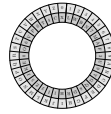




Hochschule **RheinMain**  
University of Applied Sciences  
Wiesbaden Rüsselsheim



# HARDWARE / SOFTWARE- SCHNITTSTELLEN

Hardwareentwurf mit VHDL

23. Juni 2014  
(Revision: 1341)

Prof. Dr. Steffen Reith

Theoretische Informatik  
Studienbereich Angewandte Informatik  
Hochschule **RheinMain**



Notizen

---

---

---

---

---

---

---

---

---

---

---

---

Notizen

---

---

---

---

---

---

---

---

---

---

---

---

GRAY-CODES

## GRUNDLAGEN

Digitale Schaltkreise verbrauchen durch jedem **Zustandswechsel** Energie (neben dem Energieverbrauch durch Leckströme).

Beispiel: Ein Binärzähler mit 3 Bit führt 11 Zustandswechsel durch.

Frage: Wie viele Zustandswechsel führt ein  $n$  Bit Binärzähler durch, wenn er von 0 bis  $2^n - 1$  zählt?

**Summiert** man die Zustandswechsel in den **Spalten** der Wahrheitwertetabelle auf, so ergeben sich:

$$\begin{aligned} \sum_{i=1}^n (2^i - 1) &= \sum_{i=1}^n 2^i - n \\ &= \sum_{i=0}^n 2^i - n - 1 \\ &= 2^{n+1} - (n + 2) \end{aligned}$$

Zustandswechsel.

112

Notizen

---

---

---

---

---

---

---

---

---

---

## GRUNDLAGEN (II)

Die Wahrheitwertetabelle eines 3 Bit Zählers hat  $2^3 = 8$  Zeilen, d.h. dieser führt durchschnittlich  $11/8 = 1.375$  Wechsel pro Schritt durch.

Für einen 10 Bit Zähler ergeben sich durchschnittlich 1.98828 und für einen 32 Bit Zähler 1.99999992 Zustandswechsel pro Schritt.

Grob kann man sagen, dass **pro Zählschritt zwei Zustandswechsel** durchgeführt werden müssen, da

$$\lim_{n \rightarrow \infty} \frac{2^{n+1} - (n + 2)}{2^n} = 2$$

gilt.

Geht das auch besser?

113

Notizen

---

---

---

---

---

---

---

---

---

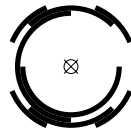
---

## GRAY-CODES

Kodiert man die Zahlen von 0 bis 7 durch

000, 001, 011, 010, 110, 111, 101, 100

so findet nur ein Zustandwechsel pro Schritt statt.



Diese Codierung wurde durch **Frank Gray** 1934 eingeführt, war aber Émile Baudot (vgl. **Baud**) schon 1878 bekannt. Implizit wurde dieser Code von Louis Gros 1872 bei der Untersuchung von Knobelspielen (chinesisches Ringpuzzle) eingeführt.

114

Notizen

---

---

---

---

---

---

---

---

---

---

---

---

## ANWENDUNGEN

Eine Anwendung finden Gray-Codes bei

- Inkrementgebern.
- Bestimmung des Drehwinkels

Verwendet ein Inkrementgeber die Binärcodierung, so **ändern** aufgrund von Laufzeitunterschieden **nicht** alle Bits **gleichzeitig den Wert**.

Es entstehen unerwünschte Zwischenwerte (vgl. Glitches)

Bei Drehwinkelgebern bewirken **kleine** Winkelfehler auch nur kleine Änderungen im Codewort. Dies gleich Fertigungstoleranzen im optischen Sensor aus.

115

Notizen

---

---

---

---

---

---

---

---

---

---

---

---

## EINIGE DEFINITIONEN

### Definition

Sei  $\Gamma$  ein beliebiges Alphabet, dann bezeichnet  $\Gamma_n$  eine **Folge von Strings der Länge  $n$**  bzgl. einer festgelegten Ordnung.

### Definition

Sei  $a \in \Gamma$  und  $\Gamma_n = (w_1, w_2, \dots, w_m)$ , dann ist

$$a\Gamma_n =_{\text{def}} (aw_1, aw_2, \dots, aw_m).$$

eine Folge von Strings der Länge  $n + 1$ . Mit  $\Gamma_n^R$  bezeichnen wir die **reflektierte** Folge  $(w_m, \dots, w_2, w_1)$  und  $\circ$  symbolisiert die **Konkatenation** von Folgen, d.h.  $(v_1, v_2, \dots, v_n) \circ (w_1, w_2, \dots, w_m) = (v_1, v_2, \dots, v_n, w_1, w_2, \dots, w_m)$ .

Notizen

---

---

---

---

---

---

---

---

---

---

---

---

## BINÄRER GRAY-CODE

Ein (binärer) Graycode  $\Gamma_n$  der Länge  $n$  kann nun leicht induktiv definiert werden:

### Definition

Sei  $\Gamma = \{0, 1\}$ , dann

**(IA)**  $\Gamma_0 = (\epsilon)$ , wobei  $\epsilon$  das leere Wort ist

**(IS)**  $\Gamma_{n+1} = 0\Gamma_n \circ 1\Gamma_n^R$

Diese Definition funktioniert ähnlich für beliebige Alphabete, wir beschränken uns auf den binären Fall.

### Fakt

Sei  $\Gamma = \{0, 1\}$ , dann  $\Gamma_n^R = \Gamma_n \oplus 10^{n-1}$ .

Notizen

---

---

---

---

---

---

---

---

---

---

---

---



## EIGENSCHAFTEN DES GRAY-CODES

Eine einfache Induktion über die induktive Struktur von  $\Gamma_n$  ergibt:

## Theorem

Sei  $i \in \mathbb{N}$ , dann ist die **Hamingdistanz** zwischen  $g(i)$  und  $g(i + 1)$  **genau 1**.

## Lemma

Sei  $k = 2^n + r$  mit  $0 \leq r < 2^n$ , dann gilt

$$g(k) = 2^n + g(2^n - 1 - r).$$

Dies ergibt sich, da  $g(k)$  ein Codewort aus dem Block  $1\Gamma_n$  ist. Der Summand  $2^n$  entspricht der führenden 1 und  $2^n - 1 - r$  entspricht dem reflektierten Wert von  $r$ .

120

Notizen

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

Eine Induktion über  $n$  zeigt dann:

## Theorem

Sei  $k \in \mathbb{N}$  mit der binären Repräsentation  $(\dots b_2 b_1 b_0)_2$ , dann ist  $g(k) = (\dots a_2 a_1 a_0)_2$ , wobei

$$a_j = b_j \oplus b_{j+1}, j \geq 0.$$

Dies führt direkt zur einer Methode eine Zahl in Binärdarstellung in Gray-Code umzuwandeln:

## Theorem

Sei  $b = (b_n \dots b_1 b_0)_2$ , dann gilt

$$(b_n \dots b_1 b_0)_2 \oplus (0b_n \dots b_2 b_1) = g(k).$$

Also gilt  $g(k) = b \oplus b/2$ .

121

Notizen

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

## EINE VHDL-IMPLEMENTIERUNG

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity GrayCnt is
6
7      generic(width : integer := 5);
8
9      port (clk      : in  std_logic;
10         reset   : in  std_logic;
11         enable  : in  std_logic;
12         gval   : out std_logic_vector(width - 1 downto 0));
13
14 end GrayCnt;
15
16 architecture BinConv of GrayCnt is
17     signal val_reg  : std_logic_vector(width - 1 downto 0);
18     signal val_next : std_logic_vector(width - 1 downto 0);
19 begin

```

122

Notizen

---



---



---



---



---



---



---



---



---



---

## EINE VHDL-IMPLEMENTIERUNG (II)

```

1  state_logic : process (clk, reset)
2  begin
3      if (rising_edge(clk)) then
4
5          -- Synchron reset
6          if(reset = '1') then
7
8              -- Set binary counter to the predecessor of 0
9              val_reg <= (others => '1');
10
11          else
12
13              -- Check if counter is enabled
14              if (enable = '1') then
15
16                  -- Set value on rising edge
17                  val_reg <= val_next;
18
19              end if;
20          end if;
21      end if;
22  end process;

```

123

Notizen

---



---



---



---



---



---



---



---



---

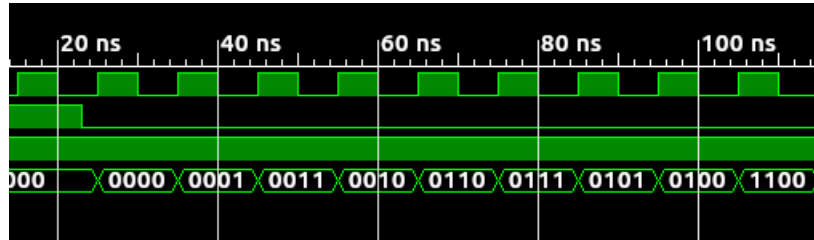


---

# EINE VHDL-IMPLEMENTIERUNG (III)

```

1  -- Next state logic
2  val_next <= std_logic_vector(unsigned(val_reg) + 1);
3
4  -- Convert to Gray-Code as output (note: g(k) = k xor k/2)
5  gval <= val_reg xor ("0" & val_reg(width - 1 downto 1));
6
7  end architecture;
```



Diese Implementierung ist korrekt, aber es wurde nichts gewonnen, da die „Next state logic“ eine Addition verwendet.

Notizen

---

---

---

---

---

---

---

---

---

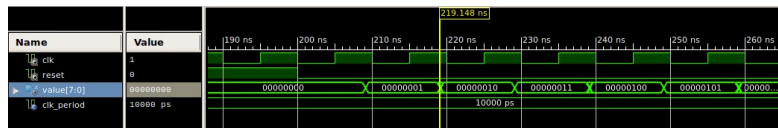
---

---

---

# EINE BEMERKUNG ZUM ENERGIEVERBAUCH

Wir haben ja schon gesehen, dass bei einer Addition unerwünschte Zwischenzustände (Glitches) entstehen, die zu unerwünschtem Energieverbrauch führen:



Der gleiche Simulationslauf mit höherer Zeitauflösung:



Gibt es einen Gray-Counter ohne Addierer?

Notizen

---

---

---

---

---

---

---

---

---

---

---

---







## EIN GRAY-COUNTER OHNE ADDIERER

```

1  architecture Native of GrayCnt is
2    signal hBit_reg  : std_logic; -- Hold the toggling bit
3    signal hBit_next : std_logic;
4
5    signal val_reg   : std_logic_vector(width - 1 downto 0);
6    signal val_next  : std_logic_vector(width - 1 downto 0);
7  begin
8
9    state_logic : process (clk, reset)
10   begin
11     if (rising_edge(clk)) then
12       if(reset = '1') then
13         hBit_reg <= '1';
14         val_reg <= (val_reg'high => '1', others => '0');
15       else
16         if (enable = '1') then
17           hBit_reg <= hBit_next;
18           val_reg <= val_next;
19         end if;
20       end if;
21     end if;
22   end process;

```

130

Notizen

---



---



---



---



---



---



---



---



---



---

## EIN GRAY-COUNTER OHNE ADDIERER(II)

```

1  gval <= val_reg; -- Output logic
2
3  next_state : process(val_reg, hBit_reg)
4    variable tmp : std_logic;
5  begin
6    hBit_next <= not hBit_reg; -- Toggle
7
8    val_next(val_next'low) <=
9      not(val_reg(val_reg'low) xor (hBit_reg)); -- LSB
10
11   -- Teil der Gleichung fuer das MSB
12   tmp := '1';
13   for j in 0 to width - 3 loop
14     -- Teste auf Muster 0^*
15     tmp := tmp and not(val_reg(j));
16   end loop;
17
18   -- Vollstaendige Gleichung fuer das MSB
19   val_next(val_next'high) <=
20     val_reg(val_reg'high) xor (tmp and hBit_reg);

```

131

Notizen

---



---



---



---



---



---



---



---



---



---

## EIN GRAY-COUNTER OHNE ADDIERER(III)

```

1  -- Erzeuge alle inneren Bits
2  for i in 1 to width - 2 loop
3
4      -- Test auf Muster 10^*
5      tmp := '1';
6
7      for j in 0 to i - 2 loop
8          tmp := tmp and not(val_reg(j));
9      end loop;
10
11     tmp := val_reg(i - 1) and tmp and hBit_reg;
12
13     -- Vollständige Gleichung fuer Bit i
14     val_next(i) <= val_reg(i) xor tmp;
15
16 end loop;
17
18 end process;
19
20 end architecture;

```

132

Notizen

---



---



---



---



---



---



---



---



---



---

## VERGLEICH

Obwohl die native Implementierung komplizierter wirkt, benötigt diese signifikant weniger Platz

Device Utilization Summary (estimated values)			
Logic Utilization	Used	Available	Utilization
Number of Slice Registers	9	126800	0%
Number of Slice LUTs	10	63400	0%
Number of fully used LUT-FF pairs	9	10	90%
Number of bonded IOBs	11	210	5%
Number of BUFG/BUFGCTRLs	1	32	3%

als die Variante mit einem binären Zähler:

Device Utilization Summary (estimated values)			
Logic Utilization	Used	Available	Utilization
Number of Slice Registers	8	126800	0%
Number of Slice LUTs	15	63400	0%
Number of fully used LUT-FF pairs	8	15	53%
Number of bonded IOBs	11	210	5%
Number of BUFG/BUFGCTRLs	1	32	3%

133

Notizen

---



---



---



---



---



---



---



---



---



---