10. kombinatorische Komponenten

Darstellung natürlicher Zahlen

Stellenwertsysteme

- **Dezimalzahlen:**

Ziffern (engl. digit) 0 – 9; Basis 10

$154_{10} = 1 \cdot 100 + 5 \cdot 10 + 4 \cdot 1$
$= 1 \cdot 10^2 + 5 \cdot 10^1 + 4 \cdot 10^0$

- **Allgemein:**

Jede natürliche Zahl ist zu beliebiger natürlicher Basis *b* darstellbar durch:

$$z = \sum_{i=0}^{n-1} a_i \cdot b^i$$

$$= a_{n-1} \cdot b^{n-1} + \ldots + a_2 \cdot b^2 + a_1 \cdot b^1 + a_0 \cdot b^0$$

Darstellung natürlicher Zahlen

- **Binärzahlen**

Ziffern 0,1; Basis 2

$$10011010_2 = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$
$$= 1 \cdot 128 + 0 \cdot 64 + 0 \cdot 32 + 1 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 1$$
$$= 154_{10}$$

Jede Stelle nennt man bit (engl. binary digit, Binärziffer).

In Mathematik gibt es unendlich viele natürliche Zahlen. In Hardwaresystemen ist die Anzahl von darstellbaren Zahlen begrenzt. Meist Darstellung von Zahlen in Einheiten gleicher Länge (Wörter) bestehend aus n Bits, damit 2^n verschiedene Zahlen darstellbar. Üblich:

$2^8 = 256$
$2^{16} = 65.536$
$2^{32} = 4.294.967.296$

Darstellung natürlicher Zahlen

- **Hexadezimalzahlen**

Ziffern 0-9, A-F; Basis 16 (A steht für den Wert 10, ..., F für 15)

$39A_{16} = 3 \cdot 16^2 + 9 \cdot 16^1 + 10 \cdot 16^0$
$= 3 \cdot 256 + 9 \cdot 16 + 10 \cdot 1$
$= 922_{10}$

In Informatik wichtig, da einfache Umrechnung zwischen Hexadezimalzahlen und Binärzahlen.
Mit $2^4 = 16$ werden 4 Bit benötigt, um eine Hexadezimalziffer darzustellen.
Damit direkte Konvertierung einer Hexadezimalziffer in 4 Binärziffern und umgekehrt:

$39A_{16} = 0011.1001.1010_2$
$1110.1101.0101.0101_2 = ED55_{16}$

Addition natürlicher Zahlen

Addition zweier Zahlen:

$78_{10} + 19_{10} = (7 \cdot 10^1 + 8 \cdot 10^0) + (1 \cdot 10^1 + 9 \cdot 10^0)$
$= 7 \cdot 10^1 + 1 \cdot 10^1 + 8 \cdot 10^0 + 9 \cdot 10^0$
$= (7+1) \cdot 10^1 + (8+9) \cdot 10^0$ Übertrag
$= (7+1) \cdot 10^1 + 1 \cdot 10^1 + 7 \cdot 10^0$
$= (7+1+1) \cdot 10^1 + 7 \cdot 10^0$
$= 9 \cdot 10^1 + 7 \cdot 10^0$
$= 97_{10}$

$$\begin{array}{rcccc} & 7 & 8 \\ + & 1 & 9 \\ & \color{blue}{1} & \\ \hline & 9 & 7 \end{array}$$

$0011_2 + 1001_2 = (1 \cdot 2^1 + 1 \cdot 2^0) + (1 \cdot 2^3 + 1 \cdot 2^0)$
$= 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^0$
$= 1 \cdot 2^3 + 1 \cdot 2^1 + (1+1) \cdot 2^0$ Übertrag
$= 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^1$
$= 1 \cdot 2^3 + (1+1) \cdot 2^1$ Übertrag
$= 1 \cdot 2^3 + 1 \cdot 2^2$
$= 1100_2$

$$\begin{array}{rcccc} & 0 & 0 & 1 & 1 \\ + & 1 & 0 & 0 & 1 \\ & & \color{blue}{1} & \color{blue}{1} & \\ \hline & 1 & 1 & 0 & 0 \end{array}$$

Subtraktion natürlicher Zahlen

Für Subtraktion gibt es zwei Möglichkeiten:
1. Direkte Subtraktion

$$\begin{array}{rr} & 7\ 8 \\ - & 1\ 9 \\ & 1 \\ \hline & 5\ 9 \end{array} \qquad \begin{array}{rr} & 1\ 0\ 0\ 1 \\ - & 0\ 0\ 1\ 1 \\ & 1\ 1 \\ \hline & 0\ 1\ 1\ 0 \end{array}$$

2. Addition von 10er Komplement

$$\begin{aligned} 78_{10} - 19_{10} &= 78_{10} + (-19_{10}) \\ &= 78_{10} + (100_{10} - 19_{10}) \\ &= 78_{10} + 81_{10} \end{aligned}$$

$$\begin{array}{rr} & 7\ 8 \\ + & 8\ 1 \\ & 1 \\ \hline & 1\ 5\ 9 \end{array}$$

Komplement

Man unterscheidet zwischen (b-1)er und b-er Komplementdarstellung einer Zahl z mit n Stellen:

(b-1)er Komplement: $K_{b-1}(z) = b^n - 1 - z$

b-er Komplement: $K_b(z) = b^n - z = b^n - 1 - z + 1 = K_{b-1} + 1$

9er Komplement bei Dezimalzahlen:

$0 \rightarrow 9$
$1 \rightarrow 8$
$2 \rightarrow 7$
$3 \rightarrow 6$
$4 \rightarrow 5$
$5 \rightarrow 4$
$6 \rightarrow 3$
$7 \rightarrow 2$
$8 \rightarrow 1$
$9 \rightarrow 0$

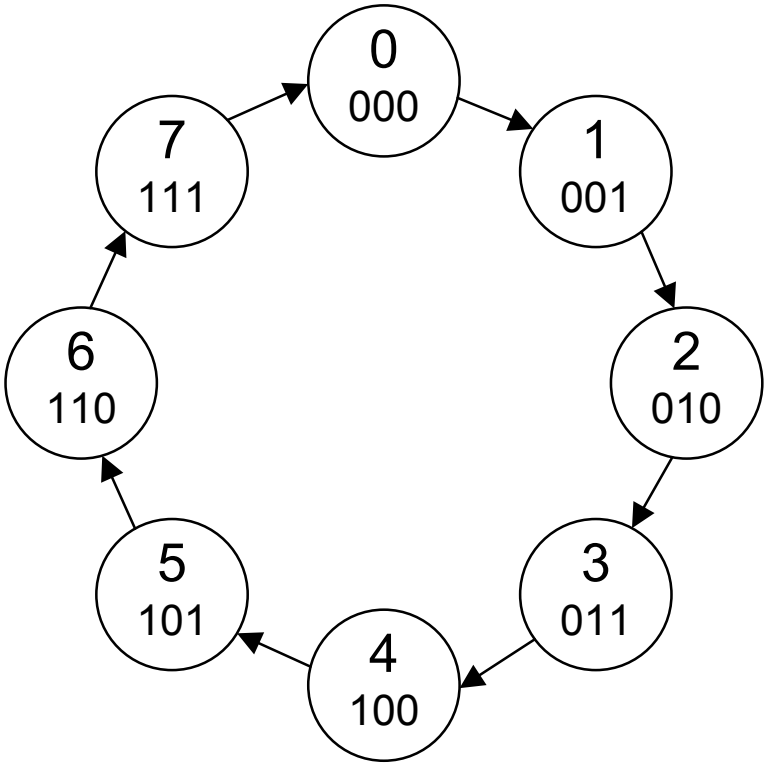
1er Komplement bei Binärzahlen:

$0 \rightarrow 1$
$1 \rightarrow 0$

entspricht Negation!

Zahlenkreis

Durch die feste Bitbreite bewegt man sich bei arithmetischen Operationen auf einem geschlossenen Zahlenkreis.



Hierarchischer Entwurf

Statt ein zu konstruierendes System vollständig neu zu entwickeln, wird es aus bestehenden Untereinheiten (Komponenten) , ähnlich wie bei einem Baukasten, zusammengesetzt. Diese können wiederum aus anderen Komponenten aufgebaut sein.

Durch den hierarchischen Aufbau des Systems wird die Komplexität des Gesamtsystems auf die einzelnen Komponenten verteilt. Die Benutzung vorgefertigter Komponenten vereinfacht den Aufbau und erlaubt eine Wiederverwendung bereits entwickelter Komponenten. Die einzelnen Komponenten lassen sich unabhängig von einander entwickeln und testen.

In Praxis gibt es immer wieder gleiche Problemstellungen. Für diese haben sich oft verwendete Standard-Komponenten etabliert.

In diesem Kapitel werden häufig vorkommende **kombinatorische** Komponenten vorgestellt.

Halbaddierer

Addition einer Binärstelle zweier Zahlen **x** und **y**. Ergebnis besteht aus Summenbit **s** und Übertragsbit **c** (engl. carry).
Bei Addition gibt es vier Möglichkeiten:

x	0	0	1	1
+ y	+ 0	+ 1	+ 0	+ 1
c s	0 0	0 1	0 1	1 0

x	y	s	c
0	0		
0	1		
1	0		
1	1		

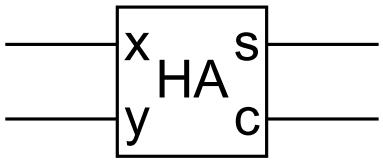
Halbaddierer

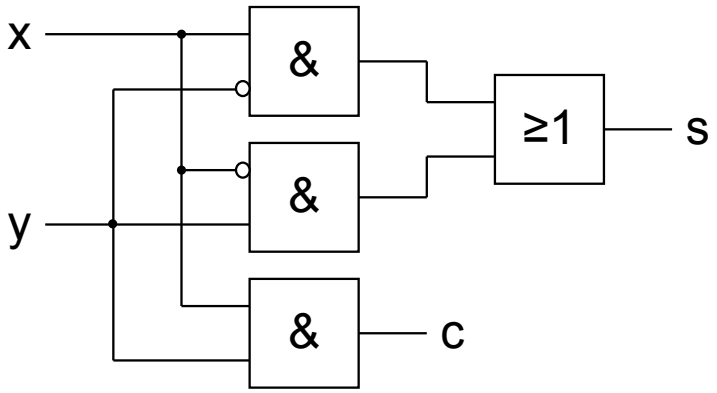
Addition einer Binärstelle zweier Zahlen **x** und **y**. Ergebnis besteht aus Summenbit **s** und Übertragsbit **c** (engl. carry).
Bei Addition gibt es vier Möglichkeiten:

x	0	0	1	1
+ y	+ 0	+ 1	+ 0	+ 1
c s	0 0	0 1	0 1	1 0

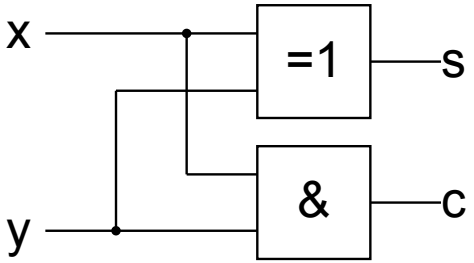
x	y	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Realisierung der Addition durch Halbaddierer:





Alternativ:



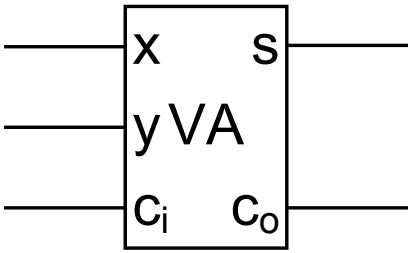
$s = (x' * y) + (x * y')$

$c = x * y$

Volladdierer

Bei der Addition mehrstelliger Zahlen muss bei der Berechnung einer Binärstelle der Übertrag von der vorherigen Stelle berücksichtigt werden.

Volladdierer zur Addition einer Binärstelle zweier Zahlen **x** und **y** unter Berücksichtigung des Eingangsübertrags $\mathbf{c_i}$.

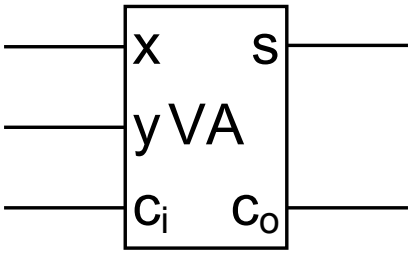


c_i	x	y	s	c_o
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Volladdierer

Bei der Addition mehrstelliger Zahlen muss bei der Berechnung einer Binärstelle der Übertrag von der vorherigen Stelle berücksichtigt werden.

Volladdierer zur Addition einer Binärstelle zweier Zahlen **x** und **y** unter Berücksichtigung des Eingangsübertrags $\mathbf{c_i}$.



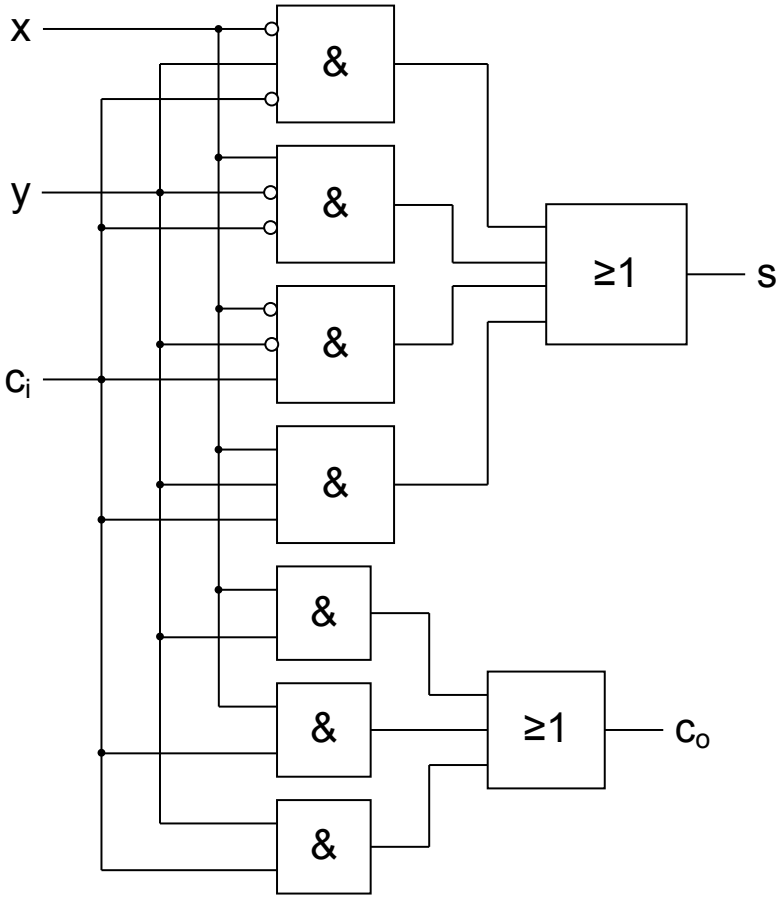
c_i	x	y	s	c_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s = (c_i \text{ ‘ } * x \text{ ‘ } * y) + (c_i \text{ ‘ } * x * y \text{ ‘}) + (c_i * x \text{ ‘ } * y \text{ ‘}) + (c_i * x * y)$$

$$c_o = (c_i * y) + (c_i * x) + (x * y)$$

Volladdierer

Schaltnetz des Volladdierers:



Unterschiedliche Verzögerungszeit für beide Ausgänge:

$$td_{x,y,ci \rightarrow s} = td_{UND3} + td_{ODER4}$$

$$td_{x,y,ci \rightarrow co} = td_{UND2} + td_{ODER3}$$

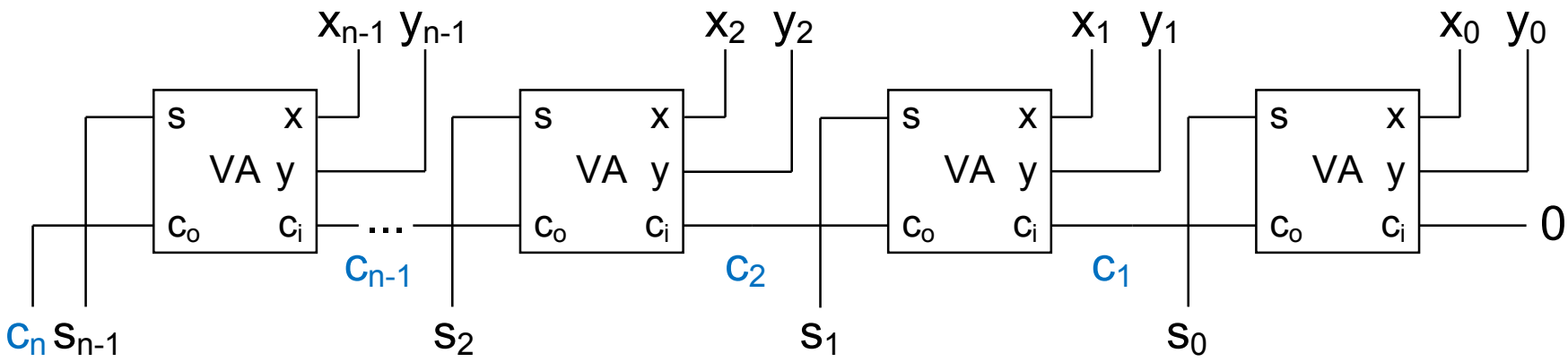
Carry-ripple Addierer

Zusammenschaltung von Volladdierern zur Addition mehrstelliger Binärzahlen.

	x_{n-1}	…	x_3	x_2	x_1	x_0
+	y_{n-1}	…	y_3	y_2	y_1	y_0
	c_{n-1}	…	c_3	c_2	c_1	
c_n	s_{n-1}	…	s_3	s_2	s_1	s_0

Übertrag wird von Stelle zu Stelle durchgereicht. Durch das Durchlaufen (ripple – dahinplätschern) des Übertragsbits wird vollständige Berechnung langsam:

$$td_{add} = td_{x,y \rightarrow co} + (n-2) \bullet td_{ci \rightarrow co} + td_{ci \rightarrow s}$$



Carry-ripple Addierer

Carry-ripple-Addierer hat folgende Komplexität:

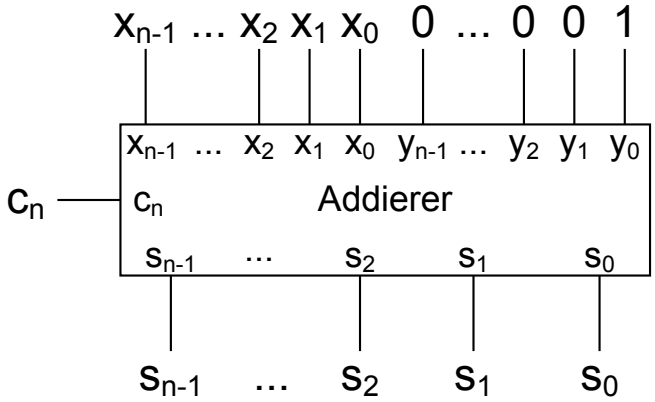
- Schaltungstiefe (Laufzeit): O(n)
- Flächenbedarf: O(n)

$$td_{add} = td_{x,y \rightarrow co} + (n-2) \bullet td_{ci \rightarrow co} + td_{ci \rightarrow s}$$

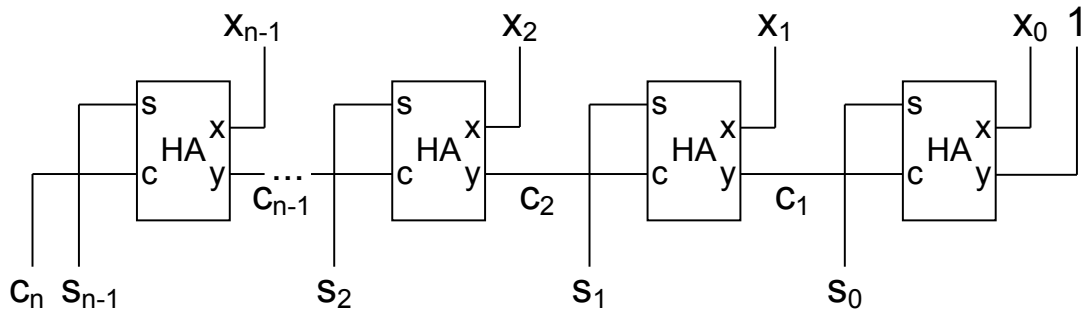
Um einen schnelleren Addierer zu entwerfen, könnte man prinzipiell auch eine Wahrheitstabelle für alle Eingänge aufstellen und ein Minimierungsverfahren aus Kapitel 3 verwenden, welches immer ein zweistufiges Schaltnetz erzeugt. Allerdings ist dies für große Bitbreiten zu aufwendig und die daraus entstehenden Schaltnetze zu groß:
Ein 16-Bit Addierer hat bereits 32 Eingänge und dadurch 2^{32} Tabelleneinträge. Der Flächenbedarf steigt mit $O(2^n)$. Ein 32-Bit Addierer bräuchte schon über 100 Milliarden Transistoren.

Inkrementierer

Oft verwendete Operation ist die Erhöhung einer n-Bit Binärzahl um den Wert 1. Hierzu kann ein n-Bit Addierer benutzt werden, dessen zweiter Eingang eine Konstante erhält:



Aufgrund der festen Werten am zweiten Eingang kann die Schaltung des Addierers optimiert werden. So bestehen die einzelnen Stufen nur noch aus Halbaddierern:



Subtrahierer

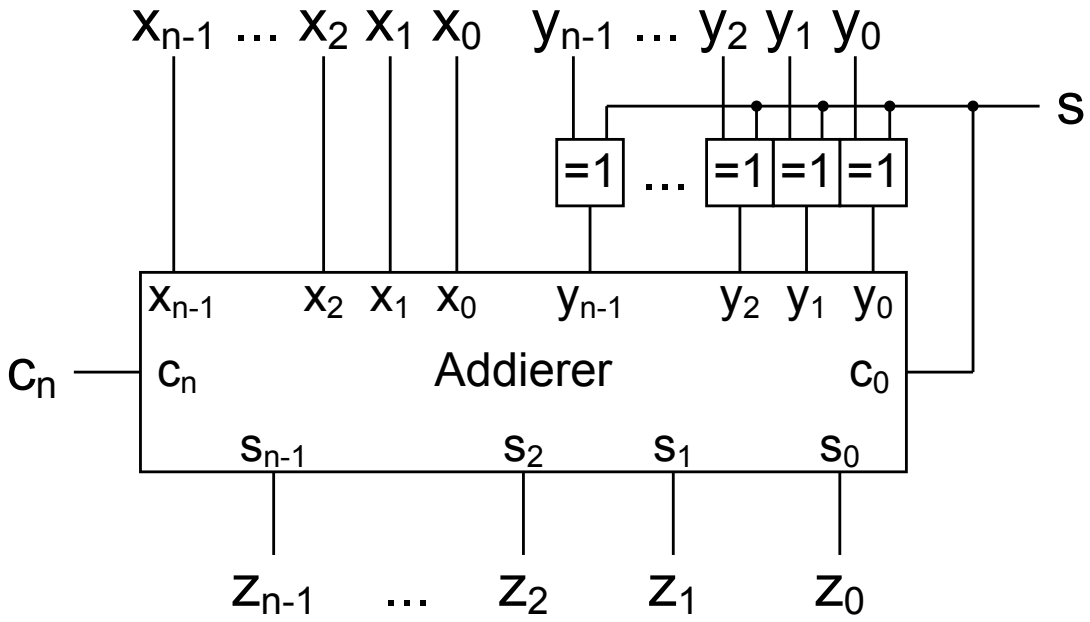
Gilt allgemein: Jeder Addierer kann auch als Subtrahierer benutzt werden, wenn y-Eingang negiert wird und c_0 mit 1 gespeist wird.

Diese Realisierung entspricht Subtraktion durch Addition des 2er Komplements!

$$
\begin{array}{lcccccc}
 & x_{n-1} & \dots & x_3 & x_2 & x_1 & x_0 \\
+ & \overline{y_{n-1}} & \dots & \overline{y_3} & \overline{y_2} & \overline{y_1} & \overline{y_0} \\
+ & & & & & & 1 \\
\hline
 & d_{n-1} & \dots & d_3 & d_2 & d_1 & d_0
\end{array}
$$

Addierer/Subtrahierer

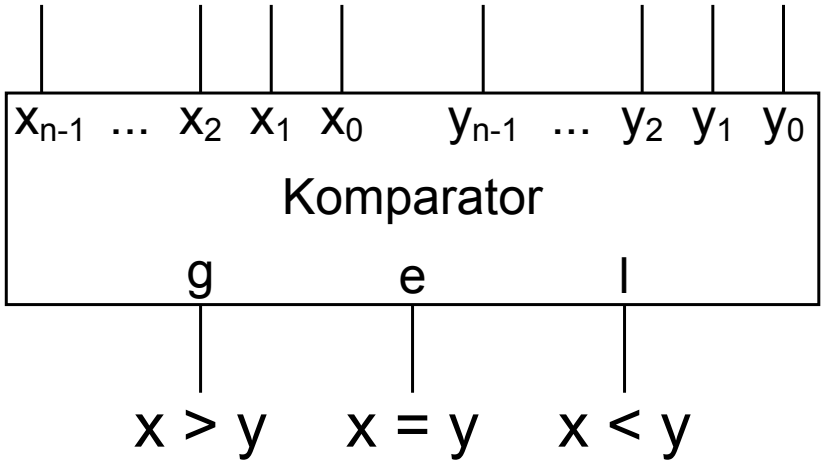
Da sowohl Addition als auch Subtraktion oft benötigt werden, werden beide Operationen in einer gemeinsamen Komponente zusammengefasst: dem Addierer/Subtrahierer. Eine Steuerleitung wählt die Operation aus.



s	Operation
0	z = x + y
1	z = x - y

Komparator

Der Komparator vergleicht zwei n-stellige Binärzahlen **x** und **y** und meldet, ob x **>** y, x **=** y oder x **<** y ist.



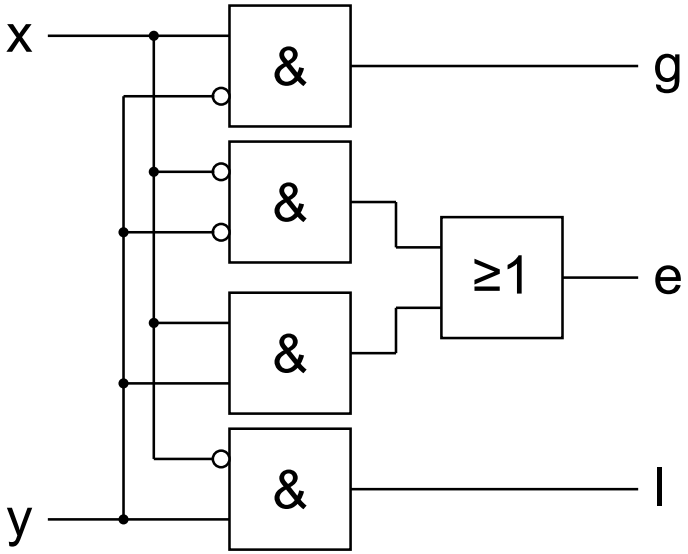
Weitere Vergleiche durch Verknüpfung dieser Relationen möglich:
x ≠ y : (x = y) ‘
x ≥ y : (x > y) + (x = y) oder (x < y) ‘
x ≤ y : (x < y) + (x = y) oder (x > y) ‘

1 Bit Komparator

Der 1 Bit Komparator vergleicht zwei einstellige Binärzahlen x und y.

x	y	g (x > y)	e (x = y)	l (x < y)
0	0			
0	1			
1	0			
1	1			

$g = x * y'$
$e = (x' * y') + (x * y)$
$l = (x' * y)$

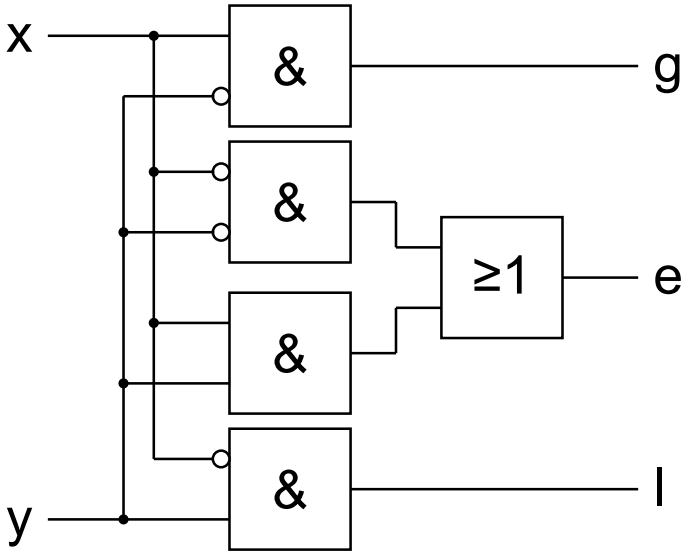


1 Bit Komparator

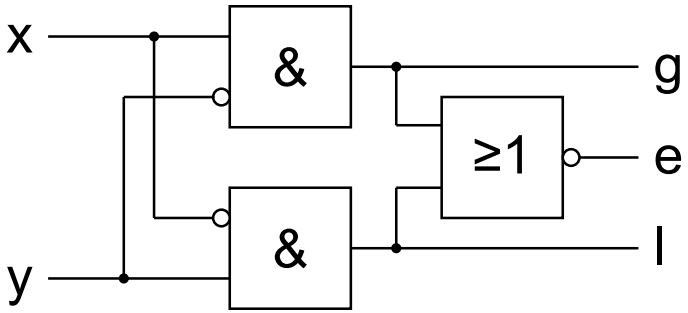
Der 1 Bit Komparator vergleicht zwei einstellige Binärzahlen x und y.

x	y	g (x > y)	e (x = y)	l (x < y)
0	0	0	1	0
0	1	0	0	1
1	0	1	0	0
1	1	0	1	0

$g = x * y'$
$e = (x' * y') + (x * y)$
$l = (x' * y)$



besser:
$e = ((x' * y) + (x * y'))'$



1 Bit Komparator

Erweiterung zum Vergleich mehrstelliger Binärzahlen:

Beispiel: 3 Bit Binärzahlen ($x_2\ x_1\ x_0$) und ($y_2\ y_1\ y_0$)

x = y : $e = (x_2 = y_2) * (x_1 = y_1) * (x_0 = y_0)$
$= e_2\hat{} * e_1\hat{} * e_0\hat{}$
$= (e_2\hat{} * e_1\hat{}) * e_0\hat{}$

x > y : $g = (x_2 > y_2) + ((x_2 = y_2) * (x_1 > y_1)) + ((x_2 = y_2) * (x_1 = y_1) * (x_0 > y_0))$
$= g_2\hat{} + e_2\hat{} * g_1\hat{} + (e_2\hat{} * e_1\hat{}) * g_0\hat{}$

x < y : $l = (x_2 < y_2) + ((x_2 = y_2) * (x_1 < y_1)) + ((x_2 = y_2) * (x_1 = y_1) * (x_1 < y_1)$
$= l_2\hat{} + e_2\hat{} * l_1\hat{} + (e_2\hat{} * e_1\hat{}) * l_0\hat{}$

Allgemein:

$e = e_0$

$g = g_{n-1} + \ldots + g_1 + g_0$

$l = l_{n-1} + \ldots + l_1 + l_0$

mit:

$e_i = e_{i+1} * e_i\hat{} \qquad e_n = 1$

$g_i = e_{i+1} * g_i\hat{}$

$l_i = e_{i+1} * l_i\hat{}$

1 Bit Komparator mit Verkettungsmöglichkeit

Allgemein:

$e = e_0$

$g = g_{n-1} + ... + g_1 + g_0$

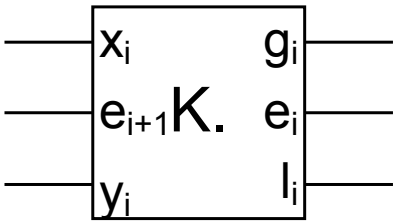
$l = l_{n-1} + ... + l_1 + l_0$

mit:

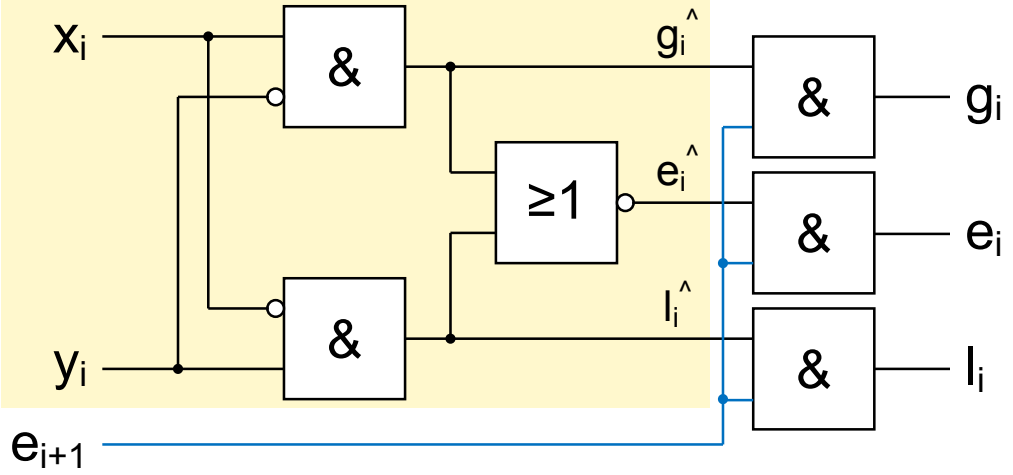
$e_i = e_{i+1} * \hat{e_i}$ $\quad e_n = 1$

$g_i = e_{i+1} * \hat{g_i}$

$l_i = e_{i+1} * \hat{l_i}$



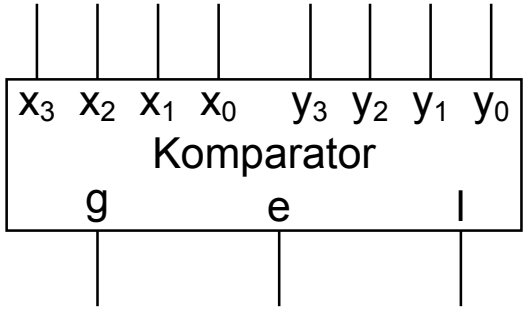
Verwendung von vorigem 1 Bit Komparator

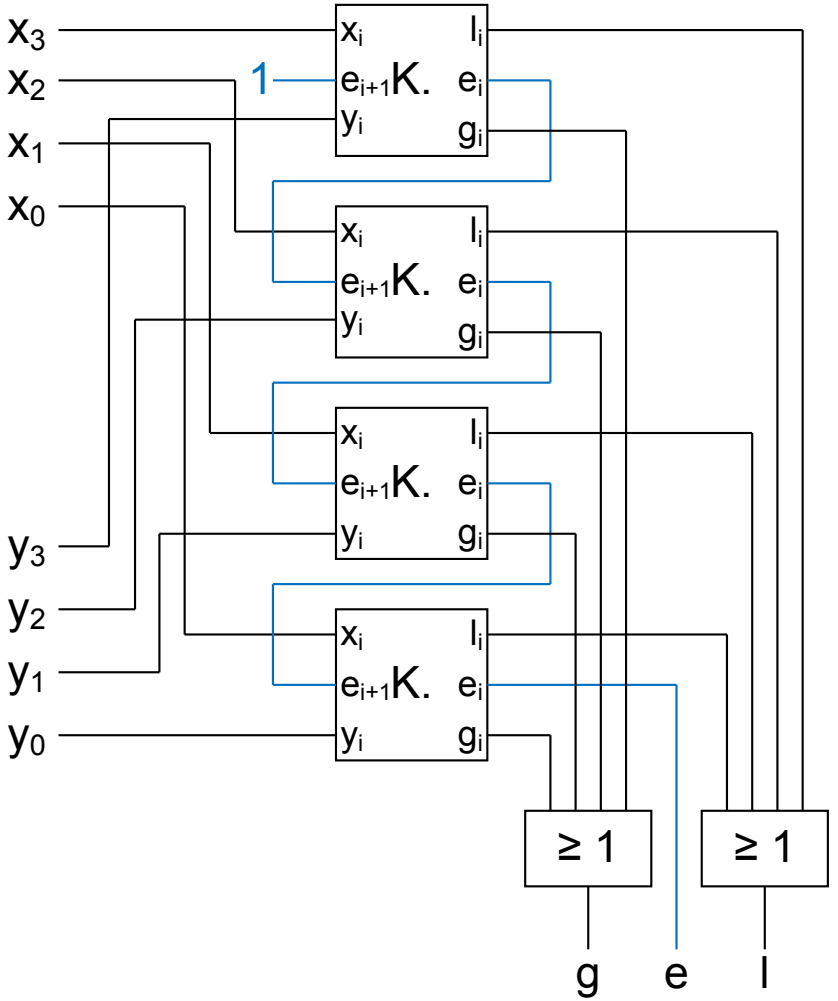


Komparator

Zusammenschaltung der 1 Bit Komparatoren zu einem 4 Bit Komparator.

$$e = e_0$$
$$g = g_{n-1} + ... + g_1 + g_0$$
$$l = l_{n-1} + ... + l_1 + l_0$$

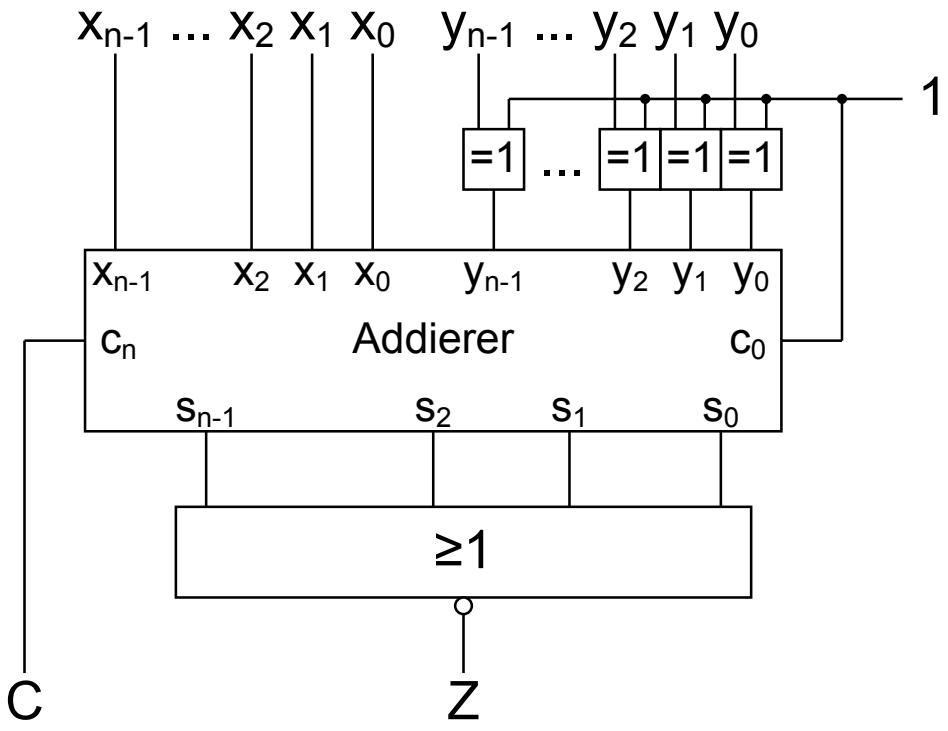




Komparator

Vergleichsoperationen per Subtraktion (Addition des 2er Komplements) und Auswertung des Übertrags (carry) und Test des Ergebnisses auf 0 möglich.

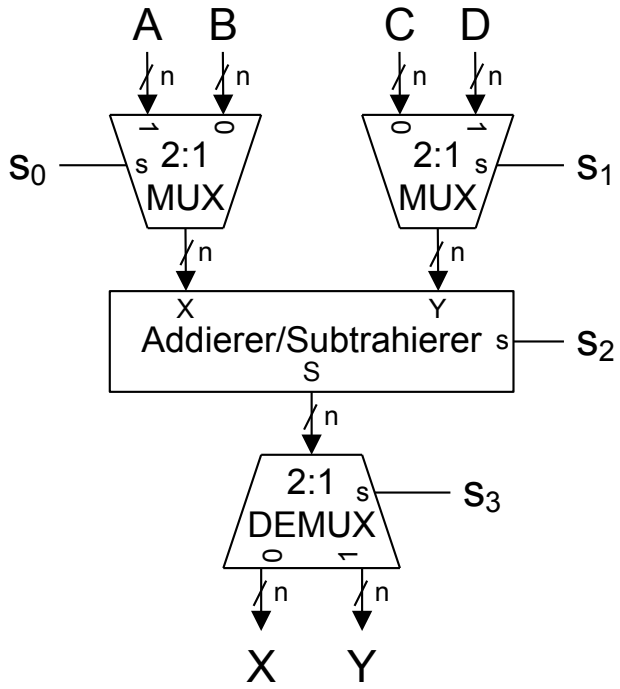
$x \geq y : C$
$x = y : Z$
$x \neq y : Z'$
$x < y : C'$
$x > y : C * Z'$
$x \leq y : C' + Z$



Datenpfad

Durch Multiplexer und Demultiplexer Bausteine kann der Pfad von Datenwörtern zu funktionalen Einheiten (z.B. Addierern) bestimmt werden. Die Zusammenschaltung solcher Komponenten nennt man Datenpfad. Steuerleitungen bestimmen die Funktion des Datenpfads.

Beispiel:

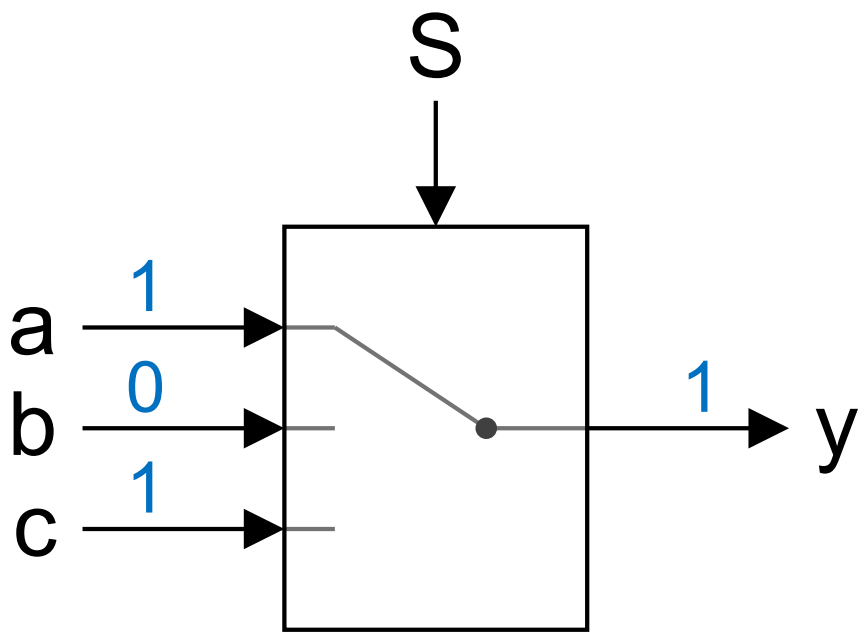


Funktionen des Datenpfads z.B.:

- X := A + C
- Y := B - D
- X := A - D

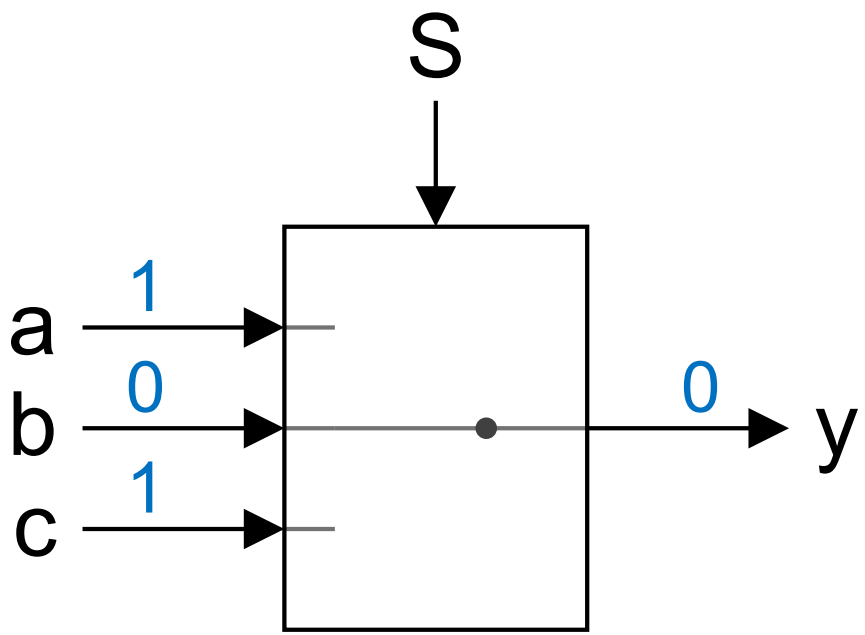
Multiplexer

Ein Multiplexer dient zum Durchschalten eines von mehreren Eingängen auf einen Ausgang. Der Eingang wird dabei über eine Auswahlleitung **s** (engl. select) festgelegt.



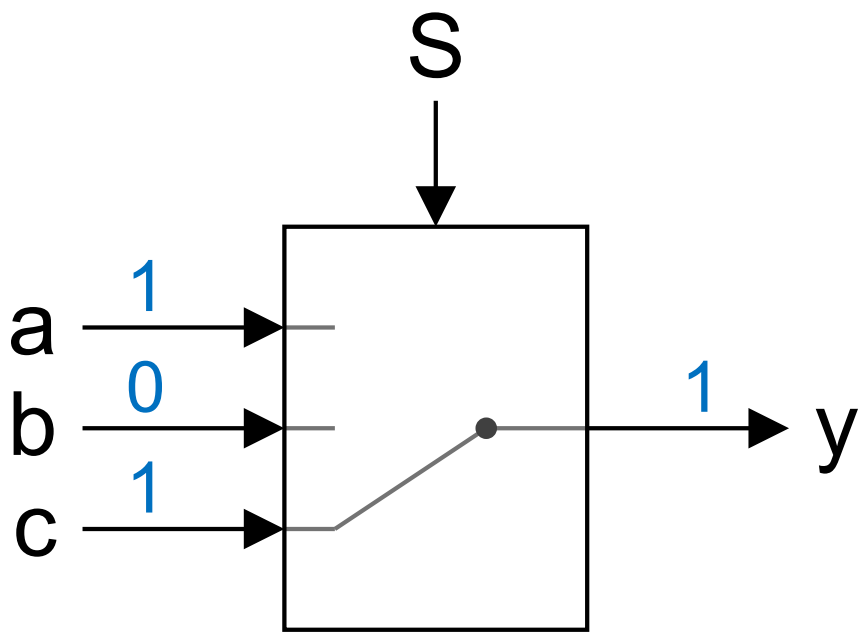
Multiplexer

Ein Multiplexer dient zum Durchschalten eines von mehreren Eingängen auf einen Ausgang. Der Eingang wird dabei über eine Auswahlleitung **s** (engl. select) festgelegt.



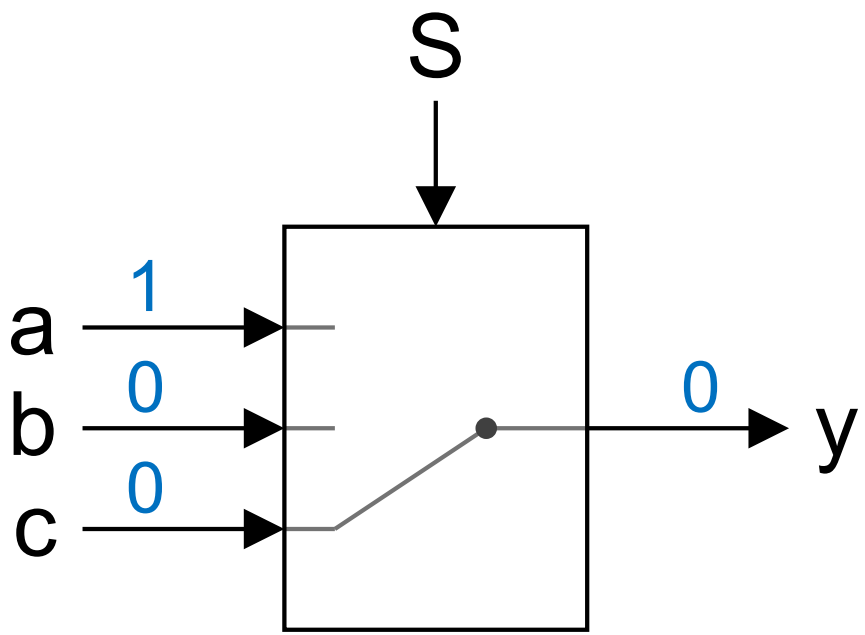
Multiplexer

Ein Multiplexer dient zum Durchschalten eines von mehreren Eingängen auf einen Ausgang. Der Eingang wird dabei über eine Auswahlleitung **s** (engl. select) festgelegt.

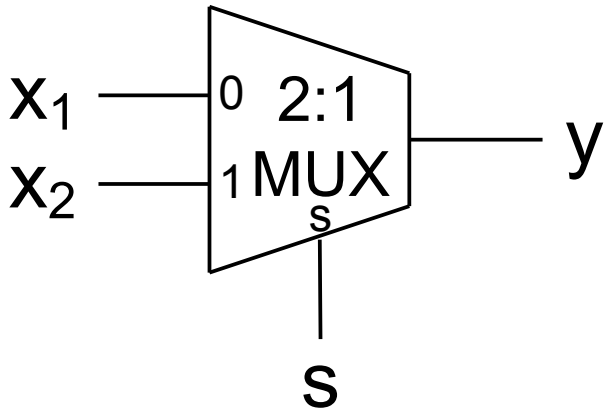


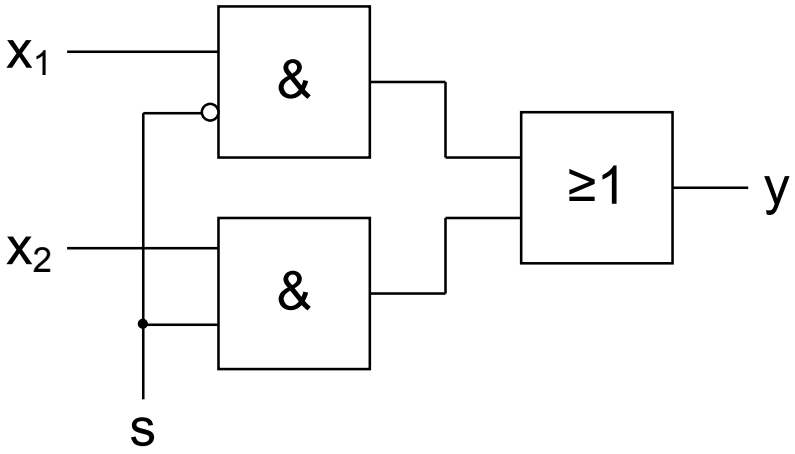
Multiplexer

Ein Multiplexer dient zum Durchschalten eines von mehreren Eingängen auf einen Ausgang. Der Eingang wird dabei über eine Auswahlleitung **s** (engl. select) festgelegt.



2x1 Multiplexer



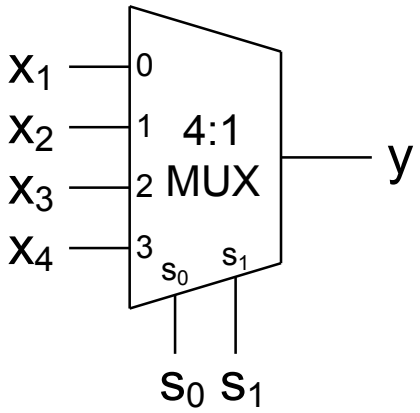


s	x_2	x_1	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

s	y
0	x_1
1	x_2

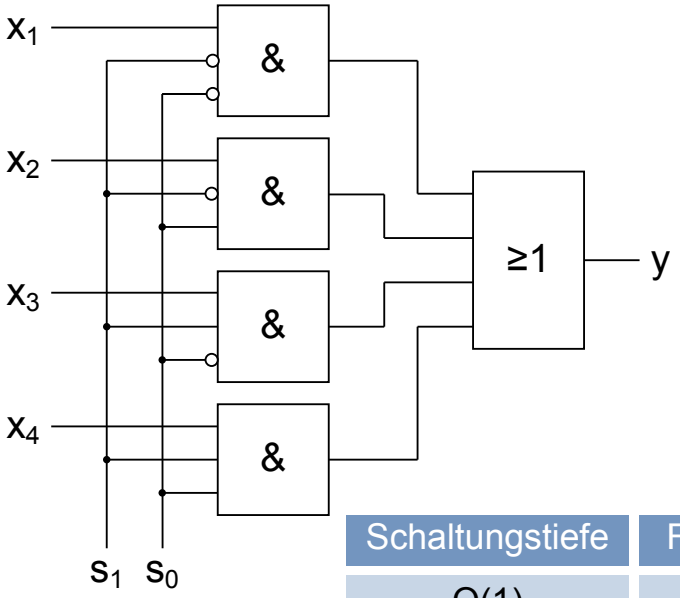
$$y = (s' * x_1 + s * x_2)$$

4x1 Multiplexer

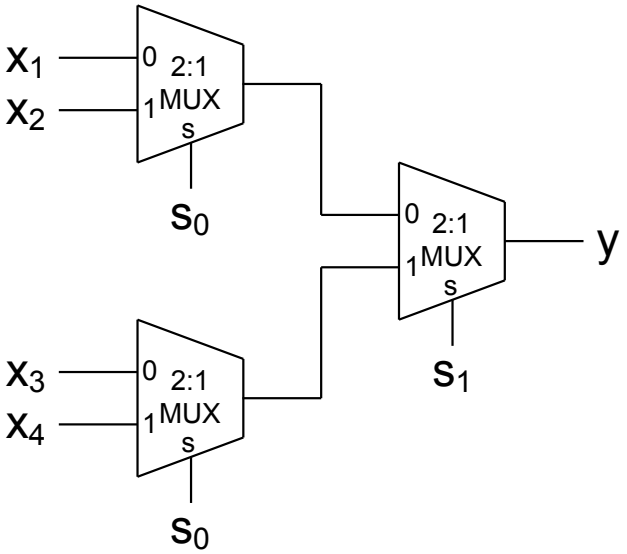


s_1	s_0	y
0	0	x_1
0	1	x_2
1	0	x_3
1	1	x_4

$$y = (s_1 \text{‘} * s_0 \text{‘} * x_1) + (s_1 \text{‘} * s_0 * x_2) + (s_1 * s_0 \text{‘} * x_3) + (s_1 * s_0 * x_4)$$



Schaltungstiefe	Flächenbedarf
O(1)	O(n log n)

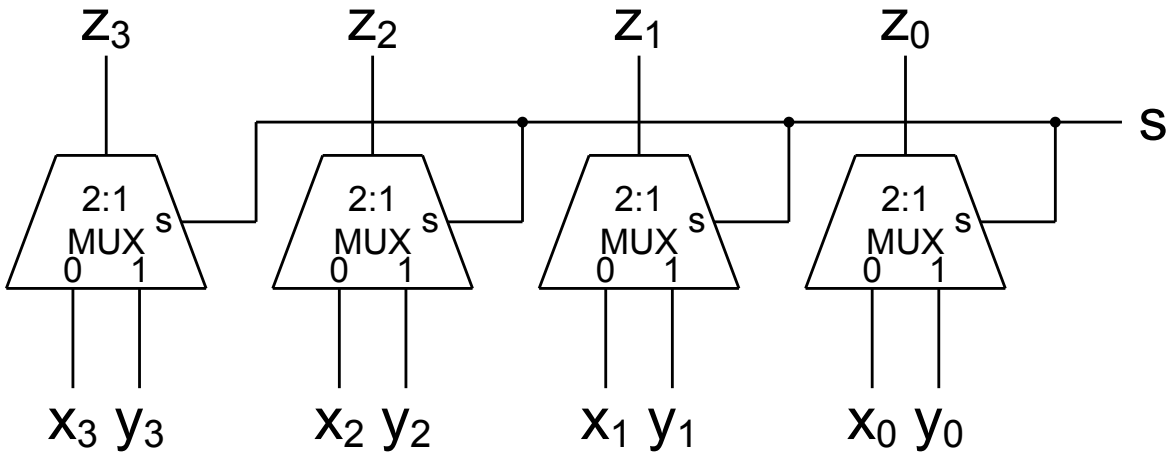


Schaltungstiefe	Flächenbedarf
O(log n)	O(n)

Datenbusse

Multiplexer wählen einen von mehreren Eingängen aus und leiten ihn an den Ausgang. Meistens werden dabei nicht nur einzelne Datenleitungen geschaltet, sondern ganze Datenwörter bestehend aus n Bit. Dazu werden n Multiplexer parallel geschaltet, welche mit der gleichen select-Leitung verbunden sind.

Beispiel: 4 Bit 2x1 MUX



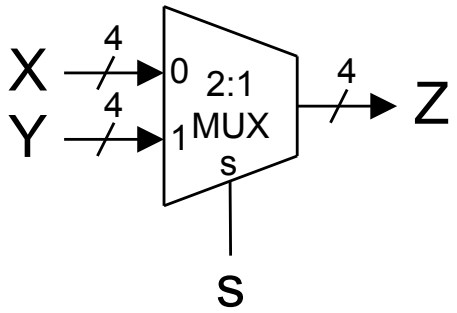
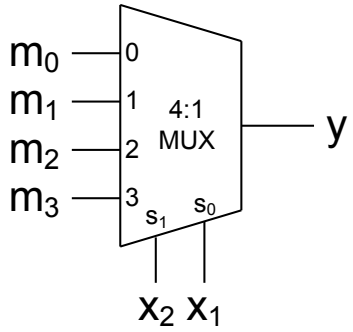


Abbildung von Funktionen mit MUX

Multiplexer lassen sich benutzen, um sämtliche Funktionen darzustellen, wie bereits bei der FPGA-Technologie gezeigt.

Beispiel: Funktion von zwei Variablen $y = f(x_1, x_2)$

x_2	x_1	$y = f(x_1, x_2)$
0	0	0
0	1	1
1	0	0
1	1	0



Aufteilung der Wahrheitstabelle in 2 Teile:

x_1	$y = f_{x2=0}$
0	0
1	1

x_1

x_1	$y = f_{x2=1}$
0	0
1	0

0

x_2	y
0	x_1
1	0

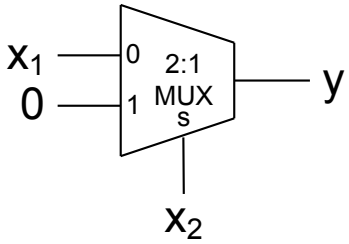
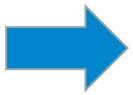


Abbildung von Funktionen mit MUX

Beispiel: Funktion von drei Variablen $y = f(x_1, x_2, x_3)$

x_3	x_2	x_1	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

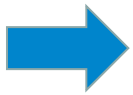


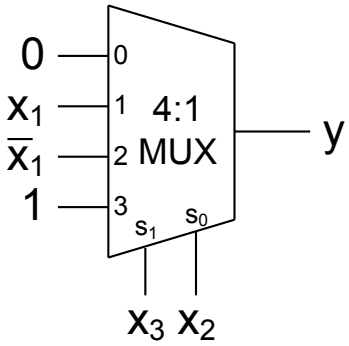
x_1	$y = f_{x2 = 0, x3 = 0}$
0	0
1	0

x_1	$y = f_{x2 = 1, x3 = 0}$
0	0
1	1

x_1	$y = f_{x2 = 0, x3 = 1}$
0	1
1	0

x_1	$y = f_{x2 = 1, x3 = 1}$
0	1
1	1





MUX und Shannon-Entwicklung

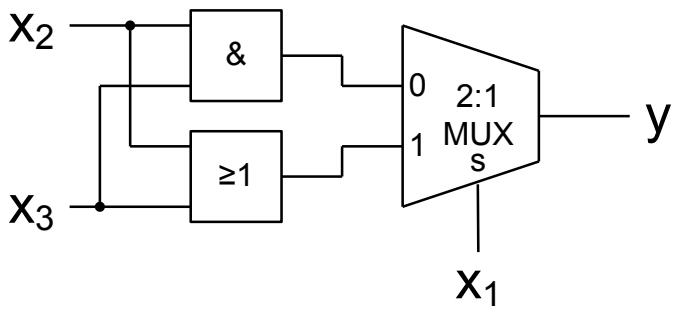
Erinnerung: Entwicklungssatz von Shannon

$$f(x_1, x_2, \ldots, x_n) = (x_i' * f_{xi=0}) + (x_i * f_{xi=1})$$

Beispiel: $y = f(x_1, x_2, x_3) = (x_1 * x_2) + (x_2 * x_3) + (x_1 * x_3)$

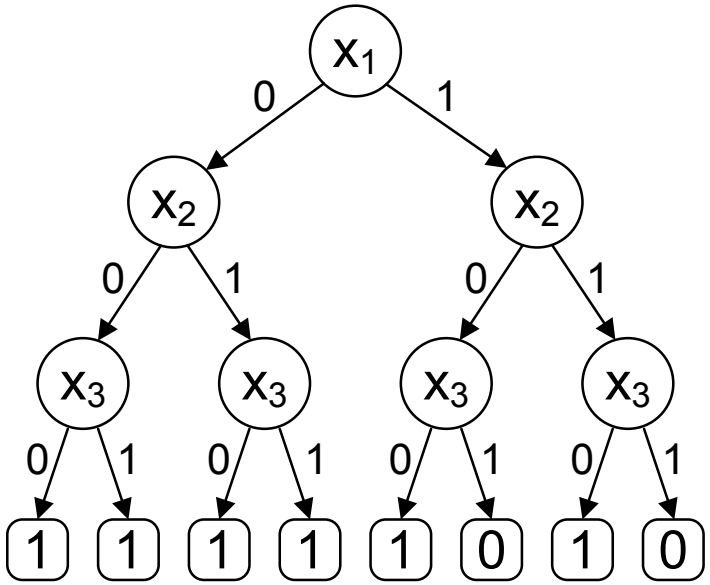
Entwicklung nach x_1:

$$\begin{aligned} y = f(x_1, x_2, x_3) &= (x_1' * f_{x1=0}) + (x_1 * f_{x1=1}) \\ &= [x_1' * (x_2 * x_3)] + [x_1 * (x_2 + (x_2 * x_3) + x_3)] \\ &= [x_1' * (x_2 * x_3)] + [x_1 * (x_2 + x_3)] \end{aligned}$$

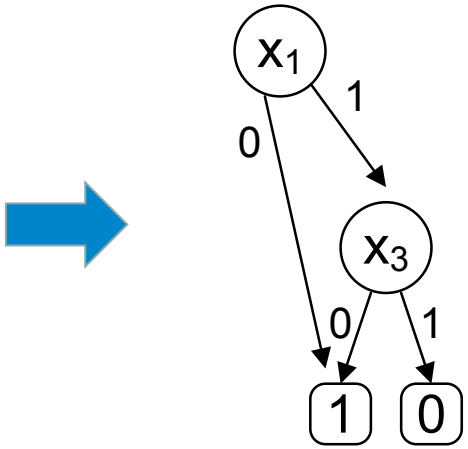


MUX und BDDs

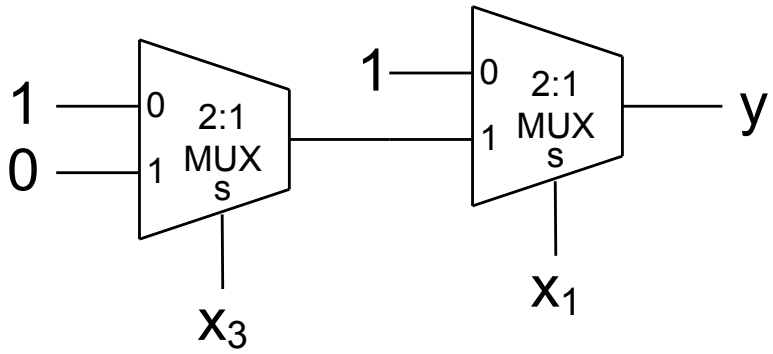
Beispiel aus Kapitel 2:



konnte reduziert werden auf:

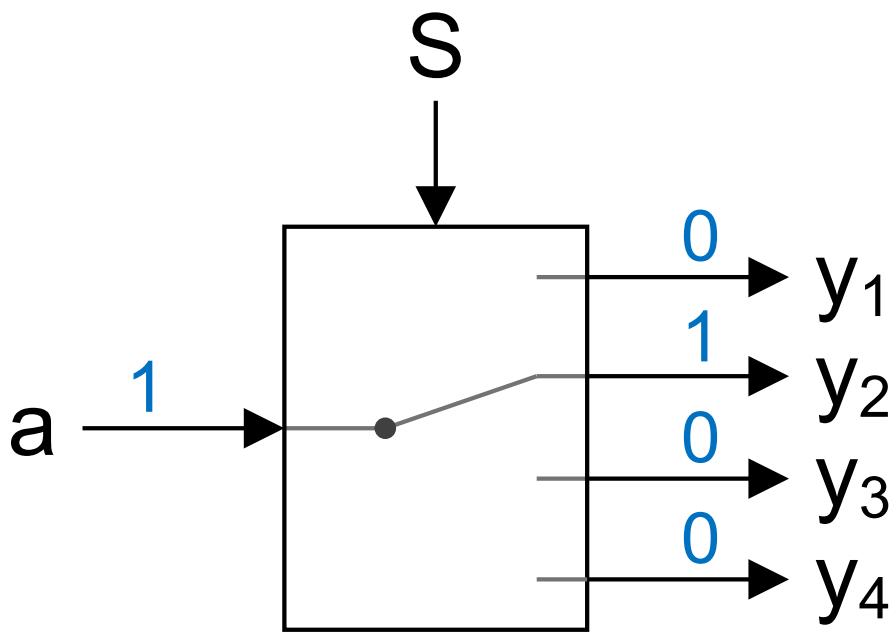


Direkte Umsetzung mit Multiplexern:



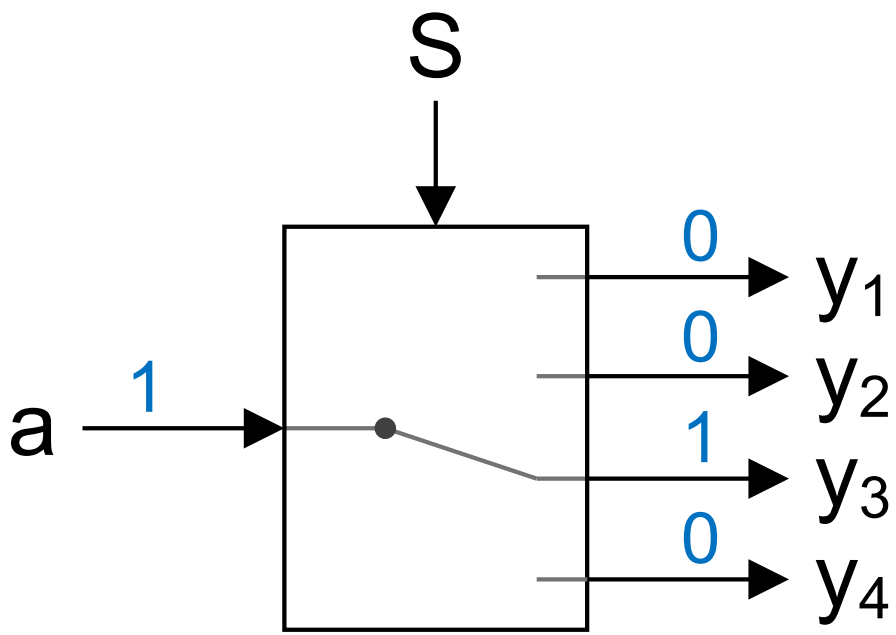
Demultiplexer

Demultiplexer sind das Gegenstück zu Multiplexern. Sie beschalten eine von mehreren Ausgangsleitungen mit dem Eingangssignal. Die anderen Ausgänge werden zu 0 gesetzt.

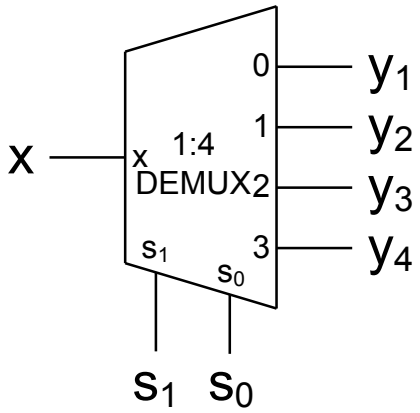


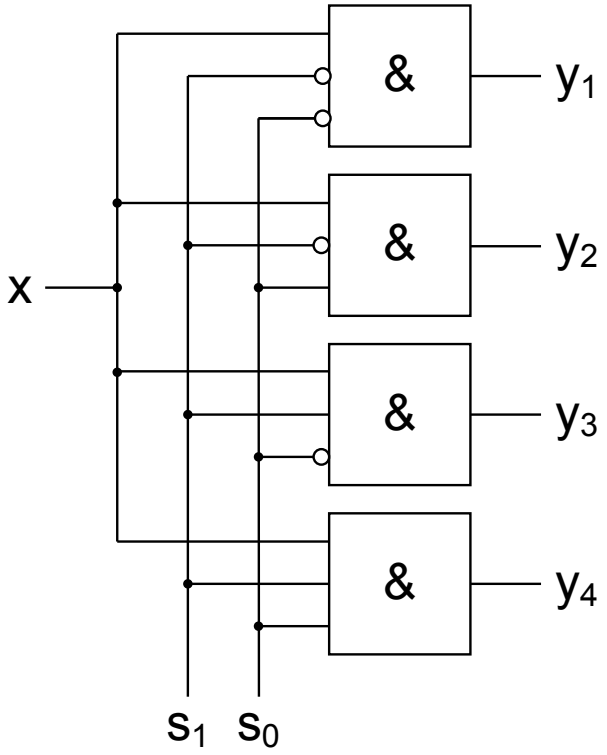
Demultiplexer

Demultiplexer sind das Gegenstück zu Multiplexern. Sie beschalten eine von mehreren Ausgangsleitungen mit dem Eingangssignal. Die anderen Ausgänge werden zu 0 gesetzt.



1x4 Demultiplexer



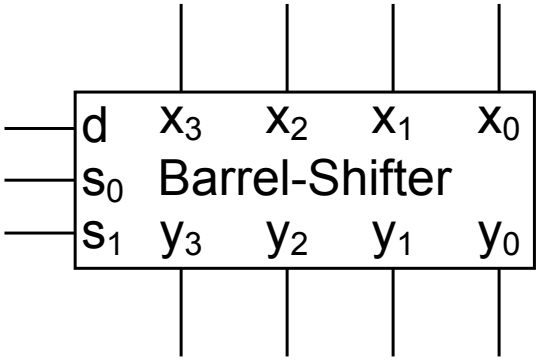


s_1	s_0	y_1	y_2	y_3	y_4
0	0	x	0	0	0
0	1	0	x	0	0
1	0	0	0	x	0
1	1	0	0	0	x

Barrel-Shifter

Barrel-Shifter werden benutzt, um n-stellige Datenwörter um eine oder mehrere Stellen nach links oder rechts zu verschieben (shiften).

Beispiel: 4 Bit Barrel-Shifter



d	s_1	s_0	y_3	y_2	y_1	y_0
0	0	0				
0	0	1				
0	1	0				
0	1	1				
1	0	0				
1	0	1				
1	1	0				
1	1	1				

rechts schieben

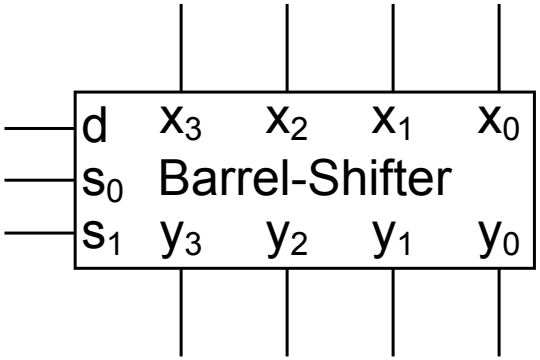
links schieben

Der d-Eingang wählt die Richtung (direction) aus, in die geschoben werden soll. Die beiden anderen Steuerleitungen geben an, um wieviel Stufen geschoben wird. Frei werdende Bitstellen werden mit 0 aufgefüllt.

Barrel-Shifter

Barrel-Shifter werden benutzt, um n-stellige Datenwörter um eine oder mehrere Stellen nach links oder rechts zu verschieben (shiften).

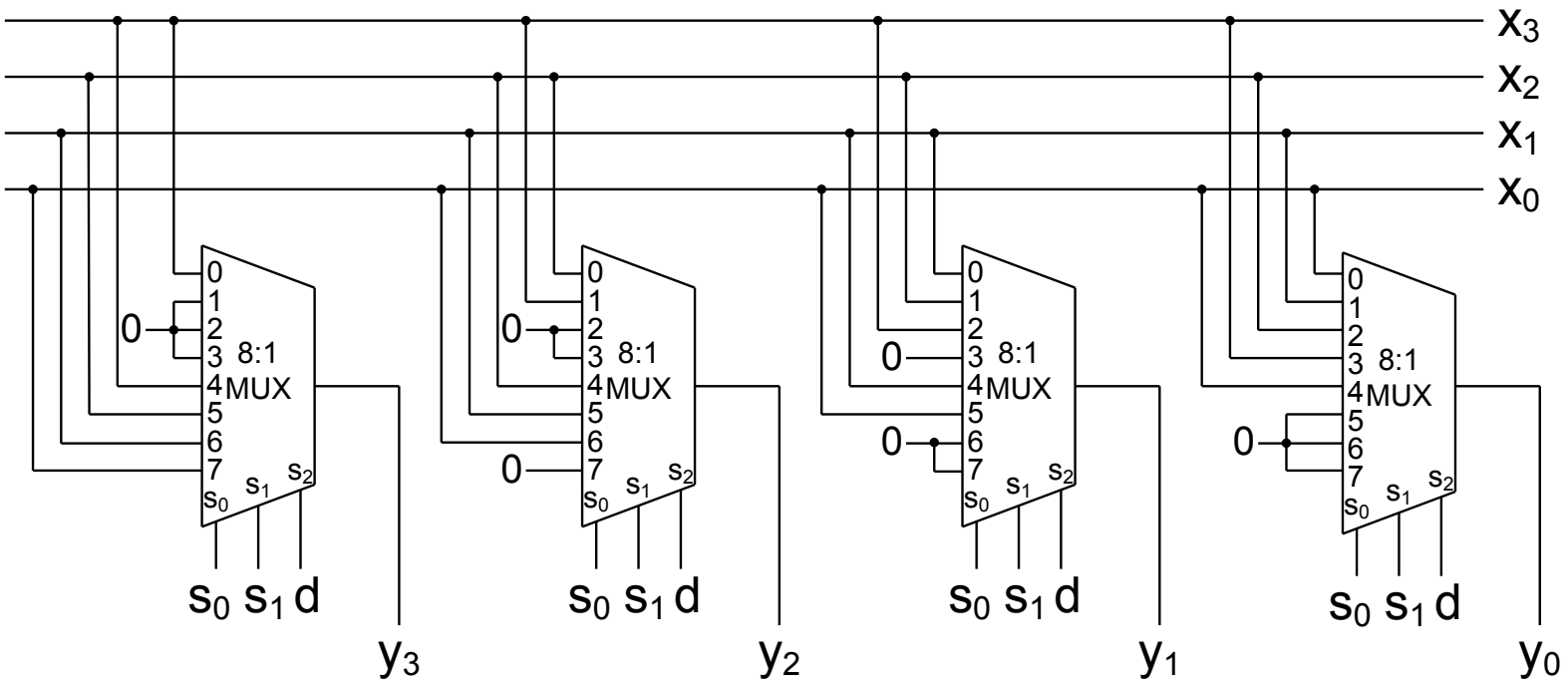
Beispiel: 4 Bit Barrel-Shifter



d	s_1	s_0	y_3	y_2	y_1	y_0	
0	0	0	x_3	x_2	x_1	x_0	rechts schieben
0	0	1	0	x_3	x_2	x_1	
0	1	0	0	0	x_3	x_2	
0	1	1	0	0	0	x_3	
1	0	0	x_3	x_2	x_1	x_0	links schieben
1	0	1	x_2	x_1	x_0	0	
1	1	0	x_1	x_0	0	0	
1	1	1	x_0	0	0	0	

Der d-Eingang wählt die Richtung (direction) aus, in die geschoben werden soll. Die beiden anderen Steuerleitungen geben an, um wieviel Stufen geschoben wird. Frei werdende Bitstellen werden mit 0 aufgefüllt.

Barrel-Shifter



Erweiterter Barrel-Shifter

Der Barrel-Shifter wird um eine Steuerleitung zum erweiterten Barrel-Shifter ergänzt, so dass er auch n-stellige Datenwörter um eine oder mehrere Stellen nach links oder rechts rotieren kann. Beim Rotieren werden Bits, die rechts rauslaufen, links wieder angefügt.

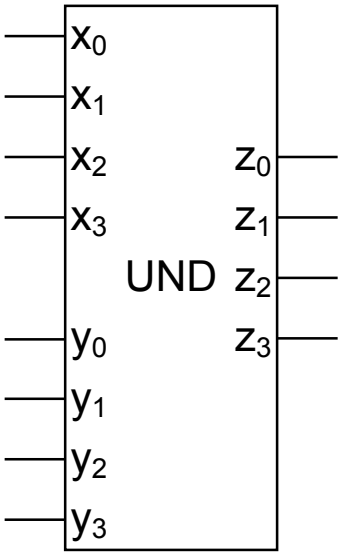
r	d	s_1	s_0	y_3	y_2	y_1	y_0
1	-	0	0	x_3	x_2	x_1	x_0
1	-	0	1	x_0	x_3	x_2	x_1
1	-	1	0	x_1	x_0	x_3	x_2
1	-	1	1	x_2	x_1	x_0	x_3

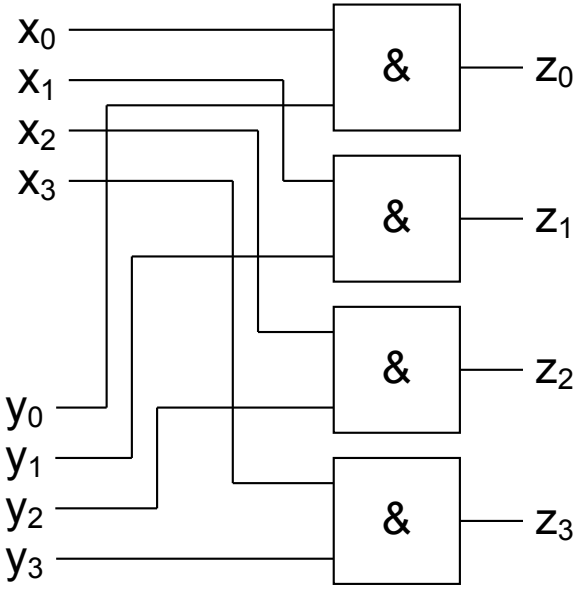
Der neue r-Eingang (rotate) gibt den Modus des Barrel-Shifters an (schieben oder rotieren). Beim Rotieren ist der d-Eingang ohne Funktion.

Logische Verknüpfungen

Neben arithmetischen werden auch oftmals logische Operationen benötigt. Um n-Bit logische Verknüpfungen zu erzeugen, werden n Grundgatter parallel geschaltet.

Beispiel: 4 Bit UND

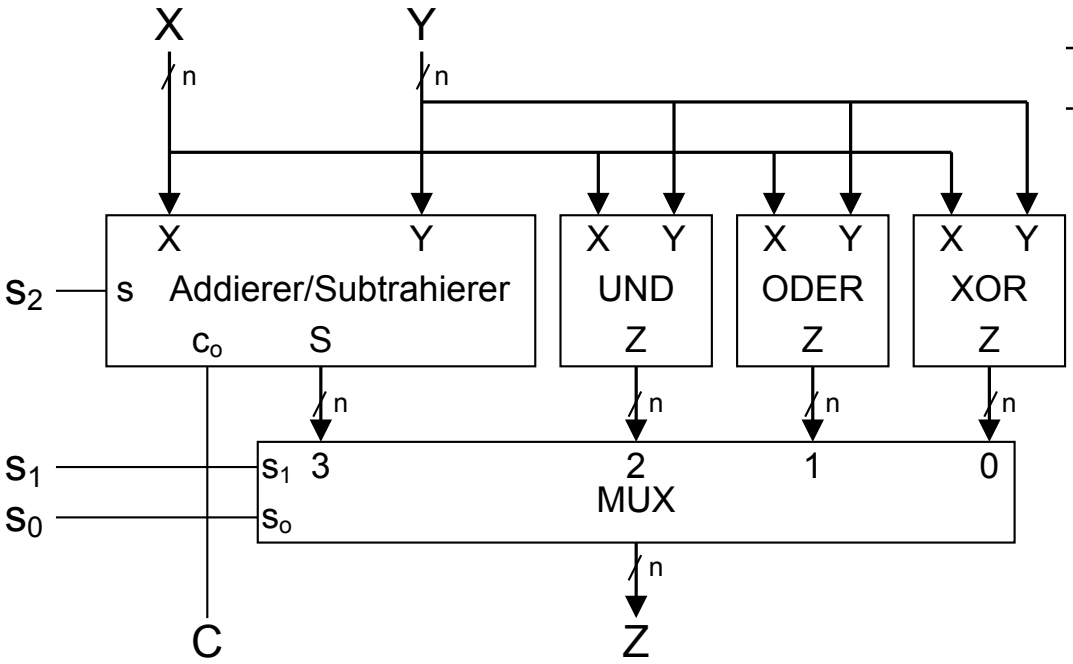


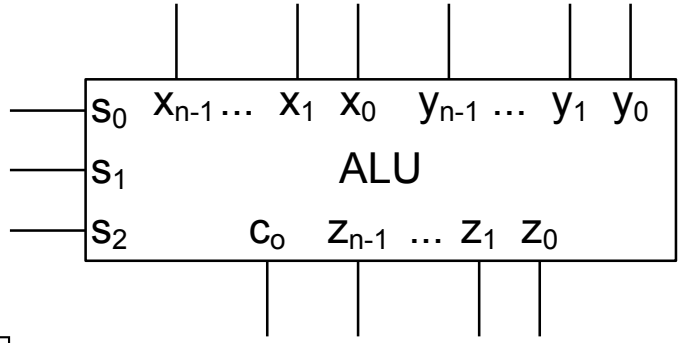


ALU

Die Arithmetische Logische Einheit (engl. arithmetic logic unit, ALU) ist ein wesentlicher Bestandteil der meisten Prozessoren. Mit ihr lassen sich arithmetische (z.B. +, -) und logische (z.B. UND, ODER, XOR) Operationen von zwei Datenwörtern durchführen. Der Funktionsumfang variiert je nach Anwendung.

Beispiel: einfache ALU



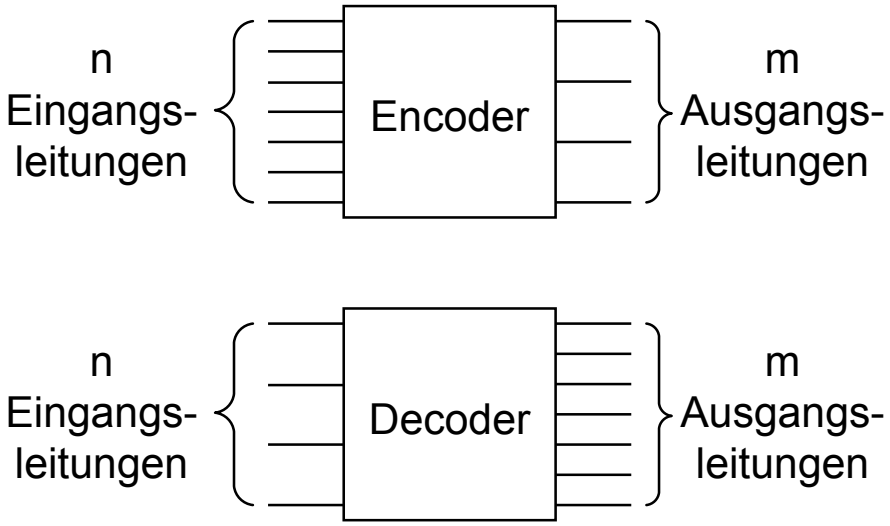


s_2	s_1	s_0	Funktion
–	0	0	Z := X XOR Y
–	0	1	Z := X OR Y
–	1	0	Z := X AND Y
0	1	1	Z := X + Y
1	1	1	Z := X - Y

Kodierer und Dekodierer

Die Wandlung von einem Kode in einen anderen wird Kodierung genannt. Entsprechende Bausteine, welche dies durchführen, werden Kodierer (Encoder) und Dekodierer (Decoder) genannt. Man spricht meist von einem Dekodierer, wenn es mehr Ausgangssignale als Eingangssignale gibt und von einem Kodierer im umgekehrten Fall.

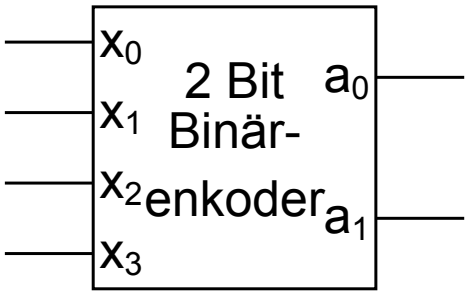
$C_1: \{0,1\}^n \rightarrow C_2: \{0,1\}^m$



Binärenkoder

Ein Kodierer wandelt eine gegebene Information in eine kompaktere Darstellung.
Der Binärenkoder kodiert Informationen von 2^n Eingängen in einen Binärcode mit n-Bit. Nur einer der 2^n Eingänge hat zu einer bestimmten Zeit den Wert 1. Der Ausgang gibt die Nummer des aktiven Eingangs im Binär-Code an.

Beispiel: 2 Bit Binärenkoder



one-hot binär

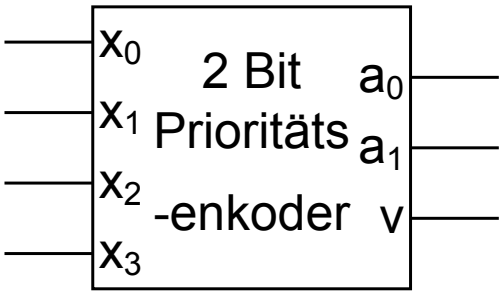
x_3	x_2	x_1	x_0	a_1	a_0
1	0	0	0		
0	1	0	0		
0	0	1	0		
0	0	0	1		

$$a_1 = x_2 + x_3$$
$$a_0 = x_1 + x_3$$

Prioritätsenkoder

Den 2^n Eingängen des Prioritätsenkoders werden Prioritäten zugeordnet. Unter allen gesetzten Eingängen (1), wird der mit der höchsten Priorität ausgewählt und seine Nummer (Adresse) erscheint am Ausgang. Ein Statussignal **v** (valid) zeigt einen aktiven Eingang an.

Bespiel: 2 Bit Prioritätsenkoder; x_3 höchste, x_0 niedrigste Priorität

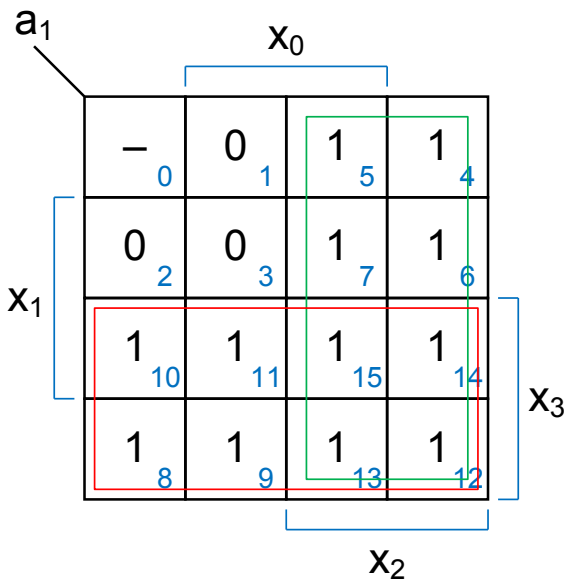


x_3	x_2	x_1	x_0	a_1	a_0	v
1	–	–	–			
0	1	–	–			
0	0	1	–			
0	0	0	1			
0	0	0	0			

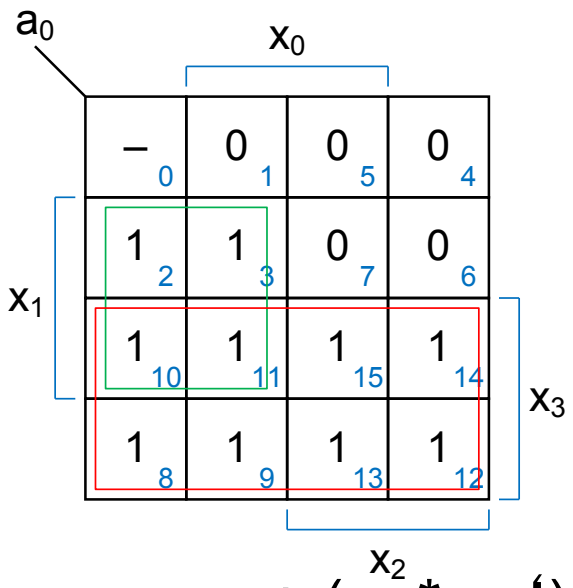
$$v = x_0 + x_1 + x_2 + x_3$$

Prioritätsenkoder

Minterm	x_3	x_2	x_1	x_0	a_1	a_0	v
8-15	1	–	–	–	1	1	1
4-7	0	1	–	–	1	0	1
2,3	0	0	1	–	0	1	1
1	0	0	0	1	0	0	1
0	0	0	0	0	–	–	0



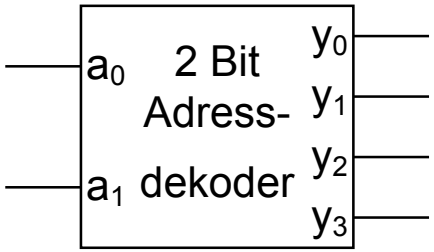
$$a_1 = x_2 + x_3$$



$$a_0 = x_3 + (x_1 * x_2')$$

Adressdekoder

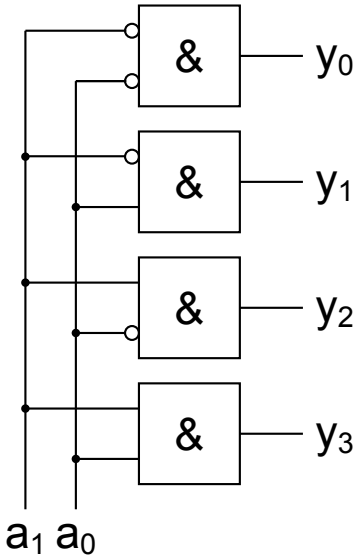
Bei der Ansteuerung von Speichern werden Adressdekoder eingesetzt. Ein n-Bit Eingangswort (Adresse) wählt eine von 2^n Adressleitungen aus.



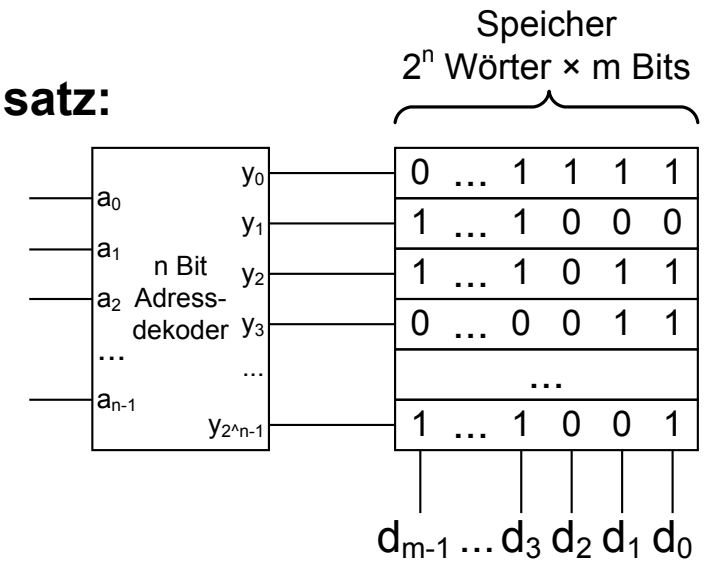
a_1	a_0	y_0	y_1	y_2	y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

binär

one-hot



Einsatz:



Kontrollfragen

- Wie funktioniert die Subtraktion mittels Addition des Komplements bei Hexadezimalzahlen? Rechnen Sie z.B. A9 – 6C mit Komplementdarstellung.
- Was ist der Unterschied zwischen einem Halbaddierer, einem Volladdierer und einem Carry-Ripple-Addierer?
- Recherchieren Sie eine andere Addiererimplementierung (z.B. Carry-Lookahead-Addierer) und vergleichen Sie diese mit dem Carry-Ripple-Addierer.
- Wie können die arithmetischen Operationen Subtraktion und Vergleich (<, =, >) allesamt auf die Addition zurückgeführt werden?
- Erläutern Sie die Funktionsweise des aus 1-Bit-Komparatoren aufgebauten n-Bit-Komparators.
- Welche Rolle spielen Multiplexer in einem Datenpfad?
- Auf welche verschiedenen Arten können größere Multiplexer aufgebaut werden? Welche unterschiedlichen Vorteile ergeben sich dadurch?
- Wie hängen Multiplexer mit der Shannon-Entwicklung und BDDs zusammen?
- Wie kann die Implementierung von Schaltfunktionen mit Multiplexern bei der Technologieabbildung auf FPGAs helfen?
- Was ist der Unterschied zwischen einem Binärenkoder und einem Prioritätsenkoder?
- Was macht ein Adressdekoder?

ANHANG

Multiplexer

VHDL-Modell eines 4-Bit Multiplexers:

```
entity MUX_4_1 is
port( a, b, c, d  : in  bit;
      s           : in  bit_vector(1 downto 0);
      y           : out bit);
end entity MUX_4_1;

architecture behaviour of MUX_4_1 is
begin
      with s select
      y <=  a when "00",
            b when "01",
            c when "10",
            d when "11";
end architecture behaviour;
```

selektive Signalzuweisung
(selective signal assignment)

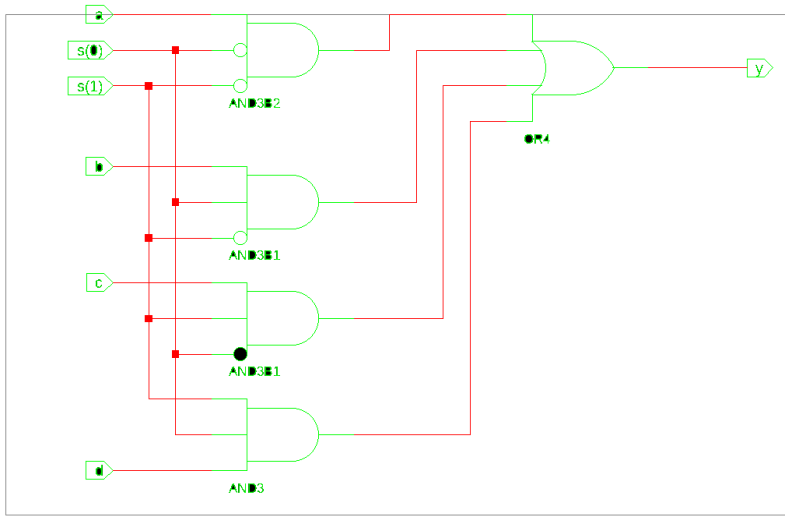
Multiplexer

VHDL-Modell eines 4-Bit Multiplexers:

```
entity MUX_4_1 is
port( a, b, c, d  : in  bit;
      s           : in  bit_vector(1 downto 0);
      y           : out bit);
end entity MUX_4_1;

architecture behaviour of MUX_4_1 is
begin
      with s select
      y <=  a when "00",
            b when "01",
            c when "10",
            d when "11";
end architecture behaviour;
```

Syntheseergebnis:



Prioritätsenkoder

VHDL-Modell eines 4-Bit Prioritätsenkoders:

```
entity priority is
port( x  : in  bit_vector(3 downto 0);
      a  : out bit_vector(1 downto 0);
      v  : out bit);
end entity priority;

architecture behaviour of priority is
begin
      a <=  "11" when x(3) = '1' else
            "10" when x(2) = '1' else
            "01" when x(1) = '1' else
            "00";

      v <=  '0' when x = "0000" else
            '1';
end architecture behaviour;
```

bedingte Signalzuweisung (conditional signal assignment)

Komparator

VHDL-Modell eines 4-Bit Komparators:

```
entity comparator is
port( a, b  : in  bit_vector(3 downto 0);
      g     : out bit;
      e     : out bit;
      l     : out bit);
end entity comparator;

architecture behaviour of comparator is
begin
      g <=  '1' when a > b else '0';
      e <=  '1' when a = b else '0';
      l <=  '1' when a < b else '0';
end architecture behaviour;
```

Volladdierer

VHDL-Modell eines Volladdierers:

```
entity full_adder is
port( a, b, ci   : in  bit;
      s, co      : out bit);
end entity full_adder;

architecture dataflow of full_adder is
begin
      s   <= a xor b xor ci;
      co  <= (a and b) or (a and ci) or (b and ci);
end architecture dataflow;
```

Ripple-Carry-Addierer

VHDL-Modell eines 4-Bit Ripple-Carry Addierers:

```
entity ripple_carry_adder is
port( a, b  : in  bit_vector(3 downto 0);
      ci    : in  bit;
      s     : out bit_vector(3 downto 0);
      co    : out bit);
end entity ripple_carry_adder;
```

Ripple-Carry-Addierer

```
architecture structure of ripple_carry_adder is
component full_adder
port(a, b, ci   : in   bit;
     s, co      : out  bit);
end component full_adder;
signal c : bit_vector(4 downto 0);
begin
           c(0) <= ci;

add_gen:   for i in 0 to 3 generate
           FA:  component full_adder
                port map(a  => a(i),
                         b  => b(i),
                         ci => c(i),
                         s  => s(i),
                         co => c(i+1));
           end generate add_gen;
           co <= c(4);
end architecture structure;
```

Addierer Verhaltensmodell

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity adder is
port(  a, b   : in   std_logic_vector(3 downto 0);
       ci     : in   std_logic;
       s      : out  std_logic_vector(3 downto 0);
       co     : out  std_logic);
end entity adder;

architecture behaviour of adder is
signal sum : std_logic_vector(4 downto 0);
begin
       sum <= ('0' & a) + ('0' & b) + ci;
       s <= sum(3 downto 0);
       co <= sum(4);
end architecture behaviour;
```