Digital Design

Zusammenfassung

Joel von Rotz & Andreas Ming / Quelldateien

Table of contents

FPGA **4**
Technologie . . . 4
Layout . . . 4
Routing Ressourcen . . . 4

Integrated Logic Analyzer (ILA) **5**
Kurzanleitung . . . 5

VHDL **5**
Einordnung VHDL als HDL . . . 6
Entwicklung . . . 6
Designflow . . . 6
Simulation . . . 7
Event Driven Simulation . . . 7
Signale & Variablen . . . 7
Sprachelemente . . . 8
Bezeichner . . . 8
Signalzuweisung / Treiber <= . . . 8
Kommentare . . . 8
Komponenten . . . 8
`.vhd`-Dateistruktur . . . 8
Entity . . . 8
Generics . . . 8
Loop & Generate Statements . . . 9
Ports . . . 9
Architektur . . . 9
Benutzung bestehender VHDL-Komponenten . . . 10
Datentypen . . . 10
Integer . . . 10
Aufzählungstypen . . . 10
Subtypes . . . 11
Array . . . 11
Record . . . 11
Signed/Unsigned (IEEE 1076.3) . . . 11
IEEE `std_logic_1164` . . . 12
Real (Simulationstype) . . . 12
Time (Simulationstyp) . . . 12
Prozesse . . . 13
Arten von Prozessen . . . 13
`wait` (nur für Simulation) . . . 14
Process Statement . . . 14
Sequential Statements . . . 14

Schaltungssynthese **14**
Synthese . . . 14

Implementation . . . 14
Bitstream . . . 15
Speichermodellierung . . . 15
kombinatorisch ROM . . . 15
Synchrones ROM "Write before Read" . . . 15

Synchrone Logik **16**
Synchronisation & Entprellung . . . 16
Metastabilität . . . 16
Reset Synchronisierung . . . 17
Entprellen . . . 17
durch Blanking . . . 17
durch Unterabtastung . . . 17
Drehgeber-Signale (Quadratur-Signale) . . . 18

Finite State Machines (FSM) **18**
FSM-Typ: Mealy . . . 18
FSM-Typ: Moore . . . 18
FSM-Typ: Medvedev . . . 19
Parasitäre Zustände . . . 19
State Encoding . . . 19
Binär . . . 19
One-Hot . . . 19
Goldene Regeln der (FSM) Implementierung . . . 19
Memoryless Process (kombinatorische Logik) . . . 19
Memorizing Process (sequentielle Logik) . . . 19

Fest-/ und Gleitkomma-Arithmetik **20**
Festkomma . . . 20
Addition . . . 20
Multiplikation . . . 20

Gleitkomma **21**

PWM-D/A **21**

Packages **22**
Libraries & Use Clause . . . 22
Liste von Packages . . . 22
Synthetisierbare Bibliotheken . . . 22
`ieee.std_logic_1164` . . . 22
Nicht-Synthetisierbare Bibliotheken . . . 22

Vivado **22**
Project Summary . . . 22
Utilization . . . 22
Debugging . . . 22

Vorlagen **23**
Positive Getriggertes D-FlipFlop . . . 23
Mit asynchronem Reset . . . 23
Ohne Reset . . . 23
Finite State Machine . . . 23
Mealy . . . 23
Moore . . . 24
Synchronisation . . . 24
einfach . . . 24

mit Flankenerkennung 24
Entprellen 25
durch Blanking & Unterabtastung 25
durch Unterabtastung 25

FPGA

! Warum FPGA?

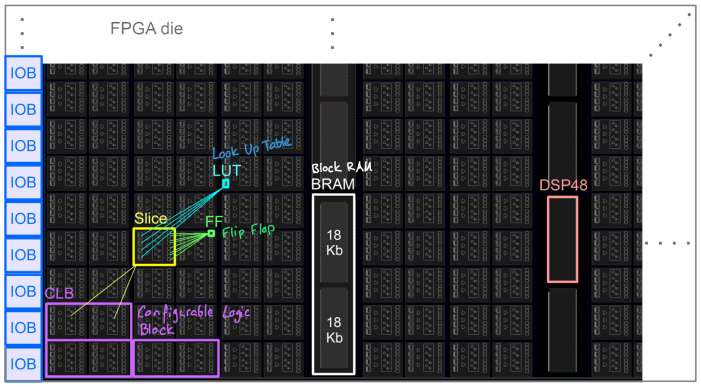
FPGA steht für **F***ield* **P***rogrammable* **G***ate* **A***rray* und ist die am weitesten verbreite Art von "programmierbarer" Logik.
Gegenüber einem anwendungsspezifischen Chip (ASIC) bieten FPGA:
+ höherer Flexibilität ; kürzere Entwicklungszeit ; geringer Entwicklungskosten
- im höhere Frequenzbereich → Um mitzuhalten sind Synchronisationen nötig, was zu Signal-Latenzen führt.
Verfügt über:

- Parrallelität → beschleunigte Verarbeitung
- Flexible Zuweisungen von Signalen & Pin-Funktionalitäten
- Deterministische Durchlaufzeiten von Signalen (z.B. 0cc, 2cc)
- Können mehrere Prozessoren beinhalten → erhöhter Integrationsgrad
- Custom Peripherie für Mikrocontroller

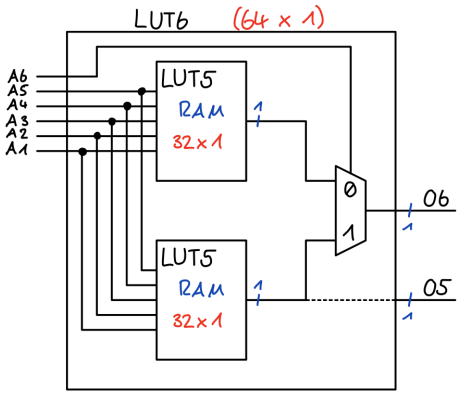
! FPGAs werden meist für extreme Bedingungen verwendet (z.B. 4000 $Op/\mu s$) **!**

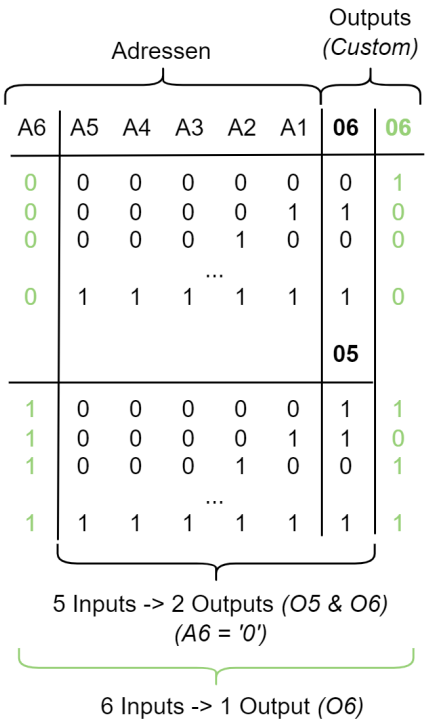
Technologie

Layout



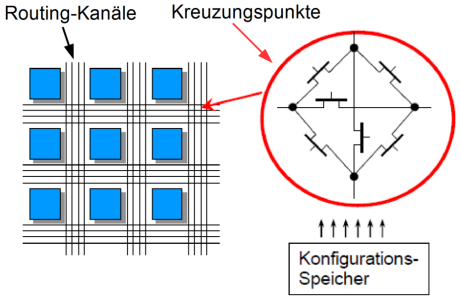
- **Logic Cell**: Mass zum Vergleich von FPGAs verschiedenere Familien/Hersteller
- **Configurable Logic Block** (CLB): Enthält 2 slices
- **Slice**: 4 Funktionsgeneratoren (4×LUT6) + 8 FFs
- **Distributed RAM**: Konfigurationsspeicher einiger Slices ist als Datenspeicher nutzbar
- **Block RAM** statische dual-port RAMs in Blöcken zu 18/36Kbit
- **DSP48 Slice**: Multiply-Accumulate und Arithmetic-Logic Unit (DSP ALU)
- **CMT**: Clock Management Tile (clock synthesis, phase shift, PLL)





Routing Ressourcen

Hierarchisches Routing mit verschiedenen langen Verbindungen ; Sechs-Transistor-Kreuzungspunkt ; Routing-Delay ≥ 70% Gesamt-Delay ; Spezielles low-skew Netze für regionale/globale Clocksignale



Integrated Logic Analyzer (ILA)

ILAs werden meist für In-System-Debugging von FPGA-Designs verwendet (z.B. bei aufwendigen Schaltungen oder fehlenden Inputsignal-Spezifikationen).

- ILA werden zusammen mit dem **D**esign-**u**nder-**T**est synthetisiert → genügend Ressourcen verfügbar
- ILA beeinflusst als Messmittel das Messobjekt → befolgen ILA & DuT die Regeln des synchronen Designs, ist dies akzeptabel.

$$T_{win} = N_S \cdot T_{Clk}$$

$$C_{SB} \geq N_S \cdot W_S$$

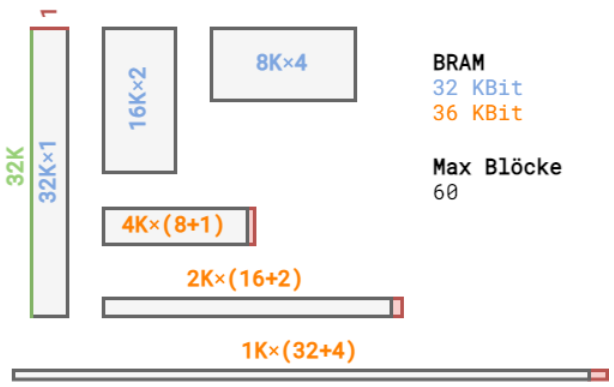
$$W_S = S + \mathbf{1}$$

1: immer da, evtl. Trigger

T_{win} : beobachtbares Zeitfenster
N_S : Anzahl zu speichernde Samples
C_{SB} : Speicherkapazität des Sample Buffers
W_S : Gesamtwortbreite aller zu anal. Signale
S : Anzahl zu analysierende Signale

Verfügbare BRAM-Blöcke

Der Sample Buffer wird mit Block-RAM-Blöcken zusammengesetzt.
Das FPGA-Board Zybo hat zwei Begrenzungen: totaler Speicher ist 36KBit und Maximale Anzahl Blöcke sind 60 Blöcke.



Beispiel

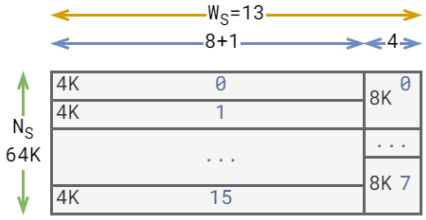
4 PWM Zyklen einer RGB-LED (Counter + Output) → $N_R = 2$, $N_G = 3$, $N_B = 4$, $T_{DAC} = 100\mu s$

$$W_S = \left.\begin{array}{ll} R: & 1+2 \\ G: & 1+3 \\ B: & 1+4 \end{array}\right\} = 12 + \mathbf{1} = 13$$

1: immer da, evtl. Trigger

$$T_{win} = 4 \cdot T_{DAC} = 400\mu s$$

$$N_S \geq T_{win} \div T_{CLK} = 50000 \rightarrow 64K$$



Kurzanleitung

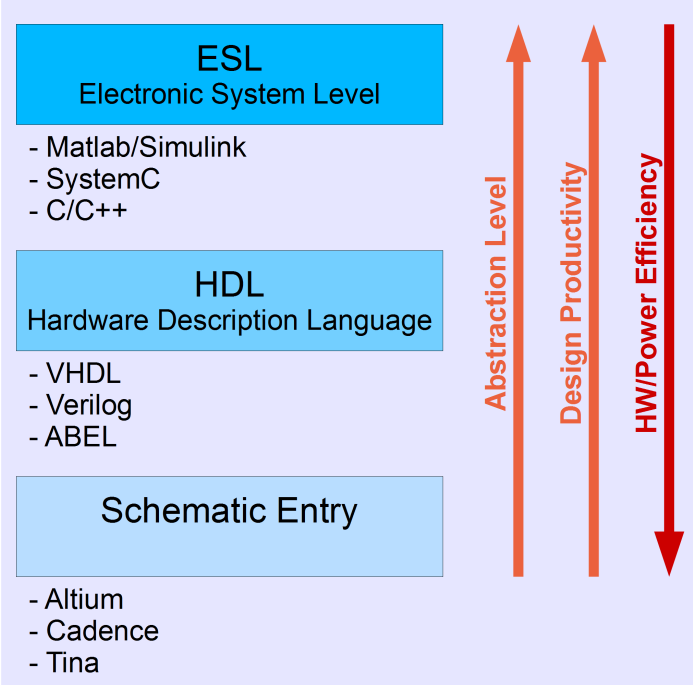
1. *Run Synthesis + Open Synthesized Design*
2. Debug-Signale auswählen
   1. Netlist Window → *Mark Debug*
   2. Schematic Window → *Mark Debug*
   3. Tcl-Console `set_property MARK_DEBUG true [<signals>]`
3. Signale mit Debug verbinden
   - *Tools* → *Setup Debug*
   - Tiefe Sample Buffer (N_S)
   - *Capture Control* + *Advanced Trigger* wählen
4. Save Constraints + existing Constraint File
5. *Run Implementation + Generate Bitstream*
   1. Target beschreiben
   2. ILA GUI öffnet sich

VHDL

VHDL?

VHDL steht für **V**ery *High Speed Integrated Circuit* **H***ardware* **D***escription* **L***anguage*, klingt schnell und ist es auch. Diese Sprache dient für die Hardwarebeschreibung von FPGAs, insbesondere wie die "Logikfläche" konfiguriert wird, und ist keine Programmiersprache, es handelt sich um eine Designsprache.

Einordnung VHDL als HDL



HW-nahes Design ermöglicht:

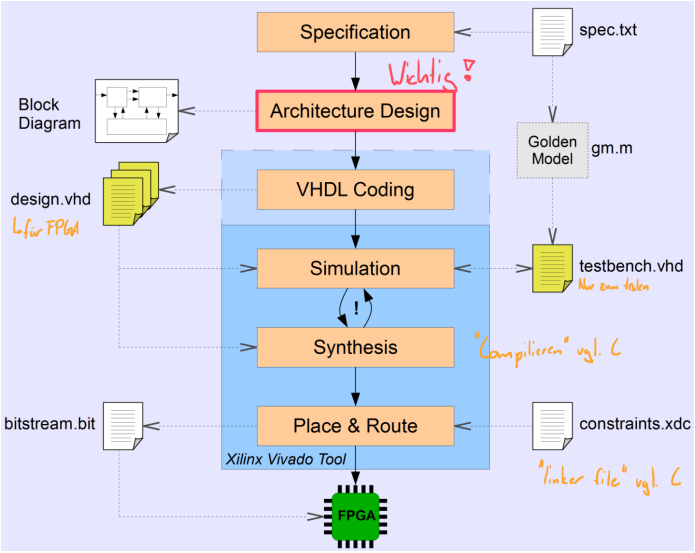
\+ kleinere, schnellere, energie-effizientere Schaltungen als ESL-Design

\- auf Kosten der Entwicklungszeit.

Entwicklung

Designflow

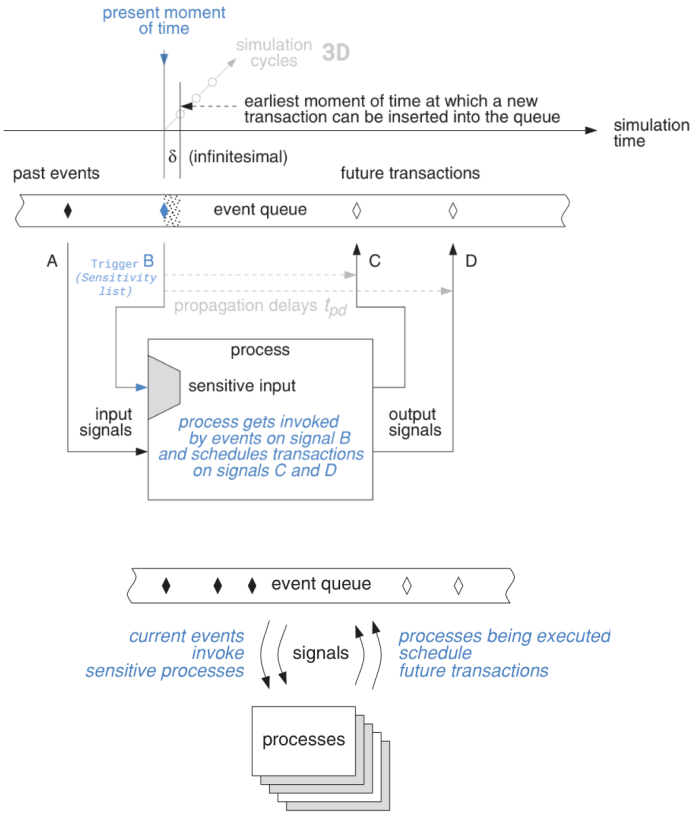
1. **Spezifikation**
   - Erstellen/Verstehen von Funktions- und Testspezifikation. Vorgaben zum strukturellen Aufbau des Designs.
2. **Architektur-Entwurf**
   - Schaltung wird in Blockdiagramm festhalten → Ableiten von Ports, Wortbreiten, Codierung
   - Je nach Komplexität Erstellung von Prozess-Dokumentation und/oder RTL-Schemas
   - Zustandsdiagramm der FSMs
3. **VHDL Implementierung**
   - Verwendung von VHDL-Templates für synchrone Logik
   - VHDL-Code kommentieren
4. **Design Constraints**
   - in `.xdc`-File *Top-Level Ports Location* & *Clock Period* Constraints setzen.
5. **Probe-Synthese**
   - VHDL-Code überarbeiten bis keine Warnungen → *Found latch for signal. . .*, *. . . signals missing in the process sensitivity list. . .*, *. . . signals form a combinational loop. . .*
   - Konsistenz-Check anhand Vergleich selbst gezählte #FF und Synthese-#FF
6. **Simulation**
   - Erstellen einer VHDL-Testbench und Simulation des Designs (MUT) gemäss Spezifikation
   - Testbench mit automatischem Vergleich von Ist & Soll ist für komplexere Designs gut
7. **FPGA Implementierung**
   - *Run Implementation* ausführen für *Technology-Optimizations* & *Place&Route*
   - Falls Timinganalyse Fehler → Architektur überprüfen
8. **HW-Test**
   - Bitstream auf HW gemäss Spezifikationen testen.



Simulation

Problem

Wie kann Parallelität auf einem sequentiell arbeitenden Computer simuliert werden? → Event Queue & Transactions



Event Driven Simulation

1. Vorrücken der Simulationszeit zur nächsten Transaktion in der Event Queue.
2. Setzt alle zu aktualisierenden Signale auf den mit der aktuellen Transaktion verbundenen Wert.
3. Arbeitet alle Prozesse ab, die auf Signale sensitiv sind, deren Wert sich durch die aktuelle Transaktion geändert hat. Signalzuweisungen werden als zukünftige Transaktionen in die Event Queue abgelegt.

Important

- Events werden durch die Signaländerung generiert, nicht die Transaktionen!
- Nicht alle Transaktionen führen zu einem Event *(Sensitivity List)*

Signale & Variablen

- Variable Assignment := → unabhängig von Simulation, Effekt ist sofort
- Signal Assignment <= → Effekt nach `after` (nur Simulation) oder bei keiner Verzögerung nach Simulationszyklus (δ-Delay ; Sim. & Synth)

```
                                                    ①
sigA <= '1',
        '0' after 1 ms,
        'Z' after 10 ms;


                                                    ②
sigB <= '1';
```

1. Effekt nach `after` (nur Simulation)
2. Bei keiner Verzögerung (Synthese & Simulation)→ Transaction für nächsten Simulationszyklus geplant (δ-Delay)

Sprachelemente

Bezeichner

z.B. Signaldeklaration oder Port-Variable

- Gross-/kleinschreibung wird nicht unterschieden → nicht case-sensitiv

```
signal some_signal, some_other_signal, result :
↪  integer;

ReSULT <= SOME_SIGNAL + sOME_oTHer_SIGNal;
```

- Namen können beliebig lang sein
- Keine Spezialzeichen ausser _ → nicht am Anfang & Ende + nicht verdoppelt

Signalzuweisung / Treiber <=

Mit dem *Treiber* <= werden Signale von der rechten Seite ausgewertet und auf das Signal auf der linken Seite zugewiesen.

```
Inv_Out <= not Inv_In; -- Inverter: Das Signal Inv_In
↪  "treibt" Signal Inv_Out
```

> **Kontext**
>
> Je nachdem wo <= verwendet wird, hat es eine andere Bedeutung → in einem `if`-Statement wird es als kleiner-gleich angesehen ; bei Signalen als Treiber.

Kommentare

Mit -- werden Kommentare begonnen → single line comments!

```
-- Das hier ist ein Kommentar
Hier aber nicht mehr :(
```

Komponenten

`.vhd`-Dateistruktur

VHDL-Code wird meistens in `.vhd` Dateien geschrieben.

```
-- Header Commment (Author, Date, Filename, etc.)
library ... -- Library einbinden
use ... -- Packages aus Library bekanntgeben
entity ...
-- Schnittstelle der Komponente gegen aussen
architecture ...
-- Funktion (Innenleben) der Komponente
```

Entity

Eine Entity beschreibt den Komponenten für äusserliche Zugriffe → nur Struktur der Komponente bekannt, aber nicht deren Inhalt.

```
entity MyComponent is
  generic(                                                    ①
    y : integer := 20;
    z : integer
  );
  port (                                                      ②
    a_pi, b_pi : in    std_logic; -- Input
    c_po       : out   std_logic; -- Output
    --x_pio    : inout std_logic  -- Bidirectional
  );
  constant c_max_cnt : integer := 20_000;                     ③
end MyComponent;
```

① Analog zu `#define` in C → werden während Kompilation eingefügt! "Ko"
② Signal Deklarationen → **müssen** bei Instanziierung des Komponenten im übergeordneten Design verdrahtet werden!
③ Konstanten können mit Generics interagieren (Wert ausrechnen und an Konstante zuweisen) → während Laufzeit nicht veränderbar!

> **Sichtbarkeit**
>
> - Alles was in der Entity bekannt ist (inkl. Libraries), ist auch in der zugehörigen Architecture bekannt.
> - Alles was in der Architecture bekannt ist, ist nicht in der Entity bekannt.

Generics

Analog zu C Preprocessor Direktive `#define`

Mit `generic` können Komponenten angepasst/parametrisiert werden → Komponent muss daher mit diesen Generics implementiert werden.

Im Gegensatz zu Konstanten kann Wert ausserhalb definiert werden.

Beim Testen sollten mit Generic-Parametern die Randwerte verifiziert werden (*Corner-Case* Testing).

Beschreibung der Komponente

```
entity NAND_gate is
  generic(
    IW : integer := 2 -- input width, def. 2
  );
  port(
    InP  : in std_logic_vector(IW-1 downto 0);
```

```
    OutP : out std_logic;
  );
end NAND_gate;
```

Verwendung der Komponente

```
architecture rtl of top is

component NAND_gate is
  generic(IW : integer := 3);   -- default
  port(
    InP : in std_logic_vector(IW-1 downto 0);
    OutP : out std_logic;
  );
end component NAND_gate;

begin
I1 : NAND_gate
port map (
  InP => In1,
  OutP => O1
);

I2 : NAND_gate
generic map (
  IW => 8     -- Instance 2 has 8 inputs
) -- note no semicolon
port map (
  InP => In1,
  OutP => O1
);

end logic;
```

Loop & Generate Statements

Bei Implementierung von Komponenten mit Generic-Parametern werden häufig `loop` oder `generate` verwendet.

```
-- architecture using loop statement
architecture A_loop of NAND_gate is
begin
  P_nand: process(InP)
    variable tmp : std_logic;
  begin
    tmp := InP(0);
    for i in 1 to IW-1 loop
      tmp := tmp and InP(i);
    end loop;
    OutP <= not tmp;
  end process;
end A_loop;

-- the same is achieved with generate statements
architecture A_gen of NAND_gate is
signal tmp : std_logic_vector(IW-1 downto 0);
begin
  tmp(0) <= Inp(0);
  tree: for i in 1 to IW-1 generate
    tmp(i) <= tmp(i-1) and Inp(i);
    invert: if i = IW-1 generate                    ①
      Outp <= not tmp(i);
    end generate;
  end generate;
end A_gen;
```

① `generate`-Statements können auch in *if*-Form verwendet werden (zum Ein- oder Ausschalten von Code-Blöcken.

Ports

Die Richtung der Ports werden mit `in`, `out` und `inout` bezeichnet.

Architektur

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

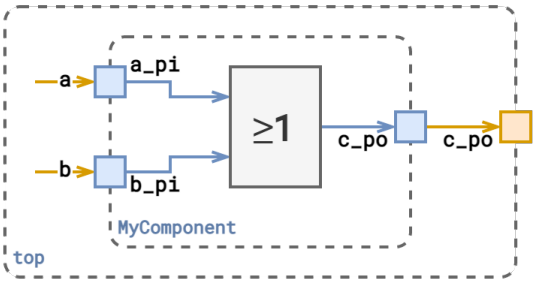
```
architecture a1 of MyComponent is                   ①
  -- Deklarationen (Signale, Komponenten)
  signal tmp : std_logic;                           ②
begin
  tmp  <= a_pi or b_pi;                             ③
  c_po <= tmp;
end a1;
```

① Siehe Section

② **Deklarationen** (für Signale & Komponenten)
③ **Implementierung**

> **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.

Benutzung bestehender VHDL-Komponenten




```
entity top is
  port (
    c_po : out std_logic;
  );
end top;

architecture struct of top is
  signal a,b : std_logic;
  component MyComponent is                    ①
    port (
      a_pi, b_pi : in  std_logic;
      c_po       : out std_logic;
    );
  end component MyComponent;
begin

  Inst1: MyComponent                          ②
  port map ( a_pi => a,                       ③
             b_pi => b,
             c_po => c_po
           );                                 ④
end struct;
```

① Deklaration der Komponente im Deklarationsteil → Name der entsprechenden Entity!
② Instanzierung eines Komponenten
③ Signale/Ports werden verbunden
④ Kein Endzeichen!

Es gibt allerdings eine Kurzschreibweise die angewendet werden kann, wenn sich die Komponente im *selben Verzeichnis* befindet

```
architecture struct of top is
  -- external module
  use work.MyComponent;

begin
  Inst1: entity work.MyComponent
    port map(
      a_pi => a,
      b_pi => b,
      c_po => c_po
    );
end struct;
```

Datentypen

Synthese & Simulation	Nur Simulation
`std_logic`, `std_ulogic` (IEEE 1164) `signed`, `unsigned` (IEEE 1176.3) `integer` (IEEE1076) `type` (IEEE 1076 userdefined)	`real`, `time`, `character`, `file` (IEEE 1076)

Integer

```
signal my_int : integer range -128 to 127;
```

- Repräsentiert Bereich $-2^{31} \ldots 2^{31} - 1$
- **!!!** Wertebereich einschränken (Konsistenzcheck für Simulation)
- Zahlen Darstellungen

```
my_int <=    61;      -- Dezimal (Default)
my_int <= 10#61#;     -- dezimal
my_int <=  2#111101#; -- binär
my_int <=  8#75#;     -- octal
my_int <= 16#3D#;     -- hexadezimal
```

Aufzählungstypen

Aufzählungstypen sind wie `enum` in C, wobei die Repräsentierung hinter den Aufzählunstypen nicht bekannt ist.

> **Reservierte Wörter**
>
> Für Aufzählungstypen dürfen keine reservierten Wörter verwendet werden. Im folgenden Beispiel wäre also `port` keine möglicher Wein, da dies ein reserviertes Wort ist.

```
type wine is (white, rose, red);
signal beverage : wine := red;  -- initialization
```

Subtypes

gleiche Operationen wie Grundtype, einfach bestimmte Teilmenge (z.B. `natural`, `positive`)

```
subtype t_day is integer range 1 to 31;
signal day : t_day;
```

Array

Gruppierung von Elementen gleichen Types

```
type t_byte is array (7 downto 0) of std_logic;
signal byte : t_byte; -- same as below
signal byte : std_logic_vector(7 downto 0);
```

Attributes `A'<atr>(N)`

Mit den Attributes eines Arrays können verschiedene Informationen entnommen werden, welche während der Synthetisierung eingefügt werden (analog zu C Preprozessoren).

```
-- Initialise array(1. Dim, 2. Dim)
type t_arr is array(2 to 4, 15 downto 0) of
↪  std_logic;
signal A : t_arr;

A'left(1)          -- 2
A'right(2)         -- 0
A'high(2)          -- 15
A'low(2)           -- 0
A'range(1)         -- 2 to 4
A'reverse_range(2) -- 0 to 15
A'length(1)        -- 3
A'ascending(2)     -- false
A'element          -- std_logic
```

Aggregates

Array Aggregates werden verwendet, um Arrays mit konstanten Werten einzusetzen.

```
type t_map is array(1 to 4) of std_logic;
-- positional association
constant c_map : t_map := ('Z', '0', '0', '1');
-- named association
constant c_map : t_map := (1=>'Z', 2=>'0', 3=>'0',
↪  4=>'1');
constant c_map : t_map := (1=>'Z', 2|3=>'0',
↪  others =>'1');
```

Zuordnungen dürfen nicht gemischt werden!

```
-- !ERROR!
constant c_map : t_map := ('1', 2=>'0', others =>
↪  '1')
```

Record

Gruppierung von Elementen unterschiedlichen Types

```
type t_date is record
  day  : t_day;
  year : positive;
end record;
```

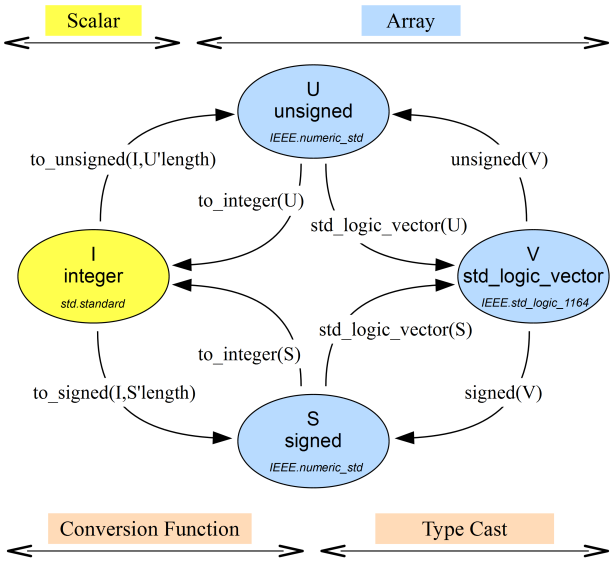
Signed/Unsigned (IEEE 1076.3)

Binärzahlen in Form von 2er-Komplement, bzw. vorzeichenlosen Binär-Arrays

Unterschied Signed/Unsigned und Integer

`integer` ist ein Skalar-Typ und fest in VHDL eingebaut. Obwohl der Typ der Zahlenbereich einer 32-Bit 2er-Komplementzahl hat, hat es keine Tool-interne Darstellung (z.B. MSB ist nicht prüfbar).
`unsigned/signed` sind definiert im Package `numeric_std` und haben eine genau definierte Darstellung, als Array von `std_logic` bits mit bekannten Definierung von MSB & LSB.

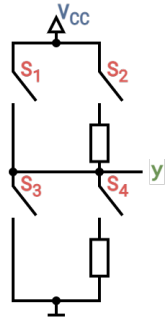
Umwandlungstabelle



IEEE `std_logic_1164`

Datentypen: `std_logic` & `std_ulogic`

> **Warum `std_logic_1164`?**
>
> In Standard-VHDL gibt es zwei binäre Datentypen: `bit` (`0`,`1`) und `boolean` (`false`, `true`) → entspricht aber nicht realen digitalen Signalen!



State	Bedeutung	Bereich
`'U'`	Uninitialized (–)	Simulation
`'X'`	Forcing Unknown (S1,S3)	Simulation
`'0'`	Forcing Low (S3)	Synthese, Simulation
`'1'`	Forcing High (S1)	Synthese, Simulation
`'Z'`	High Impedanz (–)	Synthese, Simulation
`'W'`	Weak Unknown ()	Simulation
`'L'`	Weak Low (S4)	Simulation
`'H'`	Weak High (S2)	Simulation
`'-'`	Don't Care	Simulation

– bedeutet alle Schalter offen

> **Caution**
>
> `std_ulogic` steht für *unresolved* → Signal kann nur von einem Prozess geändert werden!
>
> ---
>
> `std_logic` ist *resolved* und kann von mehreren Prozessen geändert werden. Zustände werden mit der *Resolution Table* aufgelöst.
> Wird dies gemacht, Simulation möglich, aber keine Synthetisierung!

```
-------------------------------------------------------------
-- resolution function "resolved"
-------------------------------------------------------------
constant resolution_table : stdlogic_table := (
--   ---------------------------------------------------------
--   |  U    X    0    1    Z    W    L    H    -        |   |
--   ---------------------------------------------------------
     ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
     ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
     ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
     ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
     ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
     ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
     ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
     ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
     ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |
   );
```

Real (Simulationstype)

→ Wertebereich ist Hersteller-spezifisch

```
signal float : real;
float <= 73_000.0;
float <= 7.3E4;
float <= 73000;                                          ①
```

① Fehler, da `730000` eine Ganzzahl ist

Time (Simulationstyp)

→ einziger vordefinierter physikalischer Datentyp

```
type time is range -2147483647 to 2147483647
  units fs;
    ps = 1000 fs;
    ns = 1000 ps;
    us = 1000 ns;
    ms = 1000 us;
    sec = 1000 ms;
    min = 60 sec;
    hr = 60 min;
  end units;

signal t : time
t <= now + 2.5 sec;                                    (1)
```

(1) now liefert aktuelle Simulationszeit

Prozesse

! Parallelität

In VHDL wird das Verhalten digitaler HW durch parallele Prozesse beschrieben die gleichzeitig ausgeführt werden und über Signale miteinander kommunizieren.

```
P1: process (i1, i2, i3)                               (1)
variable v_tmp : std_logic;                            (2)
begin
  v_tmp := '0';
  if i1 = '1' and i2 = '0' then v_tmp := '1'; end if;
  o1 <= v_tmp and i3;                                  (3)
  o2 <= v_tmp xor i3;
end process P1;
```

(1) Prozess mit *Sensitivity List* & Spitznamen
(2) Lokale Variablen (Scope innerhalb P1)
(3) Prozess P1 treibt Signale o1 & o2

Sensitivity List

Prozesse können mit Hilfe einer *Sensitivity List* auf ausgewählte Signale sensitiv gemacht werden → Prozess reagiert nur auf diese Signale!
! Prozesse müssen nicht auf alle ihre Inputsignale sensitiv sein! !

Caution

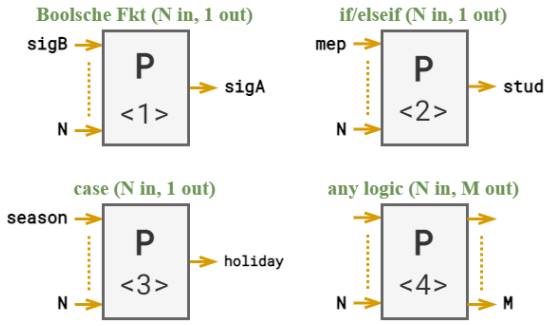
In synthetisierbaren VHDL Code darf jedes Signal nur von einem Prozess getrieben werden.
Tristate-Leitungen können simuliert werden, aber nicht synthetisiert.

Arten von Prozessen

```
sigA <= not sigB                                       (1)
stud <= happy     when mep >= C else                   (2)
        satisfied when mep >= E else
        sad;
with season select                                     (3)
holiday <= seaside when summer,
           skiing when winter | spring,
           none when others;
process                                                (4)
begin
end process;
```

(1) Concurrent Signal Assignment (komb.)
(2) Conditional Signal Assignment (komb.)

③ Selected Signal Assignment (komb.)
④ Process Statement (komb./seq.)



`wait` (nur für Simulation)

Mit dem Keyword `wait` kann ein Prozess pausiert und/oder sensitiv gemacht werden. **NUR FÜR SIMULATION!**

```
wait;           -- infinite loop
wait on s1, s2; -- wait on event from s1 or s2
wait until s1 = '1'; -- wait until condition is
↪  fulfilled
wait for 1 ms; -- wait stated time
```

Nur Simulation

```
process
begin
  sigB <= sigA;
  wait on sigA;
end process;
```

Simulation & Synthese

```
process(sigA)
begin
  sigB <= sigA;
end process;
```

Process Statement

> **!** Important
>
> Ein Process Statement ist parallel nach aussen und sequentiell im Inneren.
> Alle Signal Assignments ausserhalb von `process` (Concurrent-, Selected-, Conditional-Signal Assignment) sind Process Statements in Kurzschreibform!

Sequential Statements

Innerhalb eines Process sind alle Statements sequential statements

(sequential) signal assignment	`<=`
variable assignment	`:=`
`if, case, for, loop, while`	-
nur für Simulation:	-
`wait, assert, report`	

Folgende beschreibungen resultieren in der selben implementierungen auf dem FPGA.

```
stud <= happy     when mep >= C else
        satisfied when mep >= E else
        sad;

P1: process (mep)
begin
  if mep >= C then
    stud <= happy;
  elsif mep >= E then
    stud <= satisfied;
  else
    stud <= sad;
  end if;
end process P1;
```

Schaltungssynthese

Synthese

1. Synthese übersetzt VHDL-Modell in eine RTL-Beschreibung aus Registern und kombinatorischen Blöcken
2. Erzeugung von Netlist → Verbindungen von Elementen (Primitives), welche in den Slices verfügbar sind.

Implementation

1. **Initialize** → Einbindung Constraints & Netlist ins Design
2. **Optimize** → Packt Primitives der synth. Netlist in Slices, Versuch Gatterkomb. zu vereinfachen (Reduktion HW)
3. **Place** → Zuweisung der Slices auf CLBs (configurable logic blocks) auf dem FPGA, Zuordnung IO aus Constraints
4. **Route** → Definiert Verbindungen zwischen CLB & IO
5. **Static Timing Analysis** → Überprüfung des Designs mit den Timing Constraints.

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

Bitstream

Die Bitstream-Generierung erzeugt das FPGA-Konfigurationsfile ('kompilierte Datei').

Speichermodellierung

Adressierbare ROM & RAM können im FPGA aufgebaut werden aus: Slice-Logik (LUTs, FF), speziellen Makrozellen (BRAM) und Konfigurations-RAM (Distributed Memory).

- RAMs im FPGA sind immer getaktete/synchrone Speicher
- ROMs können auch kombinatorisch sein falls keine Latenz besteht
- Nur kleine Speicher sollte man in *behavioural* VHDL beschreiben

kombinatorisch ROM

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.Numeric_Std.all;

entity crom is
  generic(
    AW : integer := 3; DW : integer := 8);
  port (
    addr : in
    std_logic_vector(AW-1 downto 0);
    Dout : out std_logic_vector(DW-1 downto 0)
  );
end entity crom;

architecture Behav of crom is
  type t_rom is array (0 to 2**AW-1) of
  std_logic_vector(DW-1 downto 0);
  constant rom : t_rom := (X"1A", X"1B", X"1C", X"1D",
                           X"2A", X"2B", X"2C", X"2D");
begin
  Dout <= rom(to_integer(unsigned(addr)));
end architecture Behav;
```

$$C = 2^{AW} \cdot DW = 2^{10} \cdot \underbrace{x}_{[\text{kBit}]}$$

C : Speicherkapazität in $[Bits]$ oder $[kBits]$
AW : Adresswortbreite
DW : Datenwortbreite

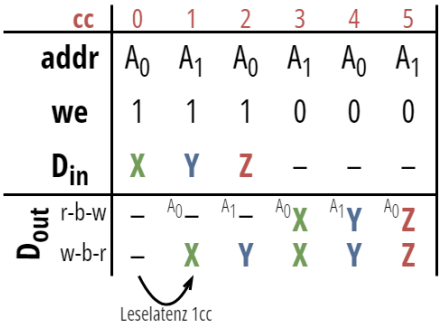
Beispiel ROM

$C_{ROM} = 2^3 \cdot 8 = 64$ Bits
$\rightarrow \#FF = 0$ (da asynchron)

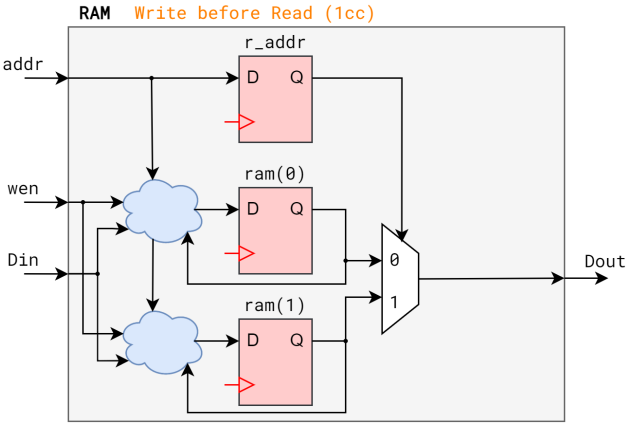
Synchrones ROM "Write before Read"

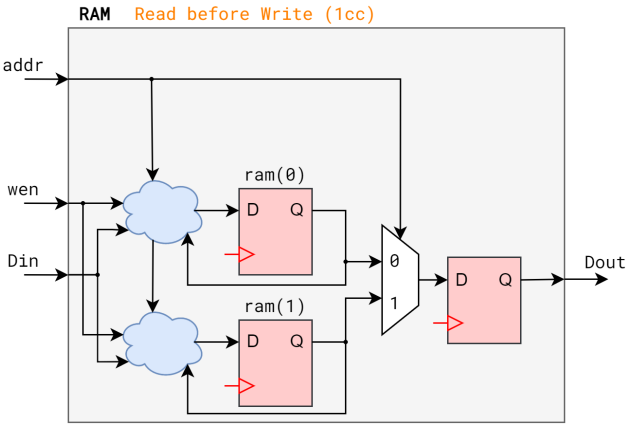
! WBR & RBW

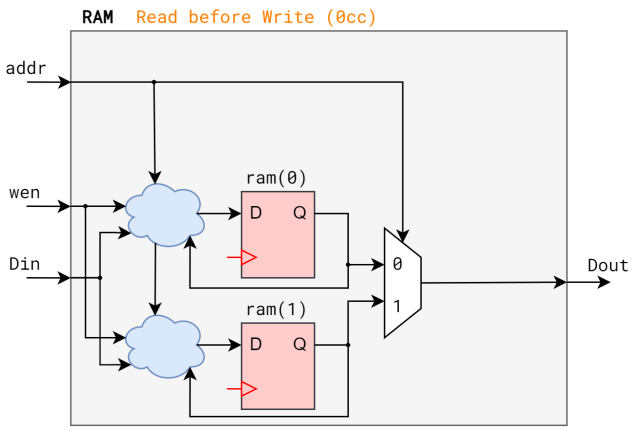
Bei *Write-Before-Read* wird zuerst geschrieben und dann das Beschriebene gelesen.
Bei *Read-Before-Write* wird zuerst die Speicherstelle ausgelesen und erst dann die Stelle überschrieben.



Folgende RTL-Schema beschreiben ein WBR & RBW Speicher mit 1cc oder 0cc Latenz.








```
entity sram is
  generic(
    AW : integer := 4; DW : integer := 8);
  port (
    clk  : in std_logic;
    we   : in std_logic;
    addr : in std_logic_vector(AW-1 downto 0);
    Din  : in std_logic_vector(DW-1 downto 0);
    Dout : out std_logic_vector(DW-1 downto 0)
  );
end entity sram;

architecture Behav of sram is
  type t_ram is array (0 to 2**AW-1) of
    std_logic_vector(DW-1 downto 0);
  signal ram : t_ram;
  signal r_addr : std_logic_vector(AW-1 downto 0);
begin
  P_ram: process(clk)
  begin
    if rising_edge(clk) then
      if we = '1' then
        ram(to_integer(unsigned(addr))) <= Din;
      end if;
      r_addr <= addr;
    end if;
  end process;

  Dout <= ram(to_integer(unsigned(r_addr)));
end architecture Behav;
```

Beispiel RAM

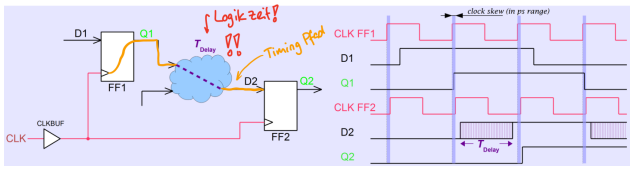
$$C_{RAM} = 2^4 \cdot 8 = 128 \text{ Bits}$$
$$\rightarrow \#FF = \underbrace{4}_{\text{AW}} + \underbrace{128}_{\text{Speicher}} = 132 \text{ FF}$$

Synchrone Logik

Warum synchrones Design?

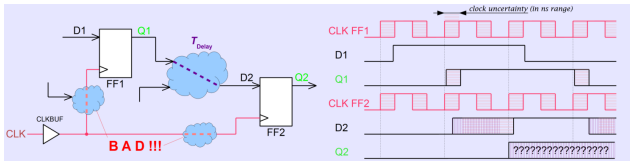
synchrones Design: **Eine** worst-case Timing-Analyse

$$max(T_{Delay}) < T_{CLK} \pm T_{skew}$$



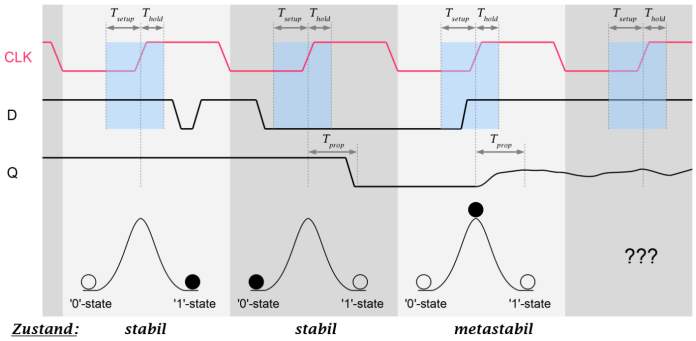
\+ Keinen Einfluss von Hazards, ungültige Zwischenwerte, etc.
\+ Signale vor Speicherung stabil
\+ Deterministisches Verhalten unabhängig von Gate-level Details
\+ Systematisches Design/Test/Debug mit etablierten Methoden & Tools
\- Max. Verarbeitungsgeschwindigkeit durch Verzögerungszeit des längsten Pfades definiert.
\- Evtl. höherer Energieverbrauch und EMV-Probleme durch CLK-Signal

asynchrones Design: ∞-**viele** Timing Analysen



Synchronisation & Entprellung

Metastabilität



$\rightarrow$ Durch Verletzung der Hold-/Setup-Zeit kann ein Speicherelemnt in den Metastabilen Zustand geraten (unbestimmter Ausgang) $\Rightarrow$ Kann durch Synchronisation reduziert werden!

Mean Time Between Failure

Zeit t_{meta_res} beschreibt die Zeit, bis von der Metastabilität wieder ein definierten Wert angenommen wird.→ Je kleiner, desto besser!

$$t_{meta\_res} < t_{allowed} = \frac{1}{f_{clk}} - t_{pd} - t_{su}$$

t_{MTBF} beschreibt die Wahrscheinlichkeit, dass die obere Bedingung nicht erfüllt ist → Je kleiner desto besser!

$$t_{MTBF} = \frac{e^{K_2 \cdot \left(\frac{1}{f_{clk}} - t_{pd} - t_{su}\right)}}{K_1 \cdot f_{clk} \cdot f_d}$$

Grösstenteils ist die t_{MTBF} abhängig von der Clockfrequenz!

K_1 : Prozess-Konstante $[s]$
K_2 : Prozess-Konstante $[Hz]$
t_{pd} : Propagation delay
t_{su} : Setup-Zeit
f_d : durchschnittliche Frequenz des asynchronen Datensignals
f_{clk} : Clockfrequenz

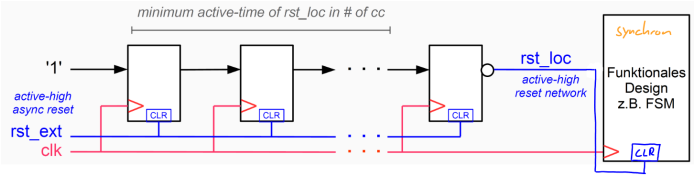
Reset Synchronisierung

Synchroner Reset

Synchronisierung wie für binäre Datensignale.
+ Geringer LUT-Verbrauch durch logische Kombination von Daten/Reset - Nur funktionstüchtig wenn Clock-Signal aktiv

Asynchroner Reset

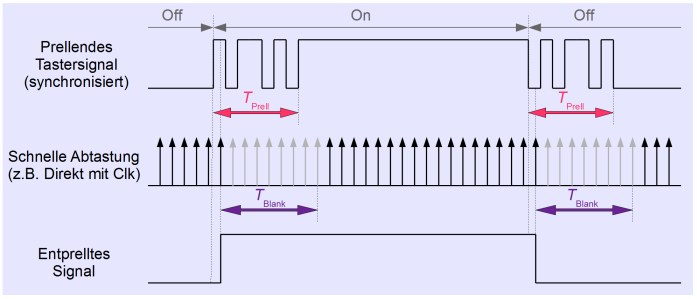
Spezielle Synchronisierung damit alle FFs im gleichen cc freigegeben werden.
+ Geringer LUT-Verbrauch - Höherer LUT-Verbrauch, keine logischen Kombinationen von Daten/Reset möglich




```
sync_rst: process(clk_pi, rst_pi)
begin
  if rst_pi = '1' then
    rst_sr <= (others => '1');
    rst_po <= '1';
  elsif rising_edge(clk_pi) then
    rst_sr(0)          <= '0';
    rst_sr(3 downto 1) <= rst_sr(2 downto 0);
    rst_po             <= rst_sr(3);
  end if;
end process;
```

Entprellen

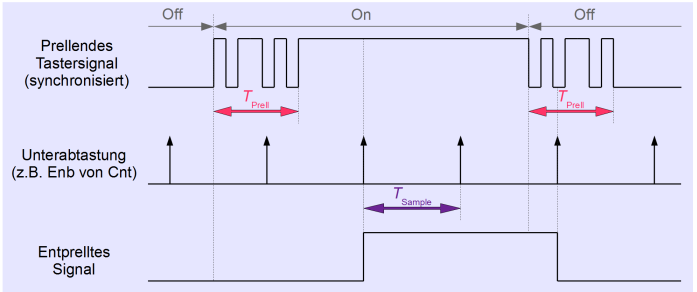
durch Blanking



- Kontaktsignal wird möglichst schnell abgetastet, um Signaländerungen auszuwerten.
- Geplante Aktion sofort beim Drücken bzw. Loslassen ausgeführt werden!

- Es muss T_{Blank} gewartet werden bis nächste Auswertung

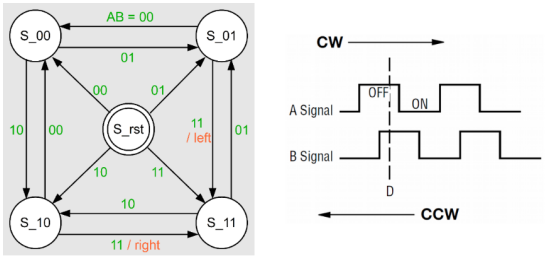
durch Unterabtastung



- Kontaktsignal wird langsam abgetastet
- $T_{Sample} > T_{Prell}$

worst-case geplante Aktion erst zwei volle Abtastperioden nach Schalterereignis ausgeführt

Drehgeber-Signale (Quadratur-Signale)



Die Reihenfolge des Auftretens der 4 Eingangskombination bestimmt die aktuelle Drehrichtung. Eine sichere Decodierung ist z.B. mit einer Mealy-FSM mit 5 Zuständen möglich.

Finite State Machines (FSM)

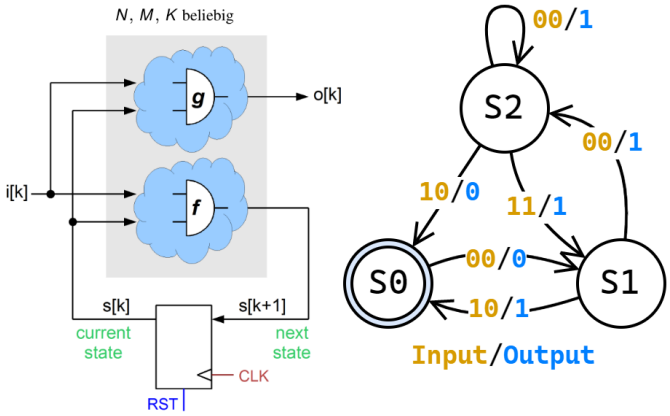
Oder auch *getaktete/synchrone/sequentielle Logik*

> **Warum FSM?**
>
> Jede (komplexe) digitale Schaltung benötigt ein "Gedächtnis" um Zustände zu speichern.

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

FSM-Typ: Mealy



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

Beim *Mealy* werden die Ausgänge sowohl vom aktuellen Zustand als auch von den aktuellen Eingängen bestimmt. Es handelt sich daher um einen *0-delay-enable-type*

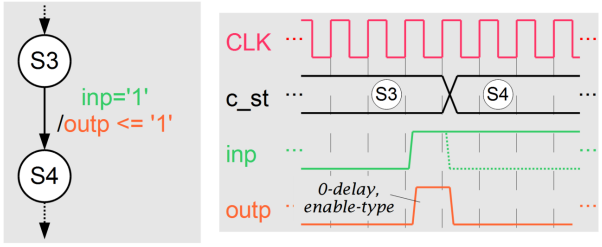
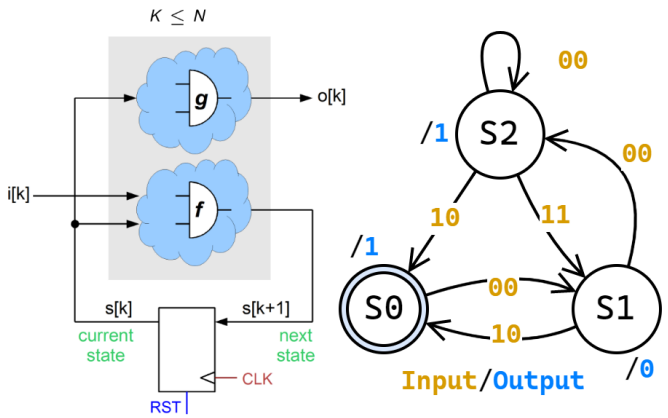


Abbildung 1: Mealy

FSM-Typ: Moore



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

Beim *Moore* werden die Ausgänge vom aktuellen Zustand bestimmt. Es handelt sich daher um einen *1-delay-state-type*

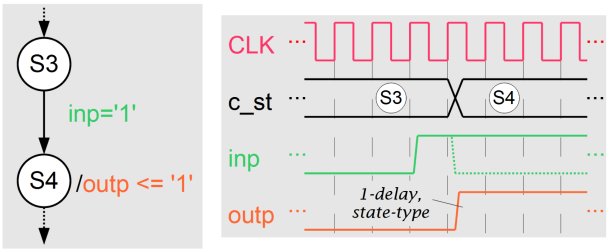


Abbildung 2: Moore

FSM-Typ: Medvedev

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

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

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

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

N : Anzahl Bits ($\widehat{=}$ Flip-Flops)
S : Anzahl verwendete Zustände

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*! Parasitäre Zustände sind umbedingt abzufangen:

```
when others =>
  n_st <= s_hold;
```

Binär

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

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

One-Hot

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

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)

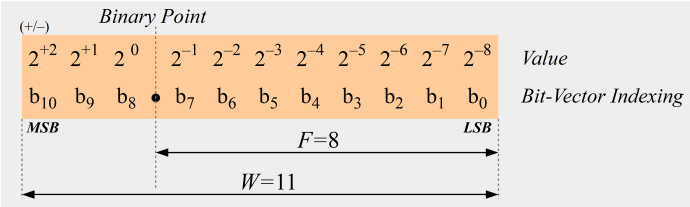
- Ausser Clock und (asynchronem) Reset dürfen keine Signale in die *sensitivity list* aufgenommen werden.
- Das den Zustand repräsentierende Signal muss einen Reset-Wert erhalten.

Latches

Latch-Warnungen bei der Synthese deuten gut auf eine Missachtung der Regeln.

Fest-/ und Gleitkomma-Arithmetik

Festkomma



$$W \geq F \geq 0$$

W : Gesamtgrösse $[Bit]$
F : Nachkomma-Bits $[Bit]$

! Vorzeichenlose Zahlen (`unsigned`) im Bereich

$$0 \leq \sum_{k=0}^{W-1} b_k \cdot 2^{k-F} \leq 2^{W-F} - 2^{-F}$$

! Zahlen mit Vorzeichen (`signed`) im Bereich

$$-2^{W-F-1} \leq -b_{W-1} \cdot 2^{W-F-1} + \sum_{k=0}^{W-2} b_k \cdot 2^{k-F} \leq 2^{W-F-1} - 2^{-F}$$

Die Auflösung R ist im Festkomma-Format im Gegensatz zu Gleitkomma über den gesamten Wertebereich konstant. Es beschreibt wie *gross* ein Bit ist.

$$R = 2^{-F} \quad \rightarrow \quad R|_{F=8} = 2^{-8} \approx 0.00391$$

Bereich ausrechnen

unsigned

$$\min = 0 \ ; \ \max = R \cdot (2^W - 1)$$

$$\text{Range} = [0, \ R \cdot (2^W))$$

Beispiel $W = 4, \ F = 3$

$$\left.\begin{array}{ll} \min: & \texttt{0.000} = 0 \\ \max: & \texttt{1.111} = 1.875 \end{array}\right\} \Rightarrow [0, \ 2)$$

signed

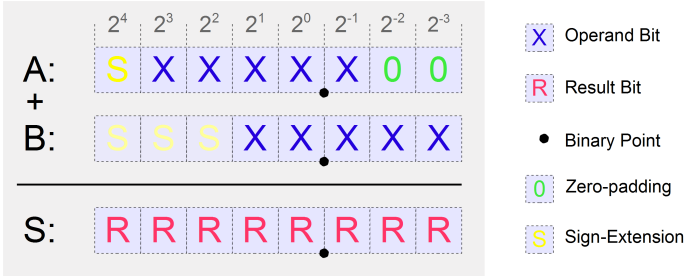
$$\min = -R \cdot (2^{W-1}) \ ; \ \max = R \cdot (2^{W-1} - 1)$$

$$\text{Range} = [-R \cdot 2^{W-1}, \ R \cdot 2^{W-1})$$

Beispiel $W = 4, \ F = 3$

$$\left.\begin{array}{ll} \min: & \texttt{1.000} = -1 \\ \max: & \texttt{0.111} = 0.875 \end{array}\right\} \Rightarrow [-1, \ 1)$$

Addition



Sign-Extension bei A wird verwendet, falls das Carry-Bit benötigt wird.

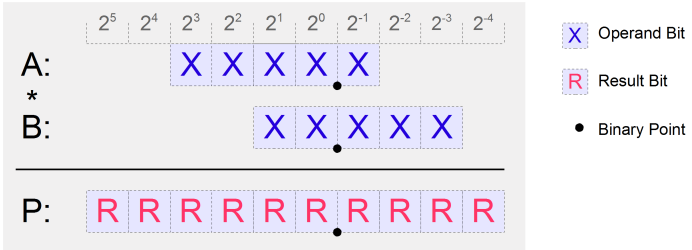
```
S <= ('0' & A & "00") + B;          ①
S <= (A(A'left) & A & "00") + B;    ②
```

① Sign-Extension = 0 falls unsigned Arithmetik
② Sign-Extension = MSB falls signed Arithmetik

$$W_S = \max(F_A, F_B) + \max(W_A - F_A, W_B - F_B) + 1$$

$$F_S = \max(F_A, F_B)$$

Multiplikation



Kein Zero-Padding oder Sign-Extension nötig. Das Produkt hat stets genauso viele *gedachte* Vor-/Nachkomma-Stelle wie die Summe der Vor-/Nachkomma-Stellen beider Operanden.

$$\texttt{P'length} = \texttt{A'length} + \texttt{B'length}$$

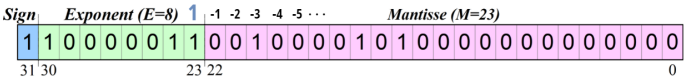
$$W_P = W_A + W_B$$

$$F_P = F_A + F_B$$

Division vermeiden

Eine Multiplikation verläuft schneller als eine Division und sollte in allen möglichen Fällen vermieden werden!

Gleitkomma



Zusätzlich werden E Bits von W Bits verwendet, um die Lage des Binärpunktes zu kodieren und $M = W - E$ Bits für die Auflösung verwendet.

$$D = (-1)^s \cdot \underbrace{(1 + m \cdot 2^{-M})}_{1+x^{-1}+x^{-2}+x^{-3}+\cdots} \cdot 2^{(e-b)}$$

$$b = 2^{E-1} - 1$$

s : Vorzeichebit (*Sign*)
m : vorzeichenloser Wert der Mantisse M
e : vorzeichenloser Wert des Exponenten E
b : Wert des Exponenten-Bias $b = 2^{E-1} - 1$

Beispiel *von oben*:

$$\underline{b} = 2^{E-1} - 1 = 2^{8-1} - 1 = \underline{127}$$

$$\begin{aligned} \underline{D} &= (-1)^1 \cdot (1 + \cdot 2^{-3} + 2^{-8} + 2^{-10}) \cdot 2^{131-127} \\ &= \underline{-18.078125} \end{aligned}$$

IEEE-754	Name	W	E	M	min / max
2008	binary16 (half)	16	5	10	$10^{\pm 5}$
1985	binary32 (single)	32	8	23	$10^{\pm 38}$
1985	binary64 (double)	64	11	52	$10^{\pm 308}$
2008	binary128 (quad)	128	15	112	$10^{\pm 4932}$

Fest- und Gleitkomma

Das Festkomma-Format (FK) kann mit einer gegebenen Anzahl W Bits

- die Auflösung (absoluter Fehler) mit $F = W$ optimieren
  `.xxxx`
- den darstellbaren Wertebereich mit $F = 0$ optimieren
  `xxxx.`
- aber nicht beides gleichzeitig! ⇒ Daher Gleitkomma

Auflösung und Fehler

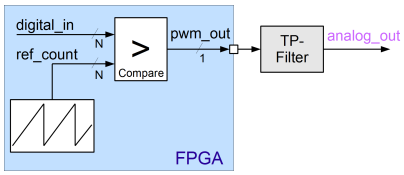
Fix-Point:
- Auflösung über Wertebereich *gleich*
- Fehler verändert sich

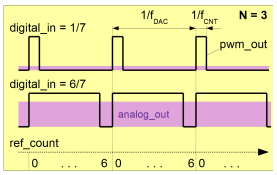
Float-Point:
- Fehler über Wertebereich *gleich*
- Auflösung verändert sich

PWM-D/A

Mit dem PWM-Verfahren (+TP-Filter) kann aus einem Digitalwert ein Analogsignal erzeugt werden. Ein Referenzcounter dimensioniert auf N-Bits zählt hoch und ab einem Schwellwert wird das PWM-Signal von High auf Low gezogen:

1. Ref-Counter $n_{cnt} < d_{in} \rightarrow$ `pwm_out` high
2. Ref-Counter $n_{cnt} \geq d_{in} \rightarrow$ `pwm_out` low

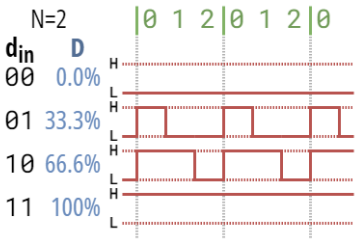




$$f_{DAC} = \frac{f_{CNT}}{2^N - 1} = \frac{f_{CLK}}{P \cdot (2^N - 1)}$$

$$D = \frac{d_{in}}{2^N - 1} \cdot 100\%$$

f_{DAC} : Frequenz des PWM-Signals
f_{CNT} : Frequenz einzelnes PWM-Bit
P : PWM-Prescaler
N : PWM-Auflösung in Bits
D : Tastgrad
d_{in} : Inputwert



Packages

In Packages sammelt man Deklarationen, die an mehreren Orten verwendet werden. Selbstdefinierte Typen, welche in Ports verwendet werden, **müssen** in Packages definiert werden. Packages mit Subprogramms **erfordern** immer eine Implementation.

```
-- Package Declaration
package my_pkg is                                    ①
  type mix_rec is record
    element1: std_logic;
    element2: natural;
  end record;
  constant const_1: natural := 7;
  function f1(a,b:mix_rec) return std_logic;
end my_pkg;

-- Package Implementation
package body my_pkg is                               ②
  function f1(a,b:mix_rec) return std_logic is       ③
  begin
    return a.element1 or b.element1;
  end f1;
end my_pkg;
```

① Package Deklaration
② Package Implementation
③ Implementation der Funktion `f1`

Libraries & Use Clause

- Design Units (Entity, Architecture, Package) sind in Libraries organisiert.
- Default-Bibliothek ist `work` → Eigene Bibliotheken in *Vivado* kann über *Source File Properties* des gewünschten Packages eingestellt/erstellt werden
- Deklarationen können auf zwei Arten zugegriffen werden

```
library myLib;                                       ①
use myLib.my_pkg.mix_rec;                            ②
use myLib.my_pkg.all;                                ③

entity E1 is                                         ④
  port(
    I : in  myLib.my_pkg.mix_rec;
    O : out myLib.my_pkg.mix_rec);
end E1;

entity E2 is                                         ⑤
  port(
    I : in mix_rec;
    O : out mix_rec);
end E2;
```

① nicht `work`-Libraries müssen mit `library` geladen werden!
② laden von spezifischen Deklarationen
③ alle Deklarationen eines Packages laden
④ Zugriff ohne `use` (direkt, ohne <2>&<3>)
⑤ Zugriff mit `use` (angenehm)

Liste von Packages

Synthetisierbare Bibliotheken

`ieee.std_logic_1164`

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

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

Nicht-Synthetisierbare Bibliotheken

```
library ieee;
use ieee.math_real.all;
```

Vivado

Project Summary

Utilization

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

Debugging

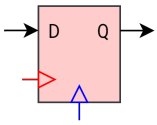
> **!** Sample Buffer Grösse des BRAMs
>
> ZEICHNUNG IM DRAWIO-PROJEKT

Vorlagen

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

Positive Getriggertes D-FlipFlop

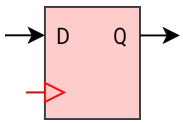
Mit asynchronem Reset




```
process (rst, clk) -- !!!
                                                    ①
begin
  if rst = '1' then
    Q <= '0';                                       ②
  elsif rising_edge(clk) then
    Q <= D;                                         ③
  end if;
end process;
```

① Deklarationen
② Asynchroner Reset
③ Getaktete Logik

Ohne Reset

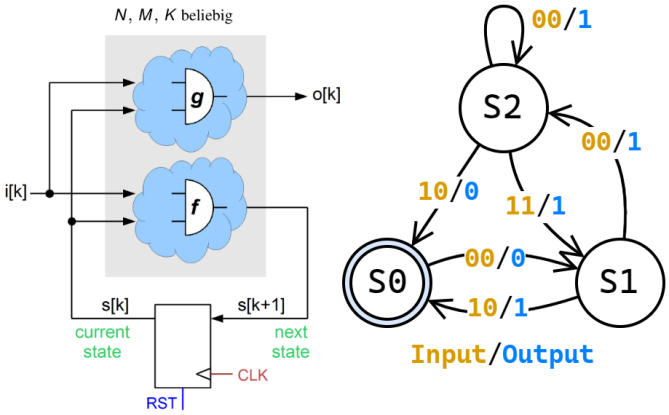



```
process (clk) -- !!!
                                                    ①
begin
  if rising_edge(clk) then
    Q <= D;                                         ②
  end if;
end process;
```

① Deklarationen
② Getaktete Logik

Finite State Machine

Mealy

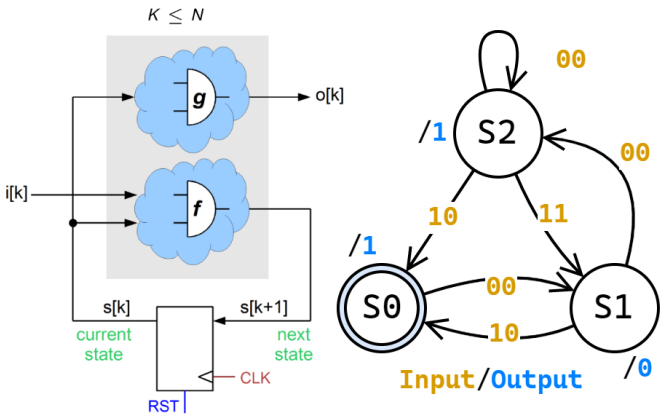



```
-- FSM initialization
type state is (S0, S1, S2);
signal c_st, n_st : state;

-- memorizing process
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;

-- memoryless 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)

Moore

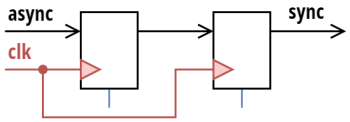



```
-- FSM initialization
type state is (S0, S1, S2);
signal c_st, n_st : state;

-- memorizing process
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;

-- memoryless 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)

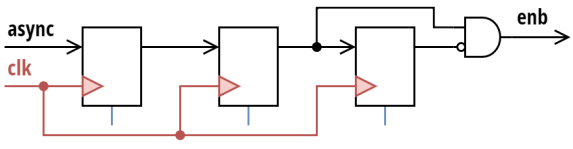
Synchronisation

einfach




```
-- initialize sync signal
signal sync: std_logic_vector(1 downto 0);

-- synchronisation
process (rst, clk)
begin
  if rst = '1' then
    sync <= "00";
  elsif rising_edge(clk) then
    sync(0) <= async;
    sync(1) <= sync(0);
  end if;
end process;
```

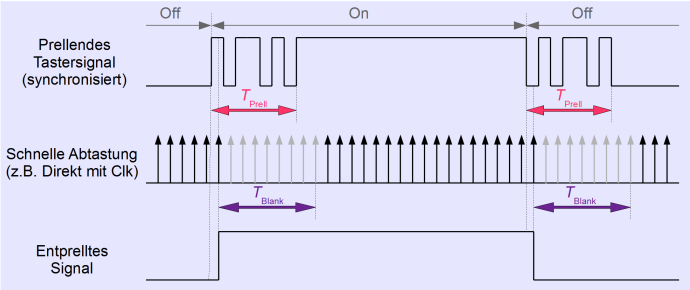
mit Flankenerkennung



```
-- initialize sync signal
signal sync: std_logic_vector(2 downto 0);

-- edge detection
enb <= sync(1) and not sync(2);

-- synchronisation
process (rst, clk)
begin
  if rst = '1' then
    sync <= "000";
  elsif rising_edge(clk) then
    sync(0) <= async;
    sync(1) <= sync(0);
    sync(2) <= sync(1);
  end if;
end process;
```

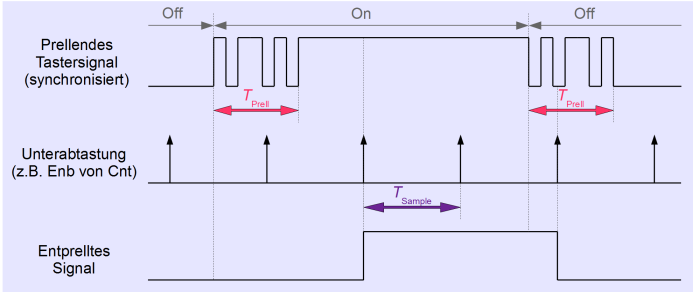
Entprellen

durch Blanking & Unterabtastung




```
deb_sig : process(clk)
begin
  if rising_edge(clk) then
    if deb_cnt = 0 then
      if (sig /= debncd_sig) then
        deb_cnt <= c_blank_time;
        debncd_sig <= sig;
      end if;
    elsif deb_cnt > 0 then
      deb_cnt <= deb_cnt -1;
    end if;
  end if;
end process;
```

durch Unterabtastung

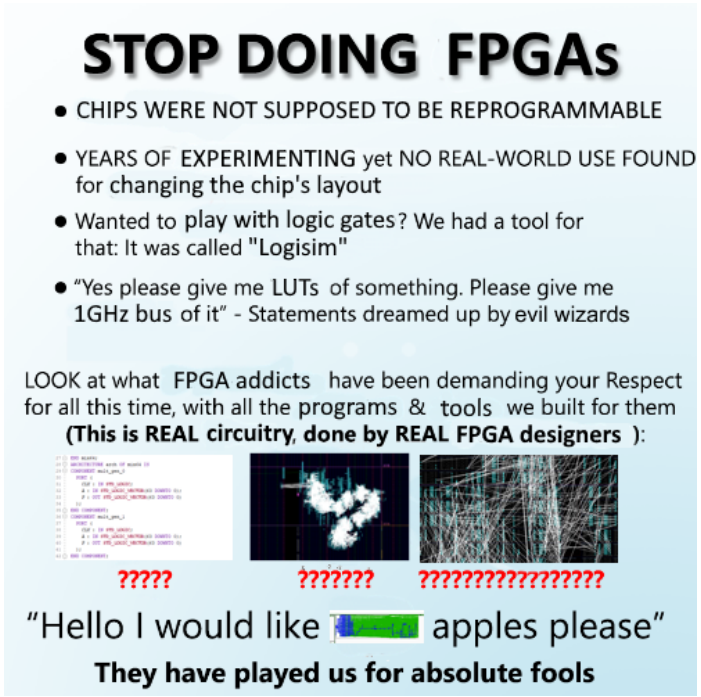



```
deb_sig : process(clk)
begin
  if rising_edge(clk) then
    if deb_cnt < c_sample_time then
      deb_cnt <= deb_cnt + 1;
    else
      debncd_sig <= sig;
      deb_cnt <= (others => '0');
    end if;
  end if;
end process;
```


IT TAKES 400,000 HOURS
TO SIMULATE THIS DESIGN...


...FOR 12 MS
imgflip.com


Vivado
Me searching for
why my design
isn't working
Warning (242)


STOP DOING FPGAs
CHIPS WERE NOT SUPPOSED TO BE REPROGRAMMABLE
YEARS OF EXPERIMENTING yet NO REAL-WORLD USE FOUND for changing the chip's layout
Wanted to play with logic gates? We had a tool for that: It was called "Logisim"
"Yes please give me LUTs of something. Please give me 1GHz bus of it" - Statements dreamed up by evil wizards
LOOK at what FPGA addicts have been demanding your Respect for all this time, with all the programs & tools we built for them
(This is REAL circuitry, done by REAL FPGA designers):
?????
???????
??????????????????
"Hello I would like apples please"
They have played us for absolute fools