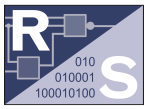
Automatisches Bauen und Parallelisieren von Microblaze Systemen

Masterarbeit

Christian Hohenbrink

4. Januar 2017

TECHNISCHE UNIVERSITÄT DARMSTADT



Erklärung gemäß § 22 Abs. 7 APB

Hiermit erkläre ich gemäß § 22 Abs. 7 der Allgemeinen Prüfungsbestimmungen (APB) der Technischen Universität Darmstadt in der Fassung der 4. Novelle vom 18. Juli 2012, dass ich die Arbeit selbstständig verfasst und alle genutzten Quellen angegeben habe und bestätige die Übereinstimmung von schriftlicher und elektronischer Fassung.

Darmstadt, den 4. Januar 2017	
Ort, Datum	Name

Fachbereich Elektro- und Informationstechnik
Institut für Datentechnik
Fachgebiet Rechnersysteme
Prüfer: Prof. Dr.-Ing. Christian Hochberger
Betreuer: M.Sc. Kris Heid

Inhaltsverzeichnis

1 Einleitung **4**
- 1.1 Einführung in die Thematik . . . 4
- 1.2 Ziele der Arbeit . . . 4
- 1.3 Gliederung der Arbeit . . . 5

2 Grundlagen **6**
- 2.1 SpartanMC Entwicklungsumgebung . . . 6
  - 2.1.1 JConfig . . . 6
  - 2.1.2 SpartanMC Toolchain . . . 6
  - 2.1.3 XML-Modulbeschreibung . . . 7
- 2.2 Xilinx Embedded Development Kit . . . 7
  - 2.2.1 Microblaze . . . 7
  - 2.2.2 XPS . . . 9
  - 2.2.3 SDK . . . 9
  - 2.2.4 FSL . . . 9
  - 2.2.5 Speicherinitialisierung im Microblaze . . . 10

3 Konzeption und Implementation **12**
- 3.1 Anforderungen . . . 12
- 3.2 Integration in JConfig . . . 12
  - 3.2.1 Hardware . . . 12
  - 3.2.2 Hardwarebeschreibung . . . 13
  - 3.2.3 Modulbeschreibungen . . . 18
  - 3.2.4 Busbeschreibungen . . . 19
- 3.3 Integration in die SpartanMC Toolchain . . . 20
  - 3.3.1 Ausgangslage . . . 20
  - 3.3.2 Erzeugung der BMM-Datei . . . 20
  - 3.3.3 Anpassungen in der Toolchain zur Speicherinitialisierung . . . 21
  - 3.3.4 Unterstützung von Libraries in der Simulation . . . 22
  - 3.3.5 Integration des Microblaze GCC . . . 23

4 Benutzungshinweise **24**

5 Evaluation **27**
- 5.1 Vorgehensweise . . . 27
- 5.2 Erstellen des Testsystems mit JConfig . . . 27
- 5.3 Verwendung der Toolchain . . . 29
- 5.4 Erstellen der Testsoftware und Erzeugung der Speicherinitialisierung für die Simulation . . . 30
- 5.5 Simulationsergebnisse . . . 32
  - 5.5.1 Lesen von Instruktionen aus dem Speicher . . . 32

5.5.2 Kommunikation über das FSL-Interface 32
5.5.3 Versenden von Daten über den UART-Core 32
5.6 SpartanMC- und Hybrid-Systeme 34

6 Fazit und Ausblick **36**
6.1 Fazit 36
6.2 Ausblick 36

1 Einleitung

1.1 Einführung in die Thematik

Am Fachgebiet Rechnersysteme der TU Darmstadt und der Professsur für eingebettete Systeme der TU Dresden wurde der CPU-Core SpartanMC entwickelt, welcher mit einer Daten- und Instruktionsbreite von 18 Bit ideal für die Verwendung in FPGAs geeignet ist. Ebenso wurde eine Toolchain entwickelt, die es dem Nutzer ermöglicht, über die grafische Oberfläche JConfig Systeme zu beschreiben und im Anschluss über Make-Regeln weitere Schritte wie Synthese und Simulation durchzuführen. [The]
Der SpartanMC wird am Fachgebiet Rechnersysteme zur Forschung an Manycore SoC verwendet. Eine dieser Forschungsarbeiten stellt das Programm μStreams dar, welches auf Basis des Source-to-Source Compilers Cetus [CET] entwickelt wurde, um die Leistungsfähigkeit von Manycore Systemen zu steigern. Hierzu kann der Programmierer den Sourcecode mit Annotationen versehen, die den Code in verschiedene Aufgabenblöcke unterteilt. Dabei ergibt sich eine Pipeline-Struktur, in der die Ergebnisse einzelener Berechnungen über Core-Konnektoren weitergegeben werden. μStreams kann sowohl eine Hardwarebeschreibung für ein Manycore System generieren, als auch Software für die einzelenen Cores, welche mit der SpartanMC Toolchain konform sind. [Web15]

1.2 Ziele der Arbeit

Um zu zeigen, dass diese Art der Perfomancesteigerung nicht nur auf den SpartanMC begrenzt ist, ist das primäre Ziel dieser Arbeit, den Xilinx Microblaze in die bestehende SpartanMC Toolchain zu integerieren.
Hierzu soll die Konfiguration des Microblaze über den SpartanMC Systembuilder JConfig ermöglicht werden. Ebenso ist es erforderlich, für einfache I/O-Anwendungen einen UART IP-Core für den Microblaze zu integrieren. Als Pendant zu den Core-Konnektoren soll es ebenfalls möglich sein, Fast Simplex Link (FSL) Blöcke von Xilinx dem System hinzuzufügen, um die Kommunikation zwischen den einzelnen Prozessoren zu ermöglichen.
Sollte es für die Integration notwendig sein, ist die Toolchain ebenfalls anzupassen, um die Generierung von lauffähigen Microblaze Systemen zu ermöglichen. Hierbei ist darauf zu achten, dass die Funktionalität sowohl für Systeme, die nur aus SpartanMC oder Microblaze bestehen, als auch für Systeme, die beide Prozessoren beinhalten, nicht negativ beeinflusst wird.

Optional wäre es wünschenswert, die erfolgreiche Paralellisierung eines Microblaze Programms durch μStreams zu zeigen. Hierzu wäre es notwendig, μStreams soweit anzupassen, dass eine entsprechende Hardwarekonfiguration mit den neu integrierten Komponenten erstellt werden kann und die Aufrufe der Core-Konnektoren durch Aufrufe der FSL-Blöcke ersetzt werden.

1.3 Gliederung der Arbeit

Im folgenden Kapitel werden die Grundlagen für diese Arbeit beschrieben. Diese umfasst neben der Beschreibung der verwendeten Software-Tools, ebenfalls Erläuterungen zum Aufbau der SpartanMC Toolchain, sowie Erklärungen zu verwendeten Dateiformaten und Busbeschreibungen. Anschließend werden Konzept und Realisierung mit allen notwendigen Schritten für die Integration des Microblaze in die SpartanMC Toolchain erarbeitet. In einem weiteren Abschnitt wird auf alle notwendigen Schritte eingegangen, um ein lauffähiges System zu generieren. Darauf folgend wird eine Evaluation zum Grad der erreichten Integration durchgeführt. Abschließend folgt eine kurze Zusammenfassung, sowie ein Ausblick auf mögliche Fortsetzungen der Arbeit.

2 Grundlagen

2.1 SpartanMC Entwicklungsumgebung

2.1.1 JConfig

JConfig ist der in Java programmierte Systembuilder der SpartanMC Entwicklungsumgebung. Mit ihm ist es möglich, auf abstrakter Ebene ein System-on-Chip (SoC) zu beschreiben, welches für die Verwendung auf FPGAs bestimmt ist.
Bei der Erstellung einer neuen Konfiguration wird zunächst die Zielhardware festgelegt. JConfig generiert aus dieser Angabe automatisch alle erforderlichen Optionen, die später von der ISE Toolchain benötigt werden, um ein Bitfile zu generieren. Desweiteren können sämtliche Inputs und Outputs des SoC verwaltet werden (Einstellungen wie Spannungsniveau und Invertierung können vorgenommen werden). Um ein SoC zu erstellen, können Hardwarekomponenten aus einer erweiterbaren Library ausgewählt werden. Ein IP-Core der Library hinzuzufügt werden, indem zunächst Hardware Description Language (HDL) Dateien und eine Beschreibung des Cores in einem bestimmten Extensible Markup Language (XML) Format hinterlegt werden (siehe 2.1.3).
Für jede Hardwarekomponente lassen sich sowohl Verbindungen zu anderen Komponenten, als auch Parameter definieren. Speicherbausteine haben mit der Angabe eines Pfades zur verwendeten Firmware noch eine weitere Konfigurationsmöglichkeit.
Ist die Systembeschreibung abgeschlossen, können alle von der SpartanMC Toolchain benötigten Dateien generiert werden. Dies umfasst eine Top-Level-Verilog-Beschreibung der Konfiguration, in der sämtliche IP-Cores instanziiert, parametrisiert und miteinander verbunden werden. Desweiteren werden Linkerskripte für die einzelnen Speicher angelegt und C-Headerdateien, die Informationen über Modulparameter und Peripherien beinhalten und beim Schreiben der Firmware genutzt werden können. Ebenfalls erstellt werden Makefiles, welche Konstanten definieren, die im weiteren Verlauf von der Maketoolchain verwendet werden. Es werden außerdem noch ein User Constraints File (UCF) und Dateien erstellt, welche für die Simulation des SoC relevant sind.

2.1.2 SpartanMC Toolchain

Die SpartanMC Toolchain ist eine auf Makefiles basierende Toolchain, welche es dem Nutzer erlaubt, neue Projekte zu erstellen, diese zu verwalten und den Workflow zu steuern. Durch die Makefiles ist es beispielsweise möglich, JConfig aufzurufen, ein Bitfile zu erstellen, welches auf ein FPGA aufgespielt werden kann oder die Simulation zu starten. Alle relevanten Dateien, die zur Ausführung der Regeln in den Makefiles eines Projektes benötigt werden, werden von JConfig erzeugt (siehe vorheriger Abschnitt). Eine Ausnahme bildet die Firmware, welche der Nutzer selbst schreiben muss (die Ordnerstruktur kann allerdings ebenfalls durch eine Make-Regel erzeugt werden).
Um ein Bitfile zu erstellen, verwendet die Toolchain die entsprechenden Tools aus der Xilinx ISE

Design Suite. Die benötigten Steuerdaten und Kommandozeilenoptionen, werden von JConfig erzeugt und in den Dateien *project.mk*, *spartanmc.xst* und *spartanmc.prj* gespeichert.

2.1.3 XML-Modulbeschreibung

Um einen neuen IP-Core in JConfig nutzbar zu machen, ist es nötig eine XML-Modulbeschreibung zu erstellen. Hierzu sind in der SpartanMC Entwicklungsumgebung, unter dem Verzeichnis *libdevxml*, XML Schema Definition (XSD) Dateien abgelegt, welche die Syntax für die verwendeten XML-Dateien vorgibt.
Eine XML-Modulbeschreibung ist folgendermaßen gegliedert:

- **Header**: Im Header werden generelle Informationen angegeben, wie Kategorie und Name der Hardware.
- **HDL**: In diesem Abschnitt wird angegeben, welche HDL-Dateien eingebunden werden sollen. Bei den Pfadangaben wird davon ausgegangen, dass sich die Dateien in einem Verzeichnis namens *src* befinden, welches sich wiederum in dem Verzeichnis befindet, in dem die XML-Datei ist.
- **Parameters**: Darauf folgen können mehrere Parameter-Abschnitte. Jeder einzelne Abschnitt wird im JConfig UI als eigene Gruppe in der Parametersektion dargestellt. Für Parameter lassen sich diverse Einstellungen treffen. So können Parameter beispielsweise über eine Auswahl von Werten definiert oder der Wertebereiche begrenzt werden.
- **Ports**: Es können beliebig viele Ports-Abschnitte definiert werden. Dadurch wird die logische Zusammengehörigkeit der einzelnen Signale, die in einem jedem Abschnitt definiert sind, dargestellt. Einzelne Signale können mit einem Namen, Datenbreite und Richtung versehen werden.
- **Bus**: Busse ermöglichen es Signale zusammenzufassen und somit das User Interface (UI) etwas übersichtlicher zu gestalten und den Arbeitsaufwand, um einzelne Signale zu verbinden, zu reduzieren.

Außerdem gibt es noch einige besondere Konstrukte, welche nur für Module einer bestimmten Kategorie zur Verfügung stehen. Darunter fällt beispielsweise das *addressLayout*, welches nur für Prozessoren relevant ist, um den Adressraum zu beschreiben oder *memory*, welches nur für Speicherblöcke relevant ist.

2.2 Xilinx Embedded Development Kit

2.2.1 Microblaze

Der Microblaze von Xilinx ist ein für die Verwendung mit Xilinx FPGAs optimierter, 32-Bit Softcore Prozessor mit einer Reduced Instruction Set Computer (RISC) Architektur. Abbildung 2.1 zeigt das Blockschaltbild des Prozessors.
Der Microblaze verfügt über 32 32-Bit breite general purpose Register, 32-Bit Instruktionen

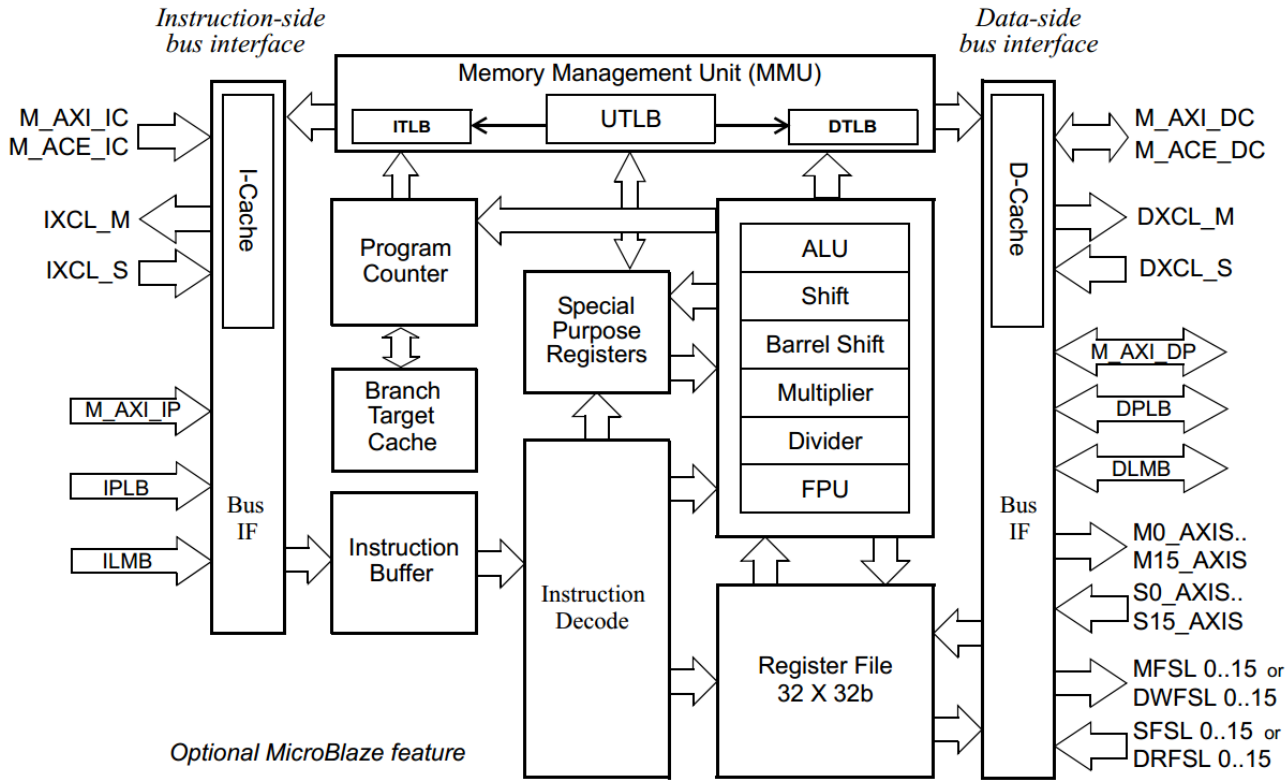


Abbildung 2.1: Blockschaltbild des Microblaze [MBR]

mit drei Operanden und zwei Adress-Modi, einen 32-Bit Addressbus sowie einer Single-Issue-Pipeline.
Neben diesen festen Eigenschaften ist der Microblaze über Parameter flexible konfigurierbar. So ist es beispielsweise möglich, über den Parameter C_AREA_OPTIMIZED die Anzahl der Pipelinestufen auf drei oder fünf festzulegen. Weitere Features des Microblaze sind in folgender Liste dargestellt:

- **Hardwarebeschleuniger**: Dem Microblaze steht dedizierte Hardware für verschieden Aufgaben zur Verfügung, die bei Bedarf per Parameter hinzugefügt werden. Dazu zählen ein Integer Dividierer, ein Integer Multiplizier, eine Floating Point Unit (FPU) und ein Barrel Shifter. In der Instruction Set Architecture (ISA) sind für die Hardwarebeschleuniger eigene Instruktionen vorgesehen (z.b. fmul für eine Gleitkomma Multiplikation).
- **MMU**: Der Microblaze verfügt über eine konfigurierbare Memory Management Unit (MMU). Dabei stehen drei verschiedene Modi zur Verfügung: Usermode, Protection und Virtual. Außerdem verfügt die MMU über je einen Translation Lookaside Buffer (TLB) für Instruktionen und für Daten.
- **Caches**: Werden externe Speicher verwendet, kann der Microblaze auf Caches sowohl für Instruktionen, als auch für Daten zurückgreifen. Hierbei lassen sich Einstellungen wie Größe des Caches, Länge einer Cacheline und Datenbreite festlegen.
- **Speicher- und I/O-Zugriff**: Der Microblaze hat für Speicher- und I/O-Zugriffe verschiedene Möglichkeiten. Einerseits existiert der Local Memory Bus (LMB), ein Bus mit geringer

Latenz, welcher für Zugriffe auf On-Chip Speicherblöcke gedacht ist. Das Advanced eXtensible Interface (AXI4) und der Processor Local Bus (PLB) sind sowohl für Speicher- als auch für I/O-Zugriffe nutzbar. Das Xilinx CacheLink (XCL) Interface ist als hochperformante Lösung für Zugriffe auf externe Speicher gedacht. Voraussetzung für die Nutzung ist, dass der Memory Controller, an den das XCL Interface angeschlossen ist, FSL-Buffer implementiert.

- **Debugging**: Der Microblaze verfügt ebenfalls über die Möglichkeit zum Hardware Debugging, es wird allerdings ein zusätzliches externes Debugging-Modul benötigt.
- **Streaming Interfaces**: Um Daten mit geringem Overhead zu übertragen, verfügt der Microblaze über die Streaming Interfaces AXI4-Stream und FSL. Für FSL muss ein Buffer zur Verfügung stehen, um versendete Daten zu puffern.
- **Exceptions**: Der Microblaze kann so konfiguriert werden, dass bei bestimmten Ereignissen (z.b. Math Exceptions, Bus Exceptions, ...) Hardware Exceptions ausgelöst werden.

Für weitere Informationen kann das Handbuch des Microblaze zu Rate gezogen werden ([MBR] *Kapitel 2 - Microblaze Architecture*).

2.2.2 XPS

Das Xilinx Platform Studio (XPS) ist der Systembuilder von Xilinx. Mit ihm lassen sich per grafischer Oberfläche SoC erstellen. Über einen Wizard kann zu Beginn ein Basis-System erstellt werden, welches sich danach einfach erweitern lässt. Aus einem Katalog von IP-Cores kann zusätzliche Hardware ausgewählt werden, welche dann über einen Wizard konfiguriert werden kann. Desweiteren bietet XPS einfache Möglichkeiten, Komponenten untereinander zu verbinden, Adressen von I/O und Speicherblöcken anzupassen und Eingangs- und Ausgangssignale zu verwalten. Außerdem ist es möglich, von der Nutzeroberfläche eine Netzliste zu generieren oder das Hardwaredesign (inklusive Bitfile) zum Software Development Kit (SDK) zu exportieren, um Code für das SoC zu schreiben. [MBT]

2.2.3 SDK

Mit dem SDK ist es möglich, Software für den Microblaze zu erstellen. Basierend auf der exportierten Hardware kann das SDK ein Board Support Package (BSP) erstellen, welches alle relevanten Treiber einbindet. Desweiteren lassen sich über Templates schnell einfache Beispielprogramme erstellen. Dem Softwareprojekt ist eine Makefile Struktur hinterlegt, die bei Änderungen an dem Source Code relevante Teile neu kompiliert. Außerdem ist es möglich, das FPGA mit dem aktualisierten Bitfile zu programmieren und auch zu debuggen. [MBT]

2.2.4 FSL

FSL ist eine unidirektionale Punkt-zu-Punkt Verbindung die eine einfache und schnelle Datenübertragung zwischen Hardware, die das FSL-Interface implementiert, ermöglicht. Xilinx bietet als Implementierung eines FSL-Buses den *FSL V20 Bus* IP-Core an [FSL]. Diese Hardware bietet eine auf First-In-First-Out (FIFO) Speichern basierte Kommunikation. Abbildung 2.2 zeigt das Blockschaltbild der FSL-Bus Komponente.

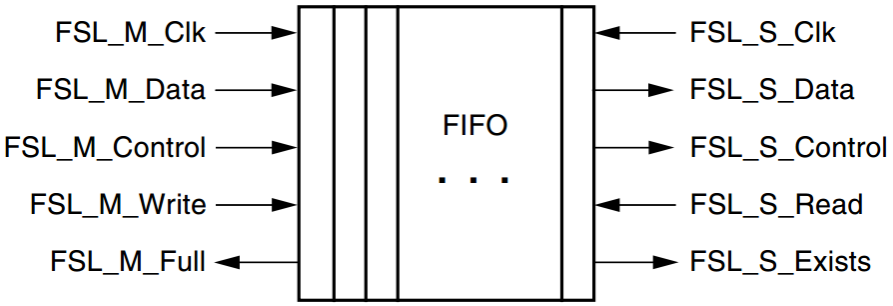


Abbildung 2.2: Blockschaltbild des Xilinx FSL IP-Cores [FSL]

Der Bus kann entweder synchron oder asynchron betrieben werden. Für den asynchronen Betrieb stellen Master und Slave eigene Taktsignale zur Verfügung (*FSL_M_Clk* und *FSL_S_Clk*), für den synchronen Betrieb wird ein globales Signal verwendet, welches nicht im Blockschaltbild dargestellt ist.
Um Daten in den FIFO-Speicher zu schreiben, muss der Master das *FSL_M_Write*-Signal anlegen. Dies ist allerdings nur dann möglich, wenn das *FSL_M_Full*-Signal anzeigt, dass der Speicher nicht voll ist. Neben einfachen Nutzdaten kann außerdem ein Steuerbit übertragen werden, welches beispielsweise dazu genutzt werden kann, um zwischen Steuer- und Nutzdaten zu unterscheiden. Der Slave wiederum kann Daten aus dem Speicher lesen, insofern das *FSL_S_Exists*-Signal gesetzt ist.
Für den IP-Core können des weiteren noch Einstellungen über Implementationsart der FIFO-Speicher (Block Random Access Memory (RAM) oder Look Up Table (LUT) RAM), Tiefe des FIFO-Speichers oder Art des Taktmodus (synchron oder asynchron) getroffen werden. Für genauere Informationen zu Signalen, Parametern und Buszugriffen, kann das Datenblatt des IP-Cores zu Rate gezogen werden [FSL].
In der ISA des Microblaze sind mit *get*, *getd*, *put* und *putd* besondere Instruktionen vorhanden, die es ermöglichen, Daten direkt vom Registersatz über das FSL-Interface zu schreiben, bzw. Daten vom FSL-Interface direkt in den Registersatz einzulesen. Dies ermöglicht Übertragungen mit niedriger Latenz.

2.2.5 Speicherinitialisierung im Microblaze

Für die Speicherinitialisierung im Microblaze stellt Xilinx das Tool data2mem zur Verfügung. [DAT] Data2mem liest sowohl eine textuelle Beschreibung der Speicherarchitektur in Form von Block RAM Memory Map (BMM) Dateien ein, als auch ausführbaren Code in Form von Executable and Linkable Format (ELF) Dateien. Mit diesen Dateien als Input ist es möglich, Speicherinitialisierungen in Form von UCF, Verilog oder Very High Speed Integrated Circuit Hardware Description Language (VHDL) Dateien zu erzeugen, die während der Synthese (UCF Datei) oder Simulation (HDL Dateien) verwendet werden können. Desweiteren ist es möglich, Block RAMs in existierenden Bitfiles zu aktualisieren, ohne den kompletten Toolflow noch einmal durchlaufen zu müssen.
BMM-Dateien haben eine eigene Syntax, die anhand des folgenden Ausschnitts einer BMM Datei erklärt wird.

```
ADDRESS_MAP microblaze_spmc_0 MICROBLAZE-LE 1
   ADDRESS_SPACE microblaze_spmc_0_bram_block_combined COMBINED [0x0:0x1fff]
      ADDRESS_RANGE RAMB16
      BUS_BLOCK
         microblaze_spmc_0_microblaze_mem_local_0/mem_core/MEM_BL[0].DPRAM [31:24];
         microblaze_spmc_0_microblaze_mem_local_0/mem_core/MEM_BL[1].DPRAM [23:16];
         microblaze_spmc_0_microblaze_mem_local_0/mem_core/MEM_BL[2].DPRAM [15:8];
         microblaze_spmc_0_microblaze_mem_local_0/mem_core/MEM_BL[3].DPRAM [7:0];
      END_BUS_BLOCK;
      END_ADDRESS_RANGE;
   END_ADDRESS_SPACE;
END_ADDRESS_MAP;
```

ADDRESS_MAP definiert die Speicherabbildung für einen bestimmten Prozessor mit dem Namen *microblaze_spmc_0* vom Prozessortyp MICROBLAZE-LE mit der Prozessor ID 1. Es können beliebig viele *ADDRESS_MAP*-Blöcke hinzugefügt werden.
ADDRESS_SPACE definiert einen kontinuierlichen Speicherbereich des entsprechenden Prozessors. Innerhalb eines *ADDRESS_SPACE*-Abschnitts können mehrere *ADDRESS_RANGE*-Blöcke vorkommen. Jeder dieser Blöcke beschreibt einen Speicherbereich, mit einer Größe die einer Potenz von Zwei entspricht. So lassen sich auch asymetrische Speicherstrukturen beschreiben.
In einem *BUS_BLOCK*-Abschnitt wird beschrieben, wie viele Bitlanes der Speicher hat und welchem Block RAM welche Bitlane zugewiesen ist.

3 Konzeption und Implementation

3.1 Anforderungen

Ziel dieser Arbeit ist es, den Xilinx Microblaze in die bestehende SpartanMC Entwicklungsumgebung zu integrieren. Um dies zu erreichen, muss zunächst der Microblaze in JConfig eingebunden werden. Der CoreGenerator von Xilinx bietet keine Alternative, da lediglich das Microblaze Micro Controller System (MCS) [MCS] zur Verfügung steht. Dies ist ein System bestehend aus einem Microblaze, Speicher und grundlegenden Peripherien, wie einem UART-Core und Timern. Die Konfigurierbarkeit des Microblaze innerhalb des MCS ist stark eingeschränkt. Features wie Caches oder FSL-Interface stehen nicht zur Verfügung und die Pipeline ist immer dreistufig. Da auf die umfangreiche Konfigurierbarkeit und das FSL-Interface nicht verzichtet werden soll, muss der Microblaze in JConfig integriert werden. Um dies zu erreichen, müssen zunächst sowohl eine Hardwarebeschreibung, als auch eine XML-Modulbeschreibung für den Microblaze erstellt werden. Dies ist ebenso erforderlich für den UART IP-Core und den FSL-Bus IP-Core. Bei der Erstellung der Hardware- und der XML-Modulbeschreibung, ist darauf zu achten, dass dem Nutzer es ermöglicht wird, die Parameter der neuen Komponenten konfigurieren zu können.
Es ist außerdem erforderlich, dass Anpassungen an der SpartanMC Toolchain vorgenommen werden. Dies ist notwendig, um die automatische Speicherinitialisierung für den Microblaze zu ermöglichen, da die bestehende Methode der Speicherinitialisierung für den SpartanMC nicht ohne Änderungen für den Microblaze anwendbar ist. Desweiteren muss der Compiler für den Microblaze in die Toolchain integriert werden, da einige der Hardwarebeschleuniger über spezielle Instruktionen angesprochen werden. Eine Vielzahl von IP-Cores von Xilinx sind in VHDL verfasst. Da in der SpartanMC Entwicklungsumgebung IP-Cores zum Großteil in Verilog verfasst wurden, ist es gegebenfalls notwendig, Anpassungen an der Toolchain vorzunehmen, um VHDL Designs zu unterstützen.
Optional wäre es wünschenswert, die erfolgreiche Paralellisierung eines Microblaze Programms durch μStreams zu zeigen. Hierzu wäre es notwendig, μStreams soweit anzupassen, dass eine entsprechende Hardwarekonfiguration mit den neu integrierten Komponenten erstellt werden kann und die Aufrufe der Core-Konnektoren durch Aufrufe der FSL-Blöcke ersetzt werden.

3.2 Integration in JConfig

3.2.1 Hardware

Bevor über die Integration nachgedacht werden kann, muss zunächst festgelegt werden, welche Hardware verwendet wird. Über die Jahre hat Xilinx diverse Versionen des Microblaze veröffentlicht. Die Implementationen des Prozessors sind allesamt verschlüsselt und können nur mit entsprechendem Key entschlüsselt werden. Mit der Einführung von Vivado und der Einstellung der Entwicklung von ISE, änderte sich auch die Art der Verschlüsselung, sodass neuere Versionen des Microblaze nicht mehr von der alten Toolchain entschlüsselt werden können. Da die

JConfig Toolchain allerdings Gebrauch von der ISE Toolchain macht, kommen für diese Arbeit nur der Microblaze v8.50c oder ältere Versionen in Frage.
Ausgehend vom ISE Installationsverzeichnis, ist das Verzeichnis für die IP-Cores an folgender Stelle zu finden: "*14.7/ISE_DS/EDK/hw/XilinxProcessorIPLib/pcores/*". Dort sind alle Versionen des Microblaze, sowie alle Versionen der restlichen Hardware abgelegt. Für sämtliche verwendete IP-Cores werden die aktuellsten Versionen verwendet.

3.2.2 Hardwarebeschreibung

Erzeugung der Hardwarebeschreibung mit XPS

Um den Microblaze in JConfig integrieren zu können, ist es zunächst notwendig, eine Hardwarebeschreibung zu erstellen, die den Microblaze instanziiert und parametrisiert. Diese kann entweder manuell erstellt werden oder mit XPS. In XPS kann dazu mit dem Base-System-Builder ein einfaches System erzeugt werden. Ein Beispiel für ein System ohne Peripherie ist in Abbildung 3.1 zu sehen. Das System besteht aus einem Micorblaze, zwei LMB, zwei Speichercontrol-

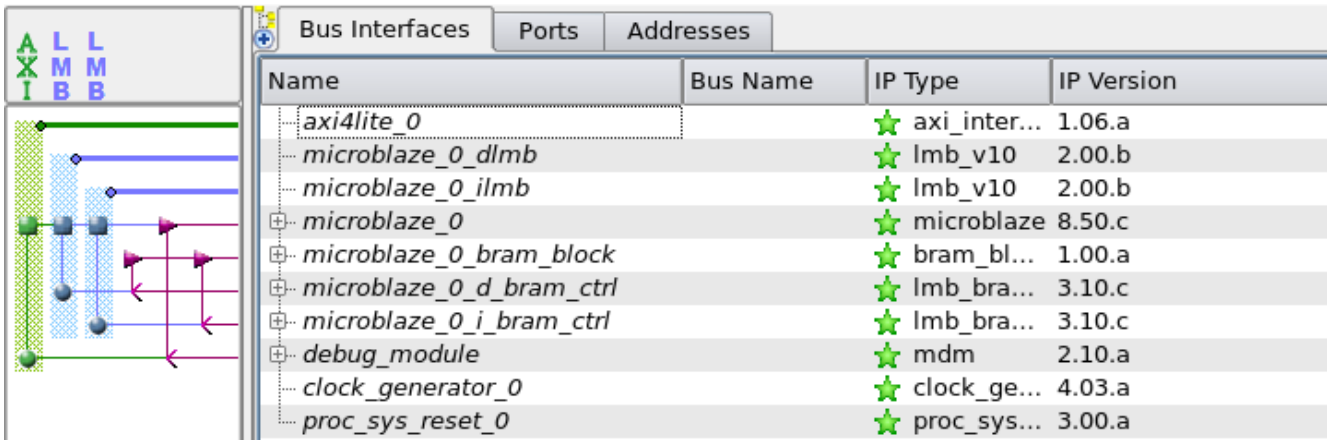


Abbildung 3.1: Einfaches mit XPS generiertes System ohne Peripherie

lern (je einen für Daten und einen für Instruktionen), einem Block RAM als Programm- und Datenspeicher, einem AXI4-Light-Bus zur Anbindung von Peripherie, einem Reset-Core, einem Taktgenerator und einem Debug Modul. Um die Komplexität des Systems und somit zusätzliche Fehlerquellen zu reduzieren, werden der AXI4-Light-Bus, der Taktgenerator und das Debug Modul zunächst entfernt. Der AXI4-Bus wird nicht benötigt, da es zunächst in der SpartanMC Entwicklungsumgebung mit dem UART IP-Core nur eine Peripherie geben wird und diese direkt mit dem Prozessor verbunden werden kann. Der Taktgenerator wird nicht benötigt, da JConfig bereits über eigene Taktgeneratoren verfügt und das Debug Modul wird nicht benötigt, um die grundsätzliche Lauffähigkeit eines Microblaze Systems zu gewährleisten, sondern nur um Softwareanwendungen zu debuggen.
Nun kann in XPS eine Netzliste erzeugt werden. Als Nebenprodukt werden Wrapperdateien erstellt, die die einzelnen Komponenten mit den im XPS angegebenen Einstellungen instanziieren. Die erzeugten Dateien werden nach Möglichkeit in der präferierten Sprache erzeugt, allerdings funktioniert dies bei manchen Wrapper Dateien nicht und sie werden in VHDL generiert. Die Top-Level-Beschreibung, welche sämtliche Komponenten instanziiert und miteinander

verbindet, kann allerdings auch in Verilog erzeugt werden. Desweiteren besitzt die Top-Level-Beschreibung lediglich die im XPS als extern makierten Signale als Eingänge und Ausgänge. Die Wrapper Datei für das Block RAM stellt eine Besonderheit dar. Für das Block RAM wird nämlich, entsprechend der angegebenen Größe des Speichers, ein elaboriertes Modell erzeugt. Dieses Modell funktioniert nur für die angegebene Speichergröße und lässt sich ohne großen Aufwand auch nicht ändern. Als Lösung des Problems wird ein generisches Speichermodul in Verilog geschrieben, welches das Verhalten der elaborierten Blöcke nachahmt (siehe 3.2.2).
Um nun eine konfigurierbare Hardwarebeschreibung zu erhalten, ist es notwendig, den Wrapperdateien und der Top-Level-Beschreibung Parameter hinzuzufügen. So können bei der Instanziierung des Top-Level-Moduls Parameter übergeben werden, die dann wiederum bei der Instanziierung der einzelnen Komponenten des Systems an diese weitergegeben werden können. Ebenso ist es erforderlich, dass Signale die verwendet werden sollen, als Eingänge bzw. Ausgänge der Top-Level-Beschreibung hinzugefügt und über Signale mit der entsprechenden Komponente verbunden werden. Für den Microblaze sind dies zu Beginn, neben Takt- und Reset-Signalen, Signale für den Programm- und Datenspeicher, AXI4-Signale und Signale, die dem FSL-Interface zuzuschreiben sind.
Unter Berücksichtigung der XML-Modulbeschreibung, ist es notwendig, den Speicher als externes Modul zu handhaben, da die XML-Syntax für Prozessoren keine integrierten Speicher unterstützt. Hierzu werden die Speichercontroller und das Block RAM in einem weiteren Modul zusammengefasst (siehe 3.2.2), sodass das Modul für den Microblaze noch aus den Instanzen für den Microblaze selbst, den beiden LMB und dem Reset-Core besteht. Die LMB-Cores erlauben es mehr als einen Speicher an den Microblaze anzuschließen.
Welche Parameter für die Konfiguration des Microblaze wichtig sind, wird im nächsten Abschnitt behandelt.

Handhabung der Parameter

In diesem Kapitel wird behandelt, welche Parameter für die Konfiguration des Microblaze-Systems wichtig sind. Für die Parameter des Microblaze wird die Tabelle aus dem Abschnitt *MicroBlaze Core Configurability* in Kapitel 3 des Microblaze Datenblattes [MBR] herangezogen. Parameter, die in der Tabelle nur einen Wert in der Spalte *Allowable Values* haben, werden fest auf diesen Wert gesetzt und können nicht vom Anwender verändert werden. In der Tabelle 3.1 werden lediglich die Parameter behandelt, welche mehr als einen erlaubten Wert haben, allerdings trotzdem auf einen festen Wert gesetzt werden. Alle anderen Parameter, die nicht genannt werden, stehen dem Nutzer zur freien Konfiguration zur Verfügung. Erklärungen zu den Bedeutungen der einzelnen Parametern werden in Kurzform in der XML-Modulbeschreibung hinterlegt, sodass der Nutzer während der Erstellung eines Systems weiß, welche Funktion die einzelnen Parameter haben. Für detailliert Beschreibungen sei noch einmal auf das Datenblatt des Microblaze verwiesen [MBR]. Neben dem Microblaze werden mit einem generischen Speichermodul, einem UART-Modul und einem FSL-Modul noch drei weitere Hardwarekomponenten für JConfig erstellt, die parametrisiert werden können. Welche Parameter zur Verfügung stehen und was diese bewirken, ist in den Tabellen 3.2, 3.3 und 3.4 aufgeführt.

Parameter	Wert	Beschreibung
C_LOCKSTEP_SLAVE	0	Im Lockstep Modus können zwei oder mehrere Microblaze das selbe Programm ausführen und die Ergebnisse miteinander vergleichen, um Hardwarefehler zu erkennen. Wird nicht benötigt.
C_ENDIANNESS	1	Endianness wird auf Little Endian festgelegt. In XPS lässt sich dieser Wert nicht manuell verändern, dementsprechend wird auch unter JConfig darauf verzichtet.
C_D_AXI	1	Das AXI-Daten-Interface kann verwendet werden.
C_I_AXI	1	Das AXI-Instruktionen-Interface kann verwendet werden. (Nützlich bei externen Speicher)
C_D_PLB	0	Da bereits AXI verwendet wird, wird der PLB Bus nicht gebraucht und deshalb deaktiviert.
C_I_PLB	0	Siehe vorheriger Eintrag.
C_D_LMB	1	Das LMB-Daten-Interface kann verwendet werden.
C_I_LMB	1	Das LMB-Instruktionen-Interface kann verwendet werden.
C_IPLB_BUS_EXCEPTION	0	Da der PLB nicht verwendet wird, werden die Exceptions auch nicht benötigt.
C_DPLB_BUS_EXCEPTION	0	Siehe vorheriger Eintrag.
C_ADDR_TAG_BITS	17	Das Datenblatt gibt kaum nützliche Informationen zu diesem Parameter, daher wird er auf den Default-Wert festgesetzt.
C_DCACHE_ADDR_TAG	17	Siehe vorheriger Eintrag.
C_USE_EXT_BRK	0	Siehe vorheriger Eintrag.
C_USE_EXT_NM_BRK	0	Siehe vorheriger Eintrag.

Tabelle 3.1: Parameter des Microblaze, die mehr als einen erlaubten Wert haben, aber dennoch auf einen Wert festegelegt werden.

Parameter	Beschreibung
C_FAMILY	Dieser Parameter wird von Xilinx IP-Cores genutzt, um, je nach verwendeter Hardware, vorhandene Primitives zu instanziieren.
RAMBLOCKS	Ähnlich wie für den Speicher des SpartanMC, kann angegeben werden, wie viel RAM instanziiert werden soll. Ein Block ist jeweils 2KB groß. Es können nur Größen gewählt werden, die einer Potenz von Zwei entsprechen, da mit jeder Vergrößerung des Speichers sich die Anzahl der Bitlanes verdoppelt.
C_BASEADDRESS	Gibt die Startaddresse des Speichers an.

Tabelle 3.2: Parameter des Speichermoduls für den Microblaze.

Parameter	Beschreibung
C_S_AXI_ACLK_FREQ_HZ	Frequenz des Taktsignals, welches automatisch aus dem Takt abgeleitet wird.
C_FAMILY	Dieser Parameter wird von Xilinx IP-Cores genutzt, um, je nach verwendeter Hardware, vorhandene Primitives zu instanziieren.
C_BAUDRATE	Für die Übertragung verwendete Baudrate. Eine Auswahl von Werten zwischen 110 und 921600 steht zur Verfügung.
C_DATABITS	Anzahl der zu übertragenden Bits pro Frame. Werte von 5 bis 8 sind möglich.
C_USE_PARITY	Gibt an, ob ein Paritätsbit mit übertragen wird.
C_ODD_PARITY	Gibt an, ob die Parität gerade oder ungerade ist.

Tabelle 3.3: Parameter des UART-Cores.

Parameter	Beschreibung
C_EXT_RESET_HIGH	Gibt an, ob das externe Reset-Signal high-aktiv oder low-aktiv ist.
C_ASYNC_CLKS	Gibt an, ob das FSL-Interface synchron oder asyncron betrieben werden soll.
C_IMPL_STYLE	Wenn dieser Parameter auf 0 gesetzt ist, wird der FIFO-Speicher mit LUT RAMs implementiert, andernfalls mit Block RAMs.
C_USE_CONTROLL	Bestimmt, ob das Steuerbit mit übertragen wird oder nicht.
C_FSL_DWIDTH	Ist auf 32 Bit festgelegt, da andere Datenbreiten im Zusammenhang mit dem Microblaze wenig sinnvoll sind.
C_FSL_DEPTH	Beschreibt die Größe des FIFO-Speichers. Diese ist von den Parametern C_ASYNC_CLKS und C_IMPL_STYLE abhängig. C_ASYNC_CLKS=0 => 1-8192. C_ASNYNC_CLKS=1 und C_IMPL_STYLE=0 => 16-128. C_ASNYNC_CLKS=1 und C_IMPL_STYLE=1 => 512-8192. Ist C_ASNYNC_CLKS=1 muss die Größe außerdem einer Potenz von Zwei entsprechen.
C_READ_CLOCK_PERIOD	Wird automatisch aus dem Taktsignal berechnet. Wird genutzt um Timing Constraints für den asynchronen Pfad durch LUT RAMs zu erzuegen.

Tabelle 3.4: Parameter des FSL-Cores.

Implementation eines generischen Speichermoduls

Bevor über die Implementation eines generischen Speichermoduls nachgedacht werden kann, sollte zunächst die Struktur, des von Xilinx verwendeten Speichers erläutert werden. Hierzu

wird Abbildung 3.2 herangezogen. In der Abbildung sind einfache Blockschaltbilder für einen

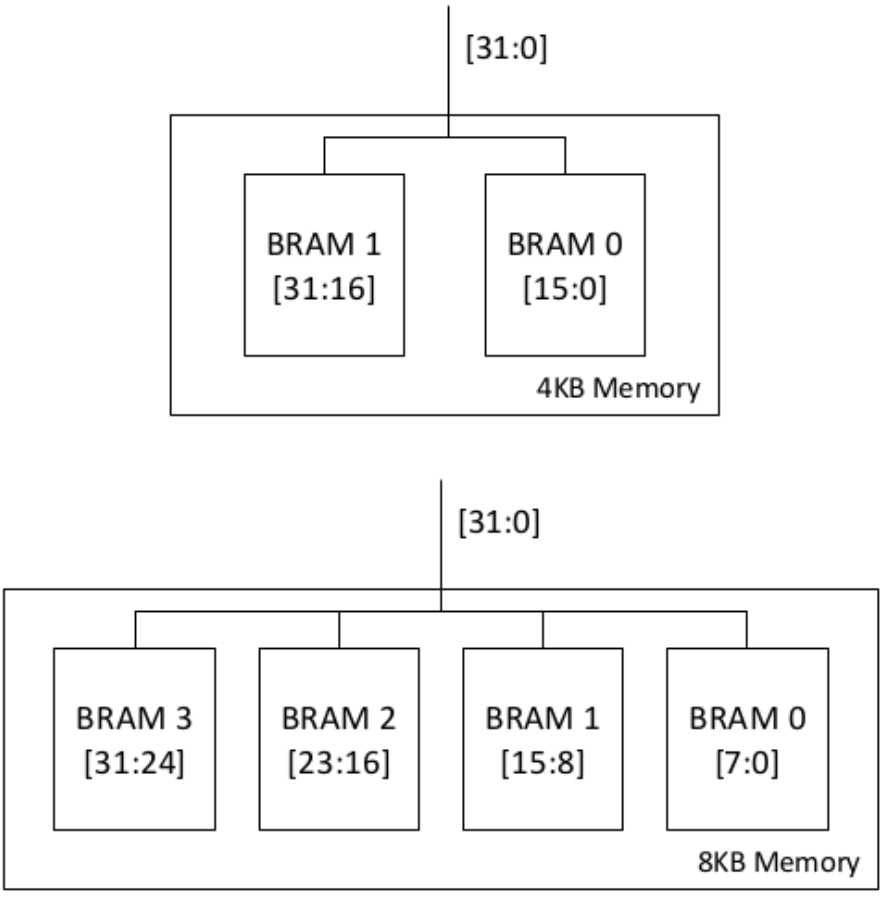


Abbildung 3.2: Speicherstruktur der von Xilinx generierten Speichermodule

4KByte und einen 8KByte großen Speicher dargestellt. Angefangen bei einer Mindestgröße von 2KByte verdoppelt sich die Anzahl der Block RAMs bei jeder Vergrößerung. Dabei halbiert sich jeweils die Größe der Bitlanes bis hin zu einem Minimum von eins bei einer maximalen Speichergröße von 64KByte. Die vom XPS generierten Modelle instanziieren immer eine fixe Anzahl an Block RAMs, sodass es nicht möglich ist, mit der selben Methode wie beim Microblaze eine generische Parametriserbarkeit herzustellen. Daher muss, auf Grundlage der elaborierten Modelle, ein generisches Speichermodul geschrieben werden, dessen Größe über einen Parameter festgelegt werden kann.
Um dies zu erreichen, wird ein Verilog-Modul geschrieben, welches die Parameter *RAMBLOCKS* und *C_FAMILY* hat. Über eine for-Schleife in einer generate-Umgebung werden entsprechend des Parameters *RAMBLOCKS* dual-ported Block RAMs instanziiert. Ein Port handhabt Daten und der andere Instruktionen. Eine Funktion berechnet dabei aus dem Parameter die Breite der einzelenen Bitlanes. Um die Signalverbindungen zu realisieren, werden wires und regs deklariert, deren Größe mit *RAMBLOCKS* skaliert. Über Indexing wird innerhalb der for-Schleife die Realisierung der einzelnen Bitlanes vorgenommen. Für das 4-Bit breite Write-Enable-Signal wird zusätzliche Kombinatorik hinzugefügt, da die Handhabung nicht durch einfaches Indexing umgesetzt werden kann.

Zuletzt werden der Speicher und zwei Speichercontroller in einer Top-Level-Beschreibung instanziiert und miteinander verbunden.

3.2.3 Modulbeschreibungen

Für den Microblaze, den Speicher, den UART-Core und den FSL-Core müssen nun XML-Modulbeschreibungen erstellt werden. Nachfolgend werden alle hierzu notwendigen Schritte besprochen:

- **Microblaze**: Der Microblaze wird in der Beschreibung als Prozessor kategorisiert. Neben den Wrapperdateien für den Microblaze, den LMBs und dem Reset-Modul müssen noch Pfadangaben für die HDL-Dateien der Implementationen dieser Cores angegeben werden. Um dies umzusetzen, wird die XML-Syntax mit *externalRoot* um ein neues Konstrukt erweitert. Dieses Konstrukt erlaubt es absolute Pfade außerhalb des Default-Verzeichnisses anzugeben und wurde nicht im Rahmen dieser Arbeit ergänzt, sondern von einem Kommilitonen, der zum Zeitpunkt dieser Arbeit für die Wartung von JConfig verantwortlich war. Alle Dateien, die sich im Scope eines *externalRoot*-Konstruktes befinden, verwenden den angegeben Pfad als Ausgangsverzeichnis. Im Rahmen dieser Arbeit, wurde in der Klasse *HDLDescription* im Package *de.tu_darmstadt.rs.spartanmc.devxml.descriptions.generic* eine Anpassung vorgenommen, die Umgebungsvariablen in Pfadangaben auflöst. So wird über die Umgebungsvariable *XILINX_ROOT* plattformunabhängig der Pfad zu den HDL-Dateien angegeben. Desweiteren ist es möglich, einen Namen für eine VHDL-Library anzugeben, unter dem die entsprechenden Dateien zusammengefasst werden. Dies ist für die Xilinx Cores notwendig, da innerhalb der Implementationen einige Libraries referenziert werden. Außerdem verkürzen Libraries die Kompilierzeit während der Simulation.
  Es folgen daraufhin die Angaben der Parameter. Hierbei wird darauf geachtet, dass die Gruppierung der Parameter in ähnlicher Weise erfolgt, wie es im Wizard des XPS der Fall ist. Die Parameter werden desweiteren mit einem Beschreibungstext versehen, sodass sich die Funktion für den Anwender bei der Parametrisierung erschließt. Außerdem haben die Konfigurationsmöglichkeiten einiger Parameter eine relativ geringe Aussagekraft. So lässt sich beispielsweise der Modus der MMU über einen Parameter steuern, der die Werte null bis drei annehmen kann. Um mehr Informationsgehalt zu vermitteln, wurde ein String-Parameter eingeführt, der die Werte *“None”*, *“Usermode”*, *“Protection”* und *“Virtual”* annehmen kann. Innerhalb der Top-Level-Verilog-Beschreibung wird dann über eine Aliasing-Funktion der String zu einem Integer umgewandelt. Dies wird für weitere Parameter durchgeführt, bei denen die Aussagekraft der Auswahlmöglichkeiten zu wenig Information vermittelt.
  Desweiteren schließen sich einige Konfigurationsmöglichkeiten gegenseitig aus. So ist es beispielsweise nicht möglich, die MMU zu verwenden, wenn der Parameter *C_AREA_OPTIMIZED* gesetzt ist. Um eine Auswahl in einem solchen Fall zu verhindern, wird das Konstrukt *relevantIf* in Kombination mit einer invertierten Version von *C_AREA_OPTIMIZED* verwendet.
  Darauf folgen Bus-Deklarationen für Speicherbusse und FSL-Busse (siehe 3.2.4), wie auch Signal-Deklarationen für Takt, Reset und dem AXI-Bus.
  Abschließend wird das *addressLayout* für den Daten- und Instruktionsspeicher definiert. Für Peripherie wurde noch kein Adressraum festgelegt, da es bis lang nur den UART-Core

als Peripherie gibt, und dieser direkt mit dem Microblaze verbunden wird. Sobald ein AXI-Bus in JConfig verfügbar ist, muss auch ein Adressraum für die Peripherie hinzugefügt werden.

- **Speicher**: Die Erstellung der Modulbeschreibung für den Speicher erfolgt ähnlich wie für den Microblaze. Die verfügbaren Parameter sind in Tabelle 3.2 aufgelistet. Der Speicher wird allerdings als Memory kategorisiert und hat mit dem *memory*-Konstrukt eine Besonderheit. Mit *memory* wird die Größe des Speichers, das Namensschema der Speicherinstanzen und die Reset-Werte beschrieben. Außerdem kann angegeben werden, ob der Speicher eine Aufteilung zwischen Daten und Instruktionen vorsieht. Diese Informationen werden später bei der Erzeugung der Memory-Map benötigt.

- **UART**: Auch hier wird die Modulbeschreibung nach ähnlichem Muster erstellt. Der UART-Core ist allerings als Peripherie kategorisiert. Über das *registers*-Konstrukt können Angaben zu Registern innerhalb des Cores gemacht werden, die später während der Erstellung der Software verwendet werden können. Im Rahmen dieser Arbeit konnte dieses Konstrukt für die UART allerdings nicht mehr erstellt werden. Die zur Verfügung stehenden Parameter sind in Tabelle 3.3 zu finden.

- **FSL**: Der FSL-Core wird als Core Interconnect kategorisiert und hat keine weiteren Besonderheiten. Die Parameter sind in Tabelle 3.4 aufgelistet.

3.2.4 Busbeschreibungen

Um das Verbinden einzelner Komponenten zu vereinfachen, werden zusammengehörige Signale zu einem Bus zusammengefasst. Im Folgenden wird auf die im Rahmen dieser Arbeit erstellten Busbeschreibungen eingegangen:

- **FSL-Busse**: Für den FSL-Bus werden zwei neue Busbeschreibungen in Form von XML-Dateien erstellt. Ein Bus beschreibt die Verbindungen für das Master-Interface und der andere die Verbindungen für das Slave-Interface (siehe Abbildung 2.2). Die Busbeschreibungen enthalten Defintionen für *masterPorts* und *slavePorts*, sowie alle dazugehörigen Signale, deren Richtung und Datenbreite. Bei der Deklaration innerhalb der Modulbeschreibung für den Microblaze wird darauf geachtet, das die einzelnen Signale in den Busbeschreibungen mit der Option *combineUsing*="*parallel*" versehen sind. Dies bewirkt, dass die einzelnen Signale, der bis zu 16 FSL-Busse konkateniert und so an den Microblaze weitergeleitet werden.

- **Speicherbusse**: Sowohl alle für den Datenport des Speichers relevanten Signale, sowie alle für den Instruktionsport relevanten Signale, werden zu je einem Bus zusammengefasst. Die Beschreibung erfolgt nach ähnlichem Schema wie bei den FSL-Bussen. Die *masterPorts* fassen alle Signale auf Seiten des Microblaze zusammen, während *slavePorts* alle Signale des Speichers umfasst.

Für die AXI-Signale wurden im Rahmen dieser Arbeit noch keine Busse erstellt, dies sollte allerdings mit der Integration eines AXI-Bus-IP-Cores nachgeholt werden.

3.3 Integration in die SpartanMC Toolchain

3.3.1 Ausgangslage

Der Microblaze und alle notwendigen Peripherien sind in JConfig eingebunden. Mit den von JConfig generierten Dateien, kann die Toolchain allerdings noch kein funktionsfähiges Bitfile erzeugen, da die Speicherarchitekturen des SpartanMC und des Microblaze verschieden sind. Der Speicher des SpartanMC ist so aufgebaut, dass dem Parameter *RAMBLOCKS* entsprechend Block RAMs instanziiert werden, dessen Bitlanes immer 18-Bit breit sind. So wird bei jedem Speicherzugriff auf genau ein Block RAM zugegriffen und nicht auf alle, wie es beim Microblaze der Fall ist.
Das Tool Initramj der SpartanMC Entwicklungsumgebung wird zur Speicherintialisierung für den SpartanMC verwendet und ist auf die vorher beschriebene Speicherstruktur angepasst. Das Tool ist unter anderem in der Lage, Speicherinitialisierungen in Form von UCF-Dateien, Verilog-Dateien oder Bitfile-Updates durchzuführen. Hierzu benötigt es eine ELF-Datei und eine Memory-Map in Form einer Datei namens *memory.xml*. Es wäre also denkbar, Initramj dahingehend anzupassen, dass es auch die Speicherinitilaisierung für die 32-Bit-Speicherarchitektur des Microblaze durchführen kann. Dies hätte den Vorteil, dass die Speicherininitialisierung gekapselt von einem einzigen Tool gehandhabt würde und somit kaum Änderungen an den Makefiles notwendig wären.
Eine Alternative zu Initramj stellt die Speicherinitialisierung mit dem Xilinx-Tool data2mem dar. Hierfür müsste JConfig allerdings eine weitere Memory-Map in Form einer BMM-Datei erzeugen. Desweiteren müssten die Makefiles angepasst werden, sodass sie data2mem aufrufen. Im Rahmen dieser Arbeit wurde die Variante mit data2mem umgesetzt, da der Aufwand zur Anpassung von Initramj als zu hoch eingeschätzt wurde.
Desweiteren ist es notwendig, den Microblaze GCC in die Toolchain zu integrieren, da der SpartanMC Compiler keinen vom Microblaze ausführbaren Code generieren kann. Um dies zu erreichen, müssen weitere Änderungen an den Makefiles vorgenommen werden.

3.3.2 Erzeugung der BMM-Datei

Damit JConfig in der Lage ist BMM-Dateien zu erzeugen, muss die Software erweitert werden. Das Package *de.tu_darmstadt.rs.spartanmc.jconfig.generation.outputProvider* des Tools libjconfig enthält diverse Klassen, welche auf Basis der über das UI eingegebenen Informationen, die von der Toolchain benötigten Dateien generieren. An dieser Stelle wird eine neue Klasse namens *BMMBuilderProvider* hinzugefügt. Diese Klasse generiert für jeden Speicher, der an einem Microblaze angeschlossen ist, einen Eintrag in der BMM-Datei, gemäß der Syntax, die in Abschnitt 2.2.5 beschrieben ist. Das Namensschema für die Speicherinstanzen wird aus der XML-Modulbeschreibung des Speichers gewonnen. Um festzustellen, ob ein Speicher an einem Microblaze oder an einem SpartanMC angeschlossen ist, werden den XML-Modulbeschreibungen zwei virtuelle Parameter hinzugefügt (*MB_FLAG* für den Microblaze und *SPMC_FLAG* für den SpartanMC). Der *BMMBuilderProvider* fügt einen Speicher nur dann zur Memory-Map hinzu, wenn der Parameter *MB_FLAG* vorhanden ist. Auf der anderen Seite muss aber auch verhindert werden, dass der Microblaze-Speicher in die *memory.xml* aufgenommen wird, da Initramj ansonsten versuchen würden, eine Speicherinitialisierung für diese Module durchzuführen. Da-

her wurde die Klasse *MemoryMapBuilderProvider* dahingehend angepasst, dass nur Speicher mit dem Parameter *SPMC_FLAG* behandelt werden.
Damit JConfig die BMM-Datei auch erstellt, muss eine Instanz des *BMMBuilderProvider* im Konstruktor der Klasse *XilinxIseDocumentBuilder* hinzugefügt werden. Des weiteren werden die Dateien *memory.xml* bzw. *memory.bmm* nur dann erstellt, wenn auch ein SpartanMC bzw. ein Microblaze im System vorhanden ist.

3.3.3 Anpassungen in der Toolchain zur Speicherinitialisierung

Mit der BMM-Datei kann data2mem nun verwendet werden (auch wenn die ELF-Dateien mit dem Microblaze noch nicht konform sind, kann data2mem diese verwenden). Die einzelnen Anpassungen an den Makefiles werden nachfolgend beschrieben.

Speicherinitialisierung mit UCF-Datei

Die Speicherinitialisierung mit einer UCF-Datei findet statt, wenn ein Bitfile über die Regel *"all"* erzeugt werden soll. Dabei wird eine weitere Regel namens *$(UCF)* aufgerufen, welche dafür sorgt, dass alle relevanten UCF-Dateien aktuell sind und diese dann über das Shell-Skript *mkucf* zu einer Datei zusammengefasst werden. Um UCF-Dateien für Microblaze-Prozessoren einzubinden, wird eine neues Makefile mit dem Namen *firmware_mb.mk* erstellt, welches die Erzeugung der UCF-Dateien handhabt. Da data2mem pro Aufruf allerdings nur eine einzige Firmware behandeln kann, muss das Tool mehrfach aufgerufen werden. Hierzu wird eine *for-each*-Schleife über alle Firmwares des Microblaze durchlaufen. Diese erzeugt für jede Firmware mit data2mem eine UCF-Datei. Um zu unterscheiden, welche Firmware zu welcher Prozessorart gehört, wird die Konstante *JCONFIG_FIRMWARE_IDS* aus dem von JConfig generierten Makefile *firmwares.mk* aufgeteilt in *JCONFIG_SPARTANMC_FIRMWARE_IDS* und *JCONFIG_MICROBLAZE_FIRMWARE_IDS*. Die hierzu notwendigen Änderungen an der Software werden in der Klasse *LegacyMkBuilderProvider* vorgenommen.
Die so erstellten UCF-Dateien werden an das Shell-Skript *mkucf* übergeben, welches daraus eine einzelne Datei macht. Die Syntax für den Aufruf von data2mem ist wie folgt:

data2mem -bm <BMM-Datei> -bd <ELF-Datei> tag <Prozessorname> -o u <UCF-Datei>

Die Option *-bm* spezifiziert die BMM-Datei und *-bd* die ELF-Datei. Der Zusatz *tag* ist notwendig, um die zum Prozessor passende *ADDRESS_MAP* in der BMM-Datei zu spezifizieren und *"-o u"* gibt an, dass das Ausgabeformat eine UCF-Datei sein soll. Mit der UCF-Datei können während der Synthese die Block RAMs initialisiert werden.

Aktualisierung des Bitfiles

Die Speicherinhalte eines bestehenden Bitfiles werden beim Aufruf der Regel *bitgen* erzeugt. Um die Updates für die Speicher des Microblaze durchzuführen, wird in die Liste der Prerequisites eine neue Regel namens *bit_mb_update* hinzugefügt. Diese Regel ruft für jede Firmware eines Microblaze data2mem auf und aktualisiert das Bitfile. Für die Aktualisierung des Bitfiles benötigt data2mem allerdings eine BMM-Datei, die Placement-Informationen der Speicherinstanzen

beinhaltet. Diese Datei kann allerdings von der Xilinx Toolchain erzeugt werden, wenn dem Translate-Tool NGDBuild beim Aufruf die originale BMM-Datei mit der Option *-bm* übergeben wird. Die Syntax für den Aufruf von data2mem ist wie folgt:

data2mem -bm <BMM-Datei> -bd <ELF-Datei> tag <Prozessorname>
-bt <zu aktualisierendes Bitfile> -o b <neues Bitfile>

Speicherintialisierung für die Simulation

Die Speicherinitialisierung für die Simulation wird mittels Verilog-Dateien durchgeführt. Durch ausführen der Regel *sim* wird eine entsprechende Verilog-Datei von Initramj erzeugt. Durch hinzufügen einer Regel in dem Makefile *firmware_mb.mk* wird durch Aufruf von data2mem für jede Firmware eine Verilog-Datei erzeugt, die die Speicherinitialisierung enthält. Der Aufruf hierfür ist ähnlich, wie bei der Erzeugung der UCF-Dateien:

data2mem -bm <BMM-Datei> -bd <ELF-Datei> tag <Prozessorname> -o v <Verilog-Datei>

Die so erzeugten Verilog-Dateien müssen nun noch zusammengefasst werden, da innerhalb der automatisch erzeugten Testbench nur ein einziges Initialisierungsmodul instanziiert wird. Es könnte auch eine Anpassung der Software vorgenommen werden, die für jede erzeugte Verilog-Datei eine Instanz in der Testbench erzeugt. Das Zusammenfassen der einzelnen Dateien stellt allerdings die elegantere Lösung dar und wird deshalb umgesetzt. Hierzu wurde ein Shell-Skript namens *mksim* auf Basis des Skripts *mkucf* erstellt. Es erhält als Eingabe eine unbestimmte Anzahl an Verilog-Dateien (sowohl für SpartanMC als auch Microblaze) und fügt diese zu einer einzigen Datei zusammen. Dabei werden noch zwei Zeilen eingefügt, die die Speicherinitialisierung als Verilog-Modul definieren (die entsprechenden Zeilen werden bei der Datei für den SpartanMC vorher entfernt). Außerdem wird der hierarchische Name der Speicherinstanzen, als Konsequenz aus der Initialisierung innerhalb der Testbench, um das Präfix *"UUT."* erweitert. Der Aufruf des Skripts findet innerhalb der Regel *sim* statt.

3.3.4 Unterstützung von Libraries in der Simulation

In Abschnitt 3.2.3 wurde bereits darauf eingegangen, dass Xilinx IP-Cores VHDL-Libraries referenzieren. Einer der Vorteile von Libraries ist, dass alle Dateien innerhalb einer vorkompilierten Library bei erneutem Kompilieren nicht berücksichtigt werden und daher Zeit gespart wird. Xilinx bietet mit dem Tool compxlib eine Möglichkeit Simulationslibraries für sämtliche IP-Cores des EDK zu erstellen. Die Auswahlmöglichkeiten bei der Erzeugung sind allerdings eingeschränkt. So ist es nicht möglich einzelne IP-Cores zu kompilieren, sondern es werden Simulationsmodelle für alle Cores des EDK erstellt. Der hierfür verwendete Speicherplatz ist mit mehr als 2GByte verhältnismäßig groß. Daher werden nur die Simulationslibraries der Hardware behalten, welche in JConfig eingebunden wurde. Die Auswahl wird mit einer Größe von ca. 70MByte dem SpartanMC Git-Repository in dem Verzeichnis *"spartanmc/simLib"* hinzugefügt, um zu gewährleisten, dass Nutzer die Libraries nicht selbst kompilieren müssen.
Um die neuen Libraries in der Simulation einzubinden, müssen Anpassungen an dem Skript *configure.tcl* vorgenommen werden. Hierzu benötigt das Skript Informationen darüber, welche

Libraries eingebunden werden müssen und wo diese zu finden sind. In der Klasse *XilinxIseProjectBuilderProvider* wird dazu ein neues Makefile namens *sim_lib.mk* erzeugt, welche eine Liste aller benötigten Libraries definiert. Desweiteren wird die Erstellung des Do-Skripts *spartanmc_worklib.fdo* angepasst, welche Befehle zur Kompilierung aller notwendigen Dateien enthält, sodass Dateien innerhalb einer Library nicht hinzugefügt werden, da diese ja vorkompiliert sind. Um die Pfadangabe zu den Simulationslibraries zu realisieren, wird in JConfig der Konfiguration unter dem Reiter Target das Feld *Simulation Library Path* hinzugefügt. Diese Pfadangabe wird im Makefile *project.mk* unter der Konstanten *JCONFIG_SIMULATION_LIBRARY_PATH* abgelegt. Mit diesen Informationen kann das Skript *configure.tcl* die entsprechenden Libraries hinzufügen. Desweiteren wurden noch Mappings für die Libraries *ieee*, *std* und *synopsys* ergänzt, da diese ebenfalls benötigt werden.

3.3.5 Integration des Microblaze GCC

Die Integration des Microblaze GCC konnte im Rahmen dieser Arbeit aus zeitlichen Gründen nicht durchgeführt werden. Es wird an dieser Stelle allerdings kurz darauf eingegangen, welche Ansätze für die Integration vorstellbar wären.
Die für den Ablauf des Build-Prozesses verwendeten Makefiles sind, ausgehend vom SpartanMC Root-Verzeichnis, unter dem Pfad *"src/scripts/make/firmware/"* zu finden. Das Makefile *fwbuild-gcc.mk* ist dabei für die Erzeugung der Firmware verantwortlich. Hierzu erstellt es auf Basis des Templates *gcc-build.mk* (zu finden im Verzeichnis *"templates"*) je ein Makefile für jede Firmware. Diese erzeugten Makefiles beinhalten alle notwendigen Regeln, um eine ELF-Datei zu erzeugen. In *fwbuild-gcc.mk* werden desweiteren alle verwendeten Tools, Source-Dateien und Flags für Compiler, Assembler und Linker festgelegt. Um den Microblaze GCC zu integrieren sollten in dieser Datei Änderungen vorgenommen werden.
Der Microblaze GCC und die Binutils sind im Xilinx ISE Installationsverzeichnis unter folgendem Pfad zu finden: *"14.7/ISE_DS/EDK/gnu/microblaze/lin/bin/"*. Desweiteren müssen basierend auf der Konfiguration des Microblaze bestimmte Compiler-Flags gesetzt werden, um Hardwarebeschleuniger zu integrieren. Hierzu ist es notwendig, dass die Software erkennt, wenn ein entsprechender Parameter gesetzt ist und das dazugehörige Flag in dem Makefile *firmwares.mk* hinzufügt. Relevante Informationen zum GCC sind im *Embedded System Tools Reference Manual* Kapitel 9 zu finden [MGN].
Außerdem müssen entsprechend der verwendeten Hardware passende Treiber inkludiert werden. Die Software zur Verwendung von Xilinx IP-Cores ist unter folgendem Pfad zu finden: *"14.7/ISE_DS/EDK/sw/XilinxProcessorIPLib/drivers/"*.

4 Benutzungshinweise

Um die Verwendung des Microblaze in der SpartanMC Entwicklungsumgebung zu erklären, wird an dieser Stelle Stück für Stück ein einfaches "*Hello World!*"-Beispiel erstellt. Hierzu wird zunächst ein neues SpartanMC-Projekt angelegt und JConfig gestartet. Der erste Schritt ist es, den Microblaze Prozessor hinzuzufügen. Der Prozessor ist unter *"Subsystem modules –> Processors –> Microblaze core"* zu finden. Als nächstes wird ein Speichermodul für den Microblaze hinzugefügt, welches unter *"Common modules –> Memory –> Microblaze Local Memory"* zu finden ist. Zur Ausgabe des Strings *"Hello World!"* wird noch ein UART-Core benötigt. Dieser findet sich unter *"Peripheral modules –> Bus –> AXI UART Light"*. Die Parametrisierung für die eingebunden Komponenten wird, bis auf die Baudrate der UART nicht verändert. Diese wird auf 115200 gestellt, um in der Simulation nicht lange auf Ergebnisse warten zu müssen. Zuletzt wird noch ein Modul zur Takterzeugung eingebunden, welches unter *"Common modules –> Clocks –> Xilinx DCM Clock"* zu finden ist. An der Konfiguration des Taktmoduls wird ebenfalls nichts verändert.
Der nächste Schritt ist das Verbinden der Signale. Beim Betrachten der Verbindungen wird festgestellt, dass der Microblaze bereits mit dem Speicher- und Takterzeugungsmodul verbunden ist. Das Signal *reset* muss noch mit dem Pin, der das Reset-Signal führt, verbunden werden (für das SP601 ist dies N4). Desweiteren müssen die AXI-Signale mit den gleichnamigen Signalen des UART-Cores verbunden werden. Bei den Adresssignalen muss eine partielle Verbindung erstellt werden. Der UART-Core wird noch mit dem Taktsignal und dem Reset-Signal *peri_resetn* des Microblaze verbunden. Das TX-Signal muss auf dem SP601 mit dem Pin K14 verbunden werden und das RX-Signal mit L12.
Desweiteren ist für die Konfiguration unter dem Reiter *Target* in das Feld *Simulation Library Path* der absolute Pfad zu den Simulationslibraries einzugeben. Das erstellte System ist in Abbildung 4.1 zu sehen.
Bevor die für die Toolchain relevanten Dateien erzeugt werden können, muss zunächst eine Dummy-Firmware erzeugt werden. Hierzu wird der Befehl *make newfirmware +path=<path-to-firmware>* verwendet. In der Verzeichnisstruktur muss im Ordner *src* eine Datei namens *main.c* hinzugefügt werden. In dieser Datei muss eine Funktion namens *main* implementiert sein. Der Inhalt dieser Funktion ist egal, da der Microblaze GCC noch nicht in die Toolchain integriert ist und die Speicherintialisierung manuel durchgeführt wird. Für dieses Beispiel wird folgender Code verwendet. Im Anschluss wird in JConfig auf *Build All* geklickt um die Dateien zu erzeugen.

```
#include <subsystems/microblaze_spmc_0/peripherals.h>
#include <stdio.h>
void main() {
   while(1){
      printf("hello world\n");
   }
}
```

Für den Fall, dass das Beispielsystem mit Hardware verwendet werden soll, muss nun eine Bitfile mit *make all* generiert werden. Im Verzeichnis *"spartanmc/hardware/microblaze/example"* ist eine ELF-Datei namens *hello_world.elf* zu finden. Diese enthält ausführbaren Code für ein

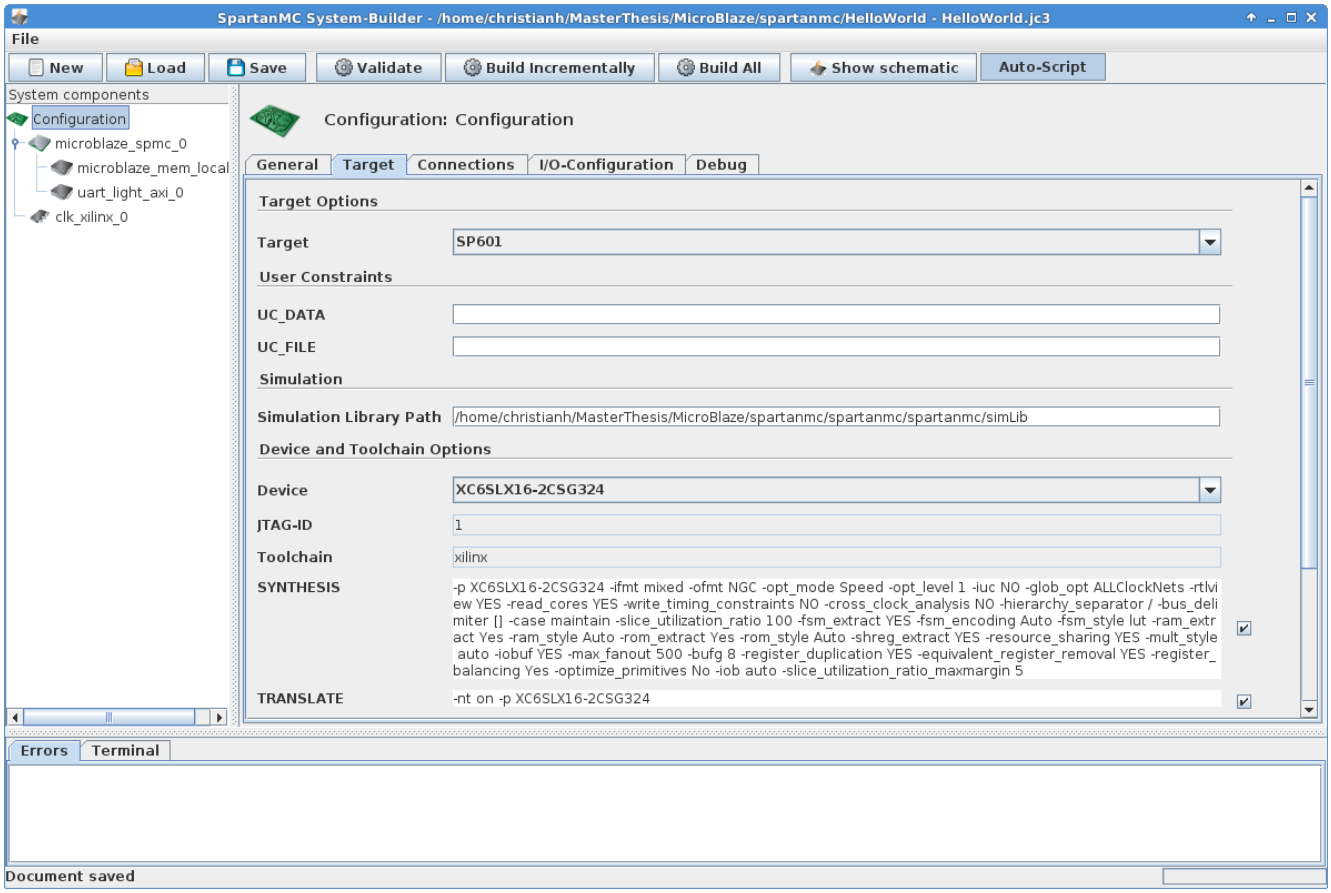


Abbildung 4.1: *"Hello World!"* Beispielsystem

einfaches, dem oben beschriebenen SoC sehr ähnlichem System. Die ELF-Datei beschreibt ein simples "*Hello World!*" Programm. Mit data2mem wird nun das Bitfile manuel aktualisiert. Der Aufruf ist wie folgt:

data2mem -bm memory_bd.bmm -bd hello_world.elf tag microblaze_spmc_0 -bt spartanmc.bit -o b spartanmc.bit

Danach kann mit *make program* das Bitfile auf das FPGA gespielt werden.
Für den Fall, dass das Beispielsystem simuliert werden soll, muss zunächst eine Speicherinitialisierung in Form einer Verilog-Datei erzeugt werden. Dazu ist folgender Aufruf von data2mem notwendig:

data2mem -bm memory.bmm -bd hello_world.elf tag microblaze_spmc_0 -o v sim1.v

Im selben Verzeichnis in dem sich die ELF-Datei befindet, existiert ein Shell-Skript namens *mksim*, welches die Verilog-Datei für die Simulation präpariert. Der entsprechende Kommandozeilenbefehl ist folgender:

sh mksim sim1.v

Dadurch wird die Datei *init_ram.v* erstellt. Diese Datei wird im SpartanMC-Projekt im Ver-

zeichnis *build* abgelegt. Nun muss noch der Eintrag *spartanmc_sim.v* in der Datei *system/spartanmc_worklib.fdo* in *init_ram.v* umgeändert werden. Als nächster Schritt folgt das Erzeugen eines Simulationsverzeichnisses mit dem Befehl *make newsim +path=<path-to-sim>* und abschließend das Starten der Simulation im neuen Verzeichnis mit *vsim -do testbench.fdo*. Der Simulatior sollte starten und das System kann simuliert werden.

5 Evaluation

5.1 Vorgehensweise

In diesem Kapitel soll der erreichte Grad der Integration des Microblaze in die SpartanMC Entwicklungsumgebung evaluiert werden. Dabei soll nicht nur gezeigt werden, dass auf Microblaze basierende Systeme erstellt und synthetisiert werden können, sodern auch, dass die Integration die ursprüngliche Funktionalität nicht negativ beeinflusst. Hierzu soll zunächst ein einfaches System mit allen hinzugefügten Komponenten mit JConfig erstellt und anschließend mit der Toolchain ein Bitfile erzeugt werden. Außerdem sollen die Aktualisierung des Bitfiles und die Speicherinitialisierung für die Simulation ausgeführt werden. Dies soll zeigen, dass die Integration des Microblaze in JConfig erfolgreich war und dass die Anpassungen an den Makefiles zur Speicherinitialisierung ebenfalls funktionieren.
Desweiteren soll Software mit dem SDK von Xilinx geschrieben werden, welche sowohl die FSL-Blöcke als auch den UART-Core verwendet. Mit data2mem soll aus dem entstehenden ELF-File eine Speicherinitialisierung für die Simulation in Form einer Verilog-Datei erstellt werden. Das System muss anschließend simuliert werden, um die Funktionalität der einzelnen Systemkomponenten zu verifizieren.
Um zu zeigen, dass die Änderungen keinen negativen Einfluss auf SpartanMC-Systeme haben, soll der Workflow mit zwei weiteren Systemen durchlaufen werden. Das eine System soll nur SpartanMC-Instanzen beinhalten, während das andere sowohl einen SpartanMC, als auch einen Microblaze beinhaltet.

5.2 Erstellen des Testsystems mit JConfig

Das für die Evaluation verwendete System ist in Abbildung 5.1 dargestellt. Im Folgenden wird der Aufbau des Systems und das Vorgehen bei der Erstellung kurz beschrieben. Den Kern des Systems stellen zwei Microblaze dar, deren Parametrisierung sich nur durch das Setzen des Parameters *C_FSL_LINKS* auf 1 von der Default-Parametrisierung unterscheidet. Desweiteren verfügen die Prozessoren über 8KByte Speichermodule. Ein UART-Modul ist an den zweiten Microblaze Prozessor angeschlossen und es existiert ein FSL-Core, der eine unidirektionale Verbindung von *microblaze_spmc_0* zu *microblaze_spmc_1* realisiert. Die Parametrisierungen entsprechen hierbei den Standardeinstellungen. Desweiteren wurde ein Modul zur Takterzeugung hinzugefügt, welches einen Takt von ca. 49.8MHz erzeugt. Dieser Taktwert wird im Hinblick auf das mit XPS generierte Referenzsystem gewählt, da dort ein Takt von 50MHz generiert wird und diese Einstellung mit dem Taktmodul aus der SpartanMC Entwicklungsumgebung nicht exakt reproduziert werden kann.
Beim Verbinden der einzelnen Module fällt auf, dass das Zusammenfassen der Signale für Speicher und FSL zu Bussen die Arbeit erleichtert und das Interface übersichtlich hält, während das Verbinden der einzelnen AXI-Signale deutlich mehr Aufwand erfordert. Desweiteren ist die Wahrscheinlichkeit ein Signal falsch zu verbinden durch die schiere Anzahl an Signalen relativ hoch.

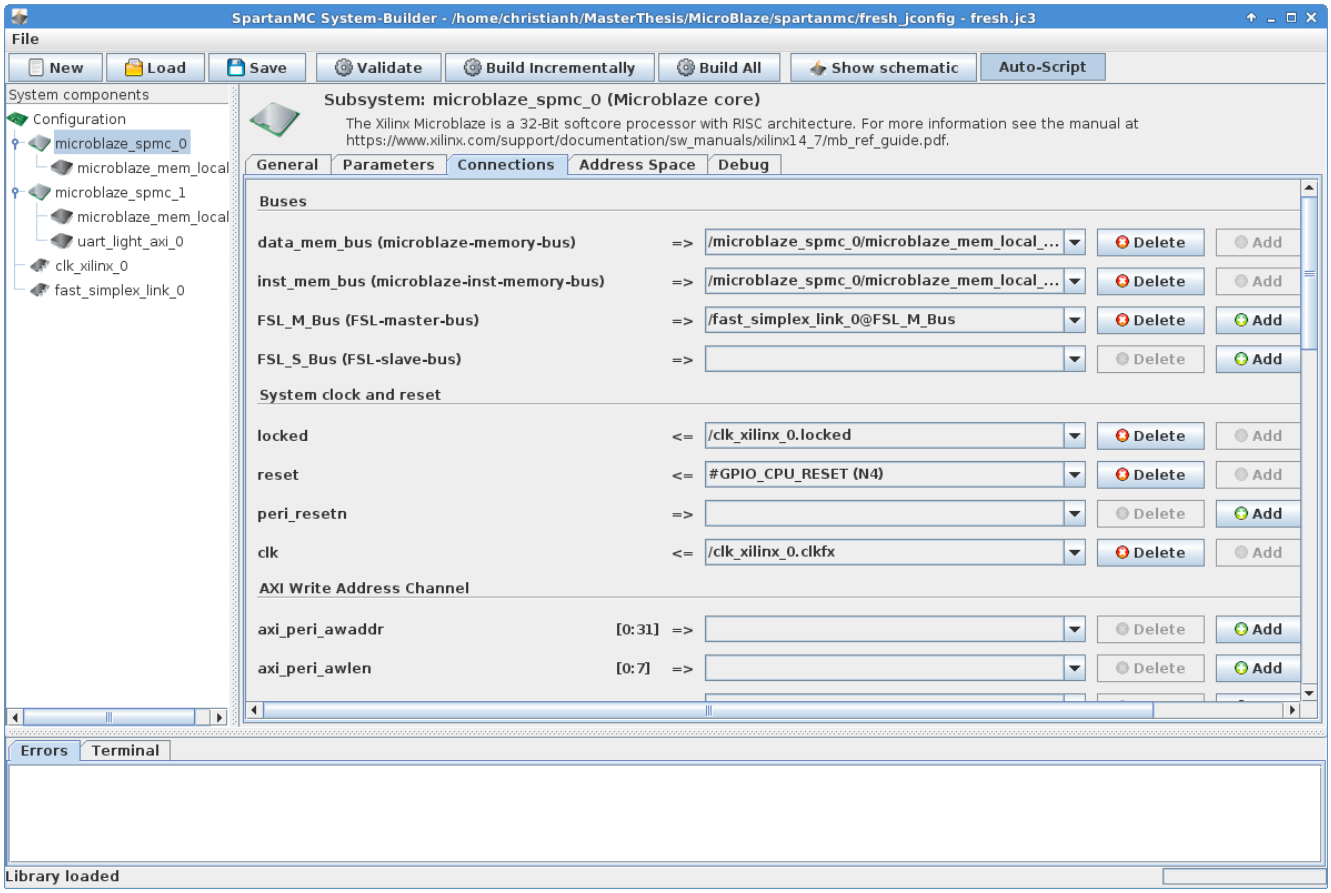


Abbildung 5.1: Testsystem zur Evalutation des erreichten Grades der Integration

Die Parametrisierung der neuen Module wird durch die Gliederung mehrerer Parameter in logische Gruppen übersichtlich gestalltet. Die Beschreibungen neben den Parametern geben Auskunft über ihre Bedeutung und in der Modulbeschreibung wird auf das Microblaze Handbuch verwiesen, um genauere Informationen zu den Parametern zu erhalten. Durch das Aliasing von einigen Parametern, sind deren Auswahlmöglichkeiten deutlich aussagekräftiger.
Im Vergleich zum SpartanMC ist der Grad der Automatisierung noch relativ gering. Für den SpartanMC existiert ein Groovy-Skript, welches automatisch eine Speicherinstanz zum Design hinzufügt und gleichzeitig Verbindungen zum Taktmodul erstellt. Dies wäre für den Microblaze auch denkbar und würde die Inkludierung eines Moduls etwas erleichtern.
Zusammenfassend lässt sich sagen, dass das Erstellen von Microblaze-Systemen dem Erstellen von SpartanMC-Systemen sehr ähnelt. Der Grad der Automatisierung lässt sich durch Groovy-Skripte noch erhöhen und der Arbeitsaufwand zum Verbinden der AXI-Signale durch Busbeschreibungen reduzieren. Desweiteren wäre es wünschenswert, dass der Parameter *C_FSL_LINKS* des Microblaze automatisch, basierend auf der Anzahl der verwendeten FSL-Interfaces gesetzt wird. Alles in allem lassen sich Microblaze-Systeme in JConfig jedoch einfach und verhältnismäßig schnell ohne die Verwendung von Xilinx-spezifischen grafischen Oberflächen erstellen.

5.3 Verwendung der Toolchain

Zum Testen der Toolchain wurden zunächst mit dem Befehl *make newfirmware* ein neues Verzeichnis für eine SpartanMC-Firmware erstellt. Diese Dummy-Firmware soll nur dem Zweck dienen, die Toolchain zu testen und kann nicht verwendet werden, um die Funktionalität der Systemkomponenten in der Simulation zu verifizieren. Bevor durch den Befehl *make all* ein Bitfile generiert wird, wird überprüft, ob die von JConfig generierten Dateien, die durchgeführten Änderungen übernommen haben. Dies umfasst folgende Dateien:

- **configuration.v**: Hier wird überprüft, ob alle Module aus JConfig instanziiert werden und sowohl die Parameter, als auch die Verbidungen korrekt gesetzt werden.
- **firmwares.mk**: In dieser Datei sollte die Liste *JCONFIG_SPARTANMC_FIRMWARE_IDS* leer sein und *JCONFIG_MICROBLAZE_FIRMWARE_IDS* die Namen der beiden Speicherinstanzen beinhalten.
- **project.mk**: Die Konstante *JCONFIG_SIMULATION_LIBRARY_PATH* muss definiert sein und den Pfad zu den Simulationslibraries angeben.
- **sim_lib.mk**: Hier muss überprüft werden, ob die Liste *JCONFIG_SIM_LIBRARIES* alle notwendigen Libraries definiert.
- **spartanmc_worklib.fdo**: Das Do-Skript darf nur Befehle zur Kompilierung von HDL-Dateien enthalten, die nicht einer der vorkompilierten Libararies angehören.
- **memory.bmm**: Hier muss sichergestellt werden, dass für jeden, mit einem Microblaze verbundenen Speicher, eine *ADDRESS_MAP* erzeugt wird. Dabei sollte auf Name und Größe des Speichers, sowie die Definitionen der Bitlanes geachtet werden.
- **memory.xml**: Diese Datei darf nicht erstellt werden, da es keinen SpartanMC im SoC gibt.

Unter Berücksichtigung der angegebenen Kriterien, wird während der Überprüfung der Dateien festgestelt, dass kein fehlerhaftes Verhalten beobachtet werden kann. Mit *make all* wird nun die Erzeugung des Bitfiles initiiert und es werden alle Schritte von der Synthese bis hin zur Generierung des Bitfiles durchlaufen. Vor dem Schritt Translate werden die ELF-Dateien erzeugt und es können in der Konsole die Aufrufe von data2mem zur Erzeugung der UCF-Dateien beobachtet werden. Ein Blick in die Datei *spartanmc.ucf* zeigt, dass die Initialisierung der Speicherblöcke vorhanden ist. Es wird von einer korrekten Erzeugung ausgegangen, da für die Inintialisierung mit data2mem ein von der Qualitätssicherung von Xilinx freigegebenes Tool verwendet wurde. Desweiteren wird im Abschnitt 5.5 darauf eingegangen, ob gültige Instruktionen während der Simulation aus dem Speicher gelesen werden können. Aus den Berichten für die einzelnen Schritte der Erzeugung des Bitfiles (von Synthese bis hin zu Place-and-Route) kann außerdem herausgelesen werden, dass tatsächlich Hardware generiert wurde.
Die Aktualisierung des Bitfiles wird mit dem Befehl *make bitgen* getestet. Beim Aufruf werden in der Konsole die einzelnen Aufrufe von data2mem dargestellt.
Um die Anpassungen an der Toolchain für die Simulation zu testen, wird mit dem Befehl *make newsim* ein neues Verzeichnis für die Simulation angelegt. In diesem Verzeichnis befindet sich unter anderem die Testbench, ein Do-Skript, welches die Simulation steuert und ein eine Intialisierungsdatei für Modelsim, welche die Library-Mappings enthält. Durch Eingabe des Befehls

vsim -do testbench.fdo wird der Simulator gestartet und das Do-Skript ausgeführt. Während der Ausführung des Skripts wird die Datei *modelsim.ini* dahingehend erweitert, das sie die Mappings der Xilinx Simulationslibraries enthält. Modelsim kompiliert alle relevanten Dateien und kann dann verwendet werden. Fehlermeldungen treten nicht auf. Die Speicherinitialisierung für die Simualtion ist in der Datei *spartanmc_sim.v* zu finden und wird ebenfalls korrekt erstellt.

5.4 Erstellen der Testsoftware und Erzeugung der Speicherinitialisierung für die Simulation

Da der Microblaze GCC noch nicht in die SpartanMC Toolchain integriert ist, muss die Software mittels Xilinx SDK erstellt werden. Hierzu wird zunächst mit XPS das System in Abbildung 5.2 erzeugt. Die Konfiguration des Systems entspricht größtenteils der des von JConfig erzeugten Systems. Einzige Unterschiede sind, dass in XPS der AXI-Bus verwendet werden muss und das Modul zur Takterzeugung ein anderes ist. Es wird ein Takt von 50Mhz erzeugt. Anschließend werden Netzliste und Bitfile generiert und das Design zum SDK exportiert.

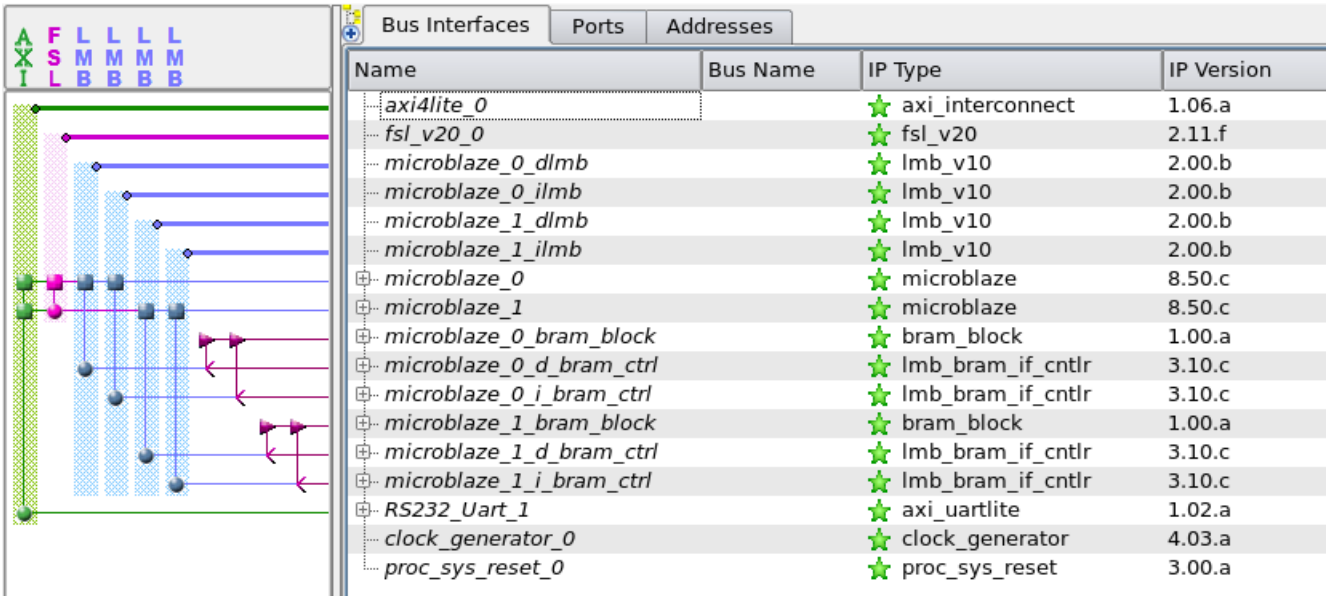


Abbildung 5.2: Referenzsystem zur Erzeugung der Testsoftware

Im SDK werden zunächst über *"New –> Application Project"* für jeden der beiden Prozessoren neue *"Hello World"*-Projekte und BSP angelegt. Nun wird folgende Applikation implementiert. Ein Prozessor sendet über das FSL-Interface den String *"Hello World!"* an den anderen Prozessor, welcher wiederum den empfangenen String über die UART verschickt. Mit dieser simplen Applikation werden alle in die SpartanMC Entwicklungsumgebung integrierten Komponenten verwendet. Der Code für beide Prozessoren ist nachfolgend aufgeführt.

```
/**Code for the sending processor*/
#include <stdio.h>
#include "platform.h"
#include <fsl.h>
#include <xparameters.h>
```

```
void print(char *str);

int main()
{
   int i = 0;
    init_platform();

    char* string = "Hello World!\n\r";
    while(1){
      for (i=0;i<13;i++){
         putfslx(string[i],0,FSL_NONBLOCKING);
      }
    }
    return 0;
}
```

```
/**Code for the receiving processor*/
#include <stdio.h>
#include "platform.h"
#include <fsl.h>

void print(char *str);

int main()
{
   int i = 0;
    init_platform();

    char* string = "";
    while(1){
      for (i=0;i<13;i++){
         getfslx(string[i],0,FSL_DEFAULT);
      }
      print(string);
    }
    return 0;
}
```

Die verwendeten Macros *putfslx* und *getfslx* sind in der Header-Datei *fsl.h* mittels Inline-Assembler definiert. Weitere allgemeine Informationen zu Treibern können in dem Dokument *OS and Libraries Document Collection* [OSL] gefunden werden. Das SDK erzeugt aus dem Code automatisch zwei ELF-Dateien. Mit diesen ELF-Dateien kann nun manuell der Speicher des Microblaze in der SpartanMC Entwicklungsumgebung initialisiert werden. Hierzu werden mittels data2mem zwei Verilog-Dateien aus den beiden ELF-Dateien und der *memory.bmm* erzeugt. Mit dem Skript *mksim* im Verzeichnis *"spartanmc/hardware/microblaze/example"* können die beiden Dateien zusammengeführt werden. Der Name der neuen Datei ist *init_ram.v*. Diese Speicherintialisierung wird im Verzeichnis *build* des SpartanMC-Projekts abgelegt. Nun muss nur

noch in der Datei *spartanmc_worklib.fdo* der Eintrag für *spartanmc_sim.v* zu *init_ram.v* geändert werden. Mit diesen Anpassungen ist das System nun bereit für die Simulation.

5.5 Simulationsergebnisse

5.5.1 Lesen von Instruktionen aus dem Speicher

In der Simulation soll zunächst gezeigt werden, dass Instruktionen erfolgreich aus dem Speicher gelesen werden. Abbildung 5.3 zeigt, wie Instruktionen in den Instruction-Buffer des Microblaze gelesen werden. Als Referenz, welche Werte zu erwarten sind, ist in Abbildung 5.4 ein Ausschnitt aus der ELF-Datei dargestellt.

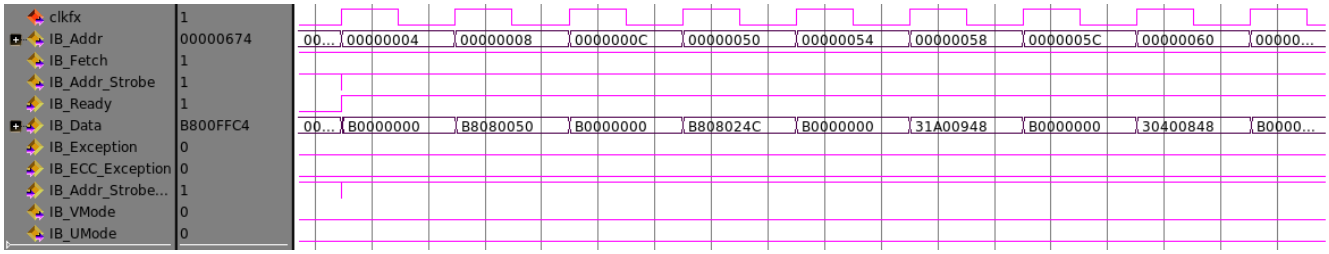


Abbildung 5.3: Simulation des Instruction-Buffers des Microblaze

Es lässt sich feststellen, dass sich die Einträge aus der ELF-Datei in der Simulation des Instruction-Buffers wiederfinden lassen. So kann davon ausgegangen werden, dass der Zugriff auf den Speicher funktioniert.

5.5.2 Kommunikation über das FSL-Interface

Als nächstes wird gezeigt, wie Daten über das FSL-Interface gesendet und aus dem FSL-Block gelesen werden. Hierzu wird Abbildung 5.5 betrachtet. Es kann beobachtet werden, wie das *'H'* des String *"Hello World!"* übertragen wird. Hierzu wird das Signal *FSL_M_WRITE* vom Master gesetzt, sodass im nächsten Takt das Signal *FSL_S_EXISTS* dem Slave signalisiert, dass Daten im FIFO-Speicher vorhanden sind. Der Slave reagiert mit dem Setzen von *FSL_S_READ*, um die Daten zu lesen. Basierend auf den Simulationsergebnissen kann angenommen werden, dass der FSL-IP-Core funktioniert.

5.5.3 Versenden von Daten über den UART-Core

Die Simulation des UART-Cores ist in Abbildung 5.6 dargestellt. Über einen Verlauf von 20μ/s kann beobachtet werden, dass die Valid- und Ready-Signale der einzelnen Kanäle konstant sind. Somit findet keine Übertragung statt. Als Vergleich wird das mit XPS generierte System simuliert, um zu prüfen, ob die selben Probleme auch im Referenzsystem auftauchen. Hierzu wird die Project-Datei des Referenzsystems in ISE importiert und eine Testbench erzeugt. Mit dem Xilinx Simulator ISIM wird die Simulation gestartet. Die Ergebnisse sind in Abbildung 5.7 dargestellt. Wie man sehen kann, wird nach ca. 12.8μ/s im Read Address Channel das Steuersignal *s_axi_arvalid* gesetzt. Auf Seiten des Microblaze, ist dieses Verhalten ebenfalls zu beobachten

```
void
disable_caches()
{
   0:   b0000000    imm 0
   4:   b8080050    brai    80  // 50 <_start1>

Disassembly of section .vectors.sw_exception:

00000008 <_vector_sw_exception>:
   8:   b0000000    imm 0
   c:   b808024c    brai    588 // 24c <_exception_handler>

Disassembly of section .vectors.interrupt:

00000010 <_vector_interrupt>:
    Xil_DCacheDisable();
  10:   b0000000    imm 0
  14:   b8080714    brai    1812    // 714 <__interrupt_handler>

Disassembly of section .vectors.hw_exception:

00000020 <_vector_hw_exception>:
    Xil_ICacheDisable();
  20:   b0000000    imm 0
  24:   b8080264    brai    612 // 264 <_hw_exception_handler>

Disassembly of section .text:

00000050 <_start1>:
  50:   b0000000    imm 0
  54:   31a00948    addik   r13, r0, 2376   // 948 <_SDA_BASE_>
  58:   b0000000    imm 0
  5c:   30400848    addik   r2, r0, 2120    // 848 <_SDA2_BASE_>
```

Abbildung 5.4: Ausschnitt aus der ELF-Datei des sendenden Microblaze

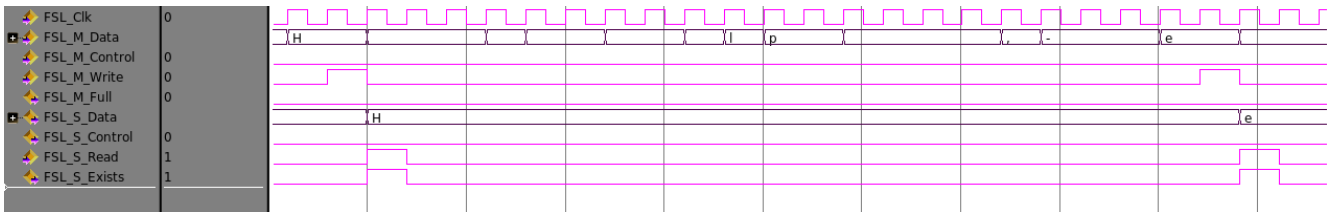


Abbildung 5.5: Simulation des FSL-Moduls

(es wird der Übersicht halber in der Abbildung nicht dargestellt). Die Kommunikation über den AXI-Bus wird fortgesetzt und nach ca. 90μ/s kann beobachtet werden, wie das TX-Signal des UART-Cores sich verändert.
Der Grund für dieses Verhalten des Microblaze in der SpartanMC Entwicklungsumgebung konnte aus zeitliche Gründen im Rahmen dieser Arbeit nicht mehr ermittelt werden. Die Unterschiede zwischen den beiden Systemen sind der AXI-Busses und die Takterzeugung. In der Simulation mit ISIM kann allerdings beobachtet werden, dass sämtliche Eingangssignale des AXI-Interfaces des Microblaze sich vor dem Setzen des *s_axi_arvalid*-Signals nicht verändern. Der Microblaze startet die Übertragung also von sich aus, was bedeutet, dass der fehlende AXI-Bus in der SpartanMC Umgebung nicht das Problem sein kann. Die Ursache des Problems muss daher beim

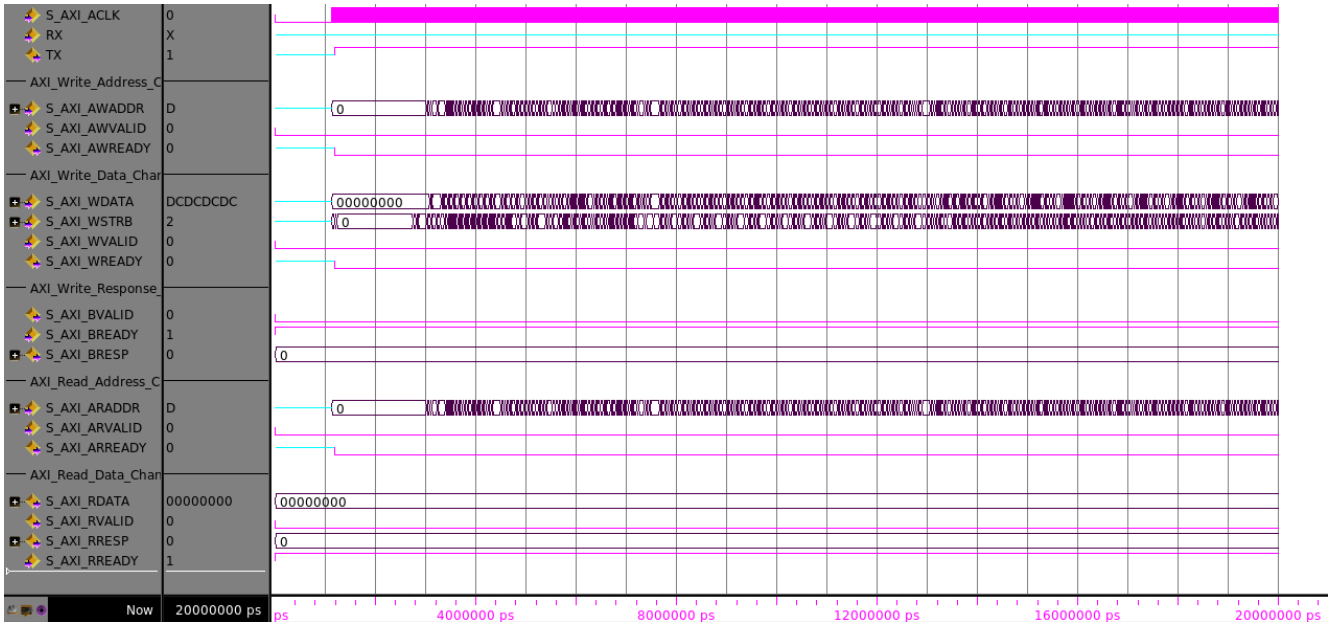


Abbildung 5.6: Simulation des UART-Cores

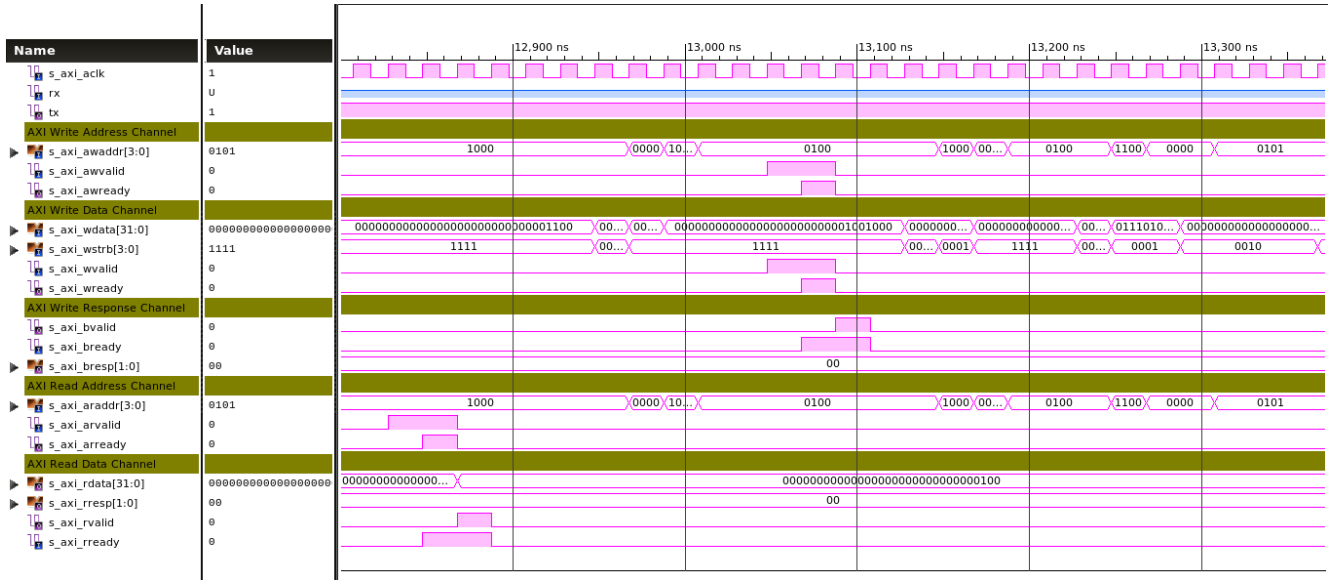


Abbildung 5.7: Simulation des UART-Cores im Referenzsystem mit ISIM

Microblaze liegen. Ein Vergleich mit der Parameterisierung des Microblaze im Referenzsystem hat allerdings keine Erkenntnis über die Ursache des Verhaltens gebracht.

5.6 SpartanMC- und Hybrid-Systeme

Da keine Veränderung am SpartanMC vorgenommen wurden, werden lediglich die gleichen Tests, wie in Abschnitt 5.3 durchgeführt. Das erste Testsystem besteht aus zwei SpartanMC und das zweite System aus je einem SpartanMC und Microblaze. Desweiteren werden zusätzlich UART-Module als Peripherie hinzugefügt, um zu verhindern, dass Hardware während der Synthese wegoptimiert wird.

Für das erste System terminieren die Tests ohne Fehler. Die UCF-Datei mit der Speicherinitialisierung wird korrekt erzeugt und auch der Aufruf von *make bitgen* aktualisiert das Bitfile entsprechend. Desweiteren lässt sich das System auch wie gewohnt simulieren. Es sei noch anzumerken, dass im Falle eines reinen SpartanMC-Systems die Datei *memory.bmm* nicht generiert wird.
Die Tests für das zweite System haben gezeigt, dass die Toolchain Hybrid-Systeme handhaben kann. Sowohl die Speicherinitialisierung per UCF-Datei, als auch die Aktualisierung des Bitfiles, durch ausführen von *make bitgen*, laufen erfolgreich durch. Die Simulation startet ebenfalls ohne Fehlermeldungen und die Verilog-Datei *spartanmc_sim.v* wird korrekt erzeugt.

6 Fazit und Ausblick

6.1 Fazit

In dieser Arbeit wurde der 32-Bit Softcore Prozessor Microblaze von Xilinx in die SpartanMC Entwicklungsumgebung integriert. Anhand der Abbildung 6.1 wird beschrieben, welche Schritte dafür erforderlich waren und zu welchem Grad der Microblaze integriert werden konnte.
Zu Beginn musste der Prozessor zusammen mit einem Speichermodul, einem FSL-Modul und einem UART-Core in den Systembuilder JConfig integriert werden. Hierzu wurden für die neue Hardware XML-Modulbeschreibungen erstellt, welche sämtliche Parameter und Signale der Komponenten definieren. Desweiteren wurden Wrapper-Module in Verilog und VHDL erstellt und der SpartanMC Hardware-Library hinzugefügt. Diese Wrapper instanziieren die neuen Cores mit den über JConfig getroffenen Einstellungen. Außerdem wurde ein neuer, generischer Speicher für den Microblaze in Verilog geschrieben, da kein generisches Modell vorhanden war. JConfig wurde dahingehend angepasst, dass es eine neue Memory-Map in Form einer BMM-Datei erzeugt, sobald ein Microblaze im System vorhanden ist. Das Xilinx Tool data2mem, welches zur Initialisierung und Aktualisierung von Block RAMs verwendet wird, wurde in die Toolchain integriert, um die Speicherinitialisierung für den Microblaze zu realisieren. Dieses Tool erlaubt es UCF-Dateien zu erzeugen, die in der Synthese dafür sorgen, dass die Speicher korrekt initialisiert werden. Außerdem ist data2mem in der Lage existierende Bitfiles zu aktualisieren ohne die Synthese erneut zu starten und es ist außerdem in der Lage Speicherinitialisierungen für die Simulation in Form von Verilog-Dateien zu erzeugen. Desweitern wurde im Rahmen dieser Arbeit das Konzept von Simulationslibraries in die Toolchain integriert. Dies erlaubt es VHDL-Dateien in Libraries zusammenzufassen und diese vorzukompilieren, sodass sie während der Simulation nicht erneut kompiliert werden müssen.

6.2 Ausblick

Im Rahmen dieser Arbeit konnte aus zeitliche Gründen allerdings nicht mehr die Integration des Microblaze GCC in die Toolchain realisiert werden. So ist die Toolchain nicht in der Lage, für den Microblaze kompatible ELF-Dateien zu erzeugen und muss daher auf extern erzeugte Varianten zurückgreifen. Desweiteren konnte keine erfolgreiche Datenübertragung über das AXI-Interface des Microblaze umgesetzt werden. In zukünftigen Arbeiten sollten daher zunächst diese beiden Probleme behoben werden. Neben diesen Punkten gibt es eine Reihe weiterer Möglichkeiten, diese Arbeit fortzusetzen. Es sollte auf jeden Fall der IP-Core für den AXI4-Bus in JConfig integriert werden, um es zu ermöglichen mehr als nur eine Peripherie an den Microblaze anzuschließen. Desweiteren wurde das optionale Ziel dieser Arbeit, die Paralellisierung eines Microblaze Programms mit μStreams, nicht erfüllt. Hierzu wäre es notwendig, μStreams dahingehend anzupassen, dass eine entsprechende Hardwarekonfiguration mit den neu integrierten Komponenten erstellt werden kann und die Aufrufe der Core-Konnektoren durch Aufrufe der FSL-Blöcke ersetzt werden. Außerdem wäre es interessant die verschiedenen Konfigurationsmöglichkeiten (z.b. externer Speicher mit Caches, MMU, Debug-Modul, ...) des Microblaze auszuprobieren,

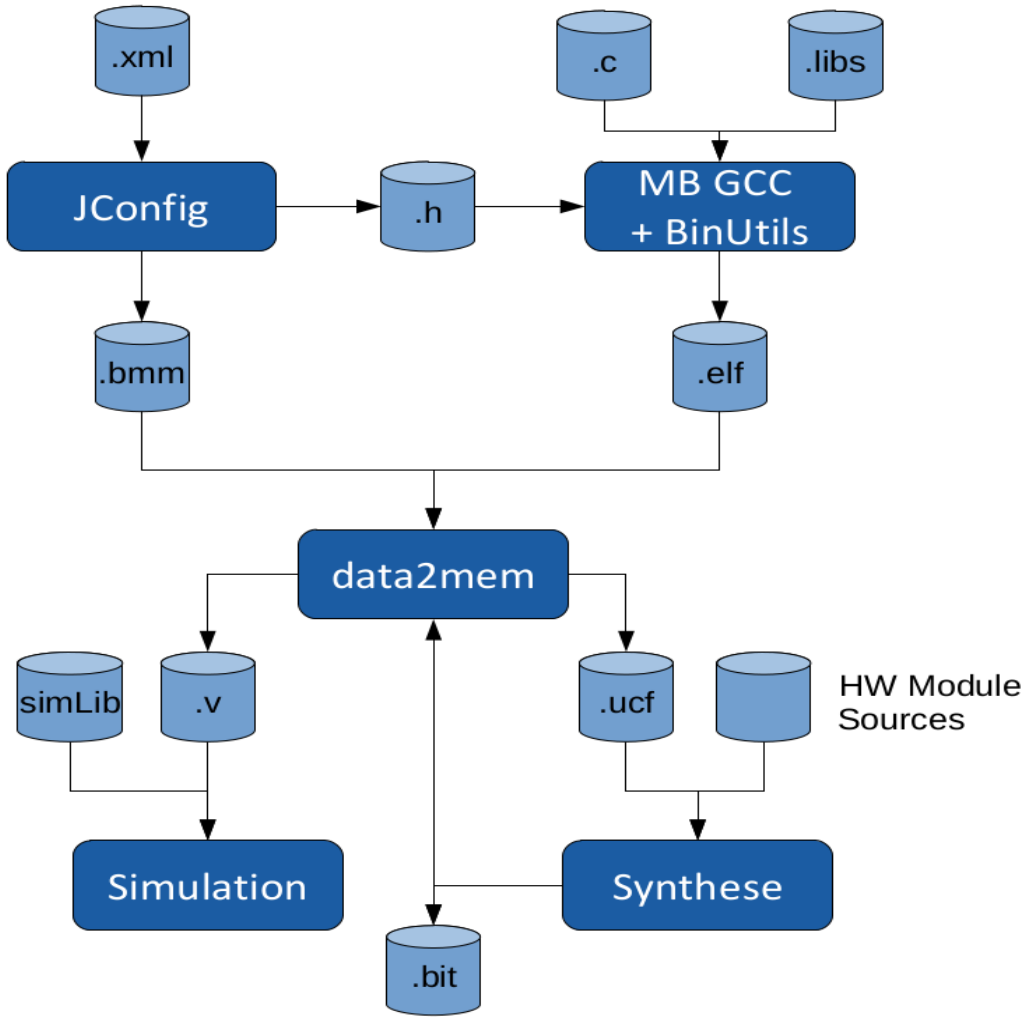


Abbildung 6.1: Schaubild zu den Neuerungen die im Rahmen dieser Arbeit vorgenommen wurden

da im Rahmen dieser Arbeit nur die Standardkonfiguration getestet wurde. Eine Möglichkeit die Nutzbarkeit von Simulationlibraries zu verbessern, wäre es, die kompletten Bibliotheken zu erzeugen und an einem, im Fachbereich Rechnersysteme global zugänglichem Verzeichnis abzulegen. Dies ist unter Berücksichtigung der Möglichkeit, dass weitere IP-Cores von Xilinx in Zukunft integriert werden und der langen Kompilierungszeit der Libraries durchaus sinnvoll.

Abkürzungsverzeichnis

CPU Central Processing Unit

MC Microcontroller

FPGA Field Programmable Gate Array

SoC System on Chip

I/O Input/Output

UART Universal Asynchronous Receiver Transmitter

IP Intellectual Property

FSL Fast Simplex Link

HDL Hardware Description Language

XML Extensible Markup Language

UCF User Constraints File

XSD XML Schema Definition

UI User Interface

FPU Floating Point Unit

ISA Instruction Set Architecture

MMU Memory Management Unit

TLB Translation Lookaside Buffer

LMB Local Memory Bus

AXI Advanced eXtensible Interface

PLB Processor Local Bus

XCL Xilinx CacheLink

XPS Xilinx Platform Studio

SDK Software Development Kit

BSP Board Support Package

FIFO First In First Out

RAM Random Access Memory

LUT Look Up Table

BMM Block RAM Memory Map

ELF Executable and Linkable Format

VHDL Very High Speed Integrated Circuit Hardware Description Language

MCS Micro Controller System

Abbildungsverzeichnis

2.1 Blockschaltbild des Microblaze [MBR] 8
2.2 Blockschaltbild des Xilinx FSL IP-Cores [FSL] 10

3.1 Einfaches mit XPS generiertes System ohne Peripherie 13
3.2 Speicherstruktur der von Xilinx generierten Speichermodule 17

4.1 "*Hello World!*" Beispielsystem 25

5.1 Testsystem zur Evalutation des erreichten Grades der Integration 28
5.2 Referenzsystem zur Erzeugung der Testsoftware 30
5.3 Simulation des Instruction-Buffers des Microblaze 32
5.4 Ausschnitt aus der ELF-Datei des sendenden Microblaze 33
5.5 Simulation des FSL-Moduls 33
5.6 Simulation des UART-Cores 34
5.7 Simulation des UART-Cores im Referenzsystem mit ISIM 34

6.1 Schaubild zu den Neuerungen die im Rahmen dieser Arbeit vorgenommen wurden 37

Tabellenverzeichnis

3.1 Parameter des Microblaze, die mehr als einen erlaubten Wert haben, aber dennoch auf einen Wert festegelegt werden. 15
3.2 Parameter des Speichermoduls für den Microblaze. 15
3.3 Parameter des UART-Cores. 16
3.4 Parameter des FSL-Cores. 16

Literaturverzeichnis

[CET] *Cetus - A Source-to-Source Compiler Infrastructure for C Programs*. https://engineering.purdue.edu/Cetus/

[DAT] *Data2MEM User Guide*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/data2mem.pdf

[FSL] *LogiCORE IP Fast Simplex Link (FSL) V20 Bus (v2.11f)*. https://www.xilinx.com/support/documentation/ip_documentation/fsl_v20/v2_11_f/fsl_v20.pdf

[MBR] *MicroBlaze Processor Reference Guide - Embedded Development Kit EDK 14.7*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/mb_ref_guide.pdf

[MBT] *EDK Concepts, Tools and Techniques - A Hands-On Guide to Effective Embedded System Design*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/edk_ctt.pdf

[MCS] *MicroBlaze Micro Controller System*. https://www.xilinx.com/products/design-tools/mb-mcs.html

[MGN] *Embedded System Tools Reference Manual*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/est_rm.pdf

[OSL] *OS and Libraries Document Collection*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/oslib_rm.pdf

[The] *The SpartanMC Project*. http://www.spartanmc.de

[Web15] Weber, Jan: *Design und Implementierung eines parallelisierenden Source-to-Source-Compilers für SpartanMC*, Technische Universität Darmstadt, Masterthesis, 2015