Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Armin Jeyrani Mamegani

Erprobung und Evaluierung von Methoden zur partiellen und dynamischen Rekonfiguration eines SoC-FPGAs

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Armin Jeyrani Mamegani

Erprobung und Evaluierung von Methoden zur partiellen und dynamischen Rekonfiguration eines SoC-FPGAs

Masterarbeit
im Studiengang Master of Science Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Erstgutachter: Prof. Dr.-Ing. Bernd Schwarz
Zweitgutachter: Prof. Dr. Bettina Buth

Abgegeben am 26. Februar 2010

Armin Jeyrani Mamegani

Thema der Masterarbeit

Erprobung und Evaluierung von Methoden zur partiellen und dynamischen Rekonfiguration eines SoC-FPGAs

Stichworte

FPGA, SoC, µBlaze, partiell rekonfigurierbare Region (PRR), partiell rekonfigurierbare Module (PRM), Petalinux, µCLinux, Kernel, Internal Configuration Access Port (ICAP), ML-505

Kurzzusammenfassung

Diese Arbeit umfasst die Realisierung eines FPGA-basierten SoC, mit dem sich mehrere SW-Anwendungen durch zur Laufzeit geladene HW-Module beschleunigen lassen. Es werden verschiedene Rekonfigurationsarchitekturen vorgestellt und miteinander verglichen. Aufbauend auf der selbstrekonfigurierenden SoC-Architektur wird ein µBlaze-System mit einer partiell rekonfigurierbaren Region (PRR) anhand der Xilinx Werkzeugkette zur partiellen Rekonfiguration entwickelt. Die partielle Rekonfiguration wird durch eine eingebettete Testapplikation verifiziert, die zwei partiell rekonfigurierbare Module (PRM) gegenseitig zur Laufzeit des Systems austauscht. Mit der Petalinux Distribution wird ein µCLinux-Kernel für das µBlaze-System konfiguriert. Dieses SW-System bildet die Plattform für die Ausführung mehrerer SW-Anwendungen. Dem Linux-Kernel wird zusätzlich ein Treiber zur Anbindung der internen Rekonfigurationsschnittstelle des FPGAs (Internal Configuration Access Port) angepasst und hinzugefügt, womit die Rekonfigurationsmechanismen für SW-Anwendungen bereitgestellt werden. Das µCLinux-PR-System wird auf der Xilinx ML-505 Evaluierungsplattform integriert und getestet. Zusätzlich wird das Verfahren zur Erstellung eines selbstrekonfigurierenden µBlaze-Systems für folgende Projekte an der HAW Hamburg aufbereitet. Diese Arbeit öffnet den Weg für partiell rekonfigurierbare Anwendungen im Bereich der Eingebetteten Systeme. Das entwickelte µCLinux-PR-System bietet eine Plattform zur Beschleunigung von SW und Einsparung von HW-Ressourcen.

Armin Jeyrani Mamegani

Title of the paper

Test and evaluation of approaches for partial and dynamic reconfiguration of a SoC-FPGA

Keywords

FPGA, SoC, µBlaze, partially reconfigurable region (PRR), partially reconfigurable module (PRM), Petalinux, µCLinux, kernel, Internal Configuration Access Port (ICAP), ML-505

Abstract

This work includes the implementation of a FPGA-based SoC that enables the acceleration of multiple sw applications by hw modules loaded at runtime. Various reconfiguration architectures are presented and compared. Based on the self reconfiguring SoC architecture a µBlaze system including a partially reconfigurable region (PRR) is developed using the Xilinx tool chain for partial reconfiguration. The partial reconfiguration is verified by an embedded test application that replaces one partially reconfigurable module (PRM) by the other at system runtime. With the Petalinux distribution a µCLinux kernel for the µBlaze system is configured. This sw system is the platform for the execution of multiple software applications. The Linux kernel is also adapted and added a driver for connecting the internal reconfiguration interface to the FPGA (Internal Configuration Access Port), that provides the mechanism for reconfiguration to sw applications, as well. The µCLinux-PR system is integrated and tested on the Xilinx ML-505 evaluation platform. In addition, the procedure of implementing a self-reconfiguring µBlaze system is developed for following projects at the HAW Hamburg. This work opens the way for partially reconfigurable applications in the field of Embedded Systems. The developed µCLinux-PR system provides a platform for accelerating sw and reducing hw resources.

Inhaltsverzeichnis

1. Einleitung

Eingebettete Systeme in den Bereichen Fahrzeugelektronik, autonomen Fahrzeugen, mobilen Unterhaltungsgeräten, Medizintechnik etc. gewinnen immer mehr an Funktionalität. Der stetige Wachstum von eingebetteten Anwendungen und die Anforderungen von eingebetteten Systemen nach Echtzeitfähigkeit und Parallelität führt FPGA-basierte Anwendungen in nahezu allen Bereichen dazu, immer mehr an Funktionen zu integrieren [2], [9]. Der derzeitige Grad der Transistorendichte von FPGAs schafft die Voraussetzung ein ganzes Systems innerhalb einer einzigen FPGA-Einheit einzubetten [17], [9].

Im Bereich des HW/SW Co-design bestehen Systeme sowohl aus HW, als auch aus SW, die im Verbund die gesamte Funktionalität des Systems unter Einhaltung der Timinganforderunen realisieren. Teile von SW-basierten Anwendungen werden somit durch dedizierte HW-Module unterstützt. Ein aus HW und SW bestehendes eingebettetes System lässt sich als **S**ystem **o**n a programmable **C**hip (SoC) in einer einzigen FPGA-Einheit integrieren. Diese SoCs bestehen aus folgenden Komponenten:

µController Verbund aus einem Prozessor und Peripheriefunktionalität. Bildet die Basis zur Ausführung von SW.

Interner Speicher Der FPGA-interne Speicher dient als Teil des µControllers zur Ablage von SW und dem gesamten System als Datenspeicher.

SW-Anwendungen Zur Realisierung der Funktionalität des Systems bieten SW-Anwendungen die Flexibilität neue Anforderungen zu erfüllen.

Beschleuniger (HW-IPs) Durch dedizierte, digitale Systeme lässt sich die Ausführungszeit der SW verkürzen und somit die Ausführungsgeschwindigkeit des Systems steigern.

Betriebssystem Bietet die Ressourcen für die quasi-parallele Ausführung von mehreren SW-Anwendungen und der Verwaltung des gesamten Systems.

Ein komplexes System ist üblicherweise aus mehreren in ihrer Funktion eigenständigen Modulen zusammengesetzt. In einigen Fällen arbeiten nicht alle Module parallel bzw. finden nicht zur selben Zeit Verwendung. Ein im FPGA integriertes Modul, dessen Funktionen gerade nicht genutzt werden, verursacht einen höheren Energie- sowie FPGA-Ressourcenbedarf.

Eine partielle Rekonfiguration eines FPGA-basierten SoCs zur Laufzeit bedeutet, dass ausgewählte IP-Module im laufenden Betrieb ausgetauscht werden, während die übrigen HW- und SW-Module weiterhin in Anwendungen ihre Funktionen ausführen. Diese Methode zur Rekonfiguration von FPGAs (PR, Partial Reconfiguration) gestattet mehreren Entwurfsmodulen die Verwendung der selben FPGA-Ressourcen im Time-Sharing-Modus ohne die Unterbrechung des Betriebs der Basisfunktionen. Diese FPGA-Technologie ist insbesondere für die sogenannten Mission-Critical-Anwendungen von Vorteil, die während der Neudefinition von Subsystemen

nicht unterbrochen werden dürfen. Die wesentlichen Vorteile der partiellen Rekonfiguration von FPGAs sind der reduzierte Energiebedarf, die Wiederverwendung von HW-Ressourcen, die Erweiterung der Funktionalität eines einzelnen FPGAs sowie die Verwendung von weniger bzw. kleineren FPGAs [5].

Untersuchungsziel dieser Arbeit sind SoC-Plattformen deren HW-Funktionseinheiten zur Laufzeit anwendungsspezifisch aktiviert bzw. deaktiviert werden. Es wurde ein FPGA-basiertes SoC aufgebaut, mit dem sich mehrere SW-Anwendungen durch zur Laufzeit geladene HW-Module beschleunigen lassen. Das System besteht aus einem FPGA-basierten µController-System und der Petalinux Distribution (vgl. Abbildung 1.1). Die Petalinux Distribution beinhaltet eine Zusammenstellung eines Linux-Kernels und Treiber für µBlaze-Systeme, mit der die Ausführung mehrerer SW-Anwendungen verwaltet wird. Dem Linux-Kernel wurde zusätzlich ein Treiber zur Anbindung der internen Rekonfigurationsschnittstelle des FPGAs (Internal Configuration Access Port) angepasst und hinzugefügt.

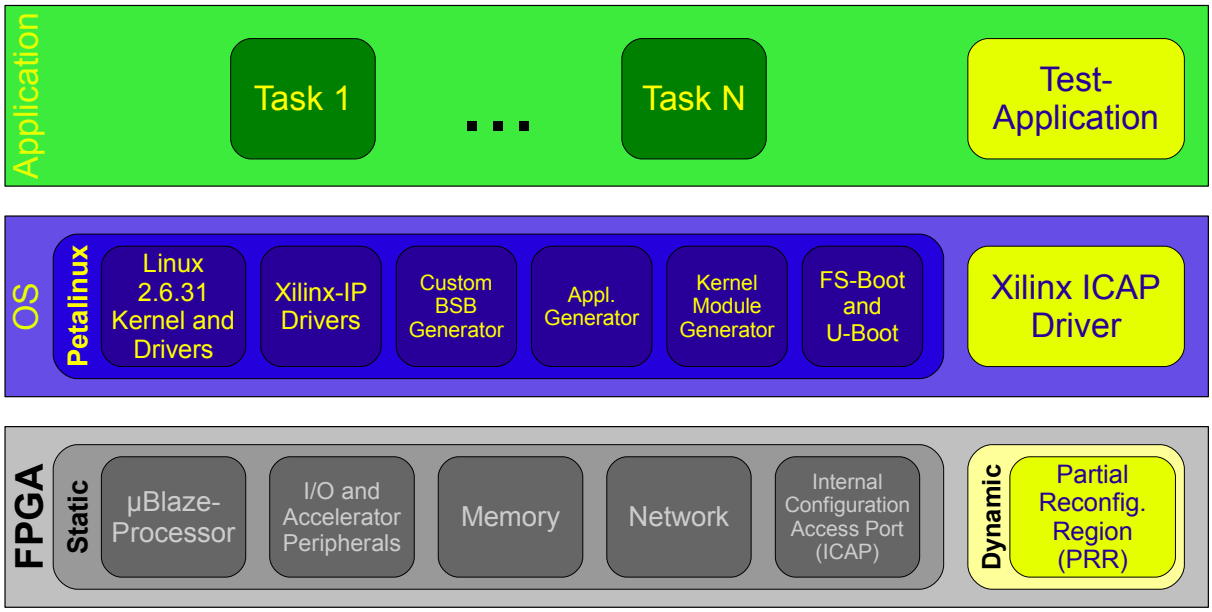


Abbildung 1.1.: FPGA-basiertes SoC; µController-System mit dynamisch rekonfigurierbaren IP-Funktionen; Betriebssystem und Anwender-Tasks

Die HW-Konfiguration wird aus einen statischem und mehreren dynamischen Teilen gebildet. Der statische Teil des Systems setzt sich aus allen Modulen zusammen, deren Funktionalitäten zur Laufzeit immer aktiv sein müssen. Der dynamische Teil lässt sich neu konfigurieren ohne den statischen Teil des Systems in seinem Betrieb zu unterbrechen. Die partiell rekonfigurierbare Region (PRR) ist ein lokaler Bereich des FPGAs, in dem rekonfigurierbare Module (PRM) platziert werden. Die PRMs implementieren verschiedene Funktionalitäten und werden je nach Anforderung durch die Anwendung dynamisch geladen. Der dynamische Teil des Systems setzt sich somit aus einer PRR und seinen zugehörigen PRMs zusammen (vgl. Abbildung 1.2):

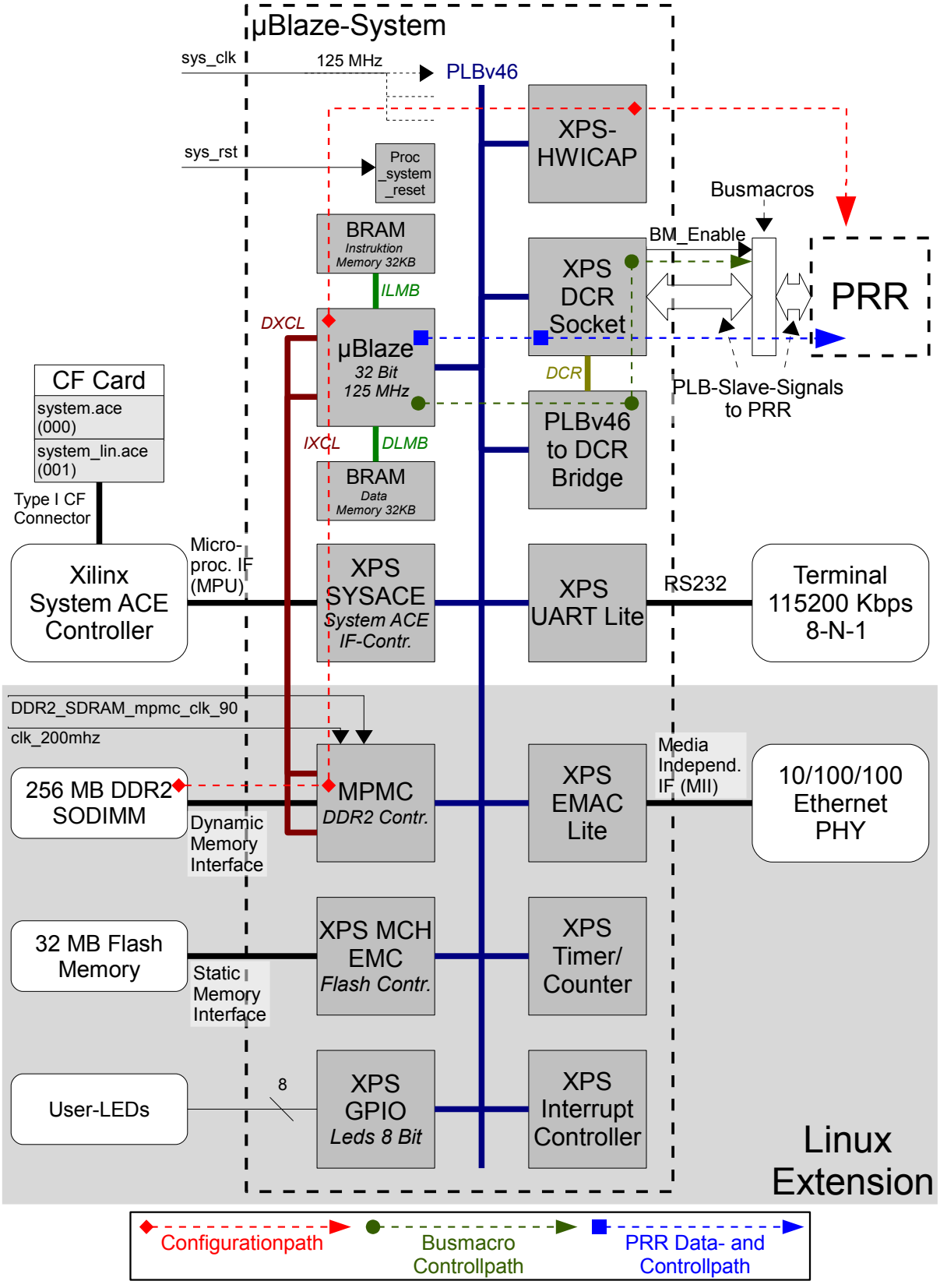


Abbildung 1.2.: Entwickeltes selbstrekonfigurierendes µBlaze-System

µBlaze-Prozessor Dieser 32-Bit RISC Prozessor, der in dieser Konfiguration mit 125 MHz betrieben wird, bildet die Grundlage zur Ausführung von SW.

XPS DCR Socket Statische Schnittstelle zur Anbindung der PRR an den PLB.

Busmacros Komponente zur Sicherung/Sperrung des Signalverlaufs zwischen statischem System und der PRR.

PLBV46 to DCR Bridge Diese Komponente erlaubt einen SW-seitigen Zugriff auf ein Register des XPS_DCR_Sockets zur Steuerung des Busmacro-Enable-Signals.

math Dynamischer Teil des Systems, der aus einer PRR und zwei PRMs besteht, die jeweils eine Multiplikation und Addition implementieren.

XPS-HWICAP Das Internal Configuration Access Port (ICAP) ist eine Bibliothekskomponente des Virtex 5 FPGAs, die eine Schnittstelle zur internen Rekonfiguration bietet [36]. Die XPS-HWICAP-Komponente instanziiert diese ICAP-Ressource des FPGAs. Zusätzlich stellt dieses Modul einen Zugang über den PLB zur Verfügung, sodass sich eine interne Rekonfiguration durch µBlaze-SW durchführen lässt.

XPS Multi Channel EMC Über diesen Speicherkontroller wird ein externer Flashspeicher angekoppelt. Der U-Boot Bootloader und der Linux-Kernel werden in diesem Speicher abgelegt, damit der Linux-Kernel nach jeder Stromunterbrechung oder Reset automatisch geladen wird.

Multiport Memory Controller (MPMC) Dient dem µBlaze-System zur Anbindung eines DDR2-RAMs, das den Arbeitsspeicher des Systems erweitert.

XPS UART Lite Serielle Schnittstelle zur Ein- und Ausgabe. Wird zur Kommunikation mit einem Terminalprogramm verwendet.

XPS-GPIO Steuerung von LEDs zur Anzeige des SW-Status.

XPS EMAC Lite Schnittstelle über Ethernet. Mit einer Transfergeschwindigkeit von 100 Mbps lassen sich neue Kernel-Images laden sowie Daten zwischen Entwicklungsrechner und Linux-Kernel austauschen.

XPS SYSACE Schnittstellen-Controller zur Anbindung von Compact-Flash Speicherkarten. Die Bitstreams (Konfigurationsdaten) des Systems sowie der PRMs werden auf dieser Speicherkarte abgelegt und zur Rekonfiguration gelesen.

XPS Interrupt Controller Zuführung der Interruptquellen vom UART-Lite, Ethernet-Lite und Timer als gebündeltes Signal zum µBlaze-Prozessor.

XPS Timer/Counter Liefert die Timer-Ticks für die Ausführung von Linux.

Zur Entwicklung des SoC nach Abb. 1.2 wurden mit dieser Arbeit die Methoden zur partiellen und dynamischen Rekonfiguration erprobt und evaluiert. Aus den sich daraus ergebenden Erkenntnissen soll die PR-Technologie zukünftigen Projekten der HAW Hamburg zugänglich gemacht werden. Die Arbeitsschwerpunkte waren:

1. Zusammenstellung und Klassifizierung von Rekonfigurationsarchitekturen und -schnittstellen.
2. Evaluierung des Entwurfsablaufs zur Erstellung eines partiell rekonfigurierbaren Systems und der Erprobung der dazu erforderlichen Werkzeuge. Die daraus resultierenden Erkenntnisse sind zu einem detaillierteren Entwurfsablauf zusammengestellt worden.

3. Entwurf und Implementierung einer HW-Plattform auf Basis einer selbstrekonfigurierbaren SoC-Architektur.
4. Konfiguration und Installation eines Linux-Kernels für die entwickelte HW-Plattform mit der Petalinux-Distribution.
5. Anpassung des Linux-Treibers zur Ansteuerung der Rekonfigurationsschnittstelle sowie der Erstellung einer Testanwendung zur Verifikation der Rekonfigurationsfunktionalität.

Der knappe Umfang der verfügbaren Dokumentation sowie der eingeschränkte Zugang zu den Xilinx Werkzeugen für die partielle Rekonfiguration zeigen, dass diese Technologie noch nicht ausgereift ist und sich noch nicht etablieren kann [34], [41]. Übersichten und Klassifizierungen von Rekonfigurationsarchitekturen und -schnittstellen sind bislang in der Fachliteratur nicht aufgearbeitet worden. Die in dieser Arbeit erstellte Übersicht zu Rekonfigurationsarchitekturen und ihren Eigenschaften stellt für zukünftige Projekte eine Entscheidungsvorlage dar.

Weiterhin ist der von Xilinx empfohlene Ablauf für einen partiell rekonfigurierbaren Entwurf noch gering erprobt und bietet somit nur einen erschwerten Zugang für neue Anwendungen [34]. Von Xilinx wird dieser Ablauf für allgemeine Entwürfe beschrieben und ist auf keine Rekonfigurationsarchitektur abgestimmt. Zudem beruht die von Xilinx bereitgestellte Anleitungsdokumentation zur Erstellung eines partiell rekonfigurierbaren SoCs auf dem Entwurfswerkzeug Planahead [35], [5]. Der Erwerb dieses Werkzeugs unterliegt jedoch weiteren Bedingungen, welche im Rahmen dieser Arbeit nicht erfüllt werden konnten. Diese Arbeit stellt einen Entwurfsablauf zusammen, der zum Aufbau und zur Implementierung eines partiell rekonfigurierbaren SoCs anhand der verfügbaren Werkzeugkette dient und neuen Anwendungen einen leichteren Zugang bietet.

Diese Dokumentation gliedert sich in folgende Kapitel:

- Ein Überblick zu den Re- bzw. Konfigurationstechniken eines Xilinx FPGAs wird in Kapitel 2 gegeben. Die Methodik der partiellen Rekonfiguration wird veranschaulicht und die Unterschiede zur vollständigen Konfiguration gezeigt.
- In Kapitel 3 werden die Rekonfigurationsarchitekturen und ihre Komponenten detailliert dargestellt und erläutert. Anhand ihrer unterschiedlichen Eigenschaften werden die Architekturen klassifiziert.
- Der Aufbau des erstellten µBlaze-HW-Systems wird in Kapitel 4 dokumentiert. Die Integration des HW-Systems in die Xilinx ML-505-Evaluierungsplattform und die Verifikation der Funktionsweise werden beschrieben.
- In Kapitel 5 wird das µBlaze-HW-System aus Kapitel 4 zur Integration eines Linux-Kernels erweitert. Die Generierung des µCLinux-Kernels mit der Petalinux-Entwicklungsumgebung für das µBlaze-System wird dokumentiert. Der xilinx_hwicap-Treiber wird in das SW-System integriert. Die Funktionsweise dieses Treibers sowie die daran vorgenommenen Anpassungen werden erläutert. Die partielle Rekonfiguration unter dem Linux-Betriebssystem wird durch eine Test-Anwendung verifiziert.
- In Kapitel 6 werden zwei Entwurfsabläufe für die Erstellung eines selbsrekonfigurierenden µBlaze-Systems aufgezeigt. Die einzelnen Schritte zum Aufbau werden erläutert und die Entwurfsdateien aufgelistet.
- Kapitel 7 gibt eine Zusammenfassung über diese Dokumentation.

2. Rekonfigurationstechniken eines FPGAs

Dieses Kapitel gibt eine Übersicht zu den Re- bzw. Konfigurationsmethoden. Die wesentlichen Unterschiede sowie die verwendeten Begriffe werden vorgestellt und erläutert. Beginnend mit der Veranschaulichung der allgemeinen Konfiguration eines FPGAs werden die auf diesem Prinzip aufbauenden Rekonfigurationstechniken bis zur partiellen und dynamischen Rekonfiguration hergeleitet.

2.1. Gesamtrekonfiguration und partielle Rekonfiguration

Ein SRAM-basierter FPGA ist eine zwei-schichtige Einheit (siehe Abbildung 2.1). Die untere Schicht ist ein SRAM-Speicher, in dem die Konfigurationsdaten abgelegt sind. Die obere Schicht ist die Logik-Ebene. Diese besteht aus universellen Blöcken, mit denen sich logische Schaltungen nachbilden lassen (vgl. Anhang B). Die aktuelle Konfiguration in der Konfigurationsspeicherschicht legt die Funktionsstruktur dieser Blöcke und deren Verschaltung untereinander fest. Eine digitale Schaltung wird in einem FPGA aus der Komposition dieser Blöcke mit einer jeweils definierten Funktionsstruktur realisiert [6].

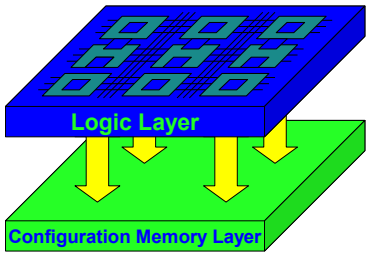


Abbildung 2.1.: Abstraktionsschichten eines FPGAs

Der Konfigurationsspeicher eines Virtex 5 FPGAs ist in Konfigurationseinheiten unterteilt [42]. Diese sind die kleinsten adressierbaren Segmente eines Virtex 5 FPGAs. Eine Konfiguration besteht aus einem oder mehreren vollen Segmenten. Somit umspannt die Konfiguration eines digitalen Systems einen bestimmten Bereich eines Virtex 5 FPGAs. Dieser Bereich ist abhängig vom Ressourcenbedarf des digitalen Systems. Die Konfiguration eines FPGAs ist demnach das Ablegen von Konfigurationsdaten im Konfigurationsspeicher des FPGAs [6]. Der Zustand eines Virtex 5 FPGAs ohne eine Konfiguration lässt sich als ein Zustand nach einer Stromunterbrechung betrachten, da diese lediglich über flüchtigen Speicher verfügen.

Bei einer Gesamtkonfiguration werden Konfigurationsdaten in den SRAM-Speicher des FPGAs geladen [6]. Jegliche Konfiguration, die sich davor in diesem Speicher befand, ist hiernach nicht mehr vorhanden (vgl. Abbildung 2.2).

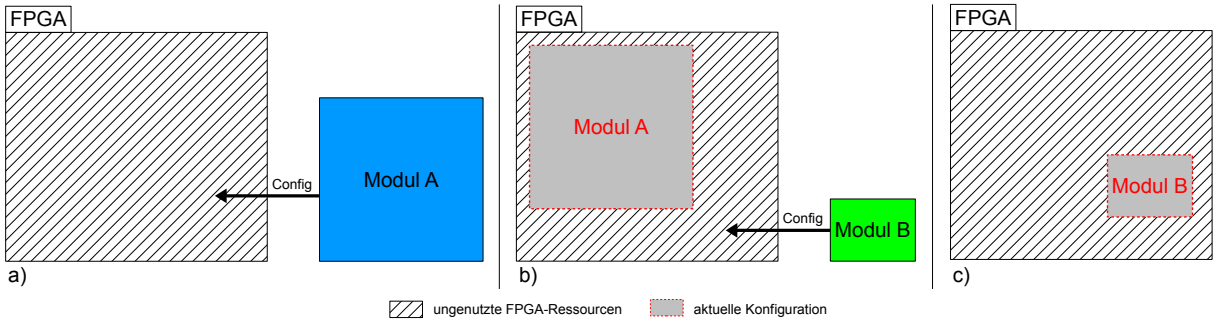


Abbildung 2.2.: Gesamtrekonfiguration eines FPGAs. a) FPGA wird mit System A konfiguriert; b) System A ist aktuelle Konfiguration des FPGAs; c) System B ist aktuelle Konfiguration des FPGAs

Es werden folgende einzelne Schritte durchgeführt:

a) Eine Konfiguration, die das digitale System A beschreibt soll in das FPGA geladen werden. In diesem Fall befand sich davor keine andere Konfiguration im Speicher des FPGAs. System A besteht aus mehreren Konfigurationseinheiten, die einen bestimmten Adressbereich aufspannen. Dieser Adressbereich legt die Position der Konfiguration im FPGA mit Koordinaten fest [38]. Diese Position hängt davon ab, welche FPGA-Ressourcen von System A beansprucht werden.

b) Nach der Konfiguration wird die Funktionsstruktur von System A im FPGA abgebildet. Soll als nächstes eine Konfiguration durch ein weniger Konfigurationseinheiten umfassendes System B erfolgen, so wird die Konfiguration von System A mit der von System B ersetzt, auch wenn genügend Ressourcen zur Integration beider Systeme vorhanden sind.

c) Nach dieser Prozedur wird einzig die Funktiosstruktur von System B im FPGA abgebildet.

Das Hauptcharakteristikum der Gesamtkonfiguration ist der Ersatz der alten Konfiguration durch die neue. Somit wird auch bei kleinen Änderungen an einer digitalen Schaltung immer der gesamte Entwurf rekonfiguriert.

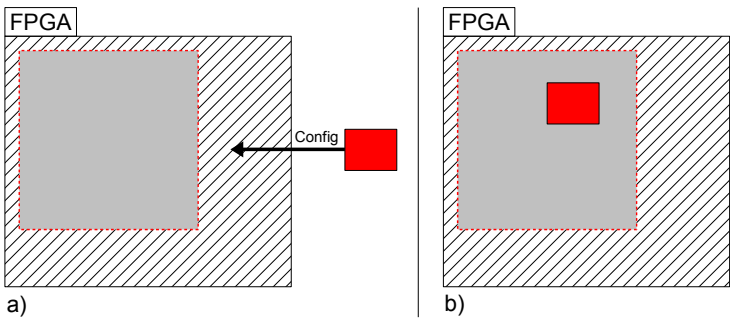


Abbildung 2.3.: Partielle Rekonfiguration eines FPGAs. a) Ein Teil der Konfiguration soll geändert werden; b) Die Konfiguration wurde nur in dem Bereich der Veränderung überschrieben

Bei der partiellen Rekonfiguration werden ein oder mehrere Teile einer Konfiguration geändert. Die Änderung der gesamten Konfiguration ist nicht erforderlich (vgl. Abbildung 2.3):

a) Im FPGA ist bereits eine Konfiguration abgebildet. Es soll ein Teil der digitalen Schaltung, die von dieser Konfiguration implementiert wird, geändert werden.

b) Der Konfigurationsspeicher des FPGAs wird mit Konfigurationsdaten geladen, die nur diese Veränderung betreffen. Es werden also Konfigurationseinheiten an die Positionen im Konfigurationsspeicher des FPGAs geschrieben, die sich auf die Veränderung am System beziehen. Einige Bereiche der alten Konfiguration werden überschrieben, sodass ein neues Schaltungsverhalten ohne eine Gesamtkonfiguration erlangt wird.

2.2. Dynamische Rekonfiguration

Die dynamische Rekonfiguration lässt sich durch folgende Charakteristika beschreiben [6]:

- Die Anwendung, die dieses System implementiert, teilt sich in einen statischen und dynamischen Teil.
- Es wird nur der dynamische Teil rekonfiguriert und nicht das gesamte System.
- Der dynamische Teil wird zur Laufzeit des Systems rekonfiguriert. Die Rekonfiguration unterbricht den Betrieb des System nicht.
- Die Mechanismen zur Rekonfiguration werden vom statischen Teil ausgeführt.
- Der dynamische Teil der Anwendung wird als digitale Schaltung in einem FPGA realisiert.

Die dynamische Rekonfiguration lässt sich je nach Architektur durch eine partielle Rekonfiguration oder als Gesamtrekonfiguration realisieren. Ein System lässt sich als dynamisch rekonfigurierbar charakterisieren, wenn der Betrieb des Systems durch die Rekonfiguration nicht unterbrochen wird [18].

2.3. Partielle und dynamische Rekonfiguration

Die partielle und dynamische Rekonfiguration ist eine Kombination aus beiden Rekonfigurationsmechanismen. In einem partiell und dynamisch rekonfigurierbarem System werden die Vorteile beider Verfahren genutzt. Diese einzelnen Rekonfigurationstechniken beschreiben völlig unterschiedliche Eigenschaften der Rekonfiguration. Ein rein dynamisch rekonfigurierbares System führt demnach eine Gesamtrekonfiguration durch, wohingegen ein rein partiell rekonfigurierbares System wiederum zu einer Unterbrechung der Laufzeit führt.

Ein partiell und dynamisch rekonfigurierbares SoC zeichnet sich durch folgende Eigenschaften aus:

- Das SoC teilt sich in einen statischen und dynamischen Teil auf.
- Die Rekonfiguration ändert den dynamischen Teil des Systems.

- Während der Rekonfiguration wird das SoC in seinem Betrieb nicht unterbrochen.
- Die Mechanismen zur Rekonfiguration werden vom statischen Teil ausgeführt.
- Der dynamische Teil der Anwendung und mindestens ein Teil des statischen Systems werden als digitale Schaltung in einem FPGA realisiert.

Der Unterschied zur rein dynamischen Rekonfiguration liegt darin, dass nicht nur der dynamische Teil einer Anwendung sondern auch mindestens ein Teil des statischen Anwendungsparts als FPGA-Implementierung realisiert werden [3]. Zweckmäßig ist daher die reine dynamische Rekonfiguration oder, im Fall der partiellen und dynamischen Rekonfiguration, die Integration aller Komponenten als FPGA-Implementierung.

Der Fokus dieser Arbeit liegt auf der partiellen und dynamischen Rekonfiguration von Virtex 5 FPGAs:

Partiell Rekonfigurierbare Region (PRR): Ein Bereich des FPGAs, der von einem dynamischen Teil gänzlich bzw. Ressourcen des Bereiches verwendet.

Partiell Rekonfigurierbare Module (PRM): Die speziellen Konfigurationen eines dynamischen Moduls, die eine PRR mit Funktionen füllen. Die Zugehörigkeit von PRMs zu einer PRR wird in der Entwurfsphase durch die Spezifikation der Schnittstellen zwischen dem dynamischen und statischen Teil festgelegt.

Initialkonfiguration: Eine Kombination aus statischem Teil und einer PRM, die für den Systemstart die erste Konfiguration des FPGAs bilden.

Die Vorgang der partiellen und dynamischen Rekonfiguration gliedert sich in drei Teilschritte (siehe Abbildung 2.4):

1. Das FPGA ist mit der Initialkonfiguration belegt.
2. Die PRR der Initialkonfiguration soll mit PRM A konfiguriert werden. Der Konfigurationsspeicher des FPGAs wird im Bereich der PRR mit den Konfigurationsdaten von PRM A überschrieben. Die Konfiguration und damit der Betrieb des statischen Teils wird nicht beeinflusst.
3. In der PRR wird die durch PRM A definierte Funktionsstruktur abgebildet. PRM A wird nun durch PRM B ersetzt. Die FPGA-Ressourcen im Bereich der PRR werden jetzt von PRM B verwendet bzw. konfiguriert.

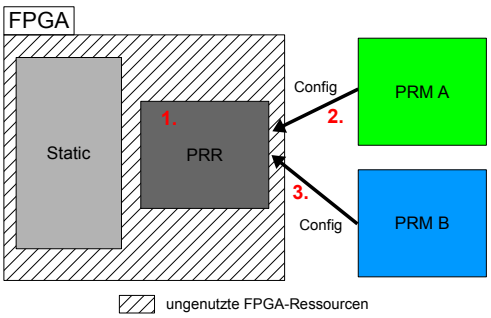


Abbildung 2.4.: Partielle und dynamische Rekonfiguration eines FPGAs. 1. Initialkonfiguration; 2. Rekonfiguration der PRR mit PRM A; 3. Rekonfiguration der PRR mit PRM B

3. SoC-Architekturen zur partiellen Rekonfiguration

In diesem Kapitel werden die verschiedenen Rekonfigurationsarchitekturen dargestellt. Es wird mit einer tabellarischen Übersicht begonnen, mit der sich die Rekonfigurationsarchitekturen und ihre groben Unterschiede überblicken lassen. Darauf folgend werden in jedem Abschnitt zur jeweiligen Rekonfigurationsarchitektur, ihre Eigenschaften spezifiziert, die Komponenten aufgezeigt und die verschiedenen Implementierungsvarianten genannt.

Ein partiell rekonfigurierbares System besteht aus einem Rekonfigurationskontroller und der eigentlichen Anwendung, die dieses System implementiert. Sämtliche Architekturen lassen sich in folgende Bestandteile gliedern (vgl. Abbildung 3.1):

Konfigurationskontroller:

- Implementiert die Mechanismen zur Rekonfiguration.
- Erhält den Auftrag zur Rekonfiguration vom statischen Anwendungsteil.
- Ist für den Zugriff auf die Rekonfigurationsschnittstelle verantwortlich.
- Ist statischer Bestandteil des Gesamtsystems.

Statischer Anwendungsteil:

- Realisiert die Basisfunktionalität des Systems.
- Die implementierte Funktionalität ist während der gesamten Laufzeit aktiv.
- Initiiert die Rekonfiguration.

Dynamische Anwendungsteile:

- Eigenständige Module, die jeweils verschiedene Funktionalitäten implementieren.
- Ersetzen sich während der Rekonfiguration gegenseitig.

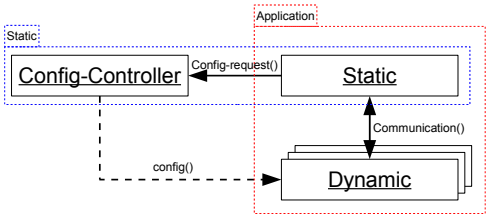


Abbildung 3.1.: Übersicht zur Rekonfigurationsarchitektur mit drei Funktionseinheiten

Die Unterschiede der Rekonfigurationsarchitekturen prägen sich in folgenden Punkten aus (vgl. Tabelle 3.1):

- **Implementierungsart:** Ausgenommen vom dynamischen Anwendungsteil lassen sich sämtliche Komponenten dieser Architekturen jeweils als SW- oder HW-Entwurf realisieren.
- **Ausführungslokalität:** Der dynamische Anwendungsteil wird als FPGA-Implementierung realisiert. Der statische Teil des Systems lässt sich entweder als interne FPGA- oder externe Prozessor- bzw. ASIC-Anwendung ausführen.
- **Konfigurationsschnittstelle:** Ein Virtex 5 FPGA verfügt über sechs Konfigurationsschnittstellen [42]. Jede Schnittstelle bietet eine oder mehrere Konfigurationsmodi und verschiedene Busbreiten für die Konfigurationsdaten. Die Wahl der Konfigurationsschnittstelle hängt von der Implementierungsart und der Ausführungsplattform des Rekonfigurationskontrollers ab.

Eigenschaft / **Architektur**	Ausführung der Rekonfiguration	Implementierung des Rekonfiguraitonskontrollers	Implementierung der stat. Anwendung	Rekonfigurationsschnittstelle
Benutzer-rekonfigurierbares HW-System	extern	SW	HW	Ser. Konfig.-schnittstelle, JTAG, SelectMap
Selbstrekonf. HW-SW-System	extern	SW	SW	Ser. Konfig.-schnittstelle, JTAG, SelectMap, SPI, BPI
FSM-Rekonfigurierbares HW-Systems	intern	HW	HW	Internal Configuration Access Port
Selbstrekonf. SoC	**intern**	**SW**	**HW-IPs u. µBlaze-SW**	**ICAP über XPS-HWICAP-IP**

Tabelle 3.1.: Übersicht zu den Rekonfigurationsarchitekturen und deren Eigenschaften. Hervorgehoben ist die Architektur, die in dieser Arbeit analysiert und bearbeitet wird.

3.1. Intern rekonfigurierbare Architekturen

Die intern rekonfigurierbaren Architekturen zeichnen sich durch folgende Eigenschaften aus:

- Der Konfigurationskontroller ist eine FPGA-Implementierung.
- Alle Einheiten eines Systems sind im FPGA integriert.
- Ausführung der Rekonfiguration über die Internal Configuration Access Port (ICAP).
- Keine externe Prozessor- oder ASIC-Einheit erforderlich.

Die Unterschiede der jeweiligen internen Architekturen liegen in der Implementierungsart des Rekonfigurationskontrollers sowie der Anbindung der PRR zum statischen Teil des Systems.

3.1.1. Selbstrekonfigurierendes SoC mit unterschiedlicher Busanbindung der PRR

In diesem Abschnitt wird zunächst der Aufbau und die Funktionsweise der selbstrekonfigurierenden SoC-Architektur erläutert. Die in ihrer Implementierung verschiedenen PRMs sind Bestandteile des Systems und daher an der Gesamtfunktionalität des Systems beteiligt. Im Weiteren wird auf die Verbindung der PRMs zum statischen Teil und die verschiedenen Varianten eingegangen.

Hauptanwendungsfeld dieser Architektur sind eingebettete HW/SW-Systeme. Anwendungen dieser Systeme lassen sich als reine SW, reine HW oder ein Verbund beider implementieren. Durch die PR-Technologie lassen sich HW-Module laden, die eigenständig eine Teilfunktion des Systems implementieren. Es lassen sich ebenso SW-basierte Anwendungen durch das Laden von HW-Modulen beschleunigen. In diesem Fall wird die Teilfunktion des Systems aus dem Verbund der SW- und der HW-Implementierung erfüllt.

Diese Architektur ist basierend auf einer partiellen und dynamischen Rekonfiguration aus folgenden Gründen sinnvoll:

- Eine rein partielle Rekonfiguration bedeutet, den Betrieb des restlichen Systems zu unterbrechen. Damit ist jede parallele Teileinheit des Systems gezwungen, seine Operationen während der Rekonfiguration anzuhalten.
- Eine rein dynamische Rekonfiguration ist für selbstrekonfigurierende SoCs nicht möglich, da die Integration des statischen Teil des Systems und der PRR im FPGA erforderlich ist.

Die Rekonfiguration einer selbstrekonfigurierenden SoC-Architektur erfolgt SW-gesteuert. Zur Ausführung der SW wird in diesen Systemen der Xilinx µBlazeProzessor verwendet. Die Architektur setzt sich aus folgenden Komponenten zusammen (siehe Abbildung 3.2):

µBlaze-Prozessor: Plattform zur Ausführung der SW.

Speicher: Interner oder externer Speicher, der mittels einer Speicherschnittstelle über den PLB zugreifbar ist. Die Konfigurationsdaten werden in diesem Speicher gespeichert.

XPS-HWICAP: Diese IP bietet dem µBlaze-Prozessor Lese- und Schreibfunktionen der FPGA-Konfiguration über die ICAP-Schnittstelle [30].

Processor Local Bus (PLB): Verbindungsplattform der einzelnen µController-Funktionselemente für einen Memory-mapped SW-Zugriff [27].

Digital Clock Manager (DCM): FPGA-Bibliotheksmodul zur Erzeugung von Taktsignalen mit parametrisierbarer Phasenlage [43].

Busmacros (BM): Freigabemodul zur Steuerung der Schnittstellensignale zwischen PRR und statischem Teil [34].

Statische Schnittstelle zur PRR: Schnittstelle zur Anbindung der PRR an den PLB.

PRR: Definierter FPGA-Bereich für austauschbare Beschleuniger-IPs.

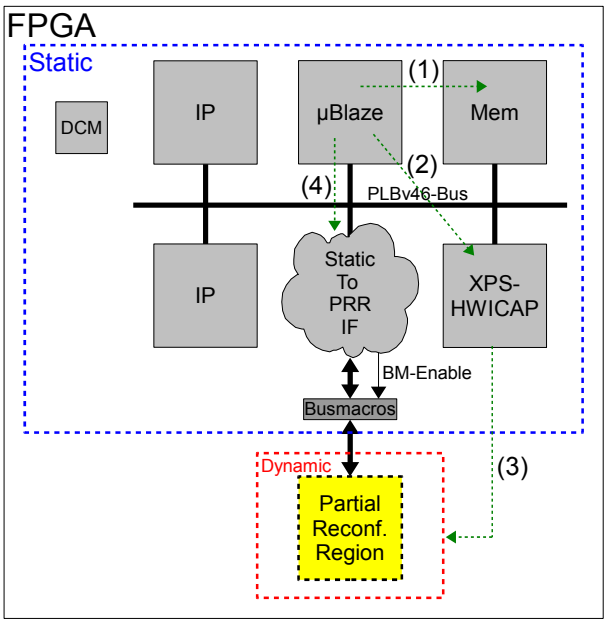


Abbildung 3.2.: Struktur eines selbstrekonfigurierenden SoC mit statischen und dynamischen IP-Modulen

Eine Kombination aus µBlaze-Prozessor, IPs und dem PLB lässt sich als µBlaze-System zusammenfassen. Den Kern dieses Systems bildet der µBlaze-Prozessor. Dieser ist als Busmaster an den PLB angeschlossen und führt SW-Anwendungen aus. Alle am PLB angeschlossenen IPs lassen sich über diese SW steuern.

Die Konfigurationsdaten (Bitstreams) sind in einem internen oder externen Speicher abzulegen. Die Speicherschnittstelle (vgl. Abb. 3.2, “Mem“) besitzt eine PLB-Slave-Schnittstelle und somit einen SW-seitigen Zugriff.

Die XPS-HWICAP Komponente bietet einen Schnittstellen-Rahmen (Wrapper) um die eigentliche ICAP-Schnittstelle. In diesem Rahmen sind die PLB-Slave-Schnittstelle, Status- und Kontrollregister und weitere Komponenten für den SW-seitigen Zugriff auf die Rekonfigurationsschnittstelle enthalten. Diese IP stellt die Funktionalität zum Lesen und Schreiben der FPGA-Konfiguration bereit. Das XPS-HWICAP stellt für zu Schreibende und Lesende Konfigurationsdaten jeweils 1024x32-Bit Fifos bereit. Die Bitstreams der PRMs werden in diesen Fifos gespeichert. Wenn der Rekonfigurationskontroller den Befehl zur Rekonfiguration sendet, werden die Konfigurationsdaten aus dem Fifo an das ICAP gesendet.

Die DCM-Komponente ist ein parametrisierbarer Baustein der Xilinx FPGAs, mit dem sich die Taktraten aus einem externen Referenztaktsignal erzeugen lassen [36].

Busmacros stellen die Pinkompatibilität zwischen dem statischen Teil und den einzelnen PRMs her. Für alle PRMs wird dadurch die Zuordnung eines PRR-Signals zu einem des statischen Systemteils festgelegt. Signale einer PRM in Richtung des statischen Bereiches werden über ein Enable-Signal gesteuert. Während der Rekonfiguration setzt diese Enable-Funktion die Signale, die von den Busmacros an das statische Teilsystem gesendet werden, auf eine logische “0“, sodass das statische Teilsystem nicht gestört wird. Das Verhalten eines Ein- und Ausgangssignals eines Busmacros wird über eine LUT6-Primitive des Virtex 5 FPGAs festgelegt [36]. Diese Look-Up-Tables werden für eine Multiplexer-Funktionalität zur Steuerung der Ausgangssignale spezifiziert (vgl. Anhang B).

Die Funktionsweise der Rekonfiguration in dieser Architektur ist unabhängig von der statischen Schnittstelle zur PRR. Der Rekonfigurationskontroller ist als SW implementiert, die auf dem µBlaze-Prozessor ausgeführt wird. Dabei erfolgt jeder Rekonfigurationsablauf wie folgt (siehe Abbildung 3.2):

1. Der Rekonfigurationskontroller liest die Rekonfigurationsdaten über die Speicherschnittstelle. Diese werden im Arbeitsspeicher des Systems zwischengespeichert.
2. Das Busmacro-Enable-Signal wird auf Low gesetzt, wodurch die PRR vom statischen Teil entkoppelt wird. Die Bitstreams werden in 32-Bit Blöcken unterteilt und an das XPS-HWICAP gesendet, der diese im Schreib-Fifo speichert.
3. Erhält XPS-HWICAP über den Rekonfigurationskontroller den Befehl zur Rekonfiguration, wird die Rekonfiguration mit dem Senden der Konfigurationsdaten aus dem Schreib-Fifo an das ICAP begonnen. Der Status der Rekonfiguration wird vom Rekonfigurationskontroller aus dem Statusregister des XPS-HWICAP gelesen. Dieser Prozess wird solange durchgeführt, bis alle Konfigurationsdaten an das ICAP gesendet worden sind.
4. Bei erfolgreicher Rekonfiguration wird das Busmacro-Enable-Signal auf High gesetzt. Die PRM ist somit mit dem statischen System verbunden. Die Kommunikation zwischen statischem Teil und der PRM ist ab diesem Zeitpunkt durchführbar.

Die folgenden Varianten der selbstrekonfigurierenden SoC-Architektur unterscheiden sich in der Anbindung der PRR an den statischen Teil des Systems. Die PRMs werden für einen SW-seitigen Zugriff mit einer PLB-Slave-Schnittstelle ausgestattet.

Wird ein µBlaze-System erstellt, passt XPS den PLB automatisch an das System an:

- Für jede am PLB angeschlossene Komponente wird eine Adresszuordnung generiert.
- Die Breite der PLB-Signale wird denen der Komponenten angeglichen.
- Die Signale der Komponenten und des PLBs werden einander zugeordnet.

Das Herausführen der PLB-Signale aus dem µBlaze-System für eine weitere Komponente erfordert die manuelle Manipulation der MHS-Datei und der VHD-Datei zur Instanziierung der Systemkomponenten. Diese Methode ist sehr aufwändig und erhöht die Entwurfszeit. Eine andere Methode ist die Instanziierung einer vordefinierten IP für das µBlaze-System mit XPS, welche folgende Vorteile gegenüber einer manuellen Manipulation des PLBs hat:

- Die Verbindung zwischen dem statischem System und der PRR wird durch eine weitere IP realisiert. Die Anpassung des PLBs erfolgt somit automatisch durch XPS.
- Die Bus-Signale für die PRR werden durch diese IP außerhalb des µBlaze-Systems zugänglich gemacht.
- Die statische Schnittstelle stellt die erforderlichen Ressourcen zur Anbindung der PRR an den PLB sowie die Steuersignale für die Busmacros zur Verfügung.

In den folgenden Abschnitten werden zwei Entwurfsvarianten zur Anbindung einer PRR an ein µBlaze-System vorgestellt. Je nach Entwurfsvariante unterscheidet sich die Position folgender Komponenten in der Entwurfshierarchie:

- Die PRR, Busmacros und die DCMs lassen sich in derselben Hierarchieebene parallel zum µBlaze-System instanziieren.
- Die PRR und die DCMs lassen sich als direkte Peripherie des µBlaze-Systems und die Busmacros als Komponente der statischen Schnittstelle instanziieren.

3.1.1.1. XPS-DCR-Socket

Bei der Anbindung der PRR über das XPS_DCR_Socket an den PLB wird die PRR in derselben Hierarchieebene wie das µBlaze-System instanziiert. Das µBlaze-System und die PRR sowie die Busmacros sind folglich Bestandteile einer übergeordneten Einheit (Top-Entity) (siehe Abbildung 3.3).

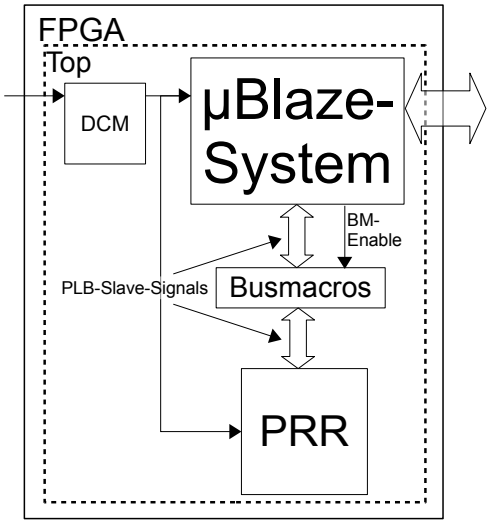


Abbildung 3.3.: Selbstrekonfigurierendes SoC mit der PRR, den Busmacros und dem µBlaze-System jeweils als parallele Bestandteile der Top-Entity

Folgende Bestandteile des µBlaze-Systems bilden die statische Schnittstelle zur PRR (vgl. Abbildung 3.4).

- Das von Xilinx im Rahmen der partiellen Rekonfiguration bereitgestellte **XPS_DCR_Socket** macht die PLB-Signale nach Außen verfügbar [41]. Somit lässt sich eine PRR über die Top-Entity an das µBlaze-System anschließen. Durch die Instanziierung des XPS_DCR_Sockets mit XPS wird der PLB somit für eine weitere Komponente angepasst.
- Die **DCR-Bridge** setzt Kommunikationstransaktionen, die an seine PLB-Schnittstelle empfangen werden, in DCR-Master-Operationen um [26]. Diese Komponente erlaubt einen SW-seitigen Zugriff auf Register des XPS_DCR_Sockets.
- Der **DCR-Bus** dient als Kommunikationsplattform für ein DCR-Master und einem oder mehreren DCR-Slaves [22]. Im µBlaze-System stellt der DCR-Bus die Verbindung zwischen dem DCR-Bridge als Master und dem XPS_DCR_Socket her. Diese Schnittstelle wird für den Zugriff auf die Status- und Kontrollregister innerhalb der PLB-Master und -Slaves verwendet.

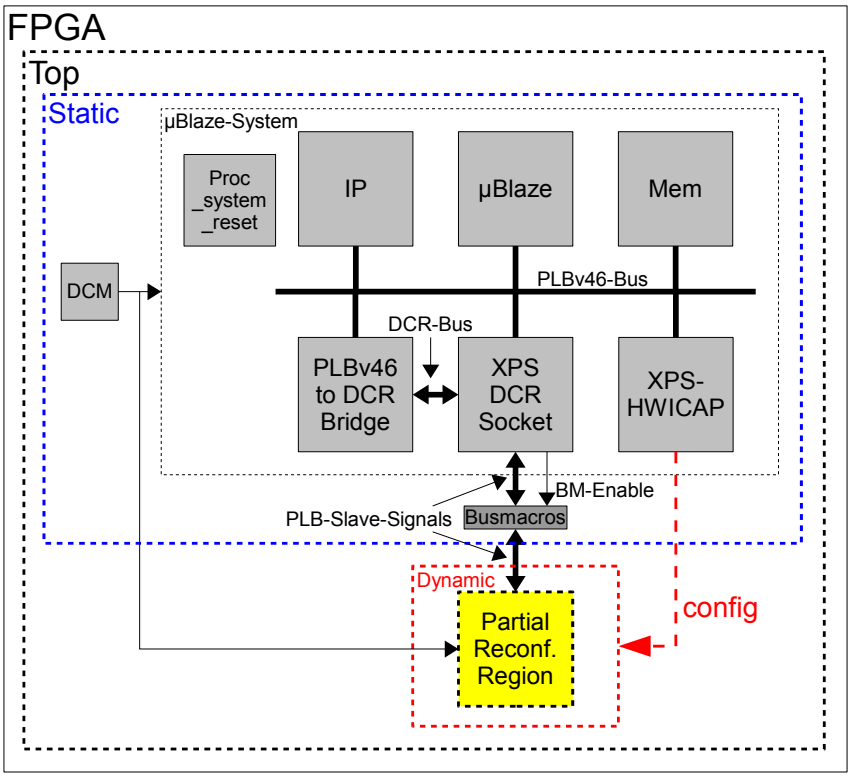


Abbildung 3.4.: Selbstrekonfigurierendes SoC mit dem XPS_DCR_Socket als statische Schnittstelle zum PRM außerhalb des µBlaze-Systems

Den Steuerpfad für das Busmacro-Enable-Signal bildet die DCR-Schnittstelle des XPS_DCR_Sockets (vgl. Abbildung 3.5). Diese Schnittstelle stellt ein 32-Bit Datensignal bereit, mit dessen höchstwertigstem Bit (MSB) das Busmacro-Enable-Signal gesteuert wird.

Für diese Architektur sind die DCM-Komponente und die Busmacros als Bestandteil der Top-Entity zu instanziieren [34]. Das µBlaze-System ist eine eigenständige Einheit. Die Schnittstelle des µBlaze-Systems definiert die Signale, die zur Kommunikation mit Komponenten außerhalb des FPGAs, zur Kommunikation mit der PRR sowie zur Steuerung der Busmacros erforderlich sind.

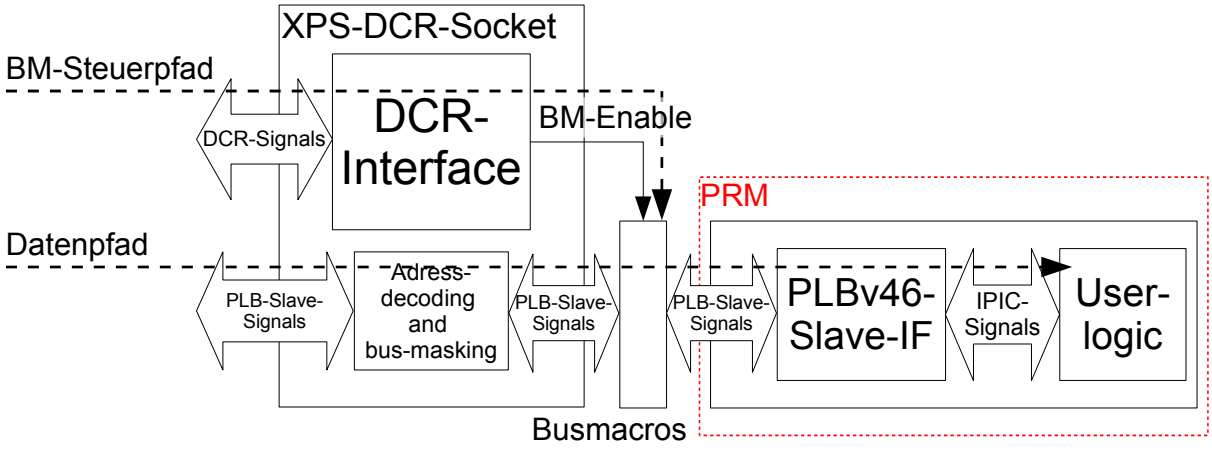


Abbildung 3.5.: XPS_DCR_Socket, transportiert Bussignale an die PRR und steuert die Busmacros

Die PRMs haben dieselbe Struktur wie eine PLB-Slave-Komponente (siehe Abbildung 3.5, rechts). Diese bestehen aus der PLBv46-Slave-Schnittstelle und der User-logic. Die Funktionalität der PRMs wird durch die User-logic implementiert. Somit lassen sich PRMs durch das **C**reate and **I**mport **P**eripheral-Werkzeug (CIP) von Xilinx erstellen, obwohl diese keine µBlaze-Komponenten sind [39].

3.1.1.2. XPS-Socket-Bridge

Wird die PRR als Bestandteil des µBlaze-Systems instanziiert, erfolgt die Anbindung über die XPS_Socket_Bridge (siehe Abbildung 3.6). Auch die Busmacros sind Bestandteile des µBlaze-Systems (vgl. Abbildung 3.7).

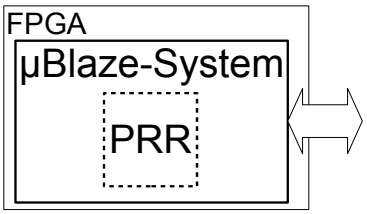


Abbildung 3.6.: Selbstrekonfigurierendes SoC mit der PRR als Bestandteil des µBlaze-Systems

Der Device-Control-Register-Bus (DCR-Bus) sowie die PLBv46-to-DCR-Bridge (DCR-Bridge) werden ebenfalls in diesem System eingesetzt. Diese Komponenten erfüllen dieselbe Funktion wie bei der Anbindung der PRR über das XPS_DCR_Socket (vgl. Abbildung 3.7).

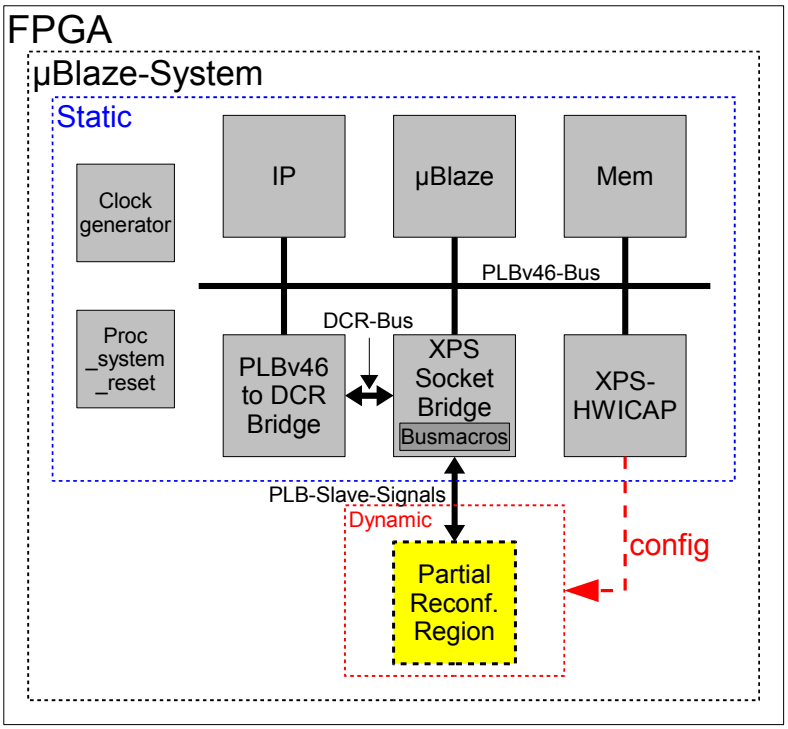


Abbildung 3.7.: Selbstrekonfigurierendes SoC mit dem XPS_Socket_Bridge als statische Schnittstelle zum PRM innerhalb des µBlaze-Systems

Das DCM-Modul wird in diesem System von der Clock-Generator-Komponente des µBlaze-Systems bereitgestellt. Diese Komponente bietet einen Rahmen um einen oder mehrere DCMs. Die durch den Base System Builder (BSB) eingestellte Taktrate für das System und die unterschiedlichen Taktsignale für die IP-Module erzeugt der Clock-Generator durch die automatische Instanziierung und Konfiguration der DCMs.

Das von Xilinx im Rahmen der partiellen Rekonfiguration bereitgestellte XPS_Socket_Bridge macht die PLB-Signale nach Außen verfügbar [41]. Durch die Instanziierung dieser Komponente mit XPS wird der PLB somit für eine weitere Einheit angepasst. Eine PRR lässt sich über die Integration in XPS an das µBlaze-System anschließen. Die PRR wird nicht direkt mit dem PLB verbunden. Die Signale des XPS_Socket_Bridge werden mit denen der PRR in der MHS-Datei des µBlaze-Systems manuell beschaltet.

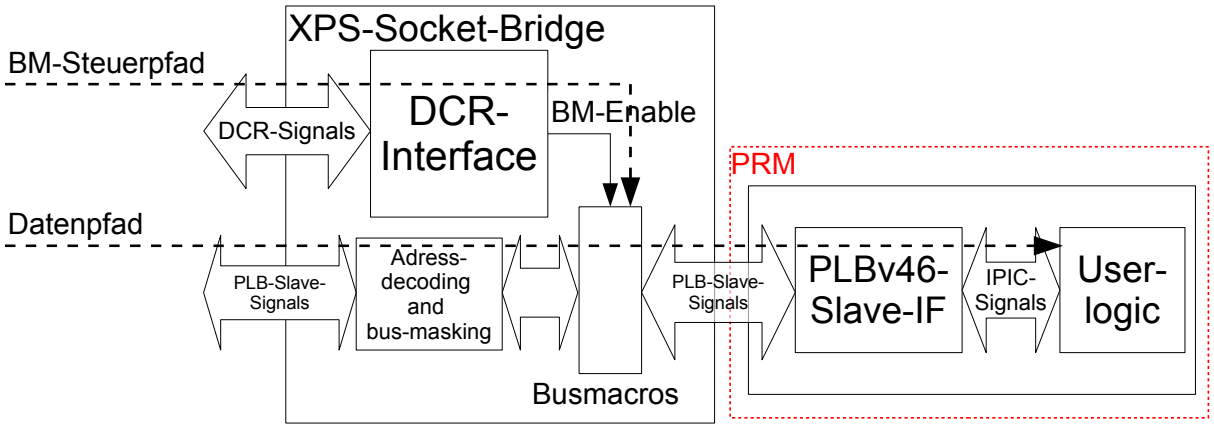


Abbildung 3.8.: XPS_Socket_Bridge mit integrierten Busmacros

Die Busmacros werden vom XPS-Socket-Bridge instanziiert und das Busmacro-Enable-Signal wird direkt an die Busmacros geführt (vgl. Abbildung 3.8). Das Setzen des Enable-Signals funktioniert auf dieselbe Weise, wie beim XPS_DCR_Socket und die PRR hat auch hier die Struktur einer gewöhnlichen PLB-Slave-IP.

3.1.1.3. PRR-IPIC-Socket

Das XPS_DCR_Socket und das XPS_Socket_Bridge stellen die gesamten PLB-Slave-Signale zur Verfügung. Die PRMs sind dafür verantwortlich, die geeignete PLB-Schnittstelle dafür zu bieten und die vereinfachten IPIC-Signale an die User-logic weiterzuleiten [21]. Die verschiedenen PRMs unterscheiden sich demnach in der implementierten User-logic. Bei der Konfiguration eines PRMs enthält der Bitstream sowohl die PLB-Schnittstelle, als auch die User-logic. Die Konfiguration der PLB-Schnittstelle bleibt bei jeder PRM unverändert.

Im Rahmen dieser Arbeit wurde ein neues Konzept einer Anbindung von PRRs an das statische System erarbeitet, das eine Reduzierung der Konfigurationsdaten der PRMs bewirkt. Beim PRR_IPIC_Socket wurde die PLB-Slave-Schnittstelle aus der PRR in die statische Schnittstelle verlagert (vgl. Abbildung 3.9).

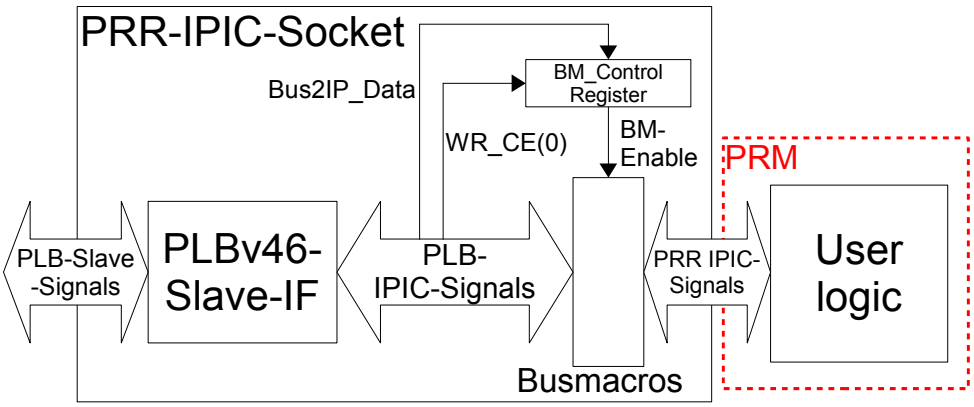


Abbildung 3.9.: PRR_IPIC_Socket mit integriertem PLB-Slave-IF und Busmacros. Die IPIC-Signale werden aus dem statischen an die User-logic im dynamischen Teil übergeben.

Das PRR_IPIC_Socket besitzt eine PLB-Slave-Schnittstelle und leitet die vereinfachten IPIC-Signale an die User-logic der PRR weiter. Die Busmacros sind Bestandteil des PRR-IPIC-Sockets. Über das BM_Control Register wird das Busmacro-Enable-Signal gesetzt. Dieses Register wird mit der ersten Registeradresse der User-logic adressiert. Dazu werden die Datenleitung und das WR_CE(0)-Signal zum Beschreiben dieses Registers ausgewertet. Das WR_CE(0)-Signal wird nicht an die PRR weitergeleitet. SW-seitig wird demnach eine Schreib-Operation auf die erste Registeradresse vollzogen. Alle weiteren IPIC-Signale werden nach Außen geführt. Folglich besitzt jede PRM eine IPIC-Schnittstelle. Die Konfigurationsdaten der PRM sind gegenüber der Busanbindung über das XPS_DCR_Socket und XPS_Socket_Bridge geringer, da die PLB-Slave-Schnittstelle in den statischen Teil ausgelagert wurde. Damit verkürzt sich die Zeit zur Rekonfiguration der PRM.

Der Einsatz der PRR als Bestandteil des µBlaze-Systems oder als Instanz in derselben Entwurfshierarchie hat keinen Einfluss auf das PRR-IPIC-Socket. In beiden Entwurfsvarianten lässt sich diese Komponente unverändert einsetzten.

3.1.2. FSM-Rekonfigurierbares HW-System

Ein als FPGA-Implementierung realisiertes HW-System ist vollständig als digitale Schaltung aufgebaut. SW findet in diesem System keinen Einsatz und somit auch keine Prozessoreinheit zur Ausführung dieser. Von der Flexibilität, die eine SW-Implementierung des Rekonfigurationskontrollers bietet, lässt sich hier nicht profitieren. Die Rekonfiguration einer selbstrekonfigurierenden SoC-Architektur erfolgt hier HW-gesteuert und setzt sich aus folgenden Komponenten zusammen(siehe Abbildung 3.10):

- **ICAP:** Virtex 5 Bibliothekselement, das eine Schnittstelle zur internen Rekonfiguration des FPGAs bereitstellt. Die Konfigurationsdaten werden von ICAP direkt über die Speicherschnittstelle gelesen.
- **Mem:** Über einen Speicher werden die Konfigurationsdaten an die ICAP-Schnittstelle gesendet. Es lässt sich entweder FPGA-interner oder externer Speicher verwenden. Dieser Speicher wird über eine Speicherschnittstelle an das System gekoppelt. Die Speicherschnittstelle wird als HW-Modell realisiert.

- **Busmacros:** Sämtliche Signale zwischen statischem und dynamischen Teil werden über Busmacros geführt, um das Routing dieser Signale für alle Module gleich zu halten.
- **Konfigurations-Controller:** Als FSM-Modell aufgebaut erzeugt dieser Controller die Steuersignale zur Rekonfiguration, das Adresssignal der Speicherschnittstelle sowie das Enable-Signal für die Busmacros.
- **Funktionalität:** Als digitale Schaltung aufgebaut. Es ist in statischen und dynamischen Teil (PRR) aufgeteilt. Das Ereignis zur Initiierung der Rekonfiguration wird vom statischen Teil der Anwendung ausgelöst. Die Kommunikation zwischen statischem und dynamischen Teil erfolgt über die Busmacros.

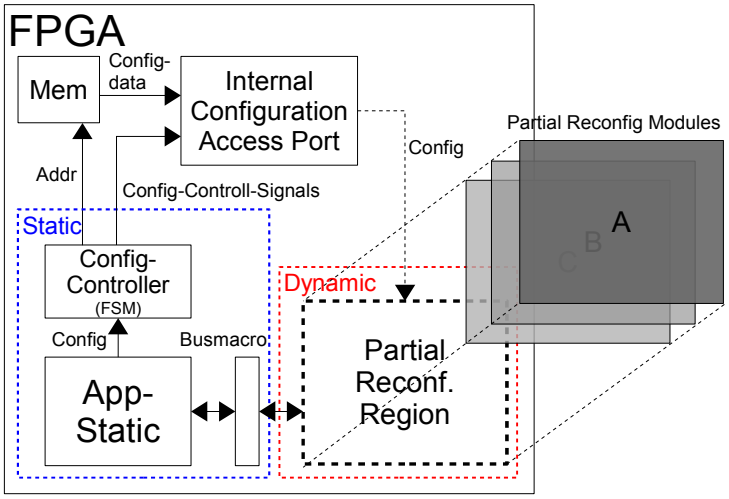


Abbildung 3.10.: FSM-rekonfigurierbares HW-System ohne integriertes µController-System

Das Hauptanwendungsfeld dieser Architektur sind tief eingebettete Systeme. Die reine HW-Implementierung erhöht die Parallelität und verkürzt somit die Ausführungsgeschwindigkeit. Der Aufwand erhöht sich ebenso bei diesem Entwurfsverfahren gegenüber dem HW-SW-Codesign.

3.2. Extern rekonfigurierbare Architekturen

Das Hauptmerkmal der extern rekonfigurierbaren Architekturen ist die Implementierung des Rekonfigurationskontrollers außerhalb des FPGAs. Die jeweiligen Architekturen unterscheiden sich in den folgenden Punkten:

- Die statische Anwendung lässt sich als SW oder HW implementieren.
- Die Initiierung der Rekonfiguration erfolgt durch einen Anwender oder durch den statischen Anwendungsteil.
- Je nach Architektur werden verschiedene Rekonfigurationsschnittstellen verwendet.

3.2.1. Anwendergestützt-rekonfigurierbares HW-System

Die Hauptanwendung für diese Architektur ist das Testen und Analysieren von HW-Modulen. Die Realisierung der Anwendung erfolgt als reine HW-Implementierung. Dabei werden PRMs durch partielle Rekonfiguration mit Impact geladen bzw. ersetzt. Das Verhalten des Systems lässt sich ohne Neukonfiguration des Gesamtsystems in jeder Kombination testen und analysieren. Diese Architektur setzt sich aus folgenden Komponenten zusammen (siehe Abbildung 3.11):

- **Xilinx Impact [20]:** Werkzeug zur Konfiguration von Xilinx FPGAs.
- **Entwicklungsrechner:** Plattform für die Ausführung von Impact sowie den Anschluss eines Konfigurationsadapters.
- **Konfigurationsadapter:** JTAG-Adapter zur Übertragung der Konfigurationsdaten (Bitstreams) an das FPGA.

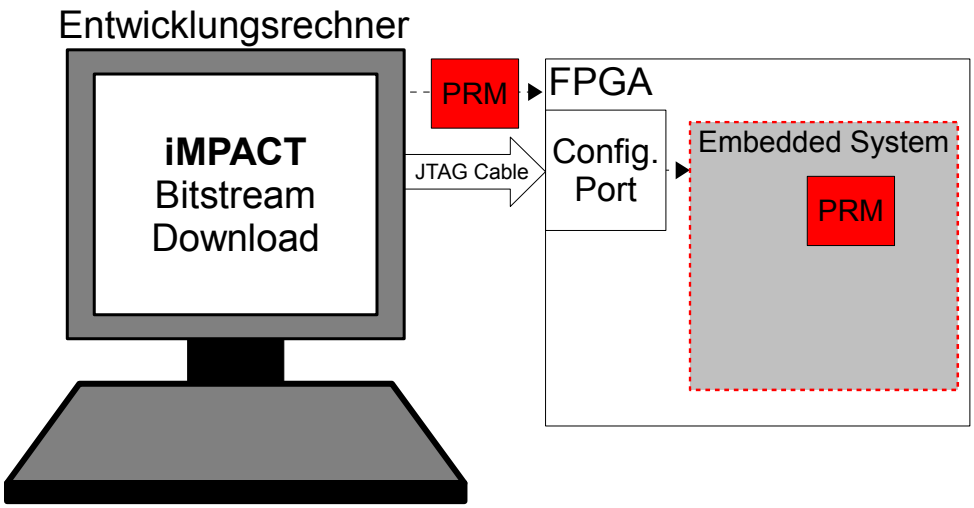


Abbildung 3.11.: Partielle Rekonfiguration eines FPGAs initiiert durch einen Anwender

Die Funktion des Rekonfigurationskontrollers wird in dieser Architektur durch Impact realisiert [20]:

- Über die Befehlszeile oder GUI-basiert werden Xilinx FPGAs konfiguriert.
- Sowohl eine initiale Gesamtrekonfiguration, als auch eine partielle Rekonfiguration mit Impact lässt sich durchführen.
- Impact erlaubt den Zugriff auf die serielle Konfigurations-, die JTAG-, als auch die SelectMap-Schnittstelle [20], [42].

Die Bitstreams (Konfigurationsdaten) der jeweiligen PRMs sind im Speicher des PCs abgelegt und werden vom Benutzer zur Rekonfiguration ausgewählt.

Diese Architektur basiert auf einer rein partiellen Rekonfiguration. Eine dynamische oder partiell und dynamische Rekonfiguration ist aus folgenden Gründen nicht sinnvoll:

- Die rein dynamische Konfiguration bedarf einer externen Plattform zur Ausführung des statischen Anwendungsteils.
- Für die dynamische Rekonfiguration ist eine geeignete Kommunikationsstruktur zwischen FPGA und externer Plattform zu erstellen, die zu einem erhöhten Entwurfsaufwand führt.
- Eine partielle und dynamische Rekonfiguration führt zu einem komplexeren Systemverhalten, da die Initiierung der Rekonfiguration durch einen Anwender zu integrieren wäre.

3.2.2. Selbstrekonfigurierendes HW-SW-System

Ein selbstrekonfigurierendes HW-SW-System führt die Rekonfiguration mittels einer zum System gehörenden SW aus. Dabei besteht das System aus zwei Plattformen (siehe Abbildung 3.12):

Prozessoreinheit: Plattform für die Ausführung von SW.

FPGA: Plattform für die Konfiguration einer HW-Schaltung.

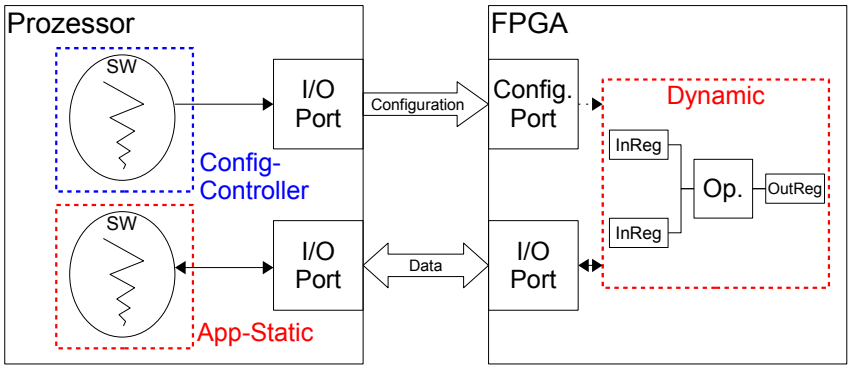


Abbildung 3.12.: Dynamische Rekonfiguration einer FPGA-Plattform initiiert durch eine externe Prozessor-Plattform

Der Rekonfigurationskontroller und der statische Teil der Anwendung werden als SW implementiert und auf der Prozessoreinheit ausgeführt. Je nach Plattform lassen sich diese Einheiten entweder als ein Ausführungsstrang realisieren oder in mehrere Threads unterteilen. Die Rekonfiguration eines Virtex 5 FPGAs lässt sich in dieser Architektur über die serielle Konfigurations-, JTAG-, SelectMap, BPI- sowie SPI-Schnittstelle durchführen [42]. SW-seitig hat dies über eine I/O-Schnittstelle der Prozessoreinheit zu erfolgen. Die Kommunikation zwischen statischem und dynamischen Teil der Anwendung erfolgt ebenso über eine I/O-Schnittstelle. Die Initiierung der Rekonfiguration erfolgt durch den statischen Teil der Anwendung. Somit ist dieses System ein SW-gesteuert-rekonfigurierbares System. Der dynamische Teil der Anwendung wird als digitale Schaltung im FPGA konfiguriert.

Diese Architektur erfordert die Anbindung eines zusätzlichen Speichers, der ausreichend Ressourcen für die Ablage der Bitstreams bietet. Der Rekonfigurationskontroller ist für das Lesen dieser Bitstreams vom Speicher und das anschließende Schreiben auf die Konfigurationsschnittstelle verantwortlich. Die Rekonfigurationszeit wird somit durch diesen Schritt ausgedehnt.

Hauptanwendungsfeld für diese Architektur ist die Beschleunigung von SW-basierten Anwendungen durch dedizierte HW-Beschleunigermodule. Zur Erfüllung der korrekten Funktion des Systems ist der Verbund aus statischem und dynamischen Anwendungsteil erforderlich. Es erfolgt eine Gesamtrekonfiguration des FPGAs, wenn ein HW-Modul für eine Anwendung geladen wird. Der statische Teil des System unterbricht den Betrieb während der Rekonfiguration nicht. Somit ist dieses System rein dynamisch rekonfigurierbar.

3.3. Gegenüberstellung der Rekonfigurationsarchitekturen

In den einzelnen Abschnitten über die jeweiligen Architekturen wurden weitere Eigenschaften und Merkmale genannt (vgl. Tabelle 3.2).

Eigenschaft / **Architektur**	Haupt-anwendung	Komponenten	Entwurfs-komplexität	Rekonfig.-technik	Ablage von Bitstreams
Benutzer-rekonfigurierbares HW-System	Testen und Analysieren von HW-Modulen	Xilinx Impact, PC, Konfigurationskabel	HW-Entwurf nach Xilinx EAPR	Partiell	Speicher des PCs
Selbstrekonf. HW-SW-System	Beschleunigung von SW durch dedigzierte HW-Module	Externer, eigenständiger Prozessor, externer Speicher	Klassischer HW-Entwurf, Klassischer SW-Entwurf	Dynamisch	Externer Speicher
FSM-Rekonfigurierbares HW-Systems	Eingebettete, zeitkritische Systeme	Externer Speicher	HW-Entwurf nach Xilinx EAPR	Partiell und dynamisch	Interner oder externer Speicher
Selbstrekonf. SoC	**Eingebettete HW-SW-Systeme, SoCs**	**Externer Speicher**	**HW-Entwurf nach Xilinx EAPR, klassischer SW-Entwurf**	**Partiell und dynamisch**	**Interner oder externer Speicher**

Tabelle 3.2.: Übersicht zu den Rekonfigurationsarchitekturen und deren Eigenschaften

Für jede Architektur wurden folgende Eigenschaften spezifiziert:

Hauptanwendung: Das geeignete Einsatzfeld der jeweiligen Architektur.

Komponenten: In jeder Architektur wird ein FPGA eingesetzt und daher in der Tabelle nicht aufgelistet. Alle weiteren Komponenten zum Aufbau der jeweiligen Architektur wurden aufgezählt.

Entwurfskomplexität: Mit der Entwurfskomplexität wird festgelegt, welche HW- und SW-Abläufe zum Entwurf der jeweiligen Architekturen durchzuführen sind.

Rekonfigurationstechnik: Mit der Rekonfigurationstechnik wurde eine sinnvolle Implementierung jeder Architektur spezifiziert.

Speicher für Konfigurationsdaten: Die Bereitstellung der Bitstreams bedarf bei jeder Architektur eines Speichers. Die Lokalität sowie der Zugriff auf diesen Speicher wurden für jede Architektur genannt.

Das zentrale Thema dieser Arbeit ist die selbstrekonfigurierbare SoC-Architektur. Für die drei Varianten dieser Architektur ergeben sich zwei Klassifikationskriterien:

- **Die Entwurfshierarchie der PRR**
  - Wird die PRR als Bestandteil des µBlaze-Systems instanziiert, lässt sich das gesamte System mit dem XPS-Werkzeug zusammenstellen und synthetisieren. Allerdings ist somit das gesamte System von Veränderungen eines Teils des Systems während der Entwurfsphase betroffen. Dieses beeinflusst sowohl die korrekte Funktionsweise einiger Komponenten, als auch die Implementierungszeit. Der PLB ist für jede Veränderung an den Komponenten neu anzupassen. Die Implementierungszeit erhöht sich, wenn aufgrund einer Veränderung am System eine erneute Synthese aller Komponenten durchzuführen ist.
  - Wird die PRR in derselben Hierarchieebene wie das µBlaze-System instanziiert, werden korrekt funktionierende Komponenten nicht mehr verändert. Dies betrifft sowohl das µBlaze-System, die PRMs, als auch weitere Komponenten. Außerdem lässt sich ein eigenständiges und fehlerfreies µBlaze-System als Teil eines größeren Systems verwenden. Die Entwurfskomplexität erhöht sich jedoch durch die Erstellung und Verwaltung einer Top-Entity. XPS wird in diesem Fall für die Zusammenstellung des µBlaze-Systems verwendet. Für die Synthese der Top-Entity samt der Komponenten ist der Einsatz eines weiteren Synthesewerkzeuges (z.B. Xilinx ISE) erforderlich.
- **Die Struktur der statischen Schnittstelle zur PRR**
  - Eine direkte Anbindung der PRR an den PLB bedarf einer komplexen und zeitaufwendigen Manipulation der Entwurfsdateien.
  - Der Einsatz des XPS_DCR_Sockets und des XPS_Socket_Bridge macht eine Manipulation der Entwurfsdateien überflüssig. Diese beiden Komponenten leiten die gesamten PLB-Slave-Signale an eine PRM weiter. Die PRMs müssen demnach eine PLB-Slave-Schnittstelle integrieren. Diese Schnittstelle ist dem PLB durch die Generics angepasst.
  - Der PRR_IPIC_Socket besitzt eine PLB-Slave-Schnittstelle und leitet die IPIC-Signale an die PRR weiter. Die PRMs bestehen hier aus der User-logic. Ihre Konfigurationsdaten sind im Vergleich zu denen der XPS_DCR_Socket und der XPS_Socket_Bridge geringer, was eine kürzere Rekonfigurationszeit zur Folge hat. Die IPIC-Schnittstelle einer IP-User-logic wird von XPS automatisch angepasst. In diesem Fall bedarf es hier jedoch der manuellen Anpassung der Schnittstelle der User-logic, also der PRMs für jede Anwendung.

Das Konzept des PRR-IPIC-Sockets wurde im Rahmen dieser Arbeit behandelt. Die Implementierung und das Testen dieser Komponente ist in zukünftigen Projekten durchzuführen.

4. Selbstrekonfigurierendes µBlaze-System mit Testmodulen

In diesem Kapitel wird der Aufbau des entwickelten, SW-gesteuerten, selbstrekonfigurierenden µBlaze-Systems mit externer PRR dokumentiert (siehe Abbildung 4.1). Das µBlaze-System ist nach der selbstrekonfigurierenden SoC-Architektur aus Kapitel 3.1.1.1 aufgebaut. Die Anbindung der PRR findet bei diesem System über das XPS_DCR_Socket statt. PRR und µBlaze-System sind Instanzen derselben Hierarchieebene und werden von einer Top-Entity instanziiert. Für die PRR wurden zwei Test-PRMs realisiert. Die erste PRM führt Additions-, die zweite Multiplikationsoperationen aus. Mit einer eingebetteten C-Testapplikation, die je nach Anforderung eine der PRMs aktiviert, wird die korrekte Funktionalität der partiellen Rekonfiguration verifiziert.

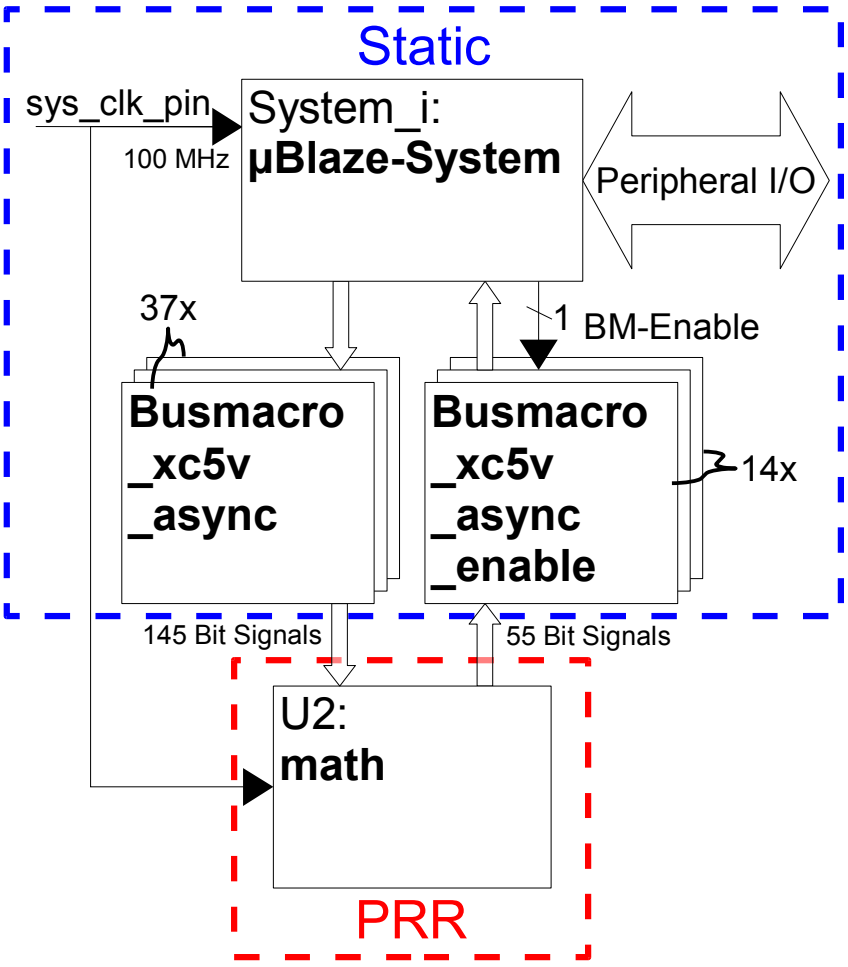


Abbildung 4.1.: Aufbau des selbstrekonfigurierenden µBlaze-Systems nach Abb. 3.3

4.1. µController-System

Die Zusammensetzung des µController-Systems bietet sämtliche Komponenten für die frühe Verifikation der partiellen Rekonfiguration durch eine eingebettete Test-SW (vgl. Abbildung 4.2).

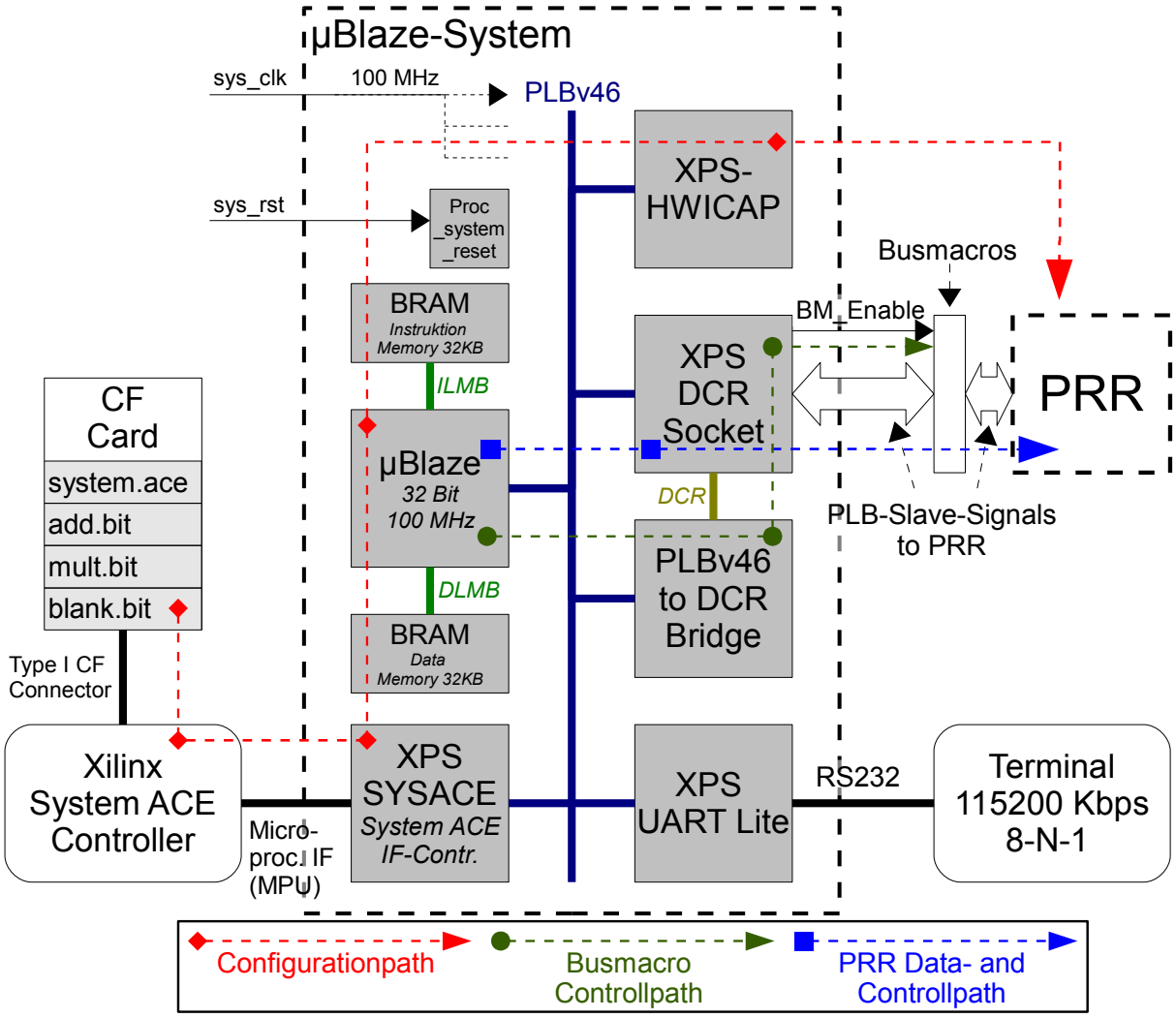


Abbildung 4.2.: Blockschaltbild des µBlaze-Systems mit allen Komponenten

4.1.1. µBlaze-Prozessor

Das Herzstück des µController-Systems bildet der µBlaze-Prozessor. Diese Plattform bildet die Grundlage zur Ausführung von SW. Der µBlaze-Prozessor wurde für dieses System folgendermaßen konfiguriert:

- **System-Bus-Takt:** 100 MHz
- **Debug IF:** Das On-chip H/W debug module dient zum Debuggen der SW über die JTAG-Schnittstelle. Die Konfiguration dieses Moduls erlaubt das Setzen eines HW-basierten Breakpoints.

- **Lokaler Speicher (BRAM):** Für Daten und Instruktionen wurde 64 KB BRAM-basierter Speicher konfiguriert. Dieser Speicher dient der Ablage der eingebetteten Testapplikation.
- **Schnittstellen:** LMB, PLB
  - **LMB:** Der Local Memory Bus ist die Schnittstelle des µBlaze-Prozessors zum lokalen Daten- und Instruktionsspeicher.
  - **PLB:** Die PLB-Schnittstelle bietet dem µBlaze-Prozessor die Initiierung von PLB-Transaktionen als Master.

4.1.2. Komponenten für die Partielle Rekonfiguration

XPS HWICAP
Das Internal Configuration Access Port (ICAP) ist ein Bestandteil des Virtex 5 FPGAs, das eine Schnittstelle zur internen Rekonfiguration bietet. Das XPS-HWICAP IP-Modul instanziiert diese ICAP-Ressource des FPGAs und stellt folgende Eigenschaften bereit:

- Diese Komponente bietet einen Zugang über den PLB, sodass sich eine interne Rekonfiguration durch µBlaze-SW durchführen lässt.
- Über Generics lässt sich die PLB-Schnittstelle anpassen.
- Es lassen sich partielle Bitreams konfigurieren bzw. lesen.
- Es operiert mit der Taktrate des PLBs (125 MHz)
- Es stellt je 1024x32-Bit breite Fifos für zu Schreibende sowie Lesende Bitstreams zur Verfügung.

Andere Eigenschaften des XPS HWICAP lassen sich nicht mit Generics konfigurieren. Dieses Modul ist Teil des Konfigurationspfades.

PLBV46 to DCR Bridge
Dieses Modul erlaubt einen SW-seitigen Zugriff auf Register des XPS_DCR_Sockets zur Steuerung des Busmacro-Enable-Signals (vgl. Kapitel 3.1.1.1).

- **Schnittstelle: D**evice **C**ontrol **R**egister, **P**rocessor **L**ocal **B**us

Diese Komponente wird über den DCR-Bus mit dem XPS_DCR_Socket verbunden. Über die PLB-Schnittstelle wird SW-seitig das Busmacro Enable-Signal gesteuert.

XPS DCR Socket
Die PLB-Slave-Signale für den Austausch von Daten zwischen PRR und statischem System und das Busmacro Enable-Signal werden über das XPS_DCR_Socket bereitgestellt (vgl. Kap. 3.1.1.1).

- **Schnittstelle: D**evice **C**ontrol **R**egister, **P**rocessor **L**ocal **B**us

Über den DCR-Bus wird das XPS_DCR_Socket mit der DCR-Bridge verbunden. Die Steuerung des BM Enable-Signals wird vom µBlaze-Prozessor über die PLB-Schnittstelle der DCR-Bridge initiiert. Die DCR-Bridge steuert das BM Enable-Signal über die DCR-Schnittstelle zum XPS_DCR_Socket. Der Datenaustausch zwischen µBlaze-System und PRR erfolgt über die PLB-Schnittstelle des XPS_DCR_Sockets. Dieser leitet die PLB-Slave-Signale zu den Busmacros weiter.

XPS System Advanced Configuration Environment
Dieser Schnittstellen-Controller dient zur Anbindung von Compact-Flash Speicherkarten, die die Bitstreams (Konfigurationsdaten) des Systems sowie der PRMs bereitstellen. Es lassen sich bis zu acht initiale Bitstreams von diesem Kontroller verwalten. Diese Bitstreams werden mit Impact erstellt. Während der Generierung dieser Bitstreams (*.ace Endung) wird ihre Konfigurationsadresse festgelegt, die aus drei Bits gebildet wird. Der Kontroller stellt hierfür drei Signalleitungen bereit. Die Bitstreams der PRMs (*.bit Endung) werden ohne eine Adresse neben den initialen Bitstreams im CF Speicher abgelegt. Die PLB-Slave-Schnittstelle dieses Kontrollers erlaubt das SW-seitige Lesen bzw. Beschreiben der CF-Karte. Somit lassen sich die partiellen Bitstreams von der Testapplikation lesen und anschließend an das XPS-HWICAP schicken.

4.1.3. Peripherie und I/O

XPS UART Lite
Verbindung zu einem Entwicklungs-PC zur seriellen Ein- und Ausgabe. Folgende Konfiguration wurde vorgenommen:

- **Baudrate:** 115200
- **Datenbreite:** 8 Bit
- **Parität:** Keine
- **Schnittstelle:** PLB

Die Testapplikation führt die Interaktion mit einem Benutzer über diese Schnittstelle durch, um Befehle zur Rekonfiguration anzunehmen und den Status der PR über ein Terminal-Programm anzuzeigen.

4.2. Test-PRMs

Mit dem Xilinx **C**reate und **I**mport **P**eripheral Wizart wurden für dieses System zwei PRMs erstellt. Eine der PRMs implementiert eine Additions-, die andere eine Multiplikationsoperation (vgl. Abbildung 4.3).

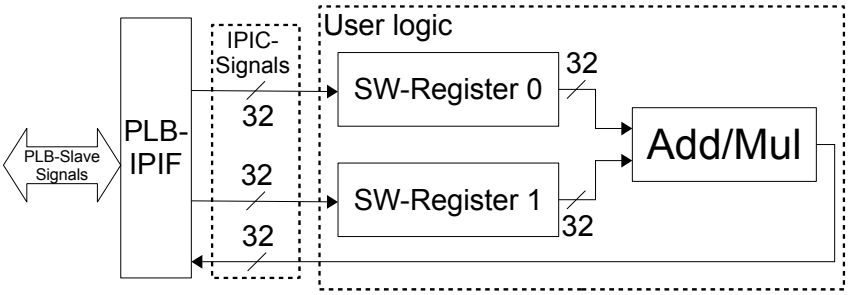


Abbildung 4.3.: Test-PRMs mit 2 SW-Registern für die Operandenübergabe an Operationen. Ergebnisse werden kombinatorisch gelesen.

Die PRM bestehen aus folgenden Komponenten:

- **PLB-Slave-Schnittstelle**
  Datentransfer mit PLB-Slave-Signalen von der statischen Schnittstelle (vgl. Abbildung 3.5, XPS_DCR_Socket).
- **User-logic**
  Die User-logic implementiert die jeweilige mathematische Operation.
  - Beide Varianten besitzen zwei 32-Bit breite SW-Register für die Operanden
  - math_v1_00_a führt eine Addition mit beiden Operanden durch und stellt das Ergebnis als 32-Bit breites Signal am Ausgang bereit.
  - math_v1_01_a implementiert eine Multiplikation mit den 16-MSB der SW-Register für die Operanden und stellt das Ergebnis als 32-Bit breites Signal am Ausgang bereit.

Für den SW-seitigen Zugriff auf diese Module stehen zwei Adressen zur Verfügung. Bei einem Schreibvorgang wird auf die Adresse der jeweiligen Register zugegriffen und die Operanden geschrieben. Eine Leseoperation auf einer dieser Adressen liefert das Ergebnis der Operation über einen 2zu1 Multiplexer an das IPIC zurück (vgl. Quelltext 4.1).

```
case slv_reg_read_sel is
        when "10" => slv_ip2bus_data <= add;
        when "01" => slv_ip2bus_data <= add;
        when others => slv_ip2bus_data <= (others => '0');
end case;
```

Quelltext 4.1: Das Ergebnis der Operation wird beim Lesen beider Adressen an den Ausgang des Multiplexers gelegt

In Quelltext 4.2 ist die Bereichseingrenzung der PRR für die PRMs in der UCF-Datei dargestellt.

```
AREA_GROUP "pblock_U2" RANGE=SLICE_X10Y70:SLICE_X19Y109;
AREA_GROUP "pblock_U2" RANGE=DSP48_X0Y28:DSP48_X0Y43;
AREA_GROUP "pblock_U2" MODE=RECONFIG;
INST "U2" AREA_GROUP = "pblock_U2";
```

Quelltext 4.2: Bestimmung der Bereichsgrenzen der PRR in der UCF-Datei

Digital Signal Processors (DSP48E) sind Ressourcen von Virtex 5 FPGAs, die für die Ausführung der Multiplikation eingesetzt werden [43]. Für die Addition und die restliche digitale Logik wird Slice-Logic verwendet [43]. In Abbildung 4.4 ist der von der PRR aufgespannte Bereich in einem Virtex XC5VLX50T visualisiert. Dieser beinhaltet Slice-Logic von der Position X10Y70 bis X19Y109 und DSPs von X0Y28 bis X0Y43.

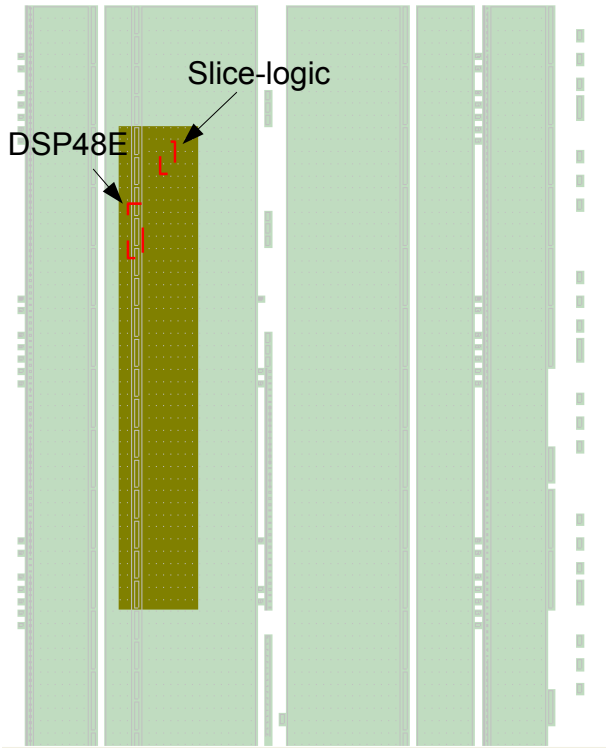


Abbildung 4.4.: Bereich der math-PRR im Virtex XC5VLX50T

Die PRR umspannt einen Bereich des FPGAs, der genügend Ressourcen für die jeweilige Implementierung der PRMs bieten muss (vgl. Tabelle 4.1).

Ress.	PRR	Addition	Multi-plikation
Slices	400	72	55
Look Up Tables	1600	103	63
DFFs	1600	206	122
DSP48E	16	0	1

Tabelle 4.1.: Ressourcenübersicht der PRR und die Verwendung durch die Additions- und Multiplikations-PRMs

Der Bereich der PRR wurde in dieser Größe konfiguriert, um den Ressourcenbedarf der PRM-Implementierungen in jedem Fall zu genügen. Damit waren mehrere Durchläufe der Implementierung nicht erforderlich.

4.3. Busmacros

Für je vier Bit Signale zwischen der PRR und dem statischen System wird ein Busmacro instanziiert. Für Signale zur PRR werden Busmacros vom Typ “busmacro_xc5v_async“ und für Signale von der PRR “busmacro_xc5v_async_enable“ eingesetzt (siehe Abbildung 4.1). Die Busmacros werden zur Fixierung der Schnittstelle zwischen PRR und dem statischen Teil des Systems verwendet. Der Signalverlauf zwischen den PRMs und dem statischen System bleibt für alle Rekonfigurationen fest zugeordnet. Ein Busmacro ordnet also jeweils vier Bit Eingangssignale zu vier Bit Ausgangssignalen zu. Während der Implementierung wird somit verhindert, dass die Signale der einzelnen PRMs zum statischem System durch das Routing unterschiedlich zugeordnet werden. Für alle PLB-Slave-Signale werden hier Busmacros instanziiert (vgl. Quelltext 4.3).

```
ABUS_0_BM_GENERATE: for i IN 0 TO 7 generate
        ABUS_BM : busmacro_xc5v_async
        port map (
                input0  => xps_dcr_socket_0_RPLB_ABus_pin(4*i+0),
                input1  => xps_dcr_socket_0_RPLB_ABus_pin(4*i+1),
                input2  => xps_dcr_socket_0_RPLB_ABus_pin(4*i+2),
                input3  => xps_dcr_socket_0_RPLB_ABus_pin(4*i+3),
                output0  => xps_dcr_socket_0_RPLB_ABus_pr(4*i+0),
                output1  => xps_dcr_socket_0_RPLB_ABus_pr(4*i+1),
                output2  => xps_dcr_socket_0_RPLB_ABus_pr(4*i+2),
                output3  => xps_dcr_socket_0_RPLB_ABus_pr(4*i+3)
        );
```

Quelltext 4.3: Instanziierung von 8 Busmacros vom Typ “busmacro_xc5v_async“ für das 32-Bit breite PLB-Adress-Signal zur PRR

Busmacros vom Typ “busmacro_xc5v_async“ werden für Signale instanziiert, die Daten vom statischen System zur PRR senden. Während der Rekonfiguration wird die PRR das statische System somit nicht beeinflussen. Signale von der PRR zum statischen System können während der Rekonfiguration undefinierte Werte vorweisen und somit zu einer Störung des statischen Systems führen. Busmacros vom Typ “busmacro_xc5v_async_enable“ besitzen zu den 4 Bit Ein- und Ausgangssignalen noch ein 4 Bit Enable-Signal, mit dem diese Busmacros eine Multiplexerschaltung realisieren. Ist dieses Enable-Signal auf eine logische “0“ gesetzt, wird am Ausgang des Busmacros konstant “0“ ausgegeben. Andernfalls wird der Eingangswert am Ausgang gesetzt.

```
Control2_0_BM : busmacro_xc5v_async_enable
        port map (
                input0  => xps_dcr_socket_0_RSl_addrAck_pr,
                input1  => xps_dcr_socket_0_RSl_SSize_pr(0),
                input2  => xps_dcr_socket_0_RSl_SSize_pr(1),
                input3  => xps_dcr_socket_0_RSl_wait_pr,
                enable0 => busmacro_0_enable,
                enable1 => busmacro_0_enable,
                enable2 => busmacro_0_enable,
```

```
            enable3 => busmacro_0_enable,
            output0 => xps_dcr_socket_0_RSI_addrAck_pin,
            output1 => xps_dcr_socket_0_RSI_SSize_pin(0),
            output2 => xps_dcr_socket_0_RSI_SSize_pin(1),
            output3 => xps_dcr_socket_0_RSI_wait_pin
    );
```

Quelltext 4.4: Instanziierung eines Busmacros für ein 4 Bit Signal von der PRR zum statischen System. Mit einem 4 Bit Enable-Signal wird das Ausgangssignal gestuert.

Quelltext 4.4 zeigt die Instanziierung eines Busmacros vom Typ “busmacro_xc5v_async_enable“ für 4 Bit Signale von der PRR zum statischen System. Jedes einzelne Signal wird über ein dediziertes Enable-Signal gesteuert. Das Enable-Signal wird von der statischen Schnittstelle (XPS_DCR_Socket) geliefert. Vor jeder Rekonfiguration wird dieses Signal auf eine logische “0“ gesetzt und nach der Rekonfiguration wieder auf “1“. Nach diesem Vorgang ist die jeweilige PRM aktiviert.

Insgesamt wurden 37 “busmacro_xc5v_async“-Busmacros für 145 Bit Signale vom µBlaze-System zur PRR und 14 “busmacro_xc5v_async_enable“-Busmacros für 55 Bit Signale von der PRR zum µBlaze-System instanziiert (siehe Abbildung 4.1). Diese Signale führen sämtliche Steuer- und Dateninformationen des PLBs, die für die Kommunikation zwischen µBlaze-Prozessor und der PRR verwendet werden. Ändert XPS die PLB-Signale, bleibt die PRM-Schnittstelle unverändert. Interne Signale, die das µBlaze-System mit den Busmacros und die Busmacros mit der PRR verbinden, werden mit dem Attribut “SAVE“ (S) versehen (siehe Quelltext 4.5). Dieses Attribut verhindert den Wegfall interner Signale während der Implementierung durch automatische Optimierung durch das Map-Tool [38]. Die Verbindung zwischen µBlaze-System und PRR bleibt somit statisch.

```
attribute S     : string;

attribute S     of xps_dcr_socket_0_RSI_MIRQ_pin   : signal is "TRUE";
```

Quelltext 4.5: Markierung eines internen Signals mit dem Attribut “SAVE“

Sämtliche Busmacros sind innerhalb der PRR zu platzieren, damit die Signalwege der PRMs die PRR-Grenze nicht überschreiten [34]. Die Position der hier verwendeten Busmacros wird in der UCF-Datei spezifiziert (vgl. Quelltext 4.6).

```
INST "TAttribute_0_BM_GENERATE[0].TAttribute_BM" LOC = SLICE_X13Y80;
INST "TAttribute_0_BM_GENERATE[1].TAttribute_BM" LOC = SLICE_X13Y79;
INST "TAttribute_0_BM_GENERATE[2].TAttribute_BM" LOC = SLICE_X13Y78;
INST "TAttribute_0_BM_GENERATE[3].TAttribute_BM" LOC = SLICE_X13Y77;
```

Quelltext 4.6: Positionsbestimmung von 4 Busmacros im FPGA. Diese besetzen jeweils einen Slice des FPGAs innerhalb des Bereiches der PRR.

4.4. Eingebettete Test-Applikation

Zur Prüfung der partiellen Rekonfiguration wurde eine SW-basierte C-Anwendung entwickelt, die den Austausch der PRMs je nach Benutzereingabe vornimmt (siehe Anhang A). Grundlage für die Integration der HW und der Ausführung der SW bildet die Xilinx ML-505 Plattform. Folgende Elemente der ML505-Plattform wurden für diesen Test verwendet (siehe Abbildung 4.5):

- **Virtex XC5VLX50T FPGA**
  Das µBlaze-System und die eingebettete Testapplikation werden in diesem FPGA integriert und ausgeführt (vgl. Abbildung 4.2).
- **Xilinx System ACE Controller**
  Dieser Kontroller bietet eine Schnittstelle für den Zugriff auf die CF-Speicherkarte. Zusätzlich lässt sich das FPGA beim Einschalten der ML505-Plattform über diesen Kontroller mit einem initialen Bitstream laden.
- **Compact-Flash Speicher**
  Der CF-Speicher dient der Ablage des initialen (*.ace) sowie der partiellen Bitstreams.
- **RS-232 Schnittstelle**
  Über die RS-232 Schnittstelle wird eine Verbindung zu einem PC realisiert. Ein- und Ausgaben der Testapplikation erfolgen über ein am PC ausgeführtes Terminalprogramm.
- **Configuration Switch**
  Über diese Schalter lassen sich die Konfigurationsart sowie die Adresse der zu konfigurierenden, initialen Bitstreams einstellen.

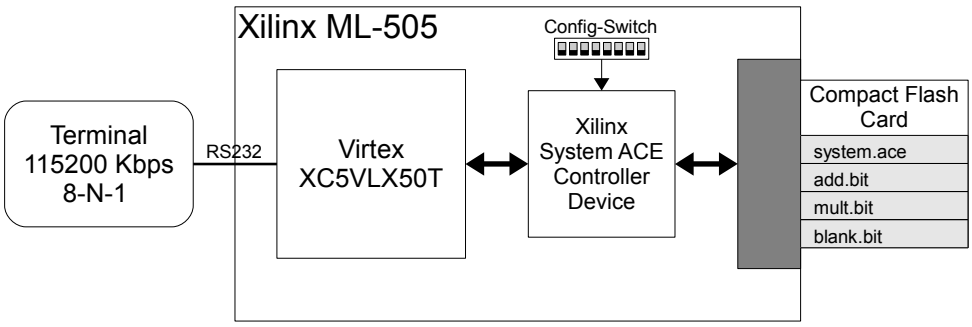


Abbildung 4.5.: Verwendete Elemente der Xilinx ML-505-Plattform [24] für die Ausführung der Testapplikation

Die 51 KB große Testapplikation ist im lokalen Speicher (64 KB BRAM) des µBlaze-Systems abgelegt. Der Zugriff auf die CF-Speicherkarte erfolgt durch die XPS SYSACE Komponente. Die partiellen Bitstreams add.bit, mult.bit und blank.bit wurden auf der CF-Karte abgelegt. Das Bitstream von blank.bit enthält keine Implementierung. Die Testapplikation teilt sich in drei Teilaufgaben auf (siehe Abbildung 4.6):

- **Initialisierung:** Beim Start des Programms werden zunächst die XPS-SYSACE und XPS-HWICAP IPs des Systems initialisiert.
- **Konfiguration:** Je nach Benutzereingabe wird der Partielle Bitstream des Addierers (add.bit), des Multiplizierers (mult.bit) oder das leere Bitstream (blank.bit) konfiguriert.

- **Ausführung der mathematischen Operation:** Die Anwendung liest zwei Operanden über das Terminal ein und gibt das Ergebnis am Terminal aus.

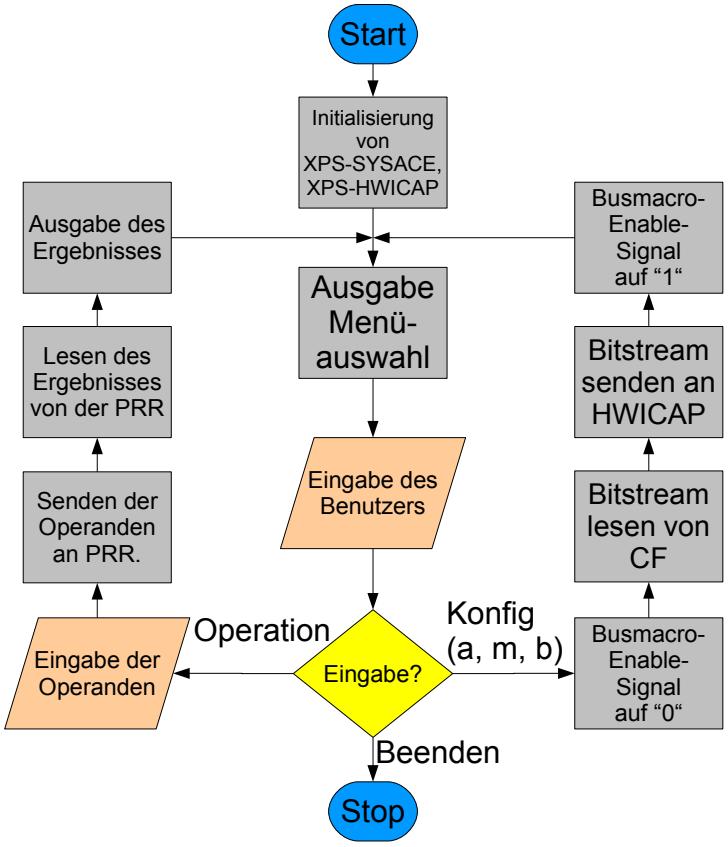


Abbildung 4.6.: Flussdiagramm der eingebetteten Testapplikation

4.4.1. Initialisierung der XPS-SYSACE und XPS-HWICAP IPs

Der Vorgang der Initialisierung durchläuft folgende Einzelschritte:

1. Die initial konfigurierte PRR wird durch das Setzen des Enable-Signals für die Busmacros auf “1“ aktiviert. Hierzu wird eine IO-Operation auf die DCR-Adresse der statischen Schnittstelle (XPS_DCR_Socket) ausgeführt (siehe Quelltext 4.7).

```
XIo_Out32(XPAR_XPS_DCR_SOCKET_0_DCR_BASEADDR,0x00000001);
```

Quelltext 4.7: Aktivieren der PRR durch das Enable-Signal. Die Daten werden über die DCR-Bridge an das XPS_DCR_Socket gesendet.

2. Als nächstes werden die beiden Komponenten XPS-HWICAP und XPS-SYSACE initialisiert (siehe Quelltext 4.8).

```
#include "xhwicap_i.h"
#include "xhwicap.h"
#include "xhwicap_parse.h"
#include "xsysace_l.h"
#include "xsysace.h"
```

```
...
Status = XSysAce_Initialize(&SysAce, XPAR_SYSACE_0_DEVICE_ID);
...
Status = XHwIcap_CfgInitialize(&HwIcap, ConfigPtr,
                                ConfigPtr->BaseAddress);
```

Quelltext 4.8: Die Header für XPS-SYSACE und XPS-HWICAP werden eingebunden. Die Funktionen zur Initialisierung dieser IPs werden aufgerufen.

3. Über die serielle Schnittstelle wird das Auswahlmenü am Terminal ausgegeben.
4. Es wird auf eine Eingabe des Benutzers über die serielle Schnittstelle gewartet.

4.4.2. Konfiguration der PRM

Wird eine Aufforderung zur Konfiguration erhalten, startet die Konfigurationsprozedur:

1. Das Busmacro-Enable-Signal wird auf “0“ gesetzt. Die Signale von der PRR zum statischen System sind somit alle auf “0“gesetzt.
2. Das ausgewählte Bitstream wird über das XPS-SYSACE Modul von der CF-Karte gelesen und zunächst im lokalen Datenspeicher zwischengespeichert (Quelltext 4.9).

```
numCharsRead = sysace_fread(systemACE_Buffer, 1,
                            XSA_CF_SECTOR_SIZE, stream);
```

Quelltext 4.9: Lesen des Bitstreams von der CF-Karte

3. Der Header des gepufferten Bitstreams wird ermittelt (Quelltext 4.10).

```
bit_header = XHwIcap_ReadHeader(systemACE_Buffer,0);
```

Quelltext 4.10: Lesen des Bitstreamheaders

4. Die Bitstream-Daten werden ohne den Header in 32-Bit Blöcken an HWICAP gesendet (Quelltext 4.11).

```
for (i=0; i<bit_header.BitstreamLength; i+=4) {

  /* Convert 4 chars into an integer */
  data3 = systemACE_Buffer[ace_buf_count++];
  data2 = systemACE_Buffer[ace_buf_count++];
  data1 = systemACE_Buffer[ace_buf_count++];
  data0 = systemACE_Buffer[ace_buf_count++];
  word = ((data3 << 24) | (data2 << 16) | (data1 << 8) | (data0));
  Status = XHwIcap_DeviceWrite(&HwIcap, &word, 1);
  ...
```

Quelltext 4.11: Senden der Bitstreamdaten in 32-Bit Blöcken an HWICAP

a) Vor dem Senden wird geprüft, ob HWICAP bereit für die Aufnahme der Daten ist (Quelltext 4.12).

```
if (XHwIcap_mIsDeviceBusy(InstancePtr) == TRUE) {
  return XST_FAILURE;
}
```

Quelltext 4.12: Prüfen der Bereitschaft von HWICAP

b) Bietet das Schreib-Fifo des HWICAP ausreichend Platz zur Aufnahme der Daten, wird das Bitstream an das Fifo des Moduls gesendet (Quelltext 4.13).

```
while ((WrFifoVacancy != 0) &&
            (InstancePtr->RemainingWords > 0)) {
  XHwIcap_mFifoWrite(InstancePtr,
                     *InstancePtr->SendBufferPtr);
  ...
```

Quelltext 4.13: Prüfen des HWICAP Schreib-Fifo und Senden der Daten

c) Nach dem Senden eines Blocks wird ein Befehl zur Ausführung der Rekonfiguration an ICAP gesendet (Quelltext 4.14).

```
XHwIcap_mStartConfig(InstancePtr);
```

Quelltext 4.14: Befehl zum Start der Rekonfiguration

5. Das Busmacro-Enable-Signal wird wieder auf “1“ gesetzt. Die konfigurierte PRR wird somit aktiviert.

4.4.3. Ausführung der Operation

Die Ausführung der Operation teilt sich in folgende Einzelschritte:

1. Über die serielle Schnittstelle werden zunächst die Operanden für die mathematische Operation eingelesen.
2. Die Operanden werden über eine IO-Operation an die Registeradressen der PRR gesendet (Quelltext 4.15).

```
XIo_Out32(XPAR_XPS_DCR_SOCKET_0_BASEADDR,first);
XIo_Out32(XPAR_XPS_DCR_SOCKET_0_BASEADDR+4,second);
```

Quelltext 4.15: Schreiben der Operanden in die Register der PRR

3. Direkt im Anschluss wird das Ergebnis der Operation über die Adresse des ersten SW-Registers gelesen (Quelltext 4.16).

```
result=XIo_In32(XPAR_XPS_DCR_SOCKET_0_BASEADDR);
```

Quelltext 4.16: Lesen des Operationsergebnisses von der PRR

4. Das Ergebnis wird über die serielle Schnittstelle ausgegeben.

Die Funktionsweise der partiellen Rekonfiguration wurde durch mehrmaliges Austauschen der add.bit, mult.bit und blank.bit getestet. Anhand der erhaltenen Ergebnisse wurde ein korrektes Verhalten des Systems verifiziert.

5. µCLinux für ein selbstrekonfigurierendes µBlaze-System

In diesem Kapitel wird das erstellte µCLinux-PR-Entwicklungssystem dokumentiert. Dieses bildet eine Plattform, mit der sich mehrere SW-Anwendungen durch zur Laufzeit geladene HW-Module beschleunigen lassen. Das µBlaze-System aus Kapitel 4 wird für die Konfiguration und Installation eines Linux 2.6.31 Kernels erweitert. Mit der Petalinux Entwicklungsumgebung wird ein µCLinux-Kernel für dieses HW-System konfiguriert. Das µCLinux-System bildet die Plattform für die quasi-parallele Ausführung mehrerer SW-Anwendungen und die Ressourcen zur Steuerung der partiellen Rekonfiguration. Ein bestehender Treiber zur Bereitstellung der Rekonfigurationsmechanismen wird zur Ausführung unter Linux 2.6.31 angepasst, für das Fifo-basierte XPS-HWICAP-Modul des µBlaze-Systems konfiguriert und dem Linux Kernel hinzugefügt. Mit einer Testapplikation wird die partielle Rekonfiguration unter Linux verifiziert.

5.1. Systemaufbau

Das Entwicklungssystem besteht aus folgenden Einheiten (vgl. Abbildung 5.1):

- Entwicklungsrechner
  - CentOS 5.4 als Entwicklungsbetriebssystem.
  - Konfiguration, Generierung und Download des u-Boot Bootloaders und Linux-Kernels.
  - Benutzer Ein- und Ausgabe über das Kermit Terminal-Programm.
- Xilinx ML-505 Zielplattform
  - Virtex XC5VLX50T FPGA zur Konfiguration des selbstrekonfiguriendenµBlaze-HW-Systems.
  - Ethernet-Phy zum Datenaustausch mit dem Entwicklungsrechner.
  - Compact-Flash Speicherkarten-Anbindung über den SysACE-Controller zur Ablage des initialen Bitstreams.
  - 32 MB Flash Memory zur persistenten Ablage des u-Boot Bootloaders und Linux-Kernels.
  - 256 MB DDR2 Memory zur Erweiterung des Arbeitsspeichers.

– RS232-Schnittstelle zur Kommunikation mit einem Benutzer über den Entwicklungsrechner.

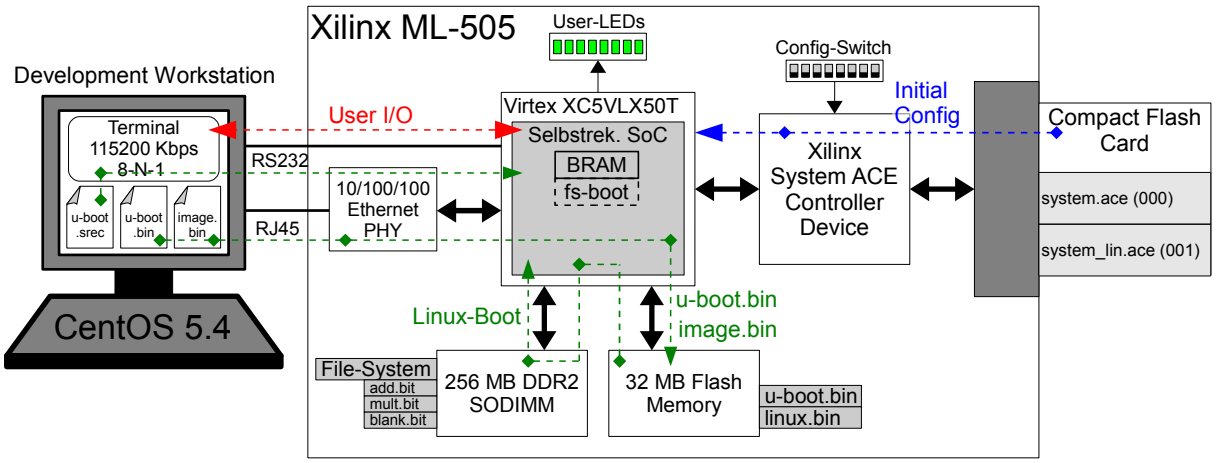


Abbildung 5.1.: µCLinux-PR-Entwicklungssystem; Entwicklungsrechner mit CentOS 5.4; Xilinx ML-505 FPGA-Plattform zur Integration des selbstrekonfigurierenden SOCs; Speicherübersicht zu den Linux-Boot-Files.

Das Einschalten des ML-505-Boards initiiert den automatischen Systemstart (vgl. Abbildung 5.2):

1. Die Konfigurationsdaten (system_lin.ace), bestehend aus Bitstream des selbstrekonfigurierenden SoCs und dem Bootloader der ersten Stufe (fs-boot) als eingebettete SW-Applikation, werden zur initialen Konfiguration des FPGA vom CF-Speicher gelesen. Über den Configuration-Switch wird die Adresse des zu wählenden Bitstreams (system_lin.ace) eingestellt.

2. fs-boot wird ausgeführt. Befindet sich der Bootloader der zweiten Stufe (u-boot.bin) bereits im Flash-Speicher, wird nach Ablauf des Time-outs dieser (u-boot.bin) vom Flash geladen. Andernfalls lädt fs-boot über die serielle Schnittstelle u-Boot (u-boot.srec) in den Arbeitsspeicher (DDR 2 Memory).

3. Anschließend wird u-Boot ausgeführt. Befindet sich der Linux-Kernel (linux.bin) bereits im Flash-Speicher, wird nach Ablauf des Time-outs Linux vom Flash geladen. Andernfalls legt u-Boot eine eigene Kopie (u-boot.bin) im Flash-Speicher ab.

Befinden sich u-Boot und der Linux-Kernel bereits im Flash-Speicher, werden diese bei jedem Systemstart nach Ablauf des Time-outs automatisch geladen.

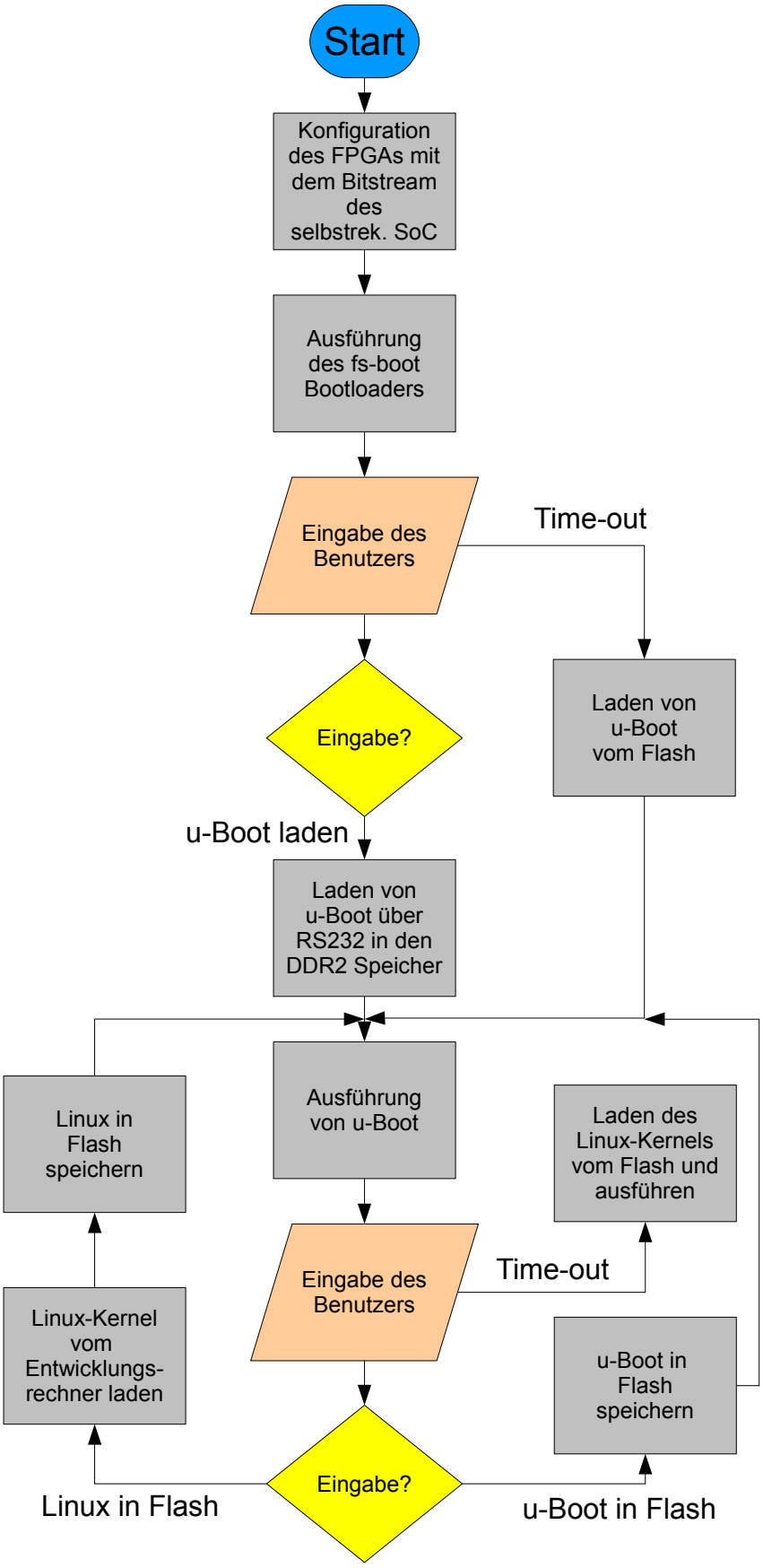


Abbildung 5.2.: Flussdiagramm des automatischen Starts des selbstrekonfigurierenden SoCs mit Linux

5.2. Erweiterung des selbstrekonfigurierenden µBlaze-HW-Systems zur Integration von Linux

Das µBlaze-System aus Kapitel 4.1 wird erweitert, um die HW-Plattform für die Installation und Ausführung von Linux anzupassen (vgl. Abbildung 5.3):

- Durch eine kaskadierte DCM-Struktur werden die drei Taktsignale des Systems generiert.
- Bestehende Module des µBlaze-Prozessor-Systems aus Kapitel 4.1 werden angepasst und weitere Komponenten zur Integration von Linux hinzugefügt (vgl. Abbildung 5.4).

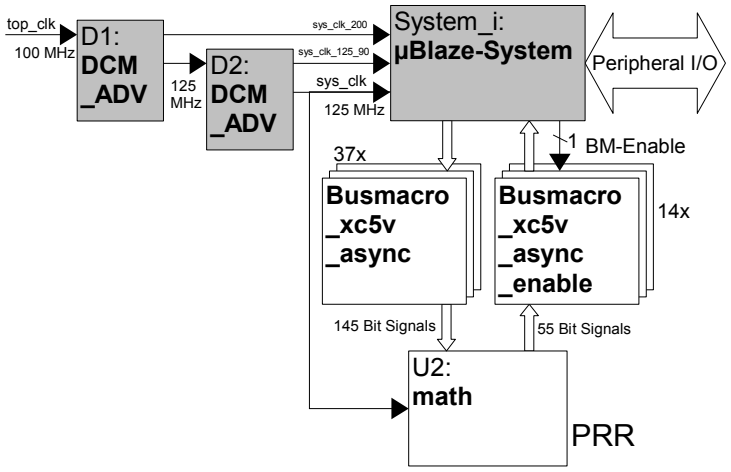


Abbildung 5.3.: Erweiterung des selbstrekonfigurierenden µBlaze-Systems aus Kapitel 4.1 zur Integration von Linux

Die Busmacros und die PRMs werden unverändert eingesetzt.

5.2.1. Erweitertes µBlaze-Prozessor-System

An dem PR-System nach Kapitel 4.1 wurden folgende Modifikationen durchgeführt, die die schnellere Programmausführung aus dem externen DDR2-Speicher und die Kommunikation mit dem Entwicklungsrechner unterstützen (vgl. Abbildung 5.4, Projektdateien s. Anhang A):

- **µBlaze-Prozessor**
  - **System-Bus-Takt:** 125 MHz
    Der System-Takt wurde erhöht, um die Kompatibilität mit der MPMC-Komponente sicher zu stellen.
  - **Zusätzl. Instruktionen:** Barrel Shifter für schnelle 2er-Potenzoperationen
    HW-Ergänzungsmodul zur ALU schiebt Datenworte mit einer zur Laufzeit parametriebaren Anzahl von Bits in einem Systemtakt [23].

– **Zusätzl. Schnittstelle:** L1 Cache Interface
Über das Xilinx Cache Link wird der Multiport Memory Controller des DDR2-RAMs mit Block-RAM-basiertem Cache verbunden, das direkt am µBlaze-Prozessor angekoppelt ist [23]. Parallele Pfade zum PLB reduzieren die Daten- und Instruktionszugriffslatenz.

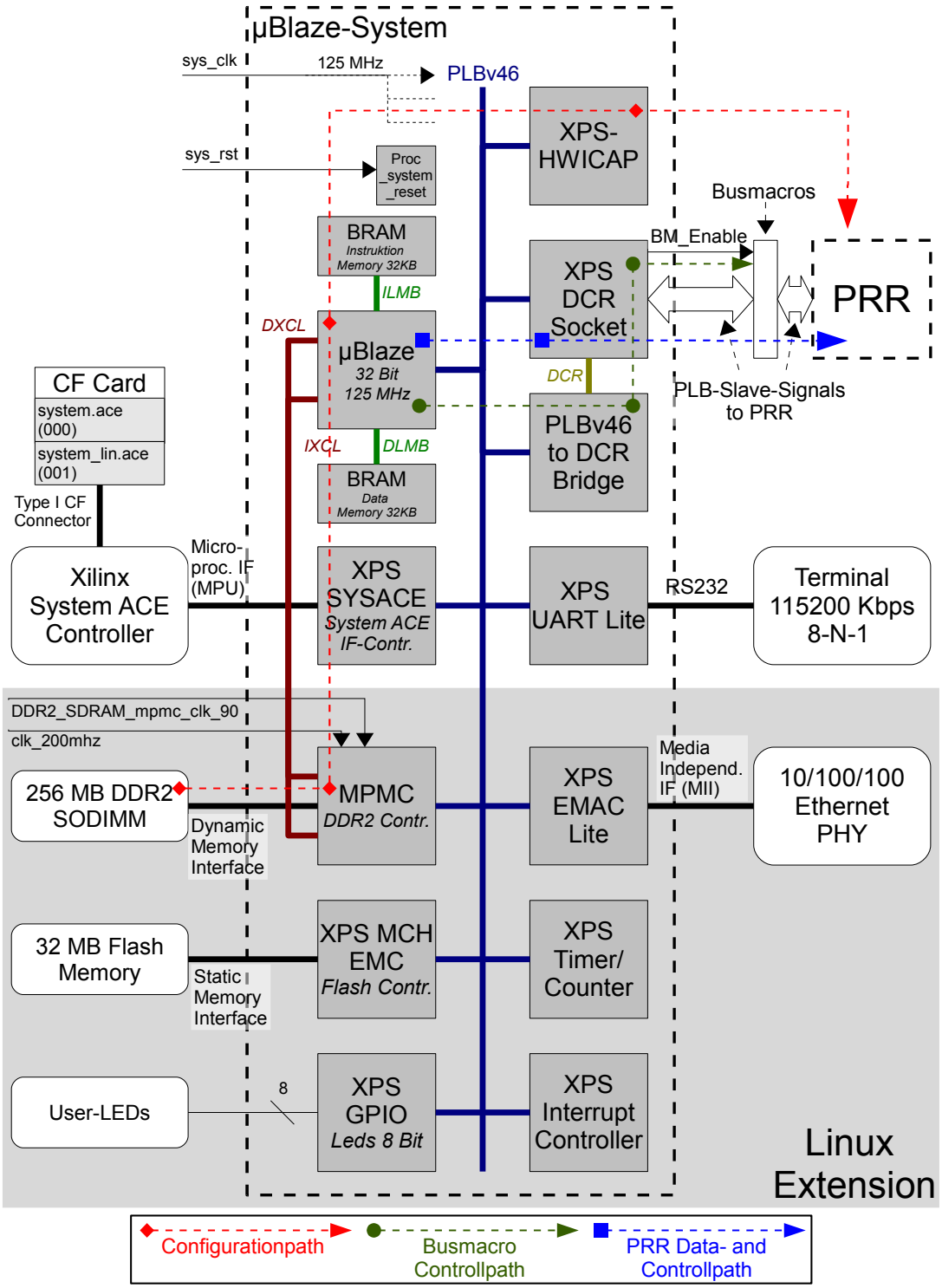


Abbildung 5.4.: Erweitertes µBlaze-System nach Abbildung 4.2 mit zusätzlichen Speicher- und Interface-Komponenten für die Integration eines Linux-Kernels. Kennzeichnung der Konfigurations-, Steuer- u. Nutzsignalpfade.

- **XPS UART Lite**
  Unter Linux wird diese Schnittstelle zur Ein- und Ausgabe über ein Terminal-Programm verwendet. Dieses Modul wurde um die Interrupt-Funktionalität erweitert, um unter Linux einen zuverlässigen Datentransfer über diese Schnittstelle sicher zu stellen.

- **Partielle Rekonfiguration**
  Die partiellen Bitstreams (add.bit, mult.bit, blank.bit) werden im Dateisystem abgelegt, das im externen DDR2-Speicher installiert ist. Den Beginn des Konfigurationspfades initiiert der µBlaze-Prozessor durch das Lesen der Bitstreams aus dem Dateisystem über die MPMC-Schnittstelle. Die Weitergabe der Bitstreamdaten erfolgt wie bei dem µBlaze-System aus Kapitel 4.1 durch das Senden der Daten an das HWICAP-Modul.

Die folgenden IPs mit PLB-Slave-Schnittstelle wurden dem System hinzugefügt:

- **XPS Interrupt Controller**
  Die Interrupt-Quellen des Timers sowie der UART-Lite und der EMAC-Lite Module werden über den Interrupt Controller gebündelt und als einzelnes Signal zum µBlaze-Prozessor geführt [31]. Register zur Prüfung, Freigabe sowie Bestätigung der Interrupts werden bereitgestellt und über die PLB-Schnittstelle zugänglich gemacht.

- **XPS Timer/Counter**
  Der Timer liefert die Timer-Ticks für die Ausführung von Linux [33]. Die Konfiguration des Timers zur Auslösung von Interrupts wird durch Linux konfiguriert.

- **XPS EMAC Lite**
  Dieses Modul bildet eine Schnittstelle über Ethernet zu einem Entwicklungsrechner [28]. Mit einer Transfergeschwindigkeit von 100 Mbps lassen sich neue Kernel-Images (linux.bin) in den Flash-Speicher laden sowie Daten zwischen Entwicklungsrechner und Linux-Kernel austauschen.

- **XPS MCH EMC**
  Der Xilinx Multi-Channel External Memory Controller wird hier zur Kopplung des 32 MB Flash-Speichers verwendet [32]. In diesem Speicher werden der U-Boot-Bootloader (u-boot.bin) und der Linux-Kernel (linux.bin) gespeichert. Nach jeder Systemunterbrechung wird zunächst der Bootloader aus dem Flash in den DDR2-Speicher geladen, der dann den Linux-Kernel ebenso in den DDR2-Speicher lädt. Durch den Einsatz dieses persistenten Speichers entfällt das manuelle Laden des Bootloaders und Linux-Kernels.

- **MPMC**
  Der Multi-Port Memory Controller wird hier zur Anbindung und Steuerung eines 256 MB DDR2-RAMs verwendet. Mit diesem DDR2-RAM wird der Arbeitsspeicher des Systems erweitert. Über die XCL-Schnittstelle zum µBlaze-Prozessor wird dieser Speicher zu einem 4 KB großen Cache für je Daten und Instruktionen gepuffert. Der MPMC arbeitet mit drei von DCMs generierten Taktsignalen (vgl. Abbildung 5.4) [25]:

  – **MPMC_Clk** 125 MHz Haupttaktsignal des MPMC; generiert das Taktsignal des angeschlossenen Speichers.

  – **MPMC_Clk90** Haupttaktsignal des MPMC um 90 Grad verschoben. Durch Invertierung des MPMC_Clk und MPMC_Clk90 Signals stehen insgesamt vier Taktsignale

zur Verfügung, die jeweils um 90 Grad von einander verschoben sind. Diese Taktsignale werden vom MPMC zur Einhaltung der Timings für DDR und DDR2 Speicher verwendet.

– **MPMC_Clk_200MHz** 200 MHz Taktsignal; Dieses Signal wird nur in Virtex-4 und Virtex-5 Architekturen für den Betrieb des Memory Interface Generator basierten PHYs verwendet.

Der MPMC wird unter Linux verwendet, um den Arbeitsspeicher des Systems zu erweitern sowie das transiente Dateisystem abzulegen.

- **XPS GPIO**
  Dieses GPIO-Modul wird zum Testen der Ausführung von SW-Applikationen unter Linux verwendet [29]. Ein Testprogramm (gpio-test) wird ausgeführt, dessen Status von den LEDs angezeigt wird.

5.2.2. DCM-Struktur und Instanziierung

Für dieses System werden aus einem 100 MHz Oszillator-Taktsignal drei Taktsignale durch Kaskadierung von zwei DCM-Komponenten generiert (siehe Abbildung 5.5):

- **125 MHz:** Systemtakt für alle Komponenten des µBlaze-Systems.
- **125 MHz/ 90 Grad:** Für die MPMC-Komponente wird ein 125 MHz Taktsignal generiert, dessen Phasenlage um 90 Grad von dem des Systemtaktes verschoben ist (vgl. Abschnitt 5.2.1).
- **200 MHz:** Ein 200 MHz Taktsignal wird ebenfalls für die MPMC-Komponente generiert.

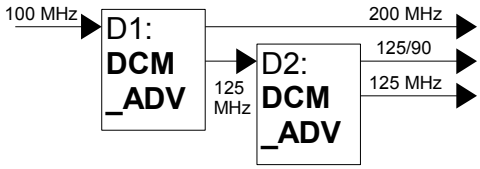


Abbildung 5.5.: Aufbau der DCMs zur Generierung der Taktsignale

Mit **D**igital **C**lock **M**anager Blöcken lassen sich aus einem Eingangstakt, Taktsignale mit sowohl höherer, als auch niedrigerer Frequenz und variabler Phasenlage generieren. Die DCM-Blöcke der Virtex-5 FPGA-Reihe lassen sich über zwei Librarymodule, DCM_ADV (dynamisch rekonfigurierbar) bzw. DCM_BASE (statische Konfiguration) instanziieren (vgl. Abbildung 5.6) [36].

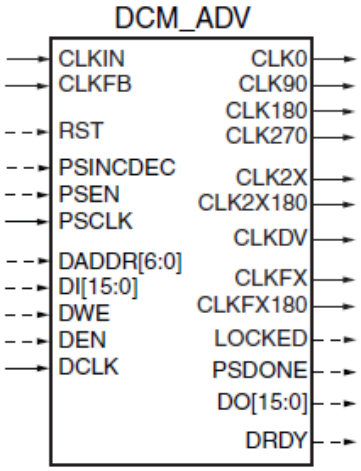


Abbildung 5.6.: Ein- und Ausgangssignale eines DCM_ADV

Zur Generierung o.g. Frequenzen werden für das entwickelte µBlaze-System zwei DCM_ADV-Komponenten instanziiert:

Eingangssignale:

- **CLKIN:** Referenzsignal (100 MHz); Aus diesem Signal werden alle Ausgangssignale generiert.
- **CLKFB:** Taktrückführungssignal (FeedBack); Das vom DCM erzeugte Signal wird zur Synchronisation mit dem externen Referenzsignal an diesen Anschluss zurückgeführt.

Ausgangssignale:

- **CLK0:** Taktsignal mit der Frequenz des Referenzsignals CLKIN.
- **CLK90:** Dieses Taktsignal liefert dieselbe Frequenz wie das Referenzsignal CLKIN und ist zu CLK0 um 90 Grad Phasenverschoben.
- **CLK2X:** Doppelte Frequenz wie das CLK0-Signal und ist mit diesem Phasengleich.
- **CLKFX:** Parametrierbare Taktfrequenz:

$$CLKFX = (M/D) \times CLKIN$$

M und D werden durch die Integer-Generics CLKFX_**M**ULTIPLY (2...32) und CLKFX_**D**IVIDE (1,5...16) gesetzt. Wie in Abbildung 5.5 gezeigt, werden hier zwei DCM_ADV D1 und D2 instanziiert (vgl. Quelltext 5.1 und 5.2):

```
1 D1: DCM_ADV
2   generic map (
3     CLKFX_DIVIDE => 4,
4     CLKFX_MULTIPLY => 5,      -- 5/4 = 1,25
5     CLK_FEEDBACK => "1X",
6     SIM_DEVICE => "VIRTEX5"
7   )
8   PORT MAP (
9     CLKIN => top_clk_pin,     -- 100 MHz Oszillator
```

```
10      RST => '0',
11      CLK0 => sys_clk_d1_fb,    -- out FB
12      CLKFB => sys_clk_d1_fb,   -- in FB
13      CLKFX => sys_clk_125,     -- Systemtakt
14      CLK2X => sys_clk_200      -- zum MPMC
15      );
```

Quelltext 5.1: VHDL Instanziierung der DCM_ADV-Komponente D1

- Als Referenzsignal am CLKIN-Eingang erhält D1 das Eingangstaktsignal des Systems (top_clk_pin) mit einer Frequenz von 100 MHz.
- Mit den Einstellungen der Generics CLKFX_DIVIDE und CLKFX_MULTIPLY generiert D1 für CLKFX folgende Frequenz:

$$CLKFX = (5/4) \times 100MHz = 125MHz$$

- Das 200 MHz Taktsignal für die MPMC-Komponente wird durch CLK2X aus dem Eingangstakt von 100 MHz erzeugt.
- Das erzeugte Signal aus CLK0 wird zur Synchronisation an CLKFB zurückgeführt.

```
1  D2: DCM_ADV
2    generic map (
3      CLK_FEEDBACK => "1X",
4      SIM_DEVICE => "VIRTEX5"
5    )
6    PORT MAP (
7      CLKIN => sys_clk_125,
8      RST => '0',
9      CLK0 => sys_clk,           -- µBlaze-System
10     CLKFB => sys_clk,
11     CLK90 => sys_clk_125_90,   -- zum MPMC
12     LOCKED => Dcm_all_locked   -- Clock-Verfügbarkeitsanzeige
13     );
```

Quelltext 5.2: VHDL Instanziierung der DCM_ADV-Komponente D2

- Als Referenzsignal am CLKIN-Eingang erhält D2 das CLKFX-Signal von D1 mit der Frequenz von 125 MHz.
- CLK0, mit 125 MHz, taktet das µBlaze-System und wird als Rückführung zu CLKFB verwendet.
- Ein um 90 Grad phasenverschobenes Signal mit einer Frequenz von 125 MHz wird aus dem Referenzsignal von D2 erzeugt und über CLK90 herausgeführt.

In der UCF-Datei wird festgelegt, welche DCM-Komponente des FPGAs von D1 bzw. D2 verwendet wird (vgl. Quelltext 5.3):

```
INST "D1" LOC = DCM_ADV_X0Y0;
INST "D2" LOC = DCM_ADV_X0Y1;
```

Quelltext 5.3: Positionsfestlegung der DCMs D1 und D2 in der UCF-Datei

In der UCF-Datei wird ebenso die Timingvorgabe des Systemtaktes gemacht. In Quelltext 5.4 wird die Periodendauer für sys_clk_pin mit 8000 Picosekunden spezifiziert. Dieses entspricht einer Taktrate von 1/8 ns = 125 MHz. Sämtliche Pfade der kombinatorischen Logik des Systems dürfen demnach eine Signallaufzeit von 8 Nanosekunden nicht überschreiten.

```
Net sys_clk_pin TNM_NET = sys_clk_pin;  -- Gruppierungsconstraint
TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 8000 ps;
```

Quelltext 5.4: Beschränkung der Signallaufzeit der kombinatorischen Logik

Mit “TNM_NET“ werden alle Pfade, die von sys_clk_pin getrieben werden zu einer Gruppe zusammengefasst. “TIMESPEC“ legt die Periodendauer für alle Elemente der Gruppe fest.

5.2.3. SWplattform

Zur Vorbereitung des µBlaze-HW-Systems für die Ausführung von Linux wird die SW Spezifikation in der MSS-Datei (Microprocessor SW Specification) des Systems für Petalinux eingestellt (vgl. Quelltext 5.5):

```
BEGIN OS
 PARAMETER OS_NAME = petalinux
 PARAMETER OS_VER = 1.00.b
 PARAMETER PROC_INSTANCE = microblaze_0
 PARAMETER lmb_memory = dlmb_cntlr
 PARAMETER flash_memory = FLASH
 PARAMETER main_memory = DDR2_SDRAM
 PARAMETER stdin = RS232_Uart_1
 PARAMETER stdout = RS232_Uart_1
END
```

Quelltext 5.5: Petalinux wird in der MSS-Datei als Betriebssystem für die erstellte µBlaze-HW spezifiziert

Dieser Eintrag in der MSS-Datei bewirkt durch Aufruf von “Generate Libraries and BSPs“ in XPS, dass das Board Support Package (BSP) für das entwickelte HW-System durch die Petalinux-Werkzeuge erstellt wird. Das BSP liefert implementierungsspezifischen Quellcode für das erstellte µBlaze-System. Dieser Quellcode dient dem Linux-Betriebssystem zur Initialisierung und Kommunikation mit den HW-Ressourcen des Systems [8]. Enthalten sind u.a. Quellcode für:

- die Boot-Phase
- Geräte-Treiber
- Initialisierung

Zusätzlich wird eine Linux-konforme Konfigurationsdatei (KConfig.auto) erstellt, in der Daten bezüglich des erstellten HW-Systems enthalten sind. Basierend auf diesen Daten wird durch den automatischen Linux-Build-Prozess der Linux-Kernel für das erstellte HW-System angepasst.

Der lokale Programm-Speicher (64 KB) des µBlaze-Systems bietet ausreichend Ressourcen für die Ablage des Bootloaders der ersten Stufe (fs-boot, 7 KB). Dieser Bootloader ist Bestandteil der Petalinux-Distribution und wird dem System als eingebettete SW-Applikation hinzugefügt. Nach der Konfiguration des FPGAs mit dem Bitstream des selbstrekonfigurierenden SoCs wird fs-boot automatisch aus dem BRAM ausgeführt. fs-boot lädt zunächst den Bootloader der zweiten Stufe (u-Boot) und dieser anschließend den Linux-Kernel.

5.3. Generierung des µCLinux-Kernels

Das erstellte Linux-System lässt sich folgendermaßen kategorisieren:

- **Embedded Linux:**
  Die durch Petalinux zur Verfügung gestellte Linux-Distribution ist speziell für FPGA-basierte HW-Systeme angepasst. Der Linux-Kernel und die Gerätetreiber sind für die Ausführung auf dem µBlaze-Prozessor und der dazugehörigen Peripherie zugeschnitten [44].
- **µCLinux:**
  Eine Linux-Distribution für µController-Systeme ohne Memory Management Unit (MMU) [44]. Für das entwickelte µBlaze-System wurde keine MMU konfiguriert. Die Wahl dieser Linux-Variante für das selbstrekonfigurierende µBlaze-System bietet folgende Vorteile:
  - komplette Funktionalität des aktuellen Linux-Kernels (Treiber, Dateisysteme, ROMFS, Konsole)
  - geringere Imagegröße gegenüber Standard-Linux
  - uneingeschränkter Zugriff auf den gesamten Speicherraum

  Ohne MMU lässt sich jeder Teil des Betriebssystems durch einen anderen Prozess manipulieren, da Prozesse gegenseitig Zugriff auf ihre Speicherbereiche haben [15]. Das erstellte selbstrekonfigurierende µBlaze-System führt somit eine direkte Kommunikation mit den PRMs über direkte Register-Lese- bzw. Schreiboperationen durch. Die Treibererstellung für die PRMs ist nicht erforderlich.
- **Zeit- und Schedulingeigenschaften:**
  Das hier erstellte Linux-System beruht auf dem Standard Linux-Kernel. Eine zuverlässige Echtzeitfähigkeit ist bei diesem System nicht gegeben, da das Standard Linux ein universelles Mehrbenutzersystem ist [1]. Es lassen sich keine definierten Latenzen einstellen. Somit ist ein deterministisches Systemverhalten nicht zu erreichen.
- **Read-only Dateisystem:**
  Die Dateien des Systems werden zur Laufzeit nicht modifiziert. Als Dateien werden die ausführbare Test-Applikation und die Bitstreams der PRMs im Dateisystem gespeichert. Diese werden vor der Generierung des Linux-Systems in das Dateisystem integriert. Das generierte Dateisystem ist nicht modifizierbar.

Mit der Petalinux Werkzeugkette wurde ein SW-System aus folgenden Modulen konfiguriert:

- **Board Support Package:**
  µBlaze-spezifischer Code zur Initialisierung der HW [8]. Dieser Hardware-nahe Code wird sowohl vom u-Boot Bootloader, als auch vom Linux-Kernel verwendet.
- **u-Boot:**
  Der Bootloader der zweiten Stufe lädt den Linux-Kernel [4]. Die Petalinux-Werkzeugkette passt u-Boot auf die µBlaze-HW an.
- **µCLinux-Kernel:**
  - Standard Linux ohne Virtual Memory Unterstützung
  - µCLibc

  Neben den Standard-Bestandteilen des µCLinux-Kernels, werden durch Petalinux Xilinx-spezifische Geräte-Treiber hinzugefügt:
  - GPIO
  - Emac-Lite
  - UART-Lite
- **Root File-System CRAMFS [44]:**
  Das Root File-System enthält eine vordefinierte Verzeichnisstruktur, in der jedes Verzeichnis spezifische Dateien für das Betriebssystem bereit hält. Als Dateisystemtyp wird für diese Linux-Implementierung das Compressed RAM File System (CRAMFS) verwendet. Dieses bietet ein Minimum an Funktionalität zum Lesen von Dateien, ist read-only und komprimiert im DDR2-Speicher des Systems implementiert.

5.3.1. Linux-Konfigurationsflow mit Petalinux

Zur Konfiguration eines µCLinux-Kernels für das entwickelte µBlaze-System wurden folgende Schritte durchgeführt [11], [12]:

1. Einfügen des µBlaze-HW-Projekts in die Petalinux-Umgebung.
   Das mit EDK erstellte µBlaze-HW-Projekt wird in der Petalinux-Verzeichnisstruktur integriert. Dadurch wird der Werkzeugkette bekannt gemacht, für welche HW die jeweiligen SW-Module zu generieren sind. Die Petalinux-Umgebung besteht aus folgenden Verzeichnissen [13]:
   - **hardware:** EDK-HW-Projekte und die Petalinux Werkzeugkette zur Generierung eines BSPs.
   - **software:**
     - Aktuelle Linux-Build Umgebung.
     - Quellcode des Linux 2.4.x und 2.6.x Kernels.
     - Unterverzeichnisse für User-Anwendungen und Treibermodule.

- **tools:** Compiler und die Petalinux Werkzeugkette.

Das gesamte Verzeichnis des EDK-HW-Projektes für das erstellte µBlaze-System wird in das “hardware/user-platforms“ Unterverzeichnis kopiert.

2. Erstellung einer neuen Zielplattform für die Linux-Build-Werkzeugkette basierend auf dem µBlaze-System.
   Zunächst wird ein Platzhalter für diese Zielplattform erstellt. Im Verzeichnis “ /petalinux/software/petalinux-dist“ wird folgendes aufgerufen:

   petalinux-new-platform -v Xilinx -p Lin_PR_HW -k 2.6

   Als Boardhersteller wird Xilinx (-v) angegeben. Der Plattformname (-p) wird als Lin_PR_HW gekennzeichnet und es wird Kernelversion 2.6 (-k) für diese Plattform spezifiziert. Diese Zielplattform wird für den Build-Prozess über den Konfigurationsdialog (menuconfig) ausgewählt.

3. Exportieren der HW-Daten in den automatischen µCLinux-Build-Mechanismus.
   Die KConfig.auto-Datei mit den Daten bzgl. des erstellten HW-Systems wird in das Verzeichnis der erstellten Zielplattform kopiert. Im Verzeichnis des HW-Projektes “‘ /petalinux/hardware/user-platforms/Lin_PR_HW“ wird folgendes aufgerufen:

   petalinux-copy-autoconfig

   Die Werkzeuge für den automatischen Linux-Build-Mechanismus erhalten anhand der KConfig.auto-Datei Informationen über

   - die Konfiguration des µBlaze-Prozessors,
   - HW-Module des Systems und ihre jeweiligen Zugriffsadressen
   - und Interrupt-Quellen.

   Anhand dieser Informationen wird das µCLinux-System dem µBlaze-Prozessor angepasst und die Treiber für die vorhandenen HW-Module zur Integration in den Kernel eingefügt.

4. Weitere Anpassungen der Konfiguration an die HW-Plattform.
   Folgende Einstellungen werden über den Konfigurationsdialog (menuconfig) vorgenommen, bevor der Build-Prozess für das µCLinux-System ausgeführt wird:

   - Die IP-Adresse des µBlaze-Systems wird auf 141.22.26.66, die des Servers (Entwicklungsrechner) auf 141.22.27.199 gesetzt. Damit wird die Kommunikation über das Netzwerk vorbereitet.
   - Der Dateisystemtyp wird auf CRAMFS eingestellt.
   - gpio-test wird als User-Applikation integriert.

5. Ausführen des µCLinux-Build-Prozesses.
   Im Verzeichnis “ /petalinux/software/petalinux-dist“ wird mit dem Aufruf “make“ der Build-Prozess initiiert und das gesamte SW-System für das erstellte µBlaze-System im Verzeichnis “ /petalinux/software/petalinux-dist/images“ generiert.

Die folgenden generierten Dateien werden verwendet:

- **u-boot.srec:** Die ausführbare u-Boot-Datei im S-record-Format. Dieses u-boot-Image wird mit fs-boot über die serielle Schnittstelle vom Entwicklungsrechenr in den DDR2-Speicher geladen.
- **u-boot.bin:** Die ausführbare u-Boot-Datei im Binär-Format. Dieses u-boot-Image wird vom Entwicklungsrechner über die Ethernet-Schnittstelle in den Flash-Speicher geladen.
- **image.bin:** Der ausführbare µCLinux-Kernel und das Root-FS im Binär-Format. Dieses Linux-Image wird von u-Boot über die Ethernet-Schnittstelle vom Entwicklungsrechner in den Flash-Speicher geladen.

5.3.2. Booten von Linux

Für das erstellte System wurde ein automatischer Boot-Prozess bei Systemstart konfiguriert. Bei diesem Vorgang wird Linux durch die Abarbeitung folgender Schritte automatisch geladen und zur Ausführung gebracht (vgl. Abbildung 5.2):

1. Beim Systemstart wird fs-boot automatisch aus dem lokalen Speicher (BRAM) geladen (vgl. Kapitel 5.2.3).
2. fs-boot wartet auf die Image-Datei von u-Boot über die serielle Schnittstelle.

   FS-BOOT: Waiting for SREC image....
3. Über die serielle Schnittstelle wird u-Boot vom Entwicklungsrechner an das µBlaze-System gesendet.

   $ cat /tftpboot/u-boot.srec > /dev/ttyS0
4. Nach vollständiger Übertragung von u-Boot wird dieser automatisch zur Ausführung gebracht. u-Boot besitzt eine Kommandozeilen-Schnittstelle, mit der folgende Funktionen von u-Boot aufgerufen werden [4]:
5. u-Boot wird in den Flash-Speicher geladen.

   U-Boot> run update_uboot
6. Mit u-Boot wird das Linux-Image über Ethernet vom Entwicklungsrechner heruntergeladen.

   U-Boot> run load_kernel
7. u-Boot speichert das Linux-Image samt Dateisystem im Flash-Speicher ab.

   U-Boot> run insatll_kernel
8. Neustart des Systems bewirkt automatisches Laden von u-Boot und anschließend des Linux-Kernels.

5.4. Kernel-integrierter ICAP-Gerätetreiber

Zur Steuerung und Auslagerung der Rekonfigurationsmechanismen über das XPS-HWICAP-Modul wurde der bereits existierende xilinx_hwicap-Treiber [37] angepasst und dem Linux-System als Kernel-Modul hinzugefügt. Der ICAP-Treiber wurde für den Linux 2.6.31 Kernel modifiziert und in den Build-Prozess für das hier erstellte Linux-System mit eingebunden. Der Treiber stellt für quasi-parallele SW-Anwendungen eine Schnittstelle zur Rekonfiguration des Virtex 5 FPGAs bereit. Die Rekonfigurationsmechanismen werden von diesem Treiber implementiert, wodurch sich die Größe der Anwendungen reduziert.

5.4.1. Aufbau des Treibers

Der ICAP-Treiber besteht aus folgenden Entwurfsdateien (vgl. Anhang A):

- **xilinx_hwicap.c/h:** Haupt-Treiberdatei; Bietet die Linux-konforme Treiberstruktur und stellt die Schnittstelle zum System-Call-Interface (SCI) bereit.
- **fifo_icap.c/h:** Zugriffsfunktionen auf Fifo-basierte XPS-HWICAP-Module. Die Rekonfigurationsmechanismen werden innerhalb dieser Datei als Funktionen definiert, die die Zugriffe auf die Register des XPS-HWICAP-Moduls vornehmen.
- **buffer_icap.c/h:** Zugriffsfunktionen auf Buffer-basierte OPB-HWICAP-Module. Für ältere Buffer-basierte HWICAP-Module mit On Chip Peripheral Bus-Schnittstelle werden die Funktionen dieses Moduls zur Rekonfiguration eingesetzt.
- **Makefile:** Compileraufruf mit speziellen Optionen zur Übersetzung des Treibers als Kernelmodul. Dieses Makefile wird im Linux-Build-Prozess integriert, um diesen Treiber als Bestandteil des µCLinux-Kernel hinzuzufügen.
- **xicap.c:** Eine “platform_device“-Struktur wird für jede XPS-HWICAP-Komponente erstellt. Die Zugriffsadressen für das HWICAP werden in dieser Struktur über die Einträge in der KConfig.auto-Datei eingefügt. Der Treiber greift über diese Struktur auf die Register des HWICAP-Moduls zu.

Das entwickelte µBlaze-System setzt das aktuellere Fifo-basierte XPS-HWICAP-Modul ein. Aus diesem Grund werden für den Treiber die Funktionen der Fifo-basierten Zugriffsschnittstelle (fifo_icap.c/h) verwendet.

Treiberstruktur und SCI-Schnittstelle [16]:

- **hwicap_module_init():** Diese Funktion wird zum Laden des Treibers in den Kernel ausgeführt.
- **hwicap_module_cleanup():** Beim Herausnehmen des Moduls aus dem Kernel werden die Init-Aktionen rückgängig gemacht.
- **hwicap_open():** Reserviert den ICAP-Treiber und initialisiert dessen Ressourcen für die aufrufende Applikation.
- **hwicap_release():** Diese Funktion wird durch den Aufruf von close() anwendungsseitig ausgeführt und gibt die Ressourcen des ICAP-Treibers frei.

- **hwicap_write():** Durch den Aufruf dieser Funktion werden Daten-Bytes zur Rekonfiguration an den Treiber übergeben.
- **hwicap_read():** Die aktuelle Konfiguration wird beim Aufruf dieser Funktion durch den Treiber gelesen und an die aufrufende Anwendung übergeben.

5.4.2. Funktionsweise des xilinx_hwicap-Treibers

Drei Teilaufgaben werden von diesem Treiber ausgeführt:

1. Initialisierung
2. Rekonfiguration des FPGAs
3. Lesen der aktuellen Rekonfiguration des FPGAs

Initialisierung

1. Während der Bootphase des Systems wird die “platform_device“-Struktur (xicap.c) des XPS-HWICAP-Moduls initialisiert und im System registriert. Die Zugriffsadressen für das HWICAP-Modul stehen über diese Struktur bereit.
2. Vom Kernel wird die hwicap_module_init()-Funktion aufgerufen:
   a) Der Treiber wird beim IO-Management des Systems angemeldet und die dazugehörige Gerätenummer registriert.
   b) Der Kernel übergibt die “platform_device“-Struktur des HWICAP-Moduls an den zugehörigen Treiber.
   c) Der Treiber initialisiert das HWICAP-Modul mit den erhaltenen Zugriffsadressen aus der “platform_device“-Struktur und steht anschließend für User-Space-Anwendungen bereit.

Rekonfiguration des FPGAs

1. Die Gerätedatei des Treibers wird mit hwicap_open() geöffnet und der Treiber initialisiert.
2. hwicap_write() wird aufgerufen und die Rekonfigurationsdaten übergeben.
   a) Die Funktion set_configuration() des fifo_icap.c-Moduls wird zur Konfiguration aufgerufen.
   b) Durch das Beschreiben der Register des HWICAP-Moduls sendet diese Funktion die Rekonfigurationsdaten in 32-Bit Blöcken an das Schreib-Fifo des HWICAPs.

Lesen der aktuellen Rekonfiguration des FPGAs

1. Die Gerätedatei des Treibers bleibt weiterhin geöffnet.
2. hwicap_read() wird mit der Anzahl der zu lesenden Rekonfigurations-Bytes aufgerufen
   a) Die Funktion get_configuration() des fifo_icap.c-Moduls wird zum Empfang der Rekonfigurationsdaten aufgerufen.
   b) Durch das Lesen der Register des HWICAP-Moduls empfängt diese Funktion die Rekonfigurationsdaten in 32-Bit Blöcken vom Lese-Fifo des HWICAPs.

5.4.3. Anpassung des Treibers an das µCLinux-System

Der in diesem Linux-System eingesetzte ICAP-Treiber [37] lässt sich für Buffer- und Fifo-basierte HWICAP-Module konfigurieren. Das Buffer-basierte HWICAP-IP unterscheidet sich von der Fifo-basierten in der Verwaltung des Zwischenspeichers zur Ablage der Rekonfigurationsdaten. Das im µBlaze-System eingesetzte HWICAP-Modul (vgl. Kapitel 5.3) verfügt über einen 1024x32-Bit Schreib-FIFO, dem die Rekonfigurationsdaten mit der Treiber-Funktion fifo_icap_set_configuration() in Blöcken übertragen werden. Durch Aufruf von fifo_icap_start_config() werden die Daten durch das HWICAP-Modul aus dem Fifo entnommen und zur Rekonfiguration verwendet. Buffer-basierte HWICAP-IPs verfügen über einen BRAM-Speicher zur Ablage der Rekonfigurationsdaten. Das Ablegen der Daten und Weiterleiten zur Rekonfiguration für dieses Modul wird vom ICAP-Treiber verwaltet. Folgende Bereiche des Treibers wurden modifiziert, um diesen Treiber unter dem Linux 2.6.31 Kernel funktionsfähig zu machen und an das Fifo-basierte HWICAP-Modul [30] anzupassen:

- In der Haupt-Treiberdatei (xilinx_hwicap.c) wird während der Initialisierung die Schnittstelle des fifo_icap.c-Moduls spezifiziert (vgl. Quelltext 5.6 und 5.7). Diese Technik gestattet die Anpassung des Treibers mit der hwicap_driver_config-Struktur für die verschiedenen HWICAP-Module (Buffer- oder Fifo-basiert) durch eine einzige Änderung.

```
1 static struct hwicap_driver_config fifo_icap_config = {
2         .get_configuration = fifo_icap_get_configuration,
3         .set_configuration = fifo_icap_set_configuration,
4         .get_status = fifo_icap_get_status,
5         .reset = fifo_icap_reset,
6 };
```

Quelltext 5.6: Schnittstelle des fifo_icap-Moduls. Diese Funktionen stehen dem ICAP-Treiber zum Schreiben und Lesen von Rekonfigurationsdaten bereit.

```
1 static int
2 __devinit hwicap_drv_probe(struct platform_device *pdev)
3 {
4 ...
5 // Schnittstelle des fifo_icap-Moduls
6 // wird an die Setup-Routine übergeben (fifo_icap_config)
7 return hwicap_setup(&pdev->dev, pdev->id, res,
8                         &fifo_icap_config, regs);
9 }
```

```
10
11 // Treiber Setup-Routine
12 static int __devinit hwicap_setup(struct device *dev, int id,
13                 const struct resource *regs_res,
14                 const struct hwicap_driver_config *config,
15                 const struct config_registers *config_regs)
16 {
17 ...
18 // Struktur (hwicap_drvdata) beinhaltet sämtliche Treiber-Daten
19 // u.a. die Schnittstelle zum fifo_icap-Modul
20 // (hwicap_driver_config)
21 struct hwicap_drvdata *drvdata = NULL;
22 ...
23 // die Schnittstelle wird in der Treiber-Struktur gespeichert.
24 drvdata->config = config;
25 ...
26 }
27
28 static ssize_t
29 hwicap_write(struct file *file, const char __user *buf,
30                 size_t count, loff_t *ppos)
31 {
32 ...
33 // Beim Schreibvorgang wird die Schnittstelle
34 // des fifo_icap-Moduls aufgerufen.
35 status = drvdata->config->set_configuration(drvdata,
36                                 kbuf, len >> 2);
37 ...
38 }
```

Quelltext 5.7: Initialisierung des HWICAP-Moduls durch den Treiber. Die Schnittstelle für den Zugriff auf das fifo_icap-Modul wird verwendet.

- Der unmodifizierte Treiber liest über das Vacancy-Register den Füllstand des Fifos, sodass nicht mehr Daten gesendet werden, als dieses Fifo empfangen kann (vgl. Quelltext 5.8).

```
1 int fifo_icap_set_configuration(struct hwicap_drvdata *drvdata,
2                 u32 *frame_buffer, u32 num_words)
3 {
4 ...
5
6 // write_fifo_vacancy = Freie Plätze des Fifos
7 while (write_fifo_vacancy == 0) {
8     // lesen des Vacancy-Registers des write-Fifos
9     write_fifo_vacancy =
10         fifo_icap_write_fifo_vacancy(drvdata);
11 ...
```

Quelltext 5.8: Senden der Rekonfigurationsdaten an das Fifo des HWICAP-Moduls im unmodifizierten Treiber. Das Vacancy-Register wird zur Vermeidung von Datenverlusten überprüft.

Die Untersuchung des unmodifizierten Treibers zeigte, dass die Rekonfigurationsdaten an das HWICAP-Modul gesendet werden. Allerdings wurde nach mehrmaligen Rekonfigurationsversuchen das partielle Bitstream nicht konfiguriert. Es wurde beobachtet, dass trotz der Abfrage des Fifo-Leerstands Rekonfigurationsdaten verloren gegangen sind. Aufgrund dessen wurde der Leerstand des Fifos auf einen festen Wert von 4 gesetzt (vgl. Quelltext 5.9).

```
1 int fifo_icap_set_configuration(struct hwicap_drvdata *drvdata,
2                 u32 *frame_buffer, u32 num_words)
3 {
4 ...
5
6 if (write_fifo_vacancy == 0) {
7                         // maximale Fifo-Kapazität
8                         write_fifo_vacancy = 4;
9 ...
```

Quelltext 5.9: Eingrenzung der Fifo-Kapazität. Es lassen sich maximal vier 32-Bit Blöcke an das Fifo senden, bevor die Rekonfiguration angestoßen wird.

Durch diese Anpassung konnte die Rekonfiguration von partiellen Bitstreams mit diesem Treiber durchgeführt werden. Die Ursache für das fehlerhafte Verhalten des unmodifizierten Treibers bietet ein Untersuchungsziel für folgende Projekte.

5.4.4. Kernel-Integration

Zur Integration des Treibers in den Kernel wird der Quellcode des Treiber in den Kernel-Sourcetree eingefügt und in die jeweiligen Makefiles zur Eingliederung des Treibers im Build-Prozess erweitert:

1. Einfügen der ICAP Geräte-Initialisierungs-Struktur.
   Die Datei xicap.c wird in das Verzeichnis

   " /petalinux/software/petalinux-dist/linux-2.6.x/arch/microblaze/platform/common"

   kopiert. Das Makefile im selben Verzeichnis zur Kompilierung und Integration dieses Moduls zum Kernel wird folgendermaßen erweitert (Quelltext 5.10):

```
1 ...
2 # xicap.o wird in den Kernel eingefügt, wenn
3 # mindestens ein HWICAP-Modul im HW-System vorhanden ist.
4 platobj-$(CONFIG_XILINX_HWICAP) += xicap.o
5 ...
```

Quelltext 5.10: Das Makefile der Geräte-Initialisierungen für die Module des µBlaze-Systems wird für den ICAP-Treiber erweitert.

2. Der Quellcode des Treibers wird in das Linux Treiber-Verzeichnis eingefügt.

   a) Der ICAP-Treiber gehört zur Geräteklasse der zeichenorientierten Geräte [15]. Daher wird im Verzeichnis

“ /petalinux/software/petalinux-dist/linux-2.6.x/drivers/char“

das Verzeichnis “/xilinx_icap“ erstellt.

b) Die Quellcode-Dateien des Treibers (xilinx_icap.c/h, fifo_icap.c/h, buffer_icap.c/h) und das zugehörige Makefile werden in dieses Unterverzeichnis kopiert.

c) Das Makefile im Verzeichnis “../char“ wird angepasst, um den ICAP-Treiber im Linux-Kernel zu integrieren (vgl. Quelltext 5.11).

```
1 ...
2 # Dieser Eintrag bewirkt, dass das Makefile
3 # im Verzeichnis xilinx_icap aufgerufen wird und
4 # somit der ICAP-Treiber zum Kernel hinzugefügt wird
5 obj-$(CONFIG_XILINX_HWICAP) += xilinx_icap/
6 ...
```

Quelltext 5.11: Das Makefile zur Generierung aller Character-basierten Treiber wird mit dem Eintrag für den ICAP-Treiber erweitert.

3. Der Aufruf der Funktion device_create() in der Setup-Routine des Treibers erzeugt zur Laufzeit die zugehörige Gerätedatei. In diesem System kann diese Funktion die Gerätedatei nicht erzeugen, da ein read-only Dateisystem verwendet wird. Daher wird eine Gerätedatei für den ICAP-Treiber vor der Generierung des Dateisystem integriert. In das Verzeichnis

   “ /petalinux/software/petalinux-dist/romfs/dev“

   wird die Gerätedatei

   “@icap,c,50,0“

   eingefügt. Diese Datei wird über die Bezeichnung “icap“ geöffnet. Mit dem “c“-Eintrag wird diese Datei als Gerätedatei für einen Character-basierten Treiber spezifiziert. Zusätzlich wird die Gerätenummer mit 50 für die Major- und 0 für die Minornummer spezifiziert. Mit dieser Gerätenummer meldet sich der Treiber während der Initialisierung beim IO-Management des Systems an, wodurch das System die Gerätedatei zum Treiber zuordnet.

5.4.5. Funktions- und Auslagerungs-Test

Es wurde eine Test-Applikation erstellt, die die korrekte Funktion der partiellen Rekonfiguration und die Auslagerung der Rekonfigurationsmechanismen durch den Treiber verifiziert (Quellcode siehe Anhang A). Es bestehen folgende Unterschiede zur eingebetteten Testapplikation aus Kapitel 4.4:

- **Applikationsgröße:** Linux: 44 KB, Eingebettet: 51 KB
  Die Linux-basierte Testapplikation weist eine geringere Größe auf, da die Rekonfigurationsmechanismen in den ICAP-Treiber ausgelagert sind und durch diesen ausgeführt werden.

- **Speicherort der Applikation:** Linux: Dateisystem, Eingebettet: Lokaler Speicher (BRAM)
  Die eingebettete Applikation wird als stand-alone Programm direkt vom lokalen Speicher des µBlaze-Systems automatisch ausgeführt. Die Linux-basierte Applikation wird als ausführbare Datei im Dateisystem gespeichert und als eigenständiger Prozess nach Aktivierung ausgeführt.
- **Speicherort der partiellen Bitstreams:** Linux: Dateisystem, Eingebettet: CF-Speicher
  Bei der Linux-Applikation werden die partiellen Bitstreams vor der Generierung des Dateisystems integriert. Über Dateioperationen werden diese vom Dateisystem gelesen.
- **HW-Initialisierung:** Linux: Treiber, Eingebettet: Applikation
  Die Initialisierung des HWICAP-Moduls wird im Linux-System bereits während der Boot-Phase durchgeführt. Die eingebettete Applikation muss die Initialisierung vor der Verwendung des Moduls anhand einer Initialisierungsroutine vornehmen.

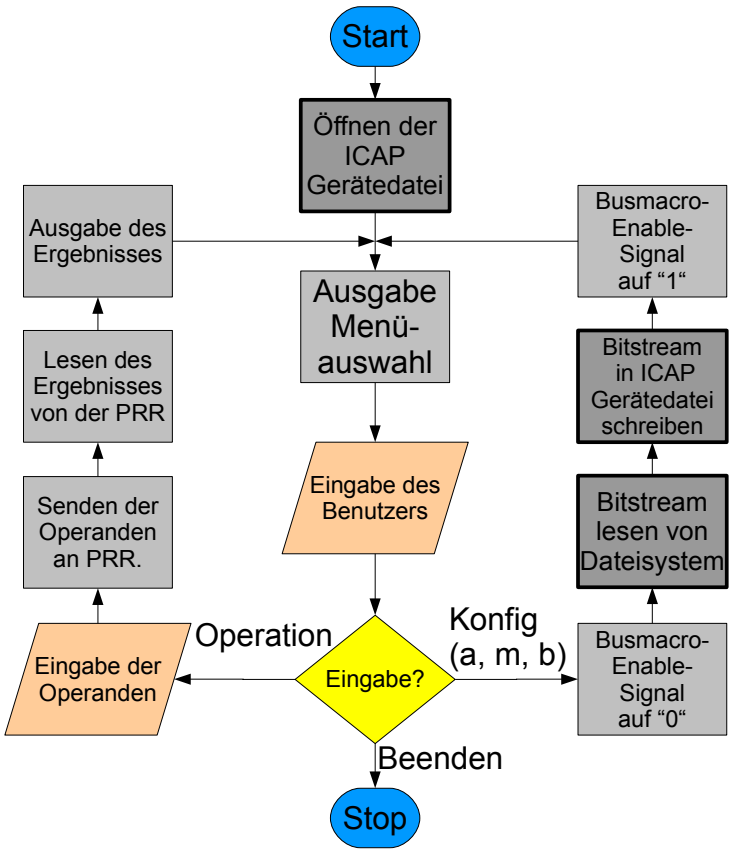


Abbildung 5.7.: Flussdiagramm der Linux-basierten Testapplikation. Die ICAP-Gerätedatei wird geöffnet; die part. Bitstreams werden vom Dateisystem gelesen; Bitstreams werden in die Gerätedatei geschrieben.

Folgende Unterschiede weist der Ablauf der Linux-basierten Testapplikation gegenüber der eingebetteten Testapplikation auf (vgl. Abbildung 5.7):

1. Anstelle der Initialisierung der HW-Module wird die ICAP-Gerätedatei geöffnet (vgl. Quelltext 5.12).

```
1 // ICAP_FILE = /dev/icap, zum Schreiben und Lesen öffnen (r+)
2 if ((icap = fopen(ICAP_FILE, "r+")) == NULL) {
3     perror("failed to open icap:");
4     return -1;
5 }
```

Quelltext 5.12: Die ICAP-Gerätedatei wird über die Funktion fopen() geöffnet.

fopen() ruft intern die open() Funktion des ICAP-Treibers auf.

2. Die part. Bitstreams werden vom Dateisystem ausgelesen (vgl. Quelltext 5.13).

```
1 // fileName = (/usr/add.bit, /usr/mult.bit, /usr/blank.bit)
2 // nur zum Lesen
3 if((partialFile = fopen(fileName, "r")) == NULL) {
4     printf("Datei %s konnte nicht geoeffnet werden\n", fileName);
5     return NULL;
6 }
7 ...
8 // das part. Bitstream wird in buf gepuffert.
9 fread(buf, sizeof(*buf), fileSize, partialFile);
```

Quelltext 5.13: Mit fopen() wird ein part. Bitstream geöffnet und anschließend mit fread() gelesen.

3. Die part. Bitstreams werden in die ICAP-Gerätedatei geschrieben (vgl. Quelltext 5.14).

```
1 // das Bitstream wird ohne Header
2 // in die ICAP-Gerätedatei geschrieben.
3 status = fwrite(&buf[bitHeader.headerLength], sizeof(*buf),
4                                 bitHeader.bitstreamLength, icap);
```

Quelltext 5.14: Das Senden der Bitstreamdaten an HWICAP erfolgt über das Beschreiben der ICAP-Gerätedatei mittels fwrite().

fwrite() ruft intern die write()-Funktion des ICAP-Treibers auf. Die Bitstreamdaten werden über diese write()-Funktion an den Treiber übergeben.

4. Anstelle der Xilinx IO-Funktionen werden Linux IO-Funktionen zum Lesen und Beschreiben der Register der PRMs und zum Steuern der Busmacros verwendet (vgl. Quelltext 5.15).

```
1 #include <asm/io.h>
2 // setzen des Enable Signals für die Busmacros
3 // BM_ENABLE = 0x00000001
4 out_be32((int*)XPAR_XPS_DCR_SOCKET_0_DCR_BASEADDR, BM_ENABLE);
5 // lesen des Operationsergebnisses
6 result = in_be32((int*)XPAR_XPS_DCR_SOCKET_0_BASEADDR);
```

Quelltext 5.15: Lesen und Beschreiben der Register erfolgt über out_be32()- und in_be32()-Funktionen.

Die Funktionsweise der partiellen Rekonfiguration wurde durch mehrmaliges Austauschen der add.bit, mult.bit und blank.bit getestet. Der Aufruf der Funktionen fopen() und fwrite() aus der Testapplikation beweist die Auslagerung der Rekonfigurationsmechanismen in den ICAP-Treiber.

6. Entwurfsablauf für selbstrekonfigurierende SoCs

In diesem Kapitel werden zwei Varianten des Entwurfablaufs zur Erstellung eines selbstrekonfigurierenden µBlaze-Systems behandelt. Präsentationsmaterial zur partiellen Rekonfiguration [41] für die Schulung von Xilinx Field Application Engineers wurde durchdrungen, erprobt, geprüft und in einem kompletten Entwurfsverfahren für die Nutzung in Projekten der HAW Hamburg aufbereitet. Dieses Verfahren für partielle Entwürfe lässt sich in zwei Varianten anwenden, die sich in der Hierarchie der Entwurfsdateien unterscheiden (vgl. Kapitel 3.1.1):

- Die PRR und das µBlaze-System sind Komponenten einer Hierarchiebene und werden von einer Top-Entity instanziiert (Abbildung 6.1 links).
- Die PRR ist Bestandteil des µBlaze-Systems (Abbildung 6.1 rechts).

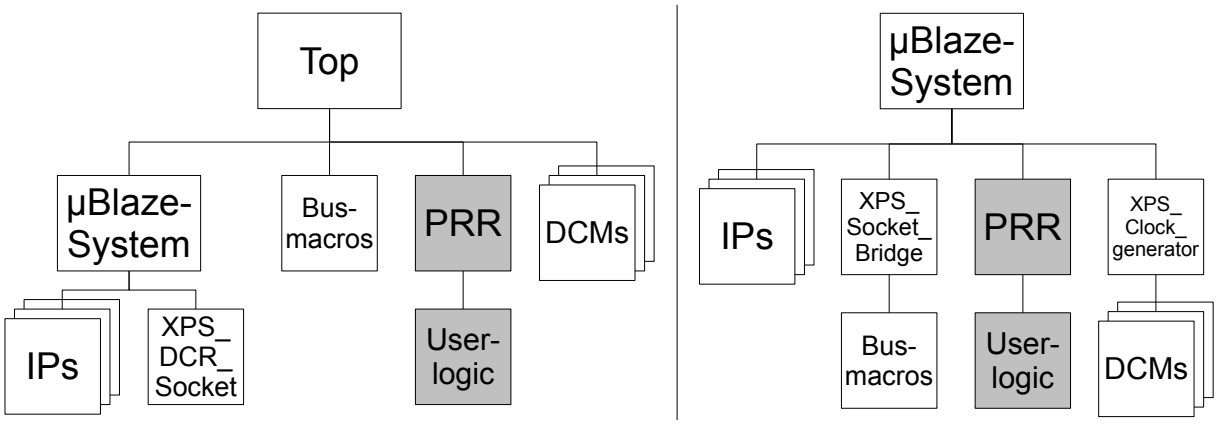


Abbildung 6.1.: Entwurfshierarchie eines selbstrekonfigurierenden µBlaze-Systems; Links: PRR und µBlaze-System in derselben Ebene, Rechts: PRR Bestandteil des µBlaze-Systems

Das resultierende System aus beiden Entwurfsabläufen weist dieselbe Funktionalität und dasselbe Verhalten auf. Je nach angewandter Hierarchie ergeben sich folgende Randbedingungen für den jeweiligen Entwurfsablauf:

- Die einzelnen Entwurfsschritte unterscheiden sich in Anzahl, Art und Reihenfolge.
- Aus den einzelnen Schritten resultieren unterschiedliche Dateien.
- Für einen jeweiligen Entwurfsschritt werden unterschiedliche Eingangsdateien verwendet.

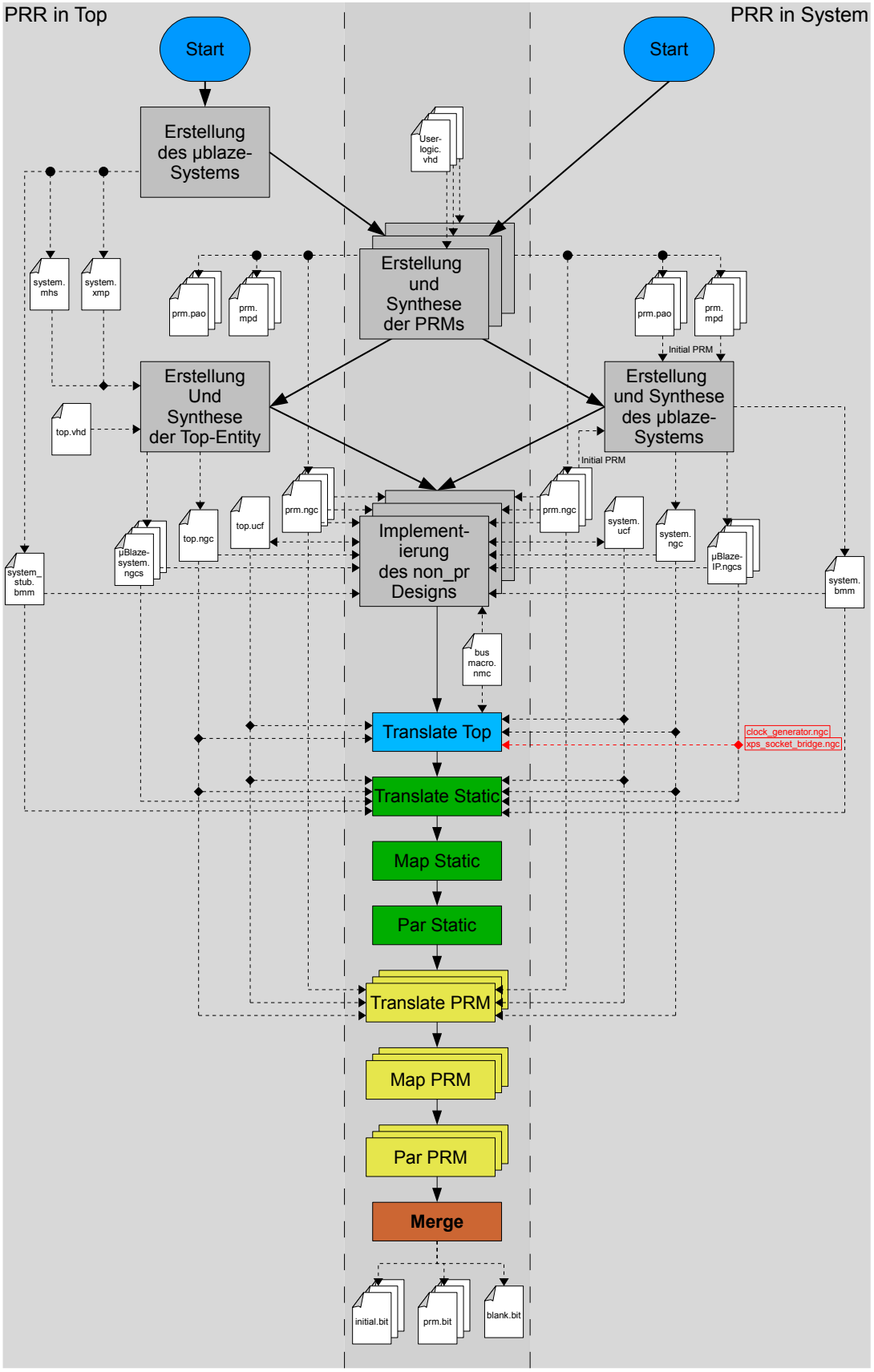


Abbildung 6.2.: Entwurfsflow eines selbstrekonfigurierenden µBlaze-Systems

In Abbildung 6.2 ist der gesamte Ablauf für beide Varianten dargestellt. Links sind die einzelnen Schritte für die Erstellung eines Systems mit der PRR als Teil der Top-Entity und rechts als Teil des µBlaze-Systems abgebildet. Die in der mitte gezeigten Schritte werden für beide Varianten gleichermaßen ausgeführt. Es kommen lediglich unterschiedliche Eingangsdaten für diese Schritte zum Einsatz.

Tabelle 6.1 gibt einen Überblick über die einzelnen Schritte der jeweiligen Entwurfsabläufe. Diese Schritte werden in Abschnitt 6.3 detailliert erläutert.

Nr.	PRR in Top	Analog	PRR in System	Nr.
1	Erstellung des µBlaze-Systems			
2		Erstellung und Synthese der PRMs		1
3	Erstellung und Synthese der Top-Entity		Erstellung und Synthese des µBlaze-Systems	2
4		Implementierung von nicht partiellen Designs		3
5		Übersetzen des Top-Designs		4
6		Implementieren des Static-Designs		5
7		Implementieren der PRMs		6
8		Merge und Bitstreamgenerierung		7

Tabelle 6.1.: Einzelne Schritte der beiden Entwurfsabläufe für ein selbstrekonfigurierendes SoC

6.1. Entwicklungswerkzeuge

Für einen partiell rekonfigurierbaren Entwurf mit Xilinx FPGAs werden folgende Xilinx Werkzeuge eingesetzt (siehe Abbildung 6.3):

- **ISE 9.2i:** Dieses Tool bildet die komplette Implementierungsumgebung mit Synthese, Mapping und Place and Route zur Erstellung von Xilinx FPGA-Konfigurationsdateien aus RTL-Entwürfen.
- **Service Pack 4 für ISE 9.2i:** Dieses Update ist die letzte Aktualisierung für ISE 9.2i.
- **PR-Patch:** Dieser Patch ist eine Anpassung von ISE 9.2i SP4 für die Durchführung von PR-Entwürfen.
- **EDK:** Das Embedded Development Kit EDK 9.2i liefert den Werkzeugsatz zur Konfiguration und Generierung von FPGA-basierten µControllern mit anwenderspezifischen Beschleuniger-IPs.

- **Floorplanner:** In diesem Entwurfsablauf dient der Floorplaner zur Platzierungsanalyse. Die Platzierungen der Busamcros und DCMs lassen sich mit diesem Werkzeug auf grafischer Ebene feststellen.

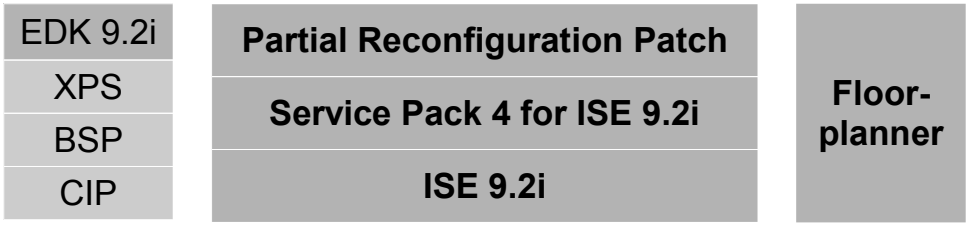


Abbildung 6.3.: Tools für PR von µBlaze-Systemen

Die PR-Technologie befindet sich noch in der Entwicklungsphase, sodass die Xilinx Werkzeuge zur partiellen Rekonfiguration gering erprobt sind [41]. Die Implementierung eines partiell rekonfigurierbaren Entwurfs mit Xilinx Werkzeugen ist noch nicht vollständig in die Xilinx Werkzeugkette integriert. Mit der zusätzlichen Anpassung dieser Werkzeugkette durch den PR-Patch lassen sich PR-Entwürfe implementieren. Dieser PR-Patch wird für ISE 9.2i mit SP4 zur Verfügung gestellt. Demnach bezieht sich die Beschreibung des Entwurfablaufs in dieser Abeit auf ISE in der Version 9.2i(zum Zeitpunkt der Erstellung dieser Arbeit war bereits ISE v11 erhältlich).

6.2. Empfohlene Verzeichnisstruktur

Zur Realisierung eines partiellen Entwurfs ist das Anlegen einer geeigneten Verzeichnisstruktur aus folgenden Gründen empfehlenswert:

- **Unterscheidung zwischen den Produkten der Synthese und der Implementierungen des Entwurfs.**
  Während der Entwurf einmal synthetisiert wird, entstehen während des Entwurfablaufs mehrere Implementierungen. Die verschiedenen Implementierungen erfordern jeweils eine Teilmenge der generierten Netzlisten des Systems. Die Unterscheidung zwischen Synthese- und Implementierungsverzeichnissen erspart wiederholende Schritte der Synthese für jede Implementierung und verringert die Redundanz.
- **Unterscheidung zwischen den verschiedenen Implementierungen.**
  Teil des Entwurfablaufs ist die Implementierung vom statischen Entwurf sowie der einzelnen PRMs. Zusätzlich wird zur Analyse des Timings und der Platzierung für jede Kombination aus statischem Entwurf und PRM eine nicht partielle Implementierung erstellt. Durch die Unterteilung dieser Implementierungen wird eine Verwechslung der jeweiligen Ein- und Ausgangsdateien verhindert.
- **Namentliche Unterscheidung der einzelnen PRMs.**
  Sämtliche Entwurfsdateien der PRMs für eine PRR, die in den einzelnen Entwurfsschritten produziert werden, besitzen dieselben Bezeichnungen. Einzig die Bitstreams werden unterschiedlich bezeichnet und lassen sich somit im letzten Entwurfsschritt unterscheiden. Für die einzelnen Entwurfsschritte lassen sich die PRM-Produkte lediglich durch unterschiedliche Verzeichnisse unterscheiden.

Mit der Unterteilung dieser Entwurfsdateien in mehrere Verzeichnisse lässt sich die Komplexität dieses Entwurfsablaufs überblicken. Folgende Verzeichnisstruktur wird für den Entwurf eines µBlaze-basierenden, selbstrekonfigurierenden SoCs empfohlen [34]:

- **Busmacros:** Ablage der NMC-Dateien für die Busmacros.
- **synth:** In diesem Verzeichnis werden alle generierten Netzlisten abgelegt.
  - **...\edk:** Das mit XPS erstellte µBlaze-System wird hier mit allen Bestandteilen eingeordnet.
  - **...\prm:** Die Netzlisten der jeweiligen PRMs werden hier in dedizierten Unterverzeichnissen unterteilt.
  - **...\top:** Ist die PRR nicht Bestandteil des µBlaze-Systems, wird in diesem Verzeichnis die Netzliste der Top-Entity sowie aller Netzlisten des statischen Entwurfs generiert.
- **non_pr:** Für jede Kombination aus statischem Entwurf und PRM wird hier im jeweiligen Unterverzeichnis die Implementierung erstellt.
- **top:** Die Netzliste der Top-Entity durchläuft in diesem Verzeichnis die Translate-Phase. Statische Module sowie PRMs werden in dieser Phase zunächst als Blackbox instanziiert.
- **static:** Zusammen mit der Top-Entity wird der statische Entwurf hier implementiert.
- **prm:** Zusammen mit der Top-Entity werden die einzelnen PRMs im jeweilingen Unterverzeichnis implementiert.
- **merges:** Die Bitstreams der jeweiligen Implementierungen werden hier generiert. In Unterverzeichnissen werden die PRMs mit der statischen Implementierung zusammengefügt.

6.3. Entwurfsschritte

6.3.1. Erstellung des µBlaze-Systems

In diesem Schritt wird ein µBlaze-System mit XPS zusammengestellt (vgl. Abbildung 6.2, Tabelle 6.1 Schritt 1). Hier werden die einzelnen Vorgänge und Entwurfsdateien (vgl. Tabelle 6.2) zur Erstellung des partiellen Entwurfs genannt (vgl. [39] und [40]).

Werkzeuge	• EDK • XPS • BSB
Eingangsdateien	• XPS_DCR_Socket-IP • Xilinx IP-Bibliothek
Ausgangsdateien	• system.xmp • system.mhs • system_stub.bmm

Tabelle 6.2.: Schritt 1 - PRR in Top: Erstellung des µBlaze-Systems

Übersicht über die Einzelschritte zur Erstellung des µBlaze-Systems:

1. Mit XPS wird ein Projekt mit der Bezeichnung “system.xmp“ erstellt.
2. Das Basis-System wird mit BSB konfiguriert.
3. Die clock_generator-Komponente wird entfernt
4. Folgende IPs werden dem System hinzugefügt:
   - XPS_DCR_Socket
   - FPGA Internal Configuration Access Port (HWICAP)
   - Device Control Register (DCR) Bus 2.9
   - PLBv46 to DCR Bridge
5. Diese IPs werden an die jeweiligen Busschnittstellen angeschlossen (Bus Interfaces Tab).
6. Die Adressen dieser IPs werden eingestellt und generiert (Adresses Tab).
7. Die Signale des XPS_DCR_Sockets werden in der MHS-Datei hinzugefügt und über externe Signale des µBlaze-Systems nach Außen geführt
8. Eine eingebettete SW-Applikation wird hinzugefügt und durch den SW-Build-Prozess generiert (SW -> Build All User Applications).
9. Das Projekt wird abgespeichert und XPS geschlossen. Die Netzlisten werden in diesem Schritt noch nicht generiert.

Die clock_generator-Komponente wird entfernt
Die clock_generator-Komponente ist eine IP, die die für das µBlaze-System die Taktraten durch Instanziierung von DCMs automatisch generiert. DCMs sind als globale Logik und in der Top-Entity zu instanziieren [34]. Aus diesem Grund wird diese Komponente als Bestandteil des µBlaze-Systems entfernt. Folgende Schritte sind auszuführen:

1. Im “System Assembly View“ mit einem Rechtsclick auf die clock_generator-IP und “Delete Instance“ auswählen.
2. In der erscheinenden Auswahlbox “Delete instance but keep its ports“ auswählen und mit “Ok“ bestätigen.
3. In der “Ports“ Auswahl werden sämtliche Taktsignale des Systems auf “External“ konfiguriert.
4. In der MHS-Datei wird das Signal “dcm_clk_s“ mit “sys_clk_s“ ersetzt.

Die DCMs werden nun nicht mehr als Teil des µBlaze-Systems instanziiert. Diese werden in der Top-Entity erstellt und erzeugen die erforderlichen Taktsignale für das System. Die Taktsignale für das µBlaze-System bleiben erhalten und mit den erzeugten Taktraten über die Top-Entity verbunden.

Die Signale des XPS_DCR_Sockets werden in der MHS-Datei hinzugefügt und über externe Signale des µBlaze-Systems nach Außen geführt
Die XPS_DCR_Socket-IP besitzt eine PLB-Slave-Schnittstelle und eine Schnittstelle, die die PLB-Slave-Signale zur PRR weiterleitet. Der Anschluss des XPS_DCR_Sockets an den PLB wird durch XPS nach der Auswahl im "Bus Interfaces Tab" automatisch durchgeführt. Die Signale des XPS_DCR_Sockets zur PRR müssen über externe Signale des µBlaze-Systems nach Außen zugänglich gemacht werden. Dazu werden in der system.mhs-Datei folgende Zeilen hinzugefügt:

1. Am Anfang der MHS-Datei werden zunächst für die Signale des XPS_DCR_Sockets zur PRR externe Signale definiert (Quelltext 6.1).

```
1 PORT xps_dcr_socket_0_RSl_MIRQ_pin = xps_dcr_socket_0_RSl_MIRQ, DIR = I, VEC = [0:1]
2 PORT xps_dcr_socket_0_RSl_MRdErr_pin = xps_dcr_socket_0_RSl_MRdErr, DIR = I, VEC = [0:1]
3 PORT xps_dcr_socket_0_RSl_MWrErr_pin = xps_dcr_socket_0_RSl_MWrErr, DIR = I, VEC = [0:1]
4 ...
5 PORT xps_dcr_socket_0_RSPLB_Rst_pin = xps_dcr_socket_0_RSPLB_Rst, DIR = O, SIGIS = RST
6 PORT xps_dcr_socket_0_RSPLB_Clk_pin = xps_dcr_socket_0_RSPLB_Clk, DIR = O, SIGIS = CLK
7 PORT xps_dcr_socket_0_RPLB_BM_enable_pin = xps_dcr_socket_0_RPLB_BM_enable, DIR = O
```

Quelltext 6.1: Definition von externen Signalen für die PRR

2. Im Bereich der Komponentendefinition des XPS_DCR_Sockets in der MHS-Datei werden die Signale dieser Komponente zur PRR mit den externen Signalen verbunden (Quelltext 6.2).

```
1  BEGIN xps_dcr_socket
2   PARAMETER INSTANCE = xps_dcr_socket_0
3   ...
4   PORT RSl_MIRQ = xps_dcr_socket_0_RSl_MIRQ
5   PORT RSl_MRdErr = xps_dcr_socket_0_RSl_MRdErr
6   PORT RSl_MWrErr = xps_dcr_socket_0_RSl_MWrErr
7   ...
8   PORT RSPLB_Rst = xps_dcr_socket_0_RSPLB_Rst
9   PORT RSPLB_Clk = xps_dcr_socket_0_RSPLB_Clk
10  PORT RPLB_BM_enable = xps_dcr_socket_0_RPLB_BM_enable
11  END
```

Quelltext 6.2: Anschluss der XPS_DCR_Socket Signale zur PRR mit externen Signalen

Die Netzlisten dieses Systems werden nicht in diesem Schritt generiert. Die Dateien system.xmp und system.mhs werden in den nächsten Schritten zur Generierung der Netzlisten verwendet. Die system_stub.bmm-Datei enthält Informationen über den Datenbereich und die Organisation des verwendeten Block-RAMs. Diese Entwurfsdatei wird zur Implementierung des statischen Entwurfs eingesetzt.

6.3.2. Erstellung und Synthese der PRMs

In diesem Schritt werden die PRMs mit dem Xilinx CIP-Wizard konfiguriert und mit ISE synthetisiert (vgl. Abbildung 6.2, Tabelle 6.1 Schritt 2). Die einzelnen Vorgänge und Entwurfsdateien (vgl. Tabelle 6.3) zur Erstellung der PRMs werden genannt (vgl. [39], [40]).

Übersicht über die Einzelschritte zur Erstellung einer PRM:

Werkzeuge	• EDK • CIP • ISE
Eingangsdateien	• user_logic.vhd(1...N)
Ausgangsdateien	• prm.ngc(1...N) • prm.pao(1...N) • prm.mpd(1...N)

Tabelle 6.3.: Schritt 2 - Analog: Erstellung und Synthese der PRMs

1. Mit CIP wird zunächst eine IP vorkonfiguriert. Die einzelnen PRMs besitzen die gleichen Namen und unterscheiden sich in der Versionsnummer. Das SW-Interface wird durch SW-Register und deren Adressen spezifiziert. Die PRMs werden in der User-Repository abgespeichert, die zu Beginn des CIP-Dialogs angegeben wird.
2. In der vom CIP erstellten user_logic.vhd wird die VHDL-Beschreibung der Funktionalität der jeweiligen PRM eingefügt.
3. Anpassung der vom CIP erstellten ISE-Projektdateien für die PRMs
4. Synthese der einzelnen PRMs

Anpassung der vom CIP erstellten ISE-Projektdateien für die PRMs

Der hier beschriebene Entwurfsablauf beruht auf Xilinx Werkzeugen in der Version 9.2.2. In dieser Version des EDK werden die Projektdateien für ISE nicht korrekt erstellt. Die vom CIP erstellten Projektdateien sind für die erfolgreiche Synthese mit ISE anzupassen. Folgende Schritte sind zur Anpassung durchzuführen:

1. Die CLI-Datei in ".../*prm_name*/devl/projnav" wird an das jeweilige FPGA angepasst. Quelltext 6.3 zeigt die Veränderungen für ein Virtex 5 FPGA.

```
SetProperty(Device Family, virtex5)
SetProperty(Device, xc5vlx50t)
SetProperty(Package, ff1136)
SetProperty(Speed Grade, -1)
```

Quelltext 6.3: Anpassung der CLI-Datei für ein Virtex 5 FPGA

2. In der Kommandozeile wird zunächst in das jeweilige PRM-Verzeichnis gewechselt (".../*prm_name*/devl/projnav") und anschließend folgendes ausgeführt:

pjcli.bat -f *prm_name*.cli

Nach dieser Prozedur lässt sich das Projekt der jeweiligen PRM mit ISE öffnen und editieren.

Synthese der einzelnen PRMs

- Die PLB-Schnittstelle der PRMs ist dem µBlaze-System anzupassen (vgl. 4.3), bevor die Synthese mit ISE angestoßen wird (vgl. Quelltext 6.4).

```
entity math is
generic
(
  C_BASEADDR             : std_logic_vector := X"FFFFFFFF";
  C_HIGHADDR             : std_logic_vector := X"00000000";
  C_SPLB_AWIDTH          : integer          := 32;
  C_SPLB_DWIDTH          : integer          := 32;
  C_SPLB_NUM_MASTERS     : integer          := 2;
  C_SPLB_MID_WIDTH       : integer          := 1;
  C_SPLB_NATIVE_DWIDTH   : integer          := 32;
  C_SPLB_P2P             : integer          := 0;
  C_SPLB_SUPPORT_BURSTS  : integer          := 0;
  C_SPLB_SMALLEST_MASTER : integer          := 32;
  C_SPLB_CLK_PERIOD_PS   : integer          := 10000;
  C_FAMILY               : string           := "virtex5"
);
```

Quelltext 6.4: Generic-Definition der PLB-Schnittstelle

Durch das Editieren dieser Passage wird die PLB-Schnittstelle der PRM automatisch an den PLB des µBlaze-Systems angepasst. Nach diesem Schritt wird die PRM mit ISE synthetisiert.

6.3.3. Erstellung und Synthese des Gesamtsystems

In diesem Schritt wird das Gesamtsystem zusammengestellt und synthetisiert (vgl. Abb. 6.2, Tabelle 6.1 Schritt 3). Für die PRR als Bestandteil einer gesonderten Top-Entity wird die Zusammenstellung über eine top.vhd und die Synthese über ISE gezeigt (vgl. Tabelle 6.4). Die Zusammenstellung und Synthese eines µBlaze-Systems mit PRR als Bestandteil wird über XPS veranschaulicht (vgl. Tabelle 6.5).

6.3.3.1. Erstellung und Synthese der Top-Entity

Werkzeuge	• ISE
Eingangsdateien	• top.vhd • system.xmp • (system.mhs)

Ausgangsdateien	• top.ngc
	• µBlaze-
	  • system.ngc
	  • IPs.ngc (1...N)

Tabelle 6.4.: Schritt 3.1 - PRR in Top: Erstellung und Synthese der Top-Entity

In diesem Vorgang werden folgende Systemelemente in der top.vhd instanziiert und miteinander verbunden:

- Das im Schritt “Erstellung des µBlaze-Systems“ erstellte µBlaze-System

- PRR
- Busmacros
- DCMs

Übersicht über die Einzelschritte zur Erstellung einer PRM:

1. Die top.vhd wird mit der Instanziierung sämtlicher Komponenten erstellt.
2. Es wird ein neues ISE-Projekt erstellt mit top.vhd und system.xmp als existierende Quelldateien.
3. Das System wird synthetisiert.

Das in Schritt "Erstellung des µBlaze-Systems" erstellte µBlaze-System wird über die XPS Projektdatei system.xmp eingebunden. Diese Projektdatei wiederum beinhaltet die system.mhs-Datei mit der Instanziierung des µBlaze-Prozessors und den ausgewählten IPs. Die einzelnen Komponenten werden in diesem Vorgang synthetisiert und ihre Netzlisten als NGC-Dateien generiert. Die Netzlisten der PRMs werden nicht erneut generiert, da diese bereits in Schritt Erstellung und Synthese der PRMs erstellt worden sind.

6.3.3.2. Erstellung und Synthese des µBlaze-Systems

In diesem Schritt wird ein µBlaze-System mit XPS zusammengestellt und synthetisiert. Hier werden die einzelnen Vorgänge genannt, die bei der Erstellung des partiellen Entwurfs durchzuführen sind. Die einzelnen Schritte zur Erstellung eines µBlaze-Systems sind [39] und [40] zu entnehmen.

Werkzeuge	• EDK • XPS • BSB
Eingangsdateien	• prm.ngc, .pao, .mpd (initial) • XPS_Socket_Bridge-IP • Xilinx IP-Bibliothek
Ausgangsdateien	• system.bmm • µBlaze- • system.ngc • IPs.ngc (1...N)

Tabelle 6.5.: Schritt 3.2 - PRR in System: Erstellung und Synthese des µBlaze-Systems

Übersicht über die Einzelschritte zur Erstellung des µBlaze-Systems:

1. Mit XPS wird ein Projekt mit der Bezeichnung "system.xmp" erstellt.
2. Das Basis-System wird mit BSB konfiguriert.
3. Folgende IPs werden dem System hinzugefügt:
   - Initial-PRM
   - XPS_Socket_Bridge
   - FPGA Internal Configuration Access Port (HWICAP)

- Device Control Register (DCR) Bus 2.9
- PLBv46 to DCR Bridge

4. Außer der PRM werden alle IPs an die jeweiligen Busschnittstellen angeschlossen (Bus Interfaces Tab).
5. Die Adressen dieser IPs werden konfiguriert und generiert (Adresses Tab).
6. Die Signale der XPS_Socket_Bridge werden in der MHS-Datei mit den Signalen der PRR verbunden
7. Eine eingebettete SW-Applikation wird hinzugefügt und durch den Build-Prozess generiert.
8. Für das HW-Projekt werden mit XPS die Netzlisten generiert.

Die Signale der XPS_Socket_Bridge werden in der MHS-Datei mit den Signalen der PRR verbunden

Die XPS_Socket_Bridge-IP besitzt eine PLB-Slave-Schnittstelle und eine Schnittstelle, die die PLB-Slave-Signale zur PRR weiterleitet. Der Anschluss des XPS_Socket_Bridge an den PLB wird durch XPS nach der Auswahl im “Bus Interfaces Tab“ automatisch durchgeführt. Die Signale der XPS_Socket_Bridge zur PRR werden direkt über XPS miteinander verbunden. Dazu werden in der system.mhs-Datei folgende Zeilen hinzugefügt:

1. Die Signale der XPS_Socket_Bridge zur PRR werden mit internen Signalen verbunden (Quelltext 6.5).

```
BEGIN xps_socket_bridge
 PARAMETER INSTANCE = xps_socket_bridge_0
 ...
 PORT RSl_MIRQ = xps_socket_bridge_0_RSl_MIRQ
 PORT RSl_MRdErr = xps_socket_bridge_0_RSl_MRdErr
 PORT RSl_MWrErr = xps_socket_bridge_0_RSl_MWrErr
 ...
 PORT RPLB_ABus = xps_socket_bridge_0_RPLB_ABus
 PORT RSPLB_Rst = xps_socket_bridge_0_RSPLB_Rst
 PORT RSPLB_Clk = xps_socket_bridge_0_RSPLB_Clk
END
```

Quelltext 6.5: Verbinden der XPS_Socket_Bridge-Signale mit internen Signalen (vgl. Abbildung 3.7)

2. Die Signale der PRR werden über die internen Signale mit denen der XPS_Socket_Bridge verbunden (Quelltext 6.6).

```
BEGIN xps_prr
 PARAMETER INSTANCE = xps_prr_0
 PARAMETER HW_VER = 1.00.a
 PORT Sl_MIRQ = xps_socket_bridge_0_RSl_MIRQ
 PORT Sl_MRdErr = xps_socket_bridge_0_RSl_MRdErr
 PORT Sl_MWrErr = xps_socket_bridge_0_RSl_MWrErr
 ...
 PORT PLB_ABus = xps_socket_bridge_0_RPLB_ABus
 PORT SPLB_Rst = xps_socket_bridge_0_RSPLB_Rst
 PORT SPLB_Clk = xps_socket_bridge_0_RSPLB_Clk
END
```

Quelltext 6.6: Anschluss der PRR-Signale mit Signalen der XPS_Socket_Bridge

Die PRMs besitzen eine PLB-Schnittstelle, die aber in der MPD-Datei nicht spezifiziert wird. Damit wird verhindert, dass die PRR in diesem Schritt direkt an den PLB angeschlossen wird. Allerdings ist die manuelle Anpassung der PLB-Schnittstelle der PRR über die Generics der PRR erforderlich. In diesem Schritt werden die Netzlisten sämtlicher Module dieses Systems generiert. Die system.bmm-Datei wird zur Implementierung des statischen Entwurfs eingesetzt.

6.3.4. Implementierung von nicht partiellen Designs

In diesem Schritt wird je eine Kombination aus einem statischen Teil des PR-Systems und einem PRM direkt gekoppelt implementiert (vgl. Abb. 6.2, Tabelle 6.1 Schritt 4). Dieser Schritt ist für folgende Entwurfsinhalte zwingend erforderlich (Entwurfsdateien vgl. Tabellen 6.6):

- **Fehlerbeseitigung:** Die korrekte Funktion des statischen Teils lässt sich mit jeder PRM verifizieren und mögliche Fehler beseitigen.
- **Timing- und Platzierungsanalyse:** Die Einhaltung der Timing- und Platzierungsbeschränkungen lassen sich für jede PRM prüfen.
- **PRR-Bereichsanlyse:** Der von der PRR verwendete Bereich des FPGAs lässt sich optimieren.
- **Positionierung der Busmacros:** Die geeignete Position der Busmacros in der PRR wird festgelegt.
- **Positionierung der DCMs und BUFGs:** Durch die automatische Platzierung dieser Komponenten durch ISE lässt sich die geeignete Position für den PR-Entwurf übernehmen.

PRR in Top	
Werkzeuge	• ISE

Eingangsdateien	• top.ucf
	• busmacros.nmc
	• top.ngc
	• prm.ngc (1...N)
	• µBlaze-
	  • system.ngc
	  • IPs.ngc (1...M)
	  • system_stub.bmm

Ausgangsdateien	• top_prm(1...N).bit • top.ucf

PRR in System	
Werkzeuge	• ISE

Eingangsdateien	• system.ucf
	• busmacros.nmc
	• prm.ngc (1...N)
	• µBlaze-
	  • system.ngc
	  • IPs.ngc (1...M)
	  • system.bmm

Ausgangsdateien	• system_prm(1...N).bit • system.ucf

Tabelle 6.6.: Schritt 4 - Analog: Implementierung des non_pr Designs

Übersicht über die Einzelschritte zur Implementierung eines non_pr-Entwurfs:

1. Für jede Kombination wird ein neues ISE-Projekt erstellt. Je nach angewandtem Entwurfsablauf werden die jeweiligen Eingangsdateien aus den Tabellen 6.6 verwendet.
2. Für die jeweilige Kombination wird die Netzliste der zugehörigen PRM hinzugefügt.
3. Die UCF-Datei wird um Platzierungs- und Bereichsbeschränkungen erweitert.
4. Das System wird mit ISE implementiert:

a) Translate (ngdbuild)

b) Mapping

c) Place and Route

d) Bitstreamgenerierung (Bitgen)

5. Das Einhalten der Timings sowie des korrekten Verhaltens des Systems wird mit den üblichen Methoden verifiziert.

Die UCF-Datei wird um Platzierungs- und Bereichsbeschränkungen erweitert
Die Implementierung jeder Kombination aus statischem Teil und einer PRM liefert den Ressourcenbedarf. Das "RANGE"-Constraint der PRR wird durch die PRM bestimmt, die den höchsten Ressourcenbedarf aufweist. Somit bietet der von der PRR verwendete Bereich des FPGAs ausreichend Ressourcen für alle PRMs.

Sind sämtliche Timingconstraints eingehalten worden, wird die Platzierung der Busmacros für den partiellen Entwurf übernommen. Andernfalls ist die Implementierung mit neuer Platzierung der Busmacros zu wiederholen.

Durch die Implementierung werden die DCMs und BUFGs automatisch platziert. Diese Platzierung ist für den partiellen Entwurf in die UCF-Datei zu übernehmen. Zusätzlich zu den üblichen Timingconstraints und Platzierungsconstraints für I/O-Ressourcen des FPGAs sind folgende Constraints für einen partiell rekonfigurierbaren Entwurf hinzuzufügen:

- **AREA_GROUP:** Mit diesem Constraint werden logische Entwurfselemente einer PRR gruppiert und mit einer Kennzeichnung versehen. Die PRR wird somit vom statischen Teil des System abgegrenzt.
- **AREA_GROUP RANGE:** Dieses Constraint legt den Bereich und die Größe der PRR im FPGA fest. Es wird die Position und Anzahl jeder verwendeten FPGA-Ressource spezifiziert.
- **MODE = RECONFIG:** Zur Erkennung der PRR als rekonfigurierbares Modul während der Translate-Phase wird dieses Constraint für jede PRR gesetzt.
- **LOC:** Für einen PR-Entwurf sind DCMs, BUFGs sowie Busmacros mit diesem Constraint zu platzieren [34].

Für den non_pr-Entwurf werden die oben genannten Constraints folgendermaßen gesetzt:

1. Die PRR-Instanz wird mit AREA_GROUP gekennzeichnet:

   INST "PRR" AREA_GROUP = "pblock_PRR";

2. Der von der PRR verwendete Bereich des FPGAs wird initial festgelegt:

   Verwendete Slice-Ressourcen:

   AREA_GROUP "pblock_PRR" RANGE=SLICE_X10Y70:SLICE_X19Y109;

   Verwendete DSP-Ressourcen:

AREA_GROUP “pblock_PRR“ RANGE=DSP48_X0Y28:DSP48_X0Y43;

Für jeden weiteren verwendeten Ressourcentyp erfolgt der entsprechende Eintrag.

3. Anstelle des '"MODE = RECONFIG“ Constraints wird hier folgender Eintrag verwendet:

   AREA_GROUP “pblock_PRR“ GROUP = CLOSED;

   Dieses Constraint verhindert, dass in einem non_pr-Entwurf Elemente des statischen Teils im Bereich der PRR platziert werden.

4. Für die Busmacros werden hier initial Platzierungsconstraints mit “LOC“ gesetzt:

   INST “signal1_BM“ LOC = SLICE_X13Y100;
   ...

6.3.5. Übersetzen des Top-Designs

In diesem Schritt wird die Top-level Netzliste in das Xilinx NGD-Dateiformat übersetzt (vgl. Abb. 6.2, Tabelle 6.1 Schritt 5). Der Top-Level-Entwurf enthält folgende Komponenten (Entwurfsdateien vgl. Tabelle 6.7):

- Sämtliche globale Logik (DCMs, BUFGs, ...)
- Externe Schnittstellen
- Instanziierung der Busmacros
- Alle als “Black-box“ instanziierten Entwurfsmodule
- Sämtliche Signale zum Verbund der Module untereinander
- Signale zum Anschluss der Module an die externen Schnittstellen

In dieser Phase werden sämtliche o.g. Elemente positioniert. Die erfolgreiche Durchführung dieser Übersetzung stellt sicher, dass diese Logikelemente korrekt positioniert sind. Die Repositionierung während der Implementierung des statischen Teils und der PRMs ist dann nicht mehr erforderlich.

PRR in Top	
Werkzeuge	• Ngdbuild
Eingangsdateien	• top.ngc • top.ucf • busmacros.nmc
Ausgangsdateien	• top.ngd

PRR in System	
Werkzeuge	• Ngdbuild
Eingangsdateien	• system.ngc • system.ucf • busmacros.nmc • clock_generator.ngc • xps_socket_bridge.ngc
Ausgangsdateien	• system.ngd

Tabelle 6.7.: Schritt 5 - Analog: Übersetzen des Top-Designs

Module, die globale Logik oder Busmacros instanziieren, werden hier hinzugefügt und demnach nicht als “Black-box“ eingebunden. Ist die PRR Bestandteil des µBlaze-Systems, so ist system.ngc die Top-level Netzliste des Entwurfs, andernfalls ist es die top.ngc.

Übersicht über die Einzelschritte zum übersetzen des Top-level-Entwurfs:

1. Die Eingangsdateien aus den Tabellen aus 6.7 werden in den "../top"-Ordner eingefügt.
2. Aktualisierung der UCF-Datei für den partiellen Entwurf.
3. Über die Kommandozeile wird in das ".../top"-Verzeichnis gewechselt.
4. Der Übersetzungsvorgang wird durch folgenden Aufruf in der Kommandozeile ausgelöst:

   Für PRR in Top mit Virtex 5 FPGA:

   ngdbuild -p xc5vlx50t-1-ff1136 -modular initial -uc top.ucf top.ngc top.ngd

   Für PRR als Bestandteil des µBlaze-Systems im Virtex 5 FPGA:

   ngdbuild -p xc5vlx50t-1-ff1136 -modular initial -uc system.ucf system.ngc system.ngd

Aktualisierung der UCF-Datei für den partiellen Entwurf
Für die Implementierung des partiellen Entwurfs sind sämtliche Constraints aus 6.3.4 endgültig zu spezifizieren. Diese werden hier anhand der Ergebnisse aus der Implementierung des non_pr-Entwurfs übernommen. Folgende Eintragungen werden vorgenommen:

- Die AREA_GROUP Kennzeichnung wird gleichbleibend aus dem non_pr-Entwurf übernommen.
- Das AREA_GROUP RANGE Constraint wird dem Ressourcenbedarf der größten PRM angepasst.
- Die PRR wird als rekonfigurierbare Region gekennzeichnet:

  AREA_GROUP "pblock_PRR" MODE = RECONFIG;

- Die Platzierung der Busmacros, der DCMs und BUFGs wird aus dem non_pr-Entwurf übernommen.

6.3.6. Implementieren des Static-Designs

In diesem Schritt wird der statische Teil des Systems implementiert (vgl. Abb. 6.2, Tabelle 6.1 Schritt 6, Entwurfsdateien s. Tabelle 6.8).

PRR in Top	
Werkzeuge	• Ngdbuild • Map • Par

Eingangsdateien	• ../top/top.ngc
	• ../top/top.ucf
	• µBlaze-
	  • system.ngc
	  • IPs.ngc (1...N)
	  • system_stub.bmm

Ausgangsdateien	• top_static_routed.ncd • static.used

PRR in System	
Werkzeuge	• Ngdbuild • Map • Par

Eingangsdateien	• ../top/system.ucf
	• µBlaze-
	  • ../top/system.ngc
	  • IPs.ngc (1...N)
	  • system.bmm

Ausgangsdateien	• system_static_routed.ncd • static.used

Tabelle 6.8.: Schritt 6 - Analog: Implementieren des Static-Designs

Übersicht über die Einzelschritte zur Implementierung des statischen Entwurfs:

1. Die Eingangsdateien aus den Tabellen aus 6.8 außer der Top-level Netzliste und der UCF-Datei werden in den "../static"-Ordner eingefügt.
2. Über die Kommandozeile wird in das "../static"-Verzeichnis gewechselt.
3. Der statische Entwurf wird übersetzt:

   Für PRR in Top mit Virtex 5 FPGA:

   ngdbuild -p xc5vlx50t-1-ff1136 -bm system_stub.bmm -uc ../top/top.ucf -modular initial ../top/top.ngc top.ngd

   Für PRR als Bestandteil des µBlaze-Systems im Virtex 5 FPGA:

   ngdbuild -p xc5vlx50t-1-ff1136 -bm system.bmm -uc ../top/system.ucf -modular initial ../top/system.ngc system.ngd

4. Die logischen Elemente des Designs werden den Ressourcen des FPGAs durch Map zugeordnet:

   map (top/system).ngd

5. Die zugeordneten Entwurfselemente werden durch Par platziert und ihre Leitungswege bestimmt:

   par -w (top/system).ncd (top/system)_static_routed.ncd

Zusätzlich zur (top/system)_static_routed.ncd-Datei wird während dieser Implementierung die static.used-Datei erzeugt. Diese Datei enthält eine Liste von Leitungswegen des statischen Teils, die innerhalb des PRR-Bereichs gesetzt sind. Während der Implementierung der PRMs wird die static.used-Datei eingesetzt, damit keine Leitungswege für die PRMs gesetzt werden, die bereits durch das statische System in Verwendung sind.

6.3.7. Implementieren der PRMs

In diesem Schritt werden die einzelnen PRMs des Systems implementiert (vgl. Abb. 6.2, Tabelle 6.1 Schritt 7, Entwurfsdateien s. Tabelle 6.9).

PRR in Top	
Werkzeuge	• Ngdbuild • Map • Par
Eingangsdateien	• ../top/top.ngc • ../top/top.ucf • prm.ngc (1...N) • static.used
Ausgangsdateien	• prm_routed.ncd (1...N)

PRR in System	
Werkzeuge	• Ngdbuild • Map • Par
Eingangsdateien	• ../top/system.ngc • ../top/system.ucf • prm.ngc (1...N) • static.used
Ausgangsdateien	• prm_routed.ncd (1...N)

Tabelle 6.9.: Schritt 7 - Analog: Implementieren der PRMs

Übersicht über die Einzelschritte zur Implementierung einer PRM:

1. Die Netzlistenbeschreibung (prm.ngc) der PRM sowie die static.used-Datei werden in den jeweiligen "../prm/*prm_name*"-Ordner eingefügt.
2. Die static.used-Datei wird in arcs.exclude umbenannt.
3. Über die Kommandozeile wird in das "../prm/*prm_name*"-Verzeichnis gewechselt.
4. Die PRM wird übersetzt:

   ngdbuild -p xc5vlx50t-1-ff1136 -modular module -uc ../../top/(top/system).ucf -active *prm_name* ../../top/(top/system).ngc

5. Die logischen Elemente der PRM werden den Ressourcen des FPGAs durch Map zugeordnet:

   map (top/system).ngd

6. Die zugeordneten Entwurfselemente werden durch Par platziert und ihre Leitungswege bestimmt:

   par -w (top/system).ncd prm_*prm_name*_routed.ncd

Dieser Ablauf wird für jede PRM des Systems durchgeführt.

6.3.8. Merge und Bitstreamgenerierung

In diesem Schritt werden die folgenden Bitstreams generiert (vgl. Abb. 6.2, Tabelle 6.1 Schritt 8, Entwurfsdateien s. Tabelle 6.10):

- Für jede Kombination aus statischem System und PRM wird ein Initial-Bitstream generiert (vgl. Kapitel 5.1, system_lin.ace).
- Für jede PRM wird ein partielles Bitstream generiert (vgl. Kapitel 5.1, add/mult.bit).
- Für die PRR wird ein Blank-Bitstream ohne Konfigurationsdaten generiert (vgl. Kapitel 5.1, blank.bit).

PRR in Top	
Werkzeuge	• PR_verifydesign • PR_assemble
Eingangsdateien	• top_static_routed.ncd • prm_routed.ncd (1...N) • system_stub.bmm
Ausgangsdateien	• top_full.bit (1...N) • prm_partial.bit (1...N) • prm_blank.bit

PRR in System	
Werkzeuge	• PR_verifydesign • PR_assemble
Eingangsdateien	• system_static_routed.ncd • prm_routed.ncd (1...N) • system.bmm
Ausgangsdateien	• system_full.bit (1...N) • prm_partial.bit (1...N) • prm_blank.bit

Tabelle 6.10.: Schritt 8 - Analog: Merge und Bitstreamgenerierung

Übersicht über die Einzelschritte zur Generierung der Bitstreams:

1. Die Eingangsdateien aus den Tabellen aus 6.10 werden in das jeweilige "../merges/*prm_name*"-Ordner eingefügt.

2. Über die Kommandozeile wird in das "../merges/*prm_name*"-Verzeichnis gewechselt.
3. Die separaten Bitstreams des statischen Systems und einer PRM werden generiert:

   PR_verifydesign top_static_routed.ncd prm_*prm_name*_routed.ncd

4. Ein initiales Bitstream aus der Kombination des Bitstreams des statischen Systems und der PRM wird zusammengefügt. Zusätzlich wird ein Blank-Bitstream ohne Konfiguration für die PRR erstellt:

   PR_assemble top_static_routed.ncd prm_*prm_name*_routed.ncd

Dieser Ablauf wird für jede PRM des Systems durchgeführt.

7. Zusammenfassung

Im Rahmen dieser Arbeit wurde ein FPGA-basiertes SoC aufgebaut, mit dem sich SW-Anwendungen unter Verwaltung eines µCLinux-Kernels durch zur Laufzeit geladene HW-Module beschleunigen lassen. Diese µCLinux-Entwicklungsplattform zur dynamischen partiellen Rekonfiguration besteht aus einem µBlaze-basierten µController-System und der Petalinux Distribution. Dem Linux-Kernel wurde zusätzlich ein angepasster Treiber zur Anbindung der internen Rekonfigurationsschnittstelle des FPGAs (Internal Configuration Access Port) hinzugefügt.

Drei Architekturen für selbstrekonfigurierende µBlaze-Systeme, die sich in ihrer Struktur und der statischen Schnittstelle zur PRR unterscheiden, waren Gegenstand der vergleichenden Untersuchungen:

- Bei einem SoC mit dem XPS_DCR_Socket als Busanbindung zur PRR werden PRR, Busmacros und die DCMs in derselben Hierarchieebene parallel zum µBlaze-System instanziiert.
- Bei der Busanbindung über die XPS_Socket_Bridge lassen sich PRR und die DCMs als direkte Peripherie des µBlaze-Systems und die Busmacros als Komponente der statischen Schnittstelle instanziieren.
- Das Konzept des PRR_IPIC_Sockets wurde in dieser Arbeit erstellt. Eine Busanbindung der PRR über diese Schnittstelle hat den Vorteil, dass sich die Konfigurationsdaten der PRM durch Verlagerung der PLB-Slave-Schnittstelle in den statischen Teil reduzieren lassen.

Basierend auf der SoC-Architektur mit dem XPS_DCR_Socket wurde ein selbstrekonfigurierendes µBlaze-System mit Peripherie-PRMs aufgebaut. Als Erprobungsbeispiele für die Rekonfiguration der PRR dienten zwei PRMs, die eine Addierer- bzw. eine Multiplizierer-Funktionalität beinhalten. Eine entwickelte, eingebettete µBlaze-SW, liest die partiellen Bitstreams aus einem Compact-Flash Speicher aus und übergibt diese zur Rekonfiguration an das XPS-HWICAP-Modul des µBlaze-Systems.

Zur Integration eines µCLinux-Kernels ist das entworfene, selbstrekonfigurierende µBlaze-System um zusätzlichen externe Speicher und Interface-Komponenten zur Kommunikation mit einem PC erweitert worden. Ein Treiber zur Bereitstellung der Rekonfigurationsmechanismen wurde zur Ausführung unter Linux 2.6.31 angepasst, für das Fifo-basierte XPS-HWICAP-Modul des µBlaze-Systems konfiguriert und dem Linux-Kernel hinzugefügt. Die partielle Rekonfiguration wurde mit einer Linux-basierten SW-Anwendung verifiziert.

Dieses µCLinux-System steht nun als Erprobungsplattform für die quasi-parallele Ausführung mehrerer SW-Anwendungen und zur Steuerung der partiellen Rekonfiguration für weiterführende Untersuchungen und Modifikationen zur Verfügung.

Im Weiteren wurde Präsentationsmaterial zur partiellen Rekonfiguration [41], das für die Schulung von Xilinx Field Application Engineers genutzt wird, durchdrungen, erprobt und geprüft und in einer kompletten Entwurfsdokumentation für eine Nutzung in Projekten der HAW Hamburg aufbereitet. Das Entwurfsverfahren für die Erstellung der selbstrekonfigurierenden µBlaze-Systeme mit XPS_DCR_Socket und XPS_Socket_Bridge ist darin detailliert erläutert.

Tabellenverzeichnis

Abbildungsverzeichnis

Quelltextverzeichnis

Abkürzungsverzeichnis

ALU	**A**rithmetic **L**ogic **U**nit
BMM	**B**lockRAM **M**emory **M**ap
BRAM	**B**lock **R**andom **A**ccess **M**emory
BSB	**B**ase **S**ystem **B**uilder
CF	**C**ompact **F**lash
CRAMFS	**C**ompressed RAM **F**ile **S**ystem
DCM	**D**igital **C**lock **M**anager
DCR Bus	**D**evice **C**ontrol **R**egister Bus
DDR	**D**ouble **D**ata **R**ate
EDK	**E**mbedded **D**evelopment **K**it
ELF	**E**xecutable and **L**inkable **F**ormat
EMAC	**E**thernet **M**edia **A**ccess **C**ontroller
EMC	**E**xternal **M**emory **C**ontroller
FIFO	**F**irst **I**n **F**irst **O**ut
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FSM	**F**inite **S**tate **M**achine
GPIO	**G**eneral **P**urpose **I**n **O**ut
GUI	**G**raphical **U**ser **I**nterface
HAW	**H**ochschule für **A**ngewandte **W**issenschaften
HDL	**H**ardware **D**escription **L**anguage
HW	**H**ard**w**are
IP	**I**ntellectual **P**roperty
IRQ	**I**nterrupt **Req**uest
JTAG	**J**oint **T**est **A**ction **G**roup
LMB	**L**ocal **M**emory **B**us
LSB	**L**east **S**ignificant **B**it
MHS	**M**icroprocessor **H**ardware **S**pecification
MPD	**M**icroprocessor **P**eripheral **D**efinition
MPMC	**M**ulti **P**ort **M**emory **C**ontroller
MSB	**M**ost **S**ignificant **B**it
MSS	**M**icroprocessor **S**oftware **S**pecification
NCD	**N**ative **C**ircuit **D**escription
NGC	**N**ative **G**eneric **C**ircuit
NGD	**N**ative **G**eneric **D**atabase

NMC	Physical Macro Library File
OPB	**O**n Chip **P**eripheral **B**us
PLB	**P**rocessor **L**ocal **B**us
PR	**P**artielle **R**ekonfiguration
PRM	**P**artiell **R**ekonfigurierbares **M**odul
PRR	**P**artiell **R**ekonfigurierbare **R**egion
RAM	**R**andom **A**ccess **M**emory
Root FS	Root **F**ile **S**ystem
SDK	**S**oftware **D**evelopment **K**it
SoC	**S**ystem **o**n a **C**hip
SW	**S**oft**w**are
SysACE	**Sys**tem **A**dvanced **C**onfiguration **E**nvironment
UART	**U**niversal **A**synchronous **R**eceiver **T**ransmitter
UCF	**U**ser **C**onstraints **F**ile
VHDL	**V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage
XCL	**X**ilinx **C**ache **L**ink
XPS	**X**ilinx **P**latform **S**tudio

Anhang A.

Übersicht zum Quellcode

Sämtliche im Rahmen dieser Arbeit erstellten Quellcodes befinden sich auf CD-ROM und können bei Herrn Prof. Dr.-Ing. Bernd Schwarz eingesehen werden.

HW

Lin_PR_HW_v5 Dieses Verzeichnis enthält die gesamten Projektdateien zur Erstellung des µBlaze-HW-System aus Kapitel 5.3.

- **bus_macros** busmacro_xc5v_async.nmc, busmacro_xc5v_async_enable.nmc
- **edk** EDK-Projekt-Ordner für das µBlaze-Prozessor-System
- **edk_ip** IP-Ordner für Addierer- und Multiplizierer PRMs.
- **merges** Zusammengefügte NCD-Dateien des statischen Teils mit den dynamischen Teilen.
- **image** Bitstreams für die initiale Konfiguration mit jeweils fs-boot (system_lin.ace), der eingebetteten Test-Applikation (system.ace) und den partiellen Konfigurationsdaten (adder.bit, mult.bit, blank.bit).
- **non_pr** Nicht partieller Entwurf aus dem statischen Teil und der adder-PRM.
- **pr_modules** Implementierungsdateien der Additions- und Multiplizierer-Module.
- **resources** IP des XPS_DCR_Sockets.
- **static** Implementierung des statischen Teils.
- **synth** Netzlisten der System-Komponenten.
- **top** Implementierung des Top-Moduls.

SW

Embedded

Block-RAM-basierte, eingebettete SW für die partielle Rekonfiguration

- **fs-boot** fs-boot.c/h, srec.c/h, time.c/h
  Bootloader der ersten Stufe.
- **TestApp** main.c, hwicap_parse.h
  Verifikation der partiellen Rekonfiguration.

Linux

- **ICAP_Treiber** xilinx_hwicap.c/h, fifo_icap.c/h, buffer_icap.c/h, Makefile
  - **Original** O.g. original Dateien von [37].
  - **Modifiziert** O.g. Dateien für die Anpassung an Linux modifiziert.
- **PRR_Test** icap_test.c, icap.c/h, xparameters.h, Makefile
  Linux-Applikation zur Verifikation der partiellen Rekonfiguration.

Anhang B.

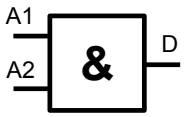
Funktionsweise von FPGAs

FPGAs sind im Grunde genommen nichts anderes als Speicher. Der Speicher ist in Speicherbereiche aufgeteilt, die mit Adressleitungen versehen sind und eine Stelle mit einer Adresse beschrieben bzw. ausgelesen werden kann. Diese Speicherbereiche können als sogenannte **L**ook **U**p **T**ables (LUT) angesehen werden. Als Beispiel wird hier eine 2-zu-1-LUT gezeigt (Abb. B.1).

A1	A2	D
0	0	
0	1	
1	0	
1	1	

Abbildung B.1.: Look Up Table mit leeren Speicherstellen

Diese LUT hat also 4 Speicherstellen, wo insgesamt 4 Bit abgespeichert werden können. Jedes Bit kann mit 2 Adressleitungen adressiert werden. Möchte man nun im FPGA ein logisches UND konfigurieren, dann wird die LUT wie in Abbildung B.2 geladen.



A1	A2	D
0	0	0
0	1	0
1	0	0
1	1	1

Abbildung B.2.: Look Up Table als UND Funktion

Eine LUT wird hierbei einmal beschrieben (konfiguriert). Danach wird nur noch gelesen. Je nachdem, welche Bitkombination an den Adressen anliegt, wird der für eine UND Operation richtige Wert ausgegeben. Diese LUTs werden je nach Kapazität des FPGAs durch ein komplexes Routingverfahren zusammengeschaltet und ergeben damit eine große digitale Schaltung. Es ist zu

beachten, dass der Zeitbedarf einer solchen UND Operation mit dem von einem einzigen Prozessorzyklus vergleichbar ist. Die Vorteile ergeben sich aber dadurch, dass mit einem FPGA mehrere solcher LUT-Ketten gebildet werden können. Damit sind keine Register- oder Speicheroperationen notwendig. Noch mehr auf die Geschwindigkeit wirkt sich aus, dass in einem FPGA parallele Ketten gebildet und somit viele Berechnungen gleichzeitig gemacht werden können. Als Beispiel soll eine Vektoraddition dienen.

$$x_1,...,x_n + y_1,...,y_n$$

Der Programmcode würde als Beispiel in der Sprache C folgendermaßen aussehen:

```
for (i = 0; i < n; i++)
    z[i] = x[i] + y[i];
```

In einem System mit einem einzigen CPU-Kern würde der gesamte Zeitbedarf für die Berechnung dem n-fachen der Ausführungszeit eines einzigen Schleifendurchganges betragen. Im FPGA könnte diese Berechnung komplett parallelisiert werden. Da alle Operanden voneinander unabhängig sind, würde dies schematisch wie in Abbildung B.3 aussehen.

Abbildung B.3.: Parallelisierte Vektoraddition

Literaturverzeichnis

[1] Doug Abbott. *Linux for Embedded and Real-Time Applications*. Elsevier, 2006.

[2] Altera. Fpga run-time reconfiguration: Two approaches. URL: http://www.altera.com/literature/wp/wp-01055-fpga-run-time-reconfiguration.pdf, 2008.

[3] Florent Berthelot, Fabienne Nouvel, and Dominique Houzet. A flexible systemlevel design methodology targeting run-time reconfigurable fpgas. *EURASIP Journal on Embedded Systems*, 2008:18–36, 2008.

[4] Denk and Zundel. *The DENX U-Boot and Linux Guide (DULG) for canyonlands*, 10 2009.

[5] Nij Dorairaj, Eric Shiflet, and Mark Goosman. Planahead software as a platform for partial reconfiguration. *Xilinx Xcell Journal*, 4:68–71, 2005.

[6] Scott Hauck and Andre DeHon. *Reconfigurable Computing*. Morgan Kaufmann, 2008.

[7] Steve Kilts. *Advanced FPGA Design*. Wiley & Sons, 2007.

[8] Rick Moleres and Milan Saini. Generating efficient board support packages. *Xilinx Embedded Magazine*, 3, 2006.

[9] Kevin Morris. A mighty wind of programmability. *FPGA and Programable Logic Journal*, January 2010.

[10] Jari Nurmi. *Processor Design: System-On-Chip Computing for ASICs and FPGAs*. Springer Publishing Company, Incorporated, 2007.

[11] Petalogix. *Getting Started with PetaLinux SDK*, 11 2009.

[12] Petalogix. Petalinux 0.40 user guide. URL: http://www.petalogix.com/resources/documentation/petalinux/userguide, 2009.

[13] Petalogix. *PetaLinux SDK Installation Guide*, 11 2009.

[14] Toomas P. Plaks, Marco D. Santambrogio, and Donatella Sciuto. Editorial: reconfigurable computing and hardware/software codesign. *EURASIP J. Embedded Syst.*, 2008:1–2, 2008.

[15] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers, 2nd Edition*. O'Reilly, 2001.

[16] Joachim Schröder, Tilo Gockel, and Rüdiger Dillmann. *Embedded Linux: Das Praxisbuch*. Springer, 2009.

[17] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght. Modular dynamic reconfiguration in virtex fpgas. *Computers and Digital Techniques, IEEE Proceedings* -, 153(3):157–164, May 2006.

[18] Ramachandran Vaidyanathan and Jerry L. Trahan. *Dynamic Reconfiguration: Architectures and Algorithms*. Kluwer, 2003.

[19] Wayne Wolf. *High Performance Embedded Computing*. Morgan Kaufmann, 2007.

[20] Xilinx. *iMPACT User Guide*, 4.1 edition, 2002.

[21] Xilinx. *PLB IPIF*, v2.02a edition, 04 2005.

[22] Xilinx. *Device Control Register Bus (DCR) v2.9*, v1.00a edition, 08 2006.

[23] Xilinx. *MicroBlaze Processor Reference Guide*, v8.1 edition, 12 2007.

[24] Xilinx. *ML505/ML506 Evaluation Platform: User Guide*, 10 2007.

[25] Xilinx. *Multi-Port Memory Controller (MPMC)*, v3.00b edition, 11 2007.

[26] Xilinx. *PLBV46 to DCR Bridge*, v1.00a edition, 04 2007.

[27] Xilinx. *Processor Local Bus (PLB) v4.6*, v1.00a edition, 08 2007.

[28] Xilinx. *XPS Ethernet Lite Media Access Controller*, v1.00a edition, 09 2007.

[29] Xilinx. *XPS General Purpose Input/Output (GPIO)*, v1.00a edition, 09 2007.

[30] Xilinx. *XPS HWICAP*, v1.00a edition, 10 2007.

[31] Xilinx. *XPS Interrupt Controller*, v1.00a edition, 09 2007.

[32] Xilinx. *XPS Multi-CHannel External Memory Controller (XPS MCH EMC)*, v1.00a edition, 09 2007.

[33] Xilinx. *XPS Timer/Counter*, v1.00a edition, 09 2007.

[34] Xilinx. *Early Access Partial Reconfiguration User Guide For ISE 9.2.04i*, 09 2008.

[35] Xilinx. *PlanAhead User Guide*, v10.1 edition, 01 2008.

[36] Xilinx. *Virtex-5 Libraries Guide for HDL Designs*, 2008.

[37] Xilinx. Xilinx icap driver for linux 2.6.x. URL: http://tomoyo.sourceforge.jp/cgi-bin/lxr/source/drivers/char/xilinx_hwicap, 2008.

[38] Xilinx. *Constraints Guide*, 09 2009.

[39] Xilinx. *EDK Concepts, Tools, and Techniques*, 06 2009.

[40] Xilinx. *Embedded System Tools Reference Guide*, 09 2009.

[41] Xilinx. Pr-lounge, partial reconfiguration early access software tools for ise 9.2i sp4. URL: http://www.xilinx.com/support/prealounge/protected/index.htm, 2009. Xilinx restricted Homepage for Partial Reconfiguration.

[42] Xilinx. *Virtex-5 FPGA Configuration User Guide*, 02 2009.

[43] Xilinx. *Virtex-5 FPGA User Guide*, 11 2009.

[44] Karim Yaghmour. *Building Embedded Linux Systems*. O'Reilly, 2003.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 26. Februar 2010	
Ort, Datum	Unterschrift