Digital Design

Zusammenfassung

Joel von Rotz & Andreas Ming / Quelldateien

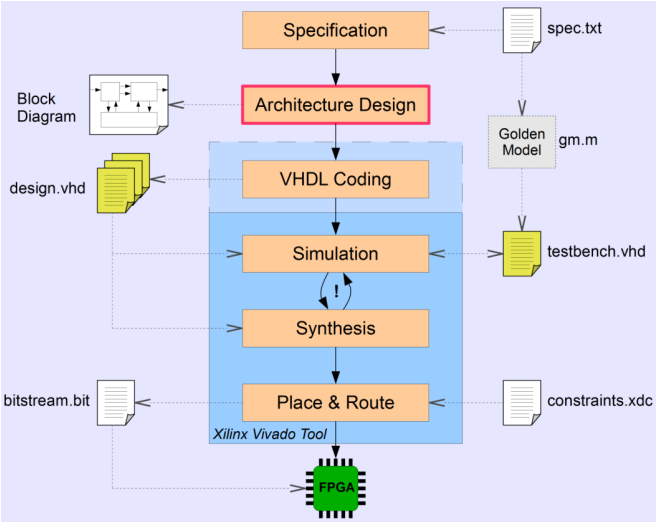
Inhaltsverzeichnis

1. VHDL

> **Hinweis**
>
> **V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage ist einer Hardwarebeschreibung und keine Programmiersprache.

1.1 Entwicklung

1.1.1 Designflow



1.1.2 Struktur Datei

```
-- File: MyComponent.vhd
-- Author: myself
-- Date: yesterday

library ...
-- Library einbinden
use ...
-- Packages aus Library bekanntgeben

entity ...
-- Schnittstelle der Komponente gegen aussen

architecture ...
-- Funktion (Innenleben) der Komponente
```

1.2 Synthesis & Simulation

! Synthesis vs. Implementation

- *Synthesis* generiert die Netlist des VHDL-Codes und beschreibt.
- *Implementation* wendet die Contraints an und sorgt für die Hardware-Implementierung.

1.2.1 Transactions

1.2.2 Propagation Delay

1.3 Architektur

Architecture beschreibt die Implementation oder das Innenleben des Komponents. Darin wird beschrieben, wie die deklarierten Signalen miteinander interagieren.

```
architecture a1 of MyComponent is
  -- Deklarationen (Signale, Komponenten)
  signal tmp : std_logic;
begin
  -- Implementierung
  tmp  <= a_pi or b_pi;
  c_po <= tmp;
end a1;
```

- Der **Deklarationsteil** startet vor dem `begin`
- Der **Implementierungsteil** startet nach `begin` und endet vor `end`

i `rtl` & `struct`

Der Name `rtl` wird verwendet, um grundlegende Logik-Komponenten zu definieren, wie zum Beispiel *OR*, *XOR*, *AND*, etc. `struct` beinhaltet eine Kombination/Anwendung von `rtl`-Komponenten.

1.4 Entity

Eine Entity beschreibt den Komponenten für äusserliche Zugriffe. Es wird nur die Struktur des Komponents bekannt gegeben, aber nicht den Inhalt des Komponenten.

```
entity MyComponent is
  port ( a_pi, b_pi : in std_logic;
         -- Input Ports
         c_po : out std_logic
         -- Output Port
         --x_pio : inout std_logic
         -- Bidirektionaler Port
       );
  constant c_max_cnt : integer := 20_000;
end MyComponent;
```

i Hinweis

Alles was in der Entity bekannt ist (inkl. Libraries), ist auch in der zugehörigen Architecture bekannt.

1.5 Components

1.6 Kombinatorische Logik

Folgend sind *Process Statements* in Kurzschreibweise - Concurrent Signal Assignments - Selected Signal Assignment - Conditional Signal Assignment

! Process Statements

Alle Signal Assignments ausserhalb von `process` (Concurrent-, Selected-, Conditional-Signal Assignment) sind **Process Statements** in Kurzschreibform!

```
sig <= not sig; -- Process Statement
stud <= happy     when mep >= C else
        satisfied when mep >= E else
        sad;    -- Process Statement
```

1.6.1 Concurrent Signal Assignments

1.6.2 Selected Signal Assignments `case`

1.6.3 Conditional Signal Assignments `when/else`

1.7 Prozesse/Sequential Statements

```
-- process sensitivity list
P1: process (i1, i2, i3)

-- local variable (only known in P1)
variable v_tmp : std_logic;
begin
  v_tmp := '0';
  if i1 = '1' and i2 = '0' then v_tmp := '1'; end if;
  o1 <= v_tmp and i3;
-- process P1 drives signal o1
  o2 <= v_tmp xor i3;
-- process P1 drives signal o2
end process P1;
```

1.7.1 Sensitivity List

Prozesse werden mit Hilfe einer *Sensitivity List* auf ausgewählte Signale sensitiv gemacht.

1.8 Grundlegende Konzepte

1.8.1 Ports & Signale

Port sind die Anschlüsse eines Komponents und Signale sind Komponent-interne Signale, welche von aussen nicht zugreifbar sind.

`std_logic`, `std_ulogic`, `std_logic_vector(a downto b)`

1.8.2 Treiber <=

Der Treiber <= beschreibt, dass das linke Signal vom rechten Signal angetrieben wird. Folgendes Beispiel beschreibt einen Inverter:

```
Inv_Out <= not Inv_In;
```

2. VHDL Syntax

```
y <= (0 => '0', 1 => '0', 2 => '0', 3 => '0');
y <= (others => '0');
y <= "0000";
```

Conditional Signal Assignment

```
y <= x when en = '1' else "0000";
y <= x when en = '1' else (others => '0');
```

Prozess Statement with sequential loop-Statement

```
process(x,en)
begin
  for k in 3 downto 0 loop
    y(k) <= x(k) and en;
  end loop;
end process;
```

3. Vivado

3.1 Project Summary

3.1.1 Utilization

Unter *Utilization* in der *Project Summary* kann die Post-Synthesis und -Implementation beschreibt die verwe

4. Finite State Machines (FSM)

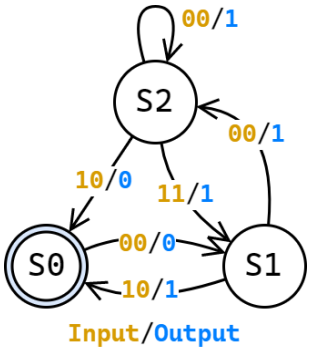
Eine Zustandsmaschine beschreibt ein System in diskreten Zuständen. In **VHDL** wird für Mealy- & Moore-Automaten jeweils ein *memoryless* und ein *memorizing* Prozess verwendet. Der *memoryless* Prozess verarbeitet die Zustandswechsel und die Ausgänge (wobei dies Abhängig vom FSM-Typ ist). Der *memorizing* Prozess ist für die Zustands-Zurücksetzung und -zuweisung zuständig.

Allgemeine Definition ZSM

$$o[k] = g(i[k], s[k])$$
$$s[k+1] = f(i[k], s[k])$$

k : diskrete Zeit mit $t = k \cdot T_{CLK}$, $k = 0$ entspricht Reset-Zeitpunkt
s : Zustand des Systems mit $s \in S = \{S_0, S_1, \dots S_N\}$
i : Input des Systems mit $i \in I = \{I_0, I_1, \dots I_M\}$
o : Output des Systems mit $o \in O = \{O_0, O_1, \dots, O_K\}$
g : Output Funktion, berechnet aktuellen Output des Systems
f : Next-State Funktion, berechnet nächsten Zustand des Systems

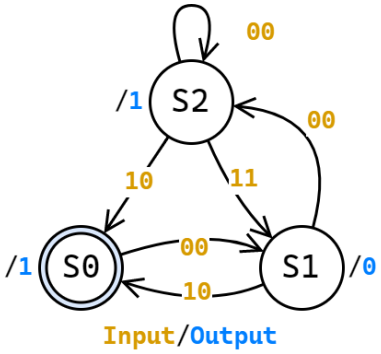
4.1 FSM-Typ: Mealy



$$o[k] = g(i[k], s[k])$$
$$s[k+1] = f(i[k], s[k])$$

Beim *Mealy* werden die Ausgänge beim Zustandswechsel geändert.

4.2 FSM-Typ: Moore



$$o[k] = g(s[k])$$
$$s[k+1] = f(i[k], s[k])$$

Beim *Moore* werden die Ausgänge im Zustand geändert.

Unterschied Mealy- & Moore-Outputs

Eine Mealy-FSM ändert den Ausgang beim Zustandswechsels, was einen *0-Delay* einführt. Eine Moore-FSM verändert den Ausgang im Zustand, was einen *1-Delay* einführt.

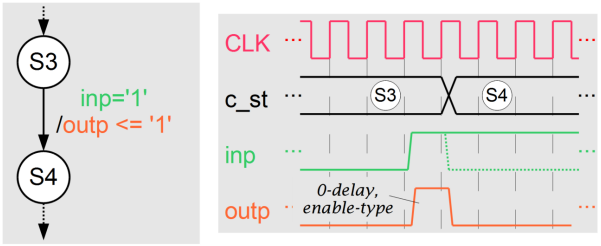


Abbildung 1: Mealy

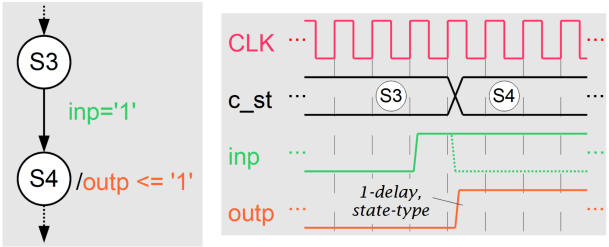


Abbildung 2: Moore

4.3 FSM-Typ: Medvedev

Medvedev hat eine ähnlichen Aufbau wie *Moore*, wobei der Ausgang direkt dem Zustandswert entspricht und keine Zwischen-Konvertierung gemacht wird.

$$o[k] = s[k]$$
$$s[k+1] = f(i[k], s[k])$$

4.4 Parasitäre Zustände

Jedes weitere Zustands-Flip-Flop erweitert die Anzahl Faktoren um den Faktor 2 ($S = 2^N$). Ungebrauchte Zustände werden *parasitäre Zustände* genannt.

$$n_{para} = 2^N - S \qquad n_{para}\big|_{S=3,N=2} = 2^2 - 3 = 1$$

Folgende Formel kann die Anzahl benötigten Flip-Flops berechnen

$$N = \lceil \log_2(S) \rceil = \left\lceil \frac{\log(S)}{\log(2)} \right\rceil \qquad N\big|_{S=3} = \lceil \log_2(5) \rceil = 3$$

N : Anzahl Flip-Flops
S : Anzahl verwendete Zustände

4.5 State Encoding

Zustände können auf verschiedene Arten dargestellt werden, bekannte Varianten sind *binär* und *One Hot*.

Zustand	Binär	One-Hot
S_0	00	001
S_1	01	010
S_2	10	100
Parasitäre Zustände	11	000, 011, 111, 110, 101

Parasitäre Zustände

Alle **ungebrauchten** Zustände sind *parasitäre Zustände*!

4.5.1 Binär

Meistverwendetes Format ist *binär*, da es **kompakt** und **einfach erweiterbar** ist.

- $S_0 \rightarrow$ 0000
- $S_1 \rightarrow$ 0001
- $S_2 \rightarrow$ 0010

4.5.2 One-Hot

Bei *One-Hot* ist **ein Bit** *high* und **alle anderen Bits** *low* oder in anderen Worten, nur ein Bit ist aktiv.

4.6 Goldene Regeln der (FSM) Implementierung

- Memoryless Process (kombinatorische Logik)
  - Alle Eingangssignale der FSM und der aktuelle Zustand müssen in der *sensitivity list* aufgeführt werden.
  - Jedem Ausgangssignal muss für jede mögliche Kombination von Eingangswerten (inkl. parasitäre Input-Symbole) ein Wert zugewiesen werden. **Keine Zuweisung bedeutet sequentielles Verhalten (Speicher)!**
  - Parasitäre Zustände sollten mittels `others` abgefangen werden.
- Memorizing Process (sequentielle Logik)
  - Ausser Clock und (asynchronem) Reset dürfen keine Signale in die *sensitivity list* aufgenommen werden.
  - Das den Zustand repräsentierende Signal muss einen Reset-Wert erhalten.

5. Vorlagen

> **Hinweis**
>
> Der Inhalt der Prozess-Templates wird in den `=>CUSTOM` gekennzeichneten Abschnitten geschrieben.

5.1 Positive Getriggertes D-FlipFlop

5.1.1 Mit asynchronem Reset

```
-- mit asynchronem Reset
process (rst, clk)
-- Deklarationen => CUSTOM
begin
  if rst = '1' then
    -- asynchr. Reset => CUSTOM
    Q <= '0';
  elsif rising_edge(clk) then
    -- getaktete Logik => CUSTOM
    Q <= D;
  end if;
end process;
```

5.1.2 Ohne Reset

```
process (clk)
-- Deklarationen => CUSTOM

begin
  if rising_edge(clk) then
    -- getaktete Logik => CUSTOM
    Q <= D;
  end if;
end process;
```

5.2 Finite State Machine

5.2.1 Mealy

```
type state is (S0, S1, S2);
signal c_st, n_st : state;

p_seq: process (rst, clk)                          ①
begin
  if rst = '1' then
    c_st <= S0;
  elsif rising_edge(clk) then
    c_st <= n_st;
  end if;
end process;

p_com: process (i, c_st)                           ②
begin
  -- default assignments
  n_st <= c_st; -- remain in current state
  o <= '1'; -- most frequent value
  -- specific assignments
  case c_st is
    when S0 =>
      if i = "00" then
        o <= '0';
        n_st <= S1;
      end if;
    when S1 =>
      if i = "00" then
        n_st <= S2;
      elsif i = "10" then
        n_st <= S0;
      end if;
    when S2 =>
      if i = "10" then
        o <= '0';
        n_st <= S0;
      elsif i = "11" then
        n_st <= S1;
      end if;
  when others =>
    -- handle parasitic states
    n_st <= S0;
  end case;
end process;
```

① *Memorizing* (sequentielle Logik)
② *Memoryless* (kombinatorische Logik)

5.2.2 Moore

```
type state is (S0, S1, S2);
signal c_st, n_st : state;

p_seq: process (rst, clk)                          ①
begin

  if rst = '1' then
    c_st <= S0;
  elsif rising_edge(clk) then
    c_st <= n_st;
  end if;
end process;

p_com: process (i, c_st)                           ②
begin
  -- default assignments
  n_st <= c_st; -- remain in current state
  o <= '1'; -- most frequent value
  -- specific assignments
  case c_st is
    when S0 =>
      if i = "00" then
        n_st <= S1;
      end if;
    when S1 =>
      if i = "00" then
        n_st <= S2;
      elsif i = "10" then
        n_st <= S0;
      end if;
      o <= '0'; -- uncondit. output assignment
    when S2 =>
      if i = "10" then
        n_st <= S0;
      elsif i = "11" then
        n_st <= S1;
      end if;
    when others =>
      -- handle parasitic states
      n_st <= S0;
  end case;
end process;
```

① *Memorizing* (sequentielle Logik)
② *Memoryless* (kombinatorische Logik)