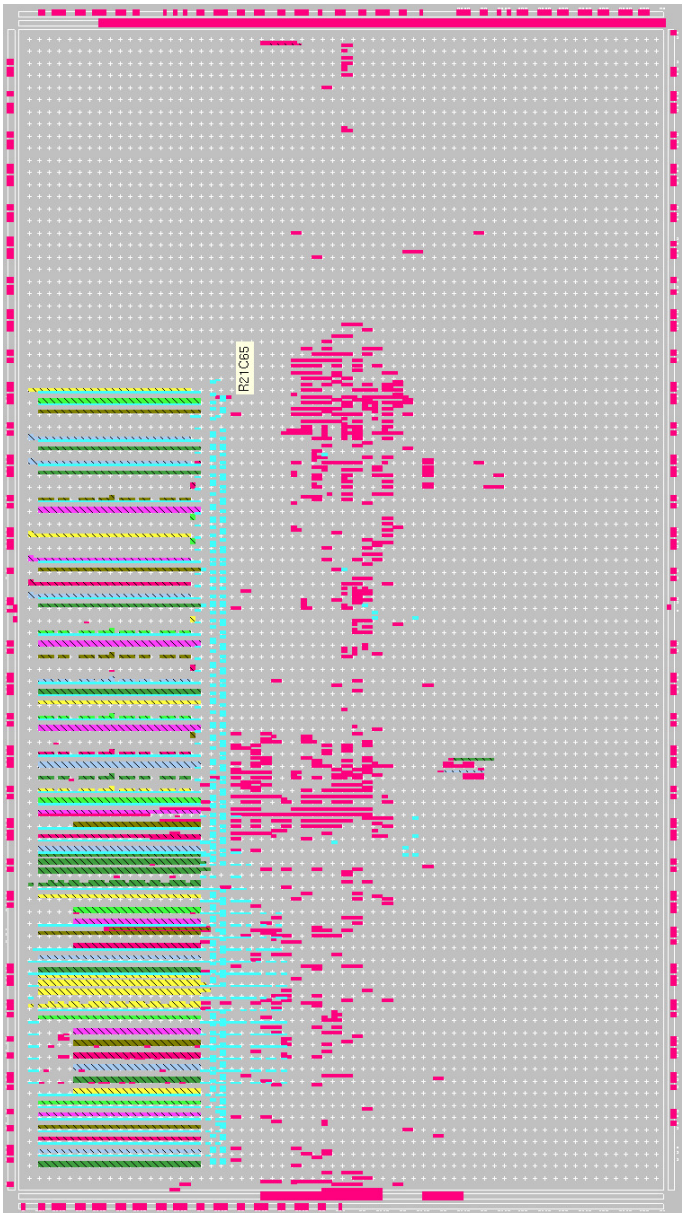
Adaptive Rechensysteme – Eine praktische Einführung

Embedded Systems & Applications

Prof. Dr.-Ing. Andreas Koch
FG Eingebettete Systeme und ihre Anwendungen
TU Darmstadt



WS 2010/2011

Inhaltsverzeichnis

Listings

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.

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.

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.

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

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.

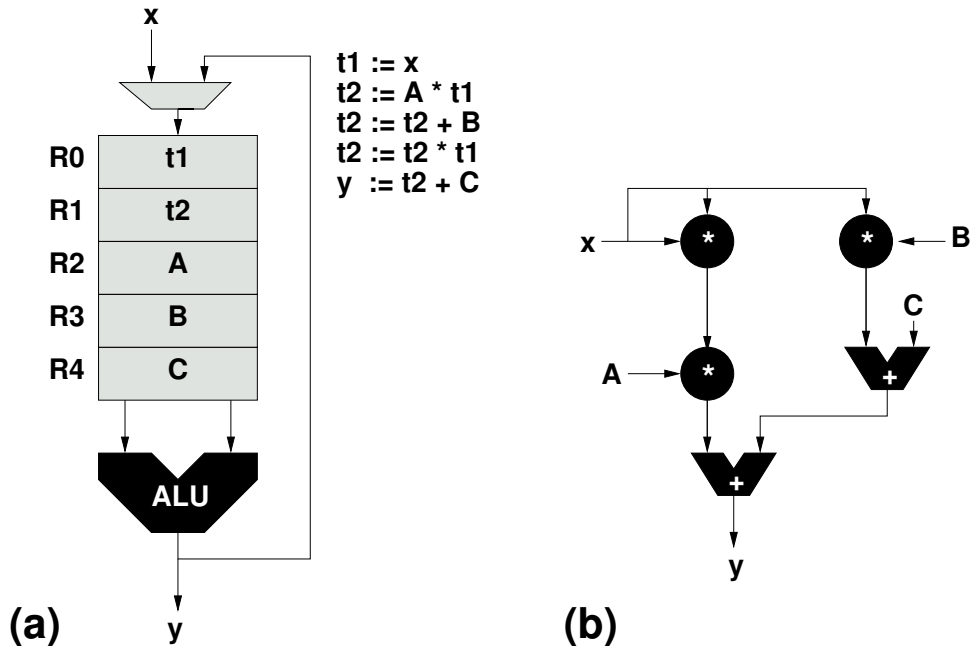


Abbildung 1: Zeitliche und räumliche Verteilung von Berechnungen

Nehmen wir an, das Polynom $y = Ax^2 + Bx + C$ soll für verschiedene Werte von x berechnet werden. Bild 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 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.

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 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.

1.4 Abstufungen von Rekonfigurierbarkeit

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

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

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

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

1.4.2 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.

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 be-

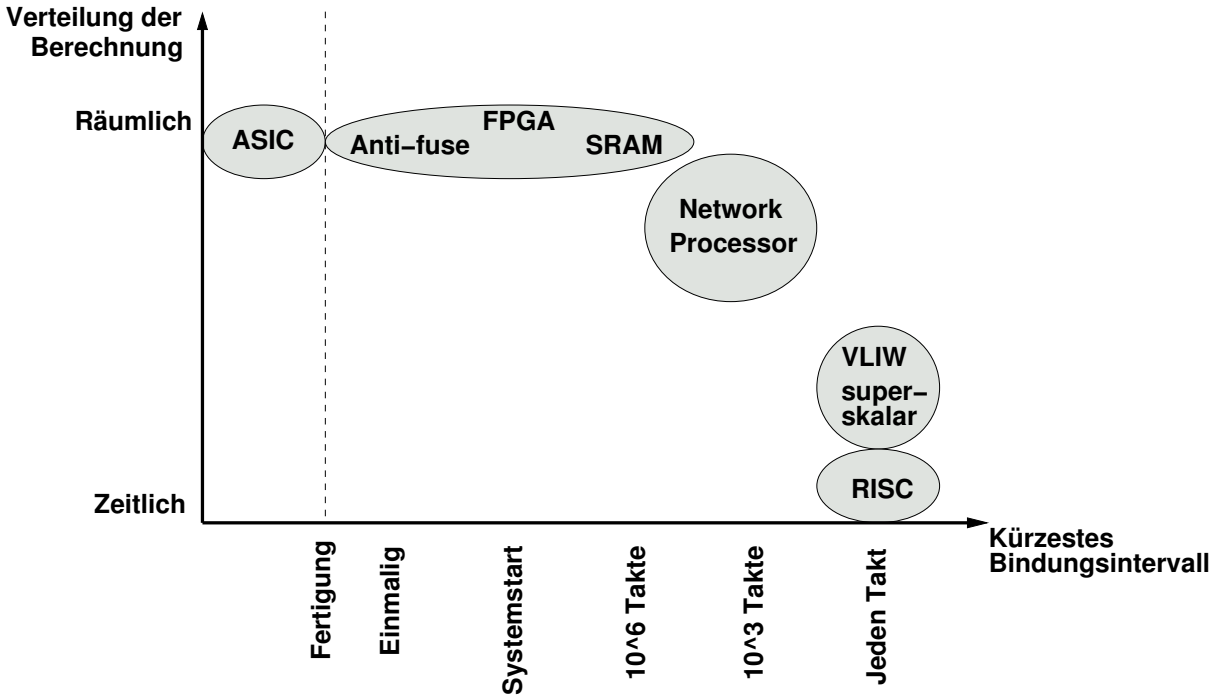


Abbildung 2: Berechnungsverteilung und Bindungsintervalle

reitzustellen.

$10^3 - 10^2$ **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 ...

1.4.3 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.

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 Recheneinheit (im folgenden *RCU* genannt) verwendet wird. Unser adaptives Rechensystem wird also einen Standardprozessor (ab jetzt mit *CPU* bezeichnet) mit rekonfigurierbarer Hardware kombinieren. Wie die nächsten Abschnitte zeigen werden, kann dies aber auf verschiedene Arten geschehen. Die Bilder 3 bis 7 stellt einige der möglichen Varianten dar.

1.5.1 Freistehende RCU

System mit sehr großem RCU Anteil haben diesen außerhalb des eigentlichen Rechners untergebracht (Bild 3). Die Kommunikation erfolgt über eine *externe I/O-Schnittstelle*. Dieser Aufbau wird auch als freistehende RCU (‘stand-alone unit’) bezeichnet.

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

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

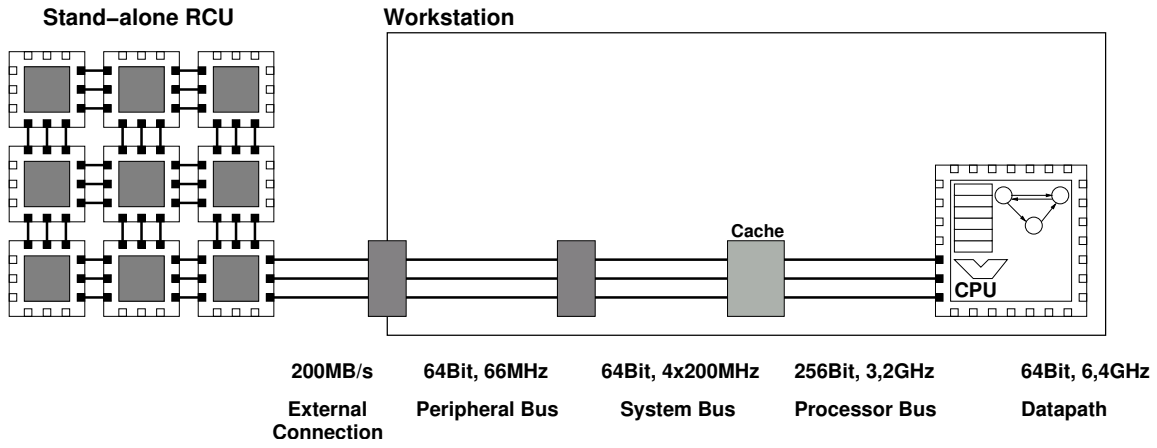


Abbildung 3: Freistehende RCU

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

1.5.2 Angeschlossene RCU

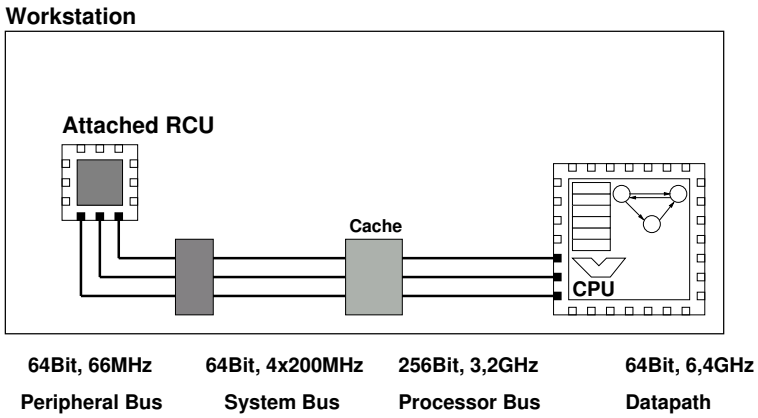


Abbildung 4: Angeschlossene RCU

Es ist daher sehr sinnvoll, die Kommunikationslatenz zwischen RCU und CPU so gering wie möglich zu halten. Auf diese Weise könnten auch kleinere (kürzere) Programmteile noch effektiv ausgelagert werden. Die heute gängige Lösung (Bild 4) verlagert die RCU direkt in den Rechner und schließt sie dort an den *Peripheriebus* (oft PCI, vereinzelt schon PCI-X oder PCI Express, aber noch sind auch ältere Bussysteme wie SBus oder VME in Gebrauch) an. So sind bei dem derzeit gängigen 32b PCI Bus getaktet mit 33 MHz theoretische Datenübertragungsraten von 132

MB/s erreichbar. Aber auch hier sind die Latenzen noch nennenswert: Ein Schreibzugriff vom PCI Bus auf den Hauptspeicher dauert circa 10 Bustakte (330ns), ein Lesezugriff gar über 30 Bustakte ($> 1\mu$s). Der Vorteil dieser Anbindung ist der problemlose Anschluß an die leicht handhabbaren Peripheriebusse von Standardrechnern mittels einer einfachen Steckkarte. Bei dieser Lösung wird von einer angeschlossenen RCU ('attached processing unit') gesprochen.

1.5.3 RCU in Multiprozessorsystem

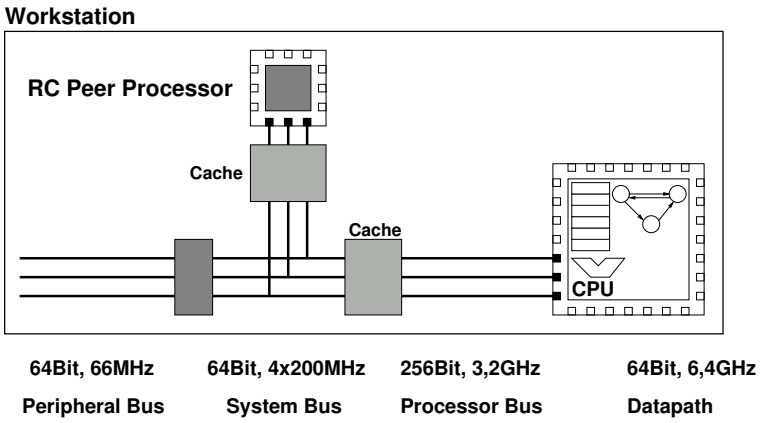


Abbildung 5: Multiprozessoren: RCU gleichberechtigt mit der CPU

Der Aufbau von adaptiven Rechnern mit noch engerer Kopplung von RCU und CPU wird nun zunehmend schwieriger. Eine Möglichkeit sieht die RCU und die CPU als *gleichberechtigte Partner* auf dem Prozessorbus an (Bild 5). Eine solche Anordnungen von gleichberechtigten Prozessoren wird als symmetrischer Multiprozessor (SMP) bezeichnet. Der Geschwindigkeitszuwachs gegenüber Peripheriebussen ergibt sich aus dem sehr viel höheren Bustakt (beispielsweise mit 800 MHz) und den kürzeren Latenzen (im Bereich von 4-40 Takten). Die Protokolle für die Interprozessorkommunikation sind zwar nicht trivial (es müssen unter anderem die unabhängigen Caches von RCU und CPU kohärent gehalten werden), lassen sich aber mit der nötigen Geschwindigkeit auch noch in FPGAs realisieren. Mittlerweile gibt es Trends, Prozessorbusse wie HyperTransport mittels gut handhabbarer Steckverbinder (HTX) zu nutzen, um verschiedenste Recheneinheiten (z.B. Kryptobeschleuniger, aber auch RCUs) leicht auf kommerziell erhältlichen PC-Motherboards einzusetzen.

1.5.4 RCU-Koprozessor

Die Anbindung der RCU zusammen mit der CPU an einen *gemeinsamen Cache* verkürzt die Latenzen in der Regel noch weiter (Bild 6). Bei dieser Kombination agiert die RCU als echter Koprozessor für die CPU. Die Kommunikation zwischen RCU und CPU kann dann in 10 Prozessortakten vorgenommen werden, ein Speicherzugriff über den Cache kann im Erfolgsfall (cache hit) in ähnlicher Zeit bewältigt werden. Solche Architekturen lassen sich zwar noch nicht mit den in PCs üblichen CPUs realisieren (diese sehen schlicht keine RCU auf dem Chip vor). Aber spezielle konfigurierbare Prozessoren wie die Tensilica Xtensa IP-Blöcke werden auch heute schon

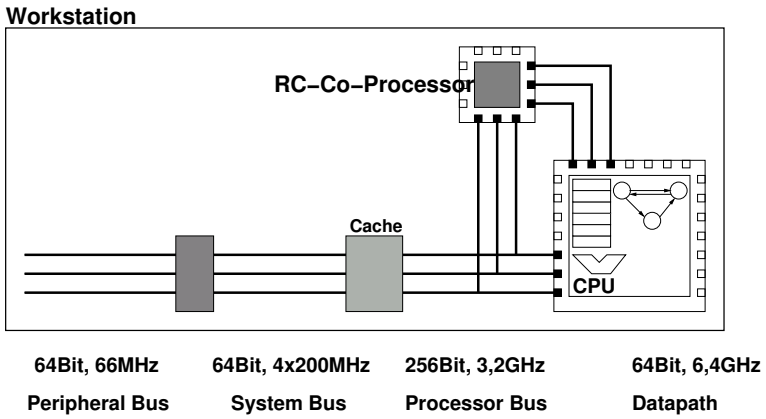


Abbildung 6: RCU als Koprozessor

mit RCUs auf einem Chip kombiniert (z.B. in der Stretch S5000 Architektur). Verschiedene sogenannte System-FPGAs (z.B. Xilinx Virtex II Pro, 4FX und 5FX) enthalten bereits einen oder mehrere Prozessoren auf dem Chip und erlauben so den Koprozessorbetrieb von RCU und CPUs. Hier dreht sich dann der Spiess um, indem die CPU nun in die RCU eingebettet wird.

1.5.5 RCU-Funktionseinheit in CPU

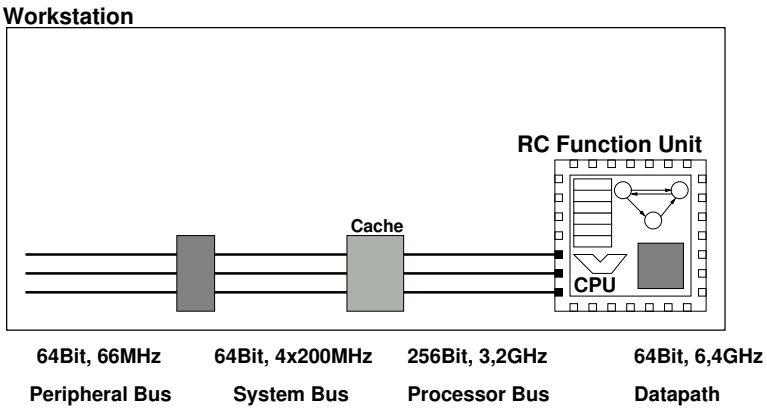


Abbildung 7: RCU als Funktionseinheit in CPU

Auch eine noch engere Integration ist denkbar: Die RCU könnte als *Funktionseinheit* (RFU) direkt in den Prozessorkern hineinintegriert werden (Bild 7). Ähnlich wie eine ALU mit festen Funktionen würde dann auch eine rekonfigurierbare Funktionseinheit bereitstehen. Im akademischen Bereich sind solche Chips bereits realisiert worden. Bei den ersten Ansätzen zeigte sich aber, dass die Anbindung zwar mit niedriger Latenz (1 Prozessortakt), aber auch mit nur niedriger Übertragungsgeschwindigkeit benutzbar war: Ähnlich wie bei den anderen CPU-Befehlen konnten auch an die RCU nur einzelne Register (in der Regel also 32b Worte) übergeben werden. Auch das Ergebnis der Berechnung wurde in einem einzelnen Zielregister abgelegt. Obwohl also in der RCU deutlich aufwendigere Berechnungen realisierbar waren, wurden diese durch die sehr niedrige

Kommunikationsbandbreite ‘ausgehungert’ (bekamen nicht genug Daten). Neuere Experimente mit RCU-Funktionseinheiten versuchen, dieses Problem durch Bereitstellen RCU-eigener Speicherschnittstellen zu umgehen. Dabei müssen aber die Interaktionen zwischen den Speicherzugriffen des Prozessors und der RCU-Funktionseinheit sorgfältig koordiniert werden. Kommerziell ist allerdings noch kein Vertreter dieser Gattung von adaptivem Rechner verfügbar. In einer anderen Spielart (z.B. den Stretch S5000 Chips) werden die RFUs über spezielle, breitere Register mit Daten versorgt, welche durch geeignete LOAD/STORE-Anweisungen der CPU gehandhabt werden. Zwar können hier in einer Instruktion auch größere Datenmengen als die üblichen 32b ausgetauscht werden (nämlich 128b), die Kommunikationsanweisungen sind aber oft eingeschränkt (erlauben beispielsweise wohl einen Datentransfer aus einem CPU in ein breites RCU-Register, aber nicht in die Gegenichtung). Als weiteres Problem erweist sich in der Praxis, die langsamere Taktfrequenz der rekonfigurierbaren Einheiten mit dem Takt der hochgezüchteten festen ALUs (die teilweise mit über 7 GHz Takt arbeiten) des Prozessors in Einklang zu bringen. RFUs kommen daher in der Regel nur auf ohnehin langsameren Prozessoren für eingebettete Systeme zum Einsatz. Hier fallen die Taktdifferenzen weniger extrem aus (z.B. 300 MHz CPU-Takt zu 100MHz RFU-Takt).

1.5.6 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 2), sondern nur die für den Betrieb erforderlichen.

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

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

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

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

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

Zusammenbau und gemeinsamen Test der einzelnen Komponenten) auf.

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

Ein aktueller Ansatz schlägt daher vor, einen Teil der zur Verfügung stehenden Transistormenge

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

Tabelle 1: Gatterkomplexitäten einiger Anwendungen

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

1.7 Beispiel: ML310 als adaptiver Computer

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

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

1.7.1 Hardware-Architektur

Als CPU des ACS fungiert einer der beiden 300 MHz PowerPC 405 Kerne des Virtex II pro-Chips (der andere Kern ist derzeit noch unbenutzt). Neben einigen vergleichsweise kleinen Hardware-Blöcken, die Schnittstellen nach außen, z.B. zum Ethernet- oder Festplatten-Controller herstellen, wird ein signifikanter Teil des verbliebenen Platzes durch den Speicher-Controller belegt. Dieser stellt für CPU und RCU den Zugang zum externen System-Hauptspeicher her, der in Form eines 256MB großen DDR-DRAM Moduls realisiert ist.

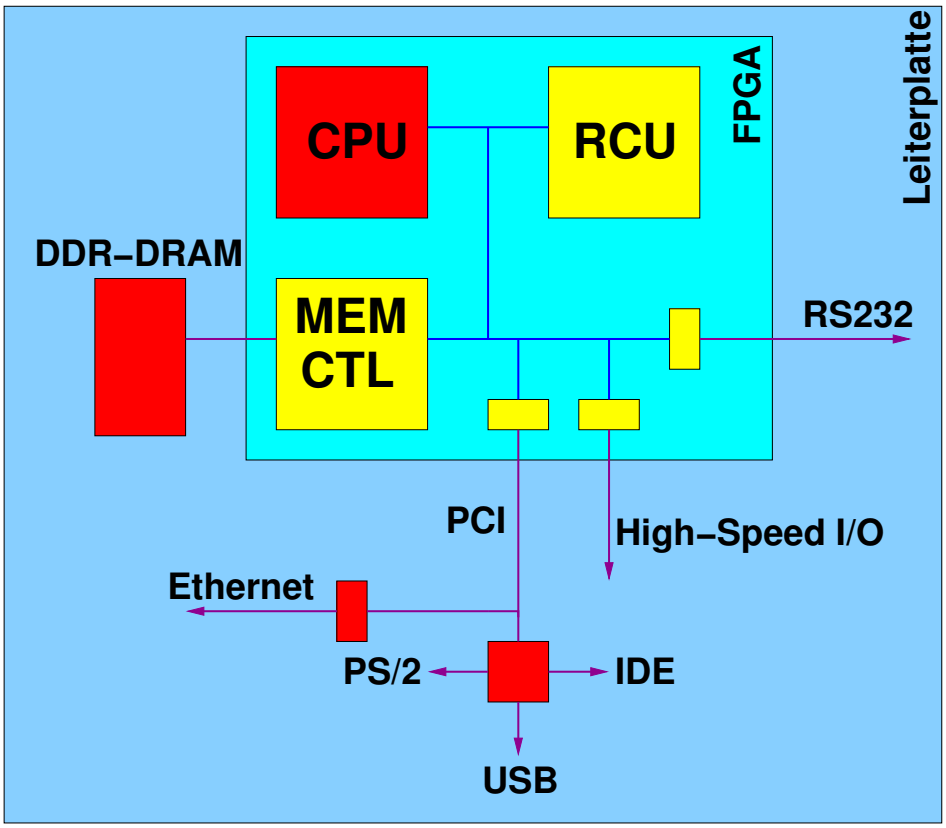


Abbildung 8: ML310 Architektur

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

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

1.7.2 Software-Architektur

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

2 Programmierung adaptiver Rechner

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

Einige Beispiele sind systolische Arrays (wie beim DNA-Sequenz-Vergleich), verschiedene andere Datenflußansätze (z.B. SDF und BDF) und Architekturen wie VLIW/EPIC oder Vektorprozessoren (SIMD). Durch Ausnutzung der 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 RCU-ablaufenden Hardware in einer HDL (in unserem Fall Verilog). Dies war früher die einzige Form, ein ACS zu programmieren. Der Programmierer hat damit zwar uneingeschränkte Freiheit, die für das aktuelle Problem passende Zielarchitektur auf der RCU zu realisieren. Diese Art der Programmierung setzt aber detailierte Kenntnise der ACS-Hardware und des Hardware-Entwurfs im allgemeinen voraus. Neben verschiedenen technischen Problemen (z.B. unzureichendes Fassungsvermögen der RCU) war dies das Haupthindernis bei einer breiteren Benutzung von ACSs: In der Regel verfügen Anwender, die nur mit konventioneller Software-Entwicklung vertraut sind, nicht über die erforderlichen Kenntnisse, ein ACS erfolgreich zu programmieren.

Um die Vorteile des adaptiven Rechnens einer breiteren Benutzerschicht zugänglich zu machen, wird heute versucht, durch geeignete Werkzeuge die Lücke zwischen den Hard- und Software-

Entwurfsebenen zu schließen. Es handelt sich dabei in erster Linie um Ansätze aus dem Hardware-Software-Codesign, bei denen der gesamte Hardware-Zweig der Anwendung vollautomatisch erstellt wird. Der Benutzer formuliert die Programme in einer ihm vertrauten höheren Programmiersprache (beispielsweise C, Java oder MATLAB) und ruft die Werkzeuge in gewohnter Form auf (ähnlich einem normalen Software-Compiler). Im Idealfall wird ihm dadurch ohne weiteres Zutun eine hybride Hardware-Software-Anwendung erzeugt, die direkt auf dem ACS ausführbar ist. Als Beispiel für einen solchen Entwurfsfluß wird im Abschnitt 4 der Compilerprototyp NIMBLE vorgestellt, der aus Standard ANSI C (ohne jede Einschränkung oder Erfordernis von Sonderkonstrukten) automatisch ACS-Anwendungen erzeugt.

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 9).

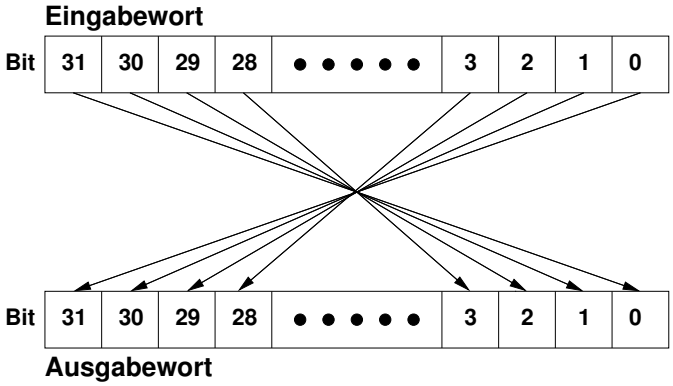


Abbildung 9: Funktion der Beispielanwendung

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 3.1).

2. Eine einfache Hardware-unterstützte Lösung, bei der sowohl Ein- als auch Ausgabedaten einzeln durch die CPU übertragen werden (slave-mode, Abschnitt 3.2)
3. Eine aufwendigere Hardware-unterstützte Lösung, bei der die RCU nach Übergabe von Parametern durch die CPU selbstständig die Daten bearbeitet (master-mode, Abschnitt 3.3).

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.

3.1 Reine Softwarelösung

Der Programmtext der reinen Software-Lösung ist in Listing 1 zu sehen. Um den Code auf das Wesentliche zu beschränken, wurde auf (für diese Erklärung) weniger wichtige Teile wie Fehlerüberprüfungen verzichtet. Reale Anwendungen sollten diese Ausnahmen sehr wohl korrekt behandeln! Die generelle Vorgehensweise zur Lösung der Aufgabe ist sehr einfach: Nach einigen administrativen Anweisungen beginnt der Programmablauf in Zeile 23. Hier werden mittels der Funktion malloc() zwei Speicherbereiche zum Aufnehmen der Ein- und Ausgabedaten angefordert. Deren Größe berechnet sich als die Anzahl der Datenworte NUM_WORDS, hier gesetzt als 256Kw, multipliziert mit der Größe eines Wortes long in Bytes, hier 4B. Der Zeiger inwords zeigt danach auf den Speicherbereich für die Eingangsdaten, der Speicher outwords auf den für die Ausgangsdaten.

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

Listing 1: Reine Softwarelösung

```
#include <stdio.h>                                                    1
#include <stdlib.h>                                                   2

// Anzahl von Ticks der Systemuhr pro Mikrosekunde                    4
#define TICKS_PER_USEC 25                                             5

// Anzahl Datensätze in Ein- und Ausgabedatei                         7
#define NUM_WORDS (1024*256)                                          8

main()                                                                10
{                                                                     11
        // Ein- und Ausgabedateien                                    12
        FILE *infile, *outfile;                                       13

        // Benutze vorzeichenlose Ganzzahlen (32b Worte) für alle Daten 15
        unsigned long n, m, mask, set, inword, outword;               16
        // Zeiger auf Ein- und Ausgabe-Speicherbereiche               17
        unsigned long *inwords, *outwords;                            18
```

```
// 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                    68
    printf ("Zeit: %lld us\n", (stop - start)/TICKS_PER_USEC);               69
}                                                                            70
```

Vor der eigentlichen Datenkonvertierung wird in Zeile 35 die aktuelle Zeit in der Variablen start gemerkt. Man beachte, dass hier als Einheit sogenannte "Ticks" der Systemuhr verwendet werden. Auf dem Ml310 hat ein Tick eine Länge von 40ns (siehe Zeile 5). Um daher die potentiell sehr großen Zahlenwerte verarbeiten zu können, sind die Variablen für die Zeitmessung als 64b Variablen (Datentyp long long) deklariert worden (Zeile 18).

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

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

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

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

3.2 Beschleunigung durch RCU im Slave-Mode

3.2.1 Auswahl geeigneter Programmteile für Beschleunigung

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

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

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

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

3.2.2 Hardware-Schnittstelle der RCU

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

Schauen wir uns zunächst einmal die Hardware an, die bei diesem Vorgehen für unsere Anwendung in der RCU realisiert werden muß (Listing 2).

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

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

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

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

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

ADDRESS Durch diesen Bus teilt die CPU der RCU während eines Lese- oder 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 RCU können die Werte auf diesem Bus von der RCU ignoriert werden.

Listing 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                                                                    49

endmodule                                                                51
```

3.2.3 Architektur der Recheneinheit

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

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

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

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

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

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

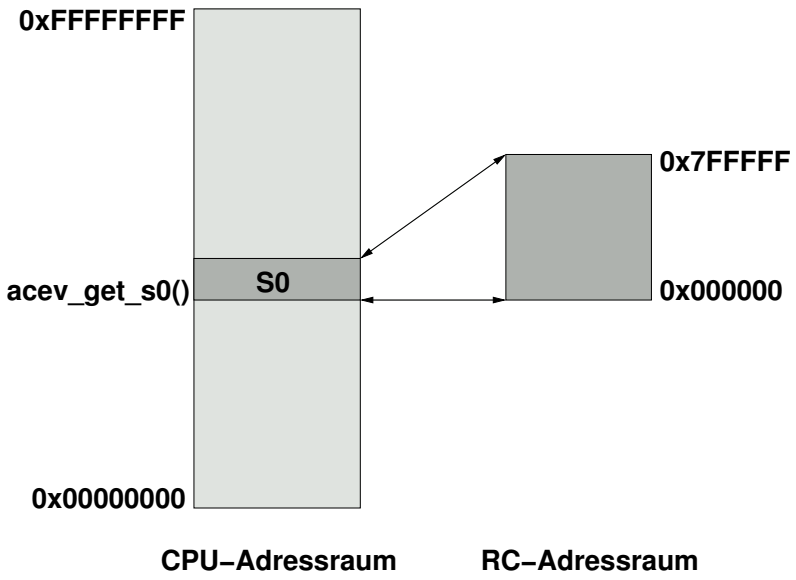


Abbildung 10: Speicheraufteilung im Slave-Mode

3.2.4 Software-Schnittstelle zur RCU

Die Kommunikation zwischen Soft- und Hardware beruht darauf, dass die RCU-internen Register (auf der ML310 auch *S0-Bereich* genannt), die in der Hardware über die Signale ADDRESSED, ADDRESS, DATAIN, DATAOUT und WRITE angesprochen werden, in den Adressraum der CPU eingeblendet werden. Diese Struktur ist in Abbildung 10 dargestellt. Dabei gilt folgendes:

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

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

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

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

3.2.5 Software-Teil der Slave-Mode Anwendung

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

die Ergebnisse abholt. Dieses ist in Listing 3 gezeigt.

Listing 3: Software-Teil der Slave-Mode Anwendung

```
#include <stdio.h>                                                    1
#include <stdlib.h>                                                   2
#include "rcu.h"                                                      3
                                                                      4
                                                                      5
                                                                      6
                                                                      7
// Anzahl Datensätze in Ein- und Ausgabedatei                         8
#define NUM_WORDS (1024*256)                                          9
                                                                      10
main()                                                                11
{                                                                     12
  // Ein- und Ausgabedateien                                          13
  FILE *infile, *outfile;                                             14
                                                                      15
  // Benutze vorzeichenlose Ganzzahlen (32b Worte) für alle Daten     16
  unsigned long *inwords, *outwords;                                  17
  unsigned long m, inword, outword;                                   18
                                                                      19
  // Marker für Zeitmessung (64b Worte)                               20
  unsigned long long start, stop, RTEMSIO_getTicks();                 21
                                                                      22
  // Zeiger auf RC-Adressraum                                         23
  volatile unsigned long *rcu;                                        24
                                                                      25
  // fordere Speicher für Ein- und Ausgabefelder an                   26
  inwords  = malloc(NUM_WORDS * sizeof(unsigned long));               27
  outwords = malloc(NUM_WORDS * sizeof(unsigned long));               28
                                                                      29
  // Öffne die Dateien zum Lesen und Schreiben                        30
  infile  = fopen("test1.in", "r");                                   31
  outfile = fopen("test1.out", "w");                                  32
                                                                      33
  // Lese komplette Eingabedatei in Eingabe-Speicherbereich           34
  fread(inwords, sizeof(unsigned long), NUM_WORDS, infile);           35
                                                                      36
  // RC initialisieren                                                37
  rcu_init();                                                         38
  // Zeiger auf RC-Adressraum holen                                   39
  rcu = rcu_get_s0(NULL);                                             40
                                                                      41
  // Merke Startzeit der Berechnung                                   42
  start = rcu_get_ticks();                                            43
                                                                      44
  // Bearbeite Daten                                                  45
  for (m=0; m < NUM_WORDS; ++m) {                                     46
```

```
    // Übertrage das Eingabedatenwort an die RCU                    47
    rcu[0] = inwords[m];                                            48
    // Hole das Ergebnis von der RCU und trage es in das Ausgabefeld ein  49
    outwords[m] = rcu[0];                                           50
  }                                                                 51

  // merke Stopzeit                                                 53
  stop = rcu_get_ticks ();                                          54

  // Schreibe das komplette Ausgabefeld in die Ausgabedatei         56
  fwrite(outwords, sizeof(unsigned long), NUM_WORDS, outfile);      57

  // Gebe Speicher für Ein-/Ausgabefelder wieder frei               59
  free(inwords);                                                    60
  free(outwords);                                                   61

  // Schließe Dateien                                               63
  fclose( infile );                                                 64
  fclose( outfile );                                                65

  // Gebe Ergebnis der Zeitmessung in Mikrosekunden aus             67
  printf("Zeit: %lld us\n", (stop-start)/TICKS_PER_USEC);           68
}                                                                   69
```

In weiten Teilen ist dieses Programm identisch zur reinen Software-Lösung (Listing 1). Die wesentlichen Unterschiede liegen in der Initialisierung der RCU und dem Ersetzen der eigentlichen Berechnung durch Kommunikation mit der RCU.

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

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

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

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

Bei unserer Beispielanwendung verwenden wir nur ein einzelnes RCU-Register (result), das unabhängig von der aktuellen Adresse im gesamten RCU-Adressraum anliegt. Wir werden es willkürlich

als Register 0 von der S0-Basis rcu aus mit dem Ausdruck rcu[0] addressieren.

Die Kommunikation der Software mit der RCU zur Übergabe von Ein- und Ausgabedaten ist im Schleifenblock in Zeile 45–51 implementiert: Hier wird, wie schon in der reinen Software-Lösung, der Eingabe-Speicherbereich elementweise durchgegangen. Statt der eigentlichen Berechnung (der inneren Schleife in Listing 1) ist aber die RCU eingebunden: Jedes Eingabedatenwort wird in Register 0 der RCU geschrieben (Zeile 48). Die Anweisung löst einen Schreibzugriff (WRITE=1) auf die RCU aus, wobei das Eingabedatenwort auf dem DATAIN-Bus der RCU erscheint. Der kombinatorische Block berechnet sofort das entsprechende bitweise verdrehte Wort (Zeile 33–38 in Listing 2), das in Zeile 47 in Listing 2 als Endergebnis in das Ausgaberegister result übernommen wird. Das Ausgaberegister wird im Software-Teil der Anwendung dann in Zeile 50 (Listing 3) via dem DATAOUT-Bus, an dem es immer anliegt, ausgelesen und in den Ausgabe-Speicherbereich geschrieben.

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

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

direkt in

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

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

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

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

3.3 Weitere Beschleunigung durch RCU im Master-Mode

Wie kann man nun die Effizienz der Hardware steigern? Der Kommunikationskanal zwischen CPU und RCU, der sogenannte Processor Local Bus (PLB) läßt sich offensichtlich nicht umgehen. Er

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

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

3.3.1 Master-Mode Speicherschnittstelle

In der Praxis erweist sich die Hardware-Realisierung von Master-Mode Zugriffen leider als trickreich. Viele Eigenheiten, z.B. versteckte Einschränkungen der Burst-Länge, unglückliche Timing-Abhängigkeiten und ähnliches, treten erst bei praktischen Versuchen mit dem gesamten System auf, scheinbar völlig losgelöst von der heilen Welt der Datenblätter und Spezifikationen. Es bietet sich daher an, den Aufwand für die RCU-Implementierung von funktionierenden Master-Mode Zugriffen nur einmal zu betreiben. Der entsprechende Hardware-Block muß dabei so flexibel ausgelegt sein, dass er daraufhin in den unterschiedlichsten Szenarien ohne größere Anpassungen wiederverwendet werden kann.

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

Abbildung 11 zeigt dabei die Schnittstelle eines MARC-Streams zur benutzerdefinierten Hardware in der RCU. Die sechs Ports sind dabei grob in drei Gruppen einteilbar: Daten-Ports erlauben den Datenfluß aus dem Stream hinaus in die Benutzer-Hardware (via READ) oder aus der Benutzer-Hardware in den Stream hinein (via WRITE_PROG). Zur Kontrolle des Datenflusses stehen zwei weitere Ports bereit. Über ENABLE kann die Benutzer-Hardware den Datenstrom anhalten, während ihr über STALL durch MARC ein Abriss im Datenstrom angezeigt wird. Letztlich werden noch zwei Steuereingänge benötigt. Durch FLUSH wird schreibenden Streams das Ende der RCU-Operation angezeigt und so eventuell noch auf der RCU lokal gepufferte Daten eines

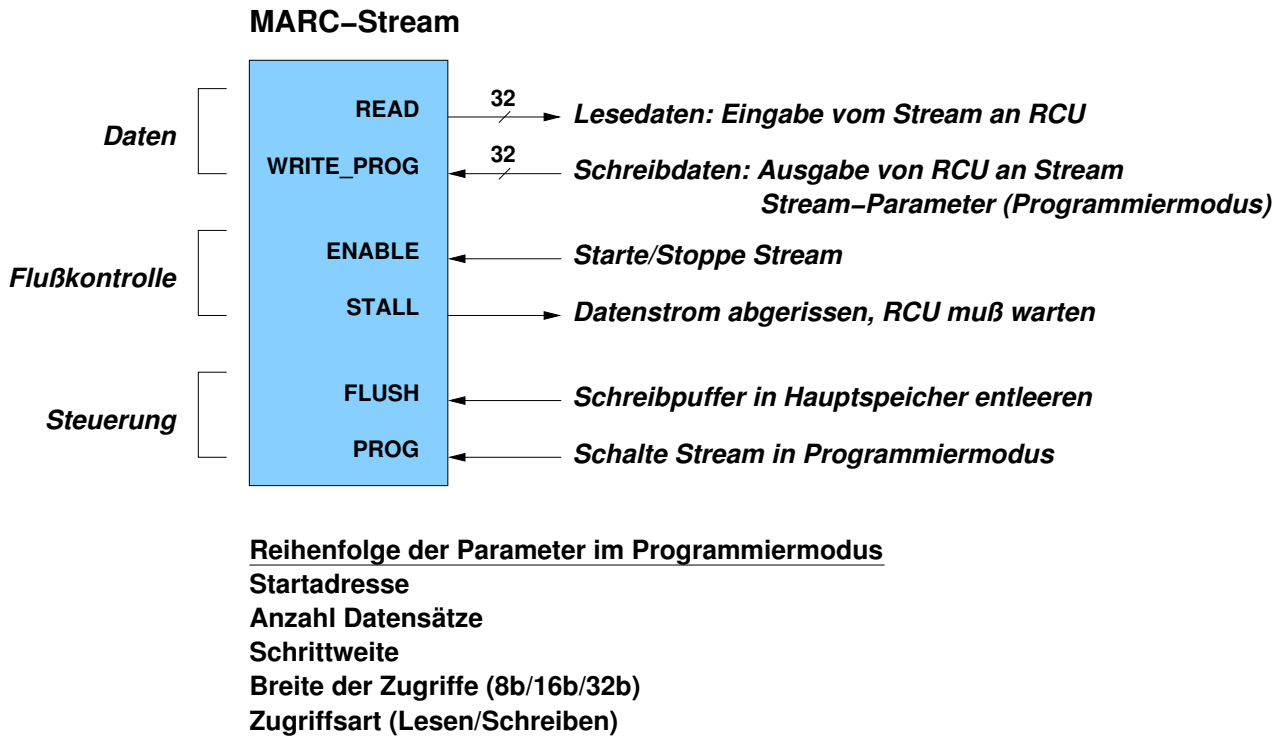


Abbildung 11: RCU-seitige Schnittstelle eines MARC-Streams

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 11 gezeigte Reihenfolge (ein Parameter pro positiver Taktflanke) einzuhalten. Wenn diese Programmiersequenz vorzeitig beendet wird (durch Wegnehmen des PROG Signals), bleiben alle noch nicht geschriebenen Parameter unverändert.

3.3.2 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 12).

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

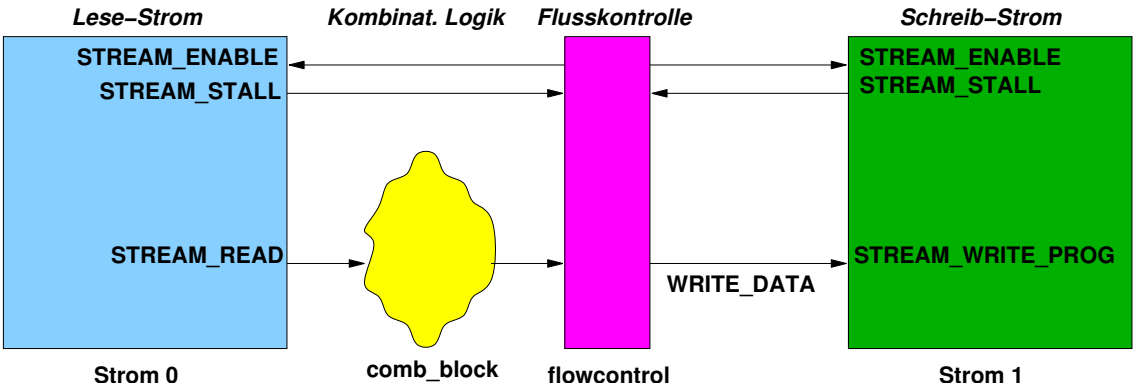


Abbildung 12: Architektur der Master-Mode Hardware

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

Listing 4 zeigt das Verilog-Modell der Master-Mode Lösung. In den folgenden Abschnitten werden die wesentlichen Teile vorgestellt.

Listing 4: Hardware-Teil der Master-Mode Anwendung

```
'include "marcdefs.v"                                                    1

module user (                                                            3
                // *** Globale Signale                                   4
                CLK,                    // Takt                         5
                RESET,                  // Systemweiter Reset           6

                // *** Slave- Schnittstelle                             8
                ADDRESSED,              // RC angesprochen im Slave-Mode? 9
                WRITE,                  // Schreibzugriff ?             10
                DATAIN,                 // Dateneingang                 11
                DATAOUT,                // Datenausgang                 12
                ADDRESS,                // Adresseingang                13
                IRQ,                    // Löst Interrupt (IRQ) an CPU aus 14

                // *** Schnittstelle für MARC-Streams                   16
                STREAM_READ,        // Read-Datenbus                    17
                STREAM_WRITE_PROG, // Write-Programm-Datenbus            18
                STREAM_STALL,       // Stall -Signale                   19
```

```
            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
```

3.3.3 Hardware-Schnittstelle der RCU

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

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

3.3.4 Interner Aufbau der Hardware

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

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

In Zeile 94 beginnt schließlich der aktive Teil der Anwendung. RUNNING ist gesetzt, wenn die Streams gestartet wurden, und noch Daten zu bearbeiten sind. Letzteres stellen wir mit dem Zähler COUNT fest, der die Anzahl der noch zu lesenden Daten zählt. In den Zeilen 97–106 werden die beiden Streams (wie bereits in Abbildung 12 gezeigt) durch ein Flußkontroll-Modul gekoppelt. Dieses (hier nicht weiter gezeigte) Modul erfüllt die in Abschnitt 3.3.2 definierten Anforderungen. Neben den STALL und ENABLE-Signalen der gekoppelten Streams muß es auch noch an die Eingangsdaten (von der kombinatorischen Logik) und due Ausgangsdaten (an den Schreib-Stream) angeschlossen werden.

In Zeile 109 wird das interne Interrupt-Statusregister IRQSTATE an die Interrupt-Leitung der CPU angeschlossen. Der Block in Zeile 114–118 ist eine Erweiterung der bekannten Implementierung von Slave-Mode Lesezugriffen. Als Dienstleistung bieten wir hier der CPU an, die aktuellen Werte der RCU-Parameter auszulesen. Da wir nun mehr als ein Register zur Verfügung stellen, muß hier

die Adresse des Zugriffs tatsächlich ausgewertet werden. Dies geschieht mit dem Umweg über die in Zeile 90 definierte Registernummer REGNUM. Für die Registernummern 0...2 geben wird das entsprechende interne Register auf den DATAOUT-Bus aus. In allen anderen Fällen (derzeit nicht benötigte Registernummern) wird für das Debugging eine weitere gute erkennbare "magic number" definiert, in die zusätzlich noch der aktuelle Zustand des internen Interrupt-Registers im MSB eingeblendet wird. Auch dies dient der Erleichterung beim Hardware-Debugging.

In den Zeilen 122–128 werden die Schreib-/Programmiereingänge der Streams verdrahtet. Im Fall des Lese-Streams (Stream 0) wird der Port lediglich zur Programmierung verwendet und kann somit immer an das Programmierdatenregister STREAM_PROGDATA_0 angeschlossen werden. Beim Schreib-Stream (Stream 1), der einen echten Datenstrom bearbeitet, muß der Port umgeschaltet werden: Wenn der Stream im 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 beendet (Zeile 238) und der Interrupt zur CPU hin ausgelöst (Zeile 240). Als letztes wird der Master-Mode Betrieb abgeschaltet (Zeile 242) und in Zeile 244 die Master-Mode Zustandsmaschine wieder in den Startzustand (für den nächsten Durchgang) zurückgesetzt.

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

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

Dieses Design läuft nach der 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).

3.3.5 Entwurfstest durch Systemsimulation

Um den Hardware-Teil unabhängig von der Software testen zu können, simulieren wir das HDL-Modell der Anwendung. Dabei ist es aber nicht ausreichend, nur das in Listing 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 zu (in Abbildung 8 als MEM CTL bezeichnet) und benutzt RCU-interne SRAM-Blöcke als Datenpuffer. Beide dieser Einheiten werden also ebenfalls mit in die Simulation aufgenommen. Diese Ausweitung des Simulationsrahmens über die gerade entworfene Hardware hinaus wird als *Systemsimulation* bezeichnet.

Wir gehen im folgenden davon aus, dass alle für die Simulation 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 RCU über den PLB-Bus. Obwohl nicht der gesamte Funktionsumfang des PLB Protokolls benötigt wird, ist die Formulierung von Testzugriffen durch direktes Treiben von PLB-Signalen recht unhandlich. Zu diesem Zweck steht für das ML310 eine Bibliothek von Hilfsfunktionen bereit, die all diese Details abstrahieren. Listing 5 zeigt einen mit ihnen erstellten Testrahmen für unsere Beispielanwendung.

Listing 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                    49
    RunUntilInterrupt ;                                            50

    // Fahre Simulationsumgebung wieder herunter                   52
    Shutdown;                                                      53
  end                                                              54
endmodule                                                          55
```

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 RCU kodiert sind. Wir lesen einen 32b Wert von Byte-Adresse 40 (=24'h28). Dies entspricht einem Zugriff auf Register 10 (alle unsere Register sind 4B groß). Da wir nur drei Register in unserem Design verwenden, erwarten wir hier die in Zeile 118 von Listing 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 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 RCU (=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 RCU gestartet. Wir lassen die Simulation nun solange laufen, bis die RCU 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.

3.3.6 Software-Teil der Master-Mode Anwendung

Der in Listing 6 gezeigte Software-Teil der Master-Mode Version unseres Beispiels ist sehr ähnlich zur Slave-Mode Software.

Listing 6: Software-Teil der Master-Mode Anwendung

```
#include <stdio.h>                                                          1
#include <stdlib.h>                                                         2
#include "rcu.h"                                                            3




// Anzahl Datensätze in Ein- und Ausgabedatei                               8
#define NUM_WORDS 256*1024                                                  9

// Nummern der verschiedenen RCU-Register in diesem Entwurf                 11
#define REG_SOURCE_ADDR 0                                                   12
#define REG_DEST_ADDR 1                                                     13
#define REG_COUNT      2                                                    14
#define REG_START      3                                                    15

// Hauptprogramm                                                            17
main()                                                                      18
{                                                                           19
  // Ein- und Ausgabedateien                                                20
  FILE *infile, *outfile;                                                   21

  // Benutze vorzeichenlose Ganzzahlen (32b Worte) für alle Variablen       23
  unsigned long volatile *inwords, *outwords, *inwords_phys, *outwords_phys;24

  // Marker für Zeitmessung                                                 26
  unsigned long long start, stop, RTEMSIO_getTicks();                       27

  // Zeiger auf S0-Bereich mit RCU-Registern                                29
  unsigned long volatile *rcu;                                              30

  // RC initialisieren                                                      32
  rcu_init();                                                               33

  // Zeiger auf S0-Bereich mit RCU-Registern holen                          35
  rcu = rcu_get_s0(NULL);                                                   36

  // fordere Speicher für Ein- und Ausgabefelder an                         38
  // *_phys zeigt auf die physikalische Speicheradresse                     39
  // aus Sicht der Hardware                                                 40
  inwords = rcu_malloc_master(2 * NUM_WORDS * sizeof(unsigned long),        41
                              (void **) &inwords_phys);                     42
  outwords      = inwords      + NUM_WORDS;                                 43
  outwords_phys = inwords_phys + NUM_WORDS;                                 44

  if (!inwords || !inwords_phys) {                                          46
    fprintf( stderr, "out_of_memory\n");                                    47
```

```
    exit (1);                                                                    48
  }                                                                              49

  // Funktioniert der Slave Zugriff?                                             51
  printf ("Magic: %.8lx\n", rcu [28]);                                           52

  // Öffne die Dateien zum Lesen und Schreiben                                   54
  infile = fopen("test1.in", "r");                                               55
  outfile = fopen("test1.out", "w");                                             56

  // Lese komplette Eingabedatei in Eingabe-Speicherbereich                      58
  fread(inwords, sizeof(unsigned long), NUM_WORDS, infile);                      59

  // Merke Startzeit der Berechnung                                              61
  start = rcu_get_ticks ();                                                      62

  // "Ubertrage Parameter an RC (nicht die Daten selbst)                         64
  rcu[REG_SOURCE_ADDR] = inwords_phys; // Physikalische(!) Startaddresse         65
  rcu[REG_DEST_ADDR] = outwords_phys; // Physikalische (!) Zieladdresse          66
  rcu[REG_COUNT]     = NUM_WORDS; // Anzahl Datensätze                          67
  rcu[REG_START]     = 1;                 // Startkommando für RC               68

  // Warte auf Ende der Berechnung (wird über IRQ angezeigt)                     70
  rcu_wait ();                                                                   71

  // merke Stopzeit                                                              73
  stop = rcu_get_ticks ();                                                       74

  // Hardware Interrupt zurücksetzen durch beliebigen HW Lesezugriff             76
  rcu [28];                                                                      77

  // Schreibe das komplette Ausgabefeld in die Ausgabedatei                      79
  fwrite(outwords, sizeof(unsigned long), NUM_WORDS, outfile);                   80

  // Schließe Dateien                                                            82
  fclose( infile );                                                              83
  fclose( outfile );                                                             84

  // Gebe Speicher für Ein-/Ausgabefelder wieder frei                            86
  rcu_free_master ((void *) inwords);                                            87

  // Gebe Ergebnis der Zeitmessung in Mikrosekunden aus                          89
  printf ("Zeit  : %8lld us\n", (stop-start)/TICKS_PER_USEC);                    90
}                                                                                91
```

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

Das Hauptprogramm ist weitestgehend unverändert geblieben. Ein entscheidender Unterschied besteht aber in der Anforderung des Speicherbereiches für Ein- und Ausgabedaten in Zeilen 41–44.

Lösung	RCU-Taktfrequenz in MHz	RCU-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 2: Übersicht über alle Realisierungen der Beispielanwendung

Die CPU und die RCU adressieren den gleichen Speicher mit unterschiedlichen Adressen: Die CPU, die ja unter Linux virtuellen Speicher unterstützt, verwendet dabei *virtuelle* Adressen. Diese werden von einer Memory Management Unit (MMU) innerhalb der CPU dann in *physikalische* Adressen umgerechnet und auf den PLB ausgegeben. Die RCU hat aber keine MMU, sondern rechnet direkt mit *physikalischen* Adressen. Wir müssen also bei der Speicheranforderung die Adressen des Speicherbereiches in *beiden* Darstellungen bestimmen. Weiterhin muss sichergestellt werden, dass der erhaltene Speicherbereich tatsächlich im physikalischen Speicher präsent ist. Die RCU kann (wieder wegen der fehlenden MMU) nämlich nicht das Einladen noch fehlender Speicherseiten (*paging*) aus dem virtuellen Speicher veranlassen. Alle benötigten Seiten müssen vorher im physikalischen Speicher liegen. Beide dieser Anforderungen erfüllt die Funktion rcu_malloc_master. Wir fordern damit einen doppelt so großen Speicherbereich an, in dessen unterer Hälfte die Eingabedaten abgelegt werden. In die oberen Hälfte wird die RCU die bitverdrehten Daten schreiben. Die Funktion gibt als Wert die *virtuelle* Adresse zurück, die weiter im Software-Programm verarbeitet werden kann. Die für die RCU benötigte *physikalische* Adresse des Speicherbereiches wird in den, als Referenz an die Funktion übergebenen, letzten Parameter eingetragen. Der so erhaltene Wert kann dann später in Zeile 65 an die RCU übergeben werden. Die Zeigerarithmetik der Zeilen 43 und 44 dient lediglich dazu, den Beginn des für die Ausgabedaten reservierten Bereichs innerhalb des Speicherblocks zu bestimmen. Auch hier finden zwei Rechnungen, getrennt für die virtuelle und physikalische Adresse, statt.

Die Schleife, die in der Slave-Mode Lösung die Eingabedaten wortweise zur Umrechnung an die RCU übergeben hat, ist nun vollständig ersetzt worden. Stattdessen werden in den Zeilen 65–68 die Parameter des aktuellen Programmlaufes in die RCU-Register geschrieben. Der Schreibzugriff in Zeile 68 startet schließlich die RCU-Ausführung. Von nun an laufen CPU und RCU parallel! Da wir für unser Beispiel aber keine weiteren Aufgaben auszuführen haben, lassen wir die Software-Ausführung einfach ruhen und warten, bis die RCU mit ihren Berechnungen fertig ist. Dieser Effekt wird durch den Aufruf von rcu_wait() in Zeile 71 erreicht. Nach dem Auslösen eines CPU-Interrupts durch die RCU wird die Software-Ausführung in Zeile 74 wieder aufgenommen. In Zeile 77 wird durch einen Lesezugriff auf die RCU der Interrupt abgeschaltet (wie im HDL-Modell auf Zeilen 252–253, Listing 4 beschrieben). Der Programmablauf unterscheidet sich danach nicht mehr von der Slave-Mode Variante.

Wie schnell läuft nun die Master-Mode Lösung? Bei einer RCU-Taktfrequenz von 100 MHz wird ein Datensatz von 256Kw in **5813 μs** bearbeitet. Das ist fast 34-mal schneller, als auf dem 300MHz PowerPC in Software! Tabelle 2 zeigt alle Ergebnisse noch einmal in einer Übersicht.

4 Hochsprachen-basierte Programmierung

Nachdem wir in den letzten Abschnitten tief in die Niederungen der hardware-nahen Programmierung von ACSs abgestiegen sind, wollen wir uns in diesem Abschnitt mit einer der Alternativen befassen. Speziell wird es hier um die automatische Erzeugung von kombinierten Hardware-Software Anwendungen aus beliebigem ANSI C Code gehen.

Die folgende Darstellung orientiert sich an den Techniken, die im NIMBLE-Compiler verwendet werden. Dieser Prototyp ist das Ergebnis eines mehrjährigen Forschungsprojektes zwischen Synopsys Inc., der UC Berkeley und Mitarbeitern des FG ESA. Natürlich können wir die Materie hier nur im Überflug vorstellen, ähnlich wie beim Hardware-Design sind auch hier Tauchfahrten in die Tiefen des modernen Compiler-Baus möglich.

Listing 7: Beispielprogramm für Hardware-Compilierung von C

```
main(int argc, char *argv[])                                          1
{                                                                     2
  int i, j, k;                                                        3

  // Wert des ersten Arguments auf der Kommandozeile                  5
  j = atoi(argv[1]);                                                  6
  // Wert des zweiten Arguments auf der Kommandozeile                 7
  k = atoi(argv[2]);                                                  8

  for (i = 0; i < k; i++)                                             10
    {                                                                 11
      j = j * 13;                                                     12
      if (j > 1000000)                                                13
        printf("j=%d too large in loop i=%d\n", j, i);                14
    }                                                                 15

  printf("result: j = %d\n", j);                                      17
}                                                                     18
```

Als Beispiel für diesen Entwurfsfluß soll das in Listing 7 gezeigte C Programm dienen. Es berechnet den Wert $j \cdot 13^k$, wobei die Werte von j und k dem Programm beim Start auf der Kommandozeile übergeben werden. Als kleine Falle soll die Berechnung beim Auftauchen eines Zwischenergebnisses größer als 1000000 eine Warnung ausgeben. Im Programm werden in den Zeilen 6 und 8 die Kommandozeilenparameter in ganze Zahlen konvertiert und zur Initialisierung der Variablen verwendet. Da C keinen Exponentiationsoperator kennt, findet die eigentliche Berechnung in einer Schleife statt, in der das aktuelle Zwischenergebnis jeweils wieder mit 13 multipliziert wird (k-mal). Hier findet auch die Prüfung auf das Überschreiten unserer willkürlich gewählten Grenze statt (Zeile 13). Abschließend wird das Ergebnis der Berechnung auf der Konsole ausgegeben. Die folgenden Ausgaben zeigen eine Beispielausführung des Programmes:

```
1 [unix] $ ./exp 10 5 ↵
2 j=3712930 too large in loop i=4
3 result: j = 3712930
```

Durch das Kommando in Zeile 1 wird das Programm (hier mit dem Namen exp) gestartet. Bei den

gewählten Parametern soll es also den Wert $10 \cdot 13^5$ berechnen. Im Laufe dieser Berechnung wird im letzten der Schleifendurchläufe die von uns gewählte Grenze von 1000000 für das Zwischenergebnis überschritten und damit eine Warnmeldung in Zeile 2ausgegeben. Das korrekte Endergebnis wird als letztes zum Ende der Programmausführung (Zeile 3) ausgegeben.

4.1 Hardware-Software Partitionierung

Auch ohne größeres Profiling (das das NIMBLE-System auch komplett automatisch durchführen kann) ist im Beispiel die Erkennung des zeitintensiven Segments wieder trivial. Die in Zeile 10 beginnende Schleife verbraucht offensichtlich den Großteil der reinen Rechenzeit. An für sich scheint sie nach den Kriterien aus Abschnitt 3.2 ein guter Kandidat für eine Auslagerung in Hardware zu sein. Wenn da nicht die Ausgabe (printf in Zeile 14) wäre, die sich ja nicht sinnvoll in Hardware umsetzen läßt ...

Aber anstatt hier einfach aufzugeben, untersuchen wir, wie oft dieser Fall tatsächlich auftritt. Kommen wir dabei zu dem Ergebnis, dass bei unseren gängigen Datensätzen Zwischenergebnisse größer als 1000000 nur sehr selten auftreten, kann sich die Hardware-Auslagerung doch noch lohnen. Wir müssen nur den (hoffentlich seltenen) Fall der Grenzwert-Überschreitung trotzdem behandeln.

Der NIMBLE-Compiler stellt diese Untersuchungen anhand von durch den Benutzer zur Verfügung gestellten repräsentativen Sätzen von Eingangsdaten automatisch an. Im folgenden nehmen wir an, dass der Überlauf tatsächlich so selten auftritt, dass die Hardware-Auslagerung auch mit der Ausnahmebehandlung noch lohnend ist.

4.2 Ausnahmebehandlung

Abbildung 13 zeigt, mit welchem Verfahren der Compiler die Ausnahmebehandlung realisiert. Für alle in Hardware ausgelagerten Teile mit Ausnahmen (in der Zeichnung grau unterlegt) bleiben ihre ursprünglichen Software-Versionen bestehen. Bei Eintritt in den betroffenen Programmteil kann zur Laufzeit entschieden werden, ob die Hard- oder die Software-Version eines Programmteils verwendet werden soll (die Abfrage (a)). Bei Verwendung der Hardware-Version werden die dort benötigten Variablen (im Beispiel i und j) von der Software in ihre entsprechenden Hardware-Register kopiert. Dann startet die Hardware-Ausführung. Falls nun der Ausnahmefall (b) tatsächlich eintreten sollte, wird wieder auf die ursprüngliche Software-Version (c) des Programmteiles umgeschaltet. Dazu werden nur die dafür *benötigten* Hardware-Register (das müssen nicht immer alle sein!) wieder in die entsprechenden Software-Variablen umkopiert. Als kleiner Exkurs: In der Terminologie von optimierenden Compilern werden nur solche Variablen zwischen RCU und CPU transferiert, die auf dem Übergang zwischen RCU- und CPU-Ausführung *live* sind. In der Zeichnung wird auch deutlich, dass NIMBLE nach Behandlung einer Hardware-Ausnahme die aktuelle Schleifen-Iteration immer in Software zu Ende führt (d). Erst am Schleifenkopf (a) wird wieder entschieden, ob die nächsten Durchgänge in Hard- oder Software ausgeführt werden.

Der dafür zuständige Entscheidungsknoten (a) kann beliebig kompliziert ausfallen. Beispielsweise kann hier eine Statistik über das Auftreten des Ausnahmefalls beim aktuellen Eingabedatensatz geführt werden. Bei Überschreiten eines gewissen Grenzwertes könnte dann zur Laufzeit entschie-

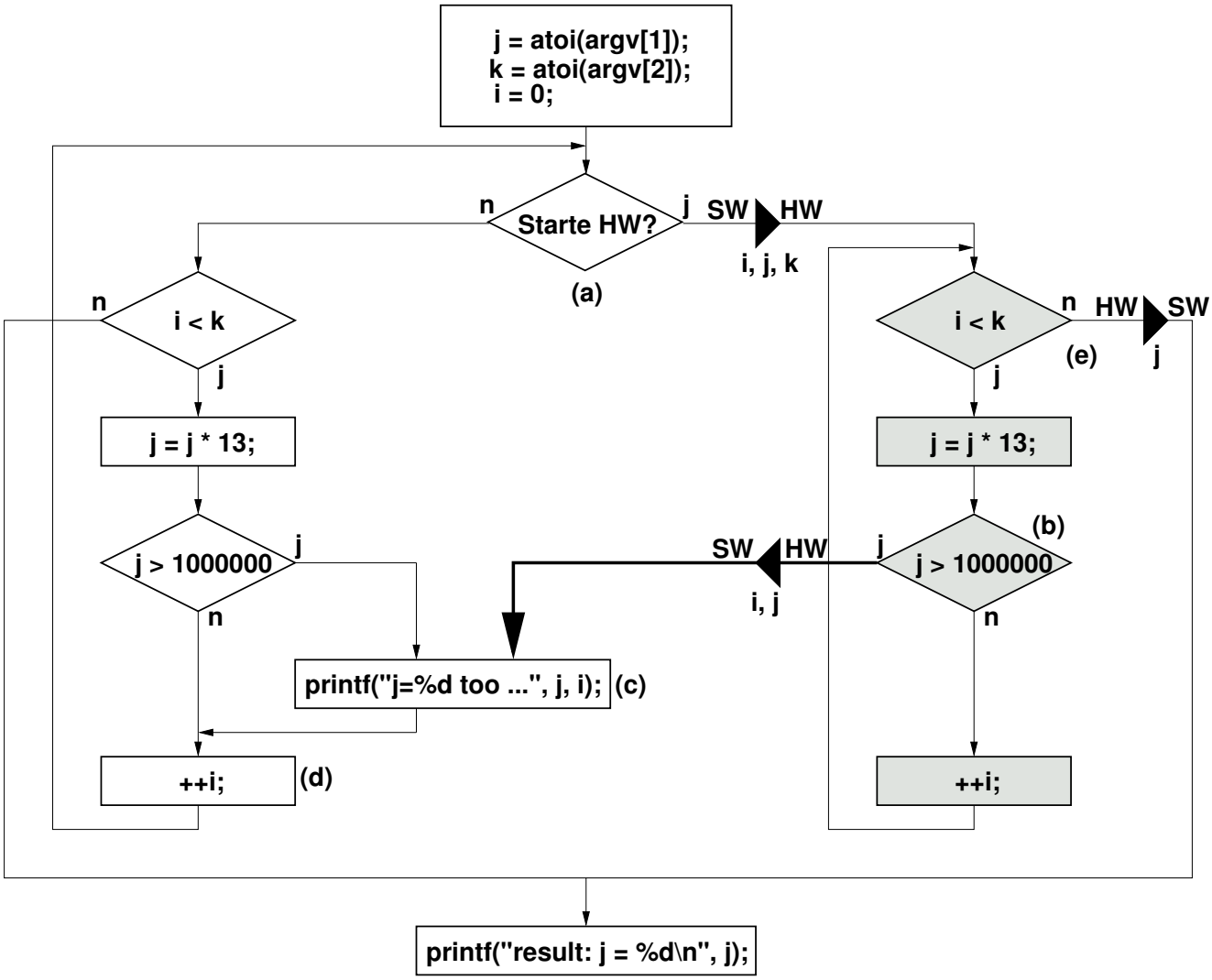


Abbildung 13: Übergänge zwischen Hard- und Software-Teilen

den werden, dass sich für diesen Datensatz die Hardware-Ausführung nicht mehr lohnt (zu hoher Kommunikationsaufwand für die Ausnahmebehandlung und das Wiederanfahren der Schleife in Hardware).

Für den Fall, dass die Hardware-Ausführung des Programmteils erfolgreich und ohne weitere Unterbrechungen beendet wird (e), müssen nur die im folgenden Programmteil benötigten Variablen wieder aus den entsprechenden Hardware-Registern befüllt werden. Man beachte, dass in unserem Beispiel die Variablen i und k nach Ende der Schleife im ganzen Programm nicht mehr verwendet werden (sie sind nicht mehr *live*). Es wird also an dieser Stelle nur die Variable j mit dem Inhalt ihres Hardware-Registers befüllt.

4.3 Aufbau eines Kontroll-Datenflußgraphen

Wir wollen nun betrachten, wie mit dem Kandidaten für die Hardware-Auslagerung (den grauen Knoten aus Abbildung 13 weiter verfahren wird. Nach verschiedenen Analyse- und Transformationsschritten entsteht ein Graph, der die Kontroll- und Datenabhängigkeiten der einzelnen Hardware-Operationen wiederspiegelt.

Dieser sogenannte CDFG ist in Abbildung 14 dargestellt. Ovale sind dabei Operationsknoten und Rechtecke Register. Dreiecke stehen für konstante Werte. Durchgezogene Kanten stellen Datenabhängigkeiten dar (bevor nicht alle Daten gültig sind, kann eine Operation nicht beginnen) und gestrichelte Kanten stehen für Kontrollabhängigkeiten. Ihr Ursprungsknoten muß in jedem Fall zeitlich vor dem Zielknoten ausgeführt werden (auch wenn sonst keine Datenabhängigkeiten zwischen den Knoten bestehen).

Der CDFG kann in drei wesentliche Teile unterteilt werden. Auf der rechten Seite ist der Mechanismus der for-Schleife aus Zeile 10 des ursprünglichen C-Programmes realisiert. Man erkennt den Schleifenzähler in Register i und die Inkrementierung um die Schrittweite 1. Wenn der Wert des Schleifenzählers nach der Inkrementierung größer oder gleich der Schleifenobergrenze (aus dem Register k) ist, wird der Hardware-Teil durch einen sogenannten Exit-Knoten verlassen. Auf der linken Seite befindet sich im oberen Teil ein Multiplizierer, der seinen Eingang (das Register j) mit der Konstanten 13 multipliziert. Zum Verständnis: Die Schiebe- und Arithmetikoperatoren realisieren die Funktion

$$((j << 4) - (j << 2)) + (j << 0) = (16j - 4j) + j = 12j + j = 13j$$

Man erkennt, dass ein solcher Konstanten-Multiplizierer wesentlich kleiner ist als einer, der mit variablen Werten multiplizieren kann. Einer der Vorteile von ACSs ist diese Spezialisierbarkeit von Operatoren auf die Erfordernisse der aktuellen Anwendung. Links unten in der Abbildung ist die Prüfung auf Überschreitung des Grenzwertes realisiert: Ist das Zwischenergebnis der Multiplikation größer als 1000000, wird der Hardware-Teil ebenfalls über einen Exit-Knoten verlassen. Das Laufzeitsystem kann dann abfragen, welcher der Exit-Knoten A und B tatsächlich gewählt wurde und so entsprechend reagieren (siehe dazu auch Abschnitt 4.5).

Der ganze CDFG hat nur eine Kontrollabhängigkeit, die aber für die Funktion der Hardware essentiell ist: Angenommen, die Kontrollkante zwischen den beiden Exit-Knoten würde fehlen. Dann könnte beim späteren Scheduling der Exit-Knoten B vor der Erkennung des Überlaufs ausgeführt werden (es gibt ja keine Datenabhängigkeiten zwischen den linken und rechten Teilgraphen). Das

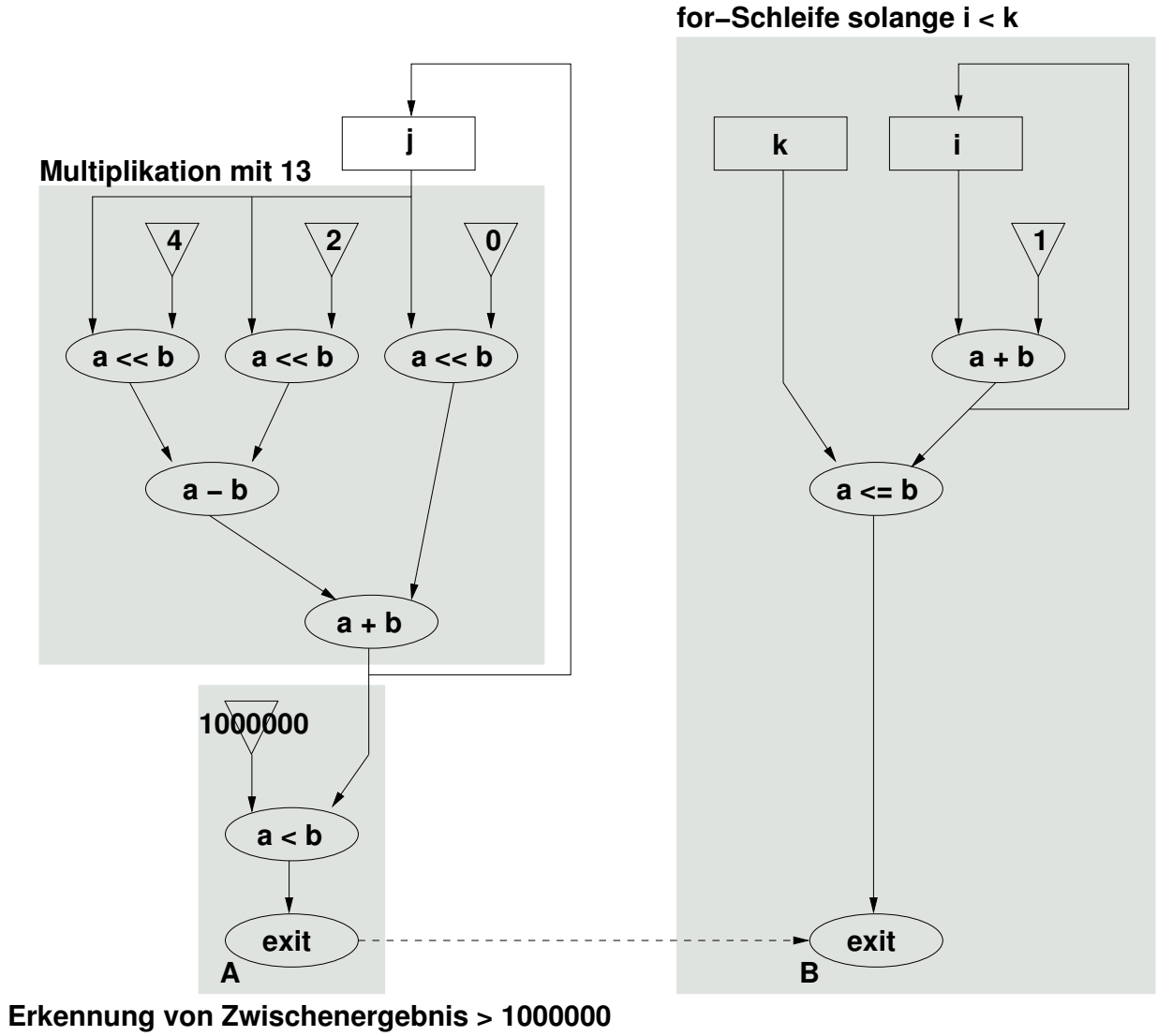


Abbildung 14: Kontroll-/Datenflußgraph der Beispielanwendung

Ergebnis wäre, dass ein Überlauf im letzten Schleifendurchlauf nicht mehr erkannt würde, sondern die Schleife normal beendet und über den Exit-Knoten B verlassen würde. Um diesen Fall zu vermeiden, wird durch Einziehen der Kontrollkante vom Exit-Knoten A zum Exit-Knoten B garantiert, dass ein Verlassen der Hardware wegen des Überlaufs *vor* dem Verlassen bei Erreichen des Schleifenendes ausgeführt wird.

4.4 Hardware-Implementierung

Nach weiteren Schritten wird schließlich aus dem CDFG eine konkrete Hardware-Architektur erzeugt. Deren Datenpfadkern und der zentrale Steuerautomat ist in Abbildung 15 dargestellt.

Zunächst beschreiben wir hier den Aufbau des Datenpfades. Wir erkennen in R2, R4 und R6 die Hardware-Register für die Software-Variablen j, i und k. Die CPU kann alle diese Register durch Setzen des entsprechenden WRITE_Rx Signals beschreiben (die Steuerung der Multiplexer ignorieren wir hier). Die aktuellen Inhalte der Hardware-Register R2 und R4 können von der CPU auch mittels der READ_Rx Signale zurückgelesen werden. Bei Register R6 ist dies nicht nötig: Die entsprechende Software-Variable k wird weder für die Ausnahmebehandlung noch nach dem Schleifenende benötigt, die dazugehörige Verdrahtung und Logik kann hier also gespart werden. Auch diese Art von Optimierung ist nur einem ACS möglich.

Auf R2 folgt der Konstant-Multiplizierer, durch die Register R0 und R1 aufgeteilt in zwei Stufen. Am Ende von R1 befindet sich der Komparator, der die Prüfung auf den Überlauf (Zwischenergebnis größer als 1000000) vornimmt. Falls dem so sein sollte, wird das Statusregister R5, das in der Software für den Exit-Knoten A steht, gesetzt und die Hardware-Ausführung angehalten (gesetztes HALT-Signal). Von R4 ausgehend findet sich die Logik für den Schleifenzähler und das Erkennen des Schleifenendes. R3 enthält den inkrementierten Schleifenzähler. Dieser wird mit der Obergrenze der Schleife (aus R6) verglichen. Falls die Obergrenze kleiner oder gleich dem inkrementierten Schleifenzähler ist, wird dies durch Setzen von R7 vermerkt. Dieses Register signalisiert der Software das Auftreten des Exit-Knotens B. Auch hier wird durch das ODER-Gatter das Steuersignal HALT gesetzt, das die Ausführung anhält.

Damit der Datenpfad aber tatsächlich die gewünschte Funktion berechnet, muß eine entsprechende Sequenz von Steuersignalen erzeugt werden. Dies geschieht durch den rechts unten in Abbildung 15 dargestellten Automaten.

Dieser Automat ist kein “normaler” Zustandsautomat. Der wesentliche Unterschied liegt darin, dass mehr als ein Zustand gleichzeitig aktiv sein kann (ähnlich einem Petri-Netz). Mit jedem Taktzyklus wird ein aktiver Zustand verlassen (deaktiviert) und alle damit verbundenen Zustände, bei denen eventuell vorhandene Bedingungen an den Verbindungskanten gültig sind. Im Beispiel ist die einzige vorkommende Bedingung !HALT. Diese ist wahr, solange der Steuerausgang HALT nicht gesetzt ist,

Wir haben diese Darstellung gewählt, da sie die Struktur der tatsächlich synthetisierten Hardware recht gut wiederspiegelt. Die Nummern der Zustände beschreiben die ENABLE_Rx-Signale, die gesetzt sind, wenn der Zustand x aktiv ist. Auf diese Weise wird der Datenpfad vom Automaten gesteuert.

Wenn die CPU die Berechnung durch ein Startkommando an die RCU gestartet hat, sind die beiden Zustände 0 und 3 aktiv. Das heißt, dass ENABLE_R0 und ENABLE_R3 gesetzt sind. Es wird also

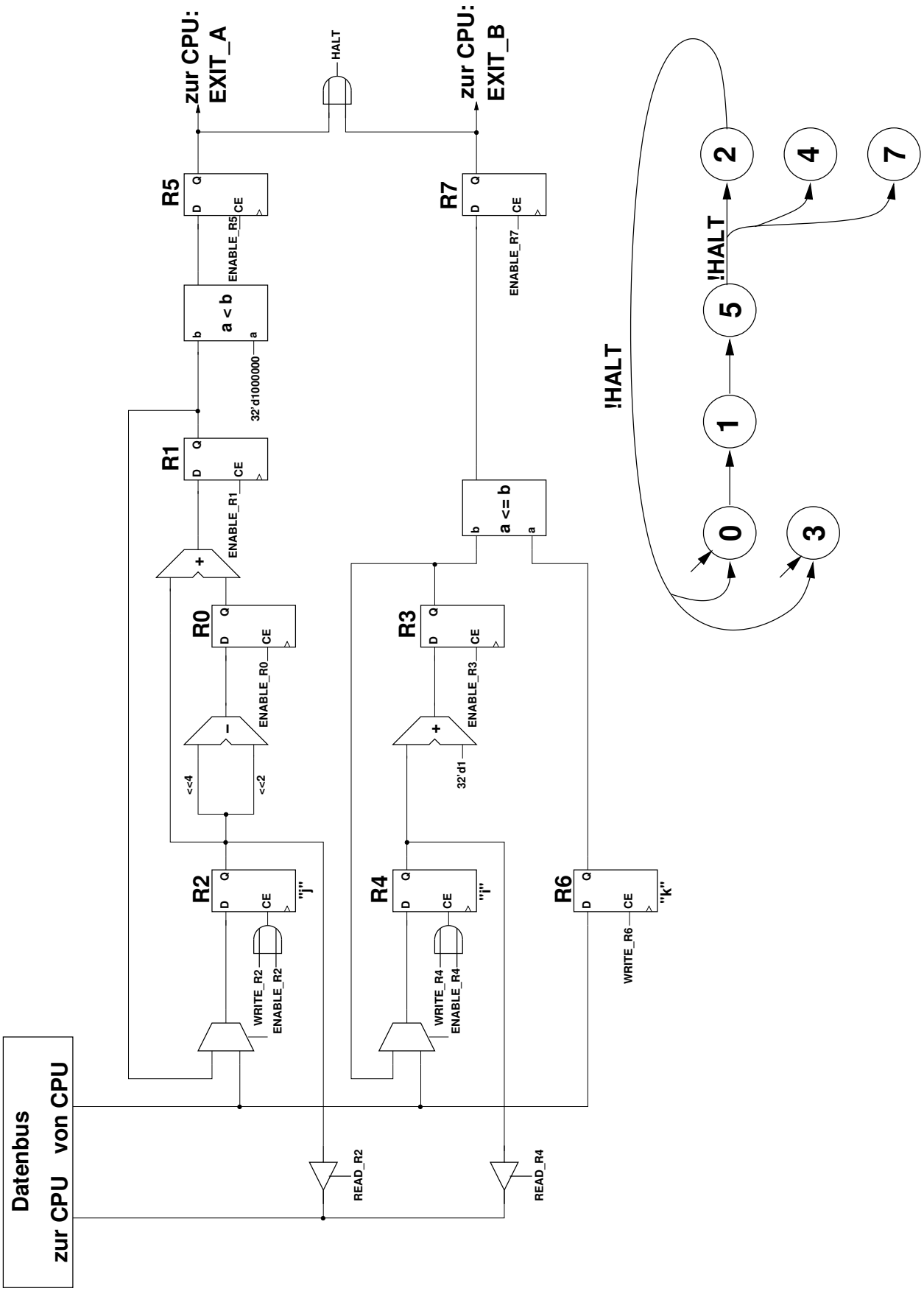


Abbildung 15: Datenpfad und Steuerungsautomat für Beispielanwendung

die erste Phase der Konstanten-Multiplikation mit 13 sowie das Berechnen des inkrementierten Schleifenzählers ausgeführt. Im nächsten Takt ist nur noch Zustand 1 aktiv. Damit wird die zweite Phase der Multiplikation ausgelöst, in R1 steht jetzt also das Multiplikationsergebnis. Im nächsten Takt ist nur Zustand 5 aktiv. Hier wird das Ergebnis der Prüfung auf Überlauf ausgewertet und ggf. der Datenpfad angehalten: Nach Setzen von HALT ist der gesamte Automat inaktiv. Erst bei einem Neustart durch die CPU würden wieder die Startzustände aktiviert. Falls kein Überlauf auftrat, werden im nächsten Takt die Zustände 2, 4 und 7 aktiviert. Dies führt zur Übernahme des neuen Zwischenergebnisses in R2 (die Variable j), die Übernahme des aktualisierten Schleifenzählers i in R4 und die Prüfung auf Erreichen des Schleifenendes (analog zum Überlauf). Falls der Datenpfad hier nicht angehalten wird, beginnt ein neuer Schleifendurchlauf wieder mit der Aktivierung der Zustände 0 und 3.

Die hier gezeigte Hardware-Architektur löst zwar die gestellte Aufgabe, sie ist aber sehr einfach und vergleichsweise wenig parallelisiert. Maximal drei Operationen laufen parallel ab (Zustände 2, 4 und 7). Auch Optimierungstechniken wie Pipelining oder Loop-Unrolling wurden für dieses Beispiel aus Übersichtsgründen nicht verwendet. Ein praktisch brauchbarer Compiler muß aber diese (und viele weitere) Verfahren ausschöpfen, um effiziente Hardware zu erstellen, die konkurrenzfähig zu manuell entworfenen Schaltungen ist.

Bis zu einer auf der RCU lauffähigen Hardware sind aber noch diverse Schritte erforderlich, die wir hier nur kurz aufzählen wollen. Dazu gehören beispielsweise die Erzeugung von Schaltungszellen für die einzelnen Operatoren und Register durch Modulgeneratoren, den geometrischen Aufbau des Datenpfad-Layouts durch reguläre Anordnung dieser Zellen in einer Floorplanning-Phase, und letztendlich der Standardfluß für FPGA-Design: Plazieren, Verdrahten und Bitstream-Generierung.

4.5 Software-Teil der Anwendung

Stattdessen beschäftigen wir uns mit dem noch fehlenden Teil der Anwendung, der auf der CPU laufenden Software-Seite.

Listing 8: Software-Seite der compilierten Lösung

```
// Definitionen für Hardwareumgebung
#define HW_START_REG 999
#define HW_EXIT_REG 998
#define HW_EXIT_A    1
#define HW_EXIT_B    2

main(int argc, char *argv[])
{
  // Variablen des Originalprogramms
  int i, j, k;
  // Die RCU rechnet hier mit vorzeichenbehafteten Zahlen
  volatile int *rcu;

  // Kommandozeilenparameter lesen und in Zahlen konvertieren
  j = atoi(argv[1]);
  k = atoi(argv[2]);
  // Schleifenz"ahler initialisieren
```

```
  i = 0;                                                                    18

  // RC initialisieren                                                      20
   rcu_init ();                                                             21
  rcu = rcu_get_s0 ();                                                      22

  // Immer Hardware-Version der Berechnung zuerst versuchen                 24

  // Software-Variablen in RCU-Register übertragen                          26
  rcu[2] = j;                                                               27
  rcu[6] = k;                                                               28
Loophead: // Sprungmarke für Neustart der Hardware nach Ausnahmebehandlung  29
  rcu[4] = i;                                                               30

  // RCU starten und auf Ende warten                                        32
  rcu[HW_START_REG] = 1;                                                    33
  rcu_wait ();                                                              34

  // Weshalb wurde die RCU-Ausführung beendet?                              36
  if (rcu[HW_EXIT_REG] == HW_EXIT_A) {                                      37
    // Ausnahmebehandlung: Überlauf im Zwischenergebnis                     38

    // Aktuelle Werte aus Hardware-Registern wieder in Software-Variablen holen  40
    j = rcu [2];                                                            41
    i = rcu [4];                                                            42

    // Schleifeniteration in Software zu Ende rechnen                       44
     printf ("j=%d too large in loop i=%d\n", j, i);                        45
    i = i + 1;                                                              46

    // Wieder an den Schleifenanfang für die nächste Iteration              48
    goto Loophead;                                                          49
  } else /* HW_EXIT_B */ {                                                  50
    // Normales Schleifenende erreicht                                      51

    // Endergebnis aus Hardware-Register in entsprechende Variable lesen    53
    j = rcu [2];                                                            54

    // Programm in Software zu Ende ausführen                               56
     printf ("result : j = %d\n", j);                                       57
  }                                                                         58
}                                                                           59
```

Listing 8 zeigt eine idealisierte Version des dafür nötigen C-Programmes. Die durch NIMBLE ebenfalls automatisch generierte Software-Seite der Anwendung ist nicht für menschliche Leser gedacht, sondern soll nur korrekt kompilierbar sein. Die gezeigte Version, die auch um schon bekannte Details wie den Interrupt Handler gekürzt wurde, demonstriert aber die wesentlichen Techniken.

Die Definitionen in den Zeilen 2–5 betreffen die bisher nicht beschriebene Hardware-Umgebung für Datenpfad und Steuerungsautomat. Sie werden später bei ihrer Verwendung erläutert. Das Lesen und Umwandeln der Kommandozeilenparameter (Zeile 15–16) wurde schon in der ursprünglichen C-Quelle vorgenommen. Da die for-Schleife im Laufe der Compilierung in Einzelanweisungen zerlegt wurde, wird die Schleifenvariable i in Zeile 21 separat initialisiert. Der Zeilenblock 21–22 enthält die bereits bekannten RCU Initialisierungen.

Der Kern der Anwendung beginnt in Zeile 27. Wir nehmen hier an, dass *immer* zuerst die Hardware-Version der Berechnung verwendet werden soll. Zu diesem Zweck schreiben wir die aktuellen Werte der in der Schleife benötigten Variablen j, k und i in ihre entsprechenden Hardware-Register R2, R6 und R4. Dies geschieht über Slave-Mode Schreibzugriffe mittels des Zeigers auf die Basis des S0-Bereichs, rcu.

In Zeile 33 starten wird die RCU. Dies ist eine der Funktionen der hier nicht beschriebenen RCU-Hardware-Umgebung. Nachdem diese Umgebung mittels eines Interrupts das Ende der Hardware-Ausführung angezeigt hat, wird die Software-Ausführung in Zeile 37 fortgesetzt. Hier wird das Statusregister der Hardware-Umgebung ausgelesen und geprüft, welcher der Exit-Knoten des CDFG benutzt wurde.

Wenn Exit-Knoten A benutzt wurde, liegt ein Überlauf des Zwischenergebnisses vor. Wir müssen also das nun erforderliche printf() in Software abarbeiten (Zeile 45). Vorher lesen wir aber noch die Werte der dafür benötigte Variablen aus den Hardware-Registern aus. Der NIMBLE-Philosophie folgend werden auch die restlichen Anweisungen dieser Schleifeniteration in Software gerechnet. Das ist in diesem Fall lediglich die Inkrementierung des Schleifenzählers in Zeile 46.

Nun müssen wir uns entscheiden, welche der beiden Versionen unserer Berechnung für die nächste Iteration verwendet werden soll. Bei diesem Programm folgen wir der Strategie, dass auch nach einer Ausnahmebehandlung in Software *immer* versucht werden soll, die nächste Iteration in Hardware zu rechnen. Zu diesem Zweck springen wir an die Marke Loophead (Zeile 29), die den Beginn einer neu gestarteten Iteration markiert. Wir haben an dieser Stelle eine kleine Optimierung vorgenommen, indem wir hier nur die Variablen, die in der Software-Version tatsächlich verändert wurden, neu an die RCU übertragen (also hier nur i in Zeile 30). Die anderen Variablen mussten nur einmal zu Beginn der gesamten Schleife transferiert werden. Danach wird die Hardware wie gewohnt neu gestartet.

Wenn bei unserem Beispielprogramm nach dem Ende der Hardware-Ausführung nicht der Exit-Knoten A genommen wurde, muß der Knoten B genommen worden sein. Dieser Fall wird im else-Zweig der Abfrage in Zeile 50 behandelt. Die Schleife ist also korrekt in Hardware beendet worden. Die Programmausführung kann ab hier in Software fortgeführt werden. Zuvor werden noch alle dafür benötigten Variablen aus ihren Hardware-Registern gelesen. Da nur noch j benötigt wird, wird nur R2 gelesen. Das Programm schließt mit der Ausgabe des Berechnungsergebnisses.

Dieses kleine Beispiel bringt in Anbetracht der langen Kommunikationszeit zwischen CPU und RCU und dem Fehlen fast aller Optimierungschritte keine Beschleunigung gegenüber der reinen Software-Ausführung. Es sollte aber seinen Zweck, alle wesentlichen Verfahren übersichtlich zu demonstrieren, erfüllt haben.

4.6 Ausblick

Der Compilerbau für adaptive Rechensysteme ist ein ungeheuer ergiebiges Forschungsgebiet. Neben dem Übergriff auf einige der klassischen Schwerpunkte der EDA-Forschung wie Logiksynthese, Modulgenerierung und Floorplanning wird die auch gesamte Rechnerarchitektur umfasst: Dies reicht vom konkreten Schaltungsentwurf (z.B. die Hardware-Umgebung für die compilierten Schaltungen oder die effiziente Realisierung einzelner hochoptimierter Module) bis hin zu grundlegenden Architekturfragen: Durch die Flexibilität der RCU kann für jedes Problem theoretisch die exakt passende Architektur generiert werden. Dabei kann die ganze Bandbreite von schon erprobten Techniken flexibel ausgeschöpft werden: SIMD, MIMD, spekulative Ausführung, tiefes Pipelining, mächtige Speichersysteme (auch MARC kratzt hier nur an der Oberfläche), der echte Parallelbetrieb von CPU und RCU, die Verwendung von teilweise seriellen Recheneinheiten und vieles mehr sind hier möglich. Diesen riesigen Lösungsraum zu durchforsten und schnell einen "guten" Kompromiss zu finden ist eines der Hauptprobleme der Werkzeuge. Aber damit endet die Arbeit nicht: Nach der Festlegung auf eine Hardware-Architektur für das aktuelle Problem steht nun eine Unmenge von Methoden aus dem klassischen Compiler-Bau zur Verfügung, effizient Algorithmen auf die gewählte Architektur abzubilden.

Es ist nicht realistisch zu erwarten, dass irgendein Werkzeug in endlicher Zeit tatsächlich für beliebige Algorithmen die optimale Abbildung findet. Schon der Versuch, automatisch compilierte Lösungen mit den besten manuell erstellten zu messen, endet in der Regel mit dem wohlverdienten Triumph der menschlichen Designer.

Aber davon sollte sich der aufstrebende Compiler-Bauer nicht entmutigen lassen: In der Praxis wird Perfektion nur ganz selten verlangt (und bezahlt). Lösungen müssen für die gestellten Anforderungen nur *gut genug* sein. Und wenn ein automatischer Entwurfsfluß solche Lösungen schneller und zuverlässiger liefern kann als ein menschlicher Designer, dann haben auch diese Verfahren ihre Daseinsberechtigung. Diese Aussage kann leicht an der allgemeinen Entwicklung von Werkzeugen, sowohl für den Hardware-Entwurf (z.B. Schaltungssynthese, automatische Plazierung und Verdrahtung) als auch im Software-Bereich (z.B. Hochsprachen-Compiler, automatische Speicherverwaltung), nachvollzogen werden. Der Trend geht eindeutig zu Werkzeugen, die dem Entwickler immer höhere Entwurfsebenen effizient nutzbar machen.