Hardw.-Beschr.-Spr. (1/4) Jens Wehrstein

1 Einführung

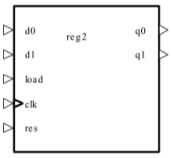
Software-lsg. vs Hardware-lsg.

Software: *Vorteile*: preisgünstige uControler, einfach mit Software zu programmieren
Nachteile: sequentielle Realisierung durch Maschinenbefehle ist langsam
Hardware: *Vorteile*: Parallele Abarbeitung ist schneller als Abarbeitung durch Prozessor
Nachteile: PLDs i.d.R. teurer, Hardwareentwurf schwieriger und aufwendiger
Abstraktionsebenen Transistorebene, Gatterebene (Downloadfähiges Hardwaremodell), Register-Tranfer-Ebene (Synthesefähiges Hardwaremodell -> VHDL-Code), Algorithmische Ebene, Systemebene
Entwurfsablauf *1 Schaltungseingabe* RTL-Beschreibung mit VHDL, Eingabe VHDL-Code
2 Simulation des Designs Simulation für größere Designs sinnvoll, da Fehler leichter in VHDL-Code zu finden sind. *3 Logiksynthese* Umsetzung in Realisierung für bestimmte ASIC- oder PLD-Technologie, Verhalten auf RT-Ebene wird in Struktur auf Gatterebene umgesetzt, nicht alle VHDL-Konstruktionen sind synthesefähig.
4 Platzieren und Verdrahten Nach P&R kann Design auf PLD geladen werden.
EDA-Werkzeuge Logik-Synthese, Simulation, Timing-Analyse, Place&Route
HDLs VHDL, Verilog, SystemC; Technologie ASICs PLDs

2 Grundlegende Konzepte von VHDL

Synthesefähige Beschreibung von Hardware Nur Untermenge von VHDL ist synthesefähig; Empfohlen nur bestimmte Muster zu verwenden; Meist technologieunabhängige Beschreibung auf RTL („Register-Transfer-Level"); Einbau von Makros möglich
Nicht-synthesefähige Beschreibungen Testbenches, Reine Simulationsmodelle
Synthesefähige RTL-Beschreibungen technologieunabh. → auf ASIC- oder PLD umsetzbar; Beschrieben werden getaktete Register mit der richtigen Bitbreite
Desission-Window = Setup-Time + Hold-Time (Δt vor und nach Clock-Flanke)
Entity Schnittestelle der Beschreibung einer Komponente; Definition von Anschlüssen (Ports) und Parametern (Generics); Ports immer std_logic oder std_logic_vector deklarieren

```
ENTITY reg2 IS
   PORT(
      clk  : IN   std_logic;
      d0   : IN   std_logic;
      d1   : IN   std_logic;
      load : IN   std_logic;
      res  : IN   std_logic;
      q0   : OUT  std_logic;
      q1   : OUT  std_logic
   );
END reg2 ;
```



std_logic Hardwaredatentyp für einzelne Bits oder Bit-Vektoren
Architecture beschreib das „Innenleben"; Verhaltensbeschreibung, Strukturbeschreibung; Zu einer Entity können mehrere Architectures gehören.
nebenläufig Ereignisse/Vorgänge die zeitweilig voneinander unabhängig auftreten/ablaufen können; abhängigkeitserzeugende Wirkungszusammenhänge bestehen können
parallel/gleichzeitg wenn zwischen ihnen keine Wirkungszusammenhänge bestehen, die Unabhängigkeit beeinflussen
Prozesse Modellierung von Nebenläufigkeiten; Reihenfolge beliebig; Anweisungen außerhalb ebenfalls nebenläufig, werden wie implizite Prozesse behandelt; innerhalb eines Prozesses sequentielle Ausführung (vgl. uProzessor-Programm); Ausführung des Prozesses, wenn sich Signal in Sensitivitätsliste ändert; Prozesse werden über Signal miteinander verbunden; bei Ausführung Signalen Werte zugewiesen;

```
ARCHITECTURE beh OF reg2 IS
  SIGNAL q0_s, q0_ns, q1_s, q1_ns : std_logic;
BEGIN
  reg: PROCESS (clk, res)
  BEGIN
    IF res = '1' THEN
      q0_s <= '0';
      q1_s <= '0';
    ELSIF clk'event AND clk = '1' THEN
      q0_s <= q0_ns;
      q1_s <= q1_ns;
    END IF;
  END PROCESS reg;

  q0 <= q0_s AFTER 2 ns;
  q1 <= q1_s AFTER 2 ns;

  mux: PROCESS (load, q0_s, q1_s, d0, d1)
  BEGIN
    IF load = '1' THEN
      q0_ns <= d0 AFTER 3 ns;
      q1_ns <= d1 AFTER 3 ns;
    ELSE
      q0_ns <= q0_s AFTER 4 ns;
      q1_ns <= q1_s AFTER 4 ns;
    END IF;
  END PROCESS mux;
END beh;
```

Deklaration der Signale
2 Nebenläufige Anweisungen (implizite Prozesse)
2 Prozesse
reg, mux: Optionale „Labels"
Sensitivitätsliste

Register-Prozesse ein oder mehrere Flipflops, zur Speicherung der Daten; taktflanken-/taktzustandsgesteuert (Latch), meist taktflankengesteuert; Muster einhalten! Folgende Beschreibung z.B.

```
ELSIF clk'event AND clk = '1' AND load = '1' THEN
```

kann simuliert werden, führt aber in der Synthese zu:

```
Error, clock expression should contain only one signal.
```

Kombinatorik/Schaltnetz kann in getakteten Prozess (Prozesse „reg") integriert werden.
Vorteil weniger Prozesse Nachteil bei großen Schaltungen unübersichtlich, aus zugewiesenen Signalen werden immer Flipflops

```
ARCHITECTURE beh1 OF reg2 IS
  SIGNAL q0_s, q1_s : std_logic;
BEGIN
  reg: PROCESS (clk, res)
  BEGIN
    IF res = '1' THEN
      q0_s <= '0';
      q1_s <= '0';
    ELSIF clk'event AND clk = '1' THEN
      IF load = '1' THEN
        q0_s <= d0;
        q1_s <= d1;
      END IF;
    END IF;
  END PROCESS reg;

  q0 <= q0_s AFTER 2 ns;
  q1 <= q1_s AFTER 2 ns;
END beh1;
```

Signale verbinden Prozesse, die im Deklarationsteil der Architecture deklar. werden müssen
Zusammenfassung RT-Verhaltensbeschreibung Beschreibung des Register-Transfer-„Verhalten" der Hardware in Prozessen; RTL-Entwurf: Wo sind die Register? Bitbreite der Register? Wie verhalten sich die Register? Wo sind die Transferfunktionen? Wie verhalten sich die Transferfunktionen?
Strukrubeschreibungen

```
ARCHITECTURE struct OF reg2 IS
  SIGNAL o1 : std_logic;
  SIGNAL o2 : std_logic;
  SIGNAL q0_internal : std_logic;
  SIGNAL q1_internal : std_logic;

  COMPONENT ff2
  PORT (
     clk  : IN   std_logic ;
     d0   : IN   std_logic ;
     d1   : IN   std_logic ;
     res  : IN   std_logic ;
     q0   : OUT  std_logic ;
     q1   : OUT  std_logic
  );
  END COMPONENT;

  COMPONENT mux2
  PORT (
     a1   : IN   std_logic ;
     a2   : IN   std_logic ;
     b1   : IN   std_logic ;
     b2   : IN   std_logic ;
     sel  : IN   std_logic ;
     o1   : OUT  std_logic ;
     o2   : OUT  std_logic
  );
  END COMPONENT;
BEGIN
  I1 : ff2
     PORT MAP (
        clk => clk, d0  => o1, d1  => o2,
        res => res, q0  => q0_internal, q1  => q1_internal
     );

  I0 : mux2
     PORT MAP (
        a1  => d0, a2  => d1, b1  => q0_internal,
        b2  => q1_internal, sel => load, o1  => o1, o2  => o2
     );

  q0 <= q0_internal;
  q1 <= q1_internal;

END struct;
```

Port Map „Positional Association" Nur Angabe des Signals/Ports der Entity/Architecture, Zuordnung zu Komponenten-Port über die Position

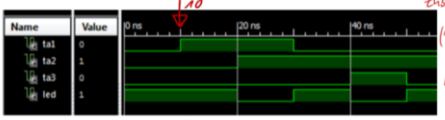
Testbench-Beispiel

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY bspltb IS
END bspltb;

ARCHITECTURE behavior OF bspltb IS

    -- Component Declaration
    COMPONENT bspl
    PORT(
         TA1 : IN  std_logic;
         TA2 : IN  std_logic;
         TA3 : IN  std_logic;
         LED : OUT  std_logic
        );
    END COMPONENT;

   --Inputs
   signal TA1 : std_logic := '0';
   signal TA2 : std_logic := '0';
   signal TA3 : std_logic := '0';

   --Outputs
   signal LED : std_logic;
```

Testbench: E/A
Bezeichner müssen hier identisch sein!
Initialisierung

```
BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: bspl PORT MAP (
            TA1 => TA1,
            TA2 => TA2,
            TA3 => TA3,
            LED => LED
          );

    -- Stimulus process
    stim_proc: process
    begin
        wait for 10 ns;
        TA1 <= '1';
        wait for 10 ns;
        TA2 <= '1';
        wait for 10 ns;
        TA1 <= '0';
        wait for 10 ns;
        TA3 <= '1';
        wait for 10 ns;
        TA3 <= '0';
        wait;(forever)
    end process;

END;
```

Ports
Instanziierung
keine Sensitivitätsliste! (und) Losstartet bei Start des Simulators.
Process
Suspendierung: Stoppen des Prozesses. Zustand des Prozess wird gespeichert. ("Sim. stellt Prozess in Schrank und holt Zettel raus")
wenn nicht, dann beginnt es wieder von vorne



Ausg. angucken: Ist das korrekt? (Erwartete Werte vgl.)

Skizze Testbench:
Testbench
Prozess
DUT Toplvl.

Register/mehrere Flipflops in Beschreibung durch Prozess darf nur Takt und ein evtl. vorhandener asynchroner Set/Reset in der Sensitivitätsliste vorhanden sein
Schaltnetze/Transfer in Beschreibung müssen alle Eingangssignale des Schaltnetzes in der Sensitivitätsliste vorhanden sein

3 Objekte, Datentypen und Operatoren

obj.basierte Sprache → Daten werden in Objekten verwaltet: Konstanten (CONSTANT), Variablen (VARIABLE), Signale (SIGNAL) [alle synth.fähig], ~~Dateien (FILE)~~ [nicht synth.fähig]
Datentypen […], Arrays: muss Enum oder Integer sein sodass synth.fähig.; Indextyp des Arrays ist rechts nach links (Little-Endian);
Feldtypen `SIGNAL breite8 : bit_vector(0 TO 7);` → Indextyp: links nach rechts (Big-Endian); […] `bit_vector(0 DOWNTO 7);` → Indextyp: rechts nach links (Little-Endian)
Zuweisungen `breite8 <= "01101001" -- binaer; breite8 <= b"0110_1001" -- binaer; breite8 <= x"69" -- hexadezimal;`
Operatoren Verkettung: `y <= a & b; [0000 <= 00 & 00] ⇔ y <= '0' & b; [00000 <= 0 & 0000]`
Attribute mit Hilfe ihrer können bestimmte Informationen zu Datentypen gewonnen werden. Bsp: `clk'event` liefert boolean zurück wenn auf Signal `clk` ein Ereignis stattfand
Mehrwertige Logik unbekannt Zwei Gatter treiben unterschiedliche Logikwerte auf das gleich Signal ('X'); hochimpedant Ein Signal wird nicht niederohmig getrieben ('Z')
schwache Logikpegel Mit Pull-Up- oder Pull-Down- Widerständen getrieben (weak, '0', '1' : 'L' bzw. 'H'); irrelevant Logikwert ist irrelevant (don't care: '-')
Auflösungsfkt bestimmt den resultierenden Wert bei einer Signalzuweisung (implizit)

```
PACKAGE std_logic_1164 IS
    ------------------------------------------------
    -- logic state system  (unresolved)
    ------------------------------------------------
    TYPE std_ulogic IS ( 'U',  -- Uninitialized
                         'X',  -- Forcing  Unknown
                         '0',  -- Forcing  0
                         '1',  -- Forcing  1
                         'Z',  -- High Impedance
                         'W',  -- Weak     Unknown
                         'L',  -- Weak     0
                         'H',  -- Weak     1
                         '-'   -- Don't care
                       );
```

'X': Aufeinandertreffen von mehreren Treibern; 'H': Pull up Widerstand; 'L': Pull down Widerstand
'U': uninitialisiert → Inititaliserungswert; In der Beschreibung nur '0', '1' und 'Z' verwenden
Arithmetik in VHDL Addition, Subtraktion, Mulitplikation synthesefähig; integer kein Hardwaredatentyp!
→ unsigned und signed verwenden (std_logic_vector), damit kann die Bitbreite explizit spezifiziert werden
Division: Divisor muss 2er-Potenz sein! (n/Divisor); `q_s <= "1001" -- dezimal 9 ;` numeric_std verwenden
Typkonversionen std_logic_vector und signed/unsigned sind verwandt → Typ Cast zulässig
integer und signed/unsigned/std_logic_vector sind nicht verwandt → Konversionsfunktionen notwendig

```
to_integer( unsigned(a) ) ; std_logic_vector( to_unsigned(s_int, 4) ); -- 2. Arg Bitbreite
```

Konversionsfunktionen aus Package numeric_std:

```
function TO_INTEGER (ARG: UNSIGNED) return NATURAL;
function TO_INTEGER (ARG: SIGNED) return INTEGER;
function TO_UNSIGNED (ARG, SIZE: NATURAL) return UNSIGNED;
function TO_SIGNED (ARG: INTEGER; SIZE: NATURAL) return SIGNED; -- interger got size of 4
```

Operatoren für Hardware-Datentypen Bei Multiplikation darf y nur doppelt so breit sein wie die Edukte/Faktoren: `y <= a * b ; [6 Bit = 3 Bit * 3 Bit]`
Boolsche Operationen and, nand, or, nor, xor, xnor, not; Bsp: `y <= a and b;` y, a, b müssen vom gl. Datentyp sein. Vektoren: Breiten müssen übereinstimmen, Anwendg: bitweise
Namensgleichheit Das hierarchisch tiefer stehende Objekt maskiert das höher stehende

4 Sequentielle und nebenläufige Anweisungen

if-Anweisung Auchtung! Wenn 2 if-Anweisungen nach einander folgen und die 2. wahr ist kann sie sie die erste überschreiben (vgl. C-Syntax) → Pro Signal nur eine if-Anweisung!
case-Anweisung Bei 3 Bit → 9 Möglichgeiten → 9^3 Fälle = 729 Fälle;

```
PROCESS(sel)
  BEGIN
    CASE sel IS
       WHEN "000" =>  y <= '0';
       WHEN "001" =>  y <= '1';
       WHEN "010" =>  y <= '1';
       WHEN "011" =>  y <= '0';
       WHEN "100" =>  y <= '1';
       WHEN "101" | "110" | "111" =>  y <= '0';
       WHEN OTHERS => y <= 'X';
    END CASE;
  END PROCESS;
```

ODER!
Kann man Fälle zusammenfassen!
Mehrere Werte können durch | (Oder) zusammengefasst werden.

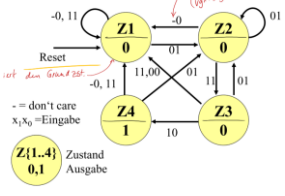
Schleifen `[Schleifenmarke :] [Iter.schema] LOOP [Sequ. Anweisungen] END LOOP [Schleifenmarke];`
`while [boolscher Ausdruck] :`
`for [Laufvariable] IN [Range] :`
Schleife kann durch EXIT verlassen werden, mit NEXT zum Schleifenkopf gesprungen werden
`exit [Schleifenmarke] [WHEN boolscher Ausdruck];`
`next [Schleifenmarke] [WHEN boolscher Ausdruck];`
Dynamische/Endlosschleife nicht möglich
Wait nur in Prozessen ohne Sensitivität
`wait on [Signal_1] {, Signal_n};`
→ entspricht Sensitivitätliste (nicht synthesefähig)
`wait until [booleschер_Audruck];`
(synthesefähig, wenn Takt abgefragt wird)
null → entspricht NOP-Anweisung

wait for `[Zeitangabe];` → Prozess suspendiert Zeit (nicht synthesefähig)

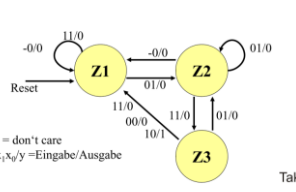
```
assert [boolscher_Ausdruck]
   report „String"
   severity (note | warning | error | failure); -- wird in Testbenches benutzt
```

Nebenläufige Anweisungen unbedingte nebenläufige Anweisung
`[Marke :] Signal <= [TRANSPORT] Ausdruck [AFTER Zeitangabe] {, Ausdruck AFTER Zeitangabe} ;` (Signalzuweisung, synthesefähig ohne optionale Wiederholung)
bedingte nebenläufige Anweisung (when) äquivalent zu IF, synthesefähig; Selektierte nebenläufige Anweisung (with) äquivalent zu CASE, synthesefähig
→ Nebenläufige Anweisungen sind implizite Prozesse, haben bzgl. Simulation und Synthese keine Vorteile! Verlust der Übersicht, u.U. mehr Prozesse → sparsam verwenden
Schaltwerke endliche Automaten, im Vgl. zu Schalnetzen speicherndes Verhalten
Zustandsspeicher wird bei synchronen Schaltwerken üblicherweise mit taktgesteuerten Flipflops realisert (flankengesteuert – üblicherweise: steigende Flanke)
Kreise: Zustände, Pfeile: Zustandsübergänge (auf Pfeil stehen Eingangsbelegungen der Eingangssignale)
Moore: vom Zustand abhängig; Pro Zustand nur eine Ausgabe möglich! Mealy: vom Zustand und Eingang abhängig; Pro Zustand mehrere Ausgaben möglich!
→ beide: Folgezustand wird mit nächstem Taktschritt eingenommen

Beispiel Moore-Automat

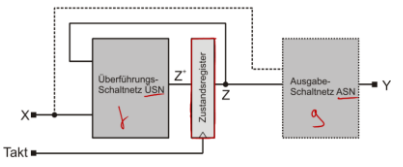


Beispiel Mealy-Automat



Digitale Implementierung von Schaltwerken

f berechnet Folgezustand, *g* die Ausgabe,
Zustandsregister mit *k* Flipflops speichert den Zustand



Binäre Zustandscodierung Zuweisung binärer Codes zu symbolischen Zuständen; Bei Schaltwerk mit *n* Zuständen sind $k \geq \lceil \log_2 n \rceil$ Bits für das Zustandsregister notwendig
Bool'sche Funktionen des ÜSN und ASN hängen von Zustandscodierung ab; entscheidend für Ressourcenverbrauch und maximale Taktrate des Schaltwerks
Übliche Zustandscodierung Binärcode (für n=3: 00,01,10) / 1-aus-N-Code „One-Hot-Code" (für n=3: 001, 010, 100) / Gray-Code (Hamming-dist.=1; für n=3: 00, 01, 11)
Beispiel Automat: Geldwechsler

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY geld_wechsler IS
PORT(
    clk: IN std_logic;
    reset : IN std_logic;
    e: IN std_logic_vector (1 DOWNTO 0);
    a: OUT std_logic_vector (1 DOWNTO 0) );
END;

ARCHITECTURE beh OF geld_wechsler IS
    -- Deklaration des Aufzählungstyps
    TYPE SW1 IS (S0, S1, S2);
    -- Zustandsregister des Schaltwerks als Signale
    SIGNAL q_s, q_ns: SW1;
BEGIN
    -- Zustandsregister
    reg: PROCESS (clk, reset)
    BEGIN
        IF reset = '1' THEN
            q_s <= S0;
        ELSIF (clk'event AND clk='1') THEN
            q_s <= q_ns;
        END IF;
    END PROCESS;
```

```
-- ÜSN
nsd: PROCESS (q_s,e)
BEGIN
    CASE q_s IS  -- Bestimmung des nächsten Zustands
        WHEN S0 =>
            CASE e IS -- geschachteltes case
                WHEN "00" => q_ns <= S0;
                WHEN "01" => q_ns <= S1;
                WHEN "10" => q_ns <= S2;
                WHEN "11" => q_ns <= S0;
                WHEN OTHERS => NULL;
            END CASE;
        WHEN S1 =>
            q_ns <= S0; -- default
            IF e="01" THEN
                q_ns <= S2;
            END IF;
        WHEN S2 =>
            q_ns <= S0;
    END CASE;
END PROCESS;
```

```
-- ASN, Mealy SW
od: PROCESS (q_s,e)
BEGIN
    CASE q_s IS
        WHEN S0 =>
            CASE e IS
                WHEN "00" => a <= "11";
                WHEN "01" => a <= "00";
                WHEN "10" => a <= "00";
                WHEN "11" => a <= "00";
                WHEN OTHERS => NULL;
            END CASE;
        WHEN S1 =>
            CASE e IS
                WHEN "00" => a <= "11";
                WHEN "01" => a <= "00";
                WHEN "10" => a <= "11";
                WHEN "11" => a <= "01";
                WHEN OTHERS => NULL;
            END CASE;
        WHEN S2 =>
            a <= "11";
            IF e="11" THEN
                a <= "10";
            END IF;
    END CASE;
END PROCESS;
END beh;
```

Hardw.-Beschr.-Spr. (2/4) Jens Wehrstein

4 Sequentielle und nebenläufige Anweisungen

Zustandscodierung von Schaltwerken kann Ressourcenbedarf im PLD, Maximale Taktfrequenz, mit der das SW betrieben werden kann beeinflussen.
Alle Zustände sollten benutzt werden um „aufhängen“ → lock-up zu verhindern.
Getaktete Prozesse beschreiben Flipflops/Register; Pro Signal, dem etwas zugewiesen wird impliziert getakteter Prozess ein Flipflop/Register.
Beschreibung mehrerer Flipflops/Register in einem Prozess möglich; In getakteten Prozessen keine Variablen verwenden!
Nur einen Takt verwenden → Synchronität! Datensignale NIEMALS als Takt verwenden! Nur synchrone Schaltungen sind synthesefähig!
Flankenabfrage Getakteter Prozess reagiert auf steigende oder fallende Flanke. In Flankenabfrage nur ein Signal verwenden!
Sensitivitätsliste Nur Takt, sowie evtl. weitere asynchrone Setz- oder Rücksetzsignale
Initialisierung Setzen auf ‘1’ o.ä. nicht auf Hardware relevant (wird auf Hardware nicht umgesetzt); nur durch a-/synchrones Setzen/Rücksetzen Ports nicht zur Beschreibung von Flipflops nehmen!

Beispiel für synchronen und asynchronen Reset

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity reset_test is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           d0 : in  STD_LOGIC;
           d1 : in  STD_LOGIC;
           q0 : out  STD_LOGIC;
           q1 : out  STD_LOGIC);
end reset_test;
```

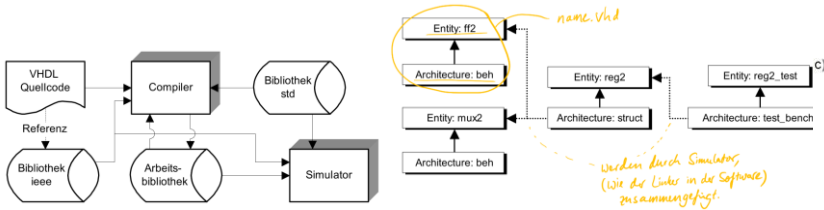
```
architecture beh of reset_test is

begin
   process(clk, reset)
   begin
      if reset = '1' then -- Asynchroner Reset
         q0 <= '0';
      elsif clk'event AND clk = '1' then
         q0 <= d0;
      end if;
   end process;

   process(clk)
   begin
      if clk'event AND clk = '1' then
         if reset = '1' then  -- Synchroner Reset
            q1 <= '0';
         else
            q1 <= d1;
         end if;
      end if;
   end process;
end beh;
```

Kombinatorische Prozesse beschreiben Schaltnetze → kein speich. Verh. beschr.! → Takt nicht verwenden! → pegelgesteuert!
Sensitiv.listenfehler kein Error →Alle Signale die verwendet werden in Sens.liste eintragen!

5 Simulation, Signal und Variable

Simulation von VHDL-Modellen Bevor Design simuliert werden kann muss Kompilat aus Quellcode erstellt werden; → Compiler
Jedes Kompilat wird in Library angelegt: über logischen Namen referenziert (z.B: ieee), über „Mapping“ im Simulator mit Dateipfad auf Festplatte verbunden

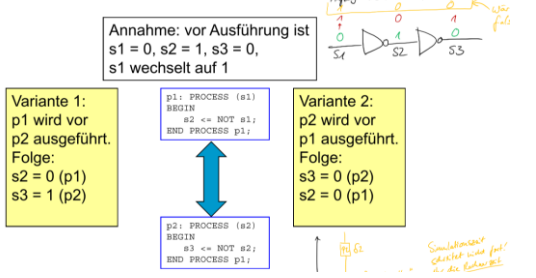
Übersetzungseinheiten Entity, Architecture, Configuration, Package (unterteilt in Header und Body)

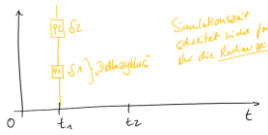


Compiler und Simulator Wird keine Arbeitsbibiliothek angegeben → „work“; Deklarationsregel Der Wert, der am weitesten links steht
Auflösungsfunktion gibt an was passiert wenn 2 Ausgänge auf dasselbe Signal treiben → Entscheidung welcher Zustand angenommen wird
Simulator Elaboration des VHDL-Modells (vgl. Linker) Jede Komponente wir durch zugeordnete Architecture ersetzt (Kompilat)
Entstehung eines Modells aus über Signale verbundenen Prozessen; Einsetzen der „Generics“-Parameter; Reservierung von Speicherplatz
Initialisierung (t = 0 s) Initialisierung von Signalen und Variablen; Jeder Prozess wird einmal ausgeführt Ausführung

Reihenfolge beim Ausführen der Prozesse: unterschiedliches Ergebnis

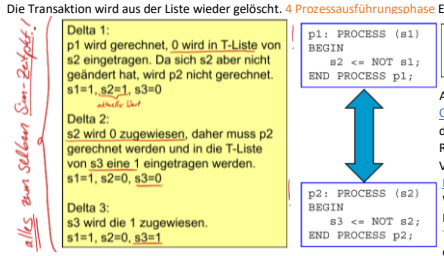
(Achtung: Simulator arbeitet so gerastert. Hier zum Anfangen des Problems)





Ergebnisgesteuerte Simulation Ergebnis
- muss unabh. von Reihenfolge sein
Zuweisung der Signal erfolgt während
- Ausführungsphase der Prozess zunächst in Transaktionslisten → andere Prozesse rechnen mit „altem“ Wert
- Erst nach Ausführung der Prozesse erfolgt ggf. Zuweisung der Werte aus Transaktionslisten an Signale
- Jeder Modellzeitpkt. wird in mehrere „Deltazyklen“ aufgeteilt

Signale und Transaktionslisten Für jedes Signal wird im Simulator eine Transaktionsliste verwaltet.
Prozesse sind Treiber für Signale, da sie den Signalen Werte zuweisen
Eine Zuweisung an ein Signal (durch einen Treiber) in der Zukunft wird in der Transaktionsliste eingetragen, z.B: `q0 <= q0_s AFTER 2 ns;`
Eine Transaktion ist die Voraussetzung für ein zukünftiges Ereignis (Ereignis: Wechsel im Wert des Signals)
Transaktionen können auch wieder gelöscht oder überschrieben werden
Ausführung des Simulators 1 Initiale Prozessausführungsphase: alle Prozess werden durchgeführt; Eintragen von Transaktionen in die –listen.
2 Durchsuchen der Transaktions- und Wecklisten nach dem nächsten Zeitpunkt. (Kann auch der gleiche Modell-Zeitpunkt sein → $\delta = 0\ s$)
3 Signalzuweisung Zuweisen einer Transaktion an das Signal: Unterscheidet sich der Wert des Signals vom vorherigen Wert → Ereignis liegt vor.
Die Transaktion wird aus der Liste wieder gelöscht. 4 Prozessausführungsphase Eintragen der Transaktionen in die Transaktionslisten 5 Gehe zu 2.



Annahme: vor Ausführung ist s1 = 0, s2 = 1, s3 = 0, s1 wechselt auf 1 (Ereignis)

Verzögerungszeiten nicht sy.fhg. → Nur zur Veranschaulichung
„Zero-Delay“ Oszillationen Modell oszilliert im Zeitpunkt t=0s
Anzahl der maximalen Iterationen im Simulator einstellbar, bis Error
Oszillationen in synchronen HW-Beschr. kann dies nicht autreten, da Rückkopplung durch Registerprozesse aufgetrennt werden.
Register nur auf Takt sensitiv, nie auf Daten; Testbench sind Verzögerungszeiten vorhanden.
Modellierung von Verzögerungszeiten Transportverzögerung Alle Wertzuweisungen werden mit der spezifiziert. Zeit wiedergegeben
Modelliert die Laufzeit auf Leitungen `q0_ns <= TRANSPORT d0`
Trägheitsverzögerung Kurze Pulse werden unterdrückt; erfolgt durch Löschen von Transaktionen aus Transaktionsliste
modelliert die träge Verzögerung von Gattern: `q0_ns <= d0 AFTER 3 ns; q0_ns <= INERTIAL d0 AFTER 3 ns;`
Getrennte Angabe von Trägheit und Transport: `q0_ns <= REJECT 2 ns INERTIAL d0 AFTER 3 ns;`
Signale und Variablen Die Wertezuweisung zu Signalen in Prozessen erfolgt nicht sofort, sondern wird zunächst in Transaktionsliste eingetragen
Hat Konsequenzen wen dieses Signal im gleichen Prozess als Quelle verwendet wird → Verwendung von Variablen (Wertzuweisung sofort)
(statische/globale) Variablen Entsprechen C; Achtung! Zuweisung: “ := “; werden nur einmal in der Initialisierungsphase ausgeführt; Werte bleiben nach Ausführen des Prozesses erhalten (sind statische Variablen, Initialisierung nur einmal, s.o.);
Globale Variablen seit 1993 auch verwendbar, aber nicht synthesefähig!
Mehrfachzuweisung von Signalen Werden einem Signal in einem Prozess mehrfach Werte zugewiesen (zum selben Zeitpkt.), so werden die alten Einträge in der Transaktionsliste wieder gelöscht:

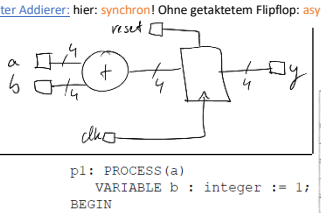
```
test: PROCESS (a,b)
 BEGIN
   b <= a + 3;    -- Addiere a plus 3
   b <= a + 2;    -- Addiere a plus 2
   c <= b * 2;    -- Multipliziere b mit 2
 END PROCESS test;
```

Diese Zeile wird de facto nie ausgeführt!

Bibliotheken/Libraries

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

Getakteter Addierer: hier: synchron! Ohne getaktetem Flipflop: asynchron!




```
p1: PROCESS(a)
    VARIABLE b : integer := 1;
BEGIN
    b := b + 1;
    a <= a * b;
END PROCESS p1;

y <= a;
```

Deltazyklen-Bsp.:

Deltazyklus	Signal a	Transaktions-liste von a	Port y
S: 0ns	1		-2^{31}
P: 0ns	1	2	-2^{24}
S: 0ns + 1δ	2		1
P: 0ns + 1δ	2	6	1
S: 0ns + 2δ	6		2
P: 0ns + 2δ	6	24	2
S: 0ns + 3δ	24		6
P: 0ns + 3δ	24	120	6
S: 0ns + 4δ	120		24
P: 0ns + 4δ	120	720	24

- Notizen zu Altklausuren

Moore-Automat mit ‘1’ - dominantem Signal:
Dominanz: Wenn `reset_req = ‘1‘`, dann `req_out <= ‘0‘;`

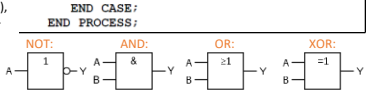
```
ARCHITECTURE beh OF a4 IS
   TYPE state_t IS (WAIT_REQ, REQUEST);
   SIGNAL q_s, q_ns : state_t;

BEGIN

   reg: PROCESS (clk, reset)
   BEGIN
      IF reset = '0' THEN
         q_s <= WAIT_REQ;
      ELSIF (clk'event AND clk='1') THEN
         q_s <= q_ns;
      END IF;
   END PROCESS;

   comb: PROCESS(q_s, req_in, reset_req)
   BEGIN
      q_ns <= q_s;
      CASE q_s IS
         WHEN WAIT_REQ =>
            req_out <= '0';
            IF req_in = '1' AND reset_req = '0' THEN
               q_ns <= REQUEST;
            END IF;
         WHEN REQUEST =>
            req_out <= '1';
            IF reset_req = '1' THEN
               q_ns <= WAIT_REQ;
            END IF;
      END CASE;
   END PROCESS;
```

Synthesefähige Objektklassen:
synthesefähig: „Variable“, „Konstanten (constant)“
nicht synthesefähig: „Datei (file)“

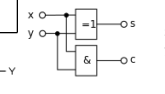
Zero-Delay-Oscillation:
Es handelt sich um eine „Zero-Delay-Oscillation“.
Der Prozess treibt ein Signal, auf welches er auch sensitiv ist (Rückkopplung). Durch die Rückkopplung kommt es zur Oszillation. Der Prozess wird in jedem Deltazyklus ausgeführt.
→ Der Programmierer kommt in die digitale Hölle

Latch: Freie Rückführung durch Signal → keine synchrone Schaltung, da freie Rückführungen nicht erlaubt
→ Der Programmierer kommt in die digitale Hölle
kein Latch: synchron! [RTLs drunter malen] Latch: asynchron!

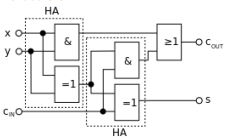
Unvollständige if-Anweisung: In Hardware entstehen Latches, führt zu Timing Problemen.
Hardware unterscheidet sich vom Verh. der Simulation.
→ Der Programmierer kommt in die digitale Hölle



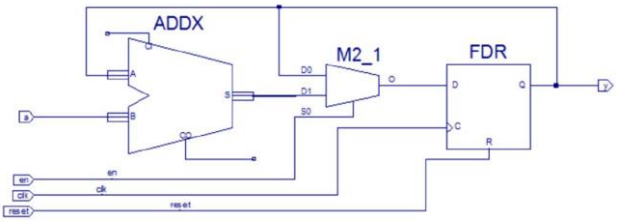
Gatter/Addierer:
Halbaddierer:



Volladdierer:



c) Zeichnen Sie bitte die Register-Transfer-Struktur der Schaltung auf (nach Korrektur der Fehler). Wieviele Flipflops sind bei der Synthese zu erwarten? (3 P) Lösung siehe Bild, 4 Flipflops:



Moore-Schaltwerk mit einer Kombinatorik kurzgehalten mit Default und nur Ausgabe der veränderten:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
      ENTITY a4 IS
                  PORT (    clk,reset,en: IN std_logic;
                                y0,y1 : OUT std_logic);
      END;

ARCHITECTURE beh OF a4 IS
      TYPE type_sreg IS (STATE0,STATE1,STATE2);
SIGNAL sreg, next_sreg : type_sreg;

BEGIN
reg: PROCESS (clk, reset)

BEGIN
      IF ( reset='1' ) THEN
            sreg <= STATE0;
            ELSIF clk='1' AND clk'event THEN
            sreg <= next_sreg;
      END IF;
END PROCESS reg;


comb: PROCESS (sreg,en)

BEGIN
      y0 <= '0';                          -- Default-
      y1 <= '0';                          -- Einweisungen
      next_sreg <= sreg;                  -- spart Code

CASE sreg IS

      WHEN STATE0 =>
                  IF (en = '1') THEN      -- Übergang zum
                  next_sreg <= STATE1;    -- nächsten State
      END IF;

      WHEN STATE1 =>
                  y0 <= '1';              -- Ausgabe nur des neuen Werts am Output
                  IF (en = '1') THEN      -- Übergang zum
                  next_sreg <= STATE2;    -- nächsten State
      END IF;

      WHEN STATE2 =>
                  y1 <= '1';              -- Ausgabe nur des neuen Werts am Output
                  IF (en = '1') THEN      -- Übergang zum
                  next_sreg <= STATE0;    -- nächsten State
      END IF;

      WHEN OTHERS =>                      -- oder      WHEN OTHERS => NULL;

END CASE;

END PROCESS comb;

END beh;
```

Type-Casts:
... = signal / variable / (vielleicht definierter Enum (type)) von einem Aufzählungsdatentyp:

```
signed(...)
unsigned(...)
std_logic(...)
std_logic_vector(...)
-- funktion: to_integer( unsigned(a) )
-- funktion: to_unsigned( s_int(4) )
```

... = String in dezimal:

```
b"..."
c"..."
```

Zustandstabelle und RTL zu Code: [RTLs unter Code malen]

```
erg := '1';
FOR i IN 0 TO 2 LOOP
erg := erg AND a(i);
END LOOP;
x <= erg;
```

a[2]	a[1]	a[0]	x
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Warum keine Verzögerungszeiten in VHDL-Code angeben?
Verzögerungszeiten werden nicht in Hardware umgesetzt.

Syntax for-Schleife:

```
for i in 0 to 3 loop
                  ...
  wait for 10 ns;
end loop;

-- nicht synthesesfähig, da wait-Anweisung
erforderlich und nicht wait-Anweisung nicht
synthesefähig
```

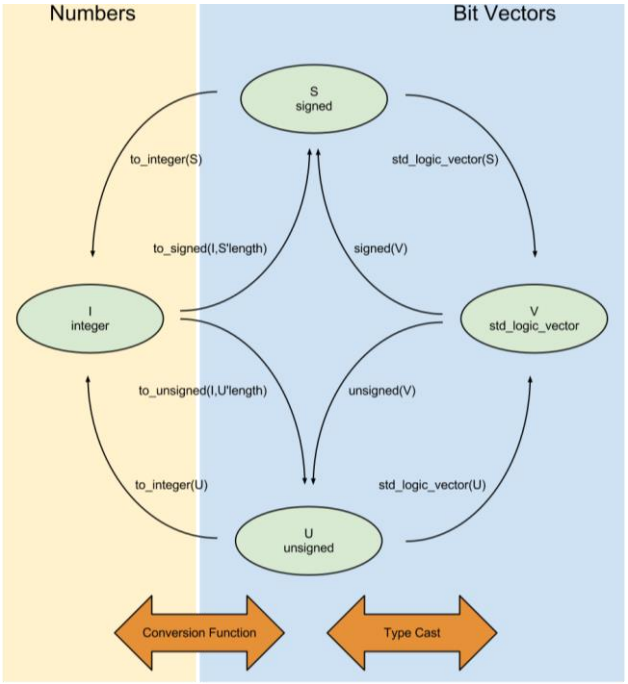
Richtig-Falsch-Fragen

Richtig:

Die Zuweisung von Werten an Signale erfolgt während der Ausführungsphase der Prozesse zunächst in einer Transaktionsliste.
Die Logiksynthese setzt eine Verhaltensbeschreibung auf Register-Transfer-Ebene in eine Strukturbeschreibung auf Gatterebene um.
Bei der IF-Anweisungen müssen sich die Bedingungen nicht gegenseitig ausschließen.
Im RTL-Modell der Hardware werden Verzögerungszeiten nicht modelliert.
Operatoren und Funktionen können überladen werden.
Variablen in Prozessen sind „statisch“, d.h. behalten ihren Wert.
Der Wert einer Variablen eines Prozesses bleibt nach Ausführung des Prozesses erhalten.
Das Attribut `'event` liefert einen Wert vom Typ boolean zurück.
Der Datentyp `unsigned` ist ein Feldtyp, wobei der Basistyp `std_logic` ist.
Der Divisionsoperator ist synthesefähig wenn der Divisor eine 2-er-Potenz ist.
Die Sensitivitätsliste ist nur für die Simulation relevant.

Falsch:

Bei Feldern ist der Indextyp immer „integer“.
Nur Ports mit dem Modus IN können gelesen werden.
Eine `for`-Schleife ist synthesefähig.
Der Datentyp `integer` ist ein Aufzählungsdatentyp.
Ein Zähler ist ein Schaltwerk, welches kein Überführungsschaltnetz benötigt.
Der Divisionsoperator für den Datentyp `unsigned` ist nicht synthesefähig.
Mit VHDL können digitale Schaltungen nur auf Gatterebene modelliert werden.
Mit Hilfe von AFTER-Anweisungen kann man für die Logiksynthese die zu implementierenden Verzögerungszeiten spefizieren.
Der Datentyp `real` ist synthesefähig.
Die Zuweisung von Werten an ein Signal erfolgt in einem Prozess sofort.



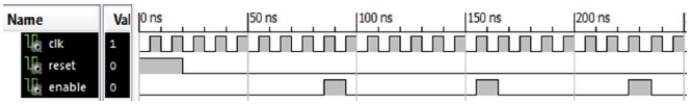
Operator	Operation	Datentyp linker Operand a	Datentyp rechter Operand b	Datentyp Ergebnis
	Diverse Operatoren			
**	a^b	nur Basis 2	integer	integer
abs	$\|b\|$	-	integer	integer
not	$\neg b$ (bitweise)	-	bit, boolean, bit_vector	wie Operand
	Multiplizierende Operatoren			
*	$a \times b$	integer	integer	integer
/	$a \div b$	integer	2er-Potenz	integer
mod rem	Rest von: $a \div b$	integer	2er-Potenz	integer
	Vorzeichen-Operatoren			
+ -	$\pm b$	-	integer	integer
	Addierende Operatoren			
+ -	$a + b$ $a - b$	integer	integer	integer
&	Verkettung	bit_vector[n]	bit_vector[m]	bit_vector[n+m]
	Schiebe-Operatoren			
sll srl sla sra rol ror	links (logisch) rechts (logisch) links (arithmetisch) rechts (arithmetisch links rotieren rechts rotieren	bit_vector	integer	bit_vector
	Vergleichs-Operatoren			
= /= < <= > >=	$a = b$ $a \neq b$ $a < b$ $a \leq b$ $a > b$ $a \geq b$	alle Typen	wie linker Operand	boolean
	Logische Operatoren			
and or nand nor xor xnor	$a \wedge b$ $a \vee b$ $\neg(a \wedge b)$ $\neg(a \vee b)$ $a \neq b$ $\neg(a \neq b)$	bit, boolean, bit_vector	wie linker Operand	wie linker Operand

y <= a & b;

y <= '0' & b;

- Notizen zu Altklausuren

Statt der Verwendung eines heruntergeteilten Taktes kann man auch mit Hife eines Zählers ein „Enable“-Signal erzeugen, welches eine Komponente periodisch (aber synchron) aktiviert (=1) oder deaktiviert (=0). Schreiben Sie den VHDL-Code für die Architecture eines solchen synchronen „Takteilers“, der das periodische `enable`-Signal aus der nachstehenden Abbildung erzeugt. Bitte achten Sie darauf, dass die zeitlichen Abstände zwischen zwei Aktivierungen durch `enable` = 1 in Relation zum Takt `clk` auch genau stimmen. Die Entity für Ihre Komponente ist wieder unten vorgegeben.




```
entity a4 is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           enable : out  STD_LOGIC);
end a4;

architecture Behavioral of a4 is
   SIGNAL q : UNSIGNED(3 DOWNTO 0);
   CONSTANT divider : UNSIGNED(3 DOWNTO 0) := x"6";
begin
   p1: PROCESS(clk, reset)
   BEGIN
      IF reset = '1' THEN
            q <= "0000";
            enable <= '0';
      ELSIF clk'event AND clk = '1' THEN
            IF q = divider THEN
                  q <= x"0";
                  enable <= '1';
            ELSE
                  q <= q+1;
                  enable <= '0';
            END IF;
      END IF;
   END PROCESS p1;
end Behavioral;
```

Synchroner Reset:

```
architecture beh of a2 is
signal shift, shift_next : std_logic_vector(3 downto 0);

begin
process (clk)

begin
   if clk'event and clk='1' then
      shift <= shift_next;
   end if;
end process;

process (shift, reset, enable)
begin
   if reset ='1' then
      shift_next <= (others => '0');
   elsif load = '1' then
      shift_next <= data;
   elsif enable = '1' then
      shift_next <= std_logic_vector(SHIFT_LEFT(unsigned(shift), 1));
   end if;
end process;

s_out <= shift(3);

end beh;
```

Counter4 (in Strukturbeschreibung):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;


entity counter4 is
            port (
                        clk   : in   std_logic;

                        reset : in   std_logic;

                        q_out : out std_logic_vector(3 downto 0)
            );
end counter4;



architecture beh of counter4 is
            signal q,q_ns: unsigned(3 downto 0);

begin       -- beh
            q_out <= std_logic_vector(q);          -- connect internal signals to output ports

            process (clk,reset) -- process for register function
            begin
               if reset='1' then
                  q <= (others => '0');
               elsif clk'event and clk = '1' then  -- rising clock edge
                  q <= q_ns;
               end if;
            end process;

            process (q)  -- process for next state decoder
            begin
               q_ns <= q + 1;
            end process;

end beh;
```

- Notizen zu Altklausuren

Strukturbeschreibung des Counter4 auf Seite 3 der Formelsammlung:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity counter4_dec7seg_struct is
    Port ( clk          : in STD_LOGIC;
           reset        : in STD_LOGIC;
           dout         : out STD_LOGIC_VECTOR (6 downto 0);
           q_out        : out STD_LOGIC_VECTOR (3 downto 0));
    end counter4_dec7seg_struct;

architecture struct of counter4_dec7seg_struct is

    signal q_intern     : STD_LOGIC_VECTOR (3 downto 0);
    signal od_dout      : STD_LOGIC_VECTOR (6 downto 0);

    component counter4 is
        Port (clk       : in  std_logic;
              Reset     : in  std_logic;
              q_out     : out std_logic_vector(3 downto 0));
    end component;

    component dec7seg is
        Port (data      : in  STD_LOGIC_VECTOR (3 downto 0);
              data_out  : out STD_LOGIC_VECTOR (6 downto 0));
    end component;

    begin

    cnt4:counter4 port map(
    clk      => clk,
    reset    => reset,
    q_out    => q_intern);

    seg:dec7seg port map(
    data     => q_intern,
    data_out => od_dout);

    dout <= od_dout;
end struct;
```

Clock-Devider:

Formeln (nicht auf Code unten bezogen, sondern von anderer Laboraufgabe):

$Taktfrequenz$: $f = 100\,MHz = 10^8\,Hz$;

$Periodendauer$: $T = 2\,\mu s = 2 \cdot 10^{-6}\,s \Leftrightarrow f = \frac{1}{T} = 500\,000\,Hz = 500\,kHz$

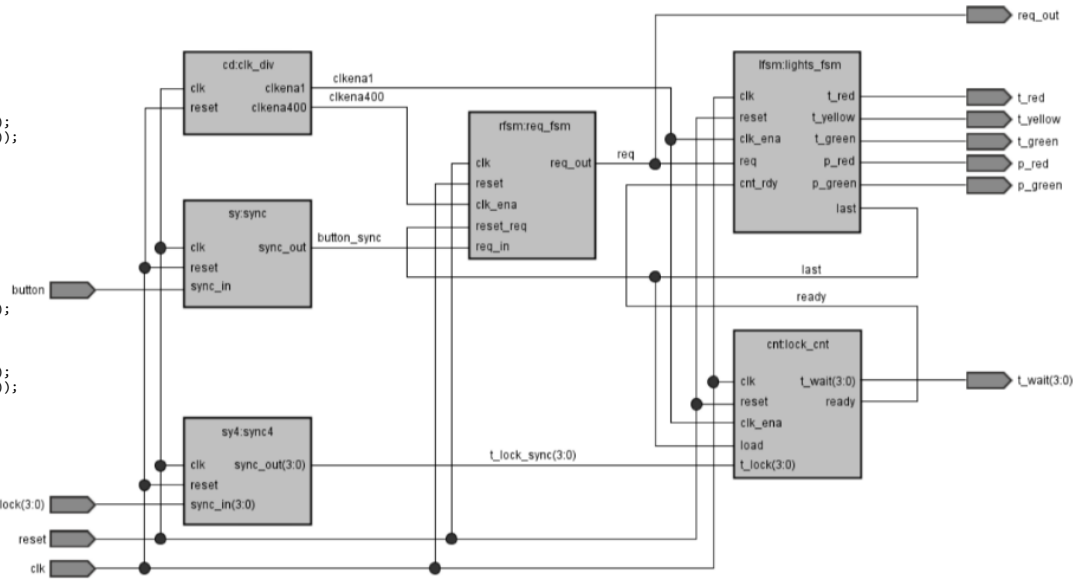
$Taktteiler$: $\frac{100\,MHz}{500\,kHz} = 200$ → Zähler muss bis 199 zählen

$Bitbreite$: $ld(200) \approx 7{,}644$ → 8 Bits

```
constant divider : integer := 200;
constant bits : integer := 8;

signal cnt, cnt_next : unsigned(bits - 1 downto 0);
signal cd_ena_out: std_logic;

-------------------

cd_reg: process (clk, reset)
begin
            if reset = '1' then
                        cnt <= (others => '0');
            elsif clk'event and clk = '1' then
                        cnt <= cnt_next;
            end if;
end process;

cd_nsd: process (cnt)
begin
            if cnt = to_unsigned(divider - 1, bits) then
                        cnt_next <= (others => '0');
            else
                        cnt_next <= cnt + 1;
            end if;
end process;

cd_ena_out <= '1' when cnt = 0 else '0';
```

- Notizen zu Altklausuren

Strukturbeschreibung der Ampelschaltung Laborversuch5



Entprellschaltwerk aus Labor:

```
üsn: process (enable, esw_in, esw_s)
begin
    case esw_s is
       when first_1 =>
          if enable = '1' and esw_in = '1' then
             esw_ns <= second_1;
          else
             esw_ns <= first_1;
          end if;

       when second_1 =>
          if enable = '1' and esw_in = '1' then
             esw_ns <= first_0;
          elsif enable = '1' and esw_in = '0' then
             esw_ns <= first_1;
          else
             esw_ns <= second_1;
          end if;

       when first_0 =>
          if enable = '1' and esw_in = '0' then
             esw_ns <= second_0;
          else
             esw_ns <= first_0;
          end if;

       when second_0 =>
          if enable = '1' and esw_in = '1' then
             esw_ns <= first_0;
          elsif enable = '1' and esw_in = '0' then
             esw_ns <= first_1;
          else
             esw_ns <= second_0;
          end if;
    end case;
end process;
```