Diplomarbeit

Implementierung eines maschinell verifizierten Prozessors

Dirk Leinenbach

Universität des Saarlandes
Fachrichtung 6.2 Informatik
Lehrstuhl für Rechnerarchitektur und Parallelrechner
Prof. Dr. W. J. Paul

Juni 2002

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt habe. Ich habe diese Arbeit keinem anderen Prüfungsamt vorgelegt.

Saarbrücken, im Juni 2002

Dirk Leinenbach

Danksagung

Ich möchte den folgenden Personen danken, die mich beim Erstellen dieser Arbeit unterstützt haben:

- Professor Paul für die Vergabe dieses interessanten Themas,
- meinen Eltern für die Unterstützung während meines Studiums,
- Christian Jacobi für die Betreuung der Diplomarbeit und seine große Geduld,
- Christoph Berg für das Korrekturlesen der Arbeit und viele Tipps zu LaTeX,
- Sven Beyer für die gute Zusammenarbeit bei der Entwicklung von PVS2HDL und beim Debugging des Speicher-Interfaces,
- Carsten Meyer und Daniel Kröning für die gute Zusammenarbeit im VAMP-Projekt,
- Frank Haase und René Richter für die Unterstützung bei Fragen zum Xilinx-Board,
- dem gesamten Lehrstuhl für die gute Atmosphäre,
- und insbesondere dem Tante-Emma Team für die ausreichende Versorgung mit gutem Kaffee.

Inhaltsverzeichnis

1 Einleitung **9**

2 Das Übersetzungstool PVS2HDL **11**

2.1 Hardwareimplementierung mit PVS . . . 11
2.1.1 Kombinatorische Hardware . . . 11
2.1.2 Hardware mit Registern . . . 12
2.1.3 Nicht übersetzbare Konstrukte . . . 12
2.1.4 Hardware-Bibliothek . . . 13
2.1.5 Beispiel: Carry-Chain-Addierer . . . 13
2.2 Übersetzung nach Verilog . . . 14
2.2.1 Grundlegende Datentypen und Operatoren . . . 15
2.2.2 Records . . . 15
2.2.3 If und Cond . . . 15
2.2.4 Funktionen und LETs . . . 16
2.2.5 Integer-Parameter und Auswertung von Konstanten . . . 16
2.2.6 Rekursion . . . 17
2.2.7 Lambda-Ausdrücke . . . 19
2.2.8 Schaltungen mit Registern . . . 19
2.3 Einbindung von externen Modulen . . . 21
2.3.1 Verbinden von externer Hardware mit generierten Modulen . . . 22
2.4 Experimentelle Ergebnisse . . . 22
2.5 Ähnliche Ansätze . . . 23

3 FPGA und PCI-Karte **25**

3.1 VirtexE-FPGA . . . 25
3.1.1 Grundstruktur . . . 25
3.1.2 Block-RAM . . . 26
3.1.3 Takt-Generierung . . . 27
3.2 PCI-Karte . . . 28
3.2.1 Aufbau der Karte . . . 28
3.2.2 UMRBUS . . . 30
3.2.3 Schnittstelle zu den CAPIMs . . . 32

4 Implementierung **35**

4.1 Reset-Logik . . . 35
4.2 Host-Interface . . . 36

4.2.1 Synchronisation 36
4.2.2 Speicherzugriffe 38
4.2.3 CPU-Reset 40
4.3 Speicher-Interface 41
4.3.1 Die Kontrolle des Speicher-Interfaces 41
4.4 Drei-Port-RAM für die Register-Files 47
4.5 CoreGen-Module und grundlegende Schaltkreise 47
4.6 Synthetisierung des Verilog-Codes 48
4.7 Zahlen 48

5 Projekt-Status 51

A Aufrufkonvention von PVS2HDL 53

B Konfigurationsdatei pvs2hdl.conf 55

C Quelltexte 57
C.1 Quelltext des Speicher-Interfaces 57
C.2 Quelltext des Host-Interfaces 64
C.3 Reset-Logik 69

Abbildungsverzeichnis

2.1 Realisierung von Registern durch eine Next-State Funktion 13
2.2 Volladdierer und n-Bit Carry-Chain-Addierer 14
2.3 Grafische Definition eines Oder-Baumes 17
2.4 Kombination von zwei Next-State-Funktionen 19
2.5 Vergleich übersetzter Hardware 23

3.1 Grundstruktur eines VirtexE-FPGAs 26
3.2 Konfigurierbarer Logikblock (CLB) 27
3.3 Timing von Block-RAMs mit invertiertem Taktsignal 27
3.4 Eliminierung von Clock-Skew 29
3.5 Aufbau der PCI-Karte 29
3.6 Verbindung von VAMP und Host-PC 31
3.7 Interface zu den Capims 33
3.8 Schreibzugriff der Host-Software auf das Capim-Interface 33
3.9 Lesezugriff der Host-Software auf das Capim-Interface 34
3.10 Interrupt-Request über das Capim-Interface 34

4.1 Das Host-Interface 37
4.2 Der zweite Zugriffs-Port des Caches 39
4.3 Zustandsautomat des Host-Interfaces 40
4.4 Schreibzugriff auf den zweiten Cache-Port 40
4.5 Lesezugriff auf den zweiten Cache-Port 41
4.6 Das Speicher-Interface 42
4.7 Zustandsautomat des Speicher-Interfaces 42
4.8 Einzelner Lesezugriff 45
4.9 Einzelner Schreibzugriff 45
4.10 Burst Lesezugriff 46
4.11 Burst Schreibzugriff 46
4.12 Drei-Port-RAM 47

Kapitel 1

Einleitung

Seit den Anfängen der Mikroprozessoren in den späten Sechziger Jahren verdoppelt sich nach Moore's Gesetz [Moo65] etwa alle 18 Monate die Komplexität von elektronischen Schaltungen. Damit steigt auch dramatisch die Wahrscheinlichkeit für Fehler. Es werden zwar umfangreiche Tests durchgeführt, um Fehler aufzudecken, dennoch steigt mit zunehmender Komplexität der Schaltungen auch die Wahrscheinlichkeit, dass ein oder mehrere Fehler unentdeckt bleiben, bis das Produkt in den Handel gelangt. Fehler, die erst nach der Auslieferung entdeckt werden sind sehr unangenehm für den Hersteller, da sie hohe Folgekosten verursachen können. Außerdem gibt es Bereiche, in denen Fehler in Mikroprozessoren und elektronische Schaltungen um jeden Preis vermieden werden müssen. Als Beispiel seien hier lebenswichtige Systeme in der Medizin oder in der Flugzeugtechnik genannt. Für dermaßen wichtige Systeme werden oft Korrektheitsbeweise durchgeführt, die sicherstellen, dass ein System in allen Situationen genau seiner Spezifikation entspricht. Allerdings können diese Korrektheitsbeweise selbst auch fehlerhaft sein, da sie von Menschen durchgeführt werden. Deshalb gibt es seit einigen Jahren Bestrebungen die Korrektheitsbeweise mit maschinellen Beweissystemen zu verifizieren. Dadurch können Fehler in den Beweisen ausgeschlossen werden.

Am Lehrstuhl für Rechnerarchitektur an der Universität des Saarlandes wird im VAMP-Projekt (**V**erified **A**rchitecture **M**icro**p**rocessor) mit Hilfe des maschinellen Beweissystems PVS [OSR92] die formale Verifikation einer kompletten CPU durchgeführt. Der VAMP-Prozessor implementiert eine Variante der DLX-Architektur aus [HP96]. Er arbeitet mit dem MIPS-Instruktionssatz [KH92]. Er unterstützt Out-Of-Order Befehlsausführung mit Hilfe eines Tomasulo-Schedulers [Tom67] sowie präzise, geschachtelte Interrupts. Es gibt eine Memory-Management-Einheit inklusive Caches und eine IEEE-konforme Fließkomma-Einheit [IEE85]. Der Prozessor basiert auf den Designs aus [MP00]. Der Integer-Teil sowie der Tomasulo-Scheduler stammen von Daniel Kröning [Kro01]. Die Memory-Management-Einheit wird von Sven Beyer entwickelt und verifiziert [Bey02]. Die Verifikation der Fließkomma-Einheit ist die Arbeit von Christian Jacobi und Christoph Berg. Christoph Berg hat den Fließkomma-Addierer in PVS implementiert und verifiziert [Ber01]. Von Christian Jacobi stammt die Implementierung und Verifikation des Fließkomma-Runders und der beiden anderen Funktionseinheiten (Multiplikation/Division, Typ-Umwandlung/sonstiges) [Jac02].

Der in der Spezifikationssprache von PVS implementierte Prozessor wird mit ei-

nem automatischen Übersetzungstool in die Hardwarebeschreibungssprache Verilog übersetzt. Die Implementierung in Verilog wird anschließend in reale Hardware umgesetzt.

Innerhalb des VAMP-Projektes entstand meine Diplomarbeit. Darin geht es zum einen um die Erstellung des Programms PVS2HDL zur automatischen Übersetzung der Prozessorimplementierung von der Spezifikationssprache des Beweissystems PVS in die Hardwarebeschreibungssprache Verilog [IEE95, Cil99]. Zum anderen wird die anschließende Implementierung des Prozessors in einem Xilinx-FPGA auf einer PCI-Prototypen-Karte sowie die Entwicklung von Hardware zur Anbindung des Prozessors an Speicher und einen Hostrechner behandelt.

Übersicht: Im folgenden Kapitel werden die Hardware-Implementierung mit PVS sowie das Übersetzungstool PVS2HDL näher behandelt. Am Ende von Kapitel zwei wird die Qualität der mit PVS2HDL erzeugten Verilog-Module mit der Qualität von handgeschriebenem Verilog-Code verglichen. Im letzten Abschnitt von Kapitel zwei werden ähnliche Ansätze zur Umwandlung von Spezifikationssprachen von Beweissystem in Hardwarebeschreibungssprachen beziehungsweise den umgekehrten Weg vorgestellt. In Kapitel drei wird der Aufbau des verwendeten Xilinx-FPGAs sowie der PCI-Karte beschrieben, die außer dem FPGA auch noch SD-RAM Speicher sowie ein Interface zum PCI-Bus zur Verfügung stellt. Zusätzlich wird erläutert, wie die verschiedenen Bestandteile der PCI-Karte und des FPGAs für die Implementierung des VAMP-Prozessors benutzt werden. Im vierten Kapitel wird dann die Hardware vorgestellt, die die Kommunikation zwischen Hostrechner und VAMP regelt. Zudem wird das Speicher-Interface vorgestellt, das die Umsetzung der Speicherzugriffe des Caches auf die SD-RAM Chips übernimmt. Danach werden noch einige weitere Schaltkreise des VAMP vorgestellt, die nicht mit PVS2HDL aus PVS-Code generiert wurden. Im weiteren Verlauf Kapitel vier wird kurz auf die Synthetisierung der Verilog-Sourcen eingegangen. Das fünfte Kapitel enthält zum Abschluss einen Statusbericht über das VAMP-Projekt.

Kapitel 2

Das Übersetzungstool PVS2HDL

Der VAMP-Prozessor wird in der funktionalen Spezifikationssprache des Theorembeweisers PVS [OSR92] implementiert. Damit der Prozessor auf einem FPGA synthetisiert werden kann, wird seine Implementierung zunächst automatisch aus dieser Spezifikationssprache in die Hardwarebeschreibungssprache Verilog übersetzt. Dies geschieht mit Hilfe des Tools PVS2HDL, das zusammen mit Sven Beyer entwickelt wurde. In diesem Kapitel wird sowohl die Beschreibung von Hardware mit PVS als auch das Übersetzungstool PVS2HDL beschrieben [BJKL02].

2.1 Hardwareimplementierung mit PVS

2.1.1 Kombinatorische Hardware

Die grundlegenden Datentypen, die zur Implementierung von Hardware in PVS benutzt werden, sind `bit` und `bvec[n]` zur Modellierung von einzelnen Bits beziehungsweise Bitvektoren der Länge n. Diese Typen sind in der Bitvektor-Bibliothek von PVS spezifiziert [BMSG96]. Dort werden auch die grundlegenden Operationen auf diesen Datentypen definiert (siehe Tabelle 2.1). Des Weiteren werden Records und Arrays über diesen Datentypen benutzt.

Um Fallunterscheidungen treffen zu können, stehen `cond`- und `if`-Ausdrücke zur Verfügung.

Kombinatorische Hardwaremodule werden in PVS als Funktionen definiert. Die Parameter der Funktionen entsprechen der Eingabe der Hardwaremodule, und der Rückgabewert entspricht der Ausgabe der Module. Die Instantiierung der Hardwaremodule in anderen Modulen entspricht Funktionsaufrufen innerhalb anderer Funktionen. Die Funktionen haben Parameter vom Typ `bit` und `bvec[n]`. Außerdem können Integer-Parameter vom Typ `nat` oder `posnat` benutzt werden, um Hardwaremodule mit parametrisierter Bitbreite zu definieren.

Mit rekursiven Funktionen kann rekursiv aufgebaute Hardware, wie zum Beispiel Carry-Chain Addierer, definiert werden.

Durch `LET` x=$ausdr1$ `in` $ausdr2$ wird x der Wert des Ausdruckes $ausdr1$ zugewiesen. Dadurch können mehrere Vorkommen von $ausdr1$ innerhalb von $ausdr2$ durch x ersetzt werden. Dadurch wird nur einmal Hardware für das Berechnen von $ausdr1$ generiert. Der Ausgang der Schaltung für $ausdr1$ wird dann mehrfach ge-

Tabelle 2.1: Übersetzung von grundlegenden Operatoren(a und b sind Bitvektoren)

Operator in PVS	Verilog-Übersetzung	Semantik
a and b	a & b	Und-Verknüpfung
a or b	a | b	Oder-Verknüpfung
a xor b	a ^ b	Exklusives Oder
not a	~a	Negierung
a(i)	a[i]	das i-te Bit von a
a^(i,j)	a[i:j]	die Bits i bis j von a
fill[n](false)	n'b00...00	Bitvektor der Länge n,bei dem jedes Bit 0 ist
fill[n](true)	n'b11...11	Bitvektor der Länge n, bei dem jedes Bit 1 ist
a o b	{a,b}	Konkatenation der Bitvektoren a und b
nat2bv[n](i)	n'di	Bitvektor mit dem Binärwert i
int2bv[n](i)	n'dj	Bitvektor mit dem Zweierkomplementwert j, mit $j := \begin{cases} i \text{ falls } i \geq 0 \\ i + 2^n \text{ falls } i < 0 \end{cases}$

nutzt, genauso wie in PVS x mehrfach genutzt wird.

Einzelne Bits können mit Lambda-Ausdrücken zu Bitvektoren kombiniert werden. Mit Lambda-Ausdrücken kann zum Beispiel ein Bitvektor gespiegelt werden:

```
flipped(a:bvec[8]): bvec[8] = LAMBDA(i:below(8)): a(7-i);
```

Der obige Ausdruck bewirkt, dass die Funktion für alle $i < 8$ als i-tes Bit der Rückgabe das ($7 - i$)-te Bit der Eingabe zurückliefert.

2.1.2 Hardware mit Registern

Bei der Entwicklung von Hardware ist es meistens notwendig, auch Register zu benutzen. Register werden normalerweise mit globalen Zustandsvariablen modelliert. Die Spezifikationssprache von PVS ist aber funktional und hat keine direkte Unterstützung für globale Zustandsvariablen. Deshalb wird in PVS das Verhalten von Schaltkreisen mit Registern mit Hilfe einer Next-State-Funktion modelliert, die als Eingabe den aktuellen Registerinhalt sowie kombinatorische Inputsignale bekommt und als Ausgabe den nächsten Registerinhalt sowie (eventuell) kombinatorische Signale liefert (siehe Abbildung 2.1). Der PVS-Datentyp, in dem dabei der aktuelle Registerinhalt kodiert wird, wird im Folgenden als Registerdatentyp bezeichnet.

2.1.3 Nicht übersetzbare Konstrukte

Nicht alle Sprachkonstrukte von PVS werden für die Übersetzung nach Verilog unterstützt. Die meisten nicht übersetzbaren Konstrukte sind für die Beschreibung von Hardware aber nicht von Nutzen.

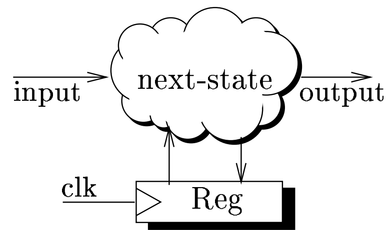

Abbildung 2.1: Realisierung von Registern durch eine Next-State Funktion

Quantoren (∀, ∃) und Ausdrücke in Logik höherer Ordnung können nicht übersetzt werden. Parameter von Funktionen dürfen nur vom Typ `bit`, `bvec[n]` oder Integer sein oder aus geschachtelten Records daraus bestehen. Das Überladen von Funktionen oder Datentypen ist nicht erlaubt.

Sprachkonstrukte, die nicht übersetzbar sind, können trotzdem in Beweisen und der Spezifikation benutzt werden. Dort werden sie durch PVS2HDL ignoriert.

2.1.4 Hardware-Bibliothek

Im Rahmen des VAMP-Projektes ist eine Bibliothek von verifizierten Schaltkreisen entstanden [BJK01]. Die Schaltkreise sind in PVS auf der Gatterebene spezifiziert und gegen eine abstrakte Definition verifiziert worden. Die Bibliothek umfasst unter anderem Inkrementierer, Addierer, Multiplizierer, Leading Zero Counter, Shifter und Dekoder. Diese Schaltkreise sind über Integerparameter in der Bitbreite parametrisierbar.

2.1.5 Beispiel: Carry-Chain-Addierer

Als Beispiel für Hardwareimplementierung mit PVS dient ein Carry-Chain-Addierer. In diesem Schaltkreis sind alle grundlegenden Konzepte außer Lambda-Ausdrücken zu sehen. Ein Beispiel mit Registern gibt es in Abschnitt 2.2.8.

Grafisch wird ein Carry-Chain-Addierer wie in Abbildung 2.2 definiert. In PVS sieht die Definition wie folgt aus:

```
fulladder(a, b, c: bit): bvec[2] =
    LET x = a XOR b IN
     ((a AND b) OR (c AND x)) o (x XOR c);

carry_chain(n: posnat, a, b: bvec[n], cin: bit):
    RECURSIVE bvec[n+1] =
      IF n = 1 THEN
        fulladder(a(0), b(0), cin)
      ELSE
        LET
          chain = carry_chain(n-1, a^(n-2, 0), b^(n-2, 0), cin)
        IN
          fulladder(a(n-1), b(n-1), chain(n-1)) o chain^(n-2,0)
      ENDIF
    MEASURE n;
```

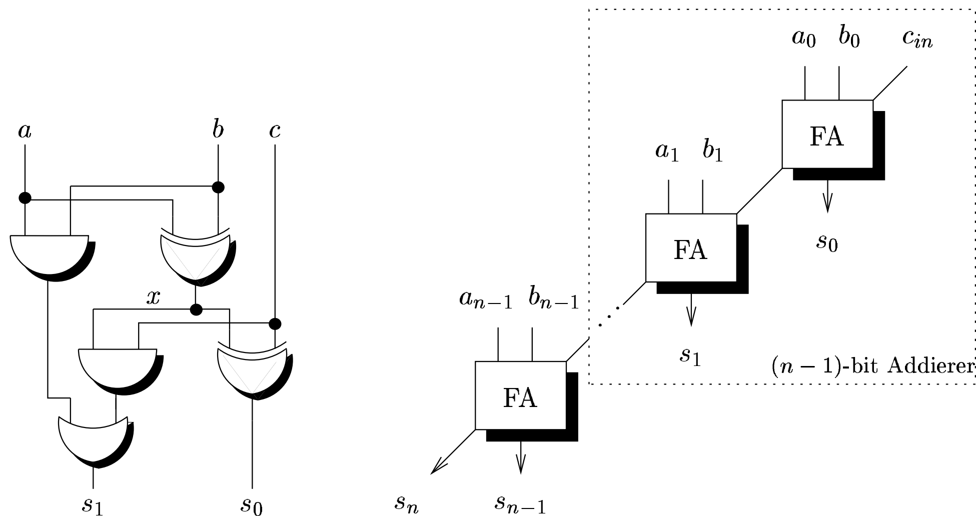

Abbildung 2.2: Grafische Definition eines Volladdierers FA und eines n-Bit Carry-Chain-Addierers

Neben der Definition des Carry-Chain-Addierers liegen in PVS auch Lemmas über den Carry-Chain-Addierer vor. Ebenso sind die Korrektheitsbeweise dieser Lemmas vorhanden. Für das Übersetzen nach Verilog sind aber weder die Lemmas noch deren Beweise notwendig. Deshalb werden diese Teile von PVS2HDL ignoriert.

2.2 Übersetzung nach Verilog

Die Übersetzung von PVS nach Verilog erfolgt in zwei Phasen. In der ersten Phase parst PVS2HDL die PVS-Eingabedatei und alle davon importierten PVS-Dateien. Aus diesen PVS-Dateien erzeugt das Tool jeweils einen Syntaxbaum. Der PVS-Parser von PVS2HDL hat einige leichte Einschränkungen bezüglich der unterstützten PVS-Syntax. Unter anderem müssen an einigen Stellen Semikolons zwingend vorhanden sein, die in PVS optional sind. Genauere Informationen zu diesen Einschränkungen sind in [Lei01] zu finden.

Nach dem Parsen erzeugt PVS2HDL zwei Listen. Die beiden Listen enthalten die Position aller Funktions- beziehungsweise Typdefinitionen innerhalb des Syntaxbaums. Dadurch können beim Übersetzen schneller zusätzliche Informationen zu diesen Funktionen beziehungsweise Typen ermittelt werden. Zusätzlich werden einige grundlegende Informationen, wie zum Beispiel die Parameter einer Funktion, direkt in den beiden Listen gespeichert, damit sie nicht mehrfach aus dem Syntaxbaum extrahiert werden müssen.

Die Bitvektor-Bibliothek wird nicht geparst, da die darin enthaltenen Typen und Funktionen dem Tool schon bekannt sind und direkt übersetzt werden (siehe Tabelle 2.1), ohne die Definitionen der Funktionen in der Bitvektor-Bibliothek zu beachten.

In der zweiten Phase übersetzt das Tool die synthetisierbaren Funktionen der PVS-Eingabedatei sowie alle weiteren Funktionen, die in diesen benutzt werden.

Die Teile der PVS-Dateien, die nur für die Beweise von Belang sind, werden von PVS2HDL ignoriert. Dazu gehören zum Beispiel Lemmas. Dadurch können in den Be-

weisen auch Sprachkonstrukte verwendet werden, die vom Tool nicht übersetzt werden können.

In den folgenden Abschnitten wird detailliert darauf eingegangen, wie die einzelnen Sprachkonstrukte von PVS nach Verilog übersetzt werden.

2.2.1 Grundlegende Datentypen und Operatoren

Die Datentypen `bit` und `bvec[n]` sind die einzig möglichen Datentypen für Parameter von Verilog-Modulen. Ein Eingabe-Parameter `a` vom Typ `bit` beziehungsweise `bvec[n]` wird zu `input a` beziehungsweise `input [n-1:0] a` übersetzt.

Wie die grundlegenden Operatoren auf Bitvektoren übersetzt werden, ist in Tabelle 2.1 zu sehen.

2.2.2 Records

Die Spezifikationssprache von PVS ermöglicht es, eigene Datentypen zu definieren, indem die grundlegenden Datentypen zu Records zusammengefasst werden. Diese Records können beliebig geschachtelt werden.

Verilog unterstützt keine Records. Deshalb werden Records beim Übersetzen von PVS nach Verilog durch PVS2HDL aufgerollt. Das bedeutet, dass ein einzelner Record-Parameter in PVS beim Übersetzen nach Verilog in mehrere Parameter aufgeteilt wird. Dabei wird aus jedem Bestandteil des Records ein neuer Verilog-Parameter. Falls der Record selbst auch wieder Records enthält, werden diese ebenfalls aufgerollt.

2.2.3 If und Cond

Die Übersetzung von COND-Fallunterscheidungen wird anhand eines Beispiels erklärt:

```
COND r = a -> e1,
     r = b -> e2,
     r = c -> e3,
ELSE -> e4
ENDCOND
```

Dieser PVS-Code wird von PVS2HDL folgendermaßen übersetzt:

```
(r==a) ? e1:
   (
     (r==b) ? e2:
       (
         (r==c) ? e3:e4
       )
   )
```

Der Fragezeichen-Operator in Verilog hat dabei folgende Semantik: falls der Ausdruck vor dem Fragezeichen wahr ist, wird der Ausdruck vor dem Doppelpunkt zurückgeliefert, ansonsten der Ausdruck hinter dem Doppelpunkt.

Bei der Übersetzung von `IF` müssen zwei Fälle unterschieden werden. Falls die Bedingung des `IF` schon während der Übersetzung zu `true` oder `false` ausgewertet werden kann, wird nur der entsprechende Teil des IFs übersetzt. Dies ist zum Beispiel

bei der IF-Abfrage für den Rekursionsanfang von rekursiv definierten Funktionen der Fall (siehe auch Abschnitt 2.2.6). Falls die Bedingung nicht ausgewertet werden kann, wird das IF in Hardware realisiert. Dazu wird in Verilog, wie schon bei der Übersetzung von Fallunterscheidungen mit `COND`, der Fragezeichen-Operator benutzt. Bei der Synthese des Verilog-Codes wird der Fragezeichen-Operator dann zu einem Multiplexer umgewandelt.

2.2.4 Funktionen und LETs

Die Definition einer Funktion in PVS übersetzt das Tool zu einem Verilog-Modul mit entsprechenden Ein- und Ausgabeparametern. Ein Funktionsaufruf wird in Verilog zu einer neuen Instanz des entsprechenden Moduls übersetzt. Insbesondere werden also mehrere Instanzen der gleichen Hardware generiert, wenn eine Funktion in PVS an verschiedenen Stellen benutzt wird.

Das doppelte Generieren von Hardware kann durch die Benutzung von `LET` verhindert werden (siehe Abschnitt 2.1.1). Für `LET a=f(b)` legt PVS2HDL einen neuen *Wire* an, dem das Ergebnis von `f(b)` zugewiesen wird. Der neue Wire wird dann überall dort benutzt, wo im PVS-Sourcecode `a` benutzt wird. Falls der Rückgabetyp von `f` aus einem Record besteht, wird für jeden Bestandteil des Records ein neuer Wire erzeugt.

2.2.5 Integer-Parameter und Auswertung von Konstanten

In PVS werden Integer-Parameter auch in Funktionen benutzt, die synthetisiert werden sollen. Diese Integer-Parameter dienen dazu die Bitbreite einer Schaltung variabel zu halten (siehe Abschnitt 2.1.1). Dadurch muss der Korrektheitsbeweis für die Schaltung nur ein einziges Mal allgemein für Bitbreite `n` durchgeführt werden (zum Beispiel mittels vollständiger Induktion). Die verifizierte Schaltung kann dann später für beliebige Bitbreiten benutzt und übersetzt werden.

PVS2HDL kann Funktionen mit Integerparametern nur übersetzen, wenn es die Parameter beim Übersetzen zu Konstanten auswerten kann. Konstante Parameter werden von übergeordneten Funktionen nach unten durchpropagiert. Grundlegende PVS-Funktionen mit konstanten Integer-Parametern (siehe Tabelle 2.2) werden während der Übersetzung der PVS-Dateien durch PVS2HDL ausgewertet. Zusätzlich werden auch Vergleiche (<, <=, =, >=, >) unterstützt.

Neben Ausdrücken mit Integerkonstanten werden auch grundlegende logische Operationen wie `and`, `or`, `xor`, `not`, `implies`, `when` und `<=>` ausgewertet. Je nachdem, ob die Bedingung erfüllt ist oder nicht, werden IFs mit einer konstanten Bedingung durch den Teil vor oder nach dem `else` ersetzt.

Die Integer-Parameter einer PVS-Funktion werden in Verilog nicht zu Parametern der Verilog-Module übersetzt, da sie zu Konstanten ausgewertet werden. Für jede vorkommende Kombination von konstanten Integer-Parametern einer Funktion wird ein neues Verilogmodul definiert (zum Beispiel ein Modul für einen 32-Bit Addierer und ein neues Modul für einen 16-Bit Addierer).

Damit diese Module eindeutige Namen haben, fügt PVS2HDL an den Funktionsnamen noch ein `x` sowie die ausgewerteten Konstanten an. Aus dem Funktionsaufruf `add(32,a,b)` in PVS wird beim Übersetzen das Modul `addx_32(a,b)`. Ohne

Tabelle 2.2: Unterstützte Funktionen auf ganzen Zahlen (a, b) und reellen Zahlen (r)

Funktion in PVS	Semantik
a+b, a-b, a*b, a/b	$a+b$, $a-b$, $a*b$, a/b
exp2(a)	2^a
a^b	a^b
floor(r)	$\lfloor r \rfloor$
ceil(r)	$\lceil r \rceil$
even(a)	$\begin{cases} true \text{ falls } a \text{ gerade} \\ false \text{ falls } a \text{ ungerade} \end{cases}$
odd(a)	$\begin{cases} true \text{ falls } a \text{ ungerade} \\ false \text{ falls } a \text{ gerade} \end{cases}$
mod(a,b)	a mod b
div(a,b)	$\lfloor a/b \rfloor$

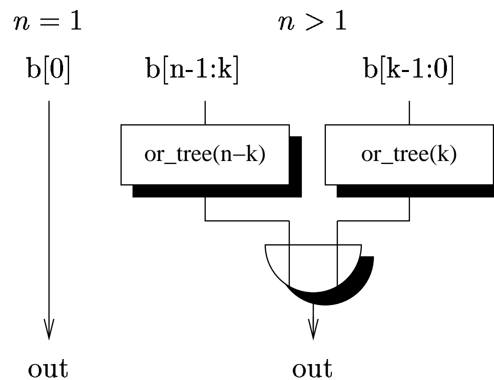

Abbildung 2.3: Grafische Definition eines Oder-Baumes

das x wären die Modulnamen nicht immer eindeutig. Zum Beispiel hätten die PVS-Funktionen add_32(a,b) sowie add(32,a,b) beim Übersetzen beide den gleichen Modulnamen add_32. Mit dem angehängten x heißt das Modul für die erste Funktion add_32x(a,b) und das für die zweite addx_32(a,b).

2.2.6 Rekursion

Um die Beweise zu vereinfachen und die Implementierungen übersichtlich zu gestalten, sind viele Funktionen in PVS rekursiv definiert. Rekursive Definitionen von Modulen sind in Verilog aber nicht möglich. Daher muss PVS2HDL die Rekursion während des Übersetzens aufrollen.

Als Beispiel für eine rekursive Definition in PVS und ihre Übersetzung nach Verilog dient die Definition eines Oder-Baums. In Abbildung 2.3 ist die grafische Definition eines Oder-Baumes zu sehen. Dabei ist b[n-1:0] der n-Bit breite Eingang und das Bit out der Ausgang der Schaltung.

In PVS sieht die Definition der entsprechenden Funktion or_tree folgendermaßen aus:

```
or_tree(n : nat, b : bvec[n]) :
  RECURSIVE bit =
    IF n = 1 THEN
      b(0)
    ELSE
      LET k=floor(n/2) IN
      or_tree( n-k , b^(n-1,k) ) OR or_tree( k , b^(k-1,0) )
    ENDIF
  MEASURE n;
```

Die Funktion besitzt einen Integer-Parameter n. Durch diesen Parameter wird die Bitbreite des zweiten Parameters festgelegt. Der Funktionsaufruf für einen vier Bit breiten Oder-Baum (or_tree(4,b)) führt dann bei der Übersetzung durch nach Verilog durch PVS2HDL zur Spezifikation von drei Verilog-Modulen. Dabei erkennt man, dass vom IF-Ausdruck in jedem Modul nur ein Teil übersetzt wird, je nachdem welchen Wert n hat:[1]

```
module or_treex_1(b, out);
   input b;
   output out;

   assign out = b;
endmodule

module or_treex_2(b, out);
   input [1:0] b;
   output out;

   wire wire0;    // Definition von Wires um die Outputs der
   wire wire1;    // kleineren Or-Trees  aufnehmen zu können

   or_treex_1 m0 (b[1], wire0); // die rekursiven Aufrufe
   or_treex_1 m1 (b[0], wire1);

   assign out = ( wire0 | wire1 );
endmodule

module or_treex_4(b, out);
   input [3:0] b;
   output out;

   wire wire0;    // Definition von Wires um die Outputs der
   wire wire1;    // kleineren Or-Trees  aufnehmen zu können

   or_treex_2 m0 (b[3:2], wire0); // die rekursiven Aufrufe
   or_treex_2 m1 (b[1:0], wire1);

   assign out = ( wire0 | wire1 );
endmodule
```

[1] zur besseren Lesbarkeit des Verilogcodes wurden die Parameternamen nachträglich wieder an den PVS-Code angepasst sowie zusätzliche Kommentare eingefügt

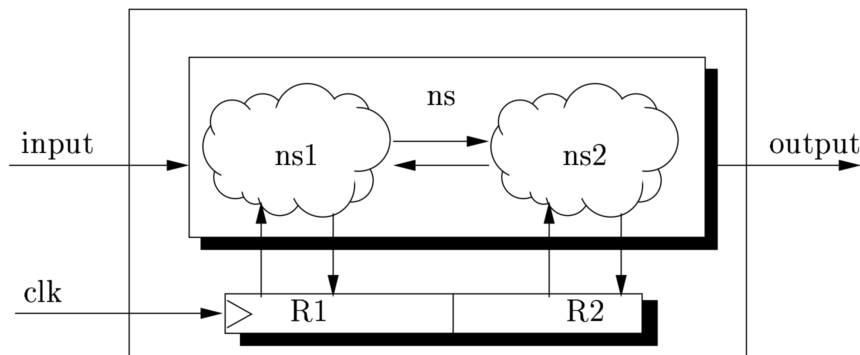

Abbildung 2.4: Kombination von zwei Next-State-Funktionen

2.2.7 Lambda-Ausdrücke

Lambda-Ausdrücke werden von Verilog nicht unterstützt. PVS2HDL übersetzt Lambda-Ausdrücke durch Auswerten für jedes einzelne Bit. Anschließend werden die Bits zu einem Bitvektor konkateniert. Für das Beispiel aus Abschnitt 2.1.1 liefert PVS2HDL folgende Übersetzung:

```
module flippedx(clk, a_0x, out_1x );
   input clk;
   input [7:0] a_0x;
   output [7:0] out_1x;

   assign out_1x = {a_0x[0], a_0x[1], a_0x[2], a_0x[3],
                    a_0x[4], a_0x[5], a_0x[6], a_0x[7]};
endmodule
```

2.2.8 Schaltungen mit Registern

PVS2HDL unterstützt bei der Übersetzung nur eine einzige Next-State-Funktion für den kompletten Schaltkreis (zum Beispiel den kompletten VAMP-Prozessor). Register kommen dagegen an vielen verschiedenen Stellen innerhalb des Prozessors vor (zum Beispiel im Reorder-Buffer des Tomasulo-Schedulers und zwischen den Pipelinestufen der FPU). Die einzelnen Next-State Funktionen, die zu diesen Registern gehören, können aber in PVS leicht von Hand zu einer einzigen großen Next-State-Funktion kombiniert werden. Dazu werden die Registerdatentypen der Funktionen zusammengefasst und die ursprünglichen Next-State Funktionen, die nun innerhalb der neuen Next-State-Funktion aufgerufen werden, greifen nur auf ihren jeweiligen Teil des kombinierten Registerdatentypes zu (Abbildung 2.4).

Bei der Übersetzung wird PVS2HDL über einen Kommandozeilenparameter angegeben, welcher PVS-Datentyp der Registerdatentyp ist (siehe auch Anhang A). Das Tool übersetzt zunächst die kombinatorischen Teile der Next-State-Funktion. Die Parameter, die in der Next-State-Funktion für die Ein- beziehungsweise Ausgabe des Register-Inhaltes zuständig sind, werden beim Übersetzen weggelassen. Ihre Funktion übernehmen der Eingang und der Ausgang des Registers, das von PVS2HDL innerhalb des Verilog-Moduls instantiiert wird. Falls mehrere Parameter den gleichen Typ

wie die Register-Parameter haben, wird nur der jeweils erste weggelassen. Die anderen werden als gewöhnliche kombinatorische Ein- beziehungsweise Ausgänge behandelt. PVS2HDL fügt anschließend zum Verilog-Modul den Takteingang `clk` hinzu.

Beispiel: Zähler

Als Beispiel für einen Schaltkreis mit Register dient ein einfacher Binärzähler mit Reset-Eingang. Falls am Reset-Eingang eine Null anliegt, inkrementiert der Zähler bei jeder steigenden Taktflanke ein internes Register. Falls der Reset-Eingang eins ist, wird das Register auf Null zurückgesetzt. Als Ausgabe hat der Zähler den aktuellen Inhalt des Registers. In PVS ist der Zähler folgendermaßen definiert:

```
counter: THEORY
BEGIN
   basics : LIBRARY ="../../basics";
   IMPORTING basics@incrementer

   reg_type:TYPE=bvec[8];

   counter(reset:bit,a:reg_type):
     [# next_counter: reg_type, output: reg_type #]=
     LET out = IF reset THEN
                 fill[8](FALSE)
               ELSE
                 incr_impl(8,a,TRUE)^(7,0)
               ENDIF
     IN
       (# next_counter := out,
          output       := a #);
END counter
```

In Zeile sechs wird der Register-Datentyp `reg_type` definiert. Ab Zeile acht folgt dann die Definition der eigentlichen Zähler-Funktion. Die Funktion hat als Eingang das `reset`-Signal sowie den Inhalt des Registers, das den nächsten Zählerwert speichert. Falls der Reset-Eingang Null ist, wird die `fill`-Funktion aufgerufen, ansonsten ein Inkrementierer aus der Hardware-Bibliothek von Abschnitt 2.1.4.

Aus der PVS-Funktion `counter` erzeugt PVS2HDL beim Übersetzen folgenden Verilog-Code[2]:

```
module counterx(clk, a, output);

  input clk;
  input a;
  output [7:0] output;

  reg  [7:0] register;

  wire [7:0] wire0;
  wire wire1;
  wire [8:0] wire2;
```

[2] Die Implementierung des Inkrementierers wird aus Platzgründen weggelassen. Die Variablennamen werden zur besseren Lesbarkeit abgeändert.

```
  wire [7:0] wire3;

  assign wire0 = register;
  assign wire1 = 1'b1;
  assign wire3 = (a) ? (8'b00000000) : (wire2[7:0]);
  assign output = register;

  always @ (posedge clk)
    begin
      register <= wire3;3
    end

  incr_implx_8 m0 (clk, wire0, wire1, wire2);

endmodule
```

Der `always`-Block bewirkt, dass der Inhalt des Registers bei jeder steigenden Taktflanke aktualisiert wird.

2.3 Einbindung von externen Modulen

Einige Teile des Prozessors werden nicht von PVS2HDL aus den PVS-Sourcen generiert, sondern von Hand in Verilog geschrieben oder mit dem Programm `CoreGen` von Xilinx generiert. Es lassen sich drei Gründe unterscheiden, warum diese Teile nicht aus den PVS-Sourcen generiert werden:

1. Effizienz
   Wie in Abschnitt Abschnitt 2.4 noch genauer gezeigt und begründet wird, sind Addierer und Multiplizierer deutlich effizienter, wenn sie mit den Verilog Operatoren `+` beziehungsweise `*` realisiert werden. Deshalb werden zum Beispiel die größeren Carry-Chain Addierer im VAMP-Prozessor durch handgeschriebene Addierer-Module ersetzt.

2. Speicherelemente
   Im VAMP-Prozessor werden relativ große Mengen an RAM für die Register des Prozessors und für den Cache benötigt. Zusätzlich wird für die Lookup-Tabelle des Divisions-Schaltkreises ROM benötigt. Damit die RAMs und das ROM optimal auf dem FPGA synthetisiert werden, werden dafür Module mit dem Programm `CoreGen` erzeugt. Auf dem im VAMP-Projekt benutzten VirtexE-FPGA sind zum Beispiel spezielle Block-RAM-Bereiche vorhanden, die von `CoreGen` benutzt werden.

3. Außenanbindung
   Einige Module, die nicht zum eigentlichen Prozessor gehören wurden von Hand

[3] Diese Registerzuweisung wird vereinfacht dargestellt. Im ursprünglichen Sourcecode lautet die Zeile `register <= #100 wire3`. Die zusätzliche Angabe `#100` bedeutet, dass die Zuweisung in einem Verilog-Simulator um hundert Zeiteinheiten verzögert wird. Ohne diese künstliche Verzögerung wird die Schaltung vom benutzten Verilog-Simulator *VerilogXL* nicht korrekt simuliert. Auf die synthetisierte Hardware hat der Zusatz keinerlei Auswirkungen.

in Verilog geschrieben. Dazu gehören das Speicher-Interface, das den Zugriff des VAMP auf die SD-RAM Chips regelt, die dem Prozessor als RAM dienen. Ebenso gehört dazu das Hostinterface, das die Anbindung des Prozessors an die Außenwelt regelt.

Bei welchen Funktionen PVS2HDL die Übersetzung abbrechen soll, weil diese schon als Verilog-Module vorliegen, wird in der Datei `pvs2hdl.conf` festgelegt[4].

2.3.1 Verbinden von externer Hardware mit generierten Modulen

Im VAMP ist es notwendig, Verilog-Module, die sich weit unten in der Modul-Hierarchie befinden, mit I/O-Pins des FPGAs oder externen Verilog-Modulen zu verbinden. Das ist auf direktem Weg nicht möglich, da in Verilog Verbindungen von einem Modul nur zum übergeordneten beziehungsweise den direkt untergeordneten Modulen angelegt werden können. Um dennoch die tief liegenden Module nach außen verbinden zu können, müssen diese schrittweise von einer Modul-Ebene zur nächsten verbunden werden, bis das Toplevel-Modul erreicht ist.

PVS2HDL führt dieses schrittweise Verbinden automatisch für alle Module aus, die in `pvs2hdl.conf` im Abschnitt `connect_outside_functions` aufgeführt sind. Beim VAMP-Prozessor sind das einige Module des Caches, die mit dem handgeschriebenen Speicher-Interface verbunden werden müssen.

2.4 Experimentelle Ergebnisse

Um zu testen, wie teuer und wie schnell die mit PVS2HDL generierte Hardware im Vergleich zu handgeschriebener Hardware ist, wurden mehrere Schaltkreise mit PVS2HDL von PVS nach Verilog übersetzt und auf einem Xilinx FPGA synthetisiert. Die Größe und die Geschwindigkeit wurden dann mit den Ergebnissen für handgeschriebenen Verilog-Code verglichen. Zusätzlich wurden einige Schaltkreise mit dem Programm Xilinx CoreGen generiert, das speziell für die Xilinx FPGAs optimierte Hardware erzeugt.

Die Ergebnisse dieses Tests sind in Abbildung 2.5 zu sehen. Die Kosten sind dabei in der Einheit *slices* angegeben, die mit dem Platzverbrauch auf dem FPGA korrespondiert (siehe Abschnitt 3.1). Die Verzögerung der Schaltkreise ist in Nanosekunden angegeben.

Auffallend ist, dass die Kosten der aus PVS übersetzten Hardware für den Addierer und den Multiplizierer wesentlich höher sind, als die der handgeschriebenen Varianten. Das ist darauf zurückzuführen, dass es auf dem VirtexE-FPGA spezielle Hardware zur schnelleren Addition gibt, die so genannte *schnelle Carry-Logik*. Diese spezielle Logik wird beim Synthetisieren des +-Operators von Verilog automatisch benutzt, beim Synthetisieren des aus PVS übersetzten Verilog-Codes dagegen nicht. Beim Multiplizierer nutzt die Xilinx Software zum Addieren der Teilergebnisse ebenfalls die schnelle Carry-Logik falls der $*$-Operator von Verilog benutzt wird. Deshalb ist der übersetzte Verilog-Code auch hier wesentlich langsamer und teurer. Alle übrigen aus PVS generierten Schaltkreise sind aber annähernd genauso schnell beziehungsweise teuer

[4]Der Aufbau der Datei `pvs2hdl.conf` wird in Anhang B beschrieben

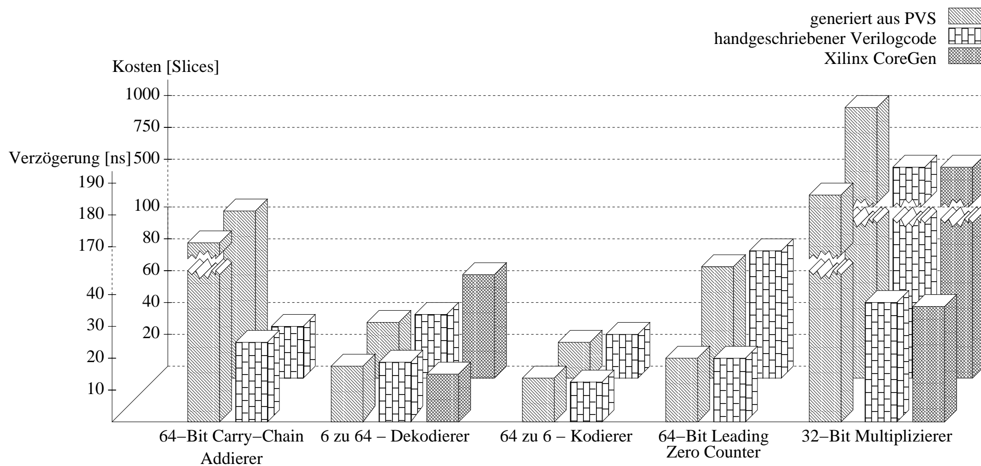

Abbildung 2.5: Vergleich zwischen übersetzter Hardware, handgeschriebenem Verilog-Code und Code aus dem Xilinx-Tool CoreGen

wie die handgeschriebenen Verilog-Module. Dies zeigt, dass die Beschreibung von Hardware in PVS und die anschließende Übersetzung nach Verilog vergleichbar gute Hardware erzeugen, wie handgeschriebener Verilog-Code. Mit dem CoreGen-Tool können leider nur wenige Schaltkreise generiert werden, aber auch bei diesen schneidet PVS2HDL zumindest beim Dekodierer sehr gut ab. Der mit PVS2HDL generierte Dekodierer ist nur unwesentlich langsamer, aber um mehr als ein Drittel kleiner.

2.5 Ähnliche Ansätze

Die Verifikation und Implementierung von Hardware mit Hilfe von Theorembeweisern und die anschließende Übersetzung in ein Hardwarebeschreibungssprache ist nicht grundsätzlich neu. Das Lambda-Toolkit[SMMB89] benutzt zum Beispiel ebenfalls diesen Ansatz. Eine ähnliche Methode benutzen Hanna und Daeche[HD85]. Die dort angegebenen Beispiele sind aber nicht annähernd so komplex wie der VAMP-Prozessor.

Den umgekehrten Weg geht zum Beispiel Russinoff in [Rus00]. Er übersetzt die Register-Transfer-Sprache von AMD in die Spezifikationssprache des Theorembeweisers ACL2[KM97]. Russinoff beweist als Beispiel die Korrektheit eines einfachen Fließkomma-Multiplizierers. Dennoch ist die Komplexität seiner Schaltkreise deutlich geringer als die Komplexität des VAMP-Prozessors. Ebenfalls diesen Weg geht Borrione mit dem PREVAIL-System[BBD+96, BPS92]. Das PREVAIL-System übersetzt Hardwarebeschreibungen in VHDL[IEE94, Ash96] in die Sprache des Boyer-Moore-Theorembeweisers Nqthm[KB95]. Er berichtet aber nicht über die Verifikation von komplexen Designs mit diesem System.

Kapitel 3

FPGA und PCI-Karte

In diesem Kapitel wird das FPGA beschrieben, auf dem der VAMP-Prozessor implementiert wird. Dabei wird zunächst in Abschnitt 3.1 auf die Grundstruktur des FPGAs eingegangen. Anschließend werden einige besondere Teile des FPGAs vorgestellt, die bei der Implementierung des Prozessors genutzt wurden. Das FPGA befindet sich auf einer PCI-Karte. Diese Karte wird in Abschnitt 3.2 näher beschrieben. Dabei werden zuerst in Abschnitt 3.2.1 die verschiedenen Komponenten der Karte vorgestellt. In den Abschnitten 3.2.2 und 3.2.3 wird dann das UMRBus-System zur Kommunikation zwischen FPGA und Host-PC beschrieben.

3.1 VirtexE-FPGA

Ein FPGA (**F**ield **P**rogrammable **G**ate **A**rray) ist ein programmierbarer Logikbaustein, der vom Benutzer beliebig oft neu programmiert werden kann. Dadurch wird die Entwicklung von Schaltkreisen wesentlich günstiger, da im Fehlerfall das FPGA einfach neu programmiert wird. Im Gegensatz zu FPGAs wird bei einem ASIC (**A**pplication **S**pecific **I**ntegrated **C**ircuit) die Funktion schon in der Fabrik festgelegt. Deshalb muss bei einem ASIC im Fehlerfall ein neuer Chip produziert werden. Neben den deutlich erhöhten Kosten fällt dabei besonders die lange Produktionsdauer von mehreren Wochen ins Gewicht.

3.1.1 Grundstruktur

Der VAMP-Prozessor wurde auf einem Xilinx FPGA vom Typ `VirtexE XCV2000` implementiert [Xil02]. Ein VirtexE-FPGA besteht aus einer Matrix von generischen Grundelementen, den konfigurierbaren Logikblöcken (CLB). Die CLBs sind untereinander durch eine komplexe Verbindungsstruktur (Routing Resourcen) verbunden. Für jeden CLB gibt es 24 kurze Routing-Verbindungen zu den benachbarten vier CLBs. Weiterhin gibt es 72 so genannte *Hex-Lines*, die eine Länge von sechs Blöcken haben, und für längere Verbindungen benutzt werden. Für noch längere Verbindungen gibt es zwölf *Long-Lines*, die über die gesamte Breite beziehungsweise Höhe des FPGAs reichen.

Die CLBs sind in mehreren großen CLB-Bereichen angeordnet, die durch *Block-RAM*-Bereiche unterbrochen sind (siehe Abbildung 3.1). Außerhalb der CLB-Matrix

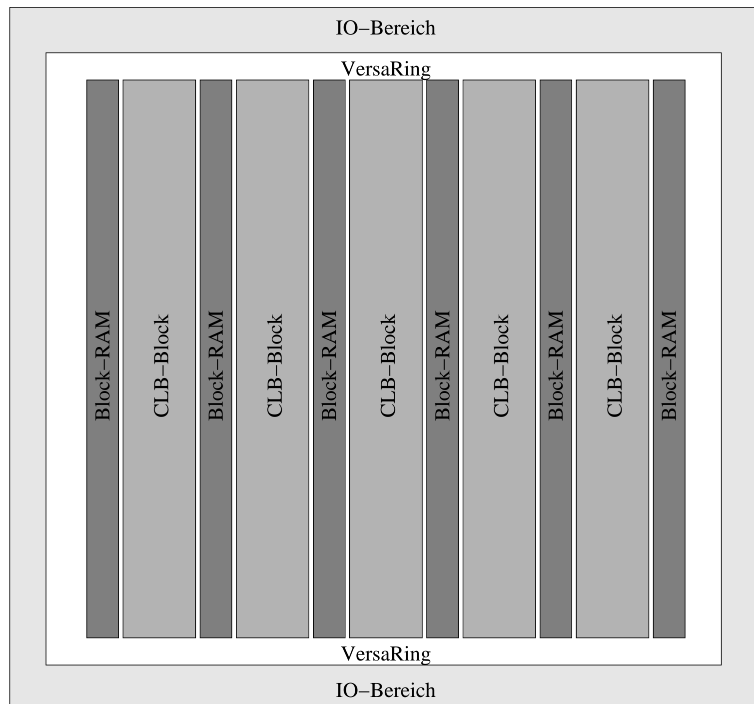

Abbildung 3.1: Grundstruktur eines VirtexE-FPGAs

befinden sich im so genannten *VersaRing* zusätzliche Routing Ressourcen für Verbindungen zwischen den CLB-Bereichen.

Das für den VAMP benutzte FPGA enthält 9.600 konfigurierbare Logikblöcke. In Abbildung 3.2 ist der Aufbau eines CLB zu sehen. Ein CLB besteht aus zwei identischen Hälften, den so genannten Slices. Ein Slice wird aus zwei Logikzellen gebildet, die eine Lookup-Tabelle (LUT), eine Kontrolleinheit und ein Register enthalten.

Die Lookup-Tabellen bestehen aus einem ein Bit breiten RAM mit vier Adressbits. Der Input der Lookup-Tabelle wird als Adresse an das RAM gelegt, und je nach RAM-Inhalt kann jede beliebige Logikfunktion mit vier Input-Bits realisiert werden. Über die Kontrolllogik der Logikzelle kann die Ausgabe der Lookup-Tabelle mit dem zusätzlichen `B`-Eingang der Logikzelle verknüpft werden. Als Ausgabe bietet eine Logikzelle zwei Bits, von denen eines über ein Register geleitet werden kann. Zusätzlich gibt es für jeden Slice noch einen speziellen Eingang und Ausgang für Überträge von Additionen. Dadurch können mit den CLBs sehr effizient Addierer konstruiert werden.

3.1.2 Block-RAM

Zusätzlich zu den Logikblöcken enthält das FPGA noch 640 KBit Dual-Port RAM. Dieser Speicher ist auf 160 Blöcke zu je vier KBit verteilt. Jeder der RAM-Blöcke kann auf eine Datenbreite von eins, zwei, vier, acht oder sechzehn Bit konfiguriert werden. Die Datenbreite kann auch für beide Zugriffsports des gleichen RAM-Blocks unterschiedlich sein. Die Anzahl der Speicherstellen des RAMs, auf die von einem Zugriffsport zugegriffen werden kann, ergibt sich dann als $\frac{4096}{Bitbreite}$. Die Block-RAMs haben zwei unabhängige Takt-Eingänge für die beiden Ports. Auf die Ausgänge der

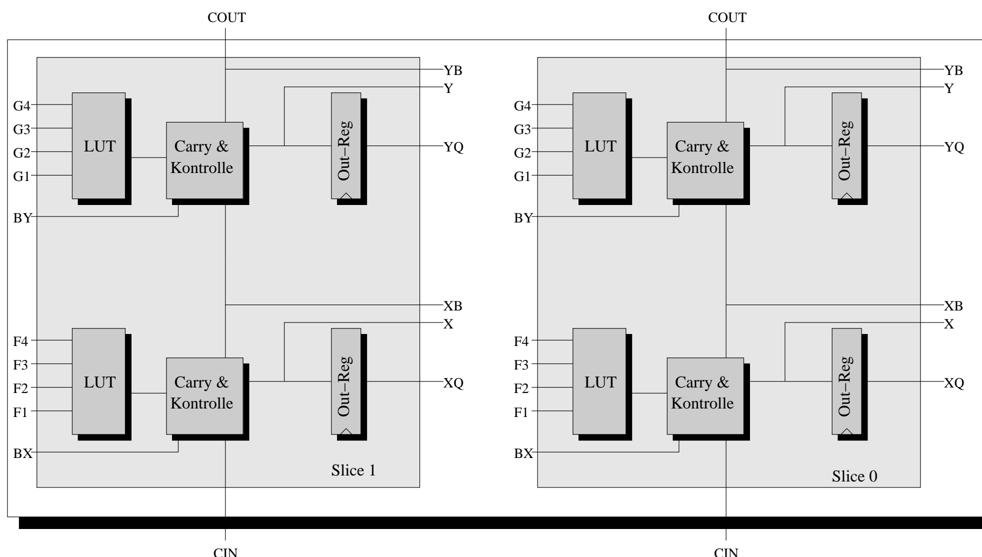

Abbildung 3.2: Konfigurierbarer Logikblock (CLB)

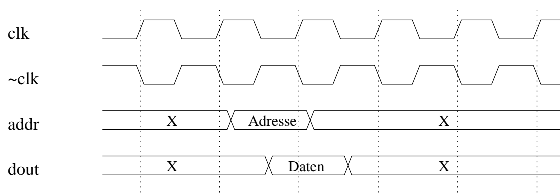

Abbildung 3.3: Timing von Block-RAMs mit invertiertem Takt am Ausgangsregister

RAMs kann nur über Register zugegriffen werden.

Im VAMP wird das Block-RAM für die Registerfiles und für den Cache benutzt. Im Cache werden die Daten aus den RAMs schon in dem Takt benötigt, in dem die Adresse angelegt wird. Da die Ausgänge der Block-RAMs über Register geschaltet sind, ist dies auf direktem Weg nicht möglich. Damit sich die Block-RAMs trotz Ausgangsregister so verhalten, als hätten sie keine, werden die Ausgangsregister mit invertiertem Taktsignal `~clk` betrieben. Das führt dazu, dass sich der Ausgang der Register bei fallender Taktflanke von `clk` ändert (also noch während des Taktes, in dem die Adresse angelegt wurde). In Abbildung 3.3 ist das Timing eines Lesezugriffs auf ein invertiert getaktetes Block-RAM zu sehen.

3.1.3 Takt-Generierung

Auf dem VirtexE-FPGA gibt es acht so genannte Delay Locked Loops (DLLs) [Xil02]. Mit diesen Schaltkreisen kann die Frequenz eines Taktsignals verdoppelt oder durch n dividiert werden ($n \in \{1.5, 2, 2.5, 3, 4, 5, 8, 16\}$).

Des Weiteren ist es mit Hilfe von DLLs möglich Clock-Skew zwischen zwei verschiedenen Clock-Signalen zu eliminieren. Unter Clock-Skew versteht man die Phasenverschiebung zwischen Taktsignalen, die entsteht, wenn die Taktsignale verschiedene Lasten treiben müssen oder verschieden lange Wege zurückzulegen haben. Ei-

ne DLL behebt den Clock-Skew zwischen Input-Clock `CLKIN` und Feedback-Clock `CLKFB`. Dazu wird die Ausgabeclock `CLK0` solange verzögert, bis die steigende Taktflanke von `CLKIN` synchron zur steigenden Taktflanke von `CLKFB` ist. Zusätzlich wird über die Ausgänge `CLK90`, `CLK180` und `CLK270` eine phasenverschobene Version von `CLK0` ausgegeben.

Nach einem Reset braucht die Delay Locked Loop einige hundert Taktzyklen, bis sie die Clocks synchronisiert hat. Währenddessen können beliebige Signale an den Ausgängen erscheinen. Damit die Benutzer-Hardware erkennen kann, wann die Initialisierung der DLL abgeschlossen ist, gibt es das Signal `LOCKED`. Nachdem die DLL am `LOCKED`-Ausgang eine `1` ausgibt, wird ein stabiles und synchronisiertes Taktsignal garantiert.

Über die PCI-Karte, auf der sich das FPGA befindet, wird der VAMP mit der 33 MHz schnellen PCI-Clock versorgt. Da der VAMP auf dem benutzten VirtexE-2000 FPGA aber mit maximal 10 MHz funktioniert, wird die PCI-Clock mit Hilfe des `CLKDV`-Ausgangs einer DLL auf 8,25 MHz geviertelt.

Der VAMP-Prozessor benutzt als Arbeitsspeicher vier SD-RAM-Chips, die sich auf der PCI-Karte befinden (siehe auch Abschnitt 3.2.1). Die Bezeichnung SD-RAM steht für *synchrones dynamisches RAM*. *Synchron* bedeutet, dass die SD-RAM-Chips im Gegensatz zu normalen D-RAM-Chips ein Taktsignal besitzen. Dieses Taktsignal muss möglichst synchron zum Taktsignal des Prozessors sein. *Dynamisch* heißen die RAMs, weil sie ihre Information in winzig kleinen Kondensatoren speichern. Da die Kondensatoren innerhalb weniger Mikrosekunden ihre Ladung verlieren, muss der Inhalt der SD-RAM-Chips ständig vom Speichercontroller in so genannten *Refresh*-Zyklen aufgefrischt werden.

Die Leitungen zwischen dem FPGA und den SD-RAM-Chips haben eine relativ große Kapazität. Deshalb entsteht zwischen der FPGA-Clock und der SD-RAM-Clock Clock-Skew, falls dies nicht durch besondere Maßnahmen verhindert wird. Um den Clock-Skew zu eliminieren wird eine Schaltung aus zwei DLLs benutzt. In Abbildung 3.4 ist zu sehen, wie die beiden DLLs dazu verschaltet werden. Die Prozessor-Clock wird aus dem FPGA auf die PCI-Karte geführt. Von dort wird die Clock dann zu den SD-RAM-Chips geführt und gleichzeitig über einen Eingangs-Pin des FPGA über den `CLKFB` Anschluss als Feedback-Signal wieder an die DLL zurückgebracht.

3.2 PCI-Karte

Die PCI-Karte vom Typ EBPCI der Firma IsyTec [ISY01a] ersparte die Entwicklung und das Debugging eines eigenen Boards für das FPGA. Die EBPCI-Karte ermöglicht dem FPGA die Kommunikation mit dem Host-PC über dessen PCI-Bus. Zusätzlich stellt sie zwei Steckverbinder zur Verfügung, deren Pins direkt mit I/O Pins des FPGAs verbunden sind. Die Karte enthält verschiedene Arten von RAM-Bausteinen, und es gibt die Möglichkeit eigene Quarze zur Generierung von Clock-Signalen einzusetzen.

3.2.1 Aufbau der Karte

Der schematische Aufbau der PCI-Karte ist in Abbildung 3.5 zu sehen. Im Folgenden werden die wichtigsten Bestandteile kurz erläutert:

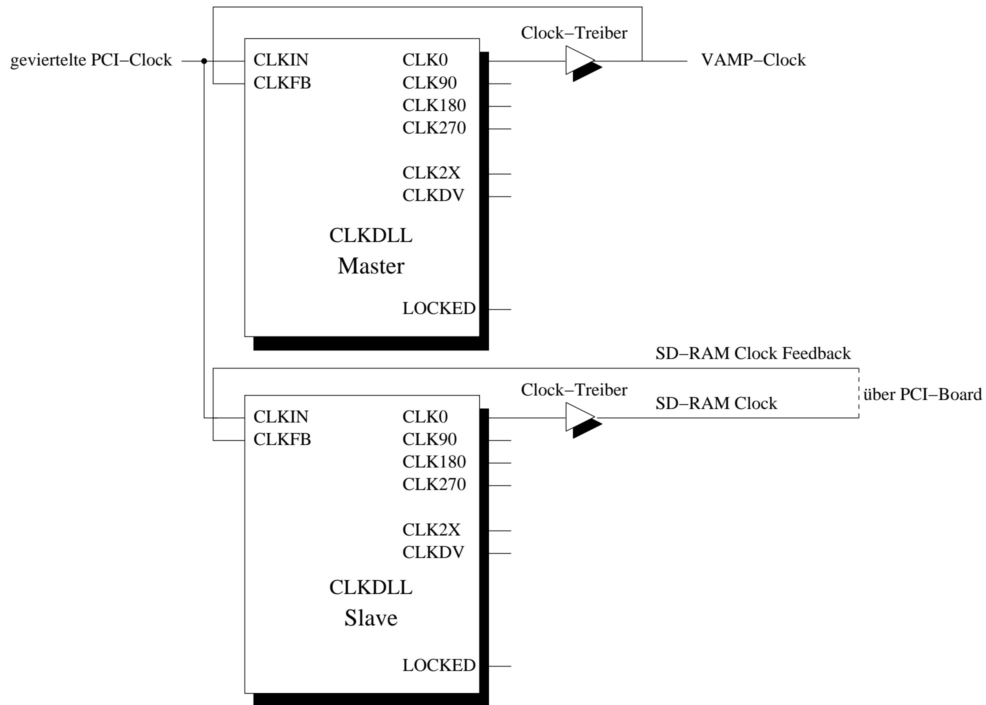

Abbildung 3.4: Eliminierung von Clock-Skew zwischen SD-RAM Clock und Prozessor-Clock

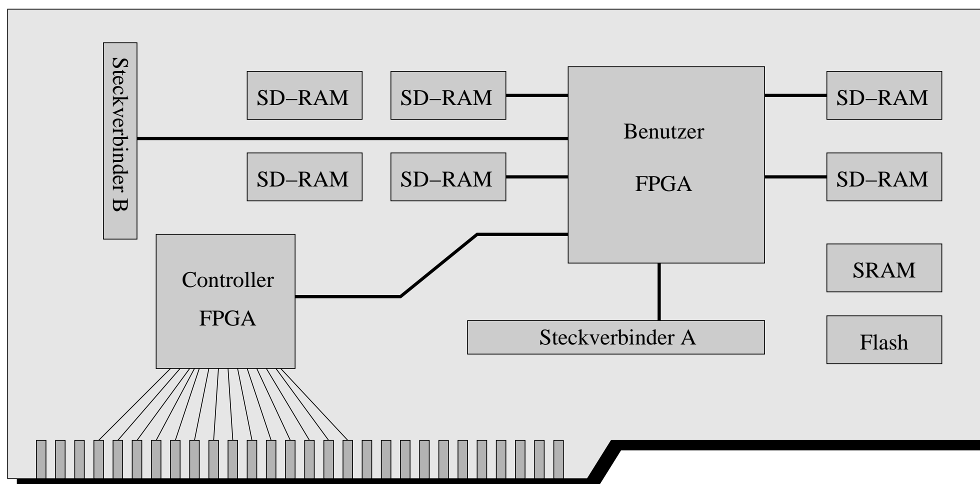

Abbildung 3.5: Aufbau der PCI-Karte

- Benutzer-FPGA
  Das Benutzer-FPGA vom Typ VirtexE-2000-6BG560 [Xil02] ist der Hauptbestandteil der Karte. Es kann vom Host-Rechner beliebig programmiert werden. Das Benutzer-FPGA nimmt sowohl das generierte Design des Prozessors als auch die handgeschriebenen Teile (Speicher-Interface, Host-Interface; siehe Kapitel 4) auf.

- Controller-FPGA
  Das Controller-FPGA kann vom Benutzer nicht programmiert werden. Es übernimmt die Anbindung der Karte an den PCI-Bus und stellt dem Benutzer-FPGA in Form des UMRBus-Protokolls eine einfache Schnittstelle zum Host-Rechner zur Verfügung.

- SD-RAM
  Auf der Karte befinden sich sechs SD-RAM Chips von Fujitsu [Fuj97]. Diese SD-RAM Chips bieten jeweils 2^{22} Speicherstellen mit 16 Bit Breite. Vier dieser SD-RAM Chips werden für den 64 Bit breiten Hauptspeicher des VAMP-Prozessors parallel angesteuert. Sie bieten somit insgesamt 32 Megabyte Hauptspeicher. Die restlichen beiden SD-RAM Chips auf der Karte bleiben ungenutzt, da sie zusammen nur 32 Bit breit sind und deshalb für den 64 Bit breiten Speicherbus des VAMP nicht genutzt werden können.

- SRAM und Flash-Speicher
  Neben dem SD-RAM sind auf der EBPCI-Karte auch noch ein Megabyte SRAM und zwei Megabyte Flash-Speicher vorhanden. Diese werden für den VAMP nicht genutzt. Für den Cache des VAMP wird statt des SRAM das interne Block-RAM des FPGAs benutzt, da dieses noch schneller ist als das externe SRAM und zudem in verschiedenen Bitbreiten organisiert werden kann.

- Schnittstellen
  Der Hostrechner kann auf das Benutzer-FPGA per PCI-Bus über das UMRBus-Interface (siehe Abschnitt 3.2.2) zugreifen. Des Weiteren befinden sich auf der EBPCI-Karte noch zwei Steckverbinder, mit 34 beziehungsweise 26 Pins, die direkt an I/O Pins des FPGAs angeschlossen sind. Über diese Steckverbinder können Signale aus dem FPGA nach außen gelegt werden, um sie mit einem Logic-Analyzer oder Oszilloskop zu untersuchen.

3.2.2 UMRBUS

Das UMRBus-System [ISY01b] der Firma IsyTec ermöglicht eine einfache Kommunikation zwischen dem Benutzer-FPGA und dem Host-PC. Dass der Transport der Daten über den PCI-Bus durchgeführt wird, ist sowohl für den VAMP als auch für die Steuer-Software auf dem Host-PC vollständig transparent.

Über den UMRBus können bis zu 32 Bit breite Datenpakete übertragen werden. Weiterhin kann das Benutzer-FPGA Interrupts generieren, die mit einer 16 Bit breiten Interrupt-ID gegenüber dem Steuerprogramm eindeutig identifiziert sind.

In Abbildung 3.6 ist zu sehen, wie der VAMP über den UMRBus mit dem Host-PC verbunden ist. Es gibt folgende Komponenten:

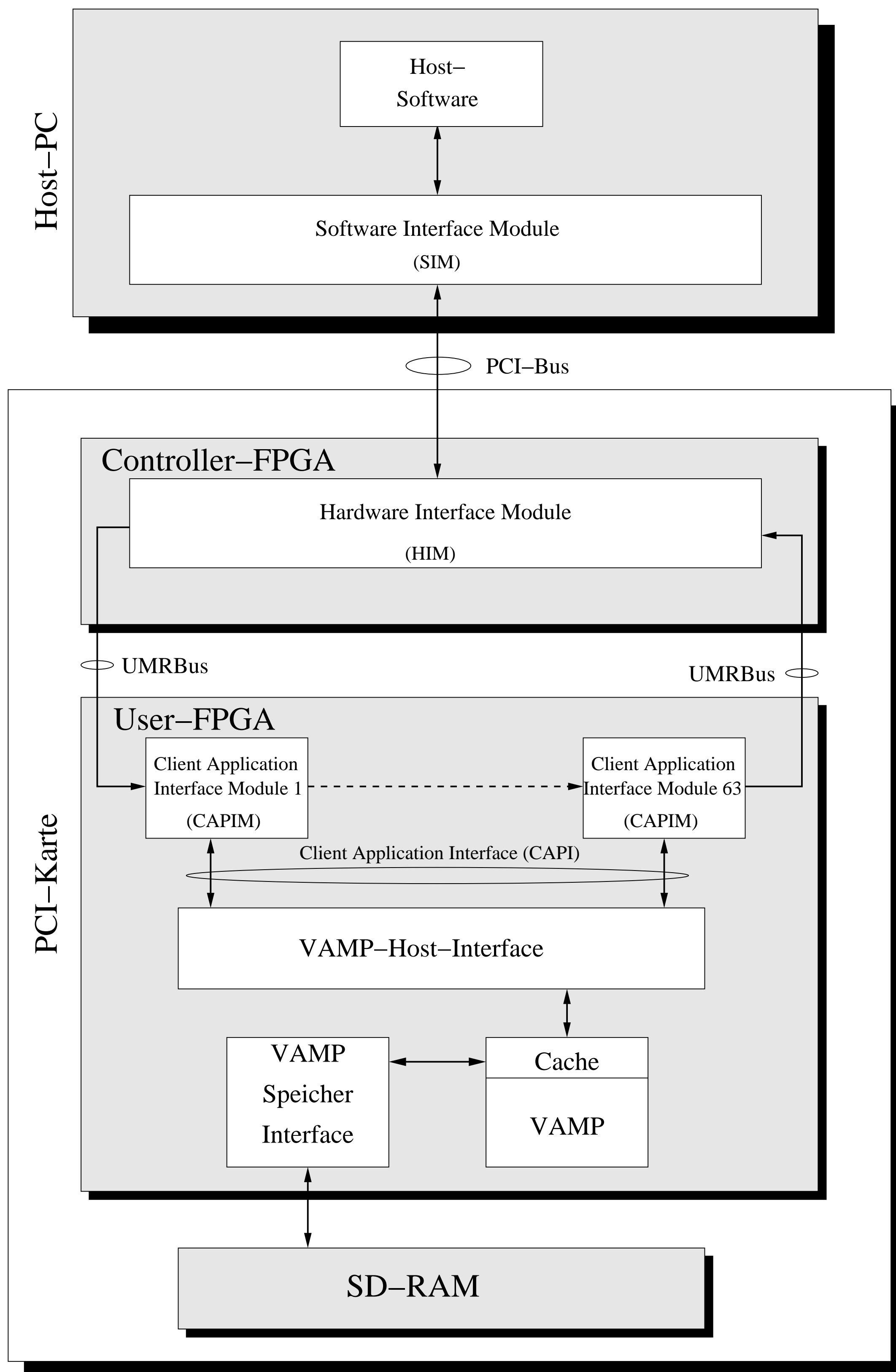

Abbildung 3.6: Verbindung von VAMP und Host-PC

- Host-Software
  Die Host-Software ist das Steuerprogramm, das auf dem Host-PC läuft und für die Verwaltung des VAMP-Prozessors zuständig ist. Die Host-Software ermöglicht es, auf den Speicher des VAMP zuzugreifen. Dadurch können von außen Programme und Daten abgelegt und die Ergebnisse der Programme ausgelesen werden. Die Steuersoftware (siehe [Mey02]) übernimmt auch sämtliche Ein- und Ausgabeoperationen für den Prozessor.

- Software und Hardware Interface Module
  Das Software Interface Module und das Hardware Interface Module verstecken das Übertragungsmedium PCI-Bus sowohl vor der Host-Software als auch vor dem Benutzer-FPGA. Dadurch halten sie den Zugriff auf den UMRBus vom zugrunde liegenden Übertragungsmedium PCI-Bus unabhängig. Beide Teile stammen von der Firma IsyTec.

- CAPIM : Client Application Interface Module
  Die Capims bilden die Schnittstelle zwischen dem UMRBus und der im VAMP-Projekt entwickelten Hardware. Sie befinden sich innerhalb des Benutzer-FPGAs. Ein einzelner Capim bietet der Benutzer-Hardware einen 32 Bit breiten Zugang zum UMRBus. Es können bis zu 63 Capims an den UMRBus angeschlossen werden. Jedes dieser Capims hat eine eindeutige Nummer und kann so vom Steuerprogramm identifiziert und einzeln angesprochen werden.

- Host-Interface, Speicher-Interface
  Das Host-Interface und das Speicher-Interface übernehmen die Anbindung des VAMP an die Umgebung. Sie bestehen aus handgeschriebenen Verilog-Code und werden in Kapitel 4 genauer vorgestellt.

3.2.3 Schnittstelle zu den CAPIMs

Die Software-Schnittstelle zum UMRBus und die VAMP-Steuersoftware wird in der Diplomarbeit von Carsten Meyer [Mey02] genauer beschrieben. Auf die Hardwareschnittstelle zu den Capims wird im Folgenden genauer eingegangen.

In Abbildung 3.7 sind die Verbindungen zwischen den Capims und dem Host-Interface des VAMP zu sehen. Die einzelnen Signale haben folgende Bedeutung:

- `DO` (Data out)
  32 Bit breiter Datenbus vom Capim zum Host-Interface.

- `WR` (Write enable)
  Mit `WR` signalisiert das Capim, dass gültige Daten an `DO` anliegen und vom Host-Interface gelesen werden können.

- `DI` (Data in)
  32 Bit breiter Datenbus von der Benutzer-Hardware zum Capim.

- `RD` (Read enable)
  Mit `RD` signalisiert das Capim, dass es die an `DI` liegenden Daten gelesen hat, und dass das Host-Interface die nächsten Daten anlegen kann.

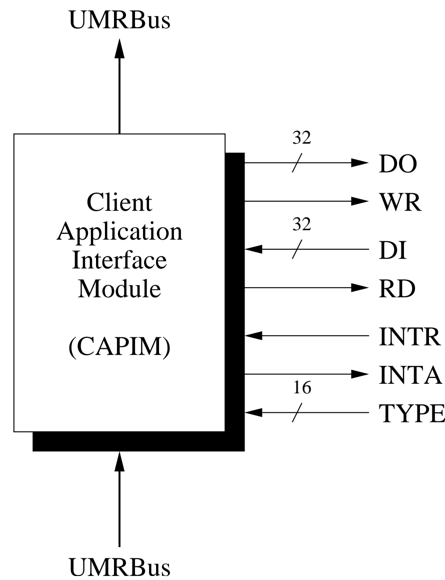

Abbildung 3.7: Interface zu den Capims

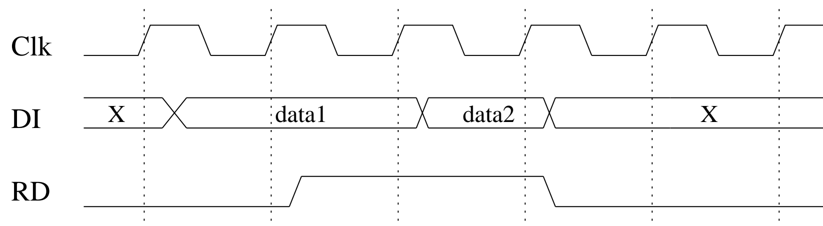

Abbildung 3.8: Schreibzugriff der Host-Software auf das Capim-Interface

- `INTR` (Interrupt request)
  Mit diesem Signal kann das Host-Interface einen Interrupt beim Steuerprogramm anfordern. `INTR` muss aktiv bleiben, bis das Signal `INTA` aktiv wird.

- `INTA` (Interrupt acknowledge)
  `INTA` bestätigt, dass ein mit `INTR` angeforderter Interrupt vom Steuerprogramm bearbeitet wurde.

- `TYPE` (Interrupt ID)
  16 Bit breiter Bus, der bei einer Interruptanforderung die Interrupt-ID enthält.

Im Folgenden sind Timing-Diagramme für die Signale zwischen Capim und Benutzer-Hardware zu sehen.

In Abbildung 3.8 ist zu sehen, wie die Benutzer-Hardware über den UMRBus Daten an die Host-Software schickt. Es gibt kein Signal, mit dem die Benutzer-Hardware der Host-Software mitteilen könnte, dass Daten gelesen werden sollen. Es gibt zwei Möglichkeiten, dieses Problem zu lösen. Zum einen kann die Benutzer-Hardware einen Interrupt generieren, wenn sie neue Daten angelegt hat. Als zweite Möglichkeit kann die Benutzer-Software die Capims regelmäßig auslesen und so auf neue Daten prüfen.

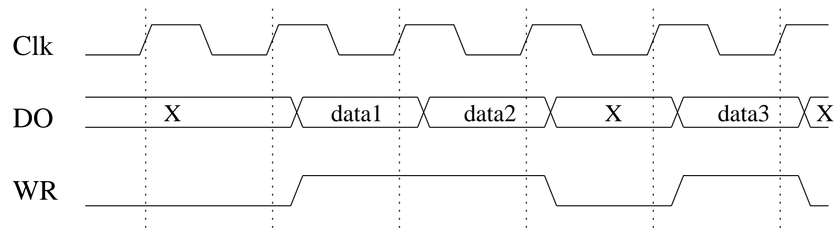

Abbildung 3.9: Lesezugriff der Host-Software auf das Capim-Interface

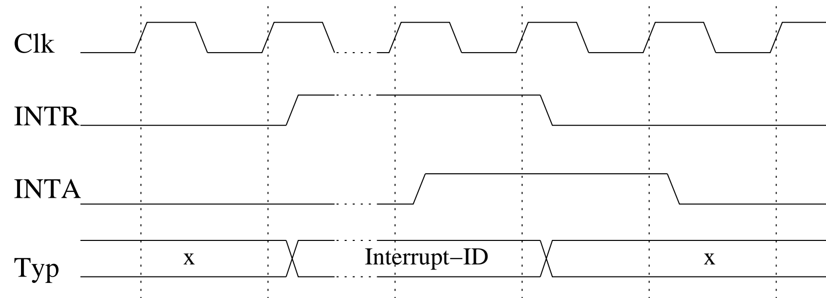

Abbildung 3.10: Interrupt-Request über das Capim-Interface

Im Host-Interface des VAMP wird die erste Möglichkeit benutzt. Nachdem die Daten von der Host-Software gelesen worden sind, aktiviert diese für einen Takt das Signal `RD`. Im nächsten Takt muss die Benutzer-Hardware das nächste Datenwort anlegen. Wenn `RD` mehrere Takte am Stück aktiv bleibt, wird in jedem Taktzyklus ein neues Datenwort übermittelt.

In Abbildung 3.9 sieht man, wie die Host-Software Daten an die Benutzer-Hardware übermittelt. Gleichzeitig mit gültigen Daten auf `DO` wird das Signal `WR` aktiviert. Falls mehrere Datenworte in einem Durchgang übertragen werden sollen, bleibt `WR` länger als einen Takt aktiv ist. Dann wird mit jeder steigenden Taktflanke ein neues Datenwort auf `DO` angelegt. Das bedeutet insbesondere, dass die Hardware genügend schnell die Daten entgegennehmen muss, wenn die Host-Software mehrere Datenworte direkt hintereinander übermittelt. Beim VAMP-Prozessor schickt die Host-Software aber mit jedem Zugriff nur ein einzelnes Datenwort, so dass damit keine Probleme entstehen.

In Abbildung 3.10 sieht man einen Interrupt-Request der Benutzer-Hardware an die Host-Software. Zuerst aktiviert die Benutzer-Hardware das Signal `INTR`. Nachdem die Host-Software den Interrupt registriert hat, reagiert das Capim mit der Aktivierung von `INTA`. Einen Takt später muss die Benutzer-Hardware `INTR` wieder deaktivieren. Solange `INTR` aktiviert ist, muss die Benutzer-Hardware eine gültige Interrupt-ID an `TYPE` anlegen.

Kapitel 4

Implementierung

Im ersten Teil dieses Kapitels wird die handgeschriebene Hardware beschrieben, die zusätzlich zu der aus PVS-Code generierten Hardware geschrieben wurde. In Abschnitt 4.1 wird die Reset-Logik des VAMP beschrieben, die die verschiedenen Reset-Signale des Prozessors steuert. Als weiterer handgeschriebener Teil wird in Abschnitt 4.2 das Host-Interface beschrieben. Das Host-Interface ist für die Anbindung des Prozessors an den Host-PC zuständig ist. In Abschnitt 4.3 wird das Speicher-Interface beschrieben, das für die Anpassung des VAMP-Speicherprotokolls an das Protokoll des verwendeten SD-RAM Controllers sorgt. Die Konstruktion von RAM mit drei Zugriffsports, wie es in den Register-Files benötigt wird, wird in Abschnitt 4.4 vorgestellt. In Abschnitt 4.5 werden weitere Hardwareteile vorgestellt, die nicht aus den PVS-Sourcen generiert werden.

In Abschnitt 4.6 wird erläutert, wie aus den Verilog-Sourcen die Binärdatei zur Konfiguration des FPGAs generiert wird. Abschließend werden in Abschnitt 4.7 noch einige charakteristische Zahlen für den implementierten VAMP-Prozessor genannt.

4.1 Reset-Logik

Die Reset-Logik (siehe Anhang C.3) berechnet die Reset-Signale für den SD-RAM-Controller und den Cache. Diese beiden Reset-Signale werden nach dem Hochladen des VAMP-Designs auf das FPGA automatisch gesteuert. Man benötigt zwei verschiedene Reset-Signale, da die SD-RAM Chips eine relativ lange Initialisierungsphase haben, während der der Cache noch nicht aktiv sein darf. Die Reset-Signale für den SD-RAM-Controller und den Cache sind unabhängig vom Reset-Signal für die CPU. Das Reset-Signal der CPU wird von der Host-Software gesteuert (siehe Abschnitt 4.2.3).

Die Reset-Logik hat zwei interne Zähler, mit deren Hilfe bestimmt wird, wann die beiden Reset-Signale deaktiviert werden. Solange die Zähler unterhalb eines Schwellwertes sind, werden sie in jedem Takt inkrementiert. Während dieser Zeit ist das zugehörige Reset-Signal aktiv. Wenn einer der Zähler seinen Schwellwert erreicht, wird er nicht mehr weiter erhöht, und das entsprechende Reset-Signal wird deaktiviert. Das Reset-Signal für den SD-RAM-Controller wird vier Takte nach der Initialisierung des FPGAs deaktiviert. Der Cache wird wegen der langen Initialisierungsphase der SD-RAMs erst 508 Takte später gestartet.

Bei der Initialisierung des FPGAs werden alle Register, und damit auch die beiden

Tabelle 4.1: Aufteilung der VAMP-Steuersignale auf die vier Capims

Capim	Bits	Signal
1	31:0	Daten [63:32]
2	31:0	Daten [31:0]
3	31:10	Adresse
	9	`wr`-Signal
	8:1	Byte-Enable-Signale
	0	ungenutzt
4	31	CPU-Reset
	30:0	ungenutzt

Zähler, mit Null initialisiert. Die Reset-Logik selbst benötigt deshalb keinen Reset-Eingang, da sie automatisch neu startet, sobald die Zähler zurückgesetzt werden.

4.2 Host-Interface

Das Host-Interface ermöglicht es dem Host-Programm, auf den Speicher des VAMP zu zugreifen und das Reset-Signal des Prozessors zu steuern. Das Host-Interface enthält dazu vier Capims, von denen drei für die Speicherzugriffe zuständig sind. Das vierte steuert das Reset-Signal der CPU. Weiterhin stellt das Host-Interface das geviertelte Takt-Signal für den VAMP zur Verfügung. In Abbildung 4.1 ist der grundlegende Aufbau des Host-Interfaces zu sehen[1]. Der Quelltext des Host-Interfaces ist in Anhang C.2 zu sehen. In Tabelle 4.1 ist die Zuordnung der Steuersignale für den VAMP-Prozessor auf die vier Capims zu sehen.

Der `DO`-Bus der Capims enthält nur einen Takt lang gültige Daten. Damit die Daten länger zur Verfügung stehen, werden sie in den Input-Registern zwischengespeichert. Als Clock-Enable-Signal für die Input-Register wird das `WR`-Signal des jeweiligen Capims benutzt. Dadurch wird erreicht, dass der Inhalt der Register nur dann aktualisiert wird, wenn am Capim auch neue Daten anliegen.

4.2.1 Synchronisation

Die Capims müssen Daten mit dem PCI-Bus austauschen und sind deshalb mit 33 MHz getaktet. Weil der VAMP-Prozessor nur mit einem Viertel dieser Frequenz arbeitet, muss eine Synchronisation zwischen den Capims und dem VAMP-Prozessor stattfinden. Diese Synchronisation übernimmt das Host-Interface. Die Input-Register des Host-Interfaces sowie ein Teil der Kontrolle werden dazu mit 33 MHz getaktet. Von der Host-Software wird garantiert, dass mit weniger als 8,25 MHz Frequenz Daten an die Capims gesendet werden. Dadurch ändert sich der Inhalt der Input-Register so langsam, dass sie vom langsam getakteten Teil des Host-Interfaces ausgewertet werden können.

[1]Die Berechnung der Clock-Enable-Signale für die Output-Register ist in der Zeichnung nicht abgebildet. Sie kann dem Quelltext des Host-Interfaces in Anhang C.2 entnommen werden.

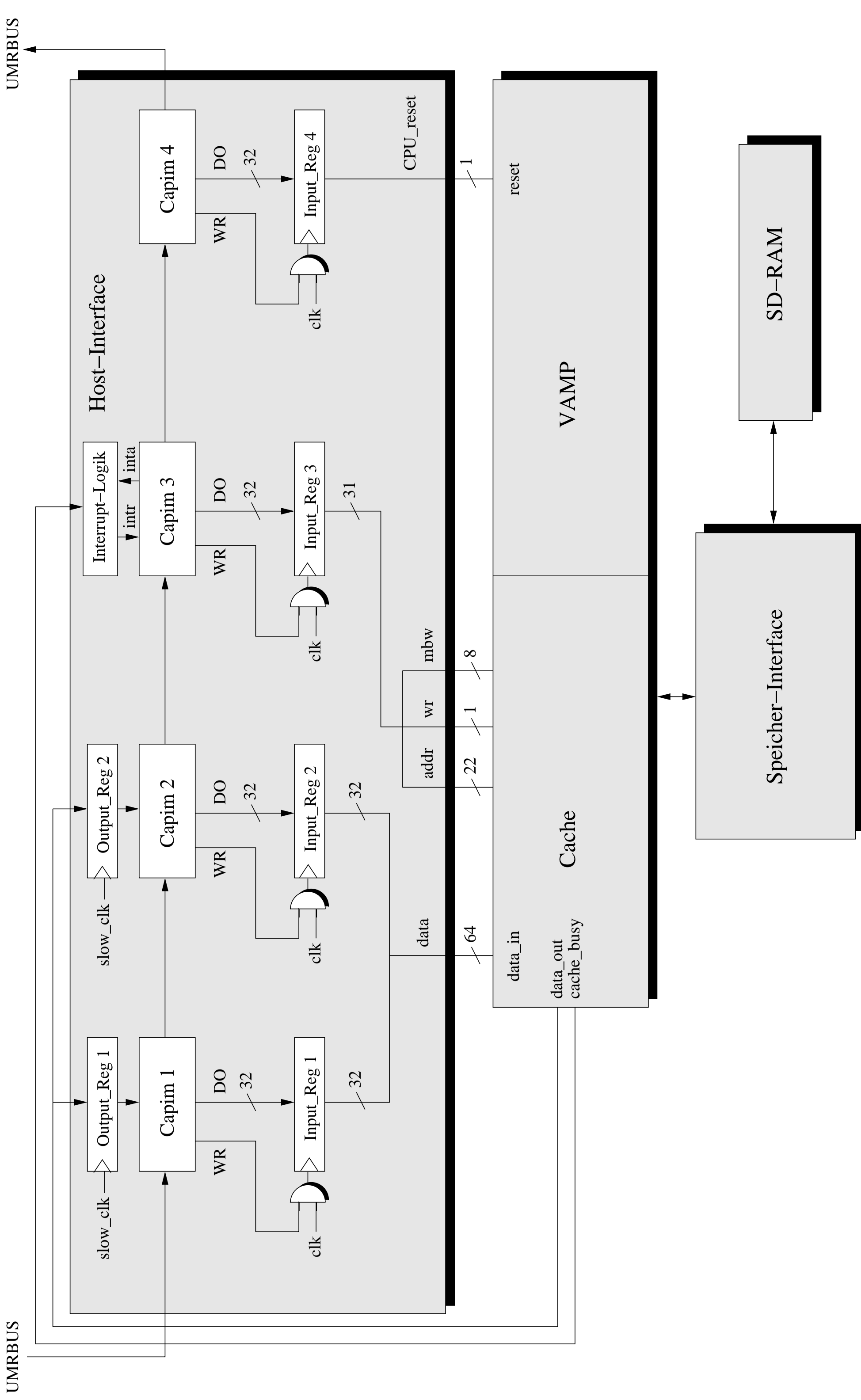

Abbildung 4.1: Das Host-Interface

Wie in Abschnitt 4.2.2 beschrieben wird, wird ein Speicherzugriff der Host-Software durch einen Schreibzugriff auf Capim 3 gestartet. Bei einem solchen Schreibzugriff wird vom Capim für einen Takt das Signal `WR` aktiviert. Da das Capim mit den vollen 33 MHz getaktet wird, muss das `WR`-Signal durch den schnellen Teil des Host-Interfaces ausgewertet werden. Der schnelle Teil des Host-Interfaces erzeugt dann das Signal `memory_req_slow`, das dem langsam getakteten Teil des Host-Interfaces signalisiert, dass ein Speicherzugriff initiiert wurde. Sobald der langsame Teil des Host-Interfaces das aktivierte `memory_req_slow` erkennt, aktiviert er für einen Takt das Request-Pending Signal `memory_reqp_slow` und beginnt mit der Ausführung des Speicherzugriffes. Nach der Aktivierung von `memory_reqp_slow` deaktiviert der schnelle Teil des Host-Interfaces `memory_req_slow` wieder. Das Signal `memory_req_slow` wird nach folgender Gleichung berechnet:

```
memory_req_slow <= (memory_req_slow | capim3_wr)
                   & ~memory_reqp_slow
```

Wenn ein Interrupt an Capim 3 für die Host-Software generiert werden soll, aktiviert der langsame Teil des Host-Interfaces das Signal `generate_intr` für einen Takt. Mit Hilfe dieses Signals generiert der schnelle Teil des Host-Interfaces dann das Signal `intr` von Capim 3. Das Signal wird folgendermaßen berechnet[2]:

```
if (generate_intr & ~capim3_intr)
  capim3_intr <= `HI;
if (capim3_intr & capim3_inta)
  capim3_intr <= `LO;
```

Wie in Anhang C.2 zu erkennen ist, wird der größte Teil des Host-Interfaces langsam getaktet. Mit der schnellen Clock werden nur die Input-Register,die Berechnung von `memory_req_slow` sowie die Steuerung des Interrupt-Signals `capim3_intr` angesteuert.

4.2.2 Speicherzugriffe

Die Host-Software kann über die Capims sowohl Schreib- als auch Lesezugriffe auf den Speicher des VAMP ausführen. Burst-Zugriffe werden über die Capims nicht unterstützt. Damit es nicht zu Inkonsistenzen mit dem Inhalt des Cache kommt, kann das Host-Interface nicht direkt auf den Speicher zugreifen, sondern muss seine Zugriffe an den Cache übergeben. Der Cache hat hierzu neben dem Zugriffsport für den VAMP-Prozessorkern noch einen zweiten Port, der für die Speicherzugriffe durch den Host-PC zuständig ist. Wie in Abbildung 4.1 zu sehen, verbindet der Cache sowohl VAMP als auch das Host-Interface mit dem 64 Bit breiten Speicher.

Schnittstelle zwischen Host-Interface und Capims

Die Capims 1 und 2 sind für die 64 Bit breiten Daten zuständig. Über die Bits 31 bis 10 von Capim 3 wird die 22 Bit breite Adresse übertragen. Das Bit neun von Capim 3 überträgt das `wr`-Signal, das bei einem Schreibzugriff aktiviert und bei einem Lesezugriff deaktiviert ist. Die Bits acht bis eins sind die Byte-Enable Signale. Diese geben für jedes der acht Bytes der 64-Bit Daten an, ob es geschrieben werden soll

[2] In den Original-Sourcen steht bei den beiden Registerzuweisungen noch eine Verzögerungsanweisung für den Verilog-Simulator. Siehe dazu auch die Fußnote auf Seite 21.

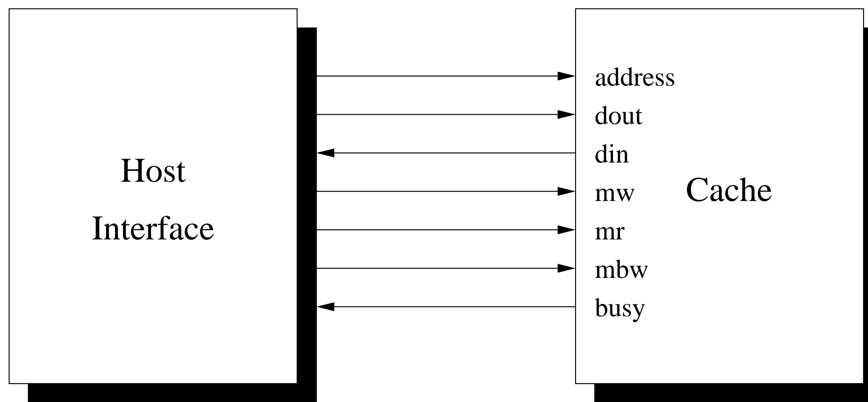

Abbildung 4.2: Der zweite Zugriffs-Port des Caches

oder nicht. Dadurch ist es möglich, nur einzelne Bytes im Speicher zu aktivieren und den Rest des 64-Bit Datenwortes unverändert zu lassen. Durch Aktivierung des `wr`-Signals von Capim 3, also einen schreibenden Zugriff der Host-Software auf dieses Capim, wird der eigentliche Speicherzugriff gestartet.

Bei einem *Schreibzugriff* auf den Speicher müssen von der Host-Software zunächst die Daten auf die Capims 1 und 2 geschrieben werden. Anschließend wird der Schreibzugriff durch einen Zugriff auf Capim 3 gestartet. Wenn der Speicherzugriff erfolgreich abgeschlossen wurde, löst das Host-Interface an Capim 3 einen Interrupt-Request aus. Die Host-Software wartet nach dem Absetzen eines Schreibbefehls auf diesen Interrupt. Wenn der Interrupt nicht auftritt gibt die Software eine Fehlermeldung aus.

Auch ein *Lesezugriff* wird durch einen Schreibvorgang auf Capim 3 gestartet. Wenn der Cache die Daten geliefert hat, werden sie in den Output-Registern gespeichert. Anschließend wird ein Interrupt-Request auf Capim 3 ausgelöst. Sobald die Host-Software diesen Interrupt registriert, kann sie die gelesenen Daten aus den Capims 1 und 2 auslesen.

Schnittstelle zwischen Host-Interface und Cache

Die Signale des zusätzlichen Zugriffs-Ports des Caches sind in Abbildung 4.2 zu sehen. Die Kommunikation des Host-Interfaces mit dem Cache übernimmt ein Zustandsautomat, der in Abbildung 4.3 zu sehen ist. Der Automat gehört zum langsam getakteten Teil des Host-Interfaces. Er generiert unter anderem das Signal `memory_reqp_slow` zur Kommunikation mit dem schnellen Teil des Host-Interfaces. Der Quelltext des Automaten ist in Anhang C.2 abgedruckt.

Bei einem *Schreibzugriff* (Abbildung 4.4) wird vom Host-Interface zunächst `mw` aktiviert. Gleichzeitig werden an `mbw` die Byte-Enable-Signale, an `address` die Adresse und an `dout` die Daten an den Cache übertragen. Im nächsten Takt aktiviert der Cache das `busy`-Signal. Das Host-Interface geht dann in den Zustand `write` und wartet, bis `busy` wieder deaktiviert wird. Danach setzt es `mw` wieder auf `0` und geht in den Zustand `interrupt`. In diesem Zustand wird ein Interrupt an Capim 3 generiert. Dadurch erhält die Host-Software die Bestätigung, dass ihr Speicherzugriff erfolgreich abgeschlossen wurde. Nach dem Aktivieren des Interrupts geht das Host-

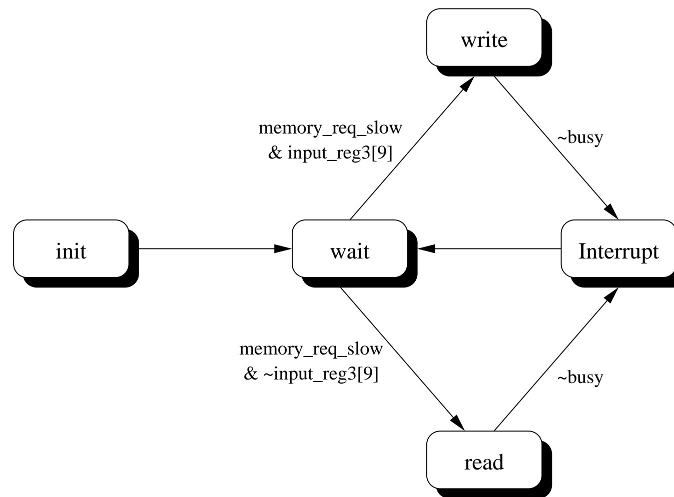

Abbildung 4.3: Zustandsautomat für die Kommunikation zwischen Host-Interface und Cache

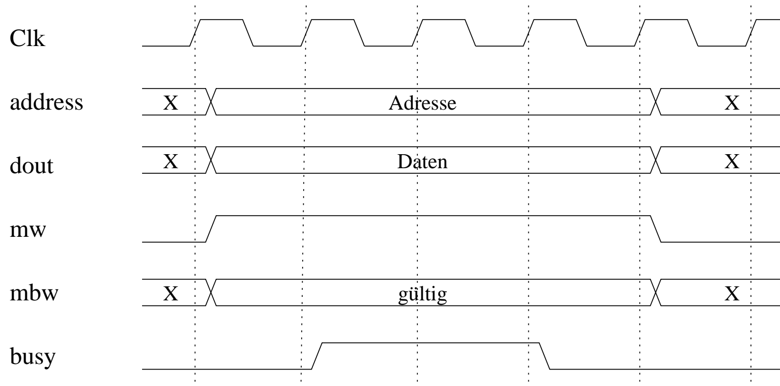

Abbildung 4.4: Schreibzugriff auf den zweiten Cache-Port

Interface wieder in den Zustand `wait`.

Bei einem *Lesezugriff* (Abbildung 4.5) aktiviert das Host-Interface zunächst `mr` und legt die Adresse an `address` an. Der Cache aktiviert im nächsten Takt `busy`. Das Host-Interface geht nun in den Zustand `read` und wartet bis der Cache `busy` wieder deaktiviert. Sobald das geschieht, wird `mr` wieder deaktiviert und von `din` werden die angeforderten Daten entgegengenommen. Diese werden dann an die Output-Register der Capims 1 und 2 weitergegeben. Anschließend wechselt das Host-Interface, wie bei den Schreibzugriffen, in den Zustand `interrupt` und informiert die Host-Software darüber, dass die gewünschten Daten in den Capims 1 und 2 bereitstehen. Danach geht das Host-Interface wieder in den Zustand `wait`.

4.2.3 CPU-Reset

Die auszuführenden Programme müssen in den Speicher geschrieben werden, bevor der VAMP-Prozessor gestartet wird. Da die Daten über den Cache in den Speicher

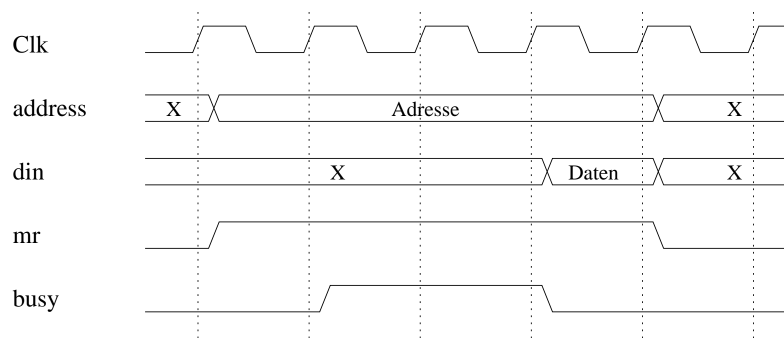

Abbildung 4.5: Lesezugriff auf den zweiten Cache-Port

geschrieben werden, ist es notwendig, dass Cache und VAMP unabhängig voneinander resettet werden. Deshalb gibt es zusätzlich zum Reset-Signal für den Cache noch den CPU-Reset, über den nur der Prozessor resettet wird.

CPU-Reset wird im Gegensatz zu den anderen Reset-Signalen (siehe Abschnitt 4.1) direkt über die Capims gesteuert. Für das Reset-Signal ist der Capim 4 zuständig. Von den 32 Bits seines `DO`-Busses wird nur das oberste benutzt. Der entsprechende Ausgang des Input-Registers 4 ist direkt mit dem Reset-Eingang des Prozessors verbunden.

4.3 Speicher-Interface

Zur Ansteuerung der SD-RAM Chips auf der PCI-Karte wird ein SD-RAM Controller der Firma IsyTec verwendet.

Der SD-RAM-Controller wird über ein proprietäres Protokoll angesteuert. Der Cache des VAMP dagegen benutzt das Speicherprotokoll aus [MP00]. Da diese Protokolle nicht direkt zueinander kompatibel sind, muss zwischen Cache und SD-RAM-Controller ein Protokollumsetzer eingesetzt werden. Diese Aufgabe übernimmt das Speicher-Interfaces. Das Speicher-Interface unterstützt sowohl Einzel- als auch Burst-Zugriffe. Burst-Zugriffe haben eine feste Länge von 4 Datenworten. Ein Datenwort umfasst 64 Bit, dadurch ergibt sich eine Burstlänge von insgesamt 32 Byte.

In Abbildung 4.6 ist zu erkennen, über welche Signale das Speicher-Interface mit dem Cache und dem Speicher verbunden ist.

4.3.1 Die Kontrolle des Speicher-Interfaces

Das Speicher-Interface implementiert einen Zustandsautomaten, der die Speicherzugriffe des Caches entgegen nimmt und daraus die passenden Steuersignale für den IsyTec SD-RAM-Controller generiert. In Abbildung 4.7 ist ein vereinfachtes Diagramm dieses Automaten zu sehen. Insbesondere die Zustände für die Burst-Zugriffe wurden darin vereinfacht, um die Zeichnung übersichtlich zu halten. Die Details sind im beschreibenden Text, in den Timing-Diagrammen und im Quelltext in Anhang C.1 zu finden.

Nach einem Reset startet die Kontrolllogik im Zustand `init`. In diesem Zustand initialisiert das Speicher-Interface die Signale `reqp`, `BRdy`, `sdc_rnw` und `sdc_req`

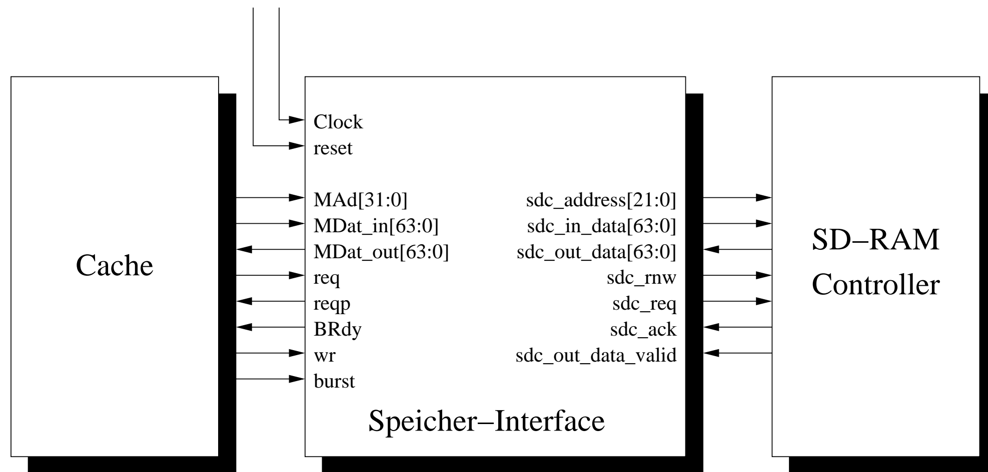

Abbildung 4.6: Das Speicher-Interface

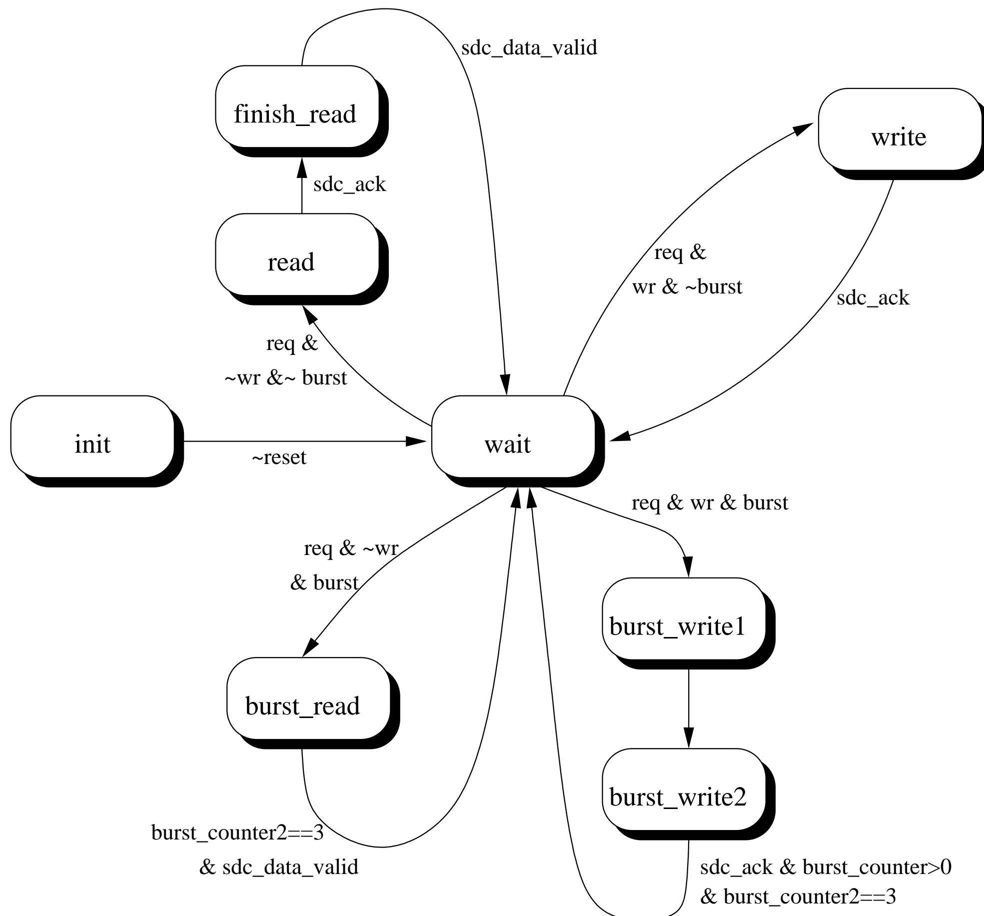

Abbildung 4.7: Zustandsautomat des Speicher-Interfaces

mit 0. Danach bleibt das Speicher-Interface im Zustand wait, bis der Cache mit dem req-Signal einen Speicherzugriff signalisiert. Einen Takt danach aktiviert das Speicher-Interface das Signal reqp. Damit signalisiert es dem Cache, dass der Speicherzugriff noch bearbeitet wird. Abhängig davon, welchen Wert die Signale wr und burst haben, führt das Speicher-Interface eine von den vier möglichen Aufgaben aus:

Einzelner Lesezugriff: wr=0, burst=0 (Abbildung 4.8)

Zu Beginn eines einzelnen Lesezugriffes wechselt das Speicher-Interface in den Zustand read. Dabei gibt es die unteren 22 Bit der Adresse MAd über sdc_address an den SD-RAM-Controller weiter und aktiviert sowohl sdc_rnw als auch sdc_req. Im Zustand read bleibt das Speicher-Interface dann, bis der SD-RAM-Controller mit sdc_ack den Lesezugriff bestätigt. Das Speicher-Interface wechselt nun in den Zustand finish_read und deaktiviert sdc_req wieder. Im Zustand finish_read bleibt das Speicher-Interface bis der SD-RAM Controller über sdc_data_valid signalisiert, dass die angeforderten Daten an sdc_out_data anliegen. Nun aktiviert das Speicher-Interface BRdy und deaktiviert reqp. Dadurch signalisiert es dem Cache, dass dieser einen Takt später die angeforderten Daten an MDat_out auslesen kann. Gleichzeitig werden die Daten in das MDat_out_delay-Register geschrieben. Durch dieses Register werden die Daten um einen Takt verzögert, da der Cache die Daten erst einen Takt nach der Aktivierung von BRdy erwartet. Danach wechselt das Speicher-Interface wieder in den Zustand wait.

Einzelner Schreibzugriff: wr=1, burst=0 (Abbildung 4.9)

Das Speicher-Interface wechselt zunächst in den Zustand write. Da der Cache die Daten erst einen Takt nach dem Aktivieren des req-Signals liefert, gibt das Speicher-Interface den Schreibzugriff erst mit einem Takt Verzögerung an den SD-RAM Controller weiter. Dabei übergibt es an sdc_in_data die Daten und an sdc_address die Adresse. Das Signal sdc_req ist währenddessen aktiviert, und sdc_rnw ist deaktiviert. Sobald der SD-RAM-Controller mit sdc_ack den Abschluss des Schreibzugriffs signalisiert, wird BRdy für einen Takt aktiviert. Danach wird reqp deaktiviert und das Speicher-Interface geht wieder in den Wartezustand.

Burst-Lesezugriff: wr=0, burst=1 (Abbildung 4.10)

Bei einem Burst-Lesezugriff wechselt das Speicher-Interface zunächst in den Zustand burst_read. Dabei gibt es die Adresse von MAd direkt an sdc_address weiter und aktiviert gleichzeitig sdc_req und sdc_rnw.

In den nächsten Takten werden die passenden Adressen für den SD-RAM-Controller generiert und die gelesenen Daten entgegengenommen. Da die ersten Daten schon eintreffen können, während noch Adressen übermittelt werden, gibt es in diesem Zustand zwei unabhängige Teil-Schaltkreise. Die beiden Teil-Schaltkreise arbeiten gleichzeitig. Ein Schaltkreise sorgt für das Generieren der Adressen für den SD-RAM-Controller, während der andere die angeforderten Daten entgegennimmt. Mit Hilfe von zwei Registern, burst_counter und burst_counter2, wird mitgezählt, wie viele Adressen schon generiert beziehungsweise wie viele Daten schon

entgegen genommen worden sind. Sobald alle Adressen an den SD-RAM-Controller übermittelt worden sind (burst_counter=3), wird das sdc_req Signal wieder deaktiviert.

Gültige Daten signalisiert der SD-RAM-Controller durch sdc_data_valid. Wenn dieses Signal aktiv ist, wird burst_counter2 inkrementiert und BRdy aktiviert. Gleichzeitig wird das aktuelle Datenwort von sdc_out_data im Verzögerungsregister MDat_out_delay gespeichert. Da der Inhalt von MDat_out_delay bei jeder steigenden Taktflanke an MDat_out übermittelt wird, wird das Datenwort einen Takt später automatisch an MDat_out ausgegeben. Wenn alle vier Datenworte übertragen worden sind, wird reqp deaktiviert und das Speicher-Interface geht wieder in den Zustand wait.

Der SD-RAM-Controller kann die Datenübertragung anhalten, indem er das Signal sdc_data_valid wieder deaktiviert. Dann wird auch BRdy deaktiviert, und das Speicher-Interface wartet bis zur nächsten Aktivierung von sdc_data_valid.

Burst-Schreibzugriff: wr=1, burst=1 (Abbildung 4.11)

Bei einem Burst-Schreibzugriff wechselt das Speicher-Interface zunächst in den Zustand burst_write1 und wartet einen Takt, weil der Cache das erste Datenwort erst einen Takt nach dem Aktivieren von req anlegt.

Dann geht das Speicher-Interface in den Zustand burst_write2. In diesem Zustand gibt es ähnlich zum Burst-Lesezugriff zwei unabhängige Teil-Schaltkreise. Einer der beiden Schaltkreise nimmt die vier Datenworte nacheinander vom Cache entgegen und speichert sie in Registern zur späteren Verwendung durch den zweiten Teil-Schaltkreis. Nach dem Empfang des zweiten Datenwortes wird reqp deaktiviert und einen Takt später auch BRdy. Nach dem Deaktivieren von BRdy übermittelt der Cache noch ein Datenwort an das Speicher-Interface. Danach ist der Zugriff für den Cache abgeschlossen, obwohl der SD-RAM-Controller die Daten noch nicht vollständig im SD-RAM gespeichert hat. Falls der Cache einen erneuten Speicherzugriff initiiert, bevor der SD-RAM-Controller mit dem Schreiben der Daten fertig ist, wird der Cache zum Warten gezwungen, bis der SD-RAM-Zugriff abgeschlossen ist. Dazu wird reqp vom Speicher-Interface aktiviert, während gleichzeitig BRdy deaktiviert ist.

Der zweite Teil-Schaltkreis übernimmt die Kommunikation mit dem SD-RAM Controller. Nach dem Übergang in den Zustand burst_write2 wird zunächst das Signal sdc_req aktiviert und sdc_rnw deaktiviert. Gleichzeitig werden mittels sdc_address die erste Adresse und an sdc_in_data das erste Datenwort übergeben. Sobald der SD-RAM-Controller mit sdc_ack seine Bereitschaft zur Übernahme der Daten signalisiert, legt der zweite Schaltkreis nacheinander die restlichen drei Datenworte an sdc_in_data an, die vom ersten Schaltkreis in den Registern zwischengespeichert worden sind. Mit jedem neuen Datenwort wird die Adresse, die an sdc_address angelegt wird, inkrementiert. Die Datenübergabe kann vom SD-RAM-Controller jederzeit durch Deaktivieren von sdc_ack angehalten werden. Die Datenübergabe wird dann erst wieder fortgesetzt, wenn sdc_ack erneut aktiviert wird. Nachdem alle vier Datenworte übertragen worden sind, deaktiviert das Speicher-Interface sdc_req und geht wieder in den Wartezustand wait.

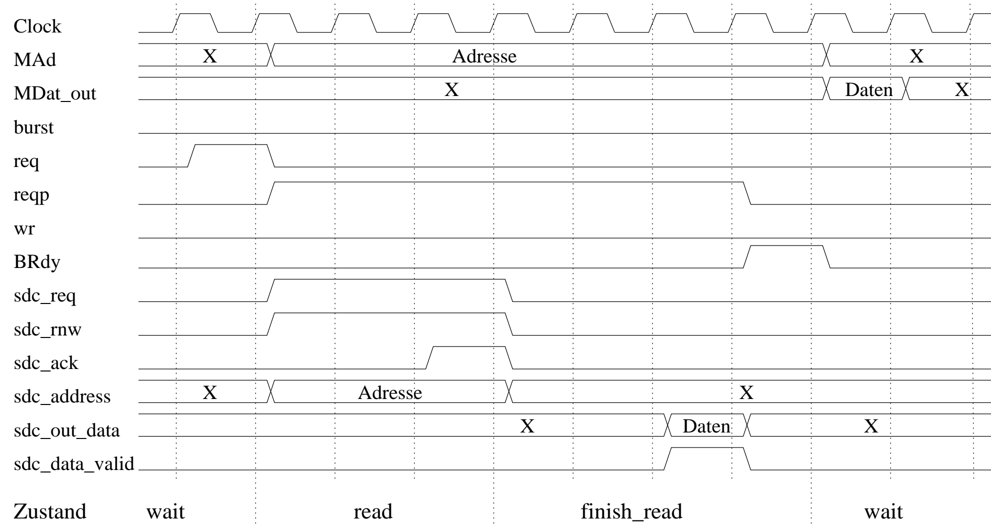

Abbildung 4.8: Einzelner Lesezugriff

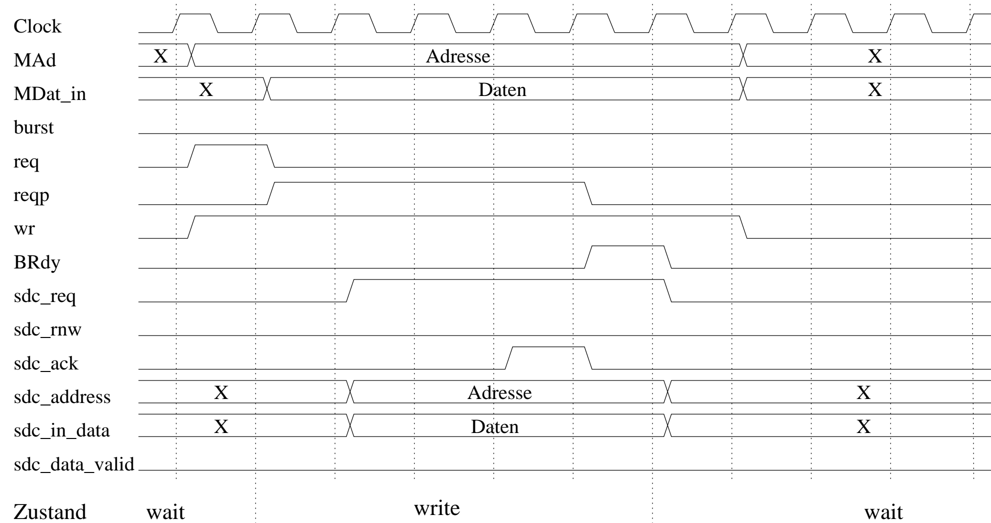

Abbildung 4.9: Einzelner Schreibzugriff

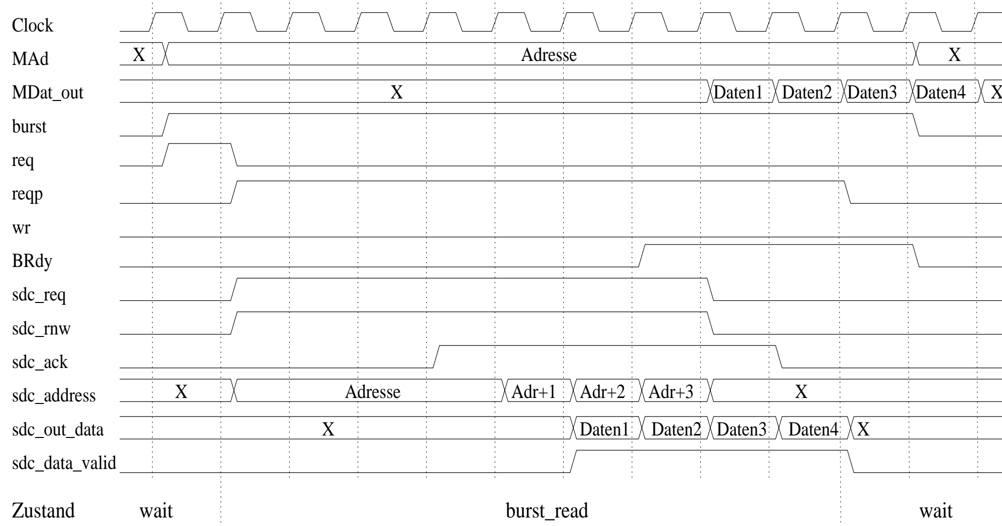

Abbildung 4.10: Burst Lesezugriff

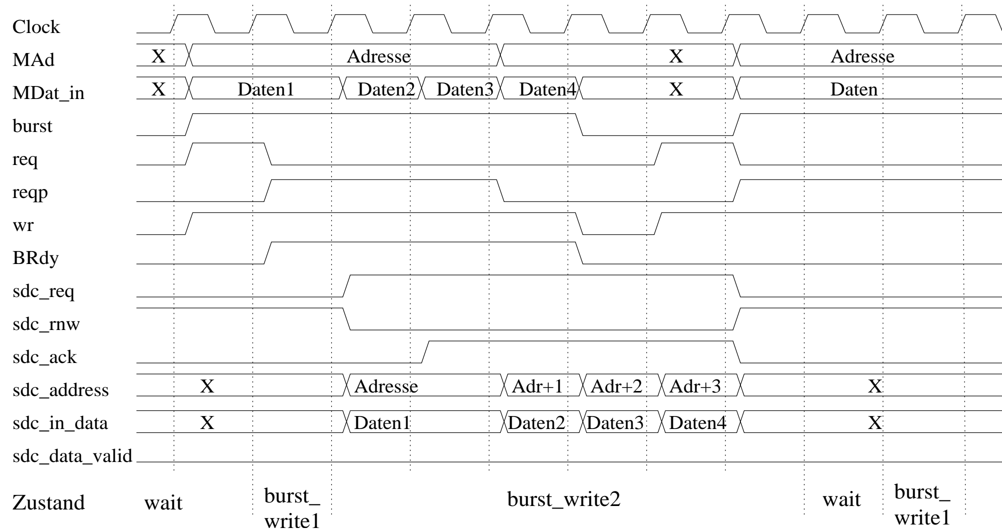

Abbildung 4.11: Burst Schreibzugriff mit erneutem Zugriff vor Beendigung des ersten Zugriffs

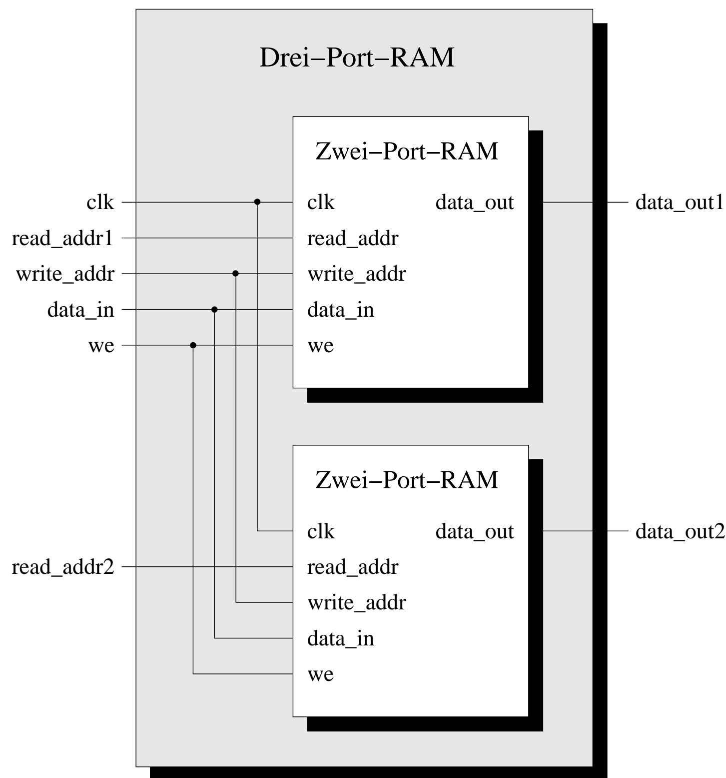

Abbildung 4.12: Zusammenschaltung von zwei Zwei-Port-RAMs zu einem Drei-Port-RAM

4.4 Drei-Port-RAM für die Register-Files

Im VAMP-Prozessor werden für einen Befehl maximal zwei Operanden aus einem Registerfile gelesen und eventuell gleichzeitig ein Ergebnis in ein Registerfile zurückgeschrieben. Für die beiden Register-Files des VAMP-Prozessors werden deshalb RAMs mit drei Zugriffsports (zwei Leseports und ein Schreibport) benötigt. Auf dem VirtexE-FPGA stehen aber nur Zwei-Port-RAMs direkt als vorgefertigte Module zur Verfügung. Deshalb wird für die Register-Files aus jeweils zwei Zwei-Port-RAMs (ein Leseport und ein Schreibport) ein Drei-Port-RAM gebaut.

Die Zusammenschaltung der beiden Zwei-Port-RAMs ist eine Standardkonstruktion. Sie ist in Abbildung 4.12 zu sehen. Schreibzugriffe auf das Drei-Port-RAM werden an beide Zwei-Port-RAMs weitergegeben, damit der Inhalt der RAMs konsistent bleibt. Die Anschlüsse `clk`, `write_addr`, `data_in` und `we` der Zwei-Port-RAMs sind dazu mit einander verbunden. Die beiden Leseports arbeiten jeweils nur auf einem der beiden Zwei-Port-RAMs.

4.5 CoreGen-Module und grundlegende Schaltkreise

Wie schon in Abschnitt 2.3 erwähnt, werden einige Module des VAMP-Prozessors nicht aus PVS-Code erzeugt. Außer den Modulen zur Außenanbindung, die in den letzten beiden Abschnitten beschrieben wurden, zählen dazu Speicherelemente sowie

einige grundlegende Schaltkreise.

Zu den Speicherelementen, die mit CoreGen generiert werden, gehören die Registerfiles sowie die verschiedenen RAMs innerhalb der beiden Caches (History-RAM[3], Tag-RAM, Valid-RAM und Data-RAM). Diese Module werden mit Hilfe von Block-RAM Elementen synthetisiert. Alle Speicherstellen des History-RAMs werden bei einem FPGA-Reset mit `11100100` (Daten-Cache[4]) beziehungsweise mit `10` (Instruktions-Cache[5]) initialisiert. Die Speicherstellen der anderen RAMs werden mit Nullen initialisiert. Die RAM-Elemente der Caches werden, wie in Abschnitt 3.1.2 beschrieben wird, mit einem invertierten Taktsignal angesteuert.

Auch die Lookup-Tabelle des Fließkomma-Dividierers wird mit CoreGen als ROM generiert. Die Datei, die den Inhalt des ROMs bestimmt, wird aber nicht von Hand erzeugt, sondern mit einem Skript aus den verifizierten PVS-Sourcen generiert.

Die grundlegenden Schaltkreise, die nicht aus PVS-Code erzeugt werden, sind auch in einer verifizierten PVS-Version vorhanden. Dennoch werden sie aus Effizienzgründen mit den jeweiligen Verilog Operatoren implementiert. Folgende Schaltkreise werden ersetzt: Gleichheitstester, Nulltester, Oder-Bäume, Inkrementer, Carry-Chain-Addierer, Multiplizierer.

4.6 Synthetisierung des Verilog-Codes

Der Verilog-Code des VAMP-Prozessors wird mit dem Programm *FPGA-Express* von Xilinx in eine Binärdatei umgewandelt. Mit dieser Binärdatei wird dann das FPGA programmiert. Für die Umwandlung wird ein Synthese-Skript der Firma IsyTec verwendet. Der Umwandlungsprozess erfolgt damit vollautomatisch ausgehend von den Verilog-Sourcen bis zur Erzeugung der Binärdatei.

Um die Zeit, die für die Umwandlung benötigt wird, einigermaßen klein zu halten, wird die Umwandlung mit minimaler Optimierung und ohne manuelles Floorplaning durchgeführt. Mit maximaler Optimierung belegt der Prozessor etwa 6 Prozent weniger Fläche auf dem FPGA. Die maximale Taktfrequenz ist dann etwa ein Megahertz höher als mit minimaler Optimierung. Mit maximaler Optimierung dauert die Umwandlung der Verilog-Sourcen aber auch fast zehn Stunden, im Vergleich zu vier Stunden bei minimaler Optimierung.

4.7 Zahlen

Der PVS-Sourcecode des VAMP-Prozessors besteht aus etwa 400 PVS-Dateien mit insgesamt über 45000 Zeilen PVS-Code. Die Übersetzung des PVS-Sourcecodes nach Verilog mit PVS2HDL dauert auf einem aktuellen Computer[6] etwa eine Minute und

[3] Die Caches des VAMP-Prozessors sind mehrfach assoziativ. Die Cache-Lines werden bei Bedarf mittels LRU-Replacement (siehe [MP00]) ersetzt. Dazu wird im History-RAM eine Liste gespeichert, die festhält, in welcher Reihenfolge die letzten Zugriffe auf die Cache-Lines erfolgt sind. Für jede Line wird im History-RAM ihre Position innerhalb dieser Liste gespeichert.

[4] vierfach assoziativ, deshalb $(4 * 2)$-Bit breites History-RAM

[5] zweifach assoziativ, deshalb $(2 * 1)$-Bit breites History-RAM

[6] AMD Athlon XP 1500+ Prozessor, 1,2 GByte Hauptspeicher, Windows XP

Tabelle 4.2: Einige Bestandteile des VAMP

Bestandteil	Größe in LUTs	Anteil am Gesamtprozessor
Speicher-Interface	506	1,3%
Host-Interface	943	2,5%
SD-RAM-Controller	164	0,4%
Additive FPU	3.091	8,0%
Multiplikative FPU	8.487	22,1%
Misc-FPU	2.422	6,3%

belegt maximal etwa 150 MByte Arbeitsspeicher. Die Verilog-Datei, die dabei erzeugt wird, ist 3,5 Megabyte groß und besteht aus etwa 150.000 Zeilen Code.

Das Umsetzen des generierten Verilog-Codes in ein FPGA-Design durch FPGA-Express dauert bei minimaler Optimierung etwa vier Stunden. Der VAMP-Prozessor belegt dann, inklusive des Host- und des Speicher-Interfaces, 38443 LUTs (siehe Abschnitt 3.1.1) und damit etwa 96 Prozent des FPGAs. Der Platzbedarf einiger Prozessorteile ist in Tabelle 4.2 zu sehen. Insgesamt benutzt der Prozessor 9095 Flip-Flops, von denen 8668 zum generierten Verilog-Code gehören und 427 durch den handgeschriebenen Code erzeugt werden. Für die Caches werden insgesamt 72 der 160 Block-RAM Blöcke benötigt. Die maximale Taktfrequenz, mit der der VAMP-Prozessor auf dem benutzten FPGA arbeitet würde, beträgt 10,7 MHz.

Kapitel 5

Projekt-Status

In diesem Kapitel wird auf den aktuellen Stand des VAMP-Projektes eingegangen.

Das Übersetzungstool PVS2HDL ist fertig gestellt. Es wurde erfolgreich durch die Übersetzung und den anschließenden Test der Fließkomma-Einheit,der Caches und schließlich des gesamten VAMP geprüft.

Die Verifikation der Fließkomma-Einheit des VAMP ist abgeschlossen. In einem Test auf dem FPGA mit mehreren hunderttausend Testvektoren arbeitete die Fließkomma-Einheit fehlerfrei. Die Beweise für die Caches sind fertig gestellt, ihre Anbindung an die Tomasulo-Pipeline ist erst teilweise verifiziert. Die Caches wurden zusammen mit dem Speicher-Interface erfolgreich auf dem FPGA getestet.

Die Verifikation des Tomasulo-Schedulers und der Einbettung der anderen Einheiten (Fließkomma-Einheit, Memory-Einheit) ist noch nicht abgeschlossen und wird zur Zeit fertig gestellt.

Die Softwareumgebung für den Prozessor (C-Compiler, Laufzeitumgebung zur Steuerung von Ein- und Ausgabe, GNU libc) ist fertiggestellt. Sie wurde mit verschiedenen kleineren C-Programmen (Berechnung von e mit Hilfe einer Näherungsformel, Berechnung von π mit einem statistischen Verfahren) getestet und funktioniert. Ein Test mit umfangreicheren Programmen steht noch aus.

Anhang A

Aufrufkonvention von PVS2HDL

Das Übersetzungtool PVS2HDLwird folgendermaßen aufgerufen:

```
pvs2hdl [Optionen] Eingabedatei Registertyp Ausgabedatei
```

oder

```
pvs2hdl --parse-only [Optionen] Eingabedatei
```

`Eingabedatei` ist dabei die PVS-Datei mit der Funktion, die zum Toplevel-Modul des Designs werden soll.

`Registertyp` ist der Name des PVS-Datentypes der im Design zur Modellierung von Registern benutzt wird (siehe dazu Abschnitt 2.2.8).

`Ausgabedatei` ist der Name der Datei, in die der generierte Verilogcode ausgegeben wird. Falls die Datei schon vorhanden ist wird sie überschrieben.

`Optionen` ist eine durch Leerzeichen getrennte Liste von keiner, einer oder mehreren der folgenden Optionen:

- `--pvs-path PATH`
  Gibt an in welchem Verzeichnis PVS installiert ist. Wenn diese Option nicht angegeben ist, wird `/usr/local/pvs` benutzt. Dieses Verzeichnis wird beim Suchen von PVS-Bibliotheken verwendet.

- `--verbose`
  Erzeugt sehr viele eventuell hilfreiche zusätzliche Ausgaben während des Übersetzens.

- `--parse-verbose`
  Erzeugt zusätzliche Ausgaben während die PVS-Dateien geparst werden.

- `--hdl-comment`
  Fügt in der erzeugten Verilogdatei Kommentare ein, die die Zuordnung von PVS2HDL generierten Wire-Namen zu den ursprünglichen Variablennamen in PVS erleichtern.

- `--parse-only`
  Wenn diese Option übergeben wird, wird kein Verilogcode generiert, sondern es wird nur getestet, ob die `Eingabedatei` von PVS2HDL fehlerfrei geparst

werden kann. Dies ist wegen den Einschränkungen des PVS-Parsers von PVS2HDL nützlich (siehe Abschnitt 2.2).

Anhang B

Konfigurationsdatei pvs2hdl.conf

Die Konfigurationsdatei von PVS2HDL ist in Sektionen aufgeteilt. Eine Sektion beginnt mit dem Schlüsselwort SECTION gefolgt vom Namen der Sektion. In der aktuellen Version von PVS2HDL gibt es zwei verschiedene Sektionen:

- SECTION connect_outside_functions
  In diesem Abschnitt der Konfigurationsdatei wird angegeben, welche Verilog-Module von PVS2HDL mit dem Toplevel-Modul verbunden werden sollen (siehe Abschnitt 2.3.1).

  An die Namen der PVS-Funktionen müssen noch ein x (siehe Abschnitt 2.2.5) sowie die konstanten Parameter der Funktionen angefügt werden.

  Als Beispiel folgt die connect_outside_functions Sektion der Konfigurationsdatei für den VAMP-Prozessor:

  ```
  SECTION connect_outside_functions
  ext_inputx
  ext_outputx
  bus_protocol_inputx_29_1_2_1_20_7_1_4_2_20_7_2_3
  bus_protocol_outputx_29_1_2_1_20_7_1_4_2_20_7_2_3
  ```

- SECTION predefined_functions
  In diesem Abschnitt wird angegeben, welche Funktionen nicht übersetzt sondern stattdessen durch vorhandene Verilog-Module ersetzt werden sollen. Dazu steht in jeder Zeile zunächst der Name der zu ersetzenden Funktion inklusive der konstanten Integer-Parameter. Dahinter folgt der Name des Verilog-Moduls, durch das der Funktionsaufruf ersetzt werden soll. Dieses Verilog-Modul muss die gleichen Parametertypen wie die zu ersetzende Funktion besitzen.

  Als Beispiel wieder einige Zeilen aus der Konfigurationsdatei für den VAMP-Prozessor:

  ```
  SECTION predefined_functions
  ram_next_conf_9_8 data_ram
  ram_next_conf_7_1 valid_ram
  ram_next_conf_7_20 tag_ram
  ram2p_next_conf_7_2 history_ram1
  ram2p_next_conf_7_8 history_ram2
  ```

Diese fünf Zeilen sorgen dafür, dass die verschiedenen RAMs im Cache des VAMP-Prozessors durch mit CoreGen generierte Verilog-Module realisiert werden.

Anhang C

Quelltexte

C.1 Quelltext des Speicher-Interfaces

```
`define state_wait                4'b0000
`define state_read                4'b0001
`define state_write               4'b0010
`define state_read_burst          4'b0011
`define state_burst_write2        4'b0100
`define state_finish_read         4'b0101
`define state_write2              4'b0110
`define state_burst_write1        4'b0111

`define LO                        1'b0
`define HI                        1'b1

module mp_mem_interface_64(sys_rst_l,
                           sys_clk,

                           // Verbindungen zum SD-RAM Controller
                           sdc_address,
                           sdc_in_data,
                           sdc_out_data,
                           sdc_rnw,
                           sdc_req,
                           sdc_ack,
                           sdc_data_valid,

                           // Verbindungen zum VAMP-Cache
                           MAd,
                           MDat_in,
                           MDat_out,
                           req,
                           reqp,
                           BRdy,
                           wr,
                           burst);
```

```
input           sys_clk;
input           sys_rst_l;

// Verbindungen zum SD-RAM Controller
output [21:0]   sdc_address;
output [63:0]   sdc_in_data;
input [63:0]    sdc_out_data;
output          sdc_rnw;
output          sdc_req;
input           sdc_ack;
input           sdc_data_valid;

// Verbindungen zum VAMP-Cache
input [31:0]    MAd;
input [63:0]    MDat_in;
output [63:0]   MDat_out;
input           req;
output          reqp;
output          BRdy;
input           wr;
input           burst;


// Außenregister

// Register zum SD-RAM Controller
reg [21:0]      sdc_address;
reg [63:0]      sdc_in_data;
reg             sdc_rnw;
reg             sdc_req;

// Register zum VAMP-Cache
reg             BRdy;
reg [63:0]      MDat_out;
reg             reqp;


// Interne Register

// Zustandsregister
reg [3:0]       next_state;

// Verzögerungsregister für die Daten
// zum Cache
reg [63:0]      MDat_out_delay;

// Register zum Zwischenspeichern der Daten
// während Burst-Schreib-Zugriffen
reg             still_reading;
reg [63:0]      reg_burst_write1;
reg [63:0]      reg_burst_write2;
reg [63:0]      reg_burst_write3;
reg             burst3written;
```

```
// burst-counter

// dieser counter zählt, wieviele Addressen
// an den IsyTec-Controller geschickt worden
// sind
reg [1:0]       burst_counter;

// dieser counter zählt, wieviele Daten vom
// IsyTec-Controller empfangen worden sind
reg [1:0]       burst_counter2;


// Zustandsautomat

always @ (posedge sys_clk)
  begin

    if (sys_rst_l==0)
      begin // Init-Zustand

        next_state <= #100 `state_wait;

        BRdy <= #100 `LO;
        MDat_out <= #100 64'bx;
        reqp <= #100 `LO;

        still_reading <= #100 `LO;

        sdc_req <= #100 `LO;
      end // if (sys_rst_l==0)
    else
      begin
        // Der Cache liefert die Daten erst einen
        // Takt nach Ready-Signal
        MDat_out <= #100 MDat_out_delay;

        case (next_state)
          `state_wait:
            begin
              burst_counter <= #100 2'b00;
              burst_counter2 <= #100 2'b00;
              BRdy <= #100 `LO;
              MDat_out_delay <= #100 64'bx;
              still_reading <= #100 `LO;
              sdc_req <= #100 `LO;

              if (req | reqp)
                begin
                  reqp <= #100 `HI;

                    if (burst)
```

```
              if (!wr)
                begin
                  sdc_address <= #100 MAd;
                  sdc_rnw <= #100 `HI;
                  sdc_req <= #100 `HI;

                  next_state <= #100 `state_read_burst;
                end
              else
                begin
                  BRdy <= #100 `HI;
                  still_reading <= #100 `HI;

                  next_state <= #100 `state_burst_write1;
                end
            else
              if (!wr)
                begin
                  // bei einem Lese-Zugriff können die
                  // Daten schon hier gesetzt werden
                  sdc_address <= #100 MAd;
                  sdc_rnw <= #100 `HI;
                  sdc_req <= #100 `HI;

                  next_state <= #100 `state_read;
                end
              else
                begin
                  // bei einem Schreib-Zugriff liegen
                  // die Daten erst im nächsten Takt
                  // an, deshalb wird jetzt noch kein
                  // Request generiert sondern erst
                  // im nächsten Takt
                  next_state <= #100 `state_write;
                end
              end
            end // case: `state_wait

  `state_write:
    begin
      sdc_address <= #100 MAd;
      sdc_in_data <= #100 MDat_in;
      sdc_rnw <= #100 `LO;
      sdc_req <= #100 `HI;

      if (sdc_ack)
        begin
          BRdy <= #100 `HI;
          reqp <= #100 `LO;

          next_state <= #100 `state_wait;
        end
    end // case: `state_write
```

```
`state_burst_write1:
  begin
    burst3written <= #100 `LO;

    next_state <= #100 `state_burst_write2;
  end

`state_burst_write2:
  begin
    if (still_reading)
      burst_counter <= #100 burst_counter + 1;

    // Die Daten vom Cache werden in den Registern
    // zwischengespeichert
    case (burst_counter)
      2'b00:
        begin
          sdc_in_data <= #100 MDat_in;
        end
      2'b01:
        begin
          reg_burst_write1 <= #100 MDat_in;

          reqp <= #100 `LO;
        end
      2'b10:
        begin
          reg_burst_write2 <= #100 MDat_in;

          still_reading <= #100 `LO;
          BRdy <= #100 `LO;
        end
      2'b11:
        begin
          if (!burst3written)
            reg_burst_write3 <= #100 MDat_in;
          burst3written <= #100 `HI;
        end
    endcase // case(burst_counter)

    // falls ein neuer Request von der CPU kommt,
    // bevor das SD-RAM fertig ist, wird er
    // zwischengespeichert und reqp auf HI gesetzt
    if (still_reading==`LO)
      begin
        if (req) reqp <= #100 `HI;
      end

    if (burst_counter==0)
      begin
        sdc_req <= #100 `HI;
        sdc_rnw <= #100 `LO;
```

```
            sdc_address <= #100 MAd;
          end
        else
          begin
            if (sdc_ack)
              begin
                burst_counter2 <= #100 burst_counter2 + 1;

                case (burst_counter2)
                  2'b00:
                    sdc_in_data <= #100 reg_burst_write1;
                  2'b01:
                    sdc_in_data <= #100 reg_burst_write2;
                  2'b10:
                    sdc_in_data <= #100 reg_burst_write3;
                endcase

                if (burst_counter2==3)
                  begin
                    sdc_req <= #100 `LO;

                    next_state <= #100 `state_wait;
                  end
                else
                  begin
                    sdc_address <= #100 sdc_address + 1;
                  end
              end
          end
      end // case: `state_burst_write2

    `state_read:
      begin
        if (sdc_ack)
          begin
            sdc_req <= #100 `LO;

            next_state <= #100 `state_finish_read;
          end
      end // case: `state_read

    `state_finish_read:
      begin
        if (sdc_data_valid)
          begin
            BRdy <= #100 `HI;
            reqp <= #100 `LO;

            MDat_out_delay <= #100 sdc_out_data;

            next_state <= #100 `state_wait;
          end
      end // case: `state_finish_read
```

```
          `state_read_burst:
            begin
              // Generierung der Addressen für den
              // SD-RAM Controller
              if (burst_counter==2'b11)
                begin
                  sdc_req <= #100 `LO;
                end
              else
                begin
                  if (sdc_ack)
                    begin
                      sdc_address <= #100 sdc_address + 1;
                      burst_counter <= #100 burst_counter + 1;
                    end
                end

              // Empfang der vom SD-RAM Controller
              // zurückgelieferten Daten, wenn das
              // Signal sdc_data_valid aktiv ist
              if (sdc_data_valid)
                begin
                  MDat_out_delay <= #100 sdc_out_data;
                  BRdy <= #100 `HI;
                  burst_counter2 <= #100 burst_counter2 + 1;

                  if (burst_counter2==2'b11)
                    begin
                      reqp <= #100 `LO;

                      next_state <= #100 `state_wait;
                    end
                end // if (sdc_data_valid)
              else
                begin
                  BRdy <= #100 `LO;
                  MDat_out_delay <= #100 64'bx;
                end // else: !if(sdc_data_valid)
            end
        endcase
      end // if (!sys_rst_l==0)
  end // always @ (posedge sys_clk)

endmodule
```

C.2 Quelltext des Host-Interfaces

```
`define state_wait            2'b00
`define state_write           2'b01
`define state_read            2'b10
`define state_interrupt       2'b11

`define LO 1'b0
`define HI 1'b1

module host_interface(clk,
                      slow_clk,
                      reset,
                      capim1_do,
                      capim2_do,
                      capim3_do,
                      capim4_do,
                      capim1_wr,
                      capim2_wr,
                      capim3_wr,
                      capim4_wr,
                      capim1_inta,
                      capim2_inta,
                      capim3_inta,
                      capim4_inta,
                      capim1_di,
                      capim2_di,
                      capim3_di,
                      capim4_di,
                      capim1_intr,
                      capim2_intr,
                      capim3_intr,
                      capim4_intr,
                      cache_address,
                      cache_dout,
                      cache_mw,
                      cache_mr,
                      cache_mbw,
                      cache_din,
                      cache_busy,
                      cpu_reset);

   input           clk;
   input           slow_clk;
   input           reset;

   // 33MHz Signale
   input [31:0]    capim1_do;
   input [31:0]    capim2_do;
   input [31:0]    capim3_do;
   input [31:0]    capim4_do;
   input           capim1_wr;
   input           capim2_wr;
```

```
input           capim3_wr;
input           capim4_wr;
input           capim1_inta;
input           capim2_inta;
input           capim3_inta;
input           capim4_inta;
output [31:0]   capim1_di;
output [31:0]   capim2_di;
output [31:0]   capim3_di;
output [31:0]   capim4_di;
output          capim1_intr;
output          capim2_intr;
output          capim3_intr;
output          capim4_intr;

// 8,25 MHz Signale
output [28:0]   cache_address;
output [63:0]   cache_dout;
output          cache_mw;
output          cache_mr;
output [7:0]    cache_mbw;
input [63:0]    cache_din;
input           cache_busy;

// CPU_Reset wird mit 33MHz getaktet. Bei
// diesem Signal gibt es keine Nachteile, wenn es
// asynchron zum langsamen Takt geändert wird
output          cpu_reset;

// die Input-Register zwischen Capims und VAMP
// schnell getaktet
reg [31:0]      input_reg1;
reg [31:0]      input_reg2;
reg [31:0]      input_reg3;
reg [31:0]      input_reg4;

// die Output-Register zwischen VAMP und Capims
// langsam getaktet
reg [31:0]      capim1_di;
reg [31:0]      capim2_di;
reg [31:0]      capim3_di;
reg [31:0]      capim4_di;

// die intr-Signale werden schnell getaktet
reg             capim1_intr;
reg             capim2_intr;
reg             capim3_intr;
reg             capim4_intr;

// die Register zwischen Host-Interface
// und Cache
// langsam getaktet
reg [28:0]      cache_address;
```

```
reg [63:0]      cache_dout;
reg             cache_mw;
reg             cache_mr;
reg [7:0]       cache_mbw;


// Berechnung von CPU-Reset
assign          cpu_reset=input_reg4[31];

// das State-Register
reg [1:0]       next_state;

// Register zur Synchronisation zwischen dem
// wr-Signal von Capim 3 und dem langsam getakteten
// Teil des Host-Interface
reg             memory_req_slow;
reg             memory_reqp_slow;

// Register zur Sychnrosisation des intr-Signals
// von Capim 3
reg             generate_intr;


// Die Ausgänge der Capims werden durch Register geführt.
// Diese Register haben als Clock-Enable-Signal das
// wr-Signal des jeweiligen Capims
// schnell getaktet
always @ (posedge clk)
  begin
     // capim1-3 regeln Speicherzugriffe
     // capimx_wr dient als Clock-Enable Signal
     // für Capim x
     if (capim1_wr) input_reg1 <= #100 capim1_do;
     if (capim2_wr) input_reg2 <= #100 capim2_do;
     if (capim3_wr) input_reg3 <= #100 capim3_do;

     // capim4 regelt den CPU-Reset
     if (capim4_wr) input_reg4 <= #100 capim4_do;

     // Berechnung von memory_req_slow
     memory_req_slow <= #100 (memory_req_slow | capim3_wr)
                           & ~memory_reqp_slow;
  end

// der folgende Automat regelt die Kommunikation mit dem
// Host-PC über die Capims
// langsam getaktet
always @ (posedge slow_clk)
  begin
    if (reset)
      begin
        // Reset
        next_state <= #100 `state_wait;
```

```
      cache_mw <= #100 `LO;
      cache_mr <= #100 `LO;

      memory_reqp_slow <= #100 `LO;
    end
  else
    begin
      case (next_state)
        `state_wait:
          begin
            cache_mr <= #100 `LO;
            cache_mw <= #100 `LO;
            generate_intr <= #100 `LO;

            if (memory_req_slow)
              begin
                memory_reqp_slow <= `HI;

                // Die Cache-Adresse ist 29 Bit breit,
                // deshalb werden vor die 22 Bit der
                // Adresse aus input_reg3[31:10] noch
                // sieben Nullen eingefügt
                cache_address <= #100 {7'b0000000,
                                       input_reg3[31:10]};

                if (input_reg3[9])
                  begin
                    // Schreibzugriff
                    cache_dout <= #100 {input_reg1,input_reg2};
                    cache_mw <= #100 `HI;
                    cache_mbw <= #100 input_reg3[8:1];

                    next_state <= #100 `state_write;
                  end
                else
                  begin
                    // Lesezugriff
                    cache_mr <= #100 `HI;

                    next_state <= #100 `state_read;
                  end
              end
          end // case: `state_wait

        `state_write:
          begin
            memory_reqp_slow <= `LO;
            if (cache_busy==`LO)
              begin
                cache_mw <= #100 `LO;

                next_state <= #100 `state_interrupt;
```

```
                end
              end // case: `state_write

            `state_read:
              begin
                memory_reqp_slow <= `LO;
                if (cache_busy==`LO)
                  begin
                    cache_mr <= #100 `LO;

                    capim1_di   <= #100 cache_din[63:32];
                    capim2_di   <= #100 cache_din[31:0];

                    next_state <= #100 `state_interrupt;
                  end
              end // case: `state_read

            `state_interrupt:
              begin
                generate_intr <= #100 `HI;
                next_state <= #100 `state_wait;
              end

          endcase // case(next_state)
      end // else: !if(reset)
    end // always @ (posedge slowclk)

  // Steuerung der Interrupt-Signale
  always @ (posedge clk)
    begin
      if (generate_intr & ~capim3_intr)
        begin
          capim3_intr <= #100 `HI;
        end
      if (capim3_intr & capim3_inta)
        begin
          capim3_intr <= #100 `LO;
        end
    end

endmodule // host_interface
```

C.3 Reset-Logik

```
module reset_logic(clk,
                   sdc_reset,
                   cache_reset);
   input      clk;
   output     sdc_reset;
   output     cache_reset;


   // Zähler für das Reset-Signal
   // des IsyTec SD-RAM-Controllers
   reg [2:0]  clr_cnt1;

   always @ (posedge clk)
     begin
       if (clr_cnt1[2]==0)
         clr_cnt1 <= clr_cnt1 + 1;
     end
   // das Signal sdc_reset ist so
   // lange aktiv, bis das Bit 2
   // des Zählers 1 wird
   assign sdc_reset = !clr_cnt1[2];


   // Zähler für das Reset-Signal
   // des Caches
   reg [10:0] clr_cnt2;

   always @ (posedge clk)
     begin
       if (clr_cnt2[10]==0)
         clr_cnt2 <= clr_cnt2 + 1;
     end
   // das Signal cache_reset ist so
   // lange aktiv, bis das Bit 10
   // des Zähler 1 wird
   assign cache_reset = !clr_cnt2[10];

endmodule // reset_logic
```

Literaturverzeichnis

[Ash96] Peter J. Ashenden. *The Designer's Guide to VHDL.* Morgan Kaufmann, 1996.

[BBD+96] D. Borrione, H. Bouamama, D. Deharbe, C. le Faou, and A. Wahba. HDL-based integration of formal methods and CAD tools in the PREVAIL environment. In *FMCAD'96*, volume 1166 of *LNCS*. Springer, 1996.

[Ber01] Christoph Berg. Formal verification of an IEEE floating point adder. Diplomarbeit, Universität des Saarlandes, Saarbrücken, 2001.

[Bey02] Sven Beyer. *Formal Verification of a Cache and a Memory Interface*. PhD thesis, Universität des Saarlandes, 2002. unfinished.

[BJK01] Christoph Berg, Christian Jacobi, and Daniel Kroening. Formal verification of a basic circuits library. In *Proc. 19th IASTED International Conference on Applied Informatics, Innsbruck (AI'2001)*, pages 252–255. ACTA Press, 2001.

[BJKL02] Sven Beyer, Christian Jacobi, Daniel Kroening, and Dirk Leinenbach. Correct hardware by synthesis from PVS. Technical report, Saarland University, 2002.

[BMSG96] Ricky Butler, Paul Miner, Mandayam Srivas, and Dave Greve. A bitvectors library for PVS. Technical Report TM-110274, NASA Langley Research Center, 1996.

[BPS92] Dominique Borrione, Laurence Pierre, and Ashraf Salem. Formal verification of VHDL descriptions in the PREVAIL environment. *IEEE Design&Test Magazine*, 9(2), June 1992.

[Cil99] Michael D. Ciletti. *Modeling, synthesis, and rapid prototyping with the Verilog HDL.* Prentice-Hall, 1999.

[Fuj97] Fujitsu, Inc. *Data Sheet* `4x1Mx16 Bit` *Synchronous dynamic RAM* `MB811641642A-100`, 1997.

[HD85] F. Hanna and N. Daeche. Specification and verification using higherorder logic. In *Proceedings of the 7th International Conference on Computer Hardware Design Languages*, pages 418–433, 1985.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[IEE85] IEEE - Institute of Electrical and Electronics Engineers. *ANSI/IEEE standard 754–1985, IEEE Standard for Binary Floating-Point Arithmetic*, 1985.

[IEE94] IEEE. *IEEE standard VHDL language reference manual : IEEE standard 1076-1993*. IEEE, 1994.

[IEE95] IEEE - Institute of Electrical and Electronics Engineers. *IEEE 1364-1995 Verilog Language Reference Manual*, 1995.

[ISY01a] ISYTEC - Integrierte Systemtechnik GmbH, Friedrich-Naumann-Str. 8, 09131 Chemnitz. *CHIPit Power Edition: FPGA-based rapid-prototyping PCI board - EBPCI*, 2001.

[ISY01b] ISYTEC - Integrierte Systemtechnik GmbH, Friedrich-Naumann-Str. 8, 09131 Chemnitz. UMR*Bus Communication System*, 2001.

[Jac02] Christian Jacobi. *Formal Verification of a fully IEEE compliant Floating Point Unit*. PhD thesis, University of Saarland, Computer Science Department, 2002.

[KB95] Matt Kaufmann and R. S. Boyer. The boyer-moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*, 29(2):27–62, 1995.

[KH92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[KM97] Matt Kaufmann and J. Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.

[Kro01] Daniel Kroening. *Formal Verification of Pipelined Microprocessors*. Dissertation, Universität des Saarlandes, Saarbrücken, 2001.

[Lei01] Dirk Leinenbach. Ein Bison basierter Parser für PVS. FoPra-Ausarbeitung, April 2001.

[Mey02] Carsten Meyer. Entwicklung einer Laufzeitumgebung für den VAMP-Prozessor. Master's thesis, Saarland University, 2002.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38, April 1965. `www.intel.com/research/silicon/moorespaper.pdf`.

[MP00] Silvia M. Mueller and Wolfgang J. Paul. *Computer Architecture. Complexity and Correctness*. Springer, 2000.

[OSR92] S. Owre, N. Shankar, and J. M. Rushby. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer, 1992.

[Rus00] David Russinoff. Mechanical verification of register-transfer logic: A floating-point multiplier. In M. Kaufmann, P. Manolios, and J Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer, 2000.

[SMMB89] S. Finn, M.P. Fourman, M.D. Francis, and B. Harris. Formal system design - interactive synthesis based on computer assisted formal reasoning. In *Intern. Workshop on Applied Formal Methods for Correct VLSI Design*, 1989.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In *IBM Journal of Research and Development*, volume 11 (1), pages 25–33. IBM, 1967.

[Xil02] Xilinx, Inc. *Virtex-E 1.8V Extended Memory Field Programmable Gate Arrays Data Sheet*, 2002. `www.xilinx.com/partinfo/ds025.htm`.