Technische Universität Dresden

Institut Betriebssysteme der Fakultät Informatik

Großer Beleg

Entwicklung eines Simulationsmodells für einen I/O-Prozessor

Vorgelegt von:	Marcel Wappler
Matrikelnummer:	2366080
Betreuender Hochschullehrer:	Prof. Dr. H. Härtig
Institut:	Betriebssysteme, Datenbanken, Rechnernetze
Lehrstuhl:	Betriebssysteme
Betreuer:	Dipl.-Ing. Frank Engel Dipl.-Inf. Sebastian Schönberg
Ort, Datum	Dresden, den 06.11.2000

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Viele der neuen Datenübertragungsverfahren mit immer höherer Übertragungsgeschwindigkeit bringen stark steigende Anforderungen an die oft eingesetzten Digitalen Signal Prozessoren (DSP) mit sich.

Da für eine signifikante Erhöhung der Übertragungsgeschwindigkeiten meist auch ein aufwendigeres Kanalkodierungsverfahren eingesetzt werden muß, steigt die benötigte Rechenleistung der Komponenten nicht nur proportional, sondern exponentiell mit der Datenrate an. Die mit der Taktrate quadratisch wachsenden Größen Energieverbrauch / Verlustleistung führen vor allem bei mobilen Anwendungen mit Batteriebetrieb dazu, daß allein durch eine Taktfrequenzerhöhung von DSPs oder Peripheriebausteinen keine entscheidende Leistungssteigerung mehr möglich ist.

Verschiedenste Ansätze können marginale bis dramatische Verbesserungen der Rechenleistung bei gleicher Taktfrequenz erreichen. Forschungen basierend auf einem dieser Ansätze brachten eine spezielle Ein-/ Ausgabeeinheit, den I/O Prozessor (IOP) hervor. Dieser Prozessor bildet die Grundlage der Belegarbeit.

1.2 Aufgabenstellung

Im Sonderforschungsbereich 358 (SFB358) für Automatisierten System-Entwurf, Teilprojekt A6, Mobilfunk wird an Entwurfssystemen für skalierbare Prozessorfamilien zur Signalverarbeitung geforscht. Der schon in Anwendung befindliche I/O Prozessor ist ein Teilprojekt.

Für den IOP sollte ein VHDL Simulationsmodell entwickelt werden, das bei Skalierung des Prozessors leicht anpaßbar ist und innerhalb eines Entwurfssystems genaue

Voraussagen über das Verhalten solcher I/O Prozessoren machen kann. Zu diesem Zweck mußte die Struktur und Funktion der vorhandene Hardware analysiert, ein Simulatormodell mit Hinblick auf gute Skalierbarkeit definiert und dann Prozessorelemente in diesem Modell implementiert werden. Der Umfang der Implementierung war mit dem Designer des IOP abzustimmen.

1.3 Aufbau des Dokuments

Im Kapitel 2 wird der I/O Prozessor mit seinen Komponenten und der DSP-Umgebung vorgestellt und analysiert. Dabei werden einige neuartige Konzepte der beschriebenen Hardware eingeführt und deren Vorteile erläutert. Jede Komponente des IOP wird einzeln im Detail besprochen.

In Kapitel 3 wird auf die sich aus der Aufgabenstellung ergebenden Anforderungen eingegangen und es werden wichtige Entwurfsentscheidungen begründet.

Kapitel 4 geht ins Detail und beschreibt verwendete Datenstrukturen, Testlaufbeispiele oder Probleme und Besonderheiten der Komponenten, so diese Einzelheiten von größerem Interesse sind. Sonstige Einzelheiten, auf die hier wegen fehlendem Platzes nicht eingegangen werden kann, sind entweder Kapitel 2, Analyse der Architektur, oder den ausführlich dokumentierten Quellen des Modells zu entnehmen.

Zum Abschluß werden die Ergebnisse der Arbeit in Kapitel 5 zusammengefaßt und es wird versucht, Optimierungsmöglichkeiten aufzuzeigen.

Anhang A: Ein Beispielprogramm dokumentiert ein kurzes Maschinenprogramm mit zwei nebenläufigen Tasks, dem zugehörigen Hexfile und einen Ausschnitt aus dem Simulator-Dump.

Um die Lesbarkeit der Arbeit zu verbessern, wurde auf Fußnoten verzichtet. In Anhang B: Glossar sind alle wichtigen Begriffe, die im Dokument verwendet werden, nocheinmal in kurzer Form erläutert.

Das Gesamte Projekt ist auf einer CD-ROM beigefügt. Sie enthält dieses Dokument in den Formaten Adobe PostScript, Adobe Acrobat, StarOffice 5.1, die kommentierten Quellen als .VHD Files und als Active VHDL Projekt. listet alle enthaltenen Verzeichnisse und Dateien auf.

Teile des Projekts sind auch über die WWW-Seiten des Autors verfügbar. [WAP99]

1.4 Danksagung

Ich möchte mich an dieser Stelle recht herzlich bei Herrn Prof. Gerhard Fettweis vom Lehrstuhl 'Mobile Kommunikationstechnik' an der Fakultät Elektrotechnik und bei Herrn Prof. Hermann Härtig, Leiter der Gruppe Betriebssysteme an der Fakultät Informatik für Ihre Unterstützung dieser fachübergreifenden Arbeit bedanken.

Ganz speziell bedanken möchte ich mich bei Frank Engel vom Lehrstuhl 'Mobile Kommunikationstechnik', ohne dessen I/O Prozessor und die fruchtbaren Diskussionen darüber es diese Arbeit nicht geben würde. Erwähnen möchte ich auch Ulrich Walther vom selben Lehrstuhl, mit dessen großzügiger Hilfe ich die Tücken der Sprache VHDL meisterte und der mir bei manchem unlösbar erscheinenden VHDL Programmierproblem den sprichwörtlichen Gordischen Knoten im Kopf durchschlug.

Außerdem möchte ich mich bei all meinen Freunden und Bekannten für ihre Freundschaft und Unterstützung bedanken.

Marcel Wappler

Dresden im Sommer 1999

2 Analyse der Architektur

2.1 M3 - Ein neues DSP-Konzept

In konventionellen DSP-Designs kann man bei der Abarbeitung von Algorithmen auf Datenströmen, die bei Anwendungen wie z.B. ADSL oder UMTS auftreten, einen hohen Overhead von Lese-/ Schreib-, Ein- und Ausgabeoperationen, Bit-Manipulationen oder Speichertransfers auf den eigentlichen Signalverarbeitungsalgorithmus beobachten. Könnte man einen auf Daten- und Datenstrommanipulationen spezialisierten, aber frei programmierbaren Prozessor statt des herkömmlichen DSP-Kerns für diese Operationen einsetzen, so würde die gesamte Datenverwaltung im Idealfall parallel zu den Berechnungen des DSP-Kerns ausgeführt. Sofort werden entscheidende Vorteile sichtbar:

- Es verbleibt mehr DSP-Rechenzeit für den eigentlichen Algorithmus. Geht man z.B. Von einem rein theoretischen Overhead von 50% aus, so könnte eine Kombination aus DSP-Kern und 'Datenstrom-Prozessor' bei halber Taktrate und damit nur einem Viertel des Energieverbrauches den gleichen Durchsatz wie ein herkömmliches DSP-Design erreichen.
- Geht man weiter und nimmt an, daß ein solcher Datenstrom-Prozessor Daten direkt auf Bitebene adressieren kann, so fallen weitere Operationen weg, die bei Algorithmen für herkömmliche DSP's ausschließlich zur Ausrichtung der interessierenden Bitbereiche an den Datentypgrenzen des jeweiligen DSP-Designs dienen.
- Weiterhin verringert sich die Komplexität des DSP-Kerns, da nur noch reine Register/Speicher und Register/Register-Befehle zu implementieren sind und alle Datenstrommanipulationen ausgelagert werden. Diese geringere Komplexität der Knoten ist eine entscheidende Voraussetzung für die Realisierung massiv paralleler Verarbeitung.

Das im SFB 358 in Erprobung befindliche DSP-Konzept 'M3' basiert auf diesen Erkenntnissen und führt neben der massiv parallelen Struktur des DSP-Kerns eine Datenflußsteuereinheit, den I/O Prozessor (IOP) ein.

2.2 I/O Prozessor

Dieser zusammen mit DSP-Kern, Speicher und weiteren Komponenten auf dem Die befindliche frei programmierbare Prozessor ist für alle Datenstromoperationen des DSP zuständig. Das Konzept der vollständigen Trennung zwischen Algorithmenverarbeitung einerseits und Datenstromverarbeitung andererseits schirmt den auf numerische Berechnungen optimierten DSP-Kern von Verwaltungsaufgaben vollständig ab. Alle Ein/Ausgabe- und Datenmanipulationsoperationen, umfangreiche Speichertransfers, Datenformatkonvertierungen, Bitmanipulationen etc. werden von dem neuartigen I/O Prozessor ausgeführt.

Der RISC-Prozessor mit Harvard- Architektur und einem 2 Operanden-Befehlssatz, hat auch die alleinige Regie über alle peripheren Einheiten, wie Parallele und Serielle Ports, Interruptquellen etc.

Der Entwurf basiert auf einer speziellen Bitadressierungs-Architektur, die es ermöglicht, Datenströme auf Bitebene effizient manipulieren zu können. Die Architektur verarbeitet Daten von 16 Bit Breite und kann diese auf Bitebene adressieren. Dadurch ist es möglich, 16 Bit-Worte an beliebigen Stellen im Speicher, ohne Rücksicht auf Wortgrenzen direkt anzusprechen. Details sind in den Kapiteln 2.3 Program Control Unit (PCU) und 2.10 Recheneinheit (ALU) erläutert.

Der IOP bietet Realtime-Multitasking in Hardware. Die neuartigen Konzepte von Scheduler und Registerfile erlauben taktfeines Umschalten zwischen Tasks. Dies ist möglich, ohne daß irgendwelcher Overhead, wie Register und Prozessorkonfiguration sichern, neuen Prozessorstatus laden etc., anfallen würde. Weiterhin kann ein Task über das Registerfile sehr leicht Daten mit anderen Tasks austauschen, gemeinsam nutzen oder deren Programmablauf beeinflussen. Allerdings gibt es keine Mechanismen zum Schutz der Tasks voreinander.

In Computersystemen birgt solch ein fehlender Schutz meist erhebliche Sicherheitsprobleme, weshalb sich bei der Programmierung von Standard-Prozessoren ein derartiges Konzept verbietet. In Geräten jedoch, die nur einmal bei ihrer Herstellung maskenprogrammiert, oder während der Lebensdauer höchstens ein paar Mal unter kon-

trollierten Bedingungen neu programmiert werden, ist ein Schutz von Prozessen voreinander in den meisten Fällen nicht erforderlich.

Eine konsequente Nutzung der neuen Konzepte ermöglicht es, die verfügbare Rechenzeit des Prozessors besser ausnutzen und mit den Vorteilen des Multitasking kombinieren zu können, ohne dabei jedoch mit den üblichen Nachteilen von Multitasking-Systemen konfrontiert zu sein. Details werden in Kapitel 2.5: Scheduler erläutert.

Der Einsatz des IOP in einem für eine spezielle Anwendung konfektionierten DSP (und damit im Verhältnis zur Lebensdauer sehr geringen Anzahl an Neuprogrammierungen) sowie seine den Anforderungen nach energiesparende und deshalb einfach aufgebaute Struktur machen eine aufwendige Behandlung von aus der Pipeline-Struktur resultierenden Anomalien überflüssig oder verbieten sie sogar; Die Pipeline ist an der Programmierschnittstelle sichtbar. Daten manipulierende Befehle zeigen erst 3 Takte nach deren Fetch Auswirkungen auf die jeweilige Resource; Ein bedingter oder unbedingter Sprung hat erst drei Takte nach dessen Fetch Auswirkungen auf den PC. Bei einem Assembler-Entwurf der Firmware oder der Entwicklung eines Compilers für den I/O Prozessor ist das zu beachten.

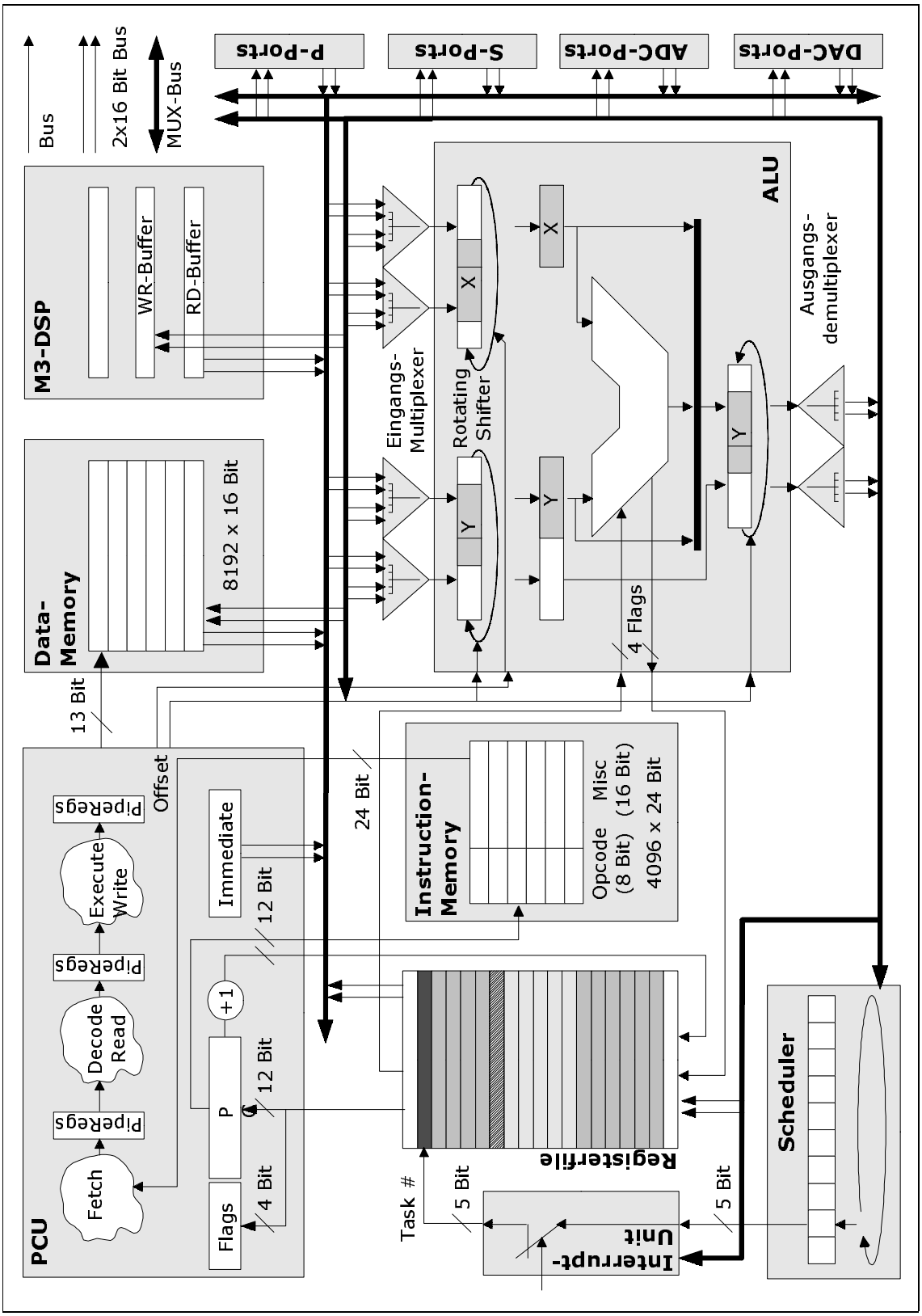


Abbildung 2.1: Schematischer Aufbau des I/O Prozessors

2.3 Program Control Unit (PCU)

Der gesamte Programmablauf des IOP wird von der PCU gesteuert. Um den Durchsatz von einem Befehl je Taktzyklus auch bei hohen Taktfrequenzen zu erreichen, wurden die komplexen Abläufe im IOP in die drei Pipeline-Phasen Fetch, Decode/Load und Execute/Store unterteilt (s. Abbildung 2.2).

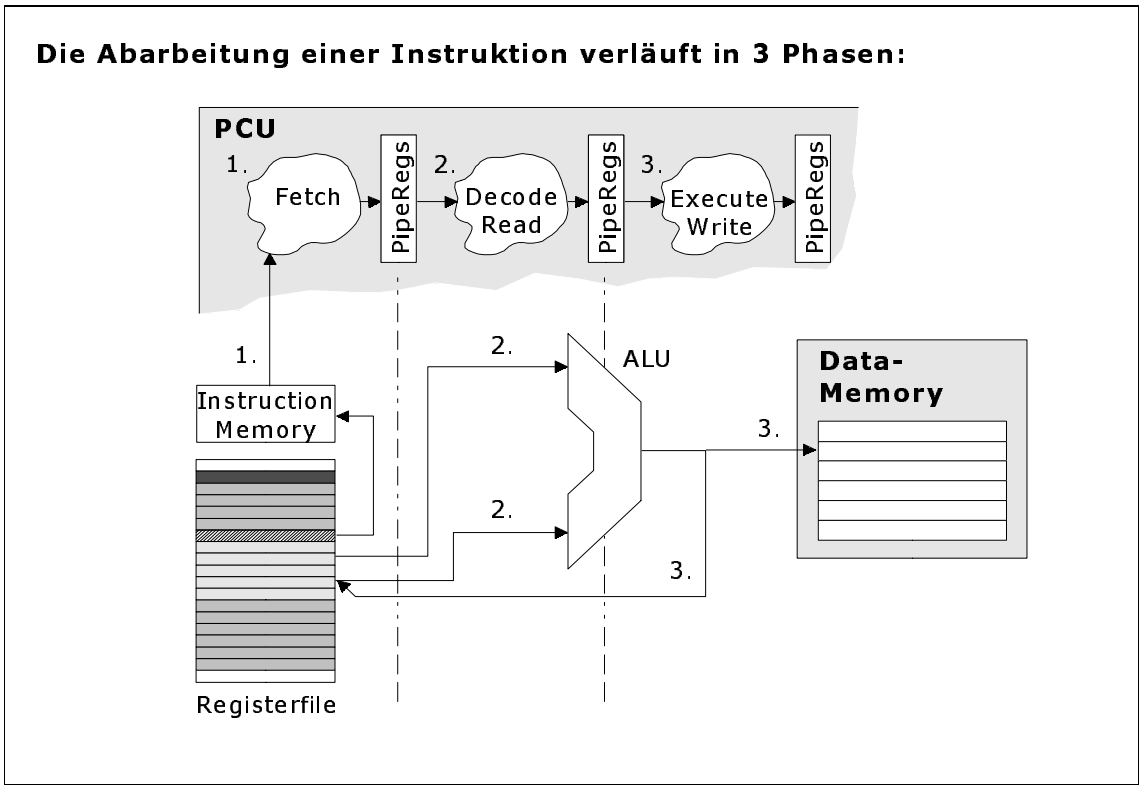


Abbildung 2.2: Die Pipeline-Struktur des IOP

In der **Fetchphase** wird eine neue Instruktion in die Befehls-Pipeline eingetragen. Je nach Instruktion unterscheidet sich der folgende Ablauf in der **Decode- und Loadphase:** Bei Register-Befehlen wird aus der Instruktion die Registerfile-Adresse einer oder beider Operanden-Register X und Y extrahiert und ans Registerfile gelegt. Gleiches geschieht bei Instruktionen, die die Ressourcen Datenspeicher, Ein- und Ausgabeports, Scheduler, Interrupt Unit und DSP-Buffer betreffen. Bei indirekter Speicheradressierung wird die Adresse aus dem Inhalt des Indexregisters und der Instruktion selbst errechnet. Weiterhin werden je nach Instruktion die Eingangsmultiplexer der ALU auf die Quellenressource geschaltet. Eine Pipelinestufe später in der **Execute- und Writephase** wird die schon vorher dekodierte Adresse der Zieleinheit ausgegeben und der Demultiplexer am ALU-Ausgang auf diese Einheit umgeschaltet. Das Ergebnis der Berechnung wird übernommen und ein Befehlszyklus ist beendet.

Abbildung 2.2 illustriert als Beispiel die folgenden zwei Datenpfade

- Registerfile ⇨ ALU ⇨ Registerfile und
- Registerfile ⇨ ALU ⇨ Datenspeicher.

Da jede Stufe der Pipeline parallel zu den anderen arbeiten kann, sind jederzeit immer alle drei Stufen aktiv. Während eines Taktes wird also eine neue Instruktion gefetcht, die vorhergehende Instruktion dekodiert und die Ergebnisse einer weiteren Instruktion werden zurückgeschrieben. So ist es möglich, daß 1 Befehl im Durchschnitt innerhalb von 1 Takt abgearbeitet wird, obwohl zur Abarbeitung eines einzelnen Befehls genau 3 Takte benötigt werden.

2.4 Registerfile

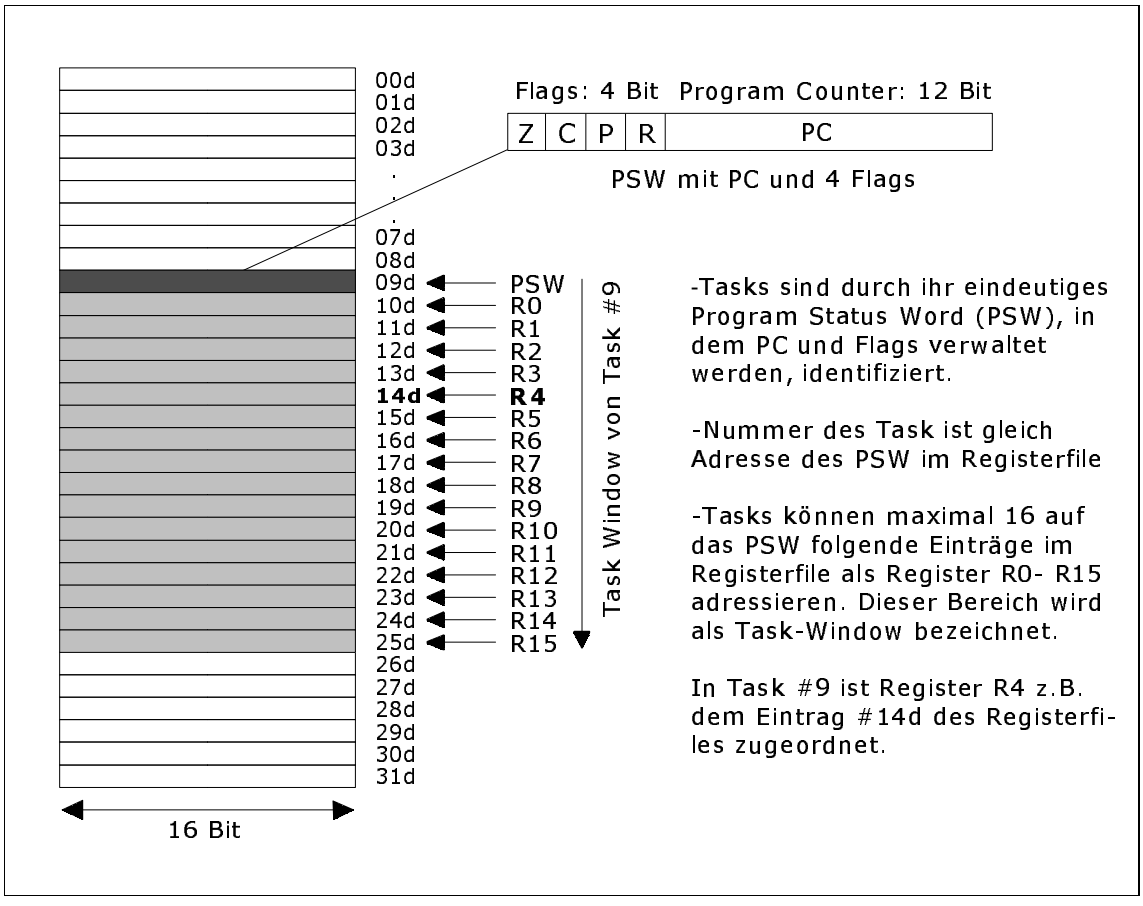


Abbildung 2.3: Registerfile und Task Window

Formal gesehen ein einfacher Speicher mit 32 Einträgen (Registern) á 16 Bit Breite und verschiedenen Lese- und Schreibports, Adreß- und Steuerleitungen ist das Regis-

terfile konzeptionell neuartig. Aufbau, Struktur und Arbeitsweise des Registerfiles sind wesentliche Voraussetzungen des verwendeten Multitasking-Konzeptes des IOP. Im Task Window (s. Abbildung 2.3) werden alle Informationen eines Tasks, wie Daten, Steuerinformationen und Prozeßkontext vermischt gespeichert.
Jeder Task wird anhand der Adresse seines Program Status Word (PSW) identifiziert. Das PSW besteht aus dem Program Counter (PC) und den Flags Carry, Zero, Page und Run. Die Flags Carry (Berechnung erzeugte einen Überlauf) und Zero (Ergebnis ist gleich 0) werden von logischen und arithmetischen Instruktionen gesetzt. Details dazu sind im Kapitel 2.10: Recheneinheit (ALU) zu finden. Das Page-Flag ist das Most Significant Bit (MSB) aller berechneten Speicheradressen des Tasks. Ist das Run-Flag gleich '0', so schläft der Task, bis ein Interrupt auftritt (s. dazu die Kapitel 2.5 und 2.8 über Scheduler und Interrupt-Unit).

Das PSW des Task steht in dessen Task Window an erster Stelle. Die 16 darauf folgenden Einträge des Registerfile sind für den Task auf die logischen Register R0 bis R15 gemappt (s. Abbildung 2.4). Mit Ausnahme von R0, das neben seiner Funktion als General Purpose Register auch als Indexregister bei indirekter Adressierung genutzt

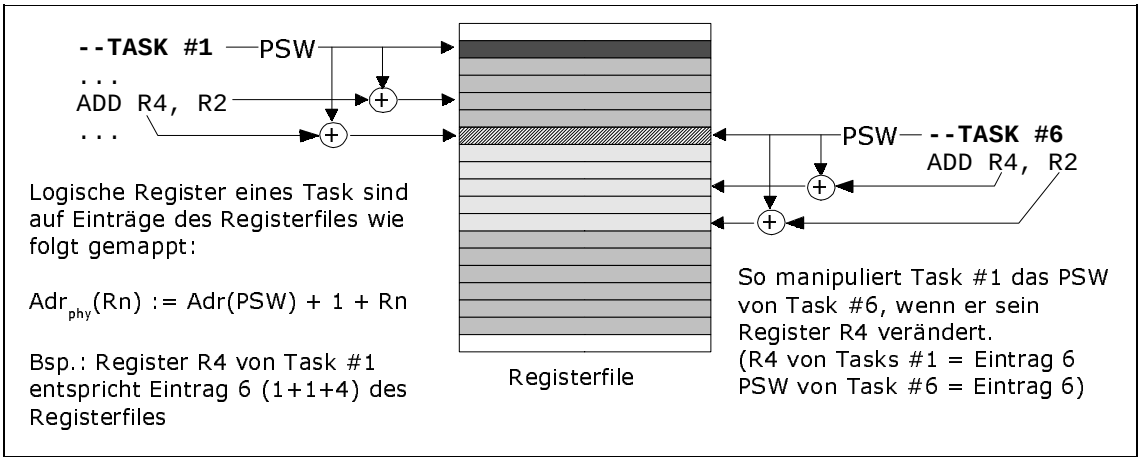


Abbildung 2.4: Mapping von logischen auf physikalische Register

wird, sind die 16 Register gleichwertig bei allen Operationen verwendbar. Da das Registerfile nur 32 Einträge besitzt, wäre nach dem PSW von Task #31 kein Eintrag für dessen Indexregister mehr verfügbar. Deswegen gibt es im IOP maximal 31 Tasks.

Bei der Taskverwaltung greifen Registerfile und Scheduler eng ineinander. Im Kapitel 2.5 sind dazu weitere Einzelheiten nachzulesen.

Die reale Größe des Task Windows wird nur durch die Nutzung seiner Register durch den Task definiert. Er kann nur ein Register nutzen oder auch alle 16.

Wie Abbildung 2.4 und Abbildung 2.5 verdeutlichen, können sich die Task Windows

verschiedener Prozesse auch überschneiden. Somit kann ein Prozeß Daten eines anderen lesen, schreiben oder sogar dessen PC und Flags ändern.

Alle Adressen zur Auswahl der Registers, die gelesen oder geschrieben werden sollen, werden mit Ausnahme des PC's von der PCU generiert.

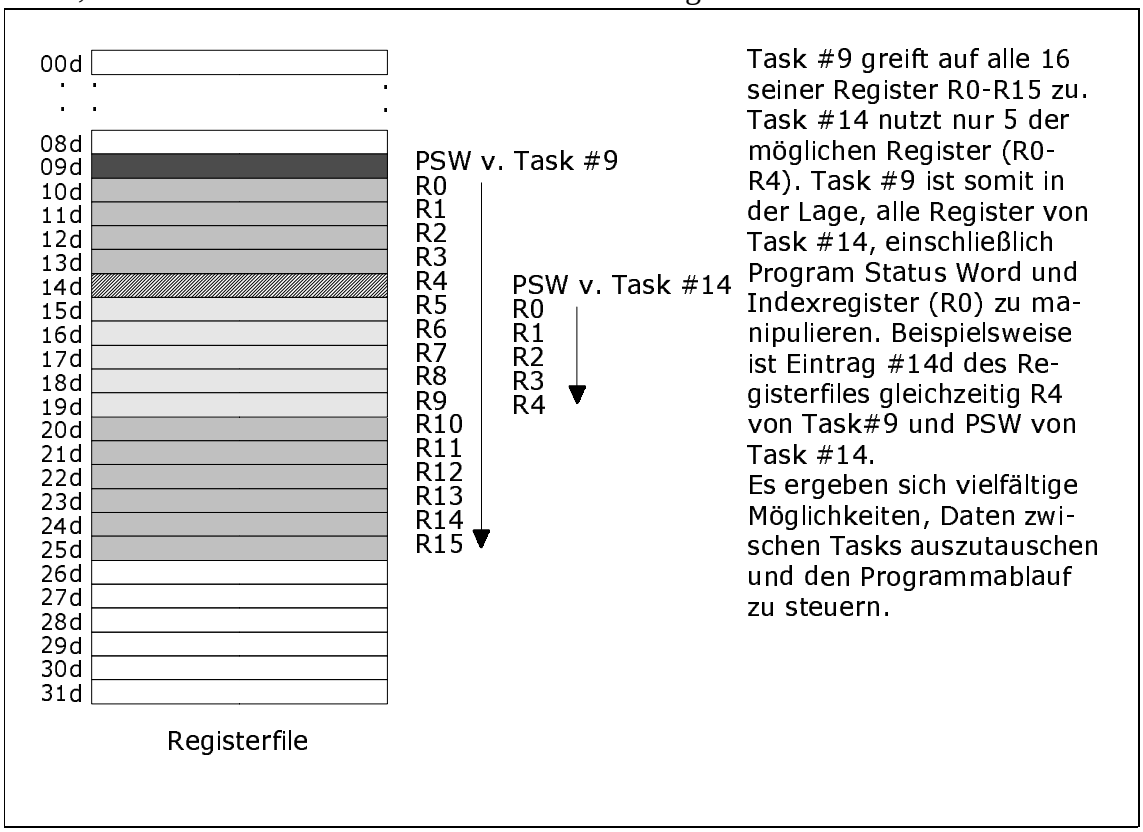


Abbildung 2.5: Nutzung des Registerfiles durch zwei Tasks

2.5 Scheduler

Die gesamte verfügbare Rechenzeit verteilt ein konzeptionell neuartiger, am Institut für Mobile Nachrichtentechnik entwickelter Scheduler auf alle Tasks. Zusammen mit dem ebenfalls neu entwickelten Registerfile-Konzept ermöglicht es der Scheduler, ohne jegliche Latenzzeit zwischen Tasks umzuschalten. Es ist möglich, innerhalb von n Takten n verschiedene Tasks auszuführen. Das feinkörnige Scheduling eröffnet neue, bisher nicht möglich gewesene Wege zur effizienten Nutzung der vorhandenen Ressourcen. Beispielsweise kann während der durch die Pipeline bedingten 3 Takte Verzögerung eines Ladebefehls ein anderer Task 2 Takte Rechenzeit erhalten. Wenn der

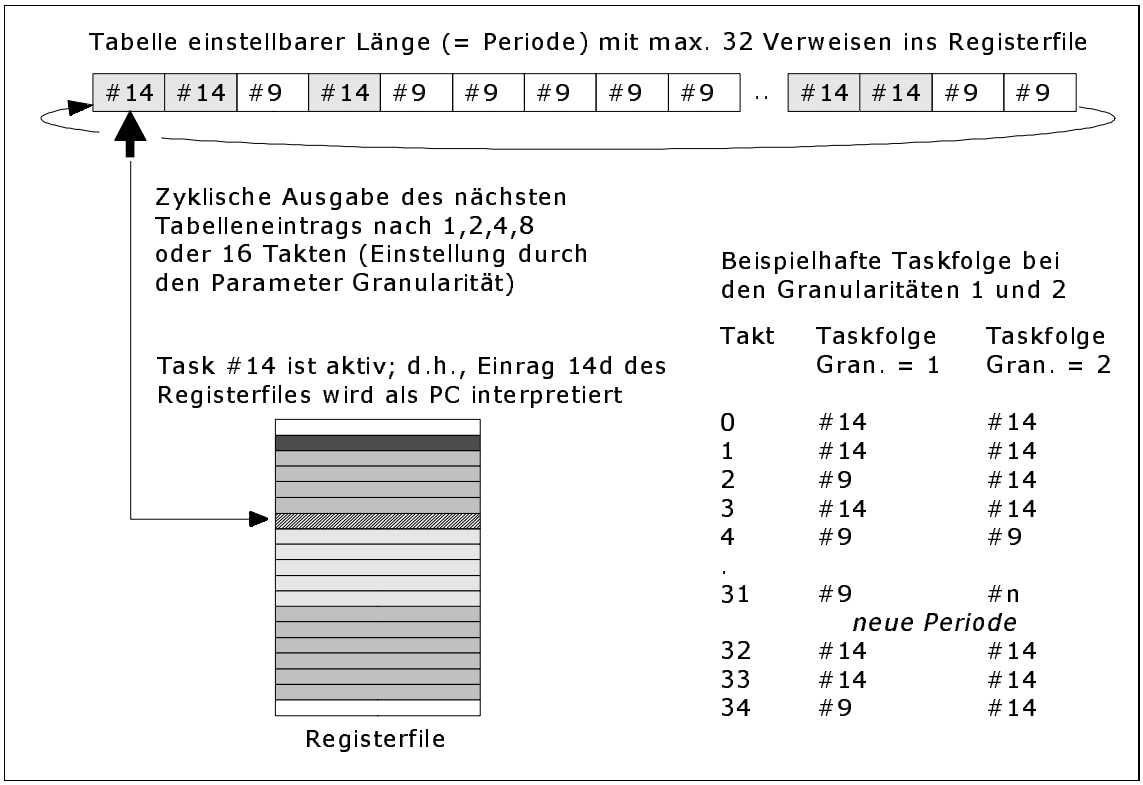


Abbildung 2.6: Funktionsweise des Schedulers

Ladebefehl die Pipeline durchlaufen hat, wird der erste Task rechtzeitig wieder aufgenommen.

Der Scheduler arbeitet statisch tabellenbasiert. Je nach eingestellter Granularität weist ein Tabelleneintrag einem Task 1, 2, 4, 8 oder 16 Takte Rechenzeit zu. Die Tabelle kann maximal 32 verschiedene Verweise auf Tasks enthalten, die nach dem 'Round Robin' Verfahren Rechenzeit zugeteilt bekommen. Die Länge der Tabelle kann zwischen 1 und 32 Einträgen Länge konfiguriert werden. Das bestimmt die Perioden-dauer der Taskfolge. Jeder Eintrag ist ein einfacher Zeiger auf eine Position im Registerfile, die als Program Status Word (PSW) des Tasks interpretiert werden soll.

In Abhängigkeit von der Einstellung des Schedulers wird der nächste Eintrag der Tabelle nach 1,2,4,8 oder 16 Takten als aktuelle Tasknummer an den Ausgang gelegt. Diese Nummer wird im Normalfall unverändert von der Interrupt Unit an das Registerfile durchgereicht. Von der durch die Tasknummer bezeichneten Registerfile-Adresse wird der PC geladen und ein Fetch/Decode/Execute-Zyklus gestartet.

Es gibt einen Supervisor Mode, der nach dem Reset des Prozessors oder nach Ausführung des Befehls OSCAL aktiv ist. Dabei ist Scheduling deaktiviert und Task #0 wird ständig ausgeführt. Nachdem der Scheduler und andere Peripherie konfiguriert

wurde, kann der normale Scheduling Modus über verschiedene Befehle aktiviertt werden.

Zu beachten ist, daß eine Neukonfiguration des Scheduler über die Befehle LTSK (Tabelleneintrag schreiben), STSK (Granularität und Länge der Tabelle setzen) und RTSK (von Tabellenposition starten) sich erst 4 Takte nach dem Fetch des Befehls auswirkt. Dies liegt in der Datenübergabe durch den normalen Datenpfad während der Execute-Phase begründet. Deswegen wird empfohlen, den statischen Scheduler nur aus dem Supervisor-Mode direkt nach dem Reset des IOP heraus zu initialisieren.

2.6 Instruction Memory

Die Harvard-Architektur des IOP bedingt je einen separaten Speicher für Instruktionen und Daten. Der Speicher ist ein asynchroner ROM mit 4096 Worten á 24 Bit Breite. Im derzeitigen Entwicklungsstand der Architektur ist keine Möglichkeit vorgesehen, den Programmspeicher nachträglich manipulieren zu können. Die Anforderungen des Marktes und die Einsatzgebiete des DSP's M3 werden zeigen, ob die einmalige Maskenprogrammierung des IOP bei der Herstellung ausreichend flexibel ist.

Es sollte kein Problem darstellen, das Design des IOP z.B. um einen EPROM zu ergänzen und den Instruktionsspeicher als von außen nicht beschreibbaren RAM zu implementieren. Der Inhalt dieses EPROM könnte bei jedem Reset des IOP automatisch in den Instruktionsspeicher übertragen werden. Hierdurch ergibt sich eine Möglichkeit für Firmware Updates. Das Verhaltens- und das Programmiermodell des IOP würden durch diese Modifikation nicht verändert werden. Für den Simulator muß auf jeden Fall eine Möglichkeit vorgesehen werden, den Speicher auf sehr einfache Weise initialisieren zu können (s. Kapitel 4.5, Seite 33).

2.7 Data Memory

Der Datenspeicher ist auch in der beschriebenen 'doppelten 16 Bit-Architektur' organisiert - in 8192 Worten mit 16 Bit Breite. Alle Adressberechnungen im IOP liefern stets ein 12 Bit-Ergebnis. Das zur Speicheradressierung fehlende 13. oder Most Significant Bit (MSB) ist im Page-Flag jedes Tasks gespeichert. Über dieses Bit wird der Speicher in 2 logische Seiten aufgeteilt.

Der Speicher liefert bei jedem Lesezugriff automatisch 2x16 Bit zurück. Beim Schreibvorgang wird anhand einer Steuerleitung entschieden, ob ein oder zwei 16 Bit-

Wort geschrieben werden sollen. Die Komponente Data Memory ist Clock synchron realisiert.

2.8 Interrupt Unit

Wie in Abbildung 2.7 schematisch dargestellt, ist die Interrupt Einheit direkt am Fetch der Instruktionen beteiligt, da sie die aktuelle Tasknummer des Schedulers durchleitet oder diese durch eine selbst generierte ersetzt.

Tritt ein Interrupt auf, leitet die Interrupt Unit statt des ursprünglichen Wertes vom Scheduler die entsprechende Tasknummer einer internen Tabelle ans Registerfile weiter. Bei dieser Funktionsweise würden Interrupts einerseits zwar ohne Verzögerung bearbeitet werden können, andererseits wäre über den zeitlichen Ablauf aller Tasks des IOP keine verbindliche Zusage mehr möglich. Es gibt jedoch eine elegante Lösung, die Realtime-Fähigkeit mit schneller Interrupt-Verarbeitung verbindet.

Wie im Kapitel 2.4 erläutert, sind nur 31 der 32 Tasks möglichen Tasks praktisch nutzbar. Somit kann Task Nummer 31 in der Scheduler-Tabelle problemlos als Platzhalter aller für Interruptbehandlung zuständigen Tasks dienen. Die benötigte Zeit für

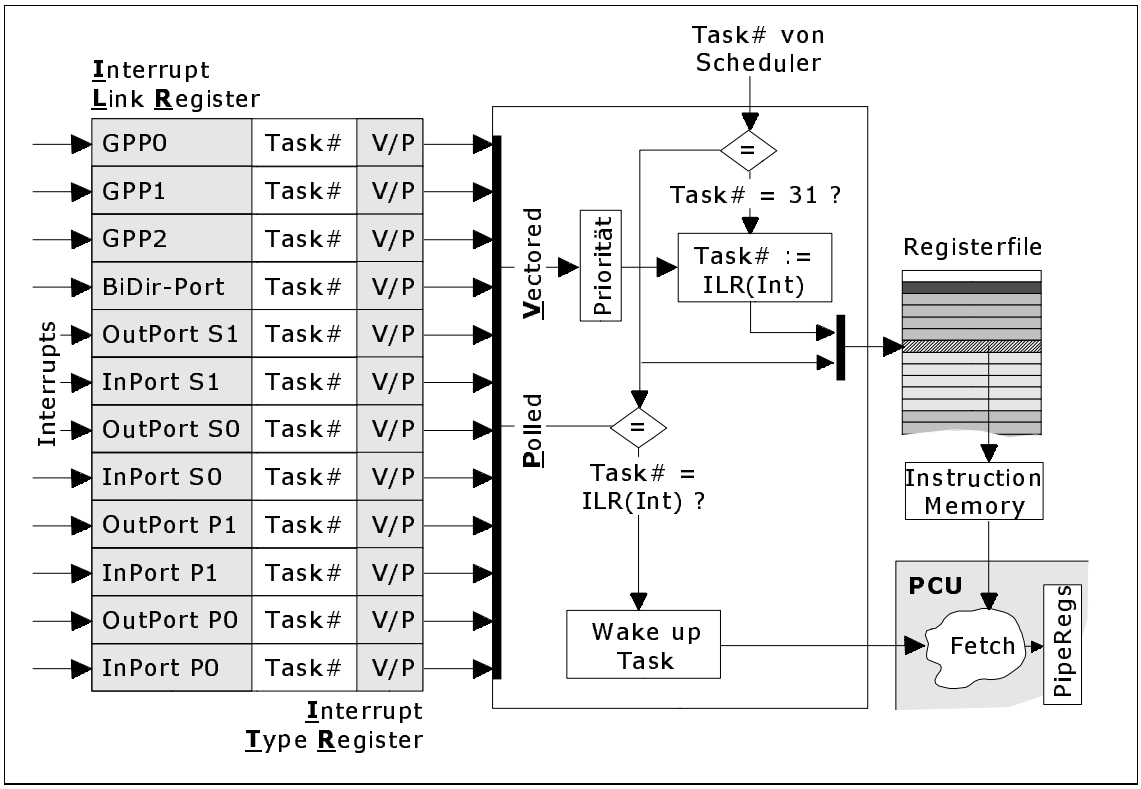


Abbildung 2.7: Funktionsweise der Interrupt Unit

deren Behandlung wird also ganz normal wie jede andere Aufgabe taktgenau über den Scheduler vorgeplant. Ist ein Interrupt aufgetreten, so gibt die Interrupt Unit nur dann die Nummer des behandelnden Tasks weiter, wenn vom Scheduler Task 31 als aktueller Task ausgegeben wird. Diese Betriebsart wird 'Vektor Based' genannt. Die zweite Betriebsart heißt 'Polled Based'. Hierbei werden verschiedene Tasks zur Interruptbehandlung der diversen Quellen eingeplant. So lange von einem Task kein Interrupt zu behandeln ist, schläft er.

Hier die komplette implementierte Logik:

Die Interrupt Einheit enthält das Interrupt Link Register (ILR), das allen Interruptquellen Tasknummern zuordnet (siehe Abbildung 2.7). Ist eine Unterbrechung ausgelöst worden, wird der Interrupt gespeichert und eine der drei folgenden Aktionen ausgelöst:

- *Im Interrupt Type Register ist die Quelle als Vectored gekennzeichnet.*
  Solange der Interrupt nicht durch die Instruktion IRET gelöscht wurde, wird die vom Scheduler eingeplante Tasknummer 31 durch die dem ILR entnommene Tasknummer ersetzt. Sollte während dessen eine weitere Quelle einen Vectored Interrupt auslösen, entscheidet eine konfigurierbare Prioritätenlogik, wer Vorrang hat. Es geht keine Anforderung verloren, da Interrupts in eine Warteschlange gestellt werden.
- *Im Interrupt Type Register ist die Quelle als 'Polled' gekennzeichnet.*
  Immer wenn der Task im Scheduler eingeplant ist, der im ILR dem Interrupt zugeordnet wurde, gibt die Interrupt-Einheit den Befehl an die PCU, daß der aktuelle Task aufgeweckt werden soll.

  In den Schlafzustand werden Tasks versetzt, die nur für die Behandlung von Polled Interrupts zuständig sind.
- *Es ist keine Interruptquelle als aktiv markiert, oder ein Vectored Interrupt wartet auf seine Behandlung, während momentan jedoch kein Task #31 eingeplant ist, oder die dem Polled Interrupt zugeordnete Tasknummer ist augenblicklich nicht eingeplant.*
  Dann wird die vom Scheduler stammende Tasknummer normal durch die Interrupt-Unit weitergereicht.

2.9 Multiplexer und Demultiplexer

Auf eine Busstruktur zum Austausch von Daten zwischen den Funktionseinheiten,

wie sie üblicherweise bei komplexen, in Funktionseinheiten untergliederten elektronischen Schaltungen verwendet werden, wurde beim Design des IOP zugunsten einer Multiplexer-Struktur verzichtet. Die Originaldokumentation verwendet aus Gründen der Übersichtlichkeit eine Bus-Symbolik. Der Autor des IOP-Designs weist jedoch ausdrücklich darauf hin, daß sich an beiden Operandeneingängen der ALU Multiplexer befinden, die durch die PCU gesteuert, nur das Daten der jeweils ausgewählten Einheit weitergeben. Am Ergebnis-Ausgang der ALU ist ein Demultiplexer plaziert, der die Daten wieder auf die verschiedenen Einheiten verteilen kann.

Da die Ausgangstreiber der einzelnen Komponenten nicht mehr gegen alle Eingänge, die an der Busleitung hängen, treiben müssen, und somit kleiner ausgeführt werden können, verringert sich der Stromverbrauch der Schaltung merklich. Auch die Schaltgeschwindigkeit kann sich entscheidend erhöhen, wenn geringere Kapazitäten, die durch den Wegfall langer und hochkapazitiver Busleitungen erreicht werden, in kürzerer Zeit umgeladen werden können.

2.10 Recheneinheit (ALU)

Die Recheneinheit ist integraler Bestandteil des IOP-Datenpfades. Jeder Daten-Transport, jede Berechnung, Sprünge, auch die Daten zur Konfiguration von Komponenten nehmen den Weg durch die Recheneinheit.

Die Hardwareimplementierung der Recheneinheit sieht eine zweistufige Pipeline vor. Diese Entwurfsentscheidung ist der größeren Verzögerung geschuldet, die durch das Bereitstellen der Daten von den verschiedenen Einheiten und von den in Abbildung 2.8 zu sehenden rotierenden Shiftern variabler Schrittweite am Eingang von Operand X und Y erzeugt wird. Wäre es theoretisch auch möglich, die Pipline durch Optimierung von Laufzeiten einzusparen, so spricht doch dafür, daß eine Implementierung der Pipeline von Anfang an später größeren Freiraum zur Erhöhung des maximalen Arbeitstaktes der ALU läßt, ohne den Entwurf nachträglich doch noch um eine Pipeline ergänzen zu müssen.

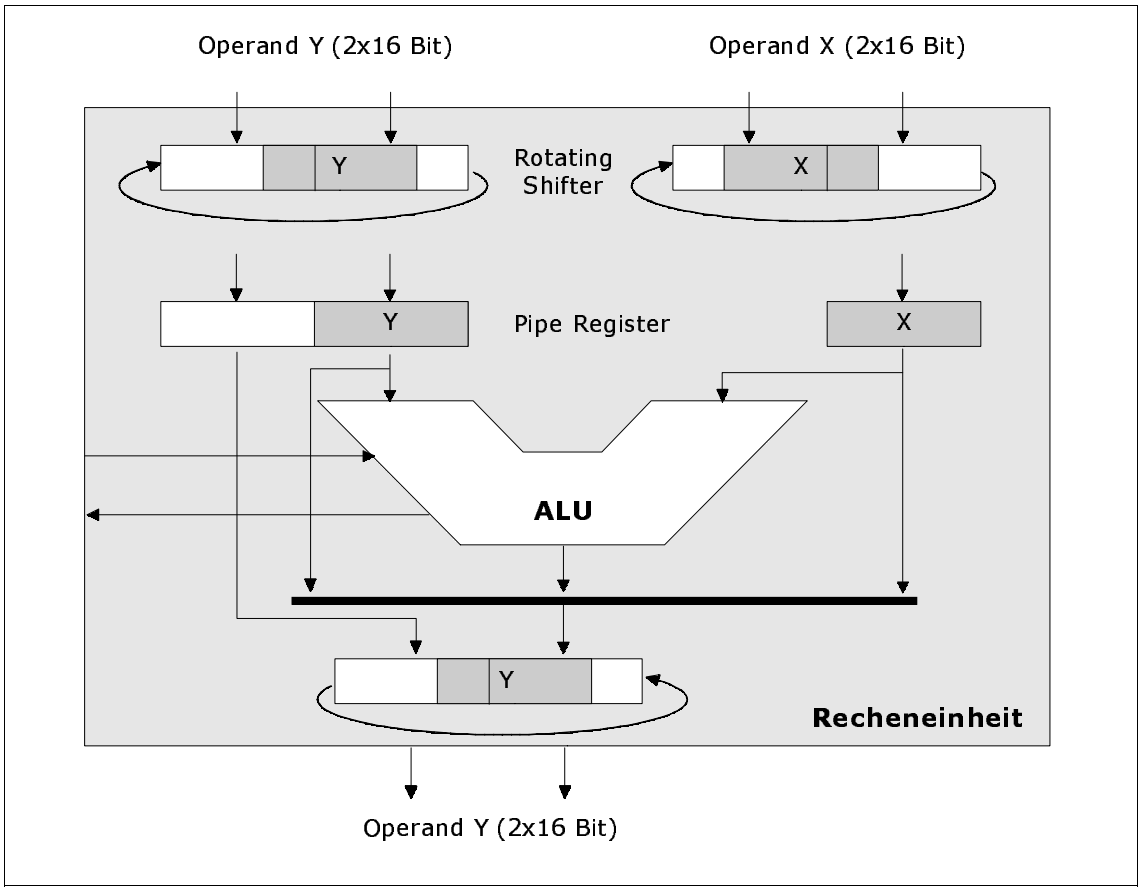


Abbildung 2.8: Die ALU im Detail

Die Recheneinheit bekommt in der Execute-Phase die auszuführende Operation von der PCU mitgeteilt.

2.11 DSP-Buffer und AGU

Der IOP ist keine General Purpose CPU. Er wurde entwickelt, um in Zusammenarbeit mit einem spezialisierten DSP-Kern den Anteil der Datenstromoperationen zu übernehmen. Der DSP-Kern selbst kann nur Berechnungen mit Daten innerhalb seines lokalen Speichers ausführen. Um Daten vom IOP zum DSP-Hauptspeicher und wieder retour transferieren zu können, benötigt man den DSP-Buffer. Dieser ist als Entkopplungselement zwischen die beiden Einheiten geschaltet.

Es gibt je einen Lese- und einen Schreibpuffer. Beide bestehen aus je 16 Slices á 16 Bit Breite. Aus Gründen der Effektivität werden immer nur komplett gefüllte Buffer von 256 Bit Breite (16 x 16 Bit breite Worte) zwischen DSP-Memory und IOP übertragen. Die Slices des DSP-Buffers werden vom Programmierer prinzipiell wie ein Registerfile

mit 16 Einträgen behandelt und folglich direkt adressiert und mit Daten gefüllt oder ausgelesen.

Es gibt Befehle, mit dem der DSP instruiert werden kann, den Buffer innerhalb einer bestimmten Deadline von maximal 32 Takten entweder mit Werten aus dem DSP-Speicher zu füllen oder Werte aus dem Puffer in den DSP-Speicher zu übertragen.

Die Adress Generation Unit (AGU) des DSP versucht dann, innerhalb der Deadline einen Zyklus zu finden, in dem der DSP nicht selbst auf seinen Speicher zugreifen muß. Erhält der IOP vor dem Erreichen der Deadline nicht rechtzeitig die Bestätigung der Übertragung vom oder zum DSP-Speicher, so erzwingt er einen Takt vor deren Ablauf die Übertragung. Der DSP wird für diesen Zyklus von seinem Speicher getrennt.

Weiterhin existiert ein Befehl, mit dem der Inhalt des Lesepuffers in den Schreibpuffer kopiert werden kann. Somit können Werte im DSP-Speicher manipuliert werden, ohne erst die kompletten 256 Bit (16x16 Bit) in den lokalen Speicher des IOP laden zu müssen. Man ändert dazu im Schreibpuffer nur den gewünschten 16 Bit-Wert (Slice) und und läßt dann den gesamten Buffer wieder in den Speicher des DSP zurück übertragen.

3 Entwurf des Modells

3.1 Allgemeines

Die grundsätzliche Aufgabe der Belegarbeit besteht darin, ein Simulationsmodell des vorgestellten IOP zu entwickeln und zu implementieren. Zusammen mit dem Entwickler des IOP wurde entschieden, die Hardwarebeschreibungssprache VHDL (zur Einführung in VHDL s. Kapitel 3.2) als Modellierungssprache einzusetzen (Zum gleichen Prozessor läuft derzeit eine Diplomarbeit [1], in deren Verlauf ein retargierbarer Simulator in der Programmiersprache 'C' konstruiert werden wird. Der Simulator wird unter anderem auch den Befehlssatz des IOP nachbilden können).

Welche Anforderungen werden an ein Modell im Sinne der Aufgabe gestellt, auf welchen Grundlagen kann aufgebaut werden?

Zur Verfügung standen verschiedene Versionen der I/O-Prozessor Architekturdokumentation des Autors [ENG98/99], die einem kontinuierlichen Entwicklungssprozeß unterliegt. Bei der letzten bis zu diesem Zeitpunkt vorliegenden Version 'Manual Version 1.5' vom 14. Juli 1999 handelt es sich leider immer noch um ein Programmierhandbuch mit detaillierter Beschreibung des Befehlssatzes und einer kurzen Einführung in die innere Struktur des Prozessors.

Es war ein VHDL Modell gefordert, daß Maschinenprogramme, die für den realen IOP assembliert wurden, laden und auf realen Daten basierend ausführen kann. Dieses sogenannte Softwaremodell soll die Maschinenprogramme derart ausführen/ interpretieren/ verarbeiten und Ausgaben auf definierten Schnittstellen liefern, daß sich Softwaremodell und reale Hardware bei gleichen Eingangsdaten identisch verhalten. Da für den praktischen Nutzen des Simulators die volle Übereinstimmung in ganzer Härte nicht notwendig und für eine möglichst hohe Simulationsgeschwindigkeit sogar negativ ist, wurde festgelegt, daß die Nachbildung nur in für den Programmablauf wichtigen Teilen

gegeben sein muß. Die Übereinstimmung ist auf dem Niveau der Schnittstellen zu Register, Datenspeicher, Program Counter, Scheduler, Bussysteme, der sogenannten Register-Transfer-Ebene,. zu erzielen. Unterschiede werden im Zeitverhalten liegen. Wegen des hohen Aufwands von Softwaremodellen ist nach bisherigen Probeläufen davon auszugehen, daß in 1 Sekunde Laufzeit des Simulators nur rund 10^3 Befehlszyklen Laufzeit des realen Prozessors simuliert werden können (siehe Kapitel 5: Ergebnisse und Ausblick).

Das Simulationsmodell soll möglichst innerhalb eines Prozessor-Entwurfssystems nutzbar sein. Der Einsatz innerhalb solch eines Entwurfssystems impliziert vor allem folgende Forderungen:

- Die Prozessoren werden an ihre jeweiligen Einsatzfelder und -Zwecke angepaßt.
  Dies wird mit hoher Wahrscheinlichkeit durch Hinein- und Herausnehmen von Einheiten oder Instruktionen geschehen. Um solche modifizierten Prozessoren simulieren zu können, muß die Struktur des Simulators möglichst modular aufgebaut sein, um schnell und mit geringem Aufwand Anpassungen vornehmen zu können. Es empfiehlt sich natürlich besonders, die durch den inneren Aufbau des realen Prozessors vorgegebene Struktur als Grundlage zu verwenden. Die Durchsetzung der auch im Originalentwurf sehr strikten Trennung in die in Kapitel 2 beschriebenen Funktionseinheiten vergrößert die Chancen, den Simulator an verschiedene Veränderungen in der Hardware erfolgreich anpassen zu können.
- Die Architektur ist skalierbar.
  Das bedeutet, Größen von Datentypen werden in Zukunft geändert, Speicher vergrößert, Kapazitäten anderer Einheiten angepaßt - der Instruktionssatz wird modifiziert, etc. Solche Anpassungen sind natürlich meist nicht voraussehbar. Sollten doch Erkenntnisse über Details solcher zukünftiger Skalierungen vorliegen, so ist es aufwendig, in der Programmierung des Modells 'Schalter' vorzusehen, die zwischen den möglichen Skalierungsstufen umschalten können. Weiterhin setzt der zusätzlich nötige Aufwand natürlich auch die mögliche Simulationsgeschwindigkeit herab. Der für den Autor offensichtlich gangbare Weg, um Skalierung der Architektur unterstützen zu können, sieht wie folgt aus:
  Hinter allen Größen, die die Hardware charakterisieren, verbergen sich einige wenige Grundlegende. Eine dieser Größen ist z.B. die Verarbeitungsbreite des Prozessors in Bit. Das Registerfile hat diese Breite, alle Datenschnittstellen haben wegen der neuartigen Bitadressierung doppelte Verarbeitungsbreite. Alle Datentypen, die mit Datenverarbeitung in Zusammenhang stehen, sind ganze Vielfache der Verarbeitungsbreite. Bei einer genaueren Analyse zeigt sich schnell, daß sich die verschiedenen Datentypen im Prozessormodell auf ein paar grundlegende

Größen wie Verarbeitungsbreite, Breite des PC, Anzahl Einträge des Registerfiles, Größe des Task Window oder Breite des Opcodes zurückführen lassen, oder ihr Typ und ihre Größe sich daraus ableitet.
Wird das bei der Modellierung berücksichtigt und wird nun z.B. die Verarbeitungsbreite von 16 auf 20 Bit geändert, so braucht das Modell im Idealfall nur einmal neu kompiliert zu werden.
Der Simulator sollte auf solch einem wohldefinierten System von aufeinander aufbauenden Größen und Datentypen basieren.

- Alle aufgezeigten möglichen Modifikationen sollten theoretisch nur an einer zentralen Stelle erforderlich und leicht erschließbar sein.
  Durch sprachliche Mittel von VHDL, die weiter hinten beschrieben werden, sollte diese Vorgehensweise konsequent durchgesetzt werden.

Ein Simulationsmodell, das diese Anforderungen optimal unterstützen soll, müßte eine dem Hardwareentwurf sehr ähnliche innere Struktur aufweisen. Im Idealfall würde ein automatisches Konfigurationsprogramm bei Änderungen des Originalprozssors alle nötigen Anpassungen am VHDL Quelltext des Simulatormodells automatisch vornehmen können. Dies setzt eine feste, unveränderliche Struktur des Quelltextes voraus. Damit stellt die Frage, warum nicht ein gemeinsamer VHDL-Quelltext als Grundlage sowohl für Simulation als auch zur Synthese dienen kann. Jede Änderung am Design wäre sofort simulierbar. Wurde ein neuer Entwurf erfolgreich simuliert, besitzt man automatisch auch das fertige, synthetisierbare Design. Gegen solch eine Vorgehensweise sprechen mehrere Argumente:

- Erstens ist ein Hardware-Entwurf fast immer an Restriktionen gebunden, die von verschiedenen Seiten her stammen. Das verwendete Synthesetool unterstützt oft nur einen Teil des vollständigen VHDL Sprachumfangs.
- Zweitens bringen verwendete Zieltechnologie-Bibliotheken die verschiedensten Einschränkungen in den Entwurf. Physikalische und herstellungsbedingte Effekte müssen aufwendig und trickreich im Design berücksichtigt werden. Ein Entwurf für die Synthese ist oft weit komplexer als die äquivalente Verhaltensbeschreibung, denn die Forschung steht auf dem Gebiet der automatischen Hardware-Synthese noch ziemlich am Anfang.
- Ein solch komplexes Design ist gut zur Synthese geeignet, da eine Synthetisierung im Normalfall nur wenige Male durchgeführt wird. Die beschriebenen Einschränkungen, Spezialfälle und trickreichen Umwege führen jedoch zu einem im Optimalfall nur mit sehr mäßiger Performance zu simulierendem Design. Um Tests durchzuführen und Fragestellungen, wie: 'Wird ein Algorithmus auf meinem

Design schneller, wenn ein Befehl verändert, neu hinzugefügt wird?' 'Welche Auswirkungen können Änderungen an Pipelinetiefe, Befehlsphasen etc. in der Praxis zeigen?' 'Ist eine Verbreiterung des Datenformats um n Bits ohne weitere Auswirkungen oder muß das gesamte Design neu entworfen werden?' sind anhand der synthetisierbaren Quellen entweder gar nicht oder nur mit immensen zeitlichen Aufwand zu beantworten.

Gemäß den Vorbetrachtungen wurde ein Simulations-Modell aufgestellt, daß in seiner äußeren und inneren Struktur das Original-Design so weit wie möglich, und dabei jedoch so einfach als möglich, nachbildet. Das Modell ist strikt in die im Kapitel 2 erläuterten Komponenten modularisiert.

3.2 Was ist VHDL

Die **V**ery High Speed Integrated Circuits **H**ardware **D**escription **L**anguage oder 'Sprache zur Beschreibung der Struktur und des Verhaltens von hochintegrierten Schaltkreisen' ist eine noch relativ junge Sprache.

Im Jahr 1980 gründete die US-Regierung das Projekt 'Very High Speed Integrated Circuit' (VHSIC), um den Designprozess von elektronischen Schaltungen zu vereinfachen, zu verbessern und zu vereinheitlichen. Infolge dessen entstand VHDL. Im Jahr 1986 der IEEE vorgestellt, wurde VHDL 1987 unter der IEEE-Nr. 1076-1987 zum internationalen Standard erhoben. Inzwischen gibt es auch einen ANSI-Standard für VHDL.

Hauptgrund der Entwicklung und Standardisierung war jedoch nicht die gute Hoffnung der Regierung, durch einen allgemein anerkannten Standard den Markt zu beflügeln und die Weltwirtschaft anzukurbeln. In Wirklichkeit hatte es das US-Verteidigungsministerium (DoD) als weltweit größter Abnehmer von Hochtechnologie für seine Lieferanten von Hardware zur Bedingung gemacht, daß jeder Chip, jedes System in seinem Aufbau und seinem Verhalten in VHDL zu beschreiben ist. Die Sprache wurde aus dem hauptsächlich im militärischen Bereich eingesetzten ADA entwickelt. So sollten die enormen Kosten für Wartung und Nachbesserung von Militärischen Systemen, die ungefähr die Hälfte der Gesamtkosten betrugen, reduziert werden. Besonders wichtig war dem DoD dabei die Möglichkeit zur klaren und sauberen Beschreibung von komplexen Schaltungen. Außerdem sollten sich Modelle leicht zwischen verschiedenen Entwicklungsgruppen austauschen lassen.[CHA97], [LEH94]

VHDL war zu Beginn seiner Kariere eher eine Art Hilfsmittel für Hardwareexperten um Systeme beschreiben zu können, als eine Computersprache. Später dann erweiterte

sich das Spektrum seines Einsatzes hin zu Erstellung und Simulation von Softwareprototypen von Schaltkreisen. Erst seit weniger als zehn Jahren entwickelte sich VHDL, vor allem durch die Firma Synopsys forciert, zu einer Sprache, aus deren 'Programmen' direkt funktionierende Hardware synthetisiert werden kann. Entsprechend dem noch recht weit am Anfang befindlichen Stand der Forschung zur automatischen Generierung von Schaltungen und dem ursprünglich nicht für diesen Zweck entwickelten Sprachumfang von VHDL ist nicht jedes syntaktisch korrekte VHDL-Programm auch automatisch in Hardware synthetisierbar.

3.3 Vorteile von VHDL

Weshalb eine weitere Programmiersprache? Ist nicht jede x-beliebige andere Sprache, z.B. 'C', genauso gut geeignet?

Diese Frage läßt sich nicht eindeutig beantworten. Man sollte immer für den jeweiligen Einsatzzweck Vor- und Nachteile abwägen. Deswegen seien hier ein paar der offensichtlichen Vorteile von VHDL gegenüber anderen Programmiersprachen aufgezählt: [CHA97]

- VHDL ist dem Grundkonzept nach eine Sprache, die parallele, gleichzeitig ablaufende Vorgänge beschreibt. Dieses Konzept wird von einem Simulationssystem auf Basis von Zeitstempeln auf jedem beliebigen Rechnersystem, auf dem eine VHDL Umgebungen existiert, vollkommen identisch durchgesetzt. VHDL ist deswegen portabel.
- VHDL enthält Sprachelemente, die das Modellieren von elektronischen Systemen stark vereinfachen. Beispielsweise gibt es ein sprachliche Mittel, die zur Nachbildung von elektrischen Signalleitungen hervorragend geeignet sind.
- VHDL ist eine sehr leicht lesbare und Programmierfehlern entgegenwirkende Sprache. Alle Daten sind streng getypt und lassen sich nur bei Kompatibilität und nur auf ausdrückliche Anweisung von einem Typ in einen anderen Umwandeln. Bereiche von Datentypen werden auch zur Laufzeit ständig überprüft.
- VHDL wurde entwickelt, um elektrische Systeme vom einzelnen Transistor bis hinauf zu komplexesten Netzwerken auf den verschiedensten Beschreibungsebenen modellieren zu können. Durch strikte Hierarchiebildung, Datenkapselung und besondere sprachliche Mittel ist VHDL die erste Wahl für umfangreichste Projekte rund um Hardware. Warum das Rad neu erfinden?

- VHDL ist sehr viel stärker als andere Sprachen selbstdokumentierend, da es eigentlich 'nur' zur beschreibenden Dokumentation komplexer Systeme entwickelt wurde.
- VHDL ist standardisiert.

Als Grundlage zur Einführung in die Sprache und ihre Konzepte diente folgende Literatur: [LEH94], [CHA97], [ASH90], [COH95].

3.4 Vorteile des Typs Std_Logic

Der Standardtyp, auf dem alle Variable und Signale des gesamten Modells basieren, ist die mehrwertige Logik 'Std_Logic'. Folgende Punkte haben zur Bevorzugung gegenüber dem Typ 'Bit' beigetragen:

- Std_Logic ist eine mehrwertige Logik, deren Alphabet mehr Information als nur 1 Bit darstellen kann. Die Buchstaben sind: 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'
  Mit dem Buchstaben 'X' für 'undefinierter Zustand' können Fehler, wie z.B. bei Reset nicht initialisierte Signale etc., im Modell leicht erkannt werden. Der Buchstabe 'Z' steht für einen definierten hochohmigen Zustand der Leitung, der aber keinen gültigen Logikpegel darstellt. Durch Verwendung von mit 'Z..Z' initialisierten Bussen, Leitungen etc. kann neben gültigen Wertebereichen auch noch ein Fehlerzustand oder Inaktivität kodiert werden.
- Std_Logic ist ein Resolved Typ, das heißt, ein Signal dieses Typs kann von mehreren Treibern gleichzeitig angesteuert werden. Auf diese Weise lassen sich Busstrukturen einfach realisieren. An Stellen, an denen solche Mehrfachtreiber nicht eingesetzt werden müssen, kann man den gleichwertigen unresolved-Typ 'Std_ULogic' verwenden.
- Der dritte und entscheidende Vorteil ist die Verfügbarkeit von Standardbibliotheken, die Konvertierungs- und arithmetische Routinen oder auch Debugging-Unterstützung auf Basis von Std_Logic anbieten.

3.5 Konstanten statt Generics

Der Simulator und seine Bestandteile, also alle verwendeten Komponenten sollen mit den Mitteln von VHDL möglichst leicht an eine Skalierung angepaßt werden können. Dafür wie geschaffen erscheint der **`GENERIC`**s-Mechanismus, den VHDL im Schnittstellenteil von Komponenten, den Entities, anbietet. Im Abschnitt **`GENERIC`** sind

```
-- Beispiel für Konfiguration
-- von Komponenten
-- durch Konstanten

-- Bibliothek mit allen projektweit
-- genutzten Konstanten

package IOP_Constants is
         constant Max: Integer := 1024;
         ...
end Constants;

-- Entität Speicher hat Max Einträge

use IOP_Constants.all;
entity IOP_Data_Memory is
port(    CLK: IN Std_Logic;
         ADR: IN Integer Range 1 to Max;
         );
end IOP_Data_Memory;
```

Die Größen von Speichern, Datenbussen etc. können mit Hilfe von Konstanten zentral verwaltet werden. Werden die Konstanten konsequent genutzt und nur an zentraler Stelle in einer Bibliothek (Package) definiert, so ist es leicht, das Modell an Veränderungen durch Modifikation oder Tausch der Bibliothek anzupassen.

```
-- Beispiel für Konfiguration
-- von Komponenten
-- im GENERIC Teil der Entity

-- Entität Speicher hat Max Einträge

entity IOP_Data_Memory is
GENERIC( Max: Integer := 1024);
port(    CLK: IN Std_Logic;
         ADR: IN Integer Range 1 to Max;
         );
end IOP_Data_Memory;
```

Die Konfiguration über den Generic-Teil in der Schnittstelle einer Komponente hat den Vorteil, daß sie einerseits anpaßbar ist, andererseits aber nicht die Verwendung einer bestimmten Bibliothek Bedingung ist.
Diese Vorgehensweise erleichtert die Entwicklung, Weitergabe und Wiederbenutzung von Komponenten in anderen Projekten.

Abbildung 3.1: Konstanten vs. Generic-Mechanismus

spezielle Konstanten deklariert, über die eine Komponente konfiguriert werden kann, ohne das der Quelltext vorliegen muß. Diese Möglichkeit ist nicht automatisch eingebaut; der Programmierer einer Komponente muß die Skalierbarkeit bestimmter Parameter seiner Komponente explizit vorgesehen haben. Er nutzt die im **GENERIC**-Teil deklarierten Konstanten, als ob sie in der Komponente durch ihn selbst definiert oder aus einer Bibliothek importiert wären.

Im Gegensatz zu normalen Laufzeitparametern, wie sie in allen anderen Programmiersprachen auch bekannt sind, wird ein **GENERIC** jedoch nur einmal zur Initialisierungszeit des Programms übergeben. Im Vergleich zur Methode, eine Bibliothek mit Konstanten zu erstellen, die von jeder Komponente importiert wird, haben **GENERIC**s den Vorteil, daß bei Änderung ihres Wertes nicht die gesamte Architektur neu kompi-

liert werden muß. Außerdem ist eine Komponente dadurch vollständig gekappselt und anhand ihrer Schnittstelle, der Entity, ist sofort ersichtlich, welche Eigenschaften sich parametrisieren 'lassen'. Bei Weitergabe, Verkauf etc. muß keine Bibliothek im Quelltext mitgegeben werden. Beispielsweise kann man eine Speicher-Komponente schreiben, die sich vom Kunden ohne Besitz des Quelltextes in Datenbreite und Speicherkapazität anpassen läßt (s. Abbildung 3.1).

Jedoch zieht die Verwendung von **`GENERICS`** einen um einiges höheren Programmier- und Wartungsaufwand nach sich. Aus diesem Grund und weil alle Komponenten des Simulators nur in seinem Kontext verwendet werden sollen, ist seine Konfiguration und Parametrisierung komplett in eine einzige Bibliothek, das Package IOP_CONSTANTS ausgelagert. Die Quellen sind derart implementiert, daß bei Anpassungen meist nur die entsprechenden Typen, Konstanten und Aliase in IOP_CONSTANTS angepaßt werden müssen. In einigen wenigen Fällen sind Anpassungen direkt im Code nötig. Das ist z.B. Der Fall, wenn sich die Parameter einer Instruktion ändern. Hierbei muß z.B. in der Komponente IOP_PCU_FSM die Case-Schleife der Decode-Phase angepaßt werden.

4 Realisierung des Simulators

Die Implementierung entstand innerhalb der integrierten Entwicklungsumgebung Active VHDL [ALD99] unter Windows 98. Die sehr gute Einführung in die Sprache VHDL, die enthaltene hervorragende Dokumentation und das sehr intuitive Konzept dieses Tools trugen maßgeblich zum erfolgreichen Gelingen der Arbeit bei.

Da VHDL standardisiert ist, wurden keine größeren Probleme bei der Migration zu Synopsys unter SUN erwartet. Um so überraschender war es, als sich zeigte, daß die aktuell benutzte Synopsys Version den Sprachstandard VHDL '93 nur partiell unterstützt. Mehrere kleinere Anpassungen ermöglichten die Portierung. Ein größeres aus der nur teilweisen Unterstützung des Sprachstandards herrührendes Problem ist im Kapitel 4.2 Dump-File beschrieben.

Das gesamte VHDL Modell wurde in 3 Bibliotheken und 11 Komponenten partitioniert. Im Folgenden werden die einzelnen Komponenten, soweit Details von allgemeinem Interesse sind, beschrieben, und Probleme, die während der Implementierung auftraten, werden erläutert.

4.1 Bibliotheken

Alle Definition von Konstanten und Datentypen befinden sich an zentraler Stelle im VHDL-Package IOP_CONSTANTS. Wie in Kapitel 3.1 erläutert, sind alle Größen auf Abhängigkeiten untereinander analysiert worden.

Nur wenn Konstanten in Hinsicht auf Skalierung ausschließlich manuell anpaßbar sind, wurden sie in den jeweiligen Komponenten deklariert. Dies tritt beispielsweise bei der Dekodierung von nicht dem Standardformat entsprechenden Instruktionen, z.B. dem

```
...
constant Flags_Anzahl:  Positive := 4;
subtype  Flags_Typ is   Std_Logic_Vector(Flags_Anzahl - 1 downto 0);

-- die ersten n Eintraege des Registerfiles
-- je Task sind fuer Spezialregister, wie
-- das PSW (PC und Flags) etc. reserviert.
-- Im Augenblick existiert nur das PSW
constant Prozessorstatusregister_Anzahl: Positive := 1;

-- Wortlaenge des Prozesscounters
constant PC_Breite: Positive := 12;
subtype  PC_Typ is STD_LOGIC_VECTOR(PC_Breite - 1 downto 0);

-- Wortlaenge des Opcodes einer Instruktion
constant OPC_Breite: Positive := 8;
subtype  OPC_Typ is Std_Logic_Vector(OPC_Breite - 1 downto 0);
...
```

Abbildung 4.1: Auszug aus dem Package IOP_Constants

Befehl Load Interrupt Priority Register (LIPR), auf. Wenn also in den nachfolgenden Kapiteln konkrete Zahlen genannt sind, so sind diese IOP_CONSTANTS entnommen.

Oft benötigte Funktionen wurden in das Package IOP_FUNCTIONS ausgelagert. Das sind Funktionen zum Protokollieren der Simulatorereignisse im Dump-File, Konvertierungsfunktionen, etc.

Weitere Details sind dem ausführlich kommentierten VHDL-Quelltext zu entnehmen.

4.2 Dump-File

Alle Vorgänge im Simulator sind über Waveforms, wie in Abbildung 4.2 dargestellt, auswertbar. Diese Methode eignet sich aber weniger zur Verfolgung einer Simulation auf Register-Transfer Ebene. Deshalb werden alle wichtigen Vorgänge im Dump-File protokolliert. Die Komponenten Scheduler, PCU, Registerfile und Data Memory rufen in IOP_FUNCTIONS implementierte Protokoll-Funktionen bei jeder Änderung und teilweise auch am Anfang jedes Befehlszyklusses auf.

Anhang A: Ein Beispielprogramm zeigt einen kommentierten Ausschnitt eines Simulator-Dumps. Auf dem IOP liefen dabei 2 Tasks.

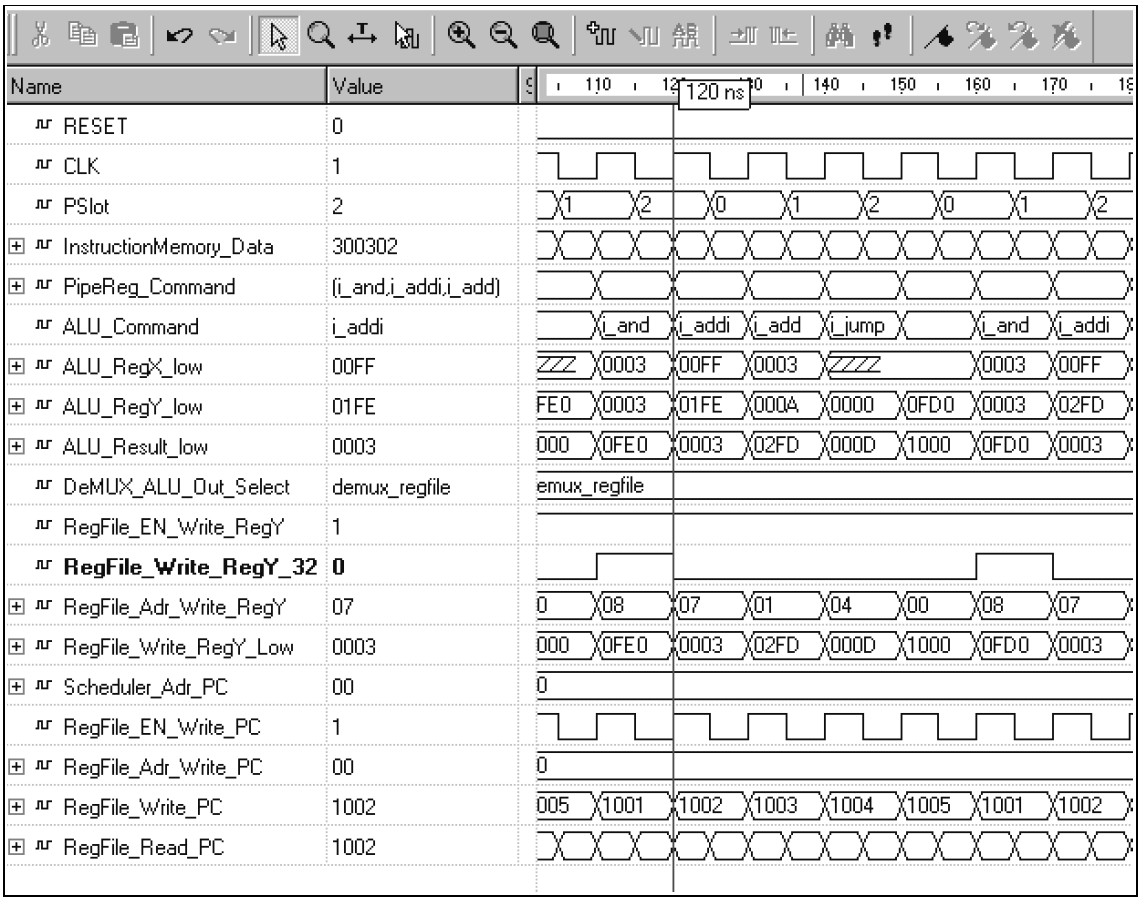


Abbildung 4.2: Waveform eines Simulatorlaufs

4.3 DSP-Environment

Der hier implementierte Simulator bildet einen Prozessor nach, der nur in der beschriebenen M3-Umgebung Sinn macht. Alle Teileinheiten des IOP sind gleichwertig mit anderen Komponenten des DSP auf dem Die untergebracht. Um eine sinnvolle Simulation durchführen zu können, muß also die Umgebung des IOP nachgebildet werden. Dazu gehört die Takterzeugung sowie die Eingangssignal-Simulation, die den IOP mit allen nötigen Stimuli versorgt. Hier sind Signale, wie Clock, Reset, Verbindungen zwischen den Komponenten etc. definiert. Alle in Kapitel 2 beschrieben Komponenten werden im DSP-Environment instanziiert und mit ihren Ein- und Ausgangssignalen miteinander verbunden.

4.4 Registerfile

Das Registerfile ist als taktsynchroner Speicher mit mehreren Lese- und Schreibports, den zugehörigen Adreßbussen und Steuersignalen realisiert. Taktsynchron bedeutet, daß Werte nur bei steigender Clock-Flanke in die internen Register übernommen werden. Während ein Adreßbus eine gültige Adresse trägt, wird der Inhalt des ausgewählten Registers ständig ausgegeben. Ist die Adresse ungültig, z.B. weil alle Leitungen hochohmig sind, wird der entsprechende Ausgang auch hochohmig (auf 'Z') geschaltet.

Während eines Taktes müssen gleichzeitig maximal 6 verschiedene 16 Bit-Einträge (PSW, Index-Register, Register X, Register X+1 und Register Y, Register Y+1) und ein 4 Bit-Eintrag (Flags) gelesen und maximal 3 verschiedene Einträge (1 x PSW, und 1 x Ergebnis der ALU-Operationen mit 2x 16 Bit Breite) gleichzeitig geschrieben werden können.

Wegen der beschriebenen Besonderheiten, die sich durch die verwendete Bitadressierung des IOP ergeben, werden mit den ausgewählten Registern X und Y immer auch ihre direkten Nachfolger X+1 und Y+1 gelesen. Beim Zurückschreiben des ALU-Ergebnisses erfolgt dies nur explizit auf Anforderung über eine Steuerleitung.

4.5 Instruction Memory

Ein Array mit 2^{12} (4096) Einträgen á 24 Bit Breite wird asynchron ausgelesen und bildet die gesamte verfügbare Funktionalität des IOP-Programmspeichers nach.

Die im Kapitel 2.6 erläuterte Forderung nach Ladbarkeit beliebiger Maschinenprogramme macht die Komponente aber schnell zu einem der komplexeren Teile des Simulators.

Es ist nicht offensichtlich, daß nicht jeder beliebig große Speicher durch einen einfachen Array in einem VHDL Modell modelliert werden kann. In [COH95], Seite 8, sind Abschätzungen gemacht worden, wieviel Speicher für Signale bestimmter Datentypen in einem VHDL-Modell zur Laufzeit benötigt wird:

Datentyp	1 Signal	4k Array	64k Array	1M Array
Std_Logic	*~ 100+1 Byte*	*~ 400 kByte*	*~ 6 MByte*	*~ 105 MByte*
Integer	*~ 100+4 Byte*	*~ 400 kByte*	*~ 6 MByte*	*~ 105 MByte*
Std_Logic_Vector (32 Einträge)	*~ (100+1 Byte) x 32*	*~ 12 MByte*	*~ 200 MByte*	*~ 3 GByte*

Tabelle 4.1: Speicherbedarf des Simulators nach [COH95] für verschiedene Datentypen

Man sieht, daß bei einem Modell des schon sehr betagten Prozessors 'Z80' nur für dessen 64 kByte Speicher insgesamt über 200 MByte Hauptspeicher vom Simulator benötigt würden. Komponenten solcher Größenordnung erfordern folglich intelligentere Vorgehensweisen bei der Modellierung. Die 4 kByte x 24 Bit des IOP sind jedoch ohne zusätzlichen Aufwand durch Arrays modellierbar.

Aus einer Datei werden zur Initialisierung des Modells Maschinenbefehle geladen. Es mußte ein verbindliches Format für die Kommandodatei festgelegt werden. Aus praktischen Gründen eignet sich das Ausgabeformat des IOP-Assemblers recht gut für diesen Zweck. Der Simulator-Dump in Anhang A: Ein Beispielprogramm zeigt einen Ausschnitt aus einem assemblierten Maschinenprogramm mit Kommentaren zum Ablauf der Simulation.

4.6 Data Memory

Der Datenspeicher ist funktionell genau wie das Registerfile implementiert. Einzig seine Kapazität ist mit 9192 Einträgen á 16 Bit größer. Bei der Modellierung gab es keine weiteren Besonderheiten zu beachten; auch hier wurde ein simples Array benutzt. Schreibzugriffe werden im Dump-File vermerkt.

4.7 Scheduler

Das beschriebene statische Scheduling und die Vermischung von Kontrollinformationen mit Daten bringt einige Probleme mit sich, die bei der Implementierung der Architektur und natürlich auch des Simulators beachtet werden mußten:

Der Program Counter erfordert einen etwas umfangreicheren Algorithmus für seine Aktualisierung.

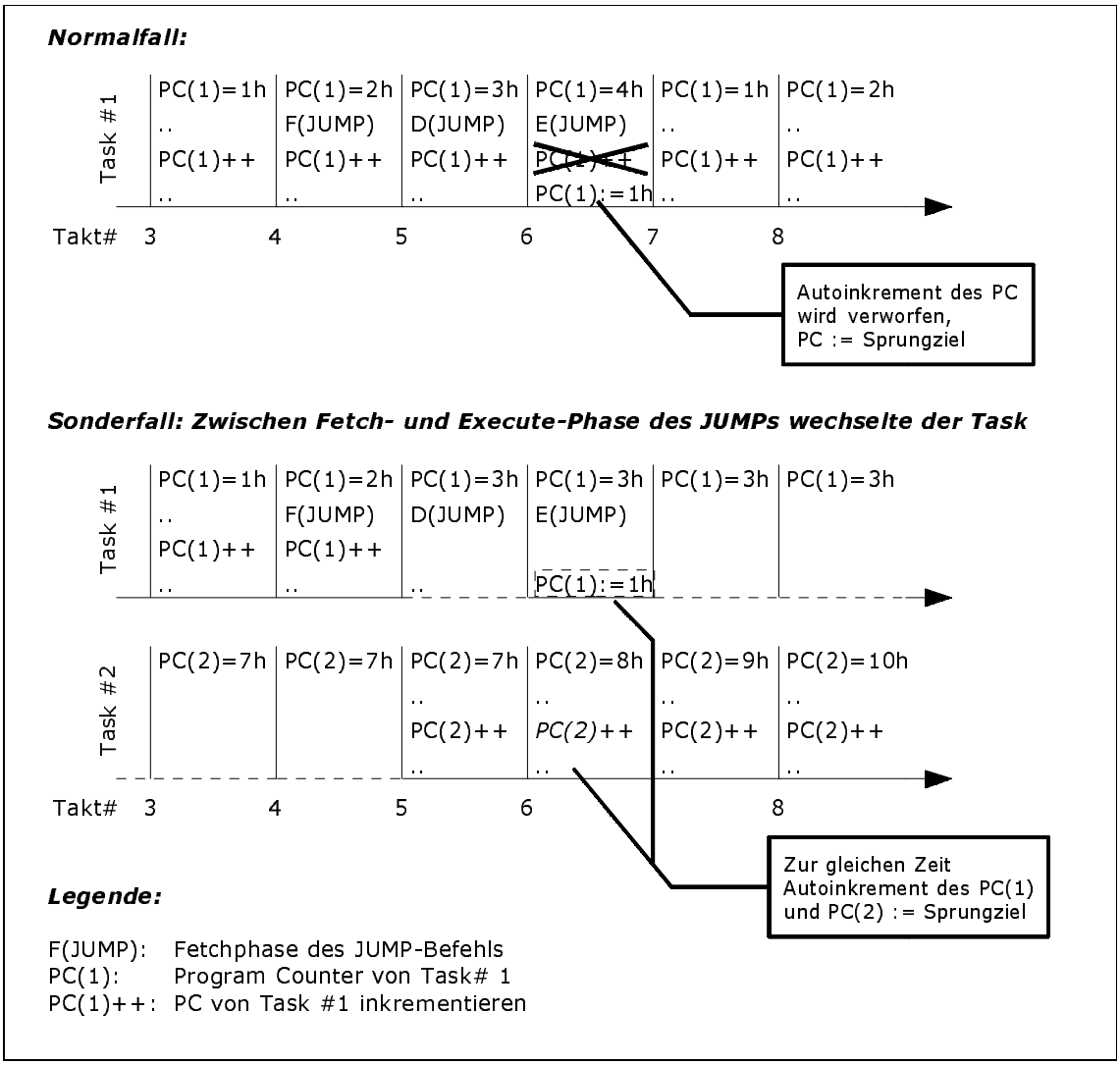


Abbildung 4.3: Aktualisierung von PC's unter verschiedenen Bedingungen

Der PC sollte im Normalfall während der Fetch-Phase einer Instruktionen um 1 inkrementiert und zurückgeschrieben werden. Im Falle eines Sprungbefehls ist das Ziel des Sprungs aber erst nach der Execute-Phase der Pipeline, also 2 Takte später verfügbar. Zudem liegt der neue Wert des PC am Ausgang der ALU. Somit muß die PCU in diesem Takt entscheiden, ob der um eins inkrementierte neue PC-Wert, der in der Fetch-Phase des aktuellen Takts erzeugt wurde, oder der durch den Sprung in der ALU erzeugte PC-Wert ins Registerfile zurückgeschrieben wird. Erschwert wird diese Entscheidung noch dadurch, daß der neue Befehl, der gerade seinen inkrementierten PC zurückschreiben möchte, gar nicht zu gleichen Task gehören muß, der den Sprungbefehl gefetcht hat, es sich also um verschiedene PSW's handelt. Abbildung 4.3 veranschaulicht die Lösung des Problems.

Das synchrone Design wurde mit Hilfe eines einzelnen Prozesses implementiert.

Dies geschah aus Überlegungen zur Verbesserung der Performance heraus. Auf diese Weise ist es möglich, alle internen Register des Schedulers als Variable zu implementieren.

Alle Aktionen zur Konfiguration des Schedulers werden über Kommandos, die von der PCU während der Execute-Phase gegeben werden, ausgelöst. Diese Kommandos sind als Aufzählungstyp in IOP_CONSTANTS definiert.

4.8 Interrupt Unit

Vorrangige Aufgabe war, die recht komplizierte Interrupt Logik in ein übersichtliches und einfach erweiterbares Konzept zu bringen:

- Eingehende Interrupts werden zu allererst abgespeichert.
- Grundlage des Algorithmus für die weitere Behandlung ist die vom Scheduler eintreffende Tasknummer und eine Statusvariable, die speichert, ob mindestens ein Interrupt anliegt und auf Behandlung wartet. Sie entscheiden über die weitere Vorgehensweise.
- Liegt keine Unterbrechung an, wird die Tasknummer zum Registerfile durchgereicht.
- Liegt eine Unterbrechung und eine Nummer ungleich Task 31 an, so wird der Teilalgorithmus für Polled Interrupts abgearbeitet. Er sucht in ILR und ITR, ob der Tasknummer ein Polled Interrupt zugeordnet ist und ob dieser ausgelöst wurde. Bei einem positiven Ergebnis gibt die Einheit zusammen mit der unveränderten Tasknummer ein Signal an die PCU aus, das zum Aufwecken des schlafenden, aber eingeplanten Tasks führt. Ist die Suche erfolglos, wird nur die Tasknummer weitergereicht.
- Bei anliegender Tasknummer 31 handelt es sich um eingeplante Rechenzeit zur Behandlung von Vectored Interrupts. Die Tabelle ITR wird danach durchsucht, ob irgendeine Quelle für Vectored Interrupts konfiguriert ist, und wenn, dann ob diese Quelle den Interrupt ausgelöst hat. Ist ein einziger solcher Interrupt aufgetreten, wird einfach die der Quelle zugeordnete Tasknummer aus dem ILR statt der Nummer 31 zum Registerfile weitergegeben. Sollte es mehrere Interruptquellen geben, die auf eine Behandlung warten, so entscheidet eine Prioritätenlogik, welcher Interrupt ausgewählt wird. Dann fährt der Algorithmus in der Art fort, wie eine einzige Quelle behandelt werden würde.

Um die Logik schnell simulierbar und trotzdem skalierbar zu implementieren, wurde

das Prioritäten-Konzept, wie es in [ENG98/99] beschrieben ist, in eine Look-Up Tabelle umgesetzt. Tabelle 4.2 ordnet Paaren aus einer IPR-Maske und einer Interrupt-Quelle eine Priorität zwischen 0 und 4 zu. Je kleiner der Wert, desto höher ist die Priorität einer Quelle. Beispielsweise ist dem Paar ('001', BiDir) die Priorität 3 zugeordnet. Das bedeutet, daß im Vector Based Mode bei der Konfiguration des IPR mit '001' die drei Software-Interrupts GPP0, GPP1, GPP2 gegenüber dem Bidirektionalen Port bevorzugt behandelt werden. In Spalte 'Others' wird an alle im Originalkonzept nicht berücksichtigten Interruptquellen eine sehr niedrige Priorität vergeben.

Priorität zwischen den Interruptquellen	IPR-Maske	Priorität der Quelle bei IPR-Maske 'nnn'				
		GPP0	GPP1	GPP2	BiDir	Others
GPP0>GPP1>GPP2>BiDir>Others	*'000'*	*0*	*1*	*2*	*3*	*4*
...	*'001'*	*1*	*0*	*2*	*3*	*4*
...	*'010'*	*0*	*1*	*3*	*2*	*4*
...	*'011'*	*1*	*0*	*3*	*2*	*4*
...	*'100'*	*2*	*3*	*0*	*1*	*4*
...	*'101'*	*3*	*2*	*0*	*1*	*4*
...	*'110'*	*2*	*3*	*1*	*0*	*4*
BiDir>GPP2>GPP1>GPP0>Others	*'111'*	*3*	*2*	*1*	*0*	*4*

Tabelle 4.2: Look Up Tabelle für Interrupt-Prioritäten

Interrupt und Änderungen der Register werden im Dump-File protokolliert.

4.9 PCU - Die Steuereinheit

In der zentralen Schaltstelle des IOP ist die Zustandsmaschine der dreistufigen Pipeline implementiert. Drei nebenläufige, wegen der Pipelinefunktion Clock synchrone Prozesse, je einer für Fetch, Decode/Load und Execute/Store, steuern die gesamten Abläufe im IOP Simulator. Alle Prozesse arbeiten zeitunabhängig immer den gleichen Algorithmus auf verschiedenen Daten ab.

Alle Instruktionen 'wandern' mit zugehörigen Verwaltungsinformationen und Daten durch die Kette der 3 Pipelineprozesse. Dazu werden Daten, die von Interesse für nachfolgende Stufen der Befehlspipeline sind, in 'Pipe Arrays' gespeichert. Aus Abbildung 4.4 ist ersichtlich, wie die Pipe Arrays in der Art einer rotierenden Scheibe arbeiten, die sich nach jedem Zyklus um eine Position weiter dreht. Nach 3 Zyklen hat die Instruktion

die Pipeline verlassen und wird im Pipe Array von den Daten der nächsten überschrieben.

Die Funktion prec() errechnet den Vorgänger des zyklischen Positionszählers auf der Scheibe mit dem Wertebereich 0-2; Prec(2) = 1, Prec(0) = 2. Diese Datenstruktur hat für den Simulator gegenüber der in Hardwareentwürfen verwendeten Implementierung von Pipeline-Registern den Vorteil, daß bei einer nachträglichen Änderung der Pipelinetiefe, z.B. im Zuge einer Skalierung, nur geringfügige Anpassungen erforderlich sind.

Wird beim Fetch festgestellt, daß das Run-Flag des Tasks auf '0' steht, so wird ein NOP statt der Instruktion im Pipe Array gespeichert. Die nachfolgenden Pipelinestufen ruhen dann automatisch bis zur nächsten Instruktion.

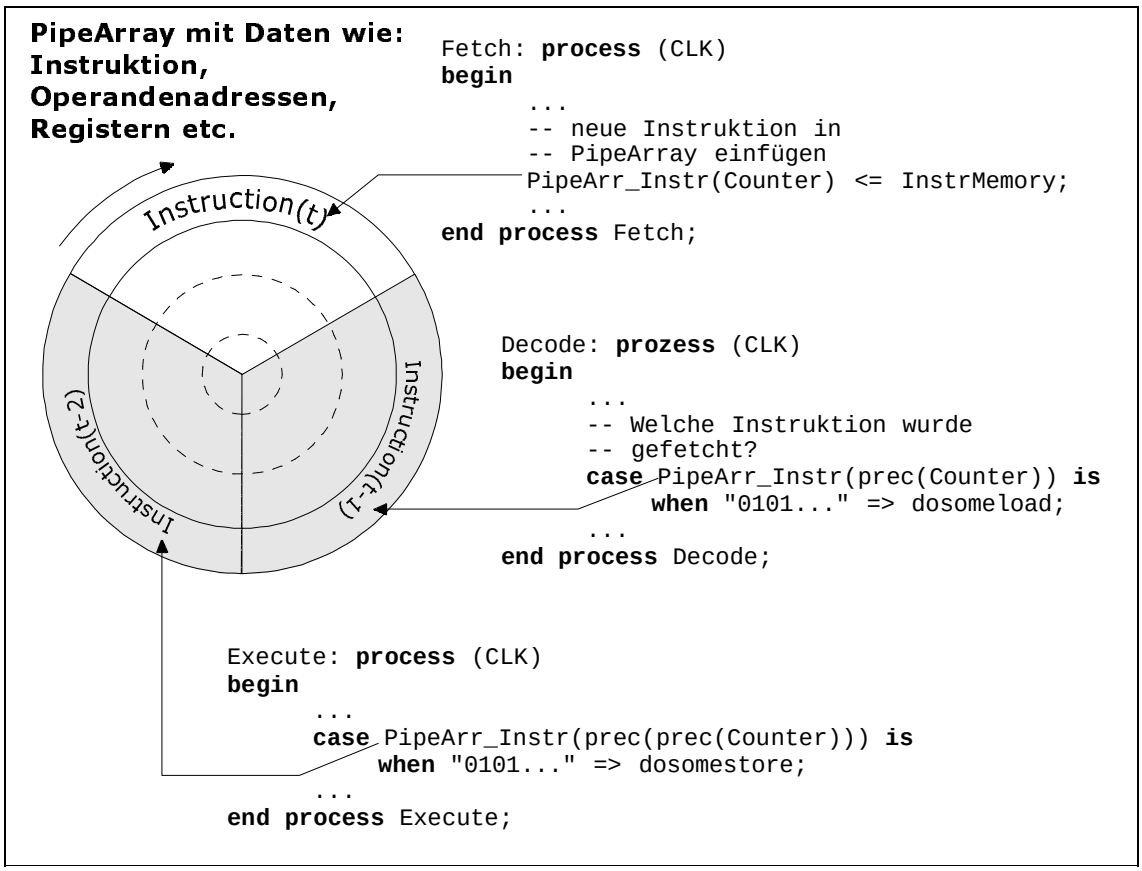


Abbildung 4.4: In der PCU arbeiten 3 Pipelinestufen zur gleichen Zeit an verschiedenen Daten

4.10 Multi- und Demultiplexer

Für beide Komponenten ist jeweils ein Aufzählungstyp definiert, der Routing zu allen angeschossenen Einheiten plus gesperrtem Durchgang auflistet. Die Komponenten arbeiten asynchron, reagieren also sofort auf Änderungen an den Daten- und Steuerein-

gängen mit Änderungen an den Ausgängen. Die Ein- und Ausgänge sind 2x16 Bit breit, die Steuereingänge sind vom genannten Aufzählungstyp.

4.11 Recheneinheit (ALU)

Die Pipeline in der ALU ist durch einen Clock-Synchronen Prozeß implementiert worden. Dieser reagiert nicht auf jede Änderung während der Decode-, sondern wird erst während der Execute-Phase der PCU gestartet. Das Alignment der Operanden wird also im Gegensatz zum Vorbild im gleichen Taktzyklus wie die restliche Verarbeitung durchgeführt. Dies hat keine Auswirkungen auf die Übereinstimmung von Modell und Original auf der Register-Transfer Ebene.

Kern der Verarbeitung ist wie in der Komponente PCU eine CASE-Schleife, die für Gruppen von Befehlen, die ein ähnliches Verhalten zeigen, Algorithmen implementiert.

```
...
case ALU_Command of
           ...
           ADD: when       -- Addition
              ADD | ADD! | ADD32 | ADDI =>

              Y := Y + X;
              ...


           SUB: when       -- Subtraktion
              SUB | SUB! | SUB32 | SUBI =>

              Y := Y - X;
              ...

           ...
           LOAD_STORE: when-- Transport, nur durchleiten
              LDRM16 | LDRM16! | LDRM32... =>

              null;        -- tue nichts
```

Abbildung 4.5: Unterteilung der Instruktionen in Gruppen mit ähnlichem Verhalten

Wie in Abbildung 4.5 zu sehen, werden beispielsweise bei den Instruktionen AND, ANDI und AND! Die beiden 16 Bit Operanden X und Y durch ein logisches 'AND' verknüpft und das ZERO-Flag wird auf '1' gesetzt, falls das Ergebnis aus lauter Nullen besteht, andernfalls ist ZERO = '0'. Die unterschiedliche Herkunft der Operanden bei AND (beide Operanden kommen aus dem Registerfile) und ANDI (Operand X stammt aus der Instruktion) spielt nur bei der Behandlung der Instruktionen in der PCU eine Rolle. Dort wurde dafür gesorgt, daß die benötigten Daten zur richtigen Zeit in den

beiden Eingangsregistern der ALU stehen. Ein Unterschied muß allerdings doch bei dieser Befehlsgruppe gemacht werden. Die Instruktion AND! Schreibt statt eines 16 Bit Wertes zwei zurück. Das ist sinnvoll, weil alle ALU-Operationen auf dem beschriebenen 16 Bit Bereich innerhalb eines 2x16 Bit Wortes ausgeführt werden.

Die Unterscheidung, ob im konkreten Fall ein oder zwei 16 Bit Worte geschrieben werden müssen, wird aber nicht jeweils im CASE-Zweig der Instruktion, sondern durch die PCU in der Execute-Phase, anhand eines Bits, daß Operationen mit 2x16 Bit Ergebnis eindeutig identifiziert, getan. Die PCU setzt die Enable-Leitungen für das Schreiben von 2x16 Bit Werten auf den Wert des Identifikationsbits der Instruktion.

4.12 DSP-Buffer und AGU

Die Komponente enthält eine Dummy-Logik. Diese wechselt bei Read- und Write-Requests immer zwischen folgenden beiden Zuständen:

- Die geforderten Deadline des Datentransfers zum oder vom DSP-Speicher wird eingehalten.
- Die Deadline wird überschritten.

Dadurch ist es einfacher, die Funktion bestimmter Mechanismen bei der DSP-Puffer Steuerung im Zusammenspiel mit DSP und dessen AGU besser nachvollziehen zu können.

Der DSP-Buffer ist ansonsten der Implementierung des Registerfiles angelehnt.

5 Ergebnisse und Ausblick

Die Anforderungen der Aufgabenstellung wurden in dem ca. 5000 Zeilen VHDL Code umfassenden Projekt realisiert.

Bei der Arbeit wurde vor allem auf Funktion, Wart- und Skalierbarkeit Wert gelegt.

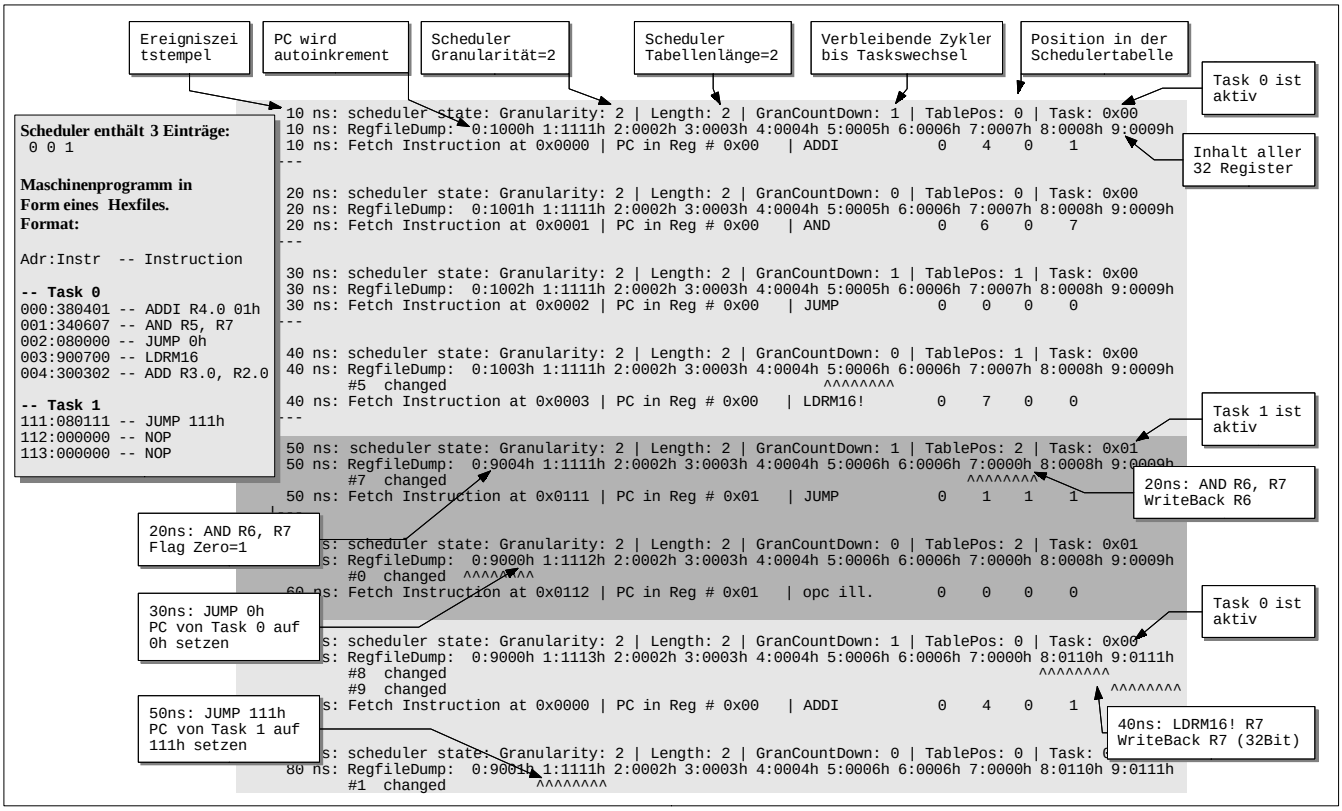
Für umfangreiche Performancetests und Optimierungen blieb leider nicht genug Zeit. Als Orientierung soll ein Lauf eines nur aus NOP's bestehenden Programms und eingeschalteter Dump-Funktion auf einem Intel Celeron, 366 MHz, 128 MB RAM, Win98, Entwicklungsumgebung Active VHDL [ALD99], genügen. Es wurden durchschnittlich 150 Takte des IOP je Sekunde Simulatorlaufzeit simuliert.

Beschleunigung und Verbesserungen sind zahlreich möglich. Beispielsweise wurde ausschließlich der Typ Std_Logic verwendet. Im Hinblick auf eine möglichst hohe Simulationsgeschwindigkeit sollte jedoch überall dort, wo dessen Resolution-Funktion nicht benötigt wird, statt dessen der nicht aufgelöste Typ Std_ULogic verwendet werden. In der Fachliteratur ist beschrieben, daß ein Aufruf der jeweiligen Resolution-Funktion des Objekts bei jeder Referenzierung nicht unerheblichen Overhead erzeugt. [LEH94],

[CHA97],[ASH90],[COH95]

Im Nachhinein hat sich gezeigt, daß die Implementierung gänzlich ohne irgendwelche Resolution-Funktionen auskäme, was vor allem im Ersatz der Bussysteme durch Multiplexer begründet liegt.

Weiterhin ist im Simulator in seiner jetzigen Form nur rudimentäre Unterstützung für Debugging enthalten. Die Ergebnisse im Dump-File könnten grafisch aufbereitet dargestellt und einer nachfolgenden schrittweisen Verfolgung zur Verfügung gestellt werden.


Ereigniszeitstempel
PC wird autoinkrement
Scheduler Granularität=2
Scheduler Tabellenlänge=2
Verbleibende Zykler bis Taskswechsel
Position in der Schedulertabelle
Task 0 ist aktiv
Inhalt aller 32 Register
Task 1 ist aktiv
20ns: AND R6, R7 WriteBack R6
Task 0 ist aktiv
40ns: LDRM16! R7 WriteBack R7 (32Bit)
20ns: AND R6, R7 Flag Zero=1
30ns: JUMP 0h PC von Task 0 auf 0h setzen
50ns: JUMP 111h PC von Task 1 auf 111h setzen
Scheduler enthält 3 Einträge:
0 0 1
Maschinenprogramm in Form eines Hexfiles.
Format:
Adr:Instr -- Instruction
-- Task 0
000:380401 -- ADDI R4.0 01h
001:340607 -- AND R5, R7
002:080000 -- JUMP 0h
003:900700 -- LDRM16
004:300302 -- ADD R3.0, R2.0
-- Task 1
111:080111 -- JUMP 111h
112:000000 -- NOP
113:000000 -- NOP
10 ns: scheduler state: Granularity: 2 | Length: 2 | GranCountDown: 1 | TablePos: 0 | Task: 0x00
10 ns: RegfileDump: 0:1000h 1:1111h 2:0002h 3:0003h 4:0004h 5:0005h 6:0006h 7:0007h 8:0008h 9:0009h
10 ns: Fetch Instruction at 0x0000 | PC in Reg # 0x00 | ADDI 0 4 0 1
20 ns: scheduler state: Granularity: 2 | Length: 2 | GranCountDown: 0 | TablePos: 0 | Task: 0x00
20 ns: RegfileDump: 0:1001h 1:1111h 2:0002h 3:0003h 4:0004h 5:0005h 6:0006h 7:0007h 8:0008h 9:0009h
20 ns: Fetch Instruction at 0x0001 | PC in Reg # 0x00 | AND 0 6 0 7
30 ns: scheduler state: Granularity: 2 | Length: 2 | GranCountDown: 1 | TablePos: 1 | Task: 0x00
30 ns: RegfileDump: 0:1002h 1:1111h 2:0002h 3:0003h 4:0004h 5:0005h 6:0006h 7:0007h 8:0008h 9:0009h
30 ns: Fetch Instruction at 0x0002 | PC in Reg # 0x00 | JUMP 0 0 0 0
40 ns: scheduler state: Granularity: 2 | Length: 2 | GranCountDown: 0 | TablePos: 1 | Task: 0x00
40 ns: RegfileDump: 0:1003h 1:1111h 2:0002h 3:0003h 4:0004h 5:0006h 6:0006h 7:0007h 8:0008h 9:0009h
#5 changed ^^^^^^^^
40 ns: Fetch Instruction at 0x0003 | PC in Reg # 0x00 | LDRM16! 0 7 0 0
50 ns: scheduler state: Granularity: 2 | Length: 2 | GranCountDown: 1 | TablePos: 2 | Task: 0x01
50 ns: RegfileDump: 0:9004h 1:1111h 2:0002h 3:0003h 4:0004h 5:0006h 6:0006h 7:0000h 8:0008h 9:0009h
#7 changed ^^^^^^^^
50 ns: Fetch Instruction at 0x0111 | PC in Reg # 0x01 | JUMP 0 1 1 1
s: scheduler state: Granularity: 2 | Length: 2 | GranCountDown: 0 | TablePos: 2 | Task: 0x01
s: RegfileDump: 0:9000h 1:1112h 2:0002h 3:0003h 4:0004h 5:0006h 6:0006h 7:0000h 8:0008h 9:0009h
#0 changed ^^^^^^^^
60 ns: Fetch Instruction at 0x0112 | PC in Reg # 0x01 | opc ill. 0 0 0 0
s: scheduler state: Granularity: 2 | Length: 2 | GranCountDown: 1 | TablePos: 0 | Task: 0x00
s: RegfileDump: 0:9000h 1:1113h 2:0002h 3:0003h 4:0004h 5:0006h 6:0006h 7:0000h 8:0110h 9:0111h
#8 changed ^^^^^^^^
#9 changed ^^^^^^^^
s: Fetch Instruction at 0x0000 | PC in Reg # 0x00 | ADDI 0 4 0 1
s: scheduler state: Granularity: 2 | Length: 2 | GranCountDown: 0 | TablePos: 0 | Task: 0
80 ns: RegfileDump: 0:9001h 1:1111h 2:0002h 3:0003h 4:0004h 5:0006h 6:0006h 7:0000h 8:0110h 9:0111h
#1 changed ^^^^^^^^

Anhang B: Glossar

AGU

Adress Generation Unit, Teil des DSP, der dessen Speicher steuert. Im Zusammenspiel mit dem IOP wird die AGU benötigt, um Lese- oder Schreibpuffer von oder zum DSP-Speicher zu transferieren.

ALU

Arithmetic Logic Unit, Recheneinheit eines Prozessors. Ist für alle Verknüpfungen zwischen Operanden zuständig. Oftmals, wie beim ➔**IOP**, ist sie zudem noch zentraler Teil des ➔**Datenpfades**. Dann läuft selbst jeder Datentransport über diese Einheit.

ANSI

American National Standards Institute

Architecture

➔**Entity**

CISC

Prozessorarchitektur, die zu einer Zeit entstand, als Computer noch hauptsächlich mühsam mit Hilfe von Assemblern programmiert wurden. Um wertvolle Entwicklungszeit zu sparen, waren die Befehlssätze deswegen sehr umfangreich. Seit dem Programme fast ausnahmslos mit Hilfe von Compilern erzeugt werden, hat sich gezeigt, daß oft 60 bis 90 Prozent der Befehle des CISC-Befehlssatzes überhaupt nicht oder sehr selten benutzt werden. siehe auch ➔RISC

Datenpfad

Ein oder mehrere Wege in einem Prozessor, den Daten während ihrer Verarbeitung gehen. Ein möglicher Datenpfad wäre z.B. vom Speicher über die ➔**ALU** wieder zum Speicher.

Die

Chipfläche, auf der sich die komplette integrierte Schaltung befindet. Begriff aus der Halbleiterfertigung.

DoD

Department of Defense, Amerikanisches Verteidigungsministerium;

Weltweit größter Abnehmer von Hochtechnologie

DSP

Digitaler Signal Prozessor; Bezeichnung für Prozessoren, deren Architektur auf die schnelle und kontinuierliche Verarbeitung von Meßdaten optimiert ist.

Entity

In VHDL sind Komponenten in die Schnittstelle zur Umgebung sowie die eigentliche Funktionalität geteilt. Eine Entity stellt in VHDL die Schnittstelle einer Funktionseinheit zu ihrer Umwelt dar. Bei dem Modell eines Mobiltelefons würde das Entity 'Handy' also mindestens die Zifferntasten '0' - '9' sowie '#' und '*' enthalten. Die Funktionalität selbst wird in VHDL in der ➔**Architecture** beschrieben.

GPR

General Purpose Register, Register können ohne jegliche Einschränkung oder Unterschiede gleichwertig benutzt werden.

Harvard-Architektur

Prozessorarchitektur, bei der Befehls- und Datenspeicher getrennte Einheiten sind. Ziel ist es, den sogenannten Flaschenhals der ➔ **v. Neumann-Architektur** aufzuweiten.
Wird sehr oft in DSP-Designs eingesetzt.

IEEE

Institute of Electrical and Electronics Engineers

ILR

Interrupt Link Register, ordnet jeder ➔**Interruptquelle** die ➔**Tasknummer** des verantwortlichen ➔**Prozesses** zu.

Interruptquellen

Im IOP können Unterbrechungen von 13 Quellen ausgelöst werden:

- Nicht maskierbarer Interrupt (NMI): Supervisorcall, kann nicht deaktiviert werden
- GPP0 - GPP2: frei verwendbare Softwareinterrupts
- je zwei serielle und parallele InPorts und OutPorts
- BiDir: Bidirektionaler Port

IMR

Interrupt Mask Register, Alle Interruptquellen können einzeln oder in Gruppen als ➔**Polled** oder ➔**Vectored** aktiviert oder deaktiviert werden.

I/O Prozessor, IOP

Ein- / Ausgabeprozessor. Speziell für die Manipulation von Datenströmen konzipierter Prozessor. Arbeitet einem hochspezialisierten ➔**DSP** zu.

IPR

Interrupt Priority Register, den 4 ➔**Interruptquellen** GPP0, GPP1, GPP2 und BiDir werden Prioritäten zugeordnet. Beachtet wird die Rangfolge nur im ➔**Vectored Mode**.

ITR

Interrupt Type Register, speichert für jede ➔**Interruptquelle**, ob sie vom Typ ➔**Polled** oder ➔**Vectored** ist.

v. Neumann-Architektur

Älteste und sehr weit verbreitete Computerarchitektur; besitzt einen frei adressierbaren Speicher, in dem sich Befehle und Daten befinden. Die Entwicklung der Speichergeschwindigkeiten konnte immer weniger mit der von Prozessoren mithalten. Man spricht deswegen auch vom 'von Neumannchen Flaschenhals'.

Multitasking

Ist die Fähigkeit eines Controllers, Prozessors, Computersystems oder Betriebsystems, mehrere Programme zur gleichen Zeit ausführen zu können. Programme, die auf solch einem System laufen, heißen Task oder Prozeß. Meist ist die gleichzeitige Ausführung nur quasiparallel, wenn die gesamte verfügbare Rechenzeit des Prozessors in sehr kleine Zeiteinheiten unterteilt wird, in denen dann jeweils nur ein Task auf dem Prozessor läuft.

MSB

Most Significant Bit, Verkörpert in der Binärdarstellung von Zahlen die höchste 2er-Potenz.

Load/Store

Befehlssatz, dessen Befehle nur Inhalte von Registern manipulieren können. Zusätzlich gibt es die Load- und Store-Instruktionen, die Register aus dem Speicher laden oder in den Speicher zurückschreiben. Ist häufig in Prozessorarchitekturen im Zusammenhang mit ➔**Pipelines** und ➔**RISC**-Befehlssatz anzutreffen.

Opcode

Ist die Bezeichnung für den Teil eines Maschinenbefehls, der nur die Operation für den Prozessor, wie Addition, Load, Store etc., kodiert enthält. Im Maschinenbefehl sind weiterhin Adressen der Operanden, teilweise die Operanden selbst oder weitere Verwaltungsinformationen enthalten.

Pipeline

Einheit in Prozessoren, die es erlaubt, auch bei hohen Taktfrequenzen einen Durchsatz von 1 Befehl/Taktzyklus zu erreichen. Charakteristisch ist, das die Ausführung aller Instruktionen in Befehlsphasen, wie z.B. Fetch, Decode, Load, Execute oder Store, unterteilt werden. Für jede Phase ist eine eigene Einheit im Prozessor zuständig.

➔**RISC**-Prozessoren eignen sich aufgrund der gleichmäßigen Struktur ihres Befehlssatzes sehr gut zur Implementierung von ➔**Pipelines**.

Polled Interrupt

Jeder Interruptquelle ist ein verantwortlicher Task zugeordnet. Diese Tasks müssen im Scheduler eingeplant sein, sind aber durch das Programm in den Schlafmodus versetzt und tun im Normalfall nichts. Ist ein Task an der Reihe, dessen zugeordneter Interrupt ausgelöst wurde, so wird der Tasks von der Interrupt einheit aufgeweckt.

Prozeß

➔**Task**

PSW

Program Status Word, beschreibt den Zustand des Tasks. Enthält Programm Counter (PC) und Flags. Ist in einem Register des ➔**Registerfile** gespeichert. Die folgenden 16 Register gehören zum ➔**Task-Window** des Tasks.

Realtime

ist im Zusammenhang mit Computersystemen ein Qualitätsmerkmal von Hard- und/ oder Software. Ein Realtime-System muß garantieren können, daß es auf jedes Ereignis innerhalb einer maximalen Zeitspanne, die von System zu System verschieden ist, reagieren kann.

Registerfile

Satz von 32 ➔**GPR** Registern im IOP, von denen je Task maximal 16 vorgegebene verwendet können.

Resolution Function

Werden einem ➔**Signal** von mehreren Treibern gleichzeitig Werte aufgeprägt, muß der Simulator wissen, nach welchem Modell das Gesamtsignal gebildet werden soll. Ein solches Modell wird bei VHDL innerhalb einer Auflösungsfunktion - Resolution Function - implementiert, die als Parameter vom System alle Werte übergeben bekommt, die einem Signal gleichzeitig aufgeprägt werden. Als Ergebnis gibt die Funktion den resultierenden Wert des Signals zurück. Auf diese Weise läßt sich das Verhalten realer Bussysteme, wie z.B. dominierende '0' oder dominierende '1' einfach nachbilden.

RISC

Prozessorarchitektur mit gegenüber CISC-Prozessoren vergleichsweise sehr kleinem Instruktionssatz, der zusammen mit der gleichmäßigen Struktur der Instruktionen zu sehr einfach aufgebauten und schnellen Prozessordesigns führt.
Man spricht auch oft von fest verdrahtetem Befehlssatz.

SFB 358

Im Sonderforschungsbereich 358 für Automatisierten System-Entwurf, Teilprojekt A6, Mobilfunk wird an Entwurfssystemen für skalierbare Prozessorfamilien zur Signalverarbeitung geforscht. Der schon in Anwendung befindliche I/O Prozessor ist ein Teilprojekt. Beteiligt sind die TU-Dresden und die TU Ilmenau.

Signale

Gekappselte Objekte eines beliebigen VHDL-Datentyps, deren Wert nach festgelegten Regeln geändert werden kann und die über ihren Signalverlauf Buch führen. Im Gegensatz zu Variablen 'weiß' ein Signal z.B. zu welchem Zeitpunkt sein Wert das letzte Mal geändert wurde. Solches 'Wissen' ist über Attribute des Signals zugänglich. Zum Beispiel ist der Zeitpunkt der letzten Änderung des Wertes eines Schalters über 'Schalter'LAST_EVENT' abfragbar.
Signale werden in VHDL hauptsächlich eingesetzt, um das Verhalten von physikalischen Leitungen zu modellieren.

Task

Programm in ➔**Multitasking**system, das parallel zu anderen Programmen ausgeführt wird. Es besitzt eigene und von allen anderen Prozessen getrennt verwaltete Ressourcen. Beim IOP sind das die Register und das ➔**MSW**

Tasknummer

Im IOP wird ein Task durch sein ➔**PSW** identifiziert. Die Tasknummer entspricht der Adresse des PSW im Registerfile.

Registerfile-Eintrag 0 ist also automatisch das PSW von Tasks Nummer 0. Eintrag 1 ist PSW von Task 1 usw.

Task-Window

Ein Task kann im IOP auf die seinem ➔**PSW** im ➔**Registerfile** maximal 16 folgenden Register über R0-R15 **zugreifen.**

Testbench

Fachbegriff für spezielle zu Testzwecken erstellte Modelle in VHDL. Diese dienen nur dazu, eine oder mehrere VHDL-Komponenten auf die Erfüllung iher Anforderungen zu überprüfen. Dazu wird die Unit Under Test (UUT) durch die Testbench mit verschiedenen ausgewählten oder allen möglichen Eingangssignalen getrieben. Das Verhalten, also die Summe der Veränderungen der Ausgangssignale wird mit vorher berechneten oder mit Hilfe vorheriger Testläufe gesammelter Prüfdaten verglichen.

Da es sich bei den Komponenten fast immer um Zustandsautomaten handelt, ist es jedoch meist wegen der unvorstellbar hohen Anzahl an möglichen Kombinationen von Eingangswerten und inneren Zuständen praktisch nicht möglich, einen vollständigen Test durchzuführen. Schon ein einfacher 32 Bit Zähler mit einem 1 Bit Eingangssignal müßte zum vollständigen Test 8 Milliarden Zyklen durchlaufen. Jedes weitere Bit verdoppelt diese gigantische Anzahl. Prozessoren enthalten aber eine Vielzahl an Registern und besitzen viele Eingansleitungen.

Vielmehr werden kritische Datenbereiche, in denen z. Beispiel Überläufe zu erwarten sind etc. getestet.

Vectored Interrupt

Unterschied zu ➔**Polled Interrupt** ist, daß hier Zeit zur Behandlung aller Vectored Interrupts über einen Token, die Task #31, eingeplant ist. Die nächste behandelte Interruptquelle wird aus allen anstehenden Interrupts nach ihrer Priorität ausgewählt. Der Token wird dann durch die zugehörige Tasknummer ersetzt.

Waveform

Sichtweise auf Fortschreiten der Simulation innerhalb einer VHDL-Simulationsumgebung. Wird zur Visualisierung und zum Debugging genutzt.

2 Operanden-Befehlssatz

Bei diesen Befehlssätzen wird im Gegensatz zu 3-Operanden-Befehlssätzen einer der Operanden mit dem Ergebnis der Operation, wie bei 'Y:=Y+1', überschrieben. Vorteile liegen in der kompakten Instruktionsbreite, da nur die Adressen zweier Operanden kodiert sein muß. 3-Operanden-Befehlssätze sind meist effizienter.

Quellenverzeichnis

[ALD99] Aldec, Inc., Active VHDL V3.3/Build 0228, auf 1 ms Simulationszeit beschränkte Demoversion von Aldec Inc., 1999, www.aldec.com

[ASH90] Ashenden, Peter J., The VHDL Cookbook, First Edition, Dept. Computer Science, University of Adelaide, South Australia

[CHA97] Chang, K.C., Digital Design and Modeling with VHDL and Synthesis; 1997, IEEE Computer Society Press, Los Alamos, California

[COH95] Cohen, Ben, VHDL Coding Styles and Methodologies, 1995, Kluwer Academic Publishers

[ENG98/99] Engel, Frank, M3's IO-Prozessor Architecture Description, Manual Version 1.5, 1999, verschiedene Abschnitte anderer Versionen, 1998,1999, Technische Universität Dresden, Institut für Mobile Nachrichtentechnik.

[LEH94] Lehmann, G., Schaltungsdesign mit VHDL, 1994, Franzis Verlag

[WAP99] Wappler, Marcel, VHDL-Simulator eines neuartigen Datenstromprozessors, 1999, http://www.marcel-wappler.de/iop_simulator