Einführung in die Programmierung

Vorlesungsteil 4
Aussagenlogik, logische Schaltungen und Bitoperationen in C

PD Dr. Thomas Hinze

Brandenburgische Technische Universität Cottbus – Senftenberg
Institut für Informatik, Informations- und Medientechnik

Wintersemester 2015/2016

Brandenburgische
Technische Universität
Cottbus - Senftenberg

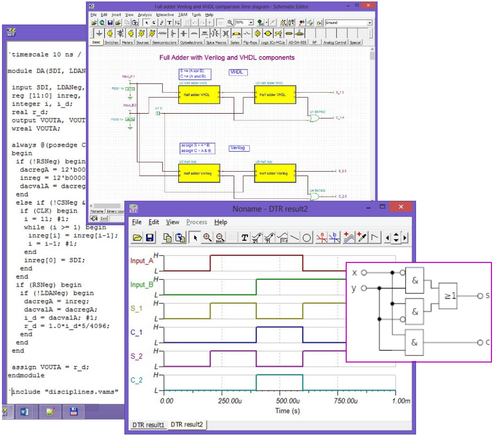
Logikgatter als Grundbausteine von Computern

"Beliebig verschaltbare NAND–Logikgatter, ein Taktgenerator und Draehte sind alles, was man braucht, um einen Computer zu bauen, der eine beliebige algorithmisch loesbare Aufgabe erledigen kann."

Grunderkenntnis der Hardwareentwicklung

Schaltungsentwurf auf Ebene logischer Gatter

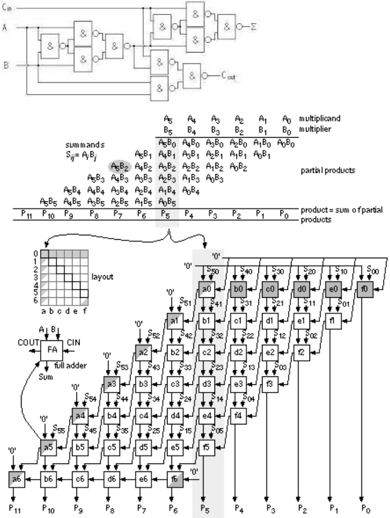
- Hardwarekomponenten am Computer konstruiert
- Verhalten dieser Schaltungen erst simuliert und optimiert, bevor physischer Aufbau (Synthese) erfolgt
- Dazu dienen Hardware-beschreibungssprachen wie *Verilog* oder *VHDL*, die syntaktisch an C angelehnt sind



verilog.com – Halbadder: Beschreibung, Simulation

Logikoperationen sehr schnell ausgeführt

- Arithmetische Ganzzahl- oder Gleitkommaoperation benötigt in Größenordnungen (> 100) mehr Ausführungszeit als eine bitweise logische Verknüpfung
- Geeignetes Ersetzen arithmetischer Operationen durch logische Bitoperationen kann C-Programme mitunter deutlich beschleunigen
- Logikoperationen bevorzugt verwenden, z.B. bei $x, y \in \{0, 1\}$ $x \wedge y$ statt $x \cdot y$
- Compiler unterstützt den Programmierer hier kaum bis gar nicht



Bilder: TUD. Volladdierer (oben) und Ganzzahl-Multiplizierer als bitversetzte Hintereinanderschaltung von Volladdierern

Vorlesung Einführung in die Programmierung mit C

Begriff Aussage (im logischen Sinne)

Eine **Aussage** ist ein *mathematischer Term* oder ein *sprachliches Gebilde*, dem ein konkreter Wahrheitswert (entweder *wahr* bzw. *falsch*) zugeordnet werden kann.

Begriff Aussage (im logischen Sinne)

Eine **Aussage** ist ein *mathematischer Term* oder ein *sprachliches Gebilde*, dem ein konkreter Wahrheitswert (entweder *wahr* bzw. *falsch*) zugeordnet werden kann.

Blau ist eine Farbe . . . .

Begriff Aussage (im logischen Sinne)

Eine **Aussage** ist ein *mathematischer Term* oder ein *sprachliches Gebilde*, dem ein konkreter Wahrheitswert (entweder *wahr* bzw. *falsch*) zugeordnet werden kann.

Blau ist eine Farbe .... wahr

Begriff Aussage (im logischen Sinne)

Eine **Aussage** ist ein *mathematischer Term* oder ein *sprachliches Gebilde*, dem ein konkreter Wahrheitswert (entweder *wahr* bzw. *falsch*) zugeordnet werden kann.

Blau ist eine Farbe .... wahr
$5 < 7$ ....

Begriff Aussage (im logischen Sinne)

Eine **Aussage** ist ein *mathematischer Term* oder ein *sprachliches Gebilde*, dem ein konkreter Wahrheitswert (entweder *wahr* bzw. *falsch*) zugeordnet werden kann.

Blau ist eine Farbe	wahr
$5 < 7$	wahr

Begriff Aussage (im logischen Sinne)

Eine **Aussage** ist ein *mathematischer Term* oder ein *sprachliches Gebilde*, dem ein konkreter Wahrheitswert (entweder *wahr* bzw. *falsch*) zugeordnet werden kann.

Blau ist eine Farbe	wahr
$5 < 7$	wahr
$9.81 \in \mathbb{N}$	

Begriff Aussage (im logischen Sinne)

Eine **Aussage** ist ein *mathematischer Term* oder ein *sprachliches Gebilde*, dem ein konkreter Wahrheitswert (entweder *wahr* bzw. *falsch*) zugeordnet werden kann.

Blau ist eine Farbe	wahr
$5 < 7$	wahr
$9.81 \in \mathbb{N}$	falsch

Begriff Aussage (im logischen Sinne)

Eine **Aussage** ist ein *mathematischer Term* oder ein *sprachliches Gebilde*, dem ein konkreter Wahrheitswert (entweder *wahr* bzw. *falsch*) zugeordnet werden kann.

Blau ist eine Farbe	wahr
$5 < 7$	wahr
$9.81 \in \mathbb{N}$	falsch
4 ist eine Primzahl	

Begriff Aussage (im logischen Sinne)

Eine **Aussage** ist ein *mathematischer Term* oder ein *sprachliches Gebilde*, dem ein konkreter Wahrheitswert (entweder *wahr* bzw. *falsch*) zugeordnet werden kann.

Blau ist eine Farbe .... wahr
$5 < 7$ .... wahr
$9.81 \in \mathbb{N}$ .... falsch
4 ist eine Primzahl .... falsch

Begriff Aussage (im logischen Sinne)

Eine **Aussage** ist ein *mathematischer Term* oder ein *sprachliches Gebilde*, dem ein konkreter Wahrheitswert (entweder *wahr* bzw. *falsch*) zugeordnet werden kann.

Blau ist eine Farbe	wahr
$5 < 7$	wahr
$9.81 \in \mathbb{N}$	falsch
4 ist eine Primzahl	falsch
Der Planet Erde wiegt exakt $5.9736 \cdot 10^{24}$kg. ..	

Begriff Aussage (im logischen Sinne)

Eine **Aussage** ist ein *mathematischer Term* oder ein *sprachliches Gebilde*, dem ein konkreter Wahrheitswert (entweder *wahr* bzw. *falsch*) zugeordnet werden kann.

Blau ist eine Farbe	wahr
$5 < 7$	wahr
$9.81 \in \mathbb{N}$	falsch
4 ist eine Primzahl	falsch
Der Planet Erde wiegt exakt $5.9736 \cdot 10^{24}$kg. . .	Wahrheitswert derzeit nicht angebbar

Begriff Aussage (im logischen Sinne)

Eine **Aussage** ist ein *mathematischer Term* oder ein *sprachliches Gebilde*, dem ein konkreter Wahrheitswert (entweder *wahr* bzw. *falsch*) zugeordnet werden kann.

Blau ist eine Farbe	wahr
$5 < 7$	wahr
$9.81 \in \mathbb{N}$	falsch
4 ist eine Primzahl	falsch
Der Planet Erde wiegt exakt $5.9736 \cdot 10^{24}$kg. ..	Wahrheitswert derzeit nicht angebbar
Gleichung $x^x = 10$ nicht exakt lösbar im Bereich rationaler Zahlen	

Begriff Aussage (im logischen Sinne)

Eine **Aussage** ist ein *mathematischer Term* oder ein *sprachliches Gebilde*, dem ein konkreter Wahrheitswert (entweder *wahr* bzw. *falsch*) zugeordnet werden kann.

Blau ist eine Farbe	wahr
$5 < 7$	wahr
$9.81 \in \mathbb{N}$	falsch
4 ist eine Primzahl	falsch
Der Planet Erde wiegt exakt $5.9736 \cdot 10^{24}$kg. ..	Wahrheitswert derzeit nicht angebbar
Gleichung $x^x = 10$ nicht exakt lösbar im Bereich rationaler Zahlen	wahr

Begriff Aussage (im logischen Sinne)

Eine **Aussage** ist ein *mathematischer Term* oder ein *sprachliches Gebilde*, dem ein konkreter Wahrheitswert (entweder *wahr* bzw. *falsch*) zugeordnet werden kann.

Blau ist eine Farbe	wahr
$5 < 7$	wahr
$9.81 \in \mathbb{N}$	falsch
4 ist eine Primzahl	falsch
Der Planet Erde wiegt exakt $5.9736 \cdot 10^{24}$kg. ..	Wahrheitswert derzeit nicht angebbar
Gleichung $x^x = 10$ nicht exakt lösbar im Bereich rationaler Zahlen	wahr
$0^0 = 1$	

Begriff Aussage (im logischen Sinne)

Eine **Aussage** ist ein *mathematischer Term* oder ein *sprachliches Gebilde*, dem ein konkreter Wahrheitswert (entweder *wahr* bzw. *falsch*) zugeordnet werden kann.

Blau ist eine Farbe	wahr
$5 < 7$	wahr
$9.81 \in \mathbb{N}$	falsch
4 ist eine Primzahl	falsch
Der Planet Erde wiegt exakt $5.9736 \cdot 10^{24}$kg. ..	Wahrheitswert derzeit nicht angebbar
Gleichung $x^x = 10$ nicht exakt lösbar im Bereich rationaler Zahlen	wahr
$0^0 = 1$	keine Aussage (nicht definiert)

Begriff Aussage (im logischen Sinne)

Eine **Aussage** ist ein *mathematischer Term* oder ein *sprachliches Gebilde*, dem ein konkreter Wahrheitswert (entweder *wahr* bzw. *falsch*) zugeordnet werden kann.

Blau ist eine Farbe	wahr
$5 < 7$	wahr
$9.81 \in \mathbb{N}$	falsch
4 ist eine Primzahl	falsch
Der Planet Erde wiegt exakt $5.9736 \cdot 10^{24}$kg. ..	Wahrheitswert derzeit nicht angebbar
Gleichung $x^x = 10$ nicht exakt lösbar im Bereich rationaler Zahlen	wahr
$0^0 = 1$	keine Aussage (nicht definiert)
Ich lüge jetzt.	

Begriff Aussage (im logischen Sinne)

Eine **Aussage** ist ein *mathematischer Term* oder ein *sprachliches Gebilde*, dem ein konkreter Wahrheitswert (entweder *wahr* bzw. *falsch*) zugeordnet werden kann.

Blau ist eine Farbe	wahr
$5 < 7$	wahr
$9.81 \in \mathbb{N}$	falsch
4 ist eine Primzahl	falsch
Der Planet Erde wiegt exakt $5.9736 \cdot 10^{24}$kg. ..	Wahrheitswert derzeit nicht angebbar
Gleichung $x^x = 10$ nicht exakt lösbar im Bereich rationaler Zahlen	wahr
$0^0 = 1$	keine Aussage (nicht definiert)
Ich lüge jetzt.	keine Aussage (Antinomie)

Begriff Aussage (im logischen Sinne)

Eine **Aussage** ist ein *mathematischer Term* oder ein *sprachliches Gebilde*, dem ein konkreter Wahrheitswert (entweder *wahr* bzw. *falsch*) zugeordnet werden kann.

Blau ist eine Farbe	wahr
$5 < 7$	wahr
$9.81 \in \mathbb{N}$	falsch
4 ist eine Primzahl	falsch
Der Planet Erde wiegt exakt $5.9736 \cdot 10^{24}$kg. ..	Wahrheitswert derzeit nicht angebbar
Gleichung $x^x = 10$ nicht exakt lösbar im Bereich rationaler Zahlen	wahr
$0^0 = 1$	keine Aussage (nicht definiert)
Ich lüge jetzt.	keine Aussage (Antinomie)
Das Wetter ist schön.	

Begriff Aussage (im logischen Sinne)

Eine **Aussage** ist ein *mathematischer Term* oder ein *sprachliches Gebilde*, dem ein konkreter Wahrheitswert (entweder *wahr* bzw. *falsch*) zugeordnet werden kann.

Blau ist eine Farbe	wahr
$5 < 7$	wahr
$9.81 \in \mathbb{N}$	falsch
4 ist eine Primzahl	falsch
Der Planet Erde wiegt exakt $5.9736 \cdot 10^{24}$kg. ..	Wahrheitswert derzeit nicht angebbar
Gleichung $x^x = 10$ nicht exakt lösbar im Bereich rationaler Zahlen	wahr
$0^0 = 1$	keine Aussage (nicht definiert)
Ich lüge jetzt.	keine Aussage (Antinomie)
Das Wetter ist schön.	keine Aussage (subjektiver Interpretationsspielraum)

Begriff Aussage (im logischen Sinne)

Eine **Aussage** ist ein *mathematischer Term* oder ein *sprachliches Gebilde*, dem ein konkreter Wahrheitswert (entweder *wahr* bzw. *falsch*) zugeordnet werden kann.

Blau ist eine Farbe	wahr
$5 < 7$	wahr
$9.81 \in \mathbb{N}$	falsch
4 ist eine Primzahl	falsch
Der Planet Erde wiegt exakt $5.9736 \cdot 10^{24}$kg. ..	Wahrheitswert derzeit nicht angebbar
Gleichung $x^x = 10$ nicht exakt lösbar im Bereich rationaler Zahlen	wahr
$0^0 = 1$	keine Aussage (nicht definiert)
Ich lüge jetzt.	keine Aussage (Antinomie)
Das Wetter ist schön.	keine Aussage (subjektiver Interpretationsspielraum)

$\Longrightarrow$ In C wird jede Aussage durch einen **Ganzzahlausdruck** beschrieben, z.B. `(5 % 4) * 2 + (5 < 7)`. Hat der Ausdruck den Wert 0, so ist seine Aussage falsch. Hat er einen Wert **ungleich** 0, so ist seine Aussage wahr.

Rechnen mit Aussagen

„Nicht alle Spielzeugbausteine
sind weiß oder rot."

ist gleichbedeutend mit

„Es gibt mindestens einen Spielzeugbaustein,
der weder weiß noch rot ist."

Rechnen mit Aussagen

„Nicht alle Spielzeugbausteine
sind weiß oder rot.“

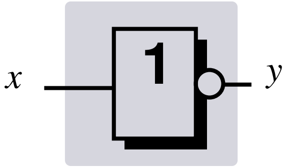
ist gleichbedeutend mit

„Es gibt mindestens einen Spielzeugbaustein,
der weder weiß noch rot ist.“

- Eine Aussage lässt sich formalisieren durch eine *Variable*, die den Wert 0 (falsch) oder den Wert 1 (wahr) annehmen kann.
- *Logische Verknüpfungen* sind Operatoren auf Aussagen.
- Entsprechende Rechengesetze (Boolesche Algebra) erlauben Vereinfachung aussagenlogischer Terme und logisches Schließen.

NICHT-Verknüpfung (NOT, Inverter)

x	y
0	1
1	0

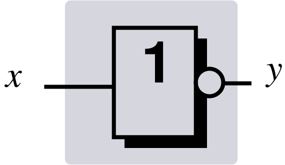


- Boolesche Notation: $y = \overline{x}$
- C-Notation: `y = !x`
- Bei $x, y \in \{0, 1\}$ arithmetische Entsprechung: $y = 1 - x$
- Jeder Ganzzahlwert ungleich 0 wird durch den `!`-Operator zu 0, z.B. `!127` ergibt `0`
- Doppelte Negation im C-Quelltext sinnvoll, um beliebige Ganzzahlen ungleich 0 in 1 abzubilden:

$$\texttt{!!x} = \begin{cases} \texttt{1} & \text{falls } \texttt{x} \neq \texttt{0} \\ \texttt{0} & \text{sonst} \end{cases}$$

NICHT-Verknüpfung (NOT, Inverter)

x	y
0	1
1	0

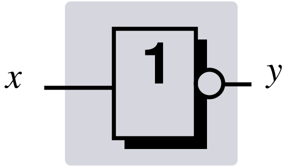


- Boolesche Notation: $y = \overline{x}$
- C-Notation: `y = !x`
- Bei $x, y \in \{0, 1\}$ arithmetische Entsprechung: $y = 1 - x$
- Jeder Ganzzahlwert ungleich 0 wird durch den `!`-Operator zu 0, z.B. `!127` ergibt `0`
- Doppelte Negation im C-Quelltext sinnvoll, um beliebige Ganzzahlen ungleich 0 in 1 abzubilden:

$$\texttt{!!x} = \begin{cases} \texttt{1} & \text{falls } \texttt{x} \neq \texttt{0} \\ \texttt{0} & \text{sonst} \end{cases}$$

NICHT-Verknüpfung (NOT, Inverter)

x	y
0	1
1	0

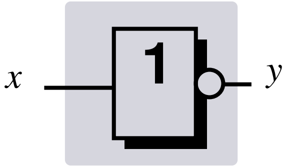


- Boolesche Notation: $y = \overline{x}$
- C-Notation: `y = !x`
- Bei $x, y \in \{0, 1\}$ arithmetische Entsprechung: $y = 1 - x$
- Jeder Ganzzahlwert ungleich 0 wird durch den `!`-Operator zu 0, z.B. `!127` ergibt `0`
- Doppelte Negation im C-Quelltext sinnvoll, um beliebige Ganzzahlen ungleich 0 in 1 abzubilden:

$$\texttt{!!x} = \begin{cases} \texttt{1} & \text{falls } \texttt{x} \neq \texttt{0} \\ \texttt{0} & \text{sonst} \end{cases}$$

NICHT-Verknüpfung (NOT, Inverter)

x	y
0	1
1	0

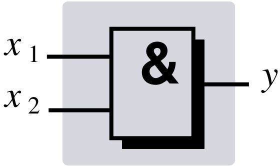


- Boolesche Notation: $y = \overline{x}$
- C-Notation: `y = !x`
- Bei $x, y \in \{0, 1\}$ arithmetische Entsprechung: $y = 1 - x$
- Jeder Ganzzahlwert ungleich 0 wird durch den `!`-Operator zu 0, z.B. `!127` ergibt `0`
- Doppelte Negation im C-Quelltext sinnvoll, um beliebige Ganzzahlen ungleich 0 in 1 abzubilden:

$$\texttt{!!x} = \begin{cases} \texttt{1} & \text{falls } \texttt{x} \neq \texttt{0} \\ \texttt{0} & \text{sonst} \end{cases}$$

UND-Verknüpfung (AND, Konjunktion)

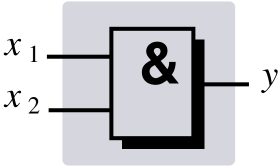
x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1



- Boolesche Notation: $y = x_1 \wedge x_2$ bzw. kurz: $y = x_1 x_2$
- C-Notation: `y = x1 && x2`
- Bei $x_1, x_2, y \in \{0, 1\}$ arithmetische Entsprechung: $y = x_1 \cdot x_2$
- Genau dann, wenn beide Ganzzahlwerte `x1` und `x2` jeweils ungleich `0` sind, wird als Ergebnis `1` geliefert, ansonsten `0`
- Beispiel: `127 && 42` ergibt `1`

UND-Verknüpfung (AND, Konjunktion)

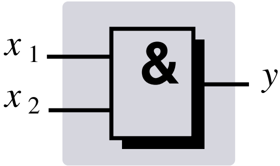
x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1



- Boolesche Notation: $y = x_1 \wedge x_2$ bzw. kurz: $y = x_1 x_2$
- C-Notation: `y = x1 && x2`
- Bei $x_1, x_2, y \in \{0, 1\}$ arithmetische Entsprechung: $y = x_1 \cdot x_2$
- Genau dann, wenn beide Ganzzahlwerte `x1` und `x2` jeweils ungleich `0` sind, wird als Ergebnis `1` geliefert, ansonsten `0`
- Beispiel: `127 && 42` ergibt `1`

UND-Verknüpfung (AND, Konjunktion)

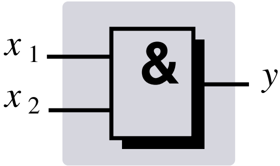
x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1



- Boolesche Notation: $y = x_1 \wedge x_2$ bzw. kurz: $y = x_1 x_2$
- C-Notation: `y = x1 && x2`
- Bei $x_1, x_2, y \in \{0, 1\}$ arithmetische Entsprechung: $y = x_1 \cdot x_2$
- Genau dann, wenn beide Ganzzahlwerte `x1` und `x2` jeweils ungleich `0` sind, wird als Ergebnis `1` geliefert, ansonsten `0`
- Beispiel: `127 && 42` ergibt `1`

UND-Verknüpfung (AND, Konjunktion)

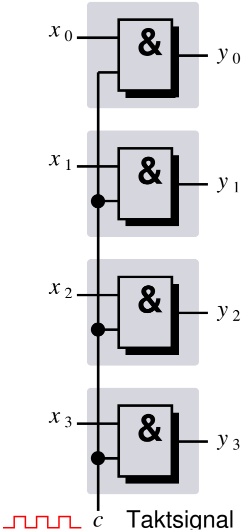
x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1



- Boolesche Notation: $y = x_1 \wedge x_2$ bzw. kurz: $y = x_1 x_2$
- C-Notation: `y = x1 && x2`
- Bei $x_1, x_2, y \in \{0, 1\}$ arithmetische Entsprechung: $y = x_1 \cdot x_2$
- Genau dann, wenn beide Ganzzahlwerte `x1` und `x2` jeweils ungleich `0` sind, wird als Ergebnis `1` geliefert, ansonsten `0`
- Beispiel: `127 && 42` ergibt `1`

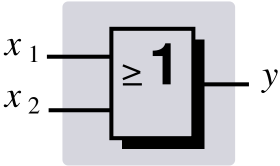
Tor aus UND-Gattern

z.B. zur Freigabe einer Speicherzelle zum Auslesen oder Beschreiben



ODER-Verknüpfung (OR, Disjunktion)

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

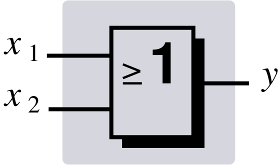


- Boolesche Notation: $y = x_1 \vee x_2$
- C-Notation: `y = x1 || x2`
- Bei $x_1, x_2, y \in \{0, 1\}$ arithmetische Entsprechung:

  $y = x_1 + x_2 - x_1 \cdot x_2$
- Genau dann, wenn mindestens einer der beiden Ganzzahlwerte `x1` und `x2` jeweils ungleich `0` ist, wird als Ergebnis `1` geliefert, ansonsten `0`
- Beispiel: `127 || 0` ergibt `1`

ODER-Verknüpfung (OR, Disjunktion)

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

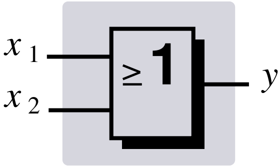


- Boolesche Notation: $y = x_1 \vee x_2$
- C-Notation: `y = x1 || x2`
- Bei $x_1, x_2, y \in \{0, 1\}$ arithmetische Entsprechung:

  $y = x_1 + x_2 - x_1 \cdot x_2$
- Genau dann, wenn mindestens einer der beiden Ganzzahlwerte `x1` und `x2` jeweils ungleich `0` ist, wird als Ergebnis `1` geliefert, ansonsten `0`
- Beispiel: `127 || 0` ergibt `1`

ODER-Verknüpfung (OR, Disjunktion)

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

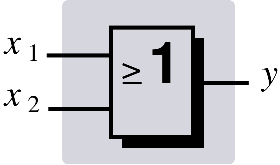


- Boolesche Notation: $y = x_1 \vee x_2$
- C-Notation: `y = x1 || x2`
- Bei $x_1, x_2, y \in \{0, 1\}$ arithmetische Entsprechung:

  $y = x_1 + x_2 - x_1 \cdot x_2$
- Genau dann, wenn mindestens einer der beiden Ganzzahlwerte `x1` und `x2` jeweils ungleich `0` ist, wird als Ergebnis `1` geliefert, ansonsten `0`
- Beispiel: `127 || 0` ergibt `1`

ODER-Verknüpfung (OR, Disjunktion)

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1



- Boolesche Notation: $y = x_1 \vee x_2$
- C-Notation: `y = x1 || x2`
- Bei $x_1, x_2, y \in \{0, 1\}$ arithmetische Entsprechung:

  $y = x_1 + x_2 - x_1 \cdot x_2$
- Genau dann, wenn mindestens einer der beiden Ganzzahlwerte `x1` und `x2` jeweils ungleich `0` ist, wird als Ergebnis `1` geliefert, ansonsten `0`
- Beispiel: `127 || 0` ergibt `1`

Priorisierung von Negation, UND- und ODER-Verknüpfung

Negation

x	1−x
0	1
1	0

`!x`

UND

x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

`x && y`

ODER

x	y	x+y−xy
0	0	0
0	1	1
1	0	1
1	1	1

`x || y`

Bitaddierer (Halbadder)

x	y	sum	$carry$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Bitaddierer (Halbadder)

x	y	sum	$carry$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

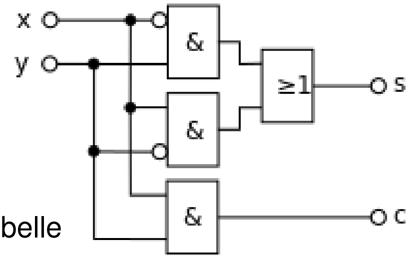
Auslesen der Schaltbelegungstabelle

$$\begin{aligned} sum &= \overline{x}\ y \vee x\ \overline{y} \\ carry &= x\ y \end{aligned}$$

Bitaddierer (Halbadder)

x	y	sum	$carry$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Auslesen der Schaltbelegungstabelle



$$sum = \overline{x}\ y \vee x\ \overline{y}$$
$$carry = x\ y$$

$\Longrightarrow$ Negation in Schaltbild durch $\circ$ an den Gattern dargestellt

Rechengesetze der Aussagenlogik

$$
\begin{aligned}
x \vee \overline{x} &= 1 \\
x \quad \overline{x} &= 0
\end{aligned}
$$

Rechengesetze der Aussagenlogik

$$
\begin{aligned}
x \vee \overline{x} &= 1 \\
x \ \overline{x} &= 0
\end{aligned}
$$

$$
\begin{aligned}
x \vee x &= x \\
x \ x &= x
\end{aligned}
$$

Rechengesetze der Aussagenlogik

$$x \vee \overline{x} = 1$$
$$x \ \overline{x} = 0$$

$$x \vee x = x$$
$$x \ x = x$$

$$\overline{\overline{x}} = x$$ .... doppelte Negation

Rechengesetze der Aussagenlogik

$$
\begin{aligned}
x \vee \overline{x} &= 1 \\
x \; \overline{x} &= 0 \\[6pt]
x \vee x &= x \\
x \; x &= x \\[6pt]
\overline{\overline{x}} &= x \quad \ldots\ldots\ldots\ldots\ldots\ldots \text{doppelte Negation} \\[6pt]
\overline{x \vee y} &= \overline{x} \; \overline{y} \quad \ldots\ldots\ldots\ldots \text{De Morgansche Regel} \\
\overline{x \; y} &= \overline{x} \vee \overline{y} \quad \ldots\ldots\ldots\ldots \text{De Morgansche Regel}
\end{aligned}
$$

Rechengesetze der Aussagenlogik

$$x \vee \overline{x} = 1$$
$$x \;\; \overline{x} = 0$$

$$x \vee x = x$$
$$x \;\; x = x$$

$$\overline{\overline{x}} = x$$ .... doppelte Negation

$$\overline{x \vee y} = \overline{x} \;\; \overline{y}$$ .... De Morgansche Regel
$$\overline{x \;\; y} = \overline{x} \vee \overline{y}$$ .... De Morgansche Regel

$$x \vee y = y \vee x$$ .... Kommutativität
$$x \;\; y = y \;\; x$$ .... Kommutativität

Rechengesetze der Aussagenlogik

$$
\begin{aligned}
x \vee \overline{x} &= 1 \\
x \quad \overline{x} &= 0 \\[1em]
x \vee x &= x \\
x \quad x &= x \\[1em]
\overline{\overline{x}} &= x \quad \dots\dots\dots\dots\dots\dots \text{doppelte Negation} \\[1em]
\overline{x \vee y} &= \overline{x} \quad \overline{y} \quad \dots\dots\dots\dots \text{De Morgansche Regel} \\
\overline{x \quad y} &= \overline{x} \vee \overline{y} \quad \dots\dots\dots\dots \text{De Morgansche Regel} \\[1em]
x \vee y &= y \vee x \quad \dots\dots\dots\dots\dots\dots \text{Kommutativität} \\
x \quad y &= y \quad x \quad \dots\dots\dots\dots\dots\dots \text{Kommutativität} \\[1em]
a \vee (b \vee c) &= (a \vee b) \vee c \quad \dots\dots\dots\dots \text{Assoziativität} \\
a \quad (b \quad c) &= (a \quad b) \quad c \quad \dots\dots\dots\dots \text{Assoziativität}
\end{aligned}
$$

Rechengesetze der Aussagenlogik

$$
\begin{aligned}
x \vee \overline{x} &= 1 \\
x \quad \overline{x} &= 0 \\[1ex]
x \vee x &= x \\
x \quad x &= x \\[1ex]
\overline{\overline{x}} &= x \quad \text{.................... doppelte Negation} \\[1ex]
\overline{x \vee y} &= \overline{x} \quad \overline{y} \quad \text{.............. De Morgansche Regel} \\
\overline{x \quad y} &= \overline{x} \vee \overline{y} \quad \text{............ De Morgansche Regel} \\[1ex]
x \vee y &= y \vee x \quad \text{.................... Kommutativität} \\
x \quad y &= y \quad x \quad \text{...................... Kommutativität} \\[1ex]
a \vee (b \vee c) &= (a \vee b) \vee c \quad \text{................. Assoziativität} \\
a \quad (b \quad c) &= (a \quad b) \quad c \quad \text{................... Assoziativität} \\[1ex]
a \quad (b \vee c) &= a \quad b \vee a \quad c \quad \text{.................. Distributivität}
\end{aligned}
$$

Rechengesetze der Aussagenlogik

$$
\begin{aligned}
x \vee \overline{x} &= 1 \\
x \quad \overline{x} &= 0 \\
\\
x \vee x &= x \\
x \quad x &= x \\
\\
\overline{\overline{x}} &= x \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{doppelte Negation} \\
\\
\overline{x \vee y} &= \overline{x} \quad \overline{y} \quad \ldots\ldots\ldots\ldots\ldots \text{De Morgansche Regel} \\
\overline{x \quad y} &= \overline{x} \vee \overline{y} \quad \ldots\ldots\ldots\ldots \text{De Morgansche Regel} \\
\\
x \vee y &= y \vee x \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{Kommutativität} \\
x \quad y &= y \quad x \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{Kommutativität} \\
\\
a \vee (b \vee c) &= (a \vee b) \vee c \quad \ldots\ldots\ldots\ldots\ldots \text{Assoziativität} \\
a \quad (b \quad c) &= (a \quad b) \quad c \quad \ldots\ldots\ldots\ldots\ldots \text{Assoziativität} \\
\\
a \quad (b \vee c) &= a \quad b \vee a \quad c \quad \ldots\ldots\ldots\ldots\ldots \text{Distributivität} \\
a \vee (b \quad c) &= (a \vee b) \quad (a \vee c) \quad \ldots\ldots\ldots\ldots \text{Distributivität}
\end{aligned}
$$

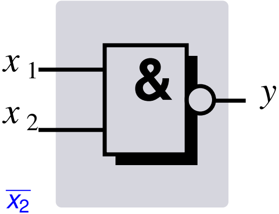
Rechengesetze der Aussagenlogik

$$
\begin{array}{rcll}
x \vee \overline{x} & = & 1 & \\
x \quad \overline{x} & = & 0 & \\[1em]
x \vee x & = & x & \\
x \quad x & = & x & \\[1em]
\overline{\overline{x}} & = & x & \ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{doppelte Negation} \\[1em]
\overline{x \vee y} & = & \overline{x} \quad \overline{y} & \ldots\ldots\ldots\ldots \text{De Morgansche Regel} \\
\overline{x \quad y} & = & \overline{x} \vee \overline{y} & \ldots\ldots\ldots\ldots \text{De Morgansche Regel} \\[1em]
x \vee y & = & y \vee x & \ldots\ldots\ldots\ldots\ldots\ldots \text{Kommutativität} \\
x \quad y & = & y \quad x & \ldots\ldots\ldots\ldots\ldots\ldots \text{Kommutativität} \\[1em]
a \vee (b \vee c) & = & (a \vee b) \vee c & \ldots\ldots\ldots\ldots\ldots \text{Assoziativität} \\
a \quad (b \quad c) & = & (a \quad b) \quad c & \ldots\ldots\ldots\ldots\ldots \text{Assoziativität} \\[1em]
a \quad (b \vee c) & = & a \quad b \vee a \quad c & \ldots\ldots\ldots\ldots\ldots \text{Distributivität} \\
a \vee (b \quad c) & = & (a \vee b) \quad (a \vee c) & \ldots\ldots\ldots\ldots \text{Distributivität}
\end{array}
$$

$\Longrightarrow$ Nachprüfbar durch Ausfüllen der Belegungstabellen

NAND-Verknüpfung als universelles Gatter

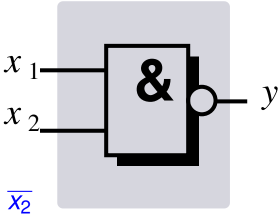
x_1	x_2	y
0	0	1
0	1	1
1	0	1
1	1	0



- Boolesche Notation: $y = \overline{x_1 \quad x_2} = \overline{x_1} \vee \overline{x_2}$
- C-Notation: `y = !x1 || !x2`
- Bei $x_1, x_2, y \in \{0, 1\}$ arithmetische Entsprechung: $y = 1 - x_1 \cdot x_2$
- Mithilfe von hintereinandergeschalteten NAND-Gattern lassen sich sowohl Negation als auch UND-Verknüpfung und ODER-Verknüpfung realisieren.
- NAND gilt deshalb (ebenso wie NOR) als universelles Gatter. Es sind kommerziell angebotene Schaltkreise verfügbar, die ausschließlich NAND-Gatter enthalten.

NAND-Verknüpfung als universelles Gatter

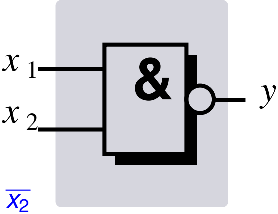
x_1	x_2	y
0	0	1
0	1	1
1	0	1
1	1	0



- Boolesche Notation: $y = \overline{x_1 \quad x_2} = \overline{x_1} \vee \overline{x_2}$
- C-Notation: `y = !x1 || !x2`
- Bei $x_1, x_2, y \in \{0, 1\}$ arithmetische Entsprechung: $y = 1 - x_1 \cdot x_2$
- Mithilfe von hintereinandergeschalteten NAND-Gattern lassen sich sowohl Negation als auch UND-Verknüpfung und ODER-Verknüpfung realisieren.
- NAND gilt deshalb (ebenso wie NOR) als universelles Gatter. Es sind kommerziell angebotene Schaltkreise verfügbar, die ausschließlich NAND-Gatter enthalten.

NAND-Verknüpfung als universelles Gatter

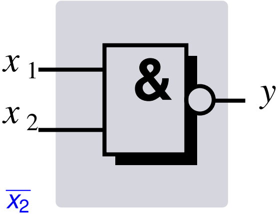
x_1	x_2	y
0	0	1
0	1	1
1	0	1
1	1	0



- Boolesche Notation: $y = \overline{x_1 \quad x_2} = \overline{x_1} \vee \overline{x_2}$
- C-Notation: `y = !x1 || !x2`
- Bei $x_1, x_2, y \in \{0, 1\}$ arithmetische Entsprechung: $y = 1 - x_1 \cdot x_2$
- Mithilfe von hintereinandergeschalteten NAND-Gattern lassen sich sowohl Negation als auch UND-Verknüpfung und ODER-Verknüpfung realisieren.
- NAND gilt deshalb (ebenso wie NOR) als universelles Gatter. Es sind kommerziell angebotene Schaltkreise verfügbar, die ausschließlich NAND-Gatter enthalten.

NAND-Verknüpfung als universelles Gatter

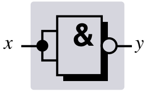
x_1	x_2	y
0	0	1
0	1	1
1	0	1
1	1	0



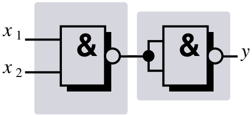
- Boolesche Notation: $y = \overline{x_1 \quad x_2} = \overline{x_1} \vee \overline{x_2}$
- C-Notation: `y = !x1 || !x2`
- Bei $x_1, x_2, y \in \{0, 1\}$ arithmetische Entsprechung: $y = 1 - x_1 \cdot x_2$
- Mithilfe von hintereinandergeschalteten NAND-Gattern lassen sich sowohl Negation als auch UND-Verknüpfung und ODER-Verknüpfung realisieren.
- NAND gilt deshalb (ebenso wie NOR) als universelles Gatter. Es sind kommerziell angebotene Schaltkreise verfügbar, die ausschließlich NAND-Gatter enthalten.

Simulation von Negation, UND, ODER durch NAND

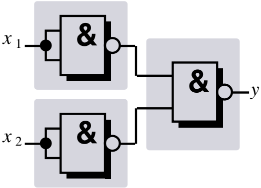
Negation	**UND**	**ODER**
$y = \overline{x}$ $= \overline{x \quad x}$	$y = x_1 \quad x_2$ $= \overline{\overline{x_1 \quad x_2}}$	$y = x_1 \vee x_2$ $= \overline{\overline{x_1 \vee x_2}}$ $= \overline{\overline{x_1} \quad \overline{x_2}}$



NICHT



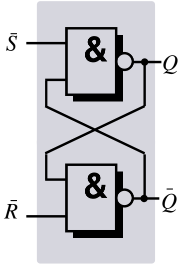
UND



ODER

NAND-basiertes RS-Flip-Flop als 1-Bit-RAM-Speicher

$\overline{S}$	$\overline{R}$	Q



NAND

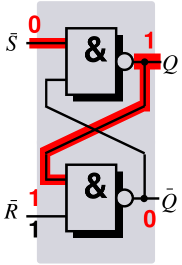
x_1	x_2	y
0	0	1
0	1	1
1	0	1
1	1	0

0 am Eingang
setzt sich durch

NAND-basiertes RS-Flip-Flop als 1-Bit-RAM-Speicher

$\overline{S}$	$\overline{R}$	Q	
0	1	1	(Set)

- Signal 0 am Eingang $\overline{S}$ setzt das Flip-Flop auf $Q = 1$ (Set)



NAND

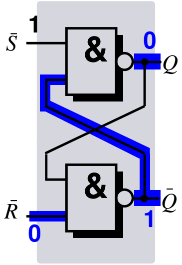
x_1	x_2	y
0	0	1
0	1	1
1	0	1
1	1	0

0 am Eingang setzt sich durch

NAND-basiertes RS-Flip-Flop als 1-Bit-RAM-Speicher

$\overline{S}$	$\overline{R}$	Q	
0	1	1	(Set)
1	0	0	(Reset)

- Signal 0 am Eingang $\overline{S}$ setzt das Flip-Flop auf $Q = 1$ (Set)
- Signal 0 am Eingang $\overline{R}$ setzt das Flip-Flop auf $Q = 0$ (Reset)



NAND

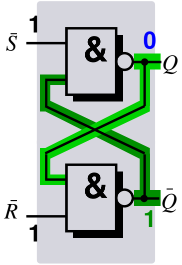
x_1	x_2	y
0	0	1
0	1	1
1	0	1
1	1	0

0 am Eingang setzt sich durch

NAND-basiertes RS-Flip-Flop als 1-Bit-RAM-Speicher

$\overline{S}$	$\overline{R}$	Q	
0	1	1	(Set)
1	0	0	(Reset)
1	1	Q	(Hold)

- Signal 0 am Eingang $\overline{S}$ setzt das Flip-Flop auf $Q = 1$ (Set)
- Signal 0 am Eingang $\overline{R}$ setzt das Flip-Flop auf $Q = 0$ (Reset)
- Beide Eingänge 1: Flip-Flop speichert zuvor gesetztes Bit, bis erneutes Set oder Reset erfolgt



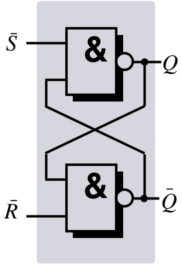
NAND

x_1	x_2	y
0	0	1
0	1	1
1	0	1
1	1	0

0 am Eingang setzt sich durch

NAND-basiertes RS-Flip-Flop als 1-Bit-RAM-Speicher

$\overline{S}$	$\overline{R}$	Q	
0	1	1	(Set)
1	0	0	(Reset)
1	1	Q	(Hold)
0	0	verboten	



- Signal 0 am Eingang $\overline{S}$ setzt das Flip-Flop auf $Q = 1$ (Set)
- Signal 0 am Eingang $\overline{R}$ setzt das Flip-Flop auf $Q = 0$ (Reset)
- Beide Eingänge 1: Flip-Flop speichert zuvor gesetztes Bit, bis erneutes Set oder Reset erfolgt
- Beide Eingänge 0: Bit nicht gleichzeitig auf 1 und auf 0 setzbar, daher verboten ($Q = 1$ und $\overline{Q} = 1$)

NAND

x_1	x_2	y
0	0	1
0	1	1
1	0	1
1	1	0

0 am Eingang setzt sich durch

Schaltbelegungstabellen und Schaltnetze

- *n* Signaleingänge x_0 bis x_{n-1}, jeder davon zeitlich variierend mit 0 oder mit 1 belegt
- ein oder mehrere Signalausgänge, jeder davon als *Boolesche Funktion* von den Signaleingängen abhängig

Beispiel: Test auf Signalgleichheit zweier Eingänge (Äquivalenz)

x_1	x_0	y
0	0	1
0	1	0
1	0	0
1	1	1

$$y = x_1 \ \ x_0 \vee \overline{x_1} \ \ \overline{x_0}$$

Boolesche Funktion direkt aus Schaltbelegungstabelle ablesbar („disjunktive Normalform“)

Schaltbelegungstabellen und Schaltnetze

- n Signaleingänge x_0 bis x_{n-1}, jeder davon zeitlich variierend mit 0 oder mit 1 belegt
- ein oder mehrere Signalausgänge, jeder davon als *Boolesche Funktion* von den Signaleingängen abhängig
- Logikschaltung (real oder in Simulation) implementiert diese Abhängigkeit in Form eines Schaltnetzes

Beispiel: Test auf Signalgleichheit zweier Eingänge (Äquivalenz)

x_1	x_0	y
0	0	1
0	1	0
1	0	0
1	1	1

$$y = x_1 \;\; x_0 \vee \overline{x_1} \;\; \overline{x_0}$$

Boolesche Funktion direkt aus Schaltbelegungstabelle ablesbar („disjunktive Normalform“)

Schaltbelegungstabellen und Schaltnetze

- n Signaleingänge x_0 bis x_{n-1}, jeder davon zeitlich variierend mit 0 oder mit 1 belegt
- ein oder mehrere Signalausgänge, jeder davon als *Boolesche Funktion* von den Signaleingängen abhängig
- Logikschaltung (real oder in Simulation) implementiert diese Abhängigkeit in Form eines Schaltnetzes
- Schaltnetze haben kein Gedächtnis, sie mappen einfach Eingangssignale in Ausgangssignale

Beispiel: Test auf Signalgleichheit zweier Eingänge (Äquivalenz)

x_1	x_0	y
0	0	1
0	1	0
1	0	0
1	1	1

$$y = x_1 \ \ x_0 \vee \overline{x_1} \ \ \overline{x_0}$$

Boolesche Funktion direkt aus Schaltbelegungstabelle ablesbar („disjunktive Normalform“)

Schaltbelegungstabellen und Schaltnetze

- n Signaleingänge x_0 bis x_{n-1}, jeder davon zeitlich variierend mit 0 oder mit 1 belegt
- ein oder mehrere Signalausgänge, jeder davon als *Boolesche Funktion* von den Signaleingängen abhängig
- Logikschaltung (real oder in Simulation) implementiert diese Abhängigkeit in Form eines Schaltnetzes
- Schaltnetze haben kein Gedächtnis, sie mappen einfach Eingangssignale in Ausgangssignale
- Kommen Speicherelemente (Flip-Flops) hinzu, spricht man von Schaltwerken. Dann können Ausgangssignale auch von ihren früheren Werten (vorheriger Systemzustand) abhängig sein

Beispiel: Test auf Signalgleichheit zweier Eingänge (Äquivalenz)

x_1	x_0	y
0	0	1
0	1	0
1	0	0
1	1	1

$$y = x_1 \;\; x_0 \vee \overline{x_1} \;\; \overline{x_0}$$

Boolesche Funktion direkt aus Schaltbelegungstabelle ablesbar („disjunktive Normalform“)

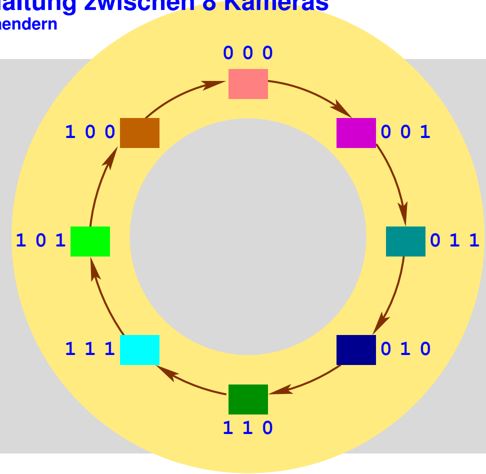
Effizientes Multiplexing mittels Logikschaltung

Umlaufende Signalumschaltung zwischen 8 Kameras

bei jedem Umschalten nur EIN Bit veraendern

Kamera 0	000	000
Kamera 1	001	001
Kamera 2	010	011
Kamera 3	011	010
Kamera 4	100	110
Kamera 5	101	111
Kamera 6	110	101
Kamera 7	111	100

3–Bit–Graycode



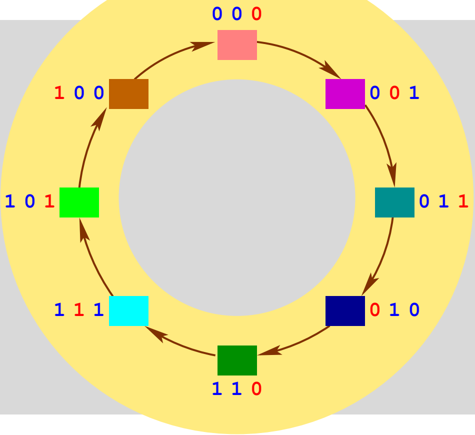
Effizientes Multiplexing mittels Logikschaltung

Umlaufende Signalumschaltung zwischen 8 Kameras

bei jedem Umschalten nur EIN Bit veraendern

Kamera 0	000	000
Kamera 1	001	001
Kamera 2	010	011
Kamera 3	011	010
Kamera 4	100	110
Kamera 5	101	111
Kamera 6	110	101
Kamera 7	111	100

3–Bit–Graycode



3-Bit-Graycode - effizientes Zählen in Einerschritten

Umlaufender Zähler mod 8. Pro Zählschritt ändert sich stets nur ein Bit in der Ausgabe

	Eingabe			Ausgabe		
x_{dez}	x_2	x_1	x_0	z_2	z_1	z_0
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	1
3	0	1	1	0	1	0
4	1	0	0	1	1	0
5	1	0	1	1	1	1
6	1	1	0	1	0	1
7	1	1	1	1	0	0

3-Bit-Graycode - effizientes Zählen in Einerschritten

Umlaufender Zähler mod 8. Pro Zählschritt ändert sich stets nur ein Bit in der Ausgabe

	Eingabe			Ausgabe		
x_{dez}	x_2	x_1	x_0	z_2	z_1	z_0
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	1
3	0	1	1	0	1	0
4	1	0	0	1	1	0
5	1	0	1	1	1	1
6	1	1	0	1	0	1
7	1	1	1	1	0	0

$$z_0 = \overline{x_2}\ \overline{x_1}\ x_0 \vee \overline{x_2}\ x_1 \overline{x_0} \vee x_2\ \overline{x_1}\ x_0 \vee x_2\ x_1\ \overline{x_0}$$

3-Bit-Graycode - effizientes Zählen in Einerschritten

Umlaufender Zähler mod 8. Pro Zählschritt ändert sich stets nur ein Bit in der Ausgabe

	Eingabe			Ausgabe		
x_{dez}	x_2	x_1	x_0	z_2	z_1	z_0
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	1
3	0	1	1	0	1	0
4	1	0	0	1	1	0
5	1	0	1	1	1	1
6	1	1	0	1	0	1
7	1	1	1	1	0	0

$$z_0 = \overline{x_2}\;\overline{x_1}\;x_0 \vee \overline{x_2}\;x_1\overline{x_0} \vee x_2\;\overline{x_1}\;x_0 \vee x_2\;x_1\;\overline{x_0}$$

$$z_1 = \overline{x_2}\;x_1\;\overline{x_0} \vee \overline{x_2}\;x_1\;x_0 \vee x_2\;\overline{x_1}\;\overline{x_0} \vee x_2\;\overline{x_1}\;x_0$$

3-Bit-Graycode - effizientes Zählen in Einerschritten

Umlaufender Zähler mod 8. Pro Zählschritt ändert sich stets nur ein Bit in der Ausgabe

	Eingabe			Ausgabe		
x_{dez}	x_2	x_1	x_0	z_2	z_1	z_0
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	1
3	0	1	1	0	1	0
4	1	0	0	1	1	0
5	1	0	1	1	1	1
6	1	1	0	1	0	1
7	1	1	1	1	0	0

$$
\begin{aligned}
z_0 &= \overline{x_2}\ \overline{x_1}\ x_0 \vee \overline{x_2}\ x_1 \overline{x_0} \vee x_2\ \overline{x_1}\ x_0 \vee x_2\ x_1\ \overline{x_0} \\
z_1 &= \overline{x_2}\ x_1\ \overline{x_0} \vee \overline{x_2}\ x_1\ x_0 \vee x_2\ \overline{x_1}\ \overline{x_0} \vee x_2\ \overline{x_1}\ x_0 \\
z_2 &= x_2\ \overline{x_1}\ \overline{x_0} \vee x_2\ \overline{x_1}\ x_0 \vee x_2\ x_1\ \overline{x_0} \vee x_2\ x_1\ x_0
\end{aligned}
$$

3-Bit-Graycode - effizientes Zählen in Einerschritten

Umlaufender Zähler mod 8. Pro Zählschritt ändert sich stets nur ein Bit in der Ausgabe

	Eingabe			Ausgabe		
x_{dez}	x_2	x_1	x_0	z_2	z_1	z_0
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	1
3	0	1	1	0	1	0
4	1	0	0	1	1	0
5	1	0	1	1	1	1
6	1	1	0	1	0	1
7	1	1	1	1	0	0

$$
\begin{aligned}
z_0 &= \overline{x_2}\,\overline{x_1}\,x_0 \vee \overline{x_2}\,x_1\overline{x_0} \vee x_2\,\overline{x_1}\,x_0 \vee x_2\,x_1\,\overline{x_0} \\
z_1 &= \overline{x_2}\,x_1\,\overline{x_0} \vee \overline{x_2}\,x_1\,x_0 \vee x_2\,\overline{x_1}\,\overline{x_0} \vee x_2\,\overline{x_1}\,x_0 \\
z_2 &= x_2\,\overline{x_1}\,\overline{x_0} \vee x_2\,\overline{x_1}\,x_0 \vee x_2\,x_1\,\overline{x_0} \vee x_2\,x_1\,x_0
\end{aligned}
$$

$\Longrightarrow$ Lassen sich die Booleschen Funktionen noch *vereinfachen*?

3-Bit-Graycode - effizientes Zählen in Einerschritten

Umlaufender Zähler mod 8. Pro Zählschritt ändert sich stets nur ein Bit in der Ausgabe

	Eingabe			Ausgabe		
x_{dez}	x_2	x_1	x_0	z_2	z_1	z_0
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	1
3	0	1	1	0	1	0
4	1	0	0	1	1	0
5	1	0	1	1	1	1
6	1	1	0	1	0	1
7	1	1	1	1	0	0

$$
\begin{aligned}
z_0 &= \overline{x_2}\ \overline{x_1}\ x_0 \vee \overline{x_2}\ x_1 \overline{x_0} \vee x_2\ \overline{x_1}\ x_0 \vee x_2\ x_1\ \overline{x_0} \\
z_1 &= \overline{x_2}\ x_1\ \overline{x_0} \vee \overline{x_2}\ x_1\ x_0 \vee x_2\ \overline{x_1}\ \overline{x_0} \vee x_2\ \overline{x_1}\ x_0 \\
z_2 &= x_2\ \overline{x_1}\ \overline{x_0} \vee x_2\ \overline{x_1}\ x_0 \vee x_2\ x_1\ \overline{x_0} \vee x_2\ x_1\ x_0
\end{aligned}
$$

$\Longrightarrow$ Lassen sich die Booleschen Funktionen noch *vereinfachen*? **Ja!**

Vereinfachung von Schaltfunktionen durch Karnaugh-Optimierung

Idee

$$\begin{aligned} a\ b\ c\ d \vee a\ b\ c\ \overline{d} &= a\ b\ c\ (d \vee \overline{d}) \\ &= a\ b\ c\ 1 \\ &= a\ b\ c \end{aligned}$$

Vereinfachung von Schaltfunktionen durch Karnaugh-Optimierung

Idee

$$
\begin{aligned}
a\ b\ c\ d \vee a\ b\ c\ \overline{d} &= a\ b\ c\ (d \vee \overline{d}) \\
&= a\ b\ c\ 1 \\
&= a\ b\ c
\end{aligned}
$$

- Wie kann man in einer Booleschen Funktion *jede* Stelle erkennen, an der sich geschickt eine 1 ausklammern lässt, so dass die Funktionsgleichung einfacher wird, ihr Werteverlauf aber unverändert bleibt

Vereinfachung von Schaltfunktionen durch Karnaugh-Optimierung

Idee

$$\begin{aligned} a\ b\ c\ d \vee a\ b\ c\ \overline{d} &= a\ b\ c\ (d \vee \overline{d}) \\ &= a\ b\ c\ 1 \\ &= a\ b\ c \end{aligned}$$

- Wie kann man in einer Booleschen Funktion *jede* Stelle erkennen, an der sich geschickt eine 1 ausklammern lässt, so dass die Funktionsgleichung einfacher wird, ihr Werteverlauf aber unverändert bleibt
- Dazu trägt man die Funktionswerte in ein spezielles Zeilen-Spalten-Schema ein, so dass unmittelbar benachbarte Einsen (sowohl horizontal wie auch vertikal) die Stellen anzeigen, an denen vereinfacht werden kann.

Vereinfachung von Schaltfunktionen durch Karnaugh-Optimierung

Idee

$$\begin{aligned} a\,b\,c\,d \vee a\,b\,c\,\overline{d} &= a\,b\,c\,(d \vee \overline{d}) \\ &= a\,b\,c\,1 \\ &= a\,b\,c \end{aligned}$$

- Wie kann man in einer Booleschen Funktion *jede* Stelle erkennen, an der sich geschickt eine 1 ausklammern lässt, so dass die Funktionsgleichung einfacher wird, ihr Werteverlauf aber unverändert bleibt
- Dazu trägt man die Funktionswerte in ein spezielles Zeilen-Spalten-Schema ein, so dass unmittelbar benachbarte Einsen (sowohl horizontal wie auch vertikal) die Stellen anzeigen, an denen vereinfacht werden kann.
- Zeilen-Spalten-Schema so aufgebaut, dass von Zeile zu Zeile und von Spalte zu Spalte genau eine Variable negiert wird

3-Bit-Graycode - Vereinfachung der Schaltfunktionen

	Eingabe			Ausgabe		
x_{dez}	x_2	x_1	x_0	z_2	z_1	z_0
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	1
3	0	1	1	0	1	0
4	1	0	0	1	1	0
5	1	0	1	1	1	1
6	1	1	0	1	0	1
7	1	1	1	1	0	0

Schaltfunktion für z_0

	x_2	$\overline{x_2}$
$x_1\ x_0$		
$x_1\ \overline{x_0}$		
$\overline{x_1}\ \overline{x_0}$		
$\overline{x_1}\ x_0$		

3-Bit-Graycode - Vereinfachung der Schaltfunktionen

	Eingabe			Ausgabe		
x_{dez}	x_2	x_1	x_0	z_2	z_1	z_0
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	1
3	0	1	1	0	1	0
4	1	0	0	1	1	0
5	1	0	1	1	1	1
6	1	1	0	1	0	1
7	1	1	1	1	0	0

Schaltfunktion für z_0

	x_2	$\overline{x_2}$
$x_1\ x_0$		
$x_1\ \overline{x_0}$		
$\overline{x_1}\ \overline{x_0}$		
$\overline{x_1}\ x_0$		1

3-Bit-Graycode - Vereinfachung der Schaltfunktionen

	Eingabe			Ausgabe		
x_{dez}	x_2	x_1	x_0	z_2	z_1	z_0
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	1
3	0	1	1	0	1	0
4	1	0	0	1	1	0
5	1	0	1	1	1	1
6	1	1	0	1	0	1
7	1	1	1	1	0	0

Schaltfunktion für z_0

	x_2	$\overline{x_2}$
$x_1\ x_0$		
$x_1\ \overline{x_0}$		1
$\overline{x_1}\ \overline{x_0}$		
$\overline{x_1}\ x_0$		1

3-Bit-Graycode - Vereinfachung der Schaltfunktionen

	Eingabe			Ausgabe		
x_{dez}	x_2	x_1	x_0	z_2	z_1	z_0
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	1
3	0	1	1	0	1	0
4	1	0	0	1	1	0
5	1	0	1	1	1	1
6	1	1	0	1	0	1
7	1	1	1	1	0	0

Schaltfunktion für z_0

	x_2	$\overline{x_2}$
$x_1\ x_0$		
$x_1\ \overline{x_0}$		1
$\overline{x_1}\ \overline{x_0}$		
$\overline{x_1}\ x_0$	1	1

3-Bit-Graycode - Vereinfachung der Schaltfunktionen

	Eingabe			Ausgabe		
x_{dez}	x_2	x_1	x_0	z_2	z_1	z_0
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	1
3	0	1	1	0	1	0
4	1	0	0	1	1	0
5	1	0	1	1	1	1
6	1	1	0	1	0	1
7	1	1	1	1	0	0

Schaltfunktion für z_0

	x_2	$\overline{x_2}$
$x_1\ x_0$		
$x_1\ \overline{x_0}$	1	1
$\overline{x_1}\ \overline{x_0}$		
$\overline{x_1}\ x_0$	1	1

3-Bit-Graycode - Vereinfachung der Schaltfunktionen

	Eingabe			Ausgabe		
x_{dez}	x_2	x_1	x_0	z_2	z_1	z_0
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	1
3	0	1	1	0	1	0
4	1	0	0	1	1	0
5	1	0	1	1	1	1
6	1	1	0	1	0	1
7	1	1	1	1	0	0

Schaltfunktion für z_0

	x_2	$\overline{x_2}$
$x_1\ x_0$	0	0
$x_1\ \overline{x_0}$	1	1
$\overline{x_1}\ \overline{x_0}$	0	0
$\overline{x_1}\ x_0$	1	1

3-Bit-Graycode - Vereinfachung der Schaltfunktionen

	Eingabe			Ausgabe		
x_{dez}	x_2	x_1	x_0	z_2	z_1	z_0
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	1
3	0	1	1	0	1	0
4	1	0	0	1	1	0
5	1	0	1	1	1	1
6	1	1	0	1	0	1
7	1	1	1	1	0	0

Schaltfunktion für z_0

	x_2	$\overline{x_2}$
$x_1 \; x_0$	0	0
$x_1 \; \overline{x_0}$	1	1
$\overline{x_1} \; \overline{x_0}$	0	0
$\overline{x_1} \; x_0$	1	1

$$z_0 = x_1 \, \overline{x_0} \vee \overline{x_1} \, x_0$$

- möglichst große *Blöcke* aus Einsen bilden (2er, 4er, 8er, ...)
- In jedem Block fallen die Variablen raus, die sowohl negiert als auch nicht negiert vorkommen
- Blöcke dürfen sich überlagern
- Blöcke dürfen zeilen- und/oder spaltenüberspannend sein

3-Bit-Graycode - Vereinfachung der Schaltfunktionen

	Eingabe			Ausgabe		
x_{dez}	x_2	x_1	x_0	z_2	z_1	z_0
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	1
3	0	1	1	0	1	0
4	1	0	0	1	1	0
5	1	0	1	1	1	1
6	1	1	0	1	0	1
7	1	1	1	1	0	0

Schaltfunktion für z_1

	x_2	$\overline{x_2}$
$x_1\ x_0$	0	1
$x_1\ \overline{x_0}$	0	1
$\overline{x_1}\ \overline{x_0}$	1	0
$\overline{x_1}\ x_0$	1	0

3-Bit-Graycode - Vereinfachung der Schaltfunktionen

	Eingabe			Ausgabe		
x_{dez}	x_2	x_1	x_0	z_2	z_1	z_0
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	1
3	0	1	1	0	1	0
4	1	0	0	1	1	0
5	1	0	1	1	1	1
6	1	1	0	1	0	1
7	1	1	1	1	0	0

Schaltfunktion für z_1

	x_2	$\overline{x_2}$
$x_1\ x_0$	0	1
$x_1\ \overline{x_0}$	0	1
$\overline{x_1}\ \overline{x_0}$	1	0
$\overline{x_1}\ x_0$	1	0

$$z_1 = \overline{x_2}\ x_1 \vee x_2\ \overline{x_1}$$

3-Bit-Graycode - Vereinfachung der Schaltfunktionen

	Eingabe			Ausgabe		
x_{dez}	x_2	x_1	x_0	z_2	z_1	z_0
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	1
3	0	1	1	0	1	0
4	1	0	0	1	1	0
5	1	0	1	1	1	1
6	1	1	0	1	0	1
7	1	1	1	1	0	0

Schaltfunktion für z_1

	x_2	$\overline{x_2}$
$x_1\ x_0$	0	1
$x_1\ \overline{x_0}$	0	1
$\overline{x_1}\ \overline{x_0}$	1	0
$\overline{x_1}\ x_0$	1	0

$$z_1 = \overline{x_2}\ x_1 \vee x_2\ \overline{x_1}$$

Schaltfunktion für z_2

	x_2	$\overline{x_2}$
$x_1\ x_0$	1	0
$x_1\ \overline{x_0}$	1	0
$\overline{x_1}\ \overline{x_0}$	1	0
$\overline{x_1}\ x_0$	1	0

3-Bit-Graycode - Vereinfachung der Schaltfunktionen

	Eingabe			Ausgabe		
x_{dez}	x_2	x_1	x_0	z_2	z_1	z_0
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	1
3	0	1	1	0	1	0
4	1	0	0	1	1	0
5	1	0	1	1	1	1
6	1	1	0	1	0	1
7	1	1	1	1	0	0

Schaltfunktion für z_1

	x_2	$\overline{x_2}$
$x_1\ x_0$	0	1
$x_1\ \overline{x_0}$	0	1
$\overline{x_1}\ \overline{x_0}$	1	0
$\overline{x_1}\ x_0$	1	0

$$z_1 = \overline{x_2}\ x_1 \vee x_2\ \overline{x_1}$$

Schaltfunktion für z_2

	x_2	$\overline{x_2}$
$x_1\ x_0$	1	0
$x_1\ \overline{x_0}$	1	0
$\overline{x_1}\ \overline{x_0}$	1	0
$\overline{x_1}\ x_0$	1	0

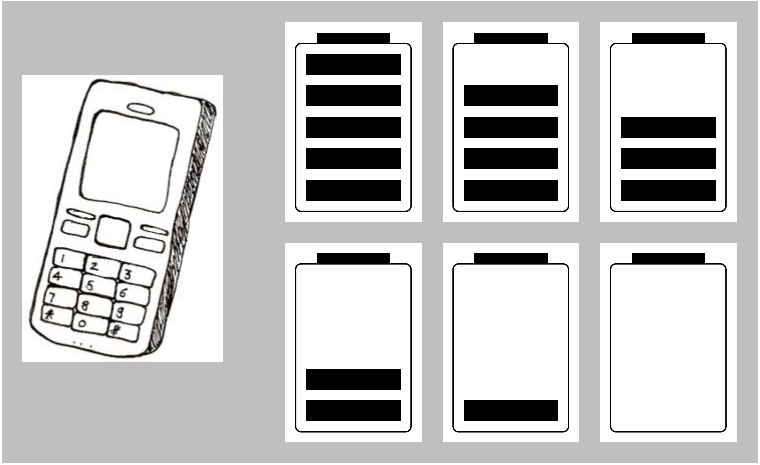
$$z_2 = x_2$$

C-Programm Berechnung 3-Bit-Graycode (graycode3.c)

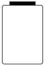
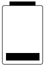
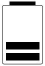
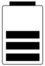
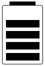
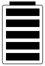
```
#include <stdio.h>

int main(void)
{
  unsigned char x0, x1, x2, z0, z1, z2;

  for(x2 = 0; x2 <= 1; x2++)
  {
    for(x1 = 0; x1 <= 1; x1++)
    {
      for(x0 = 0; x0 <= 1; x0++)
      {
        z0 = x1 && !x0 || !x1 && x0;
        z1 = !x2 && x1 || x2 && !x1;
        z2 = x2;
        printf("%u | %u %u %u | %u %u %u\n",
               (x2 << 2)+(x1 << 1)+x0,
               x2, x1, x0, z2, z1, z0);
      }
    }
  }
  return 0;
}
```

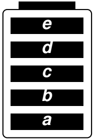
```
0 | 0 0 0 | 0 0 0
1 | 0 0 1 | 0 0 1
2 | 0 1 0 | 0 1 1
3 | 0 1 1 | 0 1 0
4 | 1 0 0 | 1 1 0
5 | 1 0 1 | 1 1 1
6 | 1 1 0 | 1 0 1
7 | 1 1 1 | 1 0 0
```

5-Balken-Anzeige Handy-Akkuladestand / Netzstärke

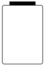
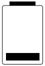
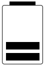
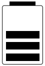
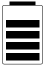
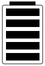


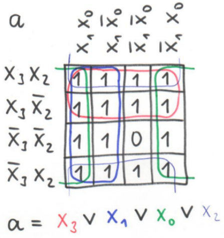
Schaltbelegungstabelle $0 \dots 15 \mapsto$ Balkendarstellung

	Eingabe				Ausgabe					
x_{dez}	x_3	x_2	x_1	x_0	a	b	c	d	e	
0	0	0	0	0	0	0	0	0	0	
1	0	0	0	1	1	0	0	0	0	
2	0	0	1	0	1	0	0	0	0	
3	0	0	1	1	1	0	0	0	0	
4	0	1	0	0	1	1	0	0	0	
5	0	1	0	1	1	1	0	0	0	
6	0	1	1	0	1	1	0	0	0	
7	0	1	1	1	1	1	1	0	0	
8	1	0	0	0	1	1	1	0	0	
9	1	0	0	1	1	1	1	0	0	
10	1	0	1	0	1	1	1	1	0	
11	1	0	1	1	1	1	1	1	0	
12	1	1	0	0	1	1	1	1	0	
13	1	1	0	1	1	1	1	1	1	
14	1	1	1	0	1	1	1	1	1	
15	1	1	1	1	1	1	1	1	1	

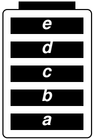


Schaltbelegungstabelle 0 . . . 15 $\mapsto$ Balkendarstellung

	Eingabe				Ausgabe					
x_{dez}	x_3	x_2	x_1	x_0	a	b	c	d	e	
0	0	0	0	0	0	0	0	0	0	
1	0	0	0	1	1	0	0	0	0	
2	0	0	1	0	1	0	0	0	0	
3	0	0	1	1	1	0	0	0	0	
4	0	1	0	0	1	1	0	0	0	
5	0	1	0	1	1	1	0	0	0	
6	0	1	1	0	1	1	0	0	0	
7	0	1	1	1	1	1	1	0	0	
8	1	0	0	0	1	1	1	0	0	
9	1	0	0	1	1	1	1	0	0	
10	1	0	1	0	1	1	1	1	0	
11	1	0	1	1	1	1	1	1	0	
12	1	1	0	0	1	1	1	1	0	
13	1	1	0	1	1	1	1	1	1	
14	1	1	1	0	1	1	1	1	1	
15	1	1	1	1	1	1	1	1	1	

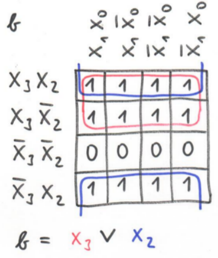


$$a = x_3 \vee x_2 \vee x_1 \vee x_0$$

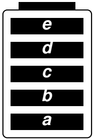


Schaltbelegungstabelle 0 . . . 15 $\mapsto$ Balkendarstellung

	Eingabe				Ausgabe					
x_{dez}	x_3	x_2	x_1	x_0	a	b	c	d	e	
0	0	0	0	0	0	0	0	0	0	
1	0	0	0	1	1	0	0	0	0	
2	0	0	1	0	1	0	0	0	0	
3	0	0	1	1	1	0	0	0	0	
4	0	1	0	0	1	1	0	0	0	
5	0	1	0	1	1	1	0	0	0	
6	0	1	1	0	1	1	0	0	0	
7	0	1	1	1	1	1	1	0	0	
8	1	0	0	0	1	1	1	0	0	
9	1	0	0	1	1	1	1	0	0	
10	1	0	1	0	1	1	1	1	0	
11	1	0	1	1	1	1	1	1	0	
12	1	1	0	0	1	1	1	1	0	
13	1	1	0	1	1	1	1	1	1	
14	1	1	1	0	1	1	1	1	1	
15	1	1	1	1	1	1	1	1	1	

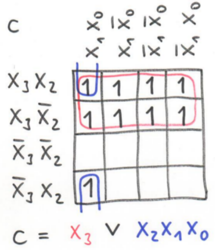


$b = x_3 \lor x_2$

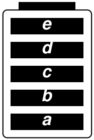


Schaltbelegungstabelle 0 . . . 15 $\mapsto$ Balkendarstellung

	Eingabe				Ausgabe					
x_{dez}	x_3	x_2	x_1	x_0	a	b	c	d	e	
0	0	0	0	0	0	0	0	0	0	
1	0	0	0	1	1	0	0	0	0	
2	0	0	1	0	1	0	0	0	0	
3	0	0	1	1	1	0	0	0	0	
4	0	1	0	0	1	1	0	0	0	
5	0	1	0	1	1	1	0	0	0	
6	0	1	1	0	1	1	0	0	0	
7	0	1	1	1	1	1	1	0	0	
8	1	0	0	0	1	1	1	0	0	
9	1	0	0	1	1	1	1	0	0	
10	1	0	1	0	1	1	1	1	0	
11	1	0	1	1	1	1	1	1	0	
12	1	1	0	0	1	1	1	1	0	
13	1	1	0	1	1	1	1	1	1	
14	1	1	1	0	1	1	1	1	1	
15	1	1	1	1	1	1	1	1	1	

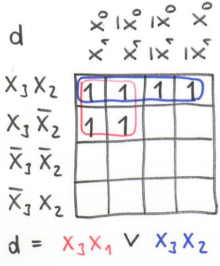


$$c = x_3 \vee x_2\ x_1\ x_0$$

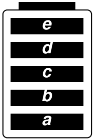


Schaltbelegungstabelle $0 \ldots 15 \mapsto$ Balkendarstellung

	Eingabe				Ausgabe					
x_{dez}	x_3	x_2	x_1	x_0	a	b	c	d	e	
0	0	0	0	0	0	0	0	0	0	
1	0	0	0	1	1	0	0	0	0	
2	0	0	1	0	1	0	0	0	0	
3	0	0	1	1	1	0	0	0	0	
4	0	1	0	0	1	1	0	0	0	
5	0	1	0	1	1	1	0	0	0	
6	0	1	1	0	1	1	0	0	0	
7	0	1	1	1	1	1	1	0	0	
8	1	0	0	0	1	1	1	0	0	
9	1	0	0	1	1	1	1	0	0	
10	1	0	1	0	1	1	1	1	0	
11	1	0	1	1	1	1	1	1	0	
12	1	1	0	0	1	1	1	1	0	
13	1	1	0	1	1	1	1	1	1	
14	1	1	1	0	1	1	1	1	1	
15	1	1	1	1	1	1	1	1	1	

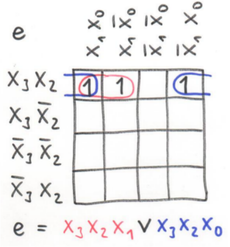


$$d = x_3 \; x_1 \vee x_3 \; x_2$$

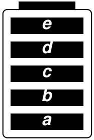


Schaltbelegungstabelle 0 . . . 15 $\mapsto$ Balkendarstellung

	Eingabe				Ausgabe					
x_{dez}	x_3	x_2	x_1	x_0	a	b	c	d	e	
0	0	0	0	0	0	0	0	0	0	
1	0	0	0	1	1	0	0	0	0	
2	0	0	1	0	1	0	0	0	0	
3	0	0	1	1	1	0	0	0	0	
4	0	1	0	0	1	1	0	0	0	
5	0	1	0	1	1	1	0	0	0	
6	0	1	1	0	1	1	0	0	0	
7	0	1	1	1	1	1	1	0	0	
8	1	0	0	0	1	1	1	0	0	
9	1	0	0	1	1	1	1	0	0	
10	1	0	1	0	1	1	1	1	0	
11	1	0	1	1	1	1	1	1	0	
12	1	1	0	0	1	1	1	1	0	
13	1	1	0	1	1	1	1	1	1	
14	1	1	1	0	1	1	1	1	1	
15	1	1	1	1	1	1	1	1	1	



$$e = x_3\ x_2\ x_1 \lor x_3\ x_2\ x_0$$



Implementierung in C (abcde.c)

```c
#include <stdio.h>

int main(void)
{
  unsigned char x3, x2, x1, x0, a, b, c, d, e;
  int x;

  printf("Bitte Zahl zwischen 0 und 15 eingeben: ");
  scanf("%d", &x);
  if ((x >= 0) && (x <= 15))
  {
    x0 = (unsigned char) x & 1;
    x1 = (unsigned char) (x >> 1) & 1;
    x2 = (unsigned char) (x >> 2) & 1;
    x3 = (unsigned char) (x >> 3) & 1;

    a = x1 || x3 || x0 || x2;
    b = x3 || x2;
    c = x3 || x2 && x1 && x0;
    d = x3 && x1 || x3 && x2;
    e = x3 && x2 && x1 || x3 && x2 && x0;
    printf("\n+-------+\n");
    if (e) {printf("| ***** |\n");} else {printf("|       |\n");}
    if (d) {printf("| ***** |\n");} else {printf("|       |\n");}
    if (c) {printf("| ***** |\n");} else {printf("|       |\n");}
    if (b) {printf("| ***** |\n");} else {printf("|       |\n");}
    if (a) {printf("| ***** |\n");} else {printf("|       |\n");}
    printf ("+-------+\n");
  }
  return 0;
}
```

```
Bitte Zahl zwischen 0 und 15 eingeben: 7

+-------+
|       |
|       |
| ***** |
| ***** |
| ***** |
+-------+
```

Bitoperationen zur Bitverschiebung und Bitmaskierung

- Die aussagenlogischen Operatoren `!`, `&&` und `||` beziehen stets einen gesamten Ganzzahlwert ein und liefern `0` bzw. `1` als Ergebnis (Beispiel: `142 && 27` ergibt `1`)

Bitoperationen zur Bitverschiebung und Bitmaskierung

- Die aussagenlogischen Operatoren `!`, `&&` und `||` beziehen stets einen gesamten Ganzzahlwert ein und liefern `0` bzw. `1` als Ergebnis (Beispiel: `142 && 27` ergibt `1`)
- *Bitoperationen* hingegen betrachten die *einzelnen Bits* der Binärdarstellung einer Ganzzahl *unabhängig voneinander* und wirken gesondert auf jedem einzelnen Bit.

  Beispiel Bitkonjunktion:

```
     142   10001110
      27   00011011
---------------------
142 & 27   00001010
```

  $\longrightarrow$ Dezimalzahl 10 als Ergebnis

Bitoperationen zur Bitverschiebung und Bitmaskierung

- Die aussagenlogischen Operatoren `!`, `&&` und `||` beziehen stets einen gesamten Ganzzahlwert ein und liefern `0` bzw. `1` als Ergebnis (Beispiel: `142 && 27` ergibt `1`)
- *Bitoperationen* hingegen betrachten die *einzelnen Bits* der Binärdarstellung einer Ganzzahl *unabhängig voneinander* und wirken gesondert auf jedem einzelnen Bit.
  Beispiel Bitkonjunktion:

```
      142   10001110
       27   00011011
--------------------
 142 & 27   00001010
```

  $\longrightarrow$ Dezimalzahl 10 als Ergebnis
- Bitoperationen werden sehr schnell ausgeführt

Bitoperationen zur Bitverschiebung und Bitmaskierung

- Die aussagenlogischen Operatoren `!`, `&&` und `||` beziehen stets einen gesamten Ganzzahlwert ein und liefern `0` bzw. `1` als Ergebnis (Beispiel: `142 && 27` ergibt `1`)
- *Bitoperationen* hingegen betrachten die *einzelnen Bits* der Binärdarstellung einer Ganzzahl *unabhängig voneinander* und wirken gesondert auf jedem einzelnen Bit.
  Beispiel Bitkonjunktion:

```
      142   10001110
       27   00011011
----------------------
 142 & 27   00001010
```

  $\longrightarrow$ Dezimalzahl 10 als Ergebnis
- Bitoperationen werden sehr schnell ausgeführt
- Durch Bitoperationen lassen sich bestimmte arithmetische Berechnungen deutlich beschleunigen

Bitoperationen zur Bitverschiebung und Bitmaskierung

- Die aussagenlogischen Operatoren `!`, `&&` und `||` beziehen stets einen gesamten Ganzzahlwert ein und liefern `0` bzw. `1` als Ergebnis (Beispiel: `142 && 27` ergibt `1`)
- *Bitoperationen* hingegen betrachten die *einzelnen Bits* der Binärdarstellung einer Ganzzahl *unabhängig voneinander* und wirken gesondert auf jedem einzelnen Bit.
  Beispiel Bitkonjunktion:

```
     142   10001110
      27   00011011
-------------------
142 & 27   00001010
```

  $\longrightarrow$ Dezimalzahl 10 als Ergebnis
- Bitoperationen werden sehr schnell ausgeführt
- Durch Bitoperationen lassen sich bestimmte arithmetische Berechnungen deutlich beschleunigen
- Bitoperationen häufig zur Auswertung digitaler Signale in Schaltungen genutzt (hardwarenahe Programmierung)

Bitverschiebung ≪ und ≫

- `x >> a` verschiebt die Binärdarstellung der Ganzzahl `x` um `a` Stellen nach *rechts*

  Beispiel `142 >> 3`:

  ```
        142   10001110
   142 >> 3   00010001
  ```
  $\longrightarrow$ Dezimalzahl 17 als Ergebnis

  Ganzzahldivision `x / 2` entspricht `x >> 1`
  Ganzzahldivision `x / 4` entspricht `x >> 2` usw.

Bitverschiebung << und >>

- `x >> a` verschiebt die Binärdarstellung der Ganzzahl `x` um `a` Stellen nach *rechts*

  Beispiel `142 >> 3`:

  ```
        142   10001110
   142 >> 3   00010001
  ```
  $\longrightarrow$ Dezimalzahl 17 als Ergebnis

  Ganzzahldivision `x / 2` entspricht `x >> 1`
  Ganzzahldivision `x / 4` entspricht `x >> 2` usw.

- `x << a` verschiebt die Binärdarstellung der Ganzzahl `x` um `a` Stellen nach *links*

  Beispiel `27 << 3`:

  ```
        27   00011011
   27 << 3   11011000
  ```
  $\longrightarrow$ Dezimalzahl 216 als Ergebnis

  Ganzzahlmultiplikation `2 * x` entspricht `x << 1`
  Ganzzahlmultiplikation `4 * x` entspricht `x << 2` usw.

Operationen zur Bitmaskierung

- `&` realisiert bitweises UND (*Bitkonjunktion*)

`142`	`10001110`	
`27`	`00011011`	
`142 & 27`	`00001010`	$\longrightarrow$ Dezimalzahl 10 als Ergebnis

Operationen zur Bitmaskierung

- `&` realisiert bitweises UND (*Bitkonjunktion*)

```
     142   10001110
      27   00011011
---------------------
142 & 27   00001010
```
$\longrightarrow$ Dezimalzahl 10 als Ergebnis

- `|` realisiert bitweises ODER (*Bitdisjunktion*)

```
     142   10001110
      27   00011011
---------------------
142 | 27   10011111
```
$\longrightarrow$ Dezimalzahl 159 als Ergebnis

Operationen zur Bitmaskierung

- `&` realisiert bitweises UND (*Bitkonjunktion*)

```
     142   10001110
      27   00011011
142 & 27   00001010
```
$\longrightarrow$ Dezimalzahl 10 als Ergebnis

- `|` realisiert bitweises ODER (*Bitdisjunktion*)

```
     142   10001110
      27   00011011
142 | 27   10011111
```
$\longrightarrow$ Dezimalzahl 159 als Ergebnis

- `^` realisiert bitweises XOR (*Bitaddition ohne Übertrag*)

```
     142   10001110
      27   00011011
142 ^ 27   10010101
```
$\longrightarrow$ Dezimalzahl 149 als Ergebnis

Operationen zur Bitmaskierung

- `&` realisiert bitweises UND (*Bitkonjunktion*)

`142`	`10001110`
`27`	`00011011`
`142 & 27`	`00001010` ⟶ Dezimalzahl 10 als Ergebnis

- `|` realisiert bitweises ODER (*Bitdisjunktion*)

`142`	`10001110`
`27`	`00011011`
`142 | 27`	`10011111` ⟶ Dezimalzahl 159 als Ergebnis

- `^` realisiert bitweises XOR (*Bitaddition ohne Übertrag*)

`142`	`10001110`
`27`	`00011011`
`142 ^ 27`	`10010101` ⟶ Dezimalzahl 149 als Ergebnis

- `~` realisiert bitweises NICHT (*Bitinvertierung*)

`142`	`10001110`
`~142`	`01110001` ⟶ Dezimalzahl 113 als Ergebnis

Umrechnung dezimal → binär mittels Bitoperationen

(dezbin.c)

```
#include <stdio.h>

int main(void)
{
  unsigned long x;

  printf("Umrechnung dezimal in binaer\n\n Dezimalzahl: ");
  scanf("%lu", &x);

  while (x != 0)
  {
    printf("%lu\n", x & 1); // x & 1 entspricht x % 2
    x = x >> 1;             // x >> 1 entspricht x / 2
  }
  return 0;
}
```

Prioritäten von Operatoren in C (Auswahl)

hoch	`()`	Funktionsaufruf und Klammerung

	`+ - ! ~`
	`++ -- &`
	`(Typ)`
		Vorzeichen, logisches NICHT, Bitinvertierung
		Inkrement, Dekrement, Adresse
		Typecast

	`* / %`	Multiplikation, Division, Modulo
	`+ -`	Addition, Subtraktion
	`<< >>`	Bitverschiebung
	`< <= > >=`	Vergleichsoperatoren
	`== !=`	Vergleichsoperatoren
	`&`	Bitkonjunktion
	`^`	bitweises XOR
	`|`	Bitdisjunktion
	`&&`	logisches UND
	`||`	logisches ODER
niedrig	`=`	Zuweisung