HOCHSCHULE
FÜR ANGEWANDTE
WISSENSCHAFTEN FH
INGOLSTADT

Praktikum Rechnerarchitektur

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN FH INGOLSTADT

Praktikum Rechnerarchitektur

Inhalt

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN FH INGOLSTADT

Literatur

Altera-Tutorials

Quartus II Introduction Using Schematic Design

Quartus II Introduction Using VHDL Design

Die Hardwarebeschreibungssprache VHDL

Peter J. Ashenden: *The Designer's Guide to VHDL*, Morgan Kaufmann, 3. Auflage, 2006

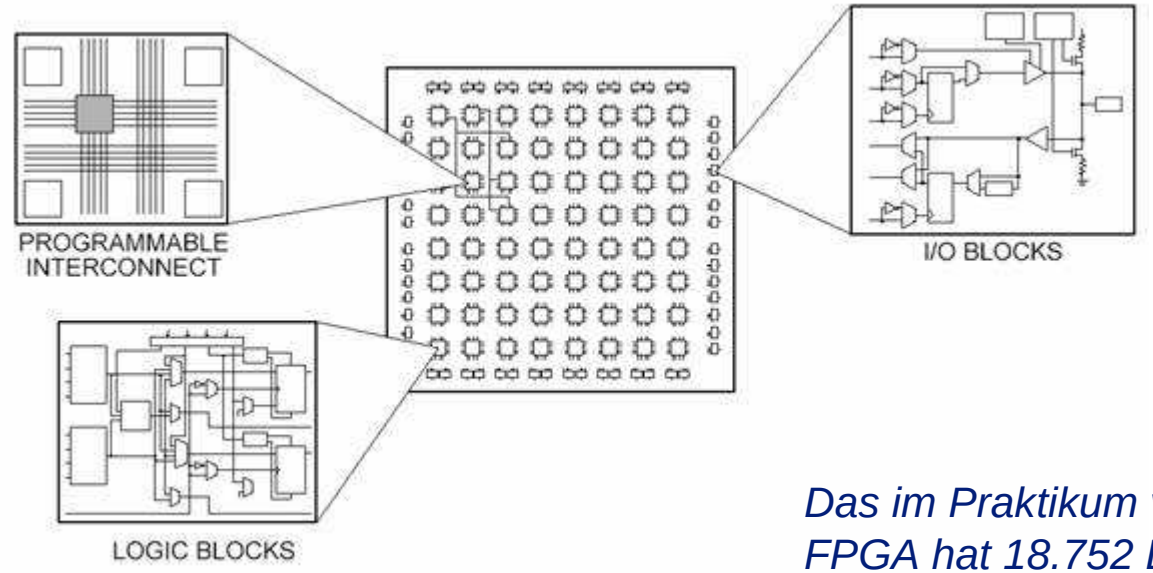
Jürgen Reichardt, Bernd Schwarz: *VHDL-Synthese: Entwurf digitaler Schaltungen und Systeme*, Oldenbourg, 5. Auflage, 2009

Paul Molitor, Jörg Ritter: *VHDL – Eine Einführung*, Pearson Studium, 2004

Andreas Mäder: *VHDL Kompakt*, http://tams-www.informatik.uni-hamburg.de/research/vlsi/vhdl/doc/ajmMaterial/vhdl.pdf

Field Programmable Gate Array (FPGA)

Field Programmable Gate Arrays (FPGAs) sind hochintegrierte Halbleiterbausteine mit einer regelmäßigen Anordnung programmierbarer Logikblöcke, die beliebig miteinander verbunden werden können. An den Bausteinrändern befinden sich spezialisierte Ein-/Ausgabeblöcke:

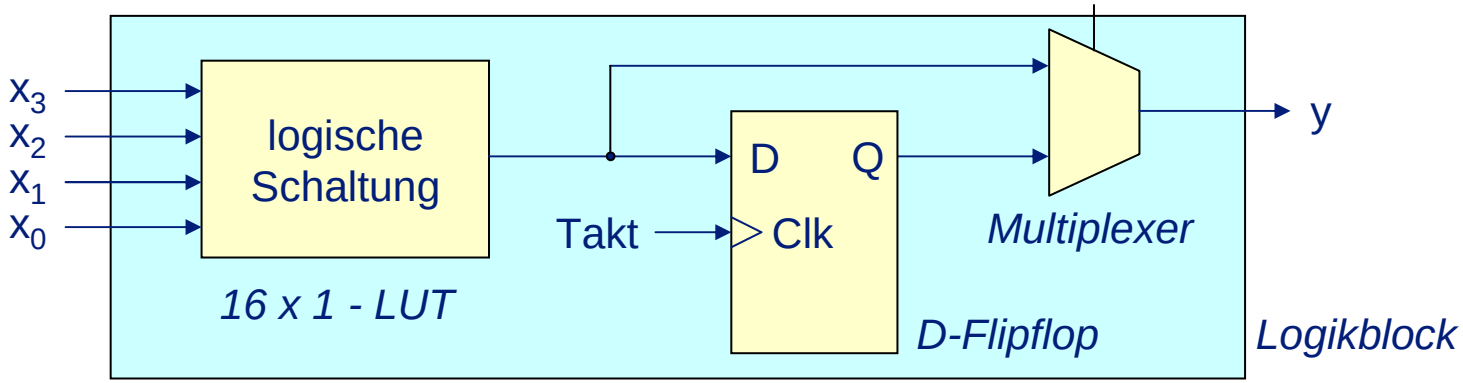


Das im Praktikum verwendete FPGA hat 18.752 Logikblöcke und 315 I/O-Pins.

Field Programmable Gate Array (FPGA)

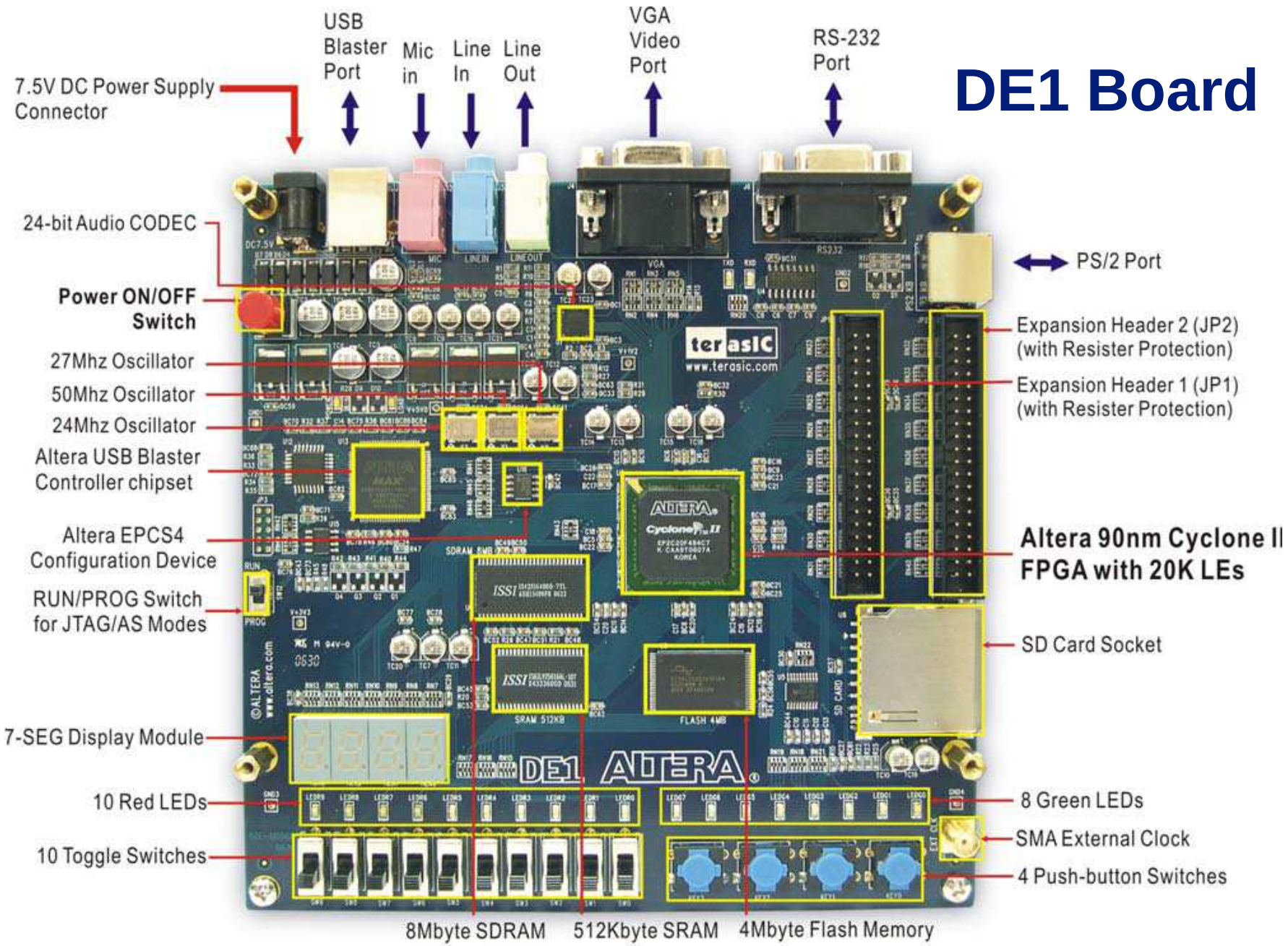
Die Logikblöcke sind meist als *Look-Up-Tabellen (LUT)* mit 4 Eingängen und 1 Ausgang implementiert. Solche Tabellen realisieren eine logische Funktion $y = f(x_3, x_2, x_1, x_0)$, indem für alle möglichen Kombinationen der Eingangsvariablen x_i der zugehörige Funktionswert y in einem statischen Speicher abgelegt wird.

Für 4 Eingangsvariablen mit den Werten 0 und 1 gibt es $2^4 = 16$ Eingangskombinationen.

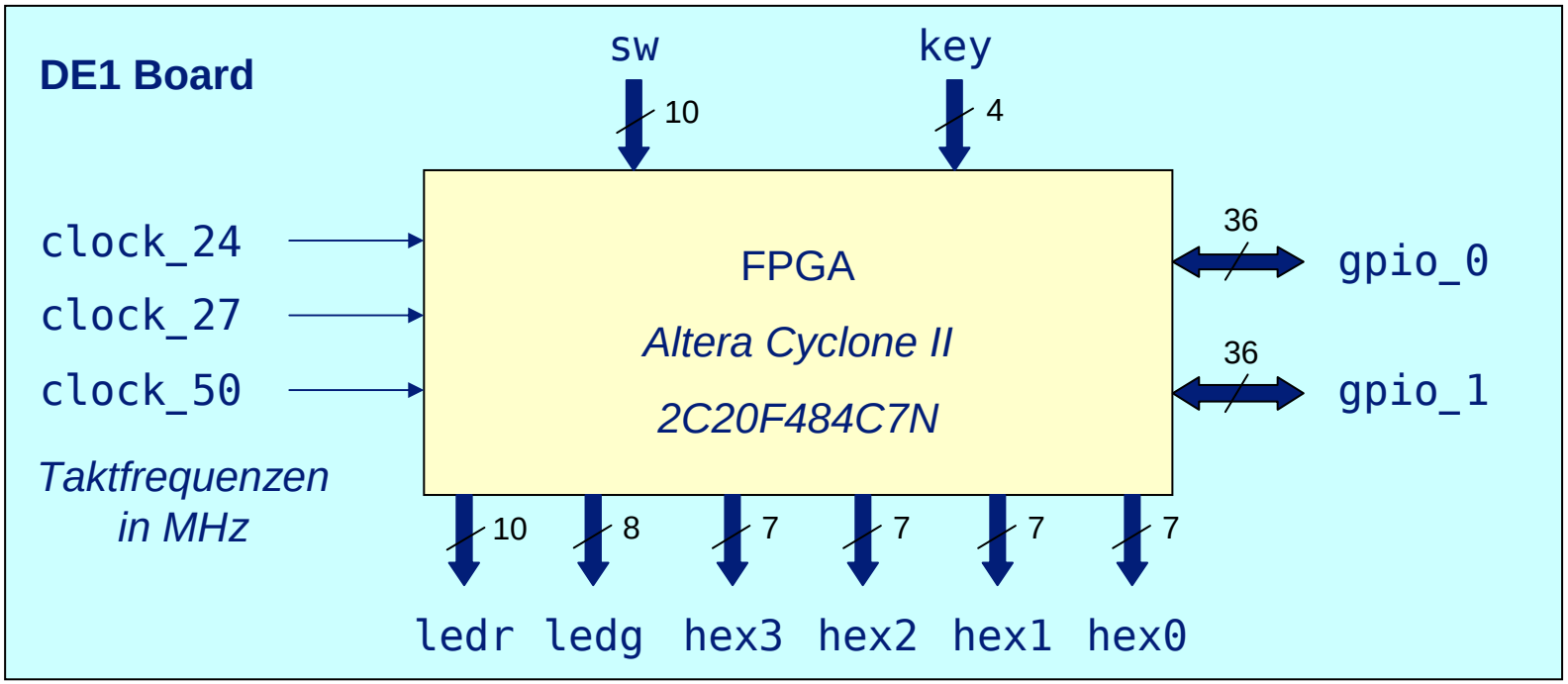


Der Funktionswert y (0 oder 1) kann *asynchron* oder *synchron* zu einem Taktsignal ausgegeben werden. Bei der asynchronen Ausgabe wird der Funktionswert sofort ausgegeben, bei der synchronen Ausgabe erst mit der nächsten Taktflanke.

DE1 Board


USB Blaster Port
Mic in
Line In
Line Out
VGA Video Port
RS-232 Port
7.5V DC Power Supply Connector
24-bit Audio CODEC
Power ON/OFF Switch
27Mhz Oscillator
50Mhz Oscillator
24Mhz Oscillator
Altera USB Blaster Controller chipset
Altera EPCS4 Configuration Device
RUN/PROG Switch for JTAG/AS Modes
7-SEG Display Module
10 Red LEDs
10 Toggle Switches
PS/2 Port
Expansion Header 2 (JP2) (with Resister Protection)
Expansion Header 1 (JP1) (with Resister Protection)
Altera 90nm Cyclone II FPGA with 20K LEs
SD Card Socket
8 Green LEDs
SMA External Clock
4 Push-button Switches
8Mbyte SDRAM
512Kbyte SRAM
4Mbyte Flash Memory

DE1 Development and Evaluation Board



`sw`	*10 Schalter*	`key`	*4 entprellte Taster (active low)*
`ledr`	*10 rote Leuchtdioden*	`hex#`	*4 Siebensegmentanzeigen (active low)*
`ledg`	*8 grüne Leuchtdioden*	`gpio_#`	*2 36-Bit-General-Purpose-IOs*

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN FH INGOLSTADT

Schaltungssynthese mit VHDL

VHDL: Very High Speed Integrated Circuit Hardware Description Language

VHDL ist – neben Verilog – eine standardisierte *Hardware-Beschreibungssprache* für den Entwurf hochintegrierter Schaltkreise. Ein VHDL-Compiler erzeugt keinen *sequentiellen Maschinencode*, sondern *parallele Hardware* (in Form von *Netzlisten*, die z.B. auf die Logikblöcke eines FPGA abgebildet werden) → *Schaltungssynthese*

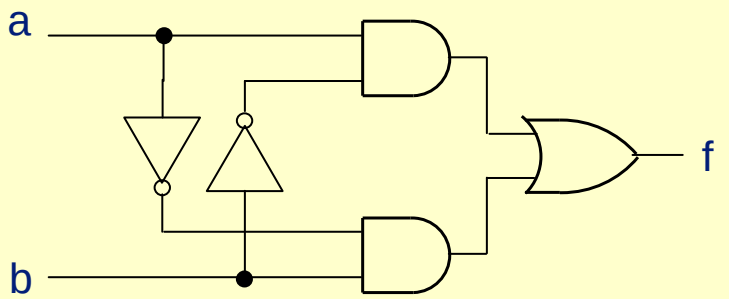
```
while (s[i] != '\0') i++;
```

↓ *C-Compiler*

```
loop    add.l   #1,d0
        tst.b   (a1)+
        bne     loop
```

```
f <= (a and not b) or (b and not a);
```

↓ *VHDL-Compiler*



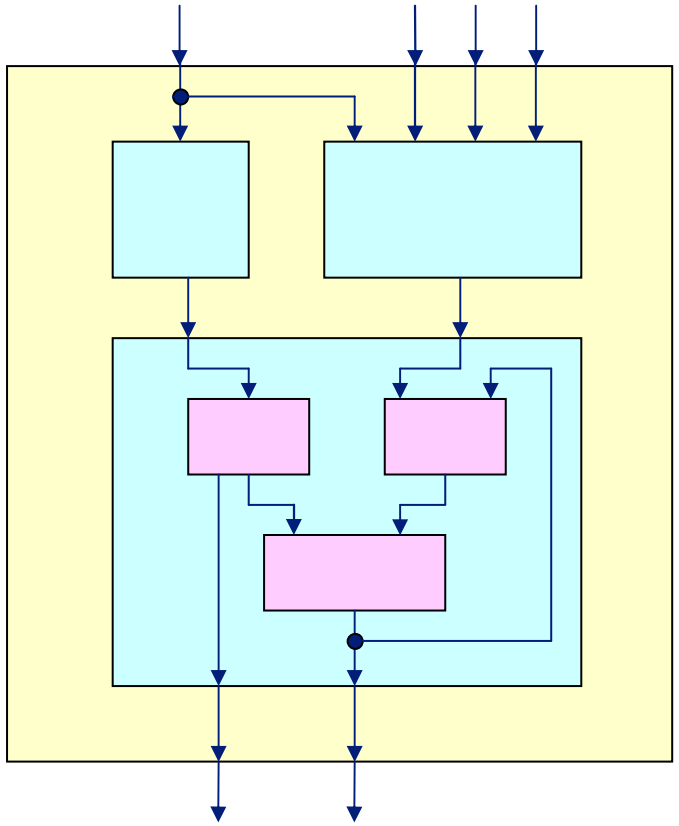
Schaltungssynthese mit VHDL

Mit VHDL lassen sich Schaltungen beliebiger Komplexität *top-down* entwerfen und realisieren.

VHDL beschreibt Schaltungen als *Hierarchie* von Komponenten, die entweder elementar sind oder aus anderen Komponenten bestehen.

Komponenten werden wie folgt beschrieben:

- *Schnittstellenbeschreibung*: Anzahl und Typ der Ein-/Ausgänge ("black box")
- *Strukturbeschreibung:* Verschaltung der Teil-komponenten (nur dann erforderlich, wenn es solche gibt)
- *Verhaltensbeschreibung*: was die Schaltung macht, sowohl als parallele *Hardware* wie auch als zeitlich sequentielle *Prozesse*



HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN FH INGOLSTADT

Schaltungssynthese mit VHDL

Die *Schnittstellen* einer Schaltung werden durch eine sogenannte **entity** beschrieben. Nachfolgend die VHDL-Beschreibung des FPGAs auf dem DE1-Board:

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

entity FPGA is  -- FPGA-Schnittstellen
    port (clock_24, clock_27, clock_50: in std_logic; -- Takt
          sw  : in  std_logic_vector(9 downto 0); -- Schalter
          key : in  std_logic_vector(3 downto 0); -- Taster
          ledr: out std_logic_vector(9 downto 0); -- rote LEDs
          ledg: out std_logic_vector(7 downto 0); -- grüne LEDs
          hex3, hex2, hex1, hex0:
                out std_logic_vector(0 to 6)); -- 7-Seg-Anzeigen
end FPGA;
```

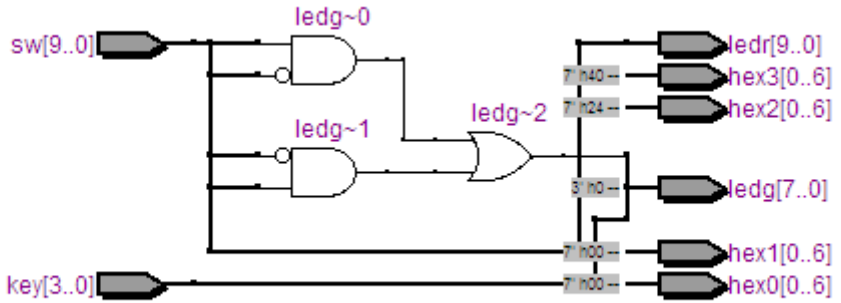
Schaltungssynthese mit VHDL

Das *Verhalten* einer **entity** wird durch eine zugehörige **architecture** beschrieben.

Im nebenstehendem Beispiel werden einige Eingangssignale auf bestimmte Ausgänge durchgeschaltet (zum Teil nach logischer Verknüpfung).

Der VHDL-Compiler synthetisiert aus der VHDL-Beschreibung eine sogenannte *Netzliste*; diese wird dann in das FPGA geladen und realisiert dadurch die Hardwarefunktionalität.

```
architecture Verhalten of FPGA is
begin
    ledr <= sw;
    ledg(3 downto 0) <=  key;
    hex3(6) <= '1';
    hex2 <= "0010010";
    ledg(7) <= (not sw(7) and sw(6))
            or (sw(7) and not sw(6));
end Verhalten;
```



HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN FH INGOLSTADT

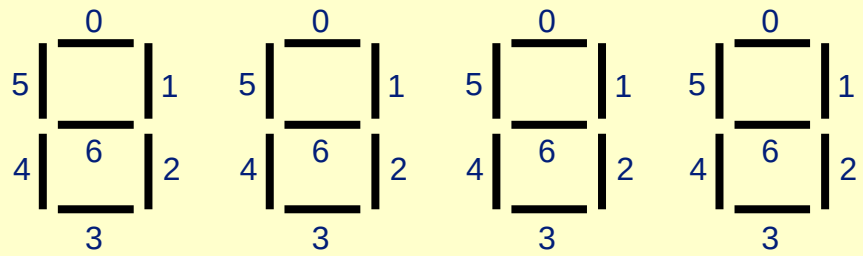
Versuch 1: kombinatorische Logik

a) Ansteuerung von Siebensegmentanzeigen

Jedes Segment einer Siebensegment-anzeige kann einzeln angesteuert werden:

bei Ansteuerung mit `'0'` leuchtet es, bei Ansteuerung mit `'1'` bleibt es dunkel.

Die Ansteuerung wird in VHDL durch eine *Signalzuweisung* realisiert:



entweder für ein einzelnes Segment:
```
hex3(6) <= '1';
```
... oder für mehrere Segmente:
```
hex2(4 to 5) <= "11";
```
... oder für alle sieben Segmente:
```
hex1 <= "0000110";
```

Der *Signalzuweisungsoperator* <= kann wie gezeigt sowohl für skalare (1-Bit) als auch für vektorwertige (n-Bit) Signale verwendet werden.

Realisieren Sie eine Schaltung zur Anzeige der letzten 4 Ziffern Ihrer Matrikelnummer.

Versuch 1: kombinatorische Logik

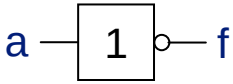
Exkurs: logische Grundoperationen und deren Realisierung als Gatter

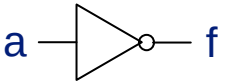
Op	Funktion	Syntax einer Zuweisung	DIN-Symbol	IEEE-Symbol

not

a	f
0	1
1	0

```
C:      f  = ~ a;
VHDL:   f <= not a;
```

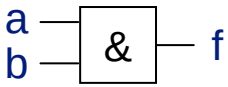


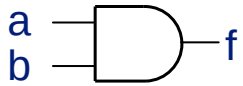


and

a	b	f
0	0	0
0	1	0
1	0	0
1	1	1

```
C:      f  = a && b;
VHDL:   f <= a and b;
```

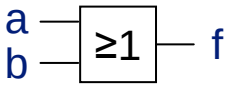


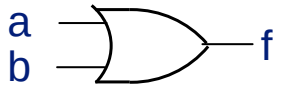


or

a	b	f
0	0	0
0	1	1
1	0	1
1	1	1

```
C:      f  = a || b;
VHDL:   f <= a or b;
```





Beachte: die C-Operatoren && und || haben verschiedenen Rang, aber die VHDL-Operatoren **and** *und* **or** *haben den gleichen Rang!*

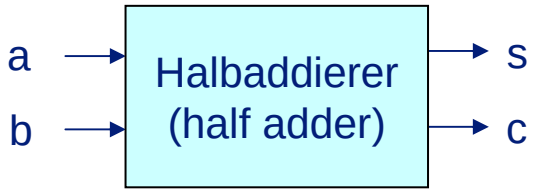
Versuch 1: kombinatorische Logik

b) Halbaddierer

a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Wahrheitstabelle des Halbaddierers

Der Halbaddierer addiert zwei Eingänge (a, b) und liefert zwei Ausgänge (Summe, Übertrag/Carry).



Die Funktionalität läßt sich durch eine *Wahrheitstabelle (truth table)* definieren. Dabei werden alle Kombinationen der binären Eingänge tabelliert und die Ausgangswerte (0 oder 1) für jede Kombination angegeben.

Eine Funktion ergibt sich dann durch *Oder-Verknüpfung* aller Kombinationen, für die ihr Ausgang 1 ist; jede Kombination ergibt sich dabei durch *Und-Verknüpfung* der Eingänge:

```
s <= (not a and b) or (a and not b);
c <= a and b;
```

Realisieren Sie den Halbaddierer mit zwei Schaltern als Eingangssignalen, einer roten LED für die Summe und einer grünen LED für den Übertrag.

Versuch 1: kombinatorische Logik

c) 2-zu-1-Multiplexer

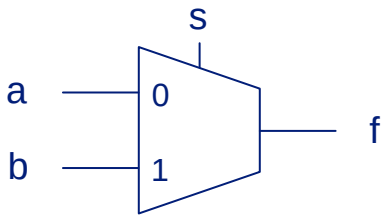
Ein *2-zu-1-Multiplexer* wählt mit Hilfe eines Steuersignales s aus zwei Eingangssignalen a und b eines aus und schaltet es auf den Ausgang f durch.

Dafür eignet sich die ***when***-*Anweisung*:

```
f <= a when s = '0' else b;
```

Multiplexer können Signale/Busse beliebiger Breite schalten. Ein *n-Bit-Multiplexer* setzt sich dabei aus n 1-Bit-Multiplexern zusammen.

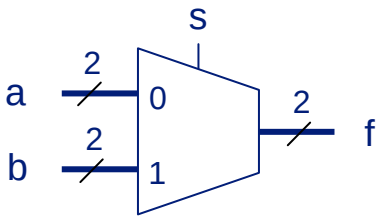
Realisieren Sie einen 5-Bit-Multiplexer, der die Schalter sw_{4-0} bzw. sw_{9-5} auf die $ledg_{4-0}$ schaltet. Das Auswahlsignal s wird durch key_0 dargestellt.



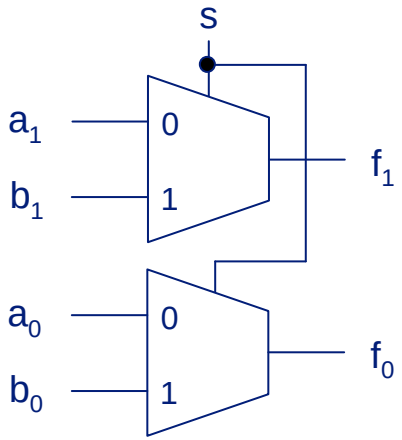
Schaltsymbol (1-Bit-Multiplexer)

s	f
0	a
1	b

Wahrheitstabelle



Schaltsymbol (2-Bit-Multiplexer)



Aufbau mit 1-Bit-Mux

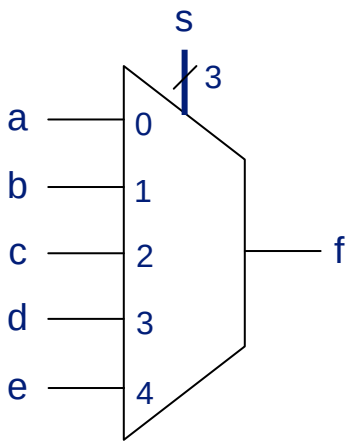
Versuch 1: kombinatorische Logik

d) 5-zu-1-Multiplexer

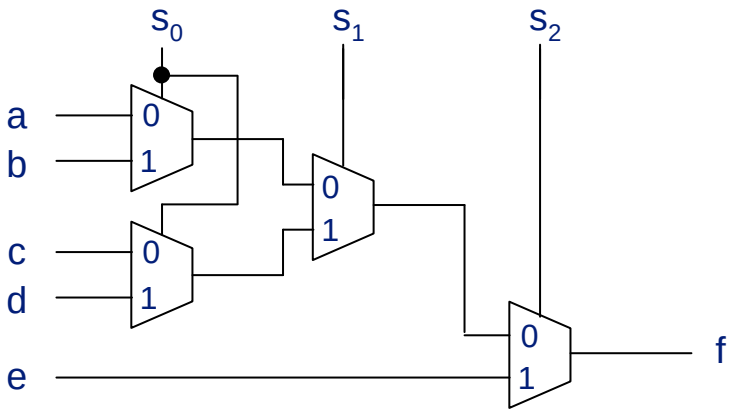
Ein *5-zu-1-Multiplexer* wählt eines von fünf Eingangssignalen aus:

s_2	s_1	s_0	f
0	0	0	a
0	0	1	b
0	1	0	c
0	1	1	d
1	0	0	e
1	0	1	e
1	1	0	e
1	1	1	e

Wahrheitstabelle



Schaltsymbol



Aufbau mit 2-zu-1-Multiplexern

Realisieren Sie einen 5-zu-1-Multiplexer, der einen der Schalter sw_{4-0} auf die $ledg_7$ durchschaltet. Das Auswahlsignal s wird durch die Schalter sw_{9-7} dargestellt.

Versuch 1: kombinatorische Logik

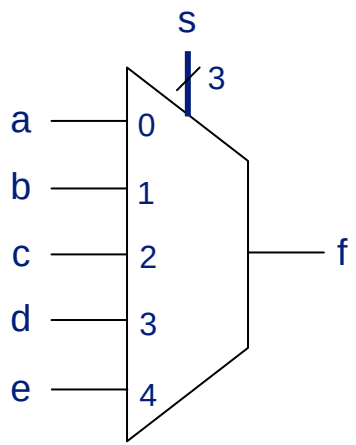
d) 5-zu-1-Multiplexer

Lösungsansatz entweder mit der *bedingten Zuweisung*

```
f <= a when s = "000" else
     b when s = "001" else
     c when s = "010" else
     d when s = "011" else
     e;
```

... oder mit der *selektiven Zuweisung*

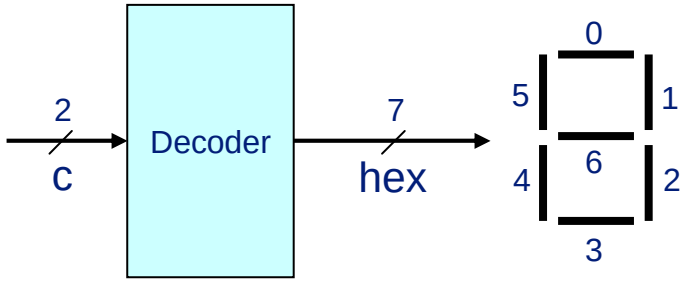
```
with s select
   f <= a when "000",
        b when "001",
        c when "010",
        d when "011",
        e when others;
```



Versuch 2: hierarchisches Design

a) Ansteuerung einer Siebensegmentanzeige

Entwerfen Sie einen Decoder zur Ansteuerung einer Siebensegmentanzeige.
Beachte: die Segmente leuchten nur bei Ansteuerung mit 0 („active low“).

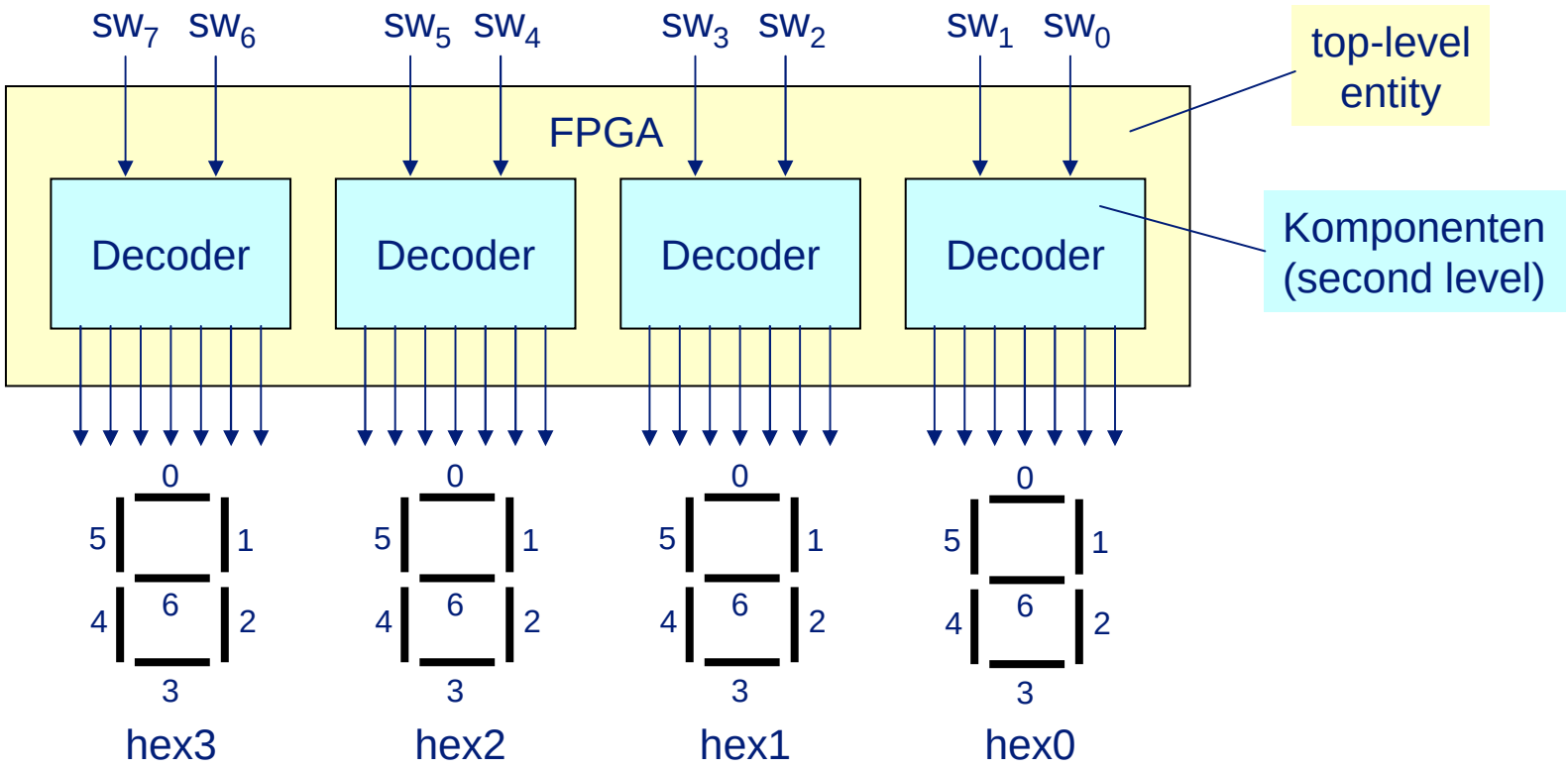


c_1	c_0	Anzeige
0	0	H
0	1	A
1	0	L
1	1	*(keine)*

Realisieren Sie den Decoder mit Hilfe einer bedingten oder einer selektiven Zuweisung.

Versuch 2: hierarchisches Design

Hierarchisches Design



HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN FH INGOLSTADT

Versuch 2: hierarchisches Design

VHDL-Komponenten wiederverwendbar gestalten

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

entity Decoder is
    port (c:   in  std_logic_vector (1 downto 0);
          hex: out std_logic_vector (0 to 6));
end decoder;

architecture a of Decoder is
begin
    ...
end a;
```

Diese Komponente verwendet symbolische Namen für die Ein- und Ausgänge. Sie ist daher *wiederverwendbar*; d.h. nicht auf bestimmte Eingänge (z.B. Schalter) bzw. Ausgänge (z.B. LEDs) des verwendeten DE1 Boards festgelegt.

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN FH INGOLSTADT

Versuch 2: hierarchisches Design

VHDL-Komponenten wiederverwenden

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

entity FPGA is
    port (sw: in std_logic_vector (9 downto 0);
          hex3, hex2, hex1, hex0: out std_logic_vector (0 to 6));
end FPGA;

architecture a of FPGA is
    component Decoder is      -- Komponente deklarieren
        port (c:   in  std_logic_vector (1 downto 0);
              hex: out std_logic_vector (0 to 6));
    end component;
begin    -- Komponente instantiieren und verdrahten
    d3: Decoder port map (c => sw(7 downto 6), hex => hex3);
    d2: Decoder port map (c => sw(5 downto 4), hex => hex2);
    d1: Decoder port map (c => sw(3 downto 2), hex => hex1);
    d0: Decoder port map (c => sw(1 downto 0), hex => hex0);
end a;   -- Beispiel für "nominal mapping" (Namensabbildung)
```

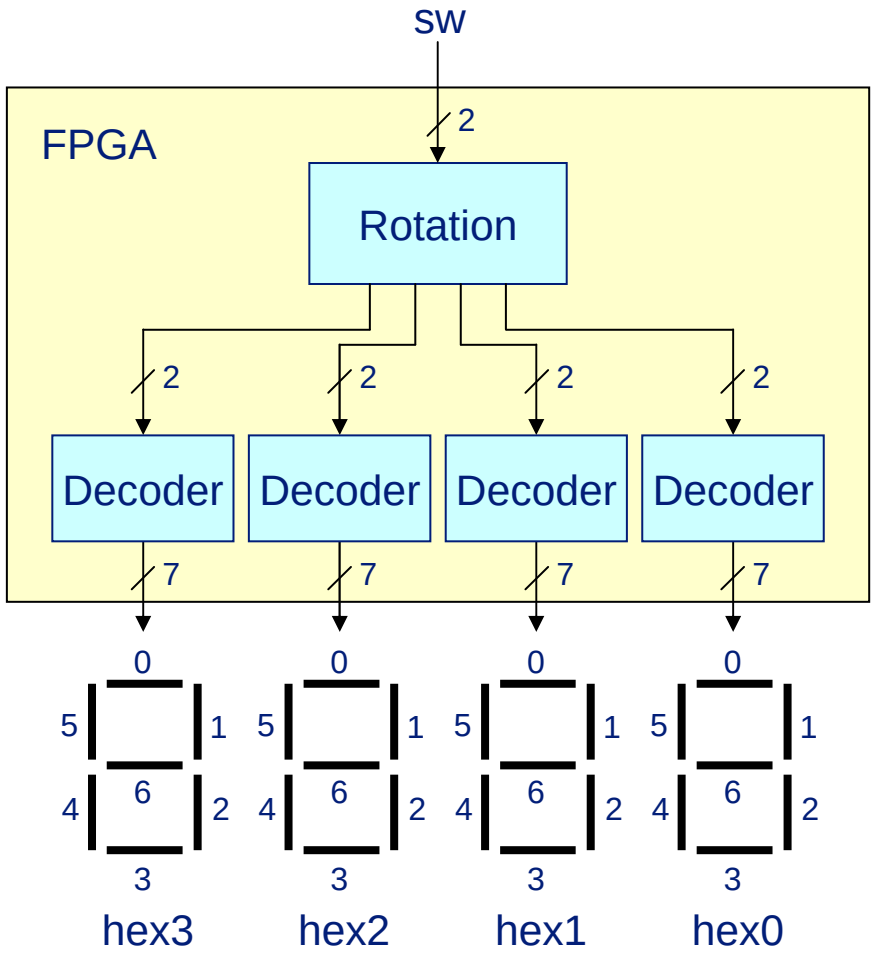
Versuch 2: hierarchisches Design

b) Rotierende Anzeige

Lassen Sie das Wort „HAL “ per Schalter durch 4 Siebensegment-anzeigen rotieren:

sw_1	sw_0	hex_3	hex_2	hex_1	hex_0
0	0	H	A	L	
0	1		H	A	L
1	0	L		H	A
1	1	A	L		H

Steuern Sie hierzu die 4 Decoder durch eine Komponente "Rotation" an.



HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN FH INGOLSTADT

Versuch 2: hierarchisches Design

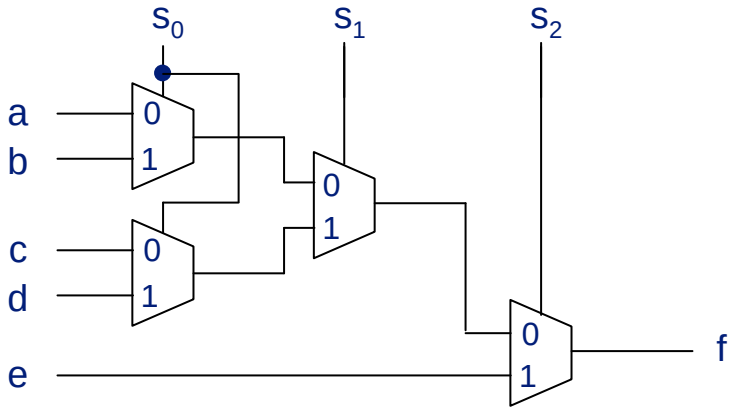
Verdrahtung von VHDL-Komponenten mit Signalen

```
architecture a of FPGA is -- deklariert zwei Komponenten
    component Rotation is
        port (x: in std_logic_vector (1 downto 0);
              y3, y2, y1, y0: out std_logic_vector (1 downto 0));
    end component;
    component Decoder is
        port (c:   in  std_logic_vector (1 downto 0);
              hex: out std_logic_vector (0 to 6));
    end component;
    signal s3, s2, s1, s0: std_logic_vector (1 downto 0);
begin -- s3 bis s0 sind "Hilfssignale" für die interne Verdrahtung
    r: Rotation port map (sw(1 downto 0), s3, s2, s1, s0);
    d3: Decoder port map (s3, hex3);
    d2: Decoder port map (s2, hex2);
    d1: Decoder port map (s1, hex1);
    d0: Decoder port map (s0, hex0);
end a; -- Beispiel für "positional mapping" (Positionsabbildung)
```

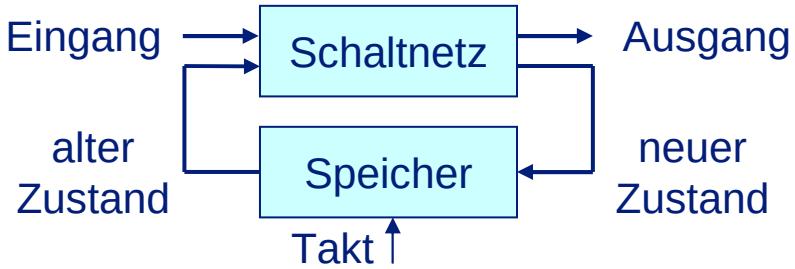
Versuch 3: sequentielle Logik

Kombinatorische und sequentielle Schaltungen

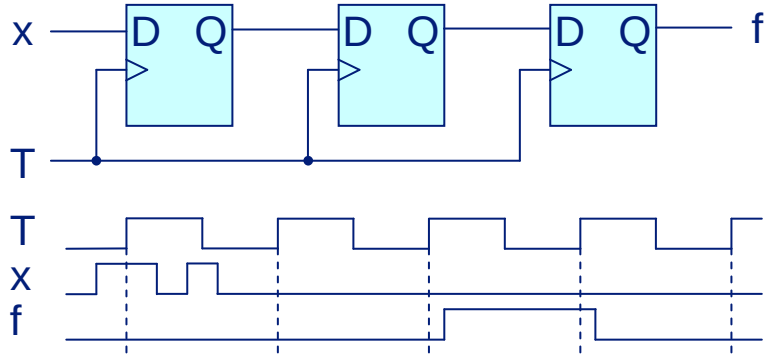
Bei *kombinatorischen Schaltungen* ist der Ausgang ausschließlich eine Funktion der aktuellen Eingangswerte.



kombinatorische Schaltung



Bei *sequentiellen Schaltungen* ist der Ausgang eine Funktion der aktuellen und/oder der früheren Eingangswerte.



HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN FH INGOLSTADT

Versuch 3: sequentielle Logik

Paralleler und sequentieller VHDL-Code

VHDL unterscheidet zwischen parallelem und sequentiellem Code. *Paralleler Code* beschreibt kombinatorische Schaltungen. Zu diesem Code gehören im wesentlichen die unbedingten, bedingten und selektiven Zuweisungen:

```
f <= (a and b) xor c;
g <= a when b = '1' else c;
with x select h <= a when "00", b when "11", c when others;
```

Sequentieller Code kann sowohl sequentielle als auch kombinatorische Schaltungen beschreiben. Sequentieller Code muss in einer **process**-Anweisung plaziert werden:

```
process (a, b, c)                    -- "sensitivity list"
begin
    -- sequentieller Code (z.B. if-, case-, loop- oder wait-Anweisungen)
end process;
```

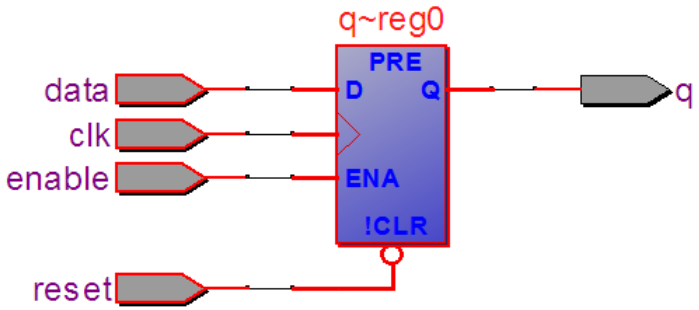
Prozesse werden zyklisch ausgeführt, und zwar genau dann, wenn sich eines der Signale aus der jeweiligen sensitivity list ändert.

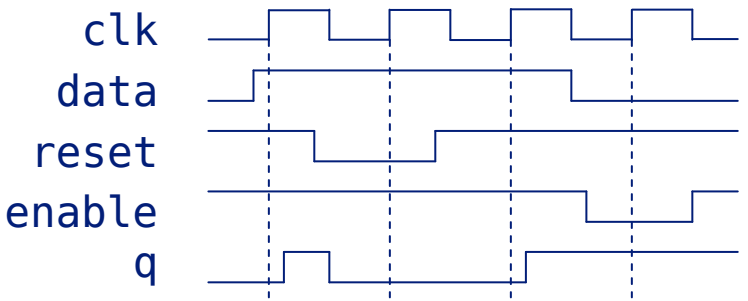
Versuch 3: sequentielle Logik

Grundelement sequentieller Schaltungen: das Flipflop als 1-Bit-Speicher

```
entity Flipflop is
  port (reset, enable, clk, data: in std_logic;
        q: out std_logic);
end Flipflop;

architecture a of Flipflop is
begin
  process (reset, clk) begin
    if reset = '0' then
      q <= '0';
    elsif rising_edge (clk) then
      if enable = '1' then
        q <= data;
      end if;
    end if;
  end process;
end a;
```





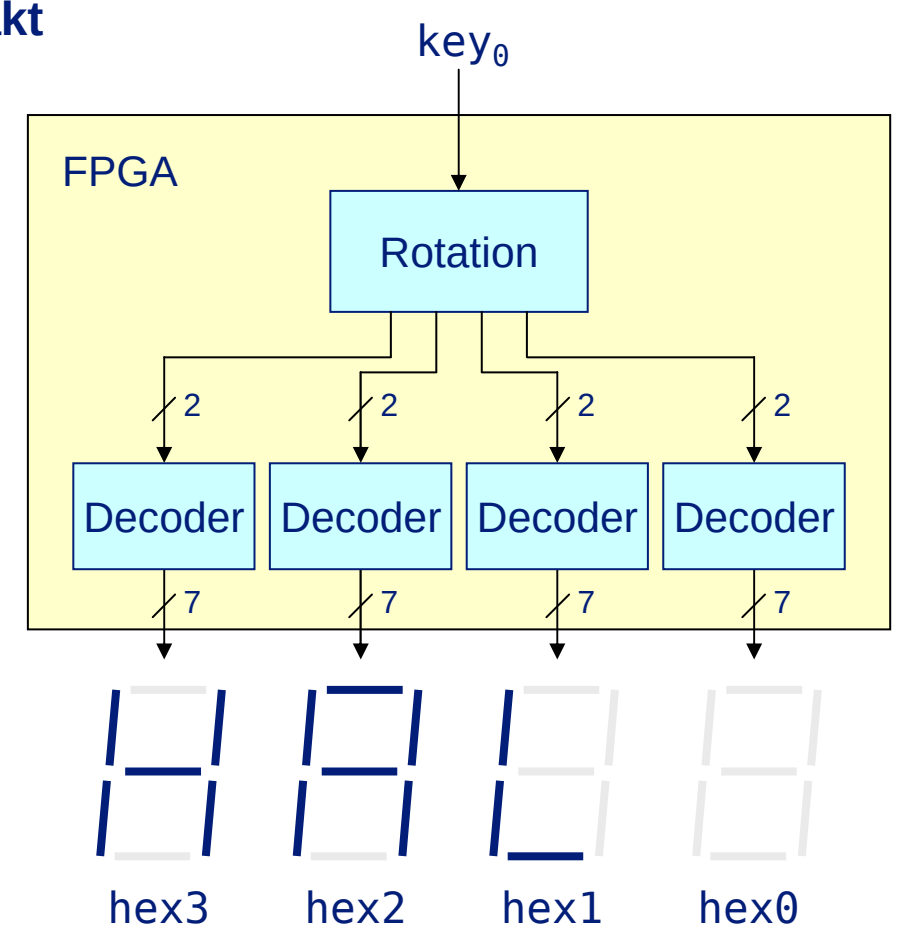
Versuch 3: sequentielle Logik

a) Rotierende Anzeige mit Handtakt

Lassen Sie das Wort „HAL “ bei jedem Tastendruck durch 4 Siebensegment-anzeigen nach rechts rotieren.

Ändern Sie hierzu die Schaltung aus Versuch 2b wie folgt:

- Ersetzen Sie die beiden Schalter als Eingangssignal der Komponente `Rotation` durch einen Drucktaster.
- Ändern Sie das Verhalten der Komponente `Rotation`, indem Sie bei jeder Taktflanke einen 2-Bit-Zähler hochzählen (wie auf der nächsten Seite gezeigt); dieser Zähler codiert die 4 möglichen Stellungen des Wortes „HAL “.



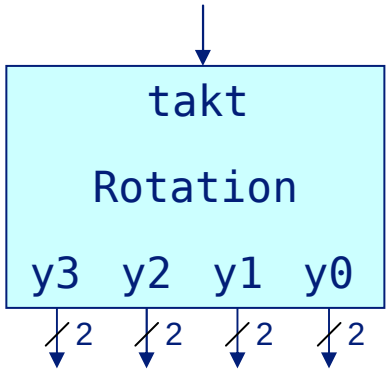
HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN FH INGOLSTADT

Versuch 3: sequentielle Logik

Rechtsrotation des Wortes "HAL ", getaktet mit fallender Flanke

```
entity Rotation is
   port (takt           : in  std_logic;
         y3, y2, y1, y0: out std_logic_vector (1 downto 0));
end Rotation;

architecture a of Rotation is
   signal z: natural range 0 to 3;
begin
   process (takt) begin
      if falling_edge (takt) then
         z <= z + 1;
      end if;
   end process;
   -- Der Wert von z entspricht nun dem Wert
   -- der beiden Schalter aus Versuch 2b.
end a;
```



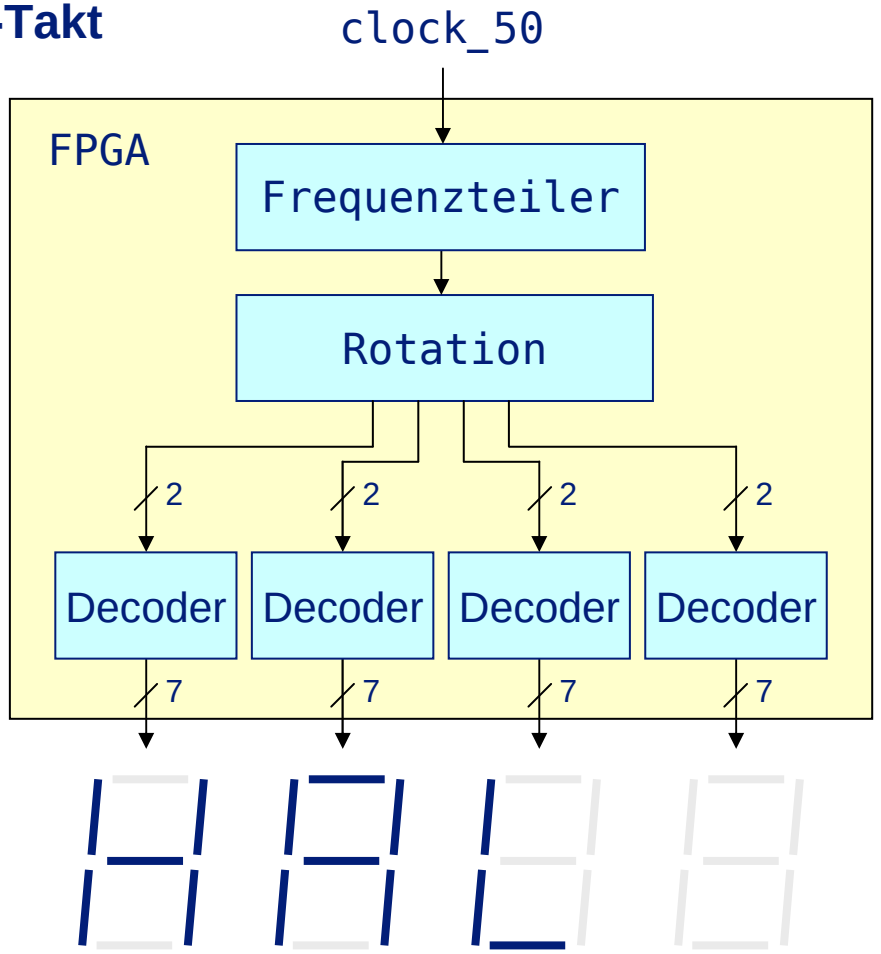
Versuch 3: sequentielle Logik

b) Rotierende Anzeige mit 50-MHz-Takt

Lassen Sie das Wort „HAL “ im Sekundentakt nach rechts rotieren.

Ändern Sie hierzu die Schaltung aus Versuch 3a wie folgt:

- Definieren Sie eine neue Komponente `Frequenzteiler`, welche den 50-MHz-Takt `clock_50` in einen 1-Hz-Takt herunterteilt (siehe nächste Seite).
- Steuern Sie die unveränderte Komponente `Rotation` mit diesem Sekundentakt an.



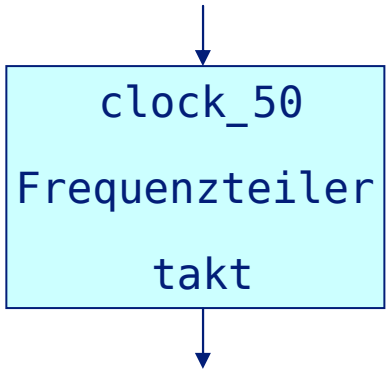
HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN FH INGOLSTADT

Versuch 3: sequentielle Logik

Frequenzteiler: von 50 MHz zu 1 Hz

```
entity Frequenzteiler is
   port (clock_50: in std_logic; takt: buffer std_logic);
end Frequenzteiler;

architecture a of Frequenzteiler is begin
   process (clock_50)
      variable zaehler: natural range 0 to 25000000;
   begin
      if rising_edge (clock_50) then
         zaehler := zaehler + 1;
         if zaehler >= 25000000 then
            zaehler := 0;
            takt <= not takt;
         end if;
      end if;
   end process;
end a;
```



HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN FH INGOLSTADT

Versuch 3: sequentielle Logik

Signale und Variablen

VHDL unterscheidet zwischen *Signalen* und *Variablen*. Signale sind *global*, d.h in der gesamten Schaltung (architecture) sichtbar, Variablen dagegen sind immer *lokal* zu einem Prozess: sie repräsentieren den aktuellen Zustand des Prozesses.

Signale und Variable haben verschiedene Zuweisungsoperatoren "<=" bzw. ":="

Beachte: Signale werden erst *am Ende* eines Prozessdurchlaufs geändert, Variablen dagegen sofort. Die Abfrage eines Signals nach einer Zuweisung kann daher zu falschen Ergebnissen führen; das Gleiche gilt für mehrfache Signalzuweisungen.

```
process (x)
begin
   z <= z + 1;
   if z >= 10 then
      z <= 0;
   end if;
end process;
```

falsch: Abfrage eines Signals

```
process (x)
   variable y: natural;
begin
   y := y + 1;
   if y >= 10 then y := 0; end if;
   z <= y;
end process;
```

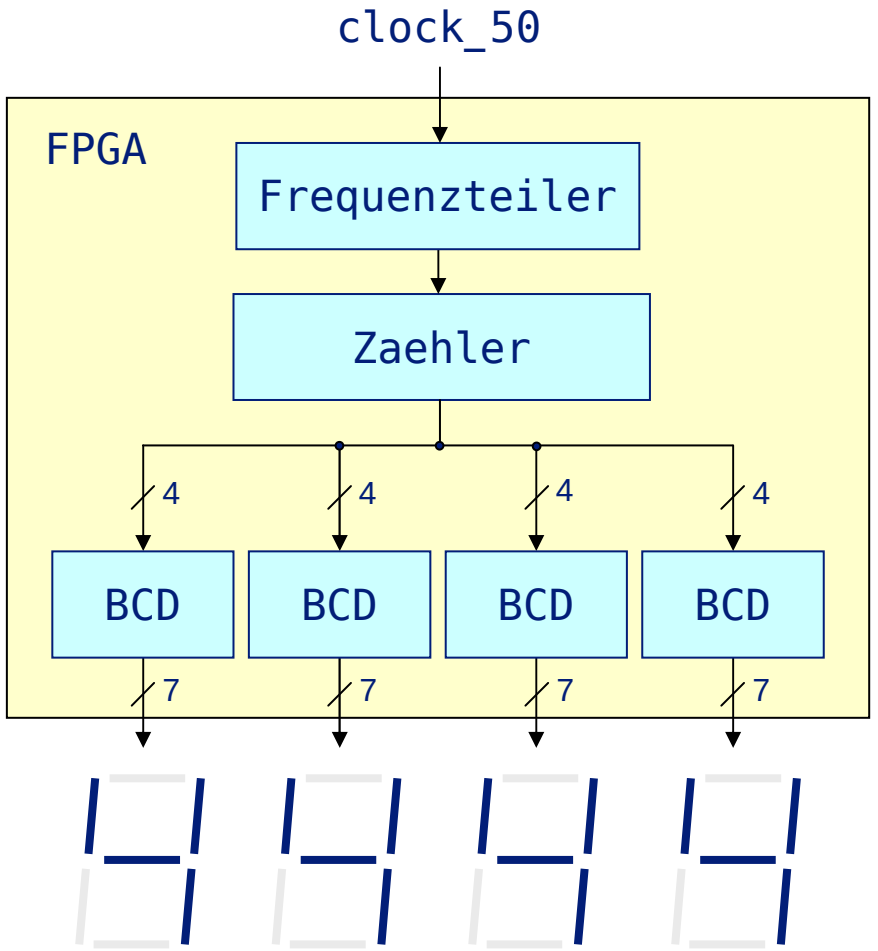
richtig: Abfrage einer Variablen

Versuch 4: Zähler

a) Zähler von 0 bis 9

Realisieren Sie einen Zähler, der die Ziffern 0 bis 9 im Sekundentakt auf den Siebensegmentanzeigen ausgibt.

- Definieren Sie eine Komponente `BCD` (für *Binary Coded Decimal*), welche für ein binäres Eingangssignal die entsprechende Dezimalziffer auf einer Siebensegmentanzeige ausgibt.
- Steuern Sie die Komponenten `BCD` durch einen `Zaehler` an, der mit 1 Hz periodisch von 0 bis 9 zählt.
- Setzen Sie einen generischen Frequenzteiler ein (siehe nächste Seiten) und lassen Sie die Schaltung mit 5 Hz laufen.



HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN FH INGOLSTADT

Versuch 4: Zähler

Generischer Frequenzteiler

```
entity Frequenzteiler is
   generic (n: positive := 1000); -- Taktperiode in Millisekunden
   port (clock_50: in std_logic; takt: buffer std_logic);
end Frequenzteiler;

architecture a of Frequenzteiler is begin
   process (clock_50) variable zyklen: natural := 0;
   begin
      if rising_edge (clock_50) then
         zyklen := zyklen + 1;
         if zyklen >= n * 25000 then
            zyklen := 0;
            takt <= not takt;
         end if;
      end if;
   end process;
end a;
```

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN FH INGOLSTADT

Versuch 4: Zähler

Instantiierung einer generischen Komponente

```
entity FPGA is
   port (clock_50: in std_logic);
end FPGA;

architecture a of FPGA is
    component Frequenzteiler is
       generic (n: positive := 1000); -- Taktperiode in Millisekunden
       port (clock_50: in std_logic; takt: buffer std_logic);
    end component;
   signal t1, t2: std_logic;
begin
   f1: Frequenzteiler generic map (1000) port map (clock_50, t1);
   f2: Frequenzteiler generic map (2000) port map (clock_50, t2);
end a;
```

In diesem Beispiel werden zwei Taktsignale mit Periodendauern von 1 bzw. 2 s erzeugt.

Versuch 4: Zähler

b) Stopuhr

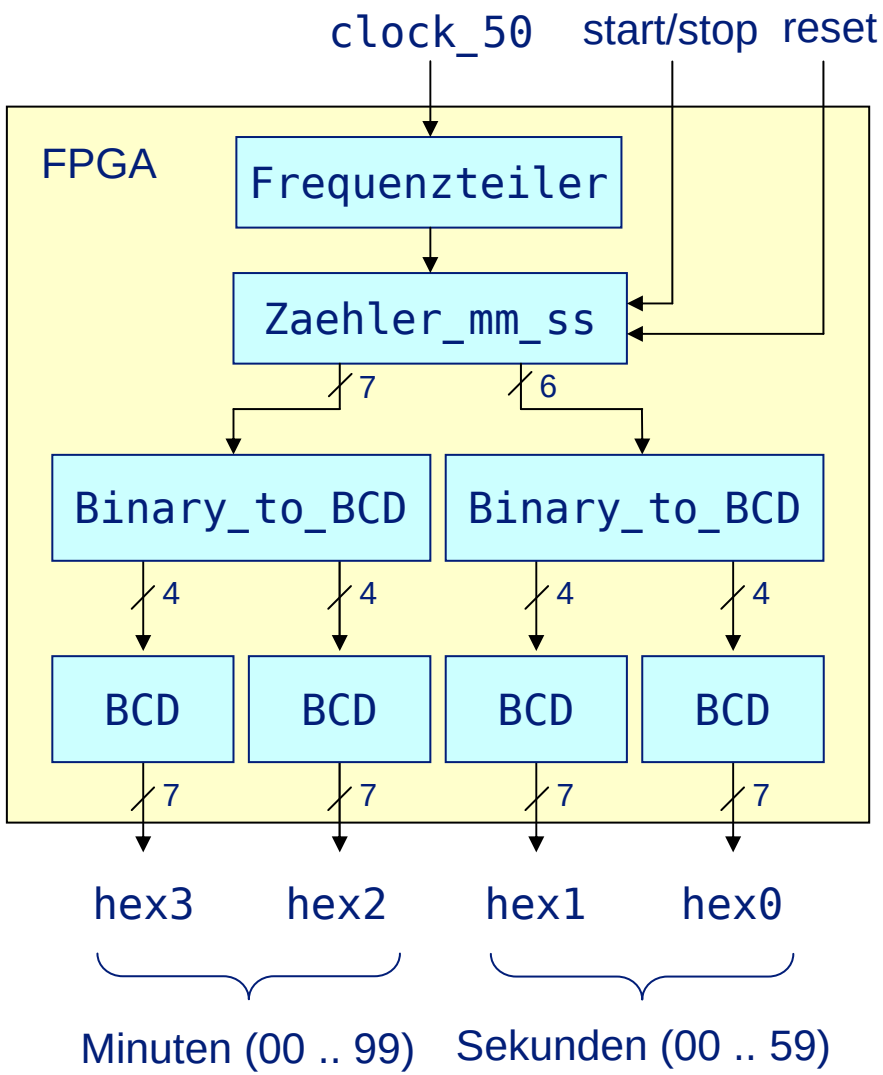
Aufgabe:

Realisieren Sie eine Stopuhr mit Minuten- und Sekundenanzeige. Die Sekunden laufen von 00 bis 59, die Minuten von 00 bis 99.

Verwenden Sie einen Drucktaster als Reset-Signal (hält die Uhr an und stellt sie auf 00:00 zurück).

Verwenden Sie einen zweiten Drucktaster als Start-/Stop-Signal (hält die Uhr an bzw. läßt sie weiterlaufen).

Die Komponente `Binary_to_BCD` können Sie aus dem Intranet herunterladen.

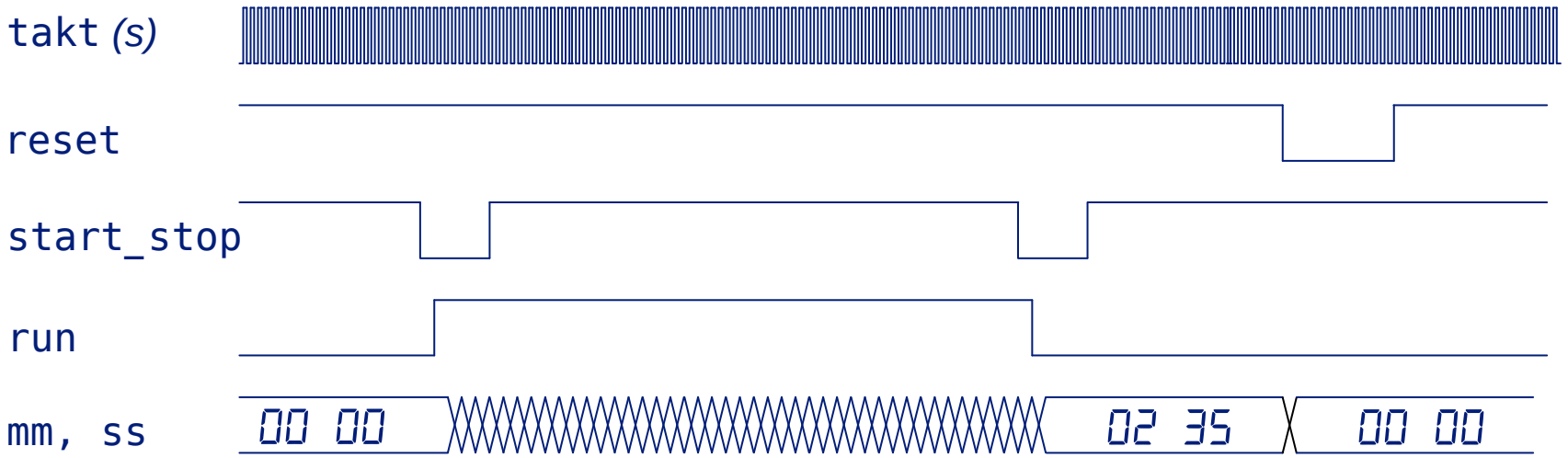


HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN FH INGOLSTADT

Versuch 4: Zähler

b) Stopuhr - Zeitdiagramm

Zeitliche Darstellung einer Signalfolge zur Realisierung der Stopuhr:



VHDL-Realisierung mit zwei parallelen Prozessen:

- `process(reset, start_stop)` erzeugt das Signal `run`
- `process(reset, run, takt)` erzeugt die Signale `mm` und `ss`

Versuch 4: Zähler

c) Messung von Reaktionszeiten

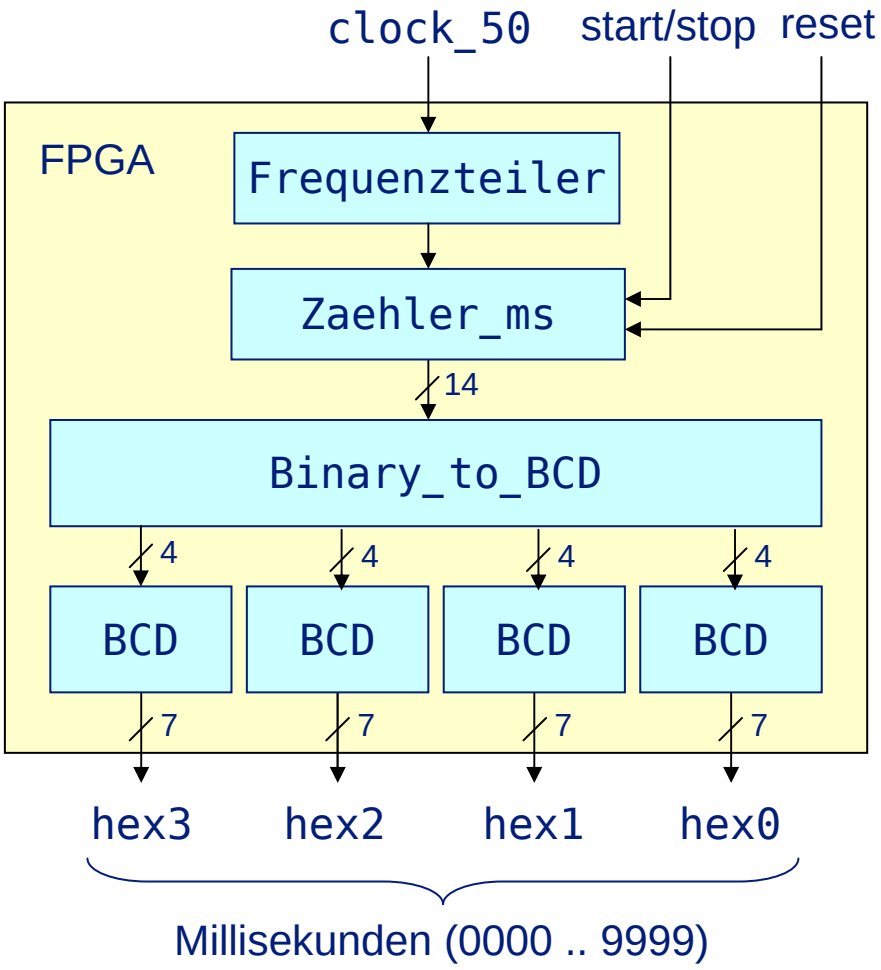
Aufgabe:

Realisieren Sie eine Stopuhr zur Messung der Reaktionszeit in ms.

Verwenden Sie einen Drucktaster als Reset-Signal (hält die Uhr an und stellt sie auf 0000 zurück).

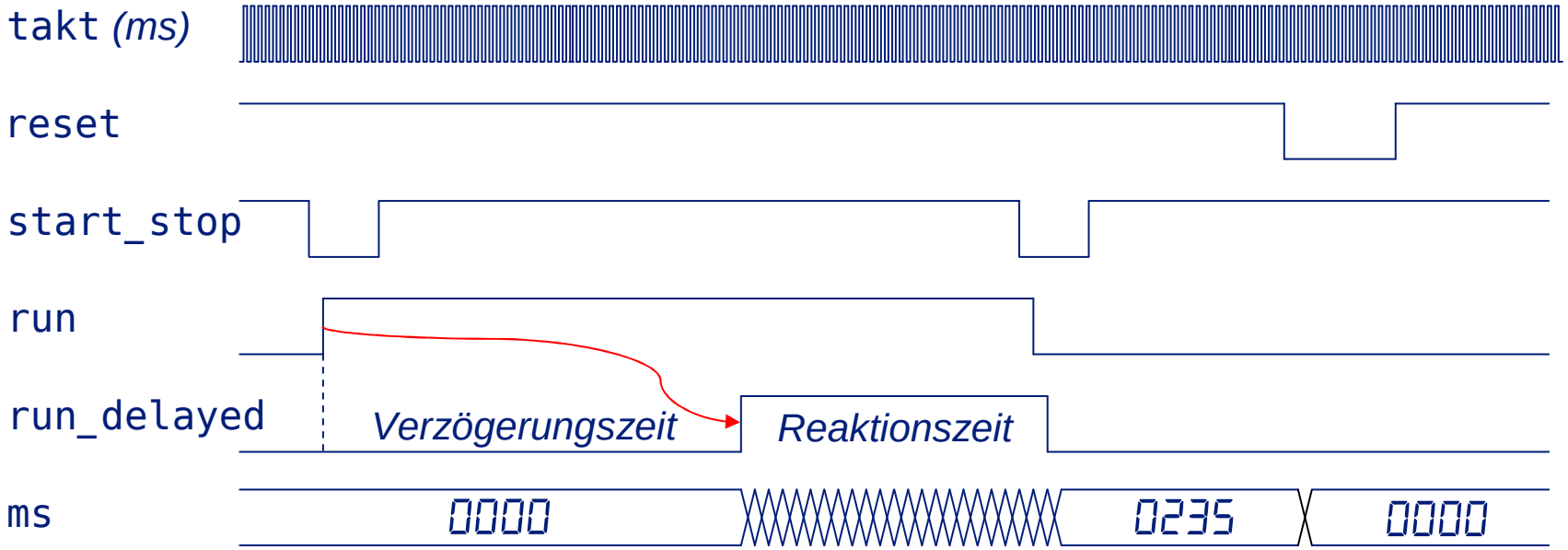
Verwenden Sie einen zweiten Drucktaster als Start-/Stop-Signal. Nach dem ersten Drücken soll die Uhr mit konstanter Verzögerung (1 – 2 s) starten und die Millisekunden hochzählen.

Ein zweites Drücken stoppt die Uhr und zeigt die Reaktionszeit in Millisekunden an.



Versuch 4: Zähler

c) Messung von Reaktionszeiten - Zeitdiagramm



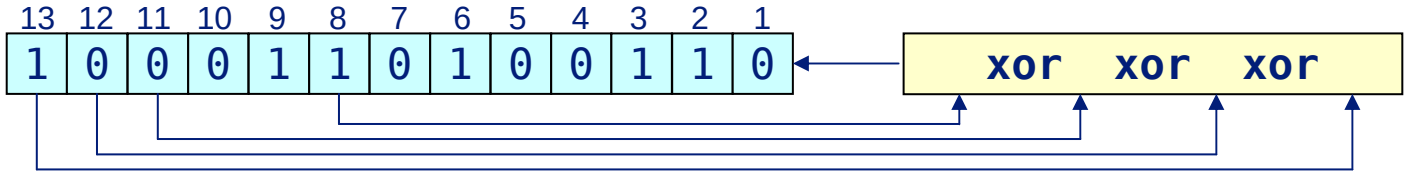
VHDL-Realisierung mit drei parallelen Prozessen:

- `process(reset, start_stop)` erzeugt das Signal `run`
- `process(reset, run, takt)` erzeugt das Signal `run_delayed`
- `process(reset, run_delayed, takt)` erzeugt das Signal `ms`

Versuch 4: Zähler

d) Messung von Reaktionszeiten – Verzögerung mit Zufallszahlen

Wir wollen jetzt die Verzögerungszeit des Stopuhranlaufs nicht mehr fest vorgeben, sondern durch einen Zufallszahlengenerator erzeugen. Solche Generatoren lassen sich z.B. mit rückgekoppelten Schieberegistern aufbauen:



Mit jedem Takt werden alle Bits des Schieberegisters um eine Stelle nach links verschoben; durch die XOR-Verknüpfungen wird für Bit Nr. 1 ein "Zufallsbit" erzeugt.

Wir ändern jetzt die Komponente `Zaehler_ms` so, dass die Verzögerung des Signales `run_delayed` nicht mehr fest vorgegeben wird, sondern so viele Millisekunden beträgt, wie es dem Bitmuster des Schieberegisters entspricht.

Bei einem Schieberegister der Länge 13 *(beachte: Index läuft ab 1, nicht ab 0)* ergeben sich Werte zwischen 0 und $2^{13} - 1 = 8191$; wenn wir diese Werte als Millisekunden interpretieren, dann erhalten wir Verzögerungen zwischen 0 und 8,2 Sekunden.

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN FH INGOLSTADT

Versuch 4: Zähler

d) Messung von Reaktionszeiten – Verzögerung mit Zufallszahlen

Die Komponente `Zaehler_ms` statten wir mit einem Schieberegister aus:

```
signal z: std_logic_vector (13 downto 1);
```

Dieses Register lassen wir in einem weiteren Prozess während der 0-Phase des Signales `run` mit jedem Takt um 1 Stelle nach links schieben, während der 1-Phase bleibt der Registerinhalt dagegen unverändert:

```
process (run, takt) begin
  if run = '0' and rising_edge (takt) then
    z <= z(12 downto 1) & (z(13) xnor z(12) xnor z(11) xnor z(8));
  end if;
end process;
```

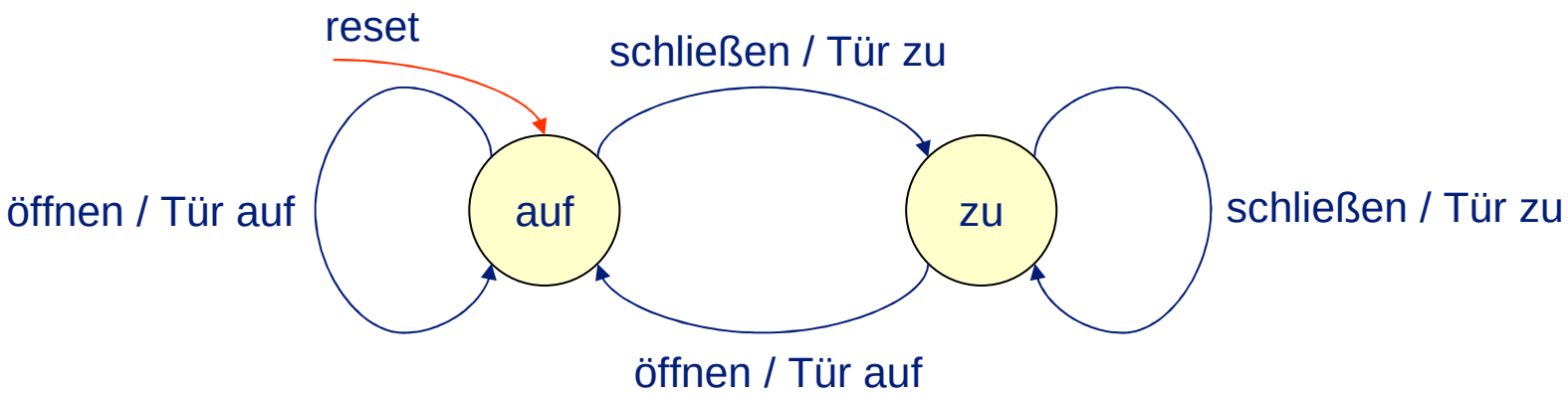
Da die Länge der 0-Phase des Signales `run` vom Eingabeverhalten des Anwenders abhängt, erhalten wir echte Zufallszahlen, nicht nur Pseudo-Zufallszahlen!

Den während der 1-Phase von `run` gleichbleibenden Wert des Schieberegisters `z` können wir nun unmittelbar zur Erzeugung des Signales `run_delayed` nutzen.

Versuch 5: Automaten

Realisierung von Automaten in VHDL

Sequentielle Schaltungen lassen sich effizient als Automaten beschreiben, mit *Zuständen*, *Zustandsübergängen*, *Eingabe-* und *Ausgabesymbolen.* Beispiel:

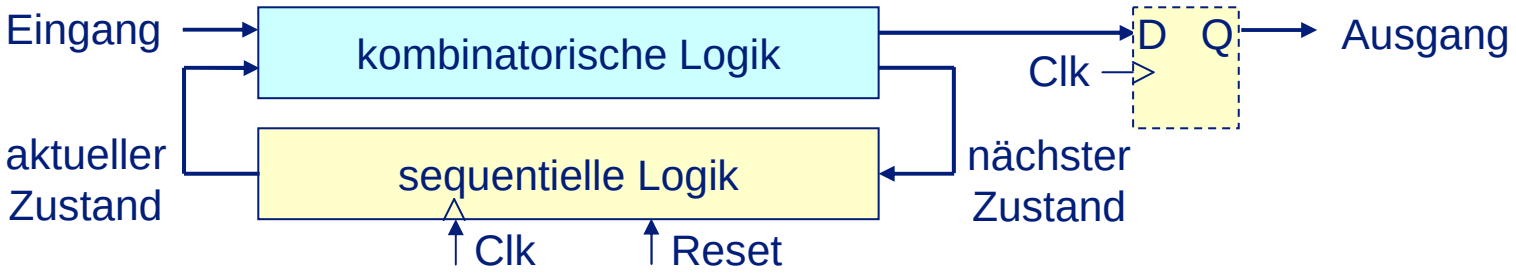


Eine Tür hat zwei Zustände: auf und zu. Die Zustandsübergänge werden durch zwei Eingangssymbole bewirkt: öffnen und schließen. Die Eingabesymbole haben zwei Ausgabesymbole zur Folge: Tür auf und Tür zu. Das Reset-Signal macht die Tür immer auf.

Versuch 5: Automaten

Realisierung von Automaten in VHDL

Bei der schaltungstechnischen Realisierung zerlegen wir den Automaten in zwei Teile:



Die kombinatorische Logik berechnet *asynchron* den Folgezustand des Automaten als Funktion des aktuellen Zustands und der Eingangssignale. Die Ausgangssignale werden ebenfalls asynchron ermittelt und optional durch getaktete Flipflops synchronisiert.

Die sequentielle Logik übernimmt mit der Flanke des Systemtakts den neu berechneten Zustand in eine entsprechende Anzahl von Flipflops; dadurch wird der Zustandsübergang mit dem Takt *synchronisiert.*

In VHDL modellieren wir den Automaten durch zwei parallele Prozesse. Die Automatenzustände werden durch einen *benutzerdefinierten Aufzählungstyp* dargestellt.

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN FH INGOLSTADT

Versuch 5: Automaten

Realisierung von Automaten in VHDL

```
entity Automat is
   port(reset, clock, oeffnen, schliessen: in  std_logic;
        tuer_auf:                          out std_logic);
end Automat;

architecture Verhalten of Automat is
   type zustand is (auf, zu); -- mögliche Zustände der Tür
   signal aktueller_zustand, naechster_zustand: zustand;
begin
   process (all) -- vereinfachte sensitivity list in VHDL 2008
   begin         -- dieser Prozess realisiert die sequentielle Logik
      if reset = '0' then
         aktueller_zustand <= auf;
      elsif rising_edge (clock) then
         aktueller_zustand <= naechster_zustand;
      end if;
   end process;
```

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN FH INGOLSTADT

Versuch 5: Automaten

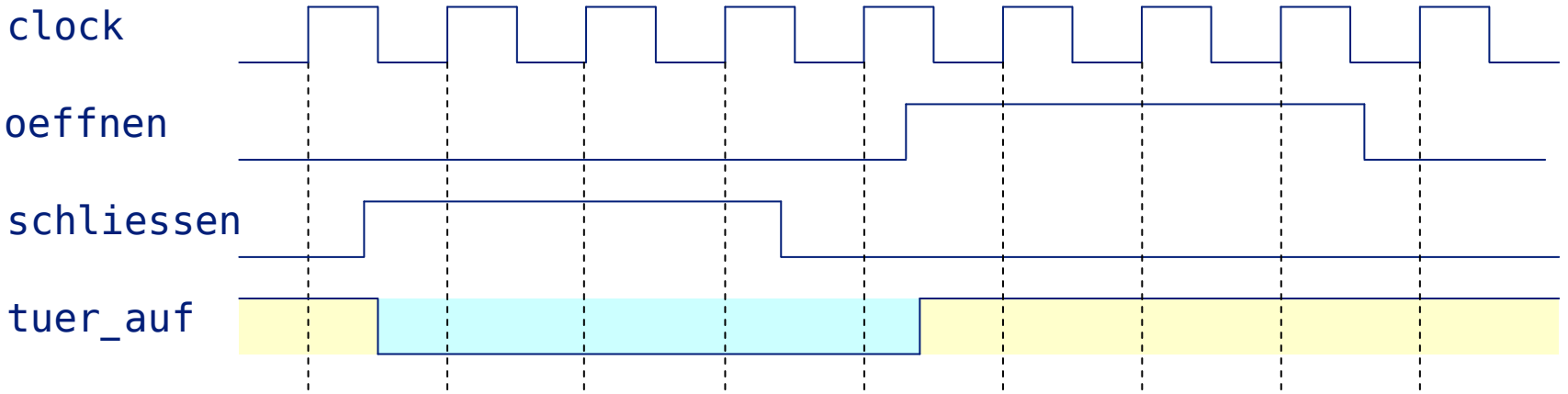
Realisierung von Automaten in VHDL

```
   process (all) begin -- dieser Prozess realisiert die kombinatorische Logik
      case aktueller_zustand is -- betrachte alle Zustands-/Input-
         when auf =>            -- Kombinationen!
            if schliessen then
               tuer_auf <= '0'; naechster_zustand <= zu;
            else
               tuer_auf <= '1'; naechster_zustand <= auf;
            end if;
         when zu =>
            if oeffnen then
               tuer_auf <= '1'; naechster_zustand <= auf;
            else
               tuer_auf <= '0'; naechster_zustand <= zu;
            end if;
      end case;
   end process;
end Verhalten;
```

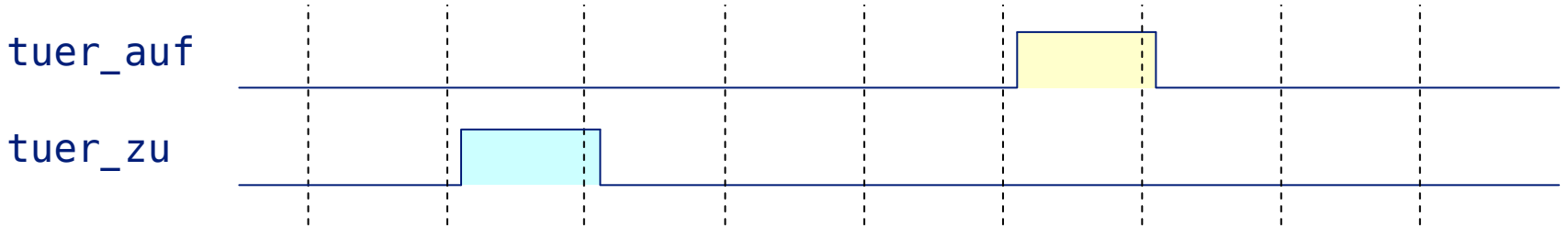
Versuch 5: Automaten

KISS (*"Keep it synchronous, stupid!"*)

Das diskutierte Beispiel funktioniert nur dann, wenn das Ausgangssignal `tuer_auf` seinen Wert beliebig lange halten darf:



In der Praxis müssen aber häufig taktsynchrone, pulsförmige Signale erzeugt werden:



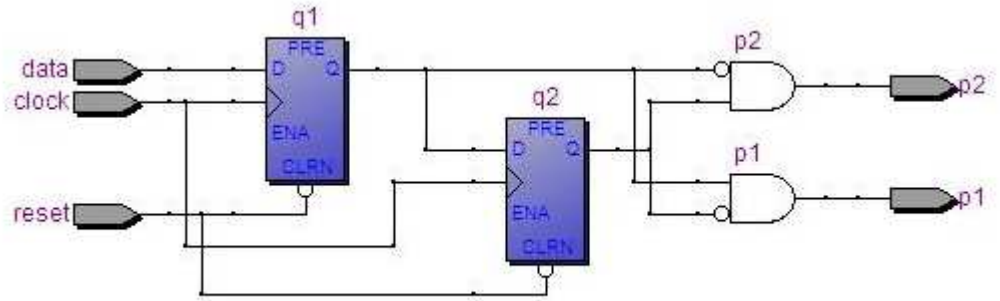
Versuch 5: Automaten

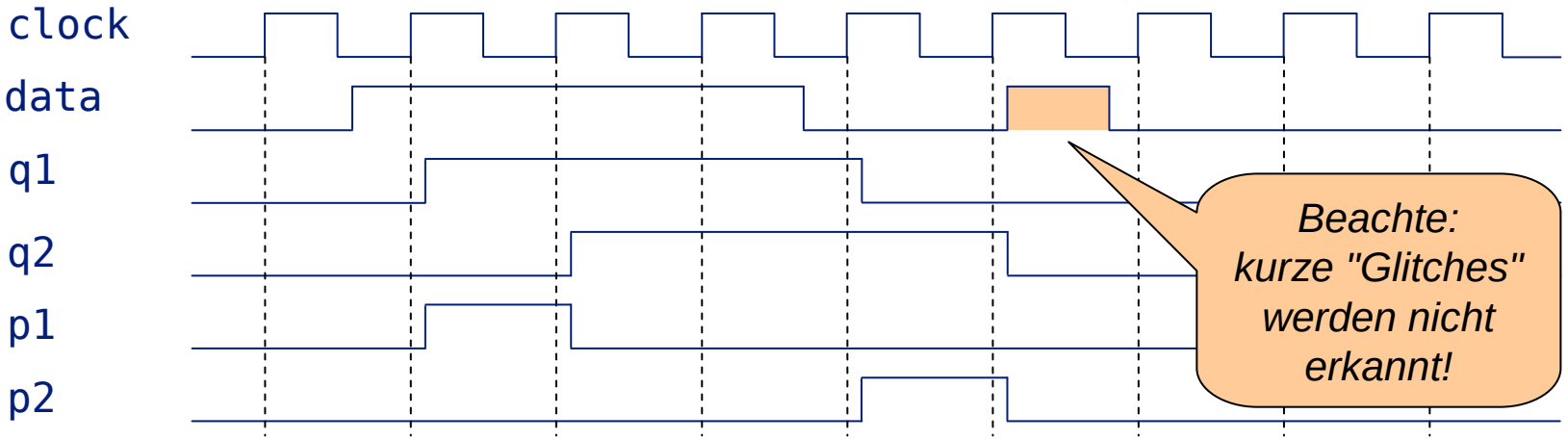
Synchronisation asynchroner Signale und Flankenerkennung

Flipflop q1 synchronisiert das asynchrone Signal mit dem Takt.

Flipflop q2 verzögert das synchronisierte Signal um eine Taktperiode.

Die Gatter erkennen die Flanken.





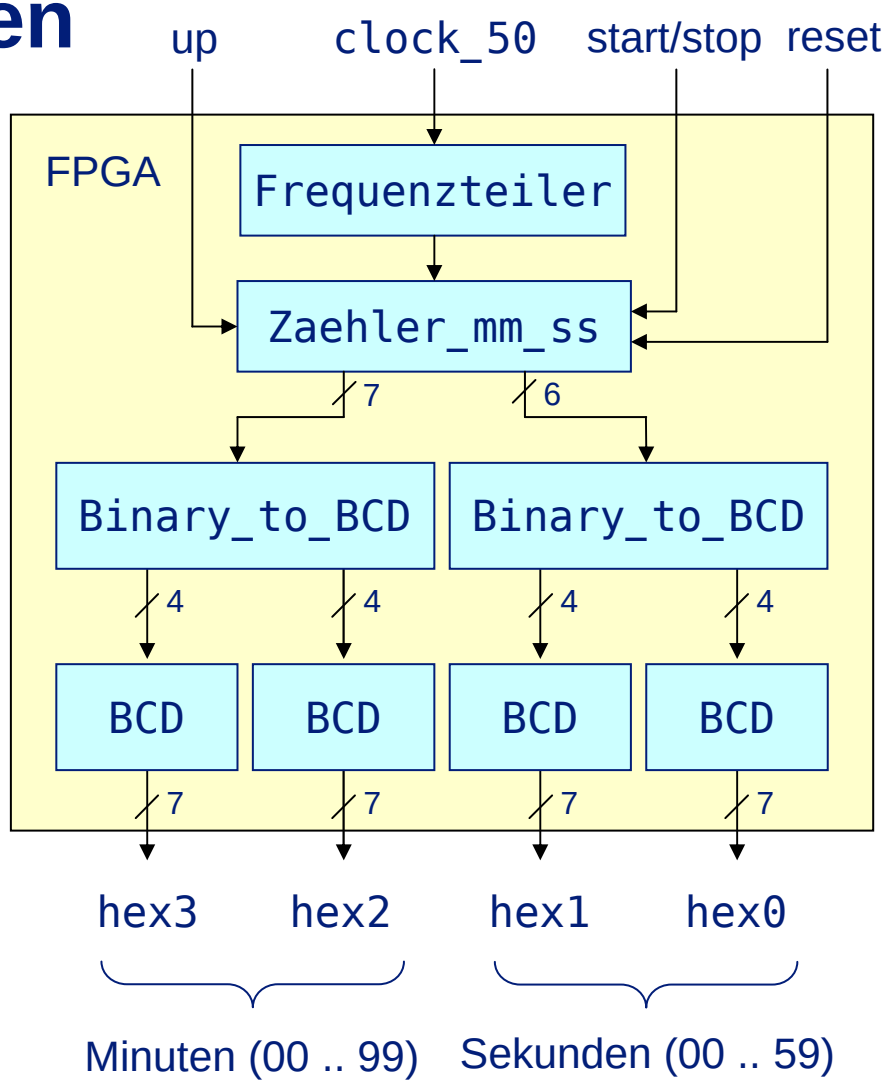
Versuch 5: Automaten

a) Flankenerkennung

Ergänzen Sie die Stopuhr aus Versuch 4b um ein weiteres Signal "up".

Realisieren Sie dieses Signal mit einem Drucktaster. Bei jedem Drücken soll die Minutenanzeige um 1 erhöht werden.

(Damit das Signal sicher erkannt wird, muß der Drucktaster mindestens 1 s lang gedrückt gehalten werden.)



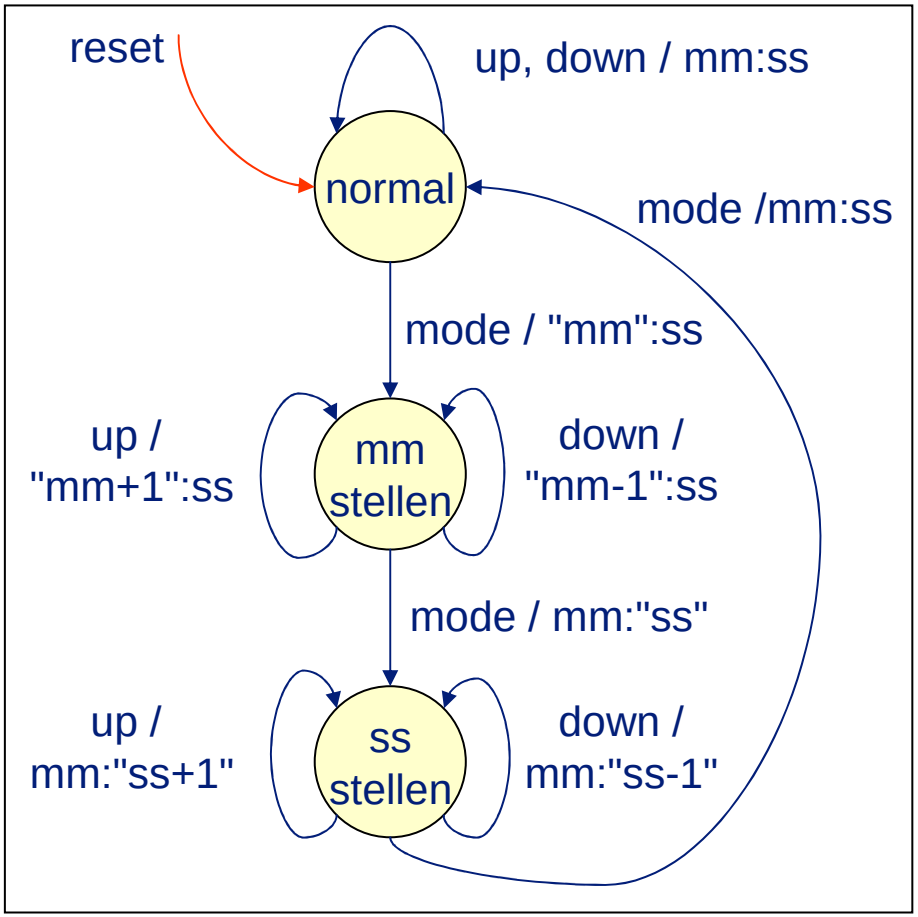
Versuch 5: Automaten

b) Digitaluhr

Eine Digitaluhr habe 3 Zustände: normale Zeitanzeige, Stellen der Minuten und Stellen der Sekunden.

Eingabesymbole:
- mode (schaltet zyklisch die Zustände)
- up (inkrementiert mm oder ss)
- down (dekrementiert mm oder ss)

Ausgabesymbole:
- mm:ss (Uhr läuft)
- "mm":ss (Uhr steht, Minuten blinken)
- "mm+1":ss (mm inkrementieren)
- "mm-1":ss (mm dekrementieren)
- mm:"ss" (Uhr steht, Sekunden blinken)
- mm:"ss+1" (ss inkrementieren)
- mm:"ss-1" (ss dekrementieren)



Versuch 5: Automaten

b) Digitaluhr

Realisieren Sie die Digitaluhr mit 4 Drucktastern für die Signale reset, mode, up und down.

Hinweise:

Realisieren Sie in der Digitaluhr einen endlichen Automaten; dieser Automat sollte von `clock_50` getaktet sein.

Die Ausgabesignale dieses Automaten steuern dann wieder den Minuten-/Sekundenzähler in der Digitaluhr; dieser Zähler läuft im 1s-Takt.

Das Blinken lässt sich durch Dunkeltasten der Siebensegmentanzeigen im Halbsekundentakt realisieren.