VHDL Datentypen: std_ulogic, std_logic

- **Mehrwertige Datentypen**
- **Datentypen mit Auflösungsfunktion**

Prof. Dr. J. Reichardt
Lehrbuch Digitaltechnik
Eine Einführung mit VHDL, 3. A
Oldenbourg Wissenschaftsverlag
München 2013
ISBN 978-3-486-72765-4

Mehrwertige Datentypen (`std_ulogic`, `std_logic`)

- **Erfordert Einbindung der IEEE-Bibliothek:**

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

- **In der Bibliothek sind neun verschiedene Signalwerte deklariert:**

Wert	Bedeutung	Verwendung
'U'	Nicht initialisiert	Das Signal ist im Simulator (noch) nicht initialisiert
'X'	Undefinierter Pegel	Simulator erkennt mehr als einen aktiven Signaltreiber (Buskonflikt)
'0'	Starke logische 0	L-Pegel eines Standardausgangs
'1'	Starke logische 1	H-Pegel eines Standardausgangs
'Z'	Hochohmig bzw. floatend	Three-State-Ausgang
'W'	Schwach unbekannt	Simulator erkennt Buskonflikt zwischen schwachen L- und H-Pegeln
'L'	Schwacher L-Pegel	Open-Source-Ausgang mit Pull-Down-Widerstand
'H'	Schwacher H-Pegel	Open-Drain-Ausgang mit Pull-Up-Widerstand
'-'	Don’t-Care	Logikzustand des Ausgangssignals bedeutungslos, kann für Minimierung verwendet werden

Datentyp mit Auflösungsfunktion (`std_logic`)

- **Auflösungsfunktion `resolved` bestimmt den Signalwert, wenn zwei Treiber für das gleiche Signal existieren:**
  - **In der Hardware: verbinden von Gatterausgängen**
  - **In VHDL: zwei Prozesse treiben das gleiche Signal**

Signalwert von Treiber B \ Signalwert von Treiber A	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'
'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
'X'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'0'	'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'
'1'	'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'
'Z'	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'
'W'	'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'
'L'	'U'	'X'	'0'	'1'	'L'	'W'	'L'	'W'	'X'
'H'	'U'	'X'	'0'	'1'	'H'	'W'	'W'	'H'	'X'
'-'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'

Vor- und Nachteile des Datentyps `std_logic`

- **Vorteile:**
  - **Erlaubt Open-Drain- und Three-State-Schaltungen**
  - **Modellierung nicht initialisierter Signale möglich**
  - **Unterstützt arithmetische Operatoren (+, -, *)**

- **Nachteile:**
  - **Höherer Simulationsaufwand durch Aufruf der Auflösungsfunktion bei jeder Signalzuweisung**
  - **Ungewolltes Kurzschließen von Standard-Gatterausgängen wird erst bei der Implementierung erkannt**
  - **Erfordert Einsatz von Konversionsfunktionen zwischen den Datentypen**

Weniger erfahrene VHDL-Entwickler sollten den Datentyp std_logic nur an den Stellen verwenden, wo er wirklich erforderlich ist.

Konversionsfunktionen

- **In der IEEE-Bibliothek definierte Konversions-funktionen:**

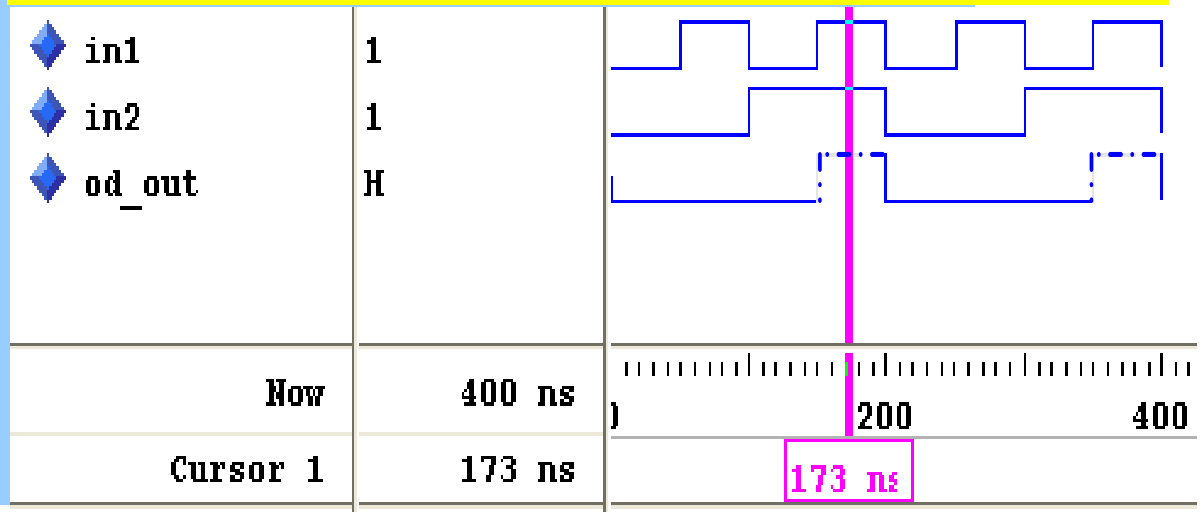
Konversionsfunktion	Argumenttyp	Ergebnistyp
`To_bit`	- `std_ulogic` - `std_logic`	- `bit`
`To_StdULogic`	- `bit`	- `std_ulogic` - `std_logic`
`To_bitvector`	- `std_ulogic_vector` - `std_logic_vector`	- `bit_vector`
`To_StdULogicVector`	- `bit_vector` - `std_logic_vector`	- `std_ulogic_vector`
`To_StdLogicVector`	- `bit_vector` - `std_ulogic_vector`	- `std_logic_vector`

Die Konversionsfunktionen `To_bit` bzw. `To_StdULogic` können für beide Datentypen `std_ulogic` und `std_logic` verwendet werden!

VHDL-Modellierung von Open-Drain-Ausgängen

```
library ieee;
use ieee.std_logic_1164.all;
entity OPEN_DRAIN is
    port(
    IN1, IN2: in bit;
    OD_OUT: out std_logic
    );
end OPEN_DRAIN;
architecture TEST of OPEN_DRAIN is
function To_stdlogic ( b : bit) return std_logic is -- wie letzte Folie…
…
begin
 P1: process(IN1)
 begin
   OD_OUT <= To_stdlogic(IN1);
   if IN1 = '1' then OD_OUT <= 'H';
   end if;
 end process P1;
 P2: process(IN2)
 begin
   OD_OUT <= To_stdlogic(IN2);
   if IN2 = '1' then OD_OUT <= 'H';
   end if;
 end process P2;
end TEST;
```

Die Schaltung hat die Funktion eines Wired-AND



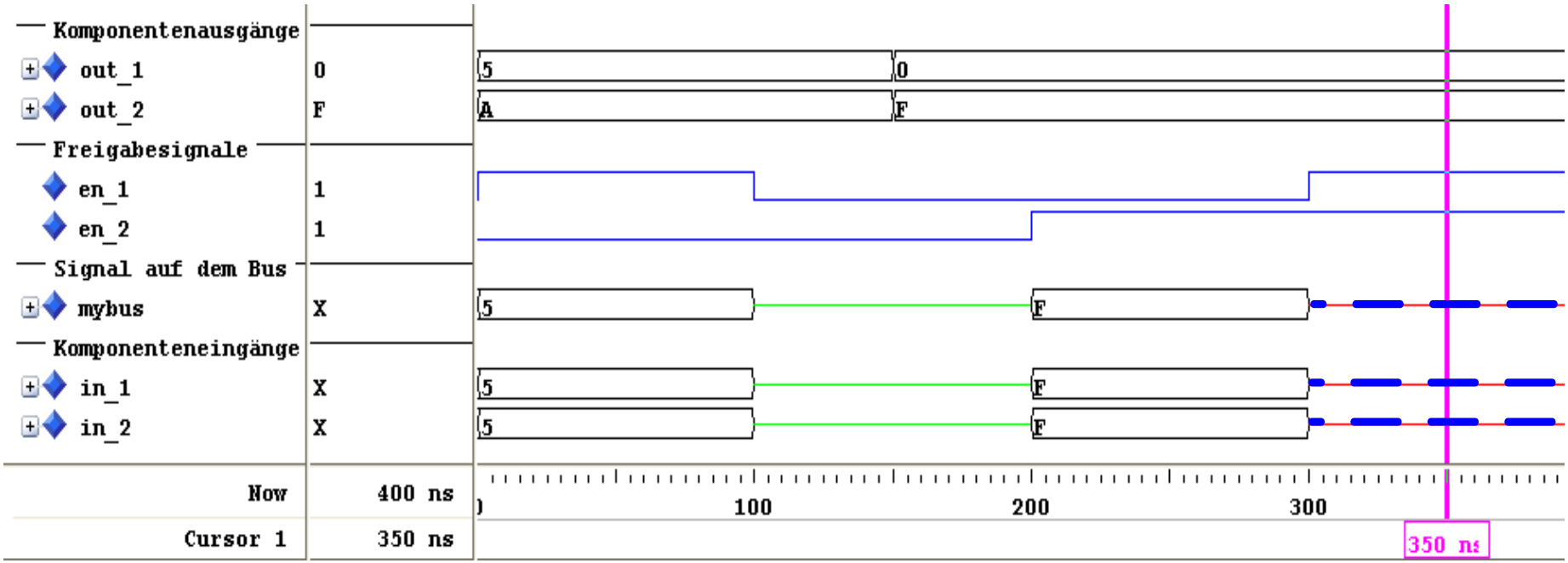
Bussystem mit Three-State-Treibern

```
library ieee; use ieee.std_logic_1164.all;
entity BUS_SYSTEM is
end BUS_SYSTEM;
architecture TEST of BUS_SYSTEM is
signal IN_1, OUT_1 : std_logic_vector(3 downto 0);
signal IN_2, OUT_2 : std_logic_vector(3 downto 0);
signal EN_1, EN_2 : bit;
signal MYBUS: std_logic_vector(3 downto 0);
begin
-- Bus_Komponente_1 mit nebenläufigen Signalzuweisungen
IN_1 <= MYBUS;
MYBUS <= OUT_1 when EN_1 = '1' else (others=>'Z');
-- Bus_Komponente_2 mit Prozess
P1: process(OUT_2, EN_2)
begin
    if EN_2 = '1' then MYBUS <= OUT_2;
    else MYBUS <= (others=>'Z');
end if;
end process P1;
IN_2 <= MYBUS;
end TEST;
```

1. Möglichkeit: bedingt nebenläufig

2. Möglichkeit: mit Prozess und `if`

Simulation des Three-State-Bussystems



- **Wenn keiner der beiden Treiber aktiv ist, zeigt der Bus einen hochohmigen Zustand an (durchgezogene grüne Linie).**
- **Wenn beide Treiber aktiv sind, so wird ein undefiniertes Signal X angezeigt (gestrichelte Linie).**

Datenpfadkomponenten

- Multiplexer
- Binärzahlendecode und Demultiplexer
- Prioritätsencoder
- Code-Umsetzer
- Komparator
- Hierarchische Strukturen in VHDL
- Addierer
- Multiplizierer
- Arithmetik in VHDL

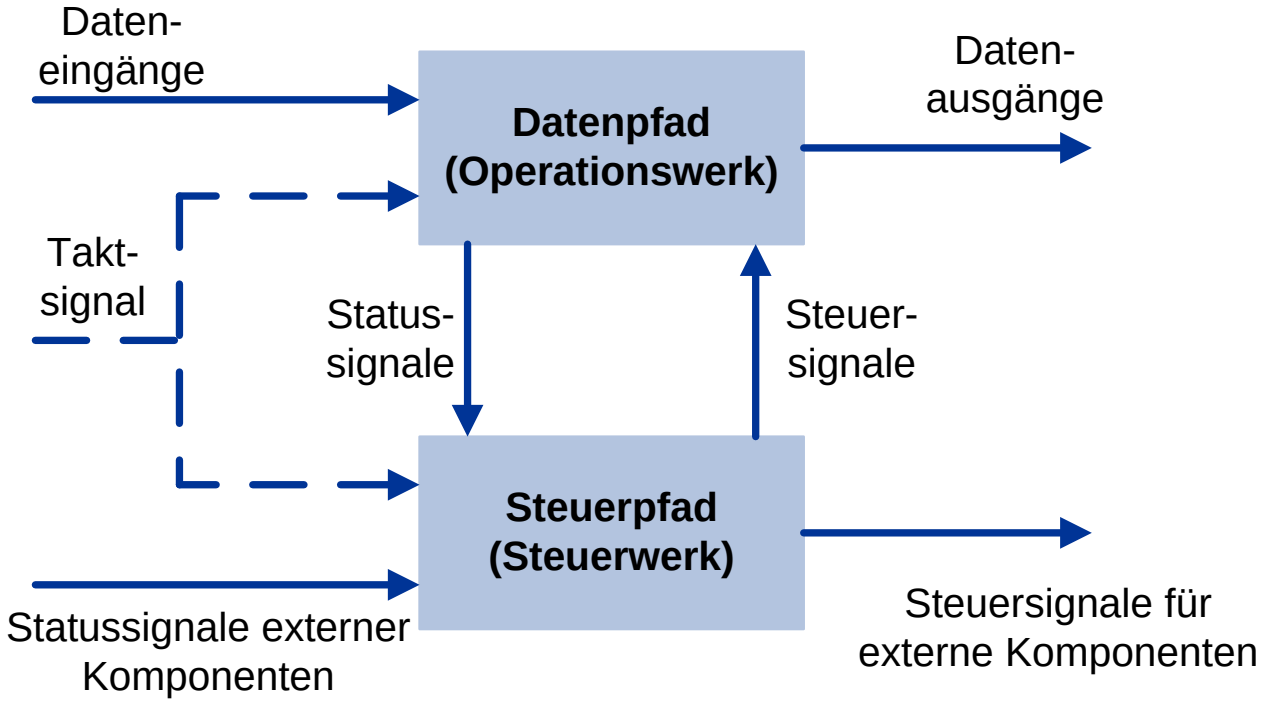
Prof. Dr. J. Reichardt
Lehrbuch Digitaltechnik
Eine Einführung mit VHDL, 3. A
Oldenbourg Wissenschaftsverlag
München 2013
ISBN 978-3-486-72765-4

Daten- und Steuerpfad

Strukturierung digitaler Systeme:

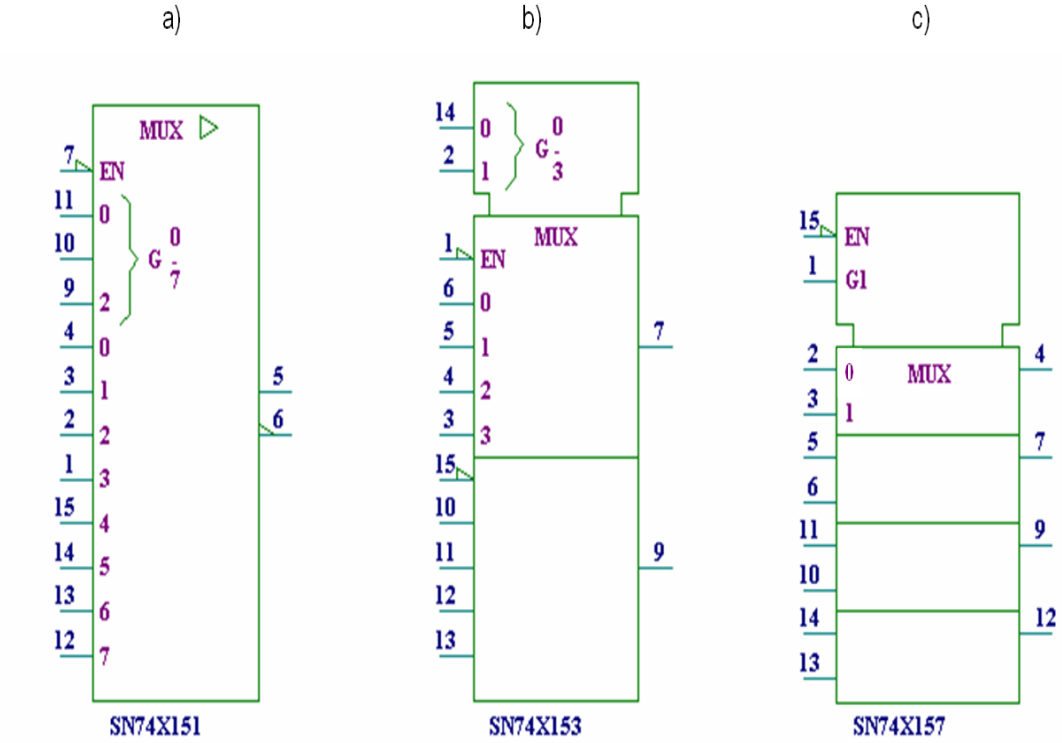
• **Datenpfad (Operationswerk): enthält die Komponenten zur Datenmanipulation und Datenflusssteuerung (kombinatorisch oder getaktet). Deren Funktion wird durch Steuersignale kontrolliert und sie können Statussignale erzeugen.**

• **Steuerpfad (Steuerwerk): ist meist ein endlicher Zustandsautomat (Finite State Machine, FSM) (vgl. Kap. 12). Dieser empfängt die Statussignale und generiert die Steuersignale für die Datenpfadkomponenten abhängig vom aktuellen Zustand des Automaten.**



Varianten von Multiplexern

- **8-zu-1 MUX 74x151**
- **Doppel 4-zu-1 MUX 74x153**
- **Vierfach 2-zu-1 MUX 74x157**



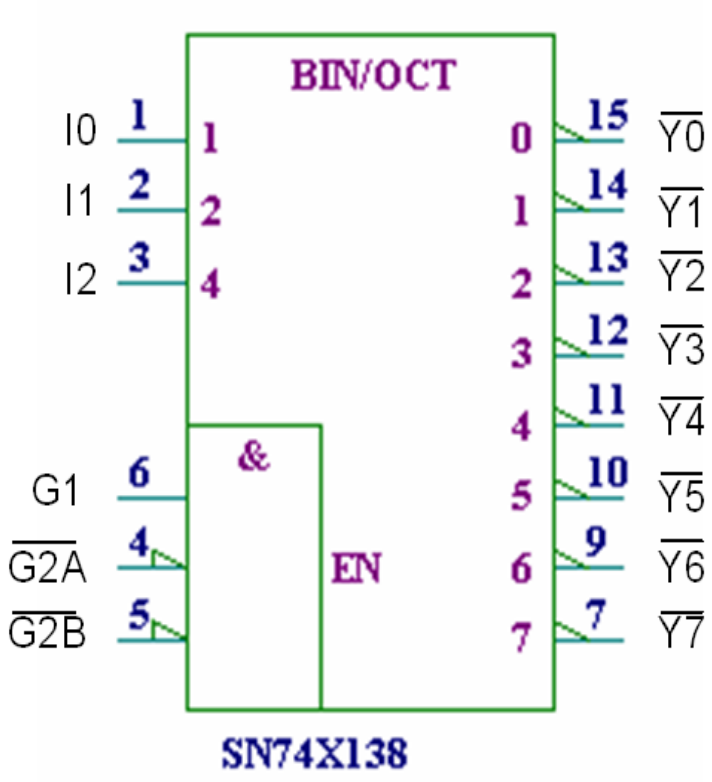

```
entity MUX4X1_EN is
   port( E : in bit_vector(3 downto 0);
         S : in bit_vector(1 downto 0);
         nEN: in bit;
         Y : out bit);
end MUX4X1_EN;
architecture VERHALTEN of MUX4X1_EN is
begin
MUXPROC: process(S, E, nEN)
   begin
    if nEN = '0' then
     case S is
       when "00" => Y <= E(0);
       when "01" => Y <= E(1);
       when "10" => Y <= E(2);
       when "11" => Y <= E(3);
     end case;
    else
       Y <= '0';
    end if;
end process MUXPROC;
end VERHALTEN;
```

Binärzahlendecoder

Ein Binärzahlendecoder hat die Aufgabe durch Auswertung eines n-Bit-Auswahlsignals, einen von 2^n Ausgängen anzusteuern.

- **Der 74x138 3-zu-8 Decoder besitzt drei Freigabesignale, die alle aktiv sein müssen.**
- **Die Eingänge I0, I1 und I2 dienen als Auswahlsignal.**

Enable			Inputs			Outputs							
G1	$\overline{G2A}$	$\overline{G2B}$	I2	I1	I0	$\overline{Y0}$	$\overline{Y1}$	$\overline{Y2}$	$\overline{Y3}$	$\overline{Y4}$	$\overline{Y5}$	$\overline{Y6}$	$\overline{Y7}$
L	X	X	X	X	X	H	H	H	H	H	H	H	H
X	H	X	X	X	X	H	H	H	H	H	H	H	H
X	X	H	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	**L**	H	H	H	H	H	H	H
H	L	L	L	L	H	H	**L**	H	H	H	H	H	H
H	L	L	L	H	L	H	H	**L**	H	H	H	H	H
H	L	L	L	H	H	H	H	H	**L**	H	H	H	H
H	L	L	H	L	L	H	H	H	H	**L**	H	H	H
H	L	L	H	L	H	H	H	H	H	H	**L**	H	H
H	L	L	H	H	L	H	H	H	H	H	H	**L**	H
H	L	L	H	H	H	H	H	H	H	H	H	H	**L**

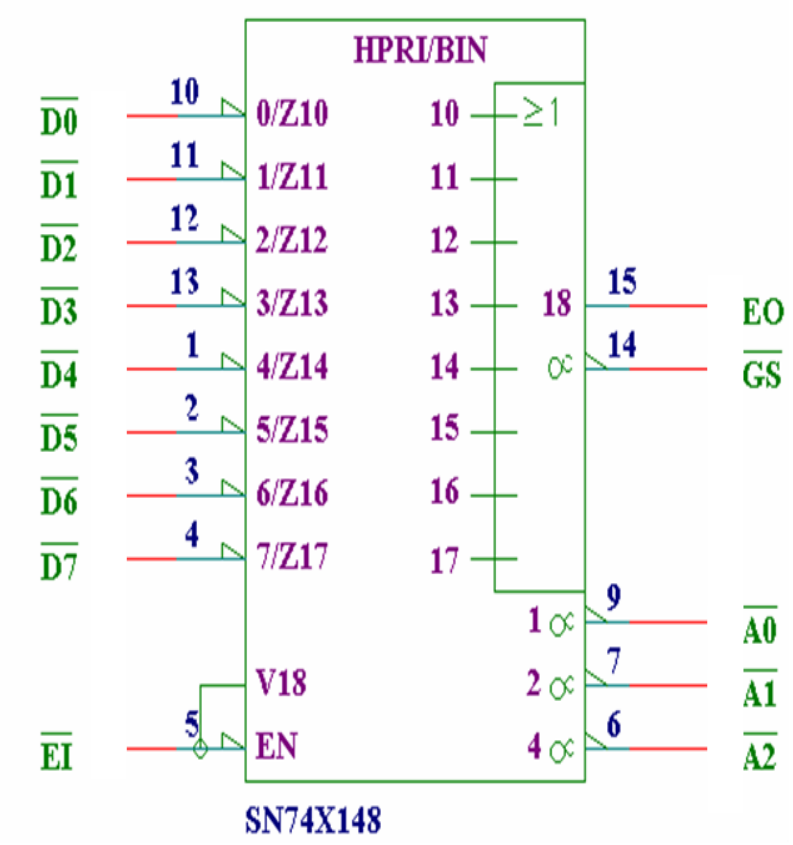


VHDL-Modell des Binärzahlendecoders

```
entity DEC_138 is
  port( I         : in bit_vector(2 downto 0);  -- Daten Eingang
        G1, nG2A, nG2B   : in bit;                -- Freigabeeingaenge
        Y_N       : out bit_vector(7 downto 0));-- Ausgangssignale
end DEC_138;
architecture DEC3_8 of DEC_138 is
signal EN: bit;                                 -- Lokales Freigabesignal
begin
EN <= G1 and not nG2A and not nG2B;
process(I, EN)
variable TEMP: bit_vector(3 downto 0);      -- Lokale Testvariable
    begin
        Y_N <= (others => '1');             -- Default Zuweisung; Aggregat
        TEMP := EN & I  ;                   -- Verknuepfung: Vektor mit Bit
          case TEMP is
                when "1000"       => Y_N(0) <= '0';
                when "1001"       => Y_N(1) <= '0';
                when "1010"       => Y_N(2) <= '0';
                ...
                when "1111"       => Y_N(7) <= '0';
                when others       => null; -- fuer EN=0: waehle Default
          end case;
     end process;
end DEC3_8;
```

Binärencoder / Prioritätsencoder

Binärencoder besitzen die umgekehrte Funktion von Binärdecodern: Sie generieren aus 2^n Eingangssignalen ein binär codiertes Ausgangssignal. Wenn gleichzeitig mehrere Eingangssignalleitungen aktiv sind, so wird das Signal mit der höchsten Priorität codiert (Prioritätsencoder).



	Dateneingänge									Ausgänge			
$\overline{EI}$	$\overline{D7}$	$\overline{D6}$	$\overline{D5}$	$\overline{D4}$	$\overline{D3}$	$\overline{D2}$	$\overline{D1}$	$\overline{D0}$	$\overline{GS}$	$\overline{A2}$	$\overline{A1}$	$\overline{A0}$	EO
H	X	X	X	X	X	X	X	X	H	H	H	H	H
L	**L**	X	X	X	X	X	X	X	L	L	L	L	H
L	H	**L**	X	X	X	X	X	X	L	L	L	H	H
L	H	H	**L**	X	X	X	X	X	L	L	H	L	H
L	H	H	H	**L**	X	X	X	X	L	L	H	H	H
L	H	H	H	H	**L**	X	X	X	L	H	L	L	H
L	H	H	H	H	H	**L**	X	X	L	H	L	H	H
L	H	H	H	H	H	H	**L**	X	L	H	H	L	H
L	H	H	H	H	H	H	H	**L**	L	H	H	H	H
L	**H**	**H**	**H**	**H**	**H**	**H**	**H**	**H**	**H**	H	H	H	**L**

VHDL-Modell des Prioritätsencoders

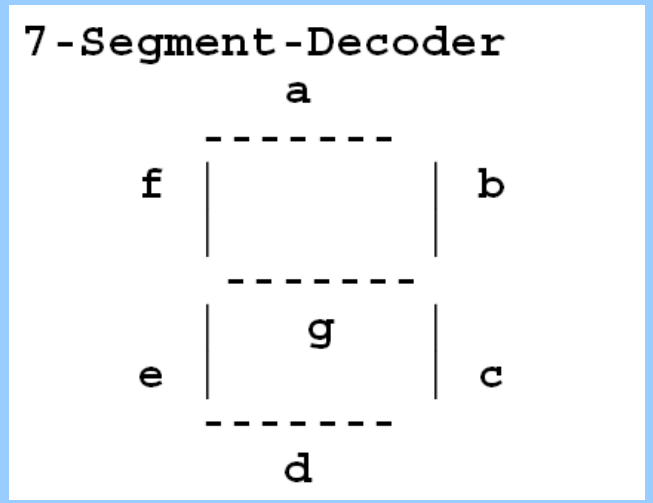
```
entity P_ENC_148 is
        port( D_N  : in bit_vector(7 downto 0); -- Prioritaets-Eingaenge
              nEI  : in bit;                    -- Freigabe
              A_N  : out bit_vector(2 downto 0);-- Binaerer Ausgang
            EO, nGS: out bit);                  -- Kaskadierungsausgaenge
end P_ENC_148;
architecture PEN8_3 of P_ENC_148 is
begin
process(D_N, nEI)
begin
  A_N <= "111"; nGS <= '1'; EO <= '1';      -- Default Zuweisungen
  if nEI = '0' then
    nGS <= '0';
    if    D_N(7) = '0' then A_N <= "000";   -- Invertierte 7
    elsif D_N(6) = '0' then A_N <= "001";
    elsif D_N(5) = '0' then A_N <= "010";
    elsif D_N(4) = '0' then A_N <= "011";
    elsif D_N(3) = '0' then A_N <= "100";
    elsif D_N(2) = '0' then A_N <= "101";
    elsif D_N(1) = '0' then A_N <= "110";
    elsif D_N(0) = '0' then A_N <= "111";   -- Invertierte 0
    else A_N <= "111"; nGS <= '1'; EO <= '0';-- Kein Eingang aktiv
    end if;
  end if;
```

Codeumsetzer

Code-Umsetzer haben die Aufgabe, einen Code in einen anderen zu überführen.

Beispiel: 7-Segment-Decoder:

```
entity SEG7 is
        port(    A: in bit_vector(3 downto 0);     -- Eingangsvektor
                 SEG: out bit_vector(6 downto 0));-- Ausgangsvektor
end SEG7;
architecture VERHALTEN of SEG7 is
begin
DECODER: process(A)
        begin
             case A is
-- Segmente                              abcdefg
             when "0000"=> SEG <="1111110"; -- 0
             when "0001"=> SEG <="0110000"; -- 1
             when "0010"=> SEG <="1101101"; -- 2
             ...
             when "1110"=> SEG <="1001111"; -- E
             when "1111"=> SEG <="1000111"; -- F
             end case;
        end process DECODER;
end VERHALTEN;
```



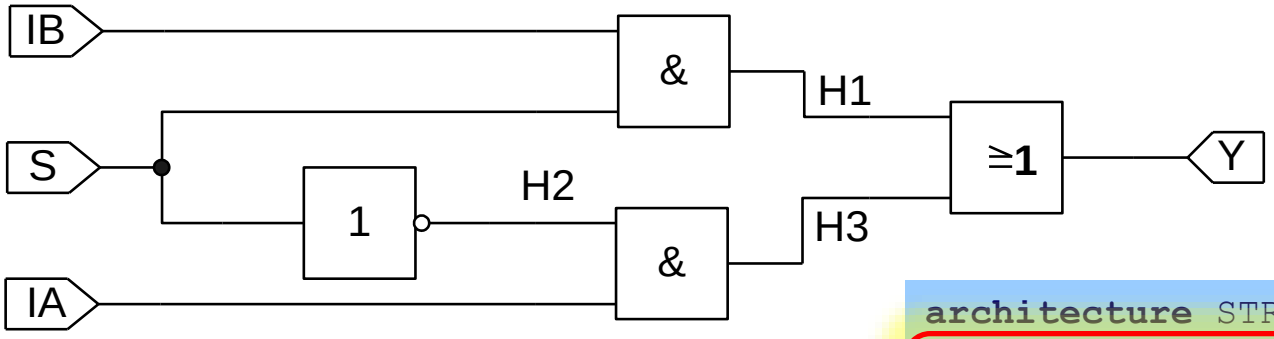
Hierarchische Strukturmodellierung in VHDL

- **Jedes `entity/architecture`-Paar lässt sich als untergeordnete Komponente wieder verwenden. Alle compilierten Hardwarefunktionsblöcke werden in eine Bibliothek mit dem Namen work geschrieben und können von dort durch ein Modell auf einer höheren Hierarchieebene als Bibliotheksmodul eingebunden werden.**

VHDL-Code, der Komponenten verwendet, muss die folgenden Codeelemente besitzen:

- **Eine Komponente*deklaration* vor dem `architecture begin`. Die Komponentendeklaration muss dieselben Port-Signalnamen und -typen verwenden, wie die zugehörige `entity`.**
- **Eine oder mehrere Komponenten*instanziierungen* mit je einer `port map`-Anweisung nach dem `architecture begin`.**
- **Sollte es zu einer `entity` in der work-Bibliothek mehrere Architekturen geben, so ist zusätzlich auch eine Komponenten*konfiguration* vor dem `architecture begin` erforderlich.**
- **Bei der Komponenten-Instanziierung werden alle Komponenteneingänge mit Signalen verbunden. Nicht benutzte Komponentenausgänge können offen bleiben (Schlüsselwort `open`).**

Hierarchisches Modell eines 2-zu-1-Multiplexers



- **Die Komponenten UND und ODER müssen in der ./work-Bibliothek vorhanden sein.**
- **Instanzen U1 und U3: Übergabe der Signale in der port map in der Reihenfolge, wie sie in der Komponentendeklaration bzw. in der Komponenten-entity angegeben sind (engl. positional order association).**
- **Instanz U2: Übergabe der Signale durch namentliche Zuordnung der aktuellen Signale (engl. named order association) zu den lokalen Signalen der Komponente in der Form**

  `<local_signal> => <actual_signal>`

  (Reihenfolge der Schnittstellensignale ist hier beliebig !)

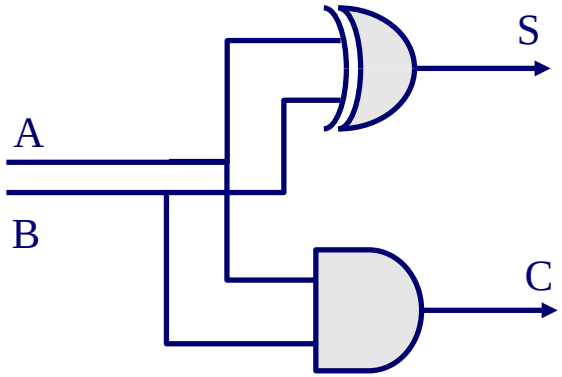
```
architecture STRUKT of MUX_STRUKT is
component UND is -- UND Komponentendeklaration
        port (A, B : in bit; Y: out bit);
end component;

component ODER is -- ODER Komponentendeklaration
        port (A, B : in bit; Y: out bit);
end component;
-- Komponentenkonfigurationen
for U1, U2: UND use entity work.UND(A);
for U3: ODER use entity work.ODER(A);

signal H1, H2, H3 : bit;
begin
H2 <= not S after 2 ns;

U1: UND port map(S, IB, H1); -- Instanziierungen
U2: UND port map(Y=>H3,
                 A=>IA,
                 B=>H2);
U3: ODER port map(H1, H3, Y);
end STRUKT;
```

Addierer

- **Halb-Addierer**

$$S = A \oplus B$$

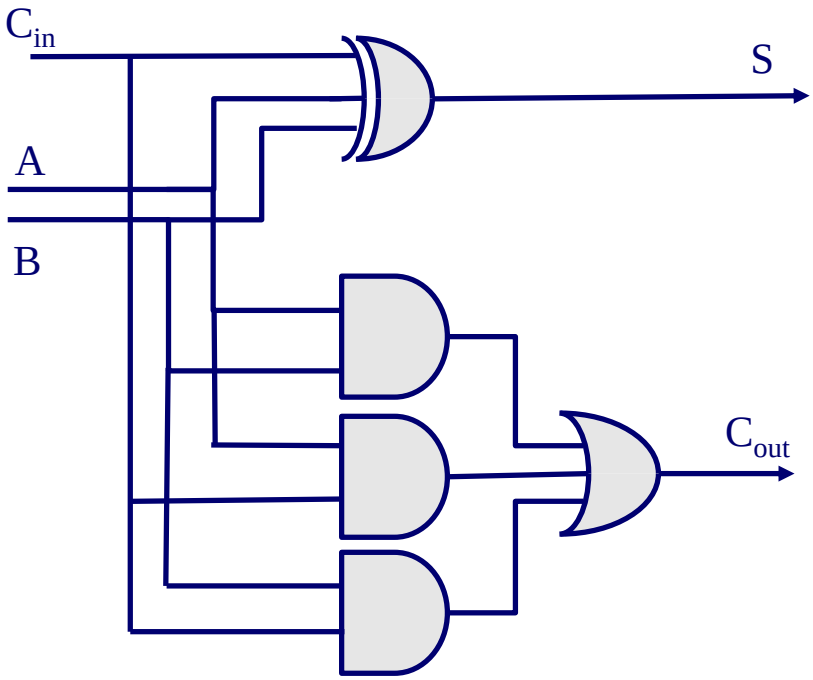
$$C = A\,B$$



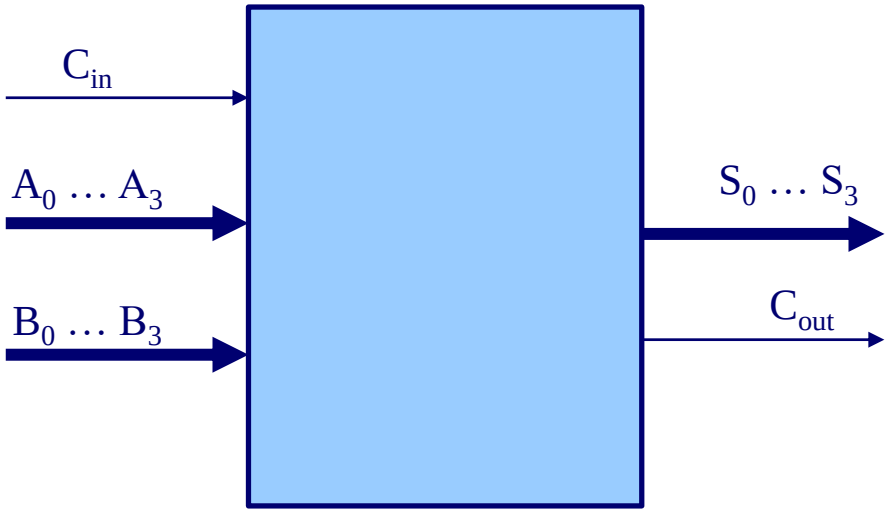
- **Voll-Addierer**

$$S = A_i \oplus B_i \oplus C_{in}$$

$$C_{out} = A\,B + AC_{in} + BC_{in}$$



4-Bit Addierer

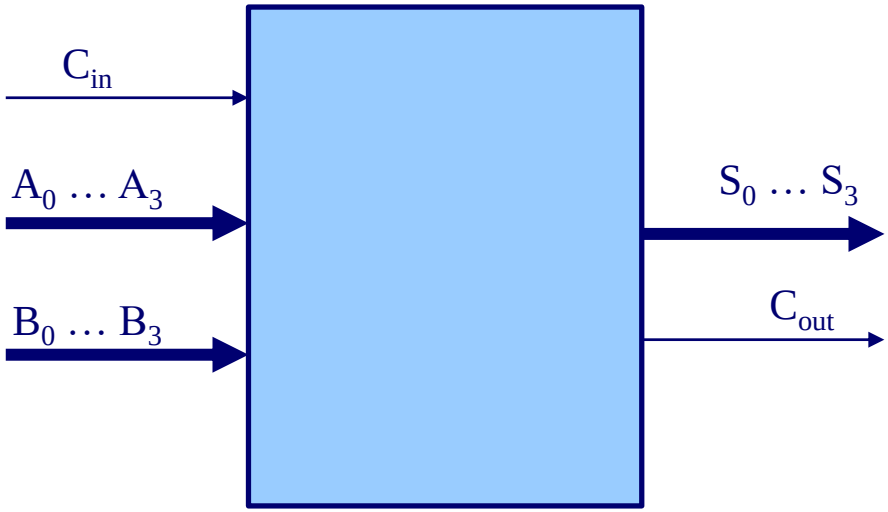


- **4 Bit Addierer**
- **Funktion mit**
  - **9 Inputs**
  - **5 Outputs**

$$S_i = A_i \oplus B_i \oplus C_{i-1}$$

$$C_i = A_i B_i + A_i C_{i-1} + B_i C_{i-1}$$

4-Bit Addierer



$$S_i = A_i \oplus B_i \oplus C_{i-1}$$

$$C_i = A_i B_i + A_i C_{i-1} + B_i C_{i-1}$$

$$S_0 = A_0 \oplus B_0 \oplus C_{in}$$
$$S_1 = A_1 \oplus B_1 \oplus C_0$$
$$S_2 = A_2 \oplus B_2 \oplus C_1$$
$$S_3 = A_3 \oplus B_3 \oplus C_2$$

$$C_0 = A_0 B_0 + A_0 C_{in} + B_0 C_{in}$$
$$C_1 = A_1 B_1 + A_1 C_0 + B_1 C_0$$
$$C_2 = A_2 B_2 + A_2 C_1 + B_2 C_1$$
$$C_{out} = A_3 B_3 + A_3 C_2 + B_3 C_2$$

4-Bit Addierer

1. Implementierung mit vollständiger Expansion

$$S_i = A_i \oplus B_i \oplus C_{i-1}$$

$$C_i = A_i B_i + A_i C_{i-1} + B_i C_{i-1}$$

$$C_0 = A_0 B_0 + A_0 C_{in} + B_0 C_{in}$$

$$C_1 = A_1 B_1 + A_1 C_0 + B_1 C_0$$
$$= A_1 B_1 + A_1 (A_0 B_0 + A_0 C_{in} + B_0 C_{in}) + B_1 (A_0 B_0 + A_0 C_{in} + B_0 C_{in})$$
$$= A_1 B_1 + A_1 A_0 B_0 + A_1 A_0 C_{in} + A_1 B_0 C_{in} + B_1 A_0 B_0 + B_1 A_0 C_{in} + B_1 B_0 C_{in}$$

$$C_2 = A_2 B_2 + A_2 C_1 + B_2 C_1$$
$$= A_2 B_2$$
$$+ A_2 (A_1 B_1 + A_1 A_0 B_0 + A_1 A_0 C_{in} + A_1 B_0 C_{in} + B_1 A_0 B_0 + B_1 A_0 C_{in} + B_1 B_0 C_{in})$$
$$+ B_2 (A_1 B_1 + A_1 A_0 B_0 + A_1 A_0 C_{in} + A_1 B_0 C_{in} + B_1 A_0 B_0 + B_1 A_0 C_{in} + B_1 B_0 C_{in})$$

$$C_{out} = A_3 B_3 + A_3 C_2 + B_3 C_2$$

4-Bit Addierer

1. Implementierung mit vollständiger Expansion

Größe der Carry Funktion:

$\mathrm{M}(C_i)$ .. Anzahl der Und-Terme in der C_i Funktion

$$\mathrm{M}(C_0) = 3$$
$$\mathrm{M}(C_1) = 7$$
$$\mathrm{M}(C_2) = 15$$
$$\mathrm{M}(C_3) = 31$$

$$\mathrm{M}(C_i) = 2 \cdot \mathrm{M}(C_{i-1}) + 1 = 2^{i+2} - 1$$

$$\mathrm{M}(C_7) = 2^9 - 1 = 511$$
$$\mathrm{M}(C_{15}) = 2^{17} - 1 = 131\,071$$
$$\mathrm{M}(C_{31}) = 2^{33} - 1 = 8\,589\,934\,591$$
$$\mathrm{M}(C_{63}) = 2^{65} - 1 \sim 36.9\,10^{18}$$
$$\mathrm{M}(C_{127}) = 2^{129} - 1 \sim 6.8\,10^{38}$$

4-Bit Addierer

1. Implementierung mit vollständiger Expansion

Kosten der Carry Funktion (Anzahl der Gates + Anzahl der Eingänge):

$\#\mathrm{Gmax}(C_i)$ .. Anzahl der Eingänge des größten Gates in der C_i Funktion

$$\#\mathrm{Gmax}(C_0) = 2$$
$$\#\mathrm{Gmax}(C_1) = 3$$
$$\#\mathrm{Gmax}(C_2) = 4$$
$$\#\mathrm{Gmax}(C_3) = 5$$
$$\#\mathrm{Gmax}(C_i) = \#\mathrm{Gmax}(C_{i-1}) + 1 = i + 2$$

$$3 \cdot M(C_i) + 1 + M(C_i) \leq \mathrm{K}(C_i) \leq M(C_i)(1 + \#\mathrm{Gmax}(C_i)) + 1 + M(C_i)$$
$$3 \cdot M(C_i) + 1 + M(C_i) \leq \mathrm{K}(C_i) \leq (\mathrm{i} + 3) \cdot M(C_i) + 1 + M(C_i)$$

$C_2: 45 \leq \mathrm{K}(C_i) \leq 75$
$C_3: 93 \leq \mathrm{K}(C_i) \leq 186$
$C_7: 1533 \leq \mathrm{K}(C_i) \leq 4088$
$C_{15}: 393\,213 \leq \mathrm{K}(C_i) \leq 25\,769\,803\,773$

$C_{31}: 2.5 \cdot 10^{10} \leq \mathrm{K}(C_i) \leq 2.9 \cdot 10^{11}$
$C_{63}: 1.1 \cdot 10^{20} \leq \mathrm{K}(C_i) \leq 2.4 \cdot 10^{21}$
$C_{127}: 2.0 \cdot 10^{39} \leq \mathrm{K}(C_i) \leq 8.8 \cdot 10^{40}$

4-Bit Addierer

1. Implementierung mit vollständiger Expansion

Worst Case Delay der Carry Funktion (Tiefe der 4-Input Gates):

$\#\mathrm{Gmax}(C_i)$ ... Anzahl der Eingänge des größten Gates in der C_i Funktion

$\mathrm{Depth}(k)$ … Tiefe der 4-input Gates bei einem k-input AND oder OR Ausdrucks

$$\mathrm{Depth}(k) = \frac{\log k}{\log 4}$$

$\mathrm{Depth}(4) = 1$
$\mathrm{Depth}(16) = 2$
$\mathrm{Depth}(64) = 3$

$$\mathrm{WCD}(C_i) = \mathrm{Depth}\big(\#\mathrm{Gmax}(C_i)\big) + \mathrm{Depth}(M(C_i))$$

$WCD(C_3) = \mathrm{Depth}(5)+\mathrm{Depth}(31)=3.5$
$WCD(C_7)=6.1$
$WCD(C_{15})=10.5$

$WCD(C_{31})=19.0$
$WCD(C_{63})=35.5$
$WCD(C_{127})=68.0$

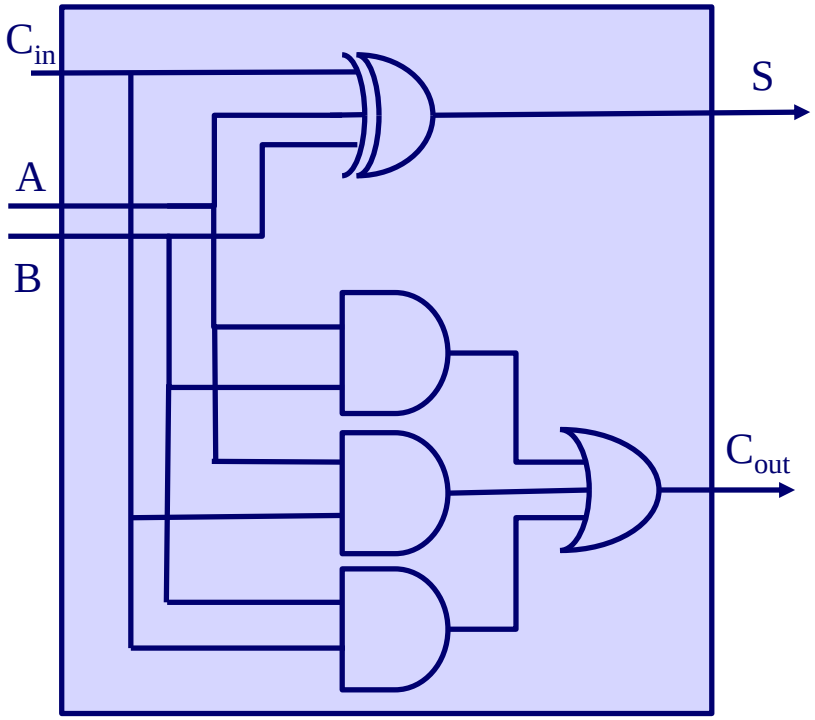
4-Bit Addierer

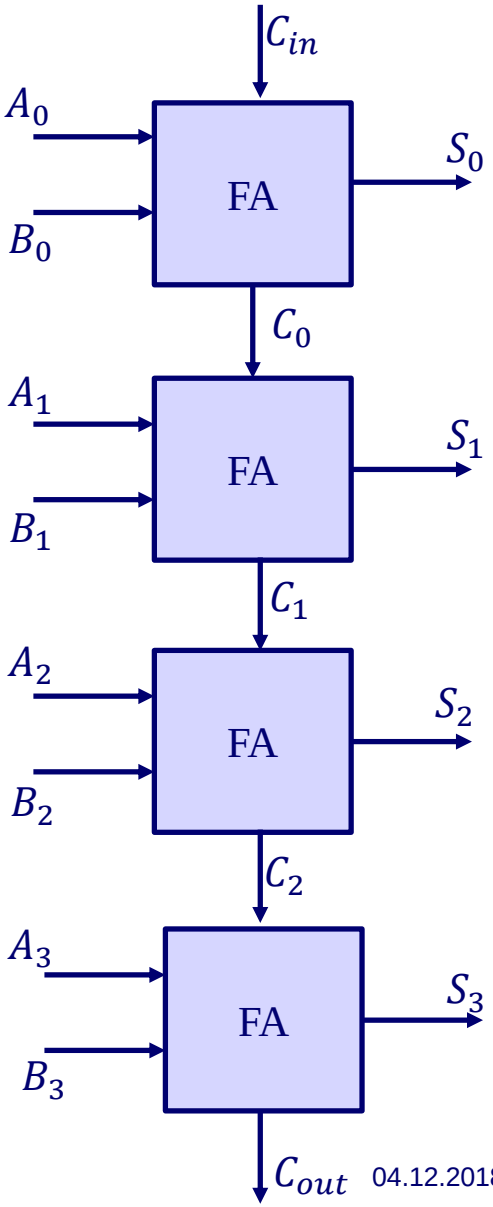
2. Implementierung mit Hilfe von Sub-Funktionen

$$S_i = A_i \oplus B_i \oplus C_{i-1}$$

$$C_i = A_i B_i + A_i C_{i-1} + B_i C_{i-1}$$

Voll-Addierer





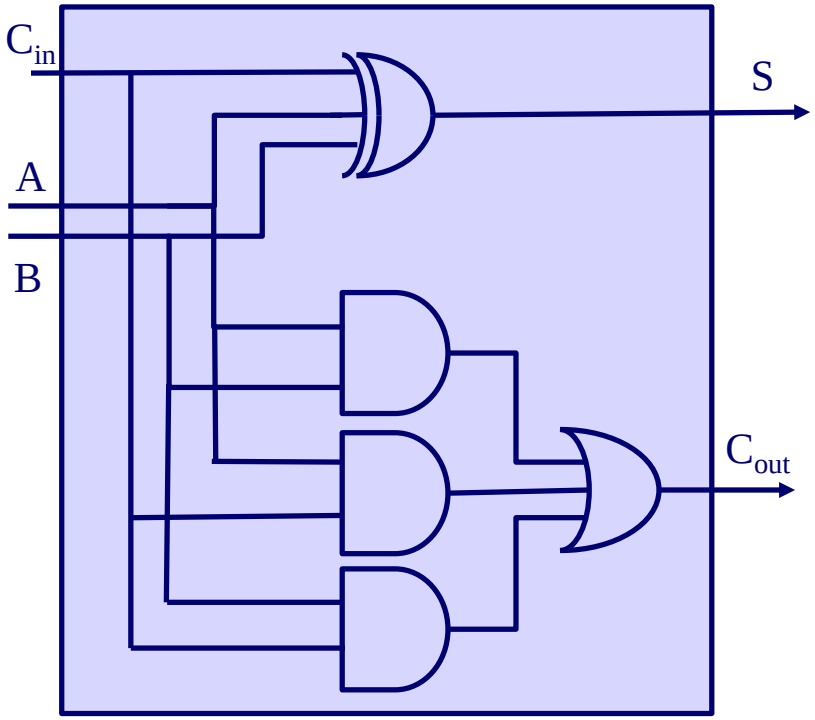
4-Bit Addierer

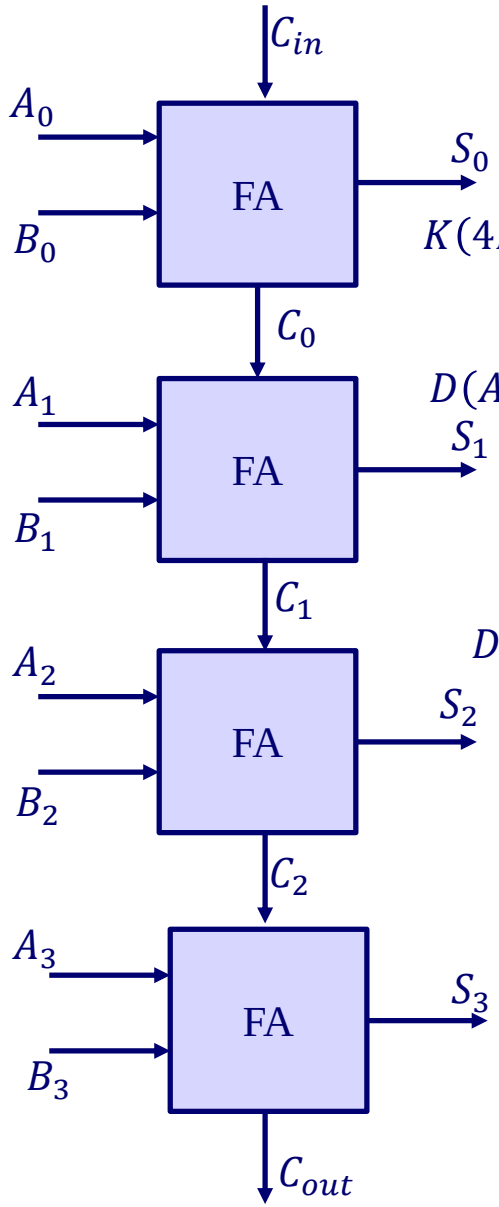
2. Implementierung mit Hilfe von Sub-Funktionen

$$S_i = A_i \oplus B_i \oplus C_{i-1}$$

$$C_i = A_i B_i + A_i C_{i-1} + B_i C_{i-1}$$

Voll-Addierer





Kosten:

$K(FA) = 2 \cdot 4 + 3 \cdot 3 = 17$

$K(4Bit\ Addierer, FA) = 4 \cdot 17 = 68$

Worst Case Delay:

$D(A_0, B_0, C_{in} \rightarrow S_0) = 1\ gate\ delay$

$D(A_0, B_0, C_{in} \rightarrow C_0) = 2$

$D(A_0, B_0, C_{in} \rightarrow S_3) = 7$

$D(A_0, B_0, C_{in} \rightarrow C_3) = 8$

$D(A_0, B_0, C_{in} \rightarrow C_i) = 2\ (i+1)$

$D(A_0, B_0, C_{in} \rightarrow C_{15}) = 32$

$D(A_0, B_0, C_{in} \rightarrow C_{31}) = 64$

$D(A_0, B_0, C_{in} \rightarrow C_{63}) = 128$

4-Bit Addierer

3. Implementierung mit Hilfe der Faktorisierung

$$S_i = A_i \oplus B_i \oplus C_{i-1}$$

$$C_i = A_i B_i + A_i C_{i-1} + B_i C_{i-1}$$

$$C_0 = A_0 B_0 + A_0 C_{in} + B_0 C_{in}$$

$$\begin{aligned} C_1 &= A_1 B_1 + A_1 C_0 + B_1 C_0 \\ &= A_1 B_1 + A_1 (A_0 B_0 + A_0 C_{in} + B_0 C_{in}) + B_1 (A_0 B_0 + A_0 C_{in} + B_0 C_{in}) \\ &= A_1 B_1 + A_1 A_0 B_0 + A_1 A_0 C_{in} + A_1 B_0 C_{in} + B_1 A_0 B_0 + B_1 A_0 C_{in} + B_1 B_0 C_{in} \end{aligned}$$

$$\begin{aligned} C_2 &= A_2 B_2 + A_2 C_1 + B_2 C_1 \\ &= A_2 B_2 \\ &+ A_2 (A_1 B_1 + A_1 A_0 B_0 + A_1 A_0 C_{in} + A_1 B_0 C_{in} + B_1 A_0 B_0 + B_1 A_0 C_{in} + B_1 B_0 C_{in}) \\ &+ B_2 (A_1 B_1 + A_1 A_0 B_0 + A_1 A_0 C_{in} + A_1 B_0 C_{in} + B_1 A_0 B_0 + B_1 A_0 C_{in} + B_1 B_0 C_{in}) \end{aligned}$$

$$C_{out} = A_3 B_3 + A_3 C_2 + B_3 C_2$$

4-Bit Addierer

3. Implementierung mit Hilfe der Faktorisierung

$$S_i = A_i \oplus B_i \oplus C_{i-1}$$
$$C_i = A_i B_i + A_i C_{i-1} + B_i C_{i-1}$$

$$G_i = A_i B_i$$
$$P_i = A_i + B_i$$

$$C_0 = A_0 B_0 + A_0 C_{in} + B_0 C_{in} = G_0 + (A_0 + B_0) C_{in} = G_0 + P_0 C_{in}$$

$$C_1 = A_1 B_1 + A_1 C_0 + B_1 C_0 = G_1 + P_1 C_0 = G_1 + P_1 G_0 + P_1 P_0 C_{in}$$

$$C_2 = A_2 B_2 + A_2 C_1 + B_2 C_1 = G_2 + P_2 C_1 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{in}$$

$$C_{out} = A_3 B_3 + A_3 C_2 + B_3 C_2$$
$$= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_{in}$$

4-Bit Addierer

3. Implementierung mit Hilfe der Faktorisierung

$$G_i = A_i B_i$$
$$P_i = A_i + B_i$$

$$C_0 = G_0 + P_0 C_{in}$$
$$C_1 = G_1 + P_1 G_0 + P_1 P_0 C_{in}$$
$$C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{in}$$
$$C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_{in}$$

Kosten:

$$K(P_i) = 3$$
$$K(G_i) = 3$$
$$K(C_0) = 3 + 3 = 6$$
$$K(C_1) = 3 + 4 + 4 = 11$$
$$K(C_2) = 3 + 4 + 5 + 5 = 17$$
$$K(C_3) = 3 + 4 + 5 + 6 + 6 = 24$$

4-Bit Addierer

3. Implementierung mit Hilfe der Faktorisierung

Kosten:

$$K(P_i) = 3$$
$$K(G_i) = 3$$
$$K(C_0) = 3 + 3 = 6$$
$$K(C_1) = 3 + 4 + 4 = 11$$
$$K(C_2) = 3 + 4 + 5 + 5 = 17$$
$$K(C_3) = 3 + 4 + 5 + 6 + 6 = 24$$

$$K(C_i) = \sum_{k=1}^{i+3} k \; - 3 + i + 3 = \frac{(i+4)(i+3)}{2} + i = \frac{i^2 + 7i + 12}{2} + i$$

$$K(C_7) = 62$$
$$K(C_{15}) = 186$$
$$K(C_{31}) = 626$$
$$K(C_{63}) = 2274$$
$$K(C_{127}) = 8642$$

4-Bit Addierer

3. Implementierung mit Hilfe der Faktorisierung

Kosten der Carry-Berechnungs-Logik:

$$K(CLA_i) = \sum_{k=0}^{i} K(C_k)$$

$$K(CLA_3) = K(C_0) + K(C_1) + K(C_2) + K(C_3) = 6 + 11 + 17 + 24 = 58$$

$$K(CLA_7) = 244$$
$$K(CLA_{15}) = 1256$$
$$K(CLA_{31}) = 7632$$
$$K(CLA_{63}) = 52\,128$$
$$K(CLA_{127}) = 382\,784$$

4-Bit Addierer

3. Implementierung mit Hilfe der Faktorisierung

$$C_0 = G_0 + P_0 C_{in}$$
$$C_1 = G_1 + P_1 G_0 + P_1 P_0 C_{in}$$
$$C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{in}$$
$$C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_{in}$$

Worst Case Delay der Carry-Look-Ahead Logic:

$$D(C_i) = Depth(i+3) + Depth(i+3)$$
$$D(C_0) = 1.6$$
$$D(C_1) = 2.0$$
$$D(C_2) = 2.3$$
$$D(C_3) = 2.6$$
$$D(C_7) = 3.3$$
$$D(C_{15}) = 4.2$$
$$D(C_{31}) = 5.1$$
$$D(C_{63}) = 6.1$$
$$D(C_{127}) = 7.0$$

4-Bit Addierer

Kosten

	i=3	7	15	31	63	127
Full Expansion	93-186	1533-4088	10^5 - 10^9	10^{10} - 10^{11}	10^{20} - 10^{21}	10^{39} - 10^{40}
Sub Function	68	136	272	544	1088	2176
Faktorisierung	58	244	1256	7632	52128	382784

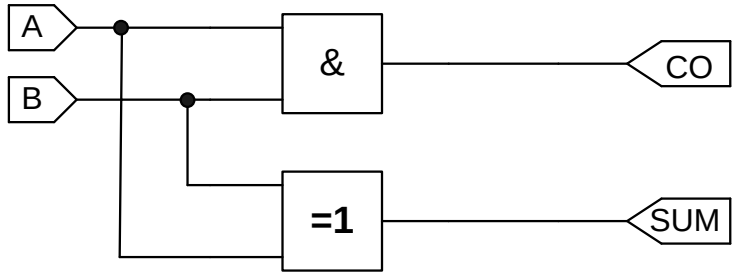
Delay

	i=3	7	15	31	63	127
Full Expansion	3.5	6.1	10.5	19.5	35.5	68.0
Sub Function	8	16	32	64	128	256
Faktorisierung	4.6	5.3	6.2	7.1	8.1	9.0

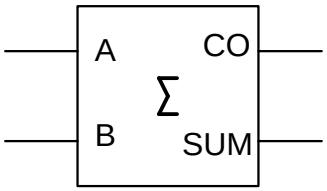
Halbaddierer

Ein Halbaddierer hat die Aufgabe, zwei Eingangsbits A und B ohne Übertragseingang miteinander zu addieren. Dabei wird ein Summationssignal SUM und ein Übertragssignal CO gebildet.

B	A	SUM	CO
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



a)



b)

$$SUM = A \leftrightarrow B \quad \text{und} \quad CO = A \wedge B$$

```
entity HALBADD is
        port (A, B :in bit; SUM, CO :out bit);
end HALBADD;
architecture VERHALTEN of HALBADD is
begin
        SUM <= A xor B after 2 ns;
        CO <= A and B after 2 ns;
end VERHALTEN;
```

Volladdierer

CIN	B	A	COUT	SUM
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

Der Wahrheitstabelle des Volladdierers entnimmt man:

$$SUM = A \leftrightarrow B \leftrightarrow CIN$$

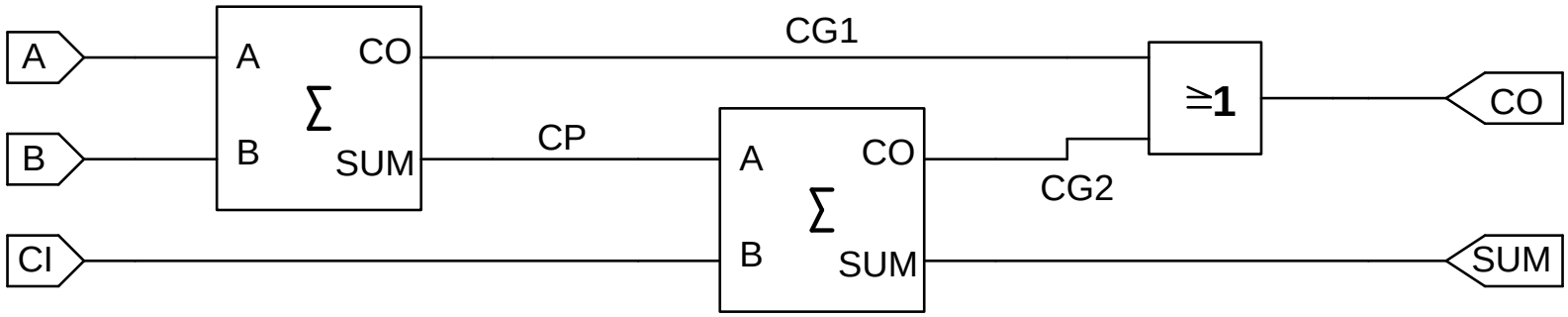
$$COUT = A \cdot B + CIN \cdot (A + B)$$

mit den Abkürzungen:

- **Carry Generate:** $G = A \cdot B$
- **Carry Propagate:** $P = A + B$

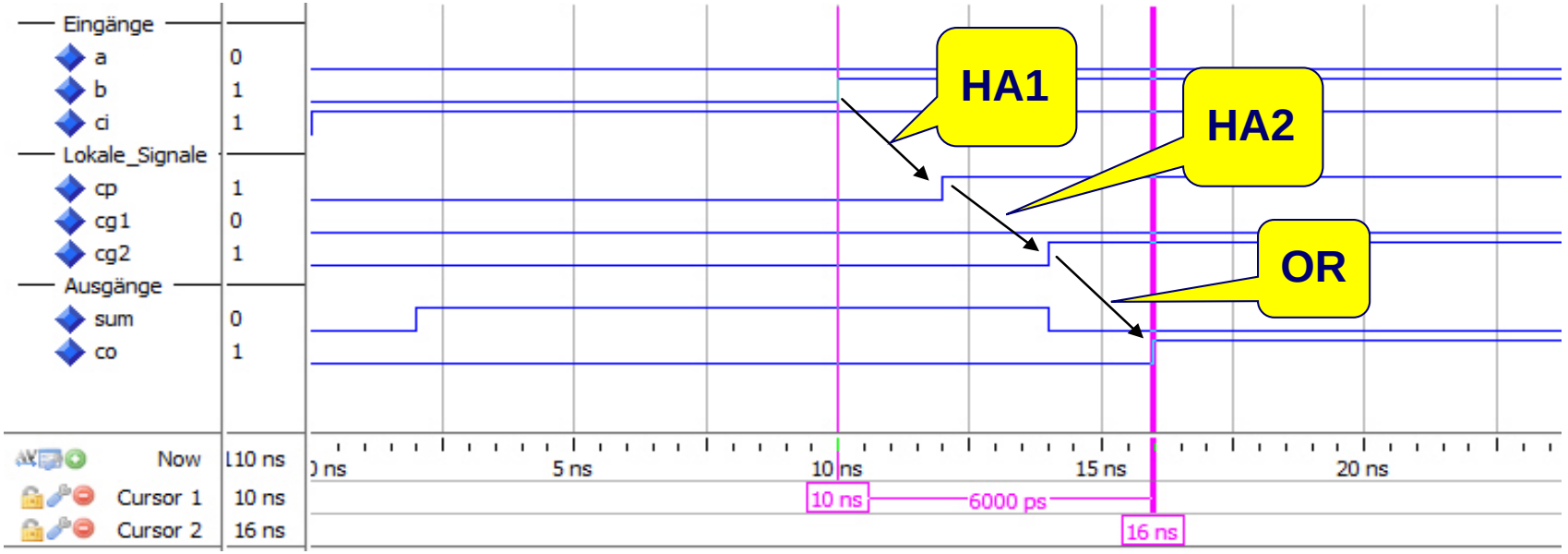
wird: $COUT = G + CIN \cdot P$

CG und CP hängen nur von den Eingangsbits A und B, nicht jedoch von CI ab !



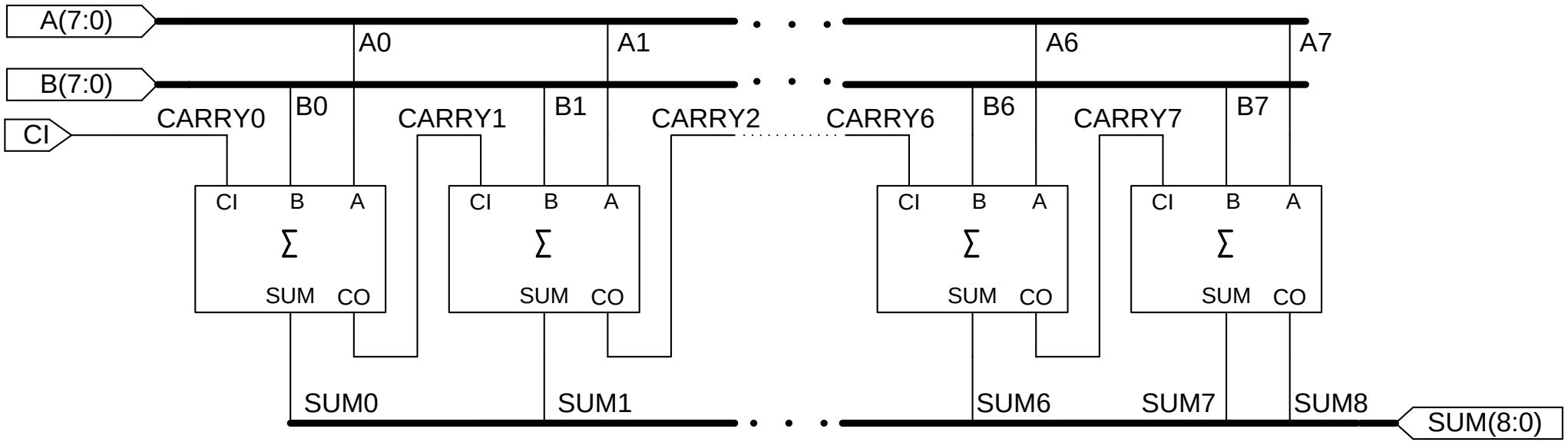
Hierarchisches Modell eines Volladdierers

```
architecture VERHALTEN of VOLLADD is
component HALBADD
        port (A, B :in bit; SUM, CO :out bit);
end component;
signal CG1, CG2, CP: bit;
begin
HA1: HALBADD
   port map(A, B, CP, CG1);
HA2: HALBADD
   port map(CP, CI, SUM, CG2);
CO <= CG1 or CG2 after 2 ns;
end VERHALTEN;
```



8-Bit-Ripple-Carry-Addierer

- **Die jeweiligen Operandenbits Ai und Bi werden an die A- und B-Eingänge der einzelnen Volladdierer geführt.**
- **Ein eventuell vorhandener Carry-Eingang für den 8-Bit-Addierer wird an den Carry-Eingang der niederwertigsten Stufe gelegt. Andernfalls wird dieser mit logisch 0 verbunden.**
- **Die Carry-Ausgänge werden in einer Kette auf die Carry-Eingänge der nachfolgenden Stufe gelegt.**
- **Das Summationsergebnis ist zur Vermeidung von Überläufen um ein Bit breiter als die Operanden. Der Carry-Ausgang der letzten Volladdiererstufe wird als höchstwertigstes Summationsbit interpretiert.**



Strukturmodell eines N-Bit-Ripple-Carry-Addierers

```
entity N_BIT_ADD is
   generic( N: integer:=8);
   port (A, B :in bit_vector(N-1 downto 0);
         CI :in bit;
         SUM:out bit_vector(N downto 0));
end N_BIT_ADD;
architecture VERHALTEN of N_BIT_ADD is
component VOLLADD
        port (A, B, CI :in bit; SUM, CO :out bit);
end component;
signal CARRY: bit_vector(N downto 0);
begin
CARRY(0) <= CI;
NBIT: for I in 0 to N-1 generate
  VA: VOLLADD
   port map(...);
end generate NBIT;
```

Parametrisierung der Bitbreite durch `generic`

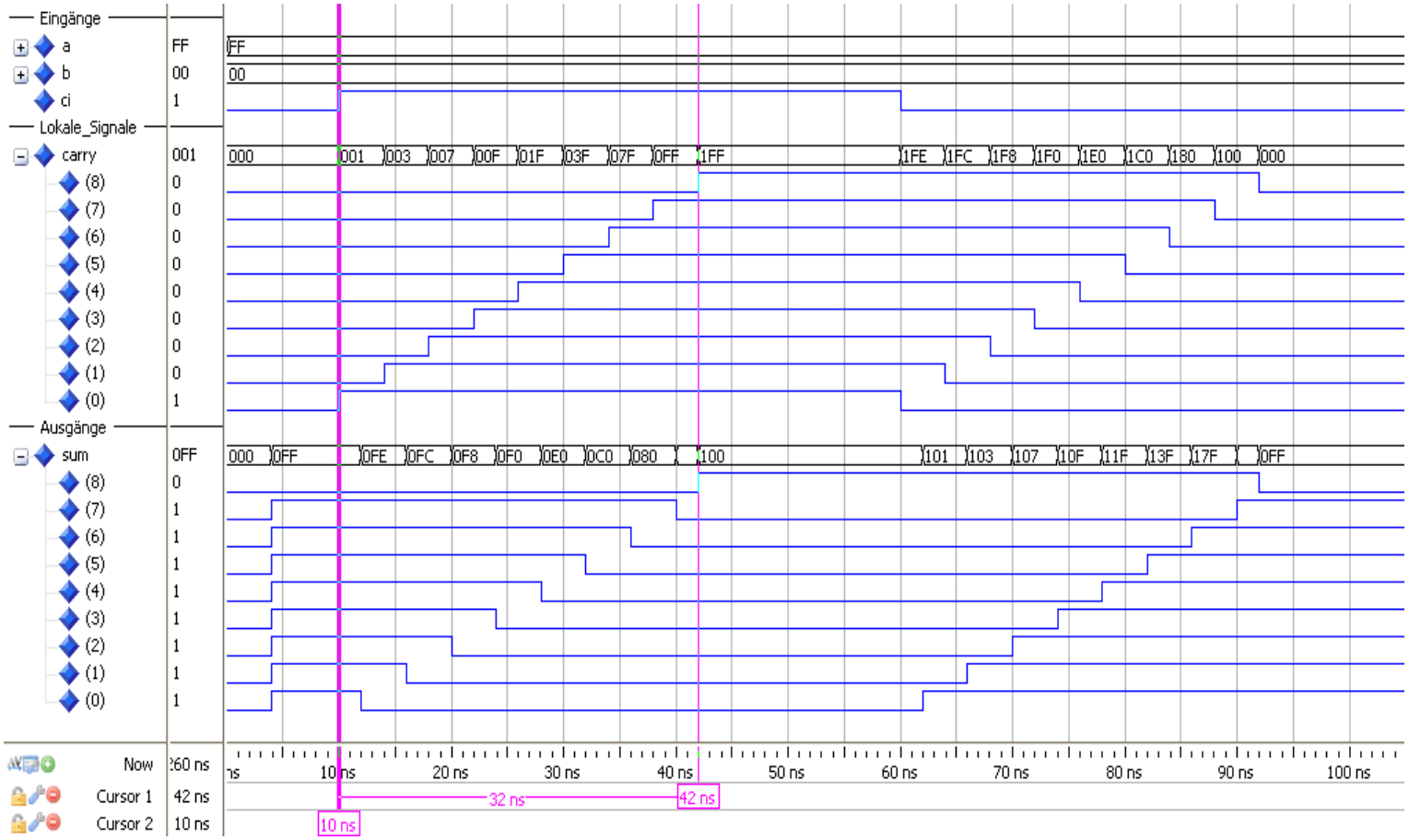
Komponenteninstanziierung in einer `for generate`-Schleife

Simulation eines 8-Bit-Ripple-Carry-Addierers

t=0: A=0xFF,
B=0x00
=> SUM = 0xFF
bei t = 4 ns

t = 10 ns: CI = 1
=> SUM = 0x100
bei t = 42 ns

t = 60 ns: CI = 0
=> SUM = 0xFF
bei t = 92 ns



Beim Ripple-Carry-Addierer werden die Carry-Ausgänge mit den Carry-Eingängen der jeweils nächsten Stufe verbunden. Dadurch entsteht eine Übertragskette (engl. carry chain). Die worst-case-Verzögerungszeit des Ripple-Carry-Addierers nimmt mit jedem zusätzlichen Operandenbit zu.

Prinzip des Carry-Lookahead-Addierers

- Lange Laufzeiten in den Carry-Signalketten werden aufgebrochen.
- Alle Carry-Signale werden gleichzeitig berechnet.
- Beim Volladdierer werden die CG(Carry-Generate)- und CP(Carry-Propagate)-Signale als Signal P_i bzw. G_i nach außen geführt.
- Die Bildung des Carry-Signals wird rekursiv betrachtet:

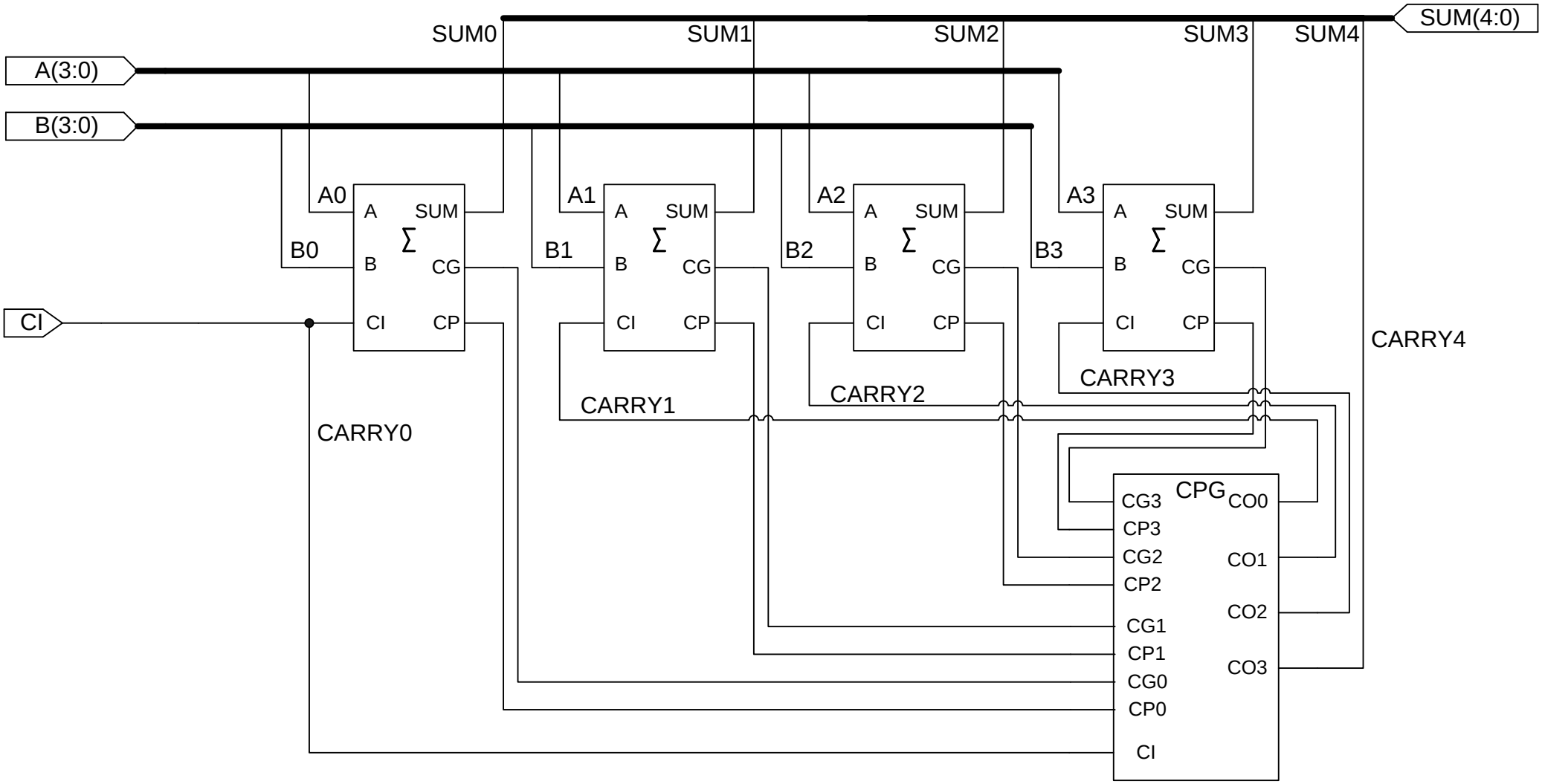
$$C_i = (C_{i-1} \cdot P_i) + G_i$$

- In der ersten Stufe (i=0) wird: $C_0 = (C_{-1} \cdot P_0) + G_0$

- in der zweiten Stufe (i=1): $C_1 = (C_0 \cdot P_1) + G_1 = G_1 + P_1 G_0 + P_1 P_0 C_{-1}$

- Alle P_i- und G_i-Signale werden im Carry-Lookahead-Generator CPG gleichzeitig zu C_i-Signalen ausgewertet und den einzelnen Volladdierern zur Verfügung gestellt.

4-Bit-Carry-Lookahead-Addierer



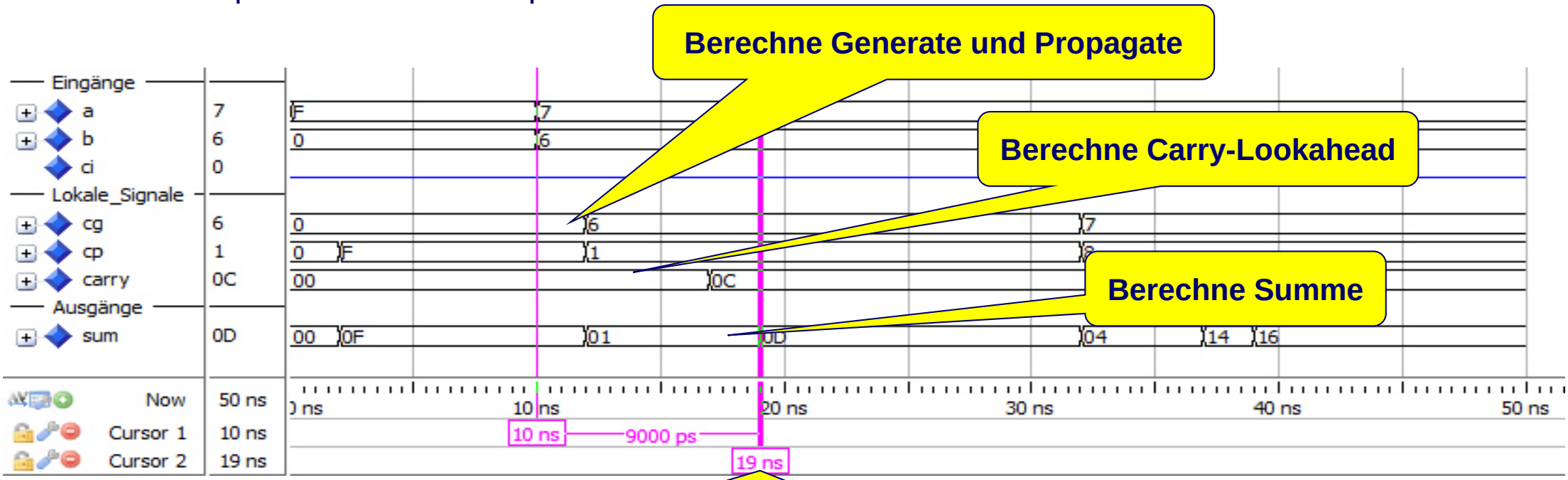
VHDL-Modell eines 4-Bit-Carry-Lookahead-Generators

```
-- 4-Bit Carry-Lookahead-Generator
entity CPG is
        port( CG, CP: in bit_vector(3 downto 0);
              CI: in bit;
              CO: out bit_vector(3 downto 0)
            );
end CPG;
architecture VERHALTEN of CPG is
begin
        CO(0) <= CG(0) or (CP(0) and CI) after 5 ns;
        CO(1) <= CG(1) or (CP(1) and CG(0)) or
                 (CP(1) and CP(0) and CI) after 5 ns;
        CO(2) <= CG(2) or (CP(2) and CG(1)) or
                  (CP(2) and CP(1) and CG(0)) or
                  (CP(2) and CP(1) and CP(0) and CI) after 5 ns;
        CO(3) <= CG(3) or (CP(3) and CG(2)) or
                  (CP(3) and CP(2) and CG(1)) or
                  (CP(3) and CP(2) and CP(1) and CG(0)) or
                  (CP(3) and CP(2) and CP(1) and CP(0) and CI) after 5 ns;
end VERHALTEN;
```

Die Breite der Produktterme nimmt mit zunehmender Bitbreite zu und erhöht damit die Signallaufzeit. Dies begrenzt den sinnvollen Einsatz von Carry-Lookahead-Addierern.

Simulation einer 4-Bit-Carry-Lookahead-Struktur

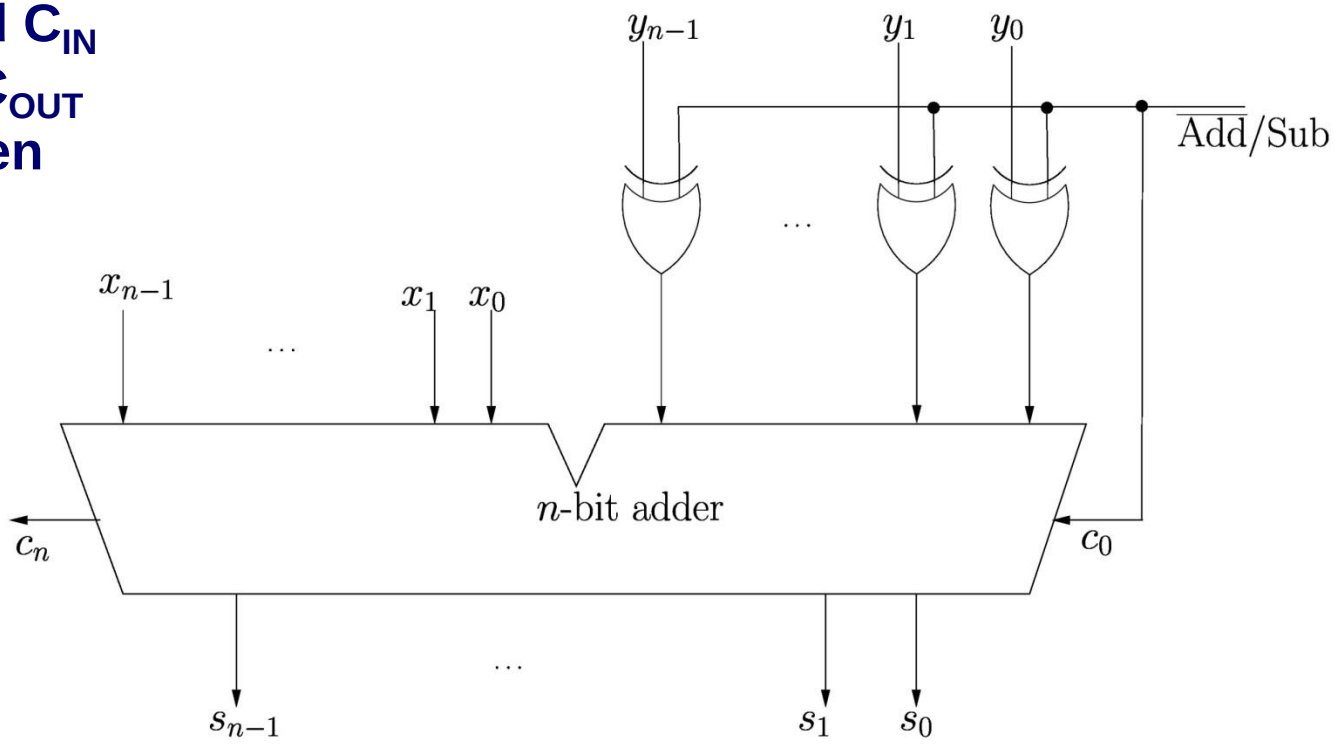
- Die Verzögerungszeit durch den Carry-Lookahead-Addierer ist unabhängig von der Anzahl der zu addierenden Bits.
- Im worst-case-Fall muss das Signal drei Stufen durchlaufen.
- Beispiel: t_{pd}(Volladd) = 2 ns, t_{pd}(Carry-Lookahead) = 5 ns



Die Verzögerungszeit beim Wechsel der Eingangssignale zum Summationsausgang beträgt im worst-case-Fall nur 9 ns.

Kombinierter Addierer / Subtrahierer

- **Einfacher Aufbau für Zweierkomplementarithmetik.**
- **Verwende das Steuersignal** $\overline{\text{ADD}}$ / SUB :
- **Falls das Steuersignal 0 ist, so wird addiert, andernfalls subtrahiert.**
- **Bei der Addition bedeutet CARRY = 1 einen Übertrag, bei der Subtraktion bedeutet CARRY = 0 einen Übertrag (Borrow).**
- **Erstelle eine Wahrheitstabelle für die Eingangssignale A, B und C_{IN} sowie die Ausgangssignale C_{OUT} und bestimme die zusätzlichen Logikfunktionen.**



Addition von Festkommazahlen im Q-Format

Beispiel: Addition einer s3Q12-Zahl mit einer s1Q14-Zahl zu einem s4Q11-Ergebnis:

- Verwende die gleichen Hardwareaddierer wie bisher (parametrisierter 16 Bit-Addierer).
- Beim B-Operanden müssen die führenden Bits *vorzeichengerecht ergänzt* werden.
- Beim B-Operanden werden die letzten beiden Bitstellen abgeschnitten. Dies bedeutet ein *Quantisierungsrauschen*.

```
entity FIX_POINT_ADD is
port( A : in bit_vector(15 downto 0);       -- s3Q12 Format
      B : in bit_vector(15 downto 0);       -- s1Q14 Format
      RESULT : out bit_vector(15 downto 0) -- s4Q11 Format
);
end FIX_POINT_ADD;
architecture A of FIX_POINT_ADD is
component N_BIT_ADD is
   generic( N: integer:=8);
        port (A, B :in bit_vector(N-1 downto 0);
              CI :in bit;
              SUM:out bit_vector(N downto 0));
end component;
constant CI: bit :='0';
signal OPA, OPB: bit_vector(15 downto 0);       -- 16 Bit
signal TEMP_RES: bit_vector(16 downto 0);
begin
    OPA <= A;
    OPB <= B(15)&B(15)&B(15 downto 2);          -- 2 + 14 Bit
ADD: N_BIT_ADD
    generic map(N=>16)
    port map( A=>OPA, B=>OPB, CI=>CI, SUM=>TEMP_RES);
    RESULT <=TEMP_RES(16 downto 1);
end A;
```

Auf welche Weise ist das Ergebnis Q-Format definiert?

Anpassung der Kommastelle

Parametrisierung auf 16 Bit !

Arithmetik in VHDL (1)

- **Die Verwendung Arithmetik Operatoren erfordert entweder den Datentyp `signed` oder `unsigned` und die Einbindung der Bibliothek `ieee.numeric_std`:**

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
...
signal OPA, OPB: signed(17 downto 0;
signal PROD: signed(35 downto 0);
...
PROD <= OPA * OPB;
```

- **Tabelle der synthesefähigen Vergleichsoperatoren:**

Vergleichsoperator	Bedeutung	Beispiel
=	gleich	... when A = B ...
/=	ungleich	... when A /= B ...
<	kleiner	... when A < B ...
<=	kleiner oder gleich	... when A <= B ...
>	größer	... when A > B ...
>=	größer oder gleich	... when A >= B ...

Arithmetik in VHDL (2)

- **Tabelle der synthesefähigen Arithmetikoperatoren:**

Operator	Bedeutung	Beispiel	Synthesefähigkeit
+	Addition	Y <= A + B	synthesefähig
-	Subtraktion	Y <= A - B	synthesefähig
abs	Absolutwertbildung	Y <= abs(A)	synthesefähig
*	Multiplikation	Y <= A * B	synthesefähig
/	Division	Y <= A / B	meist nicht synthesefähig
**	Potenzbildung	Y <= 2**A	nur Potenzen von 2 erlaubt (Links-Schieben)
mod	Rest der Division A/B Das Vorzeichen des Ergebnisses ist gleich dem von B.	Y <= A mod B	synthesefähig falls B Zweierpotenz
rem	Rest der Division A/B. Das Vorzeichen des Ergebnisses ist gleich dem von A.	Y <= A rem B	synthesefähig falls B Zweierpotenz

Arithmetik in VHDL (3)

- **Tabelle der synthesefähigen Arithmetikoperatoren (Forts.):**

Operator	Bedeutung	Beispiel	Synthesefähigkeit
shift_left()	Links schieben um N Bit	Y <= shift_left(A, 3)	synthesefähig; die höchstwertigen Bits gehen verloren, rechts wird mit Nullen aufgefüllt
shift_right()	Rechts schieben um N Bit	Y <= shift_right(A, 3)	synthesefähig; die niederwertigen Bits gehen verloren, links wird vorzeichengerecht ergänzt
rotate_left()	Links rotieren um N Bit	Y <= rotate_left(A, 2)	synthesefähig; die links heraus geschobenen Bits werden rechts hinein geschoben
rotate_right()	Rechts rotieren um N Bit	Y <= rotate_right(A, 2)	synthesefähig;die rechts heraus geschobenen Bits werden links hinein geschoben

Arithmetik in VHDL (4)

- **Wichtige Konversionsfunktionen der Bibliothek `ieee.numeric_std`:**

Konversionsfunktion	ARG1	ARG2	Ergebnistyp
`to_integer(ARG1)`	unsigned signed	- -	integer integer
`unsigned(ARG1)`	signed std_logic_vector	- -	unsigned unsigned
`to_unsigned(ARG1,ARG2)`	natural	Anzahl der unsigned Bits	unsigned
`signed(ARG1)`	unsigned std_logic_vector	- -	signed signed
`to_signed(ARG1,ARG2)`	integer	Anzahl der signed Bits	signed
`resize(ARG1,ARG2)`	signed unsigned	Anzahl der Bits des Ergebnisses	signed unsigned

Der Datentyp `integer` (1)

- Kein Zugriff auf einzelne Bits einer `integer`-Zahl möglich.
- Unterstützt keinen automatischen Zahlenbereichsüberlauf, so wie er in HW existiert.
- `integer` (vorzeichenbehaftete) oder `natural` (vorzeichenlose) Zahlen verwenden ohne weitere Einschränkung des Zahlenbereichs immer 32-Bit-Zahlen => verwende eine `subtype`-Deklaration:

```
subtype INT_4BIT is integer range -8 to 7;
signal A,B,SUM : INT_4BIT;
begin
  …
  SUM <= A + B;
  …
```

Der Datentyp `integer` (2)

- **Operationen über Integer:**
  - **+** binary oder unary
  - **-** binary oder unary
  - ***** Multiplikation
  - **/** Division
  - **mod** Modulo
  - **rem** Remainder
  - **abs** Absolutwert
  - ****** Exponent

Der Datentyp `integer` (3)

- **Indizierter Zugriff auf einzelne Bits eines Signalvektors mit dem Datentyp `integer`**

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;              -- Erforderlich für indizierten Zugriff
entity IND_ZUGRIFF is
    port( ZAHL: in signed(15 downto 0);
          BITADDR: in std_logic_vector(3 downto 0);
          BITWERT: out std_logic
        );
end IND_ZUGRIFF;
architecture A of IND_ZUGRIFF is
begin
    BITWERT <= ZAHL(to_integer(unsigned(BITADDR)));
end A;
```

- **Weitere Anwendungen:**
  - **Multiplexer, Demultiplexer**
  - **RAM- und ROM-Speicher**

Beachte die beiden erforderlichen Datentypkonversionen!

Datentypen in VHDL