

28.03.2008

Leitfaden zum Praktikum “Adaptive Rechensysteme” im SS 2008

Herzlich willkommen beim Praktikum über adaptive Rechensysteme. Dieser Leitfaden soll Ihnen den Einstieg in die Benutzung der Entwurfsumgebung und die Bearbeitung der eigentlichen Aufgaben erleichtern.

Das vorliegende Dokument ist in vier Teile gegliedert: Der erste beschreibt den Praktikumsablauf mit seinen einzelnen Phasen und Teilaufgaben, der zweite (den sie allerdings zuerst lesen sollten!) führt in den verwendeten Entwurfsfluß ein. Der dritte gibt eine Einführung in adaptive Rechner, die sie verstanden haben sollten. Für den Fall, daß Sie dieses Werk ohne nebenstehenden Rechner lesen möchten, sind im letzten Kapitel die Quellen aller besprochenen Programme abgedruckt.

Noch eine Bemerkung vorweg: Um Ihnen den Einstieg zu erleichtern und Sie behutsam an die Materie heranzuführen, beginnt das Praktikum mit vergleichsweise einfachen Aufgaben. Das Niveau steigt aber gegen Ende! Diesen Gradienten sollten Sie unbedingt bei Ihrer Zeit- und Arbeitsintensitätsplanung berücksichtigen

Bitte lassen Sie uns wissen, wo Probleme auftreten, damit wir für nachfolgende Jahrgänge Abhilfe schaffen können.

1 Praktikumsablauf

Hardware und Arbeitsraum

An jedem der Arbeitsplatzrechner im Praktikumsraum des FG ESA ist ein eigenes ML310 ACS angeschlossen, das jeweils nur von *einem* Benutzer parallel benutzt werden kann. Überprüfen Sie daher bitte mit dem Kommando `who`, ob Sie Ihren ausgewählten Rechner wirklich für sich haben! Da bei Fehlprogrammierungen (und gelegentlich [aber selten] im Regelbetrieb) das ACS den Betrieb einstellt, ist es hilfreich, die Tests mit echter Hardware *vor Ort* in unserem Praktikumsraum zu machen. Sprechen Sie dazu ein Mitglied des FG an, wir schließen Ihnen gerne auf!

0. Phase: Einführung

3.4.-6.4.2008

Am Donnerstag, den 3.4.2008 um 14:30 Uhr findet im Raum S2|02 E102 (Piloty-Gebäude Hochschulstr. 10) ein erstes gemeinsames Treffen aller Praktikumssteilnehmer statt.

Vor Lektüre dieses Kapitels sollten Sie unbedingt zwei weitere Kapitel gelesen haben: Essenziell ist das Verständnis des dritten Kapitels über adaptive Rechensysteme. Dieses nimmt auch Bezug auf die CAD-Entwurfsumgebung, die in Kapitel 2 dieses Leitfadens beschrieben wird, das Sie daher ebenfalls lesen sollten.

Falls Sie noch gar keine Vorkenntnisse über die Programmiersprache C besitzen, sollten Sie eine der vielen Einführungen überfliegen. Neben zahllosen Büchern seien hier ohne Wertung die vom HRZ vertriebene Einführung „Die Programmiersprache C - Ein Nachschlagewerk“ (in den Nutzerbüros des HRZ für EUR 3,70 erhältlich) oder die Web-Seiten

http://www.uni-giessen.de/hrz/software/programmiersprachen/C/c_teil1.html

genannt.

Für eine Auffrischung der Verilog-Kenntnisse sei auf die Veranstaltung am 4.4. in S2|02 E102 verwiesen, eine Einführung gibt es z.B. unter

<http://www.asic-world.com/verilog/index.html>

Bei dem ersten Treffen liegt der Schwerpunkt neben den organisatorischen Details (wie Kolloquiumsterminen) bei der fachlichen Einführung. Der Praktikumsleitfaden wird besprochen und Fragen werden beantwortet. Sehr wichtig ist auch die Vorführung der im Praktikum verwendeten Entwicklungswerkzeuge. Weiterhin wird der praktische Umgang mit dem adaptiven Rechner ML310 gezeigt, den Sie als Zielplattform für Ihre Experimente nutzen werden.

1. Phase: Hardware-Experimente

7.4.-20.4.2008

In dieser Woche arbeiten Sie das erste Mal selber mit den Entwicklungswerkzeugen und der Hardware. Sie werden die im Skript vorgestellten Beispiele simulieren, synthetisieren und auf dem ML310 erproben. Abgaben werden hier noch keine von Ihnen erwartet. Gehen Sie dabei wie folgt vor:

1. Legen Sie mit `mkslave` ein neues Slave-Mode-Projekt an und erproben Sie die verschiedenen in der Werkzeugeinführung (Kapitel 2) erklärten Arbeitsschritte. Sie sollten also die Funktionsfähigkeit der Anwendung sowohl in Simulation (RTL und Post-Layout) als auch in realer Hardware auf dem ML310 erproben. Letzteres soll nicht nur durch Ausführen von `./main`, sondern auch interaktiv mit dem Werkzeug `xmd` erfolgen.
2. Machen Sie analoge Experimente mit einer durch `mkmaster` angelegten Master-Mode-Anwendung. Verzichten Sie hier aber auf den interaktiven Test mit `xmd`, sondern nehmen die reale Erprobung nur durch Starten von `./main` vor.
3. Erweitern Sie Ihre in 1. angelegte Slave-Mode-Anwendung auf die Spiegelung von 32b Worten wie in Abschnitt 3.3.2 beschrieben. Nehmen Sie die gleichen Simulationen und Tests vor.
4. Erweitern Sie auch Ihre in 2. angelegte Master-Mode-Anwendung auf die Spiegelung von 32b Worten (Abschnitt 3.3.3). Erstellen Sie Testdaten mit einem Texteditor im `ReadMemFile` Format (siehe Abschnitt 2.2), die Sie dann während der Simulation in den simulierten Speicher einlesen. Schreiben Sie die Ausgabedaten mit `WriteMemFile` in eine Datei.

Berücksichtigen Sie für diese und alle weiteren Phasen folgende Regeln beim Hardware-Entwurf:

- Verwenden Sie nur positiv flankengetriggerte Flip-Flops (`@posedge`). Sonderwünsche müssen vorher mit dem betreuenden Assistenten durchgesprochen werden.
- *Alle* Register müssen nach einem Reset auf definierte Werte gesetzt werden (`if (RESET) begin . . .`).
- Ein Register darf nur in exakt einem `always`-Block Ziel einer Zuweisung sein.
- Interne Tristate-Buffer (also das explizite Setzen eines Signals bzw. Registers auf den Wert Z) sind verboten.
- Verilog `register`-Arrays dürfen nicht verwendet werden. Wenn Sie partout in Ihrem Entwurf größere Zwischenspeicher brauchen, halten Sie bitte mit dem betreuenden Assistenten Rücksprache.

Nach dieser Phase sollten sie den praktischen Umgang mit den Werkzeugen beherrschen und auch schon erste Erfahrungen mit der Arbeit auf der ML310 Hardware haben.

2. Phase: Messungen

21.4.-27.4.2008

Hier werden Sie die Slave-Mode Anwendung `reverse`, die Sie in der letzten Phase erstellt haben, um Messpunkte erweitern. Ziel ist es zu bestimmen, wie effizient der Datentransfer im Slave-Mode zwischen CPU und RC erfolgt. Dazu werden die maximalen und minimalen Zeiten zwischen zwei CPU-Zugriffen in der Hardware gemessen. Der Software-Teil muß diese Ergebnisse auslesen und dem Benutzer ausgeben. Gehen Sie dabei wie folgt vor:

1. Sie müssen den Hardware-Teil um zwei durch die Software lesbare Register erweitern. In einem steht die minimale, in dem anderen die maximale Zeit (in Takten) zwischen zwei Zugriffen.
2. Die Zeit zwischen zwei Zugriffen von der Software auf die RC muß durch einen Hardware-Zähler in Takten gemessen werden.

3. Nach einem Zugriff müssen die minimalen und maximalen Werte mit dem gerade gestoppten Wert des Zählers aktualisiert werden.
4. An Zugriffen sollen sie in drei Schleifen in der Software folgende Muster realisieren: Nur aufeinanderfolgende Lese-Operationen, nur aufeinanderfolgende Schreib-Operationen, abwechselnd je eine Lese- und eine Schreib-Operation.
5. Nehmen Sie die Messungen getrennt für jedes Zugriffsmuster vor.
6. Simulieren Sie Ihren Entwurf auf RT-Ebene.
7. Passen Sie den Software-Teil so an, daß die gemessenen Werte von der RC zurückgelesen und dem Benutzer ausgegeben werden (C-Funktion `printf`, getrennt für jedes Zugriffsmuster).
8. Compilieren Sie die gesamte Anwendung und erproben Sie `./main`.

Beachten Sie bei der Realisierung der Messungen folgende Details:

- Ihre Schaltung kann von der CPU mit sogenannten Burst-Transfers angesprochen werden. Dabei bleibt das ADDRESSED-Signal über mehrere aufeinanderfolgende Takte gesetzt. Jeder Takt wird dabei als getrennter Zugriff bearbeitet. Gegebenenfalls wechselt dabei auch der Wert auf dem ADDRESS-Bus, wenn die CPU verschiedene Adressen während des Bursts bearbeitet (schreibt oder liest). Für Ihre Zeitmessungen soll ein Burst-Transfer gleich welcher Länge aber nur als *ein* Zugriff gewertet werden.
- Schreibzugriffe von der CPU auf Ihre Hardware von der CPU können ebenfalls mehrere Takte dauern. Dabei bleibt das WRITE-Signal über mehrere aufeinanderfolgende Takte gesetzt. Wie bei Burst-Transfers soll Ihre Zeitmessung auch einen Multi-Takt-Schreibvorgang nur als *einen* Zugriff ansehen.
- Sie sollen also bei deaktiviertem RESET-Signal die Zeit zwischen zwei aufeinanderfolgenden steigenden Flanken des ADDRESSED-Signals messen.

Abgaben: Eine Erläuterung Ihrer Meßmethodik, das erweiterte HDL-Modell und C-Programm sowie die Meßergebnisse.

An dieser Stelle soll kurz auf die Art und Weise der Abgaben eingegangen werden. Mit Ausnahme der 6. Phase müssen lediglich die verlangten Angaben zusammengestellt sowie mit kurzen Erklärungen versehen werden. Die Erklärungen müssen auch klare Hinweise darauf enthalten, in welchem Verzeichnis Ihres Accounts die angesprochenen Dateien (HDL-Modelle, Simulationsdaten, etc.) liegen. Dies ist für die Durchführung von getrennten Abnahmetests erforderlich.

Vor Ort wird die Funktionsfähigkeit Ihrer Lösungen entsprechend den Anforderungen der Aufgabe durch Ihren Hiwi überprüft. Sprechen Sie diesen dazu *vor* dem Abgeben der schriftlichen Ausarbeitung an! Nicht lauffähige Abgaben werden nicht anerkannt!

Die Abgaben selbst erfolgen als E-Mail an `lange@esa.informatik.tu-darmstadt.de`, beides mit der Betreffzeile `Praktikum Gruppe NN Phase M`, wobei `NN` die Gruppennummer und `M` die Nummer der Phase ist. Die einzelnen Teile der Abgabe sind als Anhänge an diese E-Mail angehängt. Waveforms werden dabei als PostScript-Dateien (erstellt mit Print Only To File in VirSim) dargestellt. Texte als reine Text-Dateien (kein MS WORD o.ä.). In *allen* Dateien geben Sie bitte ebenfalls Ihre Gruppennummer, die Phase, sowie das Datum an. Bitte vergessen Sie nicht, die Kommentare in den Quelltexten an die aktuelle Abgabe anzupassen.

3. Phase: Bildbearbeitung

28.4.-4.5.2008

Als Kernaufgabe in diesem Praktikum werden wir uns mit einem einfachen Problem aus der Bildbearbeitung befassen. Es geht darum, den Kontrast in Graustufenbildern zu erhöhen. Solche Graustufenbilder werden auf dem Rechner als ein zweidimensionales Feld von Zahlen dargestellt, bei dem jede Zahl die Helligkeit des entsprechenden Bildpunktes angibt. In unserem Beispiel sind diese Werte 8b breit, der Wert

0 entspricht dabei vollständiger Schwärze, der Wert 255 dem hellsten Weiß. Aus Vereinfachungsgründen gehen wir davon aus, daß alle Bilder 256 Bildpunkte breit und 256 Bildpunkte hoch sind, also insgesamt 65536 Bildpunkte enthalten.

Ein einfaches Beispielprogramm, das Ihnen den Umgang mit solchen Bildern näherbringen soll, finden Sie in der Datei `/opt/cad/Prakt/ACS07/TestData/brighten.c` (im Kapitel 4 als Listing 4.5). Diese Anwendung hellt ein gegebenes Bild auf, indem auf alle Grauwerte der Wert 100 aufaddiert wird. Zur Erprobung kopieren Sie die Datei in eines ihrer Arbeitsverzeichnisse und übersetzen Sie es mit dem Kommando `make brighten`. Ein Beispielbild liegt mit dem Namen `lena256.pgm` im selben Verzeichnis. Durch das Kommando `xv lena256.pgm` können Sie es sich anzeigen lassen. Mit der Anweisung `./brighten lena256.pgm lena256b.pgm` wird in der Datei `lena256b.pgm` eine hellere Version des Bildes erzeugt. Betrachten Sie auch dieses mit `xv` und beurteilen Sie das Ergebnis der Aufhellung.

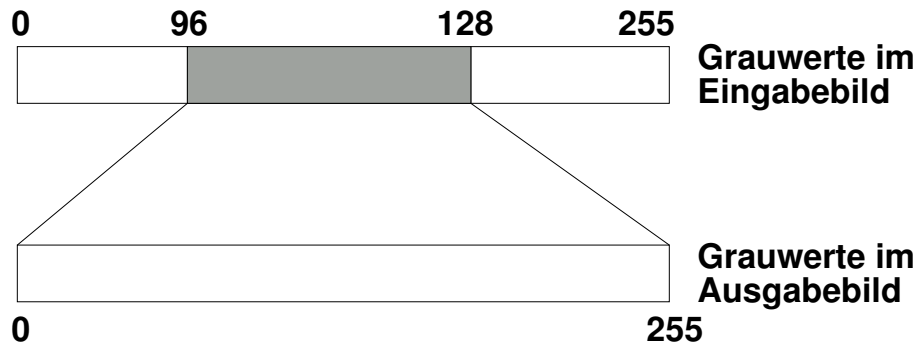


Abbildung 1.1: Idee der Kontraststreckung

Aber zurück zu unserer Aufgabe: Es gibt eine Vielzahl von Algorithmen, die verwaschene Bilder aufbereiten können. Wir schauen uns hier den einfachsten an: Die Aufspreizung des Kontrasts, der folgende Idee zu Grunde liegt.

- Der dunkelste Punkt des Eingabebilds wird immer auf den Grauwert 0 (=schwarz) im Ausgabebild abgebildet (unabhängig von seinem Ursprungswert).
- Der hellste Punkt des Eingabebilds wird immer auf den Grauwert 255 (=weiß) im Ausgabebild abgebildet (auch hier unabhängig von seinem Ursprungswert).
- Alle anderen Grauwerte des Eingabebilds zwischen diesen Minima und Maxima werden in aufsteigender Reihenfolge gleichmäßig zwischen den Werten 0 und 255 in das Ausgabebild abgebildet.

Als Ergebnis erhält man also aus einem Eingabebild, das den vollen Wertebereich 0...255 nicht vollständig ausnutzt, ein Ausgabebild, das den ganzen Wertebereich verwendet und so einen verbesserten Kontrast hat.

Im Beispiel aus Abbildung 1.1 wird also der minimale Grauwert 96 des Eingangsbildes auf den dunkelsten Wert 0 des Ausgangsbildes abgebildet. Der hellste Grauwert 128 des Eingangsbildes wird im Ausgangsbild zu 255 (weiß). Die $128 - 96 = 32$ unterschiedlichen Grauwerte des Eingangsbildes werden nun so aufspreizt, daß Sie das volle Intervall von 0 bis 255 gleichmäßig ausfüllen. Dies wird dadurch erreicht, daß jeder einzelne Grauschritt im Eingangsbild auf $255/32 \approx 8$ Grauschritte im Ausgangsbild abgebildet wird. Also $96 \rightarrow 0$, $97 \rightarrow 8$, etc. Auf diese Weise bekommen wir zwar nicht mehr unterschiedliche Graustufen ins Ausgangsbild, aber sie liegen weiter auseinander und sind somit besser voneinander zu unterscheiden (höherer Kontrast).

In dieser Phase des Praktikums sollen Sie eine Kopie von `brighten.c` nach `contrast.c` so umbauen, daß die oben beschriebene Kontraststreckung realisiert wird. Sie können dabei die Ein-/Ausgabeoperationen unverändert übernehmen. Nur die eigentliche Berechnung müssen Sie anpassen. Verwenden Sie `xv`, um die Ergebnisse Ihrer Transformation auch graphisch betrachten zu können.

Beginnen Sie hier schon die Überlegung, welche Teile Ihres Programmes wie in Hardware ausgelagert werden sollen. Wichtige Punkte sind hier beispielsweise

- Die Bitbreiten der verarbeiteten Daten.
- Die Hardware-Implementierung verschiedener Operatoren. So kann eine Multiplikation mit einer Zweierpotenz in Hardware einfach durch eine Links-Schiebeoperation realisiert werden.
- Ist Parallelverarbeitung möglich?

Abgaben: Das von Ihnen entwickelte C-Programm `contrast.c` sowie eine Beschreibung Ihrer Ergänzungen. Eine Diskussion der von Ihnen geplanten Hardware-Architektur.

Kolloquium: Über die Abgabe von Phase 2.

4. Phase: IP-Blöcke

5.5.-18.5.2008

Für die Hardware-Realisierung Ihres Algorithmus werden Sie einen Dividierer mit variablen Operanden benötigen. Dieser wird nicht automatisch bei der HDL-Synthese erzeugt und ist für dieses Praktikum auch im Entwurf zu aufwendig. Wie in der Praxis üblich, werden Sie also ein schon bestehendes Hardware-Modul, auch *IP-Block* genannt, in Ihren Entwurf einbinden.

In dieser Phase wird Ihnen ein Dividierer-Modul in der von Ihnen gewünschten Größe als Netzliste zur Verfügung gestellt werden. Sie sollen es so in einen Verilog-Testrahmen einbinden, daß zwei variable Operandenregister dividiert werden und Quotient und Rest in Ergebnisregistern abgelegt werden.

Der Dividierer hat unabhängig von den Breiten oder den Datentypen (vorzeichenbehaftet oder -los) seiner Operanden folgende Schnittstelle:

dividend : N -Bit Eingang für den Dividenden.

divisor : M -Bit Eingang für den Divisor.

quot : N -Bit Ausgang für den Quotienten.

remd : M -Bit Ausgang für den Rest der Division.

clk : Takteingang.

ce : Bei einer '1' an diesem Eingang wird der Takt aktiviert, der Dividierer arbeitet.

rfd : Unbenutzt.

Die Schaltung ist derart gepipelined, daß pro Takt ein neuer Satz Operanden an den Eingängen akzeptiert wird. Nach einer bestimmten Anzahl von Takten (Latenzzeit), die von den von Ihnen gewählten Parametern (N und M) abhängt, taucht das entsprechende Ergebnis dann an den beiden Ausgängen auf. Der Dividierer arbeitet immer: Einmal pro Takt werden die an den Eingängen anliegenden Werte eingelesen, und nach der Latenzzeit wechseln die beiden Ausgänge auf die Ergebnisse der Berechnung. Es ist also wichtig, die Ausgänge zum *richtigen* Zeitpunkt auszuwerten.

Beispiel: Bei einem Dividierer, der eine vorzeichenlose 16b Zahl durch eine ebenfalls vorzeichenlose 8b Zahl dividiert, beträgt die Latenz beispielsweise 20 Takte. Das heißt, daß nach Anlegen von Eingangswerten zur Taktflanke 0 das Ergebnis nach der 20. Taktflanke an den Ausgängen zur Verfügung steht und mit der 21. Taktflanke in Register eingelesen werden kann. In Abbildung 1.2 sind die entsprechenden Signalverläufe dargestellt. Hier werden hintereinander (aber pipeline-parallel) die Divisionen 1234/100, 1234/50 und 1234/25 ausgeführt (alle Werte dezimal). In Slave-Mode-Zugriffen wird zunächst der Dividend (1234) und dann die Divisoren (100,50,25) von der Software auf die RC übertragen. Nach dem Übernehmen jedes Divisors können also jeweils 20 Takte später die Werte für die Quotienten (12,24,49) und die Reste (34,34,9) aus den Ausgängen **quot** und **remd** in Ihre entsprechenden Register **r quot** und **r.remd** übernommen werden.

Gehen Sie zur Bearbeitung dieser Phase wie folgt vor:

1. Legen Sie ein neues Slave-Mode-Projekt für diese Phase an.
2. Lassen Sie sich von Ihrem Hiwi den gewünschten Dividierer mittels des Werkzeugs CoreGen erzeugen. Dabei werden Sie zwei Dateien erhalten: Kopieren Sie die **.edn** Datei in das Unterverzeichnis **simple** des Projektverzeichnisses. Kopieren Sie die **.v** Datei direkt in das Projektverzeichnis. Letztere enthält die Moduldeklaration der Dividierierzelle, hier können Sie die Schnittstelle im Detail sehen.
3. Öffnen Sie **user.tcl** mit einem Texteditor und entkommentieren Sie die Platzhalter **add_file** Zeile für Ihren Dividierer (führendes # löschen). Ändern Sie den Namen der Verilog-Datei auf den von Ihnen tatsächlich verwendeten Dateinamen und speichern Sie **user.tcl** ab.
4. Instanzieren Sie Ihre Dividierierzelle in **user.v**.

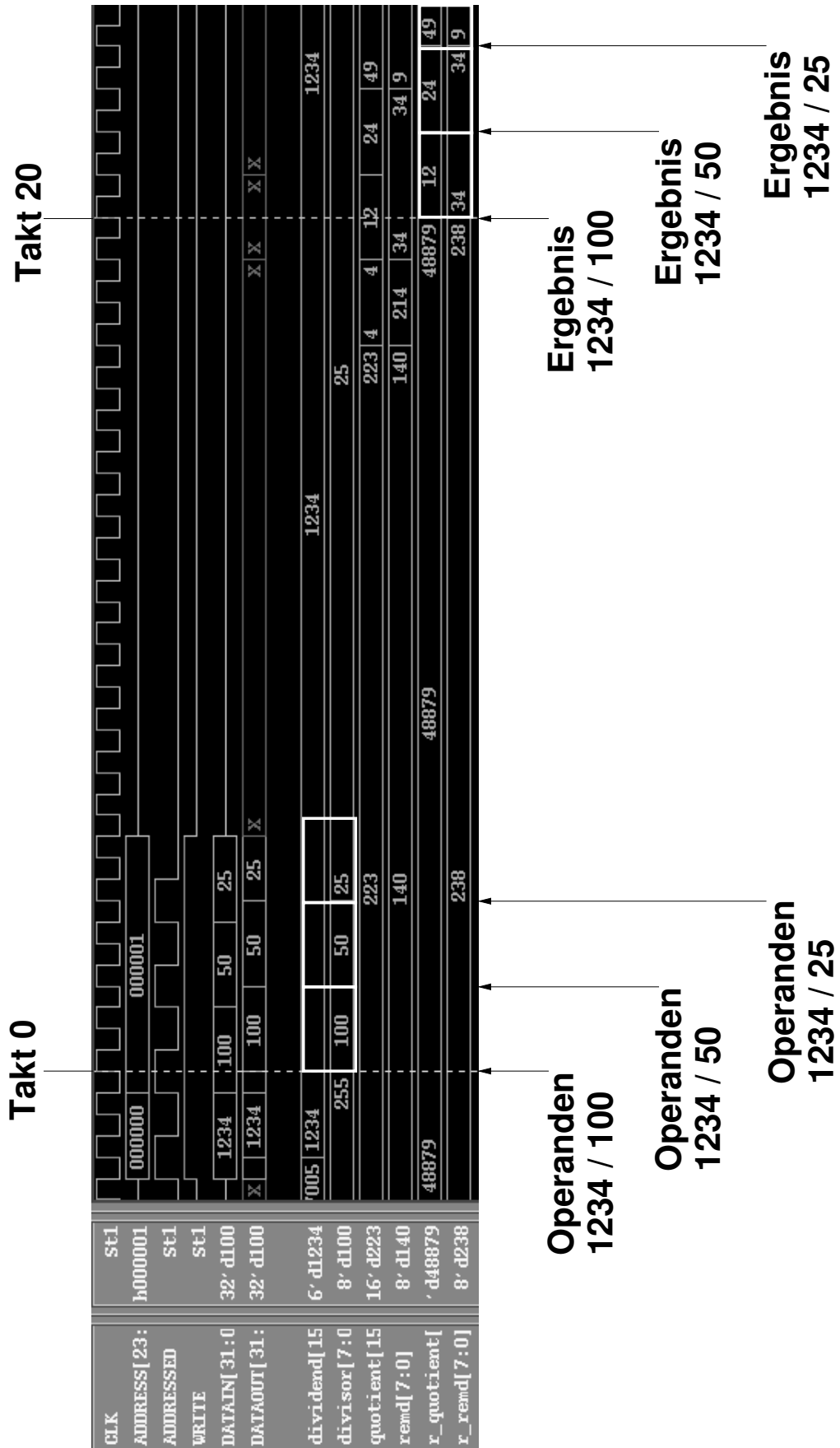


Abbildung 1.2: Pipelined Division von 1234/100, 1234/25 und 1234/25

5. Erweitern Sie `user.v` um die Realisierung der beiden schreibbaren Register für Dividend und Divisor und die beiden lesbaren Register für Quotient und Rest. Dabei müssen Sie die Adressdekodierung ergänzen. **Hinweis:** Der Dividierer-Ausgang *muss* an ein eigenes Register angeschlossen werden, er darf *nicht* über den Multiplexer direkt an DATAOUT angeschlossen werden.
6. Fügen Sie nun die Steuerung hinzu. Diese muß kontrollieren, *wann* genau die Ergebnisregister ihre Werte von den Dividiererausgängen übernehmen. Hier müssen Sie die Latenz beachten: Latchen Sie die Ausgangswerte zu früh, ist die Berechnung noch nicht abgeschlossen. Sind sie zu spät, überschreiben die durch die Pipeline nachrückenden Werte das gewünschte Ergebnis.
7. Die Software-Schnittstelle muss so ausgelegt sein, daß pro Operand je ein Schreibzugriff zur Übertragung der Daten von der CPU an die RC benutzt wird. Je ein einzelner Lesezugriff holt dann die Ergebnisse (Quotient und Rest) von der RC wieder ab. Wahrscheinlich müssen Sie bei dieser Betriebsart zwischen dem Schreiben und Lesen in Software etwas Zeit vergehen lassen (die Rechenzeit des Dividierers). Hierzu reicht beispielsweise ein einfaches `printf("Waiting ... \n");` aus.
8. Simulieren Sie Ihr Verilog Modell auf RT-Ebene. Testen Sie dabei unbedingt auch, ob das Pipelining funktioniert.
9. Schreiben Sie einen kleinen Software-Testrahmen in `main.c`, in dem zwei Zahlen im Slave-Mode zur Division an die RC übertragen werden. Quotient und Rest sollen dann aus der Hardware ausgelesen und dem Benutzer angezeigt werden.
10. Implementieren Sie die ganze Anwendung mit `make linux`. Testen Sie Ihr Programm durch Ausführen von `./main`.

Abgaben: Das HDL-Modell und C-Programm mit Beschreibung und Simulationsergebnissen (RTL und Post-Layout) als kommentierte Waveforms.

Kolloquium: Über die Abgabe von Phase 3.

5. Phase: Slave-Mode Version

19.5.-1.6.2008

Nun realisieren Sie Ihre in Phase 3 geplante Hardware-Architektur unter Verwendung des in der vorigen Phase erprobten Dividierer IP-Blocks. Dabei soll die Hardware in dieser Phase im Slave-Mode arbeiten: Die CPU überträgt die zu verarbeitenden Daten an die RC, diese führt die Berechnung aus und die CPU holt das Ergebnis ab. Sie müssen dazu das C-Programm auch entsprechend anpassen. Simulieren und erproben Sie Ihren Entwurf auf dem ML310. Führen Sie auch wie in Phase 2 Zeitmessungen an Ihrem Design durch (Effizienz der Kommunikation und Ausführungszeit). Achten Sie schon hier auf eine möglichst gute Parallelisierung der Anwendung!

Tipp: Falls Sie mehrere Schleifen in eine Slave-Mode RC auslagern wollen, packen Sie alle Funktionen in *einen* Hardware-Block und wählen Sie mittels eines von der Software beschreibbaren Registers aus, welche Operation aktuell ausgeführt wird.

Hinweis: Wir empfehlen Ihnen *dringend*, sich an die hier vorgeschlagene Lösung mittels des Dividierers zu halten. Abweichungen dürfen nur nach Rücksprache und Genehmigung durch den betreuenden Assistenten erfolgen. Analoges gilt für Phase 6!

Abgaben: Das HDL-Modell und C-Programm mit Beschreibung, kommentierte Simulationsergebnisse als Tabelle (`$monitor()`), Ergebnisse der Zeitmessungen.

Kolloquium: Über die Abgabe von Phase 4.

6. Phase: Master-Mode Version

2.6.-29.6.2008

Stellen Sie nun Ihre Lösung (Hard- und Software) wie im Skript gezeigt auf den Master-Mode Betrieb um. Die RC soll nun also lediglich Startparameter von der CPU entgegennehmen und ansonsten die gesamte Verarbeitung selbständig durchführen. Erst am Ende wird der CPU der erfolgreiche Abschluß durch einen Interrupt signalisiert. Die CPU soll dann das Ergebnisbild in die entsprechende Ausgabedatei schreiben. Wie üblich nehmen Sie auch hier die Effizienz- und Zeitmessungen vor.

Entscheidend für den Aufbau dieser Schaltung ist die Einbindung Ihrer Berechnung in den Datenstrom, insbesondere in die Flußkontrolle (siehe Abbildung 3.12). Nachdem Sie ein neues Master-Mode-Projekt mit `mkmaster` angelegt haben, finden Sie in `user.v` an der Instanz `fc` des `flowcontrol` Moduls eine mit dem Kommentar `/** Ändern */` markierte Stelle. Hier ist der Ausgangs-Port des Lese-Stroms als Eingangs-Port für den Schreib-Strom angeschlossen. Um Ihre Berechnung einzubinden, müssen Sie diese Verbindung auftrennen. Ihre Schaltung muß dann Eingangsdaten aus `STREAM_READ` akzeptieren. Der Ausgang Ihrer Schaltung muß dann an die ehemals von `STREAM_READ` belegte Stelle des `flowcontrol` Moduls angeschlossen werden.

Auch die bisher in `user.v` mit `fc` verbundenen Flußkontrollsignale müssen in analoger Form aufgetrennt und Ihre eigene Schaltung eingefügt werden. Das zu verwendende Protokoll sieht wie folgt aus. Zum besseren Verständnis sei hier an die Signalverläufe einer Master-Mode-Anwendung erinnert, die Sie ja in Phase 1 studiert haben.

STREAM_ENABLE : Dieser Eingang dient zur Steuerung des Datenstromes. Beim Lesestrom bedeutet eine '1' auf diesem Port, daß Ihre Schaltung mit der *übernächsten* positiven Taktflanke ein Eingangsdatum von `STREAM_READ` in ein Register übernehmen möchte. Beim Ausgangsstrom bedeutet die '1', daß Ihre Schaltung gültige Daten schreiben möchte, die mit der nächsten positiven Taktflanke in den Ausgangsstrom übernommen werden. Eine '0' zeigt entsprechend an, daß zur übernächsten bzw. nächsten positiven Taktflanke keine neuen Daten gelesen bzw. geschrieben werden sollen.

STREAM_STALL : Eine '1' auf diesem Ausgang zeigt an, daß die Benutzerschaltung zwar Daten lesen bzw. schreiben möchte (`STREAM_ENABLE=1`), aber der Strom leider unterbrochen ist. Beim Lesestrom bedeutet dies, daß Daten nur zu solchen positiven Taktflanken von Ihrer Schaltung übernommen werden dürfen, wenn zur *vorherigen* positiven Taktflanke `STREAM_STALL=0` war. Im anderen Fall muß Ihre Schaltung warten. Beim Schreibstrom wird nur bei einer positiven Taktflanke das Datum tatsächlich übernommen, wenn zur selben Taktflanke `STREAM_STALL=0` ist. Wenn das Signal '1' ist, muß Ihre Schaltung das Ausgangsdatum solange stabil an den Schreibstrom anlegen, bis die Übernahme tatsächlich erfolgt ist. Anderenfalls geht das Datum einfach verloren.

Da Ihre Schaltung wegen des verwendeten Dividierers nun nicht mehr rein kombinatorisch ist, müssen Sie auch die `STREAM_ENABLE` und `STREAM_STALL` Signale sequentiell verarbeiten. Hier einige (aber nicht alle!) Anhaltspunkte

- Sie dürfen den Schreibstrom erst starten, wenn tatsächlich Ergebnisse aus Ihrem Dividierer vorliegen.
- Sie dürfen den Schreibstrom erst anhalten, wenn tatsächlich alle Eingangsdaten in den Dividierer als Ergebnisse am Dividiererausgang vorliegen (Pipelining!) und erfolgreich geschrieben wurden.
- Sie müssen den Dividierer und den Lesestrom anhalten, wenn der Schreibstrom abreißt (`STREAM_STALL=1`).
- Sie müssen den Dividierer und den Schreibstrom anhalten, wenn der Lesestrom abreißt (`STREAM_STALL=1`).

Die Systemsimulation provoziert solche Stromunterbrechungen künstlich. Sie können also das Verhalten Ihrer Anwendung schon zur Simulationszeit untersuchen.

Abgaben: Das HDL-Modell und C-Programm mit Beschreibung, kommentierte Simulationsergebnisse als Waveforms und/oder Tabelle (`$monitor()`), Ergebnisse der Messungen.

An diese Endabgabe werden in der Form weitergehende Anforderungen gestellt. Hier sind keine Ansammlungen von Einzeldateien mehr erwünscht, sondern es wird ein *homogenes* Dokument gefordert, das alle

Angaben enthält. So sollen hier beispielsweise auch die Waveforms als EPS-Dateien direkt in den Textfluß eingebunden sein (Ankreuzen von Encapsulated PostScript (EPSF) beim Drucken in VirSim) Auch reichen hier stichpunktartige Erklärungen nicht mehr aus, die endgültige Lösung soll umfassend (auch anhand von Zeichnungen) beschrieben werden. Dazu gehört auch eine Kommentierung der Waveforms:

- Was soll gezeigt werden?
- Wie wird sich das in den Signalverläufen niederschlagen?
- Wo (Zeitpunkt) findet man diese Verläufe tatsächlich in den Diagrammen?

Bei dieser Abgabe ist mit einem Gesamtumfang von ca. 15-20 Seiten zu rechnen. Das Abgabeformat dafür ist PDF.

In ihrem Arbeitsverzeichnis sollten Sie ein Unterverzeichnis angelegt haben, in dem alle für den Entwurf benötigten Quellen, Testdaten, ggf. Skripte o.ä. abgelegt werden. Bitte stellen Sie durch ein `make clean` sicher, daß hier keine unnötigen Dateien mehr existieren und beschreiben Sie in einer kleinen `README` Datei den Inhalt dieses Verzeichnisses. Seinen Namen teilen Sie bitte in der Abgabe-Mail mit, er wird für unsere abschliessende Datensicherung gebraucht.

Kolloquium: Über die Abgabe von Phase 5.

7. Phase: Nacharbeiten

30.6.-6.7.2008

Die Pflichtaufgaben sollten vor dieser Phase erfolgreich abgearbeitet worden sein. Nach Rücksprache mit dem Betreuer kann aber auch eine Nacharbeit vereinbart werden.

Kolloquium: Über die Abgabe von Phase 6.

2 Werkzeugfluß

Um sich mit der ACS-Umgebung vertraut zu machen, sollten Sie zunächst mit sehr einfachen Anwendungen experimentieren. Dazu stehen zwei Musterprojekte für den Slave- und Master-Mode zur Verfügung.

2.1 Anlegen von neuen Projekten

Zum schnellen Start in die Arbeit können Sie auf bereits lauffähige Musteranwendungen zurückgreifen. Durch ein einzelnes Kommando wird ein Unterverzeichnis angelegt und mit allen nötigen Dateien versehen. Sie müssen dann lediglich Ihre Änderungen an den passenden Stellen einbauen.

Das zu verwendende Kommando unterscheidet sich nach dem Typ der zu erstellenden Anwendung: Für die Slave-Mode Betriebsart verwenden Sie das Kommando `mkslave`, für Master-Mode das Kommando `mkmaster`. In beiden Fällen folgt dem Kommando der von Ihnen gewünschte Name für das anzulegende Projekt.

Beispiel: Mit dem Kommando `mkslave simsel` wird im aktuellen Verzeichnis ein Unterverzeichnis namens `simsel` angelegt. In diesem befinden sich alle für eine Slave-Mode-Anwendung nötigen Dateien. Die Beispielanwendung realisiert ein einzelnes 32b-Register auf der RCU, das durch die CPU geschrieben und wieder ausgelesen werden kann. Nach einem Reset des Systems hat das Register den schon bekannten, leicht wiedererkennbaren Wert `0xDEADBEEF`. Andere Teile des RCU-Speicherbereiches, die nicht dieses Register enthalten, liefern beim Lesen den Wert `0xC0FFEE11`. Sie kann durch entsprechende Ergänzung leicht an die tatsächlichen Erfordernisse Ihres Entwurfs angepasst werden. Dazu bearbeiten Sie lediglich drei im folgenden Abschnitt beschriebene Dateien. Die Master-Mode Beispielanwendung kopiert einen Speicherbereich durch die RCU auf einen anderen, die CPU ist also beim eigentlichen Kopiervorgang nicht involviert.

2.2 Dateistruktur

Beim Anlegen eines Slave-Mode-Projektes erhalten Sie für Ihre ersten Experimente im wesentlichen drei interessante Dateien:

`user.v` ist die Beschreibung der Slave-Mode-RCU in Verilog. Hier erkennt man die typische Slave-Mode-Schnittstelle sowie die eigentliche Anwendung im Rumpf des Moduls. Die Beispielanwendung erlaubt den Zugriff auf bis zu vier unterschiedliche Register (Dekodierung der letzten beiden Bits der Wortadresse ADDRESS, also die Möglichkeiten 00, 01, 10 und 00). Davon ist momentan nur die Teiladresse 00 belegt (hier wird das Register `outreg` an die CPU ausgegeben). In den drei anderen Fällen wird die gut erkennbare Konstante `0xC0FFEE11` ausgegeben. Das Register `outreg` wird im `always`-Block auf `0xDEADBEEF` zurückgesetzt. Bei Schreibzugriffen auf die Register-Adresse 00 übernimmt es den von der CPU an den Eingabe-Bus `DATAIN` angelegten Wert. Die Master-Mode-Version enthält zusätzlich noch die Schnittstelle für die MARC-Streams (siehe Abschnitt 3.3.3).

`main.c` ist der Software-Teil der Anwendung, der auf der PowerPC-CPU des ML310 ACS ausgeführt wird. Nach den üblichen Vorbereitungen (Initialisierung, bestimmen der Basisadresse des RCU-Bereiches) wird der 32b-Wert an der RCU-Wortadresse 0 (`rcu` ist als Zeiger auf `unsigned long`, also auf 32b Werte deklariert) gelesen und ausgegeben. Dies führt also zu einem Zugriff auf das RCU-Register `outreg`. Danach wird ein Schreibzugriff auf die gleiche Adresse vorgenommen, gefolgt von einem Auslesen des neuen Wertes.

stimulus.v Zum Testen der RCU in der Simulation müssen die Zugriffe der CPU auf Adressen im RCU-Speicherbereich nachgeahmt werden. Dazu können in dieser Datei vordefinierte Verilog-Funktionen aufgerufen werden. Solche Funktionen stehen für das Starten und Herunterfahren der Simulationsumgebung ebenso bereit, wie für das Nachahmen von Lese- (via `Read32`) und Schreibzugriffen (via `Write32`). Beide Funktionen akzeptieren als ersten Parameter die *CPU*-Adresse für den Zugriff. Zum Test der RCU muß hier als Basisadresse des RCU-Adressbereiches die Konstante `SLAVE_BASE` angegeben werden. Dazu relativ kann nun die Adresse *innerhalb* des *RCU*-Adressraums angegeben werden. Die Funktion `Write32` erwartet als zweiten Parameter den zu schreibenden 32b-Wert, die Funktion `Read32` liest einen 32b-Wert von der RCU in ein als zweiten Parameter übergebenes Register (32b breit, im Beispiel heisst das Register `data`). Die Systemfunktion `$display` arbeitet ähnlich wie `printf` in C, indem sie einen Wert entsprechend der Formatangabe (hier: `%h` steht für hexadezimale Darstellung) als Text auf der Simulatorkonsole ausgibt.

Zur Simulation von Master-Mode-Anwendungen können in **stimulus.v** zwei weitere Kommandos zum Umgang mit dem simulierten Speicher verwendet werden. Mit dem Kommando

```
ReadMemFile("infile.mem")
```

wird der Inhalt der Datei `infile.mem` zur Simulationszeit in den Speicher geschrieben. Die Eingabedatei (hier `infile.mem`) hat folgendes Format: Die Kopfzeile enthält die Adresse des ersten Bytes und die Anzahl der folgenden 32b Worte. Nun folgen die vorher angegebene Anzahl von 32b Worten, eines pro Zeile. Dann ist die Datei zu Ende, oder es folgt eine weitere Kopfzeile. Alle Zahlen werden hexadezimal dargestellt. Eine Beispieldatei `infile.mem` könnte wie folgt aussehen:

```
1000 3
12345678
87654321
deadbeef
2000 2
10101010
01010101
```

Nach `ReadMemFile("infile.mem")` würde auf Adresse 4096 (dezimal) das Wort `0x12345678` beginnen, auf Adresse 4100 das Wort `0x87654321`, auf Adresse 4104 das Wort `0xDEADBEEF`. Der zweite Block weist Adresse 8192 das Wort `0x10101010` und Adresse 8196 das Wort `0x01010101` zu. Man beachte hier, daß alle Adressen als Byte-Adressen angegeben sind und ein 32b Wort vier Bytes an Speicherplatz benötigt.

Um einen Speicherauszug des simulierten Speichers in eine Datei zu schreiben kann das Kommando `WriteMemFile("outfile.mem", 32'h1000, 3)` verwendet werden. Mit den hier gezeigten Parametern werden drei 32b Worte beginnend bei Byte-Adresse 4096 (dezimal) in die Datei `outfile.mem` geschrieben. An das vorige Beispiel anschließend hätte diese dann den folgenden Inhalt:

```
1000 3
12345678
87654321
deadbeef
```

Um bestehende Dateien nach und von diesem Format zu wandeln stehen zwei Hilfsprogramme bereit. Bei Eingabe von `bin2mem <lena256.pgm >lena256.mem` auf Unix Kommandoebene wird die Graustufenbilddatei `lena256.pgm` als hexadezimaler Speicherauszug in die Datei `lena256.mem` geschrieben. Wichtig: Die Kopfzeile (Startadresse und Anzahl von 32b Worten) fehlt noch und muß *manuell* mit einem Texteditor in der Datei `lena256.mem` nachgetragen werden. Der umgekehrte Schritt ist mit `mem2bin <lena256contrast.mem >lena256contrast.pgm` möglich. Hier sind keine manuellen Schritte mehr nötig. `lena256contrast.pgm` enthält genau die Daten aus `lena256contrast.mem`, die Kopfzeile wurde automatisch entfernt. `mem2bin` ist auf die Bearbeitung von Eingabedateien beschränkt, die nur einen Speicherbereich enthalten.

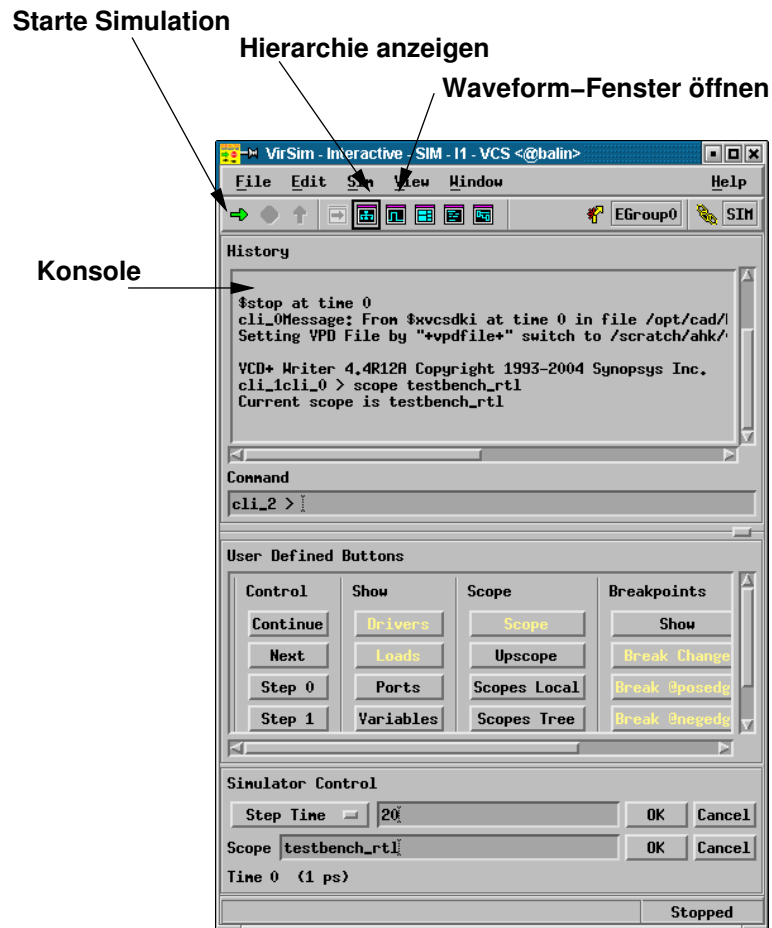


Abbildung 2.1: Graphische Simulationsumgebung VirSim

2.3 Simulation auf Registertransferebene (RTL)

Nach dem Sichten der Eingabedateien können Sie die Funktion der Slave-Mode-RCU im Verilog-Simulator erproben. Dabei werden die in der Stimulus-Datei beschriebenen Zugriffe auf die RCU ausgeführt und währenddessen verschiedene Signale der RCU aufgezeichnet. Diese Signale können bei Ende der Simulation graphisch in Form von Signalverlaufdiagrammen (*waveforms*) dargestellt werden. Gegenüber den ebenfalls möglichen Textausgaben auf der Simulatorkonsole haben die Waveforms der Vorteil, dass hier leichter zeitliche Zusammenhänge zwischen parallelen Signalverläufen erkannt werden können.

Das Unix-Kommando

```
make rtlsim
```

übersetzt die Verilog-Beschreibungen für die Simulation. Sollten hierbei keine Fehler aufgetreten sein (diese würden den Vorgang abbrechen und müssten erst in den Quelldateien behoben werden), wird das Visualisierungswerkzeug für die Signaldiagramme gestartet (siehe Abbildung 2.1).

Man könnte die Simulation schon jetzt starten, würde dann aber nur die Textausgaben durch die `$display`-Aufrufe im Verilog auf der Simulatorkonsole sehen. In der Regel ist man aber eher an Waveforms interessiert. Dazu muß dem Simulator *vor* der Simulation mitgeteilt werden, welche Signale tatsächlich aufgezeichnet werden sollen. Der Einfachheit halber werden wir den Simulator anweisen, *alle* Signale unserer im Verilog-Modul user definierten Slave-Mode RCU anzuzeigen. Dies schließt sowohl die Schnittstelle zum Restsystem als auch eventuelle interne Register ein. Um diese Auswahl vorzunehmen, lassen wir uns zunächst die komplette Schaltungshierarchie (Klick auf entsprechendes Icon aus Abbildung 2.1) sowie ein

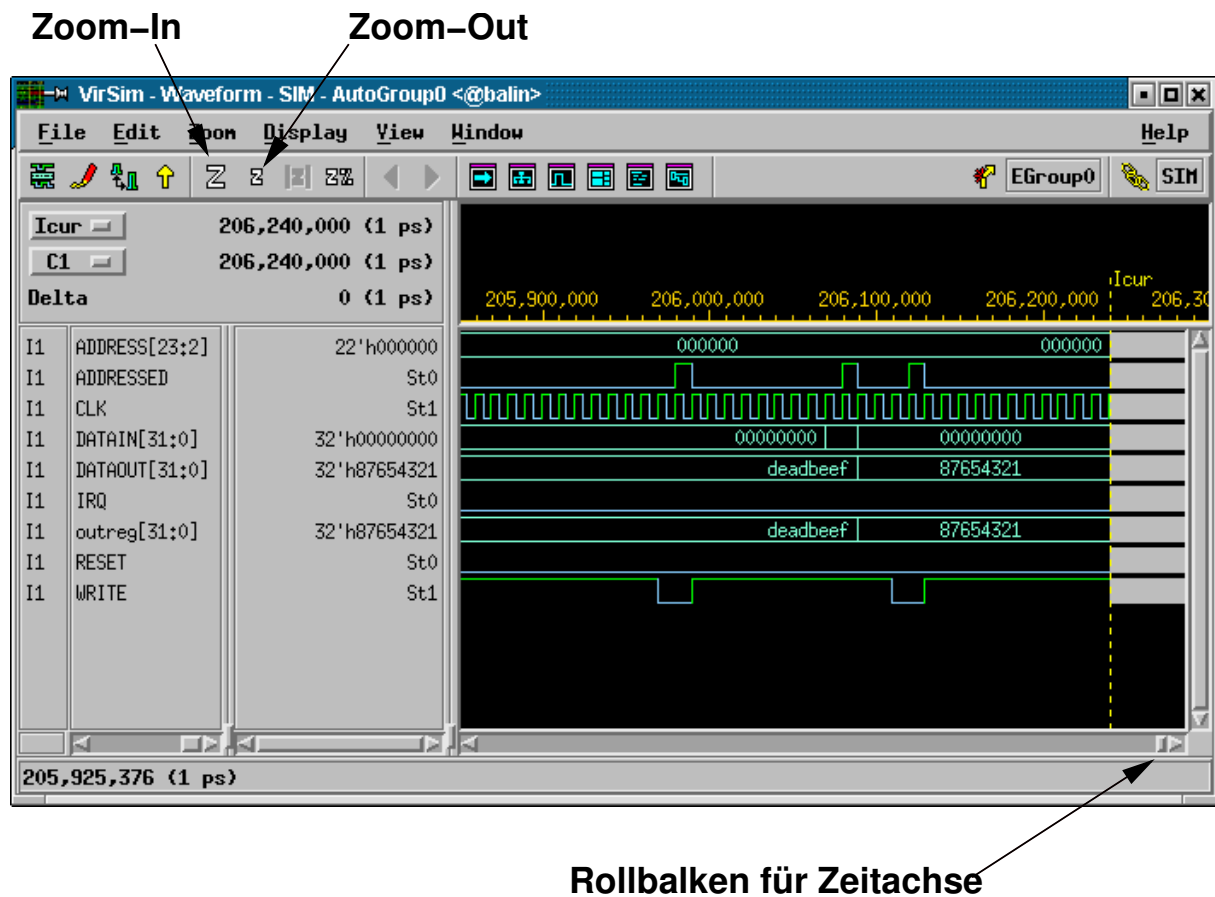


Abbildung 2.3: Waveform-Darstellung der Simulationsergebnisse

beim Lesen des Initialzustands (0xDEADBEEF) von `outreg` ausgegeben. Darauf folgt dann die Ausgabe des neuen Wertes (0x87654321) von `outreg` nach dem Schreibzugriff.

Scrollen Sie nun *horizontal* mit dem Rollbalken für die Zeitachse ans rechte Ende des Waveform-Fensters und verkleinern Sie den Maßstab solange durch Klicken auf das kleine Zoom-Out-Icon (kleines z-Symbol) der Menüleiste, bis Sie für das Taktsignal CLK eine ganze Reihen von Taktflanken erkennen können. Machen Sie nun die drei Zugriffe (Lesen, Schreiben, Lesen) ausfindig. Achten Sie jeweils auf die Korrelation zwischen den Signalen CLK, ADDRESSED und WRITE: Wenn bei einer *steigenden* Flanke von CLK das Signal ADDRESSED auf 1 liegt, liegt ein Zugriff der CPU (hier vertreten durch die Stimulus-Datei) auf die RCU vor. Wenn zu dieser steigenden CLK-Flanke nun WRITE den Werte 1 hat, liegt ein Schreibzugriff vor, sonst ein Lesezugriff. Hinweis: Die Formulierung "zu dieser steigenden Flanke" bedeutet, dass das betreffende Signal unmittelbar *vor* der steigenden Flanke den entsprechenden Wert hat (Erinnerung TGD12: Die Zeit *vor* der steigenden Flanke ist die Setup-Zeit, die nach der Flanke die Hold-Zeit. Die Hold-Zeit kann bei uns mit 0 angenommen werden, relevant ist also nur der Wert eines Signals vor der Flanke).

Um eine solche Waveform-Darstellung aus dem Simulator zu exportieren (z.B. für Einbindung in eigene Dokumentation) wählen Sie aus dem Menü `File` des Waveform-Fensters den Punkt `Print...` aus und füllen den nun angezeigten Druck-Dialog wie in Abbildung 2.4 aus. Nach dem Klicken von `Apply` wird die Ausgabe dann in die angegebene Datei geschrieben. Die Grafik in dieser ist allerdings häufig ungeschickt auf der Druckseite angeordnet. Durch das Unix-Kommando

```
ps2epsi virsim.ps
```

(geeignet angepasst für den von Ihnen in den Druckdialog eingetragenen Ausgabedateinamen) wird eine neue Datei mit der Endung `.epsi`, hier also `virsim.epsi` erzeugt. Diese kann dann leicht, beispielsweise in

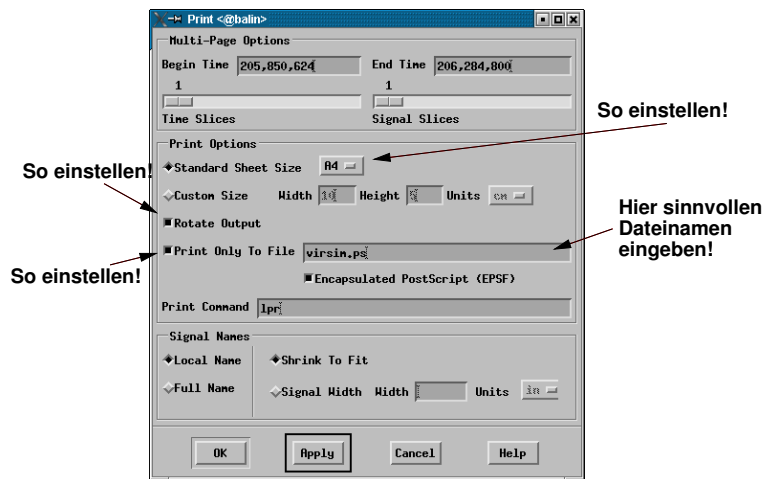


Abbildung 2.4: Export von Waveforms in EPS-Datei

L^AT_EX, eingebunden werden (Abbildung 2.5).

Beenden können Sie den Simulator durch den Menüpunkt File, Exit.

Falls Sie eigene Experimente mit dem bisherigen Werkzeugfluß machen möchten: Führen Sie beispielsweise die Lese- und Schreibzugriffe nicht Wortadresse 0 der RCU (wie in der ursprünglichen Stimulus-Datei), sondern auf Wortadresse 1 aus. Überprüfen Sie das von Ihnen jetzt erwartete Verhalten mit einer geeigneten RTL-Simulation.

2.4 Erzeugen der RCU-Hardware

Nachdem die Simulation nun den prinzipiellen Nachweis der Funktion der RCU geliefert hat, kann die Verilog-Beschreibung jetzt auf die echte FPGA-Hardware des ML310 ACS abgebildet werden. Dies geschieht durch das Unix-Kommando

```
make bits
```

Dieser Schritt ist sehr rechenaufwendig und schließt beispielsweise neben einer Kompilierung der Verilog-Quelltexte in digitale Schaltungen (sogenannte Logiksynthese) auch Platzierungs- und Verdrahtungsschritte für mehr als 7300 Logikblöcken und mehr als 47000 Zwei-Terminal-Nets ein. Die Laufzeit dieses Vorgangs liegt je nach Variation der Schaltung bei ca. 35-40 Minuten. An dieser Stelle wird offensichtlich, dass der Test einer in Entwicklung befindlichen Schaltung weitgehend durch *Simulation* und nicht durch einfaches Ausprobieren auf der ACS-Hardware geschehen sollte!

2.5 Hardware-Test der Slave-Mode-RCU

Bei Slave-Mode-RCUs kann die eigentliche RCU nach erfolgreicher Simulation auch gezielt in echter Hardware erprobt werden, selbst wenn der Software-Teil der Anbindung noch nicht bereitsteht. Dieser Vorgang kann durch das Unix-Kommando

```
make download
```

gestartet werden. Hier sollten am Ende Meldungen ähnlich zu

```
INFO:IMPACT:579 - '2': Completed downloading bit file to device.
INFO:IMPACT:580 - '2':Checking done pin ....done.
'2': Programmed successfully.
```

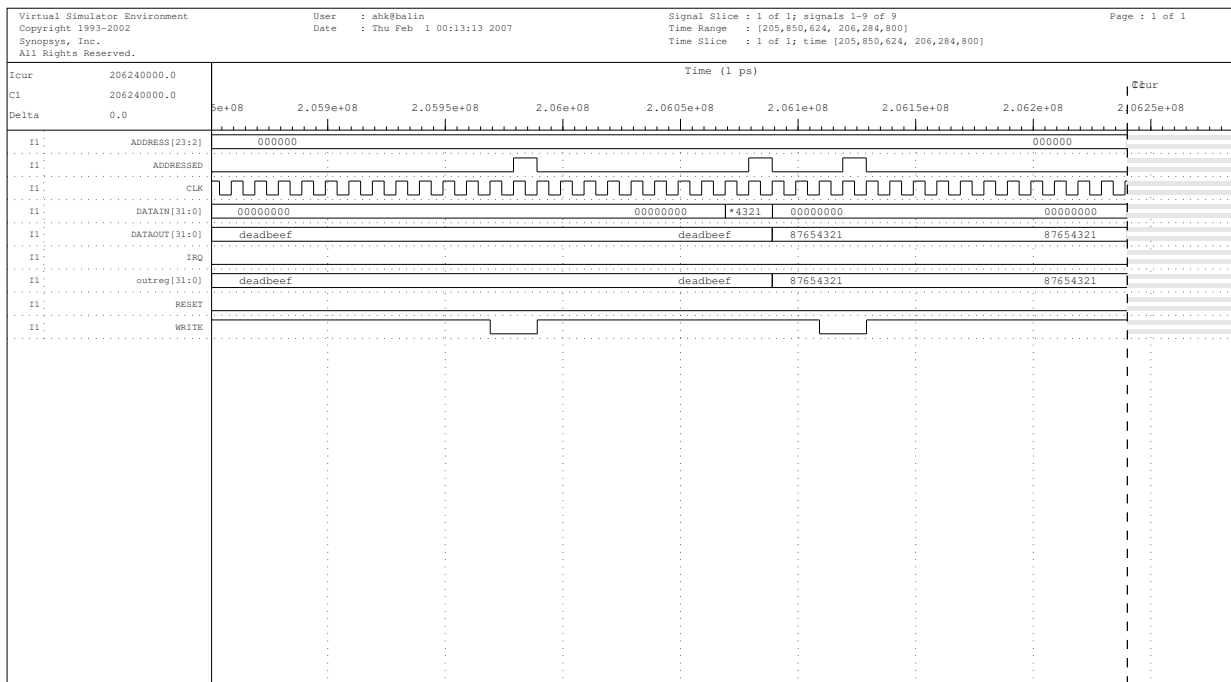


Abbildung 2.5: Beispiel für in L^AT_EX eingebundene Waveforms

ausgegeben werden. Der Bitstrom, der das FPGA mit dem Gesamtsystem einschliesslich Ihrer RCU definiert, ist nun korrekt in den FPGA-Baustein übertragen worden und Sie können jetzt durch interaktive Kommandos Zugriffe auf die Hardware durchführen. Dazu geben Sie ein weiteres Unix-Kommando ein:

```
xmd
```

Nun wird als Prompt `xmd%` angezeigt. Hier geben Sie nun folgendes Kommando ein, um die Kommunikation mit der Hardware aufzubauen:

```
connect ppc hw
```

Einige Zwischenmeldungen sollten hier mit den Zeilen

```
Connected to "ppc" target. id = 0
Starting GDB server for "ppc" target (id = 0) at TCP port no 1234
```

enden und der Prompt wieder erscheinen. Lösen Sie jetzt einen systemweiten Reset aus, um die gesamte ACS-Hardware in einen definierten Anfangszustand zu bringen. Hierzu geben Sie das Kommando:

```
rst
```

Die Basisadresse der RCU auf dieser XMD-Sicht ist `0x10000000`. Diese Adresse ist nun eine *Byte-Adresse*, keine Wortadresse (wie noch im Verilog) mehr. Das heisst also, dass Sie um z.B. auf das zweite 32b Register der RCU zuzugreifen, die Adresse `0x10000004` ansprechen würden. Dies würde zu einem Wert des ADDRESS-Eingangs der RCU von `0x000001` führen (dort RCU-relative Adresse, Adressbasis `0x10000000` im CPU-Speicherbereich entfällt, Zugriff auf Byte 4 (CPU) entspricht Zugriff auf Wort 1 (RCU), da ein Wort vier Bytes hat).

Zum Auslesen des Registers auf Wortadresse 0 (dort liegt ja `outreg`) geben Sie jetzt den Befehl:

```
mrd 0x10000000
```

Hier erfolgt hoffentlich die erwartete Ausgabe (0xDEADBEEF), die angibt, dass das Register auf RCU-Adresse 0 in der Tat korrekt initialisiert worden ist.

Jetzt schreiben wir den Wert 0x87654321 in das Register mit dem Kommando:

```
mwr 0x10000000 0x87654321
```

Zur Überprüfung lesen wir nun den Wert erneut aus:

```
mrd 0x10000000
```

Wenn unsere RCU auch in Hardware richtig arbeitet, wird nun der neu geschriebene Wert (0x87654321) auch wieder ausgelesen. Um das Verhalten zu testen, dass von den Wortadressen 1, 2 und 3 immer der in `user.v` vorgegebene Wert geliefert wird, machen wir die Probe auf's Exempel und lesen von den entsprechenden Byte-Adressen im CPU-Adressraum:

```
mrd 0x10000004  
mrd 0x10000008  
mrd 0x1000000C
```

In allen drei Fällen wird 0xC0FFEE11 geliefert, unser Marker für ein noch nicht benutztes RCU-Register. Da wir nur auf die letzten beiden Bits der Wortadresse achten, wiederholt sich diese Registeranordnung alle vier Worte. Auf der RCU-Wortadresse 4 findet sich also wieder `outreg` (die beiden untersten Bits der Wortadresse sind 00, genau hier liegt `outreg`). Das Kommando

```
mrd 0x10000010
```

beweist diese These, es wird wieder der derzeit aktuelle Wert von `outreg` (0x87654321) geliefert. Beenden Sie die Sitzung nun durch das Kommando

```
exit
```

und kehren wieder zur Unix Kommandozeile zurück.

2.6 Hardware-Test des Gesamtsystems

Nachdem jetzt Ihre RCU selbst bereits korrekt in Hardware läuft, ist es nun an der Zeit, auch das Gesamtsystem zu überprüfen. Durch das Kommando

```
make linux
```

wird auch der Software-Teil der Anwendung (aus der Datei `main.c`) für den PowerPC 405 Prozessor auf dem FPGA des ML310 ACS übersetzt, sowie ein bootfähiges Linux zusammengestellt. Dieses wird dann auf die Hardware übertragen und gestartet. Nach dem Durchlauf vieler Zwischenmeldungen (bei denen gelegentliche Pausen von 5-10s **normal** sind) kommt schliesslich ein vertrauter Anmeldeprompt, bei dem Sie sich mit Ihrem normalen Login-Namen (`gruppe01` oder ähnlich) und Passwort anmelden können. Nach erfolgreicher Anmeldung hier arbeiten Sie nun nicht mehr auf Ihrem Arbeitsplatzrechner, sondern auf dem ML310 ACS!

Dies äußert sich zum einen durch den Inhalt Ihres HOME-Bereichs. Auf dem ACS steht *nicht* mehr Ihr normaler HOME-Bereich aus dem Netz des FG ESA zur Verfügung. Sie befinden sich nun in einem dedizierten Netz, nur bestehend aus Ihrem Arbeitsplatzrechner und dem nebenstehenden ACS. Dabei wird das Verzeichnis

```
/scratch/gruppe01
```

des Arbeitsplatzrechners (wobei `gruppe01` nur ein Beispiel für Ihren Login-Namen ist) als HOME-Bereich für diesen Login-Namen dem ACS zugänglich gemacht. Wenn Sie also eine Datei `foo.txt` vom Arbeitsplatzrechner auf das ACS übertragen wollen, geben Sie auf dem Arbeitsplatzrechner das Kommando

```
cp foo.txt /scratch/gruppe01
```

ein. Die Datei `foo.txt` taucht damit in Ihrem HOME-Bereich auf dem ACS auf. In umgekehrter Richtung, also vom ACS zum Arbeitsplatzrechner lautet das Kommando für eine Datei `bar.txt` auf dem Arbeitsplatzrechner analog

```
cp /scratch/gruppe01/bar.txt .
```

Die Datei `bar.txt` taucht nun im aktuellen Verzeichnis auf dem Arbeitsplatzrechner auf. Da der `/scratch`-Bereich nicht Bestandteil der Datensicherung ist und auch bei Platzknappheit jederzeit gelöscht werden kann (siehe Leitfaden zum Rechnernetz des FG ESA), sollten Sie alle nicht automatisch wiederherstellbaren Dateien regelmäßig in Ihren HOME-Bereich auf dem Arbeitsplatzrechner kopieren.

Aber schauen wir uns an, was durch das Kommando `make linux` noch geschehen ist: Auf unserem ACS-HOME-Bereich liegt jetzt auch eine Datei `main`. Es handelt sich dabei um das auf dem Arbeitsplatzrechner übersetzte Software-Programm unserer ACS-Anwendung, das automatisch bereits in den ACS-HOME-Bereich kopiert wurde. Eventuelle Dateien mit Namen beginnend mit `vcdplus-`... können Sie ignorieren. Es handelt sich dabei um Zwischendateien einer früheren Simulation auf dem Arbeitsplatzrechner, die dieser aus Platz- und Rechenzeitgründen auch auf Ihrem `/scratch`-Unterverzeichnis abgelegt hat. Sie sind für die Arbeit auf dem ACS aber bedeutungslos.

Nicht bedeutungslos sind allerdings eventuelle Eingabedateien, die Ihre spezielle ACS-Anwendung möglicherweise noch benötigt. Da diese Dateien zwischen unterschiedlichen Applikationen variieren können, kann sie der ACS-Werkzeugfluß nicht automatisch vom Arbeitsplatzrechner auf den ACS-HOME-Bereich kopieren. Falls Ihre Anwendung also Eingabebilder etc. benötigt, kopieren Sie diese durch Absetzen geeigneter Kommandos (siehe oben!) selber in den HOME-Bereich auf dem ACS.

Wenn alle benötigten Dateien vorhanden sind (im Fall unserer einfachen Beispielanwendung hier mit dem les- und schreibbaren Register sind keine weiteren erforderlich), kann nun die vollständige ACS-Anwendung, bestehend aus RCU- und CPU-Teil (unserem übersetzten C-Programm) gestartet werden. Dazu wird einfach der Name des übersetzten Programmes, hier also

```
./main
```

als Unix-Kommando gegeben. Das nun ablaufende Programm initialisiert die RCU, stellt die Kommunikation her und führt die Lese- und Schreibzugriffe aus. Die `printf`-Funktionen geben die Registerwerte vor (`0xDEADBEEF`) und nach dem Schreiben des neuen Wertes (`0x87654321`) aus.

Tipp: Falls `make linux` länger als ca. 20s ohne weitere Ausgabe hängen sollte, tippen Sie die Tastenfolge Control-**A** Control-**A**. Dann sollten Sie wieder einen Unix-Prompt auf dem Arbeitsplatzrechner bekommen. Geben Sie dann einfach das Kommando `make linux` erneut. Sollte auch nach dem dritten Versuch der Start des ACS-Linux fehlschlagen, halten Sie am blauen ACS-Gehäuse den Reset-Taster (an der Gehäusefront unter dem größeren Einschaltknopf) ca. 2-3s gedrückt, um einen echten Hardware-Reset des ACS auszulösen. Warten Sie dann, bis sich die LEDs auf der ML310 Leiterplatte beruhigt haben, und probieren Sie `make linux` noch einmal. Wenn nun auch noch drei weitere Versuche fehlschlagen, greifen Sie auf die Rückseite des ACS-Gehäuses und unterbrechen mit dem dort gelegenen Netzschalter die Stromzufuhr für ca. 1 Minute, bevor Sie das ACS wieder einschalten. Warten Sie nun ebenfalls auf stetig leuchtende LEDs (bis sich das System grundinitialisiert hat), und versuchen dann wieder `make linux`. Wenn alle diese Versuche fehlschlagen, kontaktieren Sie bitte Ihren Betreuer für diese Lehrveranstaltung.

Falls bis hierher alles geklappt hat, haben Sie gerade erfolgreich alle Schritte von der Formulierung getrennter RCU- und CPU-Teile bis hin zur Ausführung der vollständigen Anwendung auf einem modernen ACS absolviert. Nun ist es Zeit für Ihre eigenen Entwicklungen! Dazu modifizieren Sie die `main.c`, `user.v` und `stimulus.v`-Dateien der Materialsammlung entsprechend den aktuellen Anforderungen.

3 Adaptive Rechensysteme - Eine praktische Einführung

3.1 Rechnen mit rekonfigurierbarer Hardware

FPGAs können nicht nur als preiswerter ASIC-Ersatz oder für das ASIC-Prototyping verwendet werden. Eine gerade erst im Anfang befindliche Bewegung propagiert die Verwendung von FPGAs und anderen Bauelementen mit ähnlich flexibler Struktur zur Bewältigung von Rechenaufgaben. Auf diese Weise lassen sich gegenüber Standardprozessoren teilweise erhebliche Leistungssteigerungen erzielen.

In diesem Abschnitt werden die Grundlagen des rekonfigurierbaren Rechnens vorgestellt sowie teilweise schon früher behandelte Konzepte (wie FPGAs) in neuem Licht betrachtet.

3.1.1 Anwendungen

Bevor wir uns eingehender mit der Materie beschäftigen, sollen hier als Motivation einige recht erfolgreiche Anwendungen rekonfigurierbarer Hardware zum Lösen verschiedenster Probleme beschrieben werden.

- Ein Algorithmus zum Vergleichen von Gensequenzen lief auf der FPGA-basierten SPLASH Plattform fast 200x schneller als auf Supercomputern (Connection Machine CM-2¹ und Cray-2).
- Der Weltrekord (2001) für die schnellste Entschlüsselung nach dem RSA-Verfahren wird von einem rekonfigurierbaren Rechner vom Typ PAM gehalten (600Kb/s mit 512b langen Schlüsseln).
- Auch im Bereich der DES-Verschlüsselung wird der Rekord 2001 von einer FPGA-Implementierung gehalten (10.7 Gb/s)
- Im Bereich der Signalverarbeitung (Filteralgorithmen etc.) sind rekonfigurierbare Lösungen konventionellen DSPs in der Geschwindigkeit bei einigen Anwendungen um ein bis zwei Größenordnungen überlegen.
- Einige Anwendungen der automatischen Bilderkennung laufen auf einem mit 25 MHz Taktfrequenz betriebenen FPGA mehr als 15x schneller als auf einem mit 450 MHz getakteten Standardprozessor.

Eine Unzahl von weiteren Erfolgen liegen beispielsweise in den Bereichen Arithmetik, Physik, Optimierung, Bild- und Videoverarbeitung, Audio- und Sprachverarbeitung sowie Datennetz-Infrastruktur vor. In allen Fällen werden die betrachteten Probleme deutlich schneller oder ökonomischer auf einer rekonfigurierbaren Plattform gelöst.

Eine weitere, immer wichtiger werdende Größe, ist der Energieverbrauch für eine Berechnung. So stellen moderne mobile Kommunikationssysteme immer höhere Anforderungen an Rechenleistung bei minimalem Stromverbrauch. Nach ersten Untersuchungen können auch in diesem Bereich Architekturen mit einer rekonfigurierbaren Komponente Standardprozessoren (auch in Low-Power-Versionen!) um eine Größenordnung überlegen sein.

3.1.2 Idee

Was unterscheidet nun das Rechnen mit rekonfigurierbarer Hardware vom Rechnen mit Standardprozessoren? Schließlich könnte man doch einfach einen der gängigen Prozessoren auf dem FPGA realisieren.

¹Wie im Film *Jurassic Park* zu sehen ...

Außer einer deutlich langsameren und sehr viel teureren Implementierung der bekannten Prozessorarchitektur hätte man dadurch aber nichts erreicht.

Der wesentliche Unterschied zwischen den beiden Ansätzen besteht darin, dass mit rekonfigurierbarer Hardware eine Berechnung *räumlich* verteilt wird, während sie in Standardprozessoren *zeitlich* verteilt wird. Das folgende Beispiel soll diese recht abstrakte Aussage verdeutlichen.

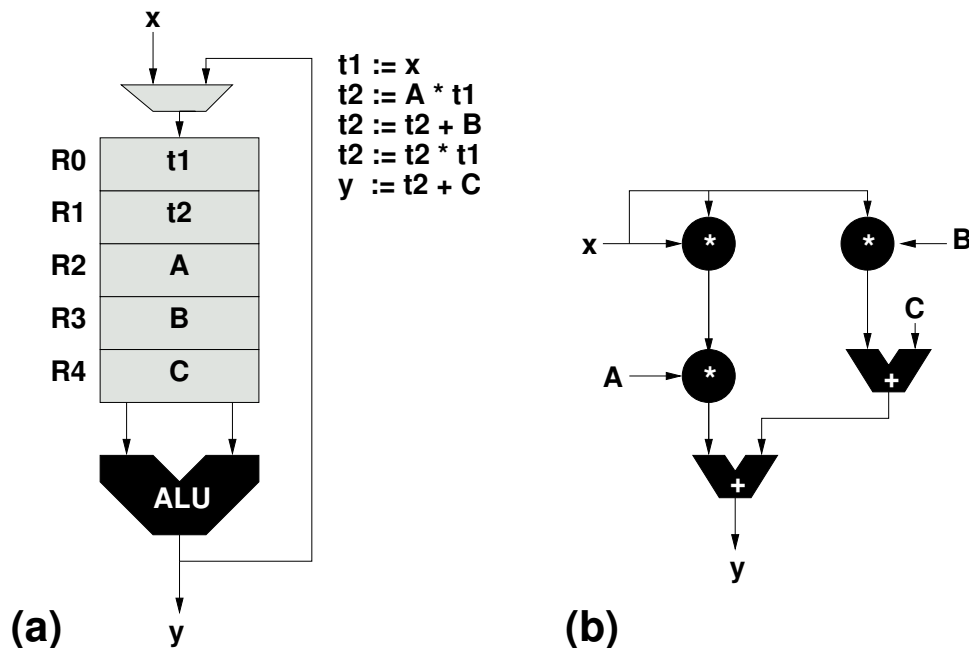


Abbildung 3.1: Zeitliche und räumliche Verteilung von Berechnungen

Nehmen wir an, das Polynom $y = Ax^2 + Bx + C$ soll für verschiedene Werte von x berechnet werden. Bild 3.1.a zeigt die Realisierung auf einem (sehr einfachen) Standardprozessor: Ein Registerfeld R0 ... R4 speichert Zwischenergebnisse und Konstanten. Für die Arithmetik steht eine ALU mit zwei Eingängen zur Verfügung. Die gesamte Berechnung wird durch das nebenstehende Programm kontrolliert: Pro Zeitschritt wird eine Teiloperation auf der immer gleichen Hardware (nämlich der ALU) ausgeführt. Nach fünf Schritten kann das Endergebnis schließlich ausgegeben werden.

In Bild 3.1.b wird die räumlich verteilte Lösung gezeigt, wie sie auf rekonfigurierbarer Hardware verwendet würde. Zur besseren Darstellung sind in der Abbildung alle aktiv rechnenden Hardware-Teile schwarz unterlegt. Hier wird auf das Registerfeld, die flexible ALU und das sequentielle Steuerprogramm verzichtet. Stattdessen werden direkt fünf Hardware-Operatoren in geeigneter Weise verschaltet. Die gesamte Berechnung wird in einem Zeitschritt ausgeführt, dabei finden aber Teiloperationen auf räumlich verschiedenen Hardware-Schaltungen statt.

Diese letztbeschriebene Vorgehensweise ist für effizienten Hardware-Entwurf allgemein üblich. Aber erst die Verwendung rekonfigurierbarer Logikbausteine erlaubt ihren Einsatz auch zur Realisierung von Universalrechnern. Es wäre ja wenig praktikabel, bei Wunsch nach Ausführung einer anderen Berechnung einen anderen ASIC entwerfen und fertigen zu müssen. In rekonfigurierbarer Hardware wird lediglich eine an die neuen Anforderungen angepasste Konfiguration geladen.

In der Realität sind die Grenzen zwischen Standard- und rekonfigurierbaren Prozessoren weniger deutlich. So können moderne superskalare CPUs auch pro Zeitschritt mehrere Operationen auf eigenen Recheneinheiten durchführen. Und auch auf rekonfigurierbaren Architekturen kann es notwendig und sinnvoll sein, eine Teiloperation in mehreren Zeitschritten oder unter Wiederverwendung desselben Hardware-Operators durchzuführen.

3.1.3 Terminologie

Nach diesem ersten Einblick in das rekonfigurierbare Rechnen soll hier die auf diesem Gebiet verwendete Terminologie etwas genauer betrachtet werden.

Rekonfigurierbarkeit (manchmal auch *Adaptionsfähigkeit* genannt) bezeichnet hier die Fähigkeit, die logische Struktur (Recheneinheiten und ihre Verbindungen untereinander) eines Bausteins bzw. Rechners (als System betrachtet) ohne Chip-Fertigungsprozesse oder Hardware-Umbauten rein durch Programmierung speziell an die Anforderungen von Anwendungen anpassen zu können.

Mit *dynamischer* oder *Laufzeit-Rekonfiguration* wird der Vorgang bezeichnet, einen rekonfigurierbaren Rechner auch noch während der Ausführung des Algorithmus zu rekonfigurieren.

Partielle Rekonfiguration liegt vor, wenn nur Teile der rekonfigurierbaren Komponenten eines Bausteins oder Systems rekonfiguriert werden. Diese Funktion wird nicht von allen Bausteinen unterstützt und ist orthogonal zur dynamischen Rekonfiguration.

Feinkörnige Parallelität besagt hier, dass sowohl die Funktion der Recheneinheiten als auch ihre Verbindungsstruktur auf der Ebene einzelner Bits konfigurierbar sind. Auf konventionellen Prozessoren werden zumeist ganze Worte (8b, 16b, 32b) betrachtet.

Spezialisierung nennt man die Fähigkeit, auf rekonfigurierbaren Rechnern auch noch jeden einzelnen Hardware-Operator an die Erfordernisse der Anwendung anpassen zu können. Beispielsweise sind so kompakte und schnelle Multiplizierer implementierbar, die mit genau einem konstanten Wert multiplizieren. Analoges gilt für Addierer und andere arithmetische und logische Operationen.

Im Beispiel aus Abschnitt 3.1.2 können bei der rekonfigurierbaren Realisierung zwei der drei Multiplizierer auf die Konstanten A und B spezialisiert werden. Auch einer der Addierer kann auf die Addition von C spezialisiert werden. Die beiden anderen Komponenten (ein Addierer und ein Multiplizierer) können nicht spezialisiert werden, da sie nur variable Eingänge haben.

3.1.4 Abstufungen von Rekonfigurierbarkeit

Der 'Grad' der Rekonfigurierbarkeit eines Chips oder Systems wird im wesentlichen durch zwei Größen bestimmt.

Granularität

Die *Granularität* beschreibt die 'Größe' oder den Funktionsumfang der konfigurierbaren Elemente (Funktionsblöcke und Verbindungsnetze). Hier einige Beispiele für Funktionsblöcke in der Reihenfolge von feinerer zu größerer Granularität:

Einzelne Transistorpaare Diese sind mittlerweile nicht mehr üblich, wurden aber früher z.B. auf FPGAs der Fa. Crosspoint verwendet

Look-Up Tables Sehr geläufig, beispielsweise in den FPGAs von Xilinx oder Lucent.

PLD-artige Strukturen Auch weit verbreitet. Anbieter sind z.B. Altera und Vantis.

ALUs Weniger weit verbreitet, benutzt für arithmetische Anwendungen. Einige Anbieter sind Elixent (4b ALUs) und PACT (24b ALUs). Solche Bausteine werden auch gelegentlich als *network processors* bezeichnet, da sie auf den Einsatz in Netzwerk- und Kommunikationssystemen ausgelegt sind.

Komplette Prozessoren Stark im Kommen. Ein Beispiel ist die MIT RAW Architektur (jeder Funktionsblock ist ein MIPS-artiger RISC mit FPU und eigenen Caches). Kommerzielle Beispiele sind Silicon Hive und Pico Chip. Diese Bausteine werden häufig für drahtlose Kommunikation (Funknetze etc.) verwendet.

Die Granularität der Verbindungsnetze hängt damit unmittelbar von der Granularität der Funktionsblöcke ab. So werden auf den grobkörnigeren Architekturen keine Einzelbitsignale mehr verdrahtet, sondern gleich Multibit-Busse (4b, 8b, 32b) geführt.

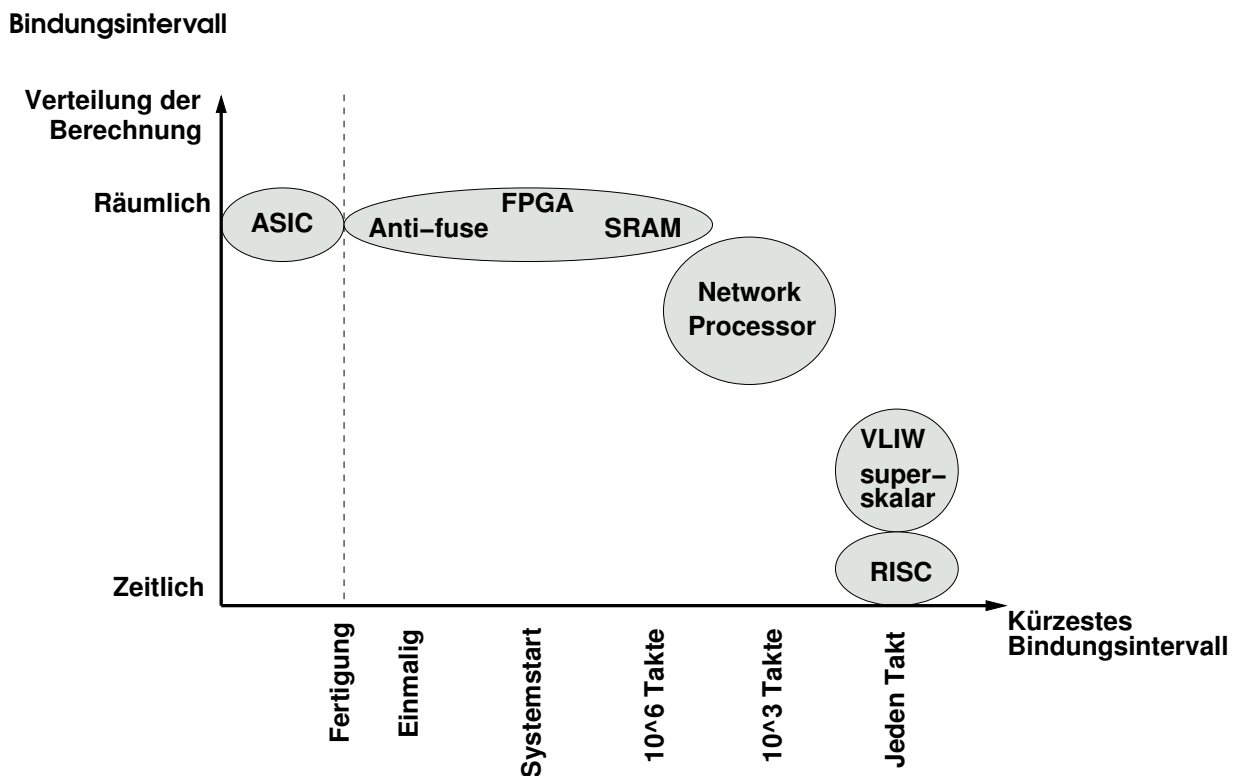


Abbildung 3.2: Berechnungsverteilung und Bindungsintervalle

Das *Bindungsintervall* charakterisiert die Mindestzeit (auch abstrakt), die zwischen zwei Änderungen der Hardware-Funktion liegen muß. Wie im folgenden beschrieben *kann* es sich dabei um Rekonfiguration handeln, dies ist aber nicht zwingend erforderlich (man beachte die beiden Extrema).

Einmalig in der Herstellung In dieses Extrem fallen klassische ASICs und MPGAs. Ihre Hardware-Funktion kann hinterher nur noch stark eingeschränkt variiert werden (in der Regel durch Eintragen von Parameter in Chip-Register).

Einmalig nach der Herstellung Hier werden 'leere Chips' erworben, die genau einmal konfiguriert werden können (z.B. Anti-Fuse basierte FPGAs). Ansonsten gelten die gleichen Einschränkungen wie für ASICs und MPGAs.

Beim Systemstart Bei dieser Variante und allen folgenden Fällen handelt es sich um Lösungen, bei der die Hardware-Funktion durch eine in RAM-ablegbare Beschreibung charakterisiert wird. In dieser Variante ist zum Wechsel der Funktion ein kompletter Neustart des Systems erforderlich. In der Regel wird ein Bindungsintervall dieser Länge für das Aufspielen von Software-Updates oder der Änderung der kompletten Systemfunktion (z.B. verschiedene exklusive Betriebsarten wie *entweder* WLAN-Access Point *oder* Router) genutzt.

10⁶ Prozessortakte Hier sind auch im laufenden Betrieb Anpassungen möglich. Es kann praktikabel werden, für jedes Programm ein oder mehrere angepasste Hardware-Funktionen bereitzustellen.

10³ – 10² Prozessortakte Bei der Kürze dieses Intervalls können auch für einzelne Programmteile (Unterprozeduren, einzelne Schleifen) jeweils angepasste Hardware-Funktionen bereitgestellt werden. Diese Bindungsintervalle markieren den interessantesten Bereich für rekonfigurierbare Rechner.

Jeden Prozessortakt Dieses Extrem wird von klassischen Prozessoren besetzt: Bei einer Instruktion pro Takt ändert sich die von der Hardware ausgeführte Funktion einmal pro Takt. Eine ähnliche Vorgehensweise ist zwar auch bei rekonfigurierbarer Hardware denkbar (Änderung der kompletten Konfiguration jeden Takt), aber impraktikabel: Für jede Neukonfiguration müssen Millionen von Transisto-

ren umgeschaltet werden. Bei der heute in der Regel verwendeten für FPGAs verwendeten CMOS-Technologie fließt zum Zeitpunkt des Umschaltens ein kleiner Strom. Bei Millionen von gleichzeitig schaltenden Transistoren summieren sich diese 'kleinen Ströme' derart auf, dass die Bausteine (ohne exotische Gehäuse und Kühlung) schlicht zu schmelzen beginnen ...

Praktische Auswirkungen

Die Granularität und das Bindungsintervall sind in der Praxis voneinander abhängig. So benötigen grobkörnigere Bausteine deutlich weniger Konfigurationsinformationen als feinkörnigere. Diese veringerte Menge kann dann auch schneller geladen werden (führt also zu kürzeren Bindungsintervallen).

Durch kürzere Bindungsintervalle kann die zur Verfügung stehende Chip-Fläche besser ausgenutzt werden: Man bezahlt schließlich einen hohen Preis (in der Anzahl der nötigen Transistoren), um die Rekonfigurierbarkeit zu erreichen. Dann sollte man sie auch möglichst effizient ausnutzen! Wie oben angedeutet erlauben kürzere Bindungsintervalle die Anpassung der Hardware selbst auf einzelne Programmteile. So können beispielsweise jeweils die Operationen einzelner Schleifen durch individuell angepasste Logik beschleunigt werden.

Letztlich hängt aber die Auswahl eines Bausteines vom Anwendungsgebiet ab: Wenn die gesamte Applikation beispielsweise nur aus einer einzelnen Kernschleife besteht (z.B. einem einfachen Filter), ist ein kurzes Bindungsintervall nicht notwendig. Hier kann die Konfiguration einmal beim Systemstart erfolgen, die Vorteile (kleinerer Standardprozessor beschleunigt durch rekonfigurierbare Komponente) werden aber trotzdem realisiert. Auch ist eine grobe Granularität nicht immer die beste Wahl: Diverse Anwendungen aus dem Krypto- und Netzwerkbereich arbeiten auf einzelnen Bits oder kleinen Bit-Gruppen. Hier würde man also zweckmäßiger feinkörnigere Bausteine einsetzen, die diese direkt (ohne Schiebe- und Maskierungsoperationen) verarbeiten können.

3.1.5 Aufbau adaptiver Rechensysteme

Nachdem wir nun festgestellt haben, dass rekonfigurierbare Rechner eine sinnvolle Alternative zu klassischen Computern darstellen können, gilt es nun zu überlegen, wie ein solches System tatsächlich aufgebaut werden könnte.

Im allgemeinen bestehen Programme zur Lösung praktischer Probleme nicht ausschließlich aus den wenigen Teilen, die das Gros der Rechenintensität ausmachen. Dazu kommen in der Regel noch administrative Aufgaben wie beispielsweise Ein-/Ausgaben, Speicherverwaltung und Fehlerbehandlung. Die rekonfigurierbare Hardware könnte zwar auch diese Aufgaben übernehmen, dies ist aber nicht besonders effizient: Diese Tätigkeiten sind überwiegend nicht besonders zeitkritisch, müssen aber in ähnlicher Form für viele Anwendungen bereitgestellt werden. Anstatt nun eine größere Menge an rekonfigurierbaren Elementen für ihre allgemeine Implementierung zu ver(sch)wenden, läßt man sie doch lieber gleich auf einem Standardprozessor ablaufen. Dieser muß noch nicht einmal besonders leistungsfähig sein, da für die 'Spitzenlast' an Rechenleistung ja die rekonfigurierbare Recheneinheit (im folgenden *RCU* genannt) verwendet wird. Unser adaptives Rechensystem wird also einen Standardprozessor (ab jetzt mit *CPU* bezeichnet) mit rekonfigurierbarer Hardware kombinieren. Wie die nächsten Abschnitte zeigen werden, kann dies aber auf verschiedene Arten geschehen. Die Bilder 3.3 bis 3.7 stellt einige der möglichen Varianten dar.

Freistehende RCU

System mit sehr großem RCU Anteil haben diesen außerhalb des eigentlichen Rechners untergebracht (Bild 3.3). Die Kommunikation erfolgt über eine *externe I/O-Schnittstelle*. Dieser Aufbau wird auch als freistehende RCU ('stand-alone unit') bezeichnet.

Ein typischer Vertreter dieser Gattung wird im wesentlichen für die ASIC-Emulation und als Beschleuniger für HDL-Simulationen genutzt. Eine RCU Kapazität von bis zu 256 Millionen Gatter steht zur Verfügung. Die Anbindung an die CPU erfolgt über das auch bei Festplatten verwendete FibreChannel-Protokoll.

Auf diese Weise lassen sich zwar sehr große rekonfigurierbare Flächen realisieren, aber (neben den anderen

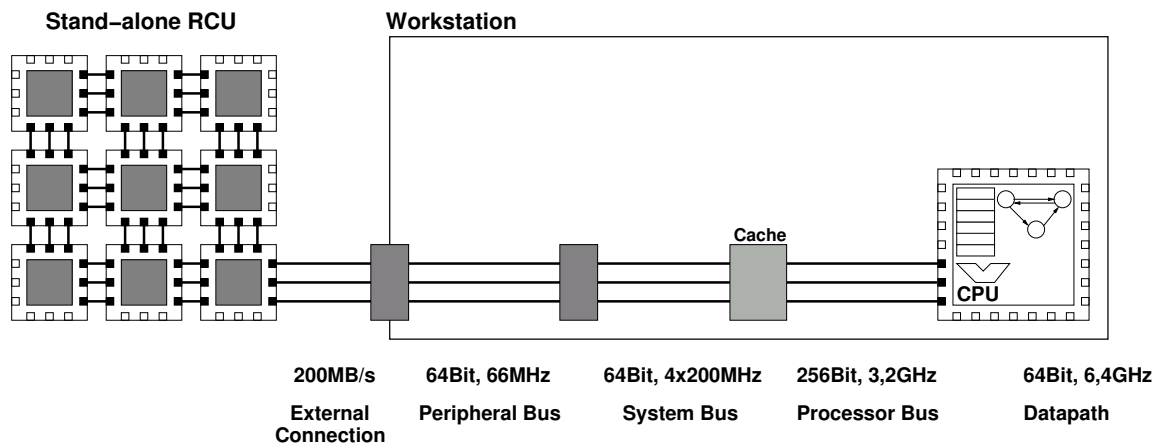


Abbildung 3.3: Freistehende RCU

praktischen Nachteilen ...) beschränkt die hohe Latenz der Kommunikation zwischen CPU und RCU die Art der realisierbaren Anwendungen, wie das folgende hypothetische Beispiel zeigen soll: Angenommen, die RCU kann pro Takt die Arbeit von 40 Prozessorbefehlen leisten. Falls aber jeder Datentransfer zwischen RCU und CPU 100 Takte braucht, lohnt sich dies nur für Algorithmen, die, relativ zu ihrer Gesamtlaufzeit gesehen, nur wenig mit der CPU kommunizieren müssen. Solche Anwendungen existieren zwar, sie stellen aber nur einen kleinen Teil der Gesamtheit der interessanten Applikationen dar.

Angeschlossene RCU

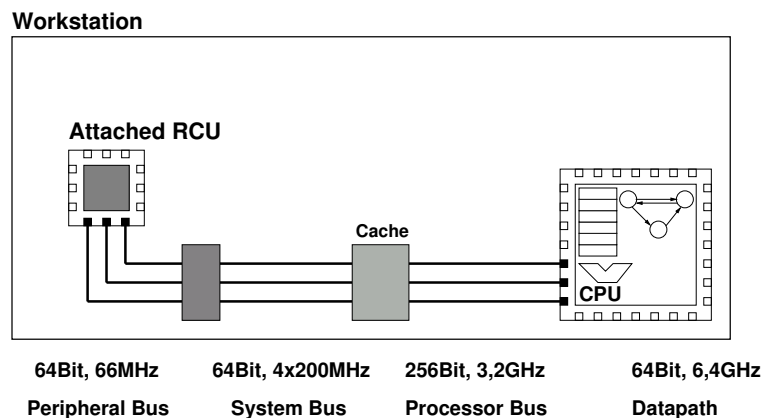


Abbildung 3.4: Angeschlossene RCU

Es ist daher sehr sinnvoll, die Kommunikationslatenz zwischen RCU und CPU so gering wie möglich zu halten. Auf diese Weise könnten auch kleinere (kürzere) Programmteile noch effektiv ausgelagert werden. Die heute gängige Lösung (Bild 3.4) verlagert die RCU direkt in den Rechner und schließt sie dort an den *Peripheriebus* (oft PCI, vereinzelt schon PCI-X oder PCI Express, aber noch sind auch ältere Bussysteme wie SBus oder VME in Gebrauch) an. So sind bei dem derzeit gängigen 32b PCI Bus getaktet mit 33 MHz theoretische Datenübertragungsraten von 132 MB/s erreichbar. Aber auch hier sind die Latenzen noch nennenswert: Ein Schreibzugriff vom PCI Bus auf den Hauptspeicher dauert circa 10 Bustakte (330ns), ein Lesezugriff gar über 30 Bustakte ($> 1\mu s$). Der Vorteil dieser Anbindung ist der problemlose Anschluß an die leicht handhabbaren Peripheriebusse von Standardrechnern mittels einer einfachen Steckkarte. Bei dieser Lösung wird von einer angeschlossenen RCU ('attached processing unit') gesprochen.

RCU in Multiprozessorsystem

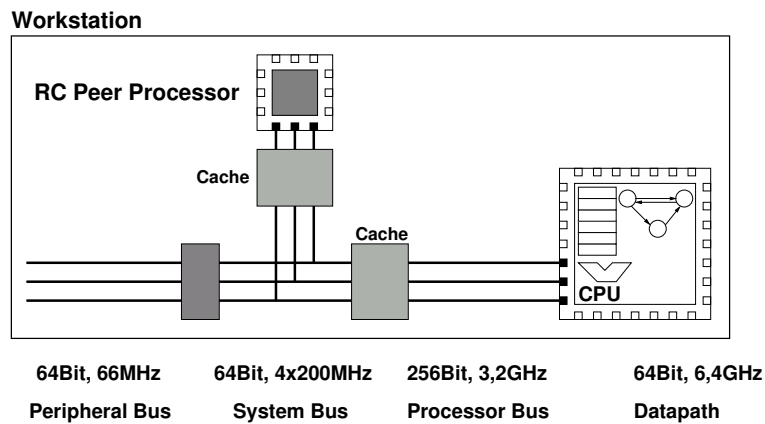


Abbildung 3.5: Multiprozessoren: RCU gleichberechtigt mit der CPU

Der Aufbau von adaptiven Rechnern mit noch engerer Kopplung von RCU und CPU wird nun zunehmend schwieriger. Eine Möglichkeit sieht die RCU und die CPU als *gleichberechtigte Partner* auf dem Prozessorbus an (Bild 3.5). Eine solche Anordnungen von gleichberechtigten Prozessoren wird als symmetrischer Multiprozessor (SMP) bezeichnet. Der Geschwindigkeitszuwachs gegenüber Peripheriebussen ergibt sich aus dem sehr viel höheren Bustakt (beispielsweise mit 800 MHz) und den kürzeren Latenzen (im Bereich von 4-40 Takten). Die Protokolle für die Interprozessorkommunikation sind zwar nicht trivial (es müssen unter anderem die unabhängigen Caches von RCU und CPU kohärent gehalten werden), lassen sich aber mit der nötigen Geschwindigkeit auch noch in FPGAs realisieren. Mittlerweile gibt es Trends, Prozessorbusse wie HyperTransport mittels gut handhabbarer Steckverbinder (HTX) zu nutzen, um verschiedenste Recheneinheiten (z.B. Kryptobeschleuniger, aber auch RCUs) leicht auf kommerziell erhältlichen PC-Motherboards einzusetzen.

RCU-Koprozessor

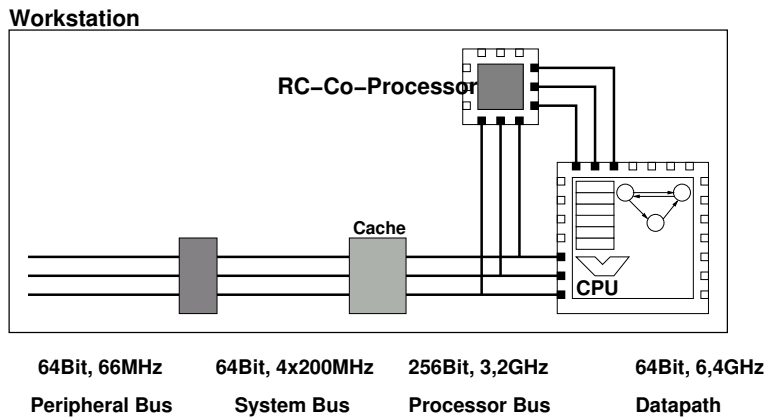


Abbildung 3.6: RCU als Koprozessor

Die Anbindung der RCU zusammen mit der CPU an einen *gemeinsamen Cache* verkürzt die Latenzen in der Regel noch weiter (Bild 3.6). Bei dieser Kombination agiert die RCU als echter Koprozessor für die CPU. Die Kommunikation zwischen RCU und CPU kann dann in 10 Prozessortakten vorgenommen werden, ein Speicherzugriff über den Cache kann im Erfolgsfall (cache hit) in ähnlicher Zeit bewältigt werden. Solche Architekturen lassen sich zwar noch nicht mit den in PCs üblichen CPUs realisieren (diese sehen

schlicht keine RCU auf dem Chip vor). Aber spezielle konfigurierbare Prozessoren wie die Tensilica Xtensa IP-Blöcke werden auch heute schon mit RCUs auf einem Chip kombiniert (z.B. in der Stretch S5000 Architektur). Verschiedene sogenannte System-FPGAs (z.B. Xilinx Virtex II Pro, 4FX und 5FX) enthalten bereits einen oder mehrere Prozessoren auf dem Chip und erlauben so den Koprozessorbetrieb von RCU und CPUs. Hier dreht sich dann der Spiess um, indem die CPU nun in die RCU eingebettet wird.

RCU-Funktionseinheit in CPU

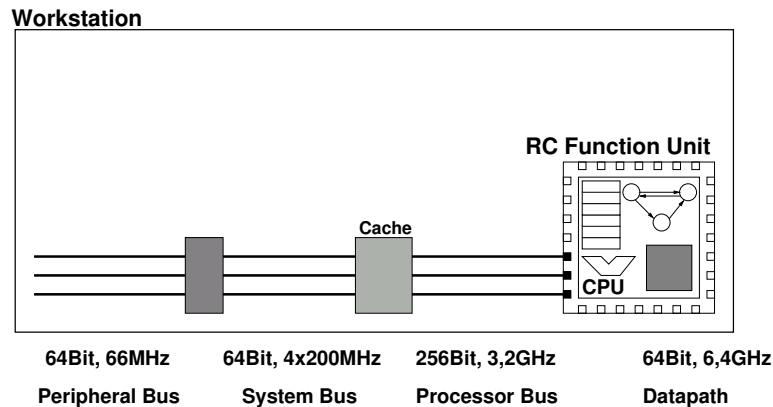


Abbildung 3.7: RCU als Funktionseinheit in CPU

Auch eine noch engere Integration ist denkbar: Die RCU könnte als *Funktionseinheit* (RFU) direkt in den Prozessorkern hineinintegriert werden (Bild 3.7). Ähnlich wie eine ALU mit festen Funktionen würde dann auch eine rekonfigurierbare Funktionseinheit bereitstehen. Im akademischen Bereich sind solche Chips bereits realisiert worden. Bei den ersten Ansätzen zeigte sich aber, dass die Anbindung zwar mit niedriger Latenz (1 Prozessortakt), aber auch mit nur niedriger Übertragungsgeschwindigkeit benutzbar war: Ähnlich wie bei den anderen CPU-Befehlen konnten auch an die RCU nur einzelne Register (in der Regel also 32b Worte) übergeben werden. Auch das Ergebnis der Berechnung wurde in einem einzelnen Zielregister abgelegt. Obwohl also in der RCU deutlich aufwendigere Berechnungen realisierbar waren, wurden diese durch die sehr niedrige Kommunikationsbandbreite 'ausgehungert' (bekamen nicht genug Daten). Neuere Experimente mit RCU-Funktionseinheiten versuchen, dieses Problem durch Bereitstellen RCU-eigener Speicherschnittstellen zu umgehen. Dabei müssen aber die Interaktionen zwischen den Speicherzugriffen des Prozessors und der RCU-Funktionseinheit sorgfältig koordiniert werden. Kommerziell ist allerdings noch kein Vertreter dieser Gattung von adaptivem Rechner verfügbar. In einer anderen Spielart (z.B. den Stretch S5000 Chips) werden die RFUs über spezielle, breitere Register mit Daten versorgt, welche durch geeignete LOAD/STORE-Anweisungen der CPU gehandhabt werden. Zwar können hier in einer Instruktion auch größere Datenmengen als die üblichen 32b ausgetauscht werden (nämlich 128b), die Kommunikationsanweisungen sind aber oft eingeschränkt (erlauben beispielsweise wohl einen Datentransfer aus einem CPU in ein breites RCU-Register, aber nicht in die Gegenrichtung). Als weiteres Problem erweist sich in der Praxis, die langsamere Taktfrequenz der rekonfigurierbaren Einheiten mit dem Takt der hochgezüchteten festen ALUs (die teilweise mit über 7 GHz Takt arbeiten) des Prozessors in Einklang zu bringen. RFUs kommen daher in der Regel nur auf ohnehin langsameren Prozessoren für eingebettete Systeme zum Einsatz. Hier fallen die Taktdifferenzen weniger extrem aus (z.B. 300 MHz CPU-Takt zu 100MHz RFU-Takt).

Weitere Systemkomponenten

Obwohl wir uns an dieser Stelle nur auf die Hardware-Aspekte konzentriert haben, besteht ein adaptives Rechensystem natürlich auch noch aus Software. Und damit sind hier noch nicht die Werkzeuge gemeint, um ein solches System zu programmieren (siehe dazu Abschnitt 3.2), sondern nur die für den Betrieb erforderlichen.

Auch ein adaptiver Rechner benötigt ein Betriebssystem auf der CPU, das im einfachsten Fall nur den Zugriff auf verschiedene Peripherie bereitstellt und in der Lage ist, Benutzerprogramme zu starten. Häufig werden noch weitere Funktionen wie die Speicherverwaltung und das schnelle Umschalten zwischen mehreren Anwendungen (Multitasking/-threading) vom Betriebssystem übernommen.

Bei adaptiven Rechnern kommt zu diesen Standardaufgaben auch noch die Interaktion zwischen CPU und RCU hinzu. Beispiele für solche Operationen sind:

- Aufbau einer Kommunikationsverbindung zur RCU.
- Laden einer Konfiguration in die RCU.
- Übertragen von Daten in die RCU.
- Starten der Berechnung auf der RCU.
- Überprüfen des Berechnungsstatus der RCU.
- Übertragen von Daten von der RCU.
- Abbau der Kommunikationsverbindung zur RCU.

Bei sehr engen Kopplungen werden viele dieser Aufgaben direkt in Hardware realisiert. So genügt dort in der Regel ein einzelner CPU-Befehl, um Daten in die RCU zu übertragen.

In jedem Fall sollte die Komplexität des Software-Aspekts nicht unterschätzt werden. Selbst wenn die Hardware schon länger fehlerfrei vorliegt, ist es noch häufig ein längerer Weg, bis das ganze *System* erfolgreich arbeitet. Und viele Probleme tauchen erst bei der Systemintegration (dem ersten Zusammenbau und gemeinsamen Test der einzelnen Komponenten) auf.

3.1.6 Auswirkungen auf die Architektur von Prozessoren

Nach den ganzen vorangegangenen Erläuterungen mag der Leser argumentieren: "Das ist ja alles gut und schön, aber ich warte einfach 18 Monate, und dann ist auch mein Prozessor von der Stange so schnell ('ganz viele Gigahertz'), dass sich der ganze rekonfigurierbare Aufwand gar nicht lohnt". Leider ist das nicht mehr selbstverständlich: Wahr ist, dass sich derzeit circa alle 18 Monate die Zahl der auf dem Chip realisierbaren Transistoren verdoppelt (Moore's Gesetz). Aber eine Verdoppelung der Transistorzahl führt nicht zwangsläufig zu einer Verdoppelung der Rechenleistung.

Daneben darf nicht vergessen werden, dass natürlich auch FPGAs von den Fortschritten der Fertigungsprozesse profitieren. Die sehr reguläre Struktur moderner FPGAs schlägt sich auch in einem sehr regulären Chip-Layout nieder. Es müssen also nur vergleichsweise kleine Blöcke 'von Hand' an die Design-Rules eines neuen Prozesses angepasst werden. Das gesamte FPGA wird dann im wesentlichen durch Vervielfältigen dieser Tiles (=Kacheln) in Schachbrett-Manier aufgebaut. Durch diese schnelle Anpassbarkeit gehören FPGAs häufig zu den ersten Schaltungen, die auf neuen Prozessen gefertigt werden. Sie treiben also deren Entwicklung voran und agieren so als 'process drivers'.

Beispiel: Wir betrachten hier die Intel Pentium-III Familie. Diese ist zwar nicht mehr ganz taufersch, aber im Gegensatz zu aktuellen Prozessoren können wir hier die Effekte bei Vervielfachung von Transistoranzahl und Taktfrequenz bei gleichbleibender Basisarchitektur über einen längeren Zeitraum beobachten. Ein Intel Pentium-III Prozessor (9.5 Millionen Transistoren, externer L2 Cache) mit 500 MHz Taktfrequenz erreicht die Werte 20.6 und 14.7 im SPEC Benchmark (für Integer und Floating Point-Operationen). Eine neuere Pentium-III Variante mit 28.5 Millionen Transistoren (jetzt mit dem L2 Cache direkt auf dem Chip) und sagenhaften 1000 MHz Taktfrequenz erreicht nun die Werte 46.8 und 32.2. Das sieht auf den ersten Blick gut aus (Beschleunigung um den Faktor 2.3 bzw. 2.2). Aber: Neben einer Verdoppelung des Taktes war für das Erreichen dieser Werte auch eine Verdreifachung der Transistorzahl erforderlich. Für immer weniger Gewinn an Rechenleistung ist immer mehr Aufwand erforderlich. Man sollte sich also nach Alternativen zu diesen Holzhammermethoden umsehen ...

Anwendung	Anzahl Gatter
JPEG Encoder	75K
MPEG-2 Decoder	105K
MPEG-2 Codec	1.5M
UMTS Basisstation Modem	4.5M
Einfacher 3D Grafikprozessor	20M

Tabelle 3.1: Gatterkomplexitäten einiger Anwendungen

Die Möglichkeit, die Struktur eines variablen FPGA-basierten Prozessors speziell auf das aktuelle Problem abzustimmen, eröffnet ganz neue Perspektiven für die Architektur von leistungsfähigen Prozessoren. Aber ist ihr Einsatz auch praktikabel?

Heutige Fertigungsprozesse erlauben die Herstellung von Prozessoren bis hin zu 681 Millionen (Nvidia G80 Grafikbeschleuniger) oder 1,7 Milliarden Transistoren (Intel Montecito). Was aber wird bisher mit dieser gewaltigen Chip-Fläche angefangen? Die klassische Prozessor-Architektur zeigt sich hier wenig erfindereich: Immer größere Caches (z.B. 1.5MB L1 beim HP PA-8700, damit läuft der SPEC Benchmark komplett aus dem Cache, oder 26MB in L1...L3 auf dem Montecito), höhere Integration (z.B. Speicher-Controller on-chip) oder gar mehrere ausgewachsene Prozessoren auf dem gleichen Chip (HP PA-8800 = 2x PA-8700). Immer häufiger wird aber die Frage gestellt, ob diese traditionellen Denkweisen in Anbetracht der immer weiter wachsenden Chip-Flächen noch sinnvoll sind.

Ein aktueller Ansatz schlägt daher vor, einen Teil der zur Verfügung stehenden Transistormenge für einen rekonfigurierbaren FPGA-artigen Bereich zu verwenden, der in Zusammenarbeit mit einer konventionellen CPU für erhöhte Leistung und/oder einen verminderten Stromverbrauch sorgt. Um eine frei konfigurierbare Kapazität von 1 Million Gatter zu erreichen, werden ca. 75 Millionen Transistoren benötigt. Es ist also durchaus sinnvoll, bei dem o.g. Transistorbudget über Kombinationen eines konventionellen Prozessors und einer rekonfigurierbaren Komponente nachzudenken.

Tabelle 3.1 zeigt die Komplexität in Gattern für einige ausgewählte Anwendungen. Man sieht, dass sich auch schon mit recht kleinen RCU-Kapazitäten sinnvolle (und für konventionelle Prozessoren sehr rechenintensive!) Probleme bearbeiten lassen. Die beiden letztgenannten Anwendungen sind für aktuelle Standard-Prozessoren überhaupt nicht mehr handhabbar (zu harte Echtzeitanforderungen). Es ist nun aber nicht erforderlich, zu ihrer Realisierung mittels RCU tatsächlich Millionen frei programmierbarer Gatter mit immensem Transistoraufwand zu verplanen. Tatsächlich verbringen die überwältigende Mehrzahl von Anwendungen das Gros (> 90%) ihrer Rechenzeit in zeitunkritischen Programmteilen. Diese können genauso gut auch auf einem Standardprozessor ausgeführt werden. Nur die wirklich zeitkritischen Teile des Algorithmus müssen als RCU-Konfiguration ausgelagert werden. Erste Bausteine die diese Kombination von CPU und RCU auf einem Chip vereinen, setzen diese Ideen mittlerweile praktisch in die Tat um.

3.1.7 Beispiel: ML310 als adaptiver Computer

Nach all diesen doch eher theoretischen Diskussionen soll nun ein real existierendes adaptives Rechen-system vorgestellt werden. Auf Basis eines Xilinx Virtex II pro-Prototypenboards wurde am FG ESA ein moderner adaptiver Computer (*adaptive computing system, ACS*) aufgebaut.

Da die für eine hohe Rechenleistung kritischen Teile (Anbindung von CPU, RCU und Speicher) hier flexibel variierbar *innerhalb* des FPGAs vorliegen, kann leicht mit verschiedensten Architekturen experimentiert werden. Dabei folgt das ML310 ACS dem Modell eines RCU-Koprozessors.

Hardware-Architektur

Als CPU des ACS fungiert einer der beiden 300 MHz PowerPC 405 Kerne des Virtex II pro-Chips (der andere Kern ist derzeit noch unbenutzt). Neben einigen vergleichsweise kleinen Hardware-Blöcken, die Schnittstellen nach außen, z.B. zum Ethernet- oder Festplatten-Controller herstellen, wird ein signifikanter Teil des verbliebenen Platzes durch den Speicher-Controller belegt. Dieser stellt für CPU und RCU den

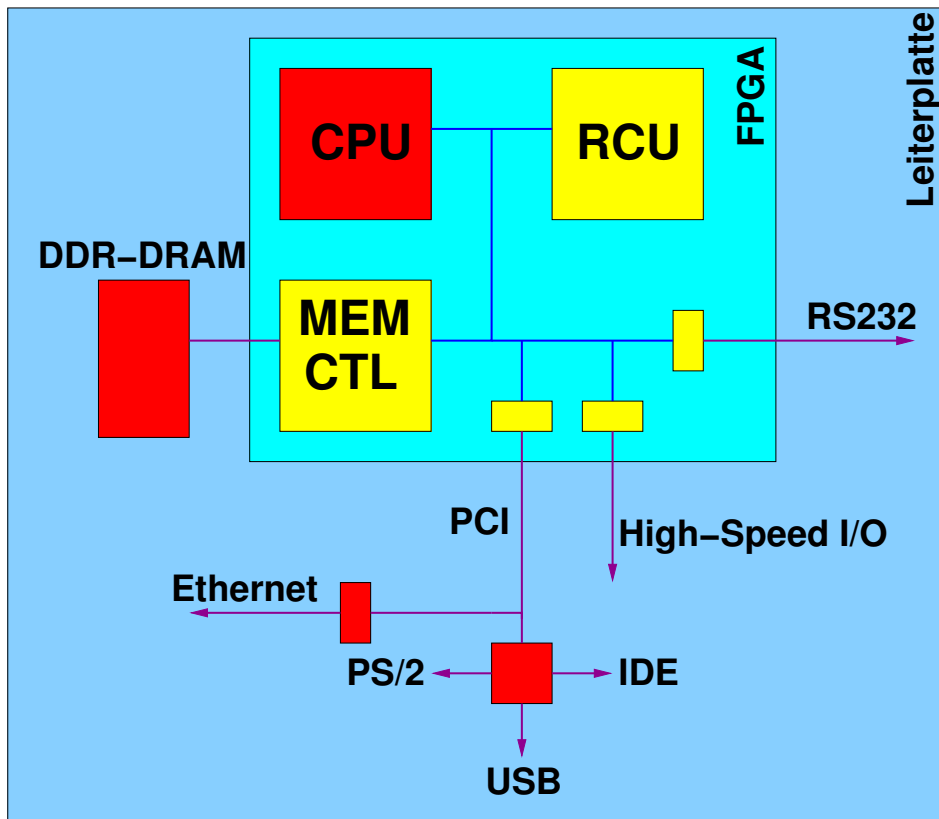


Abbildung 3.8: ML310 Architektur

Zugang zum externen System-Hauptspeicher her, der in Form eines 256MB großen DDR-DRAM Moduls realisiert ist.

Eine Eigenheit der verwendeten Hardware ist, dass sie mit Systemtaktfrequenzen unterhalb von 100 MHz nicht mehr zuverlässig arbeitet. Alle Hardware-Anwendungen für die RCU müssen also *mindestens* diese Taktfrequenz erreichen.

Das ML310 wäre komplett eigenständig läuffähig (mit eigener Grafikkarte, Tastatur, Festplatten, etc.), wird aber für einfachere Experimente stattdessen an einem Wirtsrechner (konventioneller PC) betrieben. Dabei können über eine serielle Schnittstelle Daten auch ohne großen Hard- und Software-Aufwand zwischen den beiden Rechnern ausgetauscht werden (sehr hilfreich, um bei maschinennaher Entwicklung wenigstens eine einfache Konsole für Debug-Meldungen zu haben). Im Normalbetrieb erfolgt die Kommunikation aber über ein dediziertes lokales Ethernet zwischen ML310 und Wirtsrechner, über das das ACS auch auf die Dateisysteme des Wirts (via NFS) zugreifen kann.

Software-Architektur

Von Anfang an war einer der Ziele der Entwicklung des ML310 ACS, dass das System mit einem Standardbetriebssystem laufen soll. Zu diesem Zweck wurde eine Linux-Portierung geeignet erweitert, so dass nun die Kommunikation zwischen CPU und RCU auch bei einem geschützten und virtuellen Speichermodell aus Software-Programmen gesteuert werden.

3.2 Programmierung adaptiver Rechner

Während sich die Programmierung von Standardprozessoren zwischen den unterschiedlichen Typen kaum voneinander unterscheidet, bestehen dramatische Unterschiede bei der Programmierung von verschiedenen adaptiven Rechensystemen. Da hier wegen der Freiheit bei der Strukturierung der Hardware vom üblichen von-Neumann-Modell abgewichen werden kann, können die unterschiedlichsten Ansätze praktisch zum Einsatz kommen.

Einige Beispiele sind systolische Arrays (wie beim DNA-Sequenz-Vergleich), verschiedene andere Datenflußansätze (z.B. SDF und BDF) und Architekturen wie VLIW/EPIC oder Vektorprozessoren (SIMD). Durch Ausnutzung der Adaptionfähigkeit kann hier für jedes Problem das am besten geeignete Rechenmodell zum Einsatz kommen.

Im folgenden Abschnitt erläutern wir die Programmierung mittels direkter Beschreibung der auf der RCU ablaufenden Hardware in einer HDL (in unserem Fall Verilog). Dies war früher die einzige Form, ein ACS zu programmieren. Der Programmierer hat damit zwar uneingeschränkte Freiheit, die für das aktuelle Problem passende Zielarchitektur auf der RCU zu realisieren. Diese Art der Programmierung setzt aber detaillierte Kenntnisse der ACS-Hardware und des Hardware-Entwurfs im allgemeinen voraus. Neben verschiedenen technischen Problemen (z.B. unzureichendes Fassungsvermögen der RCU) war dies das Haupthindernis bei einer breiteren Benutzung von ACSs: In der Regel verfügen Anwender, die nur mit konventioneller Software-Entwicklung vertraut sind, nicht über die erforderlichen Kenntnisse, ein ACS erfolgreich zu programmieren.

Um die Vorteile des adaptiven Rechnens einer breiteren Benutzerschicht zugänglich zu machen, wird heute versucht, durch geeignete Werkzeuge die Lücke zwischen den Hard- und Software-Entwurfsebenen zu schließen. Es handelt sich dabei in erster Linie um Ansätze aus dem Hardware-Software-Codesign, bei denen der gesamte Hardware-Zweig der Anwendung vollautomatisch erstellt wird. Der Benutzer formuliert die Programme in einer ihm vertrauten höheren Programmiersprache (beispielsweise C, Java oder MATLAB) und ruft die Werkzeuge in gewohnter Form auf (ähnlich einem normalen Software-Compiler). Im Idealfall wird ihm dadurch ohne weiteres Zutun eine hybride Hardware-Software-Anwendung erzeugt, die direkt auf dem ACS ausführbar ist. Als Beispiel für einen solchen Entwurfsfluß sei hier der Compilerprototyp NIMBLE genannt, der aus Standard ANSI C (ohne jede Einschränkung oder Erfordernis von Sonderkonstrukten) automatisch ACS-Anwendungen erzeugt.

3.3 HDL-basierte Programmierung

Die HDL-basierte Programmierung von ACSs werden wir am konkreten Beispiel einer Anwendung für die ML310 Plattform vorstellen. Aus Übersichtsgründen wird dabei ein sehr einfaches Problem bearbeitet, das aber viele wichtige Grundkonzepte bereits illustriert: Es soll eine Datenkonvertierung derart vorgenommen werden, dass in den Ausgabedaten die Reihenfolge der Bits der Eingabedaten verdreht ist. Die Daten selbst bestehen aus 32b Worten. So landet also Bit 0 eines Eingabewortes auf Bit 31 des Ausgabewortes, Bit 1 der Eingabe auf Bit 30 der Ausgabe etc. (siehe Abbildung 3.9).

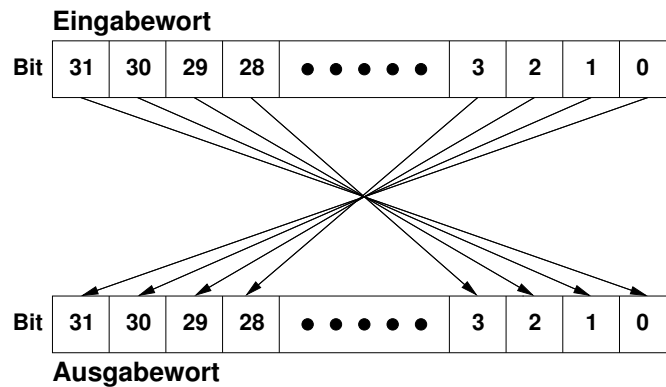


Abbildung 3.9: Funktion der Beispieldanwendung

Dieses Beispiel ist nicht so konstruiert, wie es auf den ersten Blick erscheint. Solche Transformationen treten beispielsweise bei der Umsetzung von Kommunikationsprotokollen auf. Und auch während der Schaltungsgenerierung für Xilinx FPGAs findet eine solche Konvertierung statt: Der von den Werkzeugen erstellte Bitstream muß genau so bearbeitet werden, bevor er tatsächlich in einen Chip geladen werden kann. Wir stellen drei Lösungen für das Problem vor. Dabei werden wir jeweils den Platzbedarf (für die Hardware-unterstützten Ansätze) und die Ausführungszeit des kritischen Blocks (für alle Lösungen) untersuchen

1. Eine reine Softwarelösung in C (Abschnitt 3.3.1).
2. Eine einfache Hardware-unterstützte Lösung, bei der sowohl Ein- als auch Ausgabedaten einzeln durch die CPU übertragen werden (slave-mode, Abschnitt 3.3.2)
3. Eine aufwendigere Hardware-unterstützte Lösung, bei der die RCU nach Übergabe von Parametern durch die CPU selbstständig die Daten bearbeitet (master-mode, Abschnitt 3.3.3).

Die bei der HDL-basierten Programmierung eingesetzten CAD-Werkzeuge unterscheiden sich nicht von denen für den traditionellen FPGA-Entwurf. Es kommen Simulation, Synthese, Platzierung, Verdrahtung und ggf. noch die Timing-Analyse zum Einsatz. Lediglich für das Zusammenbinden der Hardware-Komponenten mit der Software (die ebenfalls mittels eines üblichen C-Compilers bearbeitet wird) sind Spezialprogramme erforderlich.

3.3.1 Reine Softwarelösung

Der Programmtext der reinen Software-Lösung ist in Listing 3.1 zu sehen. Um den Code auf das Wesentliche zu beschränken, wurde auf (für diese Erklärung) weniger wichtige Teile wie Fehlerüberprüfungen verzichtet. Reale Anwendungen sollten diese Ausnahmen sehr wohl korrekt behandeln! Die generelle Vorgehensweise zur Lösung der Aufgabe ist sehr einfach: Nach einigen administrativen Anweisungen beginnt der Programmablauf in Zeile 23. Hier werden mittels der Funktion `malloc()` zwei Speicherbereiche zum Aufnehmen der Ein- und Ausgabedaten angefordert. Deren Größe berechnet sich als die Anzahl der Datenworte `NUM_WORDS`, hier gesetzt als `256Kw`, multipliziert mit der Größe eines Wortes `long` in Bytes,

hier 4B. Der Zeiger inwords zeigt danach auf den Speicherbereich für die Eingangsdaten, der Speicher outwords auf den für die Ausgangsdaten.

Der Zeilenblock 28-29 öffnet die Eingabedatei test1.in und legt die Ausgabedatei test1.out neu an. In Zeile 32 wird die gesamte Eingabedatei auf einen Satz in den durch inwords adressierten Speicherbereich eingelesen.

Listing 3.1: Reine Softwarelösung

```
#include <stdio.h>
#include <stdlib.h>

// Anzahl von Ticks der Systemuhr pro Mikrosekunde
#define TICKS_PER_USEC 25

// Anzahl Datensätze in Ein- und Ausgabedatei
#define NUM_WORDS (1024*256)

main()
{
    // Ein- und Ausgabedateien
    FILE *infile , *outfile ;

    // Benutze vorzeichenlose Ganzzahlen (32b Worte) für alle Daten
    unsigned long n, m, mask, set, inword, outword;
    // Zeiger auf Ein- und Ausgabe-Speicherbereiche
    unsigned long *inwords, *outwords;

    // Marker für Zeitmessung (64b Variablen)
    unsigned long long start, stop, RTEMSIO_getTicks();

    // fordere Speicher für Ein- und Ausgabefelder an
    inwords = malloc(NUM_WORDS * sizeof(unsigned long));
    outwords = malloc(NUM_WORDS * sizeof(unsigned long));

    // Öffne die Dateien zum Lesen und Schreiben
    infile = fopen("test1.in", "r");
    outfile = fopen("test1.out", "w");

    // Lese komplette Eingabedatei in Eingabe-Speicherbereich
    fread(inwords, sizeof(unsigned long), NUM_WORDS, infile);

    // Merke Startzeit der Berechnung in Ticks
    start = RTEMSIO_getTicks();

    // Bearbeite Daten wortweise
    for (m=0; m < NUM_WORDS; ++m) {
        inword = inwords[m];
        outword = 0;
        mask = 1;
        set = 1 << 31;
        // Baue das verdrehte Ausgabewort bitweise auf
        for (n = 0; n < 32; ++n) {
            if (inword & mask)
                outword |= set;
            mask <<= 1;
            set >>= 1;
        }
        // Trage das Ergebnis in das Ausgabe-Array ein
        outwords[m] = outword;
    }
}
```

<code>// Ende der Berechnung, merke Stopzeit in Ticks</code>	54
<code>stop = RTEMSIO_getTicks();</code>	55
 <code>// Schreibe das komplette Ausgabe-Array in die Ausgabedatei</code>	57
<code>fwrite(outwords, sizeof(unsigned long), NUM_WORDS, outfile);</code>	58
 <code>// Gebe Speicher für Ein-/Ausgabe-Arrays wieder frei</code>	60
<code>free(inwords);</code>	61
<code>free(outwords);</code>	62
 <code>// Schließe Dateien</code>	64
<code>fclose (infile);</code>	65
<code>fclose (outfile);</code>	66
 <code>// Gebe Ergebnis der Zeitmessung in Mikrosekunden aus</code>	68
<code>printf("Zeit: %lldµs\n", (stop-start)/TICKS_PER_USEC);</code>	69
<code>}</code>	70

Vor der eigentlichen Datenkonvertierung wird in Zeile 35 die aktuelle Zeit in der Variablen `start` gemerkt. Man beachte, dass hier als Einheit sogenannte "Ticks" der Systemuhr verwendet werden. Auf dem ML310 hat ein Tick eine Länge von 40ns (siehe Zeile 5). Um daher die potentiell sehr großen Zahlenwerte verarbeiten zu können, sind die Variablen für die Zeitmessung als 64b Variablen (Datentyp `long long`) deklariert worden (Zeile 18).

Die in Zeile 38 beginnende Schleife durchläuft alle Datenworte im Eingabe-Speicherbereich `inwords`. Die innere Schleife mit Kopf in Zeile 44 durchläuft die Bits jedes der Eingabeworte. Dabei wird nach Überprüfung, ob ein Bit im aktuellen Eingabewort `inword` gesetzt ist, das entsprechende "verdrehte" Bit im Ausgabewort `outword` gesetzt. Am Ende der äußeren Schleife wird schließlich das fertige Ausgabewort in den Ausgabe-Speicherbereich `outwords` eingetragen (Zeile 51).

Am Ende der eigentlichen Berechnung rufen wir in Zeile 55 wieder die aktuelle Zeit ab. Da die vergleichsweise langsamen Dateioperationen der ML310 Plattform (NFS via Ethernet) alle Zeitmessungen unnötig dominieren würde, beschränken wir uns bei unseren Versuchen auf die Messung der reinen Rechenzeit anstatt der Laufzeit des gesamten Programmes, was realistischer wäre.

In Zeile 58 werden die fertigen Ausgabedaten in einem Schwung aus ihrem Speicherbereich in die vorher geöffnete Datei geschrieben. Abschließend geben wir die Speicherbereiche wieder frei und schließen die Dateien. Das Programm endet mit der Ausgabe der Laufzeit der Berechnung in Mikrosekunden.

Nach Übersetzung der C-Quelldatei kann die so entstandene Binärdatei des Programms auf dem ML310 ausgeführt werden. Für die Bearbeitung des 256Kw großen Datensatzes mit dieser reinen Softwarelösung werden dazu **195942µs**, also rund 0.2s benötigt.

3.3.2 Beschleunigung durch RCU im Slave-Mode

Auswahl geeigneter Programmteile für Beschleunigung

Bei komplizierteren Anwendungen kann die Ermittlung der wirklich zeitkritischen Programmteile recht aufwendig sein. Die dabei verwendeten Methoden (auch als *Profiling* bezeichnet) basieren in der Regel auf speziellen Werkzeugen, die für jeden einzelnen Programmteil (teilweise sogar für einzelne Zeilen oder gar Maschinenbefehle) die spezifischen Ausführungszeiten messen. Aus diesen Angaben kann dann der Entwickler (oder weitere automatische Werkzeuge) lohnende Programmteile für die Auslagerung in Hardware isolieren. Im Fall unser Beispielanwendung ist solch ein Aufwand nicht erforderlich, es ist offensichtlich, dass das Gros der Berechnungszeit in den Zeilen 38–52 (den beiden verschachtelten Schleifen) liegt.

Nicht alle lohnenden Programmteile sind gleichermaßen für eine Auslagerung in die RCU geeignet. So ist es beispielsweise oft nicht sinnvoll zu versuchen, Fließkommaoperationen (Datentypen `float` und `double`) auf heutige gängige RCUs zu verlagern. Die Nachbildung der dafür benötigten Recheneinheiten benötigt relativ viel Platz. Solche Operationen sind auf den darauf spezialisierten Floating-Point Units (FPU) der

CPU besser aufgehoben. Eine ähnliche Situation liegt bei Ein-/Ausgabe-Funktionen wie `fopen()` und `fread()` vor. Diese haben häufig komplizierte Datenstrukturen und Kontrollflüsse, so dass der Versuch einer Auslagerung in die RCU die Implementierung einer fast kompletten CPU nach sich ziehen würde.

In unserem Fall ist die Lage aber sehr viel einfacher. Beide Schleifen enthalten lediglich einfache arithmetische und logische Operationen auf ganzen Zahlen (dargestellt durch 32b Worte). Solche Konstrukte lassen sich sehr einfach und effizient in Hardware abbilden.

Hardware-Schnittstelle der RCU

Aber nur die Realisierung der logischen Funktion reicht hier nicht mehr aus. Auf irgendeine Art und Weise müssen schließlich die Daten in die RCU eingespeist und die Ergebnisse ausgelesen werden. Am einfachsten (aber in der Regel nicht am effizientesten) ist dies, wenn die CPU explizit jedes Datum an die RCU überträgt, diese die Berechnung ausführt, und die CPU schließlich das Ergebnis abrufen. Dieses Füttern der RCU mit einzelnen Daten-Happen bezeichnet man als Slave-Mode Betrieb der RCU.

Schauen wir uns zunächst einmal die Hardware an, die bei diesem Vorgehen für unsere Anwendung in der RCU realisiert werden muß (Listing 3.2).

Die Schnittstelle des Moduls zur Kommunikation mit der CPU ist dabei vorgegeben. Neben den üblichen CLK und RESET-Signalen bestimmen fünf Leitungen das Interface:

ADDRESSED zeigt durch Annehmen des Wertes High einen Zugriff von der CPU auf die RCU an. In diesem Fall müssen die folgend beschriebenen Anschlüsse ausgewertet oder getrieben werden.

WRITE Wenn ADDRESSED und WRITE beide High sind, so liegt ein Schreibzugriff der CPU auf die RCU vor. Ist WRITE bei gesetztem ADDRESSED dagegen Low, versucht die CPU Daten von der RCU zu lesen. Wenn ADDRESSED nicht gesetzt ist, kann WRITE ignoriert werden, da die RCU nicht angesprochen wird.

DATAIN Bei einem Schreibzugriff liegen die von der CPU geschriebenen Daten auf diesem Eingangs-Bus so an, dass sie zur nächsten Taktflanke in ein lokales Register eingelesen werden können. Außerhalb eines Schreibzugriffes liegen auf diesem Bus keine gültigen Daten an, er kann also von der RCU ignoriert werden.

DATAOUT Bei einem Lesezugriff erwartet die CPU auf diesem Bus die angeforderten Daten von der RCU. Diese müssen für die gesamte Dauer des Lesezugriffes stabil sein. Außerhalb des Lesezugriffes ignoriert die CPU die auf diesem Bus von der RCU ausgegebenen Daten, er muß nicht explizit hochohmig (Z) gesetzt werden.

ADDRESS Durch diesen Bus teilt die CPU der RCU während eines Lese- oder Schreibzugriffes mit, *welche* Daten konkret gelesen oder geschrieben werden sollen. Die Zuordnung von Adressen zu Daten ist dabei frei, Software- und Hardware-Teile müssen sich aber über die Interpretation der Adressen einig sein. Außerhalb eines CPU-Zugriffes auf die RCU können die Werte auf diesem Bus von der RCU ignoriert werden.

Listing 3.2: Hardware-Teil der Slave-Mode Anwendung

```
module user(
  CLK,      // Systemtakt
  RESET,   // systemweiter Reset
  ADDRESSED, // High, wenn RC von CPU angesprochen wird
  WRITE,   // High, wenn CPU auf RC schreiben will
  DATAIN, // Von der CPU auf die RC geschriebene Daten
  DATAOUT, // Von der CPU aus der RC gelesene Daten
  ADDRESS // Adresse des Zugriffs (in dieser Anwendung ignoriert)
);
// Eingänge
input      CLK;
```

input	RESET;	13	
input	ADDRESSED;	14	
input	WRITE;	15	
input [31:0]	DATAIN;	16	
input [23:2]	ADDRESS;	17	
<i>// Ausgänge</i>			19
output [31:0]	DATAOUT;	20	
<i>// Beginn der Anwendung *****</i>			22
reg [31:0]	result;	<i>// Ergebnisregister</i>	24
reg [31:0]	reversed;	<i>// Zwischenergebnis</i>	25
<i>// Gebe immer (unabhängig von der Adresse) das Ergebnisregister aus</i>			27
assign	DATAOUT = result;		28
<i>// Berechne als Zwischenergebnis immer die</i>			30
<i>// bitverdrehte Reihenfolges des Dateneingangs</i>			31
<i>// Beachte: Dies ist ein kombinatorischer Block!</i>			32
always	@(DATAIN) begin : comb.block		33
integer	n;		34
for	(n=0; n < 32; n = n + 1) begin		35
reversed[n]	= DATAIN[31-n];		36
end			37
end			38
<i>// Steuerung</i>			40
always	@(posedge CLK or posedge RESET) begin		41
<i>// Initialisiere Ergebnis auf magic number für Debugging</i>			42
if	(RESET) begin		43
result	<= 32'hDEADBEEF;		44
<i>// Schreibzugriff auf RC, neu berechnetes Zwischenergebnis übernehmen</i>			45
end else if	(ADDRESSED & WRITE) begin		46
result	<= reversed;		47
end			48
end			49
endmodule			51

Architektur der Recheneinheit

Wir wählen folgende Architektur für diese RCU: Ein einzelnes Register `result` speichert das letzte Berechnungsergebnis. Dieses entsteht dadurch, dass ein von der CPU auf die RCU geschriebenes Datenwort sofort bitweise verdreht wird (dieser Wert liegt als `reversed` vor) und noch im selben Takt in `result` gespeichert wird. Diese Vorgehensweise ist möglich, da die `reversed` berechnende Logik sehr schnell ist (nur eine Permutation von Leitungen, keine aktiven Gatter) und der gesamte Vorgang in einem Taktzyklus durchlaufen werden kann.

Da wir nur ein einzelnes für die CPU sichtbares Register haben, kann auf die Dekodierung von `ADDRESS` verzichtet werden, wir geben einfach immer unser Ergebnisregister `result` an den `DATAOUT`-Bus aus (Zeile 28). Jeder Lesezugriff von der CPU erhält so immer das aktuelle Berechnungsergebnis.

Der rein kombinatorische Block in Zeile 33–38 berechnet immer aus den Eingabedaten auf dem `DATAIN`-Bus ein bitweise verdrehtes Zwischenergebnis auf seinem `reversed`-Ausgang (in Hardware wird hierfür kein Register synthetisiert werden). Man beachte, dass diese Berechnung auch außerhalb eines Schreibzugriffs, und damit auch auf ungültigen Eingabedaten, stattfindet. Wie wir gleich sehen werden, stört dies aber nicht weiter.

Der letzte Block in Zeile 41–49 steuert die gesamte Hardware-Anwendung. Die Reset-Behandlung in Zeile 43–44 scheint auf den ersten Blick etwas ungewöhnlich, da das `result`-Register nicht auf den üblichen Wert Null initialisiert wird, sondern stattdessen auf die eigenartige Zahl 3.735.928.559. Ein Grund dafür ist, dass dieser Wert auf den üblichen hexadezimalen Speicherausgängen sehr leicht wieder zu erkennen ist: DEAD-BEEF. Man kann also als ersten Hardware-Test einen Lesezugriff von der CPU auf die RCU ausführen. Wenn die erwartete "magic number" zurückgeliefert wird, so ist schon einmal die erfolgreiche Konfiguration der RCU, der abgeschlossene Reset und die Funktion einfacher Lesezugriff im Slave-Mode sichergestellt.

In Zeile 46–47 wird schließlich garantiert, dass nur bei einem Schreibzugriff von der CPU auf die RCU, wenn also auf `DATAIN` wirklich gültige Daten anliegen, das in `reversed` berechnete Zwischenergebnis tatsächlich in das Ergebnisregister `result` übernommen wird. In allen anderen Fällen bleibt `result` unverändert.

Die Timing-Analyse mittels des Xilinx Werkzeuges `trce` zeigt, dass diese Hardware-Konfiguration nach Synthese, Platzierung und Verdrahtung die gewünschte Taktfrequenz von 100 MHz problemlos erreicht. Auch die Größe der synthetisierten Logik ist überschaubar: Mit 40 LUTs wird weniger als 1% der auf dem FPGA zur Verfügung stehenden 27392 LUTs ausgenutzt. Allerdings sind auf dem Chip ja neben der eigentlichen RCU (dem Inhalt des Verilog-Moduls `user`) auch noch diverse andere Komponenten (Speicher-Controller, verschiedene Bus-Bridges, etc.) untergebracht. Alles zusammen benötigt 8617 LUTs, in dieser Hinsicht ist der ganze Chip also zu gut 31% gefüllt.

Software-Schnittstelle zur RCU

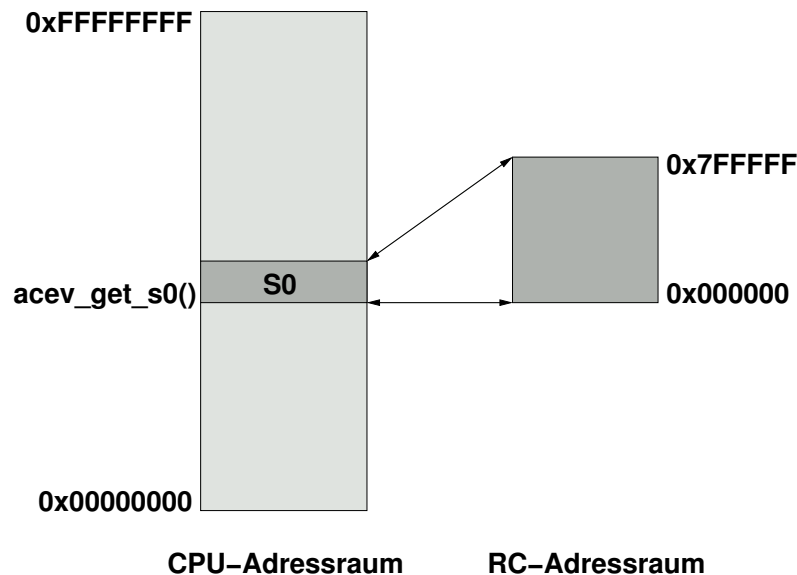


Abbildung 3.10: Speicheraufteilung im Slave-Mode

Die Kommunikation zwischen Soft- und Hardware beruht darauf, dass die RCU-internen Register (auf der ML310 auch *S0-Bereich* genannt), die in der Hardware über die Signale `ADDRESSED`, `ADDRESS`, `DATAIN`, `DATAOUT` und `WRITE` angesprochen werden, in den Adressraum der CPU eingeblendet werden. Diese Struktur ist in Abbildung 3.10 dargestellt. Dabei gilt folgendes:

- Ein CPU-Zugriff auf eine Adresse im eingeblendeten S0-Bereich wird nicht auf den Hauptspeicher ausgeführt, sondern wird auf die RCU umgeleitet. Dabei wird die Differenz zwischen Zugriffsadresse und S0-Basisadresse auf den `ADDRESS`-Bus der RCU angelegt.
- Die RCU-Adressen adressieren stets 32b Datenworte (also in 4B Schritten). Das heißt, dass die niederwertigsten beiden Bits immer Null sein müssen. Auf der RCU werden sie einfach ignoriert (`ADDRESS` ist als (23:2) definiert). Bei Software-Zeigern auf der CPU kann durch die Verwendung geeigneter Typen (Zeiger auf 32b long Worte) ein ähnlicher Effekt erreicht werden.

Beispiel: Nehmen wir an, dass der S0-Bereich bei der CPU-Adresse 32'h10000000 beginnt (dieser Wert kann durch die Software zur Laufzeit abgefragt werden). Ein Lesezugriff der CPU auf Adresse 32'h10004700 führt zu einem Lesezugriff auf die RCU-Adresse

$$32'h10004700 - 32'h10000000 = 24'h004700$$

Dieser Wert wird zusammen mit ADDRESSED=1 und WRITE=0 auf dem ADDRESS-Bus auftauchen.

Software-Teil der Slave-Mode Anwendung

Die bis hier vorgestellte Hardware-Komponente ist aber nur die eine Hälfte der Gesamtanwendung. Schließlich muß auf der CPU auch noch ein Programm laufen, dass die RCU mit Daten füttert und die Ergebnisse abholt. Dieses ist in Listing 3.3 gezeigt.

Listing 3.3: Software-Teil der Slave-Mode Anwendung

```

#include <stdio.h>
#include <stdlib.h>
#include "rcu.h"

// Anzahl Datensätze in Ein- und Ausgabedatei
#define NUM_WORDS (1024*256)

main()
{
    // Ein- und Ausgabedateien
    FILE *infile , *outfile ;

    // Benutze vorzeichenlose Ganzzahlen (32b Worte) für alle Daten
    unsigned long *inwords, *outwords;
    unsigned long m, inword, outword;

    // Marker für Zeitmessung (64b Worte)
    unsigned long long start, stop, rcu_get_ticks ();

    // Zeiger auf RC-Adressraum
    volatile unsigned long *rcu;

    // fordere Speicher für Ein- und Ausgabefelder an
    inwords = malloc(NUM_WORDS * sizeof(unsigned long));
    outwords = malloc(NUM_WORDS * sizeof(unsigned long));

    // Öffne die Dateien zum Lesen und Schreiben
    infile = fopen("test1.in", "r");
    outfile = fopen("test1.out", "w");

    // Lese komplette Eingabedatei in Eingabe-Speicherbereich
    fread(inwords, sizeof(unsigned long), NUM_WORDS, infile);

    // RC initialisieren
    rcu_init ();
    // Zeiger auf RC-Adressraum holen
    rcu = rcu_get_s0(NULL);

    // Merke Startzeit der Berechnung
    start = rcu_get_ticks ();

```

```

// Bearbeite Daten
for (m=0; m < NUM_WORDS; ++m) {
    // Übertrage das Eingabedatenwort an die RCU
    rcu[0] = inwords[m];
    // Hole das Ergebnis von der RCU und trage es in das Ausgabefeld ein
    outwords[m] = rcu[0];
}

// merke Stopzeit
stop = rcu_get_ticks ();

// Schreibe das komplette Ausgabefeld in die Ausgabedatei
fwrite(outwords, sizeof(unsigned long), NUM_WORDS, outfile);

// Gebe Speicher für Ein-/Ausgabefelder wieder frei
free(inwords);
free(outwords);

// Schließe Dateien
fclose( infile );
fclose( outfile );

// Gebe Ergebnis der Zeitmessung in Mikrosekunden aus
printf("Zeit: %lldµs\n", (stop-start)/TICKS_PER_USEC);
}

```

In weiten Teilen ist dieses Programm identisch zur reinen Software-Lösung (Listing 3.1). Die wesentlichen Unterschiede liegen in der Initialisierung der RCU und dem Ersetzen der eigentlichen Berechnung durch Kommunikation mit der RCU.

Betrachten wir die Änderungen im Einzelnen. In Zeile 24 wird eine neue Variable `rcu` als Zeiger auf ein 32b Wort definiert. Über diese Variable, genauer gesagt: dem Ziel dieses Zeigers, wird später die Kommunikation mit der RCU ablaufen. Das Attribut `volatile` ist hier sehr wichtig: Es gibt dem C-Compiler zu verstehen, dass sich die Zieldaten des Zeigers auch ohne Intervention der CPU ändern können (sie werden ja auch von der Hardware in der RCU manipuliert). Auch "sinnlos" erscheinende Software-Zugriffe dürfen deshalb nicht durch den Compiler wegoptimiert werden.

Die nächsten Anweisungen für Speicherallozierung, Öffnen der Dateien und Einlesen der Eingabedaten unterscheiden sich nicht von der reinen Softwarelösung.

Der Block in Zeile 37–40 ist aber neu. Zeile 38 initialisiert die RCU. Zeile 40 richtet die Kommunikation zwischen CPU und RCU ein.

Hier wird von der Einblendung der RC-Register in den CPU-Adressbereich (memory-mapped I/O) Gebrauch gemacht. Wir fragen dazu in Zeile 40 den Beginn des S0-Bereiches ab und weisen ihn an die Zeigervariable `rcu` zu. Diese wird als Basis für die Adressierung von RCU-Registern dienen. Da `rcu` als Feld (Array) von 32b Worten angesehen werden kann, werden durch einfache Indizierung von `rcu` aus die korrekten Register-Adressen in 4B Schritten erzeugt.

Bei unserer Beispielanwendung verwenden wir nur ein einzelnes RCU-Register (`result`), das unabhängig von der aktuellen Adresse im gesamten RCU-Adressraum anliegt. Wir werden es willkürlich als Register 0 von der S0-Basis `rcu` aus mit dem Ausdruck `rcu[0]` adressieren.

Die Kommunikation der Software mit der RCU zur Übergabe von Ein- und Ausgabedaten ist im Schleifenblock in Zeile 45–51 implementiert: Hier wird, wie schon in der reinen Software-Lösung, der Eingabespeicherbereich elementweise durchgegangen. Statt der eigentlichen Berechnung (der inneren Schleife in Listing 3.1) ist aber die RCU eingebunden: Jedes Eingabedatenwort wird in Register 0 der RCU geschrieben (Zeile 48). Die Anweisung löst einen Schreibzugriff (`WRITE=1`) auf die RCU aus, wobei das Eingabedatenwort auf dem DATAIN-Bus der RCU erscheint. Der kombinatorische Block berechnet sofort das entsprechende bitweise verdrehte Wort (Zeile 33–38 in Listing 3.2), das in Zeile 47 in Listing 3.2 als Endergebnis in das Ausgaberegister `result` übernommen wird. Das Ausgaberegister wird im Software-Teil der Anwen-

dung dann in Zeile 50 (Listing 3.3) via dem DATAOUT-Bus, an dem es immer anliegt, ausgelesen und in den Ausgabe-Speicherbereich geschrieben.

An dieser Stelle zeigt sich auch die Bedeutung des `volatile` Attributs für die Variable `rcu` (Zeile 24). Ohne dieses Attribut würde der Compiler die Anweisungsfolge

```
rcu[0] = inwords[m];
outwords[m] = rcu[0];
```

direkt in

```
outwords[m] = inwords[m];
```

umformen. Dabei würde die RCU gar nicht mehr angesprochen und stattdessen (fehlerhafterweise) die Eingabedaten direkt in den Ausgabedatenbereich kopiert. Durch `volatile` wird der Compiler von dieser Optimierung abgehalten. Die folgenden Anweisungen des Programms unterscheiden sich nicht mehr von der reinen Software-Lösung.

Wie schnell läuft nun die Berechnung auf dieser kombinierten Hardware-Software-Architektur? Die Bearbeitung von 256Kw braucht hier nur noch **32045µs**, also rund 0.03s (statt 0.2s bei der reinen Software-Lösung). Die mit 100 MHz getaktete RCU ist damit fast doppelt so schnell, wie die mit 300 MHz getaktete CPU.

Aber wie effizient ist diese Lösung? Eigentlich sollte ja pro Takt ein Datum verarbeitet werden können. Aber dafür ist die gemessene Zeit für die Berechnung viel zu lang. Weitergehende Untersuchungen zeigen, dass zwischen den einzelnen Schreib- und Lesezugriffen auf den Hardware-Teil nennenswerte Zeit vergeht: So war die kürzeste gemessene Zeit zwischen zwei solchen Zugriffen 3 RCU-Takte lang, die längste Pause dauerte gar über 100 RCU-Takte. In diesen Intervallen liegt die Hardware brach, sie führt keine sinnvollen Berechnungen aus. Der Grund dafür ist einerseits in den Schwächen des Busses zwischen CPU und RCU zu suchen (relativ lange Latenzen bei einzelnen Transfers), andererseits im Multi-Tasking-Verhalten von Linux, bei dem unsere Software die CPU mit anderen Programmen teilen muß.

3.3.3 Weitere Beschleunigung durch RCU im Master-Mode

Wie kann man nun die Effizienz der Hardware steigern? Der Kommunikationskanal zwischen CPU und RCU, der sogenannte Processor Local Bus (PLB) läßt sich offensichtlich nicht umgehen. Er läßt sich aber besser nutzen: Bei den kleinen Häppchen, die die CPU im Slave-Mode in abwechselnden Richtungen mit der RCU austauscht (ein Datenwort lesen, ein Datenwort schreiben) summieren sich sehr schnell die Latenzen, die beim PCI-Protokoll überhaupt für den Aufbau einer Verbindung gebraucht werden: Auf der ML310 wurden für das Schreiben eines 32b-Wortes auf die RCU 3 RCU-Takte gemessen, beim Lesen eines Wortes sogar 5 RCU-Takte. Wie die meisten moderneren Busse unterstützt aber auch PLB sogenannte *Burst-Transfers*. Dabei fällt der administrative Aufwand für den Verbindungsaufbau zwar immer noch an, aber diesmal wird mehr als ein Wort je Verbindung übertragen. Gerade bei größeren Datenmengen läßt sich so die pro Zeit übertragbare Menge an Nutzdaten deutlich steigern.

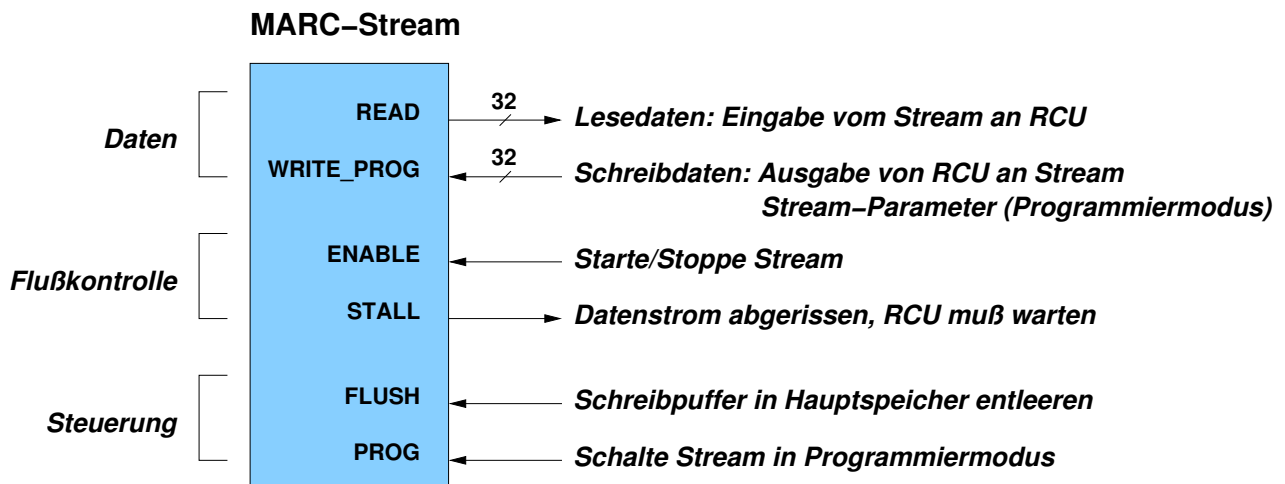
Durch einfache C-Anweisungen sind solche Burst-Transfers software-seitig aber nicht auslösbar. Es gibt zwar die Möglichkeit, auf der CPU mittels geeigneter Programmierung eine eigene dort integrierte Hardware-Einheit mit dem Transfer zu betrauen (Direct Memory Access, DMA), und so einen Burst-Transfer zu erzwingen. Als weitere Verfeinerung unserer aktuellen Anwendung werden wir aber eine vielseitigere Methode verwenden: Die RCU wird nun *eigenständig* alle Speicherzugriffe auf den Hauptspeicher durchführen (Master-Mode). Dabei kann sie bequem die entsprechenden Burst-Transfers generieren und auch Zugriffsmuster ausführen, die für die üblichen DMA-Einheiten zu komplex sind. Letzteres wird für unser kleines Beispiel zwar nicht gebraucht, die dafür benötigten Verfahren sind aber analog zu den unten beschriebenen.

Master-Mode Speicherschnittstelle

In der Praxis erweist sich die Hardware-Realisierung von Master-Mode Zugriffen leider als trickreich. Viele Eigenheiten, z.B. versteckte Einschränkungen der Burst-Länge, unglückliche Timing-Abhängigkeiten und

ähnliches, treten erst bei praktischen Versuchen mit dem gesamten System auf, scheinbar völlig losgelöst von der heilen Welt der Datenblätter und Spezifikationen. Es bietet sich daher an, den Aufwand für die RCU-Implementierung von funktionierenden Master-Mode Zugriffen nur einmal zu betreiben. Der entsprechende Hardware-Block muß dabei so flexibel ausgelegt sein, dass er daraufhin in den unterschiedlichsten Szenarien ohne größere Anpassungen wiederverwendet werden kann.

Zu diesem Zweck wurde von den Mitarbeitern der FG ESA die Memory Architecture for Reconfigurable Computers (MARC) entwickelt. Es handelt sich dabei um ein flexibles (in Bezug auf die RCU-Anwendung) und portables (in Bezug auf die ACS-Hardware) Speicherzugriffssystem. Die RCU-Anwendung wird völlig von den Eigenheiten der ACS-Hardware isoliert. Stattdessen verwendet sie abstrakte Schnittstellen, die ihr verschiedene Datenzugriffsdienste zur Verfügung stellen. Für unsere Anwendung sind dabei die sogenannten Streams interessant, die längere Datenströme über zusammenhängenden Speicherbereiche realisieren.



Reihenfolge der Parameter im Programmiermodus

- Startadresse**
- Anzahl Datensätze**
- Schrittweite**
- Breite der Zugriffe (8b/16b/32b)**
- Zugriffsart (Lesen/Schreiben)**

Abbildung 3.11: RCU-seitige Schnittstelle eines MARC-Streams

Abbildung 3.11 zeigt dabei die Schnittstelle eines MARC-Streams zur benutzerdefinierten Hardware in der RCU. Die sechs Ports sind dabei grob in drei Gruppen einteilbar: Daten-Ports erlauben den Datenfluß aus dem Stream hinaus in die Benutzer-Hardware (via READ) oder aus der Benutzer-Hardware in den Stream hinein (via WRITE_PROG). Zur Kontrolle des Datenflusses stehen zwei weitere Ports bereit. Über ENABLE kann die Benutzer-Hardware den Datenstrom anhalten, während ihr über STALL durch MARC ein Abriss im Datenstrom angezeigt wird. Letztlich werden noch zwei Steuereingänge benötigt. Durch FLUSH wird schreibenden Streams das Ende der RCU-Operation angezeigt und so eventuell noch auf der RCU lokal gepufferte Daten eines Schreib-Streams tatsächlich in den Hauptspeicher geschrieben. Mit dem PROG Signal kann die Benutzerschaltung den Stream anweisen, die nun auf WRITE_PROG anliegenden Daten nicht in den Hauptspeicher zu schreiben, sondern als Parameter in die internen Steuerregister des Streams zu übernehmen. Dabei ist die ebenfalls in Abbildung 3.11 gezeigte Reihenfolge (ein Parameter pro positiver Taktflanke) einzuhalten. Wenn diese Programmiersequenz vorzeitig beendet wird (durch Wegnehmen des PROG Signals), bleiben alle noch nicht geschriebenen Parameter unverändert.

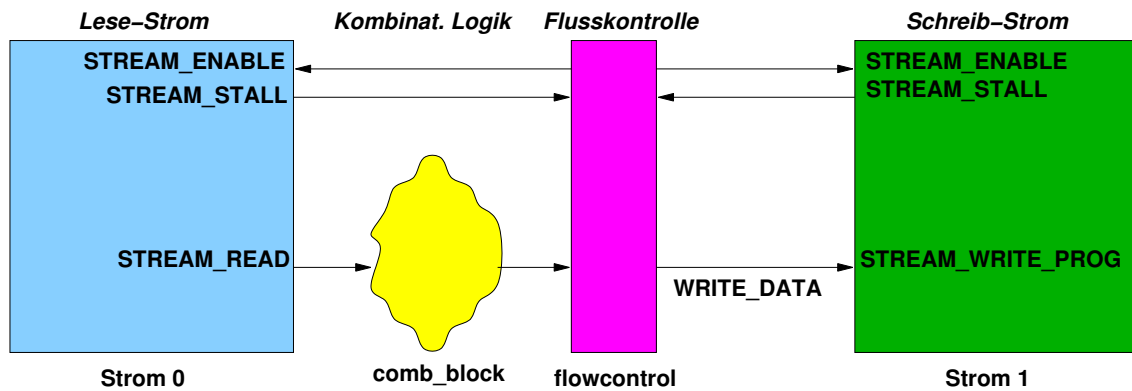


Abbildung 3.12: Architektur der Master-Mode Hardware

Architektur der Master-Mode Hardware

An der eigentlichen Form der Berechnung des bitweise verdrehten Ausgabeworts aus den Eingabedaten ändert sich auch bei diesem neuen Ansatz nichts. Wohl aber an der Art und Weise, wie die Eingabedaten entgegengenommen und die Ausgabedaten abgelegt werden.

Das Konstrukt des einfachen result Registers, das das Ergebnis der direkt durch den DATAIN-Bus gespeisten kombinatorischen Logik übernimmt, ist hier durch zwei gekoppelte Lese- und Schreib-Streams ersetzt, zwischen denen die kombinatorische Logik liegt. Die durch den Lese-Stream (Stream 0) eintreffenden Eingangsdaten werden also durch die kombinatorische Logik transformiert und durch den Schreib-Stream (Stream 1) wieder zurück in den Hauptspeicher geleitet (Abbildung 3.12).

Um dieses Konzept tatsächlich umsetzen zu können sind vier wesentliche Funktionen zu realisieren:

1. Die Entgegennahme von Parametern durch die CPU. Dazu gehören beispielsweise die Adressen der Speicherbereiche für die Ein-/ Ausgabedaten im Hauptspeicher.
2. Die entsprechende Programmierung der Stream-Parameter.
3. Die Flußkontrolle zwischen den beiden Streams. So muß bei Abriss des Eingabedatenstromes auch der Ausgabedatenstrom angehalten werden. Umgekehrt muß bei einer 'Verstopfung' des Ausgabedatenstromes auch die Eingabe angehalten werden (anderenfalls würde Daten unbearbeitet verloren gehen).
4. Der CPU, die jetzt ja die Berechnung lediglich startet, muß irgendwie deren Ende angezeigt werden.

Listing 3.4 zeigt das Verilog-Modell der Master-Mode Lösung. In den folgenden Abschnitten werden die wesentlichen Teile vorgestellt.

Listing 3.4: Hardware-Teil der Master-Mode Anwendung

```

'include "marcdefs.v"
1
2
module user (
3
    // *** Globale Signale
4
    CLK,           // Takt
5
    RESET,        // Systemweiter Reset
6
7
    // *** Slave - Schnittstelle
8
    ADDRESSED,    // RC angesprochen im Slave-Mode?
9
    WRITE,        // Schreibzugriff ?
10
    DATAIN,     // Dateneingang
11
    DATAOUT,    // Datenausgang
12
    ADDRESS,     // Adresseingang
13

```

```

    IRQ,                // Löst Interrupt (IRQ) an CPU aus
                        // *** Schnittstelle für MARC-Streams
    STREAM_READ,       // Read-Datenbus
    STREAM_WRITE_PROG, // Write-Programm-Datenbus
    STREAM_STALL,      // Stall-Signale
    STREAM_ENABLE,     // Start/Stop für Streams
    STREAM_FLUSH,      // Schreib-Streams entleeren
    STREAM_PROG        // Programmiermodus einschalten
);

// Schnittstellendeklaration *****

// Eingänge
input          CLK;
input          RESET;
input          ADDRESSED;
input          WRITE;
input [31:0]   DATAIN;
input [23:2]   ADDRESS;
input ['STREAM_DATA_BUS] STREAM_READ;
input ['STREAM_CNTL_BUS] STREAM_STALL;

// Ausgänge
output [31:0]  DATAOUT;
output ['STREAM_DATA_BUS] STREAM_WRITE_PROG;
output ['STREAM_CNTL_BUS] STREAM_ENABLE;
output ['STREAM_CNTL_BUS] STREAM_FLUSH;
output ['STREAM_CNTL_BUS] STREAM_PROG;
output          IRQ;

// Deklarationen für Stream-Schnittstelle
wire ['STREAM_DATA_BUS] STREAM_READ;
wire ['STREAM_DATA_BUS] STREAM_WRITE_PROG;
wire ['STREAM_CNTL_BUS] STREAM_STALL;
wire ['STREAM_CNTL_BUS] STREAM_ENABLE;
reg  ['STREAM_CNTL_BUS] STREAM_FLUSH;
reg  ['STREAM_CNTL_BUS] STREAM_PROG;

// Konstantendefinitionen *****

// Zustände der zentralen Controller-FSM
`define STATE_PROG_START 0 // Programmierere Startadressen in Streams
`define STATE_PROG_COUNT 1 // Programmierere Datensatzzahl in Streams
`define STATE_PROG_STEP 2 // Programmierere Schrittweite in Streams
`define STATE_PROG_WIDTH 3 // Programmierere Zugriffsbreite in Streams
`define STATE_PROG_MODE 4 // Programmierere Betriebsart in Streams
`define STATE_COMPUTE 5 // Führe Berechnung auf Streamdaten aus
`define STATE_SHUTDOWN 6 // Beende Berechnung

// Beginn der Anwendung *****

// Wurde Anwendung gestartet?
reg          START;
// Anfangsadresse der Eingabedaten im Hauptspeicher
reg [31:0]  SOURCEADDR;
// Anfangsadresse der Ausgabedaten im Hauptspeicher
reg [31:0]  DESTADDR;
// Länge des Datensatzes in 32b Worten
reg [31:0]  COUNT;
// Soll ein Interrupt ausgelöst werden?

```

```

reg      IRQSTATE;
// Aktueller Zustand der Anwendung
reg [4:0] STATE;
// Bearbeitetes ( verdrehtes ) Eingabedatenwort
reg [31:0] REVERSED;
// Sind die Streams gestartet ?
reg      STREAMSTART;
// Programmierdaten – Register für Streams
reg [31:0] STREAM_PROGDATA_0;
reg [31:0] STREAM_PROGDATA_1;

// Daten zur Ausgabe an Schreib –Stream
wire [31:0] WRITE_DATA;

// Abkürzung für Registernummer 0 ... 15
wire [3:0] REGNUM = ADDRESS[5:2];

// Streams laufen , nachdem sie gestartet worden sind und solange
// noch Daten zu bearbeiten sind .
wire RUNNING = STREAMSTART & (COUNT != 0);

// Flußkontrolle zwischen Ein– und Ausgabe–Streams
flowcontrol FC (
    CLK,           // Takt
    RUNNING,      // Streams laufen lassen ?
    STREAM_STALL[0], // Hängt Stream 0 (Eingabe–Stream)?
    STREAM_STALL[1], // Hängt Stream 1 (Ausgabe–Stream)?
    REVERSED,     // Von Anwendung zu schreibende Daten
    STREAM_ENABLE[0], // Stream 0 starten oder anhalten
    STREAM_ENABLE[1], // Stream 1 starten oder anhalten
    WRITE_DATA    // Eingangsdaten für Ausgabe–Stream
);

// Gebe IRQSTATE Register an CPU IRQ–Leitung aus
assign IRQ = IRQSTATE;

// Gebe immer das gerade adressierte Register aus .
// Nicht benötigte Register geben eine Magic–Number
// und den aktuellen IRQ–Status im MSB zurück
wire [31:0] DATAOUT =
    (REGNUM == 4'h0) ? SOURCEADDR
    : (REGNUM == 4'h1) ? DESTADDR
    : (REGNUM == 4'h2) ? COUNT
    : (32'h00C0FFEE | (IRQSTATE << 31));

// Schalte Streams zwischen Programmier– und Datenbetrieb um
// Stream0 ist Lese–Stream, sein Eingang kann immer im Programmierbetrieb sein
assign STREAM_WRITE_PROG['STREAM_0] = STREAM_PROGDATA_0;

// Stream1 ist Schreib –Stream, hier muß der Eingang umgeschaltet werden
assign STREAM_WRITE_PROG['STREAM_1] =
    (STREAM_PROG[1])
    ? STREAM_PROGDATA_1
    : WRITE_DATA;

// Berechne als Zwischenergebnis immer die
// bitverdrehte Reihenfolges das Lese–Datenstromes 0
// Beachte : Dies ist ein kombinatorischer Block !
always @(STREAM_READ[31:0]) begin: comb_block
    integer n;
    for (n=0; n < 32; n = n + 1) begin

```

```

    REVERSED[n] = STREAM_READ[31-n];
end
end

// Controller FSM überwacht gesamte Anwendung
always @(posedge CLK or posedge RESET) begin
    // Initialisiere Register bei chip-weitem Reset
    if (RESET) begin
        STATE          <= 'STATE_PROG_START;
        IRQSTATE       <= 0;
        STREAM_START   <= 0; STREAM_PROG    <= 0;
        STREAM_FLUSH   <= 0; STREAM_PROGDATA_0 <= 0;
        STREAM_PROGDATA_1 <= 0; SOURCEADDR <= 0;
        DESTADDR       <= 0; COUNT        <= 0;
        START          <= 0;
        // Schreibzugriff auf RC, schreibe in entsprechendes Register
    end else if (ADDRESSED & WRITE) begin
        case (REGNUM)
            0: SOURCEADDR <= DATAIN;
            1: DESTADDR  <= DATAIN;
            2: COUNT    <= DATAIN;
            3: begin
                START    <= 1; // Startkommando, beginne Ausführung
            end
            default: ;
        endcase
    end else begin
        // CPU hat Berechnung gestartet, keine Slave-Mode Zugriffe mehr möglich
        if (START) begin
            case (STATE)
                'STATE_PROG_START:
                    begin
                        // Beide Streams in Programmiersmodus schalten
                        STREAM_PROG[1:0] <= 2'b11;
                        // Anfangsadresse für Stream 0 schreiben
                        STREAM_PROGDATA_0 <= SOURCEADDR;
                        // Anfangsadresse für Stream 1 schreiben
                        STREAM_PROGDATA_1 <= DESTADDR;
                        // FSM weitersetzen
                        STATE <= 'STATE_PROG_COUNT;
                    end
                'STATE_PROG_COUNT:
                    begin
                        // Anzahl Datensätze - 1 eintragen (bei beiden Streams gleich)
                        STREAM_PROGDATA_0 <= COUNT - 1;
                        STREAM_PROGDATA_1 <= COUNT - 1;
                        // FSM weitersetzen
                        STATE <= 'STATE_PROG_STEP;
                    end
                'STATE_PROG_STEP:
                    begin
                        // Schrittweite: 1 Datensatz (bei beiden Streams gleich)
                        STREAM_PROGDATA_0 <= 1;
                        STREAM_PROGDATA_1 <= 1;
                        // FSM weitersetzen
                        STATE <= 'STATE_PROG_WIDTH;
                    end
                'STATE_PROG_WIDTH:
                    begin
                        // Breite der Zugriffe: 32b (bei beiden Streams gleich)
                        STREAM_PROGDATA_0 <= 'STREAM_32B;
                    end
            endcase
        end
    end
end

```

```

STREAM_PROGDATA_1 <= 'STREAM_32B;
// FSM weitersetzen
STATE <= 'STATE_PROG_MODE;
end
'STATE_PROG_MODE:
begin
// Zugriffsart für Stream 0: Lesen
STREAM_PROGDATA_0 <= 'STREAM_READ;
// Zugriffsart für Stream 1: Schreiben
STREAM_PROGDATA_1 <= 'STREAM_WRITE;
// FSM weitersetzen
STATE <= 'STATE_COMPUTE;
end
'STATE_COMPUTE:
begin
// Programmiermodus für beide Streams abschalten
STREAM_PROG[1:0] <= 0;
// Beide Streams starten (via flowcontrol –Modul)
STREAMSTART <= 1;

// Alle Datensätze bearbeitet ?
if (COUNT == 0) begin
// Dann beide Streams stoppen
STREAMSTART <= 0;
// Falls Schreib –Stream fertig
if (!STREAM_STALL[1]) begin
// alle noch gepufferten Daten wirklich schreiben
STREAM_FLUSH[1] <= 1;
// FSM weitersetzen
STATE <= 'STATE_SHUTDOWN;
end
end else if (STREAM_ENABLE[0] & ~STREAM_STALL[0])
// Nur dann einen Datensatz als bearbeitet zählen ,
// wenn Stream 0 aktiv liest (ENABLE) und nicht hängt (!STALL)
COUNT <= COUNT – 1;
end
'STATE_SHUTDOWN:
begin
// Ist Schreibpuffer schon komplett geleert ?
if (!STREAM_STALL[1]) begin
// ja , Leerung beenden
STREAM_FLUSH[1] <= 0;
// CPU durch IRQ Fertigwerden der RC anzeigen
IRQSTATE <= 1;
// FSM stoppen (RC jetzt wieder im Slave –Mode)
START <= 0;
// FSM auf Startzustand zurücksetzen
STATE <= 'STATE_PROG_START;
end
end
// Dieser Fall sollte nicht auftreten , nur für Logikoptimierung
default : STATE <= 'bx;
endcase
end
// Bei jedem Lese –Zugriff auf RC im Slave –Mode, vorhandenen IRQ ausschalten
else if (ADDRESSED)
IRQSTATE <= 0;
end
end
endmodule

```

Hardware-Schnittstelle der RCU

Auch die Master-Mode Hardware benötigt eine vollständige Slave-Mode Schnittstelle. Auf diesem Weg übergibt die CPU die aktuellen Parameter (CPU-Adressen der Speicherbereiche für Eingangs- und Ausgangsdaten, Anzahl der Datensätze, ein Startkommando). Die schon bekannten Signale werden in den Zeilen 9–13 des Modulkopfes deklariert. Neu hinzugekommen ist das Signal `IRQ`. Wenn es von der RCU auf High gelegt wird, löst es auf der CPU eine Unterbrechung der normalen Programmausführung aus, einen sogenannten *Interrupt*. Die Ausführung der CPU-Befehle verzweigt stattdessen in eine vorher dafür deklarierte Funktion, die den Interrupt bearbeitet. Wenn das Ende dieses *Interrupt-Handlers* erreicht wird, wird die Programmausführung an der Stelle vor dem Auftreten des Interrupts wieder fortgesetzt.

Die Zeilen 17–22 deklarieren die Schnittstelle zu den MARC-Streams. Dabei ist zu beachten, dass beide Streams in denselben Bussen, aber auf unterschiedlichen Bit-Intervallen geführt werden. So liegt der `WRITE_PROG`-Bus von Stream 0 im Intervall `WRITE_PROG('STREAM_0)`, während er für Stream 1 auf `WRITE_PROG('STREAM_1)` liegt. Die Kontrollsignale werden analog gehandhabt. Die Signale `STREAM_ENABLE(0)` und `STREAM_ENABLE(1)` entsprechen so den `ENABLE`-Ports für Stream 0 und Stream (respektive).

Interner Aufbau der Hardware

In den Zeilen 56–62 werden einige symbolische Konstanten für den Zustandsautomaten der zentralen Steuerung definiert. Wie man hier schon erahnen kann, ist dieser Controller deutlich aufwendiger als in der Slave-Mode Hardware.

Die Zeilen 66–73 deklarieren die Register, die die von der CPU übergebenen aktuellen Parameter enthalten. Das `IRQSTATE`-Register kann verwendet werden, einen Interrupt auf der CPU auszulösen (Interrupt Request). `STATE` gibt den aktuellen Zustand unseres Steuerungsautomaten an. `REVERSED` ist das bekannte Ergebnis (bitweise verdrehtes Eingabewort) des kombinatorischen Blockes. `STREAMSTART` wird von der zentralen Steuerung gesetzt, wenn alle Stream-Parameter komplett programmiert sind und die Datenströme nun anfangen können zu fließen. Die Programmierdaten für die beiden Streams werden in den beiden Registern der Zeilen 83–84 gehalten. In Zeile 90 führen wir eine Kurzschreibweise für die untersten 4b des RCU `ADDRESS`-Busses ein. Diese werden als Registernummern von 0 ... 15 interpretiert.

In Zeile 94 beginnt schließlich der aktive Teil der Anwendung. `RUNNING` ist gesetzt, wenn die Streams gestartet wurden, und noch Daten zu bearbeiten sind. Letzteres stellen wir mit dem Zähler `COUNT` fest, der die Anzahl der noch zu lesenden Daten zählt. In den Zeilen 97–106 werden die beiden Streams (wie bereits in Abbildung 3.12 gezeigt) durch ein Flußkontroll-Modul gekoppelt. Dieses (hier nicht weiter gezeigte) Modul erfüllt die in Abschnitt 3.3.3 definierten Anforderungen. Neben den `STALL` und `ENABLE`-Signalen der gekoppelten Streams muß es auch noch an die Eingangsdaten (von der kombinatorischen Logik) und die Ausgangsdaten (an den Schreib-Stream) angeschlossen werden.

In Zeile 109 wird das interne Interrupt-Statusregister `IRQSTATE` an die Interrupt-Leitung der CPU angeschlossen. Der Block in Zeile 114–118 ist eine Erweiterung der bekannten Implementierung von Slave-Mode Lesezugriffen. Als Dienstleistung bieten wir hier der CPU an, die aktuellen Werte der RCU-Parameter auszulesen. Da wir nun mehr als ein Register zur Verfügung stellen, muß hier die Adresse des Zugriffs tatsächlich ausgewertet werden. Dies geschieht mit dem Umweg über die in Zeile 90 definierte Registernummer `REGNUM`. Für die Registernummern 0...2 geben wird das entsprechende interne Register auf den `DATAOUT`-Bus aus. In allen anderen Fällen (derzeit nicht benötigte Registernummern) wird für das Debugging eine weitere gute erkennbare "magic number" definiert, in die zusätzlich noch der aktuelle Zustand des internen Interrupt-Registers im MSB eingeblendet wird. Auch dies dient der Erleichterung beim Hardware-Debugging.

In den Zeilen 122–128 werden die Schreib-/Programmiereingänge der Streams verdrahtet. Im Fall des Lese-Streams (Stream 0) wird der Port lediglich zur Programmierung verwendet und kann somit immer an das Programmierdatenregister `STREAM_PROGDATA_0` angeschlossen werden. Beim Schreib-Stream (Stream 1), der einen echten Datenstrom bearbeitet, muß der Port umgeschaltet werden: Wenn der Stream im Program-

miermodus ist (STREAM_PROG(1) gesetzt) wird das Programmierdatenregister in den Stream eingegeben, Im Normalbetrieb werden aber die in WRITE_DATA geführten Schreibdaten angeschlossen.

Der schon aus der Slave-Mode Hardware bekannte kombinatorische Block zum bitweisen Verdrehen findet sich in den Zeilen 133–138 wieder. Als einziger Unterschied wird hier nicht der Slave-Mode Datenbus DATAIN, sondern der Ausgang des Lese-Streams verwendet.

Damit sind die Datenpfadoperationen abgeschlossen, in Zeile 141 beginnt der endliche Zustandsautomat der zentralen Steuerung. Der obligatorische Reset-Block findet sich in den Zeilen 143–150. Slave-Mode Schreibzugriffe auf die Parameterregister werden in den Zeilen 152–161 abgehandelt. Durch einen Schreibzugriff auf Register 3 (mit beliebigem Datenwert) wird die Master-Mode Ausführung gestartet.

Der dafür zuständige Teil des Zustandsautomaten liegt in den Zeilen 164–245. Nach dem Start beginnt die Ausführung im Zustand 'STATE_PROG_START. Hier werden beide Streams in den Programmiermodus geschaltet (Zeile 169) und die Anfangsadressen eingetragen: Stream 0 wird mit der Adresse des Speicherbereichs für die Eingangsdaten gefüttert, Stream 1 mit der für die Ausgangsdaten. Dann wird in das Register STATE der nächste anzunehmende Zustand eingetragen (Zeile 175). Auf dieselbe Weise werden dann die Anzahl der Datensätze (vermindert um 1, eine Eigenheit der Streams) in den Zeilen 180–183, die Schrittweite (jeweils ein Datensatz, Zeile 188–192), die Breite der Zugriffe (32b-Worte, Zeile 196–199) und die Zugriffsart (Stream 0 lesend, Stream 1 schreibend) in Zeile 204–208 eingetragen.

Im Zustand 'STATE_COMPUTE beginnend bei Zeile 210 beginnt schließlich die eigentliche Berechnung. Dazu werden beide Streams vom Programmiermodus in die normale Datenflußbetriebsart geschaltet (Zeile 213) und mittels des Signals STREAMSTART über das flowcontrol-Modul in Betrieb genommen (Zeile 215). Wenn alle Datensätze eingelesen wurden (Prüfung in Zeile 218) werden beide Streams gestoppt. Falls der Schreib-Stream nicht anderweitig beschäftigt ist (Zeile 222), wird er in Zeile 224 angewiesen, alle eventuell intern zwischengepufferten Daten tatsächlich in den Hauptspeicher zu schreiben. Als letztes wird in den nächsten Zustand 'STATE_SHUTDOWN gewechselt.

Falls aber noch nicht das letzte Datum bearbeitet wurde (COUNT war ungleich Null), wird bei laufendem Lese-Stream (Zeile 228) die Anzahl der noch zu bearbeitenden Datensätze dekrementiert. Man könnte hier annehmen, dass die Datenströme zu früh gestoppt werden: Es werden hier ja nur die gelesenen Daten gezählt. Im flowcontrol wird aber das Abschalten der Streams für den Schreib-Stream so verzögert, dass alle bereits gelesenen Daten auch bei deaktiviertem RUNNING noch geschrieben werden.

Das korrekte Beenden der Berechnung findet im Zustand 'STATE_SHUTDOWN statt. Hier wird überprüft, ob der Puffer des Schreib-Streams schon komplett entleert wurde (Zeile 236). Falls dies der Fall ist, wird die Leerung beendet (Zeile 238) und der Interrupt zur CPU hin ausgelöst (Zeile 240). Als letztes wird der Master-Mode Betrieb abgeschaltet (Zeile 242) und in Zeile 244 die Master-Mode Zustandsmaschine wieder in den Startzustand (für den nächsten Durchgang) zurückgesetzt.

Der letzte unscheinbare Zeilenblock 252–253 hat eine wichtige Funktion: Es muß eine Möglichkeit für die CPU geben, einen durch die RCU ausgelösten Interrupt wieder abzuschalten. Anderenfalls würde die Interrupt Handler-Funktion endlos aufgerufen werden. Bei unserer Beispielanwendung wird dazu ein Mechanismus verwendet, der bei jedem Lesezugriff auf ein beliebiges RCU-Register das Interrupt-Zustandsregister IRQSTATE löscht und so die Interrupt-Anforderung zurücknimmt.

Nach der Synthese benötigt die durch das HDL-Modell beschriebene Hardware auf dem Virtex II Pro FPGA der RCU 981 der 13696 verfügbaren Slices (7%). Obwohl sich an der Berechnung selbst ja nichts geändert hat (der rechnende comb_block ist in den Slave- und Master-Mode Versionen gleich), ist der Platzbedarf gegenüber den 45 Slices der Slave-Mode Version stark angestiegen. Die neu hinzugekommenen Slices gehen fast ausschließlich auf das Konto des MARC-Kerns, der die Datenströme in Master-Mode Speicherzugriffe umsetzt.

Dieses Design läuft nach der Platzierung und Verdrahtung immer noch mit einer maximalen Taktfrequenz von mehr als den geforderten 100 MHz. Auch diese Verlangsamung ist überwiegend auf die MARC inwohnende Komplexität zurückzuführen (Grund: MARC nimmt für eine verkürzte Latenz einen langsameren Takt in Kauf).

Entwurfstest durch Systemsimulation

Um den Hardware-Teil unabhängig von der Software testen zu können, simulieren wir das HDL-Modell der Anwendung. Dabei ist es aber nicht ausreichend, nur das in Listing 3.4 gezeigte Modul zu betrachten: Sinn des Master-Modes ist ein Zugriff auf den Hauptspeicher, für den damit auch ein simulierbares HDL-Modell vorliegen muß. Der Hauptspeicherezugriff erfolgt über MARC, daher müssen auch alle MARC-Module in die Simulation einbezogen werden. MARC greift über einen lokalen Bus auf den DDR-RAM Controller zu (in Abbildung 3.8 als MEM CTL bezeichnet) und benutzt RCU-interne SRAM-Blöcke als Datenpuffer. Beide dieser Einheiten werden also ebenfalls mit in die Simulation aufgenommen. Diese Ausweitung des Simulationsrahmens über die gerade entworfene Hardware hinaus wird als *Systemsimulation* bezeichnet.

Wir gehen im folgenden davon aus, dass alle für die Simulation erforderlichen HDL-Modelle vorliegen. Als nächstes gilt es dann, den gewünschten Test selbst zu formulieren. Der Testrahmen spielt dabei die Rolle des Software-Teils. Auf dem ML310 kommuniziert dieser anstelle der CPU mit der RCU über den PLB-Bus. Obwohl nicht der gesamte Funktionsumfang des PLB Protokolls benötigt wird, ist die Formulierung von Testzugriffen durch direktes Treiben von PLB-Signalen recht unhandlich. Zu diesem Zweck steht für das ML310 eine Bibliothek von Hilfsfunktionen bereit, die all diese Details abstrahieren. Listing 3.5 zeigt einen mit ihnen erstellten Testrahmen für unsere Beispielanwendung.

Listing 3.5: Systemsimulation mit PLB-Makros

```
'timescale 1ns / 10ps
// Schnittstelle für Standardtestrahmen
module stimulus (
    // Schnittstellensignale PPC Master <-> PLB On-Chip-Bus
    PLBC405DCUADDRACK,
    PLBC405DCUBUSY,
    PLBC405DCUERR,
    PLBC405DCURDDACK,
    PLBC405DCURDDBUS,
    PLBC405DCURDWDADDR,
    PLBC405DCUSSIZE1,
    PLBC405DCUWRDACK,
    PLBC405ICUADDRACK,
    PLBC405ICUBUSY,
    PLBC405ICUERR,
    PLBC405ICURDDACK,
    PLBC405ICURDDBUS,
    PLBC405ICURDWDADDR,
    PLBC405ICUSSIZE1,
    PLBCLK,
    C405PLBDCUABORT,
    C405PLBDCUABUS,
    C405PLBDCUBE,
    C405PLBDCUCACHEABLE,
    C405PLBDCUGUARDED,
    C405PLBDCUPRIORITY,
    C405PLBDCUREQUEST,
    C405PLBDCURNW,
    C405PLBDCUSIZE2,
    C405PLBDCUU0ATTR,
    C405PLBDCUWRDBUS,
    C405PLBDCUWRITETHRU,
    C405PLBICUABORT,
    C405PLBICUABUS,
    C405PLBICUCACHEABLE,
    C405PLBICUPRIORITY,
    C405PLBICUREQUEST,
    C405PLBICUSIZE,
```

```

C405PLBICUU0ATTR,
EICC405EXTINPUTIRQ
);
// Bibliothek von Hilfsfunktionen lesen
#include "plbutils.v"
// Hier beginnt der eigentliche Test
initial begin : test
// Hilfsvariable als Ziel für Leseoperationen
reg [31:0] data;
// Initialisiere Simulationsumgebung
Startup;
// TODO: Ggf. Eingabedaten aus Datei lesen
ReadMemFile("input.mem");
// Führe einen systemweiten Reset durch
SystemReset;
// Zeichne alle Signale im ganzen System auf
StartRecordingSignals;
// Lese ein 32b Wort von der Adresse 40
// Da dort kein Register liegt, sollte
// die Magic Number 0x00COFFEE zurückkommen
Read32('SLAVE_BASE + 32'h28, data);
$display("Magic: %h\n", data);
// Startadresse 4 in Register 0 schreiben
Write32('SLAVE_BASE + 32'h0, 32'h00000004);
// Zieladresse 4096 in Register 1 schreiben
Write32('SLAVE_BASE + 32'h4, 32'h00001000);
// Datensatzlänge 48 in Register 2 schreiben
Write32('SLAVE_BASE + 32'h8, 32'h00000030);
// RC starten durch Schreiben auf Register 3
Write32('SLAVE_BASE + 32'hc, 32'h00000001);
// Warte, bis RC durch IRQ das Ende anzeigt
RunUntilInterrupt;
// TODO: Ggf. Ausgabedaten in Datei schreiben
WriteMemFile("output.mem", 32'h00001000, 32'h00000030);
// Fahre Simulationsumgebung wieder herunter
Shutdown;
end
endmodule

```

Die Schnittstelle unseres Stimulus-Moduls ist durch die Simulationsumgebung vorgegeben und darf nicht verändert werden (Zeile 6-41). Indem wir in Zeile 45 einfach das PLB Makropaket laden, brauchen wir uns um diese Details anschließend nicht mehr zu kümmern, Nach dem Initialisieren des Pakets in Zeile 54 wird in Zeile 60 die simulierte System-Hardware korrekt zurückgesetzt. Darauf weisen wir den Simulator in Zeile 63 an, alle Signale unseres Entwurfes aufzuzeichnen, damit wir sie später in aller Ruhe studieren können

(z.B. als Waves oder Speicherauszüge). Nun folgen die eigentlichen Testanweisungen, die als eine Folge von Lese- und Schreibzugriffen auf die RCU kodiert sind. Wir lesen einen 32b Wert von Byte-Adresse 40 (=24'h28). Dies entspricht einem Zugriff auf Register 10 (alle unsere Register sind 4B groß). Da wir nur drei Register in unserem Design verwenden, erwarten wir hier die in Zeile 118 von Listing 3.4 definierte 'magic number' als Ergebnis. Das Makro SLAVE_BASE definiert einen systemabhängigen Adreßversatz (S0). In Zeile 69 von Listing 3.5 wird der gelesene Wert während des Simulationslaufes auf die Konsole ausgegeben. Der Master-Mode der Hardware wird durch die folgenden vier Anweisungen getestet: In Zeile 72 schreiben wir die Hauptspeicheradresse des Bereiches mit den Eingabedaten in Register 0 (=SOURCEADDR). In diesem Beispiel nehmen wir an, dass die Eingabedaten ab Byte-Adresse 4 im Speicher liegen. Der Zielbereich für die Ausgabedaten (ab Byte-Adresse 24'h001000) wird in Zeile 75 durch einen Schreib-Zugriff in Register 1 der RCU (=DESTADDR) eingetragen. Analog wird in Register 2 (=COUNT) die Anzahl der Datensätze für diesen Test eingetragen (hier 48). Durch den Schreibzugriff in Zeile 81 auf Register 3 wird schließlich die RCU gestartet. Wir lassen die Simulation nun solange laufen, bis die RCU einen Interrupt an die CPU auslöst (Zeile 84). Dann fahren wir die Simulationsumgebung wieder herunter und beenden den Simulator. Nun können wir mit anderen Werkzeugen die Ergebnisse visualisieren und untersuchen. Letzteres läßt sich häufig auch automatisieren (z.B. Vergleich von Ist- mit Sollwerten) und schafft so die Möglichkeit für automatische Regressionstests. Dabei wird nach allen Änderungen am Design automatisch die Funktion der bereits vorher getesteten Teile überprüft. Dies ist eine Vorgehensweise, die die schnelle Erkennung von Fehlern nach Design-Änderungen unterstützt.

Software-Teil der Master-Mode Anwendung

Der in Listing 3.6 gezeigte Software-Teil der Master-Mode Version unseres Beispiels ist sehr ähnlich zur Slave-Mode Software.

Listing 3.6: Software-Teil der Master-Mode Anwendung

```

#include <stdio.h>
#include <stdlib.h>
#include <acev/rcu.h>

// Adresse des DMA-Basisregisters
#define REG_DMA_BASE 0x3BFFFFFF

// Anzahl Datensätze in Ein- und Ausgabedatei
#define NUM_WORDS 256*1024

// Nummern der verschiedenen RCU-Register in diesem Entwurf
#define REG_SOURCE_ADDR 0
#define REG_DEST_ADDR 1
#define REG_COUNT 2
#define REG_START 3

// Hauptprogramm
main()
{
    // Ein- und Ausgabedateien
    FILE *infile , *outfile ;

    // Benutze vorzeichenlose Ganzzahlen (32b Worte) für alle Variablen
    unsigned long volatile *inwords, *outwords, *inwords_phys, *outwords_phys;

    // Marker für Zeitmessung
    unsigned long long start, stop, rcu_get_ticks ();

    // Zeiger auf S0-Bereich mit RCU-Registern
    unsigned long volatile *rcu;

    // RC initialisieren
    rcu_init ();

    // Zeiger auf S0-Bereich mit RCU-Registern holen
    rcu = rcu_get_s0(NULL);

    // fordere Speicher für Ein- und Ausgabefelder an
    // *_phys zeigt auf die physikalische Speicheradresse
    // aus Sicht der Hardware
    inwords = rcu_malloc_master(2 * NUM_WORDS * sizeof(unsigned long),
                               (void **) &inwords_phys);
    outwords = inwords + NUM_WORDS;
    outwords_phys = inwords_phys + NUM_WORDS;

    if (!inwords || !inwords_phys) {
        fprintf(stderr, "out_of_memory\n");
        exit (1);
    }

    // Setze DMA-Basisregister auf die nächstniedrigere 16MB-Grenze
    rc[REG_DMA_BASE] = (unsigned long) inwords_phys & 0xFF000000;

    // Funktioniert der Slave Zugriff?
    printf ("Magic:_%08lx\n", rcu[28]);

```

```

// Öffne die Dateien zum Lesen und Schreiben
infile = fopen("test1.in", "r");
outfile = fopen("test1.out", "w");

// Lese komplette Eingabedatei in Eingabe-Speicherbereich
fread(inwords, sizeof(unsigned long), NUM_WORDS, infile);

// Merke Startzeit der Berechnung
start = rcu_get_ticks ();

// Übertrage Parameter an RC (nicht die Daten selbst )
rcu[REG_SOURCE_ADDR] = inwords_phys; // Physikalische(!) Startadresse
rcu[REG_DEST_ADDR] = outwords_phys; // Physikalische(!) Zieladresse
rcu[REG_COUNT] = NUM_WORDS; // Anzahl Datensätze
rcu[REG_START] = 1; // Startkommando für RC

// Warte auf Ende der Berechnung (wird über IRQ angezeigt)
rcu_wait ();

// merke Stopzeit
stop = rcu_get_ticks ();

// Hardware Interrupt zurücksetzen durch beliebigen HW Lesezugriff
rcu[28];

// Schreibe das komplette Ausgabefeld in die Ausgabedatei
fwrite(outwords, sizeof(unsigned long), NUM_WORDS, outfile);

// Schließe Dateien
fclose ( infile );
fclose ( outfile );

// Gebe Speicher für Ein-/Ausgabefelder wieder frei
rcu_free_master((void *) inwords);

// Gebe Ergebnis der Zeitmessung in Mikrosekunden aus
printf ("Zeit: %8lldµs\n", (stop-start)/TICKS_PER_USEC);
}

```

Die Zeilen 15–18 definieren symbolische Namen für die Hardware-Register.

Das Hauptprogramm ist weitestgehend unverändert geblieben. Ein entscheidender Unterschied besteht aber in der Anforderung des Speicherbereiches für Ein- und Ausgabedaten in Zeilen 44–47. Die CPU und die RCU adressieren den gleichen Speicher mit unterschiedlichen Adressen: Die CPU, die ja unter Linux virtuellen Speicher unterstützt, verwendet dabei *virtuelle* Adressen. Diese werden von einer Memory Management Unit (MMU) innerhalb der CPU dann in *physikalische* Adressen umgerechnet und auf den PLB ausgegeben. Die RCU hat aber keine MMU, sondern rechnet direkt mit *physikalischen* Adressen. Wir müssen also bei der Speicheranforderung die Adressen des Speicherbereiches in *beiden* Darstellungen bestimmen. Weiterhin muss sichergestellt werden, dass der erhaltene Speicherbereich tatsächlich im physikalischen Speicher präsent ist. Die RCU kann (wieder wegen der fehlenden MMU) nämlich nicht das Einladen noch fehlender Speicherseiten (*paging*) aus dem virtuellen Speicher veranlassen. Alle benötigten Seiten müssen vorher im physikalischen Speicher liegen. Beide dieser Anforderungen erfüllt die Funktion `rcu_malloc_master`. Wir fordern damit einen doppelt so großen Speicherbereich an, in dessen unterer Hälfte die Eingabedaten abgelegt werden. In die obere Hälfte wird die RCU die bitverdrehen Daten schreiben. Die Funktion gibt als Wert die *virtuelle* Adresse zurück, die weiter im Software-Programm verarbeitet werden kann. Die für die RCU benötigte *physikalische* Adresse des Speicherbereiches wird in den, als Referenz an die Funktion übergebenen, letzten Parameter eingetragen. Der so erhaltene Wert kann dann später in Zeile 71 an die RCU übergeben werden. Die Zeigerarithmetik der Zeilen 46 und 47 dient lediglich dazu, den Beginn des für die Ausgabedaten reservierten Bereichs innerhalb des Speicherblocks zu bestimmen.

Lösung	RCU-Taktfrequenz in MHz	RCU-Größe in Slices	Rechenzeit in μ s	Beschleunigung gegenüber SW
Reine Software			195942	1.00
Slave-Mode Hardware	100	45	32045	6.11
Master-Mode Hardware	100	981	5813	33.71

Tabelle 3.2: Übersicht über alle Realisierungen der Beispielanwendung

Auch hier finden zwei Rechnungen, getrennt für die virtuelle und physikalische Adresse, statt.

Die Schleife, die in der Slave-Mode Lösung die Eingabedaten wortweise zur Umrechnung an die RCU übergeben hat, ist nun vollständig ersetzt worden. Stattdessen werden in den Zeilen 71–74 die Parameter des aktuellen Programmlaufes in die RCU-Register geschrieben. Der Schreibzugriff in Zeile 74 startet schließlich die RCU-Ausführung. Von nun an laufen CPU und RCU parallel! Da wir für unser Beispiel aber keine weiteren Aufgaben auszuführen haben, lassen wir die Software-Ausführung einfach ruhen und warten, bis die RCU mit ihren Berechnungen fertig ist. Dieser Effekt wird durch den Aufruf von `rcu_wait()` in Zeile 77 erreicht. Nach dem Auslösen eines CPU-Interrupts durch die RCU wird die Software-Ausführung in Zeile 74 wieder aufgenommen. In Zeile 77 wird durch einen Lesezugriff auf die RCU der Interrupt abgeschaltet (wie im HDL-Modell auf Zeilen 252–253, Listing 3.4 beschrieben). Der Programmablauf unterscheidet sich danach nicht mehr von der Slave-Mode Variante.

Wie schnell läuft nun die Master-Mode Lösung? Bei einer RCU-Taktfrequenz von 100 MHz wird ein Datensatz von 256Kw in **5813** μ s bearbeitet. Das ist fast 34-mal schneller, als auf dem 300MHz PowerPC in Software! Tabelle 3.2 zeigt alle Ergebnisse noch einmal in einer Übersicht.

4 Beispiele

4.1 Slave-Mode-Anwendung

In diesem Abschnitt sind die Quellcodes des durch `mkslave` angelegten Slave-Mode-Projekts gezeigt.

4.1.1 main.c

Listing 4.1: Slave-Mode Software

```
#include <stdio.h> 1
#include <stdlib.h> 2
#include <acev/rcu.h> 3
#include <signal.h> 4

main() 6
{
    // Marker f"ur Zeitmessung 8
    unsigned long long start, stop, rcu_get_ticks (); 9
    // Zeiger auf RC-Adressraum 10
    volatile unsigned long *rc; 11

    // TODO Hier weitere Variablen definieren 13
    // **** ... 14

    // TODO Hier Eingabedaten lesen 16
    // **** ... 17

    // RC initialisieren 19
    rcu_init (); 20
    // Zeiger auf RC-Adressraum holen 21
    rc = rcu_get_s0(NULL); 22

    // Merke Startzeit der Berechnung 24
    start = rcu_get_ticks (); 25

    // TODO Hier Daten bearbeiten 27
    // **** ... 28

    printf("rc(0) Wert nach RESET = %08x\n", rc[0]); 30
    // neuen Wert schreiben 31
    rc[0] = 0x87654321; 32
    printf("rc(0) Neuer Wert = %08x\n", rc[0]); 33

    // merke Stopzeit 35
    stop = rcu_get_ticks (); 36

    // TODO Hier Ausgabedaten schreiben 38
    // **** ... 39

    // Gebe Ergebnis der Zeitmessung in Mikrosekunden aus 41
    printf("Zeit: %lldµs\n", (stop-start)/TICKS_PER_USEC); 42
```


4.1.2 user.v

Listing 4.2: Slave-Mode Hardware

```

// 1
// user.v 2
// 3
// Generische Slave-Mode Anwendung: Realisiert ein les-/schreibbares 4
// Hardware-Register 5
// 6

module user( 8
    CLK, 9
    RESET, 10
    ADDRESSED, 11
    WRITE, 12
    DATAIN, 13
    DATAOUT, 14
    ADDRESS, 15
    IRQ 16
); 17

    // Eing"ange 19
    input CLK; 20
    input RESET; 21
    input ADDRESSED; 22
    input WRITE; 23
    input [31:0] DATAIN; 24
    input [23:2] ADDRESS; 25

    // Ausg"ange 27
    output [31:0] DATAOUT; 28
    output IRQ; 29

    wire IRQ = 1'b0; // wird im Slave-Mode nicht gebraucht 31

    // Beginn der Anwendung ***** 33

    reg [31:0] outreg; // Ergebnisregister 35

    // Ausgabedaten auf Bus legen 37
    // TODO Weitere Register anhand dekodierter Adresse anlegen 38
    // **** ... 39
    assign DATAOUT = (ADDRESS[3:2] == 2'b00) ? outreg 40
        : 32'hC0FFEE11; 41

    // Steuerung 43
    always @(posedge CLK or posedge RESET) begin 44
        // Initialisiere Register 45
        if (RESET) begin 46
            outreg <= 32'hDEADBEEF; 47
            // TODO Weitere Register initialisieren 48
            // **** ... 49

            // Schreibzugriff auf RC 51
        end else if ( ADDRESSED & WRITE) begin 52
            // TODO Hier weitere Register dekodieren 53

```

```

// **** ...
case (ADDRESS[3:2])
    2'b00: outreg <= DATAIN;
    default: ;
endcase
end
end
endmodule

```

4.2 Master-Mode-Anwendung

In diesem Abschnitt sind die Quellcodes des durch `mkmaster` angelegten Master-Mode-Projekts gezeigt.

4.2.1 main.c

Listing 4.3: Master-Mode Software

```

#include <stdio.h>
#include <stdlib.h>
#include <acev/rcu.h>

// Adresse des DMA-Basisregisters
#define REG_DMA_BASE 0x3BFFFFFF

// Anzahl Datensätze in Ein- und Ausgabedatei
#define NUM_WORDS 256*1024

// Definitionen für RC Register
#define REG_SOURCE_ADDR 0
#define REG_DEST_ADDR 1
#define REG_COUNT 2
#define REG_START 3

// Hauptprogramm
main()
{
    // Zeiger auf Adressraum mit RC-Registern
    unsigned long volatile *rc;

    // Benutze vorzeichenlose Ganzzahlen für alle Variablen
    unsigned long volatile *inwords, *outwords,
        *inwords_phys, *outwords_phys;

    // Marker für Zeitmessung
    unsigned long long start, stop, rcu_get_ticks ();

    // TODO Weitere Variablendefinitionen
    // **** ...

    // RC initialisieren
    rcu_init ();
    // Zeiger auf Adressraum mit RC-Registern holen
    rc = rcu_get_s0(NULL);

    // fordere Speicher für Ein- und Ausgabearrays an
    // *_phys zeigt auf die physikalische Speicheradresse
    // aus Sicht der Hardware

```

```

inwords = rcu_malloc_master(2 * NUM_WORDS * sizeof(unsigned long),
                           (void **) &inwords_phys);
outwords = inwords + NUM_WORDS;
outwords_phys = inwords_phys + NUM_WORDS;

// Setze DMA-Basisregister auf die n-achstniedrigere 16MB-Grenze
rc[REG_DMA_BASE] = (unsigned long) inwords_phys & 0xFF000000;

// Funktioniert der Slave Zugriff?
printf ("Magic: %08lx\n", rc[28]);

if (!inwords || !inwords_phys) {
    fprintf(stderr, "out_of_memory\n");
    exit (1);
}

// TODO Einlesen von Eingabedaten
// **** ...

// trage erkennbare daten in Speicherbereiche ein
inwords[0] = 0xdeaddead;
outwords[0] = 0xbeefbeef;

// Merke Startzeit der Berechnung
start = rcu_get_ticks ();

// "Übertrage Parameter an RC (nicht die Daten selbst)
rc[REG_SOURCE_ADDR] = inwords_phys; // physikalische(!) startadresse
rc[REG_DEST_ADDR] = outwords_phys; // physikalische(!) zieladresse
rc[REG_COUNT] = NUM_WORDS; // anzahl datenworte

// TODO Weitere Parameter "übertragen
// **** ...

rc[REG_START] = 1; // startkommando

// Warte auf Ende der Berechnung (wird "über IRQ angezeigt)
rcu_wait ();

// merke Stopzeit
stop = rcu_get_ticks ();

// Hardware Interrupt zur"ucksetzen durch beliebigen HW Lesezugriff
rc [28];

// TODO Schreiben von Ausgabedaten
// **** ...

// jetzt mu"s outwords[0] gleich inwords[0] sein (es wurde ja kopiert)
printf ("inwords(0)=%08x,outwords(0)=%08x\n", inwords[0], outwords[0]);

// Gebe Speicher f"ur Ein-/Ausgabe-Arrays wieder frei
rcu_free_master((void *) inwords);

// Gebe Ergebnis der Zeitmessung in Mikrosekunden aus
printf ("NUM_WORDS=%d\n", NUM_WORDS);
printf ("Zeit: %08lldµs\n", (stop-start)/TICKS_PER_USEC);
}

```

4.2.2 user.v

Listing 4.4: Master-Mode Hardware

```
// 1
// Beispielschaltung f"ur Master Mode–Zugriffe mittels MARC 2
// 3
// Diese Schaltung kopiert durch die RC einen Quellspeicherbereich 4
// in einen Zielspeicherbereich (COUNT 32b Worte) 5

'include "acsdefs.v" 7

module user ( 9
    // *** Globale Signale 10
    CLK, // Takt 11
    RESET, // Chip Reset 12

    // *** Slave–Interface 14
    ADDRESSED, // RC angesprochen im Slave–Mode 15
    WRITE, // Schreibzugriff? 16
    DATAIN, // Dateneingang 17
    DATAOUT, // Datenausgang 18
    ADDRESS, // Adresseingang 19
    IRQ, // L"ost Interrupt (IRQ) an CPU aus 20

    // *** Stream–Interface 22
    STREAM_READ, // Read–Datenbus 23
    STREAM_WRITE_PROG, // Write–program–Datenbus 24
    STREAM_STALL, // Stallsignale 25
    STREAM_ENABLE, // Enables 26
    STREAM_FLUSH, // Flushsignale 27
    STREAM_PROG // Programmsignale 28
); 29

// Schnittstelle ***** 31

// Eing"ange 33
input CLK; 34
input RESET; 35
input ADDRESSED; 36
input WRITE; 37
input [31:0] DATAIN; 38
input [23:2] ADDRESS; 39
input ['STREAM_DATA_BUS] STREAM_READ; 40
input ['STREAM_CNTL_BUS] STREAM_STALL; 41

// Ausg"ange 43
output [31:0] DATAOUT; 44
output ['STREAM_DATA_BUS] STREAM_WRITE_PROG; 45
output ['STREAM_CNTL_BUS] STREAM_ENABLE; 46
output ['STREAM_CNTL_BUS] STREAM_FLUSH; 47
output ['STREAM_CNTL_BUS] STREAM_PROG; 48
output IRQ; 49

// Deklaration f"ur Stream–Interface 51
wire ['STREAM_DATA_BUS] STREAM_READ; 52
wire ['STREAM_DATA_BUS] STREAM_WRITE_PROG; 53
wire ['STREAM_CNTL_BUS] STREAM_STALL; 54
wire ['STREAM_CNTL_BUS] STREAM_ENABLE; 55
reg ['STREAM_CNTL_BUS] STREAM_FLUSH; 56
reg ['STREAM_CNTL_BUS] STREAM_PROG; 57
```

```

// Konstantendefinitionen *****
// FSM Zustände
#define STATE_PROG_START 0 // programmiere Startadressen in Streams
#define STATE_PROG_COUNT 1 // programmiere Datensatzzahl in Streams
#define STATE_PROG_STEP 2 // programmiere Schrittweite in Streams
#define STATE_PROG_WIDTH 3 // programmiere Zugriffsbreite in Streams
#define STATE_PROG_MODE 4 // programmiere Betriebsart in Streams
#define STATE_COMPUTE 5 // führe Berechnung auf Streamdaten aus
#define STATE_WAIT 6 // Warte einen Takt zum Entleeren des Streams
#define STATE_SHUTDOWN 7 // Beende Berechnung

// Beginn der Anwendung *****

// Wurde Anwendung gestartet?
reg START;
// Anfangsadresse der Eingabedaten
reg [31:0] SOURCEADDR;
// Anfangsadresse der Ausgabedaten
reg [31:0] DESTADDR;
// Länge der Daten (als 32b Worte)
reg [31:0] COUNT;
// Lost Interrupt aus
reg IRQSTATE;
// Aktueller Zustand der Anwendung
reg [4:0] STATE;
// Sind Streams gestartet?
reg STREAMSTART;
// Programmierdaten-Register für Streams
reg [31:0] STREAM_PROGDATA_0;
reg [31:0] STREAM_PROGDATA_1;

// Daten
wire [31:0] WRITE_DATA;

// Abkürzung für Registernummer 0 ... 15
wire [3:0] REGNUM = ADDRESS[5:2];

// Streams laufen, nachdem sie gestartet worden sind und solange
// noch Daten zu bearbeiten sind.
wire RUNNING = STREAMSTART & (COUNT != 0);

// Flußkontrolle zwischen Ein- und Ausgabedatenstrom
flowcontrol FC (
    CLK, // Takt
    RUNNING, // Streams laufen lassen?
    STREAM_STALL[0], // Hängt Stream 0 (Eingabe-Strom)?
    STREAM_STALL[1], // Hängt Stream 1 (Ausgabe-Strom)?
    /** "Andern **/ STREAM_READ[STREAM_0], // Von Anwendung zu schreibende Daten
    STREAM_ENABLE[0], // Stream 0 starten oder anhalten
    STREAM_ENABLE[1], // Stream 1 starten oder anhalten
    WRITE_DATA // Eingangsdaten für Ausgabe-Strom
);

// Gebe IRQSTATE Register an CPU IRQ-Leitung aus
assign IRQ = IRQSTATE;

// Gebe immer das gerade adressierte Register aus.
// Nicht benotigte Register geben eine Magic-Number
// und den aktuellen IRQ-Status im MSB zurück

```

```

wire [31:0] DATAOUT = (REGNUM === 4'h0) ? SOURCEADDR
                : (REGNUM === 4'h1) ? DESTADDR
                : (REGNUM === 4'h2) ? COUNT
                : (32'h00COFFEE | (IRQSTATE << 31));

// Schalte Streams zwischen Programmier- und Datenbetrieb um
// Stream0 ist Lese-Stream, sein Eingang kann immer im Programmierbetrieb sein
assign STREAM_WRITE_PROG[STREAM_0] = STREAM_PROGDATA_0;
// Stream1 ist Schreib-Stream, hier mu"s der Eingang umgeschaltet werden
assign STREAM_WRITE_PROG[STREAM_1] = (STREAM_PROG[1]) ?
                STREAM_PROGDATA_1 : WRITE_DATA;

// Controller FSM "uberwacht gesamte Anwendung
always @(posedge CLK or posedge RESET) begin
// Initialisiere Register bei chip-weitem Reset
if (RESET) begin
    STATE          <= 'STATE_PROG_START;
    IRQSTATE       <= 0;
    STREAMSTART   <= 0; STREAM_PROG   <= 0;
    STREAM_FLUSH  <= 0; STREAM_PROGDATA_0 <= 0;
    STREAM_PROGDATA_1 <= 0; SOURCEADDR <= 0;
    DESTADDR      <= 0; COUNT        <= 0;
    START         <= 0;
// Schreibzugriff auf RC, schreibe in entsprechendes Register
end else if (ADDRESSED & WRITE) begin
    case (REGNUM)
        0: SOURCEADDR <= DATAIN;
        1: DESTADDR <= DATAIN;
        2: COUNT <= DATAIN;
        3: begin
            START <= 1; // Startkommando, beginne Ausf"uhrung
        end
    default ;
    endcase
end else begin
// CPU hat Berechnung gestartet , keine Slave-Mode Zugriffe mehr m"oglich
if (START) begin
    case (STATE)
        'STATE_PROG_START:
            begin
// Beide Streams in Programmierungsmodus schalten
                STREAM_PROG[1:0] <= 2'b11;
// Anfangsadresse f"ur Stream 0 schreiben
                STREAM_PROGDATA_0 <= SOURCEADDR;
// Anfangsadresse f"ur Stream 1 schreiben
                STREAM_PROGDATA_1 <= DESTADDR;
// FSM weitersetzen
                STATE <= 'STATE_PROG_COUNT;
            end
        'STATE_PROG_COUNT:
            begin
// Anzahl Datens"atze - 1 (bei beiden Streams gleich)
                STREAM_PROGDATA_0 <= COUNT - 1;
                STREAM_PROGDATA_1 <= COUNT - 1;
// FSM weitersetzen
                STATE <= 'STATE_PROG_STEP;
            end
        'STATE_PROG_STEP:
            begin
// Schrittweite : 1 Datensatz (bei beiden Streams gleich)
                STREAM_PROGDATA_0 <= 1;

```

```

STREAM_PROGDATA_1 <= 1;
// FSM weitersetzen
STATE <= 'STATE_PROG_WIDTHH;
end
'STATE_PROG_WIDTHH:
begin
// Wordbreite der Zugriffe: 32b (bei beiden Streams gleich)
STREAM_PROGDATA_0 <= 'STREAM_32B;
STREAM_PROGDATA_1 <= 'STREAM_32B;
// FSM weitersetzen
STATE <= 'STATE_PROG_MODE;
end
'STATE_PROG_MODE:
begin
// Zugriffsart f"ur Stream 0: Lesen
STREAM_PROGDATA_0 <= 'STREAM_READ;
// Zugriffsart f"ur Stream 1: Schreiben
STREAM_PROGDATA_1 <= 'STREAM_WRITE;
// FSM weitersetzen
STATE <= 'STATE_COMPUTE;
end
'STATE_COMPUTE:
begin
// Programmiermodus f"ur beide Streams abschalten
STREAM_PROG[1:0] <= 0;
// Beide Streams starten (via flowcontrol –Modul)
STREAMSTART <= 1;

// Alle Datens"atze bearbeitet ?
if (COUNT == 0) begin
// Dann beide Streams stoppen
STREAMSTART <= 0;
// Falls Write–Stream fertig
if (!STREAM_STALL[1]) begin
// alle noch gepufferten Daten wirklich schreiben
STREAM_FLUSH[1] <= 1;
// FSM weitersetzen
STATE <= 'STATE_WAIT;
end
end else if (STREAM_ENABLE[0] & ~STREAM_STALL[0])
// Nur dann einen Datensatz als bearbeitet z"ahlen,
// wenn Stream 0 aktiv liest (ENABLE) und nicht h"angt (!STALL)
COUNT <= COUNT – 1;

end
'STATE_WAIT:
begin
STATE <= 'STATE_SHUTDOWN;
end
'STATE_SHUTDOWN:
begin
// Ist Schreibpuffer schon komplett geleert ?
if (!STREAM_STALL[1]) begin
// ja, Leerung beenden
STREAM_FLUSH[1] <= 0;
// CPU durch IRQ Fertigwerden der RC anzeigen
IRQSTATE <= 1;
// FSM stoppen (RC jetzt wieder im Slave–Mode)
START <= 0;
// FSM auf Startzustand zur"ucksetzen
STATE <= 'STATE_PROG_START;

```

```

        end
    end
    // sollte nicht auftreten , nur f"ur Logikoptimierung
    default: STATE <='bx';
endcase
end
// Bei jedem Zugriff auf RC im Slave-Mode, IRQ ausschalten
else if (ADDRESSED)
    IRQSTATE <= 0;
end
end
endmodule

```

4.3 Bildbearbeitung

Hier finden sich das Listing der Beispielanwendung `brighten` sowie ein Testbild vor und nach der Bearbeitung.

Listing 4.5: Datei `brighten.c`

```

//
// Erh"ohet den Grauwert jedes Bildpunktes um 100.
//
// "Ubersetzung mit: make brighten
// Aufruf mit: ./brighten lena256.pgm lena256b.pgm
//
#include "stdio.h"

main(
    int argc, // Anzahl Kommandozeilenparameter+Programmname
    char *argv[] // Programmname+Kommandozeilenparameter als Strings
)
{
    // Speicher f"ur ganzes 256x256 Bild, Werte 0 ... 255
    unsigned char image[256][256];

    // Iterationsvariable
    unsigned int n;

    // Zeiger auf aktuellen Bildpunkt
    unsigned char *p;

    // Neuer Grauwert. Wichtig: Wertebereich hier ist 0 ... 65535
    unsigned short w;

    // Eingabe- und Ausgabedatei
    FILE *infile, *outfile;

    // Aufrufparameter pr"ufen
    if (argc != 3) {
        printf("Falscher Aufruf, richtig: ./brighten Eingabe.pgm Ausgabe.pgm\n");
        exit(1);
    }

    // Eingabedatei "offnen
    if ((infile = fopen(argv[1], "r")) == NULL) {
        printf("Eingabedatei %s konnte nicht ge"offnet werden.\n", argv[1]);
    }
}

```



```

    exit (2);
}

// Ausgabedatei "offnen
if (( outfile = fopen(argv[2], "w")) == NULL) {
    printf("Ausgabedatei_%s_konnte_nicht_ge\`offnet_werden.\`n", argv[2]);
    exit (3);
}

// Reiche PGM Kopfdatensatz direkt durch
fgets((void*)image, 80, infile );
fputs((void*)image, outfile );
fgets((void*)image, 80, infile );
fputs((void*)image, outfile );
fgets((void*)image, 80, infile );
fputs((void*)image, outfile );

// Lese gesamtes Feld von Bildpunkten auf einen Satz
fread(image, sizeof(unsigned char), 256*256, infile );

// ***** Ab hier Bildbearbeitung *****

// Durchlaufe alle Bildpunkte, p zeigt jeweils auf aktuellen Punkt
for (n = 0, p = (void*) image; n < 256*256; ++p, ++n) {

    // Berechne neue Helligkeit als alter Grauwert des Punktes plus 100
    w = *p + 100;

    // Da w gr"o"ser werden kann als 255, der hellste Wert aber
    // 255 ist , setzen wir gr"o"ser Werte einfach auf 255.
    *p = (w > 255) ? 255 : w;
}

// ***** Ende der Bildbearbeitung *****

// Gebe gesamtes Feld von Bildpunkten auf einen Satz aus
fwrite(image, sizeof(unsigned char), 256*256, outfile );

// Dateien schliessen
fclose ( infile );
fclose ( outfile );

// Programm mit Status 'Kein Fehler' beenden
exit (0);
}

```



Abbildung 4.1: Testbild `lena256.pgm` vor Bearbeitung



Abbildung 4.2: Testbild nach Bearbeitung durch `brighten`

Hinweise zum Thema Plagiarismus

Im Rahmen dieser Veranstaltung wird grundsätzlich eine vorher festgelegte Arbeitsgruppe bewertet. Bei signifikanten Leistungsunterschieden sind aber ggf. auch Einzelbewertungen möglich. Fremde Code-Bibliotheken außer den vom Dozenten zur Verfügung gestellten dürfen Sie *nicht* verwenden! Zusammenarbeit über Gruppengrenzen hinweg ist in Form der Diskussion von Lösungsideen erlaubt. Es dürfen aber *keine* Artefakte wie Programm-Code, Dokumentationsteile (Text, Zeichnungen, Messergebnisse) oder ähnliches ausgetauscht werden.

Der Fachbereich Informatik misst der Einhaltung der Grundregeln der wissenschaftlichen Ethik großen Wert bei. Mit der Abgabe einer Lösung (Hausaufgabe, Programmierprojekt, etc.) bestätigen Sie, dass Ihre Gruppe die alleinigen Autoren des gesamten Materials sind. Weiterführende Informationen zu diesem Thema finden Sie unter <http://www.informatik.tu-darmstadt.de/Plagiarism>.