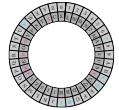




Hochschule **RheinMain**  
University of Applied Sciences  
Wiesbaden Rüsselsheim



# HARDWARE- BESCHREIBUNGSSPRACHEN

## Hardwareentwurf mit VHDL

9. November 2020

Revision: 0d5ed06 (2020-11-09 20:24:57 +0100)

Steffen Reith

Theoretische Informatik  
Studienbereich Angewandte Informatik  
Hochschule **RheinMain**



# DIE GENERIERUNG VON SCHALTKREISEN MIT VHDL

# ALGORITHMISCHE ERZEUGUNG VON HARDWARE

Oft kann **Hardware** durch **iterative oder rekursive Algorithmen beschrieben** werden, da diese eine sehr einfache und **reguläre Struktur besitzen**. Ein gutes **Beispiel** sind **Speicherbausteine**, die durch eine (rechteckige) Matrix von Bits gebildet werden.

In einem solchen Prozess werden einfache Grundbausteine dazu verwendet komplexe Bausteine zu erzeugen, d.h. man **beschreibt nur die Grundbausteine** und das **Verfahren** aus ihnen den eigentlichen Baustein zu erzeugen.

Dieses Vorgehen verspricht ein **kompaktes und platzsparendes Layout (Preis)** und einen **vereinfachten Entwurf**. Aus diesen Gründen sind die notwendigen Mechanismen in VHDL schon eingebaut:

# DIE ITERATIVE ERZEUGUNG VON STRUKTUREN

Zur iterativen Erzeugung von Hardware bietet VHDL eine spezielle **for**-Schleife mit der folgenden Syntax:

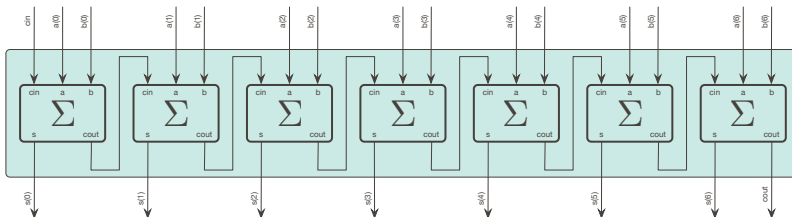
```
1 generate_label : for identifier in discrete_range generate
2   block_declarative_item
3 begin
4   concurrent_statement
5 end;
6 end generate [generate_label];
```

Der Identifier nimmt dabei den **Typ des Ranges an** und durchläuft nacheinander alle möglichen Werte. Im Schleifenkörper verhält sich der **Identifier wie eine Konstante**.

Achtung: Das Label vor dem Schlüsselwort **for** ist **verpflichtend**!

# BEISPIEL: RIPPLE-CARRY ADDER

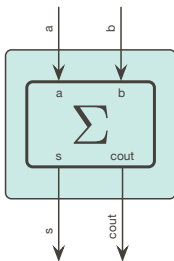
Nun soll ein Ripple-Carry Adder für  $n$ -Bit Zahlen mit VHDL beschrieben werden. Die generelle Struktur für  $n = 7$  sieht wie folgt aus:



Die einzelnen Ziffern der Binärzahl werden mit Hilfe von Volladdierern verarbeitet, die wieder aus zwei Halbaddierern bestehen.

# HALBADDIERER

Ein Halbaddierer **addiert zwei Bits** und **erzeugt** die **Summe** und einen **eventuellen Überlauf**.



a	b	s	cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Wahrheitswertetabelle eines Halbaddierers.

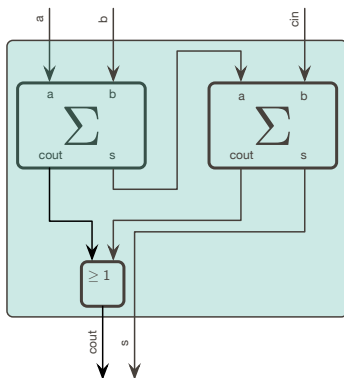
Eine **Kaskadierung** ist **nicht möglich**, da der Halbaddierer kein Carry-Bit als Eingang besitzt.

## EIN HALBADDIERER IN VHDL

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity HalfAdder is
5
6  port (a      : in  std_logic;
7        b      : in  std_logic;
8        s      : out std_logic;
9        cout   : out std_logic);
10
11 end HalfAdder;
12
13 architecture Behavioral of HalfAdder is
14 begin
15
16     -- Calculate the sum
17     s <= a xor b;
18
19     -- Calculate the overflow
20     cout <= a and b;
21
22 end architecture;
```

# VOLLADDIERER

Verwendet man zwei Halbaddierer, so gewinnt man einen **Volladdierer**, der einen **zusätzlichen Carry-Eingang** besitzt.



a	b	cin	s	cout
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

Wahrheitstabelle eines Volladdierers.



## EIN VOLLADDIERER IN VHDL

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity FullAdder is
5
6      port (a      : in  std_logic;
7            b      : in  std_logic;
8            cin    : in  std_logic;
9            s      : out std_logic;
10           cout   : out std_logic);
11
12  end FullAdder;
13
14  architecture Structural of FullAdder is
15
16      signal sum1      : std_logic;
17      signal carry1    : std_logic;
18      signal carry2    : std_logic;
```

## EIN VOLLADDIERER IN VHDL (II)

```
1  begin
2
3      adder1 : entity work.HalfAdder(Behavioral)
4      port map (a      => a,
5                  b      => b,
6                  s      => sum1,
7                  cout   => carry1);
8
9      adder2 : entity work.HalfAdder(Behavioral)
10     port map (a      => sum1,
11                b      => cin,
12                s      => s,
13                cout   => carry2);
14
15     -- Generate full-adder carry
16     cout <= carry1 or carry2;
17
18 end architecture;
```

# EIN GENERISCHER RIPPLE-CARRY ADDIERER

Nun wird ein generischer Ripple-Carry Addierer in VHDL formuliert, wobei die Bitbreite durch einen generischen Parameter frei gewählt werden kann:

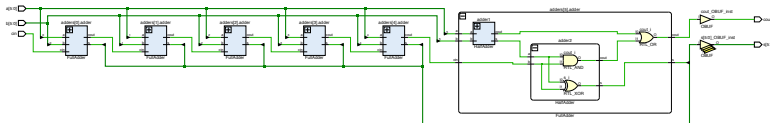
```
1
2  library ieee;
3  use ieee.std_logic_1164.all;
4
5  entity RCAdder is
6
7      generic (width : integer := 6);
8      port (a      : in  std_logic_vector(width - 1 downto 0);
9            b      : in  std_logic_vector(width - 1 downto 0);
10           cin    : in  std_logic;
11           s      : out std_logic_vector(width - 1 downto 0);
12           cout   : out std_logic);
13
14  end RCAdder;
```

## EIN GENERISCHER RIPPLE-CARRY ADDIERER (II)

```
1  architecture Behavioral of RCAdder is
2  signal carry : std_logic_vector(width downto 0);
3  begin
4
5  -- Generate an array of full adders
6  adders : for I in 0 to width - 1 generate
7  begin
8      -- Generate the Ith full adder
9      adder : entity work.FullAdder(Structural)
10     port map (a      => a(I),
11               b      => b(I),
12               cin    => carry(I),
13               s      => s(I),
14               cout   => carry(I+1));
15  end generate;
16
17  carry(0) <= cin;
18  cout <= carry(carry'high);
19
20 end architecture;
```

# DAS SYNTHESEERGEBNIS

Vivado 2015.4 liefert das folgende Syntheseergebnis:



Der **Pfad von cin nach cout** wächst direkt mit der Bitbreite  $n$  des Addierers, d.h. die Tiefe des Schaltkreises ist  $O(n)$ . Damit ist die **mögliche Taktrate** bei **großen Bitlängen gering**.

# DIE BEDINGTE ERZEUGUNG VON HARDWARESTRUKTUREN

Mit VHDL kann man die **Erzeugung von Strukturen** auch von **beliebigen Bedingungen abhängig** machen. Dazu dient die folgende Anweisung:

```
1 generate_label : if condition generate
2     generate_statement_body
3 elsif condition generate
4     generate_statement_body
5 else generate
6     generate_statement_body
7 end generate [generate_label];
```

Dies kann man auch dazu verwenden Hardwarestrukturen **rekursiv** zu erzeugen. Dazu beschreibt man eine **generische Komponente** und benutzt das **if-generate** Konstrukt dazu den **Rekursionsabbruch** und den **Rekursionsschritt** zu spezifizieren.

## BEISPIEL: EIN GENERISCHER PRIORITY-ENCODER

Ziel ist der Entwurf eines Schaltkreises, der zu einem **gegebenen Bitstring**  $b = b_1 b_2 \dots b_{2^n}$  den **Index**  $i$  berechnet, so dass  $b_i = 1$  und  $b_j = 0$  für alle  $j > i$  („1-Bit mit dem größten Index“).

Gleichzeitig soll der Schaltkreis noch ausgeben, ob **überhaupt** ein 1-Bit mit String  $b$  enthalten ist. Eine mögliche Anwendung eines solchen Schaltkreises wäre der Bau eines **Interruptcontrollers**.

Für die **zweite Aufgabenstellung ergibt sich die Funktion**

**HasOne**:  $\{0, 1\}^{2^n} \rightarrow \{0, 1\}, n \geq 1$ :

$$\text{HasOne}(b_1 \dots b_{2^n}) = \begin{cases} b_0 \text{ or } b_1, & \text{falls } n = 1 \\ \text{HasOne}(b_{2^n-1} \dots b_{2^{n-1}}) \text{ or} \\ \text{HasOne}(b_{2^{n-1}-1} \dots b_0) & \text{sonst} \end{cases}$$

Dies ist offensichtlich die rekursive Definition einer **or-Funktion mit  $2^n$  Argumenten**.

## BEISPIEL: EIN GENERISCHER PRIORITY-ENCODER (II)

Sei  $b = b_{2^n-1}b_{2^n-2} \dots b_1b_0 \in \{0, 1\}^{2^n}$ , dann bezeichnet  
 $ub = b_{2^n-1} \dots b_{2^{n-1}}$  („upper-half“) und  $lb = b_{2^{n-1}-1} \dots b_0$   
 („lower-half“).

Der Priority-Encoder kann durch die Funktion  
 $\text{PEnc}: \{0, 1\}^{2^n} \rightarrow \{0, 1\}^n, n \geq 1$  rekursiv wie folgt beschrieben  
 werden:

$$\text{PriEnc}(b_1 \dots b_{2^n}) = \begin{cases} b_1 & , \text{ falls } n = 1 \\ 1 \text{ PriEnc}(ub) & , \text{ falls HasOne}(ub) = 1 \\ 0 \text{ PriEnc}(lb) & , \text{ sonst} \end{cases}$$



## VHDL-BESCHREIBUNG DES PRIORITY-ENCODERS

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity PriEnc is
5
6  generic (idxWidth : integer := 5);
7  port (src      : in std_logic_vector(2**idxWidth - 1 downto 0);
8        idx      : out std_logic_vector(idxWidth - 1 downto 0);
9        active   : out std_logic);
10
11 end PriEnc;
12
13 architecture Recursive of PriEnc is
14 begin
15
16 abortRec : if (idxWidth = 1) generate
17
18     active <= src(0) or src(1);
19     idx(0) <= src(1);
20
21 end generate abortRec;
```

## VHDL-BESCHREIBUNG DES PRIORITY-ENCODERS (II)

```

1  stepRec : if (idxWidth > 1) generate
2
3      signal lActive  : std_logic;
4      signal rActive  : std_logic;
5      signal leftIdx  : std_logic_vector(idxWidth - 2 downto 0);
6      signal rightIdx : std_logic_vector(idxWidth - 2 downto 0);
7
8  begin
9
10     -- Generate a priority encoder for the upperhalf of src
11     leftEnc : entity work.PriEnc(Recursive)
12     generic map (idxWidth => idxWidth - 1)
13     port map (src      => src(2**idxWidth - 1 downto 2** (idxWidth - 1)),
14              idx       => leftIdx,
15              active    => lActive);
16
17     -- Generate a priority encoder for the lower half of src
18     rightEnc : entity work.PriEnc(Recursive)
19     generic map (idxWidth => idxWidth - 1)
20     port map (src      => src(2** (idxWidth - 1) - 1 downto 0),
21              idx       => rightIdx,
22              active    => rActive);

```

## VHDL-BESCHREIBUNG DES PRIORITY-ENCODERS (III)

```

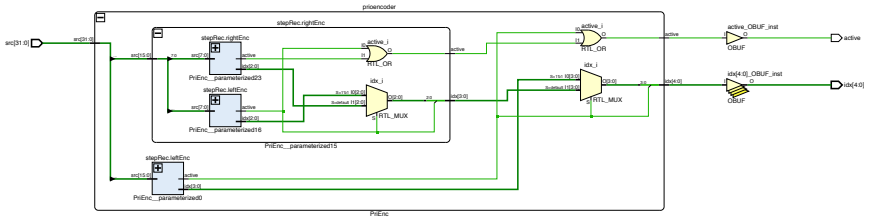
1      -- Calculate if at least one bit is set
2      active <= lActive or rActive;
3
4      -- Set MSB if upper half contains a '1'
5      idx(idx'high) <= '1' when (lActive = '1')
6                      else '0';
7
8      -- Set the lower index bits
9      idx(idx'high - 1 downto 0) <= leftIdx when (lActive = '1')
10                                     else rightIdx;
11
12      end generate stepRec;
13
14      end architecture;

```

Bei dieser Konstruktion läuft die **Rekursion** nur über die **Breite der Ausgabe** und wird abgebrochen, wenn die Eingabe zwei Bit (bzw. Ausgabe ein Bit) breit ist.

# DAS ERGEBNIS DER SYNTHESE

Vivado 2015.4 liefert das folgende Syntheseresultat:



Wenn nötig, können auch **for-generate** und **if-generate** **gemischt** werden, um noch kompliziertere Strukturen zu erzeugen.

# EINE BEMERKUNG ZU HASONE

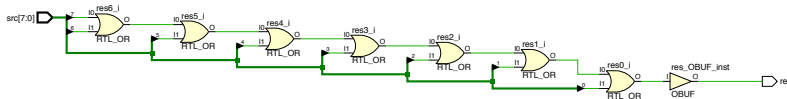
HasOne kann auch mit Schleifen modelliert werden:

```
1  entity ORn is
2  generic(width : natural := 8);
3  port(src : in std_logic_vector(width - 1 downto 0);
4        res : out std_logic);
5  end entity;
6  architecture Behavioral of ORn is
7  begin
8
9      genOr : process (src)
10         variable tmp : std_logic;
11         begin
12
13             tmp := '0';
14             for i in src'range loop
15                 tmp := tmp or src(i);
16             end loop;
17
18             res <= tmp;
19
20         end process;
21 end architecture;
```

# EINE BEMERKUNG ZU HASONE (II)

Manche Synthesetools optimieren aufgrund der **Assoziativität** von `or` die Tiefe des Schaltkreises und bauen die Schaltung als **binären Baum** auf.

Vivado 2015.4 liefert in der Grundeinstellung:



Bei der Verwendung von `generate` ist man von solchem Verhalten **unabhängig** und kann die Struktur der Hardware genau bestimmen.