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**

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 Wahrheitswertetabelle 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.

GRUNDLAGEN (II)

Die Wahrheitswertetabelle 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 1.999999992 Zustandswechsel pro Schritt.

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

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

gilt.

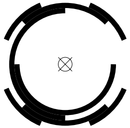
Geht das auch besser?

GRAY-CODES

Kodiert man die Zahlen von 0 bis 7 durch

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

so findet nur ein Zustandswechsel 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.

ANWENDUNGEN

Eine Anwendung finden Gray-Codes bei

- → Inkrementgebern.
- → Bestimmung des Drehwinkels

Verwendet ein Inkrementgeber die Binärkodierung, 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.

Diese Eigenschaft kann auch benutzt werden, um die **Auswirkungen von Laufzeitunterschieden** von Vektoren von Signalen zu **minimieren**.

EINIGE DEFINITIONEN

Definition

Sei Γ ein beliebiges Alphabet, dann bezeichnet Γ_n eine **Folge von Strings der Länge n** bzgl. einer festgelegten Ordnung.

Definition

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

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

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

BINÄRER GRAY-CODE

Ein (binärer) Graycode Γ_n der Länge n kann nun leicht induktiv definiert werden:

Definition

Sei $\Gamma = \{0, 1\}$, dann
(IA) $\Gamma_0 = (\epsilon)$, wobei ϵ 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 $\Gamma_n^R = \Gamma_n \oplus 10^{n-1}$.

BINÄRER GRAY-CODE (II)

Beispiel

Aus der induktiven Definition ergeben sich die folgenden Codewörter:

$$\begin{aligned} n = 1 &\quad : \quad (0, 1) \\ n = 2 &\quad : \quad (00, 01, 11, 10) \\ n = 3 &\quad : \quad (000, 001, 011, 010, 110, 111, 101, 100) \end{aligned}$$

Theorem

*Es gilt $\#(\Gamma_n) = (\#\Gamma)^n$, d.h. im binären Fall gibt es genau 2^n verschiedene Codewörter der Länge n. Der Gray-Code ist also eine **Permutation** der Binärdarstellung.*

BINÄRER GRAY-CODE (III)

Die ersten $2^n/2$ Codewörter in Γ_n entsprechen (mit führender 0) immer den Codewörtern aus Γ_{n-1}. Die Funktion g mit

$$g\colon \mathbb{N} \to \bigcup_{n\geq 1} \Gamma_n$$

mit
$g(0) =_{\text{def}} (0), g(1) =_{\text{def}} (1), g(2) =_{\text{def}} (11), g(3) =_{\text{def}} (110), \dots$
heißt **Graysche Funktion**.

Theorem

Die unendliche Folge $\Gamma_\infty = (g(0), g(1), g(2), g(3), \dots)$ *ist eine Permutation der natürlichen Zahlen.*

EIGENSCHAFTEN DES GRAY-CODES

Eine einfache Induktion über die induktive Struktur von Γ_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.

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

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

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

EINE VHDL-IMPLEMENTIERUNG

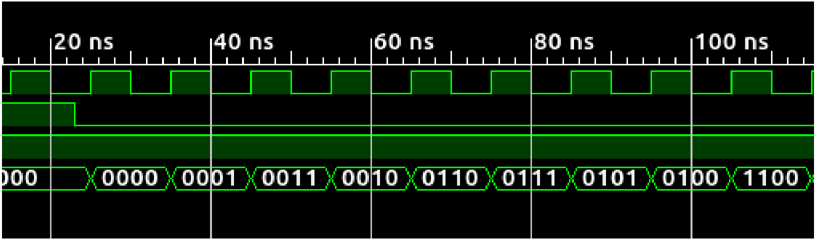
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity GrayCnt is

  generic(width : integer := 5);

  port (clk    : in  std_logic;
        reset  : in  std_logic;
        enable : in  std_logic;
        gval   : out std_logic_vector(width - 1 downto 0));

end GrayCnt;

architecture BinConv of GrayCnt is
  signal val_reg  : std_logic_vector(width - 1 downto 0);
  signal val_next : std_logic_vector(width - 1 downto 0);
begin
```

EINE VHDL-IMPLEMENTIERUNG (II)

```
state_logic : process (clk, reset)
begin
  if (rising_edge(clk)) then

    -- Synchron reset
    if(reset = '1') then

      -- Set binary counter to the predecessor of 0
      val_reg <= (others => '1');

    else

      -- Check if counter is enabled
      if (enable = '1') then

        -- Set value on rising edge
        val_reg <= val_next;

      end if;
    end if;
  end if;
end process;
```

EINE VHDL-IMPLEMENTIERUNG (III)

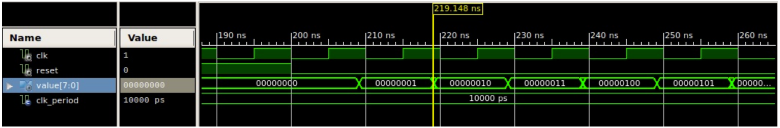
```
    -- Next state logic
    val_next <= std_logic_vector(unsigned(val_reg) + 1);

    -- Convert to Gray-Code as output (note: g(k) = k xor k/2)
    gval <= val_reg xor ("0" & val_reg(width - 1 downto 1));

end architecture;
```



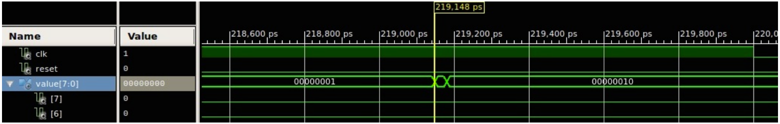
Diese Implementierung ist korrekt, aber es wurde nichts gewonnen, da die „Next state logic“ eine Addition verwendet.

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?

BEISPIEL: EIN 4-BIT GRAY-COUNTER

Wert	Code	`toggle`	Wert	Gray	`toggle`
0	0000	0	8	1100	0
1	0001	1	9	1101	1
2	0011	0	10	1111	0
3	0010	1	11	1110	1
4	0110	0	12	1010	0
5	0111	1	13	1011	1
6	0101	0	14	1001	0
7	0100	1	15	1000	1

Beobachtung

Das LSB des Gray-Counters ändert sich immer, wenn `toggle` von 0 auf 1 springt.

BESCHREIBUNG EINES GRAY-COUNTERS

Mit $Q^{(i)}$ wird ab jetzt ein Bit im Zählschritt i beschrieben.

Fakt

Das LSB Q_0 eines Gray-Counters kann mit Hilfe eine Hilfsbits Q_t Schritt für Schritt durch die Gleichungen

$$\begin{aligned} Q_t^{(0)} &= 0 \\ Q_t^{(i+1)} &= \neg Q_t^{(i)} \\ Q_0^{(0)} &= 0 \\ Q_0^{(i+1)} &= \neg(Q_0^{(i)} \oplus Q_t^{(i)}) \end{aligned}$$

beschrieben werden.

Damit entspricht $Q_t^{(i)}$ der Tabellenspalte `toggle`.

BESCHREIBUNG EINES GRAY-COUNTERS (II)

Beobachtung

Das MSB eines Gray-Counters ändert sich immer, wenn alle Bits außer dem Bit $n-1$ des Codeworts 0 sind.

Das MSB eines n Bits Gray-Counters kann durch die folgenden Gleichungen beschrieben werden:

Fakt

$$
\begin{aligned}
Q_{n-1}^{(0)} &= 0 \\
Q_{n-1}^{(i+1)} &= Q_{n-1}^{(i)} \oplus \left(Q_t^{(i)} \wedge \bigwedge_{j=0}^{n-2} \neg Q_j^{(i)}\right)
\end{aligned}
$$

BESCHREIBUNG EINES GRAY-COUNTERS (III)

Beobachtung

Ein „inneres“ Bits eines Gray-Counters ändert sich immer, wenn die niederwertigen Bits der Codeworts auf den regulären Ausdruck 10* matchen.

Die inneren Bits $Q_k^{(i)}$ mit $1 \leq k \leq n-2$ lassen sich zeitlich wie folgt beschreiben:

Fakt

$$
\begin{aligned}
Q_k^{(0)} &= 0 \\
Q_k^{(i+1)} &= Q_k^{(i)} \oplus (Q_t^{(i)} \wedge Q_{k-1}^{(i)} \wedge \bigwedge_{j=0}^{k-2} \neg Q_j^{(i)})
\end{aligned}
$$

EIN GRAY-COUNTER OHNE ADDIERER

```
architecture Native of GrayCnt is
  signal hBit_reg  : std_logic; -- Hold the toggling bit
  signal hBit_next : std_logic;

  signal val_reg  : std_logic_vector(width - 1 downto 0);
  signal val_next : std_logic_vector(width - 1 downto 0);
begin

  state_logic : process (clk, reset)
  begin
    if (rising_edge(clk)) then
      if(reset = '1') then
        hBit_reg <= '1';
        val_reg <= (val_reg'high => '1', others => '0');
      else
        if (enable = '1') then
          hBit_reg <= hBit_next;
          val_reg <= val_next;
        end if;
      end if;
    end if;
  end process;
```

EIN GRAY-COUNTER OHNE ADDIERER(II)

```
  gval <= val_reg; -- Output logic

  next_state : process(val_reg, hBit_reg)
    variable tmp : std_logic;
  begin
    hBit_next <= not hBit_reg; -- Toggle

    val_next(val_next'low) <=
      not(val_reg(val_reg'low) xor (hBit_reg)); -- LSB

    -- Teil der Gleichung fuer das MSB
    tmp := '1';
    for j in 0 to width - 3 loop
      -- Teste auf Muster 0^*
      tmp := tmp and not(val_reg(j));
    end loop;

    -- Vollstaendige Gleichung fuer das MSB
    val_next(val_next'high) <=
      val_reg(val_reg'high) xor (tmp and hBit_reg);
```

EIN GRAY-COUNTER OHNE ADDIERER(III)

```
      -- Erzeuge alle inneren Bits
      for i in 1 to width - 2 loop

        -- Test auf Muster 10^*
        tmp := '1';

        for j in 0 to i - 2 loop
          tmp := tmp and not(val_reg(j));
        end loop;

        tmp := val_reg(i - 1) and tmp and hBit_reg;

        -- Vollständige Gleichung fuer Bit i
        val_next(i) <= val_reg(i) xor tmp;

      end loop;

    end process;

end architecture;
```

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%