

Digitaltechnik

5 Verifikation

Revision 1.04

Motivation

Fehlersuche mit Simulation

Event-Driven Simulation

Combinatorial Equivalence Checking

Verifikation von sequenziellen Schaltungen

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

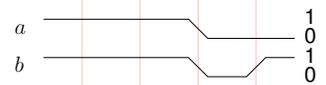
	0	1	X	Z
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

~	0	1	X	Z
0	1	0	X	X
1	0	1	X	X
X	X	X	X	X
Z	X	X	X	X

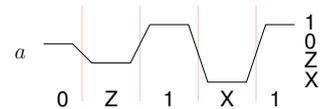
-	
0	1
1	0
X	X
Z	X

Timingdiagramme mit mehrwertiger Logik

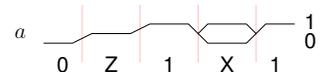
Timingdiagramm mit zwei Werten



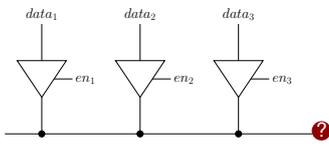
Timingdiagramm mit vier Werten



Kompaktes Timingdiagramm mit vier Werten



Resolution



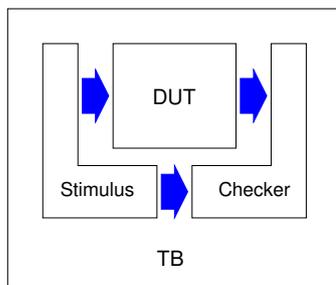
Was passiert, wenn mehrere Signale auf der selben Leitung anliegen?

	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

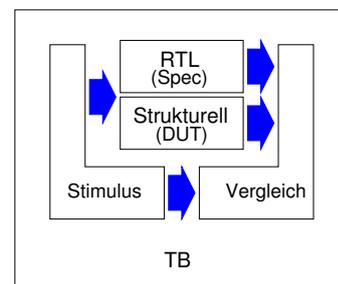
Timing Simulation

- ▶ 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;
5   assign sum[0] = a[0] ^ b[0];
    assign carry1 = a[0] & b[0];
10  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;
5   add2impl add(a, b, s);

    initial begin
        a=0; b=0; #1; assert(s==0);
10  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
15 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;
5   add2impl add(a, b, s);

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

endmodule

```

✓ Trennung von Testvektoren und Spezifikation!

Dateioperationen

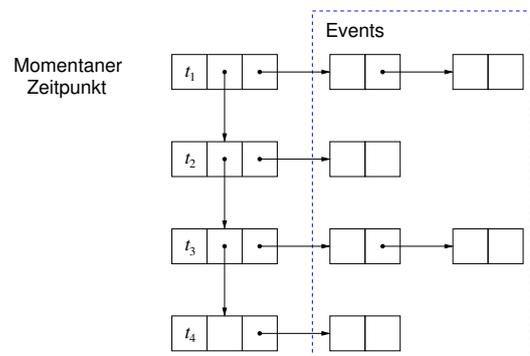
- ▶ Verilog enthält Befehle, mit denen man Dateien einlesen kann
- ▶ Stimulus kann dadurch extern erzeugt werden (z.B. durch High-Level Modell in C)
- ▶ Maschinenprogramme lassen sich so in ein Prozessor-Modell laden (Compiler erzeugt die Eingabedatei)
- ▶ Ausgabeformat kann selbst bestimmt werden!
-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

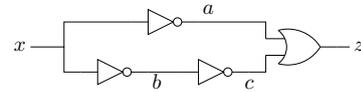


Single-Pass Event-Scheduler

```

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
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!

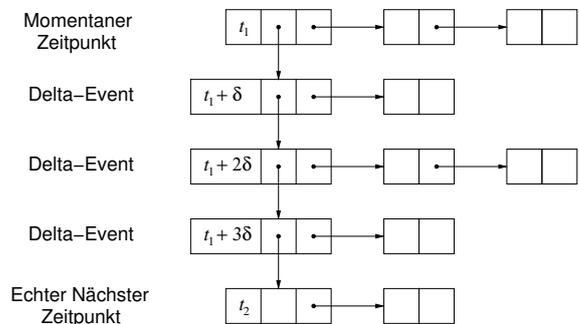
Beachtung von Delta-Delays

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
t	(1,1,1,0,1)	(a,0,t+δ)(b,0,t+δ)	Generierung zweier Events
t+δ	(1,0,1,0,1)	(b,0,t+δ)(z,0,t+2δ)	
t+δ	(1,0,0,0,1)	(z,0,t+2δ)(c,1,t+2δ)	
t+2δ	(1,0,0,0,0)	(c,1,t+2δ)	
t+2δ	(1,0,0,1,0)	(z,1,t+3δ)	
t+3δ	(1,0,0,1,1)		

Ergebnis: 0-Impuls am Ausgang z

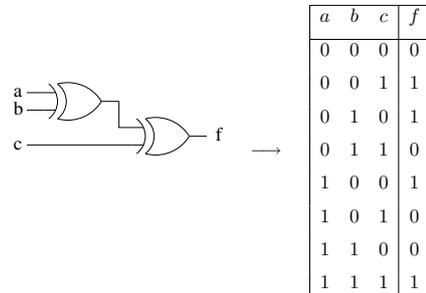
Event-Queue mit Delta-Delays



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



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 ∨ 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	¬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

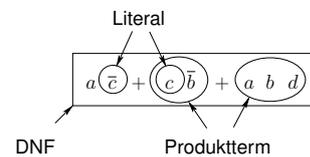
Funktionstabellen

- ▶ 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. $x, \bar{x}, a, \bar{a}, \dots$
- ▶ ein *Produktterm* ist eine Konjunktion von Literalen, z.B. $x \cdot y, \bar{b} \cdot c, \dots$
- ▶ 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

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

Ausmultiplizieren

$$a \cdot \bar{b} \cdot a \cdot \bar{b} \vee a \cdot \bar{b} \cdot \bar{b} \cdot \bar{c} \vee \bar{a} \cdot b \cdot \bar{c} \cdot a \cdot \bar{b} \vee \bar{a} \cdot b \cdot \bar{c} \cdot \bar{b} \cdot \bar{c}$$

Vereinfachen der einzelnen Min-Terme

$$a \cdot \bar{b} \vee a \cdot \bar{b} \cdot \bar{c}$$

Minimierung (z.B. mit Quine-McCluskey)

$$a \cdot \bar{b}$$

Konjunktion von DNF

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

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

	\bar{b}			
	0	1	0	1
a	1	0	1	0
	0	1	0	1
	1	0	1	0
	1	0	1	0
	d			

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

- ✗ 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. $x, \bar{x}, a, \bar{a}, \dots$
- ▶ ein *Disjunktionsterm* (Klausel) ist eine Disjunktion von Literalen, z.B. $x \vee y, \bar{b} \vee c, \dots$
- ▶ eine *KNF* ist eine Konjunktion von Disjunktionstermen:
$$(a \vee \bar{b}) \wedge (a \vee \bar{c}) \wedge (\bar{a} \vee b \vee c)$$

SAT

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

Gegeben KNF f mit n Variablen V :

Gibt es eine Zuweisung $\sigma : V \rightarrow \{0, 1\}$ 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

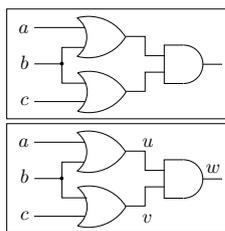
▶ Analog wie bei DNF:

- ▶ Konjunktion ist $O(n + m)$
- ▶ Disjunktion ist $O(n \cdot m)$
- ▶ Negation ist exponentiell
- ▶ Parity ist exponentiell

✗ Auch keine Lösung

Tsetin Transformation

KNF



Klauseln:

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

Tsetin Transformation: Input / Output Constraints

Negation: $x \leftrightarrow \bar{y} \Leftrightarrow (x \rightarrow \bar{y}) \wedge (\bar{y} \rightarrow x)$
 $\Leftrightarrow (\bar{x} \vee \bar{y}) \wedge (y \vee x)$

Disjunktion: $x \leftrightarrow (y \vee z) \Leftrightarrow (y \rightarrow x) \wedge (z \rightarrow x) \wedge (x \rightarrow (y \vee z))$
 $\Leftrightarrow (\bar{y} \vee x) \wedge (\bar{z} \vee x) \wedge (x \vee y \vee z)$

Konjunktion: $x \leftrightarrow (y \wedge z) \Leftrightarrow (x \rightarrow y) \wedge (x \rightarrow z) \wedge ((y \wedge z) \rightarrow x)$
 $\Leftrightarrow (\bar{x} \vee y) \wedge (\bar{x} \vee z) \wedge ((y \wedge z) \vee x)$
 $\Leftrightarrow (\bar{x} \vee y) \wedge (\bar{x} \vee z) \wedge (\bar{y} \vee \bar{z} \vee x)$

Äquivalenz: $x \leftrightarrow (y \leftrightarrow z) \Leftrightarrow (x \rightarrow (y \leftrightarrow z)) \wedge ((y \leftrightarrow z) \rightarrow x)$
 $\Leftrightarrow (x \rightarrow ((y \rightarrow z) \wedge (z \rightarrow y))) \wedge (((y \leftrightarrow z) \rightarrow x)$
 $\Leftrightarrow (x \rightarrow ((y \rightarrow z) \wedge (x \rightarrow (z \rightarrow y)))) \wedge ((y \leftrightarrow z) \rightarrow x)$
 $\Leftrightarrow (\bar{x} \vee \bar{y} \vee z) \wedge (\bar{x} \vee \bar{z} \vee y) \wedge ((y \leftrightarrow z) \rightarrow x)$
 $\Leftrightarrow (\bar{x} \vee \bar{y} \vee z) \wedge (\bar{x} \vee \bar{z} \vee y) \wedge (((y \wedge z) \vee (\bar{y} \wedge \bar{z})) \rightarrow x)$
 $\Leftrightarrow (\bar{x} \vee \bar{y} \vee z) \wedge (\bar{x} \vee \bar{z} \vee y) \wedge ((y \wedge z) \rightarrow x) \wedge ((\bar{y} \wedge \bar{z}) \rightarrow x)$
 $\Leftrightarrow (\bar{x} \vee \bar{y} \vee z) \wedge (\bar{x} \vee \bar{z} \vee y) \wedge (\bar{y} \vee \bar{z} \vee x) \wedge (y \vee z \vee x)$

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

$$x \vee \bar{y} \vee \bar{z}$$

$$\bar{x} \vee y$$

$$\bar{x} \vee z$$

Ablesen von KNF aus Funktionstabelle: Zeilen mit 0 betrachten!

Zusammenfassung Tsetin 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

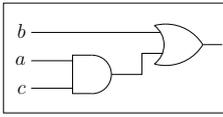
Die Transformation erhält Erfüllbarkeit:

✓ das Ergebnis der Transformation ist erfüllbar g.d.w. die ursprüngliche Formel erfüllbar ist.

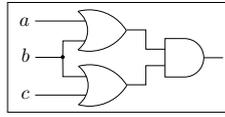


Die Formeln sind **nicht** äquivalent, da die transformierte Formel neue, zusätzliche Variablen enthält.

SAT Beispiel: Äquivalenz von kombinatorischen Schaltungen



$$b \vee a \wedge c$$



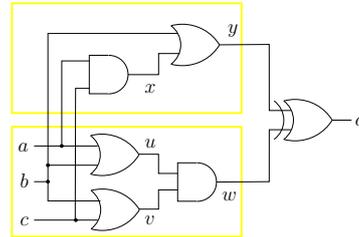
$$(a \vee b) \wedge (b \vee c)$$

äquivalent?

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

Tsetin Transformation

KNF



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

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

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, ... entspricht einer Variable
 - Namen (z.B. a, b, ...) müssen auf Zahlen abgebildet werden
- ▶ Eine negative Zahl entspricht einer Negation
 - Wenn a die Nummer 5 hat, dann entspricht \bar{a} der -5

DIMACS KNF Darstellung

- ▶ Beispiel:

$$(x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_2 \vee \bar{x}_1) \wedge (x_4 \vee \bar{x}_2 \vee \bar{x}_1)$$
- ▶ 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

Variablenzuordnung:

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

Formel:

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

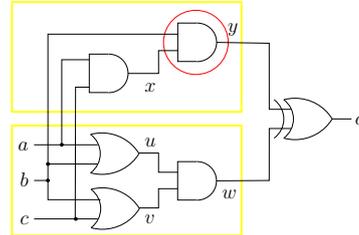
In Reihenfolge des Erscheinens.

SAT Beispiel: Äquivalenz von kombinatorischen Schaltungen

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

SAT Beispiel: Äquivalenz von kombinatorischen Schaltungen

Beispiel von vorhin geringfügig geändert:



$o \wedge$
 $(x \leftrightarrow a \wedge c) \wedge$
 $(y \leftrightarrow b \wedge x) \wedge$
 $(u \leftrightarrow a \vee b) \wedge$
 $(v \leftrightarrow b \vee c) \wedge$
 $(w \leftrightarrow u \wedge v) \wedge$
 $(o \leftrightarrow y \oplus w)$

KNF erfüllbar?

Ergebnis SAT Beispiel

► Ausgabe des SAT-Solvers:

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

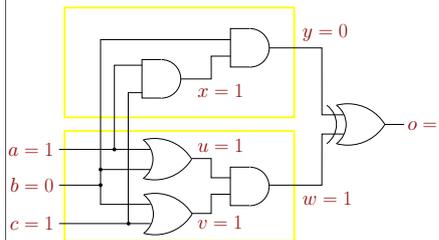
► Werte der Variablen:

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

Ergebnis SAT Beispiel

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 literal;

5     void gate_and(literal a, literal b, literal o);
    void gate_or(literal a, literal b, literal o);
    void gate_xor(literal a, literal b, literal o);
    void gate_nand(literal a, literal b, literal o);
10    void gate_nor(literal a, literal b, literal o);

    literal gate_and(literal a, literal b);
    literal gate_or(literal a, literal b);
    ...
15    literal neue_variable();

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

Größere Beispiele: C++ Klasse

```

...
    unsigned anzahl_variablen;

    typedef std::list<literal> klauselt;
5    typedef std::list<klauselt> klausel_listet;

    klausel_listet klausel_liste;

    void dimacs(std::ostream &out);
10 };
    
```

Beispiel

```

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

   return knf.gate_or(s1, knf.gate_or(s2, s3));
10 }
    
```

Beispiel

```

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

        c=carry(knf, a[i], b[i], c);
    }
15   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?

⇒ Übung

SAT für sequenzielle Schaltungen

Erinnerung:

Definition (Transitionssystem)

Ein Transitionssystem ist ein Tripel (S, I, T) mit

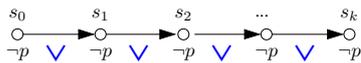
- ▶ S : Menge der Zustände
- ▶ $I \subseteq S$: Initialzustände
- ▶ $T : S \times S$: Übergangsrelation

Bounded Model Checking (BMC)

Assertion sei p (Prädikat über S)

⇒ Suche nach Zustand mit $\neg p$

Idee: begrenze Tiefe auf k Taktzyklen



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

Beispiel

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

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

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

Mit $k = 4$:

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

(noch ohne Eigenschaft)

Beispiel

Die Assertion sei $x \leq 20$.

Mit $k = 4$:

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

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](#)