Schaltnetze

Hans Jürgen Ohlbach

Keywords: Gatter, Schaltnetze, Schaltungsentwurf

Empfohlene Vorkenntnisse: Boolesche Algebra

Inhaltsverzeichnis

1 Einführung

Um die Arbeitsweise von Computern von Grund auf zu verstehen, ist es hilfreich, zu wissen, wie die elementaren Bausteine der Prozessoren aufgebaut sind, und wie sie zu komplexeren Einheiten mit komplexeren Funktionen zusammengeschaltet werden. Dabei hilft auch, den Entwicklungsprozess von komplexen Schaltungen zu betrachten, um sich zu überzeugen, dass das kein Hexenwerk ist, sondern relativ einfachen Vorgehensweisen folgt.

In diesem Miniskript werden daher zunächst die einfachsten Schaltelemente, die Gatter eingeführt, und dann gezeigt, wie man die Gatter zu komplexeren Schaltungen, sog. Schaltnetzen, zusammensetzt. Die Methodik wird am Beispiel eines Addierwerks eingeführt. Hat man das verstanden, kann

man leicht andere Schaltnetze selbst entwickeln. Schaltnetze können Berechnungen durchführen, aber keine Informationen speichern. Dazu braucht man andere Techniken, die in anderen Miniskripten behandelt werden.

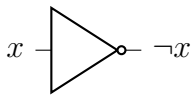
2 Gatter

Auf der untersten Ebene arbeiten Computer mit einzelnen Bits. Bits stellen die Booleschen Werte 0 und 1 dar. Die Operationen auf diesen Bits werden daher durch *Boolesche Funktionen* beschrieben. Die Theorie dahinter ist die Theorie der *Booleschen Algebra*. In dieser Theorie betrachtet man ein- und mehrstellige Boolesche Funktionen. Unter den einstelligen Booleschen Funktionen ist die *Negationsfunktion* $\neg x$ die einzig interessante. Sie kann durch folgende *Wertetabelle* beschrieben werden:

x	$\neg x$
0	1
1	0

Offensichtlich dreht sie die Bits gerade um.

In technischen Schaltungen wird hierzu das folgende Symbol verwendet.



Es stellt ein sog. **Negationsgatter** dar. Das ist ein Schaltelement mit einem Eingang und einem Ausgang. Das Gatter wandelt eine 1 am Eingang in eine 0 am Ausgang um, und umgekehrt. Auf Hardwareebene wird die 0 als 0 Volt Spannung realisiert, und die 1 als eine höhere Spannung, z.B. als 5 Volt (meist aber weniger). Für das Verständnis der Schaltungen ist diese Hardwareebene aber zunächst nicht wichtig.

Aus der Theorie der Booleschen Algebra folgt, dass es insgesamt 16 2-stellige Boolesche Funktionen gibt, wovon aber nur wenige, nämlich die *funktional vollständigen* gebraucht werden. Dabei heißt eine Menge von Booleschen Funktionen funktional vollständig, wenn man alle andere Booleschen Funktionen mit Hilfe dieser Funktionen definieren kann. Ein funktional vollständige Menge ist die Menge bestehend aus der Negationsfunktion, sowie *und* und *oder*. Sie werden durch folgende Wertetabellen spezifiziert:

x	y	x und y	x oder y
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

In der Literatur findet man sehr unterschiedliche Symbole für *und* und *oder*

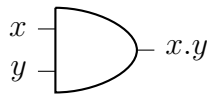
Funktion	technisch	logisch	Programme	Bedeutung
und	.	$\wedge$	&	logisches und
oder	+	$\vee$	|	logisches oder

Da es in diesem Miniskript um technische Themen geht, benutzen wir im folgenden auch die technische Notation, d.h. . steht für und und + steht für oder.

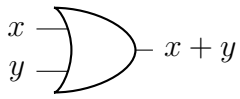
Technisch realisiert werden diese Funktionen durch *und-Gatter* und *oder-Gatter*. Beide sind Schaltkreise mit zwei Eingängen und einem Ausgang. Die Gatter verarbeiten die Bits an den Eingängen entsprechend der obigen Wertetabelle.

Folgende Symbole werden dafür benutzt:

und-Gatter



oder-Gatter

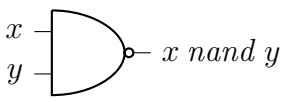


Aus der Theorie der Booleschen Algebra folgt, dass man sogar mit noch weniger Booleschen Funktionen auskommen kann: entweder mit dem negierten und (NAND), oder dem negierten oder (NOR):

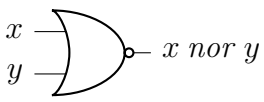
x	y	$x\ nand\ y$	$x\ nor\ y$
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

Jeweils eine von den beiden Funktionen genügt, um alle anderen Booleschen Funktionen damit zu definieren. Sie sind *minimal funktional vollständig*. Die technischen Symbole dazu sehen folgendermaßen aus:

nand-Gatter



nor-Gatter



Der kleine Kreis am rechten Ende des Symbols deutet die Negation an.

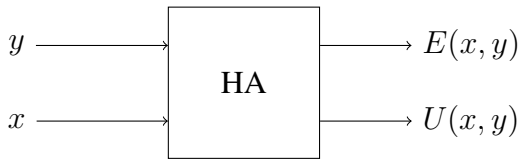
3 Schaltnetze

Einzelne Gatter kann man nun zu *Schaltnetzen* zusammenbauen, indem man die Ausgangsleitungen von Gattern mit den Eingangsleitungen von weiteren Gattern verbindet. Damit kann man komplexere Booleschen Funktionen realisieren.

3.1 Halbaddierer

Wir illustrieren die Vorgehensweise zunächst am Beispiel des *Halbaddierers*. Dieser addiert zwei Bits und erzeugt ein Ergebnisbit E und einem Übertragsbit U. Z.B. $1 + 1 = 2$ (+ ist jetzt die Addition) ist in Binärdarstellung: $1 + 1 = 10$. Man braucht hierfür also eine 1 als Übertrag.

Wir möchten also eine „Kiste“ bauen, mit zwei Eingängen, x und y, sowie zwei Ausgängen $E(x, y)$ und $U(x, y)$:



Die Wertetabelle dazu ist:

x	y	E	U
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Diese Wertetabelle spezifiziert die zwei Booleschen Funktionen $E(x, y)$ und $U(x, y)$. Um die Booleschen Terme dafür zu finden, überprüft man in den entsprechenden Spalten, wann die Funktion 1 wird.

E wird genau dann 1 wenn $x = 0$ und $y = 1$ oder wenn $x = 1$ und $y = 0$ ist. Daraus lässt sich der Boolesche Term ableiten:

$$E(x, y) = (\neg x.y) + (x.\neg y)$$

U wird genau dann 1 wenn $x = 1$ und $y = 1$ ist. Daraus lässt sich der Boolesche Term ableiten:

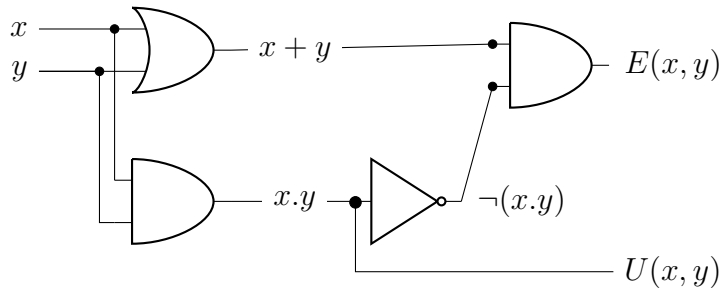
$$U(x, y) = (x.y).$$

Der Boolesche Term besteht also aus und-Termen, oder-Termen und Negationstermen. Diese lassen sich als und-, oder- und Negationsgatter realisieren. Bevor man die Umsetzung in ein Schaltnetz angeht, ist es hilfreich, die Terme so umzubauen, dass möglichst wenig verschiedene Terme auftauchen. Dann braucht man später auch weniger Gatter. Der Boolesche Term für $E(x, y)$, so wie der dasteht, braucht zwei Negationen, zwei und-Verknüpfungen und eine oder-Verknüpfung. Dazu kommt noch eine weiterer und-Verknüpfung für den Übertrag $U(x, y)$.

Folgende Umformungen ergeben eine Booleschen Term mit weniger Verknüpfungen:

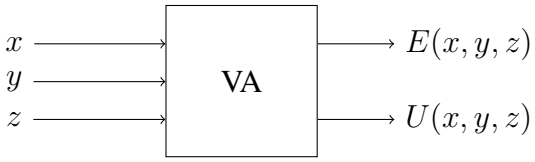
$$
\begin{aligned}
E(x,y) &= (\neg x.y) + (x.\neg y) \\
&= (\neg x + x).(\neg x + \neg y).(y + x).(y + \neg y) && \text{(Ausmulitiplizieren)} \\
&= (\neg x + \neg y).(y + x) && ((\neg X + X) = 1, 1.X = X) \\
&= \neg(x.y).(x + y) && \text{(Negation ausklammern)}
\end{aligned}
$$

Jetzt enthalten sowohl $U(x,y)$ als auch $E(x,y)$ den Term $(x.y)$, der, obwohl er zweimal vorkommt, durch ein einziges Gatter realisiert werden kann. Das Schaltnetz, welches sich daraus ergibt, sieht dann folgendermaßen aus:



3.2 Volladdierer

Ein Halbaddierer addiert zwei Bits und erzeugt ein Ergebnisbit E und ein Übertragsbit U. Um diesen Übertrag weiter zu verarbeiten, braucht man einen Volladdierer, der zwei Bits x und y und ein Übertragsbit z addiert. Die Ausgabe ist wiederum ein Ergebnisbit $E(x,y,z)$ und ein Übertragsbit $U(x,y,z)$.



Die Wertetabelle dafür ist

x	y	z	E	U
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Um den Booleschen Term für E zu finden, überprüfen wir alle Einträge in der E-Spalte, wo 1 steht, und erzeugen aus jedem zugehörigen x, y, z-Tripel einen Booleschen Term. Wenn an der x-Position eine 1 steht wird daraus x, wenn dort 0 steht, wird daraus $\neg x$. E wird in insgesamt 4 Fällen zu 1. Daraus ergibt sich:

$$E(x, y, z) = (\neg x.\neg y.z) + (\neg x.y.\neg z) + (x.\neg y.\neg z) + (x.y.z)$$

U wird auch in insgesamt 4 Fällen zu 1. Daraus ergibt sich:

$$U(x, y, z) = (\neg x.y.z) + (x.\neg y.z) + (x.y.\neg z) + (x.y.z)$$

Diese Terme muss man noch so umformen, dass möglichst wenige Boolesche Verknüpfungen, und damit auch möglichst wenig Gatter gebraucht werden.

$$\begin{aligned} U(x, y, z) &= (\neg x.y.z) + (x.\neg y.z) + (x.y.\neg z) + (x.y.z) & \\ &= (\neg x.y.z) + (x.\neg y.z) + (x.y).(\neg z + z) & \text{Ausklammern} \\ &= (\neg x.y.z) + (x.\neg y.z) + (x.y) & (\neg z + z) = 1 \\ &= z.((\neg x.y) + (x.\neg y)) + (x.y) & \text{Ausklammern} \end{aligned}$$

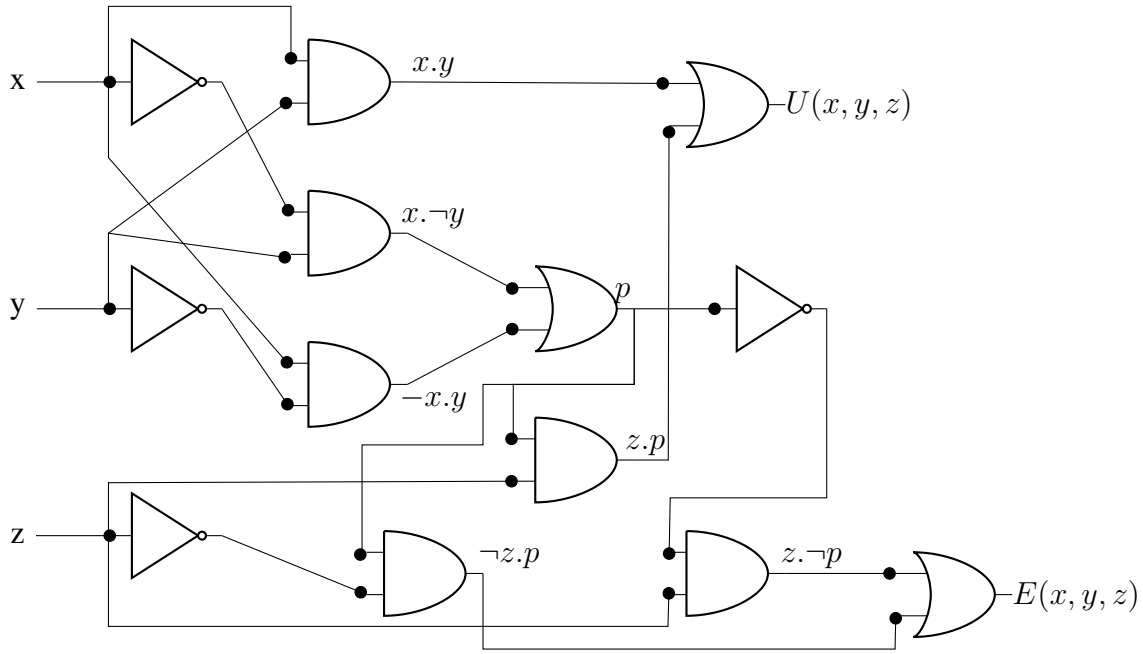
Für den E-Term bietet es sich jetzt auch an, so auszuklammern, dass Terme mit x und y übrig bleiben.

$$\begin{aligned} E(x, y, z) &= (\neg x.\neg y.z) + (\neg x.y.\neg z) + (x.\neg y.\neg z) + (x.y.z) & \\ &= z.((\neg x.\neg y) + (x.y)) + \neg z.((\neg x.y) + (x.\neg y) & \text{Ausklammern} \\ &= z.\neg((\neg x.y) + (x.\neg y)) + \neg z.((\neg x.y) + (x.\neg y)) & \text{Ausmultiplizieren} \end{aligned}$$

Für den Schaltungsentwurf müssen wir also x,y und z negieren, dann und-Gatter für $(x.y)$, $(\neg x.y)$ und $(x.\neg y)$ einführen, und dann deren Ausgaben mit z „verunden“, und das dann noch „verodern“. Mit der Abkürzung $p = (\neg x.y) + (x.\neg y))$ haben wir:

$$\begin{aligned} E(x, y, z) &= z.\neg p + \neg z.p \\ U(x, y, z) &= z.p + x.y \end{aligned}$$

Die Schaltung sieht dann so aus:



Allgemeine Vorgehensweise :
Die allgemeine Vorgehensweise sollte jetzt klar sein.

- Das was der Schaltkreis / die Boolesche Funktion leisten soll, spezifiziert man mit einer Wertetabelle. Man bekommt dabei eine bestimmte Anzahl von Eingabewerten ($x, y, z, \ldots$) und eine bestimmte Menge von gewünschten Ausgabewerten.
- Den Booleschen Term für jeden der Ausgabewerte erhält man, indem man in der entsprechenden Spalte die Einträge mit 1 betrachtet, und für jeden solchen 1er Eintrag aus den Bedingungen, wie die 1 entsteht den Booleschen Term generiert. Eine 1 erzeugt den zugehörigen Variablennamen direkt, eine 0 erzeugt die Negation davon. Diese Literale werden mit *und* (.) verbunden. Die aus den 1er-Zeilen erzeugten Terme werden mit *oder* (+) verbunden. Das ergibt übrigens sofort eine Disjunktive Normalform.
- Die so erzeugten Terme werden oft sehr redundant, und müssen daher mit den Methoden der Booleschen Algebra vereinfacht werden.

Falls die Spalte für den Ausgabewert mehr 0en als 1en enthält, kann man statt den Zeilen mit 1 auch die Zeilen mit 0 zu einem Booleschen Term machen. Dieser beschreibt dann die negierte Boolesche Funktion. Negiert man sie nochmal, erhält man die richtige Boolesche Funktion.

3.3 Ein Ripple-Carry Addierwerk

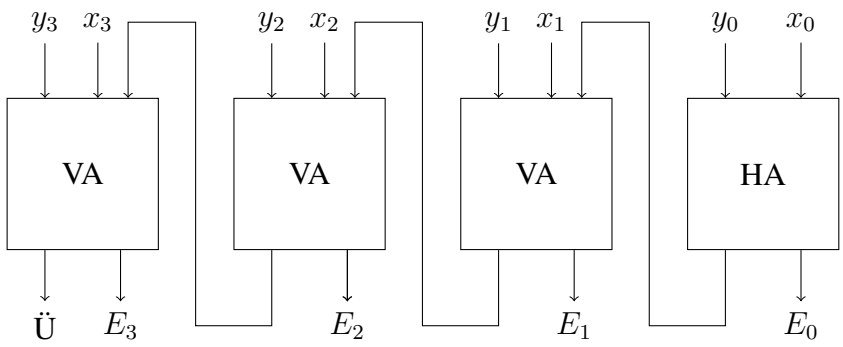
Um das Beispiel des bitweisen Addierens zu vervollständigen, skizzieren wir jetzt ein vollständiges Addierwerk, das man für beliebig lange Bitsequenzen realisieren kann. Es orientiert sich am üblichen

schriftlichen addieren, bitweise von rechts nach links, mit Übertrag, wie z.B.

$$\begin{array}{rr} & 0111101 \\ + & 0011011 \\ \text{Ü} & 1111110 \\ \hline & 1011000 \end{array}$$

Für die beiden rechtesten Bits braucht man also einen Halbaddierer, und für alle anderen einen Volladdierer, der jeweils zwei Bits plus Übertrag addiert.

Ein Addierwerk zur Addition von zweimal 4 Bits $(x_3x_2x_1x_0) + (y_3y_2y_1y_0)$ sieht dann so aus:



Das Ergebnis ist dann der Bitvektor $(E_3E_2E_2E_0)$ plus ein Überlaufbit Ü. Das Überlaufbit wird immer dann 1 wenn die Addition der zwei 4-Bit Zahlen eine Zahl mit 5 Bits ergibt.

In konkreten Prozessoren hat man nicht Wortlängen von 4 Bits, sondern 32 oder 64 Bits. Aber auch dann kann ein Überlauf passieren, wenn die Zahlen zu groß werden. In den meisten Systemen wird dann eine Fehlermeldung erzeugt.

3.4 Ein Carry Select Addierwerk

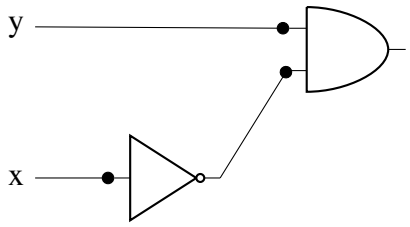
Das Ripple Carry Addierwerk braucht so viele Schritte, wie die Bitfolge lang ist. Das liegt daran, dass die Überläufe nacheinander entstehen und verarbeitet werden müssen. Das kann man mit folgender Idee beschleunigen. Betrachten wir ein Addierwerk für zweimal 32 Bit Worte. Die Idee ist, diese Worte zunächst in zweimal 16 Bits zu zerlegen. Die hinteren 16 Bits können addiert werden, und erzeugen eventuell einen Überlauf. Um die vorderen 16 Bits zu addieren, verdoppelt man das Addierwerk, eines für Überlauf = 0 und eines für Überlauf = 1. Jetzt können alle drei 16-Bit Addierwerke parallel rechnen. Sobald der Überlauf für die hinteren 16 Bit feststeht, nimmt man das Ergebnis des passenden Addierwerks für die vorderen 16 Bits.

Damit hat man die Rechenzeit verdoppelt. Man kann natürlich die drei Addierwerke für die 16 Bits wiederum mit dem gleichen Trick aufteilen, und verdoppelt die Rechenzeit nochmal. Die entstehenden 8-Bit Addierwerke lassen sich auch nochmal aufteilenen usw. Insgesamt braucht man dafür sehr viel Hardware, aber heutzutage ist Hardware billig.

3.5 Verzögerungsglieder

Die Schaltungen sind natürlich nicht für eine einzige Berechnung gedacht, wo man die Bits an die Eingänge anlegt, und dann die Ergebnisbits ausliest. Meist sind es Abermillionen Berechnungen, die nacheinander ausgeführt werden sollen. Für jede Berechnung liegen die Bits dann für eine winzige Zeitspanne an den Eingängen der Schaltungen, und danach kommen schon die Bits für die nächste Berechnung.

Leider ist es nicht so, dass im gleichen Moment, wo die Bits an die Eingänge gelegt werden, auch schon die Ergebnisbits an den Ausgängen anliegen. Jedes Gatter braucht eine bestimmte Zeit t , um das Ergebnis zu liefern. Läuft eine Berechnung durch n Gatter, dann ergibt sich eine Zeitverzögerung von $n \cdot t$ bis die Berechnung fertig ist. Hat jetzt ein Gatter 2 Eingänge (wie die und- und oder-Gatter), und der Weg von den Eingängen zu dem 1. Eingang des Gatters geht durch weniger Gatter als der Weg von den Eingängen zum 2. Eingang des Gatters, dann kommt das Bit am 2. Eingang später an als das Bit am 1. Eingang, so wie in der Schaltung:



In diesem Moment könnte aber am ersten Gatter schon die nächste Berechnung anstehen, so dass die Bits bei aufeinander folgenden Berechnungen vermischt werden. Das darf natürlich nicht geschehen! Daher muss beim Schaltungsentwurf für jedes Gatter analysiert werden, wie die Zeitverzögerung an den einzelnen Gattern genau ist. Um sicher zu gehen, dass die Bits exakt zur gleichen Zeit bei den Eingängen ankommen, gibt es spezielle Verzögerungsgatter, die die Signale auf den Wegen zu den Eingängen so verlangsamen, dass alle Bits exakt zur gleichen Zeit ankommen.

Stichwortverzeichnis