

## 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

### 1.2.2 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 vier 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.

### Spezialressourcen der programmierbaren Logik

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

### 1.3 Design Herausforderungen und Bedarf

#### 1.3.1 Herausforderungen

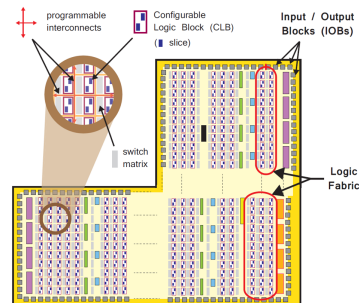
- Komplexität: Hohe Parallelität auf versch. Ebenen
- Heterogenität: Tools / Komponenten / Hersteller / Hardwarebeschreibung / Programmiersprache
- Time to market: Marktdruck

#### 1.3.2 Bedarf

- Wiederverwendbarkeit: Systemfunktionalität kann nicht länger von Anfang an entwickelt werden  $\rightarrow$  Komponenten und Subsysteme wiederverwenden  $\rightarrow$  IP-basiertes Design
- Hierarchie: Weitere Hierarchiestufen sind erforderlich (sowohl Hard- wie auch Software)  $\rightarrow$  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

### 1.2.1 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)



### 1.4 IP-basiertes Design

#### 1.4.1 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.

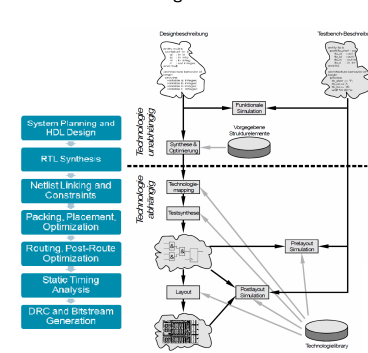
#### 1.4.2 IP Core Typen

- Hard IP Core: Blackbox in optimierter Layoutform. Sind als fertige Schaltung herstellereitig 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

### 1.4.3 Bezug von IP Core

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

### 1.5 Traditioneller Design Flow



### 1.7 Design Tools

#### 1.7.1 Vivado IDE Solution

##### Haupteigenschaften

- Interaktives Design und Analyse: Timing Analysen, Konnektivität, Ressourcen Nutzung, Timing 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

##### Design Database

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

#### 1.7.2 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 Constraints 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]
```

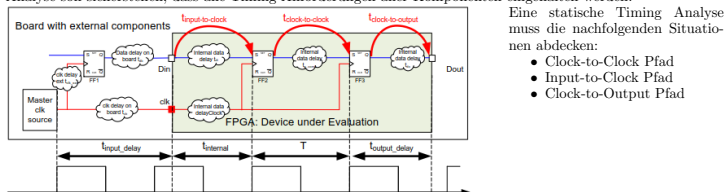
### 2.1.1 Reihenfolge

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

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

### 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.



#### 2.2.1 Input Delay

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

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

#### 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.

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

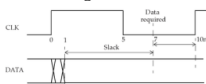
### 2.2.3 Clock Skew

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.

#### Berechnung



$Slack = RequiredTime - ArrivalTime$  wobei  $RequiredTime = T_{period} - T_{setup}(CaptureFlipFlop)$   
Nach nebenstehendem Bild:  $RequiredTime = 10ns - 3ns = 7ns$  und  $ArrivalTime = 1ns$  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 *clk\_in* zugeführt.

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

### 2.3.2 Generated Clocks

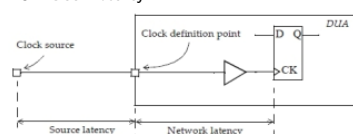
Generated Clock werden von speziellen Blöcken in einem FPGA generiert (z.B. MMCM). Sie können auch von Userlogik erzeugt werden. Diese Generated Clocks sind jedoch an einen anderen Clock gebunden und teilen diesen z.B. herunter, oder bewirken eine Phasenverschiebung.

### 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



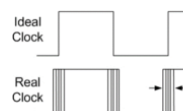
Clocks erreichen verschiedene Punkte in einem Design mit unterschiedlichen Verzögerungen. Diese Verzögerungen können in zwei Arten unterschieden werden:

- 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 (bis das Signal konstant ist) werden Jitter genannt und in Nanosekunden gemessen.

### 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
```

### System Jitter

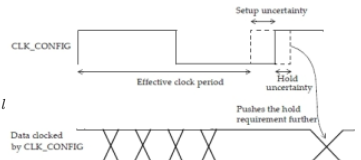
Der System Jitter spezifiziert den Jitter für alle Clocks im System (auch für die Primary Clocks). Er wird benutzt um starkes Rauschen zu modellieren. Mit dem nachfolgenden Befehl kann der Jitter definiert werden:

```
# 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]
```



## 2.4 I/O Pin Constraints

```
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

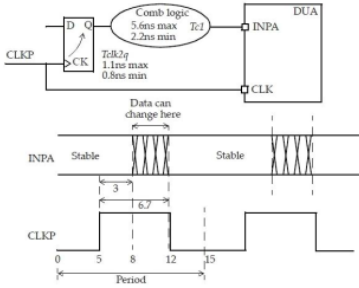
Input Delays geben an, mit welcher Verzögerung Daten an einem Eingangsport anliegen. Diese Verzögerung muss relativ zu einem Clock angegeben werden.

#### 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 Zeit für min und max verwendet.
- `-clock_fall`: Gibt an dass das 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).

#### Beispiel



```
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

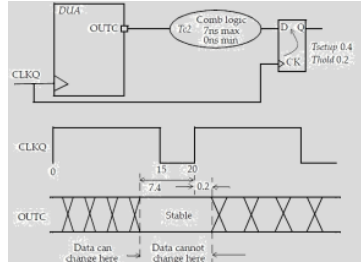
Output Delays geben an, in welchem Zeitbereich die Daten am Ausgangsport stabil sein müssen. Dieser Zeitbereich wird wiederum relativ zu einem Clock angegeben.

#### 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 Zeit für min und max verwendet.
- `-clock_fall`: Gibt an dass das 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).

## Beispiel



```
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 Entitäts, Architekturen, Funktionen und Prozesse verwenden
- Namen sollten lowercase sein und zum trennen von Wörtern sollte man Underscores verwenden
- Entitäts 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 im Bezug auf die Änderbarkeit unverzichtbar. Konstanten in VHDL-Packages können in mehreren Designseinheiten 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 downto 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
    generic (
        AddrWidth_g : positive := 16;
        DataWidth_g : positive := 32;
        EnableStreamUp : boolean := true;
    )
    port (
        MAXIS.TVALID : out std_logic;
        MAXIS.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] <functionname>;
```

#### — 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 and use statements>] — Package Declaration
package <package_name> is
    <declarations>;
```

```
end <package_name>;
[<library and 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 PARGEN(AVECT : std_ulogic_vector) return std_ulogic;
end package parity_package;
package body parity_package 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;
    end function;
```

```

    return POVAR;
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;
```

### 3.5 IP Blöcke

#### 3.5.1 Konfiguration von IP Blöcken

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

#### 3.5.2 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 Repository (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 Dokumentationen beschrieben. Ersetzt nicht HDL oder Software (enthält nur High-Level-Informationen).

#### 3.5.3 Einbinden von IP Blöcken in eigenes Design

- IP Repository (normalerweise in Projekt oder auf Firmenlaufwerk) dem Projekt bekannt machen.
  - IP Block aus Katalog auswählen (add IP).
  - 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.
  - 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 (
    clk : in std_logic;
    ena : in std_logic;
    ...
    dinb : in std_logic_vector(15 downto 0);
    doutb : out std_logic_vector(15 downto 0)
);

```

L. Leuenberger, M. Ehrler

[CC BY-NC-SA]

16. Februar 2017

```

end component;
-- End COMPONENT Declaration
-- Begin Cut here for INSTANTIATION Template
your_instance_name : blk_mem_gen_0
port map (
    clk => clk ,
    ena => ena ,
    wea => wea ,
    ...
    dinb => dinb
);

```

#### 3.5.4 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.

## 4 Speicher - Lukas

### 4.1 Typen von Speicher in FPGA

In einem FPGA sind normalerweise zwei Arten von Speicher vorhanden:

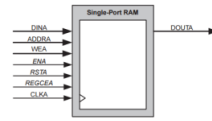
- Distributed Memory:** Distributed Memory besteht aus vielen LUT Tabellen. Der Vorteil dieser Variante ist, dass jede beliebige Grösse von Speicher realisiert werden kann. Desweiteren kann dieser Speicher an jedem Ort in einem FPGA erstellt werden. Ideal für kleine Speichergrößen.
- Block Memory:** 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).

#### 4.1.1 ROM

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

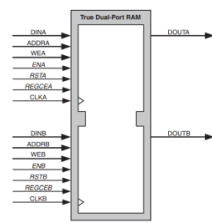
#### 4.1.2 Interface Arten

Beide Speicherarten können mit zwei verschiedenen Schnittstellen implementiert werden.



#### 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.



#### 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.

L. Leuenberger, M. Ehrler

[CC BY-NC-SA]

16. Februar 2017

### 4.2 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.

- Die Datengröße und die Adressengröße soll mit generischen Parametern definiert werden.
- Der Adressbereich soll mit einer Konstante definiert werden.
- RAM soll mit einem zweidimensionalen Array beschrieben werden.
- Der Schreibzugriff muss in einem Prozess beschrieben werden.

#### 4.2.1 Block RAM - Single-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 (
    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 MEMDEPTH : integer := 2 ** ADDR_WIDTH;
type ram_type is array (0 to MEMDEPTH - 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;
                else
                    dout <= blockram (to_integer(unsigned(addr)));
                end if;
            end if;
        end if;
    end process ram_write;
end RTL;

```

#### 4.2.2 Distributed RAM - Single-Port

-- Gleich wie bei Block Ram Single-Port

```

architecture RTL of RAM_Test is
constant MEMDEPTH : integer := 2 ** ADDR_WIDTH;
type ram_type is array (0 to MEMDEPTH - 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;
end RTL;

```

L. Leuenberger, M. Ehrler

[CC BY-NC-SA]

16. Februar 2017

```

end process ram_write;

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

4.2.3 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 MEMDEPTH : integer := 2 ** ADDR_WIDTH;
type ram_type is array (0 to MEMDEPTH - 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;

```

#### 4.2.4 ROM

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

```
constant rom : ram_type;
```

#### 4.2.5 Synthese Attribut

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";

```

L. Leuenberger, M. Ehrler

[CC BY-NC-SA]

16. Februar 2017



#### 4.2.6 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 MEMDEPTH : integer := 2 ** ADDRWIDTH;
  type ram_type is array (0 to MEMDEPTH - 1) of std_logic_vector (DATAWIDTH - 1
    downto 0);
  impure function init_mem (mif_file_name : in string) return ram_type is
    file mif_file : text open read.mode is mif_file_name;
    variable mif_line : line;
    variable temp_bv : bit_vector (DATAWIDTH - 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")
```

#### 4.3 Memory IP Generators

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

#### 5 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.

##### 5.1 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]
```

##### 5.2 Allgemeine Attribute

###### 5.2.1 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";
```

###### 5.2.2 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;
```

#### 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";
```

#### DONT\_TOUCH

DONT\_TOUCH funktioniert wie KEEP oder KEEP\_HIERARCHY. DONT\_TOUCH basiert auf Vorwärts Annotation.

#### 5.3 FSM Attribute

##### 5.3.1 FSM\_ENCODING

Bestimmt welcher Codierungsstil verwendet werden soll. Der Wert *auto* ist der Defaultwert. Es wird die am besten geeignete 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";
```

##### 5.3.2 FSM\_SAVE\_STATE

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";
```

#### 5.4 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.

#### 6 Serial Communication Interfaces

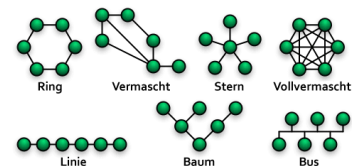
##### 6.1 Begriffe

Verschiedene Begriffe existieren im Zusammenhang mit Schnittstellen:

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

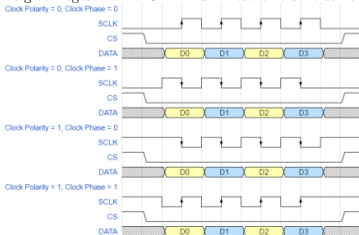
#### 6.2 Netzwerk Arten

Verschiedene Netzwerk Arten existieren in der Praxis:



##### 6.3.1 Taktpolarität und Taktphase

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



##### 6.3.2 Timing

Eine mögliche SPI Transaktion ist untenstehend zu sehen:



##### 6.3.3 Hardwaremäßige Implementierung

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



##### 6.3.4 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 statePreset, 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
      statePreset <= idle;
      counterPresent <= 0;
      shiftRegTransmitPresent <= (others => '0');
      shiftRegReceivePresent <= (others => '0');
    elsif rising_edge(clkIn) then
      statePreset <= stateNext;
      counterPresent <= counterNext;
      shiftRegTransmitPresent <= shiftRegTransmitNext;
      shiftRegReceivePresent <= shiftRegReceiveNext;
    end if;
  end process regLogic;
  nextStateLogic : process (counterPresent, start, statePreset)
  begin
    stateNext <= idle;
    counterNext <= 0;
    case statePreset 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;
    CS <= '1' when statePreset = idle else '0';
    SCLK <= '1' when statePreset = sclk1 else '0';
```

```

transmitShiftRegister : process (dataTransmit, shiftRegTransmitPresent,
  shiftRegTransmitPresent(7 downto 1), statePreset)
begin
  case statePreset is
    when idle =>
      shiftRegTransmitNext <= DataTransmit;
    when scl0 =>
      shiftRegTransmitNext <= shiftRegTransmitPresent;
    when scl1 =>
      shiftRegTransmitNext <= '0' & shiftRegTransmitPresent(7 downto 1);
    end case;
end process;
SDO <= shiftRegTransmitNext(0);
receiveShiftRegister : process (SDI, shiftRegReceivePresent,
  shiftRegReceivePresent(6 downto 0), statePreset)
begin
  case statePreset is
    when idle =>
      DataReceive <= shiftRegReceivePresent;
      shiftRegReceiveNext <= shiftRegReceivePresent;
    when scl0 =>
      shiftRegReceiveNext <= shiftRegReceivePresent(6 downto 0) & SDI;
    when scl1 =>
      shiftRegReceiveNext <= shiftRegReceivePresent;
    end case;
end process receiveShiftRegister;
busy <= '0' when statePreset = idle else '1';
end behavioral;

```

#### 6.4 I<sup>2</sup>C (Inter-Integrated-Circuit)

Das I<sup>2</sup>C Interface ist ein bidirektionales Bussystem. Es können beliebig viele Master und Slaves angehängt werden. Die folgenden Signale werden für die I<sup>2</sup>C Schnittstelle benötigt:

- SCL: Taktsignal
- SDA: Bidirektionales Datensignal

##### 6.4.1 Übertragungsgeschwindigkeiten

Die folgenden Geschwindigkeiten sind im I<sup>2</sup>C 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

##### 6.4.2 Timing

Die folgenden Zustände auf dem I<sup>2</sup>C Bus sind möglich:



Eine mögliche I<sup>2</sup>C Transaktion ist untenstehend zu sehen:

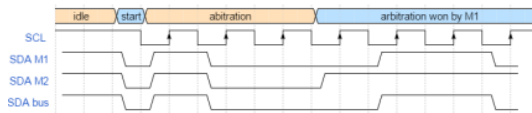


Nach dem Senden des Acknowledge-Bits, kann ein jeweils ein weiteres Byte gesendet werden, ohne die Transaktion durch Senden eines Stopbits zu unterbrechen.

Sollte ein Slave mehr Zeit brauchen um Daten zu verarbeiten, so kann er das SCL Signal auf Low ziehen, bis er wieder bereit ist.

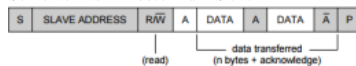
##### 6.4.3 Arbitration

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

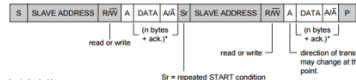


Der letzte Teilnehmer, welcher die SDA Leitung auf Low ziehen kann, gewinnt die Kommunikation.

##### 6.4.4 Kommunikations Protokoll Schreiben vom Master zum Slave

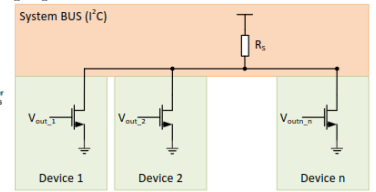


##### Lesen vom Slave



##### 6.4.5 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.



##### 6.4.6 Implementierung in VHDL

Da die beiden I<sup>2</sup>C-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;

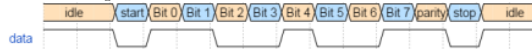
```

##### 6.5 UART (Universal Asynchronous Receiver Transmitter)

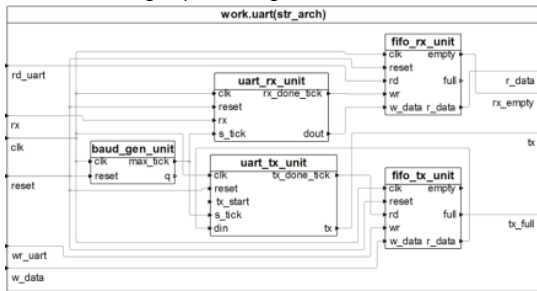
UART ist eine Punkt-zu-Punkt Schnittstelle. Es wird pro Richtung ein Datensignal benötigt.

##### 6.5.1 Timing

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

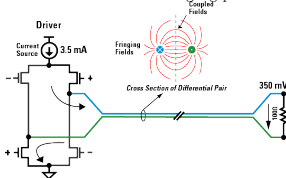


##### 6.5.2 Hardwaremässige Implementierung



##### 6.6 LVDS

LVDS definiert kein Übertragungsprotokoll sondern eine Übertragungsart.



##### 6.6.1 Prinzip

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

##### 6.6.2 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]

```

## 7 Parallel Communication Interfaces

### 7.1 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

### 7.2 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.

### 7.3 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.

#### 7.3.1 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 Arbitrer 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. Es gibt drei Möglichkeiten, wie man die Kanäle teilt bzw.

##### zusammenfasst:

- 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

##### 7.3.2 Varianten

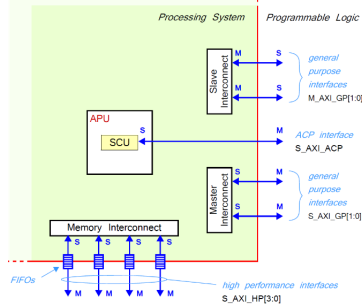
- AXI 4: Maximale Performance. Burst-Modus bis zu 256 Zyklen pro Adresse. Für Hochleistungsdatenaustausch.
- AXI-Lite: Vereinfachte Version von AXI 4. Für speicherabgebildete Einzeltransaktionen. Kein Burst. Für kleinen Datendurchsatz wie Austausch von Steuerungseinstellungen.
- AXI-Stream: Für High-Speed-Streaming.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

##### 7.3.3 Zynq: Processing System ↔ Programmable Logic

Datenübertragung zwischen PS und PL erfolgt via AXI.

Jede Schnittstelle realisiert einen kompletten AXI Bus.

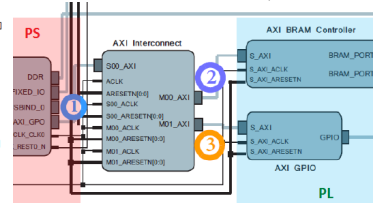
Insgesamt sind 9 AXI Schnittstellen vorhanden.



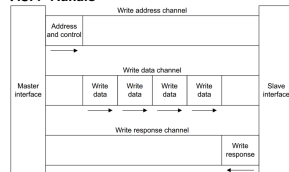
##### AXI Interconnect Block

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)



##### 7.3.4 Kanäle

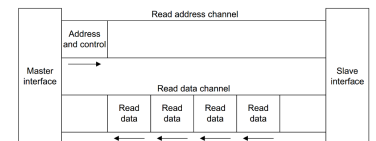


- AW (write address channel) → Master zu Slave
- W (write data channel) → Master zu Slave
- B (write response channel) → Slave zu Master

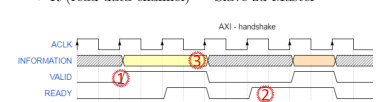
##### 7.3.5 Handshaking

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.



- AR (read address channel) → Master zu Slave
- R (read data channel) → Slave zu Master







```

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) severity error;
wait until (rising_edge(clk_tb)); — Wait for next clock cycle
end loop;
end process;
end architecture;

```

L. Leuenberger, M. Ehrler

(C) BY-NC-SA

16. Februar 2017

```

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
  - Strukturierte Simulation
  - Verifiziert die synthetisierte Netzliste
  - Timing möglich, jedoch ungewöhnlich (da kein Routing)
- Post Implementation Simulation
  - Funktionelle und Timing Simulation möglich
  - Präziseste, jedoch langsamste

#### 9.4 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:

- Fehler sensibilisierung (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.

##### 9.4.1 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.

##### 9.4.2 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 unprogrammierbaren 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:

##### 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.

##### 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.

## 10 Verification Design For Test

### 10.1 Warum testen?

In der Fabrikation von elektronischen Systemen gibt es unzählige Fehlerquellen. Je früher ein Fehler erkannt 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.

### 10.2 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.

L. Leuenberger, M. Ehrler

(C) BY-NC-SA

16. Februar 2017

- **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.

### 10.3 Fehlererkennung

#### 10.3.1 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.

#### 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	-

#### 10.3.2 Fehlerabdeckung

Die Fehlerabdeckung kann mit der folgenden Formel berechnet werden:

Fehlerabdeckung =  $\frac{\text{Anzahl detektierter Fehler}}{\text{Anzahl möglicher Fehler}} \cdot 100\%$

Im Allgemeinen sollte darauf geachtet werden, dass eine Fehlerabdeckung von mindestens 95% erreicht wird.

#### 10.3.3 Redundante Logik

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

#### 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	K4: 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	-

#### 10.4 IDDQ-Test

L. Leuenberger, M. Ehrler

(C) BY-NC-SA

16. Februar 2017

Beim IDDQ-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.

### 10.5 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 geschaltet (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.

#### 10.5.1 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.

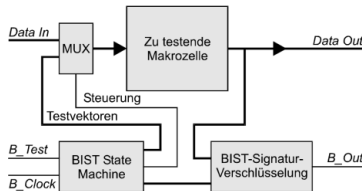
#### 10.5.2 Scan Path bei mehreren Taktdomänen

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

### 10.6 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.

#### 10.6.1 Prinzip



Obenstehend 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 Teststengenerator enthält, wird dieser Multiplexer so geschaltet, dass entweder von aussen kommende Daten an die Makrozelle geführt werden (Normalbetrieb), oder solche vom Teststengenerator (Testbetrieb). Die Ausgänge der Makrozelle werden neben dem normalen Verdrahtung dem Block BIST-Signatur-Verschlüsselung zugeführt. Dieser wertet die Testergebnisse aus und meldet sie nach aussen.

#### 10.7 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.

L. Leuenberger, M. Ehrler

(C) BY-NC-SA

16. Februar 2017

### 10.7.1 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 zusammengeschaltet (BSR - Boundary Scan Register).

#### 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

### 10.7.2 Realisierung auf dem Chip

Jeder Chip muss für BST die folgenden Elemente beinhalten:

- **Bypass-Register**: Überbrückt das 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 um z.B. auf korrekte Bestückung zu prüfen.
- **TAP (Test Access Port)**: Schnittstelle nach Aussen
- **Instruction-Register**: Steuert die internen notwendigen Multiplexer Einstellungen.

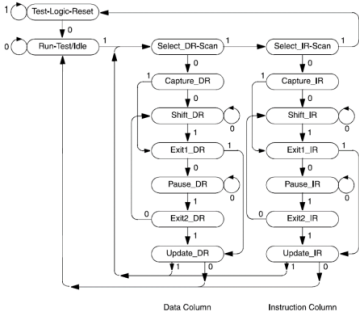
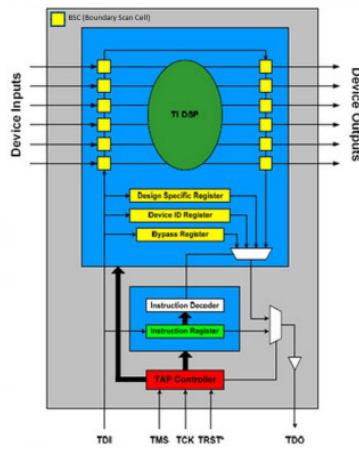
### 10.7.3 TAP-Kontrolller

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.

### 10.7.4 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-Kontrolller sich im Zustand *Shift\_IR* befindet, wird die Instruktion seriell vom Eingang *TDI* in das Instruktions-Register geladen.



### 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 geschaltet.
- **EXTEST**: Der Befehlscode besteht aus lauter Nullen (00...0). Die Chiplogik wird bei diesem Befehl von der Leiterplatte isoliert (Die Kernlogik muss so abgesichert sein, dass sie nicht beschädigt werden kann). Das

L. Leuenberger, M. Ehrler

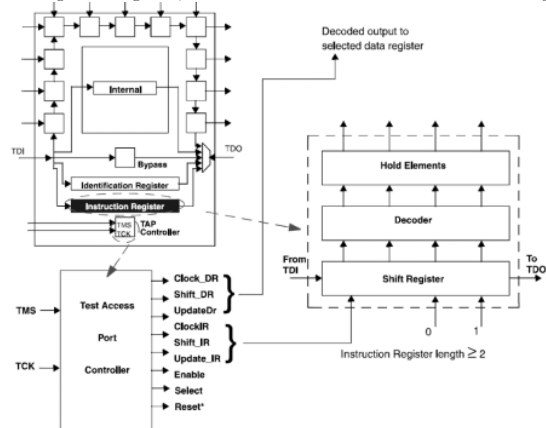
(C) BY-NC-SA

16. Februar 2017

BSR wird im Zustand *Shift\_DR* bei diesem Befehl in den *TDI/TDO* Pfad geschaltet.

### Interner Aufbau

Nachfolgend ist dargestellt, wie der TAP-Kontrolller intern mit den vorhandenen Registern verschaltet ist.

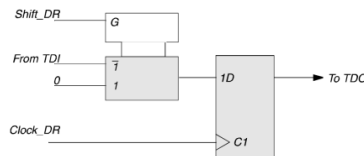


### Optionale Befehle

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

- **INTTEST**: 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 *INTTEST*) 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.

### 10.7.5 Aufbau BYPASS-Register

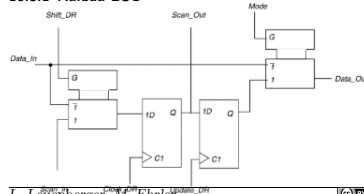


### 10.8 Aufbau ID-Register

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

### 10.8.1 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.

L. Leuenberger, M. Ehrler

(C) BY-NC-SA

16. Februar 2017

## 11 Digital Design For ASICs

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

### 11.1 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.

#### Gesamtkosten pro Einheit

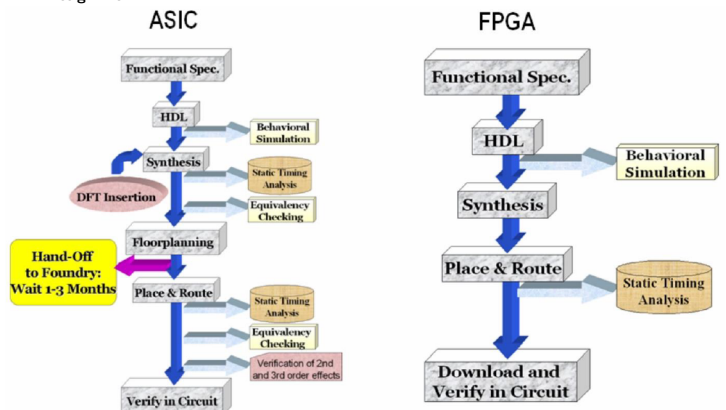
$c = \frac{a_0}{n} + c_1$   $c_0$ : Summe der einmaligen Entwicklungskosten → konstant //  $n$ : Anzahl der entwickelten ASICs //  $c_1$ : Kosteninkrement pro erzeugter Schaltung (für Rohwafer, Waferbearbeitung, Prüfung, Verpackung, Logistik usw.) → linear steigend

#### $c_0$ ist 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.

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.

### 11.2 Design Flow



#### 11.2.1 Synthese (FPGA vs. ASIC)

##### FPGA

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

L. Leuenberger, M. Ehrler

(C) BY-NC-SA

16. Februar 2017

## ASIC

- Wahl des Designhouse: ASIC-Design erfordert Wissen, Erfahrung und Infrastruktur, welche nicht immer vorhanden ist → evtl. Designhouse
- Wahl des Synthesetools (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 generischer 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 Zellenbibliothek 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.

### 11.2.2 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).

### 11.2.3 Place and Route (FPGA vs. ASIC)

#### 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

#### ASIC

Besitzt mehrere Teilschritte:

- Power Planning: Generiert den Powering, 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 steuern. 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 Füllzellen gefüllt, um "Löcher" in den Zellen zu schliessen. Dies ist notwendig, um kontinuierliche p- und n-Wannen entlang einer ganzen Standardzellenreihe zu haben.

### 11.2.4 Verification FPGA vs. ASIC

#### FPGA

Verifikation gleich nach heruntergeladen der Programmdatei möglich. Man muss nicht auf einen Prototypen warten.

#### ASIC

Verifikation nicht gleich möglich. Folgende Schritte müssen VOR der Verifikation vollzogen werden:

- Sign-off: Daten der Produktion übermitteln und prüfen lassen
- Generierung der Maske
- Waferbearbeitung
- Wafer tests, Wafer Dicing (schneiden), Verpacken

L. Leuenberger, M. Ehrler

(C) BY-NC-SA

16. Februar 2017