Digitaltechnik

5 Verifikation

Revision 1.04

Motivation

Fehlersuche mit Simulation

Event-Driven Simulation

Combinatorial Equivalence Checking

Verifikation von sequenziellen Schaltungen

Motivation

- Bugs!

Motivation

- Bugs!

Validierung stellenweise > 50 % der Entwicklungskosten!

Simulation mit vierwertiger Logik

- Verilog wurde für die Simulation erfunden!
- Verilog verwendet eine vierwertige Logik zur Simulation
- Leitung im hochohmigen Zustand hat Wert 'Z'
- Zusätzlich noch 'X' für "unbekannt"
  (z.B. Beginn der Simulation, aber auch in Vergleichen)
- Noch dazu "schwache 1" und "schwache 0"
  (betrachten wir nicht)

Semantik der vierwertigen Logik

&	**0**	**1**	**X**	**Z**
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

^	**0**	**1**	**X**	**Z**
0	0	1	X	X
1	1	0	X	X
X	X	X	X	X
Z	X	X	X	X

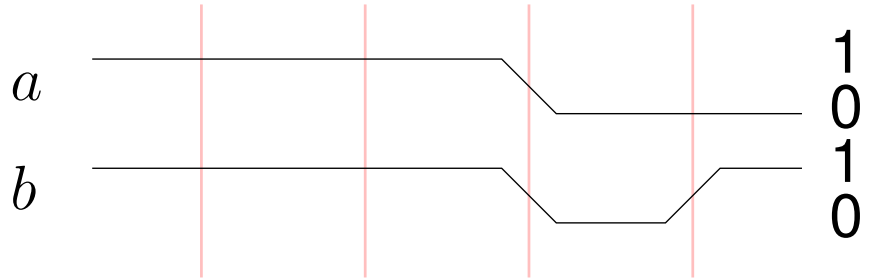
|	**0**	**1**	**X**	**Z**
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

^~	**0**	**1**	**X**	**Z**
0	1	0	X	X
1	0	1	X	X
X	X	X	X	X
Z	X	X	X	X

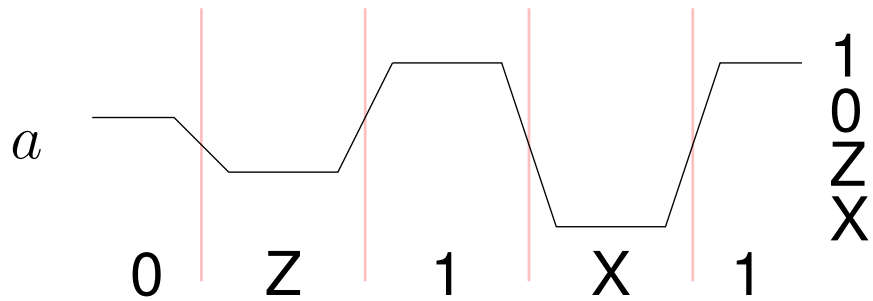
~	
0	1
1	0
X	X
Z	X

Timingdiagramme mit mehrwertiger Logik

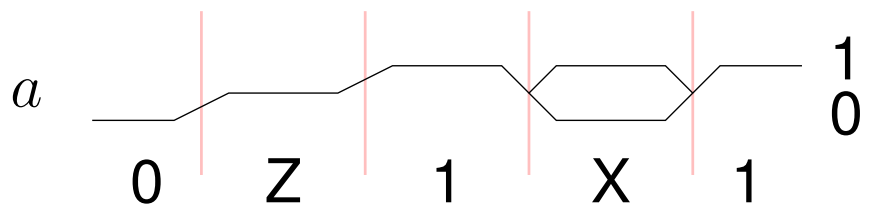
Timingdiagramm mit zwei Werten



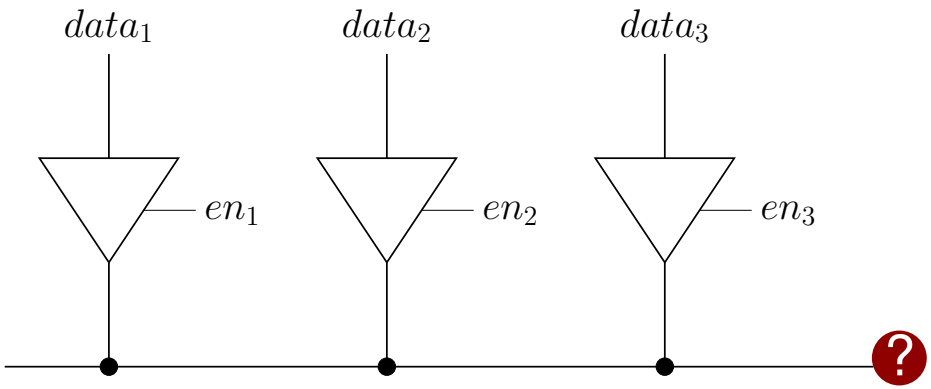
Timingdiagramm mit vier Werten



Kompaktes Timingdiagramm mit vier Werten



Resolution



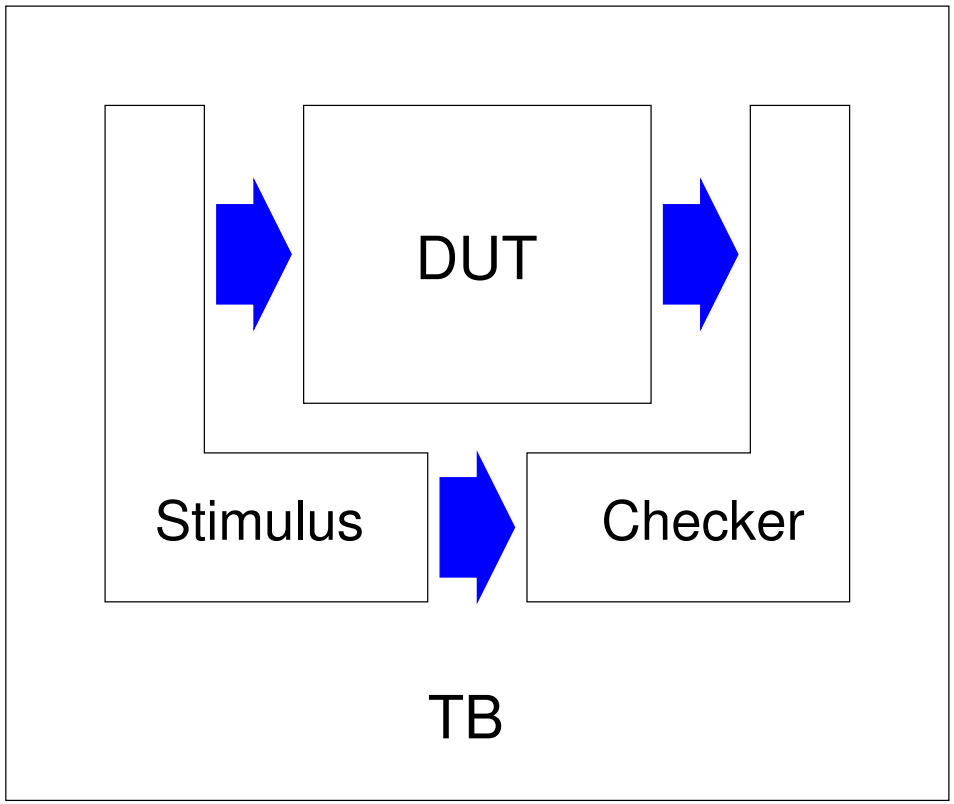
Was passiert, wenn mehrere Signale auf der selben Leitung anliegen?

	0	**1**	**X**	**Z**
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	Z

Testen durch Simulation

- Normalfall: Stimulus + Checker
- Generierung von Test Cases: Blackbox, Whitebox
- Spezifikation:
  - Cross-Checken von semantisch äquivalenten Design (z.B. RTL Spezifikation versus strukturelle/Datenfluss-Implementierung)
  - Assertions

Stimulus + DUT + Checker



DUT	Design under Test
TB	Test-Bench

Funktionale Tests vs. Timing-Simulation

Funktionale Tests

- Testen ob Eingabe / Ausgabe Verhalten der Spezifikation entspricht
- Rein qualitative Kriterien: Abstraktion von Ressourcen-Beschränkungen

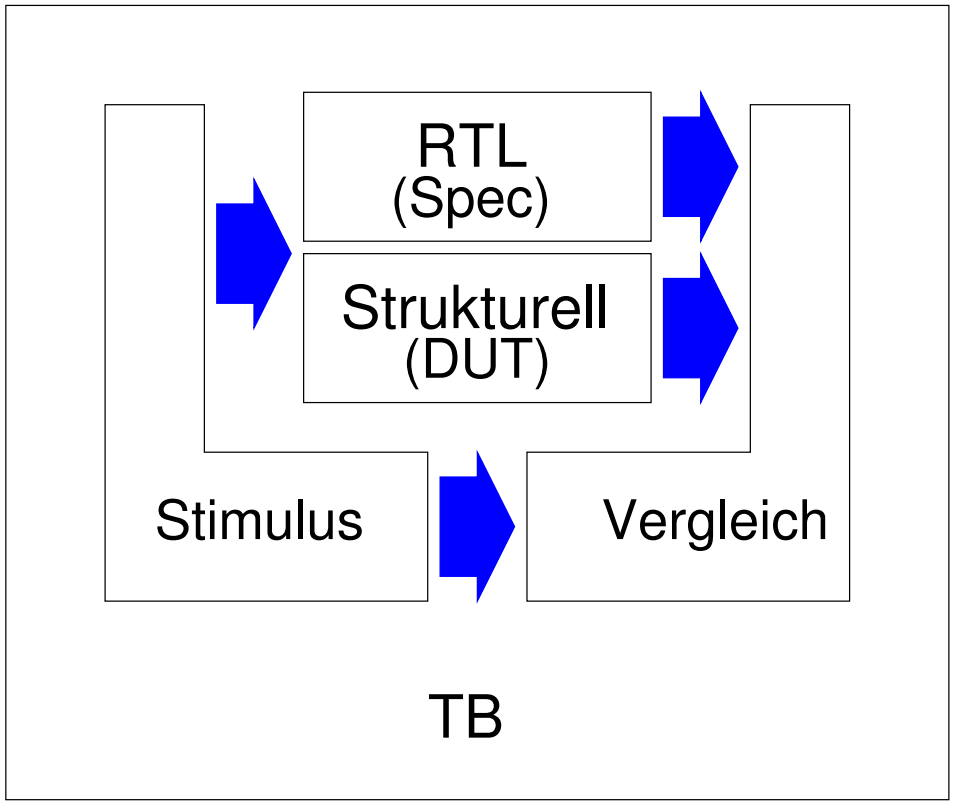
Timing Simulation

- Synthese bestimmt konkrete Verzögerungen
- Timing Simulation bestimmt Einhalten der Zeitschranken
- möglicherweise Wiederverwendung des funktionalen Stimulus

Generierung von Test Cases

- Ableiten des Stimulus und des Checkers aus der Spec (Blackbox)
- Whitebox orientiert sich an der inneren Struktur des DUT
- Ziel: Vollständige Abdeckung aller wesentlichen Fälle
- Regression: Wiederholung alter Test Cases nach jeder Änderung
- Automatisieren, Automatisieren, Automatisieren!

Spezialfall: RTL versus strukturell/Datenfluss



Vergleich der Ausgaben ergibt trivialen Checker!

Allgemein: Spezifikation und Implementierung vergleichen!

Beispiel 2-Bit Addierer: Implementierung

```
module add2impl(input [1:0] a, input [1:0] b,
                output [2:0] sum);
  wire carry1;

  assign sum[0] = a[0] ^ b[0];
  assign carry1 = a[0] & b[0];

  assign sum[1] = a[1] ^ b[1] ^ carry1;
  assign sum[2] = (a[1]&b[1])|(a[1]&carry1)|(b[1]&carry1);

endmodule
```

Wie spezifizieren wir, was der korrekte Addierer tut?

2-Bit Addierer: Testbench

```
module add2_tb();

  reg [1:0] a, b;
  wire [2:0] s;

  add2impl add(a, b, s);

  initial begin
    a=0; b=0; #1; assert(s==0);
    a=1; b=0; #1; assert(s==1);
    a=0; b=1; #1; assert(s==1);
    a=1; b=1; #1; assert(s==2);
    a=2; b=1; #1; assert(s==3);
  end

endmodule
```

Testvektoren enthalten korrekte Antwort als Assertion

2-Bit Addierer: Spezifikation mit Ausdruck

```
module main(input [1:0] a, input [1:0] b, input clk,
            output reg [2:0] x);

  wire [2:0] s;

  add2impl add(a, b, s);

  always @(posedge clk) begin
    x=s;
    assert(s==a+b);
  end

endmodule
```

✔ Trennung von Testvektoren und Spezifikation!

Dateioperationen

- Verilog enthält Befehle, mit denen man Dateien einlesen kann
- Stimulus kann dadurch extern erzeugt werden
  (z.B. durch High-Level Modell in C)
- Maschinenprogramme lassen sich so in ein Prozessor-Modell laden
  (Compiler erzeugt die Eingabedatei)
- Ausgabeformat kann selbst bestimmt werden!
- Checker muss nicht in Verilog geschrieben werden
  (z.B. Perl Programm)

Event-Driven Simulation

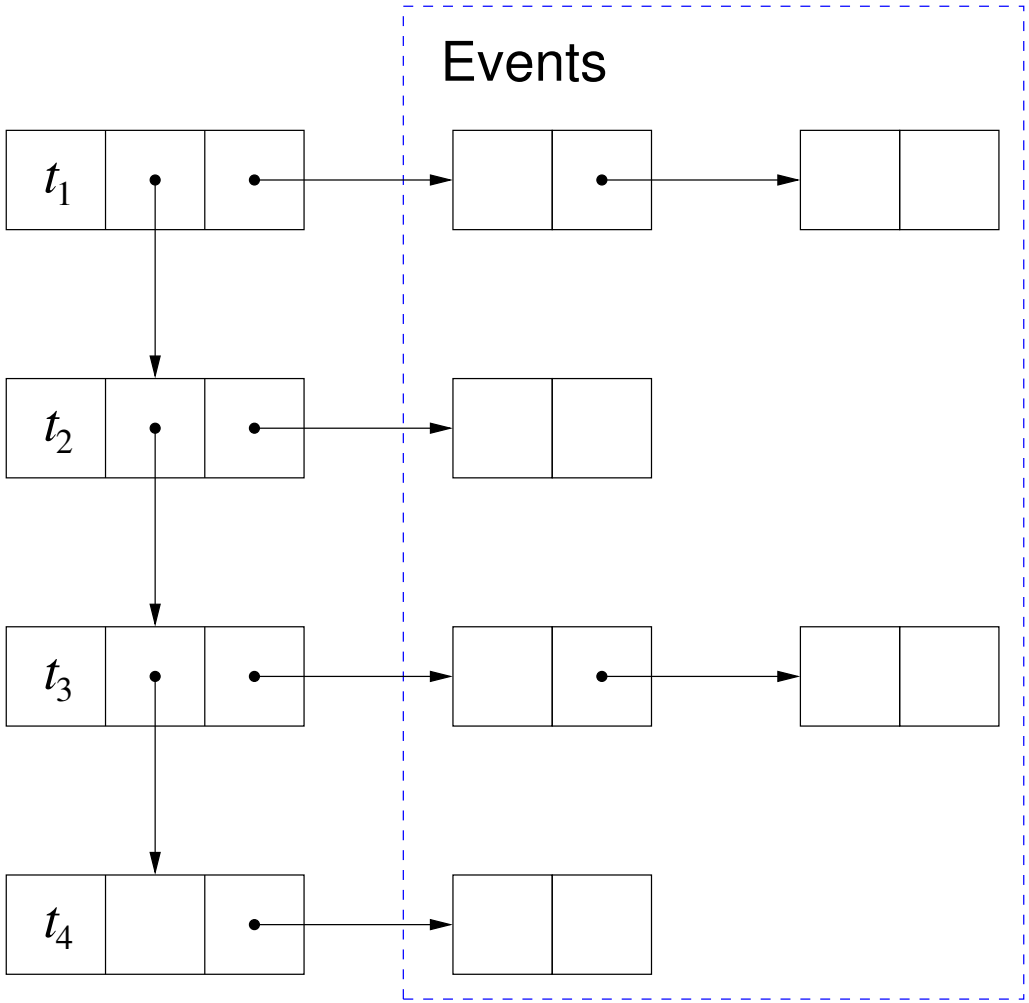
- Ziel: Minimierung des Simulationsaufwandes.
- Ausnützen der kausalen Ordnung zwischen Ereignissen!

- Änderung an beobachteten Eingängen führt zur Ausgangsänderung.
- Verzögerungen werden berechnet.

- Event: Signal plus Wert plus Zeitpunkt

Event-Queue

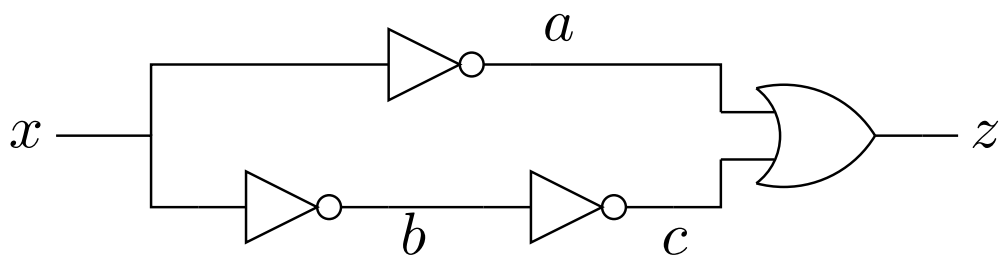
Momentaner Zeitpunkt



Single-Pass Event-Scheduler

```
1:  for e = each event at current_time do
2:      UPDATE_NODE(e);
3:
4:      for j = each gate on the fanout list of e do
5:          update input values of j;
6:          EVALUATE(j);
7:          if new_value(j) ≠ last_scheduled_value(j) then
8:              schedule new_value(j) at (current_time + delay(j));
9:              last_scheduled_value(j) := new_value(j);
10:         end if
11:     end for
12: end for
```

Beispiel (1)




```
wire x, a, b, c, z;
assign a = !x;
assign b = !x;
assign c = !b;
assign z = a | c;
```

Beispiel (2)

Von der Event-Queue wird irgendein Event geholt:

Zeit	(x, a, b, c, z)	Event-Queue	
t	(0,1,1,0,1)	(x,1,t)	Externer Event für x

Beispiel (2)

Von der Event-Queue wird irgendein Event geholt:

Zeit	(x, a, b, c, z)	Event-Queue	
t	(0,1,1,0,1)	(x,1,t)	Externer Event für x
t	(1,1,1,0,1)	(a,0,t)(b,0,t)	Generierung zweier Events

Beispiel (2)

Von der Event-Queue wird irgendein Event geholt:

Zeit	(x, a, b, c, z)	Event-Queue	
t	(0,1,1,0,1)	(x,1,t)	Externer Event für x
t	(1,1,1,0,1)	(a,0,t)(b,0,t)	Generierung zweier Events
t	(1,1,0,0,1)	(a,0,t)(c,1,t)	z.B. b Event zuerst
t	(1,1,0,1,1)	(a,0,t)	z.B. c Event zuerst

Beispiel (2)

Von der Event-Queue wird irgendein Event geholt:

Zeit	(x, a, b, c, z)	Event-Queue	
t	(0,1,1,0,1)	(x,1,t)	Externer Event für x
t	(1,1,1,0,1)	(a,0,t)(b,0,t)	Generierung zweier Events
t	(1,1,0,0,1)	(a,0,t)(c,1,t)	z.B. b Event zuerst
t	(1,1,0,1,1)	(a,0,t)	z.B. c Event zuerst
t	(1,0,0,1,1)		Schließlich a Event

(es wird kein z Event erzeugt, da sich der Wert nie ändert)

Ergebnis: Ausgang z bleibt stabil

Event-Reihenfolge

Welcher Event gewinnt bei gleichem Änderungszeitpunkt?

✘ Simulatorverhalten abhängig von der Reihenfolge

- Lösung: Künstliche Delta-Delays entlang der Kausal-Kette
  (Delta bedeutet keine echte Zeitdifferenz)

✔ Ergebnis: Standardisierung des Simulatorverhaltens!

Beachtung von Delta-Delays

Von der Event-Queue wird Event mit niedrigstem Delta-Delay geholt:

Zeit	(x, a, b, c, z)	Event-Queue	
t	(0,1,1,0,1)	(x,1,t)	Externer Event für x

Beachtung von Delta-Delays

Von der Event-Queue wird Event mit niedrigstem Delta-Delay geholt:

Zeit	(x, a, b, c, z)	Event-Queue	
t	(0,1,1,0,1)	(x,1,t)	Externer Event für x
t	(1,1,1,0,1)	(a,0,$t+\delta$)(b,0,$t+\delta$)	Generierung zweier Events

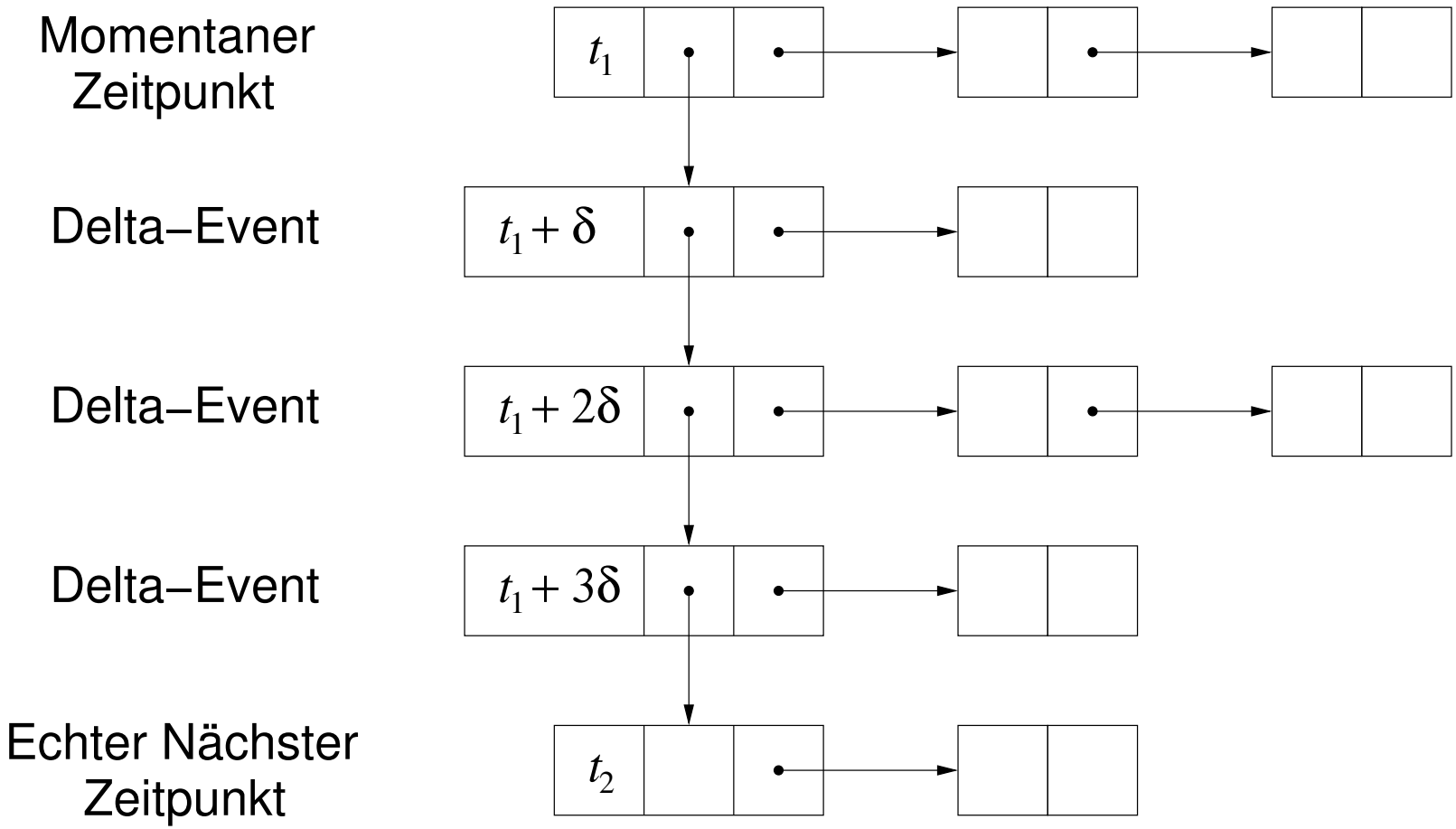
Beachtung von Delta-Delays

Von der Event-Queue wird Event mit niedrigstem Delta-Delay geholt:

Zeit	(x, a, b, c, z)	Event-Queue	
t	(0,1,1,0,1)	(x,1,t)	Externer Event für x
t	(1,1,1,0,1)	(a,0,$t+\delta$)(b,0,$t+\delta$)	Generierung zweier Events
$t+\delta$	(1,0,1,0,1)	(b,0,$t+\delta$)(z,0,$t+2\delta$)	
$t+\delta$	(1,0,0,0,1)	(z,0,$t+2\delta$)(c,1,$t+2\delta$)	
$t+2\delta$	(1,0,0,0,0)	(c,1,$t+2\delta$)	
$t+2\delta$	(1,0,0,1,0)	(z,1,$t+3\delta$)	
$t+3\delta$	(1,0,0,1,1)		

Ergebnis: 0-Impuls am Ausgang z

Event-Queue mit Delta-Delays



Fehlersuche in kombinatorischer Logik

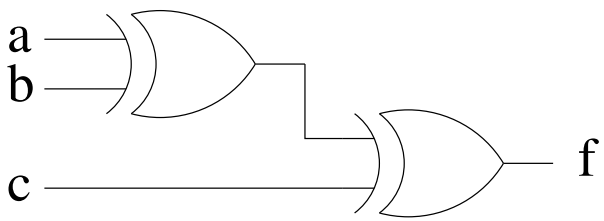
- Problem: Schaltung korrekt?

- Oft: Vergleich von komplexen, hochperformanten Schaltungen mit einfachen, aber langsamen Versionen

✘ Enumerieren von Testvektoren im Simulator wenig erfolgversprechend

- Statt dessen: Fehlersuche mit geeigneten, speziellen Algorithmen
- Schaltung als Datenstruktur darstellen
  - Operationen: Konjunktion, Disjunktion, Negation, . . .
  - Abfragen: Test auf Erfüllbarkeit, Tautologie, . . .

Implementierung als Funktionstabelle



$\longrightarrow$

a	b	c	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Implementierung als Funktionstabelle

a	b	c	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

a	b	c	g
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

a	b	c	$f \vee g$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

a	b	c	$\neg f$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Operationen einfach zeilenweise ausführen

Funktionstabellen

- Funktionstabelle ist immer 2^n Zeilen groß bei n Variablen
- Operationen sind alle linear in der Größe der Argumente:

  z.B. Konjunktion ergibt Funktionstabelle gleicher Größe
- Darstellung ist kanonisch:

  zwei äquivalente boolesche Formeln haben dieselbe Funktionstabelle
- Abfragen sind auch linear:

  Äquivalenz: überprüfe Zeilen auf Gleichheit

  Tautologie: überprüfe Zeilen auf 1

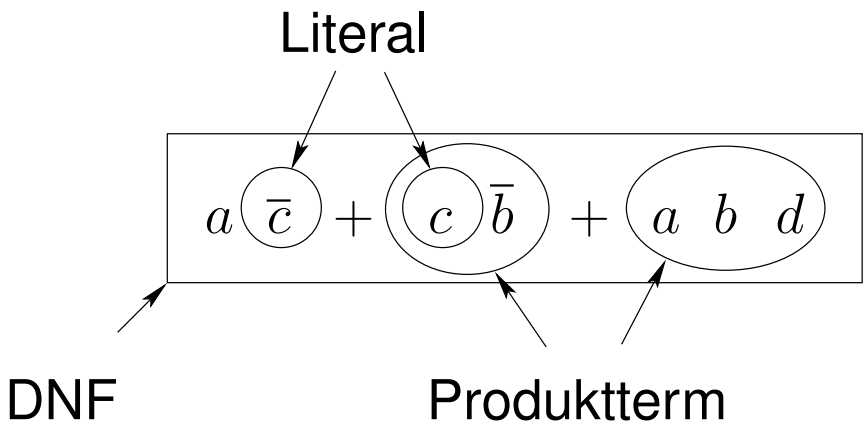
  Erfüllbarkeit: suche 1-Zeile

✘ Nicht schneller als Testvektoren

Erinnerung: Disjunktive Normalform (DNF)

- gegeben eine feste Menge von Variablen
- ein *Literal* ist eine negierte oder unnegierte Variable, z.B. x, $\overline{x}$, a, $\overline{z}$, ...
- ein *Produktterm* ist eine Konjunktion von Literalen, z.B $x \cdot y$, $\overline{b} \cdot c$, ...
- ein *Minterm* ist ein maximaler Produktterm (enthält alle Variablen)
- eine *DNF* ist eine Disjunktion von Produkttermen (wie im Beispiel)

Erinnerung: DNF-Beispiel



man bezeichnet eine DNF auch als ein Polynom

die Produktterme heißen dann Monome

DNF

- potentiell kompakter als Funktionstabelle

  nur Anzahl Kernprimimplikanten bestimmt Größe

- unmittelbare Implementierung als minimale 2-stufige Schaltung (PLA)

- Disjunktion linear (ohne Minimierung)

- Konjunktion polynomiell und Negation exponentiell (auch ohne Minimierung)

- Erfüllbarkeit mit konstantem Aufwand:

  DNF erfüllbar gdw. mind. ein Monom vorhanden

Konjunktion von DNF

$$\underbrace{(\, a \cdot \overline{b} \quad \vee \quad \overline{a} \cdot b \cdot \overline{c} \,)}_{\text{1. Operand}} \quad \wedge \quad \underbrace{(\, a \cdot \overline{b} \quad \vee \quad \overline{b} \cdot \overline{c} \,)}_{\text{2. Operand}}$$

Ausmultiplizieren

$$a \cdot \overline{b} \;\cdot\; a \cdot \overline{b} \quad \vee \quad a \cdot \overline{b} \;\cdot\; \overline{b} \cdot \overline{c} \quad \vee \quad \overline{a} \cdot b \cdot \overline{c} \;\cdot\; a \cdot \overline{b} \quad \vee \quad \overline{a} \cdot b \cdot \overline{c} \;\cdot\; \overline{b} \cdot \overline{c}$$

Vereinfachen der einzelnen Min-Terme

$$a \cdot \overline{b} \quad \vee \quad a \cdot \overline{b} \cdot \overline{c}$$

Minimierung (z.B. mit Quine-McCluskey)

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

Konjunktion von DNF

$$\underbrace{(a \vee b \vee c)}_{\text{1. Operand}} \wedge \underbrace{(d \vee e \vee f)}_{\text{2. Operand}}$$

Ausmultiplizieren

$$a \cdot d \vee a \cdot e \vee a \cdot f \vee b \cdot d \vee b \cdot e \vee b \cdot f \vee c \cdot d \vee c \cdot e \vee c \cdot f$$

Keine weitere Vereinfachung möglich!

Beispiel lässt sich leicht generalisieren:

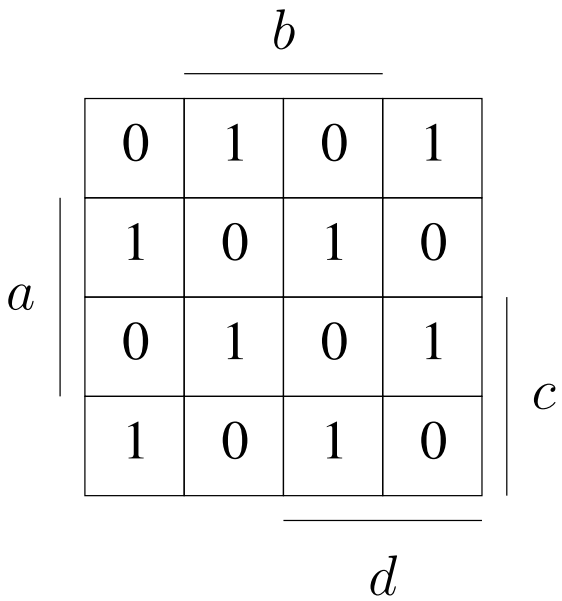
Konjunktion zweier DNF mit n und mit m Monomen hat $O(n \cdot m)$ Monome

Konjunktion von DNF

- maximal polynomiell größeres Ergebnis (in beiden Argumenten)
- man verliert Minimalität:

  Ausmultiplizieren min. DNF ergibt nicht notwendigerweise min. DNF
- anschließende Minimierung hätte wiederum exponentiellen Aufwand
- auch implizite Generierung der Monome der Konjunktion möglich

Exponentielle DNF für Parity



$$a \oplus b \oplus c \oplus d$$

- ✘ keine Zusammenfassung von Min-Termen im KV-Diagramm möglich
- nur *volle* Min-Terme als Primimplikanten (bestehend aus maximaler Anzahl von Literalen)
- DNF für Parity von n Variablen hat 2^{n-1} Monome

Konjunktive Normalform (KNF, CNF)

- Dual zu DNF.

- wie bei DNF: feste Menge von Variablen
- wie bei DNF: *Literal* ist eine negierte oder unnegierte Variable, z.B. x, $\overline{x}$, a, $\overline{z}$, $\ldots$

- ein *Disjunktionsterm* (Klausel) ist eine Disjunktion von Literalen, z.B. $x \vee y$, $\overline{b} \vee c$, $\ldots$

- eine *KNF* ist eine Konjunktion von Disjunktionstermen:

$$(a \vee \overline{b}) \wedge (a \vee \overline{c}) \wedge (\overline{a} \vee b \vee c)$$

SAT (Satisfiability = Erfüllbarkeit) ist das „klassische“ NP-vollständige Problem:

Gegeben KNF f mit n Variablen V:
Gibt es eine Zuweisung $\sigma : V \to \{0, 1\}$ mit $\sigma(f) = 1$?

„NP-vollständig“ $\rightarrow$ theoretische Informatik, Komplexitätstheorie

Wichtig für uns:
Obwohl SAT Algorithmen schlimmstenfalls exponentielle Laufzeit haben, existieren viele schnelle Implementierungen

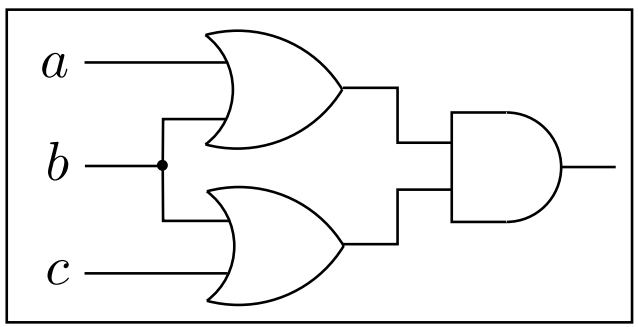
- Beispiele: zChaff, BerkMin, MiniSat, ...
- Standardtool in der Industrie

Transformation eines Schaltkreises in KNF

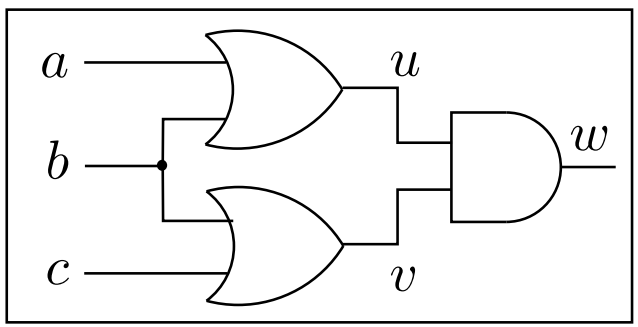
- Analog wie bei DNF:
  - Konjunktion ist $O(n + m)$
  - Disjunktion ist $O(n \cdot m)$
  - Negation ist exponentiell
  - Parity ist exponentiell

✘ Auch keine Lösung

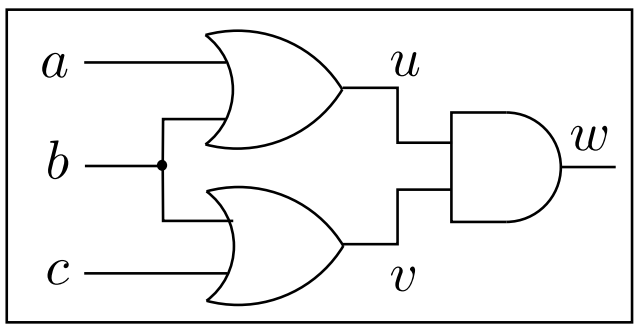
KNF



KNF



KNF



Klauseln:

$$
\begin{aligned}
&(u \leftrightarrow a \vee b) \wedge \\
&(v \leftrightarrow b \vee c) \wedge \\
&(w \leftrightarrow u \wedge v) \wedge \\
&w
\end{aligned}
$$

Tseitin Transformation: Input / Output Constraints

Negation:

$$\begin{aligned} x \leftrightarrow \overline{y} &\Leftrightarrow (x \rightarrow \overline{y}) \wedge (\overline{y} \rightarrow x) \\ &\Leftrightarrow (\overline{x} \vee \overline{y}) \wedge (y \vee x) \end{aligned}$$

Disjunktion:

$$\begin{aligned} x \leftrightarrow (y \vee z) &\Leftrightarrow (y \rightarrow x) \wedge (z \rightarrow x) \wedge (x \rightarrow (y \vee z)) \\ &\Leftrightarrow (\overline{y} \vee x) \wedge (\overline{z} \vee x) \wedge (\overline{x} \vee y \vee z) \end{aligned}$$

Konjunktion:

$$\begin{aligned} x \leftrightarrow (y \wedge z) &\Leftrightarrow (x \rightarrow y) \wedge (x \rightarrow z) \wedge (\overline{(y \wedge z)} \rightarrow x) \\ &\Leftrightarrow (\overline{x} \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{(y \wedge z)} \vee x) \\ &\Leftrightarrow (\overline{x} \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{y} \vee \overline{z} \vee x) \end{aligned}$$

Äquivalenz:

$$\begin{aligned} x \leftrightarrow (y \leftrightarrow z) &\Leftrightarrow (x \rightarrow (y \leftrightarrow z)) \wedge ((y \leftrightarrow z) \rightarrow x) \\ &\Leftrightarrow (x \rightarrow ((y \rightarrow z) \wedge (z \rightarrow y)) \wedge ((y \leftrightarrow z) \rightarrow x) \\ &\Leftrightarrow (x \rightarrow (y \rightarrow z)) \wedge (x \rightarrow (z \rightarrow y)) \wedge ((y \leftrightarrow z) \rightarrow x) \\ &\Leftrightarrow (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge ((y \leftrightarrow z) \rightarrow x) \\ &\Leftrightarrow (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge (((y \wedge z) \vee (\overline{y} \wedge \overline{z})) \rightarrow x) \\ &\Leftrightarrow (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge ((y \wedge z) \rightarrow x) \wedge ((\overline{y} \wedge \overline{z}) \rightarrow x) \\ &\Leftrightarrow (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge (\overline{y} \vee \overline{z} \vee x) \wedge (y \vee z \vee x) \end{aligned}$$

Herleitung mit Funktionstabelle

x	y	z	$x \leftrightarrow (y \wedge z)$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Herleitung mit Funktionstabelle

x	y	z	$x \leftrightarrow (y \wedge z)$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

$\rightarrow$

x	y	z	$x \leftrightarrow (y \wedge z)$
0	0	$*$	1
0	$*$	0	1
0	1	1	0
1	0	$*$	0
1	$*$	0	0
1	1	1	1

Herleitung mit Funktionstabelle

x	y	z	$x \leftrightarrow (y \wedge z)$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

$\rightarrow$

x	y	z	$x \leftrightarrow (y \wedge z)$	
0	0	$*$	1	
0	$*$	0	1	
0	1	1	0	$x \vee \overline{y} \vee \overline{z}$
1	0	$*$	0	$\overline{x} \vee y$
1	$*$	0	0	$\overline{x} \vee z$
1	1	1	1	

Ablesen von KNF aus Funktionstabelle: Zeilen mit 0 betrachten!

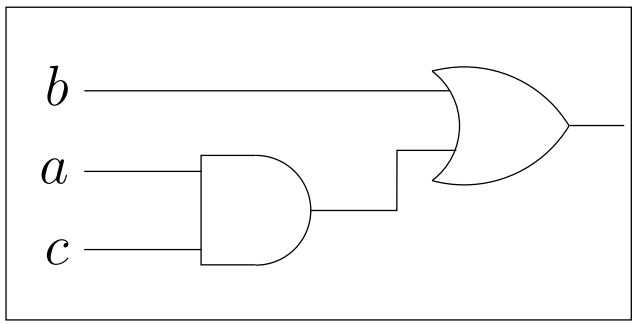
Zusammenfassung Tseitin Transformation

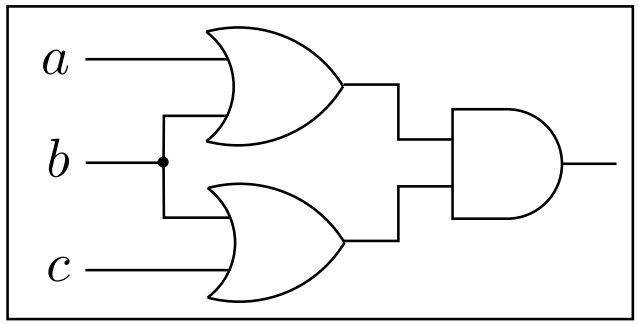
1. Erzeuge neue Variable für jedes Signal, das nicht schon Input ist
2. Erzeuge Klauseln für jedes Gatter
3. Alle Klauseln als Konjunktion zusammenfassen $\rightarrow$ KNF

Die Transformation erhält Erfüllbarkeit:
✔ das Ergebnis der Transformation ist erfüllbar g.d.w. die ursprüngliche Formel erfüllbar ist.

Die Formeln sind nicht äquivalent, da die transformierte Formel neue, zusätzliche Variablen enthält.

SAT Beispiel: Äquivalenz von kombinatorischen Schaltungen



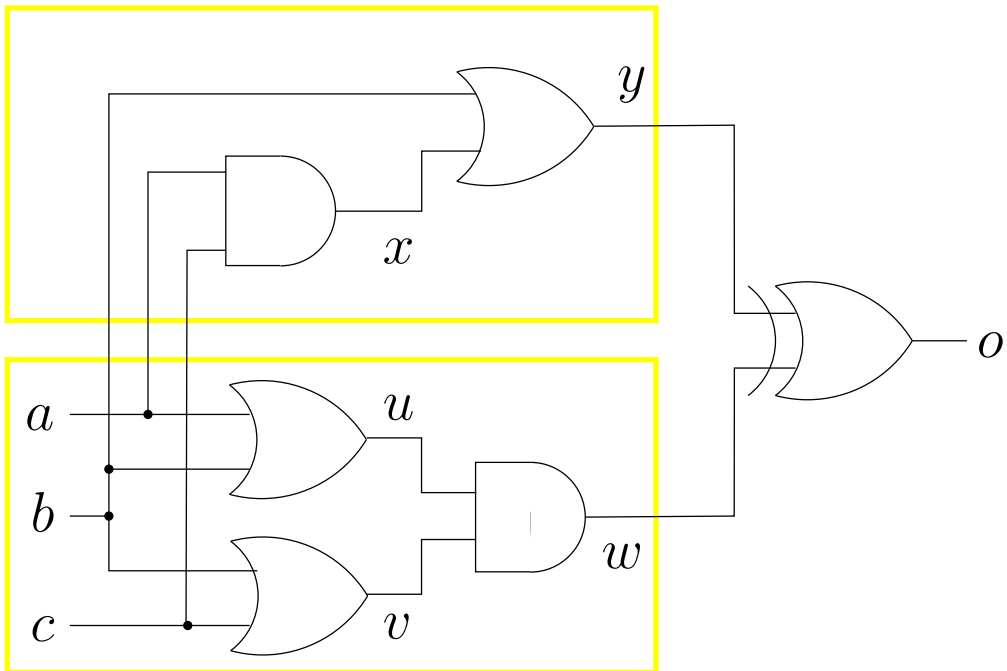


$b \vee a \wedge c$ $\qquad$ $(a \vee b) \wedge (b \vee c)$

äquivalent?

$b \vee a \wedge c \quad \Leftrightarrow \quad (a \vee b) \wedge (b \vee c)$

KNF



$$
\begin{aligned}
& o \ \wedge \\
(x \ \leftrightarrow \ & a \wedge c) \wedge \\
(y \ \leftrightarrow \ & b \vee x) \wedge \\
(u \ \leftrightarrow \ & a \vee b) \wedge \\
(v \ \leftrightarrow \ & b \vee c) \wedge \\
(w \ \leftrightarrow \ & u \wedge v) \wedge \\
(o \ \leftrightarrow \ & y \oplus w)
\end{aligned}
$$

$$o \wedge (x \rightarrow a) \wedge (x \rightarrow c) \wedge (x \leftarrow a \wedge c) \wedge \ \ldots$$

DIMACS KNF Darstellung

- DIMACS CNF Format = Standardformat für KNF
- Wird von allen üblichen SAT-Solvern verwendet
- Textdatei mit folgendem Aufbau:
  `p cnf` <# Variablen> <# Klauseln>
  <Klausel> 0
  <Klausel> 0
  ...
- Eine (oder mehrere) Zeilen pro Klausel

DIMACS KNF Darstellung

- Jede Klausel ist eine Liste von Zahlen (getrennt mit Leerzeichen)
- Eine Klausel endet mit 0

- Jede Zahl $1, 2, \ldots$ entspricht einer Variable

$\rightarrow$ Namen (z.B. a, b, $\ldots$) müssen auf Zahlen abgebildet werden

- Eine negative Zahl entspricht einer Negation

$\rightarrow$ Wenn a die Nummer 5 hat, dann entspricht $\overline{a}$ der -5

DIMACS KNF Darstellung

- Beispiel:
  $(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_2 \vee \overline{x_1}) \wedge (x_4 \vee \overline{x_2} \vee \overline{x_1})$

- 4 Variablen, 3 Klauseln

- KNF-Datei:

```
p cnf 4 3
1 2 -3 0
2 -1 0
4 -2 -1 0
```

SAT Beispiel: Äquivalenz von kombinatorischen Schaltungen

Formel:

$$
\begin{aligned}
& o \ \wedge \\
(x \ \leftrightarrow \ & a \wedge c) \wedge \\
(y \ \leftrightarrow \ & b \vee x) \wedge \\
(u \ \leftrightarrow \ & a \vee b) \wedge \\
(v \ \leftrightarrow \ & b \vee c) \wedge \\
(w \ \leftrightarrow \ & u \wedge v) \wedge \\
(o \ \leftrightarrow \ & y \oplus w)
\end{aligned}
$$

Variablenzuordnung:

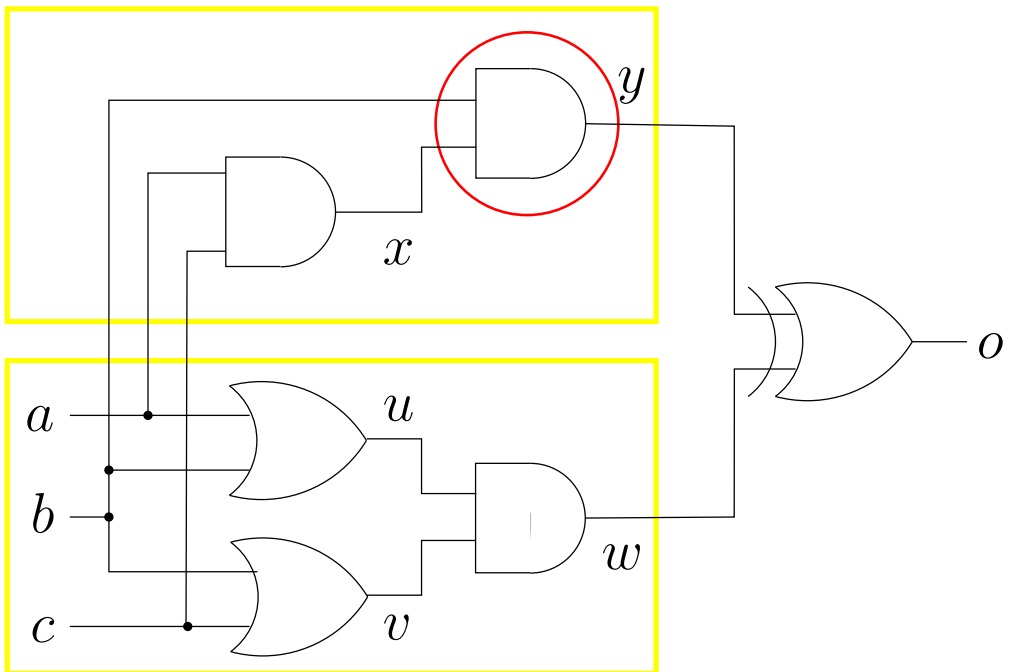
Variable	Nummer
o	1
a	2
c	3
x	4
b	5
y	6
u	7
v	8
w	9

In Reihenfolge des Erscheinens.

SAT Beispiel: Äquivalenz von kombinatorischen Schaltungen

Formel	Klauseln	DIMACS
o	o	`1 0`
$x \leftrightarrow a \wedge c$	$a \vee \overline{x}$	`2 -4 0`
	$c \vee \overline{x}$	`3 -4 0`
	$\overline{a} \vee \overline{c} \vee x$	`-2 -3 4 0`
$y \leftrightarrow b \vee x$	$\overline{x} \vee y$	`-4 6 0`
	$\overline{b} \vee y$	`-5 6 0`
	$x \vee b \vee \overline{y}$	`4 5 -6 0`
$u \leftrightarrow a \vee b$	$\overline{a} \vee u$	`-2 7 0`
	$\overline{b} \vee u$	`-5 7 0`
	$a \vee b \vee \overline{u}$	`2 5 -7 0`
$v \leftrightarrow b \vee c$	$\overline{b} \vee v$	`-5 8 0`
	$\overline{c} \vee v$	`-3 8 0`
	$b \vee c \vee \overline{v}$	`5 3 -8 0`
$w \leftrightarrow u \wedge v$	$u \vee \overline{w}$	`7 -9 0`
	$v \vee \overline{w}$	`8 -9 0`
	$\overline{u} \vee \overline{v} \vee w$	`-7 -8 9 0`
$o \leftrightarrow y \oplus w$	$\overline{y} \vee \overline{w} \vee \overline{o}$	`-6 -9 -1 0`
	$y \vee w \vee \overline{o}$	`6 9 -1 0`
	$\overline{y} \vee w \vee o$	`-6 9 1 0`
	$y \vee \overline{w} \vee o$	`6 -9 1 0`

Beispiel von vorhin geringfügig geändert:



$$
\begin{aligned}
& o \ \wedge \\
(x \ \leftrightarrow \ & a \wedge c) \ \wedge \\
(y \ \leftrightarrow \ & b \wedge x) \ \wedge \\
(u \ \leftrightarrow \ & a \vee b) \ \wedge \\
(v \ \leftrightarrow \ & b \vee c) \ \wedge \\
(w \ \leftrightarrow \ & u \wedge v) \ \wedge \\
(o \ \leftrightarrow \ & y \oplus w)
\end{aligned}
$$

KNF erfüllbar?

Ergebnis SAT Beispiel

- Ausgabe des SAT-Solvers:

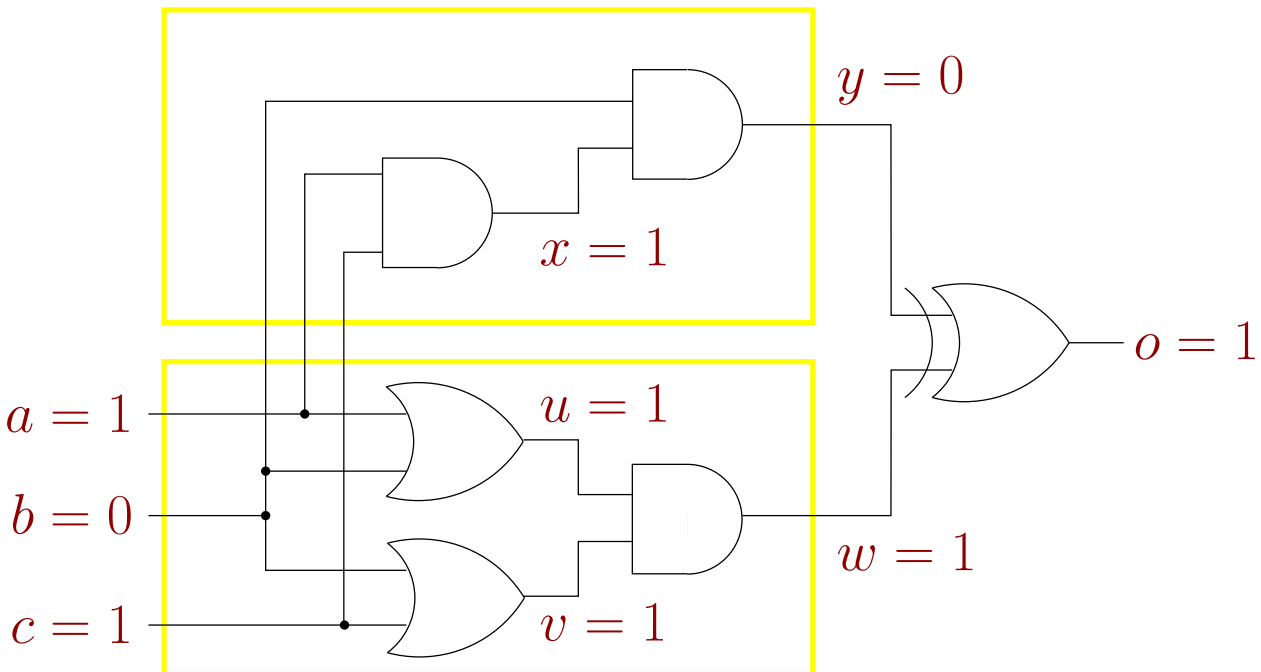
```
SATISFIABLE
1 2 3 4 -5 -6 7 8 9
```

- Werte der Variablen:

Variable	Nummer	Wert
o	1	1
a	2	1
c	3	1
x	4	1
b	5	0
y	6	0
u	7	1
v	8	1
w	9	1

- Achtung: Es gibt mehrere richtige Lösungen

Erfüllende Zuweisung im Schaltkreis:



Variable	Wert
o	1
a	1
c	1
x	1
b	0
y	0
u	1
v	1
w	1

Größere Beispiele: C++ Klasse

```
class knft
{
public:
  typedef int literalt;

  void gate_and(literalt a, literalt b, literalt o);
  void gate_or(literalt a, literalt b, literalt o);
  void gate_xor(literalt a, literalt b, literalt o);
  void gate_nand(literalt a, literalt b, literalt o);
  void gate_nor(literalt a, literalt b, literalt o);

  literalt gate_and(literalt a, literalt b);
  literalt gate_or(literalt a, literalt b);
  ...
  literalt neue_variable();

  literalt pos(literalt a) { return a; }
  literalt neg(literalt a) { return -a; }
  ...
```

Größere Beispiele: C++ Klasse

```
    ...
    unsigned anzahl_variablen;

    typedef std::list<literalt> klauselt;
    typedef std::list<klauselt> klausel_listet;

    klausel_listet klausel_liste;

    void dimacs(std::ostream &out);
};
```

Beispiel

```
knft::literalt carry(
  knft &knf,
  knft::literalt a, knft::literalt b, knft::literalt c)
{
  knft::literalt s1=knf.gate_and(a, b);
  knft::literalt s2=knf.gate_and(a, c);
  knft::literalt s3=knf.gate_and(b, c);

  return knf.gate_or(s1, knf.gate_or(s2, s3));
}
```

Beispiel

```
void rca(
  knft &knf, unsigned n,
  knft::literalt a[], knft::literalt b[],
  knft::literalt c, knft::literalt &cout,
  knft::literalt sum[])
{
  for(unsigned i=0; i<n; i++)
  {
    sum[i]=knf.gate_xor(
             knf.gate_xor(a[i], b[i]),
             c);

    c=carry(knf, a[i], b[i], c);
  }

  cout=c;
}
```

SAT für sequenzielle Schaltungen

- Bisher nur kombinatorische Schaltungen
- Fehler in Schaltungen mit Registern aber viel schwieriger zu finden!

- Gegeben:
  1. Schaltung mit Registern
  2. Testbench mit `assert` o.ä.

- Können wir SAT verwenden, um Fehler im Schaltkreis zu finden?

$\Longrightarrow$ Übung

Erinnerung:

Definition (Transitionssystem)

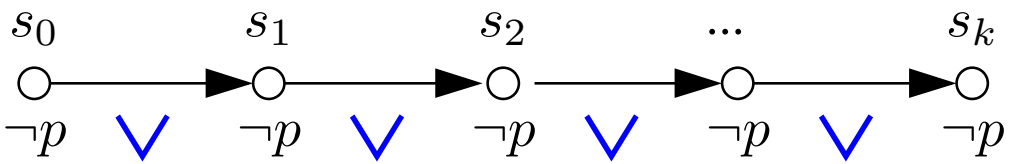
Ein Transitionssystem ist ein Tripel $\langle S, I, T \rangle$ mit

- S: Menge der Zustände
- $I \subseteq S$: Initialzustände
- $T : S \times S$: Übergangsrelation

Assertion sei p (Prädikat über S)

$\Rightarrow$ Suche nach Zustand mit $\neg p$

Idee: begrenze Tiefe auf k Taktzyklen



$$I(s_0) \wedge T(s_0, s_1) \wedge \cdots \wedge T(s_{k-1}, s_k) \wedge \bigvee_{i=0}^{k} \neg p(s_i)$$

Beispiel

$$T \subseteq \mathbb{N}_0 \times \mathbb{N}_0$$

$$T(s, s') \iff s'.x = s.x + 1$$

Und: $I(s) \iff s.x = 0 \vee s.x = 1$

Beispiel

$$T \subseteq \mathbb{N}_0 \times \mathbb{N}_0$$

$$T(s, s') \iff s'.x = s.x + 1$$

Und: $I(s) \iff s.x = 0 \vee s.x = 1$

Mit $k = 4$:

$$\begin{aligned} & (s_0.x = 0 \vee s_0.x = 1) \\ \wedge \quad & s_1.x = s_0.x + 1 \\ \wedge \quad & s_2.x = s_1.x + 1 \\ \wedge \quad & s_3.x = s_2.x + 1 \\ \wedge \quad & s_4.x = s_3.x + 1 \end{aligned}$$

(noch ohne Eigenschaft)

Beispiel

Die Assertion sei $x \leq 20$.

Mit $k = 4$:

$$
\begin{aligned}
& (s_0.x = 0 \vee s_0.x = 1) \\
\wedge \quad & s_1.x = s_0.x + 1 \\
\wedge \quad & s_2.x = s_1.x + 1 \\
\wedge \quad & s_3.x = s_2.x + 1 \\
\wedge \quad & s_4.x = s_3.x + 1 \\
\wedge \quad & (s_0.x > 20 \vee s_1.x > 20 \vee s_2.x > 20 \vee s_3.x > 20 \vee s_4.x > 20)
\end{aligned}
$$

SAT in der Praxis: BMC für Verilog

1. Verilog wird synthetisiert: Netzliste aus Gattern
2. Netzliste wird in KNF übersetzt
3. SAT findet Fehler in KNF
4. Ergebnis des SAT-Solvers (erfüllende Zuweisung) wird in Testbench übersetzt
5. Testbench wird in Simulator (z.B. ModelSim) visualisiert

Bekannte Tools: Cadence SMV, ebmc