Schaltnetze

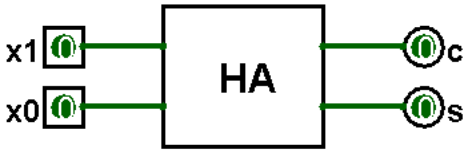
Inhalt

Mit den im Teil *Schaltalgebra* eingeführten booleschen Gesetzen und logischen Toren lassen sich kombinatorische Schaltungen realisieren. Im Folgenden werden einige ausgewählte Schaltungen vorgestellt.

1. Addierer

Halbaddierer

Der Halbaddierer (HA) weist zwei 1-Bit-Summanden x_1 und x_0 als Eingänge und ein 1-Bit-Resultat s (sum) sowie einen Übertrag c (carry) als Ausgänge auf.

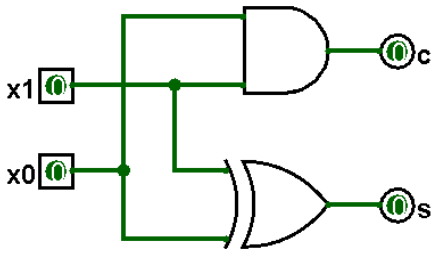


Die Wahrheitstabelle für den HA lautet:

x_1	x_0	c (carrying, übertrag)	s (summe)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

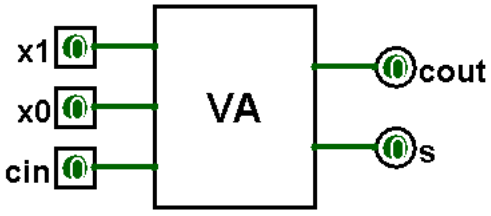
$$c = x_1 x_0$$
$$s = x_1 \oplus x_0$$

Der s-Ausgang entspricht einer Antivalenz, der c-Ausgang einer AND-Verknüpfung. Der HA – oben als Black Box (BB) gezeichnet (es sind die Ein- und Ausgänge sowie deren Verhalten bekannt, nicht aber die Schaltung, die in der BB steckt) – sieht als Schaltung realisiert wie folgt aus:



Volladdierer

Der Volladdierer (VA) weist zwei 1-Bit-Summanden x_1, x_0 und ein Carry-in (c_{in}) als Eingänge sowie ein 1-Bit-Resultat Summe (s) und einen Übertrag carry-out (c_{out}) als Ausgänge auf.



Die Wahrheitstabelle des VA lautet:

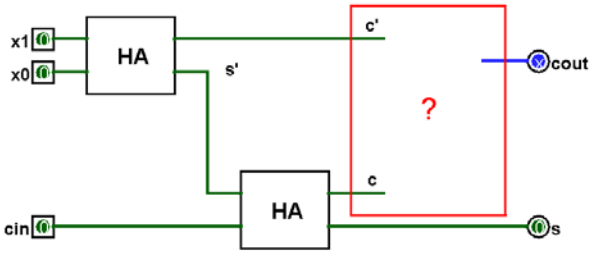
c_{in}	x_1	x_0	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$c_{out} = x_1x_0 + c_{in}(x_1 + x_0)$$

$$s = \overline{c_{in}}\,\overline{x_1}\,x_0 + \overline{c_{in}}\,x_1\,\overline{x_0} + c_{in}\,\overline{x_1}\,\overline{x_0} + c_{in}\,x_1\,x_0$$
$$s = \overline{c_{in}}\,(\overline{x_1}\,x_0 + x_1\,\overline{x_0}) + c_{in}\,(\overline{x_1}\,\overline{x_0} + x_1\,x_0)$$
$$s = \overline{c_{in}}(x_1 \oplus x_0) + c_{in}(\overline{x_1 \oplus x_0})$$
$$s = c_{in} \oplus (x_1 \oplus x_0)$$

Ein VA kann mit zwei HA realisiert werden:

Für die Summenbildung wird angesetzt:



Wie ist c_{out} zu realisieren?

Es gilt:
$c' = x_1x_0$
$c = c_{in}s' = c_{in}\,(x_1 \oplus x_0) = c_{in}\,(\overline{x_1}\,x_0 + x_1\,\overline{x_0})$
$c_{out} = x_1x_0 + c_{in}(x_1 + x_0)$

Idee: c' und c verwenden, um c_{out} zu bilden.
Könnte $c_{out} = c' + c$ funktionieren?
$c_{out} = x_1x_0 + c_{in}(\overline{x_1}\,x_0 + x_1\,\overline{x_0})$

2x verdoppeln

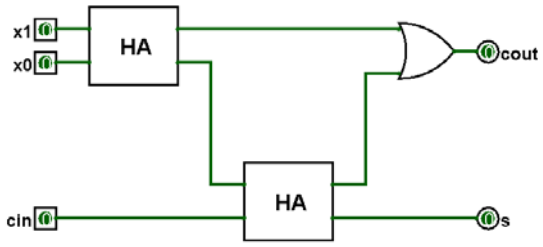
$$c_{out} = \underbrace{x_1x_0} + c_{in}(\overline{x_1}\,x_0 + x_1\,\overline{x_0}) + \underbrace{(\overline{c_{in}}\,x_1\,x_0 + c_{in}\,x_1\,x_0)} + \overbrace{c_{in}\,x_1\,x_0} + \overbrace{c_{in}\,x_1\,x_0}$$

verdoppeln, erweitern

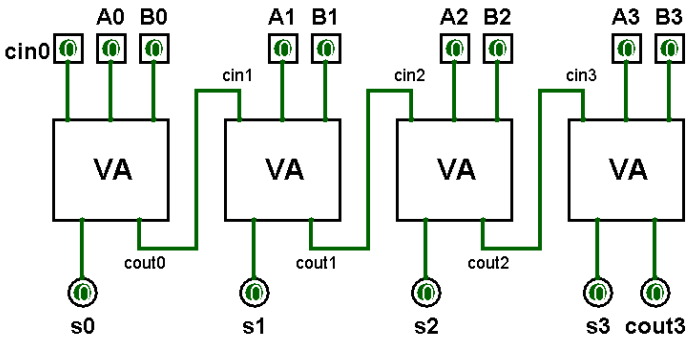
$$c_{out} = x_1x_0 + c_{in}\overline{x_1}\,x_0 + c_{in}x_1\,\overline{x_0} + \overline{c_{in}}\,x_1\,x_0 + c_{in}\,x_1\,x_0 + c_{in}\,\overline{x_1}\,x_0 + c_{in}\,x_1\,x_0$$

Obiges Resultat zusammengefasst:
$c_{out} = x_1x_0 + c_{in}(x_1 + x_0)$ → ok!

Als Schaltung:



Ripple-Carry-Addierer (RCA)



Das Resultat des MSB (s wie c) muss sich durch alle Stufen durcharbeiten (daher *ripple*). Dies bedeutet bei grossen Wortbreiten einen entsprechenden Zeitbedarf bis zum Vorliegen des Resultats am höchstwertigen VA.

Schnellste Addierer

Alle Schaltnetze lassen sich in die eine oder andere Normalform bringen. Der Aufwand zur Realisierung wird bereits ab kleinen Wortbreiten sehr gross.

Beispiel:
Ein 8-Bit-Addierer mit Übertrag soll in der Normalform realisiert werden. Die Wahrheitstabelle weist dann 8 Bit für den Summanden A, 8 Bit für den Summanden B und 1 Carry Bit auf, d.h. insgesamt 17 Bit. Dies ergibt eine Wahrheitstabelle mit 2^{17} Zeilen!
Ein Kompromiss zwischen Hardwareaufwand und Verzögerungszeit kann gefunden werden mit dem

Carry-Look-Ahead-Addierer (CLAA)

Idee: Ein Schaltnetz berechnet mit VA die Summen wie beim RCA. Die Überträge sollen jedoch in einem separaten Schaltnetz (schneller als beim RCA) berechnet werden.

Oben wurde für den Übertrag des VA gefunden:
$c_{out0} = A_0 B_0 + c_{in0}(A_0 + B_0)$

Folgende Substitutionen werden eingeführt:
$g_0 = A_0 B_0$ g: carry generate
$p_0 = A_0 + B_0$ p: carry propagate (allfälliges Weiterreichen eines Carry)

und allgemein für die Stufe i:
$g_i = A_i B_i$ und $p_i = A_i + B_i$

Über verschiedene Stufen (mit Indizes) gekennzeichnet:
$c_{out0} = g_0 + c_{in0} p_0$

$c_{out1} = g_1 + c_{in1} p_1 = g_1 + c_{out0} p_1$ $c_{in1} = c_{out0}$: verbindende Leitung

$c_{out1} = g_1 + g_0 p_1 + c_{in0} p_0 p_1$

$c_{out2} = g_2 + c_{in2} p_2 = g_2 + c_{out1} p_2$ $c_{in2} = c_{out1}$: verbindende Leitung

$c_{out2} = g_2 + g_1 p_2 + g_0 p_1 p_2 + c_{in0} p_0 p_1 p_2$

$c_{out3} = g_3 + c_{in3}p_3 = g_3 + c_{out2}p_3$ $\qquad\qquad$ $c_{in3} = c_{out2}$

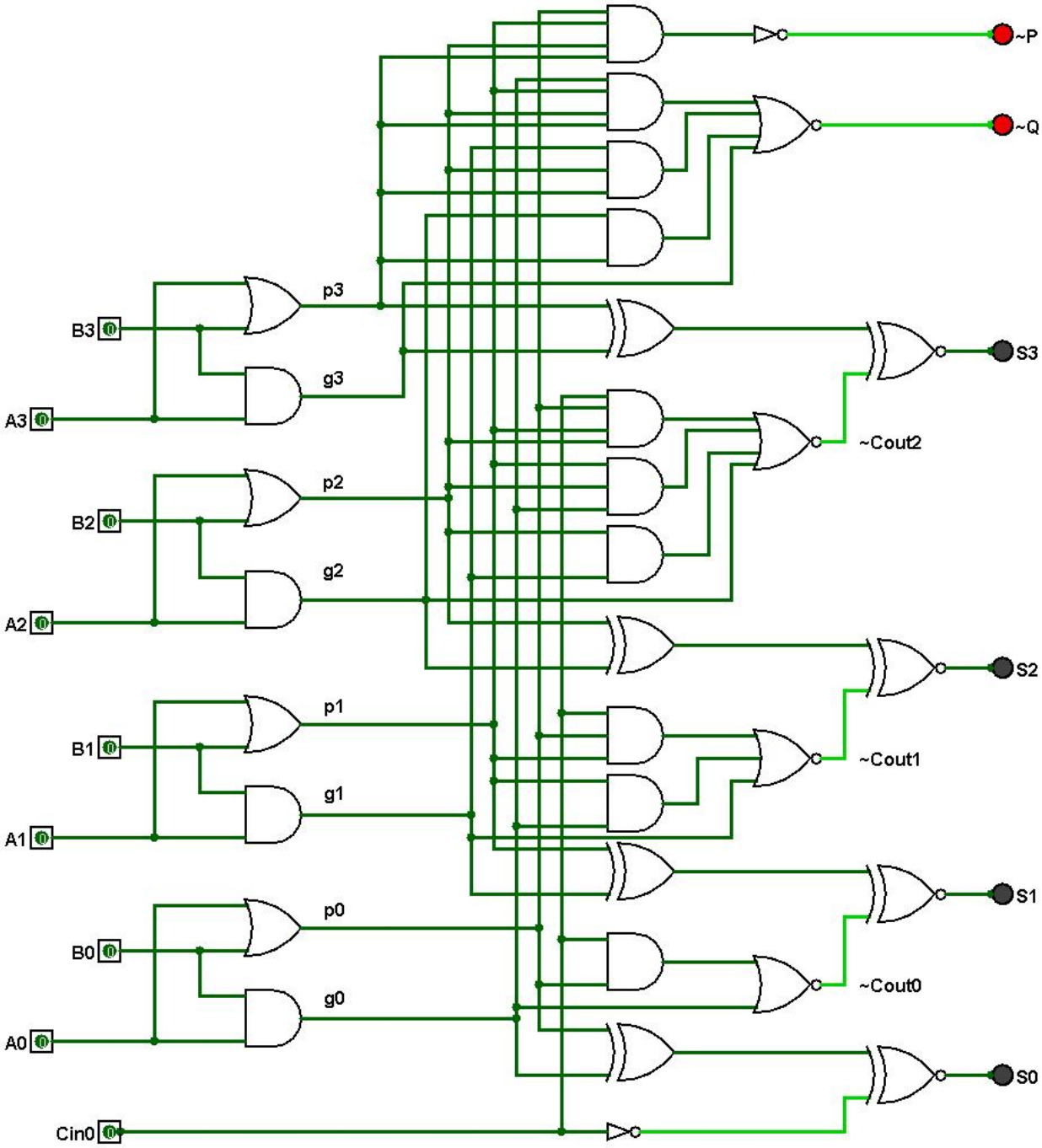
$c_{out3} = g_3 + g_2p_3 + g_1p_2p_3 + g_0p_1p_2p_3 + c_{in0}p_0p_1p_2p_3$

Sind g_i und p_i also bereits gebildet, so können diese zur Erzeugung der Summenbits verwendet werden.
Es gilt nämlich: $g_i \oplus p_i = A_i \oplus B_i$

Aufgabe:
Beweisen Sie, dass die Beziehung $g_i \oplus p_i = A_i \oplus B_i$ korrekt ist.

Es gilt ebenso:
$s_i = c_{in\,i} \oplus A_i \oplus B_i = g_i \oplus p_i \oplus c_{in\,i}$
und:
$\overline{s_i} = g_i \oplus p_i \oplus \overline{c_{in\,i}}$

Diese Beziehungen ausnutzend ergibt sich nun ein Addierschaltnetz, wie auf der nächsten Seite gezeigt:

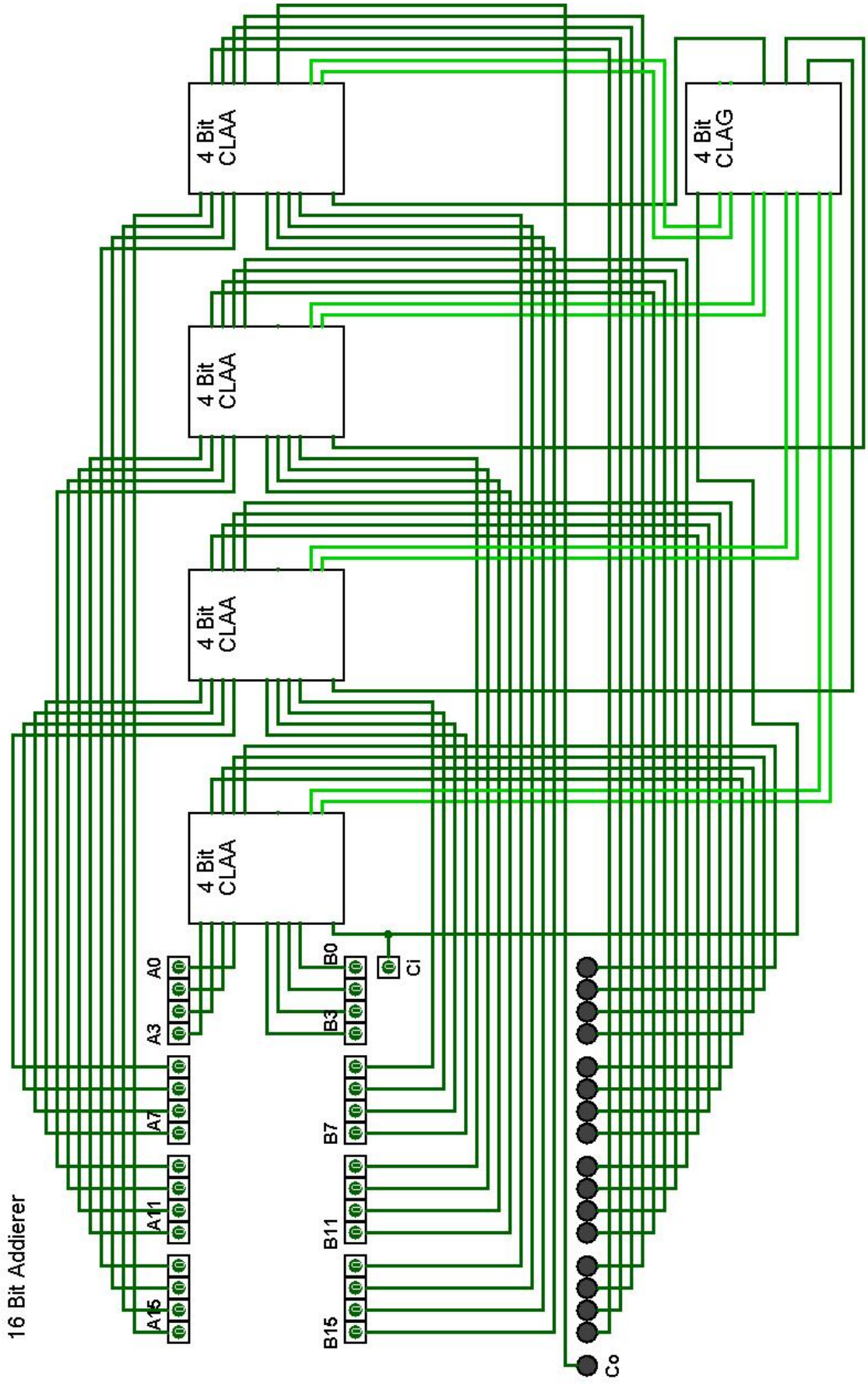


Carry-Look-Ahead-Addierer für zwei 4-Bit-Summanden mit Propagate und Generate

Aufgabe:
Ergänzen Sie obige Schaltung mit dem Ausgang $\mathrm{Cout3} = c_{out3}$, d.h. demjenigen Ausgang, der anzeigt, dass eine Addition ein Resultat > 15 dezimal ergibt.

Hinweis:
Durch geschickte Verwendung der Ausgänge P (propagate) und G (generate) hält sich der Schaltungsaufwand in Grenzen.
Notieren Sie daher vorerst $\mathrm{Cout3}$ in Funktion von P und G und nötigen weiteren Signalen, formal: $\mathrm{Cout3} = \mathrm{f}(\mathrm{P}, \mathrm{G})$.

Kaskadierung von CLA-Addierern mit einem CLA-Generator (CLA-G):



16-Bit-Addierer mit Überträgen: Carry-In und Carry-Out in Testanordnung.

2. Komparatoren

Komparatoren vergleichen zwei gleich lange Wörter. Sie testen auf *gleich*, *grösser* und *kleiner*, allenfalls auch auf Kombinationen davon. In Rechnern werden sie z.B. eingesetzt, um Sprungbedingungen zu prüfen.

Wahrheitstabelle eines 2-Bit-Komparators (Vergleichen zweier 2-Bit-Wörter):

y_1	y_0	x_1	x_0	$z_{x=y}$	$z_{x<y}$	$z_{x>y}$
0	0	0	0	1	0	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	0	1	0
0	1	0	1	1	0	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	0	1	0
1	0	0	1	0	1	0
1	0	1	0	1	0	0
1	0	1	1	0	0	1
1	1	0	0	0	1	0
1	1	0	1	0	1	0
1	1	1	0	0	1	0
1	1	1	1	1	0	0

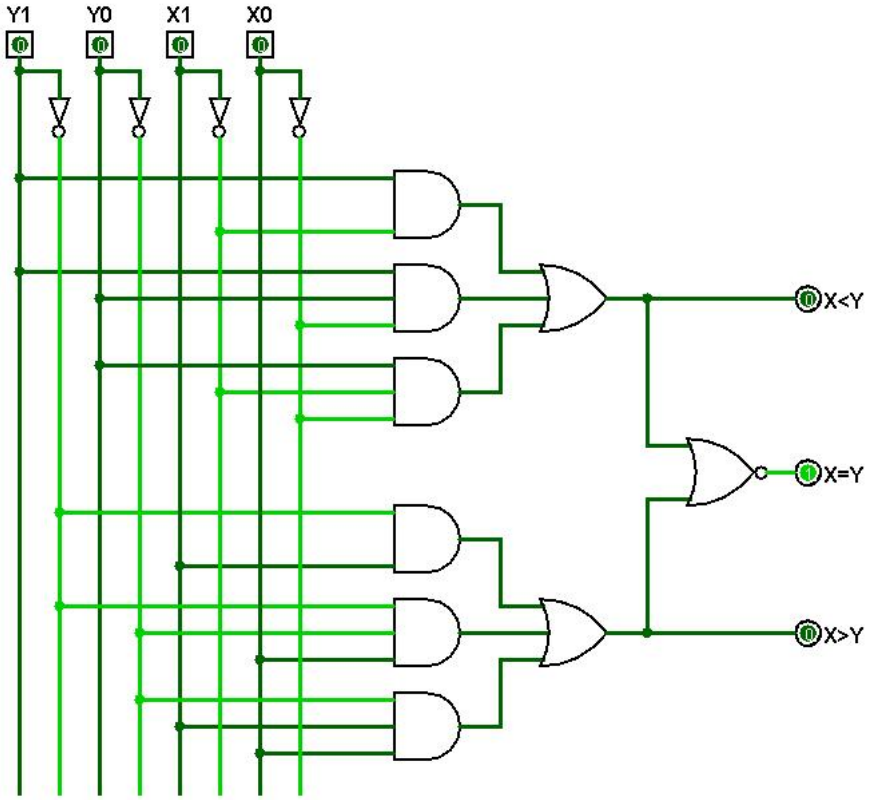
Schaltfunktionen nach dem Minimieren:

$$z_{x<y} = y_1 \overline{x_1} + y_1 y_0 \overline{x_0} + y_0 \overline{x_1} \overline{x_0}$$

$$z_{x>y} = \overline{y_1} x_1 + \overline{y_1} \overline{y_0} x_0 + \overline{y_0} x_1 x_0$$

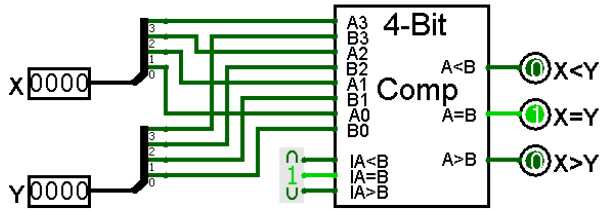
Zudem:

$$z_{x=y} = \overline{z_{x<y}} \cdot \overline{z_{x>y}} = \overline{z_{x<y} + z_{x>y}}$$

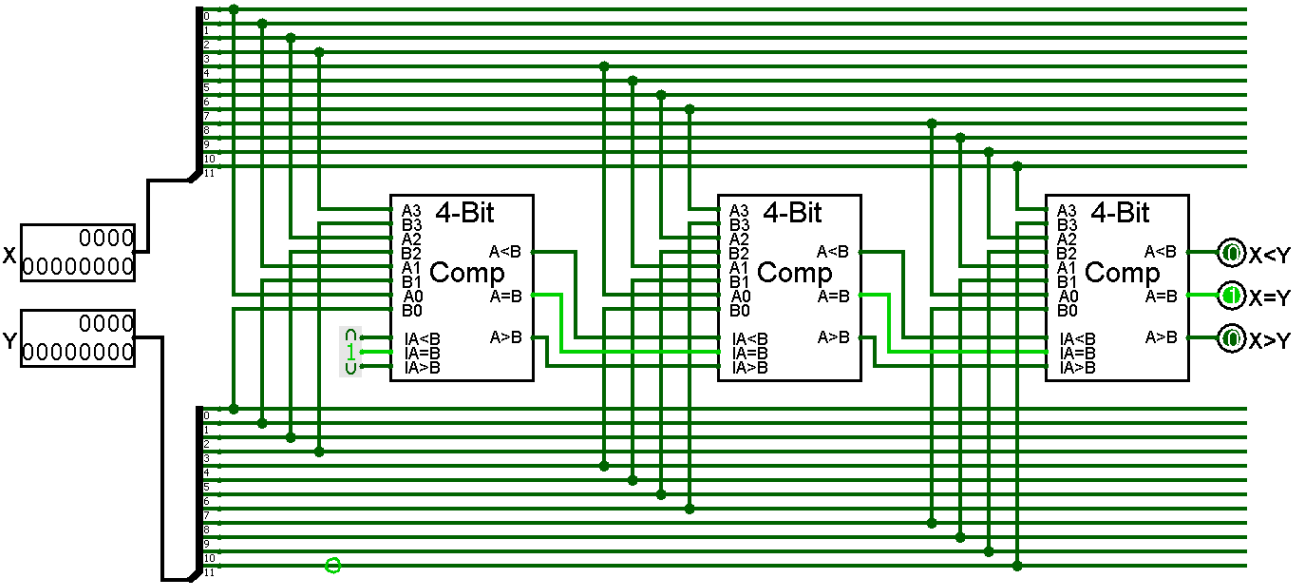


Schaltung eines 2-Bit-Komparators

Kaskadierbare Komparatoren
In der Praxis müssen meist breitere Wörter verglichen werden. Dazu werden kaskadierbare Schaltungen entworfen und in Bausteinen realisiert.



Kaskadierung von mehreren Komparatoren zu einem 12-Bit Magnitude Komparator:



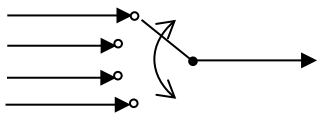
Die Überträge X<Y, X=Y und X>Y brauchen Zeit, da z.B. die niedrigsten Überträge durch alle Torebenen geführt werden müssen.

Vgl. auch 24-Bit Magnitude Comparator in *schaltnetze.circ*.

3. Multiplexer/Demultiplexer

Multiplexer

Ein Multiplexer repräsentiert eine Schaltung, die von n (digitalen) Eingängen einen einzigen Eingang auf den Ausgang durchschaltet.



Beispiel 4:1-Multiplexer

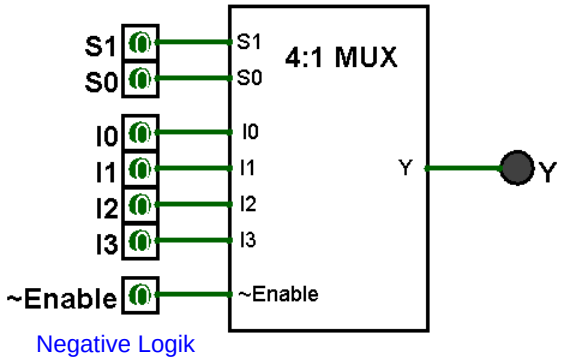
Vgl. dazu Visualisierung in *schaltnetze.circ.*

Funktionstabelle eines Multiplexers mit zusätzlichem Enable-Eingang (tief-aktiv[1]):

$\overline{\text{Enable}}$	S1	S0	I3	I2	I1	I0	Y
1	x	x	x	x	x	x	0
0	0	0	x	x	x	a	a
0	0	1	x	x	b	x	b
0	1	0	x	c	x	x	c
0	1	1	d	x	x	x	d

$\overline{\text{Enable}}$	tief-aktiver[1] Eingang, d.h. die Funktion des Durchschaltens ist für den Zustand 0 aktiv, für 1 nicht-aktiv
S1, S0	Selektiereingänge: Wahl der Kanaldurchschaltung
I0, I1, I2, I3	Dateneingänge der Eingangskanäle
a, b, c, d	digitales Signal auf dem entsprechenden Dateneingang (Ausgangskanal)
x	don't care

Multiplexer-Baustein

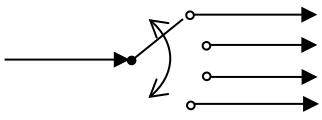


Negative Logik
wenn Wert 0 -> Enabled

Frage:
Wie ist der Baustein oben in seinem Innern (z.B. mit Logisim) zu realisieren?

Demultiplexer

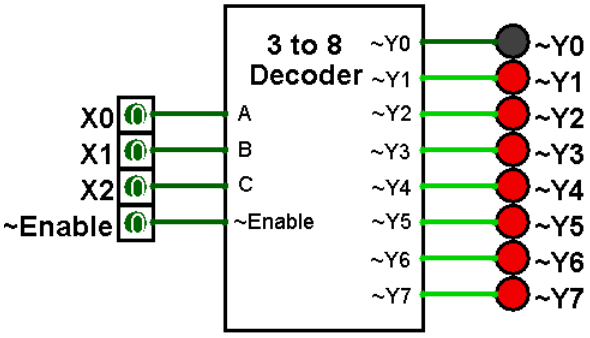
Ein Demultiplexer repräsentiert eine Schaltung/einen Baustein, der 1 (digitalen) Eingang auf 1 von n zur Verfügung stehenden Ausgängen durchschaltet. Er ist demnach das Gegenstück zum Multiplexer.



Als Demultiplexer können Code-Umsetzer wie z.B. ein Binär-nach-Oktal-Codewandler mit 8 Ausgängen (auch als 3-to-8 Decoder bezeichnet) verwendet werden (vgl. nächste Seite).

[1] Vgl. auch Anhang.

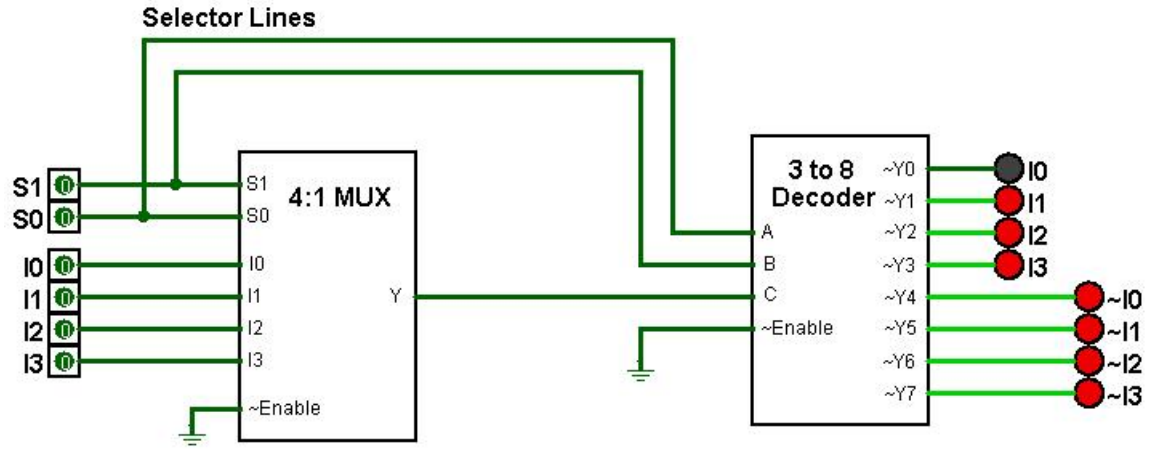
Dekodierbaustein (A: LSB; C: MSB)



$\overline{\text{Enable}}$	C	B	A	$\overline{Y7}$	$\overline{Y6}$	$\overline{Y5}$	$\overline{Y4}$	$\overline{Y3}$	$\overline{Y2}$	$\overline{Y1}$	$\overline{Y0}$
1	x	x	x	1	1	1	1	1	1	1	1
0	0	0	0	1	1	1	1	1	1	1	**0**
0	0	0	1	1	1	1	1	1	1	**0**	1
0	0	1	0	1	1	1	1	1	**0**	1	1
0	0	1	1	1	1	1	1	**0**	1	1	1
0	1	0	0	1	1	1	**0**	1	1	1	1
0	1	0	1	1	1	**0**	1	1	1	1	1
0	1	1	0	1	**0**	1	1	1	1	1	1
0	1	1	1	**0**	1	1	1	1	1	1	1

~Enable und Ausgänge **tief-aktiv**.
Vgl. S. 14

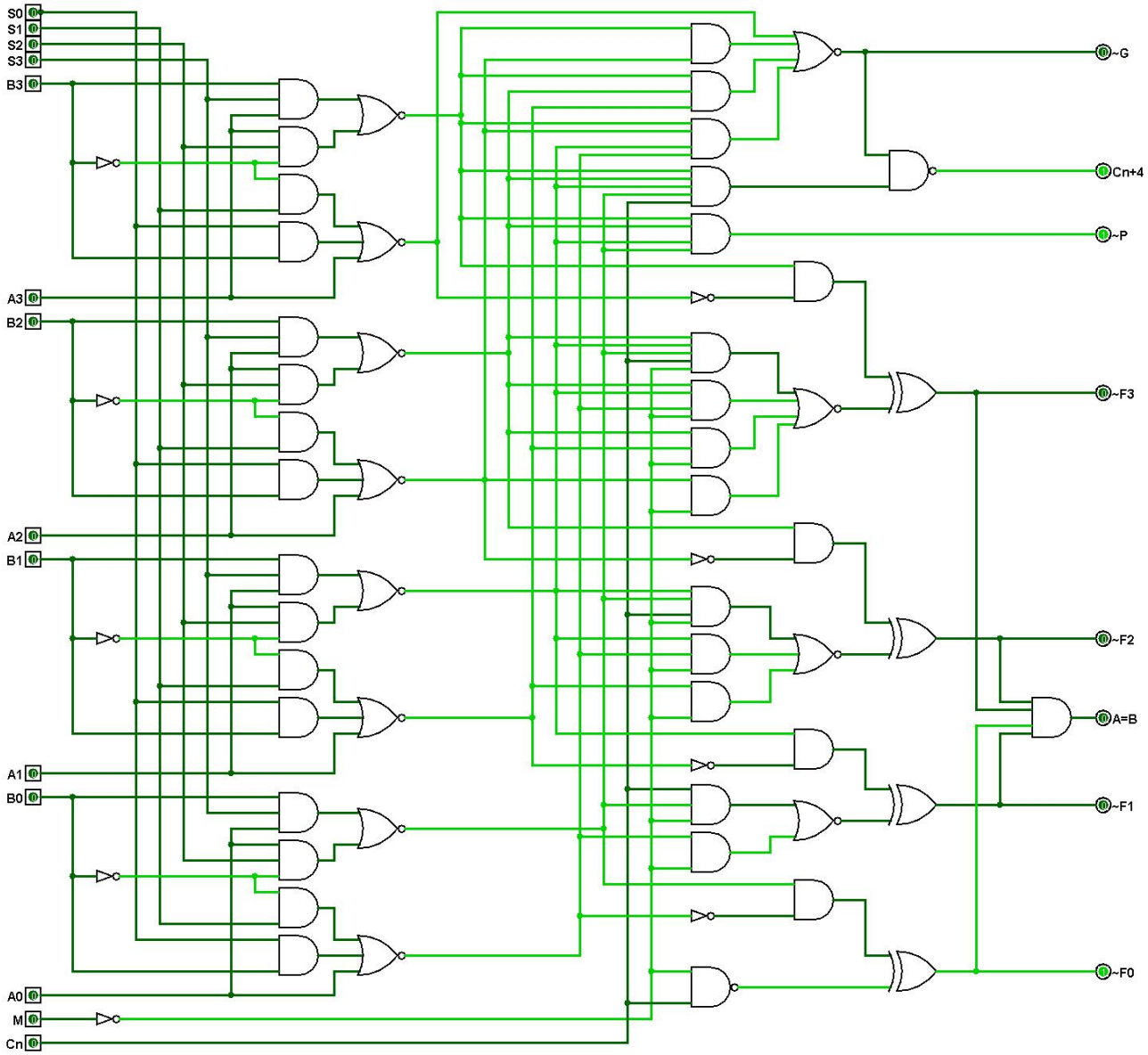
Übertragung eines ausgewählten Kanalsignals (von 4) mittels MUX und DEMUX.
Trick: Der höchstwertige Eingang (C) des Codewandlers wird als Dateneingang gewählt, B und A werden dagegen als Selektiereingänge eingesetzt:



Multiplexing kann zur Reduzierung von Übertragungsleitungen führen, vor allem bei grösserer Kanalzahl. Der Durchsatz bleibt allerdings auf einen Kanal beschränkt.

4. Arithmetische und logische Einheit (ALU)

Beispiel einer 4-Bit-ALU (Arithmetic Logic Unit 74F181, Philips) [3]



Functional Description (DM74LS181) [4]

The DM74LS181 is a 4-bit high speed parallel Arithmetic Logic Unit (ALU). Controlled by the four Function Select inputs ($S0$–$S3$) and the Mode Control input (M), it can perform all the 16 possible logic operations or 16 different arithmetic operations on active HIGH or active LOW operands. The Function Table lists these operations.
When the Mode Control input (M) is HIGH, all internal carries are inhibited and the device performs logic operations on the individual bits as listed. When the Mode Control input is LOW, the carries are enabled and the device performs arithmetic operations on the two 4-bit words. The device incorporates full internal carry lookahead and provides for either ripple carry between devices using the C_{n+4} output, or for carry lookahead between packages using the signals $\overline{P}$ (Carry Propagate) and $\overline{G}$ (Carry Generate). In the ADD mode, $\overline{P}$ indicates that $\overline{F}$ is 15 or more, while $\overline{G}$ indicates that $\overline{F}$ is 16 or more. In the SUBTRACT mode, $\overline{P}$ indicates that $\overline{F}$ is zero or less, while $\overline{G}$

indicates that $\overline{F}$ is less than zero. $\overline{P}$ and $\overline{G}$ are not affected by carry in. Carry lookahead can be provided at various levels and offers high speed capability over extremely long word lengths. The $A = B$ output from the device goes HIGH when all four $\overline{F}$ outputs are HIGH and can be used to indicate logic equivalence over four bits when the unit is in the subtract mode. The $A = B$ output is open-collector and can be wired-AND with other $A = B$ outputs to give a comparison for more than four bits. The $A = B$ signal can also be used with the C_{n+4} signal to indicate $A > B$ and $A < B$. The Function Table lists the arithmetic operations that are performed without a carry in. An incoming carry adds a one to each operation. Thus, select code LHHL generates A minus B minus 1 (2s complement notation) without a carry in and generates A minus B when a carry is applied. Because subtraction is actually performed by complementary addition (1s complement), a carry out means borrow; thus a carry is generated when there is no underflow and no carry is generated when there is underflow. As indicated, this device can be used with either active LOW inputs producing active LOW outputs or with active HIGH inputs producing active HIGH outputs. For either case the table lists the operations that are performed to the operands labeled inside the logic symbol.

Function Table

Mode Select Inputs				Active LOW Operands & F_n Outputs		Active HIGH Operands & F_n Outputs	
				Logic	Arithmetic (Note 2)	Logic	Arithmetic (Note 2)
S3	S2	S1	S0	(M = H)	(M = L) (C_n = L)	(M = H)	(M = L) (C_n = H)
L	L	L	L	$\overline{A}$	A minus 1	$\overline{A}$	A
L	L	L	H	$\overline{AB}$	AB minus 1	$\overline{A} + B$	$A + B$
L	L	H	L	$\overline{A} + \overline{B}$	$A\overline{B}$ minus 1	$\overline{A}B$	$A + \overline{B}$
L	L	H	H	Logic 1	minus 1	Logic 0	minus 1
L	H	L	L	$\overline{A} + \overline{B}$	A plus $(A + \overline{B})$	$\overline{AB}$	A plus $A\overline{B}$
L	H	L	H	$\overline{B}$	AB plus $(A + \overline{B})$	$\overline{B}$	$(A + B)$ plus $A\overline{B}$
L	H	H	L	$\overline{A} \oplus \overline{B}$	A minus B minus 1	$A \oplus B$	A minus B minus 1
L	H	H	H	$A + \overline{B}$	$A + \overline{B}$	$A\overline{B}$	$A\overline{B}$ minus 1
H	L	L	L	$\overline{A}B$	A plus $(A + B)$	$\overline{A} + B$	A plus AB
H	L	L	H	$A \oplus B$	A plus B	$\overline{A} \oplus \overline{B}$	A plus B
H	L	H	L	B	$A\overline{B}$ plus $(A + B)$	B	$(A + \overline{B})$ plus AB
H	L	H	H	$A + B$	$A + B$	AB	AB minus 1
H	H	L	L	Logic 0	A plus A (Note 1)	Logic 1	A plus A (Note 1)
H	H	L	H	$A\overline{B}$	AB plus A	$A + \overline{B}$	$(A + B)$ plus A
H	H	H	L	AB	$A\overline{B}$ minus A	$A + B$	$(A + \overline{B})$ plus A
H	H	H	H	A	A	A	A minus 1

Note 1: Each bit is shifted to the next most significant position.

Note 2: Arithmetic operations expressed in 2s complement notation.

[4]

H entspricht 1
L entspricht 0

Anhang

Tief-aktive und hoch-aktive Logik ist zusammen mit sprechenden Namen für Eingänge und Ausgänge von Bedeutung.

Beispiel für hoch-aktive Logik:

Enable	S1	S0	...
0	x	x	x
1	0	0	x
1	0	1	x
1	1	0	x
1	1	1	d

Tab. 1

Enable, S1 (= Select1) und S0 (= Select0) sind in hoch-aktiver Logik notierte Eingänge: Ist ihr Wert hoch (=1), so geschieht, was ihr Name besagt: Die Schaltung ist im Zustand "enable"; S1 und S0 wählen (= selektieren) zusammen den Ausgang der entsprechenden Nummer, etwa S1=1 und S0=1, also Ausgang 3.

Beispiel für tief-aktive Logik:

$\overline{\text{Enable}}$	...
1	
0	
0	
0	
0	

Tab. 2

Der Eingang $\overline{\text{Enable}}$ ist in tief-aktiver Logik notiert (Tab. 2): Ist sein Wert tief (= 0), so geschieht, was sein Name besagt: Die Schaltung ist im Zustand "enable". Um die tief-aktive Logik sichtbar zu machen, ist der Name mit "quer" versehen.

Ein Eingang in tief-aktiver Logik kann durch Umbenennen in hoch-aktive Logik geändert werden. $\overline{\text{Enable}}$ (Tab. 2) in Disable (Tab. 3) umbenannt, illustriert dies augenfällig. Mit Änderung des Namens und der Quer-/Nicht-Quer-Notation muss an den Tabellenwerten nichts geändert werden.

Disable	...
1	
0	
0	
0	
0	

Tab. 3

Was für die Eingänge notiert werden kann, ist auch auf die Ausgänge anwendbar. Z.B.:

Enable	LED_off
0	1
1	0

Tab. 4

Quellen

[1] Digitaltechnik, Klaus Fricke, Vieweg, 2005,
ISBN 3-528-33861-X

[2] Rechnerarchitektur, Helmut Malz, Vieweg, 2001,
ISBN 3-528-03379-7

[3] Arithmetic logic unit data sheet 74F181, p. 3, Philips Semiconductors FAST Products,
March 3, 1989

[4] Beschreibung, Tabelle: www.fairchildsemi.com