technische universität
dortmund

Diplomarbeit

Eine Familie von gemeinsamen Speichern für MPSoCs

David Austin

2. September 2010

Betreuer:
Prof. Dr.-Ing. Olaf Spinczyk
Matthias Meier

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12
Arbeitsgruppe Eingebettete Systemsoftware
`http://ess.cs.tu-dortmund.de`

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

Dortmund, den 2. September 2010

David Austin

Zusammenfassung

Die heutigen FPGAs umfassen eine Vielzahl von konfigurierbaren Logikzellen. Dabei steigt die Komplexität der auf ein FPGA abbildbaren Systeme stetig. So können Multiprozessorsysteme, inklusive Kommunikationsstrukturen und Komponenten für spezielle Aufgaben, auf einem FPGA untergebracht werden. Mittels Hardwarebeschreibungssprachen wie VHDL oder Verilog lassen sich heterogene MPSoCs entwerfen, die unterschiedlichste Komponenten umfassen. Diese Diplomarbeit befasst sich im Rahmen des LavA-Projekts mit der Erstellung einer Speicher Familie, die es ermöglicht anhand von Merkmalen ein auf ein Problem zugeschnittenes System zu erzeugen. Das Konzept von Produktfamilien wird in der Softwareentwicklung bereits angewendet und soll hier auf die Entwicklung der Speicher Familie übertragen werden.

Inhaltsverzeichnis

1 Einleitung

Heutige integrierte Schaltkreise können bis zu hundert Millionen Transistoren pro Chip umfassen und ermöglichen es damit FPGAs[1] mit einer Vielzahl an konfigurierbaren Logikzellen auszustatten. So ist es möglich komplexe Systeme auf ein FPGA abzubilden. Mehrere durchaus unterschiedliche Prozessoren, Speicher und dazugehörige Peripherie, wie Kommunikationsstrukturen und spezialisierte Komponenten, können auf einem Chip untergebracht werden. Multiprozessorsysteme auf einen Chip (MPSoC) werden bereits seit einigen Jahren eingesetzt, so ist z. B. der Nexperia PNX8526 [1] im Jahr 2004 erschienen.

Mit Hilfe von Hardwarebeschreibungssprachen wie VHDL[2] oder Verilog werden diese Systeme entworfen. Die MPSoCs können so in Kombination mit FPGAs aus Basiskomponenten, beispielsweise aus Intellectual Properties, zusammengesetzt und an die Anforderungen der Anwendung angepasst werden. Weiterhin profitieren die modernen FPGAs von effizient integrierten CPUs, SRAM Speicher oder Hardware-Multiplizierern. So sind auf dem Xilinx XC2VP30 FPGA bereits 2 PowerPC-Prozessoren, über 300kB SRAM und 136 18Bit Multiplizierer vorhanden. Bei diesen Hardwarestrukturen handelt es sich um sogenannte Hard Cores, die platzsparend und mit hohen Taktraten betrieben, allerdings nicht weiter angepasst werden können. Zusätzliche Funktionen für Spezialanwendungen oder zusätzliche CPUs, sogenannte Soft Cores, können über die konfigurierbaren Logikzellen realisiert werden.

Je größer solch ein System wird, desto komplexer wird es. Um der steigenden Komplexität gerecht zu werden, sind Entwicklungsmethoden zum strukturierten Arbeiten nötig. In der Softwareentwicklung wird bereits das Konzept der Produktfamilien [2] angewendet. Dabei spielt das identifizieren von wiederverwendbaren Komponenten eine große Rolle. Die Klassifizierung in variable und gemeinsame Merkmale ermöglicht das strukturierte Wiederverwenden von Komponenten. Dieses Konzept soll im Rahmen dieser Diplomarbeit auf den Entwurf einer Familie von gemeinsamen Speichern übertragen werden.

1.1 Motivation

Im Rahmen des LavA-Projekts [3] der Arbeitsgruppe Eingebettete System Software am Lehrstuhl 12 der Fakultät für Informatik an der TU Dortmund werden sowohl die Software- als auch Hardwarestrukturen mit konzeptionell gleichen Techniken konfiguriert. Mittels AspectC++[4] ist es bereits gelungen das Betriebssystem CiAO [5]

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

anhand von Merkmalen feingranular konfigurierbar zu gestalten. Ähnlich soll nun mit Hilfe von XVCL [6, 7, 8], eine Auszeichnungssprache basierend auf XML, generische Hardware-Komponenten entwickelt werden, so dass eine feingranulare Konfigurierung des Gesamtsystems möglich ist. Die Diplomarbeit von Matthias Meier [9] befasste sich bereits mit diesem Thema. Weiterführend untersucht diese Arbeit die Möglichkeiten die XVCL zum Beschreiben von konfigurierbaren VHDL-Komponenten bietet. Dazu wird das im LavA-Projekt bereits vorhandene Grundgerüst für MPSoCs um eine Speicher Familie erweitert. Seit einigen Jahren ist ein neues Speichermodell für den Zugriff durch mehrere Prozessoren auf einen gemeinsamen Speicher in das Interesse der Forschung gerückt: Transaktionsspeicher. Ähnlich dem Konzept von Transaktionen bei Datenbanken bietet dieses Speichermodell eine Vereinfachung des Programmiermodells. Aufgrund der unterschiedlichen Ausprägungen die der Transaktionsspeicher haben kann, wird es möglich eine für die Anwendung angepasste und anhand von Merkmalen bestimmte Instanz zu generieren.

1.2 Zielsetzung

Die Speicher Familie soll neben dem Transaktionsspeicher auch weitere Alternativen für die effiziente Nutzung der Ressourcen umfassen. Für das klassische Speichermodell, bei dem mittels atomarer *read-modify-write* Speicheroperationen unterschiedliche Threads synchronisiert werden, ist ein Speicher-Controller notwendig. Darüber hinaus sind gerade bei stufenweiser Verarbeitung von Daten einfache FIFO-Speicher interessant, die den Datenstrom zwischen Produzenten und Konsumenten puffern. Dazu muss zunächst ermittelt werden welche Ausprägungen die einzelnen Konzepte gemeinsamer Speicher annehmen können, um dann Gemeinsamkeiten und Unterschiede festzuhalten. Die so erkannten Merkmale wie z. B. Speichergröße, Schnittstellen zur CPU oder Busbreite werden schließlich, mit den Möglichkeiten die VHDL bietet, implementiert. Dabei sollen die Beschränkungen der Konfigurierbarkeit von VHDL durch XVCL ergänzt werden, so dass über die Merkmalsauswahl eine an die Anwendung angepasste System-Instanz generiert werden kann. Im weiteren Verlauf der Diplomarbeit gilt es dann die Systeme zu testen und einer Evaluierung zu unterziehen. Dabei muss die Konfigurierbarkeit der Speicher Familie, die Tauglichkeit von XVCL als Konfigurations-Tool und die Leistung der Speicher Familie einer Bewertung unterzogen werden.

1.3 Aufbau der Arbeit

Zunächst werden in Kapitel 2 Grundlagen zum Verständnis der Arbeit präsentiert. Es wird die LavA-Plattform vorgestellt, in die die Speicher Familie eingebettet werden soll. Weiterhin wird auf die Verwendung von Merkmalmodelle, die Grundlagen zur Konfigurierung mit Hilfe von VHDL und Transaktionsspeicher eingegangen. Anschließend werden in Kapitel 3 die eingesetzten FPGAs, sowie die Programme für die Synthese und Simulation besprochen. Außerdem wird die Auszeichnungssprache XVCL erläutert.

Ähnliche Arbeiten zur Konfigurierung von MPSoCs und Transaktionsspeichern werden in Kapitel 4 untersucht. Eigenschaften der Speicher Familie werden in Kapitel 5 näher betrachtet und so Gemeinsamkeiten und Unterschiede zwischen den unterschiedlichen Speichermodellen herausgearbeitet. Die Implementierung der Familie von gemeinsamen Speichern wird in Kapitel 6 vorgestellt. Kapitel 7 beschreibt, wie die Konfigurierbarkeit umgesetzt wird. Die Evaluierung und Bewertung findet in Kapitel 8 statt. Abschließend werden in Kapitel 9 die Ergebnisse dieser Diplomarbeit zusammengefasst und Anregungen für weitere Arbeiten gegeben.

2 Grundlagen

Dieses Kapitel beschäftigt sich mit einigen Grundlagen, die zum Verständnis der nachfolgenden Kapitel beitragen sollen. Zunächst wird im Kapitel 2.1 das LavA-Projekt vorgestellt. Merkmalmodelle unterstützen den Entwurf von Produktfamilien und werden in Abschnitt 2.2 besprochen. Das Unterkapitel 2.3 befasst sich mit der im Folgenden benutzten Hardwarebeschreibungssprache VHDL, wobei die Beschreibungsmöglichkeiten von generischen Komponenten im Vordergrund stehen. Das Unterkapitel 2.4 beschreibt die Funktionsweise von Transaktionsspeichern. Die verwendete FPGA-Technologie wird im letzten Abschnitt 2.5 dieses Kapitels erläutert, so dass die mit dieser Technologie in Verbindung stehenden Begriffe besser eingeordnet werden können.

2.1 Die LavA-Plattform

Das LavA-Projekt [3] befasst sich mit der Entwicklung einer Systeminfrastruktur, die es ermöglicht sowohl die Softwarebestandteile als auch die benötigte Hardware statisch konfigurierbar zu gestalten. Dabei werden zum Konfigurieren der Soft- und Hardware ähnliche Konzepte bzw. Techniken eingesetzt. Das Wissen über die zu implementierende Anwendung kann zur Maßschneiderung aller benötigten Systemkomponenten eingesetzt werden. Im Gegensatz zu klassischen Ansätzen kann die Software auf eine konfigurierbare Hardwareplattform zurückgreifen, so dass ein möglichst hohes Maß an Parallelität und Effizienz erreicht wird, ohne die Software an unterschiedliche und für bestimmte Anwendungen spezialisierte Multicore-Architekturen anpassen zu müssen.

Die LavA-Plattform umfasst eine konfigurierbare Hardwareplattform und ein entsprechend anpassbares Betriebssystem. Wie in Abbildung 2.1 zu sehen ist, wird eine LavA-Plattform Instanz durch die Analyse des Applikationscodes, aus den benötigten Komponenten der LavA-Plattform, generiert. Ein Metamodell beschreibt die generierbaren LavA-Plattform Instanzen. Durch die Benutzung der Betriebssystem- und Hardware-API können automatisch die benötigten Betriebssystem- und Hardwarekomponenten ermittelt und generiert werden. So ist es möglich ein maßgeschneidertes System zu erhalten, welches nur über benötigte Komponenten verfügt. So werden überflüssige Betriebssystem- und Hardwarekomponenten weggelassen und können damit zur Reduktion des benötigten Speichers sowie anderer benötigter Hardwareressourcen beitragen.

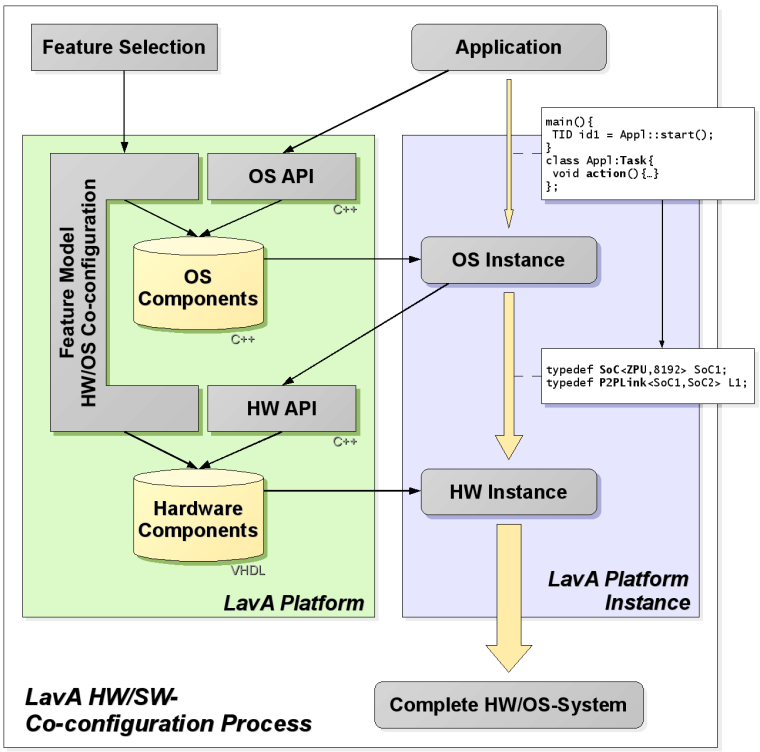

Abbildung 2.1: LavA-Plattform [10]

2.2 Das Merkmalmodell

Mit Hilfe von Merkmalmodellen kann die Variabilität eines Systems ausgedrückt werden. Sie werden beispielsweise bei der Entwicklung von Produktfamilien [2] in der Softwareentwicklung eingesetzt. Die Aufteilung des zu entwickelnden Systems in Merkmale erleichtert das Identifizieren von wiederverwendbaren Komponenten. Das Merkmalmodell spiegelt konfigurierbare und wiederverwendbare Eigenschaften des Systems wieder. Zum visualisieren kann ein Merkmaldiagramm erstellt werden.

Der Wurzelknoten eines Merkmaldiargramms wird auch als Konzeptknoten bezeichnet und ist immer Bestandteil einer Instanz des Konzepts. Alle weiteren Kindknoten bilden die Merkmale des Konzepts. Dabei unterscheidet man zwischen optionalen, dargestellt durch einen nicht ausgefüllten Kreis, und notwendigen Merkmalen, dargestellt durch einen ausgefüllten Kreis. Ein Merkmal gehört zum Konzept, wenn es ausgewählt wurde und der Elternknoten bereits zum Konzept gehört. Kindknoten eines Elternknoten können in Gruppen zusammengefasst werden. Ein nicht ausgefüllter Kreisbogen symbolisiert alternative Merkmale, von denen genau eines, oder im Falle von optionalen Merkmalen auch keines, ausgewählt werden muss. Durch das Ausmalen des Kreisbogens wird eine Oder-Beziehung ausgedrückt, so dass mindestens eines der Kindknoten der Gruppe Teil des Konzepts wird.

Abbildung 2.2 zeigt beispielhaft das Merkmaldiagramm eines PCs. Jeder PC muss über eine Hauptplatine, Steckplätze für Arbeitsspeicher auf der Hauptplatine, eine CPU

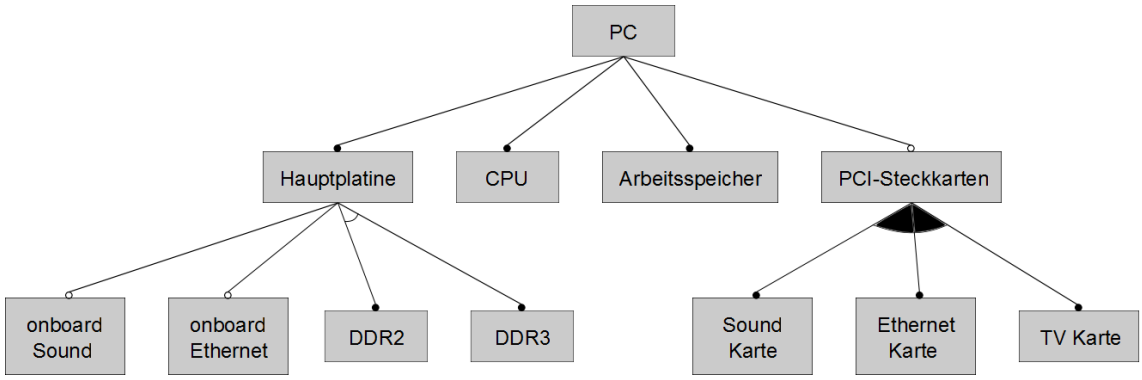

Abbildung 2.2: Merkmalmodell

und Arbeitsspeicher verfügen, weshalb diese Merkmale notwendig sind. Den Steckplätzen des Arbeitsspeichers wiederum können unterschiedliche Technologien zugrunde liegen, welche dann alternative Merkmale darstellen. Auf der anderen Seite sind auf der Hauptplatine integrierte Chips für Sound und Netzwerk sowie PCI-Steckkarten optionale Komponenten eines PCs. Es sind eine Vielzahl an PCI-Steckkarten verfügbar, von denen durchaus auch mehrere vorhanden sein können und damit hier durch eine Oder-Verknüpfung verbunden sind. Neben dieser grafischen Darstellung können auch weitere Informationen in Form von textuellen Beschreibungen vorliegen, die die Semantik eines Merkmals beschreiben oder Einschränkungen wiedergeben. So müsste zu dem Merkmalmodell in Abbildung 2.2 vermerkt werden, dass der Standard für den Arbeitsspeicher und deren Steckplätze auf der Hauptplatine derselbe sein muss. Darüber hinaus können auch weitere Beschreibungen, wie eine Begründung warum gewisse Merkmale vorhanden sind oder Beispielsysteme in denen bestimme Merkmale schon vorgekommen sind, zusätzliche Informationen bieten.

2.3 Konfigurierbarkeit in VHDL

Die einzelnen Funktionseinheiten einer digitalen Schaltung werden in der Hardwarebeschreibungssprache VHDL in sogenannte *Design Entities* unterteilt. Jede Design Entity verfügt dabei über eine *Entity Declaration* und mindestens eine *Entity Architecture*. Die Entity Declaration bestimmt die Schnittstelle der Design Entity. Das Verhalten wird über die Architektur definiert. Das Angeben unterschiedlicher Architekturen für eine Design Entity stellt damit die erste Möglichkeit dar, ein anpassbares System zu generieren. Es gibt im wesentlichen zwei Varianten eine passende Architektur für das aktuelle Design auszuwählen: über eine sogenannte *Konfigurations-Deklaration* oder direkt bei der Instanziierung einer Design Entity. Der Unterschied besteht darin, dass eine Konfigurations-Deklaration in einer zentralen Datei abgelegt werden kann und so zu allen Design Entities passende Architekturen ausgewählt werden können. Im folgenden wird ausschließlich mit der Auswahl der Architektur bei der Instanziierung gearbeitet, da diese in Kombination mit *Generics* verwendet werden können.

Listing 2.1 zeigt eine einfache Entity Deklaration. In diesem Fall umfasst die Schnittstelle einen generischen Parameter und vier Ports, die der Kommunikation dienen. Der generische Parameter bitLaenge wird hier benutzt, um die Breite der Ports summand_1, summand_2 und summe anzugeben, welches eine der gängigsten und einfachsten Möglichkeiten zur Anpassung einer Design Entity darstellt. Für komplexere Abhängigkeiten können Funktionen eingesetzt werden, um beispielsweise die benötigte Bitbreite eines Ports von mehreren generischen Parametern, die nicht notwendigerweise nummerische Werte sein müssen, zu berechnen.

Listing 2.1: VHDL Entity-Deklaration

```
entity addierer is
    generic (
        bitLaenge : POSITIVE := 32 );
    port (
        summand_1   : in  SIGNED(bitLaenge - 1 downto 0);
        summand_2   : in  SIGNED(bitLaenge - 1 downto 0);
        summe       : out SIGNED(bitLaenge - 1 downto 0);
        uebertrag   : out STD_LOGIC );
end entity;
```

Innerhalb der Architektur gibt es weitere Möglichkeiten, abhängig von Generics oder Konstanten, unterschiedliche Schaltungen zu realisieren. Die *generate*-Anweisung kann, entweder nach auswerten einer booleschen Bedingung oder als Schleife mit konstanten Schleifengrenzen, zusätzliche Logik generieren. Dabei können beide Formen, wie sie in Listing 2.2 zu sehen sind, ineinander verschachtelt vorkommen. Der deklarative Teil der generate-Anweisung kann nun z. B. dazu genutzt werden eine passende Architektur auszuwählen. Die *for. . . loop*-Anweisung stellt eine Schleife innerhalb von Prozessen, Funktionen oder Prozeduren dar, die im Gegensatz zur generate-Anweisung sequentiellen Code beinhaltet.

Listing 2.2: VHDL generate-Anweisung

```
generate_label_1 : if boolesche_bedingung generate
    Deklarationen
begin
    paralleler Code
end generate;

generate_label_2 : for i in 0 to bitlaenge generate
    Deklarationen
begin
    paralleler Code
end generate;

-- innerhalb von prozessen
for i in 0 to bitlaenge loop
    sequentieller Code
```

```
end loop;
```

2.4 Transaktionsspeicher

Bereits 1976 wurde in [11] das Konzept von Transaktionen im Rahmen von Datenbanken beschrieben. Im vergangenen Jahrzehnt wurden einige Forschungsergebnisse [12, 13, 14, 15, 16, 17, 18, 19] rund um Transaktionsspeicher veröffentlicht. Die Idee dahinter ist die Vereinfachung des Programmiermodells und eine effizientere Nutzung von Multiprozessor-Systemen. Üblicherweise werden Programme mit mehreren Threads über Locks, Monitore oder ähnliche Verfahren synchronisiert. Dabei werden kritische Bereiche durch exklusiven Zugriff gesichert und bewahren so einen konsistenten Zustand. Nachteile die sich dadurch ergeben sind zum einen, dass die mögliche parallele Ausführung unterbunden wird und zum anderen, dass das Programmieren unter Berücksichtigung vieler Locks fehleranfällig ist.

Bei Verwendung eines Transaktionsspeichers (eng. transactional memory, TM) kann der Programmierer Zugriffe auf kritische Bereiche des Programms in Transaktionen kapseln. Mit Funktionen oder Makros (z. B. TM_BEGIN() und TM_COMMIT()) werden Codebereiche eingeschlossen, die als eine Transaktion ausgeführt werden sollen. So wird sichergestellt, dass sämtliche Lese- und Schreib-Zugriffe innerhalb dieser Transaktion atomar erscheinen. Dabei kann eine laufende Transaktion i. d. R. durch einen Abort-Befehl wieder abgebrochen werden.

Während der Ausführung von parallel laufenden Transaktionen werden Konflikte automatisch erkannt und gelöst. Sobald mindestens zwei Transaktionen auf die selbe Speicheradresse zugreifen und mindestens einer davon ein schreibender Zugriff ist, liegt ein Konflikt vor. Zum Lösen des Konflikts wird eine *Policy* festgelegt, die während des Betriebs automatisch entscheidet, welche Transaktionen abgebrochen werden müssen. Der Abbruch einer Transaktion führt zu einem sogenannten *Revert*, der alle vorgenommenen Änderungen am Speicherzustand rückgängig machen muss.

Bei der Konflikterkennung wird im Wesentlichen zwischen zwei Strategien unterschieden. Die pessimistische Konflikterkennung überprüft vor dem eigentlichen Speicherzugriff zunächst, ob dieser konfliktfrei getätigt werden kann. Falls dies nicht möglich ist, werden direkt Maßnahmen zum Lösen des Konflikts eingeleitet. Im Falle einer optimistischen Konflikterkennung werden bis zum Ende einer Transaktion keine Überprüfungen vorgenommen. Im Rahmen des Commits muss zunächst die Menge der gelesenen (Read-Set) und geschriebenen (Write-Set) Adressen mit denen der anderen Transaktionen abgeglichen werden. Nur wenn sich während dieser Validierungsphase keine Konflikte ergeben, können die Änderungen dieser Transaktion sichtbar gemacht werden.

Um einen ursprünglichen Speicherzustand wiederherstellen zu können, ist es notwendig den alten Wert bis zum erfolgreichen Abschluss einer Transaktion zu sichern. Für diesen Zweck muss ein Zwischenspeicher existieren, der entweder den neuen Wert puffert oder den alten Wert sichert. Beim puffern der neuen Werte wird von einer *lazy* Versionierung gesprochen, so dass erst im Zuge des Commits und nach dem erfolgreichen Validieren der Transaktion die neuen Werte in den Speicher geschrieben werden. Im Gegensatz

dazu wird bei der *eager* Versionierung der alte Wert im Zwischenspeicher hinterlegt und anschließend im Speicher vom neuen Wert überschrieben. Nur falls es zu einem Revert kommt, müssen alle im Zwischenspeicher gesicherten Werte wieder zurück in den Speicher geschrieben werden.

Im Konfliktfall gibt es eine Vielzahl an Strategien, um zu entscheiden welche Transaktion abgebrochen werden soll. Bobba et al. [17] haben für einige Kombinationen von Policy, Versionierung und Konflikterkennung Probleme aufgezeigt, die die Leistung des Transaktionsspeichers stark beeinflussen. Daher sind für diese Arbeit zwei robuste Strategien relevant, um das TM-System testen zu können:

- *Attacker-Wins* [20] – die Transaktion, die gerade auf den gemeinsamen Speicher zugreift, darf fortfahren
- *Oldest-Wins* [13] – die Transaktion, die am ältesten ist, darf fortfahren

2.5 FPGA-Technologie

Ein FPGA-Chip besteht im Wesentlichen aus konfigurierbaren Logikzellen, die eine Reihe von booleschen Funktionen realisieren können. Mittels konfigurierbaren Verbindungen zwischen diesen Zellen ist es möglich beliebige digitale Schaltungen zu implementieren. Da die konkrete Ausstattung und Terminologie vom Hersteller abhängt, wird zur Erläuterung der Technik ein Xilinx FPGA der Virtex-5 Familie vorgestellt.

Abbildung 2.3 zeigt den schematischen Aufbau eines Xilinx FPGAs. Hauptbestandteil eines Virtex-5 FPGAs bilden die konfigurierbaren Logik-Blöcke (CLB), die wiederum aus jeweils 2 Slices bestehen. Innerhalb eines Slice befinden sich vier Look-Up Tables (LUTs), die jeweils eine beliebige boolesche Funktion mit sechs Eingangs- und zwei Ausgangssignalen realisieren können. Des Weiteren umfasst jedes Slice vier Flip-Flops und kann sowohl für synchrone als auch asynchrone Schaltungen verwendet werden. Zusätzlich verfügt das FPGA über eine Reihe von Hardwaremultiplizierern, Dual-Port SRAM Blöcken (Block-RAM), Taktmanagementelementen sowie integrierten PowerPCs. Die Ein-/Ausgabe-Blöcke (IOBs) am Rand des Chips werden für Verbindungen aus bzw. ins FPGA genutzt. Mit Hilfe von Synthese-Tools kann schließlich eine Hardwarebeschreibung in eine Konfiguration des FPGAs überführt werden, die die gewünschte Funktionalität implementiert.

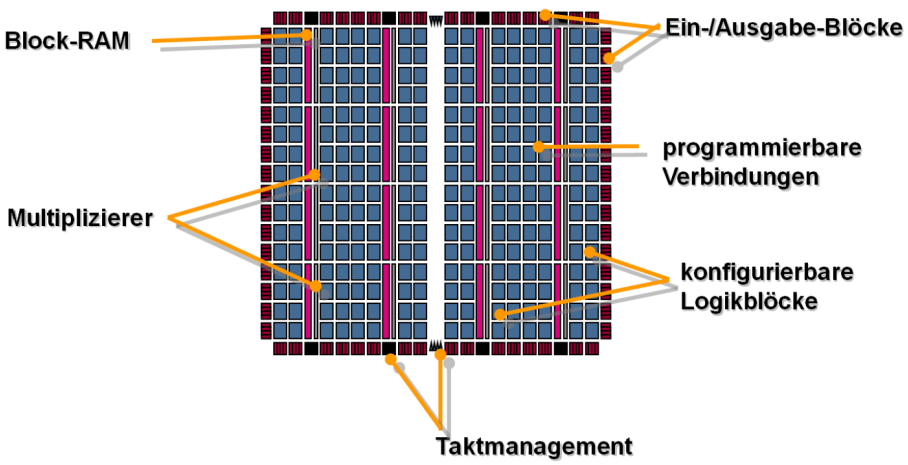

Abbildung 2.3: Aufbau eines FPGAs [21]

3 Verwendete Tools und Hardware

In diesem Kapitel werden zunächst im Abschnitt 3.1 die verwendeten FPGA-Entwicklungsboards beschrieben. Das nachfolgende Unterkapitel 3.2 stellt die benutzten Tools zum Simulieren des VHDL-Codes vor. Abschließend beinhaltet Abschnitt 3.3 eine Einführung in die Auszeichnungssprache XVCL.

3.1 Xilinx Entwicklungsboards

Zum Testen der Speicher Familie kommt neben dem Simulieren der Hardware auch die Ausführung auf Hardware zum Einsatz. Dazu werden zwei FPGA-Entwicklungsboards der Firma Xilinx eingesetzt.

3.1.1 Spartan 3E 1600E

Das Sparatan 3E 1600E Entwicklungsboard [22] besitzt den größten FPGA der Spartan-Familie. Mit 3.688 CLBs oder 29.504 LUTs lassen sich problemlos kleinere MPSoCs, mit etwa 7-9 MBLite Soft Cores [23], realisieren. Durch die Digital Clock Manager (DCM) können diese Soft Cores mit unterschiedlichen Taktraten betrieben werden. Der limitierende Faktor dieses Boards stellen die eher wenigen Block-RAMs dar. Mit nur rund 72kB integriertem RAM lassen sich nur kleinere MPSoCs unterbringen. Dies reicht aber völlig aus, um die entwickelte Hardware schnell zu synthetisieren und zu testen bzw. zu debuggen. In Tabelle 3.1 ist die Ausstattung nochmal zusammengefasst.

3.1.2 Virtex 5 Development Platform

Die XUPV5-LX110T Development Platform [24] ist ein Entwicklungsboard, bestückt mit einem FPGA der Virtex 5 Familie. Mit der dreifachen Kapazität eines Sparatans lassen sich auf diesem Board auch größere MPSoCs untersuchen. Gerade die über 500kB SRAM sind nötig, um auch Benchmarks zum Testen der Skalierbarkeit der Speicher Familie unterbringen zu können.

CLBs	LUTs	Block-RAMs	RS-232	Clock (MHz)	DCM
3.688	29.504	36	2	50	8

Tabelle 3.1: Ausstattung des Sparatn 3E 1600E Entwicklungsboards

CLBs	LUTs	Block-RAMs	RS-232	Clock (MHz)	DCM
8.640	69.120	128	1	100	12

Tabelle 3.2: Ausstattung der Virtex 5 Development Platform (XUPV5)

3.1.3 Xilinx Synthese Tools

Die Entwicklungsumgebung ISE 10.1 von Xilinx umfasst einige Programme, um den VHDL-Code zu synthetisieren und anschließend, für das eingesetzte FPGA, zu implementieren. Die Synthese übernimmt XST und überführt den VHDL-Code in eine Repräsentation auf der Registertransferebene. Anschließend werden durch NGDBuild, im Rahmen der Implementierung, die Dateien aus der Synthese und zusätzliche Design Einschränkungen in eine Xilinx spezifische Beschreibung umgewandelt. Die verschiedenen Komponenten werden nun FPGA Elementen zugeordnet (mapping) und danach übernimmt das Programm PAR die Platzierung und das Routing auf dem FPGA. Im letzten Schritt generiert BitGen eine Bit-Datei, welche zum Konfigurieren des FPGAs genutzt wird.

3.2 GHDL und GTKWave

Zum Simulieren der entworfenen Hardware kann GHDL [25] eingesetzt werden. GHDL ist ein VHDL-Simulator, der den VHDL-Code in eine ausführbare Simulationsdatei übersetzt. Diese kann anschließend ausgeführt werden und erzeugt dabei ein Protokoll des Simulationslaufs, die die Signaländerungen des Designs wiedergibt. Zum Visualisieren des Protokolls kann GTKWave verwendet werden. Um ein Design simulieren zu können, muss eine VHDL-Entity (auch Testbench genannt) erstellt werden, die keinerlei Eingangs- oder Ausgangssignale besitzt. Dort müssen auch alle vom Design benötigten Signale generiert werden.

Zum Simulieren eines Designs muss zunächst ein GHDL-Projekt erzeugt werden, indem alle benötigten VHDL-Dateien hinterlegt sind. Durch Aufrufen des GHDL-Simulators mit “gdhl −i Liste_der_Dateien” wird ein Projekt, in Form der work-obj93.cf Datei, angelegt. Anschließend kann mit “ghdl −m −−ieee=synopsys −fexplicit DesignName” eine neue ausführbare Simulationsdatei erzeugt werden. Nur Dateien die sich seit dem letzten Aufruf geändert haben, werden dabei neu übersetzt. Ein Simulationslauf wird mittels “ghdl −r DesignName −−vcd=output.vcd −−stop−time=100us” angestoßen, der das Design für 100 Mikrosekunden simuliert. Abschließend kann GTKWave (gtkwave output.vcd) genutzt werden, um das Ergebnis der Simulation anzusehen.

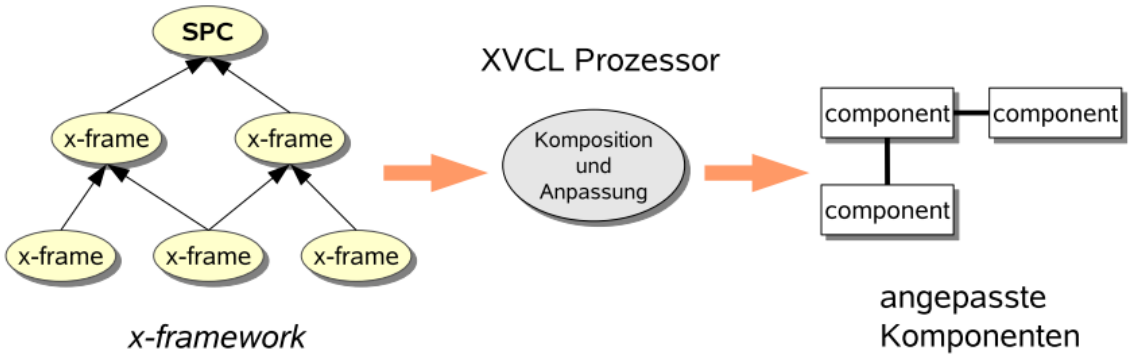

Abbildung 3.1: XVCL Konzept [8]

3.3 XVCL

Bei XVCL[1] [6, 7, 8] handelt es sich um eine XML basierte Auszeichnungssprache, die beim Entwickeln von Systemen auf textbasierten Sprachen zusätzliche Möglichkeiten eröffnet. Der Einsatz von Java und XML macht XVCL unabhängig von der benutzten Entwicklungsplattform und der eigentlich benutzten Sprache. Durch identifizieren von wiederverwendbaren oder konfigurierbaren Komponenten kann XVCL dazu beitragen, diese strukturiert und anpassbar aufzubereiten. Hierbei kommt die von Bassetts entwickelte Frame-Technologie [26] zum Tragen, indem das System in Meta-Komponenten (x-frames) aufgeteilt wird.

Die Unterteilung in x-frames ermöglicht das individuelle Anpassen von Komponenten, dabei kann jedes x-frame weitere x-frames einbinden. So entsteht eine x-frame Hierarchie, deren Wurzel kurz SPC (Specification x-frame) genannt wird. Der XVCL-Prozessor interpretiert die XVCL-Befehle und erzeugt so den gewünschten und angepassten Quelltext. Dabei werden die XVCL-Befehle in den eigentlichen Quelltext eingebettet. Abbildung 3.1 stellt dieses Konzept grafisch dar.

3.3.1 XVCL Verarbeitungsprozess

Der XVCL Prozessor startet seine Verarbeitung beim angegebenen Specification x-frame. Innerhalb eines x-frames interpretiert dieser alle Befehle und erzeugt entsprechende Ausgaben, bis ein <*adapt*>-Befehl erreicht wird. An dieser Stelle springt die Ausführung in das vom <*adapt*>-Befehl vorgegebene x-frame. Sobald das Ende eines x-frames erreicht ist, kehrt der XVCL Prozessor zum Aufrufer zurück und setzt seine Arbeit nach dem entsprechenden <*adapt*>-Befehl fort. Dabei ist es möglich einen x-frame mehrmals einzubinden, allerdings dürfen dabei keine Kreise in der Verarbeitung entstehen. Sofern keine Fehler auftreten, endet die Generierung des Quelltextes am Ende des SPCs. Ab-

[1] XML-based Variant Configuration Language

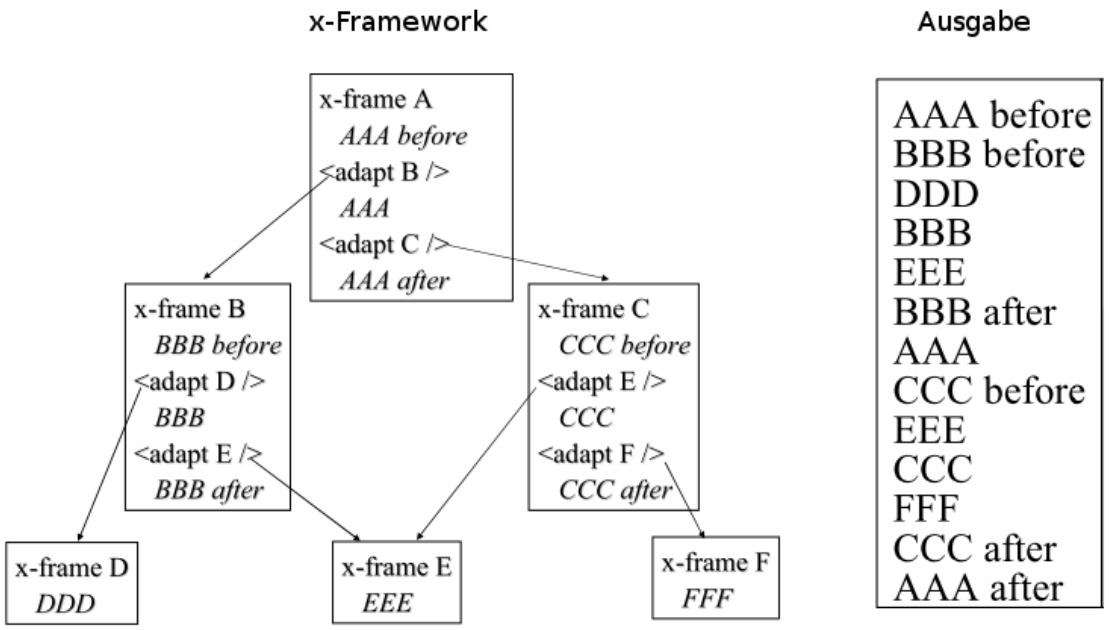

Abbildung 3.2: XVCL Verarbeitungsprozess [8]

bildung 3.2 veranschaulicht nochmal den Verarbeitungsprozess der x-frame Hierarchie.

3.3.2 XVCL-Ausdruck

An vielen Stellen kann als Wert für Attribute ein XVCL-Ausdruck verwendet werden. Dabei können sie zum Rechnen oder als Zeichenkette benutzt werden. Sofern gerechnet werden soll, darf der Ausdruck nur Zahlen, Variablen (als Wert sind nur Zahlen zugelassen) und die Operatoren +, -, * oder / enthalten. Um den Wert einer Variable zu erhalten, muss dem Namen der Variable innerhalb des Ausdrucks ein '@' vorangestellt werden. Um die Variablennamen von einfachen Zeichenketten unterscheiden zu können, müssen die Referenzen auf Variablen mit '?' eingeschlossen werden und bilden so eine *Name-Expression*. Es ist auch möglich mehrere Indirektionsstufen zu erzeugen, indem der Referenz weitere '@' voranstellt werden. So kann z. B. in einer Variable foo der Name einer weiteren Variable bar enthalten sein, so dass durch ?@@foo? auf den Wert der Variable bar zugegriffen werden kann. Darüber hinaus ist es möglich aus mehreren Variablen und konstanten Zeichenketten neue Variablennamen zu generieren. ?@bar@foo? referenziert in diesem Beispiel also die Variable mit dem Namen barbar. Die Auswertung erfolgt dabei immer von rechts nach links. Es können innerhalb eines Ausdrucks auch mehrere Name-Expressions und gewöhnliche Zeichenketten vorkommen.

3.3.3 XVCL-Befehle

Die XVCL-Befehle sind in XML-Elemente gekapselt, von denen einige mit und einige ohne Inhalt daher kommen. Sie sind an Sprachkonstrukte von Programmiersprachen angelehnt. Eine Liste mit den wichtigsten Befehlen folgt:

- **<x-frame>**

  Der *<x-frame>*-Befehl bildet die Wurzel eines XVCL-Dokuments. Er wird dazu genutzt die Meta-Komponenten festzulegen. Als Attribute können die Ausgabedatei, der Ausgabeordner, der Name des x-frames und eine Sprache[2] angegeben werden. Der Inhalt kann Quelltext oder weitere XVCL-Befehle enthalten.

```
<x-frame name = "x-frame-Name" [ outdir = "Ausgabeordner" ]
     [ outfile = "Ausgabedatei"] [ language = "Sprache"]>
...
</x-frame>
```

- **<adapt>**

  Um weitere x-frames einbinden zu können, wird der *<adapt>*-Befehl benötigt. Auch hier besteht die Möglichkeit die Ausgabedatei und den Ausgabeordner für den einzubindenden x-frame anzugeben. Diese Option überschreibt mögliche Angaben im x-frame. Das Attribut *samelevel* lässt die Variablen des eingebundenen x-frames nach Abarbeitung weiter bestehen. Durch setzen von *once* auf yes werden alle folgenden *<adapt>*-Befehl, die den selben x-frame einbinden sollen, ignoriert. Schließlich kann mit dem Attribut *src* angegeben werden, ob der Inhalt durch den XVCL Prozessor interpretiert werden soll. Als Inhalt sind nur *<insert>*-Befehle erlaubt.

```
<adapt  x-frame="Pfad/Dateiname"  [outdir="Ausgabeordner"]
    [outfile="Ausgabedatei"] [samelevel="yes-no"]
    [once= "yes-no"] [src="yes-no"]>
...
</adapt>
```

- **<insert> und <break>**

  Die Befehle *<insert>* und *<break>* werden dazu genutzt, um den einzubindenden x-frame anzupassen. Dabei definiert ein *<break>*-Befehl die Stelle an der zusätzlicher Quellcode oder sogar XVCL-Befehle eingefügt werden können. Jeder Breakpoint benötigt einen Namen und kann einen beliebigen Inhalt haben. Der *<insert>*-Befehl ersetzt ggf. den Inhalt des angegebenen Breakpoints und darf nur innerhalb eines *<adapt>*-Befehls vorkommen. Es existieren zwei weitere Varianten: *insert-before* und *insert-after*. Diese werden entsprechend direkt vor bzw. direkt nach dem Breakpoint eingefügt, dabei wird nichts im adaptierten x-frame ersetzt.

[2] Dieses Attribut hat momentan keine Wirkung

```
<break name="break-Name">
...
</break>
<insert break="break-Name">
...
</insert>
```

- **<set> und <set-multi>**

  Variablen können mit Hilfe des *<set>*- bzw. *<set-multi>*-Kommandos gesetzt bzw. geändert werden. Mit *set-multi* ist es möglich eine Variable zu erzeugen oder zu ändern, die mehrere Werte in Form einer Liste haben kann. Dabei werden die verschiedenen Werte durch Kommata getrennt und stehen dann für die Verarbeitung durch den *<while>*-Befehl zur Verfügung. Falls der Wert eine Referenz auf eine andere Variable darstellt, kann durch das Attribut *defer-evaluation* die Auswertung des Wertes bis zur Benutzung der Variable verschoben werden. Variablen bleiben bis zum Ende des x-frames bestehen, indem sie erzeugt wurden. Sie werden auch an eingebundene x-frames vererbt, können dort allerdings nicht geändert werden.

  ```
  <set var="Variablenname" value="Wert"
      [defer-evaluation="yes-no"]/>

  <set-muti var="Multi-Variablenname" value="Wert(, Wert, ...)"
      [defer-evaluation="yes-no"]/>
  ```

- **<while>**

  Eine Schleife lässt sich über den *<while>*-Befehl realisieren. Das einzige Attribut enthält eine Liste mit allen Multi-Variablen über die iteriert werden soll. Dabei können Variablen auch indirekt referenziert werden. Alle angegebenen Multi-Variablen müssen die selbe Anzahl an Elementen umfassen, da diese parallel durchlaufen werden. Innerhalb der Schleife repräsentiert die Multi-Variable nur noch eine einfache Variable mit dem Wert des aktuellen Durchlaufs. Es ist daher nicht möglich innerhalb einer while-Schleife eine Multi-Variable als solche zu benutzen, wenn über diese bereits iteriert wird. Diese Einschränkung bleibt auch bei adaptierten x-frames innerhalb der Schleife bestehen.

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

- **<select>**

  Der *<select>*-Befehl entspricht am ehesten dem if-then-else Sprachkonstrukt von Programmiersprachen. Das *select*-Element definiert eine Variable, die mit einer Menge von Werten verglichen werden soll. Das *option*-Element umfasst eine Liste mit Werten, diese können auch aus Variablen bestehen, mit denen verglichen wird

und eine Liste, gleicher Länge, mit Vergleichsoperatoren. Sofern keine Vergleichsoperatoren angegeben werden, wird auf Gleichheit getestet. Sobald ein Vergleich innerhalb seines *option*-Elements wahr ist, wird der Inhalt ausgeführt. Dieser kann wiederum Quelltext oder weitere XVCL-Befehle umfassen. Es ist möglich mehrere *option*-Elemente innerhalb eines *select*-Befehls unterzubringen. Es wird immer nur das erste zutreffende Element ausgewertet.

```
<select option="Variablenname">
    (<option value="Wert (| Wert | ...)"
        [comp-operator="Vergleichsoperator
        (, Vergleichsoperator, ...)"]>
        ...
    </option> ...)
    [<otherwise>
        ...
    </otherwise>]
</select>
```

- **<value-of>**

  Der Wert einer Variable kann für die Ausgabe durch den *<value-of>*-Befehl genutzt werden. Wie in Abschnitt 3.3.2 beschrieben wird der Ausdruck ausgewertet und anschließend der Ausgabe hinzugefügt.

  ```
  <value-of expr="Ausdruck" />
  ```

3.3.4 Ein Beispiel

Zum besseren Verständnis folgt ein kleines Beispiel, indem ein MPSoC generiert werden soll. Dazu werden im SPC einige Einstellungen in Form von Variablen festgelegt, die die Generierung beeinflussen.

Die Multi-Variable ProzessorType gibt den zu verwendenden Prozessor an. Da ein Prozessor durchaus mehrere Peripherie-Geräte haben kann, enthält die Multi-Variable lediglich den Namen einer Variablen, in der alle benötigten Peripherie-Komponenten aufgelistet sind. Entsprechend müssen MBlitePeri und MIPSPeri definiert werden. Mit Hilfe der Variable Counter wird jedem Prozessor eine ID zugewiesen. Der Prozessor Type gibt auch den benötigten x-frame an.

```
<x-frame name="SPC" outfile="mpsoc.txt">
    <set-multi var="ProzessorType" value="MBlite,MIPS" />
    <set-multi var="Peripherie" value="MBlitePeri,MIPSPeri" />
    <set-multi var="MBlitePeri" value="CAN" />
    <set-multi var="MIPSPeri" value="UART,CAN" />

    <set var="Counter" value="0" />
MPSoC enthält folgende Prozessoren:
```

```
    <while using-items-in="ProzessorType,Peripherie">
        <adapt x-frame="?@ProzessorType?">
            <insert break="ProzessorID">
Prozesser ID: <value-of expr="?@Counter?"/>
            </insert>
        </adapt>
        <set var="Counter" value="?@Counter? + 1" />
    </while>
</x-frame>
```

Die eindeutige ProzessorID wird mit Hilfe eines Breakpoints in den jeweiligen x-frames der unterschiedlichen Prozessoren hinterlegt. Der Inhalt des Breakpoints wird dabei überschrieben, sofern ein entsprechender *insert*-Befehl vorhanden ist. Schließlich wird noch ermittelt welche Peripherie-Geräte der Prozessor benötigt. Im Falle des MBlites unterstützt er beide Möglichkeiten.

```
<x-frame name="MBlite">
    <break name="ProzessorID">
Prozessor ID: unbekannt
    </break>
Prozessor Type: MicroBlaze
    <while using-items-in="?@Peripherie?">
        <select option="?@Peripherie?">
            <option value="UART">
<value-of expr="    ?@@Peripherie?"/>-Schnittstelle vorhanden
            </option>
            <otherwise>
<value-of expr="    ?@@Peripherie?"/>-Bus vorhanden
            </otherwise>
        </select>
    </while>
</x-frame>
```

Der MIPS Prozessor ist ähnlich aufgebaut und daher ähnelt der x-frame auch dem des MBlites. Allerdings unterstützt dieser Prozessor keinen CAN-Bus, weshalb hier eine entsprechende Meldung in der Ausgabedatei erzeugt wird. Mit dem *<message>*-Befehl ist es auch möglich Ausgaben in der Konsole anzeigen zu lassen und die Verarbeitung bei einer Fehlkonfiguration zu stoppen.

```
<x-frame name="MIPS">
    <break name="ProzessorID">
Prozessor ID: unbekannt
    </break>
Prozessor Type: MIPS
    <while using-items-in="?@Peripherie?">
        <select option="?@Peripherie?">
            <option value="UART">
```

```
<value-of expr="    ?@@Peripherie?"/>-Schnittstelle vorhanden
            </option>
            <otherwise>
    MIPS unterstützt kein <value-of expr="?@@Peripherie?"/>-Bus
            </otherwise>
        </select>
    </while>
</x-frame>
```

Mit diesen Einstellungen werden 2 Prozessoren erzeugt die eine bestimmte Auswahl an Peripherie-Geräten enthält. Die Ausgabe sieht wie folgt aus:

```
MPSoC enthält folgende Prozessoren:

Prozesser ID: 0
Prozessor Type: MicroBlaze
    CAN-Bus vorhanden
----------
Prozesser ID: 1
Prozessor Type: MIPS
    UART-Schnittstelle vorhanden
    MIPS unterstützt kein CAN-Bus
----------
```

4 Related Work

Die Entwicklung von MPSoCs ist ein aktuelles Thema der Forschung. Die immer komplexer werdenden Systeme, mit steigenden Anforderungen und immer größerem Funktionsumfang, bedienen sich der MPSoCs, um Energie und Kosten zu sparen und gleichzeitig die Effektivität zu steigern. Ebenso wird es immer schwieriger solche Systeme zu entwerfen. Neue Methoden, um bestimmte Aspekte der Entwicklung zu vereinfachen, wurden beispielsweise in [27, 28] präsentiert. Auch eine Vielzahl an unterschiedlichen Transaktionsspeichern sind mittlerweile veröffentlicht worden [12, 14, 15, 16, 18, 29]. Gerade die vielen Möglichkeiten Transaktionsspeicher zu implementieren eignen sich für die Betrachtung einer Speicher Familie. In diesem Kapitel werden kurz zwei Ansätze für den Entwurf von MPSoCs dargestellt und anschließend einige Arbeiten zu Transaktionsspeichern betrachtet.

In [27] wird eine Multiprozessor-Architektur vorgestellt, die über ein hierarchisches Bus-System verfügt. Alle ARM-Prozessoren besitzen einen lokalen Speicher, der über den ersten Bus-Layer angebunden ist. Über eine Bus-Bridge ist der zweite Bus-Layer angeschlossen, der alle Prozessoren verbindet. Die eingesetzten Busse beruhen auf der Advanced Microcontroller Bus Architecture [30]. Ein Arbiter steuert dabei den Zugriff auf den geteilten zweiten Bus. Für die Kommunikation zwischen den Prozessoren, befindet sich ein geteilter Speicher auf dem zweiten Bus-Layer. Mit Hilfe eines Semaphore-Controllers wird exklusiver Zugriff auf gemeinsame Daten realisiert. Die Architektur ist in Abbildung 4.1 dargestellt.

Ein Network-on-Chip (NoC) Ansatz wird in [28] verfolgt. Für die Systemkomponenten werden die von Xilinx angebotenen Intellectual Property Blocks verwendet. Das Embedded Development Kit (EDK) von Xilinx wird zum Konfigurieren und Generieren des Systems benutzt. Für die Verbindungen zwischen den Komponenten wurde ein NoC entwickelt, dass die Daten in Pakete aufteilt und über einen Router ans Ziel transportiert. Da nach [28] die Integration eigener Komponenten in das EDK schwierig ist, wird zur Unterstützung ein tlc-Skript verwendet. Anhand der Informationen über das System wird die Kommunikationsstruktur automatisch erzeugt. Ein textbasiertes Tool unterstützt die Konfigurierung des Systems.

Diese Architekturen und Frameworks dienen dazu den Prozess der MPSoC-Entwicklung zu vereinfachen. Dabei wird auf bereits vorhandene Hardwarekomponenten zurückgegriffen und eine Möglichkeit geschaffen diese miteinander zu verbinden. Dies trifft im Wesentlichen auch auf das LavA-Projekt zu, jedoch steht im Rahmen dieser Arbeit das Entwerfen von konfigurierbaren Hardwarekomponenten, in Form einer Familie von gemeinsamen Speichern, im Vordergrund.

Teil dieser Speicher Familie wird ein Transaktionsspeicher sein. In [14] wurde der *Unbounded Transactional Memory* (UTM) vorgestellt. Es handelt sich dabei um ein System

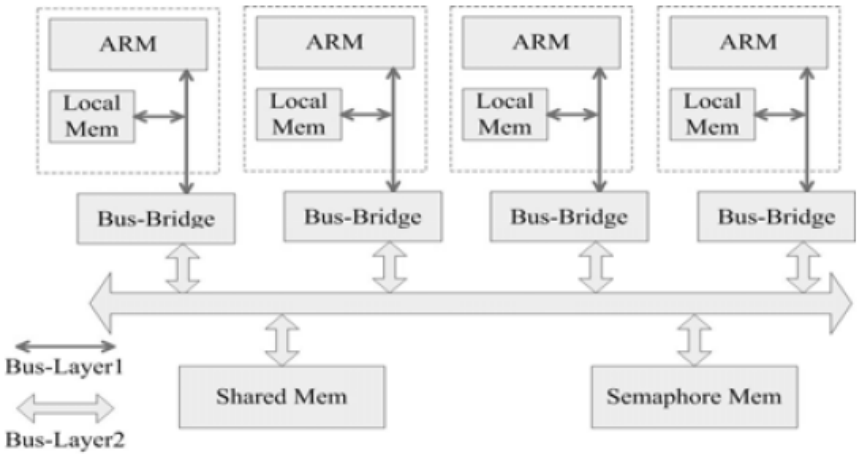

Abbildung 4.1: Architektur des MPSoCs [27]

mit eager Versionierung und pessimistischer Konflikterkennung. UTM unterstützt dabei Paging, Threads, verschachtelte Transaktionen und die Migration von Prozessen zwischen den Prozessoren. Dazu sind einige Änderungen an der CPU, als auch an der Speicherhierarchie nötig. Insbesondere wird der Umgang mit virtuellen Adressräumen ausgenutzt. Zusätzlich sind beliebig große Transaktionen möglich, sofern der virtuelle Speicher groß genug ist die alten als auch neuen Daten zu speichern. Grundsätzlich ist UTM nicht auf Caches angewiesen, allerdings ist eine Leistungssteigerung durch geschickte Ausnutzung der Caches möglich.

Neben zwei neuen Instruktionen XBEGIN und XEND, um Transaktionen nutzen zu können, muss die CPU in der Lage sein den Prozessorzustand mit der Ausführung der XBEGIN Instruktion zu sichern bzw. im Falle eines Revert wiederherzustellen. In dem Papier [14] wird ein Prozessor zugrunde gelegt, der über die Möglichkeit der Registerumbenennung verfügt. Zu Beginn einer Transaktion müssen die aktuell benutzten physikalischen Register vor der automatischen Wiederverwendung geschützt werden. Erst nach dem Commit der Transaktion dürfen diese Register überschrieben werden. Für einen Prozesswechsel oder die Migration des Prozesses, müssen die gesicherten physikalischen Register zusätzlich zu den Architekturregistern im Prozessorzustand abgelegt werden.

Damit Transaktionen nicht an einen Prozessor bzw. Cache gebunden sind, werden alle nötigen Informationen des Transaktionsverlaufs in der *xstate* Datenstruktur im Speicher gesichert. Die xstate-Struktur ist in Abbildung 4.2 illustriert. Der Speicher selbst wird in Blöcke aufgeteilt. Für jede laufende Transaktion wird ein Transaktions-Log geführt. Dort wird der aktuelle Status, das sogenannte *commit-record*, der Transaktion vermerkt (*pending*, *committed* oder *aborted*). Weiterhin gibt es für jeden Block, der innerhalb einer Transaktion gelesen oder geschrieben wird, einen Eintrag im Transaktions-Log, der den alten Wert, einen Pointer zum Speicher-Block, sowie einen Pointer zu anderen Einträgen anderer Transaktionen des selben Blocks enthält. Die Verweise zwischen den Einträgen der Transaktions-Logs bilden eine verkettete Liste. Um schnell überprüfen zu können, ob bereits andere Transaktionen einen Block gelesen oder geschrieben haben, existiert für

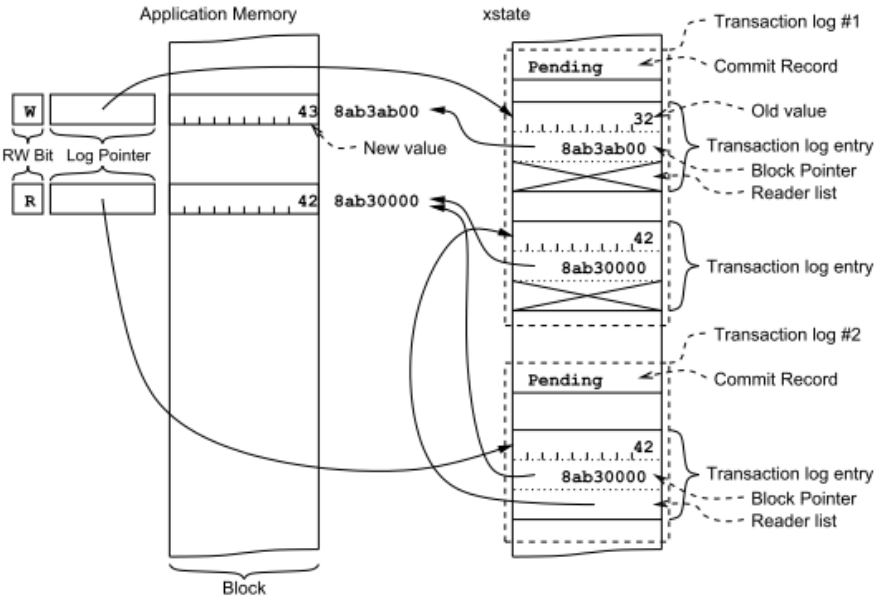

Abbildung 4.2: UTM xstate Datenstruktur [14]

jeden Block ein *Log Pointer* der auf den Eintrag eines Transaktions-Logs verweist, der diesen Block benutzt hat. Außerdem wird dort ein *RW Bit* hinterlegt, so dass festgestellt werden kann, ob dieser Block bereits gelesen oder beschrieben wurde.

Eine einfach Policy regelt die auftretenden Konflikte. Dazu bekommt jede Transaktion zu Beginn einen Zeitstempel. Je älter eine Transaktion ist, desto höher ist ihre Priorität. Die am Konflikt beteiligten Transaktionen mit der geringsten Priorität werden abgebrochen, bis der Konflikt gelöst ist.

Ein weiteres Transaktionsspeichersystem stellt LogTM [16] dar, dass eine Speicherhierarchie mit privatem L1 Cache und gemeinsamen L2 Cache voraussetzt. Als Basis für die Cache-Kohärenz wird ein verzeichnisbasiertes MOESI Protokoll [31] verwendet. Auch LogTM greift auf eine eager Versionierung zurück und platziert die alten Werte in einer extra Speicherregion, die im Cache gehalten werden kann. Jeder Thread erhält sein eigenes Transaktions-Log, dabei werden innerhalb von Transaktionen automatisch die alten Werte in das Log übertragen. Im Falle eines Transaktionsabbruchs kommt ein Software-Handler zum Einsatz, der den ursprünglichen Speicherzustand wiederherstellt.

Konflikte werden direkt beim Speicherzugriff erkannt, womit es sich hier um eine pessimistische Konflikterkennung handelt. In der sogenannten *Sharer List* wird im Verzeichnis des L2 Caches zu jeder Cache-Zeile vermerkt, welche L1 Caches eine gültige Kopie besitzen. Sobald ein schreibender Zugriff einer Transaktion erfolgt, muss für diese Cache-Zeile exklusiver Zugriff eingefordert werden. Das Verzeichnis leitet diese Anfrage an die in der Share List enthaltenen Prozessoren, die überprüfen können, ob die Cache-Zeile bereits Bestandteil einer Transaktion ist. Falls ja handelt es sich um ein Konflikt und der schreibende Prozessor kümmert sich um die Konfliktbehebung. Umgekehrt werden lesende Zugriffe auf bereits geschriebene Cache-Zeilen an den Cache mit dem aktuellen

Wert weitergeleitet und ermöglichen so das Erkennen von Konflikten. Ebenfalls muss sich der Angreifer, also der lesende Prozessor, um die Auflösung des Konflikts kümmern. Die Policy ist dabei teils in Hardware implementiert, kann aber im Falle von drohenden Deadlocks oder Livelocks einen Software-Handler starten.

Es sind einige Transaktionsspeicher publiziert [12, 14, 15, 16, 18] worden, mit sehr unterschiedlichen Lösungsansätzen. Allen gemein ist allerdings, dass sie nur in geringem Maße konfiguriert werden können. Insbesondere lässt sich keine speziell auf die Anwendung zugeschnittene Hardware Instanzen generieren. Aus den verschiedenen Transaktionsspeichern können aber Informationen gewonnen werden, um eine anpassbare Plattform für Transaktionsspeicher zu erstellen. Damit kann der Familie von gemeinsamen Speichern eine Reihe von Variationspunkten hinzugefügt werden.

5 Eigenschaften der Speicher Familie

In einer Familie von gemeinsamen Speichern können sehr unterschiedliche Speichermodelle untergebracht werden. Es bietet sich daher an, im Rahmen dieser Arbeit, diese auf drei Modelle einzuschränken. Ein sehr einfaches Speichermodell stellt der FIFO-Speicher (First In First Out) dar, der Daten zwischen Prozessoren puffern kann. Für die Verwendung klassischer Programme, die mit Hilfe von Locks, Semaphore oder Monitoren eine Menge von Threads synchronisieren wollen, benötigen die im LavA-Projekt verwendeten Prozessoren einen Speicher-Controller, der diese Synchronisationsmechanismen ermöglicht. Als letztes Speichermodell eignet sich der Transaktionsspeicher, da dieser aufgrund seiner unterschiedlichen Ausprägungen eine Menge an Konfigurationsmöglichkeiten bietet.

Dieses Kapitel beschäftigt sich mit dem Ermitteln der Unterschiede und Gemeinsamkeiten der verschiedenen Speichermodelle. Zu diesem Zweck wird ein Merkmalmodell erstellt, dass die Konfigurationsmöglichkeiten widerspiegelt. Zunächst werden in Unterkapitel 5.1 die Gemeinsamkeiten der Speichermodelle herausgefiltert. Abschnitt 5.2 geht auf die weiteren Variationsmöglichkeiten des FIFO-Speichers ein. Der Speicher-Controller für read-modify-write Speicheroperationen wird in Abschnitt 5.3 näher besprochen und schließlich werden in Unterkapitel 5.4 die Konfigurationsmöglichkeiten des Transaktionsspeichers betrachtet.

5.1 Gemeinsamkeiten der Speichermodelle

Die Clients (im normal Fall Prozessoren) und der Hintergrundspeicher müssen zunächst mal mit dem Speicher-Controller verbunden werden. Dazu gibt es eine Reihe unterschiedlicher Standards und individuell entwickelter Schnittstellen, um die unterschiedlichen Komponenten miteinander zu verbinden. Daher bildet die **Schnittstelle** ein notwendiges Merkmal. Zu den Standard on-Chip Bussystemen gehört unter anderem das von IBM veröffentlichte CoreConnect [32], die von der ARM Corp entwickelte Advanced Microcontroller Bus Architecture [30] und der Wishbone Bus ursprünglich entwickelt von der Silicore Corp. und weiterentwicklet durch OpenCores [33].

Die **Busbreite** spielt nicht nur für die Schnittstelle eine Rolle. Auch mögliche Register oder Puffer in den Speicher-Controllern können durch Anpassen an die benötigten Bitbreiten Ressourcen sparen. Dabei spielt es keine Rolle, ob es sich um den Daten- oder Adressbus handelt. In welchem Umfang die Busbreite konfiguriert werden kann hängt stark von den betrachteten Komponenten ab, weshalb eine genauere Spezifizierung der Variationsmöglichkeiten hier nicht betrachtet wird. Sofern der Datenbus breiter als 8Bit ist, kann auch ein nicht zwingender **byteweiser Zugriff** beim Ressourcen sparen helfen.

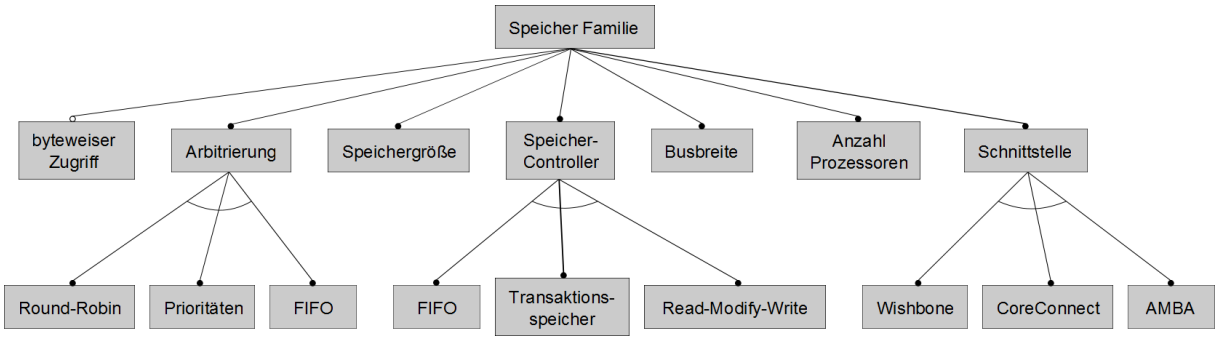

Abbildung 5.1: Merkmaldiagramm der gemeinsamen Merkmale

Die Möglichkeit jedes Byte der Daten einzeln schreiben oder lesen zu können benötigt mehr Logik für die Verwaltung.

Sobald mehrere Prozessoren sich einen gemeinsamen Speicher teilen und nicht alle gleichzeitig diesen benutzen können, ist es notwendig den Zugriff zu steuern. Die **Arbitrierung** eines von mehreren Prozessoren genutzten Busses kann wiederum auf sehr unterschiedliche Art und Weise geschehen. Für echtzeitfähige Systeme wäre sicherlich ein Zeitmultiplexverfahren von Vorteil, um das Verhalten des Gesamtsystems besser voraussagen zu können. Das eingesetzte Verfahren für die Arbitrierung des Busses kann auch als Scheduling aufgefasst werden. Daher lassen sich i. d. R. Scheduling-Verfahren, wie beispielsweise Round-Robin, Prioritätenscheduling oder First-Come First-Served, zum Arbitrieren einsetzen.

Eine weitere Größe die beim Entwerfen der Speicher Familie eine Rolle spielt, ist die **Anzahl der Prozessoren** die angeschlossen werden sollen. Gerade bei komplexeren Speicher-Controllern, wie im Falle des Transaktionsspeichers, ist es wichtig nur Ressourcen zu verbrauchen, die von der Anzahl der Prozessoren benötigt wird. In welchem Umfang die Anzahl der Prozessoren beeinflussbar ist hängt vom Speicher-Controller ab und kann hier nicht näher spezifiziert werden.

Die tatsächliche Größe des angebundenen Speichers kann ebenfalls zur Optimierung ausgenutzt werden. Dabei kann über die nötige Logik für die Verwaltung des Speichers Ressourcen gespart werden. Auch eine direkte Konfigurierung eines integrierten Hintergrundspeichers wäre denkbar. In Abbildung 5.1 sind diese Merkmale nochmal grafisch aufbereitet. Zusätzlich sind die unterschiedlichen Speichermodelle dargestellt, die in den folgenden Abschnitten beschrieben werden.

5.2 FIFO-Speicher

Die Grundausstattung eines einfachen FIFO-Speichers umfasst einen Puffer, der die Daten der schreibenden Prozessoren zwischenspeichert, bis einer der lesenden Prozessoren diese benötigt. Dabei wird das geschriebene Datum eines Speicherzugriffs direkt den lesenden Prozessoren zur Verfügung gestellt. Unter der Voraussetzung, dass die Schnittstelle zu den Prozessoren eine variable Dauer eines Speicherzugriffs erlaubt, kann im

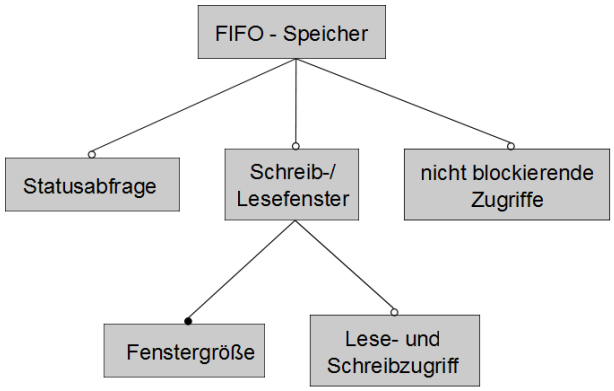

Abbildung 5.2: Zusätzliche Merkmale des FIFO-Speichers

Falle eines leeren oder vollen Puffers der entsprechende Prozessor im Falle eines Speicherzugriffs blockiert werden. Im einfachsten Fall ist es auch nicht möglich, den Status des FIFO-Speichers abzufragen.

Den **Status abfragen** zu können erfordert zusätzliche Logik und kann damit als optional betrachtet werden. Allerdings können mit Informationen wie der Füllstand des Speichers mögliche Scheduling-Entscheidungen getroffen werden. Ein lesender Prozess der keine bis nur sehr wenige Daten lesen kann, sollte ggf. einen anderen Prozess Vorrang geben, der möglicherweise sogar einen vollen Puffer abarbeiten muss. Sofern **nicht blockierende Aufrufe** benötigt werden, ist zu dem die Information über den Ausgang des letzten Zugriffes nötig.

Eine ähnliche Semantik wie sie bei Transaktionen eingesetzt wird, kann auch beim FIFO-Speicher von Vorteil sein. Ein **Schreib-/Lesefenster** könnte dazu genutzt werden Datenobjekte, die mehr Bit benötigen als der Datenbus umfasst, erst vollständig in den Speicher zu schreiben, bevor der Lesende Prozessor auf sie zugreifen kann. Darüber hinaus kann ein solches Zugriffsfenster zum direkten Erzeugen der Daten ausgenutzt werden, ohne den Programmspeicher damit zu belasten. Umgekehrt wäre ein lesender Prozessor in der Lage die Daten erst freizugeben, sobald die Daten fertig bearbeitet worden sind. So könnte das Erlauben von lesenden Zugriffen auf das aktuelle Fenster des Schreibers bzw. schreibender Zugriff auf das Fenster des Lesers unnötiges Umkopieren von Daten vermeiden. In Abbildung 5.2 sind die einzelnen Aspekte nochmal dargestellt.

5.3 Speicher-Controller für read-modify-write Speicheroperationen

In einem Multiprozessorsystem, das über einen gemeinsam genutzten Speicher verfügt, ist es allein mit Softwaremechanismen nicht möglich kritische Programmbereiche mit Hilfe von Locks oder ähnlichem zu synchronisieren. Einfache lesende und schreibende Speicheroperationen sind nicht in der Lage Zugriffe von mehrere Prozessoren in einem

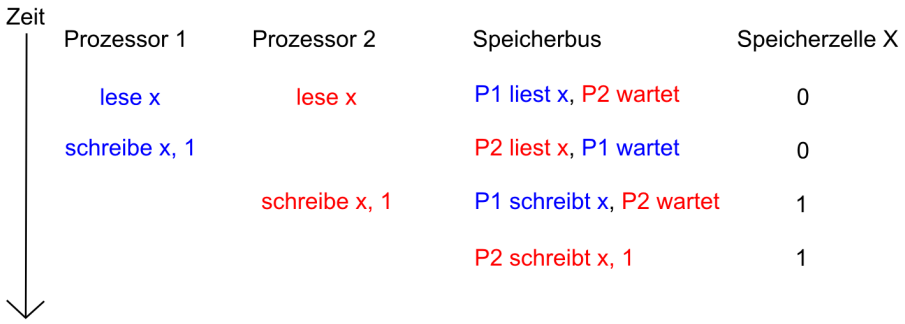

Abbildung 5.3: Problem beim Anfordern einer Lock-Variable ohne atomare read-modify-write Speicheroperation. Das Scheduling des gemeinsamen Speicherbusses sorgt dafür, dass beide Prozessoren den selben Wert lesen. Beide Prozessoren werden den kritischen Abschnitt betreten.

konsistenten Zustand zu halten, wie in Abbildung 5.3 demonstriert. Zum Betreten eines kritischen Abschnitts müssen beispielsweise zwei Dinge sichergestellt werden: Zunächst darf sich kein anderer Prozess innerhalb des kritischen Abschnitts befinden, dazu ist es nötig dies durch Lesen einer Lock-Variable zu prüfen. Erst wenn der kritische Bereich frei ist (signalisiert durch den Wert 0), können durch setzen der Lock-Variable auf den Wert 1 alle Anderen daran gehindert werden diesen Bereich zu betreten. Da der lesende und anschließend schreibende Speicherzugriff nicht atomar vollzogen werden kann, ist es möglich, dass ein zweiter Prozess genau zwischen dem lesenden und schreibenden Zugriff versucht den kritischen Bereich für sich zu beanspruchen.

Damit Prozesse auf unterschiedlichen Prozessoren synchronisiert werden können wurden in der Vergangenheit einige spezielle Speicheroperationen entwickelt. Kruskal et al. [34] haben die meisten dieser Operation als *read-modify-write* Operationen klassifiziert. Eine solche Operation muss atomar eine Speicherzelle lesen, den Wert modifizieren und anschließend zurück schreiben. Der Prozess ist dabei i. d. R. in der Lage den alten und überschriebenen Wert auszuwerten. Einige der Methoden sind in Abbildung 5.4 als Variationsmöglichkeit dargestellt. *Test and Set* stellt den alten Wert der Speicherzelle zur Verfügung und überschreibt ihn anschließend. Mit Hilfe von *Fetch and Add* wird der alte Wert gelesen und anschließend inkrementiert zurückgeschrieben. Die *Compare and Swap* Variante vergleicht zunächst den Wert der gewünschten Speicherzelle mit einem beliebigen vorgegebenen Wert und schreibt, im Falle der Gleichheit dieser Werte, einen neuen Wert in die betrachtete Speicherzelle. Um zu überprüfen, ob der Austausch erfolgreich war, muss lediglich der von der Compare and Swap Operation bereitgestellte alte Wert ausgewertet werden.

5.4 Transaktionsspeicher

Nach dem in Kapitel 2.4 bereits die Grundlagen eines Transaktionsspeichers besprochen wurden, lassen sich daraus direkt einige Merkmale ableiten. Dabei gibt es eine Reihe

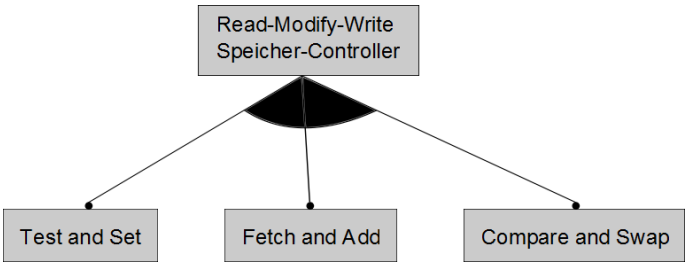

Abbildung 5.4: Zusätzliche Merkmale des read-modify-write Speicher-Controllers

von notwendigen Komponenten, die die wesentlichen Eigenschaften widerspiegeln. Mit Hilfe einiger weitere Konfigurationsmöglichkeiten kann das Transaktionsspeichersystem an die Anforderungen der Anwendung angepasst werden.

5.4.1 Grundbestandteile des Transaktionsspeichers

Ein wichtiger Bestandteil bildet die **Konflikterkennung**, die entweder direkt beim Speicherzugriff Konflikte erkennt (pessimistisch) oder erst beim Commit auf Konflikte überprüft (optimistisch). Mit lazy oder eager hingegen wird die Strategie der **Versionierung** bezeichnet. Dabei speichert die eager Versionierung das neue Datum im Hintergrundspeicher und sichert das alte Datum in einem Zwischenspeicher. Entsprechend werden bei der lazy Versionierung die neuen Daten im Zwischenspeicher gehalten und erst im Rahmen des Commits in den Speicher geschrieben. Bei der Betrachtung von Transaktionsspeichern, die auf Hardwareunterstützung setzen, kann von den vier möglichen Kombinationen aus Versionierung und Konflikterkennung eine ausgeschlossen werden: Eager Optimistic. Da bei einer eager Versionierung die neuen Werte direkt im Hintergrundspeicher gespeichert werden und damit in jedem Fall einen Speicherzugriff erzeugen, macht es keinen Sinn die Konflikterkennung bis zum Commit hinauszuzögern. Konflikte könnten direkt beim Speicherzugriff erkannt werden, so dass zum Scheitern verurteilte Transaktionen, aufgrund der optimistischen Konflikterkennung, nicht gestoppt werden und daher unnötig Rechenzeit verschwenden.

Auch bereits erwähnt wurde die Notwendigkeit Konflikte mit Hilfe einer **Policy** zu lösen. Abhängig von der Anwendung, der Konflikterkennung und der Versionierung ergeben sich Anforderungen an die Policy. Eine ungünstige Zusammenstellung kann zu großen Leistungseinbußen führen und daher sollte eine Menge möglicher Policies zur Verfügung stehen. Unter Umständen ist es nicht ratsam die Policy komplett in Hardware umzusetzen, um Problemen vorbeugen zu können. Eine flexible und anpassbare Policy kann durch Softwareunterstützung ermöglicht werden. Damit steigt die Auswahl von ausschließlich in Hardware implementierten Policies, für hohe Geschwindigkeit, um Softwarelösungen, die für hohe Flexibilität sorgen. Dabei muss berücksichtigt werden, dass die Hardware entsprechende Informationen über den Status und den Verlauf der Transaktionen, der in Software umgesetzten Policy, bereitstellen muss.

5.4.2 Erweiterungsmöglichkeiten des Transaktionsspeichers

Die **Speicheraufteilung** bietet weitere Optimierungsmöglichkeiten für den Transaktionsspeicher. Anstatt vollen Zugriff für alle Prozessoren auf den gesamten Hintergrundspeicher zu erlauben, könnten, je nach Kommunikationsstruktur der Anwendung, andere Aufteilungen beim Ressourcensparen helfen. Eine Blackboard ähnliche Struktur kann dadurch erzielt werden, dass alle Prozessoren nur für einen Teil des gesamten Speichers Schreibrechte erhalten. Die Bereiche der Prozessoren dürfen sich dabei nicht überschneiden. Lesende Zugriffe dagegen sind überall erlaubt. Durch geschickten Einsatz von Konflikterkennung, Versionierung und Policy könnten Ressourcen gespart werden. Eine weitere Einschränkung mit nur einem schreibenden Prozessor und potenziell vielen lesenden Prozessoren, führt zu einer Multicast Variante. Für die Prozessoren die ausschließlich lesend auf den Speicher zugreifen, entfällt damit das Write-Set und die Notwendigkeit für einen Zwischenspeicher. Weitere Einsparungen an Hardwareressourcen können erst nach Betrachtung der Policy, Konflikterkennung und Versionierung identifiziert werden.

Grundsätzlich sind Zugriffe auf den gemeinsamen Speicher außerhalb von Transaktionen durchaus wünschenswert. Insbesondere wenn der gemeinsame Speicher der Einzige ist, der den Prozessoren zur Verfügung steht. Aus Sicht des Programmierers wird zwischen strenger und schwacher Atomizität unterschieden, wobei die strenge Atomizität ein einfacher zu benutzendes Speichermodell darstellt. Bei der strengen Atomizität werden Speicherzugriffe außerhalb von Transaktionen auch als Transaktionen gewertet, die nur einen Speicherzugriff umfassen und automatisch ihre Änderungen über ein Commit sichtbar machen. Um dies gewährleisten zu können, muss sowohl die Konflikterkennung als auch die Policy mit solchen Speicherzugriffen umgehen können. Für Speicherzugriffe außerhalb von Transaktionen existiert kein Sicherungspunkt der im Falle eines Reverts wiederhergestellt werden könnte. Die im LavA-Projekt eingesetzte Systemarchitektur stellt i. d. R. für jeden Prozessor einen lokalen bzw. privaten Programmspeicher zur Verfügung. Daher kann der gemeinsam genutzte Speicher auch ausschließlich für die Nutzung von gemeinsame Datenstrukturen eingesetzt werden. So kann der Transaktionsspeicher, um die Fähigkeit auch **Speicherzugriffe außerhalb von Transaktionen** unterstützen zu müssen, erleichtert werden.

In [19] hat McDonald einige zusätzliche Eigenschaften von Transaktionsspeichern beschrieben, um die Akzeptanz der Programmierer gegenüber dem neuen Speichermodell zu steigern. Dazu zählt unter anderem die Möglichkeit Transaktionen ineinander zu verschachteln. Eine innere Transaktion soll unabhängig von den äußeren Transaktionen abgebrochen werden können. Dabei kann die innere Transaktion sämtliche getätigte Änderungen der äußeren Transaktion lesen. Zudem lassen sich im Wesentlichen zwei Arten von verschachtelten Transaktionen unterscheiden: offene und geschlossene. Geschlossen verschachtelte Transaktionen können zwar unabhängig von der umgebenen Transaktion abgebrochen werden, aktualisieren aber den Speicher im Falle eines Commits nicht direkt, so dass andere Transaktionen die Änderungen erst mit dem Commit der umliegenden Transaktion einsehen können. Die äußere Transaktion hat nach dem Commit der inneren Transaktion Zugriff auf die Änderungen. Die Änderungen einer offen verschachtelten Transaktion werden mit dem Commit für alle anderen Transaktionen wirksam.

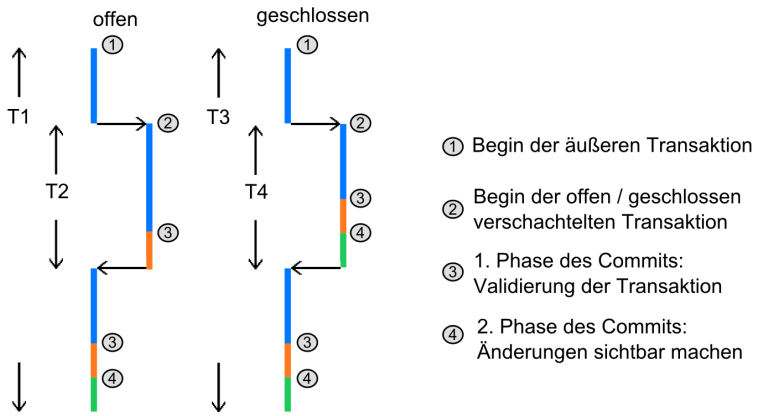

Abbildung 5.5: Ineinander verschachtelte Transaktionen

Sobald allerdings, nach dem Commit der offen verschachtelten Transaktion, die umliegende Transaktion abbricht, müssen Mechanismen vorhanden sein, um auch die bereits abgeschlossene Transaktion rückgängig zu machen. Abbildung 5.5 zeigt den Ablauf verschachtelter Transaktionen. Die genaue Semantik von verschachtelten Transaktionen ist dabei in der Literatur zum Teil unterschiedlich. Insbesondere ergeben sich einige Schwierigkeiten im Umgang mit offenen verschachtelten Transaktionen die von Agarwal et al. in [35] beschrieben worden sind.

Weiterhin können Möglichkeiten zur Virtualisierung der Zeit oder des Platzes nötig sein. So kann i. d. R. nicht ausgeschlossen werden, dass eine Transaktion mehr Platz für die Sicherung der alten Werte benötigt, als der Zwischenspeicher an Platz zur Verfügung stellt. Damit eine solche Transaktion nicht abgebrochen werden muss, ist es erforderlich die Beschränkung des Zwischenspeichers zu umgehen. Dies kann z. B. dadurch geschehen, dass ein Teil des Hintergrundspeichers als Zwischenspeicher genutzt wird. Beim Ausführen mehrerer Prozesse oder Threads auf einem Prozessor ist es darüber hinaus denkbar, dass auch mehrere Transaktionen über einen Anschluss des Transaktionsspeichers verwaltet werden müssen. Dies kann durch entsprechendes vervielfältigen der Hardwareressourcen oder über Datenstrukturen im Hintergrundspeicher erreicht werden.

Der Einsatz von **Signaturen**, die eine Menge von Speicheradressen repräsentieren, ermöglicht das Implementieren schneller Konflikterkennungsmethoden. Eine auf Bloom-Filter [36] basierende Signatur ist in der Lage sich eine Menge von Adressen, mit Hilfe einer Hashfunktion, zu merken. Nach dem Berechnen des Hashwertes einer Adresse kann dieser mit einfachen Mitteln in die Signatur aufgenommen werden. Um überprüfen zu können, ob eine Signatur eine bestimmte Adresse enthält, muss lediglich der Hashwert der Adresse mit der Signatur verglichen werden.

Nicht nur bei der Policy kann der Einsatz von Software mehr Flexibilität zur Laufzeit bewirken. Es gibt weitere Anwendungen für **Software Handler**. Im Rahmen eines Commits könnte ein *Commit Handler* dazu beitragen automatisch abschließende Aufgaben auszuführen, nachdem die Transaktion erfolgreich validiert worden ist und bevor die Änderungen der Transaktion sichtbar gemacht werden. Beispielsweise könnten Betriebs-

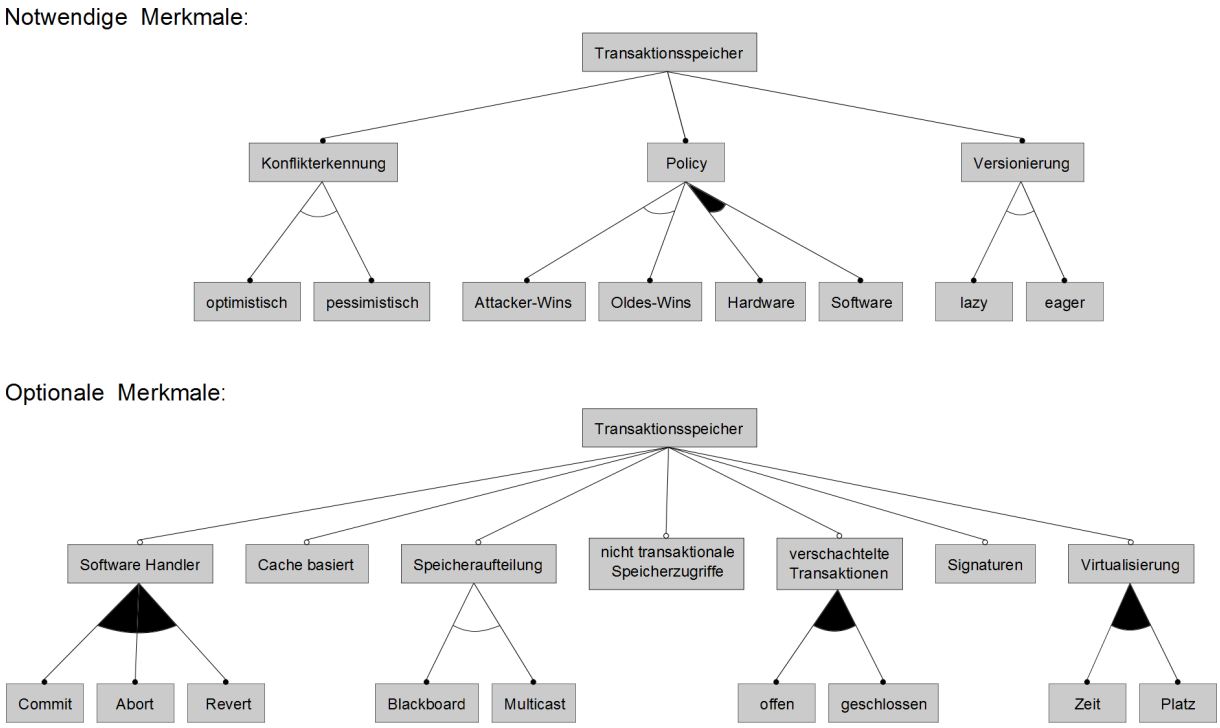

Abbildung 5.6: Zusätzliche Merkmale des Transaktionsspeichers

systemaufrufe mit permanenten Seiteneffekten durch einen Commit Handler erledigt werden. Ebenso sollte es möglich sein beim Auftreten von Problemen die Transaktion noch abzubrechen. *Violation Handler* und *Abort Handler* werden beim Abbruch einer Transaktion aktiv, bevor der ursprüngliche Speicherzustand wiederhergestellt wird und könnten den privaten Speicher aufräumen oder alternative Ausführungspfade ermöglichen, anstatt die Transaktion erneut zu starten. Der Violation Handler wird dabei im Falle eines Konflikts und der Abort Handler durch den expliziten Wunsch die Transaktion abzubrechen ausgeführt. Dabei sollten alle Software Handler in der Lage sein Informationen über die Transaktion abrufen zu können. Da die Änderungen erst mit einer Verzögerung sichtbar gemacht werden bzw. der ursprüngliche Speicherzustand später wiederhergestellt wird, können neue Konflikte auftreten sofern der weitere Zugriff nicht unterbunden wird.

Eine weitere Konfigurationsmöglichkeit ist bei den meisten in Kapitel 4 referenzierten Transaktionsspeicher umgesetzt. Es handelt sich dabei um **Cache basierte** Methoden. Es wird ausgenutzt, dass bereits Kopien der Daten des Hauptspeichers im Cache vorhanden sind. Mit Hilfe des eingesetzten Cache-Kohärenz-Protokolls können die Aktivitäten der am Hintergrundspeicher angebundenen Teilnehmer beobachtet werden. Durch Modifikationen des Protokolls können damit Transaktionen ermöglicht werden. Dabei dient der Cache als Zwischenspeicher und das Cache-Kohärenz-Protokoll der Konflikterkennung. Abschließend sind in Abbildung 5.6 die Merkmale des Transaktionsspeichers dargestellt.

6 Entwurf und Implementierung der Speicher Familie

Nachdem in Kapitel 5 auf die allgemeinen Anforderungen und Eigenschaften einer Familie von gemeinsamen Speichern eingegangen worden ist, wird im Folgenden der Entwurf vorgestellt. Die Komponenten der Speicher Familie werden dabei durch die Hardwarebeschreibungssprache VHDL beschrieben. Im Rahmen dieser Diplomarbeit ist es nicht möglich alle vorgestellten Konfigurationsmöglichkeiten umzusetzen. Die folgenden Abschnitte beschreiben die implementierten Merkmale.

Unterkapitel 6.1 beschreibt dabei die für alle Instanzen der Speicher Familie zugrunde liegenden Entwurfsentscheidungen. Ausgehend von einer Blackbox, die die Speicher Familie aus Sicht der Prozessoren darstellt, wird der Aufbau des FIFO-Speichers in Abschnitt 6.2 vorgestellt. Unterkapitel 6.3 erläutert die Struktur des Speicher-Controllers für read-modify-write Operationen. Abschließend wird in diesem Kapitel der Aufbau und die Funktionsweise des Transaktionsspeichers in Unterkapitel 6.4 betrachtet.

6.1 Die Speicher Familie als Blackbox

Die Speicher Familie kann als Blackbox angesehen werden, die über eine Standardschnittstelle mit den anderen Komponenten des MPSoCs kommuniziert. Wie in Abbildung 6.1 dargestellt verfügt die Speicher Familie für jeden Prozessor über einen eigenen Anschluss, so dass die Arbitrierung innerhalb der Blackbox geschehen kann. Neben Daten- und Adressbus sind weitere Signale zum Steuern der Kommunikation nötig. Das dabei verwendete Protokoll und die zur Umsetzung nötigen Steuersignale hängen von dem eingesetzten on-Chip Bus Standard ab. Die Schnittstelle zum gemeinsam genutzten Speicher ist optional, da eine Integration des Speichers in die Speicher Familie sinnvoll sein kann, um die im FPGA vorhandenen schnellen Block-RAMs optimal auszunutzen.

6.1.1 On-Chip Bus: Wishbone

Zunächst lässt sich feststellen, dass die Peripheriekomponenten eines Prozessors in einem MPSoC des LavA-Projekts ausschließlich über den Wishbone Standard miteinander verbunden werden. Damit ist es zunächst unwichtig andere on-Chip Bus Standards zu unterstützen. Da der im LavA-Projekt eingesetzte Wishbone Bus nur über die minimale Auswahl an Steuersignalen verfügt, die im Standard festgelegt sind, ist es ggf. nötig weitere Signale, wie beispielsweise Interrupt-Leitungen für die Bearbeitung von Ausnahmen und Fehlern durch den Prozessor, bereit zu stellen.

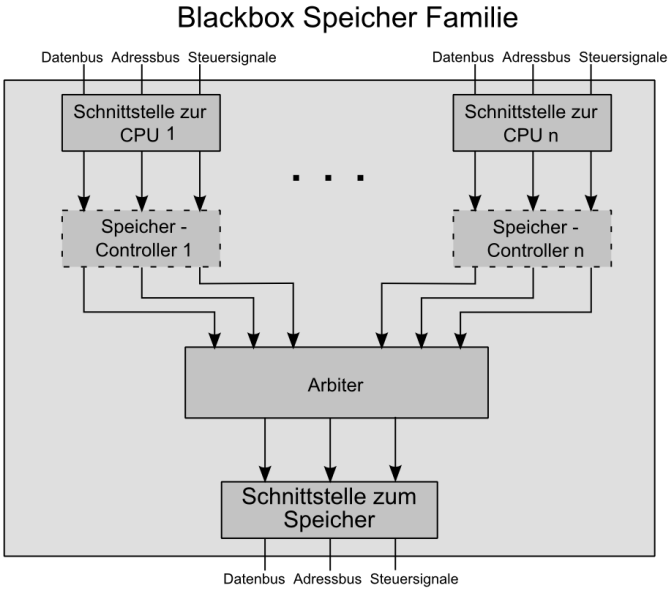

Abbildung 6.1: Speicher Familie als Blackbox

Abbildung 6.2 zeigt auf der oberen Hälfte die im LavA-Projekt umgesetzten Signale des Wishbone Standards und auf der unteren Hälfte den Ablauf eines Lesezykluses. Innerhalb eines FPGAs ist es i. d. R. nicht erwünscht Leitungen bidirektional zu verwenden und daher existieren zwei Datenbusse, um in beide Richtungen Daten austauschen zu können. Es handelt sich hier um eine Bustopologie, bei der ein Master und potenziell viele Slave-Geräte vorhanden sind. Dabei dürfen lediglich Master-Geräte Verbindungen initiieren. Anhand der Adresse kann ermittelt werden, welcher Slave als Kommunikationspartner gewünscht ist. Im LavA-Projekt sind alle Geräte über einen Datenbus an den Prozessor angebunden und sind daher in den normalen Adressraum eingebettet (memory mapped IO).

Damit der Master einen Buszyklus initiieren kann, muss er zunächst die benötigte Adresse auf den Adressbus legen. Im Falle eines Schreibzykluses muss auch auf dem Datenbus der gewünschte Wert bereitgestellt werden. Außerdem muss das WE-Signal den Wert Eins annehmen. Dann kann über die Signale CYC und STB ein Buszyklus eingeleitet werden. Dabei ist es möglich, dass ein Buszyklus mehrere Lese- oder Schreibzugriffe enthalten. Während des gesamten Zugriffes muss das CYC-Signal den Wert Eins beibehalten. Das Signal STB beeinflusst die Geschwindigkeit mit der Daten ausgetauscht werden. Um eine Pause zwischen zwei Lese- oder Schreibzugriffen zu erzeugen, muss lediglich nach dem Abschluss des Zugriffs STB auf den Wert Null gesetzt werden. Erst wenn STB wieder den Wert Eins annimmt, wird dem Slave ein weiterer Zugriff signalisiert. Der Slave muss durch das ACK-Signal den Abschluss eines Lese- oder Schreibzugriffs bestätigen. Es gibt keine Vorgabe in welcher Zeit der Slave eine Anfrage beantworten muss und daher kann auch der Slave Einfluss auf die Geschwindigkeit der Datenübertrag nehmen. Im Falle eines Datenbusses der mehr als ein Byte umfasst wird durch das SEL-Signal angegeben welche Bytes geschrieben oder gelesen werden sollen.

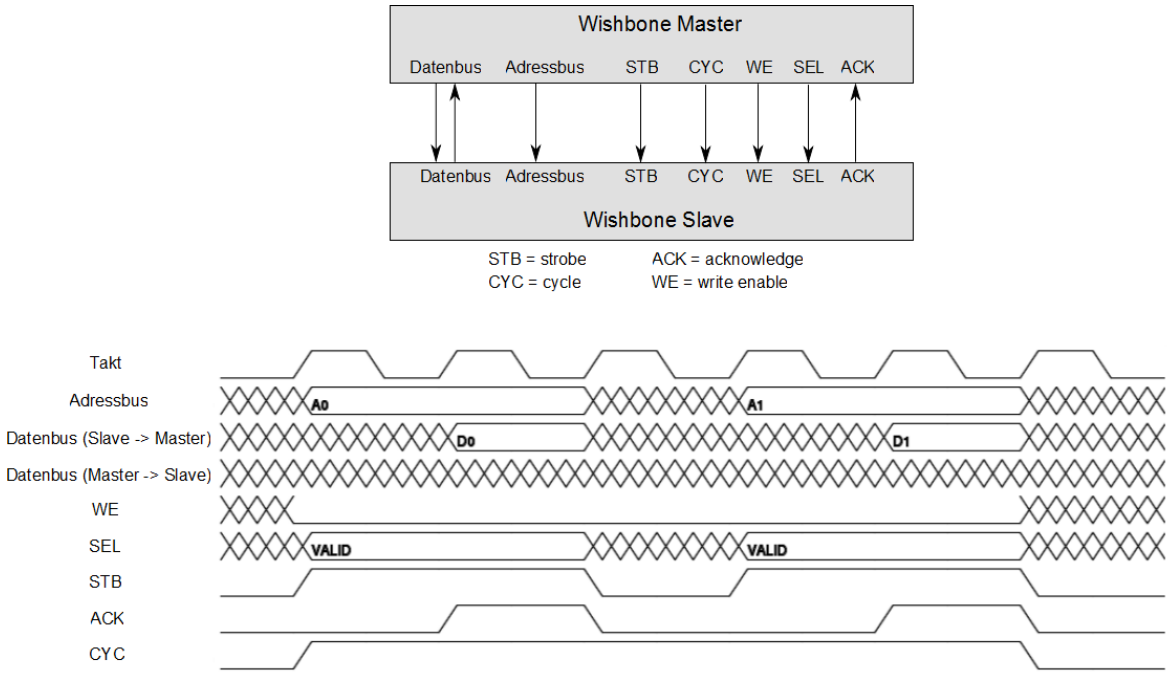

Abbildung 6.2: On-Chip Bus Standard: Wishbone (nach [33])

Indem der Master das CYC- und STB-Signal auf Null setzt, wird der Zugriff beendet. Die Signale werden dabei immer synchron, mit der steigenden Flanke des Taktsignals, ausgewertet.

6.1.2 Prioritäten Arbiter

Die Zugriffssteuerung von gemeinsam verwendeten Bussen übernimmt ein Prioritäten Arbiter. Da der Schwerpunkt auf der Entwicklung einer großen Vielfalt des Transaktionsspeichers liegt, ist für das Funktionieren der Speicher Familie zunächst nur eine Implementierung des Arbiter nötig. Eine dezentrale Verwaltung des Zustands erfordert pro Prozessor einen Speicher-Controller. Dies hat den Vorteil, dass die Zugriffe auf den Speicher-Controller, die keinen Speicherzugriff darstellen, nicht durch den Flaschenhals des gemeinsam benutzten Busses müssen. Daher ist es ausreichend durch Multiplexing des Adressbusses, Datenbusses und ggf. einige Steuersignale die vom Speicher-Controller ausgehenden Informationen durch den Arbiter an den Speicher weiterzuleiten. Dies ist in Abbildung 6.3 dargestellt. In der Gegenrichtung können die Speicher-Controller direkt mit den Ausgängen des Speichers verbunden werden, da hier keine Kollisionen entstehen können.

Die Vergabe der Prioritäten wird anhand der Anschlussreihenfolge festgelegt. Die Anschlüsse sind zu diesem Zweck durchnummeriert, wobei mit steigendem Index auch die Priorität des Anschlusses steigt. Die Anzahl an Anschlüssen ist dabei variabel und wird über ein Generic in VHDL konfiguriert. Weiterhin werden benötigte Busbreiten über Generics angegeben. Je nach dem zu welchem Speichermodell ein Arbiter generiert wird,

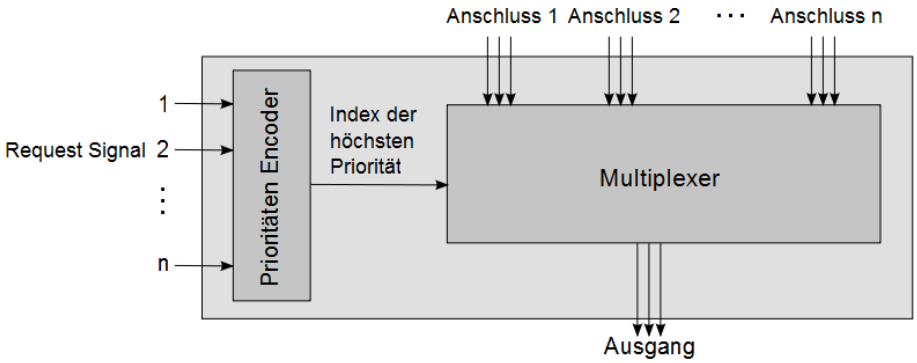

Abbildung 6.3: Arbitrierung durch Multiplexing

ändert sich die Zusammenstellung der Signale die gemultiplext werden müssen. Da dies nicht oder nur sehr umständlich in VHDL selbst erledigt werden kann, wird diese Aufgabe durch XVCL übernommen. Die Konfigurierung durch XVCL wird in Kapitel 7 beschrieben.

6.2 Aufbau des FIFO-Speichers

Aufgrund der einfachen Struktur eines FIFO-Speichers und der geringen Anzahl an Variationsmöglichkeiten wird hier ein sehr spezialisierter FIFO-Speicher implementiert, der im Gegensatz zum Transaktionsspeicher nur geringfügig angepasst werden kann. Die Dual Port Block-RAMs der Xilinx FPGAs bieten sich dabei an, sie direkt in den FIFO-Speicher zu integrieren und damit einen hohen Datendurchsatz zu erzielen.

6.2.1 Hardware Implementierung

Da die beiden Ports der Block-RAMs völlig unabhängig von einander betrieben werden können, ist es naheliegend für eine gute Leistung die Anzahl der Anschlüsse auf zwei festzulegen. So können sowohl der lesende als auch der schreibende Prozessor unabhängig und gleichzeitig auf den Speicher zugreifen. Insbesondere können sie mit unterschiedlichem Takt betrieben werden. Um die Logik so einfach und ressourcensparend wie möglich zu halten, ist es nicht möglich den Füllstand des FIFO-Speichers zu überprüfen. Es können lediglich neue Daten vom Schreiber am Ende abgelegt und Daten vom Anfang durch den Leser verarbeitet werden.

Dazu benötigt es zwei einfache Zähler, die ein Bit mehr haben als zum Durchlaufen der Speicheradressen nötig sind. Jeweils einen Zähler für den Leser und den Schreiber. Sofern beide Zähler den gleichen Wert haben ist der Speicher leer und daher ist es sehr einfach festzustellen, wann der Leser blockiert werden muss. Mit jedem Zugriff wird der entsprechende Zähler hoch gezählt. Da es ein Bit mehr gibt als zum Darstellen der möglichen Speicheradressen nötig ist, kann ein voller Speicher daran erkannt werden, dass das höchstwertige Bit der beiden Zähler unterschiedlich ist und die restlichen Bits gleiche Werte aufweisen. Damit kann auch schnell entschieden werden, wann der Schreiber

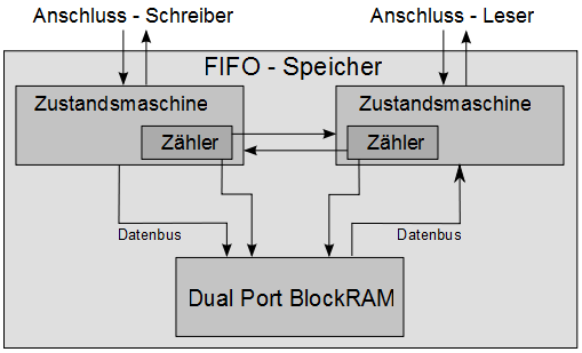

Abbildung 6.4: Aufbau des FIFO-Speichers

blockiert werden muss. Jeweils eine einfache Zustandsmaschine für die angebundenen Prozessoren steuert dabei die Wishbone konforme Kommunikation und blockiert ggf. den Zugriff indem kein ACK-Signal gesendet wird. Abgesehen von der Busbreite und der Speichergröße ist nichts weiter konfigurierbar.

6.2.2 Benutzung des FIFO-Speichers durch die Software

Aus Sicht des Programmierers ist die Ansteuerung des FIFO-Speichers sehr einfach. Da alle Peripheriegeräte über festgelegte Speicheradressen angesprochen werden, kann der Schreiber über genau eine Adresse neue Daten in den FIFO-Speicher ablegen. Entsprechend kann der Leser immer ein Datum über die konfigurierte Adresse abholen, sofern Daten vorhanden sind. Da der entsprechende Prozessor zwangsläufig angehalten wird, falls der FIFO-Speicher leer bzw. voll ist, sind keine weiteren Maßnahmen zur Synchronisation notwendig.

6.3 Struktur des Speicher-Controller für read-modify-write Operationen

Bei den in der LavA-Plattform eingesetzten Prozessoren handelt es sich um RISC[1]-Prozessoren, die keine Instruktionen für read-modify-write Operationen zur Verfügung stellen. Daher ist ein Speicher-Controller für die Umsetzung einer solchen Operation nötig, um auch Programme mit klassischen Synchronisationsmechanismen wie Locks, Semaphore und Monitore implementieren zu können. Da die unterschiedlichen read-modify-write Operationen sich nur geringfügig unterscheiden und keine signifikante Vorteile bietet, wurde eine Test and Set Variante umgesetzt.

[1]Redueced Instruction Set Computer

6.3.1 Umsetzung des Speicher-Controllers

Eingebettet in eine Blackbox können über einige Generic Parameter die Anzahl der Anschlüsse für Prozessoren, die Busbreite und die Größe des Hintergrundspeichers angegeben werden. Dabei muss die Größe des Speichers bekannt sein, damit die benötigte Adresse für das Register des Speicher-Controllers bestimmt werden kann. Da bei größeren MPSoCs die Leitungslänge vom Prozessor bis zum gemeinsamen Speicher zwischen den einzelnen Prozessoren sehr stark schwanken kann und damit die maximale Taktfrequenz negativ beeinflusst wird, puffert der Speicher-Controller die zu lesenden Daten. Da für die Umsetzung einer read-modify-write Operation ohne hin ein Register zum Puffern des alten Wertes benötigt wird, verbraucht dieses Vorgehen keine zusätzlichen Ressourcen. Es ergibt sich lediglich eine Verzögerung des Speicherzugriffs um einen Takt, zugunsten eines potenziell höheren Taktes.

Der gemeinsame Speicher wird, wie alle anderen Komponenten, in den Adressraum des Prozessors eingeblendet. Um Zugriff auf den Speicher zu erhalten muss der Speicher-Controller erst vom Arbiter dazu berechtigt werden. Dazu erzeugt der Speicher-Controller ein Request-Signal und bekommt abhängig von der Scheduling-Strategie bei nächster Gelegenheit Zugriff auf den gemeinsamen Speicher. Über einen schreibenden Zugriff auf eine konfigurierte Adresse des Speicher-Controllers kann ihm mitgeteilt werden, dass der nächste schreibende Zugriff auf den Speicher eine Test and Set Operation sein soll. Sobald der nächste schreibende Speicherzugriff erfolgt, wird der Speicher-Controller tätig und führt auf der gewünschten Adresse eine Lese-Operation durch. Der alte Wert wird in einem Register gespeichert. Anschließend wird die gewünschte Schreib-Operation durchgeführt. Dabei behält der Speicher-Controller über die Dauer beider Speicherzugriffe den Bus, so dass die zwei Speicherzugriffe atomar durchgeführt werden. Die darauf folgende Lese Operation darf nicht auf den gemeinsamen Speicher erfolgen, sondern muss das Register des Speicher-Controller auslesen, um den alten Wert der Speicheradresse auswerten zu können. Da nur ein Register vorhanden ist, würde sonst eine Lese-Operation auf den gemeinsamen Speicher den alten Wert im Register überschreiben.

Genau wie beim FIFO-Speicher existiert eine kleine Zustandsmaschine, die für jeden Prozessor die Wishbone konforme Kommunikation ermöglicht. Darüber hinaus muss sie auch die Test and Set Operation umsetzen und das Request-Signal für den Arbiter erzeugen. Ein Register mit der Breite des Datenbusses dient als Puffer für die Daten. In einem weiteren ein Bit Register wird die Absicht für die Ausführung einer Test and Set Operation vermerkt. Ein nachfolgender schreibender Speicherzugriff setzt das Bit wieder zurück. Der Arbiter führt dabei das Multiplexing durch, erzwingt allerdings niemals das Beenden einer Speicheroperation und damit müssen die Speicher-Controller von sich aus den Bus wieder frei geben.

6.3.2 Implementierung der Test and Set Operation in Software

In Listing 6.1 ist eine einfache C Implementierung der Test and Set Operation angegeben. Dazu muss zunächst schreibend auf das Register des Speicher-Controllers zugegriffen werden. Das Register kann dadurch nicht überschrieben werden, es wird lediglich die

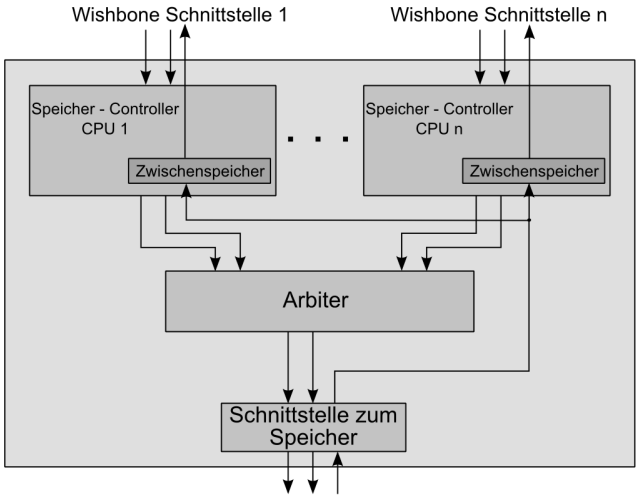

Abbildung 6.5: Aufbau des read-modify-write Speicher-Controllers

Absicht für die Ausführung einer Test and Set Operation vermerkt. Der nächste schreibende Zugriff wird schließlich die Test and Set Operation anstoßen. Abschließend muss nur noch der alte Wert zurückgegeben werden, um ihn in der Anwendung auswerten zu können.

Listing 6.1: C Implementierung der Test and Set Operation

```
// fiktive Adresse des Registers für Test and Set Operationen
#define test_and_set_reg ((volatile int*)(0x1000))

int function test_and_set(int* var, int set_val) {
    *test_and_set_reg = 0;
    *var = set_val;
    return *test_and_set_reg;
}
```

6.4 Aufbau des Transaktionsspeichers

Die Design Entity des Transaktionsspeichers umfasst eine variable Anzahl an Wishbone-Anschlüssen für die Prozessoren, dabei kommt ein Prioritäten Arbiter zum Einsatz. Die Prioritäten werden dabei wie in Abschnitt 6.1.2 beschrieben verteilt. Weiterhin können über Generics die Größe des Hintergrundspeichers und die Datenbusbreite eingestellt werden. Für jeden Prozessor kann individuell die Möglichkeit für byteweisen Zugriff, nicht transaktionale Speicheroperationen und die Anzahl an Block-RAMs für den Zwischenspeicher bestimmt werden. Alle weiteren Konfigurationsmöglichkeiten sind durch XVCL realisiert und werden in Kapitel 7 besprochen.

Die Grundkomponenten des Transaktionsspeichers sind in Abbildung 6.6 zu sehen. Die Idee dahinter besteht darin, durch Austauschen der Architekturen die unterschiedlichen

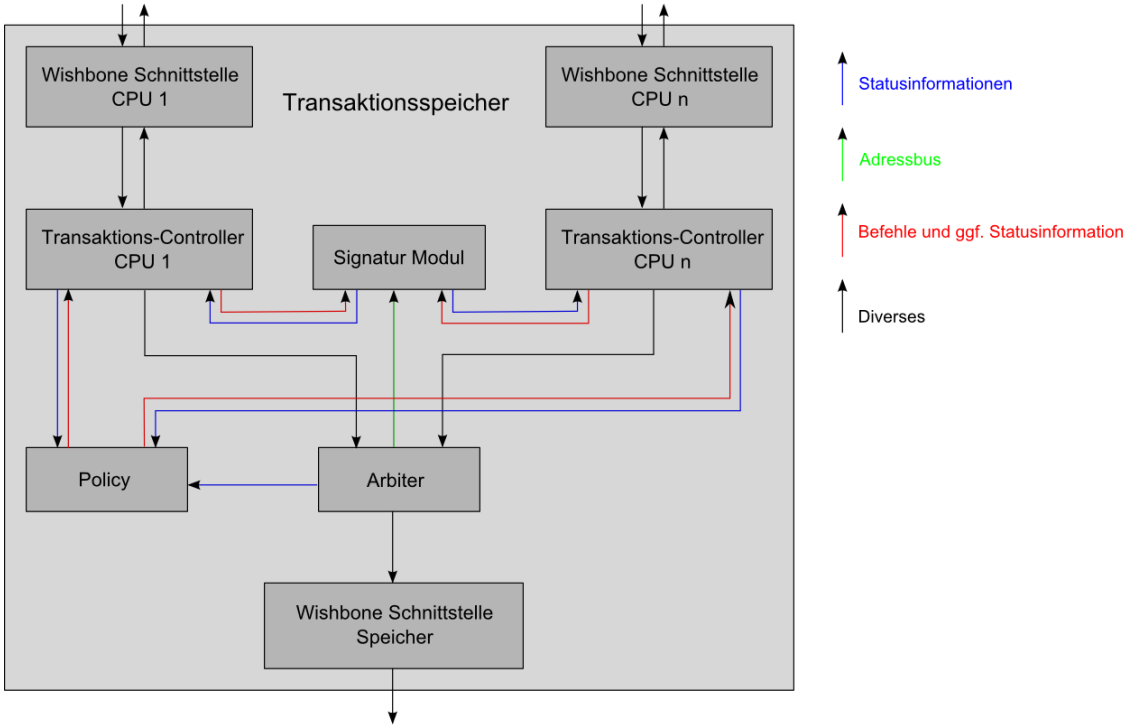

Abbildung 6.6: Struktur des Transaktionsspeichers (Verbindungen von der Schnittstelle zurück zu den Transaktions-Controllern nicht eingezeichnet)

Konfigurationsmöglichkeiten bereit zu stellen. Jeder Prozessor wird über eine Wishbone-Schnittstelle angebunden, die die Zugriffe auf den Speicher steuert und die Befehle des Prozessors (Begin, Commit und Abort) entgegen nimmt. Direkt dahinter sitzt pro Prozessor ein Transaktions-Controller, der den Zwischenspeicher beinhaltet und unter Berücksichtigung des intern genutzten Kommunikationsschemas den Zugriff auf den Speicher ermöglicht. Ein zentrales Policy-Modul entscheidet bei Konflikten welche Transaktionen abgebrochen werden müssen. Hinter dem Arbiter werden die Speicherzugriffe durch eine Wishbone-Schnittstelle wieder Wishbone konform an den Hintergrundspeicher weitergeleitet.

Für eine schnelle Konflikterkennung sorgt das Signatur-Modul, welches einen Bloom-Filter implementiert. Es wird hier auf die Verwendung von Caches verzichtet, da zum einen die Block-RAMs bereits schnelle Speicher darstellen und Caching damit keine Leistungssteierung erzielen kann. Zum anderen kann es durch eine ungünstige Abfolge von Speicheroperationen dazu kommen, dass Daten die im Write-Set enthalten sind aus dem Cache ausgelagert werden müssen. Damit sind Mechanismen notwendig, um das verdrängte Datum des Write-Sets weiterhin zwischenspeichern und eine korrekte Konflikterkennung gewährleisten zu können. Je kleiner ein solcher Cache ist, desto größer wird die Wahscheinlichkeit für das Auslagern von Daten des Write-Sets. Da die Anzahl der Block-RAMs ein limitierenden Faktor bei MPSoCs auf FPGAs darstellt, werden die Daten des Write-Sets nacheinander im Zwischenspeicher abgelegt, um die Notwendigkeit

für große Caches zu vermeiden. Darüber hinaus sind die veröffentlichten Systeme so unterschiedlich und spezialisiert, dass es nur mit sehr hohem Aufwand möglich wäre aus ihnen eine konfigurierbare Plattform zu erstellen. Eine Neuentwicklung, ohne den Einsatz von Caches und für eine kleine FPGA taugliche Lösung, liegt daher nahe, insbesondere da die MPSoCs der LavA-Plattform bis jetzt auf einen Programmspeicher in Form von Block-RAMs angewiesen sind.

6.4.1 Das interne Kommunikationsschema

Das Kommunikationsschema dient dazu die einzelnen Teilnehmer über Busaktivitäten zu informieren und ist darauf ausgelegt möglichst viele Interaktionsschemata abzudecken, die sich durch das Zusammenspiel von Policy, Versionierung und Konflikterkennung ergeben. Dabei lassen sich die Kommunikationsanfragen in zwei Klassen unterteilen: Speicherzugriffe und Überprüfung einer Adresse auf Konflikte. Um Zeit zu sparen, können auch beide Kommunikationsanfragen für eine bestimmte Adresse gleichzeitig ausgeführt werden, da die Ansteuerung des Speichers über ein separates Signal läuft. Für alle Interaktionen muss zunächst der gemeinsam genutzte Bus akquiriert werden. Danach wird über das Kommunikationsanfragesignal die benötigte Aktion angestoßen. Ein Buszyklus beschreibt dabei die gesamte Dauer für die ein Transaktions-Controller den gemeinsamen Bus nutzen muss. Es gibt vier unterschiedliche Befehle:

- Überprüfe Lesezugriff – Alle Teilnehmer müssen überprüfen, ob ein Konflikt vorliegt, falls auf der auf dem gemeinsamen Bus anliegende Adresse gelesen wird
- Überprüfe Schreibzugriff – Alle Teilnehmer müssen überprüfen, ob ein Konflikt vorliegt, falls auf der auf dem gemeinsamen Bus anliegende Adresse geschrieben wird
- Speicheroperation – Es findet eine Speicheroperation statt, ohne Überprüfung (z. B. ursprünglichen Speicherzustand wiederherstellen oder nach erfolgreichem Validieren das übertragen der Daten in den Speicher) oder eine neue Adresse muss neu geladen werden
- Buszyklus beendet – Signalisiert das Ende des Zugriffes auf den gemeinsamen Bus

Die Antwort auf die Anfrage einer Überprüfung werden auf dem Antwortbus bekannt gegeben. Hier werden drei Antwortmöglichkeiten unterschieden:

- Überprüfe – Ruhezustand bzw. signalisiert das die Überprüfung noch andauert
- Kein Konflikt – signalisiert, dass die Adresse auf dem gemeinsamen Bus keinen Konflikt verursacht
- Konflikt – signalisiert, dass die Adresse auf dem gemeinsamen Bus zu einem Konflikt geführt hat

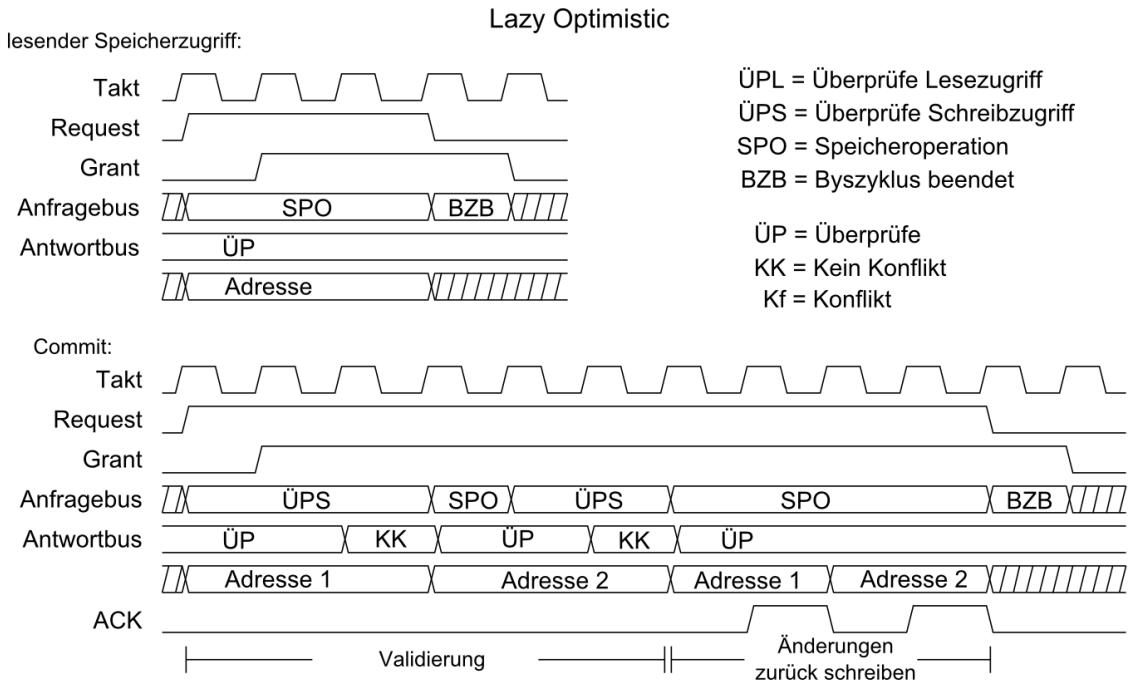

Abbildung 6.7: Kommunikationsprotokoll für Lazy Optimistic

Da beispielsweise im Rahmen eines Commits, bei einer optimistischen Konflikterkennung, mehr als eine Adresse einer Überprüfung unterzogen werden müssen, ist es erforderlich, dass sich während eines Buszykluses die durchzuführenden Befehle ändern können. Dabei sind keine festen Zeiten vorgegeben, in denen ein Befehl abgearbeitet werden muss. Da insbesondere die unterschiedlichen Teilnehmer eine Anfrage unterschiedlich schnell beantworten können, muss das Ergebnis der Überprüfung stabil auf dem Antwortbus gehalten werden, während das Kommunikationsanfragesignal einen konstanten Wert aufweist und eine Überprüfung gefordert wird. Durch den Wechsel von einer Überprüfungsaktion zu einer Speicheroperation wird signalisiert, dass die Auswertung abgeschlossen ist. Der Businhaber kann nun eine neue Adresse auf den gemeinsamen Bus legen und eine neue Überprüfung anstoßen. Das Ende eines Buszyklus muss immer mit dem *Buszyklus beendet* Befehl abgeschlossen werden.

Abbildung 6.7 zeigt wie aus diesen Regeln die nötigen Interaktionen gebildet werden können, um ein Transaktionsspeicher mit lazy optimistic zu realisieren. Bei einer lazy optimistic Strategie muss während einer Transaktion niemals eine Überprüfung stattfinden, daher werden alle lesenden Speicherzugriffe durch den *Speicheroperation*-Befehl gekennzeichnet. Schreibende Zugriffe werden gar nicht auf dem gemeinsamen Bus signalisiert, da sie im Zwischenspeicher gehalten werden, bis die Transaktion durch ein Commit abschlossen wird. Im Rahmen des Commits müssen dann zunächst alle Adressen des Write-Sets überprüft werden. Dazu wird der Zwischenspeicher durchlaufen und jede Adresse einer Überprüfung durch den *Überprüfe Schreibzugriff*-Befehl unterzogen. Nach erfolgreichem Validieren können alle geschriebenen Daten mit Hilfe des Speicheroperation-Befehls in den Speicher übertragen werden. Im Falle eines Reverts muss nichts weiter

passieren, da keine Veränderungen am Speicher vorgenommen wurden.

Im Gegensatz dazu muss bei einer eager pessimistic Strategie jeder Speicherzugriff überprüft werden. Dazu begleitet der *Überprüfe Lesezugriff*-Befehl entsprechend die lesenden Speicherzugriffe. Die Überprüfung und das Lesen des Datums kann dabei parallel geschehen. Im Falle eines Konfliktes wird ggf. das gelesene Datum einfach verworfen. Vor jedem Schreibzugriff muss die adressierte Speicherzelle erst gelesen werden, um den alten Wert im Zwischenspeicher unterzubringen. Parallel zu dem Lesevorgang kann die Überprüfung durch den Überprüfe Schreibzugriff-Befehl angestoßen werden. Im Falle eines Konflikts wird nichts geschrieben und das gelesene Datum verworfen. Da bei einem Commit nichts weiter getan werden muss, ist auch keine Kommunikation über den geteilten Bus erforderlich. Ein Revert muss nach erfolgreichem akquirieren des Busses lediglich alle ursprünglichen Werte wiederherstellen. Dazu genügt der Speicheroperation Befehl.

Die lazy pessimistic Variante stellt eine Mischung aus den beiden anderen Strategien dar. Jeder Speicherzugriff muss entsprechend überprüft werden, dabei werden die Daten allerdings nicht direkt in den Speicher zurück geschrieben. Daher muss im Falle eines Reverts auch nichts weiter getan werden. Da alle Speicherzugriffe bereits überprüft wurden, kann beim Commit direkt mit der Übertragung der Änderungen in den Speicher begonnen werden. Sobald ein Konflikt bei einem lesenden Speicherzugriff auftritt, kann das gelesene Datum gespeichert werden und nach der Auflösung des Konflikts ggf. direkt verwendet werden.

6.4.2 Die Wishbone-Schnittstelle

Im Wesentlichen realisieren die Wishbone-Schnittstellen die Wishbone konforme Kommunikation mit den Prozessoren, sowie mit dem Speicher. Dazu agiert die Wishbone-Schnittstelle zum Speicher als Master und benötigt dazu eine entsprechende Zustandsmaschine. In Richtung der Prozessoren ist ein Wishbone-Slave implementiert, der zusätzlich ein Register für Statusinformationen bereit hält, sowie ein Register zum Starten, Abbrechen und Beenden von Transaktionen. Im Falle von Konflikten und einem daraus resultierenden Revert wird der Prozessor durch einen Interrupt auf diesen Umstand hingewiesen. Die Software muss dann entsprechend reagieren und ggf. den Prozessorzustand zurücksetzen. Da kein unbegrenzter Zwischenspeicher zur Verfügung steht, wird bei einem Überlauf des Speichers auch ein Interrupt generiert. Es sind keine weiteren Mechanismen implementiert, um mit Überläufen umzugehen, so dass die Software diesem Problem begegnen muss. Die Tabellen 6.1 und 6.2 zeigen die abrufbaren Statusinformationen und die möglichen Befehle.

6.4.3 Der Transaktions-Controller

Der Transaktions-Controller setzt sowohl die Konflikterkennung als auch die Versionierung um. Für jede der drei möglichen Kombination existiert eine eigene Architektur, die durch XVCL konfiguriert werden kann. Die unterschiedlichen Transaktions-Controller

Bits (31...0)	Bedeutung
2 - 0	Status des Transaktions-Controllers (running, operating, conflicted ...)
3	ob eine Transaktion gestartet wurde
4	ob Zwischenspeicher voll ist
5	ob die letzte Transaktion aufgrund eines Konflikts zurück gerollt wurde
6	ob versucht wurde auf den Speicher zuzugreifen, ohne Transaktionen zu aktivieren (nur falls nicht erlaubt)
31 - 7	nicht benutzt

Tabelle 6.1: Statusinformation des Transaktionsspeichers

Bitmuster (2...0)	Bedeutung
000	Transaktion starten (Beginn)
001	Ende der Transaktion (Commit)
010	Abbruch der Transaktion (Abort)
100	bestätigen des Interrupts für Revert
101	bestätigen des Interrupts für vollen Zwischenspeicher

Tabelle 6.2: Befehle des Transaktionsspeichers

sind zwar sehr ähnlich, allerdings ist die Verwaltung und Steuerung der Kommunikation durch eine Zustandsmaschine realisiert, welche sich nur mit hohem Aufwand durch VHDL konfigurieren lassen. Die üblichen Eigenschaften, wie die Busbreite, die Möglichkeit für byteweise Zugriffe und die Größe des Zwischenspeichers sind über Generics konfigurierbar.

Da das Signatur-Modul nicht über die Information verfügt wer aktuell auf dem gemeinsam genutzten Bus agiert, werden alle Überprüfungsanfragen entsprechend beantwortet. Da es sehr wahrscheinlich ist, dass innerhalb einer Transaktion mehrmals die selbe Adresse gelesen und oder geschrieben wird, erzeugt das Signatur-Modul für diesen Transaktions-Controller einen Konflikt. Daher wird das Ergebnis des Signatur-Moduls im Falle des eigenen Überprüfungswunsches gefiltert und auf Kein Konflikt gesetzt. Mit dem erfolgreichem Abschluss einer Speicheroperation, wird die entsprechende Adresse dem Read- bzw. Write-Set hinzugefügt, dass von dem Signatur-Modul verwaltet wird. Bei einer lazy optimistic Strategie ist es sinnvoll zusätzlich von jedem Speicher-Controller die aktuelle Adresse an das Signatur-Modul weiterzuleiten, da so das Write-Set ohne Benutzung des gemeinsamen Busses aktualisiert werden kann. Dadurch lässt sich die Last auf den gemeinsam genutzten Bus verringern.

Tabelle 6.3 zeigt die verschiedenen Zustände in denen sich der Speicher-Controller befinden kann. Diese sind zum einen als Information für den Prozessor gedacht, als auch für die Policy, die ggf. intern genutzte Datenstrukturen aktualisieren muss. Daher müssen die Zustände *reverting* und *commiting* im Falle eines Revert bzw. Commits für mindestens einen Takt durchlaufen werden, auch wenn z. B. bei einem Revert einer lazy Versionierung nichts getan werden muss. Die Zustände reflektieren die korrekte Bear-

Zustand	Bedeutung
lazy	keine laufende Transaktion
running	Transaktion gestartet, aber aktuell untätig
operating	Bearbeiten eines Speicherzugriffs
conflicted	es ist ein Konflikt aufgetreten
reverting	Revert der Transaktion wird ausgeführt
commiting	Commit der Transaktion wird ausgeführt

Tabelle 6.3: Zustände des Speicher-Controllers

beitung der Befehle und dienen daher auch zur Überprüfung der korrekten Arbeitsweise des Speicher-Controllers.

6.4.4 Das Signatur-Modul

Die Implementierung des Signatur-Moduls umfasst einen einfachen Bloom-Filter. Zur Berechnung des Hashwertes einer Adresse wird diese in gleich große Blöcke aufgeteilt. Die Größe eines Blocks ist dabei über ein Generic einstellbar. Falls die Anzahl der Bits der Adresse nicht durch die Größe des Blocks teilbar ist, werden die letzten beiden Blöcke derart aufgeteilt, dass mindestens drei Bit pro Block vorhanden sind. Zu jedem dieser Blöcke existiert ein Block im Hashwert, der für jede mögliche Bitkombination eines Adressblocks genau ein Bit enthält. Jeder Adressblock, aufgefasst als positive Binärzahl, adressiert das entsprechende Bit im Hashwert-Block. Durch ein bitweises Oder können zwei Hashwerte zusammengeführt (Signatur) und damit das Read- und Write-Set repräsentiert werden.

Bei der Überprüfung einer Adresse handelt es sich um eine Neue, falls nicht alle korrespondierenden eins Bits des Hashwertes der zu überprüfenden Adresse auch in der Signatur gesetzt sind. Entsprechend kann davon ausgegangen werden, dass die Adresse bereits in der Signatur vorhanden ist, falls alle entsprechenden eins Bits gesetzt sind. Dabei ist es möglich, dass eine Adresse als bereits gesehen deklariert wird, wobei dies allerdings nicht der Fall ist. Da die Aussage, dass eine Adresse noch nicht in der Signatur enthalten ist, immer korrekt ist, bleibt auch bei fälschlicherweise angenommenen Konflikten die Korrektheit des Transaktionssystems unberührt. Durch die Repräsentation der Signatur als Hashwert müssen unnötig abgebrochene Transaktionen in kauf genommen werden. Abbildung 6.8 verdeutlicht nochmal die Funktionsweise der Signatur.

6.4.5 Das Policy-Modul

Das Policy-Modul ist eine reine Hardwareimplementierung, um möglichst schnell eine Entscheidung treffen zu können. Eine Implementierung der Policy in Software erfordert außerdem einige Änderungen an den CPUs, die im Rahmen dieser Arbeit nicht gemacht werden können. Sobald ein Wechsel von einem Speicherzugriff zum Ausführen eines Software-Handlers für die Umsetzung der Policy vollzogen werden muss, ist es notwendig alle Befehle nach dem Speicherzugriffs-Befehl aus der Pipeline des Prozessors zu

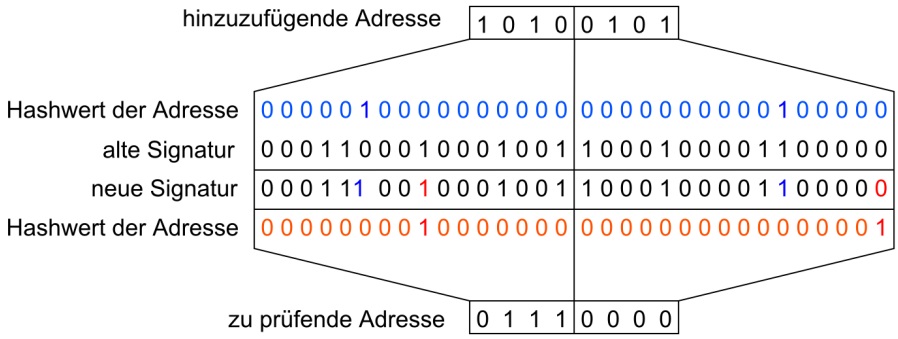

Abbildung 6.8: Funktionsweise der Signatur (Bloom-Filter). Die zu überprüfende Adresse unten ist nicht in der Signatur enthalten, da das Bit ganz rechts sonst auf 1 gesetzt wäre.

entfernen. Insbesondere muss der Speicherzugriff ggf. nach der Entscheidung der Policy wiederholt werden, damit auch tatsächlich der korrekte Wert aus dem Speicher gelesen werden kann.

Für die Entscheidung welche Transaktionen fortfahren dürfen und welche nicht, sind einige Informationen nötig. Zunächst muss erstmal ermittelt werden welche Prozessoren überhaupt eine Transaktion gestartet haben, damit ggf. eine Reihenfolge festgehalten werden kann. Weiterhin muss bekannt sein, welche Transaktionen an einem Konflikt beteiligt sind. Dazu werden die Ergebnisse einer Überprüfung von allen Transaktionen gesammelt und ausgewertet, die dann den Speicher-Controllern zur Verfügung gestellt wird. Darüber hinaus muss bekannt sein, welche Transaktion aktuell den Bus besitzt, um die Aufteilung in Angreifer (Inhaber des Busses) und Verteidiger (alle die einen Konflikt melden) vorzunehmen. Eventuell sind für hier nicht implementierte Policies weitere Informationen nötig, die entsprechend der Policy zur Verfügung gestellt werden müssen.

6.4.5.1 Attacker-Wins

Attacker-Wins ist eine sehr einfache Policy, die im Falle eines Konflikts immer den Angreifer fortfahren lässt. Dazu müssen alle anderen Transaktionen die mit dem Angreifer in Konflikt stehen abgebrochen werden. Eine einfache Zustandsmaschine kann innerhalb eines Taktes entscheiden welche Transaktionen abgebrochen werden müssen. Diese Policy hat den Nachteil, dass es nur für die lazy optimistic Strategie eine Fortschrittsgarantie gibt, da Konflikte immer nur beim Commit auftreten und der Commit immer erfolgreich ist. Bei einer pessimistischen Konflikterkennung kann es zum Livelock kommen, bei dem sich Transaktionen immer gegenseitig abbrechen.

6.4.5.2 Oldest-Wins

Für die Konfigurationen mit pessimistischer Konflikterkennung kann auch für die Oldest-Wins Policy eine Fortschrittsgarantie gegeben werden, da die älteste Transaktion garan-

tiert bis zum Commit kommen wird. Um feststellen zu können welche Transaktion die älteste ist, genügt es die Reihenfolge in der Transaktionen gestartet werden in Form einer Liste zu speichern. Im Falle eines Commits oder Reverts werden die entsprechenden Transaktion aus der Liste gelöscht. Es ist ausreichend für jeweils einen Eintrag pro angeschlossenem Prozessor Speicher in der Liste zu reservieren. Im Falle eines Konflikts muss die Liste nur vom ältesten zum neuesten Eintrag durchlaufen werden. Die erste Transaktion, die an dem Konflikt beteiligt ist, darf fortfahren. Falls es sich um einen Verteidiger handelt, reicht es aus den Angreifer abzubrechen. Sonst müssen alle Verteidiger einen Revert durchführen.

6.4.6 Verwendung von Transaktionen im Programm

Die grundlegenden Funktionen, die durch die Software gewährleistet werden müssen, sind die Sicherung des Prozessorzustandes zu Beginn einer Transaktion und im Falle eines Reverts oder Aborts den ursprünglichen Prozessorzustand wiederherzustellen. Da die im LavA-Projekt verwendeten Prozessoren keinen Mechanismus zum automatischen Sichern des Prozessorzustandes bieten, muss dies durch eine Software Routine erledigt werden. Mit Hilfe von Makros (TM_BEGIN(), TM_COMMIT(), TM_ABORT und TM_RESTOR()) werden alle benötigten Software Routinen für den Programmierer bereitgestellt.

Das TM_BEGINN() Makro ruft zunächst eine in Assembler geschriebene Funktion auf, die den Prozessorzustand sichert. Da sich die unterschiedlichen Prozessoren in der ISA[2] unterscheiden, muss für jeden Prozessor eine eigene Assembler Routine implementiert werden. Nach dem Sichern des Prozessorzustandes wird dem Transaktions-Controller mitgeteilt, dass fortan Transaktionen aktiviert sind. Danach werden alle Lade- und Speicher-Befehle automatisch als Teil der Transaktion interpretiert, sofern sie den gemeinsamen Speicher adressieren.

Mit dem TM_ABORT() Makro wird lediglich eine C-Funktion aufgerufen, die dem Transaktions-Controller den Befehl zum Abbruch erteilt. Entsprechend verhält sich das TM_COMMIT() Makro, welches nur den Commit-Befehl ausführt. Im Falle eines fehlgeschlagenen Commits wird automatisch das TM_RESTOR() Makro ausgeführt, dass einen Neustart der Transaktion einleitet. Dazu ist wieder eine vom Prozessor abhängige Assembler Routine notwendig.

Listing 6.2: Implementierung der C-Funktionen

```
#define tm_com_reg   ((volatile int*)(0x1000))
#define tm_com_enable                 0
#define tm_com_commit                 1
#define tm_com_abort                  2
#define tm_com_serv_int_revert        4
#define tm_com_serv_int_mem_full      5

inline void _tm_begin() {
```

[2]Instruction Set Architecture

```
    *tm_com_reg = tm_com_enable;
}

inline void _tm_commit() {
    *tm_com_reg = tm_com_commit;
}

inline void _tm_abort() {
    *tm_com_reg = tm_com_abort;
}

/* Konflikt der zu einem Revert geführt hat */
void _tm_ISR_revert() {
    *tm_com_reg = tm_com_serv_int_revert;
    _tm_enable_int();
    TM_RESTORE();
}

/* Zwischenspeicher voll oder nicht erlaubter Speicherzugriff */
void _tm_ISR_access_vio() {
    *tm_com_reg = tm_com_serv_int_mem_full;
    _tm_enable_int();
    TM_RESTORE();
}
```

Listing 6.2 zeigt die Implementierung der C-Funktionen für die unterschiedlichen Aktionen. Dabei handelt es sich um eine minimale Umsetzung, da es nur die Möglichkeit zum Starten und Beenden einer Transaktion gibt. Sobald ein Problem auftritt, wie ein erzwungener Revert oder ein voller Zwischenspeicher, müssen sich Interrupt Service Routinen (ISR) dieser annehmen. In dieser minimalen Implementierung wird automatisch ein Neustart der Transaktion eingeleitet.

7 Konfigurierung der Speicher Familie

Die automatische Konfigurierung der Speicher Familie ermöglicht die schnelle Generierung funktionsfähiger Speicher-Systeme. So kann durch Rapid Prototyping die Anwendung auf verschiedenen Plattformen getestet werden. Dabei reduziert die Konfigurierung durch die Auswahl von Merkmalen den Entwicklungsaufwand des gesamten MPSoCs. Gerade die Suche nach dem richtigen Transaktionsspeichersystem kann durch die einfache Generierung der unterschiedlichen Konfigurationsmöglichkeiten getestet werden. Unterkapitel 7.1 beschreibt inwieweit VHDL für die Konfiguration eingesetzt wird. Die VHDL Komponenten wiederum werden mit Hilfe von XVCL erzeugt und Abschnitt 7.2 beschreibt diesen Prozess.

7.1 Konfigurierung durch VHDL

Wie bereits im Kapitel 6 erwähnt sind einige Konfigurationsmöglichkeiten durch Generics der Hardwarebeschreibungssprache VHDL umgesetzt. Die folgenden Eigenschaften können durch Generics konfiguriert werden:

- Datenbusbreite (TS, rmwSC)[1]
- Speichergröße (alle)
- Adressbusbreite (alle)
- Adressblockgröße (TS)
- Anzahl Anschlüsse (TS, rmwSC)
- Anzahl Block-RAMs für Zwischenspeicher (TS)
- byteweise Speicherzugriff (TS, rmwSC)
- Speicherzugriffe außerhalb von Transaktionen (TS)

Mit der Datenbusbreite, Speichergröße, Adressbusbreite und Adressblockgröße werden Register und Verbindungsressourcen auf die benötigte Bitbreite angepasst. Funktionen unterstützen diesen Vorgang, so dass z. B. aus der Einstellung für byteweises Schreiben und der Datenbusbreite automatisch die Anzahl benötigter Schreibfreigabe-Bits berechnet wird. Abhängig von Adressbusbreite und Adressblockgröße wird die benötigte

[1] TS = Transaktionsspeicher; rmwSC = read-modify-write Speicher-Controller

Breite der Signatur bestimmt, so dass die Signatur an die benötigten Ressourcenbeschränkungen angepasst werden kann. Ebenso berechnet eine Kombination aus mehreren Funktionen die Aufteilung des Daten- und Adressbusses auf die zwei Anschlüsse der Block-RAMs, um eine optimale Ausnutzung des Speichers zu erreichen. Abhängig von den benötigten Bits wird die passende 16 oder 32 Bit Konfiguration der Block-RAMs bestimmt.

Die Templates für die Instanziierung von Block-RAMs unterscheiden sich geringfügig je nach Datenbusbreite. Um das korrekte Template auszuwählen wird das if−generate Sprachkonstrukt benutzt. Dabei muss für jedes Template das verwendet werden soll eine eigene if−generate Anweisung vorhanden sein. Wie in Listing 7.1 zu sehen ist, kann dies mit der Auswahl der richtigen Architektur verknüpft werden. In Zeile 2 und 9 wird durch den in Klammern stehenden Namen die Architektur ausgewählt. Ein Nachteil ergibt sich durch die Notwendigkeit für jede if−generate Anweisung die Instanziierung vornehmen zu müssen. Dabei wird es problematisch, sofern es eine Menge an Kombinationsmöglichkeiten gibt, da für jede Möglichkeit eine eigene if−generate Anweisung vorhanden sein muss und dies die Übersichtlichkeit und Wartbarkeit des Codes beeinträchtigt.

Listing 7.1: if-generate Beispiel

```
if port_width = 32 generate
    for ram_inst_32 : ram_32 use entity work.ram(ram_32);
begin
    ram_inst_32 : ram_32
    ...
end generate;

if port_width = 16 generate
    for ram_inst_16 : ram_16 use entity work.ram(ram_16);
begin
    ram_inst_16 : ram_16
    ...
end generate;
```

Je nach Konfiguration müssen unterschiedlich viele Instanzen einer Komponente erzeugt werden. Damit der Code nicht durch entsprechend viele Instanziierungen, die immer gleich aussehen, unübersichtlich wird, steht die for−generate Anweisung zur Verfügung. Da die Anzahl der Anschlüsse und die damit verbundene Anzahl an Transaktions-Controllern über ein Generic bestimmt wird, erledigt die for−generate Schleife die Instanziierung. Ebenso werden die Multiplexer für die variable Anzahl an Adressblöcken des Signatur-Moduls mit Hilfe der for−generate Schleife erzeugt. Um innerhalb der Schleife unterschiedliche Komponenten zu instanziieren muss allerdings wieder die if−generate Anweisung zur Hilfe genommen werden. Dabei besteht immer das Problem, dass durch viele unterschiedliche Instanziierungen der VHDL-Code unnötig vergrößert wird.

7.2 VHDL Code-Erzeugung durch XVCL

Bei der Konfigurierung der Speicher Familie ist die Nutzung der VHDL-Sprachmittel alleine zwar theoretisch ausreichend, doch ist im Sinne der Übersichtlichkeit und Wartbarkeit des Codes eine Alternative notwendig. Mit XVCL können, entsprechend der gewählten Merkmale, spezialisierte VHDL-Komponenten erzeugt werden, die durch einfache Textersetzungen nur den für die aktuelle Konfiguration nötigen Code enthalten. Dadurch können Fehler in den spezialisierten Komponenten schneller gefunden und behoben werden, ohne die Notwendigkeit sehr ähnliche Instanziierungen auseinander halten zu müssen. Außerdem können so komplexe Ausdrücke, die durch die Auswertung von vielen generischen Parametern entstehen und den VHDL-Code schwer verständlich machen, vermieden werden.

Abschnitt 7.2.1 beschreibt die Möglichkeiten zum Anpassen der VHDL-Komponenten. Das folgende Unterkapitel 7.2.2 befasst sich mit einem Problem bei der Generierung des VHDL-Codes. In MPSoCs kann es angebracht sein mehrere Transaktionsspeicher zu verwenden, die in einigen Teilen identisch und in anderen unterschiedlich sind. Dabei dürfen allerdings die gemeinsamen VHDL-Komponenten nicht doppelt erzeugt werden, da die vergebenen Architektur- und Entity-Namen eindeutig sein müssen. Darüber hinaus führt doppelter Code zu unerwünschtem Overhead und sollten daher durch entsprechende Vorkehrungen vermieden werden. Der letzte Abschnitt 7.2.3 erläutert die Erzeugung des VHDL-Packages, welches abhängig von den benötigten Komponenten unterschiedliche Funktionen, Datentypen und Konstanten umfasst.

7.2.1 Anpassen der Architektur

Eine Möglichkeit der Konfigurierung besteht in der Auswahl der benötigten Architekturen. Da in einem MPSoC unterschiedlich konfigurierte Transaktionsspeicher zum Einsatz kommen können, benötigen die unterschiedlichen Architekturen einer Entity eindeutige Namen. Außerdem muss bei der Instanziierung von Entities die benötigte Architektur ausgewählt werden. Dies kann durch die *<select>*-Anweisung geschehen, die abhängig von den ausgewählten Merkmalen den korrekten Architekturnamen bestimmt. Die Architekturnamen ergeben sich durch den Entity-Namen sowie den beteiligten und ausgewählten Merkmalen. Für die Speicher Familie ergeben sich die folgenden Merkmale:

- für Generierung der Komponenten
  - FIFO-Speicher
    - Schnittstelle
    - Datenbusbreite
    - RAM Type
  - read-modify-write Speicher-Controller
    - Schnittstelle
    - read-modify-write Operation

  - Transaktionsspeicher
    * Schnittstelle
    * Versionierung
    * Konflikterkennung
    * Policy
    * Arbiter
    * RAM Type

- für Instanziierung die im Abschnitt 7.1 erwähnten Parameter

Der FIFO-Speicher ist nur minimal anpassbar durch die Datenbusbreite und den RAM Type, der sich durch das verwendete FPGA ergibt. Die Datenbusbreite ist beim FIFO-Speicher nicht über ein Generic realisiert, da die Block-RAMs für die unterschiedlichen Busbreiten unterschiedliche Templates besitzen und es sich daher anbot diese auf die Top-Entity des FIFO-Speichers zu übertragen. Ähnlich ist der read-modify-write Speicher-Controller nur wenig konfigurierbar. Implementiert ist nur eine Test and Set Operation die über den Wishbone-Bus genutzt werden kann.

In der Top-Entity des Transaktionsspeichers werden alle Komponenten verbunden und entsprechend unter Berücksichtigung der oben angegebenen Merkmalen erzeugt. Allein aus den Merkmalen für Versionierung, Konflikterkennung, Policy und nicht transaktionalen Speicherzugriffen ergeben sich 12 mögliche Konfigurationen. Da die benötigten Komponenten immer gleich sind und damit nur die Architekturen ausgetauscht werden müssen beschränken sich die hier eingesetzten XVCL-Befehle auf einige *<select>*-Anweisungen, um die benötigten Architekturen auszuwählen.

Zusätzlich zum Auswählen passender Architekturen kann die Implementierung einer Basisvariante durch ergänzen von VHDL-Code neue Architekturen hervorbringen, die über mehr Funktionalität verfügen. Der Transaktions-Controller benötigt beispielsweise zum Ausführen von Speicherzugriffen außerhalb von Transaktionen ein wenig mehr Logik als die Basisvariante. Dazu wird die Zustandsmaschine um einen Zustand erweitert. Zusätzlich werden ein paar mehr Informationen benötigt, die durch zusätzliche Ports in der Entity bereitgestellt werden. Da diese Einstellung nur einige kleinere Erweiterungen erfordert, werden die entsprechenden Architekturen bzw. Entities aus einem XVCL-Dokument generiert. Mögliche Fehler in den gemeinsamen Teilen können damit an einem Ort korrigiert werden.

Für die unterschiedlichen Kombinationen aus Versionierung und Konflikterkennung ergeben sich drei Transaktions-Controller, die zwar über die selbe Schnittstelle verfügen, allerdings aufgrund der deutlichen Unterschiede eigenständige Architekturen benötigen. Es lohnt sich dabei nicht die Architekturen aus einer Vorlage zu generieren, da die Zustandsmaschine den größten Teil des VHDL-Codes einnimmt und diese in weiten Teilen deutliche Unterschiede aufweisen. Eine Aufteilung der Zustandsmaschine in gemeinsame und unterschiedliche Teile würde die Lesbarkeit und Wartbarkeit des XVCL-Dokuments dabei deutlich erschweren.

7.2.2 Einmalige Erzeugung der VHDL-Entities

Eine weitere Schwierigkeit besteht darin, dass die erzeugten Entities, Architekturen und Funktionen eindeutige Namen haben müssen. Dies kann erreicht werden, indem zu jeder benötigten Instanz der Speicher Familie sämtliche Komponenten erzeugt und die dort verwendeten Namen um ein eindeutiges Suffix erweitert werden. Damit sind alle Namen eindeutig. Dies hat den Nachteil das potentiell eine Menge Code doppelt erzeugt wird und sowohl die Erzeugung des VHDL-Codes als auch später die Synthese mehr Zeit beanspruchen.

Damit sich der Nutzer der Speicher Familie nicht um dieses Problem kümmern muss, sind die in Abschnitt 7.2.1 angegebenen Merkmale als Multi-Variablen umgesetzt. Dabei bestimmt der i-te Wert die Eigenschaften der i-ten Speicher Familien Instanz. Im ersten Verarbeitungsschritt werden im SPC der Speicher Familie die benötigten Komponenten identifiziert und die dafür zuständigen x-frames eingebunden.

Diese x-frames übernehmen Verwaltungsaufgaben und sorgen dafür, dass die entsprechenden Entities und Architekturen nur einmal generiert werden. Die Multi-Variablen, die die Codeerzeugung beeinflussen, werden dazu in einer Schleife durchlaufen. Es wird aus diesen Variablen ein Fingerabdruck gebildet, um die bereits erzeugten Entities und Architekturen in einer Liste zu speichern. Nur wenn die im aktuellen Schleifendurchgang betrachteten Werte der Variablen noch nicht in dieser Kombination vorgekommen sind, wird die Entity bzw. Architektur erzeugt. Dabei werden Entity und Architektur getrennt betrachtet, so dass zu einer Entity mehrere unterschiedliche Architekturen generiert werden können. Ein weitere x-frame erzeugt abhängig von den Variablen des aktuellen Schleifendurchlaufs den angepassten VHDL-Code.

7.2.3 Generierung des VHDL Package

Im Package werden alle generierten Komponenten-Deklarationen sowie die benötigten Funktionen, Datentypen und Konstanten definiert. Dabei wird zwischen Package Head und Package Body unterschieden. Für das unkomplizierte Instanziieren von Komponenten ist mindestens die Komponenten-Deklaration einer VHDL Entity im Package Head erforderlich, gefolgt von eventuell weiteren Deklarationen von Datentypen und Konstanten. Die Implementierung von Funktionen und Prozeduren müssen im Package Body untergebracht werden. Da durch x-frames lediglich VHDL-Code an bereits bestehende Dateien angehängt werden kann, muss die Generierung des Package durch einen zentralen x-frame erfolgen, der den VHDL-Code für die beiden Bereiche zusammenfasst und ins Package einfügt.

Zu diesem Zweck existiert ein x-frame der den Rahmen des Packages beinhaltet und für Package Head und Body jeweils einen Breakpoint besitzt. Über diese Breakpoints kann für alle Komponenten der benötigte Code eingefügt werden. Dazu müssen alle Komponenten einen dafür zuständigen x-frame in einer Liste registrieren, der den benötigten Code bereitstellt. Da abhängig von den Variablen unterschiedliche Komponenten aus einem x-frame erzeugt werden können, müssen diese auch bei der Generierung der Package-Datei berücksichtigt werden. Hier wird zwischen allgemein benötigtem und an-

gepasstem Code unterschieden. Dabei sorgt der x-frame dafür, dass der allgemeine Teil nur genau einmal pro x-frame eingebunden wird und der angepasste Teil entsprechend genau einmal für die jeweilige Kombination von Werten der benötigten Variablen.

Ähnlich wie in Unterkapitel 7.2.2 beschrieben wird zu jedem x-frame ein Fingerabdruck aus den Werten der Variablen gebildet, um das Generieren von doppeltem Code zu vermeiden. Da für jede Komponente andere Variablen zum Anpassen genutzt werden, existiert zu jedem x-frame eine Liste mit den benötigten Variablen, die den zu generierenden VHDL-Code für die Package-Datei adaptieren.

8 Evaluation

Die implementierte Familie von gemeinsamen Speichern wird in diesem Kapitel einer Evaluierung und Bewertung unterzogen. Dazu betrachtet Unterkapitel 8.1 die Konfigurierbarkeit der Speicher Familie und inwieweit die Identifizierung von Unterschieden und Gemeinsamkeiten die Entwicklung unterstützt hat. Der folgendem Abschnitt 8.2 befasst sich mit der Ergänzung von VHDL durch XVCL. Schließlich untersucht Unterkapitel 8.3 die Implementierung der Speicher Familie und bewertet die Leistung.

8.1 Konfigurierbarkeit der Speicher Familie

In Kapitel 5 sind im Rahmen der Betrachtung der Eigenschaften der unterschiedlichen Speicher Familien Instanzen einige globale Konfigurationsmöglichkeiten ermittelt worden. Abgesehen von der Arbitrierung und der zu verwendenden Schnittstelle handelt es sich um Merkmale, die entweder die Größe eines Busses oder die Anzahl an benötigten Schnittstellen wiedergeben. Diese Eigenschaften haben zwar Einfluss auf den Ressourcenverbrauch des Gesamtsystems, bieten allerdings keine Möglichkeit entwickelte Komponenten wieder zu verwenden. Dieser Umstand ist der einfachen Struktur des FIFO-Speichers als auch des read-modify-write Speicher-Controllers zu zuschreiben.

Die einzige Komponente mit potentiell vielen Variationsmöglichkeiten ist der Transaktionsspeicher. Dabei können die einmal implementierten Entities, wie z. B. die Policy, beliebig mit den übrigen Komponenten kombiniert werden. Dies ist allerdings nur möglich, da zuvor das Kommunikationsschema entworfen worden ist. Dieses Kommunikationsschema birgt allerdings auch den Nachteil, dass für die gewonnene Flexibilität potentiell Leistung eingebüßt wird. Gerade die Freiheit beliebig viel Zeit für die Ausführung einer Aktion in Anspruch nehmen zu können, verzögert i. d. R. die Bearbeitung um mindestens einen Takt. Bei einer bekannten Bearbeitungsdauer ist es möglich direkt nach der festgelegten Zeit weiter zu arbeiten. Wohingegen eine flexible Zeitspanne zum einen zusätzliche Kommunikation benötigt, um das Ende der Bearbeitung zu signalisieren, und zum anderen entsteht durch die nötige Kommunikation Verzögerungen im Ablauf.

Da im Vorfeld nur schwer abzusehen ist welche Informationen zwischen den Komponenten in den unterschiedlichen Konfigurationsmöglichkeiten kommuniziert werden müssen, kann kein allgemeines Kommunikationsschema angegeben werden. Insbesondere da so ggf. viel mehr Informationen transaportiert werden, als in einer bestimmten Konfiguration nötig sind und führt damit zwangsläufig zu unnötigem Ressourcenverbrauch. So sind unter Umständen Modifikationen an dem hier verwendetem Kommunikationsschema notwendig, um beispielsweise eine auf Caches basierende Implementierung umsetzen

zu können.

Die Anzahl an umgesetzten Konfigurationsmöglichkeiten ermöglicht das Generieren von angepassten Speicher-Systemen. Auch wenn die Implementierungen von Transaktionsspeicher, FIFO-Speicher und read-modify-write Speicher-Controller kaum Gemeinsamkeiten aufweisen, ist es gelungen unterschiedliche Speichermodelle für die Nutzung in MPSoCs bereit zu stellen. Damit konnte zwar nur wenige Komponenten innerhalb der Speicher Familie wiederverwendet werden, allerdings bieten die identifizierten Merkmale eine vielfalt an konfigurierbaren und anpassbaren Speicher Instanzen.

8.2 XVCL als Konfigurierungswerkzeug

Wie bereits in Kapitel 7 angedeutet ist VHDL alleine nicht ausreichend, um automatisch aus einer Auswahl an Merkmalen das passende MPSoC zu erzeugen. Sobald unterschiedliche Architekturen für die Umsetzung unterschiedlicher Funktionalität zum Einsatz kommen, wird der VHDL Code unübersichtlich und schwer wartbar, sofern mit Hilfe von Generics die Konfiguration erfolgt. Hier soll XVCL Abhilfe schaffen, indem durch die Werte von Variablen passende Enities generiert werden können. Da XVCL in den Quellcode eingebettet wird, ist es zunächst umständlicher eine benötigte Komponente zu implementieren. Zusätzlich leidet die Lesbarkeit des Codes, sobald an vielen Stellen der VHDL und XVCL Code parallel verwendet wird. Hier gilt es von Fall zu Fall abzuwägen, ob es sicht lohnt die gemeinsamen Teile zweier oder mehr Komponenten zusammenzuführen und die so erhaltene Vorlage mit Hilfe von XVCL zum Generieren der unterschiedlichen VHDL Komponenten zu verwenden. Auf der einen Seite wird es so ermöglicht, nötige Änderungen in den gemeinsamen Teilen des Codes an einem Ort zu erledigen. Auf der anderen Seite ist es aber umständlicher anhand von Compiler-Ausgaben die korrekten Zeilen für die Korrektur von Fehlern zu finden, da durch die XVCL-Befehle die Angaben des Compilers nicht mit den tatsächlichen Zeilennummern übereinstimmen. Verschäft wird diese Problematik, wenn durch Breakpoints Code aus anderen x-frames und damit anderen Dateien eingefügt wird.

Der geringe Befehlsumfang von XVCL ermöglicht es sich schnell in die Sprache einzuarbeiten, jedoch führt dies zu anderen Nachteilen. Die einzige Möglichkeit, abhängig von dem Wert einer Variable, Code zu generieren ist die <select>-Anweisung. Diese kann lediglich eine Variable mit potentiell vielen möglichen Werten vergleichen. Daher muss diese Anweisung ineinander verschachtelt werden, um abhängig von mehreren Variablen Code zu erzeugen. Dies führt wieder dazu, dass der Code unübersichtlicher wird. Weiterhin ist es nicht möglich über einen Befehl zu ermitteln, ob es sich bei einer Variable um eine Multi-Variable oder einer einfachen Variablen handelt. Insbesondere müssen alle Multi-Variablen einer <while>-Schleife die selbe Anzahl an Elementen aufweisen. Deswegen müssen die Multi-Variablen immer die benötigte Anzahl an Werten beinhalten, auch wenn die Werte alle gleich sind. Durch die fehlende Möglichkeit überprüfen zu können, ob es sich um eine Multi-Variable handelt, kann diese auch nicht ggf. von der <while>-Schleife ausgeschlossen werden, falls es sich um eine einfach Variable handelt.

Eine weitere Problematik ergibt sich durch die vererbten Variablen. Diese können in

den eingebundenen x-frames nicht mehr verändert werden. Da es vorkommen kann, dass Variablen nur intern in einem x-frame genutzt werden, um beispielsweise aus den im SPC vorhandenen Variablen alle benötigten Inforamtionen zu extrahieren und für die Generierung aufzubereiten, kann durch die Verwendung des selben Variablennames in zwei x-frames, die Codegenerierung negativ beeinflusst werden. Darüber hinaus ist es nicht möglich Variablen innerhalb eines x-frames anzulegen, die über die Bearbeitungsdauer des entsprechenden x-frames hinaus weiter bestehen. Damit ist es schwierig festzustellen, ob ein x-frame möglichweise zuvor schon einmal mit den selben Parametern eingebunden wurde. Damit kann potentiell Code doppelt erzeugt werden und führt zwangsläufig zu nicht synthetisierbaren VHDL Dateien. Um dies zu vermeiden können die im SPC angegebenen Variablen untersucht werden. Diese enthalten allerdings nicht unbedingt direkt die benötigten Information, sondern müssen über Implementierungsspezifische Informationen der anderen Komponenten gewonnen werden. Eine Möglichkeit dies zu bewerkstelligen wurde in Kapitel 7.2.2 beschrieben.

Da die Umsetzung der Variabilität in den meisten Fällen durch alternative Architekturen ermöglicht wird, hält sich der benötigte Aufwand, der mit XVCL betrieben werden muss, in Grenzen. So wird i. d. R. durch Texterzesetzung nur ein benötigter Architekturname ausgetauscht. Im geringen Umfang ist auch die Erweiterung um Funktionalität von Entities möglich, ohne die Lesbarkeit des Codes zu stark zu beeinträchtigen. Der erzeugte VHDL Code ist dadurch leichter verständlich und ermöglicht eine schnellere Fehlerbeseitigung.

Aus sicht des Nutzers der konfigurierbaren Speicher Familie fehlt allerdings eine explizite Repräsentation der Konfigurationsmöglichkeiten. So ist es notwendig, dass die Variationsmöglichkeiten der x-frames durch entsprechende Kommentare oder anderen Werkzeugen für den Benutzer sichtbar gemacht werden. Insbesondere müssen für die fehlerfreie Bearbeitung der x-frame Hierarchie die Variablen mit korrekten Werten im SPC initialisiert werden, da es sonst zu Fehlern bei der Konfigurierung kommen kann. Andererseits eignet sich die ausschließlich textbasiert Konfigurierung innerhalb von XVCL für die Integration in ein Kette von Entwicklungsprogrammen, so dass die nötigen Eistellungen des SPC automatisch erzeugt werden können und XVCL als Backend dient. Daher wäre eine Ergänzung durch eine GUI wünschenswert, die die möglichen Konfigurationsmöglichkeiten darstellen kann und die entsprechenden Werte der Variablen korrekt ins SPC überträgt, um so Fehler bei der manuellen Eingabe ins SPC durch den Nutzer zu vermeiden.

8.3 Evaluierung der Implementierung

Nachdem die Konfigurierbarkeit der Speicher Familie und XVCL als Konfigurierungswerkzeug bewertet wurde, befasst sich dieser Abschnitte mit der Bewertung der Implementierung. Dazu wird zunächst in Abschnitt 8.3.1 der Ressourcenverbrauch der Speicher Familie näher betrachtet. Anschließend untersucht Abschnitt 8.3.2 einige Laufzeiteigenschaften der Speicher Familie.

Speichergröße	Busbreite	LUTs	Slice Register	maximale MHz
4096kB	8	38	24	111
	16	38	24	111
	32	38	24	111
10240kB	8	38	24	111
	16	38	24	111
	32	38	24	111
20480kB	8	38	24	111
	16	38	24	111
	32	38	24	111

Tabelle 8.1: Ressourcenverbrauch des FIFO-Speichers mit jeweils einem Leser- und Schreiber-Port

8.3.1 Ressourcenverbrauch der Speicher Familie

Für die Bestimmung des Ressourcenverbrauchs wurden die unterschiedlichen konfigurierten System für ein Virtex 5 FPGA synthetisiert. Da die zugrunde liegende Technologie zwischen verschiedenen FPGAs unterschiedlich sein kann, ist es nicht möglich die Werte der Synthese für unterschiedliche FPGAs miteinander zu vergleichen. Ein Virtex FPGA hat bespielsweise sechs Eingangssignale pro LUT, statt vier wie bei einem Spartan. Dies sorgt für deutliche Unterschiede in den benötigten LUTs eines Systeme für unterschiedliche FPGAs.

Sowohl der FIFO-Speicher als auch der read-modify-write Speicher-Controller haben aufgrund ihrer einfachen Struktur nur einen geringen Ressourcenverbrauch. Beide Speicher-Controller sind mit unterschiedlichen Konfigurationen für Datenbusbreite und Speichergröße synthetisiert worden. Die Ergebnisse sind in Abhängigkeit von Speichergröße und Datenbusbreite bei 2 angeschlossenen Prozessoren in den Tabellen 8.1 und 8.2 zu sehen. Keiner der Speicher-Controller hat dabei die maximal mögliche Frequenz, mit der das MPSoC betrieben werden kann, beeinflusst. Weiterhin fällt auf, dass für alle konfigurierten Werte der Ressourcenverbrauch gleich bleibt. Damit erfüllen diese Einstellungsmöglichkeiten nicht den gewünschten Effekt, den Ressourcenverbrauch anpassen zu können. Optimierungen im Zuge der Synthese könnten für diesen Effekt gesorgt haben, so dass Eigenschaften der FPGAs besser ausgenutzt werden. So verfügt jedes CLB über 256 Bit distributed RAM, welches für die verwendeten Register eingesetzt wird. Daher ergeben sich möglicherweise keine Unterschiede, wenn statt eines 32 Bit Registers nur ein 16 oder 8 Bit Register pro CLB synthetisiert wird.

Der Transaktionsspeicher weist, wegen seiner vielen Konfigurationsmöglichkeiten, deutliche Unterschiede im Ressourcenverbrauch auf. In Abbildung 8.1 ist der Ressourcenverbrauch des gesamten Transaktionsspeichers für unterschiedlichen Kombinationen aus Konflikterkennung, Versionierung und Anzahl an angeschlossenen Prozessoren dargestellt. Alle Systeme wurden mit der Oldest Policy ausgestattet. Da die Attacker-Wins Policy nur für ein lazy optimistic System garantierten Fortschritt bietet, ist diese Policy ausschließlich mit der lazy optimistic Konfiguration getestet worden. Alle Systeme

Speichergröße	Busbreite	LUTs	Slice Register	maximale MHz
4096kB	8	132	76	111
	16	132	76	111
	32	132	76	111
20480kB	8	132	76	111
	16	132	76	111
	32	132	76	111
40960kB	8	132	76	111
	16	132	76	111
	32	132	76	111

Tabelle 8.2: Ressourcenverbrauch des read-modify-write Speicher-Controllers mit zwei Anschlüssen

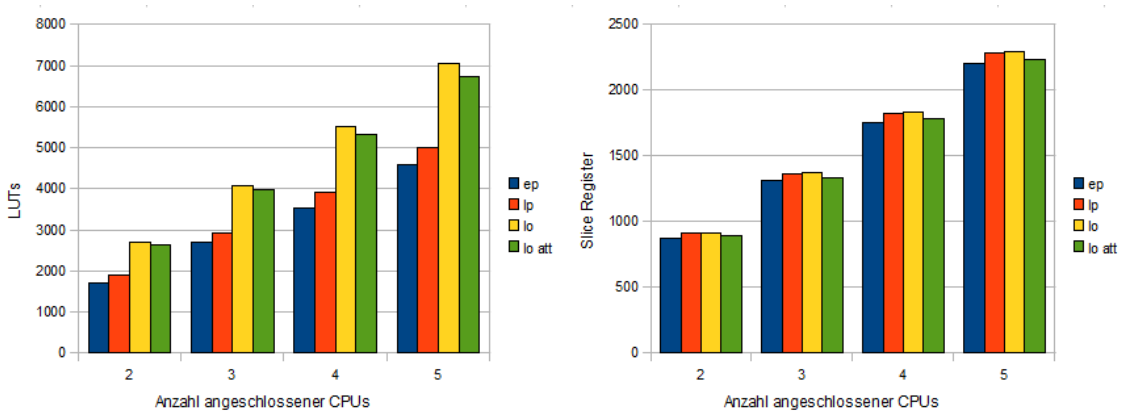

Abbildung 8.1: Ressourcenverbrauch des Transaktionsspeichers, abhängig von der gewählten Konflikterkennung, Versionierung und Anzahl an angeschlossenen Prozessoren. (e = eager, l = lazy, p = pessimistisch, o = optimistisch).

wurden für eine Speichergröße von 320kB ausgelegt. Das rechte Diagram zeigt die benötigten Slice Register, wobei die Anzahl benötigter Register bei gleicher Anzahl an angeschlossenen Prozessoren sehr ähnlich ist. Dies lässt darauf schließen, dass die nötigen Informationen, die in den Register gespeichert werden müssen, bei allen Systemen ähnlich umfangreich sind. Im Gegensatz dazu verbrauchen die Systeme mit pessimistischer Konflikterkennung bis zu 37% weniger LUTs als die optimistische Variante. Dies liegt daran, dass bei einer optimistischen Konflikterkennung nicht der gemeinsame Bus für schreibende Speicheroperationen genutzt werden soll. Daher muss für jeden Transaktions-Controller eine eigene Implementierung der Hashfunktion existieren, um den Hashwert der Adress zu berechnen und der Signatur hinzuzufügen.

In Abbildung 8.3.2 ist der relative Verbrauch an LUTs der einzelnen Komponenten des Transaktionsspeichers abgebildet. Eine genaue Auflistung der benötigten LUTs und Slice Register aller Komponenten für den eager pessimistic Transaktionsspeicher zeigt

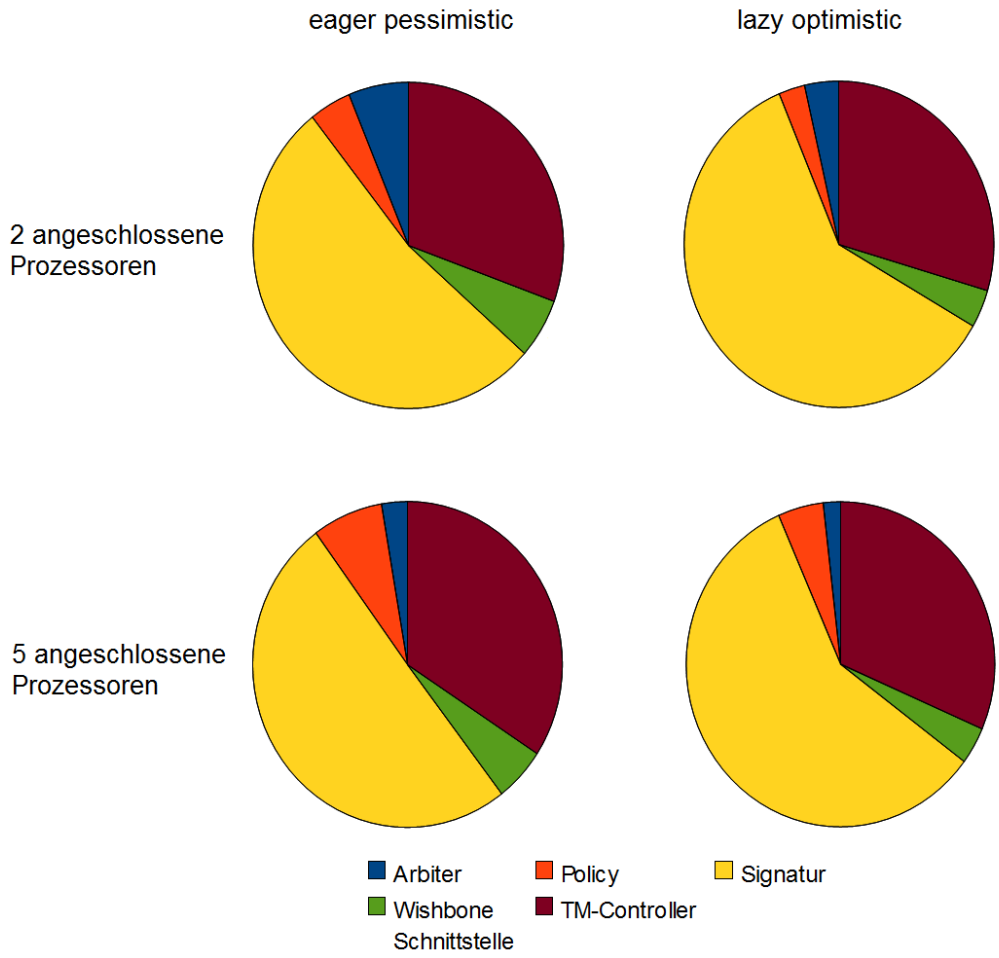

Abbildung 8.2: Benötigte LUTs der Komponenten im Verhältnis zum Gesamtbedarf.

die Tabelle 8.3. Der größte Teil der LUTs und Slice Register wird von der Signatur beansprucht. Die hier verwendete Konfiguration hat den Adressbus in Blöcke zu 6 Bit aufgeteilt, dabei ergibt sich für einen 320kB großen Speicher und 32 Bit Datenbus ein Adressbus von 17 Bit. Eine Signatur benötigt dann 160 Bit, so dass pro Transaktions-Controller 320 Bit für die beiden Signaturen benötigt werden. 640 von den 648 Slice Registern entfallen dann wahrscheinlich auf die Signatur und die nötigen Multiplexer für die Hashfunktion ergeben die restlichen benötigten LUTs. Da das Signatur-Modul als eine Entity realisiert ist, kann hier keine genaue Aufteilung pro angeschlossenen Prozessor gegeben werden.

Den zweitgrößte Anteil nehmen die Transaktions-Controller ein. Im Vergleich zu den übrigen Komponenten ist die dort verwendete Zustandsmaschine mit über 10 Zuständen die größte. Es sind bei einer lazy Versionierung einige zusätzliche Zustände nötig, um im Falle eines lesenden Speicherzugriffs ggf. die passenden Daten aus dem Zwischenspeicher laden zu können. Dies erklärt den Unterschied von bis zu 40% an benötigten LUTs zwischen eager pessimistic und lazy optimistic Variante. Die Policy und der Arbiter kommen

CPUs	Policy	Arbiter	Signatur	TM-Controller 1-5					Wishbone Schnittstelle 1-5				
eager - pessimistic - oldest													
2	75	107	906	264	260	-	-	-	51	51	-	-	-
3	157	148	1357	254	307	303	-	-	55	54	54	-	-
4	225	187	1786	260	307	307	256	-	54	53	53	52	-
5	339	124	2313	325	323	321	299	304	50	50	49	49	49
lazy - pessimistic - oldest													
2	73	97	902	356	358	-	-	-	52	52	-	-	-
3	163	117	1357	348	395	392	-	-	51	51	52	-	-
4	232	171	1787	347	406	393	363	-	51	50	52	50	-
5	339	125	2315	397	396	396	395	389	49	49	48	48	47
lazy - optimistic - oldest													
2	74	97	1634	400	401	-	-	-	52	50	-	-	-
3	159	114	2409	394	431	424	-	-	48	48	48	-	-
4	222	122	3226	437	437	437	431	-	53	53	53	52	-
5	336	127	4106	451	434	441	445	450	52	52	54	53	50
lazy - optimistic - attacker wins													
2	15	96	1606	399	397	-	-	-	53	53	-	-	-
3	22	129	2409	394	428	425	-	-	54	55	54	-	-
4	24	125	3226	436	437	432	430	-	54	54	54	54	-
5	33	126	4106	449	431	438	441	447	53	53	55	55	51

Tabelle 8.3: Ressourcenverbrauch der einzelnen Komponenten des Transaktionsspeichers abhängig von Versionierung, Konflikterkennung und Anzahl an angeschlossenen CPUs

unabhängig von der Anzahl der angeschlossenen CPUs nur einmal im Transaktionsspeicher vor und trotzdem bleibt der relative Anteil an benötigten Ressourcen nahezu gleich. Weiterhin fällt auf, dass die benutzten LUTs zwischen den Transaktions-Controllern um bis zu 20% schwankt. Dies könnte durch Komponenten übergreifende Optimierungen erklärt werden.

8.3.2 Auswertung der Laufzeiteigenschaften der Speicher Familie

Für die Beurteilung der Leistung werden im Folgenden einige Simulationsergebnisse präsentiert, die die Bearbeitungsdauer einiger Speicherzugriffe wiedergeben. Die eingesetzten MPSoCs umfassen dabei den MBLite Prozessoren und Block-RAMs als gemeinsamen Speicher. Abbildung 8.3 zeigt den Ablauf eines Speicherzugriffes auf den FIFO-Speicher. Dargestellt sind einige Signale der Wishbone Schnittstelle, die zum nachvollziehen des Schreib- bzw. Lesezyklus ausreichend sind. Der lesende Zugriff ist unten zu sehen und beginnt mit der ersten steigenden Flake des Taktes. Da zu diesem Zeitpunkt der Speicher noch leer ist, muss der Prozessor warten. Mit der dritten steigenden Taktflanke startet der Schreibzugriff. Die Block-RAMs können die gewünschten Speicheroperation

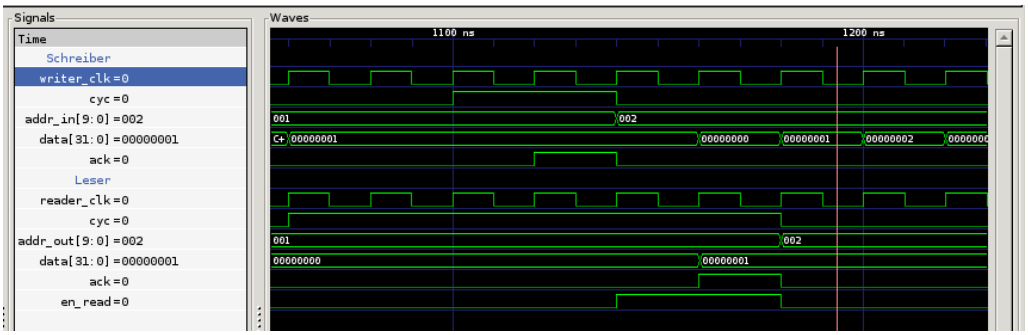

Abbildung 8.3: Waveform eines parallelen Speicherzugriffs durch den lesenden und den schreibenden Prozessor auf den FIFO-Speicher

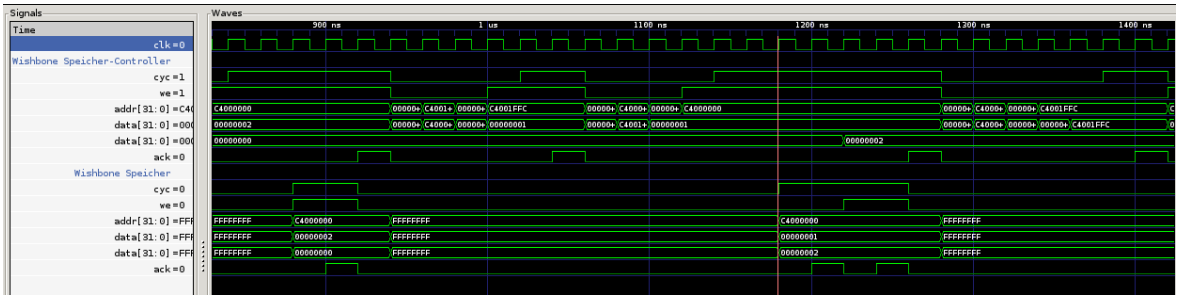

Abbildung 8.4: Waveform der Ausführung einer Test and Set Operation

innerhalb eines Taktes durchführen, da diese allerdings nur synchron mit der steigenden Flanke des Taktes arbeiten, benötigt der Schreibzugriff insgesamt zwei Takte. Entsprechendes gilt für den Lesezugriff, der bereits zwei Takte nach dem Abschluss der Schreiboperation den neuen Wert bereitstellt. Damit kann ein theoretischer Datendurchsatz von

$$\frac{Taktrate * Datenbusbreite}{2 * 8 * 1024} \frac{kB}{Sekunde} \tag{8.1}$$

erreicht werden.

Der read-modify-write Speicher-Controller kann diese Geschwindigkeit nicht erreichen, da durch die Arbitrierung des gemeinsamen Busses eine Verzögerung auftritt. Die Ausführung einer Test and Set Operation ist in Abbildung 8.4 dargestellt. Oben ist die Wishbone Schnittstelle zur CPU zu sehen und unten die zum Speicher. Der erste Speicherzugriff ist ein normaler schreibender Zugriff, der fünf Takte inanspruch nimmt. Dabei entfällt zwei Takte auf die Arbitrierung, weitere zwei für den eigentlichen Speicherzugriff und ein Takt, um das Ende der Speicheroperation an den Prozessor weiterzugeben. Der darauf folgende Speicherzugriff vermerkt im Speicher-Controller lediglich den Wunsch für eine Test und Set Operation, so dass dieser nach zwei Takten abgeschlossen ist. Die eigentliche Test and Set Operation benötigt genau zwei Takte mehr, als ein gewöhnlicher Speicherzugriff.

Im Falle des Transaktionsspeichers ergeben sich sehr unterschiedliche Zugriffszeiten. Diese sind in Tabelle 8.4 zusammen gefasst. Dabei werden, ähnlich wie zuvor auch, jeweils

	Lesezugriff	Speicherzugriff	Commit	Revert
EP	6	9	2	abhängig vom Write-Set
LP	abhängig vom Write-Set	6	abhängig vom Write-Set	2
LO	abhängig vom Write-Set	3	abhängig vom Write-Set	2

Tabelle 8.4: Speicherzugriffszeiten in Takten, abhängig von Versionierung und Konflikterkennung

zwei Takte für eine Speicheroperation, eine Überprüfung einer Adresse auf Konflikte und für die Arbitrierung zugrunde gelegt. Die Abarbeitung der unterschiedlichen Zustände und der Einsatz des Kommunikationsschemas sorgen dabei für zusätzliche Verzögerungen. Abhängig von der gewählten Versionierung kann nicht für jede Speicheroperation eine genaue Taktanzahl angegeben werden. Im Falle eines eager pessimistic Systems kann ein Revert direkt nach erhalt des Busses gestartet werden. Bei zwei Takten pro Speicherzugriff und einem weiteren Takt, um das nächste Datum aus dem Zwischenspeicher zu laden, benötigt der Revert das dreifache an Takten als Daten im Zwischenspeicher gespeichert sind. Ähnlich verhält es sich bei einem Commit in einem lazy pessimistic System, bei dem auch direkt nach erhalt des Busses die Daten in den Speicher übertragen werden können. Der Commit eines lazy optimistic Transaktionsspeichers benötigt mehr Zeit, da zunächst das gesamte Write-Set überprüft werden muss. Die Überprüfung benötigt insgesamt drei Takte pro Adresse. Zusammen mit dem übertragen der Änderungen in den Speicher macht es sechs Takte pro Datum im Zwischenspeicher.

Weiterhin ist die Dauer eines lesenden Speicherzugriffes, bei einem Transaktions-Controller mit lazy Versionierung, abhängig von der Größe des Write-Sets. Während einer Transaktion kann es vorkommen, dass zunächst ein Datum geschrieben und zu einem späteren Zeitpunkt nochmals gelesen werden muss. Um bei lesenden Zugriffen den zuletzt geschriebenen Wert zu erhalten, ist es erforderlich den Zwischenspeicher zu durch suchen. Dies führt zu potentiell deutlich längeren Zugriffszeiten als bei einer eager Versionierung. Insgesamt ist die entwickelte Speicher Familie auf die Umsetzung der Funktionalität ausgerichtet, so dass sich durch Optimierungen von Komponenten sowohl der Ressourcenverbrauch als auch die Latenz der Speicherzugriffe verringert lassen sollte.

9 Zusammenfassung und Ausblick

Zum Abschluss fasst das Unterkapitel 9.1 die Ergebnisse dieser Diplomarbeit zusammen. Anschließend werden in Abschnitt 9.2 Ideen und Anregungen für Erweiterungen und Einsatzmöglichkeiten der Speicher Familie gegeben.

9.1 Zusammenfassung

Im Vordergrund dieser Diplomarbeit stand die Entwicklung einer Familie von gemeinsamen Speichern. Dazu wurden in Kapitel 5 drei unterschiedliche Speichermodelle betrachtet. Dabei sind die Gemeinsamkeiten und Unterschiede dieser Modelle herausgearbeitet worden. Die so ermittelten Eigenschaften ermöglichen es möglichst früh im Entwicklungsprozess die Konfigurierbarkeit der Komponenten zu erkennen. Die Identifizierung der Gemeinsamkeiten unterstützt weiterhin die Entwicklung. Komponenten, die an verschiedenen Stellen in sehr ähnlicher Form auftretten, müssen so ggf. nicht mehrfach Entworfen und an die spezifischen Gegebenheiten angepasst werden. So kommt sowohl die Schnittstelle zu den Prozessoren als auch ein Arbiter für die Zugriffssteuerung gemeinsam benutzter Ressourcen in allen Speicher Familien vor. Das erste Speichermodell wird durch den FIFO-Speicher realisiert und dient als einfacher Puffer zwischen einem Produzenten und einem Konsumenten. Für die klassische Betrachtung von gemeinsamen Speicher, die mit Hilfe von atomaren read-modify-write Operationen die Synchronisation über Locks oder Semaphore zwischen Prozessoren sicherstellen, ist ein Speicher-Controller notwendig. Für diesen Speicher-Controller stehen unterschiedliche Formen von read-modify-write Operationen zur Verfügung, um die Synchronisation gewährleisten zu können. Das dritte Speichermodell ist ein Transaktionsspeicher, der automatisch Konflikte zwischen parallel laufenden Transaktionen feststellt und auflöst. Dabei sind eine Reihe unterschiedlicher Strategien für Versionierung, Konflikterkkenung und Konfliktbeseitiung möglich. Zur Visualisierung der Konfigurationsmöglichkeiten dienen Merkmaldiagramme.

In Kapitel 6 wurde eine Auswahl der zuvor identifizierten Merkmale implementiert. Für eine einfache Integration in die konfigurerbare LavA-Plattform wurden alle Implementierungen als Blackbox aufgefasst, die über den on-Chip Bus Standard Wishbone mit dem übrigen Komponenten des MPSoCs kommunizieren. Für die Arbitrierung des gemeinsamen Busses ist ein Prioritäten Arbiter umgesetzt worden, der sowohl im Transaktionsspeicher als auch im read-modify-write Speicher-Controller zum Einsatz kommt. Die meisten Konfigurierungsmöglichkeiten umfasst der Transaktionsspeicher. Die unterschiedlichen Kombinationen aus Versionierung, Konflikterkennung und Policy können dabei über unterschiedliche Architekturen beliebig miteinander kombiniert werden.

Dazu musste für die Umsetzung der Konfigurierbarkeit in Kapitel 7 ermittelt werden, welche Merkmale sich direkt mit Mitteln der Sprache VHDL umsetzen lassen. Da VHDL alleine nicht ausreichend ist, um konfigurierbaren und gleichzeitig verständlichen Code zu erzeugen, wurde XVCL als Ergänzung hinzugezogen. Insbesondere lassen sich mit Hilfe von XVCL die benötigten Komponenten automatisch anhand von einigen Variablen generieren. Dabei gilt es zu beachten, dass potentiell mehrere Instanzen einer Speicher Familien Instanz im MPSoC eingesetzt werden können und daher darauf zu achten ist, dass Komponenten die mehrfach vorkommen nicht doppelt generiert werden. Weiterhin wurde eine Technik entwickelt, um die Package-Datei automatisch mit allen benötigten Komponenten Deklaration, Datentypen und Funktionen zu erzeugen.

Abschließend wurden in Kapitel 8 die Speicher Familie einer Evaluierung unterzogen. Dabei wurde zunächst die Konfigurierbarkeit der Speicher Familie betrachtet. Die Ergänzung durch XVCL als Konfigurationswerkzeug wurde anschließend untersucht. XVCL erhöht dabei den Aufwand der Entwicklung durch die parallele Verwendung von VHDL-Code und XVCL-Befehlen. Es wurde festgestellt, dass weitere Werkzeuge zur Repräsentation des Konfigurationsraums und automatische Erzeugung der in XVCL benötigten Variablen die Konfigurierung eines Systems erleichtern würde. Die Implementierung wurde schließlich einer Bewertung bezüglich Ressourcenverbrauch und Zugriffsgeschwindikeit unterzogen. Dabei stellte sich heraus, dass einige Merkmale keinen Einfluss auf den Ressourcenverbrauch haben und daher ihren Zweck nicht erfüllen.

9.2 Ausblick

Die in dieser Diplomarbeit implementierte Speicher Familie könnte in weiterführenden Arbeiten für Vergleiche zwischen den unterschiedlichen Konfigurationsmöglichkeiten des Transaktionsspeicher eingesetzt werden. Dadurch könnten möglicherweise Eigenschaften von Programmen abgeleitet werden, die für die automatische Generierung einer bestimmten Konfiguration eingesetzt werden können, um so eine optimale Ausnutzung der Ressourcen zu erhalten. Weiterhin könnte der Einsatz von Transaktionen gegenüber klassischen Lösungen zur Synchronisation untersucht werden.

Literaturverzeichnis

[1] Philips (Hrsg.): *PNX8526 User Manual*. Philips, 2003. – `http://www.nxp.com/documents/user_manual/UM10104.pdf`

[2] Böckle, Günter: *Software-Produktlinien. Methoden, Einführung und Praxis*. 1. Aufl. Heidelberg : dpunkt-Verl., 2004. – ISBN 3–89864–257–7

[3] *LavA: Laufzeitplattform für anwendungsspezifische verteilte Architekturen*. – Forschungsprojekt an der TU Dortmund, Lehrstuhl 12, Arbeitsgruppe Eingebettete System Software `http://ess.cs.uni-dortmund.de/DE/Research/Projects/LavA/`

[4] Spinczyk, Olaf ; Lohmann, Daniel: The Design and Implementation of AspectC++. In: *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software* 20 (2007), Nr. 7, S. 636–651

[5] Lohmann, Daniel ; Hofer, Wanja ; Schröder-Preikschat, Wolfgang ; Streicher, Jochen ; Spinczyk, Olaf: CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems. In: *Proceedings of the 2009 USENIX Annual Technical Conference*. Berkeley, CA, USA : USENIX Association, Juni 2009, S. 215–228

[6] Jarzabek, Stan ; Bassett, Paul ; Zhang, Hongyu ; Zhang, Weishan: XVCL: XML-based Variant Configuration Language. In: *International Conference on Software Engineering* (2003), S. 810. – ISSN 0270–5257

[7] 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*. New York, NY, USA : ACM, 2002. – ISBN 1–58113–556–4, S. 341–349

[8] Singapore, National U. ; Inc., Netron: *XML-based Variant Configuration Language (XVCL) - Specification Version 2.10*, Juni 2006. – `http://xvcl.comp.nus.edu.sg/`

[9] Meier, Matthias: *Merkmalbasierte statische Konfigurierung von MPSoCs*, Technische Universität Dortmund, Diplomarbeit, Mai 2009

[10] Michael Engel, Matthias M. ; Spinczyk, Olaf: *LavA: An Embedded Operating System for the Manycore Age*. Oktober 2009. – Poser Session, SOSP; `http://ess.`

cs.uni-dortmund.de/Research/Projects/LavA/pres/lava_sosp09.pdf (abgerufen am 30.08.2010)

[11] Eswaran, Kapali P. ; Gray, Jim ; Lorie, Raymond A. ; Traiger, Irving L.: The Notions of Consistency and Predicate Locks in a Database System. In: *Commun. ACM* 19 (1976), Nr. 11, 624-633. http://dblp.uni-trier.de/db/journals/cacm/cacm19.html#EswarranGLT76

[12] Herlihy, Maurice ; Moss, J. Eliot B.: Transactional Memory: Architectural Support for Lock-Free Data Structures. In: *SIGARCH Comput. Archit. News* 21 (1993), Nr. 2, S. 289–300. – ISSN 0163–5964

[13] Rajwar, Ravi ; Goodman, James R.: Transactional lock-free execution of lock-based programs. In: *ASPLOS*, 2002, 5-17

[14] Ananian, C. S. ; Asanovic, Krste ; Kuszmaul, Bradley C. ; Leiserson, Charles E. ; Lie, Sean: Unbounded Transactional Memory. In: *HPCA*, IEEE Computer Society, 2005. – ISBN 0–7695–2275–0, 316-327

[15] Rajwar, Ravi ; Herlihy, Maurice ; Lai, Konrad K.: Virtualizing Transactional Memory. In: *ISCA*, IEEE Computer Society, 2005, 494-505

[16] Moore, Kevin E. ; Bobba, Jayaram ; Moravan, Michelle J. ; Hill, Mark D. ; Wood, David A.: LogTM: log-based transactional memory. In: *HPCA*, IEEE Computer Society, 2006, 254-265

[17] Bobba, Jayaram ; Moore, Kevin E. ; Volos, Haris ; Yen, Luke ; Hill, Mark D. ; Swift, Michael M. ; Wood, David A.: Performance pathologies in hardware transactional memory. In: Tullsen, Dean M. (Hrsg.) ; Calder, Brad (Hrsg.): *ISCA*, ACM, 2007. – ISBN 978–1–59593–706–3, 81-91

[18] Shriraman, Arrvindh ; Dwarkadas, Sandhya ; Scott, Michael L.: Flexible Decoupled Transactional Memory Support. In: *ISCA*, IEEE, 2008, 139-150

[19] McDonald, Austen: *Architectures for Transactional Memory*, Stanford University, Diss., June 2009

[20] Herlihy, Maurice ; Luchangco, Victor ; Moir, Mark ; III, William N. S.: Software transactional memory for dynamic-sized data structures. In: *PODC*, 2003, 92-101

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

[22] Xilinx, Inc.: *MicroBlaze Development Kit Spartan-3E 1600E Edition - User Guide, UG257 (v1.1)*, Dezember 2007

[23] *MBLite*. – http://opencores.org/project,mblite (abgerufen am 30.8.2010)

[24] Xilinx, Inc.: *ML505/ML506/ML507 Evaluation Platform - User Guid, UG347 (v3.1.1)*, Oktober 2009

[25] Gingol, Tristan: *GHDL guide.* – `http://ghdl.free.fr/ghdl/index.html` (abgerufen am 30.8.2010)

[26] Bassett, Paul G.: *Framing software reuse: lessons from the real world.* Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1997. – ISBN 0–13–327859–X

[27] 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, 2006, S. 1966 –1968

[28] Lukovic, Slobodan ; Fiorin, Leandro: An Automated Design Flow for NoC-based MPSoCs on FPGA. In: *IEEE International Workshop on Rapid System Prototyping*, IEEE Computer Society, 2008. – ISBN 978–0–7695–3180–9, 58-64

[29] McDonald, Austen ; Chung, JaeWoong ; Chafi, Hassan ; Cao Minh, Chi ; Carlstrom, Brian D. ; Hammond, Lance ; Kozyrakis, Christos ; Olukotun, Kunle: Characterization of TCC on Chip-Multiprocessors. In: *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques.* 2005

[30] ARM, Ltd.: *AMBA Specification Revision 2.0*, Mai 1990

[31] Culler, David E. ; Singh, J.P.: *Parallel Computer Architecture: A Hardware/-Software Approach.* Morgan Kaufmann Publishers, Inc., 1999

[32] IBM Corp: *The CoreConnect Bus Architecture.* 1999. – `https://www-01.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect_Bus_Architecture` (abgerufen am 30.08.2010)

[33] OpenCores: *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, 2010. – `http://opencores.org/opencores,wishbone` (abgerufen am 30.08.2010)

[34] Kruskal, Clyde P. ; Rudolph, Larry ; Snir, Marc: Efficient synchronization of multiprocessors with shared memory. In: *ACM Trans. Program. Lang. Syst.* 10 (1988), Nr. 4, S. 579–601. – ISSN 0164–0925

[35] Agrawal, Kunal ; Leiserson, Charles E. ; Sukha, Jim: Memory models for open-nested transactions. In: Hosking, Antony L. (Hrsg.) ; Adl-Tabatabai, Ali-Reza (Hrsg.): *Memory System Performance and Correctness*, ACM, 2006. – ISBN 1–59593–578–9, 70-81

[36] Bloom, Burton H.: Space/time trade-offs in hash coding with allowable errors. In: *Commun. ACM* 13 (1970), Nr. 7, S. 422–426. – ISSN 0001–0782

Abbildungsverzeichnis