1 SoC Design Hardware

1.1 System on Chip

- Repräsentiert ein komplettes System (Mikroprozessor(en), Speicher, Peripherie und anwendungsspezifische IP Blöcke) auf einem Chip
- Vorteile:
  - Kürzere Entwicklungszyklen
  - Geringere Entwicklungskosten
  - Geringere Gesamtkosten (niedriges bis mittleres Volumen)
  - Mehr Flexibilität im Betrieb
- Schlüssel zum Erreichen der folgenden Ziele: Kurze Time to market, niedrige Kosten und kleine Grösse

1.2 Zynq 7000 System

- Enthält einen Dual-Core ARM Cortex A9 Prozessor (Hard Macro) zusammen mit programmierbarer Logik auf 28-nm-Basis
- Prozessoren sind mit dem FPGA Bereich über AXI verbunden
- ARM-basierte Anwendungen können somit den vollen Nutzen aus der massiv parallelen Verarbeitung ziehen, welche im FPGA Teil möglich ist

1.2.1 Verarbeitungssystem - Processing System (PS)

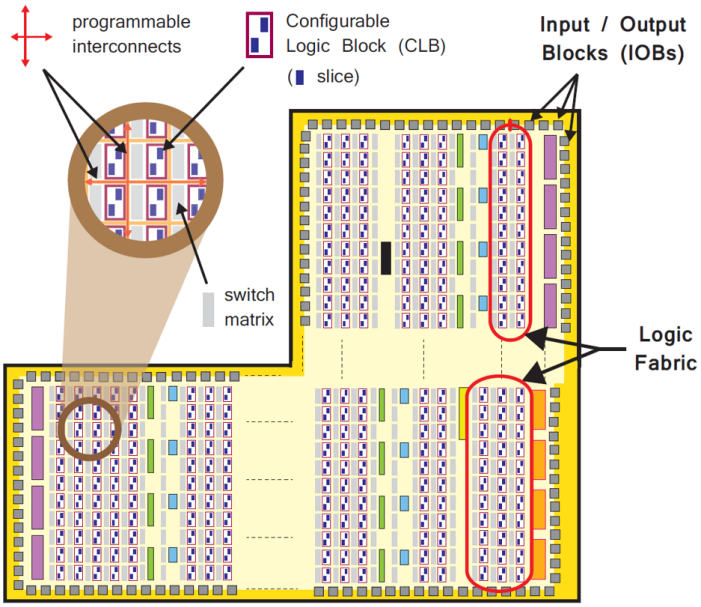
- Dual-Core ARM Cortex A9 Prozessor
- Caches
- Betrieb bis 1 GHz
- Voll integrierte Speichercontroller
- I/O Peripherie (MIO - Multiplexed In/Out to 54 Pins, EMIO - Shared with PL)

1.2.2 Programmierbare Logik (PL)

Der programmierbare Logik Teil ist nach der FPGA Struktur von XILINX Series 7 Devices aufgebaut. Er besteht aus einem regelmässigen Array von konfigurierbaren Logikblöcken (CLB) und Schaltmatrizen. Ein CLB enthält zwei Slices. Jedes Slice hat 4 Lookup-Tables (LUTs) mit sechs Eingängen, Carry-Logik, Multiplexer und acht Flip-Flops. Abhängig von der Schichtstruktur können vier Flip-Flops als Latches verwendet werden. Zu unterscheiden sind SLICEM und SLICEL. SLICEM können auch als Speicher benutzt werden. SLICEL nur für logische und arithmetische Operationen. Je nach Slicetypen wird das CLB unterschiedlich bezeichnet. Enthält ein CLB zwei SLICEL wird es mit CLB_LL bezeichnet. Wenn ein CLB ein SLICEL und ein SLICEM beinhaltet mit CLB_LM.

1.2.2.1 Spezialressourcen der programmierbaren Logik

- 7-Series Block RAM and FIFO
- DSP48E1 Slice → $P = (A+D)*B$ in einem Clockzyklus (inkl. Aufaddierung des Feedbackstranges)
- Xilinx ADC (XADC) and Analog Mixed Signal (AMS)



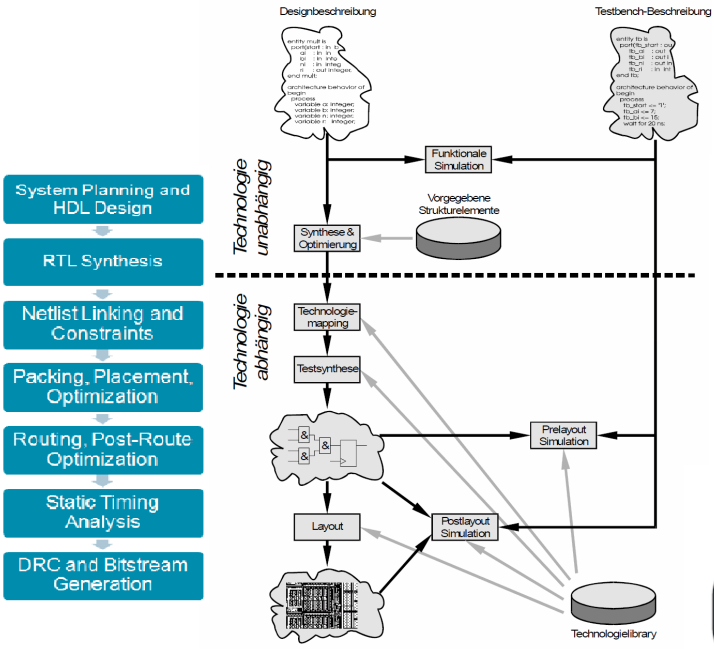
1.3 Design Herausforderungen und Bedarf

1.3.1 Herausforderungen

- Komplexität: Hohe Parallelität auf versch. Ebenen
- Heterogenität: Tools / Komponenten / Hersteller / Hardwarebeschreibung / Programmiersprache
- Time to market: Marktdruck

1.3.2 Bedarf

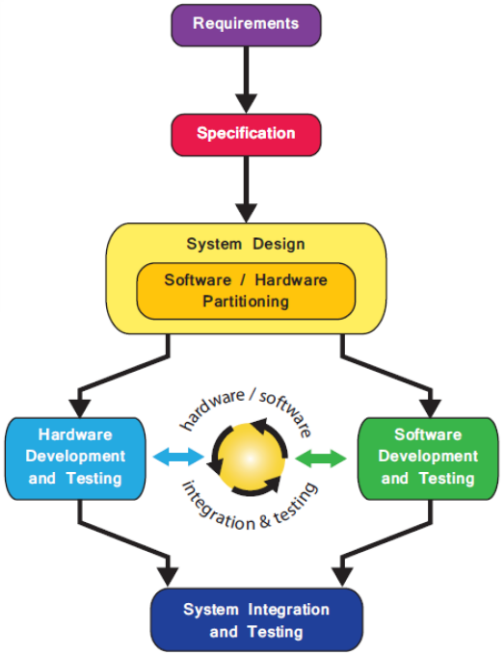
- Wiederverwendbarkeit: Systemfunktionalität kann nicht länger von Anfang an entwickelt werden → Komponenten und Subsysteme wiederverwenden → IP-basiertes Design
- Hierarchie: Weitere Hierarchiestufen sind erforderlich (sowohl Hard- wie auch Software) → IP-basiertes Design
- Parallel statt sequentiell: Hard- und Software sind parallel zu entwickeln
- Teams und Tools: Entwerfen von Systemen fordert grosse Teams und gute Werkzeuge, die die Komplexität verwalten können



1.4 Traditioneller Design Flow

1.5 System Level Design Flow

- Höhere Abstraktionsebene
- Partitionierung der SW und HW und Schnittstellendefinition
- SW/HW Co-Design
- System Integration → wird häufig unterschätzt



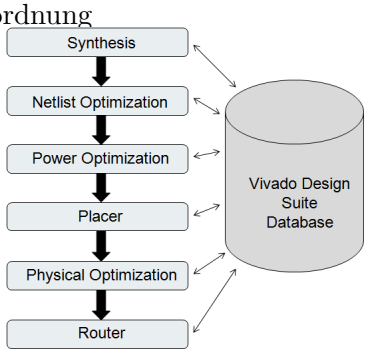
1.6 Design Tools

1.6.1 Vivado IDE Solution

1.6.1.1 Haupteigenschaften

- Interaktives Design und Analyse: Timing Analysen, Konnektivität, Ressourcen Nutzung, Timinig Constraints Analysen, ..
- RTL Entwicklung und Analyse: Entwicklung von HDL, Hierarchische Erweiterung, Schemaerzeugung
- XSIM-Simulator-Integration
- Synthese und Umsetzung in einem Package
- I/O-Pin-Planung: Interaktive regelbasierte I/O Zuordnung

1.6.1.2 Design Database Prozesse greifen auf die darunterliegende Datenbank des Designs zu. Jeder Prozess modifiziert eine vorhandene Netzliste oder erstellt eine neue. Während des ganzen Entwurfprozesses werden verschiedene Netzlisten (Elaborated, Synthesized, Implemented) verwendet.



1.6.2 Xilinx Software Development Kit (XSDK)

XSDK ist ein separates Tool von Vivado und kann eigenständig für SW-Teams installiert werden. Es ist eine voll ausgestattete Software-Design-Umgebung.

2 Constraints

2.1 ".xdc"-Dateien Physikalische Eigenschaften eines Designs können dem Synthese- und Implementierungstool mit Constraints mitgeteilt werden. Diese Constraints werden bei Vivado in ".xdc"-Dateien definiert. Sind mehrere Dateien vorhanden, so werden diese sequentiell angewendet. Wird ein Constraint mehrfach gesetzt, so wird der zuletzt gesetzte Constraint verwendet. Für jede ".xdc"-Datei kann desweiteren definiert werden, ob sie nur in der Synthese oder nur der Implementation angewendet werden soll (oder bei beidem).

```
set_property used_in_synthesis true/false [get_files xyz.xdc]
set_property used_in_implementation true/false [get_files xyz.xdc]
```

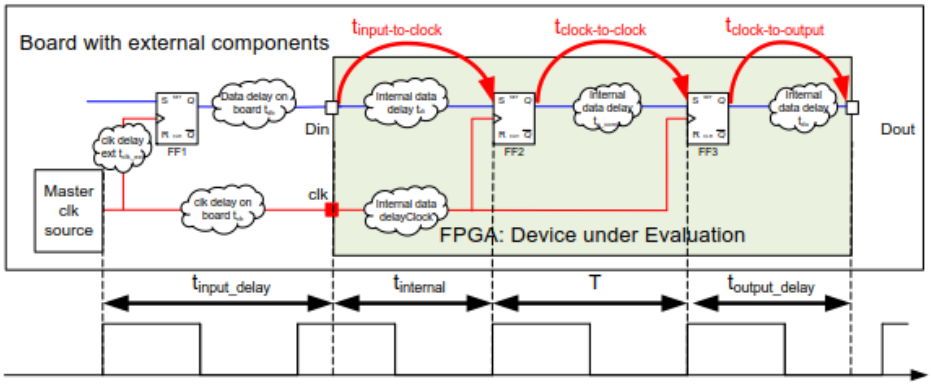
2.1.1 Reihenfolge

Es wird empfohlen die Constraints in der nachfolgenden Reihenfolge in den Dateien abzulegen, um Fehler zu minimieren.

1. Timing Assertions
   a) Primary Clocks
   b) Virtual Clocks
   c) Generated Clocks
   d) Clock Groups
   e) Input und Output Delay
2. Timing Exceptions
   a) False Paths
   b) Max Delay / Min Delay
   c) Multicycle Paths
   d) Case Analysis
   e) Disable Timing
3. Physical Constraints

2.2 Timing Analysis

Um Constraints bestimmen zu können, muss als erstes eine Timing Analyse durchgeführt werden. Eine Timing Analyse soll sicherstellen, dass alle Timing Anforderungen aller Komponenten eingehalten werden.



Eine statische Timing Analyse muss die nachfolgenden Situationen abdecken:

- Clock-to-Clock Pfad
- Input-to-Clock Pfad
- Clock-to-Output Pfad

2.2.1 Input Delay

Das Input Delay kann mit der nachfolgenden Formel bestimmt werden, weiterführende Informationen sind in Kapitel 2.5.1 zu finden.

$t_{\text{input\_delay}} = t_{\text{clk\_ext}} + t_{\text{pcq\_FF1}} + t_{\text{db}} - t_{\text{cb}}$

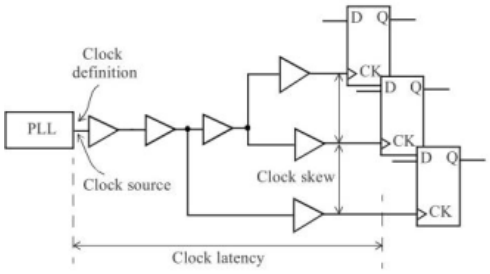
2.2.2 Output Delay

Das Output Delay kann mit der nachfolgenden Formel bestimmt werden, weiterführende Informationen sind in Kapitel 2.5.2 zu finden.

$t_{\text{output\_delay}} = t_{\text{pcq\_FF3}} + t_{\text{p\_comb}} + t_{\text{clk\_latency(FF3-master)}}$

2.2.3 Clock Skew

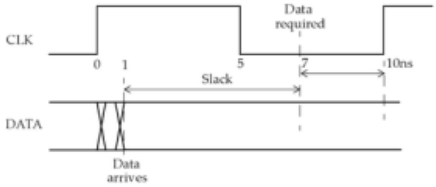
Der Skew ist die Timing Differenz zwischen zwei Signalen. Der Clock Skew ist die Differenz zwischen zwei Clock Signalen. Dieser Clock Skew sollte im Idealfall immer 0 sein. Da dies aber nicht realisierbar ist, implementieren die Synthesetools Clock Trees um den Clock Skew zu minimieren. Typischerweise ist der Clock Skew somit im Bereich von ps bis ns.



2.2.4 Slack

Der Slack bezeichnet die Differenz zwischen der benötigten Ankunftszeit eines Signales und der tatsächlichen Ankunftszeit. Solange der Slack einen positiven Wert aufweist, ist das Timing korrekt. Wird der Slack negativ, so können Timinganforderungen nicht mehr eingehalten werden.

2.2.4.1 Berechnung



$Slack = RequiredTime - ArrivalTime$ wobei $RequiredTime = Tsetup(CaptureFlipFlop)$
Nach nebenstehendem Bild: $RequiredTime = 10ns - 3ns$
$ArrivalTime = 1ns$ ergibt $Slack = 7ns - 1ns = 6ns$

2.3 Clocks

Clocks müssen definiert werden, damit die Timing Paths berechnet werden können. Ein Clock wird dabei so definiert, wie er am Startpunkt des Clock Trees aussieht (in der Regel der Eingangspin des Clocks). Es wird angegeben, wie gross die Periode des Clocks ist, sowie wo die Flanken sind. Alle Zeiten werden dabei in Nanosekunden angegeben.

2.3.1 Primary Clocks

Ein Primary Clock ist ein Clock, welcher in der Regel durch einen Eingangspin zugeführt wird. Ein solcher Clock muss immer einem Netzlistenobjekt zugewiesen werden.
Der nachfolgende Befehl definiert einen neuen Clock mit dem Namen *devclk*, einer Periode von 10ns sowie einem Dutycycle von 25%. Der Clock wird über den Port *clkin* zugeführt.

```
create_clock -name devclk -period 10 -waveform {2.5 5} [get_ports clkin]
```

2.3.2 Generated Clocks

Generated Clock werden von speziellen Blöcken in einem FPGA generiert (z.B. MMCM). Sie können auch von Userlogik erzeugt werden. Diese Generated Clocks sind jedoch an einen anderen Clock gebunden und teilen diesen

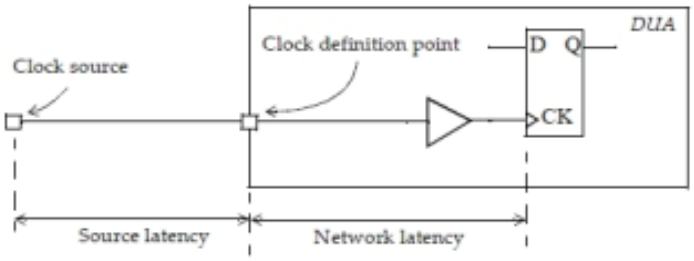
z.B. herunter, oder bewirken eine Phasenverschiebung.

2.3.3 Virtual Clocks

Ein Virtual Clock ist ein Clock welcher physikalisch nicht existiert und somit nicht an ein Netzlisten-Objekt gebunden ist. Sie werden benutzt um Input- und Output-Delays zu spezifizieren, wenn das externe Gerät mit einem anderen Clock läuft als der FPGA.
Der nachfolgende Befehl definiert einen neuen Virtual Clock mit dem Namen *virtclk*, einer Periode von 10ns sowie einem Dutycycle von 25%.

```
create_clock -name virtclk -period 10 -waveform {2.5 5}
```

2.3.4 Clock Latency



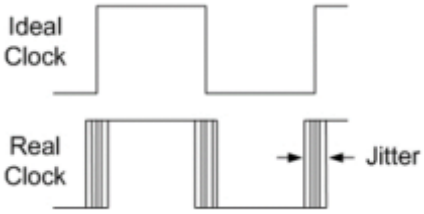
Clocks erreichen verschiedene Punkte in einem Design mit unterschiedlichen Verzögerungen. Diese Verzögerungen können in zwei Arten unterschieden werden:

- Network Latency
- Source Latency

Mit den nachfolgenden Befehlen können die Latenzen definiert werden. Wird nichts spezifisch angegeben, so werden der *min*, *max*, *rise* und *fall* Wert gesetzt.

```
# Network Latency
set_clock_latency 0.8 [get_clocks clkname]
# Source Latency
set_clock_latency 1.9 -source [get_clocks clkname]
```

2.3.5 Clock Jitter



Aufgrund der physikalischen Eigenschaften ist kein Clock ideal. Kleinere Schwankungen in der Übertragung können auftreten. Diese Schwankungen werden Jitter genannt und in Nanosekunden gemessen.

2.3.5.1 Input Jitter Input Jitter definiert Jitter auf Primary Clocks. Mit dem nachfolgenden Befehl kann der Jitter definiert werden:

```
# Set a jitter of 0.3 ns
set_input_jitter clockname 0.3
```

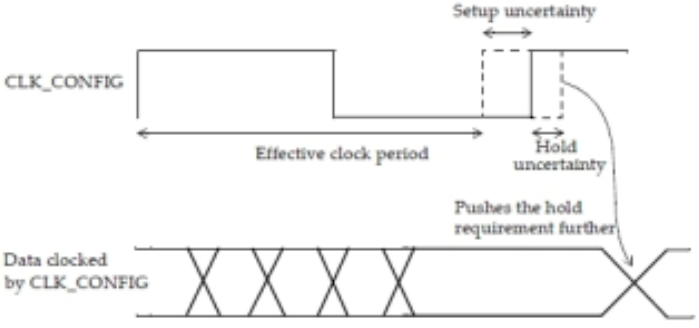
2.3.5.2 System Jitter Der System Jitter spezifiert den Jitter für alle Clocks im System (auch für die Primary Clocks). Er wird benutzt um starkes Rauschen zu modellieren. Mit dem nachfolgenden Befehl kann der Jitter definiert werden:

```
# Set a jitter of 0.1 ns
set_system_jitter 0.1
```

2.3.6 Zusätzliche Clock Unsicherheit

Sind weitere Unsicherheiten zwischen verschiedenen Clocks vorhanden, so kann mit dem nachfolgenden Befehl definiert werden, wie sich verschiedene Clocks zueinander verhalten.

```
# Set a uncertainty of 0.225ns between all
# clock domains
set_clock_uncertainty 0.225 -from
    [get_clocks] -to [get_clocks]
```




```
set_property PACKAGE_PIN W10 [get_ports cam_pclk]
set_property PACKAGE_PIN AB10 [get_ports {cam_din[1]}]
set_property PACKAGE_PIN AA11 [get_ports {cam_din[0]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports cam_pclk]
set_property IOSTANDARD LVCMOS33 [get_ports {cam_din[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {cam_din[0]}]
```

2.5 I/O Delay

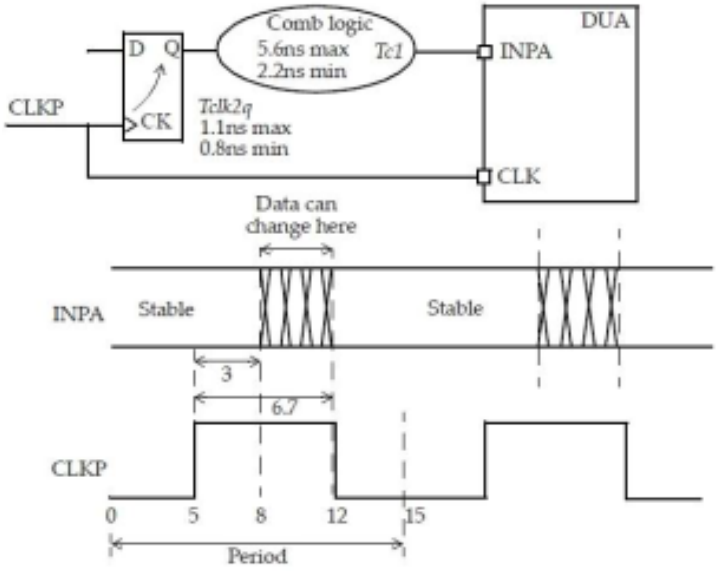
Damit externe Timing-Anforderungen korrekt in die Synthese und Implementation einbezogen werden können, ist es notwendig diese anzugeben.

2.5.1 Input Delay

Input Delays geben an, mit welcher Verzögerung Daten an einem Eingangsport anliegen. Diese Verzögerung muss relativ zu einem Clock angegeben werden.

2.5.1.1 Tcl Befehl Mit dem Befehl `set_input_delay` kann das Input-Delay angegeben werden. Die folgenden Parameter werden dabei unterstützt:

- `-clock`: Gibt den Clock an, zu welchem die Verzögerung gilt.
- `-min`, `-max`: Definiert die min Zeit (hold/removal) oder die max Zeit (setup/recovery). Wird dieser Parameter nicht spezifisch angegeben, so wird die Input-Delay-Zeit für min und max verwendet.
- `-clock_fall`: Gibt an, dass der Input-Delay relativ zur fallenden Clockflanke gilt.
- `-rise`, `-fall`: Gibt an, für welche Flanke des Eingangssignales die Angaben gelten.
- `-add_delay`: Diese Option wird benutzt, wenn ein zweites Delay angegeben werden muss (z.B. bei DDR).



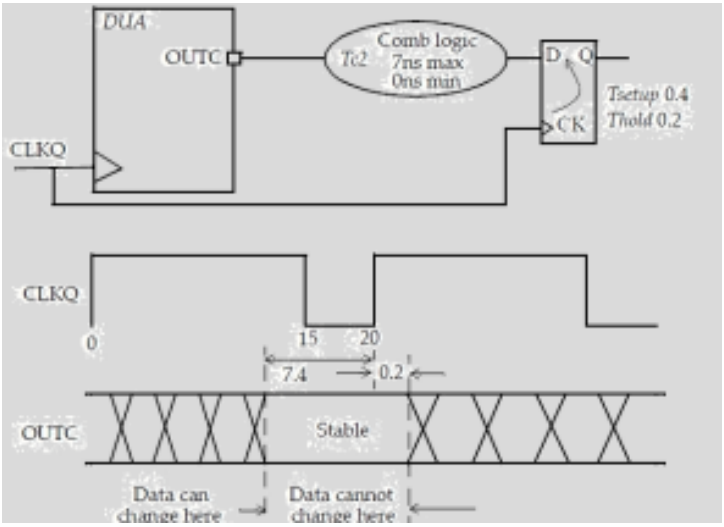

```
create_clock -name CLKP -period 15 -waveform {5 12} [get_ports CLKP]
set_input_delay -clock CLKP -max 6.7 [get_ports INPA]
set_input_delay -clock CLKP -min 3.0 [get_ports INPA]
```

2.5.2 Output Delay

Output Delays geben an, in welchem Zeitbereich die Daten am Ausgangsport stabil sein müssen. Dieser Zeitbereich wird wiederum relativ zu einem Clock angegeben.

2.5.2.1 Tcl Befehl Mit dem Befehl `set_output_delay` kann das Output-Delay angegeben werden. Die folgenden Parameter werden dabei unterstützt:

- `-clock`: Gibt den Clock an, zu welchem die Zeit gilt.
- `-min`, `-max`: Definiert die min Zeit (hold/removal) oder die max Zeit (setup/recovery). Wird dieser Parameter nicht spezifisch angegeben, so wird die Input-Delay-Zeit für min und max verwendet.
- `-clock_fall`: Gibt an, dass der Output-Delay relativ zur fallenden Clockflanke gilt.
- `-rise`, `-fall`: Gibt an, für welche Flanke des Ausgangssignales die Angaben gelten.
- `-add_delay`: Diese Option wird benutzt, wenn ein zweites Delay angegeben werden muss (z.B. bei DDR).




```
create_clock -name CLKP -period 20 -waveform {0 15} [get_ports CLKQ]
set_output_delay -clock CLKQ -max 7.4 [get_ports OUTC]
set_output_delay -clock CLKQ -min -0.2 [get_ports OUTC]
```

3 System Level VHDL

3.1 Wiederverwendbarkeit

Bereits auf der untersten Abstraktionsebene soll wiederverwendbarer Code geschrieben werden. Um einen Code wiederverwenden zu können, muss dieser gut lesbar sein. Nachfolgend ein paar Hilfsmittel um den Code gut lesbar zu gestalten.

3.1.1 Kommentare

Zeilenkommentare → beginnen mit --
Blockkommentare → beginnen mit /* und enden mit */

3.1.2 Namenskonventionen

In VHDL sind keine Namenskonventionen definiert. Es wird jedoch empfohlen folgende Mindestregeln einzuhalten:

- Aussagekräftige Namen für Entities, Architekturen, Funktionen und Prozesse verwenden
- Namen sollten lowercase sein und zum Trennen von Wörtern sollte man Underscores verwenden
- Entities sollten eindeutige Namen haben. Architekturen benötigen keine eindeutigen Namen. Ihr Name beschreibt eher die Natur der Architektur wie RTL, Struktur usw
- Signalnamen sollten lowercase sein und zum Trennen von Wörtern sollte man Underscores verwenden
- Low-Aktive Signale sollten deutlich als solche im Signalnamen markiert werden (*XXXX_l* oder *XXXX_n*)

Spezielle Namenskonventionen ermöglichen es dem Vivado IP-Packager, automatisch AXI-Schnittstellensignale abzuleiten:

- Reset: _reset, _rst, _resetn (low-aktiv), _areset
- Clock: _clk, _clkin, _clk_p (Diff. Clock), _clk_n (Diff. Clock)
- AXI Interface: _tdata (Bsp: s0_axis_tdata), _tvalid, _tready

3.1.3 Einsatz von Konstanten

Wenn möglich nie Parameter gebrauchen! Konstanten sind in Bezug auf die Änderbarkeit unverzichtbar. Konstanten in VHDL-Packages können in mehreren Designeinheiten verwendet werden. Konstanten, die in Designentitäten (Deklarationsteil der Architektur) deklariert sind, können in der gesamten Architektur einschliesslich der Prozesse innerhalb dieser Architektur gesehen werden. Der Scope einer Konstante, welche in einem Prozess deklariert wurde, ist auf diesen Prozess beschränkt.

```
constant constant_name : type := value; -- Definition
constant countersize : integer := 2**16-1; -- Example 1
constant cmdreadid : std_logic_vector(7 downto 0) := x"9F"; -- Example 2
```

3.1.4 Einsatz von Aliases

```
signal DataBus : std_logic_vector(31 down to 0);
alias FirstNibble : std_logic_vector(0 to 3) is DataBus(31 downto 28);
```

3.1.5 Einsatz von Generics

Generics werden zu Beginn der Entity deklariert.

```
generic(param_name : param_type := initial_value);  -- Definition
entity en_ffp_data_handler is                       -- Example
  generic (
            AddrWidth_g       : positive  := 16;
```

```
            DataWidth_g       : positive  := 32;
            Enable_StreamUp   : boolean   := true;
        );
  port  (
            M_AXIS_TVALID     : out std_logic;
            M_AXIS_TDATA      : out std_logic_vector(DataWidth_g-1 downto 0);
        );
end en_ffp_data_handler;
```

3.2 Funktionen Funktionen sind Subprogramme mit einer Argumentenliste von nur Eingängen. Sie geben einen einzigen Wert eines spezifizierten Types zurück. Funktionen können entweder im Deklarationsteil einer Architektur oder in einem Package (flexibler) definiert werden.

```
-- Syntax
function <function_name> [(list of arguments with type declaration)] return <type> is
  [<Declarations>];
begin
  {<sequential statements,
  except wait statements>};
  return <return expression>;
end [function] <funcionname>;
-- Example
library ieee;
use ieee.std_logic_1164.all;
entity parity_gen8 is
  port(
    A : in std_ulogic_vector(3 downto 0);
    B : in std_ulogic_vector(7 downto 0);
    POA, POB : out std_ulogic
  );
end entity parity_gen8;
architecture RTL of parity_gen8 is
  function PARGEN(AVECT : std_ulogic_vector) return std_ulogic is
    variable PO_VAR : std_ulogic;
  begin
    PO_VAR := '1';
    for I in AVECT'range loop
      if AVECT(I) = '1' then
          PO_VAR := not PO_VAR;
      end if;
   end loop;
    return PO_VAR;
  end function PARGEN;
begin
  POA <= PARGEN(A);
  POB <= PARGEN(B);
end architecture RTL;
```

Wichtig:

- Im Funktionsblock dürfen keine wait-Anweisungen oder Signalzuweisungen enthalten sein!
- := wird verwendet, wenn ein Wert einer **Variablen** zugewiesen wird. Wird sofort in einem Prozess zugeordnet.
- <= wird verwendet, wenn ein Signal einem Signal zugewiesen wird. Wird am Ende eines Prozesses zugewiesen.

Unterschied pure und impure Funktionen: Bei Funktionen, welche pure sind, bekommt man bei jedem Aufruf für jeden Input den gleichen Output (z.B. sin(x)). Bei impuren Funktionen erhält man bei gleichem Input unterschiedliche Outputs. Impure Funktionen haben Seiteneffekte, wie z.B. das Updaten von Objekten ausserhalb ihres Scopes, was bei puren Funktionen nicht erlaubt ist.

```
variable number : Integer := 0;
impure function Func(A : Integer) return Integer is
  variable counter : Integer;
begin
```

```
  counter := A * number;
  number := number + 1;
  return counter;
end Func;
```

3.3 Prozeduren Prozeduren sind sehr ähnlich wie Funktionen. Der Hauptunterschied ist, dass bei Prozeduren mehrere Ein- und Ausgangsvariablen definiert werden können.

```
-- Syntax
procedure <procedure_name> [( < argument list with type declaration >)] is
  [<declarations >];
begin
  (sequential statements >};
end [procedure] <procedure_name >;
-- Example
procedure Proc(X,Y : inout Integer) is
  type Word_16 is range 0 to 65536;
  subtype Byte is Word_16 range 0 to 255;
  variable Vb1,Vb2,Vb3 : Real;
  constant Pi : Real := 3.14;
begin
  -- Some statements
end procedure Proc_3;
```

Wichtig:

- Prozeduren können In-, Out- oder Inout-Parameter besitzen. Diese können ein Signal, eine Variable oder eine Konstante sein. Die Voreinstellung für in-Parameter ist konstant, für out und inout variabel.

3.4 Packages Konstanten, Typen, Komponenten, Funktionen und Prozeduren, die an verschiedenen Stellen in einem oder mehreren Projekten verwendet werden, können in Packages gruppiert werden.

```
-- Syntax
[<library und use statements >] -- Package Declaration
package <package_name> is
  <declarations >;
end <package_name >;
[<library und use statements >] -- Package Body
package body <package_name> is
  <list of definitions >;
end <package_name >;
-- Example
library ieee;
use ieee.std_logic_1164.all;
package parity_package is
  constant nibble : integer; -- Declaration and initialization optional
  constant word : integer; -- Declaration
  function PAR_GEN(AVECT : std_ulogic_vector) return std_ulogic;
end package parity_package;
package body parity_package is
  function PAR_GEN(AVECT : std_ulogic_vector) return std_ulogic is
    variable PO_VAR : std_ulogic;
  begin -- begin of function
    PO_VAR := '1';
    for I in AVECT'range loop
      if AVECT(I) = '1' then
        PO_VAR := not PO_VAR;
      end if;
    end loop;
    return PO_VAR;
  end function PAR_GEN; -- end of function
  constant nibble : integer := 4; -- Declaration and initialization
  constant word : integer := 8; -- Declaration and initialization
  --may contain more functions, procedures, etc.
end package body parity_package;
```

Packages werden in Bibliotheken kompiliert abgelegt (Standard = work library). Sie können im VHDL-Modul mit der use-Anweisung verwendet werden:

```
use work.my_package.all;
```

4 Fixed Point Arithmetic

4.1 Einleitung

	Unsigned integer	Signed integer
Formel	$z = \sum_{k=0}^{n-1} a_k \cdot 2^k$	$z = -(2^{n-1}) \cdot a_{n-1} + \sum_{k=0}^{n-2} 2^k \cdot a_k$
Bereich	$0 \dots 2^n - 2^0$	$-2^{n-1} \dots 2^{n-1} - 2^0$

4.2 Qn.m Format

n: Anzahl integer bits (inkl. eines allfälligen Vorzeichenbits); m: Anzahl der Nachkommastellen
Es gibt zwei Möglichkeiten, eine Fix-point Zahl in VHDL zu implementieren:

- Nur mit dem 'numeric_std'-Package und dabie unsigned und signed Typen zu verwenden. Der Programmierer muss praktisch alles selbst machen (z.B. Dezimalpunkt tracken)
- Mithilfe des 'fixed_pkg-Packages' (basiert auf 'numeric_std') → einfacherer, übersichtlicherer Code; fast alle Operatoren sind für integer und reelle Typen überladen; Overflows sind nicht möglich → führt immer zu sicherer Implementation, aber nicht immer zur optimalen Lösung in Bezug auf die Vektorbreite

Sizing-Regeln für 'fixed_pkg':

Operation	Result Range
$A + B$	Max(A'left,B'left)+1 downto Min(A'right,B'right)
$A - B$	Max(A'left,B'left)+1 downto Min(A'right,B'right)
$A \cdot B$	A'left+B'left+1 downto A'right+B'right
A rem B	Min(A'left,B'left)+1 downto Min(A'right,B'right)
Signed /	A'left-B'right+1 downto A'right-B'left
Signed A mod B	Min(A'left,B'left) downto Min(A'right,B'right)
Signed Reciprocal(A)	-A'right downto -A'left-1
Abs(A)	A'left+1 downto A'right
$-A$	A'left+1 downto A'right
Unsigned/	A'left-B'right downto A'right-B'left-1
Unsigned A mod B	B'left downto Min(A'right,B'right)
Unsigned Reciprocal(A)	-A'right+1 downto -A'left

4.2.1 ufixed/sfixed

Das Package definiert zwei neue Datentypen: ufixed und sfixed. Für negative zahlen wird das 2er-Komplement verwendet. Die Position des Dezimalpunkts ist zwischen Index "0" und "-1".

```
signal uNumber : ufixed(3 downto -6); -- uQ4.6
signal sNumber: sfixed(3 downto -6); -- sQ4.6
signal fractional : ufixed(-2 downto -3); -- "11" represents 0.011 = 0.375
signal integer : sfixed(3 downto 1); -- "111" represents 1110.0 = -2
-- upper and lower index using integer numbers
uNumber <= to_ufixed(5.25, 3, -6);
-- upper and lower index using the 'high and 'low operator
sNumber <= to_sfixed(-5.25, sNumber'high, sNumber'low);
```

4.2.2 Limitierte Unterstützung

Das Package ist Teil von VHDL 2008, weshalb es von gewissen Tools noch nicht vollständig unterstützt wird. XILINX bietet eine modifizierte Version von 'fixed_pkg' an.

4.2.3 Unsigned/Signed

	uQn.m	uQn.m
Formel	$z = \sum_{k=-m}^{n-1} a_k \cdot 2^k$	$z = -(2^{n-1}) \cdot a_{n-1} + \sum_{k=-}^{n-2} a_k \cdot 2^k$
Bereich	$0 \dots 2^n - 2^{-m}$	$-2^{n-1} \dots 2^{n-1} - 2^{-m}$

Bsp.: uQ2.6

```
-- 2.5248 -> binary 10.10000110010110010101
constant REAL_NUM : real := 2.5248;
-- unsigned Q2.6 = 10.100001 = 2.5156 (truncated)
-- the decimal point has to be shifted to the right by 6
```

```
signal uNumber : unsigned(7 downto 0) :=  to_unsigned(integer(2.0**6 * REAL_NUM), 8);

-- fixed_pkg
signal uNumber : ufixed(1 downto -6) := to_ufixed(REAL_NUM, 1, -6);
```

Bsp.: sQ2.6

```
-- -2.5248 -> binary 101.0111100110101
constant REAL_NUM : real := -2.5248;
-- signed Q3.5 = 101.01111 =  -2.5313 (truncated)
-- the decimal point has to be shifted to the right by 5
signal sNumber : signed(7 downto 0) := to_signed(integer(2.0**5 * REAL_NUM), 8);

-- fixed_pkg
signal sNumber : sfixed(1 downto -6) := to_sfixed(REAL_NUM, 1, -6);
```

4.3 Arithmetic Functions

4.3.1 Addition/Subtraction

4.3.1.1 Unsigned and Unsigned

Praktisch identisch mit Add./Sub. mit normalen Integer-Zahlen, einzig muss nun der Programmierer den Dezimalpunkt tracken.

	0	1	0	1	1	0	1	**0**	= 101.101 in uQ3.3 = 5.625
+				1	0	0	1	1	= 1.0011 in uQ1.4 = 1.1875
	0	1	1	0	1	1	0	1	= 0110.1101 in uQ4.4 = 6.8125

```
constant AUGEND : real := 5.625; -- binary 101.101
constant ADDEND : real := 1.1875; -- binary 1.0011
signal uAugend : unsigned(5 downto 0) := to_unsigned(integer(2.0**3 * AUGEND), 6); -- u
signal uAddend : unsigned(4 downto 0) := to_unsigned(integer(2.0**4 * ADDEND), 5); -- u
-- result uQ4.4 -> Q(max(n1,n2)+1).(max(m1,m2))
signal sum : unsigned(7 downto 0);
-- the '0' left of uAugend is a guard bit to reach the same width as the sum vector.
-- the '0' right of uAugend is needed to align the decimal point
sum <= ('0' & uAugend & '0') + uAddend; -- result uQ4.4

-- fixed_pkg
signal uAugend : ufixed(2 downto -3) := to_ufixed(AUGEND, 2, -3); -- uQ3.3
signal uAddend : ufixed(0 downto -4) := to_ufixed(ADDEND, 0, -4); -- uQ1.4
-- Q3.4 -> Q(max(n1,n2)+1).(max(m1,m2)) -> +1 to prevent overflow
signal sum : ufixed(3 downto -4);
-- here is the guarding and alignmnt done automatically
sum <= uAugend + uAddend;
```

4.3.1.2 Signed and Signed

Auch fast identisch, Zahl mit weniger Vorkommabits muss sign-extended werden

	1	1	0	1	1	0	1	**0**	= 101.101 in sQ3.3 = -2.375
+	**1**	**1**	**1**	1	0	0	1	1	= 1.0011 in sQ1.4 = -0.8125
	1	1	0	0	1	1	0	1	= 1100.1101 in sQ4.4 = -3.1875

```
-- fixed_pkg
signal sMinuend : sfixed(2 downto -3) := to_sfixed(MINUEND, 2, -3); -- Q3.3
signal sSubtrahend : sfixed(0 downto -4) := to_sfixed(SUBTRAHEND, 0, -4); -- Q1.4
-- Q3.4 -> Q(max(n1,n2)+1).(max(m1,m2)) -> +1 to prevent overflow
signal difference : sfixed(3 downto -4);
-- here is the guarding and alignment done automatically
difference <= sMinuend - sSubtrahend;
```

4.3.2 Multiplication

4.3.2.1 Unsigned by Unsigned

Einfachster Fall, partielle Produkte werden ohne sign-extension addiert

				1	1	0	1	= 11.01 in uQ2.2 = 3.25 (multiplicand)
			1	0	1	1	= 10.11 in uQ2.2 = 2.75 (multiplier)	
				1	1	0	1	
			1	1	0	1		
		0	0	0	0			
	1	1	0	1				
1	0	0	0	1	1	1	1	= 1000.1111 in uQ4.4 = 8.9375

```
-- fixed_pkg
signal uMultiplicand : ufixed(1 downto -2) := to_ufixed(U_MULTIPLICAND, 1, -2); -- uQ2.
signal uMultiplier : ufixed(1 downto -2) := to_ufixed(U_MULTIPLIER, 1, -2); -- uQ2.2
--  uQ4.4 -> Q(n1+n2).(m1,m2)
signal uProduct : ufixed(3 downto -4);
uProduct <= uMultiplicand * uMultiplier;
```

4.3.2.2 Signed by Unsigned

Hier braucht jedes partielle Produkt sign-extension. In VHDL ist signed·unsigned nicht erlaubt. Der vorzeichenlose Wert muss zuerst in eine vorzeichenbehaftete Zahl gewandelt werden.

				1	1	0	1	= 11.01 in sQ2.2 = -0.75 (multiplicand)
				0	1	0	1	= 01.01 in sQ2.2 = 1.25 (multiplier)
1	**1**	**1**	**1**	1	1	0	1	
0	**0**	**0**	0	0	0	0		
1	**1**	1	1	0	1			
0	0	0	0	0				
1	1	1	1	0	0	0	1	= 1111.0001 in sQ4.4 = -0.9375

4.3.2.3 Signed by Signed

Zwei Möglichkeiten:

- Sign-extension aller partiellen Produkte. Wenn der 'multiplier' negativ ist, muss das letzte partielle Produkt mit dem 2er-Komplement transformiert werden.
- Beide Zahlen werden mit dem 2er-Komplement transformiert. Das partielle Produkt wird sign-extended, wenn der neue 'multiplicand' negativ ist.

Das MSB des Produkts ist redundant. Wird das Produkt nach links geschoben, kann das redundante Bit entfernt und ein zusätzliches fractional bit hinzugefügt werden (neues Format: Q(n1+n2-1).(m1+m2+1)).

				0	1	0	1	= 01.01 in sQ2.2 = 1.25 (multiplicand)
				1	0	1	1	= 10.11 in sQ2.2 = -1.25 (multiplier)
0	**0**	**0**	**0**	0	1	0	1	
0	**0**	**0**	0	1	0	1		
0	**0**	0	0	0	0			
1	1	0	1	1				2's complement of multiplicand 0101
1	1	1	0	0	1	1	1	110.01110 in sQ3.5 = -1.5625
								(one of the sign bits is removed)
				1	0	1	1	= 2's complement of multiplicand 0101
				0	1	0	1	= 2's complement of multiplier 1011
1	**1**	**1**	**1**	1	0	1	1	multiplicand is negative → extended sign-bits
0	**0**	**0**	0	0	0	0		
1	**1**	1	0	1	1			
0	0	0	0	0				
1	1	1	0	0	1	1	1	110.01110 in sQ3.5 = -1.5625

.

```
constant S_MULTIPLICAND : real := 1.25; -- binary 01.01
constant S_MULTIPLIER : real := -1.25; -- binary 10.11
signal sMultiplicand : signed(3 downto 0) := to_signed(integer(2.0**2 * S_MULTIPLICAND)
signal sMultiplier : signed(3 downto 0) := to_signed(integer(2.0**2 * S_MULTIPLIER), 4)
signal sProduct : signed(7 downto 0);
```

```
-- Q3.5 -> Q(n1+n1-1).(m1+m2+1)
sProduct <= shift_left(sMultiplicand * sMultiplier, 1);

-- fixed_pkg
signal sMultiplicand : sfixed(1 downto -2) := to_sfixed(S_MULTIPLICAND, 1, -2); -- sQ2.
signal sMultiplier : sfixed(1 downto -2) := to_sfixed(S_MULTIPLIER, 1, -2); -- sQ2.2
-- Q4.4 -> Q(n1+n2).(m1+m2)
signal sProduct : sfixed(3 downto -4);
signal sProductShift : sfixed(2 downto -5);
-- sProduct has two sign bits
sProduct <= sMultiplicand * sMultiplier;
-- sProductShift has only one sign bit
sProductShift <= resize(sProduct,2,-5);
```

Corner case: Wenn die zwei kleinst möglichen Zahlen multipliziert werden (bei sQ2.2: 1000*1000). Dies führt zu einem Overflow und zu einem falschen Ergebnis (-4 anstatt +4). → **Lösung:** Das zweite Vorzeichenbit als Schutz stehen lassen. Dies ist so auch im 'fixed_pkg'-Package umgesetzt

4.3.3 Division

Divisionen mit Mehrfachen von 2 (2^n) sind einfache Schiebeoperationen. Für andere Werte ist es komplizierter. 'fixed_pkg' bietet eine elegante Lösung an:

```
-- fixed_pkg
signal uDividend : ufixed(4 downto -3) :=  to_ufixed(6.75, 4, -3); -- uQ5.3
signal uDivisor : ufixed(2 downto -4) := to_ufixed(1.5, 2, -4); -- uQ3.4
-- uQ9.6 -> Q(n1+m2-1).(m1+n2)
signal uFraction : ufixed(8 downto -6);
signal sDividend : sfixed(4 downto -3) := to_sfixed(-6.75, 4, -3); -- Q5.3
signal sDivisor : sfixed(2 downto -4) := to_sfixed(1.5, 2, -4); -- uQ9.6 -> Q(n1+m2-1)
signal sFraction : sfixed(9 downto -5);
uFraction <= uDividend/uDivisor;
sFraction <= sDividend/sDivisor;
```

4.4 Overflow and Saturation

Bei Verwendung des 'fixed_pkg'-Packages werden Overflow und Saturation automatisch gelöst. Ansonsten gibt es zwei Lösungen mit einem Over-/Underflow umzugehen: Das Resultat sättigen (Maximum oder Minimum) oder das Resultat erhält ein zusätzliched integer bit.

Erkennung bei vorzeichenbehafteten Zahlen:

- Overflow, wenn Summe von zwei positiven Zahlen ein negatives Resultat liefert.
- Underflow, wenn Summe von zwei negativen Zahlen ein positives Resultat liefert.
- Die Summe einer positiven und negativen Zahl ergibt nie einen Underflow.

Erkennung bei vorzeichenlosen Zahlen:

- Overflow, wenn Summe einer Addition kleiner als der erste Summand ist.
- Underflow, wenn Differenz einer Subtraktion grösser als der Subtrahend ist.

fixed_pkg: Der einzige Fall, dass ein Underflow entstehen kann, ist wenn das Resultat resized wird in eine kleiner Grösse. Die resize-Funktion von 'numeric_std' wird überladen:

```
function resize (
  arg             : unresolved_ufixed; -- input
  constant left_index : integer; -- integer portion
  constant right_index : integer; -- size of fraction
  constant overflow_style : fixed_overflow_style_type := FIXED_OVERFLOW_STYLE;
  constant round_style : fixed_round_style_type := FIXED_ROUND_STYLE)
return unresolved_ufixed;
```

Mit der nun vorhandenen resize-Funktion ist es möglich, das Overflow-Handling, 'truncation' oder 'rounding' in einem Schritt durchzuführen. Der Default 'overflow_style' ist 'fixed_saturate', kann aber auch auf 'fixed_wrap' zu setzen um einen Overflow zu ermöglichen. Der Default 'round_style' ist 'fixed_round' um das Resultat zu runden, kann aber auch auf 'fixed_truncate' gesetzt werden.

4.5 Scaling

Das Resultat einer Multiplikation hat die Breite des 'multiplicand' plus 'multiplier', mehrere aufeinanderfolgende Multiplikationen führen somit zu Bit Growth.

4.5.1 Truncation

Eine Möglichkeit: Bits mit tiefer Präzision weglassen. Die normale 'truncation' ist eine floor-Funktion. In VHDL

kann das mit 'shift_right' und 'resize' umgesetzt werden:

```
constant WIDE_NUM : real := -2.5248;
signal sWideNumber : signed(7 downto 0) := to_signed(integer(2.0**5 * REAL_NUM),8); --
signal sTruncated : signed(4 downto 0); -- Q3.2
-- Q3.2 -> 101.01 = -2.75
sTruncated <= resize(shift_right(sWideNumber,3),5);
```

5 Speicher - Custom IP Blocks

5.1 Typen von Speicher in FPGA

In einem FPGA sind normalerweise zwei Arten von Speicher vorhanden:

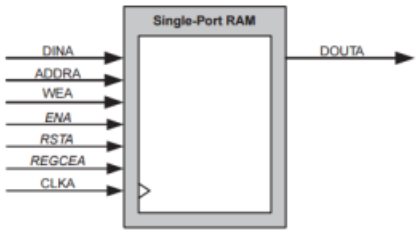
- **Distributed Memory**: Distributed Memory besteht aus vielen LUT Tabellen. Der Vorteil dieser Variante ist, dass jede beliebige Grösse von Speicher realisiert werden kann. Desweitern kann dieser Speicher an jedem Ort in einem FPGA erstellt werden. Ideal für kleine Speichergrössen.
- **Block Memory**: Block Memory sind fest implementierte Speicherzellen (Hard IP Block). Diese bestehen aus SRAM Zellen (zwei kreuzgekoppelte Inverter) und sind über das ganze FPGA hinweg verteilt. Oftmals haben sie auch gerade eine Fehlerkorrektur implementiert (bei Xilinx: Hamming Error Correction Code). Ideal für grössere Speichergrössen. Die Speichergrösse bei den Xilinx Series 7 Block Rams beträgt 32kb (36kb physikalisch vorhanden).

5.1.1 ROM

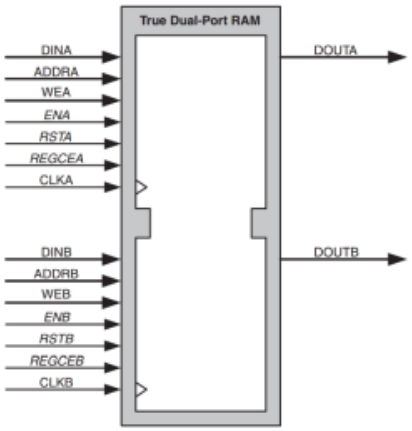
ROM wird benutzt um konstante Daten zu speichern. In einem FPGA wird die Implementierung von ROM mithilfe von RAM gemacht.

5.1.2 Interface Arten

Beide Speicherarten können mit zwei verschiedenen Schnittstellen implementiert werden.



5.1.2.1 Single-Port RAM Single-Port RAM hat einen Daten- und einen Adressbus. Darüber sind sequentielle Lese- und Schreibzugriffe möglich. Wird Distributed Memory verwendet, so ist es auch zusätzlich noch möglich asynchrone Lesezugriffe durchzuführen.



5.1.2.2 Dual-Port RAM Dual-Port RAM erlaubt zwei gleichzeitige Lese- und/oder Schreibzugriffe. Wird über beide Schnittstellen auf die gleiche Speicherzelle zugegriffen, so gibt es eine Arbitrierschaltung, welche diese Situation korrekt abwickelt.

5.2 Beschreibung von Speicher in VHDL

Es existieren vier Richtlinien für das Beschreiben von ROM und RAM in VHDL. Wird Speicher anhand dieser Richtlinien beschrieben, so sollte der Synthesizer den Speicher korrekt implementieren.

1. Die Datengrösse und die Adressengrösse sollen mit generischen Parametern definiert werden.
2. Der Addressenbereich soll mit einer Konstante definiert werden.
3. RAM soll mit einem zweidimensionalen Array beschrieben werden.
4. Der Schreibzugriff muss in einem Prozess beschrieben werden.

5.2.1 Block RAM - Single-Port

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio_type.all;
entity RAM_Test is
  generic (
    ADDR_WIDTH : integer := 8;
    DATA_WIDTH : integer := 8);
```



```
  port (
    clk :    in std_logic;
    addr :   in std_logic_vector(ADDR_WIDTH - 1 downto 0);
    din :    in std_logic_vector(DATA_WIDTH - 1 downto 0);
    dout :   out std_logic_vector(DATA_WIDTH - 1 downto 0);
    we, en : in std_logic);
end entity RAM_Test;

architecture RTL of RAM_Test is
  constant MEM_DEPTH : integer := 2 ** ADDR_WIDTH;
  type ram_type is array (0 to MEM_DEPTH - 1) of std_logic_vector (DATA_WIDTH - 1
    downto 0);
  signal blockram : ram_type;
begin
  ram_write : process (clk)
  begin
    if rising_edge (clk) then
      if en = '1' then -- if RAM is enabled
        if we = '1' then -- if write is enabled
          blockram (to_integer(unsigned(addr))) <= din;
          dout <= din;
        else
          dout <= blockram (to_integer(unsigned(addr)));
        end if;
      end if;
    end if;
  end process ram_write;
end RTL;
```

5.2.2 Distributed RAM - Single-Port

```
-- Gleich wie bei Block Ram Single-Port
architecture RTL of RAM_Test is
  constant MEM_DEPTH : integer := 2 ** ADDR_WIDTH;
  type ram_type is array (0 to MEM_DEPTH - 1) of std_logic_vector (DATA_WIDTH - 1
    downto 0);
  signal distram : ram_type;
begin
  ram_write : process (clk)
  begin
    if rising_edge (clk) then
      if en = '1' then -- if RAM is enabled
        if we = '1' then -- if write is enabled
          distram (to_integer(unsigned(addr))) <= din;
        end if;
      end if;
    end if;
  end process ram_write;

  dout <= distram (to_integer(unsigned(addr)));
end RTL;
```

5.2.3 Block RAM - Dual-Port

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std. memtextio_type.all;
entity RAM_Test is
  generic (
    ADDR_WIDTH : integer := 8;
    DATA_WIDTH : integer := 8);
  port (
```

```
    a_clk : in std_logic;
    a_addr : in std_logic_vector (ADDR_WIDTH - 1 downto 0);
    a_din : in std_logic_vector (DATA_WIDTH - 1 downto 0);
    a_dout : out std_logic_vector (DATA_WIDTH - 1 downto 0);
    a_wr : in std_logic;
    b_clk : in std_logic;
    b_addr : in std_logic_vector (ADDR_WIDTH - 1 downto 0);
    b_din : in std_logic_vector (DATA_WIDTH - 1 downto 0);
    b_dout : out std_logic_vector (DATA_WIDTH - 1 downto 0);
    b_wr : in std_logic);
end entity RAM_Test;

architecture RTL of RAM_Test is
  constant MEM_DEPTH : integer := 2 ** ADDR_WIDTH;
  type ram_type is array (0 to MEM_DEPTH - 1) of std_logic_vector (DATA_WIDTH - 1
    downto 0);
  shared variable blockram : ram_type;
begin
  ramA_write : process (a_clk)
  begin
    if rising_edge (a_clk) then
      if a_wr = '1' then -- if write is enabled
        blockram (to_integer(unsigned(a_addr))) <= a_din;
      end if;
      a_dout <= blockram (to_integer(unsigned(a_addr)));
    end if;
  end process ramA_write;
  ramB_write : process (b_clk)
  begin
    if rising_edge (b_clk) then
      if b_wr = '1' then -- if write is enabled
        blockram (to_integer(unsigned(b_addr))) <= b_din;
      end if;
      b_dout <= blockram (to_integer(unsigned(b_addr)));
    end if;
  end process ramB_write;
end RTL;
```

5.2.4 ROM

Da ROM gleich wie RAM implementiert wird, muss nur das RAM Signal als Konstante definiert werden.

```
constant rom : ram_type;
```

5.2.5 Synthese-Attribute

Mit dem Attribut `ram_style` kann Vivado mitgeteilt werden, ob das RAM als Distributed RAM (`"distributed"`) oder als Block RAM (`"block"`) implementiert werden soll.

```
attribute ram_style : string;
attribute ram_style of blockram : variable is "block";
```

5.2.6 Initialisierung

Eine einfache Art um Speicher zu initialisieren, kann mit Hilfe einer Funktion und einer Datei erreicht werden.

```
architecture RTL of ROM_Test_initialize is
  constant MEM_DEPTH : integer := 2 ** ADDR_WIDTH;
  type ROM_type is array(0 to MEM_DEPTH - 1) of std_logic_vector(DATA_WIDTH - 1
    downto 0);
  impure function init_mem(mif_file_name : in string) return ROM_type is
    file mif_file : text open read_mode is mif_file_name;
    variable mif_line : line;
    variable temp_bv : bit_vector(DATA_WIDTH - 1 downto 0);
    variable temp_mem : ROM_type;
  begin
    for i in ROM_type'range loop
```

```
      readline(mif_file, mif_line);
      read(mif_line, temp_bv);
      temp_mem(i) := to_stdlogicvector(temp_bv);
    end loop;
    return temp_mem;
  end function
  constant ROM : ROM_type := init_mem("mem_init_vhd.mif");
```

Der Speicher kann aber auch direkt im Code initialisiert werden (eher aufwändig und unübersichtlich):

```
constant ROM : rom_type :=("00000000", "00000011", ..., "00100000")
```

5.3 Memory IP Generators Vivado stellt verschiedene IP Generatoren zur Verfügung, über welche direkt Speicher instanziiert werden kann. Diese Generatoren stellen eine einfache Möglichkeit dar, um Speicher nach den eigenen Wünschen zu konfigurieren. Ebenfalls stellen sie präzise Simulationsmodelle zur Verfügung.

5.4 Intellectual Property IP

5.5 Definition IP Core Als IP Core (Intellectual Property Core) wird ein vielfach einsetzbarer, vorgefertigter Funktionsblock eines Chipdesigns bezeichnet. Dieser enthält das geistige Eigentum (intellectual property) des Entwicklers oder Herstellers und wird in der Regel lizenziert bzw. hinzugekauft, um es in ein eigenes Design zu integrieren.

5.6 IP Core Typen

- Hard IP Core: Blackbox in optimierter Layoutform. Sind als fertige Schaltung herstellerseitig unveränderbar in den Chip des FPGAs integriert
- Firm IP Core: Synthetisierte Netzliste, die simuliert und wenn nötig geändert werden kann
- Soft IP Core: RTL Design. Benutzer muss synthetisieren und layouten

5.7 Bezug von IP Core

- Ältere bereits vorhandene in-house IP
- Neu entwickelte in-house IP
- Drittanbieter IP

5.8 Lizenzmodelle

- Free und Open Source
- Kommerzielle Lieferanten (XILINX, CADENCE, ..)
- Aggregator (sammelt und kategorisiert IP Cores und verkauft sie weiter)

5.8.1 Konfiguration von IP Blöcken

- IP Blöcke sind meistens konfigurierbare Module. Jede Instanz eines solchen IP Blocks kann individuell konfiguriert werden.
- Konfiguration von Hard IP Blöcken: Beschränkt auf das Ein- und Ausschalten bestimmter Funktionen, da Hardware nicht modifiziert werden kann.
- Konfiguration von Soft und Firm IP Blöcken: Flexibler, da diese Blöcke erst nach der Konfiguration synthetisiert werden. Häufig können Funktionalität, Implementierungsstrategie, Schnittstellentyp und Dimensionen eingestellt werden. Konfigurationsparameter werden als generische Parameter an das Modul zur Synthese übergeben.

5.8.2 IP Packager

- IP Blöcke bestehen aus vielen Teilen:
  - Quelldateien (RTL, C-Code, Netzlistendateien etc.)
  - Dokumentation
  - Simulationsmodelle
  - Testbenches
  - Beispiele
- Vivado IP Packager stellt aus obigen Teilen ein Komplettpaket zusammen und legt es in ein zentrales Repositiory (IP Katalog).
- IP-XACT: Standard (in XML) für die Verpackung und Dokumentation, welcher von einer Gruppe aus IP-Anbietern unter dem Namen SPIRIT Consortium definiert wurde. Beschreibt nur Schnittstelle und Organisation des Blocks und bietet damit eine Zugangstür für die verschiedenen Werkzeuge, um ihre Informationen zu finden.
- component.xml: Enthält Metadaten, Ports, Schnittstellen, Konfigurationsparameter, Dateien und Dokumentation. Ersetzt nicht HDL oder Software (enthält nur High-Level-Informationen).

5.8.3 Einbinden von IP Blöcken in eigenes Design

1. IP Repository (normalerweise in Projekt oder auf Firmenlaufwerk) dem Projekt bekannt machen.
2. IP Block aus Katalog auswählen (add IP).
3. Anpassungen und Generierung spezifischer Ausgabeprodukte (output products): Anpassung erfolgt im IP Integrator. Die Parameter müssen an den RTL-Code des IP Blockes übergeben werden und der Code muss in das Design aufgenommen werden. Bei Generierung der Ausgabeprodukte erzeugt der IP Integrator die kundenspezifischen Designinformationen.
4. IP verwenden: Der Baustein kann nun verwendet werden, indem er mit dem IP-Integrator im Blockdesign

platziert oder in einem herkömmlichen RTL-Design instanziiert wird.
IP Blöcke können verschieden eingebunden werden:

- Via IP Integrator: Vivado führt die folgenden Schritte aus: Instanziierung (Block einfügen in Design), Erzeugung von System-Wrapper (strukturelle VHDL-Top-Level-Beschreibung) und Generierung der Ausgabeprodukte.
- Via Instanzierungs-Template im RTL Flow: Für VHDL und Verilog werden Instanzierungs-Templates zur Verfügung gestellt. Der IP Block muss in der Design-Datei, welche eine Position höher in der Design-Hierarchie ist, instanziiert werden.

```
-- Begin Cut here for COMPONENT Declaration
component blk_mem_gen_0
  port (
    clka : in std_logic;
    ena : in std_logic;
    ...
    dinb : in std_logic_vector(15 downto 0);
    doutb : out std_logic_vector(15 downto 0)
  );
end component;
-- End COMPONENT Declaration ------------
-- Begin Cut here for INSTANTIATION Template
your_instance_name : blk_mem_gen_0
  port map (
    clka => clka,
    ena => ena,
    wea => wea,
    ...
    dinb => dinb
  );
```

5.8.4 IP Life Cycle

- Vorproduktion (pre-production): IP Core, der öffentlich verwendbar ist, aber noch keine Qualifikationen für den Einsatz in der Produktion aufweist.
- Produktion (production): IP Core, der für die allgemeine öffentliche Freigabe zur Verfügung gestellt wird und verifiziert wurde.
- Eingestellt (discontinued): Ankündigung von XILINX, dass IP Core bald entfernt wird.
- Ersetzt (superseded): IP Core wurde durch eine neuere Version ersetzt.
- Entfernt (removed): IP Core wird nicht mehr länger vertrieben.

Change level	**User action**	Examples of Changes
Revision	No need to react	Add new device support. Cosmetic GUI changes. Move device support from Pre-Production to Production Extend Parameter range Bug fix for unusable configurations (no working configuration changed)
Minor	May need to react	Reduction in parameter range Remove an optional port Add a memory-mapped register whose use is optional Increased resource usage
Major	Will need to react	Add a non-optional, non-static input port Rename a non-optional port (including case change if Verilog) Change a non-optional port's size Remove a non-optional port Change the interface standard Change or remove a memory-mapped register Behavioral change for all configurations

6 Communication Interfaces

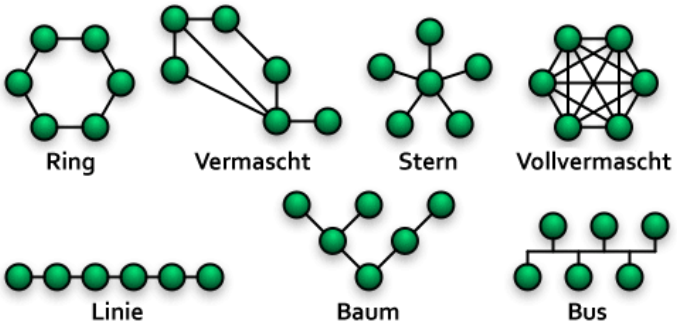
6.1 Begriffe Verschiedene Begriffe existieren im Zusammenhang mit Schnittstellen:

- **Synchron / Asynchron**: Synchrone Schnittstellen haben ein Taktsignal, welches signalisiert, wann Daten gültig sind. Asynchrone Schnittstellen brauchen dagegen kein Taktsignal. Synchronisation geschieht über

einen Handshakingprozess.
- **Seriell / Parallel**: Daten werden seriell oder parallel (mit mehreren Signalleitungen gleichzeitig) übertragen.
- **On-Chip / Off-Chip Communication**: Kommunikation im Chip selbst wird oftmals parallel gelöst. Eine Kommunikation zwischen zwei verschiedenen Chips wird oft seriell implementiert.

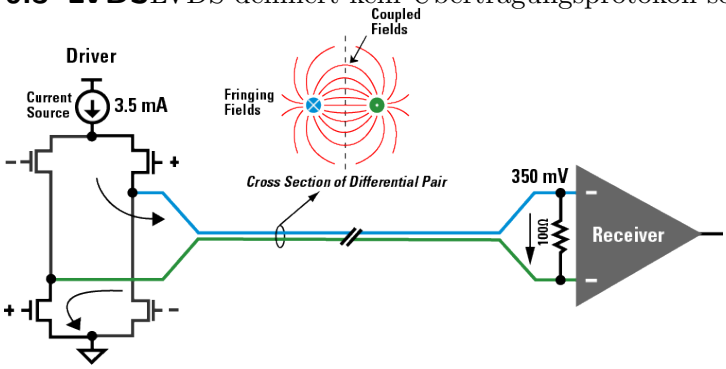
6.2 Netzwerk Arten

Verschiedene Netzwerk Arten existieren in der Praxis:



6.3 LVDS

LVDS definiert kein Übertragungsprotokoll sondern eine Übertragungsart.



6.3.1 Prinzip

Die Signale werden anstelle mit einer Spannungsänderung mit einer Stromänderung übertragen. Das Signal schwingt somit nur um eine Spannung von $\pm 350\text{mV}$.

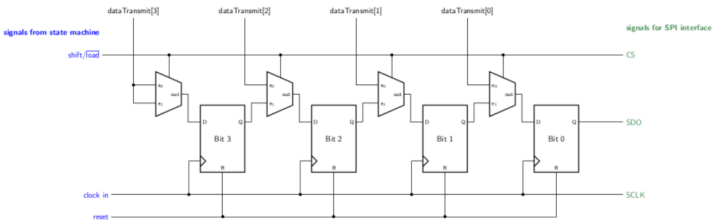
6.3.2 Implementierung in VHDL

In VHDL kann LVDS nicht direkt synthetisiert werden. Der FPGA hat jedoch dennoch Pads, welche LVDS unterstützen. Diese müssen jedoch mithilfe von Constraints zugewiesen werden.

```
set_property IOSTANDARD LVDS [get_ports
    portname]
```

6.4 VHDL Components

Es gibt generische VHDL-Komponenten die für Kommunikationsinterfaces verwendet werden können. Sie alle beschreiben Teile des data link layers.



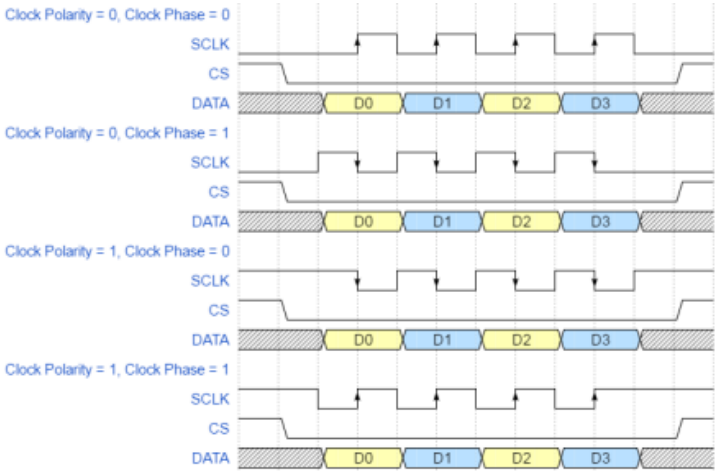
7 Serial Communication Interfaces

7.1 SPI (Serial Peripheral Interface)

In einem SPI Netzwerk kann jeweils nur ein Master vorhanden sein. Jedoch ist es möglich, dass mehrere Slaves am selben Bus angeschlossen sind. Die folgenden Signale werden für die SPI Schnittstelle benötigt:
- **SCLK**: Taktsignal
- **MOSI / SDO**: Datensignal vom Master zum Slave.
- **MISO / SDI**: Datensignal vom Slave zum Master.
- **CS**: Signalisiert dem Slave eine aktive Kommunikation. Sind mehrere Slaves am selben Bus angeschlossen, so existiert oft für jeden Slave ein eigenes CS-Signal.

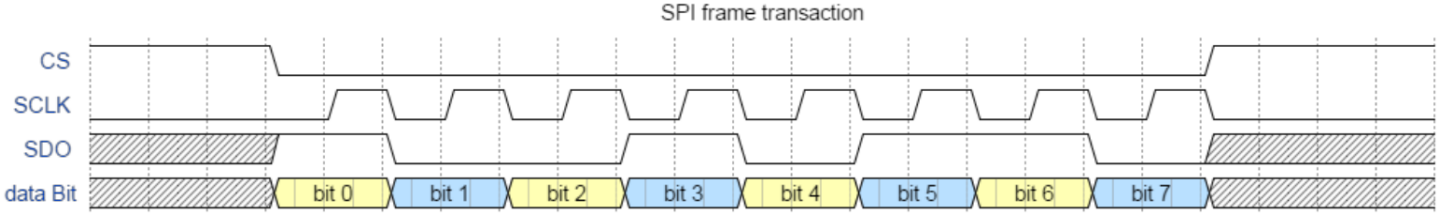
7.1.1 Taktpolarität und Taktphase

Es gibt insgesamt vier SPI-Modi. Diese unterscheiden sich in der Taktpolarität und Taktphase.



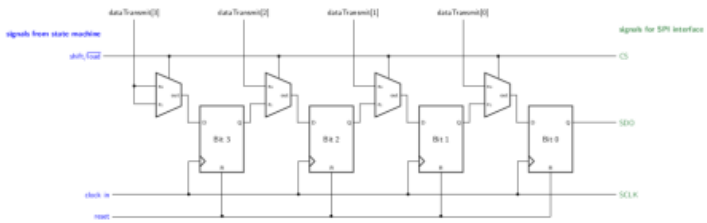
7.1.2 Timing

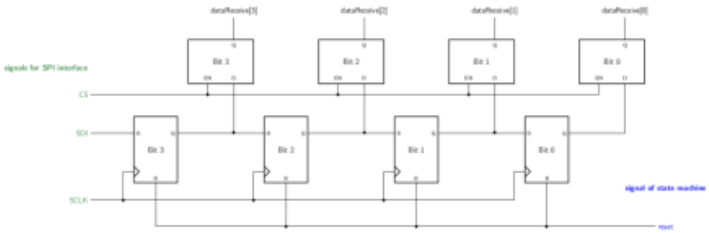
Eine mögliche SPI Transaktion ist untenstehend zu sehen:



7.1.3 Hardwaremässige Implementierung

Die Hardware für eine SPI Schnittstelle kann mittels zwei Schieberegistern (eines für das Senden, sowie eines für das Empfangen) realisiert werden.





7.1.4 Implementierung in VHDL

```
library ieee;
use ieee.std_logic_1164.all;
entity spiMaster is
  generic(
    DATAWIDTH     : integer
  );
  port(
    clkIn         : in std_logic;
    reset         : in std_logic;
    -- user interface
    start         : in  std_logic;
    busy          : out std_logic;
    dataTransmit  : in  std_logic_vector(DATAWIDTH-1 downto 0);
    dataReceive   : out std_logic_vector(DATAWIDTH-1 downto 0);
    -- SPI interface
    CS            : out std_logic;
    SCLK          : out std_logic;
    SDI           : in  std_logic;
    SDO           : out std_logic
  );
end spiMaster;
architecture behavioral of spiMaster is
  type stateType is (idle, sclk0, sclk1);
  signal statePresent, stateNext : stateType;
  signal counterPresent, counterNext : integer range 0 to DATAWIDTH-1;
```



```
  signal shiftRegTransmitNext : std_logic_vector(DATAWIDTH-1 downto 0);
  signal shiftRegTransmitPresent : std_logic_vector(DATAWIDTH-1 downto 0);
  signal shiftRegReceiveNext : std_logic_vector(DATAWIDTH-1 downto 0);
  signal shiftRegReceivePresent : std_logic_vector(DATAWIDTH-1 downto 0);
begin
  regLogic : process(clkIn, reset)
  begin
    if reset = '1' then
      statePresent <= idle;
      counterPresent <= 0;
      shiftRegTransmitPresent <= (others => '0');
      shiftRegReceivePresent <= (others => '0');
    elsif rising_edge(clkIn) then
      statePresent <= stateNext;
      counterPresent <= counterNext;
      shiftRegTransmitPresent <= shiftRegTransmitNext;
      shiftRegReceivePresent <= shiftRegReceiveNext;
    end if;
  end process regLogic;
  nextStateLogic : process (counterPresent, start, statePresent)
  begin
    stateNext <= idle;
    counterNext <= 0;
    case statePresent is
      when idle =>
        if start = '1' then
          stateNext <= sclk0;
        else
          stateNext <= idle;
        end if;
      when sclk0 =>
        counterNext <= counterPresent;
        stateNext <= sclk1;
      when sclk1 =>
        if counterPresent >= DATAWIDTH-1 then
          stateNext <= idle;
        else
          stateNext <= sclk0;
          counterNext <= counterPresent+1;
        end if;
    end case;
  end process nextStateLogic;
  transmitShiftRegister : process (dataTransmit, shiftRegTransmitPresent,
    shiftRegTransmitPresent(7 downto 1), statePresent)
  begin
    case statePresent is
      when idle =>
        shiftRegTransmitNext <= DataTransmit;
      when sclk0 =>
        shiftRegTransmitNext <= shiftRegTransmitPresent;
      when sclk1 =>
        shiftRegTransmitNext <= '0' & shiftRegTransmitPresent(7 downto 1);
    end case;
  end process;
  SDO <= shiftRegTransmitNext(0);
  receiveShiftRegister : process (SDI, shiftRegReceivePresent,
    shiftRegReceivePresent(6 downto 0), statePresent)
  begin
    case statePresent is
      when idle =>
        DataReceive <= shiftRegReceivePresent;
```

```
        shiftRegReceiveNext <= shiftRegReceivePresent;
      when sclk0 =>
        shiftRegReceiveNext <= shiftRegReceivePresent(6 downto 0) & SDI;
      when sclk1 =>
        shiftRegReceiveNext <= shiftRegReceivePresent;
    end case;
  end process receiveShiftRegister;
  CS   <= '1' when statePresent = idle else '0';
  SCLK <= '1' when statePresent = sclk1 else '0';
  busy <= '0' when statePresent = idle else '1';
end behavioral;
```

7.2 I²C (Inter-Integrated Circuit)

Das I²C Interface ist ein bidirektionales Bussystem. Es können beliebig viele Master und Slaves angehängt werden. Die folgenden Signale werden für die I²C Schnittstelle benötigt:

- **SCL**: Taktsignal
- **SDA**: Bidirektionales Datensignal

7.2.1 Übertragungsgeschwindigkeiten

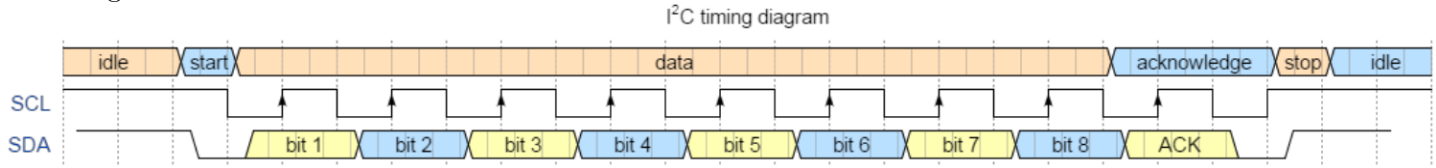
Die folgenden Geschwindigkeiten sind im I²C Standard definiert:

- standard mode: 100 kbit/s
- fast-mode: 400 kbit/s
- fast-mode-plus: 1 Mbit/s
- high-speed mode: 3.4 Mbit/s

7.2.2 Timing

Die folgenden Zustände auf dem I²C Bus sind möglich:

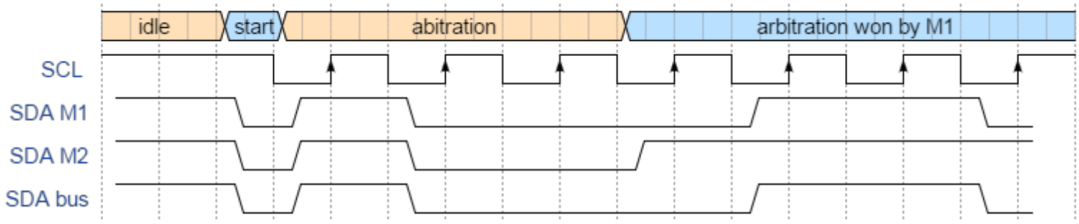
Eine mögliche I²C Transaktion ist untenstehend zu sehen:



Nach dem Senden des Acknowledge-Bits, kann jeweils ein weiteres Byte gesendet werden, ohne die Transaktion durch Senden eines Stopbits zu unterbrechen.
Sollte ein Slave mehr Zeit brauchen um Daten zu verarbeiten, so kann er das SCL Signal auf Low ziehen, bis er wieder bereit ist.

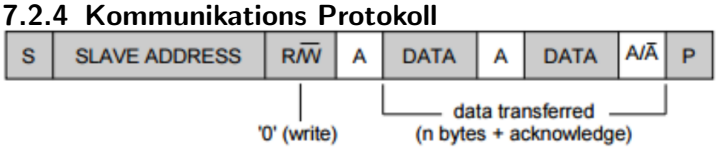
7.2.3 Arbitration

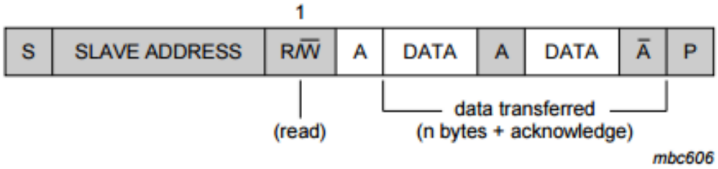
Da mehrere Master und Slaves gleichzeitig senden könnten, ist eine Arbitrierung notwendig. Diese ist untenstehend dargestellt.



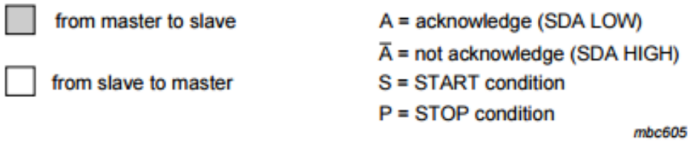
Der letzte Teilnehmer, welcher die SDA Leitung auf Low ziehen kann, gewinnt die Kommunikation.

7.2.4 Kommunikations Protokoll





mbc606



mbc605

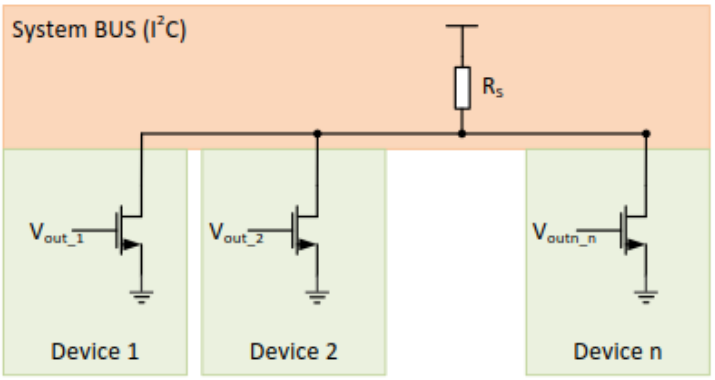
7.2.4.1 Schreiben vom Master zum Slave

7.2.4.2 Lesen vom Slave

7.2.5 Hardwaremässige Implementierung

Damit keine Kurzschlüsse entstehen, wenn zwei Master gleichzeitig senden wollen, werden die beiden Signale

über externe Pull-Up-Widerstände auf einen High Pegel gezogen.



7.2.6 Implementierung in VHDL

Da die beiden I²C-Signale bidirektional sind, muss in VHDL ein IO-Buffer für die beiden Signale instantiiert werden. Der nachfolgende Code instantiiert zwei solche Buffer.

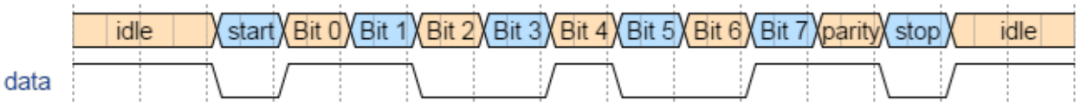
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity i2cMaster is
  port(
    SDA : inout std_logic;
    SCL : inout std_logic
  );
end i2cMaster;
architecture behavioral of i2cMaster is
  signal SDAin  : std_logic;
  signal SDAout : std_logic;
  signal SCLena : std_logic;
  signal SCLclk : std_logic;
begin
  SDA <= '0' when SDAout = '0' else 'Z';
  SDAin <= SDA;
  SCL <= '0' when (SCLena = '1' and SCLclk = '0') else 'Z';
end behavioral;
```

7.3 UART (Universal Asynchronous Receiver Transmitter)

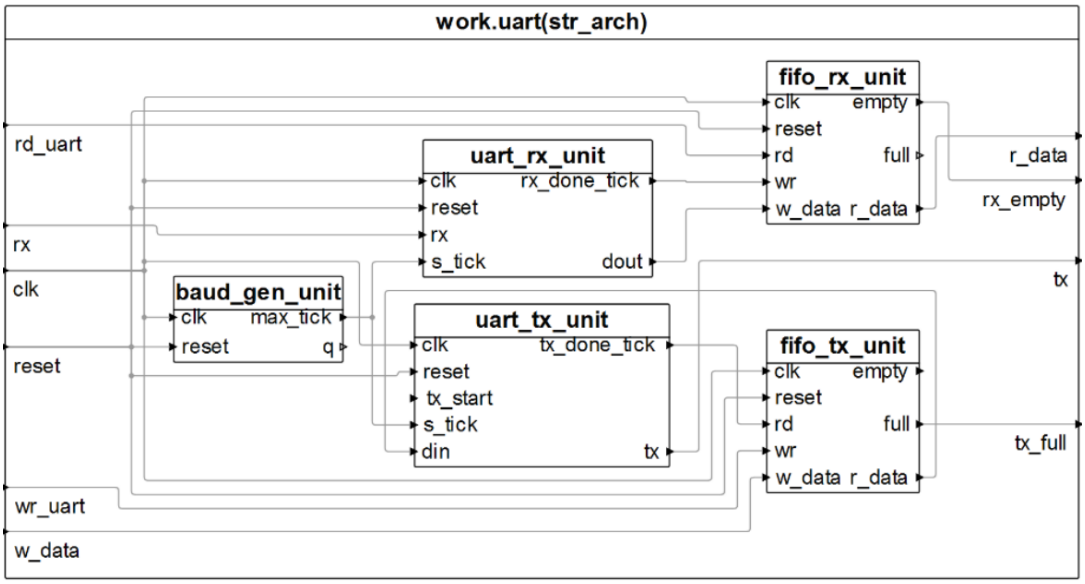
UART ist eine Punkt-zu-Punkt Schnittstelle. Es wird pro Richtung ein Datensignal benötigt.

7.3.1 Timing

Eine UART Transaktion startet immer mit einem Startbit. Anschliessend wird ein Datenbyte gesendet, eine Parität, sowie ein Stoppbit (oder mehrere Stopbits). Da kein Takt mitgesendet wird, muss im Sender und Empfänger die Geschwindigkeit bekannt sein.



7.3.2 Hardwaremässige Implementierung



7.3.3 VHDL Implementation

```
entity uart is
  generic(
    DBIT :      integer := 8;
    SB_TICK :   integer := 16;
    DVSR :      integer := 163;
    DVSR_BIT :  integer := 8;
    FIFI_W :    integer := 2;
  );
  port(
    clk, reset :          in std_logic;
    rd_uart, wr_uart :    in std_logic;
    rx :                  in std_logic;
    w_data :              in std_logic_vector(7 downto 0);
    tx_full, rx_empty :   out std_logic;
    r_data :              out std_logic_vector(7 downto 0);
    tx :                  out std_logic;
  );
end uart;
architecture arch of mod_m_counter is
  signal r_reg  : unsigned(N-1 downto 0);
  signal r_next : unsigned(N-1 downto 0);
begin
  process(clk, reset)
  begin
    if (reset = '1') then
      r_reg <= (others => '0');
    elsif rising_edge(clk) then
      r_reg <= r_next;
    end if;
  end process;
  r_next <= (others => '0') when r_reg = (M-1) else r_reg + 1;
  q <= std_logic_vector(r_reg);
  max_tick <= '1' when r_reg = (M-1) else '0';
end arch;
architecture arch of uart_tx is
  type state_type is (idle, start, data, stop);
  signal state_reg, state_next : state_type;
  signal s_reg, s_next : unsigned(3 downto 0);
  signal n_reg, n_next : unsigned(2 downto 0);
  signal b_reg, b_next : std_logic_vector(7 downto 0);
```

```
begin
  process(clk, reset)
  begin
    if reset = '1' then
      state_reg <= idle;
      s_reg <= (others => '0');
      n_reg <= (others => '0');
      b_reg <= (others => '0');
    elsif rising_edge(clk) then
      state_reg <= state_next;
      s_reg <= s_next;
      n_reg <= n_next;
      b_reg <= b_next;
    end if;
  end process;
  process(state_reg, s_reg, n_reg, b_reg, s_tick, rx)
  begin
    state_next <= state_reg;
    s_next <= s_reg;
    n_next <= n_reg;
    b_next <= b_reg;
    rx_done_tick <= '0';
    case state_reg is
      when idle =>
        if rx = '0' then
          state_next <= start;
          s_next <= (others => '0');
        end if;
      when start =>
        if (s_tick = '1') then
          if s_reg = 7 then
            state_next <= data;
            s_next <= (others => '0');
            n_next <= (others => '0');
          else
            s_next <= s_reg + 1;
          end if;
        end if;
      when data =>
        if (s_tick = '1') then
          if s_reg = 15 then
            s_next <= (others => '0');
            b_next <= rx & b_reg(7 downto 1);
            if n_reg = (DBIT-1) then
              state_next <= stop;
            else
              n_next <= n_reg + 1;
            end if;
          else
            s_next <= s_reg + 1;
          end if;
        end if;
      when stop =>
        if (s_tick = '1') then
          if s_reg = (SB_TICK-1) then
            state_next <= idle;
            rx_done_tick <= '1';
          else
            s_next <= s_reg + 1;
          end if;
        end if;
```

```
      end case;
    end process;
    dout <= b_reg
end architecture arch,
```

8 Parallel Communication Interfaces

8.1 Vorteile paralleler Kommunikation

- Parallele Kommunikation hat theoretischen Vorteil eines Durchsatzes, der N-mal grösser für eine N-Bit breite Kommunikation ist (gegenüber seriell).
- Parallele Kommunikation hat zusätzlichen Vorteil, dass keine Serialisierung / De-Serialisierung an Quelle und Senke benötigt wird → reduziert Hardwarekosten und Komplexität

8.2 Nachteile paralleler Kommunikation

- In der Praxis stimmt die Aussage des Durchsatzes nicht. Clock und Data Skew bewirken, dass die Kommunikationsgeschwindigkeit der Geschwindigkeit des langsamsten Bits entspricht.
- Crosstalk zwischen den Leitungen ist ein weiteres Problem der parallelen Kommunikation. Parallele Datenleitungen beeinflussen sich gegenseitig und dadurch kann sich die Bitfehlerrate leicht erhöhen.
- Mit zunehmender Systemgröße von SoC-Komponenten hat die Anzahl der IOs ein Niveau erreicht, in dem die parallele Kommunikation mit der grossen Anzahl von benötigten IOs für die Off-Chip-Kommunikation mehr und mehr unwirtschaftlich wird → IOs verbrauchen Chipfläche und erzeugen somit Kosten.

8.3 AXI - Advanced eXtensible Interface

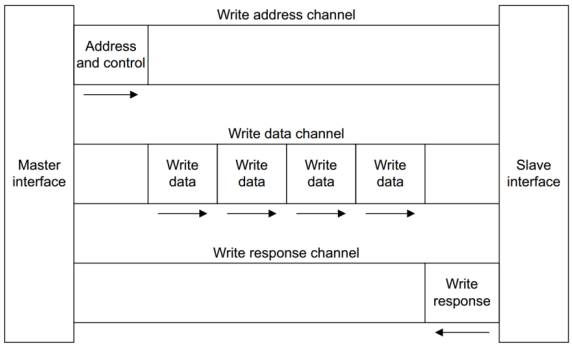
Das AXI-Protokoll ist für eine hohe Bandbreite sowie eine kleine Latenzzeit ausgelegt. Es werden getrennte Lese- und Schreibbusse unterstützt. Weiter sind getrennte Busse für die Adressen sowie die Steuersignale vorhanden. Zusätzlich können auch sogenannte Burst-Transaktionen durchgeführt werden.

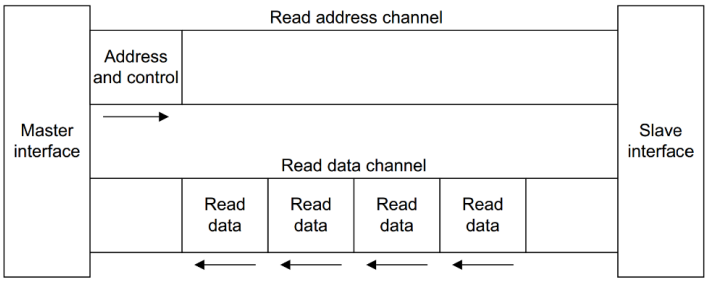
8.3.1 Bustopologie und Modes

Die AXI-Schnittstelle ist eine Multiple-Master-Multiple-Slave-Schnittstelle. Diese Topologie erfordert eine physikalische Verbindungsstruktur, die jeden Master mit jedem Slave verbindet. Die Topologie erlaubt es, dass zwei Master gleichzeitig Daten übertragen. Um Kollisionen zu vermeiden, ist ein Arbiter erforderlich, der entscheidet, welcher Master Zugriff auf einen Slave erhält. Es gibt verschiedene Verfahren, wie z. B. round-robin, preemptive priority, non-preemptive priority... Der Lese- und Schreibkanal des Adressbusses kann geteilt werden, um Platz zu sparen. Dasselbe gilt auch für den Datenbus. Im AXI-Protokoll, müssen zwei Teile unterschieden werden:

- AXI interface ist eine 1-zu-1 Verbindung mit 5 unabhängigen Kanälen für Schreib-/Lesetransaktionen. Das AXI interface verbindet immer einen Master mit einer Slave-Node.
- AXI interconnect verbindet N Master mit N Slave Nodes. Es ist ein Logikblock, bestehend aus Arbitern, Multiplexern, Pipeline-Registern, vielen Verbindungen, einigen FIFOs und anderer Logik-Primitiven. AXI interconnect arbeitet als Switchbox um Signale von einer Master zu einer Slave-Node zu routen.

8.3.2 Kanäle

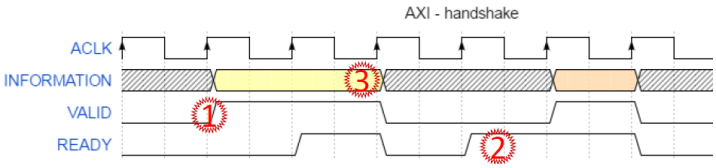




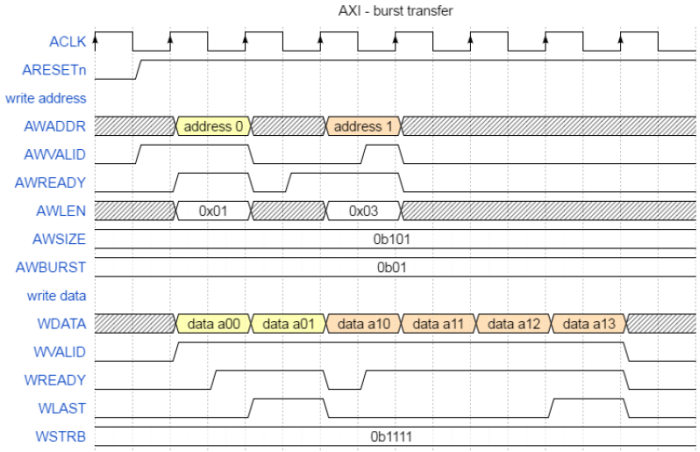
- AW (write address channel) → Master zu Slave
- W (write data channel) → Master zu Slave
- WR (B) (write response channel) → Slave zu Master
- AR (read address channel) → Master zu Slave
- R (read data channel) → Slave zu Master

8.3.3 Handshaking

Für jeden Übertragungskanal existiert ein Handshaking. Ein Vorteil dieser Funktionsweise besteht darin, dass der Empfänger sowie der Sender die Geschwindigkeit der Datenübertragung steuern können.



1. Der Datensender setzt das Valid Signal, um anzuzeigen, dass Informationen verfügbar sind.
2. Der Datenempfänger setzt das Ready Signal, um anzuzeigen, dass die Information akzeptiert werden kann.
3. Eine Übertragung von Daten tritt auf, wenn sowohl das Valid als auch das Ready Signal an einer Taktflanke high sind.



8.3.3.1 Burst Mode

Anstatt ein einziges Wort oder Byte in einer Transaktion zu senden, wird im Burst-Modus ein gesamter Block von Daten gesendet. Dies ist sehr nützlich bei der Übertragung grosser Datenmengen.

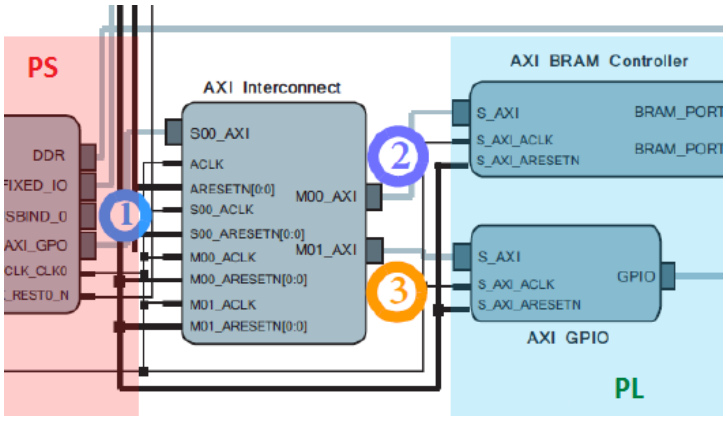
8.3.4 Varianten

- AXI 4: Maximale Performance. Burst-Modus bis zu 256 Zyklen pro Adresse. Für Hochleistungsdatenaustausch.
- AXI-Lite: Vereinfachte Version von AXI 4. Für speicherabgebildete Einzeltransaktionen. Kein Burst. Für kleinen Datendurchsatz wie Austausch von Steuerungseinstellungen.
- AXI-Stream: Für High-Speed-Streaming. Keine Adressphase (nicht speicherzugeordnet). Unbegrenzte Datenburstgrösse. Daten vom Master zum Slave.

8.3.5 AXI Interconnect Block

Für den Anschluss des AXI-Bussystems werden viele Interconnect Blöcke verwendet.

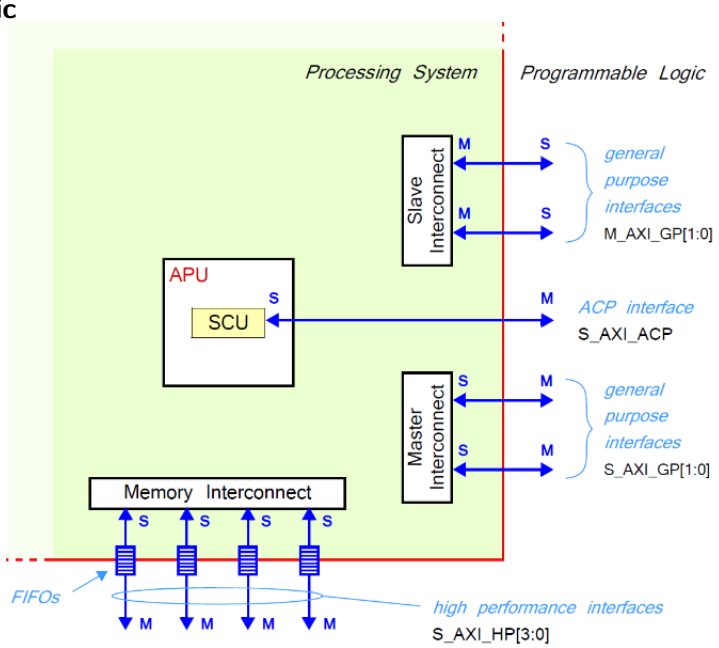
- Agiert als Slave vom PS
- Dient als Master für die Signalweiterleitung vom PS zu verschiedenen AXI-IP-Instanzen (Slaves).
- Unterstützt auch eine Protokollkonvertierung (z.b. Verbindung von AXI 3 und AXI 4)



- SASD (Shared address bus and shared data buses) → Für weniger komplexe Systeme
- SAMD (Shared address bus and multiple data buses) → Durchsatz des Adressbusses ist kleiner als derjenige des Datenbusses, aufgrund der Burst-Fähigkeit des Datenkanals.
- MAMD (Multiple address bus and multiple data buses) → Für komplexe Systeme

8.3.6 Zynq: Processing System ↔ Programmable Logic

Datenübertragung zwischen PS und PL erfolgt via AXI. Jede Schnittstelle realisiert einen kompletten AXI Bus. Insgesamt sind 9 AXI Schnittstellen vorhanden.



9 Combining FPGA and Processor

Oftmals erfolgt die Kommunikation zwischen PS und PL über AXI. Die meisten IPs aus dem IP-Katalog von Xilinx haben diese Schnittstelle implementiert.

9.1 SDK

Mit dem SDK von Xilinx kann der ARM Prozessor auf dem Zynq programmiert werden. Wird das SDK gestartet, so öffnet sich automatisch ein neues generiertes Beispielprojekt.

9.1.1 Benutzen der AXI Schnittstelle

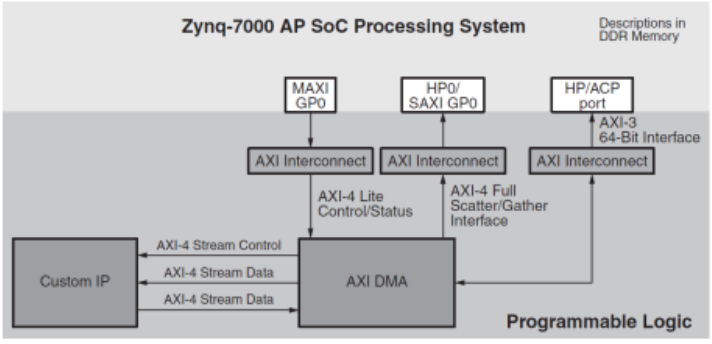
Im Beispielprojekt wird eine Datei *xparameters.h* erstellt. Diese enthält Definitionen über die vorhandenen AXI-Schnittstellen, wie zum Beispiel der Adressbereich. In der Datei *xil_io.h* sind Funktionen enthalten um Register über AXI zu lesen und zu schreiben. Das SDK erstellt zudem in der Datei *AXILite.h* oder *AXI.h* spezialisierte Funktionen, welche passend auf die über AXI angeschlossenen Komponenten zugeschnitten sind (es wird deshalb empfohlen die Funktionen aus diesen Dateien zu benutzen).

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xil_types.h" // basic types for Xilinx software IP
#include "xil_io.h"    // interface for the general IO component
#include "AXILite.h"   // board support package library for user component
int main()
{
  static u32 ArrayIn[4] = {0,0,0,0};
  static u32 ArrayOut[4] = {0x10101010, 0x20202020, 0x30303030, 0x40404040};
  init_platform();
  // write data to the AXILite component registers (xil_io.h)
  Xil_Out32(XPAR_AXILITE_0_S00_AXI_BASEADDR + 0, ArrayOut[0]);
  Xil_Out32(XPAR_AXILITE_0_S00_AXI_BASEADDR + 4, ArrayOut[1]);
  Xil_Out32(XPAR_AXILITE_0_S00_AXI_BASEADDR + 8, ArrayOut[2]);
  Xil_Out32(XPAR_AXILITE_0_S00_AXI_BASEADDR + 12, ArrayOut[3]);
  // read data from the AXILite component registers (xil_io.h)
  ArrayIn[0] = Xil_In32(XPAR_AXILITE_0_S00_AXI_BASEADDR + 0);
  ArrayIn[1] = Xil_In32(XPAR_AXILITE_0_S00_AXI_BASEADDR + 4);
  ArrayIn[2] = Xil_In32(XPAR_AXILITE_0_S00_AXI_BASEADDR + 8);
  ArrayIn[3] = Xil_In32(XPAR_AXILITE_0_S00_AXI_BASEADDR + 12);
  // write data to the AXILite component registers (AXILite.h)
  AXILITE_mWriteReg(XPAR_AXILITE_0_S00_AXI_BASEADDR, 0, ArrayOut[0]);
  AXILITE_mWriteReg(XPAR_AXILITE_0_S00_AXI_BASEADDR, 4, ArrayOut[1]);
  AXILITE_mWriteReg(XPAR_AXILITE_0_S00_AXI_BASEADDR, 8, ArrayOut[2]);
  AXILITE_mWriteReg(XPAR_AXILITE_0_S00_AXI_BASEADDR, 12, ArrayOut[3]);
  // read data to the AXILite component registers (AXILite.h)
  ArrayIn[0] = AXILITE_mReadReg(XPAR_AXILITE_0_S00_AXI_BASEADDR, 0);
  ArrayIn[1] = AXILITE_mReadReg(XPAR_AXILITE_0_S00_AXI_BASEADDR, 4);
  ArrayIn[2] = AXILITE_mReadReg(XPAR_AXILITE_0_S00_AXI_BASEADDR, 8);
  ArrayIn[3] = AXILITE_mReadReg(XPAR_AXILITE_0_S00_AXI_BASEADDR, 12);
  cleanup_platform();
  return 0 ;
}
```

9.1.2 AXI Direct Memory Access (DMA)

Im IP Katalog von Xilinx gibt es einen AXI DMA IP Core. Dieser stellt zwei AXI4-Stream-Schnittstellen für eine hohe Bandbreite zur Verfügung. Eine Schnittstelle dient der Datenübertragung vom Master zum Slave, die andere für die gegengesetzte Richtung.

9.1.2.1 Verbindung mit ARM Um die beste Performanz zu erreichen, sollte der IP Block wie nachfolgend dargestellt mit dem ARM verbunden werden:



9.1.2.2 Modi Die folgenden Modi stellt der DMA IP Block zur Verfügung:

- **Register Direct Mode**: Dieser Modus wird benutzt, wenn eine hohe Geschwindigkeit zu einem Block im PL gefordert ist. Alle DMA Transfers werden gestartet, indem dem DMA-Blcok eine Ziel- oder Startadresse sowie eine Länge mitgeteilt wird. Sobald der Transfer abgeschlossen wurde, wird ein Interrupt generiert.
- **Scatter/Gatter Mode**: In diesem Mode werden Daten direkt in einen Speicher geschrieben, ohne dass vorher spezifisch eine Anforderung getätigt werden muss (benötigt mehr FPGA Ressourcen).

9.1.3 PL330 DMA

Der Zynq stellt im PS einen eigenen DMA-Controller zur Verfügung. Bis zu acht Streams können von diesem Block gemanaged werden. Das SDK stellt in der Datei *xdmaps.h* verschiedene Funktionen zur Ansteuerung des DMA-Controllers zur Verfügung.

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xil_types.h" // basic types for Xilinx software IP
#include "AXILite.h"   // board support package library for user component
#include "xdmaps.h"    // board support package library for PS DMA control
int main()
{
  static u32 ArrayIn[4] = {0,0,0,0};
  static u32 ArrayOut[4] = {0x10101010, 0x20202020, 0x30303030, 0x40404040};
  XDmaPs_Config *DmaCfg
  XDmaPs_Cmd DmaCmd;
  XDmaPs DmaInst;
  init_platform();
  // initialize PS DMA control registers
  memset(&DmaCmd, 0, sizeof(XDmaPs_Cmd));
  DmaCmd.ChanCtrl.SrcBurstSize = 4;
  DmaCmd.ChanCtrl.SrcBurstLen = 4;
  DmaCmd.ChanCtrl.SrcInc = 1;
  DmaCmd.ChanCtrl.DstBurstSize = 4;
  DmaCmd.ChanCtrl.DstBurstLen = 4;
  DmaCmd.ChanCtrl.DstInc = 1;
  DmaCmd.BD.SrcAddr = (u32) &ArrayOut[0]; // source address
  DmaCmd.BD.DstAddr = (u32) XPAR_AXILITE_0_S00_AXI_BASEADDR; // destination address
  DmaCmd.BD.Length = 4 * sizeof(int);
  DmaCfg = XDmaPs_LookupConfig(XPAR_XDMAPS_1_DEVICE_ID);
  XDmaPs_CfgInitialize(&DmaInst, DmaCfg, DmaCfg->BaseAddress);
  // start data transfer with PS DMA
  XDmaPs_Start(&DmaInst, 0, &DmaCmd, 0);
  while(DmaInst.IsReady == 0); // wait on transfer complete
  // read data from the AXILite component registers
  ArrayIn[0] = AXILITE_mReadReg(XPAR_AXILITE_0_S00_AXI_BASEADDR, 0);
  ArrayIn[1] = AXILITE_mReadReg(XPAR_AXILITE_0_S00_AXI_BASEADDR, 4);
  ArrayIn[2] = AXILITE_mReadReg(XPAR_AXILITE_0_S00_AXI_BASEADDR, 8);
  ArrayIn[3] = AXILITE_mReadReg(XPAR_AXILITE_0_S00_AXI_BASEADDR, 12);
  cleanup_platform();
  return 0 ;
}
```

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xil_types.h" // basic types for Xilinx software IP
#include "AXILite.h"   // board support package library for user component
#include "xdmaps.h"    // board support package library for PS DMA control
int main()
{
  static u32 ArrayIn[4] = {0,0,0,0};
  static u32 ArrayOut[4] = {0x10101010, 0x20202020, 0x30303030, 0x40404040};
  XDmaPs_Config *DmaCfg
  XDmaPs_Cmd DmaCmd;
  XDmaPs DmaInst;
  init_platform();
```

```
    // initialize PS DMA control registers
    memset(&DmaCmd, 0, sizeof(XDmaPs_Cmd));
    DmaCmd.ChanCtrl.SrcBurstSize = 4;
    DmaCmd.ChanCtrl.SrcBurstLen = 4;
    DmaCmd.ChanCtrl.SrcInc = 1;
    DmaCmd.ChanCtrl.DstBurstSize = 4;
    DmaCmd.ChanCtrl.DstBurstLen = 4;
    DmaCmd.ChanCtrl.DstInc = 1;
    DmaCmd.BD.SrcAddr = (u32) XPAR_AXILITE_0_S00_AXI_BASEADDR; // source address
    DmaCmd.BD.DstAddr = (u32) &ArrayIn[0]; // destination address
    DmaCmd.BD.Length = 4 * sizeof(int);
    DmaCfg = XDmaPs_LookupConfig(XPAR_XDMAPS_1_DEVICE_ID);
    XDmaPs_CfgInitialize(&DmaInst, DmaCfg, DmaCfg->BaseAddress);
    // write data to the AXILite component registers
    AXILITE_mWriteReg(XPAR_AXILITE_0_S00_AXI_BASEADDR, 0, ArrayOut[0]);
    AXILITE_mWriteReg(XPAR_AXILITE_0_S00_AXI_BASEADDR, 4, ArrayOut[1]);
    AXILITE_mWriteReg(XPAR_AXILITE_0_S00_AXI_BASEADDR, 8, ArrayOut[2]);
    AXILITE_mWriteReg(XPAR_AXILITE_0_S00_AXI_BASEADDR, 12, ArrayOut[3]);
    // start data transfer with PS DMA
    XDmaPs_Start(&DmaInst, 0, &DmaCmd, 0);
    whiley(DmaInst.IsReady == 0); // wait on transfer complete
    cleanup_platform();
    return 0 ;
}
```

10 Verifikation

Oft wird die Verifikation als letzter Schritt nach dem Design bezeichnet. Doch die Verifikation beginnt viel früher und sie ist mehr als nur der Beweis der funktionalen Korrektheit. Sie beginnt auf Anforderungsebene und ist ein konstanter Prozess, der parallel zum Designprozess läuft.

- Verifikation bei Spezifikation: Ist das was ich möchte wirklich das was ich benötige?
- Verifikation beim Design: Habe ich tatsächlich das entwickelt, was gefordert wurde?
- Verifikation beim Testen: Kann ich defekte von fehlerhaften Schaltungen unterscheiden?

Verschiedener Quellen zufolge wird deutlich, dass der Gesamtaufwand für die Verifikation mehr als 50% des gesamten Entwicklungsaufwands beträgt - Tendenz wachsend!

10.1 Dynamische Verifikationsmethoden

Überprüft das Eingangs-/Ausgangsverhalten einer Schaltung. Eine Schaltung mit k Zuständen s und n Eingängen x wird durch ihre Zustandstabelle repräsentiert. Der Ausgang ist definiert durch: $y = f(x, s)$. Eine Möglichkeit dieser Verifikation wäre, bei jeder möglichen Kombination von x und s den Ausgang zu prüfen (Exhaustive Verification). In der Praxis ist dies nur für kleine Schaltkreise möglich, da die Anzahl der Testfälle schnell gegen Unendlich anwachst. Ein realistischer Kompromiss ist erforderlich, um viele potenzielle Fehler mit vernünftigem Aufwand zu finden. Eine Kombination folgender Methoden ist von Vorteil:

- Anwenden der assertion-basierten Verifikation
- Sammeln von Testfällen aus mehreren Quellen
- Arbeitskräfte in zwei unabhängige Teams für Schaltungsentwurf und Testdesign einteilen.
- Rapid Prototyping (schnelles und frühes Erstellen von Prototypen)

10.1.1 Assertion-basierte Verifikation

Assertions sind kleine Codefragmente, die in den regulären Code eingebaut werden. Diese Fragmente fungieren als In-Code-Sanity-Checks. Sie sind nicht für die Synthese gedacht. Synthesizer können die meisten dieser Aussagen automatisch ignorieren. Assertions müssen unerwartete Bedingungen bereits vor dem Debuggen überprüfen. Solche Bedingungen sind z.B.:

- Speicheradressen ausserhalb des erlaubten Bereichs
- Eintretende Fehlerzustände in Zustandsmaschinen
- Unvorhergesehene Eingabewerte
- Numerischer Über-/Unterlauf

```
assert condition report "message" severity error; -- Syntax
i := to_integer(unsigned(bcd)); -- Example
assert i <= 9 report
  "unexpected bcd code. bcd = " & integer'image(i);
```

In VHDL 2008:

```
assert to_integer(unsigned(bcd)) <= 9 report
  "unexpected bcd code. bcd = " & to_string(bcd);
```

Die assert-Anweisung testet eine Condition (boolesche Bedingung). Wenn diese falsch ist, wird der Report ausgegeben. Mit Severity gibt man den Typ der Meldung an (note, warning, error, failure).

10.1.2 Auswahl von Testfällen

Die Stimuli sollten mit Vorsicht angewendet werden. Nachfolgend eine Auflistung von einigen wichtigen Testfällen:

- Stimuli anwenden, welche die Standardfunktionalität der Schaltung widerspiegeln. → Typisch für den Designer: Man definiert Testfälle um zu beweisen, dass die Schaltung die erwartete Funktionalität zeigt.
- Stimuli anwenden, welche wahrscheinlich zu ungewöhnlichen arithmetischen Bedingungen führen (z.B. Unter-/Überlauf, Teilung durch sehr kleine Zahlen usw.).
- Stimuli anwenden, welche bekannte Ausnahmefälle im Kontrollfluss widerspiegeln.
- Stimuli anwenden, welche aus realen Dienstleistungen bestehen.
- Anwenden von zufällig ausgewählten Testfällen.

Real-World-Daten und zufällige Testfälle können nur angewendet werden, wenn die erwartete Antwort auf zufällige Eingabedaten bekannt ist. Diese Methode fragt nach einem goldenen Referenzmodell, das die erwartete Antwort für Zufallsdaten erzeugt. Solche goldenen Referenzmodelle können bereits bestehende Schaltungen bei einer Schaltungsrekonstruktion sein, sie kann ein High-Level-Modell sein (z.B. Matlab oder Simulink-Darstellung einer Schaltung) oder eine als Rapid Prototype implementierte Hardware-Darstellung (z.B. als SoC Direkt mit HLS oder System Generator synthetisiert).

10.2 Simulation

Für kleine Block Level Verifikationen genügt eine kleine Testbench in VHDL. Für grössere Schaltungen ist jedoch ein allgemeiner Ansatz erforderlich. Stimuli und erwartete Response werden als Eingabe- und Ausgabevektoren in einer Datei angegeben. Mittels read-Funktion (im TEXTIO Package) wird Zeile für Zeile eingelesen, mit den tatsächlichen Signalen verglichen und überprüft. Bei einer Nicht-Übereinstimmung erscheint während der Simulation eine dementsprechende Konsolenmeldung. In einem synchronen Design muss ein fester Zeitplan für Stimuli und Response eingehalten werden. Dieser regelmässige Zeitplan muss daher den folgenden Regeln entsprechen:

- Für jeden Testfall ein periodisches Taktsignal angeben
- Pro Taktzyklus ein Stimulus/Response-Paar verwenden
- Ein streng regelmässiges Muster für Stimuli/Response anwenden

Automatisierte Test-Prozedur:

```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;
  use ieee.std_logic_textio.all;
  use std.textio.all; -- Fuer externes Stimuli-File
entity TestVectors_in_File_V2008 is
end entity TestVectors_in_File_V2008;
architecture tb of TestVectors_in_File_V2008 is
  constant freq : natural := 10000; -- Clockfrequenz
  constant T : time := (1 sec) / freq;
  constant delta_t : time := 0.1 * T; -- Sampling point, at least delta_t
    -- before next clock edge
  component edgedetpos
    port(
      clk : in std_ulogic;
      nrst : in std_ulogic;
      inp : in std_ulogic;
      oup : out std_ulogic);
  end component edgedetpos;
  signal clk_tb, nrst_tb, inp_tb, oup_tb, oup_exp : std_ulogic;
begin
  dut : edgedetpos
    port map(
      clk => clk_tb,
      nrst => nrst_tb,
      inp => inp_tb,
      oup => oup_tb);
  clk_generator : process
  begin
    clk_tb <= '0';
    wait for T/2;
    clk_tb <= '1';
```

```
    wait for T/2;
  end process;
  apply_testvector : process -- Load data from file
    file test_vector_file : text open read_mode is
      "d:\testvectors.txt"; -- File mit Testvektoren
    -- Variables needed for reading the file
    variable line_number : line; -- Text line buffer, actual line
    variable line_content : string(1 to 20); -- Variable for passing read values
    variable vector_nr : integer; -- Test vector number
  begin
    while not endfile(test_vector_file) loop -- Loop reads line by line of file
      readline(test_vector_file, line_number); -- New line
      next when line_number.all(1) = '-'; -- Skip comment
      read(line_number, line_content); -- read line number
      vector_nr := to_integer(line_content(1 to 3)); -- value is identified as test
        -- vector and stored in variable
      nrst_tb <= '1' when (line_content(5) = "1") else '0'; -- Read value stored as
        -- signal nrst_tb
      inp_tb <= to_std_ulogicvector(line_content(7 to 10)); -- Read value identified
        -- as input vector
      oup_exp <= to_std_ulogicvector(line_content(12 to 20)); -- Read value identified
        -- as expected response
      -- Simulation result is compared with expected response
      wait for (T-delta_t); -- Definition of time of evaluation
      assert (oup_tb = oup_exp) report "Error in test vector " & to_string(vector_nr) s
      wait until (rising_edge(clk_tb)); -- Wait for next clock cycle
    end loop;
    -- terminate simulation
    assert false report "Simulation Completed" severity failure;
  end process apply_testvector;
end architecture tb;
```

Die Simulation kann in verschiedenen Designebenen durchgeführt werden:

- Behavioral Simulation
  - Wird auf Stufe HDL durchgeführt → schnell
  - Verifiziert nur die Funktionalität
  - Es werden keine Timing-Informationen berücksichtigt
- Post Synthesis Simulation
  - Strukturelle Simulation
  - Verifiziert die synthetisierte Netzliste
  - Timing möglich, jedoch ungewöhnlich (da kein Routing)
- Post Implementation Simulation
  - Funktionelle und Timing Simulation möglich
  - Präziseste, jedoch langsamste

10.3 Debugging

Die Simulation weist zwar auf ein Fehlverhalten der Schaltung hin, zeigt jedoch nicht die genaue Ursache. Das Lokalisieren eines Fehlers nennt man Debugging.
Es gibt zwei (bzw. drei) Bedingungen für das Finden von Fehlern:

- Fehlersensibilisierung (bug sensitization): Stimuli müssen die Knoten auf die invertierten logischen Werte treiben können um einen Bug an einem bestimmten Knoten ausfindig machen zu können.
- Fehlerfortpflanzung und Beobachtung (bug propagation and observation): Ein falscher Logikwert an einem Knoten muss beobachtbar sein oder er muss zumindest an einen Knoten weitergegeben werden können, der beobachtbar ist.

10.3.1 Simulationswerkzeuge

Mit purer Input/Output Simulation kann man die obigen Bedingungen nicht erfüllen. Spezielle Simulationswerkzeuge bieten jedoch perfekte Beobachtbarkeit, mit welchen alle internen Knoten beobachtet werden können. Auch bei der Sensibilisierung dienen diese Simulationswerkzeuge perfekt: Während eine Testbench nur die Eingänge einer zu prüfenden Schaltung ansteuern kann, können hier interne Knoten direkt durch Erzwingen von Signalen auf einen festen Wert angesteuert werden.

10.3.2 Emulation

Während bei der Simulation beinahe das ganze Spektrum der Debuggingtools verwendet werden kann, sind Fehler bei der Emulation viel schwieriger zu finden. Es gibt fast keine Möglichkeit, interne Knoten von integrierten Schaltungen beobachten zu können. Aufgrund der umprogrammierbaren Struktur eines FPGA ist jeder Schaltungsknoten inhärent zu beobachten und kann inhärent stimuliert werden. Dies geschieht über die eingebaute JTAG-Schnittstelle. XILINX stellt zwei spezielle Blöcke zur Verfügung:

10.3.2.1 Virtual input and output (VIO)
Interne Knoten können als Ein- oder Ausgänge definiert werden. Der Mechanismus funktioniert über die JTAG-Schnittstelle. VIO Blöcke sind Hardwarekomponenten. Sie müssen in den Code integriert werden. Anschliessend werden sie instanziiert, synthetisiert und schliesslich implementiert. Die neu generierten Ein- und Ausgänge des VIO können während der Debug-Session getrieben und analysiert werden.

10.3.2.2 Integrated Logic Analyzer (ILA)
ILA dient zur Überwachung interner programmierbarer Logiksignale für die Nachanalyse. ILA Blöcke bieten mehrere konfigurierbare Triggereinheiten und bis zu 64 konfigurierbare Sonden für die Analyse in einer GUI-Darstellung. ILA bietet auch Cross-Trigger-Funktionalität zwischen dem ARM-Prozessor und dem logic fabric part.

11 Verification Design For Test

11.1 Warum testen?
In der Fabrikation von elektronischen Systemen gibt es unzählige Fehlerquellen. Je früher ein Fehler erkennt werden kann, desto weniger teuer kommt er zu stehen. Eine Regel besagt, dass die Kosten mit jedem Produktionsschritt, bei welchem der Fehler nicht erkannt wurde, um den Faktor 10 ansteigen. Bei der Herstellung von digitalen Chips kann in der Regel mit einer Ausbeute von 95 bis 98% gerechnet werden.

11.2 Begriffe

Defect Physikalische Fehlerquelle in integrierten Schaltungen

Fault Messbare Fehlfunktion einer Schaltung aufgrund von einem/mehreren Fehler(n)

Quality Mass für die exakte Erfüllung der Vorgaben (4 Bereiche: Design-, Herstellungs-, Auslieferungs- und Betriebsqualität)

Yield Verhältnis von funktionierenden Teilen auf einem Wafer: $y_f = \frac{\#good\_circuits}{\#manufactured\_circuits}$

Defect level Anteil an nicht-funktionierenden Chips, die unerkannt bleiben: $D_L = \frac{\#defective\_sold}{\#total\_sold} = 1 - y_f^{1-F_c}$

Fault coverage Anteil an Fehlern die mit einem Set von Testvektoren erkannt werden können: $F_c = \frac{\#detectable\_Faults}{\#possible\_Fault}$

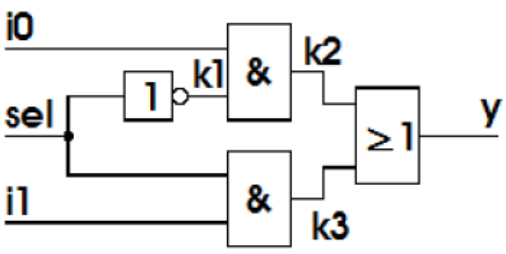
11.3 Fehlermodelle
Die nachfolgenden Fehlermodelle gibt es:
- **Stuck-At-0-Fault / Stuck-At-1-Fault**: Das entsprechende Signal wird auf einem Low-Pegel (Stuck-At-0-Fault) oder auf einem High-Pegel (Stuck-At-1-Fault) festgehalten und kann sich nicht ändern.
- **Bridging-Fault**: Bezeichnet einen Kurzschluss zwischen zwei Signalen. Ein Kurzschluss nach VDD oder VSS äussert sich wie ein Stuck-At-X-Fault. Ein Kurzschluss zwischen zwei Signalleitungen, verhält sich wie ein OR oder AND und kann im besten Fall funktional nachgewiesen werden.
- **Delay-Fault**: Dies sind eigentlich keine funktionalen Fehler, sondern Fehler in der Verzögerungszeit, hervorgerufen durch zu hohe Widerstandswerte der Verbindungen. Sie äussern sich letztlich jedoch funktional.

11.4 Fehlererkennung

11.4.1 Detektion von Stuck-At-X-Fault
Um einen solchen Fehler zu detektieren, muss der entsprechende Knoten kontrollier- und beobachtbar sein. Kontrollierbar ist er dann, wenn er über ein externes Signal auf Low oder High gesetzt werden kann. Beobachtbar ist er dann, wenn sein Zustand von aussen überprüft werden kann.

11.4.1.1 Beispiel
Anhand untenstehender Schaltung, wird nun eine Tabelle mit allen Testvektoren erstellt. In der Y-Spalte steht der Ausgangswert, wenn kein Fehler vorhanden ist. In den nachfolgenden Spalten, steht der Ausgangswert wenn der entsprechende Stuck-At-Fehler vorhanden ist.

Testvector	Sel	I1	I0	Y	K1: s-a-0	K1: s-a-1	K2: s-a-0	K2: s-a-1	K3: s-a-0	K3: s-a-1
TV0	0	0	0	0	-	-	-	1	-	1
TV1	0	0	1	1	0	-	0	-	-	-
TV2	0	1	0	0	-	-	-	1	-	1
TV3	0	1	1	1	0	-	0	-	-	-
TV4	1	0	0	0	-	-	-	1	-	1
TV5	1	0	1	0	-	1	-	1	-	1
TV6	1	1	0	1	-	-	-	-	0	-
TV7	1	1	1	1	-	1	-	-	0	-



Nun soll anhand der erstellten Tabelle, die Testvektorenanzahl auf ein Minimum beschränkt werden (Fault Collapsing). Mit den Testvektoren TV1, TV5 und TV7 kann somit jeder Fehler detektiert werden.

Testvector	Sel	I1	I0	Y	K1: s-a-0	K1: s-a-1	K2: s-a-0	K2: s-a-1	K3: s-a-0	K3: s-a-1
TV0	0	0	0	0	-	-	-	1	-	1
TV1	0	0	1	1	0	-	0	-	-	-
TV2	0	1	0	0	-	-	-	1	-	1
TV3	0	1	1	1	0	-	0	-	-	-
TV4	1	0	0	0	-	-	-	1	-	1
TV5	1	0	1	0	-	1	-	1	-	1
TV6	1	1	0	1	-	-	-	-	0	-
TV7	1	1	1	1	-	-	-	-	0	-

11.4.2 Fehlerabdeckung (Fault coverage)

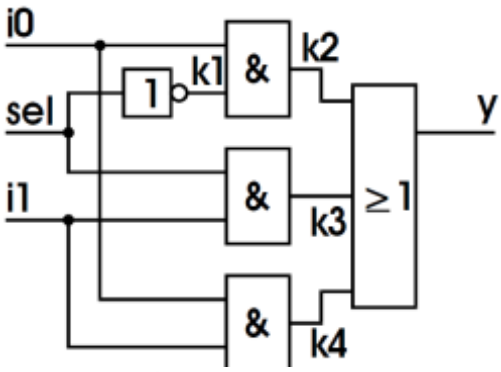
Die Fehlerabdeckung kann folgendermassen berechnet werden: Fehlerabdeckung $= \frac{\text{Anzahl detektierter Fehler}}{\text{Abzahl möglicher Fehler}} * 100\%$
Im Allgemeinen sollte darauf geachtet werden, dass eine Fehlerabdeckung von mindestens 95% erreicht wird.

11.4.3 Redundante Logik

Sollte redundante Logik eingebaut sein (zur Verhinderung von Hazards), so kann die Funktion nicht ohne Zusatzaufwand getestet werden.

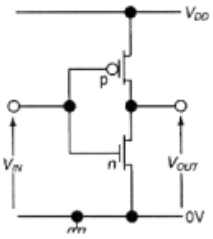
11.4.3.1 Beispiel Anhand von untenstehender Schaltung, wird die nachfolgende Tabelle erstellt. Dabei stellt sich heraus, dass ein Stuck-At-0-Fault am Knoten K4 nicht detektiert werden kann. Wenn aber die am Ausgang angeschlossene Logik darauf baut dass kein Hazard auftritt, kann dies zu einem Fehlverhalten führen.

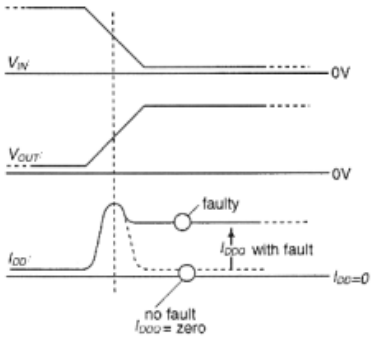
Testvector	Sel	I1	I0	Y	K1: s-a-0	K1: s-a-1	K2: s-a-0	K2: s-a-1	K3: s-a-0	K3: s-a-1	K4: s-a-0	K4: s-a-1
TV0	0	0	0	0	-	-	-	1	-	1	-	1
TV1	0	0	1	1	0	-	0	-	-	-	-	-
TV2	0	1	0	0	-	-	-	1	-	1	-	1
TV3	0	1	1	1	0	-	0	-	-	-	-	-
TV4	1	0	0	0	-	-	-	1	-	1	-	1
TV5	1	0	1	0	-	1	-	1	-	1	-	1
TV6	1	1	0	1	-	-	-	-	0	-	-	-
TV7	1	1	1	1	-	-	-	-	0	-	-	-



11.5 I_{DDQ}-Test

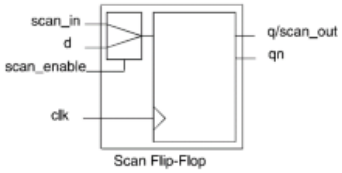
Beim I_{DDQ}-Test wird die Ruhestromaufnahme der Schaltung gemessen. Dieser Ruhestrom ist im Normalfall vernachlässigbar klein. Entsteht jedoch bei einem Signalwert ein Kurzschluss, so ist dies über den Ruhestrom feststellbar. Wird bei einer Schaltung festgestellt, dass direkt nach dem Anlegen der Versorgungsspannung ein hoher Strom fliesst, kann auf nachfolgende Tests direkt verzichtet werden, was die Testzeit stark verkürzt.





11.6 Scan Path

Die Scan Path Methode adressiert das Problem, dass interne Knoten von aussen schlecht kontrollierbar und beobachtbar sind. Dazu wird vor jedes Flipflop ein 2:1 Multiplexer geschalten (Scan Flipflop), mit dessen Hilfe alle Flipflops zu einem riesigen Schieberegister zusammengeschaltet werden.



Die folgenden speziellen Signale sind nun vorhanden:

- *scan_enable*: Legt die Multiplexer Stellung fest.
- *scan_in*: Eingangssignal für das Schieberegister. Wird während dem Normalbetrieb nicht benötigt.
- *scan_out*: Ausgangssignal des Schieberegisters. Führt auf den *scan_in* Eingang des nächsten Flipflops.

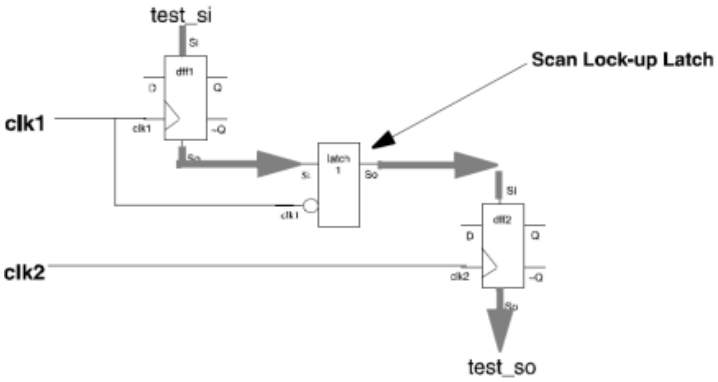
In einem ersten Schritt wird das ganze Schieberegister seriell befüllt. Damit können alle internen Eingänge kontrolliert werden. In einem zweiten Schritt wird das System für einen Taktzyklus in den Normalbetrieb versetzt. Im letzten Schritt wird das Schieberegister nun wieder seriell ausgelesen.

11.6.1 Parallele Pfade

Je mehr Flipflops sich in einem Design befinden, umso länger wird der Scan Pfad und somit auch die Testzeit. Durch die Verteilung der Flipflops auf mehrere Scan Pfade wird die Testdauer verkürzt. Es wird dabei versucht, die Scan Pfade möglichst auszubalancieren, damit es bei allen Pfaden gleich lange dauert um die Daten auszulesen oder zu schreiben.

11.6.2 Scan Path bei mehreren Taktdomänen

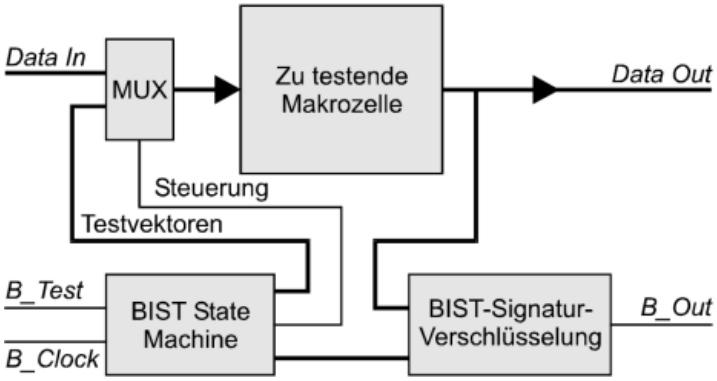
Sind in einem System unterschiedliche Taktdomänen vorhanden, so kommen Lock-Up Latches, an den Übergängen von einer Domäne zur anderen, zum Einsatz. Diese stellen sicher, dass keine laufzeitbedingten Fehler auftreten.



11.7 BIST (Built In Self Test)

Der Test von grossen programmierbaren Blöcken (z.B. RAM, ROM, PLA, ...) ist mit Scan Paths nur schwer möglich, da die Anzahl Testvektoren nicht wirklich reduziert werden kann, da auch wirklich jede Zelle überprüft werden muss. Aus diesem Grund wird jeder Makrozelle eine zusätzliche Logik zugeordnet, mit welcher diese einen Selbsttest durchführen kann.

11.7.1 Prinzip



Nebenstehend ist der prinzipielle Aufbau einer BIST-Zusatzlogik zu sehen. In den Eingangspfad der zu testenden Makrozelle ist ein 2:1-Multiplexer geschaltet. Über den Block BIST State Machine, der die Steuerlogik und einen Testmustergenerator enthält, wird dieser Multiplexer so geschaltet, dass entweder von aussen kommende Daten an die Makrozelle geführt werden (Normalbetrieb), oder solche vom Testmustergenerator (Testbetrieb). Die Ausgänge der Makrozelle werden neben der normalen Verdrahtung dem Block BIST-Signature-Verschlüsselung zugeführt. Dieser wertet die Testergebnisse aus und meldet sie nach aussen.

11.8 BST (Boundary Scan Test)

Die zunehmende Packungsdichte erschwert den Test von fertigen Baugruppen erheblich. Um die Anzahl erforderliche Prüfanschlüsse zu minimieren, wurde der BST entwickelt.

11.8.1 Konzept

Jeder Ein- und Ausgang eines Chips wird mittels einer BSC (Boundary Scan Cell) vom Kern der Schaltung abgetrennt. Alle diese BSCs sind wiederum zu einem grossen Schieberegister zusammengeschalten (BSR - Boundary Scan Register).

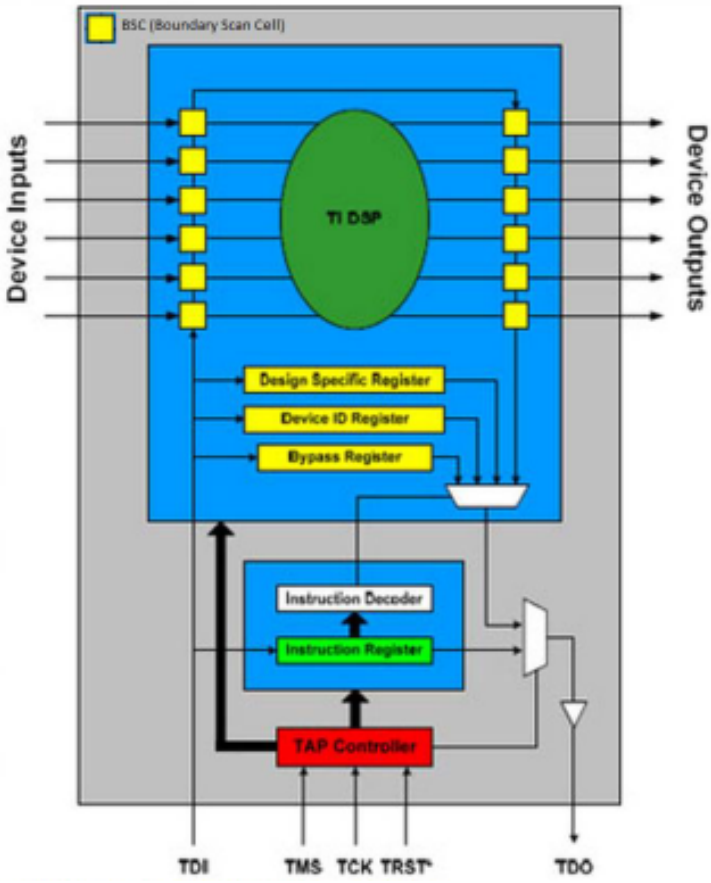
11.8.1.1 Signale Jeder Chip stellt nach Aussen die folgenden Signale zur Verfügung, resp. benötigt die folgenden Signale:

- *TDI*, *TDO*: Test Data In / Test Data Out, werden auf der Leiterplatte zu einem Schieberegister verknüpft. Sprich jeder TDO führt auf den TDI des nächsten Chips.
- *TCK*: Test Clock
- *TMS*: Test Mode Select
- *TRST**: Optional, bringt den Chip in einen definierten Zustand

11.8.2 Realisierung auf dem Chip

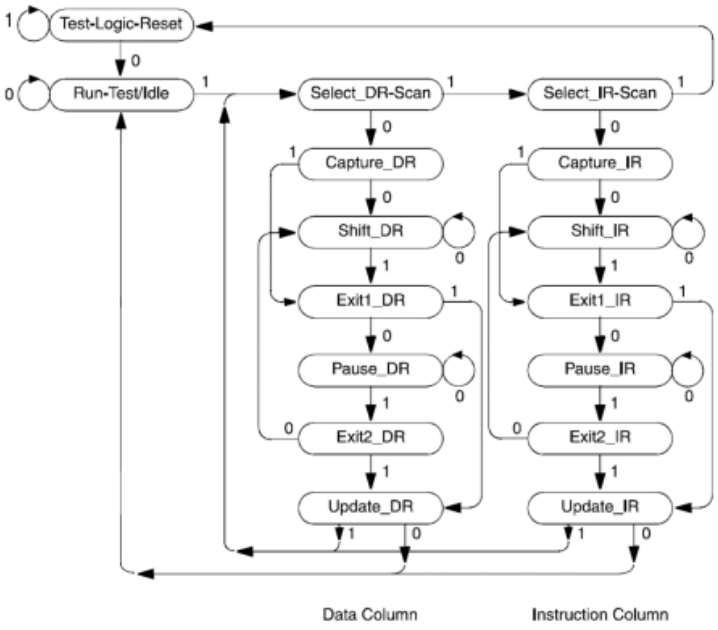
Jeder Chip muss für BST die folgenden Elemente beinhalten:

- **Bypass-Register**: Überbrückt dass BSR. So kann der entsprechende IC bei gewissen Test übersprungen werden. Dies verkürzt wiederum die Testdauer.
- **Device-ID-Register**: Enthält eine ID des Chips. Dieses Register kann während dem Test ausgelesen wertden um z.B. auf korrekte Bestückung zu prüfen.
- **TAP (Test Access Port)**: Schnittstelle nach Aussen
- **Instruction-Register**: Steuert die internen notwendigen Multiplexer Einstellungen.



11.8.3 TAP-Kontroller

Die Funktion des TAP-Kontrollers ist in nebenstehender Grafik dargestellt. In den Zustand *Test-Logic-Reset* wird gewechselt wenn das *TRST**-Signal gesetzt wird oder indem das Signal *TMS* während fünf Taktzyklen auf einem High-Pegel gehalten wird.



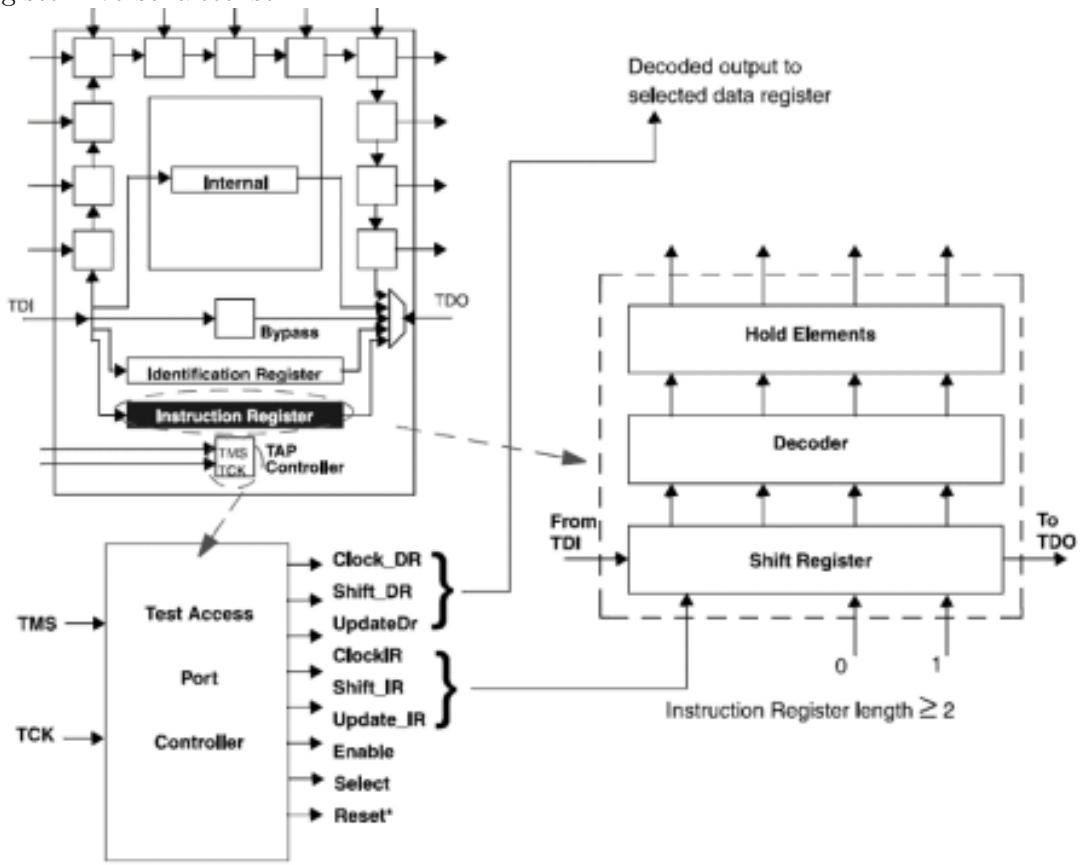
11.8.4 Instruktionen

11.8.4.1 Instruktions-Register Das Instruktions-Register muss mindestens eine Länge von zwei Bit aufweisen um die verbindlichen Befehle dekodieren zu können. Das Register kann aber durchaus grösser sein, da viele Hersteller noch eigene Befehle implementieren. Wenn der TAP-Kontroller sich im Zustand *Shift_IR* befindet, wird die Instruktion seriell vom Eingang *TDI* in das Instruktions-Register geladen.

11.8.4.2 Verbindliche Befehle Die folgenden Befehle müssen in jedem Fall implementiert sein:

- **BYPASS**: Der Befehlscode besteht aus lauter Einsen (11...1). Ist dieser Befehl aktiv, muss im Zustand *Shift_DR* das Bypass-Register in den *TDI / TDO* Pfad geschaltet sein. Ebenfalls dürfen die Testregister die Kernlogik nicht beeinflussen.
- **SAMPLE / RELOAD**: Nimmt eine Momentanaufnahme des aktuellen Zustandes der IC-Anschlüsse auf oder setzt die IC-Anschlüsse auf einen definierten Zustand. Das BSR wird im Zustand *Shift_DR* bei diesem Befehl in den *TDI / TDO* Pfad geschalten.
- **EXTEST**: Der Befehlscode besteht aus lauter Nullen (00...0). Die Chiplogik wird bei diesem Befehl von der Leiterplatte isoliert (Die Kernlogik muss so abgeschottet sein, dass sie nicht beschädigt werden kann). Das BSR wird im Zustand *Shift_DR* bei diesem Befehl in den *TDI / TDO* Pfad geschalten.

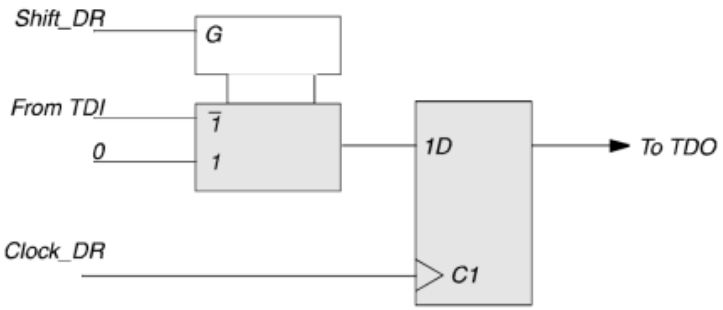
11.8.4.3 Interner Aufbau Nachfolgend ist dargestellt, wie der TAP-Kontroller intern mit den vorhandenen Registern verschaltet ist.



11.8.4.4 Optionale Befehle Die nachfolgenden Befehle sind optional, werden jedoch auch häufig implementiert:

- **INTEST**: Testet die Kernlogik eines Chips. Die Kernlogik wird zu diesem Zweck von den Ein- und Ausgängen isoliert.
- **RUNBIST**: Startet einen eingebauten BIST-Selbsttest. Der Vorteil dieser Variante ist (im Gegensatz zum *INTEST*) dass die Konfiguration des Chips nicht selber vorgenommen werden muss und am Schluss nur das Ergebnis überprüft werden muss.
- **IDCODE**: Liefert den Inhalt aus dem Identifikations-Register zurück.

11.8.5 Aufbau BYPASS-Register

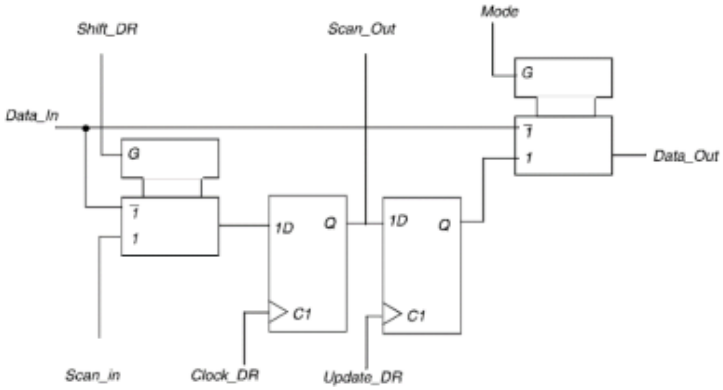


11.8.6 Aufbau ID-Register

Das Identifikations-Register ist immer 32bit breit.

- **Bit 31-28**: Chip-Version
- **Bit 27-12**: Frei verwendbar vom Hersteller
- **Bit 11-1**: Standardisierter Hersteller Code
- **Bit 0**: Immer 1

11.8.7 Aufbau BSC



Für jeden Ein- und Ausgang eines Chips wird eine Zelle benötigt. Falls aber Three State-Ausgänge oder bidirektionale Anschlüsse vorkommen, dann muss auch deren Enable-Signal mit einer BSC versehen werden. Falls mehrere solcher Signale zu einem Bus zusammengefasst werden, dann werden alle entsprechenden Enable-Signale von derselben BSC geschaltet.

12 HDL Attributes

Wenn die Vivado-Synthese das Attribut unterstützt, wird eine Logik erstellt, die das verwendete Attribut wiedergibt. Wenn das Attribut vom Tool nicht erkannt wird, übergibt die Vivado-Synthese das Attribut und seinen Wert an die erzeugte Netzliste. Es wird davon ausgegangen, dass ein Tool weiter unten im Flow das Attribut verwenden kann.

12.1 Attribute in XDC Files

Einige Synthese-Attribute können auch aus einer XDC-Datei sowie der ursprünglichen RTL-Datei gesetzt werden. Im Allgemeinen sind Attribute, die in den Endstadien der Synthese verwendet werden, in der XDC-Datei erlaubt. Attribute, die zu Beginn der Synthese verwendet werden und den Compiler beeinflussen, sind im XDC nicht zulässig. Zum Beispiel KEEP und DONT_TOUCH sind nicht erlaubt, da zum Zeitpunkt des Auslesens des Attributs die Komponenten, welche das Attribut KEEP oder DONT_TOUCH aufweisen, bereits optimiert wurden und somit zu diesem Zeitpunkt nicht mehr existieren.

set_property <attribute><value><target>

set_property MAX_FANOUT 15 [get_cells in1_int_reg]

12.2 Allgemeine Attribute

12.2.1 ASYNC_REG

Der Zweck dieses Attributs ist es, dem Tool mitzuteilen, dass ein Register asynchrone Daten im D-Eingangspin relativ zum Source Clock empfangen kann oder dass das Register ein Synchronisationsregister innerhalb einer Synchronisationskette ist. Dieses Attribut kann auf jedes Register angewandt werden. Werte sind FALSE (Default) und TRUE. Es kann im RTL oder XDC File gesetzt werden.

```
attribute ASYNC_REG : string;
attribute ASYNC_REG of sync_regs: signal is "TRUE";
```

12.2.2 KEEP

Vermeidet Optimierungen, bei denen Signale entweder optimiert oder in Logikblöcke absorbiert werden.

```
signal sig1 : std_logic;
attribute keep : string;
attribute keep of sig1 : signal is "true";
begin
  sig1 <= in1 and in2;
  out1 <= sig1 and in3;
```

12.2.2.1 KEEP_HIERARCHY

Das Synthese-Tool versucht normalerweise die gleichen allgemeinen Hierarchien zu behalten, die im RTL spezifiziert werden. Aus Gründen der Belastbarkeitsqualität (Quality of resilience (QoR)) könnte die Hierarchie abgeflacht oder modifiziert werden. Mit KEEP_HIERARCHY kann das verhindert werden. Kann nur im RTL gesetzt werden. Sollte nicht auf Tristate oder I/O Buffer Module angewendet werden.

```
attribute keep_hierarchy : string;
attribute keep_hierarchy of beh : architecture is "yes";
```

12.2.2.2 DONT_TOUCH

DONT_TOUCH funktioniert wie KEEP oder KEEP_HIERARCHY. DONT_TOUCH basiert auf Vorwärts Annotation.

12.3 FSM Attribute

12.3.1 FSM_ENCODING

Bestimmt welcher Codierungsstil verwendet werden soll. Der Wert *auto* ist der Defaultwert. Es wird die am besten geeignetste FSM-Codierung ausgewählt. Akzeptable Werte sind: *one_hot*, *sequential*, *johnson*, *gray*, *none* und *auto*.

```
attribute fsm_encoding : string;
attribute fsm_encoding of state : signal is "gray";
```

12.3.2 FSM_SAVE_STATE

Hiermit kann man bestimmen, was bei einem nicht definierten Zustand zu tun ist. Dabei wird die entsprechende Logik eingefügt. Akzeptable Werte sind:

- auto: Verwendet eine Hamming-3-Codierung für die Autokorrektur für ein Bit/Flip.
- reset_state: Erzwingt die State Machine in den Reset-Zustand mit Hamming-2-Codierung für ein Bit/Flip. Rücksetzbedingung nicht vergessen!
- power_on_state: Erzwingt die State Machine in den Power-On-Zustand mit Hamming-2-Codierung für ein Bit/Flip.
- default_state: Erzwingt die State Machine in den Zustand, der mit dem Standardzustand im RTL angegeben wird (auch wenn dieser Zustand nicht erreichbar ist), mittels Hamming-2-Codierung für ein Bit / Flip.

```
attribute fsm_safe_state : string;
attribute fsm_safe_state of state : signal is "reset_state";
```

12.4 Andere Attribute

- MAX_FANOUT: Gibt Fanout-Grenzwerte für Register und Signale bekannt.
- SHREG_EXTRACT: Ermöglicht eine Extraktion von Schieberegistern.
- SRL_STYLE: Bestimmt wie SRLs abgeleitet werden sollen.
- TRANSLATE_OFF: Gibt ein Codeblock, welcher ignoriert werden kann, an.
- USE_DSP48: Standardmässig werden mults, mult-add, mult-sub, mult-accumulate in DSP48 Blöcken dargestellt. Addierer, Subtrahierer und Akkumulatoren jedoch nicht (Logik). Mit USE_DSP48 versucht man nun möglichst viele arithmetische Strukturen in DSP48 Blöcken darzustellen.
- MARK_DEBUG: Spezifiziert, dass ein Netz mit den Vivado Lab Tools debuggt werden soll.
- IO_BUFFER_TYPE: Kann man Synthese von Puffern steuern, die auf ein bestimmtes Signal in einem Design angewendet werden.
- IOB: Teilt mit, dass bei der Implementierung ein Register in I/O Buffer gepackt werden soll.
- GATED_CLOCK: Erlaubt die Umwandlung von Gated Clocks.
- CLOCK_BUFFER_TYPE: Kann man Synthese von Puffern steuern, die auf ein bestimmtes Signal in einem Design angewendet werden.
- BLACK_BOX: Kann man mitteilen, dass ein bestimmtes Modul innerhalb eines Designs nicht synthetisiert werden soll.

13 Digital Design For ASICS

Moderne FPGA-Designs haben viel gemeinsam mit modernen ASIC-Designs. Trotzdem gibt es einige grundlegende Unterschiede.

13.1 Zielanwendungen

Im Vergleich zu FPGA-Designs sprechen die ASIC-Designs einen anderen Markt an. ASIC-Designs eignen sich in der Regel für Anwendungen mit hohem Produktionsvolumen, d.h. mindestens 100'000 Stücke pro Jahr, weil die NRE Kosten (einmalige Entwicklungskosten) sehr hoch sind.

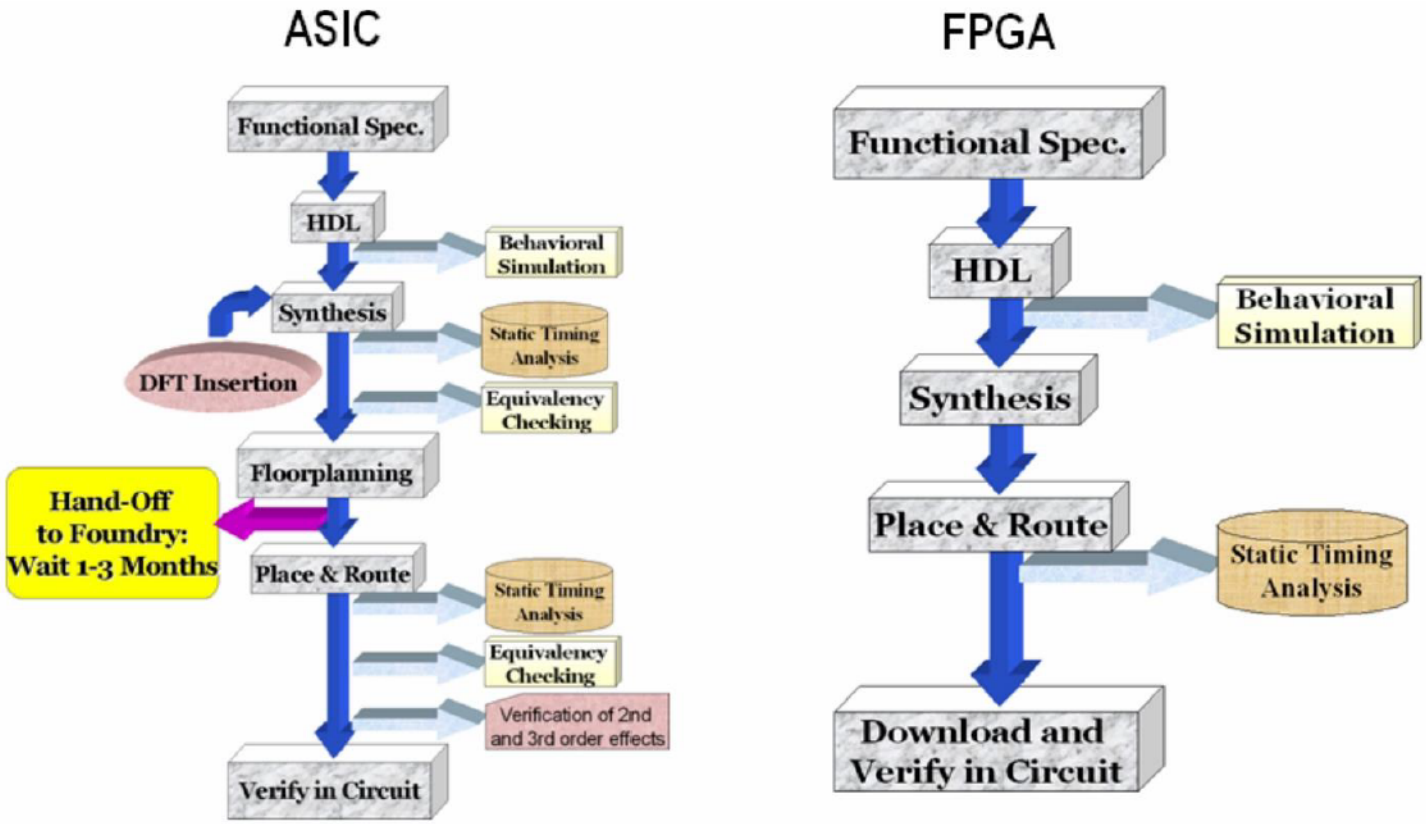
13.1.0.1 Gesamtkosten pro Einheit

$c = \frac{c_0}{n} + c_1$ c_0: Summe der einmaligen Entwicklungskosten → konstant // n: Anzahl der entwickelten ASICS // c_1: Kosteninkrement pro erzeugter Schaltung (für Rohwafer, Waferbearbeitung, Prüfung, Verpackung, Logistik usw.) → linear steigend

c_0 ist ist abhängig von:
- Herstellungsprozess und Prozessoptionen, die die Masken- und die Waferherstellungskosten bestimmen
- Lizenzkosten und Lizenzgebühren für IP-Cores
- CAD-Tool-Kosten
- Engineering-Kosten für Design der Schaltung und Design der Testlösung
- Aufbau der Versorgungs- und Logistikkette
- etc.

c_1 ist abhängig von:
- Schaltungsgrösse
- Testzeit
- Herstellungsertrag
- Verpackungskosten
- etc.

Beim FPGA-Design ist c_0 relativ klein im Vergleich zum ASIC-Design. Die Startkosten bei c_1 sind beim FPGA-Design tiefer als im ASIC-Design. Jedoch steigen die Kosten beim FPGA-Design steiler an als beim ASIC-Design. Der Schnittpunkt der beiden Gesamtkosten ergibt ungefähr Volumengrenze zwischen FPGA- und ASIC-Entwicklung.



13.2 Design Flow

13.2.1 Synthese (FPGA vs. ASIC)

13.2.1.1 FPGA

- Target device bestimmen/auswählen (Hersteller, Familie, FPGA Typ)
- Wahl des Sythetisiertools
- Constraints hinzufügen (im XDC-Format)
- Für kombinatorischen Teil werden LUTs und für den sequentiellen Teil FF erzeugt.
- Hat Scankette eingebaut, da Scan und JTAG beim Programmieren gebraucht wird.

13.2.1.2 ASIC

- Wahl des Designhouse: ASIC-Design erfordert Wissen, Erfahrung und Infrastruktur, welche nicht immer vorhanden ist → evtl. Designhouse
- Wahl des Sythetisiertools (teuer!)
- Wahl der Technologie
- Wahl der digitalen Zellenbibliothek
  - High Density (9 or 10 tracks high): Mittlere Transistorgrösse für hohe Dichte und gute Performance, low power
  - Ultra-High Density (7 or 8 tracks high): Kleine Transistorgrösse für sehr hohe Dichte und low power
  - High Performance (12 tracks high): Grosse Transistorgrösse für optimale Geschwindigkeit, auch low power features
- Wahl der Soft- und Hard IP Blöcke (Block RAMs, DSP Blöcke, etc. sind normalerweise nicht vorhanden →

kaufen
- Constraints hinzufügen (im SDC-Format)
- Erzeugung generische Netzliste (im .edif Format), welche grundlegende boolesche Funktionen, die keine realen Gatter mit Layout und Timing sind, verwendet. Technologieunabhängig.
- Mapping, bei welchem die Grundfunktionen auf reale Zellen aus der Zellbibliothek abgebildet werden.
- Hat keine eingebauten Testfunktionen. Die Testlogik wird erzeugt und während der Synthese hinzugefügt → FFs werden durch Scan-FFs ersetzt. Scan-Kette wird erzeugt.

13.2.2 Floorplanning

Floorplanning bezieht sich darauf, einzelne Elemente, Schaltungsblöcke und IOs vorläufig auf einem physikalischen Layout zu platzieren. Im FPGA-Design ist das Floorplanning ein Teil des Place and Route (Hinzufügen von Physical Constraints).

13.2.3 Place and Route (FPGA vs. ASIC)

13.2.3.1 FPGA

- Zuweisung der verschiedenen Netzlistenelemente zu vorhandenen LUTs, FFs und Hardware-Makros auf dem FPGA-Chip → wird automatisch bei "Run Implementation" und "Generating Bitstream" zugewiesen.
- Einfluss nur bedingt via Physical Constraints

13.2.3.2 ASIC Besitzt mehrere Teilschritte:
- Power Planning: Generiert den Powerring, der VDD und VSS um den gesamten Chip erzeugt. Für horizontale Verbindungen und vertikale Verbindungen werden unterschiedliche Metallschichten verwendet.
- Definition Power Grid für Standardzellen: Abhängig von der ausgewählten Standardzellenbibliothek und ihrer charakteristischen Höhe wird das Gitter für VDD und VSS der Standardzellen erzeugt.
- Platzierung der Standardzellen (iterativer Prozess): Standardzellen werden mit einem bestimmten Dichtefaktor platziert, was bedeutet, dass ein gewisser Raum zwischen den Zellen frei bleibt. Dann versucht das Tool, die Leitungen zwischen den Zellen zu setzen. Wenn das Routing nicht erfolgreich war, startet der Prozess erneut mit einem neuen Placement. Wenn das Routing erfolgreich war, kann es weiterhin sinnvoll sein, den Prozess fortzusetzen, indem versucht wird, die Dichte zu erhöhen und dadurch die Gesamtgrösse des Chips zu verringern.
- Erzeugung der Clockstruktur: Alle sequentiellen Zellen müssen mit einem Taktsignal verbunden sein. Dieses Signal muss bestimmten Qualitätsanforderungen entsprechen. Das Ergebnis der Taktgenerierung ist ein Taktnetzwerk, das aus Puffern besteht, die in Form eines Baumes mit Puffern verschiedener Treiberstärken verbunden sind, so dass der Clock Skew an den einzelnen Endpunkten minimiert wird.
- Nach erfolgreichem Place and Routing wird der Leerraum in den Standardzellenfeldern mit sogenannten Füllerzellen gefüllt, um "Löcherïn den Zeilen zu schliessen. Dies ist notwendig, um kontinuierliche p- und n-Wannen entlang einer ganzen Standardzellenreihe zu haben.

13.2.4 Verifikation FPGA vs. ASIC

13.2.4.1 FPGA Verifikation gleich nach herunterladen der Programmdaten möglich. Man muss nicht auf einen Prototypen warten.

13.2.4.2 ASIC Verifikation nicht gleich möglich. Folgende Schritte müssen VOR der Verifikation vollzogen werden:
- Sign-off: Daten der Prdouktion übermitteln und prüfen lassen
- Generierung der Maske
- Waferbearbeitung
- Wafertests, Wafer Dicing (schneiden), Verpacken