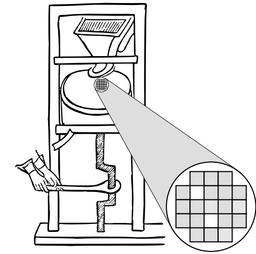
Technische Universität Darmstadt
FG Eingebettete Systeme und
ihre Anwendungen (ESA)

Holger Lange
Prof. Dr. Andreas Koch



11.04.2006

Leitfaden zum Praktikum “Adaptive Rechensysteme” im SS2006

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. Vor seiner Lektüre sollten Sie die Inhalte von Teil 3 (Adaptive Rechner) des Vorlesungsskripts verstanden haben.

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

0. Phase: Einführung

18.4.-23.4.2006

Am Dienstag, den 18.4.2006 um 16:15 Uhr findet im Raum S2|02 C110 (Piloty-Gebäude Hochschulstr. 10) ein erstes gemeinsames Treffen aller Praktikumsteilnehmer statt.

Vor Lektüre dieses Kapitels sollten Sie unbedingt zwei weitere Dokumente 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 Veranstaltungen am 19. und 20.4. in S2|02 E102 verwiesen, für Fortgeschritte gibt es eine Sprachreferenz unter

http://www-ee.eng.hawaii.edu/~msmith/ASICs/HTML/Verilog/Verilog.htm

Bei dem ersten Treffen liegt der Schwerpunkt neben den organisatorischen Details (wie Kolloquiumsterminen) bei der fachlichen Einführung. Der Praktikumsleitfaden wird besprochen und Fragen 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. Abschließend erhalten Sie pro Gruppe je einen Schlüssel für den Praktikumsraum. **Für diese Veranstaltung besteht Anwesenheitspflicht.**

1. Phase: Hardware-Experimente
24.4.-7.5.2006

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 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, 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
8.5.-14.5.2006

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 den Assi überprüft, und zwar im Rahmen der Kolloquien am Montag nach den Abgaben. Stellen Sie trotzdem sicher, daß Ihre Lösung schon *vor* dem Abgeben der schriftlichen Ausarbeitung funktioniert. 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

15.5.-21.5.2006

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/ACS06/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.

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 Aus-

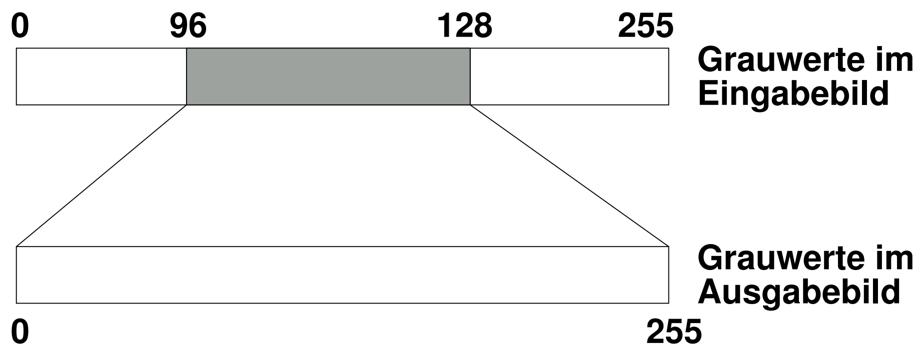


Abbildung 1.1: Idee der Kontrastspreizung

gabebild 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 $0 \dots 255$ 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 aufgespreizt, 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 Kontrastspreizung 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
22.5.-4.6.2006

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 vozeichenlose 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:

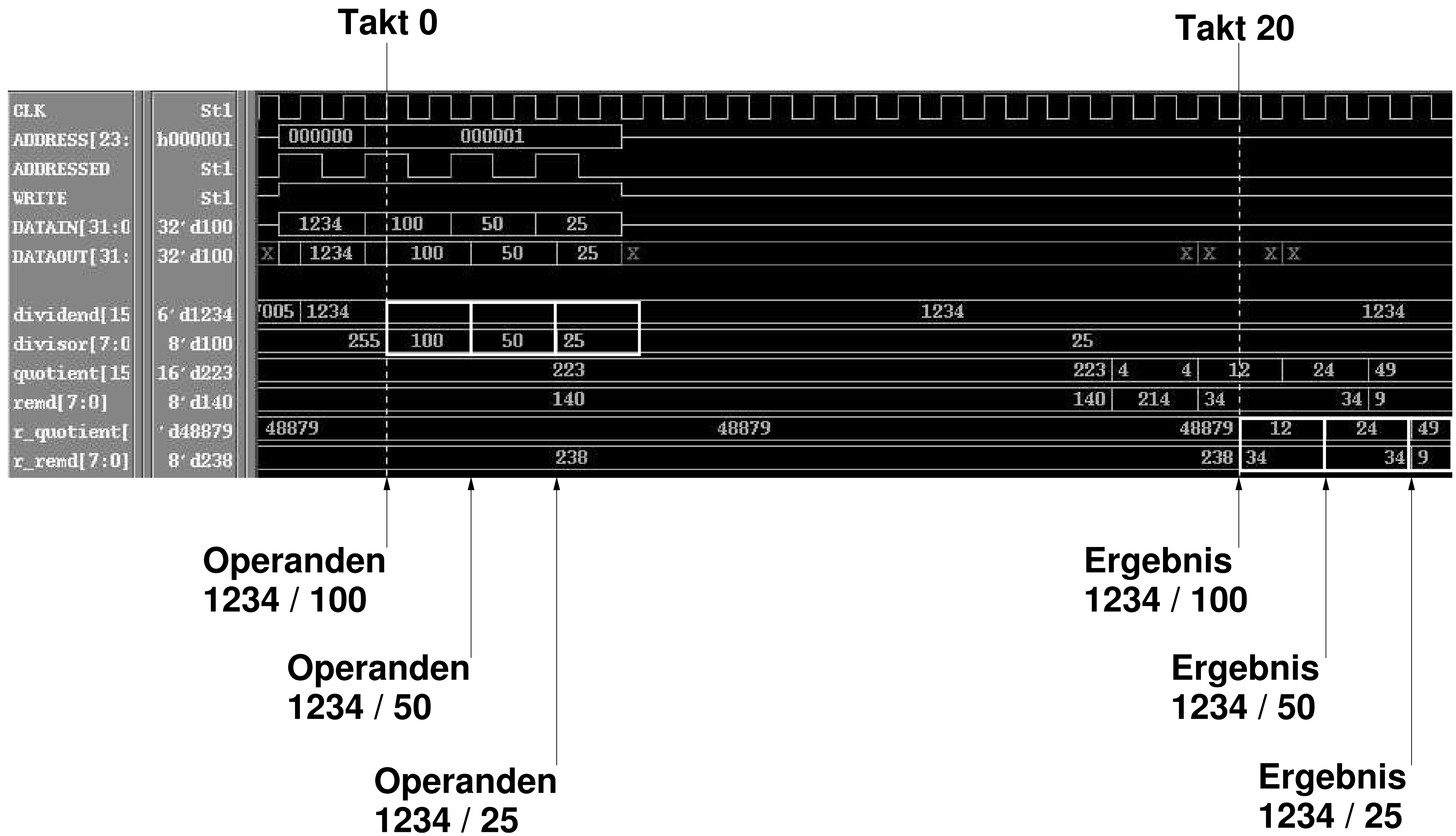


Abbildung 1.2: Pipelined Division von 1234/100, 1234/25 und 1234/25

1. Legen Sie ein neues Slave-Mode-Projekt für diese Phase an.

2. Lassen Sie sich vom Assi 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 Dividiererzelle, 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 Dividiererzelle in `user.v`.

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`.

11. Fertigen Sie eine Post-Layout-Simulation mit `make laysim` an.

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
5.6.-18.6.2006

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
19.6.-16.7.2006

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 3.13). 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

17.7.-23.7.2006

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 CAD-Umgebung

2.1 Einführung

In diesem Praktikum wird ein aus vielen Einzelprogrammen bestehender Werkzeugfluß verwendet, dazu gehören z.B. der C-Cross-Compiler für den Software-Teil Ihrer Anwendung, zum anderen werden unterschiedliche Werkzeuge für Synthese und Simulation verwendet. Da es in diesem Praktikum um ganze *Systeme* geht, müssen neben den von Ihnen entwickelten Teilen auch die restlichen Komponenten des Systems in den Fluß mit einbezogen werden. Um Ihnen die Arbeit so weit wie möglich zu erleichtern, wurde der Werkzeugfluß weitestgehend automatisiert. Dieser Leitfaden soll Sie in seine Bedienung einführen.

2.2 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 32b Register in Hardware, das durch den Software-Teil gelesen und beschrieben werden kann. Sie kann durch entsprechende Ergänzung leicht an die tatsächlichen Erfordernisse Ihres Entwurfs angepasst werden. Dazu bearbeiten Sie lediglich drei Dateien:

main.c : Enthält den Software-Teil, insbesondere die `main`-Funktion, mit der die Ausführung jedes C-Programmes begonnen wird. In der vorgeschlagenen Version wird das Hardware-Register gelesen, beschrieben, und dann wieder gelesen. Grundfunktionen wie die Initialisierung der RC und für die Zeitmessung stehen auch bereits zur Verfügung. Ersetzen Sie die Bearbeitung der Daten durch die für Ihre eigene Anwendung passenden Anweisungen.

user.v : Diese Datei enthält den Hardware-Teil Ihrer Anwendung in Form des Verilog-Moduls namens `user`. Dieses Modul hat nach aussen eine Slave-Mode-Schnittstelle und implementiert ein durch die Software les-/schreibbares Hardware-Register, das nach dem Reset auf

den gut erkennbaren Wert 0xDEADBEEF initialisiert wird. Bauen Sie ihre Änderungen in den durch dieses Modul vorgegebenen Rahmen ein.

stimulus.v : Diese Datei enthält die Stimuli für die Simulation (sowohl RTL als auch Post-Layout). Neben einigen administrativen Kommandos (`Startup`, `SystemReset`, `Shutdown`) stehen Ihnen für die Stimulation die Kommandos `Read32` und `Write32` zur Verfügung, die Zugriffe der CPU auf Ihre Slave-Mode-Schaltung nachbilden. Mit dem Kommando `RunFor` lassen Sie die angegebene Anzahl von Takten als Zeit vergehen, mit `RunUntilInterrupt` läuft die Simulation, bis die RC einen Interrupt auslöst.

Für Master-Mode-Anwendungen erzeugt Ihnen `mkmaster` die gleichen Dateien mit den gleichen Bedeutungen. Die hier verwendete Beispielanwendung kopiert die angegebene Anzahl von Datenworten von einem Quellspeicherbereich in einen Zielspeicherbereich. Um Sie an andere Aufgaben anzupassen, muß insbesondere die hier verwendete direkte Verbindung von Eingabedatenstrom an Ausgabedatenstrom (über die Instanz `FC` des Moduls `flowcontrol`) aufgetrennt werden. Stattdessen müssen die eigenen Berechnungen in den Datenstrom eingefügt werden.

2.3 RTL-Simulation

Um den Hardware-Teil Ihrer Anwendung auf RT-Ebene (ohne genaues Timing) zu simulieren, geben Sie auf der Kommandozeile die Anweisung `make rtlsim` ein. Falls Fehler in Ihrem Verilog-Modell auftraten, korrigieren Sie diese bitte.

Abbildung 2.1: VirSim Menübalken

Anderenfalls öffnet sich die interaktive Simulationsumgebung VirSim mit zwei Fenstern: Einem Menübalken (Abbildung 2.1) und dem Simulationskontrollfenster (Abbildung 2.2).

Bevor Sie die Simulation starten, müssen Sie die Signale auswählen, die Sie beobachten wollen. Dazu klicken Sie im Menübalken auf den Knopf Hierarchy, was zur Öffnung des Hierarchiefensters führt (Abbildung 2.3).

Im oberen Teil dieses Fensters können Sie nun Ihr Modul `user` auswählen. Dazu steigen Sie in das Modul `testbench_rtl` ab (Linksklick auf das Pluszeichen neben `testbench_rtl`). Nun werden die Untermodule unterhalb von `testbench_rtl` eingerückt sichtbar. Steigen Sie weiter ab in `SYSTEM` (wieder durch Linksklick auf das Plus). Nun sehen Sie eine ganze Reihe von Modulen. Finden Sie das Modul `plb_ddr_0`, steigen Sie abermals tiefer und öffnen Sie in der Hierarchiebene darunter `\plb_ddr_0/PLB_MARC_I`. Hier finden sie schließlich Ihr Modul `user`. Da es keinen Pfeil hat, instanziert es selbst keine weiteren Module. Klicken Sie mit der linken Maustaste nun einmal auf das `user`-Modul. Im rechten Kasten tauchen daraufhin die in `user` verwendeten Signale und Register auf (Abbildung 2.4).

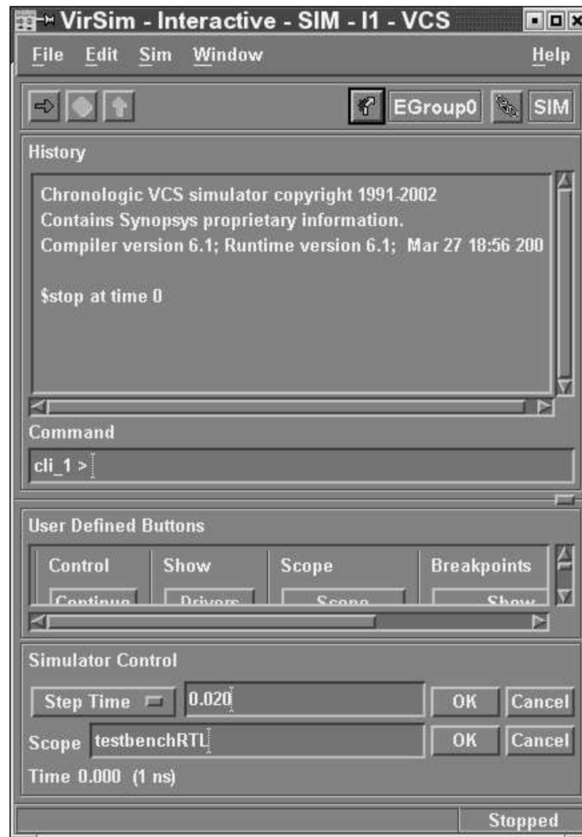


Abbildung 2.2: VirSim Kontrollfenster

Abbildung 2.3: VirSim Hierarchiefenster

Abbildung 2.4: `user`-Modul und Signale

Lassen Sie das so eingerichtete Hierarchiefenster geöffnet und legen Sie es beiseite. Klicken Sie nun in dem Menübalken auf den Knopf Waveform, um ein leeres Waveform-Fenster anzuzeigen (Abbildung 2.5).

Ordnen Sie das Waveform-Fenster so an, daß es das vorher geöffnete Hierarchiefenster nicht überlappt. Für jedes der Signale von `user`, dessen Signalverlauf Sie verfolgen möchten, gehen Sie wie folgt vor: Wählen Sie das Signal aus dem rechten Feld des Hierarchiefensters mit der *mittleren* Maustaste aus und ziehen es (bei gehaltener mittlerer Maustaste) in den noch schwarzen Bereich des Waveform-Fensters. Dort angekommen (der Cursor wird grün) lassen Sie die Maustaste los und der Name des Signals wird im Waveform-Fenster links angezeigt. Da die Simulation noch nicht gestartet ist, ist der Signalverlauf selbst noch als graues Rechteck dargestellt (das wird sich gleich ändern). Ziehen Sie auf diese Weise alle anderen interessanten Signale in das Waveform-Fenster. Ein Beispiel ist in Abbildung 2.6 dargestellt.

Sie können die Reihenfolge der Signale innerhalb des Waveform-Fenster durch Ziehen des Namens mit der mittleren Maustaste verändern. Falls Sie ein Signal doch nicht interessiert, wählen Sie seinen Namen mit der linken Maustaste an und wählen aus dem Menü Edit den Punkt Delete. Das Signal wird damit aus der Anzeige gelöscht.

Nun können Sie die Simulation tatsächlich starten. Dazu klicken sie im Simulationskontrollfenster den Knopf mit dem grünen Pfeil (unter dem Menü File). Im Waveform-Fenster sind nun die ersten Signalverläufe angezeigt. Sie können die Darstellung entsprechend Ihren Anforderungen verbessern: Verkleinern ist durch Klicken auf den Knopf mit dem kleinen weißen Z möglich, Vergrößern analog durch Klick auf den Knopf mit dem großen Z. Durch die horizontalen und vertikalen Rollbalken kann der angezeigte Ausschnitt auch verschoben werden. Sehr hilfreich ist es, gezielt einen

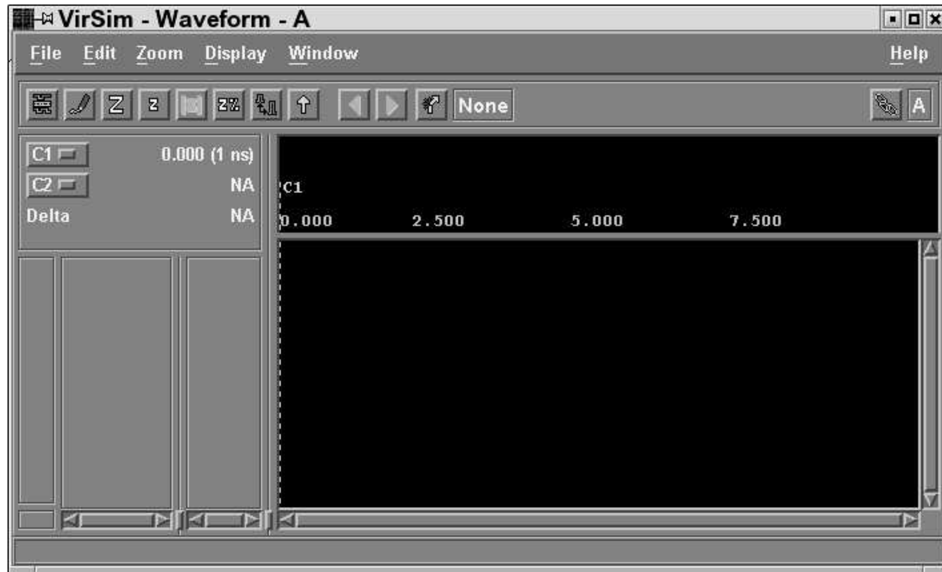


Abbildung 2.5: Leeres VirSim Waveform-Fenster

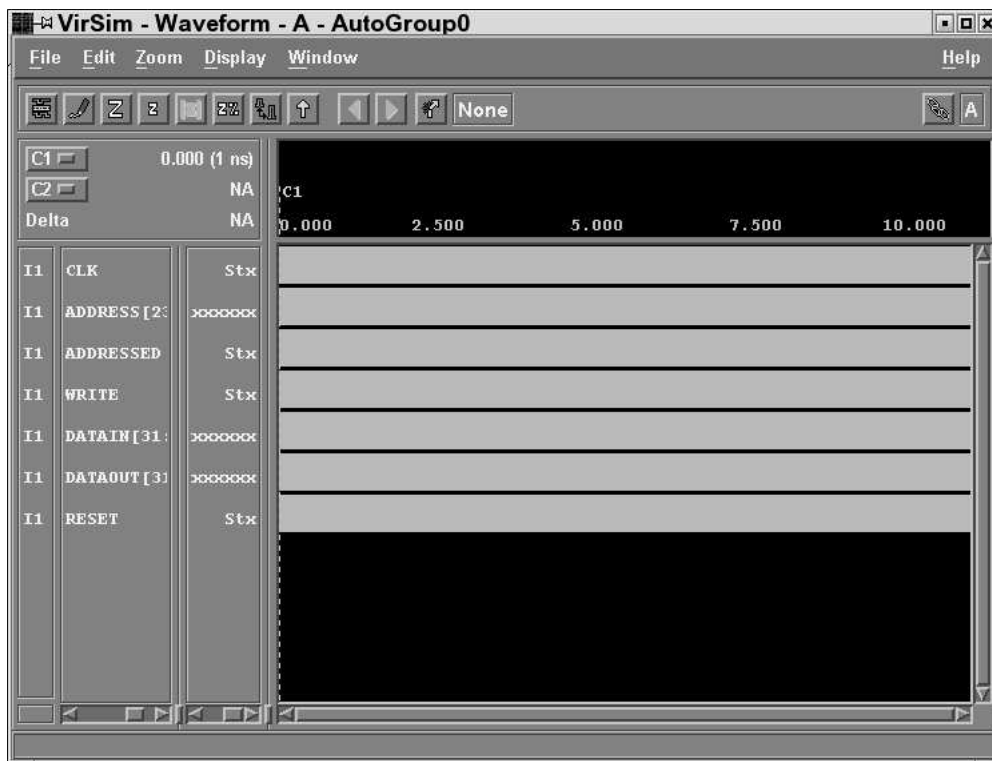


Abbildung 2.6: Befülltes Waveform-Fenster (vor Simulationsstart)

Bereich vergrößert anzuzeigen. Markieren Sie dazu die Enden des Bereichs durch je einen Klick in den Signalverlauf mit der linken und mittleren Maustaste und klicken Sie auf den Knopf mit dem Z zwischen zwei vertikalen Balken. Der ausgewählte Bereich wird jetzt vergrößert dargestellt (Abbildung 2.7).

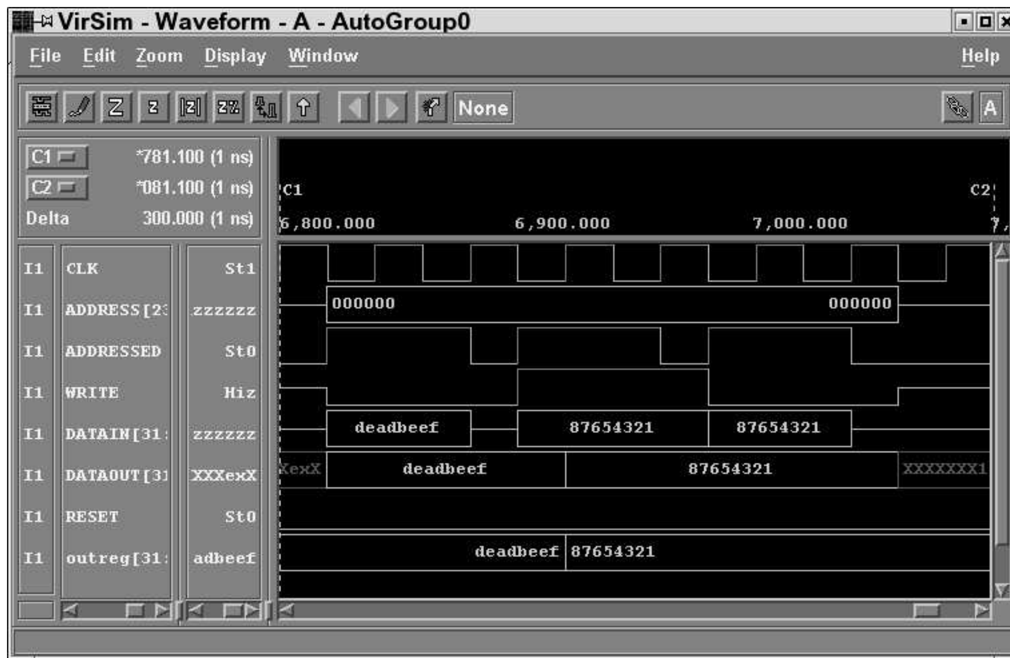


Abbildung 2.7: Slave-Mode Zugriffe als Signalverlauf

Diese Abbildung zeigt die in **`stimulus.v`** formulierten Slave Mode-Zugriffe: Ein Lesezugriff auf Adresse 0 (Ergebnis auf **`DATAOUT`** ist 0xDEADBEEF) gefolgt von einem Schreibzugriff auf diese Adresse (mit dem neuen Wert 0x87654321 auf **`DATAIN`**). Das als unterstes Signal dargestellte Register **`outreg`** übernimmt zu diesem Zeitpunkt den neuen Wert. Ein abschließender Lesezugriff auf die Adresse 0 liest **`outreg`** (und damit den neuen Wert) auch auf **`DATAOUT`** aus.

Ins Waveform-Fenster gezogene Signale werden erst ab dem Moment Ihres Eintreffens dargestellt, eine Vorgeschichte wird nicht aufgezeichnet. Wenn also nach dem Start der Simulation weitere Signale angezeigt werden sollen, werden diese in das Waveform-Fenster gezogen und anschließend die gesamte Simulation neu gestartet. Dies geschieht komfortabel innerhalb von VirSim durch Auswählen des Punktes Re-Exec aus dem Menü Sim im Simulationskontrollfenster. Man beachte, daß dabei alle anderen Einstellungen (z.B. der im Waveform-Fenster dargestellte Ausschnitt) nicht verändert werden. Die Anzeige wird lediglich um die neuen Signale erweitert.

Um die Signalverläufe zu drucken, wählen Sie durch geeignete Manipulation der Vergrößerung und des Ausschnitts im Waveform-Fenster zunächst den zu druckenden Zeitbereich aus. Hier brauchen Sie noch nicht auf Lesbarkeit der Signalverläufe zu achten, die Vergrößerung des Ausdrucks wird später getrennt eingestellt. Nun wählen Sie im Waveform-Fenster aus dem Menü File den Punkt Print aus. Daraufhin öffnet sich ein Dialog (Abbildung 2.8), in dem verschiedene Einstellungen gemacht werden können. Wählen Sie im Bereich Print Options als Standard Sheet Size bitte A4 aus. Für die Vergrößerung des Ausdrucks ist der Schieberegler Time Slices im Bereich Multi-Page Options relevant. Die hier eingestellte Zahl bestimmt die Anzahl von Druckseiten, auf

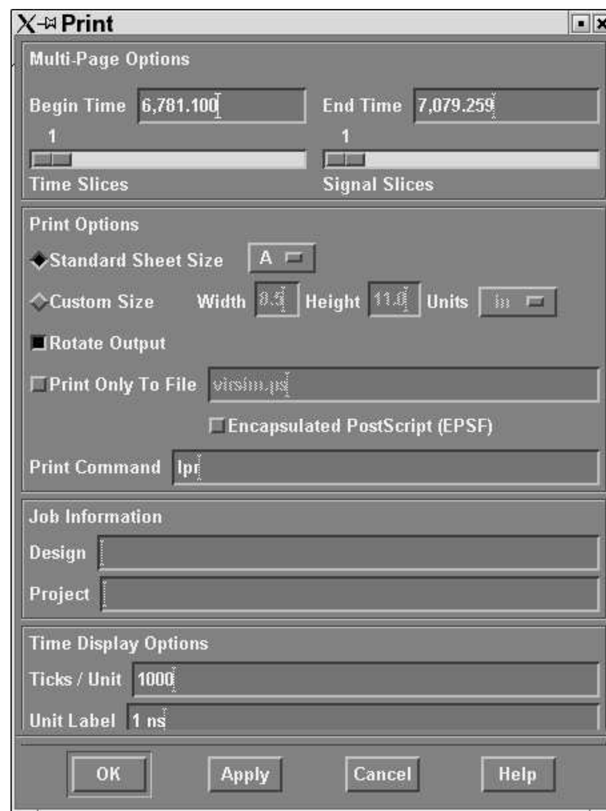


Abbildung 2.8: Drucken von Signalverläufen

die das im Waveform-Fenster gewählte Zeitintervall tatsächlich ausgedruckt wird. Durch Klicken des OK Knopfes wird der Druckauftrag abgeschickt und der Dialog geschlossen. Wenn Sie in dem Dialog die Option Print Only To File ankreuzen, können Sie in dem nebenstehenden Eingabefeld den Namen einer Datei eintragen, in die das Druckbild im PostScript-Format geschrieben wird. Eine solche Datei kann als Anhang an eine der Abgabe-E-Mails angehängt werden. Achten Sie bei der Erstellung einer solchen Druckdatei darauf, daß Sie die Option Encapsulated PostScript (EPSF) *ausgeschaltet* lassen, die Option Rotate aber eingeschaltet wird. Sie können das Ergebnis auf der Unix Shell-Kommandozeile mit dem Befehl **kghostview Dateiname.ps** ansehen und beurteilen. Achten sie auch hier darauf, daß View → Paper Size → A4 angewählt ist. Bitte geben Sie nur solche Dateien ab, bei denen Sie auf diese Weise die Lesbarkeit (Signalnamen, Wellenformen) sichergestellt haben.

Falls der Signalverlauf als Abbildung in ein anderes Dokument (z.B. für die Endabgabe) eingebunden werden soll, aktivieren Sie zusätzlich noch die Option Encapsulated PostScript (EPSF).

Zum Beenden einer VirSim Sitzung wählen Sie aus dem Menübalken den Knopf Exit. VirSim fragt Sie dann, ob die aktuelle Fensteranordnung, Signalauswahl, etc. abgespeichert werden soll. Wenn Sie hier bejahen, können Sie den Namen einer **.cfg**-Datei angeben, in der diese Daten abgespeichert werden sollen (voreingestellt ist der Name **default-**{**lay,sim**}**.cfg**). Mit einer solchen Datei kann bei einem späteren Neustart von VirSim durch Laden dieser Datei (manuell im Simulationskontrollfenster aus dem Menü File den Punkt Load Configuration anwählen, der Entwurfsfluß erledigt das aber bei vorhandener Datei automatisch für Sie) die alte Arbeitsumgebung auf einen Satz wiederhergestellt werden. Alle Signalverläufe sind aber grau (undefiniert), da der eigentliche Simulationslauf in der neuen Sitzung noch nicht stattgefunden hat.

VirSim hat eine sehr nützliche Online-Hilfe (in den Fenstern jeweils im Menü Help). Falls Sie Fragen haben, die über diese Kurzeinführung hinausgehen, lohnt es sich dort zu Stöbern.

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
```

Bitte beachten sie, daß die im Praktikum tatsächlich verwendeten Speicheradressen im Bereich 0x0F000000-0x0FFFFFFF liegen müssen, da die reale Hardware nur diesen Bereich ansprechen kann. Dies wird in den `stimulus.v` Dateien durch das Makro `` `MASTER_BASE `` realisiert.

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.

2.4 Compilierung

Nachdem Ihre Schaltung nun in der Simulation funktioniert, muß das Verilog Modell synthetisiert und auf die Zieltechnologie abgebildet werden.

Wenn Sie das Synthesewerkzeug Synplify "von Hand" bedienen wollen, starten Sie es durch Eingabe des Kommandos `make synplify` auf der Unix Kommandoebene. Das sich öffnende Fenster ist in Abbildung 2.9 dargestellt.

Beim ersten Start erscheint eine Lizenzvereinbarung, welche Sie mit YES beantworten und ein Fenster Resource Center Options, welches Sie einfach mit OK bestätigen. Schließen Sie den 'Tip of the Day' durch Klicken auf den OK Knopf.

Klicken Sie nun im Menü Run auf Run TCL Script.... In der sich öffnenden Dateiauswahl klicken sie auf die Datei `user.tcl` im gewünschten Verzeichnis, dann auf Open. Bitte achten Sie darauf, daß Sie tatsächlich die Datei für das gewünschte Projekt laden: Für alle Ihre Projekte heißt die entsprechende Synplify-Skriptdatei immer `user.tcl`. Die Projekte liegen aber in unterschiedlichen Verzeichnissen (jeweils von `mkslave` oder `mkmaster` neu angelegt).

Nach der Auswahl werden Sie gefragt, ob Sie das Projekt speichern möchten. Verneinen Sie hier *in jedem Fall* (Cancel), sonst verwenden verschiedene Teile des Werkzeugflusses unterschiedliche

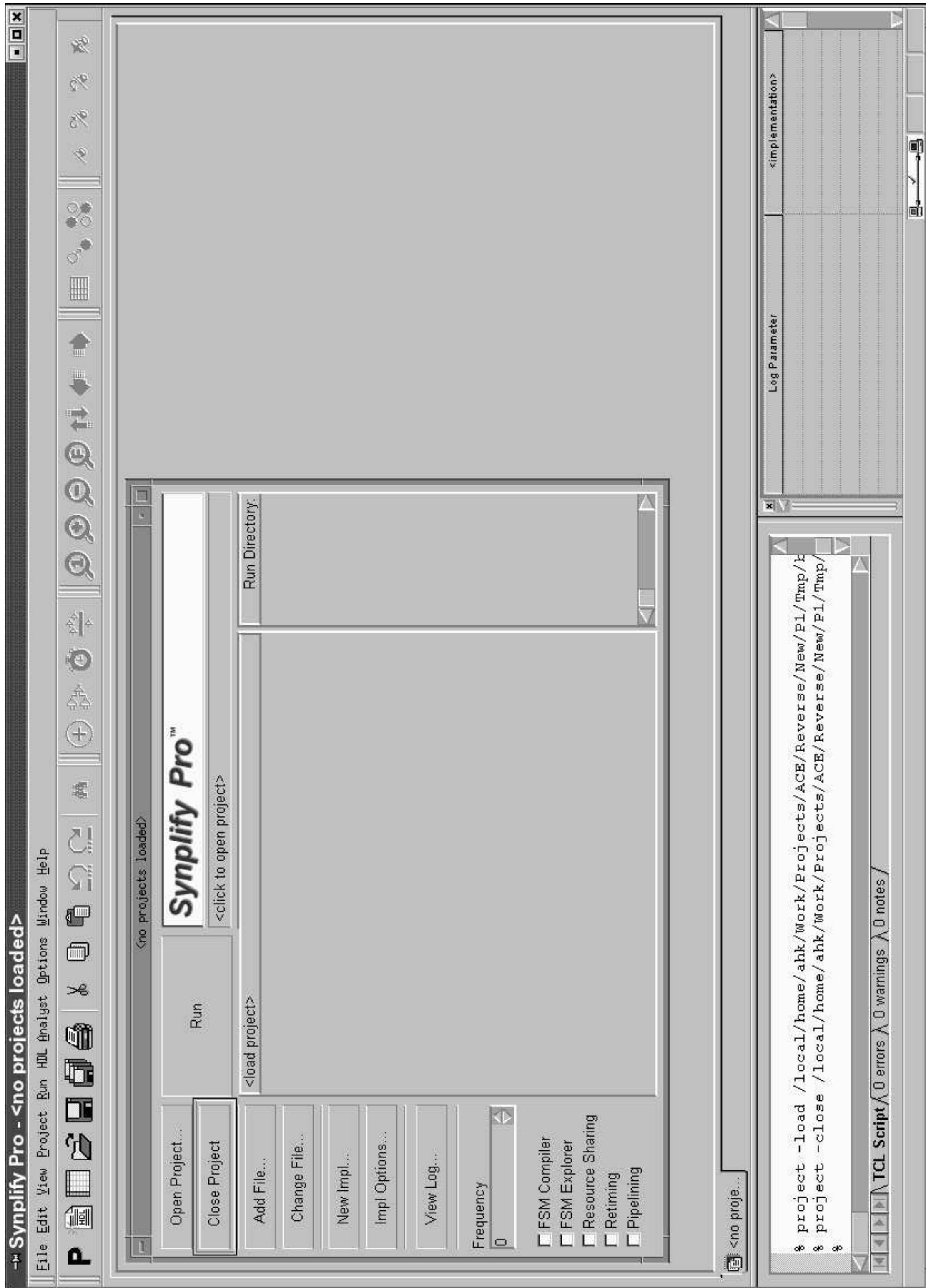


Abbildung 2.9: Startfenster von Synplify

Projektstände, was zwangsläufig zu Fehlern führt. Beantworten Sie die folgende Frage nach der Fortsetzung des Programmlaufes mit Yes.

Der Synthesprozess startet nun automatisch. Synplify gibt Ihnen Statusmeldungen über den Lauf im Textfeld rechts oben. Nach Ende des Laufes werden im Textfeld links unten verschiedene Meldungen angezeigt. Die Kategorien Fehler, Warnung, oder Bemerkung können durch Klicken des Reiters Messages und setzen eines entsprechenden Filters Set Filter... angewählt werden. Falls auf den ersten Blick nichts zu sehen ist, mit dem Rollbalken etwas nach oben blättern!

Durch einen Doppelklick auf eine Meldung öffnet sich die entsprechende Datei im internen Texteditor. Nach Korrektur kann das Editorfenster durch Doppelklick auf den linken Kasten in der Titelzeile geschlossen und ein neuer Syntheseversuch durch Klicken auf den großen Run Knopf unternommen werden. Sie brauchen sich nur um Fehler und Warnungen in **`user.v`** zu kümmern. Ignorieren Sie ruhig alle Warnungen, die anderen Modulen entstammen.

Nach dem Abschluß der Synthese werden Ihnen die Projektbestandteile angezeigt. Durch Klicken auf den kleinen verilog Ordner bekommen Sie eine Liste aller Verilog Dateien in ihrem Projekt. Änderungen sollten Sie *nur* in **`user.v`** vornehmen! Die Datei kann durch Doppelklick in einem einfachen Texteditor geöffnet werden. Für größere Änderungen ist aber die Bearbeitung mit einem mächtigeren Editor (wie kedit, emacs oder nedit) außerhalb von Synplify sinnvoll.

Nach erfolgreicher Synthese sind die Verilog Modelle in eine EDIF-Netzliste verwandelt worden, die nun mit den Xilinx ISE Werkzeugen weiterverarbeitet werden kann.

Nach ersten Experimenten mit Synplify raten wir Ihnen aber, statt der bisher beschriebenen mausorientierten Oberfläche einen automatischen Entwurfsfluß zu verwenden. Dieser kommt auch ohne GUI aus (nützlich z.B. für das entfernte Arbeiten von zu Hause aus). Alle anderen Schritte (bis hin zum ausführbaren Programm) sind auf diese Weise automatisiert.

Tippen Sie dazu auf Unix-Kommandoebene einfach den Befehl **`make`** ein. Nun werden automatisch Ihre C-Quellen aus **`main.c`** übersetzt, die Verilog-Modelle synthetisiert, die EDIF-Netzliste mit den ISE Werkzeugen auf das FPGA abgebildet, und alle verschiedenen Teile wieder zusammengefügt. Falls alles glatt ging, können Sie mit dem Kommando **`make linux`** Ihre gesamte Anwendung (Hardware und Software) auf den adaptiven Rechner ML310 laden. Anschließend wird das Betriebssystem Linux auf dem ML310 gestartet und nach kurzer Zeit erscheint der Login-Prompt **`ml310 login:`**. Melden Sie sich mit ihren Praktikumszugangsdaten an und führen Sie **`./main`** aus, ihr Programm wird auf dem ML310 gestartet. Wenn sie das ML310-Linux wieder verlassen möchten, geben sie einfach **`Strg-A Strg-A`** ein, das System braucht nicht heruntergefahren zu werden, da es keinen lokalen Dateisysteme besitzt.

Das Kommando **`make linux`** darf nur auf den Rechnern **`dwalin, bofur, bifur`** oder **`bombur`** eingegeben werden und startet das jeweils danebenstehende ML310. Sollte der Login-Prompt nicht erscheinen und das ML310 mit der Ausgabe **`Entering TERMINAL mode - Escape character is '^A'`** stehenbleiben, verlassen sie das System mit der Tastenkombination **`Strg-A Strg-A`** und versuchen sie noch einmal **`make linux`**. Sollte auch das nicht helfen, drücken sie den Reset-Knopf an der Stirnseite des blauen ML310-Gehäuses mindestens 5 Sekunden lang.

Wenn beim durch **`make`** automatisierten Entwurfsfluß Fehler auftreten, bricht der Prozess nach Ausgabe einer Meldung ab. Falls Fehler bei der C-Compilierung auftreten, ist die Meldung der Fehlerbericht des C-Compilers. Bei Synthesefehlern kommt nur eine wenig aussagekräftige Meldung, hier hilft der Blick in die Datei **`Simple/user.srr`**: Fehler haben ein führendes **`@E`**, Warnun-

gen ein `@W` und Kommentare ein `@I`. Die ISE Werkzeuge sollten Ihnen keine Fehler produzieren. Falls doch, bitte dem Assi Bescheid geben. Für Neugierige: Die Log-Dateien stehen im Unterverzeichnis `Simple`.

2.5 Post-Layout-Simulation

Durch Eingabe des Kommandos `make laysim` können Sie nach erfolgreicher Synthese und Plazierung eine Post-Layout-Simulation (also mit realen Timing-Daten) starten. Nach einigen automatisch ablaufenden Vorbereitungsschritten startet wie gewohnt VirSim. Fehler (Error) bei der SDF-Annotation (Eintragen der realen Hardware- Signallaufzeiten in die Gattermetzliste) können Sie getrost ignorieren. Leider ist Ihre gewohnte Modul-Hierarchie mit den bekannten Signalnamen durch den ISE-Implementierungsvorgang arg durcheinander geraten.

Damit Sie trotzdem sehen können, was sich abspielt, wird Ihnen eine Standard Signalkonfiguration im Waveform-Fenster vorgegeben. Folgende Signale sind dabei für Sie von Interesse:

PLBCLK : Der zentrale Takt.

ADDRESS : Der Adreßbus. Achtung: der Bus ist aus technischen Gründen bitinvertiert! (Bit 0 ≈ Bit 31, Bit 1 ≈ Bit 30 usw.) Für Sie sind nur die Bits 0-23 interessant.

ADDRESSED : Eine ‘1’ auf diesem Signal zeigt einen Zugriff auf Ihr Modul an.

DATAIN : Der Eingabedatenbus Ihres Moduls (CPU → RC). Achtung: der Bus ist aus technischen Gründen bitinvertiert! (Bit 0 ≈ Bit 31, Bit 1 ≈ Bit 30 usw.) Für Sie sind nur die Bits 0-31 interessant.

DATAOUT : Der Ausgabedatenbus Ihres Moduls (RC → CPU).

LWRITE : Schreibsignal. Ist ‘1’, wenn Daten auf `DATAIN` in die RC geschrieben werden. Bei ‘0’ werden Daten aus der RC auf `DATAOUT` ausgegeben. Nur gültig im Zusammenhang mit `ADDRESSED`.

Abbildung 2.10 zeigt die Post-Layout-Variante der in Abbildung 2.7 in RTL-Simulation betrachteten Signalverläufe. Zunächst findet ein Lesezugriff auf Adresse 0 statt (`LWRITE=0`, Ausgabe von 0xDEADBEEF von RC auf `LD`), dann wird der Wert 0x87654321 auf Adresse 0 geschrieben (`LWRITE=1`). Abschließend wird der neue Wert wieder von Adresse 0 gelesen.

2.6 Praktische Erprobung

Wie oben schon beschrieben erfolgt die praktische Erprobung auf den Rechnern `dwalin, bofur, bifur` oder `bombur`. Dabei kann die gesamte Anwendung (Software und Hardware) nach erfolgreichem `make linux` und Login durch das Kommando `./main` auf dem ML310 gestartet werden.

Die Hardware kann aber auch einzeln nach erfolgreichem `make download` mittels des Werkzeugs `xmd` getestet werden. Abschnitt 3.3.2 beschreibt die entsprechende Vorgehensweise.

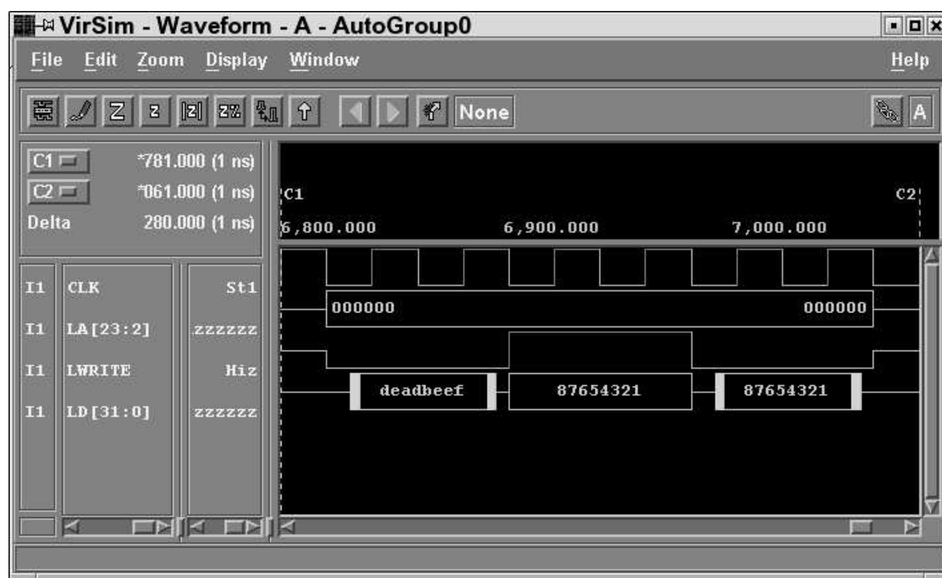


Abbildung 2.10: Slave-Mode Zugriffe in Post-Layout-Simulation

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-2[1] 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.

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

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. 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 rekonfigurierbare 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 gröberer Granularïtät:

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 Stark im Kommen für arithmetische Anwendungen. Einige Anbieter/Anwender sind Elixent (4b ALUs), Broadcom (8b ALUs) und Chameleon (32b ALUs). Solche Bausteine werden auch gelegentlich als *network processors* bezeichnet, da sie auf den Einsatz in Netzwerk- und Kommunikationssystemen ausgelegt sind.

Komplette Prozessoren Noch recht selten. Ein Beispiel ist die MIT RAW Architektur (jeder Funktionsblock ist ein MIPS-artiger RISC mit FPU und eigenen Caches).

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.

Bindungsintervall

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.

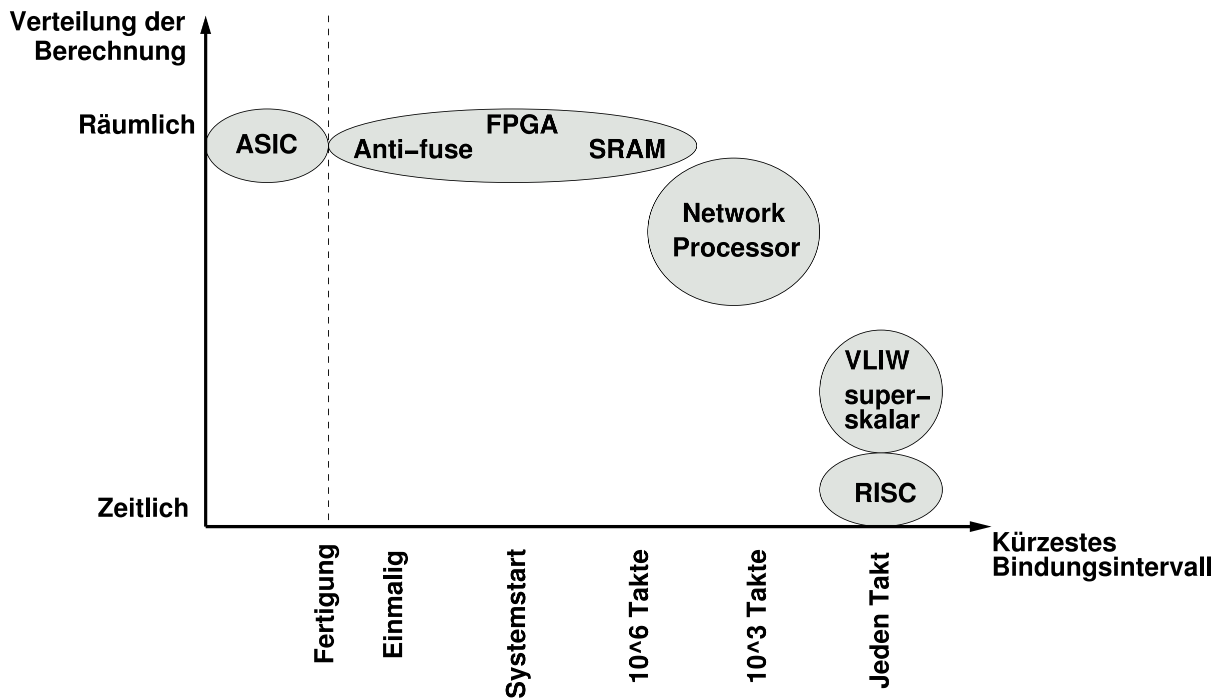


Abbildung 3.2: Berechnungsverteilung und Bindungsintervalle

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.

10^6 **Prozessortakte** Hier sind auch im laufenden Betrieb Anpassungen möglich. Es kann praktikabel werden, für jedes Programm ein oder mehrere angepasste Hardware-Funktionen bereitzustellen.

$10^3 - 10^2$ **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 Transistoren 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 einen 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 Komponente (im folgenden *RC* 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.

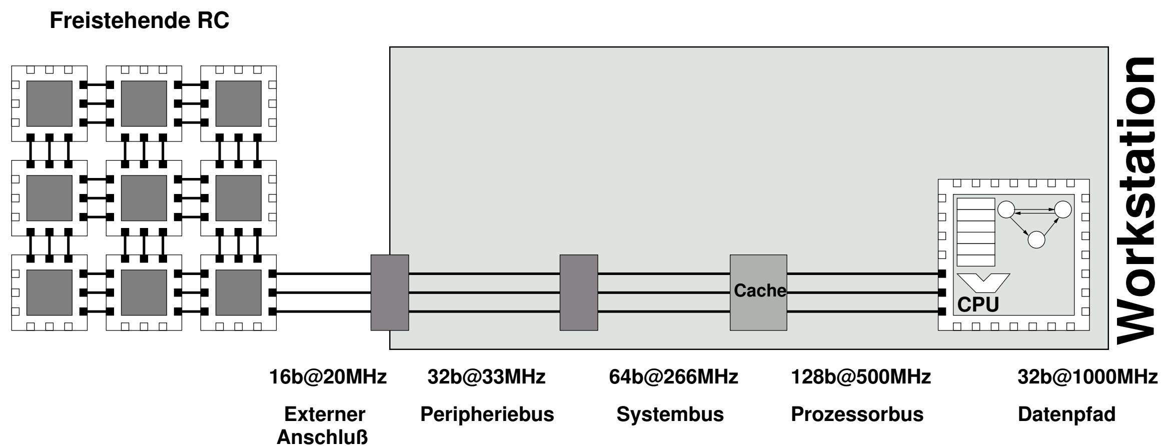


Abbildung 3.3: Freistehende RC

Freistehende RC

System mit sehr großem RC 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 RC ('stand-alone unit') bezeichnet.

Ein typischer Vertreter dieser Gattung wird im wesentlichen für die ASIC-Emulation und als Beschleuniger für HDL-Simulationen genutzt. Eine RC Kapazität von bis zu 112 Millionen Gatter steht zur Verfügung. Die Anbindung an die CPU erfolgt über das auch bei Festplatten verwendete SCSI-Protokoll. Die RC allein wiegt 1.1t und benötigt einen 12KW 350V Drehstromanschluß.

Auf diese Weise lassen sich zwar sehr große rekonfigurierbare Flächen realisieren, aber (neben den anderen praktischen Nachteilen ...) beschränkt die hohe Latenz der Kommunikation zwischen CPU und RC die Art der realisierbaren Anwendungen, wie das folgende hypothetische Beispiel zeigen soll: Angenommen, die RC kann pro Takt die Arbeit von 40 Prozessorbefehlen leisten. Falls aber jeder Datentransfer zwischen RC 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 RC

Es ist daher sehr sinnvoll, die Kommunikationslatenz zwischen RC 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 RC direkt in den Rechner und schließt sie dort an den *Peripheriebus* (oft PCI, aber noch sind auch 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 (einfache Steckkarte) von Standardrechnern. Bei dieser Lösung wird von einer angeschlossenen RC ('attached processing unit') gesprochen.

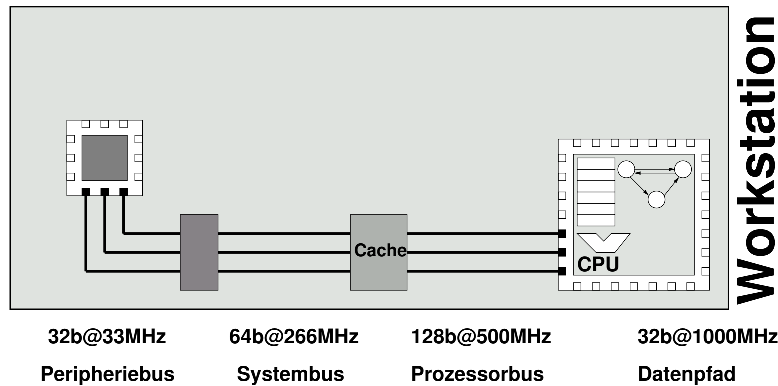


Abbildung 3.4: Angeschlossene RC

Mit der CPU gleichberechtigte RC

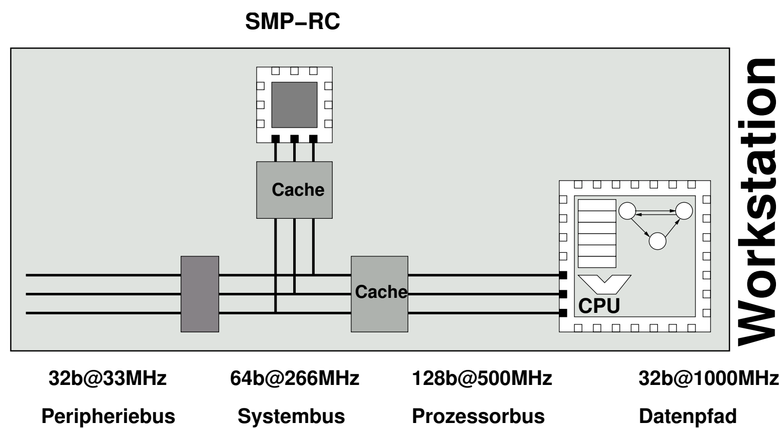


Abbildung 3.5: RC gleichberechtigt mit der CPU

Der Aufbau von adaptiven Rechnern mit noch engerer Kopplung von RC und CPU wird nun zunehmend schwieriger. Eine Möglichkeit sieht die RC 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 400 MHz), den kürzeren Latenzen (im Bereich von 4-40 Takten, also 10-100ns) und der größeren Busbreite (128b Daten + 64b Adressen). Ein einfacheres System dieser Gattung (mit nur 133MHz Bustakt) läßt sich auch noch ohne speziell gefertigte Chips aufbauen. Die Protokolle für die Interprozessorkommunikation sind zwar nicht trivial (es müssen unter anderem die unabhängige Caches von RC und CPU kohärent gehalten werden), lassen sich aber mit der nötigen Geschwindikeit auch noch in FPGAs realisieren. In vielen Fällen wird die Realisierung eines solchen Rechners aber den Entwurf und die Fertigung einer neuen Basisplatine (Motherboard) für die Komponenten CPU, RC, Speicher, und Peripherieschnittstellen erfordern.

RC-Koprozessor

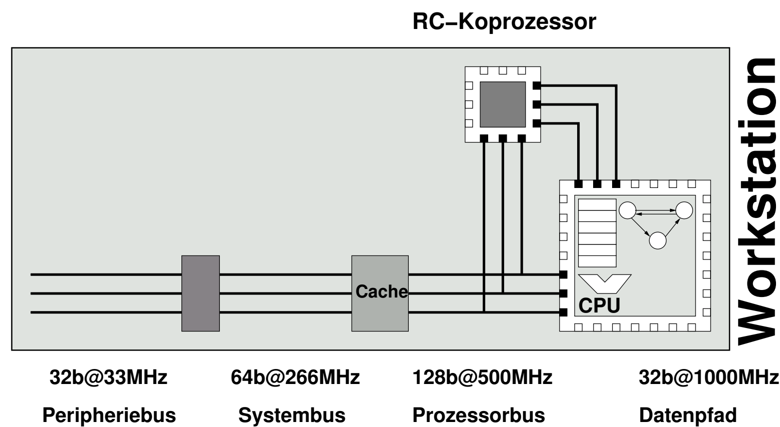


Abbildung 3.6: RC als Koprozessor

Die Anbindung der RC zusammen mit der CPU an einen *gemeinsamen Cache* verkürzt die Latenzen in der Regel noch weiter (Bild 3.6). Bei dieser Kombination agiert die RC als echter Koprozessor für die CPU. Die Kommunikation zwischen RC und CPU kann dann in 10 Prozessortakten (25ns bei angenommenen 400MHz CPU-Takt) vorgenommen werden, ein Speicherzugriff über den Cache kann im Erfolgsfall (cache hit) in ähnlicher Zeit bewältigt werden. Zur Realisierung einer solche Architektur können aber keine Standardchips mehr verwendet werden Die meisten modernen Mikroprozessoren haben den Cache (L1 und L2) auf dem Prozessor-Chip integriert. Es müssten also auch die RC zusammen mit der CPU und dem Cache auf dem gleichen Chip untergebracht werden. Praktisch wird dies vereinfacht dadurch, daß jede Komponente für sich durch schon bestehende Designs (z.b. fertige CPU und FPGA-Cores) implementiert werden kann. Lediglich das Verbindungsnetz und die Kommunikationsprotokolle zwischen den Komponenten müssen neu entworfen worden. Solche Bausteine für adaptive Rechner werden schon heute von einigen Herstellern angeboten.

RC-Funktionseinheit in CPU

Auch eine noch engere Integration ist denkbar: Die RC könnte als *Funktionseinheit* 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 RC 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 RC deutlich aufwendigere Berechnungen realisierbar waren, wurden diese durch die sehr niedrige Kommunikationsbandbreite ‘ausgehungert’ (bekamen nicht genug Daten). Neuere Experimente mit RC-Funktionseinheiten versuchen, dieses Problem durch Bereitstellen RC-eigener Speicherschnittstellen zu umgehen. Dabei müssen aber die Interaktionen zwischen den Speicherzugriffen

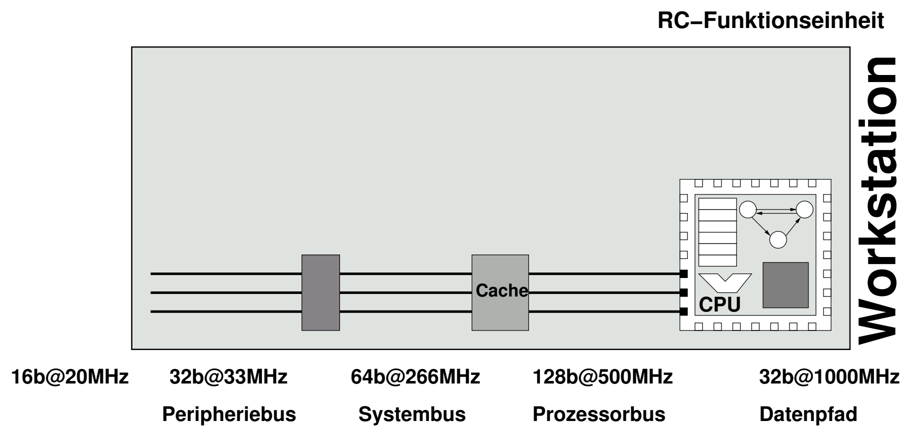


Abbildung 3.7: RC als Funktionseinheit in CPU

des Prozessors und der RC-Funktionseinheit sorgfältig koordiniert werden. Kommerziell ist noch kein Vertreter dieser Gattung von adaptivem Rechner verfügbar.

Weitere Systemkomponenten

Obwohl wir uns an dieser Stelle nur auf die Hardware-Aspekte konzentriert haben, besteht ein adaptives Rechen*system* 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 RC hinzu. Beispiele für solche Operationen sind:

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

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 RC 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 vorangegangen 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 (Moores 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 taufrisch, 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 ...

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 mit 300 Millionen Transistoren. Was aber wird bisher mit dieser gewaltigen Chip-Fläche angefangen? Die klassische Prozessor-Architektur zeigt sich hier wenig erfinderisch: Immer größere Caches (z.B. 1.5MB L1 beim HP PA-8700, damit läuft der SPEC Benchmark komplett aus dem Cache), höhere Integration (z.B. Speicher-Controller on-chip) oder gar mehrere ausgewachsene Prozessoren auf dem gleichen

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

Tabelle 3.1: Gatterkomplexitäten einiger Anwendungen

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 RC-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 RC 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 genausogut auch auf einem Standardprozessor ausgeführt werden. Nur die wirklich zeitkritischen Teile des Algorithmus müssen als RC-Konfiguration ausgelagert werden. Erste Bausteine die diese Kombination von CPU und RC auf einem Chip vereinen, setzen diese Ideen mittlerweile praktisch in die Tat um.

3.1.7 Beispiel: ML310

Als Testplattform für Ihre Programmierungen stellen wir Ihnen im Praktikum das Entwicklungsboard *ML310* von der Firma *Xilinx* zur Verfügung. Neben einem Virtex-II Pro FPGA mit zwei integrierten PowerPC-Kernen bietet diese Platine eine Fülle von Peripheriegeräten und Anschlüssen und ist daher universell einsetzbar. Man kann sich das ganze vereinfacht als ATX-PC vorstellen, wobei der Prozessor und die Northbridge (RAM, Local Bus und PCI-Controller) durch das FPGA ersetzt wurden. Für den Betrieb als adaptiver Rechner im Praktikum benötigen wir nur einige wenige dieser Features. Das FPGA, das DDR-RAM, das Parallel-Cable 4 (PC4) zur Konfiguration des FPGA sowie die seriellen Schnittstellen und der Ethernet-Controller sind für uns die wichtigsten Komponenten.

Das ML310 im ATX-Gehäuse

Das ML310 befindet sich in einem ATX-Gehäuse direkt neben Ihrem Praktikumsrechner. Bild 3.8 zeigt die Anbringung des Boards im Gehäuse.

Die Programmierung des FPGA (*Konfiguration*) erfolgt über das PC4, welches mit Ihrem Praktikums-PC verbunden ist. Es übersetzt die vom Parallel-Port des PCs kommenden Daten in JTAG-Programmierbefehle. JTAG (Joint Test Action Group) ist ein serielles Protokoll zum Programmieren, Auslesen und Debuggen von Hardware. Als Speicher für die Software und die zu verarbeitenden Daten dienen das 256 MB große DDR-RAM-Modul.

Das an der COM1-Schnittstelle des PCs angeschlossene serielle Kabel ist mit dem Onboard-COM-Interface des ML310 verbunden. Ausgaben des ML310 können so auf dem PC in einem Terminal-Fenster dargestellt werden. Das Onboard-COM-Interface ist nicht direkt mit dem FPGA verbunden, sondern hängt an einer Southbridge (Standard-Peripherie in einem Chip), die vom FPGA wiederum über den PCI-Bus angesteuert wird.

Betriebs-Modi

Je nach Zusammenspiel zwischen CPU und RC (rekonfigurierbare Hardware-Komponente) unterscheidet man bei einem adaptiven Rechner drei Betriebs-Modi:

1. *Software-Mode*,
2. *Slave-Mode* und
3. *Master-Mode*.

Im **Software-Mode** entspricht der adaptive Rechner einem Standard-Rechner, da die RC hier gar nicht benutzt wird. Bei Programmen, die diesen Modus verwenden, spricht man auch von *reinen Software-Lösungen*.

Im **Slave-Mode** wird die RC genutzt, aber die Programmkontrolle bleibt weiterhin bei der CPU. Einzelne Datenworte (typischerweise zu je 32 Bit) werden der RC übergeben, diese rechnet auf den Daten und stellt der CPU das Ergebnis zur Verfügung, das diese dann zurückliest. Häufig sind während eines Programmablaufs sehr viele dieser Datentransfers nötig, daher haben Slave-Mode-Lösungen einen hohen Kommunikations-Overhead.

Im **Master-Mode** dagegen wird die Programmkontrolle an die RC abgegeben. Die RC kann ohne Umwege über die CPU auf den Systemspeicher zugreifen, wodurch die Datenverarbeitung deutlich beschleunigt werden kann. Die Programmierung der RC ist hier jedoch umfangreicher, da sie eine State-Machine zur Programmkontrolle enthalten muss. Sobald die RC ihre Berechnungen abgeschlossen hat, teilt sie dies der CPU mit, häufig über ein Interrupt-Signal.

HW/SW-Schnittstellen

Der PowerPC-Kern im Virtex-II Pro FPGA wird mit 300 MHz getaktet und hat drei Interfaces, über die ein Datenaustausch mit Hardware-Modulen möglich ist: *DCR* (Device Control Register), *DSOCM* (Data-Side On-Chip Memory) und *PLB* (Processor Local Bus).

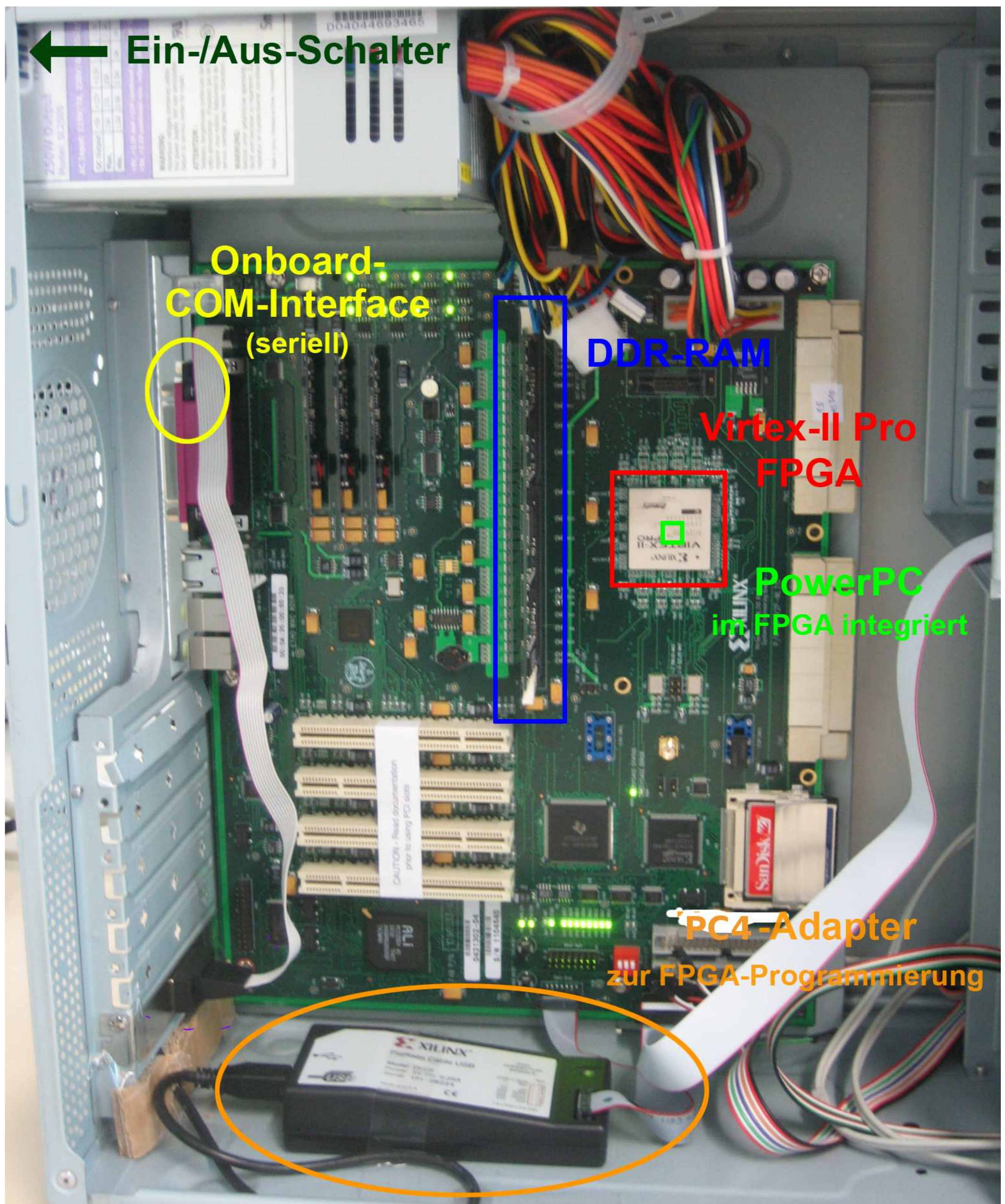


Abbildung 3.8: Xilinx ML310 - Unterbringung in einem ATX-Gehäuse

- Das **DCR**-Interface ist recht überschaubar, klein und einfach anzuwenden; eine Datenübertragung dauert jedoch vergleichsweise lange.

- Über **DSOCM** sind sehr schnelle Datenübertragungen möglich. Dieses Interface wird vom PowerPC typischerweise für den schnellen Zugriff auf das im FPGA befindliche Block-Select-RAM (BRAM) benutzt, daher auch die Bezeichnung „On-Chip-Memory“. Diese Speicher sind für unsere Zwecke jedoch zu klein.

- Der **PLB** erlaubt ebenfalls sehr schnelle Übertragungen und wird als effiziente Schnittstelle zur On-Chip-Peripherie und zum DDR-RAM genutzt. Er läuft immer mit 100 MHz Taktfrequenz. Über an den PLB angeschlossene Controller und Bridges kann die CPU auch mit Peripherie außerhalb des FPGA kommunizieren. Den seriellen Onboard-COM-Anschluss des ML310 z.B. erreicht der PowerPC über die Verbindung
  PPC $\overset{PLB}{\leftrightarrow}$ PLB/OPB-Bridge $\overset{OPB}{\leftrightarrow}$ OPB/PCI-Bridge$\overset{PCI}{\leftrightarrow}$ South Bridge $\leftrightarrow$ COM1
  (siehe Bild 3.9).

  Der OPB ist ähnlich zum PLB ein schneller On-Chip-Bus. Er ist jedoch nicht direkt mit der CPU verbunden, dafür fällt sein Bus-Interface etwas einfacher aus als das des PLB.
  Den PLB werden Sie im Praktikum für den Datentransfer zwischen CPU und Ihrer Hardware im Slave-Mode benutzen. Deshalb wird Ihre Hardware ebenfalls immer mit 100 MHz Takt betrieben. Zwischen Ihrem Design und dem PLB hängt dabei noch ein weiteres Modul, welches das PLB-Protokoll in ein einfacheres übersetzt, das Ihnen als leicht zu verwendende Schnittstelle des `user`-Moduls zur Verfügung steht. Im Master-Mode greift Ihr IP-Core mit Hilfe von MARC (s.u.) direkt auf das DDR-RAM zu, unter Umgehung des PLB (siehe grüner Pfeil in Bild 3.9).

- Die **Memory Architecture for Reconfigurable Computers (MARC)** ist zwar keine Prozessorschnittstelle, aber Ihre Hardware wird MARC nutzen, um mit dem DDR-Speicher zu kommunizieren. MARC ist eine Hardware-Umgebung, die ähnlich einem Betriebssystem in Software nun RC-Anwendungen verschiedene Basisdienste bereitstellt. Dazu gehören insbesondere leicht zu verwendende Schnittstellen sowohl für reguläre (z.B. durch ein Array) oder irreguläre Speicherzugriffe. Die Details der Hardware (z.B. Zugriffsprotokoll für DDR-Speicher) und die optimale Zugriffsstrategie (Caches, Burst-Transfer, Read-Ahead etc) laufen dabei hinter den Kulissen ohne Zutun des Benutzers ab und müssen auch nicht mehr extra von ihm entworfen werden. Abschnitt 3.3.3 zeigt die Benutzung von MARC in einer praktischen Anwendung.

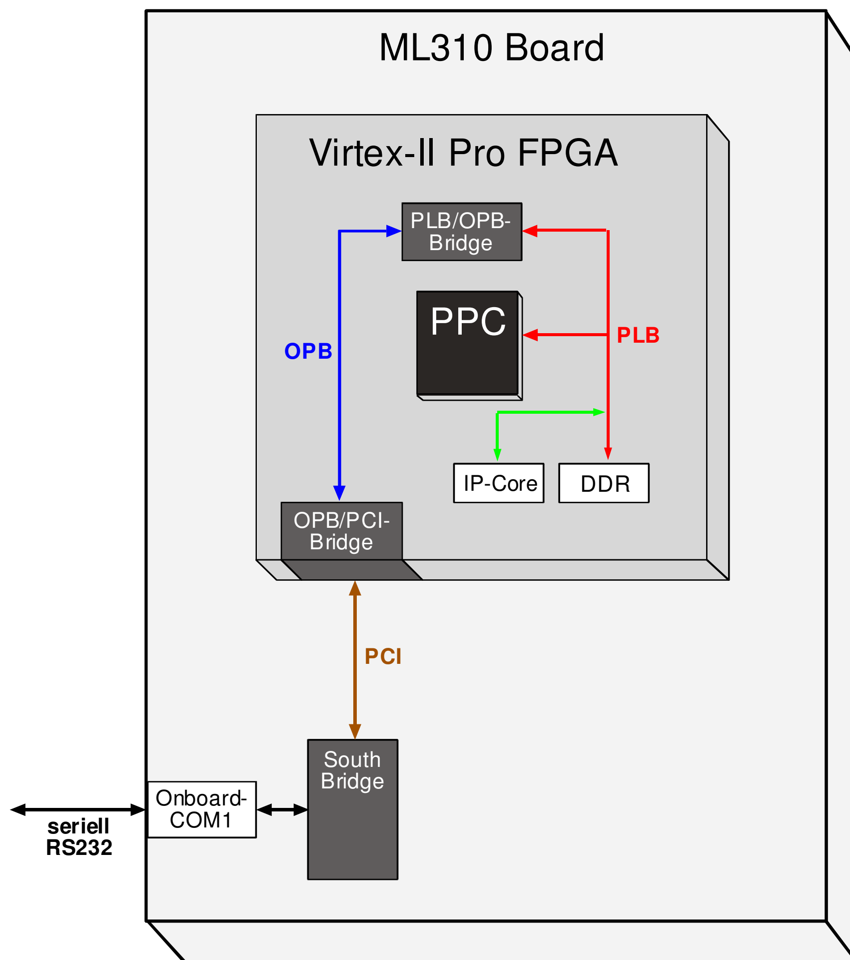


Abbildung 3.9: Busse auf dem ML310 mit Anbindung eigener Hardware (IP Core)

Datentransfer zwischen PC und ML310

Das auf dem ML310 laufende Linux-System bindet das Verzeichnis `/scratch/acsprakNN` (NN = Ihre Gruppennummer) als Home-Verzeichnis über Network-File-System (NFS) ein. Dort finden sie nach erfolgreichem `make linux` das ausführbare Binärprogramm `main`. Eventuelle Testdateien müssen Sie selbst nach `/scratch/acsprakNN` kopieren. Geben sie dazu *am PC* z.B. `cp foo /scratch/acsprak01` für die Gruppe 1 ein. Nach erfolgter Berechnung im PowerPC bzw. in der FPGA-Hardware werden die Ergebnisdateien (`fwrite()`) ebenfalls nach `/scratch/acsprakNN` geschrieben. Bitte bedenken Sie, daß alle Dateien unter `scratch` als temporär und lokal anzusehen sind, d.h., daß sie ihre Dateien *dauerhaft* nur auf dem *PC-Home-Verzeichnis* ablegen können. Lokal bedeutet, daß jeder PC ein eigenes `/scratch` verwendet und die jeweiligen Verzeichnisse und Dateien nicht repliziert werden. Kopieren sie also alle Dateien, die sie behalten wollen, *am PC* z.B. mit `cp /scratch/acsprak01/foo ~` (Gruppe 1) zurück in Ihr PC-Home-Verzeichnis.

Auf PC-Seite läuft nach dem Ausführen von `make linux` ein Terminal-Programm (`xc`), welches mit der Konsole des Linux-Systems auf dem ML310 verbunden ist und u.a. auch alle Ausgaben der ML310-Software (`printf()`) anzeigt.

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 (auch *ACS*, adaptive computing system genannt). 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 Adaptionsfähigkeit kann hier für jedes Problem das am besten geeignete Rechenmodell zum Einsatz kommen.

Wir wollen uns in den folgenden Abschnitten mit zwei sehr unterschiedlichen Programmiermethoden befassen: Zunächst erläutern wir die Programmierung mittels direkter Beschreibung der auf der RC-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 RC zu realisieren. Diese Art der Programmierung setzt aber detailierte Kenntnise der ACS-Hardware und des Hardware-Entwurfs im allgemeinen voraus. Neben verschiedenen technischen Problemen (z.B. unzureichendes Fassungsvermögen der RC) 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 er-

stellt 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. Ein Beispiel für einen solchen Entwurfsfluß ist der Compilerprototyp COMRADE, der aus Standard ANSI C (ohne jede Einschränkung oder Sonderkonstrukte) 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 3.10).

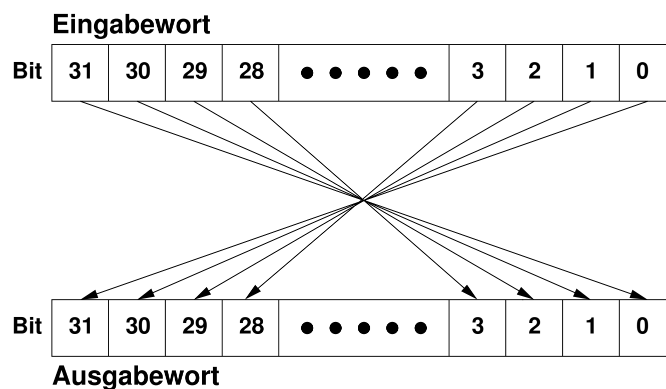


Abbildung 3.10: Funktion der Beispielanwendung

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 RC 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, Plazierung, 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 generalle 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 512Kw, 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*512)

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;
}

// Ende der Berechnung, merke Stopzeit in Ticks
stop = RTEMSIO_getTicks();

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

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

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

// Gebe Ergebnis der Zeitmessung in Mikrosekunden aus
```

```
    printf ("Zeit: %lld us\n", (stop - start)/TICKS_PER_USEC);
}
```

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 langsame serielle Konsole der ML310 Plattform 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 RC 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 RC geeignet. So ist es beispielsweise nicht sinnvoll zu versuchen, Fließkommaoperationen (Datentypen float und double) auf heutige gängige RCs zu verlagern. Die Nachbildung der dafür benötigten Recheneinheiten verlangt einfach zuviel Platz. Solche Operationen sind auf 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 RC 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 RC

Aber nur die Realisierung der logischen Funktion reicht hier nicht mehr aus. Auf irgendeine Art und Weise müssen schließlich die Daten in die RC 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 RC überträgt, diese die Berechnung ausführt, und die CPU schließlich das Ergebnis abruft. Dieses Füttern der RC mit einzelnen Daten-Happen bezeichnet man als Slave-Mode Betrieb der RC.

Schauen wir uns zunächst einmal die Hardware an, die bei diesem Vorgehen für unsere Anwendung in der RC 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 RC 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 RC vor. Ist WRITE bei gesetztem ADDRESSED dagegen Low, versucht die CPU Daten von der RC zu lesen. Wenn ADDRESSED nicht gesetzt ist, kann WRITE ignoriert werden, da die RC 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 RC ignoriert werden.

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

ADDRESS Durch diesen Bus teilt die CPU der RC während eines Lese- oder Schreibzugriffs 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-Zugriffs auf die RC können die Werte auf diesem Bus von der RC 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   // Addresse des Zugriffs (in dieser Anwendung ignoriert)
);

  // Eingänge
  input            CLK;
  input            RESET;
  input            ADDRESSED;
  input            WRITE;
  input   [31:0]   DATAIN;
  input   [23:2]   ADDRESS;

  // Ausgänge
  output  [31:0]   DATAOUT;

  // Beginn der Anwendung *************************************************

  reg  [31:0]  result ;           // Ergebnisregister
  reg  [31:0]  reversed ;         // Zwischenergebnis

  // Gebe immer (unabhängig von der Adresse) das Ergebnisregister aus
  assign DATAOUT = result;

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

  // Steuerung
  always @(posedge CLK or posedge RESET) begin
     // Initialisiere Ergebnis auf magic number für Debugging
     if (RESET) begin
        result <= 32'hDEADBEEF;
     // Schreibzugriff auf RC, neu berechnetes Zwischenergebnis übernehmen
    end else if ( ADDRESSED & WRITE) begin
        result <= reversed;
    end
  end
```

```
endmodule
```

Architektur der Recheneinheit

Wir wählen folgende Architektur für diese RC: Ein einzelnes Register result speichert das letzte Berechnungsergebnis. Dieses entsteht dadurch, daß ein von der CPU auf die RC 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 Speicherauszügen sehr leicht wieder zu erkennen ist: DEADBEEF. Man kann also als ersten Hardware-Test einen Lesezugriff von der CPU auf die RC ausführen. Wenn die erwartete "magic number" zurückgeliefert wird, so ist schon einmal die erfolgreiche Konfiguration der RC, 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 RC, 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, Plazierung und Verdrahtung mit einer maximalen Taktfrequenz von über 400 MHz lauffähig ist. Das ist für die ML310 Plattform, bei der ja die Kommunikation über den internen PLB-Bus mit 100 MHz Takt stattfindet, mehr als ausreichend. Die Größe der synthetisierten Logik ist auch recht überschaubar: Mit 45 sogenannten Slices (je zwei formen einen CLB) wird lediglich 1% aus der Gesamtzahl von 13696 der auf der RC zur Verfügung stehenden Slices ausgenutzt.

Software-Schnittstelle zur RC

Wie testet man nun den Hardware-Teil einer adaptiven Anwendung? Wie im “normalen” VLSI-Entwurf üblich, kann das HDL-Modell natürlich simuliert werden. Wir werden in Abschnitt 3.3.3 sehen, wie eine solche Systemsimulation vorgenommen werden kann. An dieser Stelle werden wir die Hardware einfach auf dem ML310 ausprobieren. Wie kann dies aber ohne den (bisher noch fehlenden) Software-Teil der Anwendung geschehen? Dafür steht uns ein Werkzeug names XMD (Xilinx Microprocessor Debugger) zur Verfügung. Es erlaubt unter anderem das Ausführen von Zugriffen innerhalb des gesamten Adressraums der CPU.

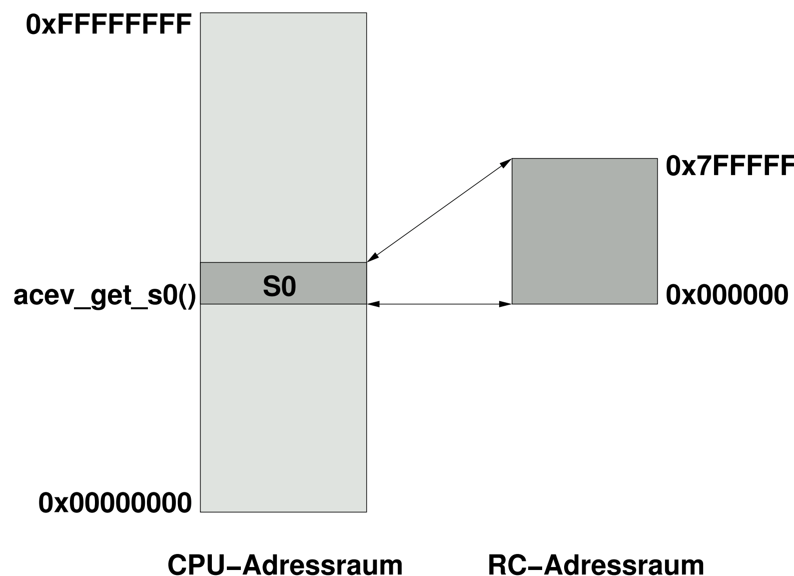


Abbildung 3.11: Speicheraufteilung im Slave-Mode

Wie hilft uns ein solches Werkzeug aber beim Hardware-Test? Die Kommunikation zwischen Soft- und Hardware beruht darauf, daß die RC-internen Register (auf dem 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 3.11 dargestellt. Dabei gilt folgendes:

- Ein CPU-Zugriff auf eine Adresse im eingeblendeten S0-Bereich wird nicht auf den Hauptspeicher ausgeführt, sondern wird auf die RC umgeleitet. Dabei wird die Differenz zwischen Zugriffsadresse und S0-Basisadresse auf den ADDRESS-Bus der RC angelegt.

- Die RC-Adressen addressieren stets 32b Datenworte (also in 4B Schritten). Das heißt, daß die niederwertigsten beiden Bits immer Null sein müssen. Auf der RC 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’h04700000 beginnt (dieser Wert kann durch die Software zur Laufzeit abgefragt werden) Ein Lesezugriff der CPU auf Adresse

32'h04800410 führt zu einem Lesezugriff auf die RC-Adresse

$$32'h04800410 - 32'h04700000 = 24'h100410$$

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

Interaktiver Hardware-Test

Betrachten wir also eine Beispielsitzung mit XMD. Bevor wir jedoch XMD starten, wird zuerst die RC durch Eingabe von `make download` auf dem PC mit der Slave-Mode Reverse-Hardware konfiguriert. Je nachdem ob Ihr Design schon übersetzt ist, kann es etwas dauern, bis der Vorgang abgeschlossen ist und die Eingabeaufforderung wieder erscheint.

Beim Start (Eingabe von `xmd` am PC) meldet sich XMD mit seinem Prompt `XMD%`. Nun müssen wir uns zunächst mit dem fernzusteuernden PowerPC-Kern verbinden. Dies geschieht durch `XMD% connect ppc hw <Enter>`. Nach einigen Statusmeldungen ist XMD bereit und der PowerPC wartet auf Befehle. Als nächstes muß die Hardware zurückgesetzt werden, der Reset wir durch Eingabe von `XMD% rst <Enter>` ausgelöst. Von nun an sollte die benutzerdefinierte Schaltung auf Anfragen reagieren. Wir prüfen zunächst, ob die Schaltung korrekt gestartet worden ist. Dies erfolgt durch Auslösen eines Lesezugriffs auf eine Adresse im S0-Bereich, bei der wir einen bekannten Datenwert zurückerwarten (unsere in Abschnitt 3.3.2 definierte 'magic number'. Da unsere Testschaltung den ADDRESS-Bus ohnehin nicht dekodiert, reicht eine beliebige S0-Adresse aus. Wir verwenden hier direkt die S0-Basis: `XMD% mrd 0x10000000 <Enter>`. In einer neuen Zeile wird die Adresse und (hinter dem '`:`') ihr aktueller Speicherinhalt hexadezimal ausgegegen, es ist der erwartete Wert 32'hDEADBEEF. Unsere Schaltung reagiert also zumindest auf Lesezugriffe im Slave-Mode korrekt. Der nächste Test soll neben der Reaktion auf Schreibzugriffe auch einen ersten Test der eigentlichen Funktion der Schaltung, nämlich des Verdrehens der Bits im Eingabewort, vornehmen. Dazu schreiben wir einen neuen Wert an die eben gelesene Adresse: 32'h00000001. `XMD% mwr 0x10000000 0x00000001 <Enter>` Wenn die Schaltung wie geplant funktioniert, sollte der nächste Lesezugriff auf die RC den neuen Wert des result-Registers liefern. Neugierig tippen wir unsere Leseadresse ein: `XMD% mrd 0x10000000 <Enter>`. Das angezeigte Ergebnis bestätigt unsere Erwartungen. Der angezeigte Wert 32'h80000000 ist in der Tat die verdrehte Form des Eingabewertes 32'h00000001. Wir beenden den Test und unsere erste XMD Sitzung durch das abschließende `XMD% exit <Enter>` Kommando.

Obwohl die so durchgeführten Tests noch nicht ausreichen, die absolute Funktionsfähigkeit unseres Entwurfes zu garantieren, können wir doch mit einem recht guten Gefühl in die nächste Phase gehen: Zumindest rudimentäre Funktionen werden scheinbar bereits korrekt ausgeführt.

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 RC 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 <acev/acevapi.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 *inwords, *outwords;
  unsigned long m, inword, outword;

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

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

  // 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
  acev_init ();
  // Zeiger auf RC-Adressraum holen
  rc = acev_get_s0 (NULL);

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

  // Bearbeite Daten
  for (m=0; m < NUM_WORDS; ++m) {
    // Übertrage das Eingabedatenwort an die RC
    rc [0] = inwords[m];
```

```
    // Hole das Ergebnis von der RC und trage es in das Ausgabefeld ein
    outwords[m] = rc [0];
  }

  // merke Stopzeit
  stop = RTEMSIO_getTicks();

  // 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: %lld us\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 RC und dem Ersetzen der eigentlichen Berechnung durch Kommunikation mit der RC.

Betrachten wir die Änderungen im Einzelnen. In Zeile 24 wird eine neue Variable rc als Zeiger auf ein 32b Wort definiert. Über diese Variable, genauer gesagt: dem Ziel dieses Zeigers, wird später die Kommunikation mit der RC 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 RC 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 RC. Zeile 40 richtet die Kommunikation zwischen CPU und RC ein.

Wie schon beim ersten Hardware-Test mit XMD wird auch hier 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 rc zu. Diese wird als Basis für die Adressierung von RC-Registern dienen. Da rc als Feld (Array) von 32b Worten angesehen werden kann, werden durch einfach Indizierung von rc aus die korrekten Register-Adressen in 4B Schritten erzeugt.

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

Die Kommunikation der Software mit der RC zur Übergabe von Ein- und Ausgabedaten ist im Schleifenblock in Zeile 45–51 implementiert: Hier wird, wie schon in der reinen Software-Lösung, der Eingabe-Speicherbereich elementweise durchgegangen. Statt der eigentlichen Berechnung (der inneren Schleife in Listing 3.1) ist aber die RC eingebunden: Jedes Eingabedatenwort wird in Register 0 der RC geschrieben (Zeile 48). Dies löst einen Schreibzugriff (LWRITE=1) auf die RC aus, wobei das Eingabedatenwort auf dem DATAIN-Bus der RC 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 Anwendung 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 rc (Zeile 24). Ohne dieses Attribut würde der Compiler die Anweisungsfolge

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

direkt in

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

umformen. Dabei würde die RC 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 **825365μs**, also rund 0.8s (statt 1.5s bei der reinen Software-Lösung). Die mit 100 MHz getaktete RC 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 nennenswert Zeit vergeht: So war die kürzeste gemessene Zeit zwischen zwei solchen Zugriffen 3 RC-Takte lang, die längste Pause dauerte gar über 100 RC-Takte. In diesen Intervallen liegt die Hardware brach, sie führt keine sinnvollen Berechnungen aus. Der Grund dafür ist einerseits in Schwächen des PLB-Busses 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 RC im Master-Mode

Wie kann man nun die Effizienz der Hardware steigern? Der PLB-Bus als einziger Kommunikationskanal zwischen CPU und RC 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 RC austauscht (ein Datenwort lesen, ein Datenwort schreiben) summieren sich sehr schnell die

Latenzen, die beim PLB-Protokoll überhaupt für den Aufbau einer Verbindung gebraucht werden: Auf dem ML310 wurden für das Schreiben eines 32b-Wortes auf die RC 3 RC-Takte gemessen, beim Lesen eines Wortes sogar 5 RC-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 RC wird nun *eigenständig* alle Speicherzugriffe auf den Hauptspeicher durchführen (Master-Mode). Dabei kann sie bequem die entsprechenden Burst-Transfers generieren und auch Zugiffsmuster 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. Ein weiterer Effekt ist, daß die Hardware im Master-Mode nicht dem Task-Wechsel-Verhalten der CPU unterworfen ist, sozusagen als eigener Ausführungsknoten agiert.

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 RC-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.

An der FG ESA wurde zu diesem Zweck die Memory Architecture for Reconfigurable Computers (MARC) entwickelt. Es handelt sich dabei um ein flexibles (in Bezug auf die RC-Anwendung) und portables (in Bezug auf die ACS-Hardware) Speicherzugriffssystem. Die RC-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.

Abbildung 3.12 zeigt dabei die Schnittstelle eines MARC-Streams zur benutzerdefinierten Hardware in der RC. 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 RC-Operation angezeigt und so eventuell noch auf der RC lokal gepufferte Daten eines

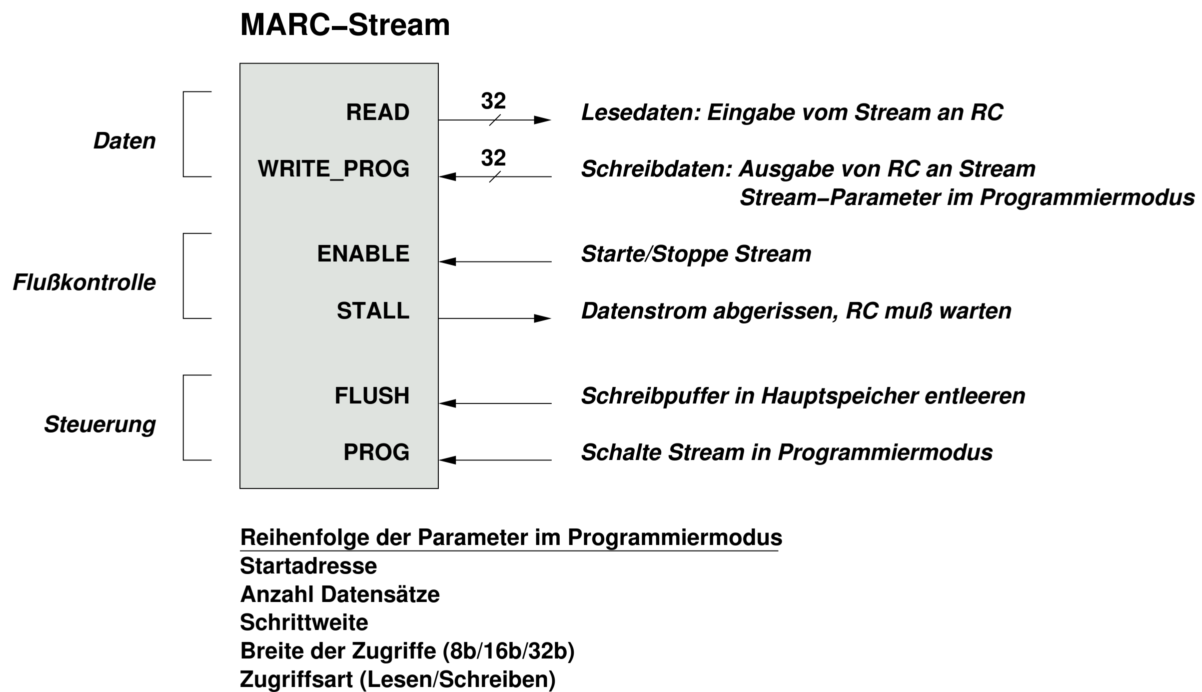


Abbildung 3.12: RC-seitige Schnittstelle eines MARC-Streams

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 3.12 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.

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 3.13).

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.

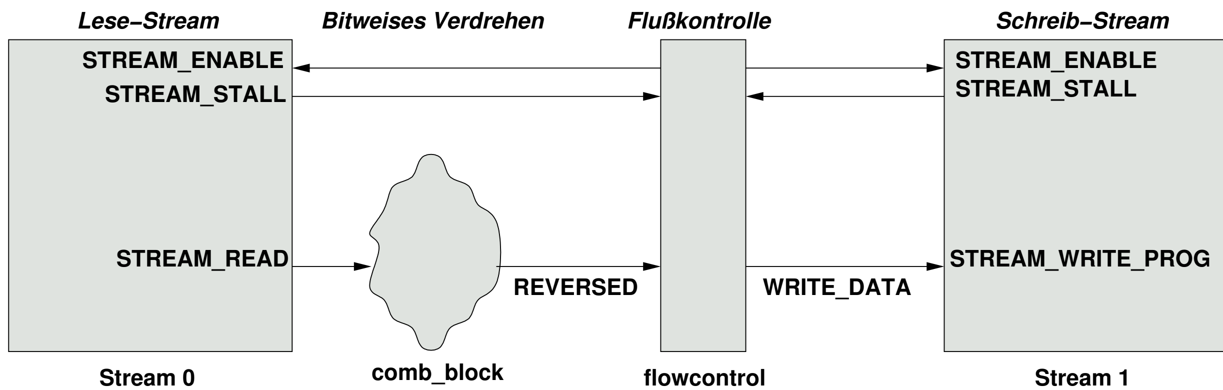


Abbildung 3.13: Architektur der Master-Mode Hardware

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"

module user (
            // *** Globale Signale
            CLK,                // Takt
            RESET,              // Systemweiter Reset

            // *** Slave-Schnittstelle
            ADDRESSED,          // RC angesprochen im Slave-Mode?
            WRITE,              // Schreibzugriff ?
            DATAIN,             // Dateneingang
            DATAOUT,            // Datenausgang
            ADDRESS,            // Adresseingang
            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  // 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_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;
    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 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;
    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 RC

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 RC 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 RC ADDRESS-Busses ein. Diese werden als Registernummern von $0 \ldots 15$ 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 3.13 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 RC-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 (siehe Abschnitt 3.3.2) 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 Programmiermodus 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 eingentliche 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 gestoppt (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 RC ausgelösten Interrupt wieder abzuschalten. Anderenfalls würde die Interrupt Handler-Funktion blockiert, denn sie reagiert nur auf steigende Flanken. Bei unserer Beispielanwendung wird dazu ein Mechanismus verwendet, der bei jedem Lesezugriff auf ein beliebiges RC-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 RC 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 Plazierung und Verdrahtung immer noch mit einer maximalen Taktfrequenz von mehr als den geforderten 100 MHz. Auch diese Verlangsamung ist überwiegend auf die MARC innewohnende Komplexität zurückzuführen (Grund: MARC nimmt für eine verkürzte Latenz einen langsameren Takt in Kauf).

Entwurfstest durch Systemsimulation

An dieser Stelle könnten wir versuchen, den Hardware-Teil unabhängig von der noch nicht entwickelten Software einem ersten Test in XMD zu unterziehen. Dies ist zwar möglich, aber etwas trickreich, da die Master-Mode Zugriffe der RC von der RAM-Schnittstelle noch einmal in einen anderen Adreßraum verschoben werden, was die Adreßrechnung unübersichtlich macht.

Sicherer ist es hier, das HDL-Modell der Anwendung zu simulieren. 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 Hauptspeicherzugriff 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 (Abbildung 3.9) zu und benutzt das RC-lokale SRAM als Datenpuffer. Beide 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 erforderlichem 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 RC 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

```
module stimulus (
                    LRESETOL, // Systemweiter Reset
                    LCLKA,    // Systemweiter Takt
                    LA,       // Lokaler Adressbuss
                    LADSL,    // Beginnt neuer i960 Adresszyklus ?
                    LD,       // Lokaler Datenbus
                    LWRITE,   // Lese−/ Schreibzugriff auf RC
                    LBEL,     // Byte−Enables 0...3 in LD
                    LBLASTL,  // Zeige Ende des i960 Zugriffes an
                    LREADYIL, // RC hat geantwortet , ist LD gültig ?
                    LINTIL    // RC−Interrupt ausgelöst ?
                 );

‘ include ” plbutils .v”

  // Hier beginnt der eigentliche Test
  initial begin : test

    // Hilfsvariable als Ziel für Leseoperationen
    reg [31:0] data;

    // Initialisiere Simulationsumgebung
    Startup ;

    // Zeichne alle Signale im ganzen System auf
    StartRecordingSignals ;

    // Führe einen systemweiten Reset durch
    SystemReset;

    // Lese ein 32b Wort von der Adresse 40
    // Da dort kein Register liegt , sollte
    // die Magic Number 0x00COFFEE zurückkommen
    Read32(‘SLAVE_BASE + 24’h28, data);
    $display (”Magic %h\n”, data);

    // Startadresse 4 in Register 0 schreiben
    Write32(‘SLAVE_BASE + 24’h0, ‘MASTER_BASE + 32’h00000004);

    // Zieladresse 4096 in Register 1 schreiben
    Write32(‘SLAVE_BASE + 24’h4, ‘MASTER_BASE + 32’h00001000);

    // Datensatzlänge 48 in Register 2 schreiben
```

```
    Write32(`SLAVE_BASE + 24'h8, 32'h00000030);

    // RC starten durch Schreiben auf Register 3
    Write32(`SLAVE_BASE + 24'hc, 32'h00000001);

    // Warte, bis RC durch IRQ das Ende anzeigt
    RunUntilInterrupt ;

    // Fahre Simulationsumgebung wieder herunter
    Shutdown;
  end
endmodule
```

Die Schnittstelle unseres Stimulus-Moduls ist durch die Simulationsumgebung vorgegeben und darf nicht verändert werden (Zeile 1-12). Indem wir in Zeile 14 einfach das PLB Makropaket laden, brauchen wir uns um diese Details anschließend nicht mehr zu kümmern, Nach dem Initialisieren des Pakets in Zeile 23 weisen wir den Simulator in Zeile 26 an, alle Signale unseres Entwurfes aufzuzeichnen, damit wir sie später in aller Ruhe studieren können (z.B. als Waves oder Speicherauszüge). In Zeile 29 wird die simulierte System-Hardware korrekt zurückgesetzt. Nun folgen die eigentlichen Testanweisungen, die als eine Folge von Lese- und Schreibzugriffen auf die RC 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. Die Makros SLAVE_BASE und MASTER_BASE definieren systemabhängige Adreßversätze (S0 respektive Master-Speicher). In Zeile 35 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 38 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 41 durch einen Schreib-Zugriff in Register 1 der RC (=DESTADDR) eingetragen. Analog wird in Register 2 (=COUNT) die Anzahl der Datensätze für diesen Test eingetragen (hier 48). Durch den Schreibzugriff in Zeile 47 auf Register 3 wird schließlich die RC gestartet. Wir lassen die Simulation nun solange laufen, bis die RC einen Interrupt an die CPU auslöst (Zeile 50). 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/acevapi.h>

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

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

// Nummern der verschiedenen RC-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, RTEMSIO_getTicks();

  // Zeiger auf S0-Bereich mit RC-Registern
  unsigned long volatile *rc;

  // RC initialisieren
  acev_init();

  // Zeiger auf S0-Bereich mit RC-Registern holen
  rc = acev_get_s0(NULL);

  // fordere Speicher für Ein- und Ausgabefelder an
  // *_phys zeigt auf die physikalische Speicheradresse
  // aus Sicht der Hardware
  inwords = acev_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);
  }

  // Funktioniert der Slave Zugriff ?
  printf ("Magic: %.8lx\n", rc [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 = RTEMSIO_getTicks();

  // "Ubertrage Parameter an RC (nicht die Daten selbst )
  rc [REG_SOURCE_ADDR] = inwords_phys; // Physikalische(!) Startaddresse
  rc [REG_DEST_ADDR] = outwords_phys; // Physikalische (!) Zieladdresse
  rc [REG_COUNT]     = NUM_WORDS; // Anzahl Datensätze
  rc [REG_START]     = 1;         // Startkommando für RC

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

  // merke Stopzeit
  stop = RTEMSIO_getTicks();

  // Hardware Interrupt zurücksetzen durch beliebigen HW Lesezugriff
  rc [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
  acev_free_master ((void *) inwords);

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

Die Zeilen 12–15 definieren symbolische Namen für die Hardware-Register.

Das Hauptprogramm ist weitestgehend unverändert geblieben. Die Schleife, die in der Slave-Mode Lösung die Eingabedaten wortweise zur Umrechnung an die RC übergeben hat, ist nun vollständig

Lösung	RC-Taktfrequenz in MHz	RC-Größe in Slices	Rechenzeit in μs	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

ersetzt worden. Stattdessen werden in den Zeilen 65–68 die Parameter des aktuellen Programmlaufes in die RC-Register geschrieben. Der Schreibzugriff in Zeile 68 startet schließlich die RC-Ausführung. Von nun an laufen CPU und RC parallel! Da wir für unser Beispiel aber keine weiteren Aufgaben auszuführen haben, lassen wir die Software-Ausführung einfach ruhen und warten, bis die RC mit ihren Berechnungen fertig ist. Dieser Effekt wird durch den Aufruf von acev_wait() in Zeile 71 erreicht. Nach dem Auslösen eines CPU-Interrupts durch die RC wird die Software-Ausführung in Zeile 74 wieder aufgenommen. In Zeile 77 wird durch einen Lesezugriff auf die RC 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 RC-Taktfrequenz von 100 MHz wird ein Datensatz von 256Kw in **5813 μs** 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>
#include <stdlib.h>
#include <acev/acevapi.h>
#include<signal.h>

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

main()
{
        // Marker f"ur Zeitmessung
        unsigned long long start, stop, RTEMSIO_getTicks();
        // Zeiger auf RC-Adressraum
        volatile unsigned long *rc;

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

        // TODO Hier Eingabedaten lesen
        // **** ...

        // RC initialisieren
        acev_init();
        // Zeiger auf RC-Adressraum holen
        rc = acev_get_s0(NULL);

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

        // TODO Hier Daten bearbeiten
```

```
        // **** ...

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

        // merke Stopzeit
        stop = RTEMSIO_getTicks();

        // TODO Hier Ausgabedaten schreiben
        // **** ...

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

4.1.2 user.v

Listing 4.2: Slave-Mode Hardware

```
//
//   user.v
//
//   Generische Slave-Mode Anwendung: Realisiert ein les-/schreibbares
//   Hardware-Register
//

module user(
  CLK,
  RESET,
  ADDRESSED,
  WRITE,
  DATAIN,
  DATAOUT,
  ADDRESS,
  IRQ
);

  // Eing"ange
  input          CLK;
  input          RESET;
  input          ADDRESSED;
  input          WRITE;
  input  [31:0]  DATAIN;
  input  [23:2]  ADDRESS;

  // Ausg"ange
```

```
  output [31:0]    DATAOUT;
  output           IRQ;

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

  // Beginn der Anwendung **************************************************

  reg [31:0] outreg;            // Ergebnisregister

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

  // Steuerung
  always @(posedge CLK or posedge RESET) begin
    // Initialisiere Register
    if (RESET) begin
      outreg <= 32'hDEADBEEF;
      // TODO Weitere Register initialisieren
      // **** ...

    // Schreibzugriff auf RC
    end else if ( ADDRESSED & WRITE) begin
      // TODO Hier weitere Register dekodieren
      // **** ...
      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/acevapi.h>

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

// Anzahl Datens"atze in Ein- und Ausgabedatei
#define NUM_WORDS 256*1024

// Definitionen f"ur 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 f"ur alle Variablen
        unsigned long volatile *inwords, *outwords,
                               *inwords_phys, *outwords_phys;
        // Marker f"ur Zeitmessung
        unsigned long long start, stop, RTEMSIO_getTicks();

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

        // RC initialisieren
        acev_init();
        // Zeiger auf Adressraum mit RC-Registern holen
        rc = acev_get_s0(NULL);

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

        // Funktioniert der Slave Zugriff?
        printf("Magic: %.8lx\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 = RTEMSIO_getTicks();

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

    // TODO Weitere Parameter "ubertragen
    // **** ...

    rc[REG_START] = 1;                 // startkommando

    // Warte auf Ende der Berechnung (wird "uber IRQ angezeigt)
    acev_wait ();

    // merke Stopzeit
    stop = RTEMSIO_getTicks();

    // 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
    acev_free_master ((void *) inwords);

    // Gebe Ergebnis der Zeitmessung in Mikrosekunden aus
    printf ("NUM_WORDS=%d\n", NUM_WORDS);
    printf ("Zeit  : %8lld us\n", (stop - start)/TICKS_PER_USEC);
}
```

4.2.2 user.v

Listing 4.4: Master-Mode Hardware

```
//
// Beispielschaltung f"ur Master Mode-Zugriffe mittels MARC
//
// Diese Schaltung kopiert durch die RC einen Quellspeicherbereich
// in einen Zielspeicherbereich (COUNT 32b Worte)

`include "acsdefs.v"

module user (
            // *** Globale Signale
            CLK,                // Takt
            RESET,              // Chip Reset

            // *** Slave-Interface
            ADDRESSED,          // RC angesprochen im Slave-Mode
            WRITE,              // Schreibzugriff ?
            DATAIN,             // Dateneingang
            DATAOUT,            // Datenausgang
            ADDRESS,            // Adresseingang
            IRQ,                // L"ost Interrupt (IRQ) an CPU aus

            // *** Stream-Interface
            STREAM_READ,        // Read-Datenbus
            STREAM_WRITE_PROG,  // Write-program-Datenbus
            STREAM_STALL,       // Stallsignale
            STREAM_ENABLE,      // Enables
            STREAM_FLUSH,       // Flushsignale
            STREAM_PROG         // Programsignale
           );

  // Schnittstelle ****************************************************

  // Eing"ange
  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;

  // Ausg"ange
  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;

// Deklaration f"ur Stream-Interface
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 ***************************************************

// FSM Zust"ande
`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 // f"uhre 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;
// L"ange der Daten (als 32b Worte)
reg [31:0] COUNT;
// L"ost Interrupt aus
reg        IRQSTATE;
// Aktueller Zustand der Anwendung
reg [4:0]  STATE;
// Sind Streams gestartet ?
reg        STREAMSTART;
// Programmierdaten-Register f"ur Streams
reg [31:0] STREAM_PROGDATA_0;
reg [31:0] STREAM_PROGDATA_1;

// Daten
wire [31:0] WRITE_DATA;
```

```
// Abk"urzung f"ur 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);

// Flu" skontrolle zwischen Ein- und Ausgabedatenstrom
flowcontrol FC (
                CLK,                 // Takt
                RUNNING,             // Streams laufen lassen?
                STREAM_STALL[0],     // H"angt Stream 0 (Eingabe-Strom)?
                STREAM_STALL[1],     // H"angt 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 f"ur Ausgabe-Strom
               );

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

// Gebe immer das gerade adressierte Register aus.
// Nicht ben"otigte Register geben eine Magic-Number
// und den aktuellen IRQ-Status im MSB zur"uck
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 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_WIDTH;
        end
      `STATE_PROG_WIDTH:
        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"oht den Grauwert jedes Bildpunktes um 100.
//
// "Ubersetzung mit: make brighten
// Aufruf mit: ./brighten <lena256.pgm >lena256b.pgm
//

#include "stdio.h"

main()
{
  // 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;

```

```
  // Reiche PGM Kopfdatensatz direkt durch
  fgets ((void*)image, 80, stdin );
  fputs ((void*)image, stdout );
  fgets ((void*)image, 80, stdin );
  fputs ((void*)image, stdout );
  fgets ((void*)image, 80, stdin );
  fputs ((void*)image, stdout );

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

  // *************** 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"sere 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, stdout );
}
```

Abbildung 4.1: Testbild `lena256.pgm` vor Bearbeitung

Abbildung 4.2: Testbild nach Bearbeitung durch `brighten`