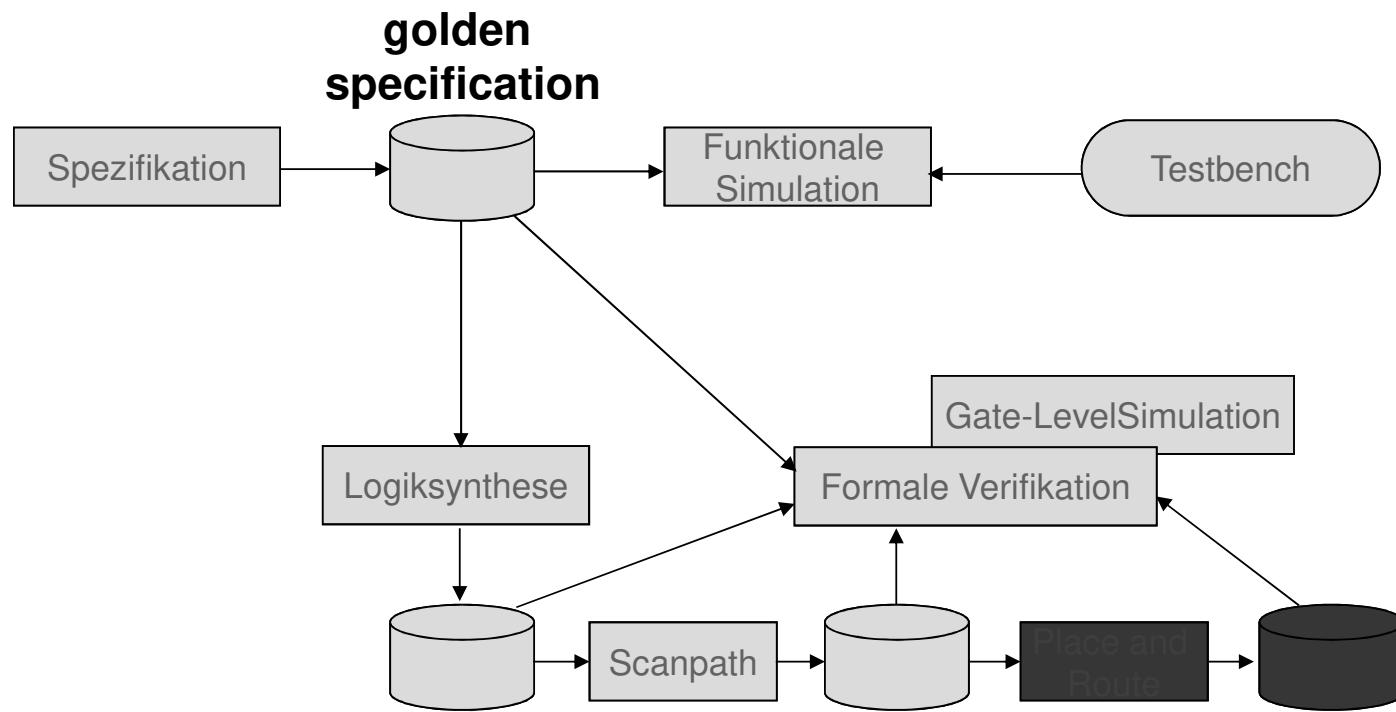
Formale Verifikation



oder:

Formaler Beweis, dass
Eigenschaft gilt oder der
goldenene Spezifikation
entspricht.

Designflow



Erstellung der Fertigungsdaten

- **Erstellung der Fertigungsunterlagen**
 - **Maskendaten:** geometrische Daten, das Layout betreffend
 - **Testprogramm für den Testautomaten**
 - **Bondplan:** Zuordnung der Pads zu den Gehäusepins

1. Die Grundlagen zum Hardwaredesign

Zur Vorlesung Rechenanlagen

SS 2019



1.1 Grundlegende Begriffe

Zur Vorlesung Rechenanlagen

SS 2019



Notationen

N Bezeichne die natürlichen Zahlen -- POSITIVE^{*})

N_0 Bezeichne die natürlichen Zahlen mit 0 -- NATURAL^{*})

Z Bezeichne die ganzen Zahlen -- INTEGER^{*})

Q Bezeichne die rationalen Zahlen

R Bezeichne die reellen Zahlen -- REAL^{*})

-- WüHDL

$B := \{0, 1\}$ Bezeichne die Menge der „booleschen“ Werte
0 (false,low) und 1 (true,high)

-- BOOLEAN (true,false) oder

-- BIT ('0','1')

^{*}) mit Einschränkungen!

Mengen in WüHDL

Neben diesen Mengen haben beliebige endliche Mengen in der Informatik eine besondere Bedeutung. WüHDL ist als VHDL Subset und damit als Sprössling der Programmiersprache ADA "streng getypt". Endliche Mengen lassen sich als Typen durch Aufzählen der Elemente definieren:

TYPE Alphabet IS (a_1, a_2, \dots, a_n);

-- a_i Bezeichner für das i-te Zeichen

TYPE Color IS (red,blue,green);

Vordefinierte Typen dieser Art sind z.B.

TYPE CHARACTER IS (...,'A','B',...,'a','b',...,'0','1',...);

-- alle ASCII Zeichen

TYPE BIT IS ('0','1');

TYPE BOOLEAN IS (FALSE,TRUE);

Durch die Enumerationsreihenfolge ist zugleich auch eine Ordnung auf der Menge definiert, von Typbezeichner'low nach Typbezeichner'high.

1.1.1 Relationen und Funktionen

Seien A und B Mengen. Dann ist die Menge $A \times B$ die Menge aller geordneten Paare, die man aus Elementen aus A und B bilden kann:

$$A \times B := \{ (a, b) \mid a \in A \text{ und } b \in B \}$$

Eine Teilmenge $R \subseteq A \times B$ heißt **Relation**. Wir schreiben auch aRb statt $(a, b) \in R$ (Infixschreibweise)

und nennen

$A := Q(R)$ die **Quelle** oder den **Quellbereich** der Relation,
 $B := Z(R)$ das **Ziel** oder den **Zielbereich** der Relation.

Für eine Teilmenge $X \subseteq A$ sei

$R(X) := \{b \mid (a, b) \in R \text{ und } a \in X\}$
das **Bild** von X unter R

und es sei

$R^{-1} := \{(b, a) \mid (a, b) \in R\}$ die **Umkehrrelation** zu R .

Funktionen

Identifizieren wir einelementige Mengen $\{x\}$ mit dem Element x , so ergibt dies für Elemente x aus A

$$R(x) := R(\{x\}) := \{b \mid (x, b) \in R\}$$

Eine Relation R heißt **(partielle) Funktion**

: $\Leftrightarrow \forall a \in Q(R): \#R(a) \leq 1$ -- #M := Anzahl der Elemente in M (Kardinalität)

Sei R eine partielle Funktion: Wir nennen

$D(R) := \{a \mid \#R(a) = 1\}$ den **Definitionsbereich** von R und

$W(R) := R(D(R)) (= R(Q(R)))$ den **Wertebereich** von R

R heißt **totale Funktion** oder **Abbildung**

: $\Leftrightarrow R$ ist Funktion und $D(R) = Q(R)$

Vereinbarungen

Für eine Funktion R und ein Element a ist offenbar $R(a)$ einelementig oder leer.

Ist $R(a)$ leer sagen wir auch **R(a) ist undefined**.

Ist R leer nennen wir R eine **überall undefinierte Funktion**, (meist notiert als Ω).

Funktionen notieren wir fortan als kleine oder griechische Buchstaben, Relationen durch Operationssymbole oder große Buchstaben.

Schreibe $f:A \rightarrow B$ für eine (partielle) Funktion $f \subseteq A \times B$.

Definition

Sei $f: A \rightarrow B$ eine (partielle) Funktion.

Dann ist die Einschränkung von f auf die Menge X , $f|_X$, definiert durch $D(f|_X) := D(f) \cap X$

und

$$f|_X(a) := \begin{cases} f(a) : a \in D(f) \cap X \\ \infty : \text{sonst} \end{cases}$$

Definition

Eine partielle Funktion $f: A \rightarrow B$ heißt

- **injektiv:** $\Leftrightarrow (f(a), f(b) \text{ def. und } f(a) = f(b)) \Rightarrow a=b$
- **surjektiv:** $\Leftrightarrow f(A) = B$
- **bijektiv:** $\Leftrightarrow f$ ist total, injektiv und surjektiv

Definition

Sei $f: A \rightarrow B$ eine partielle Funktion. Dann heißt $g: A \rightarrow B$ eine Erweiterung von f

$$f \subseteq g : \Leftrightarrow g|_{D(f)} = f$$

Weitere Notationen:

$F(A, B)$ Menge der Funktionen von A nach B

$Abb(A, B)$ Menge der totalen Funktionen von A nach B

Relationen auf einer Menge M

Definition

Eine Relation R auf einer Menge M , d.h. $R \subseteq M \times M$, heißt

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

Eine reflexive, transitive und symmetrische Relation heißt
Äquivalenzrelation.

Eine reflexive, transitive und antisymmetrische Relation heißt
partielle Ordnung.

Beispiele:

$A = \text{Menge aller Menschen}$ $B = \text{Menge aller Ortschaften}$

$$R \subseteq A \times B := \{(a, b) \mid a \text{ wohnt in } b\}$$

R ist keine Funktion (Neben bzw. Zweitwohnsitze)

$A = B = \text{Menge aller Ortschaften}$

$$R \subseteq A \times B := \{(a, b) \mid a \text{ zu Fuß erreichbar von } b\}$$

- reflexiv

R ist

- symmetrisch
- transitiv

 R ist Äquivalenzrelation

Beispiele:

A= Menge aller PKW Typen

B={0,1}

$$R = \{(a,0) \mid V_{max}(a) \leq 110 \text{ km/h}\}$$

$$\cup \{(a,1) \mid V_{max}(a) \geq 160 \text{ km/h}\}$$

R ist eine Funktion, aber nicht total.

Operationen auf Relationen

Seien R, S Relationen. Dann ist

$$R \circ S \text{ definiert} : \Leftrightarrow Z(R) = Q(S)$$

und wir setzen:

$$R \circ S := \{(a, c) \mid \exists b : (a, b) \in R \wedge (b, c) \in S\}$$

Sei $R \subseteq A \times A$ Wir setzen

$$R^0 := \{(a, a) \mid a \in A\} \quad \text{Identität auf } A$$

und $R^1 := R$

$$R^{n+1} := R^n \circ R$$

$$R^* := \bigcup_{n \in \mathbb{N}_0} R^n \text{ heißt reflexiver, transitiver Abschluss}$$

1.1.2 Zeichen und Worte

Zeichen: Elemente einer endlichen Menge A

Wir nennen A auch Alphabet

Worte: (endliche) Folgen von Zeichen

$$A^+ := \{w \mid w : [1 : n] \rightarrow A, n \in \mathbb{N}\}$$

heißt Menge der Worte über dem Alphabet A . Für i ist $w(i)$ das i -te Zeichen im Wort w .

Wir nennen n auch die Länge von w , und schreiben

$$|w| = n \quad \text{-- WüHDL: w'length}$$

Demnach ist ein Wort w also stets eine Abbildung

$$w : [1 : |w|] \rightarrow A$$

Worte, Vektoren und Arrays

In Programmiersprachen werden Worte im Sinne dieser Definition in vielerlei Hinsicht benutzt. Wir illustrieren dies an unserer Beispielsprache WüHDL:

➤ **Strings:** Betrachtet man Folgen von Zeichen mit beliebigen, nichtnegativen Längen, dann kann man diese zusammenfassen in einem Typ String. In WüHDL gibt es einen vordefinierten Typ dieser Art, allerdings nur für Folgen von CHARACTER:

```
TYPE STRING IS ARRAY(POSITIVE RANGE <> ) OF CHARACTER;
```

Hier ist der Bereich der Folge undefiniert. Konkrete Deklarationen von Strings müssen den Bereich aber spezifizieren:

```
STRING(1 TO 5);
```

Es gibt auch vordefinierte Strings über Bits, diese können aber mit Position 0 als Zuordnungsposition beginnen und heißen

```
TYPE BITVECTOR IS ARRAY(NATURAL RANGE <> ) OF BIT;
```

Worte, Vektoren und Arrays

➤ Vektoren:

Betrachtet man Folgen von Zeichen mit stets fester Länge n , so kann man diese wie Vektoren aus einem Vektorraum der Dimension n auffassen. Solche Vektormengen deklariert man als

```
TYPE Xvector32 IS ARRAY(0 TO 31) OF X;
```

Operationen auf Wörtern

Konkatenation (Hintereinanderreihung)

$$conc : A^+ \times A^+ \rightarrow A^+$$

mit $conc(u, v) : [1 : |u| + |v|] \rightarrow A$

und $conc(u, v)(i) = \begin{cases} u(i) : 1 \leq i \leq |u| \\ v(i - |u|) : |u| < i \leq |u| + |v| \end{cases}$

Einfachere Infixschreibweise: $u \cdot v := conc(u, v)$

-- WüHDL: u & v

Operationen auf Worten

Teilwort

$$sstr : A^+ \times \mathbf{N} \times \mathbf{N} \rightarrow A^+$$

mit $sstr(u, i, j) : [1 : j - i + 1] \rightarrow A$

und $sstr(u, i, j)(l) = u(i+l-1)$

$sstr(u, i, j)$ ist undefiniert, wenn $i > j$ oder $j > |u|$

Einfachere Schreibweise: $u[i:j] := sstr(u, i, j)$

Anmerkung: $u[i:j]$ liefert $u(i) \dots u(j)$

-- WüHDL: u(i TO j)

Leeres Wort $\varepsilon : \{\} \rightarrow A$ ($\varepsilon : [1 : 0] \rightarrow A$)

Anmerkung: $u \cdot \varepsilon = u = \varepsilon \cdot u$

-- WüHDL: Kein Pendant!

Die Algebra der Worte

A^+ zusammen mit dem leeren Wort bildet unter der Konkatenation ein *Monoid*

$$A^* := A^+ \cup \{\varepsilon\}$$

Axiome:

- $\forall a,b,c \in A^*: a \cdot (b \cdot c) = (a \cdot b) \cdot c$
- $\forall a \in A^*: a \cdot \varepsilon = a = \varepsilon \cdot a$

Weitere Operationen auf Worten

Fortsetzung einer binären Operation auf Mengen:

Seien $U, V \subseteq A^*$ dann setze

$$U \cdot V := \{a \cdot b \mid a \in U, b \in V\}$$

Definition

Seien $v, w \in A^*$

v heißt **Präfix** von w : $\Leftrightarrow \exists u \in A^*: v \cdot u = w$

v heißt **Suffix** von w : $\Leftrightarrow \exists u \in A^*: u \cdot v = w$

v heißt **Infix** von w : $\Leftrightarrow \exists a, b \in A^*: a \cdot v \cdot b = w$

Ein Xfix v von w heißt **echt** : $\Leftrightarrow v \neq w$

1.1.3 Kodierungen

Häufig muss man Zeichen selbst über anderen Zeichensätzen darstellen.

Definition

Gegeben sei $\varphi : A \rightarrow B^*$

Dann ist die Fortsetzung von φ auf A^*

$$\varphi : A^* \rightarrow B^*$$

gegeben durch

$$\varphi(w) := \varphi(w(1)) \dots \varphi(w(|w|)), \text{ sowie } \varphi(\varepsilon) := \varepsilon$$

Anmerkung: Eine solche Fortsetzung nennt man auch Homomorphismus, da

$$\varphi(u \cdot v) = \varphi(u) \cdot \varphi(v), \text{ und } \varphi(\varepsilon) = \varepsilon$$

Kodierungen -- ff

Definition

Eine Abbildung $\varphi : A \rightarrow B^*$ heißt **Kodierung**
dann und nur dann, wenn die Fortsetzung $\varphi : A^* \rightarrow B^*$
injektiv ist.

Bezeichnung: Wir nennen $X \subseteq A^*$ **präfixfrei**
 $\forall a, b \in X: a$ ist nicht Präfix von b

Lemma:

$\varphi : A \rightarrow B^*$, injektiv, liefert eine Kodierung, wenn

- (a) $\exists n \in \mathbf{N} \quad \forall a \in A : |\varphi(a)| = n$ Blockcode
- (b) $\varphi(A)$ ist präfixfreie Menge präfixfreier Code

1.1.4 Sprachen

Worte sind zunächst Zeichenreihen ohne jede Bedeutung.
Man bildet daraus **Sprachen**, indem man sagt

- welche Zeichenreihen (Worte) Bedeutungen haben
- welche Aneinanderreihungen erlaubt sind.

D.h. man legt die Struktur, die **Syntax**, fest.

Danach definiert man die Bedeutung

- der Worte
- und der zulässigen Aneinanderreihungen,
d.h. man legt auf der Struktur dann die Bedeutung, die
Semantik, fest.

Syntaxdefinition in BNF

Um die Syntax von Sprachen festzulegen, benutzen wir selbst eine einfache Sprache, die wir natürlich über unserem alphanumerischen Zeichensatz definieren müssen: die **Backus Naur Form** (BNF).

Die Grundlage bilden Symbole, die interpretiert werden als Wortmengen, d.h. als Teilmengen von A^* . Wir nennen sie **Wortmengenbezeichner** (Nichtterminale) und lassen dazu beliebige Buchstabenfolgen, eingefasst durch $< >$, zu.

Ferner benutzen wir das **Definitionssymbol** " $::=$ ". Mit diesem Symbol definieren wir die Wortmenge zu einem Wortmengenbezeichner unter Zuhilfenahme anderer Wortmengenbezeichner und Operatoren.

Anmerkung: Da wir zur Definition auch Zeichen benutzen müssen und diese meist aus dem gleichen Zeichensatz stammen, über dem die Zielsprache definiert wird, kennzeichnen wir die Zeichen der Zielsprache durch fette Schrift.

Syntaxdefinition -- ff

Die eigentlichen Wortmengen definieren wir nun durch Ausdrücke über Wortmengenbezeichnern, Zeichen und folgenden

Operatoren:

- Hintereinanderschreiben: Konkatenation der Wortmengen
- | Vereinigung von zwei Wortmengen
- { } Klammern für Ausdrücke
- * *Abschluss einer Wortmenge X , d.h. man betrachtet die Menge aller Worte, die man durch Aneinanderreihung von Wörtern aus X bilden kann, einschliesslich des leeren Worts.
- + +Abschluss: Wie *Abschluss aber ohne leeres Wort.

Häufig benutzt man noch weitere Operatoren:

So bedeutet etwa $\{ \mathbf{a} | \mathbf{ab} \} \mathbf{c}$, die Menge $\{ac, abc\} = \{a,ab\} \{c\}$.

Das Zeichen **b** folgt also **a** "optional".

Dies kann man auch durch einen weiteren Operator [] ausdrücken, mit $X [Y]$ gleichbedeutend mit $\{X | XY\}$.

Syntaxdefinition -- ff

Beispiele: Die Syntax von ganzen Dezimalzahlen kann man in BNF direkt wie folgt angeben

<IntegerKonstante>

```
 ::= [+ | -] {0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }+
```

Die Syntax von Bitvektorkonstanten sogar noch einfacher durch

<BitvektorKonstante> ::= "{0 | 1 }+"

Arithmetische Ausdrücke kann man durch

<ArithExpr> ::= <ArithExpr> {+ | -} <Term> | <Term>

<Term> ::= <Term> {*} | / <Factor> | <Factor>

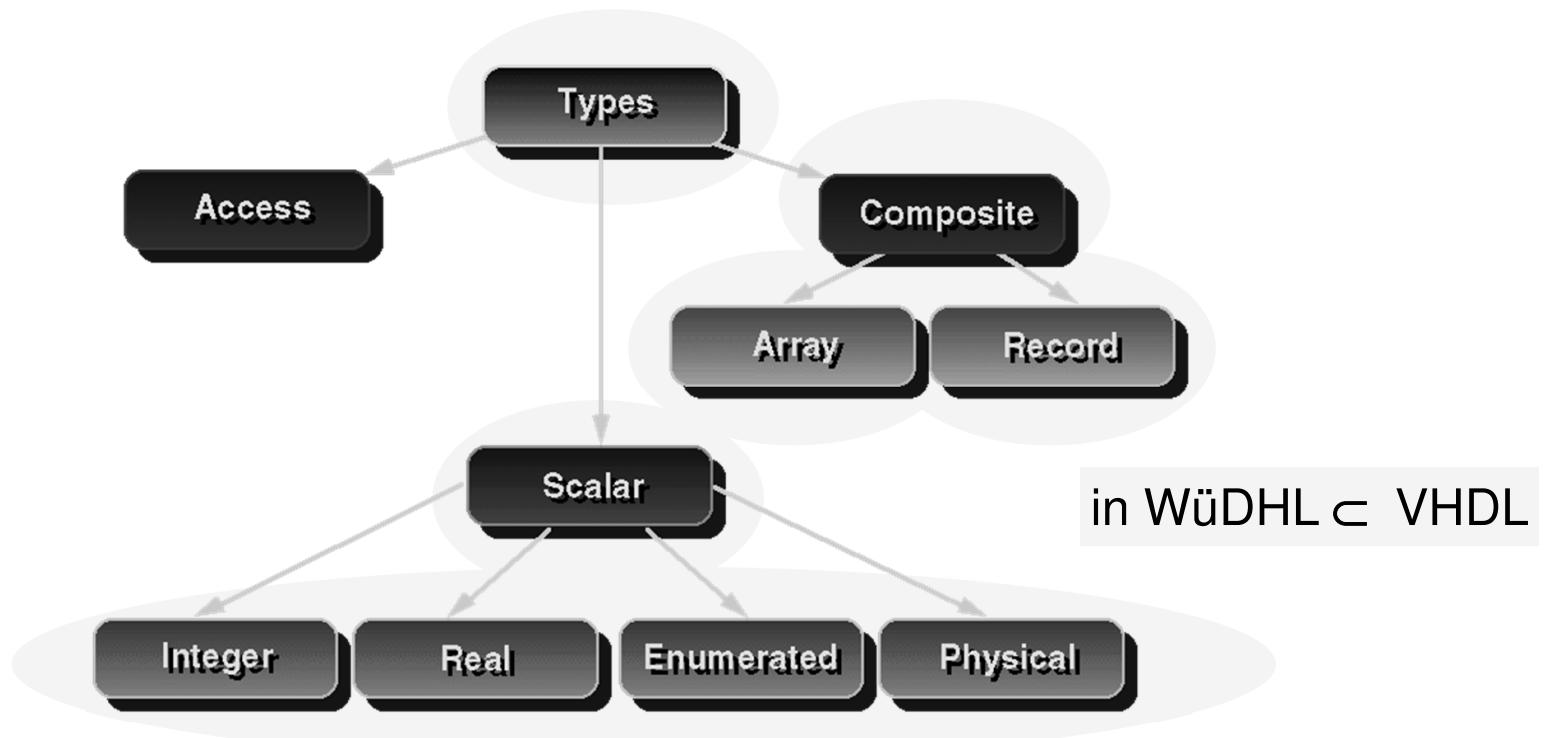
<Factor> ::= <Valueldent> | (<ArithExpr>)

<Valueldent> ::= <Constant> | <Identifier> | <ArrayExpr>

1.1.5 Typen in WüHDL

Typen stehen für Wertebereiche, über denen andere Objekte Werte annehmen können. Sie werden in Typdeklarationen definiert.

Wir wollen die wichtigsten Typen in WüHDL bzw. VHDL zusammenfassen:



Skalare Datentypen

Die Enumerationstypen BIT, BOOLEAN, CHARACTER haben wir schon behandelt.

Zu weiteren vorgegebenen Typen

INTEGER:

Garantierter Bereich **bei jeder VHDL Implementierung:**
-2.147.483.647 bis +2.147.483.647

REAL :

Garantierter Bereich
-1.0e38 bis +1.0e38
mit wenigstens 6 Dezimalstellen Genauigkeit

Skalare Datentypen: physikalische Typen

Physikalische Typen (physical) sind Skalare Typen, die zusätzlich mit Einheiten versehen sind.

Zur Definition eines physikalischen Typs gehört

- ein **Bereich** eines anderen skalaren Typs
- eine Definition der **Einheiten**.

Beispiel

```
TYPE TIME IS RANGE -9223372036854775807 TO +9223372036854775807
UNITS
    fs;                      -- femtosecond
    ps = 1000 fs;            -- picosecond
    ns = 1000 ps;            -- nanosecond
    us = 1000 ns;            -- microsecond
    ms = 1000 us;            -- millisecond
    sec = 1000 ms;           -- second
    min = 60 sec;            -- minute
    hr = 60 min;             -- hour
END UNITS;
```

```
Process (X)
variable a: TIME;
begin
    a := 1; -- bad
    a := 1 ps; -- OK
end process;
```

Vordefiniert in WüHDL ist nur der physikalische Typ TIME

Vordefinierte Typen und Funktionen

Package Standard

```
PACKAGE STANDARD IS
    TYPE BOOLEAN IS (FALSE, TRUE);
    TYPE BIT IS ('0', '1');
    TYPE CHARACTER IS (
        NUL, SH, STX, EOT, ENQ, ACK, BEL,
        ....
        'x', 'y', 'z', '{', '|', '}', '~', DEL );
    TYPE SEVERITY_LEVEL IS
        (NOTE, WARNING, ERROR, FAILURE);
    TYPE INTEGER IS RANGE -2147483648 TO +2147483647;
    TYPE REAL IS RANGE -0.1797693134862E+309 TO
        +0.1797693134862E+309;
    TYPE TIME IS ....
```

Package Standard (2)

```
FUNCTION NOW RETURN TIME;  
SUBTYPE NATURAL IS INTEGER RANGE 0 TO INTEGER'HIGH;  
SUBTYPE POSITIVE IS INTEGER RANGE 1 TO INTEGER'HIGH;  
TYPE STRING IS ARRAY ( POSITIVE RANGE <> ) OF  
CHARACTER;  
TYPE BIT_VECTOR IS ARRAY ( NATURAL RANGE <> ) OF BIT;  
END STANDARD
```

- Die Realisierung des Standard Packages ist versteckt. Es ist immer vorhanden.
- Standardisiert ist auch die IEEE Library, die auf dem Standard Package aufsetzt, und weitere Packages bereitstellt.

1.1.6 Was ist Hardware?

Bevor wir uns weiter mit der Einführung von Begriffen und einer Hardwarebeschreibungssprache beschäftigen, sollten wir uns zunächst diese Frage stellen!

Definition:

Hardware besteht aus einer Menge B parallel arbeitender **Bausteininstanzen** und einer Menge S von **Signalen**, die der Kommunikation zwischen den Bausteininstanzen und mit der Außenwelt dienen.

Eine Bausteininstanz $b \in B$ wirkt auf Signale über ihre Ports $P(b)$, ein Signal $s \in S$ verbindet eine Menge $P(s)$ von Ports. Demnach ist die Menge aller Ports

$$P = \bigcup_{b \in B} P(b) = \bigcup_{s \in S} P(s)$$

wobei $\forall b, b' \in B: b \neq b' \Rightarrow P(b) \cap P(b') = \emptyset$ -- Partition von P
und $\forall s, s' \in S: s \neq s' \Rightarrow P(s) \cap P(s') = \emptyset$ -- Partition von P

Bausteine

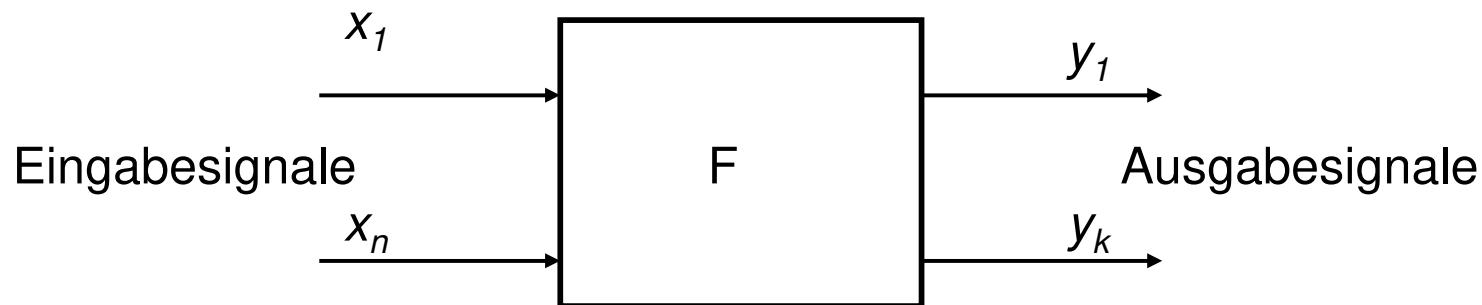
Man wird natürlich viele Bausteininstanzen benutzen, die das gleiche Verhalten haben (Transistoren, Gatter, Latches, ...). Will man ein System über Bausteininstanzen beschreiben, so ist es sinnvoll, nicht für jede Instanz eines Bausteins Ports und Verhalten zu definieren, sondern Bausteine in ihren Eigenschaften (Ports, Verhalten) allgemein zu definieren, und dann Instanzen von ihnen zu benutzen.

Wir werden sehen, dass WüHDL dazu 3 Mechanismen bereitstellt:

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

Systeme und Signale

Ein System ist eine Ansammlung von Komponenten die in ihrem Verhalten zueinander in Wechselwirkung stehen. Das Verhalten des Systems selbst ist seine Wechselwirkung mit seiner Umgebung.



System als Black Box: Verhalten $y = F(x)$

Systeme und Signale

Was sind Signale?

Signale sind Darstellungen von Mitteilungen durch zeitliche Veränderung physikalischer Größen. Sie liegen an physikalischen Orten vor, haben eine physikalische Repräsentation, können zwischen Komponenten kommuniziert werden und ggf. als Ein- oder Ausgabe des Systems beobachtet werden.

In der Regel unterliegt der Wert eines Signals einer Abstraktion.

Was ist ein Signalwert?

Allgemein betrachtet, ist ein **Signalwert (Waveform)** eine Abbildung

$$\beta(s): T \rightarrow V,$$

wobei T eine total geordnete Menge ist, die für die Zeit steht, und V eine Menge ist, die für die Werte steht.

Digitale Signale

Betrachte einen Signalwert (Waveform)

$$\beta(s): T \rightarrow V,$$

eines Signals s .

Ist T abzählbar, dann nennt man s auch **zeitdiskret**,
ist V abzählbar, nennt man s auch ein **wertdiskretes** Signal.

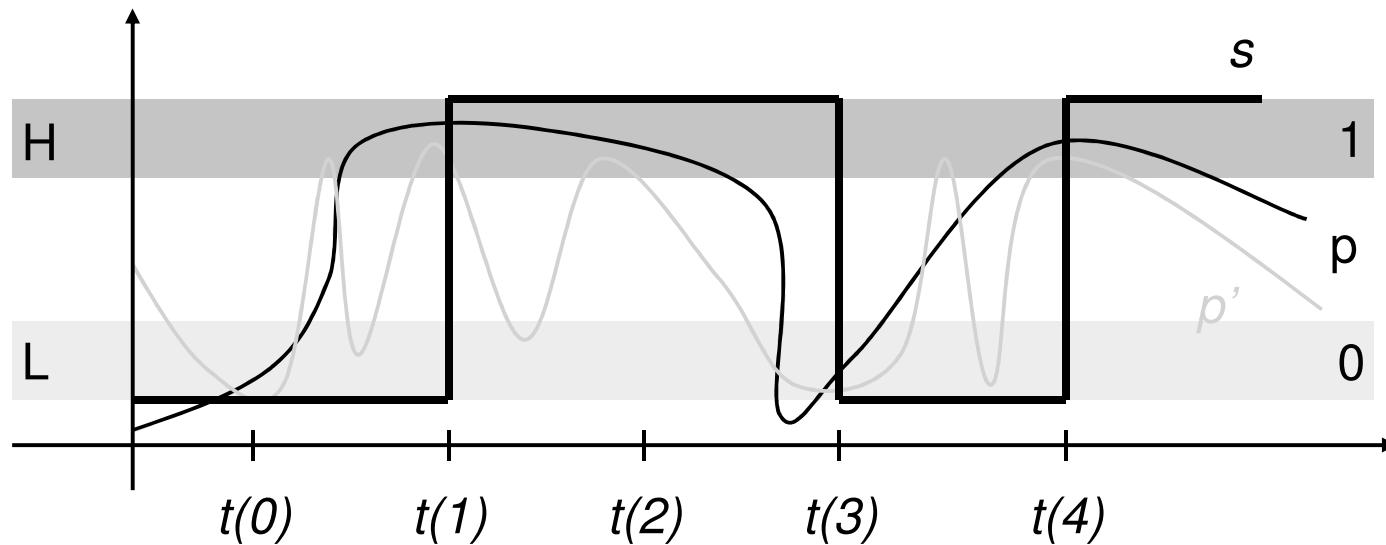
Ein **digitales Signal** ist zeit- und wertediskret mit endlichem Wertebereich. Wir betrachten zunächst nur Signalwerte der Form

$$\beta(s): \mathbb{N}_0 \rightarrow V$$

-- in der Physik betrachtet man hingegen

$$\beta(s): \mathbb{R} \rightarrow \mathbb{R}$$

Abstraktion zu einem digitalen Signal



$$s(i) := \begin{cases} 1 & \text{falls } p(t(i)) \in H \\ 0 & \text{falls } p(t(i)) \in L \\ \infty & \text{sonst} \end{cases}$$

p' liefert die gleiche Abstraktion als digitales Signal!

Klassifikation digitaler Systeme

Ein digitales System ist ein System, dessen Verhalten über digitalen Signalen beschrieben ist.

Es heißt **kombinatorisch** $\Leftrightarrow \forall t: y(t) = F(x(t))$

und **sequentiell** $\Leftrightarrow \forall t: y(t) = F(x(0:t))$

Wir werden uns im Verlauf der Vorlesung zunächst mit dem Entwurf kombinatorischer und dann mit dem Entwurf sequentieller Systeme beschäftigen.

1.2 Verbände und boolesche Algebra

Zur Vorlesung Rechenanlagen

SS 2019



1.2.1 Partielle Ordnungen

Zur Erinnerung:

Definition

Eine Relation $R \subseteq M \times M$ heißt **partielle Ordnung** auf M

- \Leftrightarrow 1. $\forall m \in M : (m, m) \in R$ (Reflexivität)
- 2. mRn und $nRm \Rightarrow n = m$ (Antisymmetrie)
- 3. mRn und $nRp \Rightarrow mRp$ (Transitivität)

Beispiel:

Die Relation Präfix ist eine partielle Ordnung auf A^*

Präfix $\subseteq A^* \times A^*$, mit $(v, w) \in \text{Präfix} \Leftrightarrow \exists u \in A^* : w = vu$

Supremum und Infimum

Definition

Sei M eine Menge, und sei \leq partielle Ordnung. Dann heißt (M, \leq) eine **partiell geordnete Menge** (partially ordered set) oder **Poset**.

Sei $U \subseteq M$. Ein Element $c \in M$ mit

$\forall u \in U: c \leq u$ heißt **untere Schranke** von U

$\forall u \in U: u \leq c$ heißt **obere Schranke** von U

Ein Element $c \in M$ heißt **Supremum (Infimum)** von U
(Schreibe $\sup U$ ($\inf U$))

- \Leftrightarrow (i) c ist obere (untere) Schranke von U
(ii) für jede obere (untere) Schranke m von U ist
schon $c \leq m$ ($m \leq c$).

Supremum und Infimum -- ff

Bemerkung:

Das Supremum von U (Infimum) ist also die kleinste obere (größte untere) Schranke einer Menge U .

Existiert zu U ein Supremum (Infimum), so ist es eindeutig bestimmt:

Annahme: c und c' seien beide Suprema von U

Dann ist $c \leq c'$, (c ist Supremum, c' obere Schranke)

aber auch $c' \leq c$, (c' ist Supremum, c obere Schranke)

also: $c' = c$, (Antisymmetrie)

1.2.2 Halbverbände

Definition

Eine Poset (M, \leq) heißt **oberer (unterer) Halbverband**
(upper / lower semi lattice)

$$\Leftrightarrow \forall \{a, b\} \subseteq M \exists c \in M : c = \sup\{a, b\}$$
$$(\forall \{a, b\} \subseteq M \exists c \in M : c = \inf\{a, b\})$$

Schreibweisen:

Wir schreiben auch $\sup(a, b)$ statt $\sup\{a, b\}$
und als Infixoperator $\sup(a, b) := a \vee b$ Lies: „oder“

Ebenso $\inf(a, b)$ statt $\inf\{a, b\}$
und als Infixoperator $\inf(a, b) := a \cdot b$ Lies: „und“

Halbverbände ff

Mit dieser Schreibweise gilt in einem oberen (unteren) Halbverband für je zwei Elemente a, b :

$$a \leq b \Leftrightarrow a \vee b = b \quad (a \leq b \Leftrightarrow a \cdot b = a)$$

Definition

Ein Halbverband (M, \leq) heißt **vollständig** (complete) : \Leftrightarrow

Zu jeder (endlichen oder unendlichen) Teilmenge $U \subseteq M$ existiert das Supremum (Infimum).

Bemerkung:

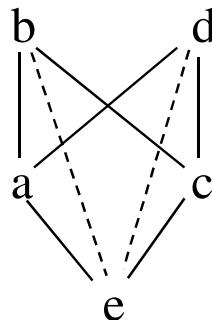
Ein vollständiger oberer (unterer) Halbverband hat stets ein größtes (kleinstes) Element: $\top := \sup M$ ($\perp := \inf M$)

1.2.3 Das Hasse Diagramm

Man kann ein endliches Poset durch folgendes Diagramm, genannt Hasse Diagramm, darstellen:

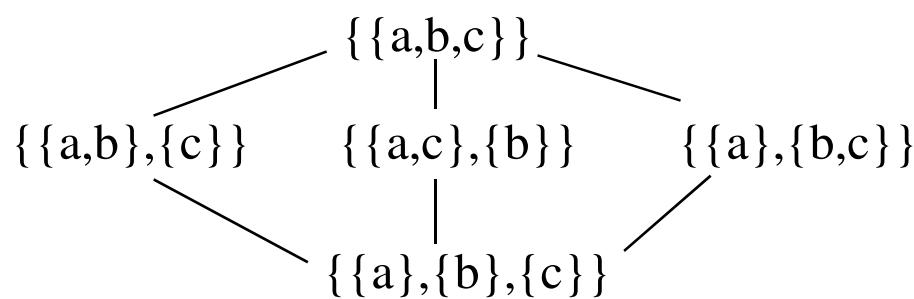
- Knotenmenge \mathcal{P}
- gibt es eine Kante zwischen Element a und b , so gilt $a \leq b$ oder $b \leq a$. Der Knoten, der weiter unten liegt, ist kleiner.
- Es wird nur der reduzierte Graph gezeichnet, d.h. transitive Kanten lässt man weg

Beispiel:

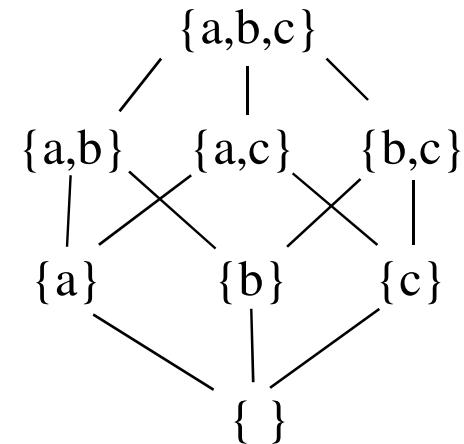


Hasse Diagramme: Beispiele für weitere Posets

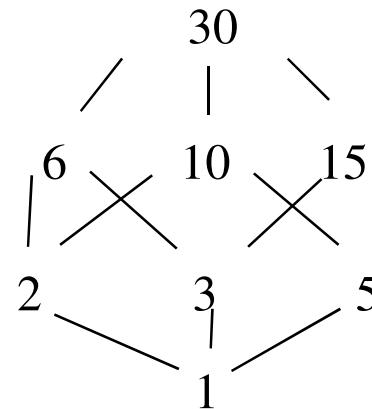
($\{\pi \mid \pi$ ist Partition von $\{a,b,c\}\}$, “ist feiner”)



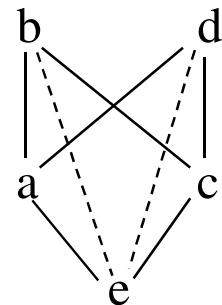
($2^{\{a,b,c\}}, \subseteq$)



($\{1,2,3,5,6,10,15,30\}$, “ist Teiler”)



Beispiel:



$$a \vee d = d$$

$b \vee d$ = nicht definiert, da kein gemeinsamer Vorfahre

$b \cdot d$ = nicht definiert, da nicht eindeutig

1.2.4 Verbände

Definition

Ein Poset (M, \leq) heißt (vollständiger) **Verband** (lattice)

$\Leftrightarrow (M, \leq)$ ist ein (vollständiger) oberer und unterer Halbverband.

Beispiel:

Sei M endliche Menge und $2^M := \{U \mid U \subseteq M\}$

die **Potenzmenge** von M . Dann gilt für beliebige U, V, W :

$$U \subseteq U$$

$U \subseteq V$ und $V \subseteq U \Rightarrow U = V$ $(2^M, \subseteq)$ ist also ein Poset

$$U \subseteq V \text{ und } V \subseteq W \Rightarrow U \subseteq W$$

Beispiel ff:

Sei U beliebige Teilmenge von 2^M

Dann hat U als Supremum die Menge $\bigcup_{u \in U} u$

denn $\forall v \in U : v \subseteq \bigcup_{u \in U} u$ } (obere Schranke!)

und gilt $\forall u \in U : u \subseteq y$,
so folgt schon $\bigcup_{u \in U} u \subseteq y$ } (kleinste o. S.!)

Insbesondere ist

$$\forall a, b \in 2^M : \sup(a, b) = a \cup b$$

Also ist die Potenzmenge ein vollständiger, oberer Halbverband.

Beispiel ff:

Analog überzeugt man sich davon, daß für $U \subseteq 2^M$

das Infimum die Menge $\bigcap_{u \in U} u$ ist:

$$\left(\begin{array}{l} \forall v \in U : v \supseteq \bigcap_{u \in U} u \\ \text{gilt } (\forall u \in U : y \subseteq u), \text{ so folgt } y \subseteq \bigcap_{u \in U} u \end{array} \right)$$

Insbesondere ist $\forall a, b \in 2^M : \inf(a, b) = a \cap b$

Also ist die Potenzmenge auch ein vollständiger, unterer Halbverband, und damit ein vollständiger Verband. Wir nennen ihn auch einfach den **Mengenverband über M**.

kleinstes Element: $\perp = \{ \}$ größtes Element: $\top = M$

Rechenregeln in Verbänden

Satz

Sei (M, \leq) ein Verband. Dann gilt für alle $a, b, c \in M$

$$(V1) \quad a \vee b = b \vee a \quad (\text{Kommutativität})$$

$$a \cdot b = b \cdot a$$

$$(V2) \quad a \vee (b \vee c) = (a \vee b) \vee c \quad (\text{Assoziativität})$$

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c$$

$$(V3) \quad a \vee (a \cdot b) = a \quad (\text{Absorption})$$

$$a \cdot (a \vee b) = a$$

Beweis: Wir zeigen nur eine Auswahl der Gleichungen.

Kommutativität: $a \vee b := \sup\{a, b\} = \sup\{b, a\} =: b \vee a$

Beweis ff

Assoziativitat:

Es ist:

$$\left. \begin{array}{l} \sup\{a, b\} \leq \sup\{a, b, c\} \\ c \leq \sup\{a, b, c\} \end{array} \right\} \Rightarrow \sup\{\sup\{a, b\}, c\} \leq \sup\{a, b, c\}$$

Umgekehrt ist:

$$a, b, c \leq \sup\{\sup\{a, b\}, c\} \Rightarrow \sup\{a, b, c\} \leq \sup\{\sup\{a, b\}, c\}$$

Damit ist: $\underbrace{\sup\{\sup\{a, b\}, c\}}_{\text{(Antisymmetrie)}} = \sup\{a, b, c\} = \underbrace{\sup\{a, \sup\{b, c\}\}}_{\text{(analog)}}$

Also:
$$\begin{aligned} (a \vee b) \vee c &= \sup\{\sup\{a, b\}, c\} \\ &= \sup\{a, b, c\} \\ &= \sup\{a, \sup\{b, c\}\} = a \vee (b \vee c) \end{aligned}$$

Beweis ff

Absorption:

Hier ist: $a \leq \sup\{a, \inf\{a, b\}\}$

Andererseits ist: $\inf\{a, b\} \leq a$

$a \leq a$

$\left. \begin{array}{l} a \leq \sup\{a, \inf\{a, b\}\} \\ \inf\{a, b\} \leq a \\ a \leq a \end{array} \right\} \Rightarrow \sup\{a, \inf\{a, b\}\} \leq a$

$\left. \begin{array}{l} a \leq \sup\{a, \inf\{a, b\}\} \\ \inf\{a, b\} \leq a \\ a \leq a \end{array} \right\} \Rightarrow \sup\{a, \inf\{a, b\}\} = a$ (Antisymmetrie)

Also: $a \vee (a \cdot b) = \sup\{a, \inf\{a, b\}\}$

$$= a$$

Die anderen Beziehungen zeigt man analog!

Dualität

Beobachtung:

Die Regeln (V1), (V2), (V3) sind **dual** unter der Vertauschung von „und“ und „oder“. Damit gilt für jede Formel, die man mit (V1), (V2), (V3) herleiten kann, auch zugleich die **duale Formel**, die man durch Tauschen von „und“ mit „oder“ erhält.

Grund:

Da man in der Herleitung stets Regeln aus (V1), (V2), (V3) benutzt, kann man durch Anwendung der dualen Regeln die duale Formel herleiten.

Bei einem dualen Axiomensystem werden wir stets nur einen Beweis führen.

Eigenschaften von Verbänden

Satz

(M_1, \leq_1) und (M_2, \leq_2) seien (vollständige) Verbände. Dann sind auch

(a) $(M_1 \times M_2, \leq_{1,2})$ mit

$$(a, b) \leq_{1,2} (c, d) \Leftrightarrow a \leq_1 c \text{ und } b \leq_2 d$$

(b) $(Abb(X, M_1), \leq)$ mit

$$f \leq g \Leftrightarrow \forall x \in X : f(x) \leq_1 g(x)$$

vollständige Verbände.

Beweis: Übung.

Distributiver Verband

Definition

Ein Verband (M, \vee, \cdot) heißt **distributiver Verband**, dann und nur dann, wenn für alle a, b, c aus M gilt

$$(V4) \quad a \vee (b \cdot c) = (a \vee b) \cdot (a \vee c) \text{ (Distributivität)}$$

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

Bemerkung:

Die Regel (V4) ist auch dual.

Wir vereinbaren für die Operatoren von nun an:

\cdot bindet stärker als \vee .

D.h. der Ausdruck $a \cdot b \vee c$ ist zu lesen als $(a \cdot b) \vee c$.

Wir lassen \cdot auch häufig weg.

1.2.5 Boolesche Algebra

Definition

(M, \vee, \cdot, \neg) heißt **boolesche Algebra**, dann und nur dann, wenn

(A1) (M, \vee, \cdot) ist distributiver Verband

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

$\forall a, b \in M : a \cdot (b \vee \neg b) = a$

Notation: Wir schreiben in Formeln statt $\neg b$ (lies nicht b)

einfacher \bar{b} (lies b quer)

Bemerkung: Auch in der booleschen Algebra sind die Axiome dual!

Eigenschaften boolescher Algebren

Lemma

In einer booleschen Algebra gilt (**Idempotenz**):

$$a \vee a = a; \text{ dual : } a \cdot a = a$$

Beweis:

$$x \vee x = x \vee x \cdot (x \vee x)$$

$$= x$$

Dualer Beweis

$$\begin{aligned} x \cdot x &= x \cdot (x \vee x \cdot x) \\ &= x \end{aligned}$$

Eigenschaften boolescher Algebren

Lemma

Jede boolesche Algebra besitzt genau ein „0“ und ein „1“ Element, die zueinander dual sind, und es gilt:

$$a \vee 0 = a; \text{ dual: } a \cdot 1 = a$$

$$a \cdot 0 = 0; \text{ dual: } a \vee 1 = 1$$

Beweis:

Höchstens eins: Seien $0, 0'$ zwei Nullelemente, dann gilt:

$$0 = 0 \vee 0' = 0'$$

$$\text{Dual: } 1 = 1 \cdot 1' = 1'$$

↑
(0 ist Nullelement)

—
($0'$ ist Nullelement)

Beweis ff

Mindestens eins: Setze für beliebiges b aus M

$$0 := b\bar{b}$$

Dann gilt für alle a aus M :

$$a \vee 0 = a \vee b\bar{b} = a$$

Ferner gilt:

$$\begin{aligned} a \cdot 0 &= a \cdot b\bar{b} \\ &\quad \text{da } b \text{ beliebig} \\ &= a \cdot a\bar{a} \\ &\quad \text{Assoziativitat} \\ &= (\bar{a}a)\bar{a} \\ &\quad \text{Idempotenz und Def. 0} \\ &= \bar{a}\bar{a} = 0 \end{aligned}$$

Eigenschaften boolescher Algebren

Lemma

In einer booleschen Algebra M gibt es zu jedem Element a genau ein Element b , mit

$$a \vee b = 1 \text{ und } a \cdot b = 0$$

Beweis:

Man nehme an, dass b und b' obigen Gleichungen genügen:
dann ist

$$\begin{aligned} b &= (a \vee b')b = ab \vee bb' = bb' = ab' \vee bb' = (a \vee b)b' = b' \\ &\quad \uparrow \qquad \uparrow \qquad \uparrow \qquad \uparrow \\ &\quad (a \vee b' = 1) \quad (ab = 0) \qquad (ab' = 0) \qquad (a \vee b = 1) \end{aligned}$$

Eigenschaften boolescher Algebren

Korollar

Die Gleichungen $a \vee b = 1$ und $a \cdot b = 0$ gelten nur für $b = \bar{a}$

Wir nennen dieses Element $b = \bar{a}$ auch **das Komplement** von a .

Satz (de Morgan'sche Regel)

In einer booleschen Algebra M gilt für alle Elemente a, b :

$$\overline{a \vee b} = \bar{a} \cdot \bar{b} \text{ und dual } \overline{a \cdot b} = \bar{a} \vee \bar{b}$$

Beweis der de Morgan Regel

Wir zeigen nach dem letzten Korollar, dass

$$ab \cdot (\bar{a} \vee \bar{b}) = 0 \text{ und } ab \vee (\bar{a} \vee \bar{b}) = 1$$

Dann folgt die erste Regel. Der Beweis der dualen Regel ist dual.

$$\begin{aligned} ab \cdot (\bar{a} \vee \bar{b}) &= \overbrace{ab\bar{a}}^{=0} \vee \overbrace{abb}^{=0} \\ &= 0 \vee 0 = 0 \end{aligned}$$

$$\begin{aligned} ab \vee (\bar{a} \vee \bar{b}) &= (ab \vee \bar{a}) \vee \bar{b} \\ (\text{Distributivit\"at}) \quad &= (a \vee \bar{a})(b \vee \bar{a}) \vee \bar{b} \\ &= 1 \vee \underbrace{(b \vee \bar{a})}_{=1} \vee \bar{b} = 1 \end{aligned}$$

Korollar

$$\bar{0} = 1 \text{ und } \bar{1} = 0$$

Beweis:

$$\bar{0} = \overline{\overline{aa}} = \overline{\overline{a}} \vee \overline{\overline{a}} = 1$$

$$\bar{1} = \overline{\overline{a \vee a}} = \overline{\overline{a}} \cdot \overline{\overline{a}} = 0$$

Eigenschaften boolescher Algebren

Satz

Seien M_1, M_2 boolesche Algebren und sei A eine Menge.

Dann sind auch

(a) $(2^A, \cup, \cap, \neg)$

(b) $(M_1 \times M_2, \vee, ;, \neg)$ mit

$$(a,b) \vee (c,d) := (a \vee c, b \vee d);$$

$$(a,b) \cdot (c,d) := (a \cdot c, b \cdot d); \quad \overline{(a,b)} := (\overline{a}, \overline{b})$$

(c) $Abb(A, M_1)$ mit

$$(f \vee g)(a) := f(a) \vee g(a);$$

$$(f \cdot g)(a) := f(a) \cdot g(a); \quad \bar{f}(a) := \overline{f(a)}$$

jeweils boolesche Algebren.

Beweis: Übung

Beispiele:

Was hat das alles nun mit Digitaltechnik zu tun?

Die Menge \mathbf{B} ist eine boolesche Algebra, die nur aus der 0 und der 1 besteht (sie ist die kleinste b.A. überhaupt).

Nach dem letzten Satz bilden dann aber auch die Mengen

$$\mathbf{B}^n := \underbrace{\mathbf{B} \times \cdots \times \mathbf{B}}_{n-mal}$$

unter komponentenweiser Verknüpfung, sowie die Mengen

$$\mathbf{S}_{n,k} := Abb(\mathbf{B}^n, \mathbf{B}^k)$$

eine boolesche Algebra. Wir nennen $\mathbf{S}_{n,k}$ auch die Menge der (totalen) k -stelligen **Schaltfunktionen**.

Beispiele ff:

Ebenso bildet die Menge der partiellen Funktionen mit gleichem Definitionsbereich D :

$$F(\mathbf{B}^n, \mathbf{B}^k) \supseteq \mathbf{S}_{n,k}^D := Abb(D, \mathbf{B}^k)$$

eine boolesche Algebra. Die 0 und 1 der Algebra sind gerade die konstanten Funktionen.

Die Grundbausteine der Digitaltechnik berechnen Schaltfunktionen. Da Schaltfunktionen eine boolesche Algebra bilden, ist diese Theorie also ein wichtiges Fundament zur Analyse von digitalen Schaltkreisen.

Atome

Definition

Ein Element a einer boolesche Algebra M heißt **Atom** genau dann, wenn

$$(At1) \quad a \neq 0$$

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

Bemerkung: Atome sind kleinste, von 0 verschiedene Elemente in der booleschen Algebra. Sie sind nicht weiter aufspaltbar, daher der Name „Atom“.

Eigenschaften von Atomen

Lemma

$$a \in M \text{ ist Atom} \Leftrightarrow \forall b \in M : b \leq a \Rightarrow (b = 0 \text{ oder } b = a)$$

Beweis:

\Rightarrow : Sei a Atom und sei $b \leq a$

$$b \leq a \Leftrightarrow b = ba = ab \in \{0, a\}$$

\Leftarrow : Es gelte nun: $\forall b \in M : b \leq a \Rightarrow (b = 0 \text{ oder } b = a)$

Sei c aus M beliebig, dann ist $ac \leq a$.

Insbesondere gilt also für $b = ac$: $ac = 0$ oder $ac = a$,
also ist a Atom.

Eigenschaften von Atomen ff

Lemma

a, b Atome mit $a \neq b$, dann gilt $ab = 0$

Beweis:

$ab \in \{a, 0\}$, da a Atom

$$\Rightarrow ab \in \{a, 0\} \cap \{b, 0\} = \{0\}$$

$ab \in \{b, 0\}$, da b Atom

Beispiele:

1. Die Atome von $(2^A, \cup, \cap, \neg)$ sind $At(2^A) = \{\{a\} \mid a \in A\}$

2. Die Atome von $\mathbf{S}_n := \mathbf{S}_{n,1}$

Für einen Punkt $p \in \mathbf{B}^n$ sei $x^p : \mathbf{B}^n \mapsto \mathbf{B}$ die Funktion, die nur auf p den Wert 1 hat, d.h.

$$x^p(q) := \begin{cases} 1 & \text{für } p = q \\ 0 & \text{für } p \neq q \end{cases}$$

Wir nennen diese Funktion auch einfach **Minterm** zu p oder **Punkt** p .

Dann ist x^p Atom, denn für eine beliebige Funktion

$$f \in \mathbf{S}_n \text{ gilt: } f \cdot x^p(q) = \begin{cases} f(p) \cdot 1 & \text{für } p = q \\ 0 & \text{für } p \neq q \end{cases} = \begin{cases} x^p(q) & \text{für } f(p) = 1 \\ 0 & \text{für } f(p) = 0 \end{cases}$$

Weitere Eigenschaften

Atome spielen eine wichtige Rolle. Man kann zeigen, dass jedes Element einer endlichen booleschen Algebra aus Atomen eindeutig zusammengesetzt ist:

Satz

Sei M eine endliche boolesche Algebra. Dann gilt für ein beliebiges Element f stets:

$$f = \bigvee_{\substack{a \in At(M) \\ a \cdot f \neq 0}} a$$

Wir zeigen den Beweis über einige Hilfsbehauptungen:

Beweis:

Behauptung 1: Sei $b \neq 0$ beliebig, dann gibt es stets ein Atom a_0 mit $a_0 \leq b$

Fall 1: $\forall a \in M, a \neq b : ab \in \{0, b\}$

Dann ist b schon ein Atom, insbesondere ist $a_0 := b \leq b$

Fall 2: $\exists a \in M : ab \notin \{0, b\}$

Dann ist aber $0 \neq ab \leq b$

Setze $b_2 = ab$ und wiederhole die Falldiskussion für b_2

Wir erhalten also eine Folge, $b=b_1 \geq b_2 \geq \dots \geq b_n = a_0$ die irgendwann mit Fall 1 abbricht (M endlich).

Also gilt Behauptung 1.

Beweis ff:

Behauptung 2: $\bigvee_{a \in At(M)} a = 1$

Annahme: $x := \bigvee_{a \in At(M)} a \neq 1$

Dann ist $\bar{x} \neq 0$

Also gibt es mit Behauptung 1 ein Atom $a_0 \leq \bar{x}$

Nun ist aber $a_0 \cdot x = a_0 \cdot \bigvee_{a \in At(M)} a$

da ja $a_0 \in At(M)$ $= \bigvee_{a \in At(M)} a_0 a = a_0$

Damit ist aber $a_0 = a_0 \bar{x}$, da $a_0 \leq \bar{x}$

$= a_0 x \bar{x}$, da $a_0 x = a_0$

$= a_0 0 = 0$

↗ a_0 Atom

Beweis ff:

Mit Behauptung 2 rechnet man nun ganz leicht nach:

$$\begin{aligned} f &= f \cdot 1 \\ &= f \cdot \left(\bigvee_{a \in At(M)} a \right) \\ &= \bigvee_{a \in At(M)} f \cdot a \\ &= \bigvee_{\substack{a \in At(M) \\ f \cdot a \neq 0}} a \end{aligned}$$

Weitere Eigenschaften

Korollar

Sei M eine endliche boolesche Algebra. Dann gibt es eine natürliche Zahl n , so dass:

$$\#M = 2^n$$

Beweis: Für ein beliebiges Element f von M gilt nach unserem Satz: $f = \bigvee_{\substack{a \in At(M) \\ f \cdot a \neq 0}} a$

Damit bestimmt die Menge

$$ON(f) := \{a \in At(M) \mid f \cdot a \neq 0\} \subseteq At(M)$$

jedes Element f eindeutig ($ON(f)$ heißt ON-Set von f)

Es gibt aber genau $2^{\#At(M)}$ verschiedene Teilmengen.

Beobachtung und Beispiel

Beobachtung

Aus dem Korollar folgt nun auch, dass man, sofern man alle Atome berechnen kann, mit der „oder“ Operation dann auch alle Elemente als Veroderung ihres ON-Sets darstellen kann.

Beispiel: Betrachten wir wieder die Schaltfunktionen. Die Minterme bilden Atome. Da nun für alle $q \in \mathbf{B}^n$

$$\begin{aligned} \left(\bigvee_{p \in \mathbf{B}^n} x^p \right)(q) &= \bigvee_{p \in \mathbf{B}^n} x^p(q) \\ &= x^q(q) \\ &= 1 \end{aligned}$$

folgt nach obigem Satz sogar $At(\mathbf{S}_n) = \{x^p \mid p \in \mathbf{B}^n\}$

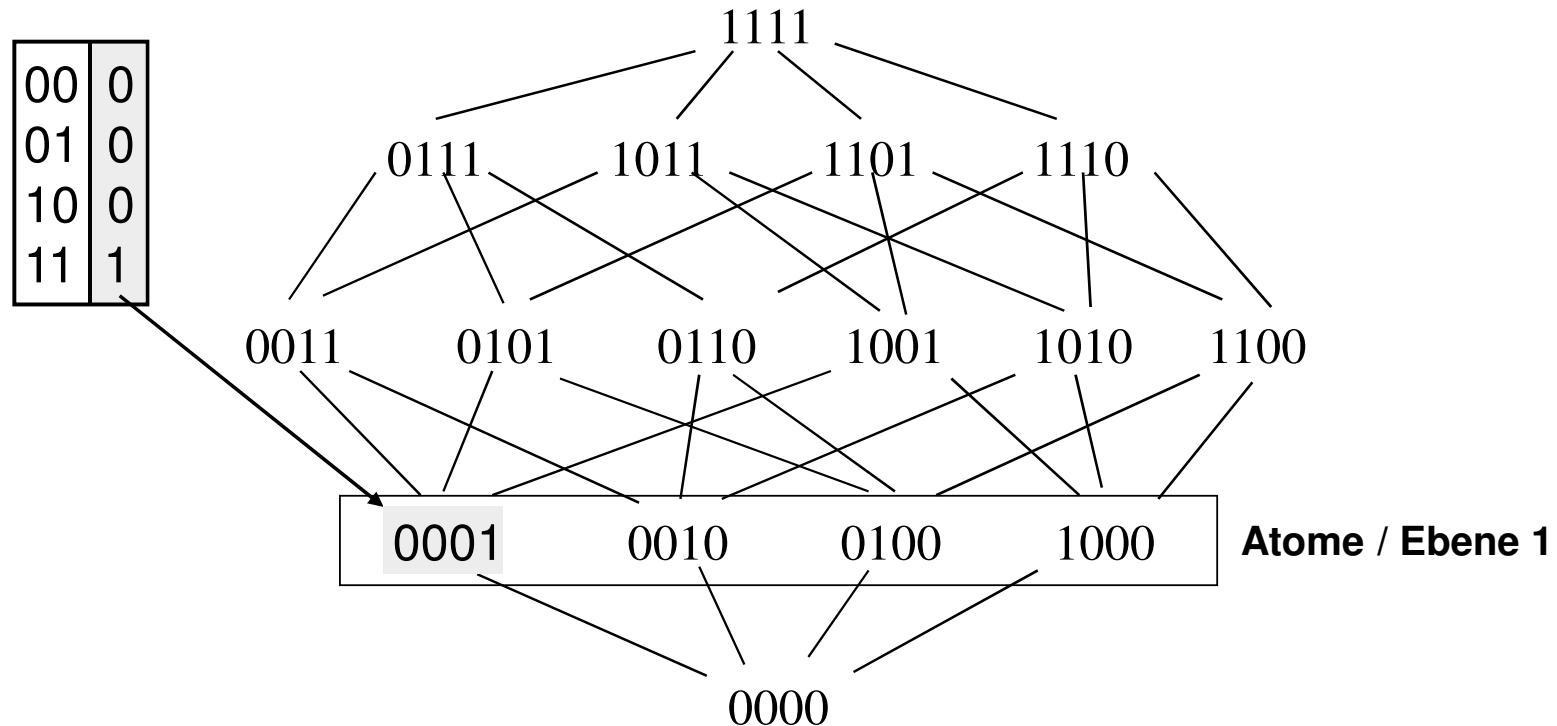
Beispiel ff:

Die Minterme sind damit genau die Atome der Algebra der einstelligen Schaltfunktionen und es gilt für eine beliebige Funktion f stets, dass f darstellbar ist durch

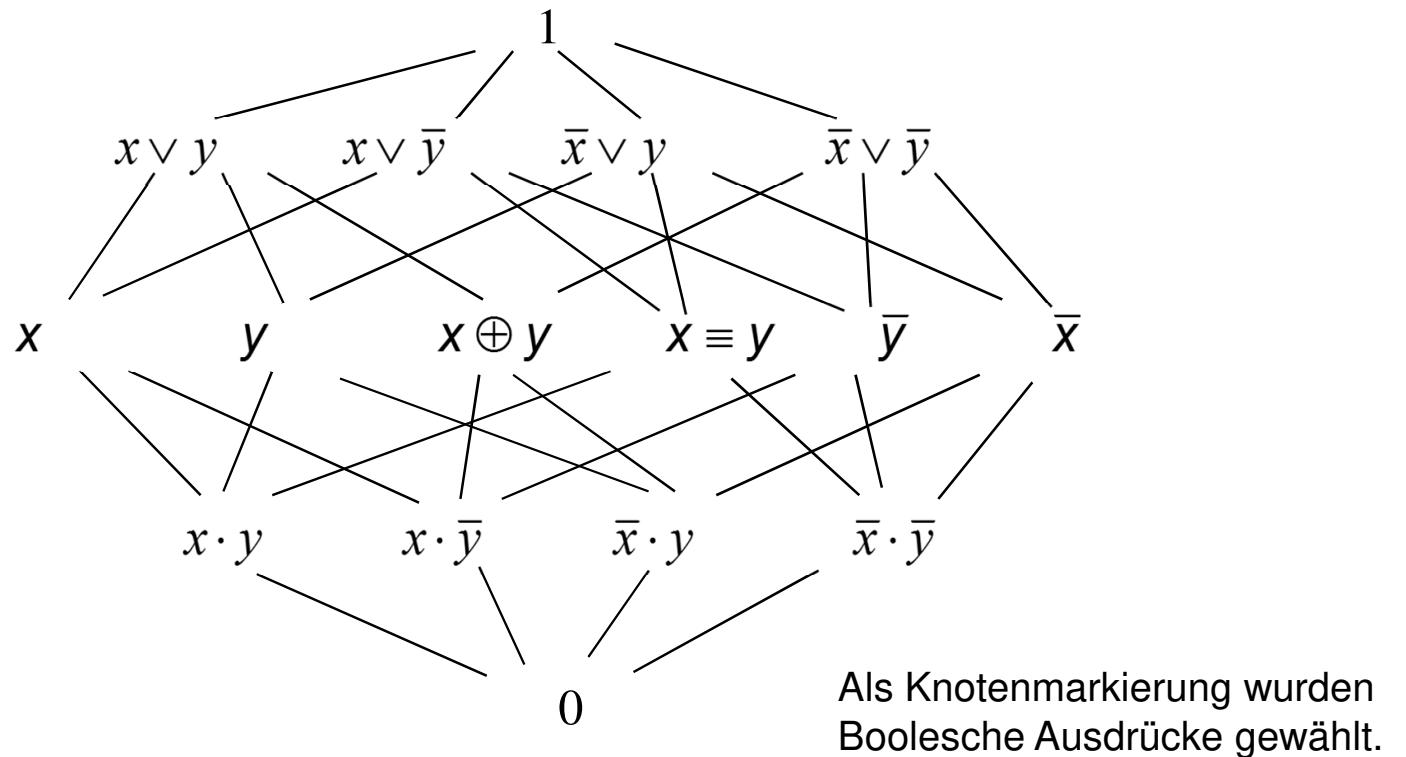
$$\begin{aligned} f &= f \cdot 1 \\ &= f \cdot \left(\bigvee_{p \in \mathbf{B}^n} x^p \right) \\ &= \bigvee_{p \in \mathbf{B}^n} f \cdot x^p \\ &= \bigvee_{p \in \mathbf{B}^n} f(p) \cdot x^p \\ &= \bigvee_{p, f(p)=1} x^p \quad \text{disjunktive Normalform von } f \end{aligned}$$

Hassediagramm der Booleschen Funktionen in 2 Variablen

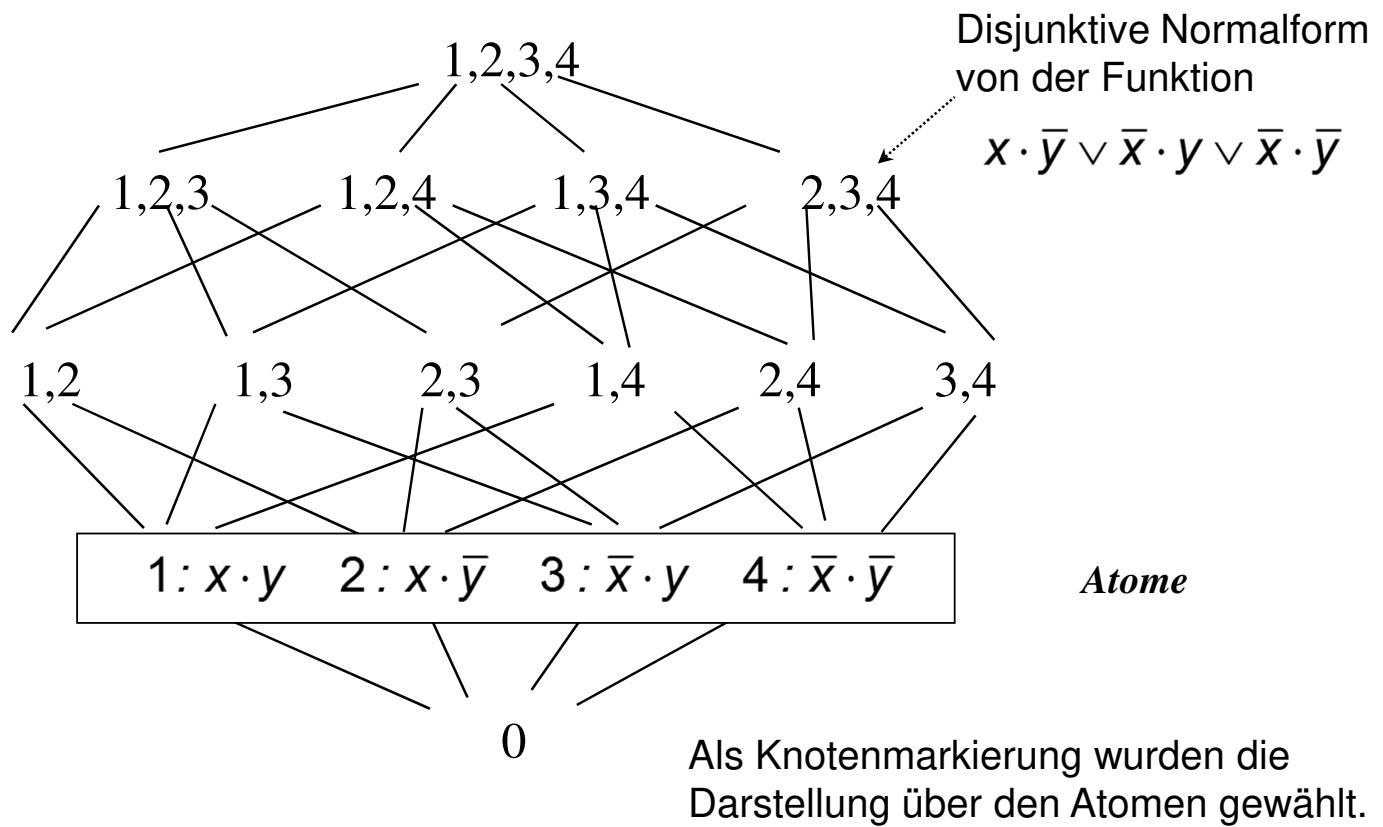
Als Knotenmarkierung wurden die Funktionstafel gewählt.



Hassediagramm der Booleschen Funktionen in 2 Variablen ff



Hassediagramm der Booleschen Funktionen in 2 Variablen ff



1.3 Grundbausteine der Digitaltechnik

Zur Vorlesung Rechenanlagen

SS 2019



1.3.1 Darstellung von Zeichen

Wir müssen Zeichen durch Zustände physikalischer Systeme realisieren.

Grundsatzfrage: Wieviele Zustände braucht man dazu mindestens?

Lemma

Sei A ein endliches Alphabet mit mindestens 2 Zeichen, und sei $\varphi : A^* \rightarrow B^*$ eine Kodierung. Dann hat B mindestens 2 Zeichen.

Beweis: indirekt

Wir nehmen an, dass $\#B=1$, d.h. etwa $B=\{t\}$ und dass

$$\varphi : A^* \rightarrow B^*$$

eine Kodierung ist.

A enthält mindestens 2 Elemente etwa a und b . Sei nun

$$\varphi(a) = t^n \text{ und } \varphi(b) = t^m \text{ für } n, m \in \mathbf{N}_0.$$

Dann ist aber

$$\varphi(ab) = t^{n+m} = t^{m+n} = \varphi(ba),$$

weswegen φ nicht injektiv ist.



Lemma

Sei A ein endliches Alphabet mit mindestens 2 Zeichen, und sei $B=\{0, 1\}$. Dann gibt es stets eine Kodierung

$$\varphi : A^* \rightarrow B^*$$

Beweis:

Wir betrachten die **Binärkodierung** von nichtnegativen Zahlen über B bei fester Wortlänge k :

$$u_k : B^k \rightarrow N_0 \text{ mit} \quad u_k(a_0, \dots, a_{k-1}) = 2^{k-1} \cdot \sum_{i=0}^{k-1} a_i \cdot 2^{-i}$$

u_k ist offenbar injektiv.

Kleinster Wert: $u_k(0, \dots, 0) = 2^{k-1} \cdot \sum_{i=0}^{k-1} 0 \cdot 2^{-i} = 0$

Größter Wert: $u_k(1, \dots, 1) = 2^{k-1} \cdot \sum_{i=0}^{k-1} 2^{-i}$
 $= 2^{k-1} \cdot (2 - 2^{-(k-1)})$
 $= 2^k - 1$

$u_k: \mathbf{B}^k \rightarrow N_0$ ist demnach bijektiv.

Nummeriert man nun $A = \{a_0, \dots, a_{k-1}\}$ einfach durch und setzt:

$$\varphi(a_i) = u_{\lceil \log(n) \rceil}^{-1}(i)$$

so erhält man einen Blockcode.

Anmerkungen:

Wir nennen eine solche Kodierung auch (eine!)
Binärkodierung von A . Allerdings gibt es

$$\binom{2^{\lceil \log \#A \rceil}}{\#A} \cdot (\#A)! \text{ viele verschiedene Binärkodierungen!}$$

Beide Lemmata zusammen besagen, dass eine zweielementige Menge notwendig und hinreichend ist, um alles zu kodieren. Natürlich kann man die Konstruktion auch für ein größeres $B = \{0, \dots, p-1\}$ durchführen, indem man eine p -näre (Basis p , Radix p Darstellung) benutzt:

$$r_{p,k}(u_0 \cdots u_{k-1}) = p^{k-1} \cdot \sum_{i=0}^{k-1} u_i \cdot p^{-i}$$

Warum tut man dies nicht?

Beispiel:

Sie möchten sich bei Dunkelheit und klarem Wetter über eine Sichtentfernung von 2 km mit Lichtzeichen verständigen. Licht an / Licht aus ist gut wahrnehmbar!

Nun haben beide Parteien sich einen Dimmer zugelegt und vereinbaren 4-wertig nach den Zuständen

- Licht stark
- Licht weniger stark
- Licht schwach
- Licht aus

zu kodieren.

Die Erkennung eines Zeichens wird nun viel schwieriger!

Wie auch immer die Darstellung von Zeichen physikalisch aussehen wird, **Störsicherheit** und **Zuverlässigkeit** sind dabei stets entscheidend.

1.3.2. CMOS Technologie

Bei modernen **MOS** (Metal Oxide Semiconductor) Technologien spielen Transistoren als spannungs-gesteuerte Schalter eine herausragende Rolle bei der Entwicklung von Elementarbausteinen.

Das Spiel ist, naiv gesehen, recht einfach:

Interpretiere Spannungen $U(x)$ zwischen x und dem Bezugspol V_{SS} als Werte aus \mathcal{B}

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

Dabei liegt an V_{SS} die Spannung von 0 V zu sich selbst, am Versorgungspol V_{DD} je nach Technologie 1...5V.

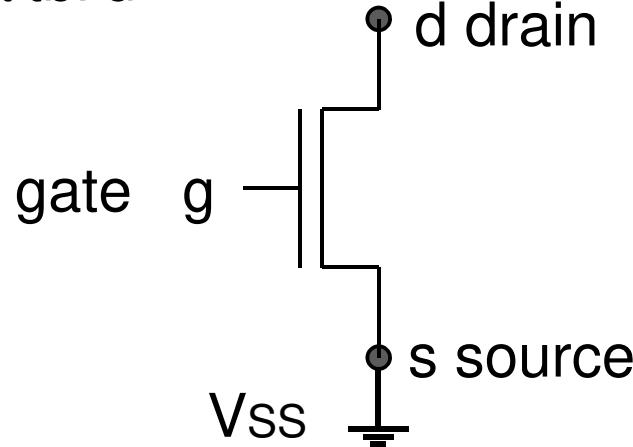
MOS FETs als Schalter

Feldeffekttransistoren (FETs) werden nun so hergestellt, dass sie als spannungsgesteuerte Schalter gegen V_{SS} bzw. V_{DD} arbeiten:

Wir unterscheiden zwei Typen, deren Schalterverhalten wir betrachten wollen:

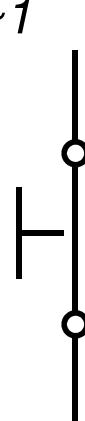
n-Kanal Transistor:

Schaltbild



Verhalten

$U(g) \sim 1$

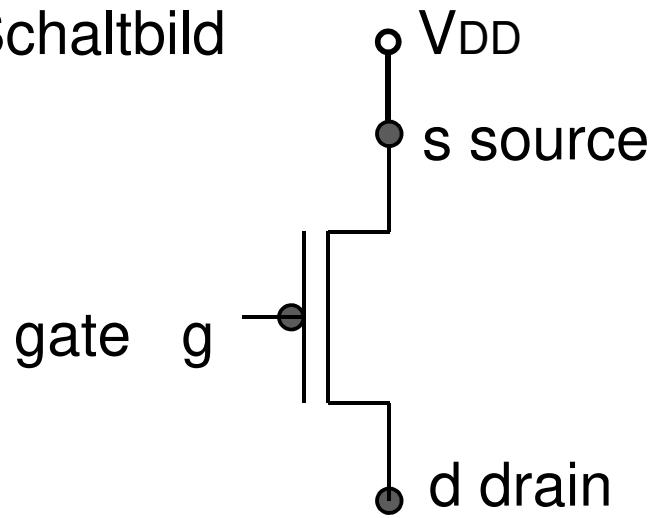


$U(g) \sim 0$

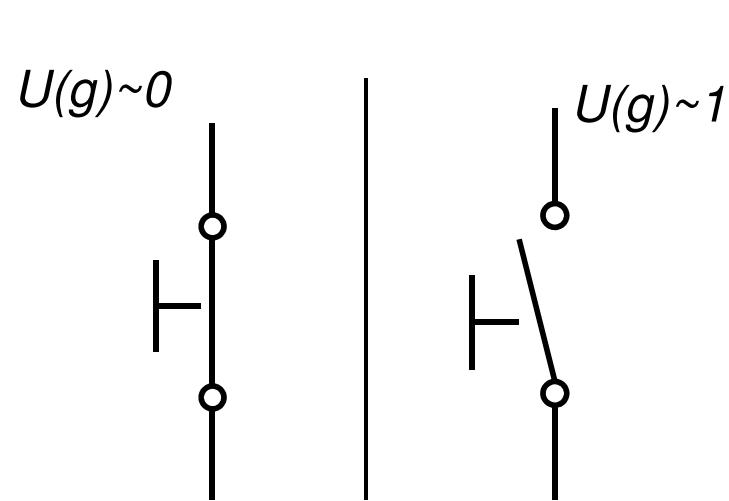


p-Kanal Transistor

Schaltbild



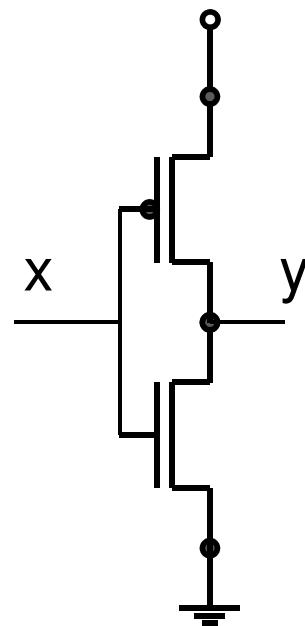
Verhalten



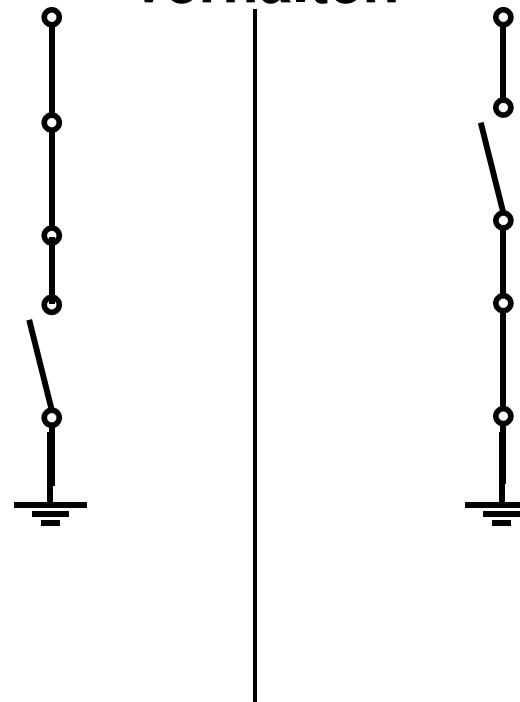
n-Kanal und p-Kanal Transistoren verhalten sich als Schalter **komplementär** zueinander.

Der CMOS Inverter

Wir können nun auf Basis solcher Schalter leicht Funktionen auf \mathcal{B} implementieren. Die wohl einfachste nichttriviale Funktion ist die Negation. Sie wird durch folgende Schaltung, den **CMOS Inverter**, (complementary) realisiert:



Verhalten



CMOS Inverter ff

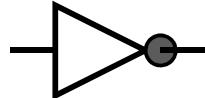
Diese einfache Schaltung berechnet also die Funktion:

$$i: \mathcal{B} \rightarrow \mathcal{B}$$

mit $i(0) = 1$ und $i(1) = 0$

D.h. die **Negation**, fasst man 0 (1) als falsch (wahr) auf.

Wir benutzen fortan als Symbol für diese Schaltung das Schaltzeichen:



Eine Sache ist noch sehr vage formuliert: Wir sagten, der Ausgang y wird verbunden mit V_{SS} (V_{DD}) und wird somit zu 0 (1). Wann aber erreicht der Ausgang tatsächlich einen Pegel, der als 0 (1) interpretierbar ist?

Schaltzeiten eines Inverters

Wir nehmen an, dass sich die Schalter zu einem bestimmten Zeitpunkt wegen einer Änderung des Eingangs sehr schnell umlegen. Eine abrupte Veränderung bewirkt dies nicht, weil die Bausteine nicht ideal sind. In heutiger CMOS Technologie dominiert:

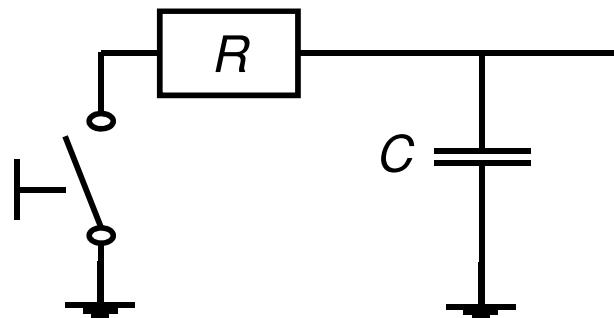
- Der **Widerstand R** des Transistors und der Leitungen

$$\text{Ohmsches Gesetz } U(t) = R \cdot I(t)$$

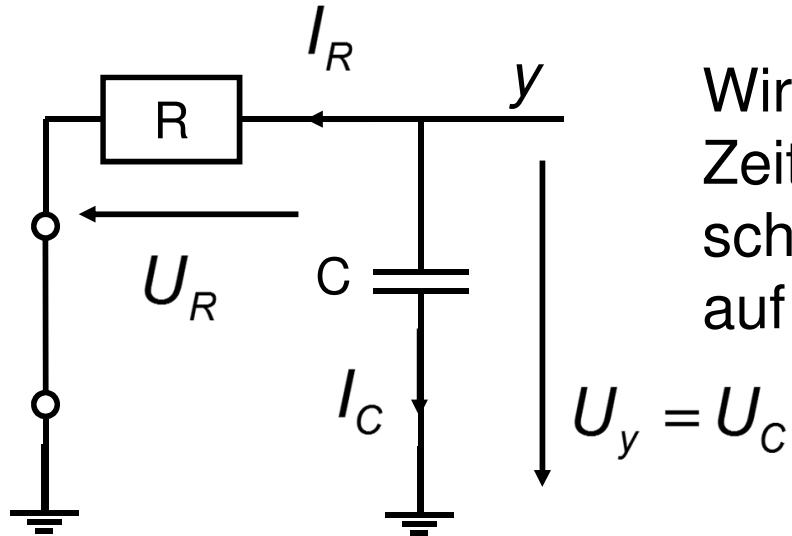
- Die **Kapazität C** der Gates der Folgetransistoren und der Leitungen

$$\text{Kondensatorgleichung } Q(t) = C \cdot U(t)$$

Modell:



Analyse



Wir nehmen an, dass sich zum Zeitpunkt $t=0$ der Schalter schließt, und der Ausgang y auf V_{DD} aufgeladen war.

$$I_R(t) + I_C(t) = 0 \quad (\text{Knotenregel})$$

$$-U_R(t) + U_y(t) = 0 \quad (\text{Maschenregel})$$

$$U_R(t) = R \cdot I_R(t) \quad (\text{Ohm'sches Gesetz})$$

$$I_C(t) = \frac{dQ_C(t)}{dt} = C \cdot \frac{dU_y(t)}{dt} \quad (\text{Kondensatorgleichung } Q(t) = C \cdot U(t))$$

Analyse ff

Nun gilt für den zeitlichen Verlauf der Ausgangsspannung:

$$\begin{aligned} U_y(t) &= U_R(t) & U_R(t) &= R \cdot I_R(t) \\ &= R \cdot I_R(t) & I_R(t) + I_C(t) &= 0 \\ &= -R \cdot I_C(t) & \\ &= -R \cdot C \cdot \frac{dU_y(t)}{dt} & I_C(t) &= C \cdot \frac{dU_y(t)}{dt} \end{aligned}$$

Lösung dieser Gleichung ist ein Verlauf der Form:

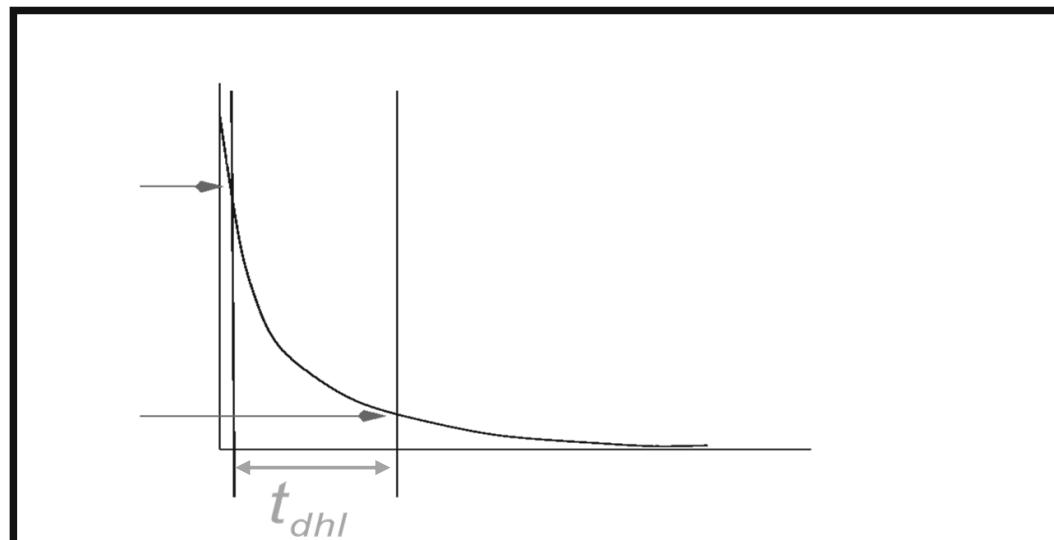
$$U_y(t) = A \cdot e^{-\frac{t}{R \cdot C}}$$

A bestimmt sich aus der Anfangsbedingung

$$V_{DD} = U_y(0) = A \cdot e^{\frac{0}{R \cdot C}} = A$$

Analyse ff

Wir nennen die Zeit, die verstreicht, bis ein fallender (steigender) Ausgang wieder als Wert über 0 (1) interpretierbar ist, die **Abfallzeit (Anstiegszeit)** des Ausgangs. Sie wird häufig definiert als die Zeit, die verstreicht, um den Ausgang von 0.9 V_{DD} auf 0.1 V_{DD} (0.1 V_{DD} auf 0.9 V_{DD}) zu bringen:



Analyse ff

Sei t_H der Zeitpunkt mit $U_y(t_H) = 0.9 V_{DD}$

$$V_{DD} \cdot e^{-\frac{t_H}{R \cdot C}} = 0.9 V_{DD} \Leftrightarrow t_H = -\ln 0.9 \cdot RC$$

Sei entsprechend t_L der Zeitpunkt mit $U_y(t_L) = 0.1 V_{DD}$

$$V_{DD} \cdot e^{-\frac{t_L}{R \cdot C}} = 0.1 V_{DD} \Leftrightarrow t_L = -\ln 0.1 \cdot RC$$

Damit ist

$$\begin{aligned} t_{dhl} &= t_L - t_H \\ &= (\ln 0.9 - \ln 0.1) \cdot RC \\ &= \ln 9 \cdot RC \approx \boxed{2RC} \end{aligned}$$

Analyse ff

R, C sind Materialgrößen. An ihnen muss man drehen, wenn man das Zeitverhalten eines Bausteins beeinflussen will.

Man erhält durch eine analoge Rechnung auch die Anstiegszeit. Sind Anstiegs- und Abfallzeit wie in dieser stark vereinfachten Modellrechnung gleich, definiert man die **Verzögerungszeit** als:

$$t_d = \frac{t_{dlh}}{2} = \frac{t_{dhl}}{2} = RC$$

Anmerkung: In der Praxis (CMOS Technologie) gilt $t_{dlh} \approx 2 \cdot t_{dhl}$

1.3.3 WüHDL-Beschreibung eines Bausteins

Den Inverter haben wir nun als ersten Baustein kennengelernt. Wir wollen nun sehen, wie man sein Verhalten in WüHDL beschreiben kann:

Eine Bausteindefinition besteht aus

- genau einer **Entity**
 - ◎ Spezifikation der Schnittstelle
- mindestens einer **Architecture**
 - ◎ Spezifikation des Verhaltens, durch
 - Programmcode (Prozesse), d.h. man beschreibt das Verhalten mit dem „Repertoire“ einer gewöhnlichen Programmiersprache (Ada ähnlich in WüHDL), oder
 - Strukturbeschreibung, d.h. man definiert das Verhalten durch Instanziierung und Verbindung anderer Komponenten (Schaltkreise).

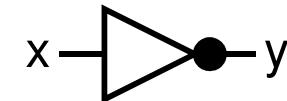
Entities -- die Schnittstellen von Komponenten

Syntax:

```
<entity_declaration>
 ::= ENTITY <entity_identifier> IS
     [<generic_declaration>] [<port_declaration>]
   END [ENTITY] [<entity_identifier>];
<port_declaration>
 ::= PORT ( <port_list> : [<mode>] <type_id>
           {;<port_list> : [<mode>] <type_id>}* );
<generic_declarations> erläutern wir später.
```

Beispiel: Deklaration eines Inverterbausteins

```
ENTITY inverter IS
  PORT (x: IN bit; y: OUT bit);
END inverter;
```



Entities: Port-Arten

Es gibt in VHDL 5 Port Modes von denen wir in WüHDL nur 3 benutzen:

- **IN** -- der Port ist ein Eingangsport, d.h. die Komponente selbst verändert ihn nicht.
- **OUT** -- der Port ist ein Ausgangsport, d.h. die Komponente selbst verändert sein Signal, verhält sich aber unabhängig von den Werten seines Signals.
- **INOUT** -- sein Signal darf von der Komponente verändert werden, kann aber auch von außen verändert werden und Einfluss auf das Verhalten der Komponente nehmen.
- **BUFFER** -- Ausgangsport, nur die Komponente darf sein Signal verändern, sie darf sich aber auch abhängig von Werten seines Signals verhalten.
- **LINKAGE** -- unbekannt, ob Eingang oder Ausgang

Architekturen -- die Definition der Komponenten

Syntax: -- vorläufig

<architecture_declaration>

::= **ARCHITECTURE** <architecture_identifier>

OF <entity_identifier> **IS**

 [<local_declarations>]

BEGIN [<statements>]

END [ARCHITECTURE] [<architecture_identifier>];

<local_declarations>

::={ **USE** <...> | **TYPE** <...> | **CONSTANT** <...>

 | **SIGNAL** <...> | **VARIABLE** <...> | **COMPONENT** <...>

 | **FUNCTION** <...> | **PROCEDURE** <...> }*

<statements> ::= { <process_statement> | <conc_sig_assign>

 | <comp_instantiation> }*

Architekturen -- die Definition der Komponenten

Definitionsmöglichkeiten

Man hat schon aus der Syntax ersichtlich mehrere Möglichkeiten, das Verhalten von Komponenten zu definieren. In WüHDL sehen wir folgende Optionen vor:

- Beschreibung durch Prozesse
- Beschreibung durch nebenläufige Signalzuweisungen (concurrent signal assignments)
- Beschreibung durch Instanziierung und Verbindung von Komponenten.

Die erste Variante kann man als Verhaltensdefinition auffassen, während die letztere eine Strukturdefinition ist (Aufbau über Instanzen vorhandener Komponenten). Die zweite Variante kann man sowohl als Verhaltens als auch als Strukturdefinition auffassen (vgl. spätere Kapitel). Wir klammern sie vorläufig aus.

Architekturen -- die Definition durch Prozesse

Prozesse sind sequentiell ablaufende Programme, ergänzt durch einige spezielle Anweisungen, zu denen wir noch kommen. Innerhalb eines Prozesses sind sichtbar:

- alle Ports vom Modus IN, INOUT als Werte zum aktuellen Zeitpunkt
- alle lokal in der Architecture definierten Signale, Variablen, Prozeduren, Funktionen, ...
- alle Ports vom Modus OUT, INOUT und alle lokal definierten Signale als Ziel einer Signalzuweisung.

Wir betrachten zunächst ausschließlich Prozesse mit einer **Sensitivitätsliste**. Eine Sensitivitätsliste enthält nur Ports der entity vom Modus IN, INOUT und ggf. lokale Signale. Ändert sich ein Signal der Sensitivitätsliste, dann wird das im <process_statement> aufgeführte Programmstück bis zur Terminierung ausgeführt.

-- Vorsicht: Terminiert das Programmstück nicht, hängt sich die Simulation bei der Ausführung auf!

Architekturen -- Beispiel zu Prozessen

Folgendes triviale Programmstück beschreibt das Verhalten unseres CMOS Inverters durch ein process statement:

```
ARCHITECTURE behavior1 OF inverter IS
  CONSTANT tdhl: TIME := 25 ps;
  CONSTANT tdlh: TIME := 35 ps;
    -- Festlegung der Verzögerungszeiten durch lokale
    -- Konstanten. Wird in der Praxis flexibler gelöst.
BEGIN
  PROCESS (x) -- sensitiv auf IN, inout Ports der entity
    BEGIN -- Signalzuweisungen
      IF x = '0'
        THEN y <= not x AFTER tdlh;
        ELSE y <= not x AFTER tdhl;
      END IF;
    END PROCESS;
END behavior1;
```

Architekturen -- Prozesse

Prozesse bestehen im Allgemeinen aus Anweisungsfolgen, die das ganze Repertoire einer klassischen, imperativen Programmiersprache umfassen (Schleifen, Prozedur- und Funktionsaufrufe, ...). Die einzigen Anweisungen, die einen Prozess zum Terminieren bringen, sind sogenannte WAIT Anweisungen.

Vorsicht: Der Abschluss END [PROCESS] der Deklaration **terminiert nicht** den Prozess, sondern bezeichnet nur das Ende der Deklaration.

Der Prozess startet bei Beendung wieder von vorne durch!

Die Angabe einer Sensitivitätsliste ist äquivalent zu einer WAIT Anweisung als letzter Anweisung des Prozesses, d.h. im Grunde ist eine Prozess Anweisung

PROCESS (x₁,...,x_k)

BEGIN

... -- Anweisungen

END PROCESS;

zu interpretieren

als

LOOP

... -- Anweisungen

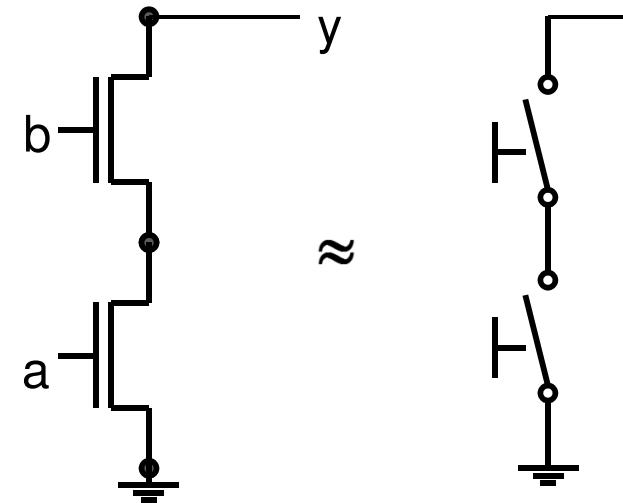
WAIT ON x₁,...,x_k

END LOOP;

1.3.4 Einfache Verknüpfungen

Beim Inverter berechneten wir die Inversion durch Schalten einer leitenden Verbindung zwischen Ausgang und VDD bzw. Vss. Dies muss stets komplementär, d.h. unter gegenseitigem Ausschluss geschehen! (sonst Kurzschluss)

Betrachte nun folgende Schaltung:
(n-Kanal Serienschaltung)

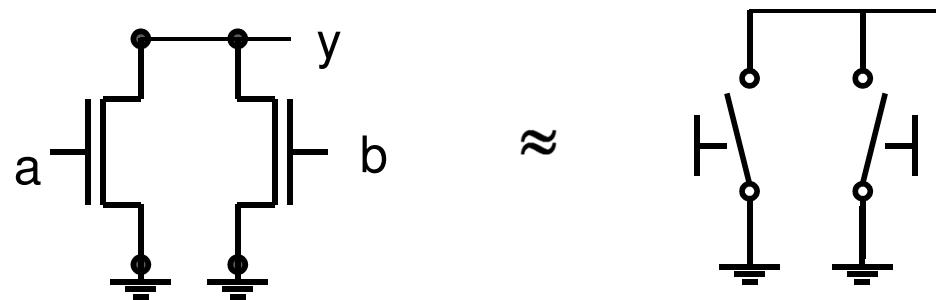


Offensichtlich entsteht eine Verbindung mit Vss dann und nur dann, wenn beide Schalter leitend sind, d.h. wenn

$$a=1 \text{ und } b=1$$

Einfache Verknüpfungen -- ff

Betrachten wir nun eine Parallelschaltung zweier n-Kanal Transistoren gegen V_{SS}:



Offensichtlich entsteht eine Verbindung mit V_{SS} dann und nur dann, wenn mindestens einer der beiden Schalter leitend ist, d.h. wenn **a=1 oder b=1**.

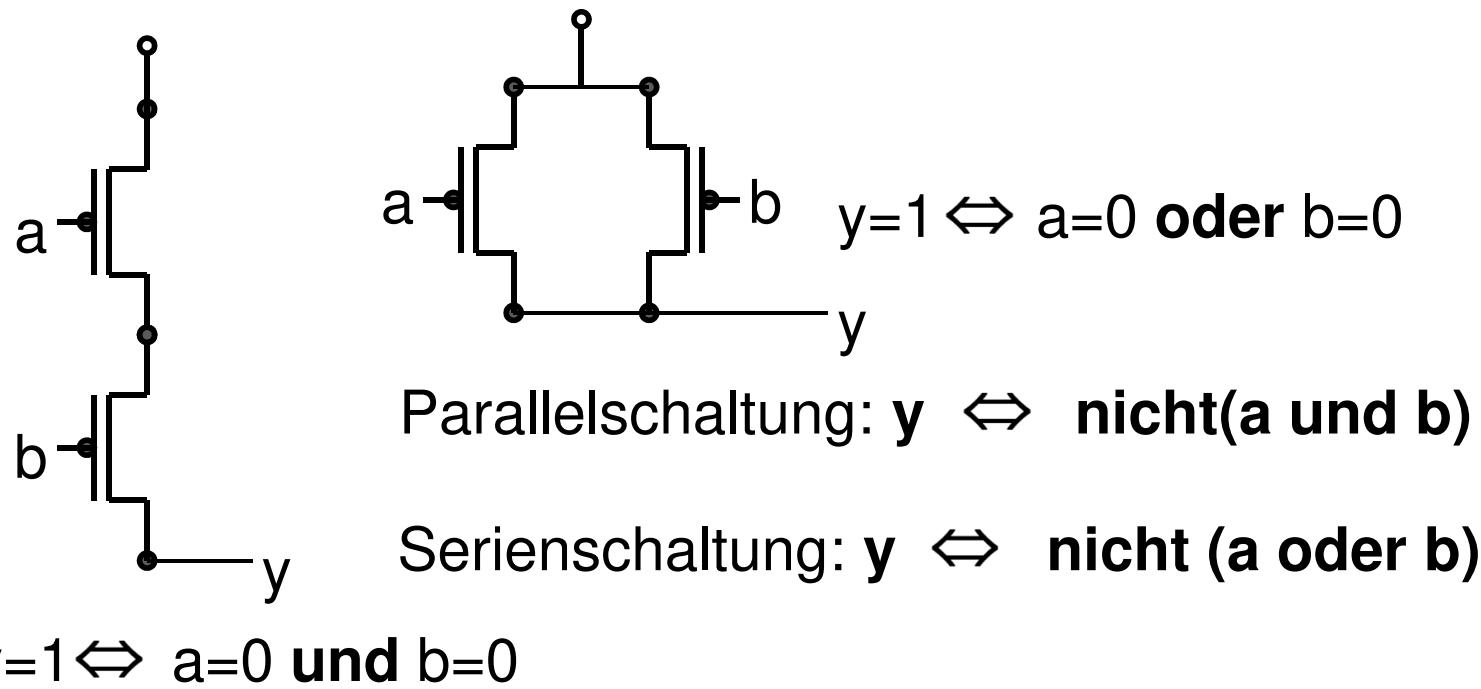
Interpretieren wir nun x=1 einfach als x bzw. „x ist wahr“ und x=0 einfach als nicht x bzw. „x ist falsch“, dann gilt:

NAND und NOR Verknüpfung ff

Serienschaltung gegen V_{SS}: **nicht y \Leftrightarrow a und b**

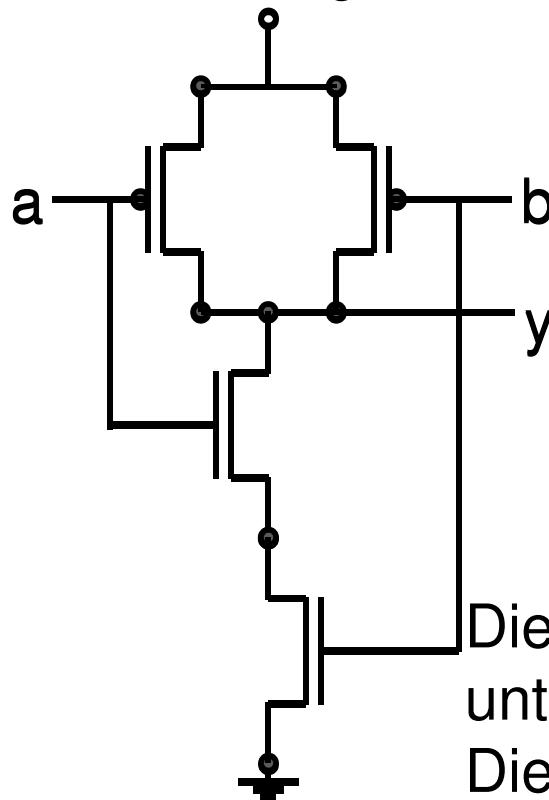
Parallelschaltung gegen V_{SS} : **nicht y \Leftrightarrow a oder b**

Analog erhalten wir für p-Kanal Transistor Schaltungen
gegen V_{DD}:



Einfache Verknüpfungen ff

Demnach liefert folgende Schaltung:



$$y \Leftrightarrow \text{nicht}(a \text{ und } b)$$

(p-Kanal Parallelschaltung)

$$\text{nicht } y \Leftrightarrow a \text{ und } b$$

(n-Kanal Serienschaltung)

Die leitenden Verbindungen bestehen unter **gegenseitigem Ausschluss**.
Die Verschaltungen im n-Kanal, p-Kanal Teil sind **dual** zueinander
(Serien \Leftrightarrow Parallel).

NAND und NOR Verknüpfung

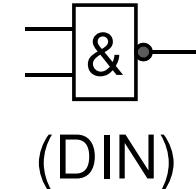
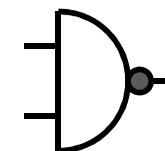
Die Schaltung berechnet also die Funktion **nand**

| nand | b | 0 | 1 |
|------|---|---|---|
| a | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | |

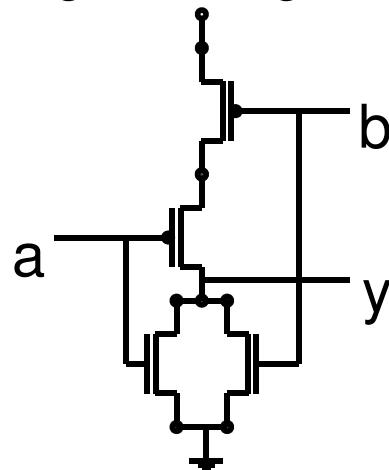
$nand: B \times B \rightarrow B$

mit $nand(a, b) := \overline{a \cdot b}$

Schaltsymbol



Analog überlegt man sich, dass die Schaltung

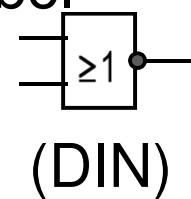
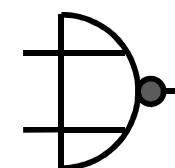


die Funktion **nor** berechnet

mit $nor(a, b) := \overline{a \vee b}$

| nor | b | 0 | 1 |
|-----|---|---|---|
| a | 0 | 1 | 0 |
| 0 | 1 | 0 | |
| 1 | 0 | 0 | |

Schaltsymbol



Beschreibung in WüHDL

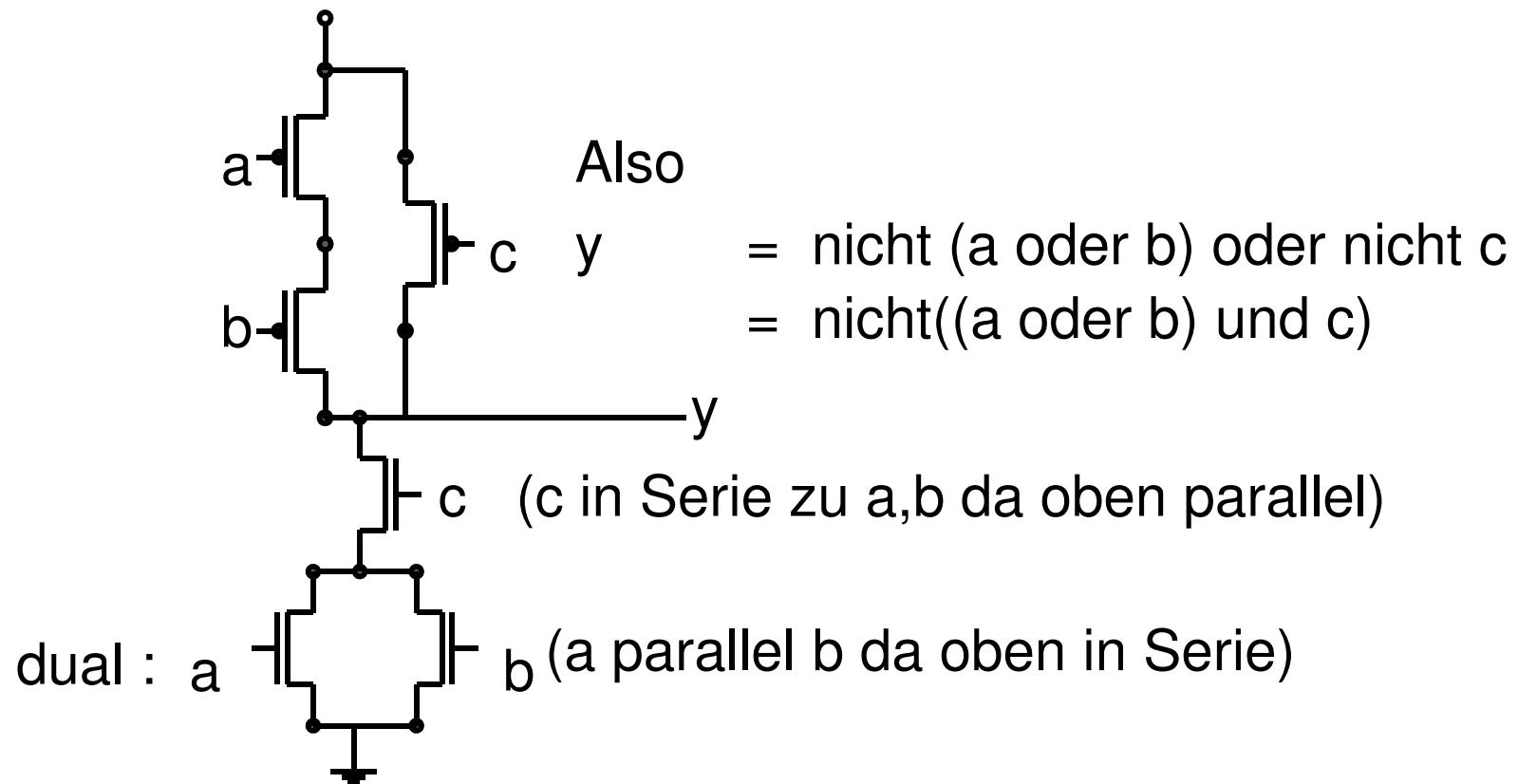
Wir können diesen Baustein ebenso leicht in WüHDL beschreiben:

```
ENTITY nand2 IS
    PORT (x,y: IN bit; z: OUT bit);
END nand2;

ARCHITECTURE behavior1 OF nand2 IS
    CONSTANT tdhl: TIME := 35 ps;
    CONSTANT tdlh: TIME := 50 ps;
    -- vgl. Inverter
BEGIN
    PROCESS (x,y) -- sensitiv auf IN, INOUT Ports der entity
    BEGIN -- Signalzuweisungen
        IF x = '0' OR y = '0'
            THEN z <= '1' AFTER tdlh;
            ELSE z <= '0' AFTER tdhl;
        END IF;
    END PROCESS;
END behavior1;
```

CMOS Komplexgatter

Man kann dieses Spiel beliebig (technisch hat es Grenzen) weitertreiben, indem man zueinander duale Serien/Parallel Netzwerke von Transistoren bildet:



CMOS Komplexgatter ff

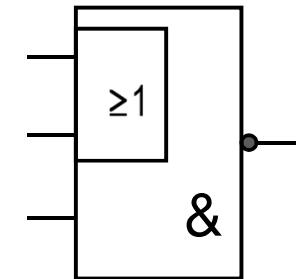
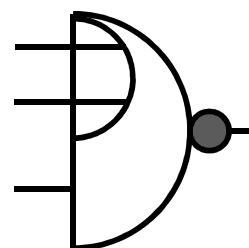
Die Schaltung berechnet also die Funktion

$$f(a, b, c) := \overline{(a \vee b) \cdot c}$$

Man nennt eine solche Schaltung, da sie nicht nur einfache Verknüpfungen berechnet, ein **Komplexgatter**.

(ein „oai21“)

Schalsymbol



(DIN)

1.3.5 Speicherelemente

Was ist ein Speicher für ein Zeichen aus \mathcal{B} ?

Allgemein ist ein Speicher für k Zeichen aus einer Menge O modelliert durch

$$\mathcal{F}([0:k-1], O)$$

-- die Menge der partiellen Funktionen von $[0:k-1]$ nach O

Für ein einziges Zeichen aus \mathcal{B} ist ein Speicherbaustein damit modellierbar durch

$$\mathcal{F}(\{0\}, \mathcal{B})$$

Es gibt also genau 3 Speicherzustände:

$$\Omega, s_0, \text{ mit } s_0(0) = 0, \text{ und } s_1, \text{ mit } s_1(0) = 1$$

Klammern wir den undefinierten Zustand Ω aus, gibt es also zwei definierte Zustände, das Speichern einer 0 und einer 1.

Speicherelemente

Wir brauchen also einen Baustein, der zwei Zustände annehmen kann. Er sollte auch kontrolliert zwischen diesen Zuständen hin und her schalten können. Der Zeitpunkt einer Zustandsänderung soll kontrollierbar sein, etwa durch das Ticken einer Uhr.

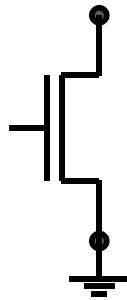
Wie realisiert man solche Bausteine?

Wir werden diese Frage auf 2 Wegen beantworten:

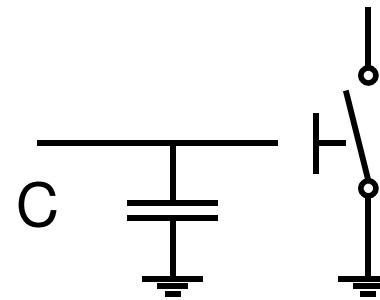
- Realisierung in CMOS Technologie
- Realisierung durch Logikverknüpfungen
(ohne konkretes Technologiewissen)

CMOS Speicherelemente

Wir können in CMOS Schaltungen die Kapazität der Transistor-gates und Leitungen zum Speichern von Informationen ausnutzen, da in das Gate nur ein winziger Leckstrom abfließt (RC Glied mit riesigem R).



Modell:



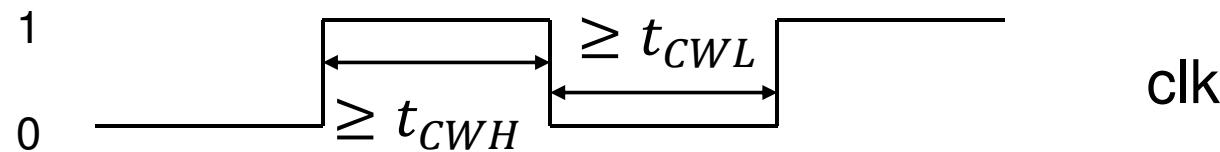
Je nach Ladung der Kapazität C haben wir die Zustände:

- C aufgeladen: dann ist $U(g) \sim 1$, Schalter ist geschlossen
- C entladen: dann ist $U(g) \sim 0$, Schalter ist offen.

Entsprechendes gilt dual für p-Kanal Transistoren.

CMOS Speicherelemente

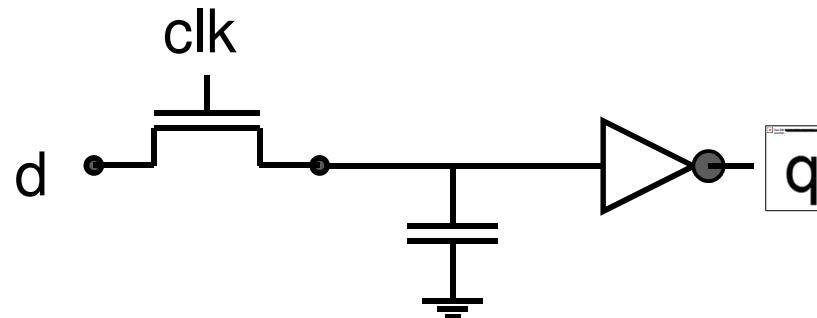
Wir nehmen nun noch einen weiteren Anschluss **clk** hinzu, mit dessen Hilfe wir die Änderung von Zuständen kontrollieren wollen. Liegt der Anschluss für eine hinreichend lange Zeit t_{CWH} auf 1, wollen wir dies als einen Übernahmetick interpretieren. Um zwei Ticks voneinander unterscheiden zu können müssen sie durch eine 0 Phase voneinander getrennt sein, die mindestens t_{CWL} dauert.



Wir nennen ein Diagramm, das den zeitlichen Verlauf von Spannungen mit Bedingungen daran wiedergibt, auch ein **Zeitdiagramm**.

CMOS Speicherelemente

Mit Hilfe des neuen Anschlusses braucht man nun nur noch das Laden der Eingangskapazität eines Inverters zu kontrollieren und kommt so zu folgender einfachen Lösung:



Bei jedem Tick ($clk=1$) wird d mit dem Eingang des Inverters verbunden und lädt die Kapazität auf. ($\bar{q} = \bar{d}$) Wird $clk = 0$ ist d und die Kapazität entkoppelt, der eingeschriebene Wert bleibt unabhängig von d erhalten, sozusagen verriegelt. Wir nennen diesen Baustein daher ein **dynamisches Daten-Latch** (D-Latch).

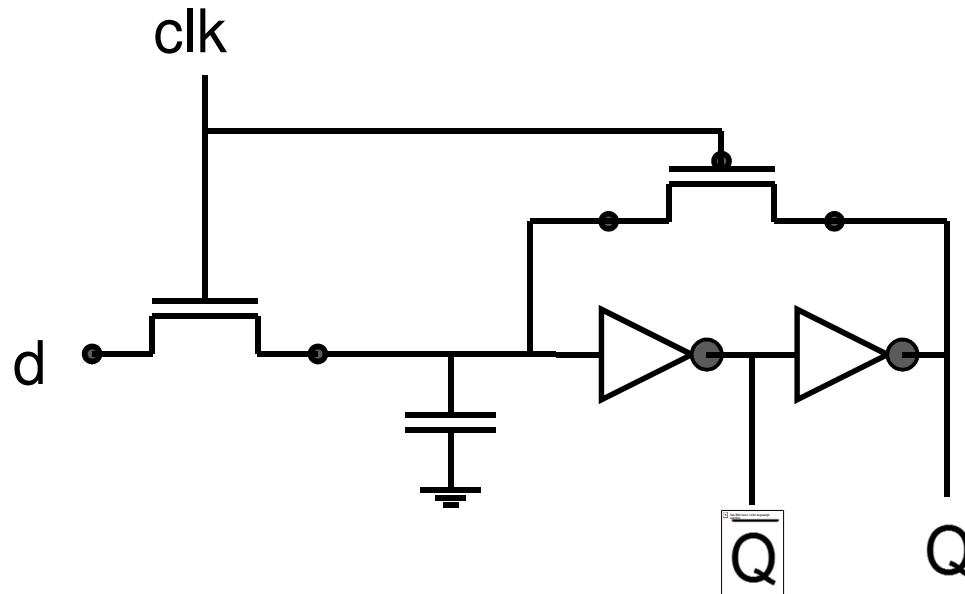
CMOS Speicherelemente

Problem: Durch die Leckströme fließt die eingespeicherte Ladung am Eingang des Inverters langsam ab. Dauert die 0 Phase von clk zu lange, kann dies zu einer ungewollten Zustandsänderung führen. Daher nennt man wegen der zusätzlichen Zeitbedingung das Latch **dynamisch**.

Auswege:

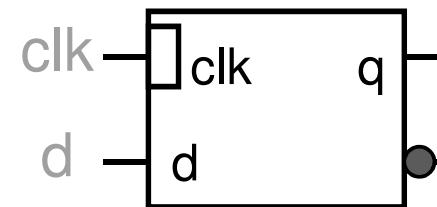
- Stelle sicher, dass die Kapazität in regelmäßigen Abständen aufgefrischt wird (refresh Bedingung).
- Ergänze die Schaltung um eine Halteschaltung
statisches D-Latch

Statisches Latch (CMOS)



Die Halteschaltung hält in der 0 Phase den Wert q über einen leitenden p-Kanal Transistor beliebig lange. Wir nennen diese Schaltung einfach ein **statisches D-Latch** (D für Data) oder kurz D-Latch.

Schaltbild:



Beschreibung in WüHDL

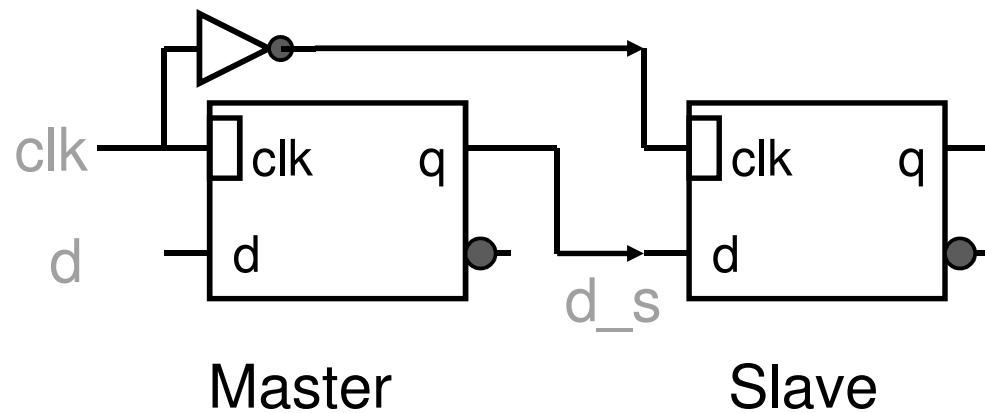
Wir können diese Bausteine ebenso leicht in WüHDL beschreiben:

```
ENTITY sdlatch IS
    PORT (clk,d: IN bit; q: OUT bit;
          qz: OUT bit := '1' ); -- explizite Initialisierung auf NOT q erforderlich!
END sdlatch;

ARCHITECTURE behavior1 OF sdlatch IS
    CONSTANT tdhl: TIME := 50 ps;
    CONSTANT tdlh: TIME := 60 ps; -- Anmerkung vgl. Inverter
BEGIN
    PROCESS (clk,d) -- sensitiv auf IN, INOUT Ports der entity
    BEGIN
        IF clk = '1'
            THEN IF d = '0'
                THEN qz <= '1' AFTER tdlh; q <= '0' AFTER tdlh+tdhl;
                ELSE qz <= '0' AFTER tdhl; q <= '1' AFTER tdhl+tdlh;
                END IF;
            END IF;
        END PROCESS;
    END behavior1;
```

Master/Slave Latch

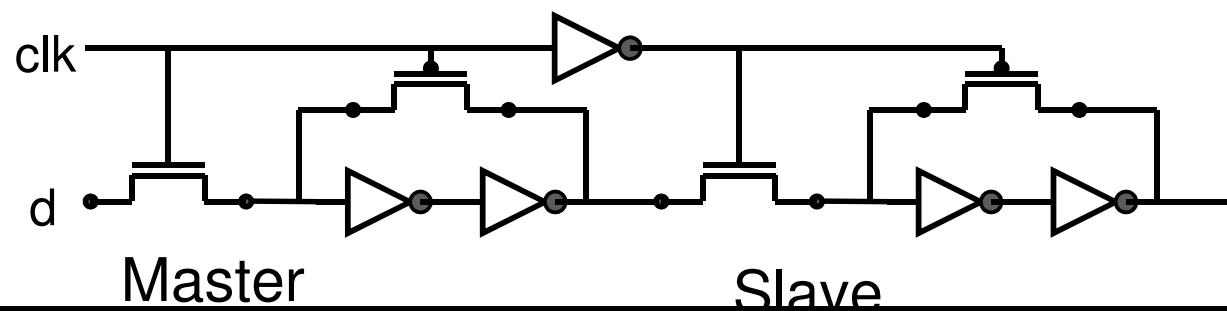
Häufig macht die Gefahr, dass sich d in der 1 Phase des Taktes clk ändern könnte, (dann ist ja d **transparent** mit q verbunden) den Entwurf schwierig. Man möchte daher die Zeit, in der sich d nicht ändern darf, so kurz wie möglich halten. Dies erreicht man, indem man den Übernahmezeitpunkt eng um die 1/0 Flanke des Taktes clk , wie in folgender Schaltung, legt:



Master/Slave Latch ff

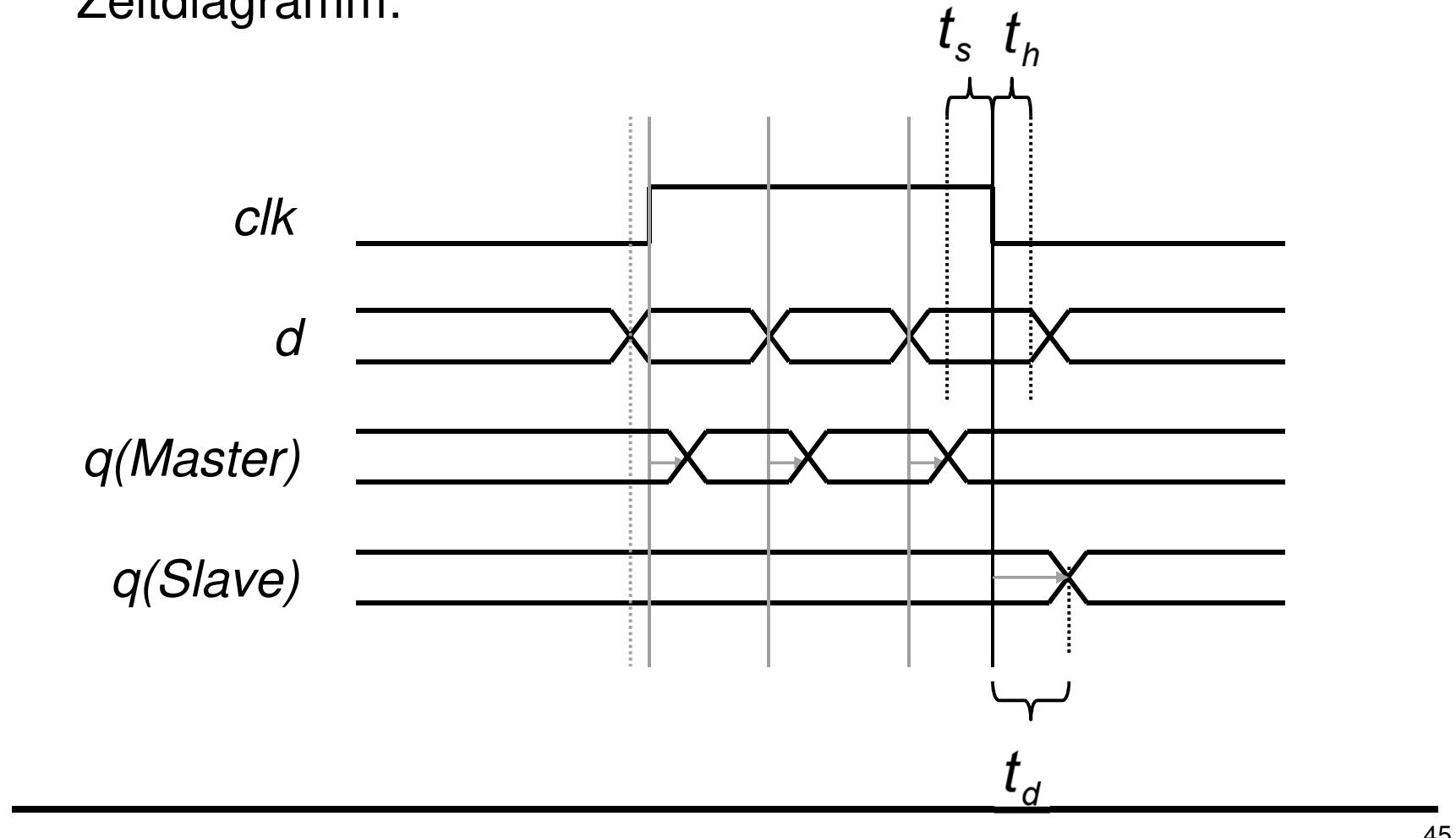
In der 1 Phase des Taktes schaltet das Master Latch d transparent nach q_m , das Slave Latch ist aber verriegelt. In der 0 Phase verriegelt der Master und der Slave schaltet den zu Beginn der 0 Phase an $q_m = d_m$ liegenden Wert transparent nach q .

Man muss also nur noch sicherstellen, dass d und q_m in einem kurzen Zeitraum um die 1/0 Flanke stabil sind, um einen korrekten Zustandsübergang mit der Flanke sicherzustellen.



Master/Slave Latch: Timing

Verdeutlichung des Übernahmeverhaltens am Zeitdiagramm:



Master/Slave Latch: Timing ff

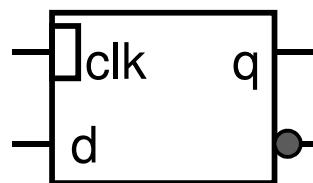
Meist gibt man 3 charakteristische Schaltzeiten an:

- die **delay** Zeit t_d , bis der Ausgang übernimmt (ggf. t_{dhl} , t_{dlh})
- die **setup** Zeit ts , die d vor der Flanke stabil sein muss
- die **hold** Zeit th , die d nach der Flanke stabil sein muss

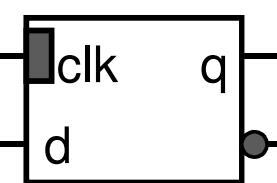
Wir nennen ein solches Latch auch

flankengesteuertes D-Latch oder D-FlipFlop

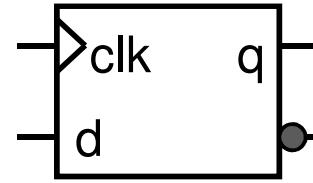
Man kann Latches zustands- oder flankengesteuert nach verschiedenen Taktzuständen oder Flanken konstruieren.
Wir zeigen dies an durch entsprechende Schaltsymbole:



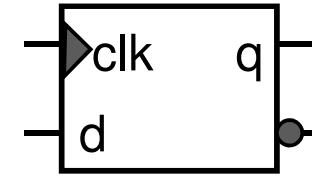
1-Phasen-



0-Phasen-



0/1-Flanken-



1/0-Flankengesteuert

WüHDL-Beschreibung eines D-FlipFlops

Wir haben bisher die Bausteine stets definiert durch

- eine **Entity**: Spezifikation der Schnittstelle
- eine **Architecture**: Spezifikation des Verhaltens, durch Programmcode (Prozesse).

Das flankengesteuerte D-Latch haben wir aber durch Benutzung von Instanzen anderer Bausteine (statische D-Latches) definiert. Also sollte man die Definition des Bausteins in WüHDL ebenso durch die schon erwähnte zweite Möglichkeit der

- Strukturbeschreibung, d.h. man definiert das Verhalten durch Instanziierung und Verbindung anderer Komponenten (Schaltkreise)

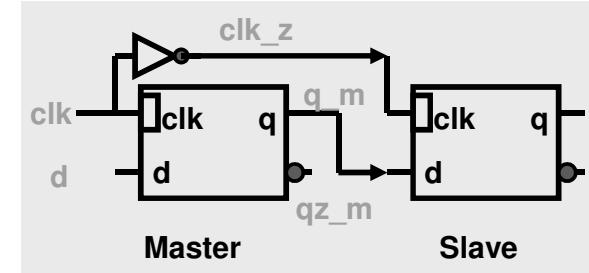
vornehmen.

WüHDL-Beschreibung eines D-FlipFlops

```
ENTITY dlatch IS
    PORT (clk,d: IN bit; q,qz: OUT bit);
END dlatch;
```

```
ARCHITECTURE structure OF dlatch IS
```

```
    COMPONENT sdlatch PORT (clk,d: IN bit; q,qz: OUT bit);
        END COMPONENT;
    COMPONENT inverter PORT (x: IN bit;y: OUT bit);
        END COMPONENT;
    SIGNAL q_m,qz_m,clk_z: bit;
BEGIN
-- Definition der Schaltung durch Instanziierungen
    INV: inverter PORT MAP (x => clk, y => clk_z);
    Master: sdlatch PORT MAP (clk => clk, d => d,q => q_m, qz => qz_m);
    Slave: sdlatch PORT MAP (clk => clk_z, d => q_m, q => q, qz => qz);
END structure;
```



Problem: Wir haben keinerlei Überwachung der Einhaltung von Setup und Hold Zeiten!

Überwachung von Bedingungen

WüHDL sieht dazu sogenannte **Assertions** vor. Diese bestehen aus einer Bedingung, die zu überwachen ist, einer Fehlermeldung die bei Verletzung auszugeben ist, und einer Schwere des Fehlers.

Wir fügen einfach unserer Definition noch einen Überwachungsprozess hinzu:

```
ARCHITECTURE structure OF dlatch IS
    ... CONSTANT setup:TIME := 40ps; CONSTANT hold: TIME := 20ps;
BEGIN ...
    Setup_hold_check: PROCESS(clk,d)
        BEGIN IF clk'event AND clk = '0'
            THEN -- aktive Flanke der Clock, Setup Check
                ASSERT d'stable(setup) REPORT "Setup-Violation "
                    SEVERITY error;
            ELSE IF d'event AND clk = '0'
                THEN -- Datenänderung nach der aktiven Flanke
                    ASSERT clk'last_event > hold
                        REPORT "Hold-Violation " SEVERITY error;
                END IF;
            END IF;
        END PROCESS setup_hold_check;
    END structure;
```

Attribute von Signalen

WüHDL erlaubt die Definition von Attributen für alle möglichen Objekte. Zu einem Signal ***sig*** gibt es folgende vordefinierte Attribute, die sehr nützlich sind:

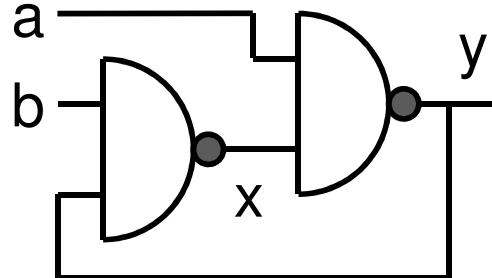
- ***sig*'stable(T)** ist vom Typ Boolean und hat genau dann den Wert TRUE, wenn der Wert von ***sig*** sich in den letzten T Zeiteinheiten nicht verändert hat.
 - ***sig*'event** ist vom Typ Boolean und liefert genau dann den Wert TRUE, wenn zu dem aktuellen Zeitpunkt eine Änderung auf ***sig*** erfolgt. Vorsicht: Da Änderungen auch mit Verzögerung 0 gemacht werden können, ist ***sig*'event** nicht zum gesamten Zeitpunkt TRUE sondern nur in der Iteration, in der das Ereignis bearbeitet wird.
 - ***sig*'last_event** ist vom Typ TIME und gibt die Zeit seit der letzten Änderung auf ***sig*** an.
 - ***sig*'last_value** ist vom gleichen Typ wie ***sig***. Es gibt den Wert von ***sig*** vor der letzten Änderung an.
- VHDL sieht noch mehr Attribute vor!

1.3.6 Bistabile Schaltungen

Wie konstruiert man Speicherglieder, wenn man lediglich über Logikgatter verfügt und keine spezielleren Eigenarten der Technologie auszunutzen weiß?

Idee: Konstruiere Bistabile Schaltungen aus Gattern.

Beispiel: Analysiere folgende Schaltung

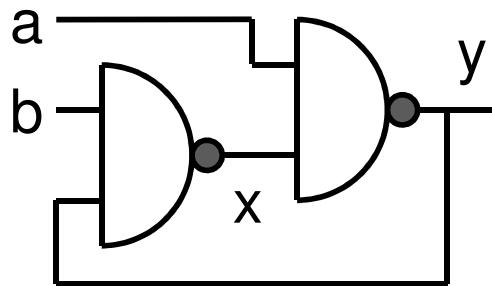


Es ist: $x = \overline{b \cdot y}$

und somit $y = \overline{a \cdot \overline{b \cdot y}} = \bar{a} \vee b \cdot y$

Bistabile Schaltungen

Falldiskussion:



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

$$a=b=1: y = \bar{a} \vee b y = \bar{1} \vee 1 y = 0 \vee y = y \quad \text{hält } y$$

$$a=0, b=1: y = \bar{0} \vee 1 y = 1 \vee y = 1 \quad \text{setzen}$$

$$a=1, b=0: y = \bar{1} \vee 0 y = 0 \vee 0 = 0 \quad \text{rücksetzen}$$

$$a=b=0: y = \bar{0} \vee 0 y = 1 \vee 0 = 1 \quad \text{unbenutzt}$$

Basis R/S FlipFlop

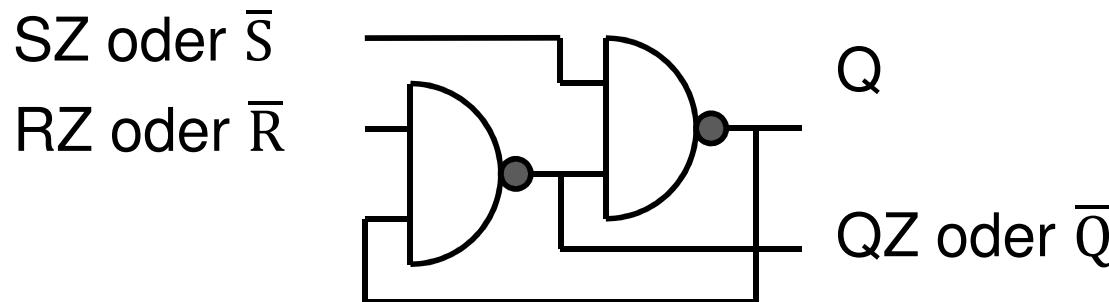
Wir nennen diese Schaltung auch **Basis R/S FlipFlop**, weil sie einen Haltezustand ($a=b=1$) hat, in dem sie jeden Wert y über die Rückkopplung halten kann, sowie eine Rücksetzung ($(a,b)=(1,0)$) und eine Setzkombination ($(a,b)=(0,1)$). Allerdings hat sie noch keine Taktkontrolle.

Die Zustandsänderung wird durch Legen eines der Eingänge auf den Wert 0 erreicht. Da der Wert 0 die Änderung bewirkt, nennt man solche Eingänge auch low active und überstreicht ihren Namen bzw. hängt ein Z an: RZ oder \overline{R}

Das Überstreichen ist als Kurzschreibweise für die Negation zu sehen. Die Aktivität, die mit dem Eingangsnamen verbunden wird, findet statt, wenn der Ausdruck wahr ist, d.h. Rücksetzen findet statt wenn $RZ=nicht(R)$ wahr, also $R=0$ ist.

Basis R/S FlipFlop ff

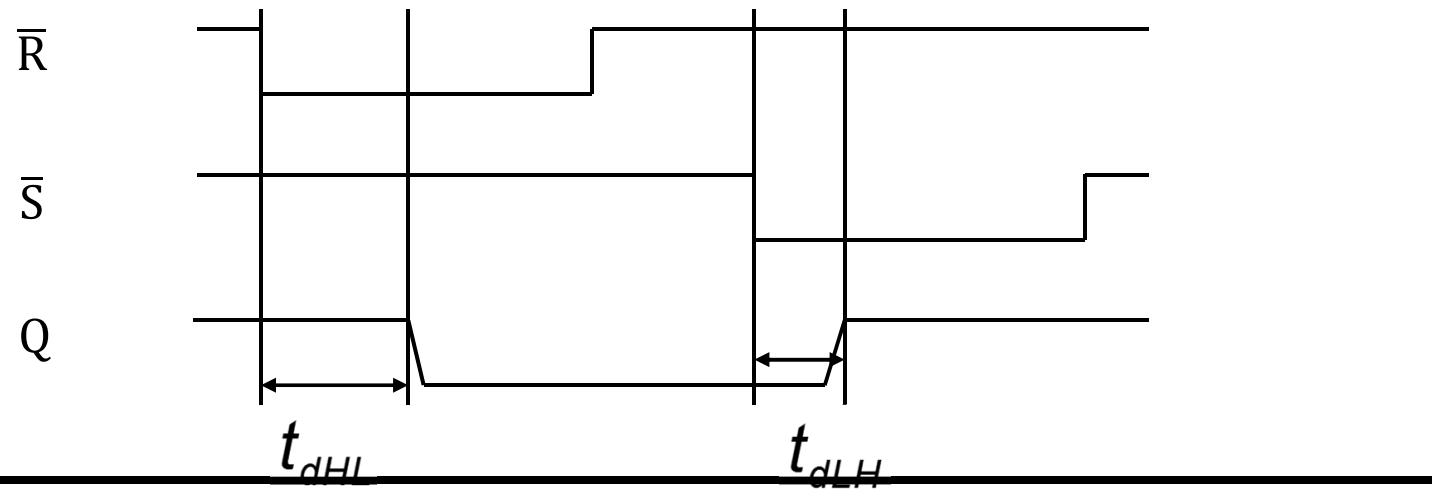
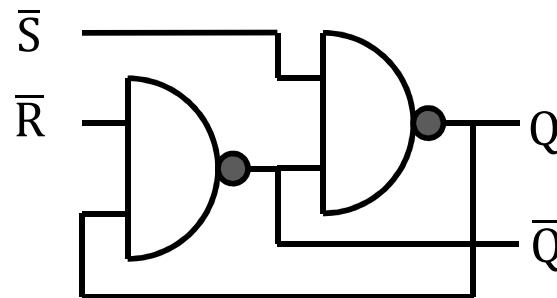
Damit ergibt sich folgende Benennung der Anschlüsse:



Folgende Tabelle gibt das Verhalten wieder: Sie gibt für die benutzten Eingangskombinationen $(RZ, SZ) \leftrightarrow (0, 0)$ den Folgezustand Q' von Q an:

Basis R/S FlipFlop ff

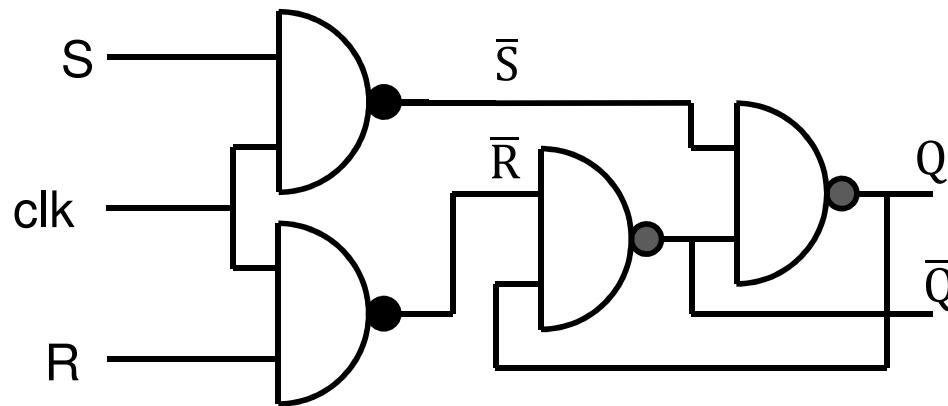
Die Reaktionszeiten ergeben sich durch die Laufzeiten der Gatter und sind für Setzen und Rücksetzen daher nicht gleich.



R/S FlipFlop ff

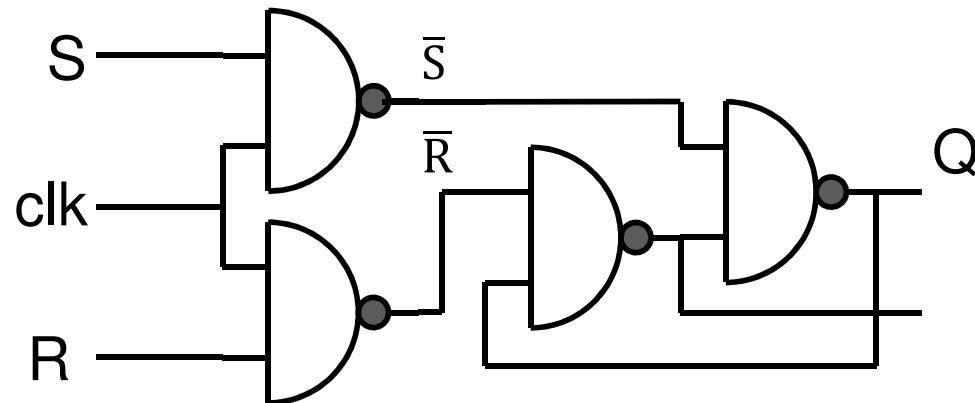
Aufgabe: Steuere das Basis R/S FlipFlop durch einen Taktzustand.

Lösung: Wir verknüpfen einfach den Takt mit den Kontrolleitungen RZ und SZ:

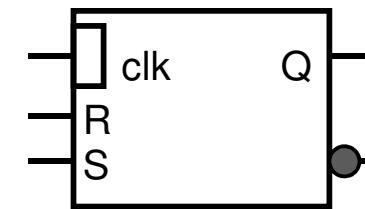


Analyse: Für $\text{clk}=0$ werden SZ, RZ auf nicht(0)=1 gezwungen.

R/S FlipFlop ff



Schaltsymbol:



Analyse:

Für $clk=0$ werden SZ, RZ auf *nicht(0)=1* gezwungen. \Rightarrow **Haltezustand**

Für $clk = 1$ werden SZ(RZ) auf $SZ = \overline{1 \cdot S} = \bar{S}$ ($RZ = \overline{1 \cdot R} = \bar{R}$) abgebildet.

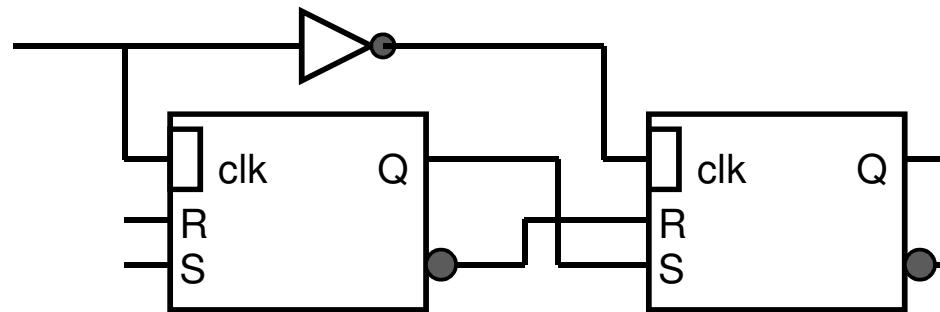
Man kann also nur in der 1 Phase setzen oder rücksetzen.

R/S FlipFlop ff

Offenbar kann man aus dem R/S FlipFlop leicht ein D-Latch bauen:



Und nach dem Master/Slave Prinzip ein flankengesteuertes R/S FlipFlop:

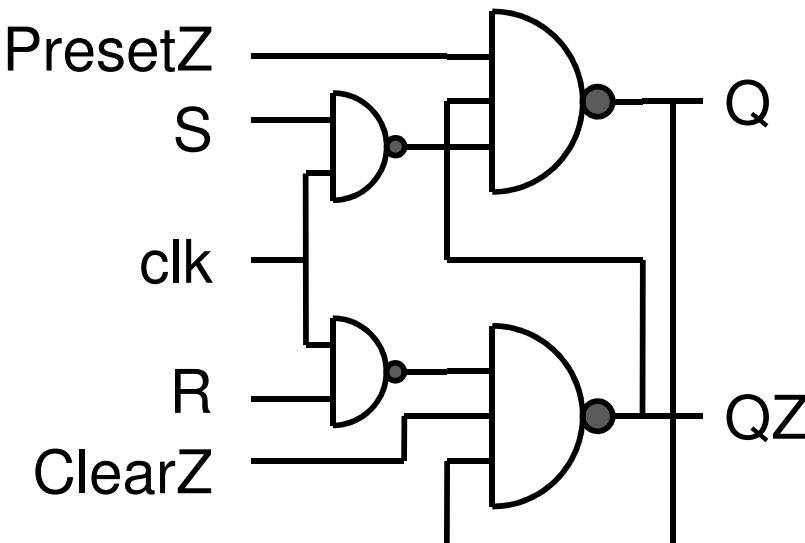


Es gibt einen wahren Wildwuchs von FlipFlop Typen, aus unterschiedlichen Zielsetzungen motiviert:

Synchrone und asynchrone Kontrolle

Ein Beispiel ist der Wunsch, den Ausgang Q synchron, d.h. durch den Takt, aber auch asynchron durch spezielle Kontrollleitungen manipulieren zu können. Ein typischer Vertreter ist das R/S FlipFlop mit asynchronem *PresetZ* und *ClearZ*:

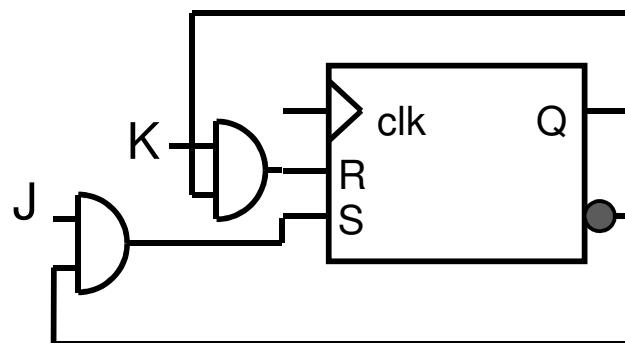
Man kann nun Q , QZ unabhängig von clk direkt setzen oder rücksetzen.



Ausbau redundanter Belegungen:

Die Belegung $(S,R) = (1,1)$ darf nicht benutzt werden. Sie ist für den Betrieb bedeutungslos. Solche Belegungen nennt man auch redundant. Gibt man dieser Kombination durch entsprechende Verschaltung eine weitere Bedeutung so hat man eine Erweiterung des R/S-FlipFlops.

Beispiel: das J/K FlipFlop

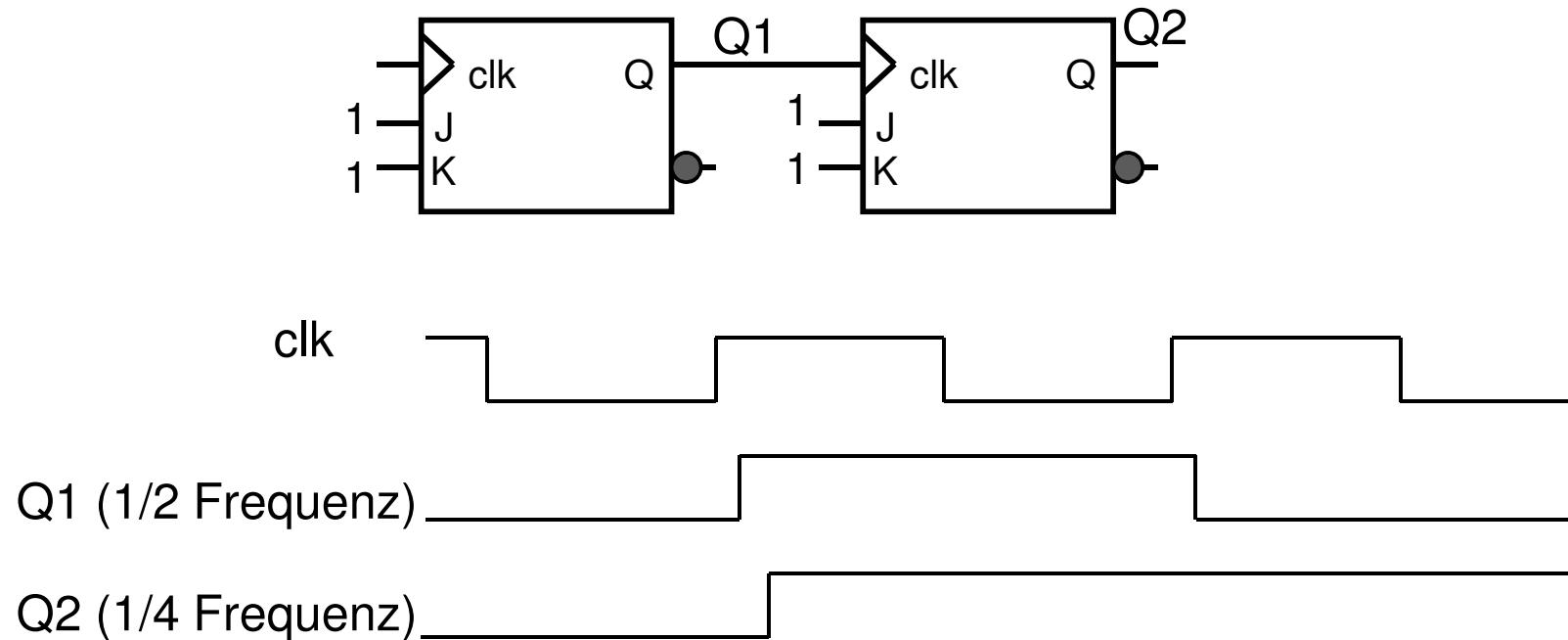


| J | K | Q | R=K und Q | S=J und QZ | Q' |
|---|---|---|-----------|------------|----|
| 0 | 0 | Q | 0 | 0 | Q |
| 0 | 1 | Q | Q | 0 | 0 |
| 1 | 0 | Q | 0 | QZ | 1 |
| 1 | 1 | Q | Q | QZ | QZ |

Das FlipFlop kippt für $(J,K)=(1,1)$ den aktuellen Zustand.

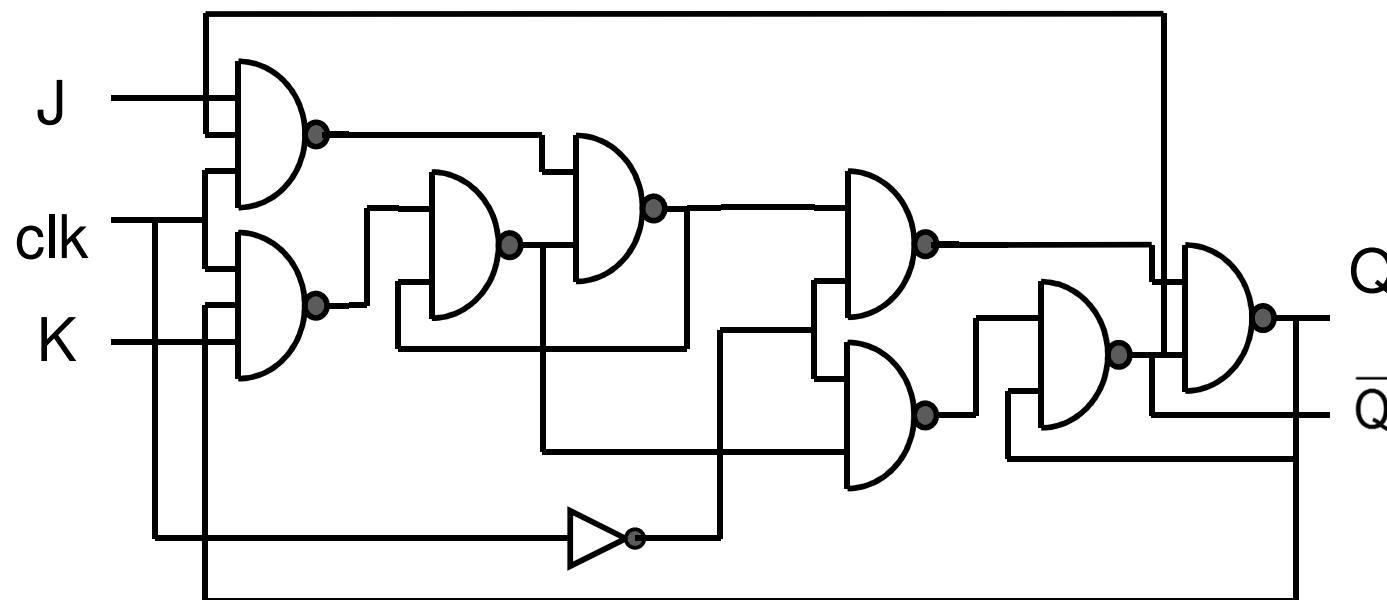
Das J/K FlipFlop

Beispiel: Ein typisches Beispiel für den Einsatz eines J/K FlipFlops ist das Teilen eines Taktes.



Das J/K FlipFlop ff

Aufbau bei direkter Benutzung von Basis R/S FlipFlops



1.3.7 Tristate Treiber und Busse

Unsere Gatter hatten bisher eine strenge Unterscheidung zwischen Eingängen (Transistorgates) und Ausgängen (getrieben durch leitende Pfade nach V_{SS} oder V_{DD}).

Dies reicht auch für viele Probleme zum Entwurf vollkommen aus.

Will man aber komplexere Komponenten miteinander verbinden, dann sind auch die Verbindungsleitungen mit hohen Kosten verbunden, besonders dann, wenn man über Chipgrenzen hinweg geht.
(Lötpunkte, Stecker, Pins am Chipgehäuse)

Problem: Schaffe die Möglichkeit n Komponenten auf möglichst einfache Weise miteinander zu verbinden, so dass jede Komponente mit jeder Komponente direkten (ohne über andere Komponenten zu gehen) Kontakt aufnehmen kann.

Busse und Tristates ff

Die einzige Möglichkeit, diese direkte Verbindung zu schaffen, setzt voraus, dass

- ↳ jede Komponente $n-1$ Eingangs- und $n-1$ Ausgangsports hat,
- ↳ insgesamt $2n(n-1)$ Verbindungen geschaffen werden müssen, und
- ↳ der Entwurf einer Komponente von vorneherein durch die Komponenten bestimmt wird, mit denen sie zusammenarbeiten soll.

Selbst wenn man die Nebenbedingung direkter Kommunikation weglässt, müsste man eine Menge von Kommunikationsbausteinen explizit miteinander verschalten, so dass alle Verbindungen möglich sind.

Ausweg:

Sehe Ports vor, die Eingang oder Ausgang sein können, oder sich neutral verhalten.

Busse und Tristates ff

Frage: Wann verhält sich ein Anschluss "neutral"?

Antwort: Ein Anschluss verhält sich neutral, wenn er einen sehr hohen Widerstand gegen beide Versorgungspole hat. Er verhält sich dann isoliert und bildet allenfalls noch eine kapazitive und induktive Last auf der Leitung an die er gebunden ist.

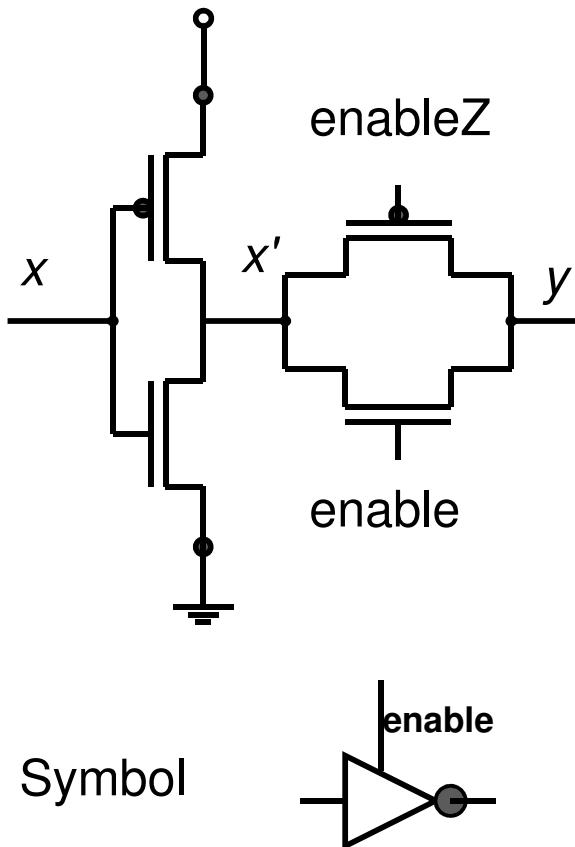
Man nennt einen solchen Zustand eines Anschlusses auch "hochohmig" oder "high impedance state".

Wir können ein solches Verhalten leicht mit CMOS Transistoren realisieren, da sie als Schalter arbeiten. Man muss lediglich mit einem Schalter den Ausgang von seinen Treibern trennen können.

Dies leistet ein sogenannter **Tristate Treiber**:

Der Tristate Treiber

Die einfachste Möglichkeit, für einen Ausgang neutrales Verhalten zu erhalten, ist die Erweiterung eines Inverters mit einem Schalter:



enable und enableZ sind komplementär zu wählen, d.h. stets

$$\text{enable} = \text{nicht enableZ}$$

Verhalten:

enable = 0: y ist von x' isoliert. Es gibt seitens des Inverters keine leitende Verbindung von y nach V_{SS} oder V_{DD} .

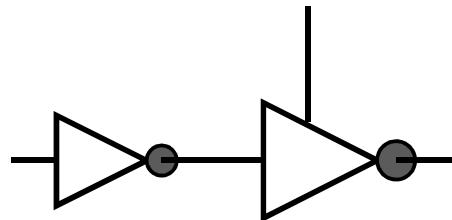
enable = 1: y ist über den (die) Transistoren mit x' verbunden.

Benutzt man nur einen Transistor, verliert man aber beim Ausgangsspeigel die Schwellenspannung eines Transistors.

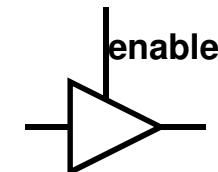
Der Tristate Treiber

Der Baustein auf der Basis eines Inverters ist allerdings invertierend, d.h. er kehrt das Eingangssignal um.

Es ist nun aber eine leichte Übung, auch einen nichtinvertierenden Treiber aufzubauen: Man benutze einfach einen Doppelinverter.



Symbol



Bemerkung:

Meistens nutzt man die Inverterkaskade zur Signalverstärkung, d.h. man benutzt im zweiten Inverter grössere Transistoren als im ersten Inverter. Dadurch kann ein solcher Treiber auch relativ hohe Lasten schnell schalten. Dies ist insbesondere bei Bussen wichtig.

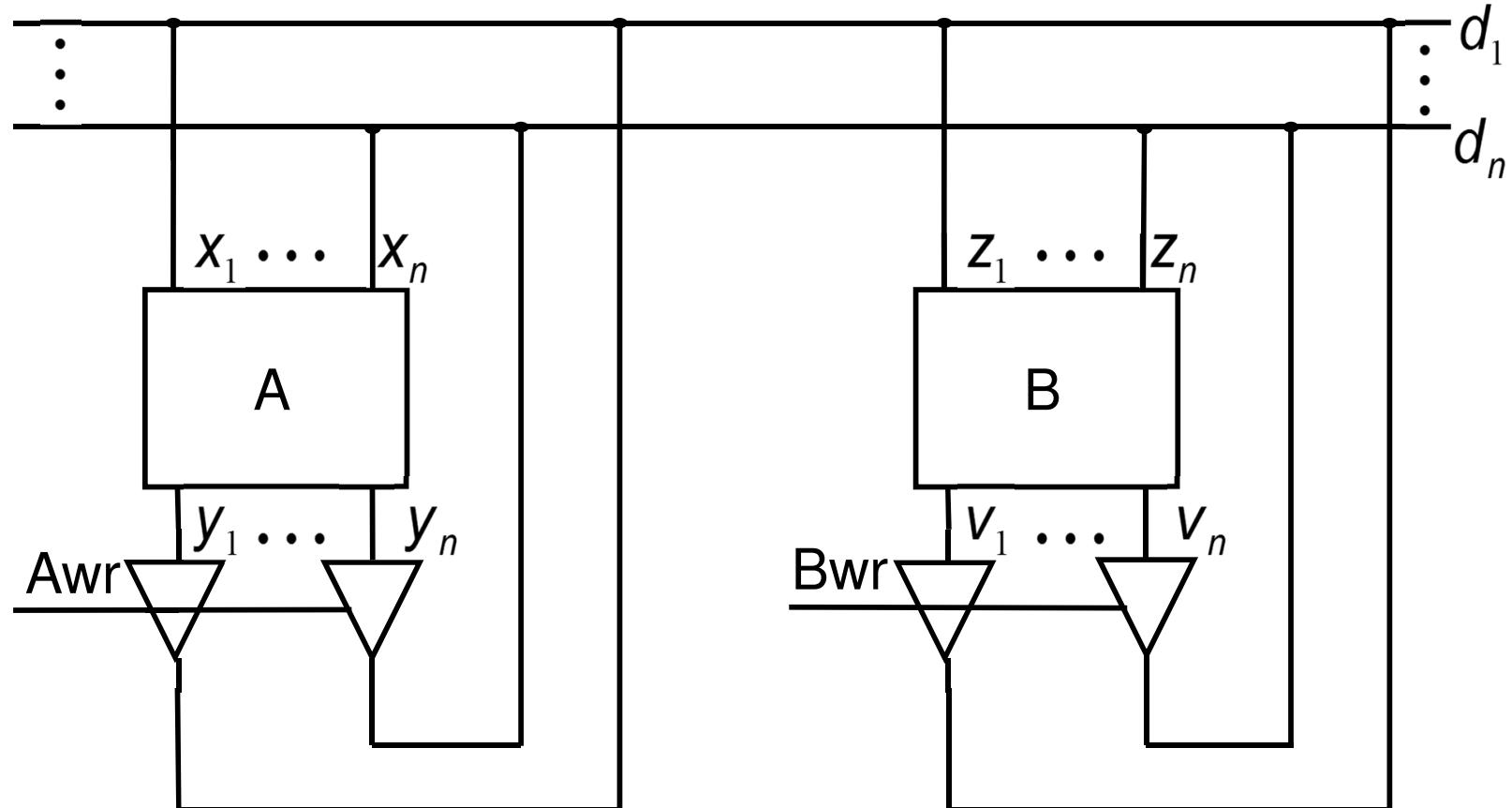
Busse

Ein Bus bietet nun auf der Basis solcher bidirektionaler Anschlüsse die Möglichkeit, viele Komponenten direkt miteinander verbinden zu können. Er besteht aus

- einer genormten Menge bidirektionaler Datenleitungen,
- einer genormten Menge von unidirektionalen Kontrollleitungen,
- ggf. einer Menge von Takteleitungen (synchrone Busse),
- einem Protokoll, in dem vereinbart wird, in welchem zeitlichen Ablauf über den Kontrollleitungen eine Komponente auf die Datenleitungen schreiben darf, bzw. von ihnen lesen kann.

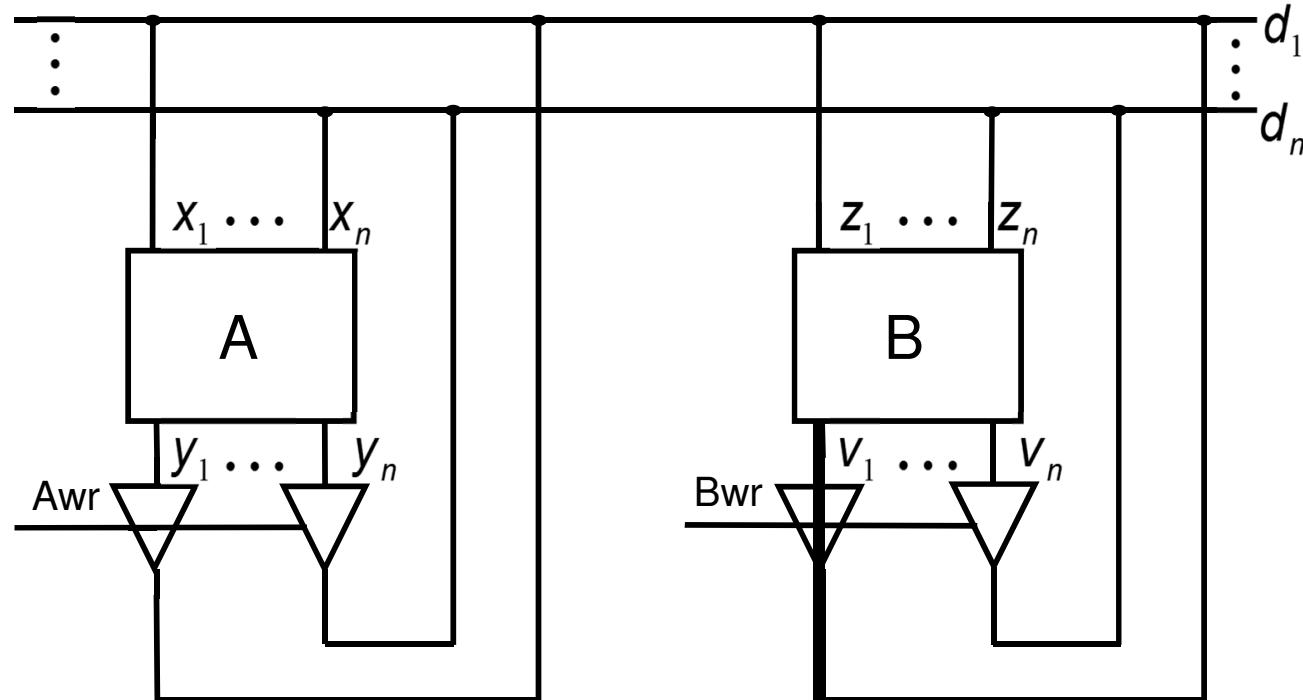
Wir wollen es vorerst bei einem einfachen Beispiel dazu belassen:

Beispiel: bidirektonaler Bus



Beide Komponenten A und B können von d lesen oder auf d schreiben.
Aus den Kontrollsignalen (nicht eingezeichnet) sind nun Awr und Bwr
korrekt zu erzeugen.

Beispiel: bidirektionaler Bus

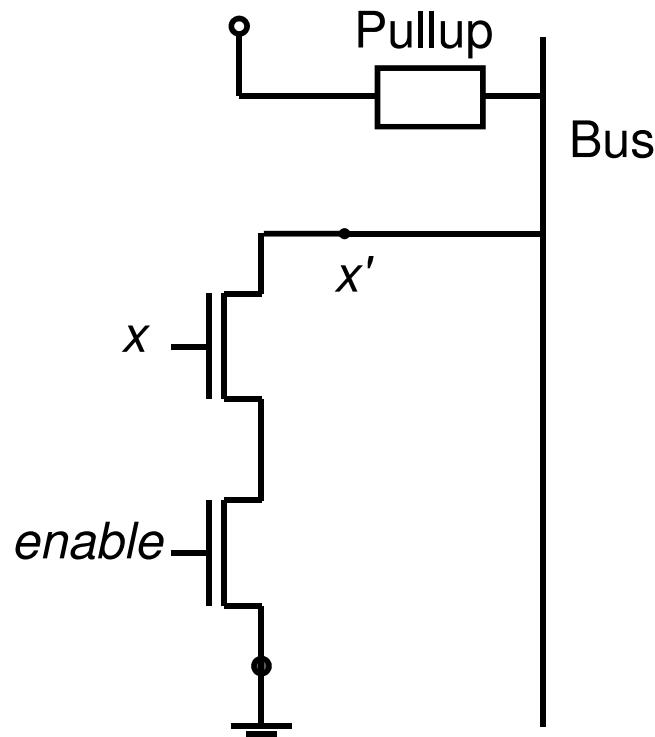


$Awr = 0, Bwr = 1$: Komponente A liest von Komponente B

$Awr = 1, Bwr = 0$: Komponente B liest von Komponente A

Ältere Tristate Techniken

Eine andere Möglichkeit, ein hochohmiges Verhalten zu realisieren, besteht darin, dass man eine der Verbindungen (V_{SS} oder V_{DD}) nicht herstellt, sondern über einen Abschlusswiderstand (Pulldown oder Pullup) realisiert, der sehr viel grösser ist, als der Widerstand des schaltbaren Pfades:



enable = 0: x' ist isoliert. Der Bus wird über den Abschlusswiderstand (Pullup) auf 1 gehalten

enable = 1: x' ist mit V_{SS} verbunden, falls $x = 1$ ist. Der Bus wird auf $x' = 0$ gezogen.

Bemerkung: Es erfolgt eine UND Verknüpfung (ODER bei Pulldown Abschluss) aller schreibenden Komponenten (Wired AND, Wired OR).

Modellierung von Tristates in WüHDL

Um diesem Konzept Rechnung zu tragen, müssen wir in WüHDL ein weiteres Konzept einführen:

- Es muss Signale geben, die von mehreren Prozessen beschrieben werden dürfen (mehrere Treiber).
- Wenn zu einem Zeitpunkt mehrere Treiber einem Signal einen Wert zuweisen, muss aus allen diesen Werten ein aufgelöster Wert berechnet werden.

Dazu sind sogenannte resolved Types vorgesehen:

Ein resolved Type *ResolvedT* muss Untertyp eines schon definierten Typs *UnresolvedT* sein, und es muss eine Auflösungsfunktion

```
FUNCTION resolveT(ARRAY(natural RANGE<>) OF  
UnresolvedT)  
RETURN UnresolvedT
```

existieren, die die Auflösung berechnet.

Wir verdeutlichen dies zunächst an einem einfachen Beispiel, das nur die Benutzung eines hochohmigen Zustandes erlaubt.

Resolved Types

```
TYPE Utri_state_logic IS -- nicht aufgelöster Typ
( 'Z', -- für den hochohmigen Zustand
  '0', -- für den Logikwert 0
  '1', -- für den Logikwert 1
  'X' -- für Fehlerbedingungen (verschiedene Pegel ungleich Z)
);
-- Wird als Parametertyp für die Auflösungsfunktion benötigt.
TYPE Utri_state_logic_vector
IS ARRAY (INTEGER RANGE <>) OF Utri_state_logic;
-- Nützlicher Typ zur Definition der Auflösung über eine Tabelle
-- von TYP x TYP
TYPE resolutiontable
IS ARRAY(Utri_state_logic, Utri_state_logic) OF Utri_state_logic;
```

Resolved Types -- ff

```
CONSTANT resolvable : resolutiontable :=
```

```
-----
```

```
--| 'z' | '0' | '1' | 'x' |  
( ('z', '0', '1', 'x'), --'z' |  
  ('0', '0', 'x', 'x'), --'0' |  
  ('1', 'x', '1', 'x'), --'1' |  
  ('x', 'x', 'x', 'x')) ; --'x' |
```

```
-----
```

Resolved Types

```
FUNCTION resolve_tri_state_logic (values: IN Utri_state_logic_vector)
    RETURN Utri_state_logic IS
    VARIABLE result: tri_state_logic := 'Z';
BEGIN
    FOR index IN values'range
    LOOP
        result := resolvable(result,values(index));
    END LOOP;
    RETURN result;
END FUNCTION resolve_tri_state_logic;

-- Definition des aufgelösten Typs
SUBTYPE tri_state_logic IS resolve_tri_state_logic Utri_state_logic;
```

Das Package IEEE.std_logic_1164

Sehr weite Verbreitung als aufgelöster Typ zum Ersatz des Typs BIT findet der in der Bibliothek **IEEE** im Paket **std_logic_1164** definierte, aufgelöste Typ **std_logic**.

Im Package werden die booleschen Operatoren entsprechend überladen und Vektoren ebenfalls als Typ bereitgestellt.

PACKAGE std_logic_1164 IS

-- logic state system

TYPE std_ulogic IS (-- Unaufgelöste Fassung von std_logic

| | | |
|--------|--------------------|-------------------------|
| 'U', | -- Uninitialized | |
| 'X', | -- Forcing Unknown | (0 gegen 1) |
| '0', | -- Forcing 0 | |
| '1', | -- Forcing 1 | |
| 'Z', | -- High Impedance | (Hochohmig) |
| 'W', | -- Weak Unknown | (Pullup gegen Pulldown) |
| 'L', | -- Weak 0 | (Pulldown) |
| 'H', | -- Weak 1 | (Pullup) |
| '-'); | -- Don't care | (später!) |

1.4 WüHDL

--

Ein erster Überblick

Zur Vorlesung Rechenanlagen

SS 2019



Wir haben an Beispielen schon gesehen, wie man das Verhalten von Bausteinen in WüHDL beschreiben kann. Da man jedes Stück Hardware im Grunde als Baustein mit den nach außen geführten Signalen als Schnittstelle auffassen kann, kann man die Definition des Verhaltens stets als

- eine **Architecture** auffassen:
 - Spezifikation des Verhaltens durch
 - Programmcode (Prozesse), d.h. man beschreibt das Verhalten mit dem Repertoire einer gewöhnlichen Programmiersprache (Ada ähnlich), oder
 - Strukturbeschreibung, d.h. man definiert das Verhalten durch Instanziierung und Verbindung anderer Komponenten (Schaltkreise)

Wir wollen nun die wichtigsten Konzepte, die wir in den Beispielen schon benutzt haben, zusammenfassen:

Architekturen -- die Definition der Komponenten

Definitionsmöglichkeiten

Man hat schon aus der Syntax ersichtlich mehrere Möglichkeiten, das Verhalten von Komponenten zu definieren. In WüHDL sehen wir folgende Optionen vor:

- Beschreibung durch Prozesse
- Beschreibung durch nebenläufige Signalzuweisungen (concurrent signal assignments)
- Beschreibung durch Instanziierung und Verbindung von Komponenten.

Die erste Variante kann man als Verhaltensdefinition auffassen, während die letztere eine Strukturdefinition ist (Aufbau über Instanzen vorhandener Komponenten). Die zweite Variante kann man sowohl als Verhaltens- als auch als Strukturdefinition auffassen (vgl. spätere Kapitel). Wir klammern sie vorläufig aus.

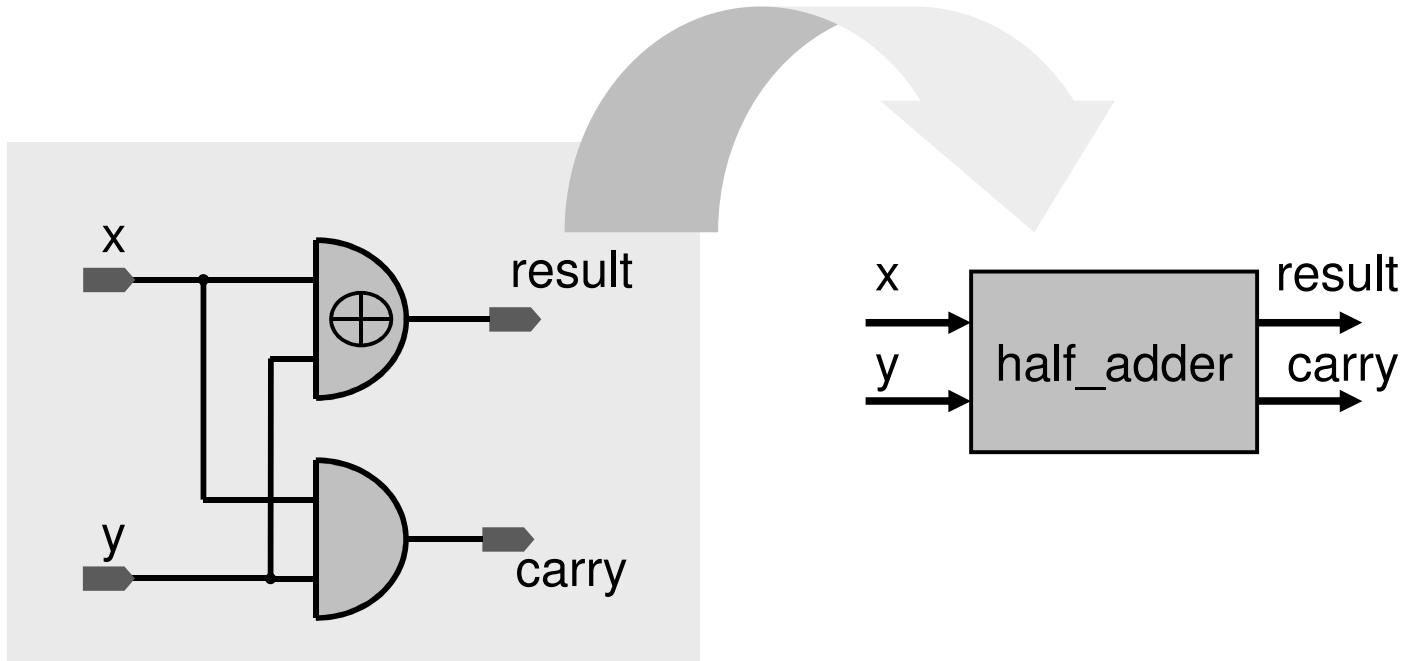
Architekturen -- Definition durch Prozesse

Das Verhalten eines Stückes Hardware kann man durch mehrere nebenläufige Prozesse beschreiben, die als sequentielle Programme bis zur Terminierung ausgeführt werden. Innerhalb eines Prozesses können neue Waveforms vom Auswertezeitpunkt in die Zukunft hin Signalen zugewiesen werden. Dadurch entstehende Änderungen an Signalen können dann wieder andere Prozesse zur Auswertung anstoßen:

Triviales Beispiel

```
ARCHITECTURE behavior1 OF half_adder IS
BEGIN
  PROCESS (x, y)
    -- x,y Signale bei deren Änderung der Prozess anläuft
    -- man nennt diese Liste auch Sensitivitätsliste
    BEGIN
      result <= x XOR y;
      carry <= x AND y;
    END PROCESS
  END behavior1;
```

Architekturen -- Definition durch strukturelle Beschreibung



Häufig ist es bequem und auch intuitiv, eine Komponente durch einen **Schaltplan** über anderen Komponenten zu definieren, d.h. man benutzt

- Subkomponenten und
- entsprechende Verdrahtung

Architekturen -- Beispiel zu Schaltplänen

Folgendes Programmstück beschreibt das Verhalten eines Halbaddierers durch Instanziierung von Komponenten:

```
ARCHITECTURE structure OF half_adder IS
    COMPONENT xor2
        PORT (a,b: IN BIT; c: OUT BIT);
    END COMPONENT;
    COMPONENT and2
        PORT (a,b: IN BIT; c: OUT BIT);
    END COMPONENT;
BEGIN
    G0: xor2 PORT MAP (x,y,result); -- positionelle Zuordnung
    G1: and2 PORT MAP (a => x; b => y; c => carry );
    -- besser: Zuordnungsliste formaler Port => Signal oder Port
END structure;
```

Architekturen -- Beispiel zu Schaltplänen

Anmerkungen:

Unser Beispiel ist sehr speziell, weil alle Signale des Schaltkreises zugleich auch Ports der Komponente Halfadder sind.

Bei komplexeren Schaltkreisen hat man meist viele Signale, die nur als Mittler zwischen den beteiligten Komponenten fungieren, aber selbst nach außen nicht sichtbar sind. Diese Signale sind dann als lokale Signale in der Architecture zu deklarieren.

Viele Entwicklungswerkzeuge sehen für solche Strukturbeschreibungen spezielle Unterstützung vor:

Einen **graphischen Symboleditor**, mit dem man Komponentensymbole mit Portbezeichnungen zeichnen kann und einen **graphischen Schaltplaneditor**, mit dem man Symbole von Komponenten platzieren und deren Ports verdrahten kann.

Die Benutzung solcher Tools nimmt einem die mühevolle Arbeit ab, WüHDL bzw. VDHL Code eingeben zu müssen:

Symboleditor → entity declaration

Schaltplaneditor → architecture Definition mit Instanziierungen

Konstanten, Variablen und Signale

Konstanten

Konstanten stehen für Objekte, deren Wert nur einmal festgelegt und dann nicht mehr geändert werden kann. Ihr Wert muss zur Übersetzungszeit, d.h. wenn eine WüHDL Spezifikation von einem Übersetzerprogramm übersetzt wird, feststehen. Sie beschreiben in der Regel typische Größen eines Entwurfs.

Beispiele:

```
CONSTANT clockperiod: TIME := 10 ns; -- 100Mhz  
CONSTANT pi : REAL := 3.1415;  
CONSTANT defaultcode: BIT_VECTOR(0 TO 3) := "1001";
```

Konstanten, Variablen und Signale

Variablen

- unterscheiden sich nicht von Variablen anderer höherer Programmiersprachen.
- haben "kein Gegenstück in der Hardware" !!
- werden nur im Rahmen von Zwischenrechnungen zur Verhaltensdefinition verwendet (nur lokal in Prozessen, Funktionen, Prozeduren, Paketen, ...nicht in Architectures ...).

Beispiele

```
VARIABLE row, column : INTEGER RANGE 0 TO 31;  
VARIABLE sum : NATURAL := 0;  
VARIABLE feld : Two_Dim (0 TO 15, 0 TO 31);
```

Konstanten, Variablen und Signale

Signale

- sind in anderen Programmiersprachen nicht zu finden.
- modellieren den zeitlichen Ablauf des Verhaltens der Hardware, sie bringen überhaupt den Zeitbegriff erst ins Spiel.
- sind die einzige Möglichkeit für nebenläufig arbeitende Komponenten, miteinander zu kommunizieren.
- können über jedem Typ definiert sein.
- tragen Werte, die stets eine Abbildung von TIME in den Wertebereich des Typs sind, während der Wert einer Variablen nur ein Wert aus dem Wertebereichs des Typs der Variablen ist!

Beispiele

```
SIGNAL clk, resetn      : BIT;  
SIGNAL counter          : INTEGER RANGE 0 TO 31;  
SIGNAL reg_ram          : Two_Dim (0 TO 15, 0 TO 31);
```

Variablen versus Signale

Verschiedene Syntax für Zuweisungen

- <variable_assignment> ::= <target> := <expression>;
- <signal_assignment> ::=
 <target> <= [**TRANSPORT**] <waveform_element>
 { , <waveform_element> }* ;
- <waveform_element> ::=
 <value_expression> [**AFTER** <time_expression>] |
 NULL [**AFTER** <time_expression>]

Variablen weist man Werte ihres Typs zu, Signalen weist man Wellenformen (Signalwerte) über ihrem Typ zu. Dabei beziehen sich die Zeitangaben (AFTER) relativ auf den Zeitpunkt der Zuweisung. Fehlt die Zeitangabe, dann ist die Verzögerung = 0 (!).

Verzögerungsarten bei Signalen

Mit einem Signal ist als Wert stets eine zeitdiskrete **Waveform** über dem Wertebereich seines Typs assoziiert. Mit einer Zuweisung fügt man nun neue Schaltereignisse in die Waveform ein, die **ab dem aktuellen Zeitpunkt** stattfinden können. Es stellt sich folgende Frage:

- ? Wie verfährt man, wenn hinter dem aktuellen Zeitpunkt schon Änderungen in der Waveform stehen?

Eine Zuweisung an ein Signal bedeutet eine **Fortführung** der Waveform **aus dem Wissen** über das Systemverhalten **bis zum aktuellen Zeitpunkt**.

Es gibt nun zwei Fälle:

- Es sollen Ereignisse vor schon bestehenden Ereignissen eingefügt werden.
- Es sollen Ereignisse nach schon bestehende Ereignissen eingefügt werden.

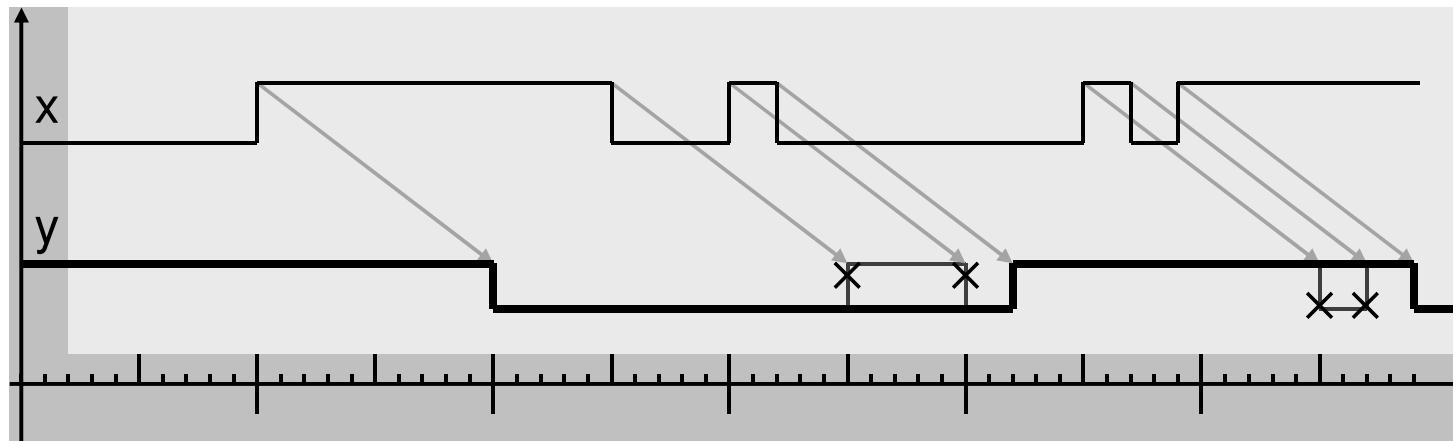
Verzögerungsarten bei Signalen

- Einfügen **vor** schon bestehenden Ereignissen:
Führt zu einem Löschen aller nachfolgenden Ereignisse
Grund: Aus dem Wissen über das Verhalten des Systems bis zum aktuellen Zeitpunkt ist die bisherige Waveform bestimmt. Nun entsteht aber ein Ereignis, das dieses Wissen zurücknimmt. Man muss also die für die Zukunft getroffenen Änderungen zurücknehmen.
- Einfügen **hinter** schon bestehenden Ereignissen
WüHDL unterscheidet dazu zwischen 2 Arten der Verzögerung
 - **träge Verzögerung** (inertial delay)
 - **nichtträge Verzögerung** (transport delay)

Die Verzögerungsart INERTIAL

- INERTIAL ist die **Default Einstellung**
- beim Einfügen eines Ereignisses werden alle vorhergehenden Ereignisse zwischen aktuellem Zeitpunkt und dem Einfügezeitpunkt gelöscht, bis auf das erste Vorgängerereignis, falls existent, das denselben Wert erzeugt.
- dieses Modell erzeugt keine Signalpulse an Komponenten, die kürzer sind als deren Verzögerung:

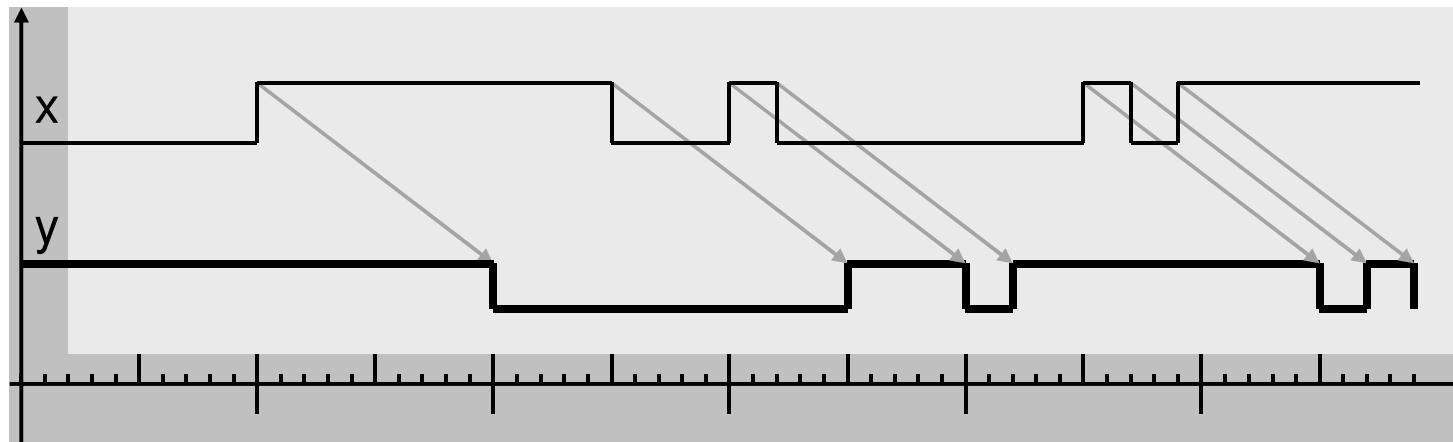
$y \leq \text{not } x \text{ AFTER } 10 \text{ ps};$



Die Verzögerungsart TRANSPORT

- Transportverzögerung muss explizit durch Voranstellen des Schüsselworts TRANSPORT erzwungen werden.
- Beim Einfügen eines Ereignisses bleiben alle vorhergehenden Ereignisse zwischen aktuellem Zeitpunkt und dem Einfügezeitpunkt erhalten.
- Dieses Modell kann sehr kurze Signalpulse an Komponenten erzeugen, die physikalisch ggf. gar nicht mehr entstehen:

$y \leq \text{TRANSPORT not } x \text{ AFTER } 10 \text{ ps};$



Fazit: Variablen versus Signale

- Deklaration
 - Variablen werden in Prozessen / Unterprogrammen deklariert und sind auch nur da sichtbar.
 - Signale können nicht in Prozessen / Unterprogrammen deklariert werden.
- Verwendung
 - Variablen werden zum Abspeichern temporärer Werte benutzt.
 - Signale stehen üblicherweise für Verbindungen in der Hardware.
- Wertzuweisung
 - bei Variablen erfolgt sofort, wenn die Variablenzuweisung ausgeführt wird.
 - bei Signalen erfolgt erst, nachdem der zuweisende Prozess terminiert ist.

Ausführung eines WüHDL Modells

Eine WüHDL Spezifikation besteht bis jetzt also aus

- einer Menge von Entities
- einer Architektur zu jeder Entity
- einer Menge von Instanzen von Bausteinen, zu denen selbst wieder
 - Prozesse und
 - Instanzen von Bausteinendefiniert sind.

Ist die Architektur einer Bausteininstanz A wieder durch Instanzen $\{b_1, \dots, b_n\}$ von Bausteinen definiert, so kann man dies auch auffassen als eine Instanziierung von Bausteinen $A.b_1, \dots, A.b_n$. Man spricht in diesem Zusammenhang auch von einer **Entwurfshierarchie**.

Entwurfshierarchie bei WüHDL Modellen

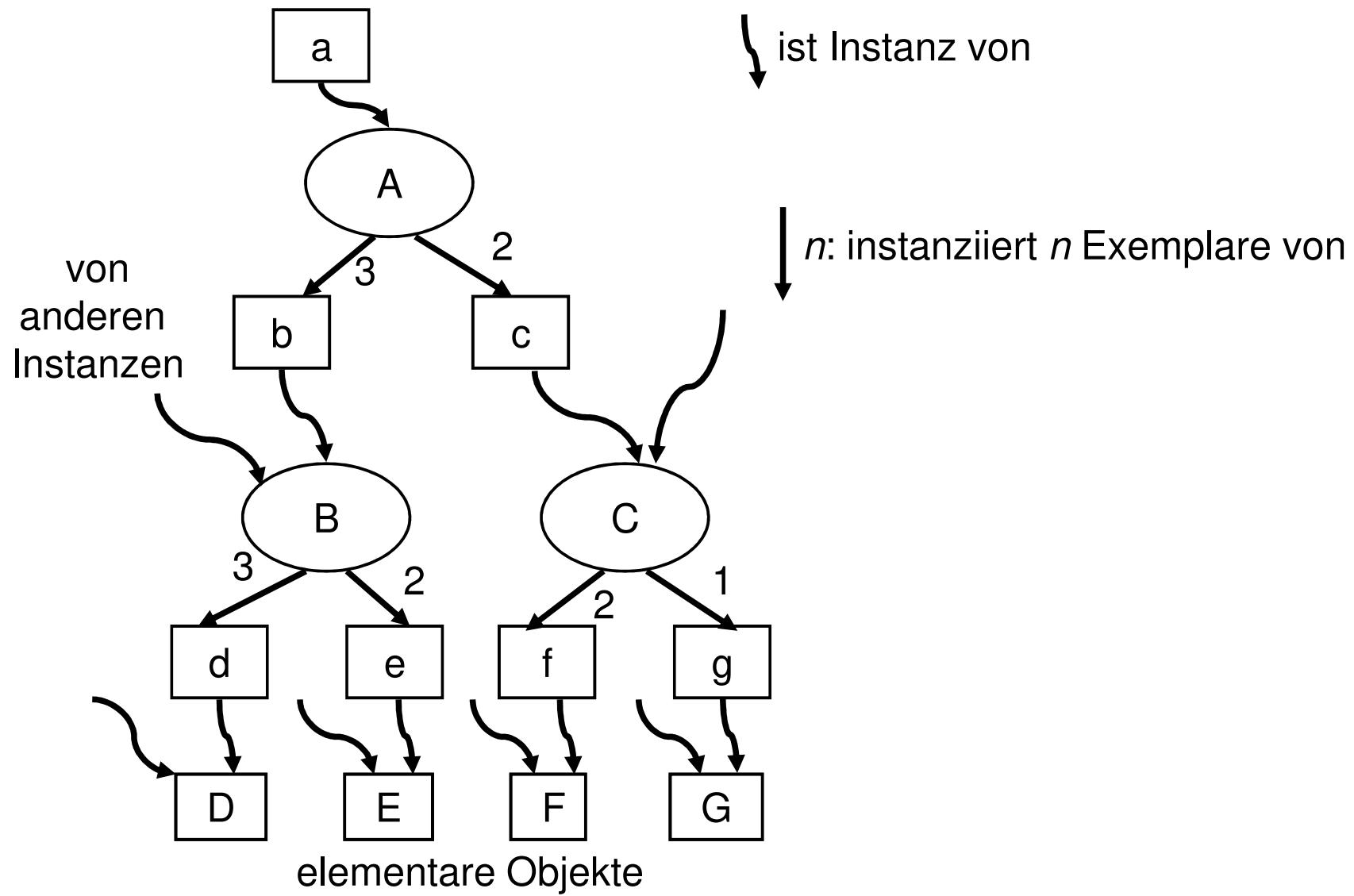
Hierarchie ist ein fundamentales Konzept zur Konstruktion komplexer Systeme, weil man eine riesige Anzahl von Instanzen auf sehr kompakte Art und Weise definieren kann. Man beschreibt Komponenten, die andere Komponenten instanziieren nur einmal, bei jeder Instanziierung der Komponente werden aber auch alle Subkomponenten instanziert.

Man erhält auf diese Weise eine sehr kompakte Spezifikation eines riesigen Objekts bestehend aus tausenden Instanzen elementarer Objekte.

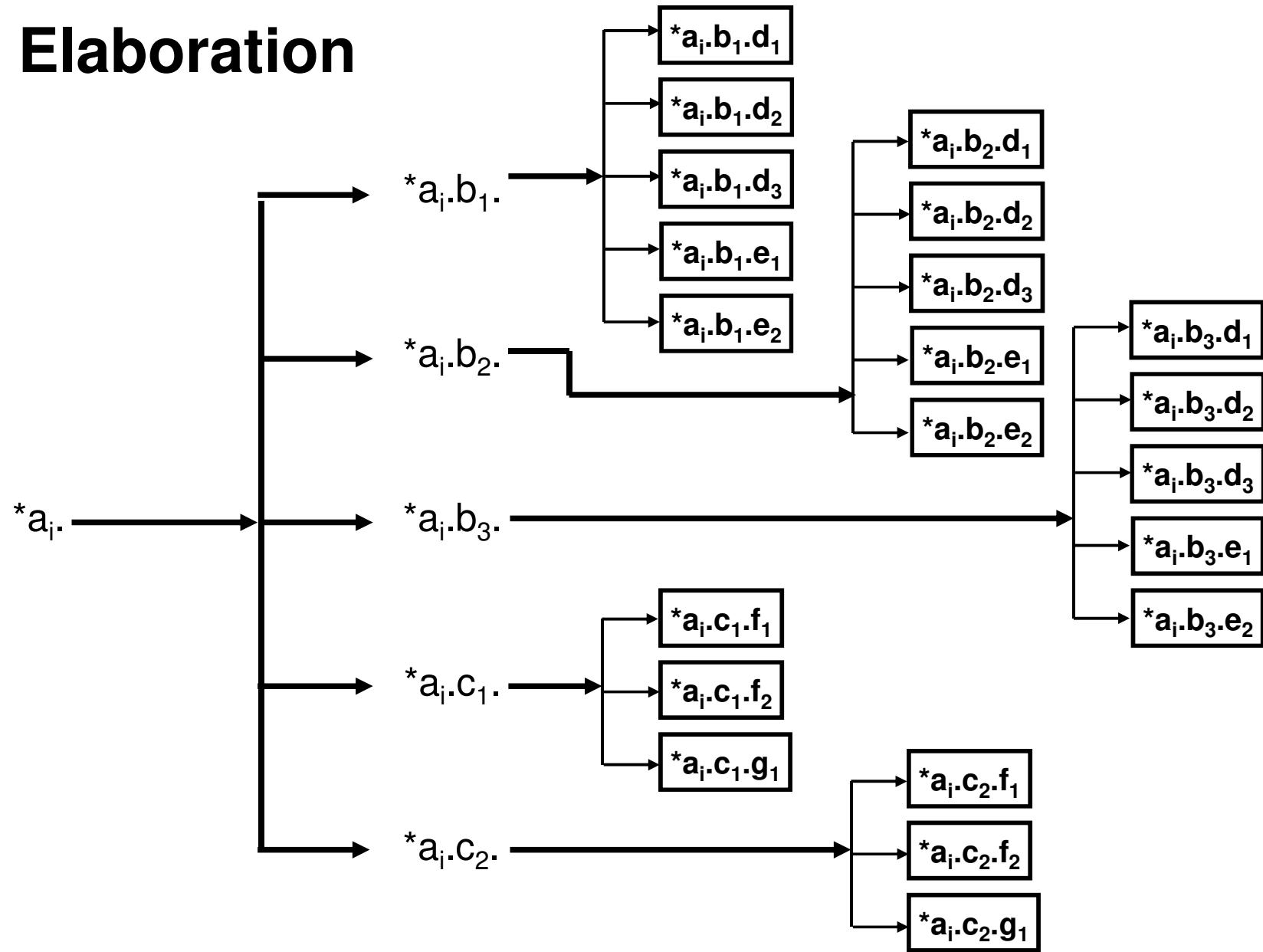
Elementar in WüHDL sind Prozesse und Signale, da sie keine Subobjekte instanziieren.

Folgendes Bild soll die Kraft der Hierarchie verdeutlichen:

Entwurfshierarchie -- ff



Elaboration



Ausführung eines WüHDL Modells -- ff

Nach der Elaboration der durch die WüHDL Modelle definierten Hierarchie hat man es also im Grunde nur mit einer Vielzahl von Instanzen von Prozessen und Signalen zu tun. Dabei muss folgendes gelten:

- Jede Signalinstanz eines unaufgelösten Signaltyps wird von höchstens einer Prozessinstanz zugewiesen. Da dies zwingend erfordert, dass in unserer Beschreibung ein Signal höchstens in einem Prozess eine Zuweisung erhalten darf, nennen wir diesen Prozess auch **den Treiber** des Signals.
Nach Elaboration bedeutet dies, dass jede Instanz eines Treiberprozesses eine entsprechende Instanz eines Signals treibt.
- Zu jeder Signalinstanz eines aufgelösten Typs wird zu jeder Prozessinstanz, die dem Signal etwas zuweist, ein Treiber angelegt.
Aufgelöste Signalinstanzen haben also mehrere Prozessinstanzen als Treiber.

Ausführung eines WüHDL Modells -- ff

Die Prozessinstanzen stehen nun für sequentielle Programme, wobei zu jeder Prozessinstanz auch entsprechend lokale Variablen instanziert werden.

Ferner hängt nach dem von uns bisher eingeführten Konzept jeder Prozess von einer Menge von Signalen ab, auf deren Änderung er reagieren muss.

Wir können also jeder Prozessinstanz eine Menge von Signalinstanzen zuordnen, gegenüber denen er sensitiv ist, die **Sensitivitätsliste**.

Die Auswertung startet nun stets zum Zeitpunkt 0.

Alle Signale und Variablen vom Typ T werden mit T'LOW, d.h. dem niedrigsten Wert des Typs initialisiert, bzw. T'LOW von 0 bis unendlich (Signale).

Die aktuelle Zeit (NOW) ist 0.

DELTA Zyklen

Nun ist das Spiel eigentlich stets dasselbe:

- Starte alle Prozessinstanzen, die sensitiv auf eine Signalinstanz sind, die ein Schaltereignis hat, und lasse sie bis zur Terminierung laufen. Zu Beginn sind dies alle(!) Prozessinstanzen, da sich alle Signalinstanzen initialisieren.
- Sind die Prozessinstanzen terminiert, dann haben sie den Zustand ihrer lokalen Variablen verändert, und Zuweisungen an Signaltreiberinstanzen gemacht. Nun werden die Signaltreiber aller Signalinstanzen ausgewertet und für Signalinstanzen, die eine Zuweisung im Signaltreiber erhalten, ein nächstes Ereignis berechnet (dies kann auch Verzögerung 0 haben).
Bei aufgelösten Signalen wird der Auflösungsfunktion ein Vektor übergeben, dessen Komponenten aus allen Zuweisungswerten von Treibern bestehen, die in diesem Zyklus eine Zuweisung machen.
Als Wert wird der zurückgelieferte, aufgelöste Wert genommen.

DELTA Zyklen -- ff

- Setze nun die aktuelle Zeit auf das Minimum aller Startzeiten von nächsten Ereignissen auf Signalinstanzen.
- Beginne wieder mit einem Delta Zyklus

Probleme machen Änderungen, die mit Verzögerung 0 geschehen. Diese können dazu führen, dass Prozessinstanzen in einem Zeitpunkt sogar mehrfach (sogar unendlich oft) anlaufen und wieder terminieren. Hier kann sich also die Simulation aufhängen.

Vorsicht bei der Modellierung.

Problematisch sind auch Prozesse mit leerer Sensitivitätsliste. Diese laufen automatisch immer wieder an, wenn sie nicht explizit suspendiert werden.

1.5 Schaltfunktionen und boolesche Ausdrücke

Zur Vorlesung Rechenanlagen

SS 2019



Schaltfunktionen

Definition

Eine partielle Funktion $f: B^n \rightarrow B^k$

heißt (allgemeine) Schaltfunktion

Wir haben schon gesehen, dass gewisse Mengen von Schaltfunktionen eine boolesche Algebra bilden:

$S_n := Abb(B^n, B)$ bildet unter punktweiser Verknüpfung

$$f \vee g(p) := f(p) \vee g(p);$$

$$f \cdot g(p) := f(p) \cdot g(p); \bar{f}(p) := \overline{f(p)}$$

eine boolesche Algebra.

Schaltfunktionen ff

Die Elemente f aus $\mathcal{S}_{n,k} := \text{Abb}(\mathbf{B}^n, \mathbf{B}^k)$

kann man darstellen als

$$f(x) = (f_1(x), \dots, f_k(x)), \text{ mit } f_i \in \mathcal{S}_n$$

Also sollte unser Hauptinteresse zunächst einmal folgenden booleschen Algebren gelten:

$$\mathcal{S}_n \text{ sowie } \mathcal{S}_n^D := \{f: \mathbf{B}^n \rightarrow \mathbf{B} \mid D(f) = D\}$$

Betrachte die Schaltfunktionen $x_i: \mathbf{B}^n \rightarrow \mathbf{B}$ mit $x_i(p) = p_i$

Sie sind trivial zu realisieren, weil sie Projektionen auf die i -te Komponente sind, also einfach der i -te Eingang:



Schaltfunktionen

Satz

Jede Schaltfunktion aus S_n kann mittels $\{x_1, \dots, x_n\}$ und den Operatoren \cdot, \vee, \neg dargestellt werden.

Beweis:

Zunächst eine Notation:

Für ein Element g einer booleschen Algebra und q aus B sei:

$$g^q := \begin{cases} g & \text{falls } q = 1 \\ \bar{g} & \text{falls } q = 0 \end{cases}$$

Betrachte nun für $p \in B^n$ das Element $x^p := x_1^{p_1} \cdot \dots \cdot x_n^{p_n}$

Beweis: ff

Dann ist $x^p(q) = x_1^{p_1} \cdot \dots \cdot x_n^{p_n}(q) = 1$

$$\Leftrightarrow \forall 1 \leq i \leq n: x_i^{p_i}(q) = 1$$

$$\Leftrightarrow \forall 1 \leq i \leq n: (x_i(q))^{p_i} = 1$$

$$\Leftrightarrow \forall 1 \leq i \leq n: (q_i)^{p_i} = 1$$

$$\Leftrightarrow \forall 1 \leq i \leq n: q_i = p_i \quad \boxed{\Leftrightarrow p = q}$$

Also liefert $x_1^{p_1} \cdot \dots \cdot x_n^{p_n}$ genau den Minterm x^p

Da aber $At(S_n) = \{x^p \mid p \in B^n\}$

können wir jedes Element f darstellen als

$$f = \bigvee_{p \in ON(f)} x_1^{p_1} \cdot \dots \cdot x_n^{p_n}$$

Disjunktive Normalform

Definition

Wir nennen die Menge

$$ON(f) := \{p \mid f(p) = 1\} \quad (= \{p \mid f \cdot x^p = x^p\})$$

ON-Set der Funktion f, und die Form

$$f = \bigvee_{p \in ON(f)} x_1^{p_1} \cdot \dots \cdot x_n^{p_n}$$

die **disjuktive Normalform von f.**

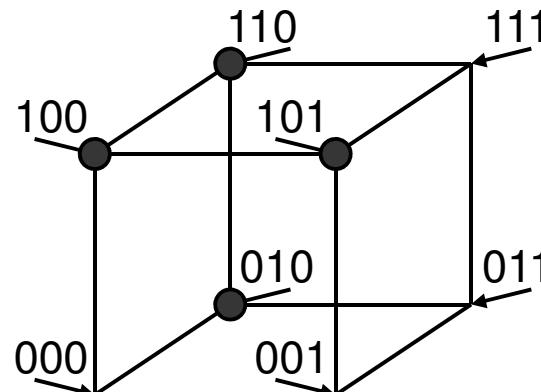
Jede totale Schaltfunktion in einem Ausgang lässt sich also eindeutig mit den Operationen and, or und not, sowie den Projektionen (Eingängen) darstellen.

Anschauung:

Man kann die Vektoren $p \in \mathbb{B}^n$ auch als Eckpunkte eines n -dimensionalen Einheitswürfels auffassen.

Eine Schaltfunktion kann man demnach assoziieren mit ihrem ON-Set, d.h. einer Teilmenge der Eckpunkte.

Einen Minterm x^p kann man gleichsetzen mit einem Eckpunkt:

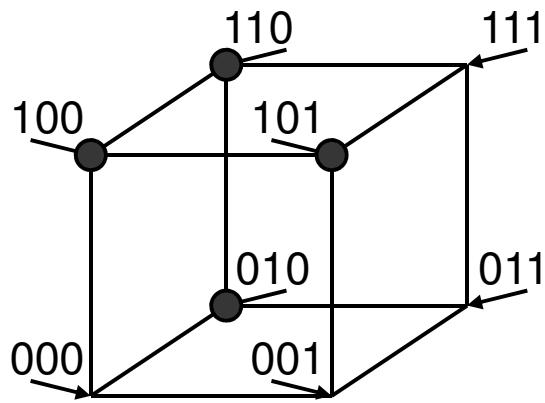


Geometrische Deutung

Funktionstafel

| x_1 | x_2 | x_3 | 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 |

Anschauung:



| x_1 | x_2 | x_3 | 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 |

Wir lesen daraus: $ON(f) = \{010, 100, 101, 110\}$

$$\begin{aligned} \text{Also: } f &= x^{010} \vee x^{100} \vee x^{101} \vee x^{110} \\ &= x_1^0 x_2^1 x_3^0 \vee x_1^1 x_2^0 x_3^0 \vee x_1^1 x_2^0 x_3^1 \vee x_1^1 x_2^1 x_3^0 \\ &= \overline{x_1} x_2 \overline{x_3} \vee x_1 \overline{x_2} \overline{x_3} \vee x_1 \overline{x_2} x_3 \vee x_1 x_2 \overline{x_3} \end{aligned}$$

Boolesche Ausdrücke

Wir benutzen schon seit Einführung der booleschen Algebra Ausdrücke. Da wir im Verlauf der Vorlesung noch viel Gebrauch davon machen werden, wollen wir sie nun auf eine solide, formale Basis stellen.

Ausdrücke sind zunächst bedeutungslos (Kreidehaufen an der Tafel oder Pixel im Display). Sie erhalten eine Bedeutung dadurch, dass man zunächst ihre Struktur, die Syntax, festlegt, und sie dann als Elemente einer Algebra interpretiert.

Definition

Sei $Y = \{y_1, \dots, y_n\}$ eine Menge von Namen. Dann nennen wir Ausdrücke der Form y_i, \bar{y}_i **Litale**.

Boolesche Ausdrücke ff

Definition

Die Menge der **booleschen Ausdrücke** über einer Variablenmenge $Y = \{y_1, \dots, y_m\}$, $BA(Y)$, ist syntaktisch gegeben durch

`<boole_Ausdruck>`

`::= <boole_Ausdruck> \vee <boole_Term> | <boole_Term>`

`<boole_Term>`

`::= <boole_Term> \cdot <boole_Faktor> | <boole_Faktor>`

`<boole_Faktor>`

`::= (<boole_Ausdruck>) | \neg <boole_Faktor>`

`| 0 | 1 | y_1 | | y_m`

Boolesche Ausdrücke ff

Bemerkungen:

Terme sind stets eindeutig zerlegbar in ein Produkt aus Term und Faktor oder sind selbst ein Faktor.

Ausdrücke sind stets eindeutig zerlegbar in eine Disjunktion aus Ausdruck und Term oder sind selbst ein Term

Faktoren sind Konstante, Literale, geklammerte Ausdrücke oder negierte Faktoren.

Beispiele: Sei $Y = \{a, b, c\}$

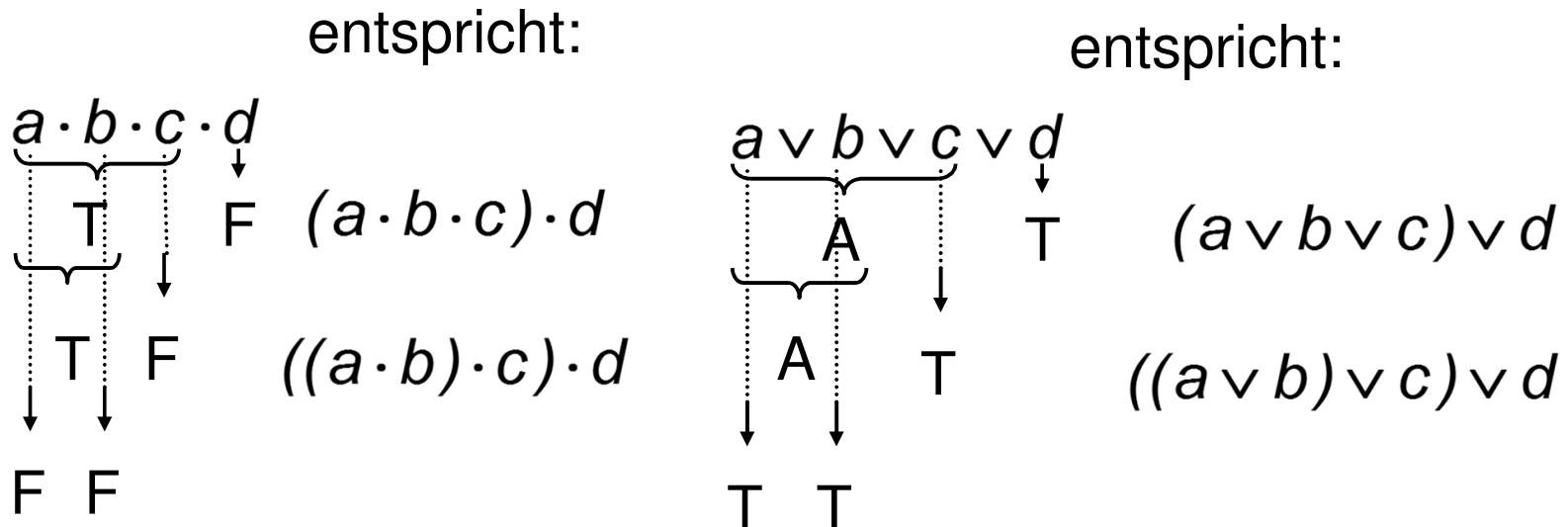
| | | |
|--|--|--|
| $a, a \vee b, a \cdot (\bar{b} \vee c)$ Ausdrücke | $a, a \cdot (\bar{b} \vee c)$ Terme | $a, \bar{b}, (\bar{b} \vee c)$ Faktoren |
|--|--|--|

Boolesche Ausdrücke ff

Bemerkungen:

Durch obige Definition ist ferner eine Zerlegung vorgegeben, die implizit eine Klammerung von links nach rechts bedeutet:

Beispiel:



Spezielle Formen von Ausdrücken

Sei $p = p_1 \cdot \dots \cdot p_n$ und jedes p_i sei ein Literal. Dann heißt p ein **Produkt** (Monom oder Cube).

Hat ein Ausdruck w die Form $w = w_1 \vee \dots \vee w_n$ und jedes w_i ist ein Produkt, dann heißt w **disjunktive Form** (DF oder boolesches Polynom oder Summe von Produkten).

Dual dazu kann man auch betrachten:

Sei $c = c_1 \vee \dots \vee c_n$ und jedes c_i sei ein Literal. Dann heißt c eine **Klausel**.

Hat ein Ausdruck w die Form $w = (w_1) \cdot \dots \cdot (w_n)$ und jedes der w_i ist eine Klausel, dann heißt w **konjunktive Form** (CF oder Produkt von Summen).

Syntaxbäume

Durch obige Definition ist eine eindeutige Zerlegung der Ausdrücke in Teilausdrücke vorgegeben. Diese Zerlegung kann man auch in einem Diagramm veranschaulichen, das wir induktiv nach der Länge eines Ausdruck definieren wollen:

Sei w ein boolescher Ausdruck, dann ist der **abstrakte Syntaxbaum** (Operatorbaum) $T(w)$ zu w gegeben wie folgt:

1. $|w| = 1$: Dann ist w eine Konstante oder Variable. Wir setzen

$$T(w) := \boxed{0} \quad \text{bzw.} \quad \boxed{1} \quad \text{bzw.} \quad \boxed{y_i}$$

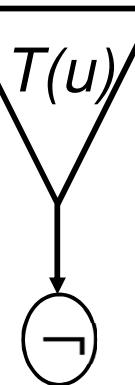
für $T(w) = y_i \in Y$

Syntaxbäume ff

2. Sei für $u, |u| < |w|$ $T(u)$ schon definiert.

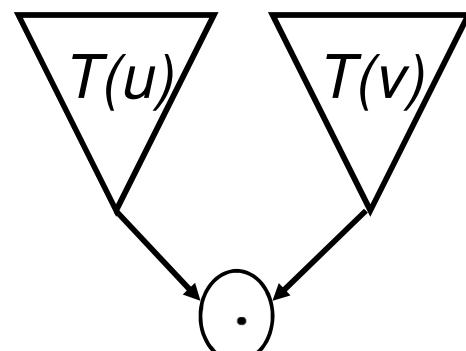
Dann setze $T(w) := T(u)$, falls $w = (u)$

$T(w) := \begin{cases} T(u) & \text{falls } w = \neg u \\ \end{cases}$

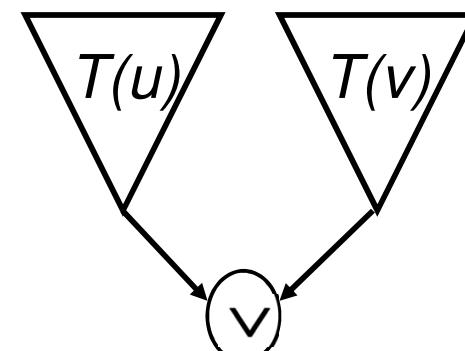


falls $w = u \cdot v$

$T(w) :=$



falls $w = u \vee v$

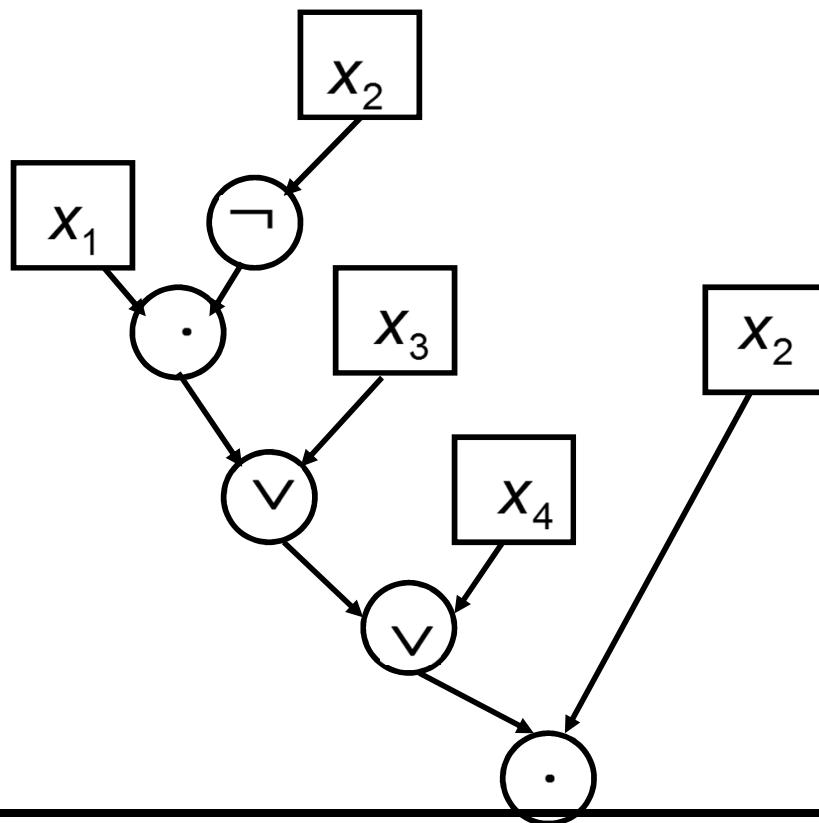


Beispiel

Der abstrakte Syntaxbaum $T(w)$ zu folgender Formel w ist demnach:

$$w = (x_1 \cdot \overline{x_2} \vee x_3 \vee x_4) \cdot x_2$$

$T(w) =$



Interpretation boolescher Ausdrücke

Definition

Die Fortsetzung I einer Zuordnung $I': Y \mapsto M$ in eine boolesche Algebra M auf Ausdrücke $w \in BA(Y)$ nachfolgender Regel

$$I(w) := \begin{cases} w & \text{falls } w \in \{0,1\} \\ I'(w) & \text{falls } w \in Y \\ \overline{I(u)} & \text{falls } w = \bar{u} \\ I(u) & \text{falls } w = (u) \\ I(u) \cdot I(v) & \text{falls } w = u \cdot v \\ I(u) \vee I(v) & \text{falls } w = u \vee v \end{cases}$$

heißt **Interpretation**.

Interpretation ff

Definition

Seien u, v boolesche Ausdrücke. Dann setzen wir

$$u \equiv_I v : \Leftrightarrow I(u) = I(v)$$

$$u \equiv v : \Leftrightarrow \forall I : I(u) = I(v)$$

Beide Relationen sind Äquivalenzrelationen und zerlegen die Menge der Ausdrücke in Klassen. Dabei ist der erste Äquivalenzbegriff schwächer als der zweite, d.h.

$$[u]_{\equiv} \subseteq [u]_{\equiv_I}$$

$u \equiv v$ besagt, dass u und v ausschließlich mit den Rechenregeln der booleschen Algebra ineinander überführt werden können.

Beispiel:

Betrachte folgende Interpretation von $BA(\{a,b\})$ nach S_2^D

mit $D=\{00,01,10\}$ und $I(a) := x_1, I(b) := x_2$

Dann gilt

$$\bar{a}b \vee a\bar{b} \equiv_I a \vee b$$

denn

| x_1 | x_2 | $\bar{x}_1x_2 \vee x_1\bar{x}_2$ | $x_1 \vee x_2$ |
|-------|-------|----------------------------------|----------------|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 |

$=$

Wählt man aber I' nach S_2

| x_1 | x_2 | $\bar{x}_1x_2 \vee x_1\bar{x}_2$ | $x_1 \vee x_2$ |
|-------|-------|----------------------------------|----------------|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |

$$\text{Also } \bar{a}b \vee a\bar{b} \not\equiv_{I'} a \vee b$$

Anmerkung: Wir notieren auch $a \oplus b := \bar{a}b \vee a\bar{b}$

(Exor oder plus mod 2 oder Antivalenz)

Es ist $a \equiv b = \overline{a \oplus b} = a \oplus b \oplus 1$ (Äquivalenz)

Darstellung

Definition

Sei $I: BA(Y) \mapsto M$ eine Interpretation.

Für $w \in BA(Y)$ sagen wir

w ist Darstellung von a : $\Leftrightarrow I(w) = a$

Die Menge der Darstellungen eines Elements a bei Interpretation I ist unendlich groß und gegeben durch

$$[w]_{\equiv_I} \text{ mit } I(w) = a$$

Es ist zum Beispiel: $w \vee 0, w \vee 0 \vee 0, \dots \in [w]_{\equiv_I}$

aber auch $w \cdot (u \vee v) \in [w]_{\equiv_I}$ für $u \in [w]_{\equiv_I}$ und v beliebig.

Das Syntheseproblem

Typisch in der Informatik sind Konstruktionsprobleme, die sich immer dann stellen, wenn man keine eindeutige Darstellung für die zu konstruierenden (darzustellenden) Objekte hat. Allgemein lautet dies:

Syntheseproblem

Gegeben: $w \in X$

eine Interpretation $I: X \mapsto M$

eine Kostenfunktion $C: X \mapsto R$

Gesucht: $v \in X$ mit $I(v) = I(w)$

und $C(v) = \min\{C(u) | I(u) = I(w)\}$

Das Syntheseproblem

Übertragen auf Ausdrücke bedeutet dies zu Beispiel:

Gegeben: ein boolescher Ausdruck w

eine Interpretation als Schaltfunktion

etwa die Länge als Kostenfunktion

Gesucht: ein kürzester Ausdruck mit gleicher
Interpretation

**Für dieses Problem gibt es bis heute
(außer erschöpfenden Verfahren)**

keine Lösungsmethoden!!

Bemerkungen

Es ist entscheidend, wie man Ausdrücke interpretiert. Offenbar gibt es boolesche Algebren in denen Rechenregeln gelten, die nicht aus den Axiomen ableitbar sind.

Eine boolesche Algebra, in der alle Beziehungen aus den Axiomen ableitbar sind heißt **frei**. Frei ist z.B. die Algebra der totalen Schaltfunktionen.

Meist interpretieren wir die Variablen x_i als Projektionen auf die i -te Komponente über der Menge der Schaltfunktionen, d.h. $x_i(p) = p_i$. In diesem Falle werden wir nichts weiter dazu sagen, und nennen diese Interpretation auch **Standardinterpretation**.

Zusätzlich benutzen wir aber oft weitere Namen bzw. Variablen, die andere Schaltfunktionen bezeichnen. Formal kann man deren Interpretation auch dadurch beschreiben, dass man sie durch einen Ausdruck ersetzt:

Substitution

Definition

Sei $w \in BA(Y)$ und seien $a_1, \dots, a_k \in BA(Y)$

Dann sei $w(y_1 = a_1, \dots, y_k = a_k)$ der Ausdruck, der durch Ersetzung aller Vorkommen von y_i durch a_i entsteht.

Wir nennen diesen Vorgang auch **Substitution**.

Beispiel:

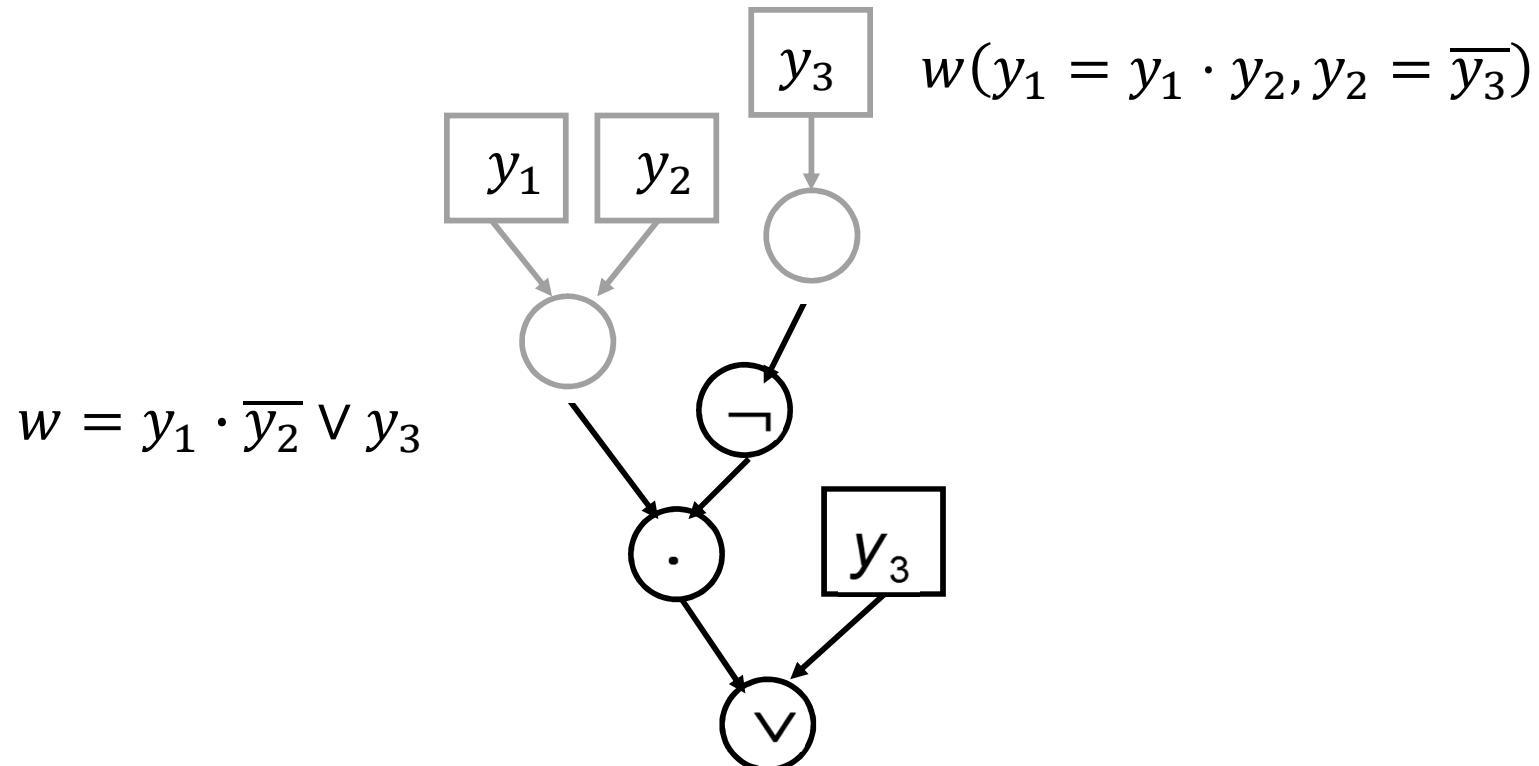
$$w = y_1 \cdot \overline{y_2} \vee y_3$$

$$w(y_1 = y_1 \cdot y_2, y_2 = \overline{y_3}) = (y_1 \cdot y_2) \overline{(\overline{y_3})} \vee y_3$$

Substitution ff

Auf Syntaxbäumen ist die Substitution sehr anschaulich:

Man ersetze einfach die entsprechenden Blätter durch Syntaxbäume für die substituierten Ausdrücke:



Substitution ff

Bemerkungen:

- Eine Substitution aller Variablen mit 0,1 entspricht einer Auswertung der dargestellten Funktion auf einem Punkt.
- Substitutionen lassen sich rekursiv auf Teilausdrücken vornehmen ($uv(a=w) = u(a=w)v(a=w)$).
- Substitutionen sind reihenfolgenabhängig, wenn der substituierte Ausdruck Variablen enthält, die ebenfalls substituiert werden:

aber

$$\begin{aligned}(y_1 \cdot \overline{y_2} \vee y_3)(y_1 = y_1 \cdot y_3)(y_3 = y_2) \\= ((y_1 y_3) \cdot \overline{y_2} \vee y_3)(y_3 = y_2) \\= ((y_1 y_3) \cdot \overline{y_2} \vee y_2) = y_2\end{aligned}$$

$$\begin{aligned}(y_1 \cdot \overline{y_2} \vee y_3)(y_3 = y_2)(y_1 = y_1 \cdot y_3) \\= (y_1 \cdot \overline{y_2} \vee y_2)(y_1 = y_1 \cdot y_3) \\= ((y_1 y_3) \cdot \overline{y_2} \vee y_2)\end{aligned}$$

Kofaktoren

Definition

Sei $f \in S_n$ eine Schaltfunktion. Dann heißen die Funktionen

$$f_{x_i} \text{ mit } f_{x_i}(x_1, \dots, x_n) := f(x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_n)$$

$$f_{\bar{x}_i} \text{ mit } f_{\bar{x}_i}(x_1, \dots, x_n) := f(x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_n)$$

Kofaktoren von f nach x_i bzw. \bar{x}_i

Offenbar hängen die Kofaktoren einer Funktion nicht mehr vom Wert der i -ten Komponente ab, da diese grundsätzlich als 1 oder 0 betrachtet wird. Damit hat man eine Methode, ein Problem für eine Funktion, die von n Variablen abhängt, auf Funktionen, die nur noch von $n-1$ Variablen abhängen, zurückzuführen.

Kofaktoren

Definition

Sei $f \in S_n$ eine Schaltfunktion. Wir sagen

f ist unabhängig von $x_i : \Leftrightarrow f_{x_i} = f_{\bar{x}_i}$

Definition

Sei $f \in S_n$ eine Schaltfunktion. Wir nennen

$$\frac{\partial f}{\partial x_i} := f_{x_i} \oplus f_{\bar{x}_i}$$

die **boolesche Differenz** von f nach x_i

Beobachtung: f ist offenbar unabhängig von x_i

$$:\Leftrightarrow f_{x_i} = f_{\bar{x}_i} \Leftrightarrow (f_{x_i} \equiv f_{\bar{x}_i}) = 1 \Leftrightarrow \overline{(f_{x_i} \equiv f_{\bar{x}_i})} = 0 \Leftrightarrow f_{x_i} \oplus f_{\bar{x}_i} = 0 \Leftrightarrow \frac{\partial f}{\partial x_i} = 0$$

Support

Definition

Sei $f \in S_n$ eine Schaltfunktion. Wir nennen die Menge der Variablen, von denen f abhängt

$$\text{supp}(f) := \{x_i \mid f_{x_i} \neq f_{\bar{x}_i}\}$$

den **Support** von f .

Beobachtungen:

Interpretiert man einen booleschen Ausdruck als Schaltfunktion, so kann diese nur von Variablen abhängen, die in dem Ausdruck vorkommen.

Darstellungen von Kofaktoren der dargestellten Funktion erhält man ferner ganz leicht durch Substitution der entsprechenden Variablen mit den entsprechenden Konstanten.

Support auf Ausdrücken

Definition

Sei $w \in BA(X)$ ein boolescher Ausdruck. Wir nennen die Menge der Variablen, die in w vorkommen, den (syntaktischen) **Support** von w , $ssupp(w)$

Lemma

Sei $w \in BA(X)$ und sei I die Standardinterpretation. Dann gilt:

- (a) $I(w(x_i = 1)) = I(w)_{x_i}, I(w(x_i = 0)) = I(w)_{\bar{x}_i}$
- (b) $ssupp(w) \supseteq supp(I(w))$

Support auf Ausdrücken ff

Beweis:

$$\begin{aligned}(a): I(w(x_i = 1)(x_1, \dots, x_n)) &= I(w)(x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_n) \\ &= I(w)_{x_i}(x_1, \dots, x_n)\end{aligned}$$

analog folgt $I(w(x_i = 0)) = I(w)_{\bar{x}_i}$

(b): Ist $x_i \notin ssupp(w)$, ist schon $w(x_i = 1) = w = w(x_i = 0)$

also ist $I(w)_{x_i} = I(w(x_i = 1)) = I(w) = I(w)_{\bar{x}_i}$

und damit $x_i \notin supp(I(w))$

Vorsicht: Es gilt nicht $ssupp(w) = supp(I(w))$

Denn: $ssupp(x_1 \vee x_1 x_2) = \{x_1, x_2\}$

Aber: $supp(I(x_1 \vee x_1 x_2)) = supp(I(x_1)) = \{x_1\}$

Eigenschaften der Kofaktoroperation

Lemma

Sei $f \in S_n$. Dann gilt:

(a) Für $\varepsilon, \delta \in \{0, 1\}$ ist $x_i^\varepsilon x_j^\delta = \begin{cases} 1 & i = j \text{ und } \delta = \varepsilon \\ 0 & i = j \text{ und } \delta \neq \varepsilon \\ x_i^\varepsilon & \text{sonst} \end{cases}$

(b) $(f_{x_i^\varepsilon})_{x_j^\delta} =: f_{x_i^\varepsilon x_j^\delta} = f_{x_j^\delta x_i^\varepsilon} := (f_{x_j^\delta})_{x_i^\varepsilon}$

(c) $(fg)_{x_i^\varepsilon} = f_{x_i^\varepsilon} g_{x_i^\varepsilon}; (f \vee g)_{x_i^\varepsilon} = f_{x_i^\varepsilon} \vee g_{x_i^\varepsilon}; \overline{f_{x_i^\varepsilon}} = \bar{f}_{x_i^\varepsilon};$

Beweis: Sei I die Standardinterpretation und w eine Darstellung von f .

Eigenschaften ff

$$\begin{aligned}
 \text{(a)}: x_i^\varepsilon |_{x_j^\delta} &= I(x_i^\varepsilon) |_{x_j^\delta} = I\left(x_i^\varepsilon (x_j = \delta)\right) \\
 &= \begin{cases} I(\delta^\varepsilon) & i = j \\ I(x_i^\varepsilon) & \text{sonst} \end{cases} \\
 &\quad \begin{cases} 1 & \text{falls } i = j \text{ und } \delta = \varepsilon \\ 0 & \text{falls } i = j \text{ und } \delta \neq \varepsilon \\ x_i^\varepsilon & \text{sonst} \end{cases} \quad \left(\begin{matrix} 0^0 = \bar{0} = 1 = 1^1 \\ 1^0 = \bar{1} = 0 = 0^1 \end{matrix} \right)
 \end{aligned}$$

$$\begin{aligned}
 \text{(b)}: (f_{x_i^\varepsilon}) |_{x_j^\delta} &= (I(w) |_{x_i^\varepsilon}) |_{x_j^\delta} = I(w(x_i = \varepsilon)) |_{x_j^\delta} \\
 &= I\left(w(x_i = \varepsilon)(x_j = \delta)\right) = I\left(w(x_i = \varepsilon, x_j = \delta)\right) \\
 &= I\left(w(x_j = \delta, x_i = \varepsilon)\right) = I(w(x_i = \delta)) |_{x_i^\varepsilon} \\
 &= (I(w) |_{x_j^\delta}) |_{x_i^\varepsilon} = (f_{x_j^\delta}) |_{x_i^\varepsilon}
 \end{aligned}$$

Eigenschaften ff

(c): Seien v, w Ausdrücke, mit $I(v) = f, I(w) = g$, dann ist

$$\begin{aligned}(f \cdot g)_{x_i^\varepsilon} &= (I(v) \cdot I(w))_{x_i^\varepsilon} = I((v \cdot w)(x_i = \varepsilon)) \\&= I(v(x_i = \varepsilon)) \cdot w(x_i = \varepsilon)) \\&= I(v(x_i = \varepsilon)) \cdot I(w(x_i = \varepsilon)) = f_{x_i^\varepsilon} \cdot g_{x_i^\varepsilon}\end{aligned}$$

Analog die anderen Operationen.

Satz

Jede Schaltfunktion $f \in S_n$ kann dargestellt werden durch
die **Shannon Expansion**

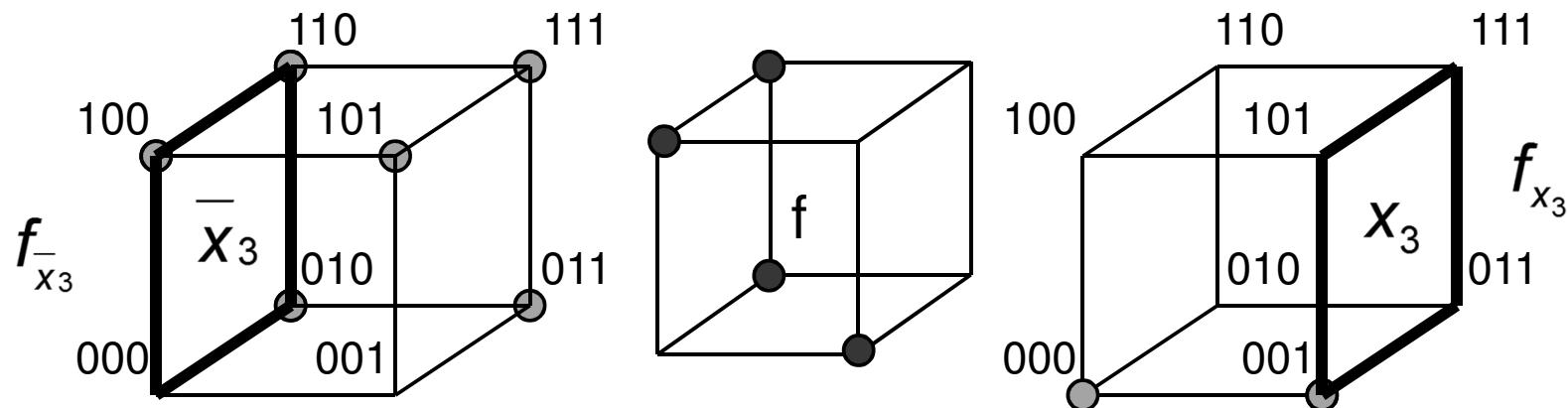
$$f = \bar{x}_i \cdot f_{\bar{x}_i} \vee x_i \cdot f_{x_i}$$

Eigenschaften ff

Beweis:

$$\begin{aligned}(\bar{x}_i f_{\bar{x}_i} \vee x_i f_{x_i})(q_1, \dots, q_n) &= \bar{x}_i f_{\bar{x}_i}(q_1, \dots, q_n) \vee x_i f_{x_i}(q_1, \dots, q_n) \\&= \begin{cases} f_{\bar{x}_i}(q_1, \dots, q_n) \vee 0 & \text{falls } q_i = 0 \\ 0 \vee f_{x_i}(q_1, \dots, q_n) & \text{falls } q_i = 1 \end{cases} \\&= f(q_1, \dots, q_n)\end{aligned}$$

Anschaung:



Boolesche Ausdrücke in WüHDL

In WüHDL (und auch VHDL) gibt es für die vordefinierten Typen BOOLEAN und BIT die binären Operatoren

AND, OR, NAND, NOR, XOR, XNOR

mit gleicher Präzedenz. D.h. es gibt nicht wie in unseren Formeln eine stärkere Bindung von AND gegenüber OR!

Lediglich der Operator NOT hat eine höhere Präzedenz.

Die Auswertung erfolgt von links nach rechts. Entscheidet der linke Operand des Operators den Wert des Ausdrucks, dann wird der rechte Operand nicht ausgewertet.

Schaltfunktionen in WüHDL

Man kann nun in WüHDL auf verschiedene Art und Weise Schaltfunktionen spezifizieren, die von speziellen Werkzeugen (Synthesewerkzeuge) in Schaltungen umgesetzt werden können. Es hängt stark vom Synthesewerkzeug ab, welche Konstrukte es als Spezifikation einer Schaltfunktion interpretieren kann, und wie es diese Spezifikationen letztlich in Schaltungen umsetzt.

Da eine Behandlung dieses Themas den Umfang der Vorlesung sprengen würde, wollen wir uns hier zunächst auf ein spezielles Konstrukt beschränken, das wir auch noch als "boolesche Gleichungen" behandeln werden:

concurrent signal assignments

Concurrent Signal Assignments

Man kann im Rumpf einer ARCHITECTURE statt innerhalb von Prozessen auch direkt Signalzuweisungen angeben. Zur Definition von Schaltfunktionen beschränken wir uns auf Signale vom Typ BIT.
Bedingungen:

- die zugewiesenen Signale sind entweder Ports der ENTITY vom Modus OUT oder lokale Signale der ARCHITECTURE
- zu jedem Signal gibt es höchstens eine Zuweisung
- auf der rechten Seite steht ein Ausdruck vom Typ des Signals ohne Angaben von Verzögerungen
- im Ausdruck auf der rechten Seite kommen nur lokale Signale oder Ports der ENTITY vom Modus IN vor
- lokale Signale dürfen nicht zyklisch voneinander abhängen

Concurrent Signal Assignments ff

Erläuterungen:

concurrent signal assignments treten in folgender Form auf:

BEGIN -- Architecture

s <= (a AND b) OR NOT c;

t <= (a XOR c) NAND b;

PROCESS ...

END Architecture;

Im Grunde ist jede einzelne Signalzuweisung $\text{sig} \leq \text{Ausdruck}(a, \dots, z)$ eine Kurznotation für

PROCESS (a, ..., z)

BEGIN $\text{sig} \leq \text{Ausdruck}(a, \dots, z)$; END PROCESS;

Nun zu den Bedingungen:

Concurrent Signal Assignments ff

Die zugewiesenen Signale sind Ports der ENTITY vom Modus OUT oder lokale Signale:

Andere Signale sind im Rumpf der Architecture nicht sichtbar bzw. schreibbar.

Zu jedem Signal gibt es höchstens eine Zuweisung:

Mit jedem Signal soll eine Schaltfunktion assoziiert werden können. Dies ist die Funktion, die man als Interpretation des Ausdrucks der rechten Seite erhält (Signaltyp = BIT).

Auf der rechten Seite steht ein Ausdruck vom Typ des Signals ohne Angaben von Verzögerungen:

Eine Schaltfunktion ist nicht zeitbezogen, deswegen sollten bei der Spezifikation die Verzögerungen = 0 sein. Ein Synthesewerkzeug erzeugt zu einer solchen synthetisierbaren Architecture dann einen Schaltkreis, der durchaus Verzögerungen hat. Manche Synthesewerkzeuge ignorieren auch einfach nur Verzögerungsangaben.

Concurrent Signal Assignments ff

Im Ausdruck auf der rechten Seite kommen nur lokale Signale oder Ports der ENTITY vom Modus IN vor:

Andere Signale sind im Rumpf der Architecture nicht sichtbar oder lesbar.

Lokale Signale sollten nicht zyklisch voneinander abhängen:

Dies ist ein subtiler Punkt. Ein Signal s hängt direkt ab von s' , wenn s' auf der rechten Seite einer Zuweisung an s steht. Abhängigkeit ist der transitive Abschluss der Relation "hängt direkt ab von". Ist s von sich selbst abhängig, dann hat man eine zyklische Abhängigkeit. Man kann nun s keine Schaltfunktion mehr zuweisen, sondern eine solche Abhängigkeit produziert im Allgemeinen Gedächtnis.

Beispiel:

Es ist leicht, unser Basis R/S Latch durch Assignments mit zyklischer Abhängigkeit zu beschreiben $qz \leq rz \text{ NAND } q;$

$q \leq sz \text{ NAND } qz;$

1.6 Schaltkreise

Zur Vorlesung Rechenanlagen

SS 2019

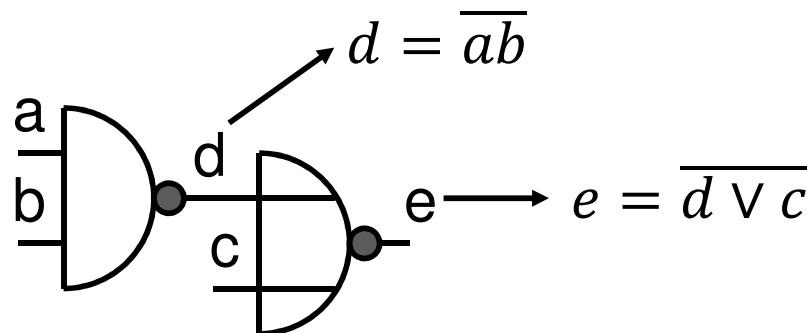


Schaltkreise

Wir können statische Aspekte des Verhaltens von digitalen Schaltkreisen mit dem nun erarbeiteten Wissen durch boolesche Gleichungen beschreiben:

- Assoziiere zu jedem Signal der Schaltung einen Namen
- Assoziiere statische Beziehungen durch Gleichungen

Beispiel:



Durch Substitution von
Namen durch Gleichungen
erhält man nun statische
Beziehungen zu anderen
Leitungen:

$$e = \overline{d} \vee c \quad (d = \overline{ab}) = \overline{\overline{ab}} \vee c = abc$$

Definitionen zu Schaltkreisen

Ein Schaltkreis C über einem Bausteinsystem A besteht aus einer Menge

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

Zu jeder Bausteininstanz g aus G sei ferner eine Menge von **Anschlüssen** (Konnektoren, Ports)

$$P(g) := \{g.a_1, \dots, g.a_k\}$$

und ein Baustein $comp(g)$ in A gegeben den g instanziert.

Gilt $comp(g) = comp(h)$ für zwei Instanzen g, h , dann müssen Namen und Anzahl der Anschlüsse jeweils gleich sein: d.h. ist $g.a$ in $P(g)$, dann ist auch $h.a$ in $P(h)$

Definitionen zu Schaltkreisen ff

Die Anschlussmengen der Bausteine bilden paarweise disjunkte Mengen, d.h. $g \neq h \Rightarrow P(g) \cap P(h) = \emptyset$

Ein Netz s aus der Netzmenge N identifizieren wir zugleich mit einer Teilmenge von Anschlüssen, die Anschlüsse die es verbindet: d.h. es ist stets $s \subseteq P(C) \cup \bigcup_{g \in G} P(g)$

Dabei liegt **jeder** Anschluss $g.x$ **auf genau einem** Netz, dem Netz zum Anschluss. Wir nennen dies $\text{net}(g.x)$.

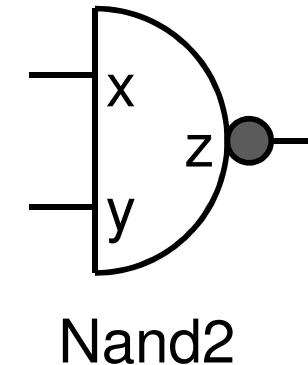
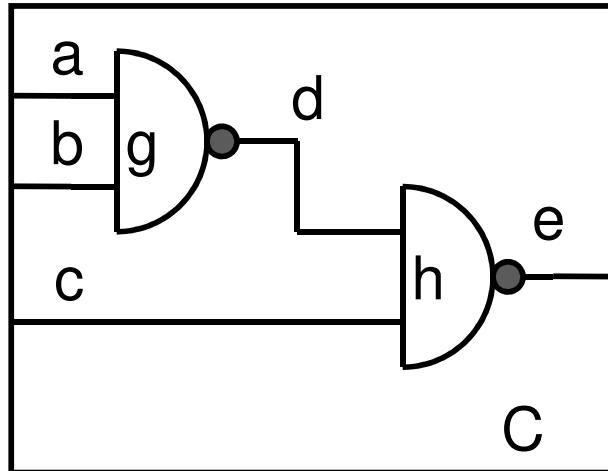
Netze bilden physikalisch die Verbindungen zwischen Bausteininstanzen und dienen zur Kommunikation. Wir benutzen daher statt des Begriffs Netz auch den Begriff Signal, wenn wir Schaltkreisen ein Verhalten zuordnen.

Schaltkreise in WüHDL

In WüHDL kann man Schaltkreise durch Instanziierung von Komponenten und das Binden ihrer Ports an Signale oder Ein-/Ausgangsports beschreiben. Das Konzept ist etwas allgemeiner, weil man mehr Komfort bei der Beschreibung von Schaltkreisen haben möchte:

1. Das Bausteinsystem ist nicht fest, sondern wird in der COMPONENT Deklaration festgelegt. Komponenten können dabei selbst wieder durch Schaltkreise modelliert sein. Dadurch erhält man eine kurze und übersichtliche Definition eines im Grunde gigantischen finalen Schaltkreises. Man nennt diese Technik auch **hierarchischen Entwurf**.
2. Prozesse und concurrent signal assignments können im Grunde auch als Bausteininstanzen aufgefasst werden, mit dem Unterschied, dass es keine Mehrfachbenutzung gibt und das Verhalten des Bausteins direkt durch Programmcode definiert wird. Wir beschränken uns bei der Theorie auf ein möglichst einfaches Schaltkreismodell, weil wir uns zunächst für spezielle Gattungen von Schaltkreisen interessieren.

Beispiel



Wir haben einen Schaltkreis C bestehend aus

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

Es ist nun etwa $P(g)=\{g.x,g.y,g.z\}$, $P(h)=\{h.x,h.y,h.z\}$
sowie z.B. $d=\{g.z,h.x\}$, $e=\{h.z,C.e\}$ oder $net(g.y)=b$

Definitionen zu Schaltkreisen ff

Meist assoziiert man mit den Anschlüssen einen Wertefluss, d.h. man unterscheidet zwischen **Eingängen** und **Ausgängen**. Bausteinsysteme mit dieser Differenzierung nennt man **orientiert**.

Wir nennen also ein Bausteinsystem *A* **orientiert**, wenn zu jeder Bausteininstanz *b* die Anschlussmenge eine disjunkte Vereinigung von *In(b)* und *Out(b)* bildet.

Damit ist $P(g) = \text{In}(g) \cup \text{Out}(g)$; $\text{In}(g) \cap \text{Out}(g) = \emptyset$

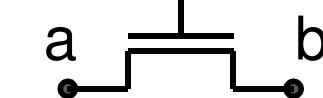
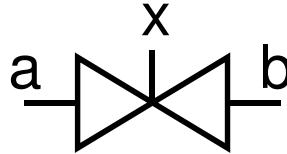
und es ist $\text{In}(g) \neq \emptyset \neq \text{Out}(g)$

Ein Schaltkreis *C* heißt **orientiert**, wenn er über einem orientierten Bausteinsystem aufgebaut ist, und seine Ports in Eingänge *In(C)* und Ausgänge *Out(C)* klassifiziert sind, d.h. $\text{In}(C) \cup \text{Out}(C) = P(C)$ und $\text{In}(C) \cap \text{Out}(C) = \emptyset$

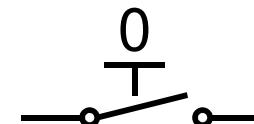
Definitionen zu Schaltkreisen ff

Bemerkung:

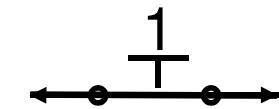
Es gibt durchaus nützliche Bausteine, die nicht orientierbar sind, wie etwa ein Schalter:



Realisierung



Verhalten
0



Orientierte Schaltkreise bilden also eine Unterklasse.
Allerdings sollte man weitere Bedingungen an ihre Struktur stellen, damit man ihnen ein Verhalten zuordnen kann:

Wir nennen einen orientierten Schaltkreis **wohlgeformt**, genau dann, wenn er folgende beiden Bedingungen (OC1) und (OC2) erfüllt:

Definitionen zu Schaltkreisen ff

$$\text{(OC1)} \forall s \in N: \# \left(s \cap (In(C) \cup \bigcup_{g \in G} Out(g)) \right) \leq 1$$

(OC1) besagt, dass höchstens ein Ausgang einer Bausteininstanz zu einem Netz s gehört. Da dieser Ausgang physikalisch gesehen den Pegel des Signals zu diesem Netz bestimmt, nennen wir diesen Ausgang, falls vorhanden, den **Treiber** von s , $treiber(s)$. Hat ein Baustein g nur einen Ausgang, g.z, so lassen wird dessen Angabe auch weg und nennen einfach g den Treiber von s .

Definitionen zu Schaltkreisen ff

(OC2) $\forall s: s \cap \bigcup_{g \in G} Out(g) = \emptyset \Rightarrow In(C) \cap s \neq \emptyset$

(OC2) besagt, dass alle Netze, die nicht von Ausgangsports getrieben werden, von außen zugänglich sein müssen (zu den Eingängen des Schaltkreises gehören). Wir nennen ein Netz s , das einen Eingang $C.x$ des Schaltkreises enthält, dann auch den Primäreingang zu $C.x$.

Wir nehmen von nun an stets an, dass unsere orientierten Schaltkreise auch wohlgeformt sind.

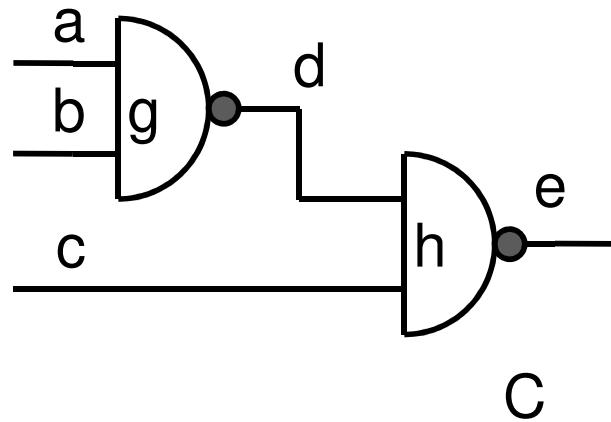
In wohlgeformten orientierten Schaltkreisen kann man nun noch weitere Begriffe betrachten:

Definitionen zu Schaltkreisen ff

Eine Folge von Netzen s_1, \dots, s_k heißt **Pfad** der Länge $k-1$

$$\Leftrightarrow \forall 1 \leq i < k \exists g_i: In(g_i) \cap s_i \neq \emptyset \neq Out(g_i) \cap s_{i+1}$$

Beispiel:



Pfade:

| |
|-----------|
| d, e |
| a, d, e |
| b, d, e |
| c, e |

Rückkopplungsfreie Schaltkreise

Ein orientierter Schaltkreis heißt **rückkopplungsfrei**

: \Leftrightarrow Für alle Pfade $s_1 \dots s_k$ gilt: $s_i = s_j \Rightarrow i=j$

Folgerung:

In einem rückkopplungsfreien Schaltkreis haben alle Pfade eine Länge kleiner gleich $\#N$.

Beweis:

Betrachte einen Pfad $s_1 \dots s_k$ mit $k > \#N$

Dann können nicht alle Netze verschieden sein!

(Schubfachprinzip!)

Also gibt es ein $i < j$ mit $s_i = s_j$ 

Rückkopplungsfreie Schaltkreise ff

Rückkopplungsfreie Schaltkreise spielen eine wichtige Rolle, weil man mit ihnen alle Schaltfunktionen realisieren kann.

Beobachtung:

Für einen rückkopplungsfreien Schaltkreis ist die Funktion

$$tiefe(s) := \begin{cases} 0 & \text{falls } s \cap In(C) \neq \emptyset \\ 1 + \max\{tiefe(r) | r \cap In(gate(treiber(s))) \neq \emptyset\} & \text{sonst} \end{cases}$$

wohldefiniert.

Beweis (*Tiefe*)

Sei s ein beliebiges Netz. Wir zeigen die Beobachtung durch Induktion nach der Länge k eines längsten Pfades

$$s_1, \dots, s_{k+1} = s$$

(Da alle Pfade Länge kleiner gleich $\#N$ haben, gibt es stets einen längsten Pfad nach s .)

$k = 0$: Dann gibt es kein Bauteil g mit $gate(treiber(s)) = g$, sonst wäre ja r, s für r aus $sig(ln(g))$ ein Pfad der Länge 1. Also ist s ein Primäreingang und damit $tiefe(s) = 0$

Beweis (*Tiefe*) ff

$k \rightarrow k+1$: Sei nun $s_1, \dots, s_{k+1}, s_{k+2} = s$ längster Pfad nach s .

Dann gilt für alle r mit $r \cap \text{In}(gate(\text{treiber}(s))) \neq \{\}$,
dass jeder Pfad $s'_1, \dots, s'_{l+1} = r$

Länge $l < k+1$ hat.

(sonst wäre ja $s'_1, \dots, s'_{l+1}, s_{k+2} = s$ länger!)

Also ist $\text{tiefe}(s) = 1 + \text{tiefe}(s_{k+1})$

$= 1 + k$ -- Induktionsannahme

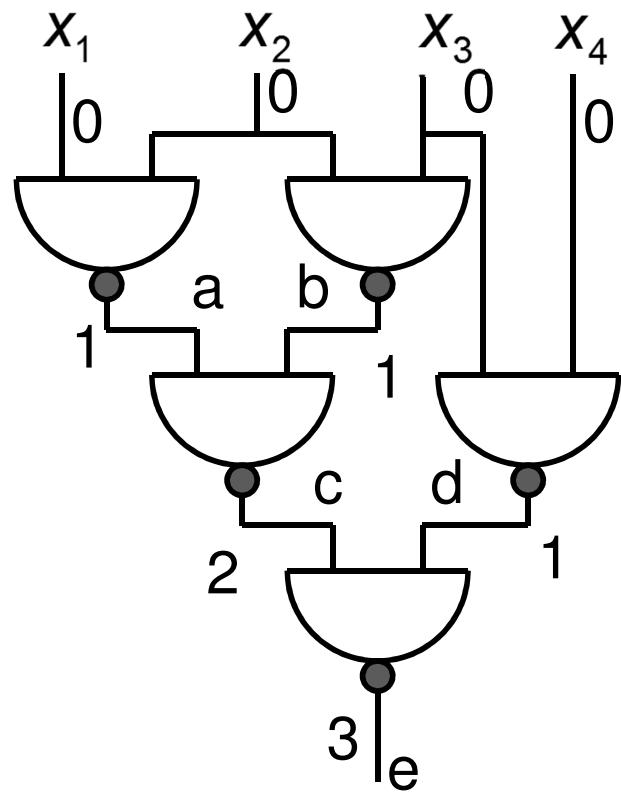
wohldefiniert.

Wir nennen

$$\max\{\text{tiefe}(s) | s \in S\} =: \text{tiefe}(C)$$

die **Tiefe eines Schaltkreises** C .

Beispiel: (Tiefe)



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

Wir wollen nun Schaltkreisen ein Verhalten zuordnen:

Statisches Verhalten

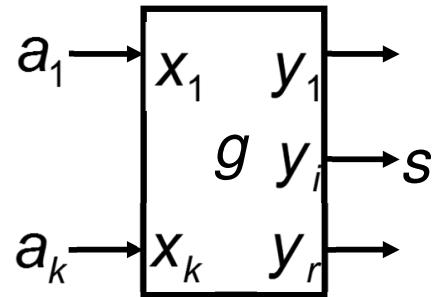
Wir nennen ein Bausteinsystem A **kombinatorisch**, wenn alle Bausteine orientierte Anschlüsse haben, und mit jedem Ausgang y , etwa durch einen booleschen Ausdruck, eine Funktion y über den Eingängen assoziiert ist.

Beispiel: wir assoziieren bei einem NAND2 mit Eingängen a und b und Ausgang y die Funktion

$$y = \overline{ab}$$

Sei A ein kombinatorisches Bausteinsystem, und sei C ein wohlgeformter, orientierter Schaltkreis über A . Dann können wir zu jedem Netz s , das von einem Gatter getrieben wird, eine Funktion über den Eingängen der Bausteininstanz g betrachten, dessen Ausgang s treibt:

Statisches Verhalten ff



Zum Bausteintyp g und Ausgang y_i sei $y_i = y_i(x_1, \dots, x_k)$ die assoziierte Funktion.

Sei ferner für alle $i=1, \dots, k$:

$$a_i = \text{net}(g.x_i) \text{ sowie } \text{treiber}(s) = g.y_i$$

Dann ordnen wir dem Netz s seine **lokale Funktion**

$$C[s] = C[s](a_1, \dots, a_k) := y_i(\dots, x_i = \text{net}(g.x_i), \dots)$$

zu.

Mit einem Primäreingang u zu $C.x$ assoziieren wir $C[u]=x$

Statisches Verhalten

Sei C ein Schaltkreis über einem kombinatorischen Bausteinsystem A . Wir können **Signalbelegungen** des Schaltkreises als $\#N$ -stellige Bitvektoren auffassen. Eine Signalbelegung nennen wir **stabil**, wenn die Belegung jedes Signals (=Netz) der Auswertung der lokalen Funktion des Signals auf den Belegungen ihrer Argumente entspricht.

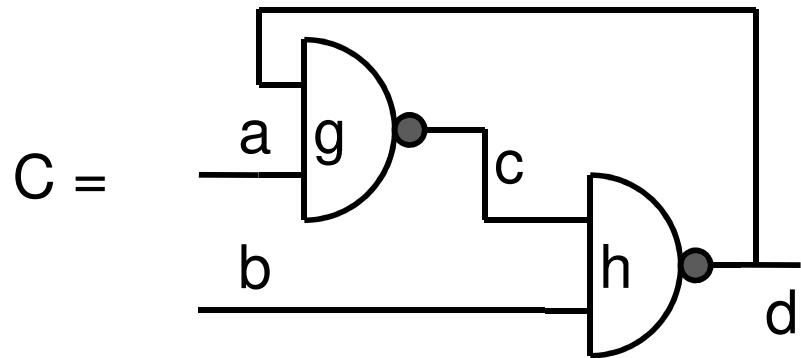
Demnach können wir jedem orientierten, wohlgeformten Schaltkreis C über kombinatorischen Bausteinen eine Funktion

$$\chi_C := \prod_{s \in N} (s \equiv C[s])$$

zuordnen, die **Charakteristische Funktion** seines **statischen Verhaltens**.

Sie wird auf einer Signalbelegung genau dann 1, wenn der Schaltkreis mit dieser Belegung stabil ist.

Beispiel:



$$\chi_C = (c \equiv \overline{ad})(d \equiv \overline{bc})$$

$$(a,b,c,d) = (0,1,1,0):$$

Stabil, da

$$\begin{aligned} \chi_C(0,1,1,0) &= (1 \equiv \overline{00})(0 \equiv \overline{11}) \\ &= (1 \equiv 1)(0 \equiv 0) \\ &= 1 \cdot 1 = 1 \end{aligned}$$

$$(a,b,c,d) = (1,0,1,0):$$

Instabil, da

$$\begin{aligned} \chi_C(1,0,1,0) &= (1 \equiv \overline{10})(0 \equiv \overline{01}) \\ &= (1 \equiv 1)(0 \equiv 1) \\ &= 1 \cdot 0 = 0 \end{aligned}$$

$$(a,b,c,d) = (1,1,\bar{y},y):$$

Stabil für jedes y , da

$$\begin{aligned} \chi_C(1,1,\bar{y},y) &= (\bar{y} \equiv \overline{1y})(y \equiv \overline{1\bar{y}}) \\ &= (\bar{y} \equiv \bar{y})(y \equiv y) \\ &= 1 \cdot 1 = 1 \end{aligned}$$

2. Hardwareentwurf -- die Disziplinen

Zur Vorlesung Rechenanlagen

SS 2019



2.1 Entwurf kombinatorischer Module

**Zur Vorlesung Rechenanlagen
SS 2019**



2.1.1 Kombinatorische Schaltkreise

Definition

Ein Schaltkreis C heißt **kombinatorisch**, genau dann, wenn

- die Bausteine in C kombinatorisch sind,
- C wohlgeformt und
- C rückkopplungsfrei ist.

Diese Untermenge der Menge aller Schaltkreise ist sehr wichtig, weil man durch sie schon alle Schaltfunktionen realisieren und ihr Verhalten sehr leicht getrennt unter rein statischen, funktionellen Aspekten und rein zeitlichen Aspekten analysieren kann (wenn auch nicht sehr exakt!).

Kombinatorische Schaltkreise ff

Definition

Sei C ein kombinatorischer Schaltkreis. Dann ordnen wir jedem Signal s , dessen lokale Funktion sich aus den Signalen s_{i_1}, \dots, s_{i_k} bestimmt, eine Schaltfunktion $F[s]$ über den Primäreingängen,

$$F[s] := C[s](s_{i_1} = F[s_{i_1}], \dots, s_{i_k} = F[s_{i_k}])$$

die **globale Funktion** von s zu, wobei für einen Primäreingang s zu $C.x$ $F[s] := x$ gilt.

Wir können damit jedes Signal in einem kombinatorischen Schaltkreis auch als Funktion über den Eingängen auffassen. Ferner sortieren wir von nun an die $m = \#N$ Signale s_1, \dots, s_m stets **topologisch**, d.h. s_1, \dots, s_n sind Primäreingänge und sonst ist $\text{tiefe}(s_i) \leq \text{tiefe}(s_j)$ für $i < j$

Kombinatorische Schaltkreise ff

Satz

Sei C ein kombinatorischer Schaltkreis. Dann gibt es zu jeder Belegung p_1, \dots, p_n der Primäreingänge genau eine stabile Belegung p_1, \dots, p_m mit

$$p_i = F[s_i](p_1, \dots, p_n)$$

Beweis:

Wir zeigen durch Induktion nach der Tiefe, dass für jedes Signal s_i in einer stabilen Belegung p gilt:

$$p_i = F[s_i](p_1, \dots, p_n)$$

tiefe(s_i) = 0: Dann ist s_i Primäreingang und damit

$$F[s_i](p_1, \dots, p_n) = x_i(p_1, \dots, p_n) = p_i$$

Beweis ff

$tiefe(s_i) > 0$: Sei für $k < tiefe(s_i)$ die Behauptung schon gezeigt. Dann ist p stabil, nur wenn

$$1 = (p_i \equiv C[s_i](p_{i_1}, \dots, p_{i_k}))$$

\Leftrightarrow

$$p_i = C[s_i](s_{i_1} = p_{i_1}, \dots, s_{i_k} = p_{i_k})$$

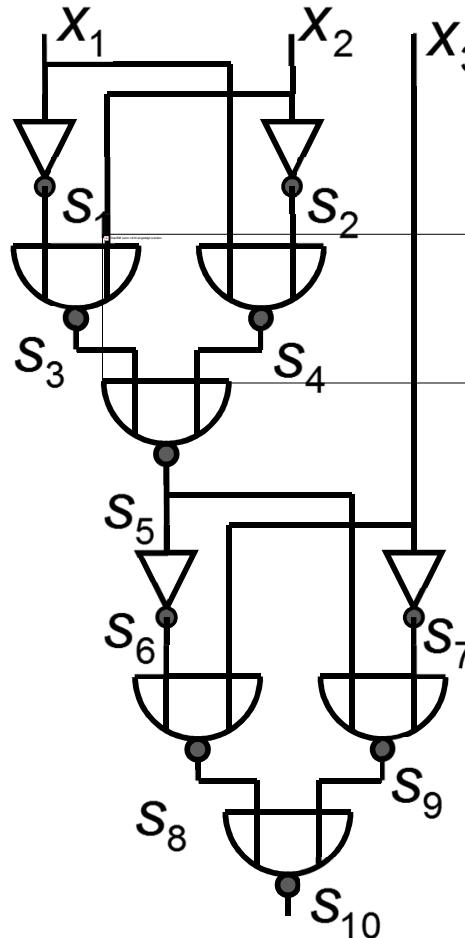
und mit I.A. $p_{i_j} = F[s_{i_j}](p_1, \dots, p_n)$

\Leftrightarrow

$$\begin{aligned} p_i &= C[s_i](s_{i_1} = F[s_{i_1}](p_1, \dots, p_n), \dots, s_{i_k} = F[s_{i_k}](p_1, \dots, p_n)) \\ &= F[s_i](p_1, \dots, p_n) \end{aligned}$$

Beispiel:

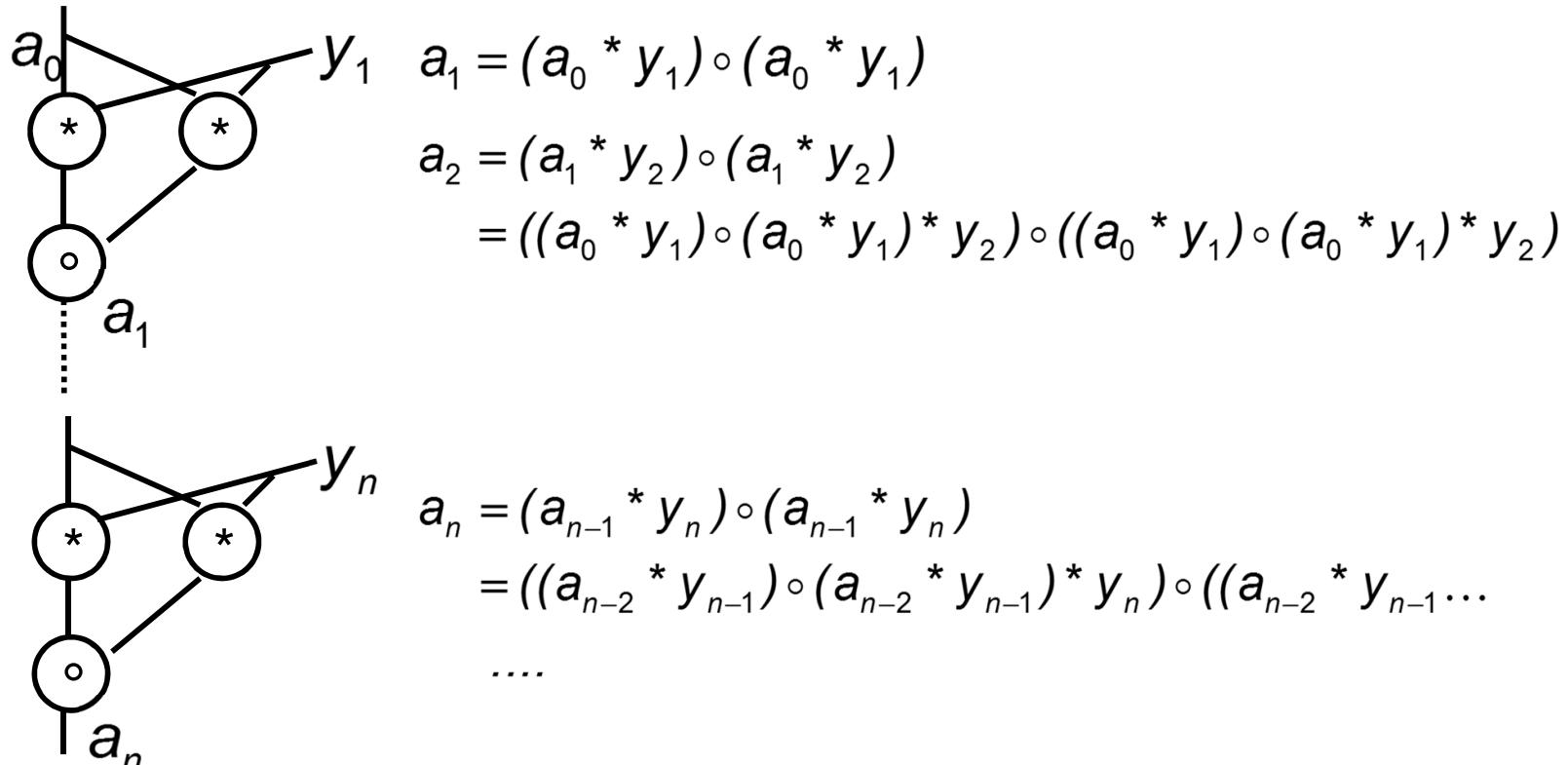
Man kann also jedem Signal durch sukzessive Substitution eine Funktion über den Primäreingängen zuordnen:



$$\begin{aligned} F[s_1] &= C[s_1](x_1 = F[x_1]) = \overline{x_1}, \quad F[s_2] = \overline{x_2} \\ F[s_3] &= C[s_3](s_1 = F[s_1], x_2 = F[x_2]) \\ &= \overline{F[s_1]} \vee F[\overline{x_2}] = \overline{\overline{x_1}} \vee x_2 \\ F[s_4] &= \overline{x_1 \vee \overline{x_2}} \\ F[s_5] &= \overline{\overline{\overline{x_1 \vee x_2}} \vee \overline{x_1 \vee \overline{x_2}}} \\ F[s_6] &= \overline{\overline{\overline{x_1 \vee x_2}} \vee x_1 \vee \overline{x_2}}, \quad F[s_7] = \overline{x_3} \\ F[s_8] &= \overline{\overline{\overline{x_1 \vee x_2}} \vee \overline{x_1 \vee \overline{x_2}} \vee x_3} \\ F[s_9] &= \overline{\overline{\overline{x_1 \vee x_2}} \vee x_1 \vee \overline{x_2} \vee \overline{x_3}} \end{aligned}$$

Beispiel ff:

Vereinfacht man die Ausdrücke nicht, dann können sie enorm wachsen: Wann immer ein Signal s zur Berechnung eines Signals t mehrfach benutzt wird, tritt der Ausdruck für $F[s]$ zunächst mehrfach im Ausdruck für $F[t]$ auf:



Beispiel ff:

$$\begin{aligned} \text{D.h. } |a_n| &= 2|a_{n-1}| + 9 = \dots = 2^n|a_0| + 9 \sum_{i=0}^{n-1} 2^i \\ &= 2^n + 9 \cdot (2^n - 1) = 5 \cdot 2^{n+1} - 9 \end{aligned}$$

Der Ausdruck wächst exponentiell in der Größe der Schaltung!

Selbst wenn man im Beispiel $F[s_{10}]$ vereinfacht, erhält man

$$\begin{aligned} F[s_{10}] &= \overline{\overline{\overline{x_1 \vee x_2}} \vee \overline{\overline{x_1 \vee \overline{x_2}}} \vee x_3} \vee \overline{\overline{\overline{x_1 \vee x_2}} \vee \overline{x_1 \vee \overline{x_2}}} \vee \overline{x_3} \\ &\stackrel{\text{(De Morgan)}}{=} (\overline{\overline{x_1 \vee x_2}} \vee \overline{x_1 \vee \overline{x_2}} \vee x_3) \cdot (\overline{\overline{\overline{x_1 \vee x_2}} \vee \overline{x_1 \vee \overline{x_2}}} \vee \overline{x_3}) \\ &\stackrel{\text{(De Morgan)}}{=} (x_1 \overline{x_2} \vee \overline{x_1} x_2 \vee x_3) \cdot ((\overline{x_1} \vee x_2)(x_1 \vee \overline{x_2}) \vee \overline{x_3}) \\ &\stackrel{\text{(ausmult.)}}{=} (x_1 \overline{x_2} \vee \overline{x_1} x_2 \vee x_3) \cdot (\overline{x_1} \overline{x_2} \vee x_1 x_2 \vee \overline{x_3}) \\ &= \overline{x_3}(x_1 \overline{x_2} \vee \overline{x_1} x_2) \vee x_3(\overline{x_1} \overline{x_2} \vee x_1 x_2) \end{aligned}$$

Zusammenfassung:

Wir wissen nun

Satz

Jede Schaltfunktion $f \in \mathcal{S}_n$ ist durch einen booleschen Ausdruck über x_1, \dots, x_n darstellbar, wobei x_1, \dots, x_n die Projektionen in \mathcal{S}_n sind.

Satz

Jede Schaltfunktion $f \in \mathcal{S}_n$ ist eindeutig darstellbar durch ihre disjunktive Normalform.

$$f = \bigvee_{\substack{p \in \mathbb{B}^n \\ f(p)=1}} x_1^{p_1} \cdots x_n^{p_n}$$

2.1.2 Konstruierbarkeit

Andererseits wissen wir auch, dass jedes Signal in einem kombinatorischen Schaltkreis eine Schaltfunktion über den Primäreingängen realisiert. Es gilt sogar:

Satz

Ist A ein kombinatorisches Bausteinsystem, und gibt es kombinatorische Schaltkreise $Cand$, $Cnot$, bzw. Cor , $Cnot$ über A mit zwei (einem) Eingängen a, b (a) und einem Ausgang y , so dass

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

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

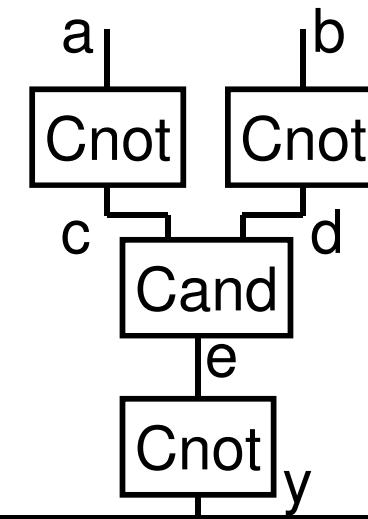
Beweis

Sei f eine beliebige Schaltfunktion. Dann gibt es einen booleschen Ausdruck, der f darstellt (schlimmstenfalls die DNF). Sei w ein solcher Ausdruck, und $T(w)$ ein Syntaxbaum dazu.

Da $Cand$ und $Cnot$ oder Cor und $Cnot$ schon existieren, muss nur noch die ggf. fehlende boolesche Operation realisiert werden. Dies ist aber nach der de Morgan'schen Regel sehr einfach:

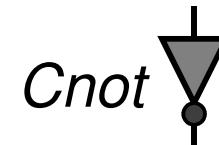
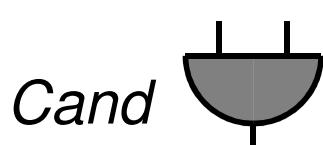
$$F[y] = \overline{F[e]} = \overline{F[c]F[d]} = \overline{\overline{a}\overline{b}} = a \vee b$$

Also liefert diese Schaltung eine Schaltung mit der Eigenschaft für Cor . (Analog erhält man im anderen Falle $Cand$)



Beweis ff

Wir können also ohne Einschränkung annehmen, dass wir zu jeder booleschen Operation einen Schaltkreis über A haben, und nehmen folgende Symbole als Kürzel dafür:



Nun transformieren wir den Syntaxbaum $T(w)$ zu f wie folgt in einen kombinatorischen Schaltkreis:

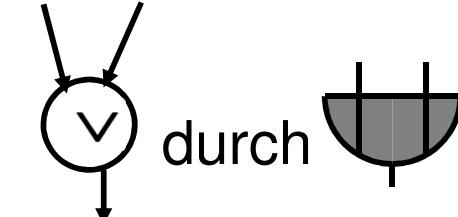
1. Ordne allen Knoten der Markierung x_i das Signal x_i zu.
2. Ersetze alle Knoten der Markierung



durch



durch



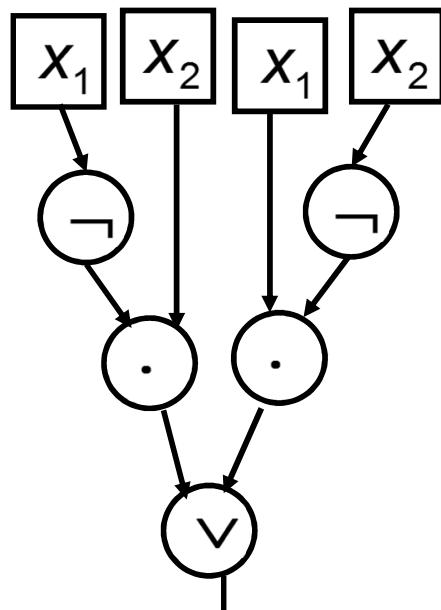
durch

unter Beibehaltung der Verbindungen.

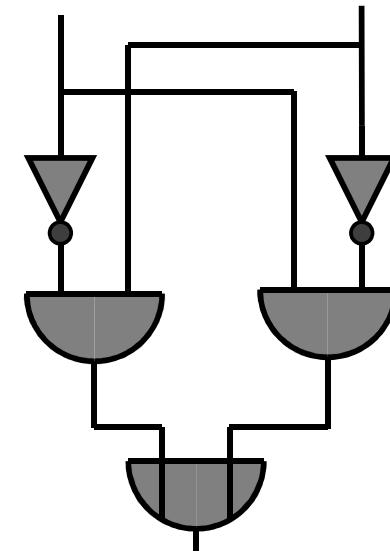
Beweis ff und Beispiel:

Dann gilt für den Ausgang y des so entstandenen Schaltkreises $F[y] = \mathbf{I}(w) = f$

Beispiel: $w = \bar{x}_1x_2 \vee x_1\bar{x}_2$



Syntaxbaum



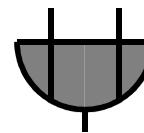
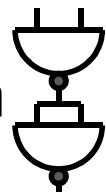
Transformation

Beispiel:

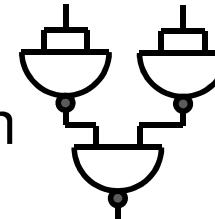
Wir nehmen nun an, dass unser Bausteinsystem nur aus einem *NAND2* Baustein besteht. Dann können wir natürlich *Cand*, *Cor*, *Cnot* realisieren:



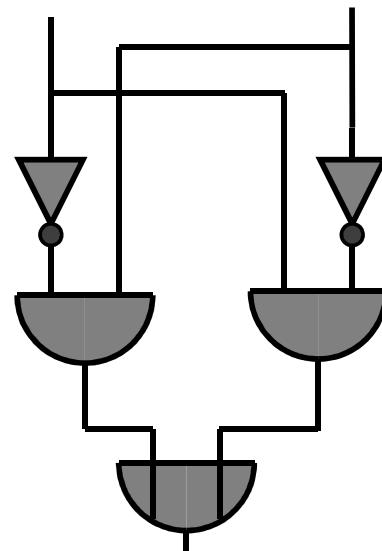
durch



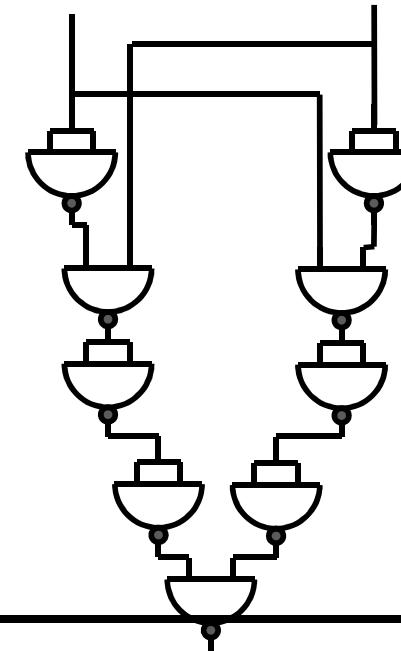
durch



durch



liefert dann



Beispiel:

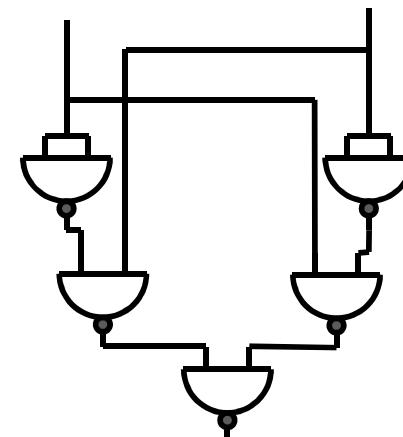
Man erhält also eine Schaltung aus **9 Gattern!?**

Diese Realisierung ist „lausig“.

Natürlich kann man nun Vereinfachungen vornehmen, die die Funktion am Ausgang nicht ändern, wie etwa das Entfernen von Doppelnegationen:

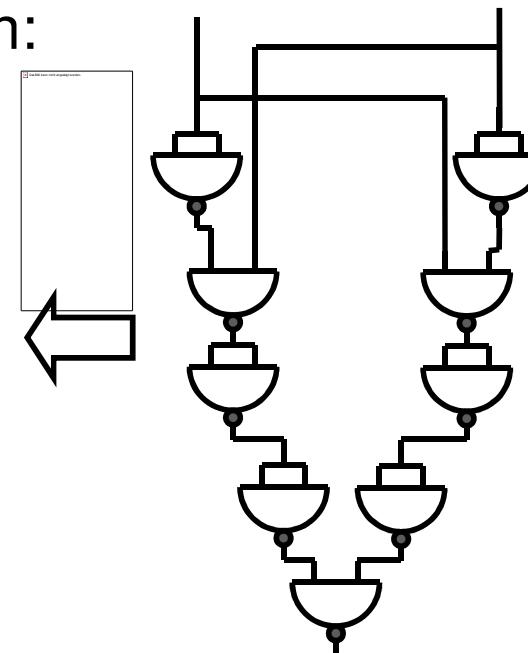


=



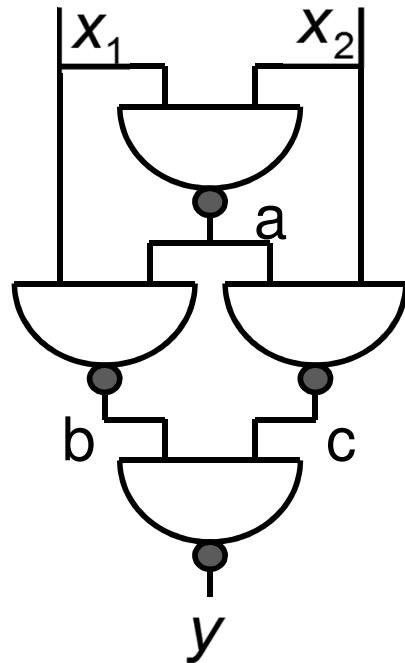
Und erhält

5 Gatter



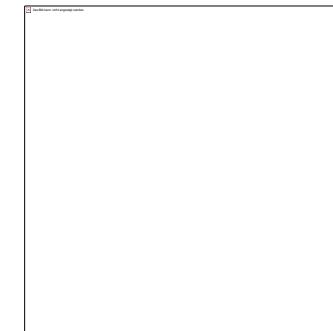
Beispiel:

Betrachte



$$\begin{aligned}F[y] &= \overline{\overline{F[b]}F[c]} = \overline{\overline{x_1}\overline{F[a]}} \cdot \overline{\overline{x_2}\overline{F[a]}} \\&= \overline{x_1}\overline{x_2} \cdot \overline{x_2}\overline{x_1} = \overline{x_1}\overline{x_2} \vee \overline{x_2}\overline{x_1} \\&= x_1(\overline{x}_1 \vee \overline{x}_2) \vee x_2(\overline{x}_1 \vee \overline{x}_2) \\&= x_1\overline{x}_2 \vee \overline{x}_1x_2\end{aligned}$$

Es geht sogar mit 4 Gattern!



Bemerkungen

Dieses Beispiel verdeutlicht:

- 1.** Wir müssen Schaltfunktionen durch Schaltkreise, nicht durch Ausdrücke, realisieren.
- 2.** Jedem Ausdruck kann man aber einen Schaltkreis zuordnen, dessen Gatterzahl proportional zur Länge des Ausdrucks ist.
- 3.** Es gibt allerdings Schaltkreise mit sehr viel weniger Gattern als die Länge des kürzesten Ausdrucks, der die gleiche Funktion darstellt.
- 4.** Um gute Lösungen zu finden, muss man die Eigenarten der Technologie, d.h. des Bausteinsystems, ausnutzen können.

Bemerkungen

Ein Kernproblem des Hardwareentwurfs ist also:

Syntheseproblem für kombinatorische Schaltkreise

Gegeben: Eine Schaltfunktion in irgendeiner Darstellung (Ausdruck, Schaltkreis, ...) und die Beschreibung einer Technologie in Form eines kombinatorischen Bausteinsystems A .

Gesucht: Ein kombinatorischer Schaltkreis C , der die Funktion berechnet unter minimaler Gatterzahl (minimaler Laufzeit) bei gegebener Laufzeit (Gatterzahl), falls dieser existiert.

Für dieses Problem gibt es bis heute kein exaktes Lösungsverfahren, das nicht alle Schaltungen erschöpfend aufzählt!

2.1.3 Disjunktive Formen

Für den Sonderfall kombinatorischer Schaltkreise zu Ausdrücken in disjunktiver Form gibt es Minimierungsverfahren, die auf **Quine und Mc Cluskey** zurückgehen. Solche und weitergehende Minimierungsverfahren für mehrstufige kombinatorische Schaltkreise sind in modernen Synthesewerkzeugen implementiert. Ihre Schilderung würde den Rahmen der Vorlesung sprengen. Wir wollen aber am Beispiel disjunktiver Formen zeigen, welche Aufgaben solche Synthesewerkzeuge lösen.

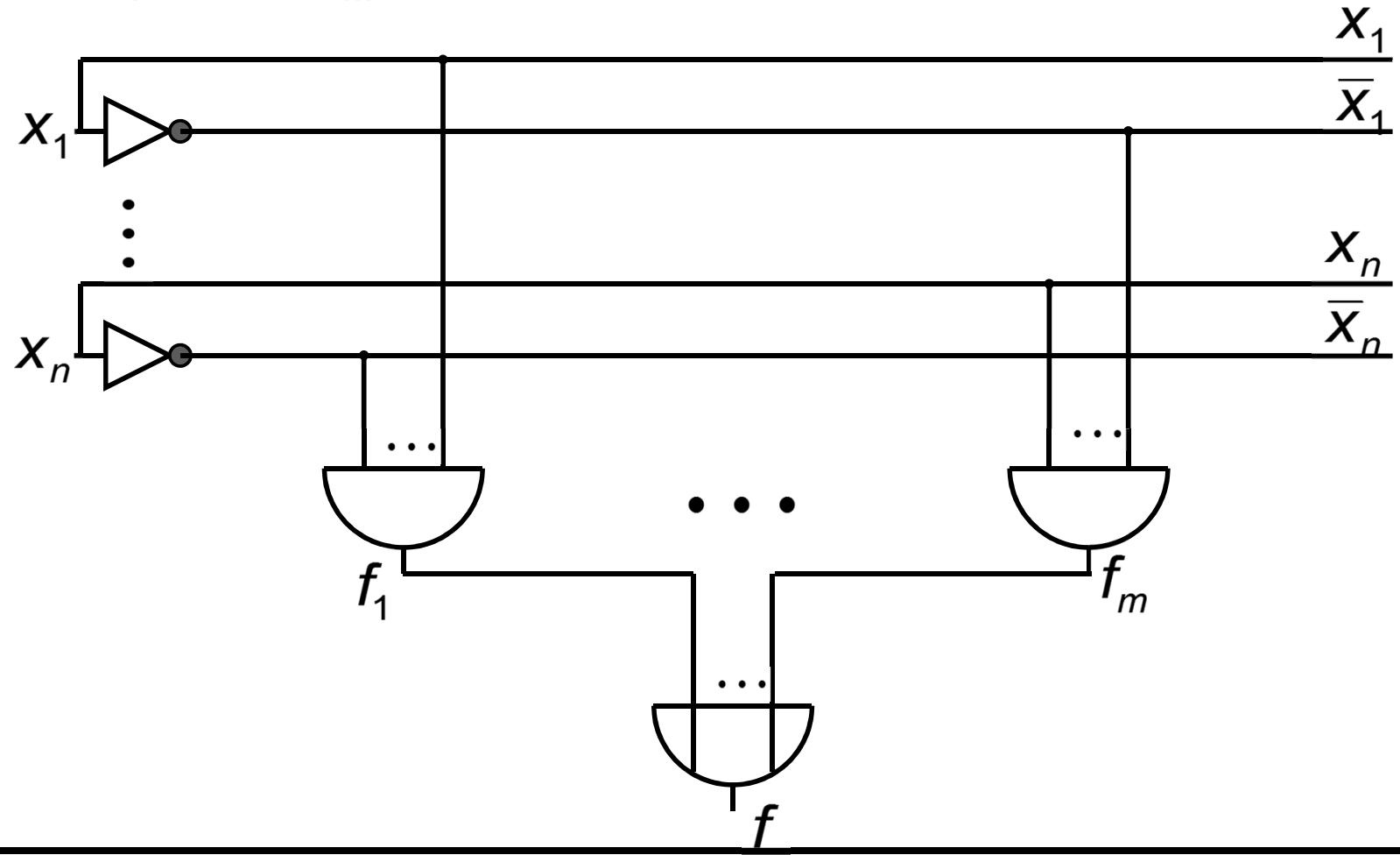
Gegeben sei eine disjunktive Form $f = f_1 \vee \dots \vee f_m$

wobei die f_i Produkte über den Variablen x_1, \dots, x_n sind.

Dann kann man einer solchen disjunktiven Form f einen kanonischen Schaltkreis $C(f)$ zuordnen, der eine Und-Stufe und eine Oder-Stufe hat. Wir nennen $C(f)$ daher auch einen **zweistufigen Schaltkreis**.

Zweistufiger Schaltkreis zur DF

$$f = f_1 \vee \dots \vee f_m$$



Zweistufige Schaltkreise

Man realisiert also einfach

- alle Literale, die in Produkten von f vorkommen, dann
- alle Produkte von f durch Und-Gatter, und dann
- f selbst durch eine m -stelliges Oder-Gatter.

Ähnlich kann man für eine Funktion f in k Ausgängen vorgehen:

Stelle f dar als

$$f_1 = f_{11} \vee \dots \vee f_{1m_1}$$

$$\vdots \qquad \vdots$$

$$f_k = f_{k1} \vee \dots \vee f_{km_k}$$

Realisiere dann die benötigten Literale, dann alle Produkte f_{ij} und schließlich alle f_i

Wir nennen solche Schaltkreise „**zweistufig**“.

Beispiel:

Aufgabe: Realisiere eine Schaltung zur Umkodierung von 3 Bit Zahlen im Binärcode in den **Graycode**:

Anm.: Der Graycode ist eine Zahlendarstellung bei der sich die Darstellung von i und $i+1$ nur um ein Bit unterscheiden.

Wir definieren die Funktion durch folgende Tabelle:

| x_1 | x_2 | x_3 | $f_1(x)$ | $f_2(x)$ | $f_3(x)$ |
|-------|-------|-------|----------|----------|----------|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

Ansatz: Realisiere jede Komponente in DNF

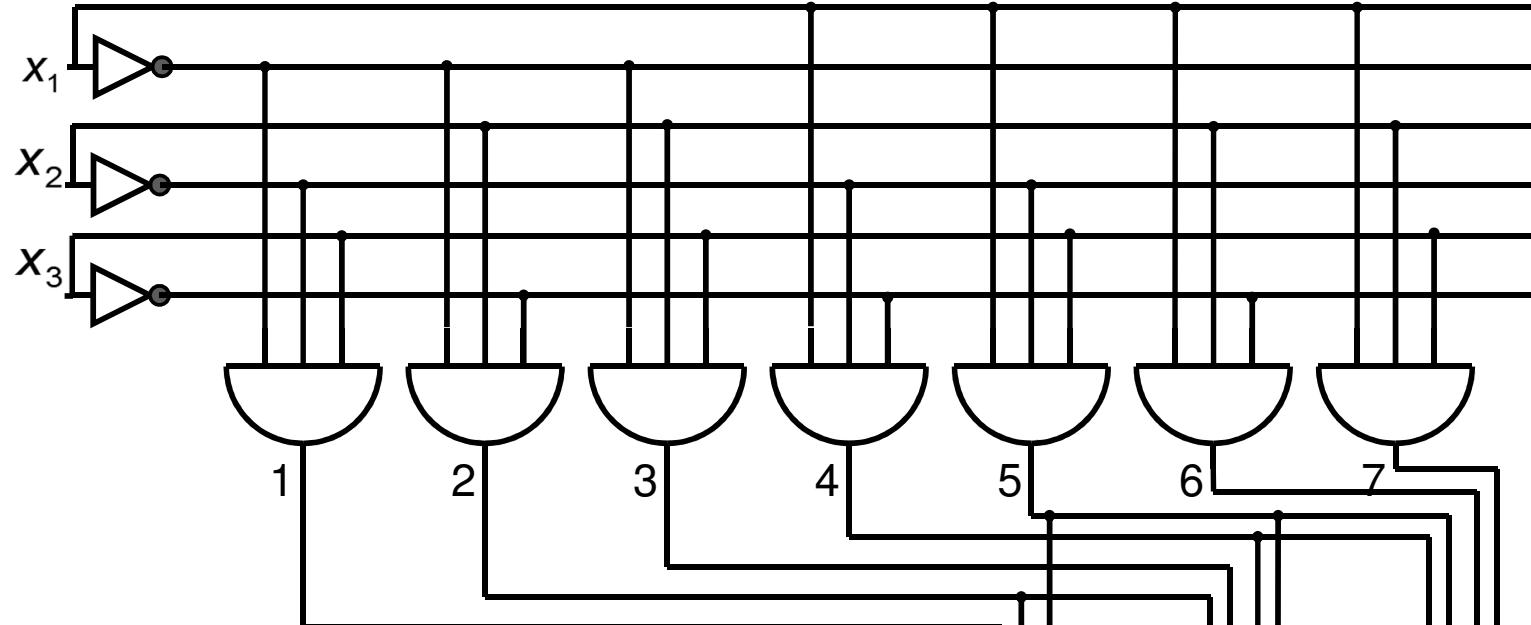
$$f_1 = \underbrace{x_1 \bar{x}_2 \bar{x}_3}_{4} \vee \underbrace{x_1 \bar{x}_2 x_3}_{5} \vee \underbrace{x_1 x_2 \bar{x}_3}_{6} \vee \underbrace{x_1 x_2 x_3}_{7}$$

$$f_2 = \underbrace{\bar{x}_1 x_2 \bar{x}_3}_{2} \vee \underbrace{\bar{x}_1 x_2 x_3}_{3} \vee \underbrace{x_1 \bar{x}_2 \bar{x}_3}_{4} \vee \underbrace{x_1 \bar{x}_2 x_3}_{5}$$

$$f_3 = \underbrace{\bar{x}_1 \bar{x}_2 x_3}_{1} \vee \underbrace{\bar{x}_1 x_2 \bar{x}_3}_{2} \vee \underbrace{x_1 \bar{x}_2 x_3}_{5} \vee \underbrace{x_1 x_2 \bar{x}_3}_{6}$$

Beispiel:

Dann liefert diese Darstellung folgende Realisierung:



$$f_1 = \underbrace{x_1 \bar{x}_2 \bar{x}_3}_{4} \vee \underbrace{\bar{x}_1 \bar{x}_2 x_3}_{5} \vee \underbrace{x_1 x_2 \bar{x}_3}_{6} \vee \underbrace{\bar{x}_1 x_2 x_3}_{7}$$

$$f_2 = \underbrace{\bar{x}_1 x_2 \bar{x}_3}_{2} \vee \underbrace{\bar{x}_1 x_2 x_3}_{3} \vee \underbrace{x_1 \bar{x}_2 \bar{x}_3}_{4} \vee \underbrace{x_1 \bar{x}_2 x_3}_{5}$$

$$f_3 = \underbrace{\bar{x}_1 \bar{x}_2 x_3}_{1} \vee \underbrace{\bar{x}_1 x_2 \bar{x}_3}_{2} \vee \underbrace{x_1 \bar{x}_2 x_3}_{5} \vee \underbrace{x_1 x_2 \bar{x}_3}_{6}$$

f_3 f_2 f_1

Kosten: 7 AND3 + 3 OR4

Beispiel ff:

Die Realisierung kostet uns also 7 AND3 plus 3 OR4. Das sind, zählt man einmal nur die benötigten Gattereingänge insgesamt: $7 \cdot 3 + 3 \cdot 4 = 33$

Es gibt aber auch andere disjunktive Darstellungen, z.B.:

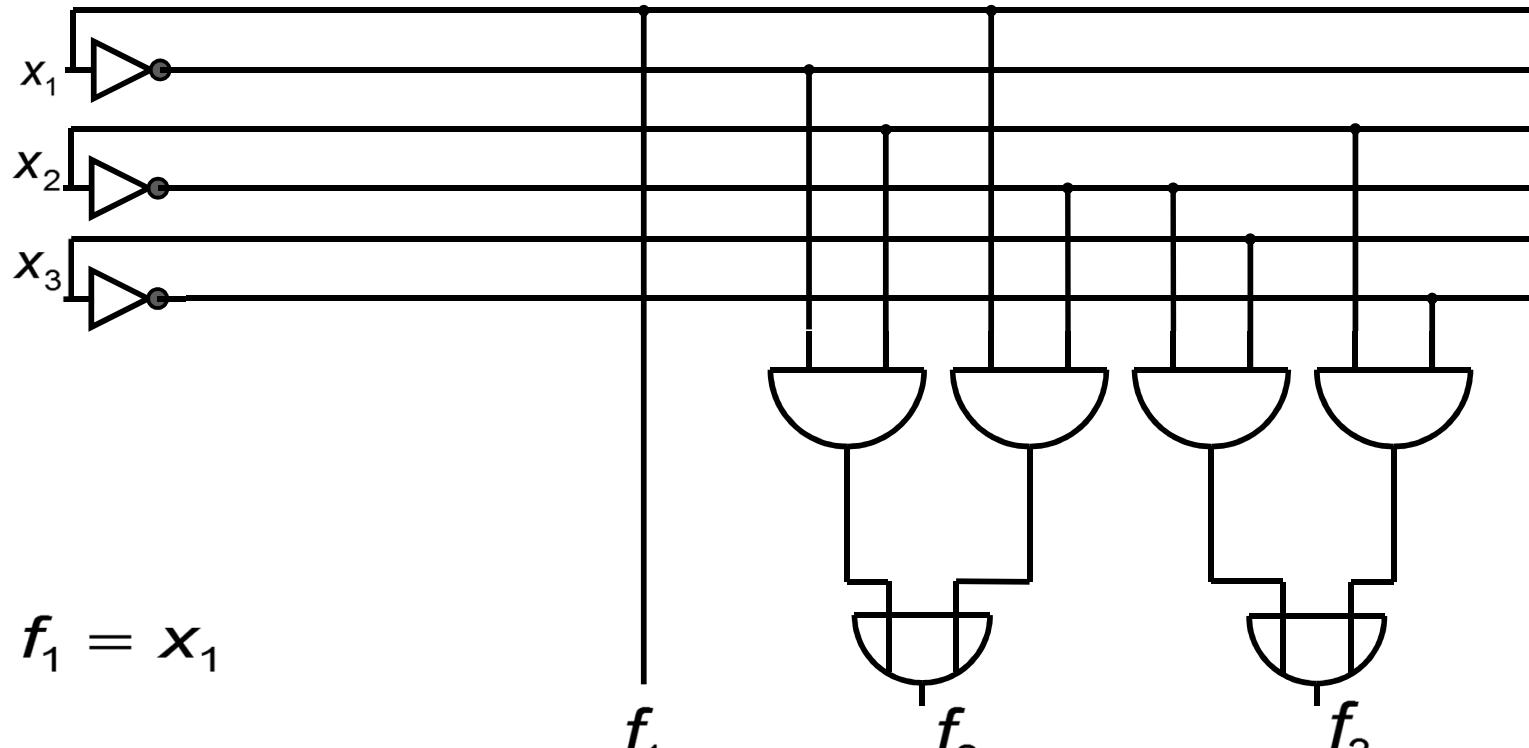
$$\begin{aligned}f_1 &= x_1 \bar{x}_2 \bar{x}_3 \vee x_1 \bar{x}_2 x_3 \vee x_1 x_2 \bar{x}_3 \vee x_1 x_2 x_3 \\&= x_1 \bar{x}_2 (\bar{x}_3 \vee x_3) \vee x_1 x_2 (\bar{x}_3 \vee x_3) \\&= x_1 \bar{x}_2 \vee x_1 x_2 = x_1 (\bar{x}_2 \vee x_2) = \boxed{x_1}\end{aligned}$$

$$\begin{aligned}f_2 &= \bar{x}_1 x_2 \bar{x}_3 \vee \bar{x}_1 x_2 x_3 \vee x_1 \bar{x}_2 \bar{x}_3 \vee x_1 \bar{x}_2 x_3 \\&= \bar{x}_1 x_2 (\bar{x}_3 \vee x_3) \vee x_1 \bar{x}_2 (\bar{x}_3 \vee x_3) = \boxed{\bar{x}_1 x_2 \vee x_1 \bar{x}_2}\end{aligned}$$

$$\begin{aligned}f_3 &= \bar{x}_1 \bar{x}_2 x_3 \vee \bar{x}_1 x_2 \bar{x}_3 \vee x_1 \bar{x}_2 x_3 \vee x_1 x_2 \bar{x}_3 \\&= \bar{x}_1 (\bar{x}_2 x_3 \vee x_2 \bar{x}_3) \vee x_1 (\bar{x}_2 x_3 \vee x_2 \bar{x}_3) = \boxed{\bar{x}_2 x_3 \vee x_2 \bar{x}_3}\end{aligned}$$

Beispiel:

Diese Vereinfachung liefert folgende Realisierung:



$$f_1 = x_1$$

$$f_2 = \bar{x}_1x_2 \vee x_1\bar{x}_2$$

$$f_3 = \bar{x}_2x_3 \vee x_2\bar{x}_3$$

Kosten: 4 AND2 + 2 OR2

12 statt 33

Syntheseproblem für 2stufige Schaltungen

Wir haben an diesem Beispiel gesehen, dass man Freiheitsgrade bei der disjunktiven Darstellung geschickt ausnutzen kann, um bessere Schaltungen zu erhalten. Wir haben es also mit folgendem Optimierungsproblem zu tun:

Gegeben: Eine Schaltfunktion $f \in \mathcal{S}_{n,k}^D$

Gesucht: Eine disjuktive Form zu f mit minimalen Kosten.

Wir können dieses Problem im Rahmen der Vorlesung nicht vollständig behandeln, wollen aber skizzieren, wie man es lösen kann.

Instanzen dieses Problems treten auf dem Weg von einer WüHDL Beschreibung zu einem Schaltkreis hin nicht selten auf:

- Minimiere die Formel der rechten Seite einer Signalzuweisung.
- Erzeuge zu einer Funktion von Bitvektoren nach Bitvektoren eine DF.

Redundanz

Satz

Sei \mathcal{I}_D die Interpretation über \mathcal{S}_n^D . Sei ferner r ein Ausdruck mit $ON(\mathcal{I}(r)) = \mathcal{B}^n \setminus D$ und g, h boolesche Ausdrücke über $X = \{x_1, \dots, x_n\}$ unter Standardinterpretation \mathcal{I} , dann gilt

$$g \equiv_{\mathcal{I}_D} h \Leftrightarrow g \vee r \equiv_{\mathcal{I}} h \vee r$$

Definition

Wir nennen eine(n Ausdruck) Funktion r , deren ON-Set genau aus den Elementen außerhalb des Definitionsbereichs D einer partiellen Funktion f besteht auch einfach (Darstellung der) **Redundanz** von f .

Beweis

$$g \vee r \equiv_{\mathcal{I}} h \vee r \Leftrightarrow \forall p \in \mathcal{B}^n : \mathcal{I}(g \vee r)(p) = \mathcal{I}(h \vee r)(p)$$

$$\Leftrightarrow \forall p \in \mathcal{B}^n : \mathcal{I}(g)(p) \vee \mathcal{I}(r)(p) = \mathcal{I}(h)(p) \vee \mathcal{I}(r)(p)$$

$$\Leftrightarrow \forall p \in \mathcal{B}^n : \left\{ \begin{array}{ll} 1=1 & p \notin D \\ \mathcal{I}(g)(p) = \mathcal{I}(h)(p) & \text{sonst} \end{array} \right\}$$

$$\Leftrightarrow \forall p \in D : \mathcal{I}(g)(p) = \mathcal{I}(h)(p)$$

$$\Leftrightarrow g \equiv_{\mathcal{I}_D} h$$

Neuformulierung des Problems:

Wir können nun das Problem neu über disjunktiven Formen unter Standardinterpretation formulieren:

Minimierung zweistufiger Schaltungen

Gegeben: disjunktive Formen r, f_1, \dots, f_k

Gesucht: disjunktive Formen $g = (g_1, \dots, g_k)$ mit

- (i) $g_i \vee r = f_i \vee r$ für alle $1 \leq i \leq k$
- (ii) $\text{cost}(g) = \min\{\text{cost}(h) \mid h \text{ erfüllt (i)}\}$

Dabei sei $\text{cost}(g) = (m, l)$, wobei m die Zahl der Produkte in g und l die Summe über die Zahl der Literale aller Produkte ist. Die Ordnung sei die lexikographische Ordnung auf Tupeln.

Implikanten

Wir beschränken uns hier auf das Problem für $k=1$, d.h.

$$f \in \mathbf{S}_n^D$$

Gegeben: disjunktive Formen r, f

Gesucht: g minimaler Kosten, mit $g \vee r = f \vee r$

Definition

Ein Produkt p heißt **Implikant** einer Funktion $f \in \mathbf{S}_n^D$ genau dann, wenn

$$p \cdot (f \vee r) = p \quad (\Leftrightarrow p \leq f \vee r \Leftrightarrow p \vee f \vee r = f \vee r)$$

Implikanten ff

Lemma

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 .

Beweis:

$$\begin{aligned} g_i \cdot (f \vee r) &= g_i \cdot (g \vee r) \\ &= g_i \cdot g \vee g_i \cdot r \\ &= g_i \cdot g_i \vee g_i \cdot \left(\bigvee_{j \neq i} g_j \right) \vee g_i \cdot r \\ &= g_i \vee g_i \cdot \left(\bigvee_{j \neq i} g_j \vee r \right) \\ &= g_i \end{aligned}$$

Primimplikanten

Wir brauchen also nur DF's zu untersuchen, die ausschließlich aus Implikanten der Funktion bestehen.

Dies kann man sogar weiter einschränken:

Definition

Ein Implikant p einer partiellen Funktion f heißt **Primimplikant** einer Funktion f genau dann, wenn es keinen Implikanten $p' \neq p$ von f gibt, mit

$$p \cdot p' = p \quad (\Leftrightarrow p \leq p' \Leftrightarrow p \vee p' = p')$$

Beispiel:

Wir betrachten wieder unsere DNF-Lösung des Umkodierers: Es war

$$f_3 = \bar{x}_1 \bar{x}_2 x_3 \vee \bar{x}_1 x_2 \bar{x}_3 \vee x_1 \bar{x}_2 x_3 \vee x_1 x_2 \bar{x}_3$$

Die Minterme sind natürlich Implikanten. Aber auch $\bar{x}_2 x_3$

denn

$$\begin{aligned}\bar{x}_2 x_3 \cdot f_3 &= \bar{x}_1 \bar{x}_2 x_3 \vee 0 \vee x_1 \bar{x}_2 x_3 \vee 0 \\ &= (\bar{x}_1 \vee x_1) \bar{x}_2 x_3 \\ &= \bar{x}_2 x_3\end{aligned}$$

Ist dieser Implikant prim?

Dazu müssen wir alle Produkte $p \neq \bar{x}_2 x_3$ mit $p \bar{x}_2 x_3 = \bar{x}_2 x_3$ auf ihre Implikanteneigenschaft hin untersuchen. In Frage kommen dazu nur: $p \in \{\bar{x}_2, x_3, 1\}$

Die 1 entfällt, da die Funktion nicht konstant 1 ist.

Beispiel ff



Ferner ist

$$\begin{aligned}\bar{x}_2 \cdot f_3 &= \bar{x}_1 \bar{x}_2 x_3 \vee 0 \vee x_1 \bar{x}_2 x_3 \vee 0 \\ &= (\bar{x}_1 \vee x_1) \bar{x}_2 x_3 = \bar{x}_2 x_3\end{aligned}$$

Also \bar{x}_2 kein Implikant!

Ferner ist

$$\begin{aligned}x_3 \cdot f_3 &= \bar{x}_1 \bar{x}_2 x_3 \vee 0 \vee x_1 \bar{x}_2 x_3 \vee 0 \\ &= (\bar{x}_1 \vee x_1) \bar{x}_2 x_3 \\ &= \bar{x}_2 x_3\end{aligned}$$

Also x_3 kein Implikant!

Damit ist $\bar{x}_2 x_3$ Primimplikant von f_3

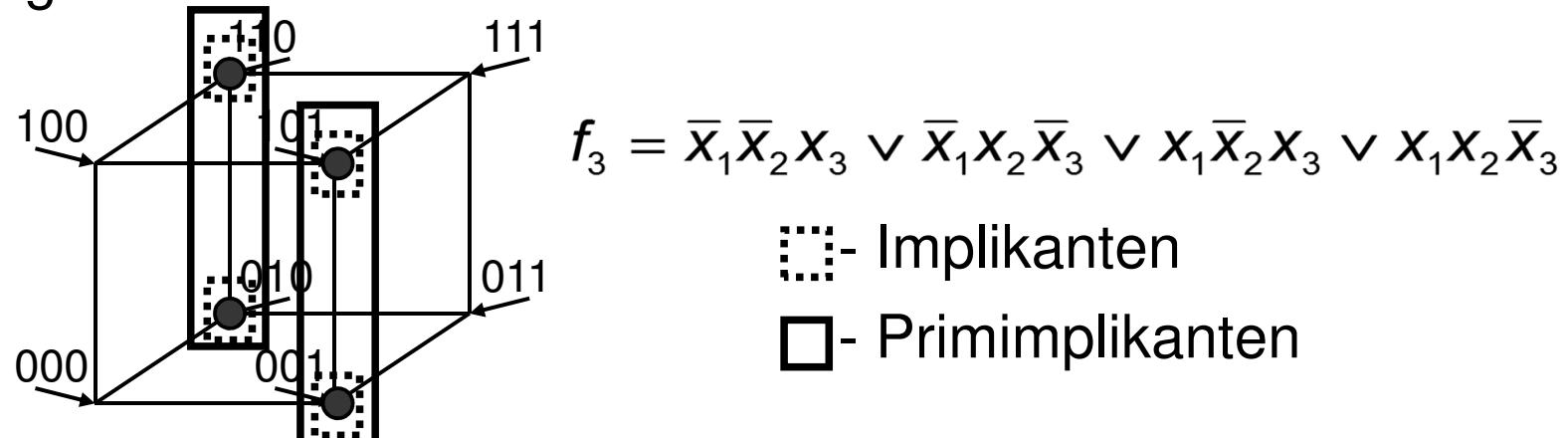
Bemerkung: Für zwei Produkte p und q gilt $p \cdot q = q$ nur dann, wenn $\text{Lit}(p) \subseteq \text{Lit}(q)$, d.h. „größere“(im Verband) Produkte sind Teilprodukte (Teiler) von „kleineren“. Daher der Begriff „prim“.

Geometrische Anschauung:

Wir hatten uns überlegt, dass der ON-Set eines Produktes stets einen Unterwürfel eines n -dimensionalen Einheitswürfels bildet.

Ein Implikant einer Funktion ist demnach ein Unterwürfel, dessen Ecken nur Punkte aus dem ON-Set und der Redundanz der Funktion enthält.

Ein Primimplikant ist ein maximaler Unterwürfel mit dieser Eigenschaft:



Das Primimplikantentheorem

Satz (Primimplikantentheorem)

Sei g eine disjunktive Form für $f \in \mathcal{S}_n^D$ unter Redundanz r mit minimalen Kosten. Dann ist jedes Produkt g_i von g ein Primimplikant.

Beweis: indirekt

Annahme: g sei kostenminimal, enthalte aber ein Produkt p , das nicht prim ist, d.h. g hat die Form: $g = p \vee h$

Dann ist $\text{cost}(g) = \text{cost}(p) + \text{cost}(h) = (1, \# \text{Lit}(p)) + \text{cost}(h)$

Da p nicht prim ist, gibt es einen Implikanten q , mit

$q \neq p$ und $p \cdot q = p$, d.h. $\text{Lit}(q) \subset \text{Lit}(p)$

Beweis ff

Demnach ist $\text{cost}(q \vee h) = \text{cost}(h) + (1, \# \text{Lit}(q))$

$< \text{cost}(h) + (1, \# \text{Lit}(p)) = \text{cost}(g)$

da $\text{Lit}(q) \subset \text{Lit}(p)$

Andererseits ist, da auch q Implikant von f , aber

$$\begin{aligned} f \vee r &= f \vee r \vee q \\ &= g \vee r \vee q \\ &= p \vee h \vee r \vee q \\ &= \underbrace{p \vee q}_{=q} \vee h \vee r = (q \vee h) \vee r \end{aligned}$$

Also wäre $q \vee h$ eine Darstellung von f , aber

$$\text{cost}(q \vee h) < \text{cost}(g) \not\leftarrow (g \text{ kostenminimal})$$

Beispiel:

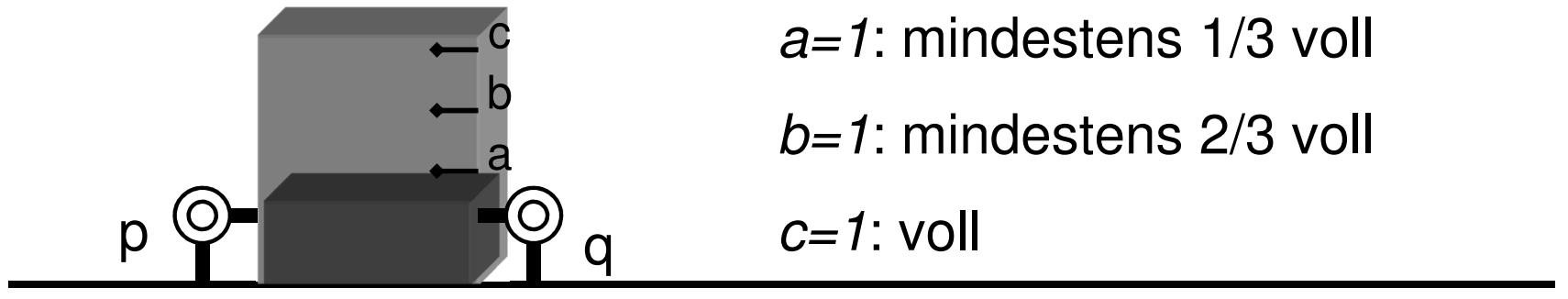
Das Primimplikantentheorem bringt uns der Sache ein gutes Stück näher:

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

Wir verdeutlichen dies an einem einfachen

Beispiel:

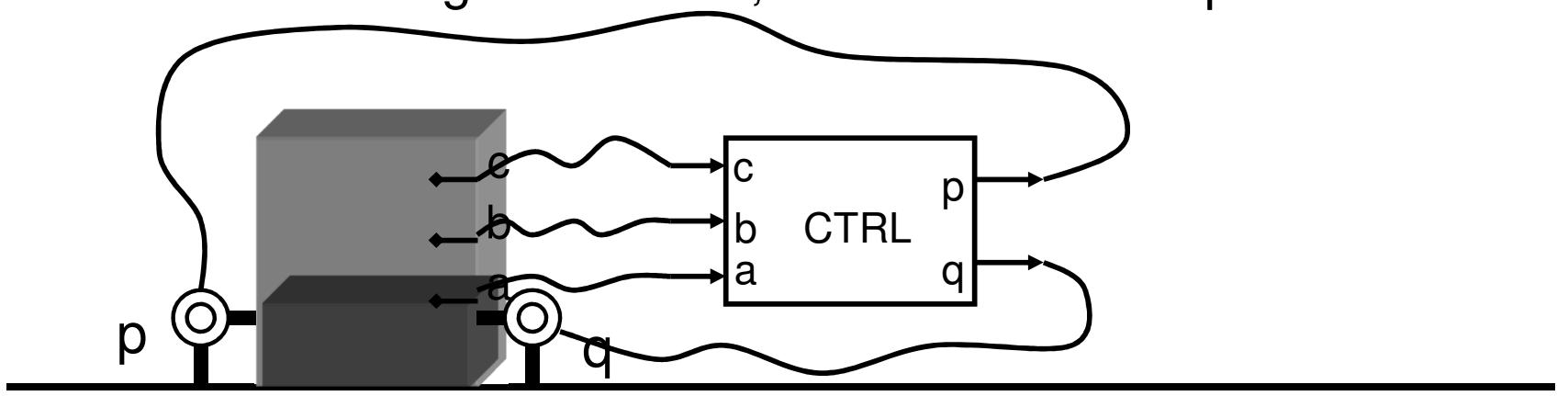
Gegeben sei ein Tank mit zwei Pumpen p,q und Sensoren a,b,c für den Füllstand:



Beispiel ff

Gesucht ist eine Schaltung *CTRL* zur Steuerung der Pumpen nach folgenden Regeln:

- ☞ Die schwächere Pumpe p ($p=1$) soll laufen, wenn der Behälter mindestens $2/3$ voll, nicht jedoch wenn er voll ist.
- ☞ Ist der Behälter weniger als $2/3$ aber mindestens $1/3$ voll, soll die stärkere Pumpe q ($q=1$) laufen.
- ☞ Sinkt der Pegel unter $1/3$, sollen beide Pumpen laufen.



Beispiel ff

Wir stellen eine Funktionstafel dazu auf:

| 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!

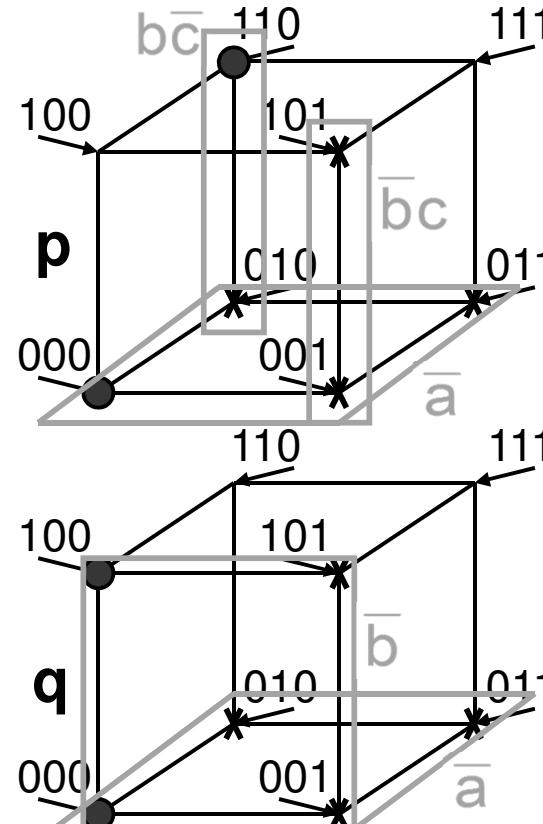
Nicht alle Eingangskombinationen kommen vor, da für die Sensorsignale einbaubedingt stets $a \geq b \geq c$ gelten muss.

Beispiel ff

Wir haben also partielle Funktionen $p, q \in S_3^D$ zu realisieren, wobei die Redundanz $r = \bar{a}\bar{b}c \vee \bar{a}b\bar{c} \vee \bar{a}bc \vee a\bar{b}\bar{c}$

| 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 :

$$\bar{a}, \bar{b}\bar{c}, \bar{b}\bar{c}$$

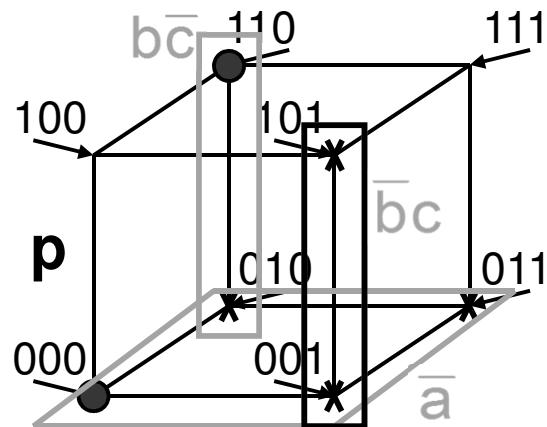
Primimplikanten
zu q :

$$\bar{a}, \bar{b}$$

Beispiel ff

Wir brauchen nun nur noch alle DF's zu betrachten, die man aus diesen Primimplikanten bilden kann, und eine billigste zu nehmen, die die Funktion realisiert.

Betrachten wir zunächst p :

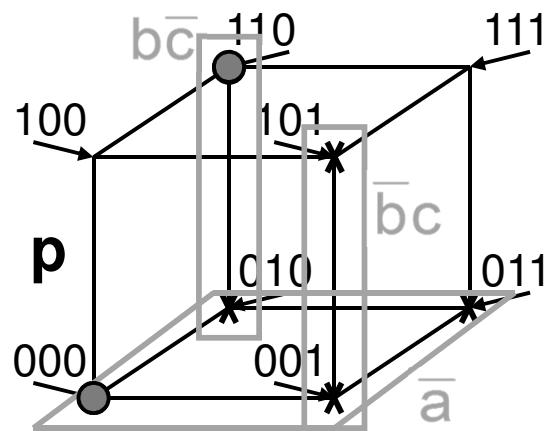


Es fällt auf, dass der Primimplikant $\bar{b}c$ nur Punkte der Redundanz überdeckt. Läßt man ihn in einer Realisierung g weg, so überdeckt

$$g \vee r = g \vee \bar{b}c \vee r$$

immer noch die gleichen Punkte. Wir nennen solche Primimplikanten **total redundant**.

Beispiel ff



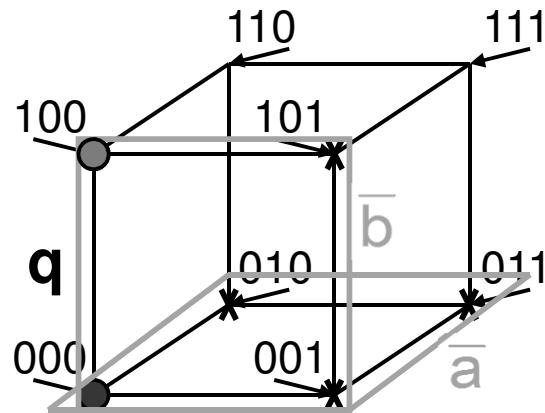
Ferner fällt auf, dass man auf \bar{a} und auch auf $b\bar{c}$ nicht verzichten kann, da sie Elemente des ON-Sets von p enthalten, die jeweils kein anderer Primimplikant enthält.

Man nennt solche Punkte auch **wesentliche Punkte**, und Primimplikanten, die wesentliche Punkte enthalten, **wesentliche** Primimplikanten.

Die billigste Lösung für p lautet also:

$$p = \bar{a} \vee b\bar{c}$$

Beispiel ff



Bei q fällt auf, dass man auf \bar{b} nicht verzichten kann.

Nimmt man aber \bar{b} , so braucht man \bar{a} nicht mehr, weil ja

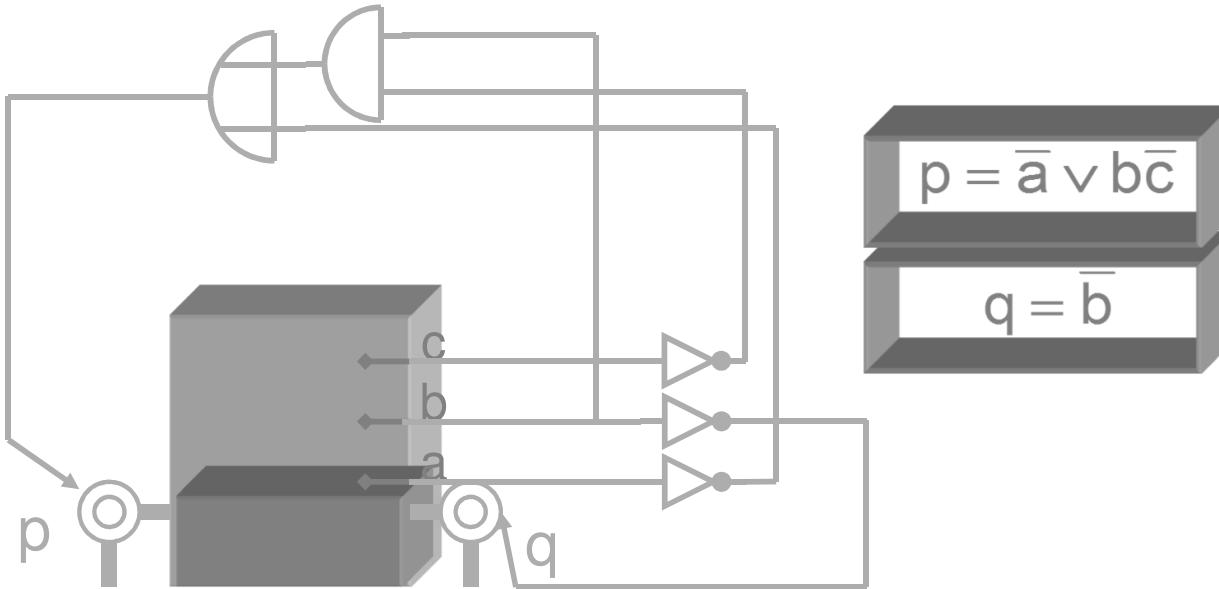
$$\bar{b} \vee \bar{a} \vee r = \bar{b} \vee r$$

Wenn die Hinzunahme eines Primimplikanten u einen anderen Primimplikanten v überflüssig macht, sagen wir auch **u dominiert v** .

Die billigste Lösung für q lautet also:

$$q = \bar{b}$$

Beispiel ff



```
ENTITY ctrl IS  
  PORT (a,b,c: IN BIT; p,q: OUT BIT);  
END ENTITY;
```

```
ARCHITECTURE implementation OF  
ctrl IS  
BEGIN  
  p <= NOT a OR ( b AND NOT c);  
  q <= NOT b;  
END ARCHITECTURE;
```

Karnaugh Diagramme

Man kann die Funktionstafel auch als 2 dimensionale Tabelle hinschreiben. Je nachdem, wie man die Eingangsvektoren ordnet, erhält man maximale Unterwürfel aus Einsen oder Redundanzen (= Primimplikanten) durch spezielle Muster in dieser Tabelle.

Im Karnaugh Diagramm ordnet man die Eingangsvektoren so an, dass bis $n=4$ die Unterwürfel stets als zusammenhängende Gebiete sichtbar sind, denkt man sich die Tabelle auf den Torus geflochten.

Im Veitch Diagramm ordnet man die Eingaben als Binärdarstellungen der Größe nach. Dadurch sind manche Unterwürfel nicht zusammenhängend.

Karnaugh Diagramme ff

Veitch Diagramme

| | | | | |
|-----|-----------|-----------|-----|---|
| a | 0 | 0 | 1 | 1 |
| b | 0 | 1 | 0 | 1 |
| c | \bar{c} | \bar{a} | a | |
| d | | | | |
| 0 0 | | | | |
| 0 1 | | | | |
| 1 0 | | | | |
| 1 1 | | | | |

| | | | | |
|-----|-----------|-----------|-----|-----------|
| a | 0 | 0 | 1 | 1 |
| b | 0 | 1 | 0 | 1 |
| c | \bar{c} | \bar{b} | b | \bar{b} |
| d | | | | |
| 0 0 | | | | |
| 0 1 | | | | |
| 1 0 | | | | |
| 1 1 | | | | |

| | | | | |
|-----|-----------|------------------|---|---|
| a | 0 | 0 | 1 | 1 |
| b | 0 | 1 | 0 | 1 |
| c | \bar{b} | \bar{d} | | |
| d | | | | |
| 0 0 | | | | |
| 0 1 | | | | |
| 1 0 | | $\bar{c}\bar{d}$ | | |
| 1 1 | | | | |

| | | | | |
|-----|-----------|-------------------------|---|---|
| a | 0 | 0 | 1 | 1 |
| b | 0 | 1 | 0 | 1 |
| c | \bar{c} | | | |
| d | | | | |
| 0 0 | | | | |
| 0 1 | | | | |
| 1 0 | | $\bar{a}\bar{b}\bar{c}$ | | |
| 1 1 | | $b\bar{c}\bar{d}$ | | |

Karnaugh Diagramme

| | | | | |
|-----|-----------|-----------|---|---|
| a | 0 | 0 | 1 | 1 |
| b | 0 | 1 | 1 | 0 |
| c | \bar{a} | \bar{a} | | |
| d | | | | |
| 0 0 | | | | |
| 0 1 | | | | |
| 1 1 | | | | |
| 1 0 | | | | |

| | | | | |
|-----|-----------|-----|-----|-----------|
| a | 0 | 0 | 1 | 1 |
| b | 0 | 1 | 1 | 0 |
| c | \bar{b} | b | b | \bar{b} |
| d | | | | |
| 0 0 | | | | |
| 0 1 | | | | |
| 1 1 | | | | |
| 1 0 | | | | |

| | | | | |
|-----|-----------|------------------|---|---|
| a | 0 | 0 | 1 | 1 |
| b | 0 | 1 | 1 | 0 |
| c | \bar{b} | \bar{d} | | |
| d | | | | |
| 0 0 | | | | |
| 0 1 | | | | |
| 1 1 | | $\bar{c}\bar{d}$ | | |
| 1 0 | | | | |

| | | | | |
|-----|-----------|-------------------------|---|---|
| a | 0 | 0 | 1 | 1 |
| b | 0 | 1 | 1 | 0 |
| c | \bar{b} | \bar{d} | | |
| d | | | | |
| 0 0 | | | | |
| 0 1 | | | | |
| 1 1 | | $\bar{a}\bar{b}\bar{c}$ | | |
| 1 0 | | $b\bar{c}\bar{d}$ | | |

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 |

$$p = \overline{a} \vee b\overline{c}$$

| p | a | 0 | 0 | 1 | 1 | |
|---|---|---|---|---|---|--|
| b | 0 | 1 | 0 | 1 | | |
| c | | | | | | |
| 0 | | 1 | * | 0 | 1 | |
| 1 | | * | * | * | 0 | |

| q | a | 0 | 0 | 1 | 1 | |
|---|---|---|---|---|---|--|
| b | 0 | 1 | 0 | 1 | | |
| c | | | | | | |
| 0 | | 1 | * | 1 | 0 | |
| 1 | | * | * | * | 0 | |

$$q = \overline{b}$$

| p | a | 0 | 0 | 1 | 1 | |
|---|---|---|---|---|---|---|
| b | 0 | 1 | 1 | 0 | | |
| c | | | | | | |
| 0 | | 1 | * | 1 | 0 | |
| 1 | | * | * | * | 0 | * |

| q | a | 0 | 0 | 1 | 1 | |
|---|---|---|---|---|---|--|
| b | 0 | 1 | 1 | 0 | | |
| c | | | | | | |
| 0 | | 1 | * | 0 | 1 | |
| 1 | | * | * | 0 | * | |

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

2.1.4 Mehrstufige Schaltkreise

Im Falle zweistufiger Schaltkreise kann man das Syntheseproblem, wenn auch mit sehr hohem Aufwand, lösen. Allerdings stößt man bei der Benutzung zweistufiger Schaltungen sehr schnell an Grenzen. Wir betrachten dazu

Aufgabe:

Entwerfe einen kombinatorischen Schaltkreis PGPC (Parity Generator Parity Checker) zur Berechnung folgender Schaltfunktion:

parity: $\mathcal{B}^n \rightarrow \mathcal{B}$, mit

$$\text{parity}(x) = \left(\sum_{i=0}^{n-1} x_i \right) \bmod 2$$

PGPC als zweistufiger Schaltkreis

Im Falle einer zweistufigen Lösung liegen die Dinge recht klar:

Beobachtung 1

Jeder Minterm x^ε der Parity Funktion ist schon ein Primimplikant.

Grund: Würde man in einem Minterm $x_1^{\varepsilon_1} \dots x_n^{\varepsilon_n}$ mit $\sum \varepsilon_i \bmod 2 = 1$ ein Literal weglassen, etwa ε_j , dann wäre schon

$$x_1^{\varepsilon_1} \dots x_{j-1}^{\varepsilon_{j-1}} x_{j+1}^{\varepsilon_{j+1}} \dots x_n^{\varepsilon_n}(\varepsilon_1, \dots, \varepsilon_{j-1}, 1, \varepsilon_{j+1}, \dots, \varepsilon_n) =$$

$$x_1^{\varepsilon_1} \dots x_{j-1}^{\varepsilon_{j-1}} x_{j+1}^{\varepsilon_{j+1}} \dots x_n^{\varepsilon_n}(\varepsilon_1, \dots, \varepsilon_{j-1}, 0, \varepsilon_{j+1}, \dots, \varepsilon_n)$$

aber $\text{parity}(\varepsilon_1, \dots, \varepsilon_{j-1}, 1, \varepsilon_{j+1}, \dots, \varepsilon_n) \neq \text{parity}(\varepsilon_1, \dots, \varepsilon_{j-1}, 0, \varepsilon_{j+1}, \dots, \varepsilon_n)$

PGPC als zweistufiger Schaltkreis

Beobachtung 2

Jeder Minterm x^ε der Parity Funktion ist schon ein wesentlicher Primimplikant.

Grund: Alle Minterme $x_1^{\varepsilon_1} \dots x_n^{\varepsilon_n}$ mit $(\sum_i \varepsilon_i) \bmod 2 = 1$ werden genau auf einem Punkt = 1. Diese Punkte sind paarweise verschieden.

Beobachtung 3

Die disjunktive Normalform ist kostenminimale 2-stufige Darstellung der Funktion *parity*.

Fazit:

Jede zweistufige Realisierung von *parity* kostet mehr als 2^{n-1} Gatter!

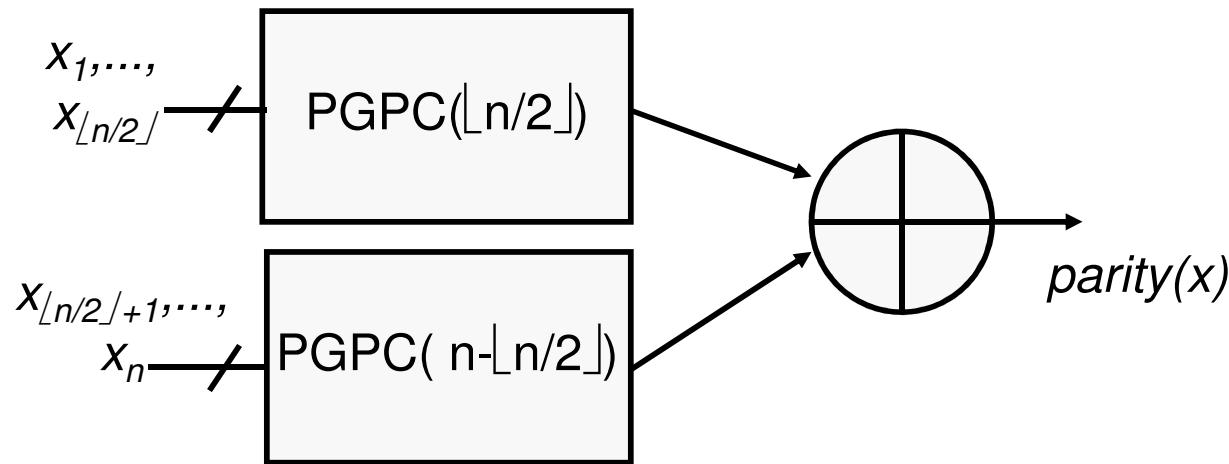
PGPC als mehrstufiger Schaltkreis

Im Falle einer mehrstufigen Lösung bietet sich aber eine denkbar einfache Lösung:

Beobachtung

Die Parity eines Vektors ergibt sich aus der Summe modulo 2 der Parity von zwei Teilvektoren.

Es ergibt sich also folgendes einfache Schema für n bit:



PGPC als mehrstufiger Schaltkreis

Wir brauchen demnach nur $n-1$ XOR Gatter:

Beweis durch Induktion nach n :

$n=2$: $\text{parity}(x_1, x_2) = \text{xor}(x_1, x_2)$. Also nur ein Gatter.

$n>2$: Dann brauchen wir $\lfloor n/2 \rfloor - 1 + (n - \lfloor n/2 \rfloor) - 1 + 1 = n-1$ Gatter

Man beobachtet auch: dem Schaltkreis liegt ein Bildungsgesetz in Abhängigkeit von der Breite n des Eingabevektors zugrunde. Es wäre also nützlich, eine Definition vornehmen zu können, die generisch in dem Parameter n ist.

WÜHDL sieht dazu bei der Definition von ENTITIES sogenannte GENERIC Parameter vor. Damit beschreibt man eine ganze Familie von Bausteinen:

Generische Definition in WüHDL

```
ENTITY pgpc IS
    GENERIC (n : POSITIVE);
    PORT (x: IN BIT_VECTOR(0 TO n-1);
          y: OUT BIT);
END ENTITY pgpc;

ARCHITECTURE structure OF pgpc IS
    COMPONENT pgpc --Benutzt sich selbst zur Rekursion
        GENERIC (n: POSITIVE );
        PORT (x: IN BIT_VECTOR(0 TO n-1);
              y: OUT BIT);
    END COMPONENT;
    COMPONENT xor2
        PORT (x,y: IN BIT; z: OUT BIT);
    END COMPONENT;
    --Zwischenergebnisse (most/least signifikant parity)
    SIGNAL msp, lsp: BIT;
```

Generische Definition in WüHDL -- ff

```
BEGIN --rekursive Definition
    verankerung2: IF n=2 GENERATE
        X0: xor2 PORT MAP (x=>x(0), y=>x(1), z=>y);
        END GENERATE;
    verankerung3: IF n=3 GENERATE
        X1: xor2 PORT MAP (x=>x(0), y=>x(1), z=>msp);
        X2: xor2 PORT MAP (x => x(2), y => msp, z=>y);
        END GENERATE;
    rekursion: IF n>3 GENERATE
        lsb: pgpc -- 1. Aufruf der Rekursion
            GENERIC MAP (n/2)
            PORT MAP (x=>x(n-n/2 TO n-1), y=>lsp);
        msb: pgpc -- 2. Aufruf der Rekursion
            GENERIC MAP (n-n/2)
            PORT MAP (x=>x(0 TO n-n/2-1), y=>msp);
        result: xor2 PORT MAP (x=>msp, y=>lsp, z=>y);
        END GENERATE;
    END ARCHITECTURE;
```

Zahlen

Die wohl wichtigsten Objekte in Rechnern sind Zahlen. Die einfachste Art und Weise, Bitstrings als Zahlen aufzufassen, ist ihre Interpretation als nichtnegative ganze Zahl:

Definition

Wir nennen $u_n: \mathbf{B}^n \mapsto \mathbf{N}_0$

mit $u_n(a_0, \dots, a_{n-1}) := 2^{n-1} \sum_{i=0}^{n-1} a_i 2^{-i}$

Darstellung als vorzeichenlose (unsigned) ganze Zahl.

Das Bit a_0 hat das höchste Gewicht 2^{n-1} . Wir nennen es auch das **signifikanteste Bit**.

Wenn klar ist, dass n die Wortbreite ist, schreiben wir auch einfach u statt u_n .

Unsigned Numbers

Der Zahlenbereich einer vorzeichenlosen ganzen Zahl ist

$$u_n(B^n) = [0: 2^n - 1]$$

Größere Zahlen kann man nicht direkt darstellen. Will man mit größeren Zahlen arbeiten, dann muss dies explizit durch Programmierung realisiert werden.

Anmerkungen:

Rechnerarithmetiker benutzen oft auch eine gespiegelte Notation der Bitstrings um Zahlen zu definieren. In diesem Fall entspricht das Gewicht eines Bits i direkt 2^i .

Den Zahlenbereich erhält man durch

$$u_n(0, \dots, 0) = 0 \quad \text{sowie}$$

$$u_n(1, \dots, 1) = 2^{n-1} \sum_{i=0}^{n-1} 1 \cdot 2^{-i} = 2^{n-1}(2 - 2^{-(n-1)}) = 2^n - 1$$

Decoder

Wir wollen nun nach und nach sehr häufig vorkommende Schaltfunktionen durch kombinatorische Schaltkreise realisieren. Ein spezielle Familie bilden Realisierungen folgender Schaltfunktion:

$decode_m: \mathbf{B}^n \mapsto \mathbf{B}^m$ mit

$$decode_m(a_0, \dots, a_{n-1}) = (y_0, \dots, y_{m-1})$$

$$\Leftrightarrow \forall i: y_i = (u_n(a) = i)$$

Diese Funktion bildet einen Binärcode in einen 1 aus m Code, einen sog. „one hot code“, ab. Deswegen muss natürlich $m \leq 2^n$ sein. Ist $u_n(a) \geq m$, liefert die Funktion den 0-Vektor.

Realisierung des Decoders

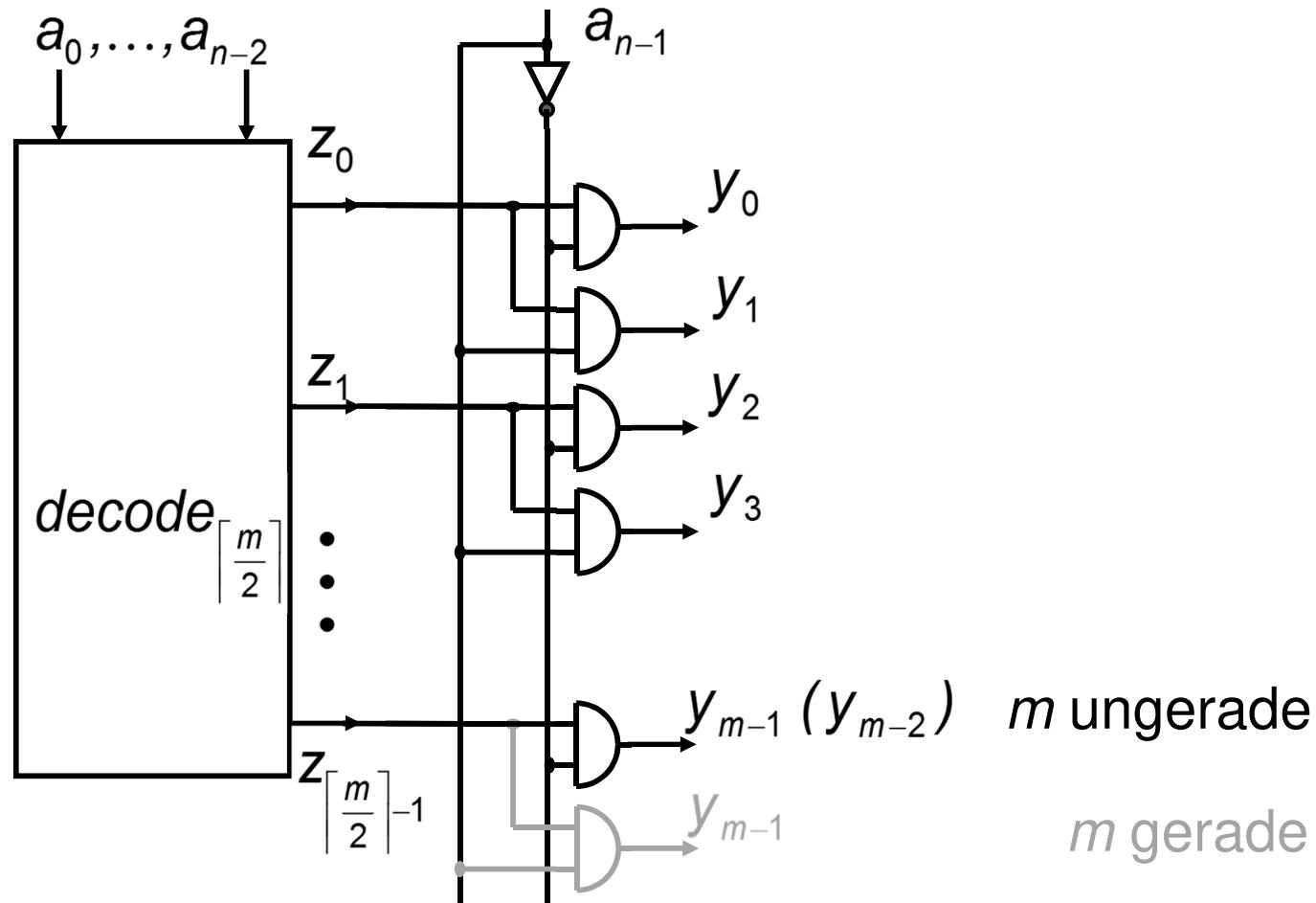
Idee: Wir setzen den Decoder rekursiv aus kleineren Decodern zusammen.

Nehmen wir an, dass wir schon einen Decoder für $\lceil m/2 \rceil$ Ausgänge haben, dann können wir diese Leitungen mit

$$u_n(a_0, \dots, a_{n-2})$$

nummerieren. Einen Decoder mit bis zu doppelt sovielen Ausgängen erhalten wir nun durch Aufspalten dieser Leitungen in eine mit gerader und eine mit ungerader Nummer in Abhängigkeit von a_{n-1} .

Realisierung des Decoders ff



Realisierung des Decoders ff

Für i gerade, ist

$$y_i \Leftrightarrow \bar{a}_{n-1} \cdot z_{\frac{i}{2}} = 1$$

$$\Leftrightarrow a_{n-1} = 0 \text{ und } u_{n-1}(a[0:n-2]) = \frac{i}{2}$$

$$\Leftrightarrow i = 2 \cdot 2^{n-2} \sum_{i=0}^{n-2} a_i 2^{-i} + a_{n-1}$$

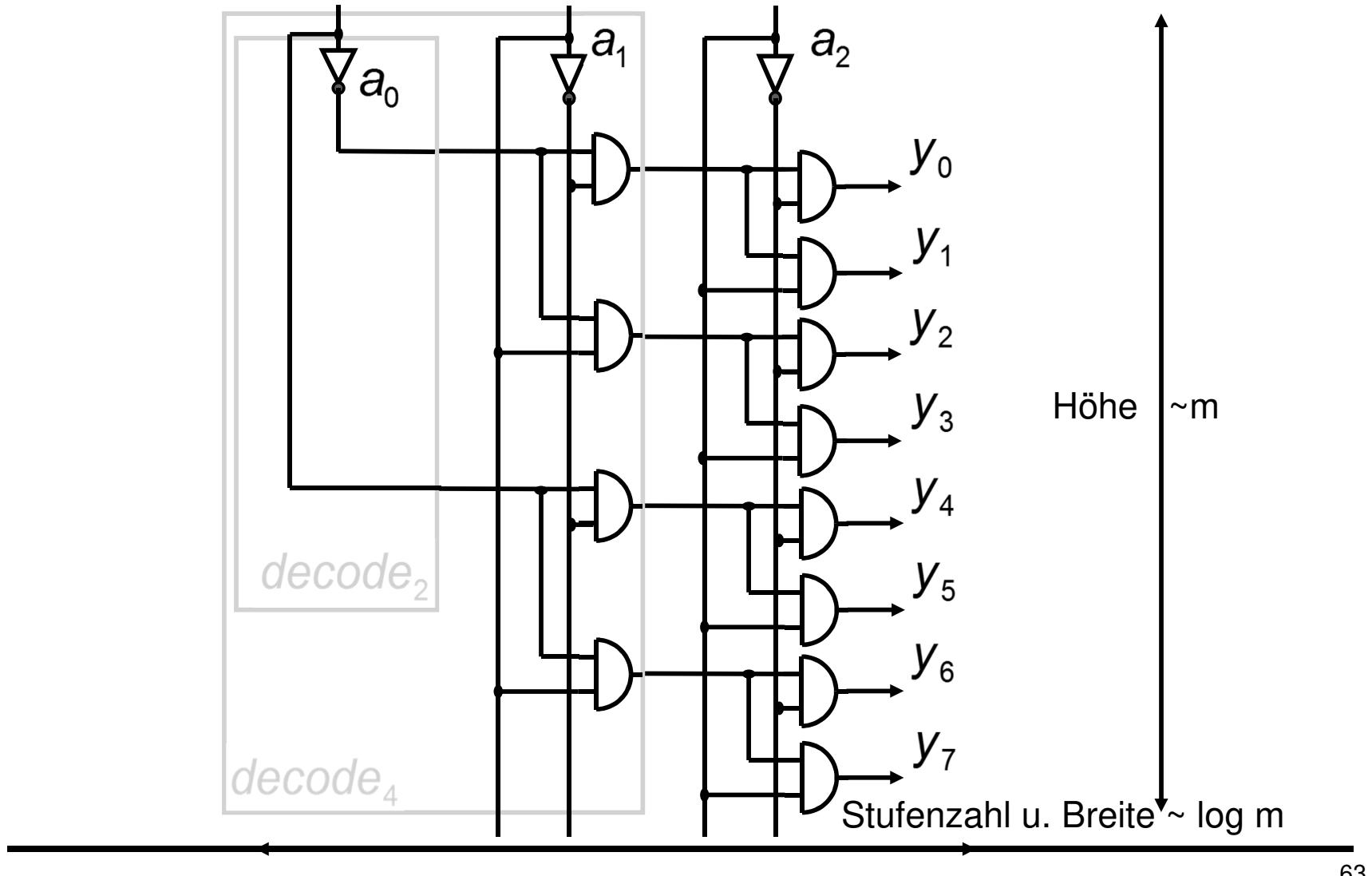
$$\Leftrightarrow i = 2^{n-1} \left(\sum_{i=0}^{n-2} a_i 2^{-i} + 2^{-(n-1)} a_{n-1} \right) = u_n(a)$$

Analog sieht man die Korrektheit für den Fall i ungerade. Es bleibt also nur noch, die trivialen Decoder für $m=1,2$ zu konstruieren, und wir haben eine Konstruktionsvorschrift für alle Decoder:

$$decode_1 = \begin{array}{c} \diagdown \\ \bullet \\ \diagup \end{array}$$

$$decode_2 = \begin{array}{c} \diagup \\ \bullet \\ \diagdown \end{array}$$

Beispiel: Konstruktion für $m=8$



Generische Definition in WüHDL

```
ENTITY decoder IS
    GENERIC ( n : POSITIVE );
    PORT (x: IN BIT_VECTOR(0 TO n-1);
          y: OUT BIT_VECTOR(0 TO 2**n-1) );
END ENTITY decoder;

ARCHITECTURE structure OF decoder IS
    COMPONENT decoder -- zur Rekursion
        GENERIC ( n: POSITIVE );
        PORT (x: IN BIT_VECTOR(0 TO n-1);
              y: OUT BIT_VECTOR(0 TO 2**n-1) );
    END COMPONENT;
    COMPONENT and2 PORT (x,y: IN BIT; z: OUT BIT);
    END COMPONENT;
    COMPONENT two_phases PORT(x:IN BIT;y,yz:OUT BIT);
    END COMPONENT;
    --Zwischenergebnis
    SIGNAL r:BIT_VECTOR(0 TO 2** (n-1)-1); SIGNAL l, lz:BIT;
```

Generische Definition in WüHDL -- ff

```
BEGIN --rekursive Definition
    verankerung2: IF n=1 GENERATE
        D1:two_phases PORT MAP (x=>x(0),y=>y(1),yz=>y(0));
        END GENERATE;
    rekursion: IF n>1 GENERATE
        RDEC:decoder -- Aufruf der Rekursion
            GENERIC MAP (n-1)
            PORT MAP (x=>x(0 TO n-2), y=>r);
        LBIT:two_phases
            PORT MAP (x=>x(n-1), y=>l, yz => lz);
    ANDGATES: FOR i IN 0 TO 2**n-1-1
        GENERATE
            AND0: and2
                PORT MAP (x=>r(i),y=>lz,z=>y(2*i));
            AND1: and2
                PORT MAP (x=>r(i),y=>l,z=>y(2*i+1));
            END GENERATE;
        END GENERATE;
    END ARCHITECTURE;
```

Konstruktion eines Linksshifters

Als weiteres Beispiel soll uns ein Linksshifter dienen:

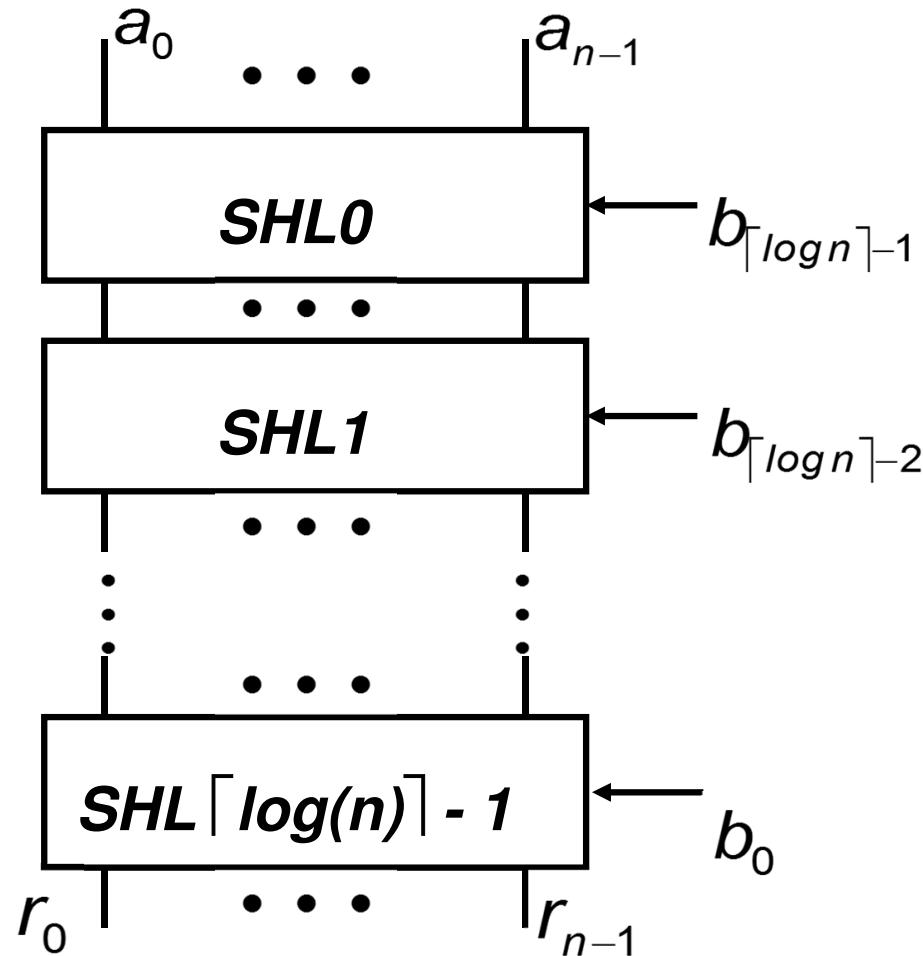
Ein Operand a habe Wortbreite n , in einem weiteren Operanden sei die Shiftweite i als unsigned Zahl kodiert, d.h. es sei

$$0 < u_{\lceil \log n \rceil}(b) < n \quad \text{und} \quad b = (b_0, \dots, b_{\lceil \log n \rceil - 1})$$

Eine 1 auf Position $\log n - i$ bedeutet, dass wir um 2^{i-1} Stellen schieben müssen, und insgesamt um alle Zweierpotenzen, die auf den Positionen vorgegeben sind.

Wir erhalten also einen Shifter, indem wir für jede Zweierpotenz einen Shifter ***SHLi*** bereitstellen, der schiebt, wenn diese mit 1 besetzt ist, und dann diese Bausteine hintereinander schalten:

Linksshifter -- Grundstruktur



Grundstruktur ff

Berechnet nun der Baustein SHL_i die Funktion

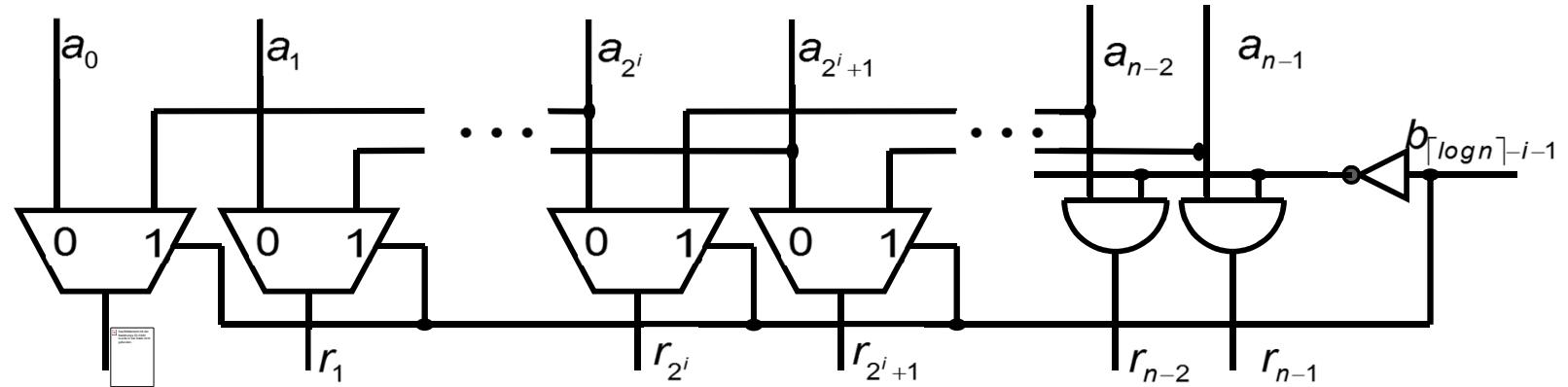
$$shl_i(a, s) = SHL(a, s \cdot 2^i), s \in \mathbb{B}$$

dann berechnet die Schaltung die Funktion

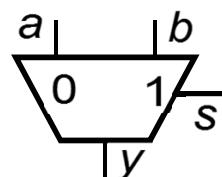
$$\begin{aligned} & shl_{\lceil \log n \rceil - 1}(shl_{\lceil \log n \rceil - 2}(\dots(shl_0(a, b_{\lceil \log n \rceil - 1}), \dots), b_1), b_0) \\ &= SHL(SHL(\dots(SHL(a, b_{\lceil \log n \rceil - 1} \cdot 2^0), \dots), b_1 \cdot 2^{\lceil \log n \rceil - 2}), b_0 \cdot 2^{\lceil \log n \rceil - 1}) \\ &= SHL(a, \sum_{i=0}^{\lceil \log n \rceil - 1} b_{\lceil \log n \rceil - i - 1} \cdot 2^i) \\ &= SHL(a, 2^{\lceil \log n \rceil - 1} \sum_{i=0}^{\lceil \log n \rceil - 1} b_i \cdot 2^{-i}) = SHL(a, u_{\lceil \log n \rceil}(b)) \end{aligned}$$

Shift um eine Zweierpotenz

Ein einzelner SHL*i* Baustein hat folgende einfache Struktur:



Dabei verwenden wir diesmal einen (gerichteten)
Multiplexer

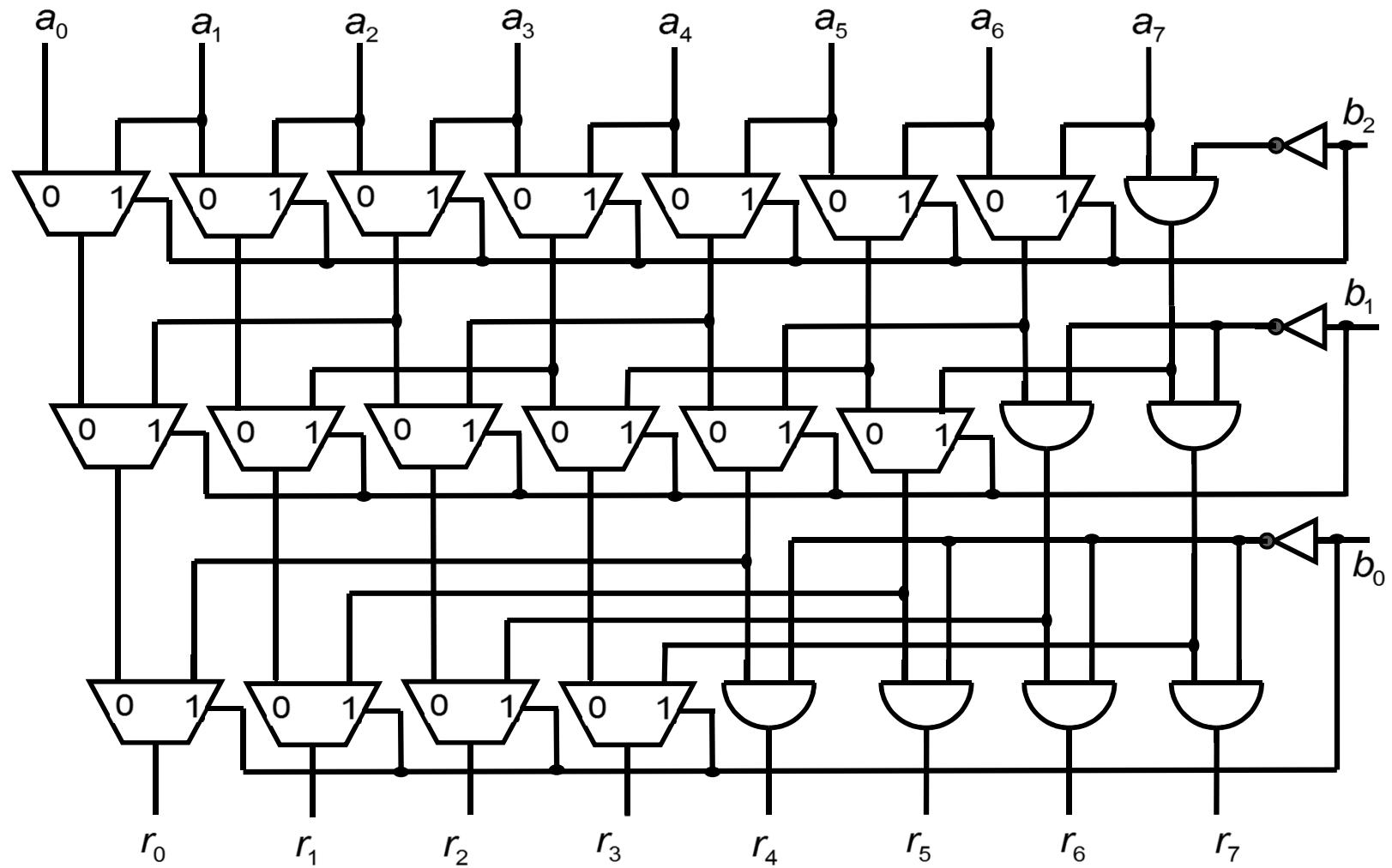


$$\text{Funktion: } y = a \cdot \bar{s} \vee b \cdot s$$

$$y = (\text{if } s \text{ then } b \text{ else } a) = \text{ite}(s, b, a)$$

Wir werden solche Multiplexer auch auf Wortbreite (vgl.
Übungen) nutzen.

Beispiel: 8-Bit Linksshifter



Generische Definition in WüHDL

```
ENTITY lshifter IS
    GENERIC ( n,k : POSITIVE); -- k <= log n
    PORT (x: IN BIT_VECTOR(0 TO n-1);
          w: IN BIT_VECTOR(0 TO k-1);
          y: OUT BIT_VECTOR(0 TO n-1) );
END ENTITY lshifter;
ARCHITECTURE structure OF lshifter IS
COMPONENT lshifter -- zur Rekursion
    GENERIC ( n,k: POSITIVE );
    PORT (x: IN BIT_VECTOR(0 TO n-1);
          w: IN BIT_VECTOR(0 TO k-1);
          y: OUT BIT_VECTOR(0 TO n-1) );
END COMPONENT;
COMPONENT MUX2 PORT (x0,x1,sel:IN BIT; y:OUT BIT);
END COMPONENT;
COMPONENT inverter PORT(x:IN BIT;y:OUT BIT);
END COMPONENT;
```

Generische Definition in WüHDL -- ff

```
COMPONENT and2 PORT(x,y:IN BIT;z:OUT BIT);
END COMPONENT;
SIGNAL r: BIT_VECTOR(0 TO n-1);
SIGNAL wz : BIT;

BEGIN --rekursive Definition
    verankerung:IF k=1 GENERATE
        shiftbyone: FOR i IN 0 TO n-2
        GENERATE
            MUXES: mux2
                PORT MAP (x0=>x(i),x1=>x(i+1),
                           sel=>w(0),y=>y(i));
        END GENERATE;
        fill:and2
            PORT MAP (x=>x(n-1),y=>wz,z=>y(n-1));
        Inv: inverter PORT MAP (x=>w(0),y=>wz);
    END GENERATE;

```

Generische Definition in WüHDL -- ff

```
rekursion: IF k>1 GENERATE
    shifter:lshifter GENERIC MAP (n=>n, k=>k-1)
        PORT MAP (x=>x, w=>w(1 TO k-1), y=>r);
    laststage: FOR i IN 0 TO n-1-2** (k-1)
        GENERATE
            MUXES: mux2
                PORT MAP (x0=>r(i), x1=>r(i+2** (k-1)),
                           sel=>w(0), y=>y(i));
            END GENERATE;
    lastfill: FOR i IN n-2** (k-1) TO n-1
        GENERATE
            ANDGATES: and2
                PORT MAP (x=>r(i), y=>wz, z=>y(i));
            END GENERATE;
            Invl: inverter PORT MAP (x=>w(0), y=>wz);
        END GENERATE;
    END ARCHITECTURE;
```

Addierer

Wir wollen zunächst die Addition von vorzeichenlosen Zahlen betrachten, d.h. wir wollen eine Funktionsscheibe **ADDU** entwickeln, die für zwei Worte a, b der Länge n als Eingänge und ein Wort s der Länge n als Ausgang, sowie ein Flag ov als weiteren Ausgang die folgende Funktion *addu* realisiert:

$addu: \mathbf{B}^{2n} \mapsto \mathbf{B}^{n+1}$ mit

$$addu(a, b) = (ov, s) \Leftrightarrow 2^n ov + u(s) = u(a) + u(b)$$

Da $u(s) \leq 2^n - 1$ zeigt das Flag ov offenbar an, dass die Summe nicht mehr in einem Maschinenwort dargestellt werden kann, da sie größer als $2^n - 1$ ist. Wir nennen dieses Flag auch die **Überlaufanzeige** (overflow flag)

Volladdierer und Halbaddierer

Wir betrachten die Funktion eines sogenannten Volladdierers

$$fa: \mathbf{B}^3 \mapsto \mathbf{B}^2 \text{ wobei } fa(x_1, x_2, x_3) = (c, s) \Leftrightarrow 2c + s = x_1 + x_2 + x_3$$

Man kann den Volladdierer direkt durch eine entsprechende Schaltung realisieren:

| | | | | | |
|---------------|---|---|---|---|--------------------------------------|
| | 0 | 0 | 1 | 1 | x_1 |
| c | 0 | 1 | 0 | 1 | x_2 |
| $\overline{}$ | 0 | 0 | 0 | 1 | |
| | 1 | 0 | 1 | 1 | |
| x_3 | | | | | $c = x_3(x_1 \vee x_2) \vee x_1 x_2$ |

| | | | | | |
|---------------|---|---|---|---|--|
| | 0 | 0 | 1 | 1 | x_1 |
| s | 0 | 1 | 0 | 1 | x_2 |
| $\overline{}$ | 0 | 0 | 1 | 1 | 0 |
| | 1 | 1 | 0 | 0 | 1 |
| x_3 | | | | | $s = x_3(x_1 \equiv x_2) \vee \bar{x}_3(x_1 \oplus x_2)$ |

oder unter Zuhilfenahme zweier **Halbaddierer** (half adder), die nur die Summe zweier Bits als zweistellige Zahl kodieren können:

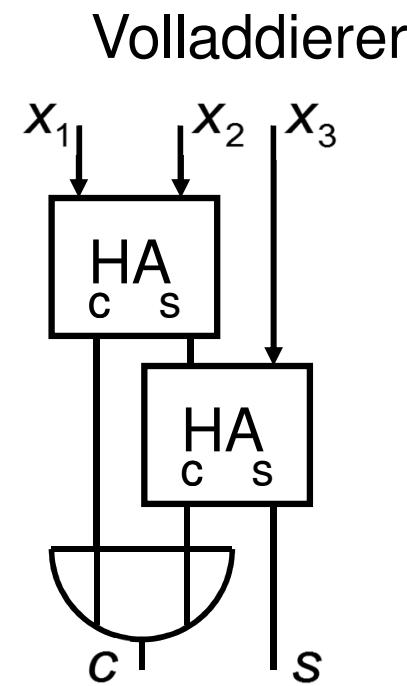
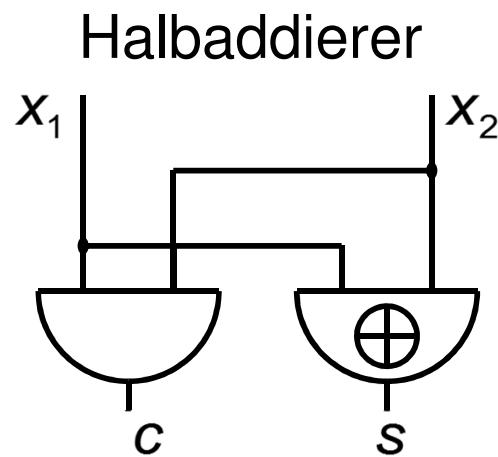
$$ha: \mathbf{B}^2 \mapsto \mathbf{B}^2 \text{ wobei } ha(x_1, x_2) = (c, s) \Leftrightarrow 2c + s = x_1 + x_2$$

| | | | |
|---------------|---|---|-------|
| c | 0 | 1 | x_1 |
| $\overline{}$ | 0 | 0 | |
| | 1 | 0 | 1 |
| x_2 | | | |

| | | | |
|---------------|---|---|-------|
| s | 0 | 1 | x_1 |
| $\overline{}$ | 0 | 0 | 1 |
| | 1 | 1 | 0 |
| x_2 | | | |

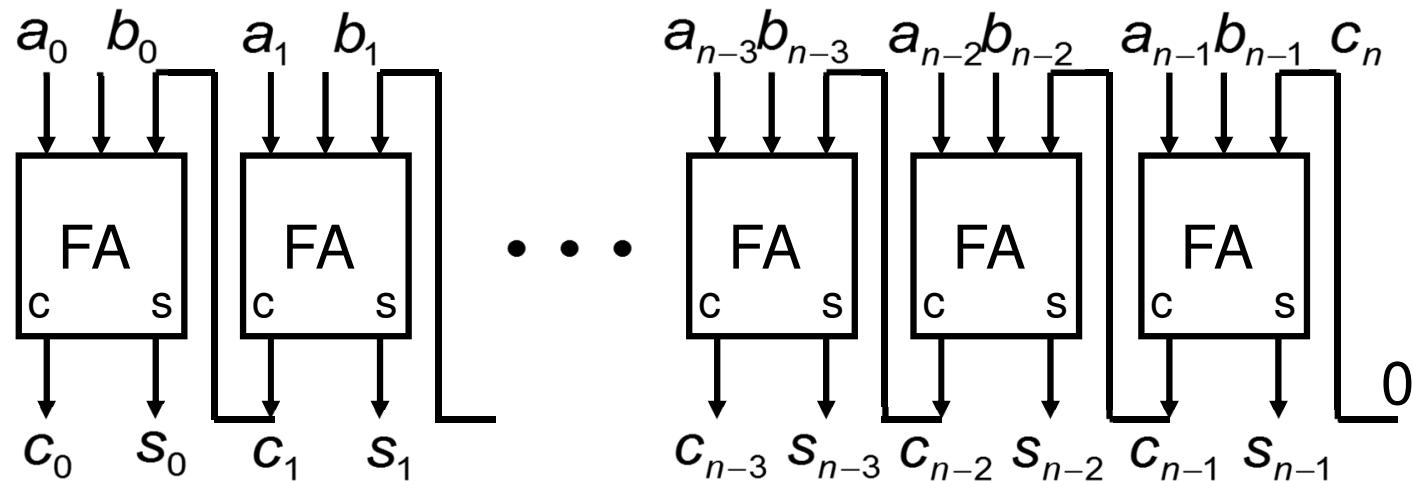
Volladdierer und Halbaddierer ff

Damit ergäbe sich z.B. folgende Schaltung:



Der ripple carry Addierer

Man erhält nun auf ganz einfache Weise einen Schaltkreis zur Addition, wenn man die uns allen bekannte Grundschulmethode auf das Binärsystem überträgt, und von den weniger signifikanten Stellen zur signifikanten Stelle hin die Ziffern jeweils mit Volladdierern addiert und einen entstehenden Übertrag weiterleitet:



Wir nennen diese Schaltung den **ripple carry Adder**.

Ripple carry Adder ff

Der Volladdierer auf Position $n - 1$ kann durch einen Halbaddierer ersetzt werden, weil ein Eingang (carry in) immer 0 ist, oder er kann dazu benutzt werden auch $u(a) + u(b) + 1$ zu berechnen (carry in = 1).

Allerdings ist die Verzögerungszeit dieses Addierers enorm hoch, da im schlimmsten Fall der Übertrag durch n Volladdiererstufen läuft:

Kann man schneller als seriell addieren?

Der conditional sum adder

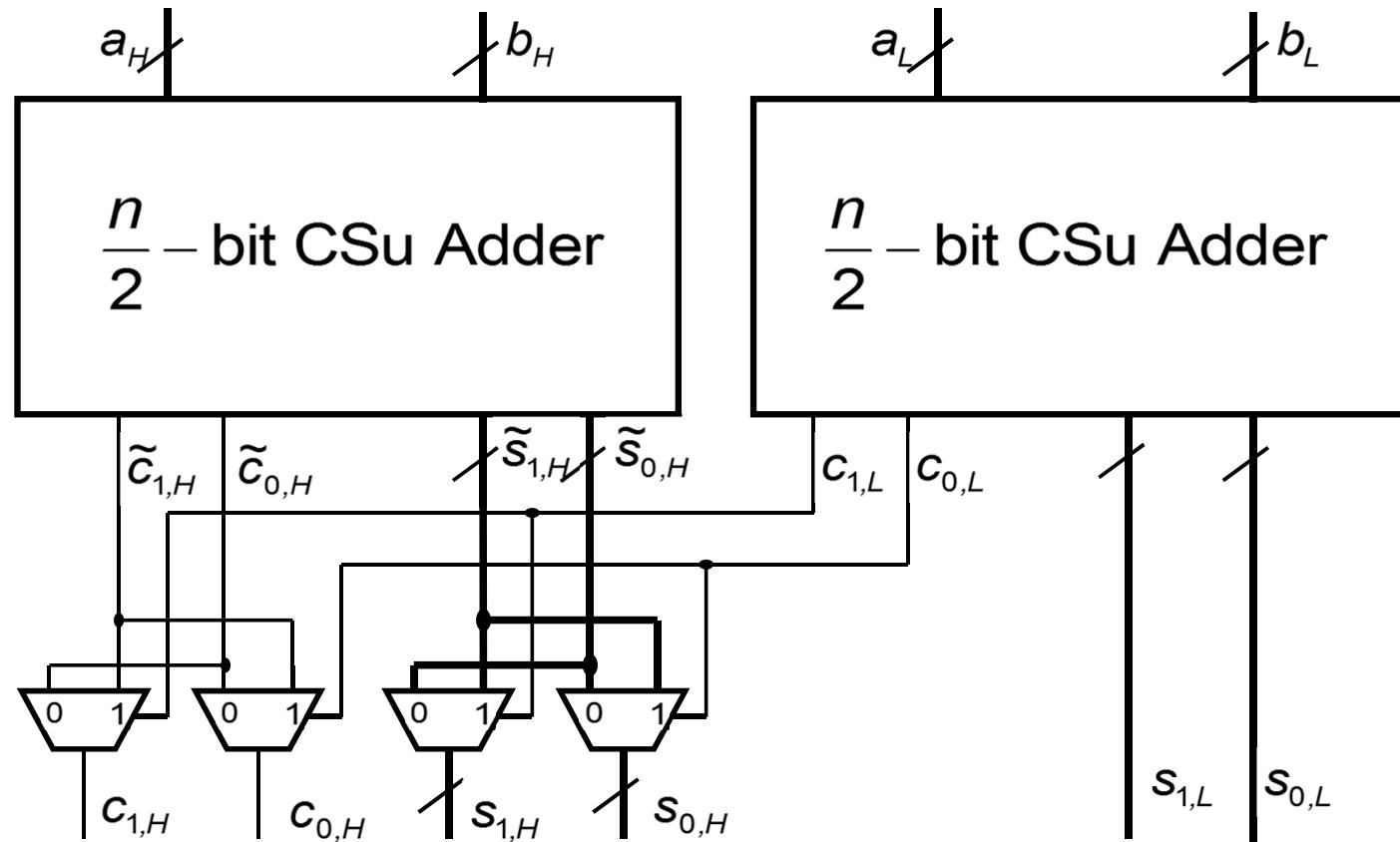
Diese Idee stammt von Sklansky und ist eines der Paradebeispiele dafür, dass man eine zunächst streng sequentiell anmutende Aufgabe schneller parallel lösen kann, wenn man etwas mehr tut:

Idee:

Berechne parallel die Summe und die Summe plus eins für die signifikantere Hälfte und die weniger signifikante Hälfte. Wähle dann in einem Schlag in Abhängigkeit vom Übertrag der weniger signifikanten Hälfte die richtigen Varianten der Summen aus.

Dies führt zu folgendem rekursiven Schema:

Conditional sum adder ff



Conditional sum adder ff

Dabei seien:

$$a_H := a[0 : \frac{n}{2} - 1]; \quad a_L := a[\frac{n}{2} : n - 1];$$
$$b_H := b[0 : \frac{n}{2} - 1]; \quad b_L := b[\frac{n}{2} : n - 1];$$

Für den einen Addierer gelte

$$u_{\frac{n}{2}}(s_{0,L}) + 2^{\frac{n}{2}} c_{0,L} = u_{\frac{n}{2}}(a_L) + u_{\frac{n}{2}}(b_L);$$

$$u_{\frac{n}{2}}(s_{1,L}) + 2^{\frac{n}{2}} c_{1,L} = u_{\frac{n}{2}}(a_L) + u_{\frac{n}{2}}(b_L) + 1;$$

Und für den identischen anderen Addierer ebenfalls

$$u_{\frac{n}{2}}(\tilde{s}_{0,H}) + 2^{\frac{n}{2}} \tilde{c}_{0,H} = u_{\frac{n}{2}}(a_H) + u_{\frac{n}{2}}(b_H);$$

$$u_{\frac{n}{2}}(\tilde{s}_{1,H}) + 2^{\frac{n}{2}} \tilde{c}_{1,H} = u_{\frac{n}{2}}(a_H) + u_{\frac{n}{2}}(b_H) + 1;$$

Conditional sum adder ff

Das Verfahren ist korrekt, denn

$$\begin{aligned} & u_n(a) + u_n(b) \\ &= 2^{\frac{n}{2}}(u_{\frac{n}{2}}(a_H) + u_{\frac{n}{2}}(b_H)) + u_{\frac{n}{2}}(a_L) + u_{\frac{n}{2}}(b_L) \\ &= 2^{\frac{n}{2}}(u_{\frac{n}{2}}(a_H) + u_{\frac{n}{2}}(b_H)) + 2^{\frac{n}{2}}c_{0,L} + u_{\frac{n}{2}}(s_{0,L}) \\ &= \begin{cases} 2^{\frac{n}{2}}(u_{\frac{n}{2}}(a_H) + u_{\frac{n}{2}}(b_H)) + u_{\frac{n}{2}}(s_{0,L}) & \text{falls } c_{0,L} = 0 \\ 2^{\frac{n}{2}}(u_{\frac{n}{2}}(a_H) + u_{\frac{n}{2}}(b_H) + 1) + u_{\frac{n}{2}}(s_{0,L}) & \text{falls } c_{0,L} = 1 \end{cases} \\ &= \begin{cases} 2^n c_{0,H} + u_n(\tilde{s}_{0,H}, s_{0,L}) & \text{falls } c_{0,L} = 0 \\ 2^n c_{1,H} + u_n(\tilde{s}_{1,H}, s_{0,L}) & \text{falls } c_{0,L} = 1 \end{cases} \quad \text{Ergebnis lt. Verdrahtung} \end{aligned}$$

Conditional sum adder ff

Analog rechnet man nach, dass

$$u_n(a) + u_n(b) + 1 = \begin{cases} 2^n c_{0,H} + u_n(\tilde{s}_{0,H}, s_{1,L}) & \text{falls } c_{1,L} = 0 \\ 2^n c_{1,H} + u_n(\tilde{s}_{1,H}, s_{1,L}) & \text{falls } c_{1,L} = 1 \end{cases}$$

Werden die Teile rekursiv genauso realisiert, berechnet die Schaltung sowohl die Summe, als auch die Summe plus 1 parallel in Tiefe

$$\begin{aligned} T_{CSU}(n) &= T_{CSU}\left(\frac{n}{2}\right) + d(MUX) \\ &= \dots \\ &= T_{CSU}(1) + \log n \cdot d(MUX) \end{aligned}$$

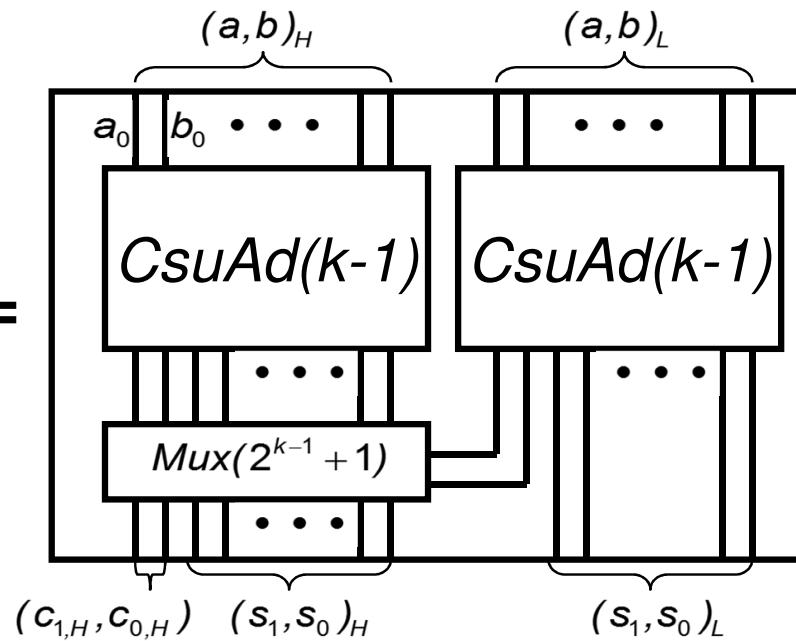
Die Tiefe wächst also nur logarithmisch mit n .

Conditional sum adder ff

Man kann nach diesem rekursiven Bildungsgesetz nun auch leicht einen übersichtlichen Schaltplan definieren. Dazu ist es günstig, die Operanden- und Ergebnisleitungen wieder bitweise gemischt zu führen, d.h. in der Reihenfolge \dots, a_i, b_i, \dots sowie $c_{1,H}, c_{0,H}, s_{1,0}, s_{0,0}, \dots, s_{1,n-1}, s_{0,n-1}$

Rekursion für $k = \log n$:

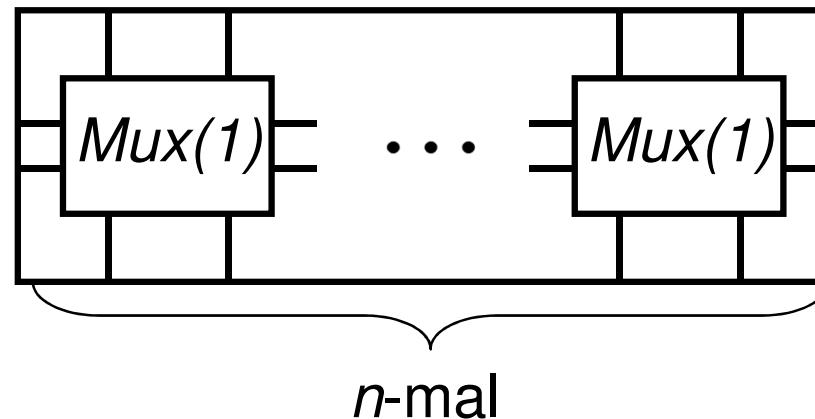
CsuAd(k) =



Conditional sum adder ff

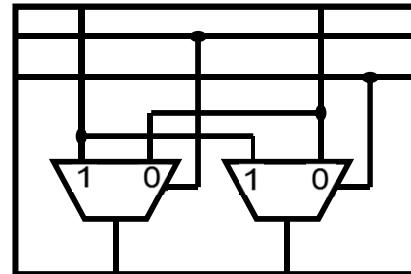
Der Auswahlbaustein wird durch einfaches Nebeneinander-setzen realisiert:

Mux(n) =

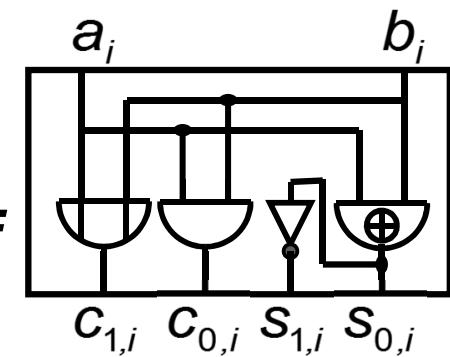


Und für ein Bitpaar haben wir

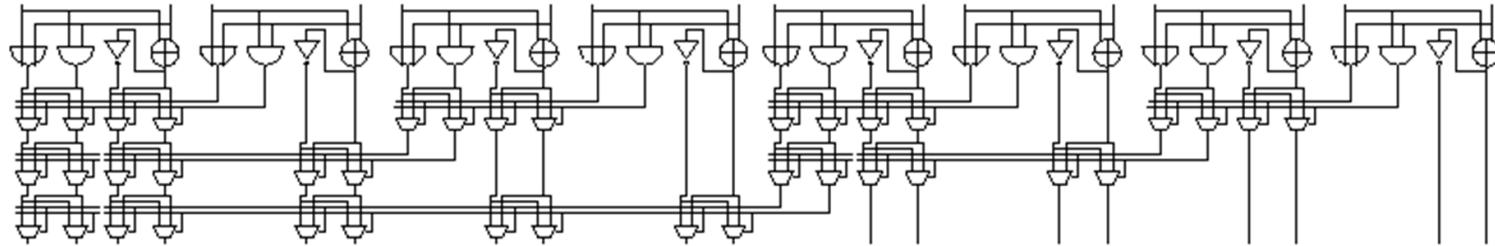
Mux(1) =



CsuAd(1) =

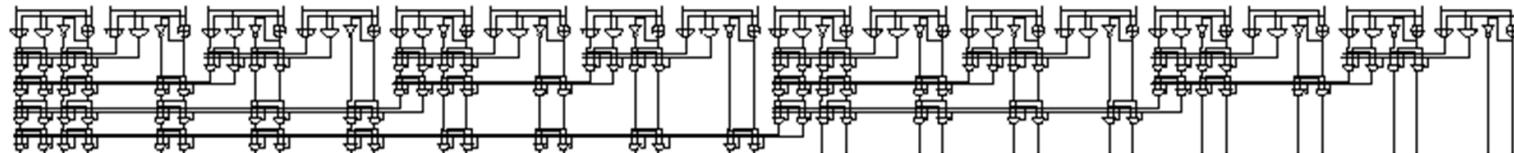


Beispiele:



8 bit Addierer

16 bit Addierer



Es gibt eine Fülle von Möglichkeiten, schnelle Addierer zu konstruieren.
Wir stellen diese zurück und tun so, als hätten wir brauchbare
Schaltungen.

Generische Definition in WüHDL

```
-- Definition des Grundbausteins fuer 1 Bit Breite
ENTITY simple_adder IS
    PORT ( a,b : IN BIT;
           s0,s1 : OUT BIT;
           c0,c1 : OUT BIT  );
END simple_adder;

ARCHITECTURE behavior OF simple_adder IS
BEGIN
    c0 <= a AND b;  s0 <= a XOR b;
    c1 <= a OR b;   s1 <= NOT(a XOR b);
END behavior;
```

Generische Definition in WüHDL -- ff

```
-- Definition des Auswahlbausteins

ENTITY auswahl IS
  GENERIC ( n: POSITIVE );
  PORT ( x0,x1: IN BIT_VECTOR(0 TO n-1);
         sel0,sel1: IN BIT;
         s0,s1: OUT BIT_VECTOR(0 TO n-1));
END auswahl;

ARCHITECTURE behavior OF auswahl IS
BEGIN
  s0 <= x0 WHEN sel0 = '0' ELSE x1;
  s1 <= x0 WHEN sel1 = '0' ELSE x1;
END behavior;
```

Generische Definition in WüHDL -- ff

```
-- CSU Addierer: Rekursive Definition der Struktur
ENTITY csu_adder IS
    GENERIC ( n : POSITIVE );
    PORT ( a, b      : IN  BIT_VECTOR(0 TO n-1);
           s0, s1 : OUT BIT_VECTOR(0 TO n-1);
           c0, c1: OUT BIT );
END csu_adder;

ARCHITECTURE structure OF csu_adder IS
    COMPONENT csu_adder ...
    COMPONENT simple_adder ...
    COMPONENT auswahl ...
    SIGNAL msb0, msb1 : BIT_VECTOR(0 TO n-n/2-1);
    SIGNAL msbc0, msbc1 : BIT;
    SIGNAL carry0, carry1 : BIT;

BEGIN -- csuadder structure
    verankerung: IF n=1 GENERATE
        anker: simple_adder
            PORT MAP ( a => a(0),b => b(0),s0 => s0(0),s1 => s1(0),
                        c0 => c0, c1 => c1);
    END GENERATE;
```

Generische Definition in WüHDL -- ff

rekursion:

```
IF n>=2 GENERATE
    adder_msb: csu_adder -- 1. Aufruf der Rekursion
    GENERIC MAP (n-n/2)
    PORT MAP (a => a(0 TO n-n/2-1), b => b(0 TO n-n/2-1),
               s1 => msb1, s0 => msb0,
               c1 => msbc1, c0 => msbc0      );
    adder_lsb: csu_adder -- 2. Aufruf der Rekursion
    GENERIC MAP (n/2)
    PORT MAP (a => a(n-n/2 TO n-1), b => b(n-n/2 TO n-1),
               s1 => s1(n - n/2 TO n-1), c1 => carry1,
               s0 => s0(n - n/2 TO n-1), c0 => carry0      );
    selektion: auswahl GENERIC MAP ( n - n/2 )
    PORT MAP (x0 => msb0, x1 => msb1,
               sel0 => carry0, sel1 => carry1,
               s0 => s0(0 TO n-n/2-1),
               s1 => s1(0 TO n-n/2-1)      );
    selektion_carry: auswahl GENERIC MAP (1)
    PORT MAP (x0(0) => msbc0, x1(0) => msbc1,
               sel0 => carry0, sel1 => carry1,
               s0(0) => c0, s1(0) => c1      );
END GENERATE;
END structure;
```

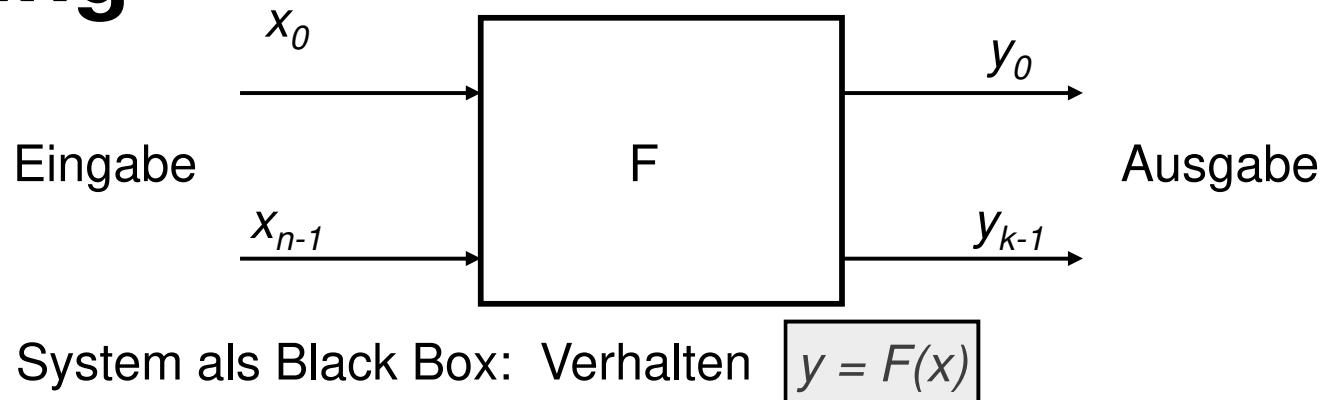
2.2 Synchrone Schaltkreise und endliche Automaten

Zur Vorlesung Rechenanlagen

SS 2019



Erinnerung



Wir hatten bei digitalen Systemen unterschieden zwischen.

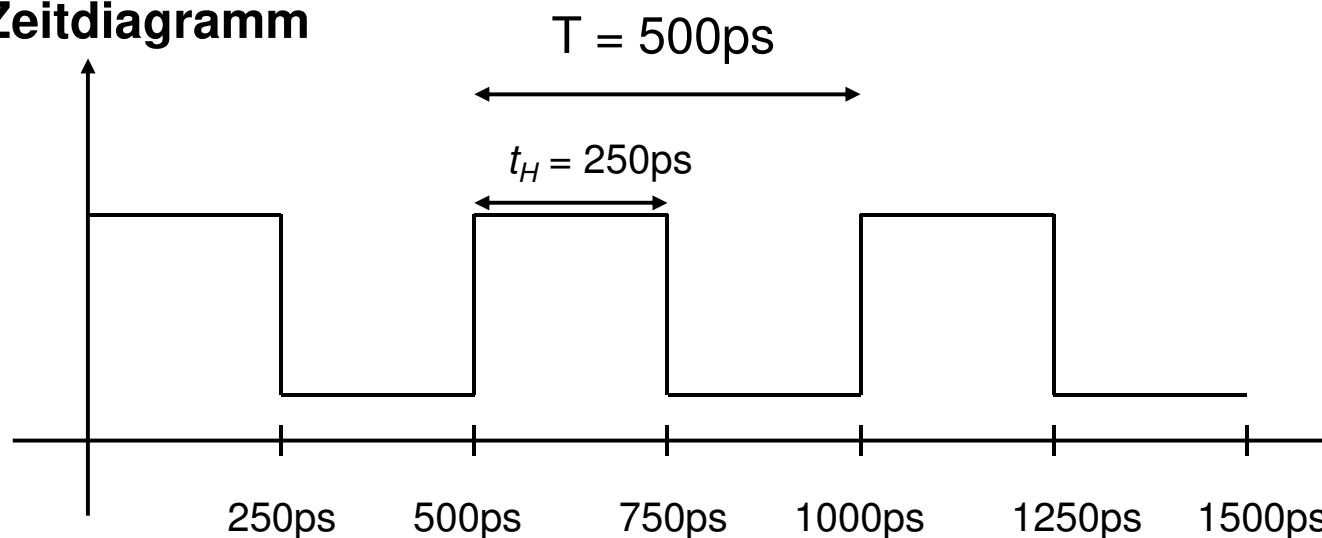
kombinatorisch $\Leftrightarrow \forall t: y(t) = F(x(t))$
und **sequentiell** $\Leftrightarrow \forall t: y(t) = F(x(0:t))$

Wir werden uns nun zunächst mit dem Entwurf von
synchronen sequentiellen Systemen beschäftigen und dann
zeigen, warum asynchrone Systeme problematisch sind.

Synchrone Systeme -- Taktung

Ein **Takt** clk ist ein periodisches binäres Signal mit der Periode T und einer Pulsbreite $0 < t_H < T$.

Zeitdiagramm

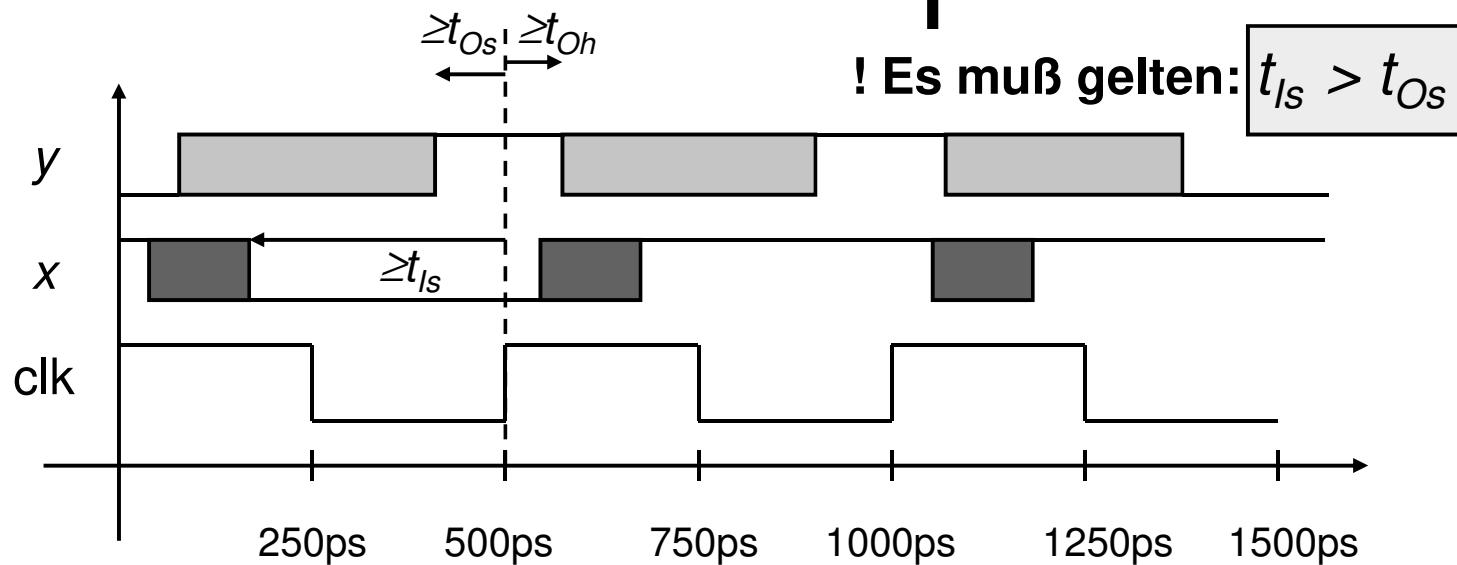


$$\text{Frequenz } f := 1/T = 2 \text{ GHz}$$

Synchrone Systeme

Ein System heißt synchron, wenn unter der Bedingung, dass die Eingabesignale $x(t)$ in einer Umgebung $[n^*T-t_{Is}, n^*T+t_{Oh}]$ um den 0/1 (1/0) Übergang eines Taktes stabil sind, die Ausgabesignale $y(t)$ in einer Umgebung $[n^*T-t_{Os}, n^*T+t_{Oh}]$ stabil sind und für alle n gilt

$$y(n^*T) = F(x(0), \dots, x(n^*T))$$



Bemerkung

Bei einem synchronen System genügt es, vorausgesetzt die Zeitbedingungen werden eingehalten, nur noch den Zusammenhang zwischen den Signalwerten zu Zeitpunkten zu betrachten, die Vielfache der Taktperiode sind. Man kann also vereinfacht auch den Signalen eine neue Zeitskala zuordnen. $s(t)$ bezeichnet dann den Wert von s im t -ten **Zyklus**, also zum Zeitpunkt $t*T$.

Ein System heißt asynchron sonst!

Synchrone Schaltkreise realisieren Synchrone Systeme - die richtige Taktung und Zeitbedingungen vorausgesetzt:

Synchrone Schaltkreise

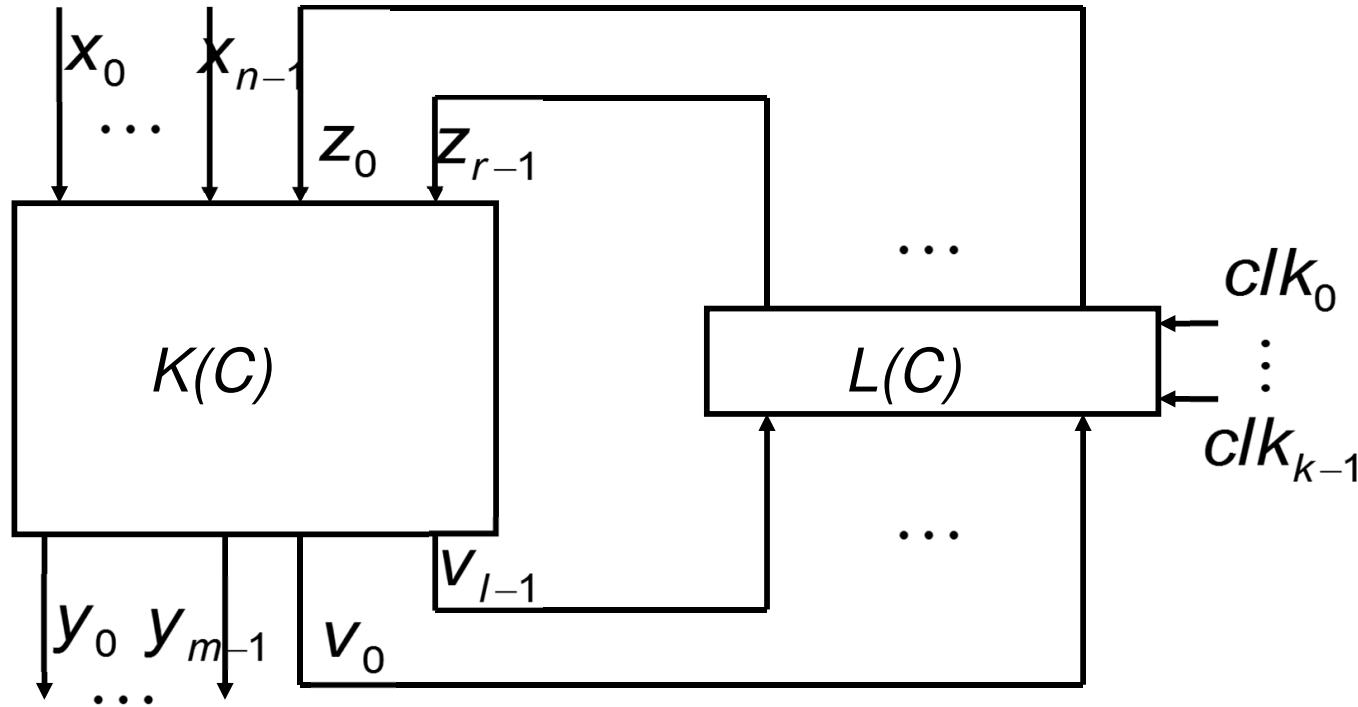
Definition

Ein Schaltkreis C über einem Bausteinsystem A heißt **synchron**, genau dann wenn

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

Nach dieser Definition ist folgende Normierung für synchrone Schaltkreise legitim:

Huffman Normalform

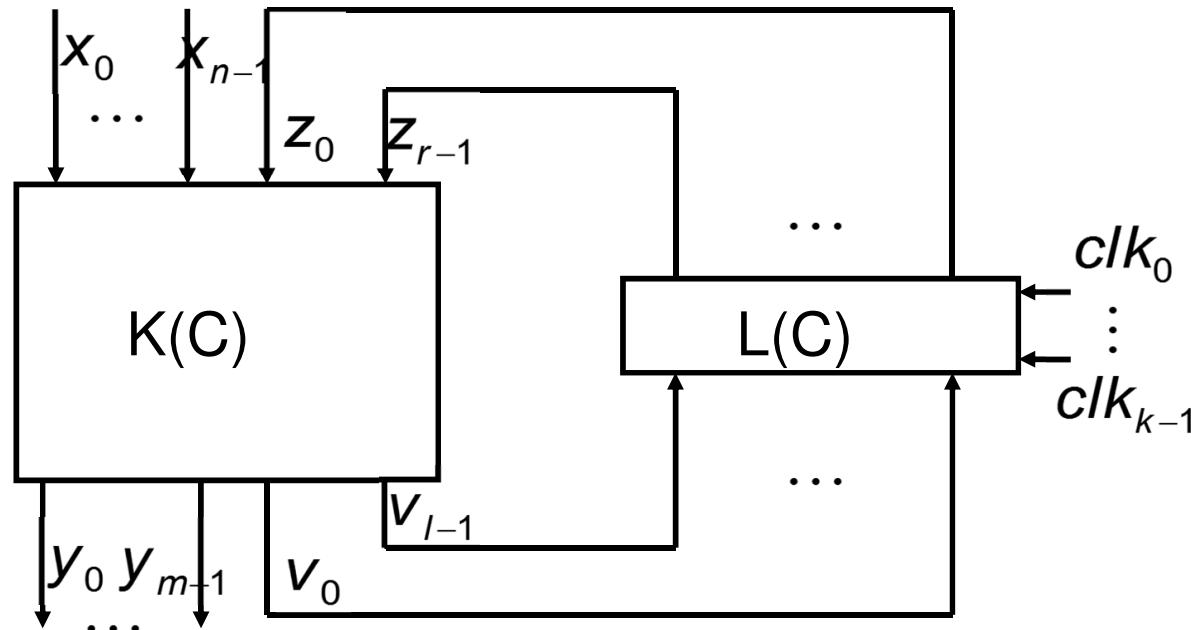


$L(C)$ besteht nur aus Latches, die von den Takten clk_0, \dots, clk_{k-1} kontrolliert werden.

v_0, \dots, v_{l-1} sind Eingänge der Latches

z_0, \dots, z_{r-1} sind Ausgänge der Latches

Huffman Normalform ff



$K(C)$ ist ein kombinatorischer Schaltkreis, die Signale

$$x_0, \dots, x_{n-1}, y_0, \dots, y_{m-1}, v_0, \dots, v_{l-1}, z_0, \dots, z_{r-1}$$

müssen nicht notwendig verschieden sein, allerdings sei:

$$\{z_0, \dots, z_{r-1}\} \cap \{x_0, \dots, x_{n-1}\} = \emptyset$$

Schaltwerke

Man kann nun zwei weitere Typen von synchronen Schaltkreisen unterscheiden:

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

Synchrone Schaltungen werden nun benutzt, um Rechenvorschriften über einer Eingabefolge auszuführen. Der kombinatorische Schaltkreis liefert dazu eine Funktion f , die auf ein neues Element der Eingabefolge und einen aktuellen Zustand der Latches einen neuen Zustand und ein Element der Ausgabefolge berechnet. Zustands- und Ein-/Ausgabemengen sind jeweils endlich.

Mealy Automaten

Definition

Ein Tupel $M=(Z, X, Y, \delta, \lambda)$ heißt **Mealy (Moore) Automat**

: $\Leftrightarrow Z, X, Y$ sind endliche Mengen,

$$\delta: Z \times X \rightarrow Z \text{ und } \lambda: Z \times X \rightarrow Y \quad (\lambda: Z \rightarrow Y)$$

Z heißt **Zustandsmenge**, X **Eingabemenge**, Y **Ausgabemenge**, δ **Übergangs-** und λ **Ausgabefunktion**.

Da Z, X, Y endliche Mengen sind, können wir ohne Einschränkung im weiteren annehmen:

$$Z \subseteq \mathcal{B}^r, \quad X \subseteq \mathcal{B}^n, \quad Y \subseteq \mathcal{B}^m$$

Mealy Automaten

Solche Automaten beschreiben formal die Steuerung zeitlicher Abläufe in einem synchronen Raster bei endlichem Gedächtnis. Man kann sie eins zu eins auf synchrone Schaltwerke abbilden, da für

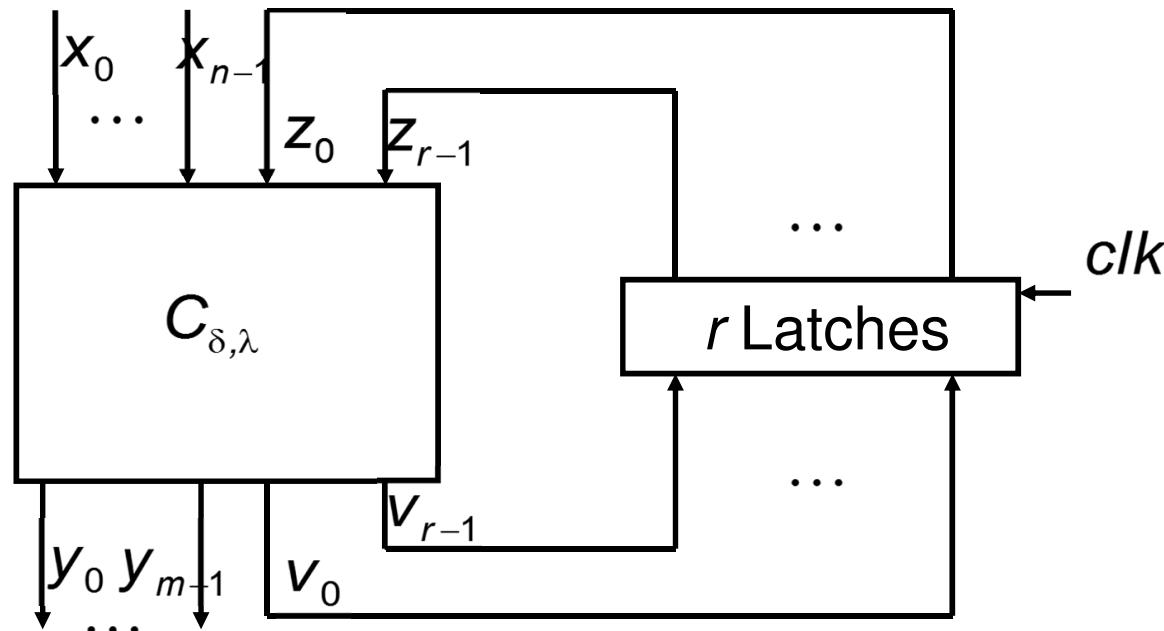
$$Z \subseteq \mathbf{B}^r, \quad X \subseteq \mathbf{B}^n, \quad Y \subseteq \mathbf{B}^m$$

die Übergangs- und Ausgabefunktion δ und λ partielle Schaltfunktionen sind:

$$\delta \in S_{n+r,r}^D \quad \lambda \in S_{n+r,m}^D \quad (\text{Moore Automat} : \lambda \in S_{r,m}^D)$$

Man kann also einen Schaltkreis $C_{\delta,\lambda}$ konstruieren, der die Funktion $f_{\delta,\lambda} \in S_{n+r,m+r}^D$ mit $f_{\delta,\lambda}(z, x) = (\lambda(z, x), \delta(z, x))$ berechnet, und dann den Zustand über Latches puffern:

Mealy Automat ff



Umgekehrt kann man die Menge der Zustände, die die Latches eines synchronen Schaltkreises annehmen als Zustandsmenge, die Eingabevektoren als Eingabe und die Ausgabevektoren als Ausgabe eines Mealy Automaten betrachten.

Zeitbedingungen in Schaltwerken

Diese eins zu eins Korrespondenz gilt allerdings nur dann, wenn der Takt gewissen Zeitbedingungen genügt. Wir wollen die Gültigkeit dieser Zeitbedingungen immer stillschweigend voraussetzen, wenn wir synchrone Schaltwerke betrachten, uns diese Bedingungen nun aber vorab erarbeiten:

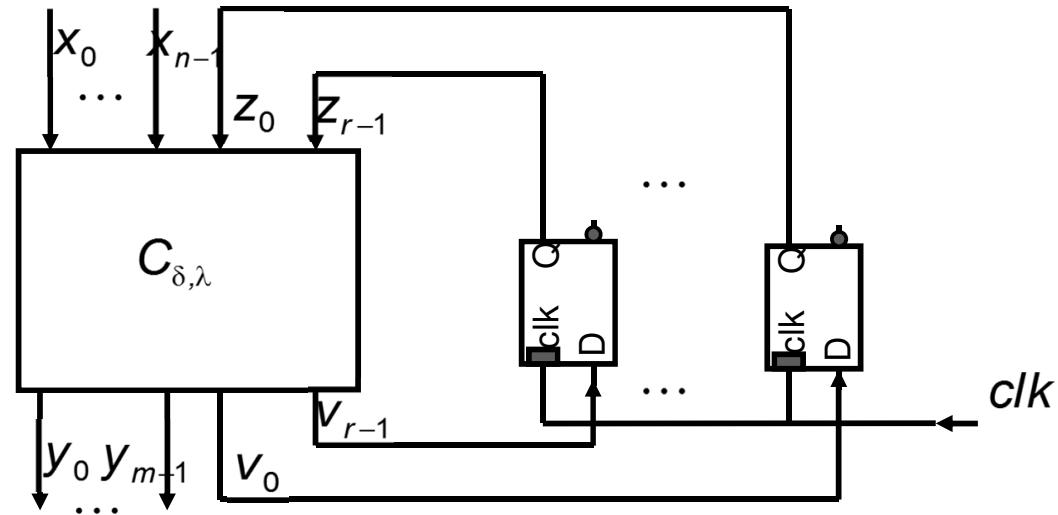
Um aus einem Synchronen Schaltwerk ein Synchrone System zu machen, nehmen wir an, dass die Eingänge stabil bleiben müssen, wenn die Latch Ausgänge stabil sind.

Wir betrachten nun

- taktzustandsgesteuerte Schaltwerke, und
- taktflankengesteuerte Schaltwerke

Zeitbedingungen in Schaltwerken ff

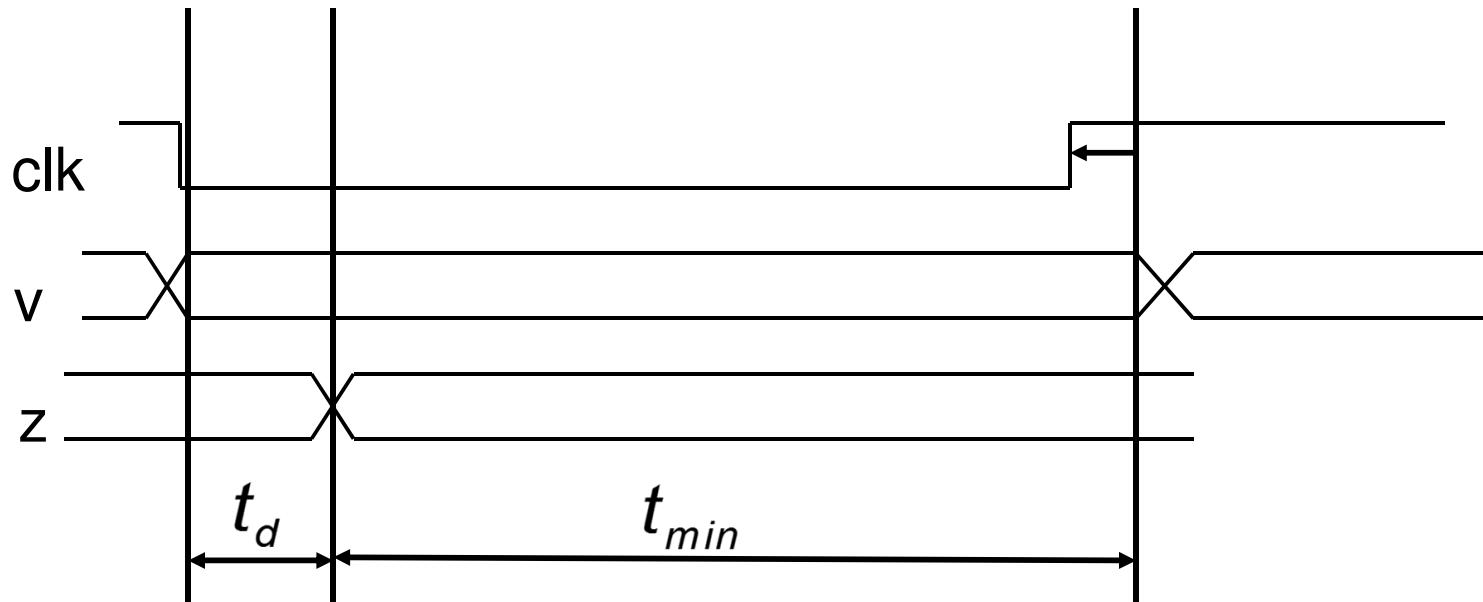
Betrachten wir zunächst ein taktzustandsgesteuertes Schaltwerk:



Da die Latches in der aktiven (hier 0) Phase des Taktes transparent sind, propagiert die Berechnung des Folgezustands direkt nach Beginn der Übernahme durch den Schaltkreis. Dies darf sich während der aktiven Phase nicht mehr an den Ausgängen bemerkbar machen:

Zeitbedingungen in Schaltwerken ff

Zeitdiagramm (I)



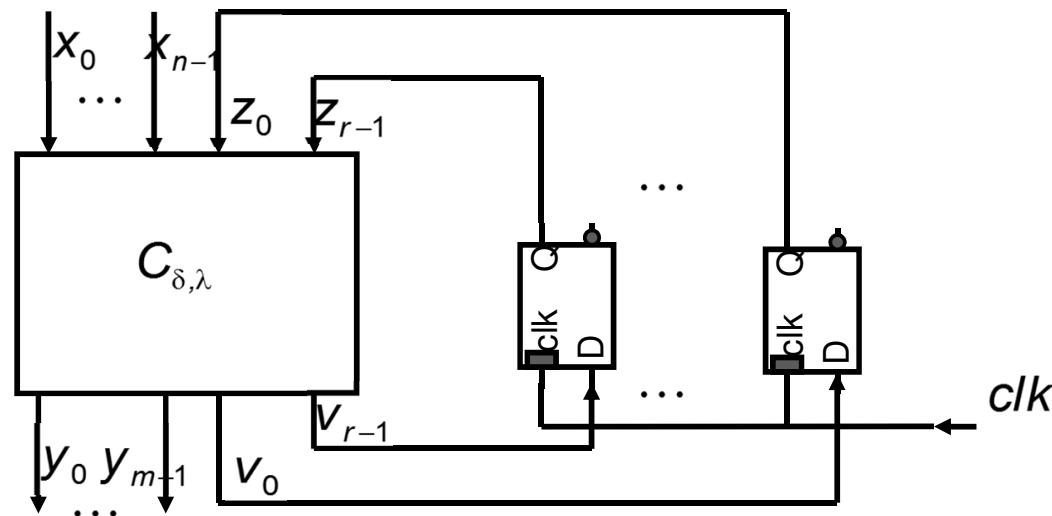
t_d Verzögerung des Latches

t_{min} minimale Reaktionszeit des Schaltkreises

$$(I) t_{CWL} \leq t_d(\text{Latch}) + t_{min}(C)$$

(t_{CWL} O Weite des Taktes)

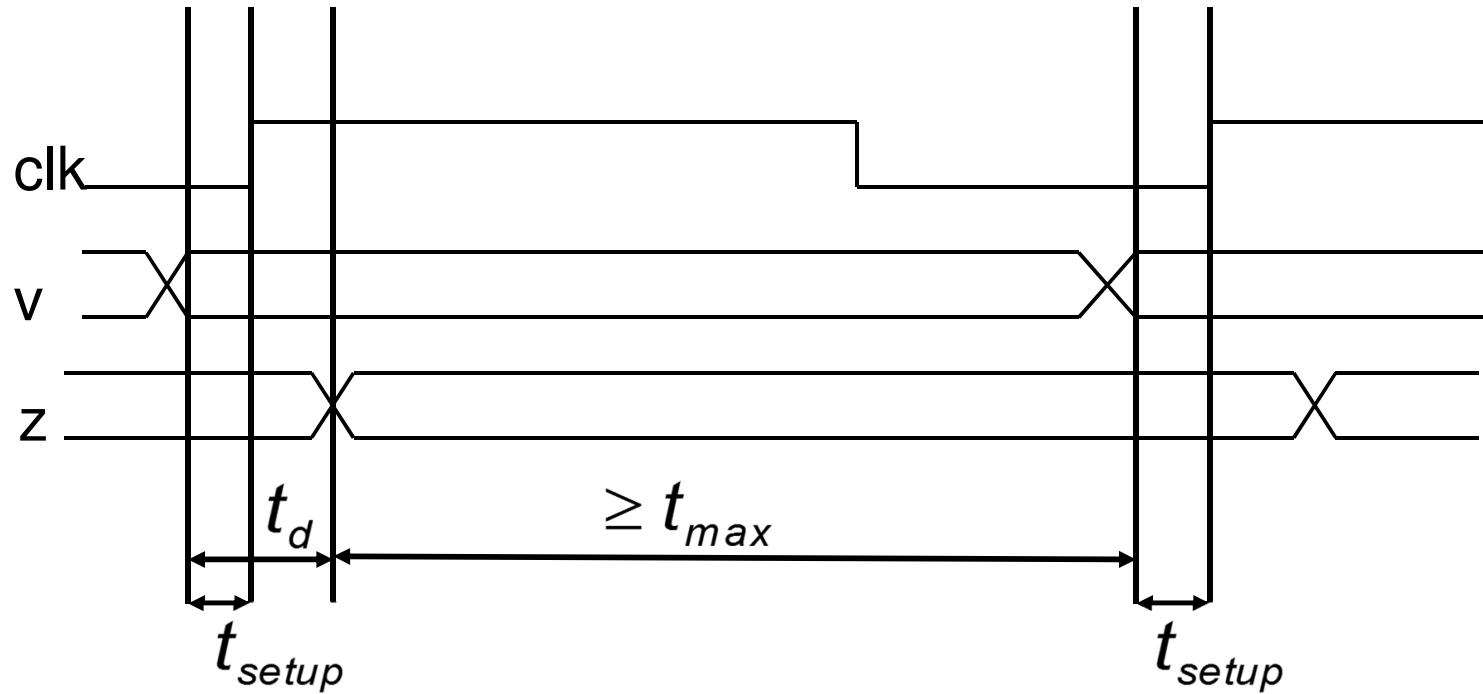
Zeitbedingungen in Schaltwerken ff



Andererseits muss v spätestens Setup Zeit vor Ende der nächsten aktiven Phase den Folgezustand von z bereitstellen, selbst wenn z erst am Ende der letzten aktiven Phase von den Latches übernommen werden konnte:

Zeitbedingungen in Schaltwerken ff

Zeitdiagramm (II)



$$t_{setup} + t_{CWH} + t_{CWL} - t_{setup} \geq t_d(\text{Latch}) + t_{max}(C)$$

$$(II) T(\text{clk}) := t_{CWH} + t_{CWL} \geq t_d(\text{Latch}) + t_{max}(C)$$

Wir nennen $T(\text{clk})$ **Taktperiode** oder **Zykluszeit**.

Zeitbedingungen in Schaltwerken ff

Die maximale Verzögerung des Schaltkreises plus die Verzögerung eines Latches bestimmen also die Länge der Taktperiode und beschränken damit die **Taktfrequenz**

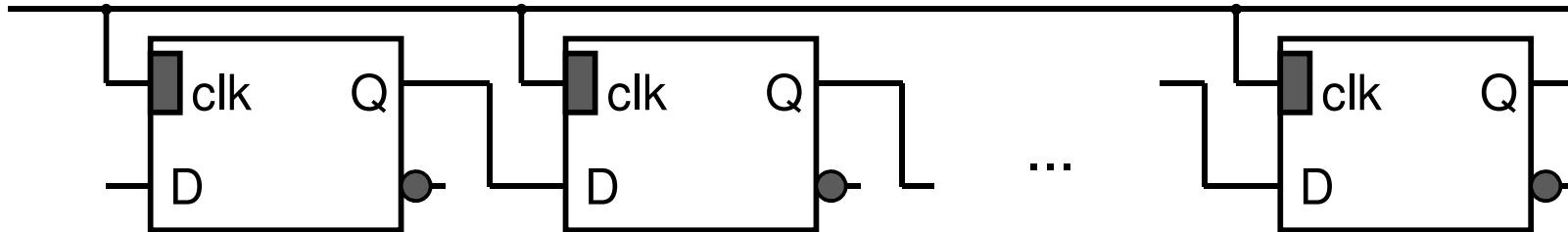
$$f_{clk} := \frac{1}{T(clk)}$$

nach oben. Hinzu kommt aber eine Maximalweitenbedingung für die aktive Taktphase. Ist z.B.

$$t_{min} + t_d(Latch) < t_{setup} (\leq t_{cwl})$$

kann das Schaltwerk gar nicht korrekt getaktet werden, weil die aktive Phase kürzer als die Setup Zeit sein müsste, um (I) einzuhalten.

Beispiel:



Hier ist $t_{min} = 0$, da $f_{\delta,\lambda}$ nur Werte neu verdrahtet.

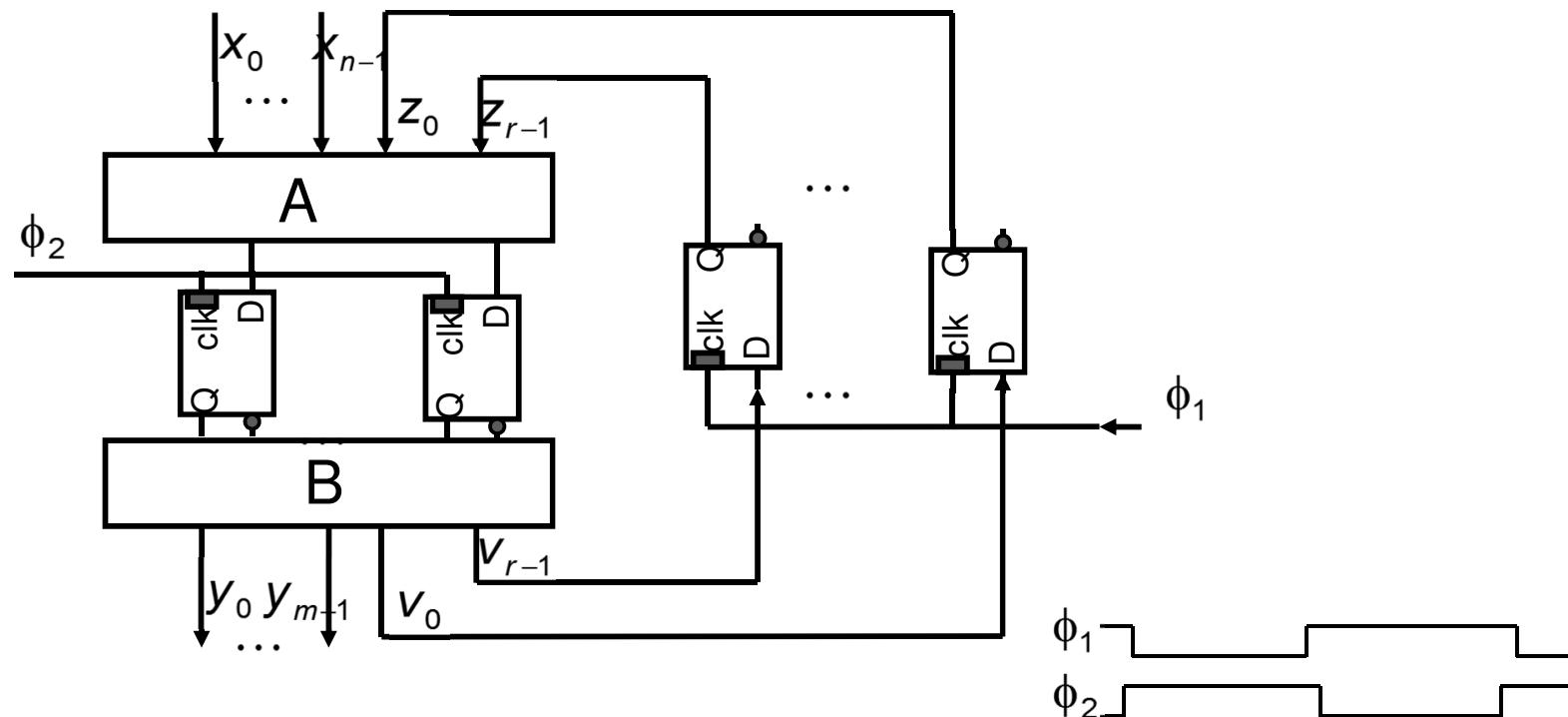
$$f_{\delta,\lambda}(z_0, \dots, z_{r-1}, x) = (z_{r-1}, x, z_0, \dots, z_{r-2})$$

Es handelt sich um einen missglückten Versuch, ein Schieberegister zu bauen. Um zu vermeiden, dass Werte in der aktiven Taktphase über mehrere Stellen rutschen, müsste man diese Phase extrem kurz halten, ggf. kürzer als die Setup Zeit eines Latches, was gar nicht geht.

Sehr kurze Taktpulse sind auch physikalisch sehr problematisch (Reflektionen, etc.). Verlängert man andererseit t_{min} , so wird auch t_{max} größer!?

Zeitbedingungen in Schaltwerken ff

Aufgrund dieser negativen Einsichten betrachten wir folgendes zustandsgesteuerte Schaltwerk, das mit zwei, in der aktiven Phase nichtüberlappenden, Takten gesteuert wird:



Zeitbedingungen in Schaltwerken ff

Diese Zerlegung widerspricht nicht unserer ursprünglichen Zerlegung, sondern macht nur die Zusammenhänge anschaulicher: C besteht aus zwei parallel arbeitenden Schaltkreisen A und B , die auf verschiedene Signale reagieren, und die Latches zerfallen in zwei unterschiedlich getaktete Gruppen.

Eine mit $\bar{\phi}_1$ übernommene Zustandsänderung muss nun durch A propagiert werden können, um von $\bar{\phi}_2$ übernommen zu werden: $t_{CWL}(\phi_1) \geq t_{max}(A) + t_d(\text{Latch})$

Analog erhalten wir

$$t_{CWL}(\phi_2) \geq t_{max}(B) + t_d(\text{Latch})$$

und den Rest durch die Bedingung

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

Zeitbedingungen in Schaltwerken ff

t_{skew} der sog. **Skew** ist eine Zeitschranke, die benötigt wird, um sicher zu sein, dass die Takte überall dort, wo sie über die Taktleitungen hinpropagiert werden, unabhängig von den Leitungslaufzeiten sicher nicht überlappend sind.

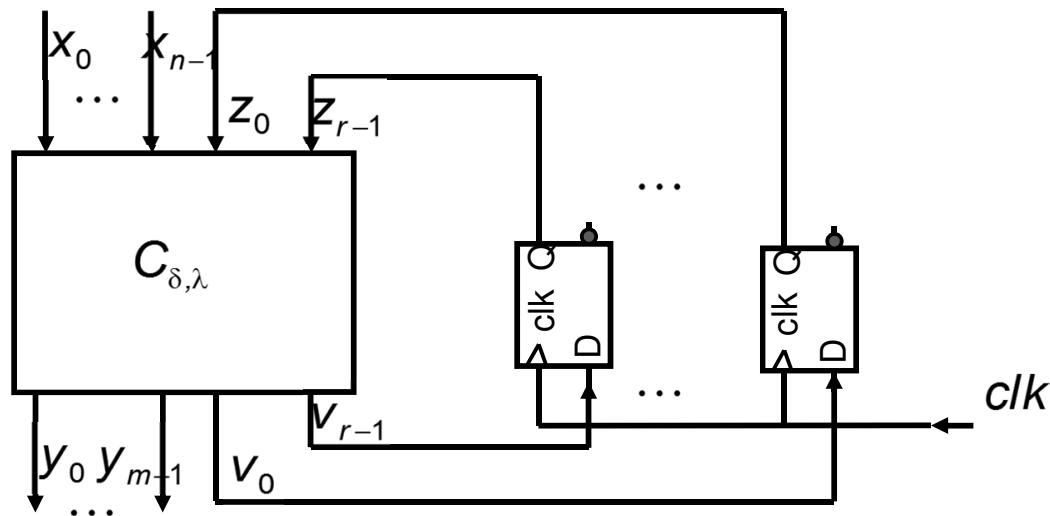
Wählt man nun die Zerlegung der Logik in zwei Teile A, B geschickt, so kann man die 0 und 1 Breiten der Takte gut balancieren.

Sehr ungeschickt wäre die Zerlegung, in der man in A die komplette Logik, in B nur Leitungen hält. Dies entspräche einem Betrieb der Latches nach dem Master Slave Prinzip, allerdings mit nichtüberlappenden Takten.

Für flankengesteuerte Schaltwerke ergeben sich aber die einfachsten Zeitbedingungen, da die Übernahme quasi punktuell erfolgt:

Zeitbedingungen in Schaltwerken ff

Betrachten wir nun zum Abschluss ein taktflankengesteuertes Schaltwerk:



In diesem Fall muss lediglich sicher sein, dass v t_{setup} vor der nächsten Flanke gültig ist. Da wir das Delay vom Setup Zeitpunkt vor der übernehmenden Flanke messen, genügt also

$$T(clk) \geq t_{max}(C_{\delta,\lambda}) + t_d(\text{Latch}) + t_{setup}$$

Entwurf von Mealy Automaten

Mealy oder Moore Automaten werden sehr oft als abstraktes Modell für die Steuerung von Abläufen eingesetzt. Wir wollen deshalb eine systematische Methode entwickeln, mit der man ausgehend von einer Definition des Automaten eine Realisierung durch einen synchronen Schaltkreis erhält.

Dazu betrachten wir zunächst eine ergonomischere Darstellung von Automaten: **Übergangsdiagramme**

Die Übergangsfunktion $\delta : Z \times X \rightarrow Z$

kann man durch eine Tabelle, aber auch anschaulich durch Pfeile in einem Diagramm darstellen. Die Knoten (Punkte) im Diagramm entsprechen den Zuständen, die Pfeile den Übergängen:

Übergangsdiagramme

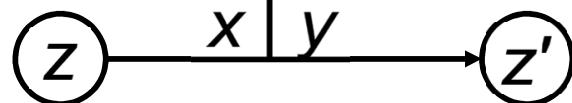


Wir können also statt einer Tabelle ein Diagramm betrachten, in dem es genau dann einen mit x markierten Pfeil von z nach z' gibt, wenn der entsprechende Übergang definiert ist. Da in Mealy Automaten

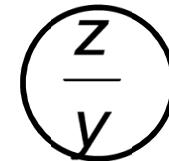
$$\delta(z, x) \text{ definiert} \Leftrightarrow \lambda(z, x) \text{ definiert}$$

können wir die Ausgabe ebenfalls an die Pfeile schreiben, bei Moore Automaten schreiben wir sie in den Zustand:

Mealy

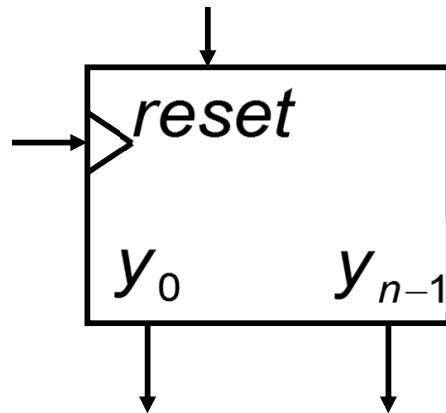


Moore



Beispiel

Ein rücksetzbarer Zähler modulo 2^n



Aufgabe:

Bei jeder aktiven Taktflanke ergebe sich

$$(y'_0, \dots, y'_{n-1}) = u_n^{-1}(u_n(y_0, \dots, y_{n-1}) + 1 \bmod 2^n)$$

$$(y'_0, \dots, y'_{n-1}) = (0, \dots, 0), \text{ falls } \text{reset} = 1. \quad \text{falls } \text{reset} = 0,$$

Beispiel ff

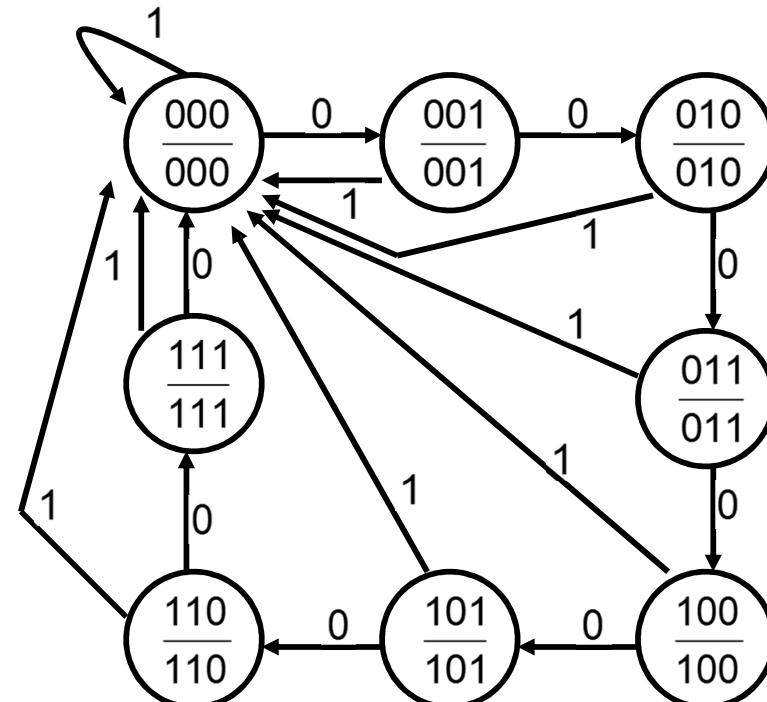
Diesen Zähler kann man sehr leicht als Moore Automaten beschreiben: Zustandsmenge: \mathcal{B}^n (y'_0, \dots, y'_{n-1})

Eingabemenge: \mathcal{B} (reset), Ausgabemenge: \mathcal{B}^n

Tabelle ($n=3$)

| 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 |
| : | : | : | : | : | : | : | : | : | : |

Übergangsdiagramm ($n=3$)



Übergangsdiagramme ff

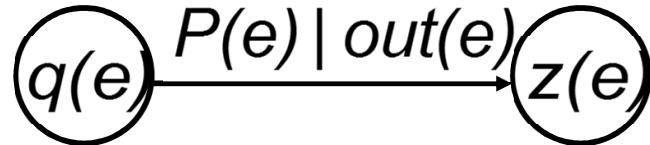
Wir vereinbaren noch ein paar Schreibweisen für Übergangsdiagramme:

Häufig haben wir mehrere verschiedenen Übergänge zwischen denselben Zuständen (Mehrfachpfeile), bei gleicher Ausgabe. Dies kann man übersichtlicher darstellen, indem man auf den Pfeil eine Bedingung (z.B. in disjunktiver Form) schreibt, unter der der Übergang stattfindet:



Übergangsdiagramme ff

Allgemein haben wir also Pfeile e , die von ihrer Quelle $q(e)$ zu ihrem Ziel $z(e)$ gerichtet und mit einer Bedingung $P(e)$ und einer Ausgabe $out(e)$ beschriftet sind.



Ein Übergang vom Zustand $q(e)$ zum Zustand $z(e)$ unter Ausgabe $out(e)$ findet statt, wenn $P(e)$ auf der Eingabe 1 wird.

Damit die Übergangsfunktion wohldefiniert wird, muss für e, e' mit $q(e) = q(e')$ gelten: $e \neq e' \Rightarrow P(e) \cdot P(e') = 0$

sonst heißt der Automat **nichtdeterministisch**

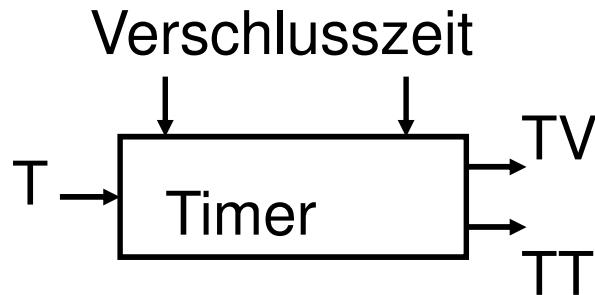
Ist $\forall z \in Z : \bigvee_{q(e)=z} P(e) = 1$ nennen wir ihn **vollständig**.

Beispiel

Wir wollen als begleitendes Beispiel die Steuerung eines primitiven Fotoapparates entwickeln:

Aufgabe

Realisiere die Verschluss- und Windersteuerung eines Fotoapparates. Gegeben sei dazu ein Timer, der nach Ablauf der eingestellten Verschlusszeit ein Signal TV setzt. Ferner setze er ein Signal TT nach Ablauf einer Transportzeitschranke. Angestoßen werde der Timer durch Setzen eines Triggersignals T.



Ferner sei ein Signal FZ gegeben, das mit 0 (low active) angeigt, dass ein Film in der Filmebene liegt.

Beispiel ff

Weiter seien die Signale

ST - Auslösertaste gedrückt ($ST=1$), sowie

TE - Vorwärtstransport beendet,
gegeben.

Zu realisieren sei eine Steuerung für Winder und Verschluss auf der Basis folgender, zu erzeugender Steuersignale:

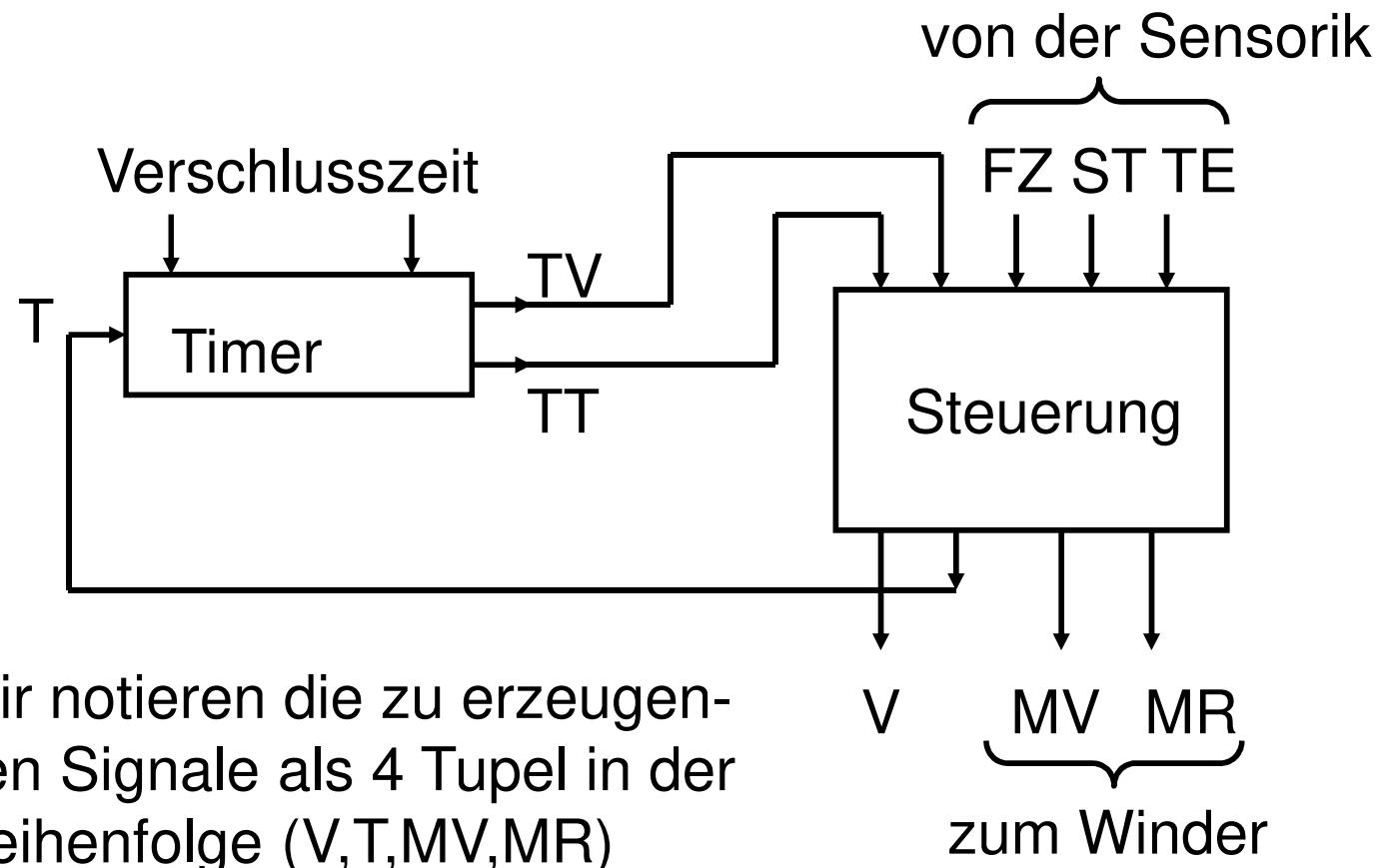
V - Verschluss öffnen ($V=1$)

MV - Motor vorwärts ($MV=1$)

MR - Motor rückwärts

Beispiel ff

Zu konstruieren ist also folgendes Teil in diesem Szenario:

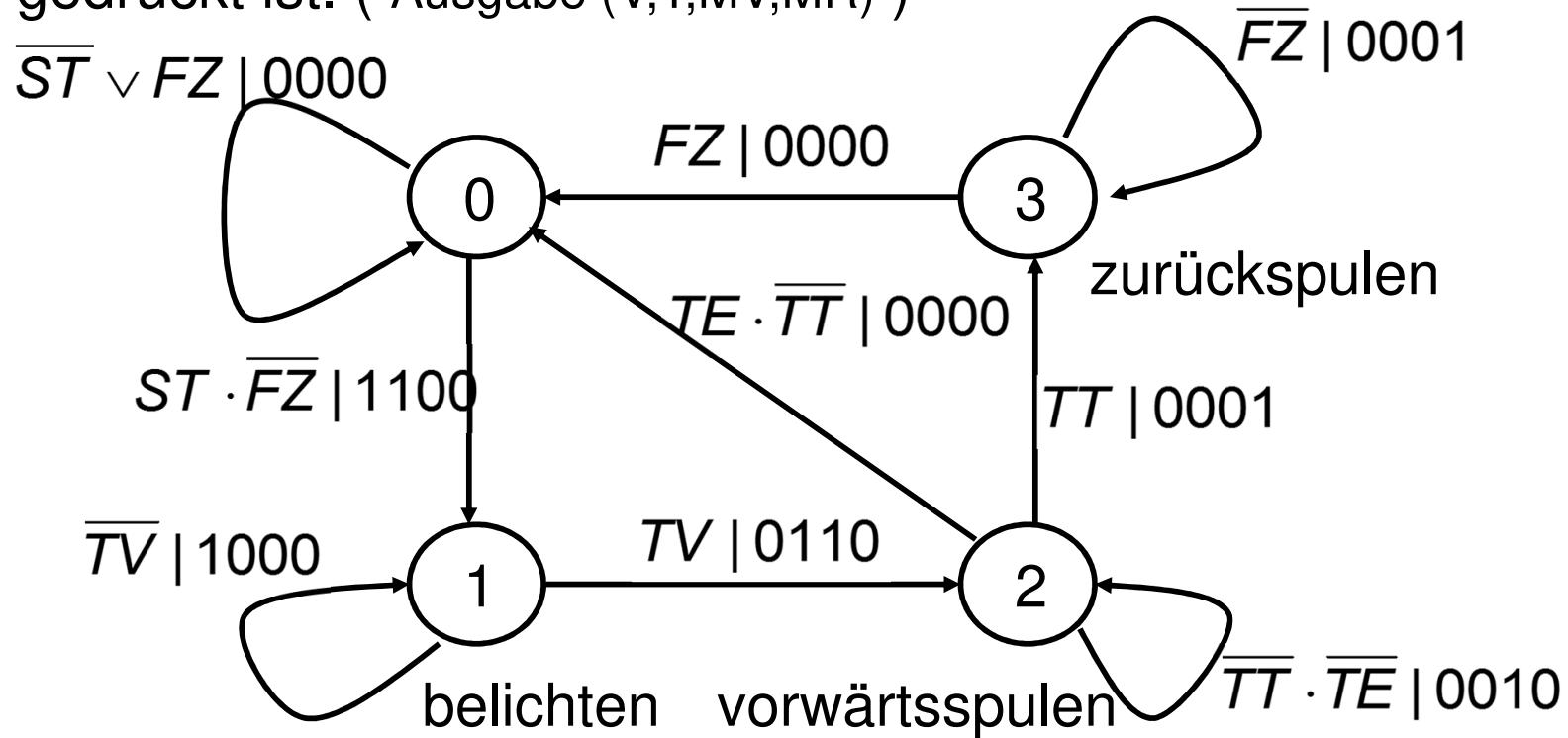


Wir notieren die zu erzeugenden Signale als 4 Tupel in der Reihenfolge (V,T,MV,MR)

Beispiel: Übergangsdiagramm

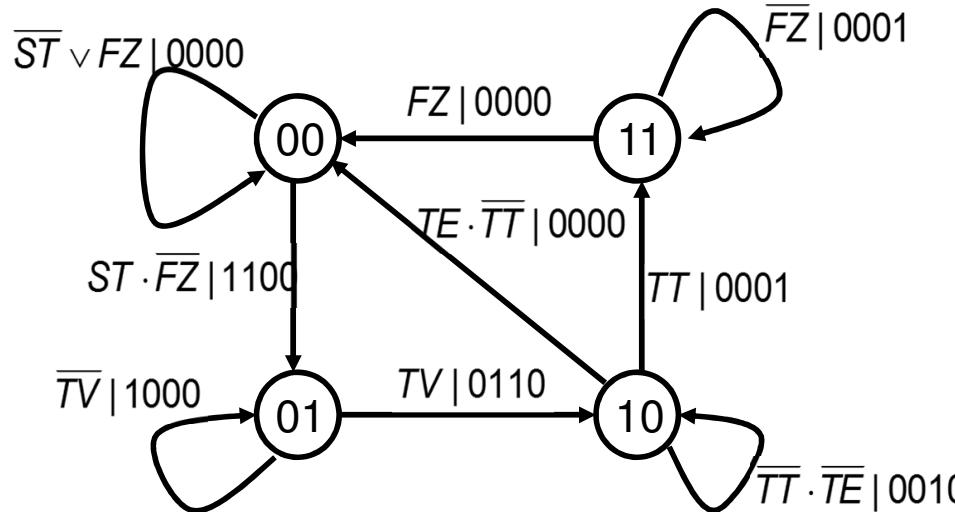
Folgendes Diagramm löst die Steuerungsaufgabe:

Wir starten dazu in einem Zustand, in dem wir verweilen, solange kein Film eingelegt oder die Starttaste nicht gedrückt ist. (Ausgabe (V,T,MV,MR))



Beispiel ff

Wir müssen nun noch die Zustände kodieren. Eine Binärkodierung nach irgendeiner Nummerierung ist eine ad hoc Möglichkeit:

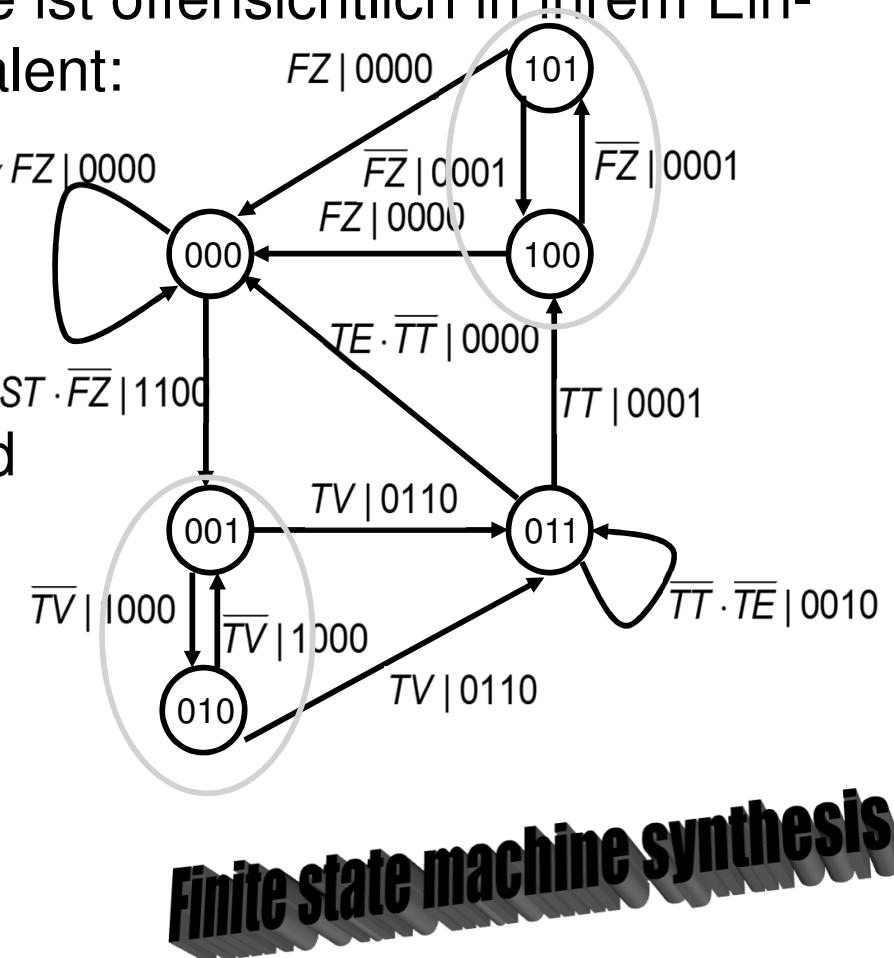


Man bemerkt aber auch, dass man sowohl bei der Kodierung, als auch bei der Struktur der Mealy Maschine selbst, einige Freiheitsgrade hat, gleiches Ein-/Ausgabe-verhalten zu realisieren:

Beispiel: äquivalenter Automat

Folgende Mealy Maschine ist offensichtlich in ihrem Ein-/Ausabeverhalten äquivalent:

Hier wurden lediglich nur die Schleifen etwas ungeschickter realisiert. Das Problem, kleinste und kostengünstigste (bzgl. ihrer Realisierung als Schaltkreis), äquivalente (kodierte) Automaten zu finden ist ebenfalls ein schweres Optimierungsproblem.



Konstruktion durch Ausdrücke

Hat man eine Tabelle oder ein Übergangsdiagramm eines Mealy Automaten gegeben, dann ist es leicht darauf boolesche Ausdrücke für die Funktion

$$f_{\delta,\lambda} = (f_{y_0}, \dots, f_{y_{m-1}}, f_{z_0}, \dots, f_{z_{r-1}})$$

wobei f_{y_i} die Funktion für Ausgabeleitung y_i

f_{z_i} die Funktion für Zustandsbit z_i ist,
abzulesen:

f_{y_i} : Sei $D = (Z, E)$ Übergangsdiagramm mit Zustandsmenge Z und Pfeilmenge E . Sei ferner $E_{y_i} \subseteq E$ die Menge der Pfeile, bei denen der Ausgabevektor an der Stelle i eine 1 hat:

$$E_{y_i} := \{ e \mid e \text{ hat Beschriftung } P(e) \mid r \text{ und } r_i = 1 \}$$

Konstruktion durch Ausdrücke ff

Dann wird f_{y_i} realisiert durch den Ausdruck

$$f_{y_i} = \bigvee_{e \in E_{y_i}} z^{q(e)} \cdot P(e)$$

wobei $z^{q(e)}$ der Minterm über z_0, \dots, z_{r-1} ist, der genau auf dem Zustand mit Code $q(e)$ 1 wird.

f_{z_i} : Sei $E_{z_i} := \{e \mid z(e) = (\dots, q_{i-1}, 1, q_{i+1}, \dots)\}$
die Menge aller Pfeile, die zu einem Zustand mit einer 1 im i -ten Bit des Zustandscodes führen. Dann ist

$$f_{z_i} = \bigvee_{e \in E_{z_i}} z^{q(e)} \cdot P(e)$$

Beispiel ff

In unserem Beispielautomaten ergeben sich folgende Ausdrücke

$$f_V = \underbrace{\bar{z}_0 \bar{z}_1 ST \cdot \bar{FZ}}_{(a)} \vee \underbrace{\bar{z}_0 z_1 \bar{T}V}_{(b)}$$

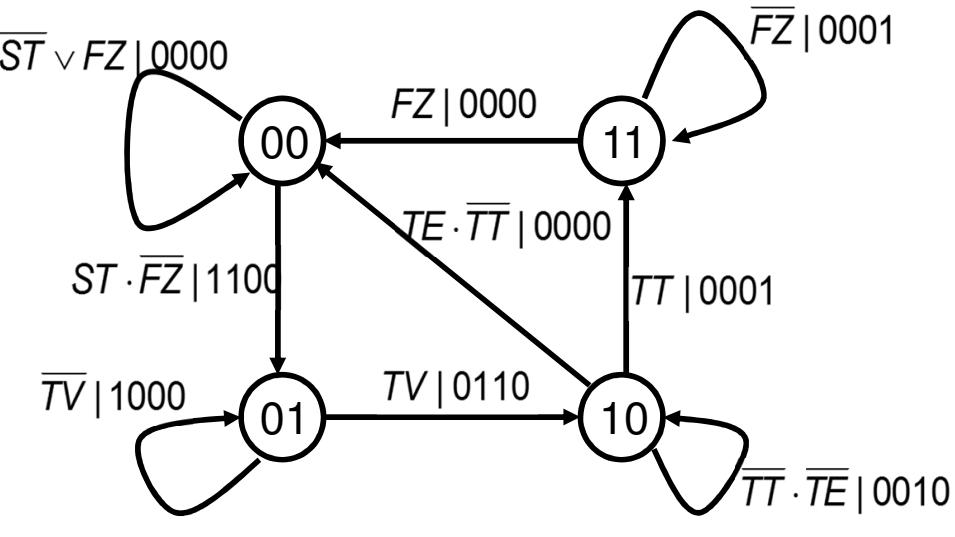
$$f_{MV} = \underbrace{\bar{z}_0 z_1 TV}_{(c)} \vee \underbrace{z_0 \bar{z}_1 \bar{TT} \cdot \bar{TE}}_{(d)}$$

$$f_{MR} = \underbrace{z_0 \bar{z}_1 TT}_{(e)} \vee \underbrace{z_0 z_1 \bar{FZ}}_{(f)}$$

$$f_T = \underbrace{\bar{z}_0 z_1 TV}_{(c)} \vee \underbrace{\bar{z}_0 \bar{z}_1 ST \cdot \bar{FZ}}_{(a)}$$

$$f_{z_0} = \underbrace{\bar{z}_0 z_1 TV}_{(c)} \vee \underbrace{z_0 \bar{z}_1 \bar{TT} \cdot \bar{TE}}_{(d)} \vee \underbrace{z_0 \bar{z}_1 TT}_{(e)} \vee \underbrace{z_0 z_1 \bar{FZ}}_{(f)}$$

$$f_{z_1} = \underbrace{z_0 \bar{z}_1 TT}_{(e)} \vee \underbrace{\bar{z}_0 \bar{z}_1 ST \cdot \bar{FZ}}_{(a)} \vee \underbrace{\bar{z}_0 z_1 \bar{TV}}_{(b)} \vee \underbrace{z_0 z_1 \bar{FZ}}_{(f)}$$



Definition in WüHDL

Moderne CAD Werkzeuge nehmen einem die eben geschilderten Aufgaben, sowie die Optimierung der Zustandskodierung und der Übergangsfunktion ab.

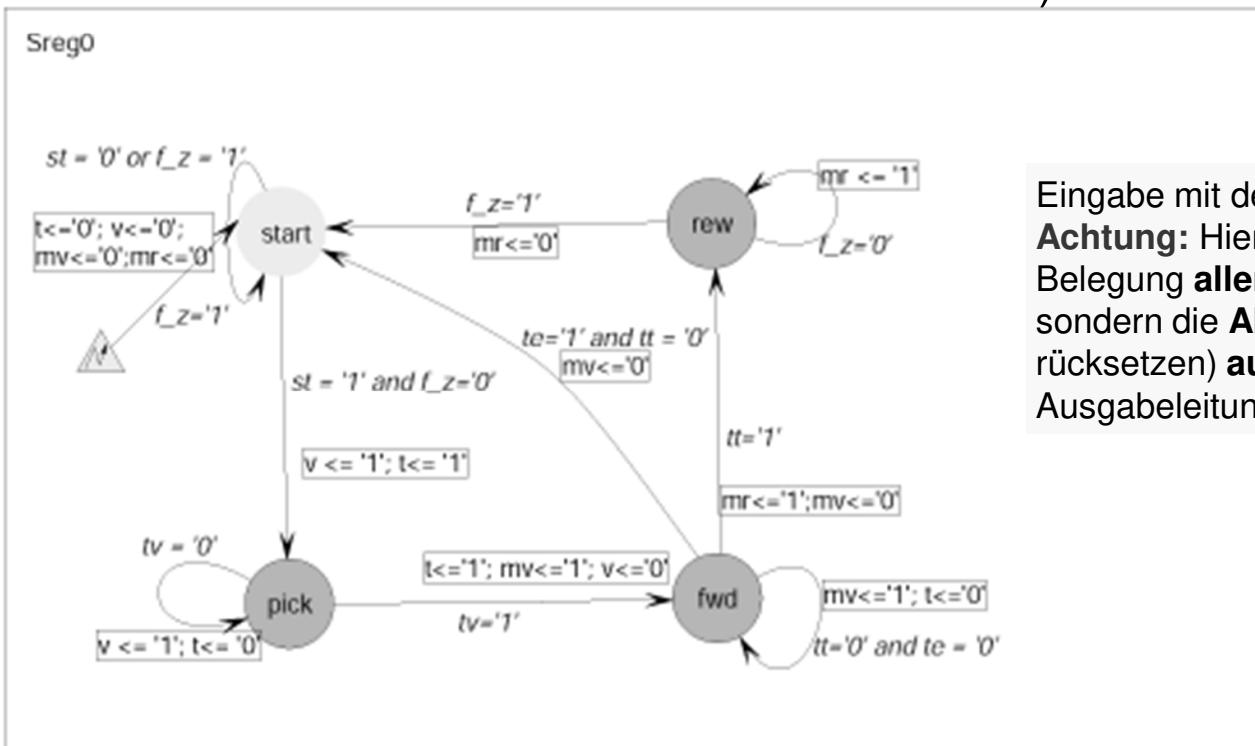
Man hat folgende Möglichkeiten:

- Direkte Definition der Mealy (Moore) Machine in WüHDL
- Benutzung eines grafischen Editors für Zustandsübergangsdiagramme.

Ein grafischer Editor setzt die Zeichnung einfach in korrespondierenden WüHDL Code um. Bei der direkten Definition hat man die Möglichkeit den Automaten **explizit** durch Falldiskussion über die Zustandsmenge oder implizit* durch den Kontrollfluss des Programmes zu definieren.

*) implizit wird selten unterstützt

Beispiel: Grafische Eingabe



Eingabe mit dem ALDEC FSM Editor

Achtung: Hier zeigt man nicht die Belegung aller Ausgabeeleitungen an, sondern die **Aktionen** (setzen, rücksetzen) **auf** den Ausgabeeleitungen!

Eingabe mit dem ALDEC FSM Editor

Achtung: Hier zeigt man nicht die Belegung aller Ausgabeeleitungen an, sondern die **Aktionen** (setzen, rücksetzen) **auf** den Ausgabeeleitungen!

Beispiel: generierter WüHDL Code

```
-- File          : c:\...\fotosteuerung\compile\steuerung.vhd
-- Generated    : 04/05/02 18:10:31
-- From         : c:/.../fotosteuerung/src/steuerung.asf
-- By          : FSM2VHDL ver. 3.0.4.1

entity steuerung is
    port (
        CLK: in BIT;
        f_z: in BIT;
        st: in BIT;
        te: in BIT;
        tt: in BIT;
        tv: in BIT;
        mr: out BIT;
        mv: out BIT;
        t: out BIT;
        v: out BIT);
end;
```

Beispiel: generierter WüHDL Code

```
architecture generated of steuerung is
```

```
-- SYMBOLIC ENCODED state machine: Sreg0  
type Sreg0 type is (pick, fwd, rew, start);  
signal Sreg0: Sreg0_type;
```

```
begin  
-- concurrent signals assignments  
-- diagram ACTION
```

```
---
```

```
-- Machine: Sreg0  
-----  
Sreg0_machine: process (CLK, f_z)  
begin  
if f_z='1' then  
    Sreg0 <= start;  
    -- Set default values for ..  
elsif CLK'event and CLK = '1' then  
    -- Set default values ...
```

Zustandsregister (stateregister) über Enumerationstyp

Automat wird durch einen Prozess definiert, der sensitiv auf den Takt, und das asynchrone Rücksetzsignal ist.

f_z wurde als asynchrones Rücksetzsignal benutzt.

0/1 Flankengesteuertes Zustandsregister

Beispiel: generierter WüHDL Code

```
case Sreg0 is
    when pick =>
        if tv='1' then
            Sreg0 <= fwd;
        elsif tv = '0' then
            Sreg0 <= pick;
            v <= '1';
        end if;
    when fwd =>
        if te='1' and tt = '0' then
            Sreg0 <= start;
        elsif tt='1' then
            Sreg0 <= rew;
        elsif tt='0' and te = '0' then
            Sreg0 <= fwd;
        end if;
    ....
```

Fallunterscheidungen nach Zustand

und dann nach Übergängen

Beispiel: generierter WüHDL Code

```
.... end case;  
end if;  
end process;  
-- signal assignment statements for combinatorial outputs  
  
t_assignment:  
t <= '1' when (Sreg0 = start and (st = '1' and f_z='0')) else  
    '1' when (Sreg0 = pick and tv='1') else  
    '0' when (Sreg0 = pick and tv = '0') else  
    '0' when (Sreg0 = fwd and (tt='0' and te = '0')) else  
    '0';  
v_assignment:  
v <= '1' when (Sreg0 = start and (st = '1' and f_z='0')) else  
    '0' when (Sreg0 = pick and tv='1') else  
    '1' when (Sreg0 = pick and tv = '0') else  
    '0';  
mv_assignment:  
mv <= '1' when (Sreg0 = pick and tv='1') else  
    '0' when (Sreg0 = fwd and (te='1' and tt = '0')) else  
    '0' when (Sreg0 = fwd and tt='1') else  
    '1' when (Sreg0 = fwd and (tt='0' and te = '0')) else  
    '0';  
....
```

Ausgaben als concurrent signal assignments über Zustand und Bedingung

2.3 Asynchrone Schaltkreise

Zur Vorlesung Rechenanlagen

SS 2019



Asynchrone Schaltkreise

Wir wollen Schaltkreise über kombinatorischen Bausteinen betrachten, die weder rückkopplungsfrei sein, noch auf jedem Zyklus ein Latch haben müssen.

Solche Schaltkreise bilden eine echte Obermenge der kombinatorischen Schaltkreise. Sie können Gedächtnis haben, ohne explizit getaktet zu werden. Wir nennen solche Schaltkreise daher auch **asynchrone** Schaltkreise.

Problem:

Welche Belegungen können diese Schaltungen erreichen und wie analysiert man sie?

Delayunabhängige Analyse

Da die Verzögerungszeiten im allgemeinen nie exakt bekannt sind, wollen wir eine robuste Analyse vornehmen, d.h. wir wollen untersuchen, welche Belegungszustände unabhängig von der Größe der Verzögerungszeit einzelner Bausteine erreicht werden können. Man kann dazu zwei Modelle betrachten:

- das **Bausteinverzögerungsmodell** (Muller model)
- das **Leistungsverzögerungsmodell** (Huffman model)

Leitungs- vs. Bausteinverzögerung

Das Leitungsverzögerungsmodell beruht darauf, dass man jeder Leitung (jedem Zweig eines Signals) eine beliebige Verzögerung größer 0 zuordnen darf. Eine Belegung r ist dann von einer Belegung q erreichbar, wenn man Verzögerungen so zuordnen kann, dass r nach Schalten aller Zweige mit der zugeordneten Verzögerung von q erreicht wird.

Das Bausteinverzögerungsmodell beruht darauf, dass man jedem Bausteknausgang (jedem Signaltreiber, bzw. jedem Signal) eine beliebige Verzögerung größer 0 zuordnen darf. Eine Belegung r ist dann von einer Belegung q erreichbar, wenn man Verzögerungen so zuordnen kann, dass r nach Schalten aller Signale mit der zugeordneten Verzögerung von q erreicht wird.

Leitungs- vs. Bausteinverzögerung

Das Leitungsverzögerungsmodell ist offensichtlich allgemeiner. Grundsätzlich kann man es aber mit dem Bausteinverzögerungsmodell simulieren, wenn man auf jeden Zweig eines Signals mit mehreren Anschlüssen einen Baustein legt, der nur die Identität berechnet.

Wir beschränken unsere Betrachtungen daher auf das einfachere Bausteinverzögerungsmodell.

Die erste Frage, die sich nun stellt, ist die, wann überhaupt ein Schaltvorgang bei einer Belegung $p = (p_1, \dots, p_m)$ der Signale stattfinden kann.

Delayunabhängige Analyse ff

Definition

Eine Belegung p' geht aus einer Belegung p durch einen Schaltvorgang hervor genau dann, wenn

$$p'_i = p_i \text{ für } i \neq s \text{ und } p'_s = C[s](p) = \overline{p_s}$$

Wir schreiben dann auch $p \succ p'$ (p kann nach p' schalten)

Gilt $p \succ p'$, so korrigiert der Schaltkreis offensichtlich eine in χ_C verletzte Gleichung der Form $s \equiv C[s]$. Also gilt offenbar

Lemma

Eine Belegung p ist stabil $\Leftrightarrow \forall p' \in \mathcal{B}^m: \neg(p \succ p')$

Delayunabhängige Analyse ff

Bemerkung:

Beobachtet man einen Schaltkreis ausgehend von einer instabilen Belegung p in der Zeit, dann wird man o.E. stets eine Folge mit

$$p = p^{(0)} \succ p^{(1)} \succ \dots \succ p^{(n)} \succ \dots$$

beobachten. Welche Folge man konkret beobachtet, hängt von den Schaltzeiten ab. Die benachbarten Elemente der Folge unterscheiden sich stets in genau einem Bit s_i , d.h. es gibt zu jedem i ein s_i , mit

$$p_{s_i}^{(i)} \neq p_{s_i}^{(i+1)} = C[s_i](p^{(i)})$$

Die Tatsache, dass wir keine gleichzeitigen Schaltvorgänge betrachten, schadet nicht, weil es „bis auf Messfehler“ keine Gleichzeitigkeit gibt.

Delayunabhängige Analyse ff

Bemerkung ff:

Solche Folgen $p = p^{(0)} \succ p^{(1)} \succ \dots \succ p^{(n)} \succ \dots$ können in einer stabilen Belegung enden, sie können aber auch endlos sein. Um herauszufinden, ob eine Folge endlos wird, genügt es aber, alle Folgen der Länge $\leq 2^{\#S}$ zu untersuchen, da es höchstens $2^{\#S}$ Belegungen gibt, sich also nach $2^{\#S}$ Schritten eine Belegung wiederholen muss.

$$p^{(0)} \succ p^{(1)} \succ \dots \succ p^{(i)} \succ \dots \succ p^{(j)}$$

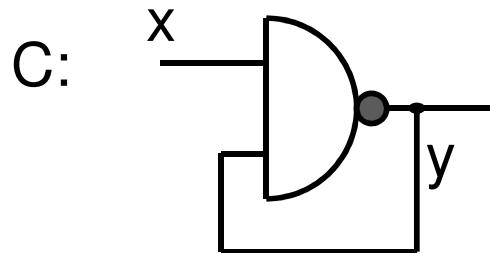


=

Wiederholt sich erstmals eine Belegung, kann man diese Teilfolge immer wieder anhängen. Wir sagen dann auch:
„der Schaltkreis schwingt“.

Beispiel:

Wir betrachten



$$C[y] = \overline{xy} \quad \chi_C = (y \equiv \overline{xy})$$

Falldiskussionen:

1. $(x,y) = (0,0)$: Instabil, da $0 \neq 1 = \overline{0 \cdot 0}$ Also $(0,0) \succ (0,1)$
2. $(x,y) = (0,1)$: Stabil, da $1 = \overline{0 \cdot 1}$
3. $(x,y) = (1,0)$: Instabil, da $0 \neq 1 = \overline{1 \cdot 0}$ Also $(1,0) \succ (1,1)$
4. $(x,y) = (1,1)$: Instabil, da $1 \neq 0 = \overline{1 \cdot 1}$ Also $(1,1) \succ (1,0)$

Damit liefert für $x = 1$ $(1,0) \succ (1,1) \succ (1,0) \succ \dots$

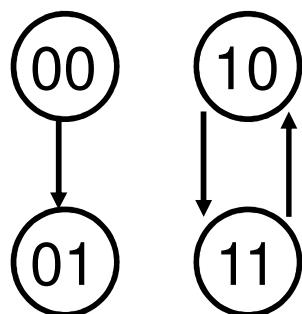
eine endlose Folge von Schaltvorgängen.

Belegungsübergangsdiagramm

Wir können die möglichen Schaltverhalten eines Schaltkreises leicht in Form von Eigenschaften eines (ggf. gigantischen) Diagrammes (im Rechner) ablesen, des **Belegungsübergangsdiagramms**:

- **Punkte** (Knoten) im Diagramm: Alle Belegungen
- **Pfeile** (Kanten) im Diagramm: von p nach q : $\Leftrightarrow p \succ q$

Im Beispiel:



Es gibt zwei bemerkenswerte Dinge:

- Punkte, die keinen ausgehenden Pfeile haben, sog. **Senken**. Sie entsprechen den stabilen Belegungen.
- **Zyklen**. Sie entsprechen endlosen Schaltfolgen.

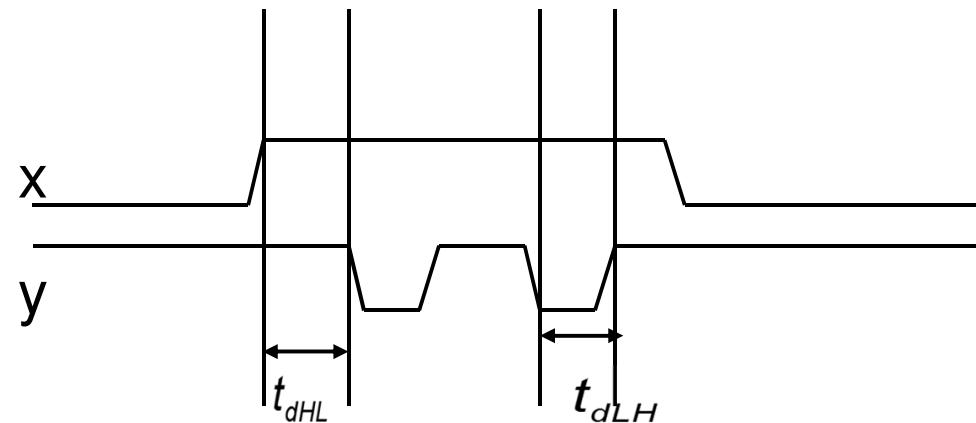
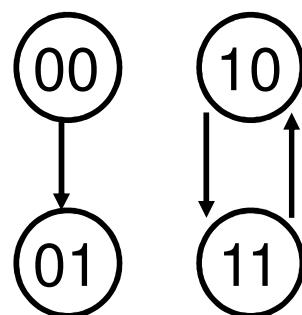
Beispiel ff

Wir erkennen also in unserem Beispiel zwei Verhaltensweisen:

Für $x = 0$ bewegt sich die Schaltung nach einer gewissen Zeit sicher in einen stabilen Zustand.

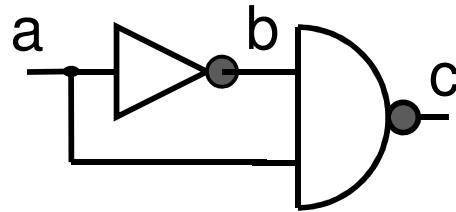
Für $x = 1$ beginnt sie zu schwingen, wobei die Periode und Form von den konkreten Schaltzeiten des Bausteins abhängt.

Zeitdiagramm:



Beispiel:

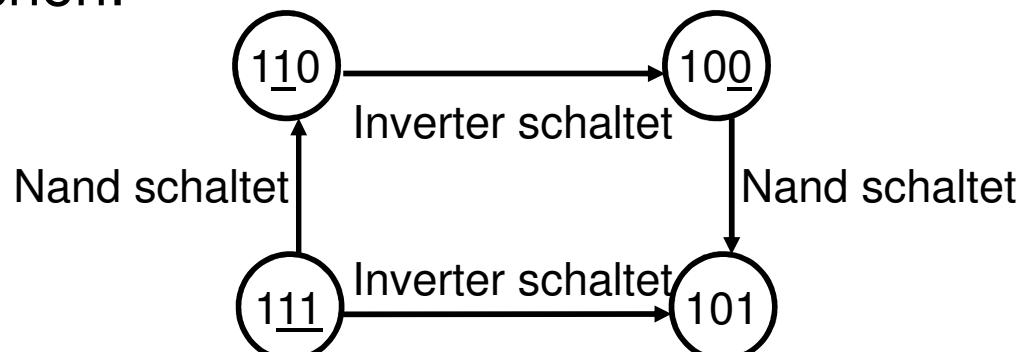
Wir betrachten den kombinatorischen Schaltkreis



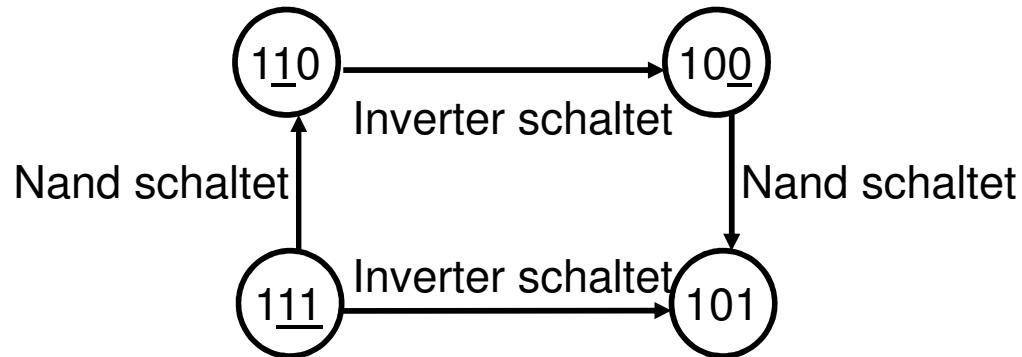
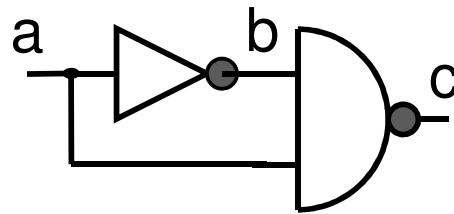
Statisch ergibt sich hier die Zuordnung

$$\begin{aligned}F[c] &= C[c](b = F[b], a) \\&= \overline{F[b] \cdot a} = \overline{\overline{a}a} = \overline{0} = 1\end{aligned}$$

Also eine AI Bundy Realisierung der Konstanten 1.
Analysieren wir nun einmal die möglichen Einschwingvorgänge, die von der instabilen Belegung $(1, 1, 1)$ erreichbar sind. Wir zeichnen die Signalwerte von Signalen s mit $p_s \neq C[s](p)$ unterstrichen:



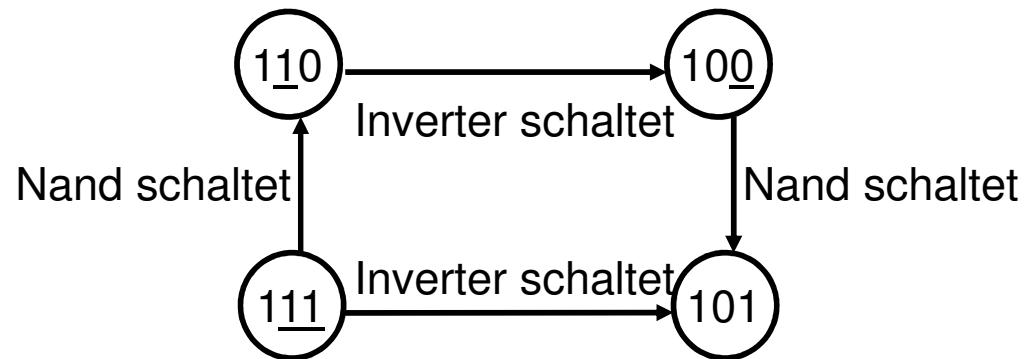
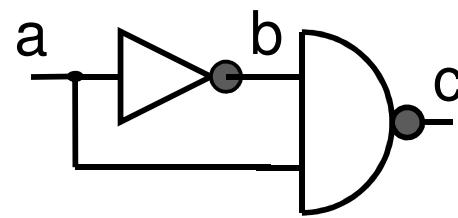
Beispiel ff



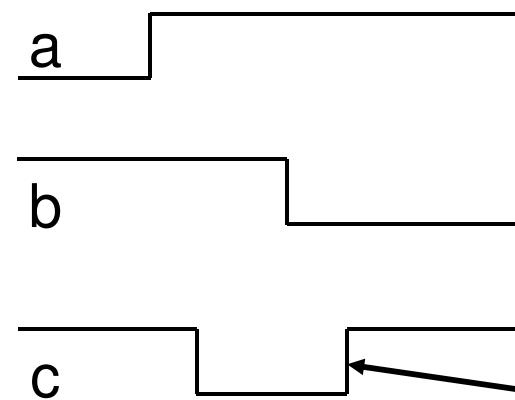
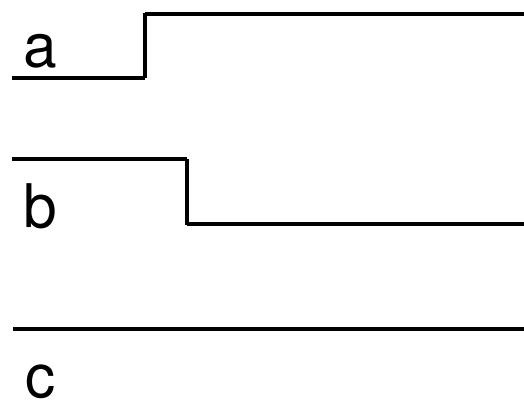
Wird die Schaltung, etwa durch Schalten des Eingangs a im stabilen Zustand 011 nach 111 gebracht, dann gibt es je nach Laufzeitzuordnungen zwei Verhaltensweisen:

- Nand schaltet schneller als der Inverter: dann wird der obere Weg mit 3 Schaltvorgängen genommen.
- Inverter schaltet schneller als das Nand: dann wird der stabile Folgezustand direkt mit einem Schaltvorgang erreicht.

Beispiel ff



Zeitdiagramme:



$111 \succ 101$

$111 \succ 110 \succ 100 \succ 101$

Anmerkungen

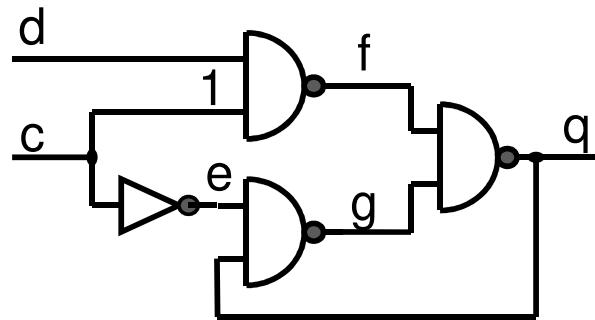
Das Diagramm zeigt uns also alle möglichen Einschwingvorgänge auf. Will man sicherstellen, dass eine Schaltung keine unerwünschten Belegungen annimmt, kann man dies durch erschöpfendes Durchsuchen im Diagramm nachprüfen.

Die Diagramme werden riesig, weil die Zahl der Punkte exponentiell mit der Zahl der Signale wächst. Interessiert man sich nur für gewisse Einschwingvorgänge, kann man das Diagramm wie im letzten Beispiel von einem interessierenden Belegungszustand her aufbauen und durchsuchen. Besser ist natürlich eine maschinelle Durchsuchung.

Weiß man mehr über Verzögerungszeiten, kann man auch Pfade im Diagramm durch dieses Wissen ausschliessen.

Noch ein Beispiel: Problemlatch

Wir untersuchen folgende, sogar in Lehrbüchern zu findende Schaltung für ein zustandsgesteuertes D-Latch:



$c = 0$: dann ist der Zustand
 $(c, d, e, f, g, q) = (0, d, 1, 1, \bar{q}, q)$
stabil für jedes d und q

$c = 1$: dann ist der Zustand
 $(c, d, e, f, g, q) = (1, d, 0, \bar{d}, 1, d)$
stabil für jedes d (das
Latch ist transparent)

$$C[e] = \bar{c} \quad C[f] = \overline{cd}$$

$$C[g] = \overline{eq} \quad C[q] = \overline{fg}$$

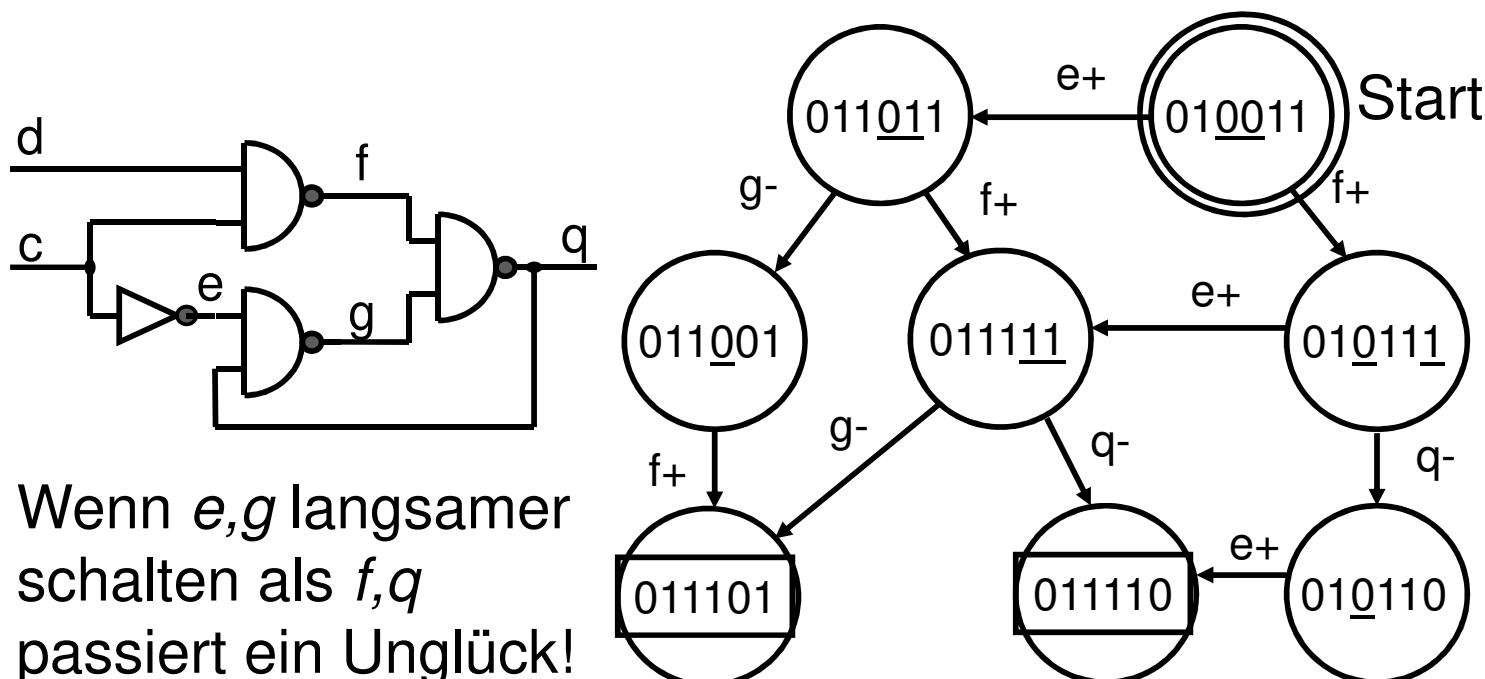
$$\chi_c = (e \equiv \bar{c})(f \equiv \overline{cd})(g \equiv \overline{eq})(q \equiv \overline{fg})$$

$$\chi_c(0, d, 1, 1, \bar{q}, q) = (1 \equiv \bar{0})(1 \equiv \overline{0d})(\bar{q} \equiv \overline{1q})(q \equiv \overline{1q})$$

$$= (1 \equiv 1)(1 \equiv 1)(\bar{q} \equiv \bar{q})(q \equiv \bar{q}) = 1 \quad (c=1 \text{ analog!})$$

Problemlatch ff

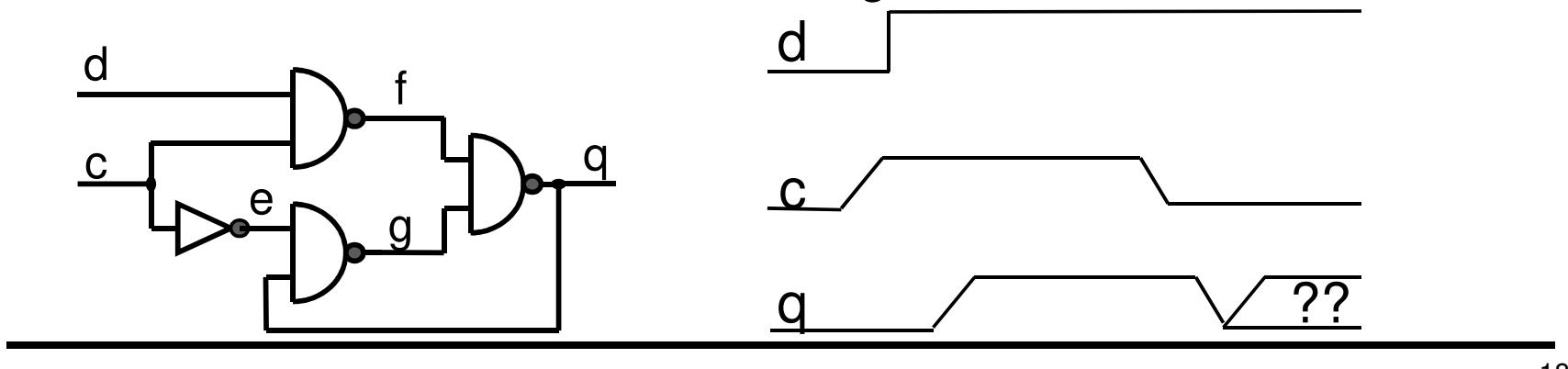
Untersuchen wir nun mal das Einschwingverhalten des Latches bei der Übernahme einer 1, wenn q schon 1 war, d.h. im stabilen Zustand $(c,d,e,f,g,q) = (1,1,0,0,1,1)$ lassen wir c auf 0 fallen und erreichen den instabilen Zustand $(0,1,\underline{0},\underline{0},1,1)$, der zur Speicherung der 1 führen soll:



Problemlatch ff

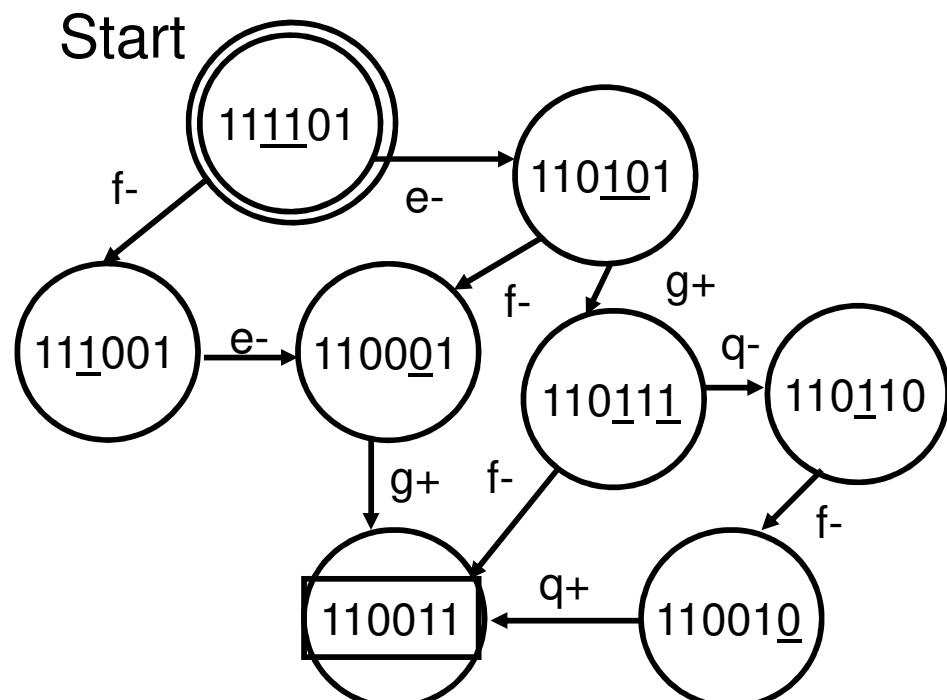
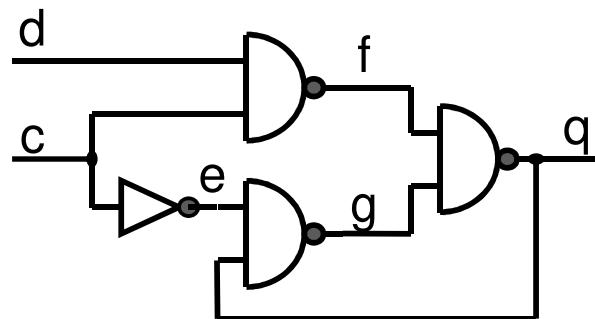
Problem: Wenn der Inverter e gefolgt vom Nand g langsamer schaltet als die beiden Nands f, q , übernimmt dieses Latch sogar eine 0 statt einer $1!$ Selbst wenn ein Inverter schneller als ein Nand schaltet, was in der Praxis der Fall ist, kann es dadurch bei ungünstigen Leitungslaufzeiten oder Streuungen der Verzögerungszeiten zu einer falschen Übernahme, zumindest zu einem leichten Pegelinbruch am Ausgang q kommen.

Zeitdiagramm:



Problemlatch ff

Auch beim Übergang in den transparenten Zustand kommt es zu Hazards, auch wenn schon $q=d=1$ war: wir lassen dazu im stabilen Zustand $(c,d,e,f,g,q) = (0,1,1,1,0,1)$ den Takt c auf 1 gehen in den instabilen Zustand $(1,1,1,1,0,1)$

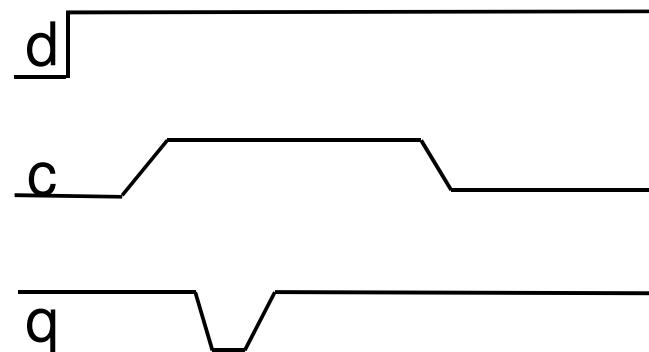
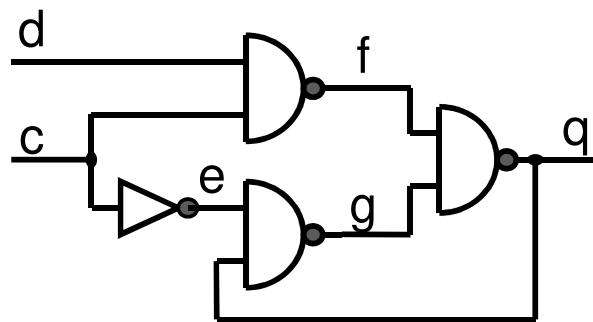


Wenn f langsamer schaltet als e, g, q gibt es einen Hazard an q .

Problemlatch ff

Problem: Wenn der Inverter e gefolgt vom Nand g und q schneller schaltet als das Nand f , bleibt der Ausgang, obwohl ein Übergang von 1 auf 1 erfolgen sollte, kurze Zeit nicht stabil. Dies schadet allerdings weniger, weil der erreichte Zielzustand korrekt ist, und das Latch ohnehin in der transparenten Phase ist, d.h. man nicht unbedingt Stabilität erwartet.

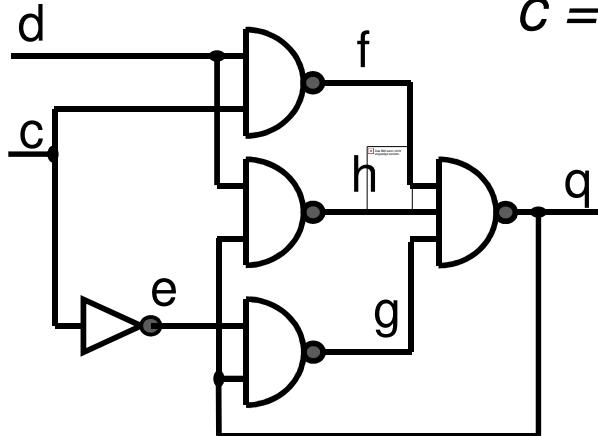
Zeitdiagramm:



Beispiel: hazardfreies Latch

Folgende Schaltung beseitigt das Problem:

Ein hazardfreies zustandsgesteuertes D-Latch



$c = 0$: dann ist der Zustand

$$(c,d,e,f,g,h,q) = (0,d,1,1,\bar{q},\overline{dq},q)$$

stabil für jedes d und q , denn

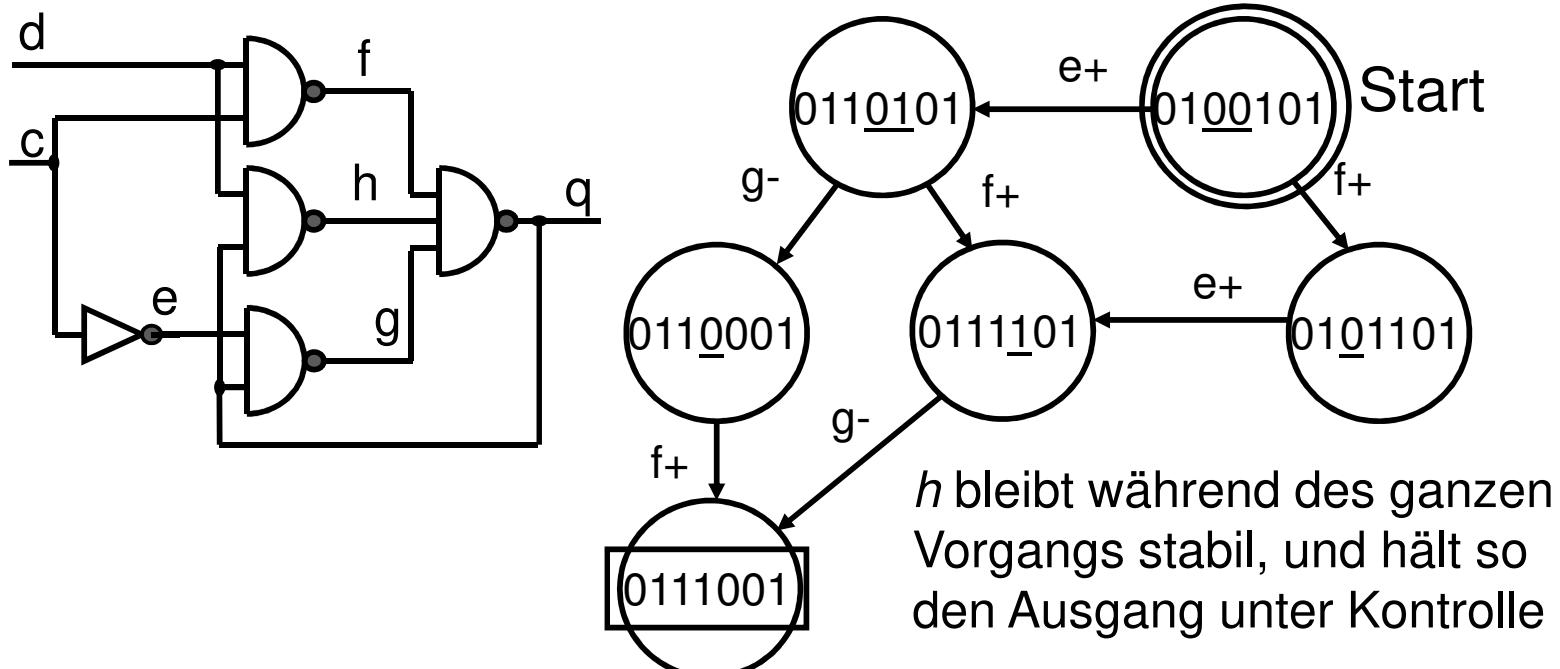
$$(q \equiv \overline{\bar{q} dq}) = (q \equiv q \vee dq)$$

$$= (q \equiv q) = 1$$

$c = 1$: dann ist der Zustand $(c,d,e,f,g,h,q) = (1,d,0,\overline{d},1,\overline{d},d)$
stabil für jedes d (das Latch ist transparent)

Hazardfreies Latch ff

Untersuchen wir nun wieder den Problemfall einer Übernahme einer 1, wenn q schon 1 war, d.h. im stabilen Zustand $(c,d,e,f,g,h,q) = (1,1,0,0,1,0,1)$ lassen wir c auf 0 fallen und erreichen den instabilen Zustand $(0,1,\underline{0},\underline{0},1,0,1)$, der zur Speicherung der 1 führen soll:

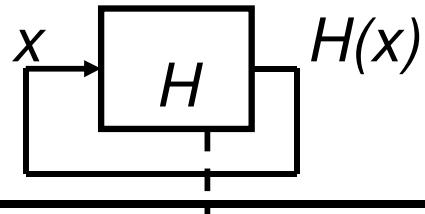


Metastabilität

In bistabilen Schaltungen kann es durch Einschwingvorgänge bei der Übernahme wie in unserem Beispiel, oder auch durch Synchronisationsprobleme mit externen Signalen zu **Metastabilen Zuständen** kommen, d.h. länger anhaltende Belegungen, die weder als 0 noch als 1 interpretiert werden können.

Metastabilität ist grundsätzlich nicht vermeidbar:

Grund: Speicherelemente halten ihren Zustand auf einer Rückkopplungsschleife. Diese können wir beschreiben durch ein Übertragungssystem H , wie in folgendem Bild



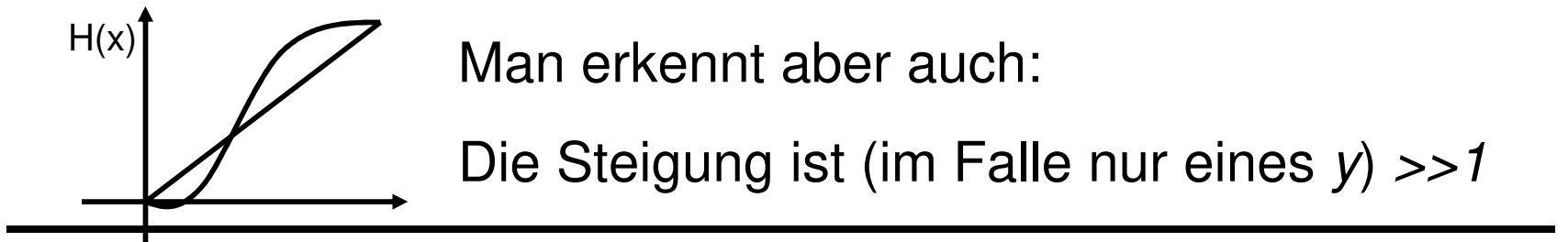
Stabiler Zustand: $x = H(x)$

Metastabilität ff

In den stabilen Zuständen $H(x)=x=0$ und $H(x)=x=1$ sollten sich Störungen an x nur sehr gering an $H(x)$ auswirken, d.h. es sollte gelten:

$$\frac{\partial H}{\partial x}(0) \ll 1, \text{ sowie } \frac{\partial H}{\partial x}(1) \ll 1$$

Die Übertragungsfunktion H der Schleife bildet also eine stetige Kurve, die mit Steigung $\ll 1$ in $(0,0)$ beginnt und mit Steigung $\ll 1$ im Punkt $(1,1)$ endet. Diese Kurve schneidet die Diagonale in mindestens einem Punkt (y,y) . D.h. y ist ein **unvermeidbarer metastabiler Zustand** der Schleife.



Metastabilität ff

Metastabile Zustände sind also grundsätzlich nicht vermeidbar. Die interessantere Frage ist aber, wann sie schädlich sind, und wie man Schaden vermeiden kann.

Sie sind offenbar nur dann schädlich, wenn sie als Werte abgefragt werden, d.h. die zentrale Frage ist, wie lange ein solcher metastabiler Zustand anhalten kann.

Wir haben gesehen, dass die Steigung in einem metastabilen Zustand sehr steil ist. Die kleinste Störung an y bewirkt also dramatische Veränderungen an $H(y)$ und damit an y selbst, so dass das System sehr schnell wieder in einen der stabilen Zustände kippt. Je größer diese Steigung ist, umso unwahrscheinlicher ist eine längere Verweildauer im metastabilen Zustand:

$$p(y \text{ hält länger als } t) \approx e^{-\alpha t}$$

Metastabilität ff

Fazit

Metastabilität ist nicht vermeidbar. Vor allem dann, wenn man asynchrone externe Signale hat, deren Änderungszeitpunkte man nicht beeinflussen kann, kann es zu metastabilen Zuständen von Latches kommen.

Maßnahmen

Konstruiere Bauteile so, dass die Wahrscheinlichkeit eines länger andauernden metastabilen Zustands sehr gering wird.

Lege Zeitpunkte, zu denen metastabile Zustände entstehen können, möglichst weit weg von Zeitpunkten, zu denen diese Einfluss auf das Systemverhalten nehmen können.

Hazards

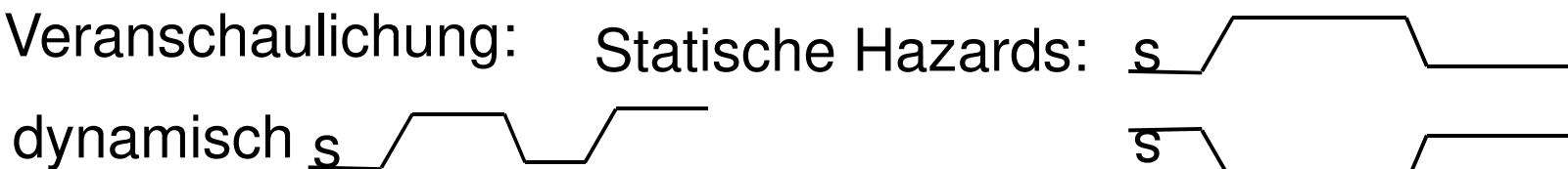
Definition

Sei C ein kombinatorischer Schaltkreis und s ein Signal in C . Seien p, q zwei aufeinanderfolgende Eingabemuster.

s hat einen (statischen) **Hazard** unter p, q genau dann, wenn $F[s](p) = F[s](q)$, die Schaltung aber beim Einschwingen instabil an s ist.

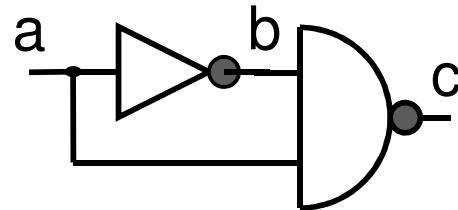
s hat einen **dynamischen Hazard** unter p, q genau dann, wenn $F[s](p) \neq F[s](q)$, aber s mehr als einer Änderung unterliegt.

Veranschaulichung:



Hazards ff

Wir haben schon bei folgender einfacher Schaltung gesehen, dass statische Hazards entstehen können.



Hier drängt sich aber der Eindruck auf, dass Hazards **vermeidbar** wären.

Wann sind Hazards in kombinatorischen Schaltungen durch geschickteren Entwurf vermeidbar?

Definition

Seien $p, q \in \mathcal{B}^n$. Dann ist

der kleinste Würfel, der p und q enthält.
(smallest cube containing)

$$scc(p, q) := \bigcap_{\substack{a \text{ Produkt} \\ a(p)=1=a(q)}} a$$

Unvermeidbare Hazards

Beispiele: $scc(0000,1011) = \bar{x}_2$

$scc(0101,0111) = \bar{x}_1 x_2 x_4$

Man kann den scc auch für größere Punktmengen oder für Funktionen definieren:

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

Beispiel:

$$scc(x_1 \bar{x}_2 \vee x_1 \bar{x}_3 \vee x_1 \bar{x}_4) = x_1$$

Definition

Seien $p, q \in \mathcal{B}^n$ und sei s ein Signal in einem kombinatorischen Schaltkreis C . Dann ist ein Hazard beim Übergang der Eingänge von p auf q **unvermeidbar**

$$\Leftrightarrow F[s]_{scc(p,q)} \notin \{0,1\}$$

Unvermeidbare Hazards ff

Offensichtlich ist für

$$F[s]_{scc(p,q)} \in \{0,1\}$$

die globale Signalfunktion $F[s]$ des Signals s auf allen Punkten des $scc(p,q)$ konstant (0 oder 1). Im anderen Falle gibt es einen Punkt q' im $scc(p,q)$ mit

$$F[s](q') \neq F[s](p)$$

Wir nehmen nun beim Übergang der Eingänge von p nach q die Schaltzeiten der Eingänge so an, dass an den Eingängen die Folge

$$p = q_1, \dots, q_{i-1}, q_i = q', q_{i+1}, \dots, q_r = q, \text{ mit } dh(q_i, q_{i+1}) = 1$$

entsteht, d.h. für kurze Zeit q' anliegt. Dies entspricht einer Durchquerung des $scc(p,q)$ von der Ecke p zur Ecke q über die Ecke q' .

*Es ist $dh(p,q)$ die **Hammingdistanz** zweier Vektoren p und q definiert als

$$dh(p, q) := \sum_{i=0}^{n-1} p_i \oplus q_i$$

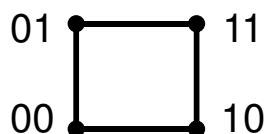
Unvermeidbare Hazards ff

Nun berechnet der Schaltkreis an s als Reaktion darauf

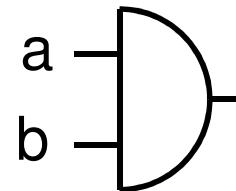
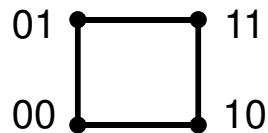
$$F[s](p) = F[s](q_1), \dots, F[s](q_i), \dots, F[s](q_r)$$
$$\parallel$$
$$F[s](q') \neq F[s](p)$$

einen Hazard, unabhängig davon, wie $F[s]$ berechnet wird. Die Funktion $F[s]$ allein bestimmt die Gefahr eines Hazards. Wir nennen solche Hazards daher auch **funktionale Hazards**.

Beispiel: schon ein einfaches AND hat einen unvermeidbaren Hazard beim Übergang von 01 nach 10:

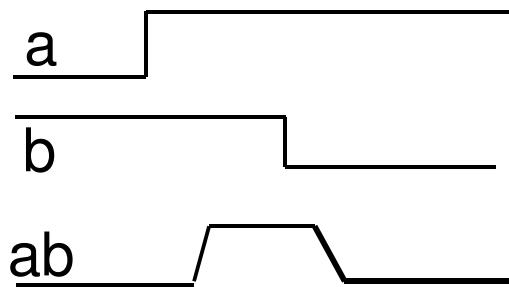
$$scc(01, 10) = 1$$

$$(ab)_{scc(01,10)} = (ab)_1 = ab \notin \{0,1\}$$

Unvermeidbare Hazards ff

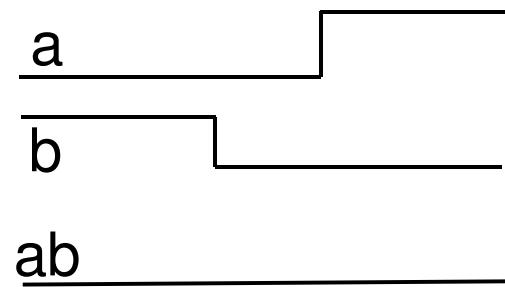


Wird beim Schalten von *01* nach *10* der Weg über *00* genommen, bleibt der Ausgang stabil *0*. Sind dagegen die Zeiten so gestreut, dass der Weg über *11* genommen wird, erhalten wir am Ausgang einen Hazard.

Zeitdiagramme:



über *11* (*a* schneller als *b*)



über *00* (*b* schneller als *a*)

Intervallabschätzung von Laufzeiten

Die bisher gemachten Annahmen waren sehr pessimistisch. Man kann auch leicht Zeitintervalle schätzen, in denen sich Signale ändern können. Mit solchen Informationen kann man Hazards ausschließen, etwa weil man weiß, dass die dazu nötigen Änderungen in disjunkten Zeitintervallen stattfinden:

Annahme: Wir kennen zu jedem Paar (Eingang x , Ausgang y) eines Bausteins eine minimale Reaktionszeit und eine maximale Reaktionszeit von y auf eine Änderung an x :

$$d_{min}(x, y) \text{ sowie } d_{max}(x, y)$$

Wir können nun leicht jedem Signal eines kombinatorischen Schaltkreises ein Zeitintervall zuordnen, in dem es sich überhaupt ändern kann:

Intervallabschätzung ff

Sei C ein kombinatorischer Schaltkreis und sei s ein Signal in C . Dann setzen wir:

$$t_{min}(s) := 0, \text{ falls } s \text{ Primäreing ang}$$

$$t_{min}(s) := \min\{t_{min}(\text{sig}(g.x)) + d_{min}(x, y) \mid g.y = \text{treiber}(s)\}$$

sonst.

Und ebenso:

$$t_{max}(s) := t_{skew}, \text{ falls } s \text{ Primäreing ang}$$

$$t_{max}(s) := \max\{t_{max}(\text{sig}(g.x)) + d_{max}(x, y) \mid g.y = \text{treiber}(s)\}$$

sonst.

Intervallabschätzung ff

Auf Grundlage dieser Definitionen zeigt man leicht durch Induktion nach der Tiefe von s :

Lemma

Ändern sich die Primäreingänge eines kombinatorischen Schaltkreises C zum Zeitpunkt t , dann ist das Signal s höchstens im Zeitintervall

$$[t + t_{min}(s), t + t_{max}(s)]$$

instabil.

3. Aufbau und Struktur eines Prozessors

Zur Vorlesung Rechenanlagen

SS 2019



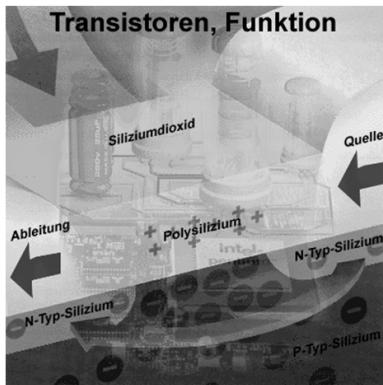
3.1 Maschinen, Instruktionen und Programme

Zur Vorlesung Rechenanlagen

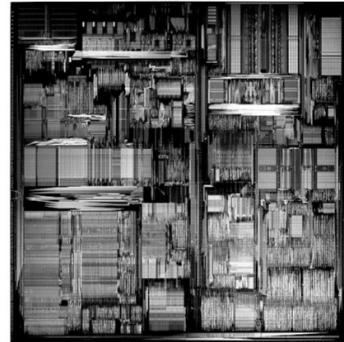
SS 2019



Was ist ein Rechner ?



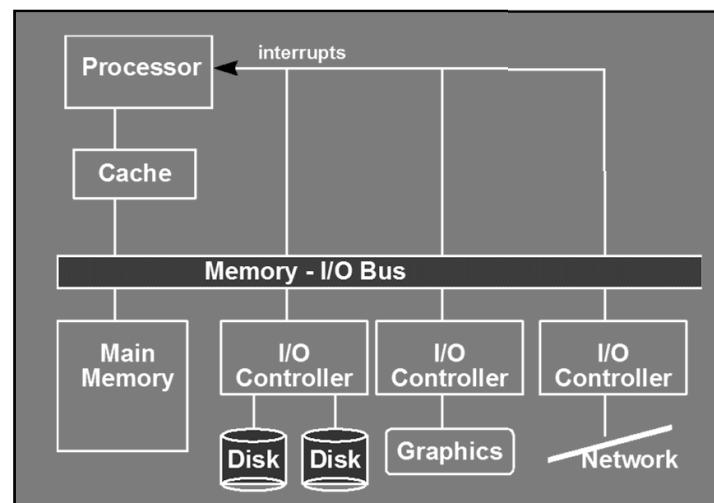
Transistor-Ebene



Layout Ebene

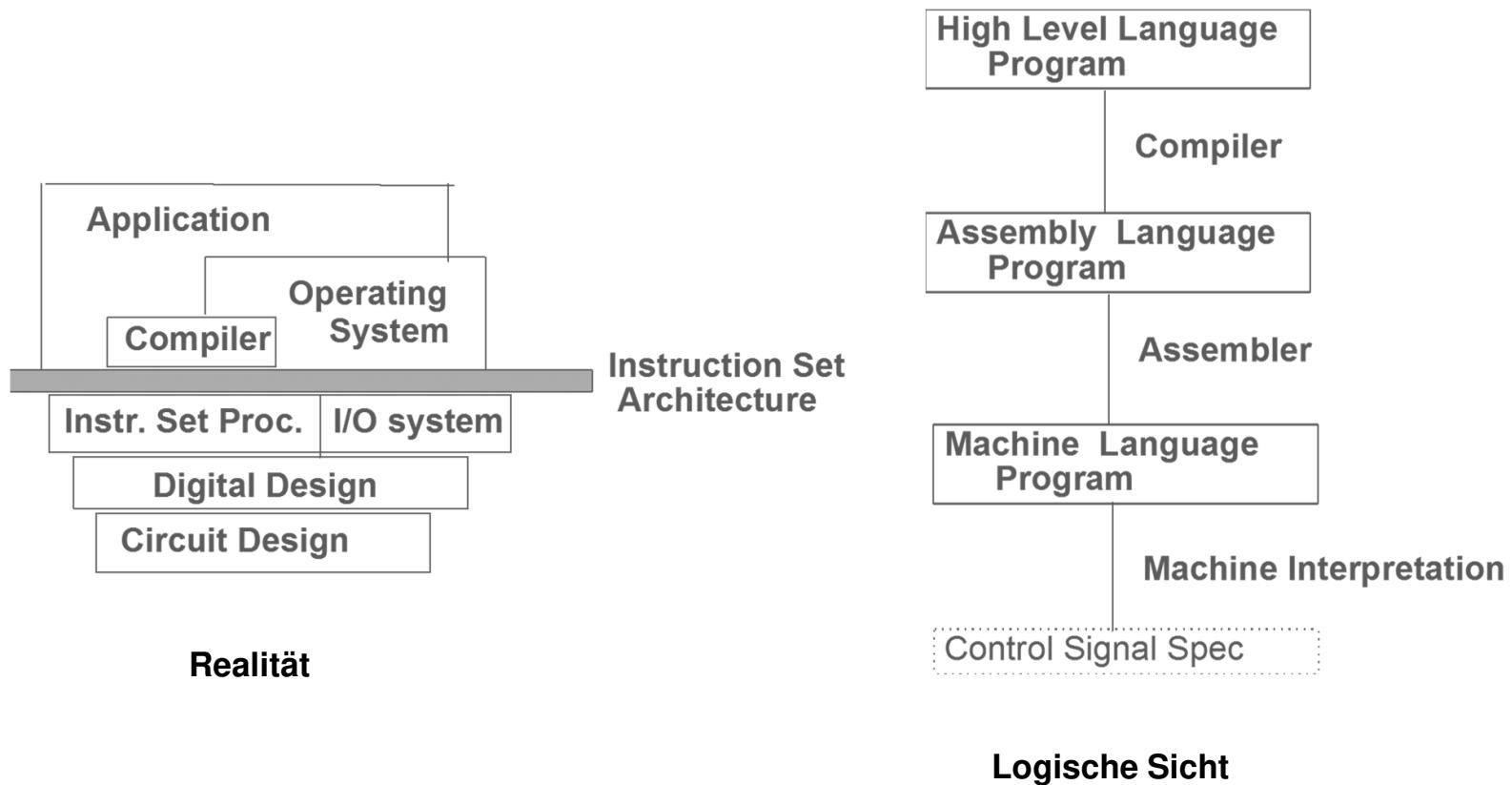


Anwendungsebene

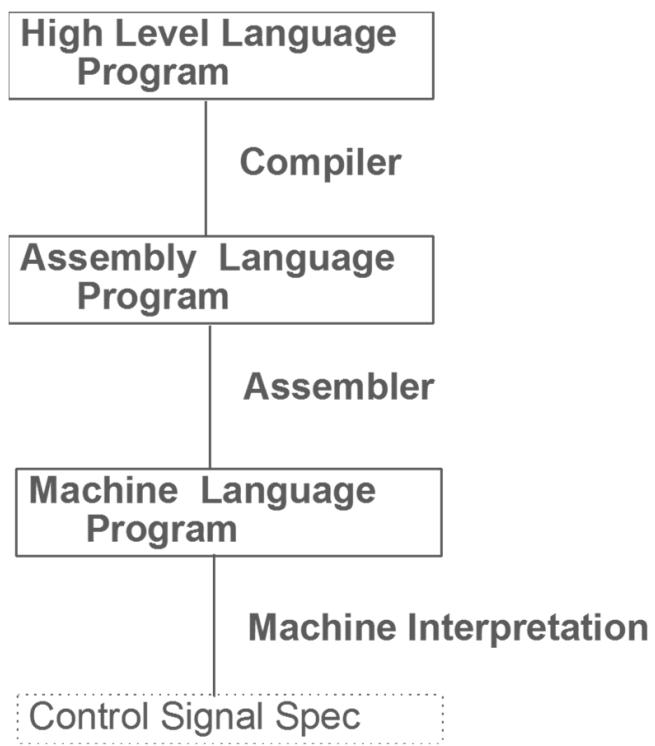


Register-Transfer-Ebene

Rechner = Hierarchie von virtuellen Maschinen



Hierarchie virtueller Rechner: Illustration



temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;

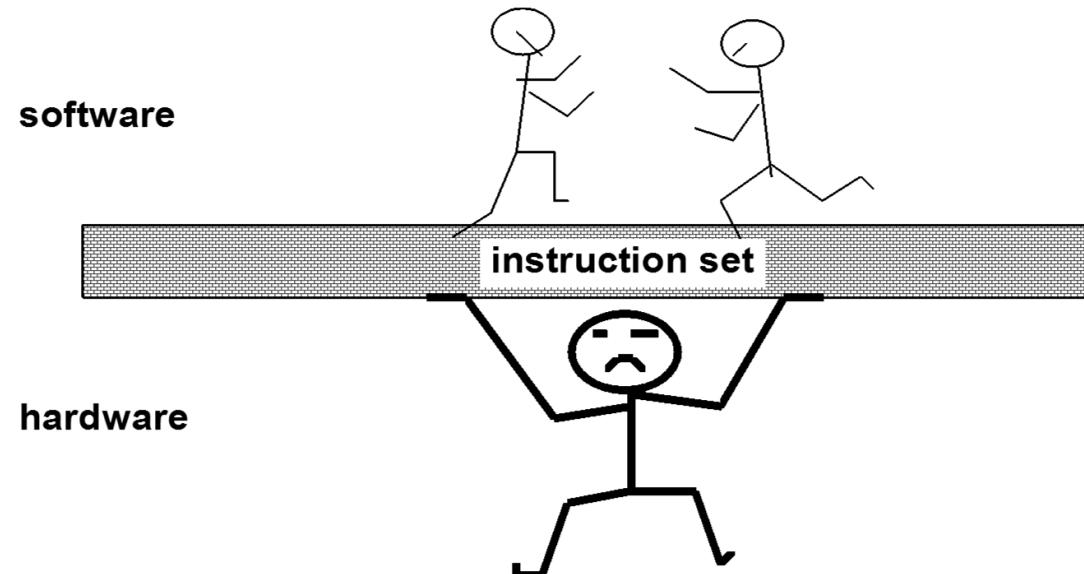
LOAD R5, R2, #0
LOAD R6, R2, #4
STORE R6, R2, #0
STORE R5, R2, #4

1001 0000 0000 0000 0010 0000 1111 0000
1001 0000 0100 0000 0010 0001 0000 0000
0001 0000 0000 0000 0010 0001 0000 0000
0001 0000 0100 0000 0010 0000 1111 0000

... Interpretation auf der Hardware

Im Mittelpunkt steht die Maschinensprache

..., die unterste Ebene, auf der der Benutzer mit dem Rechner reden kann !



Virtueller Rechner

- Jede Ebene (mit Ausnahme der untersten) ist ein "virtueller Rechner" M_i , der durch eine Sprache L_i definiert ist.
- Programme der Sprache L_{i+1} werden
 - in Programme der Sprache $L_j (j \leq i)$ übersetzt oder
 - durch ein Programm der Sprache $L_j (j \leq i)$ interpretiert.
- Die virtuelle Maschine M_i gaukelt dem Benutzer vor, seine in L_i geschriebenen Programme würden direkt auf Hardware ausgeführt.

3.1.1 Instruktionssätze

Der Befehlssatz einer Maschine bildet die eigentliche Schnittstelle zur Programmierung.

Fragen:

- **Welche Instruktionen braucht man unbedingt?**
- **Welche Eigenschaften sind notwendig?**
- **Was ist nur programmiertechnischer Komfort?**

In der Praxis gibt es viele Instruktionssätze, sehr reichhaltige (CISC) und eher magere (RISC). Sie bilden stets einen Kompromiß zwischen Anforderungen von

- höheren Programmiersprachen
- Betriebssystemen
- technischer Realisierbarkeit
- Geschwindigkeit

CISC oder RISC?

Man unterscheidet heute zwischen sogenannten

Complete **I**nstruction **S**et **C**omputer (CISC)

Beispiele: 80x86, VAX, IBM370, 68xxx,...

Sehr großer Befehlssatz, viele Befehle und Adressierungsarten.
Dadurch recht kurze Programme aber sehr unterschiedliche
Bearbeitungszeit pro Befehl und niedriger Durchsatz von Befehlen.

Reduced **I**nstruction **S**et **C**omputer (RISC)

Beispiele: SPARC, HPPA, Alpha, PowerPC,...

Magerer Befehlssatz, nur das Notwendigste. Sehr geringe
Bearbeitungszeit pro Befehl und hoher Durchsatz von Befehlen,
aber längere Programme.

Maschinenbefehle und Maschinensprache

Wir wollen nun versuchen, nach und nach den Befehlssatz *IS* einer einfachen Maschine, der **WüRC** (Würzburger RISC), und damit deren Sprache schrittweise aus Notwendigkeiten heraus zu entwickeln.

Beobachtung: Arithmetik wird stets über 2 Operanden ausgeführt und liefert ein Resultat. Es scheint natürlich, den Befehlen gleichwertig 3 Operanden zu geben, einen für das Resultat und 2 für die Operanden. Für ganze Zahlen wäre dies

| | |
|-----------------------|-----------------------|
| ADD Rd, Rs, Rt | MUL Rd, Rs, Rt |
| SUB Rd, Rs, Rt | DIV Rd, Rs, Rt |

Bemerkung: Wenn man unterschiedliche Zahlen und Zahlenformate hat, braucht man diese Befehle auch für jedes Zahlenformat, etwa

ADDU Rd, Rs, Rt . . . für unsigned numbers oder

ADDD Fd, Fs, Ft . . . für doppelt genaue Gleitkommazahlen

Wir lassen Unsigned- und Gleitkommaoperationen zunächst aussen vor, um den Rechner einfach zu halten!

Definition der Wirkung der Befehle

Die Befehle, die wir bisher angeschaut haben arbeiten alle auf einem sehr kleinen und schnellen Speicher, den **Registern**. Wir werden im Verlauf der Vorlesung noch sehen, welche Vorteile es hat, Rechnern Register zu geben. Wir definieren die Wirkung der Befehle, indem wir uns die Register als ein Array

$\text{REG}[0:\#\text{Register}-1]$ mit dem Bereich $0 \dots \#\text{Register}-1$,

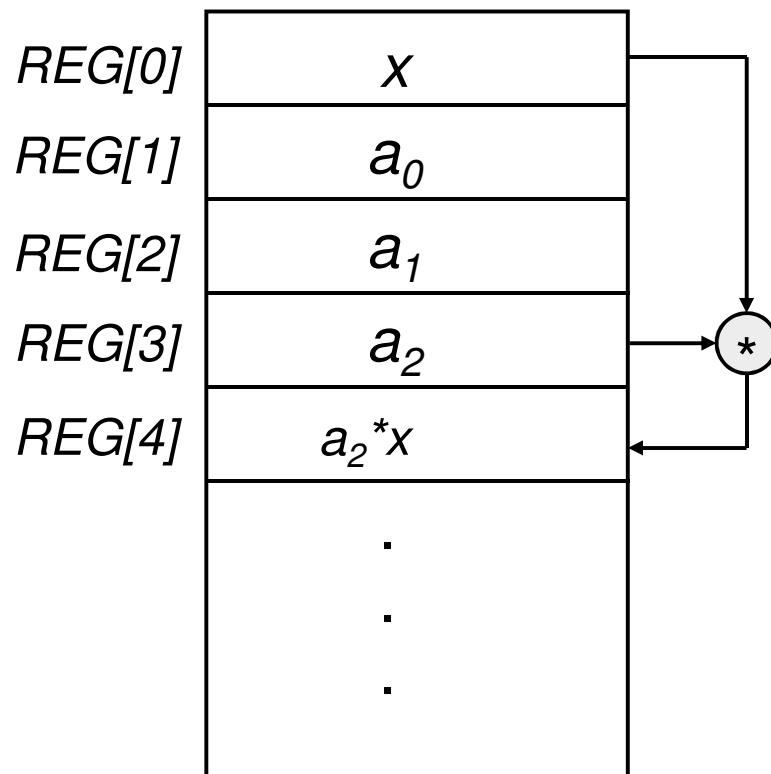
und einen Programmzähler als Variable pc vorstellen, die stets auf die auszuführende Instruktion zeigt.

Ein **Programm** sei stets eine Folge $\text{Prog}[]$ von 0 bis *Programmlänge - 1*

| $\text{Prog}[pc]$ | Wirkung |
|-----------------------|--|
| ADD Rd, Rs, Rt | $\text{REG}[d] = \text{REG}[s] + \text{REG}[t], \quad pc = pc + 1$ |
| MUL Rd, Rs, Rt | $\text{REG}[d] = \text{REG}[s] * \text{REG}[t], \quad pc = pc + 1$ |
| SUB Rd, Rs, Rt | $\text{REG}[d] = \text{REG}[s] - \text{REG}[t], \quad pc = pc + 1$ |
| DIV Rd, Rs, Rt | $\text{REG}[d] = \text{REG}[s] / \text{REG}[t], \quad pc = pc + 1$ |

Beispiel: $a_2x^2 + a_1x + a_0$

Registerfile

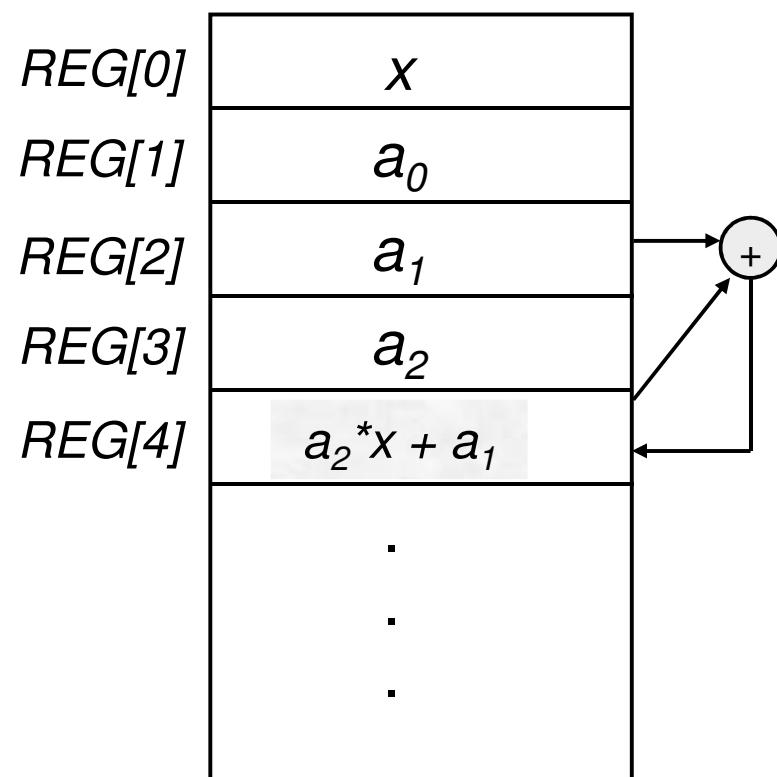


Programm

```
MUL R4, R0, R3;  
ADD R4, R4, R2;  
MUL R4, R4, R0;  
ADD R4, R4, R1;
```

Ablauf für $a_2x^2 + a_1x + a_0$

Registerfile

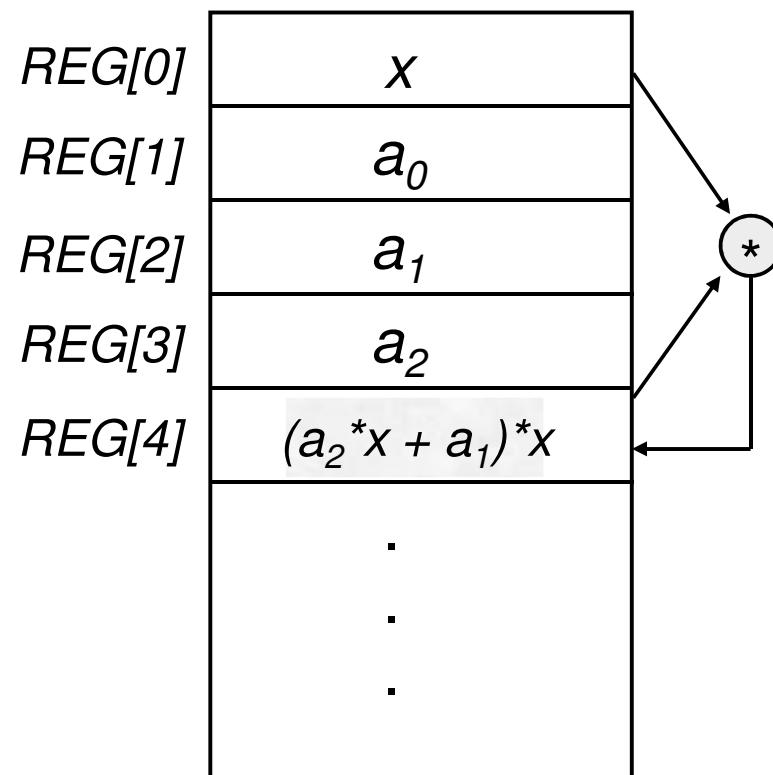


Programm

```
MUL R4, R0, R3;  
ADD R4, R4, R2;  
MUL R4, R4, R0;  
ADD R4, R4, R1;
```

Ablauf für $a_2x^2 + a_1x + a_0$

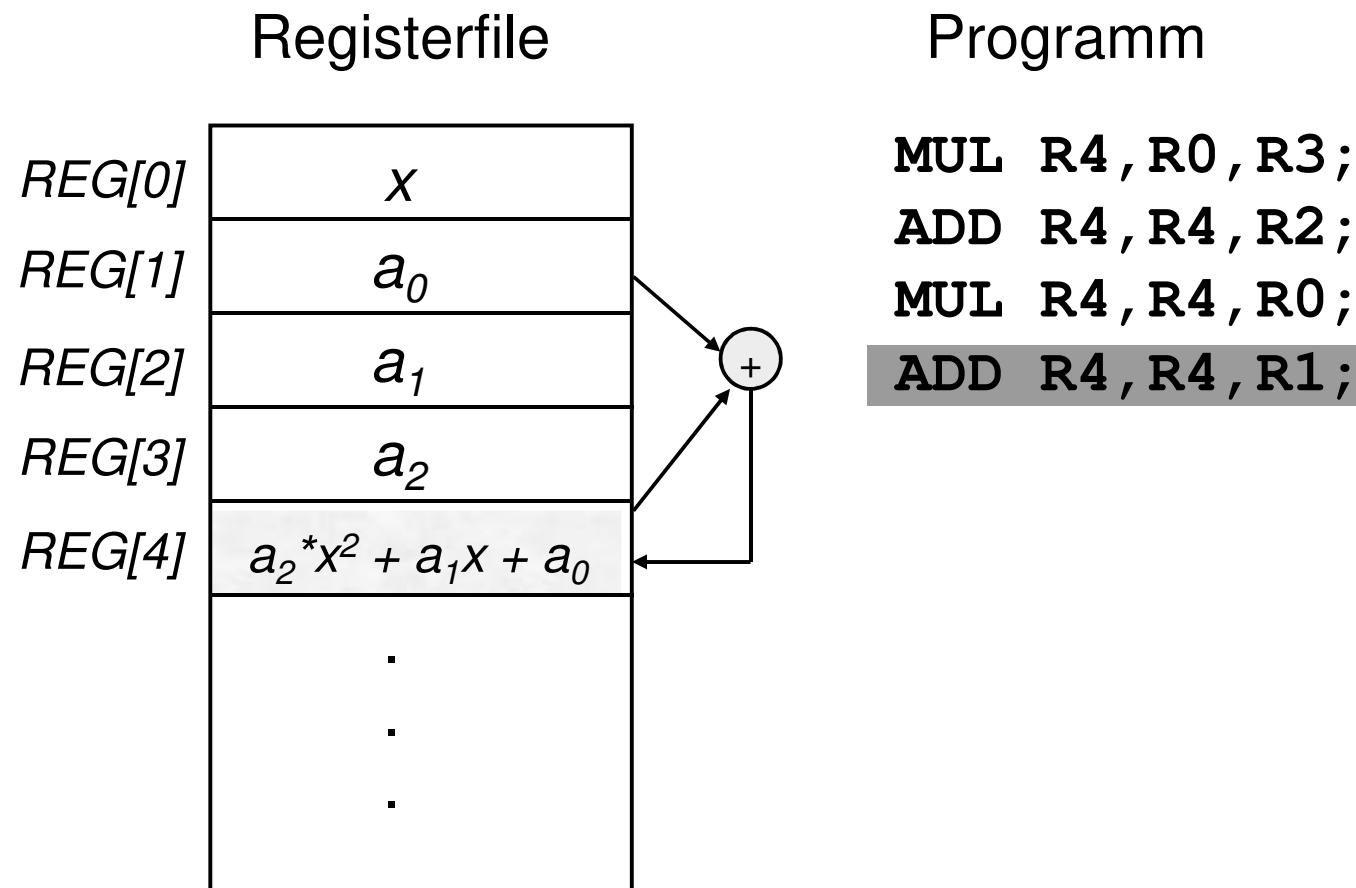
Registerfile



Programm

```
MUL R4, R0, R3;  
ADD R4, R4, R2;  
MUL R4, R4, R0;  
ADD R4, R4, R1;
```

Ablauf für $a_2x^2 + a_1x + a_0$



Verzweigungen und Sprünge

Wir können nun zwar rechnen, das war's aber auch schon!

Aufgabe: Schreibe ein Programm, das in einem beliebigen Zustand der Register $REG[0] = 5$ setzt, wenn $REG[1]$ kleiner ist als $REG[2]$, sonst $REG[0] = -5$.

Dieses Programm existiert nicht, weil wir keine Fallunterscheidung machen können. Befehle werden stets nacheinander abgearbeitet. Auf Benutzung eines Programmzählers hätte man bisher auch verzichten können und Programme von

Prog[0] bis Prog[Programmlänge -1]
einfach der Reihe nach abarbeiten können.

Wie kann man die Reihenfolge der Abarbeitung von Befehlen beeinflussen?

Verzweigungen und Sprünge ff

Wir führen Befehle zur eigentlichen Kontrolle des Programmablaufs ein:

Verzweigungen (**B**ranches) und Sprünge (**J**umps).

| Prog[pc] | Wirkung |
|--------------------|-------------------------------------|
| JMP #t | $pc=pc+t+1$ |
| BEQZ Rs, #t | $pc=pc+1$, falls $REG[s] \neq 0$ |
| | $pc=pc+t+1$, falls $REG[s] == 0$ |
| BNEZ Rs, #t | $pc=pc+1$, falls $REG[s] == 0$ |
| | $pc=pc+t+1$, falls $REG[s] \neq 0$ |

Set Condition Befehle

Man erkennt, dass diese Verzweigungsbefehle nur ausreichen, um $=$ oder \neq zu entscheiden, und danach zu verzweigen. Um auch andere Bedingungen prüfen zu können, erweitern wir die Maschine noch um Vergleichsbefehle **sxx Rd, Rs, Rt** (Set Condition XX):

| Prog[pc] | Wirkung |
|----------------|--|
| SLT Rd, Rs, Rt | $REG[d] = (REG[s] < REG[t]), \quad pc=pc+1$ |
| SLE Rd, Rs, Rt | $REG[d] = (REG[s] \leq REG[t]), \quad pc=pc+1$ |
| SEQ Rd, Rs, Rt | $REG[d] = (REG[s] == REG[t]), \quad pc=pc+1$ |
| SNE Rd, Rs, Rt | $REG[d] = (REG[s] \neq REG[t]), \quad pc=pc+1$ |
| SGE Rd, Rs, Rt | $REG[d] = (REG[s] \geq REG[t]), \quad pc=pc+1$ |
| SGT Rd, Rs, Rt | $REG[d] = (REG[s] > REG[t]), \quad pc=pc+1$ |

Load/Store Befehle

Bisher können wir mit den Befehlen nur Register ansprechen, aber keine Plätze im Hauptspeicher. Dies ist bei RISC Prozessoren in der Tat auch so. Sie sehen für die Kommunikation mit dem Speicher Transportbefehle (LOAD , STORE) vor:

Wir fassen dazu den Hauptspeicher als Array $\text{MEM}[0:\#\text{Plätze}-1]$ auf.

| Prog[pc] | Wirkung |
|-------------------------|---|
| LOAD Rd, Rs, #k | $\text{REG}[d] = \text{MEM}[k + \text{REG}[s]]$, $\text{pc} = \text{pc} + 1$ |
| STORE Rs, Rd, #k | $\text{MEM}[k + \text{REG}[d]] = \text{REG}[s]$, $\text{pc} = \text{pc} + 1$ |

Man erkennt bei diesen Befehlen noch eine weitere Besonderheit: die Adresse des Quell-(Load) bzw. Ziel-(Store) Operanden wird **indirekt**, durch den Inhalt eines Registers plus ein **Displacement** angegeben. Man spricht hier auch von **indirekter Adressierung** im Gegensatz zur bisher benutzten **direkten Adressierung**. CISC Maschinen sehen indirekte Adressierung bei jedem Operanden vor, bei dem es Sinn macht (siehe Ausblick).

Adressierungsarten

Neben der indirekten Adressierung kann es auch Sinn machen, unmittelbar im Befehl den Wert des Operanden anzugeben. Dies findet man bei RISC Maschinen meist für den zweiten Quelloperanden. Bei CISC Maschinen für jeden Operanden bei dem es Sinn macht:

| Prog[pc] | Wirkung |
|------------------------|--|
| ADDI Rd, Rs, #k | $\text{REG}[\text{d}] = \text{REG}[\text{s}] + \text{k}$, $\text{pc} = \text{pc} + 1$ |
| SUBI Rd, Rs, #k | $\text{REG}[\text{d}] = \text{REG}[\text{s}] - \text{k}$, $\text{pc} = \text{pc} + 1$ |

Man spricht hier auch von **unmittelbarer Adressierung** oder **immediate Adressierung**. Wir erlauben diese auch bei SXX Befehlen.

Adressierungsarten -- ff

Man könnte auch versuchen, die Speicherzellen bei LOAD und STORE Befehlen nur direkt anzusprechen, etwa wie folgt (absolute Adressierung):

| Prog[pc] | Wirkung |
|---------------------|-----------------------------------|
| LOAD Rd, #k | REG [d] =MEM [k], pc=pc+1 |
| STORE Rs, #k | MEM [k] = REG [s], pc=pc+1 |

Dies hätte folgende entscheidende Nachteile:

- ⌚ Man könnte mit einem Programm nur die Speicherzellen ansprechen, die explizit drin stehen. Damit könnte man keine Aufgaben lösen, die ein Berechnen von Adressen erfordern, wie zum Beispiel das Verschieben eines Speicherbereiches vorgegebener Länge.
- ⌚ Man müsste die Adressen immer explizit im Befehlscode ablegen, was sehr unausgewogene Befehlsformate zur Folge hätte.

Nullregister

Man kann nun einige Befehle und Mechanismen leicht auf bisher eingeführte Befehle zurückführen, wenn man ein Register als Nullkonstante benutzt. Wir vereinbaren ab sofort:

- Das Register REG[0] hat stets den Inhalt 0.
- Schreibzugriffe auf REG[0] sind wirkungslos.

Mit dieser Modifikation kann man einige nützliche “Befehle” als Aliase auf vorhandene Befehle zurückführen.

Beispiele:

Lade eine Konstante

LDI Rd, #k alias **ADDI Rd, R0, #k**

Transport eines Registers

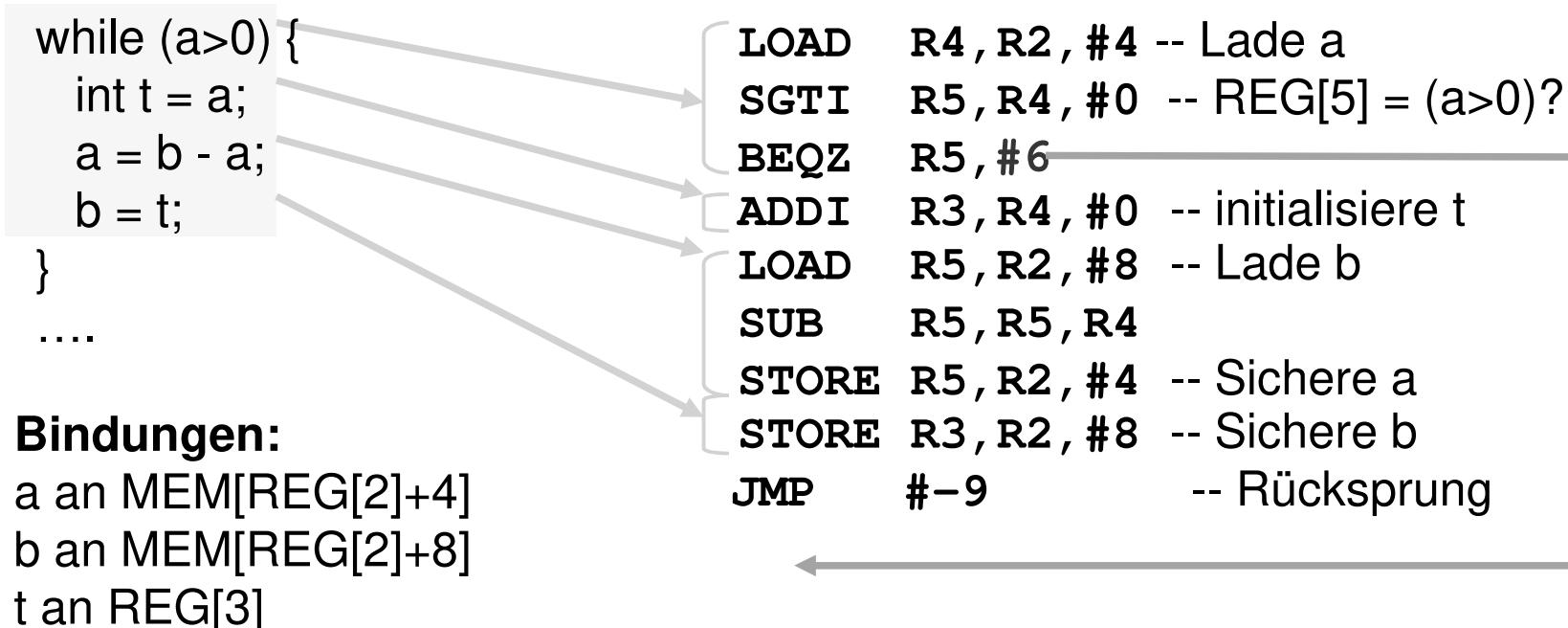
MOV Rs, Rd alias **ADD Rd, Rs, R0**

Zugriff auf absolute Speicheradresse (“Absolute Adressierung”)

LOAD Rd, #source alias **LOAD Rd, R0, #source**

Hochsprache nach Maschine

Wir illustrieren, wie man ein Programmstück in Maschinensprache umsetzen könnte:



Beispiel:

Wir wollen unsere Definitionen nun an einem einfachen Programm erproben:

Aufgabe: Schreibe ein Programm, das zu zwei positiven Zahlen ihren größten, gemeinsamen Teiler ausgibt.

Wir benutzen dazu den **Euklidschen Algorithmus**, der auf folgender Beobachtung beruht:

Für $x \leq y$ gilt:

$$ggT(x, y) = \begin{cases} y & \text{falls } x = 0 \\ ggT(y \bmod x, x) & \text{falls } x > 0 \end{cases}$$

Da das Minimum der Operanden stets abnimmt, wenn man (x, y) durch $(y \bmod x, x)$ ersetzt, terminiert das Verfahren.

Das ggT Programm der WüRC

Wir entwickeln nun ein WüRC Programm, das bei der Registerbelegung REG[2] = x und REG[3] = y den ggT in REG[3] zurückliefert:

| | | |
|------|----------------------------|----------------------------------|
| init | 0 : SLE R4, R2, R3 | |
| | 1 : BNEZ R4, #1oop | // für x>y muß man noch tauschen |
| | 2 : MOV R2, R4 | // Tausch von x,y |
| | 3 : MOV R3, R2 | |
| | 4 : MOV R4, R3 | // Tausch beendet |
| loop | 1oop: MOV R2, R5 | // rette x in R5 |
| | 6 : DIV R4, R3, R2 | // y mod x = y - (y/x)*x |
| | 7 : MUL R4, R4, R2 | |
| | 8 : SUB R2, R3, R4 | // R2 = y mod x |
| | 9 : MOV R5, R3 | // also muß nun x nach R3 |
| | 10 : SGT R4, R2, R0 | |
| | 11 : BNEZ R4, #1oop | // für x>0 das ganze nochmal! |

Anmerkungen zum ggT-Programm

Wir haben zur besseren Lesbarkeit die Instruktion 6 im Programm mit loop markiert und die Sprungziele zu dieser Anweisung ebenfalls loop genannt. Solche Markierungen nennt man **Marken** (labels). Zu ihrer Verwendung gelten folgenden Regeln:

- **Jede Marke hat im Programm genau ein definierendes Auftreten. Sie erhält den Wert i falls sie vor der Anweisung $\text{Prog}[i]$ auftritt.**
- **Alle anderen Auftreten werden durch den Wert <Markenwert - Platz des Auftretens - 1> ersetzt.**

3.1.2 Unterprogramme

Zur Lösung großer Programmieraufgaben fehlt uns immer noch Komfort. Eine weitere Programmieraufgabe soll dies demonstrieren:

Aufgabe:

Schreibe ein Programmstück, das das Skalarprodukt zweier n -stelliger Vektoren x, y unter folgender Speicherbelegung berechnet:

- $MEM[k]$ der Wert von n
- $MEM[MEM[k+1]], \dots, MEM[MEM[k+1]+n-1]$ der Vektor x
- $MEM[MEM[k+2]], \dots, MEM[MEM[k+2]+n-1]$ der Vektor y
- $MEM[k+3]$ Adresse des Resultats

Folgende Sequenz löst die Aufgabe:

```

LOAD R2,R0,#k          // R2 halte n
STORE R0,R0,#k+4       // Resultat temporär in MEM[k+4]
LDI   R3,#1            // R3 dient als Zähler i
LOAD R4,R0,#k+1        // R4 zeigt auf xi
LOAD R5,R0,#k+2        // R5 zeigt auf yi
loop:SGT  R6,R3,R2      // Zähler > n?
    BNEZ R6,#weiter     // fertig, falls ja!
    LOAD R6,R4,#0         // hole xi
    LOAD R7,R5,#0         // hole yi
    MUL   R6,R6,R7
    LOAD R7,R0,#k+4       // hole Resultat
    ADD   R7,R7,R6         // aufaddieren auf Resultat
    STORE R7,R0,#k+4      // sichern des Resultats
    INC   R3              // Zähler und Adressen
    INC   R4              // inkrementieren
    INC   R5
    JMP   loop
weiter:LOAD R3,R0,#k+3  // Holen der Resultatsadresse
    STORE R7,R3,#0         // und Rückgabe

```

Solche Aufgaben kommen in einem größeren Programm möglicherweise sehr oft vor. Man möchte aber nicht immer die Sequenz hinschreiben müssen um sie wiederzuverwenden.

Probleme

- Wie kann man die Sequenz so darstellen, dass sie für jedes k benutzt werden kann, ohne explizit dafür neu hingeschrieben zu werden? k ist offenbar die Adresse relativ zu der
 - die Argumente abzuholen, und
 - das Resultat abzulegen ist
- Wie findet man weiter an der Stelle „weiter“? Wenn wir die Sequenz abgearbeitet haben, müssen wir an die Stelle des Programmes gehen, an der die Sequenz aufgerufen wurde. Dies ist i.A. bei jedem Aufruf eine andere Stelle.

Bemerkung: Vergleiche mit Prozeduren und Funktionen in höheren Programmiersprachen!

Beobachtungen:

- ***k* kommt bei fast allen Adressargumenten vor.** Hier könnten sich die Displacements, die wir bei Transporten eingeführt haben, auszahlen, d.h. man könnte als Adresse $REG[j]+i$ nehmen, wobei in $REG[j]$ der Wert k steht.
- **Wir haben einfach die Register $R2, R3, \dots$ als Hilfszellen benutzt.** Es scheint nützlich zu sein, wenn Unterprogramme stets auf die Register als Hilfszellen zugreifen könnten, sie quasi als „Freiwild“ benutzen können.
- **Will man das Ziel bei „weiter“ bestimmen,** kommt man mit relativen Sprüngen nicht aus. Wir brauchen dazu Sprünge, die indirekt über Speicherinhalte laufen.

Indirekter Sprung

Wir erweitern die Maschine noch um die Befehle

JREG Rs // Jump Register

mit der Wirkung

$pc = REG[s];$

sowie

JAL Rd, #f // Jump and link

mit der Wirkung

$pc = pc + f + 1, REG[d] = pc + 1;$

Eine Aufrufsequenz hätte dann etwa folgendes Aussehen:

**JAL R2, #Skalarprodukt
weiter: . . .**

Und die anschliessende Rückkehr einfach

JREG R2

Programmierstil

Mit diesen neuen Befehlen kann man nun die Sequenz durch **JAL Ri, #skalarprodukt** aufrufen, und durch **JREG Ri** korrekt beenden. Das Register *i* muss sowohl dem aufrufenden als auch dem aufgerufenen Programm bekannt sein. Es sichert den Rücksprung.

Wir müssen darüberhinaus auch dafür sorgen, dass

- der Speicher korrekt benutzt wird,
- die Argumente und Resultate korrekt ausgetauscht werden,
- die Register korrekt behandelt werden.

Dazu braucht man keine neuen Befehle mehr, sondern lediglich eine Vereinbarung, wie Unterprogramme zu schreiben sind und wie sie auf den Speicher wirken dürfen. Eine solche programmiertechnische Vereinbarung nennt man auch einfach einen **Programmierstil**.

Unterprogrammregeln

- (U1) Das Register SP trägt stets die Adresse einer Speicherzelle k , so dass in $MEM[k+1], \dots, MEM[k+l]$ die Argumente stehen. Wir wählen z.B. $\mathbf{SP = R1}$.
- (U2) Keine Speicherzelle $j \leq k$ wird vom aufrufenden Programm schon benutzt.
- (U3) (a) Alle Register (außer SP) dürfen vom Unterprogramm verändert werden (das aufrufende Programm trägt Vorsorge). Oder
(b) Alle Register müssen unverändert zurückgeliefert werden (das Unterprogramm trägt Vorsorge).
- (U4) SP enthält nach Rückkehr aus dem Unterprogramm den gleichen Wert wie beim Aufruf.
- (U5) $R2$ hält beim Aufruf stets die Rücksprungadresse.

Unterprogrammregeln ff

Man nennt U3 (a) die **caller saves** Strategie,
U3 (b) die **callee saves** Strategie.

Das Register *SP* spielt nun auch eine Sonderrolle:
Es zeigt stets den Bereich des Speichers an, ab dem nur noch freie Plätze kommen. Man kann dies auch als oberes Ende eines Stapels betrachten. Wir nennen daher *SP* auch **Stackpointer**, weil es auf das obere Ende eines Stacks zeigt.

Die Rücksprungadresse könnte man auch an einem anderen Platz aufbewahren. Allerdings muss ein Unterprogramm wissen, wo sie gesichert ist, unabhängig davon, von welchem Programm es aufgerufen wurde.
Deswegen muss man sich auch auf ein Rücksprungregister einigen (U5).

Skalarprodukt Unterprogramm

Wir sind nun in der Lage, das Unterprogramm zu unserem Beispiel, "Skalarprodukt", nach (U3)(b) (callee saves) anzugeben:

Vereinbarungen:

in $MEM[SP+1]$ steht n

in $MEM[SP+2]$ steht die Adresse des Anfangs von x

in $MEM[SP+3]$ steht die Adresse des Anfangs von y

in $MEM[SP+4]$ steht das Resultat

Das Unterprogramm für 8 Register lautet nun wie folgt:

Eintritt ins Unterprogramm

```
Skalarprodukt: SUBI R1,R1,#7 //Platz zum Arbeiten schaffen
STORE R2,R1,#2 // Sichere alle Register
STORE R3,R1,#3 // die benutzt werden
STORE R4,R1,#4
STORE R5,R1,#5
STORE R6,R1,#6
STORE R7,R1,#7
LOAD R2,R1,#8 // R2 halte nun n
STORE R0,R1,#1 // Resultat temporär in MEM[SP+1]
LDI R3,#1 // R3 dient als Zähler i
LOAD R4,R1,#9 // R4 zeigt auf xi
LOAD R5,R1,#10 // R5 zeigt auf yi
```

Eigentliche Berechnung

```
loop:SGT  R6,R3,R2
        BNEZ R6,#return // noch zu tun?
        LOAD R6,R4,#0    // hole xi
        LOAD R7,R5,#0    // hole yi
        MUL  R6,R6,R7
        LOAD R7,R1,#1    // hole Resultat
        ADD   R7,R6,R7   // aufaddieren auf Resultat
        STORE R7,R1,#1   // sichern des Resultats
        INC   R3          // Zähler und Adressen
        INC   R4          // inkrementieren
        INC   R5
        JMP   loop
```

Rückkehr ins Hauptprogramm

```
return: LOAD R3,R1,#11 // Holen des Resultatsadresse
       STORE R7,R3,#0      // und Wegschreiben des Resultats
       LOAD R2,R1,#2      // Wiederherstellung der Register
       LOAD R3,R1,#3      // die benutzt wurden
       LOAD R4,R1,#4
       LOAD R5,R1,#5
       LOAD R6,R1,#6
       LOAD R7,R1,#7
       ADDI R1,R1,#7      //Stapel auf Aufrufniveau
       JREG R2            // und Tschüss
```

Aufruf des Unterprogramms

Annahmen:

Skalarprodukt wird selbst von einem Unterprogramm aufgerufen, das den Speicher von $MEM[REG[1]+1]$ bis $MEM[REG[1]+m]$ benutzt, d.h. zum Aufrufzeitpunkt sei $MEM[REG[1]]$ und alle kleineren Adressen frei.

Der Wert von n stehe in $MEM[n_adr]$, die Anfangsadressen von x und y in $MEM[x_anf]$ sowie $MEM[y_anf]$ und die Adresse des Ergebnisses in $MEM[e_ad]$.

```
// Beginn der Aufrufsequenz
SUBI R1,R1,#4      // Anpassen von SP um Zahl der Args.
LOAD R7,R0,#n_adr // Hole n
STORE R7,R1,#1     // Übergabe von n in MEM[SP+m+1]
LDI   R7,#x_anf // Hole Anfangsadresse von x
STORE R7,R1,#2     // Übergabe Anfangsadresse von x
LDI   R7,#y_anf // Hole Anfangsadresse von y
STORE R7,R1,#3     // Übergabe Anfangsadresse von y
LDI   R7,#e_ad // Hole Ergebnisadresse
STORE R7,R1,#4     // Übergabe Ergebnisadresse
JAL   R2,#skalarprodukt // Aufruf
ADDI R1,R1,#4      // Anpassen von SP auf altes Niveau
```

Kritik

Wir sehen, dass *R7* eigentlich nicht gerettet werden muss, weil es zur Parameterübergabe verwendet wird. Da wir keine Befehle haben, die nur auf dem Hauptspeicher operieren, brauchen wir stets mindestens ein Register zur Parameterübergabe.

Es liegt also auf der Hand, einige Register stets zur Parameterübergabe zu benutzen, z.B. eins für den Rückgabewert und noch ein paar weitere für die Parameter.

Dies ist dann aber fest im Programmierstil zu vereinbaren!

3.1.3 Ausblick:

Wir wollen es zunächst bei unserer ad hoc entwickelten Maschine belassen. Viele Befehle hätte man auch anders strukturieren und definieren können. Manche Dinge finden erst eine Erklärung, wenn wir wissen, welche technischen Möglichkeiten wir zum Bau einer Maschine haben. Wir schliessen daher den Abschnitt mit ein paar Anmerkungen ab:

Register:

Wir haben Register als ein programmiertechnisches Hilfsmittel im Zusammenhang mit

- Spezialaufgaben ($pc, REG[0], REG[1], REG[2]$)
 → **special purpose**
- allgemeine Aufgaben
 → **general purpose**

eingeführt. Für allgemeine Aufgaben deshalb, weil fast alle Befehle nur direkt ihre Operanden adressieren können, und wir dafür einen kleinen, schnellen Speicher vorhalten wollten.

Autoinkrement, Autodekrement:

Häufig werden Datenstrukturen mit aufsteigender Adresse durchmustert (z.B. Vektoren). Dazu benötigten wir im Skalarproduktprogramm die Sequenz:

LOAD R6, R4, #0 . . . INC R4

Dies könnte man auch in einen Befehl packen, indem man den Operanden des Ladebefehls automatisch inkrementiert. Man spricht dann von **Autoinkrement**. Entsprechendes gilt fürs Dekrementieren. Folgende Befehle (nicht im WüRC Befehlssatz vorhanden!) könnten dies leisten:

| Prog[pc] | Wirkung |
|--------------------------|--|
| LOAD Rd, Rs+, #1 | $\text{REG}[d] = \text{MEM}[1 + \text{REG}[s]]$, $\text{pc} = \text{pc} + 1$ $\text{REG}[s] = \text{REG}[s] + 1$ |
| STORE Rs, Rd+, #1 | $\text{MEM}[1 + \text{REG}[d]] = \text{REG}[s]$, $\text{pc} = \text{pc} + 1$ $\text{REG}[d] = \text{REG}[d] + 1$ |

Indirekte Operanden:

Arithmetikbefehle können nur direkt adressieren.
Deswegen haben wir bei Unterprogrammen auch Register
eingeführt. Häufig ist dann die Sequenz:

```
LOAD Ri,R1,#offset ...    ADD Rr,Rs,Ri
```

Dies könnte man auch in einen Befehl packen, indem man
einen oder gar alle Operanden des Befehls indirekt
angeben kann. Es gibt Maschinen (CISC), bei denen jeder
Operand in jeder dafür vernünftigen Adressierungsart
(unmittelbar, direkt, indirekt, indirekt mit Offset...)
angegeben werden kann.

Orthogonaler Befehlssatz (vgl. VAX)

Dies kann zu einem wahren Wildwuchs von Befehlen führen, so dass eine andere Notation für die Maschinensprache notwendig würde.

Man könnte dann Befehle folgender Art notieren:

ADD Ri, @Rj, @Rk+

Addiere mit Ziel direkt adressiert, Quelle 1 indirekt und Quelle 2 indirekt mit Autoinkrement.

d.h. $REG[i] = MEM[REG[j]] + MEM[REG[k]]$,
 $REG[k] = REG[k]+1$, $PC = PC + 1$

oder

ADD @Ri+, @Rj-, #k

Addiere mit Ziel indirekt mit Autoinkrement, Quelle 1 indirekt mit Autodekrement und Quelle 2 unmittelbar.

d.h. $MEM[REG[i]] = MEM[REG[j]] + k$,
 $REG[i] = REG[i]+1$, $REG[j] = REG[j]-1$, $PC = PC + 1$

Primitive Instruktionssätze

In manchen Lehrbüchern wird aus didaktischen Gründen häufig eine extrem vereinfachte Maschine eingeführt, bei der sämtliche Berechnungen und Transporte über ein ausgezeichnetes Register *acc*, dem **Akkumulator** laufen:

Die Akkumulatormaschine

Arithmetikbefehle haben oft nur einen (Einadresscode) oder zwei (Zweiadresscode) weitere Operanden ausser dem Akkumulator. Folgende Tabelle gibt einen Einblick über solche (unrealistischen) Befehlssätze.

Akkumulatormaschinen

| Prog[pc] | Wirkung |
|------------------------------|--|
| ADD Rs analog SUB, MUL | $acc = acc + REG[s]$, $pc=pc+1$ |
| BEQZ #k | $pc=pc+k$, falls $acc = 0$ $pc=pc+1$, sonst |
| LOAD Rs, #1 STORE Rd, #1 | $acc = MEM[1+REG[s]]$, $pc=pc+1$ $MEM[1+REG[d]] = acc$, $pc=pc+1$ |

3.2 Die Objekte der Maschine

Zur Vorlesung Rechenanlagen

SS 2019



3.2.1 Wortbreite einer Maschine

Wir wissen, dass wir alles auf Bitstrings zurückführen müssen, daher müssen die Objekte durch solche kodiert werden.

Es wäre ungeschickt, Speicherplätze mit verschieden langen Bitstrings vorzusehen, da man dann in einen Speicherplatz nur Objekte einer speziellen Sorte unterbringen könnte.

Objekte und Operationen nur auf der Basis einzelner Bits bereitzustellen und alles andere darauf sequentiell durch Befehlsfolgen zu bearbeiten wäre aber auch unsinnig.

Die **Wortbreite** n einer Maschine legt die maximale Länge von Objekten als Bitstrings fest, die in einem Register oder Speicherplatz gehalten werden können.

Maschinenworte

Sei n die Wortbreite einer Maschine. Dann nennen wir einen Bitstring der Länge n

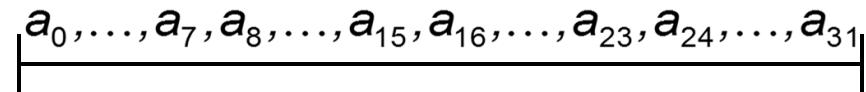
$$a = a_0, \dots, a_{n-1}$$

ein Maschinenwort. Eine heute übliche Breite ist $n = 64$ oder $n = 32$.

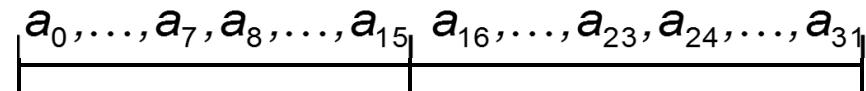
Meist unterteilt man Maschinenworte noch weiter nach 2er Potenzen in Halbworte, Viertelworte, ...

Beispiel: 32 Bit Wortbreite

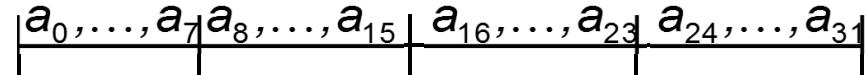
long: 32 - Bit Wort



short: 16 - Bit Wort



byte: 8 - Bit Wort



Maschinenworte ff

Vorsicht: Man spricht oft von 32-, 64-Bit Maschinen. Dies hat nicht notwendig etwas mit unserer Definition der Wortbreite zu tun. Man meint damit oft die maximale Länge, die Adressen von Speicherplätzen als Bitstrings haben können.

Die Objekte unserer Maschine definieren wir nun, indem wir Maschinenwörter oder Teilwörter davon eine geeignete Interpretation verleihen:

3.2.2 Ganze Zahlen

Die wohl wichtigsten Objekte in Rechnern sind Zahlen. Die einfachste Art und Weise, Bitstrings als Zahlen aufzufassen, ist ihre Interpretation als nichtnegative ganze Zahl:

Definition

Wir nennen $u_n : \mathbf{B}^n \mapsto \mathbf{N}_0$

mit $u_n(a_0, \dots, a_{n-1}) := 2^{n-1} \sum_{i=0}^{n-1} a_i 2^{-i}$

Darstellung als vorzeichenlose (unsigned) ganze Zahl.

Das Bit a_0 hat das höchste Gewicht 2^{n-1} . Wir nennen es auch das **signifikanteste Bit**.

Wenn klar ist, dass n die Wortbreite ist, schreiben wir auch einfach u statt u_n .

Unsigned Numbers

Der Zahlenbereich einer vorzeichenlosen ganzen Zahl ist

$$u_n(\mathbf{B}^n) = [0 : 2^n - 1]$$

Größere Zahlen kann man nicht direkt darstellen. Will man mit größeren Zahlen arbeiten, dann muss dies explizit durch Programmierung realisiert werden.

Anmerkungen:

Rechnerarithmetiker benutzen oft auch eine gespiegelte Notation der Bitstrings um Zahlen zu definieren. In diesem Fall entspricht das Gewicht eines Bits i direkt 2^i .

Den Zahlenbereich erhält man durch

$$u_n(0, \dots, 0) = 0, \text{ sowie } u_n(1, \dots, 1) = 2^{n-1} \sum_{i=0}^{n-1} 1 \cdot 2^{-i} = 2^{n-1} (2 - 2^{-(n-1)}) = 2^n - 1$$

Signed Numbers

Definition

Wir nennen $bv_n : \mathbf{B}^n \mapsto \mathbf{Z}$

mit $bv_n(a_0, \dots, a_{n-1}) := (-1)^{a_0} \cdot 2^{n-1} \cdot \sum_{i=1}^{n-1} a_i 2^{-i}$

Darstellung durch Betrag und Vorzeichen.

Der Zahlenbereich bei dieser Darstellung ist

$$bv_n(\mathbf{B}^n) = [bv_n(1\dots1) : bv_n(01\dots1)]$$

$$(-1)^1 \cdot 2^{n-1} \cdot \sum_{i=1}^{n-1} 1 \cdot 2^{-i}$$

$$= -2^{n-1} \cdot (1 - 2^{-(n-1)})$$

$$= -(2^{n-1} - 1)$$

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

$$= 2^{n-1} \cdot (1 - 2^{-(n-1)})$$

$$= 2^{n-1} - 1$$

Signed Numbers ff

Definition

Wir nennen $b_{2,n} : \mathbf{B}^n \mapsto \mathbf{Z}$

mit $b_{2,n}(a_0, \dots, a_{n-1}) := -2^{n-1}a_0 + 2^{n-1} \cdot \sum_{i=1}^{n-1} a_i 2^{-i}$

Darstellung im **Zweierkomplement**.

Der Zahlenbereich bei dieser Darstellung ist

$$b_{2,n}(\mathbf{B}^n) = [b_{2,n}(10\cdots0) : b_{2,n}(01\cdots1)]$$

$$-2^{n-1} \cdot 1 + 2^{n-1} \cdot \sum_{i=1}^{n-1} 0 \cdot 2^{-i}$$

$$= \boxed{-2^{n-1}}$$

$$-2^{n-1} \cdot 0 + 2^{n-1} \cdot \sum_{i=1}^{n-1} 1 \cdot 2^{-i}$$

$$= 2^{n-1}(1 - 2^{-(n-1)}) = \boxed{2^{n-1} - 1}$$

Signed Numbers ff

Definition

Wir nennen $b_{1,n} : \mathbf{B}^n \mapsto \mathbf{Z}$

mit $b_{1,n}(a_0, \dots, a_{n-1}) := -(2^{n-1} - 1)a_0 + 2^{n-1} \cdot \sum_{i=1}^{n-1} a_i 2^{-i}$

Darstellung im **Einerkomplement**.

Der Zahlenbereich bei dieser Darstellung ist

$$b_{1,n}(\mathbf{B}^n) = [b_{1,n}(10\cdots0) : b_{1,n}(01\cdots1)]$$

$$\begin{aligned} & \text{---} \\ & \left. \begin{aligned} & -(2^{n-1} - 1) \cdot 1 + 2^{n-1} \cdot \sum_{i=1}^{n-1} 0 \cdot 2^{-i} \\ & = -(2^{n-1} - 1) \end{aligned} \right\} \quad \left. \begin{aligned} & -(2^{n-1} - 1) \cdot 0 + 2^{n-1} \cdot \sum_{i=1}^{n-1} 1 \cdot 2^{-i} \\ & = 2^{n-1} (1 - 2^{-(n-1)}) = 2^{n-1} - 1 \end{aligned} \right\} \\ & \text{---} \end{aligned}$$

Zahlendarstellungen -- allgemein

Es gibt eine Fülle von weiteren Zahlendarstellungen (BCD, redundante Zahlendarstellungen, Residuendarstellungen, ...). Im allgemeinen ist eine Zahlendarstellung eine Funktion

$$d_n : \mathcal{B}^n \rightarrow \mathcal{Z} \quad (\mathbf{Q}, \mathbf{R}, \dots)$$

die Bitstrings als Zahlen interpretiert. Betrachten wir nun Darstellungen ganzer Zahlen, so scheinen folgende Eigenschaften wünschenswert:

1. Der Zahlenbereich $d_n(\mathcal{B}^n)$ sollte ein Intervall sein.
2. $0 \in d_n(\mathcal{B}^n)$
3. Symmetrie: $d_n(\mathcal{B}^n) = [-a:a]$
4. d_n sollte injektiv und total sein.

Zahlendarstellungen ff

Die von uns eingeführten Darstellungen erfüllen alle die Eigenschaft 1 und 2.

Symmetrie gilt nur für bv und b_1 ,

Injectiv sind nur u und b_2 , denn

$$b_{1,n}(0 \cdots 0) = 0$$

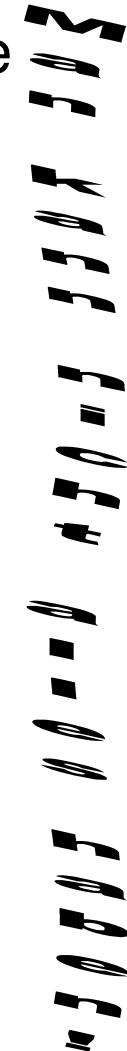
$$= -(2^{n-1} - 1) + (2^{n-1} - 1)$$

$$= -(2^{n-1} - 1) \cdot 1 + 2^{n-1}(1 - 2^{-(n-1)})$$

$$= -(2^{n-1} - 1) \cdot 1 + 2^{n-1} \cdot \sum_{i=1}^{n-1} 1 \cdot 2^{-i}$$

$$= b_{1,n}(1 \cdots 1) \quad \text{„negative 0“}$$

$$bv_n(0 \cdots 0) = 0 = bv_n(10 \cdots 0)$$



3.2.3 Gleitkommazahlen

Reelle Zahlen kann man wegen der Überabzählbarkeit nicht mehr vollständig im Rechner darstellen. Andererseits möchte man mit sehr kleinen (Atomphysik) oder riesig großen (Astronomie) Zahlen rechnen, wobei meist nur eine Näherung der tatsächlichen Zahl genügt.

Bei Handrechnungen kennen wir die Darstellung durch eine **Mantisse** und einen **Exponenten**, der die Größe anpaßt:

$$a \cdot 10^e \text{ bzw. } a \cdot 10^{-e}$$

Geht man davon aus, dass man damit eine reelle Zahl $a' \cdot 10^e$ nur durch wenige Stellen in a annähert mit $a \cdot 10^e$, ist dies nicht ungeschickt, weil der Exponent beim relativen Fehler verschwindet:

$$\left| \frac{a \cdot 10^e - a' \cdot 10^e}{a' \cdot 10^e} \right| = \left| \frac{a - a'}{a'} \right|$$

Gleitkommazahlen ff

Der relative Fehler hängt also nur von der Zahl der Stellen ab, die man in a kennt, wenn a die bestmögliche Näherung für a' mit dieser Zahl von Stellen ist.

Diese Technik macht man sich auch in Rechnern zunutze, indem man ein Raster über die reellen Zahlen legt, das durch Mantissen und Exponenten fester Länge aufgespannt wird. Es gibt einige Freiheitsgrade zur Implementierung von Operationen und zum Runden, sowie zur Festlegung von Mantissen- und Exponentendarstellung und -länge, die wir in dieser Vorlesung nicht erarbeiten wollen.

Um Probleme bei Portierungen von Programmen zu mildern legt der **IEEE Standard 754** diese Freiheitsgrade heute weitgehend fest:

Gleitkommazahlen

Allgemein ist eine Gleitkommadarstellung eine Abbildung

$$fp: \mathbf{B}^{m+e} \rightarrow \mathbf{R}$$

folgender Form:

$$fp(a, b) := d_m(a) \cdot 2^{d_e(b)}$$

wobei d_m , d_e Zahlendarstellungen für die Mantisse und den Exponenten sind. Folgende Freiheitsgrade sind festzulegen:

- m - die Mantissenbreite
- e - die Exponentenbreite
- d_m - die Mantissendarstellung, und
- d_e - die Exponentendarstellung

Dazu betrachten wir zunächst ein paar Zielsetzungen:

Gleitkommazahlen -- Zielsetzungen

Folgende Zielsetzungen sind erstrebenswert. Bei der Festlegung der Parameter und des Formates sollte man darauf Rücksicht nehmen:

- Vergleiche $=, >, <, \leq, \geq, =0, >0, <0$ sollten mit Integervergleichern möglich sein!
- Möglichst wenig Mehrdeutigkeiten, d.h. fp „weitgehend injektiv“
- Rundungsfehler kontrollierbar mit der Mantissenlänge
(normalisierte Zahlen)
- Unterstütze Rechnen mit denormalisierten Zahlen
- Balancierter Zahlenbereich, d.h. $\#fp^{-1}([-1, 1]) \approx \#fp^{-1}(\mathbb{R} \setminus [-1, 1])$
- Unterstütze spezielle Symbole zur Fehlerbehandlung

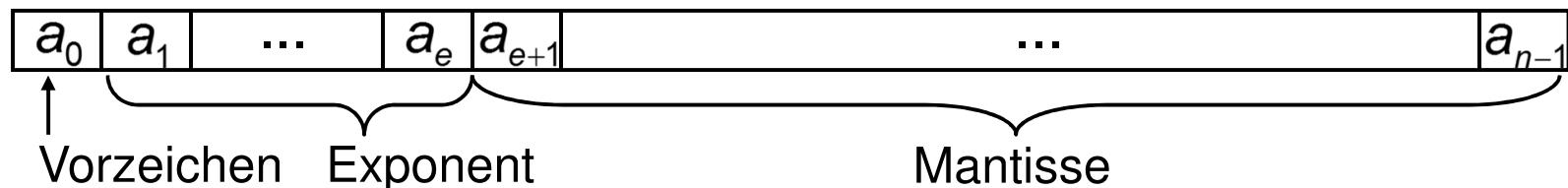
Gleitkommazahlen -- Konsequenzen

Aus diesen Zielsetzungen leitet sich der Standard nahezu zwangsläufig ab:

(1) Integer Vergleich nutzbar

Damit integer Vergleicher direkt nutzbar sind, muß a_0 das Vorzeichenbit sein.

Da ferner grosse Exponenten grosse Zahlen, und kleine Exponenten kleine Zahlen darstellen, muss dem Vorzeichen zuerst der Exponent und dann die Mantisse folgen.



Die Folge $0\dots 0$ sollte ferner der kleinste Exponent sein, $1\dots 1$ der größte Exponent $\Rightarrow d_e = \text{unsigned} - \text{bias}$

$$d_e(a_1 \dots a_e) := \left(2^{e-1} \cdot \sum_{i=0}^{e-1} a_{i+1} \cdot 2^{-i} \right) - \text{bias}$$

Gleitkommazahlen -- Konsequenzen ff

(2) Wenig Mehrdeutigkeiten

Man kann natürlich leicht durch Multiplikation mit 2 oder $1/2$ Exponent gegen Mantisse ausspielen, wenn man für die Mantisse eine Zahlendarstellung nutzt, bei der neben x auch $2x$ oder $1/2 x$ zulässig ist.

Ausweg: „Normalisierte Zahlen“

Wir wählen für die Mantisse eine Darstellung im Bereich $[1,2)$. Damit sind solche Trade offs ausgeschlossen.

$$d_m(a_{e+1} \cdots a_{e+m-1}) := 1 + \sum_{i=1}^{m-1} a_{e+i} \cdot 2^{-i}$$

Man kann also die Mantisse auffassen als eine Binärzahl der Form $1.m\dots m$, bei der stets die erste Stelle =1 ist.
(implizites führendes Bit)

Gleitkommazahlen -- Konsequenzen ff

Damit haben wir bisher schon folgende Festlegung getroffen:

Eine **normalisierte Gleitkommazahl** ist ein Bitstring der Form

$$sb_0 \dots b_{e-1} a_0 \dots a_{m-2}$$

wobei z der benutzte Bias des Exponenten ist:

$$fp(sb_0 \dots b_{e-1} a_0 \dots a_{m-2}) := (-1)^s \cdot \frac{(2^{m-1} + u(a))}{2^{m-1}} \cdot 2^{u(b)-z}$$

$$fp(s, b, a) := (-1)^s \cdot (2^{m-1} + u(a)) \cdot 2^{u(b)-(z+m-1)}$$

Gleitkommazahlen -- Konsequenzen ff

(3) Rundungsfehler

Die Menge der normalisierten Gleitkommazahlen, $fp(\mathbf{B}^{m+e})$, ist total geordnet und sei

$$fp(\mathbf{B}^{m+e}) = \{y_0 < \dots < y_{2^{m+e}-1}\} \subseteq \mathbf{R}$$

Eine **Rundung** $\rho: \mathbf{R} \rightarrow fp(\mathbf{B}^{m+e})$ ist eine Abbildung, die jedem $x \in \mathbf{R}$ eine der benachbarten Gleitkommazahlen y_i, y_{i+1} aus $fp(\mathbf{B}^{m+e})$ zuordnet, mit $x \in [y_i, y_{i+1}]$

Der relative Rundungsfehler für x ist damit stets beschränkt durch

$$\left| \frac{x - \rho(x)}{x} \right| \leq \left| \frac{y_{i+1} - y_i}{y_i} \right|$$

Gleitkommazahlen -- Konsequenzen ff

Rundungsfehler ff

Wir nehmen an, dass beide grösser gleich 0 sind, die anderen Fälle sind analog (den Fall =0 erst mal ausgenommen)

Haben y_i, y_{i+1} gleichen Exponenten, d.h. $y_i = fp(0, e, a)$, $y_{i+1} = fp(0, e, b)$ dann ist $u(b) = u(a) + 1$, und damit

$$\begin{aligned} & y_{i+1} - y_i \\ &= (2^{m-1} + u(b)) \cdot 2^{u(e)-(z+m-1)} - (2^{m-1} + u(a)) \cdot 2^{u(e)-(z+m-1)} \\ &= 2^{u(e)-(z+m-1)} \end{aligned}$$

Also ist dann

$$\left| \frac{y_{i+1} - y_i}{y_i} \right| = \frac{2^{u(e)-(z+m-1)}}{(2^{m-1} + u(a)) \cdot 2^{u(e)-(z+m-1)}} = \frac{1}{(2^{m-1} + u(a))} \leq \frac{1}{2^{m-1}}$$

Gleitkommazahlen -- Konsequenzen ff

Rundungsfehler ff

Haben y_i, y_{i+1} verschiedene Exponenten, dann liegt, da y_i, y_{i+1} benachbart sind folgende Situation vor:

$$y_i = fp(0, e, 1\dots 1), y_{i+1} = fp(0, e', 0\dots 0)$$

mit $u(e') = u(e) + 1$,

und damit

$$\begin{aligned} & y_{i+1} - y_i \\ &= 2^{m-1} \cdot 2^{u(e)+1-(z+m-1)} - (2^m - 1) \cdot 2^{u(e)-(z+m-1)} \\ &= 2^{u(e)-(z+m-1)} \end{aligned}$$

Also ist ebenfalls wieder

$$\left| \frac{y_{i+1} - y_i}{y_i} \right| = \frac{2^{u(e)-(z+m-1)}}{(2^m - 1) \cdot 2^{u(e)-(z+m-1)}} = \frac{1}{2^m - 1} \leq \frac{1}{2^{m-1}} \quad \text{-- } m > 0$$

Gleitkommazahlen -- Konsequenzen ff

(4) Balance des Zahlenbereichs

Die Zahlen 1, -1 haben die Darstellung

$$(-1)^s(2^{m-1} + 0) \cdot 2^{u(e)-(z+m-1)}$$

Also muß für den Exponenten gelten

$$m-1 + u(e) - (z+m-1) = 0 \Leftrightarrow u(e) = z$$

Da sich $u(e)$ im Bereich $[0:2^e-1]$ bewegt, ist - will man genauso viele negative wie positive Zahlen haben - $u(e)$ in der Mitte zu wählen.

Da wir die Exponenten 0 und 2^e-1 für die Darstellung der 0, von denormalen Zahlen und speziellen Symbolen reservieren wollen, ist die "Mitte" bei

$$u(e) \in \{2^{e-1} - 1, 2^{e-1}\}.$$

Wir entscheiden uns für $2^{e-1} - 1$ (mehr große Zahlen) und erhalten so für den Bias

$$z = 2^{e-1} - 1$$

Gleitkommazahlen -- Konsequenzen ff

(5) Denormale Zahlen und spezielle Symbole

Man möchte im Falle eines Exponentenunterlaufs auch noch mit denormalen Zahlen weiterrechnen können. Da wir die 0 ohnehin nicht als normalisierte Zahl interpretieren können, bietet es sich an, für den Exponenten 0...0 die Interpretation zu ändern, und auf die implizite 1 bei der Mantisse zu verzichten.

Vorsicht: Rundungsfehler können enorm ansteigen!

Andererseits gibt es Operationen, die illegal sind, oder aus dem Zahlenbereich herausführen. (Quadratwurzel mit negativem Argument, Division durch 0 ...) Wir möchten dies durch spezielle Symbole anzeigen. Dazu reservieren wir neben 0...0 auch den Exponenten 1...1 für spezielle Symbole.

Damit gilt bei normalisierten Zahlen für die Exponenten folgende

Schranke $e_{min} \leq x \leq e_{max}$

mit

$$e_{min} = 1 - z \quad e_{max} = 2^e - 2 - z$$

IEEE Standard für Symbole

Folgende Tabelle legt die Symbole für Gleitkommazahlen nach dem IEEE Standard 754 fest:

| Exponent | Mantisse | Interpretation |
|--------------------------|--------------|-----------------------------|
| 0...0 | 0...0 | ± 0 |
| 0...0 | $\neq 0...0$ | denormalisierte Zahl |
| $\in [e_{min}, e_{max}]$ | beliebig | normalisierte Zahl |
| 1...1 | 0...0 | $\pm \infty$ |
| 1...1 | $\neq 0...0$ | NaN (not a number) |

NaN wird bei illegalen Operationen erzeugt. Bei Verlassen der Zahlenbereichs hingegen $\pm \infty$.

Rundungsarten

Die Menge der Gleitkommazahlen bildet ein endliches Raster auf \mathbf{R} , das nicht abgeschlossen ist unter den arithmetischen Operationen. Demnach müssen wir, wenn wir Arithmetik auf Gleitkommazahlen betreiben, die erhaltenen Zwischenergebnisse auf das Gleitkommaraster abbilden.

Eine **Rundung** $\rho: \mathbf{R} \rightarrow fp(\mathbf{B}^{m+e})$ ist eine Abbildung, die jedem $x \in \mathbf{R}$ eine der benachbarten Gleitkommazahlen y_i, y_{i+1} aus $fp(\mathbf{B}^{m+e})$ zuordnet.

Sie muß stets folgende Eigenschaften haben:

- $x \in]y_i, y_{i+1}[$, dann ist $\rho(x) \in \{y_i, y_{i+1}\}$
- $x \in fp(\mathbf{B}^{m+e})$, dann ist $\rho(x) = x$

Der IEEE Standard 754 schreibt folgende Rundungsarten vor:

Rundungsarten nach IEEE Standard 754

- i. Zur 0 runden (Round to Zero):

Für $x \in]y_i, y_{i+1}[$ ist

$$rz(x) := \begin{cases} y_i & y_i \geq 0 \\ y_{i+1} & y_i < 0 \end{cases}$$

- ii. Nach $+\infty$ ($-\infty$) runden (Δ (∇)):

Für $x \in]y_i, y_{i+1}[$ ist

$$\Delta(x) := y_{i+1} \quad (\nabla(x) := y_i)$$

- iii. Zur nächsten Zahl runden (Round to Nearest Even):

Diese Rundungsart ist zugleich „default“ Rundungsart.

Für $x \in]y_i, y_{i+1}[$ ist

$$rne(x) := \begin{cases} y_i & x < y_i + \frac{y_{i+1}-y_i}{2} \\ y_{i+1} & x > y_i + \frac{y_{i+1}-y_i}{2} \\ (1-lsb(y_i))y_i + (1-lsb(y_{i+1}))y_{i+1} & x = y_i + \frac{y_{i+1}-y_i}{2} \end{cases}$$

wobei im Falle, dass x die Intervallmitte bildet, nach dem niedrigsten signifikanten Bit der Mantisse (lsb) entschieden wird. Man rundet zu der Zahl y mit $lsb(y) = 0$. (nearest even).

Charakteristika von Gleitkommazahlen

Aus den bisherigen Betrachtungen ergeben sich für konkrete Wortbreiten folgende Charakteristika:

Sei n die Wortbreite

m die Mantissenbreite (inclusive Vorzeichenbit!)

e die Exponentenbreite

Dann gilt $n = m + e$.

Genauigkeit: Wir haben gesehen, dass wir bei normalisierten Zahlen einen relativen Fehler von $\leq 2^{-(m-1)}$ garantieren können.

Bias: der Bias ergab sich direkt aus der Exponentenbreite e als
 $bias = 2^{e-1} - 1$

e_{min}, e_{max} : Es war $e_{min} = -bias + 1 = -2^{e-1} + 2$

$$e_{max} = 2^e - 2 - bias = 2^e - 2 - 2^{e-1} + 1 = 2^{e-1} - 1 = bias$$

Man muss nun nur noch, bei fester Wortbreite m gegen e aushandeln. Der IEEE Standard macht dazu folgende Vorschläge:

IEEE Standard für Mantisse, Exponent

Folgende Tabelle legt die Mantissen, Exponenten trade offs für Gleitkommazahlen nach dem IEEE Standard 754 fest:

| Format | single | single extd. | double | double extd. |
|------------------|--------|--------------|--------|---------------|
| Mantisse | 24 | ≥ 32 | 53 | ≥ 64 |
| $e_{max} = bias$ | 127 | ≥ 1023 | 1023 | ≥ 16383 |
| e_{min} | -126 | ≤ -1022 | -1022 | ≤ -16382 |
| Wortbreite | 32 | ≥ 43 | 64 | ≥ 79 |

Spezielle Symbole und Fehlerbehandlung

Ziele:

Ein Gleitkomma-Rechenwerk sollte

- Fehlerbedingungen spezifizieren
- ggf. mit speziellen Symbolen weiterrechnen

Beispiele:

Bei einer Division einer Zahl $\neq \pm \infty$ durch 0 sollte der Wert $\pm \infty$ zurückgegeben werden.

Hingegen bei der Quadratwurzel aus einer negativen Zahl das Symbol NaN.

Es macht durchaus Sinn, falls der Benutzer keine Traps gelegt hat, mit $\pm \infty$ Arithmetik zu betreiben:

$$\infty + \infty = \infty ; \infty * \infty = \infty ; -\infty + -\infty = -\infty ; 1 / \infty = 0 ; \dots$$

aber: $\infty - \infty = \text{NaN}$; $\infty / \infty = \text{NaN}$; ...

Alle Operatoren und Funktionen (sin,cos,...) sollten strikt auf NaN sein, d.h. ist ein Argument NaN liefern sie NaN als Ergebnis.

Flags für Benutzer-Traps

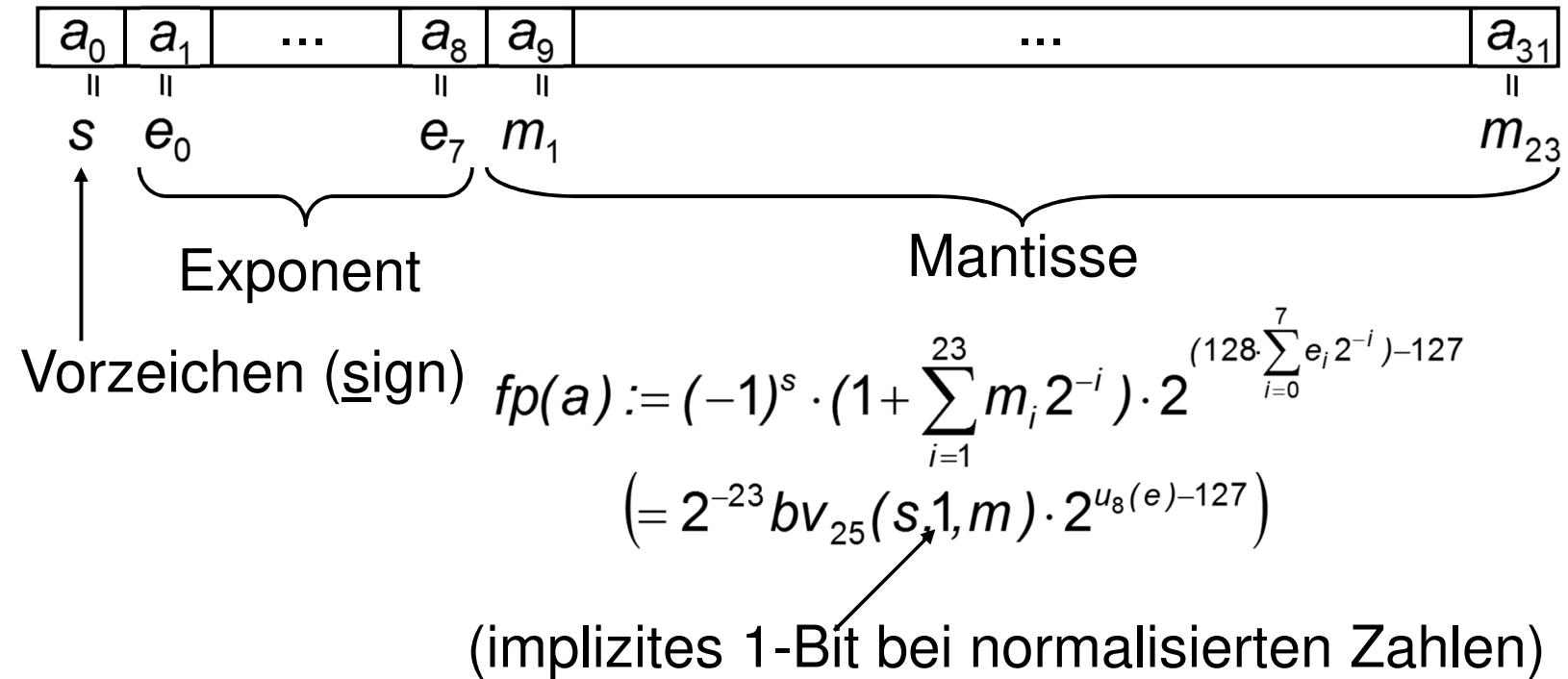
Folgende Flags sollen nach IEEE Standard 754 von einem Gleitkomma-Rechenwerk geliefert werden, damit man ggf. über benutzerspezifische Masken in gewissen Situationen Traps (Ausnahmebehandlungen) auslösen kann.

- underflow -- Ergebnis wurde denormal
- overflow -- Ergebnis $\pm \infty$
- divide by zero
- inexakt -- Ergebnis wurde gerundet
- invalid -- Ergebnis NaN

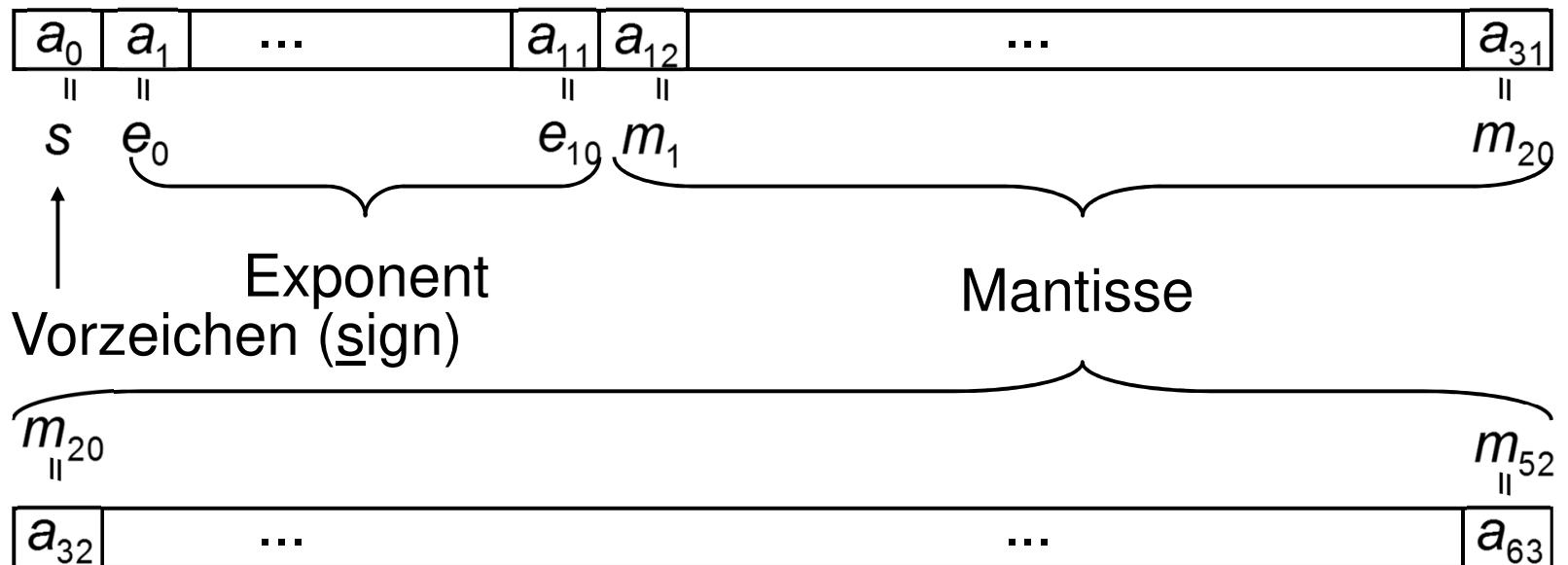
Die Flags sollen bei Vorliegen entsprechender Bedingungen gesetzt werden, und auch bei folgenden Operationen gesetzt bleiben, bis sie explizit zurückgesetzt werden **sticky flags**.

Zusammenfassung: Float

Single precision floating point numbers



Zusammenfassung : double



$$\begin{aligned}
 fp(a) := & (-1)^s \cdot \left(1 + \sum_{i=1}^{52} m_i 2^{-i}\right) \cdot 2^{\left(\sum_{i=0}^{11} e_i 2^{-i}\right) - 1023} \\
 & \left(= 2^{-53} b v_{54}(s, 1, m) \cdot 2^{u_{11}(e) - 1023}\right)
 \end{aligned}$$

3.2.4 Zeichen

Eine Zeichendarstellung ist meist gegeben durch eine injektive Abbildung $c : A \mapsto \mathcal{B}^k$

die Zeichen in einem Zeichensatz A einen Code zuordnet. Diese Abbildung kann man in sog. Codetabellen nachlesen. Sie legt typischerweise die Codes für Buchstaben und Sonderzeichen (EOT,LF,CR,...) fest. Meist ist diese Abbildung nicht surjektiv, d.h. bestimmte Bitvektoren haben keine Bedeutung als Zeichen. Diese Redundanz kann anwendungsspezifisch für weitere Sonderzeichen (Grafik, Gerätesteuerung,...) oder zur Fehlertoleranz (Parity Bit) benutzt werden.

Gebräuchliche Codes sind z.B. **ASCII** oder **EBCDIC**. In unserem Kulturraum genügt $k = 8$ um alle notwendigen Zeichen zu kodieren (1 Byte).

Zeichen

Folgende Eigenschaften für Codes sind sinnvoll (gelten aber nicht immer!)

$$u_k(c(0)) < \dots < u_k(c(9)) < u_k(c(A)) \dots < u_k(c(Z)) < u_k(c(a)) \dots < u_k(c(z))$$

Damit entspricht die Interpretation der Codes als Zahl der gewohnten lexikographischen Ordnung.

Nützlich ist es auch, wenn die Zeichen dicht liegen, weil man dann Konvertierungen leichter implementieren kann, d.h. die Codes bilden Intervalle:

$$u_k(c(\{0, \dots, 9\})) = [u_k(c(0)) : u_k(c(9))]$$

Dann erhält man den Code für die Ziffer i z.B. durch

$$c(i) = u_k^{-1}(u_k(c(0)) + i)$$

3.2.5 Der Krieg der Endians

Wir benötigen viel weniger Platz, um Zeichen zu speichern, als um Zahlen zu speichern. Bei Zahlen bietet es sich ferner an, verschiedene Genauigkeiten anzubieten, sowohl bei reellen Zahlen (double, float) als auch bei ganzen Zahlen (Multimedia).

Problem: Unterstütze den Zugriff auf verschieden große Teilworte.

Die kleinsten adressierbaren Teilworte sind sinnvollerweise Bytes, da sie für Zeichensätze benutzt werden. Andere Teilworte werden dann als Folgen von Bytes betrachtet. Ein k -Byte Teilwort mit Adresse a besteht dann aus den Bytes mit Adressen $a+0, a+1, \dots, a+k-1$

In welcher Reihenfolge legt man diese Bytes in einem Teilwort ab?

Der Krieg der Endians ff

Es gibt zwei ebenso einfache wie sinnvolle Antworten darauf:

Größere Teilworte stellen Zahlen dar, das heißt sie haben die Bits von signifikant (0) nach weniger signifikant ($n-1$) geordnet. Man kann nun die Bytes mit kleinerer Adresse auf

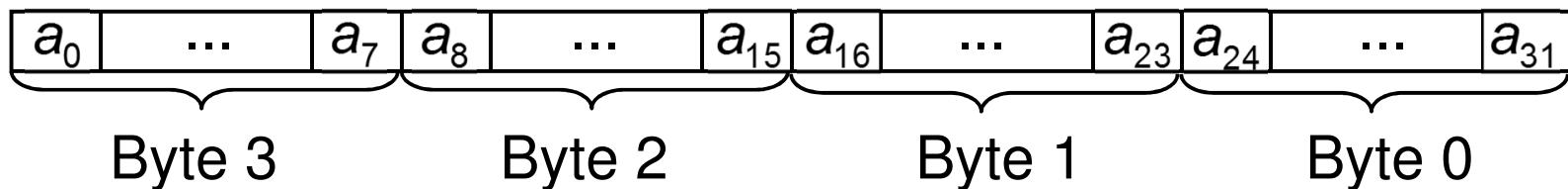
- die weniger signifikanten Positionen des Teilworts
(little endian), oder
- die mehr signifikanten Positionen des Teilworts
(big endian)

legen.

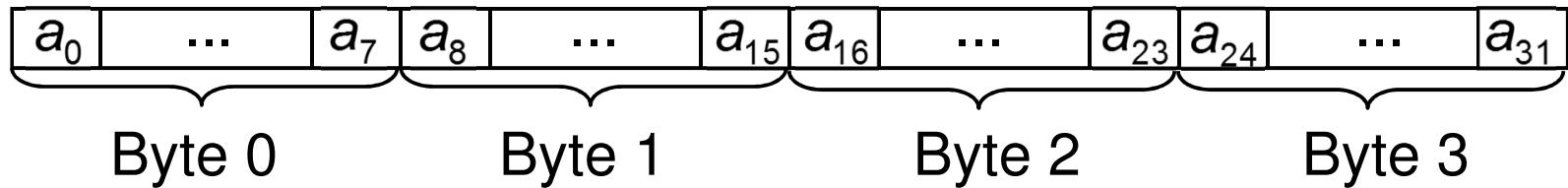
Der Krieg der Endians ff

Beispiel: 32-bit Maschine

LittleEndian (I80x86, VAX)



BigEndian (IBM370, Motorola 680x0, Hitachi H8)



Solange man Daten zwischen Maschinen gleichen Typs austauscht, sind beide Prinzipien gleich gut. Probleme treten bei wortorientierten Datensätzen zwischen Maschinen unterschiedlichen Typs auf.

Alignment

Bei byteorientierten Speichern beziehen sich die Adressen von Speicherinhalten stets auf Bytes. Beim Zugriff auf größere Teilworte (Halbworte (16b), Worte (32b), Doppelworte (64b)) hat man nun zwei Möglichkeiten:

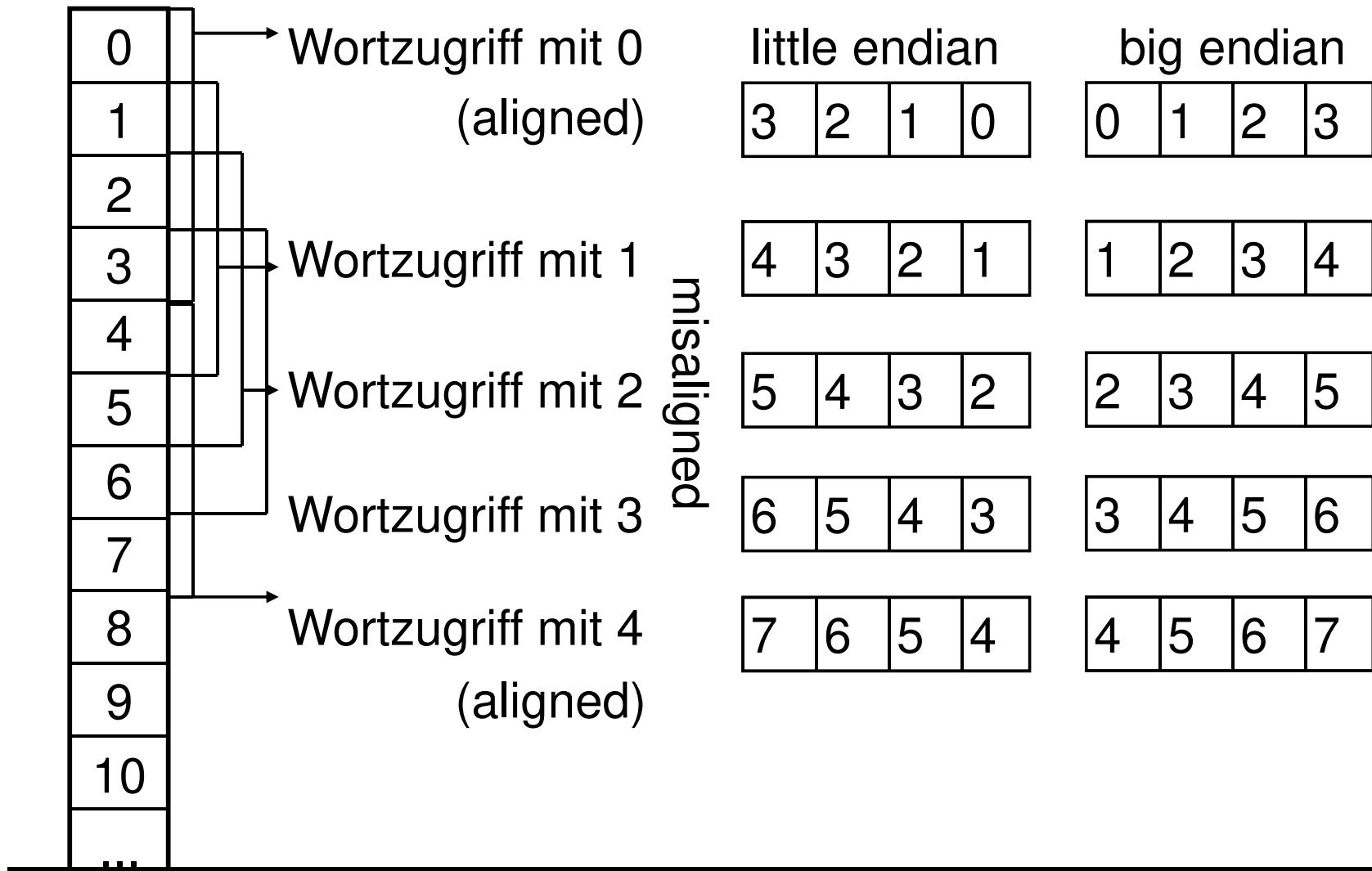
- Aligned access:

Teilwortadressen müssen stets ein Vielfaches der Länge in Bytes sein, d.h. $a \dots a0$ (Halbworten) oder $a \dots a00$ (Worte) oder $a \dots a000$ (Doppelworte), oder

- Misaligned access: Teilworte dürfen auf beliebigen Positionen beginnen.

Misalignment ist schwieriger zu realisieren und kostet Zugriffszeit. (Erklärung folgt in 3.3)

Alignment -- Veranschaulichung



Ausblick

Wir wissen schon etwas mehr darüber, wie die Objekte unserer Maschine nun aussehen müssen:

- **Adressen:**

Adressen stellen wir durch Worte als unsigned Zahlen dar.

- **Befehle:**

Befehle müssen auch in Maschinenworten oder Vielfachen davon kodiert werden. Ferner kennen wir nun mehrere Sorten von Objekten: signed und unsigned Numbers, einfach und doppelt genaue Gleitkommazahlen, Bytes, Halbworte, ...

Zu all diesen Objekten benötigen wir entsprechende Verknüpfungsbefehle wie Arithmetik auf den entsprechenden Zahlen (signed, unsigned, float, double), oder Logikverknüpfungen, Schiebebefehle etc. auf Maschinenworten....

3.3 Speicherstrukturen

Zur Vorlesung Rechenanlagen

SS 2019



Multiportspeicher

Wir haben Schaltungen kennengelernt, die Zustände 0 und 1 halten können (z.B. D-Latches). Um große Speicher bauen zu können, müssen diese Elementarbausteine sehr klein sein und speziell verschaltet werden.

Bei Halbleiterspeichern nutzt man dazu sehr ausgefeilte Strukturen, das Grundprinzip des Aufbaus ist aber sehr ähnlich zu dem, was wir hier vorstellen, so dass uns dies als Orientierung dienen kann.

Unsere Maschine WüRC hatte Befehle, in denen bis zu 2 Speicherworte geändert und bis zu 3 Speicherinhalte gelesen wurden:

Beispiel: ADD Ri, Rj, Rk

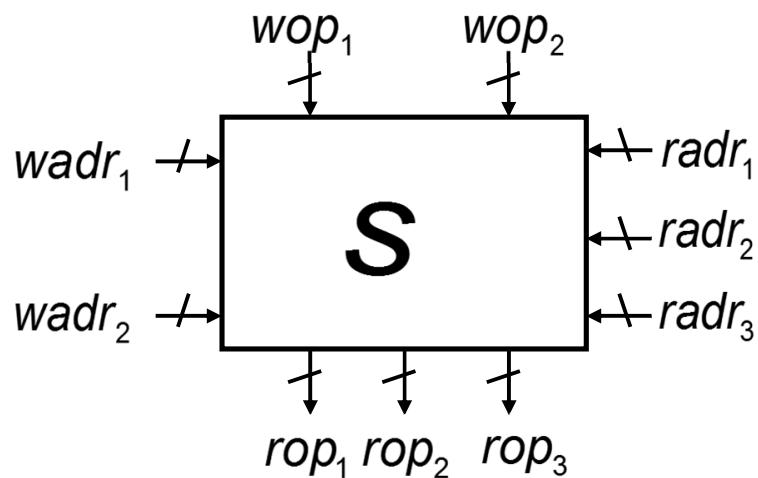
lesen: $pc, Reg[j], Reg[k]$

schreiben: $pc, Reg[i]$

Multiportspeicher ff

Wir bräuchten zur direkten Implementierung der Maschine also einen Multiportspeicher, konkret einen

„3-Lese-2-Schreib-Speicher“:



$$rop_i := s[u_n(radr_i)]$$

$$s[u_n(wadr_i)] := wop_i$$

Es seien

$wadr_i$: Adressen der Schreiboperanden wop_i , als unsigned Zahlen.

$radr_i$: Adressen der Lese-operanden rop_i , als unsigned Zahlen.

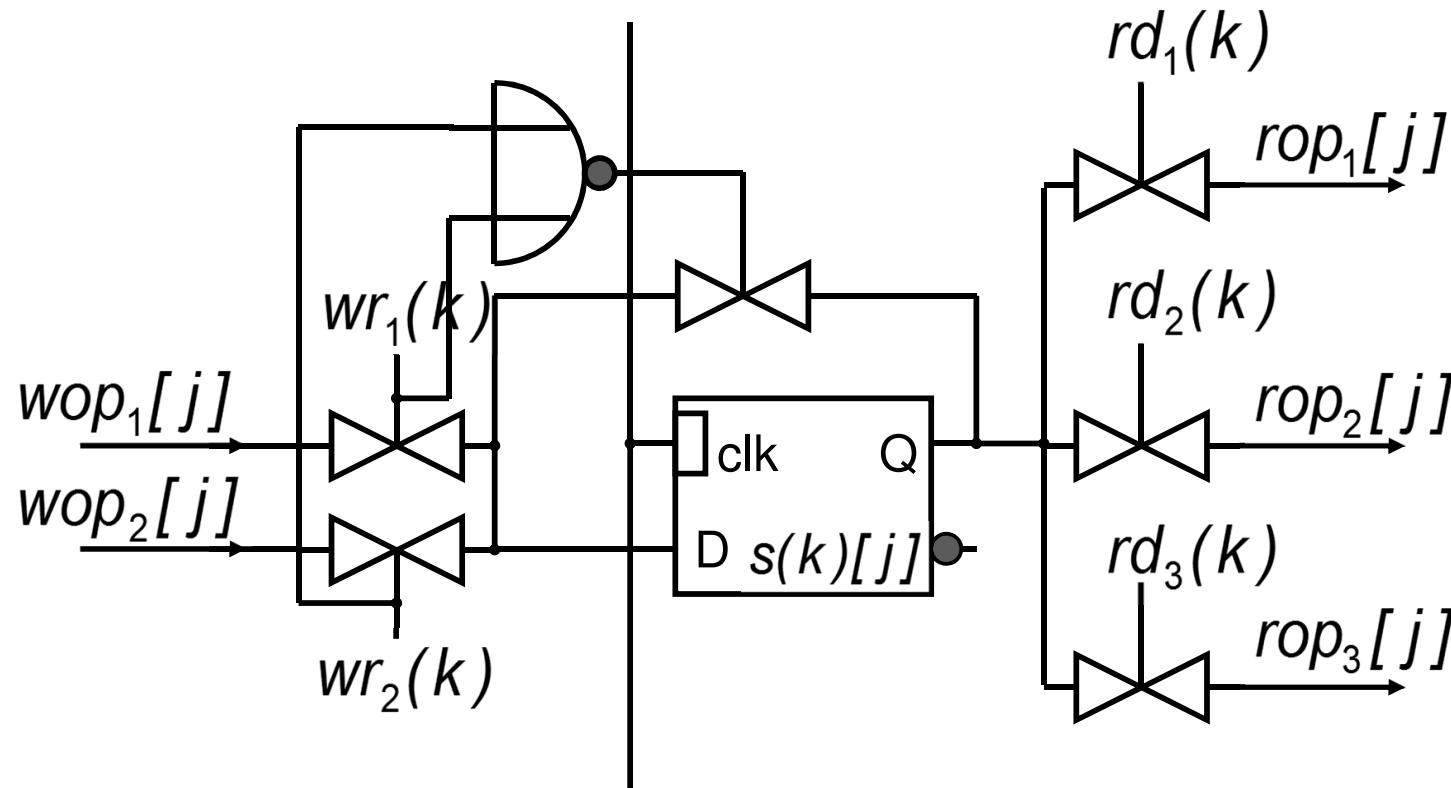
Multiportspeicher ff

Realisierung:

Führe die Schreibleitungen $wop_i[j]$ für $i=1,2$ an alle Latches, die in einer Zelle k das Bit j mit $j=0,\dots,n-1$ speichern, und lege die Leseleitungen $rop_i[j]$ an deren Ausgänge.

Wir müssen nun nur noch entscheiden, ob der Eingang oder Ausgang mit einer Schreib-, Leseleitung verbunden werden soll. Dazu nehmen wir an, dass wir zu jeder Zelle k eine Auswahlleitung $wr_i[k]$ und $rd_i[k]$ haben, die 1 sind, genau dann, wenn die Zelle k vom i -ten Schreib- oder Leseoperand angesprochen werden soll. Dies führt zu folgender Verschaltung für ein Bit j in einer Zelle k :

Multiportspeicher: Grundzelle



Schreibzugriffe an Port 1 und 2 müssen stets auf verschiedenen Zellen stattfinden, sonst hat man am Eingang der Zelle ggf. einen Kurzschluß!

Der Decoder

Die Schalter unserer Multiportzelle werden angesteuert mit

$$wr_i(k) = 1 \Leftrightarrow u_n(wadr_i) = k \quad rd_i(k) = 1 \Leftrightarrow u_n(radr_i) = k$$

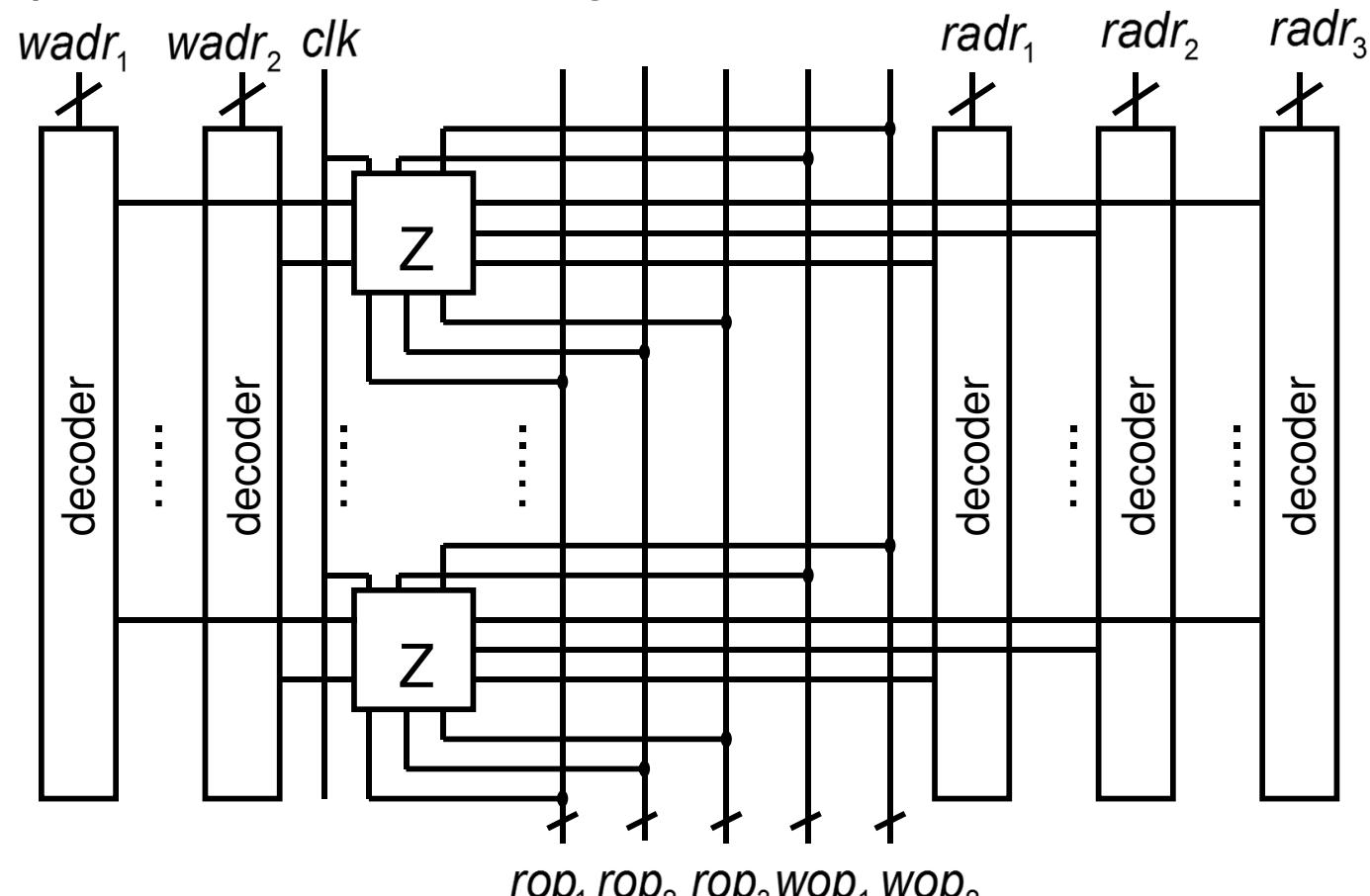
Wir müssen also den Schaltkreis aus Kapitel 2 nehmen, der folgende Dekodierungsfunktion realisiert:

$$\text{decode}_m : B^n \mapsto B^m$$

$$\text{mit } \text{decode}_m(a_0, \dots, a_{n-1}) = (y_0, \dots, y_{m-1}) \Leftrightarrow \forall i : y_i = (u_n(a) = i)$$

Multiportspeicher -- Übersicht

Der Speicher hätte also folgende Struktur:



$\not\perp$: n -bit Leitungsbündel

Multiportspeicher ff

Diese Anordnung müsste planar auf einem oder mehreren Chips untergebracht werden. Würde man dies wie oben skizziert tun, hätte man bei Speichergröße k Worte eine Höhe $\sim k$ und eine Breite $\sim \log k$. Ein unmögliches Streckungsverhältnis!

Die Realisierung einer einzelnen Zelle, die im Aufwand ja am meisten, nämlich mit dem Faktor „Wortbreite“ zu Buche schlägt, ist sehr teuer: Wir benötigen neben dem Latch auch noch ein Gatter und 6 Schalter pro Bit.

Es ist also wenig sinnvoll, solche Speicherstrukturen für wirklich große Speicher zu benutzen:

- hohe Last → lange Schaltzeiten (Fanout, Verdrahtung)
- hohe Kosten pro Bit

Speicherkonzept der Maschine

Deshalb benutzen wir folgendes Speicherkonzept für unsere Maschine:

- einen sehr kleinen Multiport Speicher für 3 Adressbefehle,
- getrennte Speicher für Zellen mit Sonderaufgaben und
- große Single Port Speicher für Transporte (Load/Store).

Die Zellen des kleinen Multiportspeichers nennen wir **GP-Register** (general purpose), die Zellen mit Sonderaufgaben, wie z.B. der Programmzähler, nennen wir **SP-Register** (special purpose).

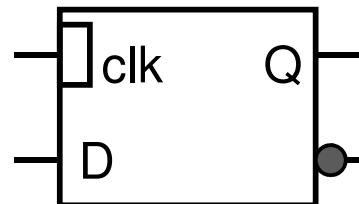
Der Haupteil des Speichers soll nun ein Single Port Speicher sein, der in einem Zyklus entweder ausgelesen oder beschrieben werden kann. Einen solchen Speicher nennen wir auch **RAM** (random access memory).

Struktur eines RAM

Wir wollen zunächst überlegen, wie wir einen RAM Speicher für ein Bit Wortbreite aufbauen können. Andere Wortbreiten erhält man dann durch Parallelschalten von solchen 1-Bit Speicherbausteinen.

- **Die Zelle:**

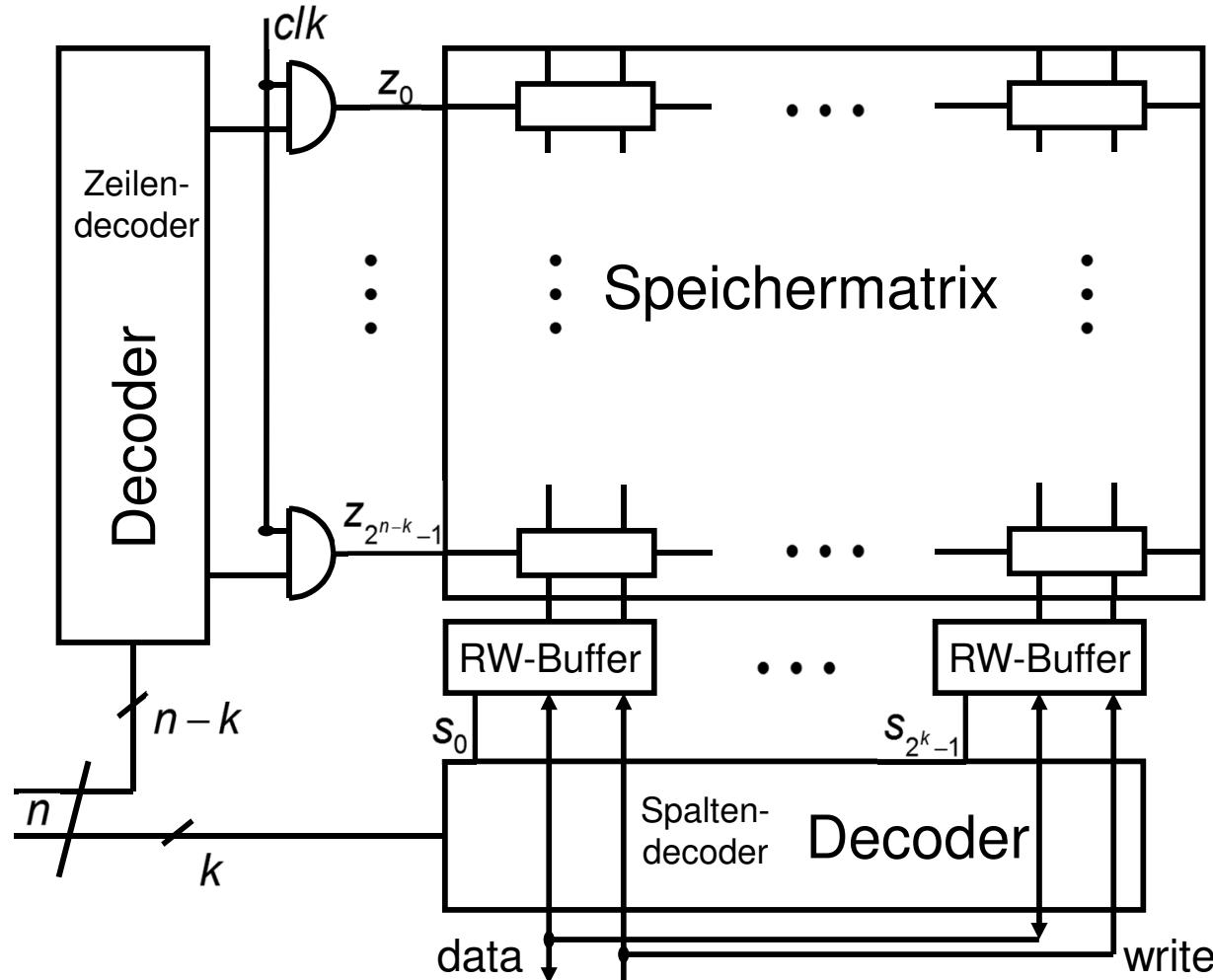
Als Zelle nehmen wir einfach ein statisches oder dynamisches D-Latch:



Mit dieser Zelle lässt sich der Speicher wie folgt aufbauen:

Struktur eines RAM ff

- Grobarchitektur eines RAM

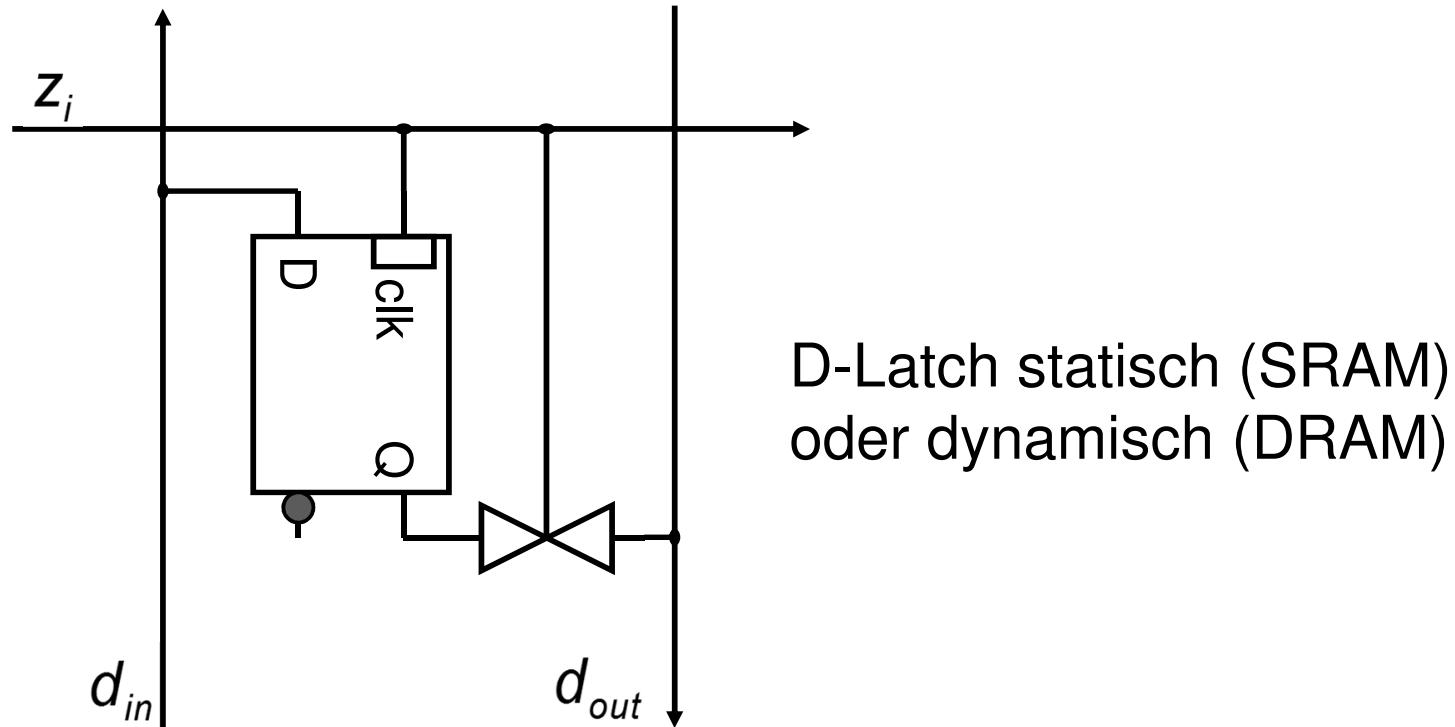


Struktur eines RAM ff

Die Aufspaltung des Decoders in Zeilen- und Spalten-decoder dient dazu, das Streckungsverhältnis des Speichers zu balancieren (Höhe/Breite ~ 1). Ferner werden die Spaltendecodersignale nur zu den RW-Buffern (Read/Write) geführt, die dann entscheiden, welche Spalte auf die bidirektionale Datenleitung gekoppelt wird. In jedem Schreib- oder Lesezyklus werden also alle Spalten über die RW-Buffer geführt, und die durch die decodierte Zeile angewählten Zellen parallel ausgelesen und ggf. wieder aufgefrischt (wichtig bei DRAMs).

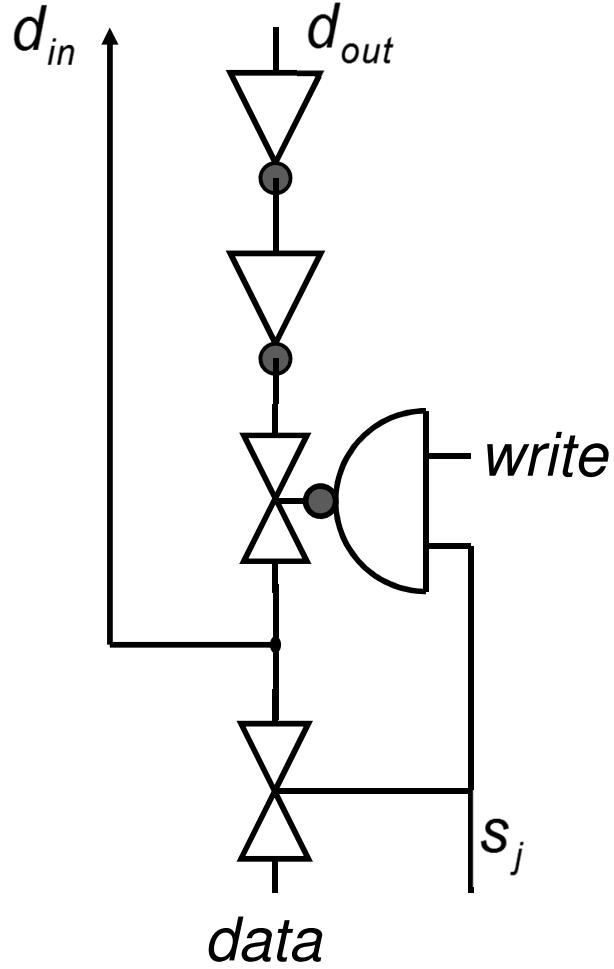
Nun zu den Komponenten:

Zellen der Speichermatrix



Das Zeilensignal des Decoders steuert die Übernahme des neuen Zustands von d_{in} . Der aktuelle Inhalt wird über den RW-Buffer von d_{out} nach d_{in} durchgeschleift und wieder aufgefrischt (lesen) oder von d_{in} überschrieben (schreiben).

Der Read/Write-Buffer



Lesezugriff (write=0):

Inhalt der in der Zeile ausgewählten Zelle wird über die Doppelinverter verstärkt, zurückgeschickt und ausgegeben, falls die Spalte selektiert ist.

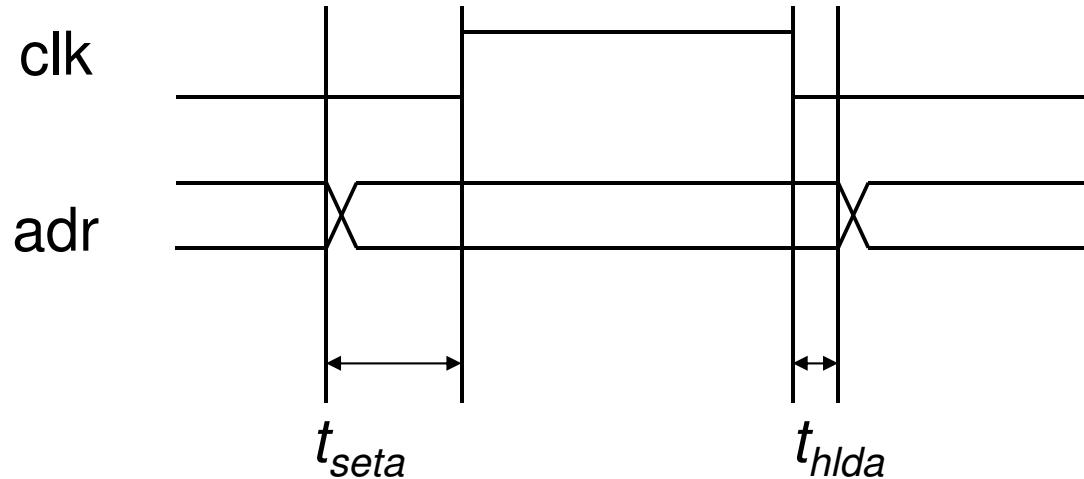
Schreibzugriff (write=1):

Inhalt der in der Zeile ausgewählten Zelle wird über die Doppelinverter verstärkt, zurückgeschickt ($s_j = 0$). Bei ausgewählter Spalte wird aber überschrieben.

Zeitbedingungen

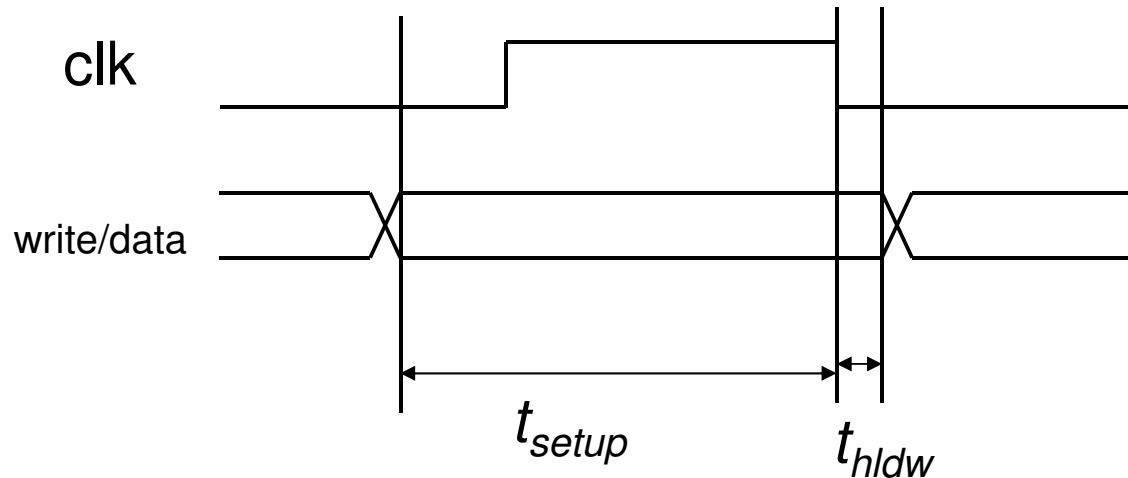
Damit Zugriffe nun korrekt ablaufen muss man Zeitbedingungen einhalten, die den Signallaufzeiten im Speicher Rechnung tragen:

- Dekoderlaufzeiten: Adressen müssen t_{seta} vor dem Clock stabil sein und bleiben, damit die Spalten-/Zeilenauswahl vor Beginn des Zugriffs stabil sind.



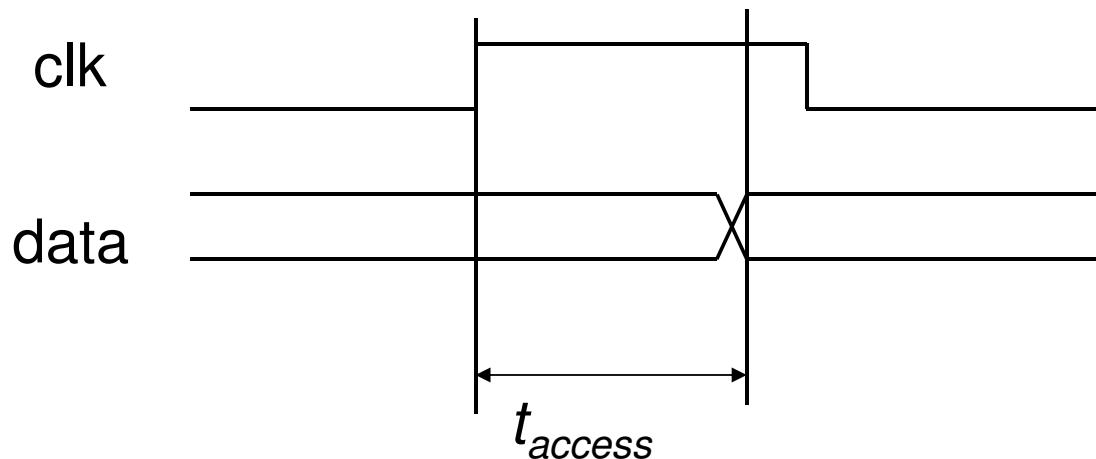
Zeitbedingungen ff

- Matrix- und RW-Bufferlaufzeit: das Kontrollsignal zum Schreibzugriff (write) (sowie das Datensignal (data), falls $\text{write}=1$,) muss t_{setup} vor dem Ende der aktiven Phase des clocks stabil sein und bleiben. Sicher wäre folgende Wahl: $t_{\text{setw}} = \text{Pulsbreite} + t_{\text{seta}}$. Es geht aber je nach Speicher auch kürzer.



Zeitbedingungen ff

- Matrix- und RW-Bufferlaufzeit ff: das Datensignal wird beim Lesezugriff erst t_{access} nach dem Beginn der aktiven Phase des Clocks stabil. Die Clockbreite muss mindestens so breit wie die Laufzeit der Rückführung sein.



Es gibt je nach weiterer interner Unterstützung in der Schaltung Möglichkeiten, die Zugriffszeiten bei bestimmten Zugriffen zu verringern (s. z.B. EDO RAMs).

Speichertypen

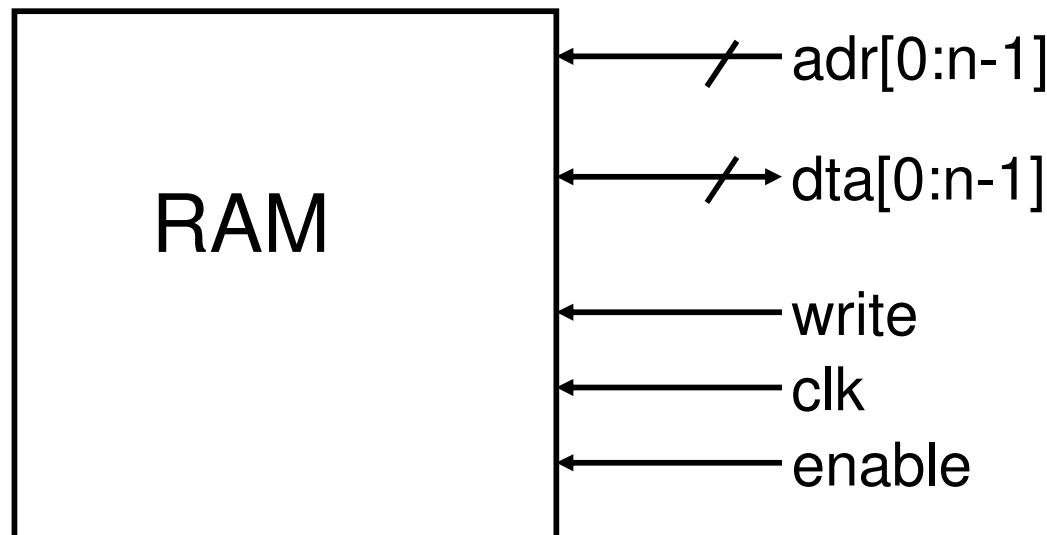
Anmerkungen:

- Man kann die Schaltung auch auf größere Wortbreiten auslegen. Aber die Flexibilität bei Konstruktion von Speicherboards sinkt dadurch (verschiedene Wortbreiten, Fehlerkorrekturbits etc.).
- Ersetzt man die Latches durch Konstanten und verzichtet auf die Schreiblogik, erhält man **ROMs** (Read Only Memories).
- Benutzt man Zellen, die durch eine höhere Spannung permanent auf einen Wert programmiert werden können erhält man **EPROMs** (Electrically Programmable Read Only Memories).

Der Speicher

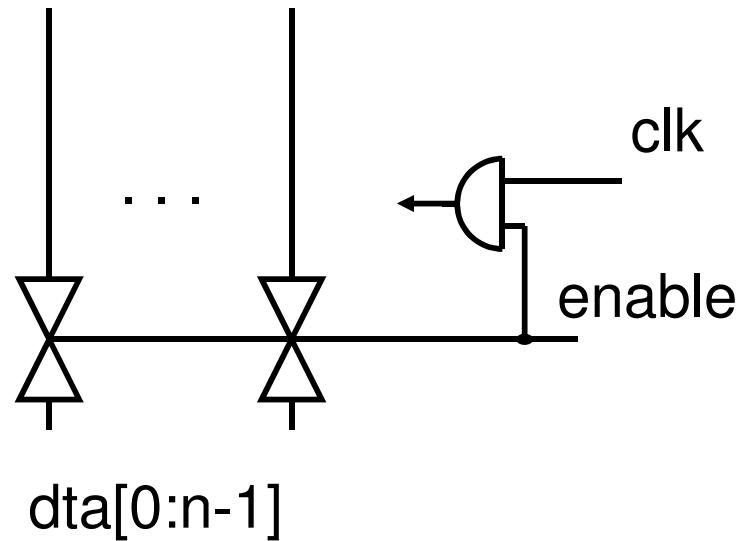
Wir wollen nun annehmen, dass wir Speicherbausteine mit 1-Bit Wortbreite haben, und skizzieren, wie wir damit den Hauptspeicher einer Maschine konstruieren:

Der "Haupt"-speicher kommuniziere mit der Maschine über folgenden einfachen Bus:



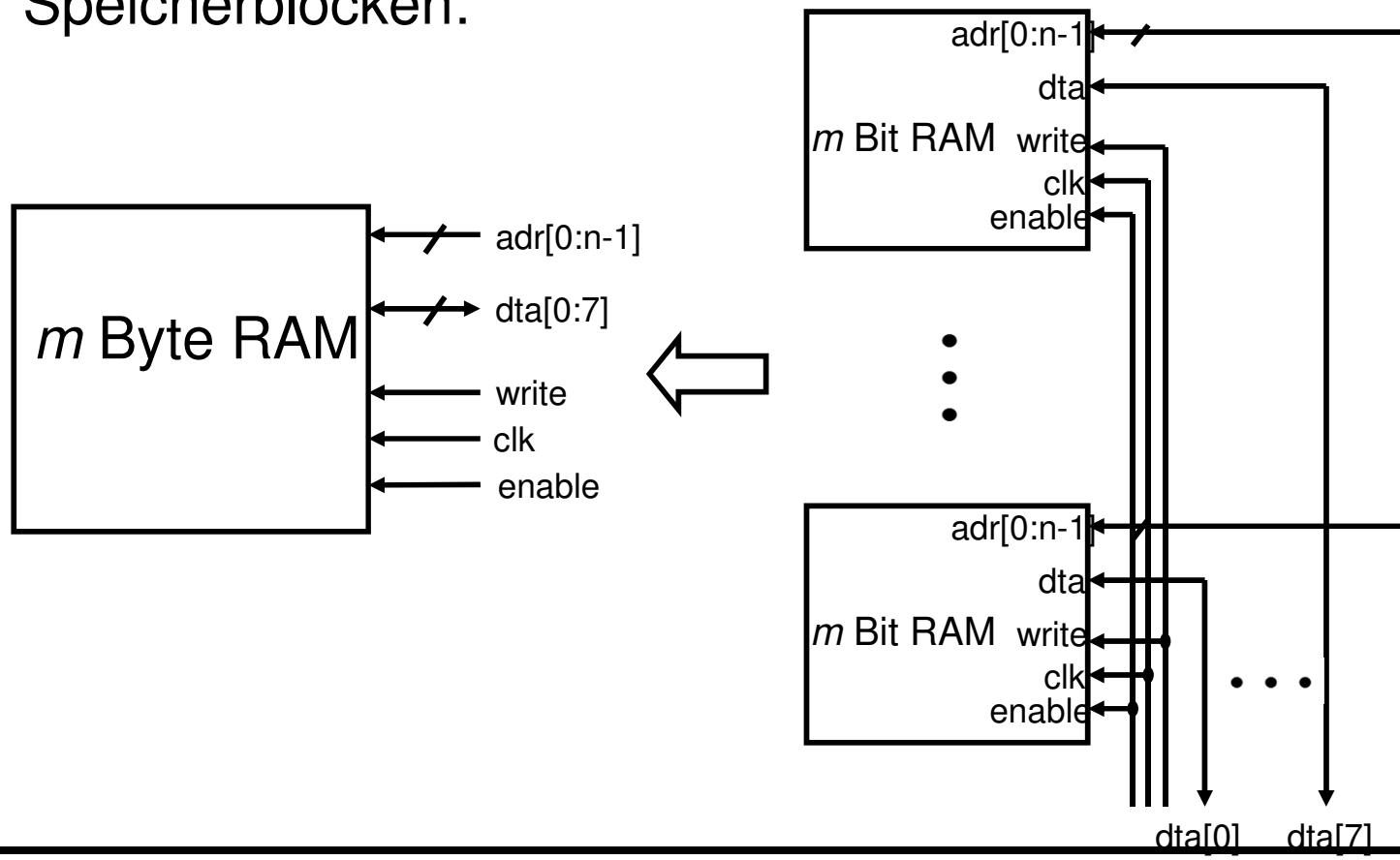
Der Speicher ff

Hinzugenommen haben wir eine enable Leitung, die dazu benutzt werden kann, den Speicher überhaupt erst zum Arbeiten zu ermächtigen. Sie trennt im Zustand 0 die Datenleitungen des Speichers vom Bus, und stellt den Arbeitstakt ggf. ab:



Der Speicher ff

Einen byteorientierten Speicherblock erhalten wir nun durch Parallelschalten von 8 bitorientierten Speicherblöcken:



Der Speicher ff

Der Speicher selbst soll nun Wortzugriffe, aber auch Teilwortzugriffe unterstützen, wobei Byte die kleinste adressierbare Einheit sei. Je nach Vereinbarung ergeben sich dadurch mehr oder weniger einfache Schaltungen. Wir unterscheiden

- aligned access, d.h. Teilworte mit k Bytes ($k=1,2,4,\dots,n/8$) Länge beginnen stets auf einer Adresse, die ein Vielfaches von k ist.
- misaligned access, d.h. Teilworte beginnen an beliebigen Adressen a und setzen sich dann aus den Bytes $a+0, a+1, \dots, a+k-1$ zusammen.

Teilwortzugriffe

Bei einem Teilwortzugriff möchte man ein Register, das ja Wortlänge viele Bits hat, nur mit einem Teilwort laden.

Dazu muss vereinbart werden:

- Wie wird das Teilwort im Register abgelegt?
- Welcher Teil des Registers wird beim Schreibzugriff weggeschrieben?

Man möchte Teilworte natürlich auch als (kleine) Zahlen interpretieren können. Deswegen soll ein Teilwort stets auf den am wenigsten signifikanten Stellen platziert werden (d.h. rechtsbündig). Allerdings hat man dann immer noch die folgenden zwei Ablegungsvarianten:

- **Unsigned**: signifikante Stellen mit 0 auffüllen.
- **Sign extended**: signifikante Stellen mit Vorzeichen füllen.

Teilwortzugriffe -- Beispiele

Big Endian

Byte $sx...x$

unsigned

| | | | |
|---------|--------|---------|----------|
| 0...0 | 0...0 | 0...0 | $sx...x$ |
| 0 ... 7 | 8...15 | 16...23 | 24... 32 |

signexd

| | | | |
|---------|--------|---------|----------|
| S...S | S... S | S... S | $sx...x$ |
| 0 ... 7 | 8...15 | 16...23 | 24... 32 |

Short $i:sa...a \ i+1:tb...b$

unsigned

| | | | |
|---------|--------|---------|----------|
| 0...0 | 0...0 | sa...a | $tb...b$ |
| 0 ... 7 | 8...15 | 16...23 | 24... 32 |

Little Endian

Byte $sx...x$

unsigned

| | | | |
|---------|--------|---------|----------|
| 0...0 | 0...0 | 0...0 | $sx...x$ |
| 0 ... 7 | 8...15 | 16...23 | 24... 32 |

signexd

| | | | |
|---------|--------|---------|----------|
| S...S | S... S | S... S | $sx...x$ |
| 0 ... 7 | 8...15 | 16...23 | 24... 32 |

Short $i:sa...a \ i+1:tb...b$

unsigned

| | | | |
|---------|--------|----------|----------|
| 0...0 | 0...0 | $tb...b$ | sa...a |
| 0 ... 7 | 8...15 | 16...23 | 24... 32 |

Speicher ff

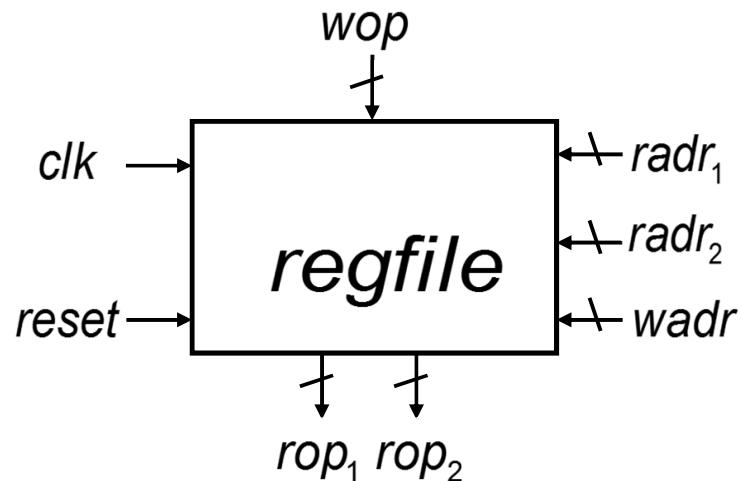
Wir vereinbaren weiter:

- Zwischen Hauptspeicher und Maschine werden stets nur ganze Worte, die das adressierte Teilwort enthalten, übertragen. Der Prozessor sorgt für die richtige Einbettung je nach Befehl. Der Speicher sorgt bei Schreibzugriffen dafür, dass nur das adressierte Teilwort überschrieben wird.
- Aligned access: Das übertragene Wort legen wir stets einfach ab $a = u(adr[0:n-k], 0..0)$ (k - Wortlänge in Bytes), oder
- Misaligned access: Das übertragene Wort beginne stets mit dem Teilwort.

Problem: Bei misaligned access muss man ggf. auch das Inkrement von $adr[0:n-k]$ berechnen (kostet Zeit!)

Die Registerbank

Da unsere Maschine relativ wenige Register hat, werden wir diesen Speicher, wie eingangs beschrieben, durch einen (sehr schnellen, da sehr kleinen) Multiportspeicher ersetzen. Den PC realisieren wir getrennt, d.h. wir benötigen lediglich einen 1-Schreib-2-Lese-Multiportspeicher, den wir aber mit 0 initialisieren können sollten:



Die Adressleitungen $xadr$ haben die Breite $\log(\#Register)$

3.4 Arithmetische und logische Operationen -- die ALU

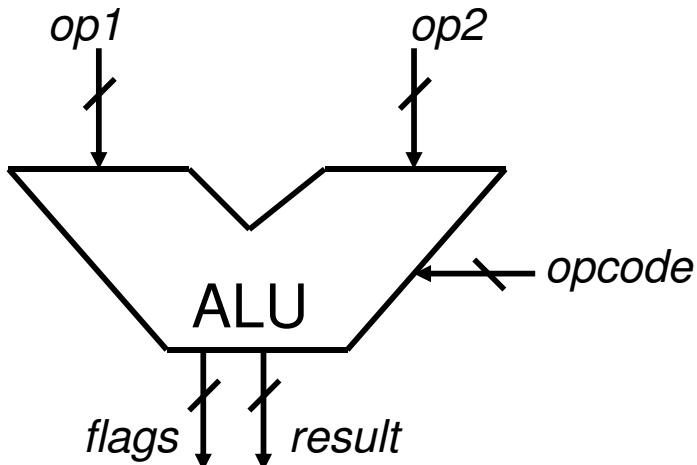
Zur Vorlesung Rechenanlagen

SS 2019



Die ALU

Die meisten Befehle betreffen arithmetische (+,-,...) oder logische (und, oder,...) Verknüpfungen von Maschinenworten. Hinzu kommen noch Vergleiche und Schiebeoperationen. Die Einheit, die im Rechner all diese Funktionen bereitstellt, nennt man **ALU** (arithmetical logical unit). Sie hat nach aussen folgende Schnittstelle:



Die ALU ff

Die Anschlüsse haben folgende Aufgaben:

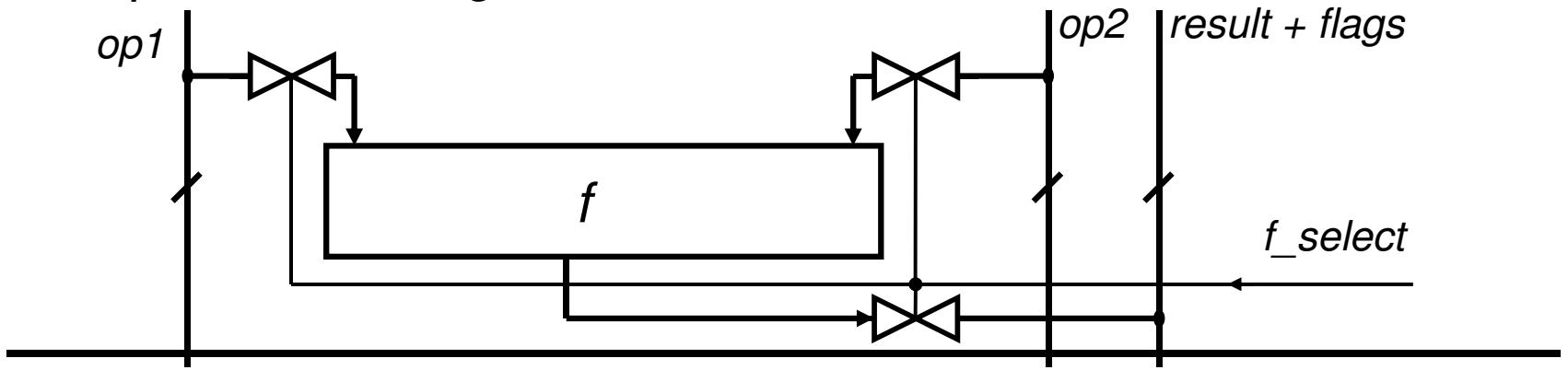
- ***op1,2***: Operanden der auszuführenden Operation
- ***result***: Resultatswort zur Operation
- ***opcode***: Auswählen der auszuführenden Operation
- ***flags***: Statusanzeigen (Fehler, Vergleichsergebnisse,...)

Wir wollen nun versuchen, diesen Baustein möglichst systematisch zu konstruieren:

Funktionsscheiben

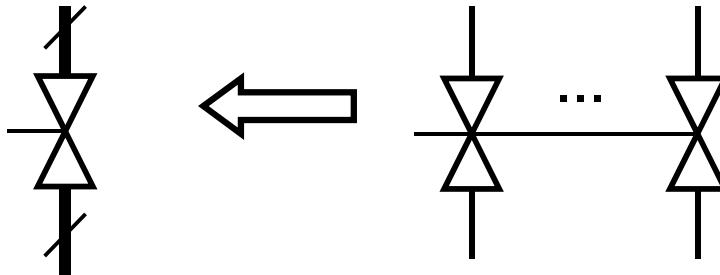
Der erste Schritt besteht darin, dass wir zu jeder Operation einen Schaltkreis bauen müssen, der diese Operation als Schaltfunktion implementiert. Wir nennen einen solchen Baustein daher auch eine **Funktionsscheibe** (function slice) der ALU.

Die Einbettung in die ALU selbst ist dann sehr einfach, wenn wir eine Steuerleitung f_select für die Operation f aus dem Operationscode decodiert haben, die 1 ist, falls die Operation f ausgeführt werden soll:

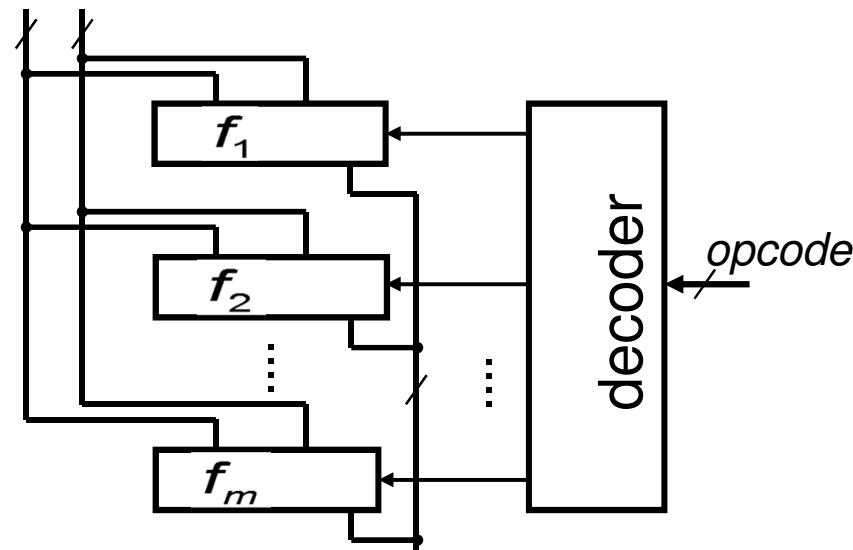


Funktionsscheiben ff

Die Schalter in obiger Zeichnung sind in Wortbreite zu lesen, d.h.



Integriert man diese Ansteuerung in die Scheibe zu f , so erhält man eine sehr einfache Grundstruktur der ALU:



Wir müssen uns also nur noch um die einzelnen Funktionen kümmern!

Funktionsscheiben ff

Bemerkung:

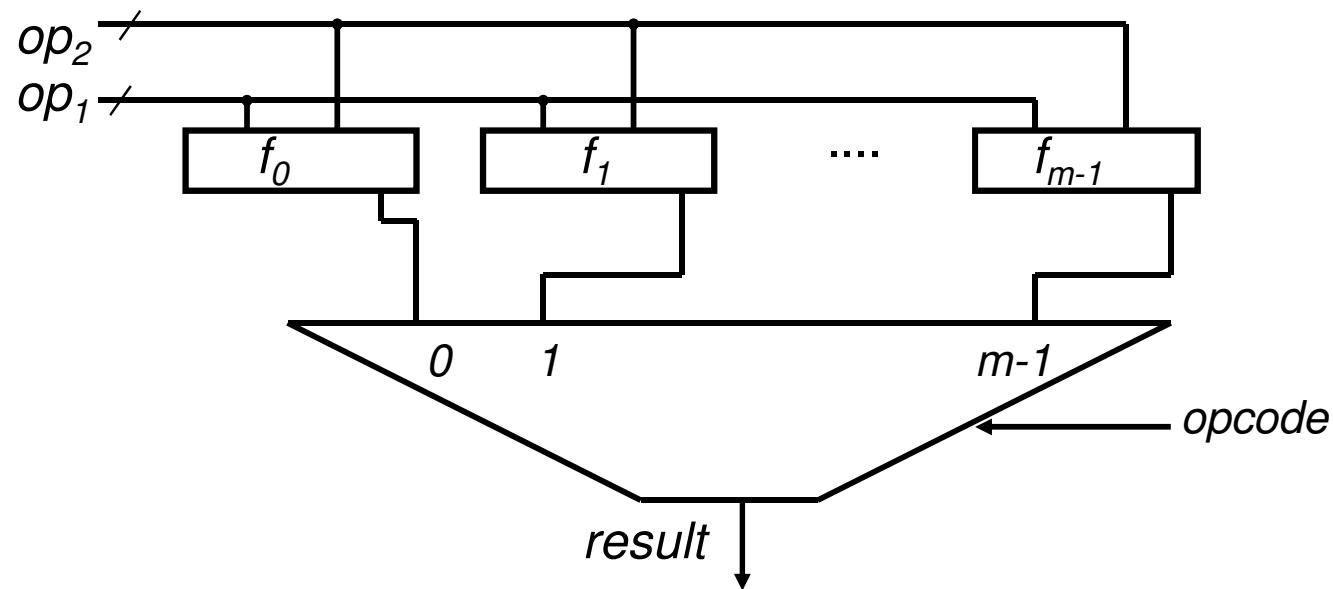
Die Schalter, mit denen die Operanden angelegt werden, scheinen überflüssig zu sein, da ja eine Funktionsscheibe auch dann rechnen kann, wenn ihr Ergebnis nicht benutzt wird.

Aber: Schaltvorgänge in Funktionsscheiben kosten Energie, d.h. rechnen Funktionsscheiben Ergebnisse aus, die nicht gebraucht werden, verbraucht die Schaltung unnötig Energie. Dies kann zu Kühlproblemen (Umschaltvorgänge erzeugen Wärme) oder niedrigen Akku-Laufzeiten führen. Sowohl im High Performance Bereich und erst recht im mobilen Bereich ist man an einer Minimierung der Verlustleistung interessiert.

Funktionsscheiben ff

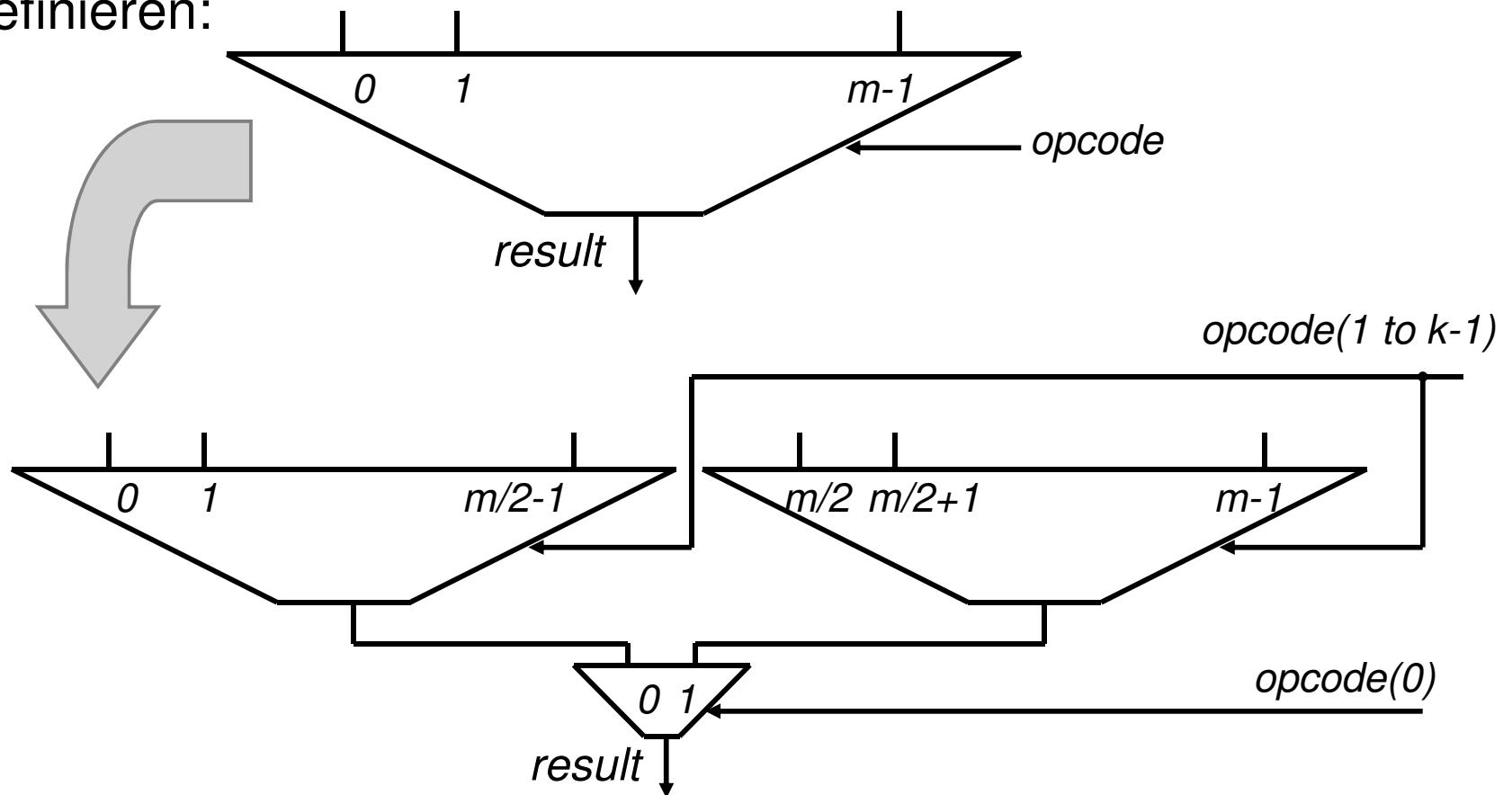
Bemerkung:

Die Resultsleitungen sind in unserem Schema aufgelöste Signale, da sie mit Schaltern angesteuert werden. In Technologien, die nur unaufgelöste Signale zulassen (z.B. FPGAs), kann man statt dieses Schemas eine **Auswahlschaltung** benutzen:



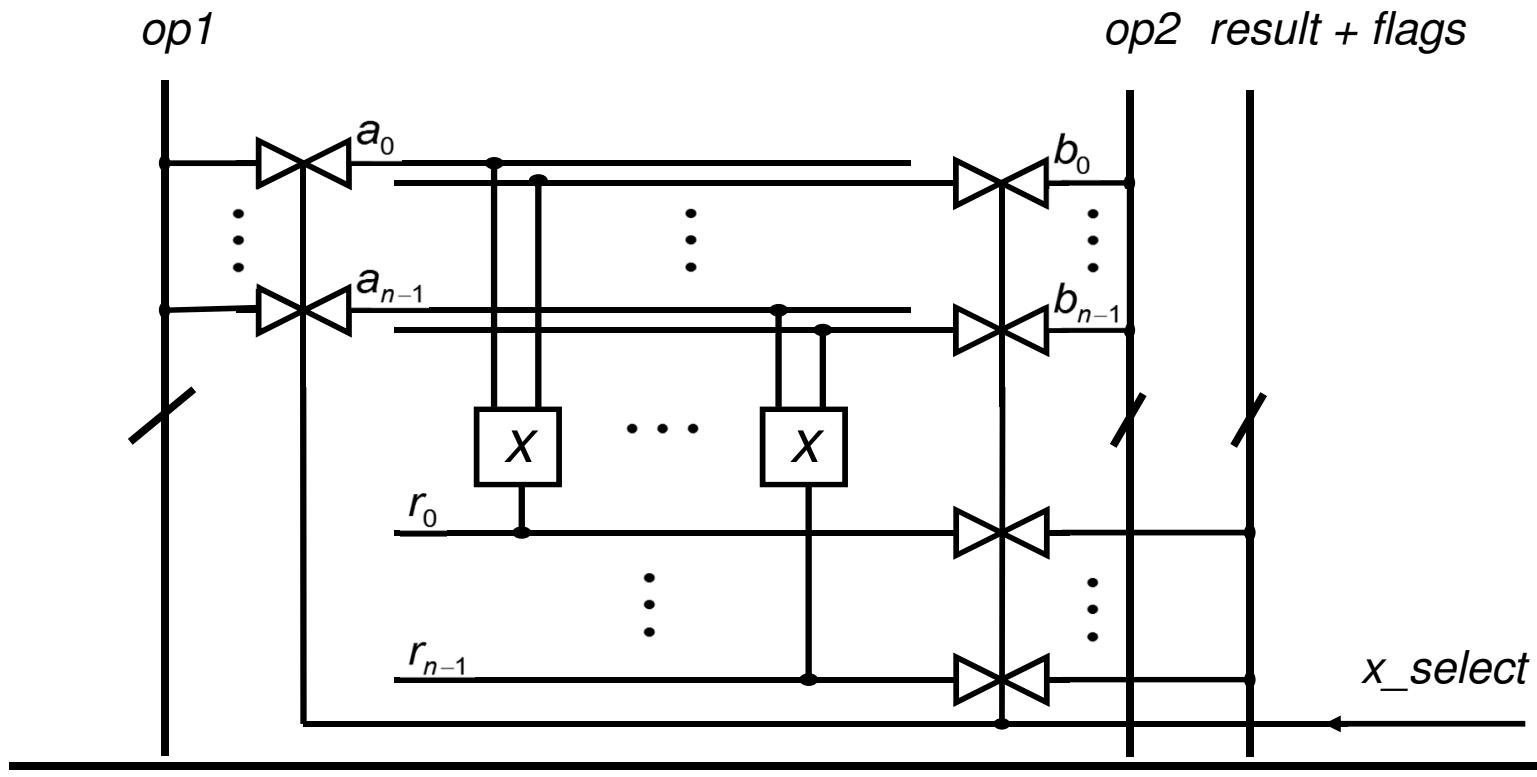
Funktionsscheiben ff

Die Auswahlschaltung kann man generisch in Wortbreite und Zahl der Operationen (Opcode-Breite) leicht rekursiv definieren:



Logikoperationen

Am leichtesten sind wohl die Funktionsscheiben für einfache Logikoperationen zu konstruieren, da diese lediglich komponentenweise boolesche Operationen auf den Bits eines Maschinenworts ausführen müssen:



Addition und Subtraktion

Wir haben schon die Addition von vorzeichenlosen Zahlen betrachtet, d.h. wir wissen, wie eine Funktionsscheibe **ADDU** aussieht, die für zwei Worte a, b der Länge n als Eingänge und ein Wort s der Länge n als Ausgang, sowie ein Flag ov als weiteren Ausgang die folgende Funktion *addu* realisiert:

$$addu : \mathbf{B}^{2n} \mapsto \mathbf{B}^{n+1}, \text{ mit}$$

$$addu(a, b) = (ov, s) \Leftrightarrow 2^n ov + u(s) = u(a) + u(b)$$

Da $u(s) \leq 2^n - 1$, zeigt das Flag ov offenbar an, dass die Summe nicht mehr in einem Maschinenwort dargestellt werden kann, da sie größer als $2^n - 1$ ist. Wir nennen dieses Flag auch die **Überlaufanzeige** (overflow flag).

Addition ganzer Zahlen

Bis jetzt können wir aber nur unsigned numbers addieren.

Problem:

Gegeben sei eine brauchbare Schaltung zur Addition von vorzeichenlosen Zahlen in Darstellung u .

Kann man daraus Additionsschaltungen für ganze Zahlen im Einer- oder Zweierkomplement oder nach Betrag und Vorzeichen gewinnen?

Dazu müssen wir wissen, wie diese Darstellungen zusammenhängen. In den Übungen klären wir:

$$(1) \quad b_2(a) = b_1(a) - a_0$$

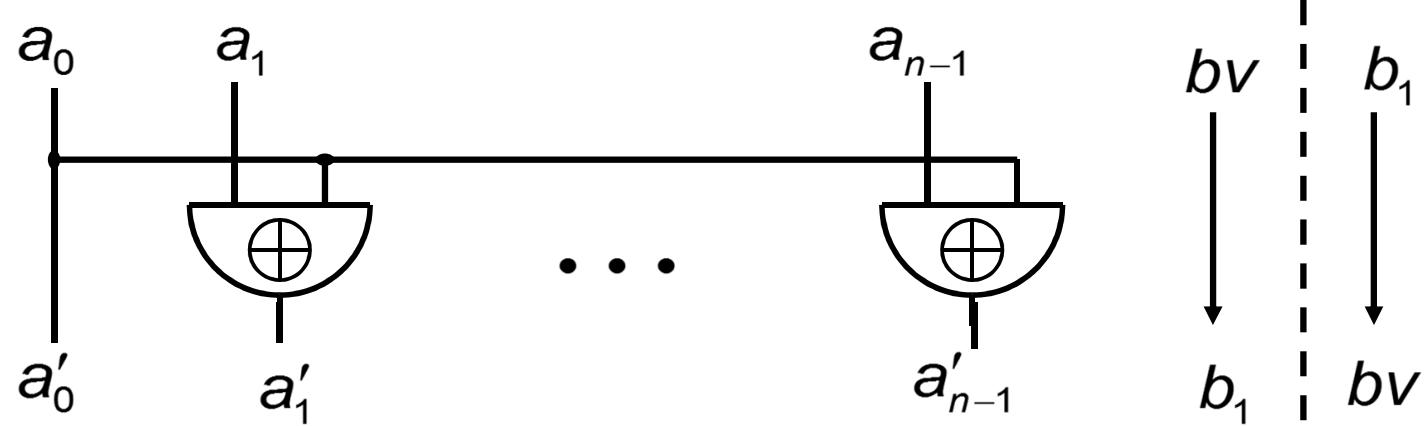
$$(2) \quad b_1(\bar{a}) = -b_1(a)$$

$$(3) \quad b_1(a_0, a_0 \oplus a_1, \dots, a_0 \oplus a_{n-1}) = bv(a)$$

$$bv(a_0, a_0 \oplus a_1, \dots, a_0 \oplus a_{n-1}) = b_1(a)$$

Konvertierer

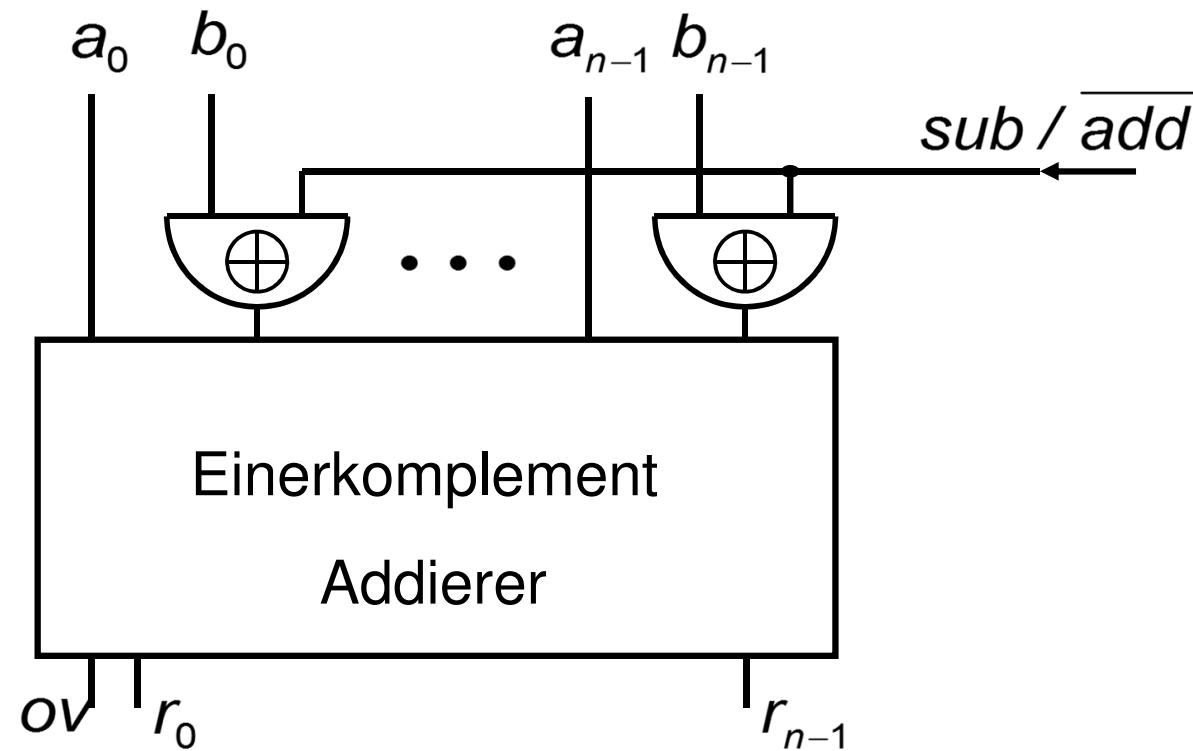
Damit können wir in einfacher Weise zwischen Einerkomplement und Betrag und Vorzeichen konvertieren:



Ebenso zwischen Zweier- und Einerkomplement, wenn man leicht eine 1 dazuaddieren oder subtrahieren kann.

Subtraktion

Die Subtraktion erhalten wir mit der Regel (2) für das Einerkomplement in einfacher Weise, wenn wir einen Addierer für Einerkomplementdarstellung haben: Wir brauchen bei einer Subtraktion nur die Bits des Subtrahenden zu invertieren:



Subtraktion

Mit den Regeln (1) und (2) kann man aber auch für das Zweierkomplement in einfacher Weise subtrahieren, wenn wir einen Addierer haben, der die Addition einer 1 zusätzlich erlaubt (z.B. beim Csu-Adder geschenkt !)

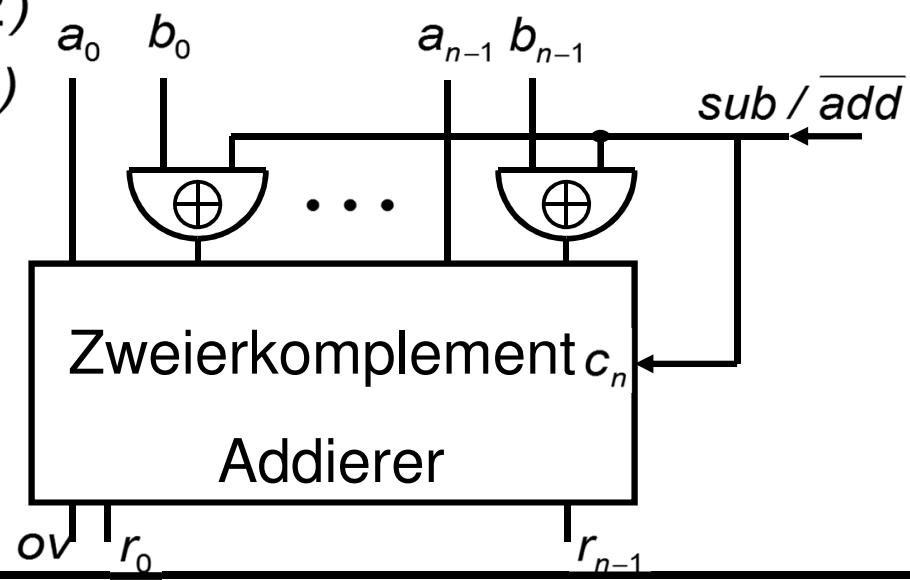
$$-b_2(x) = -(b_1(x) - x_0) \quad (1)$$

$$= -b_1(x) + x_0$$

$$= b_1(\bar{x}) + x_0 \quad (2)$$

$$= b_2(\bar{x}) + \bar{x}_0 + x_0 \quad (1)$$

$$= b_2(\bar{x}) + 1$$



Zurückführung auf unsigned Adder

Zu klären bleibt also nur noch

- wie man die Addition im 1er, 2er Komplement auf den Fall der vorzeichenlosen Zahlen zurückführt, und
- wie man Überläufe erkennt.

Der Zusammenhang zwischen unsigned und Zweierkomplementdarstellung ist

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

Zurückführung auf unsigned Adder

Nehmen wir nun einfach einmal einen unsigned Adder her, der zu zwei Worten x und y ein Wort s und ein Überlaufbit c zurückliefert mit

$$2^n c + u(s) = u(x) + u(y)$$

Dann gilt mit $b_2(a) + a_0 2^n = u(a)$

$$2^n c + b_2(s) + s_0 2^n = b_2(x) + x_0 2^n + b_2(y) + y_0 2^n$$

also

$$b_2(s) = b_2(x) + b_2(y) + (x_0 + y_0 - s_0 - c)2^n$$

Dies ist ja schon fast die Summe, bis auf den „Schmutz“

$(x_0 + y_0 - s_0 - c)2^n$ Das schauen wir uns genauer an!

Diskussion -- Fall 1

Wir nehmen an, dass $x_0 + y_0 = 0$
also **beide Operanden nicht negativ.**

Dann ist

$$u(x) + u(y) \leq 2^{n-1} - 1 + 2^{n-1} - 1 = 2^n - 2 \text{ und damit}$$

$$c = 0$$

Ist nun aber

$$(x_0 + y_0 - s_0 - c)2^n = -s_0 2^n \neq 0$$

Dann ist $u(x) + u(y) \geq 2^{n-1}$

$$\Rightarrow b_2(x) + x_0 2^n + b_2(y) + y_0 2^n \geq 2^{n-1}$$

$$\Rightarrow b_2(x) + b_2(y) \geq 2^{n-1} \Rightarrow \text{Überlauf}$$

Das Ergebnis ist also korrekt, wenn kein Überlauf vorliegt.

Diskussion -- Fall 2

Wir nehmen an, dass $x_0 + y_0 = 1$

d.h. **beide Operanden haben verschiedenes Vorzeichen.**

Dann ist für den einlaufenden Übertrag c_1 im Addierer:

$$2c + s_0 = x_0 + y_0 + c_1 \quad \text{und da } c_1 \in \{0,1\}$$

ist also

$$1 = x_0 + y_0 \leq 2c + s_0 \leq x_0 + y_0 + 1 = 2$$

Dann ist aber schon $c + s_0 = 1$

Also ist

$$\begin{aligned} (x_0 + y_0 - s_0 - c)2^n &= (1 - (s_0 + c))2^n \\ &= 0 \end{aligned}$$

Das Ergebnis ist also immer korrekt.

Diskussion -- Fall 3

Wir nehmen an, dass $x_0 + y_0 = 2$

also **beide Operanden negativ.**

Dann ist

$$u(x) + u(y) \geq 2^{n-1} + 2^{n-1} = 2^n \quad \text{und damit} \quad c = 1$$

Ist nun aber $(x_0 + y_0 - s_0 - c)2^n = (1 - s_0)2^n \neq 0$

$$\text{Dann ist } u(x) + u(y) \leq 2^n + 2^{n-1}(0 + \sum_{i=1}^{n-1} s_i 2^i)$$

$$\leq 2^n + 2^{n-1} - 1$$

$$\Rightarrow b_2(x) + x_0 2^n + b_2(y) + y_0 2^n \leq 2^n + 2^{n-1} - 1$$

$$\Rightarrow b_2(x) + b_2(y) \leq 2^n + 2^{n-1} - 1 - 2^{n+1} = -2^{n-1} - 1$$

\Rightarrow **Überlauf**

Das Ergebnis ist also korrekt, wenn kein Überlauf vorliegt.

Überlaufbedingungen

Wir erhalten also in allen 3 Fällen das korrekte Ergebnis, wenn kein Überlauf auftritt. Damit gilt also

Satz

Seien $x, y, s \in \mathcal{B}^n$ und $c \in \mathcal{B}$ mit $2^n c + u_n(s) = u_n(x) + u_n(y)$
dann gilt

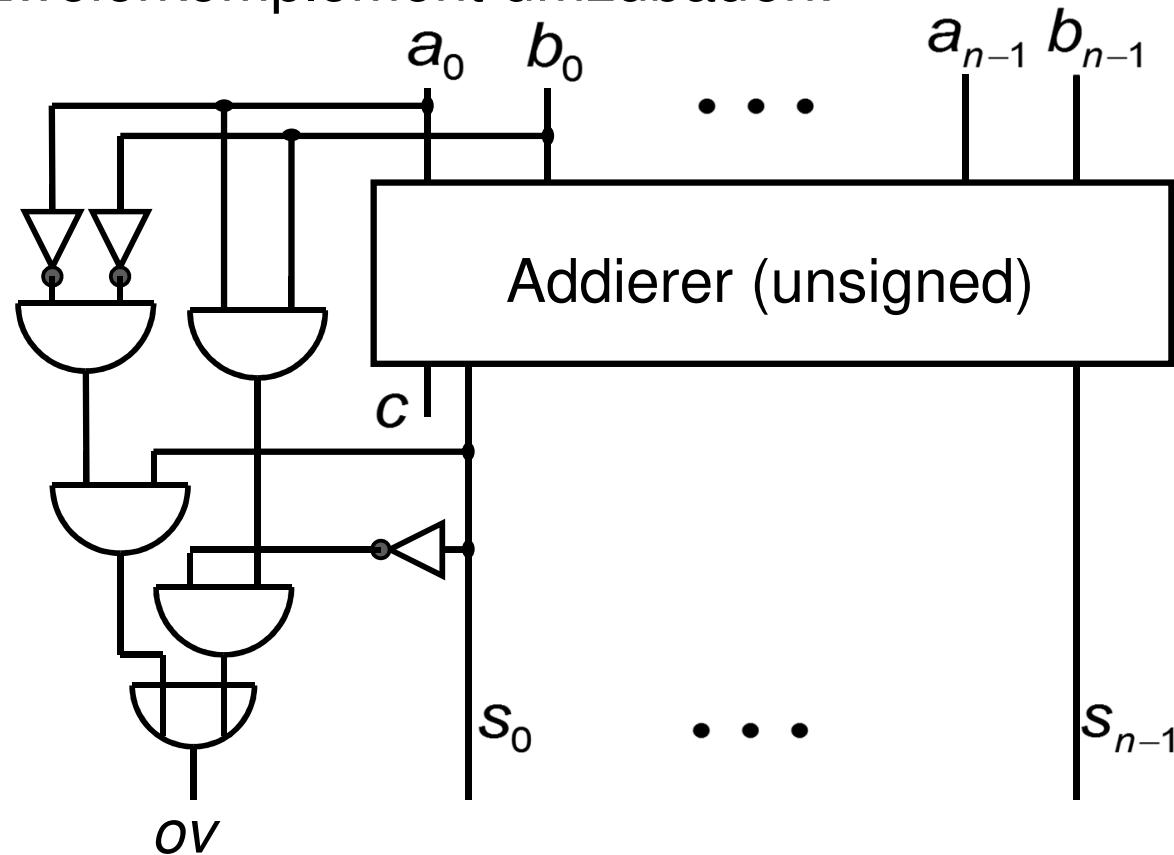
$$b_{2,n}(s) = b_{2,n}(x) + b_{2,n}(y) \text{ und } b_{2,n}(s) \in [-2^{n-1} : 2^{n-1} - 1]$$
$$\Leftrightarrow$$

$$\bar{x}_0 \bar{y}_0 s_0 \vee x_0 y_0 \bar{s}_0 = 0$$

Beweis: durch die Diskussion schon geführt.

Zweierkomplement Addierer

Es ist also sehr leicht, einen unsigned Adder für das Zweierkomplement umzubauen:



Die Multiplikation

Auch hier können wir auf unsere Grundschulkenntnisse zurückgreifen. Es ist

$$\begin{aligned} u(a) \cdot u(b) &= u(a) \cdot 2^{n-1} \sum_{i=0}^{n-1} b_i 2^{-i} \\ &= \sum_{i=0}^{n-1} \underbrace{b_i 2^{n-1-i} u(a)}_{p_i} \end{aligned}$$

Wir müssen also lediglich die n **Partialprodukte**

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

aufaddieren. Diese Produkte sind leicht zu berechnen, da die Multiplikation mit einer Zweierpotenz gerade einem Linksshift entspricht, sofern keine führenden Stellen herausgeschoben werden, d.h.

Shift und Multiplikation mit 2

Ist $a_0 = \dots = a_{j-1} = 0$

Dann gilt
$$\begin{aligned} u(shl(a, j)) &= u(a_j, \dots, a_{n-j-1}, 0, \dots, 0) \\ &= 2^{n-1}(0 + \sum_{i=0}^{n-j-1} a_{i+j} 2^{-i} + 0) \\ &= 2^{n-1}(\sum_{i=0}^{j-1} 0 + \sum_{i=j}^{n-1} a_i 2^j 2^{-i}) \\ &= 2^{n-1}(\sum_{i=0}^{j-1} 2^j 0 + \sum_{i=j}^{n-1} a_i 2^j 2^{-i}) \\ &= 2^j 2^{n-1}(\sum_{i=0}^{j-1} 0 + \sum_{i=j}^{n-1} a_i 2^{-i}) \\ &= 2^j u(a) \end{aligned}$$

Multiplikation

Problem: Wir können beim Berechnen eines Partialproduktes selbst oder aber bei der Addition einen Überlauf erhalten, denn das Produkt von zwei n -bit Zahlen kann so groß werden wie

$$(2^n - 1)^2 = 2^{2n} - 2^{n+1} + 1$$

Es wird also ein $2n$ -bit Wort benötigt, um das Produkt exakt zu berechnen. Da man dies häufig auch möchte, betrachten wir also zunächst folgende

Aufgabe: Realisiere die Funktion

$$\text{prod} : \mathbf{B}^n \times \mathbf{B}^n \mapsto \mathbf{B}^{2n}$$

$$\text{mit } u_{2n}(\text{prod}(a,b)) = u_n(a) \cdot u_n(b)$$

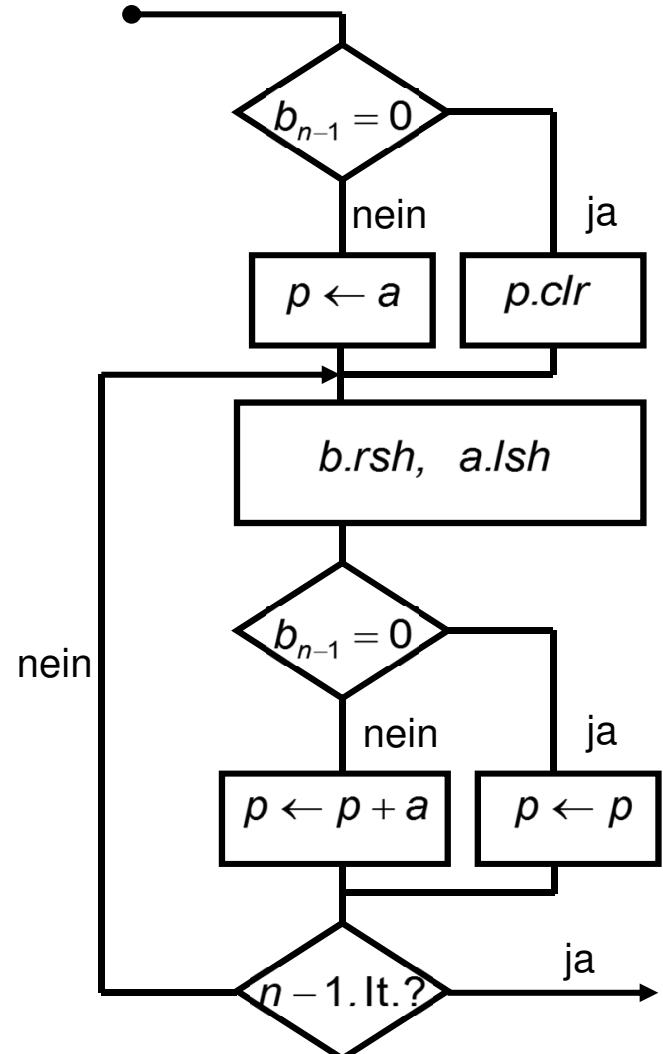
Multiplikation ff

Idee: Der einfachste Ansatz besteht nun darin, dass man einfach die Partialprodukte in $2n$ -bit breiten Registern über einer $2n$ -bit ALU, die die Addition bereitstellt, iterativ aufsummiert. Baut man die Register auch noch als Schieberegister, dann kommt man ohne einen Shifter aus, weil nur jeweils um 1 geschoben werden muß. Solche einfachen Multiplizierwerke nennt man auch **Shift&Add** Multiplizierer.

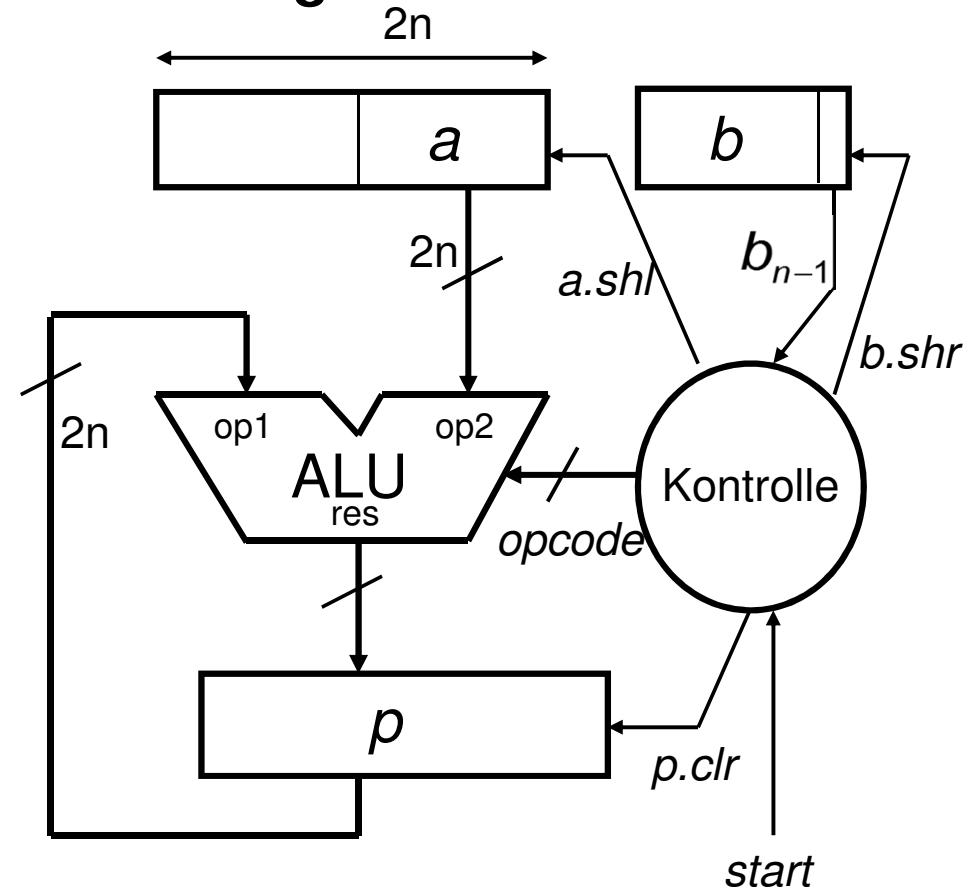
Wir nehmen also an, dass a, b in Schieberegistern der Breite $2n$ stehen, die mit den Steuerleitungen $b.rsh$, $a.lsh$ um jeweils ein Bit nach rechts bzw. links geschoben werden können.

Die Partialprodukte summieren wir in einem $2n$ Register p auf. Dies ergibt folgendes Ablaufschema:

Shift&Add Multiplikation



Umsetzung:



Shift&Add Multiplikation ff

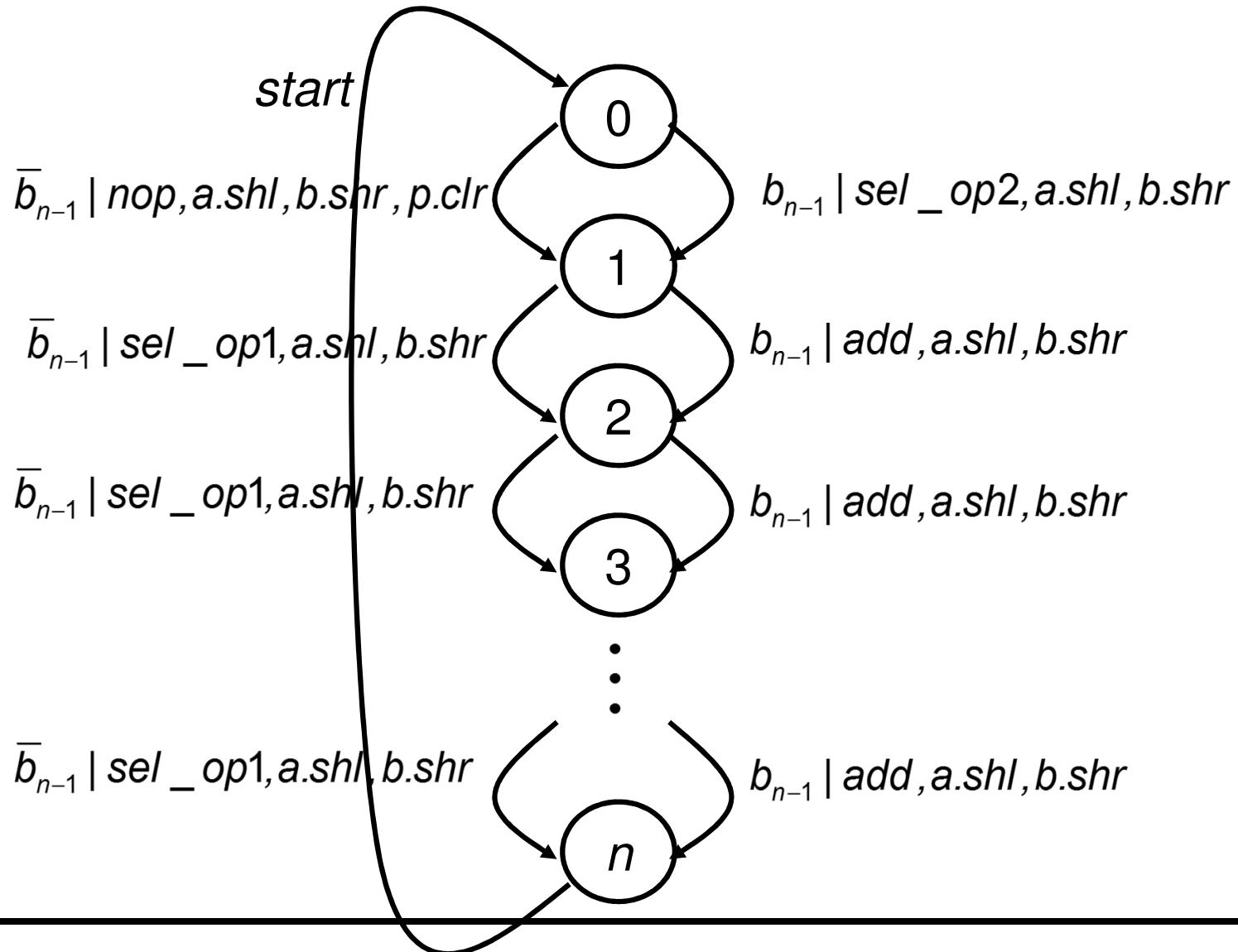
Wir stellen fest, dass wir für b sogar nur ein n -bit Register brauchen. Es bleibt noch der Entwurf der Kontrolle, die den Kontrollfluß aus unserem Diagramm durch eine entsprechende Folge von Steuersignalen erzeugen muss.

Folgende Zeichen sind zu erzeugen:

- die Steuersignale der Register (a.shl, b.shr, p.clr)
- die Opcodes der ALU, und zwar
 - *add* : Addiere (realisiert $p := p + a$)
 - *sel_op1* : $res := op1$ (realisiert $p := p$)
 - *sel_op2* : $res := op2$ (realisiert $p := a$)
 - *nop* : keine Operation

Damit hat die Kontrolle folgendes Aussehen:

Shift&Add: Die Kontrolle



Shift&Add Multiplikation ff

Die Initialisierung entspricht dem Übergang von Zustand 0 nach 1, die Schleife den Übergängen von 1 bis $n-1$.

Man kann dieses Verfahren noch auf viele Weisen verbessern, aber es hat im Grunde den selben Nachteil wie die serielle Addition, es dauert n Taktperioden.

Wie erhält man eine schnellere Schaltung?

Man kann, wie bei der Addition auch, zunächst die sequentielle Schaltung „abrollen“. Dies würde zu einer Schaltung der Tiefe $n*T(\text{Addierer})$ führen (vgl. Ripple Carry Adder).

Es geht aber besser:

Carry Save Addition

Wir sind nicht gezwungen, jedes Zwischenergebnis

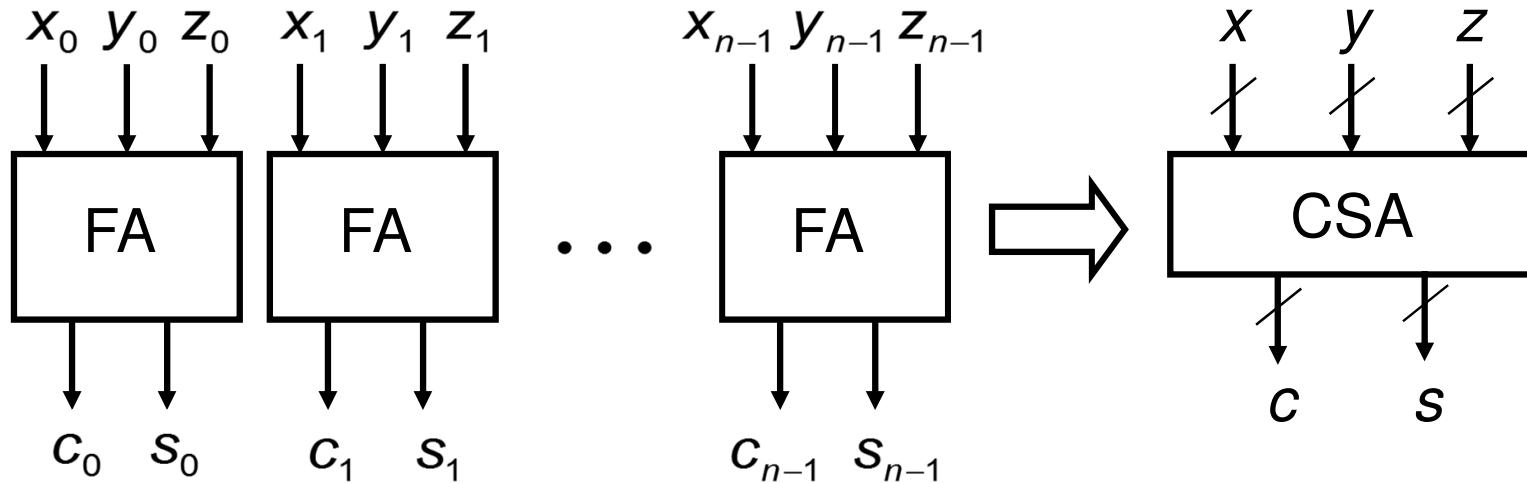
$$\sum_{i=j}^{n-1} \underbrace{b_i 2^{n-1-i} u(a)}_{p_i}$$

beim Akkumulieren der Partialprodukte durch ein einziges Maschinenwort darzustellen!

Beobachtungen:

- ein Volladdierer kann die Summe von 3 Bits durch eine zweistellige Binärzahl darstellen ($2c + s = x + y + z$)
- dieses Prinzip kann man durch folgende Schaltung, einen sog. **Carry Save Adder** auf n -bit Zahlen verallgemeinern:

Carry Save Adder

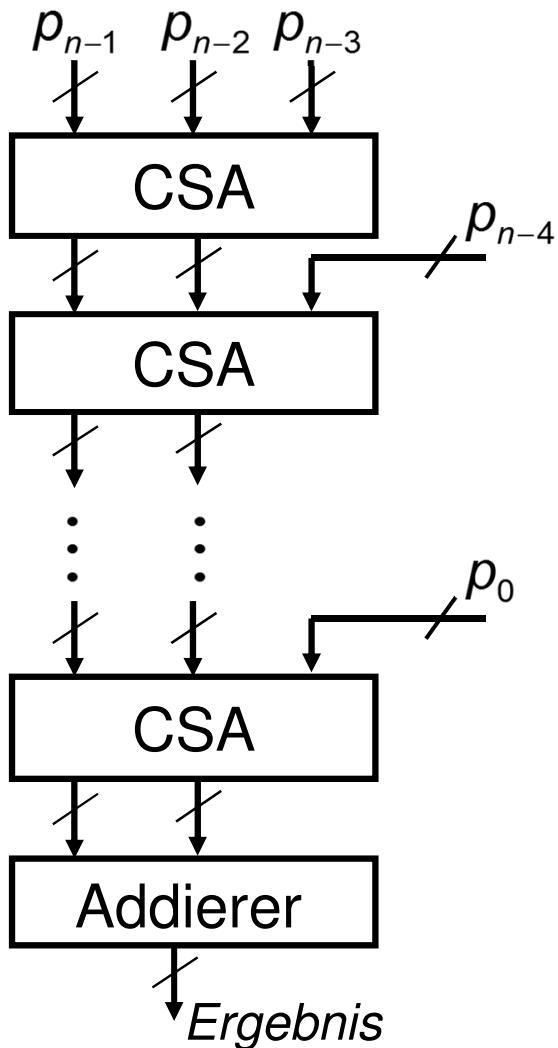


Für diese einfache Nebeneinanderschaltung von Volladdierern gilt:

$$\begin{aligned} u(x) + u(y) + u(z) &= 2^{n-1} \sum_{i=0}^{n-1} (x_i + y_i + z_i) 2^{-i} \\ &= 2^{n-1} \sum_{i=0}^{n-1} (2c_i + s_i) 2^{-i} \quad = 2u(c) + u(s) \end{aligned}$$

Wir nennen sie einen Carry Save Adder der Breite n .

CSA Multiplizierer

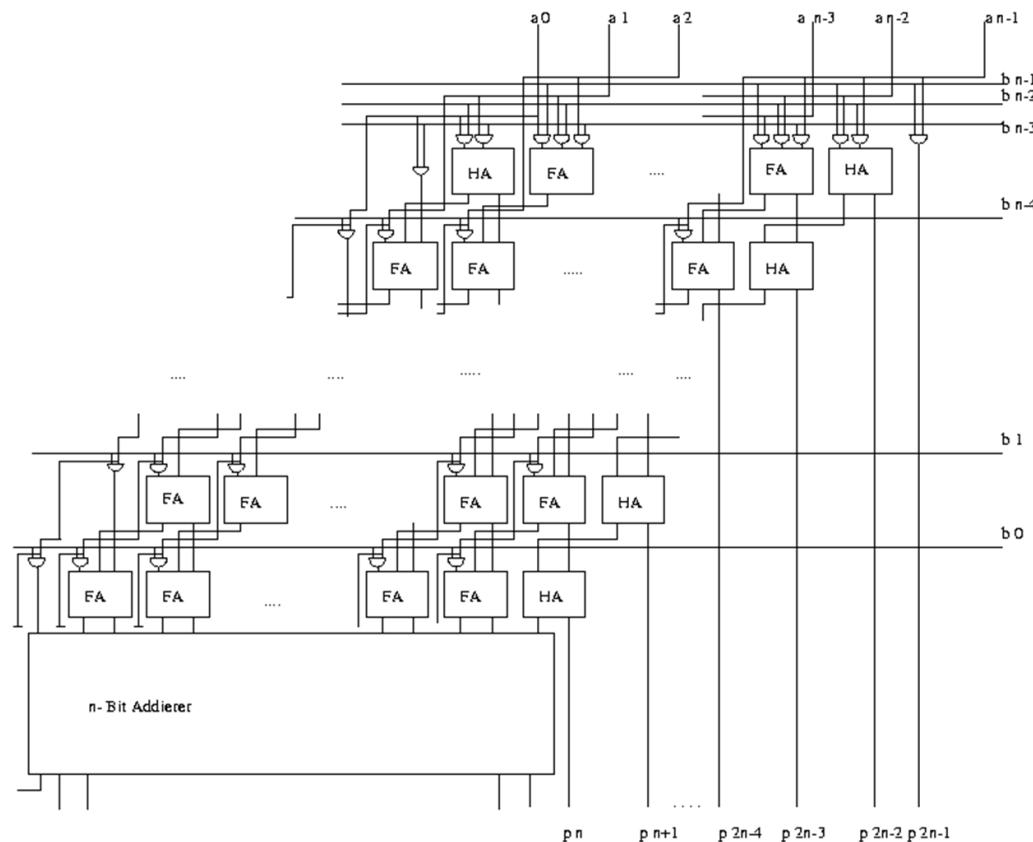


Wir können also leicht in Tiefe $(n-2)T(FA) + T(\text{Addierer})$ alle Partialprodukte aufsummieren, indem wir zunächst nur Carry Save Addierer der Breite $2n$ einsetzen.

Die Schaltung hat Breite $2n$, allerdings benötigt man diese Breite nicht überall. Man kann sie noch weiter optimieren, indem man ausnutzt, dass die Partialprodukte nur Breite n haben, und überflüssige Volladdierer weglässt bzw. durch Halbaddierer ersetzt, wenn man nur 2 Bit addiert.

CSA Multiplizierer ff

Nimmt man die Erzeugung der Partialprodukte ebenfalls noch in die so optimierte Schaltung hinein, erhält man einen einfachen Parallelmultiplizierer nach folgender Skizze:



Wallace tree

Man kann sogar in logarithmischer Tiefe multiplizieren, indem man die Partialprodukte nicht nacheinander in Carry Save Addern aufaddiert, sondern parallel erzeugt und dann in einem „Baum“ von Carry Save Addern auf zwei Zahlen reduziert. Ein solches Schema heißt **Wallace tree**:

1. Stufe: reduziere alle p_0, p_1, \dots, p_{n-1} zu $z_0^{(1)}, z_1^{(1)}, \dots, z_{2\lfloor\frac{n}{3}\rfloor+(n-3)\lfloor\frac{n}{3}\rfloor-1}^{(1)}$

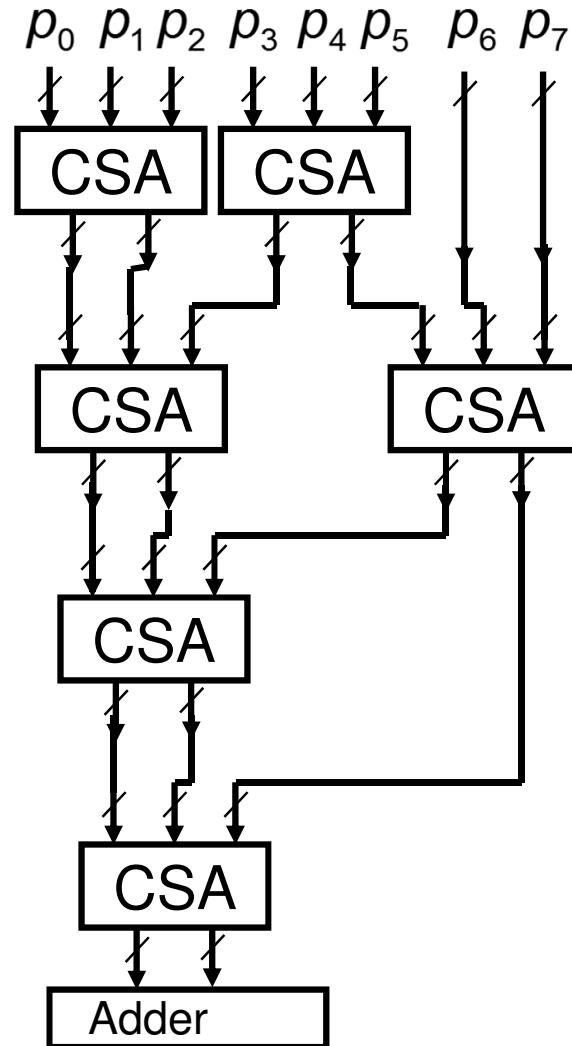
2. Stufe: reduziere alle $z_i^{(1)}$ zu $z_0^{(2)}, z_1^{(2)}, \dots, z_{2\left\lfloor\frac{2\lfloor\frac{n}{3}\rfloor+(n-3)\lfloor\frac{n}{3}\rfloor-1}{3}\right\rfloor+...}^{(2)}$

... Auf jeder Stufe sinkt die Zahl der Zwischenergebnisse um den Faktor $\sim 3/2$, d.h. nach etwa

$$\sim \log_{\frac{3}{2}} n$$

Stufen brauchen wir nur noch einen schnellen Addierer.

Beispiel: 8 - Bit Wallace tree



Ausblick:

Es gibt noch eine ganze Fülle von Techniken zur schnellen Multiplikation, deren Behandlung den Rahmen dieser einführenden Veranstaltung sprengt. Schaltkreise dazu sind recht groß und nicht immer leicht systematisch zu konstruieren. Daher nimmt man die Multiplikation in der Regel aus der ALU heraus und realisiert sie in einer extra Funktionseinheit.

Ähnliches gilt für die Division: Auch auf die Division lässt sich die Schulmethode übertragen (sukzessive Subtraktion eines (binären) Vielfachen des Divisors). Allerdings werden hier die Verfahren recht trickreich, will man schnell dividieren.

(trickreich → fehleranfällig ?!? vgl. Pentium Bug)

3.5 Realisierung eines einfachen Prozessors

Zur Vorlesung Rechenanlagen

SS 2019



Wir wollen in diesem Abschnitt alles, was wir bisher gelernt haben, zusammenführen, und unsere eingangs betrachtete, abstrakte Maschine nun realisieren.

Dazu lassen wir zunächst nochmal alle Befehle Revue passieren, um Änderungen oder Ergänzungen an unserem anfänglichen Konzept anzusprechen.

Ferner werden wir uns das Leben nicht zu schwer machen, und uns zunächst auf ein Minimum an notwendigen Befehlen beschränken.

Die Arithmetik

Wir haben Arithmetik auf ganz verschiedenen Objekten. Der Einfachheit halber wollen wir Gleitkommabefehle ganz ausschliessen. Allerdings sollen sowohl unsigned als auch signed Operationen auf Worten möglich sein. Alle Operanden beziehen sich dabei auf GP-Register der Registerbank. Befehle für Teilworte klammern wir aus. Für

$$\text{OP} \in \{ \text{ADD}, \text{SUB}, \text{MUL}, \text{DIV}, \text{SXX} \}$$

erhalten wir bei einer Zahlendarstellung $d \in \{ \text{bv}, b_1, b_2 \}$, für die man sich entscheiden muss, folgende Befehle:

$$\text{OP } R_i, R_j, R_k; \quad \text{OPU } R_i, R_j, R_k;$$

$$\text{OPI } R_i, R_j, \#k; \quad \text{OPUI } R_i, R_j, \#k;$$

Kein Suffix besagt, dass es sich um eine Operation auf einem Maschinenwort unter Darstellung d handelt. Das Suffix U steht für unsigned, I für immediate.

Arithmetik ff

Beispiel: **ADDUI Ri, Rj, #k** bezeichnet demnach eine Unsigned Addition mit (unsigned!) immediate Operand k .

Die Wirkung sei stets die folgende:

$$\text{reg}[i] = u_n^{-1}((u_n(\text{reg}[j]) \text{ op } u_n(\text{reg}[k])) \bmod 2^n)$$

bzw. $\text{reg}[i] = u_n^{-1}((u_n(\text{reg}[j]) \text{ op } k) \bmod 2^n)$

für **OPU Ri, Rj, Rk**; bzw. **OPUI Ri, Rj, #k**; Befehle,
wobei zusätzlich ein möglicher Überlauf angezeigt wird.

**Wir lassen bei unserem einfachen Beispielprozessor
Befehle, die sich auf Unsigned Zahlen beziehen, der
Einfachheit halber weg.**

Arithmetik ff

Für $OP\ Ri, Rj, Rk$; bzw. $OP1\ Ri, Rj, \#k$; Befehle sei die Wirkung folgende:

$$reg[i] = d_n^{-1}(d_n(reg[j]) op d_n(reg[k]))$$

$$\text{bzw. } reg[i] = d_n^{-1}(d_n(reg[j]) op k)$$

wobei bei Überlauf, dessen Anzeige ebenfalls erfolgt, der Inhalt vom Zielregister nicht näher festgelegt ist.

In unserem einfachen Beispielprozessor lassen wir Befehle für Multiplikation und Division der Einfachheit wegen weg, und gehen davon aus, dass sie durch Unterprogramme realisiert sind.

Verzweigungen

Verzweigungen haben von nun an die Form

BEQZ Ri, #j; **BNEZ Ri, #j;**

und als neue Befehle

BOV #j; **BNOV #j;**

wobei **BOV**, **BNOV** Verzweigungen nach dem Overflow Flag sind. Damit können wir einen Überlauf zumindest explizit abfangen. Wir werden später sehen, dass es dazu bessere Techniken gibt (vgl. Ausnahmebehandlungen).

Transporte

Als Transportbefehle hatten wir

LOAD Ri, Rj, #k; STORE Ri, Rj, #k;

i, j sind Registeradressen, und k ein immediate Operand. Die Wirkung ist:

LOAD Ri, Rj, #k: $reg[i] := mem[u_n(reg[j]) + k]$

STORE Ri, Rj, #k: $mem[u_n(reg[j]) + k] := reg[i]$

Transporte von Halbworten oder Bytes klammern wir aus. Wir benutzen einen wortorientierten Speicher!

Sprünge

Wir hatten **JMP #k;** **JREG Ri;** und **JAL Ri, #k;**
Diese Befehle werden wir unverändert übernehmen.

Nun können wir beginnen, die Befehle zu kodieren:

Kodierung der Befehle

Ziele:

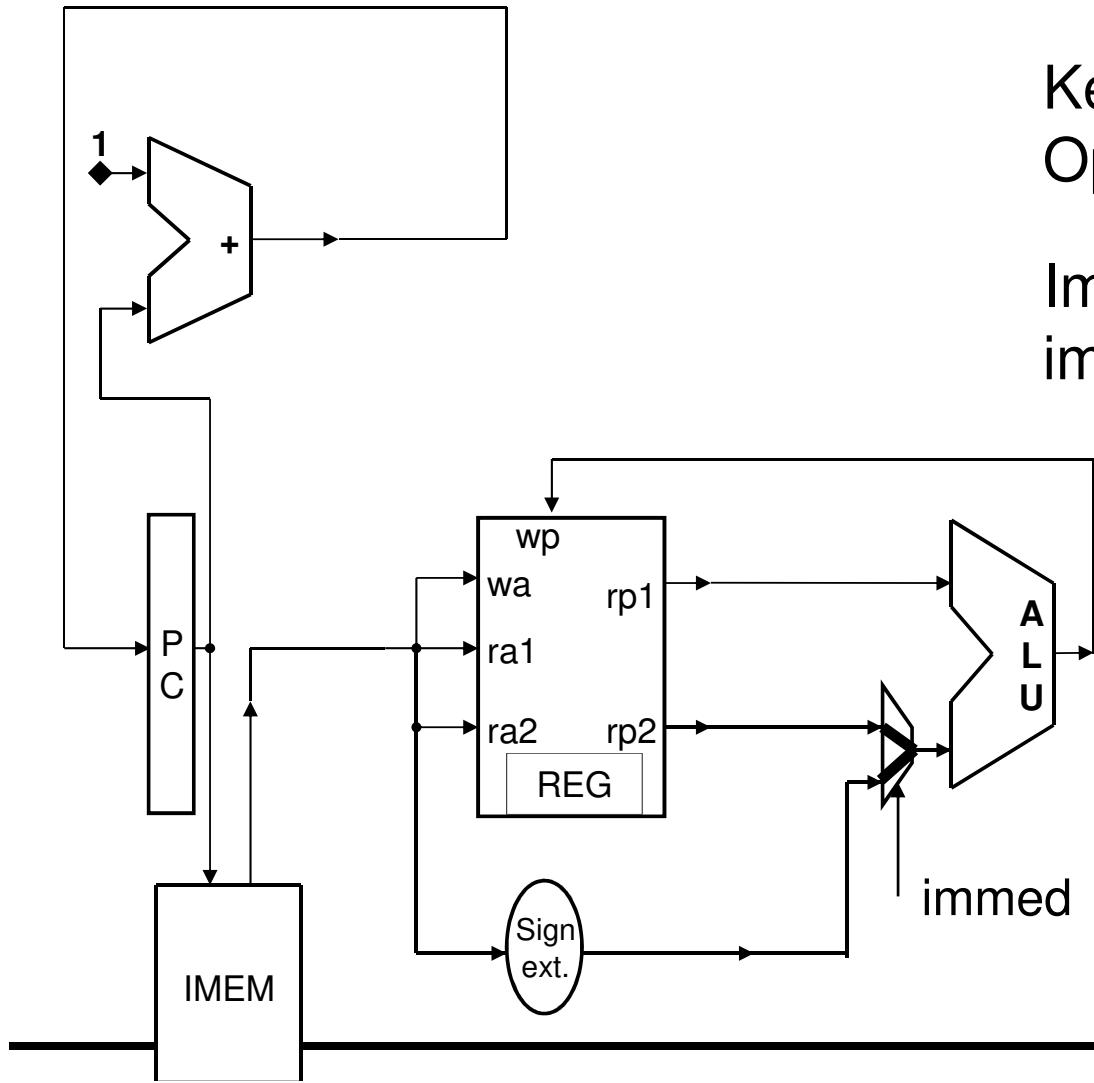
- Kodiere Instruktionen in (ggf. Vielfachen von) Maschinenworten
- Gestalte die Kodierung leicht dekodierbar.

Der Datenfluss im Prozessor hängt sehr stark davon ab, ob es sich um

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

handelt, ist aber bei Befehlen einer dieser Gruppen immer sehr ähnlich:

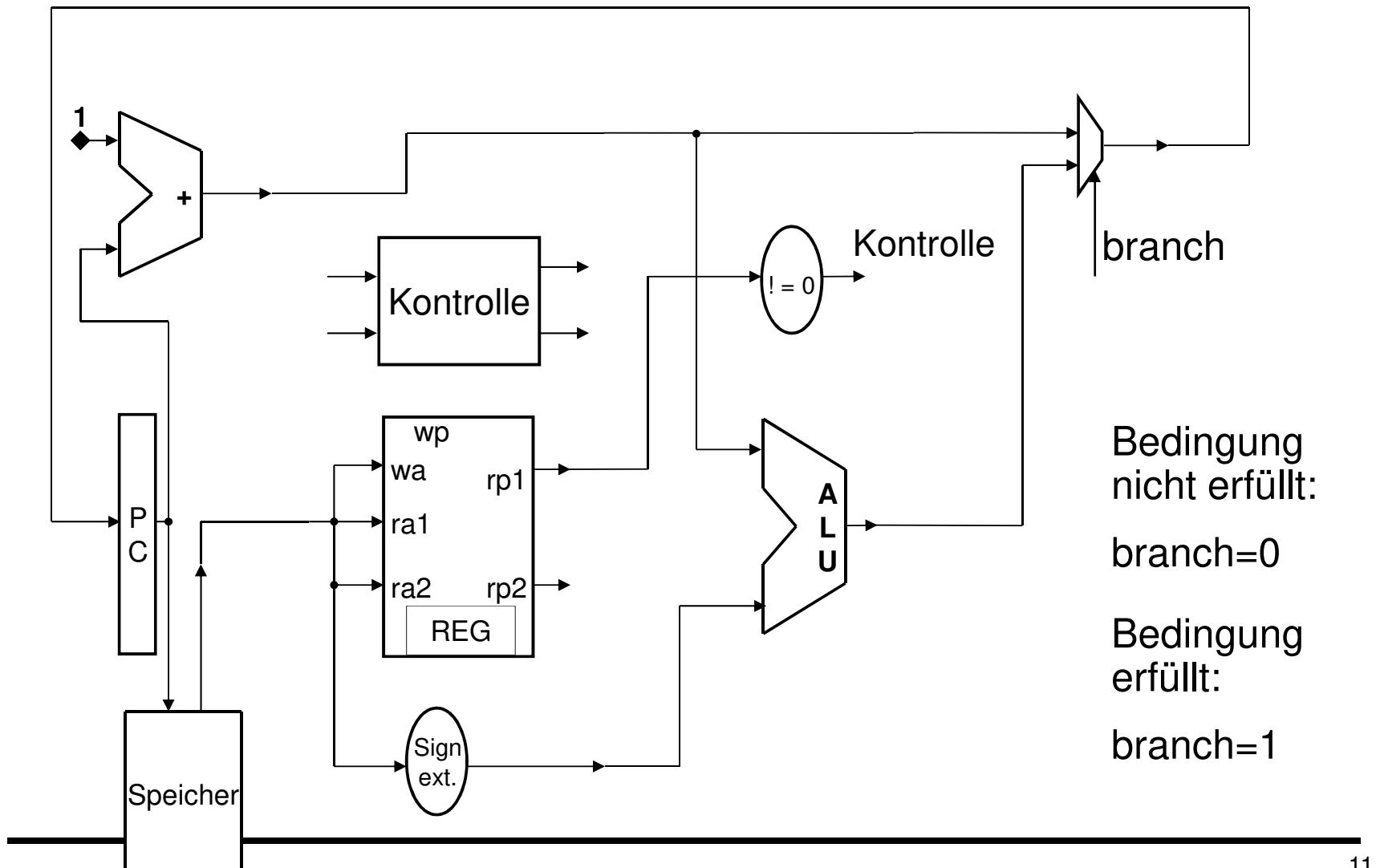
Beispiel: Datenfluss Arithmetik



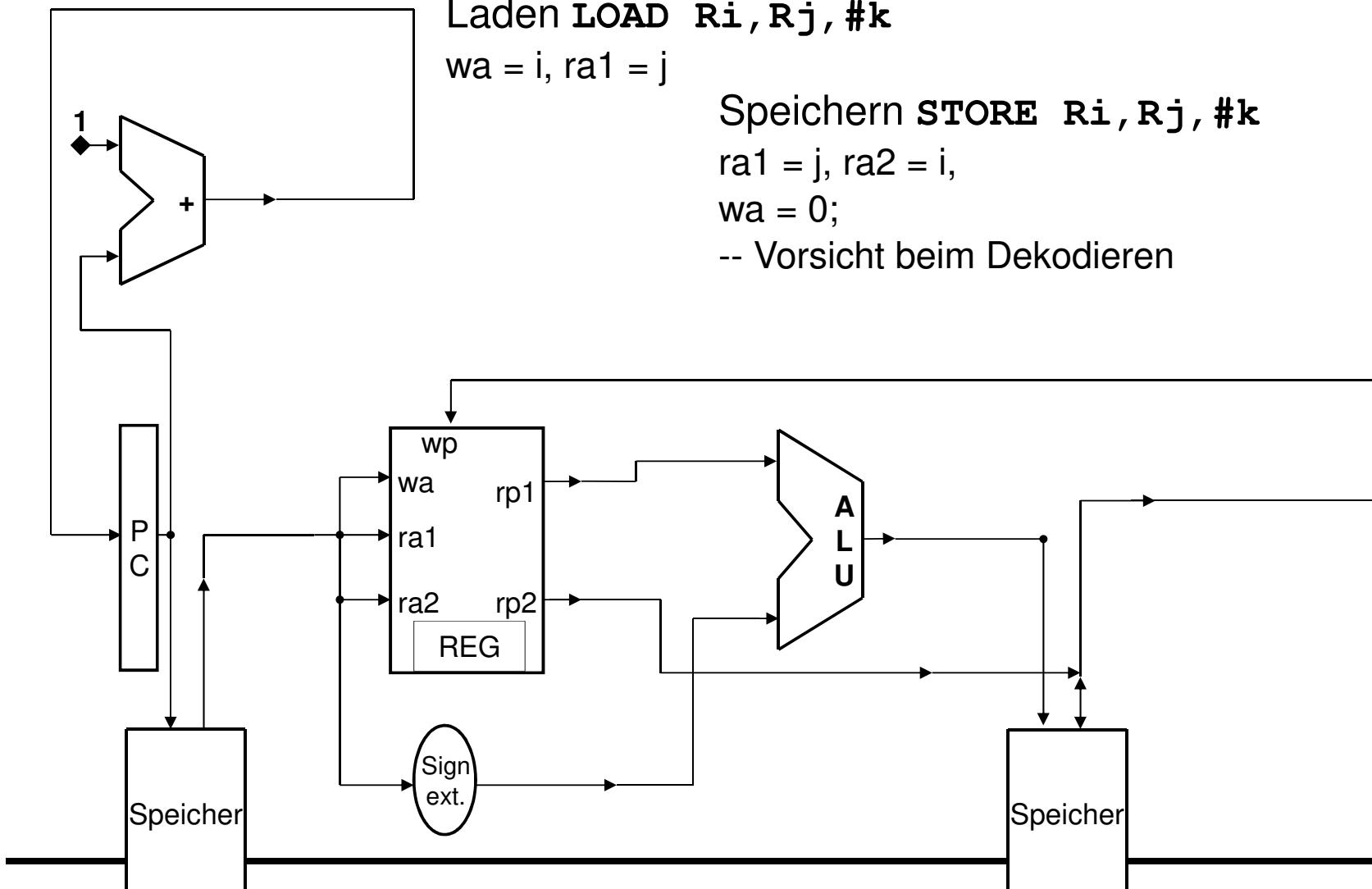
Kein Immediate
Operand: immed = 0

Immediate Operand:
immed = 1

Beispiel: Datenfluss Verzweigung



Beispiel: Datenfluss Transporte

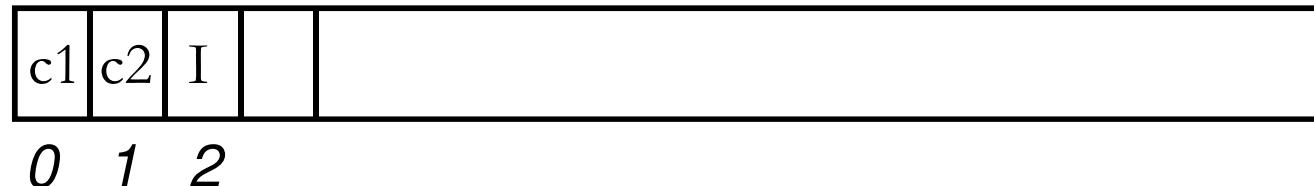


Instruktionsklassen

Die Idee besteht nun darin, alle diese Datenflüsse „übereinander“ zu legen und von der Kontrolle, die den Befehl dekodiert, Kontrollsignale zu berechnen, die durch Multiplexer den korrekten Datenfluss für den jeweiligen Befehl steuern.

Da sich die Datenflüsse ganz grundsätzlich unterscheiden, ist es sinnvoll in einem Feld des Befehlswortes zunächst Klassen mit sehr ähnlichem Datenfluss zu definieren:

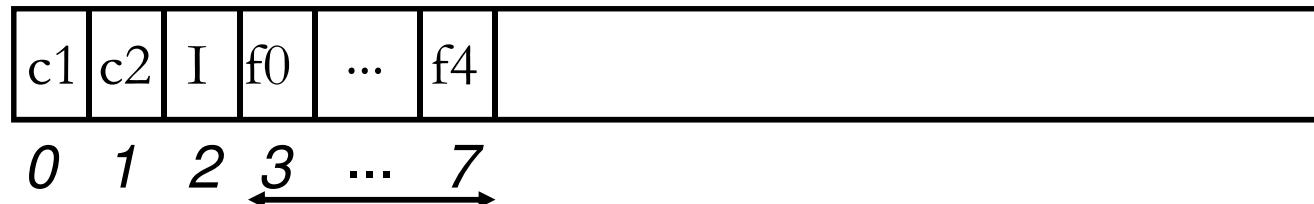
Instruktionsklassen:



- (c1c2): (00) - Arithmetik; (01) - Verzweigung; I: Immediate Bit
(10) - Transport; (11) - Sprung;

Function Codes

Da es nun immer noch mehrere Befehle in einer Klasse gibt, sollte man als nächstes die genaue Funktion des Befehls im Maschinenwort kodieren. Je nach Klasse kann dieses Feld unterschiedliche Länge haben - wir haben etwa sehr viel mehr Arithmetikbefehle als Sprünge. Bei unserer Maschine sollten dazu 5 Bit ausreichen.



Beispiele: Wir wollen die Kodes hier nicht einzeln angeben. Folgende Beispiele wären denkbar:

ADDI 00 1 00000

BEQZ 01 1 00010

ADD 00 0 00000

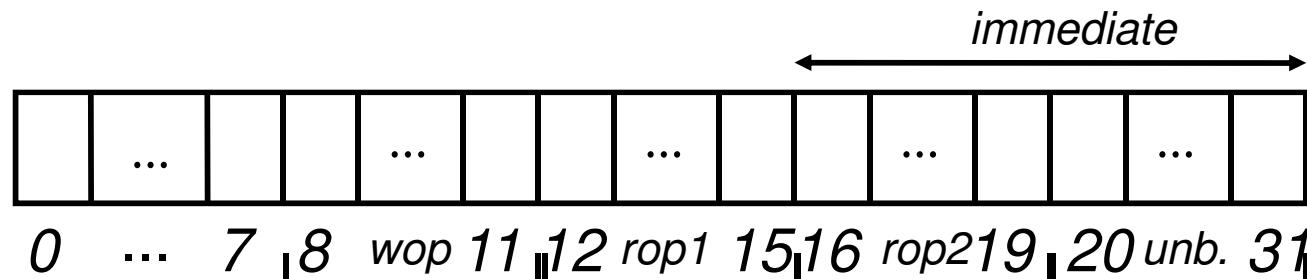
STORE 10 1 00001

Registeradressen

Wir haben nun ein Byte verheizt, könnten also bei 3 Adressbefehlen noch jeweils ein Byte für Registeradressen vorsehen, d.h. unsere Maschine hätte das Potential für bis zu 256 Register.

Problem: Bei 2 Registern und einem Immediate Operand sieht es aber schlecht aus, weil für diesen Operanden dann nur noch ein Byte bleibt, oder wir auf 64-Bit Instruktionscodes übergehen müssen.

Spendieren wir der Maschine nur bis zu 16 Register, können wir 16-Bit Immediate Operanden verkraften, was in den meisten Fällen bei Integer Arithmetik ausreicht.



Verzweigungsbefehle:

Problem: Jetzt wirds eng!

Wir haben 1 Leseoperanden und einen immediate Operanden. Selbst wenn wir nur 16 Register benutzen, bleiben nur 16-Bit für das Sprungziel, wenn der Leseoperand stets auf Bit 12...15 liegt.

Auswege:

- dekodierte den Schreiboperanden als Leseoperanden (mehr Aufwand)
- benutze längere Instruktionen (z.B. 64-Bit)
- benutze relative Sprünge in Verzweigungen

Verzweigungsbefehle ff:

Wir gehen letzteren Weg, d.h. wir definieren die Wirkung von Verzweigungen wie folgt

$$\text{BXX } \mathbf{R}i, \#k : pc := \begin{cases} pc + k + 1 & \text{falls } reg(i) \neq 0 \\ pc + 1 & \text{sonst} \end{cases}$$

Nun können wir in einem Programm immerhin in einer Umgebung von ca. $[-32000:32000]$ beliebig verzweigen. Das ist in den zeitfressenden Fällen ausreichend (Rücksprünge in inneren Programmschleifen).

Weite Distanzen kann man dann nur noch über Sprünge erreichen, d.h. man ersetzt für zu große (kleine) k :

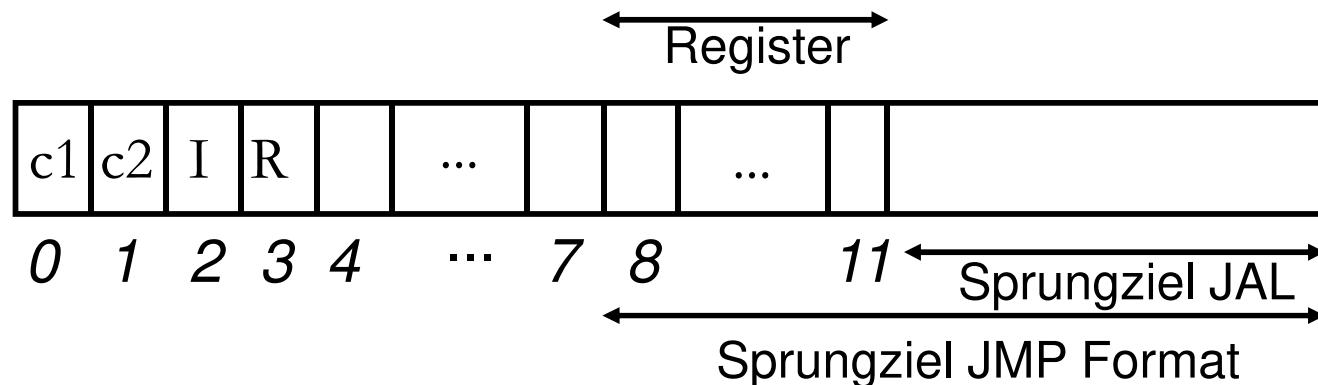
r: **BXX Ri, #k**; durch die Sequenz (YY = not XX)

r: **BYY Ri, #1; JMP #k;** -- falls $k \geq 0$

r: **BYY Ri, #1; JMP #k-1;** -- falls $k < 0$

Sprünge

Hier sind sehr große immediate Operanden wichtig, um auch sehr lange Programme bearbeiten zu können. Deshalb kodieren wir diese Befehle etwas anders:



(I,R): (1,0) - **JMP**; (1,1) - **JAL**; (0,1) - **JREG**

Damit schaffen wir bis 24 Bit Sprungziele, können also in Programmcode von ca. 64 MB Länge ($4 * 16$ MB) springen.

Es ist auch hier geschickt, mit relativen, statt absoluten Sprüngen zu arbeiten: Programme werden verschiebbar!

Eine simple 1 Zyklus Maschine

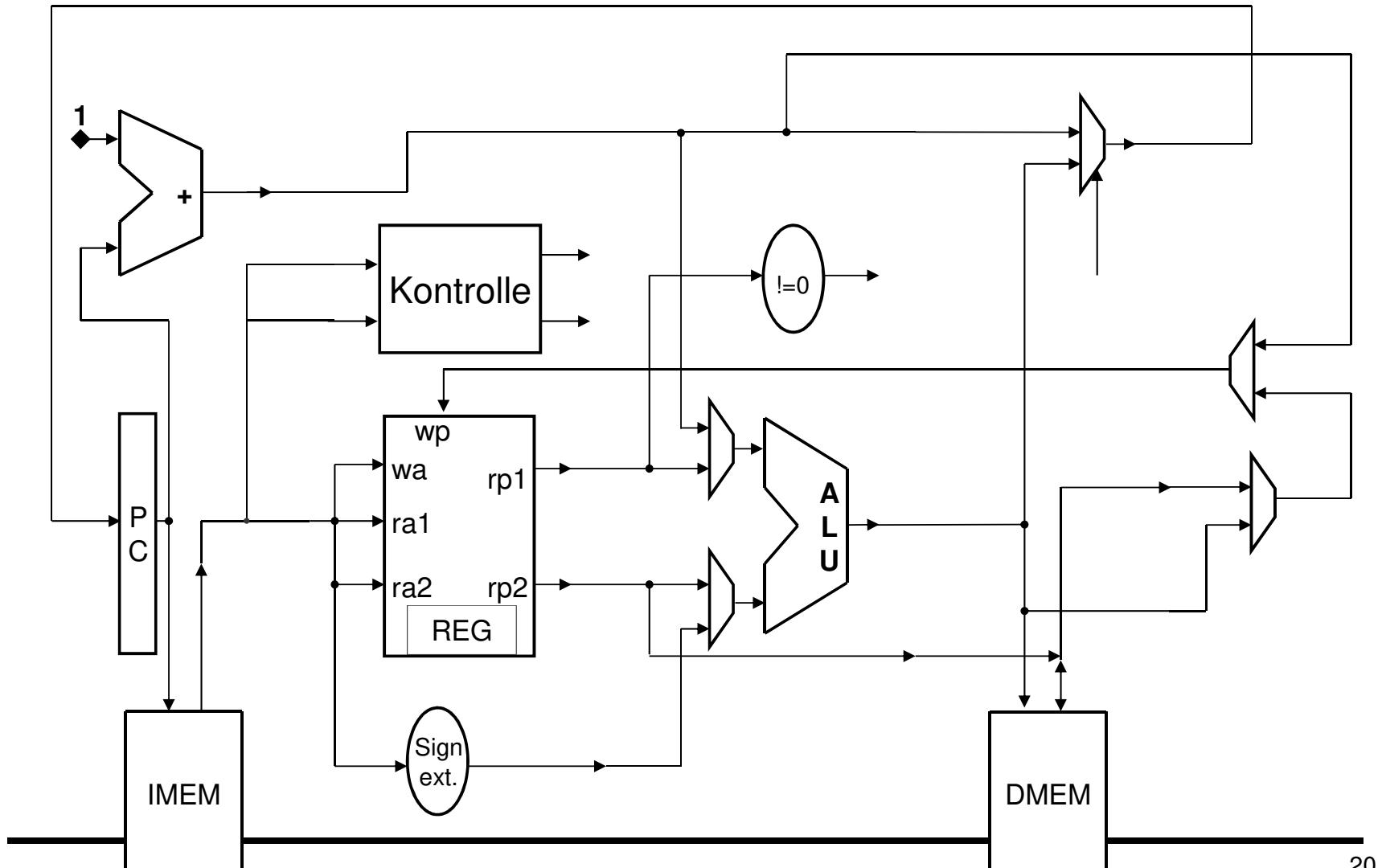
Ideen:

Lege alle Datenflüsse, wie sie bei den einzelnen Befehlstypen vorkommen, „übereinander“ und wähle in Abhängigkeit vom Befehlstyp den entsprechenden Fluss aus. Die **Kontrolle** erzeuge dazu aus dem Befehlscode

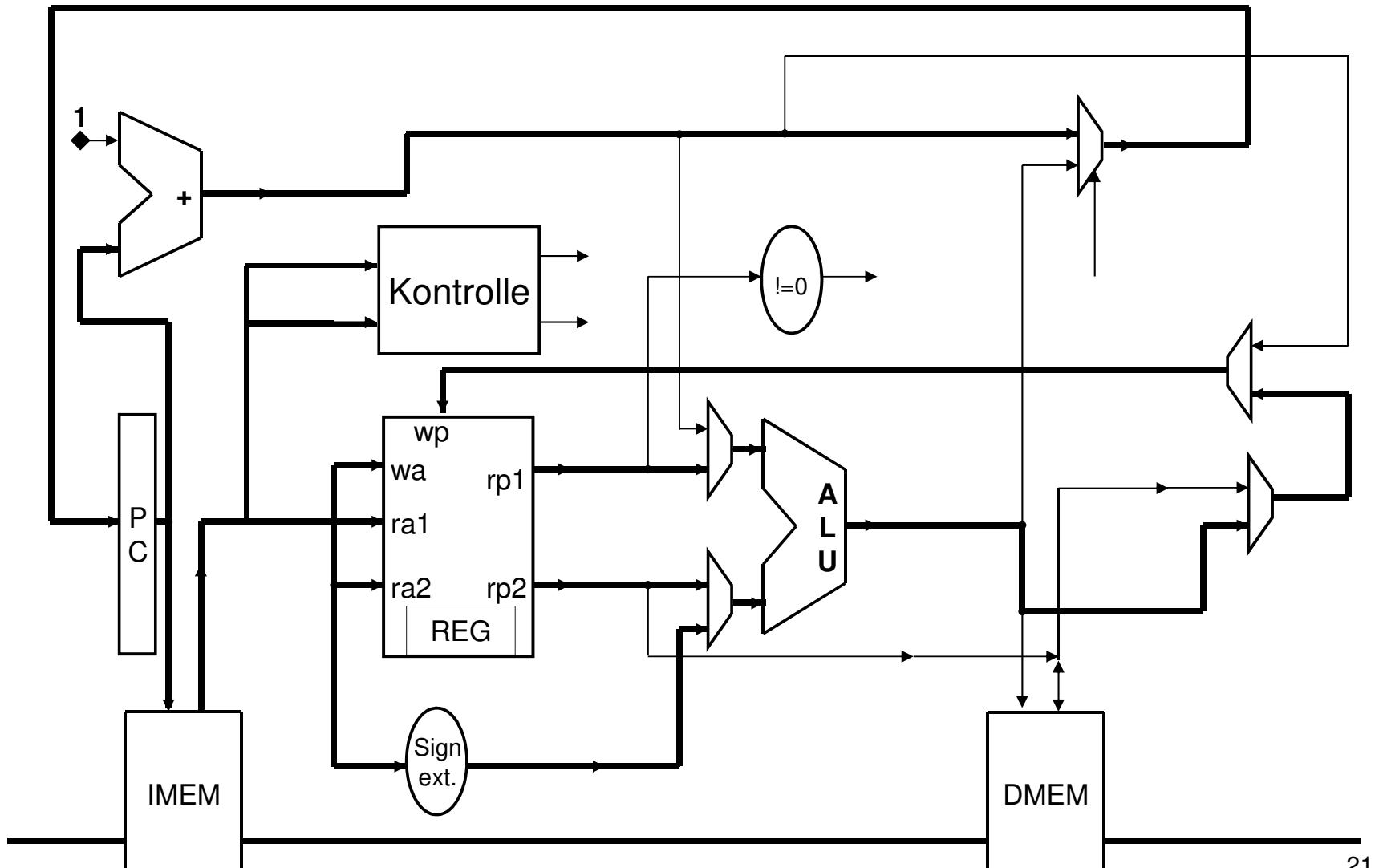
- Steuerleitungen für Multiplexer zur Kontrolle des Datenflusses
- Operationcodes, Adressen und Kontrollsingale für die beteiligten Speicher und ALUs.

Da die Maschine in jedem Zyklus des Taktes einen Befehl bearbeiten soll, benötigen wir zwei RAM Speicher, einen für Programmcode und einen für Daten. **Harvard Architektur**

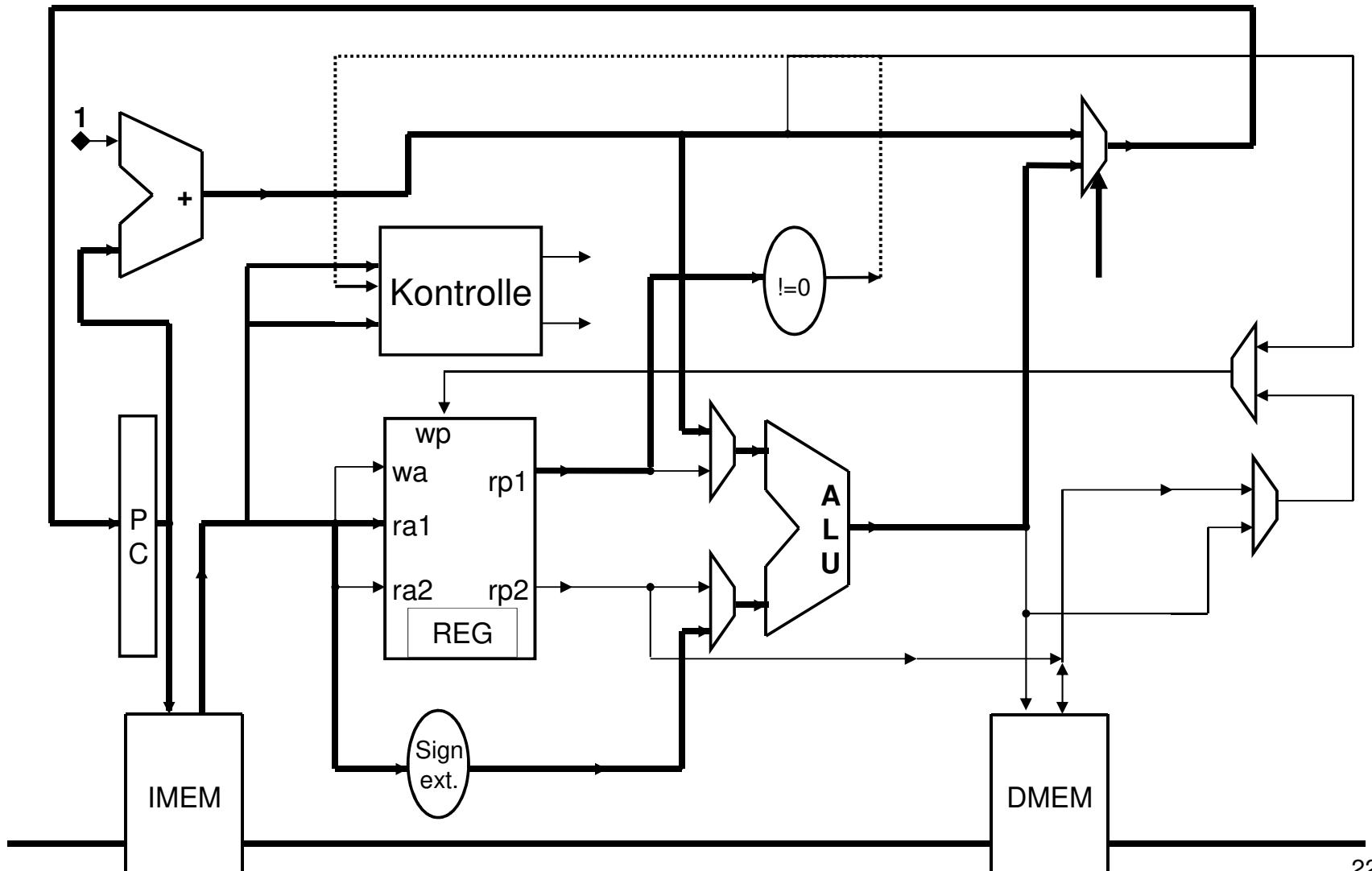
1- Zyklus WüRC



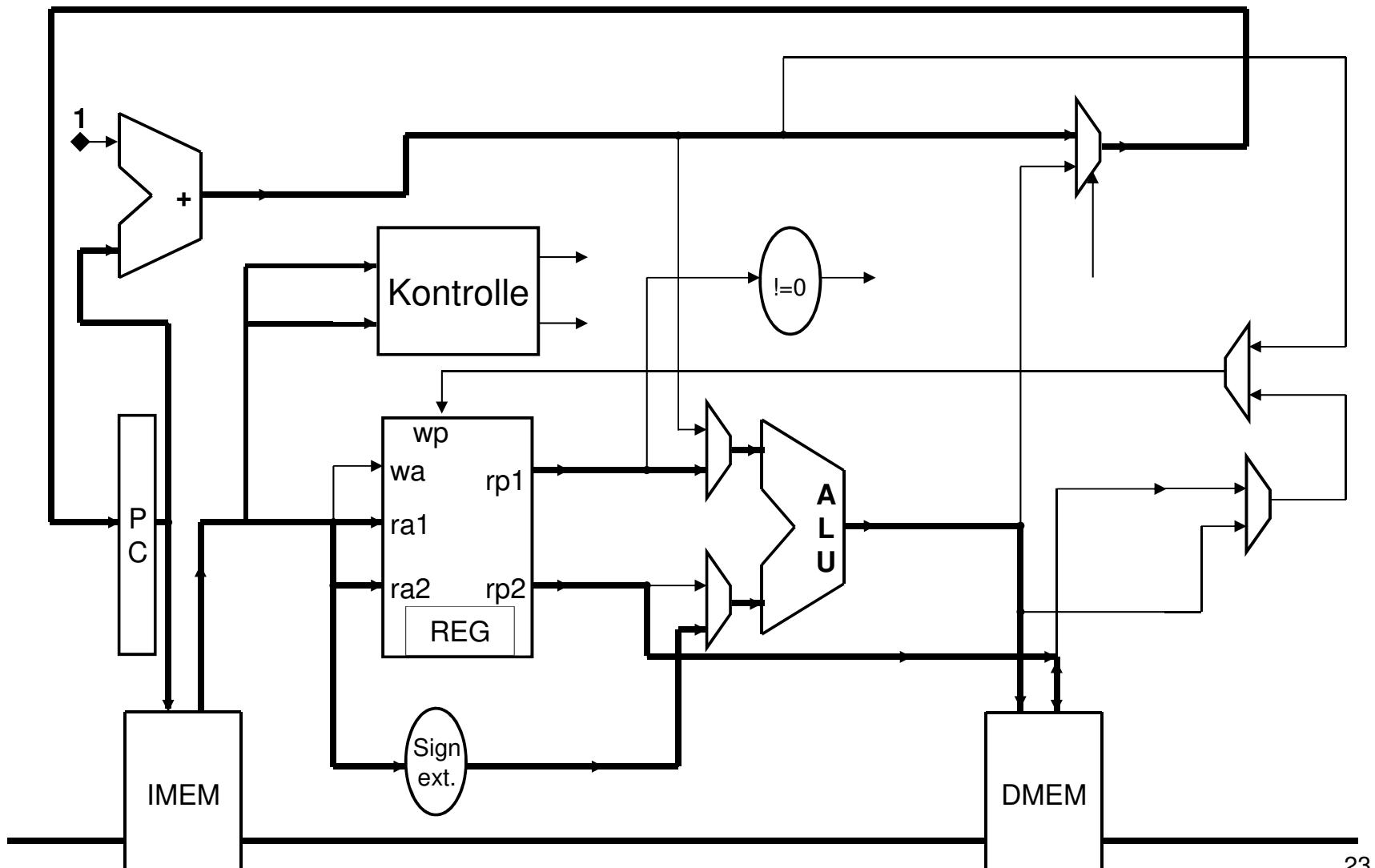
Bearbeitungsbeispiel: Arithmetik



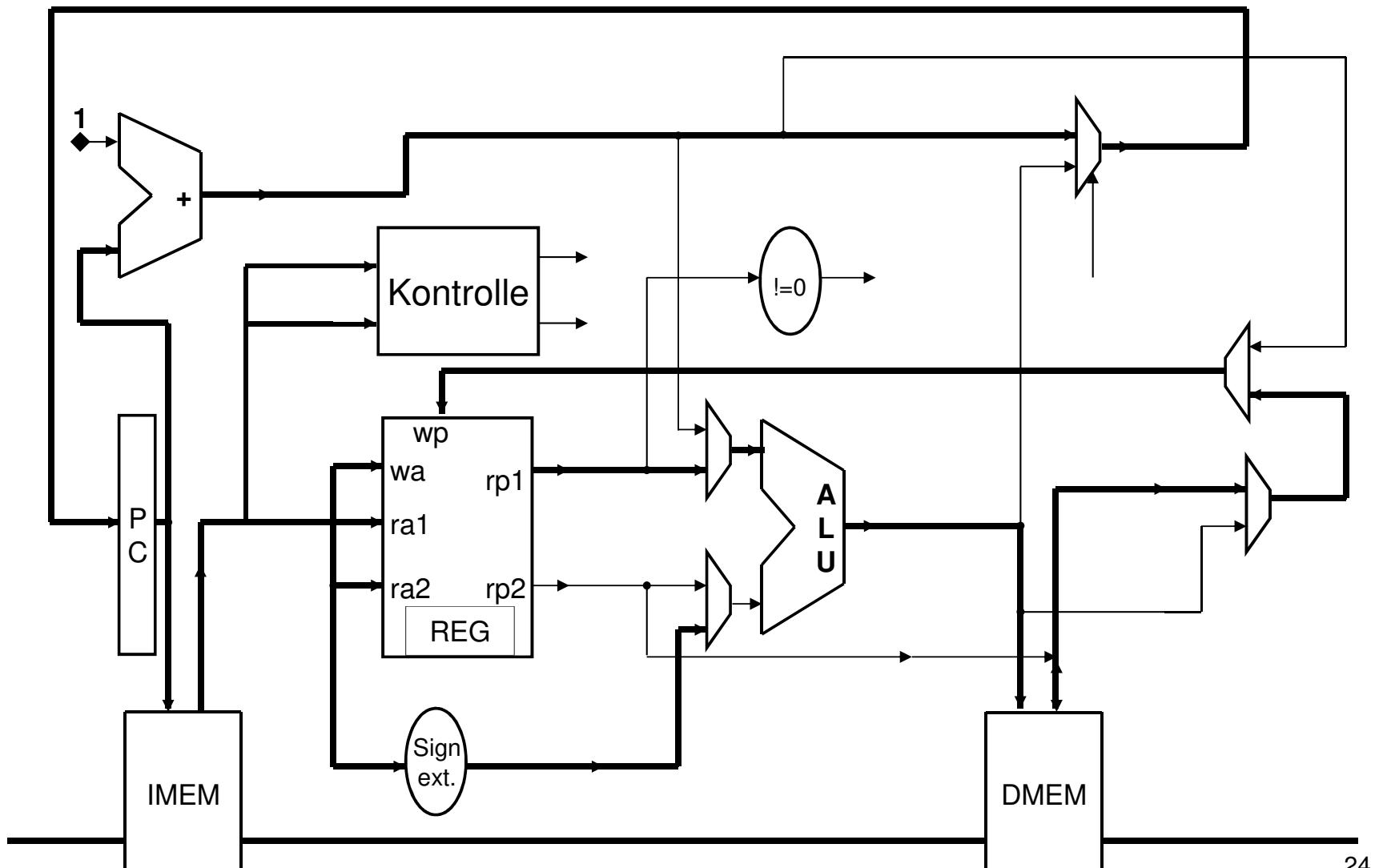
Bearbeitungsbeispiel: Verzweigung



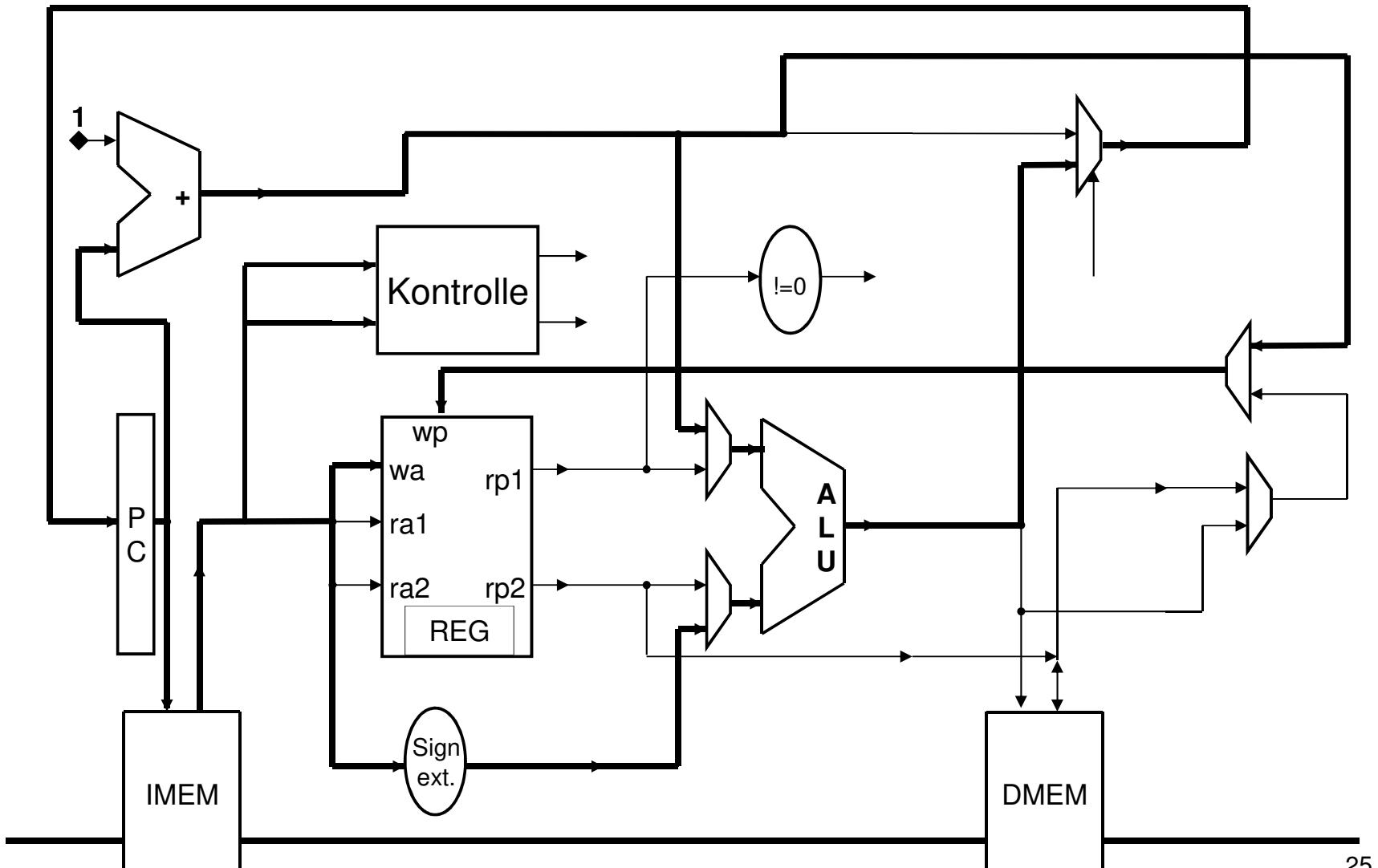
Bearbeitungsbeispiel: STORE



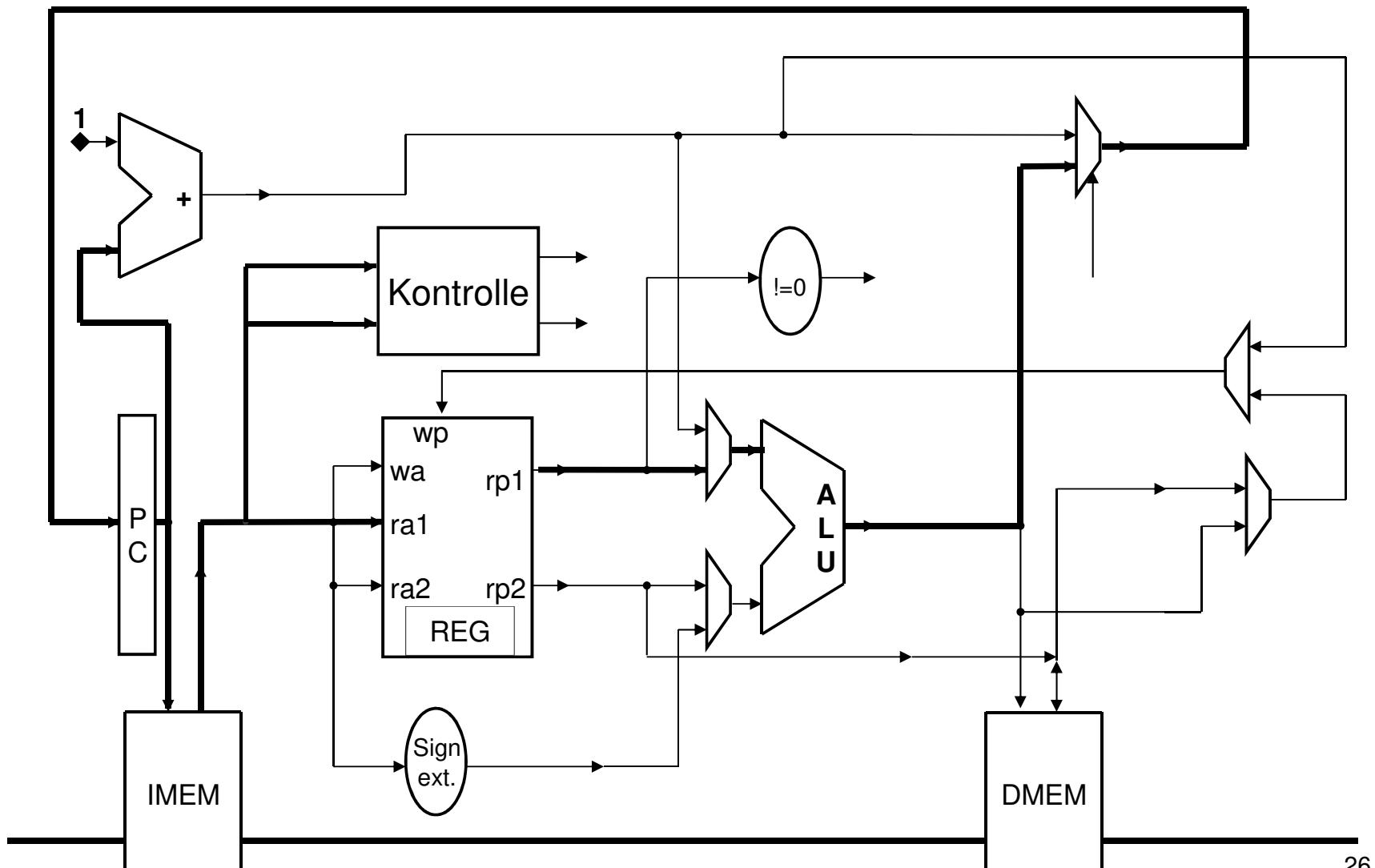
Bearbeitungsbeispiel: LOAD



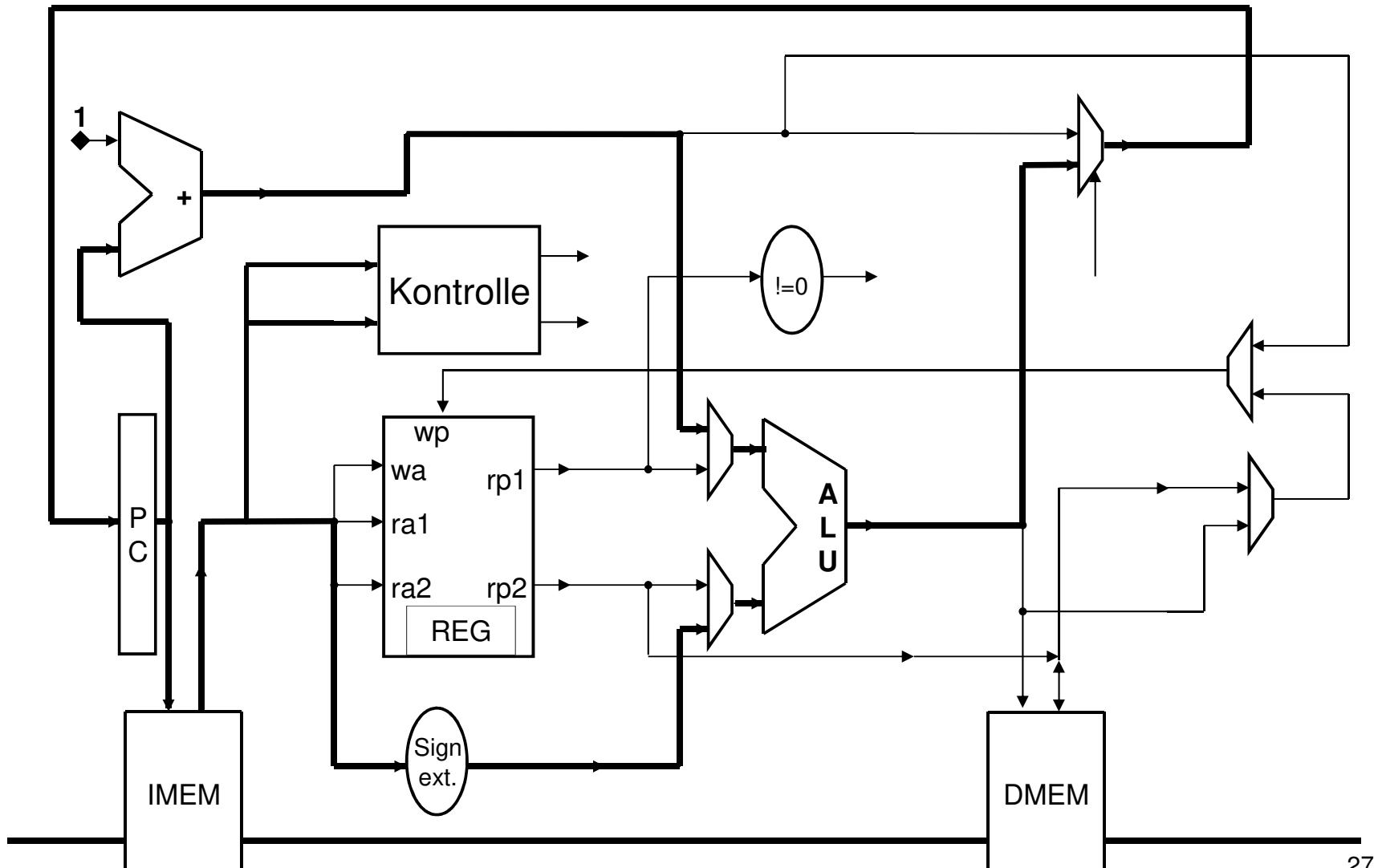
Bearbeitungsbeispiel: JAL



Bearbeitungsbeispiel: JREG



Bearbeitungsbeispiel: JMP



Kritik

Diese simple Realisierung bearbeitet pro Taktzyklus einen Befehl, die Zykluszeit ist aber mit

$$\Gamma_{clk} \geq 2 \cdot t_{access_cache} + 2 \cdot t_{access_reg} + t_{ALU} + t_0$$

enorm hoch. t_0 sei dabei die Laufzeit der Multiplexer und der Busse.

Dafür ist die Kontrolle sehr einfach zu bauen: sie besteht im wesentlichen nur aus Dekodierschaltungen, deren Kosten von der gewählten Kodierung der Befehle abhängen.

Geht es besser?

Eine Mehr-Zyklus Implementierung

Man kann die Abarbeitung der Befehle auch auf mehrere Taktzyklen abbilden.

Vorteile

- Laufen Speicherzugriffe für Daten und Befehle in unterschiedlichen Zyklen ab, kommt man wieder mit nur einem Speicher aus. **Von Neumann Architektur**
- Befehle, die nicht alle Ressourcen nutzen (z.B. Arithmetik greift nicht auf den Datencache zu) können in weniger Zyklen laufen.
- Man kann die Befehle im Fließbandverfahren bearbeiten, d.h. mehrere Befehle in verschiedenen Zyklen parallel (mit Abstrichen bei den ersten beiden Vorteilen). Dieses **Pipelining** Prinzip wird in allen modernen Prozessoren angewandt.

Beispiel: 5-Zyklus WüRC

Wir erkennen in unserer 1-Zyklus Maschine 5 zeitintensive Schritte:

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

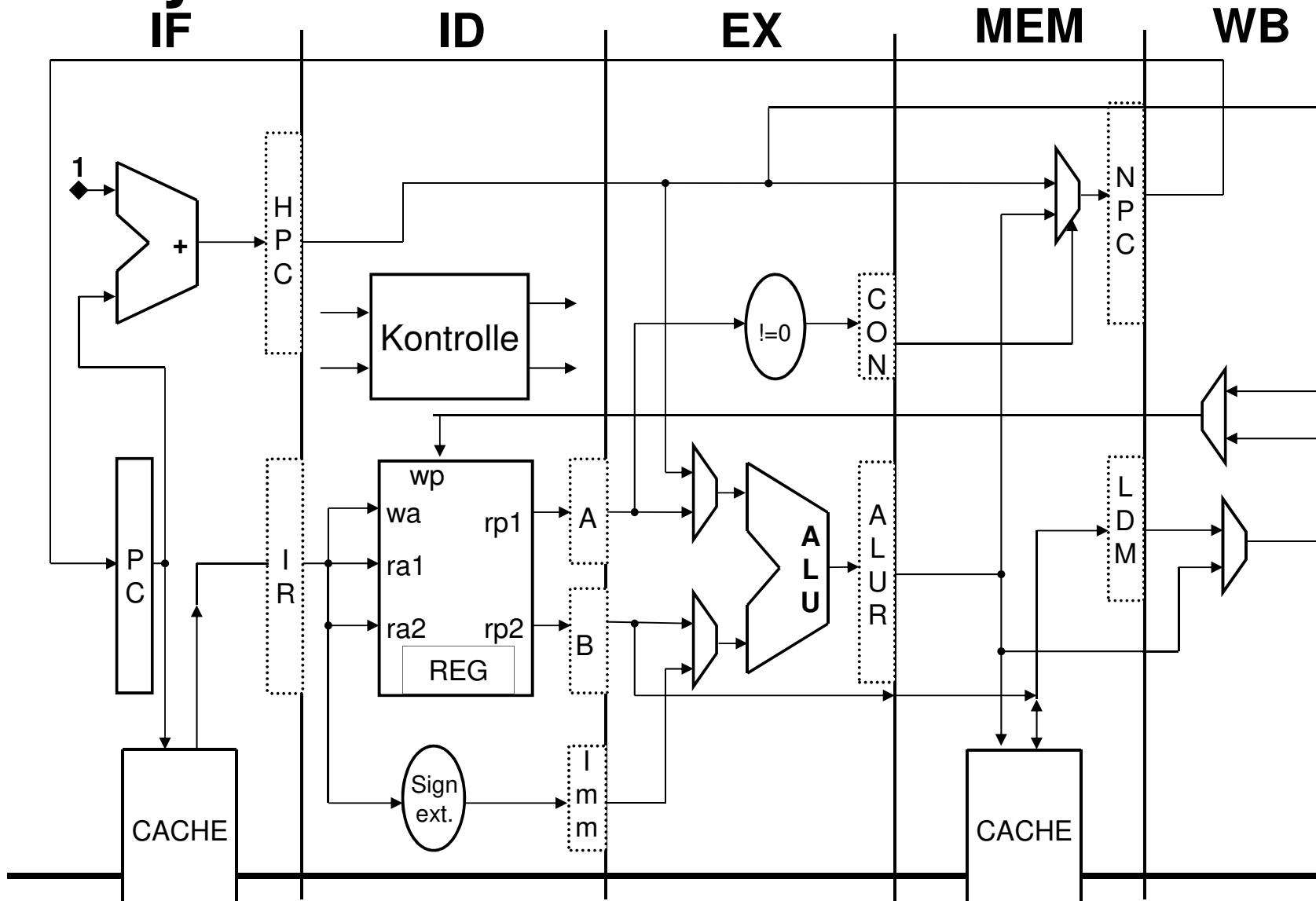
Nicht jede dieser Phasen wird von jedem Befehl echt benutzt.

5-Zyklus WüRC ff

Um diese 5 Phasen voneinander zu trennen, müssen wir

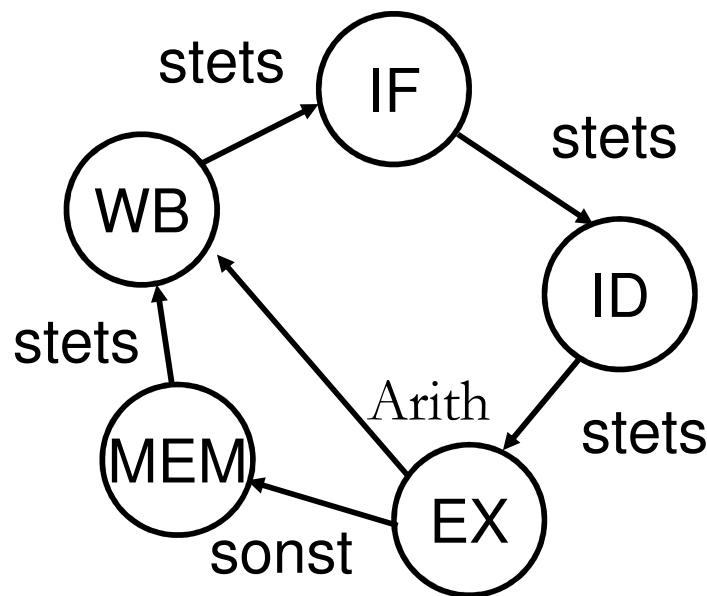
- zusätzliche Register einbauen, nämlich
 - Ein Instruktionsregister (IR), beschrieben in IF
 - Ein HilfsPC Register (HPC), beschrieben in IF
 - Operandenregister (A,B,Imm) beschrieben in ID
 - Ein Bedingungsregister (CON), beschrieben in EX
 - Ergebnisregister der ALU (ALUR) beschrieben in EX
 - Speicherdatenregister (LDM) beschrieben in MEM
 - Ein weiteres PC Register (NPC), beschrieben in MEM
- die Kontrolle in einen Automaten umwandeln, der die Kontrollssignale zu den richtigen Zeiten erzeugt.

5-Zyklus WüRC ff



5-Zyklus Kontrolle

Die Kontrolle unserer 5-Zyklus Version wird nun zur Finite State Machine, die in den jeweiligen Zyklen die entsprechenden Kontrollsignale für Register und Multiplexer erzeugt. Je nach Befehl kann sie auch Zyklen überspringen.



5-Zyklus WüRC -- Performanz

Die Taktperiode dieser Implementierung ist im wesentlichen bestimmt durch

$$\Gamma_{clk} \geq \max\{t_{access_cache}, t_{access_reg}, t_{ALU}\} + t_0$$

worin t_0 wieder die Laufzeit von Leitungen, Multiplexern und Dekodierlogik ist. Wir würden also etwa $4t_0$ gegenüber der Einzyklus Maschine verlieren, wenn Caches, Register und ALU etwa gleich schnell sind, aber

- wir benötigen eigentlich nur noch einen Cache und
- nicht alle Befehle dauern 5 Zyklen!

Die durchschnittliche Laufzeit hängt nun entscheidend vom **Instruktionsmix** ab!

Pipeline WüRC

Wenn wir darauf bestehen, dass

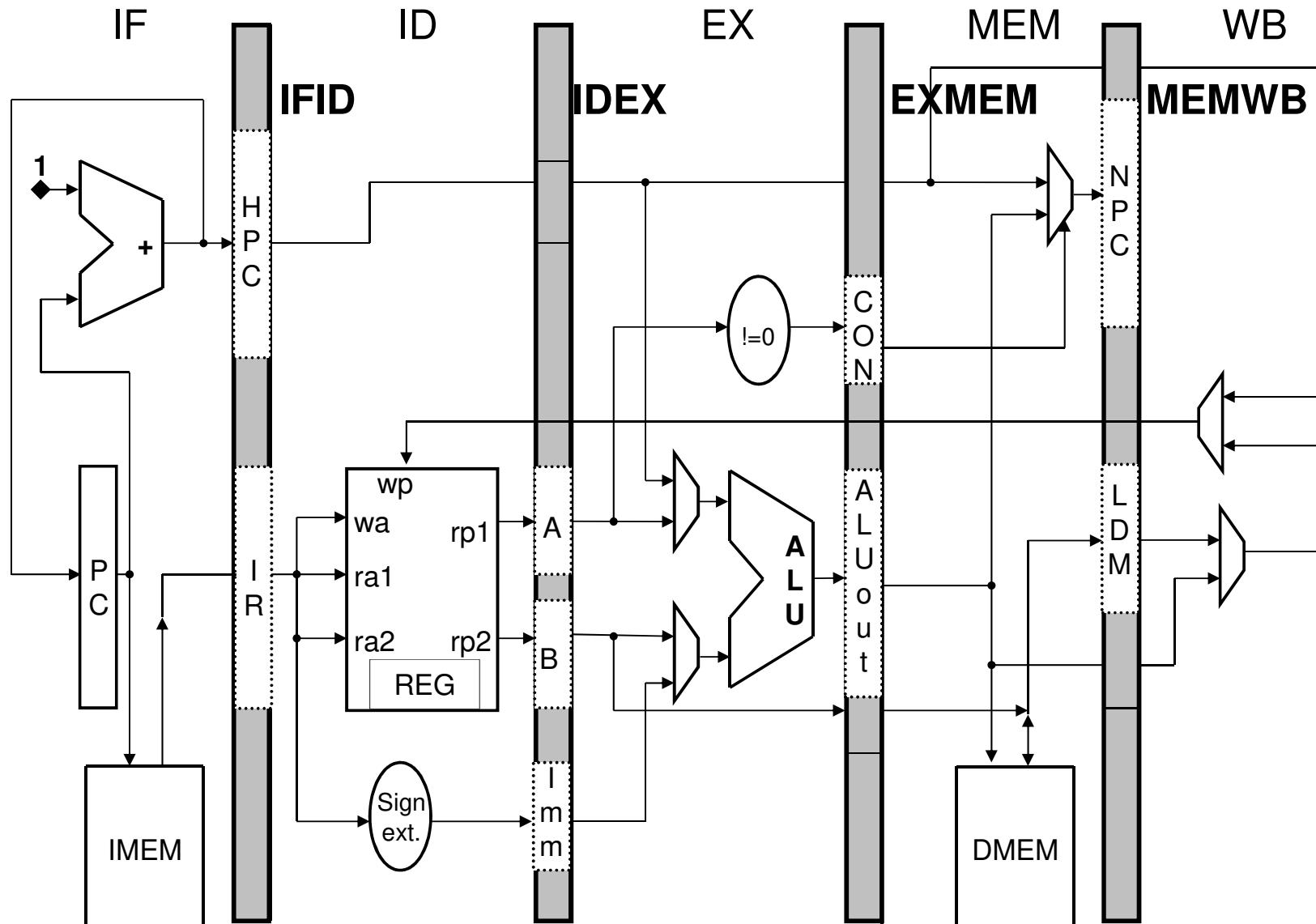
- Befehls- und Datencaches benutzt werden und
- jeder Befehl in 5 Zyklen bearbeitet wird,

dann kann man die Befehle parallel im **Fließbandprinzip** bearbeiten, d.h. zu einem Zeitpunkt in jedem Zyklus einen anderen Befehl.

Veranschaulichung für 8 Befehle: I1, ..., I8

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | Zyklus |
|----|----|----|----|----|----|----|----|----|----|----|----|--------|
| I1 | I2 | I3 | I4 | I5 | I6 | I7 | I8 | | | | | IF |
| | I1 | I2 | I3 | I4 | I5 | I6 | I7 | I8 | | | | ID |
| | | I1 | I2 | I3 | I4 | I5 | I6 | I7 | I8 | | | EX |
| | | | I1 | I2 | I3 | I4 | I5 | I6 | I7 | I8 | | MEM |
| | | | | I1 | I2 | I3 | I4 | I5 | I6 | I7 | I8 | WB |

Realisierung einer Pipeline: 1. Versuch



Realisierung: 1. Versuch

Wir nehmen zunächst an, dass wir eine Befehlsfolge ohne Sprünge verarbeiten. Dann liegt der PC + 1 schon direkt im nächsten Zyklus vor, und wir können in jedem Zyklus einen Befehl holen.

Damit uns Zwischenzustände der Maschine nicht verlorengehen, trennen wir jedes Paar von benachbarten Phasen durch ein Pipelineregister. Diese Register nennen wir nach den Phasen, die sie trennen: IFID, IDEX, EXMEM, MEMWB.

Es müssen nun sämtliche Kontrollsignale und Zwischenergebnisse, die in einer Stufe entstehen, durch alle nachfolgenden Pipelineregister weitergereicht werden. Wir benennen die Teile des Pipelineregisters entsprechend mit den Namen der Zwischenergebnisse:

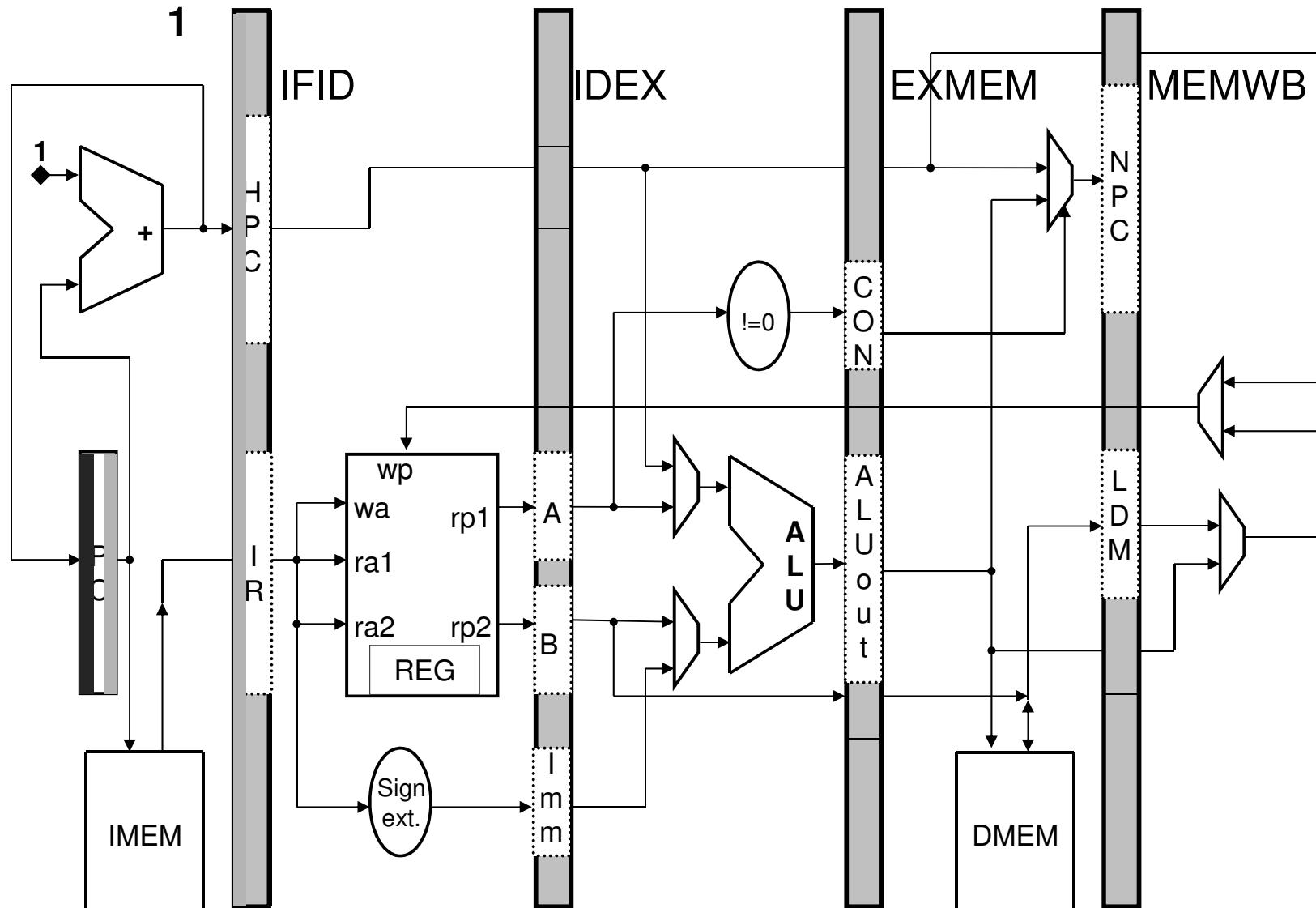
IR wird zu IFID.IR und findet sich ggf. dekodiert wieder in IDEX.IR, EXMEM.IR, MEMWB.IR

Ebenso wird HPC zu IFID.HPC, IDEX.HPC, ...

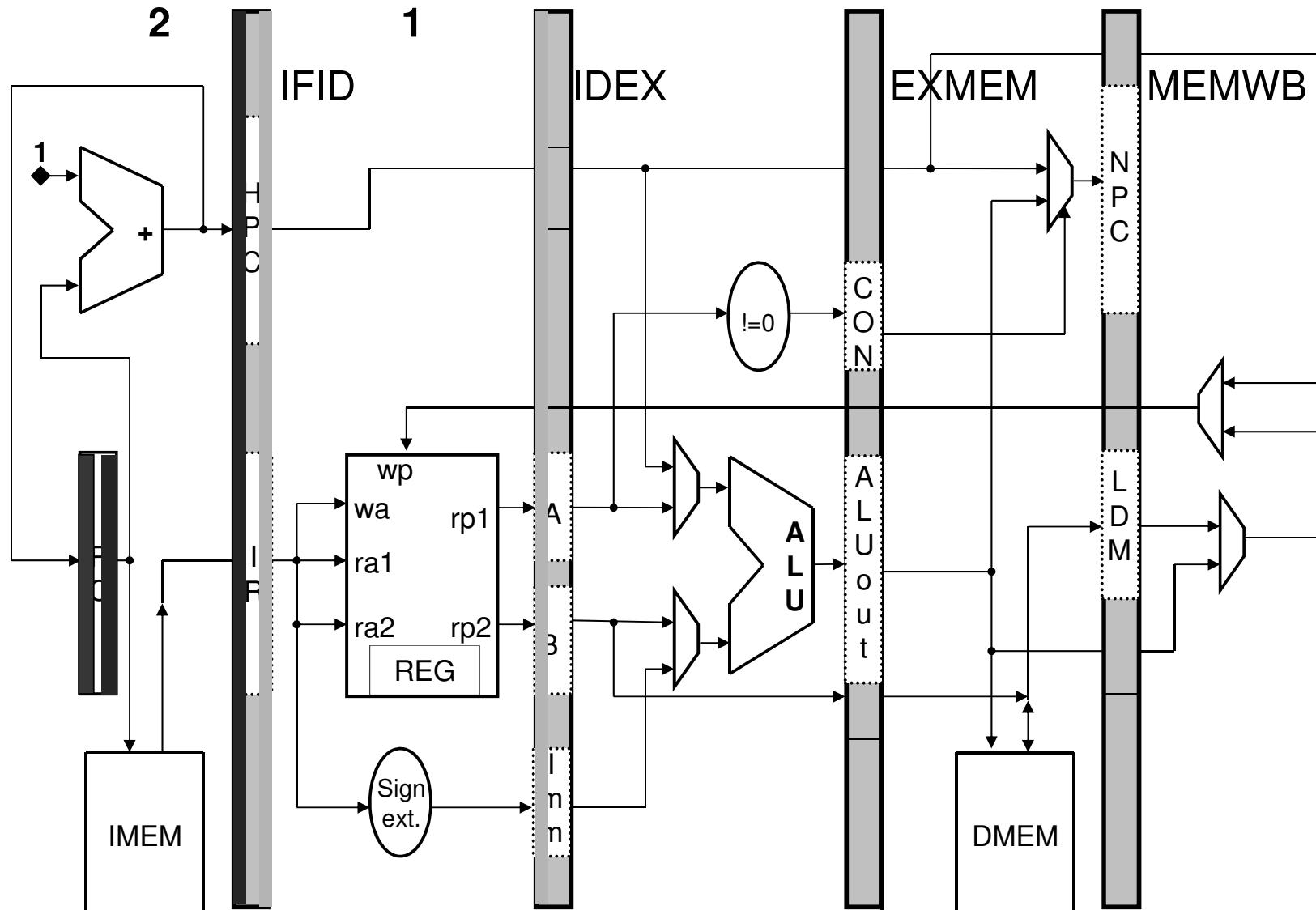
oder ALU_out zu EXMEM.ALU_out, MEMWB.ALU_out, ...

oder A zu IDEX.A, EXMEM.A,

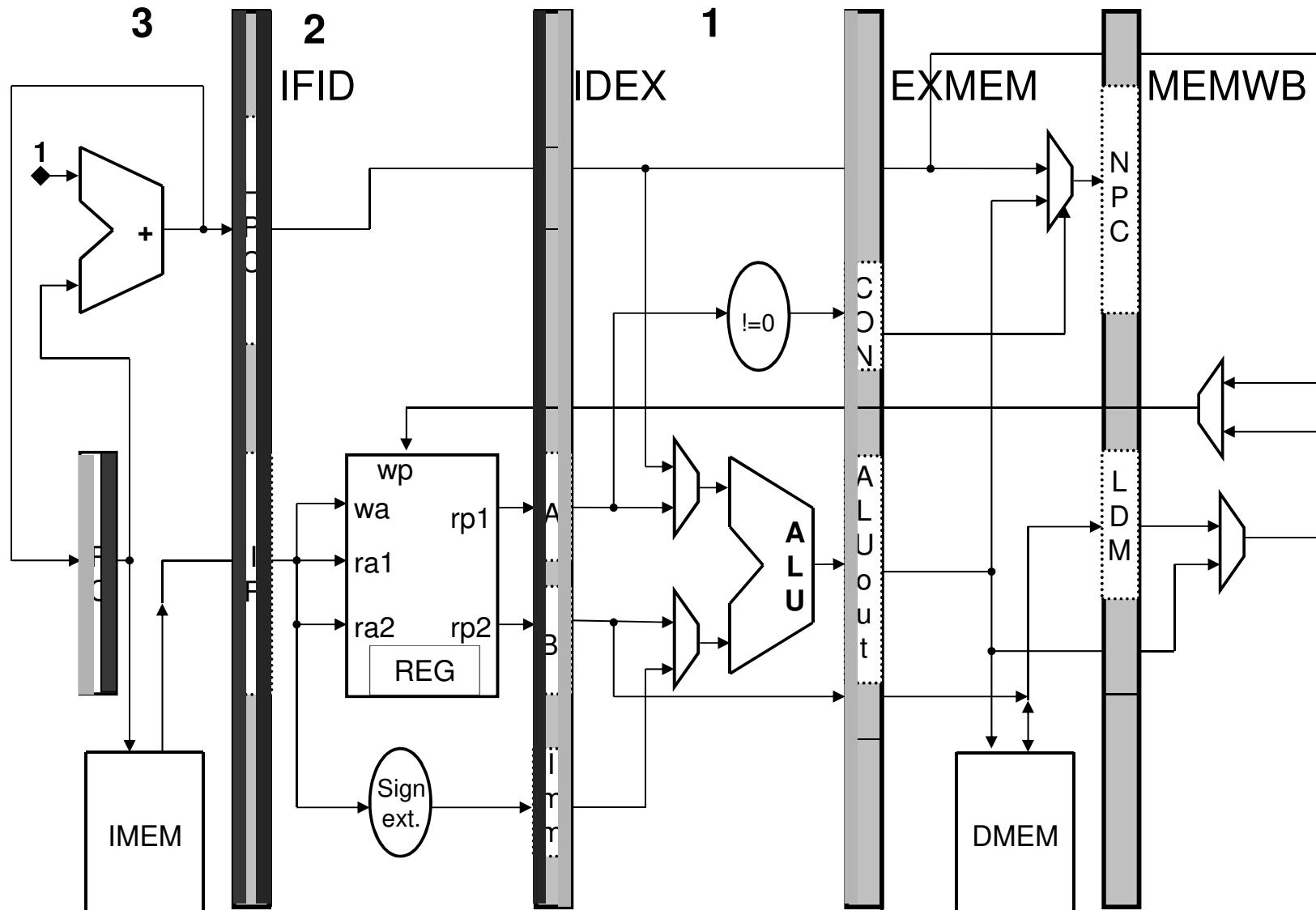
Realisierung einer Pipeline: 1. Versuch -- ff



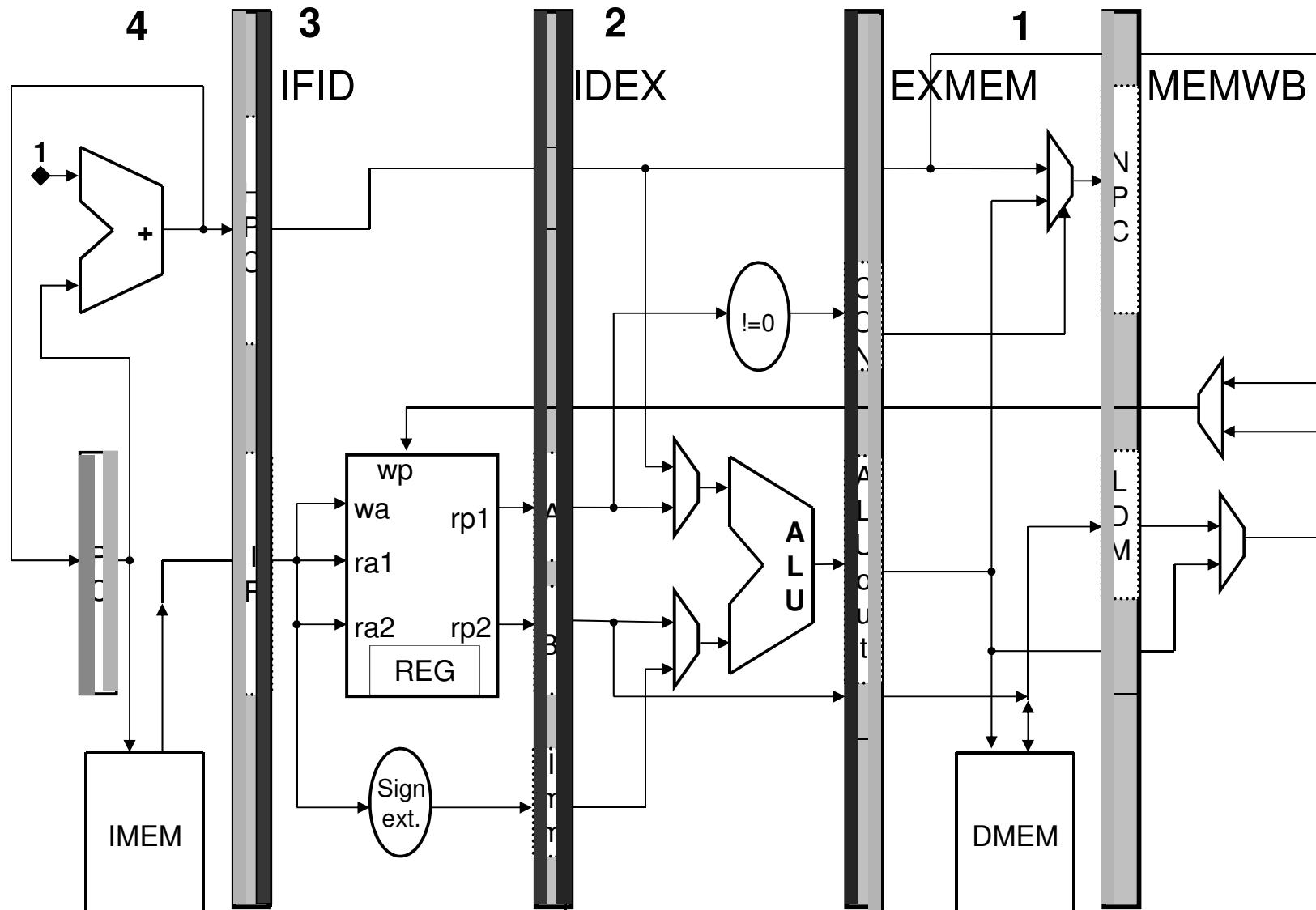
Realisierung einer Pipeline: 1. Versuch -- ff



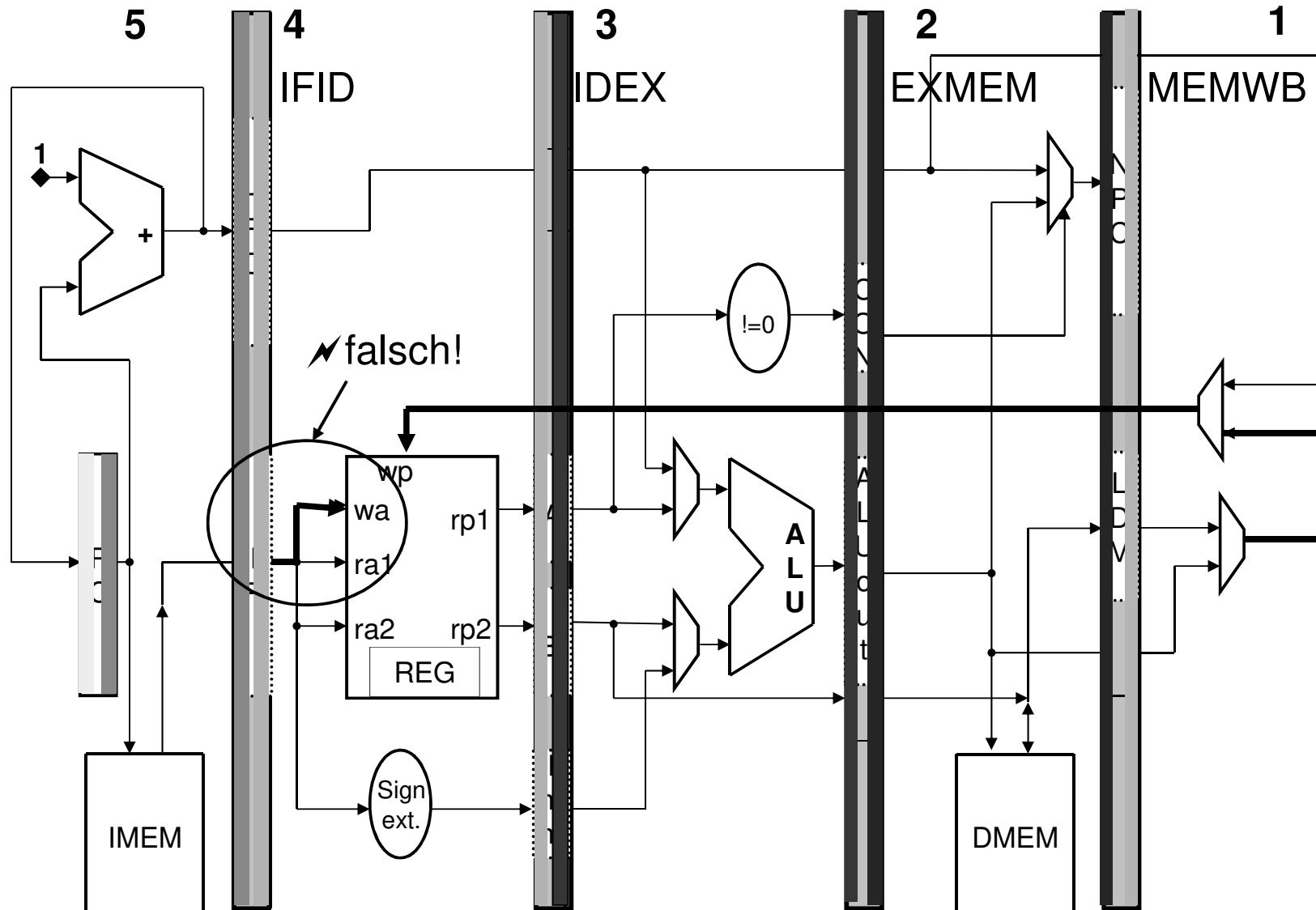
Realisierung einer Pipeline: 1. Versuch -- ff



Realisierung einer Pipeline: 1. Versuch -- ff



Realisierung einer Pipeline: 1. Versuch -- ff

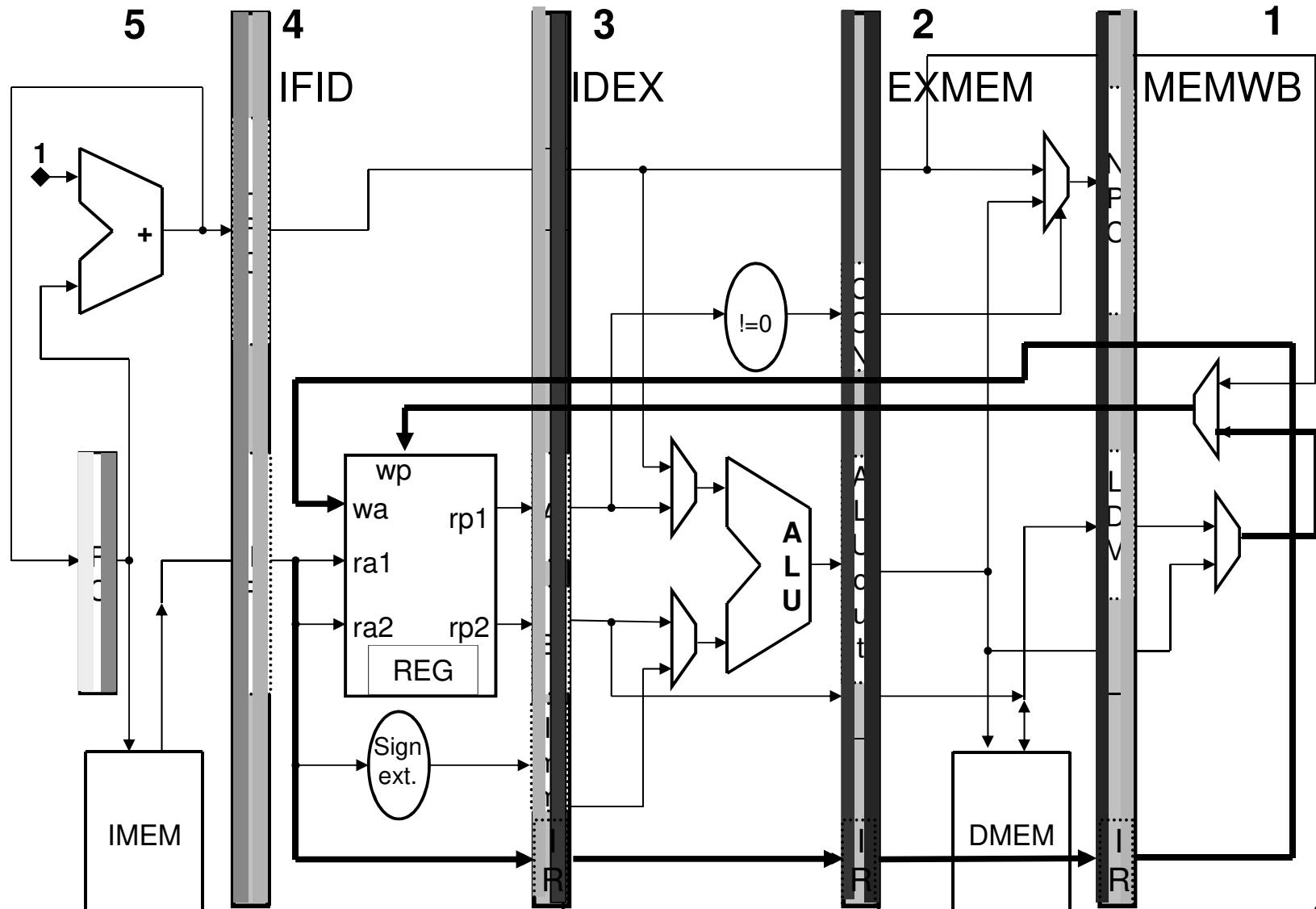


Fehler in der Hardware

Wir erkennen einen kapitalen Fehler in unserer Pipeline:
Die Adresse des Zieloperandens darf nicht direkt angelegt werden, sondern muss durch die Pipeline hindurchgeschleift werden und erst im WB als MEMWB.IR Anteil an die Registerbank gelegt werden.
Wir würden sonst den Zieloperand des Befehls 4 mit dem Ergebnis des Befehls 1 überschreiben!

Alle Kontrollinformation eines Befehls, die im *ID* Zyklus entsteht, muss bis zu der Pipelinestufe, in der sie wirkt, über Pipelineregister weitergereicht werden!

Realisierung einer Pipeline: 1. Versuch -- ff



Pipeline WüRC

Wäre es wirklich so einfach, dann könnten wir beliebige Befehlsfolgen der Länge n in $n+4$ Zyklen bearbeiten, was für große n einer Rate von ca. 1 Zyklus pro Befehl entspricht. Wir wären also fast 5 mal schneller als die ein Zyklus Maschine, wenn Cache- und ALU-Laufzeiten etwa gleich sind.

Pferdefüße:

- Datenhazards: Befehle in der Pipeline verändern Operanden anderer Befehle in der Pipeline.
- Kontrollhazards: Verzweigungen in der Pipeline machen Nachfolgebefehle ungültig.

Der prinzipielle Aufbau erfordert allerdings nur ein paar Register mehr!

Ausblick

Pipelining ist nicht so leicht umzusetzen wie hier skizziert, neben den schon erwähnten Problemen kommen noch Dinge wie

- Rechenwerke (z.B. Gleitkomma, Division, Quadratwurzel, ...) mit unterschiedlich tiefen Pipeline-stufen,
- Unterbrechungen und Ausnahmebedingungen,
- Code Scheduling, ...

Will man noch höhere Raten als einen Befehl pro Zyklus, muss man die Datenpfade breiter auslegen und versuchen, mehrere Befehle parallel zu holen und zu bearbeiten. → **Superskalare Prozessoren**

Dann wirds noch interessanter (schwieriger).

3.6 Ausnahmebehandlungen

Zur Vorlesung Rechenanlagen

SS 2019



Exceptions

Wir hatten schon beim Thema „Überlaufbehandlung“ auf spätere Kapitel verwiesen. Bisher haben wir nur die Möglichkeit wie folgt auf Überläufe zu reagieren:

...

```
ADD R3, R5, R4;  
BNOV #1;  
JAL R2, #Überlaufbehandlung;
```

Bei rückkehrender Behandlung (R2 müsste man dann immer frei halten!), bzw. sonst

...

```
ADD R3, R5, R4;  
BOV #Überlaufbehandlung;
```

Das wäre ein Riesenaufwand, wollte man das bei jeder Arithmetikoperation tun, die einen Überlauf erzeugen kann!

Exceptions

Bei einem Überlauf kann es auch sinnvoll sein das Programm fortzusetzen (rückkehrende Behandlung).

Es kann aber zu Situationen kommen, in denen ein Programm nicht mehr sinnvoll fortgesetzt werden kann:

Da wir beliebig in den Speicher schreiben dürfen, kann es zum Beispiel vorkommen, dass kein legaler Befehlscode mehr im Programmspeicher an der Stelle PC steht, weil

- wir den PC falsch gesetzt haben (z.B mit JREG)
- ein STORE auf eine Adresse im Programmreich gemacht haben.

In diesen Fällen sollte die CPU trotzdem noch sinnvoll reagieren können. **Wie?**

Exceptions

Solche, vom Programm verursachten, Situationen sollte die Maschine durch ihre Kontrolle abfangen, um

- das Programm beenden oder
- nach Bereinigung der Situation wieder fortsetzen

zu können.

Lösungsidee:

Implementiere beim Vorliegen solcher Ausnahmesituationen einen „automatischen Unterprogrammaufruf“ (ggf. des Betriebssystems), der zur Bearbeitung des auslösenden Befehls zurückkehren kann.

Wir nennen solche **interne** Unterbrechungen **Exceptions** oder **Ausnahmebehandlungen**.

Interrupts

Wenn man mal den Schritt gemacht hat, dass die CPU auf interne, vom Programm verursachte Ereignisse reagiert, kann man diesen auch auf externe Ereignisse erweitern.

Mit unserer bisherigen Maschine müssten wir langsame Geräte (Platten, Netzwerk, Tastatur, Maus,...) in mehr oder weniger regelmäßigen Zeitabständen auf ihren Zustand prüfen (polling). Dazu müsste man alle Programme so schreiben, dass sie z.B. alle 50 Befehle äußere Ereignisse abfragen und diese via Unterprogramm behandeln.

Das wäre ein unvertretbarer Aufwand ähnlich wie bei Überläufen.

Da diese Unterbrechungen nicht vom Programm selbst, sondern von außen verursacht werden, bezeichnen wir sie als (externe) **Unterbrechungen** oder **Interrupts**.

Erweiterungen der WüRC

Folgende Maßnahmen sind zur Erweiterung der WüRC nötig:

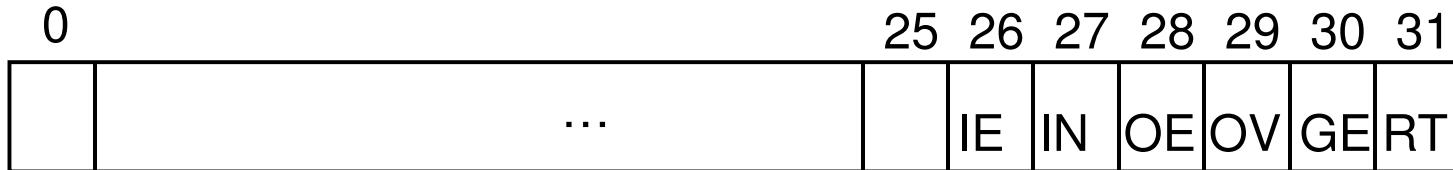
1. Wir geben dem Register R1 die Rolle eines Statusregisters.
Es sollte nicht mehr zum Zwischenspeichern von Rechenergebnissen benutzt werden. Ferner werden Flags der Funktionseinheiten bei jeder Operation zum Statusregister „geodert“. Wir nennen R1 fortan **SR**.
2. Wir geben einem festen Register die Rolle des Stackpointers, zum Beispiel R2. Wir nennen es von nun an **SP**.
3. Wir führen von aussen eine Leitung **INT** in die CPU, die Ereignisse in der Peripherie mit 1 signalisiert.

Damit bleiben der WüRC nur noch ≤ 13 General Purpose Register.

Wir stellen uns vor, dass die Hardware, die die Peripherie steuert, seitens der CPU durch den Inhalt von Spezialregistern sichtbar und bedienbar ist. Diese Spezialregister liegen auf festen Adressen im Speicher (Memory Mapped I/O).

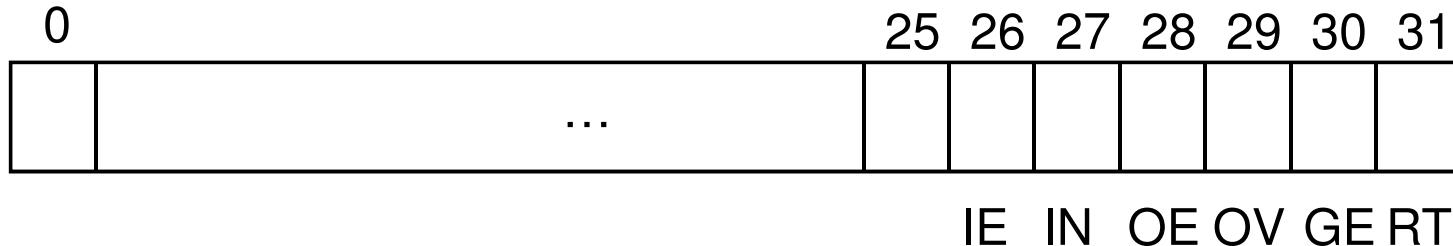
Das Statusregister

Wir stellen uns für das Statusregister zunächst mal folgende, sehr einfache Belegung vor. Da noch viel Platz im SR ist, sind auch viele Erweiterungen denkbar:



- RT: Resetbit. SR[31] wird gesetzt, wenn eine illegale Instruktion dekodiert wurde. Kann nicht disabled werden.
- GE: Global Interrupt Enable. Wird während der Behandlung von Unterbrechungen zurückgesetzt.
- OV: Das Overflowbit wird stets in SR[29] nach jeder Operation „geodert“.
- OE: Overflow Enable. Ist SR[28] = 1 wird durch ein gesetztes Overflowbit eine Exception ausgelöst, sofern SR[30] = 1 ist.
- IN: Das externe INT Signal wird nach jedem Takt auf SR[27] dazu „geodert“.
- IE: Interrupt Enable. Ist SR[26] = 1 wird durch ein gesetztes IN Bit eine Exception ausgelöst, sofern SR[30] = 1 ist.

Das Statusregister



Es macht Sinn, gleich zu allen diesen Flags folgende z.T. gleichnamige Konstanten zu definieren:

RT := 0x00000001;

GE := 0x00000002;

OV := 0x00000004;

OE := 0x00000008;

IN := 0x00000010;

IE := 0x00000020;

Auslösen von Unterbrechungen

Wir brauchen nun noch Befehle, die bei Annahme einer Unterbrechung den PC auf den Stack schieben, ohne ein Register zu verändern, und bei der Rückkehr den PC vom Stack holen, ohne ein Register zu verändern.

Wir nennen diese **TRAP** und **RFE**.

TRAP wird durch einen freien Code kodiert. Er bewirkt

- ein Store des PC auf den Stack und
- ein Dekrement des Stackpointers (Stack stets von MaxAdr abwärts!)
- ein Rücksetzen des GE-Bit im SR.
- ein Durchstarten auf Adresse 0.

RFE wird durch einen freien Code kodiert. Er bewirkt

- ein Laden des PC von Stackpointer + 1
- ein Inkrement des Stackpointers (Stack stets von MaxAdr abwärts!)
- ein Setzen des GE Bit im SR.

Auslösen von Unterbrechungen

Die Instruction Fetch Stufe schaut stets beim Laden des IR auch auf die Flags des Statusregisters.

Gilt

$$(GE \text{ and } ((OV \text{ and } OE) \text{ or } (IN \text{ and } IE) \text{ or } RT)) = 1$$

dann lädt die Instruction Fetch Stufe den Befehl **TRAP** ins IR. Dieser steht nicht im Programmspeicher, sondern wird von der Kontrolle der Instruction Fetch Stufe erzeugt.

Behandeln von Unterbrechungen

Ab Adresse 0 steht nun immer ein Programmstück, das zunächst den Fall eines echten Reset (SR = 0) oder einer angenommenen Unterbrechung unterscheidet. Bei angenommenen Unterbrechungen werden diese in Software behandelt.

Wir skizzieren möglichen Code beispielhaft:

```
0:BNEZ SR,#1; -- Überspringe Programmstart bei INT
1:JMP #Programmstart;
2:SUBI SP,SP,#1;
3:STORE R3,SP,#1; -- Rette R3
4:ANDI R3,SR,#RT;
5:BEQZ R3,#ExceptOrInter;
Reset:AND SR,SR,R0; -- SR rücksetzen
7:..... -- Reset ggf behandeln* und dann neu starten
```

Behandeln von Unterbrechungen

```
ExeptOrInter:  
    ANDI R3, SR, #OE; -- teste Overflow enable  
    BEQZ R3, #Interrupt -- Overflow disabled  
    ANDI R3, SR, #OV;  
    BEQZ R3, #Interrupt; -- 0 = keine Overflow Exception!  
    XORI SR, SR, #OV; -- Overflowflag rücksetzen  
    JAL R3, #Overflowhandler; -- Overflow behandeln*  
    JMP #Return; -- Rückkehrsequenz ausführen  
    -- Interrupt muss enabled sein und vorliegen  
    -- Interrupt behandeln:  
    Interrupt: JAL R3, #Interrupthandler; -- s.u.*  
    Return: LOAD R3, SP, #1; -- R3 Wiederherstellen  
    ADDI SP, SP, #1;  
    RFE; -- Rückkehr aus Exception  
    -- *) Achtung: alle Handler müssen in Callee Saves  
    -- mit R3 als Rücksprungübergabe implementiert sein!
```

Unterbrechungen Ausblick

Wir haben hier die einfachste Erweiterung betrachtet, damit wir nicht so viel ändern müssen.

In der Praxis können solche Konzepte komplexer sein. Ein paar Eckpunkte sind:

- Geschachtelte Unterbrechungen:

Wir haben während des Behandlins von Unterbrechungen keine weiteren Unterbrechungen mehr zugelassen. Das kann vor allem dann, wenn viel Peripherie über Interrupts zu bedienen ist, zu zu langen Reaktionszeiten führen. Also lässt man Unterbrechungen höherer Ebene zu, während man niedrigere Ebenen behandelt. (SR muss gesichert werden, TRAP #level statt TRAP, Level Enable Bits statt GE Bit....)

Unterbrechungen Ausblick

- Unterschiedliche Einsprungadressen:

Wenn man viele Ursachen für Interrupt oder Exception hat, müsste man in unserem Fall viel Zeit damit verbringen, in Falldiskussionen die Ursache herauszufinden und den richtigen Handler aufzurufen. Dies kann man vereinfachen, wenn man mehrere Einsprungadressen für TRAP Befehle hat, z.B. für jedes Level eine feste Adresse im Speicher.

In manchen Maschinen stehen diese Adressen in (einem oder mehreren) Spezialregistern, in anderen sind sie dem Level fest zugeordnet.

Unterbrechungen Ausblick

Häufig kann man die Empfindlichkeit gegenüber Ausnahmesituationen und Unterbrechungen vom Programm (oder System) aus einstellen, indem man Maskierungsbits setzt. Beispiele für weitere Unterbrechungsmechanismen sind

- Breakpoints
- Verschiedene Genauigkeitsgrade (z.B. Gleitkomma)
- Externe Signale
- Verschiedene Privilegstufen
- Systemaufrufe
- Speicherverwaltung,...

3.7 Die Speicherhierarchie

Zur Vorlesung Rechenanlagen

SS 2019



Allgemeines

Bisher haben wir nur sehr kleine, schnelle Speicher für Programme und Daten betrachtet (Harvard Architektur).

Problem:

Große Speicher haben zu hohe Zugriffszeiten,
Kleine Speicher schränken den Raum möglicher Anwendungen zu sehr ein.

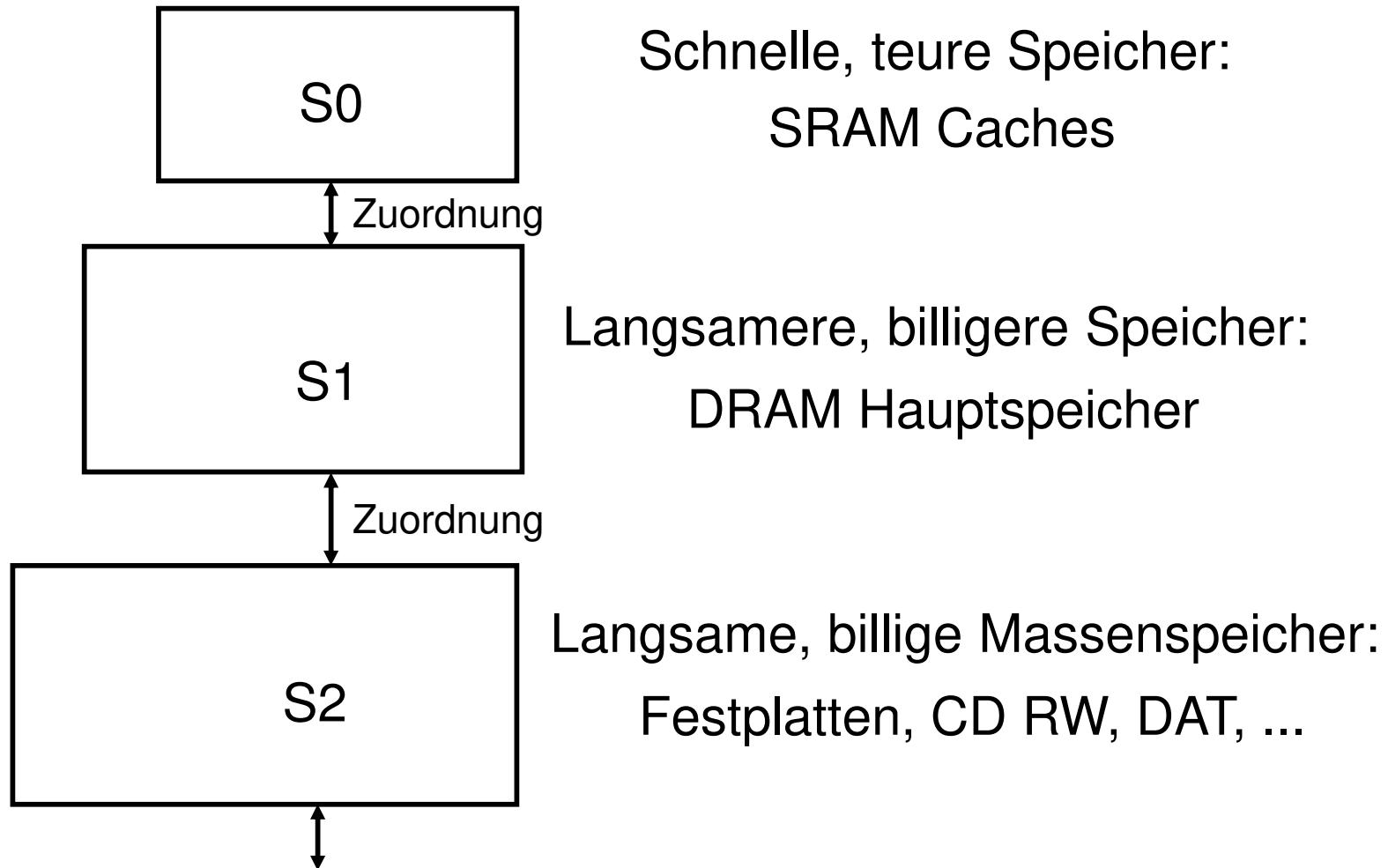
Ziel:

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

Lösung:

Konstruiere eine Hierarchie von Speichern

Speicherhierarchie



Speicherhierarchie ff

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

Probleme:

- Erhalte eine konsistente Zuordnung zwischen den Speichern
- Erzielle eine möglichst niedrige, mittlere Zugriffszeit
- Benutze möglichst einheitliche Adressen.

Zeitliche und räumliche Lokalität

Wie erreicht man eine niedrige „mittlere“ Zugriffszeit?

Wir legen dem Verhalten typischer Programme einfach folgende Annahme zugrunde:

Annahme: (Prinzip der zeitlichen und räumlichen Lokalität)

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

Solange man durch die zu konstruierende Zuordnung nicht verhindert, dass nah benachbarte Daten oder Befehle nicht gleichzeitig in einem Speicher niedriger Hierarchie liegen dürfen, sollte die Zugriffszeit im Mittel sehr niedrig sein.

Zeitliche und räumliche Lokalität ff

Bemerkungen:

Man kann in Kenntnis der Zuordnung natürlich Programme stets vorsätzlich so schreiben bzw. organisieren, dass sie möglichst hohe Zugriffszeiten liefern.

Es wird offenkundig, dass dem Compiler eine wichtige Aufgabe beim Planen und der Vergabe des Speichers zukommt, und eine genaue Kenntnis der Speicherorganisation der Zielmaschine erforderlich ist.

Architektur und Compiler stehen bei der Entwicklung neuer Maschinen in untrennbarer Wechselwirkung.

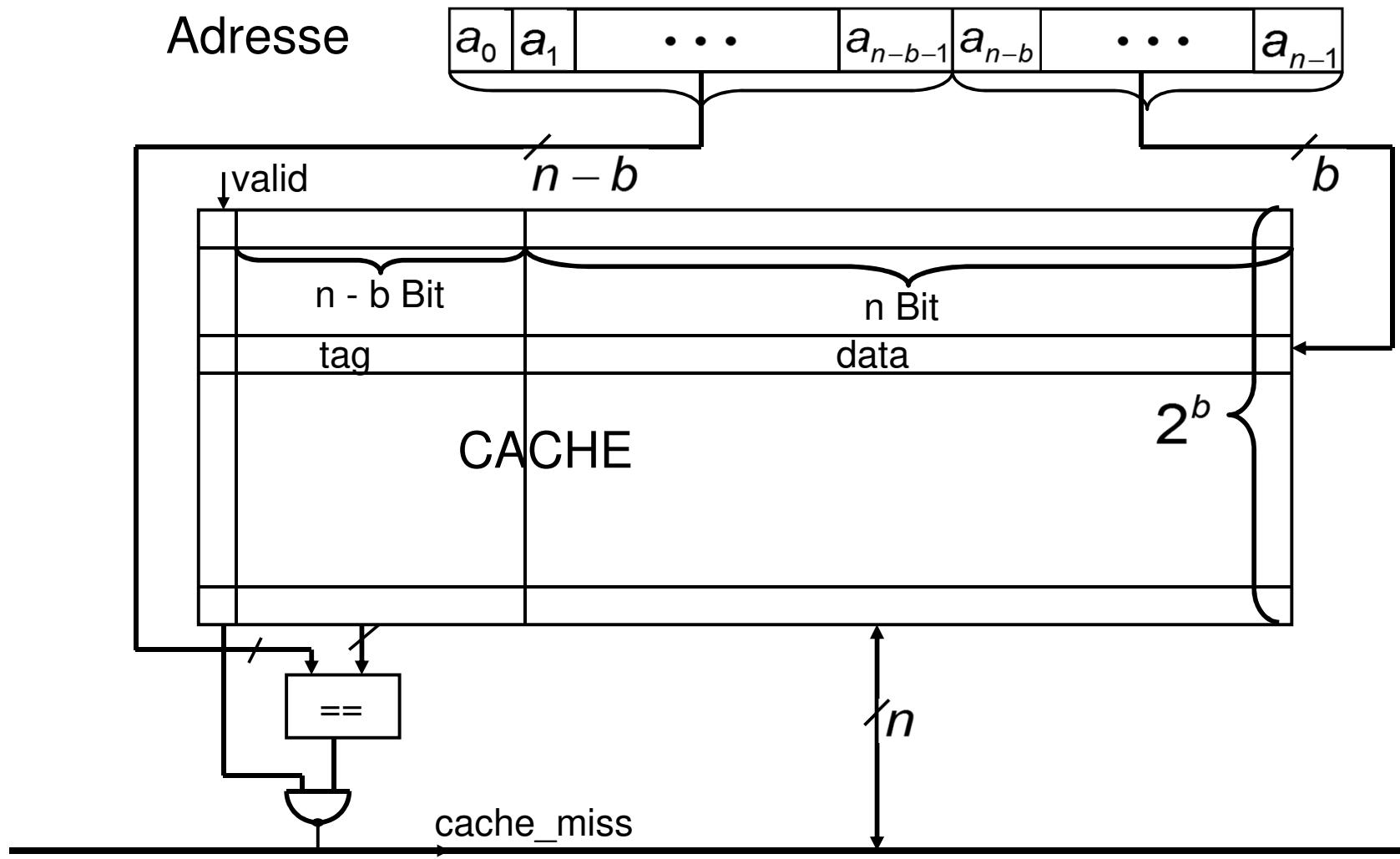
Caches

Wir wollen das Zusammenwirken zwischen Caches und Hauptspeicher zunächst in den Vordergrund stellen.

Eine sehr billige und weitverbreitete Technik ist die der sogenannten **direkten Zuordnung**.

Gegeben sei ein wortorientierter Cache der Adressbreite $b < n$. Wir realisieren die direkte Zuordnung nach folgendem Schema:

Cache: direkte Zuordnung



Direkte Zuordnung ff

Wir speichern neben einem Datenwort auch die führenden $n-b$ Bit seiner Adresse als sogenanntes tag field im Cache. Bei einem Zugriff auf den Cache mit b -bit Adressraum vergleichen wir stets das tag Feld mit den führenden $n-b$ Bits der aktuellen Adresse, um sicher zu gehen, ob der Eintrag für diese Adresse auch gültig ist. Um beim Start mit leerem Cache eine gültige Zuordnung zu haben, nehmen wir noch ein weiteres Bit, das valid Bit hinzu, das anzeigt, ob das aktuelle tag auch gültig ist.

Damit können im Cache an der Stelle r alle Speicherzellen stehen, für deren Adresse a gilt:

$$a \bmod 2^b = r$$

Solange also zwei Zellen den Abstand $< 2^b$ haben, können sie gleichzeitig im Cache aufbewahrt werden.

Direkte Zuordnung ff

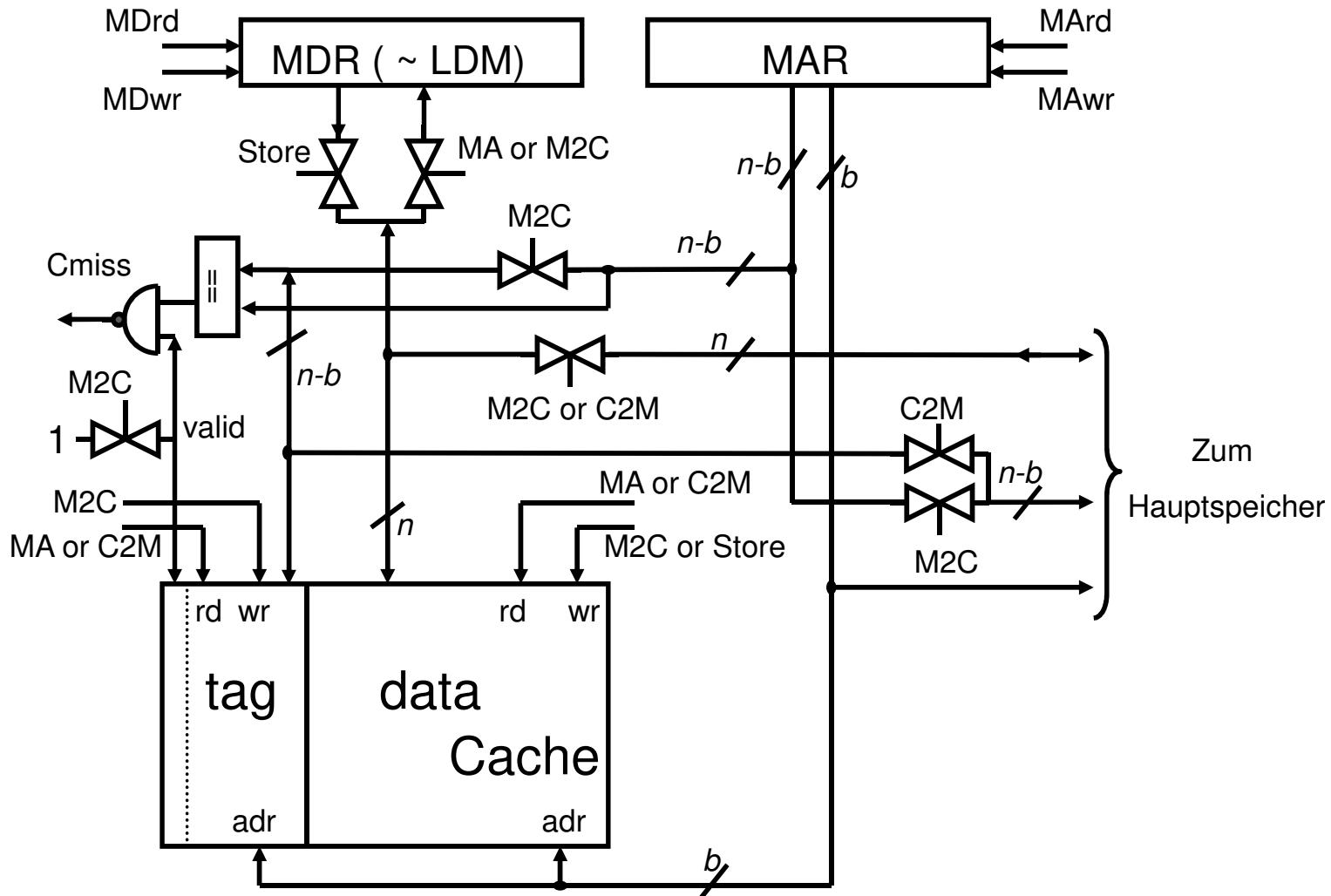
Lokalität ist also gewahrt.

Beispiel:

Betrachten wir eine Maschine mit 128 MB Hauptspeicher und 512 kB Cache. Dann benötigen wir 27-bit Adressbreite und 19-bit Adressbreite für den Cache. D.h. das tag kostet uns ein zusätzliches Byte Speicher pro Zelle. Hinzu kommt noch ein Bit für das valid bit.

Diese Lösung lässt sich alleine mit einem Vergleicher, einem Gatter und Speicherbausteinen realisieren. Hinzu kommt noch eine Erweiterung der Kontrolle des Prozessors, die cache misses in den entsprechenden Zuständen, in denen sie vorkommen können (IF, MEM) behandelt. Wir betrachten dazu zunächst die Anbindung zwischen Cache und Hauptspeicher:

Anbindung an den Hauptspeicher



Erläuterungen zur Anbindung

Wir haben nun einige Kontrollsignale dazugenommen, die Ansteuerung des Hauptspeichers selbst lassen wir weg:

- Cmiss: zeigt ein Cache miss an
- MA: zeigt Memory Access Zyklus an
- M2C: Transport zwischen Hauptspeicher und Cache
- C2M: Transport zwischen Cache und Hauptspeicher
- Store: Wegspeichern des MDR (Memory Data Register) im Cache (Store Befehl)
- MDrd,MDwr: Lese/Schreibkontrolls Signale des Datenregisters
- MArd,MAwr: Kontrolls Signale des Adressregisters

Wir skizzieren nun die Kontrolle:

Kontrolle der Cache/HS Anbindung

Ideen:

Wir führen im MEM Zyklus stets einen Lesezugriff auf den Cache aus.

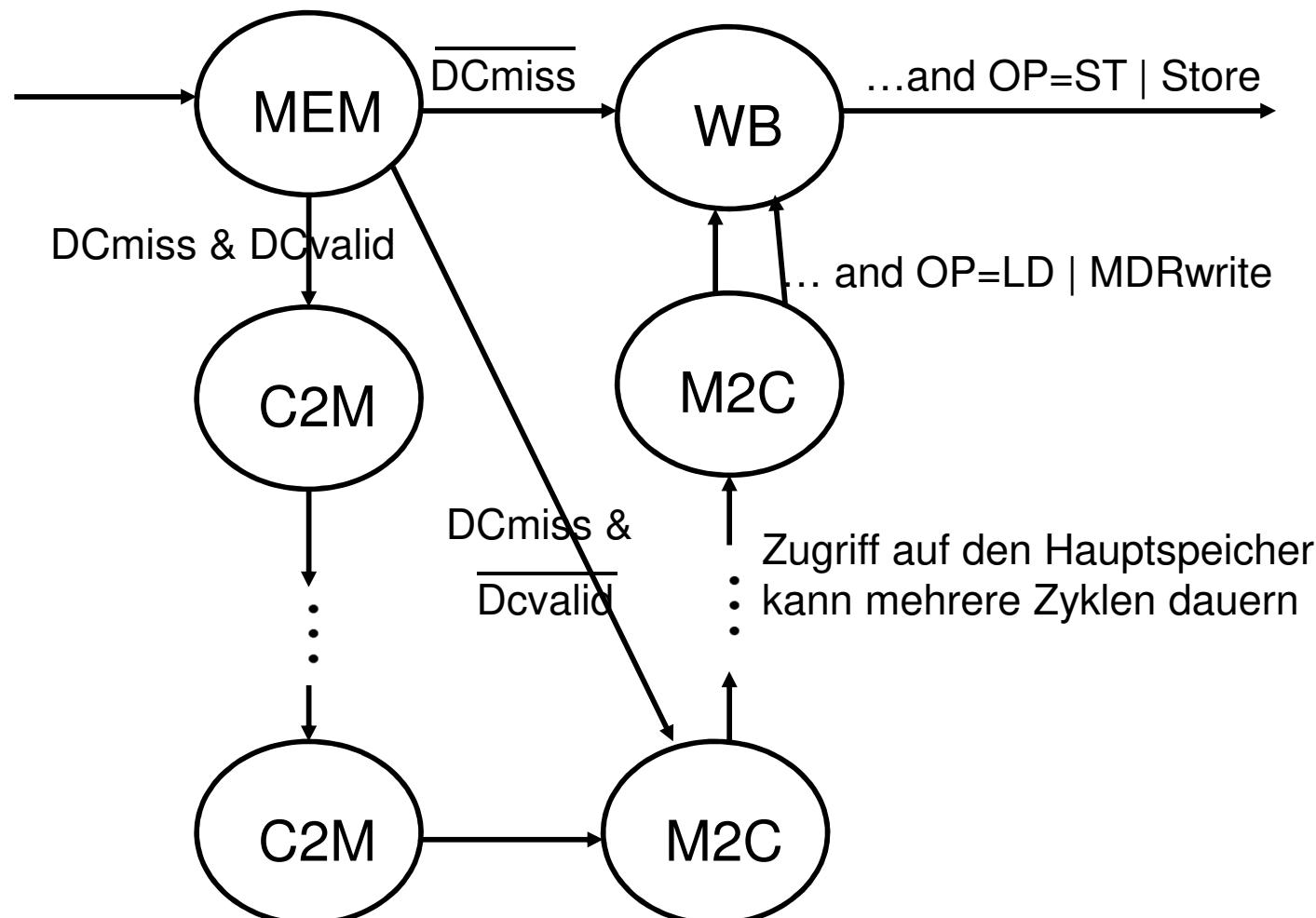
Datum gültig: Dann können wir einen **LOAD** Befehl auch schon als ausgeführt betrachten, wenn wir das MDR bei einem **LOAD** schon im MEM-Zyklus beschreiben.

Datum ungültig: Nun müssen wir erst
 das ungültige Datum retten → **C2M**
 das gültige Datum laden → **M2C**
 (**dabei das tag ersetzen!**)

Load: wir können beim Laden des gültigen Datums das MDR überschreiben.

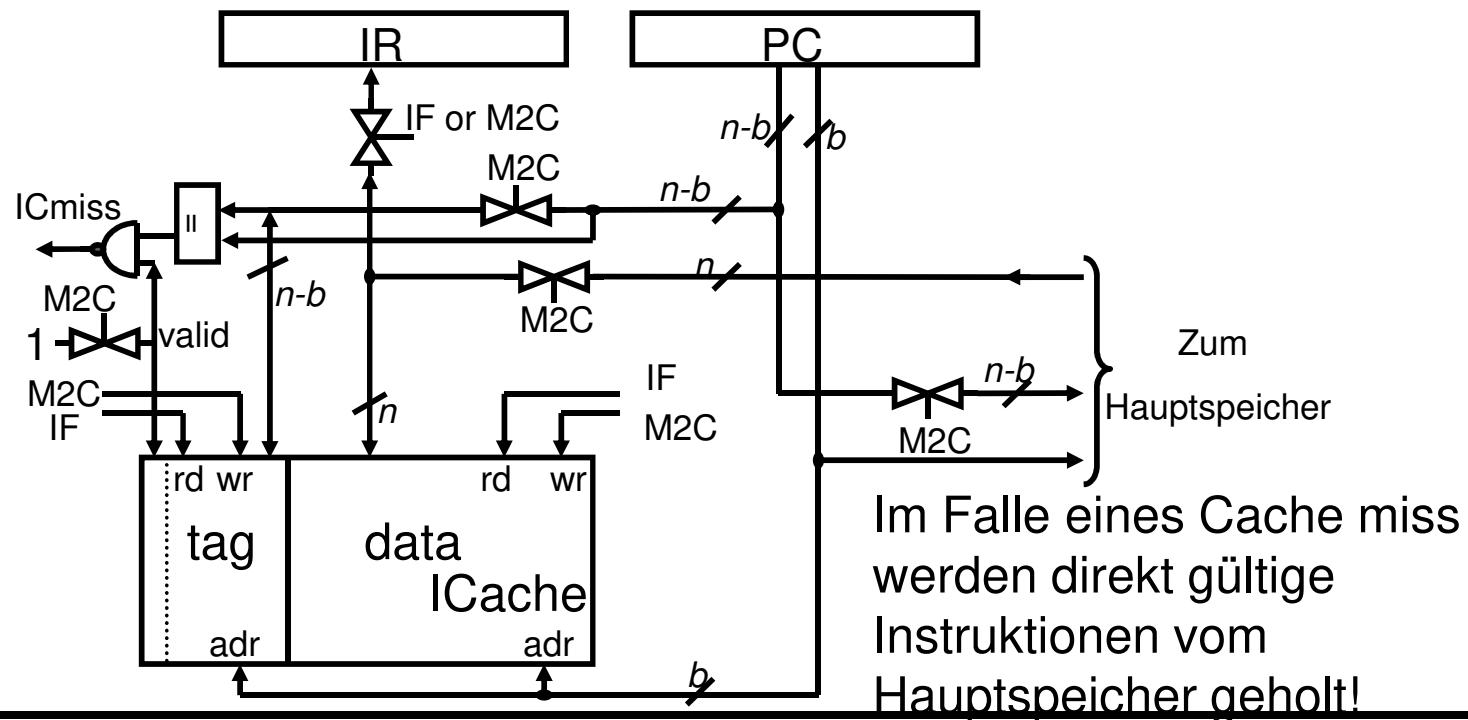
Store: wir verschieben das eigentliche Überschreiben in den WB-Zyklus.

Kontrolle für Datencache



Instruction Cache

Hier ist die Anbindung sehr viel einfacher, weil nur lesend zugegriffen wird. Im Wesentlichen entfallen einfach nur Teile der Ansteuerung und der Kontrolle zum Retten des Inhalts eines ungültigen Eintrags:



Virtuelle Adressierung

Der Umgang mit Maschinenworten als Hauptspeicheradressen ist problematisch:

- Man kann selten den ganzen Adressraum als RAM Speicher realisieren (32 Bit ~ 4,3 GB, 64 Bit ~ niemals)
- Start von Programmen in unterschiedlichen Speicherbereichen, oder unterschiedlichen Ausbaustufen von Maschinen (!?)
- Unterstützung von multi tasking, oder gar multi user Systemen

Jedes vernünftige Betriebssystem sollte den quasi-parallelen Betrieb mehrerer Programme unterstützen (Peripherie, Netzwerk,...) und für Fairness und Schutz der Programme untereinander sorgen!

Virtuelle Adressierung ff

Idee:

jedes Programm rechnet virtuell auf dem gesamten Adressraum der Maschine. Dazu müssen

Maschinenworte = **virtuelle Adressen**

in

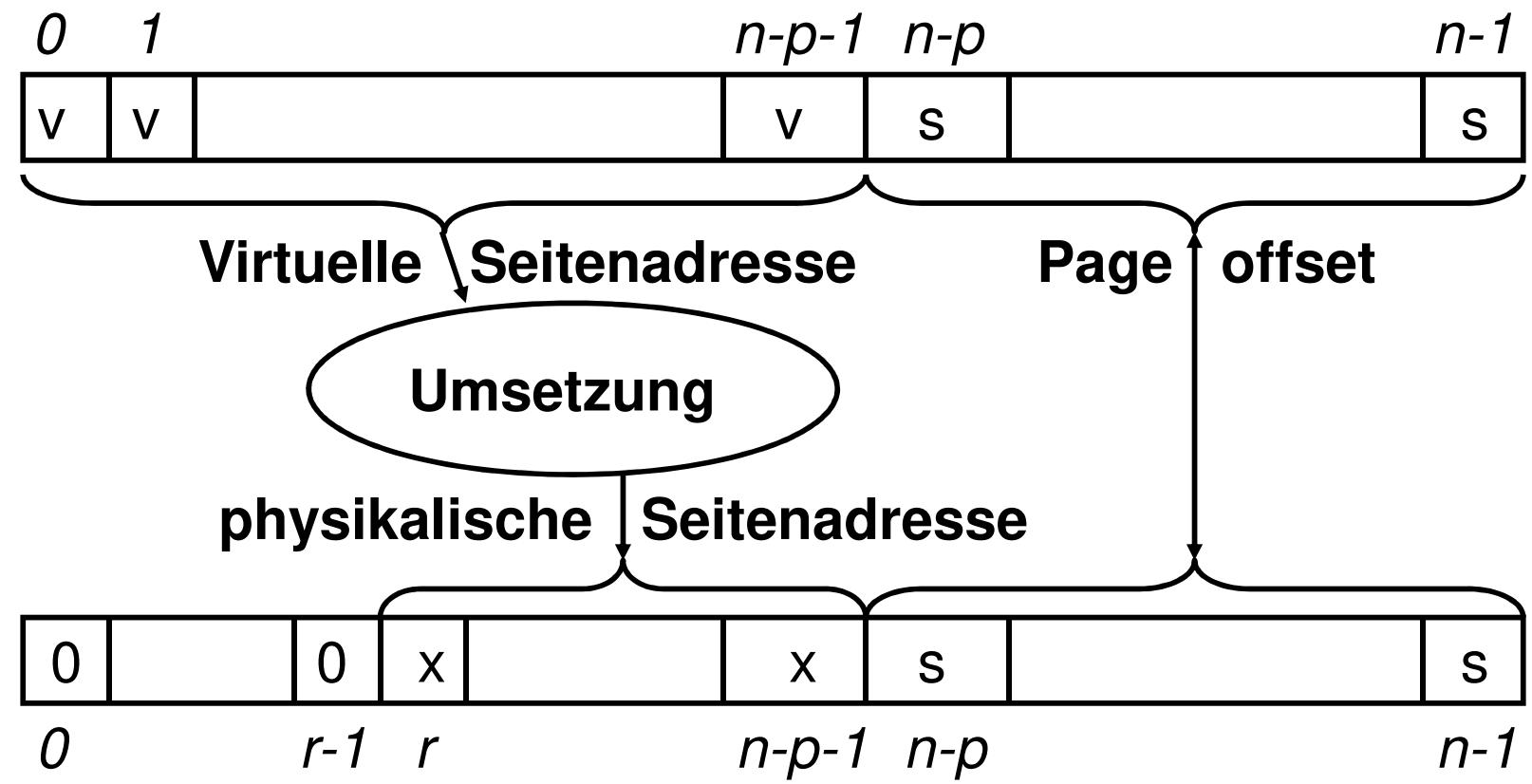
physikalische Adressen von $[0:M-1]$,

wo M die Speichergröße ist, umgerechnet werden. Diese Umsetzung übernimmt ein privilegiertes Programm, das Betriebssystem.

Bei einem solchen Vorgehen sind alle Programme vor Übergriffen geschützt, wenn die Zuteilung und Umsetzung des physischen Speichers korrekt ist.

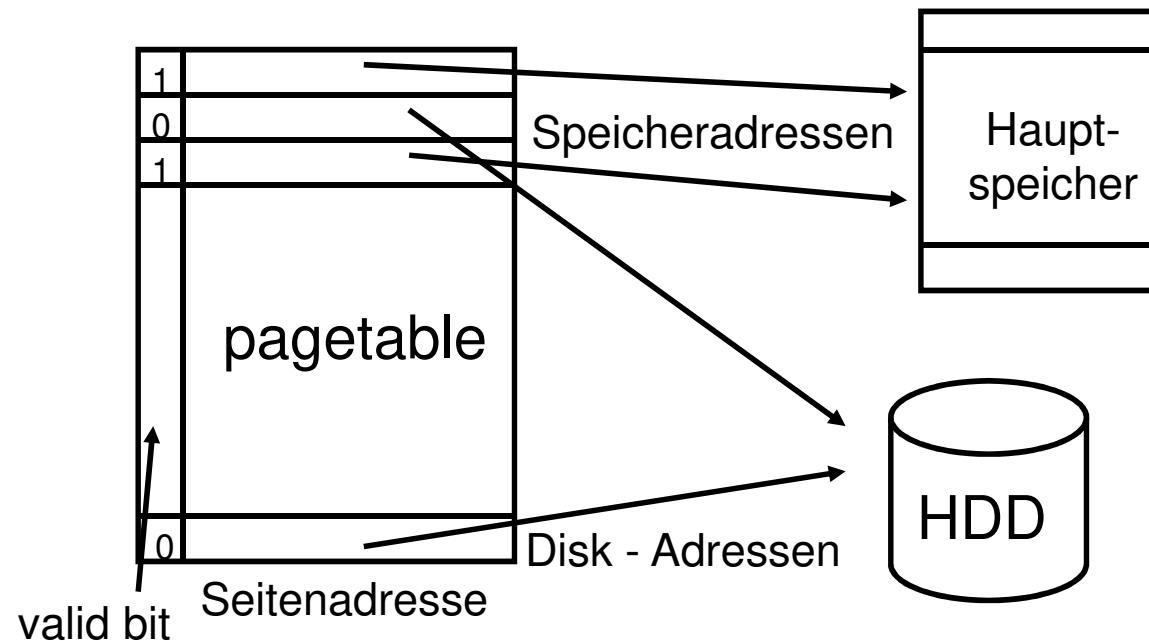
Virtuelle Adressierung ff

Zur Umsetzung gehen wir ähnlich vor wie bei Caches, d.h. wir teilen den Speicher in Seiten (pages) ein:



Virtuelle Adressierung ff

Die Umsetzung und Vergabe physischer Seiten steuert das Betriebssystem. Es muss auch Buch führen, ob eine Seite im Hauptspeicher liegt, oder ob sie zwischenzeitlich auf langsamere Massenspeicher ausgelagert wurde. Diese Umsetzung verwaltet das System in der **Pagetable**:



Virtuelle Adressierung ff

Zahlenbeispiele:

Bei einem virtuellen Adressraum vom 28 Bit (256MB) hätten wir bei

Pagegröße $2 kB \rightarrow 11$ Bit offset und 17 Bit Pageadresse, ergibt 17 Bit Adressraum zu 3 Byte = $384 kB$ Pagetable

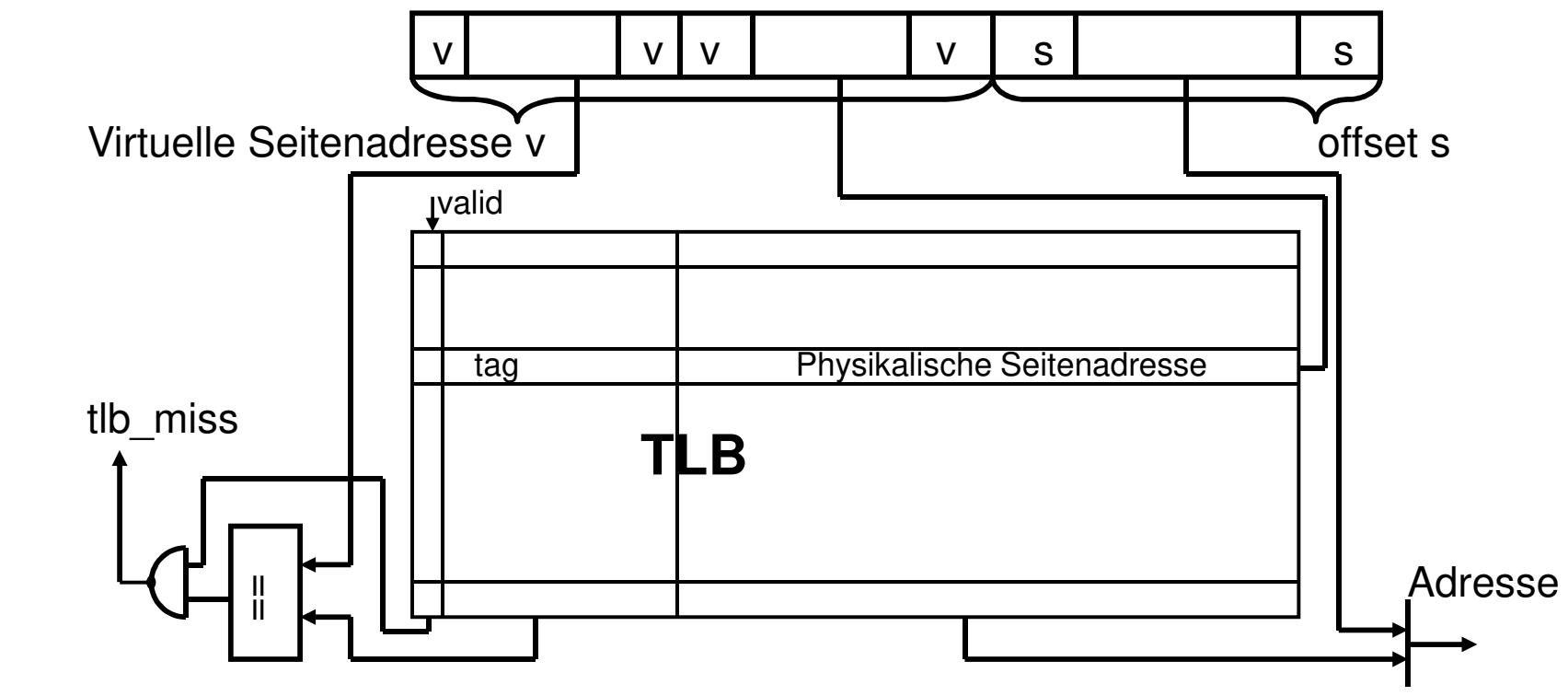
Pagegröße $16 kB \rightarrow 14$ Bit offset und 14 Bit Pageadresse, ergibt 14 Bit Adressraum zu 2 Byte = $32 kB$ Pagetable

Problem:

Nun würden Cache misses ja ewig dauern, weil zunächst das Betriebssystem (in festem Adressbereich arbeitend) aufgerufen werden müsste, um die Umsetzung nachzuschlagen. Geht das überhaupt?

Translation Look aside Buffer -- TLB

Lösung: Wir halten auch Teile der Pagetable in einem kleinen, schnellen Speicher, dem **translation look aside buffer**, in dem die Umsetzung nachgeschlagen wird. Die Zuordnung kann wie bei Caches geschehen.



Translation Look aside Buffer ff

Diese Umsetzung über den **translation look aside buffer** wird nun bei jedem Transport zwischen Cache und Hauptspeicher nachgeschlagen.

Fehlt ein gültiger Umsetzungseintrag (TLB-miss) muss die Kontrolle zunächst in der Pagetable die geforderte Umsetzung nachschlagen. Dazu kann man z.B. ein spezielles Register einführen, das **Pagetable Register**, das die (physikalische Anfangsadresse) der Pagetable des aktuell laufenden Programmes im Hauptspeicher enthält. Dieses Register darf nur im privilegierten Modus beschrieben werden.

Typische Größen für den TLB sind 32 - 1024 Einträge.

Was wir nicht geschafft haben

- CISC Architekturen, Mikroprogrammierung
- Busse
- Ein-/Ausgabegeräte
- Coprozessoren
- Pipelining
- Superskalare Architekturen
- Parallelrechner

