

Digital Microelectronics

Zusammenfassung DigMe HS2015

Autoren

Dominik Büsser

HSR Hochschule für Technik Rapperswil

1. Februar 2016

Inhaltsverzeichnis

1	How to Design a System on Chip	6
1.1	Einführung	6
1.1.1	Was ist ein System on Chip (SoC)?	6
1.1.2	Herausforderungen des Designs	7
1.1.3	Lösungen zur Meisterung der Herausforderungen	7
1.1.4	IP basiertes Design	8
1.2	Designfluss	8
1.3	Design Tools	9
1.4	Lernziele	10
2	System on Chip Design Hardware	12
2.1	Übersicht über die Architekturen	12
2.2	Elemente auf einem FPGA (Zynq)	12
2.2.1	Processing System im Detail	12
2.2.2	Programmierbare Logik im Detail	13
2.2.3	Clocking-System	14
2.3	Design Constraints	16
2.4	Lernziele	16
3	System on Chip Design Using IP	18
3.1	Einführung	18
3.2	Design-Einheiten und Subprogramme	18
3.2.1	Funktionen (function)	18
3.2.2	Prozeduren (procedure)	19
3.2.3	Packages	19
3.2.4	Distributed RAM und Block RAM	20
3.3	Xilinx IP-Katalog	21
3.3.1	Übersicht	21
3.3.2	XILINX IP Block Memory Generator	22
3.3.3	Weitere IP-Blöcke	23
3.3.4	Schnittstellen	23
3.4	Integration eines IP Blocks	24
3.5	Lernziele	24
4	Custom IP Block Designs	26
4.1	Systemdesign	26
4.1.1	Genereller Ansatz	26
4.2	Hardware / Software Partitionierung	27
4.3	Genereller Designflow	27
4.3.1	Softwaredesign für ein System on Chip	28
4.4	Entwicklung eines IP Blocks	28
4.4.1	Out-of-Context- vs. Globaler Synthese	29

4.5	IP Block Kreation	29
4.6	Lebenszyklus eines IP-Blocks	30
4.7	Lernziele	31
5	Verifying your SoC	34
5.1	Verifikation des Projekts	34
5.1.1	Verifikation während der Spezifikationsphase	34
5.1.2	Verifikation während der Designphase	34
5.2	Lernziele	36
6	Verifying your SoC II	38
6.1	Lernziele	38
7	IP Block Design using HLS	41
7.1	Lernziele	41
8	Modelling of digital Filters	45
8.1	Lernziele	45
9	SoC ASIC Design	48
9.1	Lernziele	48

Einleitung

Diese Zusammenfassung entspricht der Vorlesung "Digital Microelectronics" im Herbstsemester 2015. Die Zusammenfassung wurde aus den Folien von der Vorlesung zusammengestellt. Der Autor dieser Folien ist Dr. Paul Zbinden. Sämtliche Informationen Textabschnitte, und Bilder stammen aus diesen Folien.

1 How to Design a System on Chip

1.1 Einführung

Computing everywhere	Grundsätzlich sind zwei Trends zu beobachten: Überall sind Computer enthalten bzw. werden Computer benötigt. Diese müssen klein, stromsparend und möglichst ins System integriert sein.
Internet of things	Integrierte intelligente low power Systeme Wikipedia: Der Begriff Internet of Things (IoT) beschreibt, dass der (Personal) Computer zunehmend als Gerät verschwindet und durch "intelligente Gegenstände" ersetzt wird. Statt - wie derzeit - selbst Gegenstand der menschlichen Aufmerksamkeit zu sein, soll das IoT den Menschen bei seinen Tätigkeiten unmerklich unterstützen.
Sicherheit gefragt	Immer mehr kommt auch die Sicherheit ins Spiel. Daten müssen geschützt werden, gleichzeitig soll alles möglichst vernetzt sein.
Time to Market	Die Systeme werden immer komplexer und der Konkurrenzkampf grösser. Deshalb müssen neue Produkte schnell auf den Markt gebracht werden. Komplexe Systeme werden deshalb aus bestehenden und neu implementierten Komponenten entwickelt. → Neue Designs sind Systeme anstatt nur eine Schaltung.

1.1.1 Was ist ein System on Chip (SoC)?

System on Chip	Ein System on Chip ist ein komplettes System auf einem Chip und beinhaltet meistens folgende Komponenten: • Mikroprozessor • Speicher • Peripherien • Anwendungsspezifische Blöcke (IP-Blöcke) Weiter können SoCs Arbeitsspeicher, Bussysteme und Schnittstellen enthalten.
Vorteile	Vorteile sind: • Kürzere Entwicklungszeit • Kleinere Entwicklungskosten • Kleinere Kosten für das Endprodukt • Flexibler im Betrieb (Updates, etc.)

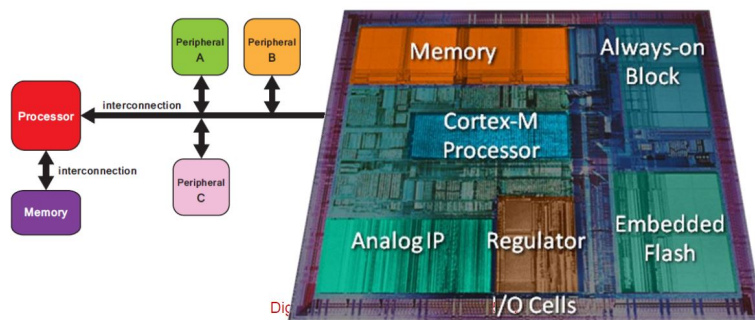


Abbildung 1.1: Möglicher Aufbau eines System on Chip

1.1.2 Herausforderungen des Designs

Die Komplexität der Systeme nimmt stetig zu. Diese entsteht unter anderem durch den hohen Grad von Parallelität auf verschiedenen Ebenen	Komplexität
Unter Heterogenität versteht man in der Informationstechnik die Zusammenarbeit verschiedenen Softwaresystemen. Probleme entstehen durch die verschiedenen Tools, Hardware-Beschreibungssprachen, Programmiersprachen und Anbieter.	Heterogenität
Der Konkurrenzkampf ist hoch. Systeme müssen möglichst schnell auf den Markt gebracht werden.	Time to Market
Da die Systeme immer grösser werden, werden auch die Anforderungen an den FPGA grösser:	Herausforderungen FPGA
• Mehr Funktionalität gefragt (DSP slices, Mikroprozessor, etc.) • Clock-Planung wird durch die grössere und hohe Performance sehr schwierig • Timing innerhalb FPGA ist ebenfalls äusserst schwierig	

1.1.3 Lösungen zur Meisterung der Herausforderungen

Bereits vorhandene Komponenten wiederverwenden oder bestehende IP-Blöcke einkaufen.	Re-use
Das System mit mehreren Hierarchiestufen abstrahieren. Dies sowohl bei der Hardware, als auch bei der Software.	Abstraktion des Systems
Die Hardware und Software eines Systems sollen möglichst parallel zueinander entwickelt werden.	Parallele Entwicklung
Gutes (und genügend grosses) Team und geeignete Entwicklungswerkzeuge einsetzen.	Teams und Tools

1.1.4 IP basiertes Design

Definition IP	Intellectual Property IP (geistiges Eigentum) ist eine wiederverwendbare logische Einheit, Zelle oder Chip-Layout einer Partei (Mensch, Firma). Parametrisierte Komponenten erleichtern die High-Level-Synthese.
IP-Typen	IP-Typen sind: • Hard IP Cores: Optimiertes, voll funktionsfähiges System (Blackbox) • Firm IP Cores: Synthetisierbare Netzliste, die simuliert und bei Bedarf angepasst werden kann. • Soft IP Cors: RTL (Register Transfer Level) Designs, die der Nutzer synthetisieren muss.
Quellen IP-Blöcke	IP-Blöcke können von folgenden Quellen stammen: • Bestehende, selbst entwickelte Blöcke • Neue, selbst entwickelte Blöcke • Bestehende Blöcke einer anderen Partei (third party IP), die man einkauft

1.2 Designfluss

Design eines SoC	Grober Ablauf beim Design eines SoC: 1. System abstrahieren 2. Aufteilung der Funktionalität zwischen Hardware und Software, definieren der Schnittstellen 3. Entwicklung und Tests der Hardware/Software 4. Systemintegration (wird oftmals unterschätzt)
------------------	---

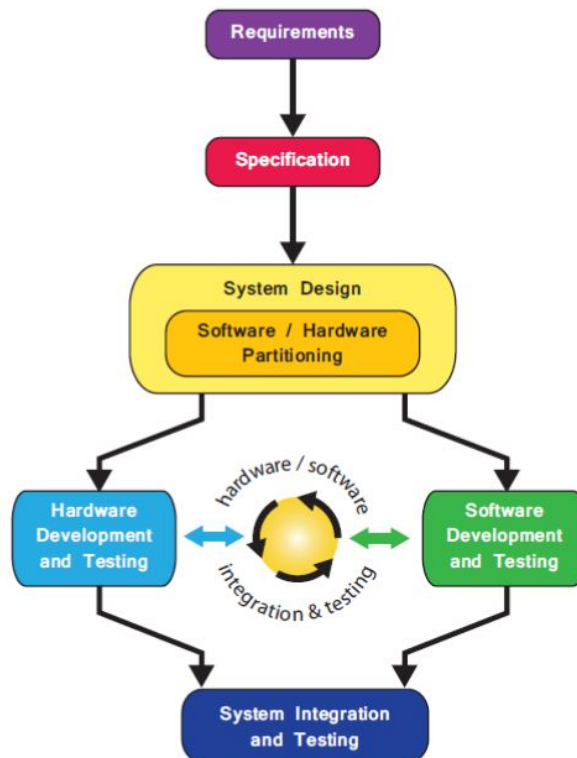


Abbildung 1.2: Designfluss eines System on Chip

1.3 Design Tools

Vivado IDE Solution ist eine hierarchisch aufgebaute Entwicklungs- und Analysierungs- Vivado IDE
Umgebung: Solution

- Design und Analyse (Timing, Konnektivität, Ressourcenverbrauch, etc.)
- RTL Entwicklung und Analyse (Schemas, etc.)
- Simulation
- Synthese und Implementierung
- I/O-Pins planen

Xilinx Software Development Kit ist eine Entwicklungsumgebung für Softwareent- XSDK
wicklung.

High-Level Synthese HLS generiert aus einem C-Code eine RTL Implementierung.

Vivado Design Flow :

High-Level
Synthese
Vivado Design
Flow

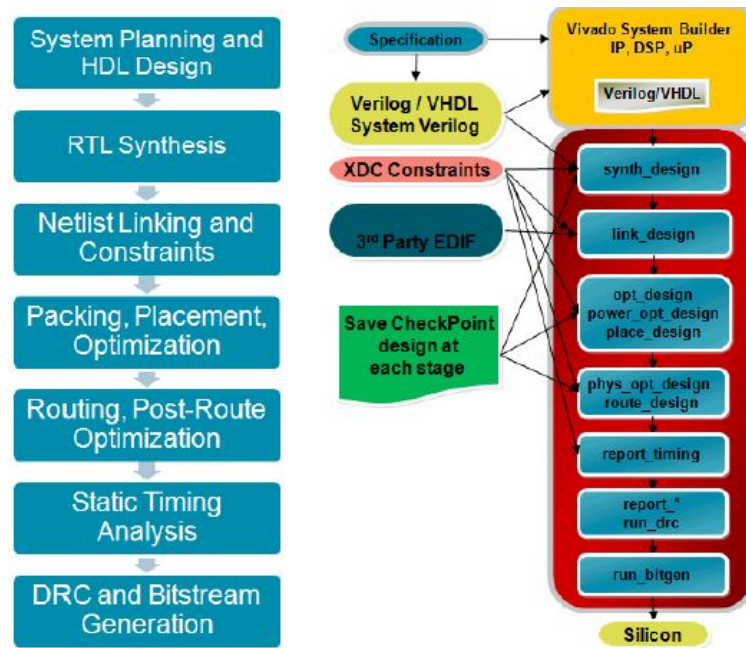


Abbildung 1.3: Vivado Design Flow

1.4 Lernziele

1. Beschreibe die Hauptcharakteristiken eines SoC!
Komplettes System auf einem Chip, bestehend aus Mikroprozessoren, Speicher, Peripherien, IPs.
2. Erkläre die grössten Herausforderungen von einem SoC-Design!
 - Systeme werden immer Komplexer (Komplexität)
 - Verschiedene Anbieter/Beschreibungssprachen und parallel laufende Systemen machen Zusammenarbeit schwierig (Heterogenität)
 - Systemen müssen aufgrund Konkurrenzkampf schnell auf den Markt gebracht werden (Time to Market)
3. Liste die wichtigsten Eigenschaften des Vivado IDE auf!
 - Gemeinsames Datenmodell durch den Gesamten Design Flow
 - Berichte in allen Designstufen
 - In jeder Designstufe kann ein Checkpoint gespeichert werden
4. Beschreibe den generellen SoC-Design-Fluss und der Vivado IDE Design Fluss

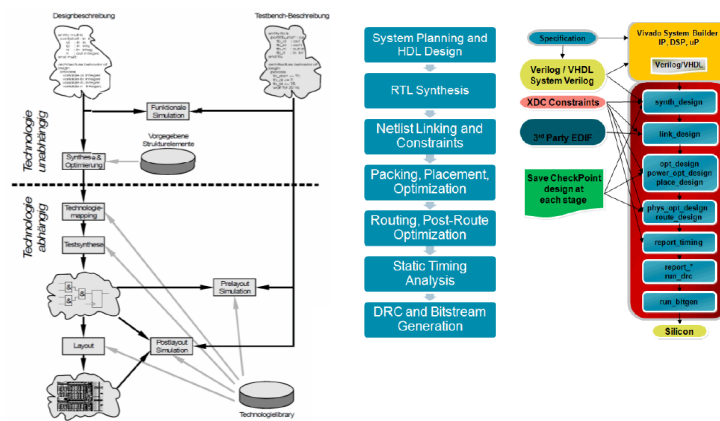


Abbildung 1.4: links: Traditioneller Design Flow, rechts : Vivado Design Flow

2 System on Chip Design Hardware

2.1 Übersicht über die Architekturen

Hauptbestandteile
Zynq-7000

Hauptbestandteile des Zynq-7000:

- Processing System (PS):
Enthält einen Dual ARM Cortex-A9 Prozessor mit mehreren Peripherien und einem hard silicon core
- Programmierbare Logik (PL):
Artix-7 oder Kintex-7 FPGA

Weiter Enthält der Zynq-7000 Komponenten zur Signalverarbeitung (2 ADCs, On-Chip Temperatur- und Spannungssensor)

2.2 Elemente auf einem FPGA (Zynq)

Aufbau Zynq-7000

2.2.1 Processing System im Detail

Zynq-7000 AP SoC:

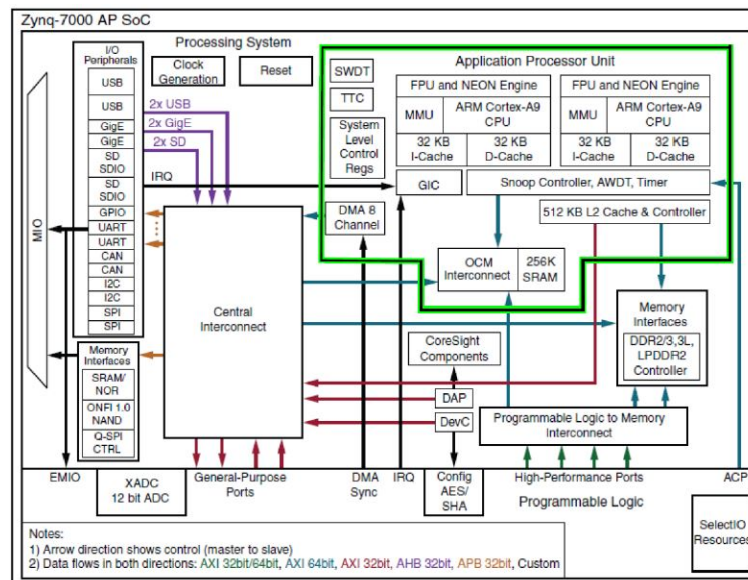


Abbildung 2.1: Aufbau Zynq 7000 AP SoC

Das Verarbeitungssystem des Zynq-7000 enthält nicht nur den ARM Prozessor, sondern ein Set von Verarbeitungsressourcen. Diese ergeben die Application Processor

Unit APU. Nebst einem Cortex-A9 Prozessor enthält sie zudem schnelle Puffer-Speicher (caches) und weitere Support-Blöcke. Das System arbeitet mit bis zu 1 GHz und hat einen vollständig integrierter Speichercontroller.

Application
Processor Unit
APU

Das PS enthält mehrere Schnittstellen. Es gibt Schnittstellen zwischen dem Processing System und der Programmierbaren Logik sowie Schnittstellen für I/O-Peripherien.

Externes Interface

2.2.2 Programmierbare Logik im Detail

CLBs (konfigurierbare Logikblöcke) sind kleine, reguläre Gruppen von Logikelementen, die in einer zweidimensionalen Anordnung auf der PL festgelegt sind und mit gleichen Ressourcen über programmierbare Verbindungen verbunden sind. Jedes CLB ist neben einer Switch Matrix und beinhaltet zwei Slices.

Konfigurierbare
Logikblöcke

Ein Slice ist ein Bestandteil des CLB und beinhaltet 4 Lookup-Tabellen, 8 Flipflops und andere Logik. Es gibt zwei Arten von Slices:

Slice

- SLICEM: Full slice
LUT kann für Logik und Speicher gebraucht werden
Hat Multiplexer und eine Carry Chain
- SLICEL : Nur für Logik und Arithmetik
hat ebenfalls eine carry chain

Ein CLB wird über eine Switch Matrix mit anderen FPGA-Ressourcen verbunden.

Switch Matrix

Input/Output Blocks (IOBs) sind Ressourcen, die eine Schnittstelle zwischen PL und physikalischen Geräten anbieten.

Input/Output
Blocks

Die Low-Voltage Differential Signaling (LVDS) gehört ebenfalls zu den IOBs. LVDS ist ein Schnittstellen-Standard für Hochgeschwindigkeits-Datenübertragung. Wichtige Merkmale sind:

LVDS

- Differenzielle Spannungspegel
- Relativ geringe Spannungspegel (low voltage)
- Die Signale werden mit einer Konstantstromquelle erzeugt

Werden z.B. bei Sigma/Delta FPGA angewendet.

Es gibt zwei verschiedene Typen von I/O in der 7er-Serie:

I/O Typen

- High Range (HR)
Unterstützt Spannungen bis 3.3 Volt
- Hight Performance (HP)
Unterstützt nur Spannungen bis 1.8 Volt, dafür höchste Performance

Spezielle Ressourcen sind

Spezielle
Ressourcen

- Block RAM
Alle Operationen werden synchron ausgeführt, die Ausgänge sind latched. (Siehe Folien 23-25)
- DSP48E1s (für High-Speed-Arithmetik)

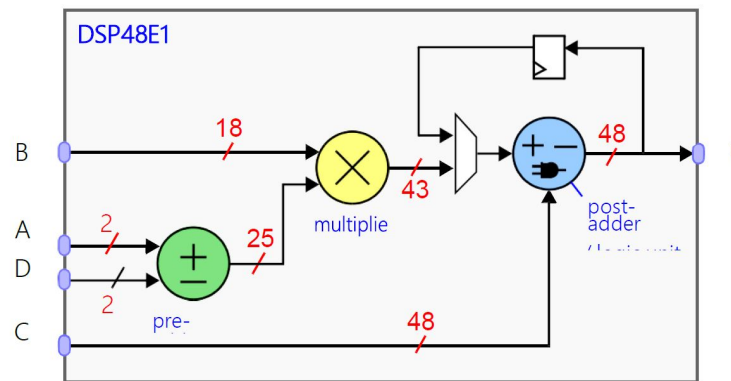


Abbildung 2.2: Aufbau eines DSP48E1 Slice

2.2.3 Clocking-System

Das Clocking-System eines FPGAs ist sehr anspruchsvoll. Es hat die Aufgabe, ein System zu synchronisieren, indem es den Takt angibt. Idealerweise schalten alle Flipflops zum selben Zeitpunkt.

Clock Distribution Delay

In der Realität schalten die Flipflops zu leicht unterschiedlichen Zeiten. Diese unterschiedlichen Delays entstehen durch die Drivers und die Leitungen des Clockverteilungs-Netzwerk. Der Delay vom Clockimpuls bis zum Schaltzeitpunkt eines Flipflops (bzw. Zustandstransition) nennt man *clock distribution delay* t_{di} .

Clock-Skew

Der *clock distribution delay* ist von Flipflop zu Flipflop unterschiedlich. Die Differenz des *clock distribution delay* zweier unterschiedlichen Flipflops nennt man den Clock-Skew t_{sk} .

$$t_{sk} = |t_{di1} - t_{di2}| \quad (2.1)$$

Massgebend ist vor allem die Zeitdifferenz vom Schaltzeitpunkt des ersten und Schaltzeitpunkt des letzten Flipflops, also $\max(t_{sk})$

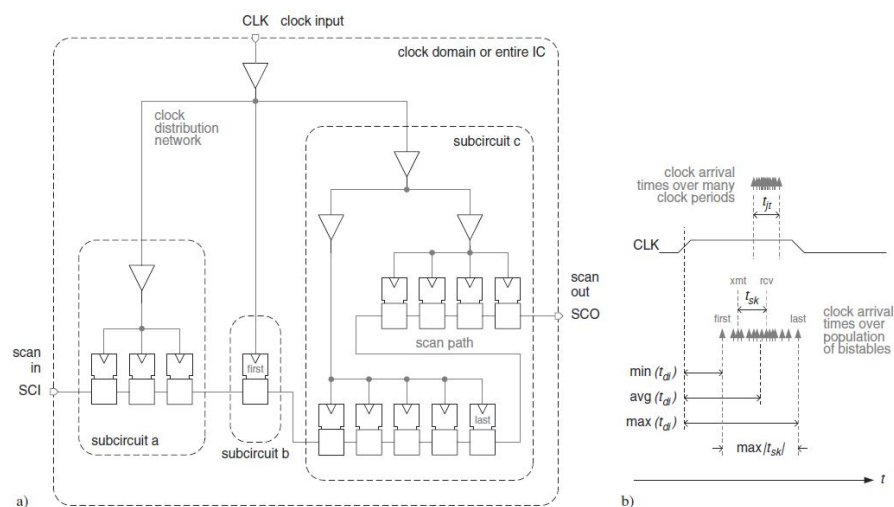


Abbildung 2.3: Clock Distribution. Clockverteilungssystem (a) und Timing-Problem (b)

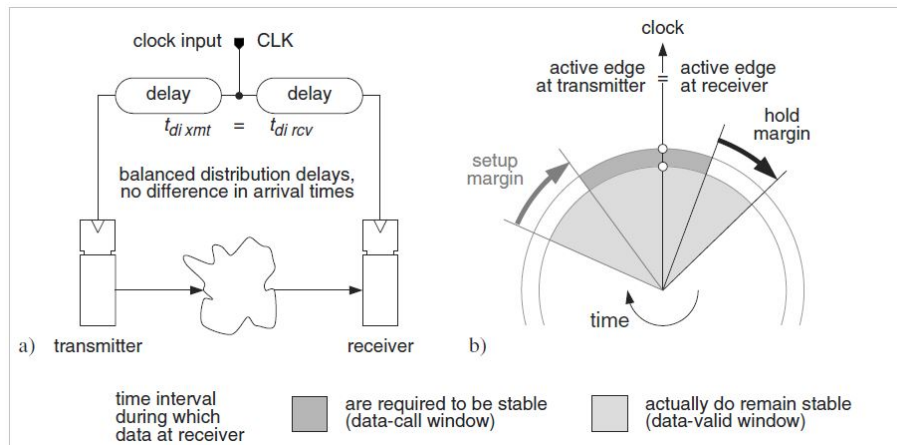


Abbildung 2.4: Delay ist überall gleich, deshalb resultiert kein Skew

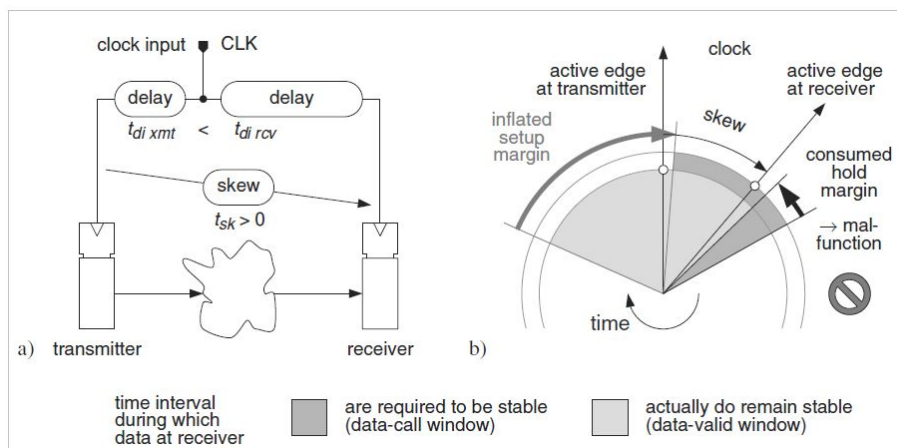


Abbildung 2.5: Delay ist unterschiedlich, es resultiert ein Skew

Zusätzlich zum Clock-Skew variieren die Ankunftszeiten eines Clockimpulses am selben Ort (z.B. Flipflop). Dies nennt man den Clock-Jitter.

Diese Delays können Timing-Probleme verursachen. Die Differenz zwischen der vorgeschriebenen Zeit $t_{required}$ und der tatsächlichen Ankunftszeit (benötigte Zeit) $t_{arriving}$ nennt man Slack. Ist der Slack negativ (benötigte Zeit ist grösser als vorgeschriebene Zeit) hat man ein Timing-Problem.

$$t_{Slack} = t_{required} - t_{arriving} \quad (2.2)$$

Damit die anspruchsvolle Aufgabe des Clockmanagements zuverlässig funktioniert, haben die 7er-Reihe von Xilinx mehrere spezielle Ressourcen:

- Globale Clock Buffers
- Low-skew Clockverteilung (regionales Clock-Routing)
- Unterschiedliche Clock-Regionen

7-Series FPGA
Clock
Management

- Clock Management Tile (CMT):
Mixed-Mode Clock Managers (MMCM) und Phase Locked Loop

2.3 Design Constraints

Jedes Projekt bzw. jede Hardware stellt unterschiedliche Anforderungen an die Plattform (Timing, Ressourcenbedarf, etc.). Solche Hardware Optionen müssen für jedes Projekt unterschiedlich gesetzt werden. Die Design Constraints geben diese Möglichkeit. Es gibt zwei Typen von Constraints:

Synthese-Constraints

Die Synthese-Constraints (synthesis constraints) bieten ein Werkzeug für Optimierungen in der Synthese. Die Constraints können im VHDL-Code eingebettet werden, oder in einem separaten constraint-File gemacht werden.

Implementation-Constraints

Die Implementation-Constraints (implementation constraints) bieten ein Werkzeug zur Einflussnahme auf das Mapping, die Platzierung, das Timing oder andere Richtlinien. Diese Constraints werden in einem .xdc-File definiert.

XDC-Constraints

Die XDC-Constraints basieren auf dem Synopsys Design Constraints (SDC) Format. Die Constraints können folgendermassen gesetzt werden:

- Über ein interaktives GUI
- Eingeben mit Tcl-Skript über die Tcl-Konsole
- Gespeichert in einem oder mehreren .xdc-Files, die dem Projekt hinzugefügt werden. Die Ausnahme bilden RTL-Constraints.

Die physikalischen Constraints (Pins, etc.) und die Timing-Constraints sollten in zwei separaten .xdc-Files gespeichert werden.

2.4 Lernziele

1. Beschreibe die generelle Architektur eines Zynq SoC.
Ein Zynq SoC besteht aus einem Processing System (PS) und einer programmierbarer Logik (PL). Weiter enthält 2 ADCs und on-Chip-Sensoren. Die externe Kommunikation wird über verschiedene I/Os gemacht, die interne Kommunikation geschieht über den AXI-Bus.
2. Beschreibe die Slice-Ressourcen, die auf den Xilinx 7-Series FPGAs vorhanden sind.
Ein Slice besteht aus folgenden Komponenten:
 - 4 6-Input Look-Up-Tables
 - Multiplexer
 - Carry Chain (Multiplexer, XOR)
 - Insgesamt 8 Flipflops
3. Führe die verschiedenen Pads und die special resources auf einem FPGA auf und beschreibe diese. Pads und I/Os:

- Standard-I/Os mit diversen Optionen (single ended, differentiell, VCC, GND, Assigned, Unassigned, Clock-tauglich)
- LVDS
- I/O: High Range (HR)
- I/O: High Performance (HP)

Spezielle Ressourcen:

- 7-Series Block RAM and FIFO
- DSP48E1 Slice
- High quality Analogschnittstelle (XADC)
- Analog Mixed Signal (AMS)
- Jeder Clock mit 1 Mixed-Mode Clock Manager (MMCMs) und 1 Phase Locked Loop (PLL)

4. Syntax von Design-Constraints, und Definition von einem Clock:

Beispiel eines Clocks mit $T = 10ns$ (100MHz) und 50% - Duty Cycle:

```
create_clock -period 10.000 -name clk_pin -waveform { 0.000 5.000} [get_ports  
clk_pin]
```

5. Interpretiere die Timing-Informationen, die in einem Timing-Report gegeben sind.

3 System on Chip Design Using IP

3.1 Einführung

Ziel von IP	Das Ziel von <i>intellectual property</i> (IP), bzw. mit IP-Blöcken ist, dass man die Komplexität von den heutigen Systemen meistern kann und gleichzeitig konkurrenzfähig bleibt.
Ziel: Komplexität meistern	Komplexität von heutigen Systemen meistern, indem man geeignete Komponenten und Architektur verwendet: • Wiederverwendung von geprüften IP • Hierarchische Architektur • Hardware und Software in verschiedene Teile trennen
Mit heterogenen Systemen umgehen können	Systeme müssen oftmals mit anderen Systemen umgehen können, bzw. nebeneinander oder miteinander laufen. Die Schnittstellen müssen deshalb gut aufeinander abgestimmt werden.
Time-to-Market	Systeme müssen möglichst schnell auf den Markt gebracht werden, deshalb gilt: • Wiederverwendung von IPs • Vorhandene IPs weiterentwickeln oder anpassen • Nur das neu Entwickeln, was wirklich nötig ist. • High-Level Design wenn möglich

3.2 Design-Einheiten und Subprogramme

3.2.1 Funktionen (function)

Funktionen	Funktionen (functions) sind kleine Unterprogramme, denen man verschiedene Parameter mitgeben kann. Eine Funktion kann nur einen Wert zurückgeben.
Hierarchische Struktur, Wiederverwendung	Funktionen ermöglichen, eine hierarchische Struktur in den VHDL-Code zu bringen. Gleichzeitig ermöglichen sie, dass man den geschriebenen Code in einem anderen Projekt wieder verwenden kann.
Syntax einer Funktion	Der Syntax einer Funktion lautet folgendermassen:

```

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>;

```

Abbildung 3.1: Syntax von einer Funktion

3.2.2 Prozeduren (procedure)

Prozeduren (procedures) sind kleine Unterprogramme, denen man verschiedene Parameter mitgeben kann. Im Gegensatz zu den Funktionen können Prozeduren mehrere Werte zurückgeben.

Prozeduren

Bei der Synthese gibt es mit Prozeduren oft Probleme. Man sollte besser Funktionen anstatt Prozeduren verwenden.

Probleme bei Synthese

Der Syntax einer Prozedur lautet folgendermassen:

Syntax einer Prozedur

```

procedure <procedure_name>
    [( < argument list with type declaration>)] is
    [<declarations>];
begin
    (sequential statements>);
end [procedure] <procedure_name>;

```

Abbildung 3.2: Syntax von einer Prozedur

3.2.3 Packages

In Packages können Konstanten, Typen, Komponenten, Funktionen und Prozeduren definiert werden, die in einem oder mehreren Projekten platziert werden.

Packages

Somit kann Files übersichtlicher gestalten (Komplexität meistern) und der bereits geschriebene Code wiederverwenden.

Komplexität meistern, Wiederverwendung

Die Packages werden in kompilierter Form in einer Bibliothek gespeichert (standard = work library). Sie können mit einem *use*-Statement einem Projekt hinzugefügt werden:

Gebrauch von Packages

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

Package-Declaration	In der Package-Declaration werden die Konstanten, Funktionen, etc. deklariert (bzw. instantiiert).
Package-Body	Im Package-Body werden die Funktionen, etc. programmiert.
Syntax eines Packages	Der Syntax eines Packages lautet folgendermassen:

```

Package Declaration:    [<library und use statements>]
                        package <package_name> is
                          <declarations>;
                        end <package_name>;

Package Body:          [<library und use statements>]
                        package body <package_name> is
                          <list of definitions>;
                        end <package_name>;

```

Abbildung 3.3: Syntax von einem Package

3.2.4 Distributed RAM und Block RAM

Grundeigenschaften Distributed RAM	Die des verteilten RAM (distributed RAM) sind: • Werte werden in LUTs gespeichert • Schreiben: synchron • Lesen : asynchron • Grosse Felder • schneller Zugriff nur für kleine Strukturen
Definition in VHDL	Der untenstehende Code wird automatisch in distributed RAM synthetisiert:

```

architecture behavioral of ram_test is
  type ram_type is array (0 to 1023) of std_ulogic_vector(15 downto 0);
  signal dist_ram : ram_type := (
    x"ffff", x"fffe", x"fffd", x"fffc", x"fffb", x"fffa", x"fff9", x"fff8",
    others => x"0000");
begin
  async_DistRAM : process(clk)
  begin
    if rising_edge(clk) then
      if en = '1' then
        if we = '1' then
          dist_ram(to_integer(unsigned(addr))) <= din;
        end if;
      end if;
    end if;
  end process async_DistRAM;
  dout_dist <= dist_ram(to_integer(unsigned(addr))); -- asynchronous read operation
end behavioral;

```

Wortbreite
Anzahl Worte

Grundeigenschaften Block RAM	Die des Block RAM (block RAM) sind: • Werte werden in RAM Blocks mit jeweils 36kB Speicherplatz gespeichert • Sowohl Schreibe-, als auch Lesezugriffe erfolgen synchron
------------------------------	--

- Optimierte Felder
- schneller Zugriff nur für grössere Strukturen

Der untenstehende Code wird automatisch in Block RAM synthetisiert:

Definition in
VHDL

```
architecture behavioral of ram_test is
    type ram_type is array (0 to 1023) of std_ulogic_vector(15 downto 0);
    signal bram: ram_type := ( -- initializes a few ram cells
        x"ffff", x"ffff", x"ffff", x"ffff", x"ffff", x"ffff", x"ffff", others => x"0000");
begin
    sync_BRAM : process(clk)
    begin
        if rising_edge(clk) then
            if en = '1' then -- if RAM is enabled
                if we = '1' then -- if write is enabled
                    bram(to_integer(unsigned(addr))) <= din; -- store din in RAM cell
                    dout_block <= din; -- read value back
                else
                    dout_block <= bram(to_integer(unsigned(addr))); -- synchronous read
                end if;
            end if;
        end if;
    end process sync_BRAM; end behavioral;
```

Der RAM-Style kann auch mit Attributen definiert werden:

RAM-Style mit
Attributen

```
Attr. value for distrib. RAM  attribute ram_style of attr_ram : signal is "distributed";
Attr. value for block RAM    attribute ram_style of attr_ram : signal is "block";
```

3.3 Xilinx IP-Katalog

3.3.1 Übersicht

Die IP-Module (oder IP-Blöcke) werden zentralisiert in einem Repository gespeichert/verwaltet. Die Quellen können von unterschiedlicher Herkunft sein:

- XILINX IP (von IP-Katalog)
- 3rd Party IP (eingekaufte IP-Module)
- Custom IP (in-house):
IP-Blöcke, die selber entwickelt worden sind, werden ebenfalls im Repository abgespeichert. Der geschriebene Code (VHDL, C-Code) wird über den *IP-Packager* in ein IP-Modul umgewandelt, sodass er als fertiger Block eingefügt werden kann (siehe Abb 3.4).

Das IP-Modul kann somit auch in unterschiedlicher Form vorliegen:

- Quellcode (source code)
- Verschlüsselter Code (encrypted)
- Etwas zwischen den beiden oben genannten Punkten

Eigenschaften des IP Katalog:

Eigenschaften

- Wird durch den IP Integrator (IPI) unterstützt

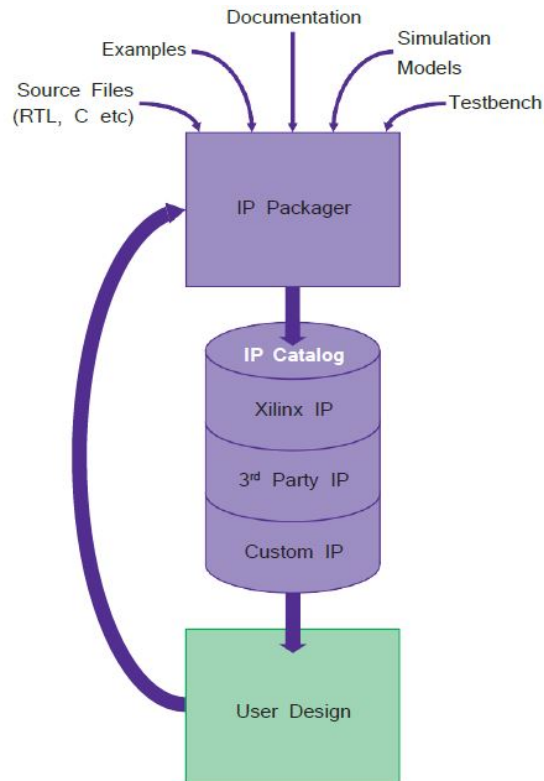


Abbildung 3.4: Vivado IP-centricflow

- Die meisten IPs sind gratis erhältlich, einige wenige benötigen Lizenzen
- Werden als Quellcode gespeichert (install dir), teils ist der Quellcode jedoch auch verschlüsselt
- Wurden immer mit dem neuesten Tool synthetisiert

3.3.2 XILINX IP Block Memory Generator

Der XILINX IP Block Memory Generator generiert Speicher von willkürlich Grösse und Tiefe mit verschiedenen Schnittstellen.

- Optimiert für minimalen Bedarf an Ressourcen und low-Power
- Konfigurierbare Initialisierung
- Selektierbare Betriebsmodi (*WRITE_FIRST*, *READ_FIRST*, *NO_CHANGE*)
- Single-Port-, Dual-Port-RAM (gleichzeitige Lese- und Speicherzugriffe von zwei Seiten) and -ROM
- Eingebaute Hamming Fehlerkorrektur

3.3.3 Weitere IP-Blöcke

Der DSP Block kann in diversen Anwendungen verwendet werden (Bsp: Design, mathematische Funktionen, etc.). Er kann auch explizit mit einem Attribut im Code definiert/verhindert werden:

```
attribute use_dsp48 : string;
attribute use_dsp48 of P_reg: signal is "no";
```

DSP Block

Der Dual ARM Cortex A9 Prozessor kann ebenfalls über den IP-Katalog in ein Design miteinbezogen werden. Die Konfiguration geschieht über ein GUI.

Dual ARM Cortex A9 Prozessor

3.3.4 Schnittstellen

Es gibt folgende Interface Typen:

Interface Typen

- Processing System $\longleftrightarrow$ Extern
- Programmable Logic $\longleftrightarrow$ Extern
- Programmable Logic/Processing System $\longleftrightarrow$ Extern
- Processing System $\longleftrightarrow$ Programmable Logic (AXI)

Das AXI-Interface ist de facto das Standardinterface für on-Chip-Kommunikation.

AXI-Interface

- AXI4:
AXI4 ist eine Hochleistungsschnittstelle für die Kommunikation bei memory mapping. Die ermöglicht bis zu 256 Datentransfer-Zyklen pro Adressphase.
- AXI4-Lite:
Die AXI4-Lite Schnittstelle ist eine leichte Variante der AXI4 Schnittstelle. Sie kann für Einzeltransaktionen bei memory mapping verwendet werden. Diese Variante hat den Vorteil, dass sie weniger Logik benötigt als die AXI Schnittstelle.
- AXI4-Stream:
Keine Adressphase bedeutet, dass es nicht memory mapped ist und ermöglicht somit unbegrenzte Daten-Burst-Größen.

Der AXI-Interconnect-Block verbindet zwei IP-Böcke. Es kann konfiguriert werden, ob das Processing System der Master und der AXI-Interconnect-Block der Slave der AXI-Interconnect-Block der Master und die Programmierbare Logik der Slave ist.

AXI-Interconnect-Block

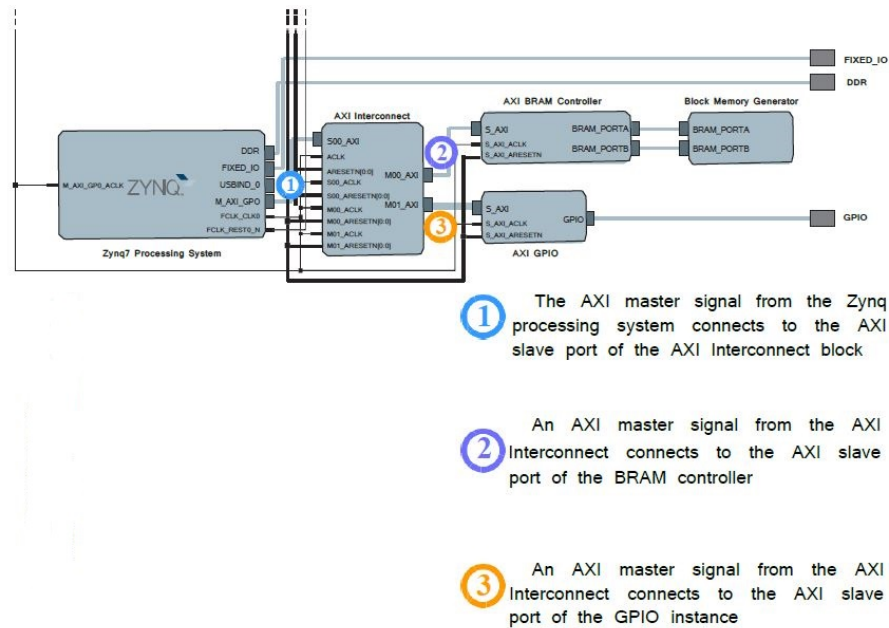


Abbildung 3.5: Beispiel AXI-Interconnect-Block

3.4 Integration eines IP Blocks

IP-Block integrieren

Der IP-Block wird integriert, indem man das Instantiierungs-Template in der Entity oder einem höheren Hierarchischen Level instantiiert. Wird der IP Block mit einem Block Design integriert, übernimmt Vivado alles (Instantiierung, Generierung des *system_wrapper*, Generierung der *output products*).

Instantiierung im HDL-Code

In HDL muss der IP im lokalen Design instantiiert werden. Ein Instantiierungs-Template ist gegeben.

Generator Files

IP generieren output-Files, welche im source-Ordner unter dem ip directory abgespeichert werden:

- Instantiierungs-Template (.vho File)
- Interne Daten für die Synthese
- Behavioral Modell für die funktionale Simulation (.vhd-File)
- Interne Daten für die Implementierung und Administration

3.5 Lernziele

1. Beschreibe den Syntax und die Verwendung von Funktionen (a), Prozeduren (b) und Packages (c) in VHDL.
 - a) Werden verwendet, um hierarchische Struktur in den Code zu bringen
 - b) Wiederverwendung von bereits geschriebenen Code

- c) Packages sind zentrale Repositories für Constraints, Typen, Komponenten, Funktionen und Prozeduren, die an verschiedenen Orten benutzt werden.
 - d) Syntax siehe Abb. 3.1, Abb. 3.2, Abb. 3.3
2. Beschreibe, wie man Parametrisierte Komponenten generiert und instantiiert.
- Werden mittel generics in der entity generiert.
 - Whizzard übernimmt den Job des Code schreiben für parametrisierte Bausteine.
3. Beschreibe, wie man den IP-Katalog nutzt und wie man IP in ein Design integriert.
- Der IP-Katalog beinhaltet eine Kollektion von IP-Blocks, die von Xilinx zur Verfügung gestellt werden, von einer 3rd-Party stammen oder selbst geschrieben wurden (in-house).
 - Ein IP-Block kann in einem Blockdesign mittels Drag and Drop hinzugefügt werden. Danach können sie spezifisch für das Projekt konfiguriert werden.
 - Verschiedene IP-Blöcke werden am Besten mit dem AXI-Bus untereinander bzw. mit dem Prozessor verbunden.
4. Liste die verschiedenen Typen von AXI-Schnittstellen von Zynq auf und beschreibe die Grundfunktionalität und Gebrauch der AXI-Schnittstelle (Verbindung PL-PS).
- AXI4: Memory-mapped Links (für Adressen-/Datenbursts)
 - AXI4-Lite: Vereinfachter Link (für Adress- und Datenworte)
 - AXI4-Stream: Für high-speed Streaming. Direkter Burst-Transfer von Datenquellen zu Endquellen mit uneingeschränkter Datengröße.

4 Custom IP Block Designs

4.1 Systemdesign

Ziele des
Systemdesigns

Die Ziele des Systemdesigns können verschieden angeschaut werden. Aus Sicht des Business steht vor allem die Konkurrenzfähigkeit im Vordergrund, aus Sicht des Ingenieurs eine Smarte Lösung eines Problems. Am wichtigsten ist die Funktionalität.

1. Priorität: Funktionalität
2. Priorität:
 - Konkurrenzfähigkeit (Time-to-Market, Kosten, Fähigkeiten)
 - Technische Aspekte (Grösse, Performance, Energieeffizienz, usw.)

4.1.1 Genereller Ansatz

Der Generelle Ansatz besteht darin, die Architektur des Systemdesigns mit einer Top-Down-Perspektive zu machen:

1. Top-Level Schnittstellen definieren
2. Top-Level Parameter definieren
3. Subsysteme definieren
4. Interaktionen zwischen den Subsystemen definieren

Dazu müssen die Systeme möglichst hoch abstrahiert werden und erst später ins Detail gegangen werden.

Möglichst viel in
Software

Systeme müssen möglichst flexibel werden. Darum sollte möglichst viel mittels Software beschrieben werden und nur dort Hardware eingesetzt werden, wo es nötig ist.

4.2 Hardware / Software Partitionierung

Hardware / Software Partitionierung entspricht dem Entscheidungsprozess, welche Teile des Systems in Hardware und welche in Software realisiert werden sollen. Diese Partitionierung wird für jedes Subsystem vorgenommen.

Definition

Das Ziel dieser Partitionierung ist, die bestmögliche Lösung für die Realisierung eines Subsystems zu finden. Dazu wird abgeschätzt, welche Vor- und Nachteile eine Realisierung in Hardware bzw. Software hat. Wichtige Entscheidungsfaktoren sind Funktionalität, Kosten, Performance, etc.

Ziel der Partitionierung

Die Software hat folgende Eigenschaften:

Eigenschaften Software

- Sequentielle Abarbeitung von Befehlen
- User-Applikationen
- GUIs

Die Hardware hat folgende Eigenschaften:

Eigenschaften Hardware

- Viel schneller als Software
- Wird dort Eingesetzt, wo ein intensiver Datenfluss auftritt
- Wird dort Eingesetzt, wo hoher Parallelismus gefordert wird

Mit geeigneten Tools kann abgeschätzt werden, welche Funktionen viel Ausführungszeit benötigen, bzw. wo Engpässe (bottlenecks) bestehen.

Hat man verschiedene Hardwareblöcke, so kann es zu einem Overhead bei der Kommunikation zwischen den Blöcken kommen, zudem sind sie teuer. Ein System on Chip löst viele dieser Probleme

Vorteile SoC

- SoC/ASIC reduziert die Grösse des Systems
- Designflow wird vereinfacht, da HW und SW integriert sind
- Die AXI-Schnittstele löst das Problem mit der Kommunikation zwischen den Blöcken. Die Daten zwischen den Blöcken können mit der höchsten Taktrate übertragen werden.

4.3 Genereller Designflow

Der Designflow besteht aus folgenden Punkten:

1. Anforderungen und Spezifikationen bestimmen
2. System Design (Subsysteme, Schnittstelle)
3. Hardware realisieren
4. Systemintegration vornehmen

4.3.1 Softwaredesign für ein System on Chip

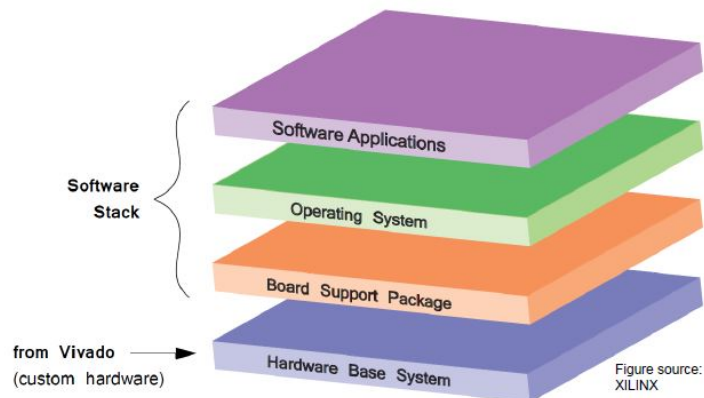


Abbildung 4.1: Zynq Design: Hardware and Software Layers

Das Board Support Package (BSP) ist die kundenspezifische Brücke zwischen dem Base System und dem OS. Es beinhaltet Hardware Parameter, Gerätetreiber, Low-Level OS-Funktionen.

Development Kit (SDK)

Xilinx unterstützt das Softwaredesign für ein SoC mit der Software Development Kit (SDK). Es hat folgende Eigenschaften:

- Projektmanagement
- Fehlernavigierung
- Kann sowohl C, als auch C++ kompilieren
- Generiert automatisch die make-Files
- Integrierte Debug-Umgebung
- Plug-Ins sind möglich

4.4 Entwicklung eines IP Blocks

Definition

Intellectual Property (IP) ist eine vorkonfigurierte Logikfunktion, die in das System integriert werden kann. Ein IP kann von einfachen arithmetischen Funktionen, bis zu komplexen Systemen (DSP, Filter, Multiplexer, Speicher, etc.) alles beinhalten.

Elemente

Die Elemente eines IP Block sind in Abb. 4.2 dargestellt.

Ein IP-Block kann mit dem IP-Packager generiert werden und so in einem späteren Design wiederverwendet werden (IP Source-Files → IP-Packager → Output Products).

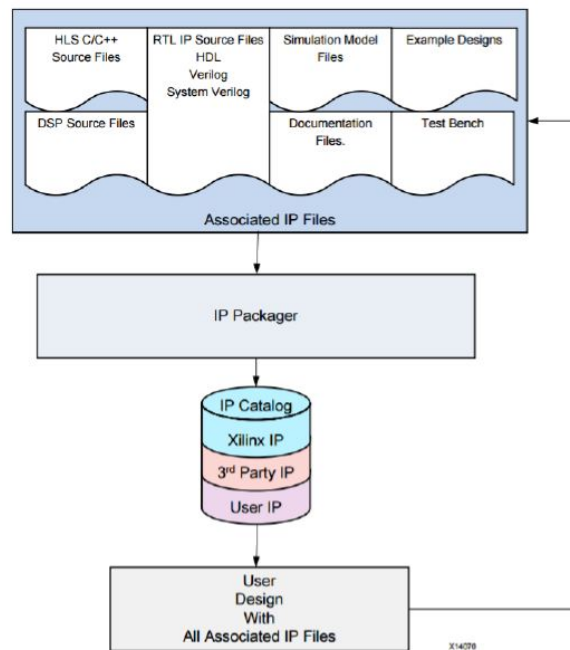


Abbildung 4.2: Elemente eines IP

4.4.1 Out-of-Context- vs. Globaler Synthese

Mit einer Out-of-Context (OOC) Synthese wird der IP ohne Kontext synthetisiert. Dabei wird der DCP (Container mit der Netzliste (für Simulation) und `_ooc.xdc` Constraints für Clock-Definitionen) mit gepackt. In anderen Worten: Alle Sachen die vom Kontext abhängig sind werden getrennt in ein eigenes File geschrieben. Der Teil der unabhängig ist wird gekapselt. Bei grossen Blöcken wird dann nur noch der Teil synthetisiert, der konextunabhängig ist. Folglich spart man Zeit.

Out-of-Context
Synthese

Mit einer Globalen Synthese wird der IP während der globalen Synthese re-synthetisiert. Mit dieser Variante hat man eine längere Compile-Zeit, dafür ist man eher offen für Optimierungen (flache Netzliste, globale Timing Constraints).

Globalen Synthese

4.5 IP Block Kreation

Wird über ein GUI aus Vivado generiert. Siehe dazu PDF-Seiten ab S.25).

IP-XACT ist ein Industriestandard (IEEE) XML-Format für die Beschreibung vom IP benutzten Metadaten (Ports, Schnittstellen, konfigurierbare Parameter, Files, Dokumentation). Es beschreibt nur High-Level-Informationen über den IP. Mit IP-XACT werden automatische Verbindung und Integration in ein System ermöglicht. Es ermöglicht ebenfalls die Integration von 3rd Party IP (und exportiert eigene IPs).

IP-XACT

Der Grundsätzliche Ablauf (IP Integration Flow) läuft folgendermassen ab:

- Erstellung eines IP Repository im aktuellen Projekt

Grundsätzlicher
Ablauf der IP
Integration

- IP Blöcke auf das System anpassen und daraus die projektspezifischen *output products* generieren
- Der IP kann mit einem Instanzierungs-Template in ein RTL-Projekt eingefügt werden oder mit dem IP-Integrator in einem Blockdesign.

kreierte Files

Folgende Ordner und Files werden kreiert:

- component.xml (Beschreibung des IP XACT)
- bd (Block Diagramm tcl-File)
- drivers (SDK und Software-Files: Einfache Register read/write Funktionalitäten)
- hdl (Verilog/VHDL Quellen)
- src (Importierte Quellen-Files)
- Example design
- xqui (GUI tcl file)

IP-Repository

Folgende Repository gibt es:

- Projektspezifisches IP-Repository:
Ist das Verzeichnis, in welches der IP-Packager die IP-Definition kreiert. Der Default ist das Quellenverzeichnis des Projekts. Vivado IDE kreiert für jeden IP ein eigenes Verzeichnis.
- Globales IP-Repository Das Repository ist an einem anderen Ort gespeichert. Dadurch sind auch sämtliche Quellen-Files extern gespeichert.

Achtung:

Verwendet man ein globales IP-Repository, können bei einer Synthese zu einem späteren Zeitpunkt die Quellen aktualisiert werden (dies ist optional, Vivado macht einen darauf aufmerksam). Dies kann zu Problemen führen, wenn es bei dem IP-Block ebenfalls Änderungen gegeben hat. Schnittstellenänderungen wären das Schlimmste.

4.6 Lebenszyklus eines IP-Blocks

Pre-Production

Ein Pre-Production IP Core ist in der Regel eine öffentliche Version für eine Gerätefamilie, hat aber noch keine abgeschlossene Qualifizierung für den Gebrauch in Produktdesigns.

Production

Ein Production IP Core ist ein IP, welcher für den öffentlichen Gebrauch vorgesehen ist und welcher bereits verifiziert wurde.

Discontinued

Ein Discontinued IP Core ist ein IP, den Xilinx abgesetzt hat.

Superseded

Ein Superseded IP Core ist ein IP, welcher durch eine neuere Version ersetzt wurde.

Removed

Ein Removed IP Core ist ein IP, der von Xilinx entfernt wurde und in neuen Releases von Xilinx nicht mehr enthalten ist.

Versionsnummerierung von IP-Blöcken

IP-Blöcke werden ständig weiterentwickelt und angepasst. Dabei kann es sein, dass

bei einem Update des IP-Blocks das System inkompatibel mit der neuen Version ist, oder dass sogar Schnittstellen geändert wurde. Überarbeitungen am IP werden deshalb mit folgenden Key-Wörter unterschiedet:

- Major
- Minor
- Revision

Die Bedeutung und Konsequenzen sind in Abb. 4.3 aufgeführt.

Change Level	User Action	Examples of Changes
Revision Interface gleich geblieben es sollte bei einem Update keine Probleme geben	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 hier könnte das Interface oder eine Option geändert worden sein. Aufpassen!	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 hier muss man sehr gut aufpassen!	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 Behavior change for all configurations

Abbildung 4.3: Versionsänderungen von IP-Blöcken

4.7 Lernziele

1. Listen Sie mindesten 4 Kriterien auf, die das Systemdesign leiten/beeinflussen.
 - Funktionalität
 - Konkurrenzfähigkeit:
Kosten, Time-to-Market, Fähigkeiten
 - Technische Aspekte:
Grösse, Performance, Energieeffizienz
2. Beschreiben Sie ein typischer Design-Flow für die Hardware - Software Code-sign und der Hardwareteil im Allgemeinen.
 1. Anforderungen und Spezifikationen bestimmen
 2. System Design (Subsysteme, Schnittstellen)
 3. Hardware realisieren
 4. Systemintegration vornehmen

Grundsätzlich gilt: Möglichst viel in Software realisieren (ist flexibler) und nur das nötigste in Hardware.

3. Führen Sie die nötigen Schritte auf, die nötig sind, um einen IP Block zu kreieren.

Mit dem IP-Packager werden die IPs generiert. Der IP-Packager kombiniert sämtliche Quellen, die zu dem IP gehören. Zu den relevanten Files gehören HLS C/C++ Source Files, RTL IP Source Files (HDL, Verilog, VHDL), DSP Source Files, Simulationsmodelle, Dokumentationsfiles, Beispieldesigns und die Test Bench (siehe Abb. 4.2). Der generierte IP Block wird dann in einem IP-Repository abgespeichert (global oder lokal). Aus diesem IP-Repository kann dann der generierte und weitere IP-Blöcke in ein Design eingefügt werden.

4. Führen Sie die zu beachtenden Punkte auf, die nötig sind, damit der ein kreierter IP-Block wiederverwendet werden kann.

Generierte IPs werden wie schon oben erwähnt in einem Repository gespeichert. Damit sie wiederverwendbar sind, sollten sie möglichst offen für Anpassungen sein. Die Schnittstellen und die generischen Parameter müssen deshalb gut überlegt definiert werden.

5. Was ist der Unterschied zwischen OOC und "Globaler Synthese"(global synthesis).

- Out-of-Context Synthese

Der Teil der unabhängig ist wird gekapselt. Bei grossen Blöcken wird dann nur noch der Teil synthetisiert, der kontextunabhängig ist. Bei der Synthese spart man somit Zeit.

- Globale Synthese

Mit einer Globalen Synthese wird der IP während der globalen Synthese re-synthetisiert.

6. Beschreiben Sie den Mechanismus für den Gebrauch eines IPs in einem neuen Kontext und der Verbindung zum Prozessor.

Die Verbindung wird über den AXI-Bus realisiert. Dieser kann über das Konfigurierungs-GUI des IPs spezifisch auf das Projekt angepasst werden.

7. Beschreiben Sie den IP Lebenszyklus.

Pre-Production → Production → Discontinued → Superseded → Removed

8. Beschreiben Sie die Integration eines IPs in ein persönliches Design.

IPs werden ständig weiterentwickelt und angepasst. Bei einer erneuten Synthese eines Designs kann es sein, dass am IP Änderungen vorgenommen wurden. Folglich kann es sein, dass die neue Version nicht mehr mit dem Design synthetisiert werden kann. Änderungen werden mit **Revision** (kleine Änderungen, sollte keine Probleme geben), **Minor** (mittlere Änderungen, könnte bereits zu Problemen kommen und **Major** (schwerwiegende Änderungen, äusserst gut Aufpassen). Bei der Synthese kann ausgeählt werden, ob man die Quellen aktualisieren möchte oder die Alte Version beibehalten möchte.

- IP integrieren:

Ein IP-Block wird integriert, indem man das Instantiierungs-Template in der Entity oder einem höheren Level instantiiert.

- Instantiierung in HDL:
In HDL muss der OP im lokalen Design instantiiert werden. Ein Instantiierungs-Template ist gegeben.
- Integration in ein Block-Design:
Vivado übernimmt sämtliche Aufgaben (Instantiierung, Generierung des *system_wrapper*, Generierung *output products*).

5 Verifying your SoC

5.1 Verifikation des Projekts

5.1.1 Verifikation während der Spezifikationsphase

Grundfrage	Die Grundfrage während der Spezifizierungsphase (Analyse) lautet: Mache ich das richtige, was ist wirklich nötig (doing the right things)
	Dazu muss ständig hinterfragt werden, ob die Spezifikationen mit der Funktionalität übereinstimmen und ob die Kunden, das Marketing und die Ingenieuren dieselben Ansichten haben.
Formale Spezifikationen	Formale Spezifikationen sind funktionale Anforderungen, die in eine formale Notifikation gebracht werden (Signalflussgraphen, Wahrheitstabellen, etc.). Diese Art der Notation ist ungeeignet für die Kommunikation zwischen Ingenieur Kunden/Management.
Rapid Prototyping	Bei Rapid Prototyping (virtual prototyping) wird ein algorithmisches Modell aufgestellt, dass die Funktionalität der Zielschaltung charakterisiert.
Hardware unterstützte Verifikation	Bei einer Hardware unterstützte Verifikation (Spezialform von rapid prototyping) wird die Schaltung auf einem speziellen Board verifiziert. Meistens ist diese Schaltung langsamer als die Zielhardware.

5.1.2 Verifikation während der Designphase

Dynamische Verifikation	Bei der dynamischen Verifikation wird eine Simulation ausgeführt. Diese besteht aus einem DUT (<i>device under test</i>) und verschiedenen Stimulis. Die Stimulis regen die Schaltung bzw. gewisse Funktionen an. Von jedem Stimuli werden dann die Systemantworten ausgewertet. Die dynamische Verifikation ist der statischen Verifikation vorzuziehen.
-------------------------	---

Folgendes sollte beachtet werden:

- Funktionalität und Parametereinstellungen sollten separat getestet werden.
- Grundsätzlich müssen alle möglichen Testfälle (*Exhaustive Verification*) (Input/State-Kombinationen) und deren Antworten geprüft werden. Dies ist oftmals wegen den vielen Möglichkeiten nicht möglich .
- Ein Kompromiss ist, dass nicht alle, dafür die wichtigen und doch genügend viele Fälle getestet werden. Folgendes ist wichtig:
 - assertion-basierte Verifikation machen

- Testfälle von verschiedenen Quellen sammeln (regulärer Bereich, Under-/Overflow, Fehlerfälle, zufällige Inputs, etc.)
- Separate (unabhängige) Teams für Schaltungsentwurf und Testdesign bilden
- Verwendung des Rapid-Prototyping
- Der Test muss reproduzierbar und automatisiert sein. Visuelle Betrachtung der Systemantworten reichen nicht, es müssen automatische Antworten generiert werden (assertions).

Achtung: Ein algorithmisches Modell kann nicht das physikalische Timing testen.

Bei der statischen Verifikation werden Codeinspektionen, Statische Timing-Analysen, DRC, ERC, etc. durchgeführt.

Statische Verifikation

Zusätzlich müssen auch der Energiebedarf, die Temperatur des FPGAs, usw. berücksichtigt werden.

Bei der assertion basierten Verifikation werden kleine Codefragmente in den Code des DUT eingebaut, die eine Meldung ausgeben, wenn es zu einem Fehler kommt. Sie sind jedoch nicht synthetisierbar und müssen für die Synthese auskommentiert oder entfernt werden. Folgende Punkte sind relevant:

Assertion basierte Verifikation

- Speicherzugriff auf einen unerlaubten Bereich
- Zustandsmaschinen, die in einen Fehlerzustand kommen
- Unvorhergesehene Input-Werte
- Numerische Under-/Overflow

Der Syntax geht folgendermassen:

assert (condition) report "message" severity error

Die condition ist dabei ein Boolescher Ausdruck. Der Schweregrad (severity: note, warning, error, failure) ist optional.

Das Ziel des Virtual Prototyping ist, den virtuellen Prototyp in der realen Umgebung zu testen und damit ein "Golden Model" zu erstellen. Das Golden Model kann dann benutzt werden, um die Systemantworten zu berechnen.

Prinzipien des Virtual Prototyping



Abbildung 5.1: Funktionsweise von Virtual Prototyping

Für Stimuli/Response basierte Test Benches werden Library zur Verfügung gestellt:

Library

Es wird zwischen folgenden Methoden zur Funktionaler Verifikation unterschieden:

Libraries	Libaray for Text IO is required: • use STD.TEXTIO.ALL;
File declaration	File to be used must be declared using <i>FILE</i> statement: -- A file of variable-length ASCII records. • FILE test_file: text open read_mode is "test1.txt"; • FILE results: text open write_mode is "results1.txt";
Line buffer declaration	Variable of <i>LINE</i> type is used to read/write text lines in file: • Variable line_buffer: line; -- Line is a pointer to a STRING value
Read entire file:	Loop: while not endfile(test_file) loop..... end loop;
Read lines from file:	Read to line_buffer: readline(test_file, line_buffer);
Supported content:	Following VHDL types are supported: bit, bit_vector, boolean, character, string, integer, time VHDL2008 adds some more, but is not supported by Vivado.

Abbildung 5.2: Folie Funktionsweise Library

- **Simulation:**
Ausführbares Modell des Design auf einem Host-Rechner. Ist gut beobachtbar und man kann Fehler gut finden. Vielfach nicht real-time und langsam in der Ausführung. Es ist deshalb für grosse SoCs nicht geeignet.
- **Emulation:**
Ausführbares Modell des Designs in rekonfigurierbarer Hardware. Diese Methode bietet etwa die gleiche Vorteile wie die Simulation, ist schneller. Dafür ist sie jedoch aufwändiger.
- **(Physisches) Prototyping:**
High Level oder RTL code synthetisiert in rekonfigurierbarer Hardware. Mit dieser Methode können real-time Verifikationen gemacht werden und für grosse SoCs geeignet. Die Möglichkeiten für die Beobachtbarkeit und Fehlerbehebung sind begrenzt.
- **Emulation and (physical) Prototyping:**
Emulation der Schaltung auf einem FPGA bevor sie auf einem ASIC implementiert wird. Erlaubt oftmals eine real-time Emulation.
- **Hardware / Software Co-Simulation:**
Teile der Schaltung werden auf Hardware geprüft, andere in Software oder mit einem High-Level Model (z.B. mit Matlab)
- **In-circuit /in-environment Emulation**

5.2 Lernziele

1. Beschreiben Sie die Verfahren zur Gewährleistung einer guten Schaltung während der Spezifikationsphase.
Grundsätzlich muss immer wieder in Frage gestellt werden, was muss gemacht werden und was wirklich nötig ist. Zudem muss sichergestellt werden, dass der Kunde, das Marketing und die Entwickler die selben Vorstellungen vom Produkt haben.

- Formale Spezifikationen:
Formale Spezifikationen sind funktionale Anforderungen, die in eine formale Notifikation gebracht werden (Signalflussgraphen, Wahrheitstabellen, etc.). Diese Art der Notation ist ungeeignet für die Kommunikation zwischen Ingenieur Kunden/Management.
 - Rapid Prototyping:
Bei Rapid Prototyping (virtual prototyping) wird ein algorithmisches Modell aufgestellt, dass die Funktionalität der Zielschaltung charakterisiert.
 - Hardware unterstützte Verifikation:
Bei einer Hardware unterstützte Verifikation (Spezialform von rapid prototyping) wird die Schaltung auf einem Speziellen Board verifiziert. Meistens ist diese Schaltung langsamer als die Zielhardware.
2. Was ist mit erschöpfende Überprüfung (*exhaustive verification*) gemeint?
Bei einer *exhaustive verification* wird der Ausgang des Systems für jeden möglichen Eingang und jeder möglichem State verifiziert.
 3. Führen Sie mindestens 4 Messungen zur Überprüfung von einer guten Schaltung während der Designphase auf.
 - Funktionale Verifikation
 - Statische/Dynamische Verifikation
 - Assertion basierte Verifikation
 - Virtual Prototyping
 4. Beschreiben Sie den Unterschied zwischen statischer und dynamischer Verifikation.
Bei der statischen Verifikation werden Codeinspektionen, Statische Timing-Analysen, DRC, ERC, etc. durchgeführt. Bei der dynamischen Verifikation wird eine Simulation durchgeführt. Dabei regen verschiedene Stimuli das DUT an, gleichzeitig wird dann jeweils die Systemantwort überprüft.
 5. Was gibt es für File-Operatoinen in VHDL und wie ist eine Test Bench mit einem File-Input aufgebaut?
Solche Tests können mit Libraris gemacht werden. Siehe dazu Abb. 5.2.
 6. Wie kann ein Code oder eine FPGA-Schaltung debuged werden?
 - Force-Konstanten in Simulation
 - Stepping
 - (Conditional) Breakpoints
 - ILA und VIO
 - Hardware Simulation.

6 Verifying your SoC II

6.1 Lernziele

1. Listen sie Gründe auf, warum System on Chip Systeme verifiziert werden sollen.
 - Verringerung der Qualität im Falle eines fehlerhaften Produktes
 - Unzufriedenheit der Kunden, Verlust des Vertrauens
 - Fehlfunktionen Verursachen massive Kosten
 Faustregel: **Die Kosten für einen unentdeckten Fehler vervielfachen sich um Faktor 10 in jedem Produktionsschritt!**
2. Beschreiben Sie zwei Fehlermodelle.
 - **Stuck-at-Fault:**
Stuck-at-Faults (dt. Haftfehler) sind Fehler, bei denen Gatter an einem Eingang oder Ausgang auf einem bestimmten Wert "feststecken". Je nach Wert unterscheidet man Stuck-at-1 oder Stuck-at-0-Fehler.
 - **Bridging Fault:**
 Als *Bridging-Faults* werden Kurzschlüsse zwischen zwei Leitungen bezeichnet. Daraus entstehen neue Funktionen der betroffenen Gatter (AND-Glied zeigt Verhalten eines OR-Gliedes, etc.). Input-Output-Bridging bedeutet, dass Ein- und Ausgänge der Schaltung miteinander verbunden sind (Rückkopplung). Dies führt zu zeitabhängigen Verhalten, z.B. zu oszillierenden Schaltungen.
 - **Delay Fault:** Als delay fault werden Timing-Fehler des Systems bezeichnet. Diese entstehen, wenn Pfad-Delays zu lange sind. Ursache dafür können zu lange Pfade sein, aber auch Fabrikationsfehler (hoher Widerstand einer Leitung).
3. Was wird unter *fault collapsing* für gegebene Vektoren verstanden?
 Wenn nur das Vorhandensein eines Fehlers von Interesse ist, aber nicht, wo genau der Fehler ist, ist normalerweise ein kompletter Test nicht notwendig. Eine Untergruppe von Testvektoren reicht dann aus. Dies wird auch *fault collapsing* genannt.

Erklärung Wikipedia:

Es kann sein, dass zwei oder mehrere Fehler für sämtliche Input-Pattern dasselbe Fehlerverhalten aufweisen (*equivalent faults*). Einer dieser Fehler von dem Set von äquivalenten Fehlern kann das ganze Set repräsentieren. In diesem Falle müssen nicht $k \cdot n$ Fehlertests gemacht werden für eine Schaltung mit

n Signallinien. Das Entfernen von äquivalenten Fehlern vom Gesamtset der Fehler wird *fault collapsing* genannt.

4. Beschreiben Sie den IDDq Test und deren Vorteile.

- Beim IDDq (Current Drain-Query) Test wird einer Schaltung verschiedene Testvektoren angeregt. Dabei wird der Strom der Schaltung gemessen und mit dem erwarteten Strom verglichen. Hat es in der Schaltung einen *bridging fault*, kann man diesen wegen dem höheren Strom erkennen. Es müssen verschiedene Messungen gemacht werden und nicht nur in einem statischen Zustand der Schaltung.
- Vorteile: schnelles und zuverlässiges Verfahren für ein erstes Screening, sehr einfacher Test

5. Beschreiben Sie den Vorgang und Anforderungen für den *scan test*.

Ein scan test ist eine Technik, um die Steuerbarkeit und Beobachtbarkeit einer logischen Schaltung zu erhöhen. Dazu werden "Scan-Register" in die Schaltung eingebaut. Im Normalbetrieb fungieren sie als normale Flipflops. Sie können aber in einen Testmodus umgeschaltet werden, wo sie zu langen Schieberegister werden. Dies erlaubt nun Daten seriell durch die Scan-Register zu schieben, indem man konstant Daten am Eingang einspeist. Die Daten am Ausgang entsprechen dann unter anderen auch internen Zuständen, die von aussen nicht sichtbar sind.

- Die Originalschaltung angepasst werden, damit ein scan test gemacht werden kann.
- Gewisse Tools bieten Design For Test Generatoren (DFT-Generatoren) an, die bei der Synthese automatisch eine Scan-Kette hinzufügen. Dies wird beim ASIC-Design gebraucht.
- Bei FPGA-Synthesen wird die Steuerbarkeit und Beobachtbarkeit bereits mit der JTAG und *debug cores* eingebaut.
- Automatic Test Pattern Generatoren (ATPG) können zudem verschiedene Test-Pattern generieren.

6. Beschreiben Sie den Ablauf von BIST und listen Sie die benötigten Komponenten auf.

Testen ist teuer (0.05 \$/s), deshalb sollen Built in Self Tests (BIST) eingesetzt werden. Built in self test (BIST) bedeutet, dass ein elektronischer Baustein eine integrierte Testschaltung besitzt, welche Testsignale erzeugt und meist auch mit vorgegebenen richtigen Antwort-Signalen vergleicht, sodass das Testresultat an ein ATE (Automatic Test Equipment) ausgegeben werden kann. Built in self tests sind durch automatisierte Designprozesse immer einfacher zu implementieren und nehmen bei der heute üblichen grossen Anzahl von Schaltelementen relativ wenig Platz ein, reduzieren aber den materiellen und zeitlichen Aufwand beim Testen erheblich. Sie werden auch zum regulären Selbsttest von Prozessoren während ihrer Anwendung oder beim Ein- bzw. Ausschalten verwendet, um Fehlfunktionen rechtzeitig zu erkennen und Folgeschäden zu vermeiden. Diese Methode benötigt zusätzliche Komponenten:

- Speicher RAM/ROM
Wird für Vergleichsvektoren benötigt. Die Antworten des Systems müssen komprimiert werden, da sonst viel Speicher benötigt wird.
- DSP
- Zusätzliche Logik

7. Beschreiben Sie die Funktionalität von *boundary scan tests* und listen Sie die benötigten Komponenten auf.

Durch die Komplexität und Kleinheit heutiger Schaltungsaufbauten wird es immer schwieriger, physisch auf bestimmte Punkte einer Schaltung zuzugreifen. Der Test von z.B. multi layer PCBs ist schwierig, da die Beobachtbarkeit und Steuerbarkeit der internen Knoten stark begrenzt ist. Testadapter können helfen, sind jedoch sehr teuer.

Die Boundary Scan-Methode verwendet zusätzliche Zellen (Latches), mit deren Hilfe Signale über vordefinierte Pfade von aussen in die zu testende Schaltung injiziert werden können. Die Signale aus der Schaltung, die an Pins des IC anliegen, können über den Scanpfad erfasst werden. Im Normalbetrieb sind die Latches passiv. Es besteht kein Unterschied zu ICs ohne Boundary Scan-Funktionalität; die Anschlüsse des Chip sind nur mit den Pins des IC verbunden. Im Testbetrieb werden sie dem Verfahren entsprechend aktiv gesteuert.

- Jede Komponente benötigt für einen boundary scan zusätzliche Hardwarekomponenten.
- Jeder Input und Output muss durch eine *boundary scan cell* (BS Cell) ersetzt werden.
- Test Access Port Controller (TAP Controller)

7 IP Block Design using HLS

7.1 Lernziele

1. Erklären Sie, warum man HLS verwenden sollte.
 - High-Level Synthesis (HLS) kreiert eine RTL Implementation aus einem C-Code. Es ist primär ein Werkzeug, um den Designprozess zu beschleunigen. Somit kann man mehr Zeit für die Funktionalität aufwenden.
 - HLS ist eine Methode, um ein Problem zu abstrahieren. Damit kann die Komplexität besser in den Griff bekommen werden.
 - Mit HLS wird der Design-Abschnitt übersprungen, bei dem die Meisten Tools und Methoden eingesetzt werden müssen.
 - Designs können mit verschiedenen Direktiven auf kleine, schnelle oder optimale Designs optimiert werden. Der Quellcode legt nicht die Architektur fest.
 - Mit HLS kann Zeit eingespart werden bzw. die Produktivität wird gesteigert.
2. Erklären Sie, wo es sinn macht HLS einzusetzen und wo es besser ist, auf Registertransferebene (RTL) den Code zu schreiben.
 - Grundsätzlich sollte man HLS so oft einsetzen wie man kann.
 - In Systemen die extrem schnell, energiesparend, oder klein sein müssen, ist HLS nicht das optimale Werkzeug. Dort muss man allenfalls "von Hand" Code schreiben, um ein optimales Ergebnis zu bekommen. Man sollte jedoch HLS soweit wie es geht einsetzen und dann auf der Ebene des Systemdesigns optimieren.
 - HLS-Code ist lesbar, jedoch nicht immer einfach zu verstehen. Wenn Systeme geändert oder erweitert werden sollen, macht man dies am besten auf HLS-Level. Für Änderungen in der untersten Hierarchiestufe ist es nicht geeignet.
3. Erklären Sie, wie HLS den Designer bei den Designherausforderungen (time-to-market, Komplexität, Heterogenität) unterstützt.
 - HLS generiert automatisch VHDL-/Verilog-/SystemC-Code und übersetzt ebenfalls die C-Testbench in den Zielcode.
 - HLS generiert automatisch alle relevanten Daten für einen IP-Block.
 - Mit HLS lässt sich ein grosses System gut abstrahieren.
 - Das System lässt sich auf verschiedene Sachen optimieren.

4. Nenne die die wichtigsten Schritte im HLS Designfluss und dem Ausführen eines HLS Design.

- Als erstes wird die Hardware/Software Partionierung gemacht. Dabei wird bestimmt, welches die kritischen Pfade des Systems sind. Kritische Pfade können Teile des Systems sein, die schnell sein müssen oder bei denen die CPU viel Rechenzeit benötigt. Diese Teile können dann in Hardware ausgelagert werden. Grundsätzlich sollte man jedoch möglichst viel in Software lösen.
- Im zweiten Schritt wird mit der eigentlichen HLS Design begonnen. Inputs des HLS Designs sind der Quellcode (C/C++/SystemC), C-Testbench, Golden Reference (Erwartete Systemantworten), Direktiven und Constraints.
- Mit den Inputs wird dann eine Verifikation der funktionalen Integrität gemacht. Eine äquivalente Testbench zur C-Testbench wird automatisch durch HLS erzeugt.
- Ist alles in Ordnung wird dann die High-Level-Synthese gemacht. Diese kann mit Direktiven und Constraints beeinflusst werden.

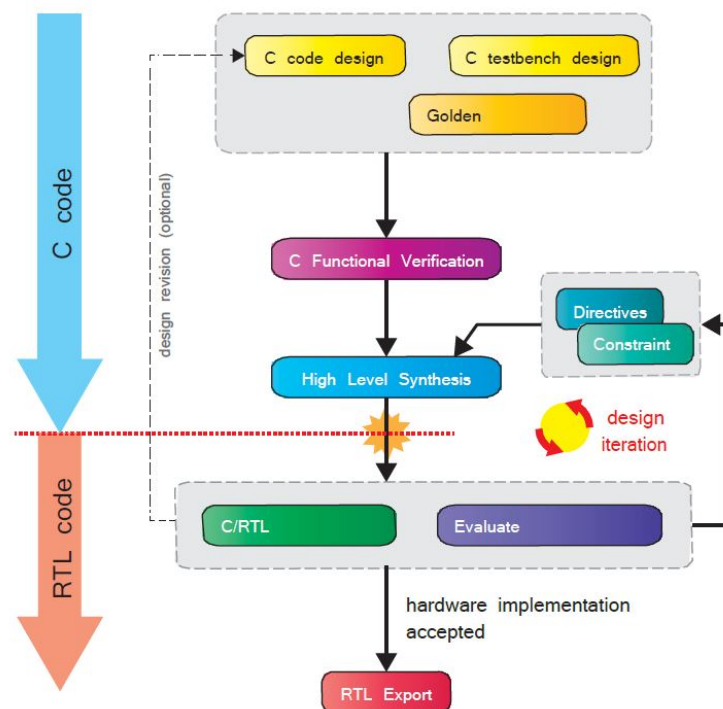


Abbildung 7.1: HLS Design Flow

5. Welches sind die wichtigsten Kennzahlen für die Leistungsbewertung?

- Benötigte Ressourcen (=area), Kosten
- Schnelligkeit (speed)
Wichtige Faktoren sind dabei die Clockfrequenz, die Latenzzeit und die Datenrate.

- Energiebedarf
 - I/O Anforderungen (Anzahl und Komplexität)
6. Erklären Sie, wie man Direktiven hinzufügt und welchen Einfluss sie auf das Design haben.
- Direktiven können im Quellenfile hinzugefügt werden (`#pragma`) oder in einem Direktiven-File (`%directive`) bestimmt werden.
 - Die Direktiven optimieren ein System auf eine gewisse Sache. Die Default-Priorität ist Optimierung auf die Grösse. Folgende Direktiven sind möglich:
 - Optimierung auf Grösse des Systems (minimum area)
 - Optimierung auf die Performance (Clock und Datenrate)
Falls ein Clock-Pfad Timing-Probleme aufweist, wird ein separater Clockpfad generiert.
 - Optimierung auf die Latenzzeit
 - Default Design:
Das System wird dabei mit dem Default-Design implementiert. Die selbe Hardware wird für jede Iteration eines Loops verwendet. Eigenschaften: kleiner Ressourcenbedarf, lange Latenzzeiten, kleine Datenrate
 - Unfrolled Loop Design:
Unterschiedliche Hardware wird für jede Iteration verwendet. Eigenschaften: grösserer Ressourcenbedarf, kurze Latenzzeit, bessere Datenrate.
 - Pipelined Design:
Verschiedene Iterationen werden gleichzeitig ausgeführt. Eigenschaften: grösserer Ressourcenbedarf, kürzere Latenzzeit, beste Datenrate
7. Wie funktioniert HLS in Vivado?
- a) Extraction of data path and Control:
Komponenten des Datenpfads und der Kontrollschaltung identifizieren und extrahieren.
 - b) Scheduling and binding
RTL Statements von C-Code generieren und dabei die Dauer (Clockzyklen) assoziieren. Physikalische Ressourcen vom Zielgerät zuweisen. Falls nötig werden hier mehrere Iterationsschritte unternommen. Beim Scheduling wird bestimmt, in welchem Clockzyklus welche Operation ausgeführt wird. Bei Binding wird bestimmt, welche library cell verwendet wird jeder Operation.
 - c) Optimierung:
Mit Hilfe von Constraints (Clock oder Ressourcen Gebrauch) und Direktiven (Wie werden Loops, Arrays, etc. behandelt) System optimieren.
8. Wie werden Attribute vom C-Code in den Zielcode übersetzt?

- Funktionen:
Jede Funktion wird in ein RTL-Block übersetzt. Kleine Funktionen werden automatisch als *inline*-Funktion übersetzt.
- Schnittstellen:
Die Argumente der Top-Level Funktionen bestimmen die Schnittstellen des RTL-Designs.
- Datentypen:
Datentypen werden so übersetzt, wie sie im Code spezifiziert sind. Man muss also genau abklären, wie gross die Datentypen sein müssen. Ansonsten werden Ressourcen verschwendet.
- Arrays:
Arrays werden als Speicher (Block-RAM) im RTL-Design implementiert. Optimal kann man sie auch als FIFO-Speicher implementieren.

8 Modelling of digital Filters

8.1 Lernziele

1. Sie können die grundlegenden Filterstrukturen aufzeichnen und wissen, wie die Struktur Einflüsse auf die benötigten Ressourcen eines FIR-Filters hat.

- Digitale Tiefpass FIR-Filter sind immer stabil. Sie haben folgende Differenzengleichung:

$$y[n] = c_0 \cdot x[n] + c_1 \cdot x[n-1] + c_2 \cdot x[n-2] + \dots + c_N \cdot x[n-N] \quad (8.1)$$

oder im z-Bereich:

$$H(z) = \frac{Y(z)}{X(z)} = c_0 \cdot z^0 + c_1 \cdot z^{-1} + c_2 \cdot z^{-2} + \dots + c_N \cdot z^{-N} \quad (8.2)$$

- Direktform (geht ganz einfach mit DSP48-Block):

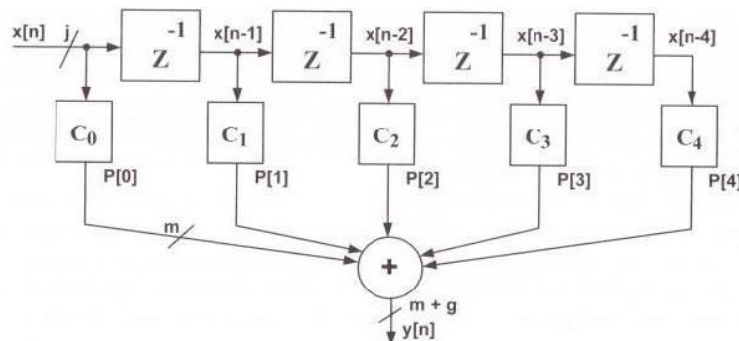


Abbildung 8.1: FIR-Filter implementiert mit Direktform

- FIR-Filter mit linearer Phase:
Die Koeffizienten dieses Filters sind symmetrisch, d.h. $c[n] = c[n-N]$. Der Vorteil ist, dass man nun eine lineare Phase und eine konstante Gruppenlaufzeit hat (keine Verzerrungen).
 - Transponierte Form:
Der Nachteil dieser Variante ist, dass jedes Register eine andere Grösse haben muss. Deshalb ist sie eher ungeeignet.
2. Sie können die benötigten Hardware-Elemente auflisten und die Anzahl benötigter Elemente für verschiedene Strukturen angeben.
Folgende arithmetische Blöcke werden für einen FIR-Filter benötigt:

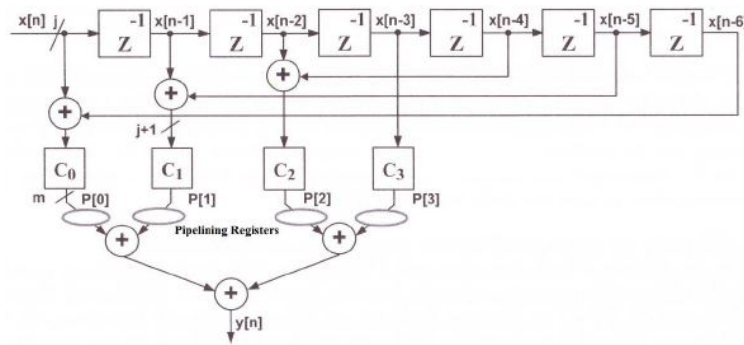


Abbildung 8.2: FIR-Filter mit linearer Phase

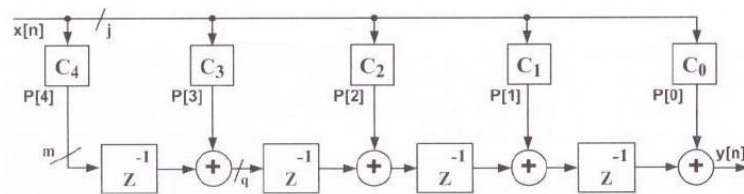


Abbildung 8.3: FIR-Filter implementiert mit Transponierter Form

- Delay-Elemente
Delay-Elemente z^{-1} werden als Register realisiert. Normalerweise sind sie Flipflops, können aber auch als RAM-Zelle implementiert werden. Dies ist nur bei grossen Ordnungen sinnvoll.
- Multiplizierer
- Addierer
- Benötigte Anzahl an Delay-Elementen ist überall gleich
- Benötigte Anzahl an Flipflops
 - Direktform und lineare Phase: $FF = N \cdot j$
 - Transponierte Form: $FF = N \cdot q_x$
- Benötigte Anzahl Addierer:
Grundsätzlich braucht es N Addierer. Da es zu Problemen mit der Latenzzeit kommen kann, müssen allenfalls zusätzliche Register hinzugefügt werden. $N - 1$ Pipeline-Register und $(N - 1)!$ Delay-Kompensationsregister. Die Latenzzeit mit dieser Variante beträgt $N - 1$ Clockzyklen.

Die direkte Form kann auch mit einem ausgeglichenen Addierer-Baum realisiert werden. Die Latenzzeit beträgt hier $\text{round}(\log_2(N + 1))$ Clockzyklen. Es werden $N - 1$ Pipeline-Register und 0 oder 1 Kompensationsregister benötigt.

3. Sie können FIR-Filter in VHDL beschreiben.

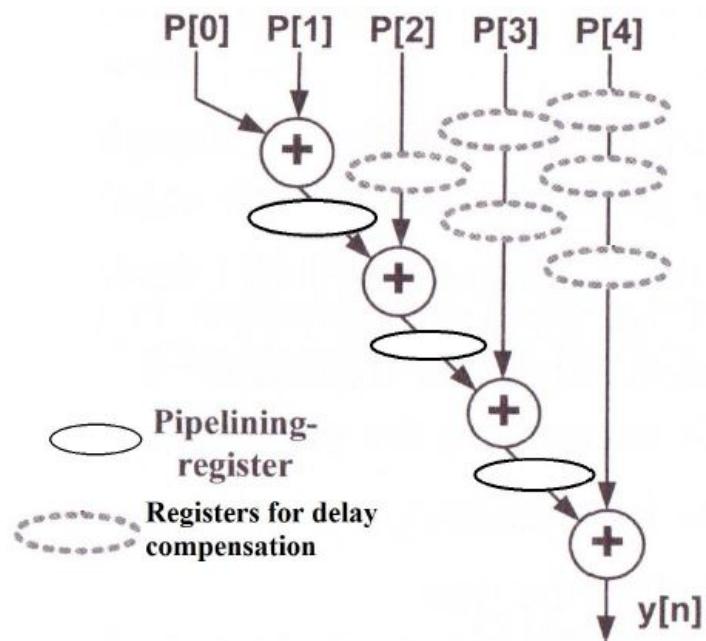


Abbildung 8.4: FIR-Filter mit Pipeline Register

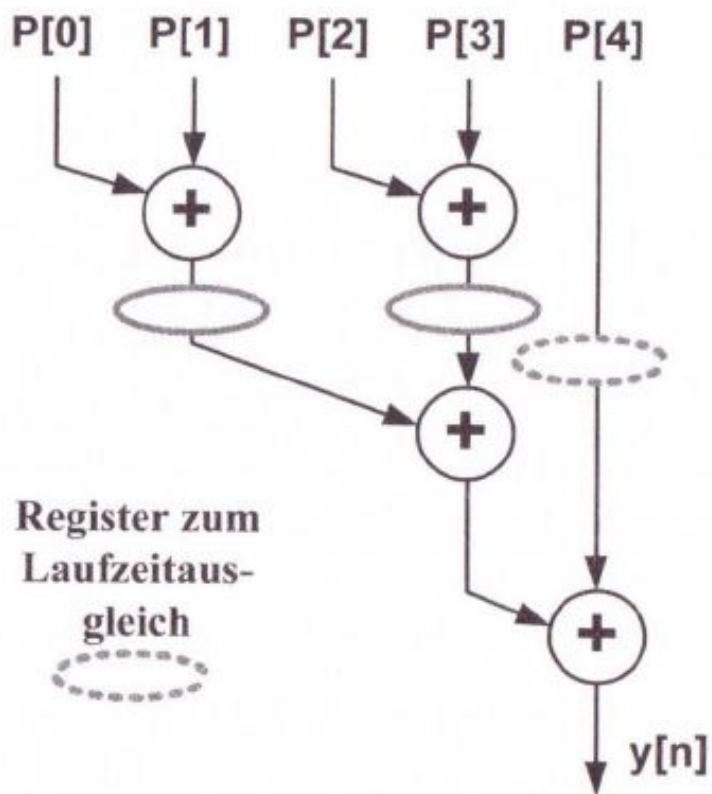


Abbildung 8.5: FIR-Filter ausgeglichenem Adder-Tree

9 SoC ASIC Design

9.1 Lernziele

1. Beschreiben Sie den typischen FPGA Design-Flow.

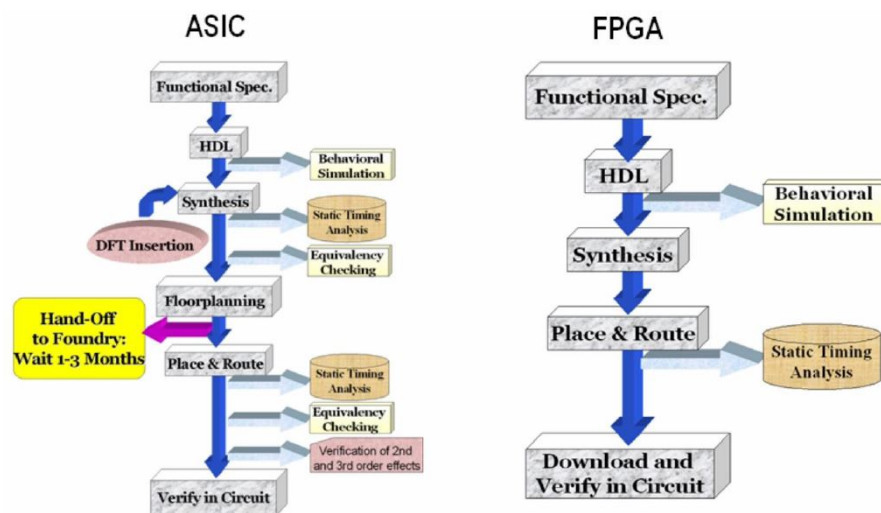


Abbildung 9.1: Design-Flow des FPGA und eines ASICs

- Beim Entry-Design gibt es keine Unterschiede. Behavioral Simulation ist unabhängig von der Technologie.
- FPGA-Synthese:
 - Bei einer FPGA-Synthese wird die RTL-Beschreibung in eine Netzliste (*mapped netlist*) transformiert.
 - Zudem wird bestimmt, welche FPGA-Familie und FPGA-Typ verwendet wird.
 - Das Ganze wird von einem Synthesetool umgesetzt, welches ein paar tausend Dollar kosten kann.
 - Zusätzlich können verschiedene Constraints gemacht werden.
- Test Logik bei Synthese:

FPGA hat eine eingebaute *scan chain*, da scan und JTAG für das programmieren benötigt werden
- Verifikation der Synthese:

FPGAs sind geprüfte Hardware. Deshalb ist eine Prüfung der Netzliste meistens nicht notwendig. Zeit kann somit gespart werden.

- Floorplaning:
Beim FPGA übernimmt das Synthesetool die Platzierung der funktionalen Blöcke. Man kann jedoch optional mit Constraints Einfluss auf die Platzierung nehmen.
- Bestimmung der I/O:
Werden in einem File bestimmt und vom Tool gelesen.
- Place and Route:
In Vivado wird dies automatisch vom Tool gemacht.
- Schaltung verifizieren:
Kann über Debug Core bzw. JTAG-Schnittstelle vorgenommen werden. Man muss nicht auf Prototypen warten.
- Endprodukt verifizieren:
Es ist nicht notwendig, da der FPGA als fertige Hardware vorliegt und bereits verifiziert wurde.

2. Beschreiben Sie den typischen ASIC Design-Flow.

- Design House wählen
ASIC Design benötigt viel Wissen und Erfahrung. Zudem benötigt man die richtige Infrastruktur. Die Tools sind dabei extrem teuer ($10^4 - 10^7$ Dollar!). Deshalb werden die Lizenzen oftmals gemietet oder geteilt.
- Technologie wählen:
Hat Einfluss auf die Performance, Kosten der einzelnen ASIC und Kosten für ein Maskenset.
- Digitale Zell Bibliothek (cell library) wählen:
Werden oftmals vom ASIC-Hersteller angeboten. Mit ihnen kann man Einfluss auf die Schnelligkeit, Energiebedarf, low-power, etc. nehmen.
- Hard- und Soft IP-Blöcke auswählen
- Constraints hinzufügen (funktioniert genau gleich wie bei FPGA)
- Test Logik bei der Synthese:
ASIC hat keine eingebaute features wie der FPGA. Es sind mehrere zusätzliche Prüfschritte notwendig.
- Verifikation der ASIC-Synthese:
Bei der ASIC-Synthese wird neue Hardware generiert. Da der Produktionsprozess sehr teuer und zeitaufwendig ist, muss eine Verifikation nach der Synthese vorgenommen werden. Man schaut, ob das synthetisierte Design das gleiche Verhalten aufweist und macht eine statische Timing-Analyse.
- Floorplaning:
Hier wird definiert, wo die funktionalen Blöcke auf dem ASIC platziert werden.
- Bestimmung der I/O:
Werden in einem File bestimmt und vom Tool gelesen.

- Place and Route:
Place and Route Tools sind extrem teuer. Häufig wird dies deshalb von externen Firmen übernommen. Beim ASIC müssen mit CAD-Tools Place and Route gemacht werden, zusätzlich müssen Spannungsversorgung, etc. geplant werden. Zusätzlich muss geprüft werden, ob die physikalischen Abstände (Metall-Metall) eingehalten werden (Design Rule Check) und extrahierte Netzliste mit Source Netzliste vergleichen (LVS: Layout vs. Schematic). Dies ermöglicht das Erkennen von Kurzschlüssen, offenen Kontakten, etc.. Auch das Timing (statische Timing-Analyse und 2nd und 3rd Effekte) muss nochmals geprüft werden.
- Tape-out:
Wird der Prozess genannt, bei dem ein ASIC in die Produktion geht. Das Datenformat der Auslieferung ist GDSII (Graphical Data System).
- Schaltung verifizieren:
Zuerst muss ein Prototyp gemacht werden. Dazu muss zuerst eine Maske produziert werden.
- Endprodukt verifizieren:
Jeder ASIC ist ein neuer Komponent und muss deshalb getestet werden, bevor er ausgeliefert wird. Dies ist ein extrem wichtiger Schritt.

3. Listen Sie Gründe auf, warum die beiden Design-Flows nicht dieselben sind.

- Bei dem ASIC-Design hat man mehr Auswahlmöglichkeiten und darum mehr Freiheit. Firmen beschränken diese jedoch oftmals.
- Bei der Synthese gibt es grosse Unterschiede. Beim ASIC kann offener Einfluss genommen werden. Dies bedeutet aber viel Zeitaufwand.
- Der behavioral-Test ist bei beiden gleich. Beim ASIC muss man jedoch nach der Synthese nochmals mehrere Tests machen.
- Das Floorplaning entfällt beim FPGA, da er als fertige Hardware vorliegt. Beim ASIC muss man dies explizit machen. Dies bedeutet zusätzlicher Aufwand.
- Beim FPGA übernimmt die Implementierung das Synthesetool. Beim ASIC sind viele Zusatzschritte und extrem teure Place and Route Tools notwendig. Zusätzlich müssen beim ASIC nach dem Place and Route nochmals statische Timing-Analysen, Äquivalent-Tests und Verifikation auf 2nd und 3rd Effekte gemacht werden.
- Bei FPGA sind fixe Clockbäume vorhanden. Beim ASIC müssen diese definiert werden.
- Verifizierung der Schaltung ist auf dem FPGA relativ einfach (Debug-Core, JTAG-Schnittstelle), beim ASIC ist sie schwieriger.

4. Sie wissen, was die individuellen Schritte im Design-Flow für ein FPGA-Design und für ein ASIC-Design sind.
siehe oben.