4 kombinatorische Schaltungen

Kombinatorik: die Ausgangsvariable ist **nur** eine Funktion der Eingangsvariablen:
$y = f(x_0, x_1, \ldots. x_n)$
- realisierbar durch Verknüpfung mit Grundgattern
- keine Rückführung des Ausgangs auf den Eingang

Synthese: ST (Schaltbelegungstabelle → SF (Schaltfunktion) → LP (Logikplan)

vollständig definierte Funktionen:

ST: Anordnung der n unabhängigen Variablen x_i in **allen** 2^n möglichen Kombinationen

SF: Aufstellen einer Normalform, z.B. KDN (kanonisch disjunktiven Normalform)
Minimierung der SF (z.B. Karnaugh/Veitch, Quine/McCluscy)

LP: Umsetzung in die verfügbaren Gattertechniken (z.B. NAND, NOR, Inverter)

unvollständig definierte Funktionen:

ST: Anordnung der n unabhängigen Variablen x_i in **p** ($0 < p < 2^n$) Kombinationen, für die Ausgangsvariable definierte Werte annehmen.
Für alle anderen **q** ($q = 2^n - p$) Kombinationen sind die Werte der Ausgangsvariablen beliebig, man bezeichnet sie als redundant.

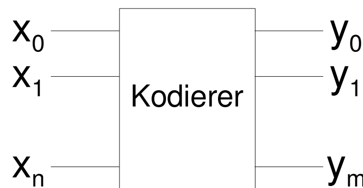
SF: Aufstellen einer Normalform ist mehrdeutig (2^q mögliche Funktionen).
Minimierung nach Karnaugh/Veitch:
1. Die q redundanten Felder werden leer gelassen oder häufig mit einem × oder **-** gekennzeichnet („don´t care“).
2. Minimierung, wobei in die leeren Felder 0 oder 1 so eingesetzt werden, daß sich möglichst große Blöcke bilden.

LP: Umsetzung in die verfügbaren Gattertechniken (z.B. NAND, NOR, Inverter)

4.1 Logikschaltungen

Logikschaltungen sind kombinatorische Schaltungen, die logische Verknüpfungen der Eingangsvariablen realisieren.
Beispiele für Logikschaltungen sind: Kodierer (Dekodierer), Multiplexer (Demultiplexer)

Kodierschaltungen:
Kodierer realisieren eine eindeutige Zuordnung eines Zahlencodes zu einer Kombinationen von Eingangsvariablen (Überführung in ein Zahlensystem)



Beispiel: 3 Eingangs- zu 2-bit-Dekoder

x_1	x_2	x_3	y_1	y_0
0	0	0	0	0
1	0	0	0	1
0	1	0	1	0
0	0	1	1	1
sonst			-	-

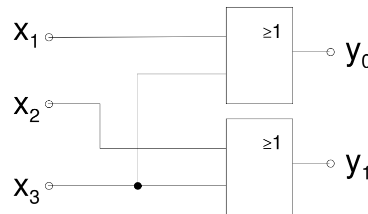
(Beispiel einer unvollständig definierte Funktion:
nur 4 Zeilen von 8 möglichen sind definiert)

KV-Diagramm für y0 (n=3):

x_3 \ x_2x_1	00	01	11	10
0	0	1	-	0
1	1	-	-	-

$y_0 = x_1 \vee x_3$

bzw. $y_1 = x_2 \vee x_3$



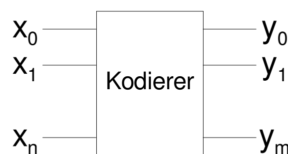
HDL (Codefragment): In Hardwarebeschreibungssprachen gibt es eine case-Anweisung:

```
case X is
      when “000“  =>   Y <= “00“;
      when “100“  =>   Y <= “01“;
      when “010“  =>   Y <= “10“;
      when “001“  =>   Y <= “11“;
      when others =>   Y <= “--“;
end case;
```

X ist ein VECTOR aus 3 Elementen: (X1,X2,X3)
Y ist ein VECTOR aus 2 Elementen: (Y1,Y0)
“- -“ bedeutet nicht definiert (dont care)
und beschreibt die unvollständige Funktion.
*Die **others** Zeile darf nicht vergessen werden!*

Die markierten Textfenster geben einen Hinweis auf eine Hardwarebeschreibungssprache VHDL. Sie dienen der besseren Anschauung und geben einen Vorblick auf den modernen Hardwareentwurf, sind jedoch keine Prüfungsleistung. In der Literatur zur Digitaltechnik und zu Rechnerarchitekturen ist eine solche Beschreibung in der Regel nicht enthalten, es gibt jedoch spezielle VHDL Literatur, z.B. Molitor, Ritter: VHDL - Eine Einführung, Pearson Verlag 2004.

Dekodierschaltungen:
Dekodierer generieren aus dem Zahlencode eine Ausgangskombination.



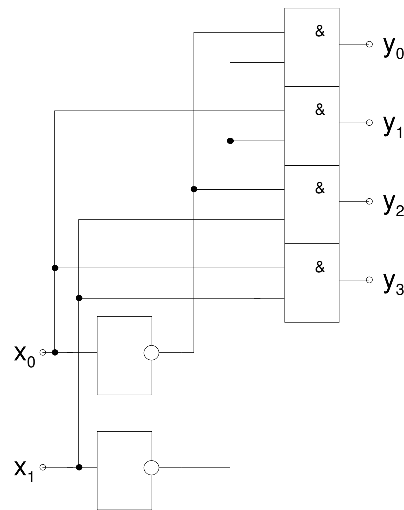
Anwendung: Adressierung von genau einer Speicherzelle in Abhängigkeit von der Eingangsbelegung (Adresse)

Beispiel: 2-bit zu 1-aus-4-Dekoder

x_1	x_0	y_3	y_2	y_1	y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

4 vollständig definierte Funktionen y_3 .. y_0:

$y_3 = x_1 \wedge x_0$ $y_2 = x_1 \wedge \overline{x_0}$ $y_1 = \overline{x_1} \wedge x_0$ $y_0 = \overline{x_1} \wedge \overline{x_0}$



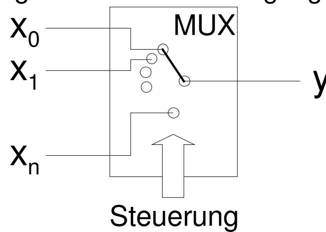
HDL (Codefragment):

```
case X is
        when "00"   =>   Y <= "0001";
        when "01"   =>   Y <= "0010";
        when "10"   =>   Y <= "0100";
        when "11"   =>   Y <= "1000";
        when others =>   Y <= "----";
end case;
```

X ist ein VECTOR aus 2 Elementen: (X1,X0)
Y ist ein VECTOR aus 4 Elementen: (Y3,Y2,Y1,Y0)
"- - - -" ist nicht definiert (dont care)
auch für vollständig definierte Funktionen, also auch wenn `others` *logisch nicht auftritt, sollte diese Zeile enthalten sein!*

Multiplexerschaltungen:

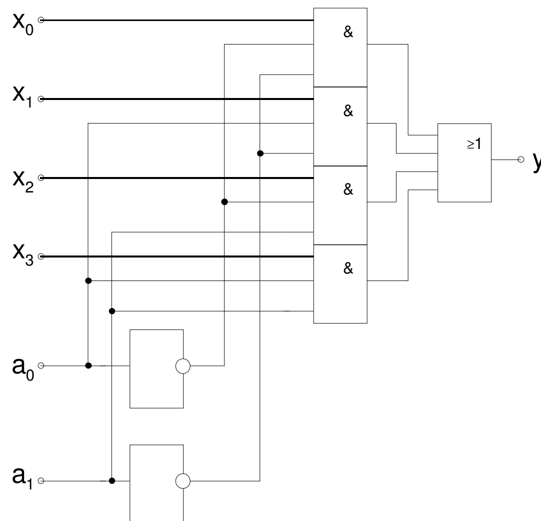
Multiplexer realisieren eine Zuordnung von einer von n Eingangsleitungen auf einen Ausgang.



Anwendung: Auswahl einer von mehreren digitalen Signalquellen, Mehrfachausnutzung einer Übertragungsleitung y durch zeitserielle Übertragung mehrerer Eingangsleitungen.

$y = x_0\, \overline{a_1}\, \overline{a_0} \vee x_1\, \overline{a_1}\, a_0 \vee x_2\, a_1\, \overline{a_0} \vee x_3\, a_1\, a_0$

LP:



HDL:

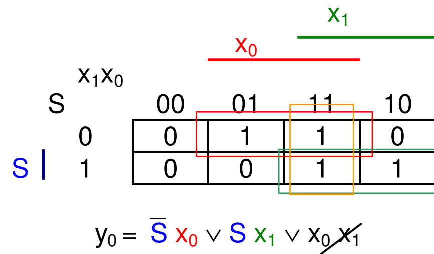
```
case A is
        when "00" =>    Y <= X(0);
        when "01" =>    Y <= X(1);
        when "10" =>    Y <= X(2);
        when "11" =>    Y <= X(3);
        when others =>  Y <= '-';
end case;
```

So werden die Signale deklariert: *-- case insensitive!*
```
Signal Y: STD_LOGIC;                          -- Kommentar
Signal X: STD_LOGIC_VECTOR(0 to 3);   -- X(0), X(1) .. X(3)
Signal A: STD_LOGIC_VECTOR(1 downto 0);-- A(1), A(0)
```
STD_LOGIC ist ein Datentyp. der neben '0 ', '1 ' auch don't care,'-', Tristate 'Z' und andere Werte annehmen kann: 9-wertige Logik STD_LOGIC (U, 0, 1, X, Z, L, H, W, -)

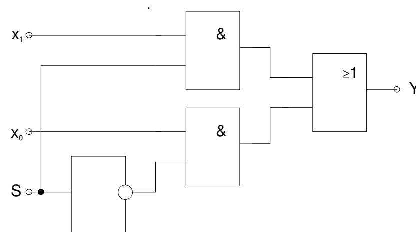
Häufig angewendete Schaltung: **1-aus-2 Multiplexer** (Umschalter):
Funktion: vollständig definiert: $y = x_1$ bei S=1, sonst $y = x_0$
ST:

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

KV:



LP:



VHDL (vollständig):

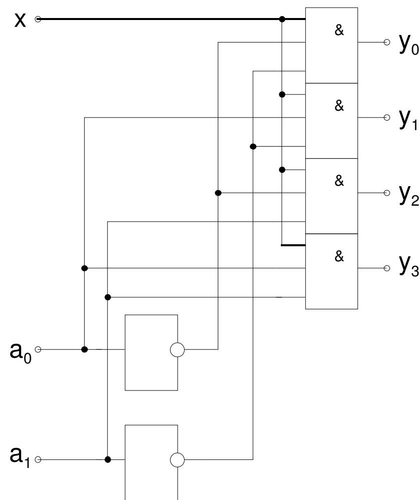
```
LIBRARY IEEE;  -- hier werden die von IEEE
USE IEEE.STD_LOGIC_1164.ALL; -- hier werden
        -- STD_LOGIC Datentypen deklariert

ENTITY mux2 is
port
(Signal Y: out STD_LOGIC;
 Signal X: in STD_LOGIC_VECTOR(1 downto 0);
 Signal S: in STD_LOGIC
);
end mux2;

ARCHITECTURE mux of mux2 is
begin
Y <= X(1) when S = '1' else X(0);
end mux;
```

Das ist schon ein vollständiger VHDL Code (VHSIC Hardware Description Language), der das Verhalten (ARCHITECTURE) eines mux2 Objektes beschreibt.

1. *Das Objekt mux2 muss in einer **ENTITY** deklariert werden, X,Y und S haben eine Richtung (in, out)*
2. *Das Verhalten mux des Objektes mux2 wird in einer **ARCHITECTURE** beschrieben. Hier:*
   `Y <= X(1) when S = '1' else X(0);`

Die Groß-Klein-Schreibung ist bei VHDL unwichtig.

Demultiplexer realisieren die Zuordnung einer Eingangsleitung auf eine von n Ausgangsleitungen:
Beispiel: 4-auf-1-Demultiplexer: $y_0 = x\ \overline{a_1}\ \overline{a_0}$ $y_1 = x\ a_1\ \overline{a_0}$ $y_2 = x\ \overline{a_1}\ a_0$ $y_3 = x\ a_1\ a_0$



LP:

Anwendung: Umschaltung einer digitalen Signalquelle
Mehrfachausnutzung einer Übertragungsleitung (Empfängerseitig)

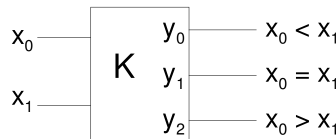
HDL (Codefragment mit Kommentaren („- - Kommentar“):

```
Y <= “0000”;                          -- der default Wert 0000
case A is
        when “00“ =>    Y(0) <= X; -- … wird an genau einer Stelle
        when “01“ =>    Y(1) <= X; -- überschrieben!
        when “10“ =>    Y(2) <= X; -- sonst bleibt der default Wert
        when “11“ =>    Y(3) <= X; -- erhalten
                                   -- daher darf others hier entfallen!
end case;
```

4.2 Arithmetikschaltungen

Arithmetikschaltungen sind kombinatorische Schaltungen, die arithmetische Verknüpfungen der Eingangsvariablen realisieren.
Beispiele für Arithmetikschaltungen sind Komparator, Addierer, Subtrahierer, Multiplizierer, Dividierer, Barrel-Shifter

Komparator:
digitale Vergleichsschaltung



Einzelbitvergleicher

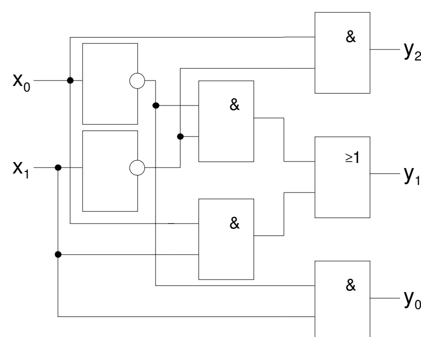
ST:

x_0	x_1	$x_0 < x_1$ y_0	$x_0 = x_1$ y_1	$x_0 > x_1$ y_2
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

SF:

$y_0 = \bar{x}_0\, x_1$
$y_1 = \bar{x}_0\, \bar{x}_1 \vee x_0\, x_1$
$y_2 = x_0\, \bar{x}_1$

LP:

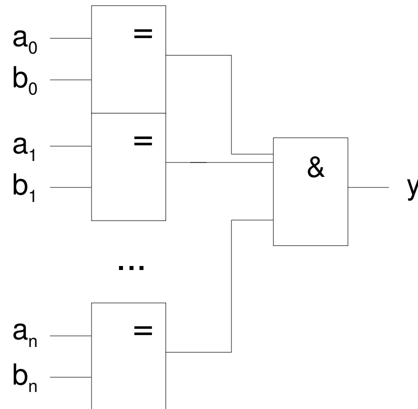


Wortvergleicher

Vergleich zweier n-stelliger Dualzahlen auf Gleichheit

SF: $y = (a_0 b_0 \vee \overline{a_0}\ \overline{b_0})\ (a_1 b_1 \vee \overline{a_1}\ \overline{b_1})\ \ldots\ (a_{n-1} b_{n-1} \vee \overline{a_{n-1}}\ \overline{b_{n-1}})$

LP:



VHDL (vollständig):

```
LIBRARY IEEE;                                    -- hier werden die von IEEE standardisierten
USE IEEE.STD_LOGIC_1164.ALL;                     -- STD_LOGIC Datentypen deklariert
                                                 -- muss immer vor ENTITY stehen !
ENTITY equal is                                  -- Das Symbol
generic (N: INTEGER :=4);                        -- beliebige Bitbreite, defaultwert 4
port ( Signal A: in STD_LOGIC_VECTOR(N-1 downto 0);
       Signal B: in STD_LOGIC_VECTOR(N-1 downto 0);
       Signal Y: out STD_LOGIC) ;                -- kein weiteres ; in der Klammer
end equal;

ARCHITECTURE arch_equal of equal is              -- Die Funktion
begin
       Y <= ’1’ when (A = B) else ’0’;           -- alle Bits müssen übereinstimmen
end arch_equal;                                  -- der ”=” Operator ist für den Typ
                                                 -- STD_LOGIC_VECTOR als lexikalischer
                                                 -- Vergleich erklärt
```

Vergleich zweier n-stelliger Dualzahlen auf Größer/Kleiner

Einzelstufe (1 Bit):

ST:

z_1 $(x_1 < x_0)_{in}$	z_0 $(x_1 > x_0)_{in}$	x_1	x_0		y_1 $(x_1 < x_0)_{out}$	y_0 $(x_1 > x_0)_{out}$
0	0	0	0		0	0
0	0	0	1		1	0
0	0	1	0		0	1
0	0	1	1		0	0
0	1	-	-		0	1
1	0	-	-		1	0
1	1	-	-		-	-

SF:

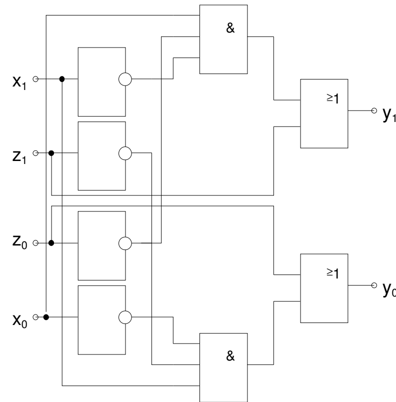
$y_1 = z_1 \vee \overline{z_0}\ \overline{x_1}\ x_0$ $y_0 = z_0 \vee \overline{z_1}\ x_1\ \overline{x_0}$

Für y_1 $\overline{z_0}\ \overline{x_1}\ x_0$

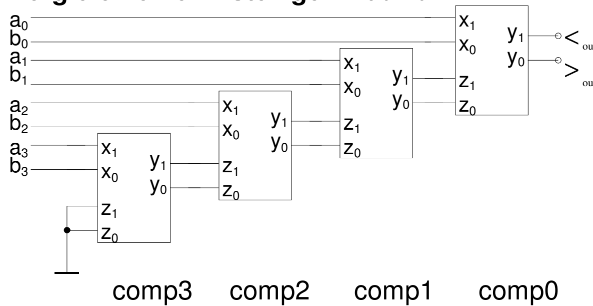
$z_1 z_0$ \ $x_1 x_0$	00	01	11	10
00	0	1	0	0
01	0	0	0	0
11	-	-	-	-
10	1	1	1	1

z_1

LP:



Vergleich einer 4-stelligen Dualzahl



VHDL Bit- und Wortvergleicher

```
LIBRARY IEEE; USE IEEE.STD_LOGIC_1164.ALL;
ENTITY bit_comp is                             -- Die Schnittstelle Bit-Vergleicher
port ( Signal X1,X0: in STD_LOGIC;
       Signal Z0,Z1: in STD_LOGIC;              -- Y0:X1>X0
       Signal Y0,Y1: out STD_LOGIC);            -- Y1:X1<X0
end bit_comp;

ARCHITECTURE arch_bit_comp of bit_comp is      -- das Verhalten
Begin
Y0 <= Z0 OR (NOT Z1 AND X1 AND NOT X0);        -- y0 = z0 ∨ z̄1 x1 x̄0
Y1 <= Z1 OR (NOT Z0 AND NOT X1 AND X0);        -- y1 = z1 ∨ z̄0 x̄1 x0
```

```
LIBRARY IEEE; USE IEEE.STD_LOGIC_1164.ALL;
ENTITY compare  is                             -- Die Schnittstelle Wort-Vergleicher
port ( Signal A: in STD_LOGIC_VECTOR(3 downto 0);   -- Bitbreite 4
       Signal B: in STD_LOGIC_VECTOR(3 downto 0);
       Signal Y1, Y0: out STD_LOGIC) ;              -- Y1:A<B  Y0:A>B
end compare;

ARCHITECTURE arch_compare of compare is
component bit_comp                             -- Die Teil-Komponente (=ENTITY bit_comp)
port ( Signal X1,X0: in STD_LOGIC;
       Signal Z0,Z1: in STD_LOGIC;              -- Y0:X1>X0
       Signal Y0,Y1: out STD_LOGIC);            -- Y1:X1<X0
end component;
Signal Z0: STD_LOGIC _VECTOR (3 downto 0);     -- lokale Signale ohne in/out Richtung
Signal Z1: STD_LOGIC _VECTOR (3 downto 0);     -- zur Kopplung der Komponenten
begin                                          -- Strukturbeschreibung
--                 port (A    ,B   ,Z0    ,Z1    ,Y0    , Y1) -- positionale Zuordnung
comp0: bit_comp port map(A(0),B(0),Z0(0),Z1(0),Y0   , Y1);      -- 4 Stufen
comp1: bit_comp port map(A(1),B(1),Z0(1),Z1(1),Z0(0),Z1(0));
comp2: bit_comp port map(A(2),B(2),Z0(2),Z1(2),Z0(1),Z1(1));
comp3: bit_comp port map(A(3),B(3),Z0(3),Z1(3),Z0(2),Z1(2));
Z0(3) <= '0';                                           -- Anfangswerte
Z1(3) <= '0';
end arch_compare;
```

VHDL: es geht noch allgemeiner zu beschreiben: der N-BIT Vergleicher

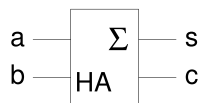
```
LIBRARY IEEE; USE IEEE.STD_LOGIC_1164.all;
entity compare  is                                    -- Eine Schnittstelle
generic (N: INTEGER:=4);                              -- für beliebige Bitbreite N
port (Signal A: in STD_LOGIC_VECTOR(N-1 downto 0);
Signal B: in STD_LOGIC_VECTOR(N-1 downto 0);
Signal Y1, Y0: out STD_LOGIC) ;                       -- Y1:A<B  Y0:A>B
end compare;
architecture arch_compare of compare is               -- Die Teil-Komponente

component bit_comp
  port (Signal X1,X0: in STD_LOGIC;
  Signal Z0,Z1: in STD_LOGIC;                         -- Y0:X1>X0
  Signal Y0,Y1: out STD_LOGIC);                       -- Y1:X1<X0
end component;
Signal Z0: STD_LOGIC _VECTOR (N-1 downto 0);          -- lokale Signale
Signal Z1: STD_LOGIC _VECTOR (N-1 downto 0);          -- zur Kopplung
begin
comp0: bit_comp port map(A(0),B(0),Z0(0),Z1(0),Y0,Y1);
generate_bits:                                        -- label
  for I in 1 to N-1 generate                          -- N-fache Wiederholung
  compx: bit_comp port map(A(I),B(I),Z0(I),Z1(I),Z0(I-1),Z1(I-1));
end generate;
Z0(N-1) <= '0';
Z1(N-1) <= '0';
end arch_compare;
```

- *mit VHDL kann man auch Strukturen (also Komponenten und Verbindungs-Signale) beschreiben.*
- *Die Teil-Komponenten werden „instanziiert", jede bekommt ihr Label (comp0, comp1 ..)*
- *Die Übergabe der Signale erfolgt exakt nach Position der **port** Liste der **entity** und der **component**.*
- *Eine namentliche Zuordnung benötigt eine andere Syntax: **port map**(X1 => A(I), X0=>B(I) , ...);*
- ***Entity** bit_comp und **component** bit_comp müssen exakt übereinstimmen!*
- *Die Laufvariable (hier: I) bei **for** ... **generate** wird implizit deklariert,*
- *Die Komponenten werden entsprechend der **generate** Durchläufe von I vervielfacht.*

Addierer:

1. Einzelbitaddierer

Halbadder



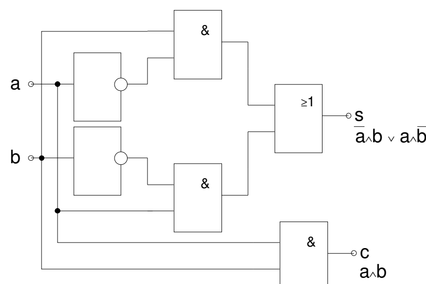
ST:

a	b	Über-trag c	Summe a+b s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

SF: $s = (a\,\overline{b}) \vee (\overline{a}\,b) = a \oplus b$ (⊕: Exklusiv-Oder) (2-1)

$c = a\,b$ (AND)

LP:

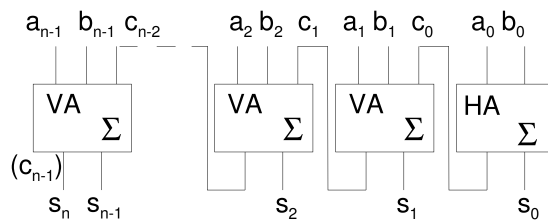


2. Wortaddierer

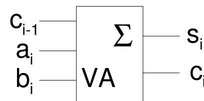
Funktion: $S = A + B$ mit $A = (a_{n-1}, a_{n-2}, \ldots a_1, a_0)$
$B = (b_{n-1}, b_{n-2}, \ldots b_1, b_0)$
$S = (s_n, s_{n-1}, s_{n-2}, \ldots s_1, s_0)$
Die Addition der einzelnen Binärstellen erfolgt mit je einem Volladder.
Das Ergebnis ist um eine Dualstelle größer als die Operanden.

mehrstellige Addition im dualen Zahlensystem:

	a_{n-1}	...	a_2	a_1	a_0	Summand A
+	b_{n-1}	...	b_2	b_1	b_0	Summand B
+ c_{n-1}	c_{n-2}	...	c_1	c_0		
s_n	s_{n-1}	...	s_2	s_1	s_0	Summe S



Volladder:



ST:

c_{i-1}	a_i	b_i	c_i	s_i
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

SF:
$$s_i = \overline{c_{i-1}}\,\overline{a_i}\,b_i \vee \overline{c_{i-1}}\,a_i\,\overline{b_i} \vee c_{i-1}\,\overline{a_i}\,\overline{b_i} \vee c_{i-1}\,a_i\,b_i \qquad (2\text{-}3)$$
$$c_i = \overline{c_{i-1}}\,a_i\,b_i \vee c_{i-1}\,\overline{a_i}\,b_i \vee c_{i-1}\,a_i\,\overline{b_i} \vee c_{i-1}\,a_i\,b_i$$

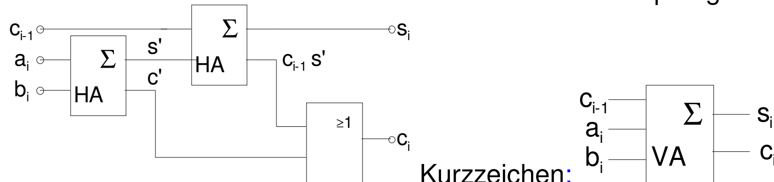
vereinfachte SF unter Ausnutzung der Exklusiv-Oder-Beziehung $a \oplus b = \overline{a}\,b \vee a\,\overline{b}$

$s_i = (a_i \oplus b_i) \oplus c_{i-1}$ (2 XOR-Gatter) (2-4)
$c_i = a_i\,b_i \vee c_{i-1}\,(a_i \oplus b_i)$ (2 AND, 1 OR)

mit der Substitution $s' = a_i \oplus b_i$ und $c' = a_i\,b_i$ (erste Halbadder-Funktion s. grauer Tabellenteil)

$s_i = s' \oplus c_{i-1}$ (2-5)
$c_i = c' \vee c_{i-1}\,s'$

s_i und c_i enthalten eine zweite Halbadder-Funktion: Summe $s' \oplus c_{i-1}$ und Übertrag: $c_{i-1}\,s'$
→ Aufbau eines Volladders aus zwei Halbaddern und einer OR-Verknüpfung:



Kurzzeichen: c_{i-1}, a_i, b_i — Σ VA — s_i, c_i

VHDL:

Versuchen Sie den Code nach dem Muster des Bit-Vergleichers bit_comp selbst zu vervollständigen!

```
LIBRARY IEEE; USE IEEE.STD_LOGIC_1164.ALL;
ENTITY bit_volladder is port (…);  end bit_volladder;
ARCHITECTURE arch_bit_volladder of bit_volladder is (…) begin (…) end arch_bit_volladder;
```

VHDL Addierer (4 bit)

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;         -- STD_LOGIC Deklaration
USE IEEE.std_logic_unsigned.all;     -- Methoden für Addition von Vektoren

ENTITY add4 IS
       PORT(  A,B            : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
              C_1            : IN  STD_LOGIC;
              Q              : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
              C              : OUT STD_LOGIC
              );
END add4 ;

ARCHITECTURE add4_beh OF add4 IS
SIGNAL Q_INT          : STD_LOGIC_VECTOR(4 DOWNTO 0);
BEGIN
Q_INT <= A + B + C_1;          -- Ergebnis ist 1 bit groesser
C <= Q_INT(4);                 -- Übertrag
Q <= Q_INT(3 DOWNTO 0);        -- Summe
END add4_beh;
```

- An dieser Stelle werden die Vorteile der VHDL-Beschreibeung sichtbar: der „+“ Operator aus der Library `std_logic_unsigned` *kann hier auf Vektoren (hier: 4-Bit-Zahlen) angewendet werden.*
- Die Bitbreite auf der linken Seite (Q_INT) muss um ein Bit größer als der größte Operand der rechten Seite (A, B) sein .
- Die Zuordnung einzelner Bits (C) bzw. Vektoren (Q) ist ohne Hardwareressourcen möglich

Multiplikation:

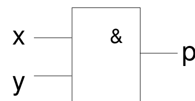
Einzelbit-Multiplizierer: $p = x \cdot y$

ST:

x	y	p
0	0	0
0	1	0
1	0	0
1	1	1

SF: $p = x\,y$

LP:



HDL: P <= X AND Y;

Wortmultiplizierer:

x_n	…	x_2	x_1	x_0	·	y_n	…	y_2	y_1	y_0
					x_0y_n		…	x_0y_2	x_0y_1	x_0y_0
				x_1y_n		…	x_1y_2	$+x_1y_1$	$+x_1y_0$	
			x_2y_n	…	…	$+x_2y_2$	$+x_2y_1$	$+x_2y_0$		
	x_ny_n	…	x_ny_1	x_ny_0						
s_{2n-1}	s_{2n-2}	…	…	…	s_5	s_4	s_3	s_2	s_1	s_0

Das Kommutativgesetz der Addition erlaubt eine beliebige Reihenfolge bei der Bildung der Spaltensumme, beispielsweise können einzelne Halb- oder Volladder kaskadiert werden.

Beispiel: Multiplikation zweier 3-stelliger Dualzahlen (n=3)

	x_2	x_1	x_0	·	y_2	y_1	y_0
					x_0y_2	x_0y_1	x_0y_0
				x_1y_2	$+x_1y_1$	$+x_1y_0$	s_{00}
1. Spaltenübertrag			$+c_{12}$	$+c_{11}$	$+c_{10}$		
1. Zeilensumme				s_{12}	s_{11}	s_{10}	
			$+x_2y_2$	$+x_2y_1$	$+x_2y_0$		
2. Spaltenübertrag		c_{22}	$+c_{21}$	$+c_{20}$			
2. Zeilensumme			s_{22}	s_{21}	s_{20}		
Ergebnis		s_5	s_4	s_3	s_2	s_1	s_0

$s_{00} = x_0y_0$		AND	
$s_{10} = x_0y_1 \otimes x_1y_0$	$c_{10} = x_0y_1 \wedge x_1y_0$	HA	(vgl. 2-1)
$s_{11} = x_0y_2 \otimes x_1y_1 \otimes c_{10}$	$c_{11} = (x_0y_2 \wedge x_1y_1) \vee c_{10}\ (x_0y_2 \otimes x_1y_1)$	VA	(vgl. 2-4)
$s_{12} = x_1y_2 \otimes c_{11}$	$c_{12} = x_1y_2 \wedge c_{11}$	HA	
$s_{20} = s_{11} \otimes x_2y_0$	$c_{20} = s_{11} \wedge x_2y_0$	HA	
$s_{21} = s_{12} \otimes x_2y_1 \otimes c_{20}$	$c_{21} = (s_{12} \wedge x_2y_1) \vee c_{20}\ (s_{12} \otimes x_2y_1)$	VA	
$s_{22} = c_{12} \otimes x_2y_2 \otimes c_{21}$	$c_{22} = (c_{12} \wedge x_2y_2) \vee c_{21}\ (c_{12} \otimes x_2y_2)$	VA	

Ergebnis:

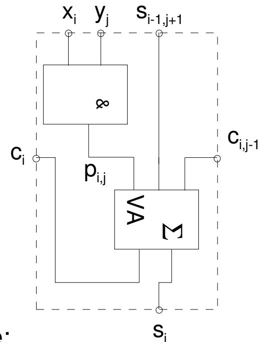
$S_0=s_{00}$ $S_1=s_{10}$ $S_2=s_{20}$ $S_3=s_{21}$ $S_4=s_{22}$ $S_5=c_{22}$

Multiplizierzelle:

SF: $s_{i,j} = s_{i-1,j+1} \otimes p_{i,j} \otimes c_{i,j-1}$ $c_{i,j} = s_{i-1,j+1} \wedge p_{i,j} \vee c_{i,j-1}\ (s_{i-1,j+1} \otimes p_{i,j})$ VA (2-6)

mit $p_{i,j} = x_iy_j$ Produkt AND
$s_{i-1,j+1}$ Spalten-Summe (von oben)
$c_{i,j-1}$ Zeilen-Übertrag (von rechts)

Streng genommen gilt die Beziehung nur unter den Bedingungen:
$i,j = 1 \ldots n-1$ (außer 1. Zeile und rechte Spaltenposition)
$i > 1$ für $j=1$, (außer linke Position der 2. Zeile)
Natürlich ist ein HA auch durch einen VA mit einem fest auf 0 gesetzten Eingang ersetzbar.



LP Multiplizierzelle:

VHDL:

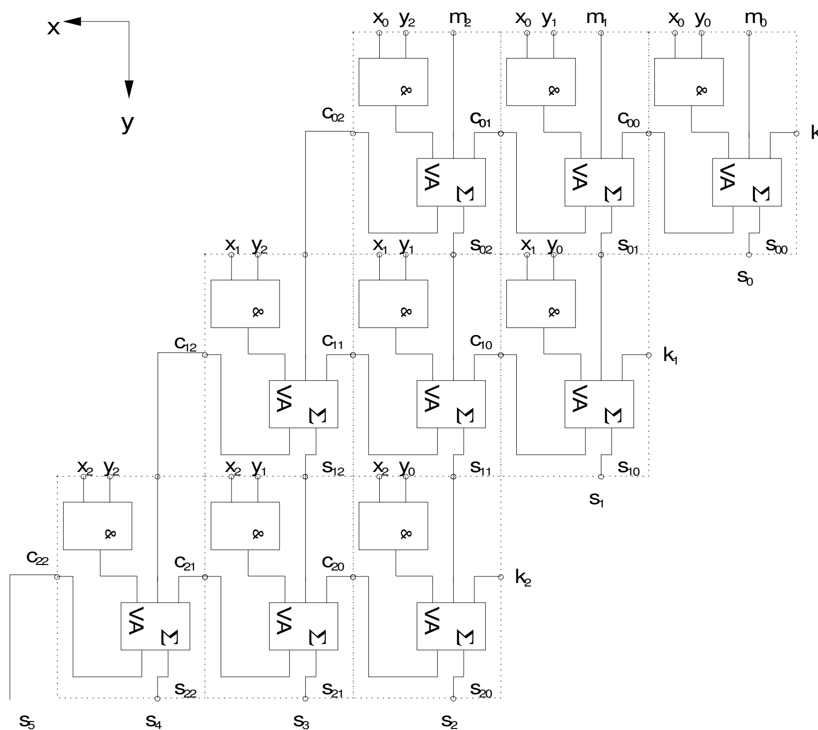
```
LIBRARY IEEE; USE IEEE.STD_LOGIC_1164.ALL;

ENTITY bit_mult is
port ( Signal X,Y:          in STD_LOGIC;
       Signal S_in, C_in:   in STD_LOGIC;
       Signal S_out,C_out:  out STD_LOGIC);
end bit_mult;

ARCHITECTURE arch_bit_comp of bit_mult is
Signal P, S: STD_LOGIC;                          -- lokale Signale
Begin
S       <= S_in XOR P;                           -- HA Summe s´
S_out   <=  S   XOR C_in;                        -- VA Summe si = s´ ⊕ ci-1
P       <=  X   AND Y;                           -- das Teilprodukt
C_out   <= (S_in AND P) OR (C_in AND S );        -- Übertrag ci = c´ ∨ ci-1 s´
end arch_bit_comp;
```

Die Reihenfolge der Notation S_Out<=, C_out <=, S<= und P<= ... spielt keine Rolle!
Immer wenn sich P oder S ändert, wird auch S_out und P_out durch neu berechnet.
Noch allgemeiner: Alle Gatter „rechnen" ständig, arbeiten also parallel bzw. nebenläufig zueinander (concurrent).
Die Codes zwischen ***begin*** *und* ***end*** *sind „concurrent statements".*

Die Reihenfolge der Abarbeitung richtet sich nicht nach der Zeile, sondern nach den Signalen, die sich ändern.
Ein Simulator, der den Code testweise auf einem Rechner ausführen soll, muss ereignisgesteuert (event-driven) arbeiten.
Der VHDL Code modelliert die Hardware. Ein Simulator berechnet mit dem VHDL Modell die erwarteten Ergebnisse.
Mit dieser Methode kann digitale Hardware effizient auf Rechnern entworfen und getestet werden.

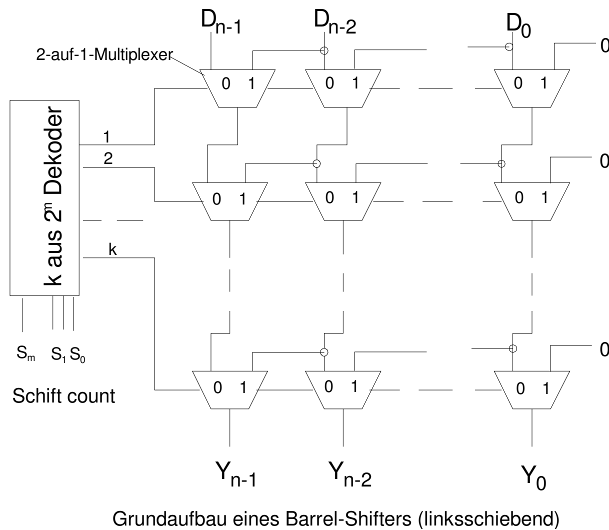
Führt man eine rechte Spalte mit der Variablen $K = \{k_{n-1} \ldots k_0\} = \{c_{n-1,-1} \ldots c_{0,-1}\}$ bzw. eine obere Zeile mit der Variablen $M = \{m_{n-1} \ldots m_0\} = (s_{-1,n-1} \ldots s_{-1,0})$, die als Anfangsbedingung den Wert {0, … 0} besitzen ein, so erhält man eine gleichmäßige Multiplikationsmatrix mit i,j = 0 … n-1.

Führt man zusätzlich die linken Überträge (alle $c_{i,n-1}$) nach außen, ist die Matrix kaskadierbar, z.B. können durch Anordnung von 4 Multiplikationsmatrizen für n-stellige Zahlen Faktoren mit einer Bitbreite von 2n verarbeitet werden. Haben die Faktoren unterschiedliche Bitbreite, muss eine nichtquadratische Multiplikationsmatrix mit zwei verschiedenen Indizes erstellt werden.



	x_2	x_1	x_0	.	y_2	y_1	y_0
					m_2	m_1	m_0
					$+x_0y_2$	$+x_0y_1$	$+x_0y_0$
1. Spaltenübertrag				$+c_{02}$	$+c_{01}$	$+c_{00}$	$+k_0$
1. Zeilensumme					s_{02}	s_{01}	s_{00}
				x_1y_2	$+x_1y_1$	$+x_1y_0$	
2. Spaltenübertrag			$+c_{12}$	$+c_{11}$	$+c_{10}$	$+k_1$	
2. Zeilensumme				s_{12}	s_{11}	s_{10}	
			$+x_2y_2$	$+x_2y_1$	$+x_2y_0$		
3. Spaltenübertrag		c_{22}	$+c_{21}$	$+c_{20}$	$+k_2$		
3. Zeilensumme			s_{22}	s_{21}	s_{20}		
Ergebnis		s_5	s_4	s_3	s_2	s_1	s_0

Anwendung des 2-auf-1-Multiplexers:
Barrel-Shifter:



Grundaufbau eines Barrel-Shifters (linksschiebend)

k aus 2^m - Dekoder

k = 0 .. 7
m = 3

S_2	S_1	S_0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0
0	1	0	1	1	0	0	0	0	0
0	1	1	1	1	1	0	0	0	0
1	0	0	1	1	1	1	0	0	0
1	0	1	1	1	1	1	1	0	0
1	1	0	1	1	1	1	1	1	0
1	1	1	1	1	1	1	1	1	1

VHDL (Codefragment)

```
case S is
        when "000"   =>   Y <= "0000000";
        when "001"   =>   Y <= "1000000";
        when "010"   =>   Y <= "1100000";
        when "011"   =>   Y <= "1110000";
        when "100"   =>   Y <= "1111000";
        when "101"   =>   Y <= "1111100";
        when "110"   =>   Y <= "1111110";
        when "111"   =>   Y <= "1111111";
        when others  =>   Y <= "-------";
end case;
```

4.3 Arithmetisch-logische Einheit

Ansatz: Die Realisierung von arithmetischen Operationen mit Logikgattern führt zu ähnlichen Verarbeitungsstrukturen für arithmetische und logische Befehle
Parallele Zahlwort- oder Logikverarbeitung ist für n bits möglich!
Operanden: $A = (a_{n-1}, \ldots a_1, a_0)$ $B = (b_{n-1}, \ldots b_1, b_0)$
Ergebnis: $Q = (q_{n-1}, \ldots q_1, q_o)$
Übertrag: c_{n-1} (Übertrag bei arithmetischen Operationen)
c_{-1} (evtl. Übertrag einer vorhergehenden Stufe)

Logik: Negation, Konjunktion, Disjunktion
$q_i = \overline{a_i}$ $q_i = a_i\, b_i$ $q_i = a_i \vee b_i$

Ergänzung zu Vollkonjunktionen (ohne Indizierung i = 0 ... n-1):

$q = \bar{a}\, b \vee \bar{a}\, \bar{b}$ $q = a\, b$ $q = a\, b \vee a\, \bar{b} \vee \bar{a}\, b$

führt auf 4 mögliche Grundverknüpfungen, abhängig von Steuergröße $S=(s_3 s_2 s_1 s_0)$

$q = s_3\, a\, b \vee s_2\, a\, \bar{b} \vee s_1\, \bar{a}\, b \vee s_0\, \bar{a}\, \bar{b}$

Es sind 16 verschiedene Funktionen realisierbar (2^{2^2})

Beispiel: S = (1,0,0,0): Konjunktion,
S = (1,1,1,0): Disjunktion,
S = (0,0,1,1): Negation von A

Die logische Verknüpfung erfolgt für alle Bitstellen parallel (n Gleichungen)

$q_i = s_3\, a_i\, b_i \vee s_2\, a_i\, \bar{b}_i \vee s_1\, \bar{a}_i\, b_i \vee s_0\, \bar{a}_i\, \bar{b}_i$ (i = 0 ... n-1)

Arithmetik: Summe:

$q = a \oplus b \oplus c_{i-1}$ ($c_{-1}=0$)

$q = a \equiv b \equiv c_{i-1}$

Differenz (Zweierkomplementdarstellung):

$q = a \oplus \bar{b} \oplus c_{i-1}$ ($c_{-1}=1$)

Additionsgleichung mit Identität (≡):

$c_i = a\, b \vee (a \vee b)\, c_{i-1}$

$q = (a\, b \vee \bar{a}\, \bar{b}) \equiv c_{i-1}$

Subtraktionsgleichung mit Identität:

$c_i = a\, \bar{b} \vee (a \vee \bar{b})\, c_{i-1}$

$q = (a\, \bar{b} \vee \bar{a}\, b) \equiv c_{i-1}$

Die Zusammenfassung aller Grundverknüpfungen ergibt:

$c_i = s_3\, a\, b \vee s_2\, a\, \bar{b} \vee (\overline{s_1\, \bar{a}\, b}\ \ \overline{s_0\, \bar{a}\, \bar{b}})\, c_{i-1}$

$q = (s_3\, a\, b \vee s_2\, a\, \bar{b} \vee s_1\, \bar{a}\, b \vee s_0\, \bar{a}\, \bar{b}) \equiv c_{i-1}$

Beispiel: S = (1,0,0,1): Addition mit Übertrag
S = (0,1,1,0): Zweierkomplement-Subtraktion ($c_{-1}= 1$)

Zusammenfassung und Indizierung i = 0 .. n-1:

mit $g_i = s_3\, a_i\, b_i \vee s_2\, a_i\, \bar{b}_i$ und $p_i = \overline{s_1\, \bar{a}\, b}\ \ \overline{s_0\, \bar{a}\, \bar{b}}$

bzw. $\bar{p}_i = s_1\, \bar{a}\, b \vee s_0\, \bar{a}\, \bar{b}$

ergibt sich $c_i = g_i \vee p_i\, c_{i-1}$ (Übertrag Arithmetik)

und $q_i = g_i \vee \bar{p}_i$ (Ergebnis Logik)

$q_i = (g_i \vee \bar{p}_i) \equiv c_{i-1}$ (Ergebnis Arithmetik)

Der Vergleich von Logik- und Arithmetikgleichung liefert einen gemeinsamen Ausdruck:

$\mathbf{c_i = g_i \vee p_i\, c_{i-1}}$

$\mathbf{q_i = (g_i \vee \bar{p}_i) \equiv (BA \vee c_{i-1})}$ (BA: Betriebsart)

Versuchen Sie den Code nach dem Muster des Bit-Vergleichers bit_comp oder der Multiplizierzelle bit_mult selbst zu vervollständigen!

```
ENTITY bit_alu  is port (…);  end bit_alu;
ARCHITECTURE arch_bit_alu of bit_alu  is
Signal G, P: STD_LOGIC;
begin (…) end arch_bit_alu;
```

Hinweis: Die Identität ist durch negierte XOR Verknüpfung realisierbar.

s_3	s_2	s_1	s_0	Q (Logik)	BA=1	Q (Arithmetik)	BA=0
0	0	0	0	0		$-1 + c_{-1}$	
0	0	0	1	$\overline{A \vee B}$	NOR	$A \vee B + c_{-1}$	
0	0	1	0	$\overline{A} \wedge B$		$A \vee \overline{B} + c_{-1}$	
0	0	1	1	$\overline{A}$	INV A	$A + c_{-1}$	INC
0	1	0	0	$A \wedge \overline{B}$		$(A \vee \overline{B}) -1 + c_{-1}$	
0	1	0	1	$\overline{B}$	INV B	$(A \vee B)+(A \wedge \overline{B})+ c_{-1}$	
0	1	1	0	$(A \wedge \overline{B}) \vee (\overline{A} \wedge B)$	XOR	$A - B - 1 + c_{-1}$	SBC
0	1	1	1	$\overline{A \wedge B}$	NAND	$(A \wedge \overline{B})+A+c_{-1}$	
1	0	0	0	$A \wedge B$	AND	$(A \wedge B) - 1+c_{-1}$	
1	0	0	1	$(A \wedge B) \vee (\overline{A} \wedge \overline{B})$	ÄQU	$A + B + c_{-1}$	ADC
1	0	1	0	B	B	$(A \vee \overline{B})+(A \wedge \overline{B})+c_{-1}$	
1	0	1	1	$\overline{A} \vee B$		$(A \wedge B) + A+ c_{-1}$	
1	1	0	0	A	A	$A -1 + c_{-1}$	DEC
1	1	0	1	$A \vee \overline{B}$		$(A \vee B) + A + c_{-1}$	
1	1	1	0	$A \vee B$	OR	$(A \vee \overline{B}) + A + c_{-1}$	
1	1	1	1	1	SET	$A + A + c_{-1}$	SHL

Anwendung der n-Bit-ALU zum Größenvergleich

S=[0,1,1,0] (Subtrahieren mit Übertrag $A - B - 1 + c_{-1}$)
ergibt in Abhängigkeit von c_{-1} einen Übertrag c_{n-1} zum Vergleich $<$/$\geq$ oder $\leq$/$>$
Spezieller Ausgang für Q=0000 ("Zero") $z=\overline{q_3 \vee q_2 \vee q_1 \vee q_0}$
Beispiel: S=[0,1,1,0] und $c_{-1}=1$ ergibt z=1 für A=B

Relation	c_{n-1}		z	
Eingang	$c_{-1}=0$	$c_{-1}=1$	$c_{-1}=0$	$c_{-1}=1$
A = B	1	0	0	1
A < B	1	1	0	0
A > B	0	0	1	0

Übertrag c_{n-1} in Abhängigkeit von c_{-1} und der Relation zwischen A und B

Generierung eines vorausschauenden Übertrages für die Addition:
Ein Übertrag entsteht in der Stufe bei der Addition (generate) oder durch weiterleiten von der vorherigen (i-1) Stufe, wenn mindestens ein Summand 1 ist (propagate)
$c_n = g_n \vee (p_n c_{n-1})$ (g Generate, p: Propagate)

der Vergleich mit $c_n = a\, b \vee (a \vee b)\, c_{n-1}$ liefert $g_i = a_i b_i$ und $p_i = a_i \vee b_i$ (einfacher Volldadder)
Rekursive Ermittlung der ersten 4 Überträge:
$c_0 = g_0 \vee (p_0 c_{-1})$
$c_1 = g_1 \vee (p_1 c_0) = g_1 \vee (p_1 (g_0 \vee (p_0 c_{-1}))$
$= g_1 \vee p_1 g_0 \vee p_1 p_0 c_{-1}$
$= G_1 \vee P_1\, c_{-1}$
$c_2 = g_2 \vee (p_2 c_1) = g_2 \vee (p_2 (g_1 \vee p_1\, g_0 \vee p_1 p_0 c_{-1}))$
$= g_2 \vee p_2 g_1 \vee p_2 p_1 g_0 \vee p_2 p_1 p_0 c_{-1}$
$= G_2 \vee P_2\, c_{-1}$
$c_3 = g_3 \vee (p_3 c_2) = g_3 \vee (p_3 (g_2 \vee p_2 g_1 \vee p_2 p_1 g_0 \vee p_2 p_1 p_0 c_{-1})$
$= g_3 \vee p_3 g_2 \vee p_3 p_2 g_1 \vee p_3 p_2 p_1 g_0 \vee p_3 p_2 p_1 p_0 c_{-1}$
$= G_3 \vee P_3\, c_{-1}$ mit $G_3 = g_3 \vee p_3\, G_2$ und $P_3 = p_3\, P_2$
usw.

Verallgemeinerung für die ALU:
Generate/Propagate-Berechnung:
$\mathbf{G_i = g_i \vee p_i\, G_{i-1}}$ $\mathbf{P_i = p_i\, P_{i-1}}$
Mit
$g_i = s_3\, a_i\, b_i \vee s_2\, a_i\, \overline{b_i}$ und $\overline{p_i} = s_1\, \overline{a}\, b \vee s_0\, \overline{a}\, \overline{b}$

Übertragsberechnung:
$\mathbf{c_i = G_i \vee P_i\, c_{-1}}$
Summenberechnung:
$\mathbf{q_i = (g_i \vee \overline{p_i}) \equiv (BA \vee c_{i-1})}$ (s. oben)

Beispiel: 4-Bit-ALU: Vergleich von Logiktiefe (Verarbeitungszeit) und Gatteraufwand

1. parallele Realisierung:
3 Stufen, jedoch immer mehr Gattereingänge für jede Bitstelle erforderlich
"Carry-Look-ahead-Technik **CLA**"
$G_3 = a_3 b_3 \vee (a_3 \vee b_3) a_2 b_2 \vee (a_3 \vee b_3)(a_2 \vee b_2) a_1 b_1 \vee (a_3 \vee b_3)(a_2 \vee b_2)(a_1 \vee b_1) a_0 b_0$

2. stufenförmige Realisierung
pro Stufe zwei 2-Eingangsgatter (zwei Zeitintervalle) mehr, jedoch minimierte Gatterzahl:
"Carry-Ripple-Addierer **CRA**"
$G_3 = a_3 b_3 \vee (a_3 \vee b_3)\, \{a_2 b_2 \vee (a_2 \vee b_2)\, [a_1 b_1 \vee (a_1 \vee b_1) a_0 b_0\,]\, \}$

3. Kombinationen CLA, CRA
Blockweise CLA-Übertragsberechnung
am Ausgang einer 4-bit-ALU werden z.B. G_3 und P_3 als CLA-Ausgang zur Verfügung gestellt:
$G_3 = g_3 \vee p_3 g_2 \vee p_3 p_2 g_1 \vee p_3 p_2 p_1 g_0$ und $P_3 = p_3 p_2 p_1 p_0$

Die Überträge der 8-Bit-ALU nutzen G_3 und P_3 des unteren 4 Bit-Blockes:
$G_4 = g_4 \vee p_4 G_3$ $P_4 = p_4\, P_3$
$c_4 = G_4 \vee P_4\, c_{-1}$
usw. bis
$G_7 = g_7 \vee p_7 g_6 \vee p_7 p_6 g_5 \vee p_7 p_6 p_5 g_4 \vee p_7 p_6 p_5 p_4 G_3$ $P_7 = p_7 p_6 p_5 p_4 P_3$
$c_7 = G_7 \vee P_7\, c_{-1}$

Weiteres dazu im 2. Teil (Rechnerarchitekturen) der Digitalen Systeme.