technische universität
dortmund

Diplomarbeit

Merkmalbasierte statische Konfigurierung von MPSoCs

Matthias Meier
12. Mai 2009

Betreuer:
Dr. Michael Engel
Prof. Dr.-Ing. Olaf Spinczyk

Fakultät für Informatik
Embedded System Software Group (LS12)
Technische Universität Dortmund
http://ess.cs.tu-dortmund.de

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 12. Mai 2009

Matthias Meier

Zusammenfassung

Durch die steigende Integrationsdichte von FPGAs ist es in der heutigen Zeit ohne weiteres möglich, komplexe Multiprozessorsysteme auf einem Chip abzubilden. Diese Systeme bestehen aus verschiedenen Komponenten, wie Prozessoren, Speichern, Peripherien und Kommunikationsstrukturen, die sich vollständig auf einem FPGA unterbringen lassen. Entworfen werden solche Systeme mittels Hardwarebeschreibungssprachen wie VHDL oder Verilog. Diese Diplomarbeit befasst sich mit der Konfigurierung von MPSoCs anhand von Merkmalen. Das Konzept der Konfigurierung mittels Merkmalmodellen wird im Bereich der Software eingesetzt und in dieser Arbeit für MPSoCs untersucht. MPSoCs bieten viele Möglichkeiten zur individuellen Konfigurierung. Es kann die Anzahl der Soft Cores, die Größe des Speichers oder die Breite des Datenbusses konfiguriert werden. Mit diesem Ansatz soll eine Grundlage geschaffen werden, um zukünftig anhand einer gemeinsamen Merkmalauswahl sowohl die Systemsoftware als auch das MPSoC mit ein und derselben Vorgehensweise an die Anforderungen einer Anwendung anpassen zu können.

Inhaltsverzeichnis

1 Einführung

Durch die steigende Integrationsdichte von FPGAs[1] ist es in der heutigen Zeit ohne weiteres möglich, komplexe Multiprozessorsysteme auf einem Chip abzubilden. Dabei bestehen diese Systeme aus Prozessoren, Speichern, Peripherien und Kommunikationsstrukturen, die sich vollständig auf einem FPGA unterbringen lassen. Im Bereich der Eingebetteten Systeme sind MPSoCs (Multiprocessor Systems-on-Chip) ein vielversprechendes Instrument [1]. So kann mit MPSoCs im Vergleich zu Einzelprozessor-Lösungen eine höhere Leistung bei einer niedrigeren Taktfrequenz erzielt werden. Dies kann sich wiederum positiv auf den Energieverbrauch auswirken, was gerade im Bereich der Eingebetteten Systeme ein wichtiger Aspekt ist.
Die Systeme für FPGAs werden mittels Hardwarebeschreibungssprachen wie VHDL[2] oder Verilog entworfen. Auch die eingesetzten Prozessoren in einem MPSoC lassen sich mit diesen Sprachen beschreiben und werden als Soft Cores bezeichnet. Soft Cores sind also nicht fest auf einem FPGA verdrahtet, sondern in der konfigurierbaren Logik des FPGAs untergebracht. Soft Cores bieten den ungemeinen Vorteil, dass sie an auftretende Problemstellungen adaptiert und um beliebige Funktionen erweitert werden können. Manche FPGAs bieten zusätzlich fest integrierte Hard Cores, wie beispielsweise den leistungsstarken PowerPC-Prozessor in dem XC2VP30 FPGA (siehe Kapitel 3.1.2). Diese Prozessoren benötigen zwar weniger Chipfläche und lassen sich höher takten als die Soft Cores, können aber dafür nicht angepasst werden. Der Soft Core MicroBlaze [2] von Xilinx benötigt, abhängig von der eingesetzten Variante, ca. 1000 Logikzellen auf einem FPGA, wobei moderne FPGAs ein Vielfaches von den benötigten Ressourcen zur Verfügung stellen können.
Diese Diplomarbeit befasst sich mit der Konfigurierung von MPSoCs anhand von Merkmalen. MPSoCs bieten viele Möglichkeiten zur individuellen Konfigurierung. Es kann die Anzahl der Soft Cores, die Größe des Speichers oder die Breite des Datenbusses konfiguriert werden. Sogar der komplette Austausch von Kommunikationsstrukturen zur besseren Anpassung an gegebene Anwendungen wird realisiert werden.

1.1 Motivation

Die Entwicklung von komplexen MPSoCs kann mitunter sehr aufwendig sein. Gerade wenn ein MPSoC auf eine spezielle Anwendung zugeschnitten werden muss, werden Automatismen benötigt, die es dem Entwickler erlauben, aus einer Menge von Merkmalen

[1] Field Programmable Gate Array
[2] Very High Speed Integrated Circuit Hardware Description Language

ein geeignetes MPSoC zu generieren. Am Lehrstuhl 12 in der Arbeitsgruppe für Eingebettete Systemsoftware der Technischen Universität Dortmund[1] wird die Konfigurierung von Betriebssystemen für Eingebettete Systeme anhand von Merkmalen bereits eingesetzt. Zu nennen wären in diesem Zusammenhang die Eigenentwicklung CiAO und das Open Source eingebettete Betriebssystem eCos. Bisher sind diese Ansätze allerdings ausschließlich Software-Zentriert. Diese Ansätze und Erfahrungen sollen nun in dieser Arbeit aus dem Software-Bereich in den Hardware-Bereich adaptiert werden. Dabei muss vor allem die Eignung der Merkmalmodelle für die MPSoCs untersucht werden. So wird eine Grundlage geschaffen, um zukünftig anhand einer gemeinsamen Merkmalauswahl sowohl die Systemsoftware als auch das MPSoC mit ein und derselben Vorgehensweise an die Anforderungen einer Anwendung anzupassen. Denkbar wäre beispielsweise abhängig von einer Anwendung, die Prozesse der Systemsoftware zu generieren und auf der darunterliegenden Hardware-Ebene einen entsprechenden Prozessor zu erzeugen.

1.2 Ziele

Die Diplomarbeit gliedert sich in zwei Phasen. In der ersten Phase wird ein möglichst flexibles konfigurierbares MPSoC aufgebaut. Dazu wurde ein geeigneter Soft Core ausgewählt und um eine universelle Schnittstelle zur Kommunikation erweitert. Dies ermöglicht neben homogenen auch die Konfigurierung von heterogenen Systemen. Zur Kommunikation zwischen den Soft Cores werden sowohl Kommunikationsmechanismen als auch verschiedene Kommunikationsstrukturen implementiert. Anschließend wird das MPSoC auf einem FPGA implementiert und getestet sowie verschiedene Konfigurationen des MPSoCs nach Ressourcenverbrauch und Anwendung eingeordnet. Die zweite Phase der Diplomarbeit beschäftigt sich mit der statischen Konfigurierung anhand von Merkmalen. Zuerst werden geeignete Merkmale gesucht, die es erlauben, ein MPSoC möglichst effizient anzupassen. Im Anschluss werden dann die Ansätze, die bisher für die Konfigurierung von Software genutzt wurden, auf den Hardware-Bereich angewendet und auf ihre Eignung in diesem Bereich untersucht.

1.3 Gliederung der Arbeit

In Kapitel 2 werden benötigte Grundlagen zu Themen dieser Diplomarbeit präsentiert. Es werden Merkmalmodelle, der Aufbau eines FPGAs und Konzepte zur Kommunikation auf einem Chip vorgestellt. Kapitel 3 stellt die in dieser Diplomarbeit verwendeten FPGAs und zwei aus dem Software-Bereich bekannte Tools zur Konfigurierung vor. In Kapitel 4 werden andere Arbeiten, die sich mit dem Thema MPSoCs und deren Konfigurierung beschäftigen, vorgestellt und bewertet. Die Entwicklung des konfigurierbaren MPSoCs ist in Kapitel 5 beschrieben. Es werden die Komponenten und der Aufbau des

[1] http://ess.cs.tu-dortmund.de

MPSoCs, die verwendeten Mechanismen zur Kommunikation und die Programmierung des MPSoCs vorgestellt. In Kapitel 6 werden verschiedene Ansätze zur Konfigurierung des MPSoCs vorgestellt. Eine Bewertung der angewendeten Konfigurierungstechniken aus dem Software-Bereich sowie eine Auswertung der Kommunikationsstrukturen und des Ressourcenbedarfs einiger MPSoC Konfigurationen finden sich in Kapitel 7. Kapitel 8 fasst die Arbeiten und Ergebnisse zusammen und gibt einen Ausblick auf weitere Arbeiten.

2 Grundlagen

In diesem Kapitel werden einige grundlegende Bereiche dieser Arbeit vorgestellt, damit die folgenden Kapitel leichter nochzuvollziehen sind. Zuerst werden in Kapitel 2.1 die Begriffe der Merkmale und der Merkmalmodelle anhand eines Beispiels erläutert. In Kapitel 2.2 werden einige Verfahren zur Kommunikation in einem MPSoC gezeigt. Damit die technischen Begriffe zu einem FPGA im weiteren Verlauf der Arbeit eingeordnet werden können, wird in Kapitel 2.3 der Aufbau eines FPGAs vorgestellt.

2.1 Merkmalmodelle

Merkmalmodelle [3, 4] werden zur Beschreibung der Anforderungen bei Softwareproduktlinien verwendet und in einem Graphen visualisiert. Neben dem Graphen kann das Merkmalmodell noch andere Informationen, wie z.B. Einschränkungen enthalten. Mit Einschränkungen können beispielsweise Konflikte zwischen zwei Merkmalen ausgedrückt werden. Die Wurzel des Merkmaldiagramms wird als Konzeptknoten bezeichnet und die darunter liegenden Knoten als Merkmale. Die Merkmale beschreiben verschiedene Eigenschaften eines Produktes. So lassen sich über eine Auswahl an Merkmalen verschiedene Varianten eines Produktes bestimmen. Anhand des Beispiels in Abbildung 2.1 werden im Weiteren die einzelnen Elemente des Merkmaldiagramms vorgestellt. Notwendige-Merkmale werden durch einen ausgefüllten Kreis gekennzeichnet und müssen zur Beschreibung des Konzepts hinzugefügt werden, insofern der Elternknoten zur Beschreibung gehört. Im Beispiel müssen also die Merkmale Motor, Getriebe und Reifen zu dem Konzept hinzugefügt werden, da die Wurzel bzw. der Konzeptknoten immer zum Konzept gehört. Das Merkmal Sonderausstattung kann optional zum Konzept hinzugefügt werden. Wiederum gilt, wie bei den Notwendigen-Merkmalen, dass der Elternknoten zum Konzept gehören muss. Alternative-Merkmale, wie Benziner oder Diesel, treten in Gruppen auf und werden durch eine Kurve unterhalb des Elternknotens gekennzeichnet. Eines dieser beiden Merkmale gehört zum Konzept, falls der Elternknoten zum Konzept gehört. Die letzte Art von Merkmalen sind die Oder-Merkmale. Oder-Merkmale werden durch eine ausgefüllte Kurve unter dem Elternknoten symbolisiert. Wie schon die Alternativen-Merkmale, treten auch die Oder-Merkmale in Gruppen auf. Wenn der Elternknoten Teil des Konzepts ist, werden in diesem Fall beide Merkmale oder zumindest ein Merkmal zum Konzept hinzugefügt. Diese Art der Modellierung wird von dem in Kapitel 3.3 präsentierten Variantenmanagement-Tool pure::variants verwendet.

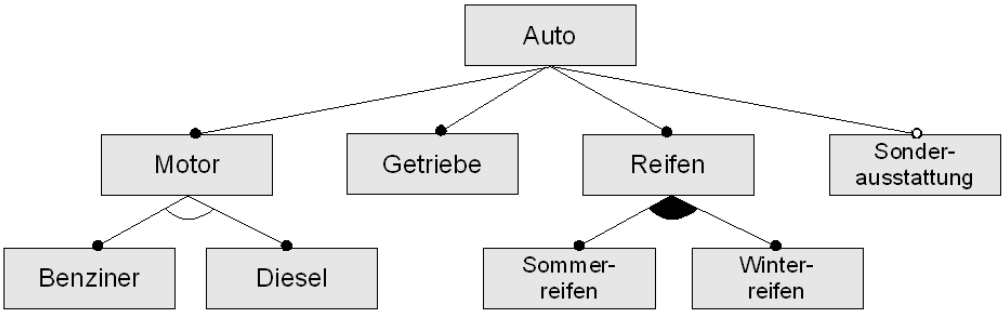

Abbildung 2.1: Merkmaldiagramm

2.2 Kommunikation in einem MPSoC

Eine wesentliche Herausforderung bei der Integration von Multiprozessorsystemen auf einem Chip ist die Kommunikation zwischen den Prozessoren. Da in dieser Arbeit jeweils nur ein Prozess zu einem Prozessor zugeordnet ist, wird die Kommunikation zwischen den Prozessen mit der Kommunikation zwischen den Prozessoren gleichgesetzt. Im weiteren Verlauf der Arbeit wird daher der Begriff der Interprozesskommunikation, kurz IPC, verwendet. Um eine Kommunikation zwischen den einzelnen Prozessen aufzubauen, werden IPC-Mechanismen und Kommunikationsstrukturen benötigt. Unter IPC-Mechanismen werden Techniken verstanden, die es den Prozessen erlauben, auf verschiedene Weise zu interagieren. Einige dieser Mechanismen werden in Kapitel 2.2.1 präsentiert. Die Kommunikationsstrukturen bzw. die Topologien bilden die nötige Infrastruktur unter den IPC-Mechanismen, um die Daten von einem Prozessor zum anderen zu leiten. Eine Auswahl der verbreitetsten Kommunikationsstrukturen wird in Kapitel 2.2.2 vorgestellt.

2.2.1 IPC-Mechanismen

Im Folgenden werden einige IPC-Mechanismen [5] gezeigt, die sich insbesondere für die Interaktion von Prozessen in dem in dieser Arbeit zu entwickelnden MPSoC eignen. Da die IPC-Mechanismen in Hardware entwickelt werden und eine einheitliche Schnittstelle zum Soft Core vorausgesetzt ist, muss eine gleichartige Ansteuerung der IPC-Mechanismen gewährleistet werden. Dies ist für speziellere IPC-Mechanismen, wie Monitore oder Signale, nur schwer bzw. nicht zu realisieren. Für die IPC-Mechanismen gibt es im Wesentlichen drei verschiedene Anwendungsbereiche, wobei manche Mechanismen auch mehrere dieser Bereiche abdecken können. Diese Bereiche sind der Datenaustausch zwischen Prozessen, die Synchronisierung von Prozessen oder der gegenseitige Ausschluss bei gemeinsamen Ressourcen. Die Auswahl des passenden IPC-Mechanismus ist dabei von der eingesetzten Anwendung abhängig.

- **Semaphore:**

  Semaphore können für den gegenseitigen Ausschluss beim Zugriff auf gemeinsame Ressourcen oder zur Synchronisierung eingesetzt werden. Dabei gibt es zwei Arten von Semaphoren, binäre und zählende. Für binäre Semaphore gibt es nur zwei Zustände. Wenn die gemeinsame Ressource von einem Prozess belegt ist, hat die Semaphore den Wert Null und sonst den Wert Eins. Zählende Semaphore können eingesetzt werden, falls nicht nur eine Ressource dieser Art in dem System vorhanden ist. Für jede Ressource, die von einem Prozess angefordert wird, erniedrigt sich der Wert der Semaphore um Eins. Wenn die Semaphore gleich Null ist, sind alle Ressourcen belegt und der anfragende Prozess muss warten, bis eine der Ressourcen wieder freigegeben wird.

- **Event Flags:**

  Mit der Hilfe von Event Flags lassen sich Prozesse synchronisieren. Event Flags sind eine Reihe von Binärwerten, die entweder gesetzt oder gelöscht werden können. Jedes Bit steht dabei für ein Event Flag, das von einem beliebigen Prozess gesetzt und von beliebig vielen Prozessen zur Synchronisierung abgefragt werden kann.

- **Message Queues:**

  Message Queues speichern die Nachrichten für den Empfänger-Prozess zwischen. Sobald der Empfänger eine Nachricht anfordert, wird diese aus der Message Queue ausgelesen und weitergeleitet. Falls die Message Queue keine Nachrichten für den Prozess enthält, gibt es drei mögliche Vorgehensweisen. Der Prozess kann seine Ausführung einfach fortsetzen, eine bestimmte Zeitspanne auf die Nachricht warten oder solange warten, bis eine Nachricht eintrifft. Die Sortierung der Nachrichten in der Message Queue kann von der jeweiligen Implementierung abhängen und von einer priorisierten Liste bis zu einer FIFO reichen.

- **Shared Memory:**

  Der Shared Memory ist ein gemeinsamer Speicherbereich, auf den alle Prozesse lesend und schreibend zugreifen können. In erster Linie kann dieser Mechanismus zum schnellen Austausch von großen Datenmengen genutzt werden. Aber auch der gegenseitige Ausschluss beim Zugriff auf gemeinsame Ressourcen ist mit dem Shared Memory umzusetzen, insofern zumindest atomare Operationen auf dem Speicher unterstützt werden.

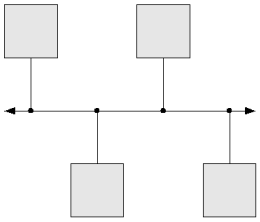

Abbildung 2.2: Bus

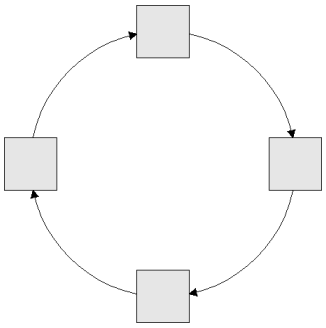

Abbildung 2.3: Ring

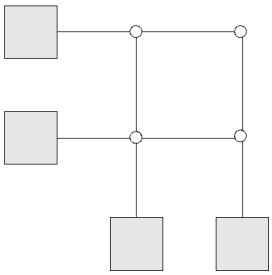

Abbildung 2.4: Crossbar

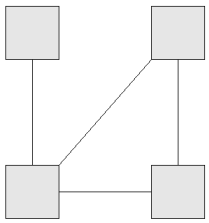

Abbildung 2.5: Custom

2.2.2 Kommunikationsstrukturen

Wie bereits zu Beginn des Kapitels erläutert, werden neben den IPC-Mechanismen noch Kommunikationsstrukturen [6] zwischen den Prozessoren benötigt. Die eingesetzte Kommunikationsstruktur hat einen erheblichen Einfluss auf Eigenschaften wie den Datendurchsatz oder die Latenz der Nachrichten. Neben diesen physikalischen Faktoren ist es wichtig, eine zur Art der Kommunikation passende Kommunikationsstruktur zu wählen. Muss zum Beispiel jeder Teilnehmer mit jedem anderen kommunizieren? Sollen die Teilnehmer parallel kommunizieren können? Müssen bei der Kommunikation unbedingt bestimmte Latenzen eingehalten werden? Im Folgenden werden einige verbreitete Kommunikationsstrukturen vorgestellt.
Bei einem herkömmlichen Bus, wie in Abbildung 2.2, kann immer nur ein Busteilnehmer zur gleichen Zeit senden. Dadurch wird zum einen die Bandbreite des Busses limitiert und zum anderen skaliert der Bus schlecht über die Anzahl der Teilnehmer. Zudem müssen Verfahren zur Kollisionskontrolle, wie z.B. der Arbitrierung eingesetzt werden, um Kollisionen zu verhindern bzw. zu erkennen. Im Gegensatz zum Bus gibt es beim Ring in Abbildung 2.3 keine Kollisionen. Die Nachrichten werden von Station zu Station weitergeleitet, bis sie ihr Ziel erreicht haben. Dieser Umstand führt dazu, dass die Nachrichten bei größer werdenden Ringen später beim Empfänger eintreffen. Der Ring kann aber für Anwendungen, die auf den einzelnen Stationen ein Teilergebnis berechnen und dieses an die nächste Station zur Berechnung weitergeben, von Vorteil sein, da

die Nachrichten direkt, parallel und ohne Kollisionen übertragen werden können. Eine kostenintensive Methode stellt eine Punkt-zu-Punkt-Verbindung zwischen allen Teilnehmern des Systems dar [7]. Die Vorteile liegen aber darin, dass die Teilnehmer parallel und beliebig untereinander kommunizieren können. Ein Kompromiss wird mit den Crossbars aus Abbildung 2.4 eingegangen. In dieser Architektur können Verbindungen zwischen beliebigen Teilnehmern geschaltet werden. So kann zwischen beliebigen Teilnehmern des Systems über eine direkte Verbindung kommuniziert werden. Im Vergleich zur Punkt-zu-Punkt-Verbindung ist der Aufwand geringer, jedoch können zwei Teilnehmer nicht zur gleichen Zeit mit einem weiteren Teilnehmer kommunizieren. Denkbar wäre auch eine an die Anwendung angepasste Punkt-zu-Punkt-Verbindung, wie sie in Abbildung 2.5 dargestellt ist. So könnten Ressourcen gespart werden und trotzdem wären wichtige anwendungsabhängige Verbindungen über Punkt-zu-Punkt-Verbindungen realisiert.

2.3 Aufbau eines FPGAs

Ein FPGA [8, 9] ist ein Chip mit rekonfigurierbarer Logik. Die Funktionalität des FPGAs lässt sich dabei vom Benutzer festlegen. Da der Aufbau eines FPGAs und die Namensgebung der Elemente eines FPGAs stark herstellerabhängig sind, wird im Weiteren exemplarisch der Aufbau eines FPGAs der Virtex-II Familie von Xilinx vorgestellt. In Abbildung 2.6 ist der schematische Aufbau dieses FPGAs dargestellt.

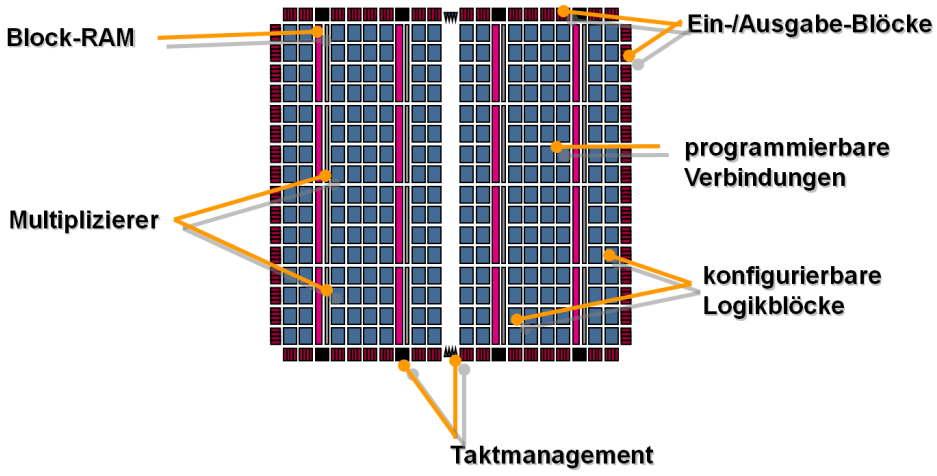

Abbildung 2.6: Aufbau eines FPGAs (Quelle: [9])

Die konfigurierbaren Logik Blöcke (CLBs) sind die Hauptbestandteile eines FPGAs und beinhalten die Logik der Schaltung. Verbunden werden die CLBs über programmierbare Verbindungen. Des Weiteren befinden sich auf einem FPGA kleine Speicher (Block-RAMs), Multiplizierer, Elemente zum Taktmanagement und Ein-/Ausgabe-Blöcke (IOBs).

Über die IOBs können Verbindungen aus den FPGAs heraus oder herein geführt werden. Jeder CLB besteht aus vier Slices. Die Slices wiederum beinhalten zwei Look-Up-Tables (LUTs) und zwei Flip-Flops. Die LUTs für dieses FPGA besitzen vier Eingänge und einen Ausgang. Sie lassen sich als Distributed-RAM mit 16 Bit Speicher, als 16 Bit Schieberegister oder als Wertetabelle konfigurieren. Mit der Wertetabelle lassen sich beliebige logische Funktionen mit vier Variablen realisieren. Prinzipiell sind auch die Wertetabellen nichts anderes als feste 16 Bit Speicher, die über eine 4 Bit breite Adressleitung ausgelesen werden können.

3 Verwendete Tools und Hardware

In diesem Kapitel werden die verwendeten FPGA-Entwicklungsboards in Unterkapitel 3.1 und zwei Tools zur Konfigurierung aus dem Software-Bereich in den Unterkapiteln 3.2 und 3.3 vorgestellt.

3.1 FPGA-Entwicklungsboard

Die MPSoC Konfigurationen werden neben den Tests in der Simulation auch in Hardware implementiert. Dazu werden die beiden im Folgenden vorgestellten FPGA-Entwicklungsboards von Xilinx genutzt.

3.1.1 Xilinx Spartan-3E

Das Spartan-3E [10] ist ein Einsteigerboard mit einem XC3S500E FPGA und aufgrund der geringen Anzahl an Block-RAMs (45 KB) nicht unbedingt für den Einsatz eines MPSoCs geeignet. Es wird aber in dieser Arbeit zum Debuggen und Testen der Kommunikationsstrukturen mit zwei Soft Cores genutzt. Vor allem die wesentlich schnellere Erzeugung des Bit-Files zur Konfigurierung des FPGAs hebt dieses Entwicklungsboard von dem nachfolgenden ab.

Slices	CLB Flip-Flops	LUTs	Block-RAMs	Clock (MHz)	RS-232	LEDs
4.656	9.312	9.312	20	50	2	8

Tabelle 3.1: Ressourcen des Spartan-3E

3.1.2 Xilinx VIRTEX-II Pro

Das VIRTEX-II Pro [11] Entwicklungsboard mit einem XC2VP30 FPGA bietet wesentlich mehr Ressourcen und verfügt fast über die siebenfache Menge an Block-RAM. Zudem hat das FPGA dieses Entwicklungsboards zwei PowerPC-Prozessoren fest integriert.

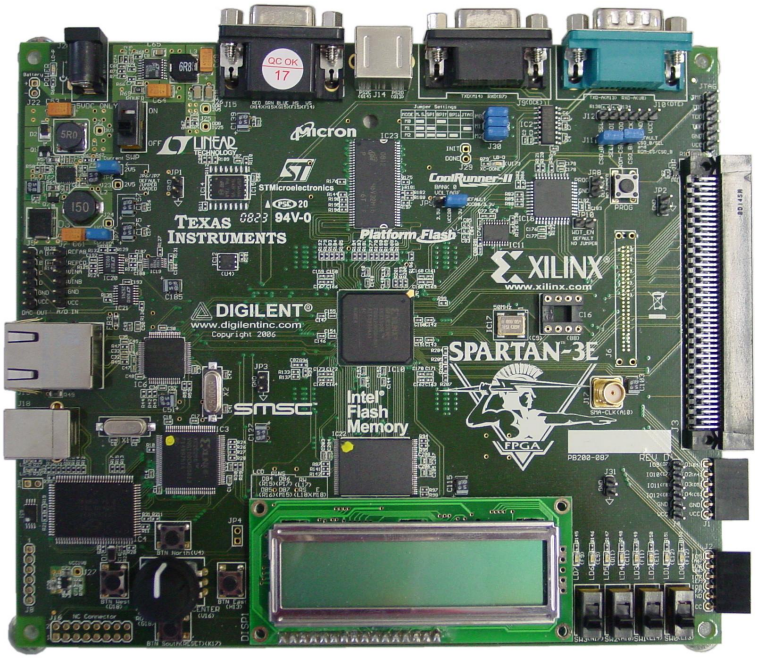

Abbildung 3.1: Xilinx Spartan-3E

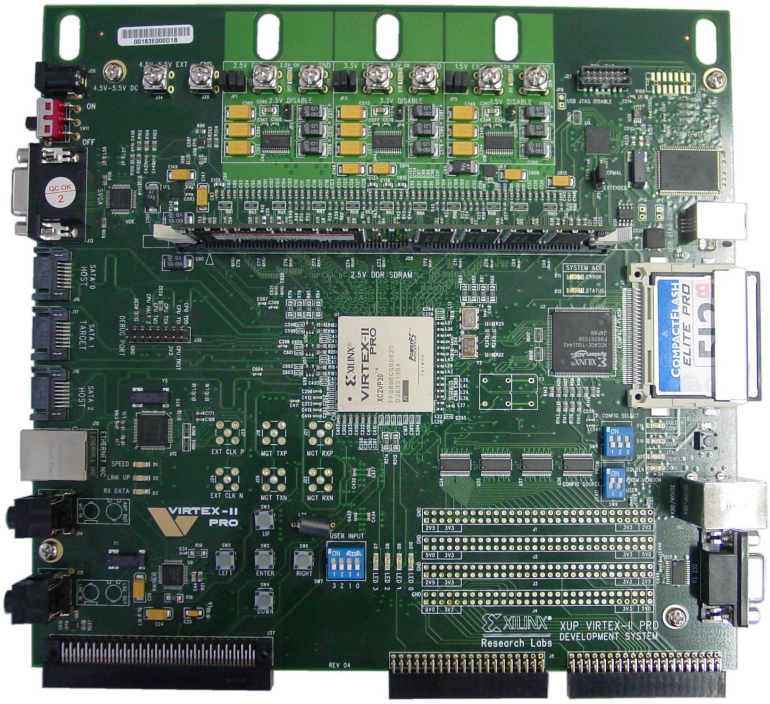

Abbildung 3.2: Xilinx VIRTEX-II Pro

Slices	CLB Flip-Flops	LUTs	Block-RAMs	Clock (MHz)	RS-232	LEDs
13.696	27.392	27.392	136	100	1	4

Tabelle 3.2: Ressourcen des VIRTEX-II Pro

3.2 XVCL

XVCL (XML-based Variant Configuration Language) [12, 13, 14] ist eine allgemeine Mark-Up Sprache, mit deren Hilfe Software oder sonstige auf text-basierende Dateien konfiguriert werden können. XVCL arbeitet nach dem Adaptionskonzept auf der Basis von Bassetts Frame-Technologie [15].
In XVCL werden sogenannte x-frames verwendet. Ein x-frame ist eine XML-Datei, die Teile des tatsächlichen Quellcodes und XVCL-Befehle zur Konfigurierung enthält. XVCL-Befehle können zwischen dem Quellcode eingefügt werden und haben die Form von XML-Tags. Mittels des XVCL-Befehls *<adapt>* können in einen x-frame weitere x-frames eingebunden werden. So ist es möglich, Teile des Quellcodes beliebig oft wiederzuverwenden. Eine Menge von in Beziehung stehenden x-frames wird als x-framework bezeichnet. Das x-framework beschreibt die Menge aller möglichen Lösungen. Um eine konkrete Lösung aus diesem Lösungsraum zu erhalten, wird eine entsprechende Problembeschreibung benötigt. Diese Problembeschreibung wird im Specification x-frame, dem SPC, festgelegt.

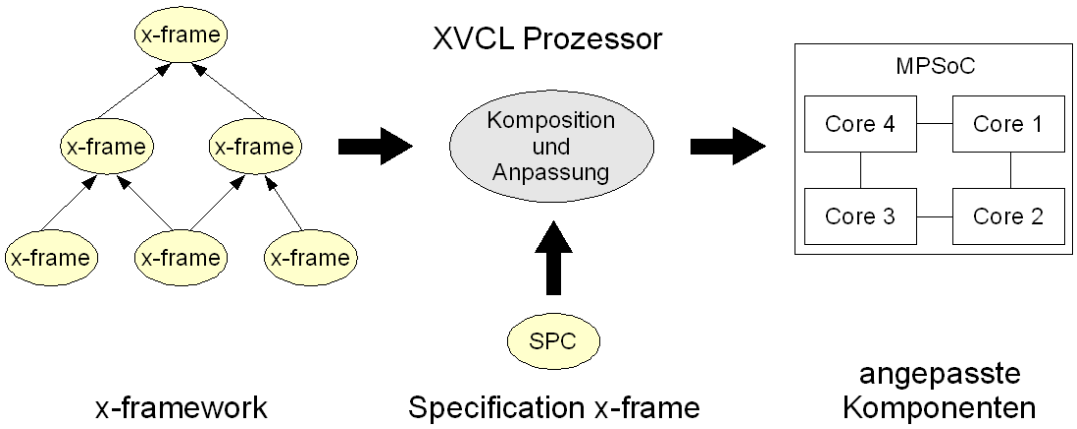

Abbildung 3.3: Konzept von XVCL (nach Quelle: [16])

Neben der Sprache bietet XVCL noch einen Frameprozessor. Mit dessen Hilfe kann anhand des x-frameworks und des Specification x-frames der fertige Quellcode, z.B. für ein MPSoC, in VHDL erstellt werden. Die Vorgehensweise des XVCL-Prozessors wird in Kapitel 3.2.1 und eine Teilmenge der zur Verfügung stehenden XVCL-Befehle wird in Kapitel 3.2.2 vorgestellt. Anschließend folgt ein einfaches Beispiel in Kapitel 3.2.3.

3.2.1 Ablauf der Verarbeitung

Der Frameprozessor startet seine Verarbeitung mit dem Specification x-frame. Wenn der Frameprozessor in einem x-frame auf einen *<adapt>*-Befehl trifft, stoppt der Frameprozessor seine Verarbeitung in diesem Frame und springt in den angegebenen x-frame. Sobald ein x-frame vollständig abgearbeitet ist, springt der Frameprozessor zurück zum aufrufenden x-frame und setzt nach dem *<adapt>*-Befehl wieder an. Wenn der Frameprozessor das Ende des Specification x-frames erreicht, ist die Verarbeitung abgeschlossen. In Abbildung 3.4 wird noch einmal die Vorgehensweise des Frameprozessors verdeutlicht.

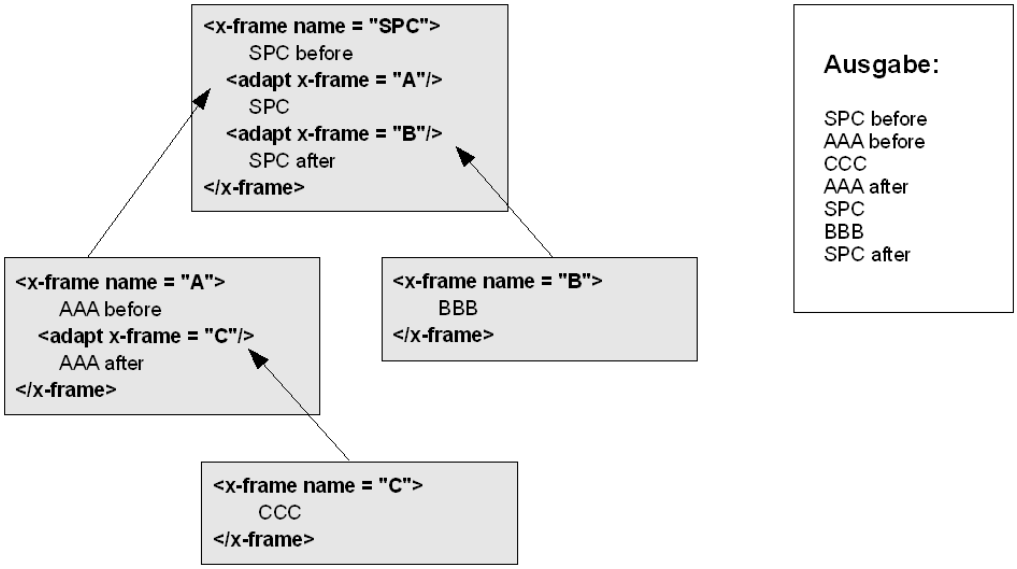

Abbildung 3.4: Arbeitsweise des XVCL Frameprozessors (nach Quelle: [14])

3.2.2 XVCL-Befehle

Neben dem bereits vorgestellten XVCL-Befehl *<adapt>*, bietet XVCL noch eine Reihe weiterer Befehle, die oftmals an grundlegende Konzepte aus Programmiersprachen erinnern.

- **x-frame Struktur**

  Der *<x-frame>*-Befehl gibt den Rahmen für den x-frame an und bestimmt seinen Namen. Innerhalb dieses Rahmens können sowohl Quellcode als auch weitere XVCL-Befehle angegeben werden. Optional kann für jeden x-frame ein Dateiname zur Ausgabe und das Verzeichnis bzw. der Pfad festgelegt werden.

```
<x-frame name = "x-frame Name" [outdir = "Pfad"] [outfile = "Datei"]>
   ...
</x-frame>
```

- **Variablen**

  Mit dem Befehl *<set>* kann einer Variable ein Wert zugewiesen werden.

```
<set var = "Variablenname" value = "Wert"/>
```

  Neben einfachen Variablen gibt es noch die Möglichkeit mit dem Befehl *<set multi>* einer Multi-Variable mehrere Werte zuzuweisen. Multi-Variablen werden für die später vorgestellten While-Schleifen benötigt.

```
<set-multi var = "Variablenname" value = "Wert1,Wert2,..."/>
```

  Der Inhalt einer Variable kann mit dem Befehl *<value-of>* in den Quelltext eingefügt werden. Dabei überschreibt der Wert den ursprünglichen Befehl. Es kann allerdings nicht direkt der Variablenname angegeben werden, sondern es muss ein Ausdruck in folgender Form verwendet werden: *?@Variable?*. Das „@“ steht für den Wert der Variable und die „Fragezeichen“ müssen diesen Ausdruck umschließen.

```
<value-of expr = "?@Variablenname?"/>
```

- **Select**

  Der *<select>*-Befehl ist eine Art Select/Case- bzw. Switch/Case-Anweisung aus bekannten Programmiersprachen. Abhängig von der angegebenen Variable kann aus einer Menge von Optionen ausgewählt werden. Falls keine der Optionen zutrifft, wird *<otherwise>* gewählt.

```
<select option = "Variablenname">
  <option value = "Wert">
    ...
  </option>
  <otherwise>
    ...
  </otherwise>
</select>
```

- **While**

  Der Befehl *<while>* iteriert abhängig von der Anzahl der Elemente in der angegebenen Multi-Variable, über seinen Rumpf. Im Schleifenrumpf können sich Quellcode oder andere XVCL-Befehle befinden.

```
<while using-items-in = "Multi-Variablenname">
  ...
</while>
```

3.2.3 Beispiel

Das folgende Beispiel zeigt eine vereinfachte Anwendung von XVCL zur Generierung eines MPSoCs.

Im SPC-Frame wird die zu konfigurierende Variante des MPSoCs definiert. Es wird der Dateiname für die Ausgabe angegeben, die Core-IDs bzw. die Anzahl der Cores werden festgelgt und welcher Core mit der UART-Schnittstelle verbunden werden soll, wird definiert.

```
<x-frame name = "SPC"  outfile = "mpsoc.vhd">
  <set-multi var = "CoreIDs" value = "1,2,3,4"/>
  <set var = "UART" value = "3"/>
  <adapt x-frame = "topmpsoc"/>
</x-frame>
```

Im MPSoC-Frame wird zuerst der eigene Text abgearbeitet. Anschließend wird in jedem Schleifendurchlauf das aktuelle Element aus der Variable *CoreIDs* der Variable *CoreID* zugewiesen und jeweils einmal der Core-Frame verarbeitet.

```
<x-frame name = "MPSoC">
Das MPSoC hat folgende Cores:

  <while using-items-in = "CoreIDs">
    <set var = "CoreID" value = "?@CoreIDs?"/>
    <adapt x-frame = "Core"/>
  </while>
</x-frame>
```

Der Core-Frame wird durch den MPSoC-Frame für jeden im SPC angegebenen Core einmal eingebunden. Der Select-Befehl wird genutzt um zu vergleichen, ob der jetzt zu erzeugende Core eine UART-Schnittstelle haben soll.

```
<x-frame name = "Core">
  <select option = "CoreID">
    <option value = ?@UART?>
Core <value-of expr = "?@CoreID?"/> benutzt die UART-Schnittstelle
    </option>
    <otherwise>
Core <value-of expr = "?@CoreID?"/>
```

```
    </otherwise>
  </select>
</x-frame>
```

Nachdem der XVCL-Prozessor die Frames durchlaufen hat, befindet sich in der Datei mpsoc.vhd folgender Inhalt:

```
Das MPSoC hat folgende Cores:

Core 1
Core 2
Core 3 benutzt die UART-Schnittstelle
Core 4
```

3.3 pure::variants

pure::variants [17, 18] ist eine Technologie zum Variantenmanagement von Softwareproduktlinien. Realisiert ist pure::variants als Plugin für die Entwicklungsumgebung Eclipse. Wie schon XVCL ist auch pure::variants unabhängig von Programmiersprachen und lässt sich ebenfalls für die Konfigurierung von Dokumentationen verwenden. In pure::variants werden Anforderungen und Lösungen getrennt voneinander behandelt, so können diese unabhängig voneinander in weiteren Projekten eingebunden werden.
Die Grundlage von pure::variants bilden 4 Modelle:

- Feature Model (Kapitel 3.3.1)
- Family Model (Kapitel 3.3.2)
- Variant Description Model (Kapitel 3.3.3)
- Result Model (Kapitel 3.3.4)

Das Feature Model enthält eine Menge von Merkmalen und bildet damit den Problemraum ab. Hier liegt ein großer Vorteil von pure::variants gegenüber XVCL. In XVCL kann mittels des Specification x-frames ein Problem spezifiziert werden. Es gibt aber keine Möglichkeit, den gesamten Problemraum darzustellen. In pure::variants ist dies mit dem Feature Model möglich. Das Family Model bildet den Lösungsraum ab und ist vergleichbar mit dem x-framework aus XVCL. Das Variant Description Model spezifiziert eine konkrete Merkmalauswahl für ein einzelnes Problem aus dem Problemraum, wohingegen das Result Model eine konkrete Lösung aus der Sicht des Family Models darstellt.
Um aus den vorgestellten Modellen zu einem spezifizierten Problem eine konkrete Lösung zu erhalten, wird die Transformation genutzt. Im Falle von Software generiert die

Transformation abhängig von den gewählten Merkmalen automatisch den gewünschten Quellcode. Die Transformation unterstützt hauptsächlich die Erstellung von Dateien, das Zusammensetzen von Code-Fragmenten und Ersetzungen im Quelltext. Neben der beschriebenen Default-Transformation können vom Anwender auch eigene Transformationen erstellt und benutzt werden. So können beispielsweise externe Programme in die Transformation eingebunden werden.
Das bereits bekannte Beispiel aus Kapitel 2.1 wird im Folgenden genutzt, um die vier Modelle detaillierter vorzustellen.

3.3.1 Feature Model

Im Feature Model können die Merkmale einer ganzen Produktlinie angegeben werden. Merkmale bzw. Features sind in Abbildung 3.5 mit einem „F“ gekennzeichnet. Es gibt für Merkmale vier Typen von Eigenschaften, die im Feature Model mit unterschiedlichen Symbolen dargestellt werden. Das Ausrufezeichen steht für *Mandatory* und bedeutet, dass das Merkmal ausgewählt werden muss. Der grüne Doppelpfeil steht für *Alternative* und erlaubt nur die Auswahl eines Merkmals aus einer Gruppe. Zum Beispiel muss entweder ein Benziner oder ein Diesel gewählt werden, um eine gültige Merkmalauswahl zu erhalten. Das Kreuz steht für *Or* und lässt eine Mehrfachauswahl von Merkmalen zu. Im Beispiel können beide Reifenarten gewählt werden, es muss aber mindestens eine Art ausgewählt werden. Das Fragezeichen steht für *Optional* und stellt die Auswahl dieser Merkmale frei.

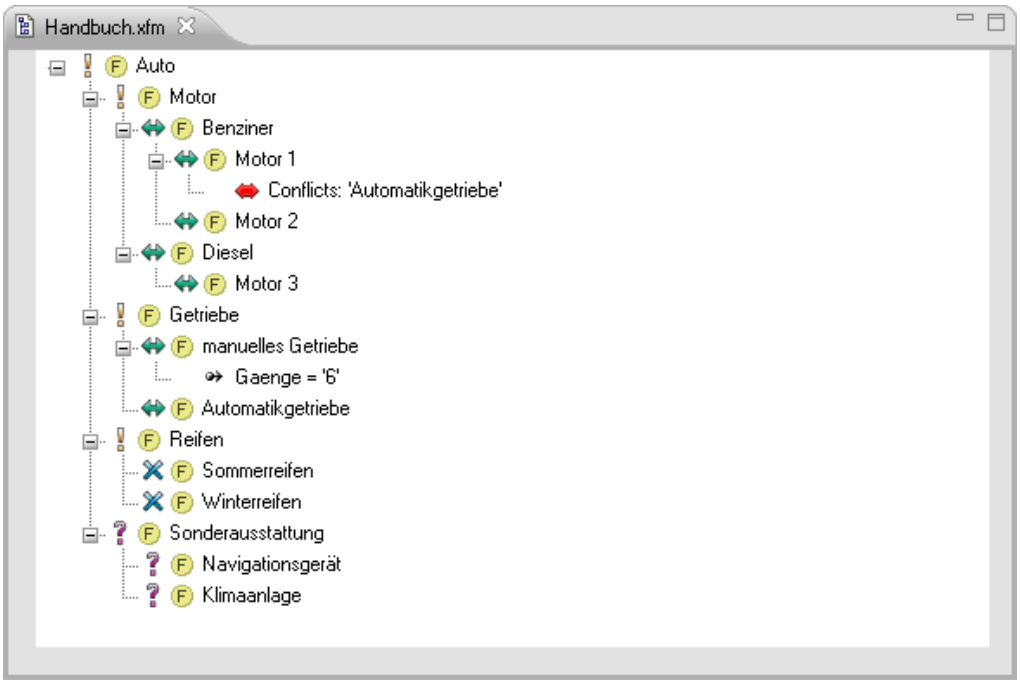

Abbildung 3.5: Feature Model

Zusätzlich können zwischen Merkmalen verschiedene Arten von Beziehungen angegeben werden. So ist in Abbildung 3.5 zwischen Motor 1 und Automatikgetriebe ein Konflikt definiert. Dieser Konflikt lässt sich nur lösen, wenn eines oder keines der beiden Merkmale ausgewählt ist. Neben Konflikten können noch weitere Arten von Beziehungen genutzt werden, wie z.B. *Required*. Diese Beziehung gibt an, dass ein anderes Merkmal vorausgesetzt wird.
Merkmalen können außer Beziehungen auch noch Attribute zugewiesen werden. Die Attribute lassen sich z.B. im Verlauf der Transformation für Textersetzungen nutzen. So könnten beispielsweise für Variablen Initialisierungswerte im Quelltext eingesetzt werden.

3.3.2 Family Model

Das Family Model gibt den Lösungsraum an. Im Falle des hier vorgestellten Beispiels enthält das Family Model alle Textteile, die zum letztendlichen Handbuch hinzugefügt werden könnten. Auf oberster Ebene des Family Models werden die logischen Komponenten des Systems angegeben. Komponenten können wiederum Komponenten oder sogenannte Teile enthalten. Jedes Teil kann weitere Teile oder auch Quelltext beinhalten.

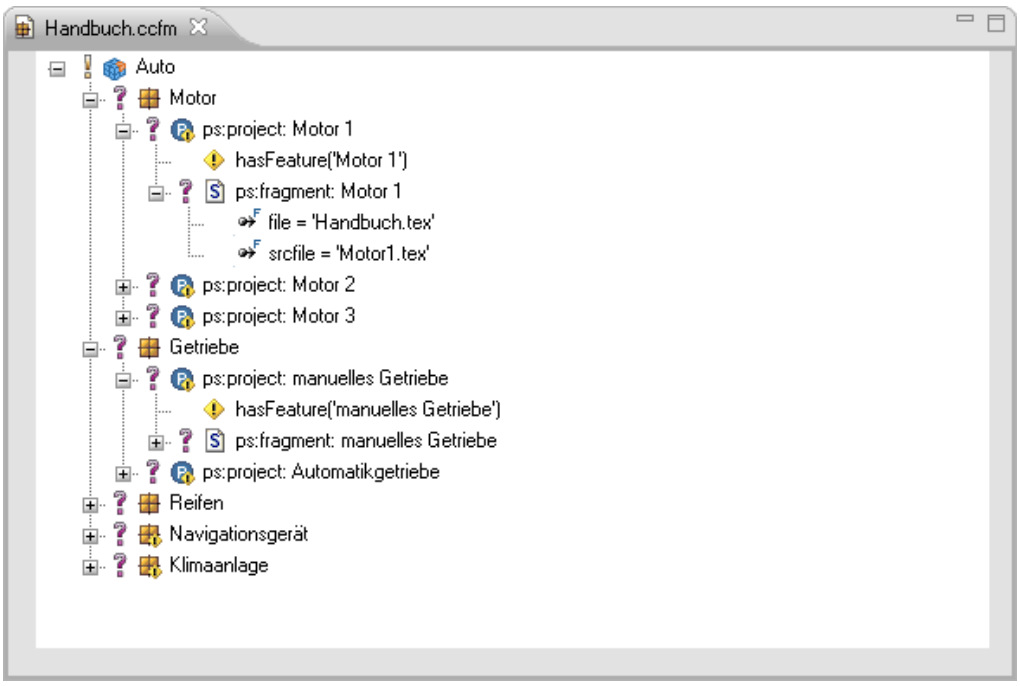

Abbildung 3.6: Family Model

Verknüpft wird das Family Model mit den Merkmalen über Restriktionen. Mit Restriktionen kann angegeben werden, welche Merkmale ausgewählt bzw. nicht ausgewählt sein müssen, damit eine Komponente oder ein Teil aus dem Family Model in das Result

Model übernommen werden. Wird für eine Komponente oder ein Teil keine Restriktion angegeben, so wird diese Komponente oder das Teil immer in das Result Model übernommen. Restriktionen sind in pure::variants mit einem schwarzen Ausrufezeichen auf gelbem Grund gekennzeichnet.

3.3.3 Variant Description Model

Anhand der im Variant Description Model gewählten Merkmalkombinationen wird entschieden, welche Komponenten und Features in den Quellcode und damit letztendlich auch in die Produktvariante integriert werden. Während der Auswahl einzelner Merkmale in diesem Modell überprüft pure::variants die Korrektheit der Auswahl. Falls Konflikte auftreten, werden diese, soweit es möglich ist, von pure::variants selbstständig behoben und ansonsten dem Anwender über eine Fehlermeldung angezeigt. In Abbildung 3.7 wird solch ein Konflikt am linken Bildrand angezeigt. In diesem Fall wurde eine ungültige Auswahl an Merkmalen gewählt. Um diesen Konflikt zu lösen, muss ein anderer Motor oder ein anderes Getriebe gewählt werden.

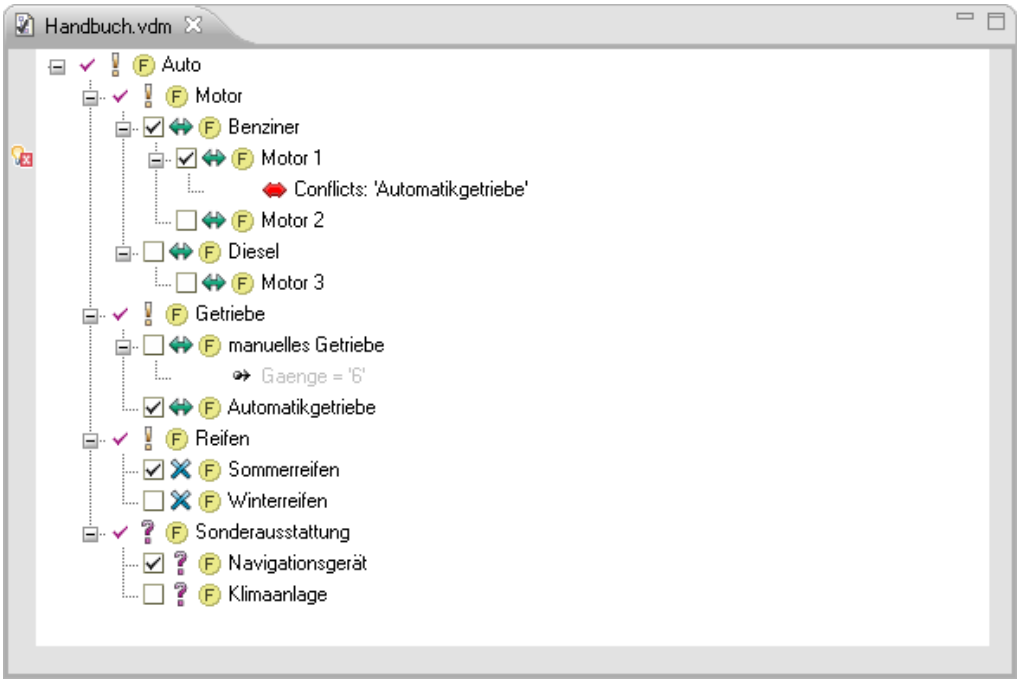

Abbildung 3.7: Variant Description Model

3.3.4 Result Model

Während das Variant Description Model angibt welche Merkmale ausgewählt sind, gibt das Result Model anhand der ausgewählten Merkmale an, welche Komponenten des Lösungsraums in die Lösung integriert werden. Alle Komponenten, auf die die Merkmalauswahl zutrifft, sind in Abbildung 3.8 mit einem Häkchen versehen und werden im Zuge der Transformation für die Erzeugung der Lösung benutzt.

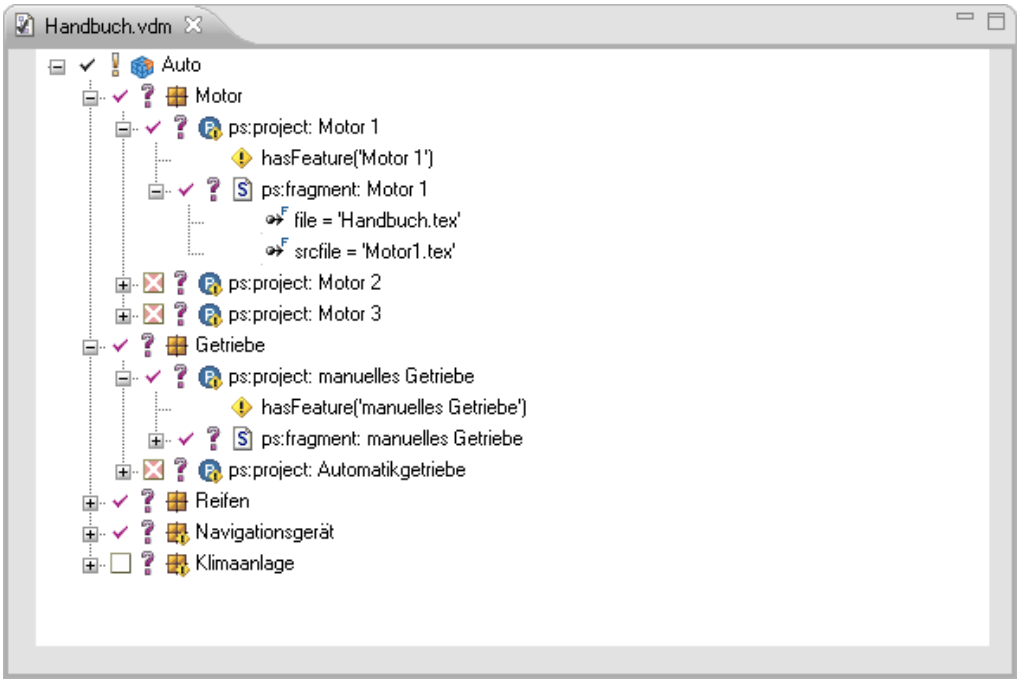

Abbildung 3.8: Result Model

4 Related Work

MPSoCs sind ein aktuelles Forschungsgebiet, in dem viele Untersuchungen im Hinblick auf die verwendeten Kommunikationsstrukturen angestellt werden. Es gibt Ansätze, die eher klassische Bussysteme zur Kommunikation verwenden, aber vor allem das viel versprechende Paradigma der NoCs (Networks on Chip) wird untersucht [19]. Von NoCs erhofft man sich, neben einer besseren Skalierbarkeit der Systeme, auch einen höheren Durchsatz bei der Kommunikation [7]. Außerdem rückt die Konfigurierung von MPSoCs mehr und mehr in den Fokus der Forschung. Im Folgenden werden einige Arbeiten in diese Richtungen vorgestellt.

In [20] wird ein MPSoC mit einem hierarchischen Bus-System vorgestellt. In diesem Ansatz werden ARM Soft Cores eingesetzt, welche über die Advanced Microcontroller Bus Architecture verbunden werden. Diese Architektur besteht aus einem Peripherie-Bus, in Abbildung 4.1 als Bus-Layer1 dargestellt, und einem Advanced High-performance Bus (Bus-Layer2). Der Peripherie-Bus verbindet den Soft Core mit seinem lokalen Speicher und der Bus-Bridge. Die Bus-Bridge stellt die Schnittstelle zwischen dem Bus-Layer1 und dem Bus-Layer2 dar. Neben vier ARM Soft Cores hängt ein Shared Memory zum Datenaustausch zwischen den Cores am Bus-Layer2. Um atomare Operationen auf dem Shared Memory zu unterstützen, ist zusätzlich ein Semaphore-Controller mit dem Bus-Layer2 verbunden. Dieser erlaubt den Soft Cores exklusiven Zugriff auf die gemeinsamen Daten.

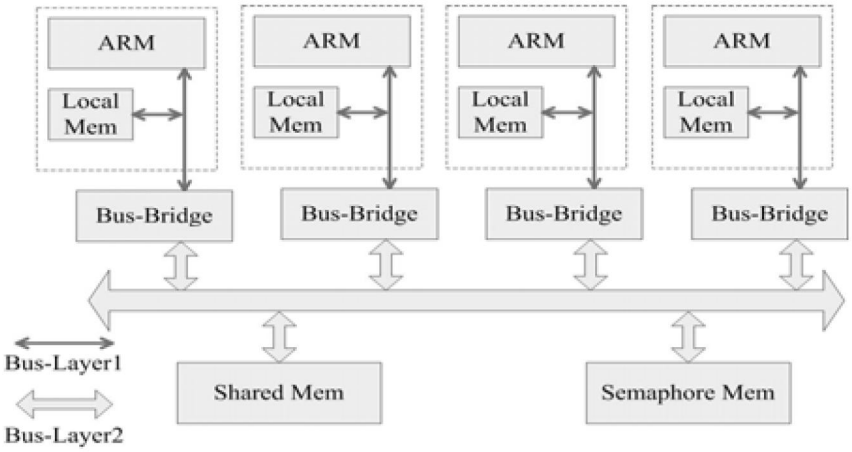

Abbildung 4.1: Architektur (Quelle: [20])

In [7] wird ein rudimentär-konfigurierbares MPSoC mit einem Crossbar-Netzwerk präsentiert. Das Crossbar-Netzwerk (Abbildung 4.2) hat eine NxN Topolgie und kann für zwei bis acht Cores konfiguriert werden. Über die Oberfläche einer Java-Anwendung werden die beiden zur Konfigurierung benötigten Parameter, die Anzahl der Knoten und die Datenbreite, übergeben. Aus den Eingaben wird automatisch ein Netzwerk in CASM (Channel-based Algorithmic State Machine) erzeugt. CASM ist eine Eigenentwicklung der Autoren und ermöglicht dem Nutzer Hardware auf einer höheren Ebene als VHDL zu beschreiben. Über den CASM-Compiler wird dann der VHDL-Code generiert. Die Autoren vergleichen die Performance ihres Crossbar-Netzwerkes mit der theoretisch maximalen Datenrate eines herkömmlichen Busses. Dabei erreichen sie mit dem Crossbar-Netzwerk und acht Cores eine Datenrate von 4,91 Wörtern pro Zyklus. Bei einem herkömmlichen Bus kann jeweils nur ein Core kommunizieren, womit im Idealfall nur ein Wort pro Zyklus übertragen werden kann.

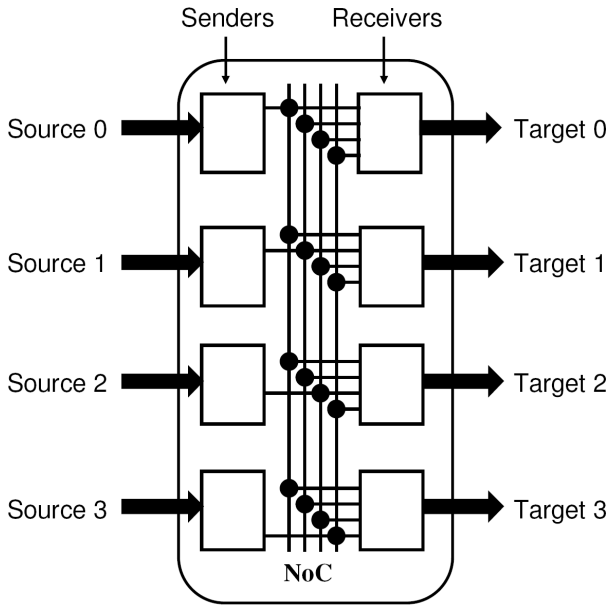

Abbildung 4.2: Netzwerk-Architektur für vier Cores (Quelle: [7])

In [21] wird auf Basis des Embedded Development Kit von Xilinx ein Framework zur Generierung eines MPSoCs vorgestellt. Dabei wird der vollständige Designprozess von der Beschreibung des Systems bis zur Erstellung des Bit-Files unterstützt. Zur Kommunikation zwischen den Komponenten des MPSoCs wurde ein NoC vorgesehen. Der eingesetzte Prozessor ist der konfigurierbare Soft Core MicroBlaze von Xilinx. Desweiteren wurde eine Shared Memory Komponente entwickelt, die sowohl in der Breite als auch in der Tiefe konfiguriert werden kann. Über zwei verschiedene Netzwerkinterfaces können die MicroBlaze Cores und der Shared Memory mit dem Netzwerk verbunden werden.

Der MicroBlaze Core wird über den standardmäßig integrierten FSL-Bus[1] mit dem Netzwerkinterface verbunden. Das Netzwerkinterface partitioniert die erhaltenden Daten in Pakete und sendet diese weiter an den Router. Der Router ermittelt den Empfänger des Pakets und leitet es an den entsprechenden Ausgang zum zugehörigen Netzwerkinterface. Die Anzahl der Ein- und Ausgänge kann für den Router konfiguriert werden. Nach [21] kann das Embedded Development Kit bereits zum Konfigurieren und Generieren von Systemen mit unterstützten Komponenten, wie dem MicroBlaze Soft Core, genutzt werden. Jedoch funktioniert dies nicht für Eigenentwicklungen, die über nicht Standardkonforme Interfaces miteinander verbunden sind. Dieses Problem wurde mit einer tcl-Skriptsprache behoben, die ein Projekt für das Embedded Development Kit erstellt, die nötige Ordnerstruktur anlegt und ein Makefile zur Synthese erzeugt. Die Angaben zur Konfigurierung können über ein auf text-basierendes Tool bestimmt werden. Zwei Konfigurationen wurden auf ihre Funktion und auf den Ressourcen-Verbrauch auf einem Xilinx Virtex-II Pro untersucht. Die erste Konfiguration (Abbildung 4.3) nimmt auf dem FPGA 3.527 Slices ein und besteht aus einem Router und drei MicroBlaze Soft Cores. Die zweite Konfiguration (Abbildung 4.4) besteht aus einem Router, zwei MicroBlaze Soft Cores und einer Shared Memory Komponente. Diese Konfiguration benötigt 4.142 Slices, was ca. 30 Prozent der verfügbaren Ressourcen des Virtex-II Pro entspricht. Für den Funktionstest wurden Szenarien zur Kommunikation erstellt und mittels Debug-Informationen über eine serielle Schnittstelle ausgewertet.

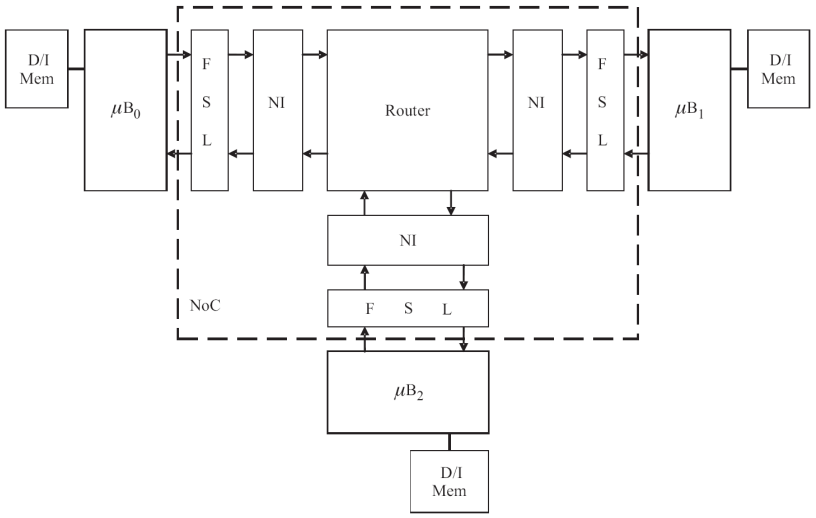

Abbildung 4.3: Architektur für drei Cores (Quelle: [21])

Ein ebenfalls konfigurierbares MPSoC wird in [22] vorgestellt. Der Fokus bei diesem Paper liegt auf einem schnell zu konfigurierenden MPSoC bei dem Echtzeit-Bedingungen eingehalten werden können. Dabei handelt es sich um ein Multiprozessor NoC, das aus

[1] unidirektionale Punkt-zu-Punkt-Verbindung zwischen zwei Komponenten

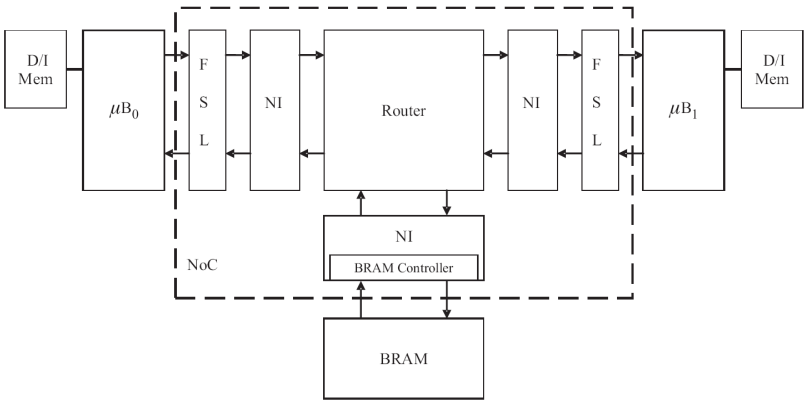

Abbildung 4.4: Architektur für zwei Cores und einen Shared Memory (Quelle: [21])

konfigurierbaren Soft Cores von Silicon Hive besteht, die über das Æthereal-Netzwerk[2] miteinander verbunden sind. Die vorgegebenen Echtzeit-Bedingungen können aufgrund des eingesetzten TDMA-Verfahrens eingehalten werden, bei dem jedem Core ein garantierter Zeit-Slot zugewiesen wird. Sowohl die Soft Cores, als auch das Netzwerk werden in den zwei unterschiedlichen Design-Prozessen der jeweiligen Hersteller erzeugt. Für die Soft Cores von Silicon Hive kann unter anderem die Anzahl der Cores, der lokale Speicher oder neue Instruktionen konfiguriert werden. Im Design-Prozess für das Æthereal-Netzwerk kann, abhängig von den Anforderungen der Applikation, das Netzwerk z.B. nach Bandbreite, Verbindungen oder Topologie konfiguriert werden. Diese beiden Design-Prozesse wurden in diesem Paper zu dem in Abbildung 4.5 dargestellten Design-Prozess verschmolzen. Die Applikation wird dabei vom Benutzer zerlegt und in den Anforderungen für die Kommunikation ausgedrückt. Aus diesen Anforderungen werden der VHDL-Code für das Netzwerk und der Code für die Konfigurierung des Netzwerks erzeugt. Der Quellcode der zuvor partitionierten Applikation und der Quellcode zur Konfigurierung des Netzwerks werden genutzt, um mittels des Silicon Hive Design-Prozesses auch den VHDL-Code für die Soft Cores zu generieren. Der aus den Design-Prozessen erzeugte VHDL-Code kann anschließend zu einem MPSoC kombiniert werden. In Abbildung 4.6 ist das bisher generierte und getestete MPSoC dargestellt. Das MPSoC setzt sich aus drei Soft Cores und dem Netzwerk zusammen. Zwei Soft Cores laufen dabei mit dem partitionierten Applikations-Code und der dritte Soft Core verwendet den Code zur Konfiguration des Netzwerks.

Einige der hier gezeigten Ansätze bieten zwar die Möglichkeit, ein MPSoC abhängig von verschiedenen Merkmalen automatisch zu generieren, aber eine durchgängie Konfigurierung mit gleichen Techniken, sowohl von der Hardware als auch von der zugehörigen Systemsoftware ist mit diesen Ansätzen nicht zu realisieren. Genau an diesem Punkt setzt die Diplomarbeit an und untersucht für die Konfigurierung von MPSoCs die bereits bekannten Mechanismen zur Konfigurierung von Software. Dieser Schritt ist insbeson-

[2]entwickelt von der Embedded Systems Architecture on Silicon-Group bei Philips

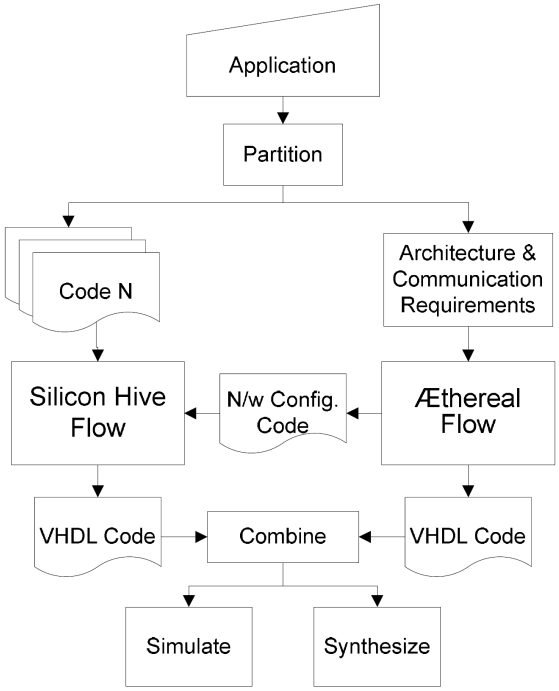

Abbildung 4.5: verschmolzener Design-Prozess (Quelle: [22])

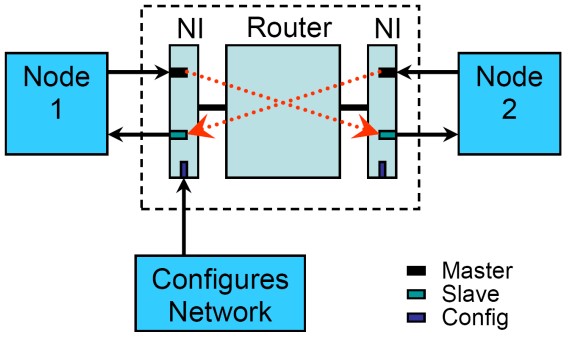

Abbildung 4.6: Architektur (Quelle: [22])

dere durch die Hardwarebeschreibungssprachen, wie VHDL oder Verilog plausibel, die eine programmiersprachen-ähnliche Beschreibung von Hardware erlauben. Desweiteren zeigen die Paper einen Querschnitt durch die verwendeten Mechanismen und Topologien zur Kommunikation. Für ein konfigurierbares System sollten idealerweise möglichst viele Mechanismen und Topologien zur Verfügung stehen. So kann für die jeweilige Anwendung eine adäquate Kommunikationsstruktur konfiguriert werden.

5 Entwurf des MPSoCs

In diesem Kapitel werden die Arbeiten an dem später zu konfigurierenden MPSoC vorgestellt. Bereits während des Aufbaus des MPSoCs mussten Überlegungen für die spätere Konfigurierung in den Designprozess einbezogen werden, um eine möglichst modulare Anbindung für verschiedene IPC-Mechanismen und Kommunikationsstrukturen zu konstruieren. Das MPSoC ist in der Hardwarebeschreibungssprache VHDL [23] entworfen.

In Kapitel 5.1 wird der für das MPSoC benutzte Soft Core vorgestellt. Anschließend wird der Aufbau des MPSoCs in Kapitel 5.2 und die Funktionsweise der Kommunikation in Kapitel 5.3 beschrieben. Abschließend wird in Kapitel 5.4 auf die Programmierung des MPSoCs und den Austausch des Programmspeichers eingegangen.

5.1 Soft Core

Begonnen hat die Arbeit am MPSoC mit der Suche nach einem frei verfügbaren und geeigneten Soft Core. Die Wahl fiel dabei auf die ZPU [24] von Øyvind Harboe. Die ZPU ist eine kleine stack-basierte 32 Bit CPU mit 8 Bit breitem Opcode. Sie hat einen einfachen und übersichtlichen Aufbau und ist deshalb optimal für die hier angestrebten Erweiterungen geeignet. Der geringe Ressourcenverbrauch von ca. 550 Slices auf dem Spartan-3E erlaubt zudem viele Cores auf einem Chip unterzubringen. Eine detaillierte Auswertung des Ressourcenverbrauchs findet sich in Kapitel 7.2. Des weiteren gibt es für die ZPU eine GCC-Toolchain, so dass sich Programme für die ZPU in C bzw. C++ schreiben lassen.

Abbildung 5.1 zeigt den schematischen Aufbau der ZPU in der „small“-Variante. Im Folgenden werden die wesentlichen der hier gezeigten Teile der ZPU vorgestellt. Außerdem werden die zur Realisierung von Kommunikation zwischen den ZPUs vorgenommenen Modifikationen an der ZPU präsentiert.

5.1.1 Kern

Der eigentliche Kern, also die Zustandsmaschine der ZPU, durchläuft bei der Abarbeitung seiner Instruktionen im Wesentlichen folgende drei Phasen:

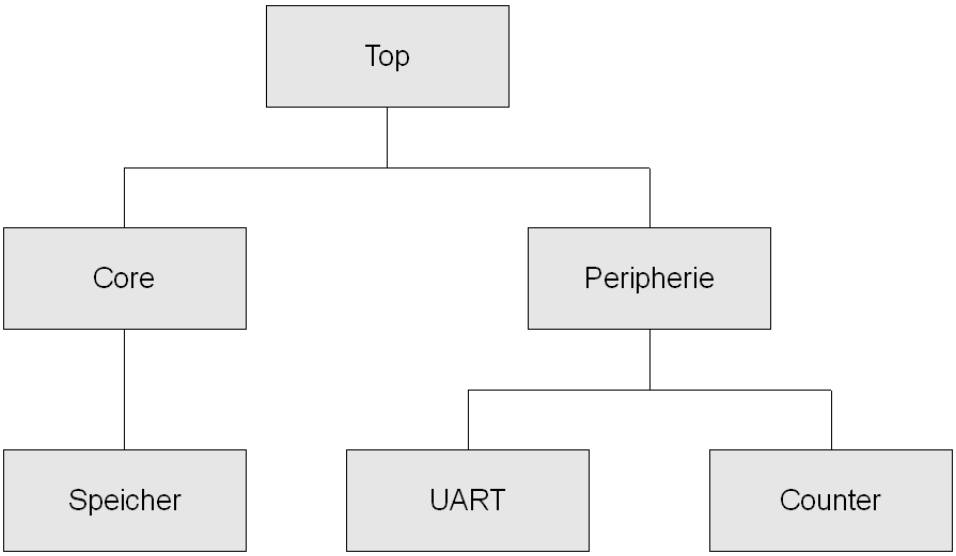

Abbildung 5.1: schematischer Aufbau der ZPU

- **Fetch / FetchNext:**
  In diesen Zuständen wird anhand des Program Counters nicht nur eine Instruktion, sondern ein kompletter 32 Bit Wert aus dem Speicher geholt.

- **Decode:**
  Da die Opcodes nur 8 Bit breit sind, wird anhand der letzten zwei Bits des Program Counters entschieden, welche 8 Bit des Speicherwertes dekodiert werden.

- **Execute:**
  Abhängig vom dekodierten Opcode springt die ZPU in der Execution-Phase in verschiedene Zustände, um die Instruktionen abzuarbeiten. Des Weiteren wird der Program Counter um Eins erhöht.

5.1.2 Speicher

Der Speicher der hier verwendeten ZPU ist ein dualport Block-RAM. Über beide Ports kann gleichzeitig lesend oder schreibend auf das Block-RAM zugegriffen werden. Natürlich kann dabei über die beiden Ports nicht zur selben Zeit schreibend auf die gleiche Speicheradresse zugegriffen werden. Das Block-RAM dient der ZPU sowohl als Programm, als auch als Datenspeicher. Zurzeit ist eine Speichergröße von 16 KB vorgegeben, die Gründe hierfür werden im weiteren Verlauf dieses Kapitels erläutert.

5.1.3 Peripherie

Die Peripherie der ZPU lässt sich über Memory Mapped I/O ansprechen. Memory Mapped I/O bedeutet, dass der Core mit der Peripherie direkt über Speicheradressen kommunizieren kann. Bei einem Schreibzugriff auf solch eine Adresse wird der Wert nicht in den Speicher geschrieben, sondern an die entsprechende Peripherie weitergeleitet. Dies hat zum einen den Vorteil, dass sich die ZPU, abhängig von den verfügbaren Adressen, leicht um weitere Peripherie erweitern lässt und zum anderen, dass die Kommunikation mit der Peripherie über einfache Speicherzugriffe realisiert werden kann. So wird kein extra Protokoll zur Kommunikation zwischen Core und Peripherie benötigt.

In der ursprünglichen ZPU ist bereits ein einfacher 64-Bit Zähler und eine ausschließlich zu Simulationszwecken bestimmte UART-Schnittstelle integriert. Salvador E. Tropea hat in seiner ZPU-Variante Namens Zealot [24] eine „echte“ UART-Schnittstelle umgesetzt. Da auch für das MPSoC eine auf einem FPGA funktionierende UART-Schnittstelle zur Visualisierung von Abläufen im MPSoC verwendet werden soll, wurde der I/O-Teil der Zealot-Variante in dieses Projekt integriert.

Im Folgenden eine Übersicht über die im MPSoC eingesetzte Peripherie und deren Speicheradressen:

I/O-Speicheradresse	Peripherie
0x080a0008	LEDs
0x080a000c	UART TX
0x080a0010	UART RX
0x080a0014	Counter, Bit 31 bis Bit 0
0x080a0018	Counter, Bit 63 bis Bit 32

Tabelle 5.1: I/O-Speicheradressen

Über die LED-Peripherie lassen sich zurzeit, abhängig vom verwendeten FPGA-Entwicklungsboard, bis zu sechs LEDs ansteuern. Zur Ansteuerung der LEDs wird an deren I/O-Adresse ein Zahlenwert zwischen 0 und 2^6 - 1 geschrieben, wobei jedes Bit direkt eine LED ansteuert.
Für die UART-Schnittstelle lässt sich die Baudrate im Soft Core konfigurieren. Dabei ist die verwendete Kodierung auf acht Datenbits, keine Paritätsbits und ein Stopbit (8N1) festgelegt. Ein Zeichen lässt sich über die UART-Schnittstelle übertragen, indem an die Adresse 0x080a000c ein Datenbyte geschreiben wird. Durch das Lesen des neunten Bits an dieser Adresse lässt sich abfragen, ob der Puffer zur Übertragung von Nachrichten zurzeit voll ist.
Der verwendete 64 Bit Zähler wird bei jeder positiven Flanke des Taktes um Eins erhöht. Die Angabe eines Prescalers oder das Vorladen des Zählers ist nicht möglich. Ein Reset des Zählers erfolgt durch das Schreiben einer Eins an einer seiner beiden I/O-Adressen.

Durch das Senden einer Zwei wird der aktuelle Wert des Zählers zwischengespeichert. So ist es möglich, den 64 Bit Wert des Counters in zwei Schritten an den Core zu übermitteln. Zum Lesen der niederwertigen 32 Bits muss der Core einen Lesezugriff auf Adresse 0x080a0014 ausführen, zum Lesen der höherwertigen 32 Bits auf Adresse 0x080a0018.

5.1.4 Kommunikationsschnittstelle der ZPU

Ein zentraler Punkt bei dem Aufbau des MPSoC ist die Kommunikation zwischen den Cores. Da die ZPU in ihrer ursprünglichen Form nur ein einzelner Prozessor ist, musste die ZPU für den Multiprozessorbetrieb erweitert werden. Zur Realisierung der Kommunikation zwischen den Cores wurde der Befehlssatz um zwei Befehle ergänzt, ein Befehl zum Empfangen und der zweite zum Senden von Nachrichten. Da bereits der gesamte Opcode-Bereich der ZPU belegt ist, wurden vorerst die Opcodes der Division und der Modulo-Rechnung für die Kommunikation verwendet. Der Opcode *0x35* wird zum Empfangen von Nachrichten, *0x36* wird zum Senden von Nachrichten genutzt.
Dabei wurde die Entscheidung getroffen, die Cores nicht direkt miteinander kommunizieren zu lassen, sondern für jeden Core einen IPC-Controller vorzuschalten. Dies hat den einfachen Nutzen, dass die Cores keine weitere Rechenzeit für die Abarbeitung des Kommunikationsprotokolls bzw. das Abhören des Busses verlieren. Folgend werden die internen Abläufe im Core bei der Abarbeitung des Empfangs- bzw. Sende-Befehls sowie die Kommunikation zwischen dem IPC-Controller und dem Core beschrieben.
Zur Abarbeitung dieser Befehle wurden die neuen Zustände *State_ReceiveBusRead*, *State_ReceiveBusWait*, *State_SendBusWrite* und *State_SendBusDone* in die ZPU integriert.

Wird der Opcode zum Empfangen von Nachrichten dekodiert, geht der Core in den Zustand *State_ReceiveBusRead* über. In diesem Zustand wird das oberste Element vom Stack genommen und dem IPC-Controller eine Leseanfrage signalisiert. Das vom Stack genommene Element enthält, abhängig vom verwendeten IPC-Mechanismus, Informationen wie z.B. die Adresse der zu lesenden Speicherzelle. In diesem Zustand bleibt der Core genau einen Taktzyklus und springt danach in den Zustand *State_ReceiveBusWait*, um auf die Antwort des IPC-Controllers zu warten. Sobald der IPC-Controller mit der angeforderten Nachricht antwortet, werden die empfangenden Daten sowie die Adresse des Senders auf dem Stack abgelegt. Im Falle von IPC-Mechanismen, wie z.B. Shared Memory, bei denen es keinen direkten Absender der Nachrichten gibt, wird eine Null für die Adresse des Absenders auf dem Stack abgelegt. Abhängig vom verwendeten IPC-Mechanismus hat der Core den Befehl im Idealfall innerhalb von acht Taktzyklen abgearbeitet.

Zum Senden einer Nachricht springt der Core zuerst in den Zustand *State_SendBusWrite*. In diesem Zustand nimmt der Core das oberste Stack-Element mit den Daten und das zweite Element mit der Empfängeradresse bzw. der Speicherzellenadresse vom Stack und setzt zur Benachrichtigung des IPC-Controller ein Write-Enable Signal. Nach ge-

nau einem Taktzyklus verlässt der Core diesen Zustand und springt in den Zustand *State_SendBusDone*. Hier wird nur noch das Write-Enable Signal zurückgesetzt. Für diesen Befehl werden zur vollständigen Ausführung, unabhängig vom IPC-Mechanismus, sieben Taktzyklen benötigt.

5.2 Struktur des MPSoC

Nachdem eine geeignete Grundlage für das MPSoC in Form der ZPU gefunden und diese mit einer Schnittstelle für den Einsatz in einem MPSoC ausgerüstet wurde, musste eine Struktur zur Kommunikation oberhalb der Soft Cores entworfen werden. An erster Stelle stand dabei die spätere Konfigurierbarkeit. Der Fokus lag hierbei auf einem möglichst flexiblen System, das es erlaubt, beliebig viele Cores einzubinden sowie verschiedene IPC-Mechanismen einzusetzen, die sich ohne große Änderungen auswechseln lassen. Die daraus entstandene Struktur ist in Abbildung 5.2 dargestellt.

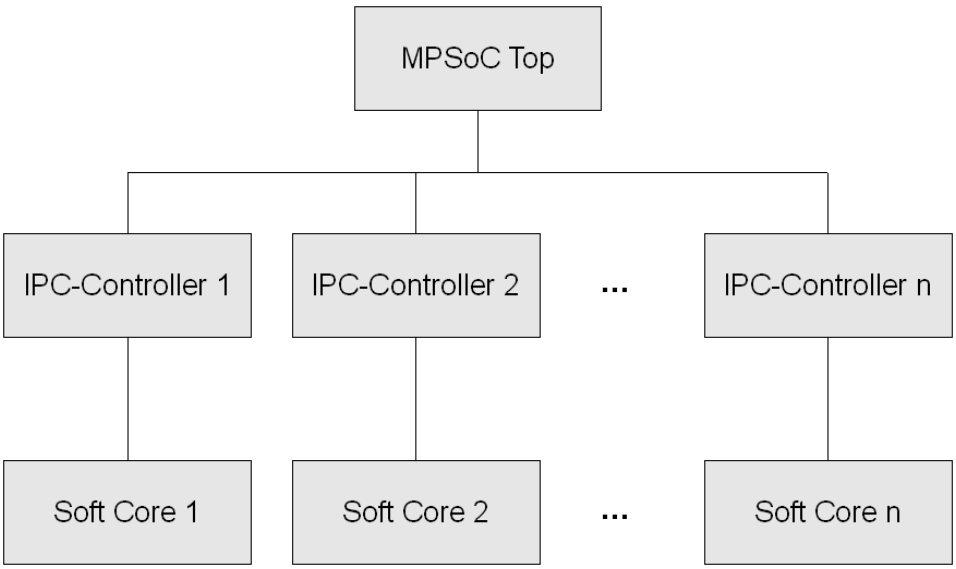

Abbildung 5.2: Struktur des MPSoC

Auf unterster Ebene liegen die bereits im vorherigen Kapitel beschriebenen Soft Cores. Die zu konfigurierende Anzahl der eingesetzten Soft Cores hängt allein von den Ressourcen des FPGAs bzw. dessen Block-RAM-Größe ab. Jeder Core ist mit einem eigenen IPC-Controller verbunden. Der IPC-Controller ist, nachdem die zu übermittelnde Nachricht vom Core an den IPC-Controller übergeben wurde, für die gesamte Kommunikation zwischen den Cores zuständig. Er puffert die Nachrichten, hört den Bus nach relevanten Nachrichten ab, legt die Nachrichten auf den Bus und achtet auf Kollisionen. Die Art und Weise wie der IPC-Controller arbeitet, hängt vom eingesetzten

IPC-Mechanismus ab. So ist es möglich, durch Austausch der IPC-Controller einen anderen IPC-Mechanismus einzusetzen, ohne dabei Änderungen an den Cores vornehmen zu müssen. An oberster Stelle steht das MPSoC Top-Modul. Dieses Modul verbindet die I/O-Pins des FPGAs, wie z.B. Takt, Reset-Schalter oder die UART-Signale, mit den Leitungen der entsprechenden Cores. Des Weiteren werden im MPSoC Top-Modul, abhängig von der konfigurierten Kommunikationsstruktur, die IPC-Controller der Cores untereinander zusammengeschlossen.

Durch diesen Aufbau besteht später die Möglichkeit sowohl verschiedene Soft Cores als auch verschiedene IPC-Mechanismen in einem MPSoC einzusetzen. So wäre ein heterogenes MPSoC denkbar, bei dem beispielsweise drei ZPU-Cores über Message Queues und zwei spezialisierte Cores über Shared Memory kommunizieren können.

5.3 Kommunikation im MPSoC

Für die Leistungsfähigkeit eines MPSoCs ist der Einsatz eines adäquaten Kommunikationsmechanismus entscheidend. Oftmals bildet gerade die Kommunikation einen Flaschenhals, welcher das ganze System ausbremst [20]. Da je nach Anwendung die Anforderung an die Kommunikation andere Schwerpunkte aufweisen können, wurden für das MPSoC drei verschiedene Mechanismen umgesetzt:

- Message Queues (Kapitel 5.3.1)
- Shared Memory (Kapitel 5.3.2)
- Event Flags (Kapitel 5.3.3)

Neben den IPC-Mechanismen mussten auch Kommunikationsstrukturen entworfen bzw. umgesetzt werden. Für alle oben genannten Mechanismen wurde eine herkömmliche Bus-Struktur implementiert. Für das Message Queue Verfahren wurde zudem noch eine Ring-Struktur umgesetzt. Für das Shared Memory Verfahren ist eine Ring-Struktur eher ineffizient, da jegliche Kommunikation über das Shared Memory-Modul läuft. Bei vielen Busteilnehmern entstehen für die Kommunikation so hohe Laufzeiten. Ebenso sieht es bei Event Flags aus, mit deren Hilfe Prozesse synchronisiert werden können. Im Falle einer Ring-Struktur würde die Synchronisierung verzögert ablaufen, da die Nachrichten die Cores zu einem jeweils verschobenen Zeitpunkt erreichen.

Eine Evaluation der Kommunikationsstrukturen findet sich in Kapitel 7.3.

5.3.1 Message Queues

Zur Kommunikation zwischen den Cores werden beim Message Queue Mechanismus Adressen benutzt. Jeder Core besitzt eine eindeutige Adresse, mit der er adressiert werden kann. Wenn ein Core eine Nachricht senden möchte, übergibt er seinem IPC-Controller die Daten und die Adresse des Ziel-Cores. Ab diesem Zeitpunkt kann der Core den nächsten Befehl abarbeiten und der IPC-Controller übernimmt die weitere Kommunikation mit den anderen Cores bzw. deren IPC-Controllern. Nach dem Erhalt der Daten vom Core ordnet der IPC-Controller die Nachricht in seine Liste für ausgehende Nachrichten ein und fügt die Absenderadresse des Cores hinzu. Sobald der Bus frei ist, startet der IPC-Controller mit der Übertragung der Nachricht. Bei dem Message Queue Mechanismus werden die Nachrichten in folgender Form auf den Bus gelegt:

Datenbus			Adressbus					
			Absender			Empfänger		
Bit n	...	Bit 0	Bit m	...	Bit 0	Bit m	...	Bit 0

Tabelle 5.2: Datenformat für Message Queue

Die Anzahl der Daten- und Adressbits ist dabei nicht fest vorgeschrieben, sondern ist konfigurierbar. Die Daten werden hier, wie auch bei den nachfolgenden Mechanismen, parallel auf den Bus gelegt und nicht seriell übertragen. So ist eine schnelle und zeitnahe Übermittlung der Nachrichten erreichbar.
Für die Message Queue mit Bus-Struktur setzt während der Nachrichtenübertragung die Kollisionskontrolle ein. Die Kollisionskontrolle überprüft, ob die Daten, die auf dem Bus liegen, mit denen übereinstimmen, die gesendet werden sollen. Ist dies aufgrund einer Kollision nicht der Fall, wird die Übertragung abgebrochen und der IPC-Controller muss die konfigurierte Zeit abwarten, bis ein erneuter Versuch gestartet werden kann. Damit auch der Empfänger einen Abbruch erkennt, werden die Nachrichten für 3 Taktzyklen auf den Bus gelegt. Liegt eine Nachricht nicht 3 Taktzyklen auf dem Bus, wird die Nachricht vom Empfänger verworfen.
Im Gegensatz zum Bus wird für den Ring keine Kollisionskontrolle benötigt, da auf jedem Teilabschnitt des Rings nur ein Core senden und ein weiterer Core Nachrichten empfangen kann.
Wenn eine Nachricht auf dem Bus gelesen wird, überprüft der IPC-Controller den angegebenen Empfänger. Falls der Empfänger mit der Adresse des zugehörigen Cores übereinstimmt, wird die Nachricht in der Message Queue gespeichert. Beim Bus werden, im Gegensatz zum Ring, alle weiteren Nachrichten ignoriert. Für den Ring werden diese Nachrichten in die Liste für ausgehende Nachrichten mit aufgenommen und zum nächst möglichen Zeitpunkt im Ring weitergeleitet. Wenn der IPC-Controller einen Lese-Request vom Core erhält, nimmt der IPC-Controller die erste Nachricht aus der Message Queue, legt die Daten und Absenderadresse auf den Bus zum Core und setzt das Data-

Ready Flag für den Core. Falls die Message Queue keine Nachrichten enthält, sendet der IPC-Controller dem Core seine eigene Adresse anstelle der Absenderadresse. So erkennt der Core direkt, dass keine Daten vorliegen und kann mit der Abarbeitung des nächsten Befehls fortfahren.

Zusätzlich gibt es für diesen IPC-Mechanismus die Option, dass der Core bei leerer Message Queue auf eine neue Nachricht wartet. Diese Option kann je nach Anwendung zu diesem IPC-Mechanismus hinzu konfiguriert werden. Zu bedenken ist aber, dass der Core während dieser Zeit angehalten wird und keine weiteren Befehle abarbeiten kann. Einzusetzen ist diese Option beispielsweise bei einer Anwendung, in der die Cores auf die Daten eines weiteren Cores angewiesen sind.

Eine weitere zu konfigurierende Option ist die „Priorität beim Senden“. Entstanden ist diese Option durch die Problematik, dass, sobald der Bus frei wird, alle wartenden Busteilnehmer sofort mit dem Senden anfangen und unvermeidbar eine Kollision entsteht. Diese Kollisionen können, wie in Abbildung 5.3 dargestellt, direkt nach jeder korrekt übertragenen Nachricht wieder auftreten. Wenn die Option aktiviert wird, tritt dieses Problem nicht mehr auf, da jeder Core abhängig von seiner Priorität verschieden lange Wartezeiten bei einem frei gewordenen Bus einhalten muss. So kann der Core mit der höchsten Priorität direkt nach einem Taktzyklus mit dem Senden beginnen und der n-te Core muss n Taktzyklen warten. Sinn kann diese Option bei hohen Buslasten machen. So kommen Nachrichten von hoch priorisierten Cores zeitnah beim Empfänger an. Jedoch erhöht sich auch erheblich die Übertragungszeit für Nachrichten von Cores mit niedriger Priorität und damit langer Wartezeit.

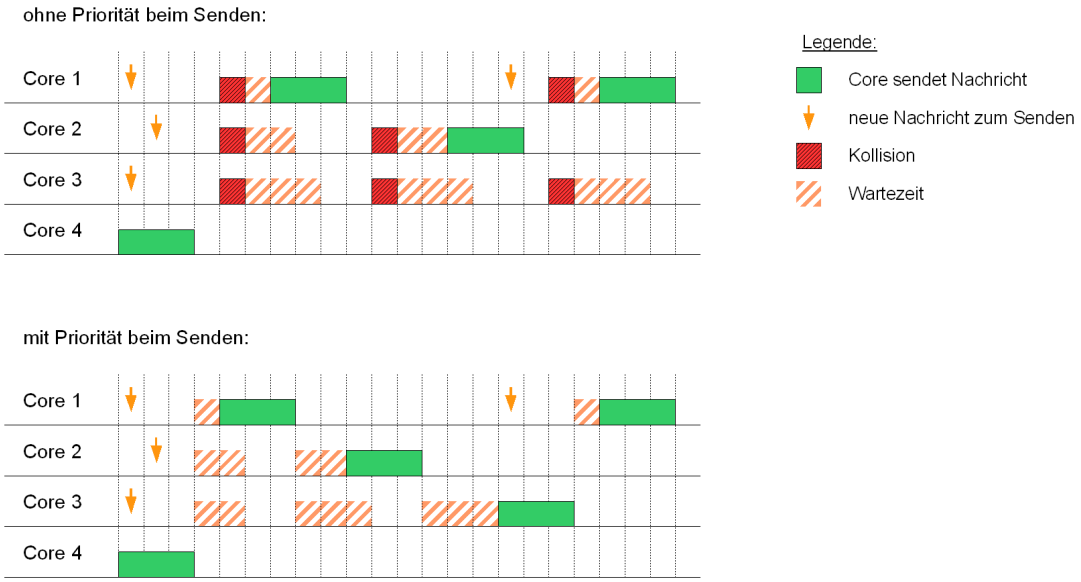

Abbildung 5.3: Priorität beim Senden

5.3.2 Shared Memory

Auch beim Shared Memory Mechanismus werden zur Kommunikation Adressen benutzt. Hier werden jedoch nicht die Cores, sondern die Speicherzellen des Shared Memory adressiert. Zusätzlich zu den IPC-Controllern hängt bei diesem Mechanismus der Controller des Shared Memory mit am Bus, um Lese- und Schreibzugriffe auf die Speicherzellen des Shared Memory zu koordinieren. Damit der Controller des Shared Memory nicht mit Nachrichten überlaufen werden kann, besitzt dieser die höchste Priorität im MPSoC, wodurch eine schnelle Abarbeitung der angeforderten Daten sichergestellt wird.

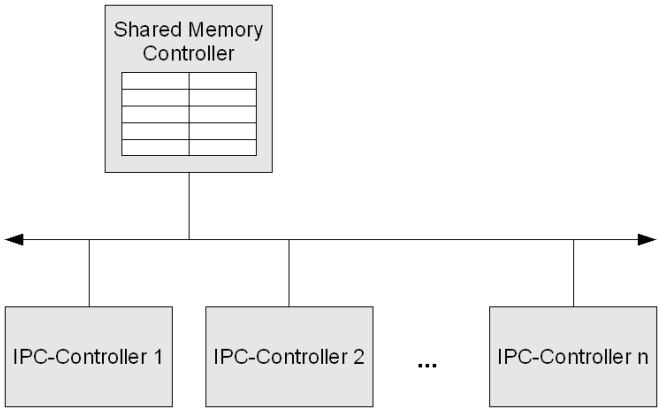

Abbildung 5.4: Shared Memory

Die Kommunikation zwischen Core und IPC-Controller läuft bei diesem Mechanismus analog zum Message Queue Mechanismus ab, jedoch wird anstelle der Empfängeradresse die Adresse der Shared Memory-Speicherzelle sowie ein weiteres Bit übermittelt. Das zusätzliche Bit wird benötigt um anzuzeigen, dass der Core einen atomaren Lese- und Schreibzugriff auf eine Speicherzelle ausführen möchte. Wird dieses Bit von einem Core bei einem Lesezugriff auf eine Speicherzelle gesetzt, sperrt der Controller des Shared Memory diese Speicherzelle für alle anderen Cores, bis auch der Schreibzugriff von dem entsprechenden Core auf diese Speicherzelle beendet wurde.
Zur Kommunikation zwischen dem Shared Memory Controller und den IPC-Controllern werden, neben der Speicherzellenadresse und dem Flag für atomare Speicherzugriffe, noch weitere Daten benötigt. Zum einen muss dem Shared Memory für einen atomaren Speicherzugriff der Absender der Nachricht bekannt sein und zum anderen muss angegeben sein, ob der Zugriff auf die Speicherzelle ein Lese- oder ein Schreibzugriff ist. Diese Daten werden von dem IPC-Controller zu den vom Core übertragenden Daten hinzugefügt. Eine vollständige Übersicht der Daten, die auf dem Bus übertragen werden, ist in Tabelle 5.3 veranschaulicht.
Der Empfang von Nachrichten ist im Verhältnis zum Message Queue Mechanismus wesentlich langwieriger, da die Daten erst über den Bus vom Shared Memory angefordert werden müssen. Hinzu kommen noch unvorhersagbare Ereignisse, wie Kollisionen oder hohe Buslast, die negativ auf die Übertragungszeit wirken. Während dieser Zeitspanne

befindet sich der Soft Core im Zustand *State_ReceiveBusWait* und kann keine weiteren Befehle abarbeiten.
Auch bei diesem Mechanismus gibt es eine Kollisionskontrolle. Da diese aber genauso wie im Falle der Message Queue arbeitet, wird an dieser Stelle nicht erneut auf die Funktionsweise eingegangen.
Wie bei der Message Queue lässt sich auch für den Shared Memory Mechanismus die Option „Priorität beim Senden“ aktivieren bzw. deaktivieren.

Datenbus			Adressbus							
Bit n	...	Bit 0	zum/ vom SM	Read/ Write	atomare Op.	Absender			Speicher-adresse		
			0/1	0/1	0/1	Bit m	...	Bit 0	Bit k	...	Bit 0

Tabelle 5.3: Datenformat für Shared Memory

5.3.3 Event Flags

Jeder IPC-Controller verwaltet für den zugehörigen Core ein Register, in dem die Zustände der Flags gespeichert sind. Die Anzahl der Flags und damit die Größe des Registers ist konfigurierbar. Bei einer Zustandsänderung von einem Flag werden die Veränderungen synchronisiert und in allen Registern der IPC-Controller zeitgleich gesetzt. So können die Anfragen der Cores umgehend und für jeden Core im System gleich beantwortet werden.
Ein Core kann ein Flag verändern, indem er seinem IPC-Controller die Adresse des Flags und den Zustand in Form einer Null bzw. einer Eins übermittelt. Der zugehörige IPC-Controller verändert das Flag nicht direkt im Register, sondern speichert die Daten zuerst zwischen. Erst wenn die Daten mit den anderen IPC-Controllern erfolgreich, also ohne eine Kollision ausgetauscht wurden, werden die Daten im Register aktualisiert. So ist sichergestellt, dass zu jedem Zeitpunkt alle Register den selben Inhalt besitzen. Über den Bus wird nicht das geänderte Flag samt zugehöriger Adresse, sondern, wie in Tabelle 5.4 zu sehen, das ganze Register ausgetauscht.

Datenbus			
Flag n	...	Flag 1	Flag 0

Tabelle 5.4: Datenformat für Event Flags

Zur Abfrage eines Flags sendet der Core dem IPC-Controller die Adresse des Flags. Da die Zustände der Flags in jedem IPC-Controller in aktueller Form vorliegen, kann dem

Core direkt mit dem Zustand des Flags geantwortet werden.
Die Kollisionsabwicklung läuft bei diesem Mechanismus wie bei den beiden vorangegangen Mechanismen ab. Genauso lässt sich auch für die Event Flags die Option „Priorität beim Senden“ konfigurieren.

5.4 Programmierung des MPSoC

In den folgenden Unterkapiteln wird der Einsatz der GCC-Toolchain erläutert. Es wird die Verwendung der IPC-Befehle, das Ansprechen der Peripherie in C und der Austausch des Programmcodes in einem bereits synthetisierten MPSoC beschrieben.

5.4.1 GCC-Toolchain

Dank der verfügbaren GCC-Toolchain kann die Software für das MPSoC direkt in C geschrieben werden. Durch Verwendung der im Anschluss aufgeführten Befehle lässt sich direkt aus dem C-Quellcode eine lauffähige Version für das Block-RAM in VHDL erzeugen. Die Konvertierung von der BIN-Datei zur BRAM-Datei übernimmt mit dem letzten Befehl das Java-Tool *ZPUSIM*. Der Inhalt der BRAM-Datei kann direkt per Copy und Paste in den VHDL-Code eingefügt werden.

```
zpu-elf-gcc -O3 -phi ‘pwd‘/code.c -o code.elf -Wl,--relax -Wl,--gc-sections  -g
zpu-elf-objdump --disassemble-all >code.dis code.elf
zpu-elf-objcopy -O binary code.elf code.bin
java -classpath ../simulator/zpusim.jar com.zylin.zpu.simulator.tools.MakeRam
  code.bin >code.bram
```

5.4.2 Programmierung in C

Die aus der Sprache C gewohnten Befehle, wie z.B. arithmetische Operationen, Schleifen oder Abfragen, können selbstverständlich für die Programmierung der Soft Cores verwendet werden. In diesem Kapitel wird jedoch nur auf die Eigenheiten des eingesetzten Soft Cores eingegangen.

5.4.2.1 IPC-Befehle

Zur Kommunikation zwischen den Cores können die Funktionen *send* und *receive* genutzt werden. Der Send-Funktion werden als erstes Argument die zu übertragenden Daten und als zweites Argument die Zieladresse übergeben. Die Zieladresse kann abhängig vom konfigurierten IPC-Mechanismus z.B. die Adresse eines Cores oder die Speicheradresse des

Shared Memory enthalten. Für die Receive-Funktion werden zwei Rückgabewerte benötigt und zwar für die Daten und für die Adresse des Absenders. Da jede Funktion nur einen Wert zurückgeben kann, wird nur die Adresse des Absenders zurückgegeben. Für die zu empfangenden Daten wird der Funktion als erstes Argument die Adresse der Variablen übergeben. So können die empfangenden Daten nach Ausführung der Receive-Funktion aus der Variablen ausgelesen werden. Das zweite übergebende Argument gibt an, von welcher Adresse gelesen werden soll. Diese Adresse wird abhängig vom eingesetzten IPC-Mechanismus verwendet, um z.B. eine Speicherzelle des Shared Memory zu adressieren. Für Message Queues ist die Angabe dieser Adresse überflüssig und kann beispielsweise auf Null gesetzt werden. Falls die Message Queue keine Nachrichten enthält und das MPSoC so konfiguriert wurde, dass nicht auf neue Nachrichten gewartet werden soll, übergibt der IPC-Controller die eigene Adresse des Cores als Absenderadresse.

```
int data, addrDest, addrSrc;
data = 87;
addrDest = 2;
send(data, addrDest);
addrSrc = receive(&data, addrDest);
```

5.4.2.2 Peripherie

Die Ansteuerung der Peripherie erfolgt, wie bereits in Kapitel 5.1.3 beschrieben, mittels Memory Mapped I/O. Dies hat neben den bereits angesprochenen Vorteilen auch noch positive Aspekte für die Programmierung. Denn die Peripherie lässt sich so sehr leicht mittels Zeigern ansprechen.

Ansteuern der LEDs:

```
volatile int* leds;                    // Zeiger auf Wert vom Typ Integer
leds = (int*)0x080a0008;               // Speicheradresse für LEDs
*leds = 0x03;                          // schaltet ersten beiden LEDs an
```

Ausgabe von Zeichen über die UART-Schnittstelle:

```
volatile int* uartTx;                  // Zeiger auf Wert vom Typ Integer
uartTx = (int*)0x080a000c;             // Speicheradresse für UART-Tx
while ((*uartTx & 0x100) == 0) {}      // warte bis Platz im Puffer frei
*uartTx = 'm';                         // Senden des Buchstaben 'm'

puts("MPSoC\n");                       // sendet direkt einen String
```

Ansteuerung und Auslesen des Counters:

```
volatile int* counter0;                    // Zeiger auf Wert vom Typ Integer
volatile int* counter1;                    // Zeiger auf Wert vom Typ Integer
counter0 = (int*)0x080a0014;               // Speicheradresse für 1. 32 Bit
counter1 = (int*)0x080a0018;               // Speicheradresse für 2. 32 Bit
volatile long long count = 0LL;            // Variable für vollst. 64 Bit
*counter0 = 0x1;                           // Reset des Timers
*counter0 = 0x2;                           // Timer samplen
count = (long long) *counter0;             // ersten 32-Bit des Counters
count |= ((long long) *counter1) << 32;    // zweiten 32-Bit des Counters
```

5.4.3 Austausch des Programmcodes

Abhängig von der Größe des Multiprozessorsystems und des verwendeten FPGAs kann die Synthese des Systems sehr langwierig sein. Damit nicht für jede Änderung am Programmspeicher der Cores das ganze System neu synthetisiert werden muss, wird von Xilinx das Kommandozeilen-Tool Data2Mem [25] in der Entwicklungsumgebung des ISE zur Verfügung gestellt.

Die Daten, die zur Konfigurierung eines FPGAs benötigt werden und damit auch die Programme der Soft Cores, befinden sich im sogenannten Bit-File. In diesem Bit-File kann Data2Mem ein altes Programm durch ein Neues ersetzen. Um diese Transformation durchzuführen, benötigt Data2Mem Informationen darüber, wo Daten ersetzt werden sollen und wie diese aufgeteilt werden. Diese Informationen werden im Block-RAM Memory Map File, dem sogenannten Bmm-File, angegeben. Das Bmm-File ist eine lesbare Textdatei und hat für das MPSoC folgenden Aufbau:

```
ADDRESS_MAP mpsoc PPC405 0
  ADDRESS_SPACE memory1 RAMB16 [0x00000000:0x00003FFF]
    BUS_BLOCK
      ipc_ctrl_1/core_top1/zpu_core_1/inst_Mram_mem7 [31:28];
      ipc_ctrl_1/core_top1/zpu_core_1/inst_Mram_mem6 [27:24];
      ipc_ctrl_1/core_top1/zpu_core_1/inst_Mram_mem5 [23:20];
      ipc_ctrl_1/core_top1/zpu_core_1/inst_Mram_mem4 [19:16];
      ipc_ctrl_1/core_top1/zpu_core_1/inst_Mram_mem3 [15:12];
      ipc_ctrl_1/core_top1/zpu_core_1/inst_Mram_mem2 [11:8];
      ipc_ctrl_1/core_top1/zpu_core_1/inst_Mram_mem1 [7:4];
      ipc_ctrl_1/core_top1/zpu_core_1/inst_Mram_mem [3:0];
    END_BUS_BLOCK;
  END_ADDRESS_SPACE;
END_ADDRESS_MAP;
```

Jedem Core stehen 16 KB (4096 x 32 Bit) Programmspeicher zur Verfügung, die in 8 Block-RAMs aufgeteilt werden. Der Adressbereich wird unter *ADDRESS_SPACE* in

Bytes angegeben. Die Aufteilung der Programmdaten in die 8 Block-RAMs ist unter *BUS_BLOCK* aufgelistet. Dabei werden nicht jeweils 512 Zeilen von 32 Bit Werten in einem Block-RAM gespeichert, sondern 4096 Zeilen mit jeweils 4 Bit des 32 Bit Wertes. Die genauen Instanznamen bzw. Pfade, wie z.B. *ipc_ctrl_1/core_top1/zpu_core_1/inst_Mram_mem*, sind mit Hilfe des Floorplaners im ISE zu ermitteln.
Nur mit diesen Angaben lässt sich das Bmm-File jedoch noch nicht für die Transformation mittels Data2Mem verwenden, da bisher die Angabe des exakten Block-RAMs fehlt. Hier empfiehlt es sich die Arbeit direkt durch das ISE, genauer gesagt durch Bitgen, erledigen zu lassen. Dazu wird das Bmm-File in das ISE-Projekt eingebunden und während des Generate Programming File-Prozesses wird das Bmm-File automatisch um die fehlenden Angaben ergänzt. Nach diesem Schritt findet sich im Projekt-Ordner ein neues Bmm-File, das im Dateinamen um „_bd“ erweitert ist und nun die Ortsangaben für alle Instanzen in folgender Form enthält:

```
ADDRESS_MAP mpsoc PPC405 0
  ADDRESS_SPACE memory1 RAMB16 [0x00000000:0x00003FFF]
    BUS_BLOCK
      ipc_ctrl_1/core_top1/zpu_core_1/inst_Mram_mem7 [31:28] PLACED = X1Y7;
      ipc_ctrl_1/core_top1/zpu_core_1/inst_Mram_mem6 [27:24] PLACED = X1Y4;
      ipc_ctrl_1/core_top1/zpu_core_1/inst_Mram_mem5 [23:20] PLACED = X1Y8;
      ipc_ctrl_1/core_top1/zpu_core_1/inst_Mram_mem4 [19:16] PLACED = X1Y6;
      ipc_ctrl_1/core_top1/zpu_core_1/inst_Mram_mem3 [15:12] PLACED = X1Y9;
      ipc_ctrl_1/core_top1/zpu_core_1/inst_Mram_mem2 [11:8] PLACED = X1Y3;
      ipc_ctrl_1/core_top1/zpu_core_1/inst_Mram_mem1 [7:4] PLACED = X1Y2;
      ipc_ctrl_1/core_top1/zpu_core_1/inst_Mram_mem [3:0] PLACED = X1Y5;
    END_BUS_BLOCK;
  END_ADDRESS_SPACE;
END_ADDRESS_MAP;
```

Neben dem Bmm-File muss Data2Mem noch das ursprüngliche Bit-File und das Programm als Elf-File übergeben werden. Mit folgendem Befehl wird die Transformation des Bit-Files ausgeführt:

```
data2mem -bm mpsoc_bd.bmm -bd code.elf -bt multicore_top.bit
```

Das Ergebnis dieser Transformation ist das neue Bit-File „multicore_top_rp.bit“ mit geändertem Programmspeicher. Es muss beachtet werden, dass immer nur bei einem Soft Core zur selben Zeit ein Programmupdate durchgeführt werden kann. Die Einträge für die anderen Cores müssen zu dieser Zeit aus dem Bmm-File entfernt bzw. auskommentiert werden. Kommentare können, wie aus der Sprache C bekannt, mit // oder /* und */ gesetzt werden.

6 Konfigurierung des MPSoCs

Die Konfigurierung soll in erster Linie den Arbeitsaufwand für die Entwicklung von MPSoCs reduzieren. So könnte ein MPSoC schnell an eine neue Problemstellung angepasst oder unkompliziert verschiedene Konfigurationen für einen Anwendungsfall getestet werden. Die Idee dabei ist, dass anhand von Merkmalen die Eigenschaften des MPSoCs festgelegt werden und anschließend über die getroffene Merkmalauswahl ein individuelles MPSoC automatisch generiert wird. In Kapitel 6.1 wird vorgestellt, welche Merkmale für das in dieser Arbeit entwickelte MPSoC konfiguriert werden können.
Neben den Merkmalen zur Beschreibung des Problems ist auch ein Blick auf den Aufbau der VHDL-Dateien des MPSoCs notwendig. Dies liegt darin begründet, dass das gesamte MPSoC mittels Textmanipulationen erzeugt wird. Dazu gehört z.B. das Zusammenkopieren von Textfragmenten oder die Ersetzungen von Schlüsselwörtern im Text. In Kapitel 6.2 wird das Konzept hinter der Fragmentierung der VHDL-Dateien präsentiert.
Das hier geschilderte Vorgehen aus Merkmalauswahl und anschließender Textmanipulation wird bereits im Software-Bereich, z.B. bei der Konfiguration von Betriebssystemen für Eingebettete Systeme, eingesetzt. Da auch Hardware, unter Einsatz von VHDL, textbasiert entworfen werden kann, ist es naheliegend, die existierenden Erfahrungen und Anwendungen auf den Hardware-Bereich zu übertragen und deren Eignung in diesem Bereich zu untersuchen. In Kapitel 6.3 wird ein erster Versuch zur Konfigurierung mit dem C-Präprozessor beschrieben. Dabei geht es vermehrt darum, erste Erfahrungen mit der Konfigurierung von Hardware zu sammeln und nicht um eine tatsächliche Herangehensweise zur Konfigurierung von komplexen Systemen. Im Anschluss werden in Kapitel 6.4 zwei Ansätze mit dem Variantenmanagment-Tool pure::variants vorgestellt.

6.1 Merkmale zur Konfigurierung des MPSoCs

Das in Kapitel 5 entworfene MPSoC ist für die Konfigurierung von folgenden Merkmalen ausgelegt:

- Soft Cores
  - Anzahl der Soft Cores
  - ID des Cores / Priorität beim Senden
- Kommunikation
  - Breite des Datenbusses

  - Priorität beim Senden
  - Größe des Puffers für ausgehende Nachrichten
  - IPC-Mechanismen
    * Message Queue
      · Bus- oder Ringstruktur
      · Größe der Message Queue
      · auf Daten warten
    * Shared Memory
      · Größe des Shared Memory
    * Event Flags
- I/O-Peripherie
  - UART-Schnittstelle
    * Zuordnung der Schnittstelle zu einem Core
    * Baudrate
  - LEDs
    * Zuordnung der LEDs zu einem Core
- FPGA-Typ, zur Erstellung des passenden Constraint-Files
- Bmm-File erzeugen

6.2 Fragmentierung der VHDL-Quelldateien

Jede der im vorherigen Kapitel beschriebenen Komponenten des MPSoCs wie z.B. IPC-Controller, Core oder Speicher, werden in einer eigenen VHDL-Datei beschrieben. Die Komponenten bzw. Design Entities setzen sich dabei aus zwei Teilen zusammen, der Entity-Deklaration und einer oder mehrerer Architekturen. Die Entity-Deklaration beschreibt die externe Schnittstelle und die Architektur das interne Verhalten der Design Entity.
Die Tatsache, dass für eine Komponente verschiedene Architekturen beschrieben werden können, lässt sich ideal für die Konfigurierung einsetzen. So kann für jede Instanz einer Komponente eine individuelle Beschreibung zu der Design Entity hinzugefügt werden. Die meisten Instanzen einer Komponente unterscheiden sich nur in sehr wenigen Punkten voneinander. Zum Beispiel ändern sich für die Core-Instanzen nur die jeweils verwendeten Speicher-Instanzen. Auf diese Weise ist es möglich eine Art Vorlage für die Architekturen zu nutzen und diese Vorlage mit wenigen Textersetzungen an die jeweilige

Core-Instanz anzupassen. Im Folgenden ein gekürztes Beispiel für dieses Vorgehen:

Vorlage für die Architektur einer Core-Instanz:

```
-- Beschreibung von ZPU-Core #COREID#
architecture behave_core#COREID# of zpu_core is
  ...
  -- gibt an, welche Architektur des Speichers für diesen Core genutzt wird
  for memory : ram use entity work.ram(ram_arch#COREID#);
  ...
end behave_core#COREID#;
```

Allein durch die Ersetzung des Schlüsselwortes *#COREID#* mit der richtigen Core-ID und das Hinzufügen zur Entity-Deklaration kann eine neue Instanz für einen Core beschrieben werden:

```
entity zpu_core is
  port(clk              : in std_logic;
       areset           : in std_logic;
         ...
       ipc_writeEnable  : out std_logic;
       ipc_dataReady    : in std_logic);
end zpu_core;


-- Beschreibung von ZPU-Core 1
architecture behave_core1 of zpu_core is
  ...
  -- gibt an, welche Architektur des Speichers für diesen Core genutzt wird
  for memory : ram use entity work.ram(ram_arch1);
  ...
end behave_core1;


-- Beschreibung von ZPU-Core 2
architecture behave_core2 of zpu_core is
  ...
  -- gibt an, welche Architektur des Speichers für diesen Core genutzt wird
  for memory : ram use entity work.ram(ram_arch2);
  ...
end behave_core2;
```

Genau wie bei den Core-Instanzen wird auch beim Speicher und anderen Komponenten, die mehrfach instantiiert werden müssen, vorgegangen.
Wie die passende Architektur ausgewählt wird, ist bereits an dem oben präsentierten

Beispiel zu erkennen. In diesem Fall wird für die Cores angegeben, welche Architektur zur Design Entity *ram* zu nutzen ist.

Der Vorteil bei der Arbeit mit solchen Vorlagen liegt, neben der einfachen Konfigurierung, in der anschließenden Wartbarkeit des Systems. Wenn eine Änderung am Core vorgenommen werden soll, muss dies nur in der Schablone für die Architektur des Core geschehen. Wenn das MPSoC anschließend neu generiert wird, sind alle Cores auf dem aktuellen Stand. Weiterhin ist es aber auch möglich, einen spezialisierten Core in dem MPSoC einzusetzen, indem für den spezialisierten Core eine eigene Design Entity erstellt wird.

6.3 Konfigurierung mit dem C-Präprozessor

Der C-Präprozessor durchläuft den Programmcode vor der eigentlichen Kompilierung und führt definierte Änderungen und Ersetzungen im Programmcode aus. Dabei ist der C-Präprozessor nicht auf die Sprache C beschränkt, sondern kann sämtliche Dateien im Textformat verarbeiten. So bietet der C-Präprozessor eine primitive Option kleine Systeme zu konfigurieren.
Der C-Präprozessor stellt im Grunde drei Methoden zur Verfügung, die zur Konfigurierung eingesetzt werden könnnen. Mit der *#include*-Anweisung können Textfragmente in Dateien eingefügt werden. Über die Anweisung *#define* können Konstanten definiert werden. Die im Quelltext auftretenden Symbole der Konstanten werden dabei vom Präprozessor bei der Verarbeitung durch den Wert der Konstante ersetzt. Die letzte Methode bietet die *#if*- bzw. *#ifdef*-Anweisung. Diese Anweisung erlaubt es, abhängig von einer Bedingung, Code-Abschnitte in den Quellcode einzufügen oder zu entfernen.

6.3.1 Beispiel

Hier wird am Beispiel der Core-Komponente erläutert, wie die Konfigurierung mit dem Präprozessor aussehen kann. Das Vorgehen ist wie in Kapitel 6.2 beschrieben. Der VHDL-Code für den Core wird einmal in die Entity-Deklaration *„core_ entity.src“* und einmal in eine Schablone für die Architektur *„core_ arch.src“* zerlegt. Nach welchen Kriterien die Core-Komponente zusammengesetzt wird, ist in der Konfigurations-Datei *„core.cfg“* festgelegt:

```
#include "core_entity.src"

#if CORES >= 1
#define ARCHITECTURE behave_core1
#define COREID 1
#define RAMARCH ram_arch1
#include "core_arch.src"
```

```
#undef ARCHITECTURE
#undef COREID
#undef RAMARCH
#endif

#if CORES >= 2
#define ARCHITECTURE behave_core2
#define COREID 2
#define RAMARCH ram_arch2
#include "core_arch.src"
#undef ARCHITECTURE
#undef COREID
#undef RAMARCH
#endif

...
```

Zuerst wird die Entity-Deklaration eingebunden. Anschließend wird anhand der Konstante *CORES* entschieden, wie viele Core-Architekturen hinzugefügt werden. Für jede Architektur werden dann drei Konstanten definiert, die zur Ersetzung der Symbole in der Architektur-Vorlage benutzt werden. An dieser Stelle fällt auch schon ein Problem des Präprozessors auf. Eigentlich wäre es ausreichend nur die Core-ID zu definieren, aber der Präprozessor ersetzt keine Symbole, die Teil eines Konstruktes sind, sondern nur allein stehende Symbole. Um den gleichen Konstanten für den nächsten Core einen anderen Wert zuzuweisen, werden die *#defines* mit *#undef* zurückgenommen. Die Konstante für *CORES* wird direkt beim Aufruf des Präprozessors übergeben. Auf diese Weise muss nicht für jede Konfigurierung die Konfigurations-Datei geändert werden. Der Aufruf des Präprozessors für drei Cores sieht folgendermaßen aus:

```
cpp -P -DCORES=3 core.cfg -o core.vhd
```

Das Ergebnis ist die VHDL-Datei *„core.vhd“*, die automatisch anhand der Anzahl der Cores erzeugt wird. Für die weiteren Komponenten des MPSoCs ist das hier beschriebene Vorgehen analog und wird nicht weiter vertieft.
Neben der Anzahl der Cores konnten mit diesem Verfahren noch die Breite des Datenbusses und die Größe der Message Queue konfiguriert werden. Der IPC-Mechanismus kann mit diesem Verfahren nicht konfiguriert werden und ist auf den Message Queue-Mechanismus festgelegt.
Eine abschließende Bewertung und Beschreibung der Schwachstellen zur Konfigurierung mit dem C-Präprozessor findet sich in Kapitel 7.1.1.

6.4 Konfigurierung mit pure::variants

In den folgenden beiden Unterkapiteln werden zwei Ansätze zur Konfigurierung mit pure::variants vorgestellt. Der erste Ansatz in Kapitel 6.4.1 lässt nur eine Konfigurierung von MPSoCs mit maximal fünf Soft Cores zu. Da diese Anzahl für die angestrebten Ziele zu wenig ist und jeder weitere Soft Core die Wartbarkeit des Family Models stark einschränkt, wurde nach einem weiteren Ansatz gesucht. Der zweite Ansatz in Kapitel 6.4.2 löst das Problem mit der geringen Anzahl der Soft Cores in einem MPSoC und zeigt eine Möglichkeit zur variablen Konfigurierung der Core-Anzahl.

6.4.1 Ansatz 1

Dieser Ansatz war der erste Versuch mit pure::variants Hardware in Form von VHDL-Quellcode zu konfigurieren. Im weiteren Verlauf dieses Kapitels werden das erstellte Feature- und Family Model für diesen Ansatz präsentiert. Zum Ende wird auf die aufgetretenen Probleme mit diesem Ansatz eingegangen.

6.4.1.1 Feature Model

Das Feature Model in Abbildung 6.1 teilt sich für diesen Ansatz in vier Abschnitte. Der erste Abschnitt stellt die Merkmale zur Konfigurierung der Cores zur Verfügung. Es können zwischen einem und fünf Cores in das MPSoC eingebunden werden. Außerdem können für die Cores Angaben zur Peripherie getroffen werden. Sowohl die UART-Schnittstelle als auch die LEDs können nur für einen Core konfiguriert werden. Aus diesem Grund wurden Konflikt-Restriktionen zu den UART- und LED-Merkmalen hinzugefügt, die eine mehrfache Konfigurierung der Peripherie verhindern. Über die Requires-Restriktion wird sichergestellt, dass im Falle einer Konfigurierung der UART-Schnittstelle für einen Core auch eine Baudrate gewählt wird. Der zweite Abschnitt gibt die Möglichkeit zur Konfiguration der Kommunikation im MPSoC. Zuerst werden global für alle IPC-Mechanismen Attribute definiert. Es kann die Breite für den Datenbus angegeben werden, die Anzahl der Bits, die zur Adressierung der Cores benötigt werden und die Puffer-Größe für ausgehende Nachrichten im IPC-Controller. Die Attributwerte sind nicht fest definiert und können im Variant Description Model den Anforderungen entsprechend parametrisiert werden. Außerdem ist in diesem Abschnitt die Auswahl der IPC-Mechanismen dargestellt. Es kann zwischen den drei IPC-Mechanismen Message Queue, Shared Memory und Event Flags gewählt werden. Für die Message Queue kann angegeben werden, ob ein Bus oder ein Ring zur Kommunikation verwendet werden soll. Des Weiteren lässt sich konfigurieren, wie viele Nachrichten in der Message Queue gespeichert werden können und ob bei leerer Message Queue auf die nächste Nachricht gewartet wird. Für den Shared Memory kann die Anzahl der Speicherzellen angegeben werden. Da sich zurzeit über die Größe des Shared Memory noch nicht die benötigte Adressbreite zur Adressierung berechnen lässt, muss die benötigte Adressbreite manuell angegeben werden. Die Anzahl der Event Flags lässt sich nicht separat konfigurieren und

wird über die angegebene Datenbus-Breite festgelegt. Für alle IPC-Mechanismen lässt sich zusätzlich die Option „Priorität beim Senden“ konfigurieren. Im dritten Abschnitt kann das verwendete FPGA konfiguriert werden. Diese Angabe ist für die Erstellung des Constraint-Files wichtig, das abhängig vom FPGA die Verbindung zwischen den Signalen des MPSoCs und den Pins des FPGAs angibt. Im letzten Abschnitt kann die Baudrate für die UART-Schnittstelle konfiguriert werden und es kann angegeben werden, ob ein Bmm-File für das MPSoC erzeugt werden soll. Um eine Baudrate zu konfigurieren, muss mindestens für einen Core die UART-Schnittstelle aktiviert werden.

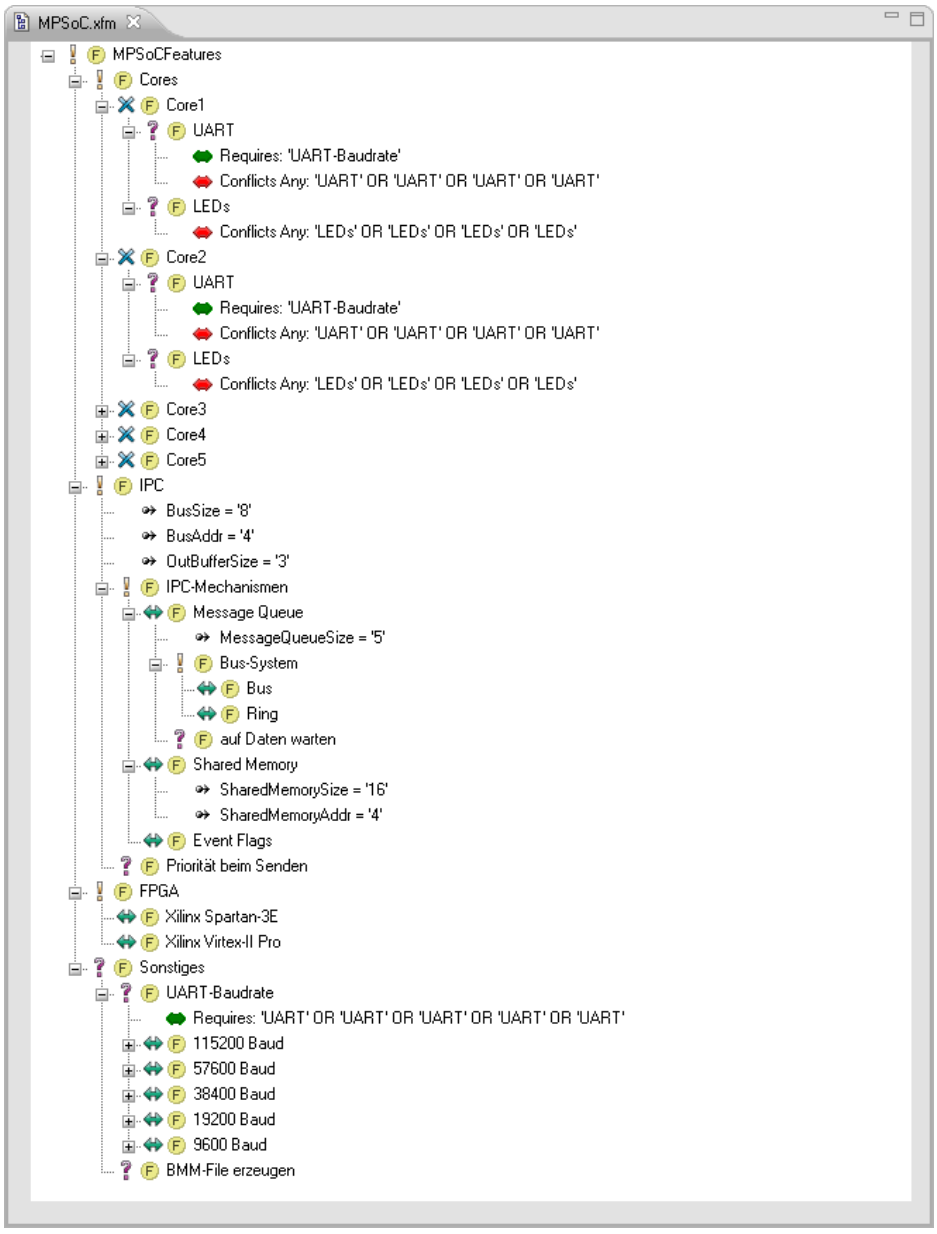

Abbildung 6.1: Feature Model

6.4.1.2 Family Model

Aufgrund der Komplexität des Family Models kann in Abbildung 6.2 nur ein kleiner Ausschnitt gezeigt werden. Das grundlegende Konzept lässt sich an diesem Ausschnitt aber ohne weiteres vermitteln.

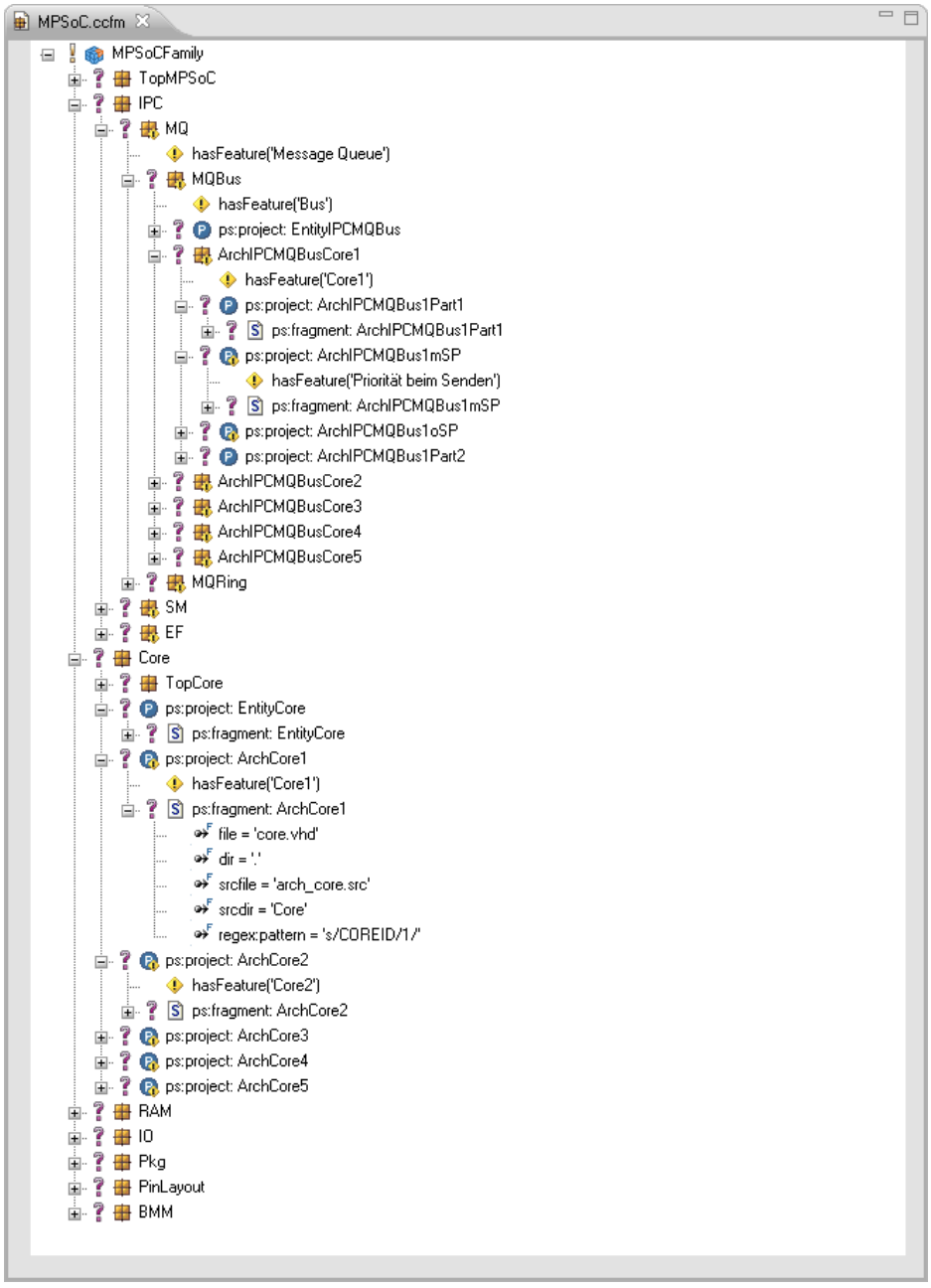

Abbildung 6.2: Family Model

Die einzelnen zu erzeugenden Teile des MPSoCs sind, wie bereits in Kapitel 6.2 erläu-

tert, in eigenen VHDL-Dateien beschrieben. Für jede zu erstellende VHDL-Datei bzw. logisch zusammenhängende VHDL-Dateien, wurde in dem Family Model eine Komponente eingefügt. Diese Komponenten enthalten jeweils die vollständige Beschreibung für alle möglichen Lösungen des Lösungsraums zu einem Teil des MPSoCs, wie z.B. dem Core oder dem Speicher. Wie der vollständige Lösungsraum für die Teile des MPSoCs beschrieben ist, wird nun exemplarisch an den Teilen Core und IPC-Controller erläutert. Der Lösungsraum für den Core-Teil des MPSoCs besteht aus zwei VHDL-Quelltext Fragmenten und zwar der Entity-Deklaration und der Architektur-Vorlage für die Cores. Die Entity-Deklaration wird über das Fragment *EntityCore* zuerst in die VHDL-Datei *„core.vhd“* kopiert. Anschließend folgen, abhängig von der gewählten Konfiguration im Variant Description Model, die Core-Architekturen. Damit nur für die ausgewählten Cores die Architekturen in die angegebene Datei *„core.vhd“* kopiert werden, sind diese mit einer Restriktion versehen, die überprüft, ob das entsprechende Merkmal ausgewählt ist. Um aus den allgemeinen Architektur-Vorlagen auf den jeweiligen Core zugeschnittene Architekturen zu erzeugen, wird in jeder Architektur eine Ersetzung durchgeführt. Die Ersetzung kann direkt als Attribut bei einem pure::variants Element vom Typ Fragment angegeben werden. Dazu wird ein Attribut mit dem Namen *regex:pattern* hinzugefügt und diesem ein Wert in folgender Form zugewiesen *'s/COREID/1'*. In diesem Fall stellt *COREID* den Ausdruck dar, der im Text durch eine Eins ersetzt werden soll.
Der IPC-Controller-Teil des MPSoCs baut sich im Family Model ähnlich, aber aufgrund der zahlreichen Optionen zur Konfigurierung wesentlich komplexer auf. Zuerst werden unter der Komponente IPC weitere Komponenten eingefügt, die den Lösungsraum für den IPC-Controller nach den zur Verfügung stehenden IPC-Mechanismen disjunkt aufteilen. Bei der Message Queue unterteilt sich der Lösungsraum nochmals in die Bereiche Bus und Ring. Auf dieser Ebene ist das Vorgehen wieder größtenteils analog zum Core-Teil des MPSoCs. Allerdings wurde die Architektur in drei Fragmente unterteilt, da abhängig von der Option „Priorität beim Senden“ verschiedene VHDL-Fragmente im Mittelteil der Architektur eingefügt werden können.

6.4.1.3 Proleme bei diesem Ansatz

Mehr als fünf Soft Cores sind mit diesem Ansatz kaum sinnvoll zu konfigurieren. Insbesondere zur Modifikation des Family Model muss jeder Arbeitsschritt für jeden Soft Core wiederholt durchgeführt werden. Bei einem MPSoC mit 16 Soft Cores bedeutet dies, dass im Falle einer Änderung im Family Model diese Änderungen 16 mal durchgeführt werden müssen. Noch schlimmer sieht es bei grundlegenden Umgestaltungen der IPC-Mechanismen aus. Das könnte dazu führen, dass für jede IPC-Mechanismus/Core-Kombination etwas im Family Model angepasst werden müsste. Auf Dauer ist solch eine Lösung deshalb nicht tragbar und in der Praxis kaum einzusetzen. Einen Ausweg aus dieser Problematik bietet der im Folgenden vorgestellte Ansatz.

6.4.2 Ansatz 2

Um das Problem mit der begrenzten Core-Anzahl zu lösen, wird für diesen Ansatz das Konzept der hierarchischen Variantenstruktur genutzt. Dazu wird, zusätzlich zum bestehenden Konfigurationsraum des MPSoCs, ein neuer Konfigurationsraum mit dem Namen *Core* erzeugt. Für diesen Konfigurationsraum werden ein eigenes Feature- und Family Model erstellt, mit dessen Hilfe die Konfiguration für einen einzelnen Core beschrieben wird. Dieser Konfigurationsraum kann dann mehrfach im Konfigurationsraum des MPSoCs instantiiert werden. Wie die Instantiierung der einzelnen Core-Varianten funktioniert und welchen Vorteil dieses Verfahren bietet, wird in den folgenden Unterkapiteln deutlich.

6.4.2.1 Feature Model

Das Feature Model in Abbildung 6.3 ähnelt zu großen Teilen dem Modell aus dem ersten Ansatz. Im Folgenden werden hauptsächlich die durchgeführten Änderungen und die Gründe dafür erläutert.
Im Vergleich zum alten Modell fällt zuallererst das Fehlen der einzelnen Cores auf. An deren Stelle ist ein Link zu dem bereits angesprochenen Konfigurationsraum *Core* getreten. Dabei handelt es sich um eine sogenannte Variant Instance. Die Variant Instance ermöglicht es, im Variant Description Model beliebig viele Instanzen dieses Konfigurationsraums hinzuzufügen. Für jeden instantiierten Core wird im Variant Description Model unter Cores ein verlinktes Variant Description Model eingefügt. In diesem verlinkten Variant Description Model lassen sich, wie gewohnt, die Merkmale der Cores auswählen.
An dieser Stelle ist direkt der Vorteil dieses Ansatzes zu erkennen. Es muss nun nicht mehr der Konfigurationsaufwand für n-Cores n-mal betrieben werden, sondern es wird einmal allgemein beschrieben, wie sich die VHDL-Fragmente für einen Core zusammensetzen und welche Ersetzungen durchgeführt werden müssen. Dieses Konfigurationswissen kann dann unkompliziert n-mal instantiiert werden.
Da es nicht möglich ist, über verschiedene Feature Models Restriktionen festzulegen, aber sichergestellt werden muss, dass nur ein Core mit der UART-Schnittstelle bzw. den LEDs verbunden ist, werden diese Merkmale nicht mehr, wie beim ersten Ansatz, lokal bei jedem Core konfiguriert, sondern einmal anhand der Core-ID angegeben.
In dem Feature Model befindet sich mit *Close* noch ein weiterer Link in einen anderen Konfigurationsraum. Jedoch handelt es sich dabei nicht um eine Variant Instance, sondern um eine Variant Reference. Der Unterschied besteht darin, dass für die Variant Reference ein konkretes Variant Description Model eines anderen Konfigurationsraums in das Variant Description Model eingefügt wird. Dieser Konfigurationsraum lässt sich weder mehrfach einbinden, noch lässt sich die Merkmalauswahl des verlinkten Variant Description Models verändern. Der Nutzen für den Konfigurationsraum *Close* wird im weiteren Verlauf dieses Kapitels deutlich.
Die Feature Models der Konfigurationsräume *Core* und *Close* sind nicht extra abgebil-

det, da das Feature Model des Konfigurationsraums *Core* nur ein Attribut zur Angabe der jeweiligen Core-ID beinhaltet und das Feature Model des Konfigurationraums *Close* überhaupt keine Elemente enthält.

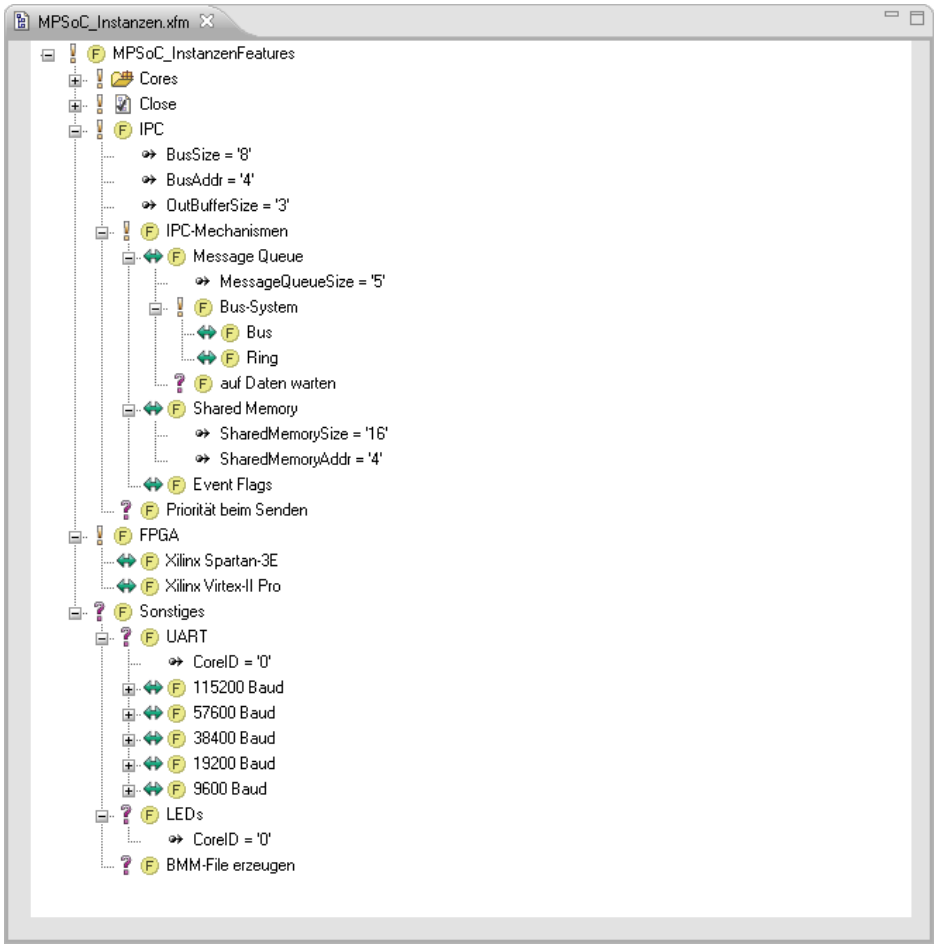

Abbildung 6.3: Feature Model aus dem Konfigurationsraum MPSoC

6.4.2.2 Family Model

Während im Family Model des letzten Ansatzes die VHDL-Dateien aufgebaut wurden, indem zuerst die Entity-Deklarationen und dann die Architekturen für die konfigurierten Cores kopiert wurden, sind bei diesem Ansatz die Aufgaben auf die Family Models der Konfigurationsräume *MPSoC* und *Core* aufgeteilt.
In Abbildung 6.4 ist das Family Model des Konfigurationsraums *MPSoC* dargestellt. Das Family Model hat sich von dem letzten zu diesem Ansatz stark vereinfacht. Sämtliche Fragmente, die die Architekturen der VHDL-Dateien beschreiben, sind aus diesem Modell verschwunden. Übrig bleiben nur noch die Entity-Deklarationen für die Komponenten und allgemeine VHDL-Dateien, wie Pakete, die unabhängig von der Anzahl der

Cores erzeugt werden können.

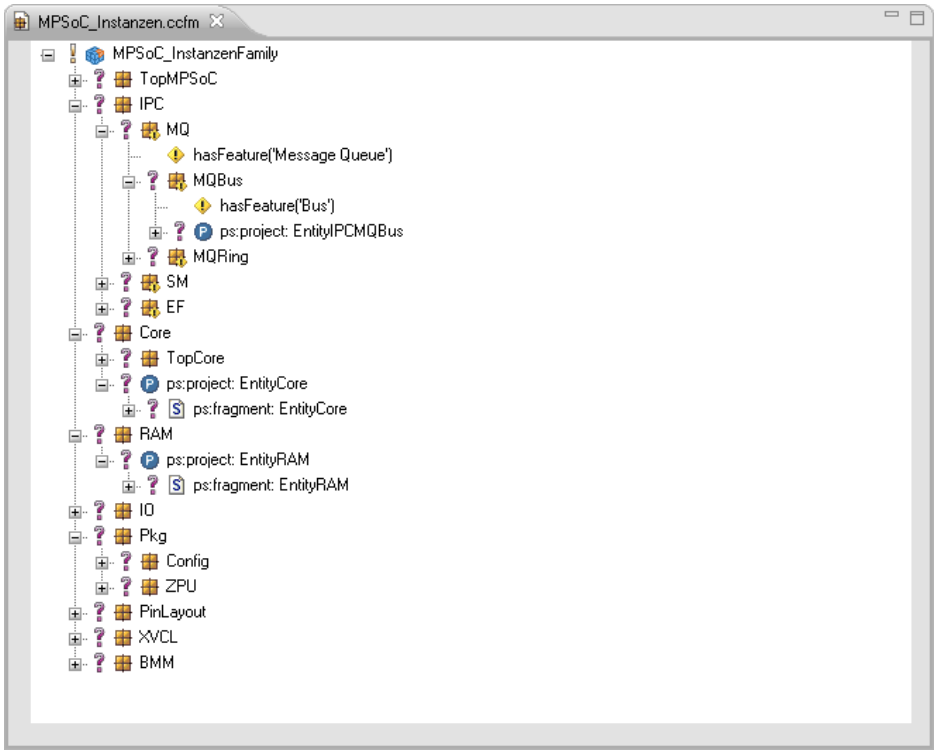

Abbildung 6.4: Family Model aus dem Konfigurationsraum MPSoC

Das Family Model des Konfigurationsraums *Core* wird in Abbildung 6.5 gezeigt. In diesem Modell wurden nun die Fragmente, die direkt in Abhängigkeit zu einem Core stehen, wie z.B. die Architekturen, beschrieben. Da der Konfigurationsraum *Core* beliebig oft instantiiert werden kann, hält sich der Konfigurationsaufwand auch für dieses Family Model in Grenzen. So muss, im Gegensatz zum ersten Ansatz, die gesamte Beschreibung nur für einen Core erfolgen.
Mit dieser neuen Flexibilität treten aber auch zwei neue Probleme auf. Wie kommen wir im Family Model des Konfigurationsraum *Core* an das Attribut *CoreID* aus dem Feature Model des selben Konfigurationsraums? Da in dem Family Model nicht mehr, wie im ersten Ansatz, für jeden Core alle Code-Fragmente fest beschrieben sind, muss zur Ersetzung der Core-ID im VHDL-Quellcode die konfigurierte Core-ID aus dem Feature Model ausgelesen werden. Zu diesem Zweck gibt es den Ausdruck *getAttribute*. Dieser Ausdruck kann, wie in Abbildung 6.5 zu sehen, direkt im Ausdruck zur Textersetzung verwendet werden. Dem Ausdruck *getAttribute* werden drei Parameter übergeben. Der erste Parameter ist das Merkmal, zu dem das Attribut gehört, in diesem Fall gehört das Attribut zu dem Merkmal *CoreFeatures*. Der zweite Parameter ist der Name des Attributs und zuletzt wird der im Quelltext zu ersetzende Ausdruck angegeben.

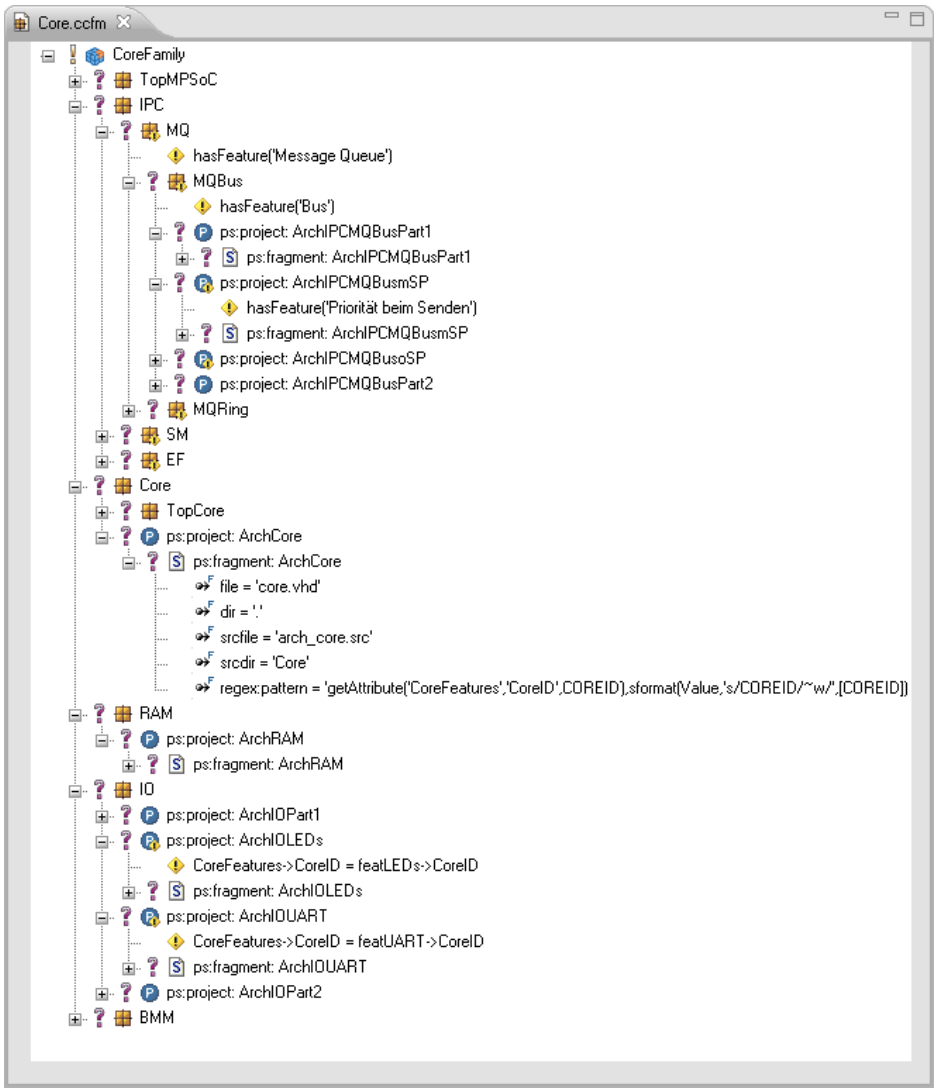

Abbildung 6.5: Family Model aus dem Konfigurationsraum Core

Das zweite Problem ist: Wie wird im Family Model des Konfigurationsraums *Core* erkannt, dass für den hier instantiierten Core im Feature Model des Konfigurationsraums *MPSoC* eine Peripherie, wie z.B. die UART-Schnittstelle, konfiguriert wurde? Damit das Code-Fragment, z.B. für die UART-Schnittstelle, nur für den passenden Core konfiguriert wird, muss eine Restriktion eingesetzt werden, die diese Rahmenbedingungen kontrollieren kann. Solch eine Restriktion lässt sich mittels der pure::variants eigenen Ausdruckssprache pvSCL realisieren. Mit pvSCL lassen sich zwei Attribute miteinander vergleichen. Dabei können die Attribute auch, wie in diesem Fall, in verschiedenen Feature Models angegeben sein. Ein Beispiel für einen derartigen Ausdruck ist in Abbildung 6.5 unterhalb der Komponente *IO* zu sehen. Der Ausdruck ist recht simpel aufgebaut. Zuerst wird das Merkmal des Attributs angegeben und über den Ausdruck „->“ mit dem Namen des Attributs verbunden. Das zweite Attribut wird auf die selbe Weise ausge-

wählt und dann über einen beliebigen Vergleichsoperator mit dem ersten Attribut in Relation gesetzt.

Nach diesem Vorgehen lassen sich fast alle VHDL-Dateien erstellen. Probleme gibt es aber bei der Datei *„multicore_top_fpga.vhd“* und dem Bmm-File. Warum sich diese Dateien nicht auf diese Weise konfigurieren lassen und welche Lösungswege eingeschlagen wurden, wird in den beiden folgenden Unterabschnitten erläutert.

Erzeugung der MPSoC Top Komponente

Die Erzeugung der MPSoC Top Design-Entity ist mit den bisher vorgestellten Mitteln nicht zu erreichen. Dies liegt an dem abweichenden Aufbau dieser Komponente. Die anderen Komponenten, wie Core, Speicher oder IPC-Controller, haben immer den gleichen Aufbau. Zuerst wird aus dem MPSoC-Konfigurationsraum die Entity-Deklaration in eine Datei geschrieben und anschließend werden die einzelnen Architekturen von den jeweiligen Instanzen des Core-Konfigurationsraums in die Dateien eingehängt. Dies funktioniert für die MPSoC Top Komponente nicht, denn diese Komponente hat nur eine Architektur in der für die konfigurierten Cores mehrfach abwechselnd geschrieben werden muss. Dieses lässt sich mit dem bisherigen Ansatz und allein mit pure::variants nicht lösen. Um zu verstehen, wieso sich die MPSoC Top Komponente nicht erzeugen lässt, werfen wir zuerst einen kurzen Blick auf das Result Model in Abbildung 6.6.

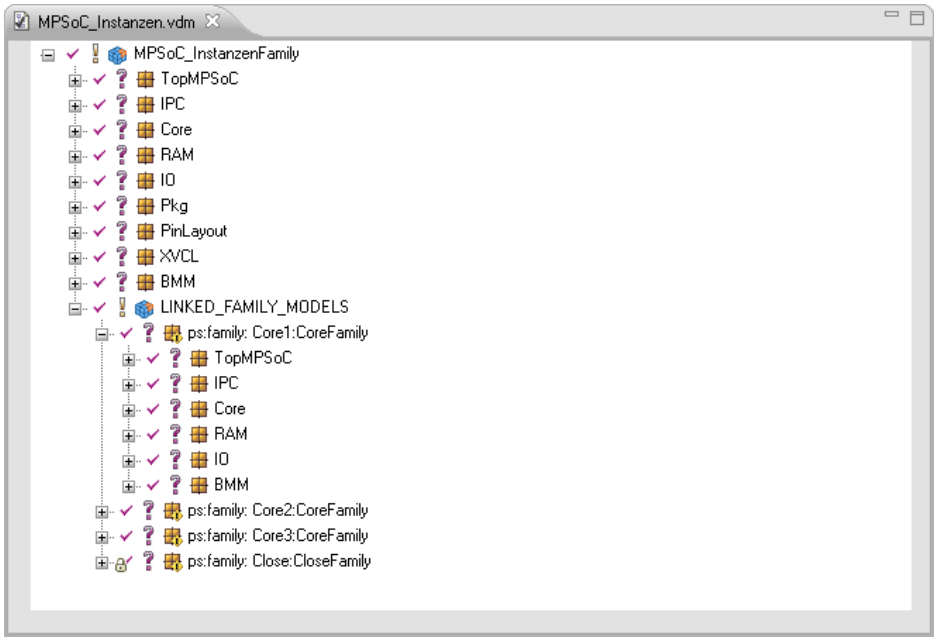

Abbildung 6.6: Result Model

Die Transformation arbeitet die Komponenten von oben nach unten ab. Zuerst wird das eigentliche Result Model des MPSoC-Konfigurationsraums abgearbeitet und anschließend die Result Models der hierarchischen Varianten, die sich unter dem Element *LINKED_FAMILY_MODELS* befinden. Das bedeutet, sobald im Verlauf der Transformation die Inhalte für Core 1 geschrieben sind und Core 2 an der Reihe ist, können für Core 1 keine weiteren Inhalte mehr geschrieben werden. Dies macht es unmöglich die MPSoC Top Komponente zu erzeugen.
Zur Lösung dieses Problems wird das in Kapitel 3.2 vorgestellte XVCL benutzt. Die Schwächen von XVCL, wie die fehlende Darstellung des Problemraums oder die schlechtere Handhabbarkeit, werden durch die Integration von XVCL direkt in pure::variants beseitigt. Dazu werden im Zuge der Default-Transformation zum einen die für XVCL benötigten x-frames in den Ausgabeordner kopiert und zum anderen wird anhand der ausgewählten Merkmale in pure::variants automatisch ein an die Lösung angepasster Specification x-frame erzeugt. Damit auch die Erzeugung der MPSoC Top Komponente durch XVCL in pure::variants gestartet werden kann, wurde neben der Default-Transformation eine weitere Transformation erstellt, die XVCL direkt auf dem generierten Specification x-frame ausführt. Um also das konfigurierte MPSoC vollständig zu erzeugen, wird zuerst die Default-Transformation ausgeführt, in der die nötigen Dateien für XVCL und alle VHDL-Dateien, außer der *„multicore_top_fpga.vhd“*, erzeugt werden. Anschließend wird die XVCL-Transformation gestartet, die anhand des vorher generierten Specification x-frames dann die Datei *„multicore_top_fpga.vhd“* erstellt.

Erzeugung des Bmm-Files und des Specification x-frames

Ein weiteres Problem ist die Erzeugung des Bmm-Files und des Specification x-frames. Das Schema dieser zu erstellenden Dateien ist folgendes:

- Header
- Core 1
- Core 2
- ...
- Footer

Zuerst wird eine Art Header mittels des MPSoC-Konfigurationsraums erstellt und die instantiierten Core-Konfigurationsräume tragen ihre Inhalte nacheinander in die zu erzeugenden Dateien ein. Bis hierher gibt es noch keine Probleme, aber diese Dateien benötigen eine Art Abschluss. Es ist nicht möglich diesen Abschluss durch den letzten Core schreiben zu lassen, da es keine Verbindung zwischen den instantiierten Core-Konfigurationsräumen gibt. Aus diesem Grunde wurde der Konfigurationsraum Close eingebunden. Wie bereits im Result Model in Abbildung 6.6 zu sehen war, wird das

Family Model des Konfigurationsraums Close zuunterst in das Result Model eingehängt und deshalb auch zuletzt abgearbeitet. So können die letzten Einträge in das Specification x-frame und das Bmm-File geschrieben werden.

7 Evaluation

Nachdem das MPSoC implementiert wurde und nach gewissen Merkmalen konfiguriert werden kann, beschäftigt sich dieses Kapitel mit den Ergebnissen dieser Arbeit. Zuerst werden in Kapitel 7.1 die eingesetzten Techniken zur Konfigurierung in Bezug auf MPSoCs bewertet. In Kapitel 7.2 wird der Ressourcenverbrauch der MPSoC Konfigurationen auf den FPGAs ausgewertet. Abschließend werden in Kapitel 7.3 die implementierten MPSoC Konfigurationen auf verschiedene Faktoren, wie den Datendurchsatz, die Latenz von Nachrichten oder die Geschwindigkeit der ZPU analysiert.

7.1 Ergebnisse zur Konfigurierung

In diesem Kapitel werden die in dieser Arbeit verwendeten Tools zur Konfigurierung des MPSoCs bewertet. Zuvor wird jedoch kurz auf das Konzept der in VHDL zur Verfügung stehenden Generics eingegangen und erläutert, warum diese nicht einzig zur Konfigurierung von MPSoCs geeignet sind. Generics sind eine Art von Konstanten, die allerdings nicht global festgelegt werden, sondern für jede Instanz einer Entity übergeben werden können. So können beispielsweise für die Core-Entities die IDs übergeben werden. Problematisch wird dies für die Erzeugung der Speicher-Entities, da sich bei diesen nicht nur die IDs ändern, sondern der gesamte Speicherinhalt. Deshalb wird für jede Instanz einer Design Entity eine eigene Architektur erzeugt, was den Einsatz der Generics so gut wie überflüssig macht.

7.1.1 C-Präprozessor

Der C-Präprozessor ist offensichtlich nicht dazu geeignet, komplexe MPSoCs zu konfigurieren. Dazu sind die folgenden Schwachstellen zu schwerwiegend:

- keine Textersetzung in Teilausdrücken
- keine Schleifen
- das Wissen zur Konfigurierung befindet sich zwischen dem Quellcode
- der Quellcode wird bei größeren Projekten schnell unübersichtlich

Aber mit dem C-Präprozessor wurde das grundlegende Konzept erarbeitet wie VHDL-Dateien automatisiert generiert werden können. Dieses Konzept wurde dann auf pure::variants angewendet und erweitert.

7.1.2 pure::variants und XVCL

Wie bereits in Kapitel 6.4.2 vorgestellt, reicht für die Konfigurierung von komplexen MPSoCs mit einer Vielzahl von Soft Cores pure::variants alleine nicht aus. Deshalb werden in diesem Kapitel pure::variants und XVCL gemeinsam auf ihre Eignung zur Konfigurierung von MPSoCs anhand von Merkmalen untersucht.
pure::variants bietet eine schnelle und bequeme Technik zur Erzeugung von MPSoC Konfigurationen. Dieses ist vor allem den Merkmalmodellen zuzusprechen, da der vollständige Problemraum abgebildet werden kann und so sehr schnell eine Merkmalauswahl für das vorliegende Problem getroffen werden kann. Des Weiteren kann es infolge der Abhängigkeiten zwischen den Merkmalen kaum zu Fehlkonfigurationen von nicht implementierbaren MPSoCs kommen. Auch der Lösungsraum in Form des Family Models lässt sich, aufgrund der Einteilung in Komponenten und Teile, strukturiert aufbauen und bei neu eingefügten Merkmalen schnell erweitern. Die Hauptprobleme von pure::variants bei der Konfigurierung von MPSoCs liegen darin, dass es kein Schleifen-Konstrukt gibt und die Reihenfolge in der die Komponenten und Textfragmente während der Transformation erzeugt werden, statisch im Result Model festgelegt sind. Schleifen könnten, wie es bisher bei der Textersetzung mittels *regex:pattern* in pure::variants möglich ist, im Family Model über Attribute zu den Textfragmenten hinzugefügt werden. So könnte ein Textfragment abhängig von einem Merkmal-Attribut beliebig oft eingefügt werden. Die bisher festgelegte Reihenfolge im Result Model, in der die Komponenten und Textfragmente während der Transformation erzeugt werden, könnte über Prioritäten verändert werden. Denkbar wäre die direkte Priorisierung der Textfragmente. So könnten Textfragmente aus verlinkten Konfigurationsräumen, die bisher erst zum Ende der Transformation erzeugt werden, vorgezogen werden.
Um diese Probleme mit pure::variants zu umgehen, wurde in dieser Arbeit XVCL in die Transformation mit eingebunden. XVCL bietet, wie bereits in Kapitel 6.4.2 dargestellt, die in pure::variants fehlenden Konzepte unter Einsatz der *<adapt>*- und *<while>*-Befehle. Die Schwäche von XVCL, dass es keine Beschreibung des Problemraums gibt, wird durch pure::variants ausgeglichen, indem der Specification x-frame für XVCL direkt aus den gewählten Merkmalen des Variant Description Models erzeugt wird.
Insgesamt bieten pure::variants und XVCL in Kombination einen praktikablen und schnellen Ansatz, um MPSoCs anhand von Merkmalen zu beschreiben und anschließend zu generieren. Neben den hier vorgestellten Konfigurationen können mit diesem Ansatz auch heterogene Systeme und MPSoCs mit verschiedenen Kommunikationsstrukturen erzeugt werden. Die Aufteilung der Cores zu den Kommunikationsstrukturen muss dazu aber disjunkt sein.
Ein wie in Abbildung 7.2 dargestelltes MPSoC kann mit dem Ansatz, wie er in dieser Arbeit vorgestellt wurde, nicht erzeugt werden. Zum einen muss dazu ein Weg gefunden werden, diese Strukturen in einem Merkmalmodell zu beschreiben und zum anderen muss aufgrund der erwähnten Schwächen von pure::variants und der komplexeren Struktur solch eines MPSoCs mehr VHDL-Quellcode durch XVCL erzeugt werden. Problematisch ist die Modellierung von Cores mit Verbindungen zu zwei oder mehr Kommunikationsstrukturen, wie z.B. in Abbildung 7.2 der zweite Core.

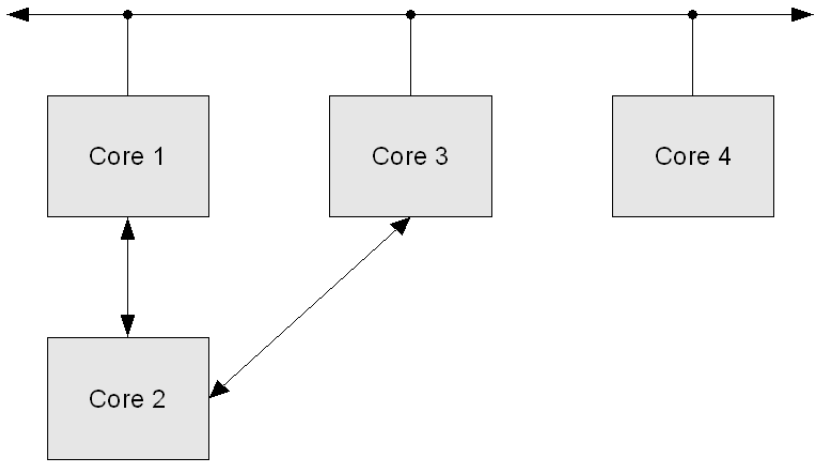

Abbildung 7.1: MPSoC mit verschiedenen Kommunikationsstrukturen

7.2 Ressourcenverbrauch der MPSoC Konfigurationen

Ein Beweggrund die ZPU für diese Arbeit zu nutzen, ist ihre geringe Größe und damit der geringe Ressourcenverbrauch. Auf diese Weise kann für jeden Prozess einer Anwendung ein eigener Core zur Verfügung gestellt werden. Im Folgenden wird der Ressourcenverbrauch für verschiedene Konfigurationen des MPSoCs analysiert.
In Abbildung 7.2 ist der relative Ressourcenverbrauch unter den Komponenten des MPSoCs dargestellt. In diesem Fall handelt es sich um eine Konfiguration mit Message Queue Mechanismus und einem 8 Bit breiten Datenbus. Die Komponenten werden für die wichtigsten Parameter eines FPGAs, wie den Slices, Flip-Flops, LUTs oder BRAMs gegenübergestellt. Auffällig ist, dass sowohl Core als auch IPC-Controller einen ähnlichen Ressourcenverbrauch haben. Dieses könnte damit zusammenhängen, dass der IPC-Controller bisher nicht auf einen optimalen Ressourcenverbrauch, sondern nur auf die Funktionalität ausgerichtet ist. Für diesen Test wurde das Spartan-3E Entwicklungsboard genutzt. Alle weiteren Tests werden mit dem Virtex-II Pro Entwicklungsboard durchgeführt.

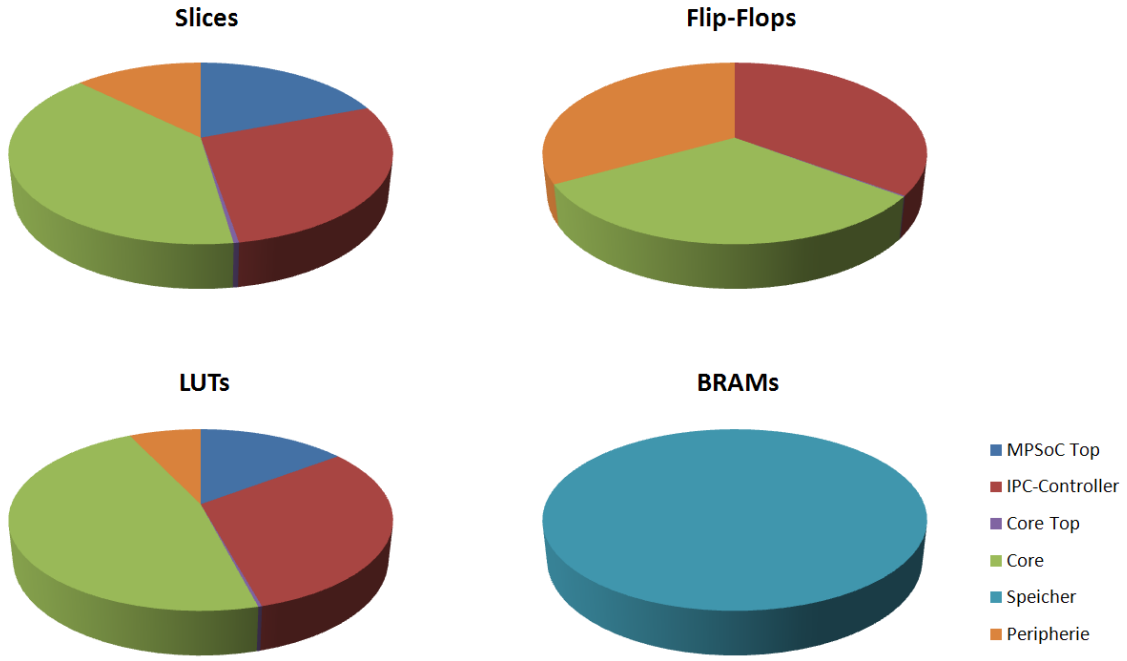

Abbildung 7.2: Ressourcenverbrauch der Komponenten eines MPSoCs

In Tabelle 7.1 wird der Ressourcenverbrauch abhängig von der Anzahl der Cores dargestellt. Wiederum handelt es sich hierbei um eine Message Queue mit 8 Bit breitem Datenbus. Mehr als die 16 implementierten Cores sind mit dem derzeitigen Ressourcenverbrauch und den zur Verfügung stehenden FPGAs nicht zu realisieren, da 98 Prozent der Slices verbraucht werden. Auch die Block-RAM-Zellen sind spätestens mit dem 17. Core bei 16 KB Speicher pro Core ausgelastet. Die Taktfrequenz, mit der die konfigurierbare Logik des FPGAs betrieben werden kann, ist abhängig von der Leitungslänge. Interessant ist in diesem Zusammenhang, dass die Anzahl der Cores offenbar keinen Einfluss auf die maximale Frequenz hat, mit der die konfigurierbare Logik des FPGAs betrieben werden kann.

Core-Anzahl	Slices	LUTs	Flip-Flops	BRAMs	max. Frequenz
4	6.009	3.341	2.499	32	69,965 MHz
10	8.312	15.017	6.153	80	69,965 MHz
16	13.466	24.301	10.023	128	69,965 MHz

Tabelle 7.1: Ressourcen in Abhängigkeit von der Core-Anzahl

In Tabelle 7.2 wird für die Datenbusbreiten 8, 16 und 32 Bit und allen Kommunikationstechniken der Ressourcenverbrauch auf dem Virtex-II Pro dargestellt. Für die Message

Queue wurde eine Größe von 5 Nachrichten konfiguriert und der Shared Memory hat in dieser Konfiguration 64 Speicherzellen. Die wenigsten Ressourcen benötigt der einfach aufgebaute Event Flag Mechanismus. Der Ring mit der Message Queue benötigt im Schnitt die meisten Ressourcen des FPGAs, obwohl der IPC-Controller des Rings keine Kollisionskontrolle benötigt und dadurch einfacher aufgebaut ist. Die wesentlich höhere Taktrate von ca. 15 bis 20 MHz für die Ring-Architektur gegenüber dem Bus, könnte mit einer kürzeren Leitung zwischen den Cores erklärt werden. Beim Bus müssen alle Leitungen der zehn Cores zusammengeführt werden, was eine höhere Leitungslänge zur Folge hätte als bei der Ring-Architektur, bei der immer nur zwei Cores über eine Leitung verbunden werden. Die höchste Auslastung einer Ressource des FPGAs wird durch den Ring mit 92 Prozent verwendeten Slices verursacht.

Kommunikation	Datenbus	Slices	LUTs	Flip-Flops	max. Frequenz
Message Queue, Bus	8 Bit	8.312	15.017	6.153	69,965 MHz
	16 Bit	9.156	14.280	7.113	69,965 MHz
	32 Bit	10.882	19.205	9.033	69,965 MHz
Message Queue, Ring	8 Bit	8.799	16.203	5.813	88,032 MHz
	16 Bit	9,883	18.114	6.693	90,785 MHz
	32 Bit	12.643	22.959	8.453	83,140 MHz
Shared Memory, Bus	8 Bit	8.451	15.217	6.078	66,738 MHz
	16 Bit	9.080	16.150	6.814	66,732 MHz
	32 Bit	10.623	18.469	8.286	66,568 MHz
Event Flags, Bus	8 Bit	7.513	13.755	5.143	69,965 MHz
	16 Bit	7.775	14.194	5.383	69,946 MHz
	32 Bit	8.349	15.125	5.863	69,955 MHz

Tabelle 7.2: Ressourcen in Abhängigkeit von der Busbreite (10 Cores)

7.3 Auswertung der Kommunikation

Im Folgenden werden der Versuchsaufbau und einige Ergebnisse von der Auswertung der ZPU sowie der Kommunikations-Strukturen vorgestellt.

7.3.1 Versuchsaufbau

Alle hier gemessenen Zeiten werden nicht in der Simulation, sondern direkt auf einem FPGA gemessen. Dazu wird ein Core verwendet, um die Zeiten zu messen und diese anschließend über die UART-Schnittstelle an ein Terminal auszugeben. Zu Beginn der

Messung wird der Timer des Cores zurückgesetzt und nach Abschluss des Versuchs wird der aktuelle Wert des Timers ausgelesen und über die UART-Schnittstelle ausgegeben.

7.3.2 Datendurchsatz der ZPU

Bei diesem Versuch wird getestet, welche maximale Last eine ZPU auf dem Bus verursachen kann. Zu diesem Zweck wird eine Konfiguration aus zwei ZPUs, dem Message Queue-Mechanismus mit einem Bus und eine Datenbreite von 32 Bit gewählt. Auf der ersten ZPU wird die Send-Funktion 1000-mal in einer Schleife aufgerufen und die benötigten Taktzyklen gemessen. Für diesen Vorgang benötigt die ZPU 503.358 Taktzyklen bei einer Frequenz von 50 MHz. Aus diesen Werten kann ein Durchsatz von ca. 388 KB pro Sekunde errechnet werden. Im Vergleich zum theoretisch maximalen Durchsatz von ca. 47 MB pro Sekunde kann eine ZPU dementsprechend nur 0,8 Prozent Last auf dem Bus erzeugen. Die geringe Geschwindigkeit der ZPU im Verhältnis zur Datenrate des Busses erschwert eine exakte Auswertung der einzelnen Kommunikations-Strukturen und der IPC-Mechanismen untereinander.

7.3.3 Unterschiede zwischen Bus und Ring

Ein Vergleich der beiden Kommunikations-Strukturen für den Message Queue-Mechanismus gestaltet sich Aufgrund der langsamen ZPU schwierig. Die Vorteile des Ringes liegen grundsätzlich in der höheren Datenrate und dem Wegfall von Kollisionen. Die höhere Datenrate entsteht in diesem Fall durch eine mögliche parallele Übertragung der Daten. Es könnte beispielsweise Core 1 mit Core 2 sowie Core 3 mit Core 4 gleichzeitig in einem Ring kommunizieren, falls die Cores entsprechend ihrer numerischen Reihenfolge angeordnet sind. Der einzige Nachteil im Vergleich zum Bus ist die längere Latenz der Nachrichten, falls diese über viele Stationen weitergeleitet werden müssen. Für den Bus ist es von Vorteil, dass die Nachrichten direkt beim Empfänger ankommen. Dieser Vorteil kann aber bei einer hohen Last des Busses ins Negative umschlagen, wenn sich die Übertragungszeiten aufgrund von Kollisionen erhöhen.
Um die Unterschiede zwischen Bus und Ring herauszustellen, wird im ersten Versuch jeweils eine MPSoC Konfiguration mit 10 ZPUs verwendet. Core 1 und Core 2 tauschen dabei immer abwechselnd eine Nachricht aus, bis insgesamt 200 Nachrichten versendet wurden. Für den Ring müssen die Nachrichten von Core 2 über acht Stationen weitergeleitet werden. Wie erwartet, dauert die Kommunikation über den Ring länger. Für die Kommunikation auf dem Bus werden ca. 1,66 ms und auf dem Ring werden ca. 1,70 ms benötigt. Hier wird wiederum deutlich, welchen Einfluss die geringe Geschwindigkeit der ZPU auf die Kommunikation hat. Denn obwohl für den Ring jede zweite Nachricht über acht Stationen weitergeleitet werden muss, ist der höhere Zeitaufwand mit ca. 40 μs relativ gering im Verhältnis zur gesamten Übertragungszeit.
Der zweite Versuch sollte die Vorteile des Ringes gegenüber dem Bus verdeutlichen. Dazu wurde für Bus und Ring jeweils ein MPSoC mit 9 Cores generiert. Die Cores 1, 3,

5 und 7 übernehmen die Rolle der Sender und die Cores 2, 4, 6 und 8 übernehmen die Empfänger-Rolle. Damit alle Sender zum gleichen Zeitpunkt ihre Übertragung starten und so Kollisionen erzwungen werden, ist das auszuführende Programm für alle Sender identisch, bis auf die Adresse des Empfängers. Die Sender übertragen ihre Nachrichten jeweils gleichzeitig zum Empfänger mit der nächst höheren ID. Sobald die Empfänger alle Nachrichten vom jeweiligen Sender erhalten haben, übertragen die vier Empfänger eine Nachricht an den neunten Core, der daraufhin den Timer ausliest und die Messung anschließend auf dem Terminal ausgibt. Bei diesem Versuch kann zwischen den beiden Kommunikations-Strukturen kein wesentlicher Unterschied gemessen werden. Dieses Verhalten lässt sich folgendermaßen erklären. Die ZPU benötigt für die Übertragung einer Nachricht ca. 500 Taktzyklen. In dieser Zeit werden die Nachrichten der vier Sender trotz der Kollisionen übertragen, da die hier auftretenden Kollisionen innerhalb von 36 Taktzyklen aufgelöst werden können.

7.3.4 Fazit

Aufgrund der langsamen ZPU lassen sich kaum Aussagen über die Kommunikation im MPSoC treffen. Weder die IPC-Mechanismen, noch die Kommunikationsstrukturen können mit 16 ZPUs so beansprucht werden, als das sinnvolle Aussagen zur Einordnung getroffen werden können. Für Auswertungen dieser Art werden schnellere Soft Cores in dem MPSoC benötigt.

8 Zusammenfassung und Ausblick

Abschließend werden die im Zuge der Diplomarbeit entwickelten Konzepte und Ergebnisse in Kapitel 8.1 zusammengefasst und in Kapitel 8.2 werden einige Ideen und Anregungen für zukünftige Forschungen in Richtung der merkmalbasierten MPSoCs gegeben.

8.1 Zusammenfassung

In dieser Diplomarbeit wurde die merkmalbasierte Konfigurierung von Multiprozessorsystemen auf einem Chip behandelt. Dazu wurde in Kapitel 5, abgesehen von dem Soft Core, ein konfigurierbares MPSoC von Grund auf neu entwickelt. Der verwendete Soft Core ist ein kleiner stack-basierter 32 Bit Core namens ZPU [24]. Damit die ZPU in ein MPSoC integriert werden konnte, musste sie sowohl um eine Schnittstelle zur Kommunikation als auch um interne Zustände zur Ausführung neuer Befehle erweitert werden. Die Kommunikationsstrukturen sind für das entwickelte MPSoC austauschbar. Um diese Strukturen möglichst flexibel zu wechseln, wurde den ZPUs jeweils ein IPC-Controller vorgeschaltet. Die Schnittstelle der ZPU bleibt unverändert und muss nicht an die verschiedenen IPC-Controller angepasst werden. Für das realisierte MPSoC können drei verschiedene Mechanismen zur Kommunikation konfiguriert werden. Zum einen steht ein Message Queue Mechanismus zur Auswahl, für den Optionen, wie die Größe der Message Queue oder die Interaktion mit dem Soft Core, bei einer leereren Message Queue konfiguriert werden können. Ein weiterer implementierter Mechanismus ist der Shared Memory. Der Shared Memory bzw. der Controller des Shared Memory hängt zusammen mit den anderen IPC-Controllern am Bus und gibt abhängig von den Anfragen der Cores die angeforderten Daten auf den Bus oder speichert eingehende Daten im Shared Memory. Die Größe des Shared Memory kann konfiguriert werden. Zuletzt wurde noch ein Event Flag Mechanismus implementiert, mit dem Prozesse synchronisiert werden können. Für den Message Queue Mechanismus kann, anstelle eines Busses, auch eine Ring-Struktur zur Kommunikation konfiguriert werden.
In Kapitel 6 wurde untersucht, ob sich MPSoCs anhand von Merkmalen und den Ansätzen aus dem Software-Bereich konfigurieren lassen. Dazu wurde zuerst ein Konzept erstellt, wie sich der VHDL-Quellcode für die anschließende Konfigurierung zerlegen lässt. Hierbei hat sich das Konzept bewährt für alle Instanzen der zu erzeugenden Komponenten eine eigene Architektur unterhalb der Entity-Deklaration zu erzeugen. So kann eine Vorlage der Architektur dazu verwendet werden, durch einfache Textersetzungen in der Schablone die einzelnen Instanzen der Komponenten zu erzeugen. Anschließend wurden diese Ansätze auf die in Kapitel 3 vorgestellten Techniken zur Konfigurierung

von Software angewendet. Der erste Ansatz mit pure::variants in Kapitel 6.4.1 lässt nur eine maximale Anzahl von fünf Cores zu. Die geringe Anzahl der Cores hängt damit zusammen, dass in pure::variants Konzepte, wie Schleifen, nicht zur Verfügung stehen und so eine beliebige Core-Anzahl mit dem ersten Ansatz nicht zu realisieren ist. Um das MPSoC flexibler zu konfigurieren, wurde ein zweiter Ansatz mit pure::variants und XVCL in Kombination untersucht. Zudem wurde für den zweiten Ansatz das neue Konzept der Variant Instanzen von pure::variants eingesetzt. Mit diesem Konzept ist es möglich, beliebig viele Cores für ein MPSoC zu generieren. XVCL musste in diesem Ansatz verwendet werden, um die Top-Entity des MPSoCs zu erzeugen. Um die Konfigurierung in einem Vorgang durchzuführen, wurde XVCL über eine neue Transformation in pure::variants integriert. Dieser Ansatz war erfolgreich und lässt die Konfigurierung von MPSoCs mit beliebig vielen Cores zu.
In Kapitel 7 wurden die Ergebnisse dieser Diplomarbeit, neben dem konfigurierbaren MPSoC, vorgestellt. Dazu wurden unter anderem die eingesetzten Verfahren zur Konfigurierung anhand von Merkmalen bewertet. Merkmalmodelle, wie sie in pure::variants verwendet werden, sind zur Konfigurierung von MPSoCs geeignet, solange jeder Core nur ein IPC-Interface hat und folglich nur mit einer Kommunikationsstruktur verbunden ist. Für komplexere Strukturen, in der ein und derselbe Core mit verschiedenen Kommunikationsstrukturen verbunden ist, muss die Eignung von Merkmalmodellen zur Konfigurierung von MPSoCs noch untersucht werden. Desweiteren wurde der Ressourcenverbrauch auf den FPGAs getestet. Für das Virtex-II Pro Entwicklungsboard kann beispielsweise ein MPSoC mit 16 Cores erzeugt und implementiert werden. Die Kommunikationsstrukturen und IPC-Mechanismen konnten nicht sinnvoll miteinander verglichen werden, da die ZPU nur ca. 1 Prozent Buslast erzeugen kann.

8.2 Ausblick

Damit die implementierten IPC-Mechanismen und Kommunikationsstrukturen bewertet werden können, muss zukünftig neben der ZPU ein schnellerer Soft Core in das MPSoC eingebunden werden. Interessant wäre zudem die Integration eines NoCs in das MPSoC, wie beispielsweise in [21] umgesetzt und ein anschließender Vergleich des NoCs mit den bisherigen Kommunikationsstrukturen. Desweiteren kann nach der erfolgreichen Konfigurierung eines MPSoCs anhand von Merkmalmodellen untersucht werden, wie sowohl die Systemsoftware als auch die darunter liegende Hardware durchgängig konfiguriert werden können. Zudem besteht die Möglichkeit das eingebettete Betriebsystem eCos, das von der ZPU unterstützt wird, auf dem MPSoC zu testen.

Literaturverzeichnis

[1] Othman, Slim B. ; Ben Salem, Ahmed K. ; Ben Saoud, Slim: MPSoC Design of RT Control Applications based on FPGA SoftCore Processors. In: *15th IEEE International Conference on Electronics, Circuits and Systems*, 2008, S. 404–409

[2] Xilinx, Inc.: *MicroBlaze RISC 32-Bit Soft Processor*. August 2002

[3] Spinczyk, Olaf: *Software-Produktlinien*. http://ess.cs.tu-dortmund.de/DE/Teaching/WS2008/BST/Downloads/index.html, 2008

[4] Kang, Kyo C. ; Cohen, Sholom G. ; Hess, James A. ; Novak, William E. ; Peterson, A. S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study / Software Engineering Institute, Carnegie Mellon University Pittsburgh. 1990 (CMU/SEI-90-TR-21). – Technical Report

[5] Åkesson, Johan F.: *Interprocess Communication Utilising Special Purpose Hardware*. December 2001

[6] Salminen, Erno ; Kulmala, Ari ; Hämäläinen, Timo D.: Survey of Network-on-chip Proposals / Tampere University of Technology. 2008. – Forschungsbericht

[7] Bafumba-Lokilo, David ; Savaria, Yvon ; David, Jean-Pierre: Generic crossbar network on chip for FPGA MPSoCs. In: *Joint 6th International IEEE Northeast Workshop on Circuits and Systems and TAISA Conference*, 2008, S. 269 – 272

[8] Marwedel, Peter: *Eingebettete Systeme*. Springer, 2007

[9] Xilinx, Inc.: *Basic FPGA Architecture*. 2005

[10] Xilinx, Inc.: *Spartan-3E FPGA Starter Kit Board - User Guide, UG230 (v1.1)*. June 2008

[11] Xilinx, Inc.: *Xilinx University Program Virtex-II Pro Development System - Hardware Reference Manual, UG069 (v1.1)*. April 2008

[12] Jarzabek, Stan ; Bassett, Paul ; Zhang, Hongyu ; Zhang, Weishan: XVCL: XML-based Variant Configuration Language. In: *25th International Conference on Software Engineering*, 2003, S. 810 – 811

[13] Swe, Soe M. ; Zhang, Hongyu ; Jarzabek, Stan: XVCL: A Tutorial. In: *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, 2002, S. 341 – 349

[14] National University of Singapore ; Netron Inc.: *XML-based Variant Configuration Language (XVCL) - Specification Version 2.10.* http://xvcl.comp.nus.edu.sg/, June 2006

[15] Bassett, Paul G.: *Framing Software Reuse - Lessons From The Real World.* Prentice-Hall, Inc., 1996

[16] Spinczyk, Olaf: *Statische Konfigurierung.* http://ess.cs.tu-dortmund.de/DE/Teaching/WS2008/BST/Downloads/index.html, 2008

[17] Pure-Systems GmbH: *Technical White Paper, Variantenmanagement mit pure::variants.* http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf, 2006

[18] Pure-Systems GmbH: *pure::variants User's Guide, Version 3.0 for pure::variants 3.0.* http://www.pure-systems.com/fileadmin/downloads/pure-variants/doc/pv-user-manual.pdf, 2008

[19] Gaoming, Du ; Duoli, Zhang ; Yukun, Song ; Liang, Ma ; Ning, Hou ; Minglun, Gao: Performance evaluation of FPGA based crossbar NoC architecture. In: *9th International Conference on Solid-State and Integrated-Circuit Technology*, 2008, S. 2058 – 2061

[20] Zhang, Wei ; Du, Gao-Ming ; Xu, Yi ; Gao, Ming-Lun ; Geng, Luo-Feng ; Zhang, Bing ; Jiang, Zhao-Yu ; Hou, Ning ; Tang, Yi-Hua: Design of a Hierarchy-Bus Based MPSoC on FPGA. In: *8th International Conference on Solid-State and Integrated Circuit Technology* ICSICT '06, 2006, S. 1966 – 1968

[21] Lukovic, Slobodan ; Fiorin, Leandro: An Automated Design Flow for NoC-based MPSoCs on FPGA. In: *The 19th IEEE/IFIP International Symposium on Rapid System Prototyping*, 2008, S. 58 – 64

[22] Kumar, Akash ; Ovadia, Ido ; Huiskens, Jos ; Corporaal, Henk ; van Meerbergen, Jef ; Ha, Yajun: Reconfigurable Multi-Processor Network-on-Chip on FPGA. In: *Proceedings of the Annual Conference of the Advanced School for Computing and Imaging*, 2006

[23] Chu, Pong P.: *FPGA Prototyping By VHDL Examples - Xilinx Spartan-3 Version.* John Wiley & Sons, Inc., 2008

[24] Harboe Øyvind: *ZPU - the worlds smallest 32 bit CPU with GCC toolchain.* http://www.opencores.org/?do=project&who=zpu, November 2008

[25] Xilinx, Inc.: *Data2MEM User Guide, UG437 (v2.0).* http://www.xilinx.com/itp/xilinx92/books/docs/d2m/d2m.pdf, April 2007

Abbildungsverzeichnis