

Wahrheitstabellen

Besteht aus einer großen Anzahl Wahrheitstabellen, unterschiedlich große je nach Hersteller. Kann einfach und viele oder komplex (3-4 Eingangsleitungen) und dafür weniger Elemente haben.

Weitere Elemente

Hat manchmal eigenen Speicher

Muss Leitungen für das Routing haben

Kann AD/DA Wandler besitzen

Multiplexer/Demultiplexer

Uvm.

Meist große Anzahl von I/O Ports (Leitungen) für massiv parallele Ein- und Ausgabe.

I1	I2	Out
0	0	—
0	1	—
1	0	—
1	1	—

I1	I2	I3	I4	Out
0	0	0	0	—
0	0	0	1	—
...				—
1	1	1	1	—

Beispiel unterschiedlicher Wahrheitstabellen, Outputspalte kann bel. gesetzt werden

FGPA/ASIC

- **F**ield **P**rogrammable **G**ate **A**rray: Kann mit unterschiedl. Schaltungen geladen werden. z.B. unser Atrix-7
- **A**pplication **S**pecific **I**ntegrated **C**ircuit: Schaltung fest verdrahtet bei Produktion, z.B. zur schnellen Berechnung von Bit Coin Hashes.

Design

Schaltung wie z.B. eine ganze CPU, eine Komponente für USB- oder Wifi-Kommunikation, ein einfacher Addierer. Dem Programm beim Programmieren entspricht das Design in der Hardwareentwicklung.

WTF Hardwareentwicklung??!? Sollen doch die Nachrichtentechniker/Ingenieure machen!

CPUs enthalten neben den üblichen CPU-Cores zunehmend auch FPGAs. Selbst entworfene Logik kann um mehrere Größenordnungen schneller sein als Assemblercode in CPUs.

Design als...

RTL: Register Transfer Level

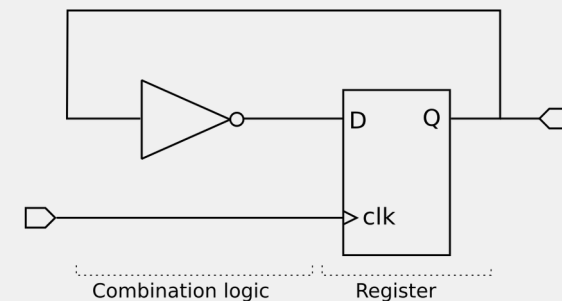
Beschreibt auf funktionaler Ebene was die Schaltung machen soll. Z.B. soll R den invertierten Wert von Q annehmen. Verbreitete Sprachen: Verilog und VHDL

Netlist:

Beschreibt welche Komponenten verbaut und wie sie verbunden werden.

```
reg R, Q;  
...  
R <= ~Q;
```

Design auf RTL Ebene in Verilog



Design als Netlist

FGPA/ASIC

- **F**ield **P**rogrammable **G**ate **A**rray: Kann mit unterschiedl. Schaltungen geladen werden. z.B. unser Atrix-7
- **A**pplication **S**pecific **I**ntegrated **C**ircuit: Schaltung fest verdrahtet bei Produktion, z.B. zur schnellen Berechnung von Bit Coin Hashes.

Design

Schaltung wie z.B. eine ganze CPU, eine Komponente für USB- oder Wifi-Kommunikation, ein einfacher Addierer. Dem Programm beim Programmieren entspricht das Design in der Hardwareentwicklung.

WTF Hardwareentwicklung??!? Sollen doch die Nachrichtentechniker/Ingenieure machen!

CPUs enthalten neben den üblichen CPU-Cores zunehmend auch FPGAs. Selbst entworfene Logik kann um mehrere Größenordnungen schneller sein als Assemblercode in CPUs.

Synthese/synthetisieren

... ist die Übersetzung unseres Verilog Codes in eine Schaltung, analog

Generierung Bitstream

... ist die darauf folgende Abstraktion. Die Schaltung braucht ein Layout beim ASIC oder ein Mapping auf Logikgatter beim FPGA. Der Bitstream definiert, wie die Schaltung für einen FPGA als Daten übertragen wird. Er kann auch auf einem Speichermedium auf dem Board fest gespeichert werden. Der FPGA selbst kann eine Schaltung nicht dauerhaft speichern.

IP/IP-core (intellectual property)

Ein bereits fertiges Modul/fertiger Funktionsblock, der etwas tut und hinzugekauft bzw. bestehenden Projekten hinzugefügt werden kann. Ähnlich einem Unterprogramm. Gibt es also Soft IP Core (z.B. in Verilog) oder Hard IP Core (als fertige Schaltung/wenig veränderbar).

Sprachen zur Entwicklung von FPGA/ASIC Designs

- VHDL
- Verilog

Neues vivado Projekt

Plattform: xc7a35tcp236-1

Sprache: verilog

Constraints Datei: *.xdc (Xilinx Design Constraint)

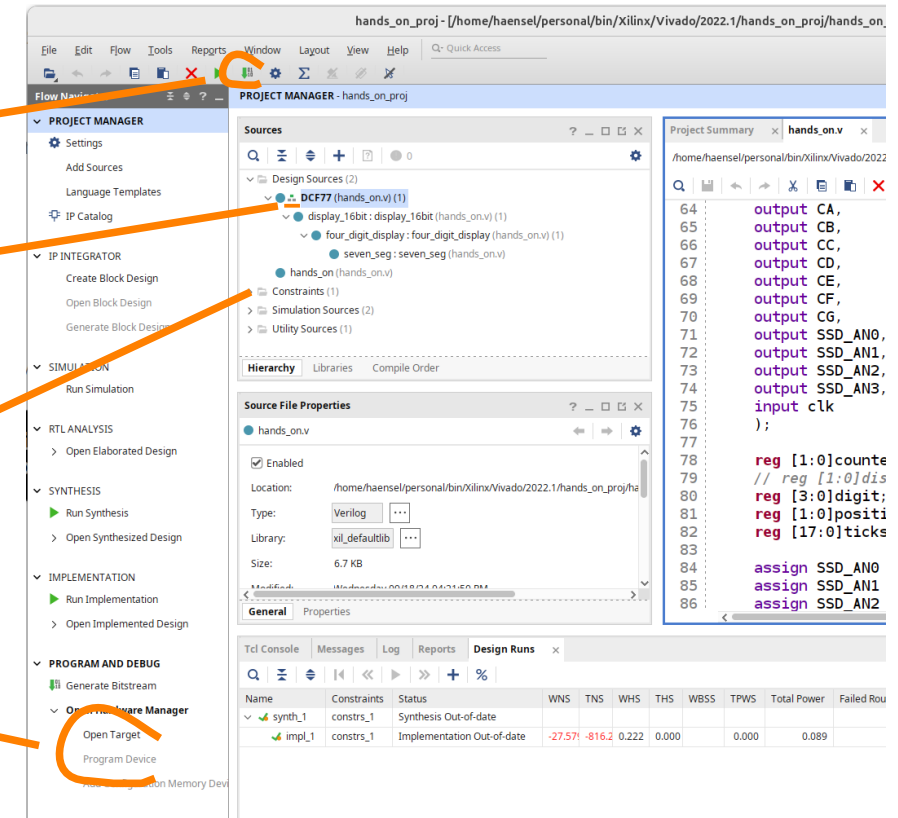
Hello World!

Projekt
übersetzen

Haupt-
modul

Mit HW ver-
binden und
Bitstream
übertragen

Unsere
constraints
Datei mit
Portnamen für
Schalter,
LEDs etc.



```
module hello_world_circuit(  
    input  HW_switch_0,  
    output HW_led_0  
);  
    assign led = switch; // verdrahte Schalter mit LED  
endmodule
```

Im Header stehen die Ports, also was aus dem Modul hinein (**input**) und raus (**output**) geht. Auch beide Richtungen sind manchmal sinnvoll (**inout**).

Im Hauptmodul  müssen die Ports aus der constraints-Datei genommen werden. Warum? Ein Hauptmodul macht nur Sinn, wenn es von außen gesteuert (**input**) werden kann und seine Ausgabe nach außen anzeigen (**output**) kann.

In unseren eigenen Modulen können wir die Ports benennen wie wir wollen. Eigene Module sind ähnlich wie Unterprogramme in Programmiersprachen.

Boolsche Variablen

Boolsche Variablen kann man sich vorstellen als

- Leitung die Strom/Spannung führt (1) oder ausgeschaltet (0) ist
- 1-Bit Speicher der 1 oder 0 speichert
- Schalter der ein (1) oder ausgeschaltet (0) ist
- Aussage die wahr (1) oder falsch (0) sein kann.

Hier schon unterscheidbar: Der 1-Bit Speicher/der Schalter kann einen Zustand speichern. Eine Leitung nicht. Ihr Zustand kann nur gemessen werden. Sie speichert aber nichts.

Boolsche Variablen

Beispiel für boolsche Variable als Aussage:

A: Das Haus hat einen Anschluss an ein Stromnetz

B: Die Lampe ist eingeschaltet

Beide Aussagen können wahr oder falsch sein. Die Lampe leuchtet, wenn A UND B wahr ist. Man kann boolsche Aussagen also je nach Anwendung mit den logischen Operatoren AND, OR, XOR und NOT verknüpfen.

Boolsche Variablen

Wahrheitstabellen

AND & (*)
A B | A&B

0	0		0
0	1		0
1	0		0
1	1		1

OR |
A B | A|B

0	0		0
0	1		1
1	0		1
1	1		1

XOR ^
A B | A^B

0	0		0
0	1		1
1	0		1
1	1		0

NOT ~
A | ~A

0		1
1		0

(*) weil A&B so häufig
vorkommt schreibt
man oft einfach AB

Das OR gibt es in einer weiteren Variante als XOR (exklusiv oder). Im Gegensatz zum ODER dürfen beim XOR nicht beiden Aussagen gleichzeitig wahr sein.

Symbole/Notation

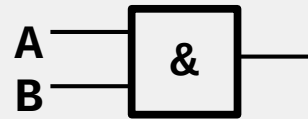
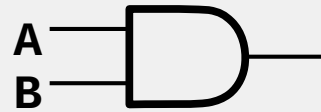
Logischer
Operator

Symbol

Alternat.
Darstl.

in Java/C
(Notation)

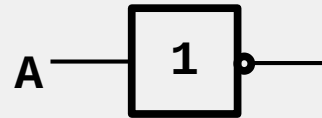
AND



A & B

(A ∧ B oder AB)

NOT

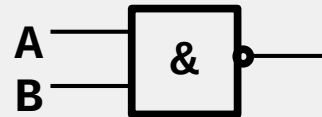
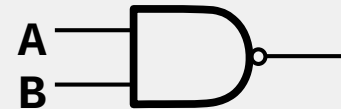


~A

$\bar{A}$

NAND

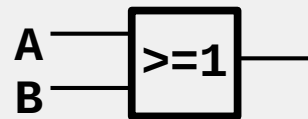
(NOT AND)



~(A & B)

$\overline{(AB)}$

OR

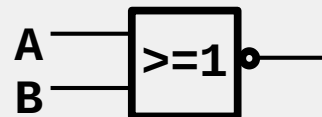


A | B

A ∨ B

NOR

(NOT OR)



~(A | B)

$\overline{(A \vee B)}$

Symbole/Notation

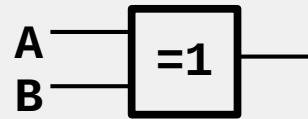
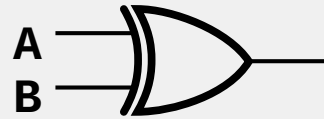
Logischer
Operator

Symbol

Alternat.
Darstl.

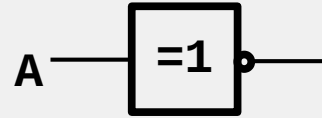
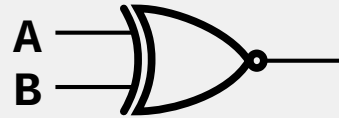
in Java/C
(Notation)

XOR
(exklusiv
oder)



$A \wedge B$

NOT XOR



$\sim(A \wedge B)$
 $\overline{A \wedge B}$

Boolsche Funktionen

Übung Ampelschaltung

Eine Ampel durchläuft die Phasen

Grün, Gelb, Rot, Rot u. Gelb gleichzeitig, zurück zum Anfang...

Wie üblich hat die Ampel drei Leuchten und wird über eine noch zu bestimmende Anzahl binärer Steuerleitungen gesteuert. Die Steuerleitungen können 0/aus oder 1/an sein.

Boolsche Funktionen

Übung Ampelschaltung

Die Anzahl der Steuerleitungen soll minimal sein, jedoch so bemessen, dass alle 4 Zustände schaltbar sind. In der Ampel dürfen die üblichen logischen Operatoren verbaut sein.

- a) Wie viele Steuerleitungen braucht die Ampel mindestens?
- b) Stellen Sie eine Tabelle für die Grüne, Gelbe und Rote Lampe auf. Dabei befinden sich die möglichen Schaltzustände der Steuerleitungen links, die Schaltung der jeweiligen Lampe rechts. Die Tabelle enthält 0en für aus und 1en für ein. Man nennt sie eine Wahrheitstabelle. Natürlich können Sie auch alle drei Tabellen in einer zusammenfassen.

Boolsche Funktionen

Übung Ampelschaltung

- c) Erzeugen Sie in Verilog eine funktionierende Schaltung, die eine ausreichende Anzahl Schalter **HW_switch_x** und LED **HW_led_y** nutzt.



Boolsche Algebra

Die offensichtlichen

$$X \& Y = Y \& X \quad X \mid Y = Y \mid X$$

$$\left. \begin{array}{l} (X \& Y) \& Z = X \& (Y \& Z) \\ (X \mid Y) \mid Z = X \mid (Y \mid Z) \end{array} \right\} \text{ (Assoziativgesetz)}$$

$$X \& X = X \quad \text{und} \quad X \mid X = X \quad \text{(Idempotenzgesetz)}$$

$$X \& 1 = X \quad X \& 0 = 0$$

$$X \mid 1 = 1 \quad X \mid 0 = X$$

$$X \& \sim X = 0 \quad X \mid \sim X = 1$$

$$\sim \sim X = X$$

$$\sim 0 = 1 \quad \sim 1 = 0$$

Boolsche Algebra

Die Interessanten (jeweils ausführliche Schreibweise/kurze Schreibweise)

$$\left. \begin{array}{l} X \& (Y \mid Z) = (X \& Y) \mid (X \& Z) \\ X(Y \mid Z) = XY \mid XZ \\ X \mid (Y \& Z) = (X \mid Y) \& (X \mid Z) \\ X \mid YZ = (X \mid Y)(X \mid Z) \end{array} \right\} \text{(Assoziativgesetz)}$$

$$\sim(X \& Y) = (\sim X \mid \sim Y) \quad \text{kurz: } \overline{XY} = (\overline{X} \mid \overline{Y})$$

$$\sim(X \mid Y) = (\sim X \& \sim Y) \quad \text{kurz: } \overline{(X \mid Y)} = \overline{X} \overline{Y}$$

$$(X \wedge Y) = (X \& \sim Y) \mid (\sim X \& Y) \quad \text{kurz: } (X \wedge Y) = (X \overline{Y}) \mid (\overline{X} Y)$$

$$(A \& B) \mid A = A \text{ bzw. } (A \mid B) \& A = A \quad \text{(Absorptionsgesetz)}$$

$$AB \mid A = A \text{ bzw. } (A \mid B)A = A$$

$$\left. \begin{array}{l} A \& C \mid \overline{B} \& \sim A \mid B \& C = A \& C \mid B \& \sim A \\ AC \mid \overline{B} A \mid BC = AC \mid \overline{B} A \end{array} \right\} \begin{array}{l} \text{(Konsensus-Regel)} \\ \text{(denn das } BC \text{ ist unnötige Verschärfung)} \end{array}$$

Boolsche Algebra

Vereinfachen Sie den folgenden logischen Ausdruck:

$$A \sim B \sim C \mid A \sim BC \mid A \sim B$$

Boolsche Algebra

Vereinfachen Sie den folgenden logischen Ausdruck:

$$AB \mid A(B \mid C) \mid B(B \mid C)$$

Boolsche Algebra

Primzahlendetektor

Eine Schaltung soll für jede mögliche 3-Bit Zahl anzeigen, ob es sich um eine Primzahl (LED an) oder nicht (LED aus) handelt.

Bemerkung: 1 gilt nicht als Primzahl.

- a) Stellen Sie die Wahrheitstabelle für die Schaltung auf
- b) Vereinfachen Sie die Schaltung so weit wie möglich
- c) Implementieren Sie die Schaltung in Verilog

Boolsche Algebra

Vereinfachung als KV-Diagramm

x		$\sim x$		
$\sim y$	y	$\sim y$	y	z
1	1			
	1	1		
				$\sim z$

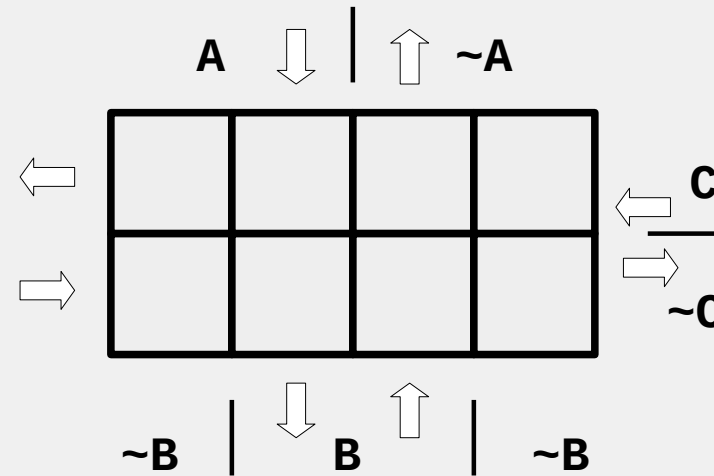
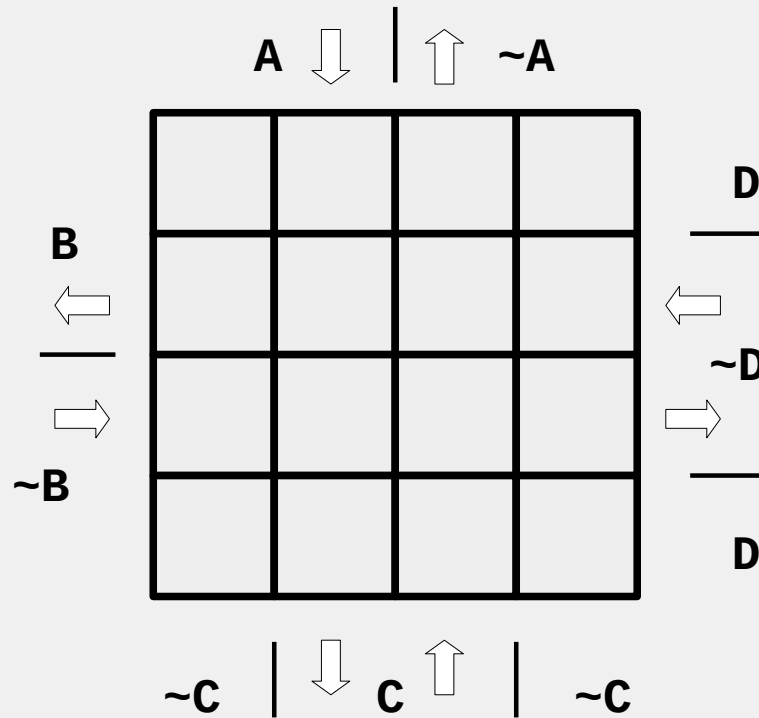
KV-Diagramme

Anwendung: Möchte meinen logischen Ausdruck minimieren.

KV-Diagramme (hier) für 4 bzw. 3 boolsche Variablen. Jedes Feld steht für eine Kombination der Variablen. Die Pfeile sollen symbolisieren, dass man links ins Diagramm eintritt, sobald man es rechts verlässt (analog umgekehrt und ebenso oben/unten). Gegenüberliegende Seiten hängen topologisch zusammen.

Boolsche Algebra

KV-Diagramme



Boolsche Algebra

KV-Diagramme: Beispiel

A B C D | E

0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

Disjunktive Normalenform aus sieben
Implikanten ohne Vereinfachung

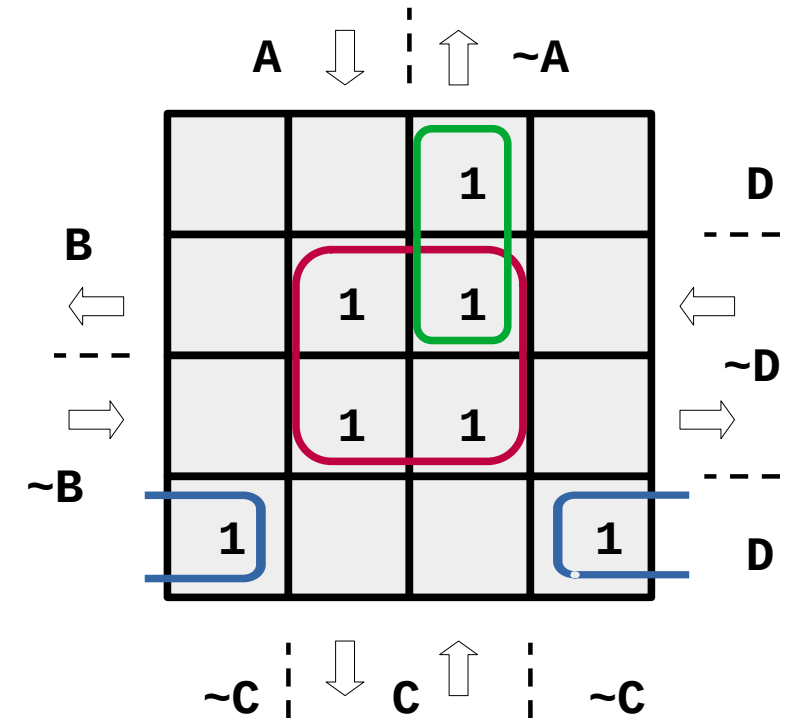
$$E = \sim A \sim B \sim C D \mid \sim A \sim B C \sim D \mid \sim A B C \sim D \mid \\ \sim A B C D \mid A \sim B \sim C D \mid A \sim B C D \mid A B C \sim D$$

Boolsche Algebra

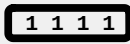
KV-Diagramme: Beispiel

A	B	C	D	E
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

$$E = C \sim D \mid \sim ABC \mid \sim B \sim CD$$



KV-Diagramme: Regeln

- Fasse nur Blöcke der Größe 1, 2, 4, 8 oder 16 zusammen (nur 2^n Blöcke). Erlaubt sind alle Blöcke außer über die Diagonalen, also z.B.:
1x1,  1x2, 
1x4,  2x1, 
2x2,  4x1, 2x4, 4x1, 4x2, 4x4.
- Je größer der Block, desto mehr Variablen sind egal. Beim grünen Block (vorherige Seite) spielt z.B. D keine Rolle, weil sowohl D als auch $\sim D$ enthalten ist. Brennt die Lampe E einer Schaltung egal ob Schalter D ein oder ausgeschaltet ist, dann hat D keine Funktion und kann entfallen. Extremfall aber unrealistisch: 4x4 Block aus Einsen. Dann spielen alle 4 Variablen keine Rolle.
- Beachte auch: Blöcke könnten sich auch über gegenüberliegende Seiten erstrecken (wie der blaue).
- Manchmal kommt eine 1 in mehreren Blöcken vor. Das ist kein Problem. Eher vorteilhaft, weil es größere Blöcke ermöglicht! Natürlich müssen alle 1 mindestens einmal überdeckt sein.
- Es kann mehrere Möglichkeiten geben alle Einsen durch Blöcke abzudecken. Nicht alle Möglichkeiten müssen minimal sein.