Inhaltsverzeichnis

Kapitel 1

Einleitung

Dieser Endbericht soll eine umfassende Übersicht der Entwicklung der Projektgruppe SYSTEMENTWURF vom WS94/95 und SS95 wiedergeben. Daneben sind sowohl die Ergebnisse der einzelnen Arbeiten sowie die Hilfsmittel beschrieben.

Ziel dieser Projektgruppe war die Entwicklung eines Koprozessors Spline Function Unit (SFU) für den Prozessor SPARC 601 (CPU) für die Berechnung von kubischen Splines, d.h. zu gegebenen Stützpunkten im kartesischen Koordinatensystem soll der Koprozessor eine Menge stetig ineinander übergehender Funktionen liefern, so daß alle Stützpunkte auf mindestens einer Funktion liegen. Dabei müssen alle Funktionen den Grad 3 (kubisch) haben.

Grundlegend für unsere Arbeit war eine Reihe von Seminarvorträgen, die alle Projektgruppenteilnehmer sowohl mit dem Projekt selber als auch mit den zu verwendenden Softwaretools vertraut machen sollte. Es gab also zu jedem größeren Tool einen eigenen Vortrag zu dessen Sinn, Nutzen und Gebrauch.

Nach Abschluß der Seminarphase, die uns nur einen theoretischen Überblick verschaffte, legten wir die kommunikationstechnischen Daten des Koprozessors fest. Dazu gehörten z.B. die Beschränkung der Anzahl der Stützstellen auf 15 der verfügbaren Register für die Prozessor-Koprozessor-Kommunikation und natürlich die nötigen Befehlssätze für die korrekte Steuerung der einzelnen Komponenten. Allerdings sei an dieser Stelle gesagt, daß uns hardwaremäßige Restriktionen zu ständigen Änderungen unserer Absprachen zwangen.

Die Interfacetechnik zwischen dem SPARC 601 und unserem Koprozessor SFU beschäftigte uns eine sehr lange Zeit. Zum einen hatten wir keine ganz genauen Vorstellungen, welche Befugnisse im Bereich der Systemsteuerung jede einzelne Komponente haben sollte, zum anderen handelt es sich ja um eine Mehr-Benutzer-Umgebung, so daß unsere SFU stets mit dem richtigen Prozeß zu synchronisieren war. Diese beiden Tatsachen ließen sich nur sehr mühselig mit unseren Vorstellungen in Einklang bringen.

Da wir nur eingeschränkte Simulationsmöglichkeiten hatten, mußten wir außerdem auf den Resourcenverbrauch achten. Der Chip durfte gewisse Dimensionen bei der Anzahl einzelner Gatter nicht überschreiten. Daraus folgte, daß wir sämtliche algorithmischen Vorgänge und auch das Interface bestmöglich zu optimieren hatten. Das hatte jedoch auch einen positiven Seiteneffekt gehabt: wir fanden einen besseren Algorithmus zur Splineberechnung, der mit kleineren Matrizen auskommt und schneller rechnet.

Die weiteren Aktionen waren weitgehend geprägt durch: Fehler entdecken und mögliche Lösungen zu deren Beseitigung suchen, wobei mit Fehler sowohl logische als auch Unzureichlichkeiten unserer Softwaretools gemeint sind. Wenn z.B. angemahnt wird, daß zwischen CPU und SFU in beiden Richtungen (bidirektionaler Bus) kommuniziert wird, dann muß man sich schon etwas einfallen lassen, schließlich ist das ja der Sinn und Zweck.

Desweiteren bestanden teilweise erhebliche Schwierigkeiten bei der Beschaffung geeigneter Literatur bzw. die vorhandene war nicht immer zufriedenstellend, wenn nicht sogar widersprüchlich.

Leider haben wir den Koprozessor nicht bis zur Produktion bringen können. Vielmehr war unser Zeitlimit schon vor der Emulation mit INCA abgelaufen. Aber wir alle konnten selber erfahren, wie teilweise aufwendig es ist, einen eigenen Chip zu entwickeln, welche Hürden dabei zu überwinden sind und, daß die uns zur Verfügung stehenden Tools in ihrer Funktionaliät, in ihrer Zusammenarbeit und Komfortabilität verbessert werden können(!).

Doch dazu mehr in den folgenden Kapiteln...

Die Projektgruppe bestand aus den folgenden Teilnehmern (in streng alphabetischer Ordnung): Thomas Bellanger, Andreas Henning, Jürgen Hochwald, Edwin Jean-Baptiste, Ralf Jungblut, Thorsten Kukuk, Sven Marcus, Michael Niechziol, Burkhard Reike, Frank Ullmann, Claas Vieler, Lars Werner und Holger Willmer, geführt wurde sie von Christof Nagel unter Prof. F. J. Rammig.

Kapitel 2

Hardware

In diesem Kapitel werden die von uns zur Realisierung des Projekts benötigten Hardware-Ressourcen und ihre Anwendung vorgestellt. Dabei gibt es zwei große Teilbereiche:

- die SPARC Architektur
- den Hardware-Emulator InCA

Die SPARC Architektur besteht aus einem Chipset, das uns leider nur in einem pinkompatiblen Satz vorlag. Im Folgenden wird das ursprüngliche Set vorgestellt. Es gibt nur eine Abweichung, die aber für das Interface keine Beeinträchtigung dastellt.
Der Emulator ist in der Lage ASIC-Bausteine zu emulieren, d.h. indem die FPGAs mit bestimmten Konfigurationen geladen werden, verhält sich das Gerät abgesehen vom Timing, wie der zu emulierende Baustein.

2.1 Die Sparc Architektur

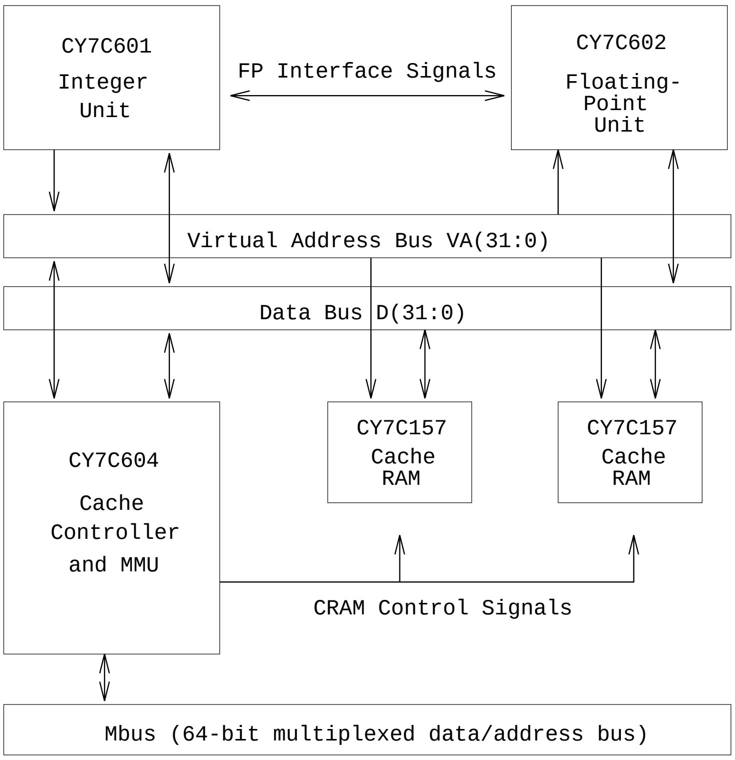

Abbildung 2.1: Übersicht des SPARC

Das Schema 2.1 veranschaulicht die einzelnen Komponenten des SPARC. Die Prozessorkomponenten sind:

- Die CY7C601 Integer Unit ist quasi das Herzstück des Prozessors. Sie enthält den Befehlsinterpreter, der alle Befehle, mit Ausnahme der Floatingpoint-Befehle ausführt, oder zumindest die Bearbeitung veranlaßt. Als weitere Merkmale verfügt die I.-Unit über einen Leistungsdurchsatz von im Mittel 1,25 bis 1,5 Taktzyklen pro Befehl, ein großes Registerfile, das in Fenster eingeteilt ist, User- und Supervisormodus sowie Unterstüzung von Semaphoren. Interrupts werden innerhalb von 4 bis 7 Taktzyklen, je nach Zustand der Pipelines, bearbeitet.
- Der CY7C602 ist der Coprozessor (FPU) zur I.-Unit. Zusammen mit dem Hauptprozessor ergibt sich der gesamte Befehlssatz des SPARC. Der Coprozessor ist über eine extra Befehlsleitung mit der I.-Unit verbunden. Beide Prozessoren können parallel arbeiten, wobei aber der Hauptprozessor die Kontrolle behält. Werden Floatingpoint-Befehle gegeben, so werden sie von der 601 addressiert wobei aber beide Prozessoren auf den Datenbus zugreifen. Der Befehl wird auch von Haupt- und Coprozessor übersetzt, wobei aber die 602 erst mit der Berechnung beginnen kann, wenn sie von der 601 das entsprechende Signal erhält. Die Floatpoint-Befehle werden während ihrer Ausführung in einer Warteschlange auf dem Chip der 602 gespeichert. Wenn das Ergebnis, der Coprozessorberechnung, durch einen Befehl abgefordert wird, die Berechnung aber noch nicht abgescchlossen ist, wird das gesamte System angehalten bis die Ergebnisse vorliegen. Zum Berechnen stehen der FPU 32 32-bit Register zur Verfügung.
- CYC7157 sind Cache-Speicher zur Pufferung von Daten. Der Cache kann sowohl als geregelter Cache (latched), als auch als 'ungeregelter' Cache (write through) eingesetzt werden. Seine Größe beträgt 16K 16-bit Register.
- CYC7604 ist die Memory Management Unit (MMU). Sie verwaltet den Cache-Speicher und bildet den Realspeicher auf den virtuellen Addressraum ab.

2.1.1 Die Integer-Unit 601

Die I.Unit, das Herzstück des Systems, verfügt über zwei Interface(s). Das Eine besteht aus den Steuerleitungen für die FPU. Das andere Interface ist für den Anschluß eines weiteren Coprozessors vorgesehen. Die Protokolle auf den beiden Interfacen sind gleich. Die Bezeichnungen der Signalleitungen unterscheiden sich durch den Anfangsbuchstaben; F steht für FPU und C für Coprozessor. Das Signalschema 2.2 gibt die Leitungen der IU an.

Auf die Signalleitungen zur Speicherverwaltung wird hier nicht im einzelnen eingegangen, da sie im Seminarband aufgeführt sind. Die Signalleitungen zwischen IU und FPU bzw. CO-Prozessor sind keine externen Leitungen. Es handelt sich dabei um quasi interne Leitungen, die zu den Sockeln der CO-Prozessoren führen. Die Schnittstelle(n) sind in ihrem Protokoll und ihrer Arbeitsweise identisch. Es gibt folgende Signale:

- CCC(1:0) (Coprocessor Conditions Codes) gibt Ergebnisse von Vergleichen an die IU weiter.
- CCCV (Coprocessor Conditions Codes Valid) dient erstens zur Synchronisation von IU und CO. Es wird gesetzt, wenn die CCC Leitungen gülitige Werte angenommen haben.
- CXEC (Coprocessor eXCEption) teilt der IU mit, falls es bei der FPU / CO zu Ausnahmen in der Ausführung kommt.
- CHOLD (Coprocessor HOLD) meldet ein Anhalten des CO, was auch ein Halten der IU zur Folge hat. Fortgesetzt wird erst, wenn der CO den Fehler (meistens Zugriffsverzögerung) behoben hat.
- CINS1 (Coprocessor INStruction buffer 1) transportiert das Kontrollsignal an den jeweiligen Coprozessor, so daß ein gültiger Befehl im Buffer 1 des Befehlsregisters steht. CINS2 dient dem gleichen Zweck. Mit dem dort stehenden Befehl wird gleich (im selben Zyklus) die Auswertung begonnen. Es werden nie beide Leitungen im gleichen Zyklusdurchlauf auf HIGH gelegt.
- CP (Coprocessor Present) gibt an, ob ein CO vorhanden ist.

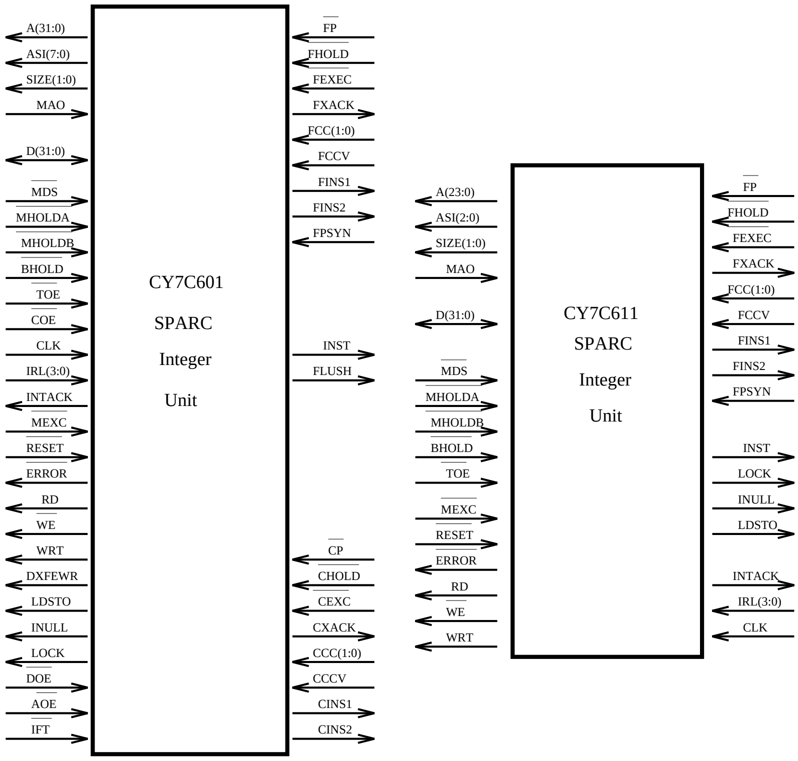

Abbildung 2.2: Übersicht der externen Signale bei 601/611

- CXACK (Coprocessor EXception ACKnowledge) meldet dem CO, daß ein Ausnahmezustand bei dem CO erkannt ist. Dieses Signal hat ein Halten der CO - Pipeline zur Folge.
- FCC(1:0) (Floatingpoint Conditions Codes) gibt Ergebnisse von Vergleichen an die IU weiter.

FCC(1)	CFF(0)	CONDTION
0	0	gleich
0	1	$Op1 < Op2$
1	0	$Op1 > Op2$
1	1	ungeordnet

- FCCV (Floatingpoint Conditions Codes Valid) dient zur Synchronisation von IU und FPU. Es wird gesetzt, wenn die FCC Leitungen gültige Wert angenommen haben.
- FXEC (Floatingpoint eXCEption) teilt der IU mit, falls es bei der FPU zu Ausnahmen in der Ausführung kommt.
- FHOLD (Floatingpoint HOLD) meldet ein Anhalten der FPU, was auch ein Halten der IU zur Folge hat. Fortgesetzt wird erst, wenn die FPU den Fehler (meistens Zugriffsverzögerung) behoben hat.
- FINS1 (Floatingpoint INStruction buffer 1) transportiert das Kontrollsignal an den jeweiligen Coprozessor, so daß ein gültiger Befehl im Buffer 1 des Befehlsregisters steht. CINS2 dient dem gleichen Zweck. Mit dem dort stehenden Befehl wird gleich (im selben Zyklus) die Auswertung begonnen. Es werden nie beide Leitungen im gleichen Zyklusdurchlauf auf HIGH gelegt.
- FP (Floatingpoint Present) gibt an, ob eine FPU vorhanden ist.
- FXACK (Floatingpoint EXception ACKnowledge) meldet der FPU, daß ein Ausnahmezustand bei der FPU erkannt ist. Dieses Signal hat ein Halten der FPU - Pipeline zur Folge.
- FLUSH (floatingpoint / coprocessor instruction FLUSH) veranlaßt die FPU / CO ihren aktuellen Befehl in den Instruction-Buffer zu sichern. Das Signal wird gesetzt, wenn die IU ein Trap-Signal erhält.

- INST (INSTruction fetch) signalisiert der FPU / dem CO, daß ein neuer Befehl 'geholt' wird. Zu diesem Zeitpunkt wird der neue Befehl in eine der Instruction-Buffer geladen.

Zur Verwaltung von Ausnahmezuständen verfügt das gesamte System über ein sogenanntes Exception–Modell. Dieses greift bei Interrupts, die den normalen Rechenbetrieb stören (z.B. Reagieren auf externe Einflüsse). Auch Fehler in Berechnungen können zu Exceptions führen. Gerade diese sind interessant, wenn man das Interfaceprotokoll zwischen der IU und den Coprozessoren betrachtet. Kommt es bei der Berechnung innerhalb der FPU zu Fehlern, sei es durch die Operanden, z.B. Teilen durch Null, oder weil es gewünscht ist (Fehler bei Unvergleichbarkeit), so setzt die FPU das FEXC-Signal. Die IU erkennt dann den 'Hilferuf' der FPU und setzt dann das FACK-Signal, welches der FPU signalisiert, daß der Fehlerfall bemerkt worden ist. Gleichzeitig wird noch das FLUSH-Signal mit gesetzt. Dieses veranlaßt die FPU ihre beiden Adress- und Instructiondecode-Register zu löschen. Sind die Register gelöscht, so werden die Befehle abgearbeitet, die vor dem Fehlerfall in Bearbeitung waren. Da kurz nach dem Fehler noch zu bearbeitende Befehle in die Warteschlangen geschrieben werden, können sie dann bearbeitet werden. Die folgende Skizze veranschaulicht die drei Zustände, die das Exception-Modell hat.

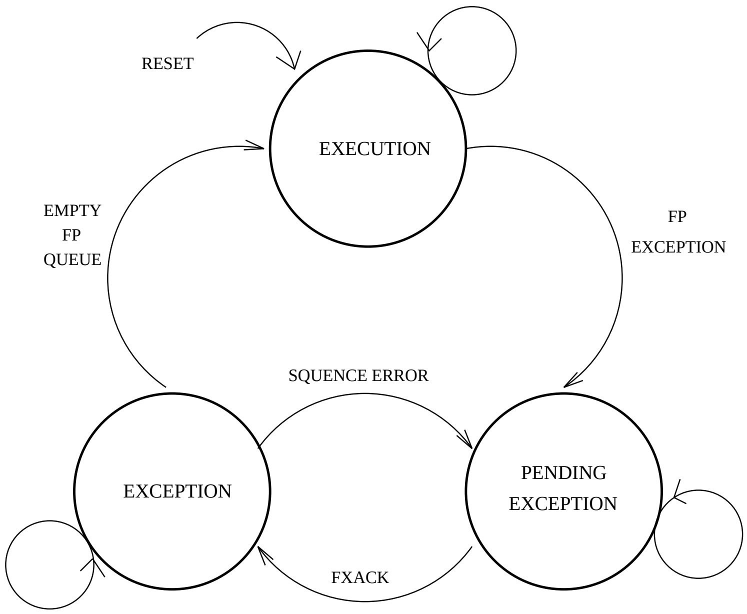

Abbildung 2.3: Exception-Modell

2.1.2 FPU und Coporzessor

Da die Schnittstellen von FPU und Coprozessor identisch sind, kann man die FPU gut als Beispiel für einen Coprozessor auffassen. Durch das strenge Protokoll der IU ergeben sich bestimmte Anforderungen, die ein an das Interface angeschlossener Coprozessor erfüllen muß. Um den Anschluß eines selbstdefinierten Coprozessors zu ermöglichen, stellt das Prozessorsystem einen zusätzlichen Sockel bereit. (Dieser ist leider aus Kostengründen auf keinem realexistierenden Rechnerboard installiert.) Um diesen zu nutzen, ist es jedoch notwendig, daß dieser bestimmte Kriterien erfüllt. Die Signalleitungen sind oben näher beschrieben. Das Protokoll dieser Leitungen ist mit dem der FPU identisch. Das genaue Protokoll wird im folgenden Kapitel beschrieben. Es gibt im Wesentlichen vier Anforderungen, die an einen selbstdefinierten Coprozessor gestellt werden:

- das Registermodell muß eingehalten werden(s.u.)
- es muß ein Statusregister geben
- es müssen Befehlswarteschlangen vorhanden sein
- es muß pinkompatibel sein (logisch)

Da die IU für die eigentliche Steuerung verantwortlich ist, kann ein Coprozessor selbst keine Daten adressieren, d.h. alle Adressierungen laufen über die IU. Hierzu ist es notwendig, daß der Coprozessor auf den gleichen Bus zugreift wie die IU. Die Coprozessoren arbeiten wie die IU nur auf Registern. Hierzu hat ein CO ein 32 x 32 Bit breites Registerfile. Alle CO-Befehle arbeiten auf diesen Registern. Sollen also Daten in die Register geladen oder aus den Registern geschrieben werden, so geschieht dieses über den Bus, auf den vorher, von der IU veranlaßt, die Daten gelegt wurden. Auch das Statusregister des CO kann nur auf diese Weise ausgelesen werden. Ein Coprozessor muß auch über zwei Address- und Instruction-Decode-Register verfügen. Alle Befehle werden simultan zur IU gelesen und interpretiert, folglich ist die Decode- Einheit identisch. Das CFlush-Signal, das von der IU gesetzt werden kann löscht diese beiden Register, läßt aber die Befehlswarteschlange unberührt. In die Warteschlangen werden Befehle eingelagert, die aufgrund von Ausnahmen nicht sofort bearbeitet werden können. (Näheres hierzu in der Beschreibung des Exceptionmodells.) Die beiden Warteschlangen werden nach dem FIFO - Prinzip verwaltet.
Will man aus z.B. Platzgründen die Anzahl der Register kleiner halten, so muß man entweder sicherstellen, daß nur die existierenden Register angesprochen werden, oder man muß in dem steuernden Teil (Pipeline) dafür sorgen, daß sinnvolle Werte zurück geliefert werden. Auch nicht benötigte Signalleitungen sollten zur Sicherheit mit Dummywerten belegt werden.
Um ein konkretes Beispiel für die Benutzung des Interfaces zu geben sei im Folgenden die Standard - FPU näher beschrieben. Die FPU 602 ist der Standard - Coprozessor zur Sparc IU. Er verfügt über eine 64-Bit ALU (Arithmetic and Logical Unit), mit deren Hilfe 64-Bit Additionen, Subtraktioen, Multipikationen, Divisoinen und Wurzelziehen möglich ist. Zu den 64-Bit Operationen müssen allerdings zwei 32-Bit Register zusammengenommen werden. Die FPU verfügt, gemäß den Anforderungen an einen CO, über 32 dieser 32-Bit Register. Verarbeitet werden Gleitkommazahlen gemäß des ANSI/IEEE-754 Standards. Wird die FPU an 40 MHz angeschlossen, so kommt man auf einen Leistung von 6,15 MFLOPS (Million-FloatingPoint-OperationsPerSecond) bei Verwendung von double-precision Zahlen.

2.1.3 Die Schnittstelle der FPU

Die FPU hat ein Interface, das genau der Spezifikation für Coprozessoren folgt. Alle Signalleitungen, die zur IU führen, sind oben beschrieben. Zusätzlich bestehen noch Verbindungen zur MMU 604 / 605:

- DOE (input, Data Output Enable) wird gesetzt, wenn die Daten wieder auf den Bus gelegt werden dürfen.
- MHOLD A/B (input, Memory HOLD) wird gesetzt, wenn die Pipeline angehalten werden muß, weil ein Cachemiss aufgetreten ist. (analog zu den HOLD A/B Leitungen der IU)
- BHOLD (input, Bus HOLD) ist gesetzt, wenn ein externer Busmaster den Bus kontrolliert. Während dieser Zeit ist die Pipeline angehalten.
- MDS (input, Memory Data Strobe) wird gesetzt um Daten in die FPU zu laden, wenn die FPU-Pipline angehalten ist. (MHOLD A/B oder BHOLD)
- FNULL (output, Fpu NULLify cycle) hat die gleiche Funktion wie das INULL-Signal.
- RESET (input, RESET) wird gesetzt um die FPU zu (re)initialisieren. Das Signal bewirkt ein Setzen aller schreibbaren Elementen auf Null. Das Signal muß mindesten 8 Taktzyklen anliegen.
- CLK (input, CLocK) ist das Taktsignal, das die FPU steuert.

Ansonsten bestehen noch Leitungen zu dem Adressbus A(0:31) und zum Datenbus D(0:31). Das folgende Schema 2.4 stellt die FPU mit den Signalleitungen dar.

Die FPU läßt sich in 9 funktionale Teilblöcke teilen, von denen jeder eine bestimmte Aufgabe erfüllt.

1. die Fetch Unit greift Daten und Adressen von den Bussen ab.
2. die Load Unit übernimmt die Daten und Adressen von der Fetch Unit und 'merkt' sich diese, bis sie in ein Register gespeichert werden.
3. die Decode Unit entschlüsselt die Befehlscodes, die von Bus gelesen werden.

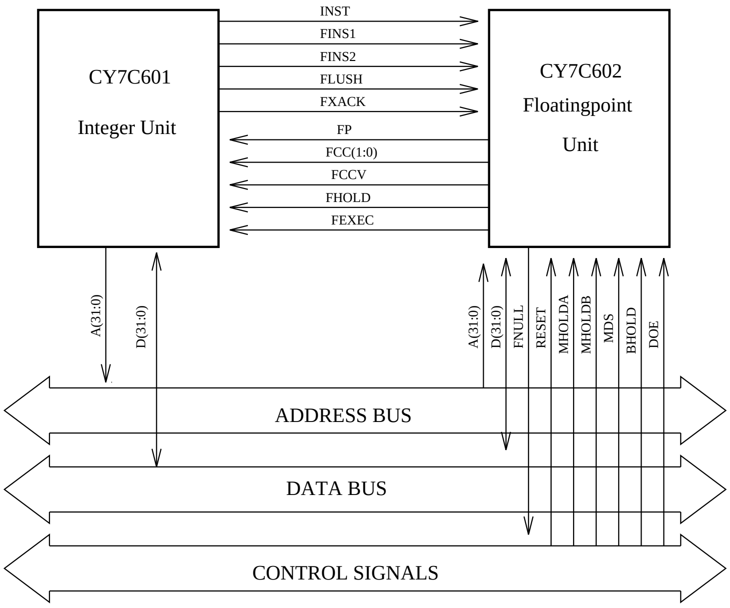

Abbildung 2.4: Interface der FPU

4. die Execute Unit enthält die Befehls- und Adressqueue. Sie führt die Befehle aus, indem von hieraus die entsprechenden Multiplexer und die ALU gesteuert werden.
5. die Store Unit 'hält' Daten, die auf den Bus zurück geschrieben werden sollen bis der Bus frei ist und die Daten übertragen werden können.
6. die Exception Unit regelt das Fehlerprotokoll mit der IU, falls sie von der Execute Unit dazu veranlaßt wird.
7. das 32-Bit Registerfeld stellt 32 32-Bit Register zur Verfügung.
8. die ALU führt die eigentlichen Berechnungen durch.
9. die Dependency checking Unit prüft ob ein Nullzyklus vorliegt, ob also Daten für einen Berechnungszyklus fehlen. Im Bedarfsfall wird dann das FNULL-Signal gesetzt.

Das folgende Übersichtsschema 2.5 stellt die Verbindungen der einzelnen Komponenten untereinander und mit den anderen Prozessorteilen dar.

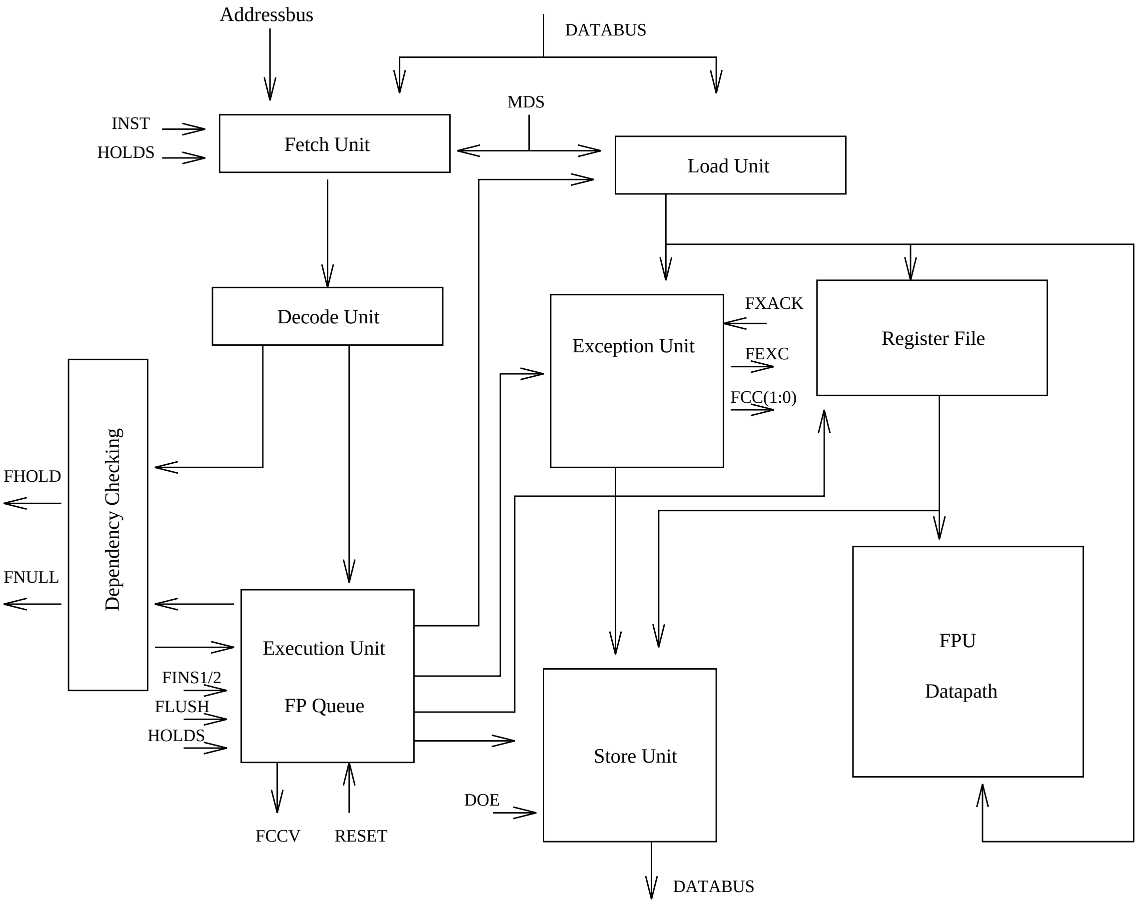

Abbildung 2.5: Blockdiagramm der FPU

Die FPU benutzt eine vierstufige Pipeline. Die Zustände sind fetch, decode, execute und write (F, D, E, W). Da die beiden Prozessoren des SPARC kooperativ arbeiten und es Befehle bei den Floatbefehlen gibt, die länger dauern als einen Bearbeitungszyklus, verweilt die FPU im W-State, falls mehrere Zyklen erforderlich sind. Aufgrund der LOAD / STORE - Architektur gibt es drei Arten von Befehlen, die hier unterschieden werden können.

- LOAD Befehle
- STORE Befehle
- Float Befehle

Die folgenden Tabellen stellen die Aktionen dar, die während dieser Zyklen ablaufen.

Ausführen eines LOAD-Befehls

Zyklus	Handlung
D-Zustand	Decodieren und Prüfen, ob Abhängigkeiten bestehen
E-Zustand	FHOLD setzen falls nötig
W-Zustand	Abgreifen der Daten vom Bus
Wh1-Zustand	Setzen des FSR
Wh2-Zustand	Schreiben der Daten in die Register

Ausführen eines STORE-Befehls

Zyklus	Handlung
D-Zustand	Decodieren und Prüfen ob Abhängigkeiten bestehen
E-Zustand	FHOLD setzen falls nötig, Daten aus Registern lesen
W-Zustand	Daten auf den Bus legen
Wh1-Zustand	Daten vom Bus nehmen
Wh2-Zustand	Bus freigeben

Ausführen eines FLOAT-Befehls

Zyklus	Handlung
D-Zustand	Übersetzen des Befehls, prüfen ob Resourcen verfügbar
E-Zustand	FHOLD setzen falls nötig, Daten aus Registern lesen
W-Zustand	Lesen noch benötigter Daten, Berechnen
FP Queue	Berechnen der FOPs in der Schlange
.	.
.	.
.	.
FP Queue	Prüfe auf Sonderfälle
FP Queue	Aktualisiere FSR

Aus dem Beispiel werden die Anforderungen, die an unseren Spline-Coprozessor gestellt werden, nun sichtbar. Die einzelne Realisierung wird in Kapitel 5 beschrieben.

2.1.4 Anschluß eines Coprozessors

Um den Coprozessor an den Prozessor anzuschliessen gibt es nun verschiedene Möglichkeiten:

2.1.4.1 Alternative Lösungsansätze

Als erste Möglichkeit (und zugleich die einfachste) kann man die Emulatorverbindung in den Sockel auf dem Board stecken. Leider ist es uns nicht möglich gewesen ein solches Board zu beschaffen. Verfügt die IU. zwar über die Signalleitungen des Interfaces, hat aber keinen extra Sockel für einen Coprozessor, so können die Pins dieser Steuerleitungen über einen speziellen Sockel ausgefüht werden. Diese Variante ermöglicht es ohne Änderungen am Betriebssystem die Steuerung des Coprozessors durchzuführen. Die vom SPARC-System vorgesehene Befehlslücke können hierfür dann benutzt werden. (Dieser Ansatz wurde von uns verfolgt.)

2.1.4.2 Entfernen der FPU

Als nächste Alternative kann man die FPU entfernen und statt dessen den Coprozessor anschliessen. Das ist im Prinzip von der Seite der Hardware kein Problem nur das Betriebsystem muß dann erstens alle Befehle, die für die FPU sind softwaremäßig emulieren. Zweitens muß zwischen FPU und Coprozessorbefehl unterschieden werden. Das hat ein Verändern des Betriebssystems zur Folge (siehe Änderung des BS bei den jeweiligen Lösungsansätzen).

2.1.4.3 Huckepack-Lösung

Die dritte von uns erwogene Lösung ist quasi eine "Huckepack"-Lösung des Problems. Das Entfernen der FPU führt zu erheblichen Leistungseinbußen des Gesamtsystems und ist damit nicht mehr vergleichsfähig mit einer reinen Softwarelösung. Die Idee besteht nun darin eine Lücke im Befehlssatz der FPU zu nutzen und hierin die Coprozessorbefehle zu codieren. Da beide Coprozessoren (FPU und SFU) auf den selben Leitungen arbeiten, muß nun ein Weg gefunden werden, wie zwischen den Coprozessoren umgeschaltet werden kann. Dieser Weg ist das FINSi Signal. Ein Coprozessor versucht nur dann einen Befehl zu interpretieren, wenn er das FINSi Signal erhält. Dieses wird in der SFU (dem Coprozessor) entsprechend ausgewertet und dann an die FPU weiter geleitet. Das Auswerten geschieht in einer neuen Pipelinestufe, dem FETCH (das Holen von Daten entfällt normalerweise, da die IU die Daten aus dem Speicher holt). In der FETCH-Stufe soll jeder auf dem Datenbus liegende

Befehl untersucht werden und dann wird das entsprechende FINSi gesetzt oder nicht gesetzt. Damit kann man steuern welcher Coprozessor nun arbeiten soll. Das Schaltbild zeigt die genaue Verdrahtung der Bausteine. Die Gatter sollen extern geschaltet werden, da der Emulator zu langsam ist.

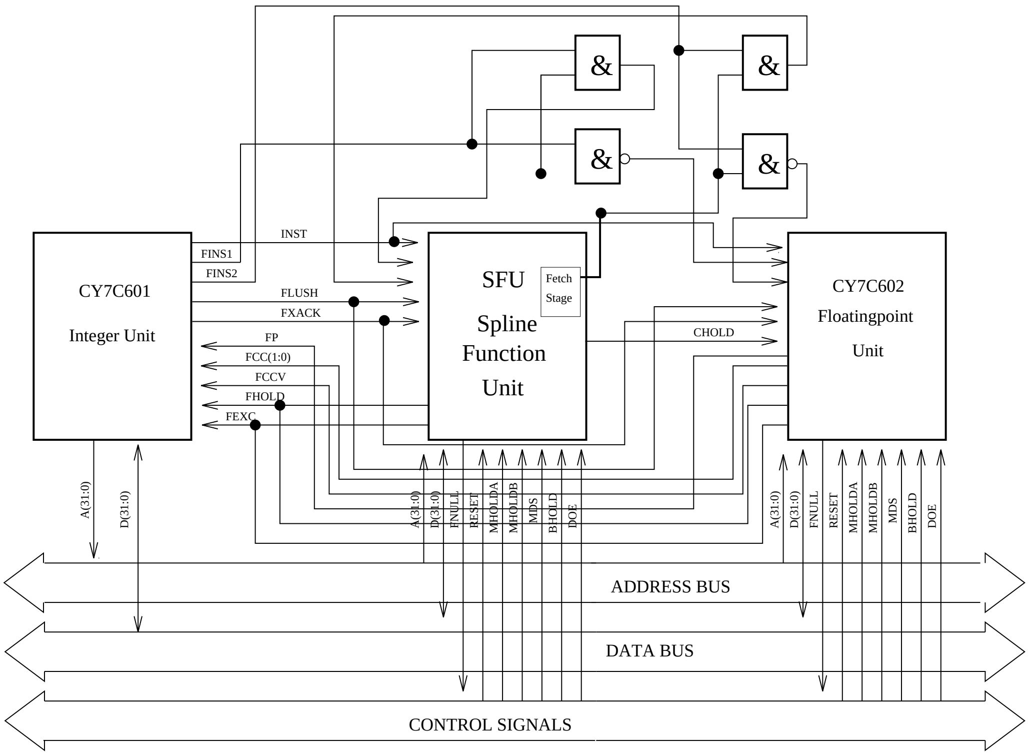

Abbildung 2.6: Schaltbild zur Verdrahtung

Die Konsequenzen für das Betiebssystem sind unter Änderung des BS bei den jeweiligen Lösungsansätzen erläutert.

2.2 Die Emulator-Hardware

Der InCA Hardwareemulator basiert auf benutzerprogrammierbaren FPGAs der Firma Xilinx. FPGAs (Field Programmable Gate Arrays) sind programmierbare CMOS-Bausteine, die aus mehreren tausend Gatter berstehen. Durch die hohe Zahl an Gattern ist es möglich Verbindungen zwischen den Logikblöcken auf den Chips herzustellen. Auf diese Weise lassen sich dann Schaltungen in dem Chip abbilden. (Details stehen im Seminarband Kapitel FPGA / InCA)
Der Hardwareemulator besteht aus Chips der 3000er und 4000er Serie der Firma Xilinx. Der gesamte Emulator (bezeichnet als Kabinett) besteht aus zwei Karten, die über die Backplane verbunden werden können. Die Karten bestehen aus je 8 DB (Daugther-Boards), die durch 18 Crossbars verbunden sind. Die DBs bestehen aus je 2 Xilinx4010 Bausteinen. Ein Crossbar ist durch einen Xilinx3090 realisiert. Die Schemadarstellung 2.7 zeigt den Aufbau einer Karte.

Laut Hersteller haben die Bausteine dann folgende Leistungen:

Xilinx4010	3.500 Gatter
Xilinx3090	1.500 Gatter
DB	7.000 Gatter
Karte	30.000 Gatter
Kabinett	60.000 Gatter

Karte

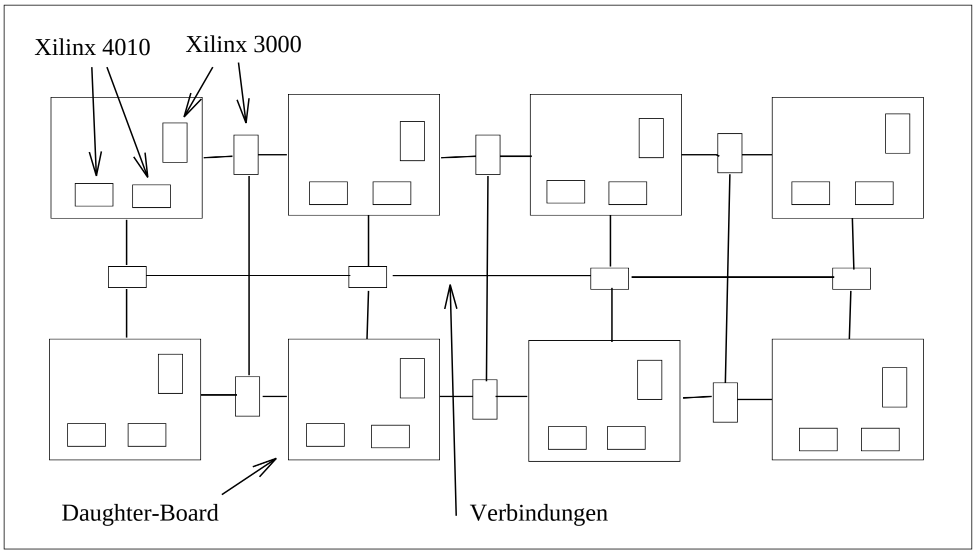

Abbildung 2.7: eine Karte des Kabinetts

Damit ergibt sich fuer die einzelnen Betrachtungsebenen:

Komponente	reale Kapazität	Erfahrungswert	Summer der Erfahrungswerte
XILINX4010	? = X1	3.500	3.500
XILINX3090	? = X2	1.500	1.500
DB	2 * X1	7.000	7.000
Karte	8*(2*X1)+18*X2	30.000	83.000
Kabinett	2 * Karte	60.000	166.000

Die einzelnen technischen Daten sind in der folgenden Tabelle aufgelistet:

Eigenschaft		XC3000	XC4000
Anzahl der Gatter		k.A.	10.000
Logik-Block-Matrix		16 x 20	20 x 20
Anzahl der Logik-Blöcke		320	400
Anzahl der Flip-Flops		928	1.120
Maximaler Speicher		0	12.800
Anzahl I/O-Bits		144	160
Funktionsgeneratoren		2	3
Anzahl Inputs pro Logikzelle		5	9
Anzahl Outputs pro Logikzelle		2	4

Programmiert werden die Karten über ein PC-Interface. Die InCA-Software erzeugt eine genaue Belegung der einzelnen Chips. Diese Belegung wird dann in den Emulator geladen. (Achtung !!! die Belegung verbleibt bis zum Ausschalten des Emulators in den Schaltungen, es muß also neu geladen werden, wenn sich die Belegung der FPGAs ändert.) Die Schaltung kann dann zunächst über ein Tester-Programm geprüft werden. Ist dieser Test zufriedenstellend, kann man die Ein- und Ausgänge auf die Pins der Backplane legen und die Schaltung real testen.

2.3 Software

2.3.1 Betriebssystem

2.3.1.1 Aufgaben eines Unix–BS beim Betrieb eines Coprozessors

Die Ansteuerung des Coprozessors erfolgt direkt durch Maschineninstruktionen. Das Betriebssystem (BS) wird im Zusammenhang mit der Floatingpoint–Unit (FPU) oder einem Coprozessor immer nur dann benötigt, wenn eines der folgenden Ereignisse auftritt:

- FPU–Exception
- Coprozessor–Exception
- Start eines Prozesses
- Exception beim Auftauchen des ersten FPU– oder Coprozessor–Befehls innerhalb eines Prozesses
- Contextswitch

Siehe hierzu auch [SunTech], S. 24f, 86ff.

SunOs unterstützt aber im Zusammenhang mit diesen Ereignissen nur die FPU. Ein Coprozessor wird von *SunOs* nur insofern "unterstützt", als für jeden Prozeß der Coprozessor ausgeschaltet wird, indem immer beim Start eines Prozesses das Coprozessor–Enabled–Bit im IU–Statusregister auf 0 gesetzt wird.

SunOs behandelt FPU–bezogene Ereignisse folgendermaßen:

FPU–Exception: Im Falle einer FPU–Exception wird ein Trap ausgelöst, das das BS entweder dazu verwendet, den Benutzer über den Fehler zu informieren (falls der Fehler nicht behebbar ist), oder aber den Fehler selbständig zu beheben (zum Beispiel in dem Fall, in welchem keine FPU vorhanden ist, so daß bei jeder FP–Instruktion, die die IU liest, ein Trap ausgelöst wird, den das BS dazu nutzt, die Instruktion softwaremäßig zu simulieren.).

Start eines Prozesses: Beim Start eines Prozesses setzt das BS das FPU–Enable–Bit im Prozessorstatuswort der IU auf 0, um die FPU auszuschalten.

Das Bit muß nicht nach jedem Contextswitch explizit wieder auf 0 gesetzt werden, da das BS bei einem Contextswitch sowieso den gesamten aktuellen Inhalt des IU–Statusregisters sichert und später bei der Wiederaufnahme des unterbrochenen Prozesses den gesicherten Wert wieder ins IU–Statusregister schreibt.

Auftauchen der ersten FP–Instruktion: Trifft die IU innerhalb eines Prozesses zum ersten Mal auf eine FP–Instruktion, kann diese nicht ausgeführt werden, da ja die FPU seit dem Start des Prozesses ausgeschaltet ist. Die IU reagiert folglich durch das Auslösen eines Traps, welches das BS dann dazu benutzt, den aktuellen Prozeß als FPU–benutzend zu markieren, damit es bei jedem folgenden Contextswitch merkt, daß nicht nur die IU–Register, sondern auch die Register der FPU gesichert werden müssen. Außerdem initialisiert das BS in diesem Trap die Register der FPU und schaltet die FPU durch Setzen des FPU–Enable–Bits auf 1 im IU–Statusregister an.

Contextswitch: Beim Contextswitch sichert das BS die Register (inklusive Statusregister) von IU und FPU (bei der FPU aber nur, falls der Prozeß als FPU–benutzend markiert ist).

Wenn nun ein Coprozessor in ähnlicher Weise durch *SunOs* unterstützt werden soll, müssen die Kernelroutinen geändert werden. Als Orientierung kann dabei die Behandlung der FPU durch *SunOs* dienen. Die Form der Lösung hängt davon ab, ob überhaupt eine FPU vorhanden ist, ob FPU und Coprozessor trickreich in Reihe geschaltet sind ("Huckepacklösung") oder ob FPU und Coprozessor unabhängig voneinander (so, wie es ja auch eigentlich sein sollte) vorhanden sind.

Die ersten beiden dieser drei Lösungsalternativen kamen in der Anfangsphase des Projektes in Betracht, als noch offen war, ob wir überhaupt eine *SPARC* mit einer *CY7C*601 Integer–Unit (die sowohl FPU als auch Coprozessor unterstützt) finden würden, oder ob wir uns mit einer *CY7C*611 Integer–Unit (die nur die FPU unterstützt) zufriedengeben mußten.

Zum Glück bekamen wir eine *CY7C601*. Änderungen am BS blieben weitgehend aus, weil weder Exceptions (Fehler sollten durch die steuernde Software abgefangen werden, siehe Abschnitt 10.1) noch Contextswitchs (weil der Einfachheit halber immer nur ein Prozeß den Coprozessor benutzen darf) vorgesehen waren.

ACHTUNG! Änderungen an *SunOs* sind sowieso sehr problematisch, weil der Fachbereich 17 keine Lizenzen für die *SunOs*–Sourcen besitzt! [1] Sollten also für ein zukünftiges Projekt an der UniGH Paderborn Änderungen am BS notwendig sein, sollte man rechtzeitig in Erwägung ziehen, ein BS zu benutzen, dessen Sourcen verfügbar sind (Beispiele für frei erhältliche Unix–BSe: *FreeBSD* oder *Linux*).

Obwohl keine Änderungen an *SunOs* vorgesehen waren, war doch gegen Ende des Projektes eine elementare Änderung notwendig, da *SunOs* das Bit im Prozessorstatusregister, daß den Coprozessor einschaltet, nicht auf 1 setzt und auch sonst nicht dazu zu bewegen ist, dies zu tun. Da die Sourcen nicht vorhanden waren, wurde die Änderung direkt im *vmunix*-Executable durchgeführt, siehe Abschnitt 9.2.2.

Obwohl wir die richtige IU (601, mit Coprozessor–Unterstützung) bekommen haben, und obwohl keine umfangreichen Änderungen an *SunOs* durchgeführt wurden, soll hier vorgestellt werden, was man an *SunOs* ändern müßte, wenn man eine 611–IU ohne Coprozessor–Unterstützung hätte:

2.3.1.2 Änderungen bei Entfernung der FPU

Falls überhaupt keine FPU vorhanden ist und der Coprozessor im FPU–Sockel steckt, müssen FP–Befehle emuliert werden. Dieses funktioniert so:
Der Coprozessor tut so, als wäre er eine FPU, daß heißt, er signalisert der IU über den entsprechenden Pin, daß eine FPU vorhanden ist.

Wenn eine FP–Instruktion ausgeführt werden soll, muß der Coprozessor in der Lage sein, diese als "echte" FP–Instruktion zu erkennen. Ist eine "echte" FP–Instruktion erkannt, löst der Coprozessor ein "fp-instruction-not-implementet" Trap aus, der entsprechende BS-Traphandler holt sich die Instruktion und die Operanden vom Coprozessor zurück und emuliert den Befehl (die Emulation von FP–Befehlen ist im normalen *SunOs* bereits implementiert, man muß dafür nur obiges Trap auslösen).

Bei dieser Lösung sehen Spline–Instruktionen zwar für die IU wie FP–Instruktionen aus (daß heißt, sie sind in bestimmten Bitfeldern als solche gekennzeichnet), aber die Codierung der Instruktions-Mnemos ist eine andere als die der normalen FP–Instruktionen. Die IU wird durch diese fremden Codes nicht irritiert, weil sie die Decodierung von FP–Instruktionen der FPU überläßt. Die FPU aber ist ja in Wirklichkeit unser Coprozessor, und der erkennt so eine Nicht–Standard–FP–Instruktion (sollte er jedenfalls) und führt die entsprechenden Spline–Berechnungen aus, ohne dabei ein Trap auszulösen.

Was aus der Sicht des BS zu beachten ist:

Exceptions: Traphandler für FP–Exceptions bleiben unverändert. Coprozessor–Exceptions darf es nicht geben, weil die 611–IU nur FP–Exceptions kennt und folglich diese von Coprozessor–Exceptions nicht unterscheiden könnte. Der Coprozessor löst zwar "fp-instruction-not-implementet"–Exceptions aus, aber das ist keine Exception, die etwas mit den eigentlichen Aufgaben des Coprozessors zu tun hat. Sie dient lediglich dazu, FP–Befehle vom BS emulieren zu lassen, da für diese ja keine FPU mehr vorhanden ist.

Start eines Prozesses: Beim Start eines Prozesses muß das FPU–Enabled–Bit im IU–Statusregister auf 0 gesetzt werden. Da *SunOs* dieses bereits tut, ist auch hier keine Änderung notwendig.

FPU–Disabled–Trap: Trifft die IU innerhalb eines Prozesses auf die erste FP–Instruktion, wird ein Trap ausgelöst, weil das FPU–Enabled–Bit im IU–Statusregister auf 0 gesetzt ist. In dem zugehörigen Traphandler markiert *SunOs* den Prozeß als FPU–benutzend, initialisiert die FPU–Register und schaltet die FPU durch Setzen des FPU–Enabled–Bits auf 1 an.

Die vermeintliche FPU ist hierbei nun in Wirklichkeit unser Coprozessor, aber solange dieser sich so verhält wie eine FPU, spielt das für das BS keine Rolle.

[1] Nach Auskunft von Michael Kutzner benötigt man für den Erwerb der *SunOs*-Sourcen a) 5 Unterschriften und b) viel Geld.

Von den Aktionen, die das Trap nach sich zieht, muß höchstens die Initialisierung der Register angepaßt werden, und zwar dann, wenn der Coprozessor nicht genau 32 Register besitzt oder die Register anders initialisiert werden sollen, als dies bei der FPU getan wird.

Contextswitch: Ist ein Prozeß als FPU–benutzend markiert, und wird dieser Prozeß durch einen Contextswitch unterbrochen, so müssen innerhalb dieses Contextswitches die Coprozessor–-Register gesichert werden (FPU–Register gibt es nicht).

Sobald alle Register gesichert sind, werden in die Register die Werte geladen, die zu dem Prozeß, dem als nächstes Rechenzeit zur Verfügung gestellt wird, gehören.

Die Coprozessor–Register werden durch genau diegleichen Befehle gelesen und beschrieben, die normalerweise für die FPU benutzt werden.

All dies wird bereits vom normalen *SunOs* erledigt. Eine Anpassung ist nur dann notwendig, wenn der Coprozessor nicht 32 Register oder kein Statusregister besitzt.

Die Lösung, die FPU zu entfernen, hat aber den gravierenden Nachteil, daß FP–Befehle durch die Softwareemulation wesentlich langsamer ausgeführt werden.

2.3.1.3 Änderungen für die "Huckepack"–Lösung

Diese Alternative, die während der Anfangsphase des Projektes entwickelt wurde, ermöglicht es, sowohl die FPU als auch Coprozessor zu benutzen, obwohl bei der 611-IU der Betrieb eines Coprozessors nicht vorgesehen ist. Bei dieser Lösung werden Coprozessor und FPU in Reihe geschaltet, der Coprozessor fungiert dabei als Interface für die FPU und leitet FP–Instruktionen an die FPU weiter. Für Coprozessor–Instruktionen wird eine Lücke im Befehlssatz der FPU genutzt.

Was aus der Sicht des BS zu beachten ist:

Exceptions: Coprozessor–Exceptions sind nicht möglich, da die 611–IU nur FP–Exceptions annehmen kann. Die Traphandler für Exceptions bleiben also unverändert.

Start eines Prozesses: Auch beim Start eines Prozesses muß nichts an den alten BS–Routinen geändert werden. Denn beim Start eines Prozesses wird nur das FPU–Enabled–Bit im IU–Statusregister auf 0 gesetzt. Ein Coprozessor–Enabled–Bit gibt es nicht. Das FPU–Enabled–Bit ist bei dieser Lösung sowohl für die FPU als auch den Coprozessor zuständig, es wird also zu einer Art FPU–and–Coprozessor–Enabled–Bit.

FPU–Disabled–Trap: Trifft die IU innerhalb eines Prozesses auf die erste FP–Instruktion (die auch eine "verkappte" Coprozessor–Instruktion sein kann), markiert das BS den Prozeß als FPU–benutzend (und damit auch implizit als Coprozessor–benutzend, denn entweder sind beide eingeschaltet oder beide ausgeschaltet), setzt das FPU–Enabled–Bit im IU–Statusregister auf 1 und initialisiert die Register von FPU und Coprozessor.

Von diesen Aktionen werden alle bis auf die Initialisierung der Coprozessor–Register bereits von *SunOs* durchgeführt, man muß also nur noch die Initialisierung des Coprozessors in den Kernelroutinen hinzufügen.

Contextswitch: Ist ein Prozeß als FPU–benutzend (und damit auch als Coprozessor–benutzend) markiert und wird der Prozeß durch einen Contextswitch unterbrochen, so müssen die Register von FPU und Coprozessor gesichert und mit Werten, die zu dem nächsten Prozeß gehören, beschrieben werden.

Was die FPU betrifft, muß *SunOs* also nicht geändert werden.

Es müssen jetzt aber noch Stellen in die Kernelroutinen eingebaut werden, die in ähnlicher Weise die Register des Coprozessors sichern und mit den Werten des nächsten Prozesses laden.

Kapitel 3

Entwicklungs-Tools

In diesem Kapitel sollen die wichtigsten, während der Projektgruppenarbeit verwendeten Tools kurz vorgestellt werden.

3.1 SPeeDCHART

Bei SPeeDCHART handelt es sich um ein Hilfs-Tool für die Entwicklung von Hardware-Komponenten. Die Grundidee hinter SPeeDCHART ist, daß eine Komponente zunächst abstrakt als Zustandsgraph entwickelt wird, welcher dann automatisch in eine Hardware-Beschreibungssprache übersetzt wird.

Zur Eingabe des Zustandsgraphen wird eine graphische Oberfläche zur Verfügung gestellt. Es besteht die Möglichkeit, Zustände und Zustandsübergänge einzugeben. Bedingungen für einen Zustandsübergang, sowie Aktionen in einem Zustand bzw. bei einem Übergang, können wahlweise in VHDL- oder C-ähnlicher Sprache angegeben werden. Die für Aktionen möglichen Befehle haben annähernd die Mächtigkeit einer höheren Programmiersprache, allerdings können Prozeduren nicht in einem Zustand angegeben werden, sondern müssen global angegeben werden. Jenachdem welche Hardware-Beschreibungssprache man für die Ausgabe gewählt hat, sind allerdings nicht alle Befehle (insbesondere Prozeduren) übersetzbar.
Zur Erhöhung der Übersichtlichkeit ist es möglich, Zustände hierarchisch zu gestalten. Dies bedeutet, daß sich hinter einem Zustand in einer höheren Ebene ein komplexer Zustandsgraph in einer tieferen Ebene verbirgt. Jeder einzelne Zustandsgraph hat hierbei einen klar definierten Startzustand (Entry), in den bei erstmaligem Aufruf des Graphen gesprungen wird. Hierbei ist allerdings darauf zu achten, daß die Zustandsübergänge der höheren Ebenen auch eine höhere Priorität besitzen, also bei einer erfüllten Bedingung in der höheren Ebene aus dem Untergraph herausgesprungen wird. Falls aus einem Untergraphen vorzeitig herausgesprungen wird, verbleibt dieser Zustand in seinem momentanen Stand, um bei einem erneuten Aufruf an dieser Stelle weiterzuarbeiten.

Die Verbindungen nach außen werden mittels globaler Signale, die vom Typ in, out oder inout sein können, hergestellt. Über diese Signale hinaus sind interne Variablen deklarierbar. Diese Variablen können in jeder hierarchischen Stufe deklariert werden, und stehen dann in dieser Ebene, sowie allen Unterebenen zur Verfügung.

Für den Test der entworfenen Komponente steht ein Waveform-Viewer zur Verfügung. Hier können alle Signale und Variablen sowie die jeweils aktiven Zustände beobachtet werden. Die Ausgabe von komplexen Datentypen, wie z.B. integer oder float, ist hierin allerdings leider nicht möglich. Hier muß man sich mit Hilfsausgaben innerhalb der Aktionen in einem Zustand oder einem Übergang behelfen. Um eine Testumgebung zu gestalten, steht der Stimuli-Editor zur Verfügung. Im Stimuli-Editor wird die Belegung der Ports von außen angegeben. Hierzu können beliebig viele parallele Stimulus-Prozesse angegeben werden. Eine Synchronisierung ist mittels wait-Statements, ähnlich zu wait-Statements in VHDL-Beschreibungen, zu erreichen.

Abschließend läßt sich bemerken, daß die Entwicklung von Komponenten mittels SPeeDCHART vom Konzept her zwar sehr schön ist, allerdings in der Praxis teilweise größere Probleme als erwartet auf-

wirft.
So ist es sehr schwierig, größere Zustandsgraphen übersichtlich zu gestalten, wenn sich keine weitere Unterteilung in Untergraphen anbietet. Bei der Arbeit während der Projektgruppenzeit ergaben sich teilweise Diagramme, die nur noch für den jeweiligen Entwickler des Diagrammes lesbar waren, was die Gruppenarbeit erheblich erschwerte.
Ein weiteres großes Problem ergab sich daraus, daß man zwar als gewünschte Synthesesprache VHDL für Synopsys-Synthese anwählen konnte, der ausgegebene VHDL-Code allerdings keineswegs direkt synthetisierbar war. Ein wiederholt auftretender Fehler war, daß der automatisch generierte Code Datentypen enthielt, die nicht vorher definiert wurden, und auch nicht in der mitgelieferten Library stehen. Hier mußte dann teilweise per Hand der Code verändert werden.

3.2 Synopsys

Das Synopsys-Paket besteht aus vielen Einzelkomponenten, die zusammen ein sehr leistungsfähiges Paket zum Entwurf etwa eines ASIC oder allgemein eines integrierten Schaltkreises ergeben. Man sollte jedoch schon zu Beginn von der Vorstellung Abstand nehmen, daß hier die Idealvorstellung die man von High-Level-Systhese hat realisiert wurde. Man hat oft mit Einschränkungen zu kämpfen und ein Design, daß erfolgreich mit einem VHDL-Simulator simuliert wurde läßt sich oft erst nach einigen Umwegen und Änderungen auf die Gatterebene abbilden.
Desweiteren stellen die Pakete auch schon bei Projekten der Grösse, wie sie in Projektgruppen an unserer Uni entstehen erhebliche Anforderungen an die Hardware. Selbst auf einer Sparc10 sind Zeiten von ein bis zwei Stunden für die Synthese eines Teildesigns keine Seltenheit, und das Starten eines Synthesetools auf einem Rechner mit weniger als 64MB Hauptspeicher ist nicht sinnvoll, oft sogar unmöglich.

3.2.1 Der Schematic-Editor

Der Schematic-Editor ist ein Werkzeug, mit dessen Hilfe Teile eines Bausssteins graphisch entworfen werden können. Man zeichnet zum Beispiel zunächst eine Komponente in einer beliebigen Form, versieht sie mit In- und Outputports und kann dann eine Schablone für eine Beschreibung in VHDL mit Entity- und Architecturedeklarationen generieren lassen, in die man „nur“ noch die entsprechende Verhaltensbeschreibung in VHDL einfügen muss. Hat man so einige Elemente beschrieben, so kann man diese nach Belieben verbinden, und sich auch den VHDL-Code des Designs, das aus den zusammengesetzten Teildesigns besteht genrieren lassen.
Die Vorteile dieses Verfahrens liegen in der Tatsache, daß man recht übersichtlich einen hierarchisch gegliederten Entwurf vornehmen kann, und mit Hilfe des Tools auch sehr gut durch diese Hierarchie navigieren kann. Man sollte sich jedoch vorher überlegen, ob es wirklich sinnvoll ist, falls man etwa ein Addierwerk für Ganzkommazahlen entwerfen möchte, damit zu beginnen ein AND/OR/XOR-Gatter zu zeichnen, sich dann daraus Halbaddierer zusammenzusetzen, um diese dann zu einem Volladdierer zu kombinieren etc. . In einem solchen Fall sollte man lieber eine großes Bauteil namens Addierer zeichnen, und dann das Verhalten dieses Bauteils komplett in VHDL beschreiben, was sicher nur einen Bruchteil der Zeit, die für das Zeichnen und Verbinden vieler Einzelkomponenten nötig ist in Anspruch nimmt, und in diesem Fall auch der Übersichtlickeit nicht schadet.
Entwirf man komplexere Rechenwerke, so wird man sich zwangsläufig über eine Modularisierung gedanken machen. Meiner Meinung nach, ist jedoch ein (insbesondere in VHDL) geübter Programmierer eigentlich nicht auf dieses Tool angewiesen, da es jemandem der mit dem einem VHDL-Debugger, wie dem „vhdldbx“ umgehen kann, keine nennenswerten weiteren Möglichkeiten bietet.

3.2.2 Der VHDL-Compiler „vhdlan“

Jeder erzeugte VHDL-Code wird sinnvoller Weise mit diesem Compiler übersetzt. Der Aufruf erfolgt wahlweise über die jeweilige shell einfach durch Eingabe von `vhdlan` *filename.vhdl*

oder auch durch Anwahl bestimmter Menüpunkte von diversen Tools (z.B. dem Schematic-Editor) aus.
Verschiedenste Einstellungen, bezüglich der Verzeichnisse, in denen sich Libraries befinden, und Verzeichnissen, in denen die eigenen Library liegen, werden mit Hilfe eines Setup-Files names *.synopsys_vss.setup* vorgenommen, ein Beispiel dazu befindet sich im Anhang.

3.2.3 Der VHDL-Debugger/Simulator „vhdldbx"

Der „vhdldbx" ist eins der wichtigsten Werkzeuge. Mit seiner Hilfe kann der mit dem „vhdlan" übersetzte Code simuliert und nach Fehlern durchsucht werden. Er bietet eine Vielzahl von Möglichkeiten Signale ständig zu überwachen und diese dann zum Beispiel auf einem Waveform-Display anzuzeigen. Desweiteren können Breakpoints gesetzt werden. Wird ein solcher Breakpoint erreicht, wird die Simulation unterbrochen und es ist möglich sämtliche Signale und Variablen zu überprüfen und evtl. abzuändern. Breakpoints können feste Posistionen im VHDL-Code sein aber auch bestimmte Flanken von Signalen.
Ein weiterer wichtiger Punkt ist das Setzen von Stimuli auf die Signale, ohne das keine Simulation möglich wäre. Alle diese Funktionen können auch mit Hilfe einer Scriptsprache gesteuert werden. Diese Scripte ersparen durch die Definition eigener komplexer Komandos viele Mausbewegungen und -klicke.

3.2.4 Das Synthesetool „design_analyzer"

Hat man den VHDL-Code erfolgreich mit dem „vhdldbx" simuliert, so ist der nächste Schritt ihn auf die Gatterebene zu übersetzen. Dies ist die Aufgabe des Logik-Synthese-Tools „design_analyzer", das Logik-Designs übersetzt und für Geschwindigkeit, sowie Chipgröße optimiert. Man kann ein Design hier in verschiedensten Formaten (z.B. VHDL, Edif, DB) einlesen, wie auch exportieren, so daß dieses Tool auch in Verbindung mit nicht-Synopsys Software zum Einsatz kommen kann. Man darf jedoch nicht vergessen, dass insbesondere nicht alle VHDL-Konstrukte unterstützt werden (zum Beispiel guarded-Signale, floating-Typen, nicht alle wait-statements etc.) Einen guten ersten Einblick gibt ausserdem die im ersten Seminarband zu dieser Projektgruppe nachzulesende Seminararbeit „Synopsys und VOTAN" von Thorsten Kukuk.

Erwähnenswert ist außerdem noch, daß sich auch der „design_analyzer" recht komfortabel über eine Script-Sprache steuern läst. Diese Scripte werden dann von der „dc_shell" interpretiert. Die Shell automatisiert dann den gesamten Prozess und ermöglicht es das Tool ohne X11, und somit „offline" zu nutzen, was einem das langwierige „Babysitting" während des Übersetzens eines großen Designs erspart.

3.3 INCA

Die InCA (Integrated Circuit Applications)-Software übernimmt die Übertragung eines Netzlistenfotmates (in unserem Fall EDIF) auf den InCA-Simulator am PC. Der Schritt von EDIF nach *.ins-Files wird genauer im Kapitel 7. erläutert. Die weiteren Arbeitschritte bis zum eigenlichen Laden des Design auf den Simulator sind dann folgende:

- `Netzlistenkonvertierung (s.o., s. Kap. 7.)`
- `Prepartitionierung`
  Dieser Schritt isoliert alle Schaltkreise und andere Konstrukte, die speziell behandelt werden

müssen, wie z.B. Tristates.

- `Partitionierung`
  Die automatische Partitionierung führt einen eingebauten Algorithmus aus, der die Schaltkreise in Blöcke einteilt, wobei jeder dieser Blöcke auf ein FPGA passt. Dabei können mehrere Algorithmen und verschiedene Optimierungsstufen gewählt werden. Sollte keiner der vorgegebenen Algorithmen zu einem akzebtablen Ergebniss führen, so kann man die gesamte Partitionierung auch von Hand durchführen.
  Integriert in diesen Schritt ist auch das sogennamte "Floor-Planning", es zeigt die Verbindungen (Connectivity) zwischen den einzelnen Subdesigns. Hier könnte man nun einzelne Schaltkreise logisch zu Gruppieren, z.B. weil sie sehr viele Verbindungen untereinander haben, um die "Partitionseffizienz" zu steigern.

- `Make Hardware`
  Hierbei muss man lediglich angeben welchen Typ von Simulator es sich hier handelt. Das Tool generiert dann die Hardware, die dann im nächsten Schritte belegt wird.

- `Plaziere Zellen`
  Hier werden die bei der Partitionierung eingeteilten Schaltkreise der physikalischen Hardware zugewiesen, zumindest symbolisch im Browse-Tool. Jeder Block wird genau einem FPGA der Hardware zugewiesen.

- `Globales Routing`
  Dieser Schritt definiert die Verbindungen zwischen den logischen Design auf jedem FPGA (FPGA-to-FPGA Interconnections). Hierbei werden auch die Verbindungen zu einem Logic-Analyser, andere Testgeräte oder dem Zielgerät (in unserem Fall das Interface des Prozessors in der SPARC) festgelegt. Desweiteren besteht die Möglichkeit, Prioritäten für die einzelnen Verbindungen zu setzen bzw. zu verändern.

- `Compilieren`
  Hierbei werden alle FPGAs (Logik-FPGAs sowie Crossbar-FPGAs) programmiert mit der dazugehörigen Logik und den Verbindungen. Im Prinzip ist dieser Schritt nur dazu da, das Globale Routing in ein maschinenlesbares Format zu bringen.

- `Generieren`
  Nach dem das obige Design auf die FPGAs compiliert ist, muß alles in ein Format gebracht werden, das man direkt auf den Simulator laden kann. Hier wird nun genau solch ein Format generiert, allerdings noch nicht geladen.

Alle obigen Schritte können natürlich auch von Hand (Drag & Drop) ausgeführt werden. Um alle Schritte automatisch auszuwählen, läd man den browser und wählt unter PROPERTIES→INSTANCE →autoconfigure. Unter Properties stellt man nun alles auf YES. Mittels EXECUTE wird dann alles ausgeführt und nach jedem Schritt gespeichert.

Wir gehen also hier davon aus, daß wir ein Design-File haben, welches im InCA-internen ins-Format vorliegt. Vor dem Arbeiten mit den InCA-Tool muß noch folgenden Environment geladen werden:

```
setenv XLILINX $INCA_SOFT/xlilinx
setenv XACT $INCA_SOFT/xilinx
setenv XACT_MEMSIZE 12
setenv XLM_ENABLE_LM_LICENCE_FILE 1
setenv LM_LICENCE_FILE /usr/local/data/licence.dat
```

gestartet wird der browser, das Tool, was die gesamte Funktionalität der InCA-Software auf der Sun beinhaltet, mit folgendem Befehl:

```
browse -top <design-file>
```

wobei das Design-File entweder ein EDIF-Datei oder eine ins-Datei ist. Genauere und weiterführende Beschreibung des browsers findet man im Seminar "FPGA/INCA" von Frank Ullmann zu dieser Projektgruppe, sowie im CONCEPT SILICON USER GUIDE.

Kapitel 4

Der Algorithmus

4.1 Ein $O(n^3)$-Algorithmus

Das Programm ist eine direkte Umsetzung des Algorithmus, den Ralf Jungblut in seinem Seminar vorgestellt hat. Der Algorithmus besteht im wesentlichen aus drei Schritten

- Erstellen des Gleichungssystemes
- Lösen des Gleichungssystemes
- Berechnung der Polynomkoeffizienten

4.1.1 Erstellen des Gleichungssystemes

Für n Stützstellen werden $(n-2)$ Gleichungen aufgestellt. Für jede Gleichung werden die Punkte $(n-1)$, n und $(n+1)$ benötigt. Die so entstandene Koeffizientenmatrix besitzt Dreibandgestalt.

$$A = \begin{pmatrix} 2(x_2 - x_0) & x_2 - x_1 & & \dots & 0 \\ x_2 - x_1 & 2(x_3 - x_1) & x_3 - x_2 & & \\ \vdots & x_3 - x_2 & \ddots & & \\ 0 & & & & \end{pmatrix}$$

Der Lösungsvektor ist eine Spalte mit $(n-2)$ Elementen, die sich folgendermaßen berechnen:

$$B_i = \frac{6(y_{i+1} - y_i)}{x_{i+1} - x_i} - \frac{6(y_i - y_{i-1})}{x_i - x_{i-1}}$$

4.1.2 Lösen des Gleichungssystemes

Nun ist das Gleichungssystem $A \cdot x = B$ zu lösen. Hierfür wird der Gaußalgorithmus verwendet. Da die Matrix aber Dreibandform hat (nur die Hauptdiagonale [1] sowie die Diagonalen unmittelbar oberhalb und unterhalb der Hauptdiagonalen sind mit Elementen ungleich Null besetzt), gibt es für diesen Spezialfall sicherlich effizientere Lösungsmethoden.

4.1.2.1 Der Gaußalgorithmus

Der Gaußalgorithmus löst ein Gleichungssystem der Form $A \cdot x = B$, wenn dieses eindeutig bestimmt ist, d.h. die Anzahl der Gleichungen und die Anzahl der Unbekannten sind gleich. Die Lösung erfolgt iterativ, indem man die Koeffizientenmatix A durch elemetare Zeilenumformungen in die Einheitsmatrix umformt. Dieselben Umformungen werden auch auf den Lösungsvektor B angewandt. Ist die Matrix A in die Einheitsmatrix umgewandelt, enthält der Lösungsvektor B die Lösung.

[1] Ein Matrixelement gehört zur Hauptdiagonale, wenn der Spalten- und der Zeilenindex gleich sind.

Im Algorithmus erfolgt die Lösung spaltenweise. Sei i die gerade aktuelle Spalte. Zuerst wird das Element $A_{i,i}$ auf 1 gebracht, indem die ganze Zeile $A_{i,x}$ durch $A_{i,i}$ dividiert wird. Dann wird ein entsprechendes Vielfaches der i-ten Zeile zu allen anderen Zeilen addiert, so daß die Elemente in der i-ten Spalte zu Null werden.

Beispiel:

Es soll folgendes Gleichungssystem gelöst werden:

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 2 & 1 & 3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 14 \\ 11 \\ 13 \end{pmatrix}$$

Element $A_{1,1}$ auf 1 bringen (ergibt eine Änderung)

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 2 & 1 & 3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 14 \\ 11 \\ 13 \end{pmatrix}$$

(-2) mal 1.Zeile zu 2.Zeile addieren

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & -1 & -5 \\ 2 & 1 & 3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 14 \\ -17 \\ 13 \end{pmatrix}$$

(-2) mal 1.Zeile zu 3.Zeile addieren

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & -1 & -5 \\ 0 & -3 & -3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 14 \\ -17 \\ -15 \end{pmatrix}$$

1. Spalte ist jetzt umgewandelt.

Element $A_{2,2}$ auf 1 bringen
Multiplikation der 2.Zeile mit (-1)

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & 1 & 5 \\ 0 & -3 & -3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 14 \\ 17 \\ -15 \end{pmatrix}$$

(-2) mal 2.Zeile zu 1.Zeile addieren

$$\begin{pmatrix} 1 & 0 & -7 \\ 0 & 1 & 5 \\ 0 & -3 & -3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -20 \\ 17 \\ -15 \end{pmatrix}$$

3 mal 2.Zeile zu 3.Zeile addieren

$$\begin{pmatrix} 1 & 0 & -7 \\ 0 & 1 & 5 \\ 0 & 0 & 12 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -20 \\ 17 \\ 36 \end{pmatrix}$$

2. Spalte ist umgewandelt

Elemet $A_{3,3}$ auf 1 bringen.
Division der 3.Zeile durch 12

$$\begin{pmatrix} 1 & 0 & -7 \\ 0 & 1 & 5 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -20 \\ 17 \\ 3 \end{pmatrix}$$

7 mal 3.Zeile zu 1.Zeile addieren

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 5 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 17 \\ 3 \end{pmatrix}$$

(-5) mal 3.Zeile zu 2.Zeile addieren

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

3. Spalte ist jetzt umgewandelt.

Die Koeffizientenmatix A ist jetzt in die Einheitsmatrix umgewandelt, der Vektor B enthält die Lösung: $x_1 = 1, \qquad x_2 = 2, \qquad x_3 = 3.$

4.1.3 Berechnung der Polynomkoeffizienten

Aus der Lösung des Gleichungssystemes müssen jetzt noch die Koeffizienten der Splinepolynome errechnet werden. Jedes Polynom hat die Form $ax^3 + bx^2 + cx + d$.

$$a_i = \frac{1}{6}(x_{i+1} - x_i)(B_i - B_{i-1})$$

$$b_i = \frac{1}{2}B_{i-1}$$

$$c_i = \frac{1}{x_{i+1} - x_i}(y_{i+1} - y_i) - \frac{x_{i+1} - x_i}{6}(B_i + 2B_{i-1})$$

$$d_i = y_i$$

4.1.4 Der Algorithmus in C

```
#include <stdio.h>

#define MAX 16

float Mat[MAX][MAX];                  // Matrix für Gauß
float MatX[MAX], MatY[MAX];           // Stützstellen
float MatA[MAX],MatB[MAX],MatC[MAX];  // Koeffizienten der Polynome
int n;                                // Anzahl der Stützstellen

void Reset(void) {
// Alle Parameter zurücksetzen
int x,y;
   printf(''Reset\n'');
   n=0;
   for (x=0; x<MAX; x++)
   for (y=0; y<MAX; y++)
      Mat[y][x] = 0.0;
   for (x=0; x<MAX; x++) {
      MatX[x] = 0.0;
      MatY[x] = 0.0;
```

```
   }
}

void MatMul(int Zeile, float Wert) {
// Eine Matrixzeile mit einem festen Wert multiplizieren
int i;
   printf(''MatMul : Zeile %i, Wert %10.5f\n'', Zeile, Wert);
   for (i=0; i<=n; i++)
      Mat[Zeile][i] *= Wert;
}

int Exchange(int Zeile) {
// Diagonalelement in Zeile Mat[Zeile,Zeile] ist Null. Da dieses Element
// aber auf 1 gebracht werden muß, würde es eine Division durch Null geben.
// Deshalb wird eine zweite Zeile gesucht, in der das Element Mat[x,zeile]
// ungleich Null ist. Diese beiden Zeilen werden jetzt miteinander
// vertauscht. Diese Zeile muss jedoch unterhalb der Zeile 'Zeile' liegen
// (x > Zeile). Wird keine solche Zeile gefunden, ist das Gleichungssystem
// nicht lösbar.
int y,i;
float f;

   for(y=Zeile+1; y<n; y++) {
      if (Mat[y][Zeile] != 0.0)
         break;                        // Zeile zum Vertauschen gefunden
   }
   if (Mat[y][Zeile]==0) return -1;   //Fehler, Gleichungssystem ist nicht lösbar

   printf(''Exchange : Vertausche Zeile %i und %i\n'',Zeile,y);
   // vertauschen
   for(i=0; i<=n; i++) {
      f=Mat[Zeile][i]; Mat[Zeile][i] = Mat[y][i]; Mat[y][i] = f;
   }
   return 0;
}

void MatAdd(int Ref, int Zeile) {
// Zeile 'Ref' wird zu Zeile 'Zeile' addiert.
// Vorher wird Zeile 'Ref' aber so multipliziert, daß nach der Addition
// das Element Mat[Zeile][Ref] zu Null wird. Die Multiplikation wird jedoch
// nur Prozedurintern ausgeführt, Zeile 'Ref' wird nicht verändert.
int i;
float Mul;

   Mul = Mat[Zeile][Ref]/Mat[Ref][Ref];
   for (i=0; i<=n; i++)
      Mat[Zeile][i] -= Mul*Mat[Ref][i];
}

int Loese(void) {
// Das Gleichungssystem nach der Gauß'schen Methode lösen
// Vorgehensweise: Die Matrix wird spaltenweise abgearbeitet. zuerst wird
//       Diagonalelement auf 1 gebracht; evtl. ist das Element 0,
//       dann muss mit einer Zeile getauscht werden, wo das Element
//       ungleich Null ist. Diese Zeile muss unterhalb der
//       aktuellen Zeile liegen. Wird keine Zeile zum Vertauschen
//       gefunden, ist das Gleichungssystem nicht lösbar.
//       Jetzt wird diese Zeile nach entspr. Multiplikation von
//       allen anderen Zeilen subtrahiert, un alle restlichen
```

```
//      Elemente in dieser Spalte auf Null zu bringen.
//      Der Lösungsvektor steht dann in Spalte (n).
int x,y;
int err;

   for (x=0; x<n; x++) {
      if (Mat[x][x] == 0)
         if (Exchange(x)<0)             // Diag.element ist Null, zwei Z. vertauschen
            return -1;                  // Fehler,kein Element zum Vertauschen gefunden
      MatMul(x, 1/Mat[x][x]);           // Diagonalelement auf 1 bringen
      for(y=0; y<n; y++) {
         if (x!=y)
            MatAdd(x,y);                // die Zeilen x und y so addieren,
                                        // daß das Element (yx)=0 wird
      }
   }
   return 0;
}

void PrintMat(void) {
// Matrix 'Mat' ausgeben
int x,y;

   for (y=0; y<n; y++) {
      for (x=0; x<= n; x++)
         printf(''%10.5f '',Mat[y][x]);
      printf(''\n'');
   }
   printf(''\n'');
}

void FillMat(void) {
// Das Gleichungssystem aufstellen
int i;
   for (i=1; i<=n; i++) {
      if (i>1)
         Mat[i-1][i-2] = (MatX[i] - MatX[i-1]);
      Mat[i-1][i-1] = 2*(MatX[i+1] - MatX[i-1]);
      if (i<n)
         Mat[i-1][i] = MatX[i+1] - MatX[i];
      Mat[i-1][n] = 6/(MatX[i+1]-MatX[i])*(MatY[i+1]-MatY[i]) -
            6/(MatX[i]-MatX[i-1])*(MatY[i]-MatY[i-1]);
   }
}

void MakeSpline(void) {
// Koeffizienten der Splinepolynome ausrechnen
int i;
float hi;                               // Abstand der X-Werte
float yi,yi1;                           // Koeffizienten der Funktion
float y_i, y_i1; // Koeffizienten der 2. Ableitung (aus LGS)

   for (i=0; i<=n; i++) {
      hi = MatX[i+1] - MatX[i];
      yi = MatY[i];
      yi1= MatY[i+1];

      if (i>0) {
         y_i = Mat[i-1][n];
```

```
      } else
         y_i = 0;
      if (i<n) {
         y_i1 = Mat[i][n];
      } else
         y_i1 = 0;

      MatA[i] = 1.0/6.0/hi * (y_i1 - y_i);    // Ai's berechnen
      MatB[i] = 0.5*y_i;    // Bi's
      MatC[i] = 1.0/hi*(yi1-yi) - hi/6.0*(y_i1 + 2.0*y_i);    // Ci's
   }
}

void PrintKoeff(void) {
// Koeffizienten der Polynome ausgeben
int i;
   printf(''\n-------------------\n'');
   for (i=0; i<=n; i++)
      printf(''%7.3f '',MatA[i]);
   printf(''\n'');
   for (i=0; i<=n; i++)
      printf(''%7.3f '',MatB[i]);
   printf(''\n'');
   for (i=0; i<=n; i++)
      printf(''%7.3f '',MatC[i]);
   printf(''\n'');
   for (i=0; i<=n; i++)
      printf(''%7.3f '',MatY[i]);
   printf(''\n'');
}

void Sort(void) {
// Stützstellen nach steigenden X-Werten sortieren
int i,j;
float f;

   for (i=n-1;i>0; i--)
   for (j=0;j<i; j++)
   if (MatX[j] > MatX[j+1]) {
      // vertauschen
      f=MatX[j]; MatX[j]=MatX[j+1]; MatX[j+1]=f;
      f=MatY[j]; MatY[j]=MatY[j+1]; MatY[j+1]=f;
   }
}


void Eingabe(void) {
int i;
   n=0;
   do {
      printf(''Anzahl der Stützstellen (3...%i) : '',MAX);
      scanf(''%i'', &n);
   } while ( n<3 || n>=MAX);

   for (i=0; i<n; i++) {
      printf(''Stützstelle %i :\nx= '',i); scanf(''%f'', &MatX[i]);
      printf(''y= ''); scanf(''%f'', &MatY[i]);
   }
   Sort();
```

```
}

void main(void) {
   Reset();
   Eingabe();
   n-=2;

   FillMat();                          // Koeffizienten für das Gleichungssystem
                                       // in die Matrix übertragen

   PrintMat();
   if (Loese()<0)
      printf(''Fehler\n'');
   PrintMat();
   MakeSpline();                       // Splinepolynome ausrechnen
   PrintKoeff();
   printf(''\n'');
}
```

4.1.5 Prozedurbeschreibungen

4.1.5.1 Globale Variablen

Mat: Diese Matrix enthält die Koeffizientenmatrix und den Lösungsvektor. Fast alle Funktionen greifen auf diese Matrix zu.

MatX: x-Werte der Stützstellen

MatY: y-Werte der Stützstellen

MatA, MatB, MatC: Speichern die Stützstellen der Splinepolynome. MatA speichert den kubischen Term, MatB den quadratischen Term, MatC den linearen Term. Der konstante Term wird nicht extra gespeichert, da der direkt aus den Stützstellen gelesen werden kann.

n: Anzahl der Stützstellen. Nach der Eingabe wird n um zwei dekrementiert, so daß n jetzt die Anzahl der Gleichungen im Gauß-Algorithmus angibt.

4.1.5.2 Reset

Parameter: keine

Rückgabe : keine

Funktion : Die Prozedur Reset dient zur Zurücksetzung aller wichtigen Programmparameter. Dies ist dann wichtig, wenn mehrere Berechnungen nacheinander erfolgen sollen, ist aber in dem Programm noch nicht möglich. Für jede Berechnung muß neu gestartet werden.

Aufruf von : main

4.1.5.3 MatMul

Parameter: Zeile (integer) Zeile, die zu multiplizieren ist
Wert(float) : Multiplikator

Rückgabe : keine

Funktion : Die Zeile "Zeile" wird Elementweise mit der Konstanten "Wert" multipliziert.

Aufruf von : Loese

4.1.5.4 Exchange

Parameter: Zeile (integer) Zeile, die das kritische Nullelemet enthält

Rückgabe : -1 = keine Zeile zum Vertauschen gefunden, 0 = Vertauschung ok

Funktion : Die Zeile "Zeile" enthält in der Diagonalen eine Null. Da dieses Diagonalelement für den Gaußalgorithmus auf 1 gebracht werden muß, kann dieses nicht durch Multiplikationen sondern nur durch Vertauschungen (evtl. auch Addition mit einer anderen Zeile) erfolgen. Da die Zeilen oberhalb der Zeile "Zeile" bereits (teilweise) umgeformt sind, kann die entspr. Zeile nur unterhalb der aktuellen Zeile gefunden werden. Wird ein solcher Tauschpartner nicht gefunden (alle nachfolgenden Zeilen besitzen an der entsprechenden Position ebenfalls eine Null) wird -1 zurückgegeben. Damit ist das Gleichungssystem nicht lösbar. Konnte die Vertauschung korrekt erfolgen, wird 0 zurückgegeben.

Aufruf von : Loese

4.1.5.5 MatAdd

Parameter: Ref (integer) Referenzzeile
Zeile (integer) : zu verändernde Zeile

Rückgabe : keine

Funktion : Ein Vielfaches der Zeile "Ref" wird zu der Zeile "Zeile" addiert, so daß das Spaltenelement Mat[Zeile][Ref] zu Null wird. die Zeile "Ref" wird nicht verändert.

Aufruf von : Loese

4.1.5.6 Loese

Parameter: keine

Rückgabe : Lösbarkeit: 0 = Gleichungssystem konnte gelöst werden, -1 = keine Lösung möglich.

Funktion : Gleichungssystem nach der gauß'schen Methode lösen. Durch elementare Zeilenumformungen auf die Koeffizientenmatrix und den Lösungsvektor wird die Koeffizientenmatrix in die Einheitsmatrix umgeformt. Der Lösungsvektor enthält dann die Lösung. Die Umformung der Koeffizientenmatrix erfolgt spaltenweise. Zuerst wird geprüft, ob das Diagonalelement Null ist. Ist dieses der Fall, muß eine Vertauschung mit einer anderen Zeile erfolgen so daß dieses Diagonalelement ungleich Null wird (Funktion Exchange). Anschließend wird die Zeile mit dem Diagonalelement mit einer entspr. Konstanten so multipliziert, daß das Diagonalelement zu 1 wird (MatMul). Nun wird durch Addition eines Vielfachen dieser Zeile zu allen übrigen Zeilen die Spaltenelemente auf Null gebracht (Funktion MatAdd).

Aufruf von : main

Unterprogramme : Exchange(), MatMul(), Matadd()

4.1.5.7 FillMat

Parameter: keine

Rückgabe : keine

Funktion : Aufstellen der Koeffizientenmatrix und des Lösungsvektors aus den Stützstellen. Aus n Stützstellen können n-2 Gleichungen generiert werden, es müssen also mindestens 3 Stützstellen gegeben sein.

Aufruf von : main

4.1.5.8 MakeSpline

Parameter: keine

Rückgabe : keine

Funktion : Aus dem Lösungsvektor des Gleichungssystemes werden die Splinepolynomkoeffizienten erzeugt.

Aufruf von : main

4.1.5.9 Sort

Parameter: keine

Rückgabe : keine

Funktion : Sortieren der Stützstellen nach steigenden x-Werten. Die Stützstellen können in beliebiger Reihenfolge eingegeben werden, der Algorithmus liefert jedoch nur dann korrekte Ergebnisse, wenn die Stützstellen nach steigendem x-Wert sortiert sind.

Aufruf von : main

4.1.5.10 PrintMat, PrintKoeff, Eingabe und main

Weniger wichtige Routinen

PrintMat: Ausgabe der Koeffizientenmatrix und des Lösungsvektors (zu Testzwecken)

Printkoeff : Ausgabe der erzeugten Splinepolynome

Eingabe : Eingabe der Stützstellen. Zuerst wird die Anzahl der Punkte angegeben, anschließend erfolgt die Eingabe der Stützstellen, zuerst der x-Wert, dann der y-Wert.

main : Das Hauptprogramm…

4.1.6 Aufwand

4.1.6.1 Speicheraufwand

Für die Aufstellung der Gleichungssysteme wird eine Matrix mit $(n+1)*n$ Elementen benötigt, die die Koeffizienten und den Lösungsvektor enthält. Obwohl die Koeffizientenmatrix Dreibandform hat, wird während des Gaußalgorithmus die gesamte Matrix benutzt, so daß hier keine "einfache" Vereinfachung möglich ist. Testläufe haben gezeigt, daß immer nur in der aktuellen Spalte Werte ungleich Null auftreten, alle anderen Elemente sind Null (bis auf die Diagonalen). Hier wäre eine Speichereinsparung möglich, indem man nur die drei Diagonalen sowie die Spalte mit den von Null verschiedenen Werten speichern würde. Dem steht jedoch ein sehr hoher Verwaltungsaufwand gegenüber so daß sich die Sache nicht lohnen würde. Was in der Matrix gespart wird, geht durch zusätzlichen Programmkode wieder verloren und verschlechtert außerdem die Laufzeit. Der Speicheraufwand beträgt also $\mathbf{O(n^2)}$.

4.1.6.2 Zeitaufwand

Der Aufwendigste Teil ist die Lösung des Gleichungssystemes nach der Gaußmethode in der Funktion *Loese*. Jedes Element der Matrix muß mindestens einmal bearbeitet werden, da es entweder auf "1" gebracht wird, wenn es ein Diagonalelement ist oder auf "0" wenn es außerhalb der Diagonalen liecht. Bei genauer Betrachtung des Programms erkennt man in der Funktion *Loese* zwei geschaltelte *for*-Schleifen. In diesen Schleifen wird die Funktion *MatAdd* aufgerufen, die ihrerseits auch eine *For*-Schleife enthält. Die *Loese*-Funktion entält also drei geschaltelte *For*- Schleifen, die alle n Durchläufe besitzen; Zeitaufwand $\mathbf{O(n^3)}$.

4.2 Ein $O(n)$-Algorithmus

Dieser Algorithmus geht vom folgendem Ansatz aus:

$\frac{\partial^2 S_i}{\partial x^2}(x) = M_{i-1}\frac{x_i - x}{h_i} + M_i \frac{x - x_{i-1}}{h_i}$,
mit $h_i = x_i - x_{i-1}$.
Daraus folgt :
$S_i(x) = M_{i-1}\frac{(x_i - x)^3}{6h_i} + M_i \frac{(x - x_{i-1})^3}{6h_i} + A_i(x - x_{i-1}) + B_i$.
Mit $B_i = y_{i-1} - M_{i-1}\frac{h_i^2}{6}$
und $A_i = \frac{y_i - y_{i-1}}{h_i} - \frac{h_i}{6}(M_i - M_{i-1})$

Zu lösen ist das folgende Gleichungssystem:

$$\begin{pmatrix} \frac{h_1+h_2}{3} & \frac{h_2}{6} & \dots & 0 \\ \frac{h_2}{6} & \frac{h_2+h_3}{3} & \frac{h_3}{3} & \\ \vdots & \ddots & \ddots & \ddots \\ 0 & & \frac{h_{n-1}}{6} & \frac{h_{n-1}+h_n}{3} \end{pmatrix} \cdot \begin{pmatrix} M_1 \\ M_2 \\ \vdots \\ M_{n-1} \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_{n-1} \end{pmatrix}$$

mit $f_i = \frac{y_{i+1} - y_i}{h_{i+1}} - \frac{y_i - y_{i-1}}{h_i}; \quad i = 1, \dots, n-1; M_0 = M_n = 0.$

4.2.1 Lösen des Gleichungssystemes $A \cdot M = f$

$$Sei \quad A = \begin{pmatrix} d_1 & c_1 & \dots & 0 \\ b_2 & d_2 & c_2 & \\ & b_3 & d_3 & c_3 \\ \vdots & \ddots & \ddots & \ddots \\ 0 & & b_{n-1} & d_{n-1} \end{pmatrix}$$

Wir machen den Ansatz $A = L \cdot R$, mit den Matrizen

$$L = \begin{pmatrix} \alpha_1 & 0 & \dots & 0 \\ \beta_2 & \alpha_2 & 0 & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & & \beta_n & \alpha_n \end{pmatrix}, R = \begin{pmatrix} 1 & \gamma_1 & \dots & 0 \\ & 1 & \gamma_2 & \vdots \\ \vdots & \ddots & \ddots & \gamma_{n-1} \\ 0 & & & 1 \end{pmatrix}$$

Für das Matrizenprodukt $L \cdot R$ erhalten wir:

$$L \cdot R = \begin{pmatrix} \alpha_1 & \alpha_1\gamma_1 & 0 & \dots & 0 \\ \beta_2 & \beta_2\gamma_1 + \alpha_2 & \alpha_2\gamma_2 & 0 & \dots \\ & \beta_3 & \beta_3\gamma_2 + \alpha_3 & \alpha_3\gamma_3 & 0 \\ 0 & & \ddots & \ddots & \end{pmatrix}$$

Koeffizientenvergleich liefert:
$b_i = \beta_i \qquad i = 2, 3, \dots, n \qquad (1)$
$c_i = \alpha_i \cdot \gamma_i; \qquad i = 1, 2, \dots, n-1 \qquad (2)$
$\alpha_1 = d_1 \qquad (3)$
$d_i = \beta_i\gamma_{i-1} + \alpha_i; \qquad i = 2, 3, \dots, n \qquad (4)$

- Nach (1) sind β_i $(i = 2, \dots, n)$ und α_1 bekannt.

- Es sei $\alpha_i (i \geq 1)$ schon berechnet
  Aus (2) folgt

$\gamma_i = \frac{c_i}{\alpha_i}$
und daraus mit (4)
$\alpha_{i+1} = d_{i+1} - \beta_{i+1}\gamma i$

4.2.2 Algorithmus:

$\alpha_1 = d_1$
$\gamma_1 = \frac{c_1}{d_1}$
FOR i=2 TO n-1 DO
BEGIN
$\alpha_i = d_i - \beta_i\gamma_{i-1};$
$\gamma_i = \frac{c_i}{\gamma_{n-1}};$
END
$\alpha_n = d_n - \beta_n\gamma_{n-1}$

Wir definieren die Hilfsvariable $z = R \cdot M$. Damit erhalten wir:
$$L \cdot z = f$$
Das Lösen dieses Systems erfolgt durch Vorwärtssubstitution. Danach ist z bekannt. Löse nun durch Rückwärtssubstitution das System $R \cdot M = z$

4.2.3 Zusammenfassung:

1. Berechne $L \cdot R$ zerlegung
2. Löse $L \cdot z = f$
3. Löse $R \cdot M = z$.

Kapitel 5

Der Coprozessor

5.1 Ablauf einer Splineberechnung

Von der hochkomplizierten Kommunikation zwischen Integer Unit (IU / Prozessor) und Special Function Unit (SFU / Koprozessor) sowie Interface und Rechenkern bekommt ein Softwareprogrammierer erst mal nichts mit. Dafür steht eine Library zur Verfügung, die die Parameter der Funktionen in korrekte Assemblerbefehle umsetzt.

Die IU schickt die Daten an das Interface der SFU. Dort werden sie in einer 4 Stufigen Pipeline dekodiert und auf den SFU internen Datenbus gelegt. 8 Eingaberegister, die sowohl von der IU als auch vom Rechenkern beschrieben werden können, lauschen am Bus, ob die Daten für sie bestimmt sind und übernehmen sie gegebenfalls. Das jeweilige Valid–Flag des Registers wird gesetzt, damit der Rechenkern mit den anliegenden Daten die Berechnung durchführen kann. Dabei werden die Zwischenergebnisse in den Eingaberegister bzw. der SRAM–Karte zwischengespeichert. Dies geht nur so lange, wie das Valid–Flag der Eingaberegister nicht zurückgesetzt wird. Solange kann aber auch die IU keine neue Daten zur SFU schicken.

Das fertige Ergebnis wird nach und nach als 4er Tupel in die 4 Ausgaberegister geschrieben, sobald die vorherigen Daten von dem Interface an die IU weitergeleitet wurden. Nachdem der Rechenkern alle 4 Register beschrieben hat, wird das Valid–Flag der Ausgaberegister auf ‘1’ gesetzt. Die SFU setzt es wieder auf ‘0’, sobald die IU alle 4 Werte bekommen hat. Der Rechenkern kann dann die nächsten Werte schreiben.

Wenn die Library feststellt, das der Rechenkern fertig ist, fordert sie die Ergebnisse von der IU an. Diese schickt die entsprechenden Signale an die IU. Sollte der Rechenkern noch nicht so weit sein, hält die SFU die IU an, bis die ersten 4 Werte in den Ausgaberegistern stehen. Diese werden dann an die IU durchgereicht.

5.2 Interface

5.2.1 Von den ersten Entwürfen bis zum endgültigen Kern

Das Interface hat die Aufgabe, von der IU die Daten zu übernehmen bzw. bereitzustellen. Dazu gehöhrt auch, auf Anfragen der IU die richtigen Signale zu setzen oder die IU anzuhalten, wenn der Rechenkern noch nicht fertig ist. Sie übernimmt die komplette Kommunikation mit der IU.

Am Anfang standen wir vor der Entscheidung, ob wir das Interface von Wolfram Hardt übernehmen oder ein eigenes Entwickeln sollten. Das Interface von W. Hardt war eigentlich für mehrere Rechenkerne in einem Chip entworfen worden, in dieser Hinsicht also überdimensioniert.

Für ein eigenes Interface sprach zum einen, das die größte Einarbeitungszeit in einen fremden Code entfiel, von dem der Original Autor uns auch nicht mehr Helfen konnten. Anstatt ein aufgeblähtes Interface abzuspecken, wollten wir ein kleines, schlankes und schnelles Interface entwickeln, da wir auch schon Engpässe auf dem Hardwareemulator vorraussahen. Aus diesem Grund wurden auch die Register von 32 auf 24 Bit gekürzt.

An dieser Stelle sei einer der vielen verworfenen Vorschläge ausgeführt. Mit jeder Woche wurde ein neuer Interfacevorschlag geliefert, der aufgrund von in der Sitzung neu definierten Anforderungen wieder verworfen wurde. Die Idee von der Pipline des Hardt'schen Interfaces sollte übernommen werden. Da der Rechenkern als Startbedingung für seine Berechnung nur das Vorhandensein von gültigen Werten in den Registern voraus setzt, muß eine Information darüber verwaltet werden. Hierzu gibt es eine Register-Control-Unit. Diese verwaltet den Zugriff auf die 32 Datenregister. Gleichzeitig setzt sie die Zustände in dem Registermonitor auf gültig bzw. ungültig. Es werden hierbei zwei Registermonitore unterschieden:

- Registermonitor READ
- Registermonitor WRITE

Beide Registermonitore bestehen aus 32-Bit Registern, die bitweise interpretiert werden: 0 = Daten ungültig, und 1 = Daten gültig. Der READ - Monitor setzt Werte für vom Datenbus gelesene Daten (also für die ALU wichtig). Das andere Register wird gesetzt wenn Daten von der ALU in die Register zurück geschrieben werden (d.h. es liegen Ergebnisse vor). Die Monitorwerte werden dann nach dem Auslesen jeweils zurück gesetzt. Das Registerfile wird auf beiden Seiten über einen Multiplexer gesteuert. Die genaue Adressierung wird von der RCU geregelt. Der davor geschaltete Multiplexer ist für den Zugriff auf den Datenbus nötig. Er wird ebenfalls von der RCU gesteuert. Sollen Daten von der ALU aus in die Register geschrieben werden, so wird jeweils die read oder write Leitung gesetzt und auf die Adreßleitung die Nummer des gewünschten Registers gelegt. Ein LOAD in die Register verläuft ähnlich. Hier wird nur die Adreßleitung von der WRITE-Stufe der Pipline gesetzt. Das Auslesen der Register mittels STORE - Befehl geschieht dann analog. Das Schaltbild stellt den genauen Leitungsverlauf dar.

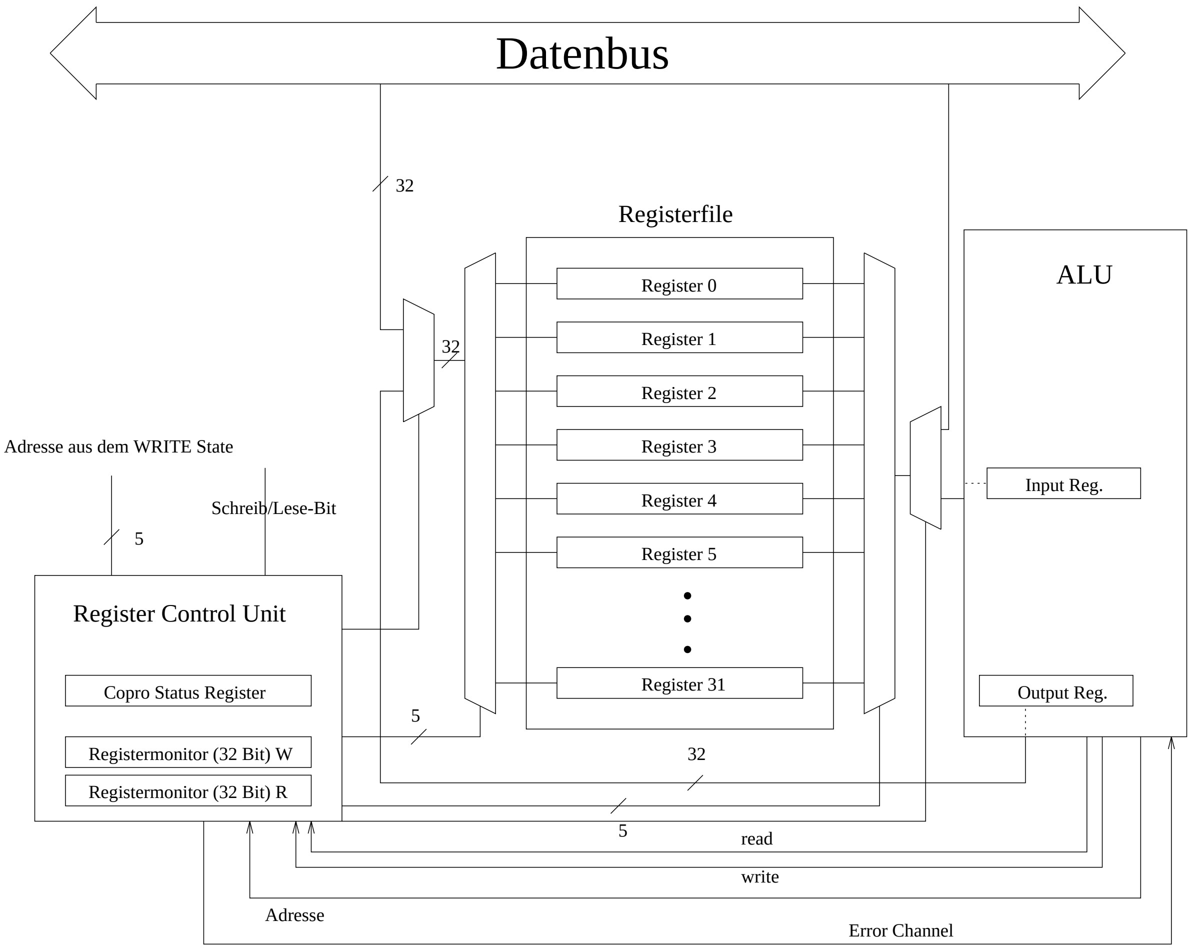

Abbildung 5.1: Registermodell

Dieser Vorschlag wurde wegen der großen Multiplexer verworfen.

Den Ausschlag für das Hardt'sche Interface gab dann letztendlich der Druck von oben und die Tatsache, daß es bereits simuliert, synthetisiert und auf den Emulator geladen worden war. Hätten wir mehr über die Kriterien gewußt, unter denen das damals gemacht worden war, unsere Entscheidung hätte anders gelautet.

Die weitere Entwicklung lief grob in 5 Stufen ab. Als erstes wurde das Interface simuliert, um damit vertraut zu werden. Dabei stellte sich schon heraus, das noch sehr viel Zeit in das Interface gesteckt werden muß, um es benutzen zu können.

In der zweiten Phase wurde dann der SFU interne Datenbus mit den Registern umgeschrieben. Der Code für die Verwaltung mehrerer Rechenkerne wurde entfernt, die Befehle der SFU auf die nötigsten abgesteckt. Die Register wurden an die neuen Anforderungen angepaßt neu implementiert, so daß die Eingaberegister auch von dem Rechenkern zu beschreiben sind. Gleichzeitig wurde die Testumgebung bedeutend verbessert sowie ein Dummy Rechenkern zum Testen geschrieben.

Dann wurde in der dritten Phase angefangen, den bisherigen Code zu synthetisieren und auf den Emulator zu laden. Gleichzeitig wurden die Pipelinestufen der SFU neu implementiert, damit zum einen die Pipline auch ihrem Namen gerecht wurde, zum anderen überhaupt das Timing zur IU stimmt. Die Synthese bereitete am Anfang keine Probleme, diese tauchten erst bei der Arbeit mit dem InCA auf. Dieser braucht PAD–Zellen für den Anschluß der externen Pins an den SFU–Bus. Zu diesem Zeitpunkt erfuhren wir dann auch, unter welchen Prämissen das Interface entwickelt worden war: Hautsache, man sieht etwas. Alle Warnungen und Fehlermeldungen wurden unterdrückt. Die dann auf den Emulator geladene Netzliste war weder lauffähig noch konnte sie von außen angesprochen werden, was eigentlich durch eine Sparc realisiert werden sollte.

Es folgte der Umstieg auf die ES2 Library, die die von InCA benötigten PAD–Zellen enthält. Nach Anpassungen am Design, die in **„Der steinige Weg von VHDL nach EDIF"** näher erläutert werden, existiert nur noch ein Problem. InCA beschwert sich, daß der Datenbus, der die IU mit der SFU verbietet, interne und externe Signale verbindet.

Während der ganzen Zeit stellte sich uns noch ein anderes Problem: Wir hatten keinen Prozessor, der einen zweiten Koprozessor unterstützt. Unsere erste Lösung sah vor, daß die SFU am Bus lauscht und ein bestimmtes Bit betrachtet. Bei gesetztem Bit sollte es sich dann um einen SFU Befehl halten. Eine externe Logik sollte dann die Leitungen zwischen der normalen FPU und der SFU umschalten. Die Lösung mußte verworfen werden, da das vermeintliche unbenutzte Bit doch von einigen Befehlen benutzt wird. Außerdem gestaltete sich die Umschaltung als zu kompliziert in Hinsicht auf das Busprotokoll. Zum Glück fand sich dann nach dem (legalen ?!? - 'Im EX.XXX steht doch auch noch eine SUN1+, laß uns mal nachschauen ...') Öffnen von einigen Rechnern unserer UNI, doch noch eine CPU, die den zweiten Koprozessor unterstützt, wenn auch nur mit Hilfe von Patches und kleineren Hardwarebasteleien.(An dieser Stelle ein dickes Dankeschön an Wolfram Hardt und Detlef (dkw), die das Suchen und den Umbau der SUN vorgenommen haben.)

5.2.2 Schnittstelle zwischen SFU–Bus und Kern

5.2.2.1 Belegung des SFU–Busses

Die nachfolgenden Tabellen 5.1 und 5.2 beschreiben die Leitungen aus Abbildung 5.2.

5.2.2.2 Bus–Protokoll

Zuerst wird er Begiff *aktivieren* erklärt, da er im Folgenden häufiger auftritt. Aktivieren sei demnach wie folgt definiert:
Die entsprechende Leitung soll von einer fallenden Taktflanke, bis zur nächsten fallenden Taktflanke auf logisch '1' gehalten werden und anschließend auf logisch '0' gezogen werden. Die entsprechenden Daten- oder Adressleitungen müssen selbstverständlich ihren Zustand behalten, bis die aktivierte Leitung wieder auf logisch '0' liegt.

Signal	Beschreibung SFU Bus
RD[31:0]	Bidirektionale Datenleitung des SFU Busses.
RWA[4:0]	Auswahl des zu beschreibenden Registers.
RRA[1:0]	Auswahl des zu lesenden Ergebnisregisters.
RWR	Wenn diese Leitung aktiv ist (RWR='1'), wird das Datum auf RD in das durch RWA adressierte Register geschrieben und das Valid Bit gesetzt.
RRD	Wenn diese Leitung aktiv ist (RRD='1'), wird der Wert des vorher mit RWA adressierte Register auf den Bus RD gelegt, ansonsten ist der Registerausgang hochohmig.
RVALID	Rechenwerk teilt dem SFU Controller mit, das alle 4 Ergebnisse in den Ausgaberegistern stehen.
RRV	Reset RVALID, setzt das Valid Bit für die Ausgaberegister zurück.
RESET	Prozessorreset, das Validflag wird zurückgesetzt.
CLK	Clocktakt der SFU.

Tabelle 5.1: Signale der Eingaberegister

Signal	Beschreibung Rechenkern
VALID	Auf dieser Leitung liegt das Validbit des Registers an.
WRITE[31:0]	Datenleitung vom Rechenkern zum Register, zum schreiben.
READ[31:0]	Datenleitung vom Rechenkern zum Register, zum lesen.
SET	Rechenkern teilt dem Register mit, das es die Daten von WRITE übernehmen soll.
RV	Reset Valid, dieses Signal wird vom Rechenkern benutzt, um das Valid Flag im Register zu löschen.
RESULT[31:0]	Datenleitung vom Rechenkern zum Ausgaberegister
SR	Set Result, damit teilt das Rechenwerk dem Ausgaberegister mit, das es den anliegenden Wert übernehmen soll.
SRV	Set RVALID, setzt das Valid Bit für die Ausgaberegister.
CLK	Clocktakt der SFU.

Tabelle 5.2: Signale der Ausgaberegister

Reset: (Reset-Leitung auf '1')

- Interface:

  Die RRV-Leitung muß für die Dauer eines Systemzyklus auf logisch '1' gelegt werden.

- Rechenkern:

  Die RV-Leitung muß für die Dauer eines Systemzyklus auf logisch '1' gelegt werden.

Input-Register vom Interface beschreiben:

1. Adresse des Registers auf RWA legen
2. Daten auf RD legen
3. zur Datenübernahme RWR aktiviern

Input-Register vom Rechenkern lesen:

1. warten bis VALID auf logisch '1' liegt
2. Daten von den READ-Leitungen des entsprechenden Registern lesen

Input-Register vom Rechenkern schreiben:

1. warten bis VALID auf logisch '1' liegt
2. Daten auf WRITE-Leitungen legen
3. SET aktivieren

Output-Register vom Rechenkern beschreiben:

1. warten bis RVALID auf logisch '0' liegt
2. Daten auf die RESULT-Leitungen legen
3. SET-Result aktivieren
4. wenn das Ergebnis vollständig im Output-Registersatz ist, dann SRV aktivieren

Output-Register vom Interface lesen:

1. warten, bis RVALID auf logisch '1' liegt
2. Registeradresse auf RRA legen
3. RRD aktivieren
4. Daten von RD lesen
5. wenn alle Daten gelesen, dann RRV aktivieren

5.2.2.3 Die Register

Da die Register vom Rechenkern beschrieben werden sollen muß ein Multiplexer den Zugriff regeln. Zunächst aber verfügt jedes Register über eine Erkennungseinheit (ICH). Diese liest den Adressbus der Register und setzt entsprechend den Multiplexer bzw. das Select der Result-Register. Die Result-Register sind Simpler, da sie nur vom Rechenkern beschrieben werden. Soll ein Wert gelesen werden, so wird der Wert nur dann geliefert, wenn der Rechenkern diesen als gültig (valid) definiert hat. Ist dieser Wert gelesen, so kann das Vaildbit zurückgesetzt werden, damit der Rechenkern wieder schreiben kann.
Die Operanden-Register sind aufwendiger. Sind die Werte vom Interface aus geschrieben, so wird ein Vaildbit für den Rechenkern gesetzt. Der Rechenkern kann dann seinerseits das Register über die Write-Leitungen beschreiben. Vorher muß er aber die RV-Leitung betätigen, damit der Multiplexer umschaltet. Über die ENable-Leitung wird bestätigt, daß die Werte gültig sind. Da der Bus von 32 auf 24 Bit gekürzt wurde, haben die entsprechenden Komponenten auch nur eine Breite von 24 Bit. Eine genaue Übersicht ist in der Schemaskizze des Interfaces zu sehen.

5.2.3 Der Controller

5.2.3.1 Die Pipeline

Die Pipeline eines Coprozessors für einen Sparc muß aus 4 Stufen bestehen, da die Pipeline der Integer–Unit auch vierstufig ist. Die Entwicklung unseres Coprozessorinterfaces orientierte sich an der Pipeline der Sparc-FPU, die laut SUN kompatibel mit der Coprozessorpipeline ist.
Im folgenden werde ich den Aufbau der Pipeline beschreiben. Die Sourcen des Coprozessorinterfaces im Anhang werden dem Leser zum parallellesen empfohlen.

5.2.3.1.1 Decode–Phase

Decode–Phase

In dieser Phase werden, wie der Name schon sagt die Befehle decodiert.
Es gibt 3 Befehle:

1. Der RESET–Befehl
2. Der LDC–Befehl
3. Der STC–Befehl

Liegt das INST–Signal der Integer Unit (IU) auf '1' so signalisiert dies der Decode–Phase, daß ein Befehl auf dem Datenbus anliegt.
Der RESET–Befehl löst einen internen Coprozessorreset aus, mit dem der Prozessor softwaremäßig zurückgesetzt werden kann. Dieser Befehl wird komplett in der Decode–Phase ausgeführt.
Die Decode-Phase besitzt zwei Pufferregister in denen ankommende Befehle zwischengespeichert werden. Handelt es sich um einen LDC– b.z.w. um einen STC–Befehl, so wird der Befehl in Puffer 2 durch den Befehl in Puffer 1 überschrieben und in Puffer 1 wird der neue Befehl abgelegt. Kommt irgend ein anderer Befehl, den der Coprozessor nicht kennt, so wird in Puffer 1 ein CNOP abgelegt (Bit 4 wird auf '1' gesetzt). Dieses Vorgehen soll Fehlinterpretationen in den nachfolgenden Pipelinestufen ausschließen. Ist Bit 4 auf '0' gesetzt handelt es sich immer um einen gültigen Befehl. Und zwar bei Bit 3 gleich '0' um einen LDC–Befehl und bei Bit 3 gleich '1' um einen STC–Befehl.
Ist das CINS1– oder CINS2–Signal der IU aktiv, so wird entweder der Befehl aus Puffer 1 oder der aus Puffer 2 in die Coprozessorpipeline geschoben.

5.2.3.1.2 Execute–Phase

Execute–Phase

In der Execute–Phase werden hauptsächlich Verwaltungssachen erledigt. Hier wird vorsorglich CCC auf "00" gesetzt.
Mit dem HOLD–Signal kann die gesamte Coprozessorpipeline angehalten werden. Mit FLUSH kann die IU die gesamte Coprozessorpipeline leeren, z.B. vor einem Kontextswitch. Ist weder HOLD noch FLUSH aktiv, so wird der Befehl aus der Inputpipeline einfach in die Outputpipeline durchgeschoben. Bei einem FLUSH wird der gerade angekommene Befehl durch ein CNOP überschrieben und in die Outputpipeline weitergeleitet.

Handelt es sich bei dem aktuellen Befehl um einen STC–Befehl, so wird in dieser Stufe überprüft, ob nicht schon ein eventuell früher beschriebenes Register nocheinmal beschrieben werden soll. Ist dies der Fall, so wird HOLD so lange aktiviert, bis das Register vom Rechenkern ausgelesen wird, d.h. die Pipeline steht so lange still.

5.2.3.1.3 Write–Phase

Write–Phase

Die Write–Phase führt im wesentlichen nur den STC–Befehl aus. Bei steigender Taktflanke wird die Adresse auf RRA gelegt. In einem Statusregister wird registriert, welche Register bereits ausgelesen worden sind. Sind alle Registerbits auf '1', so wird RRV auf '1' gesetzt und das Statusregister zurückgesetzt um dem Rechenkern zu signalisieren, daß alle Register gelesen wurden und jetzt von ihm neu beschrieben werden können.
Bei fallender Taktflanke wird das Datenwort, das dann an RD anliegt auf den Datenbus gelegt. Handelt es sich nicht um einen STC–Befehl, so wird die Stufe vom Datenbus durch anlegen von "ZZZ..." abgekoppelt.
Bei einem HOLD wird die ganze Stufe angehalten und bei einem FLUSH wird der aktuelle Befehl genau wie in der Execute–Phase durch ein CNOP ersetzt.

5.2.3.1.4 Write–Hold–Phase

Write–Hold–Phase

Die Write–Hold–Phase führt den LDC–Befehl aus. Bei einem LDC–Befehl werden die auf dem Datenbus liegenden Daten in das durch RWA adressierte Register geschrieben.
War der letzte ausgeführte Befehl auch ein LDC und liegt jetzt HOLD auf '1', so handelt es sich um einen Cache–Miss und es muß gewartet werden bis MDS auf '1' liegt. Der Wert, der dann auf dem Datenbus liegt ist dann erst gültig und dieser wird dann über den alten falschen Wert geschrieben.

5.3 Kern

5.3.1 Der Algorithmus in SPeeDCHART

Als Grundlage für die Entwicklung des Rechenkerns wurde der in Kapitel 4.2 vorgestellte O(n)-Algorithmus verwandt. Dieser wurde quasi eins-zu-eins in SPeeDCHART eingegeben.
In der obersten Ebene wurde eine Unterteilung in die vier Zustände Eingabe, Berechnung, Ausgabe und Reset vorgenommen. Diese Zustände beinhalten jeweils einen Subgraphen, der den entsprechenden Teil des Algorithmus repräsentiert. Die Berechnung ist aufgrund ihrer Komplexität noch weiter gegliedert.

In der ersten Entwicklungsphase wurde zunächst der Algorithmus in seinen einzelnen Schritten eingegeben. Hierbei entsprach jeder Zustand dem Zustand des Algorithmus nach einem Rechenschritt, wobei ein Rechenschritt durchaus aus komplexeren Operationen bestehen konnte. In dieser Phase wurde auch noch der Datentyp float verwendet, um genauer feststellen zu können, ob die Rechnung korrekt ist.

In der nächsten Phase wurden die float-Operationen durch die entsprechenden Befehle zur Ansteuerung der Rechenwerke ersetzt. Hierbei war eine große Schwierigkeit, die Rechenwerke jeweils möglichst gut auszulasten, damit maximale Parallelität erreicht werden kann. Hierdurch ergaben sich wiederum Probleme bei der Registerzuteilung, da auch diese nach Möglichkeit optimal genutzt werden sollten.
Da einige Rechenoperationen mehrere Takte andauern, wuchs in dieser Phase die Anzahl der Zustände erheblich. Es war nötig bei jedem Takt in einen neuen Zustand zu wechseln und nicht, wie vorher, nur nach jeder Operation. Durch diesen Zuwachs an Zuständen litt die Übersichtlichkeit so erheblich, daß es eigentlich nur noch für den jeweiligen Entwickler möglich ist, das entstehende Diagramm zu verstehen. Durch diesen Umstand wurde die Gruppenarbeit erheblicht eingeschränkt. Da die Abläufe sehr stark ineinander verschränkt sind, wurde auch keine Möglichkeit gesehen, die Diagramme weiter in Subdiagramme zu unterteilen.

In der abschließenden Phase wurde nun noch der Anschluß an die SRAM-Karte (siehe Kapitel 5.3.3) realisiert. Hierzu wurde der vorher vorhandene interne Speicher aus der Beschreibung entfernt und stattdessen die Operationen zur Ansteuerung der SRAM-Schnittstelle eingesetzt.

5.3.2 Realisierung in VHDL

Zur Realisierung in VHDL läßt sich eigentlich nur kurz sagen, daß das von SPeeDCHART automatisch erstellte VHDL-File benutzt wurde. Leider konnten dieses File nicht sofort in Synopsys benutzt werden, sondern es waren noch einige systematische Fehler zu beheben. In erster Linie waren dies Änderungen der verwendeten Datentypen. Zum einen wurden Datentypen benutzt, die nicht definiert waren, weder im Code noch in der Library, und zum anderen mußten einige Datentypen abgeändert werden, um die Kompatibilität zu den anderen Komponenten herzustellen.
Danach konnte der VHDL-Code für die weitere Bearbeitung unter Synopsys benutzt werden.

5.3.3 Die SRAM-Karte

Zur Reduzierung des Resourcenverbrauchs auf dem Emulator INCA haben wir beschlossen, auf eine hardwaremäßige SRAM-Karte zurückzugreifen.

Diese Karte emuliert einen Speicher mit Lese- und Schreibzugriffen im 8-, 16- oder 32-Bit-Modus. Für unsere Projekt benutzen wir den 32-Bit-Modus, jedoch reduziert auf 24 Bit, wobei wir eben 8 Leitungen unbenutzt lassen. Damit erreichen wir eine Verkleinerung der notwendigen Schnittstelle zwischen unserer SFU und der SRAM-Karte.

Als Daten fallen sowohl die Stützstellen als auch Zwischen- und Endberechnungswerte an, die nicht gleichzeitig in den vorhandenen Registern gehalten werden können und daher auf der SRAM-Karte zwischengespeichert werden.

Für die Benutzung der Karte existiert ein kleines Paper mit den notwendigen Informationen zum Ansprechen der Ports. Darin enthalten ist natürlich auch die Bedeutung der einzelnen Pins und deren Verwendung. Für weiteres möchte ich auf dieses Paper von Wolfram Hardt verweisen.

Da wir das Projekt nicht ganz bis zur Emulation führen konnten, ist auch die praktische Einbindung der SRAM-Karte auf der Strecke geblieben. Mit SPeeDCHART wurde eine theoretische Einbindung (Files: SRAM.*) nachgebildet, jedoch nicht vollständig verifiziert.

5.4 Rechenwerke

5.4.1 Einleitung

Nachdem der Algorithmus weitgehend strukturiert war, stellte sich die Frage, welche Rechenwerke wir benötigen würden. Es stellte sich heraus, daß wir einen Multiplizierer, einen Dividierer (beide für Fließkommazahlen), und jeweils einen Integer–Addierer und einen Float–Addierer programmieren mußten. Um den Algorithmus effizient parallelisieren zu können, planten wir von jedem Rechenwerk mehrere Exemplare ein (beispielsweise wünschten wir uns bis zu 16 Multiplizierer). Da der zur Verfügung stehende Platz für ein solches Unternehmen aber nicht einmal annähernd ausreichte, waren wir später gezwungen, von jedem Rechenwerk nur ein Exemplar zu installieren.

5.4.2 Addierer

Da wir irrigerweise davon ausgingen, daß der Addierer die relativ einfachste Struktur besitzt, starteten wir, um die allgemeine Vorgehensweise zu lernen, mit diesem Rechenwerk. Der Integer–Addierer wurde unter Zusammenarbeit aller Gruppenmitglieder entworfen, was unter Verwendung des SGE relativ problemlos geschah.
Was wir im Hinblick auf den Float–Addierer nicht vorausgesehen hatten war, daß vor dem eigentlichen Addiervorgang die Exponenten angeglichen werden mußten. Dieser Schritt war sowohl beim Multiplizierer, als auch beim Dividierer nicht erforderlich, weshalb sich eines dieser Rechenwerke besser als Einstieg geeignet hätte.
Nach dem eigentlichen Additionsschritt ist es erforderlich, das Ergebnis zu normalisieren.
Der Float–Addierer wurde dann modulweise mit dem SGE erstellt. Hierfür mußten sukzessive folgende

Komponenten erstellt werden:

5.4.3 Multiplizierer

Der Multiplizierer wurde ebenso wie der Addierer unter Verwendung des SGE modulweise erstellt. Um zwei Fließkommazahlen miteinander zu multiplizieren, müssen die Mantissen multipliziert und die Exponenten addiert werden. Es wurden folgende Komponenten definiert:
Es konnte hier leider nicht der Normalisierer des Addierwerkes übernommen werden, da das Ergebnis des Multipliziervorganges eine 32 Bit lange Binärzahl ist, während der Addierer nur eine 16 Bit lange Binärzahl liefert. Der Normalisierer wurde deshalb komplett in VHDL geschrieben. Leider trat beim synthetisieren des Normalisierers ein Fehler auf, welcher auf die benutzte while–Schleife zurückzuführen war. Da der Fehler mit konventionellen Mitteln nicht zu beheben war, sahen wir uns gezwungen, die while–Schleife durch sequentielle Abfragen zu ersetzen. Dies geschah durch 32 ineinander geschachtelte if–Abfragen.
Als der Multiplizierer fertig synthetisiert war, kam von Lars Werner die Anregung, ihn doch komplett in VHDL zu programmieren, wobei der Normalisierer aus dem " modularen " Multiplizierer übernommen wird und der eigentliche Multiplizierer durch zwei Zeilen VHDL–Code zu beschreiben ist. Diese zwei Zeilen VHDL–Code ergeben sich dadurch, daß

- die Bitvektoren der Mantissen in Integer–Zahlen umgewandelt werden, diese Zahlen multipliziert werden, das Ergebnis wieder in einen Bitvektor umgewandelt wird und
- die Bitvektoren der Exponenten in Integer–Zahlen umgewandelt werden, diese Zahlen addiert werden, das Ergebnis wieder in einen Bitvektor umgewandelt wird.

Da nach der Synthese des VHDL–Multiplizierers klar war, daß dieses Rechenwerk wesentlich weniger Gatter als das modular aufgebaute benötigen würde, entschlossen wir uns dazu, diesen Multiplizierer zu verwenden.

5.4.4 Dividierer

In diesem Abschnitt soll der Dividierer vorgestellt werden. Dieses Rechenwerk wurde mit Hilfe des Tools "sge" konstruiert. Eine direkte Beschreibung in Vhdl, wie es bei den vorangehenden Rechenwerken vorgenommen wurde, war leider nicht möglich. Das Problem lag darin, daß man in Vhdl nur die Division einer Zahl durch eine Konstante beschreiben kann. Da aber der Divisor keine Konstante ist, konnten auf dieses Verfahren nicht zurckgegriffen werden. Der Dividierer wurde aus den folgenden Komponenten zusammengesetzt:

- 8–Bit–Subtrahierer
- ein Normalisierer
- Integer–Dividierer
- ein NAND2–Gatter
- ein Zero–Element
- sowie ein Clock–Signal und ein Init–Signal

Es folgt nun eine Erläuterung des Zusammenspiels der einzelnen Komponenten, und daran anschließend folgt eine Erläuterung zur Funktionsweise des Integer–Dividierers. Zuerst einmal haben wir als Eingabe je zei 16–Bit Mantissen und zwei 8–Bit Exponenten. Die beiden Mantissen werden mit dem Integer–Dividierer und die beiden Exponenten werden mit dem Subtrahierer bearbeitet. Anschließend werden beide Ergebnisse an den Normalisier übergeben. Für das endgültige Ergebnis bleibt dann noch das Vorzeichenbit zu bestimmen. Dies wurde durch eine NAND2–Verknüpfung der Vorzeichenbits des Divisors und des Dividenden realisiert. Abschließend sei noch angemerkt, daß wir für den Integer–Dividierer noch ein Clock– und ein Init–Signal benötigten. Das Clock–Signal dient dabei der Taktung

des Dividierers, während das Init–Signal der Initialisierung diente. Das oben aufgeführte Zero–Element wurde für die Initialiserung des Normalisierers gebraucht.
Die wichtigsten Bestandteile des Integer–Dividierers sind zum einem die sogenannten Dividierzellen und Register. Bei der Form des Dividierers haben wir uns auf eine abgewandelte Form des kombinatorischen Dividierers geeinigt. Diese abgewandelte Form besteht darin, daß nur zwei Stufen des kombinatorischen Dividierers realisiert wurden. Die in diesen Stufen errechneten Zwischenergebnisse werden dann in Registern gespeichert, bevor Sie weiterverwendet werden. Bei der Speicherung in Registern ist zu beachten, daß die Werte immer stabil anliegen, d.h. bevor neue Schreiboperationen erlaubt werden, muß sichergestellt werden, daß die Ergebnisse auch tatsächlich schon ausgelesen wurden. Abbildung 5.3 wird eine Dividierzelle und in Abbildung 5.4 wird ein kombinatorischer Dividierer dargestellt. Im Inneren der Zelle werden folgende Logikgleichungen betrachtet:

$$Z = X \oplus \overline{A}(Y \oplus T)$$

$$U = \overline{X}Y + \overline{X}T + YT$$

Dabei besteht zwischen a und z folgender Zusammenhang:

$$Z = \begin{cases} X - (Y + T) & : \quad falls \quad A = 0 \\ X & : \quad sonst \end{cases}$$

Der von uns realisierte Dividierer beinhaltet nur die beiden ersten Stufen des abgebildeten Dividierers. Es wurden zwei Stufen realisiert, weil die erste Stufe zur Initialisierung benötigt wird und auch eine Dividionszelle weniger hat als alle nachfolgenden. Die hier beschriebene Methode wird als Pipelining bezeichnet.

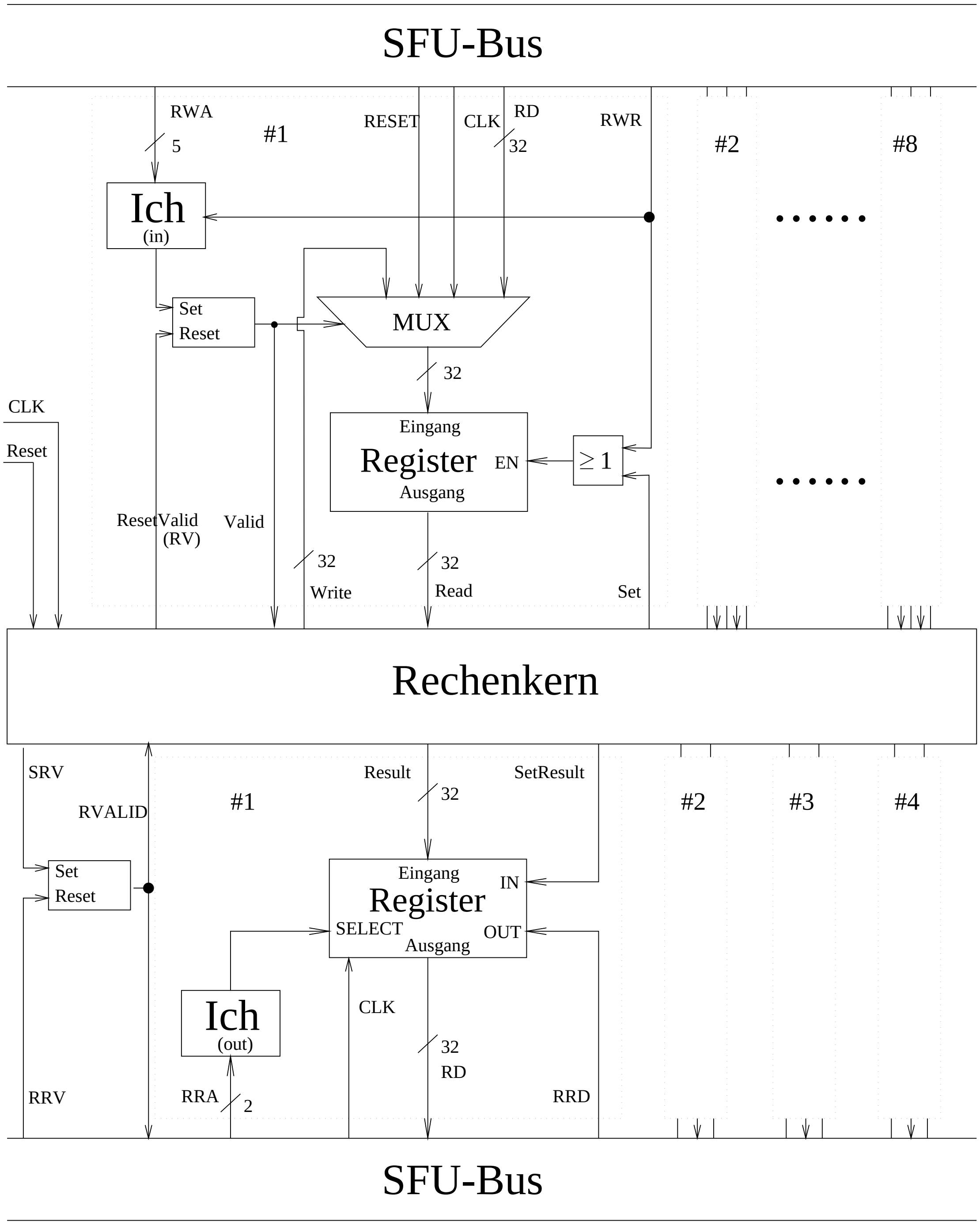

Abbildung 5.2: Schnittstelle zwischen dem SFU–Bus und dem Rechenkern

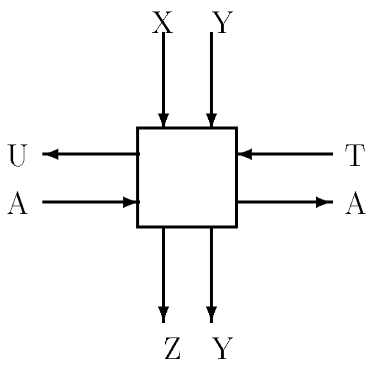

Abbildung 5.3: Einfache Divisionszelle

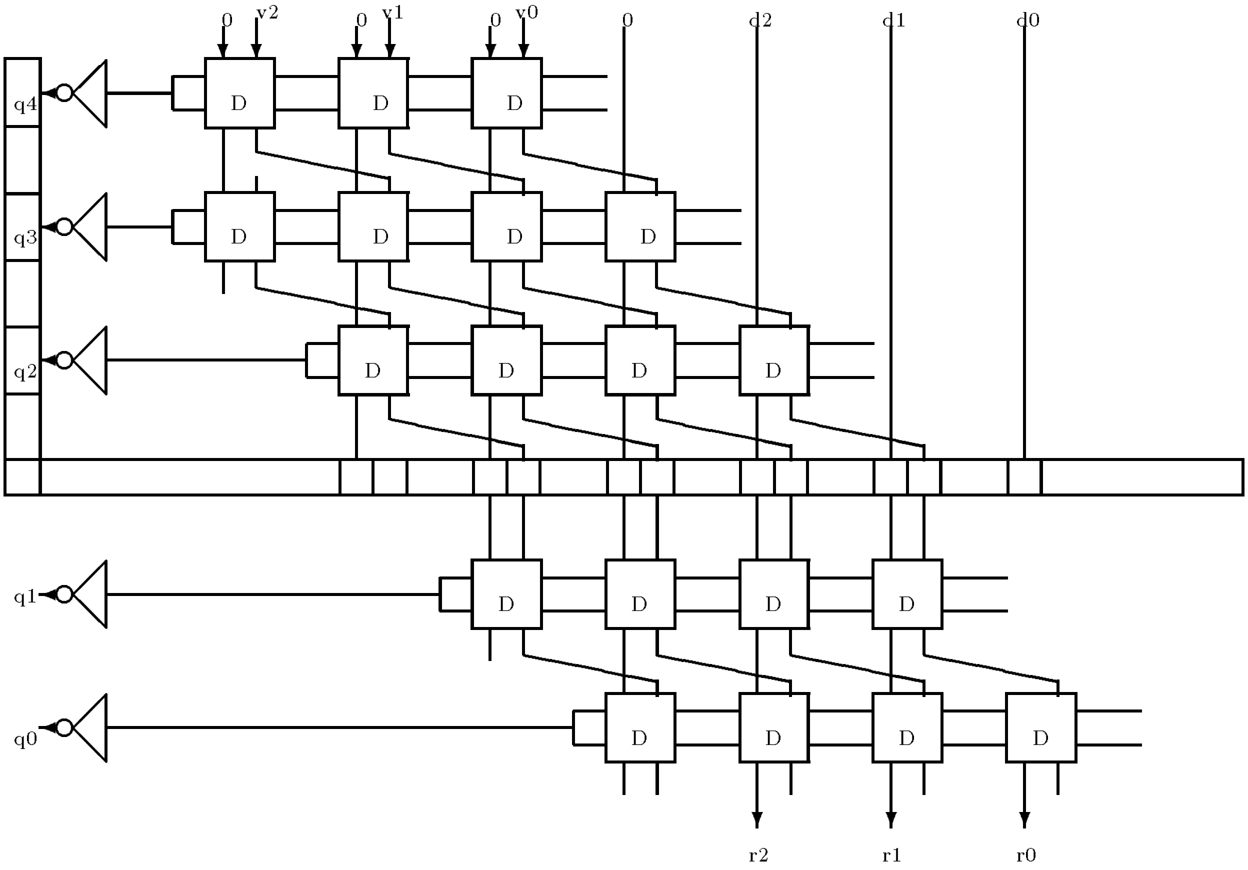

Abbildung 5.4: Kombinatorischer Dividierer

Kapitel 6

Synthese

6.1 Entwurf eines Designs mit dem SGE

Der SGE ist ein Tool, mit dem Chiplayouts graphisch erstellt werden können. Um es benutzen zu können, muß der Pfad

/homes2/zeus/synopsys/sparc/sge/bin/sge

eingebunden werden. Zusätzlich muß die Environment–Variable $SYNOPSYS auf "/homes2/zeus/synopsys" gesetzt werden. Danach muß noch eine spezielle Datei ausgewertet werden. Dies geschieht mit source $SYNOPSYS/admin/install/sim/environ.csh. Diese Datei enthält ein paar wichtige Einstellungen, die man benötigt, um mit dem SGE arbeiten zu können. Nachdem diese Vorbereitungen getroffen wurden, kann der SGE aufgerufen werden. Es öffnet sich das SGE–Fenster. Um ein Chiplayout zu ändern oder neu zu erstellen, muß der Menüpunkt Schematic Editor angeklickt werden. Wurden im aktuellen Verzeichnis bereits Layouts abgespeichert, werden diese nun angezeigt. Unterverzeichnisse werden durch eckige Klammern hervorgehoben. Soll ein neues Layout erstellt werden, muss nun der Button 'New' gedrückt werden. Soll ein bereits vorhandenes Layout bearbeitet werden, so kann dies durch Doppelklick mit der linken Maustaste oder durch einfaches Anklicken und anschließendem Drücken des nun erscheinden Buttons 'Run' geschehen. Wechseln des Verzeichnisses ist ebenfalls durch Doppelklick oder einfachem Klicken und Drücken des Buttons 'CD' möglich.
Je nach angestoßener Aktion erscheint nun ein Fenster namens Schematic Editor, welches entweder leer ist, oder das vorher ausgewählte Layout enthält. Zusätzlich erscheint ein kleines Fenster names Tool Palette. Mit den dargestellten Buttons lassen sich die gängigsten Aktionen schnell ausführen.
Bei der Erstellung eines neuen Layouts sollte auf eine hierarchische Struktur Wert gelegt werden, d.h. man überlegt sich, aus welchen Bausteinen das Layout aufgebaut ist. Falls die Möglichkeit einer Modularisierung besteht, so sollte dies genutzt werden, da bei evtl. auftretenden Fehlern das Debuggen erleichtert wird.
Kleine Tips zur Benutzung:

- Zur einfacheren Bearbeitung besteht die Möglichkeit, sich Hilfspunkte (Grid) anzeigen zu lassen: Special ⟶ Graphics Controls
- Falls die Arbeitsoberfläche zu klein ist, kann sie durch File ⟶ Sheet ⟶ Resize vergrößert werden.
- Man kann das erstellte Layout im Postscript–Format abspeichern (z.B. um es auf einem Laserdrucker auszudrucken): File ⟶ Dump Image. Man kann nun den abzuspeichernden Bereich auswählen.
- Man sollte sich mit den Hotkeys des Menüs und den Buttons der Tool–Palette vertraut machen.

Ist ein Modul eines Layouts komplett, so muß, um es weiter bearbeiten zu können, zuerst daraus ein Symbol erstellt werden: Tools ⟶ Create Symbol ⟶ This Block ⟶ Build.
Dann muß der VHDL–Code erzeugt werden: Tools ⟶ VHDL Netlist, und Tools ⟶ Verilog Netlist.

Um dieses Modul im späteren Layout benutzen zu können, ist es nun möglich, dieses Modul durch Add ⟶ Symbol einzubinden.
Diese Schritte werden nun bis zum endgültigen Layout wiederholt. Dieses Layout (wie auch jedes einzelne Modul) kann nun getestet werden:
Im SGE–Fenster wird Navigate Hierarchy angeklickt. Dann wird zu prüfende Layout ausgewählt. Es öffnen sich zwei Fenster: Hierarchy Navigator und eine Tool Palette.
Im Hierarchy Navigator klickt man zuerst Tools ⟶ Code VHDL Models, und danach Tools ⟶ Analyze VHDL Models an. (Letztere Aktion benötigt bei komplexeren Strukturen einen größeren Zeitaufwand.) Für eingehendere Tests benutzt man den VHDL–Simulator, der ebenfalls unter dem Menüpunkt Tools zu finden ist.

6.2 Der steinige Weg von VHDL nach EDIF

6.2.1 Grundsätzliches

Die Synthese ist relativ einfach. Es müssen nur einige wichtige Dinge beachtet werden, die so nirgendwo in den Handbüchern stehen. Das wichtigste ist die Auswahl der Library. Sie muß für Synopsys und für InCA existieren. Wir haben uns für die ES2 Library entschieden. Die Version, die wir benutzt haben, nennt sich *ecpd10_ind* und ist für den industriellen Einsatz konzipiert. Für diese Library sprachen zum einen die Verfügbarkeit als auch das Vorhandensein spezieller Elemente. In unserem Fall waren es die PAD–Zellen, die in der *lsi10k* leider nicht vorhanden sind. Ansonsten währe die *lsi10k* ideal gwesen, InCA kommt nur mit deren Tristates nicht zurecht.

Als Softwarepakete wurden Synopsys 3.1 – 3.3 und InCA 6.0c benutzt.

Damit die ganzen Library's und Optionen nicht bei jedem Start des Design Compiler's neu eingestellt werden müssen, sollte dafür ein Skript angelegt werden.

6.2.2 Synthese

Die Synthese kann mit Hilfe der unter X11 laufenden grafischen Oberfläche *design_analyzer* oder der im Textmodus laufenden *dc_shell* durchgeführt werden. Bei größeren Projekten sollte die *dc_shell* mit einem Skript benutzt werden, da sie ohne Probleme im Hintergrund laufen kann und dabei nicht gleich ein Monitor blockiert.

Beim *design_analyzer* können die Variablen über das Menü oder durch Einlesen eines Skriptes gesetzt werden, bei der *dc_shell* automatisch durch Angabe eines Skriptes oder interaktiv auf Prompt Ebene.

Folgendes gilt sowohl für das grafische Frontend als auch für die Shell:

Bei der Synthese wird als erstes die Library ausgewählt und eingestellt. Anschließend werden die Optionen eingegeben, die das Einlesen der VHDL Sourcen beeinflußen. In der Reihenfolge der Abhängigkeiten untereinander werden dann die VHDL Files eingelesen. Anschließend werden die generischen Beschreibungen aufgelöst. Dafür gibt es 3 Methoden. Wir verwenden *uniquify*, welche Kopien von Prozessen anlegt und eindeutig umbenennt. Dieses hat gegenüber der *ungroup* Methode den Vorteil, daß die einzelnen Prozesse einfacher wiedergefunden werden können. *Ungroup* löst die einzelnen Prozesse auf, was mehr Spielraum für die Optimierung läßt, wobei einzelne Gatter aber nachher nur noch sehr schwer den Prozessen zugeordnet werden können. Als letztes gibt es noch die *dont_touch* Methode. Nähere Informationen stehen dazu im Synopsys Tutorial. Diese Funktion sollte aber nicht benutzt werden, da dann der Synopsys Output nicht unbedingt von InCA verarbeitet werden kann.

Nach dem Setzen der Betriebsbedingungen und der Fanouts müssen die PAD–Zellen eingefügt werden. Beim Optimieren des Designs und dem Abbilden auf die Zielbibliothek sind die Parameter wichtig. Der Vorgang muß mit dem Befehl

```
compile -boundary_optimization  -map_effort high
```

gestartet werden. Bei anderen Parametern oder Stufen der Optimierung ist nicht gewährleistet, das InCA den EDIF Code von Synopsys korrekt weiterverarbeiten kann.

6.2.3 Ausgabe

Es ist nicht einfach, einen EDIF–Code von Synopsys zu bekommen, der auch von InCA verstanden wird. Dazu müssen in der *.synopsys_dc.setup* Datei folgende Befehle stehen:

```
search_path = {. \
    /homes2/zeus/EUROCHIP_CDROM_LIBs/ES2/libraries \
    /homes2/zeus/synopsys/libraries/syn }
link_library = ecpd10_ind.db
target_library = ecpd10_ind.db
symbol_library = { ecpd10_s2030.sdb }
synthetic_library = standard.sldb

alias swl set_wire_load

view_num_lines_to_auto_scroll = 0
edifout_add_power_and_ground_to_interfaces = "false"
edifout_skip_port_implementations = "false"
edifout_transform_to_origin = "false"
edifout_instantiate_ports = "true"
single_group_per_sheet = "true"
use_port_name_for_oscs = "false"
combine_vertical_logic_groups = "false"
duplicate_ports = "true"
edifout_power_and_ground_representation = "cell"
edifout_pretty_print = "true"
edifout_no_array = "true"
edifout_netlist_only = true
```

Die Datei sollte in dem Verzeichnis stehen, in der die Synthese läuft. Von einer globalen Version ist abzuraten, da sie dann auch für andere Projekte benutzt wird, für die diese Vorgaben zu Fehler führen können. Außerdem sollte sie auch nicht in einem Verzeichnis stehen, in dem das Design debugged oder simuliert wird. Bei uns traten Fehler auf, indem wir zu Ergebnissen kamen, die so definitiv nicht vom vorhandenen Programm erzeugt werden konnten. Einige Vergleiche lieferten z.B. immer den gleichen Wert zurück, was dazu führte, das Schleifen niemals terminierten oder erst gar nicht ausgeführt wurden.

6.2.4 PAD–Zellen

PAD–Zellen sind Treiber für bidirektionale Busse.

Die PAD–Zellen sind und waren das große Problem. Wenn bidirektionale Leitungen nach Außen gehen sollen, werden Tri–State–Treiber benötigt. Unter InCA müssen dies PAD–Zellen sein.

Mit diesen PAD–Zellen hat jetzt aber Synopsys Probleme. Wo es mit den Tri–State–Treibern aus der *lsi10k* Bibliothek keine Fehler gab, bekamen wir immer einen „Multiple Driver“ Fehler. Das Problem war, das auf dem bidirektionalem Datenbus einmal der Treiber für die externen Pins saß, und 3 Prozesse schreibend, bzw. in einer if–Abfrage lesend darauf zugegriffen haben.

Dieses Problem wurde gelöst, indem die Prozeßzugriffe abgekoppelt wurden. Für die lesenden Prozesse wurden neue Prozesse eingefügt, die bei Änderungen auf dem Datenbus die neuen Daten vom Bus auf ein Signal kopiert haben, auf welches die anderen Prozesse dann zugriffen.

Bei den schreibenden Prozessen war die Sache nicht ganz so einfach. Hier mußte noch ein zusätzliches Signal eingefügt werden, was dem Kopierprozeß die Erlaubnis gibt, auf den Datenbus zu schreiben. Dafür wurde das Inst–Signal abgegriffen, um nachzusehen, was für Befehle über den Bus gegangen sind.

6.2.5 Warnungen und Fehler

Zu Warnungen von Synopsys ist ganz allgemein zu sagen, das sie wie Fehler behandelt werden sollten. Wenn in einem Prozeß z.B. ein *Wait*–Statement vorkommt, welches nicht von Synopsys unterstützt wird, wird von Synopsys nur eine Warnung mit der entsprechenden Meldung ausgegeben. Intern wird aber der ganze Prozeß, in dem das *Wait*–Statement steht, gelöscht, was natürlich bei der Optimierung Auswirkung auf die Busse und anderen Prozesse hat.

Bei dem Befehl *insert_pads* empfiehlt es sich, besonderen Wert auf Fehlermeldungen und bzw. Warnungen von Synopsys zu achten. Bei etwaigen Inkonsistenzen wird die Synthese jedenfalls durchgeführt, allerdings ohne die vermeintlich spezifizierten PAD–Zellen. Damit ist das Ergebnis für Realisierungszwecke und damit auch für eine Emulation verloren.

6.2.6 Das fertige EDIF File

Mit dem Befehl

```
write -format edif -hierarchy -output sfu_ecpd10_ind.edif
```

wird die fertige Netzliste im EDIF–Format ausgegeben. Leider heißen die Zellen in der Synopsys Library etwas anders als in der InCA Library. In einem Texteditor müssen aber nur alle *LIB* entfernt werden. Das einfachste ist, einfach die 3 Buchstaben durch nichts ersetzen zu lassen, und schon ist das Eingabefile für den InCA fertig.

Kapitel 7

Portierung nach InCA

7.1 Der noch steinigere Weg von Edif nach InCA

Nachdem das entworfene Design im Edif-Format vorliegt, kommt die InCA-Software zum Einsatz: Mit ihr wird das Design auf den Hardware-Emulator abgebildet.
Hierzu startet man, nachdem man sich die entsprechenden Umgebungsvariablen gesetzt hat (source set_v6 im entsprechenden InCA-Verzeichnis, siehe Handbuch), den „browser“ mit dem Befehl

`browse -top` *filename*

Es erscheint dann die Top-Level-Ebene, und falls die entsprechende Eingabedatei korrekt eingelesen wurde kann man sie nachdem das entsprechende Icon angeklickt wurde über den Menüpunkt „Properties ⇒ Instance“ in das InCA-interne Format umsetzen. Man wählt dazu aus der Click-Box „commands“ den entsprechenden Eintrag der dem Format der eingelesenen Datei entspricht und aus der Click-Box „libraries“ die entsprechende Library mit der das Design gelinkt wurde. Hat man das Edif-File mit einem der Synopsys-Tools erstellt, so ist hier „old_edif“, und die entsprechende Library ecpd_10 zu wählen.
Man darf jedoch falls man das Edif-File mit dem „design_analyzer“ erzeugt und die entsprechende ecpd_10-Library für Synopsys verwand hat nicht vergessen vor dem Einlesen in InCA alle großgeschriebenen Teilworter „LIB“ zu entfernen. Das kommt daher, daß die ecpc_10-Library für Synopsys nicht ganz nameskompatibel zur entsprechen Library von InCA ist. So muß zum Beispiel aus einem LIBAND2 ein AND2 gemacht werden, aus einem LIBOR ein OR etc., was sich aber mit einem einfachen Editor, der Groß/Klein-Schreibung beim Suchen und Ersetzen beachtet leicht bewerkstelligen läßt. Weiterhin ist zu beachten, das etwa ein Element namens „INV“ bereits in der Library vorkommt und ein eigenes Design mit diesem Namen entsprechend umbenannt werden muß.
Nach einem Klick auf „old_edif“ müssen durch die Anwahl des „Properties“-Buttons noch einige Eigenschaften für die Übersetzung angegeben werden: ignore supply-ports ist auf „yes“ zu stellen, in der Zeile vdd-tie ist „logic_1“ und unter vss-tie „logic_0“ einzutragen. Man darf nie vergessen dann auch den Apply-Button zu drücken, da die Änderungen sonst nicht übernommen werden. Ausgeführt wird das Kommando, wie auch alle Anderen durch einen Klick auf den Execute-Button.

Hat man es so geschafft das Design in das InCA-interne Format einzulesen, so sind die nächsten Schritte in der Reihenfolge durchzuführen, wie sie nach dem Anklicken des entsprechenden Icons, und der Auswahl des Menupunktes „Properties ⇒ Instance“ und des Kommandos „auto_configure“ im Properties-Sheet zu lesen sind (einfach erst auto_configure anklicken, dann den Button Properties im Menu Properties:Instace). In einfachen Fällen ist dieser Menupunkt eine schnelle Möglichkeit das Design auf die Hardware zu portieren. Oft wird es sich jedoch nicht vermeiden lassen, etwa für Teile des Designs bestimmte Vorgaben zu machen, Teile von Hand zu partitionieren, oder zu Routen. Weitere Informationen hierzu befinden sich in den entsprechenden Handbüchern.

Kapitel 8

Funktionaler Test

Das *InCA*-Programm *tester* soll für einen funktionalen Test des Coprozessors benutzt werden, nachdem der Entwurf auf *InCA* portiert worden ist. *tester* läuft auf einem PC unter *MS–DOS*, kann aber über Netzwerk auch von einem *Unix*–System benutzt werden. Das Programm *browse*, das auf einer *SPARC* läuft, enthält zum Beispiel Funktionen, um über Netzwerk das Programm *tester* zu benutzen (siehe [RPbasics] S. 71).

tester ermöglicht es dem Benutzer, Testvektoren an *InCA* zu schicken und von *InCA* zurückgegebene Vektoren auszuwerten. Die Vektoren müssen dabei vom Benutzer entweder direkt über die Shell des Programmes eingeben oder aber in einer Datei zur Verfügung gestellt werden.

Die Erstellung der Testvektoren wird von *tester* nicht unterstützt. Entweder man erstellt die Testvektoren von Hand, was je nach Umfang der zu testenden Hardware mühsam und fehlerträchtig sein kann, oder aber man benutzt vorher Tools, die automatisch Dateien von Testvektoren erstellen.

Da bisher in der benutzten Software keine Unterstützung für *tester* gefunden werden konnte, wurden im Laufe des Projektes kleine Tools programmiert, die automatisch Testvektoren erstellen.

8.1 Starten von *tester*

Bevor *tester* benutzt werden kann, muß innerhalb des Programmes *browse* die Funktion *generate* durchgeführt werden. Diese Funktion ist i.a. die letzte, die man beim Entwurfsprozeß in *browse* ausführt. Sie erzeugt in dem Verzeichnis, in dem *browse* arbeitet, das Verzeichnis *download*. Dieses Verzeichnis enthält Dateien, ohne die *tester* nicht arbeiten kann. Siehe hierzu auch [RPbasics] S. 68.

Jetzt kann *tester* gestartet werden. Die folgende Vorgehensweise wurde im Laufe des Projektes auf einem PC mit *Sun–PC–NFS* durchgeführt. Inzwischen wird *xfstool* zum Mounten eines entfernten Dateisystems auf einem PC benutzt. Leider hat sich der Verfasser dieser Zeilen noch nicht mit *xfstool* befaßt und kann deshalb nichts darüber berichten.

1. Den PC und *InCA* einschalten.
2. Auf dem PC einloggen.
3. “net use” eintippen, um zu sehen, welche Laufwerke unbenutzt sind.
4. Einen Bezeichner eines freien Laufwerkes wählen (z.B. “m:”, falls “m” nicht benutzt wird).
5. Auf der *SPARC* feststellen, wo das Arbeitsverzeichnis von *browse* ist, und zwar genau das Verzeichnis, in welchem sich das Verzeichnis *download* befindet. Sei dieses Arbeitsverzeichnis z.B. /homes/prometheus/spline/watch.
6. Auf dem PC wird dieses Verzeichnis jetzt mit “net use” gemountet. Mit dem obigen Beispielen muß man auf dem PC “net use m: prometheus:/homes/prometheus/spline/watch” eingeben.
7. Auf dem PC “m:” eintippen.

8. Auf dem PC “dir” eintippen. Man sollte jetzt den Inhalt des Verzeichnisses auf der *SPARC* sehen, insbesondere das Verzeichnis *download*.

9. Auf dem PC “tester –server” eintippen (Achtung, wenn der Server einmal läuft, dann läuft er... nur ein Reset oder der Ausschalteknopf scheinen ihn stoppen zu können).

8.2 Bedienung von *tester*

Der Benutzer muß für *tester* zwei Dateien erstellen: ein *Vectorfile*, das die Testvektoren enthält, und ein *Mapfile*, das Signalnamen auf die Indizes des Testvektors abbildet. Zur automatisierten Erstellung dieser Dateien siehe Abschnitt 8.3.

Das Vectorfile des *watch*-Beispieles ([RPbasics]) heißt *vectors.vec* und enthält folgende Zeilen:

```
<HHHH;
<HHHL;
<HHHH;
<HHHL;
<HHHH;
<HHHL;
<HHHH;
<HHHL;
<HHHH;
<HHHL;
<HHHH;
<HHHL;
<HHHH;
<HHHL;
<HHHH;
<HHHL;
<HHHH;
<HHHL;
<HHHH;
<HHHL;
<HHHH;
<HHHL;
<HHHH;
<HHHL;
<HHHH;
<HHHL;
```

Und das Mapfile des *watch*-Beispieles heißt *names.map* und sieht so aus:

```
CLK 3;
TA1 1;
TA2 2;
TST 0;
```

Das Mapfile gibt an, welches Signal an welcher Position des Vektors liegt. *CLK* liegt im obigen Beispiel an Position 3, und entsprechend ändert das Signal im Vektor ständig seinen Wert, was man am Vektorfile erkennen kann.

Die Stellen des Vektors sind von links mit 0 anfangend durchnumeriert, Position 3 ist bei den Vektoren in diesem Beispiel also immer rechts außen (vor dem Semikolon).

Die Zeichen der Vektoren können zwei verschiedene Bedeutungen haben. Zum einen kann der Wert eines Signal an *InCA* gesendet werden. Diesen Vorgang bezeichnet man als *drive*. Zum anderen kann der Wert eines von *InCA* gesendeten Signals empfangen werden. Diesen Vorgang bezeichnet man als *strobe*. Der Wert eines gestrobeden (also empfangenen) Signals kann auf seine Korrektheit überprüft werden, indem im Vektor der Signalwert angegeben wird, den man erwartet. Entspricht der Wert des gestrobeden Signals dem erwarteten Wert, so fährt *tester* normal fort. Weicht der Wert ab, so gibt *tester* eine Fehlermeldung aus.

Ein Zeichen in einem Vektor bedeutet also entweder, daß ein bestimmter Signalwert an *InCA* geschickt werden soll, oder aber das Zeichen gibt einen Referenzwert für ein Signal an, den man von *InCA* erwartet.

Folgende Zeichen können in Vektoren verwendet werden:

H – der logische Wert 1 wird auf diesem Signal gedrived

L – der logische Wert 0 wird auf diesem Signal gedrived

1 – das Signal wird gestrobed, und es wird der logische Wert 1 erwartet

0 – das Signal wird gestrobed, und es wird der logische Wert 0 erwartet

Z – das Signal wird gestrobed, und es wird der logische Wert Z erwartet (hochohmiger Zustand in Tristate-Logik)

$*$ – das Signal wird gestrobed, das Ergebnis wird ignoriert

Man beachte, daß der logische Wert Z nur empfangen, nicht gesandt werden kann.

Vor den Vektoren steht je ein Zeichen, das angibt, was mit dem Vektor getan werden soll:

< – *Vector Drive*. H und L werden gedrived; 0,1,Z werden ignoriert.

> – *Vector Strobe*. 0,1,Z werden gestrobed; H und L werden ignoriert.

– – *Vector Drive and Strobe*. H und L werden gedrived; 0,1,Z werden gestrobed.

Eine ausführlichere Beschreibung des Programmes *tester* befindet sich in [Tester].

Hat man nun ein Vectorfile und ein Mapfile erstellt und wurde das *download*-Verzeichnis mittels *generate* im Programm *browse* erzeugt, so kann man *tester* starten (siehe Abschnitt 8.1) und von *browse* aus steuern (siehe [RPbasics] S. 71).

8.3 Tools zum Erzeugen von Testvektoren

Die Erstellung des Vectorfiles und des Mapfiles von Hand ist bei größeren Beispielen (wie zum Beispiel bei unserem Coprozessor) ein sehr langwieriges und fehlerträchtiges Unterfangen.

Deswegen wurden die Tools programmiert, die mit Hilfe des VHDL-Simulators *vhdldbx* und einer vom Benutzer erstellten Datei, die Informationen über die zu testenden Signale enthält, ein Vectorfile und ein Mapfile erzeugen, das von *tester* benutzt werden kann.

8.3.1 Zur Software

Die benötigte Software wurde während des Projektverlaufs in dem Verzeichnis *tester* untergebracht. Dieses Verzeichnis enthält die Programme *mkmon* und *mkvec*, mit deren Hilfe man

1. mit *vhdldbx* Signaländerungen in einer Datei abspeichern und
2. unter Zuhilfenahme dieser Datei für den funktionalen Tester *tester* Vector– und Mapfiles erzeugen kann.

Falls *mkmon* oder *mkvec* nicht vorhanden sind, einfach "make" eintippen.

8.3.2 Die Signaldeklarationsdatei (SDD)

Sowohl *mkmon* als auch *mkvec* benutzen als Eingabe eine Datei, in der sämtliche Signale aufgeführt sind, deren Werte von *tester* an *InCA* entweder gesendet ("drive") oder aber als Output von *InCA* gelesen und auf ihre Korrektheit überprüft werden ("strobe").

Zu jedem Signal wird außerdem angegeben, unter welchen Umständen ein Wert über dieses Signal entweder gesendet oder aber gelesen wird.

8.3.2.1 Die Syntax der SDD

```
input: sigdef
| input sigdef
;

sigdef: SIGNAL drivers
;

drivers: /* empty */
| drivers modeplusname
;

modeplusname: MODE DBXSIGNAL
;
```

SIGNAL ist ein String, der den Namen des Signals enthält. Die Liste aller möglichen Namen findet man in download/tester.tnm nachdem man in *browse* die Aktion *generate* durchgeführt hat. Man kann die Liste der möglichen Signalnamen aber auch schon eher finden, und zwar muß man dafuer in *browse* einfach *View* und dann *Show port names* anklicken.

MODE ist entweder *drive* oder *strobe*.

DBXSIGNAL ist der Name das Signals unter *vhdldbx*, dessen Signalverlauf beobachtet werden soll und dessen Werte unter dem Namen von SIGNAL im Programm *tester* gedrived oder gestrobed werden soll. DBXSIGNAL muss sich immer auf ein Signal beziehen, dass unter *vhdldbx* nur die Werte 0,1 oder *Z* annimmt. Insbesondere kann es kein Array sein. Stattdessen müssen die Elemente eines Arrays einzeln über den Index adressiert werden. Beispiel:

```
D: drive /SFU_TEST/D           /* falsch */

D<0>: drive /SFU_TEST/D(0)     /* richtig */
D<1>: drive /SFU_TEST/D(1)
.
.
D<31>: drive /SFU_TEST/D(31)
```

8.3.2.2 Die Semantik der SDD

Der einfachste mögliche Eintrag in der SDD sieht zum Beispiel so aus:

```
Blafasel:
```

Ein Signalname (gefolgt von einem ":") ohne weitere Angaben bewirkt, daß dieses Signal wohl in den Vektor von *tester* aufgenommen wird, ansonsten aber nur ignoriert wird, was sich darin äußert, daß immer nur der Wert "∗" für dieses Signal im Testvektor auftaucht.

Ein Eintrag der Form

```
CLK:    drive   /SFU_TEST/CLK
```

hat folgende Bedeutung: Das Signal /SFU_TEST/CLK soll unter *vhdldbx* gemonitort (also sein Werteverlauf aufgezeichnet) werden. Der Wert des Signals unter *vhdldbx* wird dann in *tester* über den Signalnamen "CLK" an *InCA* gesendet.

Ein Eintrag der Form

```
CNULL:  strobe  /SFU_TEST/CNULL
```

hat fast die gleiche Wirkung wie obige Version mit *drive*, mit dem Unterschied, daß die Werte von /SFU_TEST/CNULL nicht an *InCA* gesendet werden, sondern daß geprüft wird, ob die *Ausgabe* von *InCA* im Vektor an der Stelle, die CNULL entspricht, den gleichen Verlauf hat wie /SFU_TEST/CNULL. Diese Überprüfung führt *tester* selbständig durch, und falls die Ausgabewerte von den erwarteten Werten abweichen, wird dies durch Fehlermeldungen angezeigt.

Ob ein Signal nun gedrived oder gestrobed werden soll, kann man in den meisten Fällen an der Definition der Ports unter VHDL erkennen. "in"-Signale koennen nur mit *drive* benutzt werden, "out"-Signale nur mit *strobe*. In obigen Beispielen war CLK ein "in"-Signal, CNULL ein "out"-Signal.

Etwas schwieriger ist die Situation bei Signalen, die sowohl gedrived als auch (zu einem anderen Zeitpunkt) gestrobed werden sollen. Dies trifft z.B. auf Signale eines Datenbusses zu, der als Schnittstelle fuer die Ein– und Ausgabe von Daten dienen.

In diesem Fall reicht es nicht aus, nur *ein* Signal unter *vhdldbx* zu monitoren, da eine Signaländerung an diesem Signal nicht erkennen läßt, von welcher Seite diese Änderung erfolgte, ob es also Input oder Output ist.

Stattdessen müßen mindestens zwei Signale gemonitort werden. Eines, dessen Änderung eine Eingabe impliziert, und mindestens eins, dessen Änderung eine Ausgabe impliziert. Wenn dann im Laufe der Simulation das erste Signal seinen Wert ändert, das zweite aber nicht, so soll der Wert des ersten Signals gedrived werden. Wenn aber das zweite Signal seinen Wert ändert, soll grundsätzlich gestrobed werden. Dies erreicht man so:

```
D_31_:  strobe  /SFU_TEST/TEST_SFU/CONTROL/WRI/D(31)
        drive   /SFU_TEST/D(31)
```

Die Semantik hierbei ist folgende:
Um für *tester* den Zustand des Signals D_31_ zu ermitteln, wird zu jedem Zeitpunkt als erstes geprüft, ob /SFU_TEST/TEST_SFU/CONTROL/WRI/D(31) seinen Wert geändert hat. Ist dies der Fall, so wird der neue Wert von /SFU_TEST/TEST_SFU/CONTROL/WRI/D(31) fuer das Signal D_31_ gestrobed. Falls der Wert nicht geändert wurde, wird geprüft, ob /SFU_TEST/D(31) seinen Wert geändert hat. Ist dies der Fall, so wird der neue Wert von /SFU_TEST/D(31) fuer das Signal D_31_ gedrived. Falls keine Signaländerung vorliegt, wird im Vektor an dieser Stelle der alte Wert noch einmal benutzt. Es sei denn, dieses Signal ist noch undefiniert, in diesem Fall wird in den Vektor ein "$*$" geschrieben.

Die Liste der "strobe/drive DBXSIGNAL"–Einträge kann beliebig lang sein. Allgemein ist die Semantik so, daß fuer die Ermittelung des Wertes eines Signals (in diesem Fall D_31_) zu einem bestimmten Zeitpunkt die Liste der Dbxsignale von oben nach unten durchlaufen wird, und sobald ein Dbxsignal gefunden wird, bei dem zu diesem Zeitpunkt eine Wertänderung vorliegt, wird dieses Signal gedrived oder gestrobed (je nachdem, ob davor drive oder strobe steht), und dann wird der aktuelle Wert des nächsten Signals ermittelt.

Längere Listen können zum Beispiel dann sinnvoll sein, wenn es mehrere Instanzen gibt, die einen Bus zur Datenausgabe beschreiben. Beispiel:

```
D_31_:  strobe  /SFU_TEST/TEST_SFU/CONTROL/WRI/D(31)
        strobe  /SFU_TEST/TEST_SFU/CONTROL/EXE/D(31)
        strobe  /SFU_TEST/TEST_SFU/CONTROL/DEC/D(31)
        drive   /SFU_TEST/D(31)
```

für den Fall, daß die Entities WRI, EXE und DEX sich den Bus für die Ausgabe teilen.

8.3.2.3 Tutorial

Nehmen wir an, dass dies unsere Signaldeklarationsdatei ist, und zwar unter dem Namen *signals.src*: (statt D_0_, D_1_, usw. müßten die Signale wahrscheinlich eher D<0>, D<1>, usw. heißen, das hängt halt davon ab, wie die Namen in download/tester.tnm aussehen)

```
CLK:    drive   /SFU_TEST/CLK

INST:   drive   /SFU_TEST/INST

CNULL:  strobe  /SFU_TEST/CNULL

CINS1:  drive   /SFU_TEST/CINS1

CINS2:  drive   /SFU_TEST/CINS2

nMDS:   drive   /SFU_TEST/nMDS

FLUSH:  drive   /SFU_TEST/FLUSH

CCC_0_: strobe  /SFU_TEST/CCC(0)
CCC_1_: strobe  /SFU_TEST/CCC(1)

CCCV:   strobe  /SFU_TEST/CCCV

nRESET: drive   /SFU_TEST/nRESET

nCHOLD: strobe  /SFU_TEST/nCHOLD

nBHOLD: drive   /SFU_TEST/nBHOLD

nFHOLD: drive   /SFU_TEST/nFHOLD

FCCV:   drive   /SFU_TEST/FCCV

nMHOLDA:        drive   /SFU_TEST/nMHOLDA

nMHOLDB:        drive   /SFU_TEST/nMHOLDB

nCP:    strobe  /SFU_TEST/nCP

D_31_:  strobe  /SFU_TEST/TEST_SFU/CONTROL/WRI/D(31)
        drive   /SFU_TEST/D(31)
D_30_:  strobe  /SFU_TEST/TEST_SFU/CONTROL/WRI/D(30)
        drive   /SFU_TEST/D(30)
.
.
.
usw.
.
.
.

D_0_:   strobe  /SFU_TEST/TEST_SFU/CONTROL/WRI/D(0)
        drive   /SFU_TEST/D(0)
```

Damit erzeugen wir zunächst fuer *vhdldbx* ein Skriptfile, mit dem sich alle benötigten Signale einfacher monitoren lassen.

Dazu geben wir ein:

```
mkmon signals.src > mon.incl
```

Die Datei *mon.incl* enthält jetzt das Skript und sieht so aus:

```
set watchsigs /SFU_TEST/CLK /SFU_TEST/INST /SFU_TEST/CNULL /SFU_TEST/CINS1
/SFU_TEST/CINS2 /SFU_TEST/nMDS /SFU_TEST/FLUSH /SFU_TEST/CCC(0) /SFU_TEST/
CCC(1) /SFU_TEST/CCCV /SFU_TEST/nRESET /SFU_TEST/nCHOLD /SFU_TEST/nBHOLD /
SFU_TEST/nFHOLD /SFU_TEST/FCCV /SFU_TEST/nMHOLDA /SFU_TEST/nMHOLDB /SFU_TE
ST/nCP /SFU_TEST/TEST_SFU/CONTROL/WRI/D(31) /SFU_TEST/D(31) /SFU_TEST/TEST
_SFU/CONTROL/WRI/D(30) /SFU_TEST/D(30) /SFU_TEST/TEST_SFU/CONTROL/WRI/D(29
) /SFU_TEST/D(29) /SFU_TEST/TEST_SFU/CONTROL/WRI/D(28)  /SFU_TEST/D(28)/SF
U_TEST/TEST_SFU/CONTROL/WRI/D(27) /SFU_TEST/D(27) /SFU_TEST/TEST_SFU/CONTR
OL/WRI/D(26) /SFU_TEST/D(26) /SFU_TEST/TEST_SFU/CONTROL/WRI/D(25) /SFU_TES
T/D(25) /SFU_TEST/TEST_SFU/CONTROL/WRI/D(24) /SFU_TEST/D(24) /SFU_TEST/TES
T_SFU/CONTROL/WRI/D(23) /SFU_TEST/D(23) /SFU_TEST/TEST_SFU/CONTROL/WRI/D(2
2) /SFU_TEST/D(22) /SFU_TEST/TEST_SFU/CONTROL/WRI/D(21) /SFU_TEST/D(21) /S
FU_TEST/TEST_SFU/CONTROL/WRI/D(20) /SFU_TEST/D(20) /SFU_TEST/TEST_SFU/CONT
ROL/WRI/D(19) /SFU_TEST/D(19) /SFU_TEST/TEST_SFU/CONTROL/WRI/D(18) /SFU_TE
ST/D(18) /SFU_TEST/TEST_SFU/CONTROL/WRI/D(17) /SFU_TEST/D(17) /SFU_TEST/TE
ST_SFU/CONTROL/WRI/D(16) /SFU_TEST/D(16) /SFU_TEST/TEST_SFU/CONTROL/WRI/D(
15) /SFU_TEST/D(15) /SFU_TEST/TEST_SFU/CONTROL/WRI/D(14) /SFU_TEST/D(14) /
SFU_TEST/TEST_SFU/CONTROL/WRI/D(13) /SFU_TEST/D(13) /SFU_TEST/TEST_SFU/CON
TROL/WRI/D(12) /SFU_TEST/D(12) /SFU_TEST/TEST_SFU/CONTROL/WRI/D(11) /SFU_T
EST/D(11) /SFU_TEST/TEST_SFU/CONTROL/WRI/D(10) /SFU_TEST/D(10) /SFU_TEST/T
EST_SFU/CONTROL/WRI/D(9) /SFU_TEST/D(9) /SFU_TEST/TEST_SFU/CONTROL/WRI/D(8
) /SFU_TEST/D(8) /SFU_TEST/TEST_SFU/CONTROL/WRI/D(7) /SFU_TEST/D(7) /SFU_T
EST/TEST_SFU/CONTROL/WRI/D(6) /SFU_TEST/D(6) /SFU_TEST/TEST_SFU/CONTROL/WR
I/D(5) /SFU_TEST/D(5) /SFU_TEST/TEST_SFU/CONTROL/WRI/D(4) /SFU_TEST/D(4) /
SFU_TEST/TEST_SFU/CONTROL/WRI/D(3) /SFU_TEST/D(3) /SFU_TEST/TEST_SFU/CONTR
OL/WRI/D(2) /SFU_TEST/D(2) /SFU_TEST/TEST_SFU/CONTROL/WRI/D(1) /SFU_TEST/D
(1) /SFU_TEST/TEST_SFU/CONTROL/WRI/D(0) /SFU_TEST/D(0)
set outfile monitors                 -- write monitorresults into this
                                -- file

set outdevtag tmon                     -- devicetag for monitorresult-
                                       -- file

set mprefix M                          -- prefix of monitornames

open $outdevtag $outfile               -- open outputfile
delete *                               -- remove all monitors
foreach a in $watchsigs
monitor -c -e -o $outdevtag -n $mprefix -x ^fprint "time: %t signal:%%t val
ue: %r\n" \$now $a $a^ event $a
end
```

Jetzt starten wir *vhdldbx* und laden das entsprechende VHDL-Testprogramm in den Simulator. Dazu geben wir unten im Eingabefenster

```
include mon.incl
```

ein. Dadurch werden die Monitore für die einzelnen Signale gesetzt, wovon wir uns durch Eingabe des Kommandos "list" ueberzeugen können.

Jetzt lassen wir die Simulation einfach mittels "run" nach Belieben einige Zeit lang laufen. Dabei werden die Signaländerungen der zu monitorenden Signale in der Datei

```
monitors
```

mitgeloggt.

Die Einträge der Datei *monitors* sehen so aus:

```
time: 0 signal: /SFU_TEST/nRESET value: '0'
time: 0 signal: /SFU_TEST/FLUSH value: '0'
time: 0 signal: /SFU_TEST/nMDS value: '1'
time: 0 signal: /SFU_TEST/nMHOLDB value: '1'
time: 0 signal: /SFU_TEST/nMHOLDA value: '1'
time: 0 signal: /SFU_TEST/FCCV value: '1'
time: 0 signal: /SFU_TEST/nFHOLD value: '1'
time: 0 signal: /SFU_TEST/nBHOLD value: '1'
time: 0 signal: /SFU_TEST/CLK value: '0'
time: 0 signal: /SFU_TEST/CINS2 value: '0'
time: 0 signal: /SFU_TEST/CINS1 value: '0'
.
.
.
usw.
```

Nun benutzen wir *mkvec*, um ein Vektor- und ein Mapfile für *tester* zu erzeugen. Dazu einfach

```
mkvec signals.src monitors
```

eingeben. Dadurch werden die Dateien *names.map* und *vectors.vec* erzeugt.

names.map ist das Mapfile fuer Tester. Es enthält Angaben darüber, auf welche Elemente des Vektors die einzelnen Signale abgebildet werden. Das erste Signal, das in *signals.src* deklariert wurde, steht im Vektor äußerst links, das letzte Signal äußerst rechts.

names.map sieht etwas so aus:

```
CLK 0;
INST 1;
CNULL 2;
CINS1 3;
CINS2 4;
nMDS 5;
FLUSH 6;
CCC_0_ 7;
.
.
usw.
.
.
END
```

Und *vectors.vec* sieht so aus:

```
-LL*LLHL001L*HHHHH0ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ;
-HL*LLHL001L*HHHHH0ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ;
-LL*LLHL001L*HHHHH0ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ;
-HL*LLHL001L1HHHHH0ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ;
-LL0LLHL001L1HHHHH0ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ;
-HL0LLHL001H1HHHHH0ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ;
-LL0LLHL001H1HHHHH0ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ;
-HL0LLHL001H1HHHHH0ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ;
-LH0LLHL001H1HHHHH0HHLLLLLHHLLLLLLHLHHLLLLLLLLLLLLL;
-HH0LLHL001H1HHHHH0HHLLLLLHHLLLLLLHLHHLLLLLLLLLLLLL;
-LL0HLHL001H1HHHHH0LHLHLHLHLHLHLHLHLHLHLHLHLHLHLHLH;
-HL0HLHL001H1HHHHH0LHLHLHLHLHLHLHLHLHLHLHLHLHLHLHLH;
-LL0LLHL001H1HHHHH0HLHLHLHLHLHLHLHLHLHLHLHLHLHLHLHL;
-HL0LLHL001H1HHHHH0HLHLHLHLHLHLHLHLHLHLHLHLHLHLHLHL;
-LL0LLHL001H1HHHHH0LLLLLLLLLLLLLLLLLLLLLLHLLLLLLLLL;
-HL0LLHL001H1HHHHH0LLLLLLLLLLLLLLLLLLLLLLHLLLLLLLLL;
```

```
-LH0LLHL001H1HHHHH0HHLLLLHHHLLLLLLHLHHLLLLLLLLLLLLLLL;
-HH0LLHL001H1HHHHH0HHLLLLHHHLLLLLLHLHHLLLLLLLLLLLLLLL;
-LL0HLHL001H1HHHHH0LHLHLHLHLHLHLHLHLHLHLHLHLHLHLHLHLH;
-HL0HLHL001H1HHHHH0LHLHLHLHLHLHLHLHLHLHLHLHLHLHLHLHLH;
.
.
usw.
```

Mit *names.map* und *vectors.vec* kann man jetzt *tester* starten.

Kapitel 9

Verbindung von Inca und Sparc

9.1 Hardwareaspekte

9.1.1 Der Adapter

Um den fertigen Coprozessor an eine SUN–Worksation anschließen zu können, war entweder ein Motherboard mit Coprozessorsockel nötig oder die Signallleitungen mußten direkt an der Integer–Unit abgegriffen werden. Da es schon schwer genug war ein Board zu finden, das überhaupt einen Prozessor mit dem notwendigen Coprozessor–Interface hatte, stellten wir danach die Suche nach dem Board ein und gaben uns mit der Lösung zu Frieden die Signalleitungen direkt an der Integer–Unit abzugreifen. Dazu wurde ein Adaptersockel bestellt, der speziell für den CY7C601 von CYPRESS SEMICONDUCTOR's konzipiert ist. Dieser Prozessor ist pinkompatibel zu dem vorhandenen L64811 von LSI.
Die Verbindung zwischen INCA und dem Adaptersockel wollten wir nach ausgiebigen Messungen mit einem twisted–pair–Flachbandkabel realisieren. Zur Wahl standen die fliegende Verdrahtung mit einfacher Litze und die Verdrahtung mit normalem Flachbandkabeln. Als Lösung mit der geringsten Dämpfung erwies sich jedoch das twisted–pair–Flachbandkabel.

9.1.2 Umleitung des INULL-Signals

Am Adapter standen fast alle Signale zum Anschluß des Coprozessors zur Verfügung, jedoch nicht das CNULL Signal, da dies nur vom Chipsatz des Motherboards benötigt wird. Dieses Signal wird aber beötigt, um die Pipeline der IU und der FPU anzuhalten. Am Adapter liegt aber das INULL–Signal an, welches laut Prozessorhandbüchern die gleiche Funktion hat. Leider ist dieses Signal high–aktiv, d.h. es liegt im inaktiven Zustand auf Masse und kann daher mit einer TTL-Logik nicht mehr auf logisch '1' gezogen werden.
Wir haben dieses Problem durch einschleifen eines schnellen TTL–ODER–Gatters in diese Leitung gelöst. Dies geschah durch die Unterbrechung der Leitung am Adaptersockel, indem einfach ein Pin aus dem Sockel gedrückt wurde. Die Verbindung wurde dann, wie Abbildung 9.1 zeigt, wieder hergestellt und so die neue Signalleitung eingeschliffen.

9.1.3 Taktungsfrequenz

Die SUN–Workstation mit der L64811 Integer–Unit läuft ab Hersteller mit einer Taktfrequenz von 40 MHz. Der INCA Emulator kann wohl bei einer sehr kleinen Logik bis weit über 100 MHz emulieren, jedoch nimmt diese Maximalfrequenz mit steigender Komplexität der Logik ab.
Da wir befürchten, die SUN sei zu schnell für die Emulation unseres komplexen Prozessors, versuchten wir, die Taktfrequenz der SUN zu verringern. Laut dem Prozessorhandbuch von LSI läßt sich der L64811 mit 1 MHz takten.
Es wurden Quarzoszillatoren beschafft, die eine geringere Taktfrequenz als die vom Hersteller vorgesehenen 80 MHz hatten. Mit einem 40 MHz Oszillator konnten wir die Workstation noch booten,

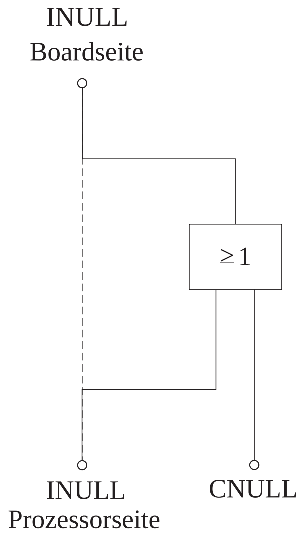

Abbildung 9.1: Erzeugung des CNULL–Signals

mit weniger jedoch nicht. Es kam bei weniger als 40 MHz zu Schwankungen in der Drehzahl des Netzteillüfters. Wie wir von der Firma SUN, erfuhren kann diese Workstation nicht mit weniger als dem 40 MHz Oszillator betrieben werden, da diese Frequenz nochmal geteilt wird und der S–BUS mit mindestens 20 MHz betrieben werden muß.
Es blieb uns also nichts anderes übrig, als unseren Coprozessor später so zu optimieren, daß er mit 20 MHz getaktet werden kann.

9.2 Softwareaspekte

9.2.1 Assemblerbefehle

Zur Ansteuerung des Coprozessors werden die *SPARC*-Assemblerinstruktionen LDC und STC benutzt. LDC dient dazu, den Inhalt einer Speicherzelle im Hauptspeicher in ein Coprozessor–Register zu laden, STC dient dazu, den Inhalt eines Coprozessor–Registers in den Hauptspeicher zu schreiben.

Die Syntax des LDC–Befehls sieht bei dem Assembler *as* so aus:

ld [Adresse],<Coprozessor–Register>

wobei <Coprozessor–Register> ein "%c", gefolgt von der Nummer des Coprozessor–Registers, ist, also %c0, %c1, %c2, usw.

Beispiel aus einem Assemblerprogramm:

```
ld [%fp+68],%c1
```

Die Syntax des STC–Befehls sieht bei dem Assembler *as* entsprechend so aus:

st <Coprozessor–Register>,[Adresse]

Beispiel aus einem Assemblerprogramm:

```
st %c0,[%fp-12]
```

Eine Dokumentation dieser und anderer *SPARC*–Assemblerinstruktionen findet man zum Beispiel in [ArchMan].

Diese Befehle kann man nun als Inline–Code innerhalb von C– bzw. C++–Programmen benutzen, und zwar mit Hilfe des *asm*–Befehls der *gcc*– und *g++*–Compiler von *Gnu*. Beispiele für dessen Benutzung kann man in den Anhängen B.1 und C.2 finden. Eine Dokumentation des *asm*–Befehls findet man in [WWWasm].

9.2.2 Manipulationen an *SunOs*

Nachdem der Adapter zum Anschluß des Coprozessors auf unserer *SPARC* installiert war, wurden die ersten Versuche durchgeführt, Coprozessor–Instruktionen auf der *SPARC* durchzuführen. Dafür wurde der Pin an der IU, der der IU die Anwesenheit eines Coprozessors anzeigt, auf logisch 1 gesetzt, und ein C–Programm führte einen einfachen LDC–Befehl aus (siehe Funktion *sendLDC* im Programm *coprotest.c*, Anhang B.1).

Zu unserer Bestürzung erzeugte diese Instruktion aber nichts weiter als eine "Illegal Instruction"–Meldung des Betriebssytems (BS). Die Ursache hierfür war, daß zur Benutzung eines Coprozessors nicht nur der Pin an der IU, sondern auch das Coprozessor–Enable–Bit im Prozessorstatusregister (PSR) der IU gesetzt werden muß, und *SunOs* setzt dieses Bit beim Start eines jeden Prozesses auf 0.

Dieses Bit im PSR kann man aber nur dann setzen, wenn man im Supervisor–Mode ist. Dieser Modus ist aber dem BS vorbehalten, und es gibt keine vorhandene Funktion, mit der man *SunOs* dazu bewegen könnte, dieses Bit zu setzen. Man muß also das BS ändern. Leider standen uns die *SunOs*–Sourcen nicht zur Verfügung (siehe Abschnitt 2.3.1.1). Deswegen wurde eine kleine Änderung direkt im ausführbaren Code (!) von *SunOs* (/vmunix) durchgeführt.

In den Supervisor–Mode gelangt man nur innerhalb eines Traps. Um das Bit zu setzen, muß man also im BS einen Traphandler einfügen, der das Bit setzt, und der durch Auslösung eines Software–Traps ausgeführt wird. Zunächst ein paar Worte dazu, wie Traps von der IU und von *SunOs* behandelt werden:

Es kann 256 unterscheidbare Traps geben. Die ersten 128 davon bestehen aus sogenannten Hardwaretraps, die andere Hälfte besteht aus Softwaretraps.

Zu den Hardwaretraps zählen zum Beispiel: Hardwarereset (Trap Nr. 0), Window–Overflow (Trap Nr. 5), FP–Exception (Trap Nr. 8) et cetera. Diese Traps haben eine von der Hardware abhängige Bedeutung, bei einem bestimmten Hardwareereignis wird also unabhängig vom BS oder anderer Software der entsprechende Trap ausgelöst. Das BS muß für jeden dieser Traps entsprechende Traphandler bereitstellen.

Softwaretraps werden durch Software ausgelöst, und zwar durch *ticc*–Maschineninstruktionen (siehe hierzu auch [ArchMan]). Es gibt mehrere *ticc*–Befehle, die abhängig von bestimmten Bedingungen ein Trap auslösen. So eine Bedingung kann zum Beispiel sein, daß das Carry–Flag im Prozessorstatusregister gesetzt ist, es können insgesamt also genau die gleichen Bedingungen wie auch bei den normalen bedingten Sprungbefehlen geprüft werden. Ein *ticc*–Befehl erhält außerdem als Argument den Index des Softwaretraps, der ausgelöst werden soll. Der Indexwert liegt zwischen 0 und 127. Wird ein Softwaretrap ausgelöst, so wird 128 zu diesem Indexwert addiert, um den entgültigen Index des Traps zu erhalten (denn die Nummern der Softwaretraps liegen zwischen 128 und 255, die Nummern davor gehören zu Hardwaretraps). Beispiele für Softwaretraps sind: Division by Zero, Systemcall.

Wenn nun ein Trap ausgelöst wurde, geht die IU in den Supervisor–Mode und ermittelt die Nummer des Traps. Bei Hardwaretraps ist diese Nummer automatisch vorgegeben, bei Softwaretraps muß diese angegeben werden (siehe oben). Die IU führt nun den Code im *Traptable* aus, der dieser Nummer entspricht.

Der Traptable enthält 256 Einträge, für jeden Trap einen. Ein Eintrag umfaßt 4 32–Bit Wörter, die direkt ausführbaren Maschinencode enthalten (der Traptable enthält also *nicht* nur die Adressen der Traphandler, es erfolgt kein indirekter Sprung, sondern ein direkter Sprung in den Traptable). Ein Eintrag des Traptables enthält üblicherweise Instruktionen zum Speichern des PSRs, zum Speichern der Nummer des Traps und einen direkten Sprung zum Traphandler. Dies ist zum Beispiel der Eintrag für den "FPU–Disabled"–Trap (Nr. 4):

```
rd %psr, %l0
sethi %hi(0xf0005400), %l3
jmpl %l3+0x10, %g0
mov 4, %l4
```

Die Stelle im Hauptspeicher, wo der Traptable zu finden ist, ist im PSR angegeben. Üblicherweise ist dies die Adresse 0, man findet den Traptable also gleich am Anfang des *vmunix*–Executables, wovon man sich mit Hilfe des Disassembler *dism* überzeugen kann. "dism /vmunix" liefert folgendes:

```
.text
        b 0x41a51
        nop
        nop
        nop
        rd %psr, %l0
        sethi %hi(0xf0005400), %l3
        jmpl %l3+0x10, %g0
        mov 0x201, %l4
        rd %psr, %l0
        sethi %hi(0xf011cc00), %l3
        jmpl %l3+0x2d4, %g0
        clr %l4
        rd %psr, %l0
        sethi %hi(0xf0005400), %l3
        jmpl %l3+0x10, %g0
        mov 3, %l4
        rd %psr, %l0
        sethi %hi(0xf0005400), %l3
        jmpl %l3+0x10, %g0
        mov 4, %l4
.
.
.
```

Das, was am Anfang als ".text" beschrieben ist, belegt die Bytes 0 bis 31 und ist der Header des *a.out* Dateiformats, siehe auch "man a.out". Der Programmcode beginnt immer erst ab Byte 32.

In den Traptable kann man nun direkt ein paar Maschineninstruktionen, die das Coprozessor–Enabled–Bit setzen, hineinschreiben und durch Auslösung des entsprechenden Traps ausführen lassen. Dies muß ein Softwaretrap sein, und zum Auslösen bietet sich der Befehl *ta* (= Trap Always, es wird also immer ein Trap ausgelöst, wenn dieser Befehl ausgeführt wird) an (siehe auch Funktion *toggleC*, Anhang B.1).

Der Softwaretrap benötigt natürlich auch noch eine Nummer. Theoretisch könnte man sich eine Nummer zwischen 0 und 128 aussuchen, aber die meisten Softwaretraps werden bereits von *SunOs* benutzt. Einen Überblick verschafft die Datei /sys/sun4m/trap.h, welche die zu den einzelnen Traps gehörenden Nummern angibt, und welche außerdem folgendes verlautbart:

```
/*
 * Software trap vectors 16 - 31 are reserved for use by the user
 * and will not be usurped by Sun.
 */
```

Für den Benutzer stehen also 16 Softwaretraps zur Verfügung. Da jeder Eintrag des Traptables 4 Maschineninstruktionen enthält, könnte man also ohne Probleme ein 64–Befehle–langes Programm im BS unterbringen, das im Supervisor–Mode läuft.

Für unsere Zwecke benötigen wir noch weniger Platz. Wir verwenden nur zwei Traps: einen Trap, der das Prozessorstatuswort zurückliefert (zum Testen, ob die Änderung auch wirklich durchgeführt wurde), und den Trap, der das Coprozessor–Enabled–Bit setzt. Der erste Softwaretrap soll die Nummer 16 (0x10), der zweite die Nummer 17 (0x11) haben, siehe auch Anhang C.2 oder B.1.

Hier ist der Assemblercode (enthalten in der Datei *t.s*), den wir in den Traptable einfügen wollen:

```
.text
        rd %psr, %i0            /* Soft-Trap Nr. 0x10 (psr lesen) */
        jmp %l2
        rett %l2 + 4
        nop
        rd %psr,%o1             /* Soft-Trap Nr. 0x11 (Copro-Bit toggeln) */
        sethi %hi(0x2000), %o2
        or %o1, %o2, %o3
```

```
mov %o3, %psr
nop                          /* Soft-Trap Nr. 0x12 (unbenutzt) */
nop                          /* 3 nops nach Aenderung von psr noetig */
nop
rd %psr, %i0
jmp %l2                      /* Soft-Trap Nr. 0x13 (unbenutzt) */
rett %l2 + 4
nop
nop
```

Der Index unseres ersten Softwaretraps im Traptable ist 0x80 + 0x10 = 0x90 = 9*16. Wir berechnen nun die Nummer des Bytes im *vmunix*–Executable, ab dem wir unseren Code einfügen müssen.

$$p = \underbrace{32}_{Header} + \underbrace{9*16}_{144.\ Eintrag} * \underbrace{4}_{4\ Wörter\ pro\ Eintrag} * \underbrace{4}_{4\ Byte\ pro\ Wort} = 2336$$

Erzeugen wir nun mittels "as -o t t.s" aus unserem Assemblerprogramm *t.s* eine Datei *t* im *a.out*–Format. Der Maschinencode beginnt in *t* ab Byte 32 (wegen des Headers) und endet bei Byte 95 (16 Befehle, 4 Byte pro Befehl, also $32 + 16 * 4 - 1 = 95$).

Um den Code in *vmunix* einzufügen, benutzen wir das selbstgeschriebene Programm *fiddle* (siehe Anhang B.3). Jetzt können wir *vmunix* ändern:

```
>>cp /vmunix .
>>fiddle
File to be changed: vmunix
File to insert from: t
Name of output-file: vmunix.new
Insert which bytes from t (<from> <to>): 32 95
Insert these bytes into vmunix at byte-position: 2336
Writing bytes 0 to 2847 from vmunix to vmunix.new
Writing bytes 32 to 95 from t to vmunix.new
Writing rest from vmunix to vmunix.new
Finished.
>>chmod a+x vmunix.new
```

Nun können wir das neue Kernel auf unserer *SPARC* installieren:

```
>>/bin/su
<Passwort>
#cd /
#/bin/mv /vmunix /vmunix.old
#/bin/cp /.../vmunix.new /vmunix
#/usr/etc/halt
b
```

Falls das neue Kernel nicht laufen sollte:

```
<stop>-a
b vmunix.old -s
mv /vmunix.old /vmunix
reboot
```

Falls alles geklappt hat (bei uns hat es das jedenfalls), das Programm *coprotest* (Anhang B.1) starten, um zu sehen, ob Coprozessorbefehle ausführbar sind.

Die C++–Benutzerschnittstelle sorgt automatisch dafür, daß mittels des Softwaretraps das Enable–Coprozessor–Bit im PSR gesetzt wird, siehe hierzu Abschnitt 10.2.1 und Anhang C.2 (Funktion *enableCP*).

Kapitel 10

Benutzerschnittstelle

10.1 Einleitung

Der größte Teil des Projektes liegt im Bereich des Hardwaredesigns, nämlich im Entwurf des Coprozessors. Dieser Coprozessor soll dann, nachdem er auf *InCA* portiert worden ist, dort emuliert und wie ein hardwaremäßig vorhandener Coprozessor von einer *SPARC*–Workstation aus angesteuert werden. Dies geschieht durch die Ausführung von bestimmten Maschineninstruktionen für die Benutzung eines Coprozessors, die vom Maschinenbefehlssatz der Sparcarchitektur bereitgestellt werden.

Damit der emulierte Coprozessor unter realen Bedingungen angemessen getestet werden kann, benötigt man ein Testprogramm, das die Funktionen des Coprozessors durch Ausführung von Maschineninstruktionen flexibel und wiederholbar testet.

Dieses Programm soll in einer höheren Programmiersprache (C++) geschrieben werden, um

- Programmierfehler im Testprogramm zu minimieren und leichter entdeckbar zu machen (Fehlerminimierung) und
- kurzfristige Änderungen im Programm zu ermöglichen (Flexibilität).

Das Programm wiederum soll auf eine Bibliothek von elementaren Funktionen zur Ansteuerung des Coprozessors zugreifen. Diese Bibliothek wird angelegt, weil

- im Verlauf des Projektes schon sehr früh die benötigten Funktionen und deren Verhalten festgelegt werden können (Schnittstelle) und
- andererseits die Art der entgültigen Implementation noch offen ist, so aber auf einen späteren Zeitpunkt verschoben werden kann (Flexibilität der Implementierung) und
- keine Exceptions o.ä. seitens des Prozessors vorgesehen sind und somit Fehler des Benutzers auf Softwareebene abgefangen werden müssen (Fehlerbehandlung) und
- auf diese Weise der Coprozessor auch durch andere Programme bequem benutzt werden kann (Portierbarkeit).

ACHTUNG! Damit der Coprozessor benutzt werden kann, muß das Betriebsystem so geändert werden, daß man im Prozessorstatusregister das Enabled–Bit für den Coprozessor setzen kann. Siehe Abschnitt 9.2.2.

10.2 C–Bibliothek

In der statischen Bibliothek *libspline.a* wurden drei Objectfiles zusammengelinkt: *sfu.o*, *spline.o* und *cspline.o*.

Um die Funktionen der Bibliothek für ein Programm zu benutzen, muß man nur das entsprechende Headerfile im Sourcecode includen und die Bibliothek hinzulinken.

Objectfile	Sourcefile	Headerfile	Inhalt
sfu.o	sfu.c	sfu.h	C–Funktionen für LDC– und STC–Befehle
spline.o	spline.C	spline.h	Die Klasse *Spline* und externe Funktionen in C++, die eine komfortable Berechnung von Splines ermöglichen.
cspline.o	cspline.C	cspline.h	C–Funktionen zur Splineberechnung (für diejenigen, die nicht in C++ programmieren wollen).

Tabelle 10.1: Inhalt der Bibliothek und wichtige Files

10.2.1 C–Funktionen für LDC– und STC-Befehle

Werden lediglich einfache LDC– und STC–Befehle benötigt, so kann man anstelle von Inline–Assemblerinstruktionen auch die dafür vorgesehenen Funktionen der Bibliothek benutzen. Dazu muß nur das Headerfile *sfu.h* im Sourcecode includet und die Bibliothek später beim Linken hinzugelinkt werden.

ACHTUNG! Jeder Prozeß, der den Coprozessor benutzt, muß als erstes das Enabled–Bit für den Coprozessor im Prozessorstatuswort setzen. Dies erfolgt durch Aufruf der Funktion *enableCP*, die in der Bibliothek enthalten ist. Werden die Berechnungen über die Klasse *Spline* oder die C–Schnittstelle durchgeführt, so ist dies nicht nötig, weil dort die Funktion *enableCP* automatisch aufgerufen wird.

Folgende Funktionen werden in der Datei *sfu.h* deklariert:

Funktion:
int enableCP()

Beschreibung:
Setzt das CP–Enabled–Bit im Prozessor–Status–Register. Das neue Prozessor–Status–Wort wird als Integer zurückgegeben. Diese Funktion muß aufgerufen werden, bevor LDC– oder STC-Befehle ausgeführt werden, sonst gibt es einen "Illegal Instruction"–Fehler.

Funktion:
int getPSR()

Beschreibung:
Gibt das aktuelle Prozessor–Status–Wort als Integer zurück.

Funktion:
void resetSFU()

Beschreibung:
Sendet einen Reset–Befehl an den Coprozessor.

Funktion:
void ldcf(float f, int r)

Beschreibung:
Die Fließkommazahl f wird in das Coprozessor–Register Nr. r geschrieben. Beachte: $0 \leq r \leq 7$

Funktion:
void ldci(int i, int r)

Beschreibung:
Der Integer i wird in das Coprozessor–Register Nr. r geschrieben. Beachte: $0 \leq r \leq 7$

Funktion:
float stcf(int r)

Beschreibung:
Der Inhalt des Coprozessor–Registers Nr. r wird als Fließkomazahl zurückgegeben. Beachte: $0 \leq r \leq 3$

Funktion:
int stci(int r)

Beschreibung:
Der Inhalt des Coprozessor–Registers Nr. r wird als Integer zurückgegeben. Beachte: $0 \leq r \leq 3$

10.2.2 Die C++–Schnittstelle

Die C++–Schnittstelle stellt die Klasse *Spline* zur Verfügung. Jedes Objekt vom Typ *Spline* kann eine individuelle Menge von Stützstellen enthalten und so behandelt werden, als wäre es das einzige *Spline*–Objekt. Die Klasse sorgt automatisch dafür, daß korrekt mit dem Coprozessor kommuniziert wird und die Ergebnisse in den richtigen Objekten abgelegt werden, auch wenn mehrere Objekte den Coprozessor benutzen. Insbesondere benötigt der Benutzer keine Kenntnisse über die Ein– und Ausgabeprotokolle des Coprozessors.

spline.h enthält nur die Deklarationen der Klasse. Der Objectcode der einzelnen Methoden und Funktionen befinden sich in der Bibliothek *libspline.a*, der Sourcecode befindet sich in *spline.C*.

Solange nur *ein* Prozeß läuft, der die Bibliothek benutzt, ist eine korrekte Behandlung des Coprozessors automatisch gewährleistet. Mehrere Prozesse, die gleichzeitig laufen und die Bibliothek benutzen, machen die Ergebnisse unvorhersehbar. Diesen Mangel könnte man (in Zukunft) durch die Benutzung von Semaphoren beheben.

Die Klasse *Spline* enthält neben Konstruktoren und Destruktoren Methoden, die eine Liste von Stützstellen manipulieren: *addPt*, *delPt*, *clrPts*.

Sie enthält Methoden, die direkten Einfluß auf den Coprozessor nehmen (welche aber nicht unbedingt benutzt werden müssen): *calc*, *reset*.

Sie enthält eine Methode, um Spline–Berechnungen nicht mit dem Coprozessor, sondern mittels Software durchzuführen: *softCalc*.

Sie enthält Methoden, die bestimmte Informationen zurückliefern: *getSize*, *getPt*, *getRange*, *getCoeff*.

Außerdem enthält sie noch Methoden zur Ausgabe der Ergebnisse: *print*, *printMaplePlot*, *plot*.

Operatoren sind: << und =, und es gibt noch eine weitere externe Funktion: *SplineErrText*.

Konstruktoren, Destruktoren, Operatoren

Konstruktor:
Spline()

Beschreibung:
Erzeugt ein neues Objekt und initialisiert es.

Konstruktor:
Spline(Spline& s)

Beschreibung:
Erzeugt ein neues Objekt und initialisiert dessen Stützstellenliste mit der des Objektes s.

Destruktor:
~Spline()

Beschreibung:
Löscht ein Objekt. Falls dieses Objekt auf Ergebnisse des Coprozessors wartete, werden diese vor dem Löschen erst von Coprozessor geholt, damit dieser weitere Berechnungen durchführen kann.

Operator:
=

Beschreibung:
Das Objekt links von dem = erhält die Stützstellenliste des Objektes rechts von dem =.

Operator:
<<

Beschreibung:
Streamoperator zur Ausgabe der Daten des Objektes, siehe *print*.

Methoden zur Manipulation der Stützstellenliste

Methode:
int addPt(float x, float y)

Beschreibung:
Der Punkt (X,Y) wird in die Stützstellenliste uebernommen, falls die Liste nicht bereits das Maximum von 15 Punkten enthält und sie nicht bereits diesen Punkt oder einen Punkt mit derselben X–Koordinate enthält.

Die Stützstellenliste ist nach X–Koordinaten aufsteigend sortiert.

Ausgabe: Fehlercode.

Methode:
int delPt(float x, float y)

Beschreibung:
Falls der Punkt (X,Y) in der Stützstellenliste enthalten ist, so wird er entfernt.

Ausgabe: Fehlercode.

Methode:
int clrPts()

Beschreibung:
Aus der Stützstellenliste werden alle Einträge entfernt.

Ausgabe: Fehlercode.

Coprozessorbezogene Methoden

Methode:
int calc()

Beschreibung:

Falls die Stützstellenliste mindestens 3 Punkte enthält, werden diese an den Coprozessor übergeben.

Davor wird noch überprüft, ob ein Objekt bereits eine Coprozessor–Berechnung gestartet hat. Ist dies der Fall, so werden die Ergebnisse der laufenden Berechnung erst an das wartende Objekt übergeben, bevor die neue Berechnung mit den Stützstellen des aktuellen Objektes gestartet wird.

Diese Funktion wird automatisch aufgerufen, falls die Koeffizienten der Spline–Polynome mit *getCoeffs* geholt werden, es ist also nicht unbedingt notwendig, *calc* zu benutzen. Es ist aber sinnvoll, wenn man während der Berechnung der Koeffizienten mittels des Coprozessors gleichzeitig schon etwas anderes im Programm erledigen will.

Ausgabe: Fehlercode.

Methode:

void reset()

Beschreibung:

Beschreibung:

Schickt einen *reset*-Befehl an den Coprozessor.

Softwarelösung zur Splineberechnung

Methode:

int softCalc()

Beschreibung:

Genau wie *calc*, nur wird hier nicht der Coprozessor zur Splineberechnung hinzugezogen, sondern ein in C++ implementierter Algorithmus.

Ausgabe: Fehlercode.

Zugriff auf diverse Daten

Methode:

int getSize()

Beschreibung:

Liefert die Länge der Stützstellenliste.

Methode:

int getPt(int nr, float *x, float *y)

Beschreibung:

Die X– und Y–Koordinaten des Punktes Nr. *nr* in der nach X–Koordinaten sortierten Stützstellenliste werden in x und y zurückgegeben.

Ausgabe: Fehlercode.

Methode:

int getRange(int nr, float *x1, float *y1, float *x2, float *y2)

Beschreibung:

getRange liefert folgendes zurück:

x1 = Minimum der X-Koordinaten der beiden Stuetzstellen des Polynoms *nr*
x2 = Maximum der X-Koordinaten
y1 = Minimum der Y-Koordinaten
y2 = Maximum der Y-Koordinaten

Ausgabe: Fehlercode.

Methode:

int getCoeff(int nr, float *a, float *b, float *c, float *d)

Beschreibung:

Falls die Koeffizienten der zur aktuellen Stützstellenliste gehörenden Splinepolynome noch nicht berechnet wurden, so wird automatisch *calc* aufgerufen.

Falls die Resultate der Berechnung noch nicht aus den Resultregistern des Coprozessors geholt wurden, so wird dies jetzt automatisch getan.

Dann werden die Koeffizienten des Splinepolynoms Nr. *nr* in den Variablen a, b, c, d zurückgegeben.

Dabei ist die Semantik der Variablen so zu verstehen, daß das Polynom *nr* die Form $a * X^3 + b * X^2 + c * X + d$ hat.

Ausgabe: Fehlercode.

Methoden zur Ausgabe von Daten

Methode:

void print(ostream& out = cout)

Beschreibung:

Gibt die Daten des Objektes formatiert als ASCII–Text aus. Per Default auf der Standardausgabe, aber man kann die Ausgabe durch Angabe eines Ostreams auch umleiten.

Methode:

int printMaplePlot(ostream& out = cout)

Beschreibung:

Gibt ein Plot–Commando als ASCII–Text aus, das dann unter *Maple* eingegeben werden kann und den Spline plotten läßt. Das Commando wird per Default auf der Standardausgabe ausgegeben, aber man kann die Ausgabe durch Angabe eines Ostreams auch umleiten.

Ausgabe: Fehlercode.

Methode:

int plot()

Beschreibung:

Mittels der *system*-Funktion wird *gnuplot* aufgerufen. Ein Fenster mit dem geplotteten Spline erscheint.

Ausgabe: Fehlercode.

Fehlercodes

Funktion:
char* SplineErrText(int nr)

Beschreibung: Diese externe ("extern" im Gegensatz zu den bisherigen Funktionen, die Elementfunktionen der Klasse *Spline* sind) Funktion liefert nach Angabe eines Fehlercodes den entsprechenden Text der Fehlermeldung zurück.

Code	Kürzel	Text
0	SP_OK	Ok
1	SP_MAXSIZE_EXCEEDED	Maximum amount of points exceeded
2	SP_REDUNDANT_POINT	Redundant point
3	SP_REDUNDANT_XCOORD	Redundant X-coordinate
4	SP_NO_SUCH_POINT	No such point
5	SP_TOO_FEW_POINTS	Too few points for calculation
6	SP_ILLEGAL_POLYNOME_NR	Illegal polynome-No.
7	SP_ILLEGAL_POINT_NR	Illegal point-No.
8	SP_NOT_CALCULATED	Coefficients haven't been calculated yet

Tabelle 10.2: Fehlercodes

10.2.3 Die C–Schnittstelle

Falls jemand lieber in C programmieren möchte, steht ihm durch *cspline.h* eine reine C–Schnittstelle zur Verfügung. Ansonsten muß auch hier wieder die Bibliothek *libspline.a* hinzugelinkt werden.

Die C–Schnittstelle enthält fast die gleichen Funktionen wie die C++–Schnittstelle, nur handelt es sich diesmal ausschließlich um externe Funktionen, während diese Funktionen in der C++–Schnittstelle als Elementfunktionen von Objekten behandelt wurden. Anstatt also eine Funktion über ein Objekt aufzurufen, wird die Funktion direkt aufgerufen (so, wie es in C nunmal üblich ist).

Dies wird intern so realisiert, daß sämtliche externen C–Funktionen nichts anderes tun, als in einem *einzigen*, für den Benutzer unsichtbaren Objekt, die entsprechenden Elementfunktionen aufzurufen. So zum Beispiel besteht der Rumpf der Funktion *SplClrPts* lediglich aus einem "$return\, s.clrPts()$" und der Rumpf der Funktion $SplAddPt(float\, x, float\, y)$ aus einem "$return\, s.addPt(x, y)$", wobei s ein statisches Objekt vom Typ *Spline* ist.

Die Handhabung, die Funktionsweise und die Restriktionen der C–Schnittstelle sind daher im Prinzip dieselben wie bei der C++–Schnittstelle.

Folgende Funktionen sind in *cspline.h* deklariert:

Funktionen zur Manipulation der Stützstellenliste

Funktion:
int SplAddPt(float x, float y)

Beschreibung:
Analog zu *addPt* (10.2.2).

Funktion:
int SplDelPt(float x, float y)

Beschreibung:
Analog zu *delPt* (10.2.2).

Funktion:
int SplClrPts()

Beschreibung:
Analog zu *clrPts* (10.2.2).

Coprozessorbezogene Funktionen

Funktion:
int SplCalc()

Beschreibung:
Analog zu *calc* (10.2.2).

Funktion:
int SplReset()

Beschreibung:
Analog zu *reset* (10.2.2).

Zugriff auf diverse Daten

Funktion:
int SplGetPt(int nr, float* x, float* y)

Beschreibung:
Analog zu *getPt* (10.2.2).

Funktion:
int SplGetRange(int nr, float *x1, float *y1, float *x2, float *y2)

Beschreibung:
Analog zu *getRange* (10.2.2).

Funktion:
int SplGetCoeff(int nr, float *a, float *b, float *c, float *d)

Beschreibung:
Analog zu *getCoeff* (10.2.2).

Funktionen zur Ausgabe von Daten

Funktion:
void SplPrint()

Beschreibung:
Analog zu *print* (10.2.2).

Funktion:
int SplMaplePlot()

Beschreibung:
Analog zu *printMaplePlot* (10.2.2).

Funktion:
int SplPlot()

Beschreibung:
Analog zu *plot* (10.2.2).

<u>**Fehlercodes**</u>

Funktion:
char* SplErrText(int nr)

Beschreibung: Analog zu *SplineErrText* (10.2.2).

10.3 Testprogramm

Das Testprogramm *spline* ist das letzte Programm, daß zum Testen des Coprozessors benutzt wird. Es ermöglicht eine relativ komfortable Eingabe von Stützstellen, die Berechnung der Splinepolynome und die Ausgabe des Ergebnisses mittels *gnuplot*. Dabei wird die Bibliothek *libspline.a* hinzugezogen.

Funktionen des Programms sind:

1. Stützstelle(n) hinzufügen
2. Stützstelle löschen (nach Nummer)
3. Stützstelle(n) löschen (nach Koordinaten)
4. Alle Stützstellen löschen
5. Berechnung der Koeffizienten mittels Software
6. Berechnung der Koeffizienten mittels Hardware (Coprozessor)
7. Plotten des Splines mit gnuplot
8. Programm beenden

Nach jeder Aktion wird der aktuelle Datenbestand angezeigt.

Kapitel 11

Projektmanagement

11.1 Leiter und Teilnehmer

Die Projektgruppe bestand aus 2 Projektgruppenleitern und 13 Studenten.

Leiter :

Prof. Franz-Josef Rammig
email : franz@uni-paderborn.de

Christof Nagel
email : nagel@uni-paderborn.de

Studenten : *(alphabetisch sortiert)*

Bellanger, Thomas
email : tombel@uni-paderborn.de

Hennig, Andreas
email : ace@uni-paderborn.de

Hochwald, Jürgen
email : cfjh@uni-paderborn.de

Jean-Baptiste, Edwin
email : belneeg@uni-paderborn.de

Jungblut, Ralf
email : jungblut@uni-paderborn.de

Kukuk, Thorsten
email : kukuk@uni-paderborn.de

Marcus, Sven
email : glueck@uni-paderborn.de

Niechziol, Michael
email : minie@uni-paderborn.de

Reike, Burkhard
email : reike@uni-paderborn.de

Ullmann, Frank
email : full@uni-paderborn.de

Vieler, Claas
email : red@uni-paderborn.de

Werner, Lars
email : lawern@uni-paderborn.de

Willmer, Holger
email : dino@uni-paderborn.de

11.2 Seminarphase

Die Seminarphase der Projektgruppe erstreckte sich über 6 Wochen. Sie diente dazu, den Projektgruppenteilnehmern einiges an Wissen zu vermitteln, welches für die Teilnahme an der Projektgruppe benötigt wurde, ohne sich jedoch selbstständig in die Materie einzuarbeiten. Jeder Teilnehmer bereitete sich in das ihm zugeteilte Gebiet ein und trug seinen Vortrag, der sich in der Regel über eine Stunde erstreckte, der gesamten Projektgruppe vor.
Um das Seminar auch für jeden nachträglich zugänglich zu machen, wurde für jeden Vortrag eine kleine, 20 Seiten umfassende Seminararbeit angefertigt.

11.2.1 Splineinterpolation

In diesem Seminar von **Ralf Jungblut** geht es um die mathematischen Hintergründe der Spline–Berechnung. Erklärt wird der Weg, aus einer bestimmten Anzahl von Punkten eine mathematische Funktion zu konstruieren ($\rightarrow$ interpolieren). Das Beispiel am Ende des Seminars dient nocheinmal zur Verdeutlichung des doch recht komplizieren Algorithmusses.

11.2.2 Der Sparc Prozessor und sein CoPro

Nach einem kurzen geschichtlichen Überblick erklärt **Andreas Hennig** in seinem Seminar die Architektur des Sparc Prozessors. Er beschreibt die einzelnen Komponenten anhand eines Blockdiagrammes und geht dabei genauer auf die (für unsere Anwendung) wichtige Integer–Unit ein. Der zweite Teil des Seminares beschreibt die Anforderungen eines Coprozessors anhand des Bespieles FPU.

11.2.3 Projektsteuerung

Um eine genau geplante Folge von Arbeitsschritten sorgte sich **Michael Niechziol** in seinem Seminar. Er bezieht sich auf das Prozeßhandbuch von Siemens–Nixdorf und beschreibt die Organisation eines Projektes *(siehe nächstes Kapitel).*

11.2.4 VHDL

Die Grundzüge der Hardware–Beschreibungssprache VHDL beschreibt **Sven Marcus** in seiner Ausarbeitung. Das Seminar gibt lediglich einen kleinen Einblick in VHDL und soll daher nicht dem Erlernen der Sprache dienen. Es ist vielmehr dafür gedacht, VHDL–Sourcen lesen zu können.

11.2.5 High–Level Synthese & PASS-Pedal

Das Seminar von **Burkhard Reike** teilt sich in zwei Hälften. Zum einen wird der Syntheseprozess erläutert, der dazu dient, das zu entwickelnde Chip auf Register–Transferebene darzustellen. Zum anderen wird das High–Level–Synthese–Tool PASS kurz vorgestellt.

11.2.6 Synopsys & Votan

Der Synopsys Design Compiler ist ein Logik–Synthese–Tool, das Logik–Designs übersetzt und für Geschwindigkeit und Chipgröße optimiert. **Thorsten Kukuk** beschreibt in seinem Seminar die Arbeitsweise dieses Compilers und geht dabei kurz auf das Toolpaket VOTAN ein. VOTAN steht für VHDL Optimization, Transformation und Analysis.

11.2.7 ALU–Konstruktion, Logiksynthese, Steuerwerksentwurf

Lars Werner erläutert in seiner Ausarbeitung einige Optimierungsverfahren, darunter die beiden bekannten Verfahren *Karnaugh–Veitch* und *Quine McCluskey*. Er beschreibt weiterhin typische Komponenten von ALUs, deren Steuerwerk und die eigentliche Erzeugung von Rechenwerken.

11.2.8 EDIF

EDIF ist ein Format, welches zur Übertragung von Designdaten zwischen verschiedenen Anwendungen dient. **Holger Willmer** beschreibt in seinem Seminar die Struktur einer EDIF-Datei.

11.2.9 Simulationstechniken

Edwin Jean Baptiste erörtert die Unterschiede zwischen den drei Haupttechniken einer Simulation (SCS = Streamline Code Simulation, EI = Equitemporal Iteration und CES = Critical Event Scheduling).

11.2.10 Platzieren und Verdrahten

In diesem Seminar von **Thomas Bellanger** geht es um die verschiedenen Möglichkeiten eines Layoutes. Er beschreibt einige Anordnungs- und Verbindungsschemata und erläutert die Problematik der Platzierung und Verdrahtung.

11.2.11 Design for Testability

Unter ,,Design for Testability" faßt man die Methoden und Konzepte zusammen, die die Testbarkeit einer Schaltung erhöhen oder überhaupt erst ermöglichen. **Claas Vieler** erklärt dabei grundlegende Konzepte des Testens mit Hilfe von Testmustern.

11.2.12 C–MOS–Technologien

Jürgen Hochwald, dessen Seminarvortrag aus zeitlichen Gründen erst gegen Ende der Projektgruppe gehalten wurde, beschreibt die physikalischen Grundlagen der C–MOS– Technologie. Er geht auf den Herstellungsprozeß ein und weist auch auf die Probleme der neuen Technologie hin.

11.2.13 FPGA / INCA

In dieser Seminarausarbeitung erklärt **Frank Ullmann** den Entwurf digitaler Schaltungen und beschreibt den Aufbau des Hardwaresimulators *Virtual ASIC*. Genauer beschrieben wird dabei der eigentliche Chip (FPGA), der eine solche Simulation überhaupt erst möglich macht.

11.3 Das Prozeßhandbuch

Die Projektgruppenteilnehmer haben sich zum größten Teil nach dem Leitprozeß bei SNI Vorgehensweise gerichtet. Das Projekt wurde dabei grob in vier Prozeßabschnitte aufgeteilt :

- Problemanalyse (P10 bis P30)
- Aufgabendefinition (A10 bis A30)
- Technische Realisierung (T10 bis B30)
- Betreuung / Einsatz (B30 bis B90)

11.3.1 Problemanalyse

Der Prozeßabschnitt ,,Problemanalyse" teilt sich in zwei Teile auf. Zum einen gibt es die **Ist–Analyse** (P10 bis P20), die derzeitige Lösungen bezüglich der weiteren Entwicklung analysiert und zum anderen das **Soll–Konzept**, das im Prinzip eine Auswertung des Ist–Konzeptes ist.

11.3.2 Aufgabendefinition

Auch dieser Abschnitt gliedert sich in zwei Bereiche auf. In dem Grobkonzept (A10 bis A20) werden die Produktanforderungen aus dem Sollkonzept hinsichtlich Eindeutigkeit und Vollständigkeit analysiert, während in dem Fachfeinkonzept (A20 bis A30) z.B. schon die Oberfläche bzw. die Schnittstellen des Produktes beschrieben werden.

11.3.3 Technische Realisierung

Bei der technischen Realisierung wird unterschieden zwischen der Designspezifikation, der Komponentenspezifikation, den getesteten Komponenten, der Integration, dem Testen und schließlich der Abnahme. Die Aufteilung des Produktes in mehrere Komponenten und das anschließende Zusammenfügen zum gesamten Produkt machte eine Aufteilung der Arbeit in verschiedene Gruppen überhaupt erst möglich.

11.3.4 Betreuung / Einsatz

Eine Betreuung des Projektes war zwar in unserem Fall erst vorgesehen, wurde dann aber wegen technischer Probleme bei der Verwaltung der einzelnen Komponenten wieder verworfen.

11.4 Gruppeneinteilungen

Während der zwei Semester der Projektgruppe wurden die Teilnehmer immer wieder in verschiedene Gruppen aufgeteilt, die dann jeweils spezielle Aufgaben erhielten.

11.4.1 Erste Phase

Die erste Aufteilung der Projektgruppe gliederte sich wie folgt :

- **Rechenkern :** Diese Gruppe machte sich Gedanken über das eigentliche Konzept der Splineberechnung. Sie machte Vorschläge, wie z.B. die benötigten Stützstellen von der CPU angefordert werden sollen und wie die Ergebnisse nach der Berechnung wieder zurückgegeben werden.
- **Benutzungsschnittstelle :** Diese Gruppe programmierte einige Funktionen, mit denen später dann der Benutzer mit dem Koprozessor kommunizieren kann.
- **Betriebssystem :** Um die notwendigen Änderungen am Betriebssystem machte sich diese Gruppe einige Gedanken. Hier mußte z.B. darauf geachtet werden, daß jeweils die richtige Einheit (IU, FPU oder Coprozessor) die Maschinenbefehle erhält.
- **Interface :** Die Gruppe ,,Interface" arbeitete, wie der Name auch schon sagt, an der Schnittstelle Coprozessor $\leftrightarrow$ CPU und hatte damit wohl die schwierigste Aufgabe in der Projektgruppe.

11.4.2 Zweite Phase

Die Neuaufteilung der Gruppen am 15. Dezember 95 erfolgte nach der Fertigstellung der P30–Dokumente wie folgt :

- **SFU–Interface :** (Konstruktion, Simulation von W. Hardt). Die Gruppe beschäftigte sich mit dem Interface von Wolfram Hardt, daß entsprechend den Anforderungen angepaßt werden sollte.

- **Algorithmus / Rechenkern :** (Arbeiten mit Speedchart). Hier wurde zunächst ein Algorithmus in der Programmiersprache ,,C" entwickelt, der dann später mit Speedchart in VHDL übersetzt werden sollte.
- **Rechenwerk :** (Datentyp–Konvertierung, Operationen, Teilrechenwerke). Diese Gruppe erstellte die einzelnen Rechenwerke, angefangen mit Halbaddierern bis hin zu fertigen Float-Dividierern.
- **Benutzerinterface :** (Software, Bildschirmausgabe, Testumgebung).

11.4.3 Dritte Phase

Am 3. Mai 95 wurden die Projektgruppenteilnehmer ein drittes Mal in andere Gruppen aufgeteilt :

- **Interface :** Diese Gruppe soll nun den Laboraufbau soweit fertig machen, eine C–Library mit Assemblerprozeduren entwerfen und die Spline–SFU mit Dummy–Kern testen.
- **Gesamtmodell :** Hier soll an einem Synopsys–Modell gearbeitet werden, das später compilierbar und synthetisierbar sein soll.
- **Algorithmus :***(keine Änderungen)*
- **RCS :** Diese Gruppe soll die gesamte Arbeit in der Projektgruppe koordinieren.

11.4.4 Vierte Phase

Am 14. Juni 95 wurde die bisherige Gruppe ,,Gesamtmodell" in eine Synthese– und eine Simulations–Gruppe aufgeteilt. Die Synthese–Gruppe hatte die Aufgabe, zunächst die einzelnen Rechenwerke zu synthetisieren, um dann eine Gesamtsynthese zu starten, während die Simulationsgruppe den Algorithmus incl. aller Rechenwerke auf Korrektheit prüfte.
Die Gruppe ,,Management" bzw. ,,RCS" wurde mit dem Testen von Inca beauftragt. Ihre eigentliche Aufgabe, das Managen der Gruppe, wurde mehr und mehr vernachläßigt.

11.5 Programmieren unter CVS

CVS – was ist das überhaupt? CVS steht für "Concurrent-Version-System" ist ein "Source-Control" und "Revision-Control" Tool, entwickelt, um größere Softwareprojekte, an denen mehrere Entwickler arbeiten, zu unterstützen. CVS ist ein Programmpacket, was wiederum auf anderen Programmen (RCS , diff s.u.) aufbaut. Beim Einsatz von CVS wird die Kontrolle der Sourcen komplett durch CVS-Befehle erledigt ("Source-Control"). "Revision-Control" bedeutet, daß man jede Änderung eines Sourcefiles mitprotokolliert und gegebenenfalls wieder rückgängig machen kann. Wird nun lauffähiger Code durch Fehlerhaften überschrieben, so restauriert man mittels CVS die vorangehende "Revision" (Revision: kleine Änderung, noch keine neue "Version"). Ein weiterer Grund, warum wir uns für CVS entschieden haben ist, daß mehrere Entwickler gleichzeitig (multiple-developer) an einer Datei arbeiten können, ohne sich gegenseitig zu behindern. Dies wird durch sog. COPY-EDIT-MERGE erreicht, d.h. jeder Entwickler arbeitet auf einer Kopie der Sourcen, macht dort seine Änderungen und "mergt" diese Daten mit dem globalen Model, dem sog. "REPOSITORY". Um nun CVS zu benutzen, müssen folgendes Environmentvariablen gesetzt sein, im .cshrc wird z.B. durch folgende Zeilen gesetzt:

```
# ENVIRONMENT fuer CVS
#
# hier liegt z.B. das "Repository", die globale Kopie
# des gesamten Projektes
#
setenv CVSROOT /homes/prometheus/spline/red/CVSROOT
#
# hier liegen die Binaries von RCS,
#
```

```
setenv RCSBIN  /usr/local/gnu/rcs/bin
#
# CVS muss ausserdem im Suchpfad sein
#
set path = ( $path /usr/local/cvs/bin)
```

Da CVS auf RCS (Revison-Control-System) aufbaut und Programme von RCS aufruft, muß RCSBIN gesetzt werden. RCS ist ein etwas veraltertes Tool, welches Komplikationen dadurch verhindert, daß höchstens ein Entwickler an einer Datei arbeiten kann (Semaphore). Durch dieses "Filelocking" kommt es zu einer "Seriallisierung", wodurch Resourcen (manpower) verschwendet wird (ein/mehrere Entwickler müssen auf Einen warten). Durch das COPY-EDIT-MERGE von CVS wird dies geschickt umgangen. Das REPOSITORY ist nicht etwa in dem Format der eigentlichen Quellfiles gespeichert, sondern im sog. DIFF-Format (s. man diff). Dies ermöglicht auch die Wiederherstellung aller Revisionen eines Files. Ein leeres REPOSITORY muß vor der Benutzung von CVS durch "cvsinit" (s. man cvsinit) erstellt werden. Das REPOSITORY, im weiteren nur noch Rep genannt, ist hierachisch gegliedert. Der Zugriff erfolgt über sogennante Module. Module sind lediglich symbolische Namen für Mengen von Files, Verzeichnissen oder ganzen Teilbäumen des Gesamtmodells. Selbst die administrativen Files (Moduldef., Logfiles, History, Scripte), welche unter $CVSROOT/CVSROOT liegen, werden durch CVS selber verwaltet. Dadurch wird auch hier die Konsistenz bewahrt. Eine Änderung des Gesamtmodells läuft dann folgendermassen ab:

```
red@pollux [/homes/prometheus/spline/red/CVS/test] >>ls
```

Mit diesem Kommando wird das Module "modules" ausgechecked.

```
red@pollux [/homes/prometheus/spline/red/CVS/test] >>cvs checkout modules
U modules/modules
red@pollux [/homes/prometheus/spline/red/CVS/test] >>ls
modules/
```

Dann wird das Modul (File, Datei) editiert und danach der Status der Datei angezeigt.

```
red@pollux [/homes/prometheus/spline/red/CVS/test] >>vi modules/modules

red@pollux [/homes/prometheus/spline/red/CVS/test] >>cvs status
cvs status: Examining modules
===================================================================
File: modules           Status: Locally Modified

    Version:            1.1     Thu Aug 10 11:43:31 1995
    RCS Version:        1.1     /homes/prometheus/spline/red/CVSROOT/CVSROOT/modules,v
    Sticky Tag:         (none)
    Sticky Date:        (none)
    Sticky Options:     (none)
```

Und zu Schluss wird alles wieder "eingecheckt" (commit). Hier werden auch evtl. Konflikte aufgedeckt und behoben.

```
red@pollux [/homes/prometheus/spline/red/CVS/test] >>cvs commit
cvs commit: Examining modules
cvs commit: Committing modules
Checking in modules;
/homes/prometheus/spline/red/CVSROOT/CVSROOT/modules,v  <--  modules
new revision: 1.2; previous revision: 1.1
done
```

```
cvs commit: Executing 'mkmodules /homes/prometheus/spline/red/CVSROOT/CVSROOT'
red@pollux [/homes/prometheus/spline/red/CVS/test] >>
```

Diese letzten Seiten beschreiben nur einen kleinen Teil des Funktionsumfangs von CVS. Weitere wichtige Features von CVS sind die Branches, d.h. man kann verschiedene "Teilstränge" der Module parrallel entwickeln und später wieder zusammenführen. Man kann bestimmte Teilbäume markieren (taggen) und diesen Zustand einfrieren (freeze). Man kann bestimmte Scripte vor/nach auschecken/commiten ausführen und beliebig mitprotokollieren. Ausführliche und weiterführende Informationen findet man im "CVS REFERENCE MANUAL Ver. 1.3.1" von Per Cederqvist sowie im CVS-FAQ.

Kapitel 12

Fazit

Um es vorwegzunehmen : Das eigentliche Ziel, einen fertigen Spline-Koprozessor zu entwickeln, um ihn nachher auf dem Hardwareemulator zu simulieren, hat leider nicht geklappt.
Der Grund lag allerdings nicht an fehlender Kompetenz — es war vielmehr der zeitliche Faktor, der es uns unmöglich machte, unser Projektgruppenziel zu erreichen. Allerdings zählten in den zwei Semestern vielmehr die kleineren Ziele, die ein jeder von uns für sich vebuchen konnte. So funktionierten z.B. alle vier Rechenwerke (Addierer, Subtrahierer, Multiplizierer und Dividierer) einwandfrei. Nicht zu vergessen ist auch die Seminarphase im ersten Semester, die als Einstieg in die Problematik der Entwicklung eines Chips diente und die Erfahrung, die wir mit den verschiedenen Tools im Laufe der Zeit gesammelt haben.
Insgesamt läßt sich also sagen, daß die Arbeit in der Projektgruppe im großem und ganzen doch recht erfolgreich war. Daß das eigentliche Ziel nicht erreicht wurde, ist zwar schade, ändert jedoch nichts an der Tatsache, daß alle Projektgruppenteilnehmer in diesem Jahr viel dazu gelernt haben.

Anhang A

Sourcen für *tester*-Tools

A.1 Makefile

```
all: mkmon mkvec

mkmon.yy.c: mkmon.l
rm -f mkmon.yy.c
flex -t mkmon.l > mkmon.yy.c

mkmon: mkmon.yy.c
gcc -o mkmon mkmon.yy.c -ll

parse1.tab.c: parse1.y table.h
bison -d -v parse1.y
sed 's/yy/p1/g' parse1.tab.c | sed 's/YY/P1/g' > p.t.c
sed 's/yy/p1/g' parse1.tab.h | sed 's/YY/P1/g' > p.t.h
rm -f parse1.tab.*
mv p.t.c parse1.tab.c
mv p.t.h parse1.tab.h

parse1.yy.c: parse1.l table.h
rm -f parse1.yy.c
flex -L -t parse1.l | sed 's/yy/p1/g' | sed 's/YY/P1/g' > parse1.yy.c

parse2.tab.c: parse2.y table.h
bison -d -v parse2.y
sed 's/yy/p2/g' parse2.tab.c | sed 's/YY/P2/g' > p.t.c
sed 's/yy/p2/g' parse2.tab.h | sed 's/YY/P2/g' > p.t.h
rm -f parse2.tab.*
mv p.t.c parse2.tab.c
mv p.t.h parse2.tab.h

parse2.yy.c: parse2.l table.h
rm -f parse2.yy.c
flex -L -t parse2.l | sed 's/yy/p2/g' | sed 's/YY/P2/g' > parse2.yy.c

mkvec: main.C parse1.tab.c parse1.yy.c parse2.tab.c parse2.yy.c
g++ -g -o mkvec main.C parse1.yy.c parse1.tab.c parse2.tab.c parse2.yy.c -ll

clean:
rm -f *.yy.c *.tab.c *.tab.h *.output mkvec mkmon
```

A.2 table.h

```
/*  LAST EDIT: Thu Sep  7 21:01:32 1995 by Burkhard Reike (reike)  */
#include <stdio.h>

#define MAXDRIVERS 10
#define MAXSTRLEN 256
#define MAXSIGNALS 512

enum TesterMode {drive, strobe, undef};
enum Value {L,H,Z};

class Driver{
public:
  char name[MAXSTRLEN];
  TesterMode mode;
  Driver(){
    name[0] = 0;
    mode = undef;
  }
};

class Signal{
private:
  char* _name;
  int _dlen;
  Driver _drivers[MAXDRIVERS];
public:
  Value value;
  TesterMode mode;

  Signal(){
    _dlen = 0;
    mode = undef;
    value = Z;
  }
  Signal(char* n){
    _name = new char[strlen(n)+1];
    strcpy(_name,n);
    _dlen = 0;
    mode = undef;
    value = Z;
  }

  ~Signal(){ delete _name; }

  int entries() { return _dlen; }
  char* name() { return _name; }

  addDriver(TesterMode m, char* n){
    if (_dlen == MAXDRIVERS){
      fprintf(stderr,"Error: Maximum amount of drivers ( MAXDRIVERS ) \
exceeded.\n");
      exit(1);
    }
    if (strlen(n) >= MAXSTRLEN){
      fprintf(stderr,"Error: Maximum stringsize ( MAXSTRLEN ) exceeded.\n");
      exit(1);
    }
```

```
    strcpy(_drivers[_dlen].name,n);
    _drivers[_dlen++].mode = m;
  }

  Driver& getDriver(int num){
    if (num < 0 || num >= _dlen){
      fprintf(stderr,"Error: Illegal arrayindex.\n");
      exit(1);
    }
    return _drivers[num];
  }
};

class Signals{
public:
  Signal* sigs[MAXSIGNALS];
  int len;
  Signals(){
    len = 0;
  }
  ~Signals(){
    int i;
    for (i=0; i<len; i++) delete sigs[i];
  }
};
```

A.3 mkmon.l

```
; LAST EDIT: Thu Sep  7 21:04:24 1995 by Burkhard Reike (reike)
%%

\/[A-Za-z0-9\/_()]+ { printf("%s ",yytext); }
[ \t\n]+
.

%%

main(int argc, char**argv){
  ++argv, --argc;
  if (argc > 0)
    yyin = fopen(argv[0],"r");
  else
    yyin = stdin;
  printf("set watchsigs ");
  yylex();
  printf("\n");
  printf("set outfile monitors        -- write monitorresults into this\n");
  printf("                            -- file\n");
  printf("\n");
  printf("set outdevtag tmon          -- devicetag for monitorresult-\n");
  printf("                            -- file\n");
  printf("\n");
  printf("set mprefix M               -- prefix of monitornames\n");
  printf("\n");
  printf("open $outdevtag $outfile    -- open outputfile\n");
  printf("delete *                    -- remove all monitors\n");
  printf("foreach a in $watchsigs\n");
  printf("monitor -c -e -o $outdevtag -n $mprefix -x ^fprint \"time: \
%%t signal: %%t value: %%r\\n\" \\$now $a $a^ event $a\n");
```

```
  printf("end\n");
}

```

A.4 parse1.l

```
%{
#include <stdio.h>
#include <string.h>
#include "table.h"
#include "parse1.tab.h"
%}

%%

[A-Za-z0-9_]+:         { strcpy(p1lval.str,p1text);
                         p1lval.str[strlen(p1text)-1] = 0;
                         return SIGNAL;
                       }
drive                  { p1lval.mode = drive; return MODE; }
strobe                 { p1lval.mode = strobe; return MODE; }
\/[A-Za-z0-9\/_()]+   { strcpy(p1lval.str,p1text); return NAME; }
[\t\n]+
.

%%

int p1error (char* s)  /* Called by p1parse on error */
{
  printf ("%s\n", s);
}

#ifndef p1wrap
int p1wrap( void ) {
  return 1;
}
#endif
```

A.5 parse1.y

```
/*  LAST EDIT: Sun Jun 25 20:02:17 1995 by Burkhard Reike (reike)  */
%{
#include <stdio.h>
#include "table.h"
extern int p1lex();
extern int p1error(char*);
extern Signals signals;
Signal* actsig;
char* sig;
%}

%union{
  char str[MAXSTRLEN];
  TesterMode mode;
}

%token <str> SIGNAL
%token <str> NAME
%token <mode> MODE

```

```
%%

input: sigdef
| input sigdef
;

sigdef: SIGNAL
{
  actsig = new Signal($1);
  signals.sigs[signals.len++] = actsig;
}
drivers
;

drivers: /* empty */
| drivers modeplusname
;

modeplusname: MODE NAME { actsig->addDriver($1,$2); }
;
```

A.6 parse2.l

```
%{
#include <stdio.h>
#include <string.h>
#include "table.h"
#include "parse2.tab.h"
%}

%%

'.' { switch(p2text[1]){
        case '0': p2lval.val = L; break;
        case '1': p2lval.val = H; break;
        case 'Z': p2lval.val = Z; break;
        default: p2lval.val = Z;
      }
      return VALUE;
    }

[0-9]+ { p2lval.l = atol(p2text); return TIME; }

\/[A-Za-z0-9\/_()]+ { strcpy(p2lval.str,p2text); return P2NAME; }
[\t\n]+
.

%%

int p2error (char* s)  /* Called by p2parse on error */
{
  printf ("%s\n", s);
}

#ifndef p2wrap
int p2wrap( void ) {
  return 1;
}
#endif
```

A.7 parse2.y

```
/*  LAST EDIT: Sun Jun 25 20:03:28 1995 by Burkhard Reike (reike)  */
%{
#include <stdio.h>
#include "table.h"
extern int p2lex();
extern int p2error(char*);
extern void printvector();
extern void addMon(char*,Value);
long t = -1;
%}

%union{
  char str[MAXSTRLEN];
  Value val;
  long l;
}

%token <str> P2NAME
%token <val> VALUE
%token <l> TIME

%%

input: /* empty */
| input event
;

event: TIME
{
  if (t == -1) t = $1;
  else
    if ($1 != t){
    /*  printf("printvector\n"); */
      printvector();
      t = $1;
    }
}
P2NAME VALUE
{
 /* printf("addMon(%s,",$3);
  if ($4 == L) printf(" L)\n");
  if ($4 == H) printf(" H)\n");
  if ($4 == Z) printf(" Z)\n");*/
  addMon($3,$4);
}
;
```

A.8 main.C

```
//  LAST EDIT: Sun Jun 25 19:58:00 1995 by Burkhard Reike (reike)
#include <stdio.h>
#include <string.h>
#include "table.h"

extern int p1parse();
extern int p2parse();
extern FILE *p1in, *p2in;
```

```
FILE *vecout, *namout;

Signals signals;

main(int argc, char**argv){
  ++argv, --argc;
  if (argc < 2){
    printf("Usage: mkvec <signaldescr-file> <monitorfile>\n");
    return 0;
  }

  p1in = fopen(argv[0],"r");
  p1parse();
  fclose(p1in);

  namout = fopen("names.map","w");
  int i,j;
  Signal* si;
  TesterMode m;
  for (i=0; i<signals.len; i++){
    si = signals.sigs[i];
    fprintf(namout,"%s %d;\n",si->name(),i);

/*
    printf("Signal: %s\n", si->name());
    for (j=0; j<si->entries(); j++){
      m = si->getDriver(j).mode;
      printf("mode: ");
      if (m == drive) printf("drive ");
      if (m == strobe) printf("strobe ");
      if (m == undef) printf("undef ");
      printf("name: %s\n",si->getDriver(j).name);
    }
*/
  }
  fprintf(namout,"END\n");
  fclose(namout);

  p2in = fopen(argv[1],"r");
  vecout = fopen("vectors.vec","w");
  p2parse();
  fclose(p2in);
  fclose(vecout);
}

typedef struct dummy2{
  char name[MAXSTRLEN];
  Value value;
} MonResult;

MonResult mon[MAXSIGNALS];
int mlen = 0;

void addMon(char* name, Value val){
  strcpy(mon[mlen].name,name);
  mon[mlen++].value = val;
}
```

```
void printvector(){
  int i,j,k;
  char* n;
  Signal* si;
  int match;

  // update
  for (i=0; i<signals.len; i++){
    match = 0;
    si = signals.sigs[i];
    for (j=0; j<si->entries() && !match; j++){
      n = si->getDriver(j).name;
      for (k=mlen-1; k>=0 && !match; k--)
if (!strcmp(n,mon[k].name)){
  si->value = mon[k].value;
  si->mode = si->getDriver(j).mode;
  match = 1;
}
    }
  }

  mlen = 0;

  // output

  Value svalue;
  TesterMode smode;

  fprintf(vecout,"-");
  for (i=0; i<signals.len; i++){
    si = signals.sigs[i];
    svalue = si->value;
    smode = si->mode;
    if (smode == undef) fprintf(vecout,"*");
    if (smode == strobe){
      if (svalue == L) fprintf(vecout,"0");
      if (svalue == H) fprintf(vecout,"1");
      if (svalue == Z) fprintf(vecout,"Z");
    }
    if (smode == drive){
      if (svalue == L) fprintf(vecout,"L");
      if (svalue == H) fprintf(vecout,"H");
      if (svalue == Z) fprintf(vecout,"*"); // 'Z' can only be strobed
    }
  }
  fprintf(vecout,";\n");
}
```

Anhang B

Manipulationen an *SunOs*

B.1 coprotest.c

```
/*  LAST EDIT: Tue Jun 13 13:33:10 1995 by Burkhard Reike (reike)  */
#include <stdio.h>

int getpsr(){
  int trapnr, ret;

  trapnr = 0x10;

  asm("ld %0,%%o1" : : "m" (trapnr) : "o1");
  asm("ta %%o1" : : : "o0");
  asm("st %%o0,%0" : "=m" (ret));

  return ret;
}

int toggleC(){
  int trapnr, ret;

  trapnr = 0x11;

  asm("ld %0,%%o1" : : "m" (trapnr) : "o1");
  asm("ta %%o1" : : : "o0");
  asm("st %%o0,%0" : "=m" (ret));

  return ret;
}

/*
int toggleNback(){
  int trapnr1,trapnr2,ret;

  trapnr1 = 0x10;
  trapnr2 = 0x11;

  asm("ld %0,%%o1" : : "m" (trapnr2) : "o1");
  asm("ld %0,%%o2" : : "m" (trapnr1) : "o2");
  asm("ta %o1");
  asm("ta %%o2" : : : "o0");
  asm("st %%o0,%0" : "=m" (ret));

  return ret;
```

```
}
*/

void sendLDC(){
  int bla = 0;
  asm("ld %0,%%c0" : : "m" (bla));
}

void printformat(int c){
  int i;
  int arr[32];

  for (i=0; i<32; i++){
    arr[i] = (c & 1) ? 1 : 0;
    c >>= 1;
  }

  for (i=31; i>=0; i--) printf("%d",arr[i]);
}

main(){
  float x=1.0;
  int choice,status;
  x *= 2.5;

  while(1){
    printf("1: PSR ausgeben\n");
    printf("2: CP-Bit im PSR aendern\n");
    printf("3: LDC-Befehl ausfuehren\n");
/*    printf("4: CP toggeln und PSR ausgeben\n"); */
    printf("\nIhre Wahl: ");
    scanf("%d",&choice);
    switch(choice){
      case 1: printformat(status=getpsr());
      if (status & (1 << 13)) printf("\nCopro-Bit (Bit 13) ist gesetzt\n");
      else printf("\nCopro-Bit (Bit 13) ist nicht gesetzt\n");
      break;
      case 2: printformat(status=toggleC());
      if (status & (1 << 13)) printf("\nCopro-Bit (Bit 13) ist gesetzt\n");
      else printf("\nCopro-Bit (Bit 13) ist nicht gesetzt\n");
      break;
      case 3: sendLDC();
      break;
/*      case 4: printformat(status=toggleNback());
      if (status & (1 << 13)) printf("\nCopro-Bit (Bit 13) ist gesetzt\n");
      else printf("\nCopro-Bit (Bit 13) ist nicht gesetzt\n");
      break;
*/
      default: printf("Unerlaubte Eingabe!\n");
    }
  }
}
```

B.2 t.s (Assemblercode für Softwaretraps)

```
.text
        rd %psr, %i0            /* Soft-Trap Nr. 0x10 (psr lesen) */
        jmp %l2
        rett %l2 + 4
```

```
        nop
        rd %psr,%o1             /* Soft-Trap Nr. 0x11 (Copro-Bit toggeln) */
        sethi %hi(0x2000), %o2
        or %o1, %o2, %o3
        mov %o3, %psr
        nop                     /* Soft-Trap Nr. 0x12 (unbenutzt) */
        nop                     /* 3 nops nach Aenderung von psr noetig */
        nop
        rd %psr, %i0
        jmp %l2                 /* Soft-Trap Nr. 0x13 (unbenutzt) */
        rett %l2 + 4
        nop
        nop

```

B.3 fiddle.C

```
/*  LAST EDIT: Sat Jun 10 20:05:48 1995 by Burkhard Reike (reike)  */
#include <stdlib.h>
#include <stdio.h>
#include <stream.h>
#include <fstream.h>

void main(){
  char src1n[64], src2n[64], trgtn[64];
  long p, from, to;

  printf("File to be changed: ");
  scanf("%s",src1n);
  ifstream src1(src1n);
  if (!src1){
    printf("No such file.\n");
    exit(0);
  }
  printf("File to insert from: ");
  scanf("%s",src2n);
  ifstream src2(src2n);
  if (!src2){
    printf("No such file.\n");
    exit(0);
  }
  printf("Name of output-file: ");
  scanf("%s",trgtn);
  ofstream trgt(trgtn);
  if (!trgt){
    printf("File couldn't be opened.\n");
    exit(0);
  }
  printf("Insert which bytes from %s (<from> <to>): ",src2n);
  scanf("%d %d",&from,&to);
  printf("Insert these bytes into %s at byte-position: ",src1n);
  scanf("%d",&p);

  long i=0;
  char ch;

  if (i< p) printf("Writing bytes 0 to %d from %s to %s\n",p-1,src1n,trgtn);
  while (i < p){
    if (src1.get(ch)) trgt.put(ch);
```

```
    else{
      if (!src1.eof() || trgt.bad()){
printf("An error occured.\n");
exit(0);
      }
    }
    i++;
  }

  printf("Writing bytes %d to %d from %s to %s\n",from,to,src2n,trgtn);
  src2.seekg(from,ios::seek_dir(0));
  while (i < p + to - from + 1){
    src1.get(ch); // advance on src1
    if (src2.get(ch)) trgt.put(ch);
    else{
      if (!src2.eof() || trgt.bad()){
printf("An error occured.\n");
exit(0);
      }
    }
    i++;
  }

  printf("Writing rest from %s to %s\n",src1n,trgtn);
  while (src1.get(ch)) trgt.put(ch);
  if (!src1.eof() || trgt.bad()){
    printf("An error occured.\n");
    exit(0);
  }
  src1.close();
  src2.close();
  trgt.close();
  printf("Finished.\n");
}
```

Anhang C

Sourcen für die Benutzerschnittstelle

C.1 sfu.h

```
/*  LAST EDIT: Tue Jun 13 23:44:35 1995 by Burkhard Reike (reike)  */

#ifndef __sfu_h
#define __sfu_h

extern void ldcf(float, int);   /* Load float into coprocessor-register */
extern void ldci(int, int);     /* Load integer into coprocessor-register */
extern float stcf(int);         /* Return float from coprocessor-register */
extern int stci(int);           /* Return integer from coprocessor-register */
extern void resetSFU();         /* Reset SFU */
extern int getPSR();            /* Get PSR */
extern int enableCP();          /* Enable SFU and return new PSR */

#endif /*!__sfu_h*/
```

C.2 sfu.c

```
/*  LAST EDIT: Tue Jun 13 23:45:51 1995 by Burkhard Reike (reike)  */

#include <stdio.h>

#define TRAP_GET_PSR 0x10
#define TRAP_ENABLE_CP 0x11

#define LDC(adr,register) \
switch (register){ \
case 0: __asm __volatile("ld %0,%%c0" : : "m" (adr)); break; \
case 1: __asm __volatile("ld %0,%%c1" : : "m" (adr)); break; \
case 2: __asm __volatile("ld %0,%%c2" : : "m" (adr)); break; \
case 3: __asm __volatile("ld %0,%%c3" : : "m" (adr)); break; \
case 4: __asm __volatile("ld %0,%%c4" : : "m" (adr)); break; \
case 5: __asm __volatile("ld %0,%%c5" : : "m" (adr)); break; \
case 6: __asm __volatile("ld %0,%%c6" : : "m" (adr)); break; \
case 7: __asm __volatile("ld %0,%%c7" : : "m" (adr)); break; \
default : printf("Unerlaubtes Register (%d) in LDC-Befehl\n",register); \
}

#define STC(register,adr) \
switch(register){ \
```

```
case 0: __asm __volatile("st %%c0,%0" : "=m" (adr)); break; \
case 1: __asm __volatile("st %%c1,%0" : "=m" (adr)); break; \
case 2: __asm __volatile("st %%c2,%0" : "=m" (adr)); break; \
case 3: __asm __volatile("st %%c3,%0" : "=m" (adr)); break; \
default : printf("Unerlaubtes Register (%d) in STC-Befehl\n",register); \
}
/* reset = "10xx xxx1 1011 xxxx xxxx xxxx xxxx xxxx" */
#define SFURESET __asm __volatile(".word 0x81b00000");
/* Function: void ldcf(float f, int reg)                     */
/* Purpose : Loads the float which is contained in 'f' into the */
/*           coprocessor-register No. 'reg'.                 */
void ldcf(float f, int reg){
  LDC(f,reg)
}
/* Function: void ldci(int i, int reg)                                 */
/* Purpose : Loads the integer contained in 'i' into the               */
/*           coprocessor-register No. 'reg'.                           */
void ldci(int i, int reg){
  LDC(i,reg)
}
/* Function: float stcf(int reg)                                       */
/* Purpose : Returns the contents of copro-register No. 'reg' (as float). */
float stcf(int reg){
  float ret;
  STC(reg,ret)
  return ret;
}
/* Function: int stci(int reg)                                        */
/* Purpose : Returns the contents of copro-register No. 'reg' (as int).  */
int stci(int reg){
  int ret;
  STC(reg,ret)
  return ret;
}
/* Function: void resetSFU()                                          */
/* Purpose : Send reset-command to SFU                                */
void resetSFU(){
  SFURESET
}
/* Function: int getPSR()                                             */
/* Purpose : Returns PSR                                              */
int getPSR(){
  int trapnr, ret;
```

```
  trapnr = TRAP_GET_PSR;

  asm("ld %0,%%o1" : : "m" (trapnr) : "o1");
  asm("ta %%o1" : : : "o0");
  asm("st %%o0,%0" : "=m" (ret));

  return ret;
}

/* Function: int enableCP()                                             */
/* Purpose : Sets CP-Bit in PSR and returns new PSR                     */

int enableCP(){
  int trapnr, ret;

  trapnr = TRAP_ENABLE_CP;

  asm("ld %0,%%o1" : : "m" (trapnr) : "o1");
  asm("ta %%o1" : : : "o0");
  asm("st %%o0,%0" : "=m" (ret));

  return ret;
}

#undef LDC
#undef STC
#undef SFURESET
```

C.3 spline.h

```
/*  LAST EDIT: Mon Jul 10 20:44:19 1995 by Burkhard Reike (reike)  */
#ifndef __spline_h
#define __spline_h

#undef _USE_SFU_   /* define this if you can really use copro-machinecode */
#define _VERBOSE_  /* define this if each copro-command shall be announced */

#include <sfu.h>
#include <stream.h>

#define OPREGS 8    // # of operand-registers
#define POINTS 15   // maximum # of points
#define COEFFS 4    // # of coefficients per spline-polynome

typedef float Coeffs[COEFFS];
typedef float Point[2];

class Spline{
private:
  int _len;                         // size of point-set
  Point _points[POINTS];            // set of points
  Coeffs _coeffs[POINTS-1];         // array of spline coefficients

  int _calculated;                  // true if coefficients have been calc.
  int _gotResults;                  // true if results were taken from SFU
  static Spline *_lastCaller;       // the object which is waiting for results
  static int _results;              // No. of results yet to get from SFU
  int getResults();                 // get coefficients from SFU
```

```
  void touch();                     // notify change in point-set

  float _M[POINTS], _A1[POINTS], _B1[POINTS]; // results of algorithm

  static int _maxnr;
  int _nr;

  void _ldcf(float x, int r){
#ifdef _VERBOSE_
    cout << _nr << "->ldcf(" << x << "," << r << ")\n";
#endif
#ifndef _USE_SFU_
    return;
#endif
    ldcf(x,r);
  }

  void _ldci(int x, int r){
#ifdef _VERBOSE_
    cout << _nr << "->ldci(" << x << "," << r << ")\n";
#endif
#ifndef _USE_SFU_
    return;
#endif
    ldci(x,r);
  }

  float _stcf(int r){
#ifdef _VERBOSE_
    cout << _nr << "->stcf(" << r << ")\n";
#endif
#ifndef _USE_SFU_
    return (float)_nr;
#endif
    return stcf(r);
  }

  void _enable(){
#ifdef _VERBOSE_
    cout << _nr << "->enable()\n";
#endif
#ifndef _USE_SFU_
    return;
#endif
    enableCP();
  }

public:
  Spline();
  Spline(Spline& s);
  ~Spline();
  Spline& operator=(const Spline& s);

  // Members for manipulating the point-set
  int addPt(float x, float y);      // add point (x,y) to set
  int delPt(float x, float y);      // delete point (x,y) from set
  int delPt(int nr);                // delete point No. nr from set
  int clrPts();                     // clear point-set

```

```
  // SFU-related members (optional, not necessary for calculation)
  int calc();                       // send points to SFU
  void reset(){                     // reset SFU
    if (_lastCaller){
      _lastCaller->_calculated = 0;
      _lastCaller->_gotResults = 0;
      _lastCaller = 0;
      _results = 0;
    }
#ifdef _VERBOSE_
    cout << _nr << "->reset()\n";
#endif
#ifndef _USE_SFU_
    return;
#endif
    resetSFU();
  }

  // Software-Calculation of coefficients
  int softCalc();

  // Members for accessing information

    // get size of set
  int getSize(){ return _len; }

    // get xy-coordinates of point No. <nr>
  int getPt(int nr, float *x, float *y) const;

    // get point No. <nr> (as array of size 2)
  const Point& getPt(int nr) const;

    // get range of polynome No. <nr>
  int getRange(int nr, float *x1, float *y1, float *x2, float *y2) const;

    // get coefficients of polynome No. <nr>, which is a*x^3+b*x^2+c*x+d
  int getCoeff(int nr, float *a, float *b, float *c, float *d);

    // return coefficient-array No. <nr>
  const Coeffs& getCoeff(int nr);

  // Misc

    // formated printing of Spline
  void print(ostream& out = cout) const;

    // print out plot-command for Maple(tm) which plots the spline
  int printMaplePlot(ostream& out = cout);

    // system()-call for gnuplot to plot the spline
  int plot();

};

// Global Spline-related functions

extern ostream& operator<<(ostream& s, Spline& x); // just like print
extern const char* SplineErrText(int nr);       // returns Errortext

```

```
// Error-Codes

#define SP_ERRORS 9 // # of errorcodes

#define SP_OK 0
#define SP_MAXSIZE_EXCEEDED 1
#define SP_REDUNDANT_POINT 2
#define SP_REDUNDANT_XCOORD 3
#define SP_NO_SUCH_POINT 4
#define SP_TOO_FEW_POINTS 5
#define SP_ILLEGAL_POLYNOME_NR 6
#define SP_ILLEGAL_POINT_NR 7
#define SP_NOT_CALCULATED 8

#endif /*!__spline_h*/
```

C.4 spline.C

```
//  LAST EDIT: Thu Sep  7 21:06:35 1995 by Burkhard Reike (reike)
#include <spline.h>
#include <String.h>
#include <std.h>
#include <stdio.h>

#define FATAL(msg) {cerr << "Internal Error: " << msg << "\n" << flush; \
if (_results) { cerr << "Getting results and exit.\n" << flush; _lastCaller \
= this; this->getResults(); exit(1); }}

#define CALL(func) {int err__ = func ; if (err__ != SP_OK) return err__;}

#define MAX(x,y) ((x > y) ? x : y)
#define MIN(x,y) ((x < y) ? x : y)


// Initialisation of static variables

Spline* Spline::_lastCaller = 0;
int Spline::_results = 0;
int Spline::_maxnr = 0;


// Constructors

Spline::Spline(){
  _len = 0;
  _calculated = 0;
  _gotResults = 0;
  if (!_maxnr){
    _enable();
    reset();
  }
  _nr = _maxnr++;
}

Spline::Spline(Spline& s){
  int i;
  _len = s._len;
  _calculated = 0;
  _gotResults = 0;
```

```
  _nr = _maxnr++;
  for (i=0; i<_len; i++){
    _points[i][0] = s._points[i][0];
    _points[i][1] = s._points[i][1];
  }
}

Spline& Spline::operator=(const Spline& s){
  int i;
  _len = s._len;
  _calculated = 0;
  _gotResults = 0;
  for (i=0; i<_len; i++){
    _points[i][0] = s._points[i][0];
    _points[i][1] = s._points[i][1];
  }
  return *this;
}


// Destructor

Spline::~Spline(){
  // before deletion: get the results if we are waiting for any
  if (_results && _lastCaller == this) getResults();
}


void Spline::touch(){
  _calculated = 0;
  _gotResults = 0;
}

// members for manipulating the point-set

int Spline::addPt(float x, float y){
  int i,j;
  float bx,by;

  if (_len == POINTS) return SP_MAXSIZE_EXCEEDED;

  for (i=0; i<_len && x > _points[i][0]; i++);

  if (i < _len){
    if (x == _points[i][0])
      return ((y == _points[i][1]) ? SP_REDUNDANT_POINT : SP_REDUNDANT_XCOORD);

    for (j=i; j<_len; j++){
      bx = _points[j][0];
      by = _points[j][1];
      _points[j][0] = x;
      _points[j][1] = y;
      x = bx;
      y = by;
    }
  }

  _points[_len][0] = x;
  _points[_len][1] = y;
```

```
  _len++;

  touch();
  return SP_OK;
}

int Spline::delPt(float x, float y){
  int i,j;

  for (i=0; i<_len && (x != _points[i][0] || y != _points[i][1]); i++);
  if (i == _len) return SP_NO_SUCH_POINT;
  for (j=i; j < _len-1; j++){
    _points[j][0] = _points[j+1][0];
    _points[j][1] = _points[j+1][1];
  }
  _len--;

  touch();
  return SP_OK;
}

int Spline::delPt(int nr){
  if (nr < 0 || nr >= _len) return SP_ILLEGAL_POINT_NR;
  return delPt(_points[nr][0], _points[nr][1]);
}

int Spline::clrPts(){
  _len = 0;
  touch();
  return SP_OK;
}


// SFU-related members

int Spline::getResults(){
  if (!_results) FATAL("Call for nonexistent SFU-results.");
  if (!_lastCaller) FATAL("There are SFU-results but no object to get them.");
  if (_lastCaller != this) FATAL("Unauthorized object tries to get \
SFU-results.");

  // now _lastCaller == this and _results > 0

  int i;
  float x[POINTS],y[POINTS],h[POINTS];

  for (i=0; i<_len; i++){
    x[i] = _points[i][0];
    y[i] = _points[i][1];
  }
  for (i=1; i<_len; i++) h[i] = x[i] -x[i-1];

  for (i=1; i <= _results; i++){
    _M[i-1] = _stcf(0);
    _M[i] = _stcf(1);
    _A1[i] = _stcf(2);
    _B1[i] = _stcf(3);
    _coeffs[i-1][0] = (_M[i] - _M[i-1])/(6*h[i]);
    _coeffs[i-1][1] = (x[i]*_M[i-1]-x[i-1]*_M[i])/(2*h[i]);
```

```
    _coeffs[i-1][2] = (x[i-1]*x[i-1]*_M[i]-x[i]*x[i]*_M[i-1])/(2*h[i])+_A1[i];
    _coeffs[i-1][3] = (_M[i-1]*x[i]*x[i]*x[i]-_M[i]*x[i-1]*x[i-1]*
x[i-1])/(6*h[i]) - _A1[i]*x[i-1] + _B1[i];
  }

  _results = 0;
  _lastCaller = 0;
  _gotResults = 1;
  return SP_OK;
}

int Spline::calc(){
  if (_len < 3) return SP_TOO_FEW_POINTS;

  if (_results){
    if (!_lastCaller) FATAL("There are SFU-results but no object to get \
them.");
    CALL(_lastCaller->getResults());
  }

  if (_results) FATAL("SFU-results were not removed for unknown reason.");
  if (_lastCaller) FATAL("Object waiting for zero SFU-results.");

  // SFU-Commands
  int reg,i;

  _ldci(_len,0); // #points into first register
  _ldci(0,1);    // dummy into second
  reg = 2;
  for (i=0; i<_len; i++){
    _ldcf(_points[i][0],reg);
    _ldcf(_points[i][1],reg+1);
    reg = (reg + 2) % OPREGS;
  }

  _lastCaller = this;
  _results = _len-1;
  _calculated = 1;
  _gotResults = 0;
  return SP_OK;
}

// Calculation of coefficients - Softwaresolution

int Spline::softCalc(){
  if (_len < 3) return SP_TOO_FEW_POINTS;

  int i;
  const int n = _len-1;
  const int m = _len;
  float x[m],y[m],f[m],h[m],Dy[m],a[m],b[m],d[m],e[m],z[m];

  // ------------------- Werte einlesen -------------------
  for (i=0; i<=n; i++){
    x[i] = _points[i][0];
    y[i] = _points[i][1];
  }

  // ------------------ Komponente der Matrix ----------------
```

```

  for (i=0; i<n; i++){
    h[i+1] = x[i+1] - x[i];
    Dy[i+1] = y[i+1] - y[i];
    if (i>0) f[i] = Dy[i+1]/h[i+1] - Dy[i]/h[i];
    b[i+1] = h[i+1]/6;
    if (i>0) d[i] = 2*(b[i] + b[i+1]);
  }

  // ---------------  LR Zerlegung  ------------------

  for (i=1; i<n; i++){
    if (i==1) a[i] = d[1];
    else a[i] = d[i] - b[i]*e[i-1];
    if (i<n-1) e[i] = b[i+1]/a[i];
  }

  // ----------------  Vorwaertssubstitution  ---------------

  for (i=1; i<n; i++){
    if (i==1) z[1] = f[1]/a[1];
    else z[i] = (f[i] - b[i]*z[i-1])/a[i];
  }

  // --------------- Rueckwaertssubstitution -----------------

  for (i=n-1; i>0; i--){
    if (i==n-1) _M[i] = z[i];
    else _M[i] = z[i] - e[i]*z[i+1];
  }

  // -------------- Koeffiziente A_i  und B_i  -------------

  _M[0]=0;
  _M[n]=0;
  for (i=1; i<=n; i++){
    _B1[i] = y[i-1] - _M[i-1]*b[i]*h[i];
    _A1[i] = Dy[i]/h[i] - b[i]*(_M[i] - _M[i-1]);
  }

  // ---------- Splinekoeffizienten -------

  for (i=1; i <= n; i++){
    _coeffs[i-1][0] = (_M[i] - _M[i-1])/(6*h[i]);
    _coeffs[i-1][1] = (x[i]*_M[i-1]-x[i-1]*_M[i])/(2*h[i]);
    _coeffs[i-1][2] = (x[i-1]*x[i-1]*_M[i]-x[i]*x[i]*_M[i-1])/(2*h[i])+_A1[i];
    _coeffs[i-1][3] = (_M[i-1]*x[i]*x[i]*x[i]-_M[i]*x[i-1]*x[i-1]*
x[i-1])/(6*h[i]) - _A1[i]*x[i-1] + _B1[i];
  }

  _calculated = 1;
  _gotResults = 1;
  return SP_OK;
}



// members for accessing information

```

```
int Spline::getCoeff(int nr, float *a, float *b, float *c, float *d){
  if (!_calculated) CALL(calc());
  if (!_gotResults) CALL(getResults());
  if (nr < 0 || nr >= _len-1) return SP_ILLEGAL_POLYNOME_NR;

  *a = _coeffs[nr][0];
  *b = _coeffs[nr][1];
  *c = _coeffs[nr][2];
  *d = _coeffs[nr][3];

  return SP_OK;
}

const Coeffs& Spline::getCoeff(int nr){
  if (!_calculated) calc();
  if (!_gotResults) getResults();
  if (nr < 0 || nr >= _len-1) nr = 0;

  return _coeffs[nr];
}

int Spline::getPt(int nr, float *x, float *y) const{
  if (nr < 0 || nr >= _len) return SP_ILLEGAL_POINT_NR;

  *x = _points[nr][0];
  *y = _points[nr][1];

  return SP_OK;
}

const Point& Spline::getPt(int nr) const{
  if (nr < 0 || nr >= _len) nr = 0;
  return _points[nr];
}

int Spline::getRange(int nr, float *x1, float *y1, float *x2, float *y2) const{
  if (nr < 0 || nr >= _len-1) return SP_ILLEGAL_POLYNOME_NR;

  *x1 = MIN(_points[nr][0],_points[nr+1][0]);
  *x2 = MAX(_points[nr][0],_points[nr+1][0]);
  *y1 = MIN(_points[nr][1],_points[nr+1][1]);
  *y2 = MAX(_points[nr][1],_points[nr+1][1]);

  return SP_OK;
}


// misc

void Spline::print(ostream& s) const {
  int i;

  s << "Spline No. " << _nr << ": " << _len << " points.\n";
  for (i=0; i<_len; i++) s << i<< ":(" << _points[i][0] << ", "
<< _points[i][1] << ") ";
  s << "\ncalculated: ";
  if (_calculated) s << "yes\n";
  else s << "no\n";
  s << "results got: ";
```

```
  if (_gotResults) s << "yes\n";
  else s << "no\n";
  if (_calculated && _gotResults){
    s << "Results of the algorithm:\n";
    for (i=0; i<_len; i++) s << "M[" << i << "] = " << _M[i] << "\n";
    for (i=1; i<_len; i++) s << "A[" << i << "] = " << _A1[i] << "\n";
    for (i=1; i<_len; i++) s << "B[" << i << "] = " << _B1[i] << "\n";
    s << "\nSplinepolynomes:\n";
    for (i=0; i<_len-1; i++)
      s << i+1 << ". : " << _coeffs[i][0] << "*X^3 + " <<_coeffs[i][1]
<< "*X^2 + " << _coeffs[i][2] << "*X + " << _coeffs[i][3] << "\n";
  }
}

int Spline::printMaplePlot(ostream& s){
  if (!_calculated) CALL(calc());
  if (!_gotResults) CALL(getResults());

  int i,j;
  float x1,y1,x2,y2;

  s << "plot({";
  for (i=0; i<_len-1; i++){
    getRange(i,&x1,&y1,&x2,&y2);
    s << "[x, ";
    for (j=0; j<COEFFS; j++)
      s << "+(" << _coeffs[i][j] << "*x^" << COEFFS-j-1 << ")";
    s << ", x=" << x1 << ".." << x2 << "]";
    if (i < _len-2) s << ", ";
  }
  s << "});";

  return SP_OK;
}

int Spline::plot(){
  if (!_gotResults) return SP_NOT_CALCULATED;
  if (_len < 3) return SP_TOO_FEW_POINTS;

  String s;
  int i,j;
  float x1,x2;
  char dum1[256], dum2[256];

  s = "(( echo \"set parametric\nset trange[0:1]\nset nokey\n";

  for (i=0; i<_len-1; i++){
    x1 = _points[i][0];
    x2 = _points[i+1][0];
    sprintf(dum1,"(%f+(%f)*t)",x1,x2-x1);
    sprintf(dum2,"g%d(t) = ",i);
    s += dum2;
    s += dum1;
    sprintf(dum2,"\nh%d(t) = ",i);
    s += dum2;
    for (j=0; j<COEFFS; j++){
      sprintf(dum2,"(%f)*",getCoeff(i)[j]);
      s += dum2;
      s += dum1;
```

```
      sprintf(dum2,"**%d",COEFFS-j-1);
      s += dum2;
      if (j+1 < COEFFS) s += "+";
    }
    s += "\n";
  }
  s += "plot ";
  for (i=0; i<_len-1; i++){
    sprintf(dum1,"g%d(t),h%d(t)",i,i);
    s += dum1;
    if (i<_len-2) s += ",";
  }

  s += "; pause 300\" ) | gnuplot) 2>> gnupl.log &";
  system(s.chars());
  return SP_OK;
}


ostream& operator<<(ostream& s, Spline& x){
  x.print(s);
  return s;
}

const char* errors[SP_ERRORS+1] = {
  "Ok",
  "Maximum amount of points exceeded",
  "Redundant point",
  "Redundant X-coordinate",
  "No such point",
  "Too few points for calculation",
  "Illegal polynome-No.",
  "Illegal point-No.",
  "Coefficients haven't been calculated yet",
  "No such error-number"
};

const char* SplineErrText(int nr){
  if (nr < 0 || nr > SP_ERRORS) nr = SP_ERRORS;
  return errors[nr];
}
```

C.5 cspline.h

```
/*  LAST EDIT: Sat Sep  2 19:18:51 1995 by Burkhard Reike (reike)  */
#ifndef __cspline_h
#define __cspline_h

/* add point (x,y) to list of points */
extern int SplAddPt(float x, float y);

/* remove point (x,y) from list of points */
extern int SplDelPt(float x, float y);

/* clear list of points */
extern int SplClrPts();

/* send points to SFU and start calculation */
extern int SplCalc();
```

```

/* reset SFU */
extern void SplReset();

/* get coordinates of point No. <nr> */
extern int SplGetPt(int nr, float* x, float* y);

/* get coefficients of spline-polynome No. <nr> */
extern int SplGetCoeff(int nr, float *a, float *b, float *c, float *d);

/* get range of spline-polynome No. <nr> */
extern int SplGetRange(int nr, float *x1, float *y1, float *x2, float *y2);

/* return errortext No. <nr> */
extern char* SplErrText(int nr);

/* formated printout of information about the spline on standard-output */
extern void SplPrint();

/* printout of Maple(tm) plot-command on standard-output */
extern int SplMaplePlot();

/* plot spline using gnuplot */
extern int SplPlot();

#endif /*!__cspline_h*/
```

C.6 cspline.C

```
//  LAST EDIT: Sat Sep  2 18:40:39 1995 by Burkhard Reike (reike)
#include <spline.h>

static Spline s;

extern "C" int SplAddPt(float x, float y){
  return s.addPt(x,y);
}

extern "C" int SplDelPt(float x, float y){
  return s.delPt(x,y);
}

extern "C" int SplClrPts(){
  return s.clrPts();
}

extern "C" int SplCalc(){
  return s.calc();
}

extern "C" void SplReset(){
  s.reset();
}

extern "C" int SplGetPt(int nr, float* x, float* y){
  return s.getPt(nr,x,y);
}

extern "C" int SplGetCoeff(int nr, float *a, float *b, float *c, float *d){
```

```
  return s.getCoeff(nr,a,b,c,d);
}

extern "C" int SplGetRange(int nr, float *x1, float *y1, float *x2, float *y2){
  return s.getRange(nr,x1,y1,x2,y2);
}

extern "C" char* SplErrText(int nr){
  return SplineErrText(nr);
}

extern "C" void SplPrint(){
  s.print();
}

extern "C" int SplMaplePlot(){
  return s.printMaplePlot();
}

extern "C" int SplPlot(){
  return s.plot();
}
```

C.7 main.C (Testprogramm *spline*)

```
//  LAST EDIT: Sun Jun 25 19:58:00 1995 by Burkhard Reike (reike)
#include <stdio.h>
#include <string.h>
#include "table.h"

extern int p1parse();
extern int p2parse();
extern FILE *p1in, *p2in;

FILE *vecout, *namout;

Signals signals;

main(int argc, char**argv){
  ++argv, --argc;
  if (argc < 2){
    printf("Usage: mkvec <signaldescr-file> <monitorfile>\n");
    return 0;
  }

  p1in = fopen(argv[0],"r");
  p1parse();
  fclose(p1in);

  namout = fopen("names.map","w");
  int i,j;
  Signal* si;
  TesterMode m;
  for (i=0; i<signals.len; i++){
    si = signals.sigs[i];
    fprintf(namout,"%s %d;\n",si->name(),i);

/*
    printf("Signal: %s\n", si->name());
```

```
    for (j=0; j<si->entries(); j++){
      m = si->getDriver(j).mode;
      printf("mode: ");
      if (m == drive) printf("drive ");
      if (m == strobe) printf("strobe ");
      if (m == undef) printf("undef ");
      printf("name: %s\n",si->getDriver(j).name);
    }
*/
  }
  fprintf(namout,"END\n");
  fclose(namout);

  p2in = fopen(argv[1],"r");
  vecout = fopen("vectors.vec","w");
  p2parse();
  fclose(p2in);
  fclose(vecout);
}

typedef struct dummy2{
  char name[MAXSTRLEN];
  Value value;
} MonResult;

MonResult mon[MAXSIGNALS];
int mlen = 0;

void addMon(char* name, Value val){
  strcpy(mon[mlen].name,name);
  mon[mlen++].value = val;
}

void printvector(){
  int i,j,k;
  char* n;
  Signal* si;
  int match;

  // update
  for (i=0; i<signals.len; i++){
    match = 0;
    si = signals.sigs[i];
    for (j=0; j<si->entries() && !match; j++){
      n = si->getDriver(j).name;
      for (k=mlen-1; k>=0 && !match; k--)
if (!strcmp(n,mon[k].name)){
  si->value = mon[k].value;
  si->mode = si->getDriver(j).mode;
  match = 1;
}
    }
  }

  mlen = 0;

  // output

  Value svalue;
```

```
  TesterMode smode;

  fprintf(vecout,"-");
  for (i=0; i<signals.len; i++){
    si = signals.sigs[i];
    svalue = si->value;
    smode = si->mode;
    if (smode == undef) fprintf(vecout,"*");
    if (smode == strobe){
      if (svalue == L) fprintf(vecout,"0");
      if (svalue == H) fprintf(vecout,"1");
      if (svalue == Z) fprintf(vecout,"Z");
    }
    if (smode == drive){
      if (svalue == L) fprintf(vecout,"L");
      if (svalue == H) fprintf(vecout,"H");
      if (svalue == Z) fprintf(vecout,"*"); // 'Z' can only be strobed
    }
  }
  fprintf(vecout,";\n");
}
```

Anhang D

Sourcen der SFU

D.1 Konfigurationsfile für Synopsys

```
search_path = {. /homes/prometheus/EUROCHIP_CDROM_LIBs/ES2/libraries
/homes2/zeus/synopsys/libraries/syn}
link_library = ecpd10_ind.db
target_library = ecpd10_ind.db
symbol_library = { ecpd10_s2030.sdb }
synthetic_library = standard.sldb

alias swl set_wire_load

view_num_lines_to_auto_scroll = 0
edifin_array_range_extraction_style    =   "%s<%d:%d>"
edifout_array_member_naming_style    =   "%s<%d>"
edifout_array_range_naming_style    =   "%s<%d:%d>"
edifout_add_power_and_ground_to_interfaces = "false"
edifout_skip_port_implementations = "false"
edifout_transform_to_origin = "false"
edifout_instantiate_ports = "true"
single_group_per_sheet = "true"
use_port_name_for_oscs = "false"
combine_vertical_logic_groups = "false"
duplicate_ports = "true"
edifout_power_and_ground_representation = "cell"
edifout_pretty_print = "true"
edifout_no_array = "true"
edifout_netlist_only = true

```

D.2 Makefile

```
all: SFU_TYPES.sim SFHW_DESCRIPTIONS.sim SFU_COMPONENTS.sim SFHW.sim OPREG.sim RESREG.sim RESREGVALID.sim WRITE_HOLD_STAGE.sim WRITE_STAGE.sim EXECUTE_STAGE.sim DECODE_STAGE.sim SFU_CONTROL.sim SFU.sim SFU_TEST.sim SFU_CONFIG.sim SFU_TEST_CONFIG.sim

##########################################################################

SFU_TEST.sim: SFU_test.vhdl SFU.sim
vhdlan SFU_test.vhdl
touch SFU_config.vhdl


##########################################################################
```

```
SFU.sim: SFU.vhdl SFU_TYPES.sim SFHW_DESCRIPTIONS.sim SFU_COMPONENTS.sim SFU_CONTROL.sim
vhdlan SFU.vhdl
touch SFU_test.vhdl


#########################################################################

SFU_CONTROL.sim: SFU_control.vhdl SFU_TYPES.sim SFU_COMPONENTS.sim DECODE_STAGE.sim EXECUTE_STAGE.sim WRITE_STAGE.sim WRITE_HOLD_STAGE.sim
vhdlan SFU_control.vhdl
touch SFU.vhdl


DECODE_STAGE.sim: decode_stage.vhdl SFU_TYPES.sim
vhdlan decode_stage.vhdl
touch SFU_control.vhdl

EXECUTE_STAGE.sim: execute_stage.vhdl SFU_TYPES.sim
vhdlan execute_stage.vhdl
touch SFU_control.vhdl

WRITE_STAGE.sim: write_stage.vhdl SFU_TYPES.sim
vhdlan write_stage.vhdl
touch SFU_control.vhdl

WRITE_HOLD_STAGE.sim: write_hold_stage.vhdl SFU_TYPES.sim
vhdlan write_hold_stage.vhdl
touch SFU_control.vhdl

#########################################################################

RESREG.sim: ResReg.vhdl SFU_TYPES.sim
vhdlan ResReg.vhdl
touch SFU.vhdl

RESREGVALID.sim: ResRegValid.vhdl SFU_TYPES.sim
vhdlan ResRegValid.vhdl
touch SFU.vhdl

OPREG.sim: OpReg.vhdl SFU_TYPES.sim
vhdlan OpReg.vhdl
touch SFU.vhdl

SFHW.sim: SFHW.vhdl SFU_TYPES.sim SFHW_DESCRIPTIONS.sim
vhdlan SFHW.vhdl
touch SFU.vhdl


#########################################################################

SFU_COMPONENTS.sim: SFU_components.vhdl SFU_TYPES.sim SFHW_DESCRIPTIONS.sim
vhdlan SFU_components.vhdl
touch SFHW.vhdl OpReg.vhdl ResReg.vhdl write_hold_stage.vhdl
write_stage.vhdl execute_stage.vhdl decode_stage.vhdl

SFHW_DESCRIPTIONS.sim: SFHW_descriptions.vhdl SFU_TYPES.sim
vhdlan SFHW_descriptions.vhdl
touch SFU_components.vhdl
```

```

SFU_TYPES.sim: SFU_Types.vhdl
vhdlan SFU_Types.vhdl
touch SFHW_descriptions.vhdl SFU_components.vhdl SFHW.vhdl OpReg.vhdl
ResReg.vhdl write_hold_stage.vhdl write_stage.vhdl execute_stage.vhdl
decode_stage.vhdl


#########################################################################

SFU_CONFIG.sim: SFU_config.vhdl
vhdlan SFU_config.vhdl

SFU_TEST_CONFIG.sim: SFU_config.vhdl
vhdlan SFU_config.vhdl

clean:
@/usr/bin/rm -f *.mra *.sim *command.log *sge.db *.sym l.out


```

D.3 Typendefinitionen

```
--------------------------------------------------------------------------------
--
-- Name: SFU Types
--
--------------------------------------------------------------------------------
--
-- Function:
-- Type declarations for all global used data.
--
--------------------------------------------------------------------------------
--
-- Parameters: [none]
--
--------------------------------------------------------------------------------
--
--  Author: Andreas Gnther
--
--  Date:   16-6-94
--
--  Aenderung: Lars Werner (15.3.95)
--
--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use IEEE.std_logic_1164.all;


--------------------------------------------------------------------------------
--
--  Package
```

```
--

package SFU_Types is


--------------------------------------------------------------------------------
-- bit vectors

subtype bit_32  is std_logic_vector(31 downto 0);
subtype bit_24 is std_logic_vector(23 downto 0);
subtype bit_19 is std_logic_vector(18 downto 0);
subtype bit_9  is std_logic_vector(8 downto 0);
subtype bit_6  is std_logic_vector(5 downto 0);
subtype bit_5  is std_logic_vector(4 downto 0);
subtype bit_4   is std_logic_vector(3 downto 0);
subtype bit_3   is std_logic_vector(2 downto 0);
subtype bit_2 is std_logic_vector(1 downto 0);
subtype bit_1 is std_logic;


--------------------------------------------------------------------------------
-- arrays of the above bit_vectors

type arr_bit_32 is array( natural range <> ) of bit_32;
type arr_bit_24 is array( natural range <> ) of bit_24;
type arr_bit_1 is array( natural range <> ) of bit_1;


--------------------------------------------------------------------------------
-- constant bit vectors

constant Z_vector : bit_32 := "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
constant Z_24_vector : bit_24 := "ZZZZZZZZZZZZZZZZZZZZZZZZ";
constant m_inputs : Natural       := 8; -- 8 Opregister


--------------------------------------------------------------------------------
-- opcodes

-- formats:

constant fmt_Load : bit_2 := "11"; -- for LDC and STC
constant fmt_Arith : bit_2 := "10"; -- for other instructions

-- load and store instructions:

constant op_LDC : bit_6 := "110000";
constant op_LDDC : bit_6 := "110011";
constant op_STC : bit_6 := "110100";
constant op_STDC : bit_6 := "110111";

-- other instructions:

constant op_CLDC : bit_6 := "110110";
constant op_Cxx : bit_6 := "110111"; -- specified by cop_...

-- coprocessor specific opcodes:

constant cop_CMOVE : bit_9 := "000000001";
```

```
constant cop_CRST : bit_9 := "000000010";
constant cop_CNOP : bit_9 := "000000000";

-- SFHW
type acht_wert is range 0 TO 7;

end SFU_Types;

package body SFU_Types is end;


```

D.4 Diverse Basisbeschreibungen

```
--------------------------------------------------------------------------------
--
-- Name: SFU components
--
--------------------------------------------------------------------------------
--
-- Function:
--  Entity descriptions of all components.
--
--------------------------------------------------------------------------------
--
-- Parameters: [none]
--
--------------------------------------------------------------------------------
--
--  Author: Andreas Guenther
--
--  Date:   16-6-94
--
--  Aenderung: Lars Werner (6.6.95), Thorsten Kukuk (22.3.1995)
--
--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use IEEE.std_logic_1164.all;

use WORK.SFU_Types.all;
use  WORK.SFHW_descriptions.all;


--------------------------------------------------------------------------------
--
-- Package
--

package SFU_components is

  component SFU

```

```
port ( -- Interface to the IU:

D : inout bit_32;
INST : in bit_1;
CNULL : out bit_1;
CINS1 : in bit_1;
CINS2 : in bit_1;
nMDS : in bit_1;
FLUSH : in bit_1;
CCC : out bit_2;
CCCV : out bit_1;
CLK : in bit_1;
nRESET : in bit_1;
nCHOLD : out bit_1;
nBHOLD : in bit_1;
nFHOLD : in bit_1;
FCCV : in bit_1;
nMHOLDA : in bit_1;
nMHOLDB : in bit_1;
nCP : out bit_1
);

  end component;


  component SFU_control

port ( -- Interface to the processor:

D : inout bit_32;
INST : in bit_1;
CINS1 : in bit_1;
CINS2 : in bit_1;
CCC : out bit_2;
CNULL : out bit_1;
CHOLD : out bit_1;
HOLD : in bit_1;
FLUSH : in bit_1;

-- Interface to the SFU bus:

RDread : in bit_24;
RDwrite : out bit_24;
RWA : out bit_3;
RRA : out bit_2;
RWR : out bit_1;
RRD : out bit_1;
RVALID : in bit_1;
           VALID : in arr_bit_1 (0 to m_inputs-1);
RRV : out bit_1;

-- others:

CLK : in bit_1;
MDS : in bit_1;
eRESET : in bit_1;
RESET : out bit_1
);

```

```
  end component;


  component decode_stage

port ( -- Interface to the IU:

D : in bit_32;
INST : in bit_1;
CINS1 : in bit_1;
CINS2 : in bit_1;

-- Interface to the execute stage:

Instruction: out bit_5;

-- others:

FLUSH : in bit_1;
HOLD : in bit_1;
MDS : in bit_1;
CLK : in bit_1;
                iRESET : in bit_1;
RESET : out bit_1
);

  end component;


  component execute_stage

port ( -- Interface to the IU:

CCC : out bit_2;
CHOLD : out bit_1;
CNULL : out bit_1;
FLUSH : in bit_1;

-- Interface to the SFU bus:

RVALID : in bit_1;
            VALID : in arr_bit_1 (0 to m_inputs-1);

-- Interface to other stages:

Instruction_in:
  in bit_5;
Instruction_out:
  out bit_5;

-- others:

HOLD : in bit_1;
CLK : in bit_1;
RESET : in bit_1
);

  end component;
```

```
  component write_stage

port ( -- Interface to the IU:

D : out bit_32;
FLUSH : in bit_1;

-- Interface to the SFU bus:

RD : in bit_24;
RRA : out bit_2;
RRD : out bit_1;
RRV : out bit_1;

-- Interface to the execute stage:

Instruction:
in bit_5;

-- Interface to the write-hold stage

Instruction_out:
out bit_5;

-- others:

HOLD : in bit_1;
CLK : in bit_1;
RESET : in bit_1

);

  end component;


  component write_hold_stage

port ( -- Interface to the IU:

D : in bit_32;
FLUSH : in bit_1;

-- Interface to the SFU bus:

RD : out bit_24;
RWA : out bit_3;
RWR : out bit_1;

-- Interface to the execute stage:

Instruction:
  in bit_5;

-- others:

MDS : in bit_1;
HOLD : in bit_1;
CLK : in bit_1;
```

```
RESET : in bit_1

);

  end component;


  component OpReg

generic ( adr : Natural;  width : Natural := 32);

        port (  -- Interface to the SFU bus:

                RD      : in    bit_24;
                RWA     : in    bit_3;
                RWR     : in    bit_1;


                -- output to the SFHW:

                READ    : out   bit_24;
                VALID   : inout bit_1;
                WRITE   : in    bit_24;
                RV      : in    bit_1;
                SET     : in    bit_1;
                RESET   : in    bit_1;
CLK : in bit_1
        );

  end component;


  component ResReg

generic ( adr : Natural;  width : Natural := 32 );

port ( -- Interface to the SFU bus:

                RD      : out   bit_24;
                RRA     : in    bit_2;
                RRD     : in    bit_1;

                -- input from the SFHW:

                Result    : in  bit_24;
                SetResult : in  bit_1;
CLK : in bit_1
);
  end component;

  component ResRegValid

        port (  -- Interface to the SFU bus:

                RRV     : in    bit_1;

                -- input from the SFHW:

                SRV     : in    bit_1;
```

```

                -- others:

                RVALID  : out   bit_1
        );

  end component;


-- Beschreibung:
-- read   : Datenleitung vom Opregistern zum Rechenkern, zum Lesen.
-- write  : Datenleitung vom Rechenkern zum Opregister, zum Schreiben.
-- rv     : Signalleitung, die der Rechenkern zum Loeschen des Validflags des
--          Opregisters benutzt.
-- set    : Signalleitung, ueber das der Rechenkern das Opregister auffordert,
--          Daten von 'winput' zu uebernehmen.
-- valid  : Signalleitung, an dem das Valitbit des Opregisters liegt.
-- result : Datenleitung vom Rechenkern zum Resregister, zum Schreiben.
-- setresult : Signalleitung, ueber das der Rechenkern das Resregister
--             auffordert, Daten von 'output' zu uebernehmen.
-- CLK    : Signalleitung, an der der Systemtakt liegt.
-- reset  : Signalleitung, ueber die der Rechenkern 'resetet' wird.
-- rvalid : Signalleitung, die in den Kern und auf den SFU-Bus fuehrt.
--          Hat der Kern die Resultate berechnet, so setzt er rvalid auf
--          '1', indem er srv auf '1' setzt.  Ueber rvalid wird erkannt, dass
--          ein Teil der Ergebnisse in den Resregs liegt, ihr Inhalt wird
--          gelesen, und ueber den SFU-Bus wird dann rrv gesetzt, um rvalid
--          zurueckzusetzen, wodurch der Kern erkennt, dass er die naechste
--          Gruppe von Ergebnissen in die Resregs schreiben muss.
-- srv    : Signalleitung aus dem Rechenkern, setzt rvalid-Leitung auf '1'.

 component SFHW
        port (
                Read     : in    arr_bit_24(0 to n_inputs-1);
                Write    : out   arr_bit_24(0 to n_inputs-1);
                RV       : out   arr_bit_1 (0 to n_inputs-1);
                Set      : out   arr_bit_1 (0 to n_inputs-1);
                Valid     : in    arr_bit_1 (0 to n_inputs-1);
                Result   : out   arr_bit_24(0 to n_outputs-1);
                SetResult       : out   arr_bit_1 (0 to n_outputs-1);

                CLK     : in    bit_1;
                Reset   : in    bit_1;
                Rvalid  : in    bit_1;
                SRV     : out   bit_1
        );
  end component;

end SFU_components;
```

```
------------------------------------------------------------------------------
--
-- Name: SFU_config
--
------------------------------------------------------------------------------
--
-- Function:
--
--  Configurations of the SFU descriptions for simulating it with the
--  Synopsys tool "vhdldbx".
```

```
--
-------------------------------------------------------------------------------
--
--  Author: Andreas Guenther
--
--  Date:   17-6-94
--
-------------------------------------------------------------------------------


configuration SFU_config of SFU is

for structure

  for control: SFU_control
use entity WORK.SFU_Control(structure);
  end for;

end for;

end SFU_config;



configuration SFU_test_config of SFU_test is

for behavior

  for test_SFU: SFU
use entity WORK.SFU(structure);
  end for;

end for;

end SFU_test_config;


```

```
-------------------------------------------------------------------------------
--
-- Name: SFHW descriptions
--
-------------------------------------------------------------------------------
--
-- Function:
-- This is a package with contains the descriptions of all SFHW interfaces.
-- The ordinal number (address) of each SFHW (equal to the SFSU address) and
-- the number of input and output ports is defined here.
-- The descriptions of the SFHWs can be found in the file "SFHW.vhdl".
--
-------------------------------------------------------------------------------
--
-- Parameters: [none]
--
-------------------------------------------------------------------------------
--
--  Author: Andreas Guenther
--
--  Date:   16-6-94
--
```

```
--  Aenderung: Thorsten Kukuk (22.3.1995)
--
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use IEEE.std_logic_1164.all;

use WORK.SFU_Types.all;


-------------------------------------------------------------------------------
--
-- Package
--

package SFHW_descriptions is


-------------------------------------------------------------------------------
-- number of input and output ports of our SFHW:

constant n_inputs : Natural := 8; -- 8 Opregister
constant n_outputs : Natural := 4; -- 4 Resregister



-------------------------------------------------------------------------------
-- bitwidth of each input and output port:

type Register_Width is array(0 to 31) of Natural;

-- NEU: (8 Opregister a 24 Bit, 4 Resregister a 24 Bit)
constant op_width : Register_Width := (24,24,24,24, 24,24,24,24, 0,0,0,0, 0,0,0,0,
0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0);
constant res_width : Register_Width := (24,24,24,24, 0,0,0,0,     0,0,0,0, 0,0,0,0,
0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0);

end SFHW_descriptions;

package body SFHW_descriptions is end;
```

D.5 Verdrahtung der Komponenten

```
-------------------------------------------------------------------------------
--
-- Name: SFU_control
--
-------------------------------------------------------------------------------
--
-- Function:
--
-- This module contains the instruction pipeline to decode and execute the
-- special SFU instructions. It serves as an interface between the
```

```
-- processor bus and the SFU bus.
--
-------------------------------------------------------------------------------
--
-- Parameters:  [none]
--
-------------------------------------------------------------------------------
--
--  Author: Andreas Gnther
--
--  Date:   16-6-94
--
--  Aenderung:  Lars Werner (4.6.95)
--
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use IEEE.std_logic_1164.all;

use WORK.SFU_Types.all;
use  WORK.SFU_components.decode_stage;
use  WORK.SFU_components.execute_stage;
use  WORK.SFU_components.write_stage;
use  WORK.SFU_components.write_hold_stage;



-------------------------------------------------------------------------------
--
-- Interface
--

entity SFU_control is

port ( -- Interface to the processor:

D : inout bit_32;
INST : in bit_1;
CINS1 : in bit_1;
CINS2 : in bit_1;
CCC : out bit_2;
CNULL : out bit_1;
CHOLD : out bit_1;
HOLD : in bit_1;
FLUSH : in bit_1;

-- Interface to the SFU bus:

RDread : in bit_24;
RDwrite : out bit_24;
RWA : out bit_3;
RRA : out bit_2;
RWR : out bit_1;
```

```
RRD : out bit_1;
RVALID : in bit_1;
           VALID : in arr_bit_1 (0 to m_inputs-1);
RRV : out bit_1;

-- others:

CLK : in bit_1;
MDS : in bit_1;
eRESET : in bit_1;
RESET : out bit_1
);

end SFU_control;


--------------------------------------------------------------------------------
--
--  Architecture
--

architecture structure of SFU_control is


signal Inst_D_E, Inst_E_W, Inst_W_WH : bit_5; -- instruction pipeline
signal res : bit_1;

begin

RESET <= res;


-- the pipeline stages with the IU connection:

dec: decode_stage
port map (D, INST, CINS1, CINS2, Inst_D_E, FLUSH, HOLD, MDS, CLK, eRESET, res);

exe: execute_stage
port map (CCC, CHOLD, CNULL, FLUSH, RVALID, VALID, Inst_D_E, Inst_E_W, HOLD, CLK, res);

wri: write_stage
port map (D, FLUSH, RDread, RRA, RRD, RRV, Inst_E_W, Inst_W_WH, HOLD, CLK, res);

whd: write_hold_stage
port map (D, FLUSH, RDwrite, RWA, RWR, Inst_W_WH, MDS, HOLD, CLK, res);

end structure;
```

D.6 Register

```
--------------------------------------------------------------------------------
--
-- Name: OpReg
--
--------------------------------------------------------------------------------
--
-- Function:
--
```

```
-- Operand register. When it is selected by its address and enable line,
-- the input value is latched and the valid bit is set. The reset input
-- resets the valid bit.
--
------------------------------------------------------------------------------
--
-- Parameters:
--    Address of this register
--    Width of data in bits (optional, default is 32)
--     The input and output is always 32 bit width.
--
------------------------------------------------------------------------------
--
--  Author: Lars Werner
--
--  Date:   5-4-95
--
------------------------------------------------------------------------------


------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.conv_std_logic_vector;

use WORK.SFU_Types.all;


------------------------------------------------------------------------------
--
-- Interface
--

entity OpReg is

generic ( adr : Natural;  width : Natural := 32);

port ( -- Interface to the SFU bus:

RD : in bit_24;
RWA : in bit_3;
RWR : in bit_1;


-- output to the SFHW:

READ : out bit_24;
VALID : inout bit_1;
WRITE : in bit_24;
RV : in bit_1;
SET : in bit_1;
           RESET : in bit_1;
           CLK   : in bit_1
);

```

```
end OpReg;


--------------------------------------------------------------------------------
--
--  Architecture
--


architecture behavior of OpReg is

signal reg_adr : bit_5;
signal flag : bit_1;

begin

reg_adr <= conv_std_logic_vector(adr, 5);



process


begin

    -- daten vom Bus lesen falls REG gemeint und Daten gueltig
if RWR = '1' and VALID='0' and RWA = reg_adr(2 downto 0) then
          READ <= RD;
          flag <= '1';
        end if;
    -- VALID - Bit setzen
if flag = '1' then
          VALID <= '1';
          flag <= '0';
end if;
    -- VALID - Bit zuruecksetzen
        if RV = '1' then
    VALID <= '0';
        end if;
    -- Werte in Register schreiben
        if SET='1' and VALID='1'  then
          READ <= WRITE;
  end if;
    -- Reset
        if RESET='1' then
          flag <= '0';
        end if;

    --    wait for 10 ns;

end process;

end behavior;
```

```
-- @(#)ResReg.vhdl 1.13
--
--------------------------------------------------------------------------------
--
-- Name: ResReg
--
```

```
--------------------------------------------------------------------------------
--
-- Function:
--
-- This register (Result Register) latches the output of the SFHW and supplies
-- this data on the SFU bus upon request.
--
--------------------------------------------------------------------------------
--
-- Parameters:
--   * Address of the register
--   * Width of the latched data (optional, default is 32 bit)
--     The width of the input and output port is always 32 bit.
--
--------------------------------------------------------------------------------
--
--  Author: Andreas Guenther / Thorsten Kukuk
--
--  Date:   27-6-94 / 5.4.1995
--
--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.conv_std_logic_vector;

use WORK.SFU_Types.all;


--------------------------------------------------------------------------------
--
-- Interface
--

entity ResReg is

generic ( adr : Natural;  width : Natural := 32 );

port ( -- Interface to the SFU bus:

RD : out bit_24;
RRA : in bit_2;
RRD : in bit_1;

-- input from the SFHW:

Result   : in bit_24;
SetResult : in bit_1;
           CLK : in bit_1
);

end ResReg;

```

```

--------------------------------------------------------------------------------
--
--  Architecture
--

architecture behavior of ResReg is


signal data    : std_logic_vector(width-1 downto 0);
-- internal data latch
signal reg_adr :     bit_5; -- address of the register as bit_vector

begin


reg_adr <= conv_std_logic_vector(adr, 5);

process(SetResult)
begin

    -- SFHW wrote to register

    if SetResult=’1’ then -- connections to the SFHW
      data <= Result(width-1 downto 0); -- latch data
    end if;

end process;


process
begin

  if RRA=reg_adr(1 downto 0) and RRD=’1’ then
    RD(width-1 downto 0) <= data; -- put data to bus
  else
    RD <= Z_24_vector;
  end if;

  --wait for 10 ns;

end process;

end behavior;
```

D.7 Registermonitor

```
-- %W%
--
--------------------------------------------------------------------------------
--
-- Name: ResRegValid
--
--------------------------------------------------------------------------------
--
-- Function:
--
-- Valid Flag for all 4 result registers.
--
```

```
-------------------------------------------------------------------------------
--
--  Author: Thorsten Kukuk
--
--  Date:   1.3.1995/22.3.1995
--
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use IEEE.std_logic_1164.all;

use WORK.SFU_Types.all;


-------------------------------------------------------------------------------
--
-- Interface
--

entity ResRegValid is

port ( -- Interface to the SFU bus:

RRV : in bit_1;

-- input from the SFHW:

SRV : in bit_1;

-- others:

RVALID : out bit_1
);

end ResRegValid;


-------------------------------------------------------------------------------
--
--  Architecture
--

architecture behavior of ResRegValid is

begin

process (SRV, RRV)
begin

    if RRV='1' then
      RVALID <= '0';
    end if;

```

```
    if SRV='1' then
      RVALID <= '1';
    end if;

end process;

end behavior;
```

D.8 Pipelinestufe decode

```
-------------------------------------------------------------------------------
--
-- Name: decode_stage
--
-------------------------------------------------------------------------------
--
-- Function:
--
-- Reads all instructions from the data bus and latches the last two ones.
-- Decodes the instruction which is selected by the CINS input and supplies
-- the read address RRA on the SFU bus. The instruction is the forwarded to
-- the execute stage.
-- The IOPs which are necessary for multi-cycle instructions are generated
-- in this stage.
--
-------------------------------------------------------------------------------
--
-- Parameters: [none]
--
-------------------------------------------------------------------------------
--
--  Author: Andreas Gnther
--
--  Date:   16-6-94
--
--  Aenderung: Lars Werner (4.6.95)
--
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use IEEE.std_logic_1164.all;

use WORK.SFU_Types.all;


-------------------------------------------------------------------------------
--
-- Interface
--

entity decode_stage is

port ( -- Interface to the IU:
```

```

D : in bit_32;
INST : in bit_1;
CINS1 : in bit_1;
CINS2 : in bit_1;

-- Interface to the execute stage:

Instruction: out bit_5;

-- others:

FLUSH : in bit_1;
HOLD : in bit_1;
MDS : in bit_1;
CLK : in bit_1;
iRESET : in bit_1;
RESET : out bit_1
);

end decode_stage;


--------------------------------------------------------------------------------
--
--  Architecture
--

architecture behavior of decode_stage is

signal Inst_1, Inst_2 : bit_5; -- last two instructions
signal data  : bit_32;
begin


process(D)
begin

  data <= D;

end process;

process(CLK)
begin



  if CLK'event and CLK='1' then -- rising edge of CLK

    if (data(24 downto 20)="11011" and data(31 downto 30) = "10") or iRESET='1' then
      RESET <= '1';
    else
      RESET <= '0';
    end if;

    if (INST='1' and HOLD='0' and FLUSH='0') or (HOLD='1' and MDS='1' and INST='1') then
      if (data(24 downto 19) = op_LDC and data(31 downto 30) = "11") or (data(24 downto 19) = op_STC and data(31 downto 30) = "11") then -- LDC or STC
```

```
    Inst_2 <= Inst_1;
    Inst_1(4 downto 3) <= '0' & data(21);
    Inst_1(2 downto 0) <= data(27 downto 25);
   else -- CNOP
    Inst_2 <= Inst_1;
    Inst_1(4) <= '1';
   end if;
  end if;

 end if;

end process;


process(CLK)
begin

  -- when CINS1 or CINS2 is asserted, we know which instruction to execute

  if CLK'event and CLK='0' then
   if HOLD='0' and FLUSH='0' then

    if CINS1='1' then -- decode instruction in buffer 1
     Instruction <= Inst_1;
    elsif CINS2='1' then -- decode instruction in buffer 2
     Instruction <= Inst_2;
    else -- if no SFU instruction then
     Instruction(4) <= '1'; -- generate CNOP instruction
    end if;

   end if;
  end if;


 -- propagate the instruction to the execute stage

end process;


end behavior;
```

D.9 Pipelinestufe execute

```
-------------------------------------------------------------------------------
--
-- Name: execute_stage
--
-------------------------------------------------------------------------------
--
-- Function:
--
-- This stage reads the status bits from the SFU bus and sets the condition
-- codes according to them if a CRST or CSST instruction enters this stage.
-- The pipeline is freezed if there is a read instruction (STC or CMOVE)
-- for which the source value is not valid yet.
--
-------------------------------------------------------------------------------
--
-- Parameters: [none]
```

```
--
-------------------------------------------------------------------------------
--
--  Author: Andreas Gnther
--
--  Date:   16-6-94
--
--  Aenderung: Lars Werner (4.6.95)
--
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use IEEE.std_logic_1164.all;

use WORK.SFU_Types.all;

-------------------------------------------------------------------------------
--
-- Interface
--

entity execute_stage is

port ( -- Interface to the IU:

CCC : out bit_2;
CHOLD : out bit_1;
CNULL : out bit_1;
FLUSH : in bit_1;

-- Interface to the SFU bus:

RVALID : in bit_1;
            VALID : in arr_bit_1 (0 to m_inputs-1);

-- Interface to other stages:

Instruction_in:
  in bit_5;
Instruction_out:
  out bit_5;

-- others:

HOLD : in bit_1;
CLK : in bit_1;
RESET : in bit_1
);

end execute_stage;


-------------------------------------------------------------------------------
```

```
--
--  Architecture
--

architecture behavior of execute_stage is


signal hld : bit_1;
signal Inst : bit_5;

begin

process (CLK)
begin

  if CLK'event and CLK='0' then -- rising edge of CLK
    CCC <= "00";
    if HOLD='0' and FLUSH='0' then
      Instruction_out <= Inst;
    elsif FLUSH='1' then
      Instruction_out(4) <= '1';
    end if;

    CNULL <= hld;

  end if;

end process;


process(CLK)
begin

    -- hold pipeline when STC with invalid data:

  if CLK'event and CLK='1' then

    Inst <= Instruction_in;

    if Instruction_in(4 downto 3)="01"  then
      CHOLD <= not RVALID;
      hld <= not RVALID;

    elsif Instruction_in(4 downto 3)="00"  then
      case Instruction_in(2 downto 0) is

        when "000" => CHOLD <= VALID(0);
                      hld <= VALID(0);
        when "001" => CHOLD <= VALID(1);
                      hld <= VALID(1);
        when "010" => CHOLD <= VALID(2);
                      hld <= VALID(2);
        when "011" => CHOLD <= VALID(3);
                      hld <= VALID(3);
        when "100" => CHOLD <= VALID(4);
                      hld <= VALID(4);
        when "101" => CHOLD <= VALID(5);
                      hld <= VALID(5);
        when "110" => CHOLD <= VALID(6);
```

```
                      hld <= VALID(6);
        when "111" => CHOLD <= VALID(7);
                      hld <= VALID(7);
        when others => null;
      end case;
    end if;

    if RESET='1' then
      CHOLD <= '0';
      hld <= '0';
    end if;

  end if;

end process;


end behavior;
```

D.10 Pipelinestufe write

```
-------------------------------------------------------------------------------
--
-- Name: write_stage
--
-------------------------------------------------------------------------------
--
-- Function:
--
-- This stage reads the data from the SFU bus and writes them to the memory
--
-------------------------------------------------------------------------------
--
-- Parameters: [none]
--
-------------------------------------------------------------------------------
--
--  Author: Lars Werner
--
--  Date:   6-6-95
--
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use IEEE.std_logic_1164.all;

use WORK.SFU_Types.all;


-------------------------------------------------------------------------------
--
-- Interface
--
```

```
entity write_stage is

port ( -- Interface to the IU:

D : out bit_32;
FLUSH : in bit_1;

-- Interface to the SFU bus:

RD : in bit_24;
RRA : out bit_2;
RRD : out bit_1;
RRV : out bit_1;

-- Interface to the execute stage:

Instruction:
in bit_5;

-- Interface to the write-hold stage

Instruction_out:
out bit_5;

-- others:

HOLD : in bit_1;
CLK : in bit_1;
RESET : in bit_1

);

end write_stage;


-------------------------------------------------------------------------------
--
--  Architecture
--

architecture behavior of write_stage is

signal status  : bit_4;  -- mark here when an outputregister was read
signal Inst : bit_5;
signal data : bit_32; --copy of data_bus
begin

process (CLK)
begin
  if CLK'event and CLK='0'  then

    if FLUSH='0' and HOLD='0' then
      Instruction_out <= Inst;
    elsif FLUSH='1' then
      Instruction_out(4) <= '1';
    end if;

    if Inst(4 downto 3)="01" and FLUSH='0' and HOLD='0' then
```

```
      data(31 downto 8) <= RD;
      data(7 downto 0) <= "00000000";

    elsif HOLD='0' then
      data <= Z_vector;

    end if;

  end if;
end process;

process (data)
begin

  if Inst(4 downto 3) = "01" then
    D <= data;
     else
         D <= (others => 'Z');
  end if;

end process;

process (CLK,RESET)
begin

  if CLK'event and CLK='1' then

    Inst <= Instruction;

    if RESET = '1' then
       RRV <= '1';
       status <= "0000";

    else

      if Instruction(4 downto 3)="01" and FLUSH='0' and HOLD='0' then -- STC instruction
        RRA <= Instruction(1 downto 0);
        RRD <= '1';
if ((Instruction(1 downto 0) = "00" and status = "1110") or
    (Instruction(1 downto 0) = "01" and status = "1101") or
    (Instruction(1 downto 0) = "10" and status = "1011") or
    (Instruction(1 downto 0) = "11" and status = "0111")) then
          RRV <= '1';
          status <= "0000";
else
  case Instruction(1 downto 0) is
             when "00" => status(0) <= '1';
             when "01" => status(1) <= '1';
             when "10" => status(2) <= '1';
             when "11" => status(3) <= '1';
             when others => null;
           end case;
        end if;

      elsif FLUSH='0' and HOLD='0' then -- LDC or CNOP instruction
        RRD <= '0';
        RRV <= '0';
      end if;

```

```
    end if;

  end if;

end process;


end behavior;
```

D.11 Pipelinestufe write hold

```
-------------------------------------------------------------------------------
--
-- Name: write_hold_stage
--
-------------------------------------------------------------------------------
--
-- Function:
--
-- This stage reads the data from the memory and writes them to the SFU bus
--
-------------------------------------------------------------------------------
--
-- Parameters: [none]
--
-------------------------------------------------------------------------------
--
--  Author: Lars Werner
--
--  Date:   6-6-95
--
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use IEEE.std_logic_1164.all;

use WORK.SFU_Types.all;


-------------------------------------------------------------------------------
--
-- Interface
--

entity write_hold_stage is

port ( -- Interface to the IU:

D : in bit_32;
FLUSH : in bit_1;

-- Interface to the SFU bus:

```

```
RD : out bit_24;
RWA : out bit_3;
RWR : out bit_1;

-- Interface to the write stage:

Instruction:
  in bit_5;

-- others:

MDS : in bit_1;
HOLD : in bit_1;
CLK : in bit_1;
RESET : in bit_1

);

end write_hold_stage;


--------------------------------------------------------------------------------
--
--  Architecture
--

architecture behavior of write_hold_stage is

signal lastinst : bit_5;
signal data : bit_32;
begin

process (D)
begin

data <= D;

end process;

process (CLK)
begin

  if CLK'event and CLK='1' then

    if (Instruction(4 downto 3)="00" and FLUSH='0' and HOLD='0') then -- LDC instruction
      RD  <= data(31 downto 8);
      RWA <= Instruction(2 downto 0);
      RWR <= '1';
      lastinst <= Instruction;
    elsif (lastinst(4 downto 3)="00" and HOLD='1' and MDS='1')  then
      RD  <= data(31 downto 8);
      RWA <= lastinst(2 downto 0);
      RWR <= '1';
    else
      RWR <= '0';
    end if;

  end if;

```

```
end process;


end behavior;

```

D.12 Die Testumgebung

```
--------------------------------------------------------------------------------
--
-- Name: SFU_test
--
--------------------------------------------------------------------------------
--
-- Function:
--
-- This entity supplies an environment to simulate and debug the functions
-- of the SFU. A sequence of instructions is simulated and all relevant
-- signals are generated to pretend a real Integer Unit.
-- Should be used with the special test-SFHW.
--
--------------------------------------------------------------------------------
--
-- Parameters: [none]
--
--------------------------------------------------------------------------------
--
--  Author: Andreas Guenther
--
--  Date:   1-7-94
--
--  Aenderung: Lars Werner (5.4.95),
--
--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use IEEE.std_logic_1164.all;

use WORK.SFU_Types.all;
use WORK.SFU_components.SFU;


--------------------------------------------------------------------------------
--
-- Interface
--

entity SFU_test is end;


--------------------------------------------------------------------------------
--
--  Architecture
```

```
--

architecture behavior of SFU_test is

signal D : bit_32;
signal INST, CNULL, CINS1, CINS2, CCCV, CLK, nRESET, nMDS, FLUSH, FCCV  : bit_1;
signal nCHOLD, nBHOLD, nFHOLD, nMHOLDA, nMHOLDB, nCP : bit_1;
signal CCC : bit_2;

begin

test_SFU: SFU
port map (D, INST, CNULL, CINS1, CINS2, nMDS, FLUSH, CCC, CCCV, CLK,
                  nRESET, nCHOLD, nBHOLD, nFHOLD, FCCV, nMHOLDA, nMHOLDB, nCP);


process
variable i : acht_wert;
variable j : acht_wert;

begin

INST <= ’0’;
CINS1 <= ’0’;
CINS2 <= ’0’;
CLK <= ’0’;
nBHOLD <= ’1’;
nFHOLD <= ’1’;
FCCV <= ’1’;
nMHOLDA <= ’1’;
nMHOLDB <= ’1’;
nMDS <= ’1’;
FLUSH <= ’0’;


-- Perform a reset

nRESET <= ’0’;

wait for 40 ns; -- operate at 25 MHz
CLK <= ’1’;
wait for 40 ns;
CLK <= ’0’;

wait for 40 ns;
CLK <= ’1’;
wait for 40 ns;
CLK <= ’0’;

nRESET <= ’1’;

INST  <= ’1’;

D <= fmt_Load & "00000" & op_LDC & "00000" & "10000000000000";
wait for 40 ns;
CLK <= ’1’;
wait for 40 ns;
CLK <= ’0’;

```

```
CINS1 <= '1';

D <= fmt_Load & "00001" & op_LDC & "00000" & "10000000000000";
wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

D <= fmt_Load & "00010" & op_LDC & "00000" & "10000000000000";
wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

INST <= '0';

D <= "00000000000000000000000100000000";
wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

CINS1 <= '0';

D <= "00000000000000000000001000000000";
wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

D <= "00000000000000000000001100000000";
wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

INST  <= '1';

D <= fmt_Load & "00011" & op_LDC & "00000" & "10000000000000";
wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

CINS1 <= '1';

D <= fmt_Load & "00100" & op_LDC & "00000" & "10000000000000";
wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

D <= fmt_Load & "00101" & op_LDC & "00000" & "10000000000000";
wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

INST <= '0';
```

```
D <= "00000000000000000000010000000000";
wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

CINS1 <= '0';

D <= "00000000000000000000010100000000";
wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

D <= "00000000000000000000011000000000";
wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

INST  <= '1';

D <= fmt_Load & "00110" & op_LDC & "00000" & "10000000000000";
wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

CINS1 <= '1';

D <= fmt_Load & "00111" & op_LDC & "00000" & "10000000000000";
wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

D <= fmt_Load & "00000" & op_STC & "00000" & "10000000000000";
wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

INST <= '0';

D <= "00000000000000000000011100000000";
wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

CINS1 <= '0';

D <= "00000000000000000000100000000000";
wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

```

```
D <= Z_vector;

wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

INST  <= '1';

D <= fmt_Load & "00001" & op_STC & "00000" & "10000000000000";
wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

CINS1 <= '1';

D <= fmt_Load & "00010" & op_STC & "00000" & "10000000000000";
wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

D <= fmt_Load & "00011" & op_STC & "00000" & "10000000000000";
wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

INST <= '0';

D <= Z_vector;

wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

CINS1 <= '0';

wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

wait for 40 ns;
CLK <= '1';
wait for 40 ns;
CLK <= '0';

wait;
end process;

end behavior;
```

Literaturverzeichnis

[1] Studienarbeit Entwickelung einer Hardware–Schnittstelle zur effizienten Anbindung von Coprozessoren, Andreas Günther, Wolfram Hardt, Juli 1994

[2] Zycad InCA-Manual, August 1994

[3] Spline-Interpolation, Ralf Jungblut 1. Seminarband

[4] Hans Ulrich Post, Entwurf und Technologoie hochintegrierter Schaltungen

[5] Handbuch der Elektronik, Analogtechnik

[6] Handbuch der Elektronik, Digitaltechnik

[7] EDIF Steering Comitee, EDIF Specification (1985)

[8] Digest of Technical Papers, EDIF User Group Workshop (1985)

[9] K. R. Bennett, D. L. Chalmers, An analysis of EDIF Version 2 0 0 - How it has changed from Version 1 0 0 - (1987)

[10] Gert Kachel, Zusammenfassung: EDIF (1986)

[11] John D. Crawford, An Electronig Design Interchange Format (1984)

[12] SPeeDCHART First Time Users Guide, Speed SA

[13] Peter J. Ashenden, The VHDL Cookbook, First Edition, July 1990

[14] Numerische Mathematik 1, J. Stoer, 5. Auflage, Springer-Verlag, Berlin Heidelberg, 1989, ISBN 3-540-51481-3

[15] Numerische Mathematik, H. R. Schwarz, 2. Auflage, B. G. Teubner, Stuttgart, 1988, ISBN 3-519-12960-4

[16] SRAM-Paper, Wolfram Hardt, U-GH Paderborn, 1995

[17] Synopsys Design Compiler Tutorial, Version 3.0 Preliminary Draft

[18] High-Level VHDL Optimization, Transformation, and Analysis, 1993

[19] The VHDL Cookbook, Peter J. Ahsenden, 1990

[20] Thomas W. Williams and K.P. Parker. *Design for Testability - - a survey. IEEE Transactions on Computers, C31(1):2 -15, January 1982.*

[21] H.Wojtkowiak. *Test und Testbarkeit digitaler Schaltu ngen. Leitfäden und Monographien der Informatik. B.G.Teubner Stuttgart, 1988.*

[22] F.J.Rammig. *Systematischer Entwurf digitaler Systeme. Leitf äden und Monographien der Informatik. B.G.Teubner Stuttgart, 1989.*

[23] Charles R.Kime and Kewal K.Saluja. *A tutorial on Build-In S elf-Test. Part 1: Principles. IEEE Design and Test of Computers, March 1993.*

[24] Charles R.Kime and Kewal K.Saluja. *A tutorial on Build-In S elf-Test. Part 2: Applications. IEEE Design and Test of Computers, June 1993.*

[25] S.W. Golomb. *Shift Register Sequences, Aegean Park Press, Laguna Hills, Calif, 1982*

[26] W.W.Peterson and E.J. Weldon. *Error-Correcting Codes, John Wiley & sons, New York, 1972*

[27] P.J. Roth. *Diagnosis od Automata Failures: A Calculus and a Method, IBM Journal, Juli 1966, pp.278-291*

[28] P.J.Roth, Bouricius, W.G.,Schneider, P.R.. *Programmed Algor ithms to Compute Tests to Detect and Distinguish between Failures in Logic Cuircits, IEEE Transactions on Electronic Computers, Vol. EC-16, no,5,October*

[29] E.Hörbst, M.Nett, H.Schwärtzel, Einführung in die Standardmethoden des Layoutdesigns (Kapitel 1.6), Venus Entwurf von VLSI- Schaltungen, S. 54-63, 1986

[30] E.Hörbst, M.Nett, H.Schwärtzel, Layoutdesignmethoden (Kapitel 3), Venus Entwurf von VLSI-Schaltungen, S. 105-143, 1986

[31] Peter Marwedel, Layout-Synthese, Synthese und Simulation von VLSI-Systemen (Kapitel 4), 1992

[32] K. Mehlhorn, Über Verdrahtungsalgorithmen, Informatik- Spektrum (1986) 9:227-234

[33] Hierachical Wire Routing, IEEE Transaction on Computer-Aided Design, p. 223-234 ,Vol. CAD-2, No. 4, October 1983

[Cyp90] Cypress Semiconductor 'SPARC RISC User's Guide.' Ross Technologie, 1990

[Xilinx] *Xilinx - The Programmable Logic Data Book*, 1994

[FPGA] H. Grünbacher und R.W. Hartenstein. *FPGA - Field Programmable Gate Arrays*, Seite 11 - 25

[INCA] *InCA - Manual*

[Esch] B. Eschermann. *Funktionaler Entwurf digitaler Schaltungen.* Springer-Lehrbuch, 1993.

[Pals] PAL Device Data Book, Advanced Micro Devices, 1990

[Esc93] B. Eschermann. *Funktionaler Entwurf digitaler Schaltungen.* Springer-Lehrbuch, 1993.

[Nag93] C. Nagel, editor. *Projektgruppe Systementwurf.* WS1992/93 SS1993

[Bui91] F. Buijs. Synthesis of multi-level logic with one symbolic input. In *Proceedings of the European Conference on Design Automation - EDAC,* pages 60 - 64. IEEE, 1991.

[Bui92] F.Buijs. Alu synthesis from hdl descriptions to optimized multi-level logic. In *European Design Automation Conference - Eurodac,* pages 175 - 180. IEEE, 1992

[Ram93] F. J. Rammig. *Vorlesung Rechnerarchitektur* WS1993/94

[Ram94] F. J. Rammig. *Vorlesung Spezielle Probleme des Hardwareentwurfs* SS1994

[Lsi90.1] LSI LOGIC *L64811 SPARC Integer Unit Technical Manual* 1990

[Lsi90.2] LSI LOGIC *L64814 Floating-Point Unit Technical Manual* 1990

[Pat90] John L. Hennessy & David Patterson *Computer Architecture A Quantitative Approach*, Morgan Kaufmann Publishers, Inc. 1990

[Rolf77] Hoffmann, Rolf: Rechenwerke und Mikroprogrammierung (1977)

[Baer82] Baer, Jeanloup: Computer Systems Architecture (1982)

[Mano76] Mano, Moshe: Computer System Architecture (1976)

[Tester] The Tester Program, *VA-II PC Interface*, InCA-Ordner

[RPbasics] *Paradigm RP – Learning the Basics*, ZYCAD-Ordner

[SunTech] M.Hall, J.Barry, *The SunTechnology Papers*, Springer Verlag, 1990

[ArchMan] *The SPARC^{TM} Architecture Manual*, Sun Microsystems Inc., 8. August 1987

[WWWasm] *http://www.uni-paderborn.de/software/gnu/gcc/gcc_8.html#SEC87*
bzw. von http://www.uni-paderborn.de/ aus: Fachbereiche, Informatik, Info's zum Rechenbetrieb, Benutzerinformationen, Die Info-Seiten der GNU-Software, gcc, GNU C Compiler, *Assembler Instructions with C Expression Operands*

[DOS87] Fa. DOSIS, *DACAPO II, User Manual, Version 3.0*, DOSIS GmbH, Dortmund, 1987.

[EwGe90] C.Ewering, G.Gerhardt, "PASS: High Level Synthesis" *16th Symposium Microprocessing and Microprogramming*, 8, 1990.

[GaDu92] D.Gajski, N.Dutt, *High-Level Synthesis. Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.

[GaKu83] D.Gajski, R.Kuhn, "Guest Editors' Introduction: New VLSI Tools" *IEEE Computer*, vol. 16, no. 12, pp. 11-14, December 1983.

[Hill74] F.J.Hill, "Introducing AHPL" *IEEE Computer*, 7(12), 12 1974.

[IEEE88] Design Automation Standards Subcommittee of the IEEE, *IEEE Standard VHDL Language Reference Manual (IEEE Std. 1076-87)*, IEEE Inc., New York, 1988

[KWK85] S.Y.Kung, H.J.Whitehouse, T.Kailath, *VLSI and Modern Signal Processing*, Prentice Hall, pages 256-264, 1985.

[Man87] H.De Man, J.Rabaey, "CATHEDRAL II: A Synthesis and Module Generation System for Multiprocessor Systems on a Chip" in: *G.De Micheli: Design Systems for VLSI Circuits-Logic Synthesis and Silicon Compilation*, Martinus Nijhoff Publishers, 1987.

[Mar93] P.Marwedel, *Synthese und Simulation von VLSI-Systemen*, Hanser Studien Bücher, 1993.

[MiLaDu92] P.Michel, U.Lauther, P.Duzy *The Synthesis Approach to Digital System Design*, Kluwer Academic Publishers, 1992.

[PaKn89a] P.G.Paulin, J.P.Knight, "Scheduling and Binding Algorithms for High-Level Synthesis" *Proceedings of the 26th ACM/IEEE Design Automation Conference (DAC)*, pages 1-6, 1989.

[PaKn89b] P.G.Paulin, J.P.Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASICs" *IEEE Transactions on CAD*, pages 661-679, June 1989.

[Paul88] P.G.Paulin, *High-Level Synthesis of Digital Circuits Using Global Scheduling and Binding Algorithms*, PhD thesis, Carleton University, 1988.

[Pilo83] R.Piloty, M.Barbacci, D.Borrione, D.Dietmeyer, F.Hill, P.Skelly, "CONLAN Report" *Lecture Notes in Computer Science No. 151*, Springer, 1983.

[Ramm80] F.J.Rammig, *Preliminary CAP/DSDL Language Reference Manual. Technical Report 129*, University of Dortmund, Dept. Computer Science, 1980.