Freie Universität Berlin
Fachbereich Informatik und Mathematik
Institut für Informatik

Bachelorarbeit

Nutzung und Möglichkeiten der High Level Synthese für einen FPGA

Erstellung eines IP Core zur Ansteuerung von externen Komponenten mittels des Xilinx ZYNQ 7000 AP SoC auf Basis des Red Pitaya

Lars Hochstetter

lars.hochstetter@fu-berlin.de

Betreuer:
Prof. Dr.-Ing. Jochen Schiller
Dr. rer. nat. Enrico Köppe

Zusammenfassung

Die High Level Synthese für FPGAs erlaubt die einfache Implementierung komplexer Algorithmen. Im Vergleich zur herkömmlichen Entwicklung mit HDLs wie VHDL oder Verilog kommen hier Hochsprachen wie C, C++ oder SystemC zum Einsatz, die eine Implementierung auf abstraktem, hardwarefernem Niveau erlauben. Die High Level Synthese erledigt dann beispielsweise das Scheduling von Operationen oder die Erzeugung von endlichen Zustandsautomaten zur Steuerung.

Entwickler können somit die hohe Rechenleistung des FPGAs nutzen, ohne dass viel Entwicklungszeit in die Steuerung der unterliegenden Hardware fließt. Ein FPGA wird traditionell auch in Bereichen eingesetzt, die eine hohe, hardwarenahe I/O Leistung voraussetzen, beispielsweise die digitale Signalverarbeitung.

Im Rahmen dieser Arbeit wird untersucht, ob und wie gut sich die High Level Synthese für hardwarenahe I/O Aufgaben eignet. Dafür werden eine Reihe IP Cores mit steigender funktionaler Komplexität implementiert, welche LEDs ein- und ausschalten und Taster auslesen. Als Testhardware wird das netzwerkfähige Messgerät Red Pitaya genutzt, das einen Xilinx ZYNQ 7000 AP SoC nutzt. Der Red Pitaya verfügt unter anderem über acht durch den Entwickler steuerbare LEDs sowie einen GPIO Header zum Anschließen von Tastern.

LEDs können abstrakt als ein ein Bit breiter Ausgang und Taster als ein ein Bit breiter Eingang betrachtet werden. Die AD und DA Wandler des Red Pitaya können als n Bit breite Ein- und Ausgänge betrachtet werden, daher wird die Ansteuerung dieser mittels High Level Synthese skizziert. Ferner wird ein allgemeiner Überblick über FPGAs und die Anwendungsentwicklung für diese gegeben, sowie Red Pitaya und Xilinx ZYNQ 7000 AP SoC vorgestellt.

Als Ergebnis wurde die Eignung der High Level Synthese für hardwarenahe I/O Aufgaben bestätigt. Beim Einsatz von C basierter High Level Synthese gibt es durch die starke Abstraktion von Hardware und Funktionalität Probleme: das Warten für eine Zeitspanne im FPGA konnte nur implizit realisiert werden, da in C die nötigen Sprachkonstrukte fehlen, die ein direktes Abgreifen des Taktsignales ermöglichen. Recherchen zufolge besteht hier eine mögliche Alternative in der Nutzung von SystemC basierter High Level Synthese.

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Ausarbeitung mit dem Titel „Nutzung und Möglichkeiten der High Level Synthese für einen FPGA Erstellung eines IP Core zur Ansteuerung von externen Komponenten mittels des Xilinx ZYNQ 7000 AP SoC auf Basis des Red Pitaya" selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Zitate sind als solche durch Anführungszeichen und Quellenangabe gekennzeichnet. Diese Arbeit wurde bisher weder in dieser noch in einer ähnlichen Form einer anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, den 09.02.2017 _____

Lars Hochstetter

Danksagung

Ich danke folgenden Personen die mich im Rahmen dieser Arbeit unterstützt haben:

Prof. Dr.-Ing. Jochen Schiller und Dr. rer. nat. Enrico Köppe für die weitreichende Betreuung dieser Arbeit.

Meinen Eltern, Luca Keidel, David Bohn, Theresa Mersini, meinen Freunden und Kommilitonen für das Korrekturlesen, Vorschläge, Kritik und Zusprache.

Spezieller Dank geht an Mitglieder der Red Pitaya und Xilinx Nutzerforen: die Moderatoren Nils Roos (Red Pitaya) und debrajr (Xilinx) und die Forumsnutzer Pavel Demin (Red Pitaya) und u4223374 (Xilinx) haben mir bei Fragen zur Funktionsweise von Red Pitaya und High Level Synthese schnell und zielführend weitergeholfen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einführung

Die High Level Synthese für FPGAs vereinfacht die Implementierung von komplexen Algorithmen im Vergleich zu der herkömmlichen Entwicklung mit **H**ardware **D**escription **L**anguages (HDL). Dies wird durch eine hohe Abstraktion von Funktionalität und deren Umsetzung im FPGA erreicht [34, S. 39 ff][61, S. 7, 8][68, S. 5, 6].
FPGAs eignen sich auf Grund ihrer hohen Rechenleistung zur Beschleunigung von Berechnungen, die mithilfe der High Level Synthese einfach genutzt werden kann. Traditionelle Anwendungsgebiete wie die digitale Signalverarbeitung setzen aber auch hohe hardwarenahe I/O (**I**nput/**O**utput, Eingabe/Ausgabe) Leistung voraus [33][37].

HDLs wie Verilog oder VHDL beispielweise bieten entsprechende Sprachelemente um hardwarenahe I/O einfach zu realisieren, wie beispielsweise die Ausführung einer Operation bei jeder Flanke eines Taktsignales. Die High Level Synthese abstrahiert aber gerade Eigenschaften wie Taktsignal und Scheduling von Operation vom Entwickler weg [37][61, S. 19]. Daher stellen sich folgende Fragen:

Können mit der High Level Synthese hardwarenahe I/O Aufgaben realisiert werden?

Welche Probleme können bei der Realisierung von Funktionalität entstehen?

Wie gut eignet sich die High Level Synthese für hardwarenahe I/O Aufgaben?

In dieser Arbeit werden die Nutzung und Möglichkeiten der High Level Synthese zur Ansteuerung von externen Komponenten am Beispiel der Ansteuerung von LEDs und Tastern demonstriert. Als Basis dient das Messgerät Red Pitaya, das ein Modell der Xilinx ZYNQ 7000 AP SoC Produktreihe nutzt. Die Software des Red Pitaya ist Open Source und für den ZYNQ stehen kostenlose Entwicklungswerkzeuge zur Verfügung. Ferner gibt es für den Red Pitaya und den ZYNQ Nutzerforen zum Wissensaustausch.

Zuerst werden der Red Ritaya und FPGAs allgemein vorgestellt. Anschließend wird die Xilinx ZYNQ 7000 AP SoC Produktreihe vorgestellt. Es folgt ein Überblick über die Anwendungsentwicklung für FPGAs mit Fokus auf die High Level Synthese. Bei der Implementierung werden unterschiedliche Möglichkeiten vorgestellt, eine LED anzusteuern: zuerst wird eine einzelne LED mittels Software ein- und ausgeschaltet und dann mit einer gegebenen Frequenz blinken gelassen. Ein einfacher Binärzähler wird durch Taster realisiert und der aktuelle Wert wird als eine Kombination aus an- und ausgeschalteten LEDs dargestellt. Die Blinkfunktion und das An- und Ausschalten über Software und Taster wird schließlich zusammengefasst und so realisiert, dass jede LED unabhängig von den anderen LEDs eine der genannten Funktionen ausführen kann. Auf dieser Grundlage wird kurz die Ansteuerung komplexerer Komponenten, beispielsweise einem AD oder DA Wandler, skizziert. Abschließend wird ein Fazit formuliert.

Im Anhang finden sich Anleitungen zum Aufsetzen der Entwicklungsumgebung, dem Erstellen, Einbinden in das bestehende Red Pitaya System und Steuern der implementierten Software. Weiterhin wird der Quellcode der Red Pitaya API analysiert, da dieser neben den, durch die Red Pitaya Entwickler bereitgestellten, Beispielen als Vorlage dient.

Der Quellcode, der im Rahmen dieser Arbeit geschrieben wurde, findet sich unter `https://git.imp.fu-berlin.de/lhochstetter/bachelor-thesis.git`.

2. Red Pitaya

Der Red Pitaya ist ein netzwerkfähiges Messgerät auf Basis eines Xilinx ZYNQ 7010 SoCs, bestehend aus einer ARM CPU und einem FPGA [11, S. 2][57, S. 1, 5]. Durch die Flexibilität des FPGAs können viele unterschiedliche Funktionen wie ein Oszilloskop, ein Signalgenerator, ein LCR Meter oder ein SDR Receiver / Transceiver realisiert werden [8][24].

Hardware

Der Red Pitaya $v1.1$ verfügt über 512MB DDR3 RAM und einen Slot für micro SD Karten, der passende Karten mit dem Betriebssystem aufnehmen kann. Für die Messfunktionen werden ein **A**nalog-**D**igital (AD) und ein **D**igital-**A**nalog (DA) Wandler eingesetzt. Ein AD Wandler wandelt eine analoge Spannung in einen digitalen Wert um [6]. Ein DA Wandler bildet das Gegenstück zum AD Wandler und wandelt einen digitalen Wert in eine analoge Spannung um [10].
Als Anschlüsse stehen zwei SATA Ports, die der Verbindung mehrerer Red Pitayas dienen, zwei GPIO Leisten, zwei mirco USB Slave Ports (Spannungsversorgung und serielle Konsole), ein USB Host Port, ein Ethernet Port, ein JTAG Header und jeweils zwei analoge Ein- und Ausgänge für den AD und DA Wandler zur Verfügung [25]. Ferner gibt es zwölf LEDs von denen acht durch den Nutzer steuerbar sind [11, S. 6]. Abbildung 1 zeigt den Red Pitaya als Schema und markiert die benannten Bauteile.

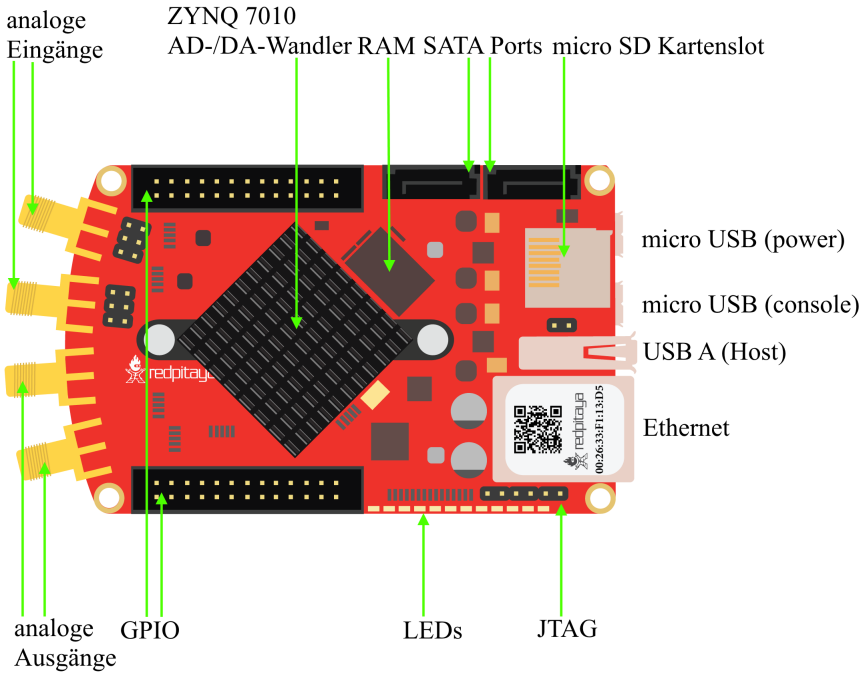

Abbildung 1: Red Pitaya als Schema [21][23]

Software

Das eingesetzte Betriebssystem, wahlweise Ubuntu und Debian, basiert auf Linux. Bei der Anwendungsentwicklung kann auf eine API zurückgegriffen werden, welche die Interaktion mit der im FPGA realisierten Funktionalität erlaubt. Die oben angesprochenen Anwendungen sind als eine Kombination aus Web Interface, dynamischer Bibliothek und im FPGA realisierter Funktionalität aufgebaut. Hierbei dient das Web Interface der Steuerung der Anwendung, Einstellungen und Messwerte werden über JSON mit einem NGINX Webserver ausgetauscht. Der Webserver verfügt über ein Modul, dass die Funktionen der Bibliothek der Anwendung aufruft. Die Bibliothek wiederum interagiert über die API mit dem FPGA [26]. Abbildung 2 stellt die Interaktion zwischen Web App, NGINX Server und Modul, API und FPGA Logik sowie einer Anwendung ohne Web Interface schematisch dar.

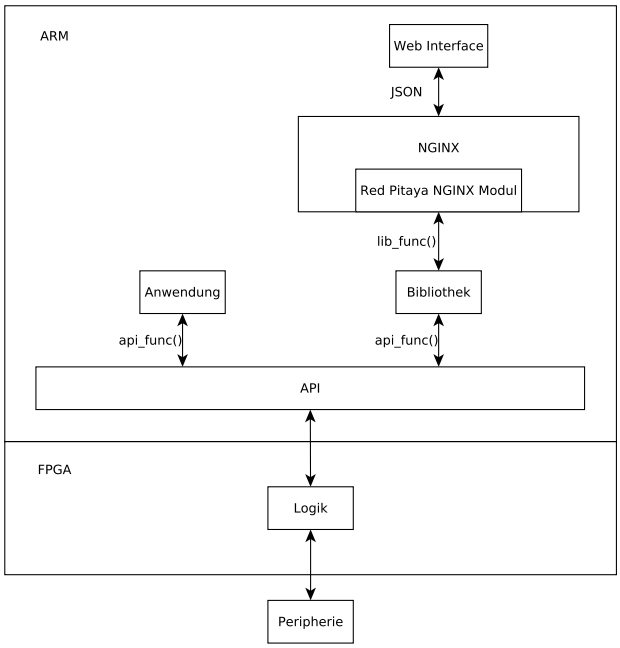

Abbildung 2: Unterschiedliche Interaktionsmodelle mit dem Red Pitaya [26]

3. FPGAs

Ein **F**ield **P**rogrammable **G**ate **A**rray (FPGA) bildet, ähnlich einem integrierten Schaltkreis (IC, **I**ntegrated **C**ircuit), eine Funktionalität in Hardware ab. Die Funktionalität eines FPGAs kann anders als beim IC nach der Herstellung im aktiven Einsatz noch angepasst werden [48][61, S. 11].

Die flexible Funktionalität schlägt sich aber in geringerer Performance, geringeren Taktfrequenzen, höherer Leistungsaufnahme, höherem Flächenbedarf (bis zu Faktor zehn) und einem höheren Preis (in großen Stückzahlen) im Vergleich zu einem **A**pplication **S**pecific **I**ntegrated **C**ircuit (ASIC) mit fester Funktionalität nieder [34, S. 36, 37][48][51, S.14].

Trotz der genannten Nachteile kommen FPGAs in Supercomputern, bei der Bildverarbeitung, in der Telekommunikation, bei der digitalen Signalverarbeitung und bei der Erstellung von ASIC Prototypen zum Einsatz [48][51, S. 13][70]. Dies liegt an der höheren Rechenleistung im Vergleich zu CPUs oder GPUs, da jeder Teil der Hardware eines FPGA echt parallel arbeitet und somit massiv paralleles Rechnen ermöglicht [51, S. 13][61, S. 25, 26]. Die Entwicklung von günstigeren und sparsameren FPGAs soll den höheren Preisen und der höheren Leistungsaufnahme entgegenwirken [34, S. 37][70].

3.1. Grundelemente

Ein FPGA besteht grundlegend aus einer zweidimensionalen Anordnung von einfachen Logikschaltungen (Gate Array). Diese Logikschaltungen können wiederum zu komplexen Funktionen verschaltet werden [49, S. 4][51, S. 12]. Abbildung 3 zeigt die zweidimensionale Anordnung der Logikschaltungen in einem einfachen FPGA. Die kleinen grauen Quadrate sind konfigurierbare Eingabe-/Ausgabepads verbunden mit den Pins, die schwarzen Linien zwischen den Logikschaltungen und Eingabe-/Ausgabepads sind programmierbare Verbindungen, die die Grundlage für die funktionale Flexibilität eines FPGA sind [61, S. 11].

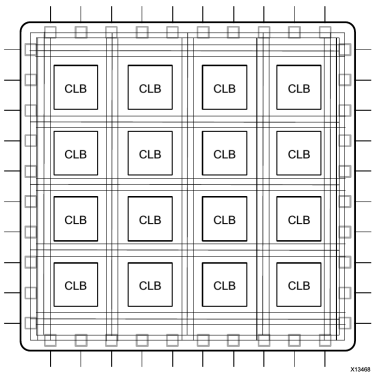

Abbildung 3: einfache FPGA Architektur [61, S. 12]

Configurable Logic Block

Der **C**onfigurable **L**ogic **B**lock (CLB, auch Slice [36], Logikzelle und Logikelement [15][51, S. 12] genannt) besteht grundlegend aus einer **L**ook **U**p **T**able (LUT) und einem Flip-Flop und fasst diese zu einer funktionalen Einheit zusammen. Er beinhaltet zusätzlich einen Multiplexer um den Flip-Flop zu überbrücken, wenn nur die LUT benötigt wird [36][51, S. 12]. Abbildung 4 zeigt einen einfachen CLB, bestehend aus einer 4-LUT, einem Flip-Flop (REG) und einem Multiplexer (MUX).

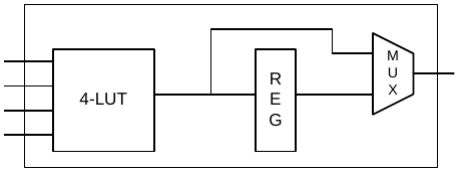

Abbildung 4: einfacher CLB [31, S. 1]

Die konkrete Architektur eines CLB ist von Hersteller, FPGA Generation und Familie abhängig [28, S. 8][36]. Abbildung 5 zeigt zwei komplexere CLBs von Xilinx und Altera.

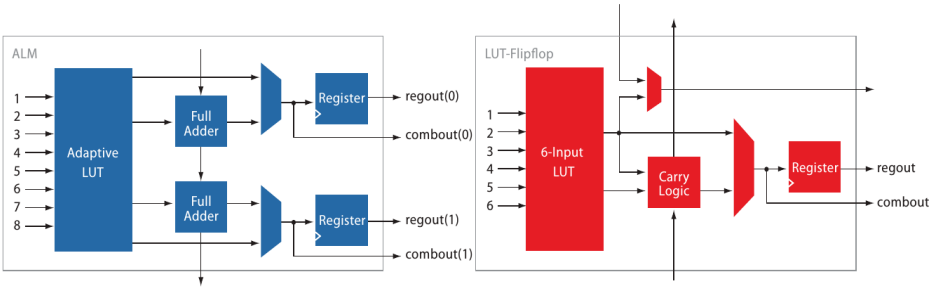

Abbildung 5: Altera Stratix II Adaptive Logic Module (links) und ein LUT Flip-Flop Paar aus einem Xilinx Virtex-5 (rechts) [28, S. 8]

Flip-Flop

Der Flip-Flop ist das einfachste Speicherelement im FPGA und speichert ein Bit zwischen zwei Takten [36][61, S. 15]. Der Flip-Flop ist flankengesteuert: der anliegende Wert wird bei einer (meistens positiven, logisch low auf logisch high) Flanke gespeichert. Es gibt unterschiedliche Flip-Flop Typen (JK, T, D), wobei der D Flip-Flop der am häufigsten genutzte ist [14][44].
Abbildung 6 stellt schematisch einen D Flip-Flop und die Funktionsweise dar. Die wichtigsten Pins sind **D** (Eingang), **Q** (Ausgang), **clock** (Taktsignal) und **clock enable**. In Takt 1 liegt an *D* der zu speichernde Wert, hier logisch *low*, an. Dieser Wert wird gespeichert, da an *clock* und *clock enable* logisch *high* anliegt. In Takt 2 wird der gespeicherte Wert an *Q* angelegt und der Wert, der dann an *D* anliegt, hier logisch *high*, gespeichert [44][61, S. 15].

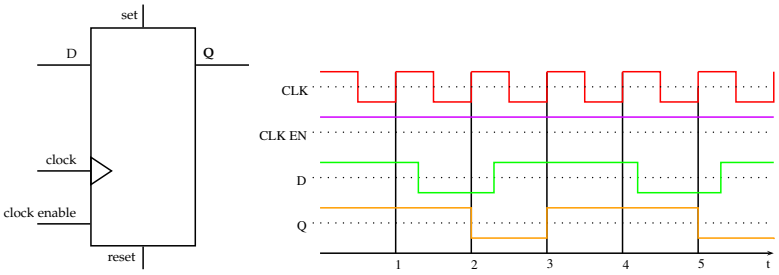

Abbildung 6: D Flip-Flop als Schema [61, S. 15] und Funktionsweise [44]

Latch

Der (SR, D, JK, Earle) Latch ist ein Sonderfall des Flip-Flops: der letzte Eingabewert wird für die folgenden Taktzyklen gehalten (gelatched), wenn am *clock enable* Pin logisch *low* anliegt. Von der Verwendung von Latches wird abgeraten, da sie schwierig als Hardware umzusetzen sind und ein ungünstigeres Zeitverhalten als Flip-Flops aufweisen [19][47]. Abbildung 7 stellt die Funktionsweise eines Latches dar, ab Takt 2 wird der an *D* anliegende Wert gelatched.

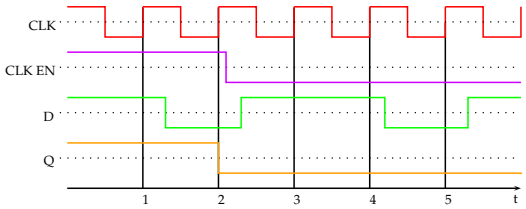

Abbildung 7: Funktionsweise eines D Latch

Look Up Table

Die LUT realisiert allgemein logische Funktionen mit n Eingabewerten und einem Ausgabewert. Die Funktion wird meist explizit als Wahrheitstabelle, die die 2^n Ergebniswerte der Funktion speichert, umgesetzt [36][45][61, S. 14]. Die LUT kann als Rechenwerk (Rechenleistung: ein Bit) oder als Speicher verwendet werden [61, S. 14, 19]. Mehrere LUTs können miteinander verschaltet werden, wenn eine Funktion mit mehr als n Eingabewerten realisiert werden soll [15][45]. Die Anzahl der Eingänge und Ausgänge einer LUT ist herstellerabhängig [15][28, S. 3,4].
In technischen Schaltungen sind LUTs aus SRAM Zellen und dahinter kaskadierten Multiplexern aufgebaut: die SRAM Zellen enthalten die Ergebniswerte der Funktion und die Eingabewerte schalten die Multiplexer so, dass zu jeder Eingabe der passende Ergebniswert ausgewählt wird. Es gibt aber auch LUTs die Logikgatter miteinander verschalten, um die gewünschte Funktion zu realisieren [15][28, S. 3].
Abbildung 8 zeigt eine SRAM basierte 2-LUT mit 2 Eingängen x_1, x_2 und einem Ausgang y, die Rechtecke stellen die Speicherzellen und die Trapeze die Multiplexer dar.

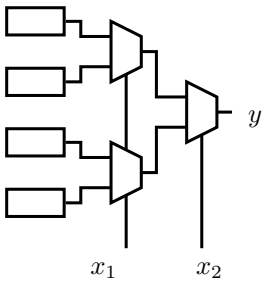

Abbildung 8: 2-LUT [61, S. 14]

Configurable Input / Output Block

Der Configurable **I**nput / **O**utput Block (I/O Block) verbindet die physischen Pins des FPGA mit der im FPGA realisierten Funktionalität. Die I/O Blöcke unterstützen unterschiedliche I/O Standards und können beispielsweise in der Polarität, der Flankensteilheit und der Spannung konfiguriert werden. Pins, die die gleichen I/O Standards unterstützen, werden zu sogenannten Bänken zusammengefasst [15][51, S. 13][70]. Abbildung 9 zeigt einen Xilinx I/O Block 4000.

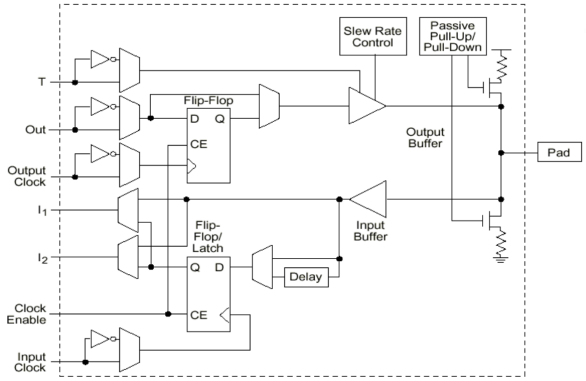

Abbildung 9: Xilinx I/O Block 4000 [60, S. 18]

Programmable Interconnect und Wire

Programmable Interconnects (auch **P**rogrammable **S**witch **M**atrizes, PSM) und Wires verbinden die einzelnen FPGA Elemente miteinander. Die Performance und Latenz einer im FPGA realisierten Funktionalität ist direkt davon abhängig, wie viele Programmable Interconnects zwischen die einzelnen CLBs geschaltet werden müssen [31, S. 7][70].
Die Wires verbinden die CLBs untereinander und die CLBs und I/O Blöcke mit den Programmable Interconnects [51, S. 12, 13][70]. Es unterschiedliche Arten von Wires: Die kurzen Wires verbinden räumlich nahe CLBs miteinander. Die lange Wires verbinden räumlich entfernte CLBs miteinander und können auch als Bus im FPGA genutzt werden. Die *global clock lines* stellen eine spezielle Variante der langen Wires da. Sie verbinden taktgesteuerte Elemente (Flip-Flops) mit dem Taktsignal [70]. Abbildung 10 zeigt mehrere Xilinx 4000 Interconnects (PSM), CLBs und Wires.

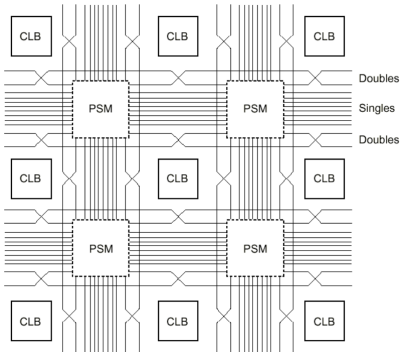

Abbildung 10: Xilinx 4000 Interconnect [60, S. 15]

Bei der Realisierung von Programmable Interconnects kommen unterschiedliche Technologien (AntiFuse, SRAM, EEPROM / Flash) zum Einsatz, welche im Folgenden kurz bezüglich ihrer Vor- und Nachteile vorgestellt werden [60, S. 7][71].

AntiFuse basierte FPGAs weisen einen geringen Energieverbrauch und geringe Latenzen auf, die Funktionalität kann aber nur einmal programmiert werden und ist danach fest. Sie sind auch schwieriger herzustellen [15][71].

SRAM basierte FPGAs können beliebig oft programmiert werden, haben aber den Nachteil, dass die SRAM Zellen flüchtig sind und bei jeder Unterbrechung der Stromversorgung neukonfiguriert werden müssen. Die Folge ist eine kurze Anlaufzeit, die je nach FPGA und Komplexität der Schaltung unterschiedlich lang ausfällt [15]. Die Neuprogrammierung ist auch ein Sicherheitsproblem, da bei der Konfiguration der Datenstrom manipuliert werden kann mit dem der FPGA programmiert wird. Auch der Energieverbauch und die Latenz sind im Vergleich zu AntiFuse FPGAs höher. Dennoch sind SRAM basierte FPGAs aufgrund ihrer vergleichsweise einfachen und günstigen Herstellung die am weitest verbreiteten FPGAs [71].

EEPROM / Flash basierte FPGAs können wie SRAM basierte FPGAs immer wieder neuprogrammiert werden. Die Programmierung bleibt auch bei einer Unterbrechung der Stromversorgung erhalten. Sie sind im Vergleich zu SRAM basierten FPGAs energieeffizienter und haben eine geringere Latenz. Sie sind trotz ihrer Vorteile gegenüber SRAM und AntiFuse basierten FPGAs weniger stark verbreitet [15][71].

3.2. Spezielle Funktionsblöcke

Neben den genannten Grundelementen gibt es noch spezialisierte Funktionsblöcke, auch Hard Cores oder Embedded Cores genannt [71][72]. Hard Cores sind in sich abgeschlossene Einheiten für zusätzliche, feste Funktionalität und erlauben eine effiziente Realisierung von beispielsweise Multiplikationen als die entsprechende Verschaltung von CLBs [36][49, S. 12][51, S. 21]. Sie bedeuten aber auch eine Abhängigkeit vom FPGA Hersteller, falls kein anderer Hersteller dieselben Hard Cores anbietet [72].

Hard Cores können physisch in den FPGA integriert, als SoC oder extern mit dem FPGA verbunden werden [49, S. 11, 12][72]. Häufig werden Mikrokontroller mit einem FPGA verbunden, um den FPGA im Betrieb neu konfigurieren zu können [15]. Abbildung 11 zeigt einen um Hardcores erweiterten FPGA.

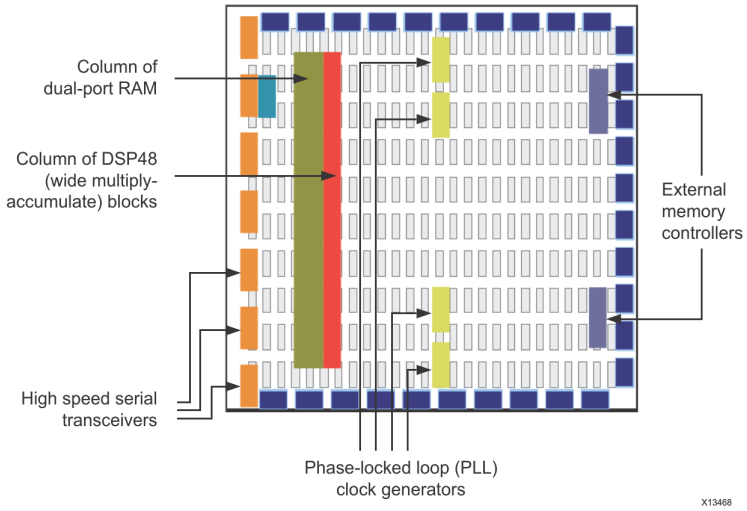

Abbildung 11: FPGA Architektur mit Hardcores [61, S. 13]

Integrierte Speicher

Integrierte Speicher gibt es in unterschiedliche Varianten und Größen, abhängig vom Verwendungszweck und Hersteller. Es gibt RAM / ROM Speicher, **F**irst **I**n **F**irst **O**ut (FIFO) Speicher und Shift Register [15][61, S. 16]. RAM Speicher gibt es als Single und Dual Port RAM, wobei Dual Port RAM zwei gleichzeitige oder asynchrone Lese-/Schreibezugriffe erlaubt. RAM wird mit Größen typischerweise zwischen 4 und 36 Kilobit verbaut [15][51, S. 23][61, S. 16]. ROM Speicher bezeichnet in diesem Fall Speicher, welcher zur Laufzeit nur ausgelesen werden, aber bei der Neuprogrammierung des FPGA mit entsprechenden Daten neu beschrieben werden kann [61, S. 16, 17]. FIFO Speicher können unter anderem genutzt werden, um Übergänge zwischen zwei Taktdomänen zu realisieren [41].

DSP Slice

Digital **S**ignal **P**rocessor Slices (DSP, auch Multiply Accumulate Block [61, S. 13]) sind darauf ausgelegt, Multiplikationen in einem Taktzyklus in Hardware auszuführen. Zusätzlich verfügen sie über Akkumulatoren, Übertragslogik und Flip-Flops um schnelle Addierer und Zähler zu realisieren [15][27, S. 14]. Die konkrete Architektur ist aber herstellerabhängig. Abbildung 12 zeigt links einen DSP von Altera und rechts einen DSP von Xilinx.

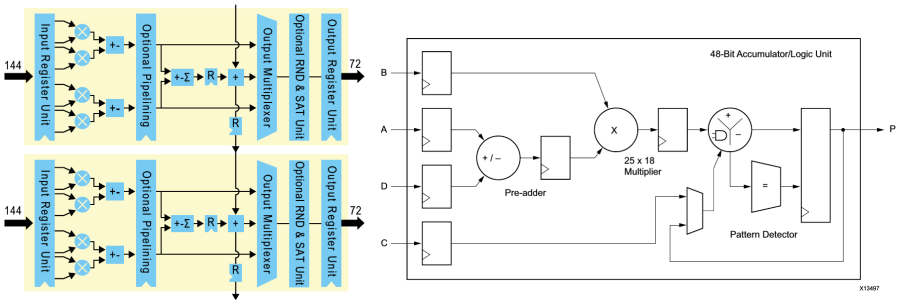

Abbildung 12: Altera DSP (links) [27, S. 14] und Xilinx DSP (rechts) [61, S. 16]

Phase Locked Loop

Ein **P**hase **L**ocked **L**oop (PLL) generiert Taktsignale unterschiedlicher Frequenz auf Basis eines Ausgangtaktes im FPGA, beispielsweise durch Taktvervielfachung oder Phasenverschiebung. Ein FPGA verfügt über mehrere Taktnetze für die verschiedenen Taktsignale [15][61, S. 13]. Der wichtigste Bestandteil einer PLL ist der spannungsgesteuerter Oszillator (**V**oltage **C**ontrolled **O**scillator, VCO). Dieser erzeugt die Frequenz F_{out} basierend auf der Referenzfrequenz F_{in}. Erzeugte Frequenz und Referenzfrequenz können durch Konstanten R (F_{in}) und N (F_{out}) geteilt werden. Beide werden in einen Phasendetektor gespeist, gefiltert und anschließend der VCO als steuernde Spannung zugeführt [39, S. 14 ff][51, S. 22]. Abbildung 13 zeigt schematisch eine PLL.

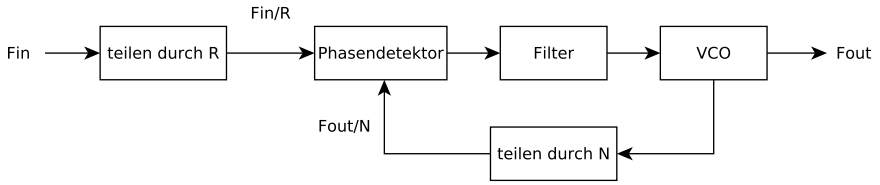

Abbildung 13: Funktionsschema einer PLL [39, S. 16]

4. Xilinx ZYNQ 7000 AP SoC

Architektur

Der ZYNQ ist ein **A**ll **P**rogrammable (AP) **S**ystem **o**n a **C**hip (SoC) und besteht aus einer ARM Cortex-A9 CPU (mit einem oder zwei Kernen) und einem FPGA. Das SoC wird in einer Strukturbreite von 28nm gefertigt [57, S. 1, 5]. Der FPGA nutzt modellabhängig die Architektur eines Xilinx Artix−7 oder Kintex−7 SRAM basierten FPGA [58, S. 2][62, S. 2][67, S. 213]. FPGA und ARM nutzen eine gemeinsame Verschlüsselungseinheit (AES, SHA, RSA), um die im FPGA realisierte Funktionalität gegen Manipulation durch Dritte zu schützen und ein sicheres Booten zu gewährleisten [67, S. 767ff.].
Abbildung 14 zeigt die Architektur des ZYNQ als Diagramm. Das **P**rocessing **S**ystem (PS) beinhaltet die ARM CPU, den AMBA Interconnect und einen Großteil der I/O Anschlüsse und Peripherieanbindung. Die **P**rogrammable **L**ogic (PL) besteht primär aus dem FPGA.

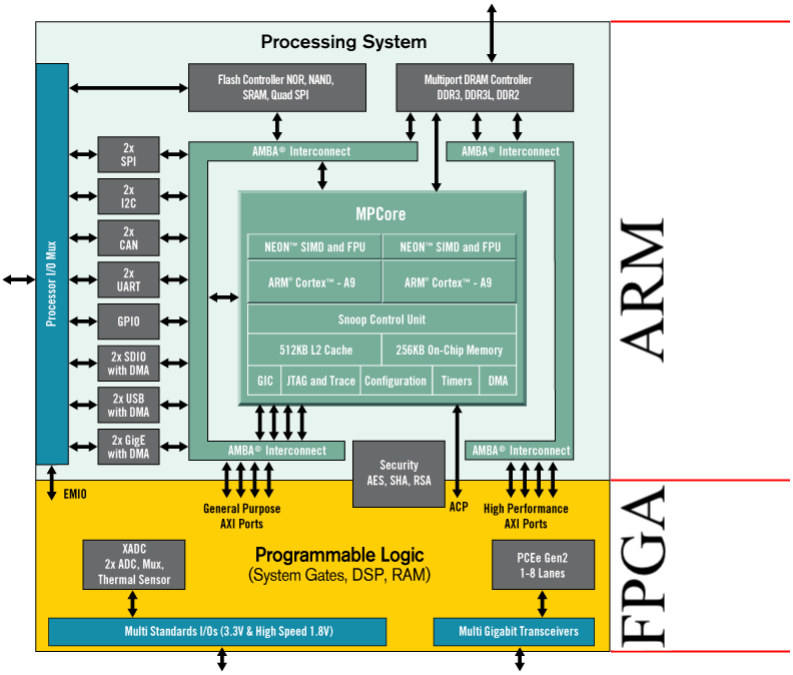

Abbildung 14: ZYNQ Architektur [54]

Tabelle 1 listet eine Auswahl der FPGA Ressourcen der ZYNQ Modelle (hier nur ARM CPUs mit zwei Kernen) auf [57, S. 3, 15][58, S. 2].

Modell	$XC7Z010$	$XC7Z015$	$XC7Z020$	$XC7Z030$	$XC7Z035$	$XC7Z045$	$XC7Z100$
LUTs	17600	46200	53200	78600	171900	218600	277400
Flip-Flops	35200	92400	106400	157200	343800	437200	554800
DSPs	80	160	220	400	900	900	2020
BRAMs	60	95	140	265	500	545	755
PLLs	2	3	4	5	8	8	8
	Low End			Mid Range			

Tabelle 1: FPGA Ressourcen der unterschiedlichen ZYNQ Modelle

ZYNQ Ultrascale+

ZYNQ Ultrascale+ ist der Nachfolger des ZYNQ 7000. Die SoCs werden mit 16nm Strukturbreite (FinFET+) gefertigt und bieten modellabhängig eine ARM Cortex-A53 CPU mit zwei oder vier Kernen, einen ARM Cortex-R5 Echtzeitprozessor mit zwei Kernen, eine ARM Mali-400 MP2 GPU und Anbindung für DDR4 RAM. Xilinx gibt an, dass die ARM CPU bis zu 2.7 mal mehr Performance pro Watt und der FPGA bis zu zweimal mehr Performance pro Watt verglichen mit ihren Gegenstücke im ZYNQ 7000 erreichen können [59, S. 1, 2].

AMBA Interconnect

AMBA (**A**dvanced **M**icrocontroller **B**us **A**rchitecture) ist eine Spezifikation von ARM und definiert in Version 3.0 die für den ZYNQ verwendet wird vier unterschiedliche Interfaces [7][67, S. 118]: AXI (**A**dvanced e**X**tensible **I**nterface), AHB (**A**dvanced **H**igh-performance **B**us), APB (**A**dvanced **P**eripheral **B**us) und ATB (**A**dvanced **T**race **B**us) [53].
Im ZYNQ verbinden AMBA Interconnects ARM CPU, Peripherie, Speicher und FPGA [62, S. 2][67, S. 118ff.]. Als Interface kommt primär AXI zum Einsatz: es gibt vier General Purpose Ports (zwei Master, zwei Slave) , vier High Performance Slave Ports speziell für DMA Speicherzugriffe durch den FPGA und ein Cache Coherency Port für den FPGA, der Zugriff auf den L2 Cache der ARM CPU erlaubt [67, S. 55, 90, 118ff., 250]. Im Anhang unter Abschnitt A.4 sind in Abbildung 24 die Interfaces farblich markiert.

AXI

AXI wurde erstmals in AMBA 3.0 als AXI spezifiziert und in AMBA 4.0 mit AXI4 erweitert [66, S. 5]. Das Interface erlaubt eine einfache Verbindung von Hardware und Software Komponenten, verfügt über einen großen Funktionsumfang und gute Modularität [53][66, S. 6].
Das AXI Interface definiert ein Interface zwischen dem sogenannten AXI Master und AXI Slave in einer $1:1$ Beziehung. Es sind aber auch $n:m$ Beziehungen über einen AXI Interconnect möglich, welcher unter anderem unterschiedliche Taktraten, Datenbreite und Protokollversionen (AXI3, AXI4, AXI4 Lite) aneinander anpasst [53][66, S. 31ff.]. Abbildung 15 zeigt schematisch einen AXI Interconnect mit zwei Mastern und drei Slaves.

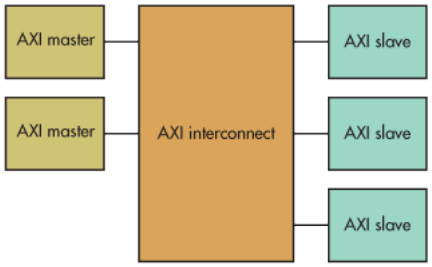

Abbildung 15: ZYNQ AMBA Interconnect Architektur [53]

Die Komplexität eines Interconnects richtet sich nach der Anzahl der Master und Slaves und der anderen Parameter wie Protokollversionen oder Taktraten. Neben AXI gibt es noch AXI Lite, eine leichtgewichtige Variante von AXI für Anwendungen, die nicht den vollen Umfang von AXI brauchen und AXI Stream, um Datenströme ohne komplexe Steuermechanismen zu realisieren [53][66, S. 5, 9].
Im FPGA Teil des ZYNQ kann AXI4 verwendet werden, trotz AMBA Version 3.0 auf ARM Seite. Xilinx stellt unterschiedliche AXI Verbindungen zu Verfügung, unter anderem einen AXI Interconnect, wobei diese im FPGA realisiert werden und die Konvertierung von dem auf ARM Seite genutzten AXI3 und dem im FPGA genutzten AXI4 übernehmen [55][56, S. 5][66, S. 9, 31ff.].

5. Anwendungsentwicklung

Die Anwendungsentwicklung für FPGAs begann auf der Ebene von Logikgattern, bei der Logikgatter miteinander verschaltet wurden, um die gewünschte Funktionalität zu erzeugen. Die nächste Stufe war die **M**edium **S**cale **I**ntegration (MSI [52, S. 4]), bei der ICs von Hand entwickelt, optimiert und schließlich miteinander verbunden wurden. Das Aufkommen von HDLs zur Simulation dieser ICs markiert den Übergang von einer rein hardwarebasierten zu einer zunehmend softwarebasierten Entwicklung [32, S. 1][50].
Die Entwicklung mit HDLs ist unter anderem durch die höhere Abstraktion von der Funktionalität und der Umsetzung dieser als Schaltung und die Verwendung von **I**ntellectual **P**roperty Cores (IP Cores) bedeutend einfacher als die auf Logikgatter oder MSI Ebene. Dennoch muss die Implementierung an Designvorgaben wie die angestrebte Taktfrequenz und die unterliegende Hardware angepasst werden [37].

Cores kapseln eine häufig genutzte Funktionalität als wiederverwendbare Einheit. Es wird zwischen den in Abschnitt 3.2 angesprochenen Hard Cores (liegen als Hardware / (AS)IC vor) und Soft Cores / IP Cores (liegen als Quellcode in HDL vor) unterschieden [71][72].

Durch die zunehmend softwarebasierte Entwicklung gleicht die Anwendungsentwicklung für FPGAs immer stärker derer für CPUs und GPUs: am Anfang jeder Anwendung steht die Anforderungserhebung. Aus dieser wird ein Design abgeleitet und implementiert, wobei auf bereits bestehende IP Cores, Frameworks und Bibliotheken zurückgegriffen wird, um die Entwicklung zu beschleunigen. Der Code wird wiederholt getestet und optimiert und gefundene Fehler werden behoben. Abschließend erfolgen Dokumentation und Freigabe [38][49, S. 17ff.]. Es gibt dennoch entscheidende Unterschiede bei der Codevalidierung und Kompilierung.

Code, der für eine CPU oder GPU geschrieben wird, wird durch einen Compiler in Maschinencode übersetzt. Die Funktion wird in eine Folge von Anweisungen in Maschinencode abstrahiert, die dann von der CPU oder GPU ausgeführt wird [38][42]. Die Abstraktion erlaubt ein hohes Maß an Flexibilität, hat aber geringere Performance und höhere Leistungsaufnahme zufolge [29, S. 8][42].
CPU und GPU können aber aufgrund ihrer festen Architektur als ASICs angesehen werden. Im Falle von solchen general-purpose ASICs wird auch von ASSPs (**A**pplication **S**pecific **S**tandard **P**roduct) gesprochen [40][49, S. 5].
Die Codevalidierung umfasst die Eingabe von Testwerten und der Vergleich der Ausgabe mit den erwarteten Ergebniswerten. Sie gestaltet sich auch einfacher als die für einen FPGA, da der Code einfach kompiliert und ausgeführt werden kann.

Code, der für einen FPGA geschrieben wird, beschreibt Hardware die gebraucht wird, um eine Funktion zu realisieren, welche dann als entsprechende Schaltung im FPGA umgesetzt wird. Daher wird auch von Hardwarebeschreibung oder Hardwarekonfiguration gesprochen [42]. Die Beschreibung der Hardware erfolgt aber in HDL Code, was, wie angesprochen, der Softwareentwicklung zuzuordnen ist. Die eindeutige Zuordnung zu Hardwareentwicklung oder Softwareentwicklung ist daher schwierig [15].

Die Codevalidierung für einen FPGA bedeutet die Simulation der Schaltung, die später im FPGA die Funktionalität umsetzen soll. Diese Simulationen werden auf einer CPU ausgeführt, was ihre Leistungsfähigkeit und Umfang einschränkt. Daher werden oft nur die kritischen Teile des Codes getestet [38][46][49, S. 22, 23]. Teilweise wird auf diese Tests auch verzichtet, da der FPGA in der Funktionalität auch im Einsatz angepasst werden kann [51, S. 14].

5.1. Von HDL Code zur Schaltung

Bei der Umsetzung von HDL in eine Schaltung werden grob zwei Schritte vollzogen: Synthese und Implementierung (auch Place and Route) [18]. Abbildung 16 zeigt die beschriebenen Schritte der FPGA Anwendungsentwicklung als Diagramm.

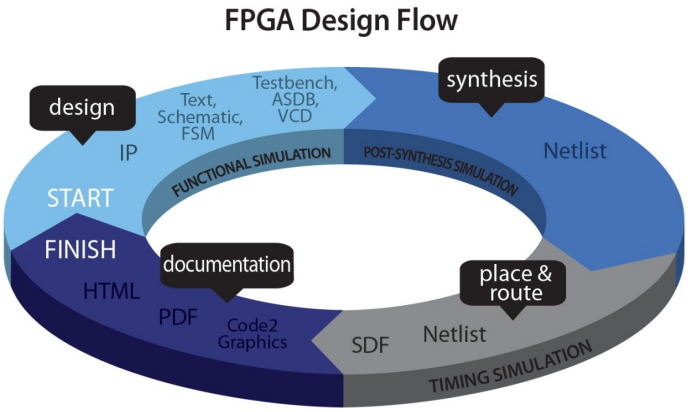

Abbildung 16: Klassische FPGA Entwicklung [17]

Die Synthese erzeugt (synthetisiert) aus dem HDL Code eine Netlist, welche die Verbindungen zwischen den einzelnen Elementen eines FPGAs beschreibt [18][38]. Da HDLs ursprünglich für die Validierung von Schaltungen entworfen und genutzt wurden enthalten sie Elemente, die nicht in eine Schaltung umgewandelt werden können. Das sind beispielsweise Befehle, die für eine Zeitspanne den Programmablauf anhalten oder dynamisch Speicher allozieren. Es wird auch von nicht synthetisierbarem Code gesprochen [38][43].

Die Implementierung übersetzt (mapped) die in der Netlist angegebenen Verbindungen und Elemente in konkrete FPGA Ressourcen (LUTs, Flip Flops, DPSs...). Es gibt verschiedene Unterschritte um beispielsweise den Energiebedarf oder Ressourcenverbrauch zu optimieren. Aus diesem Mapping wird der Bitstream generiert, der dann genutzt wird, um den FPGA zu programmieren [18][38].

Während der Synthese und der Implementierung werden die sogenannten Constraints genutzt, um funktionale Anforderungen und Einschränkungen an den Code zu formulieren. Diese können beispielsweise die Zeit festlegen, die eine Funktion maximal zur Ausführung brauchen darf oder ob eine Multiplikation als Verschaltung von LUTs oder als DSP realisiert werden soll [69, S. 7, 56]. Besonders das Einhalten vorgeschriebener Zeitquanta ist wichtig, da FPGAs oft in zeitkritischen Anwendungen zum Einsatz kommen [38][48].

5.2. High Level Synthese

Die High Level Synthese setzt auf eine noch stärkere Abstraktion im Vergleich zur HDL Entwicklung: die Funktionalität soll in einer Hochsprache wie C oder C++ entwickelt werden können, ohne dass dafür ein tieferes Verständnis der unterliegenden Hardware benötigt wird. Der Quellcode wird dann durch die High Level Synthese in HDL Code umgewandelt, welcher schließlich als Schaltung im FPGA umgesetzt wird [50][68, S. 5, 6]. Abhängig vom Anbieter kommen unterschiedliche Quellsprachen zum Einsatz: Xilinx [68, S. 5] bietet Unterstützung für C, C++, SystemC und OpenCL, Altera nutzt OpenCL [50]. Tabelle 2 stellt die Äquivalenzen der Sprachkonstrukte von Quellsprache (C, C++) und daraus synthetisiertem HDL Code (Verilog, VHDL) dar.

Quellsprache	synthetisierter HDL Code
Funktionen	Module
Funktionsargumente	Ein- / Ausgabeports
Operatoren	LUTs, DSPs, ...
Skalare	Flip Flops und Latches
Arrays	Speicherblöcke
Schleifen, *case*, *if else*	endlicher Zustandsautomat (**F**inite **S**tate **M**achine, FSM)

Tabelle 2: Äquivalenzen der Sprachkonstrukte [73, S. 1]

Spezielle Compilerdirektiven können zur Synthesezeit (hier ist die Umsetzung von Code in einer Hochsprache in HDL Code gemeint) genutzt werden, um den synthetisierten HDL Code in unterschiedlichen Aspekten zu optimieren (Ressourcenbedarf, Durchsatz). Nach Xilinx ist der Umstieg auf einen anderen FPGA oder die Veränderung der Taktfrequenz auch einfacher [68, S. 6, 7].

Die High Level Synthese erlaubt auch die einfache Portierung von bestehendem Code für CPU oder GPU auf einen FPGA. Altera demonstrierte dies anhand einer *Monte Carlo Black-Scholes* Simulation, wobei der FPGA, bei unverändertem Code, eine höhere Leistung bei gleichzeitig bedeutend geringerem Energieverbrauch erzielte. Die *Monte Carlo Black-Scholes* Methode wird im Finanzbereich eingesetzt, um den Prämienkurs von Aktien zu berechnen. Dabei werden Aktienpreise und der erwartete gemittelte Gewinn über mehrere Millionen zufällige Pfade simuliert [29, S. 7, 8].

Die gute Portierbarkeit von Code bedeutet wiederum höhere Effizienz und geringeren Zeitaufwand bei der Entwicklung. Der Aufwand für die Validierung des Codes wird gesenkt, da dieser einfach für eine CPU oder GPU kompiliert, ausgeführt und die funktionale Korrektheit validiert werden kann, ohne den Code in eine Schaltung übersetzen und diese dann simulieren zu müssen [61, S. 60, 61].
Abbildung 17 stellt das Verhältnis von Entwicklungszeit und erreichter Performance unterschiedlicher Technologien dar. Die Entwicklung mit HDLs auf **R**egister **T**ransfer **L**evel (RTL) erreicht zwar hohe Performance, die Entwicklungszeit ist aber bedeutend zu lang, während die Entwicklung mit High Level Synthese die gleiche Performance mit kürzerer Entwicklungszeit verspricht. Hervorzuheben ist hier auch das Verhältnis von FPGA und einer $x86$ CPU: die Entwicklungszeit für eine erste Version mit High Level Synthese ist nur geringfügig länger, bietet aber schon ähnlich viel Leistung wie die optimierte $x86$ Variante. Auffallend ist auch die höhere Leistung des FPGAs in der optimierten High Level Synthese Variante im Vergleich zur GPU Variante bei kürzerer Entwicklungszeit.

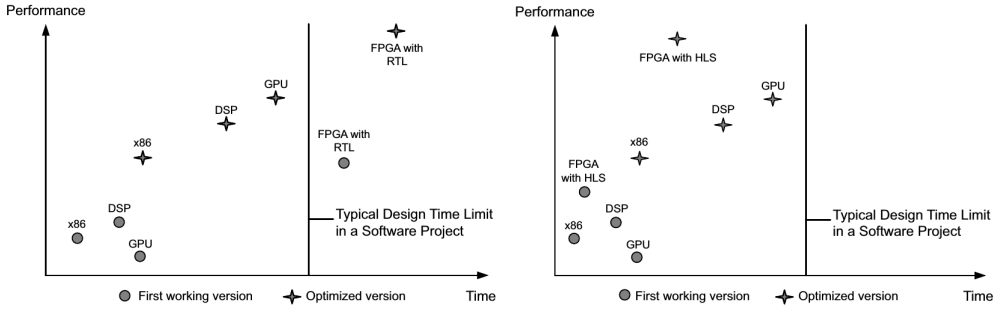

Abbildung 17: Zeit-Performance Verhältnis unterschiedlicher Technologien [61, S. 7, 8]

6. Implementierung

Die Ansteuerung der LEDs und Taster wird anhand von vier, in der funktionalen Komplexität wachsenden, IP Cores demonstriert: LED Switch, LED PWM, Push Button Counter und LED Controller. LEDs und Taster sind über Pins mit dem FPGA verbunden [11, S. 4 - 6]. Ein Pin kann abstrakt als ein ein Bit breiter Ein- oder Ausgang abhängig vom Verwendungszweck betrachtet werden. Jeder IP Core für die Ansteuerung der LEDs auf dem Red Pitaya muss daher über einen acht Bit breiten Ausgang verfügen, der mit den LEDs verbunden wird. Der Push Button IP Core braucht zusätzlich zwei, der LED Controller zusätzlich acht, ein Bit breite Eingänge, um die Taster anzuschließen. Ein Taster entspricht hierbei einem Bit. Um die IP Cores vom ARM aus steuern zu können, verfügen die implementierten IP Cores über einen AXI Port, der mit dem AMBA Interconnect (siehe Abschnitt 4) verbunden wird.

Komplexere Komponenten wie beispielsweise der AD oder DA Wandler des Red Pitaya sind über mehrere Pins mit dem FPGA verbunden [11, S. 3, 4]. Folglich werden hier n Bit breite Ein- und Ausgänge gebraucht, wobei n der Anzahl der verbundenen Pins entspricht. Datentypen in C haben Breiten, die ein Vielfaches von acht Bit sind. *Abitrary Precision Datatypes* erlauben von 1 bis zu 1024 Bit Breite Datentypen [68, S. 202, 203].
Listing 1 und 2 zeigen beide die *demo()* Funktion, die als Beispiel für die Ansteuerung einer Komponente mit einem elf Bit breiten Ein- und Ausgang dient. In Listing 1 werden C Datentypen genutzt und in Listing 2 werden *Abitrary Precision Datatypes* verwendet, um einen elf Bit breiten Ein- und Ausgang zu realisieren. Durch *Abitrary Precision Datatypes* können in diesem Fall fünf Bits und damit FPGA Ressourcen gespart werden.

Listing 1: Demo Code mit C Datentypen

```
1 void demo(short a, short *b)
2 {
3         ...
4 }
```

Listing 2: Demo Code mit *Abitrary Precision Datatypes*

```
1 #include <ap_cint.h>
2
3 void demo(int11 a, int11 *b)
4 {
5         ...
6 }
```

Abbildung 18 zeigt links den IP Core mit C Datentypen, rechts den mit *Abitrary Precision Datatypes*.

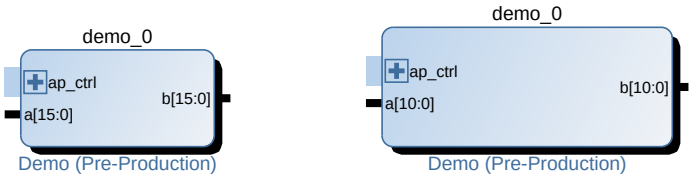

Abbildung 18: Nutzung von *Abitrary Precision Datatypes*

Alle IP Cores wurden für eine Taktfrequenz (Periodenlänge) von 100MHz (10ns) mit der standardmäßig eingestellten *Clock Uncertainty* von $12,5\%$ ausgelegt. Die *Clock Uncertainty* stellt ein Zeitbudget da, welches von den HDL Werkzeugen für Optimierungen genutzt werden kann, wird aber von der Periodenlänge bei der High Level Synthese abgezogen. Folglich dürfen die taktgesteuerten IP Cores (LED PWM, Push Button Counter, LED Controller) nur eine Ausführungszeit von $8,75$ns haben, damit sie rechtzeitig auf neue Eingabewerte reagieren können. Für die High Level Synthese (Vivado HLS) und die Umwandlung von HDL in eine Schaltung (Vivado) wird die Vivado Design Suite genutzt.
Speziell unter Berücksichtigung von Tabelle 2 werden Ausschnitte des synthetisierte Verilog Codes vorgestellt. Damit soll ein besseres Verständnis der Funktionsweise der High Level Synthese erreicht werden, da speziell bei den taktgesteuerten IP Cores bestimmte Funktionalität nur implizit umgesetzt werden konnte.

Die Grundlage für die Ansteuerung der LEDs bilden zwei Beispiele von Seiten der Red Pitaya Entwickler. Das Ziel bestand in der Verlagerung der Funktionalität vom ARM in den FPGA als einen mit High Level Synthese erstellten IP Core. Auf dem ARM soll eine entsprechende Steueranwendung den IP Core nur mit den benötigten Daten starten.
Als Quellsprache der High Level Synthese wurde C verwendet, da diese Sprache eine einfache Syntax und einen überschaubaren Sprachumfang besitzt. Die Portierbarkeit des Codes kann ebenfalls untersucht werden, da die Red Pitaya Anwendungen, die auf dem ARM ausgeführt werden und mit der FPGA Logik interagieren, in C programmiert wurden.

Das erste Beispiel lässt eine der durch den Nutzer steuerbaren LEDs auf dem Red Pitaya mit fester Frequenz blinken [9]. Daraus wurden folgende IP Cores abgeleitet: LED Switch und LED PWM. Der Beispielcode ist in Listing 3 dargestellt.

Listing 3: Beispielcode des LED Blinker [9]

```
int unsigned period = 1000000; // uS
...
int unsigned retries = 1000;
while (retries--){
    rp_DpinSetState(led, RP_HIGH);
    usleep(period/2);
    rp_DpinSetState(led, RP_LOW);
    usleep(period/2);
}
```

Das zweite Beispiel schaltet eine LED auf Tastendruck eines Tasters, der über die GPIO Leisten angeschlossen ist, an oder aus [20]. An dieser Stelle wird darauf hingewiesen, dass die LEDs beim zweiten Beispiel im Widerspruch zur Beschreibung dauerhaft an sind, ein Tastendruck schaltet die betreffende LED aus. Als einziger IP Core wurde der Push Button Counter IP Core aus dem Beispiel abgeleitet, der über zwei Taster einen Wert inkrementiert beziehungsweise dekrementiert und den Wert in Binärdarstellung mit den LEDs anzeigt. Der Beispielcode in Listing 4 dargestellt.

Listing 4: Beispielcode einer LED mit Schaltersteuerung [20]

```
rp_pinState_t state;
...
while (1) {
    for (int i=0; i<8; i++) {
        rp_DpinGetState (i+RP_DIO0_N, &state);
        rp_DpinSetState (i+RP_LED0, state);
    }
}
```

LED Switch

Der LED Switch IP Core kann eine beliebige LED an- und ausschalten. Der IP Core hat zwei Eingabeparameter: *led* und *state*. Die *led* Variable gibt an, welche LED auf den durch *state* angegebenen Zustand (aus: logisch 0 / an: logisch 1) gesetzt werden soll. Diese Angaben werden genutzt, um eine Bitmaske zu berechnen, welche in der statischen Variable *led_states* gespeichert wird. Die Variable wurde als statisch deklariert, um zwischen zwei Starts des IP Core den zugewiesenen Wert zu behalten. Der Wert von *led_states* wird schließlich in den Pointer *led_o* geschrieben. Der Pointer *led_o* wurde als *unsigned char* deklariert, da dieser Datentyp acht Bit breit ist und durch High Level Synthese in einen acht Bit breiten Port umgesetzt wird. Da nur schreibend auf *led_o* zugegriffen wird, wird *led_o* als reiner Ausgabeport umgesetzt.

Die *set_directive_interface* Direktive bündelt die Parameter *state*, *led* und die Steuersignale unter *ap_ctrl* zum AXI Lite Slave Port *ctrl*. In Listing 7 werden die in Abschnitt 5.2 angesprochenen Compilerdirektiven aufgeführt. Abbildung 19 zeigt links das Diagramm des synthetisierten IP Core, rechts denselben IP Core nach der Anwendung der Compilerdirektiven aus Listing 7. Für den *led_o* Port wurde automatisch das *ap_vld* Protokoll gewählt. Der daraus resultierende *led_o_ap_vld* Port wird aber bei der Anbindung an die LEDs nicht verbunden, da es keinen entsprechenden Eingangsport auf Seiten der LEDs gibt. Am *led_o_ap_vld* Port liegt logisch 1 an, wenn das Ergebnis valide ist, sonst logisch 0 [68, S. 84-87].

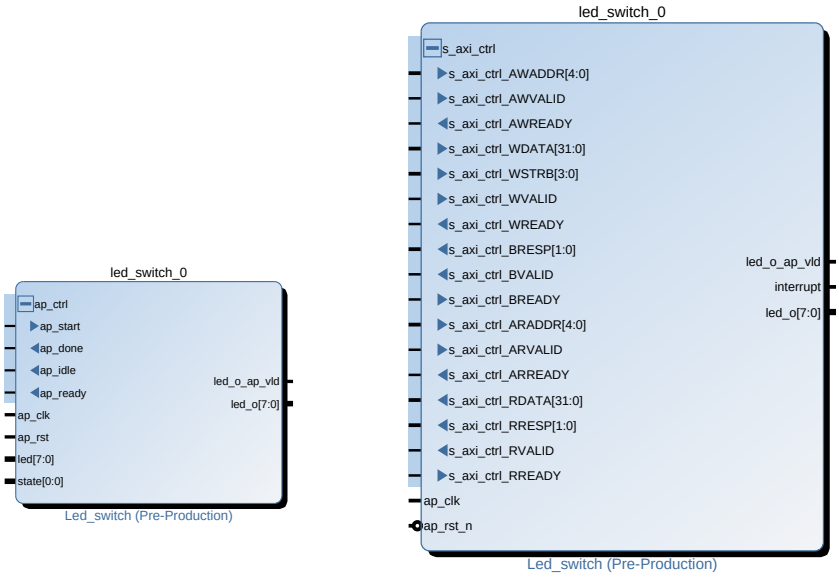

Abbildung 19: Auswirkungen von Compilerdirektiven

Listing 8 zeigt ausschnittsweise den synthetisierten Verilog Code. Die *led_switch* Funktion wur-

de als Modul umgesetzt. In Zeile 1 bis 24 werden alle Ein- und Ausgabeports deklariert, die auch in Abbildung 19 in der rechten Grafik zu sehen sind.
Der Pointer *led_o* wird in Zeile 26 als Ausgabeport deklariert. In Zeile 78 erfolgt die Zuweisung von *tmp_6_fu_81_p4* zu *led_o*, *tmp_6_fu_81_p4* wird in Zeile 32 deklariert und erhält in Zeile 84 bis 87 den neuen Wert für *led_o*. Zu beachten ist, dass *tmp_6_fu_81_p4* als *reg* $[31:0]$ deklariert wurde, eine 32 Bit breite Variable, die durch das *reg* Schlüsselwort als Latches umgesetzt wird und somit ihren Wert über mehrere Taktzyklen hält. Bei der Zuweisung in Zeile 78 werden aber nur die unteren acht Bit $[7:0]$ zugewiesen, da *led_o* nur acht Bit breit ist.
Die Variable *led_state* wurde durch das *static* Schlüsselwort als *reg* Variable übersetzt in Zeile 30 deklariert, in Zeile 36 initialisiert und in Zeile 72 bis 76 zugewiesen.
Die Funktionsparameter *led* und *state* werden als *wire* Variablen in Zeilen 28 und 29 deklariert. Durch die Bündelung mit den Steuersignalen als AXI Lite Slave Port treten sie aber nicht separat als Eingabeports auf. Sie werden aber in Zeile 68 und 69 zusammen mit den *ap_** Steuersignalen, Zeilen 63 bis 67, mit dem AXI Lite Slave Port verbunden, der in Zeile 39 bis 70 instanziiert wurde. Die Berechnung der Bitmaske erfolgt in Zeile 79 bis 82.

Hier zeigt sich erstmals die Macht der High Level Synthese: durch einige Compilerdirektiven wurde automatisch ein eigenes Modul für den AXI Lite Slave Port erzeugt und verbunden, welches sonst von Hand hätte entwickelt werden müssen. Zusätzlich wurde ein endlicher Zustandsautomat erzeugt, der die Steuerung des IP Core über die *ap_** Steuersignale übernimmt.

Tabelle 3 stellt den durch Vivado HLS geschätzten Ressourcenbedarf mit und ohne AXI Lite Slave Port gegenüber: mit AXI Slave Port werden bedeutend mehr Ressourcen benötigt. Somit muss bei der Ansteuerung eines IP Cores abgewogen werden, ob AXI genutzt werden soll, oder ob ein anderer Ansatz gewählt wird.

	ohne AXI	mit AXI
Flip-Flops	9	66
LUTs	28	86

Tabelle 3: Ressourcenbedarf des LED Switch IP Core

LED PWM

Der LED PWM IP Core realisiert die Pulsweitenmodulation durch blinkende LEDs. Pulsweitenmodulation (PWM, **P**ulse **W**idth **M**odulation) wird genutzt, um analoge Werte beliebiger Genauigkeit auf Basis von digitalen Werten zu erzeugen. Digital wird ein Rechtecksignal erzeugt, bei dem die logisch 0 und logisch 1 Phasen unterschiedlich lang ausfallen. Abhängig von der Dauer (Weite) der logischen 1 (Puls) können somit unterschiedliche analoge Werte erzeugt (moduliert) werden [35][30].

Das erste große Problem bei der Implementierung bestand in der Umsetzung der An-/ Auszeiten: die *usleep()* Funktion aus der Vorlage der Red Pitaya Beispiele gehört im Kontext der High Level Synthese zu den nicht synthetisierbaren Konstrukten [68, S. 297, 298]. Es ist nicht direkt möglich, eine bestimmte Zeitspanne im FPGA abzuwarten. Dies kann nur indirekt über eine

Zählung der Takte erfolgen. Abhängig von der Frequenz kann dann die gewünschte Zeitspanne als eine bestimmte Anzahl von Takten definiert werden [43]. Wenn beispielsweise eine LED Blinkfrequenz von einem Hertz mit jeweils einer halben Sekunde An- / Auszeit realisiert werden soll, muss bei der vorgegebenen Frequenz von 100MHz die Anzahl der jeweils zu wartenden Takte auf $50.000.000$ gesetzt werden.

In der ersten Iteration blinkte eine LED mit fester Frequenz. Listing 9 zeigt die erste, naive Umsetzung. Die *threshold* Variable gibt die Anzahl der Takte an, die die LED an oder ausgeschaltet sein soll. Die globale *clk* Variable wird als Eingang für das Taktsignal genutzt und da das Taktsignal als eine Folge von ein Bit breiten 0 und 1 Werten betrachtet werden kann, ist *clk* als *volatile bool* deklariert. Das *volatile* Schlüsselwort wird genutzt, um den Übergang von logisch 0 auf logisch 1 oder umgekehrt kenntlich zu machen. Der *if (clk) {...}* Block soll bei jeder logisch 1 Phase ausgeführt werden. Die *ticks* Variable zählt die Takte und wenn der *threshold* Wert erreicht wird, wird die LED entsprechend geschaltet. Die *switched_on* Variable speichert den aktuellen Zustand der LED, um zu verhindern, das auf *led_o* lesend zugegriffen werden muss, was die Umsetzung als Eingabe- und Ausgabeport zur Folge hätte.

Der resultierende IP Core funktionierte aber nicht wie erwartet, das Blinken der LED blieb aus. Nach einer Recherche im Xilinxforum [1] wurde das Problem im Abgreifen des Taktsignales gefunden: die C basierte High Level Synthese verfügt über keine Sprachkonstrukte, um das Taktsignal eines IP Core direkt abzugreifen. Ein möglicher Ausweg wäre die Nutzung von SystemC basierter High Level Synthese, da dort die nötigen Sprachelemente zur Verfügung stehen. Bei der C basierten High Level Synthese können die Takte nur implizit über die Anzahl der Funktionsaufrufe gezählt werden.

Die *blink_led()* (später *led_pwm()*) Funktion wurde umgeschrieben, damit sie bei jedem Taktsignal ausgeführt werden kann. Dafür muss das *Initiation Interval* (auch *Interval*, *II*) gleich eins sein [61, S. 22, 37]. Erreicht wird dies durch die *set_directive_pipeline -II 1* Compilerdirektive. Um beim LED PWM IP Core $II = 1$ zu erreichen musste zusätzlich ein *if-else-if* Konstrukt genutzt werden. Andernfalls wäre es nicht möglich, innerhalb eines Taktes zu entscheiden, welcher der *if* Zweige ausgeführt werden soll. Ferner fielen die *clk* Variable und der zugehörige *if* Block weg. Die *threshold_on* und *threshold_off* Variablen definieren die logisch 0 und logisch 1 Phasen.

Listing 10 zeigt ausschnittsweise den Verilog Code des LED PWM IP Core. Die drei *always* Blöcke sind auf die *set_directive_pipeline -II 1* Compilerdirektive zurückzuführen: der enthaltene Code wird bei jeder positiven Flanke des Taktsignales *ap_clk* ausgeführt. Im ersten *always* Block ist noch das angesprochene *if-else-if* Konstrukt zu erkennen.

Tabelle 4 stellt den durch Vivado HLS geschätzten Ressourcenbedarf mit und ohne $II = 1$ gegenüber. Der Ressourcenverbrauch ist etwa gleich groß, mit $II = 1$ werden mehr Flip-Flops benötigt.

	ohne $II = 1$	mit $II = 1$
Flip-Flops	218	241
LUTs	412	412

Tabelle 4: Ressourcenbedarf des LED PWM IP Core

Push Button Counter

Der Push Button Counter IP Core nutzt zwei über die GPIO Leisten angeschlossene Taster um einen Binärzähler zu realisieren. Der IP Core hat zwei Eingabeparameter: *inc_i* und *dec_i* realisieren das Auslesen des aktuellen Tasterwertes (gedrückt logisch 0, nicht gedrückt logisch 1, siehe Anmerkungen zum Beispiel). Die Parameter *inc_i* und *dec_i* sind als *bool* deklariert, um ein Bit breite Eingabeports zu erzeugen. Abbildung 20 zeigt den IP Core, zu erkennen sind die *inc_i* und *dec_i* Eingabeports, die an die Taster angeschlossen werden.

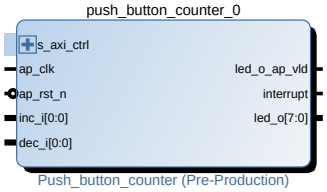

Abbildung 20: Push Button Counter IP Core

Wenn der Taster, der mit dem *inc_i* Port verbunden ist, gedrückt wird, wird die *value* Variable hochgezählt. Das Hochzählen wird bei einer positiven Flanke ausgeführt, die dann auftritt, wenn der Taster wieder losgelassen wird. Die positive Flanke wird mittels des *pos_edge_inc* Arrays erkannt, was den Tasterstand des vorigen Taktes speichert. Gleichzeitig darf der Taster, der mit *dec_i* verbunden ist nicht gedrückt sein, um *data races* zu verhindern. Zusätzlich muss der Wert von *value* unter 255 liegen, da es sonst zu einem Überlauf kommt. Dieser Überlauf resultiert in einem Ausschalten aller LEDs, da die Binärdarstellung von *value* genutzt wird, um den Wert von *led_o* zu setzen.

Für eine Betätigung des an *dec_i* angeschlossenen Tasters erfolgen die Tests spiegelverkehrt: *value* muss größer als 0 sein, um einen Unterlauf zu verhindern, welcher alle LEDs anschalten würde, der an *inc_i* angeschlossene Taster darf nicht gedrückt sein und es muss eine positive Flanke an *dec_i* gemessen worden sein. Die eigentliche Verbindung mit den GPIO Pins wird im Anhang A.6 beschrieben.

LED Controller

Der LED Controller fasst die Funktionen der vorangegangenen IP Cores in einem IP Core zusammen: LEDs können an- und ausgeschaltet werden, entweder durch einen angeschlossenen Taster oder durch ein Setzen des entsprechenden Wertes durch Software, die auf dem ARM läuft, PWM ist ebenfalls möglich. Der entscheidende Unterschied ist, dass der LED Controller jede LED einzeln ansteuern kann während die anderen IP Cores die ungenutzten LEDs ausschalten. Beispielsweise kann eine LED dauerhaft angeschaltet, eine weitere LED mittels PWM blinken gelassen und die restlichen sechs LEDs mit sechs Tastern gesteuert werden. Zusätzlich kann die Ansteuerung einer LED im laufenden Betrieb verändert werden. Ferner kann auch ein Anfangswert (an oder aus) für jede LED vorgegeben werden, in den anderen IP Cores waren alle LEDs von Anfang an aus.

Der IP Core hat zwei Eingabeparameter: *gp_i[LED_COUNT]* und *config[LED_COUNT]*. Das Array *gp_i[LED_COUNT]* stellt die Verbindung zu den Pins der GPIO Leiste her. Das Array *config[LED_COUNT]* vom Datentyp *led_config* enthält die Konfiguration für die einzelnen LEDs. Das Array *led_o[LED_COUNT]* wird mit den LEDs verbunden. Das globale Array *handles[LED_COUNT]* speichert Ausführungsdaten wie den Zählerwert *ticks* der PWM oder die aktuell ausgeführte Funktionalität *state*. *LED_COUNT* ist ein *#define* mit dem Wert 8, kann aber auch angepasst werden um mehr oder weniger als acht LEDs gleichzeitig anzusteuern.
Die Deklaration der genannten Variablen als Arrays ist nötig, um auf jeden Eintrag gleichzeitig zugreifen zu können: beispielsweise sind ein *unsigned_char* oder ein entsprechendes Bitfield auch acht Bit breit, bieten aber nicht die nötige Flexibilität bei der Implementierung von *led_o*, um den parallelen Zugriff zu realisieren. Das *config* Array und die Steuersignale werden als AXI Lite Slave Port gepackt, um den IP Core entsprechend parametrisieren zu können. Die Einträge aller Arrays werden mit der *set_directive_array_partition -type complete -dim 1* Compilerdirektive als eigenständige Ports umgesetzt, was wieder für den parallelen Zugriff notwendig ist. Wenn diese Direktive weggelassen wird, wird jedes Array mit zwei Ports versehen. Daraus ergibt sich dann mit der *set_directive_pipeline -II 1* Direktive ein Problem, da nicht genügend Bandbreite zur Verfügung steht, in einem Taktzyklus auf alle Einträge zugreifen zu können. Die *set_directive_pipeline -II 1* Direktive wird, wie oben angesprochen, für die PWM gebraucht.
Abbildung 21 zeigt den LED Controller IP Core. Gut zu erkennen sind die acht einzelnen Eingabe- und Ausgabeports der *gp_i* und *led_o* Arrays.

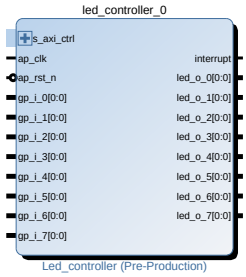

Abbildung 21: LED Controller IP Core

Um die Ansteuerung einer LED im laufenden Betrieb zu verändern, ohne die anderen LEDs zu beeinflussen, werden zwei *switch-case* Blöcke verwendet. Der äußere *switch-case* Block realisiert einen Zustandsautomaten mit den Zuständen *INIT*, *RUNNING* und *HALT*.
Der *INIT* Zustand ist der erste Zustand, der durchlaufen wird und ist dafür verantwortlich, alle benötigten Variablen zu initialisieren. Sobald dies geschehen ist wird *handles[i].state* auf *RUNNING* gesetzt und im nächsten Takt in den *RUNNING* Zustand gewechselt.
Im *RUNNING* Zustand wird geprüft, ob *config[i].reset* 1 ist. Wenn ja, dann wird *handles[i].state* auf *HALT* gesetzt und im nächsten Takt in den *HALT* Zustand gewechselt, ansonsten wird die gewählte Funktionalität ausgeführt. Die Auswahl der Funktionalität wird über den inneren *switch-case* Block realisiert, der abhängig von der *config[i].mode* Variable entsprechend die zuständige Logik auswählt und ausgeführt.
Die *CONSTANT* Funktionalität schaltet eine LED an oder aus. Da die LED schon im *INIT* Zustand auf den gewünschten Wert gesetzt wurde, muss kein weiterer Code ausgeführt werden.
Die *EXT_SWITCH* Funktionalität schaltet bei einer positiven Flanke an einem extern angeschlossenen Taster eine LED an oder aus.
Die *PWM* Funktionalität realisiert die PWM für eine LED, ähnlich dem LED PWM IP Core. Am Ende wird der Wert in *handles[i].toggle* auf die entsprechende LED in *led_o[i]* geschrieben.
Der *HALT* Zustand wird für die Veränderung der Ansteuerung genutzt, um beispielsweise zu verhindern, dass bei der PWM der Zähler auf einem Wert größer als die neue An-/Auszeit steht und dann erst überlaufen muss, um wieder bei 0 anzufangen. Wenn *config[i].reset* auf 0 gesetzt wird, wird wieder in den *INIT* Zustand gewechselt.

Die Zustände des Zustandsautomaten können nicht über einen entsprechenden Eintrag im *led_config* Struct verwaltet werden, da sowohl lesend als auch schreibend zugegriffen wird. Der Eintrag wird dann als zwei unterschiedliche Ports im AXI Lite Slave umgesetzt, was zur Folge hat, dass die Zustandsänderung im *INIT* Zustand nicht auf den selben Port geschrieben wird, wie die, die im ersten *switch-case* Block gelesen wird. Der Zustandsautomat muss daher im IP Core gesteuert werden und es muss auf die *config[i].reset* Variable zurückgegriffen werden, da diese nur gelesen wird.

Um alle LEDs parallel anzusteuern und um ein *II* gleich eins zu erreichen muss die angesprochene Logik in diesem Fall achtmal umgesetzt werden. Die *set_directive_pipeline -II 1* Direktive bewirkt das Ausrollen der *for* Schleife und die entsprechende parallele Umsetzung der Logik.

Der Quellcode des LED Controller IP Core ist auf den ersten Blick nicht intuitiv. Daher wurde die Funktionalität aus der *for* Schleife in die *handle()* Unterfunktion ausgelagert. Infolgedessen mussten die Arrays *gp_i*, *led_o* und *config* als globale Variablen deklariert werden, da keine Zugriffsmöglichkeit gefunden werden konnte, mit der ein *II* gleich eins und eine Ausführungszeit von unter 10ns erreicht und die Arrays weiterhin als Funktionsparameter von *led_controller()* erhalten werden konnten.
Um ein *II* von eins zu erreichen wurde die *set_directive_dataflow* Direktive genutzt, damit Vivado HLS die Unterfunktion bezüglich des Datenflusses analysiert. Diese Direktive hebt die *set_directive_pipeline -II 1* Direktive auf, daher wurde die *set_directive_pipeline -II 1* Direktive auf die *handle()* Unterfunktion angewendet, um ein *II* von eins zu erreichen. Ferner musste das Ausrollen der *for* Schleife manuell mit der *set_directive_unroll* Direktive vorgenommen werden. Weitere Auslagerungen von Code in Funktionen, beispielsweise den Code des *INIT* Zustand in eine eigene Unterfunktion und die *handle()* Unterfunktion nur mit dem Code des *RUNNING* Zustandes, wurden ebenfalls kurzeitig untersucht, dann aber verworfen, nachdem Vivado HLS während der Analyse des Datenflusses wiederholt abgestürzt ist.

Tabelle 5 stellt den durch Vivado HLS geschätzten Ressourcenbedarf mit und ohne *handle()* Unterfunktion gegenüber. Der Ressourcenverbrauch ist etwa gleich groß, mit *handle()* werden geringfügig mehr Flip-Flops und LUTs benötigt.

	ohne *handle()*	mit *handle()*
Flip-Flops	1158	1204
LUTs	2227	2280

Tabelle 5: Ressourcenbedarf des LED Controller IP Core

Ressourcenverbrauch der IP Cores

Tabelle 6 listet den Ressourcenbedarf (LUT, Flip-Flops) und die Ausführungszeit jedes IP Cores auf. Hierbei werden die Werte, die durch Vivado HLS geschätzt wurden und die Werte, die nach der Implementierung durch Vivado tatsächlich verbraucht wurden, gegenübergestellt. Auffällig sind die großen Unterschiede im Ressourcenverbrauch und Ausführungszeit speziell beim LED Controller IP Core (HLS Schätzung: 1204 LUTs, 2280 Flip-Flops, $9,47$ns Ausführungszeit; Implementierung: 972 LUTs, 1281 Flip-Flops, $6,324$ns Ausführungszeit). Auch der Anstieg in der Ausführungszeit nach der Implementierung bei den LED Switch (HLS Schätzung: $2,37$ns; Implementierung: $3,514$ns) und Push Button Counter (HLS Schätzung: $3,37$ns; Implementierung: $4,531$ns) IP Cores sind bemerkenswert. Ein genaue Ursache hierfür konnte nicht ermittelt werden.

		Flip-Flops	LUTs	Ausführungszeit (ns)
LED Switch (mit AXI)	HLS	66	86	$2,37$
	VE	41	48	$3,514$
LED PWM ($II = 1$)	HLS	241	412	$7,98$
	VE	235	181	$6,488$
Push Button Counter	HLS	47	89	$3,37$
	VE	30	47	$4,531$
LED Controller	HLS	1158	2227	$10,84$
	VE	933	1136	$6,891$
LED Controller (mit *handle()*)	HLS	1204	2280	$9,47$
	VE	972	1281	$6,324$

Tabelle 6: High Level Synthese (HLS) und Verilog Evaluation (VE) Ressourcenbedarf

Testumgebungen der IP Cores

Für die IP Cores LED Switch, LED PWM und Push Button Counter wurden Testbenches geschrieben um die korrekte Funktion zu validieren. Alle Testbenches erzeugten zunächst Fehler, was teilweise an einer Fehlfunktion der IP Cores lag, teilweise aber auch an einer falschen Kombination von Eingabeparametern und erwarteten Ergebnissen.

Die Testbench für den LED Switch IP Core ruft den IP Core mit allen möglichen Kombinationen für *led* und *state* auf und überprüft, ob die generierte Bitmaske mit der erwarteten übereinstimmt.

Probleme ergaben sich bei den Testbenches für die LED PWM und Push Button Counter IP Cores: beim LED PWM IP Core musste der Takt simuliert werden und beim Push Button Counter IP Core zusätzlich ein Tastendruck. Xilinx [61, S. 67] gibt Fälle an in denen die Nutzung einer C Testbench nicht möglich ist: wenn sich Parameter (hier Tastendruck und Taktsignal) außerhalb des IP Core verändern, muss eine RTL Testbench geschrieben werden, um die korrekte Funktionalität vollständig zu prüfen. Die RTL Testbench wird dann mit dem synthetisierten (hier High Level Synthese) IP Core verbunden und dieser simuliert.

Im Rahmen dieser Arbeit wurde aufgrund des Aufwands darauf verzichtet und eine Annäherung mit einer C Testbench realisiert. Da beide IP Cores so ausgelegt sind, dass die Funktionalität jeden Takt ausgeführt wird, wurde eine *for* Schleife genutzt, welche eine entsprechende Variable hochzählt, die den Takt simuliert. Die Funktion wurde in jedem Schleifendurchlauf aufgerufen. Im Fall des LED PWM IP Core wurde für die gegebenen *treshold_on* und *threshold_off* Grenzen geprüft, ob die erzeugte Bitmaske der erwarteten entspricht. Beim Push Button Counter IP Core wurden die Tastendrücke durch entsprechende Belegungen der *inc_i* und *dec_i* Parameter erreicht und ebenfalls der berechnete Wert verglichen.

Beim LED Controller wurde auf die Testbench verzichtet, da hier schon getestete Funktionalität verwendet wurde. Ein erster Ansatz für eine Testbench wurde aufgrund der unverhältnismäßig langen Simulationszeit bei der C / RTL Co-Simulation bei der Variante mit der *handle()* Unterfunktion verworfen. Die C / RTL Co-Simulation nutzt den synthetisierten HDL Code und die C Testbench, um die korrekte Funktionsweise nach der Anwendung der Compilerdirektiven zu testen [61, S. 65, 66].

Ein Ausblick auf die Ansteuerung der AD und DA Wandler

Die implementierten IP Cores für die Ansteuerung der LEDs und Taster können einfach erweitert werden, um die AD und DA Wandler auf dem Red Pitaya anzusteuern.

Wichtige Kenngrößen für den IP Core sind die Auflösung und die Sample Rate der Wandler: die Auflösung gibt an, wie viele Bits der digitale Wert hat. Je mehr Bits zur Verfügung stehen desto genauer kann im Falle des AD Wandlers die Spannung gewandelt werden. Beim DA Wandler gibt die Auflösung an, wie viele unterschiedliche Spannungslevel erzeugt werden können. Die Sample Rate gibt an, wie oft pro Sekunde ein Wert oder eine Spannung gewandelt werden kann [6][10].

Die AD und DA Wandler auf dem Red Pitaya haben eine Auflösung von 14 Bit und werden mit einer Frequenz von 125MHz betrieben. Da sie bei jedem Taktsignal samplen ergibt sich eine Sample Rate von 125 Mega Sample (Msps) [11, S. 2][13]. Der Takt wird durch einen 125MHz Oszillator auf dem Red Pitaya erzeugt, es gibt aber auch Möglichkeiten zur alternativen Taktansteuerung [12].

Beide IP Cores müssen daher für eine Taktfrequenz von 125MHz ausgelegt werden und ein *II* gleich eins erreichen. Ähnlich wie bei den oben vorgestellten IP Cores kommt ein AXI Lite Slave Port zur Steuerung zum Einsatz. Im Falle des IP Core für den AD Wandler wird ein 14 Bit Eingangsport benötigt, für den DA Wandler entsprechend ein 14 Bit Ausgangsport. Diese können durch die Verwendung von *Abitrary Precision Datatypes* erzeugt werden [68, S. 202, 203].

Das größte Problem besteht im Auslesen der Messwerte des AD Wandler und dem Bereitstellen der zu wandelnden Werte des DA Wandlers auf Seite des ARM: Der AXI Lite Slave Port kann genutzt werden, um ein Array fester Länge für die Werte in den durch den ARM zugreifbaren Adressraum abzubilden. Dadurch werden aber viele Adressen belegt und die Übertragungsgeschwindigkeit wird eingeschränkt, da hier das General Purpose AXI Interface genutzt wird.

Eine Alternative liegt in der Nutzung der High Performance Slave Ports: hierfür wird ein AXI Stream Master Port beim AD Wandler IP Core und ein AXI Stream Slave Port beim DA Wandler IP Core benötigt. Diese werden mit Xilinx DMA IP Cores verbunden, die DMA Zugriffe auf den RAM Speicher erlauben [65]. Dadurch können die Messwerte des AD Wandlers und die zu wandelnden Werte für den DA Wandler direkt im RAM abgelegt und durch den ARM weiterverarbeitet werden.

Abbildung 22 zeigt die Prototypen IP Cores für die Ansteuerung von AD und DA Wandler. Diese steuern jeweils einen Kanal des AD oder DA Wandlers an, um beide Kanäle des AD oder DA Wandlers anzusteuern werden entsprechend zwei Instanzen des IP Cores benötigt. Zu beachten ist, dass der DA Wandler mit doppelter Datenrate (DDR) bei 125MHz angesteuert wird, daher müssen ein entsprechend konfigurierter *xlconcat* und ein *SelectIO Wizard* genutzt werden, um die richtige Ansteuerung zu gewährleisten. Zu erkennen sind die AXI Lite Slave Ports (*s_axi_crtl*), die AXI Stream Ports (*in_r* Slave, *out_r* Master) und die Ein- und Ausgabeports (*in_r*$[13:0]$ und *out_r*$[13:0]$), die mit den *Abitrary Precision Datatypes* erzeugt wurden.

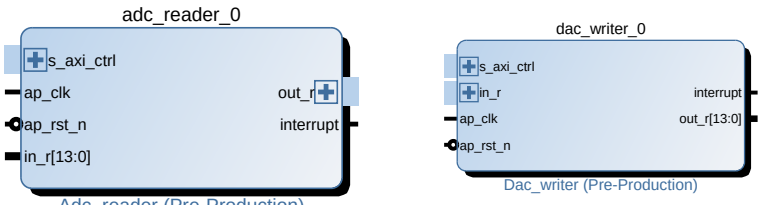

Abbildung 22: IP Cores zur Ansteuerung der AD und DA Wandler

7. Fazit

Diese Bachelorarbeit hat sich mit der Nutzung und Möglichkeiten der High Level Synthese für einen FPGA beschäftigt. Im Fokus stand die Erstellung eines IP Core zur Ansteuerung von externen Komponenten mittels des Xilinx ZYNQ 7000 AP SoC auf Basis des Red Pitaya. In der Einführung wurden folgende Fragen formuliert und um diese Fragen zu beantworten wurden mehrere IP Cores mit wachsendem Funktionsumfang implementiert:

Können mit der High Level Synthese hardwarenahe I/O Aufgaben realisiert werden?

Welche Probleme können bei der Realisierung von Funktionalität entstehen?

Wie gut eignet sich die High Level Synthese für hardwarenahe I/O Aufgaben?

Die erste Frage kann mit ja beantwortet werden: es wurde gezeigt, dass sowohl der lesende als auch schreibende Zugriff auf FPGA Pins möglich ist. Weiterhin wurde gezeigt, dass auch die Ansteuerung auf Basis eines Taktsignales möglich ist.
Die zweite Frage greift eine Stärke der (C basierten) High Level Synthese, die aber speziell im Bereich der hardwarenahen I\O eine Schwäche ist, auf: die starke Abstraktion von Hardware und Funktionalität. So war es nicht ohne weiteres möglich einen IP Core zu implementieren, welcher das Taktsignal direkt nutzen konnte. Die Lösung war eine Kombination von Compilerdirektiven und dem Umschreiben von Code. Die damit verbundene Codevalidierung gestaltete sich in diesem Fall ebenfalls schwierig und konnte nur eingeschränkt vorgenommen werden.
Eine Alternative ist die SystemC basierte High Level Synthese, welche in dieser Arbeit aber nicht genutzt wurde, um die Portierbarkeit des Ursprungscodes zu untersuchen. Bei der Portierbarkeit gestaltete sich speziell das Warten für eine bestimmte Zeitspanne schwierig. Dies wurde durch das Zählen der Taktsignale bei einer festen Frequenz realisiert.
Die dritte Frage kann nicht eindeutig beantwortet werden: so ist es möglich, beispielsweise ohne den Aufwand einen endlichen Zustandsautomaten zur Steuerung der Logik zu implementieren, schnell einen IP Core zu implementieren der aus Softwaresicht genau die Anforderungen erfüllt. Es sind aber größere Anstrengungen nötig, um dem Compiler durch Direktiven mitzuteilen, wie er den gegebenen Code umsetzen soll. Teilweise muss der Code auch umgeschrieben werden, damit der Compiler ihn gemäß den Vorgaben umsetzen kann.

Ein weiterer Kritikpunkt sind die Abschätzungen von Ressourcenverbrauch und Ausführungszeiten (siehe Tabelle 6). Die geschätzte Ausführungszeit kann im Zusammenhang mit taktgesteuerten IP Cores dazu führen, dass wiederholt versucht wird Code zu optimieren, obwohl die tatsächliche Ausführungszeit die Anforderungen erfüllt. Im Fall des geschätzten Ressourcenverbrauchs kann irrtümlich angenommen werden, dass der vorgegebene FPGA über nicht genügend Ressourcen verfügt, um die Funktionalität umzusetzen. Der synthetisierte (hier High Level Synthese) HDL Code sollte daher durch ein passendes Werkzeug (hier Vivado) evaluiert werden, um vermeintliche Engpässe zu bestätigen oder zu widerlegen.

Zusammenfassend kann gesagt werden, dass die High Level Synthese durchaus für hardwarenahe I\O genutzt werden kann. Dennoch muss ein Bewusstsein für Fehler und Einschränkungen vorhanden sein, bevor mit der Auswahl der Quellsprache und Implementierung angefangen wird.

A. Anhang

A.1. Entwicklungsumgebung

Die Entwicklungsumgebung wurde nach der Anleitung im Red Pitaya Github Repository[1] [22] als virtuelle Maschine mit VirtualBox von Oracle aufgesetzt. Tabelle 7 listet die Parameter der VM auf.

OS	Ubuntu 14.04 64Bit
zusätzliche Pakete	make, curl, xz-utils libssl-dev, device-tree-compiler, u-boot-tools schroot qemu, qemu-user, qemu-user-static lib32z1, lib32ncurses5, lib32bz2-1.0, lib32stdc++6, libgtk2.0-0:i386, libfontconfig1:i386, libx11-6:i386, libxext6:i386, libxrender1:i386, libsm6:i386, libqtgui4:i386
CPU	4 Kerne (VT-x / AMD-V, Nested Paging)
RAM	8 GB
HDD	$\geq$ 50 GB

Tabelle 7: Parameter der VM

Für die FPGA Entwicklung wird die Xilinx Vivado Design Suite (Version 2016.2, Vivado HLx 2016.2: Linux Web Installer) genutzt. Um diese herunterzuladen muss ein Xilinx Account erstellt werden. Da die Vivado Design Suite standardmäßig in */opt/Xilinx* installiert wird, werden für die Installation *root* Rechte benötigt. Das Installationsziel kann im Verlauf der Installation angepasst werden. Tabelle 8 listet die Installationseinstellungen auf.

Edition und Lizenz	Vivado HL System Edition, WebPack Lizenz
Design Tools	Vivado Design Suite Software Development Kit (SDK) DocNav
Devices	SoC $\rightarrow$ Zynq-7000
Installation Options	Acquire or Manage a License Key
Installationsordner	/opt/Xilinx

Tabelle 8: Vivado Design Suite Installationseinstellungen

Über den Vivado License Managers wird eine an den Computer gebundene *WebPack* Lizenz erstellt. Abschließend muss noch ein symbolischer Link von *make* nach *gmake* erstellt werden, da *gmake* von der Vivado Design Suite genutzt wird aber nicht unter Ubuntu verfügbar ist. Die 32Bit Bibliotheken werden für das SDK [63] und DocNav [64] benötigt.

[1] Commit 7194146ec3d789007691c171ec049cdff6d4695f

Vivado Design Suite

Die Vivado Design Suite umfasst Vivado HLS, Vivado, das SDK und DocNav. Vivado HLS erstellt beispielsweise aus C Code mittels High Level Synthese einen IP Core, der dann in der Vivado IDE importiert und genutzt werden kann. Vivado deckt unter anderem den HDL Entwicklungsfluss beschrieben in Abschnitt 5.1 ab. Das SDK dient der Entwicklung von Anwendungen auf Basis des durch Vivado erzeugten Bitstreams, dies wird im Rahmen dieser Arbeit nicht genutzt. Lediglich die mitgelieferten Crosscompiler für den ARM Prozessor werden genutzt. DocNav bietet einfachen Zugriff auf die umfangreiche Dokumentation zu unterschiedlichen Xilinx Produkten und empfohlenen Arbeitsweisen. TCL (**T**ool **C**ommand **L**anguage) Skripte erlauben eine einfache Steuerung der Vivado Design Suite. Entsprechende TCL Skripte werden in dieser Arbeit genutzt, um Vivado HLS und Vivado zu steuern.

A.2. Ordnerstruktur eines IP Core

Für jeden IP Core, der im Rahmen dieser Arbeit implementiert wurde, gibt es unter *apps* und *cores* einen Ordner *core_name*. Der Ordner unter *cores* enthält die Quellcodedateien für den IP Core. Die Datei *core_name.c* enthält den eigentlichen Quellcode für den IP Core. Die Headerdatei *core_name.h* enthält die Deklarationen der Top-Level Funktion und wird von *core_name.c* und *tes.c* inkludiert. Die Datei *test.c* enthält entsprechende Funktionen, um die Funktionalität des IP Core zunächst bei der C Simulation und anschließend bei der C / RTL Co Simulation zu verifizieren. Das *directives.tcl* Skript enthält die Compilerdirektiven. Das *run_hls.tcl* Skript dient der Steuerung von Vivado HLS.
Um den IP Core im Betrieb steuern zu können, gibt es eine entsprechende ARM Anwendung im *apps* Ordner. Diese bildet die FPGA Register auf den Adressbus ab, schreibt anschließend die Parameter in die Register und steuert den IP Core.
Im *testbench* Ordner finden sich die *testbench.c* Datei, in der die *test()* Funktion des IP Core aufruft und ausgibt, ob und wie viele Fehler aufgetreten sind und der *test.h* Header, der *test()* und die Fehlervariable *err_cnt* deklariert. Der *shared* Ordner enthält beispielsweise Header, die Datenstrukturen deklarieren, die von der Testbench und den IP Cores genutzt werden. Abbildung 23 zeigt die Ordnerstruktur eines IP Core.

A.3. Erstellung eines IP Cores

Die Erstellung der IP Cores erfolgt mit einem TCL Skript, welches Vivado HLS steuert und das in Listing 5 gezeigt wird. Allgemein werden folgende Schritte abgearbeitet: zuerst wird ein neues Projekt (*open_project*) erzeugt und die Top Level (*set_top*) Funktion spezifiziert. Dem Projekt werden die IP Core Quellcodedateien (*add_files*) und die Testbenchdateien (*add_files -tb*) hinzugefügt. Anschließend wird eine Lösung (*open_solution*) erzeugt, die den FPGA (*set_part*) und die Taktfrequenz (*create_clock*) spezifiziert und ein TCL Skript mit Compilerdirektiven (*source*) einbindet. Durch diese Form der Einbindung ist es möglich, denselben Quellcode für unterschiedliche FPGAs mit unterschiedlichen Direktiven zu synthetisieren.
Der Quellcode wird mithilfe der Testbench ohne Anwendung der Compilerdirektiven zuerst simuliert (*csim_design*), um die funktionale Korrektheit zu testen und dann mittels High Level Synthese unter Anwendung der Compilerdirektiven in VHDL und Verilog Code synthetisiert

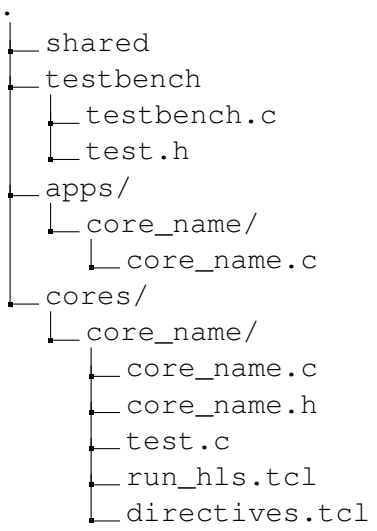

Abbildung 23: Ordnerstruktur eines IP Cores

(*csynth_design*). Der synthetisierte Code wird unter Nutzung der Testbench wieder auf funktionale Korrektheit überprüft (*cosim_design*) und als IP Core exportiert (*export_design*). Vor dem Export als IP Core wird der Verilog Code durch Vivado evaluiert (*-evaluate verilog*), um zu testen, ob dieser nach Synthese zu einer Netlist und Implementierung auf dem FPGA die Zeitvorgaben einhält.

Listing 5: TCL Skript zur Steuerung von Vivado HLS

```
1  open_project -reset led_switch
2
3  set_top led_switch
4
5  add_files "led_switch.h led_switch.c"
6
7  add_files -tb "../../testbench/testbench.c ../../testbench/test.h"
8
9  add_files -tb test.c -cflags "-I../../testbench/"
10
11 open_solution -reset "solution1"
12 set_part {xc7z010clg400-1}
13 create_clock -period 100MHz -name default
14 source "./directives.tcl"
15
16 csim_design
17 csynth_design
18 cosim_design
19 export_design -format ip_catalog -evaluate verilog
20
21 exit
```

A.4. Ansteuerung eines IP Cores

Nach Recherchen im Red Pitaya Forum [16][2][3][4][5] kann ein IP Core über zwei Wege gesteuert werden: Der erste Weg beinhaltet das Implementieren eines Treibers und wurde aufgrund des hohen Aufwands nicht realisiert. Der zweite Weg schreibt entsprechende Werte in die FPGA Register, welche über das AXI Interface auf den Systembus abgebildet wurden. Dieser Weg wurde in der Arbeit verwendet, ist aber nicht besonders sicher, da beim direkten Lesen und Schreiben auf dem Systembus bei falscher Angabe von Adressen das System abstürzen oder im schlimmsten Fall Schaden nehmen kann.
Listing 6 zeigt den Code für die Ansteuerung des LED PWM IP Cores. Die Funktion *map_registers()* bildet die FPGA Register vom Systemadressraum (zugegriffen über */dev/mem*) in einen *memory mapped file* ab. Die Funktion *clean_up()* übernimmt am Ende der Ausführung das Freigeben besagter Datei. In der *main()* Funktion wird dann in die FPGA Register mit *(*(volatile uint32_t *)(&core->...)) = ...;* geschrieben. Das *control* Register beinhaltet die *ap_ctrl* Steuersignale die *gie*, *ier* und *isr* Register sind für Interrupts zuständig. Die Register *led*, *threshold_on* und *threshold_off* entsprechen den Parametern der *led_pwm()* Funktion des LED Blinker IP Cores.
Die Startadresse und die Größe des Adressraumes des IP Cores wurden im Adresseditor in Vivado spezifiziert. Die Adressen der einzelnen Register wurden durch Vivado HLS festgelegt und können im *xled_pwm_hw.h* Header im Ordner *led_pwm/standalone/ impl/ip/drivers/led_pwm_v1_0/src/* abgelesen werden.
Abbildung 24 stellt die AMBA Interconnect Architektur da, farblich markiert sind ARM CPU (rot), High Performance Ports (blau), General Purpose Ports (hellgrün) und der Cache Coherency Port (lila). Die orange Linie markiert den Weg der Parameter von der ARM CPU aus über die Interconnects in den FPGA.

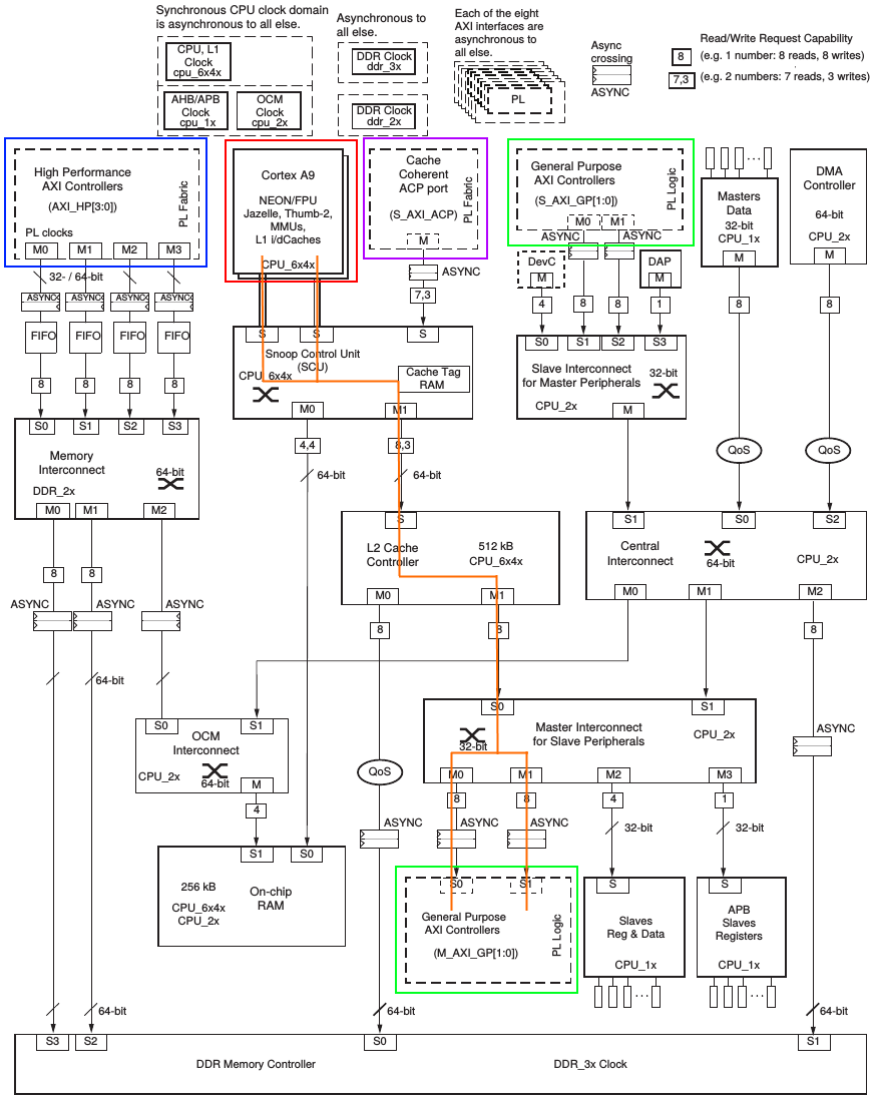

Abbildung 24: ZYNQ AMBA Interconnect Architektur und Interfaces [67, S. 120]

Listing 6: Ansteuerung des LED PWM IP Cores

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdint.h>
4  #include <sys/mman.h>
5  #include <unistd.h>
6  #include <fcntl.h>
7
8  int map_registers(int *fd, void **c, int c_addr, int c_size) {
9          if ((*fd = open("/dev/mem", O_RDWR | O_SYNC)) == -1) {
10                 fprintf(stderr, "Error: could not open /dev/mem!\n");
11                 return -1;
12         }
13         if ((*c = mmap(NULL, c_size, PROT_READ | PROT_WRITE,
14                 MAP_SHARED, *fd, c_addr)) == (void *) -1) {
15                 fprintf(stderr, "Error: could not map memory to file!\n");
16                 return -1;
17         }
18         return 0;
19 }
20 int clean_up(int fd) {
21         if (close(fd) < 0) {
22                 fprintf(stderr, "Error: could not close map file!\n");
23                 return -1;
24         }
25         return 0;
26 }
27 int main(int argc, char** argv) {
28         const int base_addr = 0x83C10000;// end: 0x83C10FFF
29         const int size      = 0x1000;     // 4k
30         volatile struct led_core {
31                 uint32_t control;
32                 uint32_t gie;
33                 uint32_t ier;
34                 uint32_t isr;
35                 uint32_t led;
36                 uint32_t reserved_0x14;
37                 uint32_t threshold_on;
38                 uint32_t reserved_0x1c;
39                 uint32_t threshold_off;
40                 uint32_t reserved_0x24;
41         } *core;
42         int map_file = 0;
43         map_registers(&map_file, (void **) &core, base_addr, size);
44
45         /* blink LED 1 at 1Hz, 0.9s on, 0.1s off */
46         (*(volatile uint32_t *)(&core->led)) = 0x0001;
47         (*(volatile uint32_t *)(&core->threshold_on))  = 90000000;
48         (*(volatile uint32_t *)(&core->threshold_off)) = 10000000;
49
50         /* start core and enable auto restart */
51         (*(volatile uint32_t *)(&core->control)) |= 0x080;
52         (*(volatile uint32_t *)(&core->control)) |= 0x001;
53
54         getchar();
55
56         /* stop core and disable auto restart */
57         (*(volatile uint32_t *)(&core->control)) &= 0xFF7E;
58
59         clean_up(map_file);
60         return 0;
61 }
```

A.5. Analyse der Red Pitaya Quellen

Ein Beispiel aus dem Red Pitaya Github Repository[2] [22] wird genauer analysiert, um die Funktionsweise der API und der FPGA Logik zu verstehen. Der Quellcode des Beispiels findet sich unter *Examples/C/digital_led_blink.c*. In den Zeilen 27 bis 33 findet sich der Code der eine LED *led* blinken lässt: *usleep()* wird genutzt, um die An-/Auszeiten zu realisieren und mit *rp_DpinSetState()* wird der Zustand der gegeben LED auf logisch 0 oder logisch 1 gesetzt. Die Funktion *rp_DpinSetState()* ist in *api/rpbase/src/rp.c* in Zeilen 307 bis 320 definiert, Zeilen 314 bis 317 realisieren das Setzen der Werte für die LEDs. Hierbei werden mit *ioread()* die aktuellen Zustände der LEDs abgefragt und dann eine Bitmaske berechnet und mit *iowrite()* an die Adresse *&hk->led_control* geschrieben. Bei *hk* handelt es sich um die Struktur *housekeeping_control_s*, die in *api/rpbase/src/housekeeping.h* definiert ist und die FPGA Register des Housekeeping Moduls abstrahiert. Diese Struktur wird auf den Adressraum des FPGA mit *cmn_Map()* abgebildet. *cmn_Map()* ist in *api/rpbase/src/common.c* in Zeilen 46 bis 59 als ein Wrapper für *mmap()* definiert. Die Funktionen *ioread()* und *iowrite()* sind als Makros definiert in *api/rpbase/src/common.h* Zeilen 44 und 45 und dienen in diesem Fall dem Schreiben / Lesen im Adressraum des FPGAs.

Auf der FPGA Seite steht das Housekeeping Modul beschrieben in *fpga/rtl/red_pitaya_hk.v*, welches mit seinen Eingabeports entsprechend dieselben Adressen nutzt, auf die die Struktur *housekeeping_control_s* abgebildet wurde. Das Auflösen der Adressen erfolgt durch einen AXI Slave beschrieben in *fpga/rtl/axi_slave.v*, welcher die Übersetzung von den durch AXI genutzten Adressen in einen Red Pitaya eigenen, FPGA internen Systembus übernimmt. Über diesen wird die auf ARM Seite erzeugte Bitmaske in das Housekeeping Modul gegeben, welches durch den Ausgabeport *led_o* in Zeile 38 die Zugriffsmöglichkeit auf die LEDs bietet. Wichtig an dieser Stelle ist das *reg* Sprachelement, welches einen Latch erzeugt und somit den Zustand für die LEDs über mehrere Taktsignale hinweg beibehält, um diesen nicht bei jedem Taktsignal neu setzen zu müssen. Der Ausgabeport des Housekeeping Moduls wird über das sogenannte Topmodul, definiert in *fpga/rtl/red_pitaya_top.v*, als Port nach außen hin sichtbar gemacht. In *fpga/sdc/red_pitaya.xdc*, Zeilen 187 bis 199, findet schließlich das Abbilden von dem *led_o* Port auf die physischen FPGA Pins statt. Abbildung 25 zeigt den Aufrufvorgang schematisch.

[2] Commit 7194146ec3d789007691c171ec049cdff6d4695f

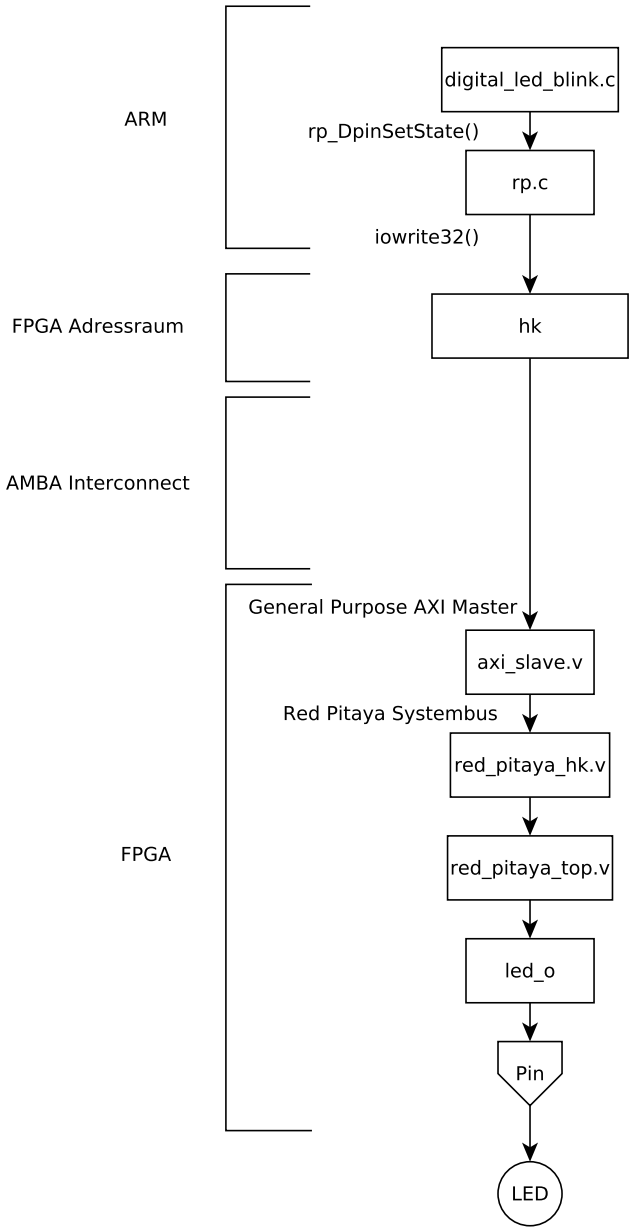

Abbildung 25: Aufrufschema

A.6. Integration eines IP Core in die bestehende Red Pitaya Logik

Die implementieren IP Cores wurden in die bestehende Red Pitaya Logik integriert, um diese zu testen. Der Aufruf von `make project` im *fpga/* Ordner startet Vivado mit dem *red_pitaya_vivado_project.tcl* Skript, das dann ein Vivado Projekt anlegt und die grafische Oberfläche startet. Die Projekteinstellungen bezüglich der Synthese- und Implementierungsschritte müssen noch nach den Vorgaben aus dem *red_pitaya_vivado.tcl* Skript angepasst werden. Tabelle 9 listet die anzupassenden Einstellungen auf.

Synthese	`-flatten_hierachy none`
	`-bufg 16`
	`-keep_equivalent_registers`

Implementierung	`power_opt_design` `phys_opt_design`

Tabelle 9: Synthese- und Implementierungseinstellungen

Um die eigenen IP Cores in Vivado nutzen zu können, müssen diese dem Vivado IP Katalog hinzugefügt werden. In *Project Settings* kann unter *IP Settings* $\rightarrow$ *Repository* der Ordner mit dem gewünschten IP Core ausgewählt und dem Katalog hinzugefügt werden. Vivado registriert auch etwaige Veränderungen am IP Core und bietet dann eine Option zum Upgrade des IP Core an. Anschließend muss das Blockschema über *Open Block Design* geöffnet werden. Hier wird einmal die Funktion *Regenerate Layout* ausgeführt, so dass das Aussehen des Blockschemas Abbildung 26 entspricht. Nachfolgend wird die Integration des LED Controller IP Cores vorgeführt.

Zuerst wird der AXI Protocol Converter entfernt. Über das Kontextmenü wird mit *Add IP* ein LED Controller IP Core hinzugefügt. Vivado bietet an dieser Stelle Hilfe *Run Connection Automation* bei der Verbindung der AXI Ports an. Da der *M_AXI_GP*1 Port der einzig freie AXI Master Port am ZYNQ IP Core ist, es aber durch den XADC und den LED Controller zwei freie AXI Slave Ports gibt, wird automatisch ein AXI Interconnect eingefügt und die AXI Interfaces und übrigen Signale wie Takt und Reset automatisch verbunden. Im *Address Editor* muss nur noch die *Address Range* vom LED Controller auf $4K$ angepasst werden, damit nicht unnötig viele Adressen belegt werden.

Des Weiteren müssen einige IP Cores über die jeweilige Konfigurationsübersicht umkonfiguriert werden, die über einen Doppelklick auf den betreffenden IP Core zu erreichen ist. Der ZYNQ IP Core muss so konfiguriert werden, so dass das Taktsignal (FCLK3) mit dem AXI Interconnect, XADC, Processor Reset System und LED Controller verbunden sind, eine Frequenz von $100MHz$ hat. Unter *Clock Configuration*$\rightarrow$*PL Fabric Clocks* muss das Taktsignal FCLK3 auf $100MHz$ gesetzt werden. Der externe Port des Taktsignals muss analog ebenfalls auf $100MHz$ gesetzt werden. In Abbildung 26 sind alle anzupassenden Blöcke markiert.

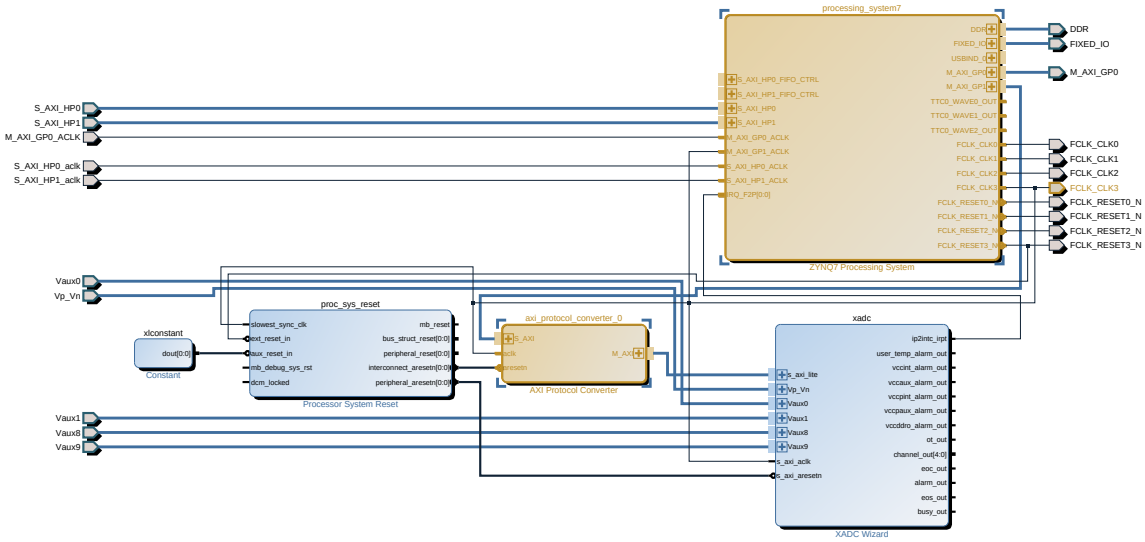

Abbildung 26: Red Pitaya Blockschema

Zusätzlich müssen acht *xlslice* und acht *SelectIO Wizard* IP Cores instanziiert werden. Anschließend wird jeweils der *Dout* Port eines *xlslice* IP Core mit dem *data_in_from_pins* Port eines *SelectIO Wizard* IP Core verbunden. Der *clk_in* Port der *SelectIO Wizard* IP Cores wird anschließend mit FCLK3 verbunden. Die *Din* Ports der *xlslice* IP Cores werden mit dem *exp_n_io* Port verbunden. Bei jedem *xlslice* IP Core muss *Din Width* über die Konfigurationsübersicht auf acht gesetzt werden. Die Werte für *Din From* und *Din Down To* sind paarweise für jeden *xlslice* IP Core gleich: sie fangen bei null für den ersten *xlslice* an und enden bei sieben für den letzten. Die *SelectIO Wizard* IP Cores müssen auch über die Konfigurationsübersicht angepasst werden: im Reiter *Data Bus Setup* muss für *I/O Signaling* die Option *LVCMOS*33 und im Reiter *Clock Setup* unter *Clocking Strategy* die Option *Internal Clock* ausgewählt werden. Dann kann jeder *data_in_to_device* Port mit einem der *gp_i* Ports des LED Controller verbunden werden.

Hierbei muss beachtet werden, dass der im *xlslice* eingestellte Wert für *Din From* und *Din Down To* mit dem Index des gewählten *gp_i* Ports übereinstimmt, ansonsten werden die Pins nicht mit den dazugehörigen LEDs verbunden und die korrekte Funktion des LED Controllers ist nicht gewährleistet. Ein *xlconcat* IP Core wird hinzugefügt, die *Number of Ports* über die Konfigurationsübersicht auf acht gesetzt und mit den *led_o* Ports des LED Controllers verbunden werden. Hierbei ist wieder auf die richtige Reihenfolge zu achten. Abschließend müssen über das Kontextmenü mit *Create Port* ein *input „exp_io_n“* und ein *output „led_o“* Port vom Typ *other* als *vector from* 7 *to* 0 angelegt werden. Der *led_o* Port wird mit *dout* Port des *xlconcat* IP Core verbunden, die *Din* Ports der *xlslice* IP Cores mit dem *exp_io_n* Port. Abbildung 27 markiert im angepassten Blockschema alle neu hinzugekommenen IP Cores und Ports.

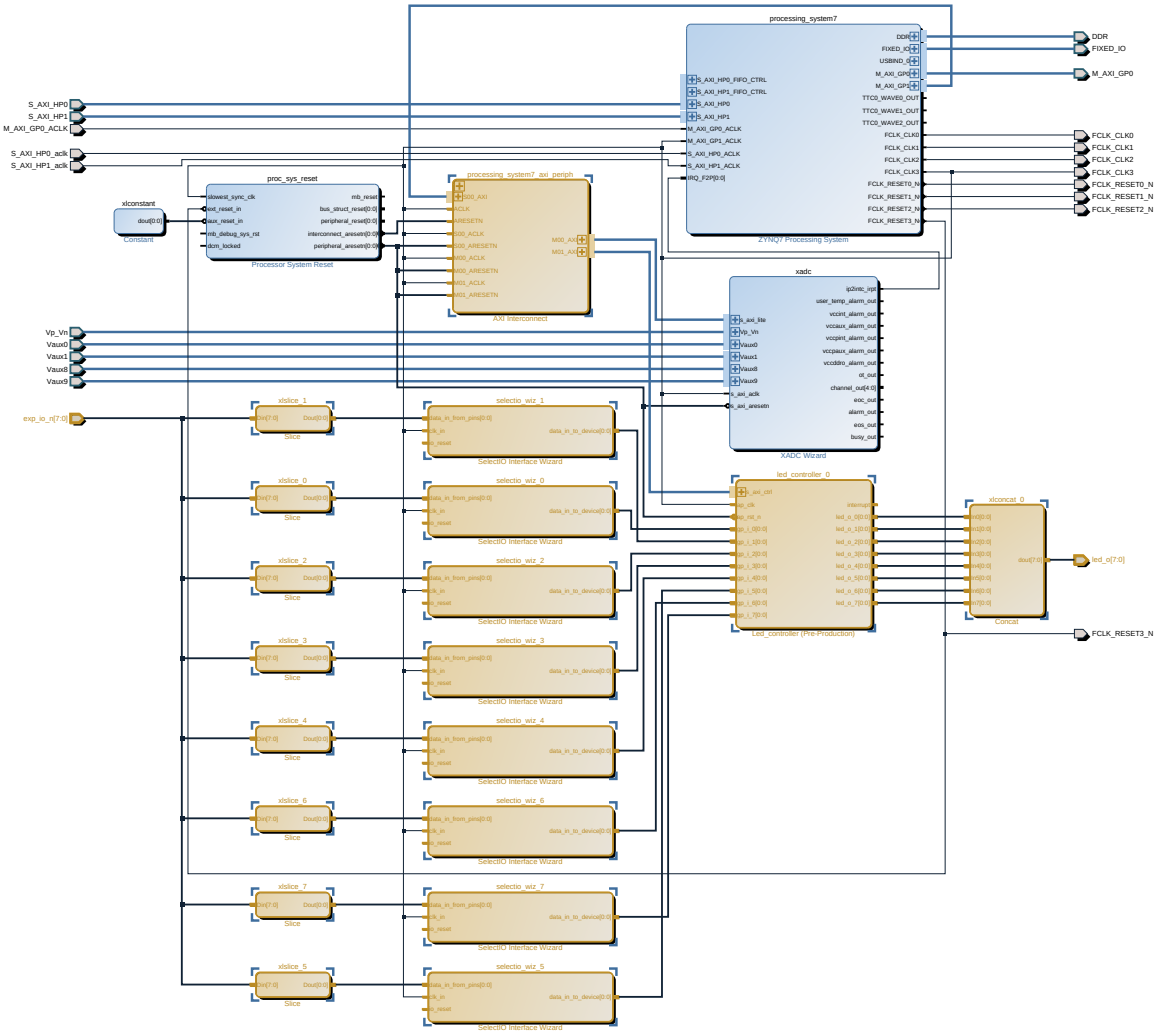

Abbildung 27: Red Pitaya Blockschema mit vorgenommenen Änderungen

Zuletzt muss noch ein HDL Wrapper über das Kontextmenü *Create HDL Wrapper* im Fenster *Sources* im Reiter *IP Sources* erzeugt werden. Dabei sollte sichergestellt werden, dass der *led_o* Port als *output* und der *exp_io_n* als *input* Port aufgeführt wird. Der HDL Wrapper dient als Schnittstelle zwischen den im Blockschema instanziierten IP Cores und der Red Pitaya Logik in HDL. Schließlich müssen noch die in Tabelle 10 angegebenen Änderungen an den Red Pitaya Quellen vorgenommen werden.

red_pitaya_hk.v	jedes Vorkommen von *led_o* und *exp_n_** mit // auskommentieren
red_pitaya_ps.v	Zeile 73: *output [7: 0] led_o,* über *// system read/write channel* Zeile 74: *input [7: 0] exp_n_io* unter *output [7: 0] led_o,* Zeile 392: *.led_o(led_o),* über *//GP0* Zeile 393: *.exp_n_io (exp_n_io),* unter *.led_o(led_o),* hinzufügen
red_pitaya_top.v	Zeile 111*: inout* in *input* ändern Zeile 180*: .led_o(led_o),* hinzufügen Zeile 180*: .exp_n_io(exp_n_io),* hinzufügen Zeilen 392, 399 bis 401 und 414 mit // auskommentieren

Tabelle 10: Änderungen an den Red Pitaya Quellen

Abschließend kann der Bitstream über *Generate Bitstream* erzeugt, auf den Red Pitaya kopiert und der FPGA damit konfiguriert werden. Mit der Steueranwendung auf ARM Seite kann der LED Controller IP Core dann gesteuert werden.

A.7. Quellcode

LED Switch

Listing 7: Compilerdirektiven

```
set_directive_interface -mode s_axilite -bundle ctrl "led_switch" state
set_directive_interface -mode s_axilite -bundle ctrl "led_switch" led
set_directive_interface -mode s_axilite -bundle ctrl "led_switch"
```

Listing 8: Verilog Code des LED Switch SW IP Cores

```
1  module led_switch (
2          ap_clk,
3          ap_rst_n,
4          led_o,
5          led_o_ap_vld,
6          s_axi_ctrl_AWVALID,
7          s_axi_ctrl_AWREADY,
8          s_axi_ctrl_AWADDR,
9          s_axi_ctrl_WVALID,
10         s_axi_ctrl_WREADY,
11         s_axi_ctrl_WDATA,
12         s_axi_ctrl_WSTRB,
13         s_axi_ctrl_ARVALID,
14         s_axi_ctrl_ARREADY,
15         s_axi_ctrl_ARADDR,
16         s_axi_ctrl_RVALID,
17         s_axi_ctrl_RREADY,
18         s_axi_ctrl_RDATA,
19         s_axi_ctrl_RRESP,
20         s_axi_ctrl_BVALID,
21         s_axi_ctrl_BREADY,
22         s_axi_ctrl_BRESP,
23         interrupt
24 );
25 ...
26 output  [7:0] led_o;
27 ...
28 wire   [7:0] led;
29 wire   [0:0] state;
30 reg   [7:0] led_states;
31 ...
32 reg   [31:0] tmp_6_fu_81_p4;
33 ...
34 initial begin
35 #0 ap_CS_fsm = 1'b1;
36 #0 led_states = 8'b00000000;
37 end
```

```
38
39 led_switch_ctrl_s_axi #(
40     .C_S_AXI_ADDR_WIDTH(C_S_AXI_CTRL_ADDR_WIDTH ),
41     .C_S_AXI_DATA_WIDTH( C_S_AXI_CTRL_DATA_WIDTH ))
42 led_switch_ctrl_s_axi_U(
43     .AWVALID(s_axi_ctrl_AWVALID),
44     .AWREADY(s_axi_ctrl_AWREADY),
45     .AWADDR(s_axi_ctrl_AWADDR),
46     .WVALID(s_axi_ctrl_WVALID),
47     .WREADY(s_axi_ctrl_WREADY),
48     .WDATA(s_axi_ctrl_WDATA),
49     .WSTRB(s_axi_ctrl_WSTRB),
50     .ARVALID(s_axi_ctrl_ARVALID),
51     .ARREADY(s_axi_ctrl_ARREADY),
52     .ARADDR(s_axi_ctrl_ARADDR),
53     .RVALID(s_axi_ctrl_RVALID),
54     .RREADY(s_axi_ctrl_RREADY),
55     .RDATA(s_axi_ctrl_RDATA),
56     .RRESP(s_axi_ctrl_RRESP),
57     .BVALID(s_axi_ctrl_BVALID),
58     .BREADY(s_axi_ctrl_BREADY),
59     .BRESP(s_axi_ctrl_BRESP),
60     .ACLK(ap_clk),
61     .ARESET(ap_rst_n_inv),
62     .ACLK_EN(1'b1),
63     .ap_start(ap_start),
64     .interrupt(interrupt),
65     .ap_ready(ap_ready),
66     .ap_done(ap_done),
67     .ap_idle(ap_idle),
68     .led(led),
69     .state(state)
70 );
71 ...
72 always @ (posedge ap_clk) begin
73     if (((1'b1 == ap_sig_cseq_ST_st1_fsm_0) & ~(ap_start == 1'b0))) begin
74         led_states <= tmp_fu_91_p1;
75     end
76 end
77 ...
78 assign led_o = tmp_6_fu_81_p4[7:0];
79 assign tmp_3_fu_59_p2 = ap_const_lv8_1 << led;
80 assign tmp_4_fu_65_p2 = (tmp_3_fu_59_p2 ^ ap_const_lv8_FF);
81 assign tmp_5_cast_fu_77_p1 = tmp_5_fu_71_p2;
82 assign tmp_5_fu_71_p2 = (led_states & tmp_4_fu_65_p2);
83
84 always @ (*) begin
85     tmp_6_fu_81_p4 = tmp_5_cast_fu_77_p1;
86     tmp_6_fu_81_p4[led] = |(state);
87 end
88
89 assign tmp_fu_91_p1 = tmp_6_fu_81_p4[7:0];
90
91 endmodule //led_switch
```

LED PWM

Listing 9: naiver Code des LED PWM IP Cores

```
#include <stdbool.h>

unsigned char led_o = 0;
volatile bool clk;

void blink_led(unsigned char led, unsigned int threshold)
{
        static unsigned int ticks = 0;
        static bool switched_on = 0;

        if (clk) {
                if (threshold == ticks) {
                        if (switched_on) {
                                led_o = 0;
                                switched_on = 0;
                        } else {
                                led_o = 1 << led;
                                switched_on = 1;
                        }
                        ticks = 0;
                }
                ticks++;
        }
}
```

Listing 10: Verilog Code des LED PWM IP Cores

```
...
always @ (posedge ap_clk) begin
    if (ap_sig_98) begin
        if (~(1'b0 == demorgan_fu_117_p2)) begin
            toggle <= 1'b0;
        end else if (ap_sig_107) begin
            toggle <= 1'b1;
        end
    end
end
...
always @ (posedge ap_clk) begin
    if (((1'b1 == ap_sig_cseq_ST_pp0_stg0_fsm_0) & (1'b1 == ap_reg_ppiten_pp0_it1)
    & ~((1'b1 == ap_reg_ppiten_pp0_it0) & (ap_start == 1'b0))
    & ~(1'b0 == tmp_4_fu_146_p2))) begin
        led_states <= tmp_1_fu_187_p2;
    end
end
...
always @ (posedge ap_clk) begin
    if (((1'b1 == ap_sig_cseq_ST_pp0_stg0_fsm_0) & (1'b1 == ap_reg_ppiten_pp0_it1)
    & ~((1'b1 == ap_reg_ppiten_pp0_it0) & (ap_start == 1'b0)))) begin
        ticks <= tmp_2_fu_200_p2;
    end
end
...
```

Literatur

[1] Accessing the ap_clock signal for a LED blinker. https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/Accessing-the-ap-clock-signal-for-a-LED-blinker/td-p/722497. Last visited: 08.02.2017.

[2] Accessing the LEDs without using the Red Pitaya API page 1. http://forum.redpitaya.com/viewtopic.php?f=14&t=1573. Last visited: 08.02.2017.

[3] Accessing the LEDs without using the Red Pitaya API page 2. http://forum.redpitaya.com/viewtopic.php?f=14&t=1573&start=10. Last visited: 08.02.2017.

[4] Accessing the LEDs without using the Red Pitaya API page 3. http://forum.redpitaya.com/viewtopic.php?f=14&t=1573&start=20. Last visited: 08.02.2017.

[5] Accessing the LEDs without using the Red Pitaya API page 4. http://forum.redpitaya.com/viewtopic.php?f=14&t=1573&start=30. Last visited: 08.02.2017.

[6] AD-Wandler. http://www.mikrocontroller.net/articles/AD-Wandler. Last visited: 19.01.2017.

[7] AMBA 3. https://developer.arm.com/products/architecture/amba-protocol/amba-3. Last visited: 19.01.2017.

[8] Applications. http://redpitaya.readthedocs.io/en/latest/doc/ appsFeatures/apps-featured/apps-featured.html#applications. Last visited: 19.01.2017.

[9] Blink. http://redpitaya.readthedocs.io/en/latest/doc/appsFeatures/examples/dig-exm1.html. Last visited: 19.01.2017.

[10] DA-Wandler. http://www.mikrocontroller.net/articles/DA-Wandler. Last visited: 19.01.2017.

[11] Electrical schematics for Red Pitaya V1.0.1 Release1. https://dl.dropboxusercontent.com/s/jkdy0p05a2vfcba/Red_Pitaya_Schematics_v1.0.1.pdf. Last visited: 19.01.2017.

[12] External ADC clock. https://redpitaya.readthedocs.io/en/latest/doc/developerGuide/125-14/extADC.html. Last visited: 19.01.2017.

[13] Fast analog IO. https://redpitaya.readthedocs.io/en/latest/doc/developerGuide/125-14/fastIO.html. Last visited: 19.01.2017.

[14] Flipflop. http://www.mikrocontroller.net/articles/Flipflop. Last visited: 19.01.2017.

[15] FPGA. http://www.mikrocontroller.net/articles/FPGA. Last visited: 19.01.2017.

[16] FPGA AXI GP0 bus <-> PS memory map? [HANDOVER]. http://forum.redpitaya.com/viewtopic.php?f=14&t=1124. Last visited: 08.02.2017.

[17] FPGA Design Creation and Simulation. https://www.aldec.com/en/products/fpga_simulation/active-hdl. Last visited: 19.01.2017.

[18] FPGA synthesis and place-and-route. http://fpga4fun.com/FPGAsoftware5.html. Last visited: 19.01.2017.

[19] Latch. http://www.mikrocontroller.net/articles/Latch. Last visited: 19.01.2017.

[20] Push button and turn on led diode. http://redpitaya.readthedocs.io/en/latest/doc/appsFeatures/examples/dig-exm3.html. Last visited: 19.01.2017.

[21] Red Pitaya diagramm. http://redpitaya.readthedocs.io/en/latest/_images/boards_1.jpg. Last visited: 19.01.2017.

[22] Red Pitaya Github Repository. https://github.com/RedPitaya/RedPitaya/tree/2a77e0ee09574171deb96c7cea971f9cf671cf3e. Last visited: 19.01.2017.

[23] Red pitaya schema. http://blog.redpitaya.com/features-compare1-2/. Last visited: 19.01.2017.

[24] SDR - Software Defined Radio (by Pavel Demin). http://redpitaya.readthedocs.io/en/latest/doc/appsFeatures/marketplace/ marketplace.html #sdr-software-defined-radio-by-pavel-demin. Last visited: 19.01.2017.

[25] STEMLab 125-10 vs. STEMLab 125-14 (originally Red Pitaya v1.1). http://redpitaya.readthedocs.io/en/latest/doc/developerGuide/125-10/top.html#stemlab-125-10-vs-stemlab-125-14-originally-red-pitaya-v1-1. Last visited: 19.01.2017.

[26] System overview. http://redpitaya.readthedocs.io/en/latest/doc/developerGuide/ software/-sysOver.html. Last visited: 19.01.2017.

[27] Altera. 40-nm FPGAs: Architecture and Performance Comparison. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01088-40nm-architecture-performance-comparison.pdf, July 2006. Last visited: 19.01.2017.

[28] Altera. FPGA Architecture. https://www.altera.com/en_US/pdfs/literature/wp/wp-01003.pdf, July 2006. Last visited: 19.01.2017.

[29] Altera. Implementing FPGA Design with the OpenCL Standard. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01173-opencl.pdf, November 2013. Last visited: 19.01.2017.

[30] Michael Barr. Introduction to Pulse Width Modulation. http://www.embedded.com/electronics-blogs/beginner-s-corner/4023833/Introduction-to-Pulse-Width-Modulation, August 2001. Last visited: 19.01.2017.

[31] Peter Cheung. Modern FPGA Architectures. http://www.ee.ic.ac.uk/pcheung/teaching/ee3_DSD/Topic%203%20-%20Modern%20FPGAs.pdf, January 2008. Last visited: 19.01.2017.

[32] Philippe Coussy, Michael Meredith, Daniel D. Gajski, and Andres Takach. An Introduction to High-Level Synthesis. http://janders.eecg.toronto.edu/1387/readings/hls.pdf. Last visited: 19.01.2017.

[33] Rory Dear. 'C' lands on FPGAs to make embedded multicore computing a reality. *Embedded Computing Design*, April 2016. Last visited: 19.01.2017.

[34] Paul Goossens. FPGA Programmieren. *elektor*, March 2008. Last visited: 19.01.2017.

[35] Timothy Hirzel. PWM. https://www.arduino.cc/en/Tutorial/PWM. Last visited: 19.01.2017.

[36] National Instruments. Wie funktionieren FPGAs? http://www.ni.com/white-paper/6983/de/, May 2013. Last visited: 19.01.2017.

[37] Dr. Nathan Jachimiec and Dr. Fernando Martinez Vallina. Vivado HL/AutoESL: Agilent packet engine case study. http://www.techdesignforums.com/practice/technique/vivado-hls-agilent/. Last visited: 19.01.2017.

[38] Ed Klingman. FPGA programming step by step. http://www.design-reuse.com/articles/7330/fpga-programming-step-by-step.html, March 2004. Last visited: 19.01.2017.

[39] Robert Lacoste. Die Magie der PLL. *elektor*, July/August 2016.

[40] Max Maxfield. ASIC, ASSP, SoC, FPGA - What's the Difference? *EE Times*, 2014. Last visited: 19.01.2017.

[41] Russell Merrick. Crossing Clock Domains in an FPGA. https://www.nandland.com/articles/crossing-clock-domains-in-an-fpga.html. Last visited: 19.01.2017.

[42] Russell Merrick. Digital Design for Beginners. https://www.nandland.com/articles/what-is-a-digital-designer.html. Last visited: 19.01.2017.

[43] Russell Merrick. Synthesizable vs. Non-Synthesizable code. https://www.nandland.com/articles/synthesizable-vs-non-synthesizable-code-fpga-asic.html. Last visited: 19.01.2017.

[44] Russell Merrick. Tutorial - How Flip-Flops Work in FPGAs. https://www.nandland.com/articles/flip-flop-register-component-in-fpga.html. Last visited: 19.01.2017.

[45] Russell Merrick. Tutorial - Performing Boolean Algebra inside an FPGA using Look-Up Tables (LUTs). https://www.nandland.com/articles/boolean-algebra-using-look-up-tables-lut.html. Last visited: 19.01.2017.

[46] Russell Merrick. Tutorial - What is a Testbench. https://www.nandland.com/articles/what-is-a-testbench-fpga.html. Last visited: 19.01.2017.

[47] Russell Merrick. Tutorial - What is an FPGA Latch? https://www.nandland.com/articles/what-is-a-latch-fpga.html. Last visited: 19.01.2017.

[48] Russell Merrick. What is an FPGA? what is an ASIC? https://www.nandland.com/articles/what-is-an-fpga-what-is-an-asic.html. Last visited: 19.01.2017.

[49] Andrew Moore. *FPGAs For Dummies Altera Special Edition*. John Wiley & Sons, Inc., 111 River St. Hoboken NJ 07030-5774, 2014. Last visited: 19.01.2017.

[50] Kevin Morris. HLS versus OpenCL Xilinx and Altera Square Off on the Future. *Electronic Engineering Journal*, March 2013. Last visited: 19.01.2017.

[51] Medien-und Elektrotechnik Prof. Dr.-Ing. Jens Onno Krah, Ingenieur Wissenschaftliches Zentrum der FH Köln Fakultät für Informations. Skript zur Vorlesung Digitale Signalverarbeitung mit Fpga. https://www.f07.th-koeln.de/imperia/md/content/personen/krah_jens/dsf.pdf, February 2014. Last visited: 19.01.2017.

[52] Eastern Washington University. Introduction to CMOS VLSI Design. http://web.ewu.edu/groups/technology/Claudio/ee430/Lectures/L1-print.pdf. Last visited: 19.01.2017.

[53] William Wong. Understanding FPGA Processor Interconnects. http://electronicdesign.com/fpgas/understanding-fpga-processor-interconnects, July 2012. Last visited: 19.01.2017.

[54] Xilinx. http://www.xilinx.com/content/dam/xilinx/imgs/block-diagrams/zynq-mp-core-dual.png. Last visited: 19.01.2017.

[55] Xilinx. AMBA AXI4 Interface Protocol. https://www.xilinx.com/products/intellectual-property/axi.html. Last visited: 19.01.2017.

[56] Xilinx. AXI Interconnect v2.1 LogiCORE IP Product Guide. https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf. Last visited: 19.01.2017.

[57] Xilinx. Zynq-7000 All Programmable SoC Overview. http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf. LastLast visited: 19.01.2017.

[58] Xilinx. Zynq-7000 All Programmable SoCs Product Tables and Product Selection Guide. http://www.xilinx.com/support/documentation/selection-guides/zynq-7000-product-selection-guide.pdf. Last visited: 19.01.2017.

[59] Xilinx. Zynq Ultrascale+ MPSoC Product Brief. https://www.xilinx.com/support/documentation/product-briefs/zynq-ultrascale-plus-product-brief.pdf. Last visited: 19.01.2017.

[60] Xilinx. Fpga. https://courses.cs.washington.edu/courses/cse467/03wi/FPGA.pdf, 2003. Last visited: 19.01.2017.

[61] Xilinx. Introduction to FPGA Design with Vivado High-Level Synthesis. http://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf, July 2013. Last visited: 19.01.2017.

[62] Xilinx. A GENERATION AHEAD FOR SMARTER SYSTEMS: 9 REASONS WHY THE XILINX ZYNQ-7000 ALL PROGRAMMABLE SOC PLATFORM IS THE SMARTEST SOLUTION. http://www.xilinx.com/publications/prod_mktg/zynq-7000-generation-ahead-backgrounder.pdf, 2014. Last visited: 19.01.2017.

[63] Xilinx. Ar# 63561 2014.4 SDK - arm-xilinx-eabi-gcc: No such file or directory. https://www.xilinx.com/support/answers/63561.html, February 2015. Last visited: 19.01.2017.

[64] Xilinx. Ar# 66184 Install - How do I find out which libraries are required to run Vivado tools in Linux? https://www.xilinx.com/support/answers/66184.html, December 2015. Last visited: 19.01.2017.

[65] Xilinx. AXI DMA Controller. https://www.xilinx.com/products/intellectual-property/axi_dma.html, February 2015. Last visited: 19.01.2017.

[66] Xilinx. Vivado Design Suite AXI Reference Guide. https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf, June 2015. Last visited: 19.01.2017.

[67] Xilinx. Zynq-7000 All Programmable SoC Technical Reference Manual. http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf, February 2015. Last visited: 19.01.2017.

[68] Xilinx. Vivado Design Suite User Guide High-Level Synthesis. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug902-vivado-high-level-synthesis.pdf, June 2016. Last visited: 19.01.2017.

[69] Xilinx. Vivado Design Suite User Guide High-Level Using Constraints. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug903-vivado-using-constraints.pdf, June 2016. Last visited: 19.01.2017.

[70] Bob Zeidman. All about FPGAs page 1. *EE Times*, 2006. Last visited: 19.01.2017.

[71] Bob Zeidman. All about FPGAs page 2. *EE Times*, 2006. Last visited: 19.01.2017.

[72] Bob Zeidman. All about FPGAs page 3. *EE Times*, 2006. Last visited: 19.01.2017.

[73] Xuan 'Silvia' Zhang. Tutorial for Vivado HLS. http://www.ese.wustl.edu/ xuan.zhang/ese566_files/tutorials/vivado_tutorial.pdf. Last visited: 19.01.2017.