Entwurf einer Schnittstelle zwischen SB-PRAM und PC auf PCI-Basis

Diplom-Arbeit

Sven Beyer
Universität des Saarlandes
email: sbeyer@wjpserver.cs.uni-sb.de

3. November 2000

Ich versichere, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Saarbrücken, den

The 9000 series is the most reliable computer ever made. No 9000 computer has ever made a mistake or distorted information. We are all, by any practical definition of the words, foolproof and incapable of error.

HAL 9000 in Stanley Kubrick's Movie "2001–A Space Odyssey"

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Einleitung

In den letzten 20 Jahren ist die Leistungsfähigkeit moderner Mikroprozessoren stark angestiegen. Während vor 20 Jahren noch Prozessoren mit einer Taktfrequenz von 1 MHz modern waren, werden heute bereits die erste Prozessoren mit 1 GHz designt. Das entspricht einer Verbesserung um den Faktor 1000 in nur 20 Jahren. Diese Entwicklung kann aber nach dem heutigen Stand des Wissens nicht beliebig fortgeführt werden. Die Physik setzt der fortgesetzten Miniaturisierung von Mikroprozessoren und der damit einhergehenden Verschnellerung Grenzen. Diese Grenzen sind heute noch nicht erreicht, doch eine Entwicklung wie in den letzten 20 Jahren kann nicht mehr lange anhalten. Paul zeigt in [Pa78], daß „unter ganz einfachen Annahmen über Schaltelemente ... jeder Schaltvorgang mindestens $5.6 \cdot 10^{-33}$ Sekunden dauert." Damit ergibt sich also für Schaltelemente eine maximale theoretische Taktfrequenz von $1.8 \cdot 10^{20}$ GHz. Eine einfache Berechnung zeigt, daß diese Grenze bei einer Entwicklung wie in den letzten 20 Jahren vor dem Jahr 2140 erreicht wäre. Es ist zu vermuten, daß man in der Realität bereits früher an eine obere Grenze mit deutlich niedrigerer Taktfrequenz stößt.

Deshalb setzt man heute mehr denn je nicht mehr unbedingt auf Prozessoren mit kürzeren Zykluszeiten, sondern auf Architekturen, in denen möglichst viele Befehle in einem einzelnen Zyklus ausgeführt werden können. Aus diesem Grund wird in den letzten Jahren verstärkt im Bereich von *Parallelrechnern* geforscht, also Rechnern, in denen *mehrere* Prozessoren auf einen Speicher zugreifen.

Ein naheliegendes Modell für solche Parallelrechner ist das PRAM-Modell der theoretischen Informatik (*Parallel Random Access Machine* [AKP91a, AKP91b]). In einer *shared-memory* PRAM ist eine Menge von n Prozessoren über ein Netzwerk mit einem gemeinsamen Speicher verbunden. Bei Problemen, die sich effizient auf die n Prozessoren verteilen lassen, sollte sich dann ein Geschwindigkeitszuwachs gegenüber der Maschine mit nur einem Prozessor von nahezu n ergeben.

An der Universität des Saarlandes wurde am Lehrstuhl von Professor Paul im Fachbereich Informatik ein shared-memory Parallelrechner nach dem PRAM-Modell entwickelt, die SB-PRAM (*Saarbrücken PRAM* [ADK93, BBFFGL97, Gö96, Wa97]). Dabei wurden Versionen mit 4, 16 und 64 physikalischen Prozessoren gefertigt. Gemessen an der verwendeten Technologie bietet diese SB-PRAM eine relativ große Rechenleistung. Ein Interface zur graphischen Darstellung berechneter Daten mit einem großen Datendurchsatz fehlt aber. Mit einem solchen Interface könnten Berechnungen oder Animationen beispielsweise in Echtzeit dargestellt werden.

Im Rahmen dieser Arbeit wurde eine PCI-Karte (*Peripheral Component Interconnect*

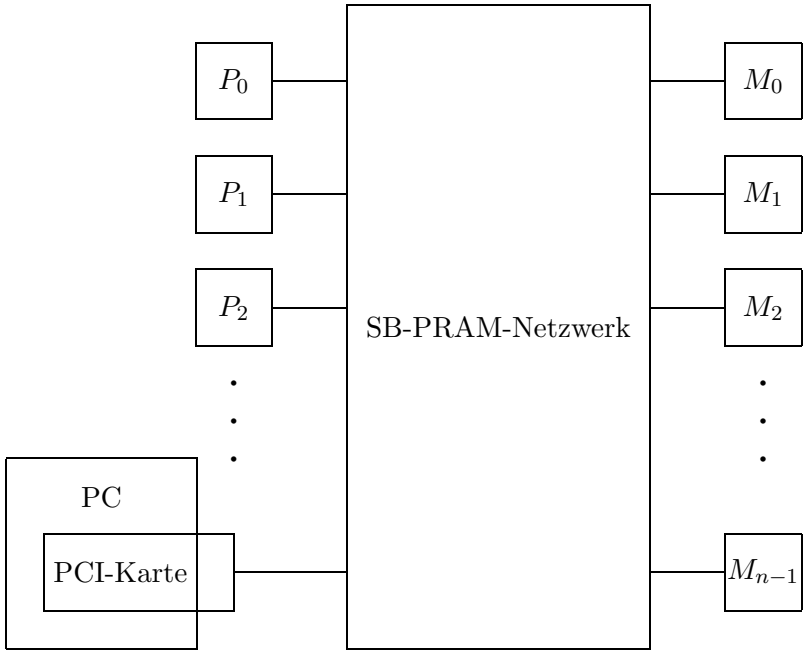


Abbildung 1: Anschluß der PCI-Karte an das PRAM-Netzwerk

[PCI95]) als Interface zur SB-PRAM designt, um beispielsweise PCs an das Netzwerk der SB-PRAM anzuschließen. Der PC übernimmt dabei die Funktion eines PRAM-Prozessors, das heißt, er greift über das PRAM-Netzwerk auf den Speicher zu. Abbildung 1 veranschaulicht dieses Szenario. $P_0, P_1, \ldots$ sind PRAM-Prozessoren; der gemeinsame Speicher ist auf die n Speichermodule M_0 bis M_{n-1} verteilt.

Damit ist der Datendurchsatz zwischen PCI-Karte und SB-PRAM durch den Datendurchsatz zwischen einem PRAM-Prozessor und dem PRAM-Netzwerk begrenzt. Zur Zeit ist dieser Datendurchsatz 27 MByte/s vom Netzwerk zum PRAM-Prozessor. Zwischen PC und PCI-Karte ist auf dem PCI-Bus ein Datendurchsatz von 132 MByte/s möglich [PCI95]; damit sollte ein Datentransfer zwischen PRAM-Speicher und PC-Speicher mit dem maximalen Datendurchsatz 27 MBytes/s möglich sein. Falls dieser Datendurchsatz für eine Anwendung nicht ausreichend ist, bietet sich immer noch die Möglichkeit, *zwei* solche PCI-Karten in einem PC zu betreiben. Dadurch verdoppelt sich natürlich der zu erwartende Datendurchsatz. Für die Anwendung der PCI-Karte als *Grafik-Karte* der SB-PRAM sollte der Datendurchsatz einer einzelnen Karte allerdings ausreichend sein.

Die PCI-Karte enthält einen lokalen Speicher; alle Datentransfers der PCI-Karte erfolgen zwischen dem lokalen RAM der PCI-Karte und dem Speicher der SB-PRAM. Der PC kann seinerseits das lokale RAM der Karte lesen oder schreiben. Dadurch ist der Datentransfer zwischen SB-PRAM und PCI-Karte von der momentanen Auslastung des PCI-Busses unabhängig. Schließlich sind moderne Grafik- und Soundkarten eines PCs ebenfalls an den PCI-Bus angeschlossen und besetzen diesen Bus ihrerseits für Datentransfers.

Für Zugriffe über die PCI-Karte auf den Speicher der SB-PRAM stellt die PCI-Karte 16 sogenannte *DMA-Kanäle* (*Direct Memory Access*) zur Verfügung. Auf diesen 16 DMA-Kanälen führt die PCI-Karte jeweils sogenannte *DMA-Programme* aus. In einem solchen DMA-Programm wird der Zugriff spezifiziert, also die Start- und Endadresse eines Speicherblocks im lokalen RAM der Karte, die Startadresse im Speicher der SB-PRAM und die Zugriffsart. Diese DMA-Programme werden auf der PCI-Karte selbst von einem *DMA-Prozessor* ausgeführt; der Prozessor des PCs wird durch ihre Ausführung nicht belastet.

Um einen Schreib- oder Lesezugriff auf den Speicher der SB-PRAM durchzuführen, reserviert sich der PC einen der DMA-Kanäle. Für einen typischen Schreibzugriff transferiert er dann die Daten aus seinem Hauptspeicher ins lokale RAM der Karte. Danach startet der PC lediglich noch für diesen Kanal ein DMA-Programm, in dem Adressen und Zugriffsart spezifiziert werden. Da keine Interrupts implementiert sind, fragt der PC den Status des DMA-Kanals ab, bis das Programm fertig ist. Bei dieser Abfrage kann der PC auch feststellen, ob bei der Ausführung des DMA-Programms ein Fehler aufgetreten ist. Eine Lese-Operation wird durch ein DMA-Programm gestartet. Wenn das Programm fertig ist, stehen die gewünschten Daten im lokalen RAM der Karte und können von dort vom PC weiterverarbeitet werden.

Neben dem hohen Datendurchsatz beim *parallelen* Ausführen von DMA-Programmen war eine weitere Anforderung an das Design eine minimale Latenz für spezielle Zugriffe auf den Speicher der SB-PRAM. Dazu unterstützt die PCI-Karte besondere DMA-Programme, sogenannte Einzelanweisungen, die sich nur auf *eine* Adresse beziehen. Diese Einzelanweisungen werden von der PCI-Karte beim Ausführen priorisiert.

Gliederung der Arbeit

In Kapitel 1 werden einige Grundlagen für das Design der PCI-Karte gelegt. Es erfolgt zunächst eine Einführung in die Funktionsweise der SB-PRAM. Anschließend wird eine Semantik für *DMA-Programme* spezifiziert und ein Algorithmus zur gleichzeitigen Ausführung mehrerer DMA-Programme erarbeitet. Dieser Algorithmus wird dann später in Hardware umgesetzt. Zusätzlich wird der Aufbau der PCI-Karte beschrieben, und einige spezielle Eigenschaften der verwendeten FPGA-Architektur werden angegeben (*Field Programmable Gate Array* [FPGA99]).

Im 2. Kapitel wird das Design des *DMA-Prozessors* angegeben. Dazu wird ein Instruktionssatz für den DMA-Prozessor spezifiziert, und es wird bewiesen, daß das vorgestellte Design diesen Instruktionssatz korrekt interpretiert. Zusätzlich wird gezeigt, daß das Design eine Implementierung des Algorithmus zur Ausführung von DMA-Programmen aus Kapitel 1.2.3 ist.

In Kapitel 3 wird eine Sortiereinheit für den DMA-Prozessor in Anlehnung an [Gö96] entworfen, da der DMA-Prozessor wie die PRAM-Prozessoren eine solche Sortiereinheit benötigt (siehe Kapitel 1.1). Es wird insbesondere auf die Unterschiede zur Sortiereinheit eines PRAM-Prozessors eingegangen.

In Kapitel 4 wird die Treibersoftware für den DMA-Prozessor vorgestellt, und in Ka-

pitel 5 wird gezeigt, wie das Design des DMA-Prozessors verifiziert werden kann. Im abschließenden Ausblick fassen wir die Ergebnisse der Arbeit nochmals zusammen und zeigen weitere Einsatzmöglichkeiten für die entworfenen PCI-Karte auf.

Danksagung

Mein besonderer Dank gilt Herrn Professor Wolfgang J. Paul für die Vergabe des interessanten Themas und die Betreuung.

Des weiteren danke ich Thomas Grün für die Betreuung dieser Arbeit. Mit Anregungen und einem offenen Ohr für meine Probleme standen mir am Lehrstuhl besonders Michael Bosch, Peter Bach, Jörg Fischer und Cederic Lichtenau zur Seite. Ein besonderer Dank gebührt auch ATMedia GmbH für das Bereitstellen einer PCI-Karte zum Test meiner Schaltungen, insbesondere Michael Braun, Jörg Friedrich, Stefan Janocha und Diethelm Schlegel für zahlreiche Anregungen.

Kapitel 1

Grundlagen

In diesem Kapitel erarbeiten wir einige Grundlagen für die weitere Arbeit. Zunächst spezifizieren wir dazu die Funktionsweise der SB-PRAM und leiten daraus einige Anforderungen an das Design der PCI-Karte und des DMA-Prozessors ab. Anschließend erarbeiten wir einen Algorithmus zum gepipelineten, parallelen Ausführen mehrerer DMA-Programme, der im 2. Kapitel in Hardware umgesetzt wird. Dazu spezifizieren wir zunächst eine Semantik von DMA-Programmen und untersuchen, inwieweit der Algorithmus diese Semantik erfüllt. Danach wenden wir uns dem Design der PCI-Karte zu. Zuletzt gehen wir noch auf einige Besonderheiten der beim Design verwendeten FPGA-Technologie ein, die bei der Betrachtung von Kosten und Delays von Schaltkreisen relevant sind.

Es folgen zunächst noch zwei grundlegende Definitionen, die in der Arbeit oft gebraucht werden.

Definition 1.0.1 *Sei* $a = a[n-1:0] = (a_{n-1} \dots a_0) \in \{0,1\}^n$. *Dann heißt*

$$\langle a \rangle := \sum_{i=0}^{n-1} a_i \cdot 2^i$$

die durch a *dargestellte* ***Binärzahl****.*

Für $n \in \mathbb{N}$ *sei* $B_n := \{\langle a \rangle | a \in \{0,1\}^n\}$ *die Menge der mit* n *Bits darstellbaren Binärzahlen. Die Abbildung* $bin_n : B_n \to \{0,1\}^n$, $bin_n(c) := a[n-1:0]$ *mit* $\langle a \rangle = c$ *ordnet einer Zahl* c *ihre eindeutige* n*-stellige* ***Binärdarstellung*** *zu.*

Definition 1.0.2 *Ein* ***gerichteter Graph*** *ist ein Paar* $G = (V, E)$. *Hierbei ist* V *eine Menge und* $E \subseteq V \times V$. *Ein Element* $v \in V$ *heißt* ***Knoten*** *des Graphen, ein Element* $e = (u, v) \in E$ *heißt* ***Kante*** *von* u *nach* v.

In einem Graphen $G = (V, E)$ *ist ein* ***Pfad*** *der Länge* k *von einem Knoten* $v_1 \in V$ *zu einem Knoten* $w_k \in V$ *eine Folge von Kanten* $e_i = (v_i, w_i) \in E$, $i = 1, \dots, k$, *mit* $v_{i+1} = w_i$ *für* $i = 1, \dots, k-1$.

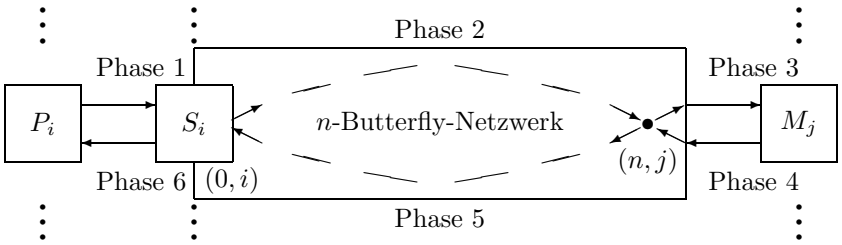


Abbildung 1.1: Zugriff von PRAM-Prozessor P_i auf Speichermodul M_j

1.1 SB-PRAM

Dieser Abschnitt führt in die Funktionsweise der SB-PRAM ein. Dazu wird zunächst der Routing-Algorithmus des SB-PRAM-Netzwerks erläutert; anschließend wird auf Hashing und virtuelle Prozessoren eingegangen. Außerdem werden noch PRAM-spezifische Speicherzugriffe und Multiplexing im Netzwerk erläutert. Abschließend werden aus den vorgestellten Eigenschaften der SB-PRAM einige Anforderungen an das Design des DMA-Prozessors abgeleitet.

Definition 1.1.1 *Unter einem* ***n-Butterfly-Netzwerk*** *versteht man einen gerichteten Graphen $G = (V, E)$ mit Knotenmenge $V = \{0, \ldots, n\} \times \{0, \ldots, 2^n - 1\}$. Dabei sagt man, ein Knoten $v = (s, z)$ steht in* Spalte s *und* Zeile z*. Die Kantenmenge ist gegeben durch $E = \{((k, j), (k + 1, j)), ((k, j), (k + 1, j \otimes 2^k)) | 0 \leq k < n, 0 \leq j < 2^n\}$. Bei der Verknüpfung $\otimes$ handelt sich dabei um eine bitweise EXOR-Verknüpfung der Binärdarstellungen beider Operanden. Die Knoten $(0, j), \ldots, (n, j)$ bezeichnet man als* Zeile j*, die Knoten $(k, 0), \ldots (k, 2^n - 1)$ als* Spalte k *des Butterfly-Netzwerks.*

Im Netzwerk der SB-PRAM greifen $N = 2^n$ physikalische Prozessoren $P_i, 0 \leq i < N$, über ein Butterfly-Netzwerk auf einen gemeinsamen 32-Bit-Speicher zu. Der Speicher ist dabei auf N Speichermodule $M_i, 0 \leq i < N$, gleicher Größe verteilt. Ein Speicherzugriff eines Prozessors P_i auf eine Adresse $\langle A[31:0] \rangle, A \in \{0, 1\}^{32}$ gliedert sich gemäß Abbildung 1.1 in 6 Phasen. Dabei nehmen wir an, daß der Speicherzugriff auf die Adresse $\langle A[31:0] \rangle$ vom Netzwerk zu einem Speichermodul M_j geroutet wird.

Phase 1: Die Speicheranfragen des Prozessors P_i werden nach aufsteigenden Adressen sortiert und in das Butterfly-Netzwerk eingespeist. Diese Aufgabe übernimmt je ein *Sortierknoten* S_i, den man als Knoten $(0, i)$ des Butterfly-Netzwerks betrachten kann. Jedem Prozessor P_i ist also ein Sortierknoten S_i zugeordnet. In [Gö96] entwirft Göler diesen Sortierknoten, und in [Sch95] entwirft Scheerer den Prozessor der SB-PRAM.

Phase 2: In einem n-Butterfly-Netzwerk gibt es genau einen Pfad von einem Knoten der Spalte 0 zu einem Knoten der Spalte n. Die Speicheranfrage wird im Butterfly-Netzwerk auf diesem eindeutigen Pfad vom Knoten $(0, i)$ zum Knoten (n, j) geschickt. Dabei wird in jedem Netzwerkknoten anhand eines Bits in der Adresse

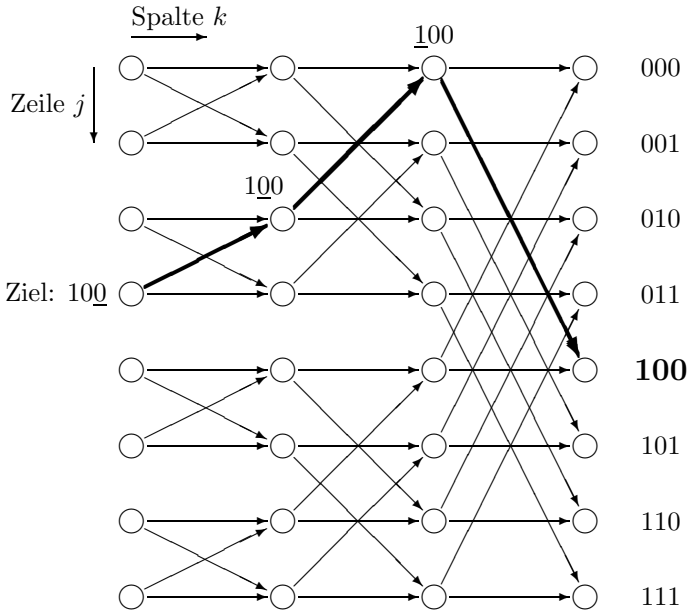


Abbildung 1.2: Routing auf einem 3-Butterfly-Netzwerk

entschieden, an welchen Knoten der folgenden Spalte die Speicheranfrage gesendet wird. Dieses Bit wird als *Routing-Bit* der Netzwerkstufe bezeichnet. Ein Netzwerkknoten routet die Speicheranfrage nach „oben“, wenn das Routing-Bit den Wert 0 hat, sonst nach „unten“.

Jede Spalte des Netzwerks wählt ihr Routing-Bit aus den oberen 16 Bit der Adresse $A[31:16]$, und die Routing-Bits der einzelnen Spalten sind paarweise verschieden. Insgesamt gibt es also eine injektive Abbildung $b : \{0, \ldots, n-1\} \to \{16, \ldots, 31\}$, die in Abhängigkeit von der Spalte k die Nummer des Routing-Bits $b(k)$ angibt. Abbildung 1.2 verdeutlicht das Routing in einem 3-Butterfly-Netzwerk; in diesem Beispiel gilt der Einfachheit halber $b(k) = k$.

Insgesamt wird damit die Speicheranfrage eines Prozessors P_i auf eine Adresse $\langle A[31:0]\rangle$ mit $A \in \{0,1\}^{32}$ von einem Netzwerkknoten $(0, i)$ zum Netzwerkknoten $(n, \langle A[b(n-1)]A[b(n-2)]\ldots A[b(0)]\rangle)$ geroutet. Der Zielknoten ist insbesondere unabhängig von der Nummer i des Prozessors. Der Netzwerkknoten der SB-PRAM wird in [Wa97] von Walle entworfen.

Phase 3: Der Knoten (n, j) sendet die Speicheranfrage an das RAM-Modul M_j, und die Speicheranfrage verläßt das Butterfly-Netzwerk. Das RAM-Modul M_j führt den Speicherzugriff der Speicheranfrage aus.

Phase 4: Das Speichermodul M_j gibt gegebenenfalls eine Antwort auf die Speicheranfrage. Der Knoten (n, j) nimmt diese Antwort wieder auf.

Phase 5: Anschließend durchläuft die Antwort im Netzwerk den gleichen Pfad wie auf dem Hinweg in umgekehrter Richtung. Die Antwort wird also vom Knoten (n, j) zum

Operator	Bedeutung
and	bitweise UND-Verknüpfung der Binärdarstellung
or	bitweise ODER-Verknüpfung der Binärdarstellung
max	Maximumbildung
add	Addition modulo 2^{32}

Tabelle 1.1: Operatoren für Multipräfix-Berechnungen

Knoten $(0, i)$ zurückgeschickt. Dabei ist der Knoten $(0, i)$ wieder der Sortierknoten S_i.

Phase 6: Der Sortierknoten S_i macht die Sortierung aus Phase 1 rückgängig und schickt die Antworten anschließend zum Prozessor P_i zurück. Damit treffen beim Prozessor P_i die Antworten genau in der Reihenfolge ein, in der P_i die zugehörigen Speicheranfragen verschickt hat.

Bei der SB-PRAM wird Hashing eingesetzt, um die logischen Adressen auf das physikalische RAM zu verteilen. Dabei wird eine *lineare Hashfunktion* der Form $g : B_{32} \to B_{32}$, $g(x) = hc \odot x$ mit einem beliebigen ungeraden $hc \in \mathbb{N}$ eingesetzt. Der Operator $\odot$ steht dabei für die Multiplikation modulo 2^{32}, also $hc \odot x = hc \cdot x \bmod 2^{32}$. Der Parameter hc wird als *Hashkonstante* bezeichnet. Die PRAM-Adressen von Speicheranfragen werden im DMA-Prozessor gehasht, bevor sie an die Sortiereinheit geschickt werden. Deshalb beziehen sich alle obigen Aussagen über die Adresse einer Speicheranfrage auf die *gehashte* Adresse.

Die SB-PRAM unterstützt gleichzeitige Lese- oder Schreibzugriffe verschiedener Prozessoren auf ein und dieselbe Speicherzelle. Bei gleichzeitigem Lesen einer Adresse erhalten alle beteiligten Prozessoren das gleiche Datum zurück; bei gleichzeitigem Schreiben setzt sich der Prozessor P_i mit dem größten Index i durch. Bei sogenannten Multipräfix- und Sync-Anweisungen werden die Daten der beteiligten Prozessoren mittels eines assoziativen und kommutativen Operators verknüpft abgespeichert (siehe nachfolgende Definition). Die SB-PRAM unterstützt für diese Anweisungen die Operatoren aus Tabelle 1.1.

Definition 1.1.2 *Sei $\circ$ ein assoziativer und binärer Operator, sei A eine Adresse mit Inhalt $M(A) = D_A$ und sei weiter $P_A = \{P_{i_1}, \ldots, P_{i_j}\}$ eine Menge von Prozessoren mit $i_1 < i_2 < \ldots < i_j$. Führen genau die Prozessoren aus P_A in einem Schritt die Operation **Multipräfix** $MP(A, \circ, D_{i_k})$ mit den Daten $D_{i_1}, \ldots D_{i_j}$ aus, dann ist am Ende des Schritts $M(A) = D_A \circ D_{i_1} \circ \ldots \circ D_{i_j}$ und jeder Prozessor $P_{i_k} \in P_A$ erhält als Antwort das Datum $D_A \circ D_{i_1} \circ \ldots \circ D_{i_{k-1}}$ zurück.*

*Seien $\circ, A, D_A, P_A$ wie oben. Führen genau die Prozessoren aus P_A in einem Schritt die Operation **Sync** $SYNC(A, \circ, D_{i_k})$ mit den Daten $D_{i_1}, \ldots D_{i_j}$ aus, dann ist am Ende des Schritts der Inhalt $M(A) = D_A \circ D_{i_1} \circ \ldots \circ D_{i_j}$ wie bei Multipräfix; die Prozessoren aus P_A erhalten kein Datum zurück.*

Die einzelnen physikalischen Prozessoren der SB-PRAM simulieren jeweils v virtuelle Prozessoren mit $v \in \{8, 16, 24, 32\}$. Die einzelnen virtuellen Prozessoren $VP_0, \ldots, VP_{v-1}$

von P_i werden nacheinander simuliert und können dabei jeweils eine Speicheranfrage an die SB-PRAM senden. Die so entstehenden maximal v Speicheranfragen eines physikalischen Prozessors P_i werden als *Netzwerkrunde* bezeichnet.

In einem Sortierknoten S_i werden jeweils bis zu 8 Speicheranfragen einer Netzwerkrunde zu einer sogenannten *Teilrunde* zusammengefaßt. Eine Netzwerkrunde besteht dann aus 1, 2, 3 oder 4 Teilrunden, je nach Zahl v der virtuellen Prozessoren pro physikalischem Prozessor. Die Speicheranfragen einer Teilrunde werden im Sortierknoten nach aufsteigenden Adressen sortiert. Anschließend werden die sortierten Anfragen an das PRAM-Netzwerk versandt. Speicheranfragen mit derselben Adresse innerhalb einer Teilrunde werden dabei vor dem Einspeisen ins Netzwerk von der Sortiereinheit zu *einer* Anfrage zusammengefaßt. Im Anschluß an jede Teilrunde sendet die Sortiereinheit ein sogenanntes *End-Of-Round-Paket*, das die maximale Adresse $\langle 1^{32} \rangle$ hat und vom Typ *EOR* ist.

Beim Versenden der Anfragen an das PRAM-Netzwerk werden Adressen und Daten über denselben Bus geschickt. Falls also eine Anfrage einen Datenteil enthält, wird dieses Datum unmittelbar nach dem Adreßteil in das PRAM-Netzwerk eingespeist. Erst im Anschluß daran kann die nächste Anfrage eingespeist werden. Die SB-PRAM verschickt ihre Antworten auf Speicheranfragen in zwei Teilen, die jeweils ein 16-Bit Datum enthalten. Diese Teilantworten, die nacheinander beim Sortierknoten ankommen, werden als *Halbpakete* bezeichnet. Dabei kommt das Halbpaket, daß die unteren 16 Bit der Antwort enthält, *zuerst* am Sortierknoten an. Ausnahme sind Multipräfix-Zugriffe mit dem Operator *max*. Bei diesen Zugriffen wird das Halbpaket mit den unteren 16 Bit Daten erst *nach* dem Halbpaket mit den oberen 16 Bit gesandt.

Die einzelnen Halbpakete, die beim Sortierknoten S_i eintreffen, enthalten keine Information darüber, zu welcher Speicheranfrage von S_i sie gehören oder welche Hälfte eines Antwortdatums sie enthalten. Der Sortierknoten setzt die beiden Halbpakete einer Antwort richtig zu einem Datum zusammen und entscheidet, zu welcher Speicheranfrage das zusammengesetzte Antwortdatum gehört.

Im Netzwerk der SB-PRAM ist es nicht erlaubt, daß beispielsweise verschiedene Prozessoren auf ein und dieselbe Adresse gleichzeitig sowohl lesend als auch schreibend zugreifen. Deshalb werden in einem *Modulo-Bit* die Netzwerkrunden Modulo 2 mitgezählt. Die einzelnen virtuellen Prozessoren der PRAM können ihre Speicheranfragen nach diesem Modulo-Bit richten, um so verschiedenartige Zugriffe auf eine Adresse in einer Netzwerkrunde zu vermeiden.

Ein Netzwerkknoten c kann durch Aktivieren eines $NWbusy$-Signals anzeigen, daß er zur Zeit keine Speicheranfragen mehr aufnehmen kann. In diesem Fall darf kein anderer Netzwerkknoten eine Speicheranfrage senden, die zu c geroutet würde. Statt dessen schickt der Knoten ein sogenanntes *Ghost-Paket*, bis c das $NWbusy$ zurücknimmt. Dieses Ghost-Paket enthält genau die Adresse der eigentlich zu verschickenden Anfrage und ist vom Typ *GHOST*. Beim Versenden des Datenteils einer Speicheranfrage werden die $NWbusy$-Signale immer ignoriert. Wenn also eine Speicheranfrage einen Datenteil hat, dann wird dieser Datenteil unabhängig von $NWbusy$ immer direkt nach dem zugehörigen Adreßteil an das PRAM-Netzwerk versandt. Ein Sortierknoten reagiert auf die $NWbusy$-Signale der beiden nachfolgenden Knoten der Spalte 1 genau wie ein Netzwerkknoten. Dazu kennt der Sortierknoten die Routing-Bit-Nummer der ersten Netzwerkstufe.

Damit ist die Arbeitsweise des Netzwerks der SB-PRAM spezifiziert. In der Einleitung haben wir festgelegt, daß die im Rahmen dieser Arbeit designte PCI-Karte wie ein PRAM-Prozessor an das Netzwerk der SB-PRAM angeschlossen wird. Damit kann der DMA-Prozessor auf der Karte genau wie ein PRAM-Prozessor auf den Speicher der SB-PRAM zugreifen. Deshalb lassen sich aus diesem Abschnitt schon einige Anforderungen an das Design herleiten.

Bemerkung 1.1.3 *Aus den hier vorgestellten Eigenschaften der SB-PRAM leiten sich die folgenden Anforderungen an das Design ab:*

- *Der DMA-Prozessor braucht wie die PRAM-Prozessoren eine Sortiereinheit, damit die Speicheranfragen einer Teilrunde nach Adressen sortiert an das Netzwerk geschickt werden.*
- *Der DMA-Prozessor soll ein Modulo-Bit synchron zur SB-PRAM simulieren. Dieses Bit wird nach jeder Netzwerkrunde umgeschaltet. Dazu ist die Zahl der Teilrunden, aus denen eine Netzwerkrunde der SB-PRAM besteht, in einem Register des DMA-Prozessors gespeichert. Der PC kann dieses Register schreiben. Der DMA-Prozessor kann die Teilrunden anhand der verschickten* EOR*-Pakete zählen und mit dem eingeführten Register dann auch die Netzwerkrunden. Damit kann ein Modulo-Bit synchron zur SB-PRAM simuliert werden.*
- *Bei Zugriffen auf das PRAM-Netzwerk über den DMA-Prozessor, also bei DMA-Programmen, soll jeweils programmierbar sein, ob die Zugriffe bei aktivem oder inaktivem Modulo-Bit erfolgen.*
- *Die logischen PRAM-Adressen müssen gehasht werden. Der DMA-Prozessor liest die dazu nötige Hashkonstante aus einem internen Register, das der PC schreiben kann. Damit kann der DMA-Prozessor immer mit derselben Hashkonstante wie die SB-PRAM betrieben werden.*
- *Die Routing-Bit-Nummer der ersten Netzwerkstufe der SB-PRAM soll in einem Register des DMA-Prozessors gespeichert sein. Der PC kann dieses Register schreiben. Damit ist die PCI-Karte auch unabhängig von der Routing-Bit-Nummer der ersten Netzwerkstufe der SB-PRAM einsetzbar.*

1.2 DMA-Programme

Die Zugriffe des PCs auf den Speicher der SB-PRAM über den DMA-Prozessor werden in sogenannten DMA-Programmen spezifiziert. Es handelt sich dabei jeweils um Blockzugriffe. DMA-Programme spezifizieren die Start- und Endadresse des Blocks im lokalen RAM der PCI-Karte, die Startadresse des Blocks im Speicher der SB-PRAM und die Art des Zugriffs.

In diesem Abschnitt erarbeiten wir einen Algorithmus zur gleichzeitigen Ausführung mehrerer DMA-Programme, der in Kapitel 2 in Hardware umgesetzt wird. Dazu spezifizieren wir zunächst formal eine Semantik für DMA-Programme und zeigen dann, daß

der angegebene formale Algorithmus diese Semantik erfüllt. Wir formen diesen formalen Algorithmus danach soweit äquivalent um, daß er sich leicht in Hardware umsetzen läßt. Abschließend optimieren wir den Algorithmus für einen maximalen Datendurchsatz.

Nach Bemerkung 1.1.3 sendet der DMA-Prozessor seine Speicheranfragen über eine Sortiereinheit and die SB-PRAM. Wenn wir in den folgenden Abschnitten davon sprechen, daß der DMA-Prozessor Speicheranfragen an die SB-PRAM sendet und von dieser SB-PRAM Antworten erhält, dann meinen wir damit, daß der DMA-Prozessor seine Speicheranfragen an die Sortiereinheit sendet und von dieser Sortiereinheit die Antworten der SB-PRAM erhält.

1.2.1 Semantik

In diesem Abschnitt werden DMA-Programme und ihre Semantik formal definiert. Außerdem wird ein rekursiver Algorithmus zur Ausführung eines DMA-Programms angegeben, der die spezifizierte Semantik eines DMA-Programms erfüllt. Die Korrektheit dieses Algorithmus wird bewiesen. Anschließend wird dieser formale Algorithmus auf die Ausführung mehrerer DMA-Programme erweitert. Wir nehmen im folgenden an, daß das lokale RAM auf der PCI-Karte ein 32×2^K RAM ist.

Definition 1.2.1 *Ein **DMA-Programm** P ist ein 5-Tupel (op, padr, sadr, eadr, data), wobei*

- $op = (op_1, op_2) \in \{LOAD, STORE, MP, SYNC\} \times \{and, or, add, max\}$ *der Modus und Operator für den Zugriff ist (der Operator hat nur bei den Modi* MP *und* SYNC *eine Bedeutung),*
- $padr \in B_{32}$ *die* logische *Startadresse im Speicher der SB-PRAM ist,*
- $sadr \in B_K$ *die Startadresse im lokalen RAM der Karte ist und*
- $eadr \in B_K$ *die Endadresse im lokalen RAM der Karte ist; es gilt* $eadr \geq sadr$.
- $data \in B_{32} \cup \{\sharp\}$ *enthält ein Datum für eine Speicheranfrage oder den Wert* $\sharp$. *Dabei gilt:* $(sadr = eadr) \vee (data = \sharp)$.

Ein DMA-Programm $P = (op, padr, sadr, eadr, data)$ *heißt **Standardanweisung**, wenn* $data = \sharp$ *ist, sonst heißt es **Einzelanweisung**. Man definiert die **Länge** von* P *durch* $l(P) := eadr - sadr + 1$. P sendet Daten, *wenn* $op_1 \neq LOAD$, *und* P erwartet eine Antwort, *wenn* $op_1 \in \{LOAD, MP\}$. *Zusätzlich führt man ein **leeres Programm** Λ ein, das an späterer Stelle nützlich sein wird, und definiert* $l(\Lambda) := 0$. *Abschließend definiert man* $\mathbf{P} := \{P | P \text{ DMA-Programm}\} \cup \{\Lambda\}$.

Wir unterscheiden also 2 Klassen von DMA-Programmen, nämlich Standardanweisungen und Einzelanweisungen. Eine Einzelanweisung bezieht sich dabei nur auf *eine* Adresse im lokalen RAM. Außerdem steht das Datum, das mit der zugehörigen Speicheranfrage an die PRAM gesandt wird, im Feld *data* des DMA-Programms selbst. Solche Einzelanweisungen sollen vom DMA-Prozessor priorisiert werden und damit mit minimaler Latenz

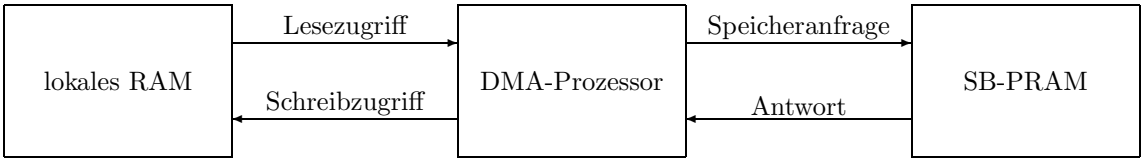


Abbildung 1.3: Ausführung eines DMA-Programms P im DMA-Prozessor

Ein DMA-Programm P kann Daten in Lesezugriffen aus dem lokalen RAM lesen und in Speicheranfragen an die SB-PRAM senden. Erwartet P eine Antwort, dann werden die einzelnen Antworten in Schreibzugriffen in das lokale RAM geschrieben.

ausgeführt werden. Bei Standardanweisungen steht dagegen der Datendurchsatz im Vordergrund.

Informell wird durch ein DMA-Programm $P = (op, padr, sadr, eadr, data)$ mit dem Modus $LOAD$ ein Speicherblock der Größe $eadr - sadr + 1$ von der logischen PRAM-Adresse $padr$ an geladen und ab Adresse $sadr$ im lokalen RAM gespeichert. Genauso wird bei einer Standardanweisung mit dem Modus $STORE$ der Datenblock von $sadr$ bis $eadr$ aus dem lokalen RAM ab der logischen Adresse $padr$ in der SB-PRAM abgelegt. Je nach Zugriffsmodus gliedert sich die Semantik eines DMA-Programms damit in bis zu 4 Teile, nämlich das Lesen von Daten aus dem lokalen RAM, das Senden von Speicheranfragen an die SB-PRAM, der Empfang der Antworten der SB-PRAM und das Schreiben der Antwortdaten ins lokale RAM. Abbildung 1.3 verdeutlicht diesen Zusammenhang.

Wir wollen nun die Semantik eines DMA-Programms formal definieren. Dazu benötigen wir zunächst einige Hilfsdefinitionen.

Definition 1.2.2 *Eine* ***Speicheranfrage*** R *ist ein Tripel* $(op, padr, data)$*, wobei*

- $op = (op_1, op_2) \in \{LOAD, STORE, MP, SYNC\} \times \{and, or, add, max\}$ *Zugriffsmodus und -operator angibt,*
- $padr \in B_{32}$ *die* gehashte *PRAM-Adresse ist und*
- $data \in B_{32} \cup \{\sharp\}$ *entweder das Datum für eine Speicheranfrage oder den Wert* $\sharp$ *enthält, falls* $op_1 = LOAD$ *ist.*

Zusätzlich definiert man $\mathbf{R} := \{R | R \text{ Speicheranfrage}\}$*. Man sagt,* R erwartet eine Antwort*, wenn* $op_1 \in \{LOAD, MP\}$*.*

Definition 1.2.3 *Sei* R *eine Speicheranfrage, die eine Antwort erwartet. Eine* ***Antwort*** *der SB-PRAM auf die Speicheranfrage* R *ist ein Paar* $A = (ret, err)$*, wobei* $ret \in B_{32}$ *das Antwortdatum der SB-PRAM auf* R *ist und* $err \in \{0, 1\}$ *das zugehörige Error-Flag. Dabei ist* $err = 1$ *genau dann, wenn bei der Speicheranfrage* R *ein Fehler im Netzwerk der SB-PRAM aufgetreten ist.*

Die Antwort A der SB-PRAM auf eine Speicheranfrage R hängt neben dem aktuellen Speicherinhalt auch von den Speicheranfragen aller anderen PRAM-Prozessoren ab. Dieser Sachverhalt trifft nicht nur bei Multipräfix-Anfragen zu, sondern auch bei einfachen Lese-Zugriffen. Greift ein PRAM-Prozessor gleichzeitig schreibend auf eine Adresse zu, die der DMA-Prozessor liest, erhält der DMA-Prozessor eine Antwort mit aktivem Error-Bit. Damit ist die Antwort auf eine Speicheranfrage mit Modus $LOAD$ nicht einfach der aktuelle Inhalt der gelesenen Adresse. Es ist also nicht möglich, eine Funktion zu definieren, die nur in Abhängigkeit vom aktuellen Inhalt des PRAM-Speichers und der Speicheranfrage des DMA-Prozessors die Antwort der SB-PRAM berechnet. Diese Funktion müßte auch von den Speicheranfragen aller anderen Prozessoren abhängen. Da die Angabe einer solchen Funktion den Rahmen dieser Arbeit sprengen würde, wird darauf verzichtet.

Für das lokale RAM auf der PCI-Karte läßt sich aber eine Funktion $M_l : B_k \longrightarrow B_{32}$ angeben, die den Inhalt einer Speicheradresse ausliest. Für eine Adresse $adr \in B_K$ im lokalen RAM ist dann $M_l(adr)$ das Datum, das bei einem *Lesezugriff* auf die Adresse adr im lokalen RAM zurückgegeben wird.

Definition 1.2.4 *Ein* ***Schreibzugriff*** *im lokalen RAM ist ein Paar $S = (adr, data)$, wobei $adr \in B_K$ eine Adresse im lokalen RAM angibt, und $data \in B_{32}$ das Datum, das an dieser Adresse im lokalen RAM abgespeichert wird. Man definiert außerdem die Menge* $\mathbf{S} := \{S | S$ *Schreibzugriff im lokalen RAM*$\}$

Nun sind wir in der Lage, die Semantik eines DMA-Programms P formal zu definieren. Dabei setzt sich die Semantik aus 3 Teilen zusammen, den versandten Speicheranfragen, den Schreibzugriffen im lokalen RAM und schließlich einem Error-Bit. Vor dem Versenden der Speicheranfragen an die SB-PRAM werden die logischen PRAM-Adressen dabei noch gehasht. Die Folge der Speicheranfragen beziehungsweise der Schreibzugriffe eines DMA-Programms P wird als ein *Wort* definiert, dessen einzelne Zeichen Speicheranfragen beziehungsweise Schreibzugriffe sind. Dazu benötigen wir eine letzte Hilfsdefinition nach [LMW86].

Definition 1.2.5 *Sei $\Sigma \neq \emptyset$ eine endliche Menge,* Alphabet *genannt. Die Elemente von Σ nennt man* ***Zeichen****. Ein* ***Wort*** *der Länge n über dem Alphabet Σ ist eine Funktion $a : \{0, \ldots, n-1\} \longrightarrow \Sigma$, wobei $n \in \mathbb{N}$ ist. Ein Wort $a : \{0, \ldots, n-1\} \longrightarrow \Sigma$ wird eindeutig beschrieben durch die Folge der Werte $(a(0), \ldots, a(n-1))$. Statt dessen schreibt man auch $(a_0, \ldots, a_{n-1})$ oder nur $a_0 a_1 \ldots a_{n-1}$. Die Menge*

$$\Sigma^n = \{a | a : \{0, \ldots, n-1\} \longrightarrow \Sigma\}$$

ist die Menge aller Worte der Länge n über Σ. Die Menge Σ^0 enthält dabei nur ein Element. Dieses Element heißt das ***leere Wort*** *und wird mit ϵ bezeichnet. Die Menge*

$$\Sigma^* := \bigcup_{n \geq 0} \Sigma^n$$

ist die Menge aller Worte über Σ.

Für die Konkatenation *zweier Worte über* Σ *definiert man einen Operator* $\cdot$ *durch*

$$\cdot : \Sigma^* \times \Sigma^* \longrightarrow \Sigma^* \text{ mit}$$
$$(a_0, \ldots, a_{n-1}) \cdot (b_0, \ldots, b_{m-1}) = (a_0, \ldots, a_{n-1}, b_0, \ldots, b_{m-1}).$$

Insbesondere gilt also $\epsilon \cdot a = a \cdot \epsilon = a$.

Definition 1.2.6 ***Semantik eines DMA-Programms***
Sei $P = (op, padr, sadr, eadr, data)$ *ein DMA-Programm der Länge* $l(P) = n$, *und sei* $hc \in B_{32}$ *die Hashkonstante. Dann definiert die Abbildung*

$$\sigma = (\sigma_1, \sigma_2, \sigma_3) : \mathbf{P} \rightarrow \mathbf{R}^* \times \mathbf{S}^* \times \{0, 1\}$$

die ***Semantik von P****. Die Abbildung* σ_1 *gibt die Folge der* Speicheranfragen *an, die durch das DMA-Programm* P *erzeugt werden,* σ_2 *gibt die Folge der* Schreibzugriffe *im lokalen RAM an, und* σ_3 *liefert ein akkumuliertes* Error-Bit.

Man setzt $\sigma(\Lambda) := (\epsilon, \epsilon, 0)$. *Für* $P \neq \Lambda$ *ist der ersten Teil der Semantik* σ *definiert durch*

$$\sigma_1(P) := R_0 R_1 \ldots R_{n-1}$$

wobei die Speicheranfrage $R_i, 0 \leq i < n$, *gegeben ist durch*

$$R_i := \begin{cases} (op, hpdar_i, M_l(sadr + i)) & \text{falls } P \text{ Standardanweisung und } op_1 \neq LOAD \\ (op, hpdar_i, data) & \text{falls } P \text{ Einzelanweisung und } op_1 \neq LOAD \\ (op, hpdar_i, \sharp) & \text{sonst} \end{cases}$$

mit $hpdar_i = (padr + i) \odot hc$

Wenn $op_1 \notin \{LOAD, MP\}$ *ist, wenn* P *also keine Antwort erwartet, dann definiert man* $(\sigma_2, \sigma_3)(P) := (\epsilon, 0)$.

Sonst sei $\sigma_1(P) = R_0 R_1 \ldots R_{n-1}$, *und sei* $A_i = (ret_i, err_i)$, $0 \leq i < n$, *die Antwort der PRAM auf* $R_i, 0 \leq i < n$. *Wir definieren* $err := \bigvee_{i=0}^{n-1} err_i$ *und* $S_i := (sadr + i, ret_i)$, $0 \leq i < n$. *Damit wird der zweite und dritte Teil der Semantik* σ *definiert durch*

$$(\sigma_2, \sigma_3)(P) := (S_0 S_1 \ldots S_{n-1}, err)$$

Das Error-Bit $\sigma_3(P)$ *hat genau dann den Wert 1, wenn in mindestens einer Antwort* A_i *auf die Speicheranfragen des DMA-Programms* $err_i = 1$ *gilt, also ein Fehler im Netzwerk der SB-PRAM aufgetreten ist.*

Damit ist die Semantik eines DMA-Programms formal definiert. Die obige Definition enthält allerdings einen impliziten Parameter, denn $\sigma_2(P)$ und $\sigma_3(P)$ hängen neben dem DMA-Programm P auch vom Speicherinhalt der SB-PRAM und den Speicheranfragen der anderen PRAM-Prozessoren ab. Wie bereits bemerkt wurde, ist die genaue Abhängigkeit der Antworten der SB-PRAM von den Speicheranfragen aller anderen Prozessoren zu komplex für den Rahmen dieser Arbeit. Deshalb ist auch die Angabe einer Funktion σ als Semantik in Abhängigkeit von den Speicheranfragen aller anderen Prozessoren der SB-PRAM zu aufwendig. Wir verwenden stattdessen einfach die obige Definition mit dem impliziten Parameter für die Semantik σ.

Der DMA-Prozessor führt aber Standardanweisungen nicht *am Stück* aus, sondern er führt mehrere DMA-Programme *parallel* aus. Dazu zerlegt er die Standardanweisungen in sogenannte Teilprogramme (siehe nachfolgende Definition 1.2.7), die bis zu 8 Speicheranfragen an die SB-PRAM senden. Der DMA-Prozessor führt jeweils ein solches Teilprogramm vollständig aus, bevor er damit beginnt, ein anderes DMA-Programm auszuführen. Das nachfolgende Lemma liefert eine formale Spezifikation für die Ausführung von DMA-Programmen in Teilprogrammen und deren Korrektheit.

Definition 1.2.7 *Seien $k, l \in \mathbb{N}$. Dann definiert man $[l]_k := [\frac{l}{k}] \cdot k$. $[\,]$ steht dabei für die* Gaußklammer $[x] := \max\{k \leq x | k \in \mathbb{Z}\}$.

Sei $P = (op, padr, sadr, eadr, data)$ ein DMA-Programm und $l(P) = n$. Gilt $P = \Lambda$ oder $[eadr]_8 = [sadr]_8$, so nennt man P ***primitiv****. Ist P primitiv, dann definieren wir das* ***Teilprogramm*** *von P als P selbst und das* ***Restprogramm*** *von P als das leere Programm Λ. Sei nun P nicht primitiv. Wir setzen $t := 8 - (sadr - [sadr]_8)$. Dann ist $P' := (op, padr, sadr, sadr + t - 1, data)$ das* Teilprogramm *von P und das* Restprogramm *von P ist $P'' := (op, padr + t, sadr + t, eadr, data)$. Mit den obigen Bezeichnungen gilt auch $l(P') = t$ und $l(P'') = n - t$.*

Lemma 1.2.8 ***Ausführung eines DMA-Programms in Teilprogrammen***
Sei $P = (op, padr, sadr, eadr, data)$ ein DMA-Programm mit dem Teilprogramm P' und dem Restprogramm P''. Wir definieren eine Abbildung ϕ rekursiv durch

$$\phi = (\phi_1, \phi_2, \phi_3) : \mathbf{P} \to \mathbf{R}^* \times \mathbf{S}^* \times \{0,1\}$$

$$\phi(P) := \begin{cases} \begin{pmatrix} \sigma_1(P') \cdot \phi_1(P'') \\ \sigma_2(P') \cdot \phi_2(P'') \\ \sigma_3(P') \vee \phi_3(P'') \end{pmatrix} & \text{falls } P \text{ nicht primitiv} \\ \sigma(P) & \text{sonst} \end{cases}$$

Für die so definierte Funktion ϕ gilt: $\phi(P) = \sigma(P)\, \forall P \in \mathbf{P}$.

Beweis: Sei P ein DMA-Programm der Länge n, das Daten sendet und eine Antwort erwartet (für alle anderen DMA-Programme ist höchstens weniger zu zeigen). Sei weiter $P' = (op, padr, sadr, eadr', data)$ das Teilprogramm von P und das Restprogramm von P sei $P'' = (op,\ padr'',\ sadr'',\ eadr,\ data)$. Mit $n' := l(P')$ ist $n'' := l(P'') = n - n'$. Die Rekursionsvorschrift für ϕ ist *endlich*, da die Startadresse $sadr''$ des Restprogramms größer ist als die Startadresse des DMA-Programms selbst, also

$$[sadr'']_8 = [sadr + t]_8 = [[sadr]_8 + 8]_8 = [sadr]_8 + 8 > [sadr]_8 \Longrightarrow n'' < n$$

Im nächsten Rekursionsschritt wird dann das Restprogramm ausgeführt, das aus nur n'' Speicheranfragen besteht, also aus *weniger* Speicheranfragen als das DMA-Programm selbst. Durch die fortgesetzte Bildung des Restprogramms aus dem bisherigen Restprogramm nimmt die Zahl der Speicheranfragen also stetig ab. Nach höchstens endlich vielen Rekursionsschritten wird das Restprogramm deshalb primitiv und die Rekursion bricht

ab. Für die Semantik von P, P' und P'' gelte

$$\begin{aligned}\sigma(P) &= (R_0R_1\dots R_{n-1}, S_0S_1\dots S_{n-1}, err)\\ \sigma(P') &= (R'_0R_1\dots R'_{n'-1}, S'_0S_1\dots S'_{n'-1}, err')\\ \sigma(P'') &= (R''_0R_1\dots R''_{n-n'}, S''_0S_1\dots S''_{n-n'}, err'')\end{aligned}$$

Der Beweis erfolgt durch vollständige Induktion über die Rekursionstiefe k.

Induktionsanfang: $k = 0$

P ist ein primitives Teilprogramm, und es gilt offensichtlich $\phi(P) = \sigma(P)$.

Induktionsvoraussetzung:

Bis zu einer Rekursionstiefe $\leq k$ gelte $\phi(P) = \sigma(P)$.

Induktionsschritt: $k \to k+1$

$$\phi(P) = (\sigma_1(P') \cdot \phi_1(P''), \sigma_2(P') \cdot \phi_2(P''), \sigma_3(P') \vee \phi_3(P''))$$

Auf $\phi(P'')$ läßt sich die Induktionsvoraussetzung anwenden und es gilt

$$\phi(P) = (R'_0R'_1\dots R'_{n'-1} \cdot R''_0R''_1\dots R''_{n-n'-1}, S'_0S'_1\dots S'_{n'-1} \cdot S''_0S''_1\dots S''_{n-n'-1}, err' \vee err'')$$

Da das Teilprogramm P' genau aus den ersten n' Speicheranfragen von P besteht, gilt $(R'_i, S'_i) = (R_i, S_i)$. Die Speicheranfragen und Schreibzugriffe des Teilprogramms und des DMA-Programms stimmen also überein. Für das Restprogramm ergibt sich für R''_i und $sadr''_i$, $0 \leq i < n - n'$:

$$\begin{aligned}R''_i &= (op, (padr'' + i) \odot hc, M_l(sadr'' + i))\\ &= (op, (padr + (n' + i)) \odot hc, M_l(sadr + (n' + i))\\ &= R_{n'+i}\\ sadr'' + i &= sadr + (n' + i)\end{aligned}$$

Die i-te Speicheranfrage R''_i des Restprogramms entspricht also der $n' + i$-ten Speicheranfrage $R_{n'+i}$ des DMA-Programms. Deshalb gilt auch für die zugehörigen Antworten der SB-PRAM $A''_i = (data''_i, err''_i) = (data_{n'+i}, err_{n'+i}) = A_{n'+i}$. Damit stimmt der i-te Schreibzugriff des Restprogramms mit dem $n' + i$-ten Schreibzugriff des DMA-Programms überein, also $S''_i = (sadr'' + i, data''_i) = (sadr + (n' + i), data_{n'+i}) = S_{n'+i}$. Für die Error-Flags gilt schließlich

$$err' \vee err'' = \bigvee_{i=0}^{n'-1} err'_i \vee \bigvee_{j=0}^{n-n'-1} err''_j = \bigvee_{i=0}^{n'-1} err_i \vee \bigvee_{j=n'}^{n-1} err_j = \bigvee_{i=0}^{n-1} err_i = err$$

Insgesamt folgt also die Behauptung $\phi(P) = \sigma(P)$ □

Das folgende Korollar faßt noch einige Aussagen über Teilprogramme zusammen, die direkt aus der Definition folgen oder im Beweis zu Lemma 1.2.8 gezeigt wurden.

Korollar 1.2.9 *Sei $P = (op, padr, sadr, eadr, data)$ ein DMA-Programm und sei weiter $P' = (op, padr, sadr, eadr', data)$ das Teilprogramm von P. Falls P nicht primitiv ist, sei $P'' = (op, padr'', sadr'', eadr, data)$ das Restprogramm von P. Dann gilt:*

- $eadr' = \begin{cases} eadr & \textit{falls } P \textit{ primitiv} \\ [sadr]_8 + 7 & \textit{sonst} \end{cases}$
- $sadr'' = [sadr]_8 + 8$
- *Ist P eine Einzelanweisung, so ist P ein primitives Teilprogramm.*
- *Ist P das Restprogramm eines DMA-Programms P^0 und kein primitives Teilprogramm, dann hat das Teilprogramm P' von P die Länge 8.*

Nach der letzten Bemerkung in Korollar 1.2.9 besteht höchstens das erste und letzte Teilprogramm eines DMA-Programms P aus weniger als 8 Speicheranfragen. Insgesamt kann ein DMA-Programm P in Teilprogrammen ausgeführt werden, also

$$\sigma(P) = (\sigma_1(P^1) \cdots \sigma_1(P^n), \sigma_2(P^1) \cdots \sigma_2(P^n), \bigvee_{i=1}^{n} \sigma_3(P^i))$$

mit Teilprogrammen P^i, $1 \leq i \leq n$.

Mehrere DMA-Programme

Nachdem wir die Semantik und die Ausführung eines DMA-Programms spezifiziert haben, betrachten wir nun die Ausführung mehrerer DMA-Programme. Dazu erweitern wir zunächst die Definition der Semantik. Die Semantik von n DMA-Programmen setzt sich wie die Semantik eines DMA-Programms aus drei Teilen zusammen. Im Unterschied zur Semantik *eines* DMA-Programms besteht der 3. Teil der Semantik von n DMA-Programmen aber aus n Error-Bits.

Definition 1.2.10 ***Semantik mehrerer DMA-Programme***
Für $n \in \mathbb{N}, n \geq 2$ und $P_1, \ldots, P_n \in \mathbf{P}$ definiert man die Semantik

$$\sigma = (\sigma_1, \sigma_2, \sigma_{3,1}, \ldots, \sigma_{3,n}) : \mathbf{P}^n \to \mathbf{R}^* \times \mathbf{S}^* \times \{0,1\}^n$$
$$\sigma(P_1, \ldots, P_n) := \begin{pmatrix} \sigma_1(P_1) \cdots \sigma_1(P_n) \\ \sigma_2(P_1) \cdots \sigma_2(P_n) \\ \sigma_3(P_1) \\ \vdots \\ \sigma_3(P_n) \end{pmatrix}$$

Wie in Definition 1.2.6 geben σ_1 und σ_2 die Folge der Speicheranfragen beziehungsweise Schreibzugriffen an, die durch die DMA-Programme P_1-P_n erzeugt werden. Für $1 \leq i \leq n$ ist $\sigma_{3,i}$ das Error-Bit von P_i. Für ein einzelnes DMA-Programm P ist $\sigma(P)$ nach wie vor gemäß Definition 1.2.6 definiert.

Die DMA-Programme werden also *nacheinander* ausgeführt; die erste Speicheranfrage eines DMA-Programms P_2 erfolgt erst nach der letzten Speicheranfrage von P_1. Der DMA-Prozessor führt jedoch DMA-Programme nicht *nacheinander* aus, sondern er führt DMA-Programme *parallel* aus, indem er jeweils die Teilprogramme von $P_1, \ldots, P_n$ nacheinander

ausführt, bevor er die Restprogramme ausführt. Wir erweitern deshalb die Definition von ϕ auf beliebige n-Tupel von DMA-Programmen $P_1, \ldots, P_n$.

Definition 1.2.11 ***Ausführung mehrerer DMA-Programme in Teilprogrammen*** *Seien $P_1, \ldots, P_n$ DMA-Programme, P_i' das Teilprogramm von P_i, P_i'' das Restprogramm von P_i und $I := \{i | P_i$ nicht primitiv$\}$. Dann definiert man für $n \in \mathbb{N}$*

$$\phi = (\phi_1, \phi_2, \phi_{3,1}, \ldots, \phi_{3,n}) : \mathbf{P}^n \to \mathbf{R}^* \times \mathbf{S}^* \times \{0,1\}^n$$

$$\phi(P_1, \ldots, P_n) := \begin{cases} \begin{pmatrix} \sigma_1(P_1', \ldots, P_n') \cdot \phi_1(P_1'', \ldots, P_n'') \\ \sigma_2(P_1', \ldots, P_n') \cdot \phi_2(P_1'', \ldots, P_n'') \\ \sigma_3(P_1') \vee \phi_{3,1}(P_1'', \ldots, P_n'') \\ \vdots \\ \sigma_3(P_n') \vee \phi_{3,n}(P_1'', \ldots, P_n'') \end{pmatrix} & \text{falls } I \neq \emptyset \\ \\ \sigma(P_1, \ldots, P_n) & \text{sonst} \end{cases}$$

Diese Definition umfaßt insbesondere für $n = 1$ die Definition von $\phi(P)$ aus Lemma 1.2.8.

Diese Funktion ϕ gibt an, wie der DMA-Prozessor mehrere DMA-Programme zusammen ausführt. Man würde jetzt gerne analog zu Lemma 1.2.8 zeigen, daß $\sigma \equiv \phi$ gilt, daß der vorgestellte Algorithmus also genau die definierte Semantik erfüllt. Dieser Beweis muß aber scheitern, da die Speicheranfragen in σ_1 in einer anderen Reihenfolge stehen als in ϕ_1 und damit genauso die Schreibzugriffe in σ_2 in einer anderen Reihenfolge ausgeführt werden als in ϕ_2.

Wir sollten aber zumindest gewährleisten, daß ϕ_1 nur eine Permutation von σ_1 ist, also aus genau denselben Speicheranfragen besteht wie σ_1; diese Speicheranfragen können allerdings in einer anderen Reihenfolge stehen. Ohne weitere Einschränkungen an die auszuführenden DMA-Programme ist selbst das nicht der Fall. So könnten beispielsweise zwei DMA-Programme P_1 und P_2 beide eine Adresse *adr* im lokalen RAM schreiben, die ein drittes DMA-Programm P_3 liest. P_3 sendet dann das gelesene Datum in einer Speicheranfrage an die SB-PRAM. Damit hängen die Speicheranfragen von P_3 davon ab, welches DMA-Programm die Speicherzelle *adr* zuletzt geschrieben hat. Falls bei der Ausführung in Teilprogrammen also P_1 die Adresse *adr* zuletzt schreibt, bei der Ausführung der Semantik nach aber P_2 diese Adresse zuletzt schreibt, dann ergeben sich verschiedene Speicheranfragen.

Das folgende Lemma besagt, daß ϕ_1 nur eine Permutation von σ_1 ist, falls keine zwei DMA-Programme auf dieselbe Adresse im lokalen RAM zugreifen.

Lemma 1.2.12 *Für $n \in \mathbb{N}$ seien $P_i = (op_i, padr_i, sadr_i, eadr_i, data_i)$ DMA-Programme $(0 \leq i < n)$, die Summe der Längen der DMA-Programme sei $N := \sum_{i=0}^{n-1} l(P_i)$. Die Folge der Speicheranfragen der n DMA-Programme gemäß der Semantik σ sei gegeben durch $\sigma_1(P_1, \ldots, P_n) = R_0 \cdots R_{N-1}$ mit $R_i \in \mathbf{R}$. Der Speicherbereich im lokalen RAM, den ein DMA-Programm P_i liest oder schreibt, sei gegeben durch $M_i := \{j | sadr_i \leq j \leq eadr_i\}$.*

Sind die Speicherbereiche der DMA-Programme $P_1, \ldots, P_n$ im lokalen RAM paarweise disjunkt, gilt also $M_i \cap M_j = \emptyset$ für alle $1 \leq i < j \leq n$, dann existiert eine Permutation

$p : \{0, \ldots, N-1\} \to \{0, \ldots, N-1\}$ *mit* $\phi_1(P_1, \ldots, P_n) = R_{p(0)} \cdots R_{p(N-1)}$. *Die Folge der Speicheranfragen nach* ϕ *entspricht der Folge der Speicheranfragen nach der Semantik* σ *also bis auf Vertauschungen.*

Beweis: Nach der Definition von ϕ für einzelne DMA-Programme sei für jedes $1 \leq i \leq n$ $\phi_1(P_i) = \sigma_1(P_i^1) \ldots \sigma_1(P_i^M)$. Dabei sei $P_i^k := \Lambda$, falls P_i^j primitiv für ein $j < k$; wir füllen also die kürzeren Programme mit leeren Teilprogrammen auf, so daß sich immer M Aufrufe von σ ergeben. Führt man die Rekursionsvorschrift von ϕ aus, erhält man

$$\phi_1(P_1, \ldots, P_n) = \sigma_1(P_1^1) \cdots \sigma_1(P_n^1) \cdot \sigma_1(P_1^2) \cdots \sigma_1(P_n^2) \cdots \sigma_1(P_1^M) \cdots \sigma_1(P_n^M)$$

Durch Umordnung der rechten Seite ergibt sich

$$\sigma_1(P_1^1) \cdots \sigma_1(P_1^M) \cdot \sigma_1(P_2^1) \cdots \sigma_1(P_2^M) \cdots \sigma_1(P_n^1) \cdots \sigma_1(P_n^M)$$

Da $M_i \cap M_j = \emptyset$ $(1 \leq i < j \leq n)$, besteht $\sigma_1(P_i^1) \cdots \sigma_1(P_i^M)$ aus genau den Speicheranfragen, die sich durch $\phi_1(P_i)$ ergeben. Also läßt sich die Definition von ϕ jetzt rückwärts anwenden und man erhält mit Lemma 1.2.8

$$\phi_1(P_1) \ldots \phi_1(P_n) = \sigma_1(P_1) \ldots \sigma_1(P_n) = \sigma_1(P_1, \ldots, P_n)$$

Damit haben wir eine Permutation der Speicheranfragen wie gefordert gefunden. □

Sind die lokalen Speicherbereiche der DMA-Programme $P_1, \ldots, P_n$ paarweise disjunkt, so besteht $\phi(P_1, \ldots, P_n)$ aus genau den Speicheranfragen, aus denen auch $\sigma(P_1, \ldots, P_n)$ besteht; sie stehen allerdings in einer anderen Reihenfolge.

Indem wir jedem DMA-Programm P einen eigenen Speicherbereich im lokalen RAM zuordnen, den kein anderes DMA-Programm schreiben oder lesen kann, gewährleisten wir, daß keine zwei DMA-Programme auf dieselbe Speicherzelle im lokalen RAM zugreifen. Erst wenn ein DMA-Programm beendet ist, wird der ihm zugeordnete Speicherbereich im lokalen RAM für neue DMA-Programme frei.

Die Antworten der SB-PRAM hängen allerdings weiterhin von der Reihenfolge der Speicheranfragen ab. Selbst wenn wir fordern würden, daß keine 2 DMA-Programme aus $P_1, \ldots, P_n$ auf dieselbe Speicherzelle im Speicher der SB-PRAM zugreifen, läßt sich dieses Problem nicht vermeiden, da alle anderen PRAM-Prozessoren auch auf diese Speicherzelle zugreifen können. Eine Antwort der SB-PRAM auf eine Speicheranfrage hängt also insbesondere vom Zeitpunkt der Anfrage ab. Deshalb haben wir auf die Forderung verzichtet, daß keine zwei DMA-Programme auf dieselbe Adresse im Speicher der SB-PRAM zugreifen dürfen.

Damit haben wir spezifiziert, wie unser DMA-Prozessor mehrere DMA-Programme parallel ausführt, und haben gezeigt, daß dabei die Semantik bis auf nicht zu vermeidende Änderungen durch Permutation der Speicheranfragen erfüllt wird. In den beiden folgenden Abschnitten werden wir angeben, wie sich die Funktion ϕ leicht berechnen und später in Hardware umsetzen läßt.

Betrachtet man Definition 1.2.11, dann kann man die Funktion ϕ in 2 Schritten berechnen. Zunächst trifft man die Entscheidung, von welchem DMA-Programm das nächste Teilprogramm gemäß der Funktion ϕ ausgeführt wird. Dieses *Scheduling*-Problem wird

im folgenden Abschnitt 1.2.2 bearbeitet. Anschließend muß nur noch ein Teilprogramm ausgeführt werden und das zugehörige DMA-Programm durch sein Restprogramm ersetzt werden. In Abschnitt 1.2.3 erarbeiten wir deshalb einen Algorithmus zur Ausführung eines Teilprogramms und optimieren diesen Algorithmus für die Umsetzung in Hardware.

1.2.2 Scheduling

In diesem Abschnitt wollen wir eine Scheduling-Funktion des DMA-Prozessors berechnen, also eine Funktion f, die uns angibt, von welchem DMA-Programm $P_i \in \{P_1, \ldots, P_n\}$ das nächste Teilprogramm gemäß der Funktion $\phi(P_1, \ldots, P_n)$ ausgeführt wird. Dabei sind nur die nichtleeren Teilprogramme von Interesse, denn leere Teilprogramme, wie sie in der Definition von ϕ vorkommen, müssen nicht ausgeführt werden.

Wir ordnen jedem DMA-Programm P_i eindeutig eine *Kanalnummer* c_i zu. Man sagt in dieser Situation, daß P_i auf dem *DMA-Kanal* c_i ausgeführt wird. Erst nachdem P_i auf c_i ausgeführt worden ist, kann ein anderes DMA-Programm P_j auf c_i ausgeführt werden. Also interessiert uns eine Funktion f, die uns die Nummer des DMA-Kanals angibt, auf dem das nächste Teilprogramm ausgeführt wird. Wir werden diese Funktion so definieren, daß sie für DMA-Programme $P_1, \ldots, P_n$ auf DMA-Kanälen $c_1, \ldots, c_n$ ein Wort zurück gibt. Das i-te Zeichen dieses Wortes ist dabei die Kanalnummer des DMA-Programms, dessen Teilprogramm als i-tes ausgeführt wird. Damit bedeutet beispielsweise $f((P_1, 1), (P_2, 2)) = 1211$, daß bei der Ausführung von P_1 auf Kanal 1 und P_2 auf Kanal 2 zuerst ein Teilprogramm von P_1, dann eins von P_2 ausgeführt wird, und anschließend zweimal ein Teilprogramm von P_1.

Die Forderung, daß keine zwei DMA-Programme auf dieselbe Speicherstelle im lokalen RAM zugreifen, läßt sich dann dadurch realisieren, daß jedem Kanal ein fester Speicherblock im lokalen RAM zugeordnet ist, den die anderen Kanäle weder lesen noch schreiben können. Die Scheduling-Funktion f ergibt sich direkt aus der Definition 1.2.11 der Funktion ϕ.

Definition 1.2.13 *Seien $P_1, \ldots, P_n$ DMA-Programme, $I := \{i | P_i$ nicht primitiv$\}$ und sei P_i'' das Restprogramm von P_i. Sei c_i die Kanalnummer von P_i, $c_i \neq c_j$ für $i \neq j$. Wir schreiben die nichtleeren DMA-Programme als $J := \{j \in \mathbb{N} | P_j \neq \Lambda\} = \{j_1, \ldots, j_l\}$ mit $j_m < j_{m+1}$. Dann ist die* ***Scheduling-Funktion*** *f rekursiv definiert durch*

$$f : (\mathbf{P} \times \mathbb{N})^* \to \{1, \ldots, n\}^*,$$
$$f((P_1, c_1) \cdots (P_n, c_n)) := \begin{cases} c_{j_1} \cdots c_{j_l} \cdot f((P_1'', c_1) \cdots (P_n'', c_n)) & \text{falls } I \neq \emptyset \\ c_{j_1} \cdots c_{j_l} & \text{sonst} \end{cases}$$

Gegenüber Definition 1.2.11 sind hier genau die Kanäle weggefallen, auf denen leere DMA-Programme ausgeführt werden. Die Funktion f ist nur partiell definiert für Worte der Form $w = (P_1, c_1) \cdots (P_n, c_n)$ mit $c_i \neq c_j$ für $i \neq j$, also für DMA-Programme, die auf verschiedenen Kanälen laufen. Wir wollen nun die Funktion f so berechnen, daß die Berechnung möglichst einfach in Hardware umgesetzt werden kann. Das nachfolgende Lemma liefert uns dazu einen Algorithmus, der in jedem Rekursionsschritt nur *ein* DMA-Programm betrachtet.

Lemma 1.2.14 ***Scheduling-Algorithmus***
Sei $g : (\mathbf{P} \times \mathbb{N})^* \to \mathbb{N}^*$ eine Abbildung, die wie die Scheduling-Funktion f nur partiell definiert ist für Worte der Form $w := (P_1, c_1) \cdots (P_n, c_n)$ mit $c_i \neq c_j$ für $i \neq j$, und sei P_1'' das Restprogramm von P_1. Wir setzen

$$w' := \begin{cases} (P_2, c_2) \cdots (P_n, c_n) \cdot (P_1'', c_1) & \text{falls } P_1 \text{ nicht primitiv} \\ (P_2, c_2) \cdots (P_n, c_n) & \text{sonst} \end{cases}$$
$$c := \begin{cases} c_1 & P_1 \neq \Lambda \\ \epsilon & \text{sonst} \end{cases}$$

Damit wird die Abbildung g rekursiv definiert durch

$$g(w) := \begin{cases} c \cdot g(w') & \text{falls } w' \neq \epsilon \\ c & \text{sonst} \end{cases}$$

Dann gilt für alle $((P_1, c_1), \ldots, (P_n, c_n)) \in (\mathbb{N} \times \mathbf{P})^n, c_i \neq c_j$ für $i \neq j$:

$$g\left((P_1, c_1) \cdots (P_n, c_n)\right) = f\left((P_1, c_1) \cdots (P_n, c_n)\right)$$

Beweis: Man sagt in der Situation vom Lemma 1.2.14, daß (P_1, c_1) aus w ausgelesen wird. Falls P_1 nicht primitiv ist, so sagt man, (P_1'', c_1) wird nach w geschrieben. Seien I, J, P_i'' wie in Definition 1.2.13. Wir schreiben die Elemente von I als $I = \{i_1, \ldots, i_k\}$ mit $i_m < i_{m+1}$. Zunächst bemerken wir, daß die Rekursionsvorschrift wohldefiniert ist, denn für das Wort w' mit den Bezeichnungen aus dem Lemma gilt: $c_i \neq c_j$ für $i \neq j$. Außerdem ist (P_2, c_2) das Datum, das im nächsten Rekursionsschritt aus w' ausgelesen wird.

Damit ergibt sich nach n Rekursionsschritten ein Wort $w^{(n)} = (P_{i_1}'', c_{i_1}) \cdots (P_{i_k}'', c_{i_k})$. Dabei gilt $w^{(n)} = \epsilon \Longleftrightarrow I = \emptyset$, und für $w^{(n)} \neq \epsilon$ gilt $g(w) = c_{j_1} \cdots c_{j_l} \cdot g(w^{(n)})$. Nach der Konstruktion von g gilt weiterhin

$$g(w^{(n)}) = g((P_{i_1}'', c_{i_1}) \cdots (P_{i_k}'', c_{i_k})) = g((P_1'', c_1) \cdots (P_n'', c_n))$$

denn für die zusätzlichen (P_k, c_k) auf der rechten Seite gilt $P_k = \Lambda$.

Der Beweis der Behauptung erfolgt durch vollständige Induktion über die Rekursionstiefe m von f.

Induktionsanfang: $m = 0 \Longleftrightarrow I = \emptyset$

$f\left((P_1, c_1) \cdots (P_n, c_n)\right) = c_{j_1} \cdots c_{j_l} = g\left((P_1, c_1) \cdots (P_n, c_n)\right)$

Induktionsvoraussetzung:

Bis zu einer Rekursionstiefe $\leq m$ von f gelte

$$g\left((P_1, c_1) \cdots (P_n, c_n)\right) = f\left((P_1, c_1) \cdots (P_n, c_n)\right)$$

Induktionsschritt: $m \to m + 1$

Mit der Definition von f ergibt sich

$$f\left((P_1, c_1) \cdots (P_n, c_n)\right) = c_{j_1} \cdots c_{j_l} \cdot f\left((P_1'', c_1) \cdots (P_n'', c_n)\right)$$

Auf den Funktionsaufruf von f auf der rechten Seite läßt sich die Induktionsvoraussetzung anwenden, und mit der Vorbemerkung folgt die Behauptung:

$$\begin{aligned} f\left((P_1,c_1)\cdots(P_n,c_n)\right) &= c_{j_1}\cdots c_{j_l}\cdot g((P_1'',c_1)\cdots(P_n'',c_n)) \\ &= c_{j_1}\cdots c_{j_l}\cdot g((P_{i_1}'',c_{i_1})\cdots(P_{i_k}'',c_{i_k})) \\ &= g\left((P_1,c_1)\cdots(P_n,c_n)\right) \end{aligned}$$

□

Wir wollen nun die Bildung von w' aus w nach Lemma 1.2.14 näher untersuchen. Es wird immer das *erste* Zeichen von w ausgelesen, und wenn zurückgeschrieben wird, dann wird ein Zeichen hinter das *letzte* Zeichen von w gehängt. Wir betrachten nun für jedes Zeichen c in w den Zeitpunkt, zu dem c nach w geschrieben wurde. Wir gehen dabei davon aus, daß in dem ursprünglichen Wort w das i-te Zeichen zum Zeitpunkt i nach w geschrieben wurde. Damit sind in dem ursprünglichen Wort w die Zeichen nach ihrer Verweildauer im Wort geordnet, denn das i-te Zeichen von w steht schon länger als das j-te Zeichen c' in w, wenn $i < j$. Insbesondere steht das erste Zeichen von w also am längsten in w und das letzte Zeichen von w am kürzesten in w.

Damit wird durch diesen Mechanismus genau das Zeichen aus w ausgelesen, daß schon am längsten in w steht. In w' sind die Zeichen wie in w nach ihrer Verweildauer im Wort geordnet, denn das neu nach w geschriebene Zeichen wird hinter das bisherige letzte Zeichen geschrieben. Damit ergibt sich induktiv, daß immer das Zeichen aus einem Wort ausgelesen wird, das am längsten in dem Wort steht. Also kann man die Funktion g mit einem Fifo-Buffer (*First-In-First-Out*) berechnen, den man auch einfach als Fifo bezeichnet. Wir nennen diese Fifo zur Berechnung von g den *DMA-Scheduler*.

Insgesamt ergibt sich beim Abarbeiten von DMA-Programmen P_i auf dem Kanal c_i $(1 \leq i \leq n)$ folgende Vorgehensweise:

Schritt 1: Schreibe die DMA-Programme und ihre Kanäle $(P_1,c_1),\ldots,(P_n,c_n)$ nacheinander in genau dieser Reihenfolge in den DMA-Scheduler.

Schritt 2: Ist der DMA-Scheduler leer, so sind $P_1,\ldots,P_n$ beendet.

Schritt 3: Lese ein (P,c) aus dem DMA-Scheduler und führe das Teilprogramm P' von P aus. Falls P nicht primitiv ist, dann schreibe (P'',c) zurück in den DMA-Scheduler, wobei P'' das Restprogramm von P ist. Fahre anschließend auf jeden Fall mit Schritt 2 fort.

Wird ein Paar (P_i'',c_i) nach obiger Vorgehensweise in den DMA-Scheduler zurückgeschrieben, so sagt man, P_i wird *reschedult*. Anstatt die ganzen DMA-Programme im DMA-Scheduler zu speichern, wollen wir den Verwaltungsaufwand im DMA-Scheduler minimieren und dort nur die zugehörigen Kanalnummern speichern. Ein DMA-Programm P_i auf dem DMA-Kanal c_i wird dann in einem Register $DMAPRG[c_i]$ des DMA-Prozessor gespeichert. Zusätzlich führen wir noch 2 Register für jeden DMA-Kanal ein, nämlich $STAT$ und $ERROR$. In dem Register $ERROR[c_i]$ wird $\sigma_3(P_i) \in \{0,1\}$ gespeichert. Das Register $STAT[c_i] \in \{0,1\}$ enthält genau dann eine 1, wenn ein DMA-Programm P_i auf Kanal c_i läuft, wenn also noch nicht alle Teilprogramme $\neq \Lambda$ von P_i ausgeführt worden

sind. Beim Start von P_i auf Kanal c_i setzt man $STAT[c_i] := 1$, $ERROR[c_i] := 0$ und $DMAPRG[c_i] := P_i$.

Der DMA-Prozessor soll nun Einzelanweisungen priorisieren. Dazu besteht der DMA-Scheduler aus 2 getrennten Fifos, einer für Standardanweisungen und einer für Einzelanweisungen. Beim Start werden die einzelnen Kanäle der DMA-Programme dann in die jeweils zugehörige Fifo geschrieben. Solange die Fifo mit den Einzelanweisungen nicht leer ist, werden die auszuführenden DMA-Programme aus dieser Fifo gelesen. Da Einzelanweisungen primitiv sind, werden sie niemals zurückgeschrieben. Ist die Fifo mit den Einzelanweisungen leer, fährt man mit dem 2. Schritt von oben fort und betrachtet dabei nur die Fifo mit den Standardanweisungen. Dabei werden DMA-Programme auch nur in die Fifo mit den Standardanweisungen zurückgeschrieben.

Zusätzlich unterscheidet der DMA-Prozessor noch zwischen 2 Klassen von DMA-Programmen, nämlich denen, die nur bei aktivem Modulo-Bit ausgeführt werden, und denen, die nur bei inaktivem Modulo-Bit ausgeführt werden. Deshalb wird der bisherige DMA-Scheduler nochmals verdoppelt, besteht jetzt also aus 4 Fifo-Buffern. Beim Start eines DMA-Programms wird durch das Schreiben in eine der 4 Fifos festgelegt, zu welcher Klasse von Programmen es gehört. Wir numerieren die 4 Fifo-Buffer dabei folgendermaßen durch: $Fifo_{mod,ea}$ mit $mod, ea \in \{0, 1\}$. Dabei sollen für $x \in \{0, 1\}$ in $Fifo_{1,x}$ genau die DMA-Programme stehen, die bei aktivem Modulo-Bit ausgeführt werden, und in $Fifo_{x,1}$ genau die Einzelanweisungen.

Beim Auslesen eines DMA-Programms aus der Scheduling-Fifo wird jetzt also zuerst anhand des aktuellen Wertes des Modulo-Bits entschieden, welche 2 Fifos für die Ausführung überhaupt in Frage kommen. Anschließend wird die Fifo mit den Einzelanweisungen in der angegebenen Weise priorisiert. Es gilt natürlich, daß ein Restprogramm in genau die Fifo zurückgeschrieben wird, aus der das zugehörige DMA-Programm auch gelesen wurde.

Sei für $1 \leq i \leq n$ P_i ein DMA-Programm auf dem Kanal c_i und $mod, ea : \mathbb{N} \to \{0, 1\}$ Funktionen mit

$$mod(c_i) = 1 \iff P_i \text{ wird bei aktivem Modulo-Bit ausgeführt}$$
$$ea(c_i) = 1 \iff P_i \text{ ist Einzelanweisung}$$

Insgesamt ergibt sich der folgende Algorithmus beim Ausführen von DMA-Programmen P_i auf Kanälen c_i:

Schritt 1: Schreibe die Kanalnummern $c_1, \ldots, c_n$ nacheinander in genau dieser Reihenfolge in den DMA-Scheduler, und zwar in $Fifo_{mod(c_i),ea(c_i)}$. Setze $DMAPRG[c_i] := P_i$, $STAT[c_i] := 1, ERROR[c_i] := 0$ $(1 \leq i \leq n)$. Dadurch werden $P_1, \ldots, P_n$ gestartet.

Schritt 2: Sind alle 4 Fifos des DMA-Schedulers leer, so sind $P_1, \ldots, P_n$ beendet.

Schritt 3: Sei $m \in \{0, 1\}$ der aktuelle Wert des Modulo-Bits. Setze $e := 1$, falls $Fifo_{m,1}$ nicht leer ist, sonst setze $e := 0$. Falls $Fifo_{m,e}$ leer ist, dann fahre mit Schritt 2 fort.

Lese ansonsten einen Kanal c aus dem DMA-Scheduler aus der $Fifo_{m,e}$ und setze $P := DMAPRG[c]$. Führe das Teilprogramm P' von P aus. Falls P nicht primitiv

ist, dann setze $DMAPRG[c] := P''$, wobei P'' das Restprogramm von P ist, und schreibe c zurück in den DMA-Scheduler, und zwar in $Fifo_{m,e}$. Ist P primitiv, dann setze $STAT[c] := 0$. Falls das Error-Bit des Teilprogramms gesetzt ist, also $\sigma_3(P') = 1$, dann setze $ERROR[c] := 1$. Fahre anschließend mit Schritt 2 fort.

Gegebenenfalls führen wir mit obigem Algorithmus im 3. Schritt gar kein Teilprogramm aus, sondern kehren direkt zu 2. zurück. Dieser Fall kann eintreten, wenn zwar noch DMA-Programme abzuarbeiten sind, jedoch keine davon beim aktuellen Wert des Modulo-Bits. Der Algorithmus terminiert aber trotzdem, denn der aktuelle Wert des Modulo-Bits ändert sich nach endlicher Zeit, und nach der Änderung wird dann in jedem Fall ein Teilprogramm ausgeführt.

Außerdem läßt sich der vorgestellte Scheduling-Mechanismus erweitern auf das Starten eines DMA-Programms P auf dem Kanal c, während $P_1, \ldots, P_n$ ausgeführt werden. Für das neu hinzukommende P wird lediglich die Kanalnummer c in die richtige Fifo des DMA-Schedulers geschrieben, das zugehörige DMA-Programm in $DMAPRG[c]$ geschrieben, und die beiden Flags $STAT$ und $ERROR$ wie in Schritt 1 initialisiert. Dabei muß man nur darauf achten, daß nicht gleichzeitig auch ein DMA-Programm P_i reschedult wird. Dann befänden sich nämlich zwei Daten gleichen Alters im DMA-Scheduler und der Fifo-Mechanismus, der immer das eindeutig bestimmte älteste Datum ausließt, würde versagen. Zu gegebener Zeit nach dem Schreiben von P in den DMA-Scheduler wird dann das erste Teilprogramm von P ausgeführt, ohne daß am Scheduling-Mechanismus eine Änderung nötig ist.

1.2.3 Ausführung von Teilprogrammen

Zur Ausführung eines DMA-Programms genügt es nach Lemma 1.2.8, Teilprogramme auszuführen, deren Länge 8 Speicheranfragen nicht übersteigt. Deshalb wird im folgenden Abschnitt nur die Ausführung von Teilprogrammen spezifiziert. Zusammen mit dem Scheduling aus dem vorherigen Abschnitt ist damit die Ausführung mehrerer DMA-Programme gleichzeitig vollständig spezifiziert.

Ein Teilprogramm liest gegebenenfalls Daten aus dem lokalen RAM, um sie in Speicheranfragen an die SB-PRAM zu senden. Außerdem schreibt es Daten ins lokale RAM, die die SB-PRAM als Antworten auf Speicheranfragen verschickt. Also kann ein Teilprogramm sowohl lesend als auch schreibend auf das lokale RAM zugreifen.

Bei dem auf der PCI-Karte verwendeten dynamischen RAM nach Abschnitt 1.3 ist der Datendurchsatz bei *einzelnen* Lese- und Schreibzugriffen sehr gering. Das lokale RAM wird durch die Speicherzugriffe des Teilprogramms also einen großen Teil der Zeit belegt. Gleichzeitig soll es dem PC aber möglich sein, selbst auf das lokale RAM zuzugreifen, um beispielsweise die Daten eines anderen DMA-Programms im RAM zu lesen oder zu schreiben. Deshalb erfolgen die Speicherzugriffe der Teilprogramme in sogenannten *Bursts*. In einem Burst können die Daten mehrerer aufeinanderfolgender Speicherzellen nacheinander gelesen oder geschrieben werden. Dabei hat der Zugriff auf die *erste* Adresse des Bursts zwar dieselbe Latenz wie bei einem Einzelzugriff, die Daten der *Folgeadressen* werden aber erheblich schneller geschrieben oder gelesen. So können beispielsweise mit dem verwende-

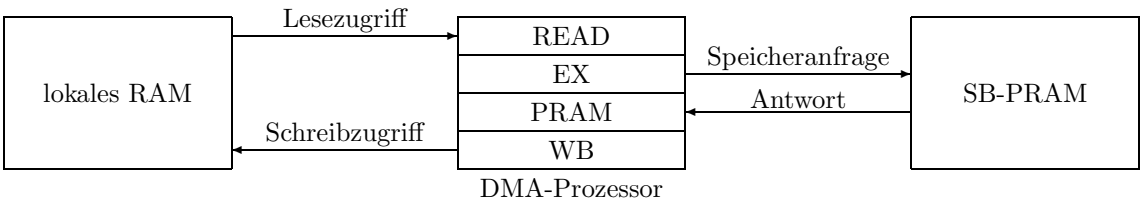


Abbildung 1.4: Ausführung eines Teilprogramms in Stufen

ten SDRAM 8 Daten im Burst in der gleichen Zeit geschrieben werden wie 3 Daten mit Einzelzugriffen [MT98]. Insgesamt ergibt sich bei Bursts also ein wesentlich höherer Datendurchsatz als bei Einzelzugriffen, wenn man bedenkt, daß die meisten Teilprogramme genau 8 Daten lesen oder schreiben.

Da ein Teilprogramm seine Daten im Burst „am Stück“ liest oder schreibt, müssen sie im DMA-Prozessor zwischengespeichert werden. Die verwendeten Zwischenspeicher sind nach dem First-In-First-Out-Prinzip aufgebaut. Die Daten, die eine Standardanweisung an die PRAM sendet, werden in eine *send-Fifo* geschrieben, aus der sie dann beim Versenden der Speicheranfragen ausgelesen werden. Die Antworten der SB-PRAM werden in eine *receive-Fifo* geschrieben, aus der sie dann im Burst ins lokale RAM geschrieben werden.

Beim Hashing einer PRAM-Adresse *padr* wird eine Multiplikation mit der Hashkonstanten *hc* ausgeführt. Da ein Hardware-Multiplizierer sehr teuer ist, multipliziert der PC selbst die logische Startadresse *padr* mit der Hashkonstanten *hc* und berechnet so die gehashte Startadresse *hpadr*. Für den DMA-Prozessor besteht dann das DMA-Programm aus $op, hpadr, sadr, eadr, data$. Der DMA-Prozessor kennt also die logische Startadresse *padr* nicht; er kennt nur die gehashte Startadresse *hpadr* und die Hashkonstante *hc*. Der DMA-Prozessor erhält die gehashten Folgeadressen für eine Standardanweisung durch sukzessive Addition von *hc* zu *hpadr* modulo 2^{32}, denn es gilt:

$$(padr + i) \odot hc = padr \odot hc + i \odot hc = hpadr + i \odot hc \bmod 2^{32}$$

Die Ausführung eines Teilprogramms gliedert sich nach Abbildung 1.4 insgesamt in die 4 Stufen *READ*, *EX* (*Execute*), *PRAM* und *WB* (*Writeback*). In *READ* und *WB* erfolgen die lesenden beziehungsweise schreibenden Speicherzugriffe des Teilprogramms. Die Stufe *EX* sendet die Speicheranfragen des Teilprogramms als eine Teilrunde an die Sortiereinheit, und in der Stufe *PRAM* verweilen die Speicheranfragen in der Sortiereinheit und im Netzwerk der SB-PRAM, und die Antworten der SB-PRAM kommen am DMA-Prozessor an.

Sei $P = (op, padr, sadr, eadr, data)$ das DMA-Programm auf Kanal c, dessen Teilprogramm nach dem Scheduling-Mechanismus als nächstes auszuführen ist, und sei P' das Teilprogramm von P, $n' = l(P')$ und $eadr'$ die Endadresse von P' im lokalen RAM. Sei weiter $hpadr = padr \odot hc$ die gehashte PRAM-Startadresse von P und sei $DMAPRG[c] = (op, hpadr, sadr, eadr, data)$.

Stufe 1: In der Stufe *READ* werden die n' Daten, die P' an die SB-PRAM sendet, aus dem lokalen RAM gelesen und in der *send-Fifo* zwischengespeichert. Wenn

P keine Daten sendet oder eine Einzelanweisung ist, werden keine Daten gelesen. Startadresse für den Lesezugriff ist $sadr$, Endadresse ist $eadr'$.

Stufe 2: In der Stufe *EX* werden die n' Speicheranfragen $R_i, 0 \leq i < n'$, von P' nacheinander an die Sortiereinheit gesandt. Dazu setzen wir zu Beginn der Stufe *EX* ein Register $PADR := hpadr$; nach dem Senden einer Speicheranfrage wird jeweils $PADR := PADR + hc \bmod 2^{32}$ gesetzt. Damit enthält $PADR$ nach dem Versenden von R_{i-1} den Wert $hpadr + i \cdot hc \bmod 2^{32}$, also genau die gehashte PRAM-Adresse von $R_i = (op, hpadr + i \cdot hc \bmod 2^{32}, data_i)$. Dabei ist $data_i := data$, falls P eine Einzelanweisung ist oder keine Daten sendet; sonst werden die $data_i$ sukzessive aus der *send-Fifo* gelesen. Insbesondere ist nach Versenden der letzten Speicheranfrage $R_{n'-1}$ $PADR = hpadr + n' \cdot hc \bmod 2^{32}$, also genau die PRAM-Startadresse des Restprogramms von P, wenn P nicht primitiv ist.

Ist P ein primitives Teilprogramm, das keine Antwort erwartet, dann setzen wir $STAT[c] := 0$. Damit ist die Ausführung von P beendet. Ist P nicht primitiv, setzen wir $DMAPRG[c] := (op, PADR, sadr + n', eadr, data)$ und reschedulen P. Erwartet das DMA-Programm P keine Antwort, dann ist die Ausführung des Teilprogramms von P hiermit beendet.

Stufe 3: Anschließend gelangen die Speicheranfragen von P' in die Stufe *PRAM*. Damit haben sie den DMA-Prozessor verlassen. Die Antworten der SB-PRAM, die aus einem Datum und einem Error-Bit bestehen, werden in *receive* zwischengespeichert. Sind die Antworten auf alle Speicheranfragen von P' in *receive* zwischengespeichert, gelangt das Programm in die Stufe *WB*. Erwartet P keine Antwort, so entfällt diese Stufe.

Stufe 4: In der Stufe *WB* werden die Datenteile der Antworten der PRAM auf die Speicheranfragen von P' ins lokale RAM zurückgeschrieben und die zugehörigen Error-Bits in $ERROR[c]$ akkumuliert. Erwartet P keine Antwort, so entfällt diese Stufe. Start- und Endadresse für den RAM-Zugriff sind $sadr$ beziehungsweise $eadr'$. Ist P ein primitives Teilprogramm, so wird beim Verlassen von *WB* $STAT[c] := 0$ gesetzt. Ist bei einer Antwort der SB-PRAM auf die Speicheranfragen von P' das Error-Bit gesetzt, also $\sigma_3(P') = 1$, so wird $ERROR[c] := 1$ gesetzt.

Die Korrektheit dieser Vorgehensweise folgt direkt aus Lemma 1.2.8. Im Unterschied zu diesem Lemma wurden lediglich die Multiplikation beim Hashing eingespart, die Speicherzugriffe eines Teilprogramms in Bursts zusammengefaßt und das Error-Bit $\sigma_3(P)$ nicht durch ein logisches ODER berechnet, sondern mit Hilfe der Äquivalenz $\exists i | err_i = 1 \Longleftrightarrow \bigvee err_i = 1$.

Pipelining

Ein Teilprogramm sendet bis zu 8 Speicheranfragen als eine Teilrunde an die Sortiereinheit, bevor das Restprogramm zurückgeschrieben wird und zum nächsten Teilprogramm gewechselt wird. Im Idealfall sollten ständig Daten an die Sortiereinheit gesandt werden.

	sequentielle Ausführung							
Zeit	0	1	2	3	4	5	6	7
READ	T_0				T_1			
EX		T_0				T_1		
PRAM			T_0				T_1	
WB				T_0				T_1

gepipelinete Ausführung					
0	1	2	3	4	5
T_0	T_1	T_2	T_3	T_4	T_5
	T_0	T_1	T_2	T_3	T_4
		T_0	T_1	T_2	T_3
			T_0	T_1	T_2

Abbildung 1.5: Idealisiertes Pipelining der Ausführung von Teilprogrammen T_i

Der DMA-Prozessor sollte auch ständig für die Antworten der Sortiereinheit aufnahmebereit sein. Sonst müßte er dem PRAM-Netzwerk mitteilen, daß er nicht mehr aufnahmebereit ist, und dadurch kann es im Netzwerk der SB-PRAM zu Paketstauungen kommen, die den Datendurchsatz senken.

Führt man aber zunächst ein Teilprogramm in allen 4 Stufen aus, bevor man zum nächsten Teilprogramm wechselt, ergibt sich ein sehr geringer Datendurchsatz. Von dem Ziel, ständig Anfragen in den Sortierer einzuspeisen, ist man weit entfernt. Ziel dieses Abschnitts ist deshalb, die beschriebene *sequentielle* Vorgehensweise zu *pipelinen*. So wollen wir erreichen, daß sich *immer* ein Teilprogramm in der Stufe *EX* befindet, also ständig Daten an die Sortiereinheit gesendet werden. Abbildung 1.5 veranschaulicht die idealisierte gepipelinete Ausführung von Teilprogrammen im Vergleich zur sequentiellen Ausführung. Bei diesem Pipelining ergeben sich die folgenden 5 Probleme:

Problem 1: Beim Auslesen eines DMA-Kanals aus der Scheduling-Fifo nach dem Algorithmus aus Abschnitt 1.2.2 darf nicht der aktuelle Wert des Modulo-Bits verwendet werden, sondern der Wert, den das Modulo-Bit haben wird, nachdem das DMA-Programm in die Stufe *EX* diese Stufe verlassen hat. Der aktuelle Wert des Modulo-Bits wird nämlich erst umgeschaltet, nachdem die letzte Speicheranfrage eines Teilprogramms in der Stufe *EX* and die Sortiereinheit gesandt worden ist, und diese umgeschaltete Version muß beim Auslesen aus der Scheduling-Fifo verwendet werden.

Problem 2: Teilprogramme verweilen wesentlich länger in der Stufe *PRAM* als in den anderen Stufen. Außerdem kann der DMA-Prozessor die Verweildauer eines Teilprogramms in der Stufe *PRAM* nicht berechnen, da sie vom PRAM-Netzwerk abhängt.

Problem 3: Die Stufen *EX* und *WB* schreiben beide das Register *STAT*. Die Schreibzugriffe der beiden Stufen können gleichzeitig erfolgen.

Problem 4: Die Stufen *READ* und *WB* greifen beide auf das lokale RAM zu. Dieses RAM kann aber immer nur *einen* Speicherzugriff zu einem Zeitpunkt behandeln.

Problem 5: In der Stufe *EX* wird das Restprogramm von P_i nach $DMAPRG[c_i]$ geschrieben, und in der Stufe *READ* des nächsten Teilprogramms wird $DMAPRG[c_j]$ gelesen, wobei $i = j$ möglich ist. Die Stufe *READ* liest also Daten, die vom vorhergehenden Teilprogramm erst am Ende der Stufe *EX* geschrieben werden.

Wir wollen nun diese 5 Probleme nacheinander lösen und anschließend den sich ergebenden Algorithmus zur Ausführung eines Teilprogramms zusammenfassen.

Problem 1 Wir lösen Problem 1 dadurch, daß wir das Modulo-Bit vorausberechnen. Das vorausberechnete Modulo-Bit hat genau dann einen anderen Wert als das aktuelle Modulo-Bit, wenn das aktuelle Modulo-Bit nach Versenden der Teilrunde des Programms in der Stufe *EX* umgeschaltet wird. Man bezeichnet dieses Umschalten als Toggeln. Das Toggeln ist vom DMA-Programm P in der Stufe *EX* völlig unabhängig; es spielt lediglich eine Rolle, nach wie vielen Teilrunden das Modulo-Bit überhaupt getoggelt wird.

Das eigentliche Modulo-Bit wird nur beim Zurückschreiben eines Kanals in den DMA-Scheduler gelesen. Da das Zurückschreiben aber immer in genau die Fifo erfolgt, aus der auch gelesen wurde, kann die Auswahl der Fifo für das Zurückschreiben auch *lokal* im DMA-Scheduler erfolgen. Wenn wir also den DMA-Scheduler so konstruieren, daß er Kanalnummern genau in die Fifo zurückschreibt, aus der sie auch gelesen wurden, dann wird das eigentliche Modulo-Bit der Stufe *EX* überhaupt nicht mehr gelesen. Dazu erweitern wir den DMA-Scheduler um die Register MOD und EA. Im Register MOD speichert der DMA-Scheduler das Modulo-Bit für den zuletzt ausgelesen Kanal, und in EA speichert er, ob das DMA-Programm auf dem Kanal eine Einzelanweisung ist. Das eigentliche Modulo-Bit des DMA-Prozessors kann deshalb weggelassen werden, und wir bezeichnen das vorausberechnete Modulo-Bit einfach als das Modulo-Bit.

Problem 2 Das 2. Problem ist die unbekannte Latenz in der Stufe *PRAM*. Diese Problem läßt sich einfach dadurch lösen, daß man mehreren Teilprogrammen erlaubt, sich gleichzeitig in der Stufe *PRAM* zu befinden. Da in der Stufe *PRAM* im DMA-Prozessor nur die ankommenden Antworten der SB-PRAM in *receive* gespeichert werden, müssen in dieser Stufe keine zusätzlichen Anpassungen vorgenommen werden. Allerdings sind dann für die Stufe *WB* ein paar Anpassungen nötig. Diese Stufe liest nämlich Teile des DMA-Programms P, nachdem *EX* dieses P im Register $DMAPRG[c]$ möglicherweise schon mehrfach überschrieben hat. Außerdem muß die Kanalnummer des DMA-Programms in der Stufe *EX* natürlich nicht mit der des Programms in der Stufe *WB* übereinstimmen. Die Stufe *WB* liest die Start- und Endadresse des Teilprogramms, $sadr$ und $eadr'$, die Kanalnummer c und ein Flag $primitiv \in \{0,1\}$ mit $primitiv = 1 \Leftrightarrow P$ primitiv. Diese Daten werden als *Write-Infos* bezeichnet.

Die Stufe *EX* schreibt die Write-Infos von P deshalb in eine *address-Fifo*, wenn P eine Antwort erwartet. Aus der *address-Fifo* werden die Write-Infos dann in *WB* ausgelesen. Das Teilprogramm, dessen Write-Info in *WB* ausgelesen werden muß, ist nämlich genau das Teilprogramm, dessen Write-Infos am längsten zwischengespeichert sind.

Beim Übergang von *PRAM* nach *WB* wird geprüft, ob alle Antworten auf Speicheranfragen des Teilprogramms von P eingetroffen sind. Dazu wird in der Stufe *EX* genau die *letzte* Speicheranfrage eines Teilprogramms mit einem Flag *lor* markiert (*last of round*). Unsere Sortiereinheit liefert dieses Flag mit der zugehörigen Antwort zurück. Wir nennen das zurückgegebene Flag das *package*-Flag. Damit kann ein Teilprogramm genau dann von der Stufe *PRAM* in die Stufe *WB* übergehen, wenn das *package*-Flag einer Antwort

aktiv ist, die in die *receive*-Fifo geschrieben wurde.

Problem 3 Das 3. Problem ist ein gleichzeitiger Schreibzugriff auf das *STAT*-Register aus den beiden Stufen *WB* und *EX*. Dieses *STAT*-Register haben wir in der Hardware als *dual-port-RAM* realisiert (siehe Abschnitt 1.4). Damit unterstützt das Register einen Lesezugriff parallel zu einem Schreibzugriff, allerdings keine 2 parallelen Schreibzugriffe. Wir benutzen den Lesezugriff parallel zum Schreibzugriff später zur Berechnung eines Statusworts des DMA-Prozessors.

Wir lösen dieses Problem, indem wir die Schreibzugriffe von *WB* auf *STAT* aussetzen, bis *EX* das Register *STAT* nicht mehr schreibt. Das Aussetzen eines Zugriffs auf eine Ressource, während gerade ein anderer Zugriff auf diese Ressource erfolgt, wird als *Stalling* bezeichnet. Wir stallen also den Schreibzugriff von *WB* gegebenenfalls.

Problem 4 Um das 4. Problem zu lösen, also die Speicherzugriffe in verschiedenen Stufen, müssen wir gewährleisten, daß *nie* ein Teilprogramm in der Stufe *READ* ist, während sich ein anderes in der Stufe *WB* befindet. Gegebenenfalls wird ein Teilprogramm deshalb vor der Stufe *WB* aufgehalten, wenn sich ein anderes in der Stufe *READ* befindet. Wenn allerdings *immer* ein Teilprogramm in der Stufe *EX* ist, so hat es sich zuvor in der Stufe *READ* befunden, wenn es Daten sendet und keine Einzelanweisung ist. Also befindet sich schlimmstenfalls immer ein Teilprogramm in der Stufe *READ* und ein Teilprogramm *vor* der Stufe *WB* wird auf unbestimmte Zeit angehalten.

Wir lösen dieses Problem, indem wir Teilprogramme in der Stufe *EX* doppelt so lang verweilen lassen wie in den Stufen *READ* und *WB*. Diese Vorgehensweise ist auch durch die Tatsache motiviert, daß in *READ* oder *WB* im Burst in jedem Takt ein Datum gelesen oder geschrieben werden kann, pro gelesenem Datum aber 4 Takte lang Daten an die Sortiereinheit geschickt werden. Dann kann beispielsweise in der *zweiten* Hälfte der Stufe *EX* die Stufe *READ* des nächsten Teilprogramms auf das RAM zugreifen. Anschließend gelangt dieses Teilprogramm in die Stufe *EX*, und in der ersten Hälfte von *EX* kann dann die Stufe *WB* für ein anderes Teilprogramm aktiv werden.

Insgesamt ergibt sich dann ein Pipelining, wie es in Abbildung 1.6 gezeigt ist. Die Stufe *READ* wird dabei immer parallel zu einer 2. Hälfte von *EX* aktiv. Die Stufe *WB* kann jederzeit aktiv werden, wenn die Stufe *READ* nicht aktiv ist.

Da die Stufe *WB* jetzt gegebenenfalls gestallt wird, funktioniert der Übergang von *PRAM* zu *WB* bei aktivem *package*-Flag einer Antwort nicht mehr. Während ein DMA-Programm vor der Stufe *WB* gestallt wird, könnte ein weiteres *package*-Flag eintreffen. Dieses Flag darf nicht ignoriert werden, denn jetzt muß die Stufe *WB* noch für *zwei* Teilprogramme aktiv werden. Deshalb werden in der Stufe *PRAM* die ankommenden *package*-Flags in einem Zähler *pcnt* gezählt (*package count*). Hat dieser Zähler nicht den Wert Null, dann wartet ein DMA-Programm darauf, die Stufe *WB* zu betreten. Wenn ein DMA-Programm die Stufe *WB* betritt, wird *pcnt* dekrementiert. Betritt ein DMA-Programm die Stufe *WB*, während gleichzeitig ein *package*-Flag ankommt, wird *pcnt* nicht verändert.

Zeit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
READ		T_0		T_1				T_3				T_5		T_6		T_7
EX			T_0		T_1		T_2		T_3		T_4		T_5		T_6	
PRAM					T_0		T_0		T_2 T_0		T_3 T_2		T_4 T_3 T_2		T_5 T_4 T_3	
WB											T_0				T_2	

Abbildung 1.6: Gepipelinete Abarbeitung von Teilprogrammen T_i

Die Teilprogramme T_2 und T_4 senden keine Daten oder sind Einzelanweisungen. Das Teilprogramm T_1 erwartet keine Antwort. Es befindet sich nie ein Teilprogramm in der Stufe *WB*, während ein anderes sich in der Stufe *READ* befindet.

Problem 5 Die Tatsache, daß Teilprogramme doppelt so lange in der *EX*-Stufe verweilen wie in der Stufe *READ*, kommt uns auch beim Lösen des letzten Problems zu Gute. Die Stufe *READ* liest *nur* die Felder *sadr* und *eadr* eines DMA-Programms. Das Feld *eadr* wird beim Zurückschreiben in der Stufe *EX* nie geändert, kann also ohne Probleme verwendet werden. Lediglich *sadr* wird von der Stufe *EX* modifiziert.

An dieser Stelle stellt sich das Problem, daß das DMA-Programm erst nach dem Versenden der letzten Speicheranfrage zurückgeschrieben werden kann, weil erst dann durch Addition die PRAM-Adresse des Restprogramms, $hpadr + n' \cdot hc \bmod 2^{32}$, berechnet worden ist. Andererseits ist die Startadresse des Restprogramms im lokalen RAM, $sadr + n'$, gleich zu Anfang von *EX* berechenbar. Deshalb erfolgt das Überschreiben des DMA-Programms durch sein Restprogramm in der Stufe *EX* in *zwei* Schritten: in der *ersten* Hälfte von *EX* wird die lokale Startadresse $sadr + n'$ zurückgeschrieben, und nach Versenden der letzten Speicheranfrage erst die PRAM-Adresse $hpadr + n' \cdot hc \bmod 2^{32}$. Dann kann die *neue* lokale Startadresse zu Beginn der *zweiten* Hälfte von *EX* wieder ausgelesen werden, und das DMA-Programm kann die Stufe *READ* betreten. Die neue PRAM-Adresse wird erst in der Stufe *EX* gelesen; damit genügt es, wenn dieses Register erst am Ende der Stufe *EX* zurückgeschrieben wird.

Zusätzlich gewährleisten wir, daß P in der *ersten* Hälfte von *EX* reschedult wird. Dieser Sachverhalt ist sehr nützlich, wenn nur *ein* DMA-Programm ausgeführt wird. Dann kann dieses DMA-Programm nämlich die Stufe *READ* betreten, während es sich noch in der 2. Hälfte der Stufe *EX* befindet. Damit kann sich dieses eine DMA-Programm nämlich immer, wenn das Modulo-Bit den richtigen Wert hat, in der Stufe *EX* befinden. Würde dagegen ein DMA-Programm erst in der 2. Hälfte von *EX* reschedult, also *nachdem* das nächste DMA-Programm aus dem DMA-Scheduler ausgelesen worden ist, dann würde in obigem Fall kein Teilprogramm ausgeführt. Erst beim *nächsten* Auslesen stünde das DMA-Programm dann wieder im DMA-Scheduler und könnte ausgeführt werden. Rescheduling in der ersten Hälfte von *EX* ist also für die Korrektheit des Algorithmus nicht notwendig, wird aber wegen der Performancesteigerung implementiert.

Nun ist auch klar, warum in der Abbildung 1.6 *READ* nur parallel zur 2. Hälfte von

EX aktiv wird. Die erste Hälfte wird benötigt, um die lokale Startadresse *sadr* in der angegebenen Weise zu überschreiben und P zu reschedulen. Die Daten, die parallel zur 2. Hälfte von *EX* aus dem lokalen RAM gelesen werden, können dann gleich zu Beginn der *nächsten EX*-Stufe versandt werden.

Zusammenfassung des Algorithmus Insgesamt ergibt sich also bei der Ausführung eines Teilprogramms die folgende Vorgehensweise:

Stufe 1: Die Stufe *READ* wird nur in der 2. Hälfte einer *EX*-Stufe aktiv. Falls *WB* gerade noch auf das RAM zugreift, wartet *READ*, bis dieser Zugriff abgeschlossen ist. Dann werden die Daten von *sadr* bis $eadr'$ aus dem lokalen RAM gelesen und in die *send-Fifo* geschrieben.

Stufe 2: In der *EX*-Stufe werden die Write-Infos in die *address-Fifo* geschrieben. Ist P kein primitives Teilprogramm, dann wird in $DMAPRG[c]$ der Parameter *sadr* durch $sadr + n'$ ersetzt und P wird reschedult. Die n' Speicheranfragen R_i, $0 \leq i < n$, werden an die SB-PRAM versandt; dabei ist R_i genau wie zuvor aufgebaut. Genau die letzte Speicheranfrage des Teilprogramms ist dabei mit einem *lor*-Flag $lor = 1$ markiert. Anschließend wird auch die PRAM-Adresse *hpadr* nach $DMAPRG[c]$ zurückgeschrieben, wenn P nicht primitiv ist. Ist P primitives Teilprogramm, das keine Antwort erwartet, setzt man $STAT[c] := 0$. Damit ist P beendet.

Stufe 3: In der Stufe *PRAM* werden die Antworten der PRAM in die *receive-Fifo* geschrieben. Ist bei einer Antwort das Flag *package* aktiv, wird *pcnt* hochgezählt. Hat *pcnt* nicht den Wert Null, wartet ein DMA-Programm darauf, die Stufe *WB* zu betreten.

Stufe 4: Die Stufe *WB* kann nur aktiv werden, wenn *READ* gerade nicht aktiv ist. In diesem Fall liest *WB* die ältesten Write-Infos $(sadr, eadr', c, primitiv)$ aus der *address-Fifo*, und die Datenteile der Antworten der SB-PRAM werden aus der *receive*-Fifo ausgelesen und von Adresse *sadr* bis zur Adresse $eadr'$ abgespeichert. Der Zähler *pcnt* wird dekrementiert. Falls das Error-Bit einer Antwort den Wert 1 hat, wird $ERROR[c] := 1$ gesetzt; falls $primitiv = 1$ ist, setzt man $STAT[c] := 0$. Damit ist P beendet. Dieser Schreibzugriff auf das *STAT*-Register wird gestallt, falls *EX* das *STAT*-Register gerade schreibt.

Formal ergibt sich der folgende Algorithmus in einer C-ähnlichen Notation. Dabei gebe ein Fifo-Buffer `F` durch Aufruf von `F.pop()` das Datum zurück, das aus `F` ausgelesen worden ist; `F.push(d)` schreibe ein Datum `d` in den Fifo-Buffer `F`. Außerdem werde durch `DMAScheduler.pop()` die nächste Kanalnummer gemäß des Scheduling-Algorithmus aus dem DMA-Scheduler ausgelesen und als Ergebnis des Funktionsaufrufs zurückgegeben. Durch `DMAScheduler.push(c)` werde das DMA-Programm auf Kanal c reschedult. Weiter werde durch `Send_Request(op,padr,data,lor)` die Speicheranfrage `R=(op,padr,data)` mit *lor*-Flag `lor` an die Sortiereinheit versendet. Die Funktion `Get_Answer()` warte auf die nächste Antwort der SB-PRAM und gebe ein Tripel `(package,err,dat)` zurück, wobei `(dat,err)` die Antwort der SB-PRAM ist und `package` das zugehörige *package*-Flag.

Im DMA-Prozessor werden DMA-Programme durch ihre Kanalnummer eindeutig identifiziert. Deshalb verwenden wir im nachfolgenden Algorithmus die Variable P für die Kanalnummer eines DMA-Programms. Diese Kanalnummer P steht dann für das ganze DMA-Programm.

Algorithmus Stufe *READ*

```
P    = DMAScheduler.pop();  /* Kanalnummer auslesen */
EA   = DMAScheduler.EA;     /* merken, ob Einzelanweisung */
SADR = DMAPRG[P].sadr;      /* DMA-Programm (bis auf padr) lesen */
EADR = DMAPRG[P].eadr;
DATA = DMAPRG[P].data;
OP   = DMAPRG[P].op;
primitiv = SADR/8==EADR/8;  /* primitiv nach Definition berechnen*/
EADR'= primitiv? EADR: 8*(SADR/8)+7;  /* EADR' nach Def. berechnen */
if (!EA && OP1!=LOAD)       /* Standardanweisung, die Daten liest? */
{
    while(RAM_Zugriff)      /* WB gerade aktiv? */
    {}                      /* warten, bis WB beendet */
    RAM_Zugriff=1;          /* RAM-Zugriff beginnen */
    for (adr=SADR; adr<=EADR'; adr++)
    {
        d = M(adr);         /* Lesezugriff auf lokales RAM */
        send.push(d);       /* Datum in send-Fifo schreiben */
    }
    RAM_Zugriff=0;          /* RAM-Zugriff beenden */
}
```

Algorithmus Stufe *EX*

```
PADR     = DMAPRG[P].padr;       /* PADR lesen */
n'       = EADR' - SADR + 1;
if (OP1==LOAD || OP1==MP)        /* erwartet eine Antwort? */
    address.push(primitiv,SADR,EADR',P);
if (!primitiv)                   /* nicht primitiv? */
{
    DMAPRG[P].sadr=SADR + n';    /* sadr zurueckschreiben */
    DMAScheduler.push(P);        /* und Kanal reschedulen */
}
for (i=0; i<n'; i++)
{
    if (EA)
        d = DATA;                /* Datum bei EA ist DATA */
    else
        d = OP1==LOAD? #: send.pop();
        /* bei Standardanweisung aus send-Fifo lesen, */
```

```
        /* falls Daten gesendet werden, sonst # */
    lor = (i==n'-1);              /* lor bei letzter Anfrage */
    Send_Request(OP,PADR,d,lor); /* Speicheranfrage und lor senden */
    PADR = PADR + HC;             /* PADR um HC erhoehen */
}
if (!primitiv)
    DMAPRG[P].padr = PADR;        /* padr zurueckschreiben */
if (primitiv && (OP1==STORE || OP1==SYNC))
    /* primitives Programm, das keine Antwort erwartet? */
{
    STAT_Zugriff=1;               /* Zugriff auf STAT beginnen */
    STAT[P]=0;                    /* Programm beendet */
    STAT_Zugriff=0;               /* Zugriff auf STAT beenden */
}
```

Algorithmus Stufe *PRAM*

```
(package,err,dat)=Get_Answer();  /* Antwort und package-Flag lesen */
receive.push(err,dat)            /* Antwort in receive-Fifo schreiben */
if (package)                     /* letzte Antwort von Teilprogramm? */
    pcnt++;                      /* pcnt inkrementieren */
```

Algorithmus Stufe *WB*

```
if (pcnt!=0)
{
    while(RAM_Zugriff)           /* READ gerade aktiv? */
    {}                           /* warten, bis READ beendet */
    RAM_Zugriff=1;               /* RAM-Zugriff beginnen */
    (primitiv_wb,SADR_wb,EADR'_wb,P_wb)=address.pop();
        /* Write-Info aus address-Fifo lesen */
    sn_error=0;                  /* sn_error initialisieren */
    pcnt--;                      /* pcnt dekrementieren */
    for (adr=SADR_wb; adr<=EADR'_wb; adr++)
    {
        (error,d)=receive.pop(); /* Antwort aus receive-Fifo lesen */
        M(adr) = d;              /* Datum speichern */
        if (error)
            sn_error=1;          /* Fehler akkumulieren */
    }
    RAM_Zugriff=0;               /* RAM-Zugriff beenden */
    if (sn_error)                /* Fehler aufgetreten? */
        ERROR[P_wb] = 1;         /* ERROR-Flag setzen */
    if (primitiv_wb)             /* primitives Teilprogramm? */
    {
        while (STAT_Zugriff)     /* warten, bis STAT-Register frei */
        {}
```

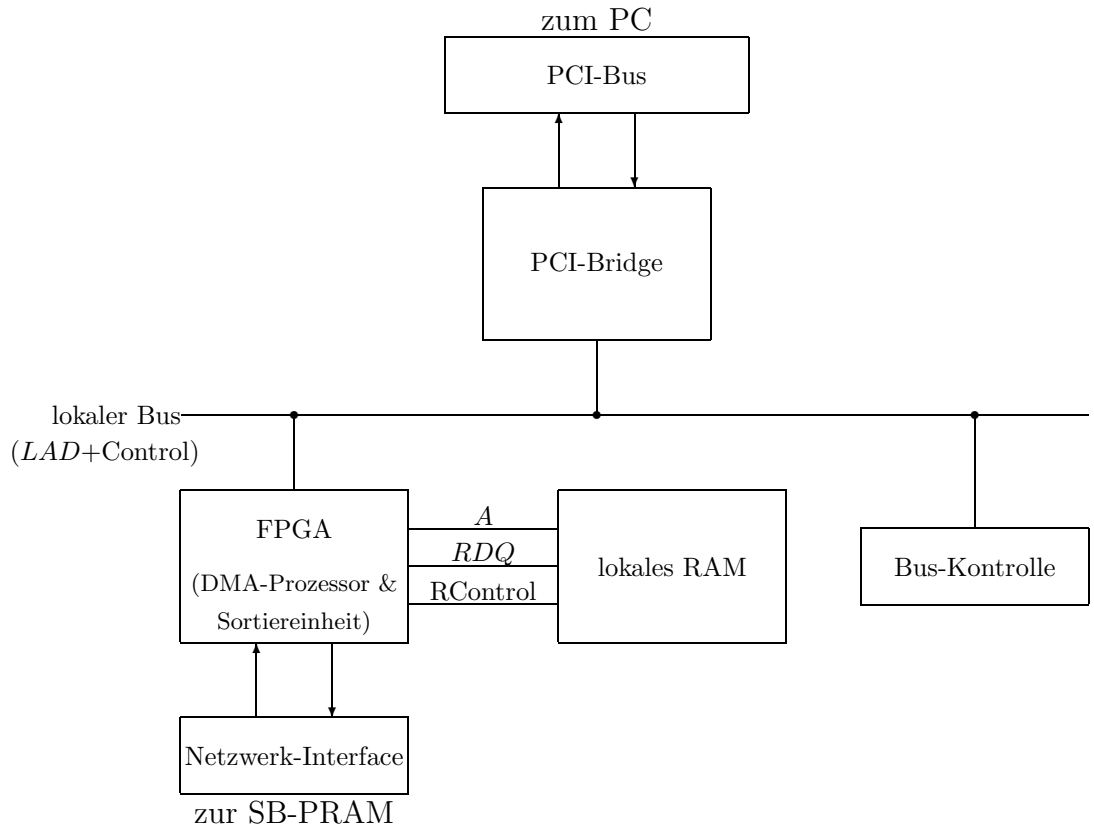


Abbildung 1.7: Blockschaltbild der PCI-Karte

```
        STAT[P_wb] = 0;          /* Programm beendet */
    }
}
```

Stalling wird in diesem Algorithmus mit den beiden Hilfsvariablen `RAM_Zugriff` und `STAT_Zugriff` realisiert. Ist `RAM_Zugriff=1`, dann erfolgt gerade ein Zugriff auf den lokalen Speicher. In diesem Fall kann die Stufe *WB* beziehungsweise *READ* nicht aktiv werden. Der Schreibzugriff von *WB* auf das *STAT*-Register wird so lange gestallt, wie `STAT_Zugriff` aktiv ist.

In Kapitel 2 haben wir genau diesen Algorithmus zur gepipelineten Ausführung von DMA-Programmen und den Scheduling-Mechanismus aus dem letzten Abschnitt in Hardware umgesetzt.

1.3 PCI-Karte

Die PCI-Karte besteht neben dem PCI- sowie dem Netzwerkinterface im wesentlichen aus einem lokalen Speicher und einem FPGA, in dem der DMA-Prozessor und die Sortiereinheit untergebracht sind (siehe Abbildung 1.7). Der PC kann nur über das FPGA

auf den lokalen Speicher zugreifen. Für das Design wurde eine *PCIPRO*-Karte der Firma ATMedia mit zugehöriger Treibersoftware verwendet [Ja00]. Diese PCI-Karte bietet genau die benötigten Datenpfade. Zusätzlich kann eine weitere Karte über sogenannte CMC-Konnektoren (*Common Mezzanine Card* [CMC95]) auf diese PCI-Karte aufgesetzt werden. Auf dieser Aufsatzkarte, der sogenannten *Mezzaninkarte*, ist dann das Netzwerkinterface untergebracht.

Die PCI-Bridge stellt einen lokalen 32-Bit Bus *LAD* mit Kontrollsignalen *Control* zur Verfügung, der mit einer Frequenz von 33 MHz betrieben wird. Auf diesem Bus liegen nacheinander Adressen und Daten von Zugriffen. Das zugehörige Busprotokoll und die Kontrollsignale *Control* werden in Abschnitt 2.1.1 eingeführt. Das FPGA ist über einen 32 Bit breiten Datenbus *RDQ* mit dem lokalen Speicher verbunden. Das FPGA treibt außerdem den Adreßbus *A* und die Kontrollsignale *RControl* für das lokale RAM. Diese Kontrollsignale sowie das verwendete Busprotokoll werden in Abschnitt 2.1.3 eingeführt. Eine Buskontrolle hängt am lokalen Bus und ermöglicht unter anderem die Konfiguration des FPGAs. Über die CMC-Konnektoren ist das FPGA mit der Aufsatzkarte verbunden, dem Netzwerkinterface. Dabei sind die Sortiereinheit, die nach Bemerkung 1.1.3 benötigt wird, und der DMA-Prozessor beide in dem einen FPGA der PCI-Karte realisiert.

Als PCI-Bridge verwenden wir einen Chip PCI9080 der Firma PLX Technologies [PLX98]. Die Buskontrolle ist in einem CPLD realisiert (*Complex Programmable Logic Device*). Der DMA-Prozessor und die Sortiereinheit sind beide zusammen in *einem* FPGA (*Field Programmable Gate Array* [FPGA99]) der Firma Xilinx untergebracht. Dieses FPGA wird wie das CPLD mit einer Spannung von 3.3V betrieben. Als lokaler Speicher wird SDRAM verwendet (*Synchronous Dynamic RAM* [MT98]). Dieses SDRAM wird mit einer eigenen Clock unabhängig von der Clock des lokalen Busses betrieben, da es Zugriffe mit einer Frequenz von bis zu 100 MHz unterstützt. Auch das SDRAM wird mit 3.3V betrieben.

Das FPGA muß bei jedem Hochfahren des Rechners neu konfiguriert werden. Mit Hilfe der Treibersoftware der Firma ATMedia kann das FPGA über die Buskontrolle konfiguriert werden. Zusätzlich bietet die Karte die Möglichkeit, das FPGA über ein EPROM (*Electrically Programmable Read Only Memory*) auf der Karte zu konfigurieren. Beim Hochfahren des PCs wird das FPGA dann automatisch aus dem EPROM der PCI-Karte konfiguriert. Diese EPROM kann auch über die Buskontrolle geschrieben werden, so daß der DMA-Prozessor beim Hochfahren des Rechners ohne weitere Zugriffe auf die PCI-Karte automatisch aus dem EPROM konfiguriert werden kann. Die Konfiguration des CPLD ist nicht flüchtig; es muß also beim Hochfahren des PCs nicht neu konfiguriert werden. Über das FPGA kann man programmieren, mit welcher Taktfrequenz ein Clockgenerator das Interface zum SDRAM betreibt. Damit läßt sich die Taktfrequenz für das SDRAM der Taktfrequenz im FPGA anpassen.

1.3.1 Mezzaninkarte

Auf der Mezzaninkarte befindet sich das Netzwerkinterface, das wie die SB-PRAM mit einer Spannung von 5.0V betrieben wird. Dieses Interface besteht im wesentlichen aus den CMC-Konnektoren, PRAM-Konnektoren, registrierten Treibern für alle Signale und

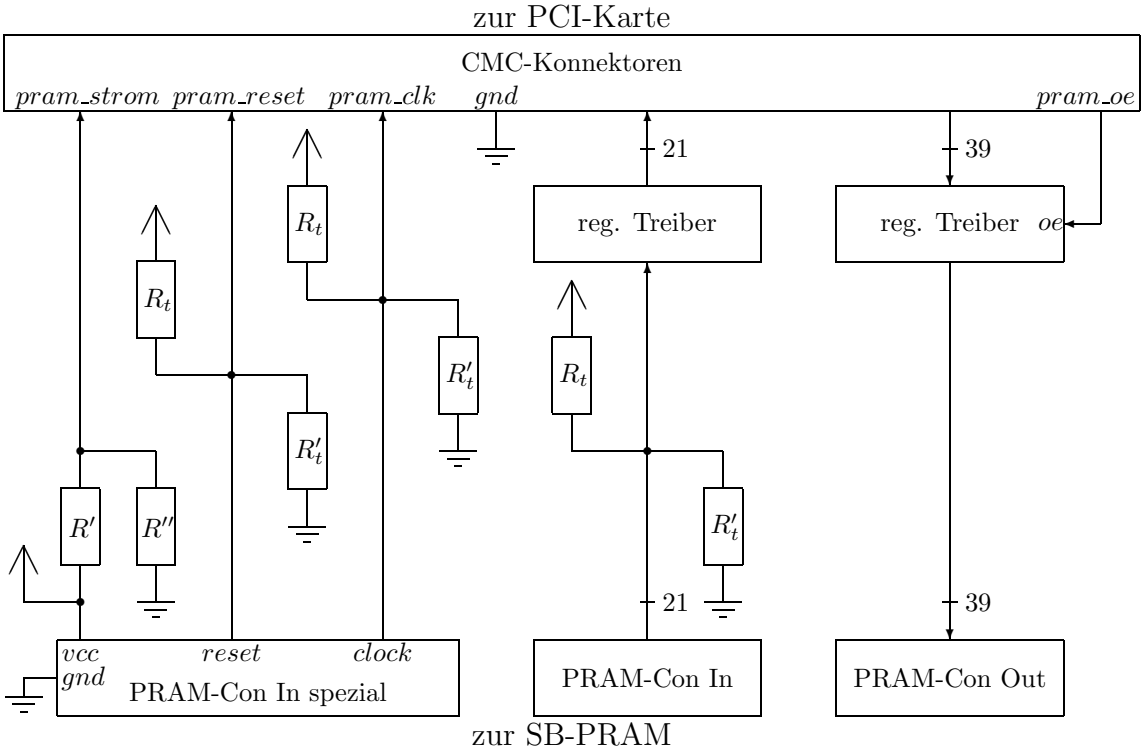


Abbildung 1.8: Blockschaltbild der Mezzaninkarte

einer Terminierung für diejenigen Signale, die von der SB-PRAM zur PCI-Karte getrieben werden. Abbildung 1.8 zeigt den Aufbau der Aufsatzkarte. Dabei bilden die Konnektoren *In* und *Out* das Interface zwischen PRAM-Prozessor und PRAM-Netzwerk; der Konnektor *In spezial* führt noch zusätzliche Signale von der SB-PRAM zur Mezzaninkarte.

Alle von der SB-PRAM ankommenden Signale sind durch je ein Paar von Widerständen R_t, R'_t terminiert. Die registrierten Treiber der Karte werden mit der PRAM-Clock betrieben, die ebenfalls an der Karte anliegt. Dadurch können alle Timing-Vorgaben der SB-PRAM eingehalten werden. Die PRAM-Clock wird auch zur PCI-Karte weitergegeben. Die Synchronisierung der PRAM-Daten mit der PCI-Clock erfolgt dann im DMA-Prozessor.

Damit durch die Terminierung keine Signale zur PRAM getrieben werden, während die SB-PRAM noch beim Hochfahren ist, wird die Aufsatzkarte wie eine Prozessorplatine der SB-PRAM komplett mit PRAM-Strom betrieben. Dazu wird auch der PRAM-Strom an die Karte angelegt. Die PCI-Karte selbst wird mit PCI-Strom betrieben; Aufsatzkarte und PCI-Karte besitzen aber nach wie vor eine gemeinsame *ground-plane*.

Zusätzlich wird noch das Reset-Signal der SB-PRAM auf die Karte geführt und zur PCI-Karte durchgereicht. Damit der DMA-Prozessor ein Modulo-Bit synchron zur SB-PRAM simulieren kann, wartet er auf einen Reset der SB-PRAM. Nach diesem Reset kann er mit der Simulation eines Modulo-Bits synchron zur SB-PRAM beginnen. Damit muß

die SB-PRAM vor dem Hochfahren an die PCI-Karte im PC angeschlossen sein, der PC muß bereits hochgefahren und der DMA-Prozessor konfiguriert sein.

Mit Hilfe der beiden Widerstände R' und R'' in Abbildung 1.8 kann festgestellt werden, ob PRAM-Strom anliegt. Liegt kein PRAM-Strom an, dann ist R'' ein pull-down-Widerstand und R' hat keine Auswirkung; es gilt also $pram\_strom = 0$. Liegt dagegen Strom von der SB-PRAM an, dann dient R' als pull-up-Widerstand und R'' kann ignoriert werden; deshalb gilt $pram\_strom = 1$. Das so erzeugte Signal $pram\_strom$ wird ebenfalls über die CMC-Konnektoren an die PCI-Karte gegeben.

Falls kein PRAM-Strom anliegt, ignoriert der DMA-Prozessor alle Eingabe-Signale von der SB-PRAM, also insbesondere das Reset-Signal. Die Ausführung von DMA-Programmen wird vollständig gestallt; der DMA-Prozessor kann keine DMA-Programme ausführen, wenn kein PRAM-Strom anliegt. Sobald PRAM-Strom anliegt, wartet der DMA-Prozessor darauf, daß das Reset-Signal der SB-PRAM inaktiv wird. Anschließend kann er das Modulo-Bit simulieren und DMA-Programme ausführen. Durch Auslesen eines Status-Registers des DMA-Prozessors kann der PC feststellen, ob PRAM-Strom anliegt und ein PRAM-Reset erfolgt ist.

1.4 FPGA-Architektur

Beim Design mit FPGAs können die klassischen Kosten- und Delay-Maße auf Gatter-Basis nach [KP95, MP95, MP00] nur bedingt verwendet werden. In diesem Abschnitt wird die Architektur der verwendeten FPGAs der Firma Xilinx [FPGA99] kurz erläutert, damit im späteren Verlauf der Arbeit vernünftige Kosten- und Delay-Betrachtungen erfolgen können.

Ein FPGA besteht aus einer $n \times n$-Matrix von CLBs (*Configurable Logical Blocks*). In jedem dieser CLBs befinden sich zwei sogenannte Funktionsgeneratoren f und g , die unabhängig voneinander beliebige 4-stellige binäre Funktionen $b : \{0,1\}^4 \to \{0,1\}$ berechnen können. Diese Funktionsgeneratoren sind als 4-Bit Lookup-Tables realisiert; sie sind also prinzipiell ein 16×1-ROM. Darüber hinaus gibt es in jedem CLB noch einen Funktionsgenerator h als 3-Bit Lookup-Table. Dieser Funktionsgenerator h kann die Ausgaben von f und g als Eingaben haben. Damit können in einem CLB einige 9-stellige binäre Funktionen berechnet werden (Abbildung 1.9).

Aus diesem Grund macht es wenig Sinn, bei den verwendeten FPGAs von Gatter-Delay und Gatter-Kosten zu sprechen. Alle vierstelligen binären Funktionen sind als Lookup-Table gleich schnell und gleich teuer. Kosten und Delay von Schaltkreisen kann man also allenfalls in Einheiten von Lookup-Tables der Größe 4 oder 3 berechnen. Es gibt in dieser FPGA-Architektur also keine „schnellen“ Gatter, wie in vielen anderen Technologien beispielsweise das NAND-Gatter. Insbesondere werden Inverter normalerweise in den Funktionsgeneratoren absorbiert und haben so Kosten und Delay 0.

Das gesamte Delay eines Schaltkreises setzt sich neben dem oben erwähnten logischen Anteil aus dem sogenannten Routing zusammen, sozusagen den Drähten zwischen den CLBs. Darauf hat der Hardware-Designer nur bedingt Einfluß; er setzt per Software lediglich Timing-Vorgaben. Anhand dieser Vorgaben vergibt die Software dann die ver-

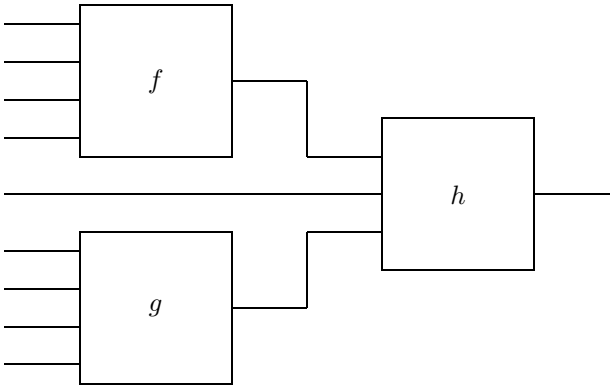


Abbildung 1.9: Aufbau eines CLBs aus den Funktionsgeneratoren f, g und h

schiedenen Routing-Ressourcen. Pfade mit dem längsten *logischen* Delay müssen also nicht unbedingt kritisch in Bezug auf ihr Gesamtdelay sein. Umgekehrt kann dieselbe Schaltung durch unterschiedliches Routing möglicherweise verschieden schnell geclockt werden. Bei den verwendeten FPGAs entfallen im Mittel etwa die Hälfte bis zwei Drittel des Gesamtdelays auf das Routing.

Unabhängig von den normalen Routing-Ressourcen besitzen die FPGAs ein *GSR*-Netz (*General Set/Reset*), das alle Flip-Flops entweder setzen oder löschen kann. Dieses GSR-Netz kann von einem beliebigen internen Signal oder einem Signal an einem Input-Pin getrieben werden. Damit bietet das GSR-Netz die Möglichkeit, auf ein Reset-Signal hin alle Flip-Flops ohne zusätzliche Kosten mit dem gewünschten Wert zu initialisieren. Der *GSR*-Reset ist asynchron; alle Flip-Flops werden unabhängig von der Clock und von *clock-enable* initialisiert. Das GSR-Netz initialisiert allerdings keine RAMs.

Die verwendeten FPGAs können in einem CLB auch 2 synchrone 16×1-RAMs simulieren. In ein synchrones RAM werden Daten bei aktivem Signal *write-enable* nur zu einer steigenden Flanke einer *write-clock* geschrieben analog zu einem Register.

Zusätzlich bieten die verwendeten FPGAs noch die Möglichkeit, in einem CLB ein synchrones 16×1-dual-port RAM zu simulieren. Dieses dual-port-RAM unterstützt einen Lesezugriff parallel zu einem Schreibzugriff, aber keine 2 parallelen Schreibzugriffe. Dazu besitzt das RAM zwei Adreßports $RA[3:0]$ und $WA[3:0]$, die unabhängig voneinander genutzt werden können. Das Datum an Adresse $\langle RA[3:0]\rangle$ wird dabei aus dem RAM ausgelesen, und das anliegende Datum wird an die Adresse $\langle WA[3:0]\rangle$ geschrieben, wenn das *write-enable WE* zu einer steigenden Flanke der *write-clock* aktiv ist. Dual-port RAMs kosten so viel wie zwei single-port RAMs. Sie sind beim Design von sogenannten Fifo-Buffern nützlich.

Kapitel 2

DMA-Prozessor

In diesem Kapitel wird der Aufbau des DMA-Prozessors vorgestellt. Der DMA-Prozessor verfügt über einen Instruktionssatz, mit dem der PC beispielsweise DMA-Kanäle reservieren und DMA-Programme schreiben und starten kann. Die Ausführung der DMA-Programme erfolgt dabei mit dem in Abschnitt 1.2.3 spezifizierten Algorithmus. Mit weiteren Befehlen kann der PC beispielsweise noch die Hashkonstante oder die Routing-Bit-Nummer der ersten Netzwerkstufe setzen oder ein Statuswort abfragen.

Zunächst beschreiben wir die Interfaces des DMA-Prozessors zum lokalen Bus, zur Sortiereinheit und zum SDRAM. Anschließend spezifizieren wir das User-Interface des DMA-Prozessors, die sichtbaren Register und den Instruktionssatz. Danach geben wir das Blockschaltbild des DMA-Prozessors an und betrachten anschließend die einzelnen Komponenten des DMA-Prozessors. Dabei beweisen wir, daß der Instruktionssatz genau wie spezifiziert interpretiert wird. Zuletzt zeigen wir noch, daß der DMA-Prozessor die DMA-Programme genau nach dem Algorithmus aus Kapitel 1.2 ausführt.

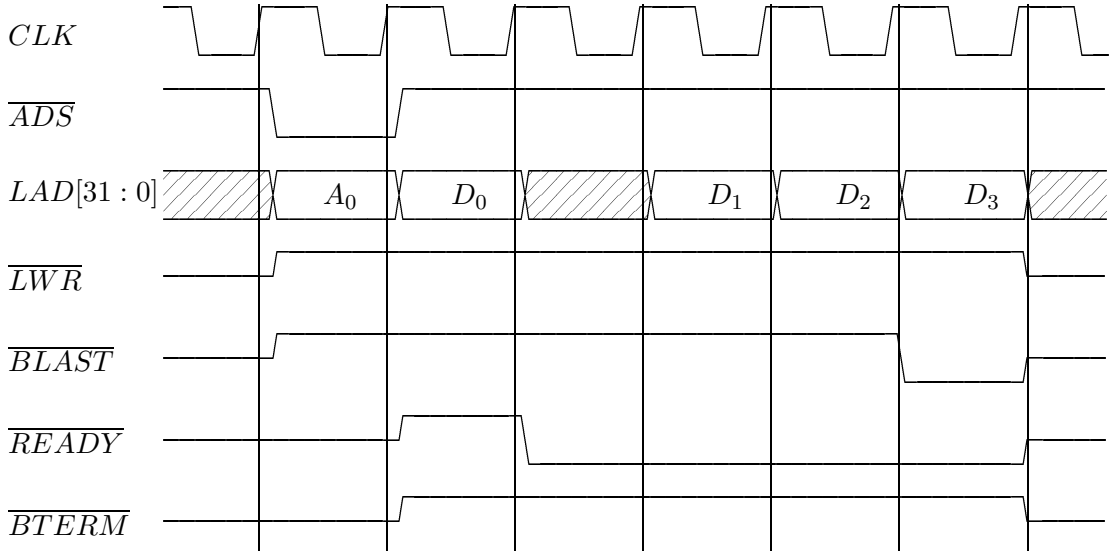
2.1 Interfaces

In diesem Abschnitt spezifizieren wir nacheinander die Signale der Interfaces des DMA-Prozessors zum lokalen Bus, zur Sortiereinheit und zum SDRAM. Zusätzlich geben wir die Datenübertragungsprotokolle für diese drei Interfaces an.

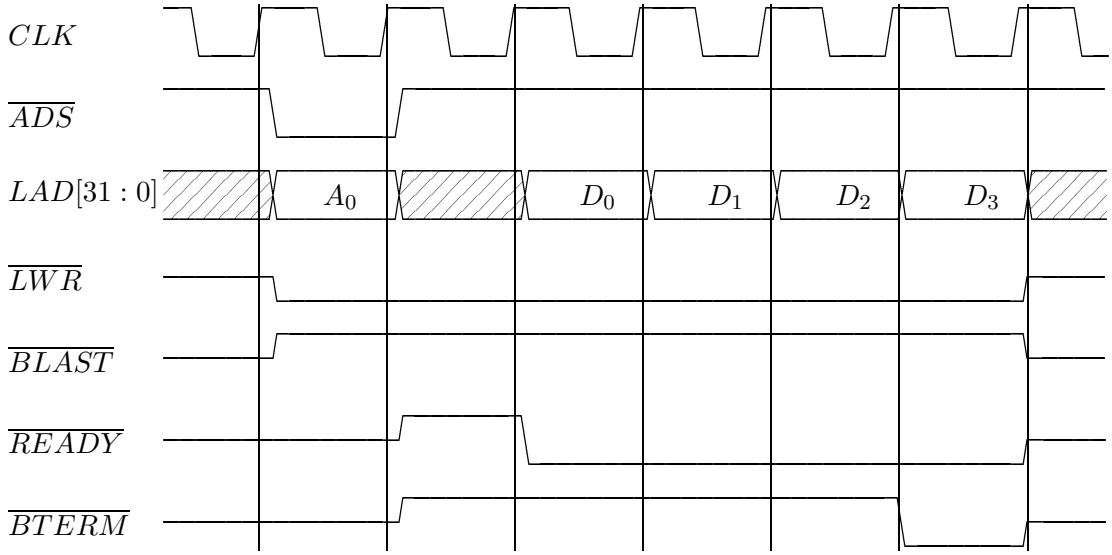
2.1.1 lokaler Bus

Das Busprotokoll des lokalen Busses ist durch die PCI-Bridge vorgegeben [PLX98]. Adressen und Daten liegen abwechselnd auf einem gemeinsamen 32-Bit-Bus LAD. In Tabelle 2.1 sind die Signale des lokalen Busses zusammengefaßt. Abbildung 2.1 zeigt typische idealisierte Timing-Diagramme bei einem Buszugriff.

Die PCI-Bridge aktiviert das Signal */ADS*, wenn sie eine gültige Byte-Adresse auf den Bus LAD gelegt hat. Da der DMA-Prozessor nur 32-Bit-Zugriffe unterstützt, die *aligned* erfolgen, werden die untersten beiden Bits $LAD[1:0]$ dieser Adresse immer ignoriert. Ein aktives Signal LWR gibt gleichzeitig zum Anlegen der Adresse an, ob die PCI-Bridge einen



Schreibzugriff, der von der PCI-Bridge mit $BLAST$ beendet wird



Lesezugriff, der vom DMA-Prozessor mit $BTERM$ beendet wird

Abbildung 2.1: Idealisiertes Timing von typischen Buszugriffen

In der Zeile $LAD[31:0]$ steht A_0 für die erste Adresse eines Bursts und D_i für das i-te Datum, das im Burst gelesen oder geschrieben wird. Nur die Kontrollsignale $\overline{READY}$ und $\overline{BTERM}$ sowie die Daten D_i bei Lesezugriffen werden vom DMA-Prozessor getrieben.

Signal	Art	Bedeutung
CLK	I	lokale Clock, getaktet mit dem PCI-Takt 33 MHz
$/RESET$	I	Reset-Signal für lokalen Bus
$/ADS$	I	*address strobe*; aktiv, wenn gültige Adresse anliegt
$LAD[31:0]$	I/O	*local address/data*; gemeinsamer Bus für Adressen und Daten. Eine Adresse auf diesem Bus ist immer eine *Byte-Adresse*.
LWR	I	*local write*; aktiv mit $/ADS$, wenn Daten geschrieben werden
$/BLAST$	I	*burst last*; aktiv, wenn das letzte Datum eines Bursts gelesen beziehungsweise geschrieben wird
$/READY$	O	aktiv, wenn das gelesene Datum auf dem Bus liegt oder wenn bei einem Schreibzugriff das Datum geschrieben worden ist. In diesem Fall kann im nächsten Takt das nächste Datum gelesen oder geschrieben werden.
$/BTERM$	O	*burst terminate*; aktiv, wenn das aktuelle Datum das letzte ist, das im Burst geschrieben oder gelesen werden kann. Die PCI-Bridge leitet für das folgende Datum einen neuen Adreßzyklus ein.

Tabelle 2.1: Interface zwischen DMA-Prozessor und lokalem Bus

Signale, die mit / beginnen, sind *active-low*. Die Buchstaben *I* und *O* in der Spalte *Art* stehen für Input- beziehungsweise Output-Signale aus Sicht des DMA-Prozessors.

Schreibzugriff durchführen will. In diesem Fall liegt das zu schreibende Datum genau einen Takt nach der Adresse auf dem Bus LAD. Durch Aktivieren von $/READY$ signalisiert der DMA-Prozessor, daß er das Datum geschrieben hat. Im nächsten Takt legt die PCI-Bridge dann ein Datum an, das an die nächste Adresse gespeichert werden soll. Die PCI-Bridge aktiviert das Signal $/BLAST$ zusammen mit dem letzten Datum, das sie schreiben will (oberer Teil von Abbildung 2.1). Der DMA-Prozessor kann seinerseits durch Aktivieren von $/BTERM$ anzeigen, daß er das nachfolgende Datum nicht mehr schreiben kann und einen neuen Adreßzyklus benötigt. Ganz analog erfolgen Lesezugriffe der PCI-Bridge auf den DMA-Prozessor (unterer Teil von Abbildung 2.1).

Um das Timing dieses Interfaces möglichst einfach zu halten, werden alle ausgehenden Daten, Adressen und Kontrollsignale in Outputregistern des FPGAs gespeichert; es werden also direkt Registerinhalte auf den lokalen Bus getrieben. Die ankommenden Daten und Kontrollsignale werden ebenfalls registriert, bevor sie vom DMA-Prozessor verwendet werden, bis auf das Signal $/BLAST$. Dieses Signal wird direkt von der Kontrollogik des DMA-Prozessors verwendet, damit im nächsten Takt bereits die richtigen Daten in die Outputregister geschrieben werden können.

Betrachtet man die Spezifikationen des verwendeten FPGAs [FPGA99] und der PCI-Bridge [PLX98], dann ist ersichtlich, daß die Timing-Vorgaben des lokalen Busses durch die Verwendung der erwähnten Register im FPGA ohne weitere Vorkehrungen eingehalten werden. Nur für $/BLAST$ wird die Setup- und Holdzeit der PCI-Bridge beachtet.

Signal	Art	Bedeutung
$SN_{in}[33:0]$	O	$SN_{in}[33:32]$ enthält den Modus $op[1:0]$ oder Operator $op[3:2]$ der Speicheranfrage gemäß Tabelle 2.3. In $SN_{in}[31:0]$ steht die gehashte PRAM-Adresse oder das Datum für die Speicheranfrage.
$validO$	O	aktiv, wenn an SN_{in} gültige Daten liegen
$VPlast$	O	aktiv, wenn die aktuelle Anfrage die letzte einer Teilrunde ist
HT	O	*half tact*; halbierter Takt zur Synchronisierung
lor	O	*lor*-Flag; markiert letzte Speicheranfrage eines Teilprogramms
$RBIT[3:0]$	O	$\langle RBIT[3:0]\rangle + 16$ ist die Routing-Bit-Nummer der ersten Netzwerkstufe der SB-PRAM
$fbusy$	I	*forward busy*; aktiv, wenn die Sortiereinheit keine Speicheranfragen mehr aufnahmen kann
$SN_{out}[32:0]$	I	$SN_{out}[31:0]$ enthält das Datum, das die PRAM als Antwort auf eine Speicheranfrage gesandt hat, und $SN_{out}32$ das Error-Bit
$validI$	I	aktiv, wenn an $SN_{out}[32:0]$ ein gültiges Datum liegt
$package$	I	*package*-Flag; markiert letzte Antwort eines Teilprogramms
$FifoFull$	O	aktiv, wenn der DMA-Prozessor keine Daten vom Sortierer mehr aufnehmen kann

Tabelle 2.2: Interface zwischen DMA-Prozessor und Sortiereinheit

Die Buchstaben *I* und *O* in der Spalte *Art* stehen für Input- beziehungsweise Output-Signale aus Sicht des DMA-Prozessors.

2.1.2 Sortiereinheit

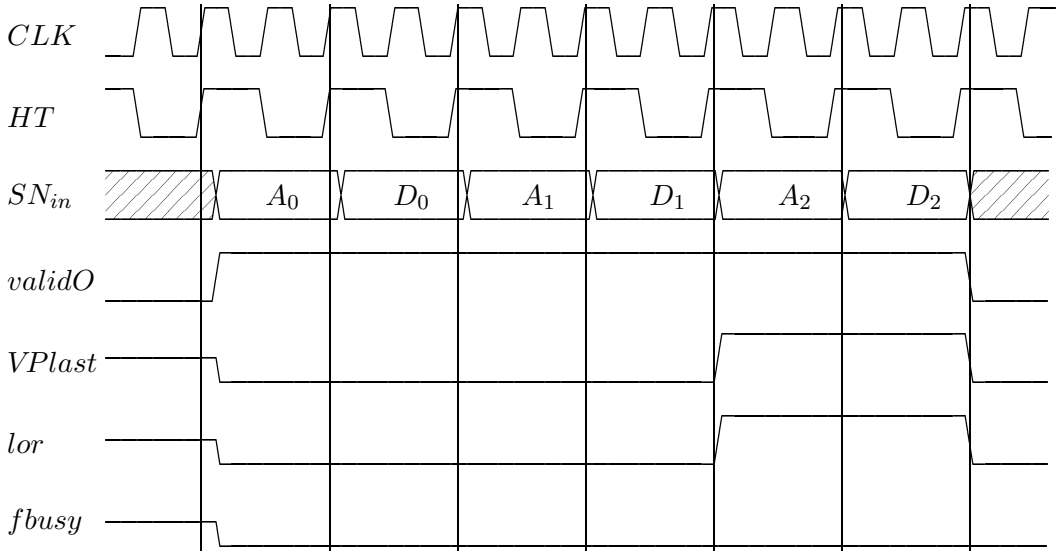
Die Sortiereinheit und der DMA-Prozessor sind in *einem* FPGA untergebracht und nutzen die Signale CLK und $RESET$ vom lokalen Bus gemeinsam. Das Busprotokoll entspricht dabei im wesentlichen dem Busprotokoll zwischen einem PRAM-Prozessor und einem Sortierknoten in [Gö96]. In Tabelle 2.2 sind die Signale des Interfaces zwischen DMA-Prozessor und Sortiereinheit zusammengefaßt.

Abbildung 2.2 verdeutlicht dieses Protokoll mit Hilfe eines Timing-Diagramms. Adreß- und Datenteil der Speicheranfragen werden vom DMA-Prozessor dabei nacheinander über denselben Bus SN_{in} bei aktivem $validO$ verschickt. Zunächst liegt 2 Takte lang die

op[1:0]	Modus
00	LOAD
01	STORE
10	MP
11	SYNC

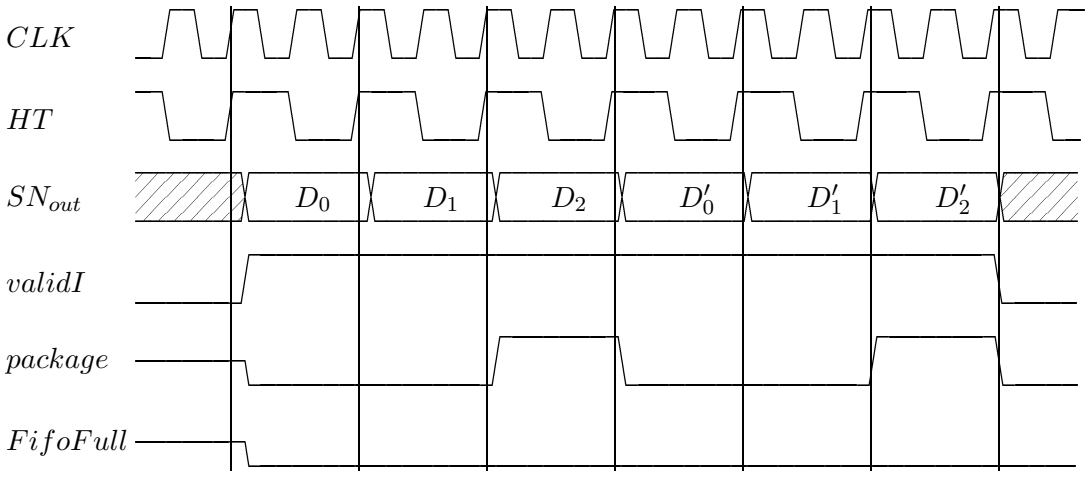
op[3:2]	Operator
00	and
01	or
10	max
11	add

Tabelle 2.3: Kodierung des Zugriffsmodus und -operators



Senden von Speicheranfragen an die Sortiereinheit (=Hinweg)

A_i steht für den Modus und die Adresse der i-ten Speicheranfrage einer Teilrunde und D_i für den zugehörigen Operator und das zugehörige Datum. Die Sortiereinheit treibt nur das Signal $fbusy$.



Empfang von Antworten der Sortiereinheit (=Rückweg)

D_i steht für das Antwortdatum auf die i-te Speicheranfrage einer Teilrunde und D'_i steht analog für ein Antwortdatum einer anderen Teilrunde. Der DMA-Prozessor treibt nur das Signal $FifoFull$.

Abbildung 2.2: Timing zwischen DMA-Prozessor und Sortiereinheit

Signal	Art	Bedeutung
CKE	O	*clock-enable*
$DQ[31:0]$	I/O	Data Input/Output
$A[11:0]$	O	*address*; hier liegt eine Hälfte der Adresse an.
$BA[1:0]$	O	*bank address*; selektiert eine der RAM-Bänke
$/DQM[3:0]$	O	Byte-enables für die 4 Bytes aus $DQ[31:0]$
$/CS$, $/RAS$, $/CAS$, $/WE$	O	Kontrollsignale; Bedeutung unten

Kodierung der wichtigsten Kommandos für das SDRAM				
Kommando	$/CS$	$/RAS$	$/CAS$	$/WE$
`nop` (no operation)	0	1	1	1
`act` (set bank & activate row)	0	0	1	1
`read` (set bank & start read burst)	0	1	0	1
`write` (set bank & start write burst)	0	1	0	0
`pre` (deactivate row)	0	0	1	0

Tabelle 2.4: Interface zwischen DMA-Prozessor und SDRAM

Signale, die mit / beginnen, sind *active-low*. Die Buchstaben *I* und *O* in der Spalte *Art* stehen für Input- beziehungsweise Output-Signale aus Sicht des DMA-Prozessors.

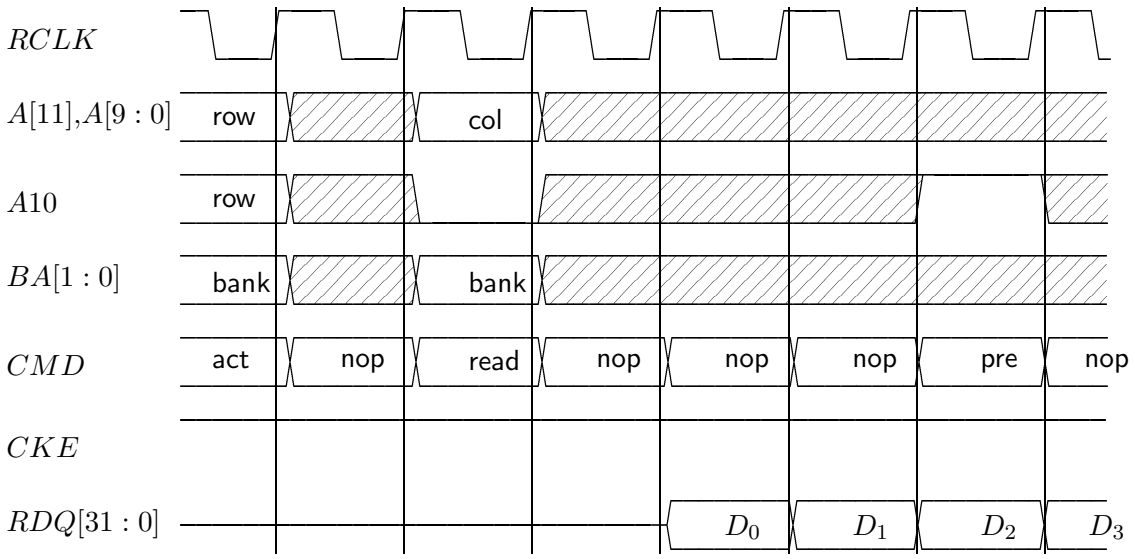
gehashte PRAM-Adresse und der Modus des Zugriffs an, anschließend 2 Takte lang das Datum und der Operator. Signalisiert die Sortiereinheit durch Aktivieren von *fbusy*, daß sie keine Speicheranfragen mehr aufnehmen kann, dann nimmt der DMA-Prozessor das Signal *validO* zurück, sendet also keine Speicheranfragen mehr, bis die Sortiereinheit das Signal *fbusy* wieder deaktiviert. Über den Bus $RBIT[3:0]$ gibt der DMA-Prozessor die Routing-Bit-Nummer der ersten Netzwerkstufe an die Sortiereinheit weiter.

Die Antworten der SB-PRAM legt die Sortiereinheit jeweils 2 Takte lang bei aktivem *validI* an SN_{out} an. Signalisiert der DMA-Prozessor durch Aktivieren von *FifoFull*, daß er keine weiteren Antworten mehr aufnehmen kann, dann nimmt die Sortiereinheit das Signal *validI* zurück, sendet also keine Antworten mehr, bis der DMA-Prozessor *FifoFull* wieder deaktiviert.

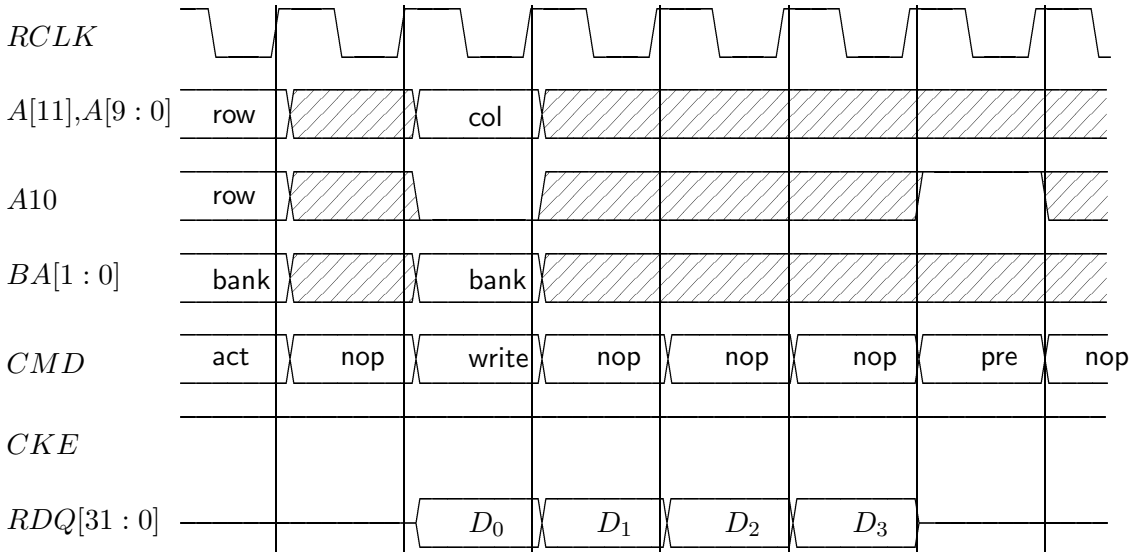
Da die Sortiereinheit und der DMA-Prozessor in *einem* FPGA untergebracht sind, ist eine gesonderte Betrachtung des Timings für diese Schnittstelle überflüssig.

2.1.3 SDRAM

Nach Abbildung 1.7 auf Seite 30 hängt das SDRAM direkt am FPGA und nicht am lokalen Bus. Die Tabelle 2.4 listet die Signale des Interfaces zwischen DMA-Prozessor und SDRAM auf und Abbildung 2.3 zeigt die zugehörigen Timing-Diagramme. Das verwendete SDRAM ist ein 32×2^{22}-RAM, enthält also 16 MByte Daten beziehungsweise 4 MWord Daten, wenn ein *Word* ein 32-Bit-Datum ist, also aus 4 Bytes besteht. Adressen, die am SDRAM anliegen, sind also im Gegensatz zu den Adressen auf dem lokalen Bus LAD



Lesezugriff auf das SDRAM; Burstlänge 4



Schreibzugriff auf das SDRAM; Burstlänge 4

Abbildung 2.3: Idealisiertes Timing für Lese- und Schreibzugriffe auf das SDRAM

In der Zeile CMD ist angegeben, welches Kommando jeweils an das SDRAM angelegt wird. Die Kodierung der Kommandos kann Tabelle 2.4 entnommen werden. In der Zeile RDQ steht D_i für das i-te Datum, das im Burst gelesen oder geschrieben wird.

immer Word-Adressen.

DMA-Prozessor und SDRAM arbeiten mit einer gemeinsamen Clock, der *RCLK*. Das SDRAM ist in 4 Bänken zu 4 MB organisiert, die über $BA[1:0]$ ausgewählt werden, und unterstützt Burstzugriffe. Bevor das SDRAM nach einem Reset genutzt werden kann, muß es über die Kontrollsignale und Adreßleitungen programmiert werden. Wir haben dabei zur Programmierung einen Modus ausgewählt, in dem das SDRAM lesende und schreibende Bursts bis zu einer Länge von 8 Daten ausführen kann, da diese Burstlänge für den DMA-Prozessor zur Ausführung von Teilprogrammen ausreichend ist.

Die Abbildung 2.3 zeigt typische Timingdiagramme für Lese- und Schreibzugriffe auf das SDRAM im Burst. Nachdem der DMA-Prozessor die obere Hälfte *row* einer Adresse angelegt hat und gleichzeitig beim Kommando *act* mit $BA[1:0]$ eine Bank ausgewählt hat, kann er 2 Takte später beim Kommando *read* beziehungsweise *write* den unteren Teil *col* einer Adresse anlegen. Bei einem Schreibzugriff wird gleichzeitig das an diese Adresse zu schreibende Datum an *RDQ* angelegt; bei einem Lesezugriff treibt das SDRAM das gelesene Datum 2 Takte später auf den Bus *RDQ*. Nach dem Lesen beziehungsweise Schreiben der letzten Adresse im Burst wird mit dem Kommando *pre* das SDRAM wieder freigegeben.

Um das Timing für dieses Interface möglichst einfach zu halten, werden alle Daten, Adressen und Kontrollsignale im FPGA registriert, bevor sie auf die PCI-Karte getrieben werden; auch alle auf *RDQ* ankommenden Daten werden registriert, bevor sie im FPGA verwendet werden. Mit dieser Vorgehensweise ist das Interface zum SDRAM von ATMedia GmbH fehlerfrei mit 100 MHz betrieben worden [Ja00]. Damit müssen für die von uns angestrebte Taktfrequenz von 50 MHz für dieses Interface keine weiteren Timing-Betrachtungen erfolgen.

2.2 User-Interface

Der PC greift über den PCI-Bus auf den DMA-Prozessor und das lokale RAM der PCI-Karte zu. Dazu ist der PCI-Karte ein lokaler Adreßraum von 32MB zugeordnet. Gemäß Abbildung 2.4 kann über die untere Hälfte dieses Adreßraums auf die 16 MB SDRAM der Karte zugegriffen werden; die oberen 16 MB sind für diverse Befehle des DMA-Prozessors reserviert. Über diese obere Hälfte des Adreßraums werden also nicht direkt sichtbare Register adressiert, sondern *ein Zugriff auf eine Adresse ist ein Befehl des DMA-Prozessors*, der seinerseits ein oder mehrere sichtbare Register des DMA-Prozessors schreibt. Die Befehle des DMA-Prozessors sind also in Adressen von Zugriffen auf den lokalen Bus kodiert. Dabei kann ein Befehl grundsätzlich durch einen schreibenden *oder lesenden* Zugriff auf eine Adresse ausgelöst werden. Die Ausführung dieser Befehle kann noch von weiteren sichtbaren Status-Registern abhängen. Insgesamt können also durch einen einzigen Befehl, also den lesenden oder schreibenden Zugriff auf eine Adresse, mehrere sichtbare Register des DMA-Prozessors in Abhängigkeit von Status-Registern geschrieben werden. Nach der Einführung der sichbaren Register des DMA-Prozessors in Abschnitt 2.3 erfolgt im darauf folgenden Abschnitt 2.4 eine Spezifikation des Instruktionssatzes.

Für einen typischen Lesezugriff auf die SB-PRAM reserviert sich der PC zunächst mit

<table>
<tr><th>Adresse</th><th colspan="2">Inhalt</th></tr>
<tr><td>0000000h
⋮
00FFFFCh</td><td>Speicher für DMA-Kanal 0</td><td rowspan="4">lokales RAM der PCI-Karte</td></tr>
<tr><td>0100000h
⋮
01FFFFCh</td><td>Speicher für DMA-Kanal 1</td></tr>
<tr><td>⋮</td><td>⋮</td></tr>
<tr><td>0F00000h
⋮
0FFFFFCh</td><td>Speicher für DMA-Kanal 15</td></tr>
<tr><td>1000000h
⋮
1FFFFFCh</td><td colspan="2">Befehle des DMA-Prozessors</td></tr>
</table>

Abbildung 2.4: Memory-Map des DMA-Prozessors

einem Befehl einen Kanal i. Anschließend schreibt er das DMA-Programm für diesen Kanal i und startet das DMA-Programm. Der PC fragt den Status für diesen Kanal ab und stellt fest, ob das Programm vollständig ausgeführt worden ist. Dann stehen die gelesenen Daten im lokalen RAM für Kanal i und können vom PC ausgelesen werden. Der PC kann jetzt ein neues DMA-Programm auf Kanal i schreiben und starten oder den Kanal mit einem Befehl wieder freigeben.

Für einen Schreibzugriff auf die SB-PRAM schreibt der PC die an die PRAM zu sendenden Daten in das lokale RAM des Kanals i, den er sich zuvor mit dem entsprechenden Befehl reserviert hat. Das Schreiben und Starten des DMA-Programms erfolgt analog zu Lesezugriffen; wenn die Statusabfrage ergibt, daß das DMA-Programm vollständig ausgeführt worden ist, dann wurden alle gewünschten Daten in den Speicher der SB-PRAM geschrieben.

Damit haben wir bereits die wichtigsten Befehle des DMA-Prozessors eingeführt. Die Befehle zum Schreiben eines DMA-Programms sind dabei als *Schreibzugriffe* auf den DMA-Prozessor implementiert. Der Befehl zum Reservieren eines DMA-Kanals gibt die Nummer des reservierten Kanals zurück oder signalisiert in einem zusätzlichen Bit, daß kein DMA-Kanal mehr frei ist. Deshalb ist dieser Befehl als *Lesezugriff* realisiert. *Damit wird durch einen Lesezugriff auf den DMA-Prozessor ein Schreibzugriff im DMA-Prozessor ausgelöst*, denn der reservierte Kanal i wird natürlich in einem Scheduling-Register $SR[i]$ des DMA-Prozessors als reserviert markiert.

Aus Bemerkung 1.1.3 auf Seite 6 über die Funktionsweise der SB-PRAM und dem

Algorithmus zur Ausführung von DMA-Programmen aus Abschnitt 1.2 ergeben sich die meisten sichtbaren Register des DMA-Prozessors, die in der folgenden Aufzählung aufgelistet sind. Eine genaue Übersicht wird im nächsten Abschnitt in Tabelle 2.5 gegeben.

1. Die Register HC, $RBIT$ und $MODC$ speichern die Hashkonstante der SB-PRAM, die Routing-Bit-Nummer der ersten Netzwerkstufe und die Zahl der Teilrunden, aus denen eine Netzwerkrunde besteht (siehe Bemerkung 1.1.3).

2. Im Scheduling-Register SR wird gespeichert, welche der 16 DMA-Kanäle vergeben beziehungsweise frei sind.

3. In einem Registerfile wird in den Teilregistern $PADR$, $START$ und AUX das DMA-Programm für jeden DMA-Kanal gespeichert. Die genaue Kodierung des DMA-Programms in diesen 3 Registern kann Abschnitt 2.3 entnommen werden.

4. In den 4 Fifo-Buffern $Fifo_{mod,ea}$ für $mod, ea \in \{0, 1\}$ werden die DMA-Kanäle gespeichert, auf denen gerade DMA-Programme ausgeführt werden.

5. In den Status-Registern $STAT$ und $ERROR$ wird gespeichert, ob gerade ein DMA-Programm auf einem Kanal läuft und ob bei der Ausführung des DMA-Programms ein Fehler aufgetreten ist.

6. In einem weiteren Status-Register $TERM$ wird gespeichert, ob ein DMA-Programm auf einem Kanal mit einem speziellen Befehl beendet wurde.

2.3 Sichtbare Register des DMA-Prozessors

Im letzten Abschnitt haben wir die sichtbaren Register des DMA-Prozessors bereits kurz eingeführt. Im diesem Abschnitt zeigen wir noch, wie ein DMA-Programm in den 3 Registern $PADR$, $START$ und AUX kodiert werden kann, und geben in Tabelle 2.5 eine genaue Übersicht über die sichtbaren Register.

Bei den Ausführungen zur Semantik von DMA-Programmen in Kapitel 1.2.1 sind wir davon ausgegangen, daß das lokale RAM ein 32×2^K-RAM ist. Wir verwenden auf unserer PCI-Karte 16 MB SDRAM, also ein 32×2^{22}-RAM. Nach den Bemerkungen zum lokalen Bus der PCI-Karte aus Kapitel 2.1.1 ist der PCI-Bus aber *Byte-adressierbar*. Da die Zugriffe des PCs auf das SDRAM auch über das FPGA und damit über den lokalen Bus erfolgen, ist folglich das SDRAM für den PC Byte-adressierbar–es entspricht also einem 8×2^{24}-RAM. Im FPGA erfolgt eine Umrechnung einer solchen Byte-Adresse $badr$ vom lokalen Bus LAD in die Word-Adresse $wadr$ für das SDRAM; dabei gilt $badr = 4 \cdot wadr$ beziehungsweise $wadr = badr$ DIV 4.

Wenn also ein DMA-Programm gemäß seiner Semantik eine Adresse adr im lokalen RAM schreibt, dann kann der PC das Datum an dieser Adresse lesen, indem er auf die Byte-Adresse $4 \cdot adr$ im SDRAM zugreift. Beim Schreiben eines DMA-Programms schreibt der PC auch die Start- und Endadresse im lokalen RAM, $sadr$ und $eadr$. Dabei handelt es sich in Einklang mit obigen Bemerkungen auch um die Byte-Endadresse, also $4 \cdot sadr$ und $4 \cdot eadr$. Der PC benutzt also ausschließlich Byte-Adressen.

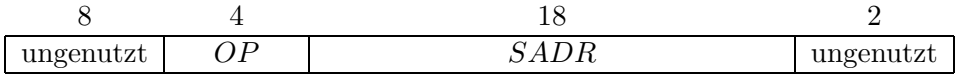


Abbildung 2.5: Format des Parameters *START* in DMA-Programmen

$\langle SADR[17:0]\rangle = relsadr = sadr \bmod 2^{18}$ ist die relative Startadresse im RAM des Kanals. Die Kodierung von Operator und Modus in $OP[3:0]$ kann Tabelle 2.3 entnommen werden.

Der DMA-Prozessor enthält ein Registerfile, in dem die DMA-Programme für die 16 Kanäle gespeichert sind. Ein einzelnes DMA-Programm $P = (op, padr, sadr, eadr, data)$ mit gehashter PRAM-Startadresse $hpadr = padr \odot hc$ läßt sich dabei in 3 32-Bit-Worten kodieren. Die Binärdarstellung der gehashten PRAM-Startadresse $\mathrm{bin}_{32}(hpadr)$ wird in einem 32-Bit-Register $PADR$ des DMA-Prozessors abgespeichert.

Ein zweites 32-Bit-Register *START* kann die Parameter *sadr* und *op* zusammen speichern. Der lokale Speicher der PCI-Karte hat eine Größe von 16 MB und wird für die 16 DMA-Kanäle in 16 Speicherseiten der Größe 1 MB aufgeteilt. Dabei ist für $0 \leq c \leq 15$ Kanal c genau der Speicherbereich von der Byte-Adresse $c \cdot 2^{20}$ bis einschließlich zur Byte-Adresse $(c+1) \cdot 2^{20} - 1$ zugeordnet (vergleiche Abbildung 2.4). Damit läßt sich die relative *Byte-Startadresse* im Speicherbereich des Kanals in 20 Bits kodieren. Da die relative Startadresse $relsadr$ eines DMA-Programms eine Word-Adresse ist, ist die zugehörige relative Byte-Startadresse durch 4 teilbar; die beiden untersten Bits können also ignoriert werden. Also werden lediglich 18 Bits gebraucht, um die *relative* Startadresse $relsadr = sadr \bmod 2^{18}$ im RAM des Kanals c zu kodieren. Die absolute Startadresse im lokalen RAM ergibt sich dann durch $sadr = c \cdot 2^{18} + relsadr$.

Die relative Startadresse $relsadr$ sowie Modus und Operator *op* sind gemäß Abbildung 2.5 in einem 32-Bit-Wort kodiert. Dieses Wort wird im Register *START* gespeichert. Genaugenommen reicht bereits ein 22-Bit Register, da die oberen 8 Bits des Parameters ungenutzt sind und die untersten 2 Bits ebenfalls ignoriert werden. In $START[23:20]$ ist damit Modus und Operator kodiert; die relative Startadresse des DMA-Programms in lokalen RAM ist $\langle START[19:2]\rangle$.

Der Parameter *data* ist nur bei Einzelanweisungen vom Wert $\sharp$ verschieden, und in diesem Fall gilt nach Definition 1.2.1 gerade $eadr = sadr$. Der Parameter *eadr* hat bei Einzelanweisungen also keine Bedeutung. Deshalb genügt *ein* 32-Bit-Register *AUX*, um *eadr* und *data* zu speichern, wenn bekannt ist, ob P eine Einzelanweisung ist. Analog zur Vorgehensweise bei der Startadresse *sadr* im lokalen RAM speichern wir auch nur die relative Byte-Endadresse statt der absoluten Word-Endadresse *eadr*. Das *AUX*-Register speichert also entweder den Parameter *data* einer Einzelanweisung oder die relative Byte-Endadresse einer Standardanweisung. Für das Register *AUX* gilt damit:

$$AUX[31:0] = \begin{cases} \mathrm{bin}_{32}(4 \cdot releadr) & \text{falls } data = \sharp \\ \mathrm{bin}_{32}(data) & \text{falls } data \in B_{32} \end{cases}$$

Also ist $\langle AUX[19:2]\rangle$ die relative Word-Endadresse einer Standardanweisung. Die Re-

Register	Beschreibung
Register für jeden der 16 DMA-Kanäle	
SR	Scheduling-Register. Hier gilt die Invariante $SR[i] = 1$ genau dann, wenn DMA-Kanal i zur Zeit vergeben ist.
$PADR[31:0]$	gehashte PRAM-Startadresse *hpadr* von DMA-Programm P
$START[23:2]$	Das $START$-Register ist gemäß Abbildung 2.5 in die beiden Teile $OP[3:0] = START[23:20]$ und $SADR[17:0] = START[19:2]$ aufgeteilt. In $OP[3:0]$ steht der Zugriffsmodus und -operator von P, $\langle SADR[17:0]\rangle = relsadr$ ist die relative Startadresse im lokalen RAM.
$AUX[31:0]$	Auxiliary Data für DMA-Programm P. Wenn P eine Einzelanweisung ist, dann enthält dieses Register das zugehörige Datum, also $\langle AUX[31:0]\rangle = data$, ansonsten steht hier die relative Endadresse, es gilt also $\langle AUX[19:2]\rangle = releadr$.
$STAT$	Status-Register; es gilt $STAT[i] = 1$ genau dann, wenn auf Kanal i ein DMA-Programm läuft.
$ERROR$	Error-Register; $ERROR[i] = 1$ genau dann, wenn beim Ausführen des DMA-Programms auf Kanal i ein Fehler aufgetreten ist.
$TERM$	Terminierungs-Register; $TERM[i]$ wird durch einen Befehl *enddma* für Kanal i auf 1 gesetzt. Ist bei der Ausführung eines Programms auf Kanal i $TERM[i] = 1$, dann wird die Ausführung des DMA-Programms beendet.
allgemeine Register	
$Fifo_{mod,ea}[3:0]$	Fifo-Buffer; enthält für $ea = 1$ genau die DMA-Kanäle, auf denen eine Einzelanweisung ausgeführt wird, und für $mod = 1$ genau die DMA-Kanäle, deren DMA-Programme bei aktivem Modulo-Bit ausgeführt werden.
$HC[31:0]$	Hashkonstante
$RBIT[3:0]$	Routing-Bit; der DMA-Prozessor verwendet $\langle RBIT[3:0]\rangle + 16$ als Routing-Bit-Nummer. Nach Kapitel 1.1 wird das Routing-Bit einer Adresse $\langle A[31:0]\rangle$ immer aus den Bits $A[31:16]$ gewählt.
$MODC[1:0]$	Modulo-Count; eine Netzwerkrunde der SB-PRAM besteht aus $\langle MODC[1:0]\rangle + 1$ Teilrunden. Nach Abschnitt 1.1 kann eine Netzwerkrunde nämlich aus $1-4$ Teilrunden bestehen.

Tabelle 2.5: Sichtbare Register des DMA-Prozessors

gister *PADR*, *START* und *AUX* enthalten also zusammen mit der Kanalnummer c des DMA-Programms P alle Informationen, die P auch enthält, bis auf die Frage, ob P eine Einzelanweisung ist.

Bei Verwendung des vorgestellten Scheduling-Mechanismus wird diese Information beim Start eines DMA-Programms benötigt. Dann wird nämlich der DMA-Kanal in die zugehörige Fifo des DMA-Schedulers geschrieben. Beim späteren Ausführen eines DMA-Programms P wird der zugehörige Kanal aus einer Fifo des DMA-Schedulers gelesen. Ob P eine Einzelanweisung ist, kann dann schon daran erkannt werden, aus welcher Fifo der Kanal von P gelesen wurde. Also benötigen wir gar kein eigenes Register, um zu speichern, ob P eine Einzelanweisung ist. Es genügt, wenn der PC beim Start eines DMA-Programms dem DMA-Prozessor mitteilt, ob das auszuführende Programm eine Einzelanweisung ist. Dann kann der DMA-Prozessor $releadr := relsadr$ und $data := \langle AUX[31:0] \rangle$ ergänzen, sonst ergänzt er $releadr := \langle AUX[19:2] \rangle$ und $data := \sharp$.

Genauso kann die Information, ob P bei aktivem Modulo-Bit ausgeführt werden soll, beim Start des DMA-Programms an den DMA-Prozessor übergeben werden und benötigt kein eigenes Register. Damit kann der DMA-Prozessor den Kanal in die zugehörige Fifo des DMA-Schedulers schreiben. Beim Auslesen eines Kanals aus dem DMA-Scheduler erkennt der DMA-Prozessor dann, ob das zugehörige Programm eine Einzelanweisung ist beziehungsweise ob es bei aktivem Modulo-Bit ausgeführt werden soll.

Damit werden also zum Schreiben eines DMA-Programms die 3 Register *PADR*, *START* und *AUX* gesetzt. Der anschließende Befehl zum Start des DMA-Programms gibt zusätzlich an, ob das Programm bei aktivem Modulo-Bit ausgeführt wird und ob es eine Einzelanweisung ist.

Die Register des DMA-Prozessor sind in Tabelle 2.5 zusammengefaßt. Zur Erfüllung der SB-PRAM-spezifischen Anforderungen aus Bemerkung 1.1.3 ist der DMA-Prozessor mit den 3 allgemeinen Registern HC, $RBIT$ und $MODC$ ausgestattet. Bis auf das Register $TERM$ wurden die Register für DMA-Kanäle bereits eingeführt. Das Register $TERM$ wird für eine Instruktion benötigt, die die Ausführung eines DMA-Programms vorzeitig beendet.

2.4 Instruktionssatz

Die Instruktionen des DMA-Prozessors sind in den Adressen von Lese- und Schreibzugriffen auf dem lokalen Bus der PCI-Karte kodiert. Beim Schreiben des DMA-Programms ergibt sich die Schwierigkeit, daß 32-Bit-Daten–beispielsweise die PRAM-Startadresse–für einen der 16 Kanäle gesetzt werden. Die Kodierung der Kanalnummer erfordert wieder mindestens 4 Bits, also werden insgesamt 36 Bit Daten übertragen. Der PCI-Bus ist aber nur 32 Bit breit.

Die einfachste Lösung dieses Problems ergibt sich, wenn man die 4 Bits der Kanalnummer ebenfalls in der *Adresse* des Schreibzugriffs kodiert. So können in einem Schreibzugriff auf dem PCI-Bus mehr als 32 Bit „Daten“ übertragen werden. Bei allen Befehlen, die sich auf einen DMA-Kanal beziehen, ist die Kanalnummer deshalb in denselben Adreßbits kodiert. Damit sind sowohl die Instruktion als auch einige Parameter in der Adresse eines

8	4	12	1	1	4	2
$0^7 1$	CMD	ungenutzt	MOD	EA	CH	00

Abbildung 2.6: Befehlsformat des DMA-Prozessors mit Parametern

Alle Adressen liegen im lokalen Adreßraum der PCI-Karte und sind Vielfache von 4. MOD, EA und CH sind die Parameter eines Befehls CMD, der nach Tabelle 2.6 kodiert ist.

Zugriffs auf dem lokalen Bus kodiert.

Reserviert der PC mit einem Befehl einen DMA-Kanal, dann gibt der DMA-Prozessor die Nummer des reservierten Kanals zurück. Durch einen *Lesezugriff* auf dem PCI-Bus wird also im DMA-Prozessor ein *Schreibzugriff* ausgelöst, denn der reservierte Kanal wird im *Scheduling-Register SR* als vergeben vermerkt. Darüber hinaus sind auch andere Schreibzugriffe im DMA-Prozessors als Lesezugriffe auf dem PCI-Bus realisiert, so zum Beispiel der Befehl zum Starten eines DMA-Programms. Der Vorteil dieser Lesezugriffe ist, daß der PC anhand des zurückgegebenen Statusworts feststellen kann, ob der Befehl erfolgreich ausgeführt worden ist. Er muß den Bus nicht noch für eine weitere Statusabfrage belegen und auf das Antwortdatum warten. Die einzigen „echten“ Schreibbefehle sind die 3 Befehle zum Schreiben eines DMA-Programms und der Befehl zum Schreiben der Hashkonstante, da die Parameter dieser Befehle nicht in den wenigen freien Adreßbits kodiert werden können.

Die Kodierung der Befehle und ihrer Parameter in der Adresse erfolgt gemäß Abbildung 2.6. Die Bits 31-25 einer Adresse enthalten dabei alle eine 0, da der DMA-Prozessor zur Adressierung von 32 MB Speicher nur einen 25-Bit-Adreßraum hat. Bei allen Befehlen steht in Bit 24 der Adresse eine 1. Zugriffe auf Adressen, die in Bit 24 eine 0 enthalten, werden als Zugriffe auf das SDRAM interpretiert. Damit sind mit den Bits 0 bis 23 der Adresse 16 MB SDRAM adressierbar.

Die untersten beiden Bits einer Adresse auf dem lokalen Bus sind immer 0, da der PCI-Bus Byte-adressierbar ist, 32-Bit-Zugriffe aber nur aligned erfolgen. Wenn ein Befehl sich auf einen bestimmten DMA-Kanal bezieht, steht die Binärdarstellung der Kanalnummer in *CH*. Nur beim Starten eines DMA-Programms haben auch die Bits 7 und 6 eine Bedeutung; sie setzen das Modulo-Bit für das Programm (MOD) und legen fest, ob es sich um eine Einzelanweisung handelt (EA=1).

In *CMD* sind die Befehle gemäß Tabelle 2.6 kodiert; Bit 24 enthält bei Befehlen des DMA-Prozessor immer eine 1. Alle übrigen Bits können beliebig gewählt werden; sie werden einfach ignoriert.

Der Instruktionssatz des DMA-Prozessors umfaßt die bereits erwähnten Befehle zum Reservieren und Freigeben von DMA-Kanälen, *getdma* und *freedma*, sowie die Befehle *startdma* zum Starten von DMA-Programmen und *status* zur Statusabfrage. Die 3 Teile eines DMA-Programms werden mit *setpadr*, *setstart* und *setaux* geschrieben. Darüber hinaus kann die Hashkonstante im DMA-Prozessor mit *whc* gesetzt und mit *rhc* ausgelesen werden sowie die Routing-Bit-Nummer und die Zahl der Teilrunden einer Netzwerkrunde

CMD[3:0]	Befehl	Bedingung	Effekt
Lesezugriffe			
0000	getid		$ret = \text{bin}_{32}(\underbrace{1398162766}_{\text{='SVEN' ASCII-kodiert}})$
0010	status		$ret = STATUS[31:0]$
0011	rhc		$ret = HC[31:0]$
0100	getdma	$\langle HC[31:0]\rangle \neq 0$ $SR[15:0] \neq 1^{16}$	$SR[get] := 1$, $STAT[get] := 0$ $ret = STATUS[31:0]$
0101	freedma	$SR[CH] = 1$ $STAT[CH] = 0$	$SR[CH] := 0$, $ERROR[CH] := 0$ $ret = STATUS[31:0]$
0110	setmod		$NMODC[1:0] := CH[1:0]$ $ret = STATUS[31:0]$
0111	setbit		$RBIT[3:0] := CH[3:0]$ $ret = STATUS[31:0]$
1000	startdma	$SR[CH] = 1$ $STAT[CH] = 0$	$STAT[CH] := 1$, $ERROR[CH] := 0$, $TERM[CH] := 0$, $Fifo_{MOD,EA} := CH[3:0]$, $ret = STATUS[31:0]$
1001	enddma	$SR[CH] = 1$ $STAT[CH] = 1$	$TERM[CH] := 1$ $ret = STATUS[31:0]$
Schreibzugriffe; $\langle DATA[31:0]\rangle$ ist das zu schreibende Datum vom lokalen Bus			
0011	whc		$HC[31:0] := DATA[31:0]$
1100	setpadr	$SR[CH] = 1$ $STAT[CH] = 0$	$PADR[CH][31:0] := DATA[31:0]$
1101	setstart	$SR[CH] = 1$ $STAT[CH] = 0$	$START[CH][23:2] := DATA[23:2]$
1110	setaux	$SR[CH] = 1$ $STAT[CH] = 0$	$AUX[CH][31:0] := DATA[31:0]$

Tabelle 2.6: Kodierung und Effekt der Befehle

ret ist das 32-Bit-Wort, das bei einem Lesezugriff auf dem lokalen Bus zurückgegeben wird. *get* ist ein Teil des Statusworts $STATUS[31:0]$ nach Abbildung 2.7.
In den Spalten *Bedingung* und *Effekt* gelten folgende Konventionen: Sei R ein Register und sei $X \in \{CH, get\}$. Dann steht $R[X]$ für $R[\langle X[3:0]\rangle]$.
Der Befehl *setmod* schreibt das $MODC$-Register nicht direkt; statt dessen setzt er ein Zwischenregister $NMODC$, aus dem $MODC$ dann geladen wird. In Abschnitt 2.6 wird erläutert, warum dieses Zwischenregister $NMODC$ benötigt wird.
Der Ausdruck $Fifo_{MOD,EA} := CH[3:0]$ beim Befehl *startdma* steht für das Schreiben von $CH[3:0]$ in den Fifo-Buffer $Fifo_{MOD,EA}$

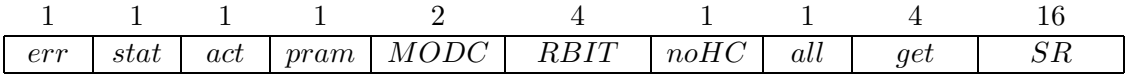


Abbildung 2.7: Aufbau des Statusworts des DMA-Prozessors

mit *setbit* beziehungsweise *setmod* gesetzt werden. Zusätzlich kann noch mit *getid* eine ID des DMA-Prozessors abgefragt werden, und die Ausführung eines DMA-Programms auf einem Kanal kann mit *enddma* abgebrochen werden.

Der Effekt der einzelnen Befehle in Bezug auf die eingeführten sichtbaren Register ist ebenfalls in Tabelle 2.6 spezifiziert. Nicht spezifizierte Schreibzugriffe werden auf dem lokalen Bus lediglich quittiert, haben jedoch keine Auswirkungen im DMA-Prozessor. Nicht spezifizierte Lesezugriffe führen nur dazu, daß der Befehl *getid* ausgeführt wird.

Zusätzlich ist in Tabelle 2.6 unter *Bedingung* angegeben, wann der spezifizierte Effekt nur eintritt. In Abhängigkeit vom aktuellen Wert der sichtbaren Register haben die Befehle also entweder die spezifizierte Wirkung oder überhaupt keine. Bei Lesezugriffen wird natürlich das spezifizierte Datum zurückgegeben, ganz gleich, ob die Bedingung erfüllt ist oder nicht. Die Bedingungen für die einzelnen Befehle sind dabei bewußt gewählt, um Befehle zu ignorieren, die beim aktuellen Zustand des DMA-Prozessors „unsinnig“ sind. So ist es ist beispielsweise nicht möglich, ein DMA-Programm auf einem Kanal zu starten, der gar nicht reserviert ist oder ein DMA-Programm auf einem Kanal zu starten, während auf diesem Kanal gleichzeitig noch ein Programm läuft. Hat die Hashkonstante hc den Wert 0, dann ist es nicht möglich, einen Kanal zu reservieren und damit natürlich auch nicht, ein DMA-Programm auszuführen, da ja kein DMA-Kanal reserviert werden kann. So wird gewährleistet, daß bei einer Hashkonstante $hc = 0$ keine DMA-Programme ausgeführt werden.

Um ein DMA-Programm auszuführen, reserviert sich der PC zunächst mit dem Befehl *getdma* einen Kanal c. Dann schreibt er mit *setpadr*, *setstart* und *setaux* das DMA-Programm für diesen Kanal c. Anschließend startet der PC das Programm mit *startdma*, wobei er auch festlegt, ob das Programm eine Einzelanweisung ist und ob es bei aktivem Modulo-Bit ausgeführt werden soll. Durch Auslesen des Statusworts mit dem Befehl *status* für den Kanal c stellt der PC fest, ob das Programm auf Kanal c beendet ist. Anschließend kann er den DMA-Kanal c mit *freedma* freigeben oder ein neues DMA-Programm schreiben und es auf dem Kanal c starten.

Mit dem Befehl *enddma* kann der PC die Ausführung eines DMA-Programms auf Kanal i vorzeitig beenden. Wenn das nächste Teilprogramm auf Kanal i ausgeführt wird, dann wird die Ausführung des DMA-Programms auf Kanal i hiermit beendet.

2.4.1 Statuswort

Das Statuswort des DMA-Prozessors wird bei allen Lesezugriffen außer *getid* und *rhc* zurückgegeben. Es ist gemäß Abbildung 2.7 aufgebaut. Dabei beziehen sich jeweils nur die drei obersten Bits auf den beim Lesezugriff im Feld CH spezifizierten Kanal. Es wird immer das Statuswort *vor* der Ausführung des Befehls zurückgegeben. Die einzelnen Felder des Statusworts haben die folgende Bedeutung:

- $SR[15:0]$: Das ist das gleichnamige Register des DMA-Prozessors (siehe Tabelle 2.5 auf Seite 46). Die Bits dieses Scheduling-Registers geben an, welche der 16 DMA-Kanäle vergeben beziehungsweise frei sind. Kanal i ist dabei genau dann vergeben, wenn $SR[i] = 1$.

- $get[3:0]$: Beim Befehl *getdma* steht hier der durch diesen Befehl reservierte Kanal. Die Angaben sind gemäß den Bedingungen aus Tabelle 2.6 gültig, wenn noch nicht alle DMA-Kanäle vergeben sind und eine Hashkonstante $\neq 0$ eingestellt ist. In diesem Fall ist $\langle get[3:0]\rangle = \min\{i|SR[i] = 0\}$ die Nummer des reservierten Kanals. Bei allen anderen Befehlen kann dieser Teil ignoriert werden; er gibt dann nur an, welches der nächste freie DMA-Kanal ist.

- *all*: Genau dann, wenn zur Zeit alle 16 DMA-Kanäle vergeben sind, wenn also $SR[15:0] = 1^{16}$, gilt $all = 1$. Gilt beim Befehl *getdma* $all = 1$, dann wurde kein Kanal reserviert, da die Bedingung für den Befehl nicht erfüllt ist.

- *noHC*: Dieses Bit gibt an, daß keine Hashkonstante im DMA-Prozessor gesetzt ist, also $\langle HC[31:0]\rangle = 0$. Gilt $noHC = 1$, dann lassen sich keine DMA-Kanäle reservieren.

- $RBIT[3:0]$: Das ist das gleichnamige Register des DMA-Prozessors. Die Sortiereinheit verwendet $16 + \langle RBIT[3:0]\rangle$ als Routing-Bit-Nummer.

- $MODC[1:0]$: Eine Netzwerkrunde der SB-PRAM besteht aus $1 + \langle MODC[1:0]\rangle$ Teilrunden. Auch das ist das gleichnamige Register des DMA-Prozessors.

- *pram*: Hier steht nur dann eine 1, wenn PRAM-Strom anliegt und ein Reset der PRAM erfolgt ist, wenn die SB-PRAM also hochgefahren ist.

- *act*: Eine 1 signalisiert, daß der Kanal $\langle CH[3:0]\rangle$ zur Zeit vergeben ist, wobei $CH[3:0]$ der Adresse des Befehls gemäß Abbildung 2.6 entnommen wird. Also ist $act = SR[\langle CH[3:0]\rangle]$.

- *stat*: Dieses Bit enthält genau dann eine 1, wenn auf dem Kanal $\langle CH[3:0]\rangle$ zur Zeit ein DMA-Programm läuft. Es gilt also $stat = STAT[\langle CH[3:0]\rangle]$.

- *err*: Hier steht genau dann eine 1, wenn durch das Programm des Kanals $\langle CH[3:0]\rangle$ ein Fehler im Netzwerk der SB-PRAM aufgetreten ist. Dieser Fall kann beispielsweise eintreten, wenn bei den Antworten des Netzwerks ein Parity-Fehler aufgetreten ist. Es ist $err = ERROR[\langle CH[3:0]\rangle]$.

Das Statuswort ist so aufgebaut, daß das folgende nützliche Lemma gilt.

Lemma 2.4.1 *Ob die Bedingungen zur Ausführung der Befehle aus Tabelle 2.6 erfüllt sind, kann bei allen Lesezugriffen anhand des zurückgegebenen Datums ret festgestellt werden.*

Beweis: Die einzigen Befehle, die nicht das Statuswort zurückgeben, sind *getid* und *rhc*, und diese Befehle haben keine Bedingungen. Also genügt es, Befehle zu betrachten, die das Statuswort zurückgeben. Die Felder *act* und *stat* des Statusworts enthalten nach Definition genau die Bits $SR[\langle CH[3:0]\rangle]$ und $STAT[\langle CH[3:0]\rangle]$. Damit sind schon alle Bedingungen im Statuswort kodiert, abgesehen von denen beim Befehl *getdma*, nämlich $HC \neq 0^{32}$ und $SR \neq 1^{16}$. Nach Definition gilt $noHC = 1 \Leftrightarrow \langle HC[31:0]\rangle = 0 \Leftrightarrow HC = 0^{32}$ und $all = 1 \Leftrightarrow SR = 1^{16}$. Also gilt die Behauptung auch für den Befehl *getdma*. □

2.4.2 Stalling

Einige der Instruktionen des DMA-Prozessors greifen auf Ressourcen zu, die auch während der Ausführung von DMA-Programmen benutzt werden. So wird beim Befehl *startdma* beispielsweise in den DMA-Scheduler und ins Status-Register geschrieben. Beide Ziele können auch beim Ausführen von DMA-Programmen beschrieben werden, und zwar schlimmstenfalls im selben Takt, in dem auch *getdma* sie schreiben will. Deshalb muß einer der beiden Zugriffe warten.

Im DMA-Prozessor wird generell die Ausführung von Instruktionen gestallt und nicht die Ausführung der DMA-Programme, damit ein ununterbrochener Datenstrom zur Sortiereinheit gewährleistet werden kann. Will also beispielsweise der Befehl *startdma* gerade einen Fifobuffer $Fifo_{mod,ea}$ des DMA-Schedulers schreiben, während gerade ein anderes DMA-Programm reschedult wird, dann wird die Ausführung von *startdma* um einem Takt gestallt. Nur das *STAT*-Register kann von 3 Seiten geschrieben werden: durch einen Befehl, in der Stufe *EX* und in *WB*. Wie in Abschnitt 1.2.3 bereits erwähnt, wird der Schreibzugriff von *WB* gestallt, wenn *EX* gleichzeitig dieses Register schreibt. Damit wird die Ausführung eines Befehls, der das *STAT*-Register schreibt, 2 Takte lang gestallt, wenn gleichzeitig mit dem Befehl sowohl die Stufe *EX* als auch *WB* das *STAT*-Register schreiben.

Insgesamt können genau die folgenden Instruktionen gestallt werden:

- Der Befehl *startdma* wird gestallt, wenn die Stufe *EX* oder *WB* gerade das Status- oder Error-Register schreiben oder gerade ein DMA-Programm reschedult wird.
- Der Befehl *freedma* wird gestallt, wenn die Stufe *WB* gerade das Error-Register schreibt.
- Der Befehl *getdma* wird gestallt, wenn die Stufen *EX* oder *WB* gerade das Status-Register schreiben.
- Die Befehle *setpadr*, *setstart* und *setaux* werden gestallt, wenn gerade ein Teil eines DMA-Programms ins Registerfile zurückgeschrieben wird oder wenn ein Teil eines DMA-Programm gerade aus dem Registerfile gelesen wird. Der Einfachheit halber werden gleich *alle* schreibenden Instruktionen gestallt. Dadurch werden auch der Befehl *whc* und alle nicht spezifizierten Schreibzugriffe gegebenenfalls gestallt.

In allen anderen Fällen kann es zu keinen Konflikten mit den Ausführen von DMA-Programmen kommen.

2.5 Prozessor

Der DMA-Prozessor führt die in Abschnitt 2.4 eingeführten Instruktionen aus und erlaubt die Ausführung von DMA-Programmen nach dem Algorithmus aus Abschnitt 1.2.3. Außerdem erfolgen die Zugriffe auf das SDRAM der PCI-Karte ebenfalls über den DMA-Prozessor. Abbildung 2.8 zeigt die top-level Datenpfade des DMA-Prozessors. Für den SDRAM-Zugriff muß die Adresse eines RAM-Zugriffs auf dem lokalen Bus LAD an den Adreßport A des SDRAMs gelegt werden. Die Adresse auf dem lokalen Bus LAD wird deshalb in einem Register *pciadr* zwischengespeichert. Beim Anlegen einer Adresse *ramadr* an das SDRAM erfolgt noch eine Auswahl, welche Adreßhälfte angelegt wird. Die in das RAM zu schreibenden Daten werden in einer *store*-Fifo gespeichert, bevor sie an das SDRAM gesendet werden. Analog werden die aus dem SDRAM gelesenen Daten von RDQ in eine *load*-Fifo gespeichert, bevor sie auf den lokalen Bus LAD gegeben werden. Das Design von Fifo-Buffern wird in Abschnitt 2.7.2 beschrieben.

Nach den bisherigen Bemerkungen ist schon weitgehend klar, wie das Blockschaltbild des DMA-Prozessors aussieht. LAD ist der lokale Bus, auf dem Adressen und Daten nacheinander anliegen. RDQ und A sind der Daten- beziehungsweise Adreßbus zum SDRAM, und SN_{in} und SN_{out} das Interface zur Sortiereinheit. Wesentliche Bestandteile des DMA-Prozessors sind der *DMA-Scheduler* und das *DMA-Program*-Environment, welches das Registerfile für $PADR$, $START = (OP, SADR)$ und AUX enthält und das HC-Register. Außerdem enthält dieses Environment weitere Kontrollogik für die Stufe EX. Das *Status-Register*-Environment enthält die Register $STAT$, $ERROR$, $TERM$, SR, $RBIT$ und $MODC$ sowie die nötige Logik, um aus SR die Nummer des nächsten freien Kanals für den Befehl *getdma* zu berechnen. Darüber hinaus besteht der DMA-Prozessor noch aus den bereits eingeführten Fifo-Buffern, der *address*-Fifo, der *send*-Fifo und der *receive*-Fifo sowie einigen Multiplexern.

Bei der Ausführung eines DMA-Programms liest der DMA-Prozessor Daten aus dem lokalen RAM in die *send*-Fifo und sendet diese Daten über SN_{in} abwechselnd mit den zugehörigen PRAM-Adressen $PADR$ an die Sortiereinheit. Bei Einzelanweisungen wird statt eines Datums aus der *send*-Fifo der Inhalt des AUX-Registers an die Sortiereinheit gesendet. Die von der PRAM ankommenden Daten werden in der *receive*-Fifo zwischengespeichert, bevor sie ins lokale RAM geschrieben werden. Die *address*-Fifo dient als Zwischenspeicher für die sogenannten Write-Infos, die im wesentlichen aus der Adresse bestehen, an der die in der *receive*-Fifo ankommenden Daten gespeichert werden.

Ins lokale RAM schreibt der DMA-Prozessor entweder die Daten des PC vom lokalen Bus aus der *store*-Fifo oder die Antworten der PRAM auf Speicheranfragen aus der *receive*-Fifo. Bei Zugriffen des PCs auf das lokale RAM ist $\langle pciadr \rangle$ die Adresse für den RAM-Zugriff, bei Lesezugriffen von DMA-Programmen in der Stufe $READ$ ist es $\langle SADR \rangle$, und beim Schreiben in WB die Adresse aus der *address*-Fifo. Eine zusätzliche Adresse $\langle INIT \rangle$ wird zum Initialisieren des SDRAMs benötigt.

Greift der PC auf eine Adresse im Adreßraum des DMA-Prozessors zu, dann merkt sich der DMA-Prozessor in *pciadr* die Adresse dieses Zugriffs. Bei einer Instruktion des DMA-Prozessors ist in dieser Adresse die Kanalnummer CH eines DMA-Kanals gespeichert. Diese Kanalnummer CH wird an das *Status-Register* gegeben, damit das Statuswort be-

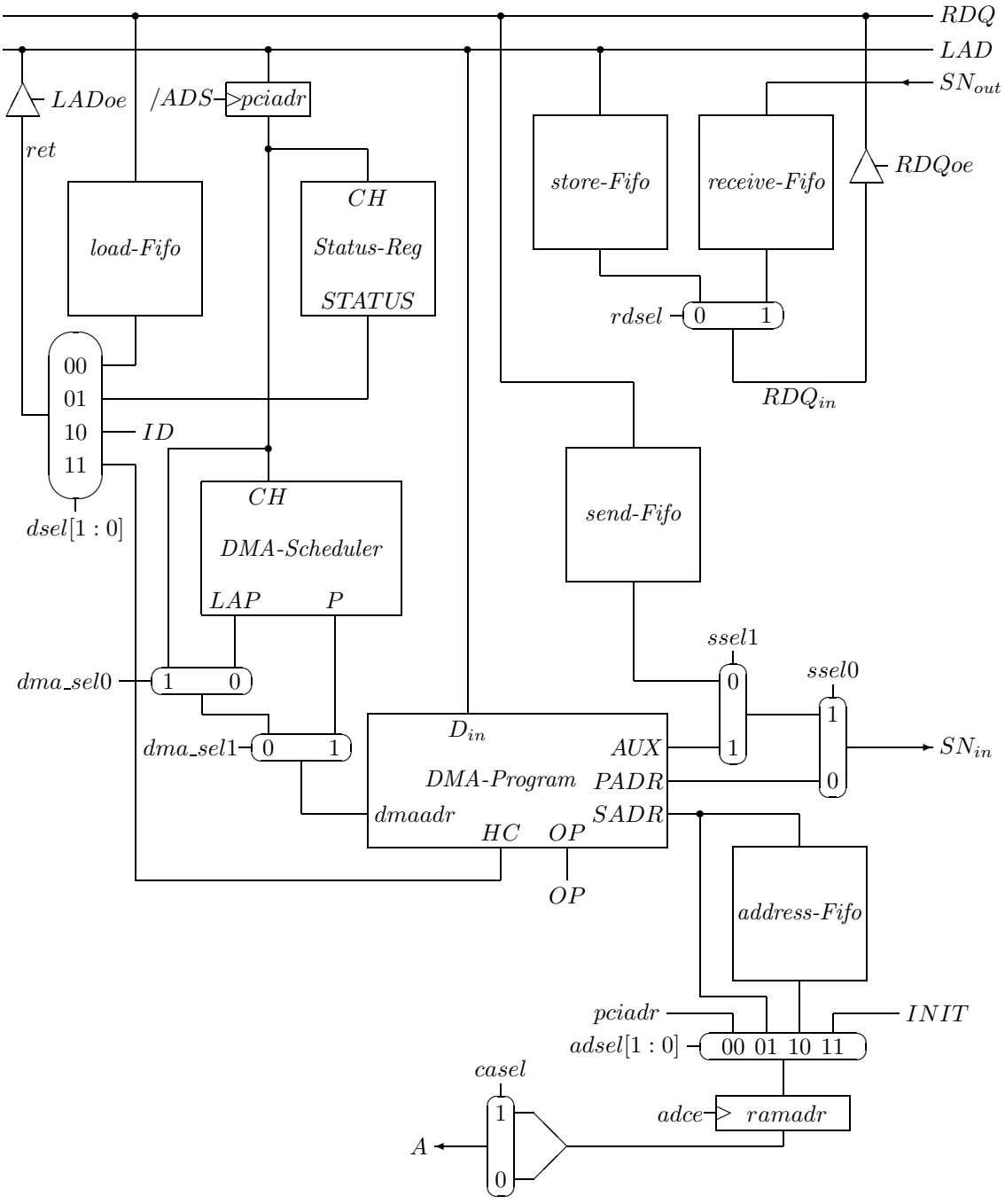


Abbildung 2.8: Datenpfade des DMA-Prozessors

ID steht für $\text{bin}_{32}(1398162766)$ (='SVEN' ASCII-kodiert) und $INIT$ steht für eine spezielle Adresse, die zum Konfigurieren des SDRAMs nach einem Reset anliegt.
Das *Status-Register* enthält die sichtbaren Register SR, $ERROR$, $STAT$, $TERM$, $MODC$ und $RBIT$ und der *DMA-Scheduler* die 4 Fifo-Buffer $Fifo_{mod,ea}$.

rechnet werden kann. Beim Befehl *startdma* wird die Kanalnummer CH aus *pciadr* in den *DMA-Scheduler* geschrieben, und zwar in $Fifo_{MOD,EA}$. Dazu wird der entsprechende Teil von *pciadr* an den *DMA-Scheduler* angelegt. Beim Schreiben des DMA-Programms dient die Kanalnummer CH aus *pciadr* als Adresse im Register-File für $PADR$, $START$ und AUX. Dazu liegt CH am *DMA-Program*-Environment an. Der PC kann als Rückgabewert *ret* bei einem Lesezugriff die Hashkonstante lesen, die ID, das Statuswort und ein Datum über die *load*-Fifo aus dem lokalen RAM.

Im DMA-Scheduler werden die Kanalnummern der laufenden DMA-Programme gespeichert. Dieser DMA-Scheduler liefert zusätzlich *zwei* Kanalnummern als Ausgabe: die Kanalnummer des DMA-Programms in der Stufe *READ*, $\langle LAP \rangle$, und die Kanalnummer des DMA-Programms in der Stufe *EX*, $\langle P \rangle$. Bei der Ausführung von DMA-Programmen wird das Register-File im *DMA-Program*-Environment beim Reschedulen eines DMA-Programms in der Stufe *EX* geschrieben und in der Stufe *READ* gelesen. Deshalb kann das Registerfile im *DMA-Program*-Environment auch mit den oben eingeführten Kanalnummern P und LAP adressiert werden.

Am Eingang der Sortiereinheit, SN_{in}, liegen nach Tabelle 2.2 und 2.3 entweder Adresse und Modus eines Zugriffs oder Datum und Operator. Aus Gründen der Übersichtlichkeit ist in Abbildung 2.8 nur die Adresse beziehungsweise das Datum aufgeführt, die an SN_{in} anliegen, und nicht Modus beziehungsweise Operator. Diese Teile sind nach Abbildung 2.5 als Teil von $START$ in OP kodiert; das *DMA-Program*-Environment liefert die beiden Teile $SADR$ und OP von $START$. Genau gilt also:

$$SN_{in}[33:0] = \begin{cases} OP[1:0]PADR[31:0] & \text{falls } ssel0 = 0 \\ OP[3:2]send.D_{out}[31:0] & \text{falls } ssel0 = 1 \text{ und } ssel1 = 0 \\ OP[3:2]AUX[31:0] & \text{falls } ssel0 = 1 \text{ und } ssel1 = 1 \end{cases}$$

In der *address*-Fifo werden Write-Infos gespeichert. Diese Write-Infos setzen sich aus mehreren Teilen zusammen, von denen der Übersichtlichkeit halber nur der Hauptteil $SADR$ in Abbildung 2.8 eingezeichnet ist. Die Berechnung der zusätzlichen Teile *primitiv* und $EADR'[2:0]$ wird in den Abschnitten 2.9 und 2.9.2 erläutert.

$$address.D_{in}[25:0] = (primitiv, EADR'[2:0], P[3:0], SADR[17:0])$$
$$address.D_{out}[25:0] = (primitiv\_wb, eadr'\_wb[2:0], p\_wb[3:0], sadr\_wb[17:0])$$

Im Register *ramadr* wird aus der *relativen* Startadresse aus dem $SADR$-Register mit Hilfe der Kanalnummer aus LAP die *absolute* Startadresse zusammengesetzt. Der Beitrag von LAP ist der Übersichtlichkeit wegen in Abbildung 2.8 nicht eingezeichnet. Für *ramadr* gilt:

$$ramadr[21:0] := \begin{cases} pciadr[21:0] & \text{falls } adce = 1 \text{ und } adsel[1:0] = 00 \\ LAP[3:0]SADR[17:0] & \text{falls } adce = 1 \text{ und } adsel[1:0] = 01 \\ p\_wb[3:0]sadr\_wb[17:0] & \text{falls } adce = 1 \text{ und } adsel[1:0] = 10 \\ INIT[21:0] & \text{falls } adce = 1 \text{ und } adsel[1:0] = 11 \\ ramadr[21:0] & \text{sonst} \end{cases}$$

Befehl	aktive Kontrollsignale
Lesezugriffe; *LADoe* ist immer aktiv	
getid	*dsel*1
status	*dsel*0
rhc	*dsel*1, *dsel*0
getdma	*srce*, *dsel*0, *status_we*, *status_sel*0
freedma	*srce*, *srsel*, *error_we*, *dsel*0
setmod	*nmodce*, *dsel*0
setbit	*bitce*, *dsel*0
startdma	*dma_push*, *dsel*0, *status_we*, *status_in*, *error_we*, *term_we*
enddma	*term_we*, *term_in*, *dsel*0
Schreibzugriffe	
whc	*hcce*
setpadr	*pwe*, *padrsel*, *dma_sel*0
setstart	*swe*, *sadrsel*, *dma_sel*0
setaux	*awe*, *dma_sel*0

Tabelle 2.7: Befehle und ihre aktiven Kontrollsignale

Abgesehen von den Select-Signalen $dsel[1:0]$ werden die Kontrollsignale beim Ausführen der Befehle nur dann aktiviert, wenn die jeweiligen Bedingungen aus Tabelle 2.6 erfüllt ist.

In Tabelle 2.7 sind die aktiven Kontrollsignale zu allen Befehlen aufgeführt. Mit Hilfe von Abbildung 2.8 und den aktiven Kontrollsignalen aus Tabelle 2.7 läßt sich das erste Lemma für die Korrektheit des DMA-Prozessors zeigen; die Gültigkeit folgt unmittelbar aus dem Blockschaltbild des DMA-Prozessors, da mit den Signalen $dsel[1:0]$ bei allen Befehlen genau das in Tabelle 2.6 spezifizierte Datum *ret* zurückgegeben wird.

Lemma 2.5.1 *Aktivieren die Befehle die Kontrollsignale gemäß Tabelle 2.7, so gibt der DMA-Prozessor aus Abbildung 2.8 bei allen Lesezugriffen das in Tabelle 2.6 spezifizierte Datum ret zurück.*

Als nächstes werden wir auf das Design des *Status-Registers*, des *DMA-Schedulers* und des *DMA-Program*-Environments eingehen. Dabei werden wir für jedes dieser Environments zeigen, daß die Befehle korrekt ausgeführt werden, deren Effekt sich auf dieses Environment bezieht. Für den *DMA-Scheduler* zeigen wir weiterhin, daß er bei geeigneter Aktivierung der Kontrollsignale den in Abschnitt 1.2.2 vorgestellten Algorithmus implementiert. Abschließend geben wir die Kontrollsignale zum Ausführen von DMA-Programmen an und zeigen, daß mit diesen Kontrollsignalen der Algorithmus zum Ausführen von DMA-Programmen implementiert wird.

2.6 Status-Register

Im Status-Register-Environment werden 31 der 32 Bit des Statusworts berechnet; insbesondere enthält es einen Schaltkreis, der die Nummer $\langle get[3:0]\rangle$ des nächsten freien DMA-Kanals für den Befehl *getdma* berechnet. Im Status-Register-Environment gemäß Abbildung 2.9 befindet sich die Register $STAT$, $ERROR$, $TERM$, $MODC$ und $RBIT$ sowie das Scheduling-Register SR.

Die Register $STAT$, $ERROR$ und $TERM$ des DMA-Prozessors sind jeweils platzsparend als 16×1-dual-port-RAM realisiert. Damit können diese Register bei der Ausführung von DMA-Programmen geschrieben werden, während sie parallel zur Statusabfrage ausgelesen werden. Das Scheduling-Register SR ist als 16-Bit-Register realisiert, da zur Berechnung des nächsten freien Kanals alle 16 Bits von SR gleichzeitig gelesen werden müssen.

Während die Register in einem FPGA durch einen $RESET$ wieder in einen definierten Zustand übergehen, wird RAM in einem FPGA nicht initialisiert. Deshalb ist der DMA-Prozessor so designt, daß er den ursprünglichen Inhalt der Register $STAT$, $ERROR$ und $TERM$ ignoriert, da diese Register jeweils als 16×1-RAM realisiert sind.

Nach einem $RESET$ muß zunächst mit *getdma* ein DMA-Kanal reserviert werden. Der Befehl *getdma* funktioniert nach den Bedingungen aus Tabelle 2.6 unabhängig von den erwähnten Registern und setzt das $STAT$-Register für den reservierten Kanal auf 0. Damit können die Befehle *setpadr*, *setstart*, *setaux*, *freedma* und *startdma* ausgeführt werden. Durch den Befehl *startdma* werden dann alle 3 Register $STAT$, $ERROR$ und $TERM$ initialisiert; der undefinierte Inhalt nach dem $RESET$ wird überschrieben. Damit funktionieren alle Befehle unabhängig vom Initialwert der Register $STAT$, $ERROR$ und $TERM$.

Die Anzahl der virtuellen Prozessoren, die ein PRAM-Prozessor simuliert, kann bei laufender SB-PRAM geändert werden. Das PRAM-Interface der PCI-Karte liefert keine Informationen über eine solche Änderung; deshalb wird sie nicht synchron auf der PCI-Karte ausgeführt. Allerdings soll eine entsprechende Änderung der Zahl der virtuellen Prozessoren für die PCI-Karte möglich sein, so daß das Modulo-Bit nach dieser Änderung auf der PCI-Karte und in der SB-PRAM wieder synchron ist.

Um diese Synchronizität des Modulo-Bits zu erreichen, gehen wir davon aus, daß in der SB-PRAM und auf der PCI-Karte die Umschaltung der virtuellen Prozessoren nur nach gewissen, ausgezeichneten Netzwerkrunden erfolgt. Abbildung 2.10 verdeutlicht, daß eine Umschaltung alle 24 Teilrunden problemlos möglich ist, da nach 24 Teilrunden in jedem Fall eine neue Netzwerkrunde mit Modulo-Bit 0 beginnt. Dabei wird von der ersten Teilrunde nach einem Reset an gezählt; diese Teilrunde erhält die Nummer 0. Eine Umschaltung nach $12 = \mathrm{kgv}(1,2,3,4)$ Runden dagegen reicht nicht, um die Synchronizität zu gewährleisten. Nach 12 Teilrunden beginnt zwar eine neue Netzwerkrunde, allerdings ist der Zustand des Modulo-Bits dann unterschiedlich, je nachdem, wie viele virtuelle Prozessoren simuliert werden.

Wenn wir also dafür sorgen, daß der Prozessor und die SB-PRAM die Zählung des Modulo-Bits nur alle 24 Teilrunden umschalten, ist die Synchronizität nach den beiden Umschaltvorgängen gewährleistet. Wir gehen deshalb davon aus, daß im Netzwerk der SB-

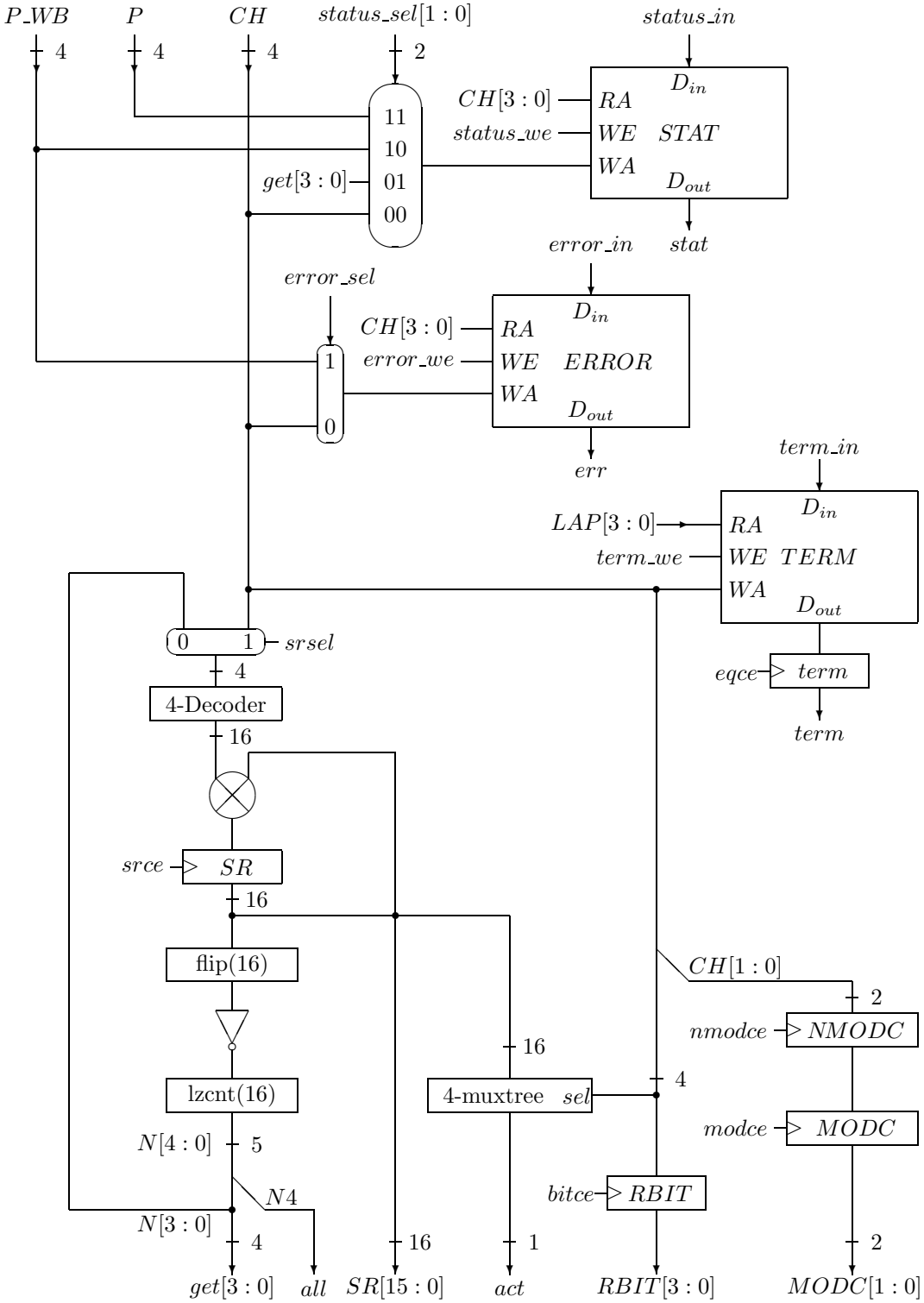


Abbildung 2.9: Status-Register

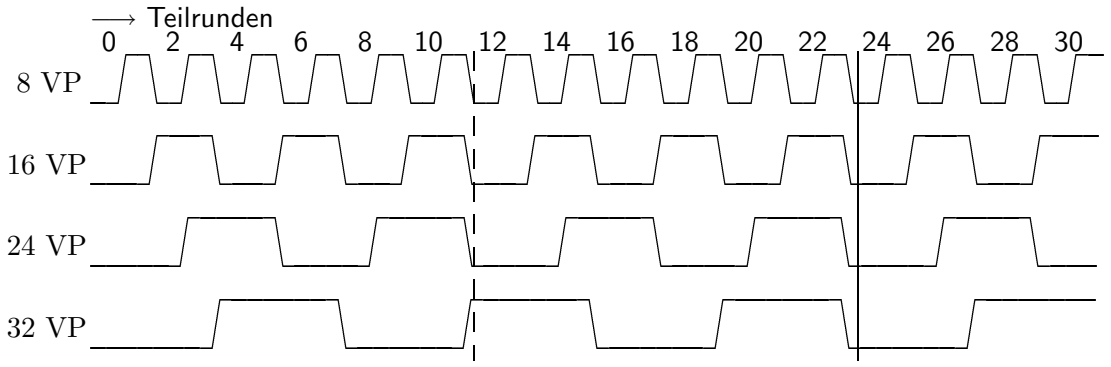


Abbildung 2.10: Modulo-Bit bei 8, 16, 24 oder 32 virtuellen Prozessoren

PRAM die Umschaltung nur alle 24 Teilrunden erfolgt. Um dieselbe Voraussetzung auch für den DMA-Prozessor zu gewährleisten, enthält das Status-Register des DMA-Prozessors ein zusätzliches Register $NMODC$, das wir bereits beim Befehl *setmod* erwähnt haben (vergleiche Tabelle 2.6). Bei diesem Befehl wird $NMODC$ gesetzt, und alle 24 Teilrunden wird das $MODC$-Register geclockt und aus diesem $NMODC$-Register geladen. Das $MODC$-Register gibt dann an, aus wie vielen Teilrunden eine Netzwerkrunde der SB-PRAM besteht.

Das $TERM$-Register ist das einzige Register des *Status-Registers*, das bei der Ausführung von DMA-Programmen gelesen wird, da es die Ausführung eines DMA-Programms beenden kann. Es wird nur über ein zusätzliches Register *term* ausgelesen, damit der Wert dieses Registers *term* während der Ausführung eines DMA-Programms in der Stufe *EX* konstant bleibt, denn ein DMA-Programm bleibt mehrere Takte lang in der Stufe *EX*.

2.6.1 Korrektheit

Für den beim Befehl *getdma* reservierten Kanal $\langle get[3:0]\rangle$ und das Flag *all* gelten nach der Spezifikation des Statusworts aus Abschnitt 2.6 die folgenden Gleichungen. Im Anschluß zeigen wir, daß $get[3:0]$ und *all* im *Status-Register* korrekt berechnet werden und daß die Befehle *getdma* und *freedma* den spezifizierten Effekt auf das Scheduling-Register SR haben.

$$get[3:0] = \begin{cases} \mathrm{bin}_4(\min\{i | 0 \le i \le 15, SR_i = 0\}) & \text{falls } SR[15:0] \neq 1^{16} \\ 0^4 & \text{sonst} \end{cases}$$

$$all = \begin{cases} 1 & \text{falls } SR[15:0] = 1^{16} \\ 0 & \text{sonst} \end{cases}$$

Bevor wir uns der Korrektheit des Status-Register zuwenden, erfolgen hier noch Definitionen für die verwendete Makros wie einen Decoder oder einen Leading-Zero-Counter.

Definition 2.6.1 *Ein* ***n-Decoder*** *ist ein Schaltkreis mit Eingaben* $x[n-1:0]$ *und Aus-*

gaben $Y[2^n - 1 : 0]$ *so, daß für* $0 \leq i \leq n-1$ *gilt:*

$$Y[i] = \delta_{\langle x \rangle, i}$$

mit der δ-Funktion $\delta_{i,k} := \begin{cases} 1 & \text{falls } i = k \\ 0 & \text{falls } i \neq k \end{cases}$

Definition 2.6.2 *Ein* ***n-Muxtree*** *ist ein Schaltkreis mit den Eingaben* $a[2^n - 1 : 0]$ *und* $sel[n-1:0]$ *sowie der Ausgabe y so, daß*

$$y = a[\langle sel[n-1:0] \rangle]$$

Definition 2.6.3 *Für* $n \in \mathbb{N}$ *hat ein Schaltkreis* ***n-flip*** *die Eingabe* $a[n-1:0]$ *und die Ausgabe* $b[n-1:0]$ *so, daß* $b[n-1:0] = a[0:n-1]$*. Die n Eingangssignale werden also genau vertauscht.*

Definition 2.6.4 *Für* $n = 2^k, k \in \mathbb{N}$ *ist ein* ***Leading-Zero-Counter*** *ein Schaltkreis mit Eingaben* $x[n-1:0]$ *und Ausgaben* $y[k:0]$*, der die Zahl der führenden Nullen im Binärstring x berechnet. Formal ausgedrückt:*

$$\langle y \rangle = lz(x) := \begin{cases} n - 1 - \max\{i \in B_k | x_i = 1\} & \text{falls } x \neq 0^n \\ n & \text{sonst} \end{cases}$$

Der *4-Decoder* und der *4-Muxtree* sind als vordefinierte und optimierte Makros von Xilinx realisiert. Der *Leading-Zero-Counter* ist gemäß des folgenden Lemmas nach [MP00] rekursiv aufgebaut. Ein Korrektheitsbeweis für einen solchen Leading-Zero-Counter ist dort ebenfalls gegeben.

Lemma 2.6.5 *Ein Leading-Zero-Counter für* $n = 2^k$ *Bits wird rekursiv definiert durch:*
$n = 1$*:* $y[0] = \overline{x[0]}$
$n = 2^{k+1}$*:* *Seien* $\langle y_L[k:0] \rangle = lz(x[\frac{n}{2} - 1 : 0])$ *und* $\langle y_H[k:0] \rangle = lz(x[n-1 : \frac{n}{2}])$*. Dann ist*

$$y[k+1:0] = \begin{cases} 0 \cdot y_H[k:0] & \text{falls } y_H[k] = 0 \\ y_L[k] \cdot \overline{y_L[k]} \cdot y_L[k-1:0] & \text{sonst} \end{cases}$$

Zunächst wollen wir hier zeigen, daß die Teile $get[3:0]$ und *all* des Statusworts korrekt berechnet werden. Nach Abbildung 2.9 gilt $all \cdot get[3:0] = N[4:0]$. Deshalb genügt es, das folgende Lemma zu zeigen.

Lemma 2.6.6 *Im* Scheduling-Register *(Abbildung 2.9) gilt:*

$$\langle N[4:0] \rangle = \begin{cases} \min\{i \in B_3 | SR[i] = 0\} & \text{falls } SR[15:0] \neq 1^{16} \\ 16 & \text{sonst} \end{cases}$$

Beweis:

$$\begin{aligned}
\langle N[4:0]\rangle &= lz(\overline{SR[0:15]}) \\
&= \begin{cases} 15 - \max\{i \in B_3 | \overline{SR_{15-i}} = 1\} & \text{falls } \langle \overline{SR[0:15]} \rangle \neq 0 \\ 16 & \text{sonst} \end{cases} \\
&= \begin{cases} 15 - \max\{i \in B_3 | SR_{15-i} = 0\} & \text{falls } SR[15:0] \neq 1^{16} \\ 16 & \text{sonst} \end{cases} \\
&= \begin{cases} \min\{i \in B_3 | SR_i = 0\} & \text{falls } SR[15:0] \neq 1^{16} \\ 16 & \text{sonst} \end{cases}
\end{aligned}$$

□

Als nächstes zeigen wir, daß das Scheduling-Register SR durch die Befehle *getdma* und *freedma* wie spezifiziert geändert wird. Es wird also genau dann ein Bit i in SR aktiviert, wenn mit dem Befehl *getdma* der DMA-Kanal i reserviert wird, und genau dann ein Bit i in SR ausgeschaltet, wenn Kanal i mit dem Befehl *freedma* freigegeben wird. Dazu benötigen wir noch 2 Hilfsdefinitionen.

Definition 2.6.7 *Sei S ein Kontrollsignal und R ein Register. Dann bezeichnet S^T den Wert des Kontrollsignals im Takt T und R^T den Wert des Registers im Takt T.*

Definition 2.6.8 *Sei $a[n-1:0], b[n-1:0] \in \{0,1\}^n$. Dann heißt*

$$H(a,b) := |\{i | 0 \leq i \leq n-1, a[i] \neq b[i]\}|$$

*die **Hamming-Distanz** zwischen a und b.*

Nach Tabelle 2.7 wird das Kontrollsignal *srce* genau dann aktiv, wenn einer der Befehle *getdma* oder *freedma* ausgeführt wird und die zugehörige Bedingung erfüllt ist. Damit bleibt nur noch zu zeigen, daß sich bei inaktivem *srce* der Inhalt von SR nicht ändert und bei aktivem *srce* genau die spezifizierte Änderung auftritt.

Lemma 2.6.9 *Im Status-Register gilt in jedem Takt T: $H(SR^T, SR^{T+1}) = srce$, bei aktivem srce ändert sich also genau* ein *Bit in SR und bei inaktivem srce ändert sich* kein *Bit. Wenn die Kontrollsignale gemäß Tabelle 2.7 aktiv werden, erfüllt SR außerdem die Spezifikation. Bei einem gültigen* getdma *wird also das Bit $SR[\langle get[3:0]\rangle]$ aktiviert und bei einem gültigen* freedma *wird Bit $SR[\langle CH[3:0]\rangle]$ deaktiviert.*

Beweis: Es gilt offensichtlich $srce^T = 0 \Longrightarrow H(SR^T, SR^{T+1}) = 0$. Der *4-Decoder* liefert nach Definition eine Ausgabe $DEC[15:0]$ mit $H(DEC, 0^{16}) = 1$. Nach Konstruktion von SR als Toggle-Register gilt deshalb $srce^T = 1 \Longrightarrow H(SR^T, SR^{T+1}) = 1$, und der erste Teil der Behauptung ist bewiesen.

Nach Konstruktion ist auch klar, das dasjenige Bit i in SR, das seinen Wert bei *srce* ändert, das spezifizierte Bit ist, also bei *getdma* ist wegen $srsel = 0$ $i = \langle get[3:0]\rangle$ und

bei *freedma* ist $i = \langle CH[3:0]\rangle$. Damit bleibt nur noch zu zeigen, das dieses Bit i bei $srsel = 1$ ausgeschaltet und bei $srsel = 0$ eingeschaltet wird.

Betrachten wir zuerst den Fall $srsel^T = 1$ und $srce^T = 1$. Dann wird gerade der Befehl *freedma* ausgeführt, und die zugehörige Bedingung $SR[\langle CH\rangle]^T = 1$ ist erfüllt. Damit wird das Bit $SR[\langle CH[3:0]\rangle]$ getoggelt, und $SR[\langle CH[3:0]^T\rangle]^T = 1$. Dieses Bit ändert durch das Toggeln also seinen Wert von 1 auf 0.

Ist $srsel^T = 0$ und $srce^T = 1$, dann wird der Befehl *getdma* ausgeführt, und es gilt die Bedingung $SR[15:0]^T \neq 1^{16}$. Damit gilt $\langle get[3:0]^T\rangle = \min\{i \in B_3 | SR_i^T = 0\}$, da $SR[15:0]^T \neq 1^{16}$. Insbesondere ist deshalb $SR[\langle get[3:0]^T\rangle]^T = 0$. Dieses Bit ändert durch das Toggeln also seinen Wert von 0 auf 1. □

Damit wird das Statuswort bis auf das Bit *noHC* im *Status-Register* nach Spezifikation in Abschnitt 2.6 berechnet.

Korollar 2.6.10 *Das Status-Register aus Abbildung 2.9 liefert die in Abschnitt 2.6 spezifizierten Teile des Statusworts* $err = ERROR[\langle CH[3:0]\rangle]$, $stat = STAT[\langle CH[3:0]\rangle]$, $act = SR[\langle CH[3:0]\rangle]$, $MODC[1:0]$, $RBIT[3:0]$, $all = N4$, $get[3:0] = N[3:0]$ *und* $SR[15:0]$.

Als letztes können wir noch beweisen, daß der Teil der Befehle, der sich auf Register im Status-Register bezieht, korrekt ausgeführt wird.

Lemma 2.6.11 *Aktivieren die Befehle des DMA-Prozessors die Kontrollsignale gemäß den Tabellen 2.6 und 2.7, so wird der Teil des Effekts der Befehle, der sich auf die Register* SR, $STAT$, $ERROR$, $TERM$, $NMODC$ *und* $RBIT$ *bezieht, korrekt ausgeführt.*

Beweis: Wir betrachten hier nur den Befehl *getdma*. Der Beweis für alle übrigen Befehle verläuft analog. Sei die Bedingung von *getdma* nach Tabelle 2.6 erfüllt.

Aktive Kontrollsignale nach Tabelle 2.7: *srce*, *dsel*0, *status_we*, *status_sel*0

Aus Abbildung 2.9 und Lemma 2.6.9 folgt dann direkt:

$$STAT[\langle get[3:0]\rangle] := 0$$
$$SR[\langle get[3:0]\rangle] := 1$$

□

2.7 DMA-Scheduler

Im DMA-Scheduler werden die DMA-Kanäle verwaltet, auf denen zur Zeit gerade ein DMA-Programm läuft. Die Datenpfade des DMA-Schedulers können Abbildung 2.11 entnommen werden. Im wesentlichen besteht der DMA-Scheduler aus den 4 Fifo-Buffern $Fifo_{mod,ea}$, in denen die Nummern der DMA-Kanäle gespeichert werden, auf denen gerade ein DMA-Programm läuft.

Der DMA-Scheduler enthält auch das bereits eingeführte Register $P[3:0]$, in dem die Kanalnummer des DMA-Programms in der Stufe *EX* gespeichert ist. Ein analoges Register für die Stufe *READ* kann entfallen; die Kanalnummer $\langle LAP[3:0]\rangle$ der Stufe *READ* wird direkt über einen Multiplexer aus den Fifos gelesen.

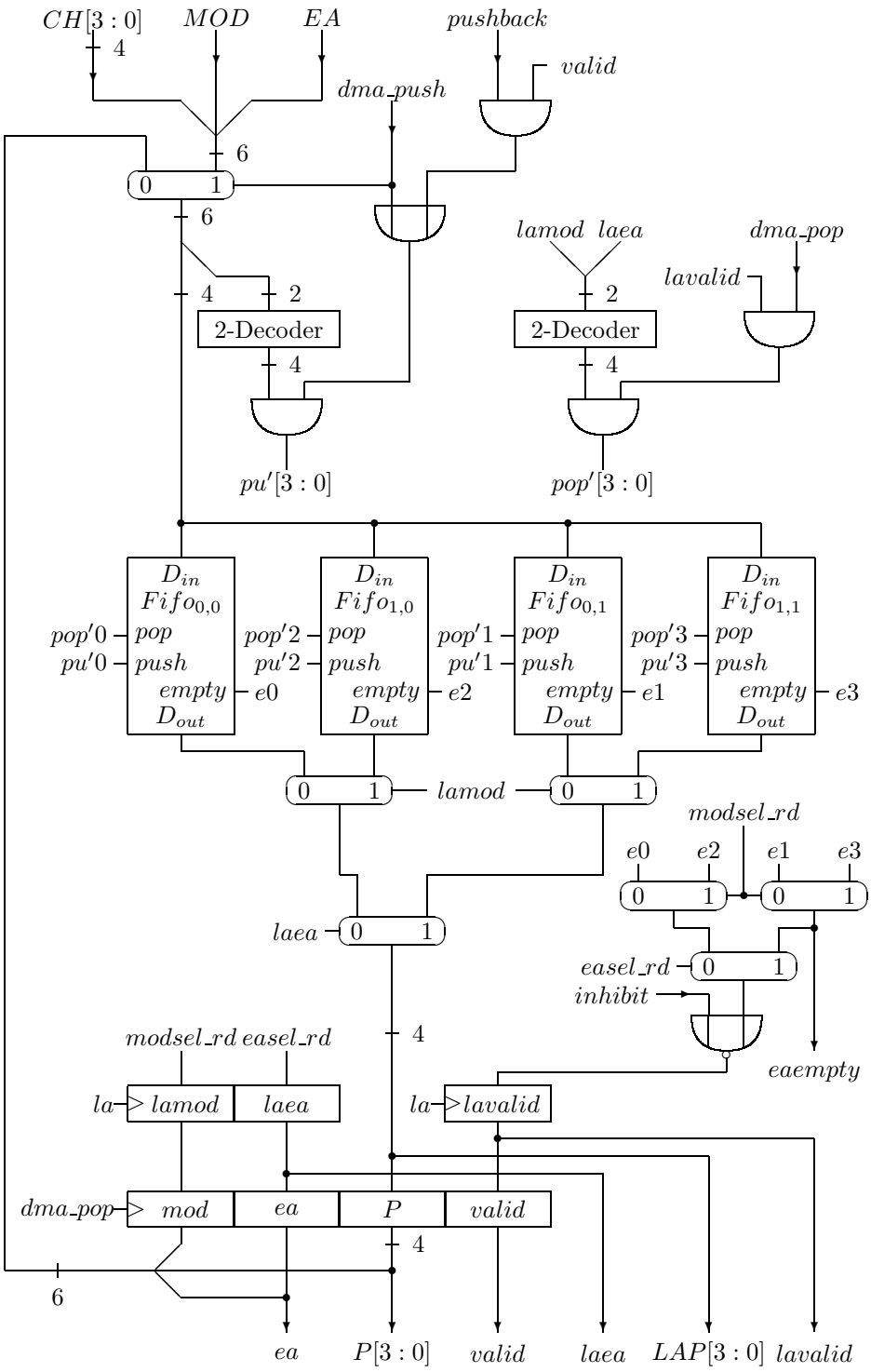


Abbildung 2.11: DMA-Scheduler

Für den DMA-Scheduler gliedert sich die Ausführung eines Teilprogramms in 3 Teile: der Übergang des Teilprogramms in die Stufe *READ*, wobei die zugehörige Kanalnummer dann an LAP anliegt, der Übergang in die Stufe *EX* mit der Kanalnummer im Register P und schließlich das Rescheduling des Teilprogramms in der Stufe *EX*, also das Schreiben der Kanalnummer $\langle P \rangle$ in diejenige Fifo, aus der die Kanalnummer auch gelesen wurde. Dabei kann natürlich ein DMA-Programm in die Stufe *READ* gelangen, während sich noch ein anderes DMA-Programm in der Stufe *EX* befindet.

Beim Ausführen von DMA-Programmen wird zunächst das Signal *la* aktiviert, um festzulegen, aus welchem Fifo-Buffer das DMA-Programm kommt, das als nächstes in die Stufe *READ* gelangt. Ist im folgenden Takt $lavalid = 0$, dann gelangt kein DMA-Programm in die Stufe *READ*. Sonst liegt dann die Kanalnummer des auszuführenden DMA-Programms an $LAP[3:0]$ und die Register *lamod* und *laea* geben an, aus welcher Fifo $LAP[3:0]$ stammt. Also ist das Programm auf dem Kanal $\langle LAP[3:0] \rangle$ genau dann eine Einzelanweisung, wenn $laea = 1$ ist, und *lamod* gibt den Zustand des Modulo-Bits an, bei dem das DMA-Programm ausgeführt wird.

Das Signal *eaempty* ist genau dann aktiv, wenn die Fifo mit Einzelanweisungen beim aktuellen Zustand des Modulo-Bits *modsel_rd* leer ist. Die gewünschte Priorisierung der Einzelanweisungen läßt sich also mit $easel\_rd = /eaempty$ erreichen. Außerdem wird *mpsel_rd* direkt vom Modulo-Bit-Register *modul* getrieben, dessen Berechnung später in Abschnitt 2.9.1 erläutert wird.

Durch Aktivieren von *inhibit* bei *la* wird in jedem Fall $lavalid := 0$ gesetzt, also gelangt kein DMA-Programm in die Stufe *READ*. Dabei ist $inhibit = address.full$, also gelangen bei voller *address*-Fifo keine DMA-Programme mehr in die Stufe *READ*. Jedes DMA-Programm, das in die Stufe *READ* gelangt, kann also noch ein Datum in die *address*-Fifo schreiben. Dadurch wird beim Schreiben der Write-Infos eine Abfrage überflüssig, ob die *address*-Fifo voll ist.

Beim Übergang eines DMA-Programms in die Stufe *EX* wird dann *dma_pop* aktiv. Dadurch wird die Kanalnummer aus der entsprechenden Fifo ausgelesen und in $P[3:0]$ gespeichert. Das Auslesen aus der Fifo erfolgt dabei nur, wenn $lavalid = 1$. Außerdem werden die Register *ea*, *mod* und *valid* aus den entsprechenden *la*-Registern geclockt. Ist nach *dma_pop* das Signal $valid = 0$, befindet sich kein DMA-Programm in der Stufe *EX*. Während eine Kanalnummer im Register $P[3:0]$ steht, kann die Kanalnummer für das nächste DMA-Programm in der Stufe *READ* mit *la* nach $LAP[3:0]$ gelesen werden.

Schließlich wird durch das Signal *pushback* die aktive Kanalnummer $P[3:0]$ wieder in die Fifo zurückgeschrieben, die durch die Register *mod* und *ea* ausgewählt wird. Auch hier wird nur zurückgeschrieben, wenn $valid = 1$.

An dieser Stelle definieren wir zunächst formal eine Fifo-Queue. Auf das Design von Fifo-Queues gehen wir in Abschnitt 2.7.2 ein.

Definition 2.7.1 *Unter einer $m \times n$-**Fifo-Queue** versteht man ein Schaltwerk mit $n+2$ Eingängen $e = e[n-1:0], pop, push$ und $n+2$ Ausgängen $a = a[n-1:0], empty, full$, der nach dem first-in-first-out Prinzip arbeitet. Am Ausgang a liegt genau das Datum, das sich am längsten in $Q_{m \times n}$ befindet. Betrachtet man $Q_{m \times n}$ als dynamische Menge von*

Elementen $(d,t) \in \{0,1\}^n \times \mathbb{N}$ *mit Startzustand* $Q_{m\times n}(0) = \emptyset$*, so gilt in Takt* $t \geq 0$*:*

$$
\begin{aligned}
Q_{m\times n}(t+1) = \;& Q_{m\times n}(t) \cup I(t) \backslash O(t) \\
I(t) = \;& \begin{cases} \{(e(t),t)\} & \textit{falls } push(t)=1 \textit{ und } full(t)=0 \\ \emptyset & \textit{sonst} \end{cases} \\
O(t) = \;& \begin{cases} \{(d_{f(t)}, f(t))\} & \textit{falls } pop(t)=1 \textit{ und } empty(t)=0 \\ \emptyset & \textit{sonst} \end{cases} \\
\textit{wobei} \quad & f(t) = \min\{i | (d_i, i) \in Q_{m\times n}(t)\} \\
a(t) = \;& d_{f(t)} \\
empty(t) = \;& \delta_{|Q_{m\times n}(t)|,0} \\
full(t) = \;& \delta_{|Q_{m\times n}(t)|,m}
\end{aligned}
$$

2.7.1 Korrektheit

Zuerst zeigen wir, daß der Befehl *startdma* korrekt ausgeführt wird, daß also mit *startdma* die richtige Kanalnummer in die richtige Fifo geschrieben wird.

Lemma 2.7.2 *Werden die Kontrollsignale gemäß Tabelle 2.7 aktiv und sind MOD, EA, CH aus dem Befehlswort, so wird der Befehl* startdma *nach der Spezifikation aus Tabelle 2.6 korrekt ausgeführt.*

Beweis: Vom Effekt des Befehls *startdma* nach Tabelle 2.6 ist nur noch zu zeigen, daß $Fifo_{MOD,EA} := CH[3:0]$, was sich in die zwei Teile $Fifo_{MOD,EA}.D_{in} = CH[3:0]$ und $Fifo_{MOD,EA}.push = 1$ aufspalten läßt. Nach Tabelle 2.7 wird bei dem Befehl *startdma* das Kontrollsignal *dma_push* aktiv. Nach den Bemerkungen über Stalling aus Kapitel 2.4.2 wird dann gerade kein DMA-Programm reschedult; in diesem Takt gilt $pushback = 0$. Damit ist in Abbildung 2.11 $Fifo_{MOD,EA}.D_{in} = CH[3:0]$. Da $dma\_push = 1$ ist, werden die Signale $pu'[3:0]$ direkt mit dem *2-Decoder* in Abbildung 2.11 berechnet; es gilt $Fifo_{i,j}.push = 1 \Longleftrightarrow i = MOD \wedge j = EA$, also genau $Fifo_{MOD,EA}.push = 1$. □

Wir wollen jetzt zeigen, daß der DMA-Scheduler bei geeigneter Aktivierung der Kontrollsignale die gepipelinete Verarbeitung aus Abschnitt 1.2.3 unterstützt. Werden die Signale *la*, *dma_pop* und *pushback* also nacheinander aktiv, dann wird der DMA-Kanal, der durch *la* in die Stufe *READ* gelangt, durch *pushback* wieder in die Fifo zurückgeschrieben, aus der er auch gelesen wurde. In Abschnitt 1.2.3 haben wir außerdem eingeführt, daß die Kanalnummer eines DMA-Programms in der Stufe *EX* zurückgeschrieben wird, *bevor* das nächste DMA-Programm in die Stufe *READ* gelangt, damit gegebenenfalls das gerade zurückgeschriebene DMA-Programm wieder in die Stufe *READ* gelangen kann. Damit wird das *la*-Signal erst wieder nach einem *dma_push* aktiv; wir betrachten deshalb im folgenden nur sogenannte zusammenhängende Folgen der Signale *la*, *dma_pop* und *pushback*.

Definition 2.7.3 *Seien* $S_0, S_1, \ldots, S_{n-1}$ *Kontrollsignale und* $T_0 < T_1 < \ldots < T_{n-1}$*. Dann heißt* $(S_0^{T_0}, S_1^{T_1}, \ldots, S_{n-1}^{T_{n-1}})$ *eine* ***zusammenhängende Folge von Kontrollsignalen****, wenn* $\forall i \in \{0,1,\ldots,n-1\} \forall T' \in \{T_0, T_0+1, \ldots, T_{n-1}\}$*:* $S_i^{T'} = \delta_{T',T_i}$*. Ein Signal* S_i *wird also im Intervall von* T_0 *bis* T_{n-1} *nur genau zum Zeitpunkt* T_i *aktiv.*

Ein Kontrollsignal S_n wird ***nur in zusammenhängender Folge*** *mit $(S_0, \ldots, S_{n-1})$ aktiv, wenn $\forall T_n$ mit $S_n^{T_n} = 1 \exists T_0, \ldots, T_{n-1}$ so, daß $(S_0^{T_0}, \ldots, S_n^{T_n})$ eine zusammenhängende Folge von Kontrollsignalen ist.*

Bevor wir uns diesen Beweisen zuwenden, machen wir noch die Beobachtung, daß die $full$-Signale aller Fifos des DMA-Schedulers ignoriert werden. Werden die Kontrollsignale bei der Ausführung von Befehlen mit den Bedingungen aus Tabelle 2.6 aktiv, dann können höchstens 16 DMA-Kanäle durch den Befehl *startdma* in den DMA-Scheduler geschrieben werden. Kanäle, auf denen bereits ein DMA-Programm läuft, können wegen der Bedingung $STAT[\langle CH \rangle] = 0$ mit diesem Befehl nämlich nicht noch einmal in den DMA-Scheduler geschrieben werden. Wenn also bei der Ausführung von DMA-Programmen im DMA-Scheduler kein DMA-Kanal mehrfach zurückgeschrieben wird und das *STAT*-Register genau dann eine 1 enthält, wenn auf dem zugehörigen Kanal ein DMA-Programm läuft, dann dürfen die $full$-Signale ignoriert werden

Zunächst zeigen wir, daß durch Aktivieren von *dma_pop* nach *la* das DMA-Programm in die Stufe *EX* gelangt, das durch dieses *la* in die Stufe *READ* gelangt war.

Lemma 2.7.4 *Sei $(la^T, dma\_pop^{T'})$ eine zusammenhängende Folge von Kontrollsignalen. Dann wird zum Zeitpunkt T' die Kanalnummer genau aus der Fifo gelesen, die zum Zeitpunkt T durch die Signale modsel_rd und easel_rd ausgewählt worden ist. Formal gilt:*

$$\begin{aligned}
P[3:0]^{T'+1} &= Fifo'.D_{out}[3:0]^T \text{ falls } Fifo'.empty^T = 0 \\
(mod, ea)^{T'+1} &= (modsel_rd, easel_rd)^T \\
valid^{T'+1} &= \overline{Fifo'.empty^T \vee inhibit^T} \\
Fifo_{i,j}.pop^{T'} = 1 &\iff i = modsel_rd^T \wedge j = easel_rd^T \wedge \overline{Fifo'.empty^T \vee inhibit^T} \\
mit \quad Fifo' &= Fifo_{modsel_rd^T, easel_rd^T}
\end{aligned}$$

Das Signal valid wird also genau dann aktiv, wenn tatsächlich ein Datum aus einer Fifo ausgelesen wurde und die Register mod und ea geben an, aus welcher Fifo das Datum gelesen wurde.

Beweis: Sei $(la^T, dma\_pop^{T'})$ eine zusammenhängende Folge von Kontrollsignalen. Dann folgt unmittelbar aus der Definition 2.7.3 für zusammenhängende Kontrollsignale:

$$\begin{aligned}
T_{la} &:= \max\{Z | T \leq Z \leq T', la^Z = 1\} = T \\
T_{dma_pop} &:= \min\{Z | T \leq Z \leq T', dma_pop^Z = 1\} = T'
\end{aligned}$$

Wegen $dma\_pop^T = 1$ gilt:

$$P[3:0]^{T'+1} = LAP[3:0]^{T'}$$

Um auf $LAP[3:0]^{T+1}$ zurückschließen zu können, genügt es zu zeigen, daß die Select-Signale *lamod* und *laea* des Multiplexers sich nicht geändert haben und am Ausgang der Fifo immer noch das selbe Datum liegt, wie zum Zeitpunkt $T+1$. Wegen $T_{la} = T$ gilt $(lamod^{T'}, laea^{T'}) = (lamod^{T+1}, laea^{T+1})$, die Select-Signale sind also gleich geblieben.

Damit das Datum, das zum Zeitpunkt T' am Ausgang der Fifo liegt, mit dem Datum zum Zeitpunkt T übereinstimmt, genügt es zu zeigen, daß in der Zwischenzeit kein Datum

aus der Fifo ausgelesen wurde, da für den Fall einer leeren Fifo zum Zeitpunk T nichts über das Register P behauptet wird. Wegen $T_{dma\_pop} = T'$ ist dieser Sachverhalt offensichtlich. Damit erhalten wir

$$P[3:0]^{T'+1} = LAP[3:0]^{T+1} \text{ falls } Fifo'.empty^T = 0$$

und mit $la^T = 1$ ergibt sich die Behauptung

$$P[3:0]^{T'+1} = Fifo'.D_{out}[3:0]^T \text{ falls } Fifo'.empty^T = 0$$

Der Beweis für die übrigen Teilbehauptungen verläuft analog. □

Jetzt zeigen wir, das der ganze Scheduling-Mechanismus korrekt funktioniert, wenn *la*, *dma_pop* und *pushback* in zusammenhängender Folge aktiv werden, daß also durch *pushback* in diesem Fall das DMA-Programm korrekt reschedult wird.

Lemma 2.7.5 *Sei $(la^{T_0}, dma\_pop^{T_1}, pushback^{T_2})$ eine zusammenhängende Folge von 3 Kontrollsignalen. Dann wird genau das Datum zum Zeitpunkt T_2 in die Fifo geschrieben, das zum Zeitpunkt T_0 am Ausgang der Fifo gelegen hat, wenn die Fifo nicht leer war. Formal gilt:*

$$\begin{aligned} Fifo'.D_{in}[3:0]^{T_2} &= Fifo'.D_{out}[3:0]^{T_0} \textit{ falls } Fifo'.empty^{T_0} = 0 \\ Fifo_{i,j}.push^{T_2} &= \delta_{i,modsel\_rd^{T_0}} \cdot \delta_{j,easel\_rd^{T_0}} \cdot \delta_{inhibit^{T_0},0} \cdot \delta_{Fifo'.empty^{T_0},0} \\ mit \quad Fifo' &= Fifo_{modsel\_rd^{T_0}, easel\_rd^{T_0}} \end{aligned}$$

Beweis: Nach den Bemerkungen zum Stalling in Abschnitt 2.4.2 wird *dma_push* nicht aktiv, wenn gerade *pushback* aktiv ist. Mit Hilfe von Abbildung 2.11 und Lemma 2.7.4 zeigt man dann direkt:

$$\begin{aligned} Fifo'.D_{in}[3:0]^{T_2} &= P[3:0]^{T_2} && (pushback^{T_2} = 1, dma\_push^{T_2} = 0) \\ &= P[3:0]^{T_1+1} && (dmapop^{T_1}, pushback^{T_2} \text{ zusammenhängend}) \\ &= Fifo'.D_{out}[3:0]^T && \text{(Lemma 2.7.4)} \end{aligned}$$

Analog läuft die Argumentation für $Fifo_{i,j}.push$.

$$\begin{aligned} Fifo_{i,j}.push^{T_2} = 1 &\iff i = mod^{T_2} \wedge j = ea^{T_2} \wedge valid^{T_2} = 1 && (dma\_push^{T_2} = 0) \\ &\iff i = mod^{T_1+1} \wedge j = ea^{T_1+1} \wedge && \text{(zusammenhängend)} \\ &\quad valid^{T_1+1} = 1 \\ &\iff i = modsel\_rd^{T_0} \wedge j = easel\_rd^{T_0} \wedge && \text{(Lemma 2.7.4)} \\ &\quad \overline{inhibit^{T_0} \vee Fifo'.empty^{T_0}} \end{aligned}$$

□

Korollar 2.7.6 *Aktiviert die Kontrolleinheit das Kontrollsignal pushback nur in zusammenhängender Folge mit (la, dma_pop), dann wird durch jedes pushback eine Kanalnummer genau einmal in genau die Fifo zurückgeschrieben, aus der sie auch gelesen wurde. Wird bei dma_pop keine Kanalnummer ausgelesen, dann wird auch keine Kanalnummer zurückgeschrieben. Aktiviert die Kontrolleinheit nur la und dma_pop ohne pushback in zusammenhängender Folge, dann wird eine Kanalnummer aus einer Fifo ausgelesen und nicht zurückgeschrieben.*

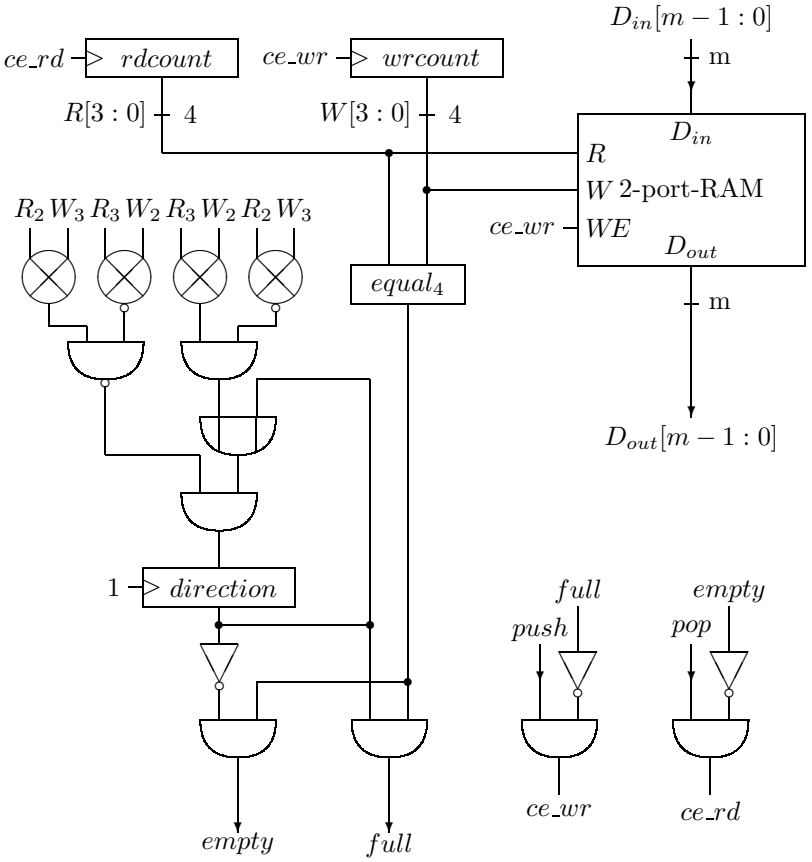


Abbildung 2.12: Aufbau einer Fifo-Queue

2.7.2 Fifo-Queues

Die Hardware-Implementierung einer Fifo-Queue besteht im wesentlichen aus einem RAM und zwei Zählern *rdcount* und *wrcount*. Abbildung 2.12 zeigt den Aufbau einer $m \times 16$-Fifo-Queue nach [XA96]. Um gleichzeitige Lese- und Schreibzugriffe zu erlauben, ist das RAM als 2-port-RAM realisiert. Der Zeiger *rdcount* zeigt auf das Element der Fifo-Queue, das als nächstes ausgelesen wird, und *wrcount* zeigt auf einen freien Platz, an den das nächste Element geschrieben wird.

Dazu kommt noch eine Kontrollogik, die die beiden Sonderfälle $full$ und *empty* erkennt. Ein *push* bei aktivem $full$ wird genauso wie ein *pop* bei aktivem *empty* von der Fifo-Queue selbst ignoriert. Da die Fälle $full$ und *empty* sich beide dadurch auszeichnen, daß *rdcount* und *wrcount* auf das gleiche Element zeigen, wird in einem zusätzlichen Register *direction* vermerkt, ob der Lesezähler dabei ist, den Schreibzähler einzuholen. Der Vergleich, ob *rdcount* und *wrcount* auf das gleiche Element zeigen, erfolgt mit einem sogenannten Equaltester, der mit einem vordefinierten Makro von Xilinx realisiert ist.

Definition 2.7.7 *Ein* ***n-Equaltester*** $equal_n$ *ist ein Schaltkreis mit Eingaben* $a[n-1:0]$,

	⟶ Schritte																	
Q_3	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	...
Q_2	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	...
Q_1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	...
Q_0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	...
	1. Quadrant				2. Quadrant				3. Quadrant				4. Quadrant					

Tabelle 2.8: 4-Bit-Zähler mit Ausgängen $Q[3:0]$

Die 16 Schritte des Zählers werden durch die doppelten Linien derart in 4 Quadranten unterteilt, daß die oberen beiden Bits $Q[3:2]$ in einem Quadranten immer den gleichen Wert haben und sich beim Übergang von einem Quadranten in den nächsten genau ein Bit aus $Q[3:2]$ ändert.

$b[n-1:0]$ *und Ausgabe eq so, daß*

$$eq = \delta_{a[n-1:0],b[n-1:0]}$$

Für das *direction*-Register im Fifo-Buffer gemäß Abbildung 2.12 gilt:

$$direction := \Big(\Big((R_3 \otimes W_2) \wedge (\overline{R_2 \otimes W3})\Big) \vee direction\Big) \wedge \overline{(\overline{R_3 \otimes W_2}) \wedge (R_2 \otimes W_3)}$$

Daraus kann man offensichtlich für das *direction*-Register in jeden Takt T folgern:

$$direction^T = 1 \wedge direction^{T+1} = 0 \Longrightarrow (\overline{R_3^T \otimes W_2^T}) \wedge (R_2^T \otimes W_3^T) = 1$$
$$direction^T = 0 \wedge direction^{T+1} = 1 \Longrightarrow (R_3^T \otimes W_2^T) \wedge (\overline{R_2^T \otimes W_3^T}) = 1$$

In Tabelle 2.8 ist dargestellt, wie der spezielle 4-Bit-Zähler für diese Fifo-Queue zählt. Ein Vergleich der obigen Formeln mit dieser Tabelle liefert das gewünschte Ergebnis: *direction* wird genau dann auf 1 gesetzt, wenn der Schreibzähler den Quadranten *unmittelbar vor* dem Lesezähler betritt, und *direction* wird genau dann auf 0 gesetzt, wenn der Lesezähler den Quadranten *unmittelbar vor* dem Schreibzähler betritt. In allen anderen Fällen behält *direction* den Wert, den es im Takt vorher hatte. Insbesondere beobachten wir, daß *direction* 4 Takte lang den gleichen Wert enthält, bevor Lese- und Schreibzähler auf das gleiche Element zeigen, bevor also die Fälle $full$ oder *empty* eintreten können.

Damit ist die Berechnung der Kontrollsignale $full$ und *empty* korrekt:

$$\begin{aligned} full &= direction \wedge \delta_{R[3:0],W[3:0]} \\ empty &= \overline{direction} \wedge \delta_{R[3:0],W[3:0]} \end{aligned}$$

Wir haben dieses Design nach [XA96] für synchrone Fifo-Buffer gewählt, da es sich leicht auf *asynchrone* Fifo-Buffer erweitern läßt, also auf Fifo-Buffer mit unabhängigen Lese- und Schreibclocks. Mit Hilfe solcher asynchroner Fifo-Buffer werden die Daten im DMA-Prozessor zwischen den verschiedenen Clocks synchronisiert. So ist die *load*-Fifo

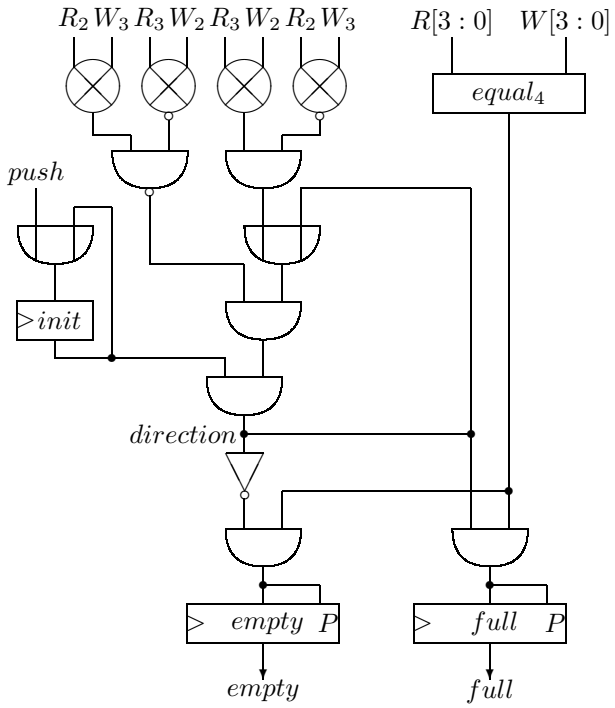


Abbildung 2.13: Berechnung von $full$ und $empty$ in einer asynchronen Fifo-Queue

des DMA-Prozessor eine asynchrone Fifo, die mit der RAM-Clock $RCLK$ geschrieben und mit der lokalen Clock CLK ausgelesen wird. Auch in der Sortiereinheit werden diese Fifo-Buffer zur Synchronisierung zwischen der lokalen Clock CLK und der PRAM-Clock $PCLK$ benutzt.

Asynchrone Fifo-Queues

Bei einem asynchronen Fifo-Buffer wird der Lesezähler *rdcount* mit einer Leseclock $RCLK$ betrieben, während der Zähler *wrcount* und der Schreibport des dual-port-RAM mit der Schreibclock $WCLK$ betrieben werden. Die einzige echte Änderung zum synchronen Design besteht in der Berechnung der Kontrollsignale $full$ und *empty*. Das Hilfssignal *direction* wird jetzt mittels eines Latchs berechnet und zur Synchronisierung werden $full$ und *empty* registriert. Dabei werden die Register $full$ und *init* mit $WCLK$ betrieben und *empty* mit $RCLK$. Diese Schaltung ist in Abbildung 2.13 wiedergegeben.

Mit Hilfe des Registers *init* wird das Latch zur Berechnung von *direction* mit 0 initialisiert, da *init* nach einem Reset den Wert 0 hat, bis das erste *push* ausgeführt wird. Nach dem ersten *push* behält *init* den Wert 1; damit wird *direction* wie in synchronen Fall berechnet, nur in einem Latch statt einem Flip-Flop. Insbesondere ist *direction* nach den Beobachtungen beim synchronen Fall mindestens 4 Takte lang stabil, bevor $full$ oder *empty* aktiv werden können. Damit treten bei der Berechnung von $full$ und *empty* keine sogenannten *Spikes* [KP95] durch das *direction*-Signal auf.

Die Register $full$ und *empty* sind so konfiguriert, daß sie durch einen Reset beide auf 1

gesetzt werden. Außerdem liegt der Input der beiden Register jeweils auch am $PRESET$-Input. Dadurch wird beim Übergang des Inputs von 0 auf 1 das Register *asynchron* auf 1 gesetzt. Diese Vorgehensweise ist nötig, da die Register $full$ und *empty* in dem Takt aktiv werden müssen, in dem die Bedingung aktiv wird, und nicht erst einen Takt später. Das Setzen der Register per $PRESET$ ist unproblematisch, da man alle Pfade von den Zählern *durch* den $PRESET$-Eingang des Flip-Flops in die Zielregister von der Design-Software wie gewöhnliche Pfade von Register zu Register behandeln lassen kann. Diese Pfade bestimmen also mit, wie schnell ein Design geclockt werden kann.

Da *direction* nach einem Reset 0 ist, wird $full$ nach dem 1. Takt synchron auf 0 gesetzt, während *empty* den Wert 1 behält. Wird ein Datum mit der Schreibclock $WCLK$ in die Fifo geschrieben, so wird *empty* zur *nächsten* steigenden Flanke von $RCLK$ auf 0 gesetzt. Die Kontrollsignale $full$ und *empty* werden also möglicherweise erst verspätet auf 0 zurückgesetzt. Dadurch wird die Korrektheit des Fifo-Buffers nicht beeinflußt, lediglich die Performance des Designs sinkt etwas zugunsten der Synchronizität der Kontrollsignale. Die Register $full$ und *empty* werden nämlich mit der Clock betrieben, mit der sie auch ausgelesen werden–das $full$-Register mit der Schreibclock und das *empty*-Register mit der Leseclock.

Im schlimmsten Fall kann es aber trotzdem dazu kommen, daß der Input von *empty* zur steigenden Flanke von $RCLK$ gerade keinen definierten Wert hat, da er sich gerade von 1 auf 0 ändert, und das Register *empty* in einen sogenannten *metastabilen Zustand* tritt. Dann hat das Register für einen gewissen Zeitraum keinen definierten Wert mehr.

Bei den Flip-Flops in den verwendeten FPGAs dauert dieser metastabile Zustand laut [XA96] in der Regel nur wenige Nanosekunden, also deutlich weniger als ein ganzer Takt, so daß man mit diesem Register *fast* normal arbeiten kann. Die Praxis zeigt aber, daß ein Restproblem bleibt. Verwendet man die Signale $full$ und *empty* in relativ langen Pfaden und hängt der Input mehrerer Register von $full$ und *empty* ab, dann ist die korrekte Verarbeitung von $full$ und *empty* nicht in jedem Takt gewährleistet.

Um diesem Problem aus dem Weg zu gehen, werden die Register $full$ und *empty* nochmals registriert und alle Kontrollsignale mit diesen registrierten Signalen berechnet. Dadurch ist gegebenenfalls noch eine gewisse Zusatzlogik nötig, da $full$ und *empty* jetzt erst einen Takt „zu spät“ aktiv werden. In der Praxis hat sich gezeigt, daß bei direkter Verwendung von $full$ und *empty*, beispielsweise bei Lesezugriffen auf das lokale RAM der Karte, gelegentlich ein Datum *doppelt* aus einer asynchronen Fifo gelesen wurde. Bei insgesamt 16 MB Daten traten so etwa 10 bis 200 Fehler auf. Nach einem Redesign mit registrierten Signalen $full$ und *empty* verschwand dieses Problem; so wurden 16 MB Daten wiederholt fehlerfrei übertragen.

Betrachten wir die beiden asynchronen Fifo-Buffer *send* und *receive*, dann werden die $full$- und *empty*-Signale dieser Fifo-Buffer fast gar nicht benutzt. Bei der *send*-Fifo werden $full$ und *empty* beide ignoriert; es werden in der Stufe *EX* immer genau so viele Daten ausgelesen, wie in der Stufe *READ* geschrieben wurden, und die Fifo kann bei geeigneter Kontrolle nie voll werden, da höchstens 8 Daten im Burst gelesen werden. Genauso wird das Signale *receive.empty* ignoriert; es wird also lediglich das Signal $receive.full$ von der weiteren Kontrolle tatsächlich benutzt. Damit reduziert sich das Problem der asynchron berechneten Signale $full$ und *empty* deutlich.

READ		P_1		P_2		P_3		...
EX			P_1		P_2		P_3	
LAOP, *SADR*, *AUX*		P_1		P_2		P_3		...
PADR, *OP*			P_1		P_2		P_3	

Abbildung 2.14: Versetzte Arbeitsweise in der Stufe *EX*

Ein Tabelleneintrag P_i in den Zeilen 3 oder 4 besagt, daß die Register dieser Zeile die Daten des DMA-Programms P_i enthalten; in den ersten beiden Zeilen kann abgelesen werden, welches DMA-Programm in der jeweiligen Stufe ist.

2.8 DMA-Program-Environment

Das DMA-Program-Environment enthält im wesentlichen das Registerfile für DMA-Programme, das aus den Teilen *PADR*, *START* und *AUX* besteht und als RAM realisiert ist, und das *HC*-Register. Es ist gemäß Abbildung 2.15 aufgebaut. Die Register *HC*, *PADR*, *START* und *AUX* können durch Befehle des DMA-Prozessors geschrieben werden. Außerdem kann das *HC*-Register beim Befehl *rhc* ausgelesen werden, und das Bit *noHC* für das Statuswort wird berechnet.

Neben dem Registerfile für DMA-Programme als RAM enthält das *DMA-Program*-Environment noch zusätzliche Register, die das aktuelle DMA-Programm in der Stufe *READ* beziehungsweise *EX* speichern. Wie wir gleich in Anlehnung an das Pipelining aus Abschnitt 1.2.3 sehen werden, sind dabei keine getrennten Registersätze für die Stufen *READ* und *EX* nötig.

Die Stufe *READ* liest nur die Register *START* und *AUX* des Registerfiles in die Register *LAOP*, *SADR* und *AUX*. Das *PADR*-Register des Registerfiles wird erst in der Stufe *EX* in das gleichnamige Register gelesen. Da erst in der zweiten Hälfte von *EX* die Stufe *READ* wieder aktiv wird, nutzt die Stufe *EX* die Register *SADR* und *AUX* der Stufe *READ* noch in der ersten Hälfte mit. Dadurch werden 2 zusätzliche Register für die Stufe *EX* eingespart. Nur das Register *LAOP* wird in der Stufe *EX* nach *OP* durchgereicht, da es auch in der zweiten Hälfte von *EX* gebraucht wird. Abbildung 2.14 verdeutlicht diese versetzte Arbeitsweise.

In der ersten Hälfte von *EX* wird das *SADR*-Register ins Registerfile zurückgeschrieben; dabei wird die Startadresse des Restprogramms geschrieben, die im DMA-Program-Environment berechnet wird. Parallel zum Senden einer Speicheranfrage wird *PADR* um *HC* erhöht, und nach dem Versenden der letzten Speicheranfrage wird auch *PADR* ins Registerfile zurückgeschrieben.

Außerdem wird im DMA-Program-Environment noch das Flag *noHC* des Statusworts berechnet. Ein Register *eq* speichert, ob gewisse Bits im *AUX*- und im *SADR*-Register übereinstimmen. Wir benötigen dieses Register später, um zu entscheiden, ob ein DMA-Programm primitiv ist.

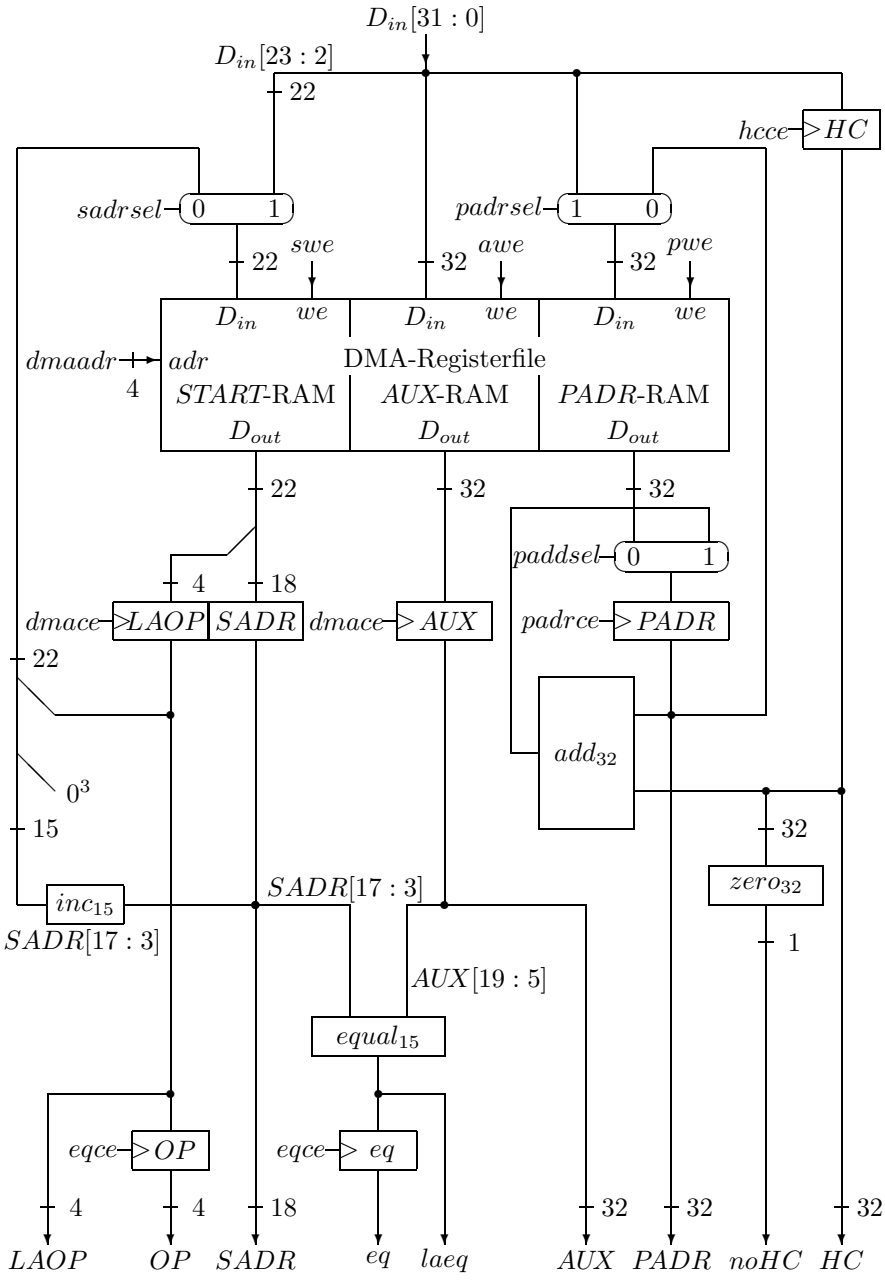


Abbildung 2.15: DMA-Program-Environment

2.8.1 Korrektheit

Bevor wir uns dem Beweisen zuwenden, daß alle noch übrigen Instruktionen korrekt ausgeführt werden, definieren wir hier noch die verwendeten Makros add_n, inc_n und $zero_n$. Sie sind alle als vordefinierte Makros von Xilinx realisiert.

Definition 2.8.1 *Ein* ***n-Addierer*** add_n *ist ein Schaltkreis mit Eingaben* $a[n-1:0]$, $b[n-1:0]$ *und Ausgaben* $s[n-1:0]$ *so, daß*

$$\langle s[n-1:0]\rangle = \langle a[n-1:0]\rangle + \langle b[n-1:0]\rangle \bmod 2^n$$

Definition 2.8.2 *Ein* ***n-Incrementer*** inc_n *ist ein Schaltkreis mit Eingaben* $a[n-1:0]$ *und Ausgaben* $s[n-1:0]$ *so, daß*

$$\langle s[n-1:0]\rangle = \langle a[n-1:0]\rangle + 1 \bmod 2^n$$

Definition 2.8.3 *Ein* ***n-Zerotester*** $zero_n$ *ist ein Schaltkreis mit Eingaben* $a[n-1:0]$ *und Ausgabe zero so, daß*

$$zero = \delta_{a[n-1:0],0^n}$$

Trivialerweise gilt in Abbildung 2.15 nach Definition des *Zerotesters* $noHC = 1 \Longleftrightarrow HC = 0^{32}$; zusammen mit Korollar 2.6.10 gilt deshalb, das das Statuswort komplett richtig berechnet wird. Jetzt können wir ein letztes Lemma zur korrekten Ausführung von Befehlen zeigen. Zusammen mit den bisherigen Lemmata folgt dann, daß alle Befehle korrekt ausgeführt werden.

Lemma 2.8.4 *Aktiviert die Kontrolleinheit die Kontrollsignale gemäß Tabelle 2.7, so werden die Befehle* rhc*,* whc*,* setpadr*,* setstart *und* setaux *nach Tabelle 2.6 korrekt ausgeführt.*

Beweis: Die Bedingung für einen Befehl gemäß Tabelle 2.6 sei erfüllt.

- *rhc*: Durch Aktivieren von $dsel1$ und $dsel0$ ist $ret = HC[31:0]$.
- *whc*: Durch Aktivieren von *hcce* wird $HC[31:0]$ vom lokalen Bus geclockt.
- *setpadr*, *setstart* und *setaux*: Nach den Bemerkungen zum Stalling auf Seite 52 wird zu dem Zeitpunkt, zu dem diese Befehle ausgeführt werden, das DMA-Program-Registerfile beim Ausführen von DMA-Programmen weder geschrieben noch gelesen. Die Befehle *setpadr*, *setstart* und *setaux* aktivieren alle das Kontrollsignal dma_sel0. Dadurch ist $dmaadr[3:0] = CH[3:0]$ mit $CH[3:0]$ aus dem Befehlswort. Außerdem ist das Schreibsignal *pwe*, *swe* beziehungsweise *awe* aktiv. Bei *setpadr* und *setstart* wird durch Aktivieren von *padrsel* beziehungsweise *sadrsel* zusätzlich das Datum vom lokalen Bus selektiert, und zwar bei *setstart* nur $D_{in}[23:2]$. Damit ergibt sich insgesamt genau der spezifizierte Effekt.

□

Aus den Lemmata 2.5.1, 2.6.11, 2.7.2, und 2.8.4 ergibt sich direkt der folgenden Satz:

Satz 2.8.5 *Aktiviert die Kontrolleinheit die Kontrollsignale gemäß Tabelle 2.7, so haben* alle *Befehle die in Tabelle 2.6 spezifizierte Wirkung.*

2.9 Ausführung von DMA-Programmen

Bisher haben wir nur angegeben, wie die Kontrollsignale auf Instruktionen hin aktiviert werden. Deshalb wenden wir uns jetzt der Aktivierung von Kontrollsignalen bei der Ausführung von DMA-Programmen zu. Da der DMA-Prozessor gemäß dem Interface zur Sortiereinheit aus Abschnitt 2.1.2 Daten an die Sortiereinheit sendet, und zwar 4 Takte lang pro Speicheranfrage, dauert das Senden einer Teilrunde aus 8 Speicheranfragen 32 Takte. Bei den Ausführungen zur Semantik von DMA-Programmen in Kapitel 1.2.1 haben wir festgestellt, daß fast alle Teilprogramme aus 8 Speicheranfragen bestehen. Deshalb haben wir festgelegt, daß das Senden einer Teilrunde *immer* 32 Takte dauert. Kürzere Teilprogramme verbleiben also gegebenenfalls mehrere Takte in der Stufe *EX*, ohne Daten an die Sortiereinheit zu senden. Wir numerieren diese 32 Takte von T_0 bis T_{31} und aktivieren die Kontrollsignale in Abhängigkeit von diesen Signalen T_i. Ein Zähler $cnt[4:0]$ gibt die Nummer des Taktes $T_{\langle cnt[4:0]\rangle}$ in der Stufe *EX* an. Dieser Zähler wird jeden Takt inkrementiert. In Tabelle 2.9 sind die aktiven Kontrollsignale in Abhängigkeit von den T_i zusammengefaßt.

Einige der Kontrollsignale aus Tabelle 2.9 werden auch als Reaktion auf einen Befehl hin aktiviert. Die vollständige Kontrollgleichung für diese Signale setzt sich dann aus zwei Teilen zusammen. Bei der Beschreibung des Instruktionssatzes haben wir darauf hingewiesen, daß Befehle gestallt werden, wenn sie auf Ressourcen zugreifen, die bei der Ausführung von DMA-Programmen gerade benutzt werden. Bedingungen für das Stalling eines Befehls sind hier die T_i, zu denen die entsprechenden Kontrollsignale bei der Ausführung von DMA-Programmen aktiviert werden. So ergibt sich beispielsweise für das Kontrollsignal *pwe* zum Schreiben der PRAM-Adresse in das Registerfile nach den Tabellen 2.6, 2.7 und 2.9 mit $act = SR[\langle CH\rangle]$ und $stat = STAT[\langle CH\rangle]$ nach Abschnitt 2.4.1 insgesamt die Kontrollgleichung

$$pwe = (setpadr \wedge act \wedge \overline{stat} \wedge \overline{T_{30}}) \vee (T_{30} \wedge valid \wedge \overline{primitiv})$$

Dabei ist $act \wedge \overline{stat}$ die Einschränkung aufgrund der Bedingungen an die Befehle aus Tabelle 2.6. Die Bedingung $\overline{T_{30}}$ sorgt gegebenenfalls dafür, daß der Befehl *setpadr* gestallt wird. Der zweite Teil der Kontrollgleichung, $T_{30} \wedge valid \wedge \overline{primitiv}$, ergibt sich direkt aus Tabelle 2.9.

Außerdem ergibt sich durch die Aktivierung der Kontrollsignale nach Tabelle 2.9 die erwähnte versetzte Arbeitsweise der Stufe *EX* aus Abbildung 2.14: In T_{15} werden mit *dmace* die Register *LAOP*, *SADR* und *AUX* aus dem DMA-Program-Registerfile gelesen. Dabei wird von der Adresse $\langle LAP[3:0]\rangle$ gelesen, da $dmasel[1:0] = 00$. Wegen der Korrektheit des Scheduling-Mechanismus ist $\langle LAP[3:0]\rangle$ zu diesem Zeitpunkt die Kanalnummer des DMA-Programms in der Stufe *READ*. Also werden die Register *LAOP*, *SADR* und *AUX* für das DMA-Programm in der Stufe *READ* gelesen.

Takt	unbedingt aktive Signale	bedingt aktive Signale
T_0	*dma_sel*1	*address.push*, *swe*
T_{13}	*mcce*, *status_sel*0, *status_sel*1	*pushback*, *status_we*
T_{14}	*la*	
T_{15}	*dmace*	
T_{16}		*pread*
T_{30}	*dma_sel*1	*pwe*
T_{31}	*eqce*, *dmapop*, *padrce*	
$T_{1 \bmod 4}$	*paddsel*	*padrce*
$T_{2 \bmod 4}$	*ssel*1	*ssel*0
$T_{3 \bmod 4}$	*ssel*1, *ece*	*send.pop*, *ssel*0

In den letzten 3 Zeilen sind Signale zusammengefaßt, die in der Stufe *EX* bei jeder Speicheranfrage aktiv werden. Dabei gilt: $T_{j \bmod 4} = \{T_i | 0 \leq i \leq 31, j \equiv i \bmod 4\}$. Die Bedingung, unter der die Signale der 3. Spalte aktiv werden, kann der folgenden Tabelle entnommen werden.

Signal	Bedingung
address.push	$valid \wedge antwort$
swe	$valid \wedge /primitiv$
pushback	$/primitiv$
status_we	$valid \wedge primitiv \wedge /antwort$
pread	$lavalid \wedge lasendet \wedge /laea$
pwe	$valid \wedge /primitiv$
padrce	$valid \wedge /ende$
*ssel*0	$valid \wedge ea$
send.pop	$valid \wedge sendet \wedge /ende \wedge /ea$

Die in der Spalte Bedingung verwendeten neuen Kontrollsignale werden wie folgt berechnet:

Bedingungen
$primitiv = eq \vee ea \vee term$
$antwort = /OP0$
$lasendet = LAOP0 \vee LAOP1$
$sendet = OP0 \vee OP1$
$ende := \begin{cases} 0 & \text{falls } ece = 1 \text{ und } cnt[4:2] = 1^3 \\ 1 & \text{falls } ece = 1 \text{ und } cnt[4:2] \neq 1^3 \text{ und } \langle cnt[4:2] \rangle \geq \langle NUM[2:0] \rangle \\ ende & \text{sonst} \end{cases}$

Tabelle 2.9: Aktive Kontrollsignale des DMA-Prozessors

In T_{31} wird OP aus $LAOP$ geclockt und $PADR$ aus dem Registerfile, da $padrce = 1$ und $paddsel = 0$. Leseadresse im Registerfile ist auch hier $\langle LAP[3:0]\rangle$. Gleichzeitig ist *dma_pop* aktiv, so daß das DMA-Programm von der Stufe *READ* in die Stufe *EX* übergeht. Damit enthalten die Register OP, $LAOP$, $SADR$, AUX und $PADR$ in T_0 die Daten des DMA-Programms in der Stufe *EX*. Nur die Register OP und $PADR$ behalten die Daten dieses DMA-Programms in der *ganzen* Stufe *EX*. Alle anderen erhalten in T_{15} die Daten des *nächsten* DMA-Programms in der Stufe *READ*.

Wir machen hier noch die Beobachtung, daß die Kontrollsignale *la*, *dma_pop* und *pushback* nach Tabelle 2.9 nur in zusammenhängender Folge aktiv werden, was wir in Korollar 2.7.6 vorausgesetzt hatten. Wir werden in diesem Kapitel zunächst auf die korrekte Berechnung des Modulo-Bits *modul* und die korrekte Umschaltung der Zahl der Teilrunden im Register $MODC$ alle 24 Teilrunden eingehen. Abschließend beweisen wir, daß Teilprogramme in allen 4 Stufen nach dem Algorithmus aus Kapitel 1.2.3 ausgeführt werden.

2.9.1 Modulo-Bit

Wir zeigen zunächst, daß das aktuelle Modulo-Bit in Abhängigkeit von der Zahl der Teilrunden $\langle MODC[1:0]\rangle + 1$ korrekt berechnet wird. Anschließend beweisen wir, daß die Zahl der Teilrunden im DMA-Prozessor nur alle 24 Teilrunden umgeschaltet werden kann; damit ist das Modulo-Bit im DMA-Prozessor auch nach Umschaltung der Zahl der Teilrunden im DMA-Prozessor und der SB-PRAM synchron.

Zur Berechnung des Modul-Bits gibt es zunächst einen Zähler $mcnt[4:0]$, der die Teilrunden modulo 24 mitzählt. Dieser Zähler wird bei aktivem *mcce* nach Tabelle 2.9 hochgezählt und ist in zwei Teilen realisiert. Während $mcnt[3:0]$ die Teilrunden modulo 12 mitzählt, wird $mcnt[4]$ alle 12 Teilrunden getoggelt. Also ist $12 \cdot mcnt[4] + \langle mcnt[3:0]\rangle$ die Nummer der Teilrunde modulo 24. Ein solcher zweigeteilter Zähler ist auf FPGA-Basis weder teurer noch langsamer als ein normaler 5-Bit-Zähler; er ermöglicht aber eine billigere und schnellere Berechnung des aktuellen Modulo-Bits (siehe unten).

$$mcnt[3:0] := \begin{cases} \mathrm{bin}_4(\langle mcnt[3:0]\rangle + 1) & \text{falls } mcce = 1 \text{ und } \langle mcnt[3:0]\rangle \neq 11 \\ 0000 & \text{falls } mcce = 1 \text{ und } \langle mcnt[3:0]\rangle = 11 \\ mcnt[3:0] & \text{sonst} \end{cases}$$

$$mcnt[4] := \begin{cases} \overline{mcnt[4]} & \text{falls } mcce = 1 \text{ und } \langle mcnt[3:0]\rangle = 11 \\ mcnt[4] & \text{sonst} \end{cases}$$

Das Signal *mcce* zum Inkrementieren dieses Zählers wird nach Tabelle 2.9 *immer* in T_{13} aktiv. Damit werden auch leere Teilrunden gezählt, wenn sich kein DMA-Programm in der Stufe *EX* befindet. Der DMA-Prozessor muß diese leeren Teilrunden auch mitzählen, da die Sortiereinheit auch für diese Runden ein *End-Of-Round*-Paket an die SB-PRAM sendet.

Dadurch, daß die Teilrunde bereits in T_{13} gezählt wird, ergibt sich eine *vorausberechnete* Zahl der Teilrunden. Wenn das Kontrollsignal *la* in T_{14} aktiv wird, muß bereits das

Modulo-Bit für das nächste Teilprogramm in der Stufe *EX* berechnet worden sein, damit der zugehörige DMA-Kanal aus der richtigen Fifo des *DMA-Schedulers* gelesen werden kann. Deshalb wird das aktuelle Modulo-Bit *modul* bei *mcce* in T_{13} gegebenenfalls getoggelt. Damit erfolgt das Umschalten von *modul* einen Takt *vor* dem Aktivieren von *la*.

$$modul := \begin{cases} \overline{modul} & \text{falls } mcce = 1 \wedge \langle mcnt[3:0] \rangle + 1 \equiv 0 \bmod (\langle MODC[1:0] \rangle + 1) \\ modul & \text{sonst} \end{cases}$$

Für das Umschalten des Modulo-Bits ist nur die Zahl der Teilrunden modulo 12 interessant, denn $12 = \mathrm{kgv}(1, 2, 3, 4)$. Da unser Zähler für Teilrunden entsprechend konstruiert ist, geht damit in die Berechnung des Modulo-Bits nur $mcnt[3:0]$ ein. Da das aktuelle Modulo-Bit eigentlich das vorausberechnete Modulo-Bit ist, wird es in Abhängigkeit von $\langle mcnt[3:0] \rangle + 1$ gesetzt, also von der Zahl der Teilrunden *nach* Inkrementieren des Zählers. Das Modulo-Bit wird genau dann umgeschaltet, wenn $\langle mcnt[3:0] \rangle + 1$ ein Vielfaches der Zahl der Teilrunden $\langle MODC[1:0] \rangle + 1$ ist.

Die Gleichung für *modul* läßt sich in konkrete Hardware umsetzen, indem man die 4 möglichen Fälle für $MODC[1:0]$ getrennt betrachtet. Der Fall $\langle MODC[1:0] \rangle + 1 = 1$ ist dabei trivial; das Modulo-Bit wird in diesem Fall nach *jeder* Teilrunde getoggelt.

$$\begin{aligned} &\langle mcnt[3:0] \rangle + 1 \equiv 0 \bmod 2 \Longleftrightarrow mcnt[0] = 1 \\ &\langle mcnt[3:0] \rangle + 1 \equiv 0 \bmod 3 \Longleftrightarrow mcnt[3:0] \in \{0010, 0101, 1000, 1011\} \\ &\langle mcnt[3:0] \rangle + 1 \equiv 0 \bmod 4 \Longleftrightarrow mcnt[1:0] = 11 \end{aligned}$$

Der Schaltkreis, der mit obigem Kriterium prüft, ob $\langle mcnt[3:0] \rangle + 1 \equiv 0 \bmod 3$, berechnet eine 4-stellige binäre Funktion. Damit läßt sich diese Funktion in *einem einzigen* Funktionsgenerator realisieren, wie in Abschnitt 1.4 erwähnt. Hätten wir den Zähler *mcnt* auf die naheliegende Weise realisiert, dann wäre der Schaltkreis an dieser Stelle teurer und langsamer geworden. Es hätte sich dann um eine 5-stellige Funktion gehandelt, die ein ganzes CLB belegt.

Insgesamt ergibt sich damit ein Lemma:

Lemma 2.9.1 *Zum dem Zeitpunkt, wo la aktiviert wird und ein DMA-Programm die Stufe* READ *betritt, enthält das Register modul das korrekt vorausberechnete Modulo-Bit.*

Beim Befehl *setmod* wird die Zahl der Teilrunden in ein Register $NMODC$ geclockt. Das Register $MODC$ wird nur alle 24 Teilrunden aus $NMODC$ geclockt, damit die Umschaltung des Modulo-Bits synchron zur PRAM erfolgen kann. Dazu wird das Kontrollsignal *modce* zum Clocken des $MODC$-Registers in Abbildung 2.7 in Abhängigkeit von $mcnt[4:0]$ aktiviert.

$$modce = \begin{cases} 1 & \text{falls } mcce = 1 \text{ und } \langle mcnt[3:0] \rangle = 11 \text{ und } mcnt[4] = 1 \\ 0 & \text{sonst} \end{cases}$$

Damit ergibt sich mit Hilfe des vorangehenden Lemmas ohne weiteren Beweis der nachfolgende Satz.

Satz 2.9.2 *Die Umschaltung für die Zahl der Teilrunden einer Netzwerkrunde im Register MODC des DMA-Prozessors erfolgt nur alle 24 Teilrunden, und das Modulo-Bit modul wird in Abhängigkeit von der Zahl der Teilrunden korrekt vorausberechnet.*

2.9.2 Stufe *READ*

Wir wollen jetzt zeigen, daß die Stufe *READ* eine Implementierung des Algorithmus aus Kapitel 1.2.3 ist. Dazu greifen wir hier zunächst nochmals den Algorithmus auf.

Algorithmus Stufe *READ*

```
P    = DMAScheduler.pop();  /* Kanalnummer auslesen */
EA   = DMAScheduler.EA;     /* merken, ob Einzelanweisung */
SADR = DMAPRG[P].sadr;      /* DMA-Programm (bis auf padr) lesen */
EADR = DMAPRG[P].eadr;
DATA = DMAPRG[P].data;
OP   = DMAPRG[P].op;
primitiv = SADR/8==EADR/8;  /* primitiv nach Definition berechnen*/
EADR'= primitiv? EADR: 8*(SADR/8)+7;  /* EADR' nach Def. berechnen */
if (!EA && OP1!=LOAD)       /* Standardanweisung, die Daten liest? */
{
    while(RAM_Zugriff)      /* WB gerade aktiv? */
    {}                      /* warten, bis WB beendet */
    RAM_Zugriff=1;          /* RAM-Zugriff beginnen */
    for (adr=SADR; adr<=EADR'; adr++)
    {
        d = M(adr);         /* Lesezugriff auf lokales RAM */
        send.push(d);       /* Datum in send-Fifo schreiben */
    }
    RAM_Zugriff=0;          /* RAM-Zugriff beenden */
}
```

Nach den Bemerkungen zur versetzten Arbeitsweise im Abschnitt 2.9 wird das DMA-Programm beim Übergang in die Stufe *READ* korrekt in die Register *SADR*, *AUX* und *LAOP* geschrieben. Dabei enthält das *AUX*-Register die beiden Teile `EADR` oder `DATA` von obigen Algorithmus und *LAOP* Modus und Operator `OP`. Das Register *laea* des DMA-Schedulers aus Abbildung 2.11 enthält das `EA`-Flag aus obigen Algorithmus. Damit bleibt noch zu zeigen, daß die Endadresse `EADR'` aus obigem Algorithmus im DMA-Prozessor korrekt berechnet wird und genau dann ein RAM-Zugriff erfolgt, wenn der Algorithmus auf das RAM zugreift. In diesem Fall werden alle gelesenen Daten in die *send*-Fifo geschrieben.

Wir spezifizieren jetzt noch die restlichen Signale der Stufen *READ* und *EX*, bevor wir uns der Korrektheit der Stufe *READ* zuwenden. In $EADR'$ stehen die 3 untersten Bits der Endadresse eines Teilprogramms in der Stufe *READ*, im Register NUM wird die um 1 verminderte Länge des Teilprogramms in der Stufe *EX* gespeichert. Damit kann NUM

als 3-Bit-Register realisiert werden.

$$EADR'[2:0] = \begin{cases} SADR[2:0] & \text{falls } laea = 1 \\ AUX[4:2] & \text{falls } laea = 0 \text{ und } laeq = 1 \\ 111 & \text{falls } laea = 0 \text{ und } laeq = 0 \end{cases}$$

$$NUM[2:0] := \begin{cases} \text{bin}_3(\langle EADR'[2:0]\rangle - \langle SADR[2:0]\rangle) & \text{falls } eqce = 1 \\ NUM[2:0] & \text{sonst} \end{cases}$$

Betrachten wir zunächst die Bedingungen *primitv*, *antwort*, *sendet* und *lasendet* aus Tabelle 2.9. Wegen der Kodierung von Modus und Operator eines Zugriffs nach Tabelle 2.3 ist $antwort = 1$ genau dann, wenn das DMA-Programm in der Stufe *EX* entweder den Modus *LOAD* oder *MP* hat, also eine *Antwort* von der SB-PRAM erwartet. Analog ist $lasendet = 1$, wenn das DMA-Programm in der Stufe *READ* Daten *sendet* beziehungsweise $sendet = 1$, wenn das Programm in der Stufe *EX* Daten sendet.

Weiter ist $primitiv = 1$ genau dann, wenn das DMA-Programm in der Stufe *EX* primitiv ist, oder wegen $term = 1$ infolge eines Befehls *enddma* so behandelt wird, als ob es primitiv ist. Ein DMA-Programm in der Stufe *EX* ist nämlich primitiv, wenn es eine Einzelanweisung ist ($ea = 1$) oder nach Definition 1.2.7 eines Teilprogramms auf Seite 11 $[sadr]_8 = [eadr]_8$ gilt, also die Start- und Endadresse bis auf die untersten 3 Bit übereinstimmen. Genau diese Bedingung wird mit dem Signal *laea* aus dem *DMA-Program*-Environment aus Abbildung 2.15 überprüft. Mit diesen Vorbemerkungen können wir zeigen, daß $EADR'$ und NUM korrekt berechnet werden.

Lemma 2.9.3 *Sei $P = (op, padr, sadr, eadr, data)$ ein DMA-Programm mit einem Teilprogramm $P' = (op, padr, sadr, eadr', data)$, und sei $n' = l(P')$ die Länge des Teilprogramms. Ist P in der Stufe* READ *oder in T_0 der Stufe* EX*, dann gilt für die Endadresse des Teilprogramms $eadr' = \langle LAP[3:0]SADR[17:3]EADR'[2:0]\rangle$. Ist P in der Stufe* EX*, dann gilt für die Länge des Teilprogramms P' zusätzlich noch $n' = \langle NUM[2:0]\rangle + 1$.*

Beweis: Nach Korollar 1.2.9 auf Seite 12 ist

$$eadr' = \begin{cases} eadr & \text{falls } P \text{ primitiv} \\ [sadr]_8 + 7 & \text{sonst} \end{cases}$$

Da P nach Definition 1.2.7 primitiv ist, wenn $[sadr]_8 = [eadr]_8$, gilt mit obiger Formel $[eadr']_8 = [sadr]_8$. Damit ist $[eadr']_8 = \langle LAP[3:0]SADR[17:3]000\rangle$, denn $LAP[3:0]$ enthält die Kanalnummer der Stufe *READ*, die zusammen mit der relativen Adresse $\langle SADR\rangle$ im RAM des Kanals die absolute Adresse im SDRAM ergibt. Also ist für $eadr'$ nur noch zu zeigen, daß $eadr' = \langle EADR'[2:0]\rangle \bmod 8$.

In T_0 der Stufe *EX* enthalten die Register $SADR$ und AUX noch die Werte, die sie in der vorangehenden Stufe *READ* hatten (siehe versetzte Arbeitsweise). Nach der Spezifikation des DMA-Schedulers gilt das auch für das Register *laea*. Damit hat $EADR$ in T_0 der Stufe *EX* denselben Wert wie in der vorangehenden Stufe *READ*, da es nur von Signalen abhängt, die sich in diesem Zeitraum nicht geändert haben. Nach der Spezifikation für

$EADR'$ gilt dann:

$$EADR'[2:0] = \begin{cases} SADR[2:0] & \text{falls } P \text{ eine Einzelanweisung} \\ AUX[4:2] & \text{falls } P \text{ primitive Standardanweisung} \\ 111 & \text{sonst} \end{cases}$$

Falls P eine Einzelanweisung ist, so gilt $\langle LAP[3:0]SADR[17:0]\rangle = sadr = eadr$, und sonst gilt $\langle LAP[3:0]AUX[19:2]\rangle = eadr$. Also lassen sich die beiden oberen Fälle zusammenfassen und es folgt die Behauptung

$$\langle EADR'[2:0]\rangle = \begin{cases} eadr \bmod 8 & \text{falls } P \text{ primitiv} \\ 7 & \text{sonst} \end{cases}$$

Das Register $NUM[2:0]$ wird beim Übergang von der Stufe *READ* in die Stufe *EX* in Takt T_{31} geclockt. Zu diesem Zeitpunkt gilt nach den gerade angestellten Betrachtungen für die Endadresse des Teilprogramms $eadr' = \langle LAP[3:0]SADR[17:3]EADR'[2:0]\rangle$. Damit ergibt sich sofort auch die Behauptung für NUM und die Länge n' des Teilprogramms:

$$\begin{array}{lll} n' & = eadr' - sadr + 1 & \text{(Definition 1.2.7)} \\ & = eadr' - sadr + 1 \bmod 8 & (0 \le n' \le 8) \\ & = \langle EADR'[2:0]\rangle - \langle SADR[2:0]\rangle + 1 & \text{(gezeigte Behauptung für } EADR') \\ & = \langle NUM[2:0]\rangle + 1 & \text{(Spezifikation von } NUM) \end{array}$$

□

Nach Tabelle 2.9 wird das Signal *pread* zum Lesen von Daten aus dem lokalen RAM aktiv, wenn $lavalid \wedge lasendet \wedge /laea$ gilt. Wir wollen deshalb im folgenden zeigen, daß diese Bedingung genau dann erfüllt ist, wenn nach obigen Algorithmus Daten aus dem lokalen RAM gelesen werden.

Durch das Signal *pread* wird das RAM-Interface dazu veranlaßt, die Daten von Adresse $\langle LAP[3:0]SADR[17:0]\rangle$ bis $\langle LAP[3:0]SADR[17:3]EADR'[2:0]\rangle$ aus dem lokalen RAM zu lesen und in die *send*-Fifo zu schreiben. Falls gerade ein RAM-Zugriff der Stufe *WB* erfolgt, wartet das RAM-Interface, bis dieser Zugriff beendet worden ist, bevor die Stufe *READ* auf das RAM zugreifen kann. Damit bleibt nur noch zu zeigen, daß die Bedingung für *read* richtig kodiert ist und die angegebene Start- und Endadresse für den RAM-Zugriff korrekt sind.

Satz 2.9.4 *Sei P ein DMA-Programm in der Stufe* READ*. Dann wird pread genau dann aktiviert, wenn P Daten aus dem lokalen RAM liest. In diesem Fall sind Start- und Endadresse des Lesezugriffs genau die absolute Start- und Endadresse des Teilprogramms. Insgesamt arbeitet die Stufe* READ *also korrekt, wenn das RAM-Interface korrekt arbeitet.*

Beweis: Sei $P' = (op, padr, sadr, eadr', data)$ das Teilprogramm in der Stufe *READ*. Aus Tabelle 2.9 folgt direkt, daß *pread* genau dann aktiv wird, wenn P eine Standardanweisung ist, die Daten sendet. In diesem Fall ist $sadr = \langle LAOP[3:0]SADR[17:0]\rangle$ und nach Lemma 2.9.3 ist auch $eadr' = \langle LAOP[3:0]SADR[17:3]EADR'[2:0]\rangle$. Also werden genau die richtigen Daten aus dem lokalen RAM gelesen und in die *send*-Fifo geschrieben, wenn das lokale RAM-Interface korrekt arbeitet. □

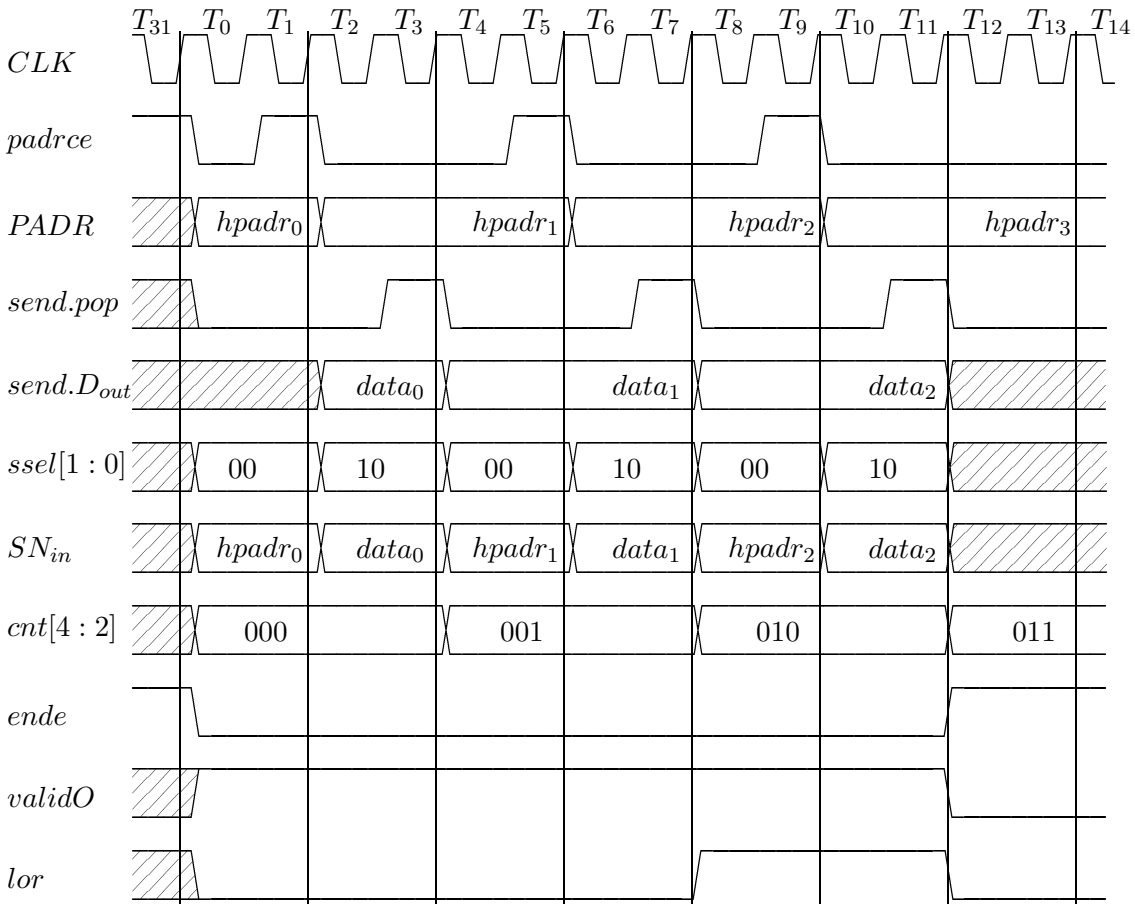


Abbildung 2.16: Ausführung eines Teilprogramms der Länge 3 in der Stufe *EX*

In den Zeilen *PADR* und SN_{in} steht $hpadr_i$ für $hpadr + i \cdot hc \bmod 2^{32}$

2.9.3 Stufe *EX*

In der Stufe *EX* werden die Speicheranfragen eines Teilprogramms nach dem Protokoll aus Abbildung 2.2 auf Seite 39 an die Sortiereinheit geschickt. Außerdem wird das DMA-Programm gegebenenfalls reschedult und gegebenenfalls wird ein Write-Info in die *address*-Fifo geschrieben. Zusätzlich wird das *STAT*-Register auf 0 zurückgesetzt, wenn das Teilprogramm in der Stufe *EX* primitiv ist. In Abbildung 2.16 ist das Versenden von Speicheranfragen der Stufe *EX* anhand eines Beispiels zusammengefaßt.

Die Kontrollsignale $ssel[1:0]$ werden so aktiviert, daß in Takten $T_{0 \bmod 4}$ und $T_{1 \bmod 4}$ die Adresse einer Speicheranfrage an der Sortiereinheit anliegt und in $T_{2 \bmod 4}$ und $T_{3 \bmod 4}$ das Datum. Dieses Datum kommt von der *send*-Fifo, wenn $ea = 0$, sonst aus dem *AUX*-Register. Wenn das Datum aus der *send*-Fifo kommt, wird es im Anschluß an das Versen-

den an die Sortiereinheit mit *send.pop* aus der *send*-Fifo ausgelesen, falls das Teilprogramm Daten sendet. Sonst wird als Dummy-Datum $d = \#$ das Datum am Ausgang der *send*-Fifo versendet, ohne es aus der Fifo auszulesen.

Nach dem Senden der Adresse einer Speicheranfrage wird das $PADR$-Register geclockt, wodurch wegen $paddsel = 1$ in Abbildung 2.15 die nächste gehashte PRAM-Adresse ins $PADR$-Register geschrieben wird. Damit liegt eine Folge von Speicheranfragen an der Sortiereinheit an. Das Signal $validO$ ist genau dann aktiv, wenn eine gültige Speicheranfrage an die Sortiereinheit geschickt wird und das Signal lor ist aktiv, wenn die letzte Speicheranfrage eines Teilprogramms anliegt. Das Signal $VPlast$ für die Sortiereinheit ist aktiv, wenn die letzte Speicheranfrage einer Teilrunde anliegt. Das ist genau dann der Fall, wenn die letzte Speicheranfrage eines Teilprogramms anliegt. Für diese drei Signale gilt:

$$\begin{aligned} validO &= valid \wedge /ende \\ lor &= \delta_{NUM[2:0],cnt[4:2]} \\ VPlast &= \delta_{NUM[2:0],cnt[4:2]} \end{aligned}$$

Nach den Bemerkungen zur versetzten Arbeitsweise im *DMA-Program*-Environment wird die PRAM-Adresse korrekt in T_{31} ins Register $PADR$ geschrieben. Wir haben bereits gezeigt, daß die Länge des Teilprogramms in EX nach Lemma 2.9.3 $n' = \langle NUM[2:0] \rangle + 1$ ist. Für die übrigen Beweise greifen wir zunächst den Algorithmus der Stufe EX wieder auf.

Algorithmus Stufe *EX*

```
PADR     = DMAPRG[P].padr;       /* PADR lesen */
n'       = EADR' - SADR + 1;
if (OP1==LOAD || OP1==MP)        /* erwartet eine Antwort? */
    address.push(primitiv,SADR,EADR',P);
if (!primitiv)                   /* nicht primitiv? */
{
    DMAPRG[P].sadr=SADR + n';    /* sadr zurueckschreiben */
    DMAScheduler.push(P);        /* und Kanal reschedulen */
}
for (i=0; i<n'; i++)
{
    if (EA)
        d = DATA;                /* Datum bei EA ist DATA */
    else
        d = OP1==LOAD? #: send.pop();
        /* bei Standardanweisung aus send-Fifo lesen, */
        /* falls Daten gesendet werden, sonst # */
    lor = (i==n'-1);             /* lor bei letzter Anfrage */
    Send_Request(OP,PADR,d,lor); /* Speicheranfrage und lor senden */
    PADR = PADR + HC;            /* PADR um HC erhoehen */
}
if (!primitiv)
```

```
    DMAPRG[P].padr = PADR;       /* padr zurueckschreiben */
if (primitiv && (OP1==STORE || OP1==SYNC))
    /* primitives Programm, das keine Antwort erwartet? */
{
    STAT_Zugriff=1;              /* Zugriff auf STAT beginnen */
    STAT[P]=0;                   /* Programm beendet */
    STAT_Zugriff=0;              /* Zugriff auf STAT beenden */
}
```

Wir zeigen zunächst, daß das Teilprogramm die korrekte Folge von Speicheranfragen an die Sortiereinheit produziert.

Lemma 2.9.5 *Sei P ein DMA-Programm in der Stufe* EX *mit einem Teilprogramm P' der Länge n', und die Kontrollsignale seien wie in Tabelle 2.9 aktiv. Falls P eine Standardanweisung ist, die Daten sendet, dann enthalte die* send*-Fifo zum Zeitpunkt T_2 zumindest das erste Datum, das P sendet, und das RAM-Interface arbeite korrekt. Dann liegen an SN_{in} nacheinander genau die n' Speicheranfragen des Teilprogramms an. Genau für diese n' Speicheranfragen ist validO aktiv und lor ist genau bei der n'-ten Speicheranfrage aktiv.*

Beweis: Im Algorithmus der Stufe *EX* wird nur dann ein Datum mit `send.pop()` aus der *send*-Fifo gelesen, wenn das zugehörige DMA-Programm eine Standardanweisung ist, die Daten sendet, wenn das zugehörige Programm in der Stufe *READ* also auch Daten in diese Fifo geschrieben hat. In diesem Fall werden genau $n' = \langle NUM[2:0] \rangle + 1$ Daten gelesen, nach Lemma 2.9.3 also genau so viele Daten, wie in der Stufe *READ* geschrieben wurden, falls das RAM-Interface korrekt arbeitet. Enthält die *send*-Fifo zum Zeitpunkt T_2 zumindest das erste Datum, das P sendet, dann liefert das RAM-Interface in jedem folgenden Takt ein weiteres Datum und schreibt es in die *send*-Fifo. Da nur alle 4 Takte ein Datum aus der *send*-Fifo ausgelesen wird, ist damit *immer* ein Datum in der *send*-Fifo, wenn ein Datum ausgelesen werden soll. Wenn das RAM-Interface korrekt arbeitet, handelt es sich also immer um das richtige Datum.

Mit Hilfe des *ea*-Registers des DMA-Schedulers wird das korrekte Datum einer Speicheranfrage an die Sortiereinheit angelegt, nämlich das Datum am Ausgang der *send*-Fifo bei Standardanweisungen oder das Datum aus dem *AUX*-Register bei Einzelanweisungen. Durch Aktivieren von *padrce* alle 4 Takte wird die gehashte PRAM-Adresse sukzessive um die Hashkonstante inkrementiert; damit liegt auch die korrekte PRAM-Adresse an der Sortiereinheit an. Modus und Operator der Speicheranfrage werden dem OP-Register entnommen, daß nur am Ende der Stufe *EX* in T_{31} geclockt wird und damit in der ganzen Stufe *EX* den korrekten Wert enthält.

Da nach Lemma 2.9.3 gilt, daß $n' = \langle NUM[2:0] \rangle + 1$ ist, wird das Kontrollsignale *ende* nach der n'-ten Speicheranfrage aktiv; damit ist *validO* genau für die ersten n' Speicheranfragen der Stufe *EX* aktiv. Das Signal *lor* wird genau bei der letzten dieser Speicheranfragen aktiv. □

Für die korrekte Arbeitsweise der Stufe *EX* bleibt damit noch zu zeigen, daß es korrekt reschedult wird, daß also das Restprogramm eines DMA-Programms korrekt zurück-

geschrieben wird, daß außerdem die Write-Infos korrekt geschrieben werden und daß das Status-Register gegebenenfalls auf 0 zurückgesetzt wird.

Lemma 2.9.6 *Sei $P = (op, padr, sadr, eadr, data)$ ein DMA-Programm in der Stufe* EX *mit gehashter PRAM-Adresse $hpadr = padr \odot hc$. Wenn die Kontrollsignale wie in Tabelle 2.9 aktiv werden, dann werden die Parameter sadr und hpadr des DMA-Programms P korrekt zurückgeschrieben. Insbesondere wird sadr zurückgeschrieben, bevor das nächste DMA-Programm die Stufe* READ *betritt und sadr liest. Der Parameter hpadr wird zurückgeschrieben, bevor das nächste DMA-Programm der Stufe* EX *seinerseits hpadr wieder liest.*

Beweis: Sei $P = (op, padr, sadr, eadr, data)$ das DMA-Programm, das gerade in der Stufe *EX* ist, und sei n' die Länge des Teilprogramms von P. Falls P primitiv ist oder mit dem Befehl *enddma* beendet wurde ($term = 1$), werden nach Tabelle 2.9 wegen $swe = pwe = 0$ weder *sadr* noch *padr* zurückgeschrieben. Wir zeigen zunächst, daß im Falle eines Zurückschreibens als relative Startadresse im lokalen RAM $[relsadr]_8 + 8$ und als gehashte PRAM-Adresse $hpadr + n' \cdot hc \bmod 2^{32}$ ins Registerfile zurückgeschrieben werden. Es gilt:

$$[relsadr]_8 + 8 = \langle SADR[17:3]000\rangle + 8 = 8 \cdot (\langle SADR[17:3]\rangle + 1)$$

Mit Hilfe des Incrementers im DMA-Program-Environment nach Abbildung 2.15 auf Seite 73 wird also genau $[relsadr]_8 + 8$ berechnet und dieses Datum wird nach Tabelle 2.9 durch *swe* bei gleichzeitig inaktivem *sadrsel* zurückgeschrieben. Zugriffsmodus und -operator aus *LAOP* werden ebenfalls zurückgeschrieben, um ein zusätzliches *WE*-Signal zu sparen. Wegen der versetzten Arbeitsweise der Stufe *EX* enthält *LAOP* zu diesem Zeitpunkt noch die Daten des DMA-Programms in der Stufe *EX*. Das Schreibsignal *swe* ist in T_0 aktiv und damit vor T_{14}, wo wegen *la* das nächste DMA-Programm die Stufe *READ* betritt und in T_{15} erst das *SADR*-Register liest. Damit wird das *SADR*-Register mit dem korrekten Wert zurückgeschrieben, bevor es wieder gelesen wird.

Nach Lemma 2.9.3 ist $n' = \langle num[2:0]\rangle + 1$ Das Signal *padrce* ist nach Tabelle 2.9 in $T_{1 \bmod 4}$ aktiv, solange *ende* nicht aktiv ist, also insgesamt genau n' mal. Dabei ist *padrce* in T_{30} und T_{31} auf keinen Fall aktiv. In T_{30} ist deshalb schon $\langle PADR[31:0]\rangle = hpadr + n' \cdot hc \bmod 2^{32}$, da in T_0 gilt $hpadr = \langle PADR[31:0]\rangle$. Dieses Datum wird mit *pwe* in T_{30} ins Registerfile zurückgeschrieben. Das DMA-Programm, das als nächstes von der Stufe *READ* in die Stufe *EX* übergeht, liest das *PADR*-Register in T_{31}, also in jedem Fall einen Takt *nach* dem letzten Zurückschreiben. □

Als nächstes befassen wir uns mit der Korrektheit der Write-Infos. Ein korrektes Write-Infos wird also genau dann in die *address*-Fifo geschrieben, wenn das Teilprogramm eine Antwort erwartet.

Lemma 2.9.7 *Sei P ein DMA-Programm in der Stufe* EX *und die Kontrollsignale seien wie in Tabelle 2.9 aktiv. Dann wird das Write-Info genau dann geschrieben, wenn P eine Antwort erwartet. In diesem Fall enthält es die korrekte Start- und Endadresse des Teilprogramms von P, die korrekte Kanalnummer von P und das korrekte primitiv-Flag von P.*

Beweis: Das Kontrollsignal *address.push* wird nach Tabelle 2.9 nur dann aktiv, wenn sich ein DMA-Programm in der Stufe *EX* befindet, das eine Antwort erwartet. In diesem Fall werden $P[3:0]$, $SADR[17:0]$, $EADR'[2:0]$ und *primitiv* in die *address*-Fifo geschrieben. Dabei ist $\langle P[3:0]SADR[17:0]\rangle$ die Startadresse des Teilprogramms und nach Lemma 2.9.3 ist auch $\langle P[3:0]SADR[17:3]EADR'[2:0]\rangle$ die Endadresse des Teilprogramms. Zu diesem Zeitpunkt hat $EADR'$ nämlich immer noch denselben Wert, den es hatte, als das zugehörige DMA-Programm in der Stufe *READ* war. Zuletzt wird noch das *primitiv*-Flag in die Fifo geschrieben. Dieses Flag ist nach den bereits angestellten Betrachtungen genau dann aktiv, wenn das DMA-Programm primitiv ist oder $term = 1$ gilt, also der Befehl *enddma* für diesen DMA-Kanal aufgerufen wurde. □

Als letztes zeigen wir noch, daß das *STAT*-Register genau wie im Algorithmus angegeben zurückgesetzt wird und das DMA-Programm entsprechend reschedult wird.

Lemma 2.9.8 *Sei P ein DMA-Programm in der Stufe* EX *und die Kontrollsignale seien wie in Tabelle 2.9 aktiv. Dann wird das Status-Register für P genau dann auf 0 zurückgesetzt, wenn P keine Antwort erwartet und entweder primitiv ist oder mit dem Befehl* enddma *beendet worden ist. P wird genau dann reschedult, wenn P nicht primitiv ist und nicht mit dem Befehl* enddma *beendet worden ist.*

Beweis: Nach Tabelle 2.9 wird das Status-Register in T_{13} mit *status_we* geschrieben, wenn ein primitives Teilprogramm keine Antwort erwartet oder ein Programm, das mit *enddma* beendet wurde, keine Antwort erwartet. In T_{13} gilt $status\_sel[1:0] = 11$. Nach den Bemerkungen über Stalling aus Kapitel 2.4.2 schreibt gleichzeitig kein Befehl das Status-Register. Damit ist $status\_in = 0$. Also wird nach Abbildung 2.9 auf Seite 58 genau $STAT[\langle P[3:0]\rangle] = 0$ gesetzt.

Das Kontrollsignal *pushback* wird genau dann in T_{13} aktiviert, wenn P nicht primitiv ist oder mit *enddma* beendet worden ist. Also wird ein primitives P nicht reschedult. □

Aus den Lemmata 2.9.5, 2.9.6, 2.9.7 und 2.9.8 folgt der Satz über die Korrektheit der Stufe *EX*:

Satz 2.9.9 *Sei P ein DMA-Programm in der Stufe* EX *und die Kontrollsignale seien wie in Tabelle 2.9 aktiv. Das RAM-Interface arbeite korrekt, und falls P eine Standardanweisung ist, die Daten sendet, enthalte die* send*-Fifo zum Zeitpunkt T_2 zumindest das erste Datum, das P sendet. Dann arbeitet die Stufe* EX *korrekt.*

2.9.4 Stufe *PRAM*

Für die Stufe *PRAM* haben wir in Kapitel 1.2.3 den nachfolgenden Algorithmus spezifiziert. Wir zeigen im Anschluß, daß wir diesen Algorithmus korrekt implementiert haben.

Algorithmus Stufe *PRAM*

```
(package,err,dat)=Get_Answer();  /* Antwort und package-Flag lesen */
receive.push(err,dat)            /* Antwort in receive-Fifo schreiben */
if (package)                     /* letzte Antwort von Teilprogramm? */
    pcnt++;                      /* pcnt inkrementieren */
```

Die Stufe *PRAM* zählt in einem Register *pcnt* vollständige Teilrunden, die sich in der *receive*-Fifo befinden. Der Zähler wird genau dann hochgezählt, wenn die letzte Antwort einer Teilrunde eingetroffen ist. Diesen Sachverhalt kann die Stufe *PRAM* anhand des Flags *package* von der Sortiereinheit feststellen. Der Zähler wird dekrementiert, wenn eine Teilrunde in der Stufe *WB* ins RAM geschrieben worden ist. Dazu aktiviert *WB* ein Kontrollsignal $pwrite\_res$. Die Stufe *PRAM* aktiviert ein Kontrollsignal *pwrite*, falls mindestens eine Teilrunde in der *receive*-Fifo steht, also wenn $\langle pcnt \rangle \neq 0$.

$$pcnt[4:0] := \begin{cases} \text{bin}_5(\langle pcnt[4:0] \rangle + 1) & \text{falls } package = 1 \text{ und } pwrite\_res = 0 \\ \text{bin}_5(\langle pcnt[4:0] \rangle - 1) & \text{falls } package = 0 \text{ und } pwrite\_res = 1 \\ pcnt[4:0] & \text{sonst} \end{cases}$$

$$pwrite := \begin{cases} 1 & \text{falls } \langle pcnt[4:0] \rangle \neq 0 \\ 0 & \text{sonst} \end{cases}$$

Das $full$-Signal der *receive*-Fifo wird als *FifoFull*-Eingabe an die Sortiereinheit weitergegeben. Da die Sortiereinheit als Reaktion auf ein aktives Signal *FifoFull* keine Antworten mehr an den DMA-Prozessor sendet, gehen keine Antworten der SB-PRAM wegen voller *receive*-Fifo verloren. Außerdem kann *pcnt* als 5-Bit-Zähler realisiert werden, da die *receive*-Fifo höchstens 16 Antworten aufnehmen kann, also schlimmstenfalls die Antworten von 16 Teilprogrammen der Länge 1. Eine 4-Bit-Zähler reicht für *pcnt* nicht aus, da der Zähler 17 verschiedene Zustände speichern können muß: Die Daten von 0 bis einschließlich 16 Teilprogrammen können in der *receive*-Fifo stehen.

Aus dieser Implementierung des DMA-Prozessors und dem Algorithmus der Stufe *PRAM* folgt damit unmittelbar der folgende Satz.

Satz 2.9.10 *Die Sortiereinheit arbeite korrekt, liefere also insbesondere genau dann mit einer Antwort der SB-PRAM ein aktives package-Flag, wenn bei der zugehörigen Speicheranfrage das lor-Bit aktiv war. Dann arbeitet die Stufe* PRAM *korrekt, denn in pcnt werden die Teilprogramme, die die Stufe* WB *noch zu betreten haben, korrekt gezählt, und alle Antworten werden in der* receive*-Fifo gespeichert.*

2.9.5 Stufe *WB*

In der Stufe *PRAM* werden die Daten eines Teilprogramms aus der *receive*-Fifo gelesen und an die Adresse abgespeichert, die aus der *address*-Fifo ausgelesen wird. Gegebenenfalls wird das $ERROR$-Register gesetzt beziehungsweise das $STAT$-Register zurückgesetzt. Wir greifen hier zunächst den Algorithmus für die Stufe *WB* aus Kapitel 1.2.3 nochmals auf und zeigen dann, daß dieser Algorithmus korrekt implementiert wurde.

Algorithmus Stufe *WB*

```
if (pcnt!=0)
{
    while(RAM_Zugriff)          /* READ gerade aktiv? */
    {}                          /* warten, bis READ beendet */
    RAM_Zugriff=1;              /* RAM-Zugriff beginnen */
```

Signal	Bedingung
address.pop	*pwrite_res*
status_we	$(pwrite\_end \wedge /T_{13} \vee stall\_wb) \wedge PRIMITIV\_WB$
*status_sel*1	$(pwrite\_end \wedge /T_{13} \vee stall\_wb) \wedge PRIMITIV\_WB$
error_we	$pwrite\_end \wedge sn\_error$
error_sel	*pwrite_end*

Tabelle 2.10: Aktive Kontrollsignale der Stufe *WB*

```
    (primitiv_wb,SADR_wb,EADR'_wb,P_wb)=address.pop();
        /* Write-Info aus address-Fifo lesen */
    sn_error=0;                  /* sn_error initialisieren */
    pcnt--;                      /* pcnt dekrementieren */
    for (adr=SADR_wb; adr<=EADR'_wb; adr++)
    {
        (error,d)=receive.pop(); /* Antwort aus receive-Fifo lesen */
        M(adr) = d;              /* Datum speichern */
        if (error)
            sn_error=1;          /* Fehler akkumulieren */
    }
    RAM_Zugriff=0;               /* RAM-Zugriff beenden */
    if (sn_error)                /* Fehler aufgetreten? */
        ERROR[P_wb] = 1;         /* ERROR-Flag setzen */
    if (primitiv_wb)             /* primitives Teilprogramm? */
    {
        while (STAT_Zugriff)     /* warten, bis STAT-Register frei */
        {}
        STAT[P_wb] = 0;          /* Programm beendet */
    }
}
```

Bisher haben wir aktive Kontrollsignale in der Stufe *WB* noch nicht spezifiziert. In Tabelle 2.9 sind nur die Kontrollsignale der Stufen *READ* und *EX* zusammengefaßt. Die Signale von *WB* können hier nicht mit aufgenommen werden, da die Stufe *WB* nur dann aktiv wird, wenn die Daten einer Teilrunde in der *receive*-Fifo stehen und gerade kein RAM-Zugriff erfolgt. Die Kontrollsignale der Stufe *WB* werden also nicht in Abhängigkeit von den T_i aktiv.

In Tabelle 2.10 sind die aktiven Kontrollsignale und die jeweils zugehörigen Bedingungen zusammengefaßt. Die triviale Bedingung, daß sich jeweils überhaupt ein Teilprogramm in der Stufe *WB* befindet, ist dabei schon in *pwrite_res* und *pwrite_end* kodiert. Man kann erkennen, daß die Signale der Stufe *WB* entweder als Reaktion auf *pwrite_res* oder *pwrite_end* aktiviert werde. Das RAM-Interface aktiviert *pwrite_res* als Reaktion auf *pwrite*, wenn es Start- und Endadresse des Teilprogramms in der Stufe *WB* gelesen hat und der RAM-Zugriff begonnen hat. Analog wird *pwrite_end* aktiviert, wenn alle Daten

des Teilprogramms ins lokale RAM geschrieben worden sind. In der Tabelle 2.10 werden noch die Kontrollsignale *stall_wb* und $PRIMITIV\_WB$ erwähnt, die sich wie folgt berechnen:

$$stall\_wb := pwrite\_end \wedge T_{13}$$

$$PRIMITIV\_WB := \begin{cases} primitiv\_wb & \text{falls } pwrite\_res \\ PRIMITIV\_WB & \text{sonst} \end{cases}$$

$$P\_WB[3:0] := \begin{cases} p\_wb[3:0] & \text{falls } pwrite\_res \\ P\_WB[3:0] & \text{sonst} \end{cases}$$

Durch Aktivieren des Signals *pwrite_res* für einen Takt signalisiert das RAM-Interface, daß es damit begonnen hat, einen Zugriff der Stufe *WB* auszuführen. Dann hat das RAM-Interface auch die Start- und Endadresse des Zugriffs verarbeitet. Deshalb aktiviert die Stufe *WB* als Reaktion auf *pwrite_res* nach Tabelle 2.10 das Signal *address.pop*, um das Write-Info aus der *address*-Fifo auszulesen und die Registern $P\_WB[3:0]$ und $PRIMITIV\_WB$ zu clocken und aus $p\_wb[3:0]$ und *primitiv_wb* zu laden. Außerdem wird der Zähler *pcnt* bei *pwrite_res* heruntergezählt.

Das RAM-Interface aktiviert das Signal *pwrite_end* für einen Takt, wenn alle Daten des gewünschten RAM-Zugriffs der Stufe *WB* geschrieben worden sind, und liefert in *sn_error* gleichzeitig ein akkumuliertes Error-Bit für die Antworten dieser Teilrunde. Gilt zu diesem Zeitpunkt $sn\_error = 1$, dann setzt *WB* wegen $error\_we = 1$ und $error\_sel = 1$ nach Tabelle 2.10 das Error-Register im Status-Register nach Abbildung 2.9. Die zugehörige Kanalnummer wird dabei dem Register $P\_WB[3:0]$ entnommen, das bei *pwrite_res* geclockt wurde und damit die Kanalnummer des Programms in der Stufe *WB* enthält. War das zugehörige Teilprogramm primitiv ($PRIMITIV\_WB = 1$), dann wird das Status-Register in Abbildung 2.9 wegen $status\_we = 1$ und $status\_sel1 = 1$ für den Kanal auf 0 zurückgesetzt. Falls die Stufe *EX* gerade das Status-Register schreibt, wird der Schreibzugriff von *WB* auf dieses Register um einen Takt gestallt. Das Stalling erfolgt dabei mit Hilfe eines Registers *stall_wb*, das genau dann einen Takt nach T_{13} aktiv wird, wenn der Zugriff von *WB* eigentlich in T_{13} erfolgen sollte. In T_{13} schreibt nämlich die Stufe *EX* nach Tabelle 2.9 gegebenenfalls das *STAT*-Register.

Insgesamt erhalten wir somit den folgenden Satz über die Korrektheit der Stufe *WB*.

Satz 2.9.11 *Wenn das RAM-Interface korrekt arbeitet, werden genau die spezifizierten Daten aus der* receive*-Fifo gelesen und ins SDRAM geschrieben, und zwar an die korrekte Start- und Endadresse, die der* address*-Fifo entnommen wird. Der Zähler pcnt wird dekrementiert, das Write-Info aus der* address*-Fifo ausgelesen und gegebenenfall werden das STAT- und ERROR-Register nach dem Algorithmus für die Stufe* WB *geupdatet.*

Beweis: Da die Write-Infos nach Lemma 2.9.7 korrekt in die *address*-Fifo geschrieben werden, enthält die *address*-Fifo in der Stufe *WB* noch die korrekten Write-Infos. Wenn das RAM-Interface korrekt arbeitet, werden die Kontrollsignale *pwrite_res* und *pwrite_end* genau einmal für jedes Teilprogramm in der Stufe *WB* aktiv, da nach Satz 2.9.10 die Stufe *PRAM* korrekt arbeitet und damit ankommende Teilprogramme in *pcnt* gezählt

werden. Damit wird für jedes Teilprogramm *genau ein* Write-Info aus der *address*-Fifo gelesen und die Register $P\_WB[3:0]$ und $PRIMITIV\_WB$ werden genau einmal geclockt und aus der *address*-Fifo geladen. Da das RAM-Interface *sn_error* korrekt berechnet, wird das *ERROR*-Register genau dann auf 1 gesetzt, wenn ein Fehler aufgetreten ist, also $sn\_error = 1$ galt. Da die Write-Infos korrekt sind, enthält insbesondere auch das Register $PRIMITIV\_WB$ den korrekten Wert, da es aus *primitv_wb* geladen wird. Damit wird das *STAT*-Register genau dann auf 0 zurückgesetzt, wenn das Teilprogramm in der Stufe *WB* primitiv ist. □

2.9.6 Gesamte Kontrolle

In den vorangehenden Abschnitten haben wir die Kontrollsignale der einzelnen Stufen spezifiziert und gezeigt, daß die einzelnen Stufen dabei nach dem in Abschnitt 1.2.3 spezifizierten Algorithmus arbeiten. Wir wollen in diesem Abschnitt für einige Kontrollsignale die vollständigen Kontrollgleichungen angeben, die sich durch die Ausführung von Befehlen und die Stufen *EX* und *WB* ergibt.

Das *ERROR*-Register kann durch die Befehle *startdma* und *freedma* sowie von der Stufe *WB* geschrieben werden. Damit ergeben sich nach den Tabellen 2.6, 2.7 und 2.10 die folgenden Kontrollgleichungen.

$$\begin{aligned} error\_we &= startdma \wedge act \wedge \overline{stat} \wedge \overline{pwrite\_end} \vee \\ &\quad freedma \wedge act \wedge \overline{stat} \wedge \overline{pwrite\_end} \vee \\ &\quad pwrite\_end \wedge sn\_error \\ error\_in &= sn\_error \wedge pwrite\_end \\ error\_sel &= pwrite\_end \end{aligned}$$

Die erste Zeile steht für den Befehl *startdma*, bei dem die Bedingung aus Tabelle 2.6 wegen $act \wedge \overline{stat}$ erfüllt ist. Bei aktivem *pwrite_end* erfolgt für *startdma* kein *error_we*, da die Instruktion nach den Bemerkungen aus Abschnitt 2.4.2 gestallt wird. Analog steht die zweite Zeile für den Befehl *freedma* und die dritte Zeile für einen Schreibzugriff der Stufe *WB*.

Das *STAT*-Register kann von den Stufen *EX* und *WB* sowie durch Befehle geschrieben werden. Damit setzt sich das Kontrollsignal *status_we* zum Schreiben des *STAT*-Registers aus insgesamt *drei* Teilen zusammen. Für das Stalling der Schreibzugriffe der Stufe *WB* auf das *STAT*-Register haben wir im vorangehenden Abschnitt 2.9.5 ein zusätzliches Register *stall_wb* eingeführt. Damit ergeben sich insgesamt mit den Tabellen 2.6, 2.7, 2.9 und 2.10 die folgenden Kontrollgleichungen:

$$\begin{aligned} status\_we = \; & getdma \wedge \overline{noHC} \wedge \overline{all} \wedge \overline{T_{13}} \wedge \overline{stwe\_wb} \vee \\ & startdma \wedge act \wedge \overline{stat} \wedge \overline{T_{13}} \wedge \overline{stwe\_wb} \vee \\ & stwe\_wb \vee \\ & T_{13} \wedge valid \wedge primitiv \wedge \overline{antwort} \end{aligned}$$

$$status\_in = startdma \wedge \overline{T_{13}} \wedge \overline{stwe\_wb}$$

mit $stwe\_wb = (pwrite\_end \wedge \overline{T_{13}} \vee stall\_wb) \wedge PRIMITIV\_WB$

$$status\_sel[1:0] = \begin{cases} 00 & \text{falls } startdma \wedge \overline{stwe\_wb} \wedge \overline{T_{13}} = 1 \\ 01 & \text{falls } getdma \wedge \overline{stwe\_wb} \wedge \overline{T_{13}} = 1 \\ 10 & \text{falls } stwe\_wb = 1 \\ 11 & \text{sonst} \end{cases}$$

Die ersten beiden Zeilen der Kontrollgleichung von *status_we* gehören zu den Befehlen *getdma* und *startdma*. Diese Befehle werden gestallt, wenn das *STAT*-Register gerade von *EX* oder *WB* geschrieben wird. Die Stufe *EX* schreibt dieses Register nach Tabelle 2.9 höchstens in T_{13}. Die Stufe *WB* schreibt dieses Register nach Tabelle 2.10, wenn $(pwrite\_end \wedge \overline{T_{13}} \vee stall\_wb) \wedge PRIMITIV\_WB = 1$ gilt, also genau, wenn das oben eingeführte Signal *stwe_wb* den Wert 1 hat. Deshalb enthalten die beiden oberen Zeilen der Gleichung von *status_we* den Term $\overline{T_{13}} \wedge \overline{stwe\_wb}$.

Die 3. Zeile steht für die Schreibzugriffe der Stufe *WB*, die nach Bemerkungen aus Abschnitt 2.9.5 bereits richtig gestallt sind. Die letzte Zeile steht dann für die Schreibzugriffe der Stufe *EX* nach Tabelle 2.9, die nicht gestallt werden.

2.10 Designverbesserung

Das vorgestellte Design erreicht einen hohen Datendurchsatz bei Standardanweisungen und eine geringe Latenz bei Einzelanweisungen. Jede Einzelanweisung schickt aber sieben leere Speicheranfragen an die SB-PRAM, da sie die vollen 32 Takte in der Stufe *EX* bleibt und damit in einer Teilrunde nur eine Speicheranfrage an die SB-PRAM geschickt wird. Gleichzeitig liest keine Einzelanweisung Daten aus dem lokalen RAM. Deshalb ist es möglich, mehrere Einzelanweisungen nacheinander in einer *EX*-Stufe von 32 Takten auszuführen. Dadurch wird die Latenz bei Einzelanweisungen weiter gesenkt und der Datendurchsatz weiter erhöht. Allerdings muß man dann an den PC die Bedingung stellen, daß keine 2 laufenden Einzelanweisungen auf dieselbe Adresse im Speicher der SB-PRAM zugreifen. Diese beiden Zugriffe könnten im verbesserten Design nämlich in einer Teilrunde an die SB-PRAM gesendet werden. Die SB-PRAM erwartet aber, daß die Sortiereinheit bereits alle Zugriffe auf ein und dieselbe Adresse kombiniert hat.

Theoretisch bietet sich die Möglichkeit, bis zu 8 Einzelanweisungen in einer Stufe *EX* auszuführen. Dabei stellt sich aber das Problem, daß das DMA-Programm in der nachfolgenden Stufe *EX* möglicherweise eine Standardanweisung ist, die ihre Daten parallel zur 2. Hälfte der *EX*-Stufe lesen will, in der gerade die 8 Einzelanweisungen ausgeführt werden. Deshalb führen wir höchstens 4 Einzelanweisungen in der Stufe *EX* aus–und zwar in der ersten Hälfte von *EX*. Damit kann eine Standardanweisung ganz normal zur 2. Hälfte von *EX* in die Stufe *READ* gelangen, ohne daß eine grundsätzliche Anpassung der Hardware nötig ist.

Bei Standardanweisungen haben wir eine versetzte Arbeitsweise nach Abbildung 2.14 implementiert. Da Einzelanweisungen keine Daten aus dem lokalen RAM lesen, ist hier keine solche versetzte Arbeitsweise nötig. Deshalb lesen wir bei Einzelanweisungen immer das ganze DMA-Programm aus dem Registerfile aus. Da Einzelanweisungen primitiv sind, schreiben sie nie Teile des DMA-Programms zurück; damit entfällt auch das versetzte Zurückschreiben der Standardanweisungen.

Takt	Signal
T_1, T_5, T_9	*address.push*
T_2, T_6, T_{10}	*la*
T_3, T_7, T_{11}	*dmace*, *padrce*, *dmapop*, *status_we*, *status_sel*0, *status_sel*1
T_{13}	*address.push*
T_{15}	*status_we*, *status_sel*0, *status_sel*1

Tabelle 2.11: Aktive Kontrollsignale bei Einzelanweisungen

Die *erste* Einzelanweisung in der Stufe *EX* wird allerdings nach wie vor in zwei Teilen gelesen: in T_{15} werden $SADR$ und $LAOP$ gelesen und in T_{31} das Register $PADR$. Anschließend erfolgt in T_2 ein *la*, wodurch die nächste Einzelanweisung die Stufe *READ* betritt, und in T_3 wird dann das DMA-Programm vollständig gelesen und gelangt in die Stufe *EX*. Dabei dürfen bei *la* natürlich nur Einzelanweisungen betrachtet werden. Nach 4 solchen Einzelanweisungen erfolgt in T_{14} ein normales *la*, wodurch dann wieder eine Standardanweisung in die Stufe *READ* gelangen kann. Die Aktivierung der Kontrollsignale bei Einzelanweisungen ist in Tabelle 2.11 zusammengefaßt. Für die Signale *address.push* und *status_we* gelten dabei dieselben Bedingungen wie in Tabelle 2.9. Dabei ergibt sich die vollständige Kontrollgleichung für ein Signal *sig* aus dem bisherigen Signal *sig_std* und dem neuen Anteil für Einzelanweisungen *sig_ea* zu $sig = (sig\_std \wedge /ea) \vee (sig\_ea \wedge ea)$.

Wir wir oben bemerkt haben, dürfen nur Einzelanweisungen in die Stufe *EX* gelangen, wenn das erste DMA-Programm in der Stufe *EX* eine Einzelanweisung ist. Damit können wir bei einem *la* nicht mehr nur dann die Fifo mit den Einzelanweisungen auswählen, wenn diese Fifo nicht leer ist. Bei einem *la* nicht in T_{14} wird deshalb *immer* die Fifo mit den Einzelanweisungen ausgewählt. Außerdem müssen die Kontrollsignale *validO*, *lor* und *VPlast* für Einzelanweisungen anders berechnet werden. Dabei ergeben sich die folgenden Kontrollgleichungen. Der Zusatz $/cnt4$ bei Einzelanweisungen ist hier nötig, da nur bis zu 4 Einzelanweisungen in der Stufe *EX* ausgeführt werden. Die 4. Einzelanweisung bleibt aber bis zu T_{31} in der Stufe *EX*; damit kann in der 2. Hälfte einer Stufe *EX* mit Einzelanweisungen das *valid*-Bit der Stufe *EX* aktiv sein. Dieses aktive *valid* wird durch den Zusatz $\wedge/cnt4$ in der 2. Hälfte von *EX* ignoriert. Das Signal *VPlast* wird bei Einzelanweisungen aktiviert, wenn $cnt[4:2] = 011$, wenn also gerade die vierte und letzte Einzelanweisung einer *EX*-Stufe ihre Speicheranfrage sendet.

$$
\begin{aligned}
easel\_rd &= /eaempty \wedge T_{14} \vee /T_{14} \\
validO &= (valid \wedge /ende \wedge /ea) \vee (valid \wedge /cnt4 \wedge ea) \\
lor &= (\delta_{NUM[2:0],cnt[4:2]} \wedge /ea) \vee (valid \wedge /cnt4 \wedge ea) \\
VPlast &= (\delta_{NUM[2:0],cnt[4:2]} \wedge /ea) \vee (ea \wedge /cnt4 \wedge cnt3 \wedge cnt2)
\end{aligned}
$$

Neben diesen Änderungen der Kontrolle ist an einer Stelle auch eine Änderung an den Datenpfaden nötig. Bisher wurden Modus und Operator einer Speicheranfrage aus dem Register OP gelesen, das aus $LAOP$ geclockt wurde und damit die ganze Stufe *EX* über gültig war. Bei Einzelanweisungen verwenden wir an dieser Stelle das Register $LAOP$, da das Register OP nicht rechtzeitig für eine Einzelanweisung geclockt werden kann. Eine

Einzelanweisung wird nämlich vollständig gelesen, und erst im *nächsten* Takt kann dann OP geclockt werden. Damit liegt an der Sortiereinheit und an der *address*-Fifo jetzt nicht mehr das OP-Register an, sondern OP' mit

$$OP'[3:0] = \begin{cases} OP[3:0] & \text{falls } /ea \\ LAOP[3:0] & \text{falls } ea \end{cases}$$

Auch die Kontrolleinheit verwendet zur Berechnung von *address.push* und *status_we* dieses Signal OP' anstelle von OP. Da *send.pop* bei Einzelanweisungen nie aktiv wird, kann es nach wie vor mit OP berechnet werden.

Damit haben wir den Datendurchsatz des DMA-Prozessors noch weiter gesteigert und gleichzeitig die Latenz gesenkt. Der PC darf jetzt allerdings keine zwei Einzelanweisungen mehr gleichzeitig starten, die auf dieselbe Adresse im Speicher der SB-PRAM zugreifen. Erst nachdem eine Einzelanweisung mit PRAM-Adresse *adr* beendet ist, darf der PC eine zweite Einzelanweisung mit dieser PRAM-Adresse *adr* starten. Die Korrektheit dieser Designverbesserung folgt unmittelbar aus der Korrektheit des alten Designs, denn *la* und *dma_pop* werden im neuen Design nur in zusammenhängender Folge aktiv und die Korrektheit für *eine* Einzelanweisung in der Stufe *EX* haben wir bereits gezeigt.

Kapitel 3

Sortiereinheit

Die Sortiereinheit entspricht im wesentlichen dem Sortierknoten, der in [Gö96] von Thomas Göler entworfen und verifiziert wurde. Dieser Sortierknoten faßt Speicheranfragen mit gleicher Adresse innerhalb einer Teilrunde zu *einer* Speicheranfrage zusammen. Da der DMA-Prozessor innerhalb einer Teilrunde aber keine 2 Speicheranfragen mit gleicher PRAM-Adresse an die Sortiereinheit versendet, ist die von Göler implementierte „Paketkombinierung“ nicht nötig. Dadurch vereinfacht sich das Originaldesign deutlich. Darüber hinaus wurde das Design für die verwendete FPGA-Architektur optimiert.

Der größter Unterschied zum originalen Sortierknoten besteht in der Tatsache, daß die Sortiereinheit des DMA-Prozessors *asynchron* arbeitet. Während das Prozessor-Interface mit der lokalen Clock CLK arbeitet, läuft das PRAM-Interface mit der SB-PRAM-Clock $PCLK$. Die Synchronisierung der Daten erfolgt mittels asynchroner Fifo-Buffer. Bei der Beschreibung des Designs liegt der Schwerpunkt auf den Stellen, die sich gegenüber dem Sortierknoten der SB-PRAM verändert haben.

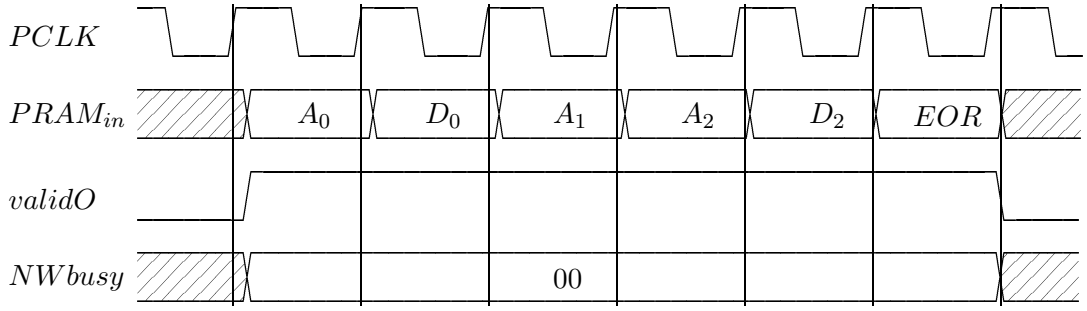
In diesem Kapitel spezifizieren wir zunächst das Interface zur SB-PRAM, dann geben wir an, wie die Sortiereinheit aus den 3 Komponenten Hinsortierung, Rücksortierung und Sort-Queue aufgebaut ist. Anschließend gehen wir auf diese 3 Komponenten getrennt ein.

3.1 Interfaces

Das Interface zwischen Sortiereinheit und DMA-Prozessor wurde bereits im vorigen Kapitel in Abschnitt 2.1.2 erörtert. Damit bleibt nur noch das Interface zwischen Sortiereinheit und SB-PRAM. Alle ausgehenden Signale dieses Interfaces werden im FPGA registriert, bevor sie auf die PCI-Karte getrieben werden. Genauso werden alle eingehenden Signale dieses Interfaces im FPGA registriert, bevor sie verwendet werden. Nach Abbildung 1.8 werden auf der Mezzaninkarte nochmals alle ausgehenden Signale registriert, bevor sie von der Karte zur SB-PRAM getrieben werden, und auch alle eingehenden Signale, bevor sie zum FPGA auf der PCI-Karte getrieben werden. Damit ist nach der Spezifikation des FPGAs in [FPGA99] gewährleistet, daß zwischen FPGA und den registrierten Treibern der Aufsatzkarte die Timing-Vorgaben für eine Taktfrequenz von 32 MHz eingehalten werden. Außerdem wird durch die Verwendung der registrierten Treiber auf der Aufsatz-

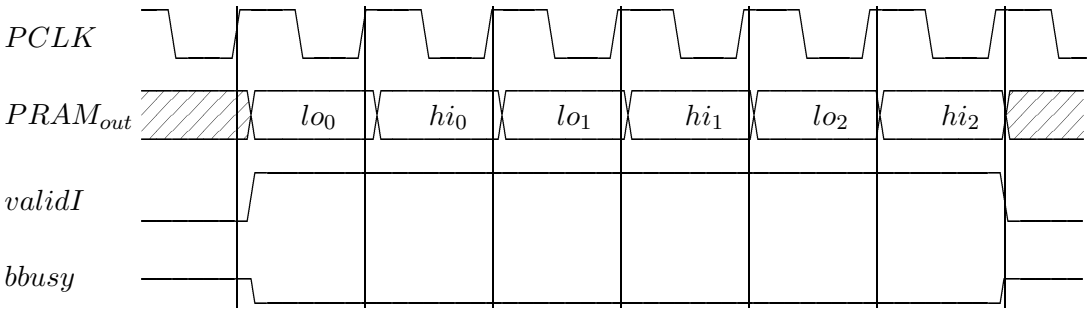
karte das Timing des SB-PRAM-Netzwerks eingehalten, denn die Prozessorplatine der SB-PRAM nach [Ba96] arbeitet mit der gleichen Schaltung aus registrierten Treibern am Netzwerkinterface.

Abbildung 3.1 zeigt das Busprotokoll des verbleibenden Interfaces zwischen Sortiereinheit und SB-PRAM, das mit der PRAM-Clock $PCLK$ betrieben wird. Dieses Protokoll wird auch in [Gö96] eingeführt. Ähnlich wie das Interface zwischen Prozessor und Sortiereinheit in Abbildung 2.2 auf Seite 39 gliedert es sich in zwei Teile, den *Hinweg*, also das Senden von Speicheranfragen an die SB-PRAM, und den *Rückweg*, den Empfang der Antwortden der SB-PRAM.



Senden von Speicheranfragen an die SB-PRAM (=Hinweg)

A_i steht für den Modus und die Adresse der i-ten Speicheranfrage einer Teilrunde und D_i für den zugehörigen Operator und das zugehörige Datum, wenn die Speicheranfrage ein Datum hat. *EOR* steht für ein *EOR*-Paket zur Trennung von Teilrunden. Die SB-PRAM treibt nur die Signale $NWbusy[1:0]$.



Empfang von Antworten der SB-PRAM (=Rückweg)

lo_i steht für das untere Halbpaket des i-ten Antwortdatums und hi_i steht analog für das obere Halbpaket. Die Sortiereinheit treibt nur das Signal *bbusy*.

Abbildung 3.1: Timing zwischen Sortiereinheit und SB-PRAM

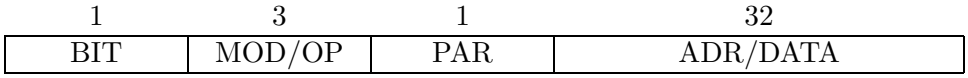


Abbildung 3.2: Aufbau einer Speicheranfrage zwischen Sortiereinheit und SB-PRAM

$PRAM_{in}[35:33]$	Modus	Operator
000	LOAD	and
001	STORE	ERR
010	MP	or
011	SYNC	%
100	GHOST	max
101	EOR	%
110	ERR	add
111	ERR	ERR

Tabelle 3.1: Zugriffsmodi und -operatoren zwischen SB-PRAM und Sortiereinheit

Die Speicheranfragen an die SB-PRAM in $PRAM_{in}$ werden bei aktivem *validO* geschickt. Dabei wird zunächst die PRAM-Adresse und der Modus des Zugriffs geschickt. Enthält die Speicheranfrage auch ein Datum, handelt es sich also um einen *MP*, *SYNC* oder *STORE*-Zugriff, dann wird im anschließenden Takt auch das Datum mit dem Operator für den Zugriff geschickt. Die Speicheranfragen der einzelnen Teilrunden werden durch ein *EOR*-Paket getrennt (vergleiche Abschnitt 1.1 auf Seite 2). Durch ein aktives *NWbusy* signalisieren die beiden nachfolgenden Netzwerkknoten der SB-PRAM, daß sie keine weiteren Speicheranfragen mehr aufnehmen können.

Die Antworten der SB-PRAM kommen in Form von Halbpaketen in $PRAM_{out}$ bei aktivem *validI* bei der Sortiereinheit an. Dabei trifft das Halbpaket mit der unteren Hälfte des Antwortdatums *vor* der oberen Hälfte ein, es sei denn, bei der zugehörigen Speicheranfrage handelt es sich um einen Multipräfix-Zugriff mit dem Operator *max* (siehe Abschnitt 1.1). Durch ein aktives *bbusy* signalisiert die Sortiereinheit, daß sie keine Antwortdaten mehr aufnehmen kann.

Die Speicheranfragen auf dem Hinweg sind dabei nach Abbildung 3.2 aufgebaut. In *DATA/ADR* steht die Adresse oder das Datum der Speicheranfrage. In *MOD/OP* ist beim Adreßteil der Speicheranfrage der Modus des Zugriffs, beim Datenteil der Operator gemäß Tabelle 3.1 kodiert. Gegenüber der Kodierung von Modus und Operator zwischen DMA-Prozessor und Sortiereinheit nach Abbildung 2.3 auf Seite 38 ist die Kodierung zwischen Sortiereinheit und SB-PRAM um ein Bit erweitert. Dadurch können auf diesem Interface mehr Modi kodiert werden, die beispielsweise zum Senden eines *EOR*-Pakets zum Trennen der einzelnen Teilrunden benötigt werden. Für Fehler, die im Sortierknoten auftreten können, wenn mehrere Zugriffe auf dieselbe Adresse innerhalb einer Teilrunde erfolgen, gibt es weitere Modi, die in Tabelle 3.1 mit *ERR* bezeichnet sind. Da in der Sortiereinheit des DMA-Prozessors keine zwei gleichen Adressen in einer Teilrunde auftreten, sind diese Modi und Operatoren nicht implementiert.

Das Paritätsbit *PAR* einer Speicheranfrage $R[36:0]$ wird nur über die Teile *ADR/DATA*

Signal	Art	Bedeutung
$PRAM_{in}[36:0]$	O	Adreß- oder Datenteil einer Speicheranfrage der SB-PRAM. $PRAM_{in}[36:0]$ ist dabei gemäß Abbildung 3.2 aufgebaut.
$validO$	O	aktiv, wenn an $PRAM_{in}$ gültige Daten liegen
$NWbusy[1:0]$	I	*network busy* der beiden Netzwerkknoten, die im Butterfly-Netzwerk der SB-PRAM mit der Sortiereinheit verbunden sind; aktiv, wenn der entsprechende Netzwerkknoten keine Speicheranfragen mehr aufnahmen kann
$PRAM_{out}[17:0]$	I	$PRAM_{out}[15:0]$ enthält eine Hälfte des Datums, das die PRAM als Antwort auf eine Speicheranfrage gesandt hat, $PRAM_{out}16 = \bigoplus_{i=0}^{15} PRAM_{out}i$ das Paritätsbit und $PRAM_{out}17$ das Error-Bit
$validI$	I	aktiv, wenn an $PRAM_{out}[17:0]$ eine gültige Antwort der SB-PRAM liegt
$bbusy$	O	aktiv, wenn die Sortiereinheit keine Daten von der SB-PRAM mehr aufnehmen kann

Tabelle 3.2: Interface zwischen Sortiereinheit und SB-PRAM

Die Buchstaben I und O in der Spalte Art stehen für Input- beziehungsweise Output-Signale aus Sicht der Sortiereinheit.

und MOD/OP berechnet, also $PAR = \bigotimes_{i=0}^{31} R[i] \otimes \bigotimes_{i=33}^{35} R[i]$. Das Bit BIT enthält das duplizierte Routing-Bit, also $BIT = R[\langle RBIT[3:0]\rangle + 16]$, wobei das $RBIT$-Register die Routing-Bit-Nummer der ersten Netzwerkstufe der SB-PRAM enthält. In Tabelle 3.2 sind nochmals alle Signale des Interfaces zwischen Sortiereinheit und SB-PRAM zusammengefaßt.

3.2 Funktionsweise

Die Sortiereinheit nimmt die Speicheranfragen des DMA-Prozessors entgegen und sortiert die bis zu 8 Anfragen einer Teilrunde nach aufsteigenden PRAM-Adressen. Diese sortierten Adressen werden dann an das Netzwerk der PRAM verschickt, und zwar gegebenenfalls mit den zugehörigen Daten. Die Antworten treffen vom Netzwerk in der Reihenfolge ein, in der die Anfragen an das Netzwerk versandt wurden. Die Sortiereinheit setzt die eintreffenden Halbpakete der Antworten wieder zu einem Datum zusammen. Anschließend werden die Antworten des Netzwerks wieder rücksortiert und an den DMA-Prozessor zurückgegeben. Die Antworten werden somit von der Sortiereinheit in der Reihenfolge an den DMA-Prozessor zurückgegeben, in der der DMA-Prozessor zuvor seine Anfragen abgeschickt hat.

Die Sortiereinheit gliedert sich damit in drei Teile, die Hinsortierung, die Rücksortierung und die Sort-Queue, die ausschließlich mit der PRAM-Clock betrieben wird. In der

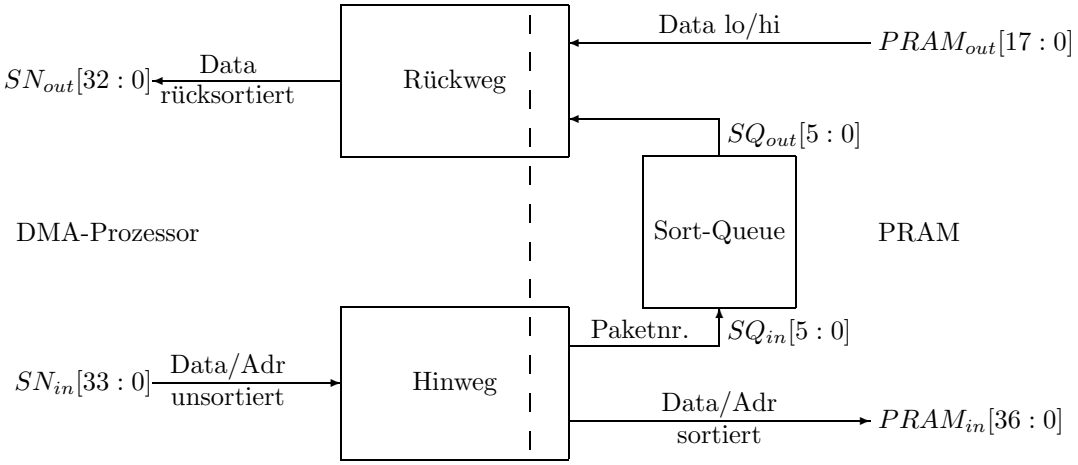


Abbildung 3.3: Arbeitsweise der Sortiereinheit

Die gestrichelte senkrechte Linie trennt die lokale Clock CLK des DMA-Prozessors von der PRAM-Clock $PCLK$.

Sort-Queue wird die Reihenfolge der Hinsortierung vermerkt, um die Sortierung auf dem Rückweg wieder rückgängig machen zu können. Hin- und Rücksortierung arbeiten jeweils mit beiden Clocks. Abbildung 3.3 veranschaulicht diesen Zusammenhang.

3.3 Hinweg

Der Hauptbestandteil der Hinsortierung ist ein sogenanntes *lineares Sortierfeld*, das die PRAM-Adressen einer Teilrunde sortiert. Der Aufbau eines linearen Sortierfelds und die Korrektheit der Arbeitsweise dieses Sortierfelds kann [Gö96] entnommen werden. Nachdem 8 Adressen in das lineare Sortierfeld geschrieben wurden, kann im nächsten Takt die kleinste Adresse ausgelesen werden und gleichzeitig eine weitere Adresse einer neuen Teilrunde eingespeist werden. Diese neuen Adressen werden unabhängig von den alten Adressen im linearen Sortierfeld sortiert. Sind alle alten Adressen ausgespeist, kann im nächsten Takt dann die kleinste der neuen Adressen ausgelesen werden. Gleichzeitig kann erneut eine Adresse einer weiteren Teilrunde in den Sortierer eingespeist werden. Somit ist ein einzelnes lineares Sortierfeld in der Lage, kontinuierlich eintreffende Adressen in Runden der Länge 8 zu sortieren.

Abbildung 3.4 gibt den Aufbau der Hinsortierung wieder. Die vom DMA-Prozessor an SN_{in} eintreffenden Adreßteile von Speicheranfragen werden mit ihrer Nummer innerhalb der aktuellen Teilrunde versehen, also einer Zahl zwischen 0 und 7. Diese Nummer wird als *Paketnummer* bezeichnet. Die Paketnummer wird durch einen Zähler generiert, nämlich *Count*. Die Datenteile der Speicheranfragen sind für die Sortierung der Adressen nicht relevant; deshalb werden sie in einem separaten RAM, dem *Hinweg-RAM* gespeichert. Die Paketnummer der Speicheranfrage gibt dabei die Schreibadresse WA für den

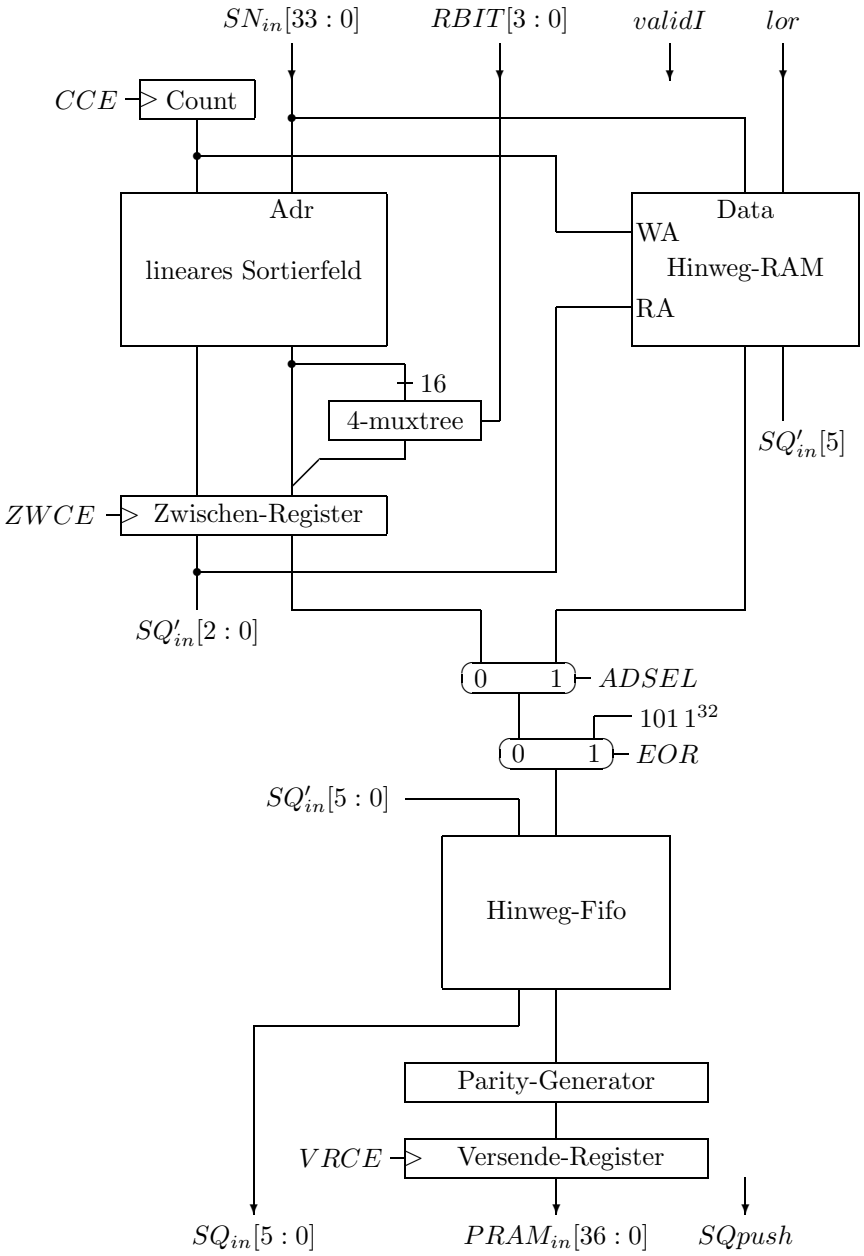


Abbildung 3.4: Blockschaltbild des Hinwegs

zugehörigen Datenteil und das *lor*-Flag im *Hinweg-RAM* an. Außerdem wird die Paketnummer mit der zugehörigen PRAM-Adresse im linearen Sortierfeld nach der Größe der Adresse mitsortiert. Anhand der so sortierten Paketnummern ist es später möglich, die zugehörigen Datenteile zu identifizieren und die von der PRAM eintreffenden Antworten wieder rückzusortieren.

Eine Adresse wird mit ihrer Paketnummer aus dem linearen Sortierfeld ausgespeist und in das *Zwischen-Register* geschrieben. Gleichzeitig wird mit einem *4-Muxtree* das Routing-Bit für diese Adresse aus den 16 oberen Adreßbits ausgewählt und zusätzlich ins *Zwischen-Register* geclockt. Von dort wird der Adreßteil einer Speicheranfrage zur Synchronisierung mit der PRAM-Clock in eine *Hinweg-Fifo* geschrieben. Sendet die Speicheranfrage ein Datum an die SB-PRAM, dann wird im nachfolgenden Takt auch der korrespondierende Datenteil der Speicheranfrage aus dem *Hinweg-RAM* gelesen und in die *Hinweg-Fifo* geschrieben. Dabei gibt die Paketnummer der zugehörigen PRAM-Adresse im *Zwischen-Register* die Leseadresse *RA* im *Hinweg-RAM* an. Nach dem Ausspeisen einer vollständigen Teilrunde aus dem linearen Sortierfeld wird ein *EOR*-Paket in die *Hinweg-Fifo* geschrieben. Dieses *EOR*-Paket wird auch dann geschrieben, wenn der DMA-Prozessor in dieser Teilrunde keine Speicheranfragen versandt hat.

Das *Hinweg-RAM* besteht aus 2 RAM-Bänken, damit es parallel zum linearen Sortierfeld kontinuierlich Datenteile aufnehmen kann, während die Daten der vorhergehenden Runde gleichzeitig ausgelesen werden. Da der kleinste RAM-Baustein in der verwendeten FPGA-Technologie ein 16×1-RAM ist, sind die beiden RAM-Bänke in *einem* dual-port RAM realisiert. Mit dem obersten Adreßbit wird zwischen den beiden Bänken ausgewählt. Während also die Datenteile einer Teilrunde an den Adressen 0 bis 7 des *Hinweg-RAMs* gespeichert werden, können die Datenteile der vorhergehenden Teilrunde von den Adressen 8 bis 15 gelesen werden und umgekehrt.

Von der *Hinweg-Fifo* werden die einzelnen Teile der Speicheranfragen sukzessive mit der PRAM-Clock $PCLK$ ausgelesen und über das *Versende-Register* an das PRAM-Netzwerk versandt. Dieses *Versende-Register* wird ebenfalls mit der PRAM-Clock $PCLK$ betrieben. Zusätzlich wird noch jeweils ein Parity-Bit berechnet und mit ans Netzwerk gesendet.

Handelt es sich bei dem gerade zu versendenden Paket um einen Adreßteil mit Routing-Bit $BIT \in \{0, 1\}$ und gilt $NWbusy[BIT] = 1$, dann wird dieser Adreßteil nicht verschickt. Statt dessen wird nach Abschnitt 1.1 ein *GHOST*-Paket mit dem gleichen Adreßteil verschickt, bis die SB-PRAM das entsprechende $NWbusy$-Signal zurücknimmt.. *EOR*-Pakete gelten ebenfalls als Adreßteile von Speicheranfragen. Also sendet die Sortiereinheit gegebenenfalls auch *GHOST*-Pakete eines *EOR*-Pakets.

Bei Speicheranfragen, die eine Antwort von der SB-PRAM erwarten, wird die Paketnummer für die Rücksortierung in die Sort-Queue geschrieben. Außerdem wird in der Sort-Queue vermerkt, welche Speicheranfrage die letzte einer Teilrunde war. Damit ist eine Trennung der Runden für die Rücksortierung möglich. Zusätzlich wird in der Sort-Queue vermerkt, ob es sich bei der Speicheranfrage um eine Multipräfix-Anfrage mit dem Operator *max* handelt. In diesem Fall kommen die beiden Halbpakete der Antwort nämlich in umgekehrter Reihenfolge an, also zuerst die obere Hälfte und dann die untere Hälfte. Zuletzt wird noch das *lor*-Flag des DMA-Prozessors für diese Speicheranfrage in die Sort-

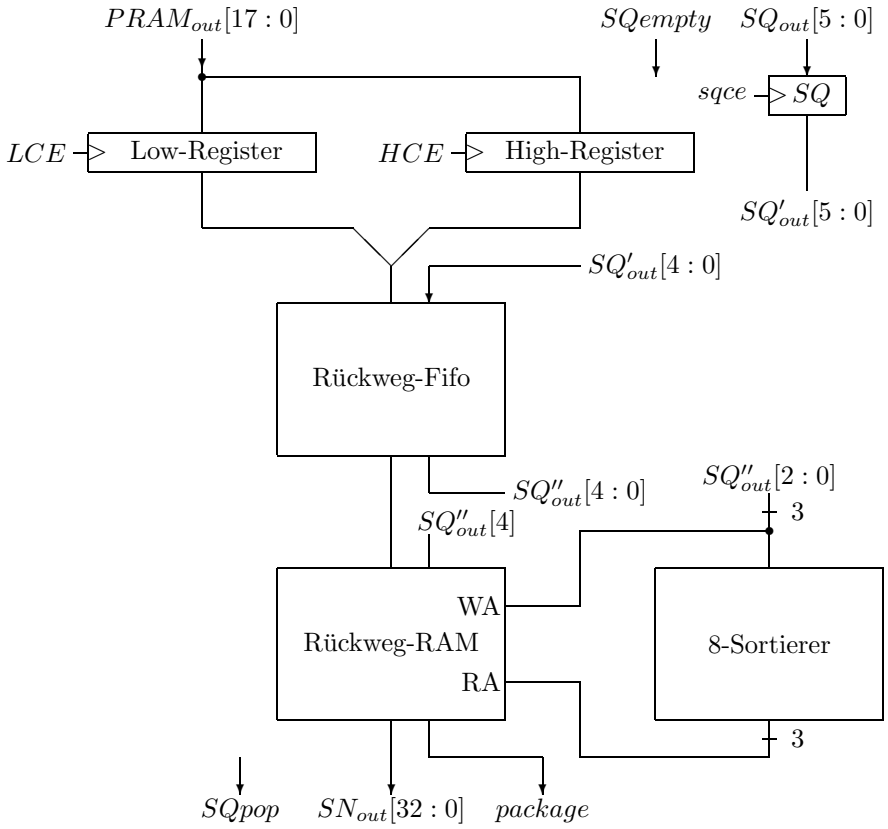


Abbildung 3.5: Blockschaltbild des Rückwegs

Queue geschrieben. Dieses Flag wird mit der zugehörigen Antwort der SB-PRAM an den DMA-Prozessor zurückgegeben.

3.4 Rückweg

Die Antworten auf Speicheranfragen der Sortiereinheit bestehen aus 2 Halbpaketen mit je einem 16-Bit-Datenteil und je einem Error- und einem Parity-Bit. Die PRAM sendet eine Antwort nicht vollständig in einem Takt; die beiden zugehörigen Halbpakete werden nacheinander über denselben Bus versandt.

Der Rückweg beginnt gemäß Abbildung 3.5 mit zwei Registern, dem *Low-Register* und dem *High-Register*. Diese beiden Register werden mit der PRAM-Clock $PCLK$ betrieben und nehmen die Antworten des Netzwerks auf, die in Form von Halbpaketen ankommen. Diese Halbpakete werden dann zu einem einzigen Datum zusammengesetzt und mit der PRAM-Clock $PCLK$ in eine *Rückweg-Fifo* geschrieben. In *einem* Error-Bit wird dabei vermerkt, ob bei einem der beiden Halbpakete ein Netzwerk-Fehler oder ein Parity-Fehler aufgetreten ist. Bei Multipräfix-Zugriffen mit dem Operator *max* kommen die beiden Halbpakete in umgekehrter Reihenfolge an. Damit die beiden Hälften richtig zusammengesetzt

werden können, liest der Sortierer den zugehörigen Sort-Queue-Eintrag ins SQ-Register, bevor das erste Halbpaket einer Antwort eintrifft. Mit Hilfe des SQ-Registers kann der Sortierer dann entscheiden, in welches Register ein ankommendes Halbpaket geschrieben wird. Der Inhalt des SQ-Registers wird für die Rücksortierung mit dem zugehörigen Antwortdatum in die *Rückweg-Fifo* geschrieben.

Die Sortiereinheit kann durch das Setzen eines *bbusy*-Signals anzeigen, wenn sie zur Zeit keine Halbpakete mehr von der SB-PRAM aufnehmen kann. In diesem Fall können wegen der Registerstufen zwischen Sortiereinheit und Netzwerkchip noch bis zu 8 Halbpaketen an der Sortiereinheit eintreffen. Diese Halbpakete dürfen nicht verloren gehen, und werden von der Sortiereinheit in jedem Fall noch gespeichert. Das *bbusy*-Signal des Sortierers an die PRAM wird deshalb bereits gesetzt, wenn die *Rückweg-Fifo* noch mindestens 4 Antworten aufnehmen kann, also 8 Halbpakete. Erst wenn der Sortierer das *bbusy*-Signal wieder zurücksetzt, werden wieder Halbpakete von der SB-PRAM gesandt.

Die *Rückweg-Fifo* dient also nicht nur der Synchronisierung der Daten, sondern auch als Puffer zur Aufnahme von Halbpaketen. Die *Rückweg-Fifo* wird mit der lokalen Clock CLK ausgelesen. Ein Datum mit seinem Error-Bit wird in das *Rückweg-RAM* geschrieben, und zwar an die Adresse, die der zugehörigen Paketnummer entspricht. Parallel dazu wird die Paketnummer an den Rücksortierer weitergegeben. Die Antworten und Paketnummern einer vollständigen Runde werden sukzessive aus der *Rückweg-Fifo* gelesen und in das *Rückweg-RAM* und den Sortierer eingespeist. Eine Runde steht komplett im Sortierer, wenn der Sort-Queue Eintrag des letzten eingespeisten Datums der letzte Eintrag einer Runde auf dem Hinweg war. Dann kann die Rücksortierung beginnen.

Die Rücksortierung erfolgt genau wie im Design von Göler mit einer Variante von Bucketsort, da ja nur nach den paarweise verschiedenen Paketnummern von 0 bis 7 aus der Sort-Queue sortiert werden muß und nicht nach 32-Bit-Adressen. Auch dieser Sortierer erlaubt eine gepipelinete Arbeitsweise: während die Paketnummern einer Teilrunde eingespeist werden, können die Paketnummern der vorhergehenden Runde sortiert ausgelesen werden. Dazu besteht auch das *Rückweg-RAM* aus 2 Bänken, die wie beim *Hinweg-RAM* in einem dual-port-RAM realisiert sind.

Wenn die Paketnummern aus dem Sortierer ausgespeist werden, dienen sie als Lese-Adresse im *Rückweg-RAM*. Das Datum an dieser Adresse wird an den DMA-Prozessor versandt. Zusätzlich wird noch das *lor*-Flag aus dem zugehörigen Sort-Queue-Eintrag als *package*-Flag an dem DMA-Prozessor versandt.

Bevor wir noch kurz auf die Rücksortierung eingehen, zeigen wir hier noch, wie das *bbusy*-Signal des Sortierers aktiviert wird, damit der Sortierer noch 8 Halbpakete aufnehmen kann.

Durch ein geeignetes Kontrollsignal *last* der *Rückweg-Fifo* wird gewährleistet, daß die Fifo minimal noch die geforderten 4 Einträge aufnehmen kann. Dabei wird *last* ähnlich wie das *direction*-Signal einer asynchronen Fifo aus Kapitel 2.7.2 auf Seite 68 mit Hilfe eines Latchs berechnet. Das Signal *direction* wird aktiv, wenn der Schreibzähler den Quadranten unmittelbar vor dem Lesezähler erreicht (vergleiche Tabelle 2.8 auf Seite 69). Dann kann die Fifo noch 4 Daten aufnehmen, was den geforderten 8 Halbpaketen entspricht. Allerdings kann gleichzeitig mit dem Schreiben in die Fifo noch ein Halbpaket ankommen, und das *direction*-Signal aus dem Latch wird noch durch ein zusätzliches Register synchro-

nisiert. Damit ergeben sich bei Verwendung von *direction* nur eine Aufnahmekapazität von 6 Halbpaketen–das ist nicht genug.

Deshalb berechnen wir ein neues Kontrollsignal *last* in einem Latch. Dabei wird *last* schon aktiv, wenn der Schreibzähler *zwei* Quadranten von dem Lesezähler ist. Das Signal *last* wird genau wie *direction* inaktiv, wenn der Lesezähler den Quadranten unmittelbar vor dem Schreibzähler erreicht. Als *bbusy*-Signal wird dann das registrierte *last* verwendet.

Für das *last*-Signal ergibt sich also in Gleichungsschreibweise:

$$last = ((R_2 \otimes W_2) \wedge (R_3 \otimes W_3) \vee last) \wedge \overline{(R_2 \otimes W_3) \wedge (\overline{R_3 \otimes W_2})}$$
$$bbusy := last$$

Lemma 3.4.1 *Solange das Kontrollsignal bbusy nicht aktiv ist, kann die Sortiereinheit noch mehr als 8 Halbpakete aufnehmen. bbusy ist nicht aktiv, wenn die* Rückweg*-Fifo höchstens 3 Daten enthält.*

Beweis: Nach einem Reset ist $bbusy = 0$ und $last = 0$. Damit *bbusy* aktiv werden kann, muß zunächst *last* aktiv werden. *last* wird genau dann aktiv, wenn sich der Schreibzähler zwei Quadranten vor dem Lesezähler befindet. Zu diesem Zeitpunkt enthält die Fifo mindestens 5 Daten und höchstens 8 Daten, kann also mindestens noch 16 Halbpakete aufnehmen. In dem Takt, in dem das Datum in die Fifo geschrieben wurde, kann ein weiteres Halbpaket ankommen, und wenn *bbusy* aus *last* geclockt wird, noch eines. Abzüglich dieser beiden Halbpakete bietet die *Rückweg*-Fifo noch Platz für 14 Halbpakete, wenn *bbusy* aktiv wird.

last wird inaktiv, wenn der Lesezähler den Quadranten vor dem Schreibzähler betritt. Damit ist *last* also auf jeden Fall inaktiv, wenn die *Rückweg*-Fifo höchstens 4 Daten enthält. Im folgenden Takt gilt dann auf jeden Fall $bbusy = 0$. □

3.4.1 Rücksortierer

Der Rücksortierer arbeitet in 2 Phasen. Zunächst werden die unsortierten Daten eingespeist, dann können sie sortiert wieder ausgelesen werden. Abbildung 3.6 zeigt den Aufbau eines 8-Sortierers, der gleichzeitiges Ein- und Ausspeisen erlaubt. Die Register A und B sind dabei als Toggle-Register realisiert.

Sei zunächst $InSel = 0$. Beim Einspeisen einer Paketnummer i wird in dem 8-Bit Toggle-Register A das Bit $A[i]$ gesetzt. Nachdem alle Paketnummern einer Teilrunde in den Sortierer eingespeist wurden, wird $InSel$ getoggelt. Der Sort-Queue-Eintrag, der die Paketnummer enthält, gibt dabei Aufschluß darüber, welche Paketnummer die letzte einer Teilrunde ist. Ist nun $InSel = 1$, können weitere Paketnummern in das Register B eingespeist werden. Gleichzeitig wird der Inhalt von Register A für das Ausspeisen selektiert. Das erste auszuspeisende Paket ist das mit der kleinsten Paketnummer, also das kleinste m mit $A[m] = 1$. Die Berechnung dieses m erfolgt mittels eines Leading-Zero-Counters, der zusammenmit dem Flip-Schaltkreis aus Abbildung 3.6 ein *Trailing-Zero-Counter* ist. Dieses m wird aus dem Sortierer ausgespeist und gleichzeitig wird $A[m]$ auf 0 zurückgesetzt. Damit wird als nächstes das kleinste noch verbliebene Datum ausgespeist.

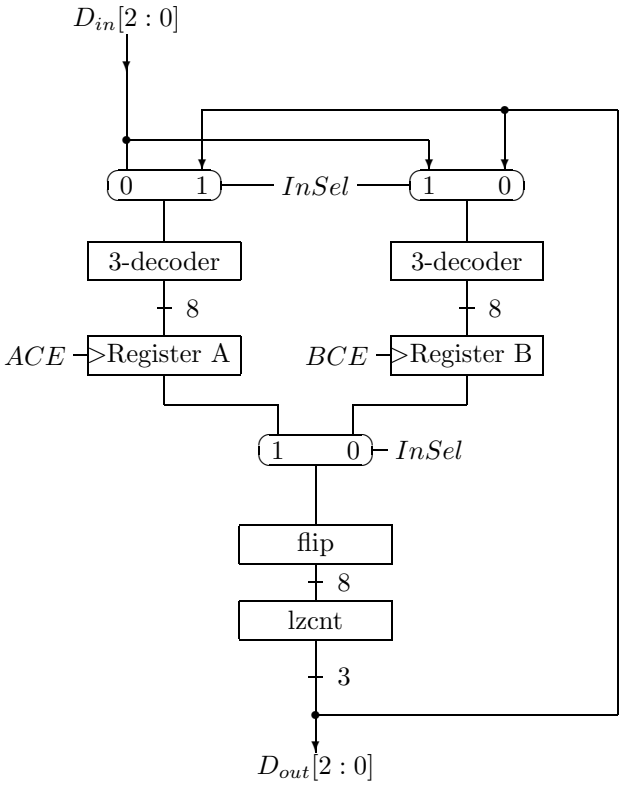


Abbildung 3.6: Blockschaltbild eines 8-Sortierers

Sind alle Bits im Register A wieder auf 0 zurückgesetz, dann sind alle Daten der Teilrunde ausgespeist und es können neue Daten in das Register A eingespeist werden.

3.5 Sort-Queue

Die Sort-Queue ist eine synchrone 6×64-Fifo-Queue, da sowohl das Schreiben als auch das Lesen mit der PRAM-Clock $PCLK$ erfolgt. Damit kann sie wie der Sortierknoten der SB-PRAM nach [Gö96] 64 Einträge aufnehmen. Für die Sort-Queue haben wir das Design der synchronen Fifo-Queue aus Abschnitt 2.7.2 von einer $m \times 16$-Queue auf eine $m \times 64$-Queue erweitert.

Die Zähler *rdcount* und *wrcount* sind jetzt jeweils 6-Bit-Zähler, deren obere 4 Bits genau wie die Zähler der $m \times 16$-Fifo-Queue zählen. Die unteren beiden Bits zählen wie in einem normalen 2-Bit-Zähler. Damit kann die *full*- und *empty*-Logik im wesentlichen von der $m \times 16$-Queue übernommen werden. Lediglich der 4-Bit Equal-Tester wird auf einen 6-Bit Equal-Tester erweitert.

Kapitel 4

Software

Die Treibersoftware für den DMA-Prozessor ermöglicht es, alle Instruktionen des DMA-Prozessors auszuführen und das lokale RAM der PCI-Karte zu lesen oder zu schreiben. Beim Lesen oder Schreiben auf das lokale RAM werden Burst-Zugriffe unterstützt.

Die Treibersoftware für den DMA-Prozessor setzt dabei auf der allgemeinen PCI-Treibersoftware von ATMedia GmbH auf, die speziell für die verwendete PCI-Karte entwickelt wurde. Diese PCI-Treibersoftware unterstützt beliebige 32-Bit Zugriffe auf den lokalen Adreßraum und Burstzugriffe variabler Länge.

Damit beschränkt sich die Entwicklung der Treibersoftware für den DMA-Prozessor darauf, aus den low-level Routinen des PCI-Treibers einen high-level Treiber zu schreiben. Dieser Treiber besitzt für die Befehle des DMA-Prozessor nach Tabelle 2.6 auf Seite 49 jeweils eine eigene Funktion und setzt die Kanalnummer selbständig in die Adresse eines Zugriffs ein. Der PC übergibt dem Treiber die *Word*-Adressen von Speicherzugriffen; diese Adressen werden von Treiber in die *Byte*-Adressen auf dem PCI-Bus umgerechnet. Außerdem setzt die Treibersoftware beispielsweise den Parameter *START* für ein DMA-Programm P nach Abbildung 2.5 auf Seite 45 aus *sadr* und *op* zusammen und berechnet die gehashte PRAM-Startadresse *hpadr* aus *padr* und *hc*. Um ein vollständiges DMA-Programm zu schreiben, müssen die drei Register des DMA-Programms nicht getrennt geschrieben werden; ein Funktionsaufruf des Treibers ruft alle nötigen Befehle des DMA-Prozessors auf.

Der DMA-Treiber besteht aus einer abstrakten Datenklasse `DMAChannel`, die einen DMA-Kanal im DMA-Prozessor repräsentiert und damit auch die Kanalnummer eines DMA-Kanals verwaltet. Über Instanzen dieser Klasse können dann alle Befehle des DMA-Prozessors als eigene Methoden aufgerufen werden. Außerdem werden DMA-Transfers zwischen dem Hauptspeicher des PCs und dem *lokalen RAM des Kanals* unterstützt. Befehle des DMA-Prozessors, die sich nicht auf einen DMA-Kanal beziehen, sind als *statische* Methoden der `DMAChannel`-Klasse implementiert. Diese Methoden lassen sich unabhängig von einer Instanz der `DMAChannel`-Klasse aufrufen. Tabelle 4.1 faßt die Methoden der `DMAChannel`-Klasse zusammen.

Um also beispielsweise eine Standardanweisung $P = (op, padr, sadr, eadr, data)$ mit Modus $op_1 = LOAD$ bei Modulo-Bit *mod* auszuführen, ruft der PC folgende Befehle auf:

Methode	Beschreibung
statische Methoden	
`CheckID()`	ruft *getid* auf und prüft, ob die korrekte ID nach Tabelle 2.6 zurückgegeben wird
`SetHashConstant(hc)`	setzt das HC-Register mit *rhc* auf den Wert `hc`, liest das HC-Register mit *whc* wieder aus und vergleicht den gelesenen Wert mit `hc`
`SetRoutingBitNumber(rbit)`	setzt das $RBIT$-Register mit *setbit* auf den Wert `rbit` mod 16, liest das $RBIT$-Register wieder aus und vergleicht den gelesenen Wert mit `rbit` mod 16
`SetRoundsPerNetworkRound(num)`	setzt das $MODC$-Register mit *setmod* auf den Wert `num` -1, da $\langle MODC[1:0]\rangle + 1$ die Zahl der Teilrunden pro Netzwerkrunde ist
`Reset()`	führt einen Reset des FPGAs aus
`SetDMAConf(offset,size,thresh)`	konfiguriert DMA-Zugriffe; ab `thresh` Bytes erfolgt ein Burstzugriff auf einen Adreßraum ab Adresse `offset`, der `size` Bytes groß ist.
nichtstatische Methoden	
`Get()`	reserviert DMA-Kanal (*getdma*)
`Free()`	gibt DMA-Kanal frei (*freedma*)
`WriteProgram(padr,sadr,aux,op)`	schreibt DMA-Programm (*setpadr*, *setstart* und *setaux*)
`StartProgram(mod,ea)`	startet DMA-Programm (*startdma*)
`TerminateProgram()`	beendet Ausführung eines DMA-Programms (*enddma*)
`WriteDataToPCI(start,num,buf)`	liest `num` Words von der Adresse `buf` im Hauptspeicher PCs und schreibt sie an die relative Adresse `start` im lokalen RAM des Kanals
`ReadDataFromPCI(start,num,buf)`	liest `num` Words von der relativen Adresse `start` im lokalen RAM des DMA-Kanals und schreibt sie an die Adresse `buf` im Hauptspeicher des PCs `start` im lokalen RAM des Kanals
`IsProgramRunning()`	gibt genau dann einen Wert $\neq 0$ zurück, wenn auf dem Kanal ein DMA-Programm läuft (Bit *stat* aus Statuswort nach Abschnitt 2.4.1)
`Error()`	gibt den Error-Code zurück (siehe Tabelle 4.3)

Tabelle 4.1: Methoden der `DMAChannel`-Klasse

Der Parameter `buf` ist vom Typ `int*`, alle anderen Parameter der Methoden sind vom Typ `int`. Alle Methoden geben ein Ergebnis vom Typ `int` zurück. Soweit nicht anders erwähnt, geben alle Funktionen den aktuellen Wert von `Error` zurück.

Zugriffsmodus	Beschreibung
`LOAD`	Lesezugriff
`STORE`	Schreibzugriff
`SYNC_AND, SYNC_OR, SYNC_MAX, SYNC_ADD`	*SYNC*-Zugriff mit Operator *and*, *or*, *max* oder *add*
`MP_AND, MP_OR, MP_MAX, MP_ADD`	Multipräfix-Zugriff mit Operator *and*, *or*, *max* oder *add*

Tabelle 4.2: Zugriffsmodi des DMA-Treibers

```
DMAChannel ch;                                      // abstrakten DMA-Kanal anlegen
ch.Get();                                           // DMA-Kanal reservieren
                                                    //   (getdma)
ch.WriteProgram(padr,sadr,eadr,DMAChannel::LOAD);// DMA-Programm schreiben
                                                    //   (setpadr,setstart,setaux)
ch.StartProgram(mod,0);                             // DMA-Programm starten als
                                                    //   Standardanweisung
                                                    //   (startdma)
while (ch.IsProgramRunning())                       // warten, bis Programm beendet
    {}                                              //   (STAT[ch]=0)
if (ch.Error())                                     // Fehler in SB-PRAM?
    cout << "Fehler in der SB-PRAM!";               //   (ERROR[ch]=1?)
else
    ch.ReadDataFromPCI(sadr,eadr-sadr+1,buf);       // DMA-Transfer nach Adresse
                                                    //   buf im Speicher des PC
ch.Free();                                          // DMA-Kanal freigeben
                                                    //     (freedma)
```

Dabei hasht der Treiber die logische PRAM-Startadresse *padr* selbständig. Nach Ausführung des Programms stehen die gelesenen Daten ab Adresse `buf` im Hauptspeicher des PCs. Die möglichen Werte für den Parameter *op* eines DMA-Programms sind als `enum`-Konstanten innerhalb der Klasse deklariert. Sie können also durch Voranstellen von „`DMAChannel::`“ ausgelesen werden. In Tabelle 4.2 sind diese Zugriffsmodi zusammengefaßt. Mit der Funktion `WriteDataToPCI` kann der PC Daten in das lokale RAM der Karte schreiben für DMA-Programme, die Daten senden. Beim Start einer Einzelanweisung gibt der PC bei der Funktion `StartProgram` als zweiten Parameter eine 1 an und setzt bei `WriteProgram` statt *eadr* den Parameter *data* ein.

Der Übersichtlichkeit halber wurde in obigem Beispiel auf Fehlerüberprüfungen verzichtet. Gibt eine der Funktionen `Get`, `WriteProgram`, `StartProgram`, `ReadDataFromPCI` oder `Free` einen Wert $\neq 0$ zurück, dann ist ein Fehler aufgetreten. Alternativ kann auch nach der Ausführung eines Befehle überprüft werden, ob `Error()` einen Wert $\neq 0$ zurück gibt. In diesem Fall gibt der konkrete Wert von `Error` Aufschluß darüber, welche Fehler aufgetreten sind. In Tabelle 4.3 sind die verschiedenen Fehler zusammengefaßt. Dabei sind die einzelnen Fehlercodes unär kodiert, so daß der tatsächliche Wert von `Error` das logische Oder mehrerer Fehlercodes sein kann. Um beispielsweise zu prüfen, ob der Fehler `NoFreeChannel` aufgetreten ist, führt der PC also den folgenden Test durch:

Fehlercode	Beschreibung
`NoActiveChannel`	Der `DMAChannel`-Instanz ist noch kein DMA-Kanal zugeordnet (`Get` wurde noch nicht aufgerufen).
`WrongID`	Der Befehl *getid* lieferte nicht die erwartet ID
`NoFreeChannel`	$SR[15:0] = 1^{16}$
`ProgramRunning`	$STAT[ch] = 1$
`PRAMError`	$ERROR[ch] = 1$
`noHashConstant`	$\langle HC[31:0] \rangle = 0$
`PRAMnotPresent`	Die SB-PRAM ist nicht angeschlossen oder nicht angeschaltet.
`InitError`	Init wurde noch nicht aufgerufen.
`ChannelAlreadyActive`	`Get` wurde aufgerufen für eine `DMAChannel`-Instanz, der bereits ein DMA-Kanal zugeordnet ist.
`HashConstantError`	Das Setzen der Hashkonstante ist fehlgeschlagen; Lesen der Hashkonstante lieferte einen anderen Wert, als gerade gesetzt wurde.
`RoutingBitError`	Das Setzen der Routing-Bit-Nummer ist fehlgeschlagen.
`InternalError`	Interner Fehler; tritt beispielsweise auf, wenn ein DMA-Kanal freigegeben ist, den der Treiber aber nicht freigegeben hat.
`CardNotPresent`	Es ist keine PCI-Karte angeschlossen.
`CardError`	Fehler beim Zugriff auf PCI-Karte
`BurstError`	Fehler beim Burst-Zugriff auf RAM
`DMAConfError`	Fehler beim Konfigurieren der Burstzugriffe auf das RAM

Tabelle 4.3: Fehlercodes des DMA-Treibers

Die Fehler *NoFreeChannel*, *ProgramRunning* und *noHashConstant* treten auf, wenn bei der Ausführung eines Befehls die Bedingungen aus Tabelle 2.6 nicht erfüllt sind.

```
if (ch.Error() & DMAChannel::NoFreeChannel)
{
    cout << "Es ist kein DMA-Kanal mehr frei!";
    return;
}
```

Da die einzelnen Fehlercodes als `enum`-Konstanten innerhalb der `DMAChannel`-Klasse definiert sind, erfolgt der Zugriff durch Voranstellen von „`DMAChannel::`“. Dadurch wird auch eine namentliche Kollision mit den Konstanten anderer Treiber vermieden.

Kapitel 5

Schaltungsverifikation

Im Rahmen dieser Arbeit wurde ein komplexes Design aus einem DMA-Prozessor und einer Sortiereinheit entworfen. Um dieses vorgestellte Design unabhängig von der SB-PRAM und der Mezzaninkarte zu testen, haben wir anstelle der SB-PRAM ein kleines Design in das FPGA eingebaut, das die SB-PRAM simulieren soll. Dieser *PRAM-Simulator* gibt auf diejenigen Speicheranfragen der Sortiereinheit eine Antwort, die eine Antwort erwarten; als Antwortdatum wird die gehashte PRAM-Adresse der Speicheranfrage zurückgegeben. Mit Hilfe dieses PRAM-Simulators läßt sich das Design als Ganzes weitgehend testen.

Die synchronen Teile der Schaltung lassen sich mit Hilfe eines Logik-Simulators testen. Mit diesem Simulator können beliebige Signale stimuliert werden und die dadurch entstehenden Ergebnisse abgefragt werden. Eine korrekte Simulation einer synchronen Schaltung impliziert einen entsprechend korrekten Ablauf in konkreter Hardware. Asynchrone Schaltungsteile lassen sich mit Hilfe des Simulators nur unzureichend testen; hier ist man auf einen Test an der konkreten Hardware angewiesen.

5.1 PRAM-Simulator

Das Interface zwischen PRAM-Simulator und Sortiereinheit ist genau das Interface zwischen SB-PRAM und Sortiereinheit nach Abschnitt 3.1. Der PRAM-Simulator erkennt eintreffende Speicheranfragen; er unterscheidet dabei insbesondere zwischen dem Adreßteil und dem Datenteil einer Speicheranfrage. *GHOST*-Pakete ignoriert er. Der PRAM-Simulator liefert genau dann eine Antwort auf eine Speicheranfrage, wenn die Speicheranfrage eine Antwort erwartet. Das Antwortdatum ist dabei genau die Adresse der besagten Speicheranfrage. Das Versenden der Antwort an die Sortiereinheit erfolgt in 2 Halbpaketen, für die jeweils eine eigene Parity-Berechnung erfolgt. Die untere Hälfte des Antwortdatums wird als erstes Halbpaket versandt. Handelt es sich bei einer Speicheranfrage um einen Multipräfixzugriff mit dem Operator max, so werden die beiden Halbpakete der Antwort in umgekehrter Reihenfolge geschickt.

Der PRAM-Simulator reagiert auf ein aktives *bbusy* der Sortiereinheit frühestens nach 8 Takten. In diesen 8 Takten kann der Simulator jeweils noch ein Halbpaket an die Sortiereinheit schicken. Nimmt die Sortiereinheit das Signal *bbusy* wieder zurück, kann der

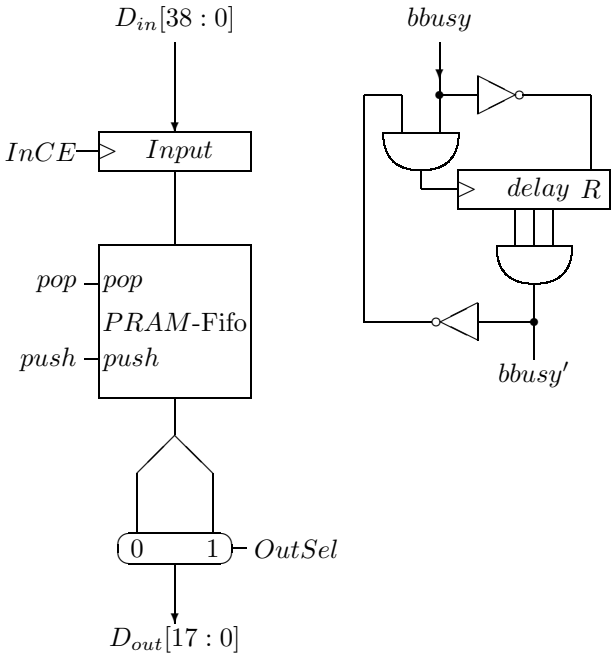


Abbildung 5.1: Datenpfade des PRAM-Simulators

PRAM-Simulator schon im nächsten Takt das nächste Halbpaket schicken. Der PRAM-Simulator aktiviert beide *NWbusy*-Leitungen des Sortierers gemeinsam, wenn er zur Zeit keine Speicheranfragen mehr bearbeiten kann. Es werden also keine zwei verschiedenen *NWbusy*-Signale simuliert.

Diese Funktionsweise des PRAM-Simulators wird mit den Datenpfaden aus Abbildung 5.1 und einer geeigneten Kontrollogik realisiert. Der Adreßteil einer eintreffenden Speicheranfrage wird im *Input*-Register gespeichert. Erst wenn der zugehörige Datenteil beziehungsweise der Adreßteil der nächsten Speicheranfrage eintrifft, wird das Datum im *Input*-Register gegebenenfalls in die *PRAM*-Fifo geschrieben. Dann steht nämlich erst fest, ob es sich um einen Multipräfixzugriff mit Operator *max* handelt, denn der Operator wird zusammen mit dem Datenteil einer Speicheranfrage verschickt. Die *PRAM*-Fifo nimmt deshalb neben dem Antwortdatum noch ein zusätzliches Bit auf, das beim Auslesen darüber entscheidet, welche Hälfte zuerst versendet wird.

Der *Output*-Multiplexer wählt eine der beiden Hälften des Datums am Ausgang der *PRAM*-Fifo aus, die an die Sortiereinheit geschickt wird. Sind beide Hälften versandt, wird das Datum aus der *PRAM*-Fifo ausgelesen. Ist die *PRAM*-Fifo voll, so aktiviert der PRAM-Simulator beide *NWbusy*-Signale. Damit wird gewährleistet, daß keine Speicheranfragen der Sortiereinheit verloren gehen, denn die Sortiereinheit sendet in diesem Fall keine Speicheranfragen mehr, sondern nur noch *GHOST*-Pakete, die der PRAM-Simulator ignorieren darf.

Der 3-Bit-Zähler *delay* aus Abbildung 5.1 dient dazu, ein *aktives bbusy*-Signal der Sor-

tiereinheit 7 Takte lang zu verzögern und ein *inaktives bbusy* mit einem Takt Verzögerung durchzureichen. Ist das Kontrollsignal $bbusy'$ aktiv, dann sendet der PRAM-Simulator keine Halbpakete mehr an die Sortiereinheit. Dadurch können wir also testen, ob die Sortiereinheit noch 8 Halbpakete aufnehmen kann, wenn sie das Kontrollsignal *bbusy* aktiviert.

$$delay[2:0] := \begin{cases} 000 & \text{falls } bbusy = 0 \\ \text{bin}_3(\langle delay[2:0]\rangle + 1) & \text{falls } bbusy = 1 \text{ und } \langle delay[2:0]\rangle \neq 7 \\ delay[2:0] & \text{sonst} \end{cases}$$

$$bbusy' = \delta_{\langle delay[2:0]\rangle,7}$$

5.2 Simulation des Gesamtdesigns

Der Hinweg und der Rückweg der Sortiereinheit lassen sich zunächst getrennt simulieren. Die Sortiereinheit als Ganzes läßt sich am besten in Zusammenhang mit dem PRAM-Simulator testen. Man kann dann beliebige Folgen von Speicheranfragen in die Sortiereinheit einspeisen und überprüfen, ob die richtigen Antworten den Sortierer in der richtigen Reihenfolge verlassen.

Die Ausführung einzelner Befehle im DMA-Prozessor läßt sich mit geringen Aufwand testen; die Bedingungen und der spezifizierte Effekt sind leicht überprüfbar. Interessant wird es, wenn ganze DMA-Programme ausgeführt werden. Dann ergeben sich insgesamt wieder die besten Testmöglichkeiten, wenn man mit der Sortiereinheit und dem PRAM-Simulator zusammen simuliert. So läßt sich leicht feststellen, ob bei der Ausführung eines DMA-Programms die richtigen Schreibzugriffe auf das lokale RAM erfolgen, ob also die richtigen Daten an die richtigen Adressen geschrieben werden. Die richtigen Daten sind dabei die gehashten PRAM-Adressen; die Daten von zwei aufeinanderfolgenden Adressen unterscheiden sich also genau um die Hashkonstante. Zusätzlich kann man noch testen, ob die Speicheranfragen beim Verlassen der Sortiereinheit die richtigen Datenteile haben. Diese Datenteile werden beim Ausführen von DMA-Programmen aus dem lokalen RAM gelesen.

5.3 Test des Gesamtdesigns

Bei Test des Gesamtdesigns auf der PCI-Karte geht man Schritt für Schritt vor. Zunächst werden einfache Befehle getestet. Dadurch wir die korrekte Implementierung des lokalen Busprotokolls der PCI-Karte sichergestellt. Anschließend kann das lokale RAM-Interface getestet werden; dadurch werden erstmals asynchrone Schaltungsteile getestet, denn das RAM-Interface arbeitet mit einer eigenen Clock. Zuletzt werden die komplexen Befehle und die Ausführung von DMA-Programmen getestet. Dazu wird die PRAM-Clock zunächst synchron zur PCI-Clock betrieben, um eine grundlegende Korrektheit der Schaltung zu testen. Abschließend wird dann auch das PRAM-Interface asynchron zur PCI-Clock betrieben.

Die folgende Testreihe stellt sicher, daß alle Befehle außer *startdma*, *enddma*, *setpadr*, *setstart* und *setaux* fehlerfrei funktionieren. Die vollständige Funktionalität dieser 5 Be-

fehle kann erst in Zusammenhang mit dem lokalen RAM getestet werden. Wenn es bei den einzelnen Punkten offensichtlich ist, was getestet wird, so wird kein Test erwähnt.

1. Teste, ob bei dem Befehl *getid* der erwartete Wert zurückgegeben wird.
2. Lies das Statuswort und vergleiche dessen Inhalt mit dem erwarteten Wert nach einem Reset.
3. Schreibe eine Hashkonstante $\neq 0$ mit *whc* und lies sie mit *rhc* wieder zurück. Lies das Statuswort nochmals und vergleiche, ob jetzt $noHC = 0$ ist.
4. Setze die Routing-Bit-Nummer mit *setbit* und lies sie mit *status* wieder zurück.
5. Setze die Zahl der Teilrunden pro Netzwerkrunde mit *setmod* und lies sie mir *status* wieder zurück. Dabei kann es bis zu $24 \cdot 32$ Takte dauern, bis die Zählung des Modulo-Bits umgestellt wird, da nur alle 24 Teilrunden umgeschaltet werden kann.
6. Reserviere einen DMA-Kanal mit *getdma* und lies anschließend das Statuswort wieder aus.
7. Reserviere 16 weitere DMA-Kanäle. Beim Reservieren des insgesamt 17. DMA-Kanals sollte ein Fehler auftreten und im Statuswort $all = 1$ sein. Bei nochmaligem Lesen des Statusworts sollte immer noch $all = 1$ sein.
8. Gib die DMA-Kanäle nacheinander mit *freedma* wieder frei. Beim Versuch, einen Kanal mehrfach freizugeben, passiert nichts.

Der erste Schritt stellt sicher, daß der DMA-Prozessor richtig konfiguriert ist und das Interface zum lokalen Bus zumindest grundlegend funktioniert. Schritt 2 prüft, ob das Status-Wort gelesen werden kann. Schritt 3 testet das HC-Register und die Korrektheit des Bits $noHC$ im Statuswort. Anschließend werden das $RBIT$- und das $MODC$-Register getestet. Die übrigen Schritte stellen sicher, daß das Reservieren und Freigeben von DMA-Kanälen ordnungsgemäß funktioniert. Dabei werden insbesondere Teile der Bedingungen für diese Befehle getestet.

Anschließend wird das lokale RAM getestet. Wird die folgende Testreihe ohne Fehler durchgeführt, gehen wir davon aus, daß das RAM-Interface korrekt arbeitet.

1. Schreibe eine 0 an Adresse 0 im lokalen RAM und lies den Inhalt der Adresse 0 wieder aus.
2. Schreibe die Daten 2^i, $i = 0 \ldots 31$ (*rolling-1-data*) an Adresse 0 im lokalen RAM und lies jedes einzelne geschriebene Datum wieder aus.
3. Wähle ein beliebiges Datum und schreibe es an die Adressen 2^i, $i = 0 \ldots 17$ (*rolling-1-address*) und lies diese Adressen wieder aus.
4. Schreibe 8 Daten im Burst ab Adresse 0 und lies die Daten wieder im Burst aus. Schreibe ein anderes Datum an Adresse 0 und lies wieder alle 8 Daten im Burst aus. Vergleiche, ob sich nur genau das Datum an Adresse 0 geändert hat.

5. Schreibe 16 Daten im Burst ab Adresse 0 und lies die Daten wieder im Burst aus. Schreibe 8 andere Daten im Burst ab Adresse 0 und lies alle 16 Daten wieder im Burst aus.

6. Initialisiere das vollständige RAM im Burst und lies es vollständig wieder aus.

Die ersten beiden Schritte stellen sicher, daß bei einem Einzelzugriff auf das lokale RAM zumindest dieses eine Datum geschrieben wird. Schritt 3 stellt sicher, daß Zugriffe mit der korrekten Adresse erfolgen. Anschließend erfolgt ein grundlegender Test für Burstzugriffe und es wird überprüft, ob bei einem Einzelzugriff nicht mehr als ein Datum geschrieben wird. Der folgende Test 5 stellt sicher, daß auch bei Bursts nicht mehr Daten geschrieben werden als beabsichtigt. Der letzte Schritt schließlich besteht aus sehr vielen Zugriffen, um eventuelle Effekte durch den asynchronen Betrieb des lokalen RAMs festzustellen. Dabei bietet es sich *nicht* an, an jede Adresse des lokalen RAMs jeweils die Adresse selbst als Datum zu schreiben, da Adressen und Daten auf der PCI-Karte auf dem selben Bus anliegen. Statt dessen schreibt man beispielsweise an jede Adresse das bitweise Inverse der Adresse als Datum.

Jetzt sind wir in der Lage, die korrekte Ausführung von DMA-Programmen zu testen.

1. Initialisiere das lokale RAM mit Daten.

2. Reserviere einen DMA-Kanal, schreibe ein DMA-Programm auf diesem Kanal, das keine Antwort erwartet, und starte es als Standardanweisung. Lies den Status dieses Kanals, bis das DMA-Programm beendet ist. Es sollte beim Ausführen des DMA-Programms kein Fehler aufgetreten sein. Lies anschließend das komplette RAM zurück. Der Inhalt sollte sich nicht geändert haben.

3. Führe denselben Test mit einem DMA-Programm durch, das eine Antwort erwartet. Im RAM sollten dann genau von der Startadresse bis zur Endadresse auf der Speicherseite des DMA-Kanals andere Daten stehen als vorher. Das erste Datum sollte die gehashte PRAM-Startadresse des DMA-Programms sein, die nachfolgenden Daten die gehashten Folgeadressen.

4. Führe die beiden Tests 2. und 3. anschließend für Einzelanweisungen durch.

5. Teste die korrekte Ausführung mehrerer DMA-Programme

6. Teste die korrekte Ausführung von DMA-Programme, während gleichzeitig auf das lokale RAM zugegriffen wird. Diese Zugriffe auf das lokal RAM müssen dabei auch getestet werden.

In Schritt 2 sollte die Länge des DMA-Programms nicht zu klein sein. Beim Auslesen des Status für das DMA-Programm unmittelbar nach dem Start sollte man beobachten können, daß das DMA-Programm läuft. Der Schritt 2 stellt sicher, daß das DMA-Programme terminiert und keine Daten ins lokale RAM geschrieben werden, wenn das DMA-Programm keine Antwort erwartet. Schritt 3 zeigt dann, daß die Register *PADR*,

START und *AUX* des DMA-Programms korrekt geschrieben wurden und daß das DMA-Programm genau die Schreibzugriffe im lokalen RAM hervorruft, die spezifiziert sind. Die nächsten beiden Tests beschäftigen sich mit Einzelanweisungen beziehungsweise mehreren DMA-Programmen. Der letzte Test stellt schließlich sicher, daß während der Ausführung von DMA-Programmen korrekte Speicherzugriffe auf das lokale RAM erfolgen können.

In der letzten Version des Designs liefen die beiden ersten Testreihen fehlerfrei, während in der dritten Testreihe bei der Ausführung mehrerer DMA-Programme Fehler auftraten. Bei 8 parallel ausgeführten DMA-Programmen, die insgesamt mehr als 1 MByte Daten ins lokale RAM schrieben, ergaben sich genau 19 falsche Daten, die von 2 Bursts herrührten, die an eine falsche Adresse erfolgten. Die beiden *falschen* Bursts der Länge 8 produzierten 16 Daten an Adressen, an denen sie nicht erwartet waren, während die 3 eigentlich zu burstenden Daten nicht an die erwartete Adresse geschrieben wurden.

Der zugehörige Fehler konnte mit Hilfe des Logik-Simulators lokalisiert und beseitigt werden; allerdings war es nicht mehr möglich, das verbesserte Design zu testen. Auch nach mehreren Versuchen konnte dieses neue Design nicht mit einer lokalen Clock von 33MHz getaktet werden; die Design-Software ermittelte für das Design auf dem verwendeten FPGA eine maximale Taktfrequenz von nur 29.17MHz. Ein schnelleres FPGA stand zu diesem Zeitpunkt nicht zur Verfügung. Durch das Gesamtdesign inklusive des PRAM-Simulators ist das verwendete FPGA zu 63% belegt. Auch ein größeres FPGA stand nicht zum Test zur Verfügung.

Ausblick

Ziel dieser Arbeit war das Design einer PCI-Karte als Interface zur SB-PRAM. Es sollte dabei möglich sein, über mehrere sogenannte DMA-Kanäle parallel auf den Speicher der SB-PRAM zuzugreifen. Ein möglichst großer Datendurchsatz sollte erreicht werden und für spezielle Zugriffe auch eine möglichst kleine Latenz.

Das vorgestellte Design ermöglicht die parallele Ausführung von DMA-Programmen auf 16 DMA-Kanälen. Dabei werden auch PRAM-spezifische Multipräfixoperationen unterstützt. Der DMA-Prozessor unterscheidet 2 Arten von DMA-Programmen, sogenannte Einzel- und Standardanweisungen. Die Verteilung von Einzel- und Standardanweisungen auf die 16 DMA-Kanäle ist dabei vollständig frei. Einzelanweisungen werden bei der Ausführung von DMA-Programmen priorisiert, so daß sich nur eine kleine Latenz ergibt. Standardanweisungen werden in Teilprogramme zerlegt, die fast alle aus 8 Speicheranfragen bestehen. Die Speicheranfragen eines solchen Teilprogramms bilden eine Teilrunde im Netzwerk der SB-PRAM. Da eine solche Teilrunde aus maximal 8 Speicheranfragen bestehen kann, ergibt sich damit ein maximaler Datendurchsatz innerhalb einer Teilrunde. Durch den gepipelineten Algorithmus zur Ausführung von DMA-Programmen wird gewährleistet, daß möglichst wenige leere Teilrunden an die SB-PRAM gesendet werden. Insgesamt ergibt sich zwischen DMA-Prozessor und SB-PRAM genau derselbe Datendurchsatz wie zwischen PRAM-Prozessor und SB-PRAM.

Durch die Simulation eines Modulo-Bits des DMA-Prozessors synchron zum Modulo-Bit der SB-PRAM kann verhindert werden, daß in einer Netzwerkrunde sowohl lesend als auch schreibend auf eine Adresse im gemeinsamen Speicher der SB-PRAM zugegriffen wird. Die Routing-Bit-Nummer der ersten Netzwerkstufe der SB-PRAM ist frei programmierbar, so daß ein Anschluß der PCI-Karte an verschiedene Konfigurationen der SB-PRAM (z.B. 16-Prozessor-PRAM oder 64-Prozessor-PRAM) möglich ist. Als Hashkonstante kann im DMA-Prozessor die Hashkonstante der SB-PRAM einprogrammiert werden.

Für die synchronen Teile der Schaltung wurden größtenteils Korrektheitsbeweise angegeben. So wurde gezeigt, daß der Instruktionssatz des DMA-Prozessors korrekt interpretiert wird und daß der DMA-Prozessor den erarbeiteten gepipelineten Algorithmus zur Ausführung von DMA-Programmen realisiert. Da ein Test des Designs in Zusammenhang mit der SB-PRAM nicht möglich war, wurde ein zusätzliches Schaltwerk designt, welches das Verhalten der SB-PRAM simuliert. In Zusammenhang mit diesem PRAM-Simulator wurde dann die Ausführung mehrerer DMA-Programme auf der PCI-Karte getestet. Dabei wurden einige Fehler im Design zwischen Registern mit verschiedenen Clocks lokalisiert

und beseitigt. So werden beispielsweise alle Register, die mit einer Clock b betrieben werden und mit einer Clock a ausgelesen werden, zunächst noch einmal in einem Register mit Clock a registriert, bevor sie weiter mit der Clock a verwendet werden.

In der letzten Version des DMA-Prozessors wurden bei der parallelen Ausführung von 8 DMA-Programmen nur 19 von mehr als einer Million Daten falsch geschrieben; wie im letzten Kapitel beschrieben, konnte auch dieser Fehler lokalisiert und beseitigt werden. Allerdings war ein Test mit einer neuen Version des Designs nicht mehr möglich, da das Design auf dem verwendeten FPGA nur mit einer lokalen Clock von 29.17MHz betrieben werden konnte. Wir gehen aber davon aus, daß auch die letzte Testreihe in Zusammenhang mit dem PRAM-Simulator aus dem vorherigen Kapitel jetzt durchläuft.

Eine vollständige Verifikation des Designs kann natürlich nur in Zusammenhang mit der SB-PRAM erfolgen. Dazu müßte die vorgestellte Aufsatzkarte gefertigt werden. Anschließend kann der DMA-Prozessor vollständig überprüft werden, und die PCI-Karte kann wirklich eingesetzt werden. Dabei läßt sich die PCI-Karte beispielsweise als Grafik-Karte der SB-PRAM einsetzen. Daten, die von den Prozessoren der SB-PRAM im gemeinsamen Speicher berechnet werden, können über die PCI-Karte mit großem Datendurchsatz ausgelesen und graphisch dargestellt werden.

Literaturverzeichnis

[ADK93] Ferri Abolhassan, Reinhard Drefenstedt, Jörg Keller, Wolfgang J. Paul, Dieter Scheer: *On the physical design of PRAMs*, The Computer Journal, 36(8):756-762, 12/1993.

[BBFFGL97] Peter Bach, Michael Braun, Arno Formella, Jörg Friedrich, Thomas Grün, Cederic Lichtenau: *Building the 4 Processor SB-PRAM Prototype*, Proceedings of the Hawaii 30th International Symposium on System Science HICSS-30, pages 14-23, 01/1997.

[Ja00] Stefan Janocha: *Die PCIPro-Karte - Entwurf und Realisierung eines Systems zur schnellen Datenmanipulation*, Diplomarbeit, FB14 Informatik, Universität des Saarlandes, Saarbrücken, 2000.

[HP90] J. L. Hennessy, D. A. Patterson: *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, San Mateo, California, 1990.

[Gö96] T. Göler: *Der Sortierknoten der SB-PRAM*, Diplomarbeit, FB14 Informatik, Universität des Saarlandes, Saarbrücken, 1996.

[AKP91b] Ferri Abolhassan, Jörg Keller, Wolfgang J. Paul: *On the cost-effectiveness and realization of the theoretical PRAM model*, SFB-Report (SFB-124-D4) 09/1991, FB14 Informatik, Universität des Saarlandes, Saarbrücken, 1991.

[Ba96] P. Bach: *Entwurf und Realisierung der Prozessorplatine der SB-PRAM*, Diplomarbeit, FB14 Informatik, Universität des Saarlandes, Saarbrücken 1996.

[PLX98] PLX Technologies: *PCI 9080 Data Sheet*, Version 1.04, 01/1998.

[Pa78] Wolfgang J. Paul: *Komplexitätstheorie*, Teubner Studienbücher Informatik, Stuttgart 1978.

[MT98] Micron Technologies Inc.: *Synchronous DRAM: MTL48LC4M16A1/A2 - 1 Meg* $\times$ *16* $\times$ *4 banks*, Rev. 5/98, 1998.

[MP95] Silvia M. Müller, Wolfgang J. Paul: *The Complexity of Simple Computer Architectures*, LNCS 995, Springer, Berlin, 1995.

[AKP91a] Ferri Abolhassan, Jörg Keller, Wolfgang J. Paul: *On the cost-effectiveness of PRAMs*, Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing, pages 2-9, IEEE Computer Society Press, Los Alamitos, California, 12/1991.

[LMW86] Jacques Loeckx, Kurt Mehlhorn, Reinhard Wilhelm: *Grundlagen der Programmiersprachen*, Teubner-Texte zur Informatik, Stuttgart 1986.

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

[CMC95] Bus Architecture Standards Committee of the IEEE Computer Society: *Draft Standard for a Common Mezzanine Card Family: CMC*, P1386, Draft 2.0, 04/1995.

[Sch95] Dieter Scheerer: *Der Prozessor der SB-PRAM*, Dissertation, FB14 Informatik, Universität des Saarlandes, Saarbrücken 1995.

[FPGA99] Xilinx Inc.: *XC4000 XLA/XV Field Programmable Gate Arrays*, DS015 Version 1.2, 05/1999.

[PCI95] PCI Special Interest Group: *PCI Local Bus Specification Rev. 2.1*, Portland, Oregon, 06/1995.

[KP95] Jörg Keller, Wolfgang J. Paul: *Hardware Design: formaler Entwurf digitaler Schaltungen*, Teubner-Texte zur Informatik, Band 15. Teubner, Stuttgart 1995.

[Wa97] T. Walle: *Das Netzwerk der SB-PRAM*, Dissertation, FB14 Informatik, Universität des Saarlandes, Saarbrücken 1997.

[XA96] Peter Alfke: *Synchronous and Asynchronous FIFO Designs*, Xilinx Application Note XAPP 051, Version2.0, 09/1996.