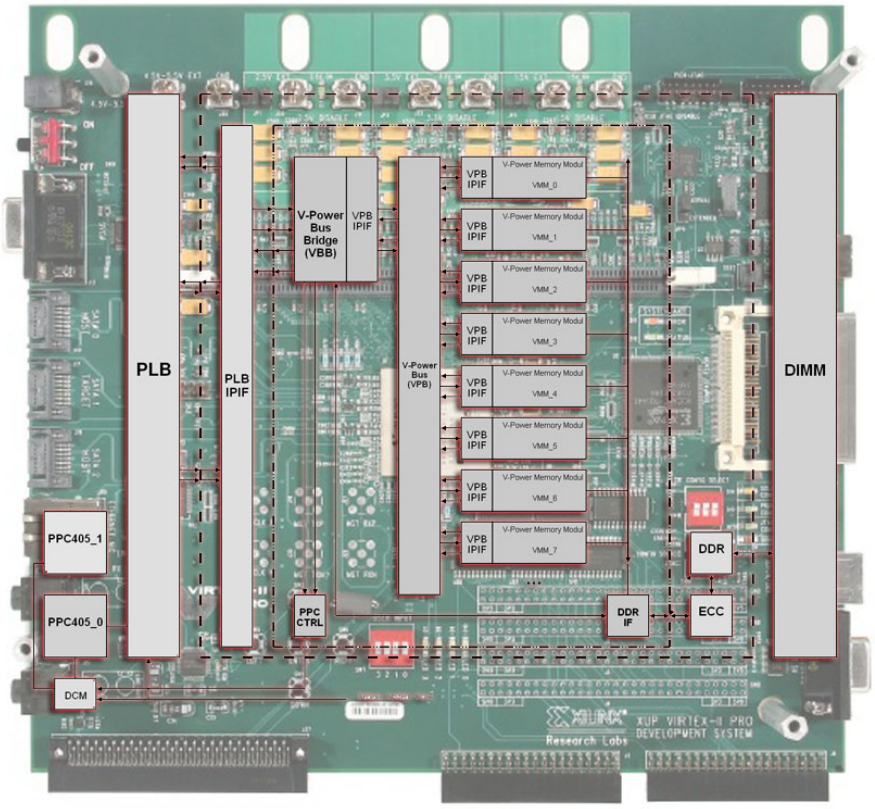
Endbericht

Projektgruppe 496

V-Power

Ein Speicher-Hierarchie Profiler mit Virtex-FPGAs und PowerPCs

Sommersemester 2007



Projektgruppe

V-Power

Endbericht

Projektgruppe 496

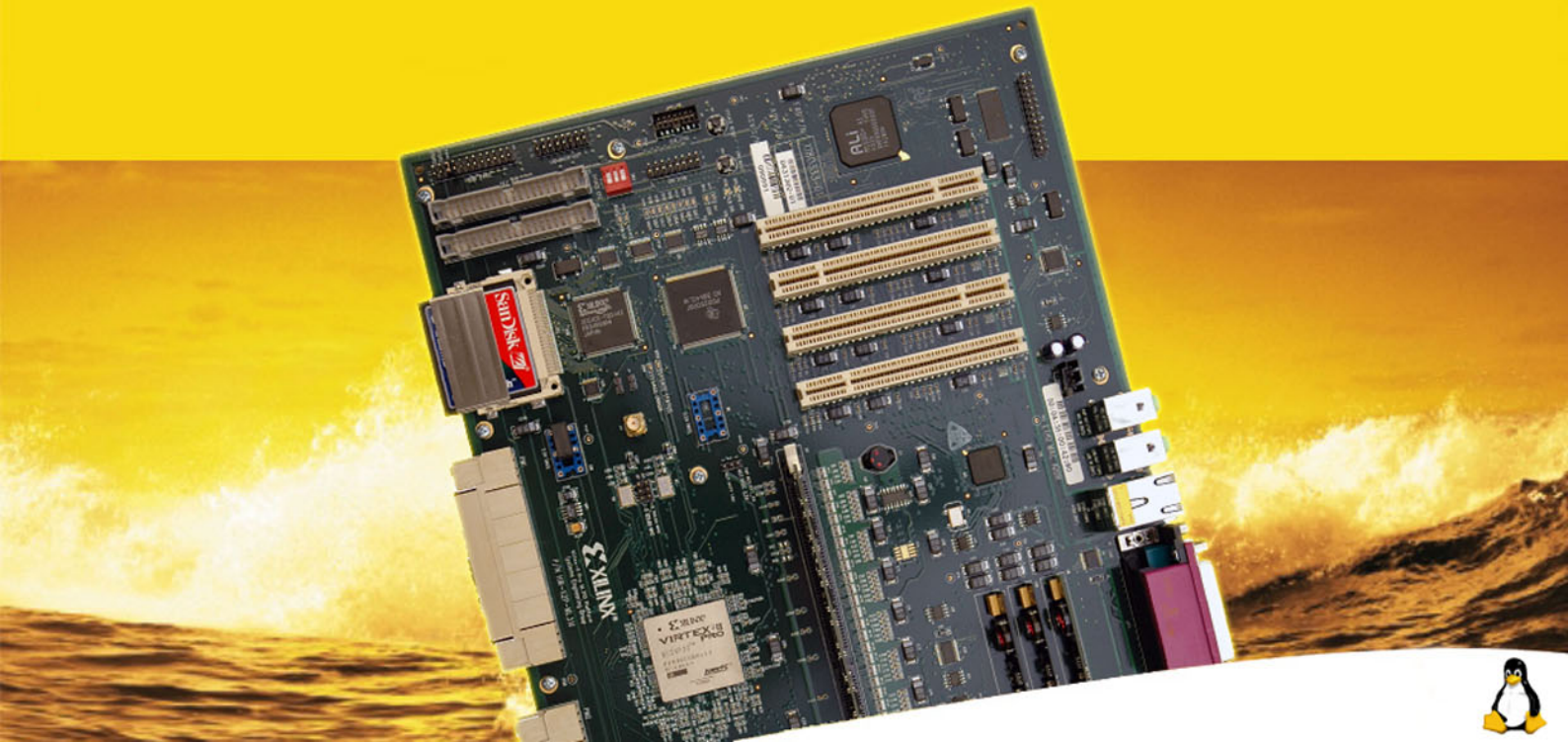
Ein Speicher-Hierarchie Profiler mit Virtex-FPGAs und PowerPCs

Sommersemester 2007

Teilnehmer:
Daniel Cordes, Arthur Falarz
Michael Gattmann, Sebastian Hanloh
Matthias Jung, Alexander Katriniok
Jan Kleinsorge, Alexander Milewski
Thomas Pucyk, Felix Rotthowe
Florian Schmoll, Stefan Wonneberger

Betreuer:
Heiko Falk, Robert Pyka

Universität Dortmund
Fachbereich Informatik
Lehrstuhl Informatik XII
Prof. Dr. Peter Marwedel



Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Im Bereich akkubetriebener portabler Eingebetteter Systeme hat sich gezeigt, dass die Struktur der Speicher- Hierarchie entscheidenden Einfluss auf den Energieverbrauch des gesamten Gerätes hat. Durch Ausnutzung einer geschickten Speicher-Hierarchie ist es möglich, bis zu 80% des Energieverbrauchs einzusparen [1]. In der Konsequenz bedeutet dies, dass sich die Zeit zwischen zwei Lade-Zyklen des Akkus entsprechend verlängert, so dass man mit seinem Gerät wesentlich länger ohne Strom-Anschluss telefonieren, Musik hören, Videos sehen, surfen, usw. kann.

Vor diesem Hintergrund stellt sich unmittelbar die Frage, welche Art von Speicher-Hierarchie denn für eine gegebene auf einem Prozessor auszuführende Applikation die am besten geeignete ist. So hat sich zum Beispiel gezeigt, dass sog. Scratchpad-Speicher schon in kleinen Größen von wenigen Kilobytes mit Compiler-Unterstützung zu massiven Energie-Einsparungen im Vergleich zu Caches führen [2]. Da Scratchpads aber gewissen praktischen Einschränkungen in ihrer Nutzung unterliegen, so dass man z.B. nur größere Code-Segmente und Daten ohne zusätzlichen Overhead darauf ablegen kann, können Caches durchaus auch vorteilhaft sein. Sie verwalten zusätzliche Informationen über ihren Inhalt, wie beispielsweise deren Nutzungsverhalten. Jedoch kann eine geschickte Speiche-Hierarchie nur dann zu Einsparungen führen, wenn der ausgeführte Programm-Code auch bezüglich des Speichers optimiert ist. Energiesparen in Eingebetteten Systemen bedarf somit der gemeinsamen Optimierung von Code und Speicher-Hardware.

In der Vergangenheit sind am Lehrstuhl Informatik 12 etliche Verfahren zur Energieoptimierung von Programm-Code entstanden [3, 4]. Die Güte dieser Verfahren wurde in der Vergangenheit gemessen, indem für eine gegebene Speicher-Hierarchie zunächst der unoptimierte Code komplett simuliert und sämtliche Speicher-Zugriffe protokolliert (referenzierte Adresse, Lese- / Schreib-Zugriff, Bitbreite, . . .) wurden. Anschließend wurde dieses Protokoll schrittweise für die Speicher-Hierarchie analysiert und der Energieverbrauch pro einzelnem Zugriff bestimmt. Durch Kumulieren dieser Werte erhält man abschließend den durch die Speicher verursachten Energieverbrauch der gesamten

simulierten Applikation. Dieses Verfahren ist letztlich noch einmal für den optimierten Programm-Code durchzuführen.

Es liegt auf der Hand, dass Software-Simulationen von Prozessor und Speicher-Hierarchien sowie das Summieren von Energiewerten laufzeitintensive Verfahren sind. Daher soll in dieser Projektgruppe diese Modellierung von Speicher-Hierarchien in realer konfigurierbarer Hardware vorgenommen werden.

Dazu werden die FPGA Bausteine der Virtex II Serie [5] der Firma Xilinx genutzt. Diese Bausteine bieten grundsätzlich alle Voraussetzungen, um die Effizienz von Programm-Code und Speicher-Hierarchien in Hardware zu evaluieren. Durch die ebenfalls enthaltenen zwei PowerPC Kerne und der Möglichkeit, ein PC-DIMM Speichermodul aufzunehmen, bieten sie die ideale Basis für einen solchen Hardware Entwurf.

1.2 Ziel der PG

In der Projektgruppe *V-Power* soll auf der Grundlage der oben genannten Evaluations-Hardware ein Speicher-Profiler entstehen. In seiner Realisierung stellt dieser Speicher-Profiler ein komplettes System-On-Chip dar, das aus Prozessoren, Speichern und I/O-Hardware besteht. Die Besonderheit dieses Systems ist, dass es zwischen dem physikalisch vorhandenen Speicher (z. B. einem DIMM Hauptspeicherriegel) und dem Speicher, wie er dem Prozessor bereitgestellt wird, keine direkte Entsprechung geben muß. Beispielsweise sähe der Prozessor virtuell einen Scratchpad- sowie einen gecacheten Hauptspeicher, während auf der physikalischen Seite die Speicherinhalte in einem DIMM-Modul vorgehalten werden. Entsprechend den Zugriffen auf diese virtuellen Speicher erfährt der Prozessor unterschiedliche Verzögerungen. Ferner soll die Hardware in der Lage sein, Statistiken über die Speicher-Nutzung zu sammeln; z. B. soll die Anzahl der Zugriffe pro Speichertyp, die Anzahl der Cache-Misses oder der Energieverbrauch pro Speichertyp in Hardware mitgezählt werden.

Als Ergebniss soll ein Hardware Speicher-Profiler entstehen, welcher den Programm-Code zu messender Applikationen auf den PowerPC-Kernen in realer Hardware ausführt und die konfigurierbare Logik der FPGAs nutzt, um eine weitgehend modulare und konfigurierbare Speicher-Hierarchie zu implementieren. Jeder Speicher-Zugriff der PowerPCs wird durch die FPGA-Hardware bearbeitet. Diese soll in der Lage sein, verschiedene Cache-Typen, Scratchpads und RAMs in mehreren Ebenen geschachtelt zu modellieren. Abhängig von der aktuellen Speicher-Hierarchie und einem konkreten Speicher-Zugriff sollen die FPGAs u. a. Cache-Inhalte virtuell innerhalb der FPGAs aktualisieren und Energie- und Zyklen-Zähler für alle modellierten Speicher inkrementieren. Die modellierte Speicher-Hierarchie soll einerseits dahingehend konfigurierbar sein, dass verschiedene Speicher-Typen instantiiert und in mehreren Ebenen angeordnet werden können. Andererseits soll jeder einzelne instantiierte Speicher wiederum konfigurierbar sein (z.B. Größe, Energieverbrauch, Assoziativität,...). Das im Evaluations-Board enthaltene DIMM-Modul soll als physikalischer Contai-

ner für sämtliche Programm-Codes und Daten des PowerPCs genutzt werden. Das zu realisierende System hat also die Struktur, daß der Prozessor kein Wissen von dem physikalisch vorhandenen DIMM-Modul hat. Statt dessen ”sieht” der Prozessor lediglich das Speicher-Interface, welches vom FPGA modelliert wird. Die konfigurierbare FPGA-Hardware dient als flexible Schicht zwischen Prozessor und tatsächlichem Speicher, die Speicher-Zugriffe in Hardware modelliert, protokolliert und bewertet.

Das System selber soll aus einigen Standard-Hardwarekomponenten bestehen. Diese sollen zum einen die Benutzer-Schnittstelle in Form einer VGA-Konsole und ggf. einem Audioausgang realisieren. Zum anderen sollte die Verbindung zu weiteren Geräten über eine Ethernet-Schnittstelle möglich sein. Ferner soll auf dieser Hardwareplattform ein Betriebssystem installiert werden. Den Anforderungen entsprechend hat sich hier der MontaVista Linux Kernel angeboten. Um die Funktionsfähigkeit des Systems zu demonstrieren, soll im Rahmen der Projektgruppe eine Video-Streaming Anwendung auf dem System implementiert werden. Grob skizziert sollte das System eine IP-TV Set-Top Box sein. Über das Ethernet-Interface soll ein MPEG-komprimierter Video-Stream empfangen und auf dem Bildschirm angezeigt werden können.

1.3 Mitglieder

Auf Seiten des Lehrstuhls 12 wird diese Projektgruppe betreut von

- Heiko Falk *und*
- Robert Pyka

Auf studentischer Seite besteht die Projektgruppe *V-Power* aus

- Daniel Cordes
- Arthur Falarz
- Michael Gattmann
- Sebastian Hanloh
- Matthias Jung
- Alexander Katriniok
- Jan Kleinsorge
- Alexander Milewski
- Thomas Pucyk

- Felix Rotthowe
- Florian Schmoll
- Stefan Wonneberger

Alle Teilnehmer an der Projektgruppe sind zugleich gemeinschaftlich Autoren dieses Berichtes.

1.4 Aufbau des Dokumentes

Neben dieser ersten Einleitung gliedert sich das Dokument in weitere 4 Bereiche.

Im ersten Teil von Kapitel 2 soll die Basistechnologie - das Xilinx Virtex II Pro - vorgestellt werden und etwas konkreter auf bereits von der Firma Xilinx entwickelte Konzepte zur Programmierung der FPGAs eingegangen werden. Im Abschnitt 2.3 wird dann der PG Entwurf im Konzept vorgestellt.

Das Kapitel 3 beschäftigt sich mit der VHDL basierten Entwicklung der Hardware und beschreibt unseren Entwurf bis runter auf die Register Transfer Ebene der Schaltungsentwicklung. Im nachfolgenden Kapitel werden die simulierten Speichermodule vorgestellt.

Nachdem die Hardwareumsetzung erfolgt ist, wird das Betriebssystem Linux portiert, die dazu nötigen Schritte und das benötigte Basiswissen werden in Kapitel 5 vermittelt. Nach der Portierung wird in einem separaten Kapitel auf die Treiberentwicklung zur Ansteuerung unseres Entwurfes eingegangen.

In Kapitel 7 soll unser Entwurf innerhalb einer Testumgebung überprüft werden. Dabei wird in einem ersten Schritt der VHDL Entwurf und in einem Zweiten der Gesamtentwurf auf dem FPGA getestet.

Eine Zusammenfassung soll noch einmal den aktuellen Stand über alle Bereiche der Entwicklung dokumentieren und einen Überblick über die noch anstehenden Aufgaben geben. Zudem soll ein Fazit zum enstandenen Entwurf gezogen werden.

Kapitel 2

Konzept

In diesem Kapitel soll das Basiskonzept unseres Speicherhierarchieprofilers vorgestellt werden. Dazu wird zunächst das Xilinx Virtex II Pro FPGA und das Basislayout einer PowerPC Architektur vorgestellt, so wie es mit Hilfe des von Xilinx verfügbaren EDKs [6] als Hardwareentwurf bereitgestellt wird (Abschnitt 2.2).

Im Abschnitt 2.3 wird der Hardwareentwurf der Projektgruppe vorgestellt und allgemein auf die entworfene Antwort zur PG Anforderung eingegangen.

2.1 XUP

Das Virtex-II Board stellt eine ganze Reihe von Hardware-Bausteinen und Features zur Verfügung, die dem User die Ein- und Ausgabe, Kommunikation, Debugging, Konfiguration und die Implementierung vielseitiger und mächtiger Hardware-Designs ermöglichen.

Das Kernstück dieses Boards ist das Virtex-II Pro FPGA. Dieser Hardware-Baustein kann durch den internen Flash PROM, den internen Compact Flash-Speicher oder über die externe USB-Schnittstelle konfiguriert werden.

Über das FPGA kann die Peripherie des Boards angesteuert werden. Die User LEDs, User Switches und die User Push-Button Switches können frei belegt werden. So können die LEDs z.B. dazu eingesetzt werden, den Zustand des Systems anzuzeigen. Ein AC97 Audio Ausgang sowie ein Mikrofoneingang, wie man sie aus dem PC-Bereich kennt, sind auf dem Board untergebracht. Der XSGA-Ausgang sorgt für die graphische Ausgabe, mit einer Auflösung von max. 1.92 Megapixel bei 70Hz. Das Virtex-II Board stellt eine dem IEEE-Standard entsprechende Ethernetschnittstelle (10 M/bit und 100 M/bit) zur Verfügung. Darüber hinaus bietet das Board drei serielle Ports an, einen RS-232 Port und zwei PS/2 Ports. Vier von insgesamt acht Multi Gigabit Transceivers (MGTs) stehen dem User zur Verfügung. Drei dieser MGTs sind als SATA-Schnittstellen realisiert. Wobei zwei SATA-Verbindungen als HOST-Port und eine als TARGET-Port, um eine einfache Board-zu-

Board Verbindung zu erlauben, konfiguriert sind. Das vierte MGT ist durch den User zu definieren. Der Speicher des FPGAs kann mit einem bis zu 2 GB großem DDR SDRAM DIMM Modul erweitert werden. Die Versorgungsspannung des Virtex-II Boards liegt bei 5 V und wird intern auf 3.3 V, 2.5 V, und 1.5 V runtergeregelt, um das FPGA und die Peripherie zu versorgen. Sollte die Stromversorgung nicht ausreichend sein, verfügt das Board über zusätzliche Anschlüsse. Die Taktung erfolgt durch den 100 MHz System- und 75 MHz SATA-Takt. Außer diesen gibt es eine Bereitstellung für zwei benutzerdefinierte Taktgeber.

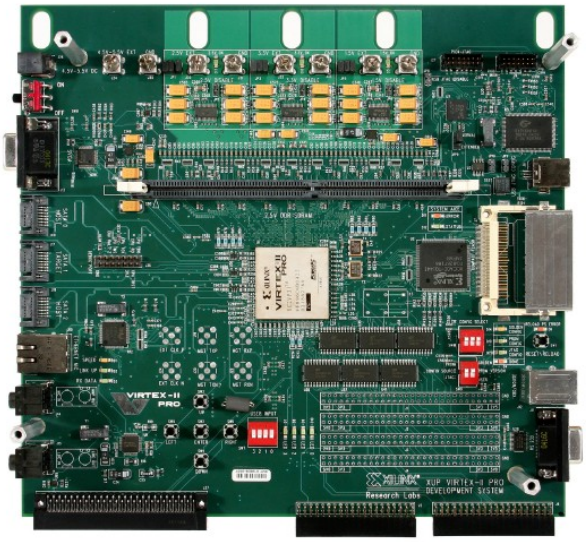


Abbildung 2.1: *Xilinx XUP Development Board mit Virtex II Pro FPGA*

Die Besonderheit des Virtex-II Boards sind die beiden PowerPCs, welche in dem FPGA integriert sind. Diese sind mit dem lokalen Block RAM des FPGAs verbunden. Der Block RAM kann beispielsweise als Scratchpad-Speicher verwendet werden. Die PowerPCs können ebenfalls, über die Busse PLB und OPB, auf alle Hardwarekomponenten zugreifen. PLB steht für Prozessor Local Bus. Über diesen Bus kann der Prozessor beispielsweise auf das DIMM Modul oder die Ethernetschnittstelle zugreifen. Andere Hardwarebausteine, wie z.B. die Audioschnittstelle, RS232-Schnittstelle oder die LEDs, sind über den On-chip Peripherie Bus (OPB) anzusprechen. Über eine Bridge sind diese Busse miteinander verbunden, so dass der PPC auch auf die an den OPB angeschlossenen Hardwarebausteine Zugriff hat.

2.2 Xilinx Basissystem

Xilinx setzt zur Nutzung des PowerPCs auf seinen FPGAs die IBM CoreConnect Architektur [7] um. Diese schreibt die Kommunikation zwischen den einzelnen Modulen auf dem Hardware Board und den darüber hinaus entwickelten IP Cores über ein Bussystem vor.

In der CoreConnect Architektur stehen drei Busse zur Verfügung, die über Brücken miteinander

verbunden sind:

- **On-chip Peripheral Bus (OPB)**
- **Processor Local Bus (PLB)**
- **Device Control Register Bus (DCR)**

Der PowerPC selbst ist an den Processor Local Bus (PLB) angebunden. Auch der PG IP Core, der die Simulation der Speicherzugriffe übernimmt, wird an diesen Bus angeschlossen. Die anderen beiden Busse sind daher für die weitere Betrachtung zunächst nicht interessant.

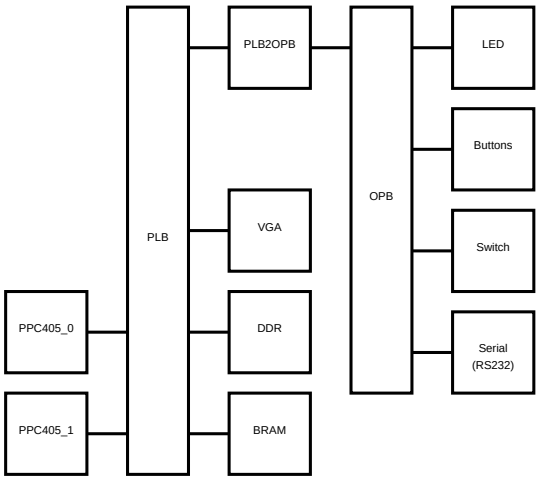


Abbildung 2.2: *Umsetzung der IBM CoreConnect Architektur auf dem Xilinx FPGA*

Abbildung 2.2 zeigt einen Teil der Umsetzung der IBM CoreConnect Architektur konkret für das verwendete Xilinx FPGA, dabei sind hier nur PLB und OPB dargestellt, sowie einige Komponenten, die auch für den Betrieb des verwendeten Embedded Linux Betriebssystems wichtig sind, wie beispielsweise VGA Controller und RS232 Schnittstelle. Für die nachfolgende Vorstellung des Konzepts für den Speicherhierarchieprofiler wird der DDR Controller von zentraler Bedeutung sein.

Neben dem DDR Controller werden auch noch weitere Komponenten mit den Entwurf einbezogen, deshalb soll im folgenden Abschnitt auf diese näher eingegangen werden, bevor dann der eigentliche IP Core Entwurf vorgestellt wird.

2.2.1 Standard Komponenten der PowerPC Architektur

2.2.1.1 PowerPC

Wie in Abbildung 2.2 dargestellt, ist der PowerPC an den Processor Local Bus angeschlossen. Über diesen und ggf. über eine Brücke zum langsameren OPB kommuniziert er mit den Ressourcen des Boards wie DIMM Speicher etc. und vor allem auch mit den individuell entwickelten IP Cores.

Die PowerPC Kerne selbst sind kein IP Core sondern reale Hardware auf dem FPGA. Sie werden jedoch von einem Wrapper umschlossen, der wiederum auch den Cache-Speicher bereitstellt. Dieser ist normalerweise für ein Betriebssystem unerlässlich und auch sinnvoll, muss aber im Fall der Speichersimulation deaktiviert werden, wie im Hardwareentwurf im Abschnitt 2.3.1 näher erläutert.

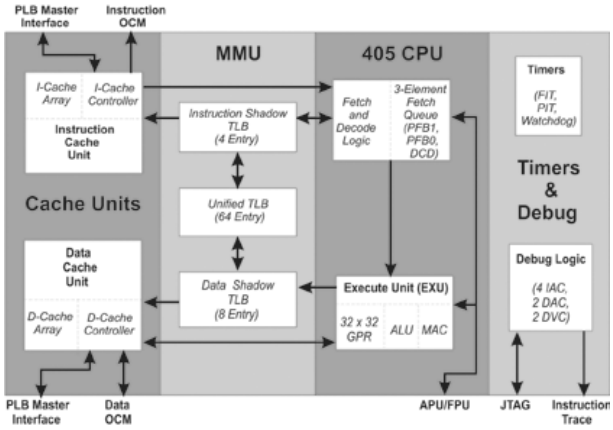


Abbildung 2.3: *Übersicht des PowerPC 405 Core*

Ferner stehen auf dem Die aber noch die Memory Managament Unit (MMU, inkl. TLB) und eine Schnittstelle zum Instruktionscache (ISOCM) und Datencache (DSOCM) zur Verfügung (Abbildung 2.3). Über letzteren wird auch die Verbindung zum PLB hergestellt, um mit den Modulen zu kommunizieren. Hierbei wird zwischen I/O Adressen und Memory Adressen unterschieden, um die Module bzw. den Speicher zu adressieren.

2.2.1.2 Processor Local Bus (PLB)

Der Prozessor Local Bus (PLB) selbst wird als IP Core realisiert [8]. Das heißt, er wird mittels FPGA Logik erstellt. Somit kann er beliebige Module anschließen aber auch ganze Module ausblenden, indem sie einfach nicht angeschlossen werden. Vor jedem Modul sitzt ein PLB-IPIF (IP Core Interface), welches zwischen Bus Logik und Modullogik (IP) vermittelt. Somit ist die Schnittstelle zum PLB standardisiert und die Module sehen nur die für sie wichtigen Ports.

Der PLB selbst ist mittels Master/Slave Steuerung realisiert (Abbildung 2.4), das heißt pro Takteinheit ist genau ein Modul Busmaster für Leseanfragen und ein Modul Busmaster für Schreiban-

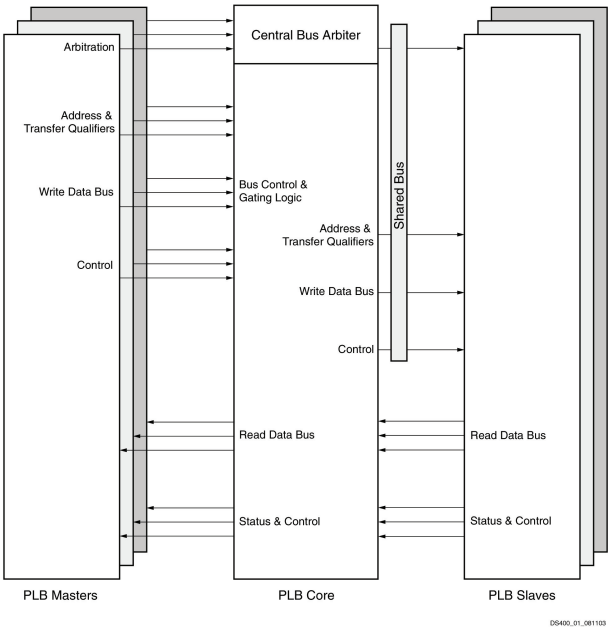


Abbildung 2.4: *PLB Interconnect Diagramm*

fragen (falls beide gleichzeitig benötigt). Manche Module, wie beispielsweise der DDR Controller (PLB_DDR) treten nur als Slave auf, da es nicht nötig ist, dass sie eigenmächtig den Bus beanspruchen. Sie werden von anderen Modulen direkt angesprochen.

Ferner versorgt der PLB die angeschlossenen Module über deren IPIF mit einem zentralen Bus Takt, dem PLB Clock. Über diesen Takt wird auch der PowerPC mit dem Bus synchronisiert, während der Prozessor selbt mit einer höheren Geschwindigkeit laufen kann. Es wurde versucht den Prozessor anzuhalten, um diesen in Abhängigkeit mit der Simulation steuern zu können, aufgrund der Aktivität des PLB war dies aber nicht möglich, nähere Ergebnisse hierzu sind im Abschnitt 3.5 zu finden.

2.2.1.3 Processor Local Bus IP Interface (PLB-IPIF)

Jedes Modul, das an den PLB angeschlossen wird, muss nach aussen (also zum PLB) über die Ports des PLB IPIF verfügen. Dass heißt, der angeschlossene IP Core enthält wiederum den PLB IPIF Core in sich selbst. Er leitet dann die Ports des IPIF nach aussen (durch eine identische Portschnittstelle) zum PLB und nutzt die für ihn wichtigen IP Ports für seine Interaktionen mit dem Bus (Abbildung 2.5). Damit wird die komplexe Bussteuerung abstrahiert.

Dabei stellt der IPIF IP Core immer alle Busfunktionalitäten zur Verfügung, der Designer des IP Cores muss entscheiden, was er davon nutzt.

Auch für den vorgestellten Entwurf wird ein PLB IPIF eingesetzt, um einen IP Core zwischen PLB und DIMM zu platzieren.

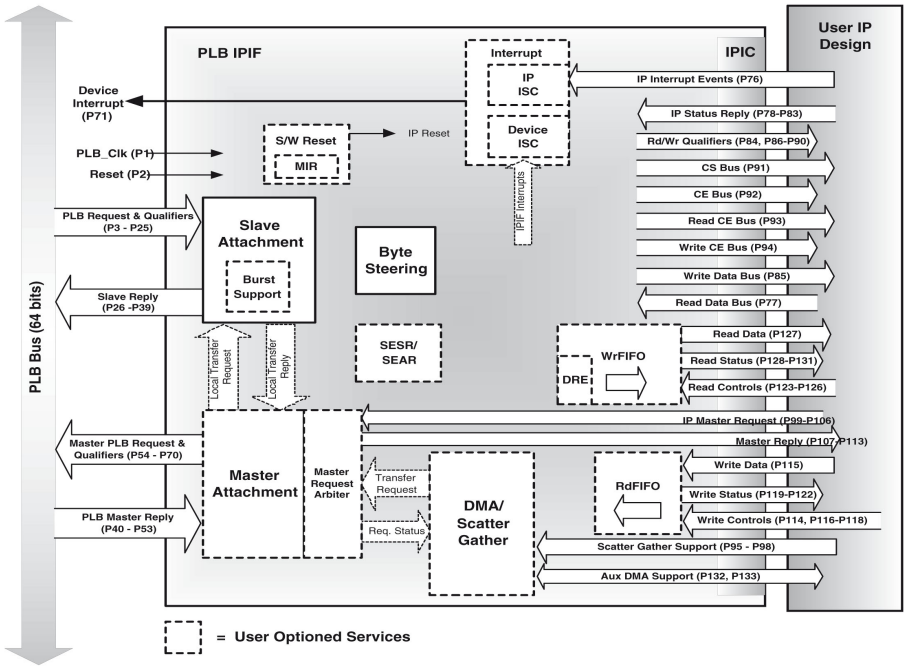


Abbildung 2.5: *PLB IPIF Übersicht*

2.2.1.4 Processor Local Bus DDR Controller (PLB-DDR)

Der PLB DDR Controller (PLB-DDR) ist das Ausgangsmodul für den vorgestellten Entwurf, denn er verbindet bereits den Bus mit dem Hauptspeicher durch einen IP Core. Er besteht wiederum nur aus maximal 3 internen IP Cores [9]:

- dem **PLB IPIF** wie im vorherigen Abschnitt beschrieben,
- ggf. einem **ECC** Modul / Register falls der DIMM dieses unterstützt
- und einem **DDR Controller**, der 0 den Hauptspeicher abstrahiert.

Abbildung 2.6 zeigt sowohl den PLB DDR Controller als auch den eigentlichen DDR Controller (rechts). Für den gezeigten Entwurf ist aber nur der linke Teil interessant, da der DDR Controller selbst nicht modifiziert werden soll. Der IP Core wurde von Xilinx so gestaltet, dass der DDR-Controller-Teil in einen eignen IP Core ausgelagert wurde. Somit besteht der PLB-DDR IP Core nur aus den drei oben genannten Teilen.

Aufgrund dieser Struktur ist der PLB-DDR Controller ideal, um die benötigte Simulationslogik zu beherbergen. Er wird daher in nachfolgenden Konzept als **V-POWER** *Memory Controller* verwendet, ergänzt um die Simulations-IP-Cores (siehe Abschnitt 2.3.1).

[1] Nur wenn der DIMM ECC unterstützt
[2] Falls 64-Bit Unterstützung gewünscht

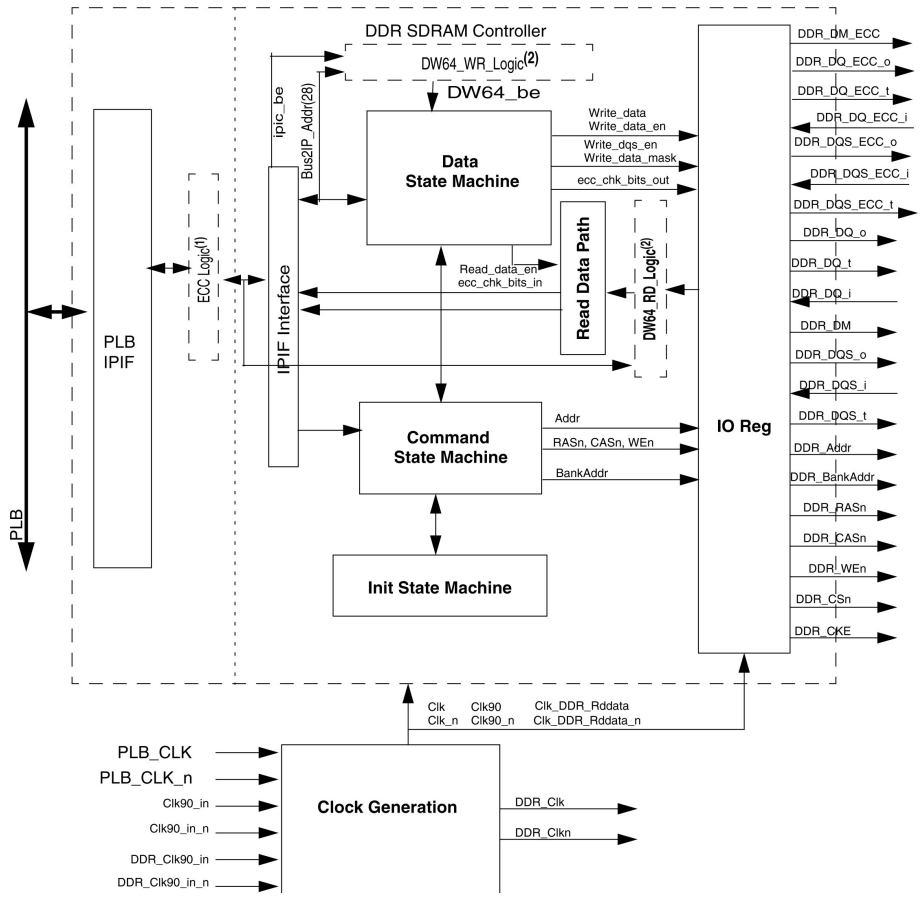


Abbildung 2.6: *PLB DDR SDRAM Controller Überblick*

2.2.1.5 Digital Clock Manager (DCM)

Als letzter wichtiger IP Core für den Simulationsentwurf ist der Digital Clock Manager (DCM) für die Takterzeugung verantwortlich. Er kann auf Basis eines 25 MHz Grundtaktes sowohl den Bustakt (100 MHz), als auch den Prozessor Takt (max. 300 MHz) generieren. Im Fall von zwei unterschiedlichen Taktraten für Bus und Prozessor werden auch zwei DCM Module eingesetzt.

Im PG Entwurf, der im nachfolgenden Abschnitt vorgestellt wird, wurde vorgesehen den Prozessor an gewissen Stellen anzuhalten. Dieser Anhaltevorgang wäre über ein Anhalten des DCMs erfolgt. Leider hat das komplexe PLB Protokoll das Anhalten des Prozessors während eines Speicherzugriffs (indem die Simulation parallel in Echtzeit erfolgen sollte) verhindert.

2.3 V-Power Speicherhierarchieprofiler

Nachdem nun alle genutzten Xilinx IP Cores vorgestellt wurden, auf die im Entwurf zurückgegriffen wird, soll in diesem Abschnitt das Gesamtkonzept des Speicherhierarchieprofilers vorgestellt werden. Auf der einen Seite wurde ein eigener Simulations-IP-Core entwickeln, welcher den Ablauf des

Speicherzugriffs überwacht, eigene Speicherbausteine simuliert und seine benötigten Daten (Simulationsdaten, echte Daten, Konfigurationen) im realen DIMM Modul ablegt. Auf der anderen Seite wurde im Betriebssystem ein Treiber für den IP Core entwickelt, welcher die Simulationshardware konfiguriert und auch die Ergebnisse auslesen kann. Dem Betriebssystem selbst ist die Simulationshardware unbekannt, da sie sich wie eine normale Speicherhierarchie aus Cache-Ebenen, Hauptspeicher und ggf. schnellen Flash bzw. Scratchpad-Speichern verhält, welche normal über physikalische Adressen und mithilfe der MMU adressiert werden.

Nach einer Vorstellung des Hard- und Softwareteils soll die Arbeit des Entwurfs anhand eines Simulationablaufes verdeutlicht werden.

2.3.1 Hardwarekonzept

Wie bereits im Abschnitt 2.2.1.4 beschrieben, wurde der gesamte Simulations-IP-Core, genannt **V-POWER** *Memory Controller*, zwischen Bus (Processor Local Bus) und dem DDR Controller platziert. Dazu wird der PLB DDR Controller von Xilinx genutzt, und dort eine Simulationsblackbox, die **V-POWER** *Memory Logic*, platziert. Diese ist zur einen Seite mit dem PLB IPIF verbunden und zur anderen Seite mit dem DDR Controller.

Abbildung 2.7 zeigt den schematischen Aufbau des IP Cores. Dabei befinden sich die Module innerhalb der **V-POWER** *Memory Logic* auf verschiedenen Ebenen, wiederum innerhalb eigener IP Cores.

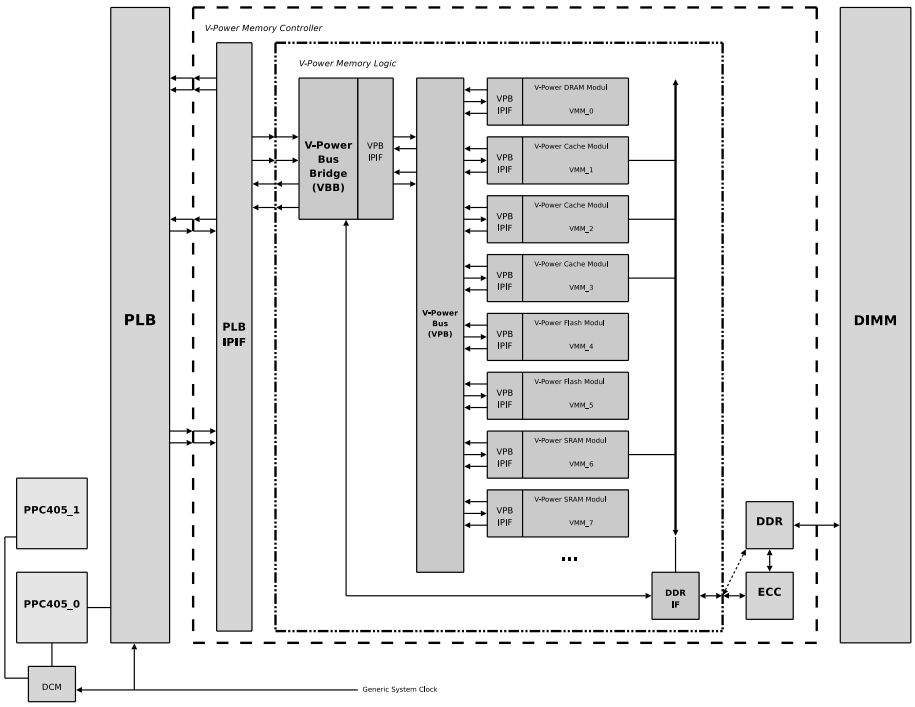


Abbildung 2.7: *Simulationshardware:* **V-POWER** *Memory Controller*

Es sollen nun zunächst die einzelnen Bestandteile des Entwurfs vorgestellt werden, bevor in Abschnitt 2.3.3 die Interaktion der einzelnen Bestandteile bei einer Speichersimulation vorgestellt wird. Die Module werden an dieser Stelle nur vom Konzept her vorgestellt, die genauere Implementierung wird im nachfolgenden Kapitel vorgestellt.

2.3.1.1 V-POWER Memory Controller

Der **V-POWER** Memory Controller ersetzt den PLB DDR Controller (siehe Abschnitt 2.2.1.4). Er enthält das benötigte PLB-IPIF, um mit dem Processor Local Bus und darüber hinaus mit dem Prozessor zu kommunizieren. Ferner wird er über den enthaltenen DDR Controller an den DIMM Hauptspeicher angeschlossen. Falls gewünscht, wird eine ECC Logik für die Fehlerkorrektur generiert. Dies kann mittels *VHDL Generics* über das EDK gesteuert werden. Generics können als Konstanten zur Entwicklungszeit verstanden werden, über die es möglich ist, wichtige Schaltungselemente mit wenigen Handgriffen aus dem gesamten Entwurf zu entfernen bzw. wieder einzubinden. Während der Berechnung der Schaltung (Synthese) wird dieses Element dann auf Gatterebene (RTL) platziert oder nicht. Abstrakt lassen sich mit Generics Abschnitte von Schaltungsquellcode (Beispiel VHDL) aktivieren bzw. deaktivieren.

Als wichtigsten Teil enthält der **V-POWER** Memory Controller den Simulations-IP-Core **V-POWER** *Memory Logic*, welcher wiederum alle benötigten Module für die Simulation enthält.

2.3.1.2 V-POWER Memory Logic

Als zentraler IP Core enthält die **V-POWER** *Memory Logic* die gesamte Simulationshardware. Sie enthält zur einen Seite eine Schnittstelle zum PLB-IPIF und zur anderen Seite ein Interface zum DDR Controller. Ausser der Signalverbindung übernimmt dieser IP-Core keine Steuerungsfunktionen, er dient lediglich der Kapselung.

Im realisieten Entwurf wurden für jeden Speichertyp eine ausreichende Anzahl an generischen Memory Modulen innerhalb der Memory Logic platziert. Diese können über den Treiber aktiviert bzw. deaktiviert und konfiguriert werden. Beispielsweise kann eine Cachehierarchie und eine Aufteilung in DRAM und Flash Speicher konfiguriert werden.

2.3.1.3 V-POWER Bus Bridge (VBB)

Die Kommunikation mit dem PLB übernimmt auf dieser Seite die **V-POWER** Bus Bridge (VBB). Sie nimmt die Anfragen vom Bus entgegen und entscheidet, ob es sich um eine Speicheranfrage an den simulierten Speicher, eine neue Simulationskonfiguration oder das Auslesen der Ergebnisse handelt.

Nachdem die Simulation abgeschlossen ist, lädt sie ggf. das reale Wort aus dem Hauptspeicher und

gibt sie über den PLB an den Prozessor.

Die Steuerung der Simulation erfolgt in der VBB mittels eines Befehlsregisters, über welche die Simulation angehalten und neu gestartet wird, sowie eine neue Konfiguration geschrieben werden kann.

2.3.1.4 V-POWER Bus (VPB)

Die Simulationshardware verfügt über einen eigenen Bus - den **V-POWER** Bus. Über diesen werden unsere Simulationsspeicher (**V-POWER** Memory Modul - VMM) mit der VBB verbunden. Dazu dient auch hier - ähnlich wie beim PLB - ein IPIF, das VPB-IPIF.

Das besondere am realisierten V-Powe Bus ist das Kommunikationsprotokoll. Da mehrere Module in Serie geschaltet sind, z.B. bei einer mehrstufigen Cache-Hierarchie, wird ein spezielles Protokoll benötigt. Es wurde ein Token-Bus-Protokoll realisiert. Dabei wird immer das nachfolgende Modul über den VPB adressiert, das heißt die Module müssen lediglich ihren Vorgänger kennen und den Adressbereich, für den sie zuständig sind. Danach ist jedes Modul eindeutig bestimmt und kann auf Anfragen reagieren. Es wird keine zentrale Arbitersteuerung benötigt. Näheres hierzu im Abschnitt 2.3.3.

2.3.1.5 V-POWER Bus IP-Interface VPB-IPIF

Das **V-POWER** Bus IP-Interface findet sich sowohl in den Memory Modulen als auch in der Bus Bridge wieder. Es übernimmt die Kommunikation mit dem **V-POWER** Bus. Dabei unterstützt es unser Token Bus Protokoll.

Dazu werden lediglich eine ID, der zuständige Adressbereich (bei VBB gesamter Adressbereich) und die Vorgänger-ID gespeichert. Danach kann das IPIF entscheiden, ob eine Anfrage für das dahinter liegende Modul gilt oder nicht. Ist dies der Fall, gibt es die Daten aufbereitet an das Modul weiter.

Weiterhin kann ein IPIF auch wiederum Daten von einem in der Simulationshierarchie dahinterliegenden Modul anfordern, indem es seine eigene ID auf den Bus legt und widerum die gewünschte Adresse. Wenn es einen Nachfolger gibt wird er darauf reagieren, anderenfalls gibt es einen Timeout.

Somit übernimmt das aktuell angesprochene IPIF immer die Buskontrolle.

2.3.1.6 V-POWER Memory Modul (VMM)

Die Memory Module stellen den eigentlichen Simulationsspeicher dar. Der IP Core selbst hat nach aussen immer dieselbe Schnittstelle zum IPIF. Innerhalb des Moduls entscheidet der Aufbau über

dessen Typ:

- DIMM Speicher
- Cache
- Flash
- Scratchpad (SRAM)

Dabei können an den VPB theoretisch beliebig viele Module eines Typs angeschlossen werden. Sie werden dann einfach innerhalb der Memory Logic instantiiert und mit dem VPB verbunden, da sie das IPIF laut Konvention enthalten müssen.

Durch das IPIF wird dann beispielsweise eine Level-3 Cache Hierarchie geschaffen, oder auch ein gecachter Flash ist denkbar. Lediglich die Anzahl der zur Synthese instantiierten Module des jeweiligen Typs entscheidet über die möglichen Simulationshierarchien.

Für den Grundentwurf stehen 3 Caches, ein Hauptspeicher, ein Flash und ein Scratchpad zur Verfügung.

Dabei verfügen alle Memory Module immer über die folgenden Eigenschaften, unabhängig von ihrem Typ:

- VPB-IPIF
- DDR Interface
- Konfigurationsregister
- Simulationsregister

Die Cache Module haben die Möglichkeit, über ein eigenes DDR-IF auf den realen Hauptspeicher zuzugreifen, um beispielsweise eine Cache-Tabelle anzulegen, etc.

In dem Konfigurationsregister werden die Parameter für das Modul, also seine ID, ggf. Nachfolger, Adressbereich(e), und speziell für Caches bspw. Assoziativität, Blockgröße etc. festgehalten.

Die zur Berechnung der Simulationsergebnisse wichtigen Daten werden in Simulationsregistern gespeichert, welche hinterher ausgelesen werden können.

Da die Memory Module selbst auf den Hauptspeicher zugreifen müssen um ihren Zustand abspeichern zu können, wird hier auch deutlich, dass es wichtig ist, dass die VBB erst nach der Simulation die realen Daten abruft, da sonst konkurrierende DIMM Zugriffe erfolgen können.

2.3.2 Treiber

Nachdem nun die Module auf Hardwareseite vorgestellt wurden, soll nun die Funktionsweise des Treibers erläutert werden.

Der Treiber wird zur Ansteuerung des Simulators benötigt, um ihn zu konfigurieren, die Simulation zu starten und zu stoppen, die gesammelten Daten auszulesen und sie zu löschen, d.h. alle Werte auf Null zu setzen. Hierfür notwendige Funktionen, die einfacher in Software als in Hardware zu implementieren sind, werden in ihn ausgelagert. Dies hat die Vorteile, dass die Implementation erleichtert wird, die für die Realisierung des Simulators notwendige Logik auf der Hardwareseite einfach gehalten werden kann und CLBs eingespart werden können. Es wird davon ausgegangen, dass auch für einen Treiber mit dem zusätzlichen Funktionsumfang ausreichend Programmspeicher vorhanden sein wird.

Für die Ausführung der simulierten Speicherzugriffe wird der Treiber hingegen nicht verwendet, da die Hardware dem Prozessor real existierende Speichermodule vorgibt, auf die ganz gewöhnlichen zugegriffen werden kann. Der dahinterliegende Hardware-Dimm ist für den Prozessor nicht mehr sichtbar.

Aus diesem Grund hat die Realisierung von Funktionen durch den Treiber statt durch Hardware keinen Einfluss auf die Simulationsergebnisse; mit einer Ausnahme: Für das Anhalten der Simulation werden Treiberbefehle ausgeführt, die veranlassen, dass die Simulation gestoppt wird und keine Daten mehr erhoben werden. Die damit verbundenen Speicherzugriffe können nicht von denen des Betriebssystems unterschieden werden und fließen mit in die Berechnung der Simuationsergebnisse ein, bis schließlich der Befehl fürs Stoppen an den Simulator gesendet wird. Es wird angenommen, dass die zu untersuchenden Programme, die auf dem Profiler ausgeführt werden, vergleichsweise viele Befehle enthalten und viele Speicherzugriffe vornehmen, so dass der Einfluss der Treiberausführung, die sich ohnehin nicht vermeiden lässt, vernachlässigbar klein sein wird. Kompensationsmaßnahmen werden daher als nicht notwendig betrachtet und nicht vorgenommen.

Für sehr kleine Programme ist es auch möglich, einen ausgewiesenen Speicherbereich zu verwenden, der nur von dem zu bewertenden Programm genutzt wird. Speicherzugriffe des Betriebssystems und der Treiber haben auf die Ergebnisse für diesen Bereich dann keinen Einfluss. Ähnlich verhält es sich, wenn die Ergebnisse aktualisiert, ausgelesen oder gelöscht werden, während die Simulation läuft. Um hier Einflüsse zu vermeiden, sollte die Simulation gestoppt werden, bevor die Aktionen durchgeführt werden, und anschließend wieder gestartet werden. Speicherzugriffe auf Speicherstellen, in denen Ergebnisse abgelegt sind, und die Ausführung der Befehle zum Aktualisieren und Zurücksetzen der Ergebnisse gehen nicht in diese ein, allerdings Speicherzugriffe für das Laden der Instruktionen.

Ansonsten erfolgt die Ausführung des Treibers nur dann, wenn keine Simulationsdaten erhoben werden. Hier wirkt sich der Treiber nur auf die für den gesamten Simulationsprozess benötigte Laufzeit aus. Es wird angenommen, dass diese von der Zeit dominiert wird, die zum Ausführen des zu unter-

suchenden Programms und der Speicherzugriffe notwendig ist, und dass die Laufzeit für den Treiber dagegen nur einen vernachlässigbaren Einfluss hat, der vom Benutzer nicht wahrgenommen wird.

Um eine initiale Konfiguration schreiben bzw. die Ergebnisse der Simulation auslesen zu können, muss der Treiber auf Konfigurations- bzw. Simulationsregister zugreifen. Ähnlich wie die übrige Peripherie am PLB können sie über spezielle Adressen angesprochen werden. Somit erhält der **V-POWER** Memory Controller einen zweiten Adressraum, der auf die einzelnen Module und die VBB in jeweils Konfigurations- und Simulationsregisteradressen aufgeteilt wird. Auf ihn kann nur der Treiber, nicht aber das Betriebssystem zugreifen. Ihm stehen aus dem Adressbereich keine Speicheradressen zur Verfügung, da sonst dort aus Versehen auch Nutzerdaten landen könnten.

2.3.3 Simulationsablauf

Die einzelnen Bestandteile des **V-POWER** Konzepts wurden nun vorgestellt. Weitere Implementierungsdetails sind im nachfolgenden Kapitel genauer erläutert. Jedoch soll nun anhand einer fiktiven Simulation inkl. vorheriger Konfiguration und anschließendem Auslesen der Ergebnisse einmal das Zusammenspiel der Memory Module erläutert werden und die Möglichkeiten der dynamischen Speicherhierarchie aufgezeigt werden.

2.3.3.1 Konfiguration schreiben

Bevor eine simulierte Anfrage an den Hauptspeicher abgegeben werden kann, müssen die Module zunächst konfiguriert werden. Es wird davon ausgegangen, dass eine vorhergehende Simulation beendet wurde und die Simulationsregister mittels eines Resets zurückgesetzt wurden. Für das Beispiel wird ein kleiner Adressraum angenommen. Verwendet werden die Adressen 0x00 - 0xFC als Adressen für den Speicher und die Adressen 0x100 - 0x14C für die Konfiguration der Module ID 0-3 mit jeweils 4 Konfigurationsregistern. Die Simulationsregister beginnen bei 0x200 bis 0x24C. Jedes Register ist 32 Bit breit und benötigt 4 Adressen (Byteadressierung, aligned). Abbildung 2.8 zeigt unsere gewünschte Konfiguration. In diesem Beispiel gibt es nur einen Cache, ein DIMM Modul und einen Flash Baustein.

Der Treiber adressiert nun die Konfigurationregister des Caches (0x110 - 0x11C) und schreibt sowohl seine Vorgänger (Predecessor) ID 0 für die VBB und den gültigen Adressbereich. In diesem Beispiel sollen die Adressen 0x00 - 0x9C auf eine Level 1 Cache Struktur gemappt werden. Die restlichen 24 Register (0xA0 - 0xFC) werden auf einen Flash-Speicher gelegt, welcher also 768 Bit groß ist.

Danach schreibt er in das DIMM Modul (ID 2, 0x120 - 0x12c) den gleichen Adressbereich, aber mit der Vorgänger ID 1, also dem Cache. Der Cache selber muss nicht wissen, dass nach ihm direkt der Dimm kommt, er weiß aber, dass zwingend noch ein Modul nach ihm kommen muss - egal ob Cache oder Dimm - da er selbst als Cache Modul realisiert ist.

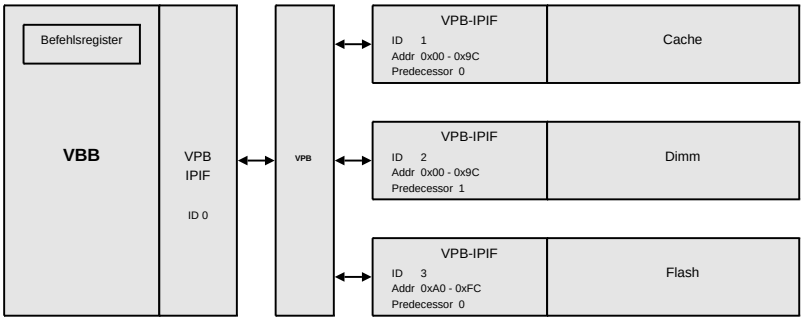


Abbildung 2.8: *Beispiel: Mögliche Konfiguration mit Level 1 Cache und Flash Speicher*

Zuletzt wird der Flash Speicher konfiguriert, in gleicher Weise wie die vorherigen, also mit seinem Adressbereich (0xA0-0xFC, immer aligned) und der Vorgänger ID 0.

Die Simulation kann nun gestartet werden, dazu wird in das Befehlsregister der VBB (0x100) der Startbefehl geschrieben.

2.3.3.2 Simuliertes Lesen aus dem Hauptspeicher mit Level 1 Cache

Ab jetzt werden alle Zugriffe des Betriebssystems auf den Hauptspeicher simuliert und erst dann gegen den realen DIMM ausgeführt. Dem Betriebssystem steht jedoch nicht der gesamte physikalische DIMM zur Verfügung. Es sind nur die Hälfte des realen Speichers gemappt, da der Rest für die Speicherung von Cache Tabellen etc. der einzelnen Module benötigt wird, also Arbeitsspeicher exklusiv für die Simulationshardware.

Es soll nun betrachtet werden, was bei einer Anfrage an die Adresse 0x00 passiert, die bis jetzt noch nicht im Level 1 Cache vorgehalten wird, um das Bus Protokoll näher kennenzulernen.

Zunächst adressiert die MMU die physikalische Adresse 0x00 auf dem Bus, daraufhin reagiert das PLB-IPIF des **V-POWER** Mem Controllers und leitet die Anfrage an die VBB weiter. Diese erkennt die Adresse als zum realen Speicher gehörig und merkt sich die Adresse zum späteren Auslesen aus dem Hauptspeicher. Bevor sie tatsächlich auf den Hauptspeicher zugreift, muss sie aber zunächst einmal den Speicherzugriff von den Speichermodulen am **V-POWER** Bus simulieren lassen. Da zu diesem Zeitpunkt noch nicht bekannt ist, wie lang dieser simulierte Zugriff dauert, hält sie den PowerPC vorsorglich an. Aus der Sicht des Prozessor soll schließlich nur so viel Zeit vergehen, wie benötigt würde, wenn die simulierte Speicherhierarchie tatsächlich vorläge. Die für das Ausführen der Simulation darüber hinaus benötigte Zeit, muss der Prozessor angehalten werden.

Damit die Speichermodule eine Simulation des Speicherzugriffs durchführen können, benötigen sie einige Informationen, die ihnen die VBB über den **V-POWER** Bus mitteilen muss. Zu diesen Informa-

tionen gehören die Speicheradresse, die Art des Zugriffs, also ob an die Speicherstelle geschreiben oder von ihr gelesen wird, als auch die aktuelle Simulationszeit. Letztere ist besonders für den simulierten DRAM wichtig, da er hieraus ermitteln muss, wie viele Refreshs er in dieser Zeit vorgenommen hätte und ob ein Refresh mit einem Speicherzugriff auf ihn zusammen fällt, wodurch dieser länger dauern würde. Auch auf den Inhalt des PreLoad-Buffers lassen sich dann Rückschlüsse ziehen. Nach einem Refresh befindet sich womöglich nicht mehr die zuletzt angefragte Zeile im Puffer, sondern die zuletzt aufgefrischte Zeile, was wiederum die Dauer des Speicherzugriffs beeinflusst.

Jede aktive Speichereinheit addiert zu der über den VPB übertragenen Anzahl an Takten ihre Anzahl an Takten hinzu, die sie auf Grund der Ergebnisse der Simulation für den Speicherzugriff bisher benötigt hätte, bevor sie die Kontrolle abgibt. Damit ist eine taktgenaue Simulation gewährleistet, wie in Abbildung 2.9 zu sehen ist. Würden z.B. pro Speichereinheit alle benötigten Zeiten zusammen gefasst werden und diese sofort beim ersten mal übertragen, wenn die Speichereinheit in dem Simulationsschritt aktiv ist, ergäbe sich für alle nachfolgend aktiven Speichereinheiten eine unterschiedliche Zeit, was in Abbildung 2.10 der Fall ist. Dies kann bei der DRAM-Speichereinheit dazu führen, dass sich der Refresh-Vorgang mit dem Speicherzugriff überschneidet, wodurch für diesen eine längere Dauer berechnet wird. Bei einer taktgenauen Simulation hätte diese Überschneidung und Verlängerung des Zugriffs aber nicht stattgefunden.

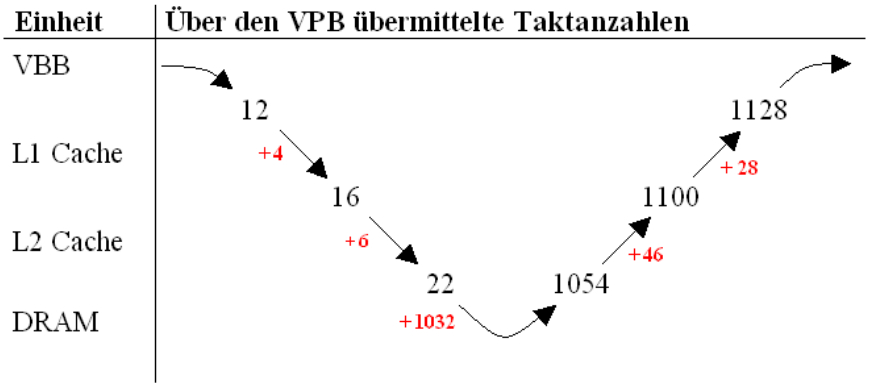


Abbildung 2.9: *Taktgenaue Simulation*

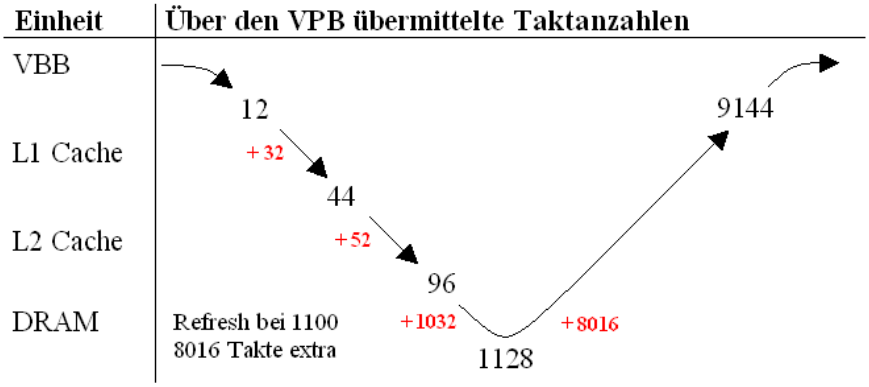


Abbildung 2.10: *Ungenaue Simulation*

Da die VBB immer zu Beginn einer Operation das sogenannte Bustoken besitzt (also das Modul ist, welches zur Zeit aktiv ist), darf sie auf den **V-POWER** Bus schreiben und legt die Adresse 0x00 sowie ihre eigene ID (0) auf den Bus. Zusätzlich legt sie die Simulationszeit, die durch die Anzahl der Prozessortakte angegeben wird, die seit Beginn der Simulation vergangen sind, auf die hierfür vorgesehenen Busleitungen. Nachdem die VBB alle oben genannten Informationen auf den Bus gelegt hat, gibt sie nun das Token frei (Nähere Details zur Umsetzung dieses Protokolls in Hardware finden sich im Kapitel 3.2.2). Dieses startet gleichsam die Simulation des Speicherzugriffs durch die Speichermodule.

Das VPB IPIF unseres Cache Moduls erkennt diese Adressierung als für ihn gültig und gibt die Anfrage an das Cache Modul, dieses lädt nun seine Cache Tabelle über seinen eigenen Anschluss an den realen DIMM aus dem Hauptspeicher und prüft die Adresse. Da sie nicht im Cache ist, adressiert nun das Cache Modul erneut den Bus (es hält im Moment das Token) und legt dieselbe Adresse, aber die ID 1 (seine eigene), auf den Bus. Nachdem es auch die Simulationszeit um den Wert erhöht hat, den der simulierte Cache bisher benötigt hätte, gibt es das Token frei,

Hierauf reagiert nun das DIMM Modul (der Simulation) und verfährt ähnlich, jedoch gibt dieses Modul sofort ein Adress Ack zurück sowie die Zeit, die dieser simulierte Zugriff gedauert hat. Um das Adress Ack zu senden, legt es seine Vorgänger ID 1 und die Adresse auf den Bus (das Token wurde ja abgegeben) und setzt das Adress-Ack. Ferner setzt es die Token-Richtung auf Reverse, um wieder das Cache Modul zu adressieren.

Dessen IPIF registriert nun die Reverse Situation und ist auf seine eigene ID sensitiv, und nicht auf die des Vorgängers. Es erhält das Adress-Ack und die simulierte Zeit. Die Simulationsregister werden aktualisiert und die neue Cache Tabelle in den Hauptspeicher geschrieben.

Nun kann das Cache Modul ebenfalls mittels Reverse seine Zeit an die VBB zurückgeben, ergänzt um die Zeit des DIMM Moduls. Mit diesen Daten lässt die VBB den PowerPC weiterlaufen und zählt die Takte. Parallel hierzu erfolgt der echte Zugriff auf den Speicher, aus dem das gewünschte Wort geladen wird. Je nachdem, welcher Vorgang länger dauert, wird entweder der PowerPC erneut angehalten, oder die VBB hält das geladene Wort zurück. Nach den errechneten Takten, die dieser Vorgang auf einer realen Architektur des simulierten Typs benötigt hätte, wird das Wort über den PLB zurückgegeben.

2.3.3.3 Ergebnisse aus Simulationsregister lesen

Das Lesen der Ergebnisse funktioniert äquivalent zum Schreiben der Konfiguration, jedoch als Lesezugriff über den Treiber. Zunächst wird jedoch die Simulation mittels Schreiben eines Befehlswortes in die Konfigurationsregister angehalten. Danach werden die Werte aus den Adressen 0x200-0x24C geladen. Da dem Treiber die Konfiguration bekannt ist (er hat sie ja vorher geschrieben), können nun die Werte eindeutig den Modulen zugeordnet und die Simulation ausgewertet werden.

Kapitel 3

Hardware

Das folgende Kapitel beschreibt die Spezifikation der Module des Profilers auf Hardware-Ebene. Im einzelnen sind dies die V-POWER Bus Bridge (VBB), der V-POWER Bus (VPB), das V-POWER Bus IP Interface (VPB-IPIF), sowie die verschiedenen Memory Module. Zum Schluss werden noch weitere IP-Cores vorgestellt, die im Rahmen der PG implementiert wurden und es wird noch beschrieben, welche Ergebnisse auf den FPGA-Boards erzielt worden sind.

3.1 V-Power Bus Bridge (VBB)

Die V-POWER Bus Bridge ist diejenige Komponente im V-POWER Speicherhierarchieprofiler, die die Speicheranfragen des Prozessors empfängt und deren Ausführung koordiniert. Sie ist aus diesem Grund sowohl mit dem Processor Local Bus, dem DDR-Controller als auch dem V-POWER Bus verbunden.

3.1.1 Spezifikation

In diesem Unterkapitel werden die Anforderungen an die V-POWER Bus Bridge beschrieben. Hierbei wird auch konkret auf die möglichen Anwendungsfälle eingegangen.

3.1.1.1 Aufgaben

Die V-POWER Bus Bridge hat im Wesentlichen die folgenden Aufgaben:

- Verarbeitung aller Eingaben, die über das PLB-IPIF erfolgen.
- Simulation starten und stoppen, d.h. Speicheranfragen entweder nur zum DIMM-Modul durchleiten, oder auch auf der Speicherhierarchie zu simulieren.

- Konfiguration der Speichermodule und Auslesen ihrer gesammelten Daten ermöglichen.
- Zugriffe auf den Speicher an das DDR-IF weiterleiten; schreibende Zugriffe nur erlauben, wenn an diese Speicheradresse auch geschrieben werden darf (Read-Only Flash).
- Den Prozessor anhalten bis (diese Funktion wird unterbunden, nähere Informationen dazu in Kapitel 3.5)
  - ein Simulationsschritt abgeschlossen ist.
  - bei lesenden Speicherzugriffen das Datenwort aus dem DIMM-Modul vorliegt.
  - die Konfiguration eines Speichermoduls erfolgt ist.
  - das Simulationsteilergebnis aus Speichermodul ausgelesen ist.
  - die Verzögerungszeit verstrichen ist.

Zusätzliche Randbedingungen:

- Aus Sicht des Prozessors sollen Speicheroperationen genau so ausgeführt werden, als wäre er direkt mit dem Speicher verbunden. Dies gilt auch bezüglich der Fehlerbehandlung z.B. bei Schreibschutzverletzungen.
- Die Anzahl der real verstreichenden Takte zwischen Beginn der Speicheroperation und Anhalten des Prozessors sollte nicht zufällig und möglichst klein sein.

3.1.1.2 Schnittstellen

Die **V-POWER** Bus Bridge kommuniziert und empfängt Daten über vier Schnittstellen.

1. Sie weist eine Schnittstelle zum PLB-IPIF auf, über das sie Anfragen für lesende und schreibende Speicherzugriffe des Prozessors erhält und Daten oder Bestätigungen aber auch Fehlermeldungen zurück gibt.
2. Sie kann mit dem DDR-IF kommunizieren, um die gewünschten Speicherzugriffe des Prozessors zu realisieren.
3. Des weiteren verfügt die VBB über zwei Leitungen, über die der Prozessor angehalten und zum Fortsetzten seiner Operationen gebracht werden kann. Wobei jedoch beachtet werden sollte, dass diese Funktionalität aus technischen Gründen nicht genutzt wird.
4. Die VBB enthält ein VPB-IPIF, das an den VPB angebunden ist, der sie mit den Speichereinheiten der Profilers verbindet.

Eine vollständige Auflistung der Ein- und Ausgangssignale der VBB befindet sich im Anhang in Tabelle A.2.

3.1.1.3 Verhaltensbeschreibung

Da die VBB Befehle wie das Starten, Anhalten, Fortfahren und Stoppen der Simulation entgegen nehmen muss, verfügt sie über ein Register mit der Adresse 0x10000000 des realen Adressraums. Die Befehle können so einfach an die VBB gesendet werden, indem sie wie Daten mittels einfacher Speicherzugriffsbefehle unter diese Adresse geschrieben werden. 0x1 codiert das Starten und Fortfahren der Simulation, 0x0 das Anhalten und Stoppen, zwischen denen jeweils kein Unterschied gemacht wird. Mit 0x4 können alle erhobenen Daten gelöscht und alle Zähler auf Null gesetzt werden (Reset).

Aus der Aufgabenstellung geht hervor, dass die Gesamtzahl der Takte ermittelt werden soll, die aus Sicht des Prozessors vergehen, wenn die Simulation läuft. Sie hat dafür ein 64Bit Register, welches für die Simulation ausreichend groß ist, da es die Takte von über 1900 Jahren Laufzeit bei einem 300MHz Prozessor erfassen kann. Da die VBB an den 100MHz Systemtakt angeschlossen ist, wird bei jedem Taktsignal, das die VBB erfährt, dieses Register um das Verhältnis Prozessortakt zu eigenem Takt erhöht. Bei einem Prozessortakt von 300MHz und einem Systemtakt von 100MHz also um 3, um die unterschiedliche Taktung auszugleichen. Das Register hat die Adresse 0x10000004 des realen Adressraums und kann mit gewöhnlichen Speicherzugriffen durch den Treiber ausgelesen werden. Da im Profiler nur die VBB weiß, wann der Prozessor angehalten sein sollte und wann er aktiviert ist, bestimmt sie diese Anzahl. Da der Prozessor nicht angehalten werden kann, wird als Approximation ein zweiter Zähler eingeführt, der die Takte zählt, zu denen der Prozessor angehalten sein sollte. Der Zähler kann an der Adresse 0x1000000C ausgelesen werden. Dieser Wert und die Zahl der simulierten Takte wird für die in Kapitel 3.5 beschriebene Approximation benötigt.

Die Konfiguration der Speicherhierarchie erfolgt, indem vom Treiber Konfigurationsdaten in Register der jeweiligen Speichereinheiten geschrieben werden. Solche Schreibzugriffe müssen von der VBB an die Speichereinheiten vom PLB-IPIF über das VPB-IPIF weitergeleitet werden. Ebenso müssen Lesezugriffe des Treibers auf die Register der Speichereinheiten, in denen die Simulationsergebnisse abgelegt sind, von der VBB durchgereicht werden, wobei eine Anpassung auf das VPB-Bus-Protokoll stattfinden muss. Dies übernimmt das VPB-IPIF.

Wenn auf die oben genannten Register oder Adressräume zugegriffen wird, sollen in jedem Fall keine Simulationsdaten von den Speichereinheiten erhoben werden. Auch für andere Zugriffe wie die des VGA-Controllers sollen keine Daten erhoben werden. Dies wird dadurch realisiert, dass ein weiterer Adressbereich zur Verfügung gestellt wird. An Hand der Adresse können dann Zugriffe, für die Daten erhoben werden sollen, von solchen unterschieden werden, für die keine erhoben werden sollen. Damit aber dennoch auf den gleichen realen Speicher zugegriffen wird, werden diese Adressen auf die korrespondierenden Adressen des normalen Adressraums abgebildet, der auch für die Simulation verwendet wird.

Insgesamt muss die VBB daher folgende Adressräume unterscheiden:

- 0x00000000 bis 0x07FFFFFF den Speicherbereich für die Simulation,

- 0x08000000 bis 0x0FFFFFFF den Speicherbereich, über den keine Simulationsdaten erhoben werden,
- 0x10000000 die Adresse des Befehlsregisters der VBB,
- 0x10000004 die Adresse des Registers für die Anzahl der in der Simulation vergangenen Prozessortakte,
- 0x1000000C die Adresse des Registers für die Anzahl der Takte, die der Prozessor angehalten wurde,
- 0x10000800 bis 0x10001FFF den Adressbereich der Konfigurationsregister sowie der Simulationsergebnisregister.

Der Adressraum 0x80000000 bis 0xFFFFFFFF bleibt den Speichermodulen vorbehalten. Hierzu wird ein Decoder eingesetzt. Aus den eingehenden Speicherzugriffsanfragen vom PLB-IPIF wird die Adresse und das Datum extrahiert und je nach Adressbereich an die betroffenen Schnittstellen weitergeleitet. Durch das Busprotokoll der VPB ist zudem immer sichergestellt, dass zu Beginn jeder Aktion die VBB sofort Daten an das VPB-IPIF senden kann. Die VBB ist zu Beginn jeder Aktion Master des Busses und beendet eine Aktion erst dann, wenn sie diesen Status wieder erlangt hat.

Konfiguration der Speicherhierarchie

Der **V-POWER** Speicherhierarchie-Profiler verfügt über verschiedenartige Speichereinheiten, die Caches, DRAM, SRAM oder Flash-Speicher simulieren können. Die Eigenschaften jeder Einheit sind über Parameter festlegbar. Dazu zählen z.B. ihre Größe, ihr Adressbereich und Zugriffszeiten. Weitere Einstellungsmöglichkeiten hängen von dem Typ des Speichers ab.

Die Festlegung sowohl der Konfigurationen der einzelnen Speichereinheiten als auch des Aufbaus der Speicherhierarchie erfolgt über den Treiber und ist nur dann möglich, wenn die Simulation angehalten ist. Der Treiber schreibt für die Konfiguration mittels gewöhnlicher Speicherbefehle Daten in die Konfigurationsregister der Speichereinheiten. Diese haben Adressen des realen Adressraums in dem Bereich 0x10000800 bis 0x10001FFF. An Hand dieser erkennt die VBB, dass Schreibzugriffe für die Konfiguration vorgenommen werden und leitet die Adressen und Daten über das VPB-IPIF an die Speichereinheiten weiter.

Zurücksetzen der Ergebnisse

Beim Zurücksetzen der Ergebnisse werden die Ergebnisregister der Speichereinheiten und des 64-Bit Zählregisters für die Anzahl der vergangenen Prozessortakte auf Null gesetzt. Zusätzlich wer-

den die Inhalte der Caches invalidiert. Die Konfiguration der Speichereinheiten bleibt hingegen erhalten. Der Treiber schreibt dazu das Datum 0x4 in das Befehlsregister der VBB, das die Adresse 0x10000000 des realen Adressraums hat. Die Ausführung des Befehls erfordert zeitintensive Zugriffe auf das DIMM-Modul, um die Cachelines zu invalidieren. Der Prozessor hätte während dessen angehalten werden müssen.

Da die modellierte Speicherhierarchie mehrere Caches enthalten kann, muss der exklusive Zugriff auf das DIMM-Modul koordiniert werden, um gleichzeitige Zugriffe mehrerer Module zu vermeiden. Die VBB wartet, bis alle Module bestätigen, das Zurücksetzen der Ergebnisse und Zustände vorgenommen zu haben und vorerst keine weiteren Zugriffe auf das DIMM-Modul auszuführen, bevor sie dem PLB-IPIF die erfolgreiche Ausführung des Befehls mitteilt. Hierdurch wird ebenfalls sichergestellt, dass andere Busteilnehmer des PLBs in dieser Zeit Speicherzugriffe auf das DIMM-Modul ausführen können.

Starten der Simulation

Das Starten und Fortsetzen der Simulation erfolgt dadurch, dass der Treiber an die Adresse des Befehlsregisters der VBB, dies ist die Adresse 0x10000000, das Datum 0x1 schreibt. Mit der folgenden positiven Taktflanke setzt die VBB den Simulationszustand auf *Simulation aktiv*. Der Befehl kann so schnell ausgeführt werden, dass ein Anhalten des Prozessors nicht notwendig ist.

Ablauf eines Simulationsschrittes

Speicherzugriffe des Prozessors auf den Adressbereich 0x00000000 bis 0x07FFFFFF, den unteren 128MB des realen Adressraums, bedürfen der Ausführung der Simulation, sofern diese gestartet und nicht angehalten wurde. Ansonsten werden sie von der VBB vom PLB-IPIF an das DDR-IF unverändert weitergeleitet und auch die Ergebnisse direkt zurückgegeben.

Bei der Simulation hätte nach Eingang der Speicherzugriffsanfrage der Prozessor gestoppt werden sollen, da zu diesem Zeitpunkt noch nicht bekannt ist, wie lange er auf Grund der Ergebnisse der Simulation auf das Ergebnis warten muss und wie lange der Simulationsaufwand selbst dauert. Aus Sicht des Prozessors sollte nur die Zeit verstreichen, die der simulierte Speicher benötigen würde. Er hätte daher so viele Takte angehalten werden müssen, wie die Simulation zur Bestimmung der Simulationswerte und für den Speicherzugriff auf das DIMM-Modul mehr benötigt. Durch das sofortige Anhalten des Prozessors wäre auf jeden Fall dafür Sorge getragen worden, dass genügend Zeit zur Ermittlung der Simulationswerte und den Zugriff auf das DIMM-Modul zur Verfügung gestanden hätte, der Prozessor davon nichts mit bekäme, keinen Unterschied zum simulierten System feststellen könnte und die Ergebnisse nicht verfälscht würden.

Weiterhin ermittelt die VBB, wie viele Prozessortakte seit Beginn der Simulation vergangen sind. Diese Anzahl und die Adresse des Speicherzugriffs, sowie die Art des Zugriffs, d.h. ob er schreibend oder lesend ist, werden über das VPB-IPIF an die Speichereinheiten übermittelt. Mit der übermittelten Adresse können die Speichereinheiten feststellen, ob sie für den Bereich zuständig sind. An Hand der Art des Zugriffs können sie die Zulässigkeit der Operation sowie die unterschiedlichen Zugriffszeiten bestimmen, die jedes Modul zu der übertragenden Anzahl an Prozessortakten hinzuaddiert.

Die VBB wartet nun so lange, bis sie wieder die Kontrolle über den VPB erhält. Erhält die VBB über das VPB-IPIF ein Error-Signal, ist während der Ausführung des Simulationsschrittes ein Fehler aufgetreten und ein Error-Signal wird auch zum Prozessor geschickt. Andernfalls kann die VBB an den Pegeln der Acknowledge-Leitungen ablesen, dass der Speicherzugriff zulässig war und erfolgreich simuliert wurde. Ebenso wird die Anzahl der Prozessortakte zurückgegeben, die nach der Ausführung des Speichzugriffes vergangen sein müssten.

Anschließend muss die Speicheroperation noch auf dem realen Speicher ausgeführt werden. Bevor das Ergebnis des Speicherzugriffs aber an den Prozessor über das PLB-IPIF weitergegeben wird, muss dieser noch mindestens so lange darauf warten, wie die simulierte Speicheroperation in Wirklichkeit gedauert hätte. Diese Dauer wird durch die Differenz der Prozessortakte, die anfangs an die Speichermodule übermittelt wurde, und derjenigen, die nach Ausführung des Simulationsschrittes zurückgegeben wird, bestimmt.

Um die Dauer eines Simulationsschrittes zu verkürzen, kann der Prozessor immer dann aktiviert und angehalten werden, wenn Zwischenzeiten feststehen. Insbesondere könnte sich dann das aktive Warten des Prozessors mit der Ausführung des Speicherzugriffs auf dem DIMM-Modul überlappen.

Anhalten der Simulation

Zum Anhalten und Stoppen der Simulation schreibt der Treiber an die Adresse 0x10000000 das Datum 0x0. Die VBB ändert dann mit der nächsten positiven Taktflanke den Simulationszustand auf *Simulation inaktiv*. Der Befehl kann so schnell ausgeführt werden, dass ein Anhalten des Prozessors nicht notwendig ist.

Auslesen der Simulationsergebnisse

Wie auch die Konfigurationsdaten, werden die Simulationsergebnisse in Registern innerhalb der Speichermodule festgehalten. Da diese über Adressen des realen Adressraums in dem Bereich 0x10000800 bis 0x10001FFF verfügen, können sie mittels einfacher Speicherlesebefehle ausgelesen werden. An der Adresse des Lesezugriffs erkennt die VBB, dass kein Speicherzugriff auf das

DIMM-Modul sondern auf die Ergebnisregister stattfindet und liest über das VPB-IPIF das angefragte Register aus. Anschließend gibt sie das Datum über das PLB-IPIF an den Prozessor weiter. Werden die Simulationsergebnisse der Speichermodule von der VBB ausgelesen während die Simulation läuft, hätte der Prozessor während der Bearbeitung der Anfrage durch die VBB angehalten werden sollen, so dass für ihn nur minimal viele Takte vergangen wären.

Zusätzlich enthält die VBB zwei 64Bit-Register. Aus dem ersten Register kann die Anzahl aller während der Simulation vergangen Prozessortakte ausgelesen werden. Das obere Wort des ersten Registers hat die Adresse 0x10000004 des realen Adressbereichs, das untere Wort die Adresse 0x10000008. Das zweite Register enthält dagegen nur die Anzahl der Prozessortakte, zu denen der Prozessor angehalten sein sollte. 0x1000000C ist die Adresse des oberen Wortes des zweiten Registers, 0x10000010 ist die Adresse des unteres Wortes.

3.1.2 Implementierung

In diesem Unterkapitel wird zunächst der Aufbau der VBB erläutert, bevor auf die Funktionsweise einzelner Komponenten und ihr Zusammenspiel eingegangen wird.

Ein detailiertes RTL Schema der VBB befindet sich im Anhang B.1. Es zeigt alle verwendeten Komponenten und wie diese miteinander verbunden sind. Auch angrenzende Komponenten, mit denen die VBB über ihre Schnittstellen kommuniziert, sind dargestellt. Auf Grund seiner Größe wurde das gesamte Schema in vier Teile zerlegt.

3.1.2.1 Komponenten der VBB

Die VBB ist die einzige Schnittstelle des Speicherhierarchieprofilers zum Prozessor, d. h. eigentlich zum Prozessor Local Bus (PLB), und damit auch zum Nutzer. Sie muss alle Anwendungsfälle abdecken und dabei mit mehreren anderen Komponenten des Speicherhierarchieprofilers kommunizieren. Wäre die gesamte Funktionalität, die die VBB hierfür zur Verfügung stellen muss, in einem einzigen VHDL-Modul implementiert worden, hätte dies sicherlich zu großer Unübersichtlichkeit geführt. Statt dessen wurde das Aufgabenspektrum in logisch zusammengehörige Bereiche unterteilt, die von Teilkomponenten eigenständig gelöst werden können.

Die Unterscheidung der Anwendungsfälle nimmt der VBB-Use Case Decoder vor. An Hand der adressierten Speicherstelle und des Simulationszustandes entscheidet er, ob eine Konfiguration der Speichermodule, ein zu simulierender oder ein nicht zu simulierender Speicherzugriff erfolgt, ob Ergebnisregister der VBB oder der Speichermodule ausgelesen werden, oder ob der VBB ein Befehl erteilt wird. Der Decoder verfügt über einige Ausgangsleitungen über die er dem Steuerwerk den vorliegenden Anwendungsfall mitteilt und den Datenfluss steuert. Die Steuerung des Datenflusses durch den Decoder hat den Vorteil, dass das Steuerwerk z. B. nicht zu unterscheiden braucht, welches ihrer Ergebnisregister ausgelesen wird. Somit stellt nicht jedes Auslesen eines Ergebnisses einen eigenen Anwendungsfall dar. Einige Fälle können verallgemeinert und zusammengefasst

werden. Dies ermöglicht eine Vereinfachung des Steuerwerkes.

Für die Kommunikation mit den Speichermodulen über den **V-POWER** Bus wird das VPB-IPIF verwendet. Dieses verbirgt den **V-POWER** Bus mit den daran angeschlossen Speichermodulen vor der VBB, so dass aus ihrer Sicht ein einzelner Speicherbaustein vorliegt, auf den mit gewöhnlichen Lese- und Schreibanfragen zugegriffen werden kann. Auf diese Weise benötigt die VBB keine Kenntnis über die vorhandenen Speichermodule und deren Ansteuerung. Eine ausführliche Beschreibung des VPB-IPIFs wird in dem Kapitel 3.3 vorgenommen.

Die Ausführung der Speicherzugriffe auf das DIMM-Modul erfolgen mit Hilfe des Speicherzugriffsmoduls **vpower_mem_access**, sofern die Zugriffe des PLB-IPIFs nicht direkt an den Speicherbaustein durchgeleitet werden können. Auch hier werden die Details der Kommunikation mit dem Speicher versteckt und aus Sicht der VBB eine vereinfachte Schnittstelle bereitgestellt. Im Kapitel 3.4.2 wird dieses Modul in seiner Funktionsweise und seinem Aufbau erklärt.

Wo es sich zwecks einer einfacheren Realisierung anbot Funktionen verschiedener Aufgabenbereiche durch eine Komponente bereitzustellen, wurde dies auch vollzogen. Wie zuvor schon kurz erwähnt wurde, gibt es in der VBB ein zentrales Steuerwerk, den VBB-Controller, der das Zusammenwirken der Teilkomponenten der VBB koordiniert. Gleichzeitig übernimmt dieser auch die Kommunikation mit dem PLB-IPIF.

Insgesamt setzt sich die VBB aus diesen VHDL-Modulen zusammen:

- **vpower_vbb.vhd**
- **vpower_vbb_controller.vhd**
- **vpower_vbb_useCaseDecoder.vhd**
- **vpower_accu.vhd**
- **vpower_flipflop.vhd**
- **vpower_reg.vhd**
- **vpower_vpb_ipif.vhd**
- **vpower_mem_access.vhd**

Neben den bereits genannten VHDL-Modulen tauchen in dieser Auflistung noch nicht erwähnte Komponenten auf. Dazu zählt das VHDL-Modul **vpower_accu.vhd**. Es realisiert einen Akkumulator, wovon in der VBB zwei Instanzen verwendet werden. Eine zählt die Anzahl der Prozessortakte, die während der Simulation vergehen. Da der Prozessor um ein Vielfaches schneller getaktet ist als die Logikbausteine im FPGA, die für die Umsetzung des Speicherhierarchieprofilers einschließlich der VBB in Hardware verwendet werden, hätte ein einfacher Zähler nicht ausgereicht. In jedem Takt muss schließlich der Zähler um die Anzahl der Prozessortakte erhöht werden, die während eines Taktes der Logikbausteine des FPGAs vergehen, und nicht nur um Eins.

Die zweite Instanz des Akkumulator wird dazu verwendet, um die Anzahl der Takte zu zählen, zu denen der Prozessor während der Simulation angehalten sein sollte, tatsächlich aber weiterläuft. Sein Wert wird bei Abschluss einer simulierten Speicheroperation um die Differenz der aktuellen Anzahl an Prozessortakte und der Sollprozessortaktanzahl erhöht. Der Spezifikation entsprechend weisen beide Akkumulatoren, die auch als die Ergebnisregister der VBB bezeichnet werden, eine Breite von 64 Bit auf. Sie sind an den Datenpfad angebunden, damit sie ausgelesen werden können.

Das Flipflop, das im VHDL-Modul **vpower_flipflop.vhd** implementiert wird, wird lediglich dazu verwendet, den Simulationszustand zu speichern. Da die Simulation zu jedem Zeitpunkt entweder nur gestartet oder angehalten sein kann, reicht dieser 1-Bit-Speicher hierfür aus.

Um die Simulationszeit und Daten zwischenspeichern zu können, werden Speicher für mehrere Bits benötigt, die aber kein Zurücksetzmechanismus aufweisen müssen. Ein solches Register mit frei wählbarer Bit-Breite wird durch das VHDL-Modul **vpower_reg.vhd** beschrieben. Die Daten, die von dem VPB-IPIF, den Akkumulatoren oder dem Speicherzugriffsmodul ausgegeben werden, werden in einem 64-Bit-Register zwischengespeichert, bevor sie an das PLB-IPIF weitergegeben werden. Auf diese Weise wird gewährleistet, dass die Daten stabil anliegen. Aus dem gleichen Grund und noch einem weiteren wird auch die Simulationszeit zwischengespeichert, bevor sie an das VPB-IPIF übertragen wird. Das Register weist ebenfalls eine Breite von 64 Bit auf, und dient zusätzlich dazu, die vom dem VPB-IPIF ausgegebene Simulationszeit mit der eingegebenen vergleichen zu können. Aus der Differenz der beiden Zeiten wird ermittelt, wie lange der Prozessor während der Simulation auf den Abschluss einer Speicheroperation warten muss.

Ein drittes nur 4-Bit großes Register ist das Befehlsregister der VBB. Da sich die drei möglichen Befehle Start, Stopp und Zurücksetzen nur in den vier am wenigsten signifikanten Bits unterscheiden, werden auch nur diese Bits der Befehlsworte gespeichert und ausgewertet. Wenn das im Befehlsregister befindliche Befehlswort ausgelesen wird, werden die 28 signifikantesten Bits mit Nullen aufgefüllt.

Neben diesen Komponenten, die durch eins der in der Auflistung genannten VHDL-Module beschreiben werden, wurden noch zahlreiche andere Komponenten verwendet, die auf Grund ihrer in VHDL einfach zu beschreibenden Funktionsweise nicht in eigenen VHDL-Dateien implementiert wurden. Hierzu zählen Multiplexer, Tri-State-Buffer, Subtrahierer, Vergleicher und andere primitive Logikbausteine wie z. B. UND-Gatter.

3.1.2.2 Implementierung der Anwendungsfälle

Der Use Case Decoder der VBB unterscheidet insgesamt acht Anwendungsfälle, wovon in der folgenden Auflistung vier Fälle, die das Auslesen der unterschiedlichen Worte aus den Simulationsregistern zu einem Anwendungsfall zusammengefasst wurden:

- normaler Speicherzugriff

- Speicherzugriff, der eine Simulation erfordert
- Zugriff auf ein Speichermodul
- Ausführung eines Befehls
- Auslesen eines Simulationsregisterwortes der VBB

In dem ersten Fall wird zwischen dem PLB-IPIF und dem DDR-Controller eine direkte Verbindung hergestellt, so dass normale Speicherzugriffe nicht verzögert werden und die übrigen Komponenten der VBB sie nicht wahrnehmen. Der VBB-Controller unterscheidet daher nur noch zwischen den letzten vier Anwendungsfällen, fügt aber noch den Anwendungsfall der ungültigen Operation hinzu. In diesem wird lediglich ein Fehlersignal an das PLB-IPIF ausgegeben.

Das Vorgehen in den übrigen Anwendungsfällen wird nun an Hand des endlichen Automaten des DDR-Controllers erläutert. Er gibt vor, wann welche Komponente wie angesteuert wird. Daher eignet sich seine Betrachtung, um das Zusammenwirken der Komponenten und die Abfolge der nötigen Einzelschritte zum Ausführen einer Aktion strukturiert zu beschreiben.

Abbildung 3.1 zeigt eine starke Vereinfachung des Zustandsdiagramms des VBB-Controllers. Das Zustandsdiagramm ist sehr umfangreich, so dass es in mehreren Abbildungen dargestellt wird, die jeweils nur die für den Anwendungsfall relevanten Zustände und Übergänge enthalten. Auf die Angabe der Ausgabefunktion wurde verzichtet, es werden aber ihre wesentliche Wirkung beschrieben. Zu Beginn jeder Aktion befindet sich der Automat immer im Startzustand *Start*.

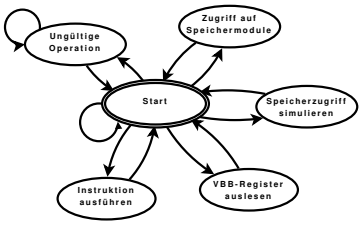


Abbildung 3.1: *Stark vereinfachte Übersicht über das Steuerwerk der VBB*

Simulation eines Speicherzugriffes

Die Simulation eines Speicherzugriffes lässt sich grob in zwei Teilschritte unterteilen. Zum einem muss der Zugriff von den Speichermodulen simuliert werden, zum anderen muss der Zugriff auf den Speicher auch tatsächlich ausgeführt werden. Aus der Sicht des Prozessors soll dabei nur die Zeit vergehen, die ein Zugriff auf den simulierten Speicher benötigen würde. Zu Beginn des Speicherzugriffs steht diese Dauer noch nicht feststeht. Es ist aber zu erwarten, dass die Simulation und die Ausführung des Speicherzugriffes auf den DRAM häufig länger dauern wird, als ein Zugriff auf den

simulierten Speicher, wenn dieser tatsächlich vorläge, gerade wenn Caches oder andere schnellere Speicher simuliert werden. Vorsorglich wird deshalb zu aller erst im Zustand *FreezeSimTime* versucht den Prozessor anzuhalten.

In der zur Zeit vorliegenden Version des Speicherhierarchieprofilers ist ein Anhalten des Prozessors mit anschließender Wiederaufnahme seiner Tätigkeit nicht möglich. Dies wurde beim Entwurf der VBB schon berücksichtigt, so dass der Stopp des Prozessors keine Prämisse für die folgenden Schritte ist. Das Fehlen der Möglichkeit zum Anhalten des Prozessors kann die VBB aber nur sehr begrenzt kompensieren, da sie lediglich die Zeit, die der Prozessor weiterläuft, obwohl er angehalten sein sollte, mit seiner Wartezeit auf den Abschluss der Speicheroperation verrechnet.

Auf Grund der längeren Dauer eines simulierten Speicherzugriffes wird ebenfalls gleich zu Beginn einer solchen Speicheroperation im Zustand *FreezeSimTime* der TimeOut des PLB-IPIFs unterdrückt. Dieser ist von vornherein auf 64 Takte festgelegt. Wird die Operation innerhalb dieser Zeitspanne nicht abgeschlossen, wird sie von dem PLB-IPIF abgebrochen und ein Fehlersignal generiert. Dies soll verhindern, dass während der Erprobungsphase eines neuen Systems durch eine Fehlfunktion der PLB dauerhaft belegt wird, was zu einem Stillstand des Gesamtsystems führen kann. Bei der Simulation und Ausführung der Speicherzugriffe besteht hingegen die Gefahr, dass diese Funktion des PLB-IPIFs Operationen vorzeitig abbricht, obwohl kein Fehlverhalten vorliegt. Aus diesem Grunde wird sie während der Speicherzugriffe deaktiviert.

Als nächstes muss nun der Speicherzugriff durch die Speichermodule simuliert werden. Dazu müssen alle für die Simulation relevanten Informationen über den **V-POWER** Bus übertragen werden. Um dem VPB-IPIF stabile Eingaben garantieren zu können, obwohl sich die Simulationszeit, als eine der Eingaben, während der Simulation mit jedem Takt ändert, wird noch im Zustand *FreezeSimTime* der Wert des Simulationszeitakkumulators zwischengespeichert und somit auf den Wert der Simulationszeit zu Beginn der Speicheroperation eingefroren.

Zu Beginn jeder Speicheroperation sollte die VBB eigentlich immer Master des VPBs sein und sofort die Signale an ihn anlegen dürfen, die zu einer Ausführung der Simulation durch die Speichermodule führen, was im Zustand *SimOp* erfolgt. Sollte aber wider Erwarten ein sofortiger Zugriff auf den Bus nicht möglich sein, wartet der VBB-Controller im Zustand *SimOpButVPBNotReady* solange, bis ihm der Zugriff auf den Bus gewährt wird, um im Zustand *SimOp* fortzufahren.

Im Zustand *VPB_SimOp* wartet der VBB-Controller so lange, bis der vom VPB-IPIF die Bestätigung erhält, dass die Simulation des Speicherzugriffs von den Speichermodulen erfolgreich vollzogen wurde. Sollte während dieses ersten Teils der Ausführung des simulierten Speicherzugriffes ein Fehler am VPB auftreten, wechselt der VBB-Controller in den Zustand *VPB_Error*, generiert eine Fehlermeldung und bricht die Operation ab.

Wurde der Speicherzugriff erfolgreich simuliert, steht nun fest, wie lange der Prozessor auf den Abschluss des Speicherzugriffs warten müsste, wenn der simulierte Speicher tatsächlich vorläge. Jetzt muss noch die Ausführung des Zugriffs auf dem DIMM-Modul erfolgen. Dazu wechselt der Controller in den Zustand *SimOp_Succesful* und führt mit Hilfe des Speicherzugriffsmoduls den vom

Prozessor gefordert Zugriff auf den Speicher aus.

Muss der Prozessor laut Simulationsergebnis auf den Abschluss der Speicheroperation warten, wechselt der VBB-Controller in den Zustand *MemOp*. In diesem wartet er auf das Ergebnis der Speicheroperation und startet wieder den Prozessor, sofern er angehalten werden konnte, damit dieser schon Wartezeit verstreichen lassen kann. Ist der Speicherzugriff beendet, bevor der Prozessor die gesamte Wartezeit erfahren hat, wechselt der VBB-Controller in den Zustand *ProcessorActiveWait* und verharrt in diesem, bis die gesamte Wartezeit des Prozessors verstrichen ist. Erst dann wechselt er in den Zustand *MemOpFinish*. Sollte das Ende des Speicherzugriffs mit dem Ende der Wartezeit zusammenfallen, wechselt der Controller direkt von *MemOp* in diesen Zustand.

Sollte der Speicherzugriff auf das DIMM-Modul allerdings länger dauern, als der Prozessor laut Simulation auf die Ergebnisse warten soll, wechselt der VBB-Controller in den Zustand *WaitForMemAck*. Um dies festzustellen, wird in jedem Takt vom aktuellen Wert des Akkumulators, der die Anzahl der Prozessortakte zählt, an denen der Prozessor aktiv war, die Anzahl der Prozessortakte subtrahiert, die der Simulation zur Folge nach dem Speicherzugriff vorliegen soll. Ist die Differenz größer Null, muss der Prozessor erneut angehalten werden. Genau dies erfolgt wird im Zustand *WaitForMemAck*. Nachdem die Ausführung der Speicheroperation auf dem DIMM-Modul erfolgreich beendet wurde, wechselt der VBB-Controller in den Zustand *MemOpFinish*. Das Signal mit dem die erfolgreiche Ausführung der Speicheroperation bestätigt wird, wird gleichzeitig auch dazu verwendet, um dem Pufferregister, das die Daten zwischenspeichert, die an das PLB-IPIF ausgegeben werden sollen, zu signalisieren, dass es die anliegenden Daten übernehmen soll. So liegen im nächsten Takt schon die richtigen Daten an den Ausgängen des Registers an.

Sollte zu irgendeinem Zeitpunkt während der Ausführung der Speicheroperation ein Fehler auftreten, wechselt der VBB-Controller in den Zustand *Mem_Error*. In diesem erzeugt er ein Fehlersignal und bricht die simulierte Speicheroperation ab. Kann die Speicheroperation hingegen erfolgreich ausgeführt werden, wird der Zustand *MemOpFinish* erreicht. Befindet sich der VBB-Controller in diesem Zustand, hebt der Controller das Anhalten des Prozessors und die Deaktivierung der TimeOut-Funktion des PLB-IPIFs auf. Zusätzlich werden die Ausgaben für das PLB-IPIF erzeugt, die eine erfolgreiche Ausführung der Speicheroperation bestätigen. Außerdem wird die Ausgabe des Subtrahierers, die Differenz zwischen der aktuellen Simulationszeit und der Simulationszeit, die nach den Berechnungen der Speichermodule vorliegen sollte, auf den bestehenden Wert des Akkumulators addiert, der die Anzahl der Prozessortakte speichert, die der Prozessor zu lange auf das Ergebnis einer Speicheroperation gewartet hat. In folgenden Takt nimmt der Controller wieder den Startzustand *Start* ein. Die Simulation des Speicherzugriffes ist damit beendet.

Zugriff auf ein Speichermodul

Die Ausführung eines Speicherzugriffs auf die Konfigurations- oder Ergebnisregister der Speichermodule erfolgt in ähnlicher Weise wie die Ausführung eines simulierten Speicherzugriffes. Die Un-

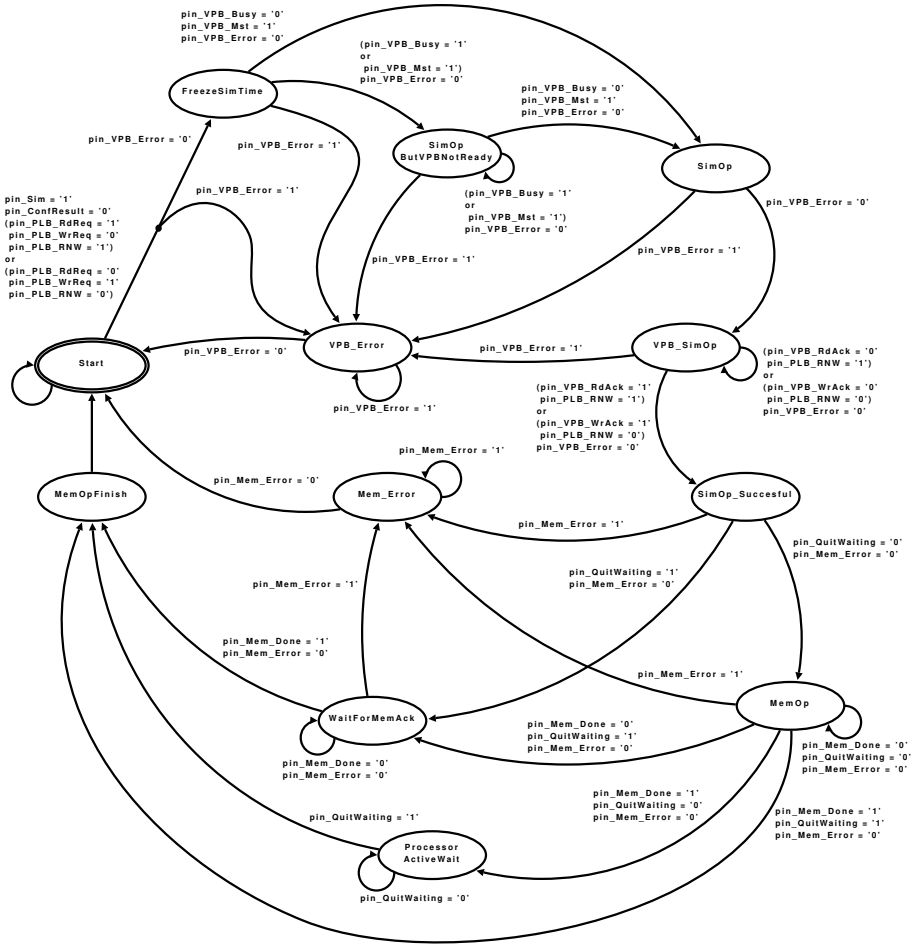


Abbildung 3.2: *Für die Ausführung einer simulierten Speicheroperation relevanter Teil des Zustandsdiagramms*

terschiede bestehen darin, dass vor der Kommunikation mit dem betroffenen Speichermodul die Simulationszeit nicht zwischengespeichert wird und anschließend kein Zugriff auf das DIMM-Modul erfolgt. Schließlich müssen bei einer Leseoperation nicht die Daten, die vom Speicherzugriffsmodul ausgeben werden, sondern die Daten, die das VPB-IPIF ausgibt, im Pufferregister zwischengespeichert werden. Da die Daten zusammen mit der Bestätigung für die erfolgreiche Ausführung der Operation ausgegeben werden, wird dieses Bestätigungssignal genutzt, um die Daten in das Pufferregister zu laden.

Zur nächsten Taktflanke hat das Pufferregister die Daten zwischengespeichert und der VBB-Controller befindet sich im Zustand *ModuleOp_Succesful*, in dem er die Operation abschließt. Die Bestätigungssignale für das PLB-IPIF werden gesetzt, der Timeout des PLB-IPIF wird nicht weiter unterdrückt und der Prozessor reaktiviert. Die Operation ist schließlich abgeschlossen, wenn der VBB-Controller zum nächsten Takt in den Ausgangszustand *Start* gewechselt hat.

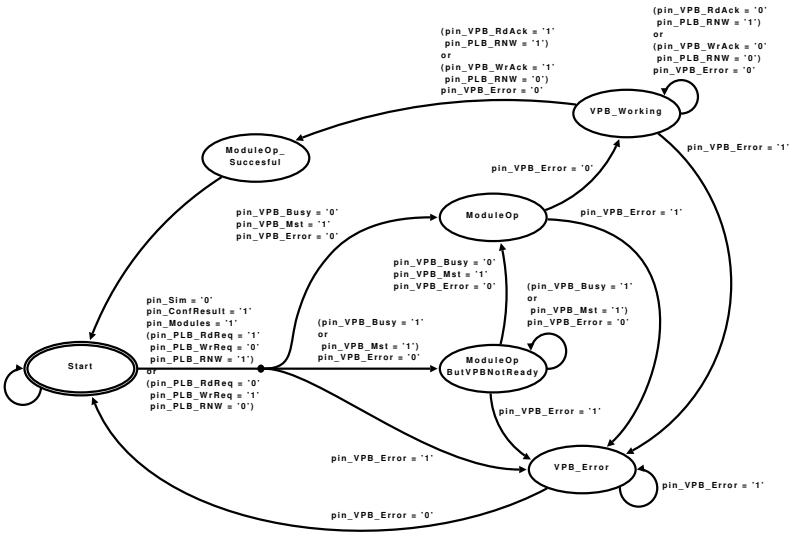


Abbildung 3.3: *Für den Zugriff auf ein Speichermodul relevanter Teil des Zustandsdiagramms*

Ausführung eines Befehls

Wird ein Datum an die Adresse 0x10000000 geschrieben, wird dieses als Befehl interpretiert. Zuerst wird im Zustand *LoadInstruction* die 4 am wenigsten signifikantesten Bits in das Befehlsregister geladen. Der VBB-Controller wartet einen Takt im Zustand *InstructionDecode*, bis die Daten im Register gespeichert sind.

Entspricht das im Befehlsregister befindliche Datum keinem der drei zulässigen Befehlsworte für das Starten, Anhalten oder Zurücksetzen der Simulation, wechselt der VBB-Controller im nächsten Takt in den Zustand *InvalidRequest* und generiert in diesem eine Fehlermeldung. Schließlich wechselt der Controller wieder in den Ausgangszustand *Start*.

Wurde in das Befehlsregister der Befehl für das Starten der Simulation geschrieben, so geht der Controller für einen Takt in den Zustand *StartSim* über, bevor in den Ausgangszustand wechselt. Wie in der Spezifikation angekündigt, erfolgt das Starten der Simulation sehr schnell. Im Zustand *StartSim* wird nur das Flipflop gesetzt, das den Simulationszustand speichert.

Ganz ähnlich erfolgt das Anhalten der Simulation. Statt in den Zustand *StartSim* wechselt der Controller in diesem Fall in den Zustand *StopSim*. Hier wird das Flipflop, das den Simulationszustand speichert, zurückgesetzt.

Das Zurücksetzen der Simulation erfordert schon mehr Aufwand, nicht nur die VBB muss ihre Simulationsregister auf Null setzen, sondern auch die Speichermodule müssen zurückgesetzt werden. Speichermodule, die Caches repräsentieren, müssen ihre Daten, die im DIMM-Modul abgelegt sind, auf die initialen Werte setzen. Dieser Vorgang ist mit mehreren Speicherzugriffen verbunden, weswegen nach dem Decodieren des Befehls in den Folgezuständen *ResetInstr* und *ResetButVPBNotReady* sofort versucht wird, den Prozessor anzuhalten.

Der Zustand *ResetButVPBNotReady* wird nur angenommen, falls entgegen den Vorgaben ein Zugriff auf den VPB nicht sofort möglich sein sollte. Ist der Bus bereit Daten von der VBB anzunehmen, wird schließlich in den Zustand *ResetInstr* gewechselt. In diesem werden die Signale so gesetzt, dass die Speichermodule reihum ihre Simulationsregister auf Null setzen und ihre Zwischenspeicher invalidieren. Ihre Konfiguration bleibt hingegen erhalten.

In dem Zustand *ResetInstr* verharrt der VBB-Controller so lange, bis er vom VPB-IPIF die Bestätigung erhält, dass alle Speichermodule zurückgesetzt sind. Erhält er die Bestätigung, nimmt der VBB-Controller den Zustand *ResetDone* an. Jetzt erst werden die Ergebnisregister der VBB auf Null gesetzt. Da ein Zurücksetzen der Simulation auch dann möglich ist, wenn die Simulation aktiviert wurde, akkumulieren die Zähler während des länger andauernden Zurücksetztvorgangs der Speichermodule fortlaufend die Zahlen der vergangenen Prozessortakte. Würde das Löschen der Simulationsergebnisse bereits in einem Vorgängerzustand erfolgen, wären die Werte der Simulationsregister eindeutig von Null verschieden.

Abschließend hebt der Controller das Anhalten des Prozessors und die Deaktivierung der TimeOut-Funktion des PLB-IPIFs auf. Im nächsten Takt befindet sich der VBB-Controller wieder im Ausgangszustand *Start* und die Bearbeitung des Befehls ist abgeschlossen.

Es ist aber denkbar, auch wenn dieser Fall nicht eintreten sollte, dass das VPB-IPIF ein Fehlersignal ausgibt. Dies führt dazu, dass der VBB-Controller zunächst in den Zustand *VPB_Error* wechselt, wo ein Fehlersignal erzeugt und die Ausführung des Befehls abgebrochen wird, bevor er im Ausgangszustand *Start* fortfährt.

Das Auslesen des Befehlsregisters wird wie das Auslesen eines Wortes aus den Simulationsregistern gehandhabt.

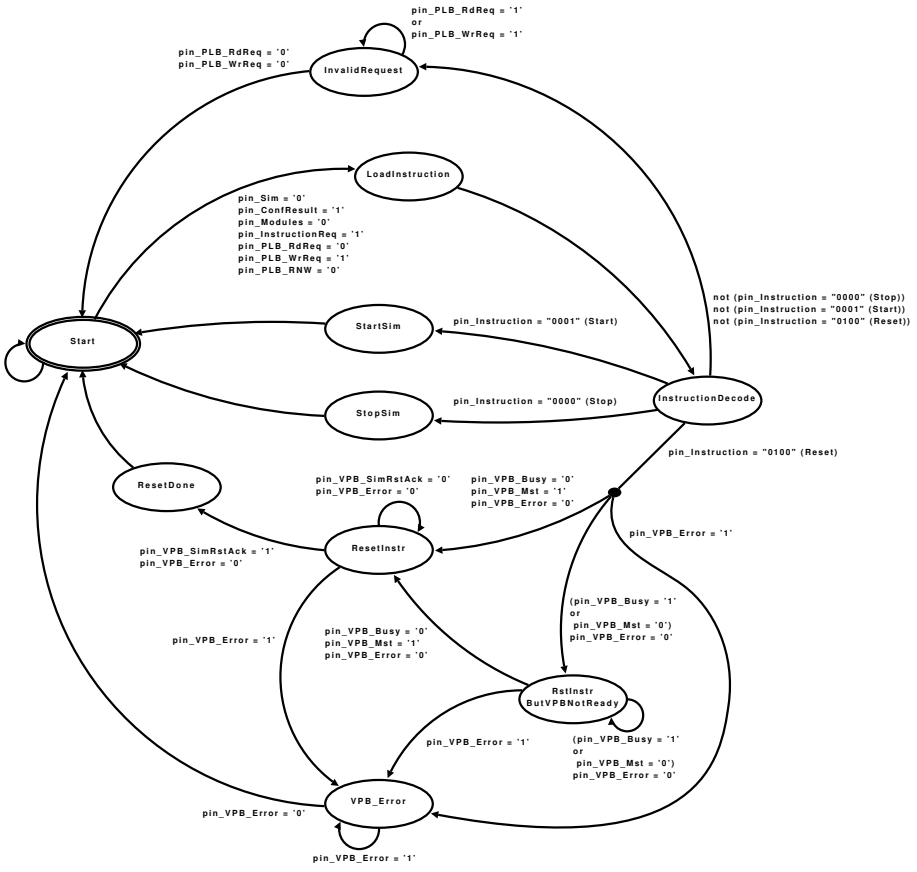


Abbildung 3.4: *Für die Ausführung einer Instruktion relevanter Teil des Zustandsdiagramms*

Auslesen eines Simulationsregisterwortes der VBB

Soll ein Registerwert der VBB ausgelesen werden, steuert der Use Case Decoder die Multiplexer auf dem Datenpfad. Dies hat zur Folge, dass das mittels der Adresse ausgewälte Register mit dem Pufferregister für Daten verbunden ist, bevor der VBB-Controller den Zustand *CntResult* einnimmt. So kann er direkt veranlassen, dass das Pufferregister die Daten zwischenspeichert, die an das PLB-IPIF ausgegeben werden sollen. Im nächsten Takt, sind die Daten übernommen worden. Der Controller befindet sich im Zustand *DeliverData* und gibt die Daten an das PLB-IPIF aus. Anschließend wechselt er in den Startzustand.

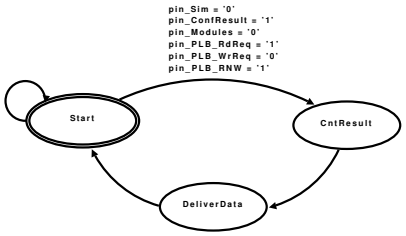


Abbildung 3.5: *Für das Auslesen von Simulationsergebnissen relevanter Teil des Zustandsdiagramms*

3.1.3 Fazit

Die VBB wurde mit allen vorgesehenen Funktionen implementiert. Einzig die Möglichkeit, die CPU anzuhalten konnte aus technischen Gründen nicht realisiert werden. Dafür wurde der zweite Zähler implementiert, welcher die Takte zählt, die die CPU angehalten worden wäre, um so eine Annäherung an den realen Wert berechnen zu können. Sollten sich jedoch die technischen Voraussetzungen einmal ändern, so lässt sich leicht ein zusätzliches Modul an die VBB anschließen, welches das Anhalten und wieder Starten der CPU steuert. Die Signale, um ein solches Modul anzusteuern, sind in der VBB vorhanden und funktionieren auch der Spezifikation entsprechend.

Obwohl der Entwurf fehlerfrei simuliert werden konnte, hat das Gesamtsystem beim abschließenden Test auf dem FPGA-Board nicht wie gewünscht funktioniert. Es wird vermutet, dass die Abweichung der realen Verhaltens von der Simulation in einer zu schnellen Taktung der Hardware begründet ist. Eine Reduzierung des Taktes ließ sich aber nicht mehr durchführen. Der Fehler lies sich somit nicht isolieren, so dass mit dem System im jetzigen Zustand keine Simulation einer Speicherhierarchie möglich ist.

3.2 V-Power Bus (VPB)

3.2.1 Übersicht

Der VPower Bus (VPB) bildet die zentrale Einheit zur Kommunikation aller in der Speicherlogik verbauten Einheiten untereinander. Er bietet die Möglichkeit, über ein Token-Vergabe-Protokoll die Bus-Arbitrierung dezentral zu gestalten. Das Hauptaugenmerk bei der Modellierung des Busses ist darauf gerichtet, möglichst geringe Latenzzeiten zu erzeugen. Das zweite Ziel der Modellierung ist es, eine dynamische Umgebung für alle Busteilnehmer herzustellen, die es erlaubt, diese während der Laufzeit an den Bus anzubinden oder sie wieder zu entkoppeln. Der Bus ist ausgelegt auf die speziellen Anforderungen, die eine Simulation einer Speicherhierarchie stellt. Besondere Rücksicht verlangt dabei die Tatsache, dass die einzelnen Busteilnehmer niemals isoliert funktionieren können. Die Teilnehmer sind ausschließlich die Bus-Bridge (VBB, siehe 3.1) und Speichersimulationseinheiten. Diese realisieren effektiv die Simulationsumgebung. Bedenkt man übliche, realistische Szenarien von Speicherhierarchien, lassen sich zwei grundlegende Hierarchietypen identifizieren. Zum einen existieren vertikale Anordnungen, also sich im Adressraum überlappende Speicher. Konkret sind dies typische Cachehierarchien oder -stacks, in denen verschiedene Cachetypen hintereinander aufgereiht werden. Zum Anderen sind jedoch auch horizontale Anordnungen möglich. Also überlappungsfreie. So können verschiedene Intervalle des Adressraums auf mehrere Speicher verteilt sein, ohne dass zwischen diesen Speichern irgendeine Beziehung bestehen würde.

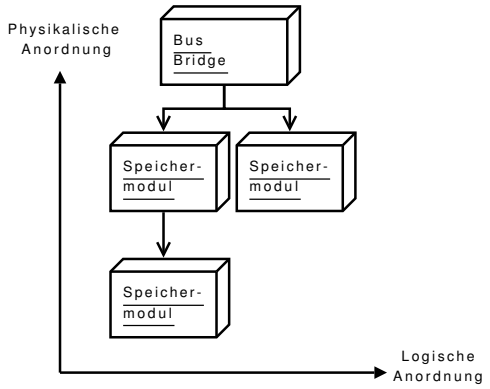


Abbildung 3.6: *Baumförmige Modulanordnung*

In der Realität kommen beide Hierarchietypen in Kombination vor, so dass modellhaft ein Baum aus Bausteinen vorliegt, wie in der Abbildung 3.6 veranschaulicht. Diese Tatsache wurde im VPower Bus berücksichtigt. Es bestehen neben diesen noch weitere Anforderungen an den Bus. Dieser muss so ausgelegt sein, dass zu jeder Zeit die Konsistenz der Transaktionen sichergestellt bleibt. Das bezieht sich sowohl auf normale Fälle der Kommunikation, als auch auf Ausnahmefälle. Ausnahmen sind fehlerhafte Speicher, fehlerhafte Speicherzugriffe, nicht vorhandene Einheiten, Dateninkonsistenzen oder -verluste.

Der VPower Bus nimmt im Gesamtdesign eine zentrale Stellung ein. Wie schon erwähnt, sind alle übrigen Komponenten an diesen gebunden.

Da über diesen Bus ein *Token Passing Protocol* implementiert ist, besitzt der Bus an sich keinerlei eigene Logik. Wie im Abschnitt 3.3 beschrieben, besitzen alle Teilnehmer am Bus die gleichen Privilegien. Das Token, welches zwischen diesen hin und her wandert, bestimmt die Aktivität der Module. Zu jeder Zeit ist nur ein Modul aktiv am Bus und führt selbständig alle notwendigen Operationen durch. Das Token-Passing Protokoll des Busses (3.2.3) beschreibt hinreichend generisch alle notwendigen Operationen.

Zwar existiert zwischen den einzelnen Modulen bezüglich der Kommunikation kein Unterschied, jedoch nimmt die VPower Bus-Bridge eine Sonderstellung ein. Sie stößt exklusiv neue Transferzyklen an und dient in entgegengesetzter Richtung als Abschlussmodul jedes Transfers.

Aus der Implementierungsperspektive arbeitet der Bus also nicht selbstständig, sondern bildet lediglich das Trägermedium für die Kommunikation, die durch das VPower IPIF fest vorgegeben ist. Aus Sicht der Modellierung jedoch formalisiert der Bus die Kommunikation. Die IP-Interfaces (IPIF) sind effektiv lediglich Nutzer des Busprotokolls.

3.2.2 Token Passing Protocol

Das sogenannte *Token Passing Protocol* (TPP) dient im VPower Memory-Controller der dezentralen Busarbitrierung. Wie schon erwähnt, besitzen alle Busteilnehmer die Möglichkeit, die Bussteuerung vollständig zu übernehmen. Solange der Masterstatus vergeben bleibt, hat kein zweiter Teilnehmer die Möglichkeit, steuernd in die Transferzyklen einzugreifen.

Die Tatsache, dass sowohl horizontale als auch vertikale Kommunikation stattfindet, und dass jeweils nur ein Teilnehmer die Buskontrolle übernehmen darf, führte zu der Entscheidung ein Token-Passing Protokoll einzusetzen. Der jeweilige Tokenbesitzer übernimmt demnach vollständig die Buskontrolle.

Das Protokoll ist kein Ringprotokoll. Die Bus-Bridge “erzeugt“ bildlich ein Token auf dem Bus, welches sich frei von Teilnehmer zu Teilnehmer bewegen kann und welches bei Abschluß eines Transferzyklus wieder von dieser entgegengenommen wird.

Das Annehmen und Weiterreichen des Tokens liegt allein in der Verantwortung des aktuellen Tokenbesitzer. Alle anderen Teilnehmer nehmen eine passive Rolle am Bus ein.

Es ist entscheidend für die Leistungsfähigkeit des Busses, dass ein effizientes aber dennoch zuverlässiges Schema zur Weitergabe und/oder Rücknahme des Tokens entwickelt wird. Die Implementierungsdetails werden im folgenden Abschnitt beschrieben.

3.2.3 Implementierungsübersicht

Im Abschnitt 3.3.4 des nächsten Kapitels werden die im Folgenden angesprochenen VHDL-Ports definiert. Konkret wird hier eine abstrakte Übersicht über das Busprotokoll gegeben. Die konkreten Details zur Implementierung finden sich ebenfalls im Kapitel 3.3.

Taktung

Der primäre Takt **pin_vpb2vip_Clk** entspricht genau dem Systemtakt. Das Signal wird hierzu direkt von der Speicherlogik aus eingespeist.

Tokenübergabe

Die Übergabe des Tokens entspricht Pegeländerungen auf dem TokenTaken-Signal. Dieses wird jeweils über den Port **pot_vip2vpb_Tt** gespeist. Das jeweils aktive VPower IPIF erhält die jeweiligen Pegel an diesem Port. Alle Busteilnehmer - also Speichermodule und Bus-Bridge - sind mit identischen Schnittstellen (VPower IPIF) ausgestattet. Eine Unterscheidung zwischen beiden genannten ist also nicht nötig. Entsprechend ist die Tokenübergabe von Modul zu Modul immer identisch.

Zur Vereinfachung der Beschreibung setzen wir den Begriff *Signal* gleich mit dem Signal am Port; unabhägig davon ob es die Master- oder Slavevariante ist.

Zur Übergabe des Tokens wird das TokenTaken Signal (**pot_vip2vpb_Tt**) auf 0 gesetzt. Dies signalisiert allen anderen Teilnehmern, dass einer von ihnen nun den *master*-Status erhalten soll. Gleichzeitig wird ein Teilnehmer eindeutig über anliegende Signale identifiziert. Dies wird nachfolgend noch erläutert. Zusätzlich wird die Art der bevorstehenden Aktion angezeigt. Dies geschieht durch setzen der Signale *ReadRequest* (**pot_vip2vpb_RdReq**, Lesezugriff) oder *WriteRequest* (**pot_vip2vpb_WrReq**, Schreibzugriff),

Das jeweils referenzierte Modul muss das Token entgegennehmen. Dies geschieht indem das referenzierte *slave*-Modul seinerseits wieder das *TokenTaken*-Signal setzt. Gleichzeitig werden Bestätigungen vom nun neuen *master*-Modul zum alten für die jeweils angefragte Aktion gesendet. Für das Lesen *ReadRequestAcknowlegement* (**pot_vpb2vip_RdReq**). Für das Schreiben *WriteRequestAcknowlegement* (**pot_vpb2vip_WrReqAck**).

Module werden auf dem Bus durch ein Tripel bestehend aus Modul-ID (**pot_vpb2vip_Id**), Speicheradresse (**pot_vpb2vip_Addr**), und Tokenrichtung (**pot_vpb2vip_TDir**) identifiziert. Jedes Modul kennt seine eigene Identität, bestehend aus dem Tupel von Modul-ID und Speicheradressintervall und die ID des logisch übergeordneten Moduls. Die Ausnahme ist dabei die Bus-Bridge; diese hat kein übergeordnetes Modul. Ist das Tokenrichtungssignal gesetzt, so sind alle Module sensitv auf die Modul-ID des jeweils übergeordneten Moduls. Die eigene ID spielt dann keine Rolle. Zusätzlich muss die angelegte Adresse in das eigene Adressintervall fallen. Dies entspricht einer logischen

“Abwärtsbewegung“ durch die Modulhierarchie. Umgekehrt - das Tokenrichtungssignal ist gelöscht - sind alle Module nur auf ihre eigene Modul-ID sensitiv. Zusätzlich muss die angelegte Adresse in das eigene Adressintervall fallen.

Da jedes Modul selbständig die Signale am Bus setzen darf, sofern es aktiviert wurde, kann es die logische Fortsetzung der Aktionen bestimmten. Es kann also über die Weitergabe einer Anfrage, über die Erzeugung einer Anfrage oder den Abschluss eines gesamten Transferzyklus entscheiden.

Initiierung eines Transferzykels

Jeder simulierte Zugriff, lesend oder schreibend, wird durch die Bus-Bridge ausgelöst. Diese ist jeweils Start- und Endpunkt eines Transferzyklus. Mit Transferzyklus ist die gesamte Kette der Modulzugriffe von der Bus-Bridge aus bis zu ihr zurück gemeint. Potentiell wäre jedes andere Modul auch dazu in der Lage. Der Abschluss liegt jedoch derzeit ausschließlich in der Verantwortung der Bus-Bridge.

Zyklus: Simulationsergebnisse lesen

Während der Simulation agieren alle Module autonom am Bus. Sofern sie Tokenbesitzer sind, unterliegen sie keinerlei Beschränkungen. Die Tokenrückgabe geschieht wie erwähnt auf freiwilliger Basis. Da alle Module unterschiedliche Funktionen erfüllen, ist auch jedes einzelne verantwortlich für die Verwaltung der Simulationsdaten. Die Daten umfassen üblicherweise Zugriffszähler, Zähler für Zugriffsfehler oder -erfolge und Zugriffszeiten. Diese Ergebnisse müssen nicht zu jeder Zeit frei verfügbar sein, sondern werden gleichzeitig mit dem Token an nachfolgenden Module übergeben.

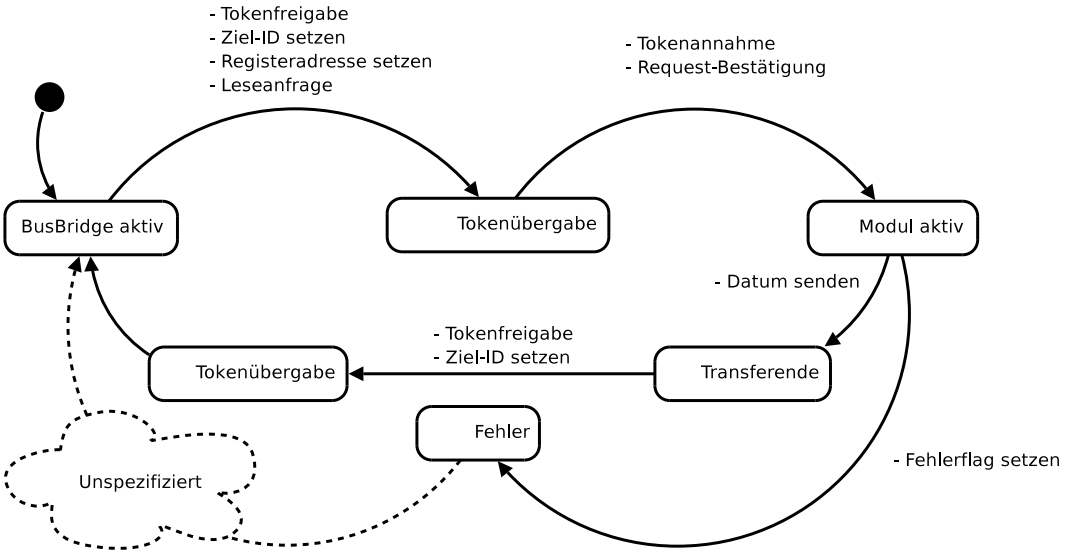


Abbildung 3.7: *Schema: Lesen der Simulationsergebnisse*

In der Abbildung 3.7 wird für den Startzustand vorausgesetzt, dass die BusBridge aktiv ist. Dies entspricht also einem konsistenten Ausgangspunkt.

Um die Simulationsergebnisse zu lesen, wird das Token abgegeben. Dies wird durch das Löschen des TokenTaken-Signals (**pin_vpb2vip_Tt**) und gleichzeitigem Anlegen der Modul ID (**pin_vpb2vip_Id**) erreicht. Zugleich wird der Lesezyklus initiiert. Das bedeutet, dass ein ReadRequest (**pin_vpb2vip_RdReq**) signalisiert wird. Anhand der ebenfalls nun anliegenden Speicheradresse (**pot_vpb2vip_Addr**) kann das nun adressierte Modul zwischen regulären und speziellen (Simulationsergebnisse lesen, Konfigurationen schreiben) Operationen unterscheiden. Wie bereits in der Einleitung erwäht, sind die Kontroll- und Statusregister in den gleichen Adressraum abgebildet wie die Nutzdaten - jedoch in unterschiedliche Intervalle.

Es ist garantiert, dass genau ein Modul sensitiv auf das Tupel von Modul-Id und Speicheradresse ist. Somit kann ohne eine Verhandlungsprozedur das Token weitergereicht werden. Um sich selbst zu aktivieren, nimmt das entsprechende Modul das Token entgegen. Dies bedeutet, dass das nun aktivierte Modul die Buskontrolle übernimmt. Es liefert folglich jetzt selbst das TokenTaken-Signal auf den Bus. Zur Bestätigung erfolgt das Signal ReadAcknowlegde (**pin_vpb2vip_RdReqAck**). Das Modul ist jetzt vollständig aktiviert und erwartet Daten zu empfangen. Alle anderen Busteilnehmer verbleiben passiv. Sollte das Modul nicht bereit sein, so muss dieses ein Fehlersignal (**pin_vpb2vip_Error**) senden. Die genauen Folgen eines Fehlersignals sind noch zu spezifizieren.

Als Daten wird jeweils genau ein Wort parallel gesendet. Der Empfang wird daraufhin mit einem ReadAcknowlegde (**pin_vpb2vip_RdAck**) bestätigt.

Nun ist es Aufgabe des jeweils aktiven Moduls, sein Token wieder an die Bus-Bridge zu übergeben. Analog zum Beginn des Zyklus wird das Token durch das Löschen des TokenTaken-Signals freigegeben. Gleichzeitig liegt die Modul-ID der Bus-Bridge an. Die ist fixiert auf die Adresse 0. Die Übergabe an die BusBridge erfolgt in diesem Fall ohne Quittierung. Diese erwartet Reaktionen der Busteilnehmer innerhalb von festgelegten Timeout-Intervallen. Überschreiten dieser Intervalle führt zwangsläufig zu einem Fehlerzustand.

Fehler oder erfolgreiche Transaktionen haben zwangsläufig die Rückkehr des Tokens an die Bus-Bridge zur Folge.

Zyklus: Modulkonfigurationen schreiben

Bevor eine Simulation gestartet werden kann, müssen zunächst alle Module ihre Funktionskonfiguration erhalten. Dies umfasst insbesondere die Initiierung, Wahl der sensitiven Adressintervalle und Vergabe von Parametern, die die genaue Funktionsweise eines Modultyps definieren. Die Konfigurationsvergabe erfolgt sequentiell an alle Module.

Abbildung 3.1 zeigt schematisch den Ablauf eines Konfigurationszyklus für ein Modul. Die Bus-Bridge ist im Startzustand aktiv, sie hält also das Token. Dies entspricht also einem konsistenten

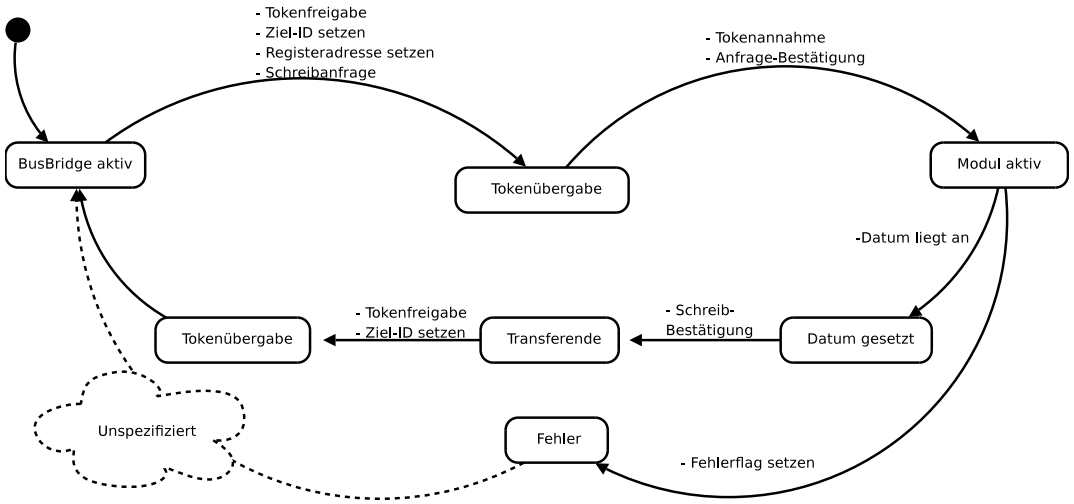


Tabelle 3.1: *Schema: Schreiben der Modulkonfigurationen*

Ausgangspunkt.

Um ein Konfigurationsdatum in ein Modul zu schreiben, muss das Token an das jeweilige Modul abgegeben werden. Dies wird durch das Löschen des TokenTaken-Signals (**pin_vpb2vip_Tt**) und gleichzeitiges Anlegen der Module ID (**pot_vpb2vip_Id**) erreicht. Zeitgleich wird ein Schreibzyklus initiiert. Es liegen also, analog dem Zyklus zum Lesen der Simulationsergebnisse, die Modul-ID des Ziels (**pot_vpb2vip_Id**) und ein WriteRequest (**pin_vpb2vip_WrReq**) am Bus an. Das adressierte Modul übernimmt nun das Token. Dazu setzt es das TokenTaken-Signal (**pin_vpb2vip_Tt**) auf dem Bus. Gleichzeitig wird eine Bestätigung, das Signal WriteRequestAcknowledgement (**pin_vpb2vip_WrReqAck**), angelegt. Das Modul hat die Buskontrolle übernommen. Die Bus-Bridge ist nun für das Senden verantwortlich. Dieser Zustand widerspricht dem generellen Paradigma, dass jeweils nur ein aktiver Busteilnehmer existiert. In diesem Fall jedoch ist die Tokenvergabe notwendig, um die Bereitschaft des Moduls zum Empfang von Daten abzufragen. Alle anderen Busteilnehmer verbleiben jedoch passiv. Die Bus-Bridge kann nun mit dem Senden eines Datums beginnen. Sollte das Modul nicht bereit sein, so muss dieses ein Fehlersignal (**pin_vpb2vip_Error**) senden. Die genauen Folgen eines Fehlersignals sind noch zu spezifizieren.

Als Daten wird jeweils genau ein Wort parallel gesendet. Das Empfängermodul quittiert den Empfang mit einem WriteAcknowlegde-Signal (**pin_vpb2vip_WrAck**). Das Setzen dieses Signals beendet den Sendevorgang.

Zum Abschluß des Transfers muss das Token wieder an die Bus-Bridge zurückgereicht werden. Der Vorgang ist äquivialent zum Verfahren, wie es im vorigen Abschnitt beschrieben wurde.

Zyklus: Lesenden Speicherzugriff simulieren

Sofern alle Module erfolgreich ihre Konfiguration erhalten haben, kann die Simulation gestartet werden.

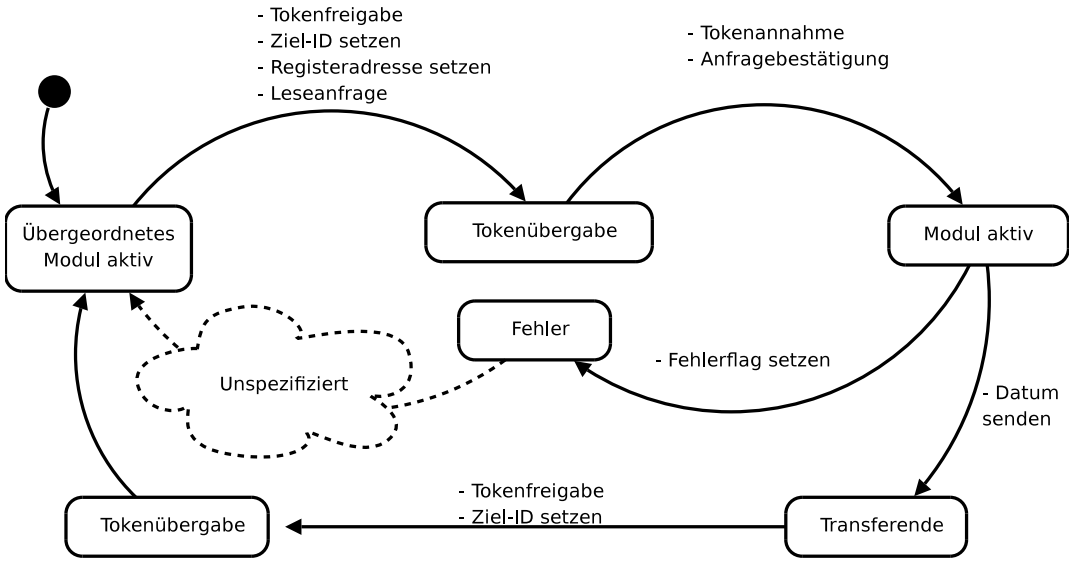


Abbildung 3.8: *Schema: Simulierter, lesender Zugriff*

Ähnlich wie im Zyklus zum Lesen der Konfigurationsdaten, ist auch der Ablauf zum simulierten Lesen über die Module. Das Verfahren zum simulierten Lesen unter Einbeziehung der Module gestaltet sich ähnlich wie das schon beschriebene Verfahren zum Schreiben der Modulkonfiguration. Der in Abbildung 3.8 schematisierte Ablauf zeigt diesen aus der Perspektive des übergeordneten Containers, nicht aus der des aktivierten Moduls. Analog zum Lesen von Konfigurationsdaten wird jeweils genau ein Datum gesendet und quittiert. Der Ablauf ist jedoch für jedes Modul der gleiche. Somit ist nicht die Bus-Bridge notwendigerweise aktiv, sondern ein beliebiges übergeordnetes Modul. Das kann die Bus-Bridge, ein Modul welches sich im gleichen Speicheradresssraum aber in einem anderen Adressintervall befindet, oder aber ein Modul in exakt dem gleichen Adressintervall wie das referenzierte sein.

Die Tokenübergabe vom übergeordneten Modul zum nächsten erfolgt vollkommen analog zum Lesen von Konfigurationsdaten. Alle bereits genannten Ports werden in gleicher Weise verwendet. Alle Module sind, wie bekannt, mit identischen Bus-Logiken ausgestattet. Nach der Aktivierung des nächsten, also untergeordneten Moduls (*Modul aktiv*), kann dieses sein Datum senden und sein Token wiederum freigeben. Da es selbst Bus-Master war, bestimmt es nun, wie das Token nun weitergereicht wird. Hier sind zwei Fälle möglich. Entweder entspricht die Modul-ID der des referenzierenden (bzw. des übergeordneten) Moduls oder aber der irgendeines anderen. Im ersten Fall läuft das Token also zurück. Im zweiten Fall wird ein weiterer Lese- oder Schreibzyklus auf dem

Bus aktiviert. In der Abbildung 3.8 ist der Übergang vom Zustand *Tokenübergabe* zum Zustand *Übergeordnetes Modul aktiv* entweder derjenige rückwärts zum Aufrufenden oder aber das eben aktiv gewesene Modul wird selbst das übergeordnete Modul für die folgende Operation.

Nach Rückkehr des Tokens an die Bus-Bridge ist der Lesezyklus beendet.

Zyklus: Schreibenden Speicherzugriff simulieren

Der Schreibzugriff ähnelt stark dem schreibenden Zugriff auf die Modulkonfiguraton. Ähnlich wie dort und wie in allen anderen Zyklen wird ein Modul durch die Übergabe des Bus-Tokens aktiviert. Das simulierte Schreiben hat, wie das Lesen, unter Umständen Referenzierungen untergeordneter Module zur Folge. Abbildung 3.9 verdeutlicht schematisch den Ablauf eines Schreibzyklus.

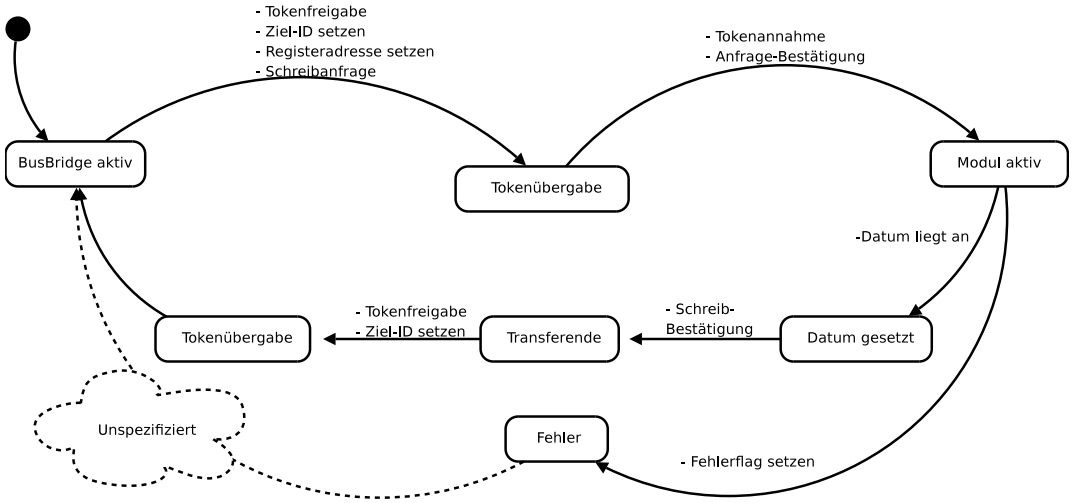


Abbildung 3.9: *Schema: Schreiben der Modulkonfigurationen*

Nachdem das Modul über das Token aktiviert wurde, ist es bereit ein Datum vom Bus zu empfangen. Daraufhin wird, wie im Fall für das Schreiben der Konfigurationsdaten, eine Bestätigung gesendet. Anschließend wird das Token zurück- oder weitergereicht. Wie im Fall für den lesenden Zugriff besteht entweder die Möglichkeit, das Token an das referenzierende Modul zurückzureichen oder einen neuen Schreibzyklus für untergeordnete Module zu initiieren.

Nach Rückkehr des Tokens an die Bus-Bridge ist der Schreibzyklus beendet.

3.2.4 Stand

Die VPower Bus-Bridge wurde früh in der Entwurfsphase spezifiziert. Revisionen beschränkten sich bislang auf die Anpassung der Signalverbindungen. Nicht berücksichtigt wurde dabei die Logik der Transferzyklen. Diese werden allerdings in näherer Zukunft nochmals überprüft und gegebenenfalls

angepasst werden. Insbesondere die Behandlung von Fehlerzuständen bleibt bislang unspezifiziert. Eine einheitliche Fehlerbehandlungsstrategie wurde noch nicht entwickelt.

3.3 V-Power Bus IP Interface (VPB-IPIF)

3.3.1 Motivation

Im **V-POWER** Hardwareentwurf sind alle Speichermodule über den **V-POWER** Bus (VPB) verbunden. Die Module können über diesen unter Einhaltung des Token-Passing Protokolls (TPP) kommunizieren. Dabei handelt es sich aufgrund des TPPs um eine dezentrale Bussteuerung, die eine komplexe State Machine in den Modulen erfordert. Damit nicht jedes Modul dieses komplexe Busprotokoll implementieren muss, wurde der VPB-IPIF-IP-Core (**V-POWER** IP-Core Interface) entwickelt. So muss ein Modul, das für den VPB konzipiert wird, nur diesen IP-Core einbinden und einige wichtige Interaktions-Constraints, die später genauer erläutert werden, einhalten, um mit anderen Modulen am VPB kommunizieren zu können. Dies reduziert deutlich die Komplexität und verbessert die Wartbarkeit und Übersichtlichkeit der einzelnen Busmodule. Vom Konzept her entspricht dies dem Xilinx-Entwurf, der jedem Busteilnehmer des PLB (Processor Local Bus) den PLB-IPIF (PLB IP-Core Interface) bereitstellt.

3.3.2 Aufbau der Library vpower_vpb_ipif_v1_00_a

Die Library **vpower_vpb_ipif_v1_00_a** besteht aus den folgenden Dateien:

- **vpower_vpb_ipif.vhd** Der IPIF Rahmen (Structural).
- **vpb_ipif_sm.vhd** Die State Machine des IPIFs, d.h. die Implementierung des Busprotokolls (Behavioral).
- **ipif_reg.vhd** Registerfile des IPIFs, das die Grundkonfiguration beinhaltet (Behavioral).

Des Weiteren werden die folgenden Libraries verwendet:

- **vpower_common_v1_00_a** Globale **V-POWER** -Funktionen und Konstanten (Package).

3.3.3 Aufbau und Integration des VPB-IPIF

Abbildung 3.10 zeigt den schematischen Aufbau des VPB-IPIF. Es enthält eine Transceiver-Schaltung, die verhindert, dass ein nicht am VPB aktives Modul Signale aktiv treiben und damit einen Kurzschluss am VPB erzeugen kann. Eine Sonderrolle nimmt dabei die VBB ein, die z.B. das SimHold-Signal dauerhaft auf dem VPB treibt. Bussignale werden nur dann an das innere Modul

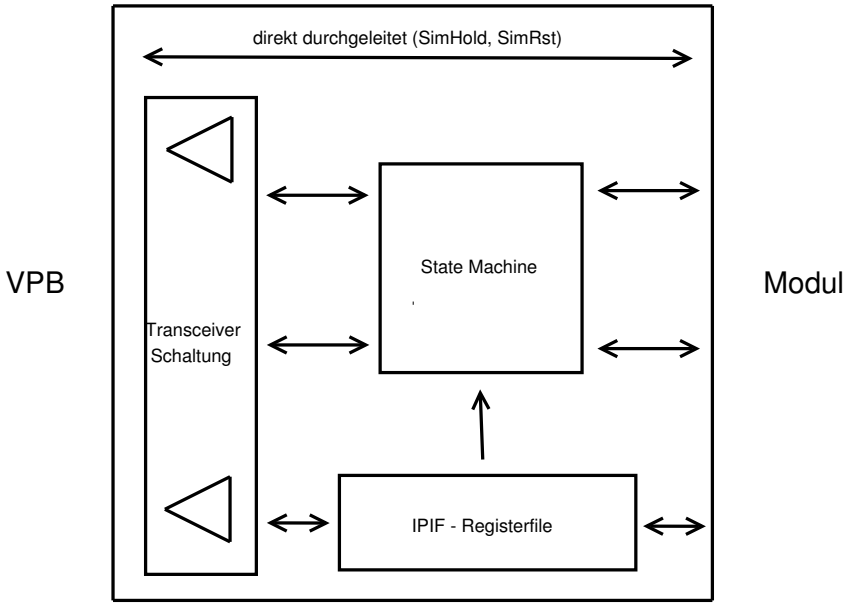


Abbildung 3.10: *VPB IPIF - Schematischer Aufbau*

weitergeleitet und für dieses für den Zeitraum eines Kontextes zwischengespeichert, wenn dieses über die Vorgänger-Id und die jeweilige Adresse adressiert wird. Ein Modul erkennt seine aktive Masterrolle am VPB durch das Chip Select- bzw. Master-Signal. Des Weiteren enthält dieser IP-Core eine interne State Machine, die das VPB-Busprotokoll implementiert und Daten für einen Request-Context zwischenspeichert, bis dieser abgeschlossen ist. Das TokenTaken-Signal dieser State Machine steuert ebenso die o.g. TriStates. Das VPB-IPIF enthält ferner ein internes Registerfile, das die Grundkonfiguration (Aktiv/Inaktiv, Vorgänger-Id, Base Address, High Address) eines jeden Moduls beinhaltet.

Wird ein VPB-Busteilnehmer erzeugt, so muss dieser die VPB-seitige Portsignatur, die in A.1 genauer nachzulesen ist, bereitstellen und diese Ports mit dem VPB-IPIF verbinden. Des Weiteren ist das integrierte VPB-IPIF durch die Generics, wie in Abschnitt 3.3.4 beschrieben, zu konfigurieren.

3.3.4 Generics und Ports

Im folgenden werden zunächst die Generics und anschließend die Ports des VPB-IPIF erläutert. Über die Generics kann dieser IP-Core den individuellen Anforderungen jedes umgebenden Moduls angepasst werden.

Name	Beschreibung
C_VPB_AWIDTH	Adressbreite des VPB-Busses
C_VPB_DWIDTH	Datenbreite des VPB-Busses
C_VPB_ID_WIDTH	Breite der ID-Leitung
C_VPB_SIMTIME_DWIDTH	Datenbreite zur Propagierung der Simulationszeit einer Transaktion
C_VPB_ID_ARRAY	Interne Id für jedes Submodul mit eigenem Adressbereich
C_VPB_ADDR_RANGE_ARRAY	Adressbereiche des Moduls

Name	Beschreibung
C_VPB_ID	Id des Moduls am VPB
C_VPB_PREID	Id des Vorgängermoduls
C_VPB_ENABLE	Modul aktivieren ('1') / deaktivieren ('0')
C_VPB_BASIC_TOKEN_OWNER	Konfiguration des initialen Token Owners ('1')
C_VPB_ID_CONF_ARRAY	Zugriffsberechtigungen der Adressbereiche

Tabelle 3.2: *VPB-IPIF Generics*

Die Adress- und Datenbreite des VPB-Busses ist auf 32 Bit festgelegt worden. Des Weiteren sind 32 Bit für die Id-Leitung und 64 Bit für die Simulationszeit vorgesehen. Es existiert genau ein initialer Token Owner, der über das Generic **C_VPB_BASIC_TOKEN_OWNER** festgelegt wird. Das Aktivieren eines Moduls erfolgt über das Generic **C_VPB_ENABLE**. Die Werte der Generics **C_VPB_ENABLE**, **C_VPB_PREID** und die Base- und High Address des simulierten Speicherbereichs dienen dem internen Registerfile als Initialwerte und können zur Laufzeit bei angehaltener Simulation überschrieben werden.

Das Generic **C_VPB_ID_ARRAY** definiert den Typ der in **C_VPB_ADDR_RANGE_ARRAY** definierten Adressbereiche und kann die folgenden als Konstanten definierten Werte annehmen:

- **C_MOD_REAL** Simulierter Adressbereich, der einem Teil des realen DIMMs entspricht.
- **C_MOD_CONF** Konfigurations-Adressbereich des Moduls.
- **C_MOD_SIMR** Simulationsergebnis-Adressbereich des Moduls.
- **C_IPIF_CONF** Konfigurationsbereich des IPIFs (internes Registerfile). In diesem Registerfile werden grundlegende Konfigurationen abgelegt, wohingegen im Modulkonfigurationsregisterfile modulspezifische Eigenschaften definiert werden.

Einem Adressbereich können über das Generic **C_VPB_ID_CONF_ARRAY** Zugriffsrechte zugewiesen werden:

- **C_RW** Lesender und schreibender Zugriff.
- **C_R** Nur lesender Zugriff.
- **C_W** Nur schreibender Zugriff.
- **C_NA** Kein Zugriff erlaubt.

Die Benennung der Ports des IPIFs bzw. der FSM folgen folgendem Schema:

(pin/pot)_(dir1)2(dir2)_(potname)

Dabei sind *dir1* und *dir2* durch folgende Abkürzungen zu ersetzen:

- **vpb V-POWER** Bus (VPB)
- **vip V-POWER** IPIF (Modul, das IPIF integriert)
- **ip V-POWER** IP-Core (Modul, das IPIF integriert)

Die genaue Portspezifikation dieses Hardware-Moduls kann im Anhang unter A.1 nachgelesen werden.

3.3.5 Das Busprotokoll / die State Machine

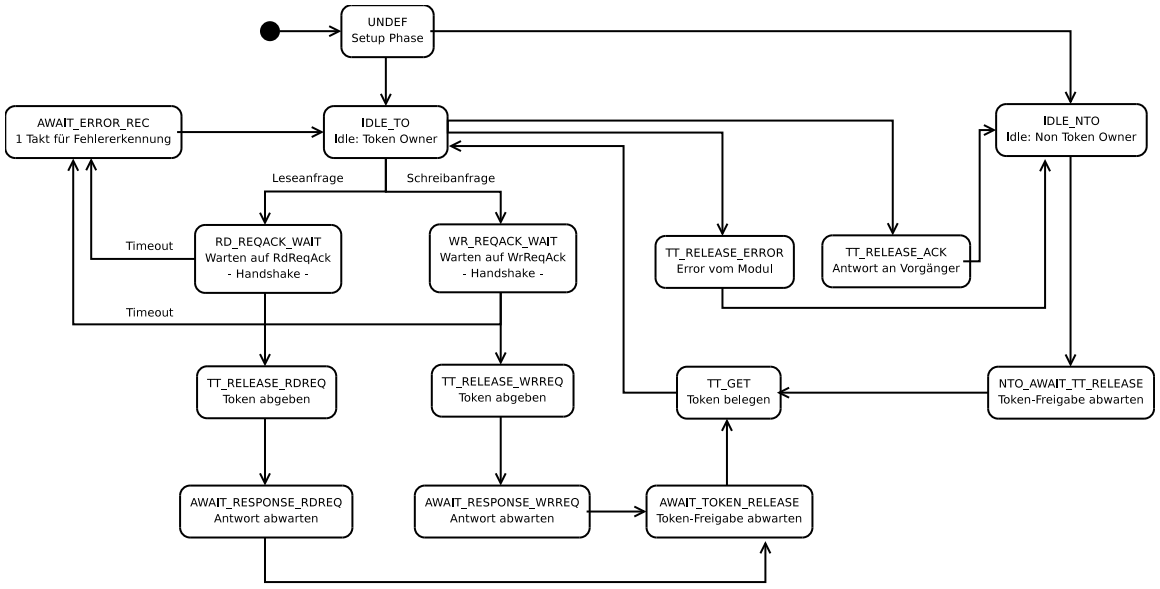


Abbildung 3.11: *VPB IPIF - FSM*

Grundlegend gelten folgende Konventionen:

- **Zustandsübergänge/Signalzuweisungen** Zustandsübergänge und Signalzuweisungen finden grundsätzlich mit einer steigenden Taktflanke statt.
- **Validität von Signalen** Signale gelten für ein Modul nur dann als valide und damit als verarbeitungsbereit, wenn das entsprechende Chip Select bzw. das Master-Signal dies signalisieren ('1').
- **Speichern/Löschen von Kontextdaten** Die Daten (Adresse, Daten, Rd/WrReq, etc.) werden nur bis zum Abschluss des aktuellen Kontextes (Leseanfrage/Schreibanfrage) gespeichert. Nach dem Senden eines Ack-Signals oder mit einer neuen Anfrage werden diese Werte auf 'Z' zurückgesetzt. Nötig wird dieses Zwischenspeichern dadurch, dass bei der Token-Übergabe

die relevanten Anfrage-/Antwortsignale nicht mehr aktiv am VPB getrieben werden ('Z' auf dem VPB).

- **SimHold:** Das Ändern des SimHold-Signals kann nur die VBB und nur im Zustand **IDLE_TO** durchführen.
- **Timeouts:** Timeouts geschehen auf dem VPB nur dann, wenn ein Adressbereich unzulässig bzgl. seiner Zugriffsrechte adressiert wurde. In diesem Fall wird eine Anfrage am Bus ignoriert. Die Anzahl von Taktzyklen für einen Timeout beträgt 15 Zyklen.

Im Wesentlichen existierten 4 Anwendungsfälle für das VPB IPIF, die im folgenden beschrieben werden:

- Initiale Setup Phase beim Einschalten der Hardware
- IPIF Konfigurations-Register lesen / beschreiben
- Modulregister lesen / beschreiben (Konfiguration / Simulation)
- Simulierten (realen) Speicherbereich lesen / schreiben

Initiale Setup Phase beim Einschalten der Hardware

Nach Einschalten der Hardware befindet sich das IPIF, wie in Abb. 3.11 dargestellt, im Zustand **UNDEF**. Abhängig vom Wert des Generics **C_VPB_BASIC_TOKEN_OWNER** nimmt dies den ersten wohldefinierten Zustand an. Im Fall eines initialen Token Owners ist dies der Zustand **IDLE_TO** (Idle: Token Owner), ansonsten **IDLE_NTO** (Idle: Non Token Owner). Damit ist die initiale Setup-Phase abgeschlossen.

IPIF Konfigurations-Register lesen / beschreiben

1. Ausgangspunkt zum Auslesen / Beschreiben des IPIF-Konfigurationsregisters ist stets die VBB. Diese legt dazu die folgenden Signale an die modulseitige Schnittstelle des IPIFs, das sich im Zustand **IDLE_TO** (wird der VBB über das Signal pot_vip2ip_Mst signalisiert) befinden muss:
   - **pin_ip2vip_Addr** Adresse des jeweiligen IPIFs.
   - **pin_ip2vip_Succ** Auswahl des Nachfolgers ('1').
   - **pin_ip2vip_Rd/WrReq** Lese-/Schreibanfrage ('1'). Dabei darf nur eins der beiden Signale auf '1' gesetzt sein, da sonst diese Anfrage vom IPIF ignoriert wird.
   - **pin_ip2vip_WrData** Ggf. zu schreibende Daten.

   Das VBB-IPIF gibt daraufhin die folgenden Signale mit dem nächsten Takt an den VPB weiter:

- **pot_vip2vpb_Id** VBB-Id (eigene Id).
- **pot_vip2vpb_TtDir** Vorwärtsadressierung (’1’).
- **pot_vip2vpb_SimTime** Aktuelle Simulationszeit.
- **pot_vip2vpb_Rd/WrReq** bzw. **pot_vip2vpb_WrData** Entspricht den Eingabedaten.

Das VBB-IPIF befindet sich mit der nächsten Taktflanke im Zustand **RD/WR_REQACK_WAIT** und signalisiert über das Busy-Signal **pot_vip2ip_Busy** dem Modul den Arbeitszustand.

2. Das IPIF, dessen Konfigurationsregister gelesen / beschrieben werden soll, registriert die Anfrage im Zustand **IDLE_NTO**. Unabhängig davon, ob das IPIF aktiviert ist, wird geprüft, ob ein Konfigurationsadressbereich (in diesem Fall der Adressbereich des internen Registerfiles) angesprochen wird. Ist dies der Fall, so muss zusätzlich als Id die VBB-Id auf dem VPB anliegen, da nur die VBB und damit der initiale Token Owner als Konfigurator in Frage kommt (d.h. ein Speichermodul konfiguriert nie seinen Nachfolger), und die jeweilige Zugriffsberechtigung existieren. Andernfalls wird die Anfrage der VBB ignoriert und somit beim anfragenden IPIF ein Timeout auftreten, das dazu führt, dass das anfragende IPIF für einen Takt in den Zustand **AWAIT_ERROR_REC** wechselt, dem Modul ein Error-Signal (**pot_vip2ip_Error**) sendet und seinen Timeout-Counter zurücksetzt. Dieser eine Takt dient der Fehlererkennung durch das Modul. Danach wechselt das IPIF wieder in den Ausgangszustand **IDLE_TO** und überlässt damit dem Modul die Fehlerbehandlung. Im positiven Fall nimmt das IPIF die Anfrage entgegen, speichert folgende Eingangssignale zwischen und leitet diese mit dem nächsten Takt an das interne Registerfile:

   - **pin_vpb2vip_Rd/WrReq** Lese-/ Schreibanfrage (das jeweils andere Signal wird auf ’0’ gesetzt).
   - **pin_vpb2vip_WrData** Ggf. zu schreibende Daten.
   - **pin_vpb2vip_Addr** Zu lesende / schreibende Adresse.

   Als Bestätigung (Handshake) wird dem anfragenden IPIF das Signal **pot_vip2vpb_RdReqAck** bzw. **pot_vip2vpb_WrReqAck** (’1’) übermittelt und in den Zustand **NTO_AWAIT_TT_RELEASE** gewechselt, in dem auf die Freigabe des Bus-Tokens gewartet wird. Diese Änderungen treten ebenso mit dem nächsten Takt in Kraft.

3. Das VBB-IPIF empfängt die Anfrage-Bestätigung, setzt den internen Timeout-Counter zurück und wechselt in den Zustand **TT_RELEASE_RD/WRREQ**.

4. In diesem Zustand gibt das VBB-IPIF das Token ab und trennt mit der nächsten Taktflanke alle Signale vom VPB (durch die TriStates). Daraufhin findet ein Wechsel in den Zustand **AWAIT_RESPONSE_RD/WRREQ** statt, in dem auf eine Antwort des anderen IPIFs gewartet wird.

5. Das IPIF des adressierten Moduls erkennt die Freigabe des Bus-Tokens und wechselt in den Zustand **TT_GET**.

6. In diesem Zustand findet die Allokation des VPBs statt. Es werden folgende Signalzuweisungen mit dem nächsten Takt wirksam:

   - **pot_vip2vpb_Tt** Bus-Token wird allokiert ('1').
   - **pot_vip2ip_Mst** Signalisierung des Master-Zustands an das Modul ('1').
   - **pot_vip2ip_CS** Berechnung des Chip-Selects (in diesem Fall wird das IPIF-Registerfile selektiert).
   - **pot_vip2ip_Busy** FSM des IPIFs befindet sich nicht mehr im Arbeitszustand ('0').

   Des Weiteren wechselt das IPIF mit der nächsten Taktflanke in den Zustand **IDLE_TO**.

7. Im Idle-Zustand wird nun das selektierte interne IPIF-Registerfile aktiv und sendet eine Antwort-Nachricht an die FSM des IPIFs. Diese umfasst ein Rd/WrAck und ggf. die gelesenen Daten. Bedingt durch diesen Input wechselt die FSM und damit das IPIF mit der nächsten Taktflanke in den Zustand **TT_RELEASE_ACK** und legt die folgende Signale auf den VPB:

   - **pot_vip2vpb_TtDir** Adressierung des Vorgängers ('0').
   - **pot_vip2vpb_Rd/WrAck** Anfragebestätigung ('1').
   - **pot_vip2vpb_RdData** Ggf. die gelesenen Daten.
   - **pot_vip2vpb_SimTime** Die aktuelle unveränderte Simulationszeit.

   Ebenso wird dem internen Modul wieder der Arbeitszustand über das Signal **pot_vip2ip_Busy** ('1') signalisiert und der zwischengespeicherte Anfrage-Kontext gelöscht.

8. Das VBB-IPIF erhält nun im Zustand **AWAIT_RESPONSE_RD/WRREQ** die o.g. Antwort, speichert die Antwortdaten inkl. der aktuellen Simulationszeit, zwischen und leitet diese an die VBB. Es erfolgt des Weiteren ein Wechsel in den Zustand **AWAIT_TOKEN_RELEASE**.

9. Im Zustand **TT_RELEASE_ACK** des durch die VBB adressierten IPIFs werden alle Signale vom VPB durch die Abgabe des Tokens getrennt und in den Ausgangszustand **IDLE_NTO** gewechselt.

10. Die VBB, bzw. ihr IPIF, erkennt die Busfreigabe, wechselt in den Zustand **TT_GET**, allokiert, wie bereits zuvor beschrieben, den VPB, signalisiert ohne Chip Select-Berechnung (für VBB deaktiviert, da für diese keine VPB-seitigen Adressbereiche existieren) der VBB den Master-Zustand und gelangt schlussendlich in den Ausgangszustand **IDLE_TO** zurück, in der die Antwortdaten verarbeitet werden können.

Modulregister lesen / beschreiben (Konfiguration / Simulation)

Dieser Anwendungsfall gleicht im Wesentlichen dem zuletzt beschriebenen. Jedoch werden im 2. Schritt die Signale **pin_vpb2sm_Rd/WrReq** und **pin_vpb2sm_Addr** an das Modul weitergeleitet, in dem diese Signale mit einem internen Registerfile verbunden sind. Da diese Registerfiles in den Grundlagen dem internen IPIF-Registerfile entsprechen, gleicht auch deren Antwortverhalten dem

des IPIF-Registerfiles. Damit ist auch dieser Anwendungsfall spezifiziert. Etwas komplexer stellt sich jedoch der folgende Fall dar, da in diesem die Möglichkeit einer verketteten Adressierung (z.B. L1-Cache, L2-Cache, DRAM) besteht.

Simulierten (realen) Speicherbereich lesen / schreiben

Beim Zugriff auf einen simulierten (realen) Speicherbereich ist auch eine Adressierungskette denkbar (z.B. L1-Cache, L2-Cache, DRAM). Die folgende Transaktionsfolge beschreibt diesen Anwendungsfall im Detail:

1. Ausgangspunkt des Lesens / Beschreiben des simulierten Speicherbereichs ist stets die VBB. Diese legt dazu die folgenden Signale an die modulseitige Schnittstelle des IPIFs, das sich im Zustand **IDLE_TO** (wird der VBB über das Signal pot_vip2ip_Mst signalisiert) befinden muss:

   - **pin_ip2vip_Addr** Adresse des simulierten Speicherbereichs.
   - **pin_ip2vip_Succ** Auswahl des Nachfolgers ('1').
   - **pin_ip2vip_Rd/WrReq** Lese-/Schreibanfrage ('1'). Dabei darf nur eins der beiden Signale auf '1' gesetzt sein, da sonst diese Anfrage vom IPIF ignoriert wird.

   Das VBB-IPIF gibt daraufhin die folgenden Signale mit dem nächsten Takt an den VPB weiter:

   - **pot_vip2vpb_Id** VBB-Id (eigene Id).
   - **pot_vip2vpb_TtDir** Vorwärtsadressierung ('1').
   - **pot_vip2vpb_SimTime** Aktuelle Simulationszeit.
   - **pot_vip2vpb_Rd/WrReq** Lese-/Schreibanfrage

   Das VBB-IPIF befindet sich mit der nächsten Taktflanke im Zustand **RD/WR_REQACK_WAIT** und signalisiert über das Busy-Signal **pot_vip2ip_Busy** dem Modul den Arbeitszustand.

2. Das adressierte IPIF registriert die Anfrage im Zustand **IDLE_NTO**. Ist dieses Modul, bzw. sein IPIF, aktiviert (**C_VPB_ENABLE** = '1'), liegt die verwendete Adresse im simulierten Adressraum, ist die korrekte Vorgänger-Id (in diesem Fall die VBB-Id) angelegt und ist die Zugriffsoperation konform zu den Zugriffsrechten, so nimmt das IPIF die Anfrage entgegen, speichert folgende Eingangssignale zwischen und leitet diese mit dem nächsten Takt an das Modul:

   - **pin_vpb2vip_Rd/WrReq** Lese-/ Schreibanfrage (das jeweils andere Signal wird auf '0' gesetzt).
   - **pin_vpb2vip_Addr** Zu lesende / schreibende Adresse.
   - **pin_vpb2vip_SimTime** Aktuelle Simulationszeit.

   Als Bestätigung (Handshake) wird dem anfragenden IPIF das Signal **pot_vip2vpb_RdReqAck** bzw. **pot_vip2vpb_WrReqAck** ('1') übermittelt und in den Zustand **NTO_AWAIT_TT_RELEASE**

gewechselt, in dem auf die Freigabe des Bus-Tokens gewartet wird. Diese Änderungen treten ebenso mit dem nächsten Takt in Kraft. Sind die o.g. Bedingungen nicht erfüllt, so tritt beim VBB-IPIF ein Timeout auf, das ein Verhalten hervorruft, wie es im 2. Anwendungsfall beschrieben wird.

3. Das VBB-IPIF empfängt die Anfrage-Bestätigung, setzt den internen Timeout-Counter zurück und wechselt in den Zustand **TT_RELEASE_RD/WRREQ**.

4. In diesem Zustand gibt das VBB-IPIF das Token ab und trennt mit der nächsten Taktflanke alle Signale vom VPB (durch die TriStates). Daraufhin findet ein Wechsel in den Zustand **AWAIT_RESPONSE_RD/WRREQ** statt, in dem auf eine Antwort des anderen IPIFs gewartet wird.

5. Das IPIF des adressierten Moduls erkennt die Freigabe des Bus-Tokens und wechselt in den Zustand **TT_GET**.

6. In diesem Zustand findet die Allokation des VPBs statt. Es werden folgende Signalzuweisungen mit dem nächsten Takt wirksam:

   - **pot_vip2vpb_Tt** Bus-Token wird allokiert ('1').
   - **pot_vip2ip_Mst** Signalisierung des Master-Zustands an das Modul ('1').
   - **pot_vip2ip_CS** Berechnung des Chip-Selects (in diesem Fall wird das IPIF-Registerfile selektiert).
   - **pot_vip2ip_Busy** FSM des IPIFs befindet sich nicht mehr im Arbeitszustand ('0').

   Des Weiteren wechselt das IPIF mit der nächsten Taktflanke in den Zustand **IDLE_TO**.

7. Im Zustand **IDLE_TO** kann das ursprünglich adressierte IPIF folgende Operationen ausführen:

   - **Lese-/Schreib-Operation lokal ausführen und Antwort an Vorgänger zurückgeben:** Dieser Fall wird nachfolgend geschildert.
   - **Nachfolgendes Modul adressieren (z.B. L2-Cache):** Die Transaktionskette beginnt wieder bei 1., wobei die VBB durch den aktuellen Token Owner zu ersetzen ist.
   - **Ausgabe eines Errors:** In diesem Fall wird dem anfragenden IPIF das Error-Signal **pot_vip2vipb_Error** ('1') übermittelt, in den Zustand **TT_RELEASE_ERROR** gewechselt, in dem das Token abgegeben und damit alle Signale vom Bus getrennt werden, und daraufhin in den Ausgangszustand **IDLE_NTO** zurückgekehrt. Das anfragende IPIF, wechselt bei Erkennen des Error-Signals vom Zustand **AWAIT_RESPONSE_RD/WRREQ** in den Zustand **AWAIT_TOKEN_RELEASE**, in dem auf die Freigabe des Tokens gewartet wird, allokiert in **TT_GET** das Bus-Token und gelangt in den ursprünglichen Zustand **IDLE_TO** zurück.

   Im ersten Fall, in dem die Anfrageoperation lokal ausgeführt wird, wird nach Beendigung dieser Operation eine Antwort-Nachricht an das IPIF durch das Modul übermittelt, die ein Rd/WrAck und die aktualisierte Simulationszeit umfasst. Bedingt durch diesen Input wechselt

die FSM und damit das IPIF mit der nächsten Taktflanke in den Zustand **TT_RELEASE_ACK** und legt die folgende Signale auf den VPB:

- **pot_vip2vpb_TtDir** Adressierung des Vorgängers ('0').
- **pot_vip2vpb_Rd/WrAck** Anfragebestätigung ('1').
- **pot_vip2vpb_SimTime** Die neue Simulationszeit.

Ebenso wird dem internen Modul wieder der Arbeitszustand über das Signal **pot_vip2ip_Busy** ('1') signalisiert und der zwischengespeicherte Anfrage-Kontext gelöscht.

8. Das VBB-IPIF erhält nun im Zustand **AWAIT_RESPONSE_RD/WRREQ** die o.g. Antwort, speichert die Antwortdaten inkl. der aktuellen Simulationszeit zwischen und leitet diese an das anfragende Modul. Es erfolgt ein Wechsel in den Zustand **AWAIT_TOKEN_RELEASE**.

9. Im Zustand **TT_RELEASE_ACK** des adressierten IPIFs werden alle Signale vom VPB durch die Abgabe des Tokens getrennt und in den Ausgangszustand **IDLE_NTO** gewechselt.

10. Die anfragende Modul, bzw. ihr IPIF, erkennt die Busfreigabe, wechselt in den Zustand **TT_GET**, allokiert, wie bereits zuvor beschrieben, den VPB, signalisiert dem Modul den Master-Zustand, berechnet die Chip Select-Signale (nicht im Fall der VBB) und gelangt schlussendlich in den Ausgangszustand **IDLE_TO** zurück, in der die Antwortdaten verarbeitet werden können. Bei einer Transaktionskette können in diesem Zustand die in 7. beschriebenen Aktionen ausgeführt werden, sofern der momentane Token Owner nicht die VBB (also der initiale Token Owner) ist.

Dauer einer Anfrage am VPB

Für die Minimierung der Latenzzeit einer Speicheranfrage ist es relevant die durch den VPB induzierte Latenzzeit zu betrachten. Unter Berücksichtigung der in 3.3.5 beschriebenen Anwendungsfälle ergibt sich eine durchschnittliche Latenzeit von 10 Zyklen für eine Ende-zu-Ende-Kommunikation auf dem VPB bedingt durch das TTP. Die Latenzzeiten der Speichermodule sind zusätzlich zu berücksichtigen.

3.4 Weitere IP Cores

3.4.1 Generisches Registerfile

3.4.1.1 Motivation

Die Busteilnehmer des VPB (**V-POWER** Bus) müssen einige Register für z.B. die Modulkonfiguration, Simulationsergebnisse oder aber auch Befehlsregister enthalten. Daher ist es nötig, jedem Modul eine gewisse Menge an Registern zur Verfügung zu stellen. Um aber den Overhead zu minimieren,

in jedem Busteilnehmer ein eigenes Registerfile implementieren zu müssen, wurde ein generisches Registerfile entwickelt, das im folgenden genauer spezifiziert werden wird.

3.4.1.2 Generics und Ports

Das Registerfile kann über einige VHDL-Generics den individuellen Ansprüchen des jeweiligen Busteilnehmers angepasst werden. Andere sind durch das umgebende System vorgegeben.

Name	Beschreibung
C_VPOWER_REG_AWIDTH	Adressbreite des Registers (in Bit)
C_VPOWER_REG_DWIDTH	Datenbreite des Registers (in Bit)
C_VPOWER_REG_BASEADDR	Base-Adresse des Registerfiles
C_VPOWER_REG_HIGHADDR	High-Adresse des Registerfiles

Tabelle 3.3: *Registerfile Generics*

Es muss stets gelten:

$$(HIGHADDR - BASEADDR + 1)\ mod\ (AWIDTH/8)\ =\ 0 \tag{3.1}$$

Erläuterung: Angenommen die High-Adresse ist 0x7 und die Base-Adresse 0x0. Um nun die Größe (in Byte) des Registerfiles zu ermitteln, muss die Differenz aus diesen beiden Werten ermittelt werden. Da die erste Adresse 0x0 jedoch mitgezählt werden muss, wird noch eine 1 addiert. Zur Bestimmung der Anzahl der Register ist die Byte-Größe des Registerfiles durch die Byte-Adressenbreite (Division durch 8, da **C_VPOWER_REG_AWIDTH** in Bit definiert ist) zu dividieren. Die Gesamtgleichung ergibt sich nun aus der Tatsache, dass diese Division ohne Rest stattfinden muss.

Aus der High-, der Baseadresse und der Adressbreite berechnet die Synthese die Anzahl der Register, die dieses Registerfile enthält. Daher ist es auch unbedingt notwendig den o.g. Constraint einzuhalten. Andernfalls wird die Synthese fehlschlagen.

Signal	I/O Typ	Breite	Beschreibung
pin_ip2reg_Clk	I	1	Takteingang
pin_ip2reg_Rst	I	1	Resetsignal
pin_ip2reg_CS	I	1	Chip-Select
pin_ip2reg_CE	I	1	Chip-Enable
pin_ip2reg_WrCE	I	1	Chip-Enable zum Beschreiben
pin_ip2reg_RdCE	I	1	Chip-Enable zum Lesen
pin_ip2reg_Addr	I	**C_VPOWER_REG_AWIDTH**	Adresse des Registers
pin_ip2reg_Data	I	**C_VPOWER_REG_DWIDTH**	Zu schreibende Daten
pin_ip2reg_RdReq	I	1	Determiniert Leseanfrage
pin_ip2reg_WrReq	I	1	Determiniert Schreibanfrage
pot_reg2ip_AddrAck	O	1	Bestätigung einer Anfrage
pot_reg2ip_RdAck	O	1	Bestätigung einer durchgefhrten Leseoperation

Signal	I/O Typ	Breite	Beschreibung
pot_reg2ip_WrAck	O	1	Bestätigung einer durchgefhrten Schreiboperation
pot_reg2ip_Data	O	**C_VPOWER_REG_DWIDTH**	Antwortdaten

Tabelle 3.4: *Registerfile Ports*

3.4.1.3 Funktionsweise

Um ein Wort des Registerfiles auszulesen, müssen die folgenden Signale die definierten Werte annehmen:

pin_ip2reg_CS '1'
pin_ip2reg_CE '1'
pin_ip2reg_RdCE '1'
pin_ip2reg_RdReq '1'
pin_ip2reg_Addr Auszulesende Adresse (aligned)

Das Beschreiben des Registerfiles erfolgt auf analoge Art und Weise:

pin_ip2reg_CS '1'
pin_ip2reg_CE '1'
pin_ip2reg_WrCE '1'
pin_ip2reg_WrReq '1'
pin_ip2reg_Addr Auszulesende Adresse (aligned)
pin_ip2reg_Data Zu schreibende Daten

3.4.2 Speicherzugriffsmodul

Der IP-Core **vpower_mem_access_v1_00_b** bietet dem Nutzer mit dem im Rahmen dieser PG erstellten VHDL-Modul vpower_mem_access eine Schnittstelle zum Lesen und Schreiben von Datenwörtern aus dem bzw. ins DIMM-Modul, die auf einfache Weise verwendet werden kann. Im Vergleich zur Schnittstelle des DDR-Controllers der Firma Xilinx (**ddr_v1_11_a.ddr_controller**) wurde die Anzahl der Eingaben auf das Nötigste verringert. Das Modul ersetzt diesen Controller aber keinesfalls, sondern wird diesem nur vorgeschaltet. Die Funktionalität des DDR-Controllers bildet weiterhin die Grundlage für die Ausführung von Speicheroperationen.

Die Ausgangsports zum DDR-Controller können durch Tri-State-Buffer von der übrigen Logik im Modul getrennt werden. Auf diese Weise ist es möglich, mehrere Module an einem Bus zu betreiben, die alle mit nur einem einzigen DDR-Controller kommunizieren. Wie im Falle des PG-Entwurfes, bei dem es nur ein DIMM-Modul gibt, ist es ohnehin nur sinnvoll, einen Controller für dieses DIMM-Modul zu instanziieren und die komplexe Logik nicht zu duplizieren, obwohl in mehreren Modulen

Speicheroperationen durchführen werden müssen. Die Arbitrierung des Busses wird von den Modulen nicht vorgenommen und muss durch den Nutzer erfolgen.

3.4.2.1 Schnittstellendefinition

Die Ports des Speicherzugriffsmoduls können logisch zwei Schnittstellen zugeordnet werden. Notwendigerweise weist das Modul eine Schnittstelle zum oben genannten DDR-Controller auf. Die Eingangs- und Ausgangsports dieser Schnittstelle müssen mit den bedeutungsgleichen Ausgangs- und Eingangsports des DDR-Controllers verbunden werden. Über die Ports der zweiten Schnittstelle erfolgt die Kommunikation mit der Nutzer-Schaltung. Dies sind:

Signal	I/O Typ	Breite	Beschreibung
pin_Clk	I	1	Taktsignal
pin_Rst	I	1	Reset-Signal
pin_CS	I	1	Modul aktivieren
pin_Addr	I	**C_AWIDTH**	Speicheradresse des ersten Datums
pin_Data	I	**C_DWIDTH**	Datum, das in den Speicher geschrieben werden soll
pin_RNW	I	1	Read Not Write-Signal, zur Unterscheidung von Lese- ('1') und Schreibzugriffen ('0')
pin_Req	I	1	Ausführung einer Speicheroperation anfordern
pin_Mode	I	2	Operationsart
pin_DblWord	I	1	die übertragenden Daten sind Wörter ('0') oder Doppelwörter ('1')
pin_Quantity	I	4	Anzahl der Daten **-1** in Binärdarstellung
pot_Data	O	**C_DWIDTH**	Datum, das aus dem Speicher gelesen wurde
pot_Done	O	1	signalisiert, dass die Speicheroperation erfolgreich durchgeführt wurde
pot_Error	O	1	signalisiert, ob bei einer Speicheroperation ein Fehler auftrat

Tabelle 3.5: *Schnittstelle zur Schaltung des Nutzers*

Das Taktsignal am Port **pin_Clk** muss synchron zu dem Taktsignal des DDR-Controllers sein, damit eine fehlerfreie Kommunikation zwischen den zwei Komponenten gewährleistet ist. Wie üblich werden alle Signale zu steigenden Taktflanken verarbeitet. Dies gilt auch für das Reset-Signal, d. h. es sind nur synchrone Resets möglich. Durch Setzen des Reset-Signals am Port **pin_Rst** wird das Speicherzugriffsmodul in den initialen Zustand versetzt. Ein Zurücksetzen des DDR-Controllers erfolgt dadurch aber nicht.

Ist der Pegel am Port **pin_CS** hoch ('1'), befindet sich das Modul im aktiven Zustand. Nur in diesem Zustand können Speicheroperationen durchgeführt werden, da dann die Tri-State-Buffer die Ausgaben der Schaltung an die Ausgangsports zum DDR-Controller weiterleiten. Im passiven Zustand sind dagegen die Verbindungen zum DDR-Controller durch die Tri-State-Buffers im Modul unterbro-

chen und auch alle Eingaben werden ignoriert.

Mit den Pegeln an den übrigen Eingabeports werden schließlich die Speicheroperationen gesteuert. Eine logische Eins (’1’) am Port **pin_RNW** unterscheidet Leseoperationen von schreibenden Speicherzugriffen, bei denen eine logische Null (’0’) anliegen muss. Durch eine ’1’ am Port **pin_DblWord** wird festgelegt, dass jedes Datum, das an der Nutzer-Schnittstelle übertragen wird, aus einem Doppelwort besteht. Liegt eine logische Null an, werden nur die weniger signifikanten 32 Bits der Datenports zur Übertragung der Daten verwendet und jedes Datum besteht nur aus einem Wort.

Mit einer ’1’ am Port **pin_Req** wird signalisiert, dass an allen Eingangsports der Nutzer-Schnittstelle gültige Werte anliegen. Auf diese Weise werden die Speicheroperationen begonnen. Bis zum Ende der Speicheroperation müssen von nun an an allen anderen Ports die Werte unverändert beibehalten werden. Ein Ausnahme bilden die Werte an den Datenports **pin_Data**. Sollen bei einem schreibenden Zugriff mehrere Daten an den Speicher übermittelt werden, so müssen diese nacheinander an den Datenports **pin_Data** angelegt werden.

Die Anzahl der Daten, die bei der Operation aus dem Speicher gelesen oder in ihn geschrieben werden, muss immer eine Zweierpotenz sein. Sie ist abzüglich Eins in Binärdarstellung an den Ports **pin_Quantity** anzulegen. Je nachdem welcher Pegel am **pin_DblWord** anliegt, bezieht sich diese Anzahl auf Wörter oder Doppelwörter.

Müssen in einer Operation insgesamt mehr als 64 Bit zum oder vom Speicher übertragen werden, so ist dies nur im Cacheline-Mode möglich. Eine Operation kann dann bis zu 16 Datenwörter umfassen, die aufeinanderfolgende Speicheradressen aufweisen müssen. Dazu muss an den Ports **pin_Mode** der Wert ”01” anliegen. Operationen, die nicht mehr als ein Doppelwort oder zwei Wörter umfassen, werden im Single-Mode ausgeführt. Hier muss der Wert ”00” an den Ports **pin_Mode** anliegen.

Unabhängig davon, wie viele Datenwörter gelesen oder geschrieben werden, muss immer die Speicheradresse des ersten Datums an den Ports **pin_Addr** angelegt werden. Wie bereits oben erwähnt, darf sie während der Operation nicht verändert werden, also auch nicht erhöht werden. Ausgehend von der Speicheradresse des ersten Datums erhöht das Speicherzugriffsmodul selbständig die Adressen für die darauffolgenden Daten.

Der Ausgangsport **pot_Done** dient zur Signalisierung, ob Operationen vollständig und erfolgreich durchgeführt werden konnten. Bei Schreiboperationen liegt für einen Takt eine ’1’ an. Bei Leseoperationen liegt jedesmal, wenn ein valides Datum an den Ports **pot_Data** zurückgegeben wird, gleichzeitig für einen Takt ebenfalls eine ’1’ an. In beiden Fällen wird über den Port **pot_Error** fortwährend ’0’ ausgegeben. Trat während einer Operation hingegen ein Fehler auf, liegt statt am Port **pin_Done** am Port **pin_Error** eine logische Eins an. Ein Fehler tritt in jedem Fall auch dann auf, wenn ein Wert oder eine Kombination von Werten an den Eingangsports unzulässig ist.

Eine vollständige Übersicht über alle Ports befindet sich im Anhang unter A.3.

Obwohl die Busbreiten der Adress- und der Datenports durch Generics festgelegt sind, bestehen hier keine Wahlmöglichkeiten. Damit das Speicherzugriffsmodul korrekt funktioniert muss der Gene-

ric **C_AWIDTH** auf 32 und **C_DWIDTH** auf 64 gesetzt werden. Dies sind auch die Voreinstellungen. Darüber hinaus gibt es noch drei weitere Generics, deren Wert der Nutzer auch tatsächlich frei wählen darf.

Name	Beschreibung
C_AWIDTH	Typ: integer; Breite des Adressports - muss auf 32 gesetzt sein
C_DWIDTH	Typ: integer; Breite der Datenports - muss auf 64 gesetzt sein
C_MEM_FAST_DELIVERY_MODE	Typ: boolean; wenn 'true' gibt es zwischen der Rückgabe von zwei ausgelesenen Daten keinen Wartezyklus
C_MEM_FAST_WRITE_MODE	Typ: boolean; wenn 'true' erfolgen Schreibzugriffe im *Fast Mode*, ansonsten im *Slow Mode*
C_MEM_FAST_READ_MODE	Typ: boolean; wenn 'true' erfolgen Lesezugriffe im Cacheline-Mode im *Fast Mode*, ansonsten im *Slow Mode*

Tabelle 3.6: *Generics des Speicherzugriffsmoduls*

Der DDR-Controller, dem das Speicherzugriffsmodul vorgeschaltet ist, unterstützt bei den Speicheroperationen zwei Modi. Sie können entweder im *Slow Mode* oder im *Fast Mode* durchgeführt werden. Für die Durchführung von Speicheroperationen, die mehrere Datenwörter umfassen, werden im *Fast Mode* in der Regel weniger Taktzyklen benötigt. Dies setzt aber voraus, dass beim DDR-Controller das Generic **C_INCLUDE_BURSTS** auf 1 gesetzt ist. Über die Werte der Generics **C_MEM_FAST_WRITE_MODE** und **C_MEM_FAST_READ_MODE** kann festgelegt werden, auf welche Weise der DDR-Controller angesteuert werden soll. Auf Lesezugriffe im Single-Mode wirkt sich dies jedoch nicht aus. Hier wird nicht zwischen den beiden Modi unterschieden.

Mit dem Wert des Generics **C_MEM_FAST_DELIVERY_MODE** kann ein wenig Einfluss darauf genommen werden, wie die Daten nach einem Lesezugriff ausgegeben werden. Ist es auf 'true' gesetzt und es sollten N Daten gelesen werden, liegt bei erfolgreicher Durchführung der Operation an N aufeinanderfolgenden Takten am Port **pot_Done** eine logische Eins an und zu jeder Taktflanke wird fortlaufend ein Datum zurückgegeben. Ist der Wert dagegen 'false' wird zwischen der Ausgabe von zwei Daten jeweils ein Wartezyklus eingefügt. In diesem Takt liegen die Daten weiterhin stabil an den Ausgangsports **pot_Data** an, so dass die Schaltung des Nutzers einen Takt mehr Zeit hat, um die Ausgaben zu übernehmen. Der Wert des Ports **pot_Done** wechselt zu diesem Takt jedesmal wieder auf eine '0'. Somit gilt weiterhin, dass zu N steigenden Taktflanken am Port **pot_Done** eine '1' anliegt, ehe die Operation abgeschlossen ist.

3.4.2.2 Ausführung von Speicheroperationen

Das Speicherzugriffsmodul erlaubt die Ausführung von lesenden und schreibenden Speicheroperationen mit bis zu 16 Datenwörtern. Die Operationen können prinzipiell immer in zwei Phasen unterteilt werden. In der ersten Phase werden die Datenwörter vom Modul eingelesen, und in der zweiten Phase wieder ausgegeben. An welchen Schnittstellen die Daten eingelesen bzw. ausgegeben werden, hängt von der Operation ab. Bei Lesezugriffen werden erst alle Datenwörter vom

DDR-Controller erfragt, und dann an der Schnittstelle zur Schaltung des Nutzers ausgegeben. Bei Schreiboperationen wird zunächst gewartet bis alle Daten von der Schaltung des Nutzers eingegeben wurden, bevor sie dem DDR-Controller übermittelt werden.

Zum Ausführen einer Operation muss sich das Speicherzugriffsmodul im aktiven Zustand befinden. Bevor am Port **pin_Req** das erstemal eine logische Eins zu einer steigenden Taktflanke anliegt, müssen an allen anderen Eingangsports die Werte gemäß der Schnittstellendefinition stabil anliegen.

Liegt am Port **pin_RNW** eine '1' an, wird eine Leseopertion durchgeführt. Diese startet sofort, wenn zu einer steigenden Taktflanke am Port **pin_Req** ebenfalls eine '1' anliegt. Zur nächsten steigenden Taktflanke sollte der Wert am Port wieder auf '0' gesetzt werden. Das Speicherzugriffsmodul liest nun mittels des DDR-Controllers beginnend an der angegebenen Speicheradresse so viele aufeinanderfolgende Datenwörter aus dem Speicher aus, wie an den Ports **pin_Quantity** zuzüglich Eins angegeben ist. Dabei muss es sich immer um eine Zweierpotenz handeln. Müssen nur zwei Datenwörter oder ein Datendoppelwort gelesen werden, geschieht dies im Single-Mode. Bei mehreren Datenwörtern muss der Cacheline-Mode eingestellt sein.

Wurden alle Datenwörter gelesen, ohne dass ein Fehler auftrat, wird dies mit einer '1' am Port **pot_Done** signalisiert, und gleichzeitig die Datenwörter, so wie sie auch gelesen wurden, in aufsteigender Reihenfolge ihrer Speicheradressen zurückgegeben. Nacheinander werden die Daten an den Ausgangsports **pot_Data** angelegt. Ist der Wert des Generics **C_MEM_FAST_DELIVERY_MODE** auf 'true' gesetzt, wird zu jeder steigenden Taktflanke ein anderes Datum ausgegeben. Der Wert am Port **pot_Done** bleibt so lange eine '1', bis das letzte Datum ausgegeben wurde. Erst im darauf folgenden Takt, gibt der Port wieder eine logische Null aus.

Wurde das Generic **C_MEM_FAST_DELIVERY_MODE** auf 'false' gesetzt, werden ebenfalls die Daten nacheinander zurückgegeben. Es wird aber nur zu jeder zweiten steigenden Taktflanke ein weiteres Datum ausgegeben. Dies liegt dafür dann an beiden aufeinanderfolgenden steigenden Taktflanken stabil an den Datenausgangsports.

In der so entstehenden Pause entspricht der Pegel am Port **pot_Done** jedesmal wieder einer logischen Null, so dass an diesem Port nur dann eine '1' anliegt, wenn zu dem gleichen Takt ein weiteres Datum ausgegeben wurde. Bis alle Daten ausgegeben wurden, wechselt der Wert am Port **pot_Done** somit mit jedem Takt. Nach der Ausgabe des letzten Datums kann sofort die nächste Lese- oder Schreiboperation begonnen werden.

Eine Schreiboperation wird dann ausgeführt, wenn am Port **pin_RNW** eine logische Null anliegt. Bevor der eigentliche Zugriff auf den Speicher mittels des DDR-Controllers erfolgt, werden zunächst alle Datenworte über die Ports **pin_Data** eingelesen. Ihre Anzahl wird durch den Wert an den Ports **pin_Quantity** bestimmt, zu dem eine Eins hinzuzuaddieren ist. Jedesmal wenn ein Datum von dem Speicherzugriffsmodul eingelesen werden soll, muss dies für einen Takt stabil an die Datenports **pin_Data** angelegt werden, und der Pegel des Ports **pin_Req** im gleichen Zeitraum auf '1' gesetzt werden. Sofort innerhalb dieses Taktes wird bei der nächsten steigenden Taktflanke das Datum

eingelesen. Zwischen dem Einlesen zweier Daten kann eine beliebig lange Zeitspanne liegen. D.h. auch, dass alle Daten ohne Wartezyklen an das Speicherzugriffsmodul übergeben werden können. Entscheidend für die Übernahmen der Daten sind nur die steigenden Taktflanken und der Wert am Port **pin_Req**. An dem Port muss zu insgesamt N steigenden Taktflanken eine '1' anliegen, wenn N Daten geschrieben werden sollen. Das erste Datum wird bereits eingelesen, wenn die Schreiboperation mit einer logischen Eins am Port **pin_Req** gestartet wird. Zu diesem Zeitpunkt müssen auch schon die Werte an den übrigen Eingangsports stabil anliegen. Sind schließlich alle Daten an das Speicherzugriffsmodul übertragen worden, erfolgt das Schreiben der Daten in den Speicher. War dies erfolgreich, ist der Wert am Ausgangsport **pot_Done** für einen Takt eine '1'.

Wie schon bei den Leseoperationen kann auch nach Beendigung der Schreiboperation sofort im nächsten Takt die nächste Operation ausgeführt werden. Dies gilt auch dann, wenn eine Operation nicht erfolgreich war. In diesem Fall entspricht der Pegel am Port **pot_Error** einer logischen Eins und die Operation wird sofort abgebrochen. Dies bedeutet, dass bei Leseoperationen kein einziges Datum ausgegeben wird, auch wenn bereits einige Datenwörter aus dem Speicher gelesen wurden. Schreiboperationen sind dagegen als nicht atomar anzusehen. Es werden keine Aussagen darüber getroffen, welche Daten erfolgreich geschrieben werden konnten, bis der Fehler auftrat, und welchen Inhalt der Speicher an den adressierten Speicherstellen aufweist. Der Pegel am Port **pot_Error** wird sobald wie möglich, aber mit mindestens einem Takt Verzögerung, wieder auf '0' gesetzt, wenn der DDR-Controller für eine neue Operation wieder bereit ist, oder nachdem im Falle einer fehlerhaften Eingabe der Pegel am Port **pin_Req** wieder '0' ist.

3.4.2.3 Implementierung

Zur Realisierung des Speicherzugriffsmoduls vpower_mem_access wurde neben primitiven logischen Gattern, ein Steuerwerk und drei Zähler zur Koordination der Speicheroperationen, und 64 8 Bit-Schieberegister für die Zwischenspeicherung der Daten benötigt. Neben dem eigentlichen vpower_mem_access-Modul umfasst der IP-Core vpower_mem_access_v1_00_b daher noch Untermodule, die diese Komponenten beschreiben. Der IP-Core besteht aus insgesamt vier VHDL-Entwurfseinheiten:

- **vpower_mem_access.vhd**
- **vpower_mem_access_controller.vhd**
- **vpower_counter.vhd**
- **vpower_shiftReg.vhd**

Das VHDL-Modul **vpower_mem_access_controller.vhd** beschreibt das Steuerwerk des Speicherzugriffsmoduls. Es implementiert den Zustandsautomaten des Moduls, der alle Abläufe in dem Modul steuert. Für die Ausführung der Speicheroperationen imitiert er an der Schnittstelle zum DDR-

Controller das Verhalten des PLB-IPIFs **plb_ipif_v1_00_f** der Firma Xilinx, wie es in den Datenblättern zum PLB-IPIF [10] angegeben ist.

Wie bei dem Entwurf von Steuerwerken üblich erfolgt eine strikte Trennung der Kontroll- und Steuersignale von den Signalen des Datenpfades. Auf den Datenpfad wird nur mittels Steuersignalen eingewirkt, die das Zwischenspeichern oder Ausgeben der Daten veranlassen können. Bei dieser Betrachtung werden auch die Inhalte der Zähler als Daten angesehen, so dass deren Inhalte ebenfalls nicht an das Steuerwerk geleitet werden.

Einer der Zähler der, die durch das VHDL-Modul **vpower_counter.vhd** beschrieben sind, wird für das Inkrementieren der Speicheradressen verwendet. Zu Beginn einer Speicheroperation wird die angelegte Adresse bis auf die drei am wenigsten signifikanten Bits in den Zähler geladen. Die drei Bits, die nicht in den Adresszähler geladen werden, werden direkt an den DDR-Controller durchgeleitet, da sie sich während der Speicheroperation nicht ändern. Dies liegt daran, dass sobald mehr als ein Datenwort übertragen werden soll, je zwei Datenwörter vor der Übertragung zu einem Doppelwort zusammengefasst werden, wenn nicht schon ohnehin Doppelwörter an der Schnittstelle zur Schaltung des Nutzers übertragen werden. Die Speicheradressen müssen somit immer um Acht erhöht werden. Der einfache Zähler, der seinen Inhalt immer nur um Eins erhöht, macht genau dies, weil er nur die oberen 29 der Adresse berücksichtigt. Der Wert dieses Zählers wird genau dann erhöht, wenn der DDR-Controller am Port **pin_mem_AddrAck**, die Übernahme der Adresse durch eine logische Eins bestätigt hat.

Ein zweiter Zähler zählt die Anzahl der übertragenden Daten. Vor jeder Übertragung ist sein Wert auf Null gesetzt. Zu jeder Zeit wird sein Wert mit dem Wert an den Ports **pin_Quantity** verglichen, der die Gesamtanzahl der zu lesenden oder schreibenden Daten abzüglich Eins angibt. Dass der Wert an den Ports um Eins kleiner ist als die tatsächliche Anzahl, hat den Vorteil, dass einfacher festgestellt werden kann, ob in einem Takt gerade die letzte Transaktion erfolgt. Dazu muss der Wert des Zählers mit dem Wert an den Ports übereinstimmen und in dem Takt eine weitere Transaktion statt finden. In solch einem Fall kann dann das Steuerwerk bereits zum nächsten Takt in einen anderen Zustand wechseln und mit der weiteren Ausführung der Speicheroperation fortfahren. Eine schnellere Reaktion ist somit ein weiterer Vorteil dieses Vorgehens.

Bei dem Vergleich des Zählerwertes mit der Anzahl der zu übertragenden Daten muss berücksichtigt werden, dass einzelne Datenwörter zu Doppelwörtern zusammengefasst werden, bevor sie über die Schnittstelle zum DDR-Controller übertragen werden. Während diese Tatsache keinen Einfluss auf den Datentransfer an der Schnittstelle zur Schaltung des Nutzer hat, müssen an der Schnittstelle zum DDR-Controller nur halb so viele Datenübertragungen erfolgen, wenn die Schaltung des Nutzers nur mit einzelnen Datenwörtern operiert. Folglich muss in diesem Fall der Wert des Zählers mit den Wert verglichen werden, der der Hälfte des an den Ports **pin_Quantity** anliegenden Wertes entspricht. Da der Wert an den Ports **pin_Quantity** nie gerade ist, muss er nach der Halbierung noch abgerundet werden. Dies wird genau dann erreicht, wenn bei dem Vergleich alle Bits des an den Ports anliegenden Wertes bildlich um eine Position nach rechts, also in Richtung des weniger signifikanten Bits, geschoben werden und die höchste signifikante Stelle mit einer Null aufgefüllt

wird.

Um das Modul bei der Ausgabe mehrerer Daten schon bei der Ausgabe des vorletzten Datums auf das Ende der Operation vorbereiten zu können, wird der Inhalt des Zählers vor der Ausgabe des ersten Datums um Eins erhöht, so dass er gegenüber der tatsächlich übertragenden Anzahl an Daten immer um Eins zu groß ist. Dies wird genutzt, um mit der Ausgabe des vorletzten Datums in einen neuen Zustand zu wechseln. In diesem wird bei Schreibzugriffen im *Fast Mode* dem DDR-Controller die Übergabe des letzten Datums signalisiert. Bei Lesezugriffen wird ebenfalls noch das letzte Datum ausgegeben und gleichzeitig die Zähler zurückgesetzt, damit im nächsten Takt schon die nächste Operation begonnen werden kann.

Der dritte Zähler wird nur dazu verwendet die Anzahl der Adressübernahmebestätigungen des DDR-Controllers zu bestimmen. Im sogenannten *Leading Mode* erfolgen die Adressübernahmebestätigungen nicht synchron zu den Bestätigungen der Daten. Jene müssen daher seperat von der Anzahl der übertragenden Daten ermittelt werden, was einen eigenen Zähler erfordert. Die Kenntnis ist nicht nur wichtig, um feststellen zu können, ob jede Übertragung bestätigt wurde. Im *Fast Mode* muss dem DDR-Controller mit der Übertragung der letzten Adresse das Ende der Gesamtoperation angezeigt werden. Dies ist nur möglich, wenn die Anzahl der übertragenden und bestätigten Addressen zu jeder Zeit bekannt ist.

Obwohl es laut den Datenblättern des PLB-IPIFs [10] zu erwarten ist, dass der *Leading Mode* nur bei Leseoperationen im *Fast Mode* Anwendung findet, zeigte sich bei Tests mit dem DDR-Controller, dass bei allen Schreib- und Lesemodi, die Bestätigungen asynchron erfolgen können.

In der VHDL-Datei **vpower_shiftReg.vhd** sind die Schieberegister beschrieben. Das Speicherzugriffsmodul besitzt davon 64 Stück mit 8 Bit Länge. Sie werden zur Zwischenspeicherung der Daten verwendet, die je nach Art der durchzuführenden Operation zunächst entweder an der Schnittstelle zum DDR-Controller oder der Schnittstelle zur Schaltung des Nutzers eingelesen werden und dann anschließend an der jeweils anderen Schnittstelle ausgegeben werden. Um Logikbausteine einzusparen, werden zur Zwischenspeicherung immer die selben Register verwendet, unabhängig von welcher Schnittstelle die Daten eingelesen wurden. Von welche Schnittstelle den Schieberegistern die Daten zugeführt werden, wird an Hand des Wertes des Ports **pin_RNW** ermittelt. Bei einer logischen ’1’, also einer Leseoperation, werden die Daten von der Schnittstelle zum DDR-Controller zugeführt, andernfalls von der Schnittstelle zur Schaltung des Nutzers.

Jeweils 32 Schieberegister bilden einen Registersatz, der jeweils ein Datenwort speichert. Alle Schieberegister in einem Registersatz sind daher parallel geschaltet. Die zwei Registersätze können jedoch unabhängig von einander gesteuert werden. Auf diese Weise ist es möglich, sowohl einzelne Datenwörter als auch Daten, die aus zwei Wörtern bestehen, zu speichern. Überträgt die Schaltung des Nutzers nur einzelne Datenwörter, was durch eine ’1’ am Port **pin_DblWord** signalisiert wird, werden diese abwechselnd in die zwei Registersätze geladen und so zu Doppelwörtern zusammengefasst. An der Schnittstelle zum DDR-Controller werden schließlich immer nur Doppelwörter übertragen, sobald mehr als ein einzelnes Datenwort gelesen oder geschrieben werden

muss. Im Vergleich zur Übertragung einzelner Datenwörter halbiert dies zum einen die Anzahl der Datenübertragungen, zum anderen können dadurch Operationen mit doppelt so vielen einzelnen Datenwörter durchgeführt werden. Bei Lesezugriffen können die vom DDR-Controller empfangenen Doppelwörter wieder in einzelne Datenwörter zerlegt an die Schaltung des Nutzers ausgegeben werden.

Da die Schieberegister eine Länge von 8 Bit haben, können in den 64 Registern bis zu 8 Doppelwörter zwischengespeichert werden. Dies ist gleichzeitig die maximale Anzahl an Daten, die bei einer einzelnen Speicheroperation übertragen werden kann, und somit ausreichend. Die Ausgaben der Schieberegister können nach dem ersten, zweiten, vierten oder achten Bit abgegriffen werden. So können die Daten, auch wenn weniger als 8 Doppelwörter gespeichert werden müssen, immer sofort ausgelesen werden, und müssen nicht erst bis ans Ende des Schieberegisters durchgeschoben werden. Von welcher Stelle aus die Daten an die Ausgangsports weitergeleitet werden, wird durch den Wert an den Ports **pin_Quantity** bestimmt, der die Anzahl der zu übertragenden Daten festlegt.

3.4.2.4 Fazit

Das VHDL-Speicherzugriffsmodul vpower_mem_access ermöglicht Speicherzugriffe auf einfache Art und Weise. Dies versetzt den Nutzer in die Lage, dass er die Funktionsweise des DDR-Controllers, dem das Modul vorgeschaltet wird, nicht kennen muss. Gleichzeitig wurde eine Schnittstelle zum Nutzer geschaffen, die ein Austauschen des DDR-Controllers durch einen anderes Steuerwerk erlaubt, eventuell sogar für eine andere Speicherart, ohne dass der Entwurf des Nutzers angepasst werden muss. Nur eine Anpassung des vpower_mem_access-Moduls ist erforderlich. Da unter Umständen an verschiedenen Stellen im Entwurf Speicherzugriffe von unterschiedlichen Modulen durchgeführt werden müssen, kann dies eine erhebliche Arbeitserleichterung bedeuten.

Obwohl der DDR-Controller neben den zwei Modi Single- Mode und Cacheline-Mode noch zwei Modi für Bursts fester und variabler Länge unterstützt, hat es sich gezeigt, dass durch die Implementierung der ersten beiden Modi der Bedarf für die PG vollständig abgedeckt ist. Zur Auswahl der zwei übrigen Modi waren ursprünglich die zwei verbleibenden Werte am Port **pin_Mode** gedacht. An den Generics **C_AWIDTH** und **C_DWIDTH** kann man zudem erkennen, dass zunächst variable Busbreiten angedacht waren. Auch dies wurde nicht umgesetzt, da sich hierfür ebenfalls kein Bedarf ergab und so der Implementierungsaufwand eingespart werden konnte. Die Möglichkeit durch Anlegen einer logischen Null am Port **pin_DblWord** die verwendete Busbreite der Datenports auf 32 Bit einzuschränken reicht für den Entwurf der PG aus.

Die Funktionalität des Speicherzugriffsmodul konnte in ModelSim erfolgreich getestet werden.

3.5 Taktung des PowerPC

Da real ausschließlich auf einen Speichertyp zugegriffen wird (DIMM-Speicher), muss bei der Speicherhirarchie-Simulation der Prozessor u.a. angehalten werden, um die unterschiedlichen Energie-Kosten beim Zugriff auf die einzelnen Speicher-Stufen zu simulieren. Deshalb ist es notwendig den PowerPC Takt kontrollieren zu können. Mit “Kontrolle” ist das deterministische Anhalten und wieder Starten des PowerPC-Taktes gemeint. Was hier zunächst als einfaches Problem eingeschätzt wurde, stellte sich im Verlauf des ersten PG-Semesters als schwierig heraus und ist noch nicht gelöst. Der Takt kann nicht ohne Nebenwirkungen durch ein definiertes Signal (repräsentiert durch eine Logik) einfach ab- und wieder eingeschaltet werden. Beispielsweise muss die Phasensynchronisation zwischen Prozessor- und PLB-Takt beachtet werden. Außerdem muss man sicherstellen, dass auf dem PLB keine Transaktionen mehr stattfinden, wenn der PowerPC-Takt angehalten wird. Erschwert wird diese Arbeit durch die nicht vorhandene Dokumentation dieses Praxisproblems seitens Xilinx. Auch im Internet sind keine Informationen, die direkt auf dieses Problem eingehen, zu finden. Daher ist die PowerPC Taktkontrolle in das Kapitel “Forschung” eingeordnet worden. Es wurde im bisherigen Verlauf der PG versucht anhand VHDL-Code-Analyse und Tests auf dem Xilinx-FPGA zu verstehen, wie die Taktung funktioniert, welche Hardware darauf Einfluss hat und wie sie letztendlich für die Zwecke der PG kontrollieren werden kann. Es werden daher hier zunächst einige theoretische Grundlagen über die Bausteine vorgestellt, die mit der Taktung zu tun haben, bzw. welche direkt durch ein Anhalten des Prozessortaktes beeinflusst werden. Begonnen wird hierbei mit einer Einführung in den Digital Clock Manger (DCM), der die unterschiedlichen Systemtakte liefert. Es folgt eine kurze Beschreibung der von Xilinx zur Verfügung gestellten Clock-Buffer, um Taktdomänen zu synchronisieren (Taktphasensynchronisation), gefolgt von der Beschreibung des Processor Local Bus (PLB), da insbesondere dieser eine zentrale Rolle für die Lösung des Problems zu spielen scheint. Zum Abschluss dieses Kapitels wird noch einmal detailliert das bisherige Vorgehen und die dadurch erworbenen Erkenntnisse beschrieben.

3.5.1 Digital Clock Manager (DCM)

Höhere System-Bandbreiten können nur durch höhere Daten-Raten zwischen Komponenten in einem System realisiert werden. Den Datendurchsatz bestimmt der Takt, und nicht jede Komponente kann mit der maximal erzeugten Taktrate betrieben werden. Je tiefer die Logik einer Schaltung, desto längere Schaltzeiten besitzt diese, und um so niedriger ist die Taktrate, mit der sie betrieben werden kann. Um nicht alle Komponenten mit der niedrigsten Taktrate zu betreiben ist es sinnvoll, wenn mehrere Frequenzen in einem System zur Verfügung gestellt werden und jede Komponente mit einer angemessenen Taktrate betrieben wird. Komponenten mit ungefähr gleich schnellen Taktraten können in gleichen Taktdomänen untergebracht werden. Es muss allerdings darauf geachtet werden, dass ein synchroner Signalübergang zwischen den verschiedenen Taktdomänen herrscht. Der DCM ist diesen Anforderungen gewachsen. Er verwendet selbst kalibrierende Taktverteilung, Laufzeitkompensation, Taktvervielfachung, Taktteilung sowie Taktphasenverschiebung.

Dem PPC405 wird mit dem DCM ein eigenes Taktsignal, welches um ein Vielfaches schneller als der Systemtakt ist, zur Verfügung gestellt.

Clock Management Besonderheiten

Nachfolgend werden die Besonderheiten des DCM etwas näher erläutert.

Clock De-Skew

Der DCM generiert neue Systemtakte, die zu dem Eingangstakt ausgerichtet sind. Ein synchrones System hängt von einer präzisen Verteilung des Taktsignals ab. Der DCM verteilt das Taktsignal so, dass keine Laufzeitdifferenzen zwischen den einzelnen Komponenten auftreten.

Frequency Synthesis

Der DCM generiert ein breites Spektrum von Ausgangstakten mit unterschiedlicher Frequenz. Diese werden mit einfachen Multiplizierern und Dividierern aus dem Systemtakt realisiert. Für den Multiplizierer kann ein Faktor von 2 bis 32 und für den Dividierer einer von 1 bis 32 gewählt werden. Mit dieser Eigenschaft lassen sich Hochgeschwindigkeits-OnBoard-Systemtakte realisieren.

Phase Shifting

Der DCM liefert grobkörnige sowie feingranulare Phasenverschiebung mit dynamischer Einstellmöglichkeit. Damit lassen sich z.B. gängige Phasenverschiebungen, die eine 90, 180 oder 270 Grad Phasenverschiebung zum Systemtakt aufweisen sollen, realisieren.

Die folgende Abbildung 3.12 zeigt alle Eingangs- und Ausgangsports des DCM sowie alle seine Kontroll- und Statusregister.

Die einzelnen Ein- und Ausgänge des DCM sind in Tabelle 3.7 genauer beschrieben.

Signal	I/O Typ	Art	Beschreibung
CLKIN	I	Taktsignal	Dieser Takteingang wird mit dem von Board erzeugten Takt (100MHz) gespeist.

Signal	I/O Typ	Art	Beschreibung
CLKFB	I	Taktsignal	Dient als Feedback vom CLK0 oder CLK2X fr den DCM. Dadurch können alle vom DCM erzeugten Taktausgangssignale synchron mit dem Takteingangssignal erzeugt werden. Das hat den Vorteil, dass Taktsignale anderer DCMs ihre Taktsignale mit anderen DCMs nicht synchronisieren müssen.
RST	I	Kontrollsignal	Das Master Rest-Signal. Setzt den DCM bei Aktivierung zum initialen PowerOn Zustand zurück.
DSSEN	I	Kontrollsignal	Steht für ”Digital Spread Spectrum Enabled”. Aktiviert bzw. deaktiviert das ”Digital Spread Spectrum”.
PSINCDEC	I	Kontrollsignal	Inkrementiert bzw. dekrementiert den dynamischen Phasenverschieber. Wenn PSINCDEC = 1, dann wird der Wert des dynamischen Pahsenverschiebers erhöht, sonst vermindert.
PSEN	I	Kontrollsignal	Steht für ”Dynamic Phase Shift Enabled”. Wenn PSEN = 0, dann wird der dynamische Phasenverschieber deaktiviert und alle Eingangssignale ignoriert. Wenn PSEN = 1, dann wird die dynamische Phasenverschiebungsoperation mit der nächsten steigenden Taktflanke aktiviert.
PSCLK	I	Taktsignal	PSCLK ist die Abkürzung für ”Phase Shift Clock”. Takteingang für den dynamischen Phasenverschieber. Getaktet mit der steigenden Taktflanke.
CLK0	O	Taktsignal	Wird meistens für den Systemtakt verwendet. Die 0 deutet darauf hin, dass das Taktsignal keine Phasenverschiebung zum Eingangstakt aufweist.
CLK90	O	Taktsignal	Hat die gleiche Frequenz, wie das CLK0 Taktsignal, ist jedoch um 90 Grad zu diesem phasenverschoben.
CLK180	O	Taktsignal	Hat die gleiche Frequenz, wie das CLK0 Taktsignal, ist jedoch um 180 Grad zu diesem phasenverschoben.
CLK270	O	Taktsignal	Hat die gleiche Frequenz, wie das CLK0 Taktsignal, ist jedoch um 270 Grad zu diesem phasenverschoben.
CLK2X	O	Taktsignal	Zwiefache Frequenz von CLK0.
CLK2X180	O	Taktsignal	Zwiefache Frequenz von CLK0 und um 180 Grad phasenverschoben.
CLKDV	O	Taktsignal	Hat die (1/n) fache Frequenz vom CLK0 (n=CLKDV_DIVIDE).
CLKFX	O	Taktsignal	Dient dem PPC405 als Taktquelle. Die Frequenz dieses Taktausgangs wird vom Multiplizierer und dem Dividierer beeinflusst. Sie ist meistens höher als die am CLK0.

Signal	I/O Typ	Art	Beschreibung
CLKFX180	O	Taktsignal	Gleich dem CLKFX Takt, nur um 180 Grad phasenverschoben.
LOCKED	O	Kontrollsignal	Alle ermöglichten DCM Features sind blockiert.
STATUS[7:0]	O	Kontrollsignal	Beschreibung der einzelnen Signale findet sich unterhalb.
STATUS[0]	O	Kontrollsignal	Wenn Signal = 1, dann Phase Shift Überlauf
STATUS[1]	O	Kontrollsignal	Wenn Signal = 1, dann CLKIN gestoppet.
STATUS[2]	O	Kontrollsignal	Wenn Signal = 1, dann CLKFX gestoppet.
STATUS[7:3]	O	Kontrollsignal	N/A
PSDONE	O	Kontrollsignal	Steht für ”phase shift done” und ist mit der steigenden Flanke von PSCLK synchron. Wenn PSDONE = 0, dann ist keine Phasenverschiebung aktiviert oder die Phasenverschiebung wird gerade ausgeführt. Wenn PSDONE = 1, dann ist die angefragte Phasenverschiebung abgeschlossen (’high’ nur für einen PSCLK Taktzyklus).

Tabelle 3.7: *DCM I/O Ports*

3.5.2 Clock and Power Management (CPM) Interface des PPC405 Prozessors

Der PowerPC 405 stellt dem Benutzer Schnittstellen und Methoden zur Verfügung, mit denen der Takt gesteuert werden kann (Abbildung 3.13). Es gibt zwei Möglichkeiten, den Prozessor-Takt zu steuern. Entweder mit dem Global Local Enables oder mit dem Global Gating. Beide Methoden werden hier vorgestellt.

Global Local Enables

Der PPC405 besitzt 3 disjunkte Clock-Zonen, die mit der Global Local Enables Methode einzeln angehalten werden können.

1. Die Core Clock Zone mit der größte Logik des PowerPC wird durch das CPMC405CPUCLKEN Signal gesteuert.

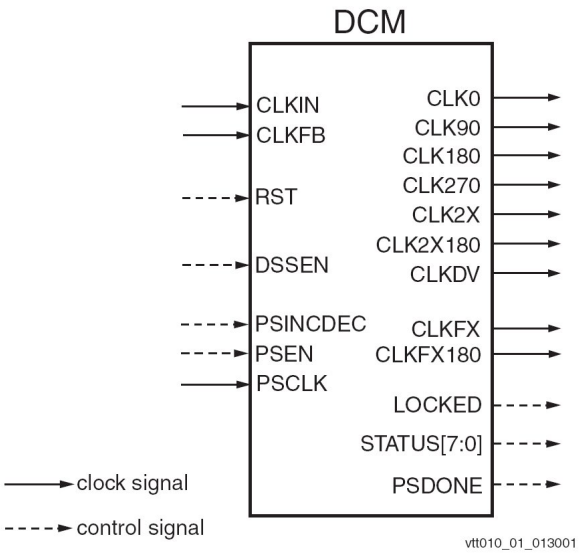


Abbildung 3.12: *DCM-Blockdiagramm*

2. Die Timer Clock Zone beinhaltet die PowerPC405 Timer Logik. Die Logik kann auf eingetretene Timer-Ereignisse reagieren und die Core Logik z.B. mit einem “wake-up” aus dem Schlaf wecken. Gesteuert wird diese Zone durch das CPMC405TIMERCLKEN Signal.

3. Die JTAG Clock Zone enhält die JTAG Logik und kann mit dem Signal CPMC405JTAGCLKEN de- bzw. aktiviert werden.

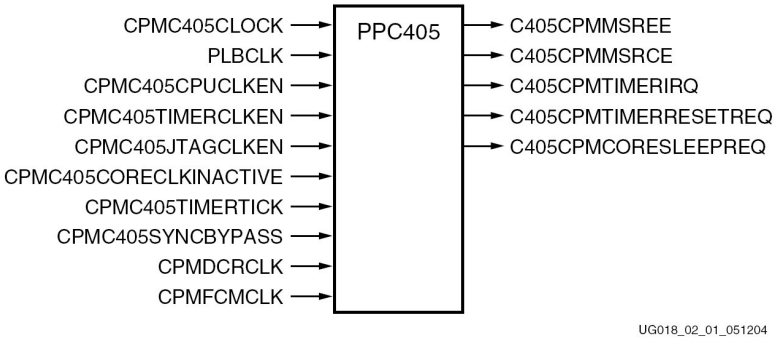


Abbildung 3.13: *CPM-Schnittstellen Block Symbol*

Global Gating

Anstelle des Global Local Enables, das das Durchdringen des Taktsignals in eine Zone verhindert, wird beim Global Gating das CPMC405CLOCK Taktsignal von toggeln, also dem hin und her Springen von 0 auf 1, abgehalten. Bei Verwendung des Global Gating ist darauf zu achten, dass nach

dem Unterbrechen des Taktsignals am Takteingang des PowerPC kein low sondern ein high Signal anliegt. Es sollte erwähnt werden, dass mit dem Unterbrechen des Taktsignals CPMC405CLOCK nur die Core Zone und die Timer Zone angehalten werden, nicht aber die JTAG Zone. Diese besitzt ihren eigenen Takt, der nicht abgegriffen werden kann. (Ein Deaktivieren der Zone ist nur mit der Global Local Enables Methode möglich).

Tritt eines der unten aufgeführten Ereignisse auf, sollte der PPC aus seinem “Schlaf-Modus” geweckt werden. Diese sind zwar für die Lösung des vorgestellten Problems irrelevant, da die hier entwickelte Memory-Management-Unit den Prozessor aktivieren wird, soll hier aber der Vollständigkeit halber erwähnt werden.

- Ein Timer Interrupt oder Timer Reset wurde vom PPC405 ausgelöst.
- Ein Chip- oder System-Reset wurde ausgelöst (aus einer anderen Quelle als dem PPC405).
- Ein externer Interrupt oder kritischer Interrupt-Eingang ist gesetzt und der entsprechende Interrupt wurde durch das zugehörige Machine-State Register (MSR) bit aktiviert.
- Das DBGC405DEBUGHALT Signal ist auf high gesetzt. Dies signalisiert, dass ein externes Debug-Tool die Kontrolle über den PPC405 Prozessor übernommen hat.

In der Tabelle 3.8 sind alle wichtigen CPM Signale des PPC405 aufgelistet und kurz erläutert.

3.5.3 Global Clock Buffer (BUFG)

Im FPGA realisierte Komponenten, die getaktet werden, werden durch Leitungspfade mit dem Takt versorgt. Aufgrund der unterschiedlichen Lokalität der Komponenten im FPGA, haben die einzelnen Leitungspfade unterschiedliche Längen und die Takte somit unterschiedliche Laufzeiten. Diese unterschiedlichen Laufzeiten führen dazu, dass die Komponenten nicht mehr synchron getaktet sind und ein Signalaustausch nicht mehr möglich ist. Um diese Laufzeitdifferenzen zu kompensieren, werden so genannte Global Clock Buffer (BUFG) eingesetzt. Der Global Clock Buffer ist ein im FPGA fest integrierter Baustein mit einem Eingangs- und Ausgangsport. Ein FPGA besitzt nicht nur einen sondern mehrere solcher BUFGs. Jedem Takteingang einer Komponente wird beim Synthetisieren automatisch ein BUFG zugewiesen. Dieser puffert das Taktsignal und gibt es mit den anderen BUFGs synchron an die Komponente weiter. Der DCM, der eine Vielzahl von Takten verwaltet, besitzt auch entsprechend viele BUFGs. Abbildung 3.14 zeigt, wie für ein Taktsignal CLK* der BUFG eingesetzt wird. Es besteht auch die Möglichkeit, den Clock Buffer selber zu instantiieren bzw. konfigurieren, was aber nur dann gemacht werden sollte, wenn das Design einen speziellen architekturspezifischen Clock-Buffer erfordert.

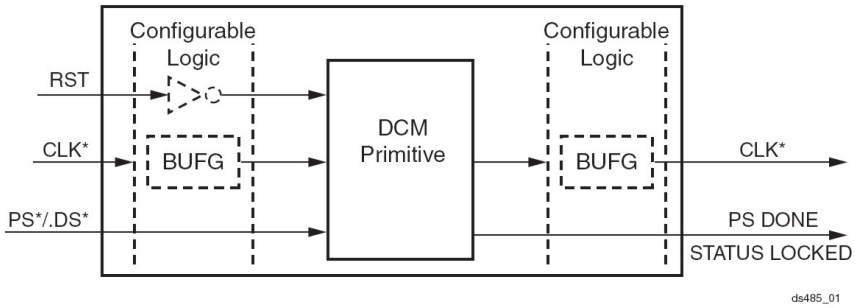


Abbildung 3.14: *DCM/BUFG-Blockdiagramm*

3.5.4 Processor Local Bus (PLB)

Der Prozessor Local Bus (PLB, siehe auch Abbildung 3.15) ist ein Standard Highspeed-Prozessor-Bus zur Integration externer Ressourcen. Der PLB stellt einen 32-bit Address-Bus und 64-bit Daten-Busse für Instruktionen und Daten zur Verfügung. Zwei 64-bit Busse sind mit der Data-Cache Einheit verbunden. Einer davon ist ein reiner Lese-Bus, der andere ein reiner Schreib-Bus. Ein dritter 64-bit Bus ist mit der Instruction-Cache Einheit verbunden. Dieser Bus ist für die Übertragung der Instructions zuständig.

Der PLB ist das primäre E/A-Bus-Protokoll und unterstützt allgemein Peripheriegeräte mit einer höheren Geschwindigkeit. Außerdem gibt es den OPB als sekundären Bus für langsamere und weniger komplexe Peripherie-Geräte. Der Prozessor kommuniziert mit dem OPB über die PLB-OPB- bzw. OPB-PLB-bridge. Desweiteren existiert der Device Controll Register-Bus (DCR), ein zusätzlicher 10-bit Bus, der genutzt werden kann, um den PLB um langsame Operationen zu entlasten. Er wird hauptsächlich für den Zugriff auf Status-/Kontrollregister der Master- und Slave-Geräte verwendet. Dieser Bus ist nicht Teil des systemweiten Adressraums. Der DCR Bus erlaubt direkten Datentransfer zwischen OPB-Geräten, unabhängig von Transfers auf dem PLB.

Master-Slave-Anbindung

Der PLB ermöglicht Lese- und Schreib-Operationen zwischen (maximal 16) Master- und (einer beliebigen Anzahl an) Slave-Geräten, die über eine PLB-Bus-Schnittstelle verfügen [11]. Jeder Master besitzt eine eigene Adress-, Lese- und Schreib-Anbindung zum PLB und mehrere transfer qualifier Signale. Slaves greifen über geteilte, aber gegenseitig entkoppelte Adress-, Lese- und Schreib-Busse auf den PLB zu und besitzen ebenfalls mehrere transfer qualifier und Status Signale. Der PLB ist ein synchroner Bus. Die Taktung erfolgt über ein zentrales Takt-Signal, an welches alle Master- und Slave-Geräte angeschlossen sind.

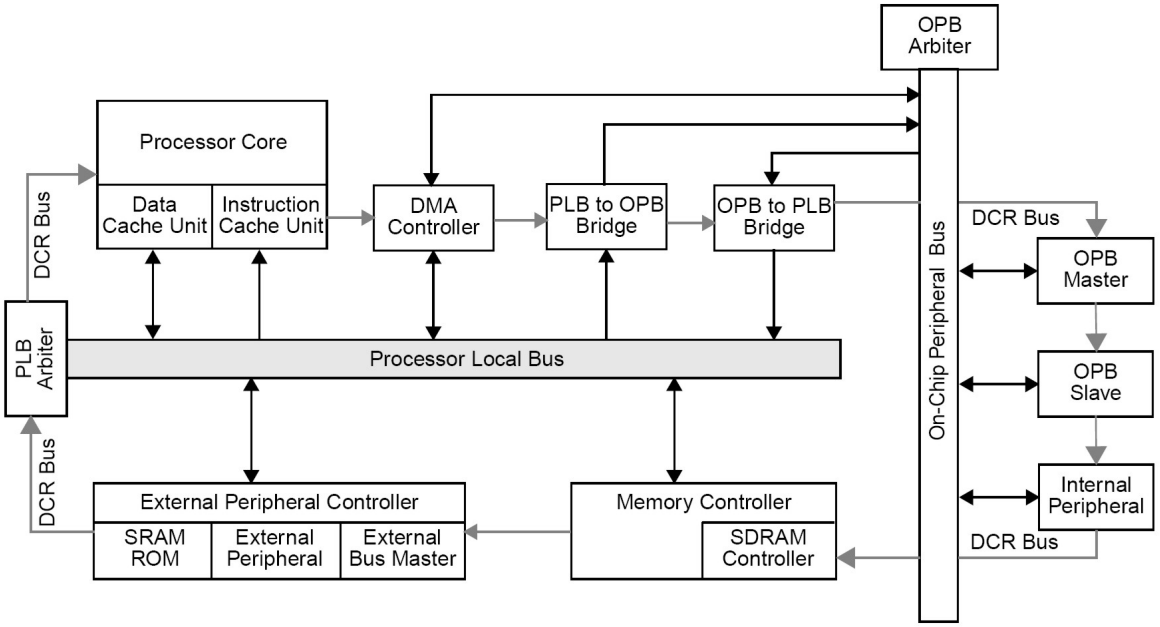


Abbildung 3.15: *Einordnung des PLB*

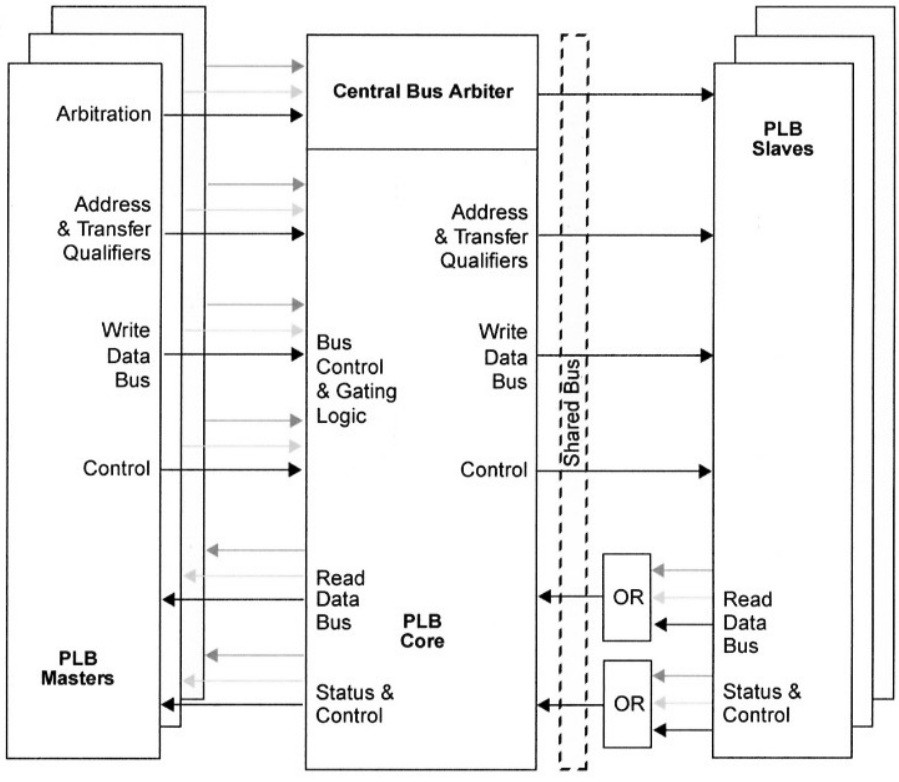


Abbildung 3.16: *Anbindung von Master- und Slave Geräten an den PLB*

Arbitrierung

Ein zentraler Arbitrations-Mechanismus regelt die Anfragen der Master für den PLB. Dieser Mechanismus ist flexibel und erlaubt unterschiedliche Prioritäts-Schemata. Außerdem gibt es einen

arbitration-locking Mechanismus, um von Master-Geräten notwendige atomare Operationen durchführen zu können.

Das Besondere ist, dass mittels eigener FPGA Logik (IP) auch ein eigener Bus Arbiter bereit gestellt werden kann. Das könnte bezüglich der PG-Aufgabe (Cache-Anbindung und deren Prioritätsvergabe) interessant sein.
Die PLB bus control unit (BCU) besteht aus einer bus arbitration Kontrolleinheit [12], welche den Adress- und Datenfluss über den PLB und DCR regelt. Die Bus arbitration control unit unterstützt 16 PLB Master. Die Adress- und Datenfluss Kontrolllogik arbeitet mit Adress Pipelining und bietet 16 Mastern und 16 Slaves Adress- und Daten-Steering Unterstützung.

PLB arbitration Operationen:

- Single Read Transfer Bus Time-Out
- Single Write Transfer Bus Time-Out
- Line Read Transfer Bus Time-Out
- Line Write Transfer Bus Time-Out
- Burst Read Transfer Bus Time-Out
- Burst Write Transfer Bus Time-Out
- Read Transfer
- Pipelined Read Transfer
- Pipelined Write Transfer

Detailierte Informationen zur Arbitrierung befinden sich in der PLB Dokumentation von Xilinx [12].

Timing

Der Trend bei eingebetteten Systemen ist die Integration aller Hauptfunktionen in einem einzigen Baustein. Die Haupt-Sub-Systeme von eingebetteten Entwicklungen sind:

- Prozessor (PowerPC 405 Processor)
- “schneller” Perepherie-Bus (Processor Local Bus)
- “langsamer” Perepherie-Bus (Onchip Peripheral Bus)
- On-Chip Memory und On-Chip Block RAM (OCM und BRAM)

Um die korrekte Funktionsweise der kompletten, unterschiedlich schnell getakteten, Hardware zu gewährleisten, müssen gewisse Timing-Bedingungen eingehalten werden. So ist es z.B. nicht möglich, den Prozessor mit 300 MHz und den PLB mit einer beliebigen anderen Frequenz zu takten. Vielmehr muss ein gewisses Verhältnis eingehalten werden. Der PLB-Takt muss phasengleich mit dem Prozessor-Takt ausgerichtet sein. Kommunikation zwischen Prozessor und PLB findet immer mit der steigenden Taktflanke des Prozessor-Takts statt. Die unterschiedlichen Takt-Domänen müssen durch DCM- und BUFG-Komponenten synchronisiert werden [13].

Die kritischen Taktsignale, welche mit den Prozessorblock-Taktsignalen gekoppelt betrachtet werden müssen, sind folgende:

- Main Processor Block Clock (CPMC405CLOCK)
- Primary I/O Bus Clock (PLBCLK)
- Reference Clock for the Instruction-Side OCM Controller (BRAMISOCMCLK)
- Reference Clock for the Data-Side OCM Controller (BRAMDSOCMCLK)

Power-On Reset

Damit sichergestellt ist, dass der PLB beim Einschalten zurückgesetzt wird (ohne externes Signal), existiert ein power-on reset Schaltkreis. Beim Einschaltvorgang wird für 18 Taktzyklen ein active-high PLB reset ausgelöst und anschließend negiert. Wird ein externes Reset-Signal bereitgestellt, wird es mit dem PLB-Takt durch einen 2-stufigen FlipFlop-Schaltkreis synchronisiert, und anschließend der PLB-Reset ausgelöst. Deshalb hat ein externes PLB-Reset-Signal 1-2 Takte Verzögerung. Auf jeden Fall muss ein Reset synchron zum PLB-Takt erfolgen.

Wenn ein externes Reset-Signal vor dem Ablauf der 18 Zyklen bei einem power-on Reset negiert wird, hat dies keine Auswirkungen. Der interne Reset “geht vor”. Nach dem Einschalten wird der PLB also erst nach frühestens 18 Zyklen zurückgesetzt.

Übersicht über das PLB-Transfer-Protokoll

Bei einer PLB Transaktion wird zwischen Adress- und Daten-Zyklus unterschieden [11]. Der Adress-Zyklus hat 3 Phasen:

1. Anfrage
2. Transfer

3. Bestätigung

Eine PLB-Transaktion im Adresszyklus beginnt, wenn ein Master seine Adresse und transfer qualifier Signale anlegt und eine Anfrage auf Bus Belegung stellt. Wenn der Arbiter den Bus freigegeben hat, legt der Master Adresse und transfer qualifiers für die Slaves auf den Bus. Im Normalfall wird der Adresszyklus durch die Bestätigung eines Slaves von Masteradresse und transfer-qualifiers beendet.

Jeder Daten-Puls im Datenzyklus hat 2 Phasen:

1. Transfer
2. Bestätigung

Während der Transferphase legt der Master seine Daten auf den Schreib-Datenbus oder liest Daten vom Lese-Datenbus. Für jeden Datenpuls folgen in der Bestätigungsphase Bestätigungssignale.

Bei einer Ein-Puls-Übertragung bestätigen die Bestätigungssignale gleichzeitig das Ende des Datentransfers. Bei Line-Burst-Transfers wird das Ende des Datenzyklus erst nach dem letzten Puls signalisiert.

Durch die Unterscheidung zwischen Schreib- und Lesebus können auf dem PLB überlappende Transaktionen stattfinden. Dies macht den Bus sehr leistungsfähig, das Busprotokoll aber dafür recht komplex.

PLB-Signale unterliegen folgender Beschriftungskonvention:

- PLB-Output-, Slave-Input-Signale beginnen mit *PLB_*
- Slave-Output, PLB-Input-Signale beginnen mit *Sl_*
- PLB-Output, Master-Input-Signale beginnen mit *PLB_Mn_*, wobei “n” Platzhalter für die Master-Nummer ist
- Master-Output, PLB-Input-Signale beginnen mit *Mn_*, wobei “n” Platzhalter für die Master-Nummer ist

PLB-Signale können in folgende Gruppen eingeteilt werden:

- PLB System Signale
- PLB Arbitration Signale
- PLB Status Signale
- PLB Transfer Qualifier Signale

- PLB Schreib-Datenbus Signale
- PLB Lese-Datenbus Signale
- Zusätzliche Slave Output Signale

Eine detaillierte Beschreibung der einzelnen Signale befindet sich in der Dokumentation zum PLB [11].

3.5.5 Entscheidung / Vorgehen

Es gibt insgesamt drei Möglichkeiten, den Prozessor anzuhalten.

- Das Global Local Enables
- Das Global Gating
- Das JTAGHALT Signal

Die Entscheidung fiel auf das Global Gating, da es im Gegensatz zum Global Local Enables feingranularer ist. Beim Global Local Enables ist nicht bekannt, wie genau eine einzelne Zone reagiert, wenn sie durch das Enable-Signal deaktiviert wird. Das gleiche gilt für das JTAGHALT Signal, bei dem nur bekannt ist, dass nach jedem Halt die Instruktionen-Pipeline des Prozessors geleert wird. Der PPC würde in diesem Fall beim Aktivieren nicht mehr den gleichen Zustand wie im Moment des Anhaltens haben.

Global Gating (Vorgehen)

Beim ersten Vorgehen wurde ein VHDL Modul entwickelt, welches zwischen den DCM und dem PPC geschaltet wurde. Das Taktsignal vom DCM zum PPC wurde getrennt und in das VHDL Modul geleitet, dort von der Logik verarbeitet und an den Ausgang geleitet, der wiederum mit dem Takteingang des PPC verbunden war. Ein Tastknopf diente dabei als Schalter. Bei Betätigung wurde nicht das Taktsignal sondern eine konstante ' logische 1' an den Takteingang des PPC gelegt.

Ein simples C-Programm, welches permanent eine Zeichenfolge auf einem Terminal über die RS232 Schnittstelle ausgab, sollte Auskunft darüber geben, ob es gelungen war den PPC anzuhalten. Wurde die Zeichenfolge auf dem Terminal nicht mehr aktualisiert, war der Prozessor angehalten.

Nachdem das VHDL Modul ins Hardwaresystem eingebunden und das Board gestartet worden war, lieferte das Terminal keine Ausgabe. Die Ursache für das Verhalten waren die Schaltverzögerungen des, zwischen den DCM und PPC geschalteten Moduls, die zu Laufzeitdifferenzen zwischen dem Prozessortakt und dem Systemtakt führten. Das führte dazu, dass die Taktflanken des PLB nicht

synchron zu denen des PPC waren. Die Synchronität der beiden Takte ist aber Voraussetzung für den Datenaustausch zwischen Prozessor und PLB.

Etwas zwischen den DCM und PPC zu schalten ist wegen der Schaltzeit also nicht möglich. Somit stand das Global Local Enables jetzt im Vordergrund.

Global Local Enables (Vorgehen)

Das VHDL Modul zum Steuern des PPC nach der Global Local Enables Methode hatte jetzt keine Taktein- und ausgänge, sondern lediglich einen Kontrollausgang. Dieser war mit einer logischen 1 initialisiert und an die Ports CPMC405CPUCLKEN und CPMC405TIMERCKLEN angeschlossen. Wie aus der Tabelle 3.8 ersichtlich, muss an den beiden vorgestellten Ports ein high Signal anliegen, damit diese nicht gleich beim Einschalten des Boards den PowerPC deaktivieren. Das Modul reagierte wie das aus dem ersten Versuch auf einen Tastendruck. Das Betätigen der Drucktaste führte dazu, dass eine logische 0 dem Kontrollausgang zugewiesen wurde. Damit lagen an den beiden Ports CPMC405CPUCLKEN und CPMC405TIMERCKLEN eine logische 0 an, die wie schon oben erwähnt, das Durchdringen des Taktsignals in die Core- und Timer-Zone verhindert. Damit konnte der PPC angehalten und wieder zum Laufen gebracht werden. Leider klappte das Anlaufen nicht immer. Nach einigen Start-/Stopp-Vorgängen ist das System eingefroren und konnte nur durch einen Soft-Reset wieder in Gang gesetzt werden. Die Vermutung war, dass die Zonen nicht an den steigenden Flanken des PPC und PLB deaktiviert wurden, sondern genau an der Stelle, an der das Kontrollsignal eingetreten ist. Die Abbildung 3.17 links oben veranschaulicht das. Nähere Informationen diesbezüglich waren in den Xilinx bzw. IBM Dokumentationen [14] nicht zu finden. Bekannt ist, dass eine “On The Fly” Änderung der PPC-Frequenz nur dann erfolgen kann, wenn die positiv steigende Taktflanke vom PLB und die vom PPC gleichzeitig eintreten (Abbildung 3.17 unten).

Das Hauptaugenmerk war jetzt darauf gerichtet eine Methode zu finden, die beim Auftreten des Kontrollsignals den Takt erst bei der nächsten positiven Taktflanke deaktiviert (Abbildung 3.17 rechts).

Global Gating mit BUFG (Vorgehen)

Bei genauer Betrachtung der verschiedenen BUFGs fallen die speziellen Eigenschaften des BUFGCE_1 auf. Dieser BUFG besitzt einen zusätzlichen Eingang, den Clock Enable (CE), welcher den Taktausgang des BUFG steuert. Nur bei einer logischen 1 auf dem CE-Eingang wird der Takt durchgeschleust. Beim Anliegen einer logischen 0 am CE Eingang wird der Taktausgang des BUFG konstant auf 1 gehalten. Das interessante dabei ist, dass der BUFG die Takte erst bei einer positiven steigenden Taktflanke de- bzw. aktiviert. Also ist mit dem BUFGCE_1 die Komponente gefunden,

Signal	I/O Typ	Wenn unbenutzt	Funktion
CPMC405CLOCK	I	benötigt	Takteingang des PPC405 (einschließlich Timer, nicht für die JTAG-Logik).
PLBCLK	I	benötigt	PLB-Takt Schnittstelle. Das Taktsignal mit dem der PLB getaktet ist.
CPMC405CPUCLKEN	I	1	De- bzw. aktiviert die Core Clock Zone.
CPMC405TIMERCKLEN	I	1	De- bzw. aktiviert die Timer Clock Zone.
CPMC405JTAGCLKEN	I	1	De- bzw. aktiviert die JTAG Clock Zone.
CPMC405CORECLKINACTIVE	I	0	Zeigt dem Core an, ob eine CPM Logik die Takte deaktiviert hat.
CPMC405TIMERTICK	I	1	Inkrementiert oder dekrementiert die PPC405 Timer mit dem CPMC405CLOCK.
C405CPMMSREE	O	nicht angeschlossen	Gibt den Wert von MSR[EE] an.
C405CPMMSRCE	O	nicht angeschlossen	Gibt den Wert von MSR[CE] an.
C405CPMTIMERIRQ	O	nicht angeschlossen	Weist den PPC darauf hin, dass eine Timer-Interrupt Anfrage aufgetreten ist.
C405CPMTIMERRESETREQ	O	nicht angeschlossen	Weist den PPC darauf hin, dass eine Watchdog-Interrupt Anfrage aufgetreten ist.
C405CPMCORESLEPPREQ	O	nicht angeschlossen	Signalisiert eine Anfrage des Cores in den Schlaf-Modus versetzt zu werden.

Tabelle 3.8: *CPM Schnittstelle I/O Signale*

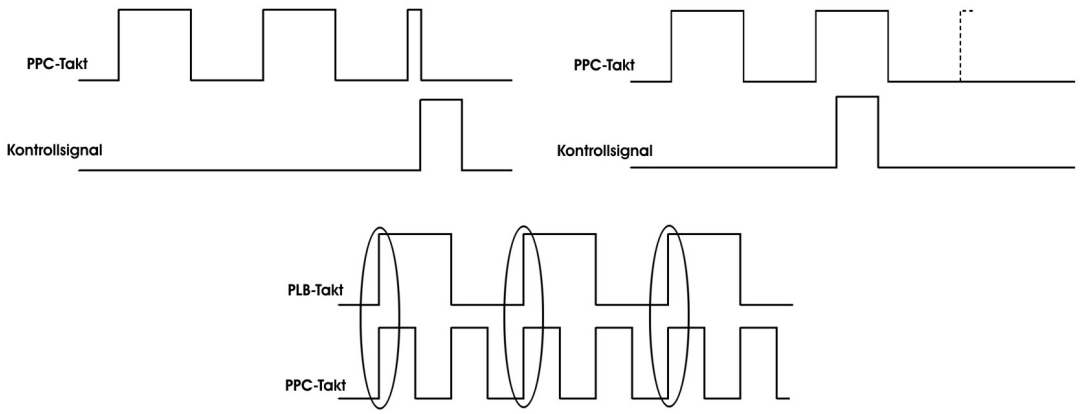


Abbildung 3.17: *Takt- und Kontrollsignale*

die alle obigen Voraussetzungen erfüllt. Um den BUFGCE_1 zu verwenden, wurde der zu steuernde Taktausgang CLKFX im DCM mit dem BUFGCE_1 instantiiert und ein neuer Port CONTROL für die Steuerung des CE-Eingangs deklariert. An dem CONTROL Port wurde ein neues VHDL-Modul COUNTER angeschlossen. Dieses Modul hatte zur Aufgabe, bei einem Tastendruck eine logische 0 an den CONTROL-Port zu schicken, diese eine gewisse Zeit (5s@100MHz) zu halten und danach den CONTROL-Port wieder mit einer logischen 1 zu belegen. In dieser Wartezeit lag somit auf dem CE-Eingang des BUFG ein low Signal an, was zur Folge hatte, dass am Taktausgang CLKFX kein Takt sondern, wie oben schon erwähnt, eine konstante logische 1 an lag. Der Taster war in soweit entprellt, dass nach dem ersten Tastendruck bis zum Ende der Wartezeit ein zweites Betätigen der Taste keinen Einfluss auf das System hatte. Doch leider hatte auch dieses System ein nicht deterministisches Verhalten gezeigt.

Das Hardwaresystem auf dem gearbeitet wurde, umfasste den PPC405, einen PLB, BRAM, RS232 u.a. Komponenten. Versuche auf einen Hardwaresystem mit dem OCM-BUS und PPC-BRAM anstatt des PLB und BRAM zeigten ein anderes, ein deterministisches Verhalten. Das zeigte, dass der PLB bei der Taktunterbrechung eine wichtige Rolle spielen musste.

Global Gating mit BUFG und PLB_Spy (Vorgehen)

Die Idee war jetzt, den PPC abhängig vom Zustand des PLBs zu stoppen. Wenn kein Schreib- bzw. Lesevorgang auf dem PLB stattfand, der PLB sich also im Idle Zustand befand, sollte der PPC angehalten werden. Der Zustand des PLB findet sich in der Arbiter State Machine, die ein Teil des PLB ist. Da die Arbiter State Machine nicht über den PLB angesprochen werden kann, wurde ein VHDL-Modul PLB_Spy geschrieben, das eingebunden im PLB, die Zustandsinformation nach außen

leitet. Abhängig von diesem Zustand wurde der Takt im COUNTER Modul des DCM unterbrochen. Der PPC konnte also nur angehalten werden, wenn sich der PLB im IDLE Zustand befand und der Taster betätigt wurde.

Mit dieser Vorgehensweise konnten erste Erfolge verzeichnet werden. Die Ausgabe auf dem Terminal konnte angehalten und wieder fortgesetzt werden. Zusätzlich zu der Terminalausgabe wurden noch die auf dem Board befindlichen LEDs mit dem PPC über den PLB angesteuert. Auch hier konnte der PPC ohne Probleme de- bzw. aktiviert werden.

Global Gating mit BUFG, PLB_Spy und Framebuffer (Vorgehen)

Das es sich um ein einfaches System, bestehend aus einem PPC, PLB, RS232 Schnittstelle und BRAM gehandelt hat, sollte dies jetzt auf einem etwas komplexeren System, welches zusätzlich einen Framebuffer umfasste, getestet werden. Auf diesem System, entwickelt von der VGA-Gruppe (siehe 5.3), lief ein C-Programm C, welches den Bildschirm abwechselnd mit den Farben Rot, Grün, Blau füllte. Beim Deaktivieren des Taktes sollte sich das System so verhalten, dass die Bildschirmausgabe an der momentanen Ausgabestelle für (5s@100MHz) stehen bleibt und nach dieser Zeit von dort wieder mit der Ausgabe anfängt. Doch nachdem der Entwurf in das Hardwaresystem implementiert und das Board gestartet wurde, zeigte der Bildschirm ein verrauschtes Bild (ohne das der PPC angehalten wurde). Nach langer Versuchsreihe stellte sich heraus, dass die Zählerlogik COUNTER des Entwurfs auf unerklärliche Weise, da diese bei der Initialisierung keinen Einfluss auf das System hatte, dieses Rauschen verursacht hatte. Diese Zählerlogik wurde entfernt und das Reaktivieren des Taktsignals einfach durch einen zweiten Taster gelöst. Durch die Zwei-Tasten-Lösung war auch ein Entprellen, mit genügend Wartezeit (ca. 2s) zwischen zwei Tastendrücken, der Taster garantiert. Mit dieser Methode wurde das System einwandfrei gestartet und die Bildschirmausgabe auch richtig ausgeführt. Ein deterministisches Verhalten beim Deaktivieren des Taktes war jedoch auch hier nicht festzustellen. Das System konnte angehalten aber nicht nach jedem Anhalten wieder reaktiviert werden.

Lösen von Lizenzproblemen

Zu Beginn des 2. PG-Semesters traten Probleme auf, die die Weiterarbeit an der Untersuchung der Taktkontrolle zunächst verzögerten. Im Testprojekt der VGA-Gruppe wurden IP-Cores benutzt, deren Lizenzen mittlerweile ausgelaufen waren. Für weitere Untersuchungen der Taktkontrolle war es aber notwendig, dass ein lauffähiges VGA-Projekt zur Verfügung steht um unsere Überlegungen testen zu können. Der erste Schritt bestand also darin zunächst wieder ein lauffähiges Projekt zu bauen. Versuche die Software neu zu installieren oder das Projekt von Grund auf neu zu erstellen führten nicht zu dem erhofften Erfolg. Als nächstes wurde versucht die kostenpflichtigen IP-Cores

durch adäquate freie IP-Cores zu ersetzen und nicht zwingend benötigte IP-Cores zu entfernen. Dies machte allerdings einige Anpassungsarbeiten notwendig, da die freien IP-Cores eine andere Struktur aufwiesen und bei Löschung von IP-Cores Fehlermeldungen aufgrund unverbundener Ports entstehen. Die Alternative Open Cores zu benutzen wurde verworfen, da diese über einen eigenen Bus (Wishbone) angebunden werden und somit zunächst noch eine Bridge hätte implementiert werden müssen.

Trotz des Entkernen/Neuerstellen des Framebuffer-Projektes mit Hilfe des Wikis gelang es zunächst nicht eine lauffähige Version zu bauen. Die Synthese gelang zwar, aber das Testprogramm hatte nicht den gewünschten Effekt (Bildschirm-Farbwechsel) zur Folge. Der Monitor blieb schwarz bzw. zeigte ein stehendes, verrauschtes Bild mit scheinbar zufälligen Farbpixeln. Auch mit Hilfe aus der VGA-Gruppe konnte das Problem zunächst nicht behoben werden.

Als nächstes wurde ein neues Projekt erstellt, welches ausschließlich die minimal benötigte Hardware enthielt. Unter anderem wurde dabei auch auf den DCR-Bus verzichtet und eine falsch berechnete Base-Adresse als Urheber der Probleme vermutet. Diese Vermutung führte dann letztendlich wieder zu einem lauffähigen Test-Projekt, so dass mit der eigentlichen Arbeit fortgefahren werden konnte.

Taktunterbrechung mit dem Sl_wait Signal

In der vergangenen Zeit wurde versucht den Takt zu kontrollieren wenn auf dem PLB keine Daten anlagen. Dafür wurde das PLB-Idle Signal der PLB-State-Machine verwendet. Aufgrund von gruppeninternen Diskussionen und Literaturrecherche wurde klar, dass die ausschließliche Beobachtung des PLB-Idle Signals nicht ausreichend ist. Es kommt auf das Zusammenspiel mehrerer am PLB angeschlossener Komponenten an. Auch die Signale aus dem DDR-Kontroller müssen bei der Taktkontrolle betrachtet werden. Bei den ersten Versuchen wurde das Slave-Wait-Signal (Sl_wait) anstelle des PLB-Idle Signals benutzt, um den PPC anzuhalten. Dieses Signal wird vom DDR-Kontroller gesetzt und führt dazu, dass der Arbiter keinen PLB-Reset durchführen kann. Es signalisiert dem Arbiter, dass ein Slave, in diesem Fall der DDR-Kontroller, die Beantwortung einer Master-Anfrage, hier PPC, nicht sofort durchführen kann. Für die Dauer des positiv angelegten Sl_wait-Signals wird der Zähler des Arbiters für das Zurücksetzten des PLB nicht erhöht. Ergebnis: Die Taktkontrolle funktionierte schon viel besser als vorher, aber leider immer noch nicht zu 100% deterministisch. Es hat sich herausgestellt, dass beim Sl_wait-Signal noch andere, überlappende, Speicherzugriffe ausgeführt, und beim Anhalten mittendrin unterbrochen wurden

Taktunterbrechung mit den Sl_rdComp und Sl_wrComp Signalen

Damit der Prozessor nicht mitten in einer Speicheranfrage unterbrochen werden konnte, musste die Zeitspanne in der der Prozessor angehalten werden sollte, so kurz wie möglich sein. Die Slave-Read-Complete (Sl_rdComp) und Slave-Write-Complete (Sl_wrComp) Signale waren dazu sehr gut geeignet, da sie nach einem Speicherzugriff für nur einen Takt lang angelegt waren. Das Sl_rdComp-Signal stellte jedoch ein kleines Hindernis dar: es wurde einen Takt vor dem Ende der tatsächlichen Leseoperation angelegt. Würde der Prozessor bei diesem Signal angehalten werden, würde die Speicherlese-Operation nicht zu Ende geführt werden und ein Absturz des Systems wäre die Folge. Um nicht extra ein Verzögerungsmodul für die Betrachtung dieses Signals zu programmieren, wurde nur das SL_wrComp Signal für die Taktunterbrechung herangezogen. Außerdem wurde mit dem Sl_rdDAck Signal darauf geachtet, dass keine Leseoperation parallel zur Schreiboperation stattfand. Die Bildschirm- und Terminalausgabe konnten mit diesen Signalen zum ersten mal Kontrolliert angehalten werden.

PowerPC Kontrolle unter Linux

Da die Taktunterbrechung auch Einfluss auf das verwendete Linux-Betriebssystem hat, wurde im nächsten Schritt eben dieser Einfluss untersucht. Dazu wurde das System von der Compact-Flash Karte gebootet. Bei den ersten Tests zeigte sich kein deterministisches Verhalten. In einigen Fällen kam es zum Kernel-Fehler oder gar Komplett-Absturz des Systems. Es wurde vermutet, dass dies in den Fällen passiert, in denen ein Interrupt behandelt wird. Im Interrupt-Controller gibt es ein Signal, welches dem PPC die Interrupt-Behandlung signalisiert. Als nächstes wurde mit Hilfe dieses Signals die Taktunterbrechung nicht erlaubt, wenn das Signal auf 1 lag. Es zeigte sich ein verbessertes, aber immer noch nichtdeterministisches Verhalten. Vermutlich darf der Takt auch während der Bearbeitung von Software-Interrupts nicht unterbrochen werden. Dafür existiert allerdings kein Hardware-Signal.

Hilfe von Aussen

Aufgrund des Zeitdrucks wurde es sinnvoll parallel zu den laufenden Untersuchungen außerhalb Hilfe zu suchen. Dementsprechend wurde im Internet nach adäquaten Foren gesucht, die sich mit dem Thema ”Xilinx FPGA”, insbesondere ”XUP”, beschäftigen. Dazu wurde Accounts bei folgenden Foren angelegt: http://www.embeddedrelated.com/ http://www.power.org/ http://www.fpga4fun.com/ http://www.progforum.com/ http://www.mikrocontroller.net/ Außerdem wurde über den Lehrstuhl der Xilinx-Support kontaktiert.

Leider wurde auf keine Anfrage geantwortet, so dass eine Lösung nur durch weitere Test und

Literatur-Recherche gefunden werden konnte.

Überlegung zu einer approximativen Lösung

Aufgrund der knappen Zeit wurde jetzt über eine approximative Lösung nachgedacht. Die Approximation soll hier den Fehler korrigieren, der entsteht, wenn der PPC nicht angehalten wird.

Um den Fehler zu bestimmen, muss erst die Zeit festgehalten werden, in der der PPC angehalten werden sollte. Dies geschieht in unserem Entwurf mit Hilfe eines Counters, der mit der Aktivierung der Simulation jeden Takt inkrementiert wird. Am Ende der Simulation steht in einem Register die Anzahlt der Takte, in denen der PPC angehalten werden sollte. Näheres dazu, siehe 3.1.1.3.

Das Verhalten des Prozessors und seiner Komponenten wird durch Befehle, die er verarbeitet beeinflusst. Diese Befehle müssen bei der Approximation mit berücksichtigt werden. Für das Bestimmen der Befehle zur Laufzeit ist in der Kürze der Zeit keine Lösung gefunden worden. Es wird stattdessen eine Statistik 3.18 herangezogen, die Wahrscheinlichkeiten des Auftretens eines Befehls in den meisten Anwendungen angibt.

Die unten aufgeführte Tabelle zeigt die Ausführungslatenzen gängiger Instruktionen mit eingeschalteten Caches.

Table 1. Instruction mix for different benchmarks

4xx Processor	Operation type	EEMBC benchmarks									
		Convolutional encoder data1		Convolutional encoder data2		Convolutional encoder data3		Packet flow (512K)		Packet flow (2M)	
		Instr cnt	% of total	Instr cnt	% of total	Instr cnt	% of total	Instr cnt	% of total	Instr cnt	% of total
	Add / subtract	2059	6%	2059	6%	2057	8%	7859	19%	29657	19%
	Multiply	0	0%	0	0%	0	0%	0	0%	0	0%
	Divide	0	0%	0	0%	0	0%	0	0%	0	0%
	Logic	2561	7%	2561	8%	1537	6%	1874	5%	7064	5%
	Shift / rotate	0	0%	0	0%	0	0%	4862	12%	18356	12%
	Compare	5633	15%	4609	14%	3585	14%	2621	6%	9887	6%
	Branch	5633	15%	4609	14%	3585	14%	6362	16%	24008	16%
	Load / store	20998	56%	18949	57%	13828	55%	15725	39%	59321	39%
	Other	512	1%	512	2%	512	2%	1122	3%	4236	3%
	Total	**37396**		**33299**		**25104**		**40425**		**152529**	
440GP (500MHz)	IPC	0.87		0.83		0.94		0.95		0.45	
	Iterations/ sec	11614		12507		18755		11694		1490	
405GPr (400MHz)	IPC	0.65		0.66		0.67		0.47		0.43	
	Iterations/ sec	6911		7969		10697		4644		1127	

Abbildung 3.18: *Statistiken*

Common Instructions	*Execution Latency*	
Arithmetic	1	
Traps	2	
Logical	1	
Shifts/rotates	1	
Divide	35	
Loads[b] (cache hit)	1	
Load multiples/strings (cache hit)	1/transfer	
Stores (cache hit)	1	
Store multiples/strings (caches hit)	1/transfer	
Move to/from Device Control Register (DCR)	3	
Move to/from Special Purpose Register	1	
Note: Load instructions have a 1 cycle use penalty. Data acquired during the execute pipeline stage is not available until after the completion of the write back pipeline stage. Write back takes one clock cycle. Note: DCR access instructions depend on chip-top bus clocking implementation; three cycles is the shortest latency.		

Abbildung 3.19: *Ausführungslatenzen*

3.5.6 Überblick und Auswirkungen auf den PPC bei der Taktunterbrechung

3.5.6.1 Funktionseinheiten, Programmiermodel und Schnittstellen

Der PowerPC405 Kern besteht aus einer Fünfstufigen ?single issue? Pipeline, einen 32 elementaren Satz an 32-bit General Purpose Registern (GPRs), einer 64-Eingangs TLB-Memory Management Unit, L1 Instruktionen und Data Caches von je 16 KB, Zeitmessern, einen Hardware Debug Module und verschiedenen Schnittstellen. Abbildung 3.20 zeigt das Block-Diagramm vom PPC405 Kern.

3.5.6.2 CPU Pipeline

Der PPC405 arbeitet bei Instruktionen mit einer fünfstufigen Pipeline, bestehend aus einer Abruf- (engl. fetch), Entschlüsselungs- (engl. decode), Ausführungs- (engl. execute), Zurückschreib- (engl. write back) und einer Lade-Zurückschreib-(engl. load-write back) Phase. Die unterschiedlichen Instruktionstypen benutzen die Pipeline auf unterschiedliche Arten, wie in der Abbildung 3.21 abgebildet.

Die Abruf- und Entschlüsselungs-Logik hält einen stetigen Instruktionenfluss zur Ausführungslogik aufrecht, indem mindestens zwei Instruktionen in die Abruf-Warteschlange geladen werden. Die Abruf-Warteschlange besteht aus zwei Vor-Abruf Puffern (angl. pre-fetch buffer), PFB1 und PFB2 und der Entschlüsselungsstufe DCD (engl. decode stage). Bei einer leeren Warteschlange fließen die Instruktionen direkt in die Entschlüsselungsstufe.

3.5.6.3 Instruktion und Data Caches

Der PowerPC405 Core greift auf den Speicher über die Instruktion- und Data-Cachekontroller zu. Alle Cachezugriffe werden erst an die Cache-Einheiten und anschließend an den Hauptspeicher geschickt. Cachehits werden in einem Taktzyklus abgearbeitet und Cachemisses resultieren in einem Hauptspeicherzugriff über den PLB, der mind. 14 Taktzyklen dauert. Um die CPU Performance zu maximieren sind die Caches non-blocking. Non-blocking chaches erlauben dem PPC405 die

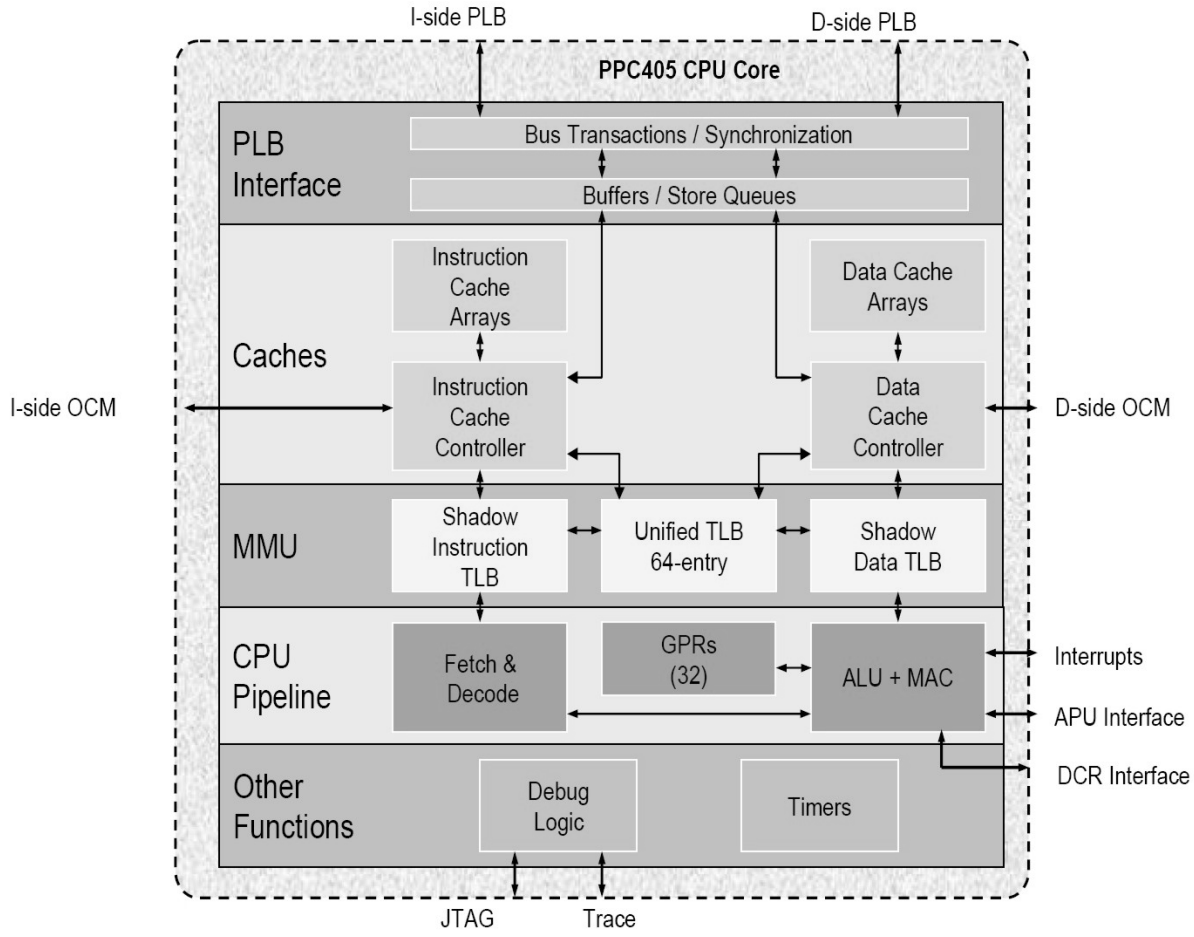


Abbildung 3.20: *PPC405 Core Block*

Pipeline Stage --- Instruction Type	Instruction Fetch Buffer	Decode / Dispatch	Execute	Write-Back	Load Write-Back
Integer / Logical	Fetch	Decode / Dispatch	Execute	Write GPR	
Branch	Fetch	Predict / Decode / Dispatch	Execute		
Store	Fetch	Decode / Dispatch	Address Gen	Cache write	
Load	Fetch	Decode / Dispatch	Address Gen	Cache read	Write GPR

Abbildung 3.21: *Pipeline*

Ausführung von Instruktionen, während eine Cachezeile neu gefüllt wird, zu überlappen. Die Caches stellen Instruktionen und Daten ohne die Pipeline zu unterbrechen bereit. Die Instruktionskontrollereinheit (ICU) (engl. insctruction cache controller) verwendet einen 32 Byte (acht Instruktionen Wort) Füllpuffer, um die Cache-Performanz zu steigern. Schreiboperationen auf gecachten Speicher passieren zuerst den Füllpuffer, bevor sie in den Cachearray aufgenommen werden. Erst wenn der Füllpuffer alle acht Wörter enthält, schreibt die ICU den Inhalt des Füllpuffers in den Cache. Dies vermeidet Cacheblockaden, wenn eine Cachezeile gefüllt wird. Es werden somit 2 statt 8 Taktzyklen benötigt, um einer Cachezeile zu füllen.

3.5.6.4 Zeitgeber (Timers)

Der PPC405 besitzt ein 64-bit Zeitbasis-Register und drei Zeitgebermodule:

1. Decrementing Counter (DEC)
2. Fixed Interval Timer (FIT)
3. Watch Dog Timer (WDT)

Alle Zeitgeber werden bei der Taktunterbrechung, durch das CPMC405TIMERCLKEN Signal mit angehalten. Damit besteht nicht die Gefahr, dass diese den Zustand des PPCs bzw. den des Systems beeinflussen.

Kapitel 4

Speichersimulation

4.1 Grundlegende Attribute der Memory-Module

Die Speichermodule Cache, Flash, SD-RAM und Scratch Pad werden als ein Speichertyp fest implementiert. In unserem Entwurf gibt es eine feste Anzahl von Speichermodulen, wobei es mehrere Instanzen eines Typs geben darf. Die Wichtigkeit der Instantiierung wird besonders bei der Realisierung der Cachehierarchie deutlich. Jedes Modul hat eine ID und einen Adressbereich, auf dem es sensitiv auf dem VPB ist. Diese Kombination ist eindeutig. Die Konfiguration hängt vom Typ ab. Wie eine Konfiguration für einen Speichertyp im Einzelnen aussieht, hängt von den Anforderungen, die zu jedem Modul weiter unten aufgeführt sind, ab. Das Gleiche gilt für die die Simulationsergebnisse enthaltenden Register. Das Cachemodul hat die größte Anzahl an verschiedenen Konfigurationen und Registern und ist somit auch, bezüglich der Realisierung, der zeitaufwändigste Speichertyp. Aus diesem Grund wurde die Entscheidung getroffen, dieses als erstes zu spezifizieren und zu implementieren. Die anderen Module wurden noch nicht detaillierter betrachtet. Die Anforderungen des ScratchPads, Flashes und des SDRAM werden im Anschluss an die Beschreibung des Caches, bzw. der bis zu diesem Zeitpunkt erarbeiteten Ergebnisse, vorgestellt.

4.2 Cache Modul

4.2.1 Allgemeine Informationen zu Caches

Einleitung

Der Geschwindigkeitsunterschied zwischen Prozessor und Speicher wird immer größer. In diesem Zusammenhang spricht man auch vom Speed-Gap. Abbildung 4.1 verdeutlicht diesen Zusammenhang [15] .

Um nun die Geschwindigkeitsdefizite auszugleichen und trotzdem die Speicherkosten gering zu

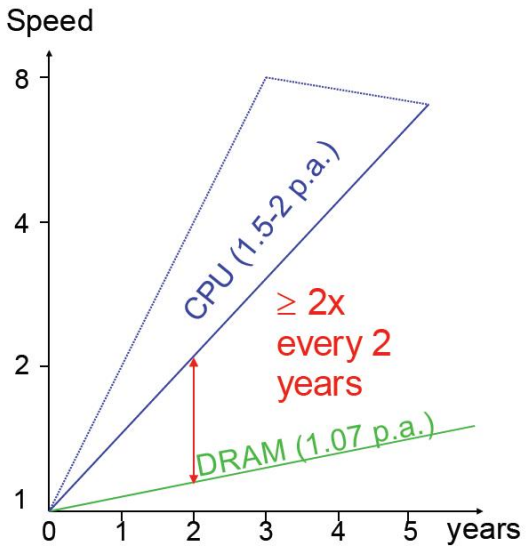


Abbildung 4.1: *Prozessor DRAM Speed-Gap*

halten, setzt man kleine schnelle Pufferspeicher (Caches) ein, die sich zwischen Hauptspeicher und Prozessor befinden. Speicheranfragen der CPU werden nun durch den Cache bearbeitet. Ggf. müssen Daten aus dem Hauptspeicher in den Cache geladen werden. Hier gibt es verschiedene Strategien, auf die weiter unten detaillierter eingegangen wird. Wenn der größte Teil der Speicherzugriffe durch den Cache abgedeckt wird, werden im Vergleich zu den Zugriffszeiten auf den Hauptspeicher sehr kurze Zugriffszeiten realisiert.

Lokalitätseigenschaft

Untersuchungen des Speicherzugriffsverhaltens von Programmen haben ergeben, dass diese Zugriffe eine große Lokalität besitzen. Man kann also durchaus davon ausgehen, dass Caches die Gesamtperformance des Systems verbessern. Aber um genau dies detaillierter zu untersuchen, kann der V-Power-Speicher-Profiler eingesetzt werden. Man unterscheidet folgende Lokalitäten:

- Temporale Lokalität:
  Ein Programm verteilt seine Speicherreferenzen innerhalb von kurzen Zeitspannen uneinheitlich über seinen Adressraum. Die Bereiche, die dabei bevorzugt werden, bleiben weitgehend über längere Zeitspannen dieselben. Daraus folgt, dass ein Programm zu jedem Zeitpunkt in naher Zukunft mit großer Wahrscheinlichkeit nur solche Adressen referenzieren wird, die es in der unmittelbaren Vergangenheit referenziert hat. (z.B. Programmschleifen, Stacks)
- Räumliche Lokalität:
  Die von einem Programm benutzten Teile des Adressraums bestehen generell aus einer Anzahl von individuellen, jeweils zusammenhängenden Segmenten. Das bedeutet, zu jedem Zeitpunkt werden in naher Zukunft mit großer Wahrscheinlichkeit nur solche Adressen referenziert, die nahe bei denen liegen, die in der unmittelbaren Vergangenheit referenziert wurden. (z.B. sequentieller Code, Arrays)

Aufbau von Caches

Abbildung 4.2 zeigt eine typische Struktur eines Cache-Hauptspeicher-Systems [16] .

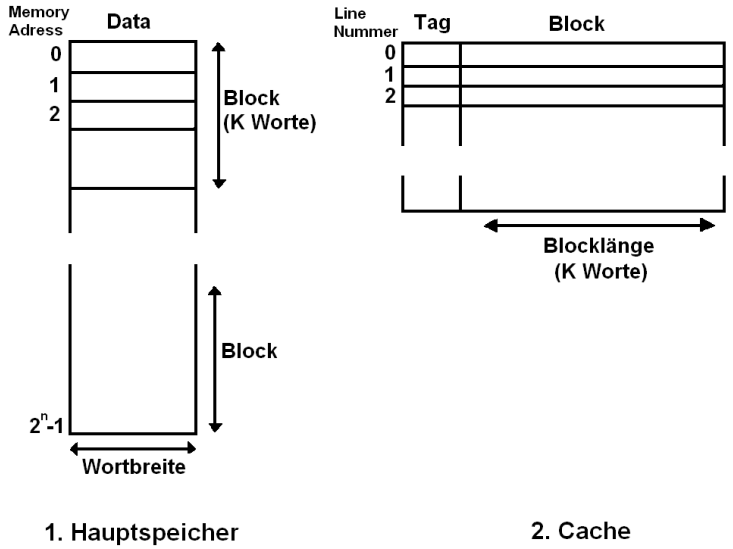


Abbildung 4.2: *Cache-Hauptspeicher-Struktur*

Der Hauptspeicher besteht aus maximal 2^n adressierbaren Worten. Jedes Wort hat dabei eine eindeutige n-Bit Adresse. Um das Mapping zwischen Cache und Hauptspeicher zu vereinfachen, unterteilt man den Hauptspeicher in $M = \frac{2^n}{K}$ gleich große Blöcke zu je K Worten. Der Cache besteht seinerseits aus C Zeilen zu ebenfalls je K Worten. Die Anzahl der Cache-Zeilen ist dabei sehr viel kleiner als die der Hauptspeicherblöcke. Da zu jedem Zeitpunkt immer nur ein Teil der Hauptspeicherblöcke in Cache-Zeilen gespeichert sein kann, und daher eine Cache-Zeile nicht eindeutig und permanent einem speziellen Block zugeordnet werden kann, enthält sie noch einen so genannten Tag, der den jeweils gespeicherten Block eindeutig identifiziert. Das in Abbildung 4.3 dargestellte Valid-Bit zeigt die Gültigkeit einer Cachezeile an. Nach einem Neustart sind zunächst alle Cachezeilen ungültig. Erst wenn eine Cachezeile durch die Anfrage des Prozessors mit Daten gefüllt wird, wird das Valid-Bit gesetzt. Ein weiteres wichtiges Bit ist das Dirty-Bit. Dieses ist in der Abbildung nicht aufgeführt und wird im Abschnitt Schreibstrategien weiter unten näher erläutert. Um möglichst viele Cachehits zu erreichen, sollte ein Cache möglichst viel Speicherkapazität aufweisen. Diesem Ziel stehen hohe Kosten und Geschwindigkeitseinbussen entgegen. Daher ist es üblich, Caches zu hierarchisieren. In der Regel werden ein so genannter Level 1-Cache, mit einer Speichergröße von 4KB bis 256KB und ein dahinter angeordneter Level 2-Cache, mit einer Speichergröße von 256KB bis 4096KB, eingesetzt. Theoretisch könnte man die Hierarchisierung weiter fortsetzen. Grundsätzlich kann zwischen Blocking- und Non-Blocking-Caches unterschieden werden. Blocking bedeutet in diesem Zusammenhang, dass immer nur eine Schreib- oder Leseanfrage bearbeitet werden kann. Non-blocking Caches erlauben im Gegensatz dazu weiteren lesenden Zugriff auf den Cache während einer Miss-Behandlung. Das V-Power Projekt beschränkt sich auf die Betrachtung von Blocking-Caches.

Organisation von Caches

Caches unterscheiden sich in ihrer Assoziativität [17]. Beim Direkt Mapping gibt es für eine Adresse nur einen möglichen Cacheblock. Es liegt also eine Assoziativität von 1 vor. Ein weiterer Spezialfall eines N-assoziativen Caches stellt eine vollassoziative Organisation dar. Hier können die Daten einer Adresse in jedem Block liegen. Folgendes Beispiel zeigt einen 2-Way Associative Cache.

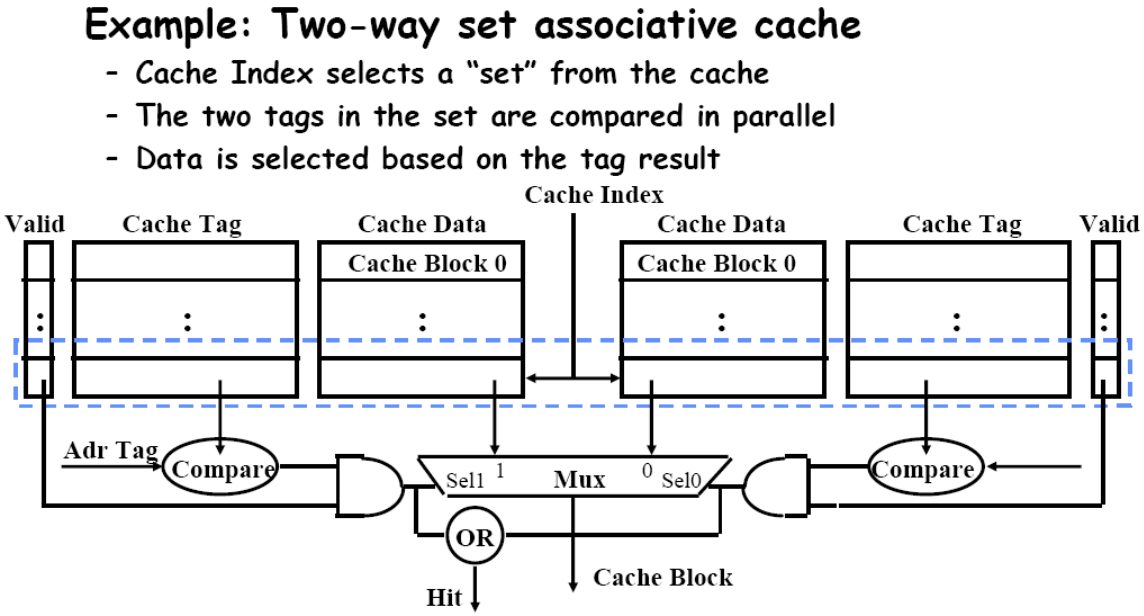


Abbildung 4.3: *Beispiel eines Two-Way-Associative-Cache*

Wir nehmen allerdings von der oben beschriebenen Sichtweise der Assoziativität Abstand. Wenn man von Abbildung 4.3 ausgeht, hat bei einer Assoziativität von 2 jeder Set 2 Blöcke, bei einer Vielzahl von Sets. Das führt dazu, dass nun das Set mit Hilfe der Adresse (Index) eindeutig bestimmbar ist und man die Zeilen innerhalb eines Sets vergleichen muss.

Ersetzungs-Algorithmen

Da die Caches nicht den gesamten Bereich des Hauptspeichers abdecken, müssen regelmäßig Daten aus dem Cache verdrängt werden, um aktuell vom Prozessor benötigte Daten aufnehmen zu können. Funktionen, die bestimmen, welcher alte Block beim Laden eines Neuen aus dem Cache entfernt werden soll, haben natürlich das Ziel, die Miss-Rate zu optimieren. Gleichzeitig müssen sie aus Performancegründen in Hardware implementiert werden und sollten daher nicht zu aufwendig sein. Es gibt eine ganze Reihe solcher Funktionen. Die Wichtigsten sind hier aufgeführt:

- LRU (least-recently-used):
  Ersetzt wird der am längsten unreferenzierte Block. Da die zuletzt referenzierten Blöcke am wahrscheinlichsten in der Zukunft wieder referenziert werden, darf man vermuten, dass dieser Algorithmus die niedrigste Miss-Rate ergibt. Implementiert werden kann er mit Hilfe von zusätzlichen Statusbits pro Zeile.

- FIFO (first-in-first-out):
  Ersetzt wird der Block, der am längsten im Cache war. Die Implementation kann einfach z.B. mit einem round-robin-schedule erfolgen.

- LFU (least-frequently-used):
  Ersetzt wird der Block, der am wenigsten benutzt wurde. Bei der Implementierung muß ein Zähler pro Zeile hinzugefügt werden.

Schreibstrategien

Alle Schreibzugriffe des Prozessors müssen spätestens dann, wenn eine Cachezeile überschrieben werden soll, in den Hauptspeicher zurück geschrieben werden. Es gibt unterschiedliche Strategien, dies zu realisieren.

- Bei einem Cache-Hit kann man folgende Möglichkeiten unterscheiden:

  – Write-Through:
    In diesem Fall werden Schreiboperationen, die auf den Cache erfolgen, direkt an die nächste Hierachieebene weitergeleitet. Dies ist relativ einfach zu realisieren. Außerdem werden Konsistenzprobleme vermieden. Nachteilig an diesem Verfahren ist allerdings der hohe Cache-Hauptspeicher-Verkehr, der sich negativ auf die Gesamtleistung des Systems auswirkt.

  – Write-Back:
    Die Write-Back-Strategie vermeidet jeden unnötigen Cache-Hauptspeicher-Verkehr. Hier wird eine Cachezeile erst dann in den Hauptspeicher zurückgeschrieben, wenn diese durch eine andere verdrängt werden soll. Um dies realisieren zu können, ist pro Cachezeile ein so genanntes Dirty-Bit erforderlich. Dieses Bit wird gesetzt, wenn eine Schreiboperation auf eines der Worte in dieser Zeile erfolgt. Der Nachteil dieser Strategie sind Konsistenzprobleme, da sich der Inhalt des Hauptspeichers nicht ständig auf dem aktuellen Stand befindet.

- Bei einem Cache-Miss kann man folgende Möglichkeiten unterscheiden:

  – Write-Allocate:
    Hier wird zunächst das zu beschreibende Wort aus der nächst höheren Ebene geholt, bevor es aktualisiert wird.

  – Non-Write-Allocate
    Hier wird das zu beschreibende Wort, vorbei an der Cachehierarchie, direkt im Hauptspeicher aktualisiert. Das kann für Anwendungen vorteilhaft sein, bei denen geschriebene Daten nicht wieder gelesen werden müssen. Dadurch werden später benötigte Blöcke nicht aus dem Cache verdrängt.

4.2.2 Anforderungen

Im konkreten Fall des Cache Moduls sind folgende Anforderungen zu erfüllen:

- Es soll die Gesamtzahl aller Zugriffe auf ein Cache Modul gezählt werden.
- Für die Leseanfragen auf den Cache sollen alle Cache Hits und alle Cache Misses gezählt werden.
- Für die Schreibanfragen auf den Cache sollen alle Cache Hits und alle Cache Misses gezählt werden.
- Es soll die Anzahl der WriteBack-Operationen protokolliert werden.
- Die Häufigkeit des Verdrängens einer Zeile (Line Flushes) aus dem Cache soll aufgezeichnet werden.
- Die Zugriffsdauer, gemessen in der Anzahl benötigter Taktzyklen, soll protokolliert werden.
- Wir beschränken uns auf achtfach assoziative Caches. D.h. das max. 8 Zeilen pro Set auftreten können.

4.2.3 Konzept

Die Ergebnisse der Konzeptphase sind in Abbildung 4.4 festgehalten.

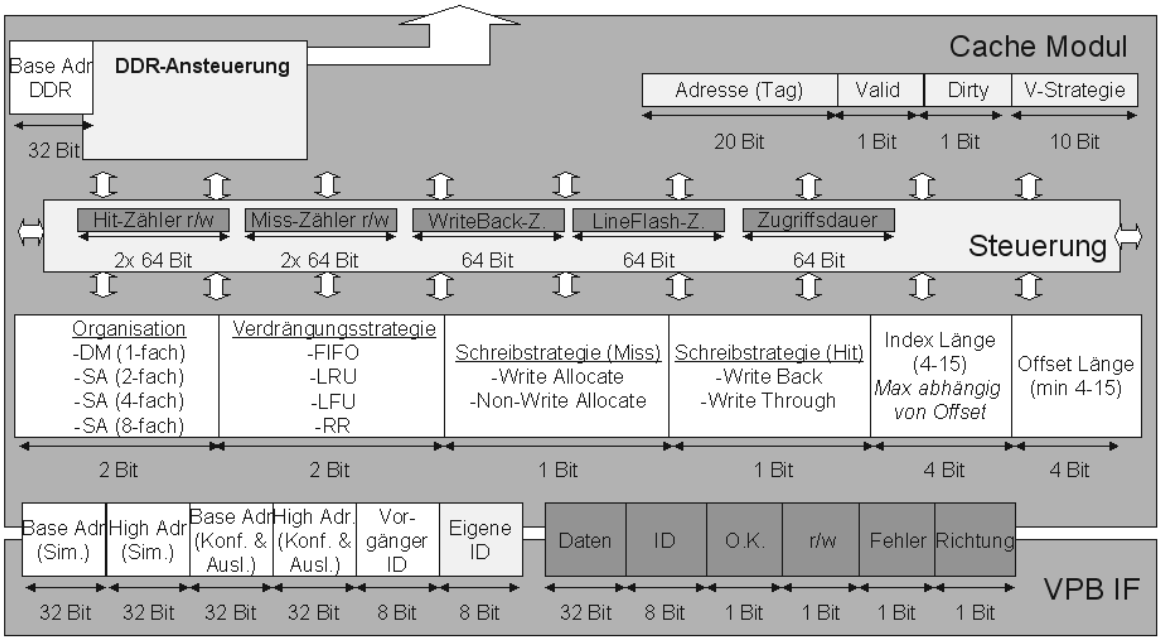


Abbildung 4.4: *Cache Schema*

Im unteren Bereich ist das VPB IF zu sehen, das die Schnittstelle zwischen dem Speichermodul und dem Bus realisiert. Die Boxen auf der linken Seite zwischen dem VPB IF und dem Cache Modul stehen für interne Speicher. Über diese soll das VBP IF überprüfen können, ob der Speicherbaustein aktiviert werden muss. Auf diese Speicher hat zusätzlich auch das Cache Modul selbst Zugriff, da es diese u.a. für die Verwaltungsdatenanfragen benötigt.

Die Boxen auf der rechten Seite symbolisieren die Leitungen, die vom Bus über das VPB IF zum Cache Modul durchgeleitet werden.

In der nächsten Zeile befinden sich alle Konfigurationsdaten, die für das Verhalten des Caches notwendig sind. Dazu gehören die Organisationsmethoden, die Verdrängungsstrategien sowie die Schreibstrategien und die Zusammensetzung von Tag und Index. Die unterschiedlichen Einstellungen werden über eine Bitkodierung zugeordnet. Die jeweils reservierten Bitlängen sind unterhalb der Boxen zu sehen.

In dem Steuerungsblock oberhalb sind alle wichtigen Zähler mit den entsprechenden Bitbreiten angegeben.

Da die Verwaltungsdaten im Hauptspeicher abgelegt werden sollen, befindet sich im oberen linken Bereich der Abbildung die DDR Steuerung, deren Zweck das Auslesen und Speichern der Daten ist.

Im rechten Bereich ist angegeben, wie sich eine Cachezeile aus den vier Teilbereichen Valid-Bit, Dirty-Bit, Tag und Verdrängungsbits zusammensetzt.

4.2.4 Ressourcenbedarf

Der Speicherbedarf der relevanten Verwaltungsdaten eines Cacheblocks variiert je nach Konfiguration des Bausteins. Um die benötigte Größe nach oben abschätzen zu können, haben wir den Worst-Case bestimmt. Nach Vorgabe sollten die Cache-Bausteine eine Maximalgröße von bis zu 4 MB unterstützen. Bei der minimalen Offsetlänge von 4 Bit werden also 4 Worte pro Zeile, die jeweils 4 Byte groß sind, gespeichert. So werden also in einer Cachezeile 16 Byte abgelegt.

Umgerechnet werden so 262.144 Cachezeilen benötigt, um alle Daten bei dieser Konfiguration zu repräsentieren. Da, wie in dem oben stehenden Diagramm zu sehen, jede Cachezeile aus 32 Bit zusammengesetzt ist (Tag, Valid-Bit, Dirty-Bid und Verdrängungsbits), werden in diesem Fall 1MB Speicher für die Verwaltung benötigt.

Bei einem Standard Level1 Cache mit einer Größe von 256KB und 4 Worten pro Cachezeile ergeben sich 64KB Verwaltungsdaten.

Da im Betrieb der Hardware mehrere Cache-Module gleichzeitig aktiv sein werden, die jeweils, wie oben gezeigt 1MB Speicher benötigen, kommt lediglich der DDR Ram als Speicher für die Verwaltungsdaten in Frage.

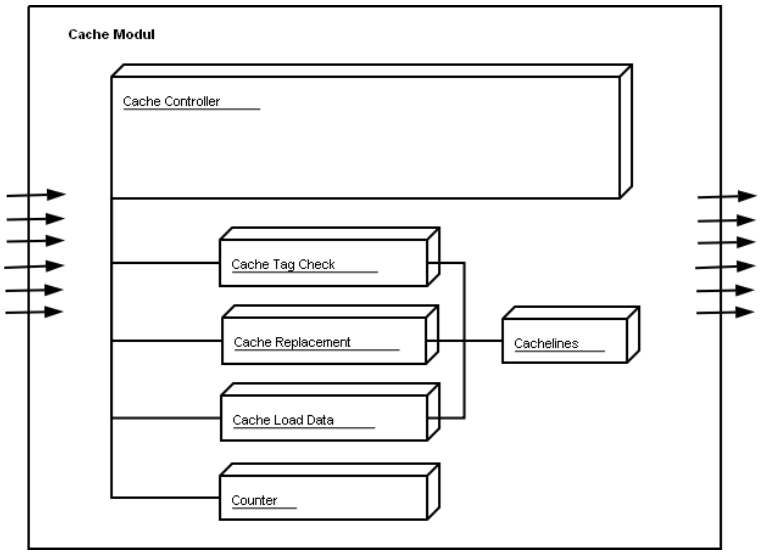


Abbildung 4.5: *Aufbau des Cache Moduls*

Bei einem Speicherzugriff werden die bis zu 8 Sets (dies ist die obere Schranke der Assoziativität) aus dem Speicher in interne Register geladen, dort modifiziert und anschließend wieder in den DDR Ram zurückgeschrieben. So werden pro Modulaktivität bis zu 16 Speicherzugriffe benötigt.

4.2.5 Realisierung

4.2.5.1 Cache Aufbau

Das Cache Modul ist als VHDL-Structure umgesetzt worden, so dass es selbst wieder aus kleineren Unterkomponenten besteht. Diese werden nach diesem Abschnitt näher erläutert. Abbildung 4.5 gibt einen Überblick des Aufbaus.

Die Portmap der Cache Entity ist in Tabelle A.4 (Portmap des Cache-Moduls) zu sehen. Die ein- und ausgehenden Leitungen sind, wie in Kapitel 4.2.3 beschrieben, mit dem VPB IF Modul verbunden. Eingangsleitungen sind mit cacheIN_xxx und Ausgangsleitungen mit cacheOUT_xxx bezeichnet. Zusätzlich kommuniziert das Cache Modul zum Laden und Speichern der Verwaltungsdaten mit einem DDR-Controller. Diese Leitungen sind mit cacheInFromDDR_xxx bzw. mit cacheOutToDDR_xxx bezeichnet.

4.2.5.2 Cache Control

Das erste der genannten vier Module ist das Cache Control Modul. Es übernimmt die Steuerung aller Zugriffe und aktiviert bzw. deaktiviert die restlichen Untermodule.

In Abbildung 4.6 ist ein High Level State-Chart Diagramm zu sehen, das einen Lesezugriff des

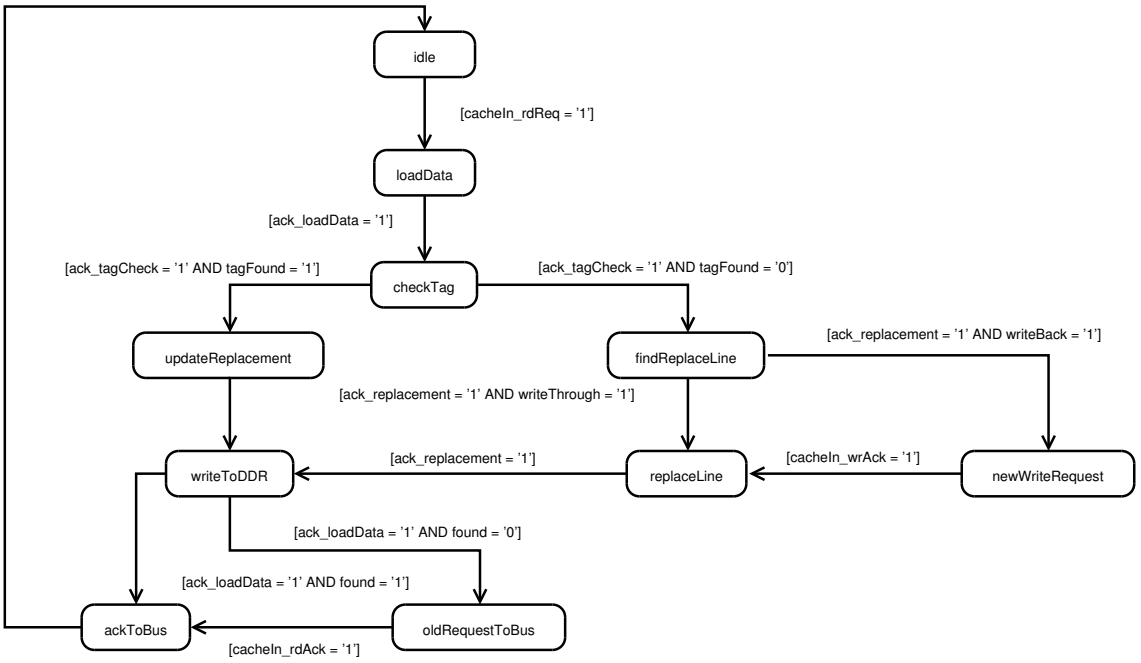


Abbildung 4.6: *Cache State Chart Diagramm - Lesezyklus*

Cache Control Moduls repräsentiert. Der Lesemodus ist etwas einfacher als der Schreibmodus, da dort lediglich nach *write_back* und *write_trough* unterschieden werden muss. Bei der Schreibvariante kommt noch die Unterscheidung von *write_allocate* und non_*write_allocate* hinzu.

Im Lesefall wird zuerst der *loadData* Zustand erreicht. Hier aktiviert der Controller das *Load Data Modul* (siehe Abschnitt 4.2.5.3), welches die passenden Verwaltungsdaten aus dem DDR Speicher in die acht internen Zwischenspeicherregister lädt. Sobald alle Register gefüllt sind, wird in den Zustand *checkTag* gewechselt und das *Load Data Modul* deaktiviert. Dort wird das zu lesende Datum vom *Tag Check Modul* mit den Einträgen aus den Registern verglichen. Falls das Datum gefunden wird, werden zuerst die Verdrängungsbits vom *Replacement Modul* im Zustand *update-Replacement* angepasst. Anschließend werden die Register vom *Load Data Modul* wieder zurück in den DDR RAM geschrieben, ein Ack auf den Bus gelegt (*ackToBus*) und schließlich zurück in den *idle* Zustand gewechselt. Hier wartet das Modul auf die nächste Lese- oder Schreiboperation. Falls das Datum nicht gefunden wurde, muss bei der *write_back* Variante zuerst ein neuer Schreibbefehl auf den Bus gelegt werden (falls das Dirty Bit auf 1 steht), um den dahinterliegenden Speicherbausteinen die Änderung mitzuteilen. Ist dieser Vorgang abgeschlossen oder die *write_trough* Variante aktiv, wird die ausgewählte Line ersetzt. Nach dem Zurückschreiben der Daten in den DDR Speicher (durch das *Load Data Modul*) wird der Request an die anderen Speicher Module weitergeleitet, da das Datum bisher noch nicht gefunden wurde. Sobald das Ack dieses Vorgangs eingeht, wird ein Ack von diesem Modul auf den Bus gelegt und in den *idle* Zustand gewechselt.

Beim Schreibzugriff (vgl. Abbildung 4.7) ist der einfachste Fall, dass ein Datum nicht gefunden wird und non_*write_allocate* aktiviert ist. Falls dies gegeben ist, werden wie beim Lesezugriff zuerst die

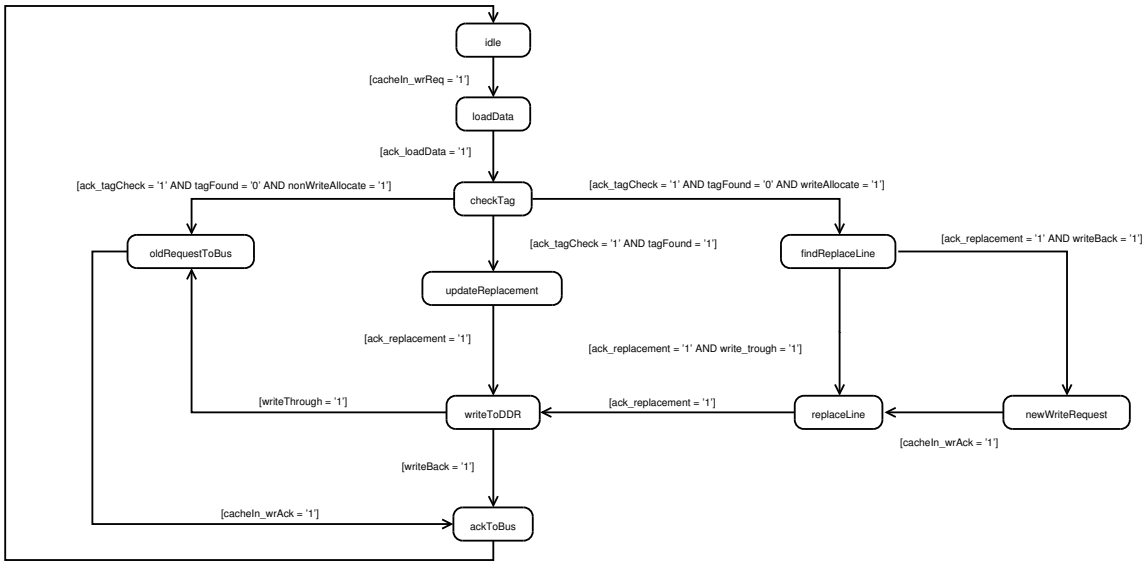


Abbildung 4.7: *Cache State Chart Diagramm - Schreibzyklus*

Sets in die Register geladen (vom *Load Data Modul*) und anschließend mit dem Tag der anliegenden Adresse verglichen (vom *Tag Check Modul*). Da das Datum nicht im Speicher vorhanden ist und das Datum auch nicht aufgenommen werden soll, muss lediglich der WriteRequest an die dahinterliegenden Speicher Module weitergeleitet werden. Nach Eintreffen des Ack-Signals von diesen Modulen wird ein weiterer Ack für die vorgelagerten Module auf den Bus gelegt und in den *idle* Zustand gewechselt. Wird das Datum hingegen gefunden, so werden die Verdrängungsbits des gefundenen Sets aktualisiert (vom *Replacement Modul*) und die Änderungen in den DDR zurückgeschrieben (vom *Load Data Modul*). Falls *write_back* aktiviert ist, wird direkt ein Ack zurückgegeben ohne den Request an die dahinterliegenden Module weiterzugeben. Andernfalls wird der alte Request auf den Bus gelegt, damit die anderen Module die Änderung ebenfalls verarbeiten können. Nach Eintreffen des Ack-Signals wird auch hier schließlich in den *idle* Zustand gewechselt.

Im letzten Fall, dass nach dem Überprüfen des Tags (*checkTag*) das Datum nicht gefunden wurde und *write_allocate* aktiviert ist, muss das Datum in den Cache eingelagert werden. In dem Zustand *findReplaceLine* wird das *Replacement Modul* aktiviert und bestimmt die zu verdrängende Zeile. Ist *write_back* aktiviert und wurde das zu verdrängende Datum modifiziert (an dem Dirty-Bit erkennbar), muss den anderen Modulen über einen separaten WriteRequest zuerst die Änderung mitgeteilt werden. Sobald das Ack-Signal dieses Vorgangs angekommen ist oder *write_trough* aktiviert ist, wird in den Zustand *replaceLine* gewechselt, welcher das *Replacement Modul* erneut aktiviert und das vorher ausgewählte Set ersetzt. Die Änderungen werden vom *Load Data Modul* als nächstes im DDR aktualisiert, bevor der ursprüngliche WriteRequest auf den Bus gelegt wird, da die Anfrage nicht durch diesen Cache beantwortet werden konnte. Nach dem Empfang des Ack-Signals dieses Requests wird ein neuer Ack auf den Bus gelegt und in den *idle* Zustand gewechselt.

Während eines Simulationszyklus werden diverse Zähler aktualisiert, die später als Ergebnis der Simulation bereitgestellt werden. So wird z.B. abhängig vom Ergebnis des *checkTag* Zustands ent-

weder der Read Hit, Read Miss, Write Hit oder Write Miss Zähler um 1 erhöht. Gleichzeitig wird die Simulationszeit um die vordefinierte Zeit (je nachdem, ob Read/Write Hit/Miss vorliegt) erhöht. Zusätzlich zu dieser modulinternen Simulationszeit wird nach jedem gestarteten Request die Bearbeitungsdauer der dahinterliegenden Module mit auf den Zähler addiert.

Diese Zeiten werden vorerst für den aktuellen Zyklus zwischengespeichert und vor dem Wechsel in den *idle*-Zustand auf den Gesamtsimualtionszähler addiert.

4.2.5.3 Cache Load Data

Das *Load Data Modul* kümmert sich, wie bereits im vorherigen Abschnitt erwähnt, um die Kommunikation mit dem DDR Speicher. Die zu große Menge an Verwaltungsdaten wird im DDR Speicher ausgelagert und bei jedem Simulationsdurchlauf zuerst geladen und später wieder zurückgeschrieben.

Insgesamt beinhaltet das Cache Modul acht 32-Bit Register, in denen sich die CacheLines für einen Request befinden.

Ist die readData Leitung, die vom Cache Controller kommt, auf 1 gesetzt, so werden die Daten vom DDR Speicher geladen. Analog werden die Daten bei writeData auf 1 zum DDR Modul zurückgeschrieben.

Je nach Konfiguration des Cache Bausteins (Direct Mapping, 2-, 4- oder 8-fach Assoziativ) werden gegebenenfalls auch weniger als die genannten acht Register befüllt, bzw. zurückgeschrieben.

4.2.5.4 Cache Tag Check

Das Tag Check Modul wird bei jeder Anfrage direkt nach dem Ladevorgang des *Load Data Moduls* aktiv. Es überprüft, ob der Tag des anliegenden Datums der cacheIn_Adress mit dem Tag aus einem der acht Zwischenspeicherregistern übereinstimmt.

Auch bei diesem Modul können je nach Organistionskonfiguration auch weniger, als acht Register betrachtet werden.

Ist die Prüfung des Tags abgeschlossen, so wird ein Flag über die Leitung found, sowie ggf. die Nummer des Registers, in dem das Datum gefunden wurde, an den Cache Control zurückgegeben.

4.2.5.5 Cache Replacement

Das Replacement Modul ist nach dem *Cache Control* Modul das umfangreichste Modul.

In Abbildung 4.8 ist ein vereinfachtes State Chart Diagramm der Funktionalität abgebildet. Der Zustand *updateStrategie* ist in der Umsetzung für jede Strategie einzeln angelegt (updateFIFO, upda-

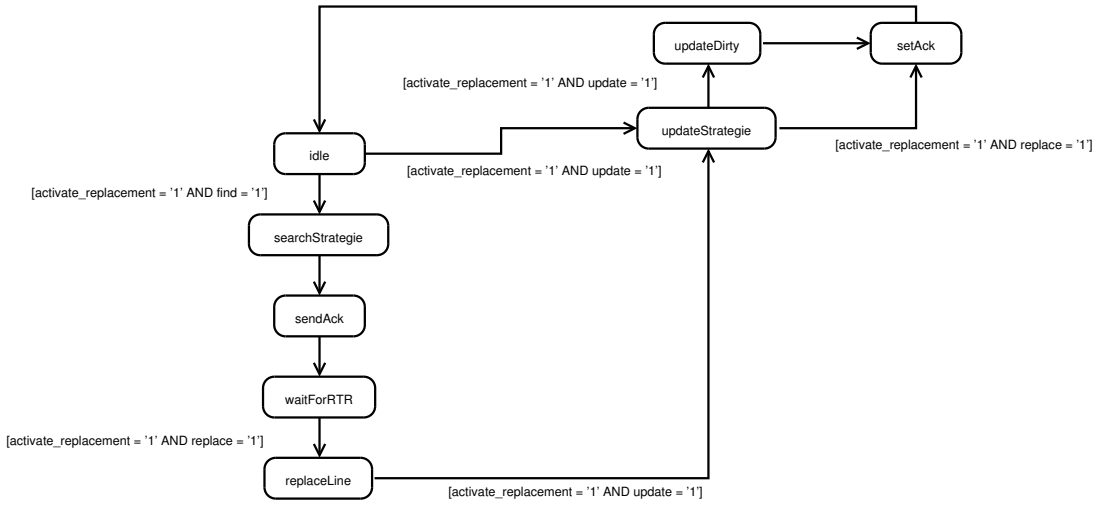


Abbildung 4.8: *Cache State Chart Diagramm - Replacement Modul*

teRoundRobin und updateLRU). Je nach Konfiguration wird der passende Zustand angesteuert und wieder verlassen. Gleiches gilt für den Zustand *searchStrategie*. Diese Vereinfachung wurde aus Komplexitäts- und Anschaulichkeitsgründen durchgeführt.

Das Modul startet im *idle* Zustand. Diesen kann es über zwei Wege verlassen. Der eine ist für die Aktualisierung der jeweiligen Verdrängungsstrategie da, der andere für das Heraussuchen, bzw. Ersetzen einer Cache-Line.

Wird auf ein vorhandenes Datum zugegriffen, so wird das Replacement Modul vom *Cache Control* Modul aktiviert und das update Signal ist auf 1 gesetzt. Dadurch wechselt das Replacement Modul in den Zustand *updateStrategie*. In diesem Zustand wird je nach Konfiguration eine Aktualisierung der Verdrängungsstrategiebits vorgenommen. Mögliche Verfahren sind hierbei Round Robin (FIFO), LRU (Least Recently Used) und LFU (Least Frequently Used).

Anschließend wird in den Zustand *updateDirty* gewechselt, der bei einem Schreibzugriff auf das Datum das Dirty Bit des Datums setzt. Abschließend sendet das Modul ein Ack an das *Cache Control* Modul und wechselt in den *idle* Zustand.

Im anderen Fall wurde das anliegende Datum vom *Tag Check* Modul nicht gefunden, so dass eines der Daten verdrängt werden soll. Dazu wird das Replacement Modul vom Controller aktiviert und das Signal find auf 1 gesetzt. Dieses signalisiert, dass kein simples Update der Verdrängungsbits vorgenommen werden soll, sondern ein Datum bestimmt wird, dass aus dem Cache ausgelagert wird. Dafür wird in den Zustand *searchStrategie* gewechselt, der je nach Strategie die entsprechende Line auswählt. Im Anschluss teilt das Modul dem Controller das zu verdrängende Datum mit, sendet ein Ack und wechselt in den Zustand *waitForRTR* (wait for return to replacement).

Der Controller gibt nun, je nachdem ob *write_trough* oder *write_back* bei der Konfiguration vorliegt,

das Datum über eine neue Schreibanfrage zurück. Anschließend wird das Replacement Modul erneut aktiviert, wobei nun das replace Signal auf 1 gesetzt ist. Daher wechselt das Modul in den Zustand *replaceLine*, in der die Daten aus dem cacheIn_Adress Eingang in das entsprechende Zwischenspeicherregister eingelagert werden.

Im Controller wird zusätzlich das update Signal auf 1 gesetzt, wodurch nach jedem Einlagern automatisch in den *updateStrategie* Zustand gewechselt wird. Auch hier wird das Dirty-Bit im Zustand *updateDirty* aktualisiert. Nach dem Bestätigen des Vorgangs via Ack an den Controller geht das Modul wieder in den *idle* Zustand und wartet dort auf die nächste Anfrage.

Im Folgenden werden die Umsetzungen der drei möglichen Verdrängungsstrategien beschrieben, die ebenfalls in diesem Modul implementiert sind.

4.2.5.6 LRU

Die LRU Strategie (Least Recently Used) ist eine der Strategien, die sich in der Praxis als äußerst effizient herausgestellt hat. Daher ist diese in vielen Caches umgesetzt worden. Es wird, wie der Name bereits sagt, das Datum verdrängt, welches am längsten nicht mehr aufgerufen worden ist.

Die Umsetzung wird über einen Zähler realisiert. Wird auf ein Datum zugegriffen, so bekommt es die höchstmögliche Markierung (bei 8-fach Assoziativität wird es auf „111“ gesetzt, bei 4-fach Assoziativität auf „011“ etc). Der Zähler aller anderen Daten, auf die nicht zugegriffen wurde, wird um 1 erniedrigt.

Bei einem Cache-Miss wird immer das Datum verdrängt, dessen Zähler auf „000“ steht. Am Anfang steht dieser Zähler bei allen Cache Lines auf „000“, da diese noch nicht valide sind. Daher ist gewährleistet, dass immer ein Datum zum Verdrängen zur Verfügung steht.

4.2.5.7 LFU

Die LFU Strategie (Least Frequently Used) verdrängt das Datum, auf das am wenigsten oft zugegriffen worden ist. Der Nachteil dieser Strategie liegt darin, dass neu eingelagerte Worte sehr schnell wieder verdrängt werden. Daten, auf die weit in der Vergangenheit oft zugegriffen worden ist, bleiben hingegen sehr lange im Cache.

Realisiert wird diese Strategie in der PG dadurch, dass die 10 möglichen Bits für die Verdrängungsstrategie durch einen großen Zähler reserviert werden. Dieser wird bei jedem Cache-Hit in der entsprechenden Zeile hochgezählt. Mit 10 Bits können so bis zu 1024 Zugriffe auf ein Datum festgehalten werden. Sollte dennoch ein Überlauf vorkommen, behält der Zähler den Wert von 1024.

Bei einem möglichen Cache-Miss wird das Datum mit dem geringsten Zählerstand ermittelt und verdrängt.

4.2.5.8 FIFO

Die Verdrängungsstrategie FIFO (First In First Out) verdrängt das Datum, das zuerst eingelagert worden ist. Dieses Verfahren kann als Queue realisieren werden, in der ein neu eingelagertes Element grundsätzlich vorne an die Liste angefügt wird. Das Datum, dass sich am Ende der Queue befindet, wird bei einem Cache-Miss verdrängt.

Eine andere mögliche Realisierung, die die Projektgruppe gewählt hat, ist die Round Robin Methode. Bei dieser wird ein Zeiger benötigt, welcher auf die als nächstes zu verdrängende Zeile zeigt. Dieser Zeiger wechselt nach jeder Verdrängung auf die nächste Zeile. Wurde das letzte Element (in unserem Fall das achte) verdrängt, so zeigt der Zeiger als nächstes wieder auf das erste Element der Liste.

In der Hardwarerealisierung der Projektgruppe wurde ein Bit für die Markierung reserviert. Bis auf eine Zeile ist dieses Bit überall auf 0 gesetzt. Die Zeile, in der das Bit auf 1 gesetzt ist, wird als nächstes verdrängt.

4.2.6 Konfigurieren des Moduls

Die Konfigurationsregister des Cache Moduls liegen außerhalb des eigentlichen Moduls. Der komplette Konfigurationsstring wird über die Leitung cacheIn_config (128 Bit breit) ans Modul übergeben.

Der Konfigurationsstring ist wie in Tabelle 4.1 aufgebaut.

Bezeichnung	Position	Belegung
Organisation	0-1	00: Direct Mapping 01: 2-fach Assoziativität 10: 4-fach Assoziativität 11: 8-fach Assoziativität
Verdrängungsstrategie	2-4	000: FIFO 001: LRU 010: LFU 011: RR
Schreibstrategie (Miss)	5	0: Non-Write Allocate 1: Write Allocate
Schreibstrategie (Hit)	6	0: Write Through 1: Write Back
Index Länge (4-15)	7-10	Binärcodierung der Länge (z.B. 0100 für minimale Länge von 4)
Offset Länge (4-15)	11-14	Binärcodierung der Länge (z.B. 0100 für minimale Länge von 4)
Leer (Puffer)	15	0
Read Hit Dauer (Dauer eines Hits bei einem Lesezugriff in Ticks)	16-31	Binärcodierung der Ticks
Write Hit Dauer (Dauer eines Hits bei einem Schreibzugriff in Ticks)	32-47	Binärcodierung der Ticks

Bezeichnung	Position	Belegung
Read Miss Dauer (Dauer eines Miss bei einem Lesezugriff in Ticks)	48-63	Binärcodierung der Ticks
Write Miss Dauer (Dauer eines Miss bei einem Schreibzugriff in Ticks)	64-79	Binärcodierung der Ticks
Leer (Puffer)	80-95	0000000000000000
Baseadress	96-127	Binärcodierung der Baseadress

Tabelle 4.1: *Aufbau des Konfigurationsstrings cacheIn_config*

Die Puffer an den Positionen 15 und 80-95 sind eingefügt um die Daten in 32-Bit Pakete aufteilen zu können.

4.2.7 Auslesen der Ergebnisse

Für die Ergebnisse gilt ähnliches wie für die Konfiguration. Das bedeutet, dass auch hier die Ergebnisregister außerhalb des Cache Moduls liegen. Die Ergebnisse liegen während der Simulation permanent an dem 448 Bit großen Ausgang cacheOut_simResult an.

Dabei ist der Bitstring wie in Tabelle 4.2 zu interpretieren.

Bezeichnung	Position
Read Hit Counter	0-63
Write Hit Counter	64-127
Read Miss Counter	128-191
Write Miss Counter	192-255
Write Back Counter	256-319
Line Flash Counter	320-383
Sim Time Counter	384-447

Tabelle 4.2: *Aufbau des Simulationsergebnisregisters cacheOut_simResult*

4.3 DRAM

4.3.1 Allgemeine Informationen zu DRAMs

Der DRAM Baustein ist ein Speichermodul, das die Daten in Kondensatoren vorhält. Diese Kondensatoren sind in großen Flächen über eine Zeilen- und Spaltenmatrix adressierbar. Meistens gibt es pro DRAM-Modul mehrere Bänke, die jeweils eine unabhängige Kondensatormatrix besitzen.

Da wenig Logik erforderlich ist kann relativ kostengünstig ein großer Speicher zur Verfügung gestellt werden. Allerdings haben die Kondensatoren die negative Eigenschaft, dass sie Ihre gespeicherte Ladung aufgrund von sogenannten Leckströmen verlieren, so dass diese immer wieder neu aufgeladen werden müssen. Diesen Zyklus bezeichnet man als Refresh-Phase. In einem vordefinierten Intervall werden nach und nach alle Zeilen der Matrix ausgelesen und die Werte neu in die Kondensatoren zurückgeschrieben. Während dieser Zeit können keine weiteren Operationen angewendet werden.

Kommt eine Lese- oder Schreibanfrage an das DRAM-Modul, so wird anhand der Adresse zuerst ermittelt in welcher Bank sich das betreffende Datum befindet. Als nächstes wird die betreffende Zeile ermittelt und diese über einen Controller aktiviert und aufgeladen. Im Anschluss folgt das gleiche für die Spalte.

Da dieser Aufladevorgang einige Zeit kostet kann das DRAM-Modul die Daten einer Speicherzeile in einem Schieberegister zwischenpuffern. Kommen nun nacheinander mehrere Anfragen auf die gleiche Speicherzeile, so müssen die Steuerleitungen nicht extra aufgeladen werden, da die Daten bereits im Schieberegister liegen. Diesen Zugriff nennt man Burst-Mode. Im realen DRAM-Modul wird dieser Modus von außen über eine Steuerleitung aktiviert. Da unser Modul diese Informationen nicht bekommen kann, wird davon ausgegangen, dass das Datum stets zwischengepuffert wird. Kommen in der Folge mehrere Anfragen auf die gleiche Zeile werden diese innerhalb der verkürzten Burst-Zeit beantwortet.

4.3.2 Anforderungen

An das DRAM Modul werden folgende Anforderungen gestellt:

- Die Anzahl der Lesezugriffe soll festgehalten werden.
- Die Anzahl der Schreibzugriffe soll festgehalten werden.
- Die Zugriffsdauer, gemessen in der Anzahl benötigter Taktzyklen, soll protokolliert werden.
- Die Anzahl der Refreshes soll protokolliert werden.
- Die Anzahlt der PreLoads soll gespeichert werden.
- Außerdem soll die Zeilenlänge des DRAM Moduls konfigurierbar sein.

4.3.3 Ressourcenbedarf

Der Bedarf der Ressourcen wird im Wesentlichen, neben der Logik, durch die fünf 64 Bit Zähler (Lesezugriffszähler, Schreibzugriffszähler, Zugriffszähler pro Zugriff, Zugriffszähler für die gesamte Aktivitätsdauer des Moduls, Zähler für die Anzahl der Preloads) und den 32 Bit Zähler (Refreshzähler) bestimmt. Des weiteren muss noch festgehalten werden, welche Zeile in welcher Bank

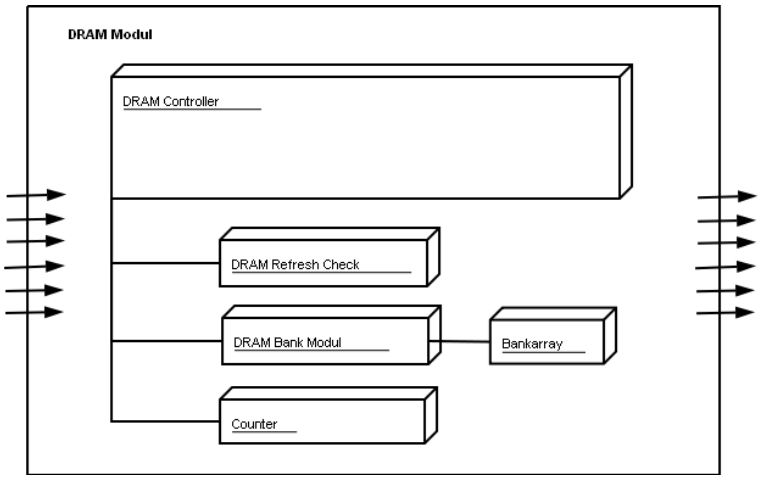


Abbildung 4.9: *Aufbau des DRAM Moduls*

geöffnet ist. Da wir von einer maximalen Anzahl von 8 Bänken pro DRAM Modul ausgehen, wird eine 8 mal 32 Bit große Matrix benötigt.

4.3.4 Realisierung

DRAM Modul Aufbau

Das DRAM Modul besteht aus verschiedenen Einzelkomponenten, wobei eine zentrale Steuereinheit die Kontrolle übernimmt. Es sind mehrere Zähleinheiten im Modul untergebracht, welche die Anzahl der Preloads sowie die Anzahl der Schreib- und Lesezugriff zählen. Außerdem zählt ein Counter wie viele Taktzyklen das DRAM Modul aktiv gewesen ist. Ein weiterer Zähler ist für die Aktualisierung der Simulationszeit zuständig. Das so genannte Bankmodul stellt fest, in welcher Bank sich die angefragte Adresse befindet und ermittelt ob ein Burstzugriff vorliegt. Ob sich der zu simulierende DRAM im Refreshmodus befindet wird durch das Refreshmodul berechnet. Hier ist auch der Zähler für die Anzahl der Refreshes untergebraucht. Eine Auflistung der Eingangsports und der Ausgangsports zusammen mit Beschreibung und Breite beschreibt die Tabelle A.5. Im Folgenden werden die einzelnen Komponenten des DRAM Moduls genauer beschrieben. Abbildung 4.9 verdeutlicht die Struktur des Moduls.

DRAM Controller

Der Controller ist die zentrale Steuerungsinstanz. Das Statechart in Abbildung 4.10 verdeutlicht die Arbeitsweise. Der Controller befindet sich solange im *Idle* Zustand bis vom *VPB_IPIF* eine Lese- oder Schreibanfrage bzw. eine Resetsignalisierung kommt. Bei einer Lese- oder Schreibanfrage wird in den *Refresh* Zustand gewechselt. Hier aktiviert der Controller das Refreshmodul. Dieses

liefert den Zeitoverhead zurück, der, wenn das DRAM-Modul im Refreshzyklus ist, auftritt. Die erfolgreiche Berechnung wird durch das *ack_refresh* Signal signalisiert und im Anschluss wird der Zähler für die Taktzyklen aktualisiert. Der darauf folgende Zustand *Check Burst* aktiviert das Bank Modul. Hier wird geprüft ob ein Burstzugriff vorliegt oder ob eine neue Zeile geöffnet werden muss. Entsprechend der Auswertung und der Art der Anfrage, wird in den nächsten Zustand gewechselt. In Abhängigkeit des Zustands, werden die jeweiligen Zähler erhöht und dann die neue Simulationszeit gesetzt. Im Zustand *AckToBus* wird dem zugehörigen *VPB_IPIF* mitgeteilt, dass das DRAM Modul die Simulation abgeschlossen hat. Der letzte Zustand *Add OverallSimtime* addiert die Simulationszeit, die der DRAM für den aktuellen Zyklus benötigt hat, zu der Gesamtsimulationzeit. Die Trennung von Gesamtsimulationszeit und der Simulationszeit des aktuellen Zyklus ist wichtig für die Bestimmung des potenziellen Overheads durch die Refreshzeit. Im Falle der Resetsignalisierung werden alle Zähler auf Null gesetzt und das Bankmodul zurückgesetzt. D.h., dass alle Zeilen in den einzelnen Bänken des DRAMs geschlossen werden.

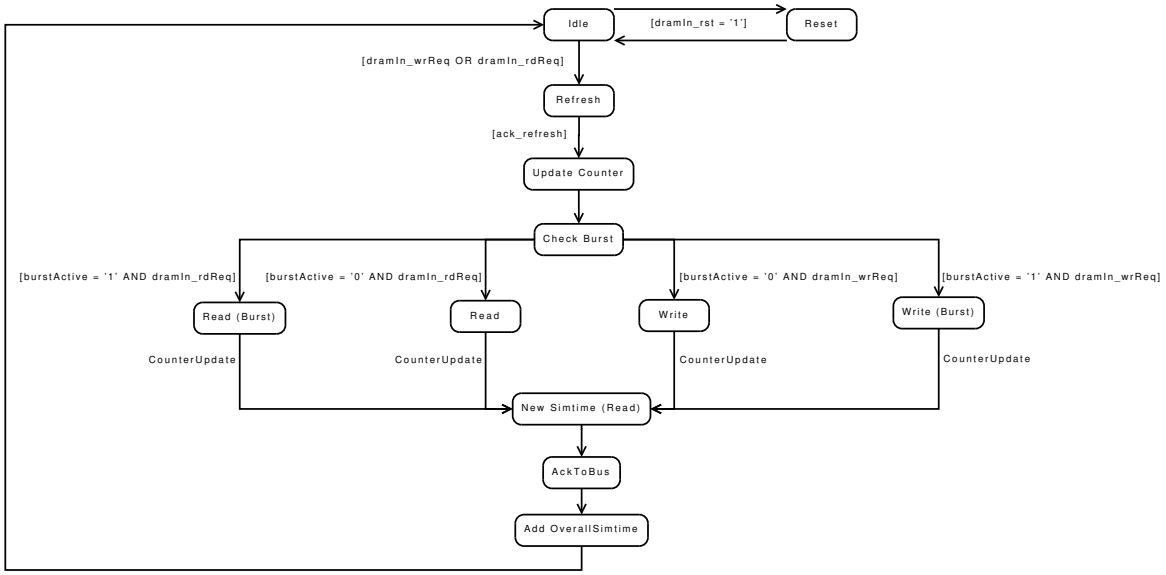


Abbildung 4.10: *StateChart DRAM Control Modul*

DRAM Refesh Modul

Die Abbildung 4.12 verdeutlicht die Arbeitsweise dieses Moduls. Der *Idlezustand* wird verlassen, wenn von dem Controller das *activate_refresh* Signal auf '1' gesetzt wird. Im Zustand *Init* wird die aktuelle Simulationszeit zwischengespeichert und der interne Refreshzähler auf Null gesetzt. Im *Refreshzustand* wird nun die Hauptarbeit gemacht. Hier soll nun festgestellt werden, ob das DRAM Modul, zu dem Zeitpunkt der Anfrage, ein Refresh durchführen würde. Um hardwareaufwendige Divisionen zu umgehen, wird sukzessive die Dauer eines solchen Refreshzyklus und die Intervalldauer zwischen den Refreshzyklen von der aktuellen Simulationszeit abgezogen. Dies wird solange gemacht, bis die Differenz kleiner ist als ein Refreshdurchlauf (Refreshzyklus + Intervalldauer). Jetzt

ist der Zeitpunkt erreicht, in dem gerade wieder ein Refresh durchgeführt würde. Um festzustellen, ob dieser zu der aktuellen Simulationszeit weiterhin anhält, wird nun nur noch die Dauer der Refreshphase abgezogen. Ist das Ergebnis positiv, so würde sich das DRAM Modul nicht in einem Refeshzyklus befinden und der Overhead ist Null. Ist das Ergebnis aber negativ, so stellt der Betrag die Zeitspanne dar, die das Modul noch braucht um den Refreshzyklus abzuschließen. Mit jeder Subtraktion wird der Refreshzähler um 1 erhöht, sodass hier immer die Gesamtzahl der durchgeführten Refreshzyklen festgehalten wird. Dies wird an das Ergebnisregister weitergeleitet. Das Beispiel in Abbildung 4.11 verdeutlicht die Vorgehensweise. Die Dauer für ein Refreshintervall soll hier 50 Takte und die Dauer für die Refreshphase soll 30 Takte betragen. Die aktuelle Simulationszeit ist 650. Diese Werte sind natürlich nicht repräsentativ. Das Pfeildiagramm zeigt wie sukzessive die Summe der Dauer für ein Refreshintervall und der Dauer für einen Refeshzyklus von der aktuellen Simulationszeit abgezogen wird. Am Ende hält der *Refreshcount* fest, dass zu dem Zeitpunkt 650, 9 Refreshphasen durchlaufen worden sind. Außerdem Signalisiert die -20, dass wir uns gerade in einer solchen Phase befinden und diese noch 20 Takte benötigt um abgeschlossen zu werden. Dies wird über das *Overhead* Signal dem Controller angezeigt. Im Zustand *Ack* wird der *Overhead* zusammen mit einer Bestätigung, dass die Berechnung erfolgreich durchgeführt wurde, an den Controller zurückgegeben.

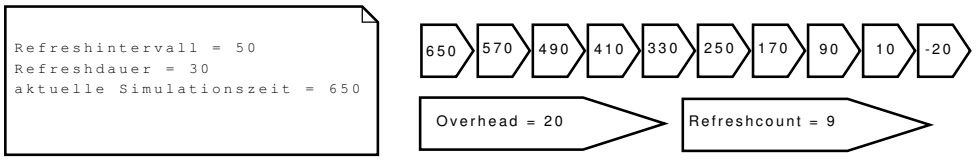


Abbildung 4.11: *Beispiel einer Refreshberechnung*

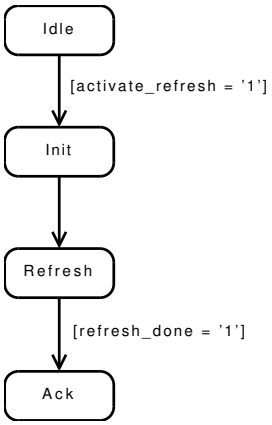


Abbildung 4.12: *StateChart DRAM Refresh Modul*

DRAM Bank Modul

Das Statechart in Abbildung 4.13 verdeutlicht die Arbeitsweise des DRAM Bank Moduls. Wenn von dem Controller das Signal *activate_banks* auf '1' gesetzt wird, geht das Bank Modul in den Zustand *ComputeAdress*. Hier wird, durch das Abziehen der Basisadresse von der zu verarbeitenden Adresse, die Adresse normalisiert, so dass jedes DRAM Modul intern bei der Adresse Null beginnt. Im Zustand *Init* wird die Bank bzw. die Position im Bankarray bestimmt, in der die Adresse liegen würde und es wird der Teil der Adresse herausgeschnitten der im Zustand *Searchadress* mit der Adresse im Bankarray verglichen wird. Stimmen diese überein, so wäre die entsprechende Zeile in der Bank geöffnet und es liegt somit ein Burstzugriff vor. Im *Sendack* Zustand wird dem Controller die erfolgreiche Berechnung durch das Senden eines *Acks* bestätigt und eine '0' an den Ausgangsport *Burst* angelegt, falls kein Burstzugriff erfolgt ist bzw. eine 1 falls ein Burstzugriff erfolgt ist.

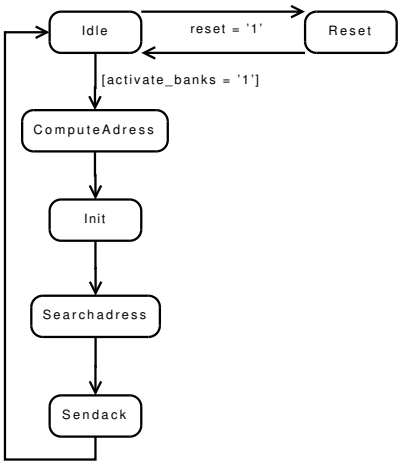


Abbildung 4.13: *StateChart DRAM Bank Modul*

4.3.5 Konfigurieren des Moduls

Die Konfiguration für das RAM-Modul wird über einen 192 Bit langen Bitstring, dessen Aufbau die Tabelle 4.3 beschreibt, realisiert.

Bezeichnung	Position	Belegung
Anzahl an Bänken	0-2	Die Anzahl an Bänken muss im Intervall von 1-8 liegen
Anzahl Zeilen pro Bank	3-20	Die Anzahl an Zeilen muss zwischen 1 und 262.144 liegen
Anzahl an (32-Bit) Worten pro Zeile	21-31	Die Anzahl an Worten pro Zeile muss zwischen 1 und 2.048 liegen
Zugriffsdauer Read normal	32-47	Zugriffsdauer für einen Read-Zyklus, der nicht vorher zwischengespeichert wurde
Zugriffsdauer Read Burst-Mode	48-63	Zugriffsdauer für einen Read-Zyklus, der vorher zwischengespeichert wurde
Zugriffsdauer Write normal	64-79	Zugriffsdauer für einen Write-Zyklus, der nicht vorher zwischengespeichert wurde

Bezeichnung	Position	Belegung
Zugriffsdauer Write Burst-Mode	80-95	Zugriffsdauer für einen Write-Zyklus, der vorher zwischengespeichert wurde
Refresh-Zeit	96-127	Zeit zwischen zwei Refresh-Zyklen
Refresh-Dauer	128-159	Dauer eines Refresh-Zyklus
Baseadress	160-191	Binärcodierung der Baseadress

Tabelle 4.3: *Aufbau des Konfigurationsarrays*

4.3.6 Auslesen der Ergebnisse

Die Simulationsergebnisse der Zähler werden auf das 288 Bit lange Ausgangsport *dramOut_simResult* abgebildet. Der Aufbau dieses Bitstrings beschreibt die Tabelle 4.4.

Bezeichnung	Position
Read Counter	0-63
Write Counter	64-127
Preload Counter	128-191
Sim Time Counter	192-255
Refresh Counter	256-287

Tabelle 4.4: *Aufbau des Ergebnisarrays*

Anmerkung: Der Preload Counter gibt an, wie oft ein Datum zwischengespeichert wurde. Es gilt also: Anzahl der Preloads ergibt sich aus der Differenz der Anzahl der Gesamtzugriffe und der Burstzugriffe.

4.4 Scratch Pad Modul

4.4.1 Allgemeine Informationen zu Scratch Pad Speicher

Der Scratch Pad Speicher stellt eine gute Alternative zu Caches dar. Hierbei handelt es sich um kleine Speichereinheiten, welche in einen Teil des Adressraumes abgebildet werden. Häufig verwendete Variablen oder Befehle werden dem Scratch Pad Speicher zugewiesen. Hierfür ist der Compiler verantwortlich. Der große Vorteil hierbei ist, dass im Gegensatz zum Cache, keine Hardware erforderlich ist um den Speicherinhalt zu verwalten. Daraus Resultiert ein sehr geringer Energiebedarf pro Zugriff, sowie eine schnelle Antwortzeit. Abbildung 4.14 verdeutlich die Energieeffizienz und vergleicht den Energiebedarf von 4fach- und 2fach assoziativen Caches, Caches im Direct Mode und den Energieverbrauch eines Scratch Pad. Der Nutzen und die Qualität des Scratch Pad Speicher hängt vom Compiler ab und ist dadurch schwierig vorher zu sagen. Technisch werden Scratch Pad

Speicher, wie Caches auch, mit SRAMs (statischen RAM) realisiert. Die hohe Perfomance dieses Speichertyps wird mit Hilfe von Transitoren, die eine bistabilen Kippstufe realisieren, erreicht [15].

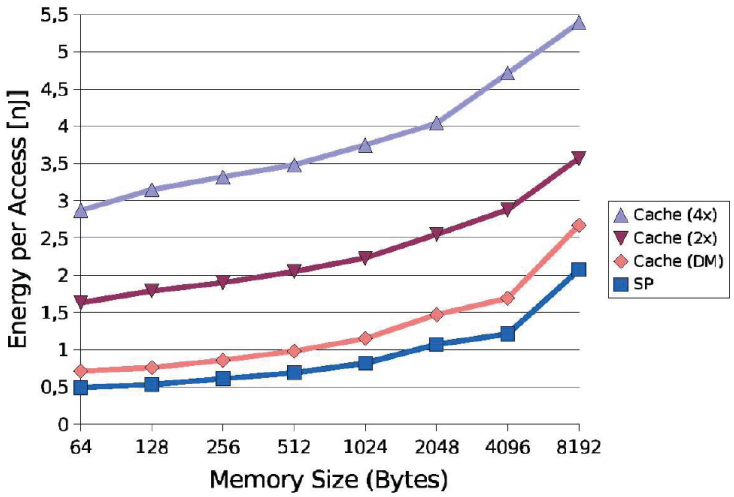


Abbildung 4.14: *Vergleich des Energiebedarf pro Zugriff von Scratch Pad und Caches*

4.4.2 Anforderungen

Aufgrund der im Vergleich zum Cache Modul geringen Komplexität des Scratch Pad Moduls, reichen folgende Punkte aus, um aussagekräftige Simulationergebnisse zu erhalten:

- Die Lesenzugriffe sollen gezählt werden.
- Die Schreibzugriffe sollen gezählt werden.
- Die Zugriffe, gemessen in der Anzahl benötigter Taktzyklen, soll protokolliert werden.

4.4.3 Ressourcenbedarf

Der Bedarf an Ressourcen wird durch die fünf 64Bit Zähler dominiert. Wie auch bei dem DRAM Modul gibt es einen Zähler der die Dauer eines Zugriffzyklus zählt und einen der die Gesamtdauer zählt, die das Modul aktiv gewesen ist. Zudem werden ein Lesezugriffs- und ein Schreibzugriffszähler benötigt. Des Weiteren ist ein 64Bit Zähler für die Berechnung der neuen Simulationszeit erforderlich.

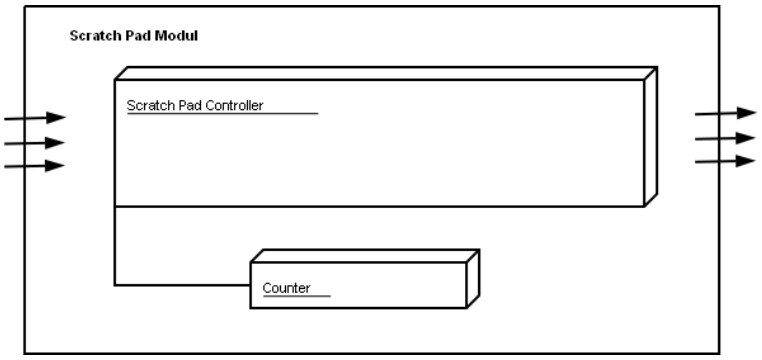


Abbildung 4.15: *Aufbau des Scratch Pad Moduls*

4.4.4 Realisierung

Scratch Pad Modul Aufbau

Das Scratch Pad Modul besteht aus einer zentralen Steuerungseinheit, dem Controller, und einer Hand voll Zählern. Die Funktionsweise des Controllers wird weiter unten beschrieben. Abbildung 4.15 verdeutlicht die Struktur dieses Moduls. Eine Aufzählung zusammen mit einer Beschreibung der Eingangs- und Ausgangsports enthält die Tabelle A.6.

Scrach Pad Controller

Das Verhalten des Controllers wird durch das StateChart in Abbildung 4.16 realisiert. Im Falle einer Schreib- oder Leseanfrage verlässt der Controller den *Idle* Zustand und wechselt in den Zustand *Read* bzw. in den Zustand *Write*. Hier wird dann die Dauer für einen Zugriff auf den Zähler hinzuaddiert, der die Zugriffsdauer für einen Zyklus verwaltet. Dieser wird im *Idle* Zustand wieder auf Null gesetzt. Im darauf folgenden Zustand, *Readcounter* oder *Writecounter*, wird der entsprechende Zugriffszähler erhöht. Der Zustand *NewSimTime* berechnet die neue Simulationszeit und leitet diese an das Ausgangsport *sramOut_simTime* weiter. Zu guter Letzt wird der Zähler für die gesamte Aktivitätsdauer aktualisiert und das erfolgreiche Simulationsende des Scracht Pad Moduls durch das Senden eines *Ack*, im Zustand *AckToBus*, signalisiert. Auch für dieses Modul ist ein *Reset* Zustand vorgesehen, der alle Zähler wieder zurücksetzt.

4.4.5 Konfigurieren des Moduls

Die Konfiguration des Scratch Pad Moduls wird über einen 64 Bit langen Bitstring, dessen Aufbau die Tabelle 4.5 beschreibt, realisiert.

Bezeichnung	Position	Belegung

Bezeichnung	Position	Belegung
Zugriffsdauer Read	0-31	Zugriffsdauer für einen Read-Zyklus
Zugriffsdauer Write	32-63	Zugriffsdauer für einen Write-Zyklus

Tabelle 4.5: *Aufbau des Konfigurationsstrings sramIn_config*

4.4.6 Auslesen der Ergebnisse

Die Simulationergebnisse der Zähler werden an das 192 Bit lange Ausgangsport *sram_simResult* geleitet. Der Aufbau dieses Bitstrings veranschaulicht die Tabelle 4.6.

Bezeichnung	Position
Read Counter	0-63
Write Counter	64-127
Sim Time Counter	128-191

Tabelle 4.6: *Aufbau des Ergebnisarrays*

4.5 Flash Modul

4.5.1 Allgemeine Informationen zu Flash Speichern

Der Hauptunterschied zu den bisher vorgestellten Speichermedien ist, dass der Flash Speicher Daten ohne eine permanente Stromversorgung speichern kann. Bei einem Flash-EEPROM werden einzelne Bits in den Floating Gate Schichten von Transistoren gespeichert, die mit Hilfe einer Oxid Isolierschicht von der Stromversorgung entkoppelt sind (vgl. Abbildung 4.17 [18]). So kann der Strom nicht abfließen und die gespeicherten Daten bleiben erhalten.

Um den isolierten Transistor zu beschreiben werden die Elektronen über eine hohe Versorgungsspannung zwischen der Source und Drain Schicht des Transistors stark beschleunigt. Über den so genannten Fowler-Nordheim-Tunneleffekt (ein quantenmechanischer Effekt benannt nach seinem Entdecker) können die Elektronen die isolierende Oxidschicht überwinden um die Floating Gate Schicht zu erreichen.

Bei den Flash Speichern gibt es zwei verschiedene Architekturen. Dazu gehören die NAND (Serienschaltung der Transistoren) und die NOR (Parallelschaltung der Transistoren) Flash Speicher. NAND Speicher werden häufig als kleine Massenspeicher mit geringer Zugriffsgeschwindigkeit benutzt. Diese Technologie findet sich z.B. in den aktuellen USB-Sticks wieder. Die NOR Technologie hingegen erreicht durch Ihre Parallelschaltung eine deutlich höhere Zugriffsgeschwindigkeit, der durch einen höheren Preis und mehr Fläche zu Buche schlägt.

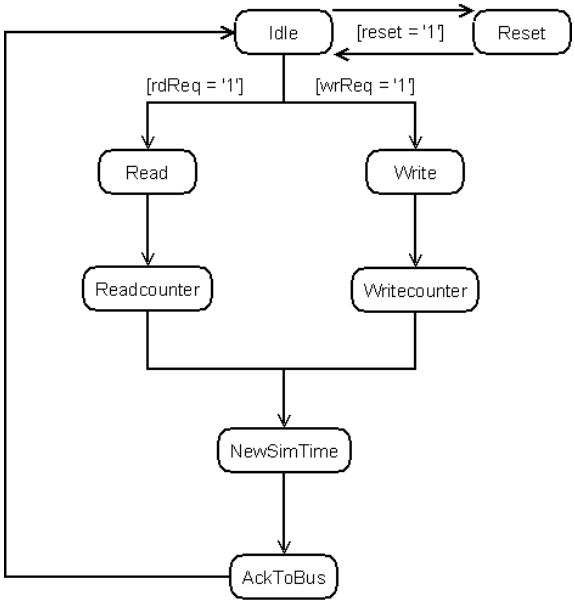


Abbildung 4.16: *StateChart Scratch Pad Control Modul*

Da die Isolierschicht bei dem weiter oben genannten Tunneleffekt nach und nach beschädigt wird, und der Transistor irgendwann direkt mit der Stromquelle verbunden wird, haben die Flash-Speicher eine maximale Schreib- und Löschanzahl. Diese liegt bei NOR Speichern bei ca. 10.000-100.000 Operationen und bei NAND Speichern bei ca. 1.0 00.000 Operationen. Um sicherzustellen, dass die begrenzten Schreib- und Löschvorgänge auch gewünscht sind, haben viele Flash Speicher einen Schreibschutz eingebaut. Dieser kann dadurch deaktiviert werden, dass eine vorher festgelegte Kombination aus Adress- und Datenbits an den Speicher angelegt wird.

4.5.2 Anforderungen

Die Anforderungen an das Flash Modul sind sehr gering und bestehen nur aus den folgenden beiden Punkten:

- Alle Lesezugriffe sollen mitgezählt werden.
- Die Zugriffsdauer, gemessen in der Anzahl benötigter Taktzyklen, soll protokolliert werden.

Es soll also ein Read-Only Flash Modul realisiert werden.

4.5.3 Ressourcen Bedarf

Der Ressourcenbedarf des Flash Moduls ist von den bisher realisierten Speichern am geringsten. Es werden lediglich 3 64-Bit Zähler benötigt. Einer misst die Zeit eines Lesezyklus, einer hält die Gesamtsimulationszeit aller Lesezyklen fest und ein letzter speichert die Anzahl an Leseanfragen.

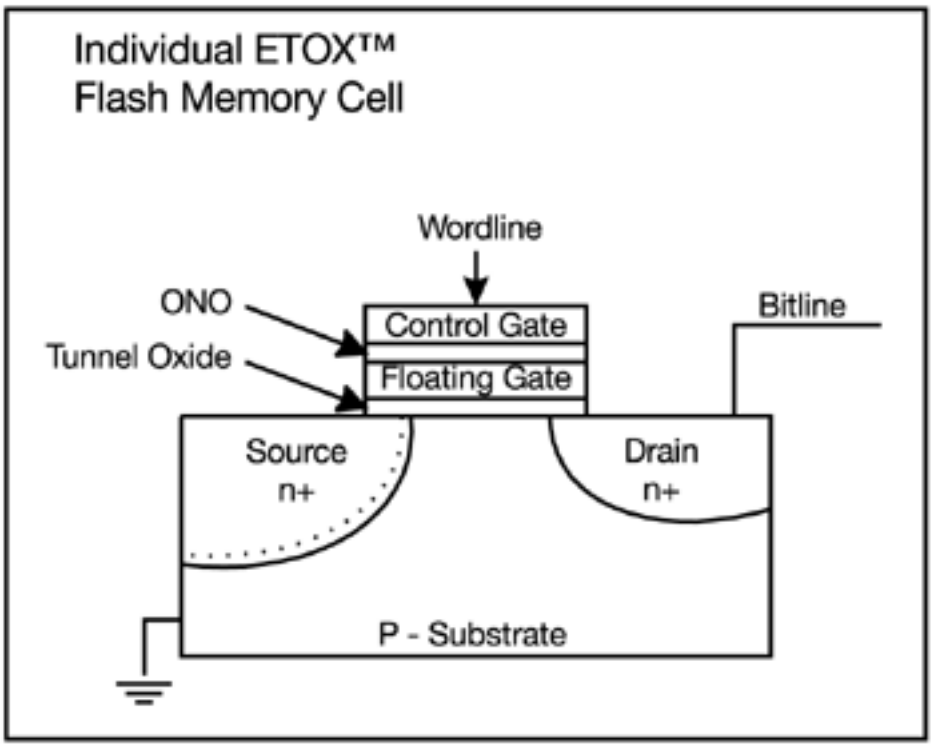


Abbildung 4.17: *Flash Speicher - Transistorzelle*

4.5.4 Realisierung

4.5.4.1 Flash Aufbau

Da das Flash Modul aus Transistoren besteht und daher eine relativ kleine Verwaltungslogik besitzt, besteht das Modul lediglich aus einem Controller, der diverse Zähler ansteuert, Abbildung 4.18. Die Portmap des Bausteins ist in Tabelle A.7 zu sehen.

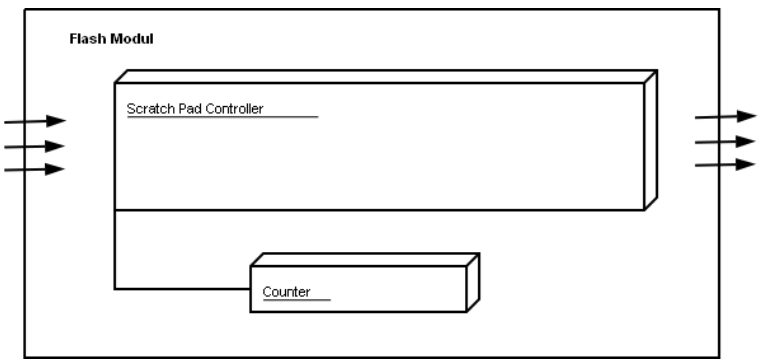


Abbildung 4.18: *Aufbau des Flash Moduls*

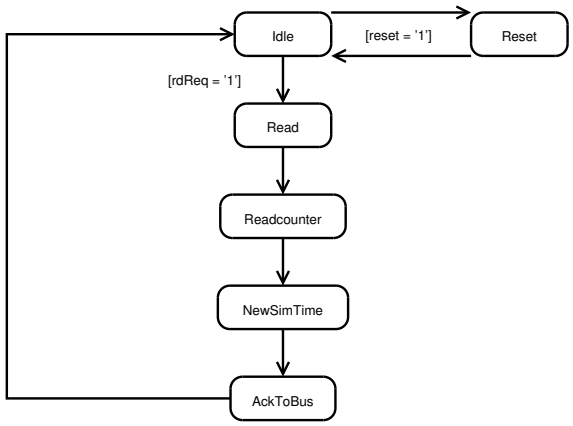


Abbildung 4.19: *Flash State Chart Diagramm*

4.5.4.2 Flash Controller

Das Verhalten des Flash Controllers ist in der Abbildung 4.19 dargestellt.

Der Controller startet im *Idle* Zustand und wartet auf eine eintreffende Leseanfrage. Sobald diese über das Signal rdReq eintrifft wechselt der Baustein in den Zustand *Read*. Dort wird der Zeitzykluszähler um die vorher konfigurierte Anzahl an Taktzyklen für eine Leseanfrage erhöht. Dieser Zykluszähler wird später vor dem erneuten Wechsel in den *Idle* Zustand auf den Gesamtzeitzähler aufaddiert.

Nach der Aktualisierung des Zählers wird in den *Readcounter* Zustand gewechselt, in dem der Zähler erhöht wird, der die Anzahl an Leseanfragen darstellt. Im Folgenden wechselt der Controller in den Zustand *NewSimTime*, in dem die neue Simulationszeit berechnet und auf den Bus gelegt wird. Über den Zustand *AckToBus* wechselt der Controller schließlich in *Idle* Zustand zurück. Wie bereits vorher angemerkt wird neben dem Ack, dass auf den Bus gelegt wird, ebenfalls der Zeitzykluszähler auf den Gesamtzeitzähler hinzu addiert.

Im *Idle* Zustand wartet das Modul anschließend auf die nächste Leseanfrage.

4.5.5 Konfigurieren des Moduls

Die Konfiguration des Scratch Pad Moduls besteht nur aus der Länge eines Lesezyklus. Daher ist die Konfigurationstabelle 4.7 sehr einfach.

Bezeichnung	Position	Belegung
Zugriffsdauer Read	0-31	Binärcodierung der Ticks eines Lesezyklus

Tabelle 4.7: *Aufbau des Konfigurationsstrings flashIn_config*

4.5.6 Auslesen der Ergebnisse

Die Simulationsergebnisse liegen am Port flashOut_simResult an. Dabei ist der Bitstring wie in Tabelle 4.8 zu interpretieren.

Bezeichnung	Position
Read Counter	0-63
Sim Time Counter	64-127

Tabelle 4.8: *Aufbau des Simulationsergebnisregisters flashOut_simResult*

4.6 Fazit

Bei der Realisierung der Module wurde Wert auf die wesentlichen Eigenschaften der Speichetypen gelegt, so dass aussagekräftige Ergebnisse zur Verfügung gestellt werden. Aus Zeitgründen mussten dennoch einige Einschränkungen bei der Realisierung gemacht werden. So wurden beispielsweise für das Cache Modul nicht alle Verdrängungsstrategien implementiert. Im DRAM Modul könnte man die Burstlängen konfigurierbar machen und die Möglichkeit bereitstellen die Burstfunktion zu deaktivieren. Die Einfachheit des Scratch Pad Moduls läst keine Luft für Verbesserungen wohingegen der Flashspeicher in unserer Simulation als *Nur-Lese-Speicher* realisiert ist. Zusammenfassend läst sich bezüglich der Speichermodule aber sagen, dass die Simulation das reale Verhalten der verschiedenen Speichertypen dennoch sehr gut widerspiegelt.

Kapitel 5

Linux

Wenn der Begriff Linux fällt, ist meist die Rede von einer kompletten Desktopumgebung, häufig als Alternative zu kommerziellen Produkten wie Microsoft Windows oder Apple Mac OS X. In diesem Zusammenhang konnte Linux in den letzten Jahren seinen Marktanteil erheblich vergrößern. Auch, dass Linux aus dem Serverbereich inzwischen nicht mehr wegzudenken ist, ist kein Geheimnis.

Es gibt jedoch noch eine weitere Art von Systemen, in denen Linux inzwischen immer häufiger anzuteffen ist: Eigebettete Systeme. Dies ist kein Wunder, schließlich ist Linux dank offenem Quellcode extrem anpassbar und lässt sich somit exakt auf die eingesetzte Hardware, die in diesem Bereich häufig eben nicht aus Standardkomponenten besteht, abstimmen. Zusätzlich fallen keine Lizenzgebühren an, so dass die Hemmschwelle zum Einsatz eines Linux - Betriebssystem im Embedded Bereich nicht sehr hoch ist.

Ein weiterer Vorteil des offenen Betriebssystems ist die große Anzahl verfügbarer Software für nahezu jeden Einsatzzweck. Hierdurch kann die Entwicklungszeit eines eingebetteten Systems drastisch verkürzt werden, da Anwendungen wie Beispielsweise ein Webserver nicht selbst entwickelt werden müssen.

Da mit Hilfe des V-POWER Systems verschiedene Speicherhierarchien in möglichst vielen Anwendungsfällen getestet werden sollen, war der Einsatz von Linux als Betriebssystem bereits Teil der Aufgabenstellung der Projektgruppe. Da jedoch noch keine Linux Distribution existiert, die sich direkt auf dem XUPV2P Board starten lässt und dessen Hardware (insbesondere auch den von uns benötigten Framebuffer) unterstützt, mussten hier einige Anpassungen vorgenommen werden, die im Folgenden beschrieben werden sollen.

Weiterhin musste im Betriebssystemkernel die Benutzung des eingebauten L1 Caches der PowerPC CPU deaktiviert werden, da die Speicherhierarchie komplett im FPGA simuliert werden soll und ein Hardwarecache diese Simulation unmöglich machen würde. Die hierfür erforderlichen Schritte sind ebenfalls in diesem Kapitel dokumentiert.

5.1 Portierung auf das XUP Development Board

Das quelloffene Betriebssystem Linux ist inzwischen auf unzählige Prozessoren portiert worden. Hierzu zählt auch der PowerPC, so dass für eine Portierung auf das XUPV2P Board lediglich die Treiber für die – meist im FPGA simulierte – Hardware benötigt werden. Diese stellt Xilinx glücklicherweise für das ML300 Board, das eine dem XUPV2P sehr ähnliche Konfiguration aufweist, zur Verfügung [19]. Somit beschänkten sich die nötigen Anpassungen auf einige kleine Konfigurationsänderungen, die in [20] bzw. [21] zu finden sind. Diese Anleitungen erklären allerdings nur den Aufbau eines Grundsystems mit Zugang über die serielle Schnittstelle. Genauere Anleitungen zum Einbinden von weiterer Hardware wie beispielsweise Keyboard, Maus oder Framebuffer waren im ersten Semester der Projektgruppe noch nirgendwo dokumentiert. Inzwischen findet sich unter [22] eine detaillierte Schritt für Schritt Anleitung zur Portierung von Linux auf das XUP Board. Dennoch musste der Treiber für den Framebuffer, welcher eine graphische Ausgabe über den VGA Port des Boards ermöglicht, angepasst werden, damit der Grafikspeicher eine feste Stelle im Systemspeicher einnimmt. Dies ist im Kapitel 5.3 beschrieben.

Die von der PG erstelle Version von Linux, also insbesondere den angepassten Kernel, die Treiber für den Memory Controller und die Zusammenstellung der Programme, wollen wir im Folgenden **V-POWER** Linux nennen.

Das Testsystem

Um Linux überhaupt auf dem Board starten zu können, muss zunächst im Base System Builder des Xilinx EDK eine Hardwarekonfiguration erstellt werden, die kompatibel zu den Treibern im Kernel ist. Im WWW finden sich hierfür sehr genaue Anleitungen (siehe z.B. [20]), in zahlreichen Versuchen stellte sich allerdings heraus, dass die meisten sinnvollen Konfiguration auch mit Linux funktionieren. Wichtig ist hierbei, dass eine serielle Schnittstelle vorhanden ist und natürlich, dass der DIMM Speicher eingebunden wird. Falls das System von Compact Flash Karte booten soll, muss weiterhin die SysAce Schnittstelle bereitstehen. Alle weitere Hardware ist optional, es ist jedoch darauf zu achten, dass für alle Komponenten, wo dies möglich ist, Interrupts aktiviert werden. All diese Schritte lassen sich durch den *New Project Wizard* des EDK leicht vornehmen. In der finalen Version des **V-POWER** Systems, wurde anschließend noch der DRAM Controller durch den **V-POWER** IP-Core ersetzt. Weitere Anpassungen sind nicht nötig, da der **V-POWER** Memory Controller für den PLB transparent ist, sich also wie ein normaler Speichercontroller verhält.

5.1.1 Toolchain

Eingebettete Systeme laufen meist auf einer völlig anderen Hardware als das System, auf dem sie entwickelt werden. Daher müssen auf dem Entwicklungssystem Compiler und Tools installiert

sein, die den Binärcode für die Zielarchitektur erstellen können. Diesen Vorgang nennt man Cross-Compiling. Für Linuxsysteme bringt die GNU Cross Platform Development Toolchain [23] alle Programme mit, die benötigt werden, um ausführbare Binaries für eine Hardwareplattform zu bauen, unter anderem den GCC Compiler und Linker.

Das Erstellen einer Cross-Toolchain ist ein recht aufwändiger Vorgang. Zunächst muss auf dem Host-System, das ist der Rechner, unter dem die Tools später laufen sollen, bereits eine native GNU-Toolchain installiert sein. Diese wird glücklicherweise durch alle modernen Linux Distributionen bereitgestellt. Im Folgenden werden kurz die Schritte erläutert, durch welche sich eine funktionierende Cross-Toolchain erstellen lässt. Der genaue Vorgang ist sehr ausführlich in [24] beschrieben.

1. **Kompilieren der GNU binutils für das Zielsystem**
   Die binutils enthalten eine Reihe von Programmen, die zur Bearbeitung von Binärdateien einer bestimmten Plattform benötigt werden, unter anderem den GNU Linker `ld`.

2. **Kompilieren einer eingeschränkten Version des GNU C Compilers**
   Da zu diesem Zeitpunkt die GNU C-Standardbibliothek glibc noch nicht für das Zielsystem vorliegt, kann zunächst lediglich eine sehr funktionsbeschränkte Version des Crosscompilers gebaut werden, die beispielsweise keine Shared Libraries unterstützt.

3. **Crosskompilieren der glibc**
   Nun kann mit Hilfe das in Schritt 2 erstellten Compilers die glibc übersetzt werden. Für diesen Vorgang werden auch die Headerdateien des Linuxkernels benötigt, da die glibc betriebssystemnahe Funktionalitäten wie beispielsweise Threads und Dateizugriff bereitstellt.

4. **Kompilieren der endgültigen Compiler**
   Da jetzt alle Funktionen der Standardbibliothek zur Verfügung stehen, kann nun die gesamte GNU Compiler Collection übersetzt werden.

Das Ergebnis dieses recht zeitaufwändigen Prozesses ist eine komplette Compiler Suite, sofern gewünscht inklusive des GNU C++ Compilers und der Unterstützung von dynamisch gelinkten Bibliotheken und Threads. Diese haben jeweils den Namen der entsprechenden GNU - Tools, allerdings mit der Bezeichnung der Plattform, für die sie erstellt wurden, vorangestellt, also zum Beispiel `powerpc-405-linux-gnu-gcc`.

Erheblich vereinfachen lassen sich all diese Schritte durch Dan Kegels CrossTool, das unter [25] frei verfügbar ist. Das Skript benötigt lediglich die Angabe der gewünschten Versionen von glibc und GCC und übernimmt dann selbstständig den Download und das Kompilieren der Quelldateien.

Für das Übersetzten des Kernels und der Userspace - Programme werden die Version 3.4.5 der GCC und die Version 2.3.6 der glibc verwendet.

5.1.2 Kernel

Im Zusammenhang mit Linux ist meist ein komplettes System, inklusive Anwendungsprogrammen, gemeint. Im eigentlichen Sinne steht das Wort Linux jedoch nur für den Betriebssystemkernel. Dieser umfasst lediglich die Komponenten, die für den Betrieb des Rechners zwingend notwendig sind und eine Umgebung schaffen, in der Anwendungsprogramme ausgeführt werden können. Diesen Programmen stellt er eine einheitliche Schnittstelle bereit, die unabhängig von der Rechnerarchitektur ist. Der Kernel umfasst bei Linux auch einen großen Satz von Treibern für unterschiedlichste Hardware.

Generell gibt es keine Unterschiede zwischen dem Linux Kernel in eingebetteten Systemen oder im Desktop-/Serverbereich. Auch wenn es Modifikationen gibt, die dem Betriebssystem beispielsweise Echtzeitfähigkeiten hinzufügen, so ist der Standardkernel durch seine Konfigurierbarkeit und Portierbarkeit sehr gut für den Einsatz auf beschränkter Hardware geeignet. Der Begriff "Embedded Linux" steht somit nicht für eine spezielle Linuxversion, sondern lediglich für ein Linuxsystem in einem speziellen Einsatzbereich.

5.1.2.1 Die PowerPC Version des Kernels

Die aktuelle Versionsnummer des Linux Kernels ist zum Zeitpunkt des Verfassens dieses Berichts ist 2.6.22 [26]. Besonders mit dem Versionssprung von 2.4 auf 2.6 wurden weitreichende Änderungen am Quellcode, den Treibern und dem Buildsystem vorgenommen. Daher sind leider die Treiber für die Hardwarekomponenten des XUPV2P Boards noch nicht alle für diese Version portiert worden und es existieren keine Anleitungen, wie man diese Version auf den PowerPC Prozessoren der Xilinx Boards startet. Aus diesem Grund basiert **V-POWER** Linux noch auf dem alten 2.4 - Zweig; genauer auf Version 2.4.26 mit einigen Patches von MontaVista [27], einer Firma, die den Linuxkernel für unterschiedliche Hardwareplattformen, darunter das ML300 Board von Xilinx, anpasst. Das ML300 ist auch mit einem Virtex 2 Pro Chip bestückt und bindet weitere Hardware über den Processor Local Bus ein. Daher werden diese Komponenten von dem MontaVista Kernel direkt unterstützt. Weiterhin bietet MontaVista komplette Linux - Distributionen an, die für ein bestimmtes Board angepasst sind und komplett mit Kernel und Programmen ausgeliefert wird. Vor allem aber umfassen diese Pakete Dokumentationen und Schritt-für-Schritt Anleitungen, welche die Entwicklung eigener Anwendungen erheblich vereinfachen. Leider steht der PG kein Budget zur Verfügung, das den Erwerb eines solchen Paketes erlaubt, dennoch gewährt MonaVista – den Regeln der GPL [28] entsprechend – Zugriff auf die Kernelquellen, so dass diese die Grundversion des Kernels bilden konnten.

5.1.2.2 Portierung des Linux Kernels auf das Xilinx XUPV2P Board

Der Kernel von MontaVista ist zwar bereit für den Einsatz auf PowerPC 405 Prozessoren wie dem des Virtex 2Pro und unterstützt auch den Processor Local Bus, die Hardwarekomponenten eines im EDK erstellten Systems unterstützt er aber noch nicht. Außerdem müssen dem System zur Lauf-

zeit die Adressen der verschiedenen Devices bekannt sein, denn Plug and Play gibt es auf dem Xilinx Board nicht. Dafür muss im EDK ein sogenanntes Board Support Package (BSP) erstellt werden. Dies umfasst Treiber für alle von Linux unterstützten Hardwarekomponenten und stellt deren Adressen in einer speziellen Headerdatei bereit.

Hierfür muss lediglich die Größe des von Linux zu verwendenden Speichers sowie die Geräte, die dem Kernel bekannt gemacht werden sollen, angegeben werden. Letztere sind im Fall von **V-POWER** Linux der Interrupt Controller, die serielle Schnittstelle, der Ethernet Adapter und das SysAce Interface, das den Zugriff auf die Compact Flash Karte unter Linux ermöglicht. Der Framebuffer wird in diesem Menü nicht gewählt, da er nicht wie empfohlen über den DCR (Device Control Register) Bus angeschlossen ist (siehe Kapitel 5.3). Diese Konfiguration ist von den Skripten, die das BSP generieren, nicht unterstützt und führt zu einem Absturz. Weiterhin lässt sich die Geschwindigkeit des Prozessors konfigurieren, hier wurde die maximal empfohlene Geschwindigkeit von 300 MHz gewählt.

All diese Parameter und die Adressen der gewählten Komponenten werden in einer Headerdatei des Kernels definiert. Zusammen mit den Treibern wird diese beim Generieren des Board Support Packages in einen Verzeichnisbaum kopiert, der anschließend in den des Kernels kopiert werden muss. Somit sollte theoretisch ein funktionierender Kernel gebaut werden können. In der Praxis mussten allerdings noch eine Reihe von Anpassungen gemacht werden.

Änderungen am gepatchten MontaVista Kernel

Das Kompilieren des durch die obigen Schritte erzeugten Kernels schlug leider an einigen Stellen fehl. Beispielsweise fehlte in einem Makefile die Angabe einer bestimmten Datei, die Adressen des PS/2 Controllers wurden nicht definiert, und im Ethernet Controller stimmten einige Funktionsnamen nicht mit den Bezeichnungen der exportierten Funktionen überein. Für einige diese Fehler existieren Patches, die unter [21] zur Verfügung gestellt werden, die restlichen ließen sich relativ leicht finden und beheben.

Außerdem wurde die Geschwindigkeit der seriellen Schnittstelle für Logausgaben von voreingestellten 9600 auf 38400 kbps erhöht. Da sowohl diese Kernelausgaben als auch die Konsole später über die RS232 Schnittstelle laufen sollen, beide aber an unterschiedlichen Stellen konfiguriert werden, muss für beide die selbe Geschwindigkeit eingestellt sein.

Schließlich muss der Kernel nur noch für das Crosskompilieren vorbereitet werden. Dafür werden im Haupt - Makefile der Sourcen die Variablen `ARCH` und `CROSS_COMPILE` gesetzt. Im Unterverzeichnis `/arch/`) der Kernelquellen befinden sich für alle unterstützen Architekturen die systemspezifischen Quelldateien jeweils in einem Unterverzeichnis mit dem Namen der Architektur. Für PowerPC ist dies `ppc`, daher ist dies der Wert, auf den die Variable `ARCH` gesetzt wird. Bei `CROSS_COMPILE` handelt es sich um den Präfix, der vor Befehle für Compiler, Linker und weitere Buildprogramme

gesetzt werden muss. Für die **V-POWER** Toolchain ist dies `powerpc-linux-gnu-`.

Konfiguration und Kompilation

Nun ist alles bereit, um den Kernel zu konfigurieren. Am einfachsten geht dies über das cursorgesteuerte Konfigurationsmenü, das sich durch den Befehl `make menuconfig` aufrufen lässt. Dort muss zunächst im Untermenü *Platform Support* der Prozessortyp auf *PowerPC 405* und der Architekturtyp (*Machine Type*) auf *Xilinx ML300* gestellt werden. Erst dadurch erfährt der Kernel, dass er auf einem Xilinx Board läuft, über den PLB angeschlossen ist, und welche Hardwarebefehle durch den Prozessor zur Verfügung stehen. Die meisten weiteren Konfigurationspunkte müssen deaktiviert werden, lediglich die Treiber, von denen bekannt ist, dass auch die entsprechende Hardware vorhanden ist, dürfen aktiviert werden, da sich der Kernel sonst nicht kompilieren lässt. Welche Komponenten aktiviert sein müssen, bzw. nicht aktiviert sein dürfen, ist in [21] aufgelistet. Nachdem die Konfiguration abgeschlossen ist, wird diese in der Datei `.config` im Kernelverzeichnis gespeichert.

Der Kompilationsprozess kann nun mit `make zImage` gestartet werden. Dies baut den Kernel, komprimiert ihn mittels gzip (daher das *z* im Befehl) und setzt ihm einen Bootloader voran, der sich um das Entpacken und Starten kümmert. Die entstandene Datei, `zImage.elf` kann nun durch ein Tool von Xilinx zusammen mit der Hardwaredefinition für das Board in eine `system.ace` Datei gepackt werden, die benötigt wird, um das XUP Board von Compact Flash zu booten.

Schließlich werden noch durch `make modules` die zur Laufzeit nachladbaren Kernelmodule gebaut. Diese Funktion wird oft genutzt, um die Größe des Kernel Binaries nicht durch selten oder nicht auf allen Maschinen benötigte Funktionen aufzublähen. Da wir jedoch nur eine fest definierte Systemumgebung haben und wenige Zusatzfunktionen (wie Unterstützung zahlreicher Dateisysteme oder Hardwaregeräte) benötigen wird diese Möglichkeit in der finalen Version von **V-POWER** Linux nicht genutzt, alle Funktionen werden direkt in den Kernel kompiliert. Die einzige Ausnahme bildet der **V-POWER** Treiber, der als Modul kompiliert wird, jedoch ein eigenes Build Environment besitzt (siehe 6.1).

5.1.3 Userland und Bootvorgang

Das Root Filesystem ist das Dateisystem, das der Kernel im Bootvorgang als erstes einbindet. Hier finden sich alle Dateien, die für den Systemstart benötigt werden, insbesondere der Binärcode für den `init`-Prozess, welcher nach dem Initialisieren der Treiber gestartet wird und den restlichen Bootvorgang steuert (siehe Seite 129).

Die Verzeichnisstruktur des Root Filesystems ist auf allen Linuxsystemen gleich (tatsächlich sogar bis auf kleine Feinheiten ebenfalls auf allen weiteren Unix-Varianten). Eine Beschreibung der Einzel-

heiten des Aufbaus dieses Dateisystems sowie der Bedeutung der Verzeichnisse und deren Inhalte würde an dieser Stelle den Umfang des Endberichtes übersteigen. Weitere Informationen zu diesem Thema finden sich in [24].

Bis auf den Mediaplayer `mplayer`, dessen Portierung im Anhang beschrieben ist (siehe D), werden alle weiteren Standardanwendungen in **V-POWER** Linux durch das Programm Busybox, das im Folgenden genauer beschrieben wird, bereitgestellt.

5.1.3.1 Busybox

Auch wenn sie im kompilierten Zustand lediglich aus einer Binärdatei besteht, ist die Busybox [29] eigentlich eine ganze Programmsuite. Das Projekt versucht, möglicht viele der aus einer Unix-Umgebung bekannten Tools in einer Datei zu verpacken und dabei den Speicherbedarf so gering wie möglich zu halten. Die BusyBox realisiert so zum Beispiel die Befehle `ls`, `kill`, `mount` oder `grep`. Dadurch, dass alle diese Befehle durch ein Programm bereitgestellt werden, kann ein großer Teil des Programmcodes mehrfach verwendet werden. Zusätzlich ist der Funktionsumfang gegenüber den richtigen Tools eingeschränkt. So ist eine typische BusyBox nur 260kb groß, während die vergleichbaren Programme in einem Unixsystem ca. 1,5MB belegen. In einem System mit nur 2 MB Speicher bedeutet dies einen großen Unterschied. Dieser Größenunterschied fällt natürlich auf den 128 MB großen Compact Flash Karten, die der PG zur Verfügung stehen, nicht sonderlich ins Gewicht. Die Busybox hat jedoch noch einen ganz anderen Vorteil: Statt zahlreiche einzelne Programme kompilieren zu müssen und dabei jeweils die Abhängigkeiten untereinander beachten zu müssen, lassen sich alle Applikationen, die zum Betrieb eines eingebetteten Systems nötig sind, durch dieses eine Paket bauen und installieren. Die Einschränkungen im Funktionsumfang der Busybox Befehle fallen dabei nicht sonderlich ins Gewicht, schließlich erwartet niemand in einem eingebetteten System die gesamte Funktionalität eines Linux-Desktops. Daher wird für **V-POWER** Linux die Busybox (Version 1.5) benutzt

5.1.3.2 Bootvorgang

Sind der Kernel und das Root - Filesystem erstellt, ist das System prinzipiell bereit für den ersten Bootvorgang. Dafür muss die Logik des XUPV2P Boards sowohl die Hardwarekonfiguration des FPGAs als auch den zu startenden Code kennen. Diese Informationen – also einerseits die vom EDK erzeugte download.bit Datei, welche die Systembeschreibung des FPGA enthält und andererseits der kompilierte Kernel in der Datei `zImage.elf` – werden durch ein im EDK enthaltenes Skript zu einer Datei namens `system.ace` zusammengefügt. Diese ist alles, was das Board benötigt, um eine Systemkonfiguration zu erstellen und den Code auf dem Prozessor zu starten. Diese Datei wird für **V-POWER** Linux auf einer Compact Flash Karte bereitgestellt.

Der genaue Prozess des Startens von Compact Flash Karten auf dem XUP - Board ist im Handbuch des XUPV2P Boards [30] beschrieben.

Wahlweise kann auf der Karte auch das Root - Filesystem untergebracht werden, so dass alle benötigten Daten bereit stehen und das System ohne Netzwerk- oder USB - Verbindung gestartet werden kann. Im Verlauf der Projektgruppe wurde jedoch ein anderer Weg gewählt und das Dateisystem beim Booten über NFS eingebunden, weil dies einen viel schnelleren Entwicklungsablauf gewährleistet.

Der Bootloader

Wie im Kapiel 5.1.2 bereits angesprochen, enthält die erzeugte zImage.elf am Anfang des Codes einen Bootloader, der den Speicher initialisiert und den Kernel startet. Der Code dieses Bootloaders findet sich für das **V-POWER** System im Kerneltree in der Datei `misc-embedded.c` im Verzeichnis `arch/ppc/boot/simple/`.

Der Kernel Bootloader ist realtiv simpel. Er erhält einige systemspezifische Adressen aus den Headerdateien des Board Support Packages und entpackt den komprimierten Code des Kernels in einen freien Speicherbereich. Danach springt er an diese Speicherstelle und überlässt dem Kernel somit die Kontrolle über das System.

Zusätzlich wurde der Bootloader für **V-POWER** Linux so modifiziert, dass er als erstes den Framebufferspeicher mit Einsen überschreibt, um den Bildschirm zum Startvorgang weiß zu färben. Dies ist nur möglich, weil der VGA Core einen festen Speicherbereich nutzt (wie in Kapitel 5.3 beschrieben). An dieser Stelle ist auch die Anzeige eines **V-POWER** Startup - Logos denkbar, was allerdings nicht umgesetzt wurde.

Booten des Kernels

Der Kernel enthält vom Bootloader einige Variablen, die sich auf den meisten anderen Linuxsystemen direkt im Bootloader eingeben lassen. Für unsere Version müssen diese Parameter zur Compilezeit in der Kernelkonfigurationvariable `CONFIG_CMDLINE` gesetzt sein. Mit Hilfe dieser Variable wird im **V-POWER** System konfiguriert, ob das Dateisystem über NFS oder Compact Flash eingebunden werden soll.

Nach dem Einlesen der Bootparameter beginnt der Kernel, die Hardware zu initialisieren; zunächst den Interrupt Controller, danach alle weiteren Komponenten, die für die Verwendung in Linux ausgewählt wurden. Bis zu dieser Stelle agiert der Kernel ausschließlich im Kernelspace, das heißt mit vollem Zugriff auf alle Speicheradressen und Geräte. Nun, da alle Treiber eingerichtet sind, kann erstmals in den Userspace, in dem jeder Prozess lediglich seinen eigenen virtuellen Addressraum sieht und mit der Hardware nur über Treiber kommuniziert werden kann, gewechselt werden.

init

Das init - Programm ist schließlich der erste Prozess, der von Kernel gestartet wird. Dieses Programm ist somit auch der erste Code, der im Userspace läuft. Es wird, wie alle weiteren Systemprogramme, durch die Busybox realisiert (vgl. Seite 129) und liest beim Starten die Datei `/etc/inittab`, in der für verschiedene Systemereignisse Programme bestimmt sind, die bei deren Eintreten ausgeführt werden. Diese Datei sorgt beispielsweise dafür, dass dem Benutzer nach dem vollständigen Hochfahren des Systems ein Login Bildschirm angezeigt wird oder beim Druck der Tastenkombination *Ctrl-Alt-Del* das Betriebssystem neu gestartet wird. Außerdem wird hier das Skript `/etc/init.d/rcS` aufgerufen, die beim Systemstart das System vorbereitet (zum Beispiel das Einbinden des Kerneldateisystems `/proc` oder das Konfigurieren der Netzwerkschnittstelle).

Am Ende all dieser Schritte steht ein vollständig hochgefahrenes Linuxsystem bereit, das sowohl auf der seriellen Konsole als auch über Tastatur auf das Login seiner Nutzer wartet.

5.2 Caches

Der Cache des PowerPC auf dem XUP-2 ist für die Logik des Speicherkontrollers der Projektgruppe transparent, es besteht keine Möglichkeit zwischen dem Prozessor und seinem Cache Informationen abzugreifen. Der Speicherkontroller benötigt diese Informationen aber für das Profiling zwingend. Jede Operation und jedes Datum, das aus dem Cache geladen wird, könnte sonst nicht registriert und gezählt werden. Im Extremfall würde dieser Umstand dazu führen, dass bei Ablauf eines Programms nur einige, wenige Zugriffe auf den Hauptspeicher registriert, aber beliebig viele weitere Operationen ausgeführt werden, z.B. durch eine Schleife, die komplett im Cache liegt. Der Cache muss also deaktiviert werden.

5.2.1 Grundlagen MMU auf PPC

Der PowerPC 405 verfügt über eine Speicherverwaltungseinheit (MMU), welche virtuelle in reale Adressen umrechnet. Diese MMU dient dazu, Anwendungen stets den gesamten theoretisch verfügbaren Adressraum zur Verfügung zu stellen, auch wenn dieser real gar nicht vorhanden ist (z.B. weil der Hauptspeicher zu klein ist). Die Tätigkeit der MMU wird Paging genannt, das Paging stellt also den eigentlichen Vorgang der Umrechnung dar. Beim Paging wird der virtuelle Speicher in Teile gleicher Größe eingeteilt - diese Teile werden Seiten (pages) genannt - und die Verwaltung der Seiten in einer Seitentabelle vorgenommen.

Die Seitentabelle enthält neben der Information welcher virtuelle Speicher wo im realen Speicher untergebracht ist auch noch weitere Attribute, die Eigenschaften dieses Speicherbereiches kennzeichnen (z.B. ob der Speicherbereich vom Cache abgedeckt wird oder nicht).

Ist also ein Zugriff auf den realen Speicher nötig, bekommt die MMU diesen vorgelegt, sieht in der

Seitentabelle nach, wo der Speicher liegt und legt dann die eigentliche physikalische Adresse auf dem Adressbus an. Der Zugriff auf die reale Adresse erfolgt für die Prozesse transparent.

5.2.2 Real Mode Patch

Der PPC 405 besitzt spezielle Konfigurationsregister (special register). Einige dieser Register sind extra dafür vorgesehen, den Cache des Prozessors zu konfigurieren, wenn sich dieser im Realmode befindet. Das ICCR Register dient dazu, den Instruktionscache an- bzw. abzuschalten. Das DCCR Register wird verwendet, um den Adressbereich einzustellen, der gecached wird. In unserem Fall liegt der Bereich von 0x00000000-0x00000000, das bedeutet der Cache wird gar nicht benutzt. Der Assemblerbefehl mtspr (move to special register) schreibt einen Wert in ein Konfigurationsregister.

Konkret implementiert wird diese Methode geschickterweise dort, wo die übrigen Einstellungen der Konfigurationsregister des PPC vorgenommen werden. Dies wird in der Datei

`kernelarchppcmm4xx_mmu.c`

erledigt.

Am Ende dieser Datei fügen wir folgende Zeilen ein:

```
//DISABLE CACHE FOR VPOWER LINUX
mtspr(SPRN_DCCR, 0x00000000);   /* 0 MB of data space at 0x0. */
mtspr(SPRN_ICCR, 0x00000000);   /* 0 MB of instr. space at 0x0. */
```

In beiden Fällen kommt ein C-Macro zum Einsatz, das den Assemblerbefehl mtspr aufruft.

Diese Möglichkeit ist leider nur für Programme geeignet, die im Real Mode laufen. Für den Linux Kernel wirken sich diese Einstellungen nicht aus, da der Kernel nicht im Real Mode läuft, sondern im sogenannten Virtual Real Mode.

5.2.3 Virtual Mode Patch

Die einzige Möglichkeit in diesem Modus den Cache zu deaktivieren, besteht darin, jede neu angelegte Page als non-cacheable zu markieren. Dazu wird das Verwaltungsregister der MMU für diese Page Bit 29 auf 1 gesetzt. Im Kernel sind bereits vorgefertigte Defines für verschiedene Page Arten vorgegeben, welche in der Datei includeasm-ppcpgtable.h einzusehen sind. Für uns ist das Define _PAGE_NO_CACHE wichtig. Wann immer in kernelarchppcmmpgtable.c eine neue Page angelegt wird, ist dafür zu sorgen, dass diese mit _PAGE_NO_CACHE verodert wird. Der Kernel muss nach dieser Änderung komplett neu übersetzt werden.

Diese Möglichkeit funktioniert auch mit unserem System, der Cache ist also wirksam deaktiviert. Allerdings gibt es spezielle Assembler Befehle, die zu Optimierungszwecken verwendet werden und bestimmte Einträge gezielt aus dem Cache verdrängen, bzw. den Inhalt ersetzen können. Ein solcher Befehl ist z.B. dcbz (Data Cache Block Set to Zero), der einen bestimmten Cache-Block mit 0 überschreibt. Wird dieser Befehl aufgerufen, obwohl der Cache abgeschaltet ist, landet der Prozessor in einem undefinierten Zustand - ein Absturz des Systems ist die Folge. Es ist also dafür Sorge zu tragen, dass diese Befehle nicht zum Einsatz kommen.

Die glibc verwendet dcbz in einer Inline-Assembly Datei, um die Funktion memset() zu optimieren. Memset schreibt Nullen in Speicherbereiche. Die Optimierung besteht nun darin, dass die Nullen zunächst in den Cache geschrieben werden, was sehr viel schneller geht, und automatisch, wenn der entsprechende Eintrag aus dem Cache verdrängt wird, zurück in den Speicher geschrieben werden. Solange Nullen aber noch im Cache vorgehalten werden, ist der Inhalt des Speichers an der Stelle, an die die Nullen geschrieben werden sollen, aber ohne Bedeutung, da nur bei einem Cache-Miss auf den Speicher zugriffen wird.

Wie erwartet läuft das System nach Entfernen der optimierten Variante von Memset, allerdings auch so langsam wie erwartet. Es stellt sich zusätzlich ein sehr starkes Flimmern der grafischen Ausgabe ein, wenn auf den Speicher zugegriffen wird. Dies liegt mit hoher Wahrscheinlichkeit an dem nun nicht mehr vorhandenen Instruktionscache. Anstatt häufig wiederkehrende Instruktionen aus dem Cache direkt laden zu können, ist nun der sehr viel langsamere Weg über den Hauptspeicher nötig. Dadurch wird sehr viel häufiger auf den Dimm-Riegel zugegriffen und es treten deutlich mehr konkurrierende Lese- und Schreibzugriffe auf, weil normale Zugriffe der Programme und der Videohardware unverändert vorhanden sind. Das Flimmern ist so zu erklären, dass Zugriffe auf den Grafikspeicher, welcher im Hauptspeicher liegt, nun viel länger dauern.

5.3 Framebuffer und VGA Ausgabe

5.3.1 Xilinx VGA IP-Core

Eines der Ziele der Projektgruppe war eine Präsentation auf dem Campusfest 2007. Auf Systemen mit unterschiedlichen Speicherhierarchien wurde jeweils das gleiche Video abgespielt. Der Geschwindigkeitsunterschied bei der Wiedergabe sollte den Einfluss der Speicherhierarchie auf die Geschwindigkeit des Gesamtsystems auch Laien anschaulich machen. Um eine solche Videopräsentation zu ermöglichen, muss die Videoausgabe sowohl Hardware- wie Softwareseitig unterstützt werden. Dies ist im Falle des Xilinx XUP Virtex 2 Pro Boards mittels des FPGAs und eines nachgeschalteten RAMDACs (Random Access Memory Digital to Analog Converter, ermöglicht die Generierung von Analogsignalen aus im Speicher gelesenen Werten) möglich. Nötig dazu ist ein IP-Core für den FPGA, welcher fertig von Xilinx implementiert ist, sowie eine Software-Schicht zwischen diesem zum Betriebssystem hin. Auf beide Komponenten wird in den nachfolgenden Abschnitten näher eingegangen und die Eigenheiten derer sowie eventuelle Modifikationen dieser

näher erläutert.

5.3.1.1 Eigenschaften des IP-Cores

Der VGA IP-Core von Xilinx ermöglicht die Generierung eines VGA Signals an der VGA Buchse des XUP Boards. Die Auflösung des Signals ist fest auf 640x480x24Bit eingestellt und kann nicht ohne erhebliche Modifikation der Verilog Quelltexte geändert werden, da diese fest auf eine horizontale sowie vertikale Frequenz hin programmiert wurden.
Um nun ein Bild darstellen zu können, wird dem IP-Core die Adresse des Speicherbereiches übergeben, welcher die Bilddaten enthält. Dabei gilt zu beachten, dass diese Bilddaten im Speicher nur an 2-Megabyte-Grenzen liegen dürfen. Dies ist darin begründet, dass man für 24 Bit an Farbinformation 32 Bit Speicherplatz benötigt, wenn man keine Probleme mit dem Alignment haben möchte. Des Weiteren wären mehr als eine Bustransaktion nötig, um solche misaligned Daten lesen zu können, da der Adress- sowie Datenbus 32 Bit breit sind. Dies zusammen mit der Tatsache dass der IP-Core pro Zeile 1024 Pixel allokiert, aber nur 640 davon nutzt und 480 solcher Zeilen anzeigt, bringt uns nahe an die 2 Megabyte. Das ist dann auch die Größe jedes Bildes, dass durch den IP-Core ausgegeben werden soll. Diese Eigenschaft spiegelt sich auch in der internen Abbildung der Bildschirmspeicheradresse wider. Es werden nur die 11 höherwertigen Bits einer übergebenen Adresse gespeichert, wodurch der 32 Bit Addressraum in 2 Megabyte Segmente unterteilt wird. Im Base System Builder ist dafür auch nur ein 11 Bit Feld zur Angabe einer festen Bildschirmadresse vorgesehen, falls diese nicht dynamisch übergeben werden soll.

5.3.1.2 Einbindung des IP-Cores

Es gibt zwei Möglichkeiten, den VGA IP-Core in ein Hardware System einzubinden. Beide unterscheiden sich nur darin, ob die Device Control Register (DCR) Bridge und damit der DCR Bus mit dem IP-Core verbunden wird, oder nicht.

Ohne DCR Bridge

Um den VGA IP-Core in ein Hardwaresystem ohne Device Control Register (DCR) Bus einzubinden, muss der IP-Core zum System hinzugefügt werden und mit dem Processor Local Bus (PLB) verbunden werden. Zusätzlich muss der SYS_tftclk des VGA IP-Cores mit sys_clk_s verbunden werden, um den Takt für den VGA IP-Core vorzugeben. Dies ist auch die Ausgangssituation, wenn mittels Wizard ein System zusammengestellt wird, welches den Framebuffer enthält.
Der Nachteil dieser Variante ist, dass ohne den DCR Bus und Bridge keine Registerunterstützung durch den VGA IP-Core besteht.

Mit DCR Bridge

In dieser Variante muss sowohl die DCR Bridge als auch der DCR Bus zusammen mit dem VGA IP-Core zum Hardwaresystem hinzugefügt werden. Nun wird der PLB Bus sowie der DCR Bus mit dem VGA IP-Core verbunden. Zusätzlich ist die OPB2DCR Bridge nötig, diese wird mit dem OPB sowie DCR Bus verbunden. Als letztes wird der SYS_tftclk des VGA IP-Cores mit sys_clk_s verbunden werden.

Das Einbinden mit DCR Bus hat den Vorteil, dass interne Werte eines IP-Cores mittels Register von aussen lesbar sowie modifizierbar sind (falls der IP-Core dies unterstützt). Im Fall des VGA IP-Cores werden zwei Register zur Verfügung gestellt. Deren Basisadressen sind wegen der Implementation über die OPB2DCR Bridge von deren Basisadresse sowie den Adressen des VGA IP-Cores innerhalb des DCR Busses abhängig.

- **Adressregister des Bildschirmspeichers**
  *Basisadresse: opb2dcr_bridge_0 BasisAdresse + (VGA_FrameBuffer DCR_BaseAddr * 4)*

  Über dieses Register kann dem VGA IP-Core die Basisadresse eines Speicherbereichs angegeben werden, welcher ab dann als Bildschirmspeicher verwendet wird. Durch diese Möglichkeit kann z.B. Doublebuffering implementiert werden, oder eine Slideshow, die mehrere Bilder in den Speicher vorlädt und den Zeiger auf die Bilder mittels Register ändert. Dies wurde in einer Xilinx Demo verwirklicht, auf die noch im Unterpunkt Testprogramme eingegangen wird.

- **Kontrollregister der Bildschirmanzeige**
  *Basisadresse: opb2dcr_bridge_0 BasisAdresse + (VGA_FrameBuffer DCR_HighAddr * 4)*

  Dieses Register ermöglicht das Ein- sowie Ausschalten der Bildschirmanzeige. Wird eine 0 in das Register geladen, so wird der Bildschirm schwarz geschaltet. Eine 1 veranlasst den VGA IP-Core, wieder das Bild im Bildschirmspeicher anzuzeigen.

Es wurde dann im Nachhinein entschieden, den VGA IP-Core, trotz der Nachteile durch weniger Flexibilität bei Verlust der Kontrollregister, ohne DCR Bridge an das Hardwaresystem anzubinden. Diese Entscheidung lag darin begründet, dass unter Linux kein Zugriff auf die Register möglich war, trotz ioremap() auf die jeweiligen Adressen. Es trat stets eine Speicherzugriffsverletzung auf, wenn sowohl auf die reale als auch auf die virtuelle Adresse eines Registers zugegriffen wurde. Dadurch war keine vernünftige Bildschirmausgabe möglich, da standardmässig bei einem Reset die Adresse 0 als Basisadresse des Bildschirmspeichers feststeht, bis ein anderer Wert in das Adressregister geladen wird. Somit konnte man zwar dem Linux Kernel beim Entpacken zusehen, es war danach aber nicht mehr möglich, auf den als Bildschirmspeicher vorgesehenen Speicherbereich umzuschalten.

5.3.1.3 Testprogramme

Es wurden im Zuge der Einarbeitungsphase und während der Anpassung von Linux mehrere Testprogramme geschrieben, um die VGA Ausgabe zu testen, sowie sich mit der Programmierung der Hardware vertraut zu machen. Die Testprogramme basieren auf der Slideshow Demo von Xilinx, welche Bilder im BMP Format von einer CompactFlash Karte laden soll, um diese nacheinander auf dem Bildschirm anzuzeigen.

Xilinx Slideshow Demo

Die Slideshow Demo funktioniert, indem sie alle Bilder lädt und hintereinander im Speicher ablegt. Diese werden dann durch Setzen des Adresszeigers auf die Anfangsadresse eines jeden Bildes in das Adressregister des IP-Core auf dem Bildschirm angezeigt.

Leider scheint die Xilinx Slideshow Demo einen Fehler zu enthalten, bzw. die Implementierung der Ladefunktionen von der CompactFlash Karte. So bricht diese nach 17 Zeilen eines jeden Bildes ab. Durch eine zusätzlich eingebaute Debug-Ausgabe wurde die Leseroutine als Fehlergrund identifiziert.

Um trotzdem sicherzugehen, dass das Scheitern der Demo an zu großen Bilddateien liegt, wurde ein 16 Zeilen großes Bild in MS Paint erstellt und hiermit die Slideshow Demo getestet. Nachdem dieses Bild ohne die vorhergehenden Probleme am Bildschirm angezeigt wurde, wurde eine Routine in die Demo eingebaut, um die zusätzlichen Bildzeilen mit Zufallsfarben zu füllen. Die Zufallsfarben wurden durch die Random() Funktion in math.h generiert. Da dies zufriedenstellend funktionierte, wurde das so modifiziertes Programm um eine Routine erweitert, die Zufallsfarben-Bilder generiert und abwechselnd anzeigt, wodurch sich ein „bewegtes“ Bild einstellt.

Performance-Testprogramm

Nun galt es herauszufinden, wie viele Bilder pro Sekunde von der Hardware anzeigbar sind. Dazu war die Zufallsfarben-Routine aber nicht schnell genug. Es wurde entschieden, die Routine zu modifizieren und nur eine Farbe pro Bild statt Pixel zu generieren. Somit wurde der Einfluss der Verzögerungen durch den Prozessor selbst minimiert. Der Bildschirmspeicher wurde anschliessend mit der so generierten Farbe gefüllt. Es stellte sich heraus, dass mindestens 30 fps möglich sind.

Zwischenzeitlich wurde außerdem untersucht, ob der Bildschirmspeicher auch direkt mittels einer Zeigervariable modifizierbar ist, ohne dass Probleme auftreten. Diese Zeigervariable sollte auf eine Pixeladresse zeigen und durch die Dereferenzierung des Zeigers an der Adresse einen neuen Wert setzen. Es stellte sich als möglich heraus, und sogar ein wenig performanter als die bereitgestellten

Xilinx Methoden. Dies ist allerdings mit hoher Wahrscheinlichkeit auf eine Synchronisation dieser Methoden mit dem Bus oder sogar Vertikalimpuls des Bildes zurückzuführen. Es macht deswegen durchaus Sinn, diesen Methoden den Vorzug zu geben, um ein stabileres, flackerfreies Bild zu gewährleisten.

Der Quelltext zu diesem Testprogramm ist im Anhang C als Quelltext C.2 zu finden.

5.3.2 Treiberanpassung

Der Treiber von Xilinx für den VGA IP-Core wurde im Zuge der Projektgruppe modifiziert. Zum einen wurde die Unterstützung mehrerer IP-Cores entfernt, da das XUP Board nur einen VGA Ausgang besitzt und es deshalb nicht möglich ist, diese Fähigkeit auszuschöpfen. Es wurde zum anderen durch Wegfall der DCR Bridge die Anfangsadresse des Bildschirmspeichers mittels eines #define im Buildprozess und im Base System Builder festgelegt. Dies war, wie schon erwähnt, nötig, da ein Zugriff auf die Register des VGA IP-Cores von Linux aus nicht möglich ist. Es wurde stattdessen entschieden, das RAM für Linux auf 64 Megabyte zu limitieren und unterhalb dessen den Grafikspeicher anzusiedeln. Die Adresse ist fest in der Hardware eingestellt.

5.4 Fazit

Die Portierung von Linux auf das XUPV2P Board war bereits nach dem ersten Semester der Projektgruppe abgeschlossen, und auch alle benötigten Programme standen zu diesem Zeitpunkt schon bereit bereit. Im zweiten Semester wurden daher in diesem Bereich keine großen Änderungen gemacht.

Auch der MPlayer steht in der gewünschten Funktion bereit, bei abgeschaltetem Cache ist eine ruckelfreie Wiedergabe damit aber leider nicht möglich. Dennoch beweist die erfolgreiche Portierung, dass prinzipiell die meisste Software, die auf anderen Linux Systemen lauffähig ist, auch unter **V-POWER** Linux läuft.

Kapitel 6

Treiber und Konfiguration

Damit die V-POWER Hardware überhaupt sinnvoll genutzt werden kann, muss es für den Nutzer eine Möglichkeit geben, eine Speicherkonfiguration in das System zu spielen, die Simulation zu kontrollieren und deren Ergebnisse auszulesen. All diese Funktionen sollen ohne Neustart des Systems direkt per Software erreichbar sein, so dass unterschiedliche Konfigurationen schnell getestet und verglichen werden können.

Hierzu muss direkt mit der Systemhardware kommuniziert werden, wozu unter Linux ein Kerneltreiber erforderlich ist. Der Aufbau dieses Treibers und seine Schnittstelle zum Benutzer, bzw. anderen Programmen wird in diesem Kapitel erläutert.

Zum einfachen Editieren der Hardwareeinstellungen wurde auch ein Konfigurationsprogramm geschrieben, mit welchem es möglich ist mittels Drag&Drop verschiedene Speicherhierarchien effizient zu erstellen und abzuändern.

6.1 Das V-Power Kernelmodul

Die Kernkomponente zur Konfiguration des V-POWER Systems bildet das Kernelmodul. Es kommuniziert direkt mit der Hardware und bietet eine für Anwender und andere Programme nutzbare Schnittstelle zu deren Bedienung. In diesem Abschnitt soll das Kernelmodul beschrieben und seine Benutzung dokumentiert werden. Dazu werden zunächst kurz einige grundlegende Informationen über Linux Kerneltreiber gegeben, bevor dann speziell auf den V-POWER Treiber eingegangen wird.

6.1.1 Linux Treiber

Wie unter jedem modernen Betriebssystem laufen unter Linux Programme normalerweise in einem stark eingeschränkten Betriebsmodus, der keinen direkten Zugriff auf die Systemhardware bietet. Dies ist zwingend notwendig, um ein sicheres und stabiles Multibenutzersystem zu schaffen,

verkompliziert allerdings auch einige ansonsten sehr simple Vorgänge. So zum Beispiel das Konfigurieren der **V-POWER** Hardware. Anstatt eine Konfiguration direkt in den Konfigurationsspeicher zu schreiben, muss ein Prozess unter Linux immer den Umweg über den Kernel nehmen, da er lediglich seinen virtuellen Adressraum sehen kann. Daher muss die gesamte Grundfunktionalität für Konfiguration, Simulationssteuerung und -auswertung als Treiber direkt im Betriebssystemkernel implementiert sein.

Hierfür gibt es unter Linux zwei verschiedene Wege. Zum einen lassen sich Gerätetreiber direkt in den Kernel kompilieren, sie gehören dann fest zum Betriebssystem und werden beim Starten mit in den Speicher geladen. Zum anderen gibt es sogenannte Kernelmodule. Diese können, bis auf wenige Ausnahmen, den gleichen Funktionsumfang bieten, werden aber erst bei Bedarf geladen. Dies hat einige Vorteile, vor allem in der Entwicklung, da ein Module während der Laufzeit beliebig oft neu geladen werden kann und so die Änderung-Kompilierung-Test-Zyklen weitaus kürzer sind. Dies ist der Grund, weshalb für den **V-POWER** Treiber dieser Ansatz gewählt wurde.

6.1.1.1 Funktionsweise eines Kernelmoduls

Ein Linux Kernelmodul kann während der Laufzeit des Systems aus dem Userspace vom Administratoraccount *root* mit Hilfe der Befehle `insmod` oder `modprobe` geladen werden, wobei letzterer Abhängigkeiten zu anderen Modulen automatisch auflöst, was für den **V-POWER** Treiber jedoch nicht benötigt wird. Nun wird zuerst eine Initialisierungsfunktion des Moduls aufgerufen, in der je nach Funktionalität Speicher allokiert oder die zugehörige Hardware initialisiert wird. Das Kernelmodul hat hierbei Zugriff auf alle globalen Funkionen und Variablen im Linux Kernel und kann auf beliebige Speicheradressen zugreifen. Daraus wird sofort ersichtlich, dass fehlerhafter Code sehr schnell zum Absturz des gesamten Systems führen kann. Weiterhin impliziert dies, dass im Kernelspace keine Bibliotheken genutzt werden können. An Hilfsfunktionen stehen also nur solche bereit, die Teil des Linux-Quelltextes sind (so zum Beispiel `printk`, das einen Ersatz für das aus dem Userspace bekannte `printf` darstellt und so Informationen in das Kernel Log schreiben kann). Alle Symbole werden erst beim Einbinden des Moduls gelinkt, was den Nebeneffekt hat, dass eventuelle Fehler beim Kompilieren gar nicht auffallen. Nach dem erfolgreichen Laden verbleibt das Modul im Speicher und wartet auf Aufrufe seiner Funktionen, die meist beim Initialisieren als Callbacks für bestimmte Ereignisse registriert werden, bis es schließlich mit dem Kommando `rmmod` oder beim Herunterfahren des Systems wieder entladen wird.

Weitere Informationen über Linux Kernelmodule finden sich in [31], [32] und [33], die auch bei der Entwicklung des **V-POWER** Moduls eine große Hilfe waren.

6.1.2 Aufbau des Quelltextes

Aufgrund der oben erwähnten Unterschiede des von Kernelspace zu Userspace Code müssen eine Reihe von Vorraussetzungen erfüllt sein, damit das Kompilieren des **V-POWER** Treibers funktioniert.

Zunächst muss statt dem normalen GCC des Hostsystems der PowerPC Crosscompiler (siehe Kapitel 5.1.1) genutzt werden. Zusätzlich muss sich das `include` Verzeichnis des Kernels im Suchpfad für Headerdateien befinden und diverse Konstanten definiert werden, damit der Treiber als Modul gebaut und so auch hinterher in den Kernel eingebunden werden kann. Dies geschieht vollständig innerhalb des Makefiles, so dass ein Aufruf von `make` im Verzeichnis des Treibers das Modul und ein Testprogramm erstellt. In der Datei lassen sich über Variablen der Pfad zu den Kernelquellen, das Zielverzeichnis für die Installation via `make install`, die Version des Treibers und ein Flag, das erweiterte Debugausgaben ermöglicht, setzen.

Ein Aufruf von `make doc` erstellt mit Hilfe des Tools doxygen [34] eine HTML Dokumentation aller Komponenten des Treibers, in der sich teilweise ausführlichere Informationen finden als in diesem Bericht.

6.1.2.1 Konfiguration von Adressen und anderen Hardwarewerten

Einige Werte müssen in den VHDL Quelltexten der Hardware und dem Kerneltreiber übereinstimmen, so zum Beispiel die Anzahl der verfügbaren Module pro Speichertyp, die Adressen für Konfiguration und Simulationsergebnisse oder die möglichen Befehle für das Befehlsregister der **V-POWER** Bus Bridge (siehe 3.1). Zunächst wurde geplant, diese Werte mit Hilfe eines Skriptes aus den VHDL Quelltexten zu extrahieren und daraus eine C-Headerdatei zu generieren. Dieses Vorhaben erwies sich als zu aufwändig, so dass nun die entsprechende Werte als Konstanten in der Datei `vpowerhw.h` definiert werden müssen. Alle Einträge der Datei sind jedoch ausführlich dokumentiert, bei einer Hardwareänderung kann so schnell der Treiber angepasst werden.

6.1.2.2 Das Testprogramm

Zum Testen des **V-POWER** Treibers existiert ein einfaches Programm, das ebenfalls durch `make` erstellt wird. Dieses Programm dient dazu, aus dem Userspace alle grundlegenden Funktionen des Treibers zu testen. Es kann auf dem Zielsystem durch den Aufruf von `testvpower` gestartet werden und bestätigt bei korrektem Durchlauf ein funktionsfähiges **V-POWER** System.

6.1.3 Funktionen des V-Power Kernelmoduls

Das Kernelmodul des **V-POWER** Treibers übernimmt sämtliche Kommunikation zwischen der Hardware des Speichersimulators und dem Benutzer, bzw. den Anwendungen. Daher muss er eine Reihe von Funktionen implementieren, die im Folgenden beschrieben werden sollen. Zudem wurden eine Reihe von Datentypen definiert, um die komplexen Datenstrukturen zur Konfiguration und Simulationsauswertung einfacher handhaben und übergeben zu können. Diese Typen finden sich in der Headerdatei `vpowertypes.h` und werden im jeweils passenden Abschnitt kurz beschrieben. Für weitergehende Informationen kann auf die Dokumentation des Quelltextes zurückgegriffen werden.

6.1.3.1 Konfiguration des Speichers

Der Code, der für die Konfiguration der simulierten Speicher zuständig ist, befindest sich in der Quelldatei `conf.c`. Hier finden sich Funktionen zum Auslesen und Beschreiben der Konfigurationsregister der Speichermodule. Diese sind im physikalischen Speicher an einer in der Headerdatei `vpowerhw.h` zu spezifizierenden Adresse abgelegt, die beim Initialisieren des Treibers in den Kernelspeicher eingeblendet wird. Somit ist jederzeit ein Zugriff auf die Konfiguration möglich. Das Auslesen bzw. Beschreiben geschieht jedoch nur in jeweils einer Funktion damit ausgeschlossen ist, dass von anderen Teilen des Codes Änderungen direkt in dem Speicher der Hardware vorgenommen werden.

Da Auslesen und Schreiben des Speichers sich eigentlich lediglich in der Richtung unterscheiden, in der die Daten kopiert werden, soll hier exemplarisch nur das Beschreiben des Speichers beschrieben werden. Hierzu wird der Funktion *vpower_set_config* ein Zeiger auf eine Variable des Typs *vpower_conf_t* übergeben. Diese Struct beschreibt eine komplette Speicherkonfiguration inklusive den Busparametern wie Adresse oder Vorgänger-ID und den Modulparameten für jedes Speichermodul. Da diese Daten allerdings nicht in genau dieser Form im Speicher vorliegen, müssen die Werte in einer Schleife pro Modul einzeln geschrieben werden.

Da die einzelnen Speichermodule als Konfigurationsparameter Bitstrings erwarten, deren Felder nicht an Speichergrenzen ausgerichtet sind, war in diesem Zusammenhang das GCC spezifische Attribut *packed* [35], das es erlaubt Kompileroptimierungen bei der Ausrichtung von Structs auszuschalten und so einer Feste Größe und Position für jedes Bitfeld zu bestimmen, eine große Hilfe. Da Linux Kernelmodule sowieso nur mit dem GCC kompiliert werden können, verursacht die Verwendung dieses Attributs keine Kompatibilitätseinschränkungen für das **V-POWER** System.

6.1.3.2 Simulationssteuerung

Für die Steuerung der Simulation sind verschiedene Register der Hardware, genau genommen der **V-POWER** Bus Bridge, auszulesen bzw. zu beschreiben. Zum Einen kann über das Kontrollregister ein Start, Stop oder Reset der Simulation vorgenommen werden, zum Anderen über die 2 Taktzähler der Simulationstakt oder die Anzahl der Stoptakte ausgelesen werden. Die entsprechenden Speicherbereiche sind wie alle weiteren Hardwareadressen in der Headerdatei `vpowerhw.h` definiert und werden beim Start des Treibers in den Kernelspeicher gemappt.

6.1.3.3 Simulationsauswertung

Das Auslesen der Simulationsergebnisse ist ebenfalls in der Datei `sim.c` implementiert und geschieht in der gleichen Art und Weise wie das Auslesen der Konfiguration der Speichermodule. Der Datentyp, in dem die Simulationsdaten zurückgegeben werden heißt *vpower_sim_results_t*.

6.1.4 Schnittstelle zum Userspace

Bisher wurde lediglich beschrieben, welche Funktionalität der Treiber bereitstellt und wie diese im Kernel implementiert ist. Um das **V-POWER** System interaktiv zu nutzen muss jedoch die Möglichkeit geboten werden, aus dem Userspace mit dem Treiber zu kommunizieren, zum Beispiel um eine Speicherkonfiguration zu laden. Da Prozesse unter Linux jedoch keinen Code im Kernelspace aufrufen können und auch die Speicherbereiche strikt getrennt sind, muss dies über einem Umweg geschehen. Prinzipiell gibt es verschiedene Möglichkeiten, Informationen mit einem Treiber auszutauschen, so gut wie alle führen allerdings über Devicedateien im `/dev` Verzeichnis des Systems. Diese können einerseits wie normale Dateien verwendet werden (so liefert zum Beispiel ein Lesen aus der Datei `/dev/random` zufällige Werte, oder es können über einem Mechanismus namens IO-Control Funktionsaufrufe in den Kernelspace simuliert werden. Für den **V-POWER** Treiber wurde letzterer Weg gewählt, da sich so mit sehr wenig Aufwand eine Abstraktionsebene schaffen lässt, die Funktionsaufrufe für Userspace Programme transparent macht. Zusätzlich besteht die Möglichkeit, über einen Eintrag im Pseudodateisystem `/proc` den Status des Treibers abzufragen.

6.1.4.1 IO-Control

Bei *ioctl* handelt es sich um einen Systemaufruf, der auf einem Dateideskriptor arbeitet. Dazu wird eine Nummer, die einen bestimmten Befehl identifiziert und (optional) ein weiteres Argument, üblicherweise ein Zeiger, übergeben. Mittels IO-Control lassen sich so Funktionen im Kernelspace aufrufen. Alle Aufrufe werden in einer bestimmten Funktion des Treibers behandelt, die als Funktionszeiger beim Erstellen des Devices übergeben wird. Hier findet dann die Unterscheidung nach Befehlscodes und die Interpretation der übergebenen Daten statt, so dass diese vollständig vom Entwickler bestimmt werden kann.

Im **V-POWER** Treiber sind die Befehlscodes für die IO-Controls in der Headerdatei `vpowerio.h` definiert, wobei Gebrauch von bestimmten Kernelmakros gemacht wird, damit diese den Konventionen des Kernels entsprechen. Näheres hierzu ist in [33] erläutert, aber für die Funktionalität des Treibers nicht von Bedeutung.

Das Handling der *ioctl* Aufrufe geschieht in der Datei `dev.c`. Hier wird nach zunächst nach Befehlsnummer unterschieden, und dann die entsprechende Funktion im Kernel aufgerufen. Für die Übergabe von Daten wie beispielsweise der struct mit der Konfiguration der Speichermodule muss jeweils noch der Speicherinhalt der entsprechenden Variablen via *copytouser* bzw. *copyfromuser* zwischen Kernel- und Userspace kopiert werden.

6.1.4.2 Der `/proc` Dateisystemeintrag

Das Proc-Dateisystem ist ein vom Kernel bereitgestelltes Pseudo-Datesystem, über das Statusausgaben von Treibern oder Subsystemen des Kernels in Form von Dateien bereitgestellt wer-

den können. Wurde der V-POWER Treiber erfolgreich geladen, findet sich dort ein Eintrag mit dem Namen `/proc/vpower`. In dieser Datei sind Informationen über den aktuellen Status des Systems und die momentane Speicherkonfiguration aufgelistet. Eine Beispielausgabe des Aufrufs `cat /proc/vpower` sieht folgendermaßen aus:

```
RUNNING
234728376
2121133

ID MODULE    EN BASEADDR   HIGHADDR    PREID
 1 DRAM0     1  0x00400000 0x00800000  5
 2 DRAM1     0  0x00000000 0x00000000  0
 5 CACHE0    1  0x00400000 0x00800000  0

V-Power kernel module, debug mode, version information:
         conf.c:   887 2007-06-13 14:40:53Z rotthowe
          dev.c:  1170 2007-07-10 14:13:29Z rotthowe
         main.c:   887 2007-06-13 14:40:53Z rotthowe
         proc.c:  1177 2007-07-10 16:01:13Z rotthowe
          sim.c:  1170 2007-07-10 14:13:29Z rotthowe
         test.c:   970 2007-06-19 15:43:30Z rotthowe
       vpower.h:  1172 2007-07-10 14:28:18Z rotthowe
 vpowerdriver.h:  1170 2007-07-10 14:13:29Z rotthowe
     vpowerhw.h:   917 2007-06-14 20:44:04Z rotthowe
     vpowerio.h:  1170 2007-07-10 14:13:29Z rotthowe
  vpowertypes.h:  1175 2007-07-10 15:52:12Z rotthowe
```

Hier wird zunächst des aktuelle Simulationszustand, gefolgt von den beiden Simulationstaktzählern ausgegeben. Dann folgt die Basiskonfiguration (also lediglich Adressen, Vorgänger und Enabled-Flag) jedes in Hardware verfügbaren Speichermoduls. Schließlich werden noch detaillierte Versionsinformationen ausgegeben, die beim Aufruf von `make` jedes Mal generiert werden.

6.1.4.3 Nutzung der Treiberfunktionen aus dem Userspace

Sämtliche Funktionen, die der Treiber bereit stellt, sind in der Headerdatei `vpower.h` definiert und deklariert. Das heißt, beim Einbinden dieser Headerdatei in selbst geschriebenen C-Quelltext werden die Funktionen direkt in das erstellte Object-File kompiliert. Es handelt sich hierbei lediglich um kleine Hilfsfunktionen, die den Aufruf der `ioctl` Funktion kapseln und somit die Kommunikation mit der V-POWER Hardware ermöglichen, ohne dass der Programmierer diese Feinheiten der Interaktion mit dem Kernel kennen muss. Zusätzlich wird eine Hilfsfunktion zur Verfügung gestellt, mit der alle konfigurierten Speicher in den virtuellen Speicher des laufenden Prozesses abgebildet werden.

Im Folgenden wird ein kurzer Überblick über die Schnittstelle zum Treiber gegeben, genauere Hinweise finden sich im Quelltext der Headerdatei `vpower.h` bzw. in der daraus generierten Doxygen Dokumentation.

Öffnen und Schließen des V-Power Devices

Um einen Kontext zu erhalten, in dem mit dem Treiber kommuniziert werden kann, muss zunächst die Gerätedatei `/dev/vpower` geöffnet werden. Hierzu und zum Schließen des Devices stellt der Treiber zwei Funktionen bereit:

- int **vpower_open** ()
  Gibt ein Dateihandle auf das geöffnete `/dev/vpower` Device oder im Fehlerfall `-1` zurück.
- int **vpower_close** (int fd)
  Schließt das geöffnete Device.

Simulationssteuerung und -auswertung

Zur Steuerung und Auswertung der Simulation werden folgende Funktionen definiert:

- int **vpower_start_sim** (int fd)
  Startet die Simulation.
- int **vpower_stop_sim** (int fd)
  Stoppt die Simulation.
- int **vpower_reset_sim** (int fd)
  Setzt sämtliche Simulationsdaten, also Zähler, Speicherinhalte usw. zurück.
- **vpower_sim_clock_t vpower_get_sim_clock** (int fd)
  Liest den Taktzähler der **V-POWER** Bus Bridge aus.
- **vpower_sim_clock_t vpower_get_stop_clock** (int fd)
  Liest den Stop-Taktzähler der **V-POWER** Bus Bridge aus.
- **vpower_sim_state_t vpower_get_sim_state** (int fd)
  Gibt den Simulatonsstatus zurück.
- int **vpower_get_sim_results** (int fd, **vpower_sim_results_t** *results)
  Liest die Simulationsergebnisse aller simulierten Speicher aus.

Konfiguration

- int **vpower_get_config** (int fd, **vpower_conf_t** *config)
  Liest die aktuelle Konfiguration der Speichermodule aus.

- int **vpower_set_config** (int fd, **vpower_conf_t** *config)
  Setzt eine Speicherkonfiguration.

Einbinden des simulierten Speichers

- int **vpower_map_devices** (int devfd)
  Blendet alle Speichermodule in den virtuellen Speicher des laufenden Prozesses ein. Die Module werden werden hierbei an die Speicherstelle eingebunden, in der sie auch im physikalischen Speicher liegen. Dies ist genau die Adresse, die auch in der Konfiguration als *base address* gesetzt wird. Ist diese Speicherstelle bereits belegt (was nicht passiert, solange der Benutzer nicht schon zuvor anderen Speicher in den Speicherbereich des Prozesses gemappt hat), gibt die Funktion einen Fehler zurück.

6.1.4.4 Beispiel

Konkret kann der **V-POWER** Kerneltreiber folgendermaßen genutzt werden: Zunächst muss das Modul durch den Befehl `insmod vpower` geladen werden. Auf der seriellen Konsole erscheinen nun einige Logmeldungen und die Version des Treibers. Je nachdem, ob das *debug* Flag beim Kompilieren gesetzt wurde, fallen diese unterschiedlich ausführlich aus.

Nach dem erfolgreichen Laden sind im `/dev` und `/proc` Verzeichnis des Dateisystems entsprechende Einträge mit dem Namen `vpower` erstellt worden auf die nun wie zuvor erläutert zugegriffen werden kann. Nun können beliebige Applikationen im Userspace die **V-POWER** API nutzen, sofern sie im Kontext des Benutzers *root* gestartet werden. Dafür muss die Headerdatei `vpower.h` eingebunden werden, so dass die oben beschriebenen Hilfsfunktionen verfügbar sind. Im Anhang C.1 befindet sich ein ausführlich dokumentierter Codeschnipsel, der die Nutzung der Schnittstelle veranschaulicht.

Es gibt jedoch außer dem direkten Aufruf der **V-POWER** API noch weitere Möglichkeiten, das System zu konfigurieren und Simulationsergebnisse auszulesen. Diese sollen in den folgenden Abschnitten genauer beschrieben werden.

6.2 V-Power Konfigurationsprogramm

Das V-POWER Konfigurationsprogramm, nachfolgend auch Konfigurationsprogramm genannt, ist die Benutzeroberfläche des V-POWER Speicher-Hierarchie Profilers. Geschrieben wurde es in Java, um auf unterschiedlichen, mit der Java Technologie ausgestatteten, Betriebssystemen lauffähig zu sein.

6.2.1 Erste Schritte

6.2.1.1 Starten des Konfigurationsprogramms

Da das Konfigurationsprogramm in Java geschrieben ist, muss es mittels Kommandozeile (oder, falls vom Fenster-Manager unterstützt, auch durch Doppelklick auf das JAR-Archiv) gestartet werden. Dazu ist folgendes in die Kommandozeile einzugeben:

```
> java -jar VPowerGUI.jar
```

6.2.1.2 Hauptfenster

Das Hauptfenster des Konfigurationsprogramms gliedert sich, wie in Abbildung 6.1 zu sehen, horizontal in zwei Bereiche. Auf der linken Seite befindet sich eine Anzeige, welche graphisch die Baumstruktur einer Speicher-Hierarchie darstellen kann. Es ist zudem möglich einen Knoten, welcher ein Speichermodul repräsentiert, im Baum mittels linker Maustaste auszuwählen und ihn bei gedrückter Maustaste an einen anderen Speichertyp anzuhängen. Somit kann effektiv die Speicher-Hierarchie verändert werden, ohne jedwede Konfigurationsdateien per Hand verändern zu müssen.

6.2.2 Speicher-Hierarchie bearbeiten

Das Konfigurationsprogramm erlaubt es sowohl eine neue Speicher-Hierarchie zu erstellen, als auch eine zuvor gespeicherte Hierarchie zu laden. Beide Möglichkeiten sind über das Menü des Hauptfensters zugänglich, wie in Abbildung 6.2 zu sehen ist. Über dieses Menü ist es zudem möglich eine bearbeitete Hierarchie in eine Datei abzuspeichern.

6.2.2.1 Speichermodul hinzufügen/entfernen

Ein neues Speichermodul kann hinzugefügt werden, indem mittels rechter Maustaste in die Anzeige der Baumstruktur geklickt wird. Es öffnet sich ein Aufklappfenster, welches nun die Möglichkeit bietet einen Typus von Speichermodul zur Speicher-Hierarchie hinzuzufügen bzw. ein ausgewähltes Speichermodul zu entfernen. Abbildung 6.3 zeigt dies im Beispiel für eine leere Hierarchie.

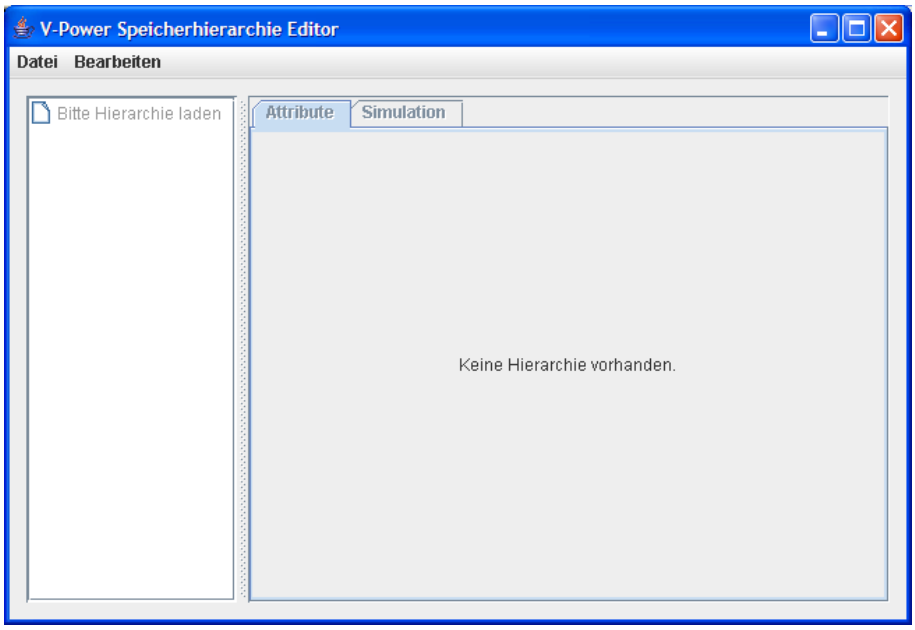


Abbildung 6.1: *Das Hauptfenster des Konfigurationsprogramms nach dem Start.*

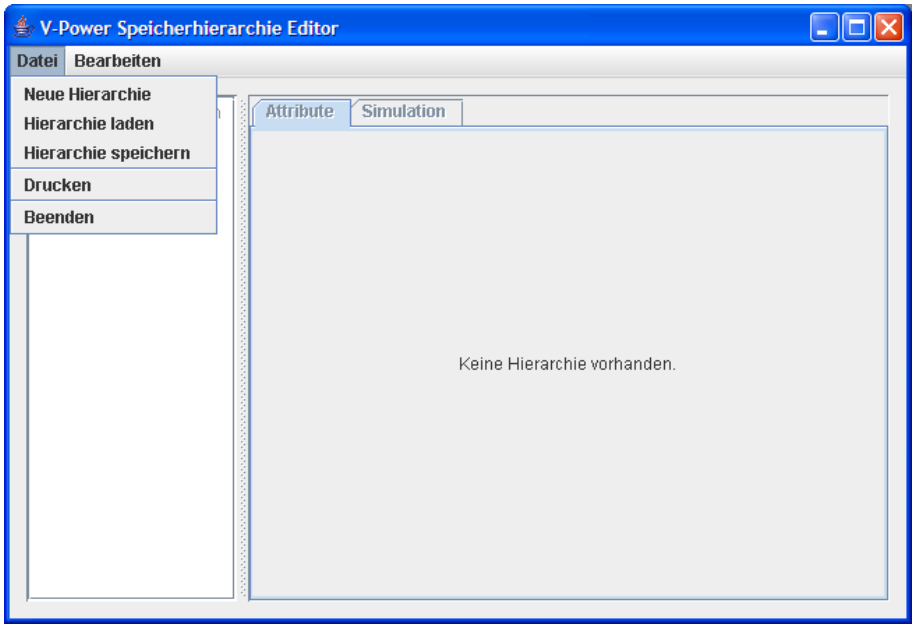


Abbildung 6.2: *Das Menü des Hauptfensters.*

Hierbei ist zu beachten, dass alle drei hierbei auswählbaren Speichermodul-Typen nur auf der obersten Hierarchiestufe einfügbar sind. Dies liegt daran, dass sowohl das SRAM-, DRAM- sowie Flash-Speichermodul die elementaren Speichertypen darstellen. Oberhalb dieser können mehrere Cache-Stufen liegen.
Wird eines dieser Speichermodul-Typen eingefügt und mit Rechtsklick ausgewählt, so ist nun auch die Option des „Modul entfernen“s gegeben als auch das Einfügen eines Cache-Speichermoduls unterhalb des ausgewählten Speichermoduls. Dies ist in Abbildung 6.4 ersichtlich.

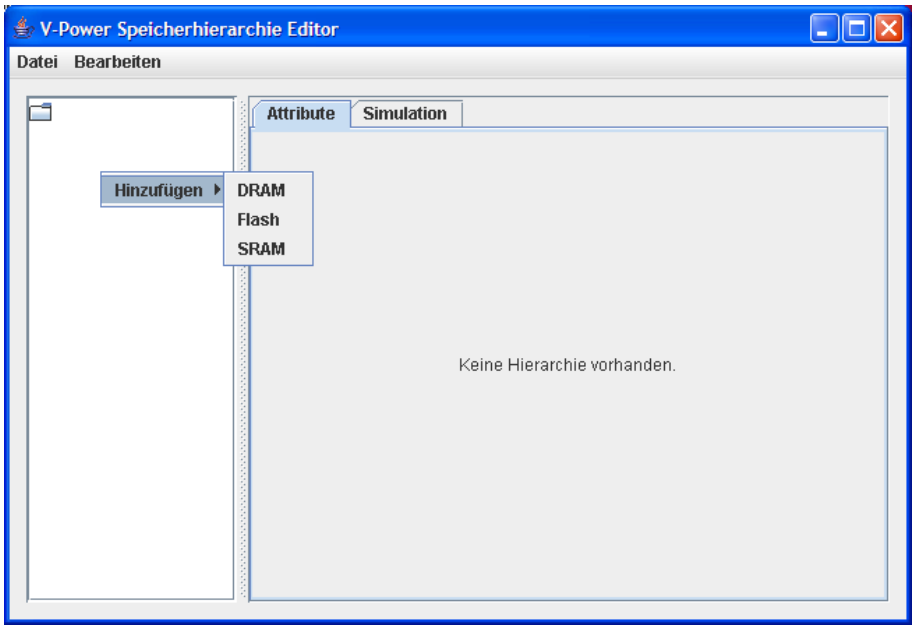


Abbildung 6.3: *Das geöffnete Aufklappfenster.*

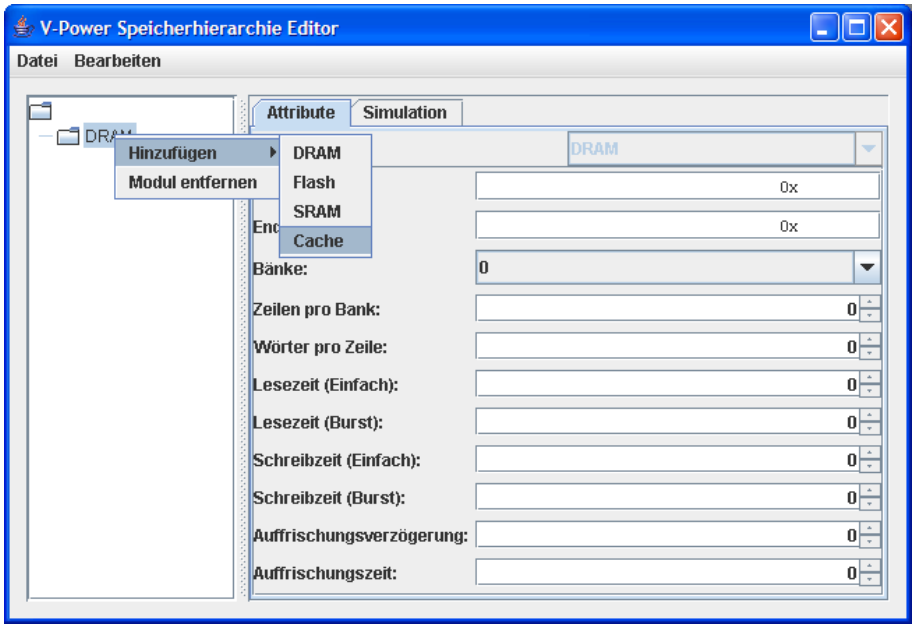


Abbildung 6.4: *Möglichkeit des Entfernens eines Speichermoduls sowie Einfügen eines Caches.*

Nachfolgend soll nun auf die einzelnen Einstellungsmöglichkeiten der jeweiligen Speichermodule eingegangen werden. Diese können im Konfigurationsprogramm erreicht werden, indem das entsprechende Speichermodul in die Hierarchie eingefügt wird und nachfolgend mit rechtem Mausklick in der Anzeige der Baumstruktur ausgewählt wird. In der rechten Hälfte des Fensters wird sich sodann das zum Speichermodul passende Formular anzeigen, in welchem eine Modifikation der Einstellungen möglich ist.
Jedem Speichermodul gleich sind die ersten beiden Einstellungen, die Basis- sowie Endadresse.

Beide werden in Hexadezimaler Form vom Benutzer eingegeben.

- **Basisadresse**
  Bestimmt die Anfangsadresse des Speicherbereiches, den das betreffende Speichermodul enthalten soll
- **Endadresse**
  Bestimmt die Endadresse des Speicherbereiches, den das betreffende Speichermodul enthalten soll

Es ist zu beachten, dass beide Adressen auf den Anfang von 32 Bit Wörtern ausgerichtet sein müssen. Um dies zu gewährleisten werden vom Konfigurationsprogramm nur Adressen akzeptiert, die auf 0, 4, 8 sowie C enden.
Ein zusätzliches Leistungsmerkmal des Konfigurationsprogramms bezüglich der Adressen kommt beim Einfügen eines Cache-Speichermoduls zum tragen. So übernimmt der Cache die Basis- sowie Endadresse von seinem Elter in der Hierarchie. Falls schon eines oder mehrere Caches den selben Elter besitzen, so wird der größte noch freie und zusammenhängende Speicherbereich dem neuen Cache-Speichermodul zugewiesen.

6.2.2.2 DRAM Einstellungen

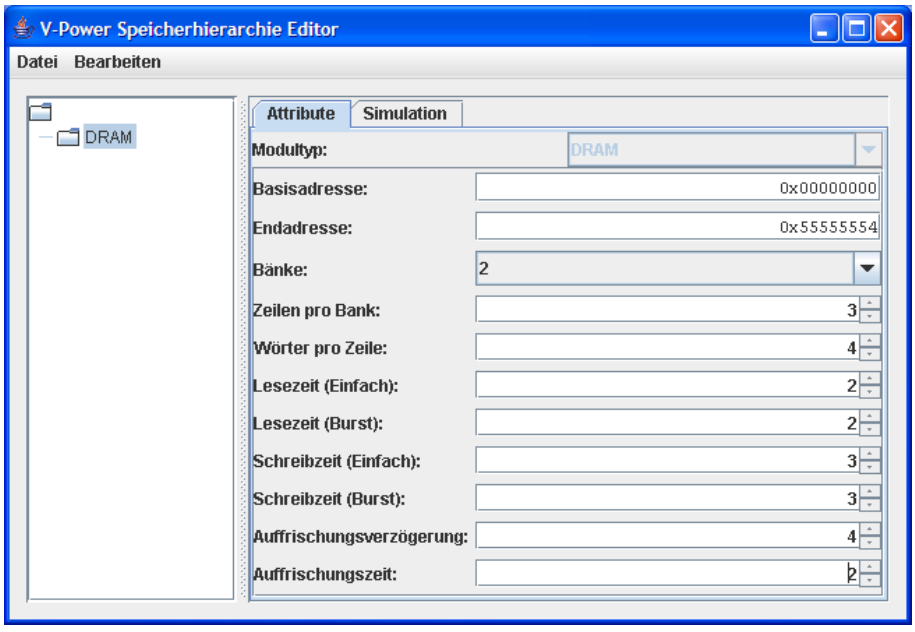


Abbildung 6.5: *Einstellungen des DRAM-Speichermoduls.*

Abbildung 6.5 zeigt die Einstellungsseite des DRAM-Speichermoduls. Im einzelnen sind folgende Einstellungen möglich:

- **Bänke**
  Bestimmt die Anzahl der Speicherbänke. Es können 0 bis 7 Speicherbänke simuliert werden.

- **Zeilen pro Bank**
  Bestimmt die Anzahl von Zeilen der DRAM-Speicherzellen in einer Bank.

- **Wörter pro Zeile**
  Bestimmt die Anzahl der Wörter (32 Bit) pro Zeile. Definiert mit der Anzahl der Bänke und Anzahl der Zeilen pro Bank die Speicherkapazität des betreffenden DRAM-Speichermoduls.

- **Lesezeit (Einfach)**
  Bestimmt die Anzahl an Taktzyklen, die ein auf ein einzelnes Speicherwort zugreifender Lesezugriff dauert.

- **Lesezeit (Burst)**
  Bestimmt die Anzahl an Taktzyklen, die ein auf mehrere zusammenhängende Speicherwörter zugreifender (Burst) Lesezugriff dauert.

- **Schreibzeit (Einfach)**
  Bestimmt die Anzahl an Taktzyklen, die ein, ein einzelnes Speicherwort schreibender, Schreibzugriff dauert.

- **Schreibzeit (Burst)**
  Bestimmt die Anzahl an Taktzyklen, die ein, mehrere zusammenhängende Speicherwörter schreibender, (Burst) Schreibzugriff dauert.

- **Auffrischungsverzögerung**
  Bestimmt die Anzahl an Taktzyklen, die zwischen zwei Auffrischungszyklen vergehen.

- **Auffrischungszeit**
  Bestimmt die Anzahl an Taktzyklen, die ein Auffrischungszyklus dauert.

6.2.2.3 SRAM Einstellungen

Abbildung 6.6 zeigt die Einstellungsseite des SRAM-Speichermoduls. Im einzelnen sind folgende Einstellungen möglich:

- **Lesezeit**
  Bestimmt die Anzahl an Taktzyklen, die ein auf ein einzelnes Speicherwort zugreifender Lesezugriff dauert.

- **Schreibzeit**
  Bestimmt die Anzahl an Taktzyklen, die ein, ein einzelnes Speicherwort schreibender, Schreibzugriff dauert.

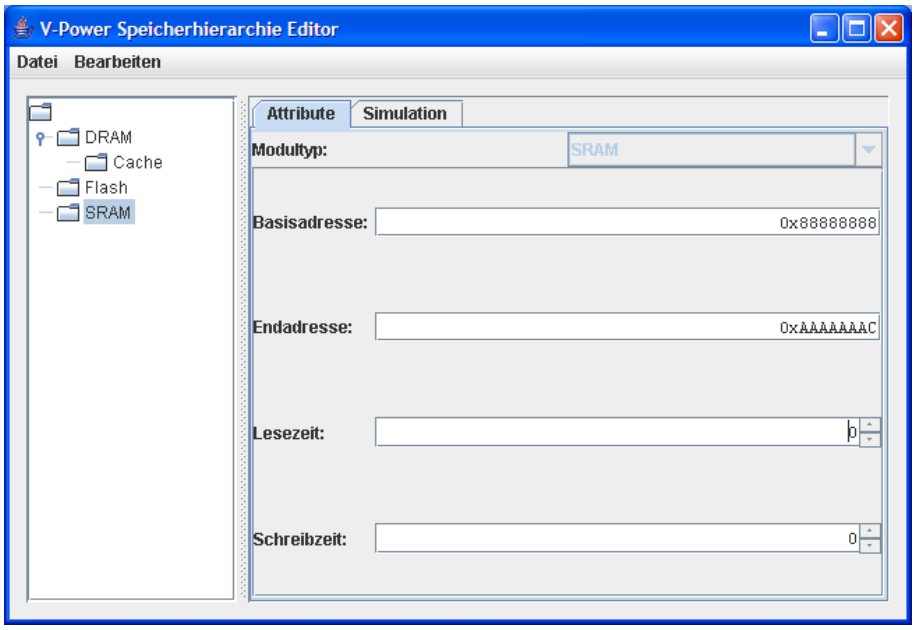


Abbildung 6.6: *Einstellungen des SRAM-Speichermoduls.*

6.2.2.4 Flash Einstellungen

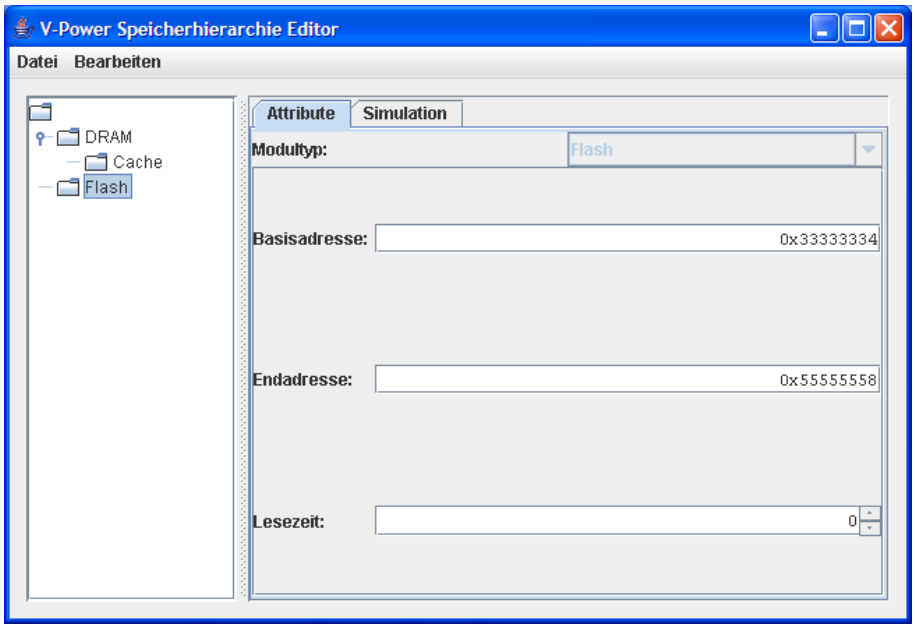


Abbildung 6.7: *Einstellungen des Flash-Speichermoduls.*

Abbildung 6.7 zeigt die Einstellungsseite des Flash-Speichermoduls. Im einzelnen sind folgende Einstellungen möglich:

- **Lesezeit**
  Bestimmt die Anzahl an Taktzyklen, die ein auf ein einzelnes Speicherwort zugreifender Lese-

zugriff dauert.

6.2.2.5 Cache Einstellungen

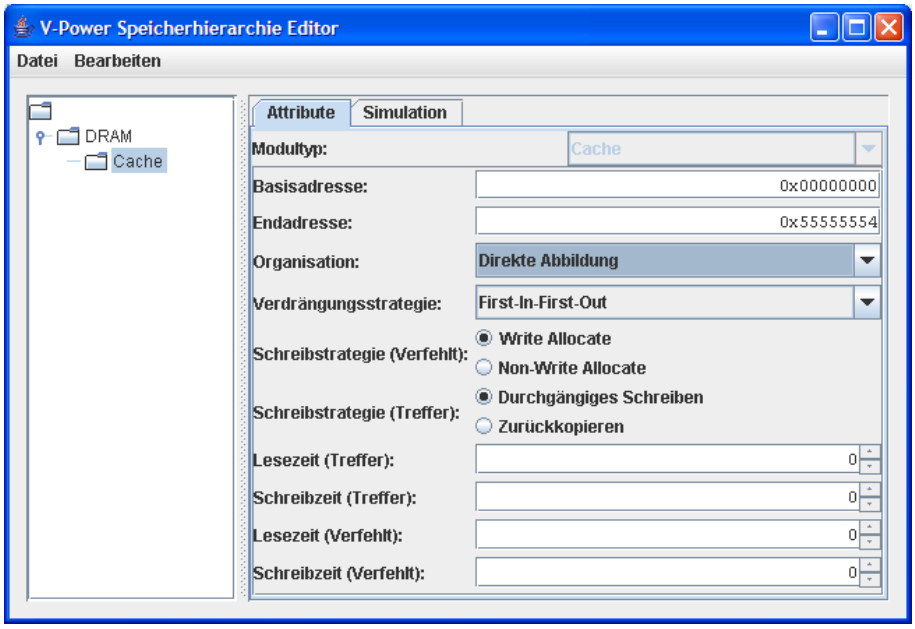


Abbildung 6.8: *Einstellungen des DRAM-Speichermoduls.*

Abbildung 6.8 zeigt die Einstellungsseite des Cache-Speichermoduls. Im einzelnen sind folgende Einstellungen möglich:

- **Organisation**
  Bestimmt die Organisation des Cache-Speichermoduls. Verfügbar sind folgende Organisationen:
  - **Direkte Abbildung (engl. direct mapping)**
  - **2-fach Assoziativ**
  - **4-fach Assoziativ**
  - **8-fach Assoziativ**
- **Verdrängungsstrategie**
  Bestimmt die Verdrängungsstrategie des Cache-Speichermoduls. Verfügbar sind folgende Verdrängungsstrategien:
  - **„First-In-First-Out“**
  - **„Least Recently Used“**
  - **„Least Frequently Used“**
  - **„Round Robin“**

- **Schreibstrategie (Verfehlt)**
  Bestimmt die Schreibstrategie bei einem nicht im Cache gepufferten Speicherwort. Verfügbar sind folgende Schreibstrategien:
  - **„Write-Allocate“**
  - **„Non-Write-Allocate“**
- **Schreibstrategie (Treffer)**
  Bestimmt die Schreibstrategie bei einem im Cache gepufferten Speicherwort. Verfügbar sind folgende Schreibstrategien:
  - **„Write-Through“**
  - **„Write-Back“**
- **Lesezeit (Treffer)**
  Bestimmt die Anzahl an Taktzyklen, die ein auf ein einzelnes Speicherwort zugreifender Lesezugriff dauert, falls das Speicherwort im Cache gepuffert ist.
- **Schreibzeit (Treffer)**
  Bestimmt die Anzahl an Taktzyklen, die ein, ein einzelnes Speicherwort schreibender, Schreibzugriff dauert. Das Speicherwort ist im Cache vorhanden.
- **Lesezeit (Verfehlt)**
  Bestimmt die Anzahl an Taktzyklen, die ein auf ein einzelnes Speicherwort zugreifender Lesezugriff dauert, falls das Speicherwort nicht im Cache gepuffert ist.
- **Schreibzeit (Verfehlt)**
  Bestimmt die Anzahl an Taktzyklen, die ein, ein einzelnes Speicherwort schreibender, Schreibzugriff dauert. Die Speicherwort ist im Cache vorhanden.

6.2.3 Simulationsergebnisse anzeigen

Es ist möglich sich im Konfigurationsprogramm die Simulationsergebnisse einer **V-POWER** Speichersimulation anzeigen zu lassen. Hierzu wird eine **V-POWER** Konfigurationsdatei vom Userspace Programm zurückgegeben, welche diese Daten zusammen mit der simulierten Speicherhierarchie enthält. Diese können dann mittels Klick auf die Karteikarte „Simulation“ angezeigt werden, siehe Abbildung 6.9. Es ist zudem möglich die Ergebnisse zwecks Weiterverwendung mittels „Kopieren und Einfügen“ (engl. copy&paste) aus dem Konfigurationsprogramm zu kopieren.

6.3 Fazit

Erst durch den Kerneltreiber kann die **V-POWER** Hardware überhaupt sinnvoll genutzt werden. Sämtliche Änderungen an der Konfiguration und das Auslesen der Ergebnisse laufen durch diesen.

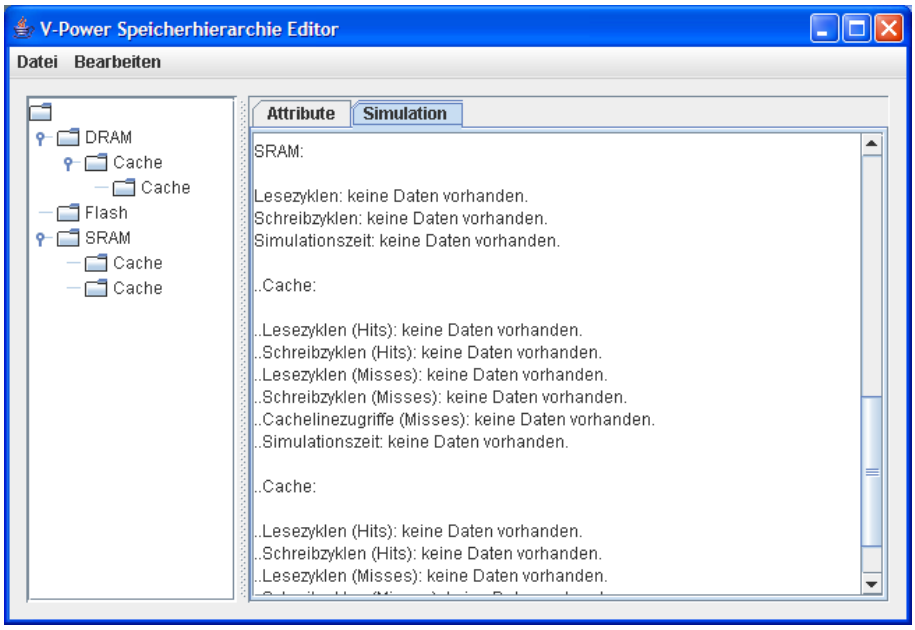


Abbildung 6.9: *Anzeige der Simulationsdaten.*

Damit diese Vorgänge möglichst transparent für den Nutzer bzw. die laufende Anwendung sind, wurde eine API bereitgestellt, die von den Kernelschnittstellen abstrahiert und einfache C Funktionen bereitstellt. Damit ist der gesamte Funktionsumfang der V-POWER Hardware unter dem Linux Betriebssystem einsetzbar.

Für das Editieren der Hardwarekonfiguration ist es allerdings in einigen Fällen unpraktisch, dies über die bereitgestellte API in C Code zu erledigen. Daher wurde ein Programm geschrieben, mit dem auf einfache Art und Weise eine zu testende Hardwarekonfiguration erstellt werden kann. Durch die Implementierung in Java ist dies auf den meisten Workstations lauffähig.

Die Kombination der oben erwähnten Software ermöglicht es, schnell ohne ohne genauere Kenntnisse über die Funktionsweise des V-POWER Systems Speicherkonfigurationen zu erstellen, zu testen und die Ergebnisse zu vergleichen.

Kapitel 7

Integration und Analyse

Nach Beschreibung der Hard- und Softwarekomponenten soll der Entwurf mittels Modelsim von Mentor Graphics validiert werden. Ein näheres Tutorial zum Thema Modelsim wurde in der Seminarphase erarbeitet und ist separat verfügbar.

Dieses Kapitel gliedert sich in 2 Abschnitte. Zum einen in den Bereich 7.1 Integration, hier soll aus den einzelnen Hardwarebausteinen, die bereits in Kapitel 3 beschrieben wurden, ein konkreter Aufbau für eine Simulation entstehen.

Im Bereich 7.2 Analyse wird für den vorliegenden Aufbau die Konfiguration der Bausteine, eine simulierte Anfrage, sowie das abschließende Auslesen der Simulationsregister mittels Modelsim validiert.

7.1 Integration

Bevor ein Test ausgeführt werden kann, muss zunächst der zu testende Entwurf sowie dessen Konfiguration festgelegt werden.

Bis jetzt wurden nur die einzelnen Oberklassen von Speicher-Bausteinen im Kapitel 4 vorgestellt. Hier standen die Module

- Cache
- DRAM
- Flash
- SRAM

zur Verfügung. Jeder dieser Bausteine wird durch eine Reihe von typenspezifischen Eigenschaften

zu einem konkreten Speichermodul, welches dann von den Spezifikationen her in realer Hardware realisiert werden könnte.

Da die konzipierten Bausteine selber nur die Simualtionslogik für das reale Gegenstück enthalten, stellen sie allerdings keine echten Speicherbausteine dar. Sie passen ihr Verhalten entsprechend an und speichern lediglich die für die Auswertung benötigten Daten. Echte Daten werden nicht gespeichert, sondern wie bereits beschrieben in einem großen Speicher beliebigen Typs abgelegt, nachdem der Simulationsbaustein die benötigten Werte einer Speichertransaktion erfasst hat.

Unter der Integration wird der Zusammenschluss einzelner Bausteine ohne Konfiguration verstanden. Das bedeutet man legt in dieser Phase lediglich die Anzahl von benötigten Bausteinen aus jeder der 4 Speicherklassen fest. Zudem werden in der Integration die gewählten Speichermodule mittels Interfaces über Bus-Systeme mit der Steuerungslogik verbunden.

Ferner werden in dieser Phase jedem einzelnen Speicherbaustein ein Konfigurations- und Simulationsregister zugeordnet. Die Größe der Register richtet sich nach dem benötigten Bedarf der Module (Anzahl der Konfigurationsbits und benötigte Anzahl an 32-bit Ergebnisregistern).

Dabei werden allen Registern (Konfiguration, Simulation und Befehlsregistern der Steuerungslogik) konkrete Adressbereich zugeordnet, damit diese später über die Software adressierbar sind. Dabei ist zu beachten, dass in dieser Phase keine Adressbereiche auf die Speicherbausteine selbst gelegt werden. Dies geschieht zur Laufzeit mittels der Konfiguration. Diese kann nun über eine Folge von 32-Bit Wörtern an die Adressen der Konfigurationsregister geschrieben werden. Dabei spezifiziert jede Speicherklasse die Bedeutung der Konfiguration selbst (Siehe Kapitel 4 für Details).

Vorbereitung

Für die hier beschriebene Testumgebung soll ein sehr einfaches Layout gewählt werden. Dies hat zwei prinzipielle Gründe. Zum einen verhalten sich der **SRAM-** und **Flash-Baustein** innerhalb eines Gesamtentwurfs wie der **DRAM-Baustein**, d.h. sie haben keine separate Speicheranbindung für die Auslagerung zusätzlicher Verwaltungsdaten. Ihre Simulationsergebnisse und ihren Zustand verwalten sie nur intern, d.h. hier reicht ein Test der einzelnen Module ohne Integration in ein Gesamtlayout. Zum anderen lässt sich die Speicheranbindung des **Cache-Bausteins** nur mit unverhältnismäßig hohem Aufwand testen, denn es müssten alle Antworten aus dem realen Datenspeicher (der die Cachetabellen enthält, da die Zellen des FPGAs hierfür nich ausreichen) innerhalb der Testumgebung von Hand eingegeben werden. Der reale Datenspeicher (im FPGA Entwurf ist dies der reale DIMM Speicher) kann in Modelsim nicht genutzt werden, da Modelsim keine reale Hardware, wie sie auf dem Board beispielsweise eben in Form des DIMMs zu finden ist, simuliert.

Der **DRAM-Baustein** wird in der Testumgebung also stellvertretend für die beiden anderen Module eingesetzt. Die zu erwartenden Ergebnisse werden aus Tests, die nur exklusiv mit dem DRAM Baustein (vergleiche Kapitel 4) vorgenommen wurden, abgeleitet.

Die Speicheranbindung findet zentral über die MemAccess Komponente (siehe Abschnitt 3.4.2) statt und wurde bereits mit Testfällen direkt auf dem Board validiert. Daher muss in Modelsim lediglich das korrekte Ansprechen der Komponente überprüft werden.

In der folgenden Testsituation sollen vor allem die folgenden Punkte untersucht werden

- Entgegennahme von Anfragen über den Processor Local Bus durch die VBB (insb. Betrachtung des inneren Zustands und der Ausgabedaten)
- Validierung des VBB internen VPB-IPIFs insb. Betrachtung der korrekten Anfragestellung an den VPB
- Korrekte Übergabe des Token und der Anfragedaten über den **V-POWER** -Bus
- Verarbeitung der Anfragen von einem Memory-Module-IPIF (Validierung des neuen VPB Zustandes und korrekte Ansprechung der Bestandteile eines Memory Moduls)
- Bearbeitung der Anfrage durch ein Memory-Modul (hier DRAM-Baustein) und Generierung neuer Simulationsdaten
- - *oder* - Ablegen von Daten im Config-Register
- - *oder* - Auslesen von Simulationsergebnissen aus dem Simulation-Register

Erstellung einer Testumgebung

Es werden nun die Kapitel 3 beschriebenen Funktionskomponenten wie **V-POWER** -Bus, **V-POWER** -Bus-Bridge und **V-POWER** -VPB-IPIF, sowie der in Kapitel 4 beschriebene **DRAM-Baustein**, zusammengesetzt.

Dabei wird zunächst ein neues Modul `vpower_mm_dram` geschaffen, das aus 4 Teilen besteht:

- DRAM-Baustein
- Config-Register (0x10000810-0x10000823, 160 Bit)
- Simulation-Register (0x10001000-0x10001023, 288 Bit)
- VPB-IPIF (0x10000800-0x1000080F, 128 Bit)

Die Angaben in Klammern beziehen sich auf die Adressbereich der einzelnen Bestandteile. Beispielsweise besteht das Config-Register aus 5 Registern mit jeweils 32 Bit. Der Config Speicher wird mit dem Konfigurationseingang des DRAM-Baustein verbunden. Dabei hat die Config-Schnittstelle eine Breite von 192 Bit, das Config Register jedoch nur 160 Bit. Die ersten 32 Bit der Konfiguration (laut Tabelle 4.3, Seite 115) stellen die Startadresse, auf die der DRAM sensitiv sein soll, dar.

Da diese laut Konvention des VPB-IPIF auch dort benötigt werden, wird dieses Datum aus dem VPB-IPIF abgegriffen (Das VPB-IPIF stellt hierzu einen separaten Ausgang `pot_vip2ip_RealBA` bereit).

Die 288 Bit der Simulationsdatenbreite werden direkt mit einem Eingang des Simulationsregister verbunden.

Für die Register wird das generische Registerfile eingesetzt (Siehe Kapitel 3). Als Verbindung zum DRAM-Baustein wurde jeweils ein Unclocked Output bzw. Input hinzugefügt. Unclocked bedeutet in diesem Fall, dass der DRAM-Baustein taktunabhängig das Simulationsregister beschreiben, bzw. das Konfigurationsregister auslesen kann. Vom **V-POWER** -Bus sind die Register nur sequentiell in 32 Bit Blöcken über Adressen ansprechbar.

Die 4 Register des IPIFs enthalten den Aktivierungszustand des Moduls (Enable), Vorgänger-ID (PreID) sowie die Start- und Endadresse, auf welche der dahinterliegende Baustein reagieren soll.

Nach der Bereitsstellung des `vpower_mm_dram` Moduls erfolgt die Realisierung der **V-POWER** -Busleitungen und eines Speicherbusses, welcher später mit dem DDR-Interface (Siehe Kapitel 2 Konzept) verbunden wird. An den **V-POWER** -Bus wird nun das `vpower_mm_dram` Modul angeschlossen. Ferner wird die **V-POWER** -Bus-Bridge welche ebenfalls über ein VPB-IPIF verfügt an den VPB angeschlossen. Zudem wird die VBB als einzige mit dem IPIF des Processor Local Bus und dem Speicherbus zum DDR-Interface verbunden. Die gesamte Verdrahtung wird in einem Container namens `vpower_mem_logic` platziert. Dieser wird nun wie in Kapitel 2 beschrieben zwischen PLB und DIMM Speicher platziert.

Damit wurde eine minimal lauffähige Version des **V-POWER** Speicherhierarchieprofilers geschaffen, welche in allen Eigenschaften einem komplexeren Entwurfs in nichts nachsteht. Im nächsten Abschnitt wird nun diese Zusammenstellung auf die bereits beschriebenen Punkte hin untersucht.

7.2 Analyse

In diesem Abschnitt wird nun eine Modelsim Simulation untersucht, welche folgendes leisten soll:

- Schreiben eines vollständigen Konfigurationsstrings in 32 Bit Paketen
- Starten der Simulation mittels Schreiben ins Befehlsregister in der VBB
- Simuliertes Ausführen einer Anfrage an den DRAM
- Stoppen der Simulation mittels Schreiben ins Befehlsregister in der VBB
- Lesen der Simulationsergebnisse in jeweils 32 Bit Paketen

Beim Start des Systems, nach der Initialisierungsphase, beinhalten alle Konfigurationsregister Standardwerte, die zur Synthesezeit eingegeben wurden. Dieses Vorgehen soll verhindern, dass sich zu Beginn des Systems eine ungültige Konfiguration innerhalb der Module einschwingt.

Bei dieser Testumgebung bedeutet dies, dass die Simulation deaktiviert ist und das DRAM-Simulationsmodul so eingestellt ist, dass es den kompletten Speicherbereich abdeckt. Zu Beginn ist der DRAM also auf die unteren 64 MB eingestellt (`0x0000_0000 -- 0x03FF_FFFF`). Der restliche reale Speicherbereich steht dem System nicht zur Verfügung, sondern kann nur von der Hardware (bspw. den Simulationscaches) als Arbeitsspeicher genutzt werden.

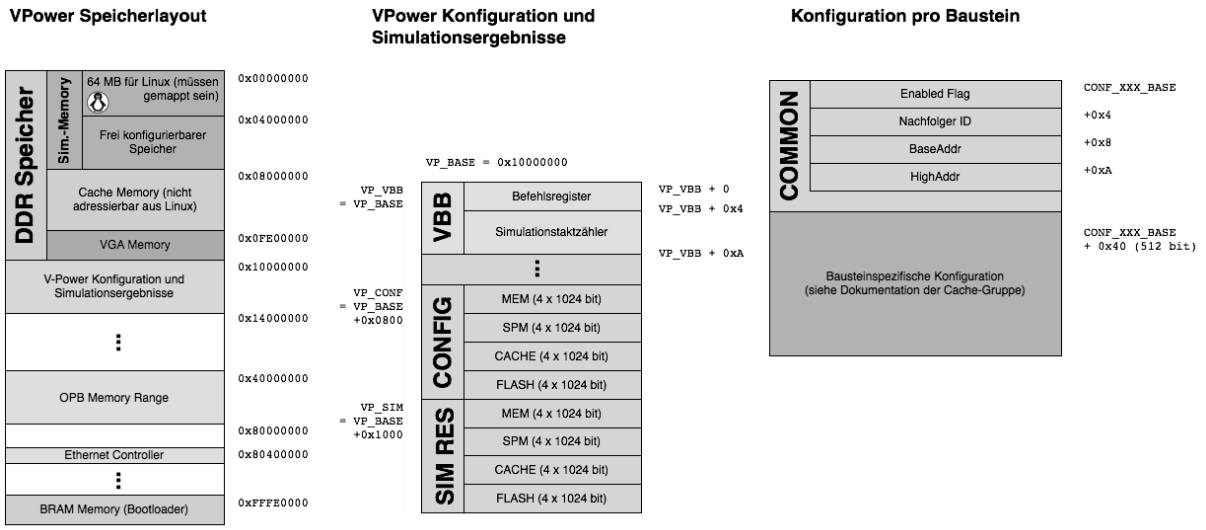


Abbildung 7.1: *Übersicht der Adressbereich eines* **V-POWER** *Speicherhierarchieprofiles*

Abbildung 7.1 zeigt eine abstrakte Übersicht wie die Adressbereiche innerhalb einer beliebigen Simulationsumgebung zu vergeben sind. Zu beachten ist, dass hier wie bereits erwähnt keine endgültige Konfiguration vorgenommen werden muss, sondern lediglich der zu simulierende Speicherbereich, sowie die Adressbereiche der Konfigurations- und Simulationsergebnisregister der Memory Module und die Adressen der Befehlsregister innerhalb der VBB angegeben werden müssen.

Schreiben der Konfiguration

Als ein erster Test soll nun die Basisadresse, auf die das DRAM-Simulationsmodul sensitiv ist, geändert werden. Zu beachten ist, dass die Standardkonfiguration die einzig legale Konfiguration darstellt, da diese die realen 64 MB exakt auf den DRAM Baustein abbildet (`0x0000_0000 -- 0x03FF_FFFF`). Für die Analyse der Funktionen soll nun der Bereich bis `0x000_00FF` ausgeblendet werden. Nach der Änderung wird in diesem Bereich kein Simulationsmodul mehr sensitiv sein. Es können daher auch später Tests für den Fehlerfall vorgenommen werden (Timeouts, etc.). Dies soll aber nicht Teil der Dokumentation sein, da hier lediglich das Vorgehen nachvollzogen werden soll. Aufgrund der Realisierung in Hardware kann der **V-POWER** -Speicherhierarchieprofiler keine Plausibilitätsprüfung der gesetzten Konfiguration leisten. Dies wird in der Linuxumgebung durch den

Treiber realisiert.

Um den Bereich wie geplant zu ändern, muss das Register `BASEADDR`, dass in jedem Memory Modul ein Teil des VPB-IPIFs ist, neu beschrieben werden. Da das Modul das einzige in der Testumgebung ist, beginnt es laut Abbildung 7.1 bei `0x1000_0800`. Als drittes Register liegt die `BASEADDR` also bei `0x1000_0808`.

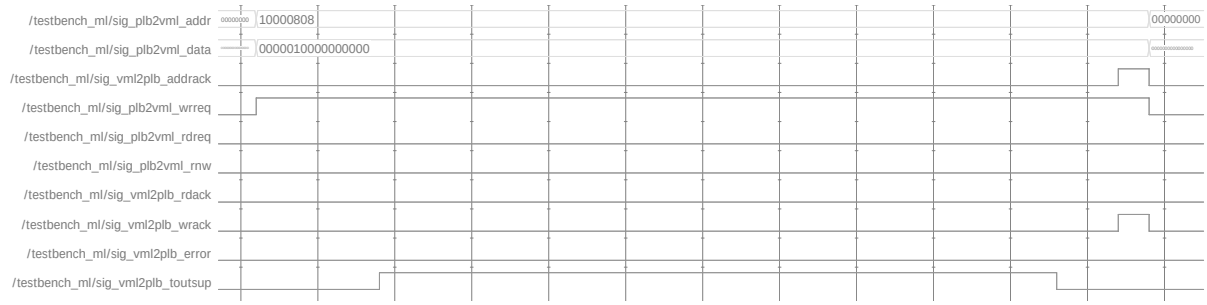


Abbildung 7.2: *Schreiben einer Konfiguration (PLB Ansicht)*

Abbildung 7.2 zeigt den Ablauf auf dem Processor Local Bus. Vom Prozessor aus werden die Signale `Write Request`, `Address`, und `Data` entsprechend gesetzt. Nach Ablauf der Transaktion bestätigt die VBB mit den Signalen `AddressAck` und `WriteAck`. Zu beachten ist hierbei, das die Ack-Signale solange anliegen bis eins der Anfragesignale geändert wird (In diesem Fall werden sofort alle Signale auf 0 zurückgesetzt).

Interessant ist nun die Untersuchung im Inneren der **V-POWER** -Logik. Dazu soll zunächst das Verhalten der VBB untersucht werden. Dazu reicht es aus, die Zustandsübergänge des VBB-Controllers zu betrachten (Abbildung 7.3).

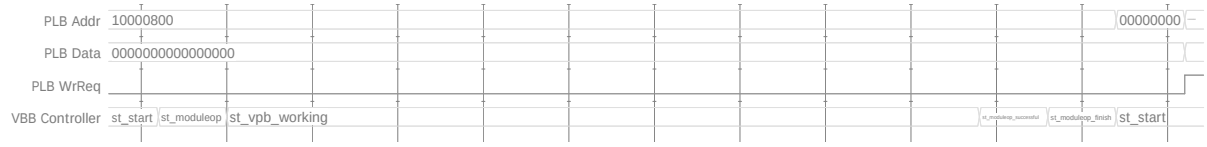


Abbildung 7.3: *Zustandsübergänge der VBB beim Vorgang "Konfiguration schreiben"*

Im Gegensatz zu einer normalen Speichertransaktion muss bei der Speicherung der Konfiguration nur der VPB aktiviert werden. In diesem Fall geht der Controller also von `st_start` in den Modus `st_vpb_working`. Durch den Zustand `st_moduleop` wird gekennzeichnet, dass es sich um eine Interaktion mit einem Modul handelt und keine Simulationsoperation. Während des Zustandes `st_vpb_working` findet die Übergabe der VPB Kontrolle an das betroffene Modul statt. Sobald die Kontrolle zurückgegeben wird findet der Zustandsübergang in `st_moduleop_successful`.

Eine Ebene tiefer wird also der VPB aktiviert. Wie bereits beschrieben findet hier die Tokenübergabe statt.

Da auf dem VPB keine Arbitrierung stattfindet, wird ein Token von einem Modul an das nächste

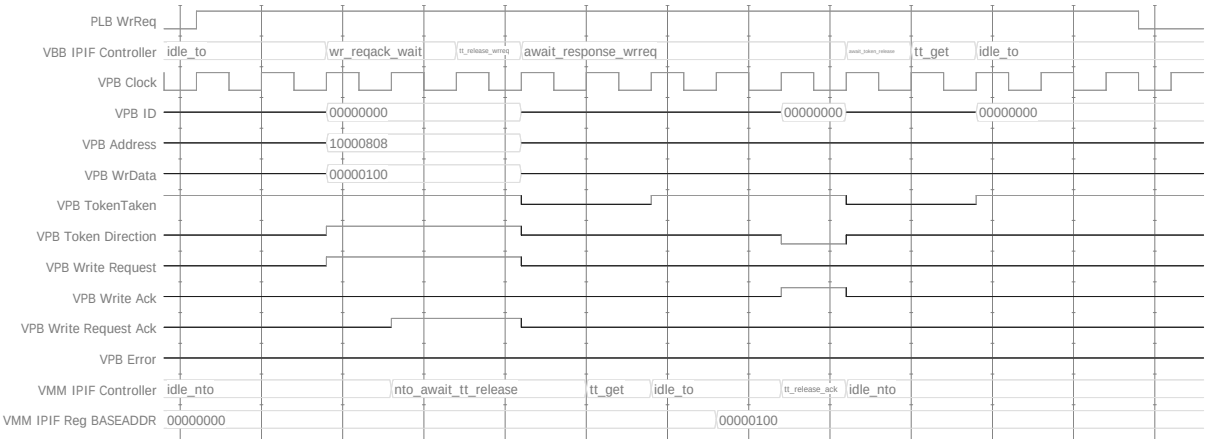


Abbildung 7.4: *Analyse der Tokenübergabe auf dem VPB*

weitergegeben. In diesem Fall leitet die VBB die Write Request Anfrage an den VPB weiter. Das betroffene Modul, welches den adressierten Konfigurationsspeicher enthält, antwortet mit einem Request Ack. Sobald dieser registriert wird, lässt die VBB das Token-Signal fallen. Das betroffene Modul registriert dies und setzt das Token von sich aus wieder auf 1. Hierbei handelt es sich in jedem Modul um eine Transceiver Schaltung, die sowohl das Signal auslesen, als auch setzen kann. Das Busprotokoll verhindert hier ein gleichzeitiges Setzen des Signals. Sobald das Modul das Token ”hat”, beginnt intern die Verarbeitung, d.h. der Wert wird in das Register geschrieben. Danach wird ein Write Ack gesetzt und parallel das Token wieder freigegeben. Eine Bestätigung durch die VBB muss nicht erfolgen, da dieses Modul die ursprüngliche Anfrage gestellt und somit definitiv vorhanden und aktiv ist. Bei der ersten Übergabe ist eine Bestätigung erforderlich um Anfragen an nicht vorhandene Module auszuschliessen (Timeout). Die VBB übernimmt das Token sofort wieder, nachdem die Richtung (Direction) des Token auf ”Zurück” (0) und das Token Signal auf 0 gesetzt wird.

Nun lässt sich durch Auslesen des Konfigurationsstrings für das DRAM Modul bestimmen, ob das Schreiben des neuen Wertes `BASEADDR` erfolgreich war:

```
0x 00000100 00000000 00000000 00000000 00000000 00000000
```

Über weitere Transaktionen wird der Konfigurationsstring nun auf einen sinnvollen Wert gesetzt, der die Einstellung für den DRAM setzt. Innerhalb dieser Simulation soll nur die Simulationszeit als Prüfwert im Vordergrund stehen, diese wird nach dem Start der Simulation über den Takt gezählt. Sobald eine simulierte Anfrage an den DRAM gestellt wird, wird die Zeit, die ein realer DRAM Baustein benötigt hätte noch hinzuaddiert. Der Overhead durch den VPB wird erst in einem Postprozess nach Abschluss der Analyse wieder herausgerechnet.

Starten der Simulation

Solange die Konfiguration nicht gestart ist, werden alle Speicheranfragen direkt an den realen DIMM weitergeleitet und nicht vorab von der Hardware simuliert. Dazu muss die Simulation zunächst gestartet werden. Dies geschiet durch Schreiben des Startbefehls `0x0000_0001` in das Befehlsregister `0x1000_0000` der VBB.

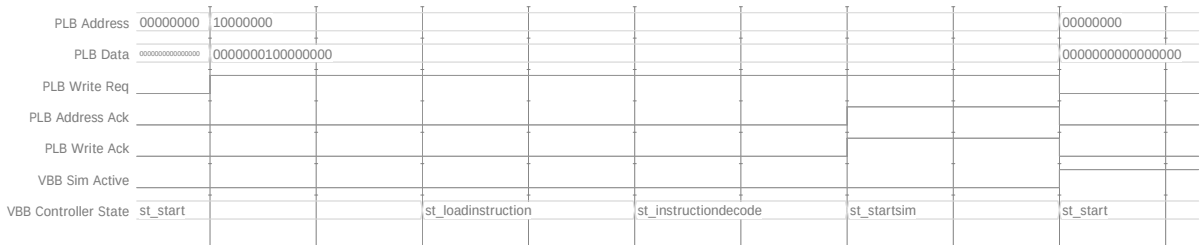


Abbildung 7.5: *Starten der Simulation durch Beschreiben des VBB Befehlsregister*

Abbildung 7.5 zeigt den Schreibvorgang in das Befehlsregister der VBB. Interessant ist hierbei der Zustand des VBB Controllers. Dieser erkennt die Adresse als die des Befehlsregisters und dekodiert den Wert des Datums online, d.h. ohne den Schreibvorgang zunächst zu bestätigen.

Das Write Ack nach der Dekodierungsphase signalisiert also auch, das der Befehl erfolgreich ausgeführt wurde. Dabei wird das interne Register **VBB Sim Active** auf 1 gesetzt.

Ab jetzt beginnt die VBB parallel die Zeit zu messen, während die Speichermodule ihre eigenen Werte bei einem Zugriff festhalten. Während der Simulation oder auch nach dem späteren Stoppen können die Werte sowohl aus der VBB als auch aus den einzelnen Modulen ausgelesen und verarbeitet werden.

Anfrage an den DRAM-Baustein

Nachdem nun die Simulation gestartet wurde soll eine fiktive Anfrage an den DRAM Baustein, die Analyse der Simulationstaktberechnung ermöglichen.

Abbildung 7.6 zeigt den Verlauf einer fiktiven Speicheranfrage an den DRAM. Zu bemerken ist hier der Verlauf der VBB State Machine. Zunächst findet die Simulation über den VPB statt (`st_vpb_simop`), erst danach wird die Anfrage an den Memory Access Baustein gelegt und auf ein Ack vom realen Speicher gewartet (`st_waitformemack`). Da in der Simulation kein realer Speicher existiert, wird die Antwort hier in der Simulationsumgebung generiert und stellt damit keine reale Antwortzeit dar.

Zur Analyse soll besonders die Berechnung der neuen Simulationszeit innerhalb des DRAM Moduls betrachtet werden. Das Signal `DRAM Cycle Counter` stellt den aktuellen Zustand des internen Simulationszähler des DRAM Moduls dar. Zunächst wird die von der VBB gezählte Vorlaufzeit (hier 32) aufgenommen und dann um den Wert, der für die Schreibzugriffsdauer gesetzt wurde, erhöht

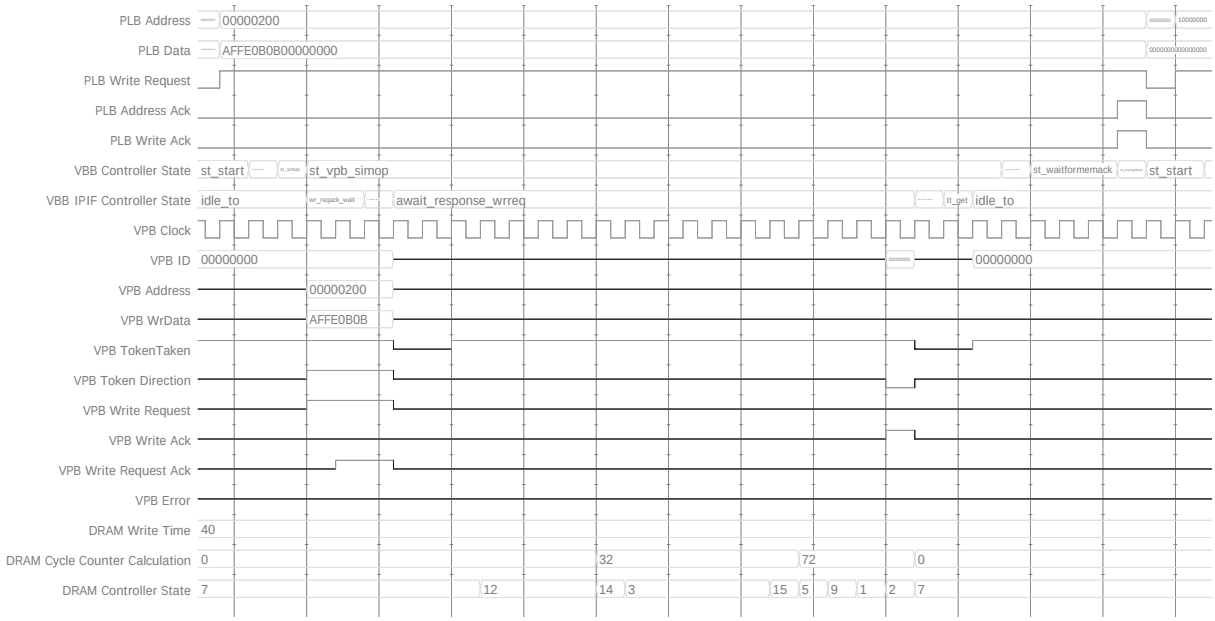


Abbildung 7.6: *Schreiban*

(Sprung um 40 auf 72). Mit Übergabe des Token zurück an die VBB wird der interne Zähler wieder auf 0 gesetzt. Während der Tokenübergabe wird auch der neue Wert an die VBB über den VPB übermittelt.

Stoppen der Simulation

Das Stoppen einer Simulation wurde ebenfalls untersucht und verläuft analog zur Startsituation. Das Befehlswort zum stoppen ist `0x0000_0000`. Die VBB hält den Simulationstaktzähler an und leitet ab sofort die Speicheranfragen direkt an den DRAM. Das Dekodieren des Befehls erfolgt analog zum Startfall und wird an dieser Stelle nicht weiter untersucht.

Lesen der Simulationsergebnisse

Auch der Fall des Lesens der Simulationsergebnisse erfolgt analog zum Schreiben der Konfiguration, d.h. es läuft die Tokenübergabe über den VPB mit Rückgabe des Ergebnisses.

Eine Besonderheit ist hier das Simulationstaktzählregister der VBB, es teilt sich auf in ein HIGH-Register `0x1000_0004` und ein LOW-Register `0x1000_0008`. Da unsere Simulation den Wertebereich des Low-Registers nicht übersteigt, reicht ein einfaches Lesen des Low-Registers aus.

Nach Auslesen des Simulationsregisters wird der Wert

`0x0000_005A` (90)

zurückgeliefert. Er entspricht exakt der errechneten 72 Takte des DRAM Controllers plus 18 weiterer Takte, die der Antwort über den PLB und einer Nachlaufzeit bis der Stop-Befehl an die VBB dekodiert wurde entsprechen.

7.3 Fazit

Dieses Kapitel sollte eine Übersicht über die durchgeführte Modelsim-Analyse liefern. Dabei lag der Schwerpunkt auf der Betrachtung eines Testfalls, der alle möglichen Situationen des V-Power Busses abdeckt. Zu beachten ist dabei, dass sich mit Modelsim eine Vielzahl von Situationen nicht testen lässt. Dazu gehören u.a. die realen Komponenten die auf dem FPGA bereitgestellt werden, wie beispielsweise der DIMM Baustein.

Für die Behebung von Fehlern und Analyse der Bausteine in Zusammenspiel mit der FPGA Hardware wurden spezielle Debug-Methoden direkt auf dem FPGA mittels sperater Messhardware entwickelt. Deren Beschreibung würde den Umfang dieses Dokumentes aber übersteigen.

Kapitel 8

Fazit und Zusammenfassung

Die Teilaufgaben der Projektgruppe **V-POWER** konzentrierten sich im Wesentlichen auf zwei Bereiche. Zum einen die Portierung des Betriebssystems Linux auf die FPGA Hardware. Hierbei stand zunächst die abstrakte Portierung des Systems auf das Board ohne die Integration der entwickelten Spezialhardware im Vordergrund. Die Anpassung an die Simulationshardware erfolgte in einem zweiten Schritt über die Treiberentwicklung.

Die zweite große Teilaufgabe beschäftigte sich mit der Umsetzung des Hardwarekonzeptes mittels VHDL. Dabei wurden zunächst kleinere Tests direkt an der FPGA Hardware durchgeführt und dann ein zusammenhängender Simulationsbaustein entwickelt, der sich nahtlos in die von der Firma Xilinx bereitgestellte Konfigurationsumgebung einbettet.

Der entstandene Entwurf wurde dann mittels einer Simulationsumgebung noch während des Entstehungsprozesses fortlaufend validiert. Somit konnten kleinere Fehler, die auch ggf. in der Konzeptphase nicht korrekt geprüft wurden, noch in der Designphase korrigiert werden.

Während sich die Portierung des Linux-Systems und der Entwicklungsprozess der Simulationshardware nebst Testcases als sehr vielversprechend und vor allem sehr stabil erwiesen, gab es bei der Installation auf dem FPGA Bord zunehmend Schwierigkeiten. Hierbei musste zum Abschluss der PG ein besonderes Hardwareprüfverfahren entwickelt werden, um Abweichungen von den Modelsim-Testcases direkt auf dem FPGA Chip zu erkennen, da hier konventielle Methoden wie Digital-Oszilloskop etc. schnell an ihre Grenzen stießen.

Die Arbeit an der FPGA Hardware gestalltete sich einerseits sehr vielseitig und abwechslungsreich (Trennung von Hard- und Softwareentwicklung). Leider stellt die FPGA Hardware aber auch eine Vielzahl von Problemen für Einsteiger dar, die sich besonders in großen Projekten als fast unüberschaubar erwiesen haben. Von kleineren Teilprojekten kann nur sehr schwer auf die Funktionsfähigkeit komplexer Projekte geschlossen werden.

Auch gibt es besondere Grenzen der Hardware, die sich erst bei größeren Entwürfen zeigen.

Trotz dieser Einschränkungen bei dieser Form des rechnergestützen Entwurfs von Mikroelektronik lässt sich in kürzester Zeit ein komplexer Hardwarebaustein ohne besonderen Overhead an Material etc. realisieren. Dabei sollte aber neben der rechnergestützten Simulation, der Entwicklungsprozess fortlaufend von Tests an der Hardware begleitet werden und vor allem das Konzept von vornherein so angelegt werden, das Tests von kleinen Teilbereich auf der Hardware auch ohne den Gesamtentwurf möglich sind.

Selbst bei Berücksichtigung dieser Empfehlung muss zudem ein erhöhter Aufwand bei der Endvalidierung mit eingeplant werden. Eine solche Abschlussphase entfällt i.d.R. bei reinen Softwareprojekten.

Anhang A

VHDL Portbeschreibungen

A.1 VPB-IPIF Ports

Signal	I/O Typ	Breite	Beschreibung
pot_vip2ip_Clk	O	1	Systemtakt vom VPB
pot_vip2ip_SimRst	O	1	Resetsignal
pot_vip2ip_SimTime	O	**C_VPB_SIMTIME_DWIDTH**	Aktuelle Simulationszeit
pot_vip2ip_SimHold	O	1	Simulation angehalten (’1’)
pot_vip2ip_Busy	O	1	FSM des VPB IPIFs ist busy (’1’)
pot_vip2ip_Addr	O	**C_VPB_AWIDTH**	Adresse
pot_vip2ip_RdData	O	**C_VPB_DWIDTH**	Gelesene Daten
pot_vip2ip_WrData	O	**C_VPB_DWIDTH**	Zu schreibende Daten
pot_vip2ip_Mst	O	1	Modul ist Master am VPB (’1’)
pot_vip2ip_CS	O	**C_VPB_AWIDTH**	Chip Select Vektor
pot_vip2ip_RdReq	O	1	Leseanfrage (’1’)
pot_vip2ip_WrReq	O	1	Schreibanfrage (’1’)
pot_vip2ip_RdAck	O	1	Lesebestätigung (’1’)
pot_vip2ip_WrAck	O	1	Schreibbestätigung (’1’)
pot_vip2ip_Error	O	1	Errorsignal
pot_vip2ip_RealBA	O	**C_VPB_AWIDTH**	Basisadresse des simulierten Speicherbereichs des Moduls
pot_vip2ip_RealHA	O	**C_VPB_AWIDTH**	Hochadresse des simulierten Speicherbereichs des Moduls
pin_ip2vip_Succ	I	1	Vorgänger (’0’)/Nachfolger(’1’) Adressierung
pin_ip2vip_Addr	I	**C_VPB_AWIDTH**	Adresse
pin_ip2vip_RdData	I	**C_VPB_DWIDTH**	Gelesene Daten
pin_ip2vip_WrData	I	**C_VPB_DWIDTH**	Zu schreibende Daten
pin_ip2vip_RdReq	I	1	Leseanfrage (’1’)
pin_ip2vip_WrReq	I	1	Schreibanfrage (’1’)
pin_ip2vip_RdAck	I	1	Lesebestätigung (’1’)
pin_ip2vip_WrAck	I	1	Schreibbestätigung (’1’)
pin_ip2vip_Error	I	1	Errorsignal
pin_ip2vip_SimTime	I	**C_VPB_SIMTIME_DWIDTH**	Aktuelle Simulationszeit

Signal	I/O Typ	Breite	Beschreibung
pin_ip2vip_SimHold	I	1	Simulation angehalten (’1’)
pin_ip2vip_SimRst	I	1	Resetsignal
pin_vpb2vip_Clk	I	1	Systemtakt
pin_vpb2vip_SimRst	I	1	Resetsignal
pin_vpb2vip_SimTime	I	**C_VPB_SIMTIME_DWIDTH**	Aktuelle Simulationszeit
pin_vpb2vip_SimHold	I	1	Simulation angehalten (’1’)
pin_vpb2vip_Id	I	**C_VPB_ID_WIDTH**	Id des aktiven Moduls
pin_vpb2vip_Addr	I	**C_VPB_AWIDTH**	Adresse
pin_vpb2vip_RdData	I	**C_VPB_DWIDTH**	Gelesene Daten
pin_vpb2vip_WrData	I	**C_VPB_DWIDTH**	Zu schreibende Daten
pin_vpb2vip_Tt	I	1	Token-Signal
pin_vpb2vip_TtDir	I	1	Tokenübergabe-Richtung (Vorgänger ’0’, Nachfolger ’1’)
pin_vpb2vip_RdReq	I	1	Leseanfrage (’1’)
pin_vpb2vip_WrReq	I	1	Schreibanfrage (’1’)
pin_vpb2vip_RdAck	I	1	Lesebestätigung (’1’)
pin_vpb2vip_WrAck	I	1	Schreibbestätigung (’1’)
pin_vpb2vip_RdReqAck	I	1	Leseanfragen-Bestätigung
pin_vpb2vip_WrReqAck	I	1	Schreibanfragen-Bestätigung
pin_vpb2vip_Error	I	1	Errorsignal
pot_vip2vpb_Id	O	**C_VPB_ID_DWIDTH**	Id des aktiven Moduls
pot_vip2vpb_Addr	O	**C_VPB_AWIDTH**	Adresse
pot_vip2vpb_RdData	O	**C_VPB_DWIDTH**	Gelesene Daten
pot_vip2vpb_WrData	O	**C_VPB_DWIDTH**	Zu schreibende Daten
pot_vip2vpb_Tt	O	1	Token-Signal
pot_vip2vpb_TtDir	O	1	Tokenübergabe-Richtung (Vorgänger ’0’, Nachfolger ’1’)
pot_vip2vpb_RdReq	O	1	Leseanfrage (’1’)
pot_vip2vpb_WrReq	O	1	Schreibanfrage (’1’)
pot_vip2vpb_RdAck	O	1	Lesebestätigung (’1’)
pot_vip2vpb_WrAck	O	1	Schreibbestätigung (’1’)
pot_vip2vpb_RdReqAck	O	1	Leseanfragen-Bestätigung
pot_vip2vpb_WrReqAck	O	1	Schreibanfragen-Bestätigung
pot_vip2vpb_Error	O	1	Errorsignal
pot_vip2vpb_SimTime	O	**C_VPB_SIMTIME_DWIDTH**	Aktuelle Simulationszeit
pot_vip2vpb_SimHold	O	1	Simulation angehalten (’1’)
pot_vip2vpb_SimRst	O	1	Resetsignal

Tabelle A.1: *VPB-IPIF Ports*

A.2 VBB Ports

Signal	I/O Typ	Breite	Beschreibung
pin_plb2vbb_Rst	I	1	Reset Signal vom PLB
pin_plb2vbb_Clk	I	1	Systemtakt

Signal	I/O Typ	Breite	Beschreibung
pin_plb2vbb_CS	I	2	Chip Select Vektor
pin_plb2vbb_CE	I	2	Chip Enable Vektor
pin_plb2vbb_RdCE	I	2	Chip Enable Vektor für lesende Zugriffe
pin_plb2vbb_WrCE	I	2	Chip Enable Vektor für schreibende Zugriffe
pin_plb2vbb_Addr	I	32	Adresse
pin_plb2vbb_Data	I	64	Daten
pin_plb2vbb_RdReq	I	1	Leseanfrage
pin_plb2vbb_WrReq	I	1	Schreibanfrage
pin_plb2vbb_BE	I	8	Byte Enable Vektor
pin_plb2vbb_RNW	I	1	Read Not Write
pin_plb2vbb_Burst	I	1	Burst Zugriff
pin_plb2vbb_IBurst	I	1	IBurst Zugriff
pot_vbb2plb_AddrAck	O	1	Adressbestätigung
pot_vbb2plb_RdAck	O	1	Lesebestätigung
pot_vbb2plb_WrAck	O	1	Schreibbestätigung
pot_vbb2plb_Busy	O	1	Signal wenn Beschäftigt
pot_vbb2plb_Error	O	1	Signal bei Fehler
pot_vbb2plb_Retry	O	1	Signal zum erneuten Versuch
pot_vbb2plb_ToutSup	O	1	Timeout Supression
pot_vbb2plb_Data	O	64	Daten
pot_vbb2plb_IntrEvent	O	3	Interrupt Ereignis
pin_mem2vbb_AddrAck	I	1	Adressbestätigung
pin_mem2vbb_RdAck	I	1	Lesebestätigung
pin_mem2vbb_WrAck	I	1	Schreibbestätigung
pin_mem2vbb_Busy	I	1	Signal wenn Beschäftigt
pin_mem2vbb_Error	I	1	Signal bei Fehler
pin_mem2vbb_Retry	I	1	Signal zum erneuten Versuch
pin_mem2vbb_ToutSup	I	1	Timeout Supression
pin_mem2vbb_Data	I	64	Daten
pin_mem2vbb_IntrEvent	I	3	Interrupt Ereignis
pot_vbb2mem_Rst	O	1	Reset Signal vom PLB
pot_vbb2mem_Clk	O	1	Systemtakt
pot_vbb2mem_CS	O	2	Chip Select Vektor
pot_vbb2mem_CE	O	2	Chip Enable Vektor
pot_vbb2mem_RdCE	O	2	Chip Enable Vektor für lesende Zugriffe
pot_vbb2mem_WrCE	O	2	Chip Enable Vektor für schreibende Zugriffe
pot_vbb2mem_Addr	O	32	Adresse
pot_vbb2mem_Data	O	64	Daten
pot_vbb2mem_RdReq	O	1	Leseanfrage
pot_vbb2mem_WrReq	O	1	Schreibanfrage
pot_vbb2mem_BE	O	8	Byte Enable Vektor
pot_vbb2mem_RNW	O	1	Read Not Write
pot_vbb2mem_Burst	O	1	Burst Zugriff
pot_vbb2mem_IBurst	O	1	IBurst Zugriff
pin_pcu2vbb_PC_Stoped	I	1	CPU wurde gestoppt
pot_vbb2pcu_PC_Stop	O	1	CPU stoppen
pin_vpb2vbb_Clk	I	1	Systemtakt vom VPB
pin_vpb2vbb_SimRst	I	1	Reset Signal
pin_vpb2vbb_SimTime	I	64	Aktuelle Simulationszeit

Signal	I/O Typ	Breite	Beschreibung
pin_vpb2vbb_SimHold	I	1	Simulation angehalten
pin_vpb2vbb_Id	I	32	ID des aktiven Moduls
pin_vpb2vbb_Addr	I	32	Adresse
pin_vpb2vbb_RdData	I	64	Gelesene Daten
pin_vpb2vbb_WrData	I	64	Zu Schreibende Daten
pin_vpb2vbb_Tt	I	1	Token Signal
pin_vpb2vbb_TtDir	I	1	Richtung für Tokenübergabe
pin_vpb2vbb_RdReq	I	1	Leseanfrage
pin_vpb2vbb_WrReq	I	1	Schreibanfrage
pin_vpb2vbb_RdAck	I	1	Lesebestätigung
pin_vpb2vbb_WrAck	I	1	Schreibbestätigung
pin_vpb2vbb_RdReqAck	I	1	Leseanfragebestätigung
pin_vpb2vbb_WrReqAck	I	1	Schreibanfragebestätigung
pin_vpb2vbb_Error	I	1	Signal bei Fehler
pot_vbb2vpb_SimRst	O	1	Reset Signal
pot_vbb2vpb_SimTime	O	64	Aktuelle Simulationszeit
pot_vbb2vpb_SimHold	O	1	Simulation angehalten
pot_vbb2vpb_Id	O	32	ID des aktiven Moduls
pot_vbb2vpb_Addr	O	32	Adresse
pot_vbb2vpb_RdData	O	64	Gelesene Daten
pot_vbb2vpb_WrData	O	64	Zu Schreibende Daten
pot_vbb2vpb_Tt	O	1	Token Signal
pot_vbb2vpb_TtDir	O	1	Richtung für Tokenübergabe
pot_vbb2vpb_RdReq	O	1	Leseanfrage
pot_vbb2vpb_WrReq	O	1	Schreibanfrage
pot_vbb2vpb_RdAck	O	1	Lesebestätigung
pot_vbb2vpb_WrAck	O	1	Schreibbestätigung
pot_vbb2vpb_RdReqAck	O	1	Leseanfragebestätigung
pot_vbb2vpb_WrReqAck	O	1	Schreibanfragebestätigung
pot_vbb2vpb_Error	O	1	Signal bei Fehler

Tabelle A.2: *Portmap der VBB*

A.3 Speicherzugriffsmodul Ports

Signal	I/O Typ	Breite	Beschreibung
pin_Clk	I	1	Taktsignal
pin_Rst	I	1	Reset-Signal
pin_CS	I	1	Modul aktivieren
pin_Addr	I	**C_AWIDTH**	Speicheradresse des ersten Datums
pin_Data	I	**C_DWIDTH**	Datum, das in den Speicher geschrieben werden soll

Signal	I/O Typ	Breite	Beschreibung
pin_RNW	I	1	Read Not Write-Signal, zur Unterscheidung von Lese- ('1') und Schreibzugriffen ('0')
pin_Req	I	1	Ausführung einer Speicheroperation anfordern
pin_Mode	I	2	Operationsart
pin_DblWord	I	1	die übertragenden Daten sind Wörter ('0') oder Doppelwörter ('1')
pin_Quantity	I	4	Anzahl der Daten **-1** in Binärdarstellung
pot_Data	O	**C_DWIDTH**	Datum, das aus dem Speicher gelesen wurde
pot_Done	O	1	signalisiert, dass die Speicheroperation erfolgreich durchgeführt wurde
pot_Error	O	1	signalisiert, ob bei einer Speicheroperation ein Fehler auftrat
pot_mem_CS	O	2	Chip Select Vektor
pot_mem_CE	O	2	Chip Enable Vektor
pot_mem_RdCE	O	2	Chip Enable Vektor für lesende Zugriffe
pot_mem_WrCE	O	2	Chip Enable Vektor für schreibende Zugriffe
pot_mem_Addr	O	32	Speicheradresse des aktuellen Datums
pot_mem_Data	O	64	Datum, das in den Speicher geschrieben werden soll
pot_mem_RdReq	O	1	Ausführung einer Leseoperation anfordern
pot_mem_WrReq	O	1	Ausführung einer Schreiboperation anfordern
pot_mem_BE	O	8	Byte Enable Vektor, Auswahl der Byte Lanes zur Übertagung der Daten
pot_mem_RNW	O	1	Read Not Write-Signal, zur Unterscheidung von Lese- ('1') und Schreibzugriffen ('0')
pot_mem_Burst	O	1	Burst Zugriff durchführen
pot_mem_lBurst	O	1	Burst Zugriff hat variable Länge
pin_mem2vbb_AddrAck	I	1	Bestätigung der Adressübernahme
pin_mem2vbb_RdAck	I	1	signalisiert, dass ein Datenwort gelesen wurde
pin_mem2vbb_WrAck	I	1	signalisiert, dass ein Datenwort geschrieben wurde
pin_mem2vbb_Busy	I	1	signalisiert, dass zur Zeit keine Operation ausgeführt werden kann
pin_mem2vbb_Error	I	1	signalisiert, ob bei einer Speicheroperation ein Fehler auftrat
pin_mem2vbb_Retry	I	1	signalisiert, dass eine erneute Ausführung der Operation versucht wird
pin_mem2vbb_ToutSup	I	1	Timeout Supression
pin_mem2vbb_Data	I	64	Datum, das aus dem Speicher gelesen wurde

Tabelle A.3: *Ports des Speicherzugriffsmoduls*

A.4 Cache Ports

Signal	I/O Typ	Breite	Beschreibung
pin_Clk	I	1	Takt
cacheIn_addr	I	32	Adresse, die zu simulieren ist
cacheIn_rdReq	I	1	1, falls es sich um einen Read Request handelt
cacheIn_wrReq	I	1	1, falls es sich um einen Write Request handelt
cacheIn_wrAck	I	1	Write Ack. Wird benötigt, falls ein Write Request an einen anderen Speicherbaustein gesendet wurde
cacheIn_rdAck	I	1	Read Ack. Wird benötigt, falls ein Read Request an einen anderen Speicherbaustein gesendet wurde
cacheIn_rst	I	1	Setzt alle Zählregister auf 0 zurück
cacheIn_hold	I	1	Hält die laufende Simulation an
cacheIn_config	I	128	Hält alle Konfigurationsdaten bereit. Aufbau des Strings siehe Tabelle 4.1
cacheIn_simTime	I	64	Aktuelle Simulationszeit vom Bus
cacheIn_busy	I	1	Zeigt an, ob das VPB_IF nicht im Idle ist
cacheIn_cs	I	1	chip_select: Cache Modul nimmt nur Request entgegen, wenn es im Idle Zustand ist und cache_cs = 1 anliegt
cacheIn_error	I	1	Für Fehler vom VPB_IF
cacheOut_addr	O	32	Adresse, die zu simulieren ist (für andere Bausteine)
cacheOut_rdReq	O	1	Read Request = 1, falls ein Lesezugriff ausgeführt werden soll
cacheOut_wrReq	O	1	Write Request = 1, falls ein Schreibzugriff ausgeführt werden soll
cacheOut_rdAck	O	1	Read Ack zum Bestätigen eines erfolgreichen Lesezyklus
cacheOut_wrAck	O	1	Write Ack zum Bestätigen eines erfolgreichen Schreibzyklus
cacheOut_succ	O	1	Gibt die Richtung des nächsten Speicherbausteins an (Nachfolger = 0, Vorgänger = 1)
cacheOut_error	O	1	Gibt ggf. einen Fehler zurück
cacheOut_simTime	O	64	Aktuelle Simulationszeit für den Bus
cacheOut_simResult	O	384	Bit Array, dass alle Simulationsergebnisse nach außen führt (siehe Tabelle 4.2)

Tabelle A.4: *Portmap des Cache Moduls*

A.5 DRAM Ports

Signal	I/O Typ	Breite	Beschreibung
pin_Clk	I	1	Takt

Signal	I/O Typ	Breite	Beschreibung
dramIn_addr	I	32	Adresse, die zu simulieren ist
dramIn_rdReq	I	1	'1', falls es sich um einen Read Request handelt
dramIn_wrReq	I	1	'1', falls es sich um einen Write Request handelt
dramIn_rst	I	1	Setzt alle Zählregister auf '0' zurück
dramIn_hold	I	1	Hält die laufende Simulation an
dramIn_config	I	192	Hält alle Konfigurationsdaten bereit. Aufbau des Strings siehe Tabelle 4.3
dramIn_simTime	I	64	Liefert die aktuelle Simulationszeit
dramIn_cs	I	1	chip_select: DRAM Modul nimmt nur Request entgegen, wenn es im Idle Zustand ist und dram_cs = '1' anliegt
dramIn_error	I	1	Zeigt Fehler vom VPB_IF an
dramOut_rdAck	O	1	Read Ack zum Bestätigen eines erfolgreichen Lesezyklus
dramOut_wrAck	O	1	Write Ack zum Bestätigen eines erfolgreichen Schreibzyklus
dramOut_succ	O	1	'1' bei Adressierung des Nachfolgers, '0' zur Adressierung des Vorgängers
dramOut_error	O	1	'1' wenn ein Fehler aufgetreten ist
dramOut_simTime	O	64	Liefert die neue aktuelle Simulationszeit
dramOut_simResult	O	288	Simulationsergebnisse siehe Tabelle 4.4

Tabelle A.5: *Portmap des DRAM Moduls*

A.6 Scratchpad Ports

Signal	I/O Typ	Breite	Beschreibung
pin_Clk	I	1	Takt
sramIn_rdReq	I	1	'1', falls es sich um einen Read Request handelt
sramIn_wrReq	I	1	'1', falls es sich um einen Write Request handelt
sramIn_rst	I	1	Setzt alle Zählregister zurück
sramIn_hold	I	1	Hält die laufende Simulation an
sramIn_config	I	64	Hält alle Konfigurationsdaten bereit. Aufbau des Strings siehe 4.5
sramIn_simTime	I	64	Liefert die aktuelle Simulationszeit
sramIn_cs	I	1	chip_select: Modul nimmt nur Request entgegen, wenn es im Idle Zustand ist und dram_cs = '1' anliegt
sramIn_error	I	1	Für Fehler vom VPB_IF
sramOut_rdAck	O	1	Read Ack zum Bestätigen eines erfolgreichen Lesezyklus
sramOut_wrAck	O	1	Write Ack zum Bestätigen eines erfolgreichen Schreibzyklus

Signal	I/O Typ	Breite	Beschreibung
sramOut_succ	O	1	'1' bei Adressierung des Nachfolgers, '0' zur Adressierung des Vorgängers
sramOut_error	O	1	'1' wenn ein Fehler aufgetreten ist
sramOut_simTime	O	64	Liefert die neue Simulationszeit
sramOut_simResult	O	192	Bit Array, dass alle Simulationsergebnisse nach außen führt siehe 4.6

Tabelle A.6: *Portmap des Scratch Pad Moduls*

A.7 Flash Ports

Signal	I/O Typ	Breite	Beschreibung
flashIn_rdReq	I	1	1, falls es sich um einen Read Request handelt
flashIn_rst	I	1	Setzt alle Zählregister auf 0 zurück
flashIn_hold	I	1	Hält die laufende Simulation an
flashIn_config	I	32	Hält alle Konfigurationsdaten bereit. Aufbau des Strings siehe Tabelle 4.7
flashIn_simTime	I	64	Aktuelle Simulationszeit vom Bus
flashIn_busy	I	1	Zeigt an, ob das VPB_IF nicht im Idle ist
flashIn_cs	I	1	chip_select: Flash Modul nimmt nur Request entgegen, wenn es im Idle Zustand ist und flash_cs = 1 anliegt
flashIn_error	I	1	Für Fehler vom VPB_IF
flashOut_rdAck	O	1	Read Ack zum Bestätigen eines erfolgreichen Lesezyklus
flashOut_succ	O	1	Gibt die Richtung des nächsten Speicherbausteins an (Nachfolger = 0, Vorgänger = 1)
flashOut_error	O	1	Gibt ggf. einen Fehler zurück
flashOut_simTime	O	64	Aktuelle Simulationszeit für den Bus
flashOut_simResult	O	128	Bit Array, dass alle Simulationsergebnisse nach außen führt (siehe Tabelle 4.8)

Tabelle A.7: *Portmap des Flash Moduls*

Anhang B

RTL Schemata

B.1 RTL Schema der VBB

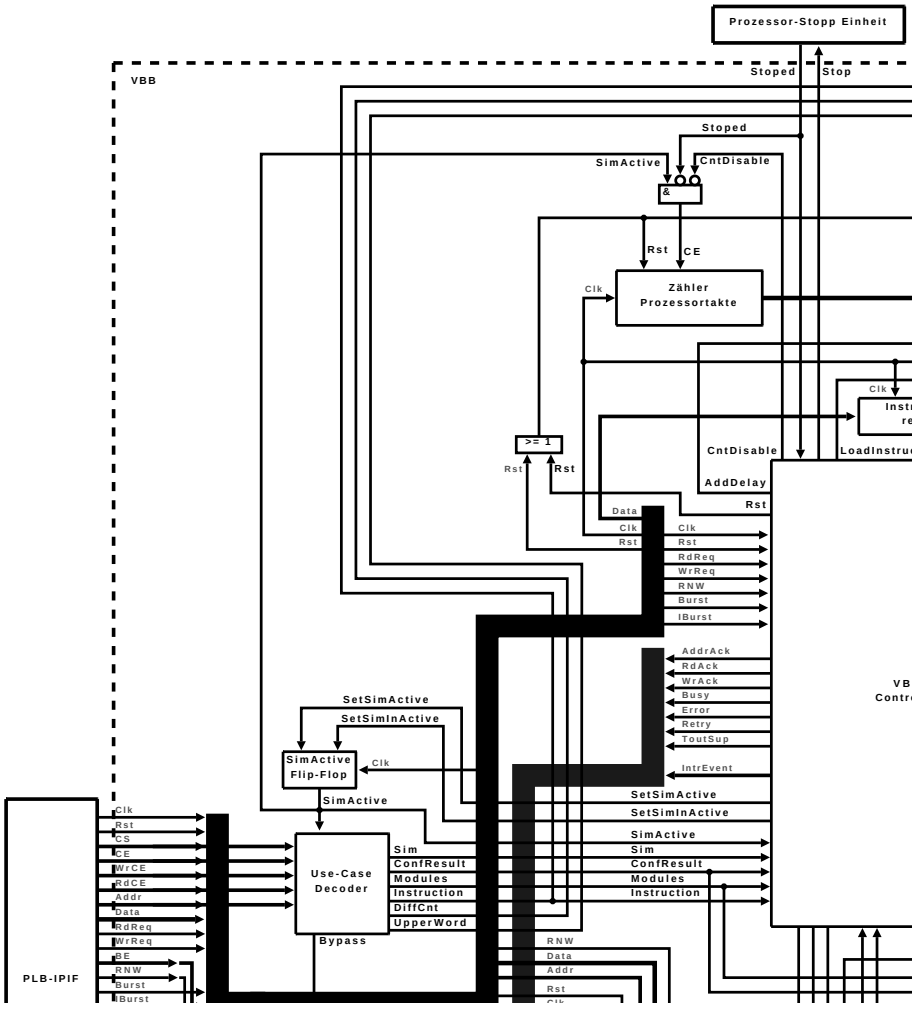


Abbildung B.1: *Auschnitt des RTL Schemas der VBB links oben*

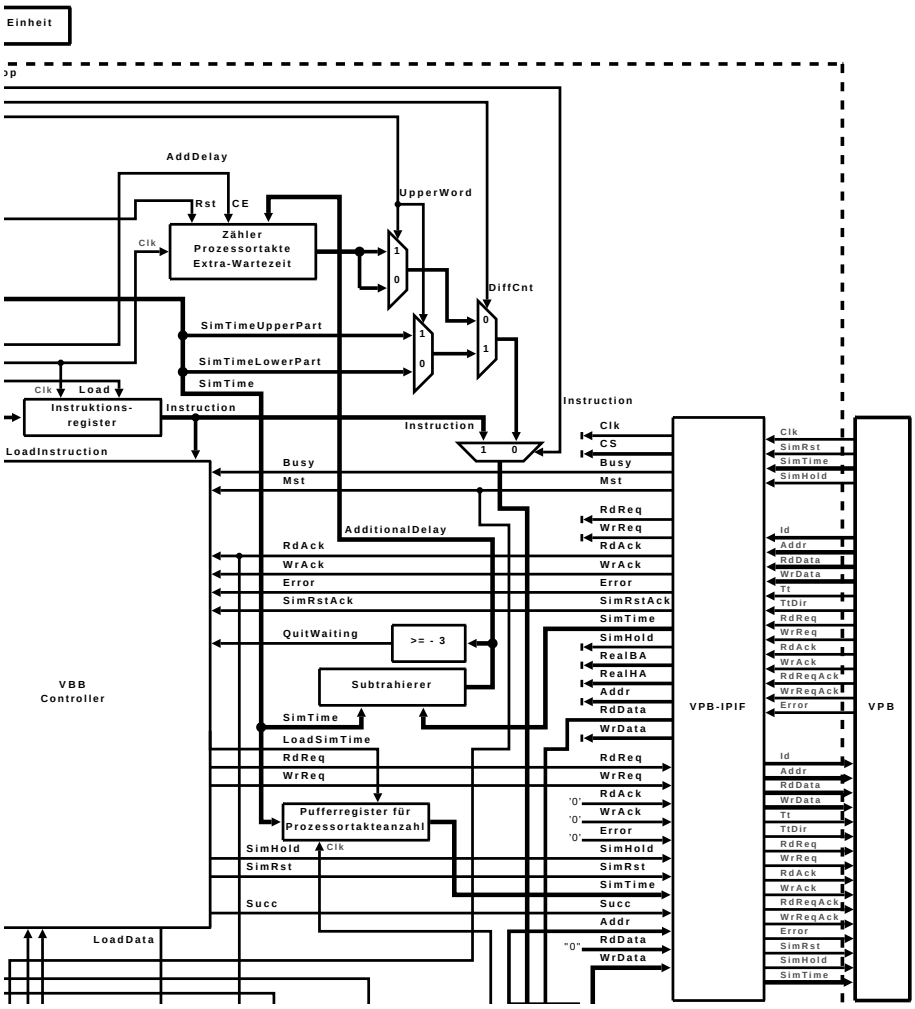


Abbildung B.2: *Auschnitt des RTL Schemas der VBB rechts oben*

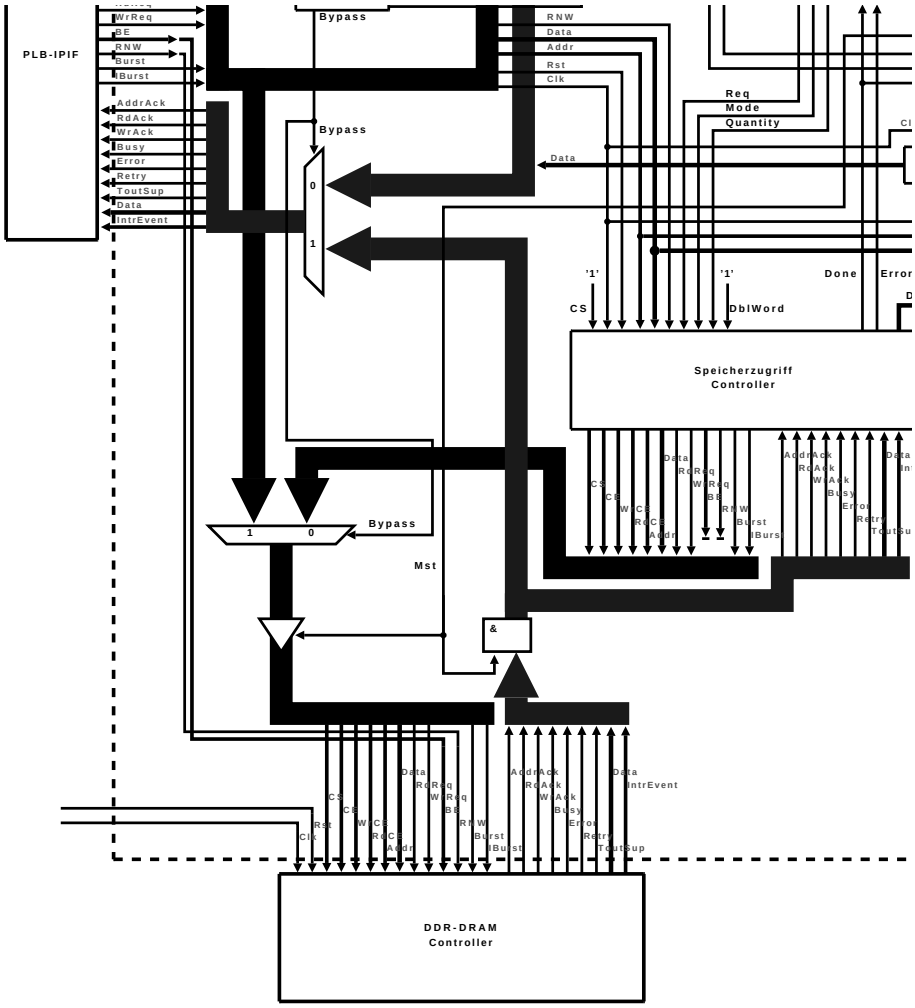


Abbildung B.3: *Auschnitt des RTL Schemas der VBB unten links*

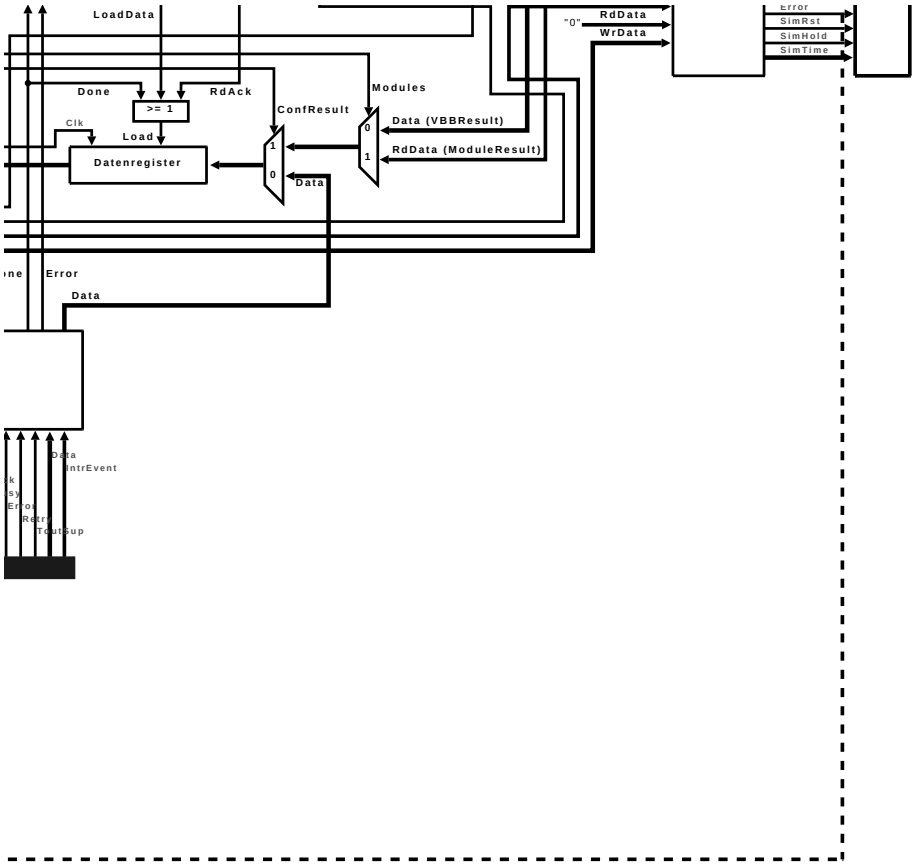


Abbildung B.4: *Auschnitt des RTL Schemas der VBB unten rechts*

Anhang C

Quelltexte

C.1 Beispielquelltext zur Nutzung des V-Power Kerneltreibers

```
1
// Öffnen des V-Power Devices
int fd = vpower_open();

5
// Erstellen einer leeren Konfiguration
// Da alle Speicherstellen auf 0 gesetzt werden,
// sind zunächst alle Speicher deaktiviert
vpower_conf_t conf;
9 memset( &conf, 0, sizeof( conf ) );

// Konfigurieren eines DRAM Speichers
conf.dram[0].common.enabled = 0x1;
13 conf.dram[0].common.baseaddr = 0x04000000;
conf.dram[0].common.highaddr = 0x04002000;

// Laden der Konfiguration in die Hardware
17 vpower_set_config ( fd, &conf );

// Einbinden alle konfigurierten Speicher
vpower_map_devices( fd );
21
// Initialisieren einer Variable im eingebundenen Speicher
unsigned int *test = (unsigned int *) 0x04000000

25 // Starten der Simulation
vpower_start_sim( fd );

// Durchführen einer beliebigen Aktion auf der Variable
29 for (int i = 0; i <= 10000; i++) {
  *test++;
}

33 // Stoppen der Simulation
vpower_stop_sim( fd );

// Abfrage der benötigen CPU Takte
37 vpower_sim_clock_t clocks = vpower_get_sim_clock( fd );

// Simulationsergebnisse auslesen
vpower_sim_results_t sim;
41 vpower_get_sim_results( fd, &sim );

//  Schließen des Devices
vpower_close( fd );
```

Listing C.1: *Beispielquelltext zur Nutzung des V-Power Kerneltreibers*

C.2 Performance-Testprogramm

```
#include "xio.h"
#include <string.h>
#include <stdlib.h>

// Demo Parameter
// Anfangsadresse des VGA Speichers, irgendein freier Speicherbereich
#define FRAME_BASEADDR   0x00000000

#define FRAME_WIDTH      640
#define FRAME_HEIGHT     512

// Zeilenbreite des VGA IP-Cores ist immer 1024
#define IPCORE_WIDTH     1024

// Adressen für die zwei Register des VGA Framebuffers
// REGISTER_ADDR: Anfangsadresse des VGA Speichers
// REGISTER_ON: Der Bildschirm wird an/aus geschaltet
// Die Adressen sind folgendem Schema unterworfen:
// REGISTER_ADDR = opb2dcr_bridge_0 BasisAdresse
//               + (VGA_FrameBuffer DCR_BaseAddr * 4)
// REGISTER_ON   = opb2dcr_bridge_0 BasisAdresse
//               + (VGA_FrameBuffer DCR_HighAddr * 4)

#define REGISTER_ADDR    0x40000200
#define REGISTER_ON      0x40000204

void read_image(int baseaddr, unsigned char r, ´
                unsigned char g, unsigned char b)
{
  int *tmp = baseaddr;
  int *max = tmp + IPCORE_WIDTH * FRAME_HEIGHT;

  // Bildschirmspeicher mit Farbe (r,g,b) füllen
  for (; tmp < max; tmp++)
  {
    *tmp = ((((r << 8) | g) << 8) | b);
  }
}

// Erzeuge verschiedenfarbige Einzelbilder
void step_images() {

  struct rgb
  {
    unsigned char pad;
    unsigned char r;
    unsigned char g;
    unsigned char b;
  };

  unsigned int color = 0xffffffff;
  unsigned char r, g, b;

  while(1)
  {
    read_image(FRAME_BASEADDR, r, g, b);

    if (color == 0) color = 0xffffffff;
    else color--;

    r = ((struct rgb *) &color)->r;
    g = ((struct rgb *) &color)->g;
    b = ((struct rgb *) &color)->b;
  }
}

int main()
{
  print("\r\nProgram running.\n\r");
  read_image(FRAME_BASEADDR, 0, 0, 0);

  // Setze Basisadresse des Grafikkartenspeichers
  XIo_Out32(REGISTER_ADDR, FRAME_BASEADDR);

  // Schalte Bildschirm an
```

```
  XIo_Out32(REGISTER_ON, 0x1);

79  step_images();
  return 0;
}
```

Listing C.2: *Quelltext des Performance Testprogramms*

Anhang D

Portierung des MPlayer als Benchmarksoftware

D.1 Der MPlayer

MPlayer ist eine plattformübergreifende Videoabspielsoftware für Unix. Die Codecs, die zum Abspielen aller gängigen Formate notwendig sind, bringt der Player bereits mit. Der Quellcode ist frei verfügbar. Die eigentliche Darstellung auf dem Bildschirm und die Ausgabe des Tons wird über Ausgabe-Plugins realisiert. Je nach den Möglichkeiten des Betriebssystems kann so die geeignete Ausgabevariante gewählt werden. Unter Unix sind die gängigen Video-Plugins X11 (unbeschleunigte Wiedergabe unter X), xv (beschleunigte Wiedergabe unter X), sdl (Wiedergabe über das Simple Direct Media-Framework) und fbdev (Wiedergabe auf einer Framebufferkonsole). Die Audioausgabe kann unter Linux über oss (Open Sound System), alsa (Advanced Linux Sound Architecture) oder null (keine Ausgabe) erfolgen. Interessant sind für uns auch die vielfältigen Konfigurationsmöglichkeiten, die als Stellschrauben zur Verfügung stehen. Beispielsweise lässt sich die Ausgabe von Audio und Video gänzlich unterbinden (jeweils indem das Ausgabeplugin null gewählt wird).

Für uns kommt nur fbdev in Frage, da ein X-Server nur unnötig zusätzlichen Overhead und Platzbedarf bedeutet und ohnehin keine Beschleunigung durch Hardware und Treiber zur Verfügung steht. Da die Audioausgabe nicht benötigt wird, kommt hier das Ausgabeplugin null zum Einsatz.

Der MPlayer ist in voller Montur für eingebettete Systeme aufgrund seiner Größe weniger geeignet. Allerdings lässt sich die Anwendung an den individuellen Zweck bereits beim Kompilieren sehr leicht anpassen und auch abspecken.

Vorteile des MPlayers:

1. Unterstützt (unbeschleunigte) Wiedergabe auf der Framebufferkonsole

2. Eine Fülle von Konfigurationseinstellungen
3. Leicht an unterschiedliche Zwecke anzupassen

Da für PowerPC keine Binaries zur Verfügung gestellt werden, bleibt also nur den Code selbst zu übersetzen. MPlayer-1.0pre8 lässt sich problemlos wie folgt übersetzen:

```
tar xzf MPlayer-1.0pre8.tar.bz2
cd MPlayer-1.0pre8
./configure
make
```

Die einfachste Möglichkeit, unnötige Dinge zu entfernen besteht darin, beim Aufruf von ./configure die gewünschten Optionen explizit anzugeben und alle anderen Optionen abzuschalten. Dies geschieht über Parameter, die dem configure-Skript mit übergeben werden. In unserem Fall ist Crosstool zu bemühen, um die Übersetzung für die Zielarchitektur PowerPC vorzunehmen.

Folgende Optionen haben wir verwendet (einige Optionen sind redundant, da sie auch automatisch erkannt werden):

```
./configure \
--enable-linux-devfs --enable-fbdev \
--enable-cross-compile --enable-static \
--cc=powerpc-405-linux-gnu-gcc \
--as=powerpc-405-linux-gnu-as \
--target=ppc \
--host-cc=/usr/bin/gcc \
--enable-big-endian \
--disable-mencoder --disable-iconv --disable-langinfo --disable-vm \
--disable-xf86keysym --disable-tv --disable-tv-bsdbt848 \
--disable-network --disable-dvdread --disable-mpdvdkit \
--disable-cdparanoia --disable-freetype --disable-fontconfig \
--disable-unrarlib --disable-sortsub --disable-enca --disable-macosx \
--disable-inet6 --disable-gethostbyname2 --disable-ftp \
--disable-vstream --disable-win32 --disable-qtx --disable-xanim \
--disable-real --disable-x264 --disable-libpostproc \
--disable-libpostproc_so --disable-speex --disable-libdv \
--disable-toolame --disable-twolame --disable-musepack \
--disable-amr_nb --disable-amr_nb-fixed --disable-amr_wb \
--disable-tga --disable-pnm --disable-md5sum --disable-alsa \
--disable-internal-tremor --disable-arts --disable-esd \
--disable-polyp --disable-jack --disable-openal --disable-nas \
--disable-sgiaudio --disable-sunaudio
```

Die wichtigsten dieser Optionen erklärt die unten stehende Tabelle.

Option	Wirkung
–enable-linux-devfs	benutze das Standarddateisystem für Geräte unter Linux Kernel 2.4
–enable-fbdev	baue Unterstützung für das Ausgabeplugin fbdev ein
–enable-static	linke statisch gegen sämtliche Bibliotheken
–enable-cross-compile	verwende Crosscompiler
–cc	hier den Ort/Namen des Crosscompilers angeben
–as	der zu letzterem zugehörige Assembler
–enable-big-endian	der erzeugte Binärcode soll auf einer Big Endian Architektur laufen
–host-cc	der Compiler der Maschine, die den Code übersetzt
–disable-*	lasse Option * weg

Tabelle D.1: *Wichtige configure-Optionen*

Die Ausführung von make erzeugt ein ca. 5,5MB großes Binary namens mplayer, welches alles Notwendige für die Wiedergabe eines MPEG-1 Videos enthält. Mit strip (entfernt unnötige Symbole aus dem Binärcode) kann die Größe auf 5,1MB reduziert werden. Das fertige Binary wird auf der 2. Partition der CompactFlash-Karte platziert.

D.2 Video abspielen

Das Abspielen eines Videos mit dem MPlayer erfolgt durch den Befehl mplayer gefolgt von dem Videonamen auf der Kommandozeile. Allerdings werden noch weitere Konfigurationseinstellungen vorgenommen, um die Ausgabe zu beschleunigen. Diese Einstellungen können entweder als Parameter beim Aufruf übergeben werden oder in der Datei .mplayer/config im Heimatverzeichnis des verwendeten Benutzers eingetragen werden. Wir haben uns für letztere Möglichkeit entschieden, da sie den eigentlichen Aufruf des Programms vereinfacht.

Option	Wirkung
vo=fbdev:/dev/fb0	gibt das Video auf der Framebufferkonsole aus
screenw=640	setzt die Anzahl der Pixel pro Zeile auf der Framebufferkonsole
screenh=480	setzt die Anzahl der Zeilen der Framebufferkonsole
bpp=32	setzt die Farbtiefe auf 32 Bit
nosound=1	unterbindet die Dekodierung und Ausgabe der Audiospur
really-quiet=1	schaltet die Ausgabe unnötiger Debuginformationen auf der Konsole aus

Tabelle D.2: *Verwendete Wiedergabeoptionen des MPlayers*

D.3 Performance

Mit diesen Einstellungen lief ein 320 x 240 Video (MPEG-1) ruckelfrei bei etwa 30fps. Bei deaktiviertem L1-Cache reduzierte sich die Anzahl der Bilder pro Sekunde auf 3-4.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] P. Marwedel L. Wehmeyer M. Verma S. Steinke U. Helmig. Fast, predictable and low energy memory references through architecture-aware compilation. In *ASP DAC*, 2004.

[2] P. Marwedel M. Verma, L. Wehmeyer. Cache Aware Scratchpad Allocation. In *DATE*, 2004.

[3] S. Steinke. *Untersuchung des Energieeinsparungspotenzials in eingebetteten Systemen durch energieoptimierende Compilertechnik*. PhD thesis, Universität Dortmund, 2003.

[4] L. Wehmeyer. *Efficient and Predictable Memory Accesses - Optimization algorithms for memory architecture aware compilation*. PhD thesis, Universität Dortmund, 2005.

[5] Xilinx Inc. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, Document No. DS083 (v4.5) edition, 2005. Available from: http://www.xilinx.com/univ/xupv2p.html.

[6] Xilinx Inc. *Platform Studio Documentation*, 11 2005. EDK 8.1i. Available from: http://www.xilinx.com/edk.

[7] Xilinx Inc. Xilinx Core Connect IP Core Documentation. Available from: http://www.xilinx.com/ipcenter/processor_central/coreconnect/coreconnect.htm.

[8] Xilinx Inc. Processor Local Bus (PLB) v3.4, 07 2003. DS400 (v1.6). Available from: http://www.xilinx.com/ipcenter/catalog/logicore/docs/plb_v34.pdf.

[9] Xilinx Inc. PLB Double Data Rate (DDR) Synchronous DRAM (SDRAM) Controller, 12 2005. DS425. Available from: http://www.xilinx.com/bvdocs/ipcenter/data_sheet/plb_ddr.pdf.

[10] Xilinx Inc. PLB IPIF v1.00f, 02 2006. DS458.

[11] IBM Corporation. *64-Bit Processor Local Bus Architecture Specifications*, Mai 2001. Version 3.5. Available from: http://www-306.ibm.com/chips/techlib/techlib.nsf/pages/main?OpenDocument.

[12] Xilinx Corporation. *Processor Local Bus (PLB) Arbiter Design Specification*, Februar 2002. Available from: http://www.xilinx.com.

[13] Xilinx Corporation. *Timing Constraints for Virtex-II Pro Designs*, Januar 2003. v 1.1. Available from: http://www.xilinx.com.

[14] Xilinx Inc. *PowerPC 405 Processor Block Reference Guide*, UG018 (v2.1) edition, 2005. Available from: http://www.xilinx.com/bvdocs/userguides/ug018.pdf.

[15] Prof. Dr. Marwedel. *Embedded System Design*. Springer, 2006.

[16] Holger Morgenstern. Ein konfigurierbarer, visueller Cache-Simulator unter spezieller Berücksichtigung komponentenbasierter Modellierung mit Java Beans, 2001. Studienarbeit. Available from: http://www-ti.informatik.uni-tuebingen.de/~heim/arbeiten/holger_morgenstern/Cache.html.

[17] J. Hennessy and D. Patterson:. *Computer Architecture - a Quantitative Approach*. Morgan Kaufmann, 2003.

[18] What is Flash Memory [online]. Available from: http://www.intel.com/design/flash/articles/what.htm.

[19] Peter Ryser. *Getting Started With EDK and MontaVista Linux*. Xilinx, 2004. Available from: http://www.xilinx.com/bvdocs/appnotes/xapp765.pdf.

[20] Brent Nelson and Brad Baillio. *Configuring and Installing Linux on Xilinx FPGA Boards*. Brigham Young University, November 2007. Available from: http://splish.ee.byu.edu/projects/LinuxFPGA/configuring.htm.

[21] Department of Computer Science and Engineering, University of Washington. *EMPART - Configuring linux for the XUPV2P Development Board*, June 2006. Available from: http://www.cs.washington.edu/research/lis/empart/xup_ppc_linux.shtml.

[22] John H. Kelm. Running linux on a xilinx xup board. Technical report, ECE Illinois, 2006. Available from: http://courses.ece.uiuc.edu/ece412/MP_Files/mp2/20060623-XUP-Linux-Tutorial-REVISION-FINAL.pdf.

[23] Free Software Foundation. GNU Software. Available from: http://www.gnu.org/software.

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

[25] Dan Kegel. Building and Testing GCC/glibc Cross Toolchains, December 2006. Available from: http://kegel.com/crosstool/.

[26] The Linux Kernel Archives [online]. Available from: http://www.kernel.org.

[27] Montavista [online]. Available from: http://www.mvista.com.

[28] Free Software Foundation. GNU General Public License, 1991. Available from: http://www.gnu.org/licenses/gpl.html.

[29] Busybox [online]. Available from: http://www.busybox.org.

[30] Xilinx Inc. Hardware Reference Manual, XUP Virtex-II Pro Development System, 12 2004. UG069. Available from: http://mlxup.ee.byu.edu/docs/UG069_DRAFT_12_22_04.pdf.

[31] Peter Jay Salzman and Ori Pomerantz. *The Linux Kernel Module Programming Guide*, April 2003. Available from: http://tldp.org/LDP/lkmpg/2.4/html/index.html.

[32] Jürgen Quade and Eva-Katharina Kunst. *Linux-Treiber entwickeln*. dpunkt.verlag, 2004. Available from: http://ezs.kr.hs-niederrhein.de/TreiberBuch/html/.

[33] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers Second Edition*. O'Reilly, 2001. Available from: http://www.xml.com/ldd/chapter/book/.

[34] Doxygen source code documentation tool [online]. Available from: http://www.doxygen.org.

[35] Gcc packed structures [online]. Available from: http://sig9.com/articles/gcc-packed-structures.