Einführung in die SPARC-Architektur

mit Beispielen zur Assemblerprogrammierung unter Solaris

Bernd Rosenlecher

(2.4.4) Generiert am 9. Dezember 2015. Dieses Dokument wurde mit TEX/LATEX gesetzt.

Inhaltsverzeichnis

1 Vorwort

Diese Einführung wurde erstellt auf Anregung von Dr. Martin Lehmann und aufgrund eigener Ideen zur Unterstützung eines begleitenden Kurses zu seiner Vorlesung über Computer-Architektur im Rahmen der Technischen Informatik am Fachbereich Informatik der Universität Hamburg.

Die Idee ist, durch intensive Beschäftigung mit *einer* spezifischen Architektur allgemeine Konzepte erfahrbar zu machen und so das Verständnis zu fördern.

Hierfür wurde die SPARC v8 gewählt, Standard bei den Institutsrechnern, die den Studierenden ohnehin als Übungsmaschinen zur Verfügung stehen. Von Vorteil erweist sich, daß die SPARC in der inflationären »Computer-Literatur« praktisch nicht vorkommt, und man sich deshalb schon um die – üblicherweise nur auf englisch vorhandenen – Original-Quellen bemühen muß, in diesem Falle also *The SPARC Architecture Manual, Version 8* ([SPARC92]) und das *SPARC Assembly Language Reference Manual* ([SPARC00]).

Diese Einführung will nun nicht etwa die Lektüre der Originale ersetzen, sondern sie soll den Zugang zu dem nicht immer gleich durchschaubaren Material erleichtern.

In der Präsentation des Materials, der Beschreibungen und Tabellen habe ich mich größtenteils eng an die Quellen gehalten, oft auch in der geradezu legalistischen Diktion, um ein wenig das Flair dieser Art von Literatur zu übermitteln. Die Auswahl des Materials wurde durch den vorgegebenen Rahmen begrenzt. So mußte auf eine Beschreibung jedes einzelnen Befehls mit Syntax, Semantik und Beispielen verzichtet werden, denn dies hätte den Umfang mehr als verdreifacht.

Im Teil zur Assembler-Programmierung wird eine praxisorientierte Kurzeinführung in das C-Entwicklungssystem geboten, zum Assembler wurde eine diesem Zweck angepaßte Auswahl aus Suns Handbuch übernommen. Für weitergehende Informationen bediene man sich der Literaturhinweise. Die Beispielprogramme sind so gestaltet, daß sie einen ersten Einblick in das Arbeiten der Architektur vermitteln und als Ausgangspunkt für eigene Experimente dienen können.

Meinen Kollegen Helmut Schönborn, Wolfram Roisch und Alexis Pangalos danke ich für aufmerksames Korrekturlesen und mancherlei Hinweise zur ersten Version, die dann im Wintersemester 2002/2003 sogleich einem Praxistest unterzogen wurde.

Die zweite Auflage dieser Einführung wurde geringfügig verbessert und erweitert, ein Index wurde beigefügt. Noch vorhandene (hoffentlich die letzten!) Fehler konnten gefunden und beseitigt werden. Für zweckdienliche Hinweise dazu danke ich den Studierenden und Übungsleitern der Kurse des WS 2002/2003.

Mein besonderer Dank gilt Herrn Dr. Martin Lehmann, der dies Projekt als Mentor förderte, mehrfach die Mühe des Korrekturlesens auf sich nahm und durch präzise Hinweise sachlicher und sprachlicher Art zur Verbesserung und Klärung beigetragen hat. Außerdem danke ich ihm für viele anregende Freitagnachmittag-Diskussionen.

Alle noch verbleibenden Fehler und Unschönheiten fallen selbstverständlich in meine Verantwortung.

– Bernd Rosenlecher
Hamburg, September 2004

2 Einleitung

SPARC[1] ist nicht die Bezeichnung für einen Chip, sondern die Beschreibung einer Prozessor-Architektur (ISA[2]), die von Sun Microsystems Mitte der 80er Jahre, aufbauend auf Forschungen an der Berkeley-Universität unter Prof. David Patterson (RISC I, RISC II, 1980-82) als sog. RISC[3]-Architektur, entwickelt wurde. Sun erklärte das Design zum offenen Standard und übergab seine Weiterentwicklung der SPARC International Inc., die auch die definierenden Dokumente und Compatibilitätsanforderungen (u.a. SCD[4] und ABI[5]) herausgibt.

Nach diesem publizierten Design können nun beliebige Hardware-Hersteller unter Lizenz Chips oder Chipsätze herstellen. Die erste kommerzielle Implementation, benutzt in der Sun-4 Workstation, basierte auf einer Ausführung mit Gate Arrays von Fujitsu (ca. 20000 Gates). Inzwischen sind weitere Implementationen von vielen verschiedenen Herstellern als Mehrchip- oder Einchip-Ausführungen dazugekommen: Fujitsu, Cypress Semiconductor, Bipolar Integrated Technology, LSI Logic, Texas Instruments (MicroSPARC, SuperSPARC), Solbourne/Matsushita, Philips, ...

SPARC kann als typische RISC-Architektur – mit interessanten Besonderheiten – betrachtet werden. Sie wurde seit ihrer ersten Definition in verschiedenen Stufen entsprechend dem jeweiligen Stand der Technik weiterentwickelt. Als Basis dieser Abhandlung dient die Version 8, kurz SPARC v8, kürzer noch v8 genannt. Das definierende Dokument dazu ist [SPARC92]. Um die weitere Entwicklung aufzuzeigen, werden abweichende Eigenheiten der Version 9 (einer 64-Bit-Architektur, siehe [SPARC95]) im Abschnitt 5, ab S. 33, in aller Kürze besprochen.

Zum besseren Überblick hier einige prägnante Eigenschaften der SPARC:

- linearer 32-Bit Adreßraum
- wenige und einfache Befehlsformate – Alle Befehle sind 32 Bit breit und auf 32-Bit-Grenzen im Speicher ausgerichtet. Es gibt nur drei Grundformate mit einheitlicher Belegung von Befehlscode und Registeradressen. Nur die Lade- und Speicherbefehle haben Zugriff auf Speicher und I/O.
- wenige Speicheradressierungsarten – Adressen werden entweder durch zwei Register oder durch ein Register und einen Direktwert angegeben.
- triadische Registeradressen – Die meisten Befehle haben als Operanden zwei Quellregister oder Quellregister und Direktwert, und legen das Ergebnis in einem Zielregister ab. Das Quellregister wird also nicht überschrieben.

[1] Acronym für Scalable Processor ARChitecture. Ein Verzeichnis der verwendeten Abkürzungen ist im Anhang zu finden.

[2] Instruction Set Architecture

[3] Reduced Instruction Set Computer, ein Computer mit reduziertem Befehlssatz, oder auch, anders ausgedrückt: mit einem Satz reduzierter Befehle, siehe [Dewar90], S. 237

[4] SPARC Compliance Definition

[5] Application Binary Interface

- ein großes verschiebbares Registerfenster, das auf einen Ausschnitt aus einer noch größeren Registerbank zeigt – Zu jedem Zeitpunkt sieht ein Programm 8 globale Register, sowie ein Fenster auf 24 weitere Register, die zur Übergabe von Parametern, für lokale Variablen, und für Rückgabewerte genutzt werden können, mit oder ohne den separat zu bedienenden Verschiebemechanismus.
- separater Satz von 32 Registern für Fließkomma-Operationen, beliebig konfigurierbar für 32-Bit, 64-Bit oder 128-Bit Operationen
- verzögerte Sprungbefehle – Der auf den Sprungbefehl folgende Befehl wird i.d.R. ausgeführt, seine Ausführung (und damit seine Wirkung) kann jedoch, abhängig vom sog. *Annul Bit* des Sprungbefehls, unterdrückt werden.
- schnelle Ausnahmebehandlung durch Sprung über Tabelle und Weiterschalten des Registerfensters
- Synchronisierungsbefehle für Multiprozessorbetrieb
- simpler Befehlssatz zur Einbindung eines optionalen Coprozessors

3 Architektur der SPARC Version 8

SPARC ist ein Architektur*modell*, das das Verhalten von Software auf einem solchen System definiert, nicht notwendig die Eigenschaften der Hardware einer spezifischen Implementation. SPARC Systemkomponenten erlauben verschiedene Varianten für I/O[6], MMU[7] und Cache-Subsysteme. Diese sollen von der jeweiligen Hardware- und Systemumgebung optimal definiert werden und sind normalerweise transparent für Anwendungs-Software. Als Entwurfsvorschlag wird eine Referenz-MMU spezifiziert, die ein weites Anwendungsspektrum abdeckt. Als Standard-Speichermodell ist TSO[8] spezifiziert, als weiteres Modell ist PSO[9] möglich.

Gleichheit im Verhalten von System-Software wird *nicht* gefordert. Das Vorhandensein von bestimmten Status- und Kontrollregistern, die nur mit Systemprivilegien sichtbar sind, und die entsprechenden zugehörigen Befehle oder deren Verhalten können sich also von Version zu Version ändern. Nicht alle Eigenschaften müssen in Hardware implementiert sein. Unterschiedliche Ausführungen bezüglich Befehlsparallelität und Ausnahmebehandlung sind möglich. Nicht implementierte Befehle können in Software emuliert werden. SCD[10]-Dokumente definieren die Anforderungen an eine Implementation. SPARC International stellt eine *SPARC Architecture Test Suite* sowie einen *SPARC Architectural Simulator* zur Verfügung.

[6] Input/Output
[7] Memory Management Unit
[8] Total Store Ordering
[9] Partial Store Ordering
[10] SPARC Compliance Definition

4 SPARC-Prozessor

SPARC beschreibt eine Architektur mit 32-Bit Ganzzahl und 32-, 64- und 128-Bit Fließkommazahlen als den grundlegenden Datentypen. Definiert sind allgemeine Ganzzahl-, Fließkommazahl- und spezielle Status- und Kontrollregister, sowie 72 Basisbefehle, alle kodiert im 32-Bit-Format. Lade- und Speicherbefehle adressieren einen linearen Adreßraum von 2^{32} Bytes. Zusätzlich werden Befehle für einen optionalen Coprozessor definiert.

4.1 Datentypen und -formate

SPARC kennt drei grundlegende Datentypen:

vorzeichenbehaftete Ganzzahl	8, 16, 32 und 64 Bit
nicht vorzeichenbehaftete Ganzzahl	8, 16, 32 und 64 Bit
Fließkommazahl	32, 64 und 128 Bit

Die Formate sind wie folgt definiert:

Byte	8 Bit
Halfword	16 Bit
Word	32 Bit
Tagged Word	32 Bit (30 Bit für den Wert, plus 2 Bit für die Kennung)
Doubleword	64 Bit
Quadword	128 Bit

Die vorzeichenbehafteten Ganzzahlformate beschreiben Zahlen im Zweierkomplement. Die nichtvorzeichenbehafteten Ganzzahlformate sind allgemein verwendbar, sie können verschiedene Objekte repräsentieren: Ganzzahlen, Boole'sche Werte, Bitmasken, Strings, etc. Das *Tagged* Format definiert ein Wort, in dem die beiden niederwertigsten Bit für das *Tag* (Etikett, Kennung) genutzt werden.

Die Fließkommaformate entsprechen dem Standard IEEE 754-1985.

4.2 Speicheradressierung und Ausrichtung

SPARC erfordert die korrekte Ausrichtung (*alignment*) von Code und Daten im Speicher, und zwar entsprechend ihrer Breite:

Code ist immer auf 32-Bit-Grenzen ausgerichtet,
Bytes können auf beliebigen Adressen liegen,
Halfwords müssen auf geraden Adressen liegen,
Words müssen auf ohne Rest durch 4 teilbaren Adressen liegen,
Doublewords müssen auf ohne Rest durch 8 teilbaren Adressen liegen,
Quadwords müssen auf ohne Rest durch 8 teilbaren Adressen liegen.

Zugriff auf eine nicht diesen Ausrichtungsrestriktionen entsprechende Adresse löst eine *mem_address_not_aligned*-Ausnahme aus.

adr-1	adr	adr+1	adr+2	adr+3	adr+4
	0x12	0x34	0x56	0x78	

Abbildung 1: Ein *Word* mit dem Wert 0x12345678 im *Big Endian*-Speicherformat

SPARC ist eine sog. *Big Endian*-Architektur, d.h. die Adresse eines Datums im Speicher entspricht immer der Adresse seines höchstwertigen Bytes. Sei z.B. `adr` die Adresse eines Word-Datums mit dem Wert `0x12345678`, dann enthält:

– das Byte bei `adr` den Wert `0x12`,
– das Byte bei `adr`+1 den Wert `0x34`,
– das Byte bei `adr`+2 den Wert `0x56`,
– das Byte bei `adr`+3 den Wert `0x78`,
– das Halfword bei `adr` den Wert `0x1234`,
– das Halfword bei `adr`+2 den Wert `0x5678`.

In gleicher Weise sind Double-, Quadword- und Fließkommaformate im Speicher abgelegt. Dies ist wichtig zu wissen, falls von einer SPARC abgelegte Daten von einer CPU mit anderer *Endianness* bearbeitet werden sollen (z.B. MIPS oder Intel x386), wenn Arrays kleiner Daten (z.B. Bytes) in größeren Einheiten verarbeitet werden sollen oder umgekehrt größere Daten (z.B. FP-Formate) in kleineren Einheiten zu verarbeiten sind, z.B. um einzelne Bit oder Bit-Felder zu extrahieren.

Die Ladebefehle für Sub-Wort-Größen Byte und Halfword existieren in jeweils zwei Varianten für vorzeichenbehaftete und vorzeichenlose Größen, da es hier auf die korrekte, vorzeichenrichtige Erweiterung auf 32 Bit ankommt, denn im Prozessor können in typischer RISC-Manier nur komplette 32-Bit-Werte in den Registern bearbeitet werden. Bei Wort- oder Doppelwort- Größen und zum Speichern ist dies natürlich nicht notwendig.

4.3 Funktionseinheiten

Ein SPARC-Prozessor besteht logisch aus einer Ganzzahleinheit IU[11], einer Fließkommaeinheit FPU[12], sowie einem optionalen Coprozessor CP mit jeweils eigenem Registersatz. Diese Organisation erlaubt maximale Befehlsausführungsparallelität zwischen den Einheiten. Alle Register, mit möglicher Ausnahme des Coprozessors, sind 32-Bit weit. Befehlsoperanden sind i.A. einzelne Register, Registerpaare oder -quadrupel.

[11] Integer Unit
[12] Floating-Point Unit

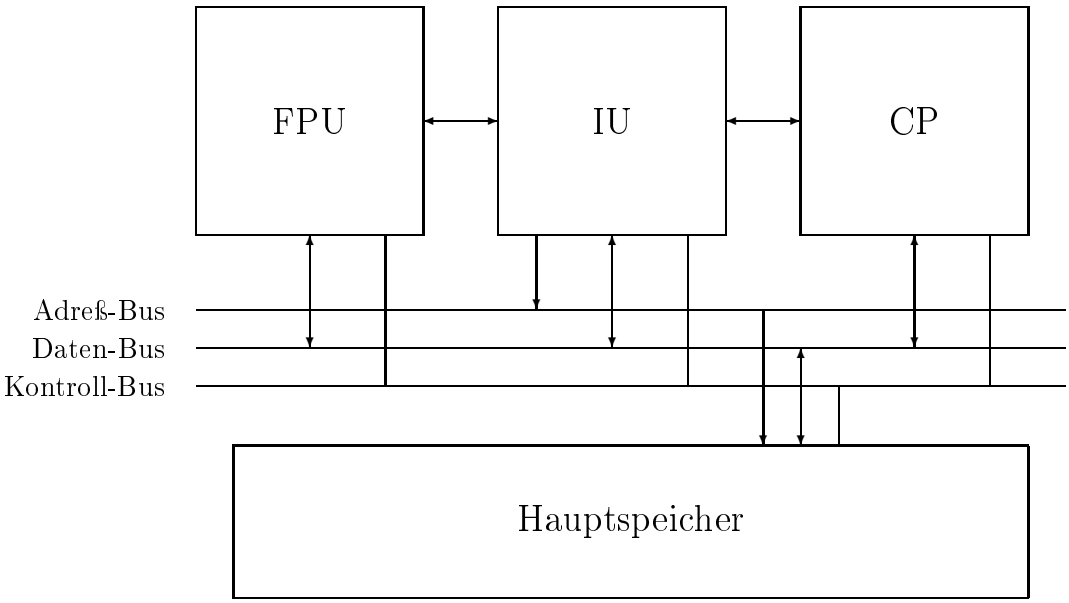

Abbildung 2: Logische Organisation der SPARC-Funktionseinheiten

4.4 Arbeitszustände und Operationsmodi

Der SPARC-Prozessor befindet sich immer in einem von drei möglichen Zuständen: Im Ausführungszustand (*execute state*), im Rücksetzzustand (*reset state*) oder im Fehlerzustand (*error state*). Der normale Arbeitszustand ist der Ausführungszustand.

Bei Auslösung einer synchronen Ausnahme, während der Zeit, in der Ausnahmen gesperrt sind, geht der Prozessor in den Fehlerzustand und hält an. Er bleibt in diesem Zustand, bis das Rücksetzsignal aktiviert wird, wofür externe Beschaltung verantwortlich ist. Daraufhin geht der Prozessor in den Rücksetzzustand, der solange andauert, bis das Rücksetzsignal aufgehoben wird, worauf der Prozessor in den Ausführungszustand übergeht. Er beginnt dann im Supervisor-Modus (s.w.u.) an der Adresse 0 im Supervisor-Befehlsadreßraum mit der Ausführung des ersten Befehls.

Während des Ausführungszustands befindet sich der SPARC-Prozessor in einem von zwei Operationsmodi, im **Supervisor**-Modus oder im **User**-Modus.

Im Supervisor- oder privilegierten Modus sind *alle* Befehle zugelassen, einschließlich der nur in diesem Modus erlaubten. Der Supervisor-Modus ist gemeinhin Betriebssystemroutinen und der Systemsoftware vorbehalten. Man gelangt in ihn nur durch einen Reset des Prozessors oder, vom User-Modus aus, durch die Ausführung einer *Trap* oder Ausnahmebehandlung.

Der User-Modus ist der übliche Modus für Anwendungs-Software, die somit nur die nichtprivilegierten Befehle nutzen kann und sonst auf den Aufruf von Systemroutinen angewiesen ist.

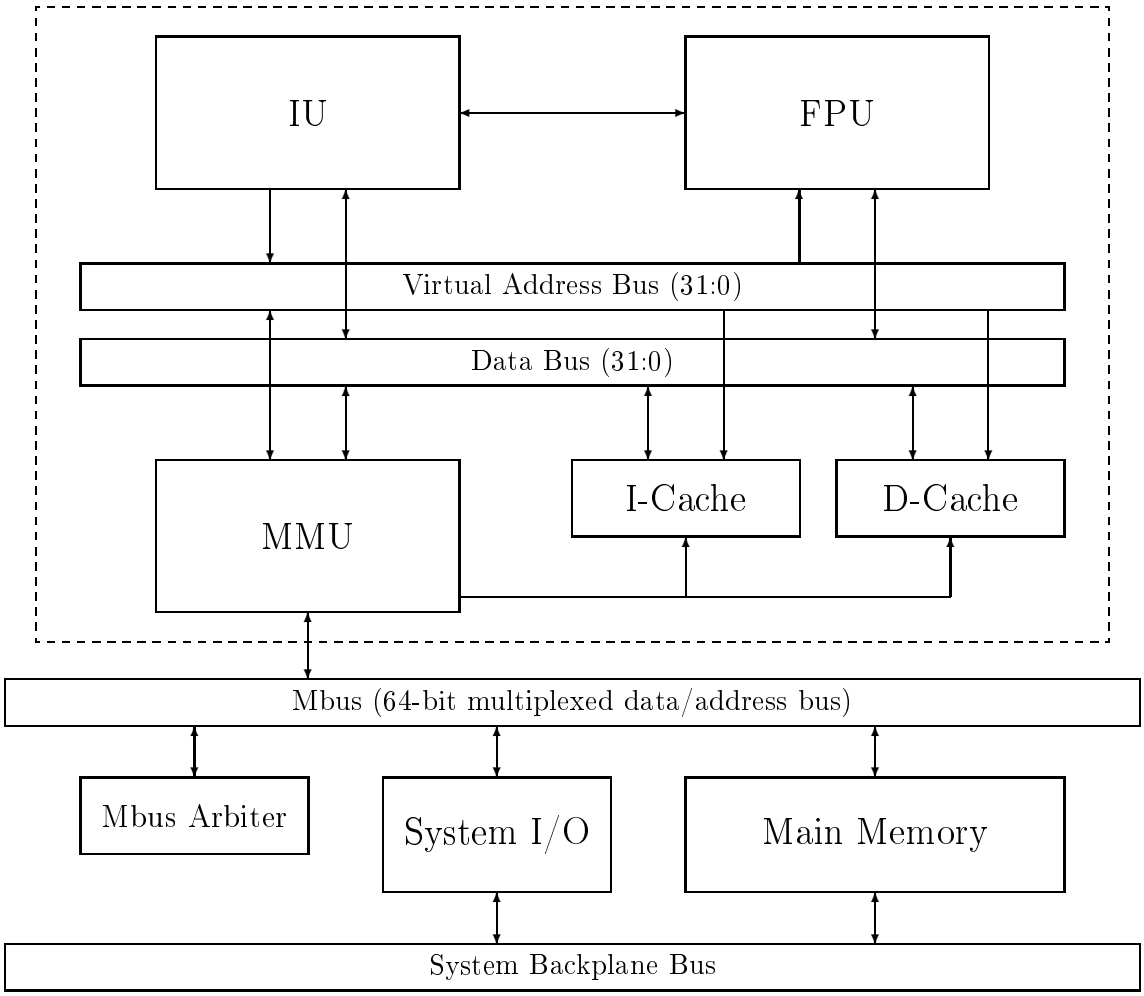

Abbildung 3: Typische Konfiguration einer SPARC v8 Implementation

Die obige Abbildung zeigt die Konfiguration einer typischen SPARC v8 Implementation, basierend auf dem von Sun definierten und geförderten Mbus.

Der strichliert umrandete Teil kann bei Multiprozessorimplementationen mehrfach vorhanden sein. Dieser enthält außer IU und FPU, den internen Bussen, sowie je einem separaten Instruktions- und Daten-Cache (Harvard-Architektur) noch die MMU, die für die Adreßumsetzung, Beschickung und Verwaltung der Caches zuständig ist und der Kommunikation mit dem Mbus dient.

Am Mbus hängen auch der Hauptspeicher und der System-Input/Output. Diese sind ihrerseits über einen Systembus mit der Außenwelt verbunden.

Der Mbus-Arbiter entscheidet über die Erteilung der Zugriffsberechtigung an die mit dem Mbus verbundenen Einheiten.

4.5 Die Ganzzahleinheit IU

Die IU ist die zentrale Steuerungseinheit des SPARC-Systems. Sie enthält die allgemeinen Arbeitsregister, sowie die Status- und Kontrollregister für Ganzzahloperationen, Adreßberechnung, Programmkontrolle, etc. Sie kontrolliert Lade- und Speicheroperationen und Befehlszuführung auch für FPU und CP.

Beim Zugriff der IU auf den Speicher wird an die Adresse ein 8-Bit großer ASI[13] angehängt, in dem codiert ist, ob der Prozessor im Supervisor- oder User-Modus ist und ob es sich um einen Daten- oder Code-Zugriff handelt. Somit können insgesamt 256 (separate oder überlappende) Adreßräume von je 2^{32} Byte bearbeitet werden.

4.6 Die Fließkommaeinheit FPU

Die FPU verfügt über 32 32-Bit Arbeitsregister, die in flexibler Weise zur Speicherung und Verarbeitung der FP-Datentypen dienen.

Fließkomma-Lade- und Speicherbefehle transferieren Werte zwischen FPU und Speicher. Die Adreßberechnung geschieht in der IU. Transfer zwischen IU- und FP-Arbeitsregistern ist nicht vorgesehen. FPop[14]-Befehle bewirken die eigentlichen Fließkommaoperationen.

FPU-Datenformate und Befehlssatz entsprechen dem IEEE[15]-Standard für binäre Fließkommaarithmetik 754-1985.

Falls die FPU nicht vorhanden sein sollte oder falls das EF[16]-Bit im PSR[17] gelöscht sein sollte, wird ein Versuch der Ausführung eines FP-Befehls zur Auslösung einer *fp_ disabled*-Ausnahmebehandlung führen.

4.7 Der Coprozessor CP

Der SPARC-Befehlssatz enthält Unterstützung für einen einzelnen implementationsabhängigen Coprozessor CP. Der CP enthält seinen eigenen spezifischen Registersatz, dessen Konfiguration ebenfalls implementationsabhängig ist, aber im Normalfall auch eine Anzahl von 32-Bit-Registern darstellt. CP-Lade- und Speicherbefehle besorgen den Transfer von Werten zwischen CP-Registern und Speicher. Für jeden Fließkomma-Lade- und Speicherbefehl im Befehlssatz existiert ein entsprechender CP-Befehl.

Falls der CP nicht vorhanden sein sollte oder falls das EC[18]-Bit im PSR gelöscht sein sollte, wird ein Versuch der Ausführung eines CP-Befehls zur Auslösung einer *cp_ disabled*-Ausnahmebehandlung führen.

[13] Address Space Identifier
[14] Floating-Point operate
[15] Institute of Electrical and Electronic Engineers
[16] Enable Floating-Point
[17] Processor State Register
[18] Enable Coprocessor

Da ein CP in den meisten allgemeinen SPARC-Computer-Implementationen nicht vorhanden ist, wird er hier auch nicht weiter behandelt.

4.8 IU-Registersatz

Der IU-Registersatz läßt sich in zwei Gruppen unterteilen, allgemeine oder Arbeitsregister und Status- bzw. Kontrollregister. Um die Notwendigkeit vieler Vorwärtsrefenzen zu vermeiden, wird hier, anders als im Handbuch [SPARC92], die Beschreibung der Status-und Kontrollregister vorgezogen.

4.8.1 IU-Status- und Kontrollregister

Die Status- und Kontrollregister der IU sind:

PSR	Processor State Register
WIM	Window Invalid Mask
TBR	Trap Base Register
Y	Multiply/Divide Register
PC	Program Counter
nPC	next Program Counter

PSR, WIM und TBR sind nur im Supervisor-Modus zugänglich. Optional kann eine Implementation noch folgende Register zur Verfügung stellen:
– Maximal 31 *Ancillary State Registers* (ASR) für besondere Aufgaben, sowie
– eine IU-*Deferred-Trap Queue*. Näheres siehe [SPARC92], 4.2.

Das PSR ist 32 Bit breit und wie folgt belegt:

Abbildung 4: Das Prozessor-Status-Register PSR

	Bedeutung der einzelnen Bit-Gruppen:
impl	*implementation* – fest verdrahtet, zeigt die Implementation an
ver	*version* – implementationsabhängig, zeigt die Version an
icc	*integer condition codes* – enthält die 4 Bedingungsanzeigen
reserved	reserviertes, nicht benutztes Feld von 6 Bit
EC	Enable Coprocessor – 1 = CP aktiv, 0 = CP nicht aktiv oder nicht da
EF	Enable Floating-Point – 1 = FPU aktiv, 0 = FPU nicht aktiv oder nicht da
PIL	Processor Interrupt Level – 16 verschiedene Werte sind möglich
S	Supervisor – 1 = Supervisor-Modus, 0 = User-Modus
PS	Previous Supervisor – Wert des S-Bit bei der letzten *Trap*
ET	Enable Traps – 1 = Traps enabled, 0 = Traps disabled
CWP	Current Window Pointer – Zeiger auf aktuelles Registerfenster

Das PSR wird mit dem Befehl RDPSR gelesen und mit WRPSR geschrieben.

CWP enthält den Zähler, der als Zeiger auf das aktuelle Fenster auf die Registerbank dient. Die Hardware dekrementiert den CWP (modulo NWINDOWS, d.h. der Gesamtzahl der möglichen Fenster) bei SAVE und dem Auftreten von Ausnahmen (*Trap*), sie inkrementiert ihn bei RESTORE und RETT (modulo NWINDOWS). Supervisor-Software kann CWP allerdings auch lesend und schreibend ganz anders manipulieren, z.B. um verschiedene Fenstersätze für unterschiedliche Prozesse zu ermöglichen.

Das *icc*-Feld im PSR umfaßt 4 Bit, die in der Reihenfolge *nzvc* belegt sind. Sie zeigen an, wie das Ergebnis der letzten Operation der 32-Bit-ALU, die dieses Feld beeinflußt hat, bezüglich verschiedener Ereignisse ausgefallen ist:

Bit	Bedeutung		gesetzt	gelöscht
n	*negative*	Ergebnis der Operation ist negativ	negativ	nicht negativ
z	*zero*	Ergebnis der Operation ist Null	Null	nicht Null
v	*overflow*	Ergebnis ist nicht korrekt darstellbar	Überlauf	kein Überlauf
c	*carry*	Übertrag aus oder Borgen ins Bit 31	Übertrag	kein Übertrag

Das WIM-Register ist 32 Bit breit und enthält einen Bitvektor, der für belegte Fenster mit 1, für freie mit 0 belegt ist. Das von CWP=n addressierte Fenster korrespondiert mit dem n-Bit in WIM. Nur NWINDOWS Bit sind beschreibbar, die Bit-Positionen für nicht implementierte Fenster lesen als 0. WIM wird mit RDWIM gelesen und mit WRWIM geschrieben. Durch Beschreiben mit lauter Einsen und anschließendes Lesen, könnte man also herausbekommen, wieviele Fenster tatsächlich implementiert sind: Dazu muß man aber im Supervisor-Modus sein!

Das TBR ist 32 Bit breit und wie folgt belegt:

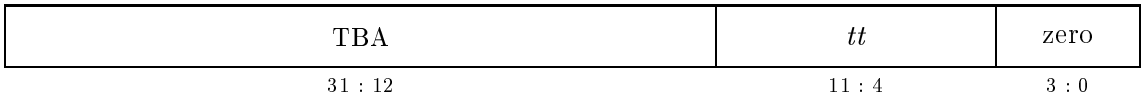

Abbildung 5: Das Trap Base Register TBR

Hier die Bedeutung der einzelnen Bit-Felder:

- TBA — Trap Base Address – enthält die 20 höchstwertigen Bit der Adresse der Trap-Tabelle, es kann mit dem WRTBR-Befehl beschrieben werden.
- *tt* — *trap type* – Dieses 8-Bit-Feld wird von der Hardware beschrieben, wenn eine *Trap* auftritt. Es ist ein Offset in die Tabelle und wird von WRTBR nicht beeinflußt.
- zero — ist immer 0, und sollte von WRTBR als 0 geschrieben werden.

Das 32 Bit breite Multiply/Divide Register Y enthält das höchstwertige Wort des Ergebnisses der 32 Bit $\times$ 32 Bit $\rightarrow$ 64 Bit Multiplikation. Bei der Division, in der

Form 64 Bit $\div$ 32 Bit $\rightarrow$ 32 Bit, enthält es das höchstwertige Wort des Dividenden. Es kann im User-Modus mit RDY gelesen und mit WRY geschrieben werden.

Die beiden Programmzähler PC[19] und nPC[20] sind 32 Bit breit. PC zeigt auf die Adresse des gerade in Ausführung befindlichen Befehls, nPC auf den nächsten abzuarbeitenden Befehl. Bei Abarbeitung eines verzögerten Sprungbefehls schaltet PC auf den unmittelbar folgenden Befehl (*Delay Instruction*), d.h. PC $\leftarrow$ nPC, während nPC, statt zu inkrementieren, auf das Ziel des Sprungbefehls gesetzt wird. Auf diese Weise wird mittels der beiden Programmzähler das Konzept der verzögerten Verzweigung realisiert.

PC wird von CALL und JMPL gelesen, nPC zusätzlich bei Auftreten einer Ausnahme. Die Programmzähler werden wie üblich automatisch von der Hardware beschrieben, und nur indirekt, nämlich durch Sprungbefehle und auftretende Ausnahmen, die ja auch Sprünge im Programmablauf zur Folge haben, beeinflußt.

Für eine ausführlichere Beschreibung konsultiere man [SPARC92], 4.2.

4.8.2 IU-Arbeitsregister

Eine Besonderheit der SPARC, die sie von anderen modernen RISC-Architekturen abhebt, ist die Technik des überlappenden, verschiebbaren Arbeitsregisterfensters.

Eine IU-Implementation kann von 40 bis 520 allgemeine 32-Bit-Register enthalten. Dies entspricht einer Gruppierung in 8 globale, sowie einem zirkulären Stapel von 2 bis 32 Sätzen von je 16 Registern, sog. Fensterregistern. Die Anzahl der tatsächlich vorhandenen Fenster (NWINDOWS) ist implementationsabhängig, und damit auch die Größe der Registerbank innerhalb der oben erwähnten Grenzen.

Die 32 jeweils sichtbaren 32-Bit breiten Arbeitsregister der IU heißen r-Register. 24 davon (*r8 – r31*) stellen einen verschiebbaren Ausschnitt aus der größeren Registerbank dar, während acht (*r0 – r7*) immer die gleichen Register bezeichnen. Entsprechend ihrer Rolle in der SPARC-Programmierung haben sie als solche auch noch andere Namen. Sie sind folgendermaßen organisiert:

r-Registername		*rollenspezifischer* Registername
r24 - r31	$\Longleftrightarrow$	i0 - i7
r16 - r23	$\Longleftrightarrow$	l0 - l7
r8 - r15	$\Longleftrightarrow$	o0 - o7
r0 - r7	$\Longleftrightarrow$	g0 - g7

Da die Breite des verschiebbaren Fensters 24 Register beträgt, die Verschiebung aber jeweils nur um 16 Register erfolgt, überlappen sich jeweils die untersten und obersten Gruppen von 8 Registern mit den entsprechenden Registern im Vorgänger- bzw. Nachfolgerfenster. Dies ist die zentrale Idee, die der Technik des verschiebbaren Registerfensters zugrunde liegt.

[19] Program Counter
[20] next Program Counter

Ein Bild der Wirkung der Technik des überlappend verschiebbaren Registerfensters kann man sich in der folgenden Abbildung machen:

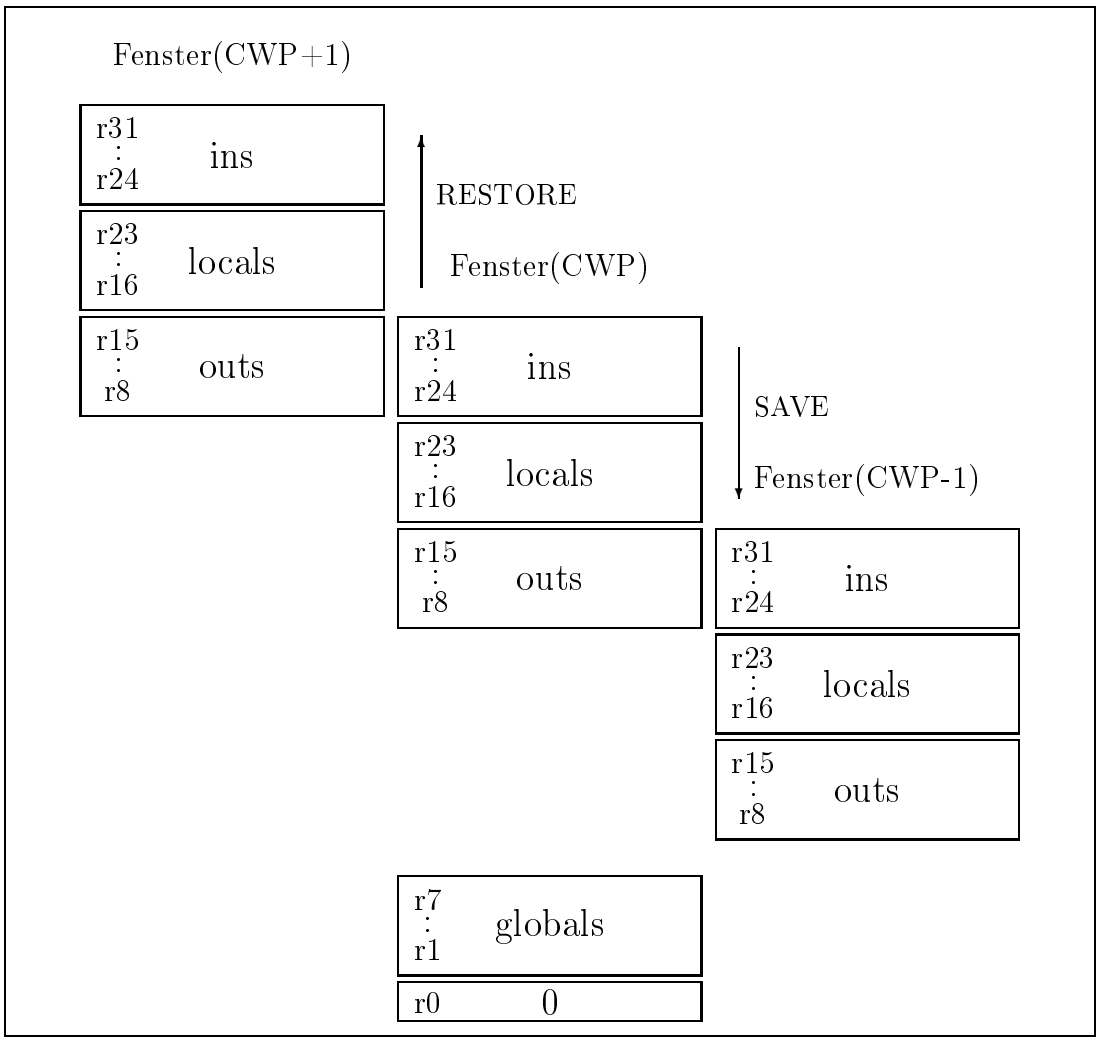

Abbildung 6: Drei überlappende Fenster und die globalen Register

Zu jedem Zeitpunkt kann ein Befehl auf 8 globale, sowie ein Fenster von 24 r-Registern, organisiert als 8 *in* (entspr. den 8 *out* des vorigen Fensters), 8 *local* und 8 *out* (entspr. den 8 *in* des nächsten Fensters), aus dem zirkulären Stapel der Registerbank zugreifen. Das aktuelle Fenster wird durch den CWP bezeichnet. Fensterüber- oder -unterlauf werden durch Vergleich mit der Belegung des WIM detektiert und lösen eine Ausnahmebehandlung aus, in der die Systemsoftware durch Retten bzw. Restaurieren von Fensterinhalten wieder Platz schafft. Für Anwendungsprogramme ist dieses Verhalten transparent und die Anzahl der tatsächlich vorhandenen Fenster ist für User-Modus-Software unsichtbar.

Die Überlappung der Registerfenster bietet einen sehr effizienten Weg der Parameterübergabe bei Prozeduraufruf und -rückkehr. Die rufende Prozedur legt die

zu übergebenden Parameter in ihre *out*-Register und führt dann einen CALL aus. Dieser Befehl schreibt – durch die Hardware bedingt – seine eigene Adresse in das Register *r15* (*o7*). Die gerufene Prozedur führt zuerst einen SAVE aus – damit wird das Fenster weitergeschaltet – und die übergebenen Parameter liegen nun in den *in*-Registern bereit, außerdem hat sie selbst einen frischen Satz *local*- und *out*-Register zur Verfügung (und auf Wunsch auch noch einen frischen *stack frame*). Vor der Rückkehr plaziert sie etwaige Rückgabewerte in ihren *in*-Registern (die ja identisch sind mit den *out*-Registern des Rufers) und führt nach einem *Return* mit JMPL auf die Adresse *i7*+8 noch ein RESTORE aus, wobei das Fenster (und gleichzeitig der *stack frame*) wieder zurückgeschaltet wird. Jetzt ist die rufende Prozedur wieder aktiviert und kann etwaige Rückgaben in den vereinbarten *out*-Registern entgegennehmen.

In eingebetteten Anwendungen, wo u.U. schnelle *Task*-Umschaltung wichtiger ist als Prozeduraufrufe, kann die Fenstertechnik des SPARC-Prozessors auch noch ganz anders genutzt werden. Es kann z.B. die Registerbank in Gruppen von je zwei Fenstern aufgeteilt werden, in denen dann unter Benutzung der *local*-Register jeweils ein separater Prozeß läuft. Die Interprozeßkommunikation könnte über die jeweils überlappenden *in*- und *out*-Gruppen realisiert werden.

Die acht globalen Register, die von der Fenstertechnik nicht berührt werden und immer für alle sichtbar sind, können natürlich auch zur Parameterübergabe genutzt werden, oder als globale Variablen für ganze Prozedurgruppen, z.B. in Bibliotheken – wenn das so vereinbart ist.

Das Register *r0* bzw. *g0* liest immer als Null, es zu beschreiben hat keine Wirkung. So ist zwar die Anzahl der zur Datenspeicherung zur Verfügung stehenden globalen Register um eins verringert, aber man hat eine sehr häufig gebrauchte Konstante immer gleich zur Hand. Diese Technik ist übrigens im Reich der RISC-Prozessoren durchaus üblich.

4.9 FPU-Registersatz

Die Arbeitsregister der FPU heißen *f*-Register. Der Registersatz besteht aus 32 32-Bit-breiten *f*-Registern. 64-Bit-Werte belegen Paare von geraden und ungeraden Registern, 128-Bit-Werte belegen an Vierer-Grenzen ausgerichtete Registerquadrupel. So können die FPU-Register wahlweise 32 Werte in einfacher, 16 Werte in doppelter, 8 Werte in erweiterter Genauigkeit oder eine zuträgliche Mischung davon enthalten.

Die Status- und Kontrollregister der FPU sind:

- FSR Floating-Point State Register
- FQ optionale implementationsabhängige FP Deferred-Trap Queue

Die Bit-Felder im FSR enthalten Information zu Modus und Status der FPU. Das FSR wird mit den User-Modus-Befehlen STFSR und LDFSR gelesen und geschrieben. (Also Transfer zwischen Speicher und FSR!)

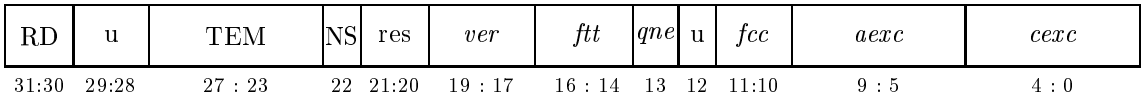

Abbildung 7: Das Floating-Point State Register FSR

Die Bedeutung der einzelnen Bit-Felder:

RD round – diese beiden Bit bestimmen den Rundungsmodus nach IEEE 754-1985:

RD	Rundungsmodus
0	zum nächsten, bei gleichem Abstand zum geraden Wert
1	nach 0, d.h. abschneiden (*truncate*)
2	nach $+\infty$
3	nach $-\infty$

u unused – Bit 29, 28 und 12 sind unbenutzt und sollten aus Kompatibilitätsgründen mit LDFSR nur als Nullen geschrieben werden.

TEM Trap Enable Mask – Bit 27 ... 23 aktivieren je eine der fünf FP-Ausnahmen, die im *cexc*-Feld eingetragen werden können. Eine 1 an der entsprechenden Bit-Position läßt die Auslösung einer Ausnahme zu, eine 0 verhindert sie.

NS Non Standard – wenn dieses Bit gesetzt ist, kann die FPU implementationsabhängige Resultate, abweichend vom IEEE-Standard produzieren, z.B. um höhere Leistung zu erreichen. Näheres dazu im Anhang L von [SPARC92].

res reserved – diese Bit lesen als Nullen und sollten auch so geschrieben werden.

ver *version* – enthält die Versionsnummer der FPU.

ftt *floating-point trap type* – nach Auftreten einer FP-Ausnahme kennzeichnen diese drei Bit den Typ der Ausnahme, solange bis ein STFSR-Befehl oder eine weitere FPop ausgeführt ist. Das *ftt*-Feld kann mittels STFSR gelesen werden. Ein LDFSR-Befehl verändert es nicht. Supervisor-Software für die FP-Ausnahmebehandlung muß STFSR ausführen, um das *ftt*-Feld lesen zu können. Ob STFSR *ftt* ausnullt, ist implementationsabhängig. Wenn STFSR *ftt* nicht ausnullt, muß die System-Software garantieren, daß der nächste STFSR-Befehl *ftt* als Null anzeigt. Die folgenden acht Werte sind möglich für *ftt*:

ftt	Trap Type
0	keine Ausnahme
1	IEEE 754 Ausnahme
2	unvollendete FPop
3	nicht implementierte FPop
4	Sequenz-Fehler
5	Hardware-Fehler
6	ungültiges FP-Register
7	reserviert

qne *queue not empty* – zeigt nach einer *deferred_ fp_ exception trap* oder nach einem STDFQ-Befehl den Zustand der optionalen *floating-point deferred-trap queue* FQ an: qne = 0, FQ ist leer, qne = 1, FQ ist nicht leer.

fcc *floating-point condition codes* – Diese 2 Bit sind Bedingungsanzeigen für Fließkommaverzweigungsoperationen. Sie werden von den FP-Befehlen FCMP und FCMPE beeinflußt. Lesen und Schreiben kann man sie mit den Befehlen STFSR und LDFSR. Sie steuern die Verzweigungen der FBfcc-Befehle. In der folgenden Tabelle bezeichnen f_{rs1} und f_{rs2} die *single*, *double* oder *quad* Werte in den f-Registern, die durch die *rs1* und *rs2*-Felder im Befehlsformat belegt werden. Das Fragezeichen (?) bezeichnet eine ungeordnete Relation. Diese ist wahr, wenn f_{rs1} oder f_{rs2} eine NaN darstellen. Man beachte, daß *fcc* unverändert bleibt, falls FCMP oder FCMPE eine *IEEE_ 754_ exception* auslösen.

fcc	Relation	FBfcc
0	$f_{rs1} = f_{rs2}$	E
1	$f_{rs1} < f_{rs2}$	L
2	$f_{rs1} > f_{rs2}$	G
3	f_{rs1} ? f_{rs2} (ungeordnet)	U

aexc *accrued exception* – Dieses Feld sammelt IEEE_754 *floating-point exceptions*, während der Zeit, in der die FP-Ausnahmen durch das TEM-Feld abgeschaltet sind. Nach Beendigung der Ausführung eines FPop-Befehls werden die TEM- und *cexc*-Felder bit-logisch verundet. Falls das Ergebnis ungleich Null ist, wird eine *fp_ exception* ausgelöst. Andernfalls wird das neue *cexc*-Feld in das *aexc*-Feld geodert. Während der Zeit ihrer Maskierung werden die so aufgelaufenen Ausnahmen im *aexc*-Feld akkumuliert.

cexc *current exception* – Dieses Feld zeigt an, ob eine oder mehrere IEEE_754 *floating-point exceptions* im jüngst abgearbeiteten FPop-Befehl aufgetreten sind. Bei Abwesenheit einer Ausnahme wird das entsprechende Bit gelöscht. Es wird empfohlen, daß eine IEEE_754 *floating-point exception* genau ein Bit im *cexc*-Feld setzt. Weitere Details findet man in [SPARC92], S. 38 ff.

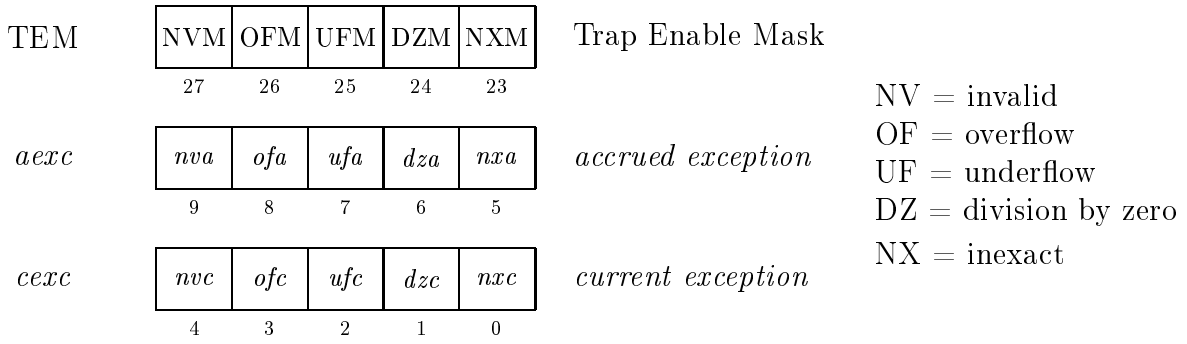

Abbildung 8: Der Ausnahmemechanismus im FSR

Die Abbildung zeigt den Zusammenhang zwischen den Feldern TEM, *aexc* und *cexc*, die Belegung der einzelnen Positionen für die fünf Arten von IEEE-Ausnahmen, sowie die Bedeutung der Kürzel entsprechend IEEE 754-1985.

Ein gesetztes Bit in *aexc* oder *cexc* bedeutet, daß ein entsprechender Fall eingetreten ist, ein gelöschtes Bit, daß dies nicht der Fall war.

nvc, nva — *invalid operand* – ein Operand ist ungültig für die in Frage kommende Operation (z.B. $0 \div 0$ oder $\infty - \infty$).

ofc, ofa — *overflow* – das gerundete Resultat wäre größer als die größte im spezifizierten Format darstellbare normalisierte Zahl.

ufc, ufa — *underflow* – das gerundete Resultat ist ungenau und wäre betragsmäßig kleiner als die kleinste im spezifizierten Format darstellbare normalisierte Zahl.
Dies Bit wird nicht gesetzt, wenn das korrekte, ungerundete Ergebnis Null ist. Ansonsten,
falls UFM=0: Beide Bit werden gesetzt, wenn das korrekte, ungerundete Ergebnis einer Operation betragsmäßig kleiner als die kleinste normalisierte Zahl ist und das korrekt gerundete Ergebnis ungenau ist. Sie werden ebenfalls gesetzt, wenn das korrekte, ungerundete Ergebnis kleiner als die kleinste normalisierte Zahl ist, aber das korrekt gerundete Ergebnis die kleinste normalisierte Zahl ist. Auch *nxc* und *nxa* werden in diesem Fall immer gesetzt.
falls UFM=1: Eine *IEEE_754_exception* wird ausgelöst, wenn das korrekte, ungerundete Ergebnis einer Operation kleiner als die kleinste normalisierte Zahl sein würde. Eine Ausnahme wird ebenfalls ausgelöst, wenn das korrekte, ungerundete Ergebnis kleiner als die kleinste normalisierte Zahl ist, aber das korrekt gerundete Ergebnis die kleinste normalisierte Zahl sein würde.

dzc, dza — *division by zero* – eine Operation $X \div 0$, wobei X eine subnormale oder normalisierte Zahl ist. Man beachte, daß $0 \div 0$ das *dzc*-Bit **nicht** setzt.

nxc, nxa — *inexact result* – das gerundete Ergebnis einer Operation unterscheidet sich vom unendlich genauen ungerundeten Ergebnis.

In [SPARC92] werden einige Vorschläge zu möglichen Implementationen des FSR gemacht. Wie auch immer die Hardware einer SPARC-FPU realisiert wird, die Software muß – und sei es durch Simulation der gesamten FPU – in der Lage sein, die folgenden Ausnahmen in Übereinstimmeung mit dem Standard IEEE 754-1985 korrekt abzuhandeln: *unimplemented_FPop*, *unfinished_FPop* und *IEEE_754_exception*. Auf diese Weise »sieht« ein Anwendungsprogramm immer ein FSR, das den Anforderungen des Standards voll entspricht.

FQ kann – falls vorhanden – im Supervisor-Modus mit dem Befehl STDFQ gelesen werden. Evtl. können auch privilegierte Befehle LDDA/STDA (load/store double alternate) oder RDASR/WRASR (read/write ancillary state register) zu diesem

Zweck implementiert werden.

Falls in einer gegebenen Implementation FP-Befehle nebenläufig (und asynchron) zu Ganzzahlbefehlen ablaufen sollen, *muß* eine FQ vorgesehen werden. Wenn FP-Befehle synchron mit Ganzzahlbefehlen ausgeführt werden, ist die Einrichtung einer FQ optional.

Der Inhalt der FQ und die darauf möglichen Operationen sind implementationsabhängig. Jedoch muß es für Supervisor-Software möglich sein, über den FP-Befehl, der die Ausnahme ausgelöst hat, folgendes herauszufinden:

– seinen Befehlscode (das *opf*-Feld),
– seine Operanden,
– seine Speicheradresse.

Das gleiche gilt für etwaige weitere Einträge in dieser Warteschlange.

In einer Implementation ohne FQ ist das *qne*-Bit immer 0, ein STDFQ-Befehl würde damit eine *sequence_ error*-Ausnahme auslösen.

4.10 Befehlssatz

4.10.1 Befehlsformate

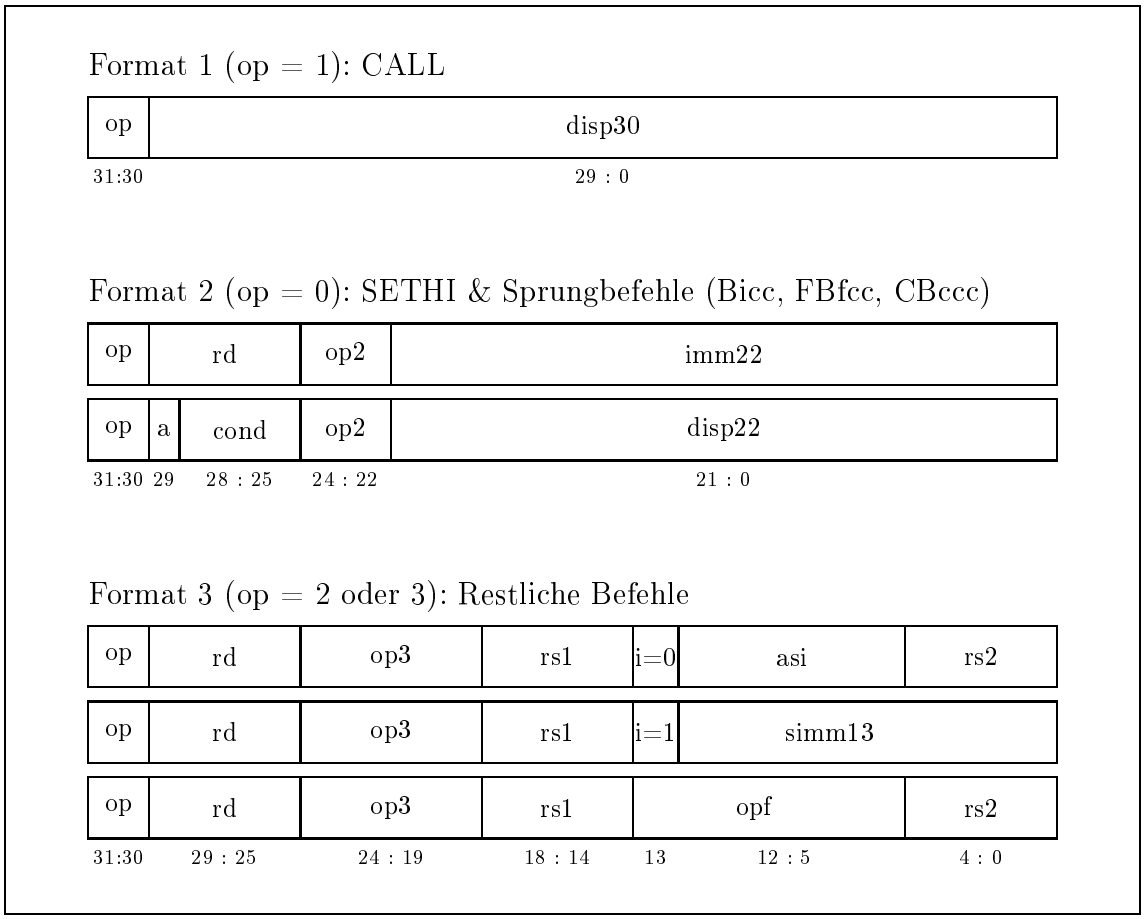

Abbildung 9: Die Befehlsformate des SPARC-Prozessors

Format	op	Befehle
1	1	CALL
2	0	SETHI, Bicc, FBfcc, CBccc
3	3	Lade- und Speicherbefehle
3	2	sonstige Befehle

op2	Befehle
0	UNIMP
1	nicht implementiert
2	Bicc
3	nicht implementiert
4	SETHI
5	nicht implementiert
6	FBfcc
7	CBccc

Abbildung 10: op- und op2-Codierung

SPARC-Befehle sind in drei verschiedenen 32-Bit-Formaten codiert. Das 2-Bit *op*-Feld codiert die drei Hauptformate, und zusammen mit *op2* die Befehlsgruppe von Format 2. Die weiteren Felder in den Befehlsformaten werden folgendermaßen interpretiert:

rd	Dieses 5-Bit-Feld bezeichnet Ziel (oder Quelle) für *r*- oder *f*- oder CP-Register eines arithmetischen, Lade- (oder Speicher-)Befehls. Für Double- oder Quad-Werte wird das letzte (die beiden letzten) Bit(s) nicht benutzt und sollte(n) von der Software als Nullen bereitgestellt werden.
a	Dieses Bit in einem Verzweigungsbefehl annulliert die Ausführung des folgenden Befehls (im sog. *delay slot*), und zwar bei einer bedingten Verzweigung, wenn sie nicht genommen wird, sowie bei einer unbedingten, wenn sie genommen wird.
cond	Mit dieser 4-Bit-Gruppe wird die Art der Bedingung ausgewählt, d.h. die zu testenden Bit im *icc*-Feld des PSR oder im *fcc*-Feld des FSR.
imm22	Diese 22-Bit-Konstante wird in den oberen 22 Bit des Zielregisters gepeichert.
disp22	Dies sind die auf 32 Bit vorzeichenrichtig erweiterten, um 2 Bit nach links
disp30	geschobenen PC-relativen Sprungdistanzen für CALL und alle Verzweigungsbefehle.
op3	Diese 6 Bit (mit 1 Bit aus *op*) codieren die Befehle aus Format 3. Details dazu in Anhang F von [SPARC92]
i	Dies Bit selektiert den zweiten Operanden für das IU-Rechenwerk. Bei i=0 ist der Operand *rs2*, bei i=1 ist der Operand *simm13*, vorzeichenrichtig erweitert auf 32 Bit.
asi	Dies 8-Bit Feld enthält den ASI für einen *load/store alternate*- Befehl.
rs1	Diese beiden 5-Bit-Felder bezeichnen jeweils den ersten oder zweiten Operanden aus den *r*- oder *f*- oder CP-Registern Für Double- oder Quad-Werte wird das letzte (die beiden letzten) Bit nicht benutzt und sollte(n) von der Software als Nullen bereitgestellt werden.
rs2	
simm13	Zweiter Operand, wird vorzeichenrichtig auf 32 Bit erweitert, siehe auch oben unter *i* für i=1.
opf	Diese 9 Bit codieren die FP- (FPop) oder CP-Befehle (CPop). Näheres zur Codierung in Anhang F von [SPARC92]

4.10.2 Befehlsausführung

Ein Befehl wird von der Adresse gelesen, auf die der Programmzähler PC zeigt. Er wird dann ausgeführt – oder auch nicht, wenn der vorausgehende Befehl ein annullierender Sprungbefehl war (siehe dort). Ein Befehl kann aber auch zu einer Ausnahmebehandlung (*Trap*) führen, bedingt durch das Auftreten einer Ausnahmesituation im Befehl selbst (*direct trap*), in einem vorausgehenden Befehl (*deferred trap*), durch eine Unterbrechungsanforderung von außen (*interrupting trap*), oder durch ein ebenfalls äußeres *Reset*-Signal.

Wird ein Befehl ausgeführt, kann er den für das Programm sichtbaren Status des Prozessors oder des Speichers ändern.

Läuft der Befehl normal ab, wird der nPC in den PC kopiert und der nPC wird um 4 inkrementiert. War der Befehl ein Sprungbefehl, wird der nPC stattdessen auf das Sprungziel gesetzt. So wird mittels der beiden Programmzähler PC und nPC das Modell einer verzögerten Programmverzweigung verwirklicht.

Anmerkung: Die Architekturbeschreibung macht bewußt keine Angaben zu Befehlsausführungszeiten, zur Ausführungsparallelität oder zur Hardware, die Befehle decodiert und ausführt. Dies alles, und auch das genaue Verhältnis von PC und nPC ist implementationsabhängig. Eine Implementation sollte sich jedoch so verhalten, daß in der Abwesenheit von *Traps* das von der Architektur vorgeschriebene serielle Ausführungsmodell eingehalten wird. Dem Verhalten bei Anwesenheit von *Traps* ist ein gesondertes Kapitel gewidmet.

In Handbüchern tatsächlicher Implementationen, z.B. [Cypress90], findet man aber durchaus Angaben zu Taktzeiten, zur Verarbeitungs-*Pipeline*, sowie Hinweise, daß man das Ergebnis einer Ladeoperation möglichst erst nach Dazwischenschieben einer weiteren Operation verwenden solle, um die *Pipeline* nicht ins Stocken geraten zu lassen. Gleiches gilt für die Verwendung des Ergebnisses eines direkt vorhergehenden Befehls. Dies entspricht dem üblichen Verhalten dieser Architekturen. Einige Hinweise zum Verhalten verschiedener Hardware-Implementationen finden sich im Anhang L von [SPARC92].

4.10.3 Befehlskategorien

SPARC-Befehle lassen sich in sechs Kategorien einteilen:

– Lade- und Speicherbefehle (*load/store*)
– Ganzzahlarithmetik (mit Logik- und Schiebebefehlen)
– Verzweigungsbefehle (*control transfer instructions* CTI)
– Lesen und Schreiben von Steuerregistern
– Fließkommaoperationen (FPop)
– Coprozessoroperationen (CPop)

Lade- und Speicherbefehle: Wie auch in anderen RISC-Architekturen üblich, können *nur* die Lade- und Speicherbefehle lesend und schreibend auf den Speicher zugreifen. Die Beschränkungen beim Speicherzugriff auf Code und Daten wurden schon im Abschnitt über Adressierung und Ausrichtung erläutert.

Die Speicheradressierungsarten sind, im Gegensatz zu CISC-Architekturen, und zwar unter Berücksichtigung der Tatsache, daß alle Möglichkeiten im 32-Bit-Format unterzubringen sind, sehr einfach:

- Zwei Register – Der Inhalt der beiden durch *rs1* und *rs2* bezeichneten Register wird addiert und bildet die Adresse für den Zugriff.

- Register + 13-Bit Direktwert – Der Inhalt des durch *rs1* bezeichneten Registers und der Wert des vorzeichenrichtig auf 32 Bit erweiterten *simm13* Feldes werden addiert und bilden die Zugriffsadresse.
- 13-Bit Direktwert – Sonderfall der vorigen Adressierungsart, wenn *rs1* das Register *r0* bezeichnet, dessen Wert ja immer Null ist. Auf diese Weise können die untersten, bzw. obersten 4 KByte des Adreßraums direkt adressiert werden.

Für jeden Speicherzugriff auf Daten oder Code hängt die IU automatisch einen 8-Bit großen ASI an den Befehl, der einen von 256 möglichen Adreßräumen codiert, und per Architekturfestlegung auch den aktuellen Prozessormodus bezeichnet. Vier dieser ASI-Werte sind festgelegt, der Rest ist implementationsabhängig:

ASI	Adreßraum
0x00-0x07	implementationsabhängig
0x08	User-Befehle
0x09	Supervisor-Befehle
0x0A	User-Daten
0x0B	Supervisor-Daten
0x0C-0xFF	implementationsabhängig

Es gibt auch spezielle privilegierte Befehle (*load/store alternate*), mit denen ein beliebiger ASI angegeben werden kann, der dann den automatisch vergebenen ersetzt. Einzelheiten und Implementationsvorschläge findet man im Anhang I von [SPARC92].

Ganzzahlarithmetik, logische und Schiebebefehle: Addition, Subtraktion, bitmäßige Verknüpfung: Diese Sorte von Befehlen hat meist eine triadische Registeradressierung, wobei aus zwei Quellregistern als Operanden ein Ergebnis gebildet und im durch den dritten Operanden bezeichneten Zielregister eingeschrieben wird. Durch diese zerstörungsfreie Form, im Gegensatz zur Zweioperandentechnik, sind weniger Umspeicher- bzw. Rettungsoperationen im Programm erforderlich. Der zweite Operand kann aber auch ein 13-Bit-Direktwert sein, abgelegt im *simm13*-Feld, der dann auf 32 Bit vorzeichenrichtig erweitert zur Berechnung herangezogen wird.

Register *r0* bzw. *g0* als Quelle liest immer als Null; dient es als Ziel, wird das Ergebnis der Bearbeitung verworfen und nur die – in diesem Fall gewünschten – Seiteneffekte bleiben wirksam (z.B. die Beeinflussung der Bedingungsanzeigen).

Die meisten dieser Befehle existieren in zwei Versionen, eine setzt die *integer condition codes – icc*, die andere läßt sie unbeeinflußt. Spezielle Vergleichsbefehle sind überflüssig, da man sie leicht aus Subtraktion mit Setzen von *icc* und dem Zielregister *g0* konstruieren kann.

Schiebebefehle schieben das Bitmuster eines Registerquelloperanden um einen Wert in einem zweiten Register oder eine 13-Bit-Konstante nach links oder rechts – auch vorzeichenrichtig – und legen das Ergebnis im Zielregister ab. Schiebebefehle beeinflussen niemals die *icc*.

SETHI lädt eine 22-Bit Konstante in die oberen 22 Bit des Zielregisters und löscht dessen untere 10 Bit. *icc* wird nicht beeinflußt. Es dient insbesondere dazu, in einem Register mit Hilfe eines weiteren Befehls, z.B. OR, eine 32-Bit Konstante zu konstruieren. Der Befehl ist insofern eine Ausnahme, als er nur einen Registeroperanden hat.

Ganzzahlmultiplikation und -division sind – unter Einbeziehung des Spezialregisters Y – in den Versionen 32-Bit $\times$ 32-Bit $\rightarrow$ 64-Bit und 64-Bit $\div$ 32-Bit $\rightarrow$ 32-Bit vorhanden. Es gibt Versionen, die *icc* beeinflussen und solche, die es nicht tun. Division durch Null löst eine *division_by_zero*-Ausnahme aus.

Außerdem gibt es noch einen *multiply step*-Befehl, der zur schrittweisen Konstruktion eines Multiplikationsergebnisses per Software dient (aus v7, die anfangs noch keine vollen Divisions- und Multiplikationsbefehle hatte) und jetzt als veraltet (*deprecated*) betrachtet werden muß.

Die *Tagged Add/Subtract*-Befehle wurden dazu vorgesehen, gewisse Techniken beim Schreiben von z.B. LISP-Übersetzern zu erleichtern. Näheres dazu findet man im Handbuch ([SPARC92]).

Verzweigungsbefehle: Ein Verzweigungsbefehl (*Control Transfer Instruction, CTI*) verändert den Wert des nPC. Es gibt fünf grundlegende CTI-Typen:

- Bedingte Verzweigung, *Conditional Branch*, Bicc, FBfcc, CBccc
- Prozeduraufruf mit Rückkehrverbindung, *Call and Link*, CALL
- Sprung mit Rückkehrverbindung, *Jump and Link*, JMPL
- Rückkehr aus der Ausnahmebehandlung, *Return from Trap*, RETT
- Ausnahme (*Trap*), *Trap on integer condition codes*, Ticc

CTI's können nach Art der Zieladressenberechnung (PC-relativ oder registerindirekt) und nach der Zeit, wann der Transfer stattfindet (verzögert, nicht verzögert oder bedingt verzögert) kategorisiert werden:

Verzweigungsbefehl	Zieladreßberechnung	relative Transferzeit
Bicc, FBfcc, CBccc	PC-relativ	bedingt verzögert
CALL	PC-relativ	verzögert
JMPL, RETT	register-indirekt	verzögert
Ticc	register-indirekt über Tabelle	nicht verzögert

SPARC unterscheidet sechs verschiedene Arten von CTI:

PC-relativ
: Die Zieladresse wird berechnet durch vorzeichenrichtige Erweiterung des Direktwert-Operanden (30 Bit bei CALL, sonst 22 Bit) und anschließendes Linksschieben um 2 Bit. Der so erhaltene Wert wird nun zum Inhalt des PC addiert und in den nPC geschrieben. Auf diese Weise kann bei CALL der gesamte Adreßraum, sonst – bei Bicc, FBfcc und CBccc – eine Distanz von ±8 MB überstrichen werden.

registerindirekt
: Die Zieladresse wird berechnet als die Summe der Inhalte der beiden durch *rs1* und *rs2* bezeichneten Register oder die Summe des Inhalts des durch *rs1* bezeichneten Registers und des vorzeichenrichtig auf 32 Bit erweiterten *simm13*-Feldes. Das Ergebnis wird in den nPC geschrieben. Der gesamte Adreßraum kann überstrichen werden. Hierzu gehören die Befehle JMPL und RETT.

registerindirekt über Tabelle
: Die Zieladreßberechnung geschieht wie folgt: Ein *trap type* wird berechnet aus der Summe von 128 und den Inhalten der beiden durch *rs1* und *rs2* bezeichneten Register oder der Summe von 128, des Inhalts des durch *rs1* bezeichneten Registers und des vorzeichenrichtig auf 32 Bit erweiterten *simm13*-Feldes. Das Ergebnis wird in das *tt*-Feld des TBR geschrieben, dessen gesamter resultierender Inhalt dann die Zieladresse bildet. Zu dieser Gruppe gehören ausschließlich die Ticc-Befehle.

verzögert
: Eine verzögerte Verzweigung verzweigt zur Zieladresse nach einer Verzögerung von *einem* Befehl. Dieser direkt folgende Befehl (sog. *delay instruction*, die Position wird *delay slot* genannt) wird *nach* dem Verzweigungsbefehl ausgeführt, aber *bevor* die Kontrolle an das Ziel der Verzweigung übergeht.

nicht verzögert
: Bei einer nicht verzögerten Verzweigung geht die Kontrolle unmittelbar nach Ausführung des Verzweigungsbefehls an das Ziel der Verzweigung über. Zu dieser Gruppe gehören ausschließlich die Ticc-Befehle.

bedingt verzögert
: Ein bedingt verzögerter Verzweigungsbefehl bewirkt entweder eine verzögerte oder eine nicht verzögerte Verzweigung, abhängig vom Wert seines *Annul Bit*, und davon, ob die Verzweigung selbst bedingt war oder nicht. Hierunter fallen alle Befehle aus den Gruppen Bicc, FBfcc und CBccc.

Der CALL-Befehl trägt seine Adresse in das Register *r15* (*o7*) ein. Sie kann dann – um acht inkrementiert – als Rücksprungziel dienen.

Der JMPL-Befehl trägt seine Adresse in das durch *rd* bezeichnete Register ein. Wenn man auch hier *r15* als Zielregister angibt, kann man JMPL als registerindirek-

ten Unterprogrammaufruf benutzen. Bei *rd* = *r0* dient dieser Befehl als Rücksprung aus einer Subroutine.

Über Einzelheiten der Wirkungsweise dieser Befehle informiere man sich im Handbuch [SPARC92].

Tabelle der Bicc-Verzweigungsbefehle

Opcode	*cond*	Operation	*icc*-Bit-Test
BA	1000	Branch Always	1
BN	0000	Branch Never	0
BNE	1001	Branch on Not Equal	not Z
BE	0001	Branch on Equal	Z
BG	1010	Branch on Greater	not(Z or(N xor V))
BLE	0010	Branch on Less or Equal	Z or(N xor V)
BGE	1011	Branch on Greater or Equal	not(N xor V)
BL	0011	Branch on Less	N xor V
BGU	1100	Branch on Greater Unsigned	not(C or Z)
BLEU	0100	Branch on Less or Equal Unsigned	C or Z
BCC	1101	Branch on Carry Clear (GE Unsigned)	not C
BCS	0101	Branch on Carry Set (Less Unsigned)	C
BPOS	1110	Branch on Positive	not N
BNEG	0110	Branch on Negative	N
BVC	1111	Branch on Overflow Clear	not V
BVS	0111	Branch on Overflow Set	V

Tabelle der FBfcc-Verzweigungsbefehle

Opcode	*cond*	Operation	*fcc*-Bit-Test
FBA	1000	Branch Always	1
FBN	0000	Branch Never	0
FBU	0111	Branch on Unordered	U
FBG	0110	Branch on Greater	G
FBUG	0101	Branch on Unordered or Greater	G or U
FBL	0100	Branch on Less	L
FBUL	0011	Branch on Unordered or Less	L or U
FBLG	0010	Branch on Less or Greater	L or G
FBNE	0001	Branch on Not Equal	L or G or U
FBE	1001	Branch on Equal	E
FBUE	1010	Branch on Unordered or Equal	E or U
FBGE	1011	Branch on Greater or Equal	E or G
FBUGE	1100	Branch on Unordered or Greater or Equal	E or G or U
FBLE	1101	Branch on Less or Equal	E or L
FBULE	1110	Branch on Unordered or Less or Equal	E or L or U
FBO	1111	Branch on Ordered	E or L or G

Auf die Tabelle der möglichen CBccc-Befehle verzichten wir hier, Interessierte finden sie im Handbuch [SPARC92] auf Seite 123.

Hier folgt nun eine Übersicht über die Eigenheiten der bedingten Verzweigungsbefehle der Gruppen Bicc, FBfcc und CBccc.

Da im *cond*-Feld des Befehlsformats 2 vier Bit vorgesehen sind, umfassen diese Verzweigungen jeweils Gruppen von 16 Befehlen, von denen 14 auf die Bedingungs-Codes im entspr. Statusregister reagieren.

Eine Besonderheit der SPARC gegenüber anderen RISC-Architekturen mit verzögerten Verzweigungsbefehlen ist die Möglichkeit, die Ausführung des im sog. *delay slot* befindlichen Befehls bei den bedingt verzögerten Gruppen (Bicc, FBfcc und CBccc) mittels des *Annul Bit* zu annullieren, d.h. unwirksam zu machen. Diese Technik bietet interessante Möglichkeiten für Supervisor-Software, sowie weit bessere Möglichkeiten für den Compiler, hier sinvolle Befehle einzusetzen und damit schlankeren und schnelleren Code zu erzeugen. (Anmerkung: Das *Annul Bit* wird gesetzt, indem an das Assembler-Befehlswort ein `,a` angehängt wird, also z.B. `bge,a label`).

Das Verhalten bei gesetztem *Annul Bit* ist unterschiedlich:

- Bei den unbedingten Verzweigungsbefehlen BA und BN (bzw. FBA, FBN etc.), *Branch Always, Branch Never* wird der Folgebefehl *immer* annulliert.
- Bei den eigentlichen bedingten Verzweigungsbefehlen, wird der Folgebefehl nur dann annulliert, wenn die Verzweigung *nicht* genommen wird

Verzögerte Verzweigungsbefehle – bedingt oder unbedingt – werden als DCTI (*Delayed Control Transfer Instruction*) bezeichnet. Ein Problem kann sich nun ergeben, wenn diese Verzweigungen in Paaren auftreten, sog. DCTI-Paaren, d.h. im *delay slot* eines DCTI steht wieder ein DCTI. Solche Konstellationen sind möglich, und die Diskussion dazu ist ziemlich langwierig ([SPARC92] Seite 55 ff.). Hier sei nur soviel gesagt, daß ein Bicc-Befehl (außer BA) niemals von einem DCTI gefolgt werden darf, weil dann die Fortführadresse zwar im gleichen Adreßraum läge, aber sonst unbestimmt wäre. Für die Einhaltung dieser Einschränkung müssen also Compiler oder Programmierer sorgen!

Zwar handelt es sich nicht um Verzweigungsbefehle, aber da sie meist bei Subroutinenaufrufen gebraucht werden, ist eine Besprechung der Befehle SAVE und RESTORE an dieser Stelle angebracht.

SAVE entspricht einem Additionsbefehl, bis auf die Tatsache, daß der CWP um eins dekrementiert wird. Dabei wird das Fenster CWP-1 zum aktuellen Fenster, und das Fenster des Aufrufers wird »gerettet«. Außerdem stammen die Quellregister aus dem (alten) CWP-Fenster, das Zielregister hingegen liegt im (neuen) Fenster CWP-1. Der SAVE-Befehl wird benutzt, um *atomar* das Registerfenster weiterzuschalten *und* der Subroutine einen neuen *Stack Frame* zu verschaffen.

RESTORE entspricht ebenfalls einem Additionsbefehl, bis auf die Tatsache, daß der CWP um eins inkrementiert wird. Dabei wird das Fenster CWP+1 zum aktuellen

Fenster, und das Fenster des Aufrufers wird »wiederhergestellt«. Außerdem stammen die Quellregister aus dem (alten) CWP-Fenster, das Zielregister hingegen liegt im (neuen) Fenster CWP+1. Der RESTORE-Befehl wird meist in seiner trivialen Form (alle drei Register *g0*) benutzt, um zur aufrufenden Routine zurückzukehren. Er wird vorteilhaft im *delay slot* des Rücksprungbefehls untergebracht.

SAVE und RESTORE vergleichen beide den neuen CWP mit dem enspr. Wert im WIM und lösen bei Fensterüberlauf oder -unterlauf eine Ausnahme aus.

Die Befehle der Gruppe Ticc und RETT werden im Abschnitt Ausnahmebehandlung besprochen.

Lesen und Schreiben von Kontrollregistern: Diese Befehle transferieren Information zwischen den r-Registern und den Status- und Steuerregistern (siehe dort). Die Befehle sind privilegiert (z.B. auch auf die *Ancillary State Registers*, wenn diese Register privilegiert sind).

Fließkommaoperationsbefehle FPop haben i.A. triadische Registeradressen. Sie berechnen ein Resultat aus zwei f-Registerquelloperanden und legen es in dem f-Zielregister ab. Ausnahmen sind die FP-Konvertierungsoperationen, die nur einen Quell- und einen Zieloperanden haben, und die FP-Vergleichsoperationen, die keinen Zieloperanden haben, aber das *fcc*-Feld des FSR beeinflussen.

Falls eine FPU nicht vorhanden oder das EF-Feld im PSR gelöscht ist, löst der Versuch der Ausführung eines FPop-Befehls eine *fp_ disabled*-Ausnahmebehandlung aus.

Alle FPop-Befehle löschen das *ftt*- und setzen das *cexc*-Feld im FSR. Einige FPop-Befehle setzen auch das *fcc*-Feld. Alle FPop-Befehle, die IEEE-Ausnahmen auslösen können, setzen die *cexc*- und *aexc*-Felder, es sei denn es käme zur Ausnahme. FABS, FMOV und FNEG können solche Ausnahmen nicht hervorrufen, also löschen sie *cexc* und lassen *aexc* unberührt.

Coprozessoroperationsbefehle CPop werden vom angeschlossenen CP ausgeführt. Wenn ein solcher nicht vorhanden oder das CE-Feld im PSR gelöscht ist, löst der Versuch der Ausführung eines CPop-Befehls eine *cp_ disabled*-Ausnahmebehandlung aus.

Der Terminus CPop beinhaltet keine CP-Sprung und -Lade/Speicherbefehle. Näheres zur Befehlscodierung findet man im Anhang F von [SPARC92].

4.11 Ausnahmebehandlung: *Traps*

Ausnahmebehandlung auf Prozessorebene findet mittels sog. *Traps* statt. *Trap* heißt eigentlich Falle, und sie wirkt auch wie eine Falltür: Man gelangt mit ihrer Hilfe sofort in den *Supervisor*-Modus, in dem dann privilegierte System-Software das Notwen-

dige zur Behandlung einer aufgetretenen Ausnahmesituation veranlassen kann, um anschließend die normale Befehlsabarbeitung wiederaufzunehmen – oder auch nicht, falls dies nicht sinnvoll sein sollte.

Eine *Trap* bewirkt einen nicht verzögerten Kontrolltransfer in den Supervisor-Modus über eine besondere, sog. *Trap*-Tabelle, die die ersten vier Befehle des Codes enthält. Diese Tabelle ist zuvor von der System-Software eingerichtet worden, die dann ihre Basisadresse in das TBA-Feld des TBR einträgt. Die Ablage in die Tabelle wird vom Typ der *Trap* bestimmt: Die eine Hälfte der Tabelle ist reserviert für Hardware-induzierte *Traps*, die andere Hälfte für Software-*Traps*, die mittels der Ticc-Befehle erzeugt werden.

Eine *Trap* wirkt wie ein unerwarteter Prozeduraufruf. Sie dekrementiert den CWP, um ein neues Registerfenster zu erhalten, und die Hardware des Prozessors schreibt PC und nPC sofort in zwei *local*-Register des neuen Fensters (*l1* und *l2*). Üblicherweise rettet die Behandlungsroutine sofort das PSR in ein weiteres *l*-Register und darf dann die anderen fünf *l*-Register nach Belieben benutzen.

Anmerkung: Interessanterweise geschieht das Umschalten auf ein neues Fenster sofort und ohne Rücksicht auf einen Fensterüberlauf, auch eine *window_overflow*-Ausnahme wird nicht ausgelöst. Das bedeutet, daß die System-Software immer ein Fenster für allfällige *Traps* freihalten muß. Sie tut dies, indem sie im WIM ein entsprechendes Fenster zu diesem Zweck als benutzt markiert. Also kann bei einer solchen Konstruktion immer ein Fenster weniger von Anwendungsprogrammen benutzt werden, als eigentlich vorhanden wäre. Typischerweise wird hierfür das letzte Fenster gewählt, das ja wegen der zirkulären Struktur mit dem ersten überlappt und deswegen sowieso nicht voll genutzt werden könnte.

Eine *Trap* kann durch einen ausnahmeauslösenden Befehl bewirkt werden, oder durch eine externe Unterbrechungsanforderung. Sie kann präzise sein, d.h. direkt ausgelöst werden, bevor der Zustand des Programms sich geändert hat, oder aber auch mit Verzögerung von einigen Befehlen auftreten. Eine externe Unterbrechungsanforderung muß gar keinen Zusammenhang mit gerade ablaufenden Befehlen haben. Spezielle *Trap*-Befehle Ticc (*Trap on integer condition codes*) existieren, um Anwenderprogrammen Dienste des Betriebssystems zugänglich zu machen. *Traps* werden vielfältig von Hard- und Software genutzt, zur Emulation nicht implementierter Befehle, zum Einzelschrittverfahren beim Testen, für Zeitmessungen etc., von der Implementation und vom Betriebssystem. Aufgelaufene *Traps* können in einer optionalen Warteschlange zwecks späterer Abarbeitung gesammelt werden. Alle Eizelheiten können in dieser Einführung nicht ausgebreitet werden, dazu sei auf das Kapitel 7 in [SPARC92] verwiesen.

Der *Trap*-Mechanismus wird von den Bit-Feldern ET (*enable traps*) und PIL (*Processor Interrupt Level*) im PSR, sowie TEM im FSR kontrolliert. Siehe dazu die Tabelle der Ausnahme- und Unterbrechungsprioritäten. Die höchste Priorität ist 1, die niedrigste 31.

Tabelle der Ausnahme- und Unterbrechungsprioritäten

Ausnahme oder Unterbrechung	Priorität	*tt*
reset	1	—
data_store_error	2	0x2B
instruction_access_MMU_miss	2	0x3C
instruction_access_error	3	0x21
r_register_access_error	4	0x20
instruction_access_exception	5	0x01
privileged_instruction	6	0x03
illegal_instruction	7	0x02
fp_disabled	8	0x04
cp_disabled	8	0x24
unimplemented_FLUSH	8	0x25
watchpoint_detected	8	0x0B
window_overflow	9	0x05
window_underflow	9	0x06
mem_address_not_aligned	10	0x07
fp_exception	11	0x08
cp_exception	11	0x28
data_access_error	12	0x29
data_access_MMU_miss	12	0x2C
data_access_exception	13	0x09
tag_overflow	14	0x0A
division_by_zero	15	0x2A
trap_instructions	16	0x80 - 0xFF
interrupt_level_15	17	0x1F
interrupt_level_14	18	0x1E
interrupt_level_13	19	0x1D
interrupt_level_12	20	0x1C
interrupt_level_11	21	0x1B
interrupt_level_10	22	0x1A
interrupt_level_9	23	0x19
interrupt_level_8	24	0x18
interrupt_level_7	25	0x17
interrupt_level_6	26	0x16
interrupt_level_5	27	0x15
interrupt_level_4	28	0x14
interrupt_level_3	29	0x13
interrupt_level_2	30	0x12
interrupt_level_1	31	0x11
impl.-dependent exception	impl.-dep.	0x60 - 0x7F

Als Aufrufmechanismus von Systemroutinen dienen die sog. Software-*Traps*, die mithilfe der Ticc-Befehle realisiert werden. Das sind 16 Befehle im Format 3, strukturiert wie die Bicc-Gruppe, jedoch allesamt *nicht* verzögert.

Tabelle der Ticc-Verzweigungsbefehle

Opcode	*cond*	Operation	*icc*-Bit-Test
TA	1000	Trap Always	1
TN	0000	Trap Never	0
TNE	1001	Trap on Not Equal	not Z
TE	0001	Trap on Equal	Z
TG	1010	Trap on Greater	not(Z or(N xor V))
TLE	0010	Trap on Less or Equal	Z or(N xor V)
TGE	1011	Trap on Greater or Equal	not(N xor V)
TL	0011	Trap on Less	N xor V
TGU	1100	Trap on Greater Unsigned	not(C or Z)
TLEU	0100	Trap on Less or Equal Unsigned	C or Z
TCC	1101	Trap on Carry Clear (GE Unsigned)	not C
TCS	0101	Trap on Carry Set (Less Unsigned)	C
TPOS	1110	Trap on Positive	not N
TNEG	0110	Trap on Negative	N
TVC	1111	Trap on Overflow Clear	not V
TVS	0111	Trap on Overflow Set	V

Ablauf einer *Trap*: Die Bedingung dazu ist ET = 1, d.h. *Traps* sind zugelassen.

- *Traps* werden abgeschaltet: ET $\leftarrow$ 0
- Der aktuelle Prozessormodus wird gerettet: PS $\leftarrow$ S
- Der Prozessor wird in den Supervisor-Modus geschaltet: S $\leftarrow$ 1
- Das Registerfenster wird ohne (!) Prüfung weitergeschaltet:
  CWP $\leftarrow$ ((CWP - 1) modulo NWINDOWS)
- Die Programmzähler werden in die *local*-Register 1 und 2 geschrieben:
  l1 $\leftarrow$ PC, *l2* $\leftarrow$ nPC
- Das *tt*-Feld im TBR wird auf den die jeweilige Ausnahme oder Unterbrechungsanforderung identifizierenden Wert gesetzt, außer im Falle von *reset* und *error_mode*
- Im Falle einer *Reset Trap* geht die Ausführung zur Adresse 0:
  PC $\leftarrow$ 0, nPC $\leftarrow$ 4
  Andernfalls geht die Ausführung in die *Trap*-Tabelle:
  PC $\leftarrow$ TBR, nPC $\leftarrow$ TBR + 4

Falls bei ET = 0 dennoch eine direkte (d.h. befehlsinduzierte) *Trap* auftreten sollte, geht der Prozessor in den Fehlerzustand und hält an. Unterbrechungsanforderungen oder verzögerte *Traps* werden ignoriert.

Rückkehr aus der *Trap*: Dadurch daß bei einer *Trap* beide Programmzähler gerettet werden, bieten sich interessante Möglichkeiten, nämlich den Befehl zu wiederholen, nachdem man etwas passendes getan oder ihn u.U. ausgetauscht hat, oder aber am nPC, also nach dem auslösenden Befehl fortzufahren, nachdem man ihn vielleicht emuliert hat.

Für die Rückkehr aus einer *Trap* ist der (privilegierte) Befehl RETT (*Return from Trap*) vorgesehen. Diesem *muß* (sic!) ein JMPL-Befehl direkt[21] vorausgehen, sonst landet man u.U. im falschen Adreßraum!

RETT kann selbst auch wieder eine *Trap* auslösen. Wenn das nicht der Fall ist, geschieht folgendes:

- Das Registerfenster wird zurückgeschaltet:
  CWP $\leftarrow$ ((CWP + 1) modulo NWINDOWS)
- Eine verzögerte Verzweigung zur Zieladresse wird begonnen[22]
- Der Prozessor wird in seinen vorherigen Modus geschaltet: S $\leftarrow$ PS
- *Traps* werden wieder zugelassen: ET $\leftarrow$ 1

Falls der *Trap*-auslösende Befehl wiederholt werden soll, wird diese Schlußsequenz empfohlen:

```
jmpl    %l1, %g0    ! alter PC

rett    %l2         ! alter nPC
```

Falls nach dem *Trap*-auslösenden Befehl fortgefahren werden soll, ist diese Sequenz angesagt:

```
jmpl    %l2, %g0    ! alter nPC

rett    %l2+4       ! alter nPC + 4
```

[21] Diese Unschönheit (Software-Zwang) wurde bei SPARC v9 beseitigt, siehe dort.

[22] RETT ist ein unbedingter, verzögerter Sprungbefehl, der selbst im *delay slot* des ihm vorausgehenden JMPL steht (stehen muß) und dadurch mit diesem ein DCTI-Paar bildet. Das bewirkt, daß der auf RETT folgende Befehl nie zur Ausführung kommt, deshalb braucht ihm auch kein NOP zu folgen. Siehe hierzu [SPARC92], S.55 ff., *Delayed Control Transfer Instruction Couples*.

5 Neuerungen bei SPARC Version 9

In der Entwicklung der SPARC gab es von v7, in der die ersten kommerziellen Implementationen erschienen, zu v8, dem Subjekt dieser Einführung und zu diesem Zeitpunkt noch mit der größten Anzahl installierter Systeme, keine allzugroßen Änderungen. Das ist mit dem Sprung zur Version 9, kurz v9 genannt, deutlich anders: Es ist die Weiterentwicklung zu einer modernen **64-Bit-Architektur**. Allerdings wurde auch hier dafür gesorgt, daß für v8 geschriebene – und kompilierte – Anwendungssoftware problemlos auf v9-Installationen ablaufen kann.

Dieser Abschnitt kann natürlich keine komplette Einführung in die v9-Architektur bieten. Das definierende Dokument dazu ist [SPARC95]. Auch [SPARC00] enthält im Anhang E (S. 85 ff.) eine Beschreibung der Änderungen im Hinblick auf die Assembler-Programmierung. Hier soll nur ein Kurzüberblick gegeben werden. Dieser kann Tendenzen erkennen lassen und ist auch interessant im Zusammenhang mit der Entwicklung anderer RISC-Architekturen (z.B. [Sites92], [Dobb92], [Marg90] und [Weiss94]).

Änderungen im Registersatz:

- Alle IU-Arbeitsregister sind von 32 auf 64 Bit erweitert, + 8 alternative *globals*
- Die Register FSR, PC, nPC, Y sind von 32 auf 64 Bit erweitert, im FSR wurden die zusätzlichen Felder *fcc1, fcc2* und *fcc3* definiert
- Die Kontroll- und Statusregister PSR, TBR und WIM sind verschwunden
- In v8 ehedem als Bit-Felder in Kontrollregistern vorhanden, wurden daraus nun eigenständige Register:
  CCR *Condition Codes Register*
  CWP *Current Window Pointer*
  PIL *Processor Interrupt Level*
  TBA *Trap Base Adress*
  TT[MAXTL] *Trap Type*
  VER *Version*
- Diese Register wurden neu hinzugefügt:
  ASI *Address Space Identifier*
  CANRESTORE *Restorable Windows*
  CANSAVE *Savable Windows*
  CLEANWIN *Clean Windows*
  FPRS *Floating-Point Register State*
  OTHERWIN *Other Windows*
  PSTATE *Processor State*
  TICK *Hardware clock tick-counter*

TL *Trap Level*
TNPC[MAXTL] *Trap Next Program Counter*
TPC[MAXTL] *Trap Program Counter*
TSTATE[MAXTL] *Trap State*
WSTATE *Window State*

- Außerdem werden zusätzlich 16 FP-Arbeitsregister in doppelter Genauigkeit (64 Bit) bereitgestellt, *f32...f62*. Diese überlappen (und sind abgebildet auf) 8 zusätzliche FP-Register in vierfacher Genauigkeit (128 Bit), *f32...f60*.

- Das CWP-Register in v9 wird bei SAVE inkrementiert und bei RESTORE dekrementiert, genau umgekehrt wie bei v8. Diese Änderung hat keine Wirkung auf nichtprivilegierte Software.

Änderungen im Alternativen Adreßraum:

- Lade- und Speicherbefehle in einer Hälfte des Alternativen Adreßraums sind nunmehr auch im User-Modus möglich: Der Zugriff auf ASI's 0x00 ... 0x7F bleibt weiterhin privilegiert, derZugriff auf ASI's 0x80 ... 0xFF ist – im Gegensatz zu v8 – nicht privilegiert.

Änderungen bezüglich der Byte-Anordnung im Speicher:

- Sowohl *Big Endian*, als auch *Little Endian* Speicherordnung werden unterstützt, allerdings nur beim Zugriff auf Daten! Befehlszugriffe werden wie bisher ausschließlich in *Big Endian*-Manier ausgeführt.

Änderungen im Befehlssatz:

- Die folgenden Befehle sind verschwunden:
  CP-Lade- und Speicherbefehle (CP-Konzept wurde aufgegeben)
  RDTBR/WRTBR TBR existiert nicht mehr, es wurde durch das TBA ersetzt, das mit RDPR/WRPR bearbeitet werden kann
  RDWIM/WRWIM WIM existiert nicht mehr, durch andere Register ersetzt
  RDPSR/WRPSR PSR existiert nicht mehr, es wurde durch mehrere separate Register ersetzt, die mit anderen Befehlen bearbeitet werden
  RETT *Return from Trap* wurde ersetzt durch DONE/RETRY
  STDFQ *Store Double from Floating-point Queue* ersetzt durch RDPR FQ

- Die folgenden Befehle wurden geändert oder hinzugefügt:
  IMPDEPn *Implementation-dependent instr.* ersetzen die CPop-Befehle von v8
  MEMBAR *Memory Barrier* Unterstützung für Speichersynchronisation

- Folgende Befehle wurden definitionsmäßig für 64-Bit-Operationen erweitert:
  FCMP, FCMPE *Floating-point Compare* setzt beliebige der vier *fcc*'s
  LDFSR/STFSR *Load/Store FSR* nur die niederwertigen 32 Bit des FSR
  LDUW, LDUWA *Load Unsigned Word* entsprechen LD und LDA bei v8
  RDASR/WRASR *Read/Write State Registers* bearbeiten weitere Register
  SAVE/RESTORE
  SETHI
  SRA, SRL, SLL Schiebebefehle jetzt aufgeteilt in 32-Bit- und 64-Bit-Operationen
  Tcc *Trap on condition codes*, ersetzt Ticc und operiert entweder auf 32-Bit (*icc*) oder 64-Bit (*xcc*) Bedingungscodes

- Die folgenden neuen Befehle dienen der Unterstützung von 64-Bit-Operationen:
  F[sdq]TOx *Convert floating-point to 64-bit word*
  FxTO[sdq] *Convert 64-bit word to floating-point*
  FMOV[dq] *Floating-Point Move, double and quad*
  FNEG[dq] *Floating-Point Negate, double and quad*
  FABS[dq] *Floating-Point Absolut Value, double and quad*
  LDDFA, STDFA, LDFA, STFA *Alternate address space forms of* LDDF etc.
  LDSW *Load Signed Word*
  LDSWA *Load Signed Word from alternate address space*
  LDX *Load Extended Word*
  LDXA *Load Extended Word from alternate address space*
  LDXSFR *Load all 64 bits of the FSR register*
  STX *Store Extended Word*
  STXA *Store Extended Word into alternate address space*
  STXSFR *Store all 64 bits of the FSR register*

- Diese neuen Befehle sollen die Implementation von Hochleistungssystemen unterstützen:
  BPcc *Branch on integer condition codes with prediction*
  BPr *Branch on integer register contents with prediction*
  CASA, CASXA *Compare and Swap from an alternate space*
  FPBfcc *Branch on floating-point condition code with prediction*
  FLUSHW *Flush Windows*
  FMOVcc *Move FP-register if condition code is satisfied*
  FMOVr *Move FP-register if integer register satisfies condition*
  LDQF(A)/STQF(A) *Load/Store Quad Floating-point (in an alternate space)*
  MOVcc *Move integer register if condition code is satisfied*
  MOVr *Move integer register if register contents satisfy condition*
  MULX *Generic 64-bit Multiply*
  POPC *Population Count*
  PREFETCH, PREFETCHA *Prefetch Data (from alternate space)*
  SDIVX, UDIVX *Signed and unsigned 64-bit Divide*

Wie man aus den oben zusammengestellten Daten ersehen kann, ist der Datenzugriff auf eine Breite von 64 Bit aufgeweitet und der Adreßraum auf 2^{64} Byte vergrößert worden. Die Befehlswortgröße bleibt aber weiterhin 32-Bit, wie auch bei anderen 64-Bit Architekturen üblich. Das ermöglicht u.U. die Abarbeitung von zwei Befehlen pro Takt (*Dual Issue*). Das in der Praxis zu selten genutzte Konzept der Integration eines Coprozessors (CP) wurde aufgegeben.

Die Bereitstellung eines speziellen TICK-Registers mit einem Zähler (max. 63 Bit, min. 10 Jahre) im Prozessortakt ermöglicht sehr feine und genaue Zeitmessungen. Die Abschaffung des PSR und die Implementierung dessen wichtiger Bit-Felder als eigenständige Register dient der Beschleunigung des System-Codes (Bit-Schiebeoperationen zur Extraktion und Manipulation entfallen). Desgleichen beschleunigt die Aufteilung der Aufgaben des WIM auf mehrere neue Register den Fenstermechanismus. Eine erhebliche Umstrukturierung, Verfeinerung und Verbesserung erfuhr der *Trap*-Mechanismus, der mit eigenem Stack, Sicherungs- und Statusregistern, sowie zusätzlichen acht alternativen *global*-Registern ausgestattet ist und nun auch eine sichere Schachtelung von *Traps* erlaubt.

Für beschleunigten Ablauf auch im User-Modus sorgen die neuen bedingten Verzweigungsbefehle mit Vorhersage (BPcc und FBPcc), sowie die Verzweigungsbefehle mit Vorhersage (BPr), die Inhalte von Arbeitsregistern testen. Das gleiche gilt für die bedingten Registertransferbefehle (MOVcc, MOVr, FMOVcc, FMOVr), die bei Bedingungskonstrukten so manche Verzweigung einsparen helfen.

Bei den neuen 64-Bit Multiplikations- und Divisionsbefehlen wurde mit der umständlichen Einbeziehung des Y-Registers Schluß gemacht. Dies, wie auch die alten Befehle existieren zwar noch, sind aber als *deprecated* bezeichnet und sollten in neuem Code nicht mehr verwendet werden.

Im neuen Handbuch [SPARC95] fällt auf, daß die bisherige Terminologie der fachlichen Bezeichnungen gegenüber v8 geändert und vereinheitlicht wurde. Dies ist hilfreich bei der Beschreibung der immer komplexer werdenden Materie.

Insgesamt stellt die Entwicklung zur Stufe v9 der SPARC einen sehr großen Schritt in Richtung Erweiterung und Modernisierung dar, bei der die Wahrung der Kompatibilität im User-Modus gelungen ist, ohne den Fortschritt zu behindern. Eine Betrachtung dieser Entwicklung der SPARC im Vergleich mit anderen modernen RISC-Architekturen[23] ist sicher interessant.

Erweiterungen zu SPARC Version 9: Hier handelt es sich hauptsächlich um die von Sun eingebrachten Erweiterungen zu v9, nämlich UltraSPARC und VIS (*Visual Instuction Set*). Diese basieren auf v9 und stellen weitere Register (u.a. GSR *Graphics Status Register*), Befehle (z.B. SHUTDOWN) und insbesondere einen Satz von Grafikbefehlen (aus der Gruppe IMPDEP1) zum schnellen Bearbeiten von Bilddaten durch die FPU zur Verfügung. Programme, die diese Erweiterungen benutzen, laufen natürlich nur auf entsprechend ausgerüsteten Systemen.

[23] Einige Anregungen dazu finden sich im Literaturverzeichnis.

6 Assemblerprogrammierung

Die Wichtigkeit der Assemblerprogrammierung hat seit geraumer Zeit stark abgenommen und sie wird – zum kleineren Anteil – außer in der Systemprogrammierung, wo man auf alle, also auch die Spezial- und Kontrollregister zugreifen muß, nur noch für spezielle Routinen verwendet, bei denen es auf höchste Effizienz oder Geschwindigkeit ankommt (Numerik, Grafik, Akustik, kritische Echtzeitanwendungen).

Für den Hochsprachenprogrammierer ist dennoch eine gewisse Kenntnis der Assemblerprogrammierung von Nutzen, um ein Gespür für die Maschine und das Arbeiten des Prozessors zu bekommen, und das nicht nur, wenn gerade die Entwicklung eines Codegenerators für einen Compiler oder eines Assemblers ansteht. Es hilft auch, wenn's manchmal hakt mit der Hochsprachenprogrammierung, z.B. bei der Fehlersuche. Interessantes und Anregendes zu dieser Thematik – und dazu die Beschreibung einer hypothetischen RISC-Architektur – findet man in [Knuth99].

Assemblerprogramme sind in Syntax und Organisation nicht nur spezifisch von dem Prozessor abhängig, für den sie geschrieben werden, sondern auch von der jeweiligen Entwicklungs-Software und dem Betriebssystem. Wir verwenden hier Komponenten des C-Entwicklungssystems unter Sun Solaris, einer Unix SVR4 Variante, das ebenfalls von Sun bereitgestellt wird und das gängige System auf SPARC Workstations darstellt. Da die erzeugten Programme als Anwendungen im User-Modus laufen, können wir unter diesem System leider nur die Nutzung der nicht privilegierten Befehle der SPARC demonstrieren.

6.1 Das C-Entwicklungssystem

Das C-Entwicklungssystem besteht aus mehreren Programmen, die in modularer Form die Bearbeitung von gemischten Programm-Quelltexten bis zur Erzeugung von Programmbibliotheken oder einer lauffähigen Version übernehmen. Die wichtigsten Komponenten – mit ihren traditionellen Namen – sind:

`cc`	C-Compiler-Kontrollprogramm	(C compiler driver)
`cpp`	C-Präprozessor	(C preprocessor)
`cc1`	eigentlicher C-Übersetzer	(C compiler, 1st stage)
`as`	Assembler	(assembler)
`ld`	Binder	(linker, orig. loader)

Ferner gibt es noch verschiedene Debugger, Profiler und eine Reihe weiterer Dienstprogramme, die helfen, Quelltexte und Bibliotheken zu verwalten, sowie größere, modulare Programmierprojekte zu organisieren, die uns in diesem Zusammenhang aber nicht weiter interessieren sollen.

`cc1` heißt bei Sun neuerdings `acomp`, das braucht uns jedoch nicht weiter zu stören, da wir alle Komponenten, also vornehmlich `as` und `ld`, über `cc` bedienen werden. `cc` ist nicht der C-Compiler, sondern das Steuerprogramm des ganzen Übersetzungssystems, dem Optionen und die zu bearbeitenden Dateien als Parameter übergeben

werden und das dann nach Bedarf die einzelnen Komponenten mit ihren zugehörigen Optionen aktiviert.

`cc` erkennt die Art der ihm zugeführten Dateien an ihrer Endung, die konventionsgemäß so aussehen:

`xxx.c`	C-Quelltext
`xxx.i`	Zwischendatei, von `cpp` erzeugt
`xxx.s`	Assembler-Quelltext, von `cc1` erzeugt oder selbst geschrieben
`xxx.S`	Assembler-Quelltext, der noch von `cpp` bearbeitet werden muß
`xxx.o`	Object-Code, Binärdatei im Prozessor-Format, von `as` erzeugt
`a.out`	Default-Name für lauffähiges Programm, von `ld` erzeugt

Suns cc kennt eine Fülle von Optionen oder Schaltern, die den Verlauf der weiteren Verarbeitung steuern. Einige hier verwendete sind:

`cc -flags`	liste Optionen (ca. 2 Seiten DIN A4)
`cc -# ...`	verboser Modus
`cc -c ...`	nur compilieren und Object-Code (`.o`) erzeugen
`cc -S ...`	nur compilieren und Assembler-Code (`.s`) erzeugen
`cc ...`	erzeuge lauffähiges Programm a.out
`cc -o <ofile> ...`	erzeuge Ausgabedatei mit Namen `ofile`

Besonders interessant ist der Schalter `-S`, mit dem der C-Compiler aus einem C-Quelltext eine assemblierfähige Assembler-Datei erzeugt, die man dann studieren, evtl. editieren und wieder zu lauffähigem Code assemblieren lassen kann. In der Datei ist in auskommentierter Form auch noch der C-Code enthalten, was bei der ersten Orientierung helfen kann. Außerdem befinden sich, je nach eingeschalteter Optimierungsstufe, noch mehr oder weniger (auskommentierte) Identifikationstexte, Adreßangaben, etc. in der Datei, die der Übersichtlichkeit halber besser entfernt werden sollten. Da wenige Zeilen C seitenlange Abschnitte in Assembler erzeugen können, sollte man den zu bearbeitenden C-Code möglichst kurz halten.

6.2 C-, Unix- und SPARC-Konventionen

Wenn wir unter dem Unix/Sun-Solaris Betriebssystem im Rahmen des C-Entwicklungssystems Assemblerprogramme schreiben wollen, sind wir auf die Nutzung dieser Laufzeitumgebung angewiesen, d.h. wir müssen in der Lage sein, Bibliotheks-Routinen der C-Bibliothek (*library calls*) und Dienstroutinen von Unix (*system calls*) nutzen zu können (z.B. für Eingabe/Ausgabe etc.). Auch muß unser Programm selbst von dieser Umgebung aufgerufen werden können. Für diese besonderen Zwecke unterliegt die Programmierung den **C-Konventionen**.

Die hier relevante C-Variante ist das standardisierte ANSI/ISO-C [ISO99]. Das Standardwerk für eine konzise Einführung in die Sprache, ihre Bibliotheken und den Programmierstil ist [Kern89]. Weitere exzellente Titel zu diesem Thema sind [Plau95], [Harb95] und speziell für die Unix-Systemprogrammierung [Stev92].

C ist eine eingeschränkt blockstrukturierte Sprache, ihr Hauptgliederungselement sind Funktionen. Sie haben die Form:

Rückgabetyp `Funktionsname(` *Typ param1, Typ param2, ..., Typ param_ n* `)`
`{`
    Funktionskörper
`}`

Die Systemaufrufe ließen sich natürlich in Assembler direkt ansprechen, jedoch ist diese Technik erstens schlecht dokumentiert und zweitens haben sie sämtlich traditionell eine C-Schnittstelle, dokumentiert im Kap. 2 des Unix-Manuals. Sie können also wie C-Funktionen bedient werden, die in Kap. 3 dokumentiert sind.

Zur Übergabe der Argumente, für die Speicherung lokaler Variablen und zur Entgegennahme des Rückgabewertes von Funktionen – falls vorhanden oder gewünscht – sind auch weitere **Unix-** und spezifische **SPARC-Konventionen** zu beachten. Für die Zuordnung der Registernamen siehe den umseitigen Registerbelegungsplan.

- Der Aufruf einer Funktion in C hat die Form: `func(arg1, arg2, ..., argn)` oder `func()`, falls keine Argumente vorhanden sind. Die Argumente sind von links nach rechts in aufsteigender Reihenfolge in den Registern `%o0 ... %o5` zu übergeben, wenn das nicht reicht, werden weitere auf dem *Stack* abgelegt, wo die gerufene Routine sie beginnend bei `%sp+92` erwartet. Zeiger und Ganzzahlen werden als Rückgabewert vom Rufer in Register(n) – je nach Größe, z.B. Doubleword in `%o0`, `%o1` – beginnend mit `%o0` erwartet, Fließkommawerte ab `%f0`. Für die (seltenen) Funktionen, die ein C `struct` zurückgeben, gilt ein besonderes Verfahren (siehe [SPARC92], D.3., S.196).

- Jedes ausführbare Programm enthält genau eine Funktion **main()**. Sie stellt den Einstiegspunkt bei der Abarbeitung des Programms dar, und wird als solche vom automatisch hinzugebundenen Startup-Code aufgerufen. Der Typ ist `int main(...)` und tritt üblicherweise in den Varianten `int main(void)` bzw. `int main(int argc, char* argv[])` auf. Dabei ist der Rückgabewert ein Ganzzahltyp, der erfolgreiche (=0) oder fehlerhafte ($\neq$0) Ausführung signalisieren soll. Die beiden optionalen Parameter sind die Anzahl der Argumente (argc) und der Argumentvektor (argv), ein Zeiger auf ein 0-terminiertes Array von Adressen, die auf C-Strings weisen und die dem Programm übergebenen Parameter darstellen. Der erste Parameter, `argv[0]` ist gewöhnlich der Aufrufname des Programms, somit ist `argc` immer mindestens 1.

- Der von `main()` an das OS retournierte Wert sind die niederwertigen 8 Bit des Inhalts des Standard-Rückgaberegisters (*o0*). Bei normaler Terminierung eines aus der Bourne Shell gestarteten Programms wird dieser Wert in der Variablen `$?` gespeichert.

- Falls der Platz im Registersatz nicht ausreichen sollte oder ihre Adressen benötigt werden, sind lokale Variablen auf dem *Stack* abzulegen.

- Eine Routine, die selbst keine weiteren Routinen aufruft (*Leaf Procedure*), kann auf das Weiterschalten des Registerfensters (mittels SAVE-Befehl) verzichten. Sie läuft also im Fenster und im *Stack Frame* des Aufrufers und darf dann nur die *o*-Register benutzen, in denen dann auch die evtl. übergebenen Argumente stehen. Möglich wäre auch die Benutzung von *g*-Registern, falls diese nicht reserviert sind (siehe dazu die nächsten Punkte).

- Die Benutzung der Register ist durch einen Registerbelegungsplan (s.w.u.) vorgegeben. Insbesondere ist auf Erhaltung von `%o6` bzw. `%i6` – Stack- und Frame Pointer, sowie `%o7` und `%i7` – Adresse des CALL-Befehls – zu achten.

- Die globalen Register unterliegen der zur Zeit gültigen Konvention, die besagt, daß die Inhalte der *g*-Register, natürlich mit Ausnahme von *g0*, über Prozeduraufrufe hinweg als flüchtig zu betrachten sind. Suns C-Compiler macht davon, besonders bei *Leaf Procedures*, auch reichlichen Gebrauch, z.B. zur Speicherung temporärer Werte.

  In der Belegungstabelle abgebildet ist die im SPARC ABI geforderte Konvention, nach der nur das Register *g1* als flüchtig angenommen wird. Diese Konvention gilt z.B. für Bibliotheks-Code. Die Register *g2...g4* und *g5...g7* können dann zwischen den Routinen als globale Variablen oder globale Zeiger auf häufig gebrauchte Variablenbereiche verwendet werden, um die Ablaufgeschwindigkeit des Codes zu steigern. Wenn dies in einer solchen Umgebung nicht erforderlich sein sollte, müßten die Registerinhalte, wie auch sonst bei anderen Architekturen üblich, vor Verwendung der Register für andere Zwecke gesichert und danach wieder restauriert werden.

- Der Wert des Stackpointers wird beim Start eines Programms von Unix in `%o6` eingetragen (nach einem SAVE erscheint er dann in `%i6` als Framepointer). Selbstverständlich darf er von der aktiven Routine manipuliert werden, nur müssen die ersten 64 Byte (16 Words) ab `%sp` *immer unbedingt freigehalten* werden, sie dienen nämlich zur Rettung der aktuellen *i*- und *l*-Register im Falle einer Ausnahmebehandlung, die ja bekannterweise jederzeit und asynchron auftreten kann. Zur vom Unix-System vorgegebenen Stack-Auslegung siehe weiter unten.

- Die Fließkommaregister müssen wie die globalen Register mittels Software verwaltet werden. Über Prozeduraufrufe hinweg gelten sie als flüchtig. Compiler nutzen sie zuweilen zur Speicherung irgendwelcher interner Variablen und von Zwischenergebnissen. Die Rückgabe von Funktionswerten im Fließkommaformat geschieht in den Registern ab `%f0`. Existierende Compiler nutzen die Fließkommaregister *nicht* zur Parameterübergabe. Die zur Zeit gültige Konvention besagt, daß der Aufrufer seine ihm wichtigen FP-Register vor dem Aufruf einer Prozedur zu retten hat.

6.2.1 Registerbelegungsplan

	%r31	%i7	Rückkehradresse - 8
	%r30	%i6	Frame Pointer %fp
	%r29	%i5	hereinkommender Parameter 6
in	%r28	%i4	hereinkommender Parameter 5
	%r27	%i3	hereinkommender Parameter 4
	%r26	%i2	hereinkommender Parameter 3
	%r25	%i1	hereinkommender Parameter 2
	%r24	%i0	hereinkommender Parameter 1 / Rückgabe an Aufrufer
	%r23	%l7	local 7
	%r22	%l6	local 6
	%r21	%l5	local 5
local	%r20	%l4	local 4
	%r19	%l3	local 3
	%r18	%l2	local 2
	%r17	%l1	local 1
	%r16	%l0	local 0
	%r15	%o7	temporärer Wert / Adresse des CALL-Befehls
	%r14	%o6	Stack Pointer %sp
	%r13	%o5	Übergabeparameter 6
out	%r12	%o4	Übergabeparameter 5
	%r11	%o3	Übergabeparameter 4
	%r10	%o2	Übergabeparameter 3
	%r9	%o1	Übergabeparameter 2
	%r8	%o0	Übergabeparameter 1 / Rückgabewert vom Gerufenen
	%r7	%g7	global 7 (SPARC ABI reserviert)
	%r6	%g6	global 6 (SPARC ABI reserviert)
	%r5	%g5	global 5 (SPARC ABI reserviert)
global	%r4	%g4	global 4 (SPARC ABI globale Variable)
	%r3	%g3	global 3 (SPARC ABI globale Variable)
	%r2	%g2	global 2 (SPARC ABI globale Variable)
	%r1	%g1	temporärer Zwischenspeicher (scratch register)
	%r0	%g0	0
	%y		Y-Register für Multiplikation und Division
Status		(*icc*)	Integer Condition Codes
		(*fcc*)	Floating-Point Condition Codes
		(*ccc*)	Coprocessor Condition Codes
	%f31		Fließkommawert
floating	:		:
point	:		:
	%f0		Fließkommawert

6.2.2 Stack-Auslegungsplan

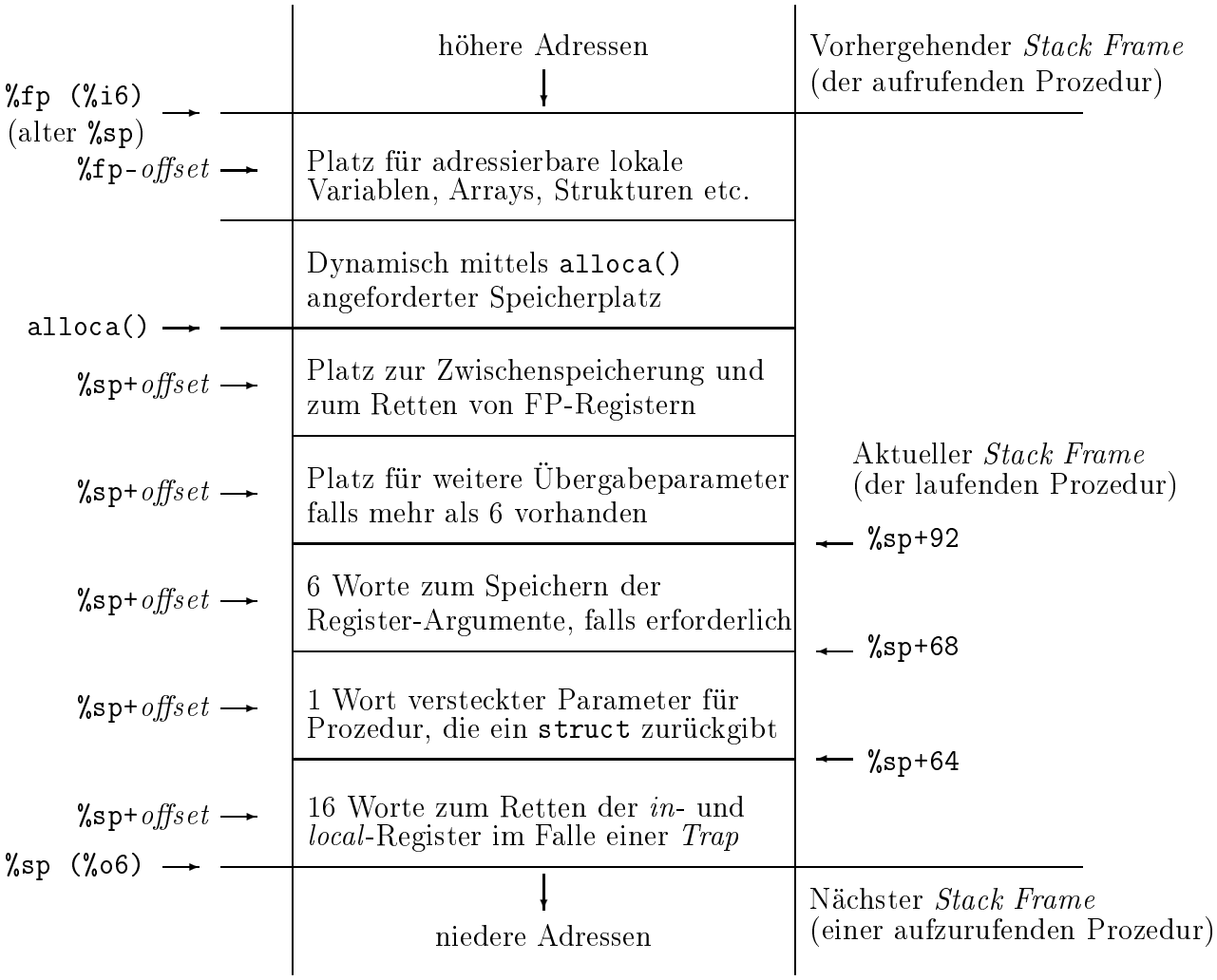

Abbildung 11: Typische Auslegung eines *Stack Frame* im User-Modus

Bei Beachtung dieser Konventionen sind wir nun in der Lage, Assemblerroutinen zu schreiben, die C-Routinen rufen, als auch solche, die von C aus gerufen werden. Für den Rest genießen wir völlige Freiheit in dem Rahmen, den uns der SPARC-Prozessor im **User-Modus** bietet.

6.2.3 Speicherauslegung eines Unix-Prozesses

Als Prozeß wird unter Unix die Instanz eines zur Ausführung in den Speicher geladenen Programms bezeichnet. Der Prozeß läuft in einem separaten virtuellen Adreßraum, sein von ihm zur Laufzeit belegter Speicher wird in verschiedene Bereiche – auch Segmente genannt – aufgeteilt:

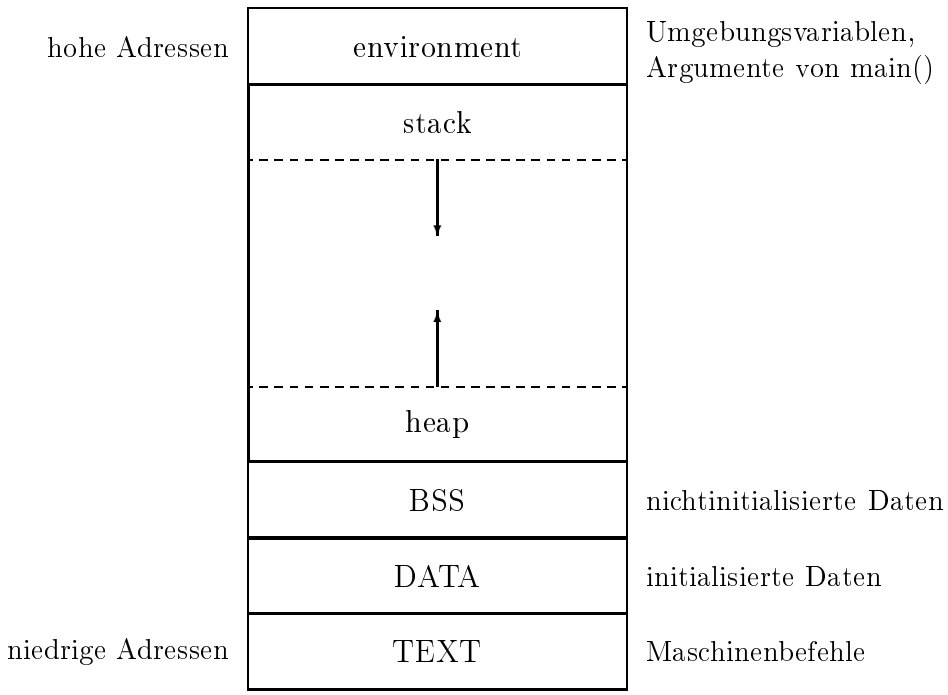

Abbildung 12: Typische Speicherauslegung eines Unix-Prozesses

- TEXT – der ausführbare Programm-Code – Maschinenbefehle, compiliert, assembliert, handcodiert, hinzugebunden, so wie in der Programmdatei abgelegt.
- DATA – initialisierte Daten zum Lesen und Schreiben, wie beim TEXT, nur eben keine Maschinenbefehle, sondern Daten. Auf vielen Systemen gibt es hier noch ein besonderes Segment RODATA[24] – als Bereich für initialisierte Daten nur zum Lesen, z.B. Platz für Strings, Konstanten, etc. Auch diese Daten müssen in der Programmdatei vorhanden sein.
- BSS – Block Storage Segment – Bereich für nicht initialisierte Daten. Dieser Bereich wird nicht in der Programmdatei gespeichert, nur die Größe des Bedarfs. Unter C und Unix wird dieses Segment zum Programmstart ausgenullt bereitgestellt.
- *Heap* – dieser Bereich zur Datenspeicherung wird nach Bedarf zur Laufzeit vom Programm angefordert und verwaltet. Dazu dienen die C-Bibliotheks-Funktionen `malloc()`, `calloc()`, `realloc()`, `free()` und die Systemaufrufe `brk()` und `sbrk()`.
- *Stack* – hier werden u.a. lokale Variablen gespeichert, Register gesichert etc. Dieser Bereich wird durch den *Stack Pointer* (SP) verwaltet, dessen Wert vom System beim Programmstart vorgegeben wird. Ein anderer Zeiger in diesen

[24] Read Only DATA

Bereich ist der *Frame Pointer* (FP), von dem aus die im *Stack Frame* liegenden Variablen angesprochen werden. Jede Routine hat ihren eigenen *Stack Frame*, der beim Start aufgebaut und bei der Rückkehr wieder abgebaut wird. Dadurch ist z.B. Rekursion möglich. Der *Stack* wächst von oben nach unten, Speicherplatz dafür wird vom System nach Bedarf zur Verfügung gestellt. Die C-Funktion `alloca()` reserviert z.B. Speicherplatz auf dem Stack. Der *Stack Pointer* sollte auch auf einer 32-Bit-Maschine immer die Ausrichtung 8 haben, weil die Ausnahmeroutinen beim Retten von Registerinhalten aus Geschwindigkeitsgründen mit den Befehlen LDD und STD arbeiten.

Anmerkungen zu Besonderheiten der SPARC bei Benutzung des *Stack*:
Bei den üblichen CISC[25]-Architekturen, die zur Benutzung des *Stack* über sehr kurze Spezialbefehle wie PUSH und POP o.ä. verfügen (Speichern auf dem Stack mit Prädekrement und Laden vom Stack mit Postinkrement des SP) ist der SP dauernd in Bewegung: Für die Argumentübergabe an aufzurufende Prozeduren, zum Sichern ihrer Rückkehradresse und zur kurzfristigen Zwischenspeicherung - bedingt durch die relativ kleinen Registersätze mit teils spezialisierten Registern. Der FP hingegen, über den sowohl die übergebenen Argumente (+ Ablage) als auch die lokalen Variablen (– Ablage) adressiert werden und in dem der Wert des SP der aufrufenden Prozedur gespeichert ist, bleibt im Verlauf der Prozedur stabil.

Allgemein ist bei RISC-Architekturen und insbesondere bei der SPARC mit ihrer Fenstertechnik den SP betreffend ein anderes Vorgehen erforderlich: Es gibt keine speziellen Befehle PUSH und POP, der Registersatz ist groß genug und die Register sind größtenteils allgemein verwendbar. Beim Einsatz des SAVE-Befehls wird am Beginn der aufgerufenen Prozedur ein neues Registerfenster besorgt und gleichzeitig - atomar - kann ein neuer *Stackframe* geschaffen werden. Dieser wird so ausgelegt, daß er für den gesamten Lauf der Prozedur ausreichend[26] bemessen ist. Der SP der aufrufenden Prozedur, `%sp = %o6`, erscheint ohne weiteres Kopieren bei der gerufenen Prozedur als `%fp = %i6`, die ihn nun von hier zum Adressieren der im *Stackframe* des Aufrufers liegenden zusätzlichen Argumente verwenden kann (ab `%fp+92`). Falls erforderlich, wenn z.B. die Adresse benötigt wird, werden die bis zu sechs[27] in Registern übergebenen Argumente ab `%fp+68`, also im *Stackframe* des Aufrufers (!) gespeichert. RESTORE gibt Fenster und *Stackframe* wieder frei.

- *Environment* – Der sog. Umgebungsbereich wird dem Prozeß vom Betriebssystem zur Verfügung gestellt. Hier liegen die Strings der Umgebungsvariablen. Auch die Argumente, die dem Programm auf der Kommandozeile mitgegeben werden, befinden sich in diesem Bereich.

[25] Complex Instruction Set Computer, von RISC-Propagandisten erfundenes Acronym
[26] mindestens 64, üblich sind 96 und mehr, nach Bedarf, siehe Abbildung 11 auf S. 42
[27] siehe hierzu die Fußnote 3 in [SPARC92], Anhang D, S.189 ff.

6.3 Assembler-Syntax

Das definierende Dokument für die vom Sun/Solaris-Assembler akzeptierte Syntax (und den Assembler überhaupt, seine Kommandozeile, das von ihm generierte Objektformat ELF, etc.) ist [SPARC00]. Hier wird nur ein kleiner Ausschnitt geboten, soweit es für das Verständnis und zum Testen unserer Beispielprogramme erforderlich ist.

Im angelsächsischen Gebrauch unterscheidet man zwischen *assembly language* – der Programmiersprache – und *assembler* – dem Programm, das diese Sprache in Objekt-Code übersetzt, während man im Deutschen üblicherweise beides Assembler nennt.

Assemblerquelltext hat die Form einer Textdatei mit der Endekennung .s und stellt so eine Übersetzungseinheit für den Assembler dar. Man erstellt sie z.B. mit dem jeweiligen Lieblingseditor. Der Assembler akzeptiert auch mehrere Quelldateien, die er dann konkateniert, oder die Standard-Eingabe.

Hat man den Quelltext erzeugt, z.B. `myasm.s`, so ruft man den Assembler, wie weiter oben beschrieben, falls `myasm.s` eine eigenständige `main()`-Funktion enthält, mit

```
cc myasm.s
```

andernfalls mit

```
cc -c myasm.s
```

und erwartet als Ergebnis dieser Aktion entweder ein ausführbares Programm namens `a.out` oder eine zu anderm Zwecke bestimmte Objektdatei namens `myasm.o`.

Möglicherweise wird die Erwartung aber nicht erfüllt und man wird vom Assembler mit diversen Fehlermeldungen begrüßt. Diese sollte man sorgfältig studieren, notwendige Verbesserungen am Quelltext vornehmen, und es ein weiteres Mal versuchen.

Aus der reichhaltigen Liste der Assembler-Optionen hier noch einige, die dazu dienen, Objekte für bestimmte SPARC-Versionen zu generieren:

Option	Wirkung auf das Ausgabeformat (Details siehe Handbuch)
`-xarch=v7`	ELF für SPARC v7
`-xarch=v8`	ELF für SPARC v8
`-xarch=v8a`	ELF für SPARC v8, aber ohne fsmuld
`-xarch=v8plus`	ELF für SPARC v9, Solaris V8+ (64-Bit Proz./32-Bit OS)
`-xarch=v8plusa`	ELF für SPARC v9, + VIS, läuft auf Solaris V8+
`-xarch=v9`	64-Bit ELF für SPARC v9, nur mit Obj. gleichen Formats

Assembler ist eine zeilenorientierte Sprache. Eine Assemblerdatei besteht also aus Zeilen.

Zeilen können leer sein – als visuelles Gliederungselement, um die Übersichtlichkeit und Lesbarkeit des Quelltextes zu erhöhen. Den Assembler stört das nicht, den Menschen – wenn sinnvoll angewandt – erfreut's. Zeilen, oder Teile davon, können

auch Kommentare enthalten, um das Verständnis des menschlichen Lesers zu erhöhen und über Sinn, Zweck und Wirkungsweise des Programms Auskunft zu geben. Für den Assembler haben sie keine Bedeutung.

! leitet den Kommentar ein und wirkt bis zum Ende der Zeile. C-Kommentare, in `/*` *alles hierzwischen ist Kommentar* `*/` eingefaßt, werden auch akzeptiert. Sie können sich über mehrere Zeilen erstrecken und sind nicht schachtelbar.

Der Rest der Datei, das eigentliche Programm also, besteht aus Zeilen, die Anweisungen enthalten. Dies sind die Hauptelemente:

- Direktiven – das sind Anweisungen an den Assembler, bezüglich der Organisation des Programms und des zu erzeugenden Objekt-Codes. Direktiven heißen bei Sun *pseudo op*.

- Marken – auf englisch *label*, stehen für Adressen von zu erzeugenden Objekten im Speicher, die der Assembler dann ausrechnet. Dies können Adressen von Daten sein, z.B. von Variablen, die man anlegen möchte, von zu reservierendem Speicherplatz, den man ansprechen will, oder von Code, zu dem man springen möchte. Anstatt das alles selbst mühsam auszurechnen, setzt man eine Marke und überläßt die Arbeit dem Assembler.

- Befehle – das sind die Befehle des SPARC-Prozessors, mnemonische Schlüsselwörter, die dann unter Einbeziehung ihrer etwaigen Argumente vom Assembler in binären Maschinen-Code (32-Bit Wörter) übersetzt werden. Der Sun Assembler kennt auch noch sog. synthetische Befehle[28], die wir aber hier kaum benutzen wollen, um ein Gefühl für die Architektur zu bekommen.

Die Syntax einer Zeile sieht so aus:
Zeile ::= [*Anweisung* [; *Anweisung*] *] [! *Kommentar*]

Also: Eine Zeile besteht aus einer Folge von beliebig vielen, semikolonseparierten Anweisungen, gefolgt von einem optionalen Kommentar. So weit wollen wir es aber nicht treiben, und lassen es der Übersichtlichkeit halber bei einer Anweisung pro Zeile bewenden. Das ist auch der übliche Stil.
Eine Anweisung wird definiert als:

Anweisung ::= [*Marke*:] [*Direktive* | *Befehl*]

Eine Anweisung besteht aus einer optionalen Marke, das ist ein Symbolname[29] mit unmittelbar folgendem Doppelpunkt (:), optional gefolgt von einer Direktive oder einem Befehl.

[28] [SPARC92], A.3, S.85 oder [SPARC00], 5.5, S.56. Eine Auswahl findet sich im Anhang A.

[29] Der Sun Assembler kennt auch numerische Marken. Sie bestehen aus einer Ziffer mit folgendem Doppelpunkt, sind lokal und können mehrfach vorkommen. Wir verwenden sie hier nicht.

Ein Symbolname hat folgende Syntax (als regulärer Ausdruck in Unix-Notation):

```
[A-Za-z_$\.][A-Za-z_$\.0-9]*
```

Also: Buchstabe, Unterstrich, Dollarzeichen ($) oder Punkt (.), gefolgt von beliebig vielen (auch 0) aus dieser Gruppe, inclusive der Ziffern von 0–9.

Symbolnamen mit führendem Unterstrich sind vom C-System reserviert und sollten nicht verwendet werden! Symbolnamen mit führendem Punkt werden als lokale Symbole betrachtet und sollten nur mit Bedacht verwendet werden!

Das Symbol Punkt (.) ist vordefiniert: Es ist der sog. *Location Counter*, LC, der Zähler für den Ort in der Code-Generierung, an dem man sich gerade befindet. Man könnte ihn auch den »Hier« -Zähler nennen.

Zahlen erscheinen in der C-Notation (jedoch ohne Suffixe) als Dezimal-, Oktal- (mit führender 0), oder Hexadezimalzahlen (mit führendem `0x` oder `0X`). Fließkommakonstanten werden mit der Notation `0r` oder `0R` (für REAL) angeführt, gefolgt von einem String, wie ihn auch die C-Funktion atof(3) akzeptiert. Die besonderen Namen `0rnan` und `0rinf` bezeichnen die Fließkommawerte *Not-A-Number (NaN)* und *INFinity*. Ihre negativen Werte bezeichnet man mit `0r-nan` und `0r-inf`.

Zeichenketten oder *Strings* notiere man wie in C, desgleichen einzelne Zeichen. Die in C üblichen *Escape Codes* werden vom Assembler innerhalb von Strings erkannt.

Spezialsymbol- und Registernamen beginnen sämtlich mit dem Prozentzeichen (%) und sind vom Assembler reserviert. Hier eine Auswahl:

Objekt des Symbols	Name	Bemerkungen
Allgemeine Register	`%r0...%r31`	
Allgemeine globale Register	`%g0...%g7`	entspricht `%r0...%r7`
Allgemeine out Register	`%o0...%o7`	entspricht `%r8...%r15`
Allgemeine local Register	`%l0...%l7`	entspricht `%r16...%r23`
Allgemeine in Register	`%i0...%i7`	entspricht `%r24...%r31`
Stackpointer-Register	`%sp`	`%sp = %o6 = %r14`
Framepointer-Register	`%fp`	`%fp = %i6 = %r30`
Fließkommaregister	`%f0...%f31`	
FP-Status-Register	`%fsr`	
Beginn der FQ	`%fq`	nur Supervisor
Prozessor-Status-Register	`%psr`	nur Supervisor
Trap Base Register	`%tbr`	nur Supervisor
Window Invalid Mask	`%wim`	nur Supervisor
Y Register	`%y`	
Unärer Operator	`%hi`	extrahiert die oberen 22 Bit
Unärer Operator	`%lo`	extrahiert die unteren 10 Bit

In der Anwendung werden die rollenspezifischen Registernamen (z.B. `%g0`, `%o2`, `%sp`, etc.) bevorzugt, weil sie die Lesbarkeit erhöhen. Im Gegensatz zu den anderen Symbolnamen ist bei den Spezialsymbolen Groß- und Kleinschreibung äquivalent, `%fsr` ist also identisch mit `%FSR`, Kleinschreibung wird allerdings bevorzugt.

Der Assembler kann auch rechnen und kennt allerlei Operatoren für Ganzzahlarithmetik etc.:

`+`	Addition	`<<`	Links Schieben		`+`	ohne Wirkung
`-`	Subtraktion	`>>`	Rechts Schieben		`--`	Zweierkomplement
`*`	Multiplikation	`^`	Bit-XOR		`-`	Einerkomplement
`/`	Division	`&`	Bit-UND		`%hi(addr)`	entspr. `addr >> 10`
`%`	Modulo	`|`	Bit-ODER		`%lo(addr)`	entspr. `addr & 0x3FF`

Die Operatoren `%hi` und `%lo` haben allerhöchsten Vorrang, so daß sich empfiehlt, ihre Argumente immer in Klammern zu setzen!

z.B.: `%hi a+b` wird interpretiert als `(%hi a) + b` ACHTUNG!

Um den Modulo-Operator (%) nicht mit einem der Spezialsymbole zu verwechseln, darf er nicht unmittelbar von einem Buchstaben gefolgt werden. Er steht üblicherweise vor einem Leerzeichen oder einer öffnenden Klammer.

Als Zuweisungsoperator dient wie in C das Gleichheitszeichen (=), damit realisiert man die in Assembler-Parlance sog. *Equates*, meist Zuweisungen an Symbole, z.B. für Adreßrechnungen und zum Ausrechnen konstanter Ausdrücke.

Symbol = *Ausdruck* z.B. `len = m2 - m1`

weist dem Symbol `len` den Wert der Adreßdifferenz der Marken `m1` und `m2` zu. `len` kann dann im Programm als symbolische Konstante benutzt werden.

Direktiven sind ebenfalls vom Assembler reservierte Symbolnamen und beginnen in Unix-Tradition immer mit einem Punkt (.). Die Auswahl der hier vorgestellten Direktiven steuert den Assembler bei der Auslegung der zu erzeugenden Objektdatei und sorgt für notwendige Linker-Informationen.

Wie im weiter oben im Abschnitt über die Speicherauslegung eines Unix-Prozesses erläutert, besteht so ein Prozeß aus verschiedenen Abschnitten, Segmente oder auch manchmal Sektionen genannt, die ihr Abbild in der Objekt-Datei haben. Ausführbarer Code und Daten werden in solchen Segmenten angelegt. Dazu dient die `.section`-Direktive mit folgender Syntax:

`.section` *Sektions-Name* [, *Attribute*]

Der Linker setzt dann alle gleichnamigen Sektionen zusammen und organisiert sie in entsprechenden Bereichen der Objekt-Datei, die ein ausführbares Programm oder

auch eine Programm-Bibliothek sein kann. Für Anwendungsprogramme sind folgende vordefinierte Sektions-Namen gedacht (eine Auswahl):

Sektions-Name	vorgesehener Inhalt
`.text`	ausführbarer Programm-Code
`.data`	initialisierte Daten zum Lesen und Schreiben
`.data1`	initialisierte Daten zum Lesen und Schreiben
`.rodata`	initialisierte Daten nur zum Lesen
`.rodata1`	initialisierte Daten nur zum Lesen
`.bss`	nicht initialisierte Daten zum Lesen und Schreiben

Diese Sektions-Namen werden in vom Sun-Compiler generierten Assembler-Code mit Anführungsstrichen versehen, was in dieser Form, zusammen mit der `.section` Direktive, auch für neuere Versionen des Assemblers notwendig ist. (Sonst kann man auch einfach `.data`, `.text` etc. verwenden, ohne `.section` Direktive.) Die optionalen Attribute, `#write`, `#alloc` und `#execinstr` sind nicht unbedingt erforderlich, so daß wir uns hier auch nicht weiter damit beschäftigen wollen. (Die Dokumentation dazu im Handbuch [SPARC00] ist auch nicht sehr aufschlußreich.)

In den auf diese Weise spezifizierten Sektionen sollen schließlich auch nützliche Objekte in Form von ausführbarem Code oder Daten angelegt werden. Die nun folgenden Direktiven sorgen für korrekte Ausrichtung, Speicherplatzreservierung und das Bekanntmachen von Symbolen:

Direktive	Parametersyntax	Wirkung
`.align`	*Grenze*	sorgt für die Ausrichtung (*alignment*) des folgenden Codes (Daten oder Befehle) im Speicher. *Grenze* muß eine Zweierpotenz sein. Bei einer 32-Bit-Maschine ist eine Ausrichtung auf 4 zu Beginn eines Segmentes eigentlich immer zu empfehlen, bei einer 64-Bit-Maschine sogar auf 8.
`.skip`	*n*	erhöht den *Location Counter* um *n* und schafft damit *n* Byte freien Speicherplatz.
`.global`	*Symbol* [, *Symbol*]*	macht *Symbol* nach außen bekannt, oder besagt, daß *Symbol* anderswo existiert, so daß Referenzen darauf vom Linker gefunden und verbunden werden können (d.h. sowohl Import als auch Export von Symbolen).

Die folgenden Direktiven dienen der Reservierung von Speicherplatz für Daten. Achtung: Diese Direktiven bewirken keine automatische Ausrichtung, dafür muß man, falls erforderlich, mittels `.align` sorgen!

Direktive	Parametersyntax	Anmerkungen
`.ascii`	*String* [, *String*]*	ohne autom. Null-Byte
`.asciz`	*String* [, *String*]*	autom. nullterminiert
`.byte`	*8-Bit-Wert* [, *8-Bit-Wert*]*	
`.half`	*16-Bit-Wert* [, *16-Bit-Wert*]*	
`.word`	*32-Bit-Wert* [, *32-Bit-Wert*]*	
`.single`	`0r`*Fließkommawert* [, `0r`*Fließkommawert*]*	einfache Genauigkeit
`.double`	`0r`*Fließkommawert* [, `0r`*Fließkommawert*]*	doppelte Genauigkeit
`.quad`	`0r`*Fließkommawert* [, `0r`*Fließkommawert*]*	erweiterte Genauigkeit

Die beiden Direktiven `.ascii` und `.asciz` unterscheiden sich nur dadurch, daß letztere bei der Ablage im Speicher automatisch ein Null-Byte anfügt, so daß ein korrekter C-String produziert wird. Das kann man natürlich auch selber machen:

`.ascii "Hello World\n\0"` entspricht `.asciz "Hello World\n"`

Bei der zweiten Version vermeidet man jedoch, die terminierende Null zu vergessen.

Bleiben schließlich noch die Befehle. Im Handbuch [SPARC92] werden im Anhang A Vorschläge zur Assembler-Syntax gemacht, sowie im Anhang B ausführlich zu jeder Beschreibung eines Befehls die empfohlene Syntax vorgestellt. Das Handbuch [SPARC00], das ja die tatsächliche Implementation des Sun-Assemblers beschreibt, folgt diesen Vorschlägen. Hier findet man im Kapitel 5, *Instruction Set Mapping*, alle implementierten Befehle, ihre Syntax, die Zuordnung zu den SPARC-Befehlen und Anmerkungen.

Das alles hier zu reproduzieren, würde den Rahmen dieser kleinen Abhandlung bei weitem sprengen. Außerdem wird man ohnehin für die Klärung von Details und zum Studium aller in dieser Einführung nicht behandelten Eigenschaften die Originalliteratur konsultieren müssen. Die Arbeit mit den – oft sehr guten – Originaldokumenten hat auch so ihre Tücken, ist aber eine gute Übung.

Übersicht über einige typische Befehlsformate:

Befehlskürzel Quellregister 1, Quellregister 2 oder Direktwert, Zielregister

```
add     %o3, %o4, %l2           /* l2 = o3 + o4 */
sub     %o1, 815, %i0           /* i0 = o1 - 815 */
or      %g0, %g0, %o0           /* o0 = 0 */
```

Beim Speicherzugriff wird der Adreßausdruck in eckige Klammen gesetzt:

Ladebefehlskürzel [*Adresse*], *Zielregister*

```
ld      [%o1], %o5              /* o5 = *o1 */
```

Speicherbefehlskürzel Quellregister, [*Adresse*]

```
st      %i5, [%fp-4]            /* *(fp-4) = i5 */
```

Weitere Beispiele kann man den kommentierten Beispielprogrammen entnehmen.

6.4 Beispielprogramme

Assembler-Beispielprogramme[30] und -routinen sind in der Literatur leider etwas rar. Die kleine Auswahl der hier angebotenen Beispielprogramme soll einen Einblick in verschiedene Techniken der SPARC-Programmierung bieten und ein Gefühl für diese interessante RISC-Architektur vermitteln.

Alle Beispielprogramme sind lauffähig. Falls es sich um Unterroutinen handelt, ist – bis auf wenige Ausnahmen – ein kleiner Testrahmen in C angegeben. Sie wurden auf einer UltraSPARC unter Solaris 8 mit dem Sun C-Compiler (Version Sun WorkShop 6 update 2 C 5.3 Patch 111679-03 2001/09/29) und dem Assembler (Version Sun WorkShop 6 99/08/18) getestet. Die Quelltexte wurden mittels eines Hilfsprogramms in LaTeX überführt und in den Quelltext dieser Einführung eingebunden. Die Zeilennummern dienen der Referenzierung und gehören nicht zum Quelltext.

Programmierstil: Ein einheitlicher Stil, man spricht in diesem Zusammenhang von *Coding Conventions*, erleichtert das Verständnis und den Austausch von Quelltexten; denn bekanntlich sind Lesen und Verstehen von Programmen ja um etliches schwerer als das Schreiben derselben.

Äußerst wichtig sind konsistente Einrückung und gute Formatierung. Die Beispielprogramme versuchen auch dies zu demonstrieren. Einrückungen erfolgen mit TAB 4, die Texte enthalten aber keine TABs, damit sie auf allen Medien (Drucker, Editor, Konsole) gleich aussehen. C wird nach Kontrollebene eingerückt, in Assembler ist dies nicht üblich. Man orientiere sich an den Beispielen.

Einige Bemerkungen zur Kommentierung: In Assembler sind Restzeilenkommentare auf der Befehls- oder Direktivenzeile üblich. Kommentare können, falls erforderlich, auch gebündelt vor größeren Blöcken von Code stehen.

Es sollte allerdings nur kommentiert werden, was *nicht* offensichtlich ist. Daß ein ADD-Befehl addiert, sollte auch so klar sein, aber vielleicht ist es hilfreich zu wissen, was, weshalb oder warum gerade hier addiert wird.

Überflüssige Kommentare sind nicht nur lästig, sie sind *schädlich*. Weniger ist oft mehr! Durch geschickte Wahl der Namen von Variablen und Marken kann man sich viel quälende Kommentierung sparen und damit die Lesbarkeit wesentlich erhöhen!

Ein Registerbelegungsplan hilft, die Übersicht zu wahren, und erleichtert das Verständnis; dies ist besonders wichtig bei RISC-CPUs mit ihren großen durchnumerierten Registersätzen.

Wie bei Einführungsmaterial üblich, sind einige der Beispiele stark überkommentiert. Das ist kein Produktionsstil und sollte *nicht* nachgeahmt werden!

[30] Unter dem Titel *An Example Language Program* findet sich ein compilergeneriertes und von Sun reichlich nachkommentiertes Beispielprogramm in [SPARC00], Anhang D, S.79-83, das Fibonacci-Zahlen berechnet. Die Lektüre kann einen guten Einblick in die Organisation von compilergeneriertem Assembler-Code geben.

Ein C-Testrahmen für ein Assembler-Unterprogramm könnte so aussehen:

```
 1: /* t_xyz.c   -   Testrahmen fuer xyz.s             lisa 7/98 */
 2: /* ------------------------------------------------------- */
 3: #include <stdio.h>   /* Definitionsdatei der Std. Bibliothek */
 4:
 5: int myint = 4711;                 /* eine (globale) Variable */
 6:
 7: int xyz(int x);                /* sog. Prototyp der Funktion */
 8:
 9: int main(void)                          /* das Hauptprogramm */
10: {
11:     int tmp;                    /* lokale Variable in main() */
12:
13:     tmp = xyz(myint);       /* Funktionsaufruf mit Zuweisung */
14:     printf("%d\n", tmp);      /* Aufruf einer Bibl.-Funktion */
15:
16:     return 0;                               /* beenden mit OK */
17: }
18: /* ------------------------------------------------------- */
```

Und so könnte das zu testende Assemblerprogramm aussehen:

```
 1: ! xyz.s   -   ixypsilont ein zett                    hugo 7/98
 2: ! ------------------------------------------------------------
 3: ! int xyz(int x); C-Prototyp der Funktion
 4: ! ------------------------------------------------------------
 5: ! IN:  Word %i0                Was kommt herein?
 6: ! OUT: Word %i0                Was wird zurueckgegeben?
 7: ! ------------------------------------------------------------
 8:     .section    ".text"       ! Beginn des Code-Segments
 9:     .align 4                  ! Ausrichtung
10:     .global xyz               ! Funktionsnamen veroeffentlichen
11: ! ------------------------------------------------------------
12: xyz:
13:     save    %sp, -64, %sp     ! neues Registerfenster
14:     andn    %i0, 0xF, %i0     ! durch 16 teilbar machen
15:
16:     jmpl    %i7+8, %g0        ! zurueck zum Aufrufer
17:     restore                   ! dessen Fenster wiederherstellen
18: ! ------------------------------------------------------------
```

Programm 1

```
 1: ! atimesb.s -  SPARC Assembler Demo                    br 12/2002
 2: ! ------------------------------------------------------------
 3: ! IN:   -                        ! Eingaben des Programms
 4: ! OUT:  %i0 = 0                  ! Rueckgabewert
 5: ! ------------------------------------------------------------
 6: ! %o0   format string            ! Registerbelegungsplan
 7: ! %o1   1st factor
 8: ! %o2   2nd factor
 9: ! %o3   result high, %o4 result low
10: ! ------------------------------------------------------------
11:     .section    ".rodata"
12: ! ------------------------------------------------------------
13: aaa: .word   1024               ! initialisierte Daten
14: bbb: .word   -1
15: fmt: .asciz  "0x%08X * 0x%08X = 0x%08X%08X\n"
16: ! ------------------------------------------------------------
17:     .section    ".text"
18:     .align  8
19:     .global main                ! Marke bekannt machen
20: ! ------------------------------------------------------------
21: main:
22:     save    %sp, -64, %sp       ! neues Fenster & Stackframe
23:     sethi   %hi(aaa), %o1       ! oberer Teil Adr. 2. Parameter
24:     sethi   %hi(bbb), %o2       ! oberer Teil Adr. 3. Parameter
25:     or      %o1, %lo(aaa), %o1 ! unterer Teil Adr. 2. Parameter
26:     or      %o2, %lo(bbb), %o2 ! unterer Teil Adr. 3. Parameter
27:     ld      [%o1], %o1          ! 2. Parameter laden
28:     ld      [%o2], %o2          ! 3. Parameter laden
29:     sethi   %hi(fmt), %o0       ! oberer Teil Adr. 1. Parameter
30:     smul    %o1, %o2, %o4       ! Resultat low: 4. Parameter
31:     rd      %y, %o3             ! Resultat high: 3. Parameter
32:     call    printf              ! Formatier- & Ausgabe-Routine
33:     or      %o0, %lo(fmt), %o0 ! unterer Teil Adr. 1. Parameter
34:
35:     or      %g0, %g0, %i0       ! Rueckgabewert des Programms
36:     jmpl    %i7+8, %g0          ! zurueck zum Aufrufer
37:     restore                     ! dessen Fenster wiederherstellen
38: ! ------------------------------------------------------------
```

Ein simples eigenständiges Hauptprogramm in SPARC-Assembler: Es multipliziert zwei 32-Bit-Ganzzahlen und gibt sie samt 64-Bit-Ergebnis an die Standardausgabe. Übermäßige Kommentierung soll die erste Zuordnung erleichtern.

Programm 2

```
 1: ! printbin.s - Wort-Binaerdarstellung zur Std.-Ausgabe   br 11/2001
 2: ! ---------------------------------------------------------------
 3: ! void printbin(int n)
 4: ! ---------------------------------------------------------------
 5: ! IN:    %o0 auszugebender 32-Bit-Wert
 6: ! OUT:   -
 7: ! USES:  %l0 - %l2, %o0 - %o1
 8: ! ---------------------------------------------------------------
 9:     .section     ".rodata"
10: fmt: .asciz     "%c"        ! Formatparameter fuer Datentyp "char"
11: ! ---------------------------------------------------------------
12:     .section    ".text"
13:     .align  8
14:     .global printbin
15: ! ---------------------------------------------------------------
16: ! %i0 : Eingangsparameter
17: ! %l0 : fuer Bit-Isolation gebraucht
18: ! %l1 : Schleifenzaehler, Startwert 31
19: ! %l2 : gerettete Adr. des Format-Strings
20: ! %o0 : Adr. des Format-Strings beim printf-Aufruf
21: ! %o1 : auszugebendes Zeichen, '1' oder '0'
22: ! ---------------------------------------------------------------
23: printbin:
24:     save    %sp, -96, %sp
25:
26:     sethi   %hi(fmt), %l2
27:     or      %g0, 31, %l1        ! Schleifenzaehler initialisieren
28:     or      %l2, %lo(fmt), %l2
29:     srl     %i0, %l1, %l0       ! gewuenschtes Bit in Pos. 0 bringen
30: .loop:
31:     and     %l0, 1, %o1         ! Bit isolieren
32:     or      %g0, %l2, %o0       ! Fmt. zum Aufruf bereitstellen
33:     call    printf
34:     add     %o1, 0x30, %o1      ! Wert in Zeichen wandeln
35:
36:     subcc   %l1, 1, %l1         ! herunterzaehlen
37:     bpos    .loop
38:     srl     %i0, %l1, %l0       ! gewuenschtes Bit in Pos. 0 bringen
39:
40:     jmpl    %i7+8, %g0
41:     restore
42: ! ---------------------------------------------------------------
```

```
 1: /* testprintbin.c                                        br 11/2001 - */
 2: /* ------------------------------------------------------------------ */
 3: #include <stdio.h>
 4: #include <stdlib.h>
 5:
 6: void printbin(int);
 7:
 8: int main(int argc, char *argv[])
 9: {
10:     if (argc < 2) {
11:         printf("Usage: %s some_int_number\n", argv[0]);
12:         return 1;
13:     }
14:     else
15:         printbin(atoi(argv[1]));
16:     printf("\n");
17:
18:     return 0;
19: }
20: /* ------------------------------------------------------------------ */
```

Bei printbin() handelt es sich um eine Subroutine in SPARC-Assembler, die von obigem Testrahmen in C aufgerufen wird. Mittels Bit-Schiebe-, Maskier- und Wandeloperationen werden die 32 Bit eines SPARC-Wortes (entspr. dem C-Datentyp int) in einer Schleife Bit für Bit von links nach rechts mittels der Bibliotheksfunktion printf(3) ausgegeben, so daß sich eine Darstellung im Binärformat ergibt.

Das Nachladen der Adresse des Formatparameters in *o0* in der Schleife (Z.32) ist notwendig, weil printf() den Inhalt von *o0* zerstört, indem es dort die Anzahl der ausgegebenen Zeichen hineinschreibt.

Programm 3
Eine ähnliche Routine ist das folgende printhex() zur Hexwandlung. Ein Testrahmen dazu wird nicht angegeben, er ist bis auf den Namen der aufzurufenden Funktion mit dem vorigen identisch. Hier geschieht die Wandlung von Wert nach Zeichen über eine Tabelle, dazu nutzt man den Speicherzugriff über Basis mit Index (Z.40).

```
 1: ! printhex.s  -  Wort-Hexdarstellung zur Std.-Ausgabe    br 11/2001
 2: ! ---------------------------------------------------------------------
 3: ! void printhex(int n)
 4: ! ---------------------------------------------------------------------
 5: ! IN:    %o0 auszugebender 32-Bit-Wert
 6: ! OUT:   -
 7: ! USES:  %l0 - %l4, %o0 - %o1
```

```
 8: ! ---------------------------------------------------------------
 9:     .section    ".rodata"
10: hex: .asciz     "0123456789ABCDEF"   ! Hex-Tabelle
11: fmt: .asciz     "%c"        ! Formatparameter fuer Datentyp "char"
12: ! ---------------------------------------------------------------
13:     .section    ".text"
14:     .align  8
15:     .global printhex
16: ! ---------------------------------------------------------------
17: ! %i0 : Eingangsparameter
18: ! %l0 : fuer Bit-Isolation gebraucht
19: ! %l1 : Schleifenzaehler, Startwert 7
20: ! %l2 : gerettete Adr. des Format-Strings
21: ! %l3 : Adr. der Hex-Tabelle
22: ! %l4 : Schiebewert
23: ! %o0 : Adr. des Format-Strings beim printf-Aufruf
24: ! %o1 : auszugebendes Zeichen, '0' ... 'F'
25: ! ---------------------------------------------------------------
26: printhex:
27:     save    %sp, -96, %sp
28:
29:     sethi   %hi(fmt), %l2
30:     sethi   %hi(hex), %l3
31:     or      %l2, %lo(fmt), %l2
32:     or      %l3, %lo(hex), %l3
33:     or      %g0, 7, %l1         ! Schleifenzaehler initialisieren
34:     sll     %l1, 2, %l4         ! Schiebewert fuer 4-Bit-Gruppe
35:     srl     %i0, %l4, %l0       ! gewuenschte Bit-Gruppe ans Ende
36: .loop:
37:     and     %l0, 0xF, %o1       ! Bit-Gruppe isolieren
38:     or      %g0, %l2, %o0       ! Fmt. zum Aufruf bereitstellen
39:     call    printf
40:     ldub    [%l3+%o1], %o1      ! Wert in Zeichen wandeln
41:
42:     subcc   %l1, 1, %l1         ! herunterzaehlen
43:     sll     %l1, 2, %l4         ! Schiebewert fuer 4-Bit-Gruppe
44:     bpos    .loop
45:     srl     %i0, %l4, %l0       ! gewuenschte Bit-Gruppe ans Ende
46:
47:     jmpl    %i7+8, %g0
48:     restore
49: ! ---------------------------------------------------------------
```

Programm 4

```
 1: ! onecount.s   -   population count                    br 1/2002
 2: ! ------------------------------------------------------------
 3: ! unsigned int onecnt(unsigned int a[], const int n)
 4: ! ------------------------------------------------------------
 5: ! %o0 : addr of int array    %o1 : dimension of int array
 6: ! %o2 : bit counter          %o3 : working element
 7: ! %o4 : offset into array    %o5 : running element counter
 8: ! %g1 : scratch
 9: ! ------------------------------------------------------------
10:     .section   ".text"
11:     .align  8
12:     .global onecnt
13: onecnt:
14:     orcc    %g0, %o1, %o1   ! n == 0 ?
15:     ble     .done
16:     or      %g0, 0, %o2     ! bit counter
17:
18:     or      %g0, 0, %o4     ! offset into array
19:     or      %g0, 0, %o5     ! array element counter
20:     ld      [%o0+%o4], %o3  ! get 1st array element
21: .oloop:
22:     subcc   %o3, 0, %g0     ! is it 0 ?
23:     be      .nextw
24:     sub     %o3, 1, %g1     ! prepare for x &= x-1
25:
26: .iloop:
27:     andcc   %o3, %g1, %o3   ! delete rightmost bit
28:     add     %o2, 1, %o2     ! inc bit counter
29:     bne     .iloop
30:     sub     %o3, 1, %g1     ! prepare for x &= x-1
31:
32: .nextw:
33:     add     %o5, 1, %o5     ! inc element cnt
34:     add     %o4, 4, %o4     ! advance to next elem
35:     subcc   %o5, %o1, %g0   ! are we finished ?
36:     bl,a    .oloop
37:     ld      [%o0+%o4], %o3  ! load next word
38:
39: .done:
40:     jmpl    %o7+8, %g0
41:     or      %g0, %o2, %o0   ! return count
42: ! ------------------------------------------------------------
```

```
 1: /* t_onecnt1.c     test driver for onecnt.s            br 1/2002 */
 2: /* ------------------------------------------------------------ */
 3: #include <stdio.h>
 4: unsigned int onecnt(unsigned int a[], const int n);
 5:
 6: int main(void)
 7: {
 8:     unsigned int a[] = {0xf0f0f0f0, 0xffffffff, 0x0101};
 9:     int i;
10:
11:     for (i=0; i<sizeof a / sizeof(int); ++i)
12:         printf("%08x\n", a[i]);
13:     printf("%d\n", onecnt(a, sizeof a / sizeof(int)));
14:     return 0;
15: }
16: /* ------------------------------------------------------------ */
```

Die Routine onecnt() implementiert den sog. *Population Count* über ein Array von Words, d.h. es zählt die Anzahl der gesetzten Bit. Hier werden verschiedene geschachtelte Schleifen und Sprünge geboten. Da es sich um eine sog. *Leaf Procedure* handelt, werden o und g Register genutzt. Der zweite Testrahmen eignet sich zur Leistungsmessung, evtl. auch einmal im Vergleich zu einer selbst zu schreibenden Routine mit dem in v9 vorhandenen Maschinenbefehl POPC.

```
 1: /* randarr.c - onecnt.s performance evaluation         br 1/2002 */
 2: /* ------------------------------------------------------------ */
 3: #include <stdio.h>
 4: #include <stdlib.h>
 5:
 6: #define SIZE 10000000
 7: static unsigned int ua[SIZE];
 8:
 9: unsigned int onecnt(unsigned int a[], const int n);
10:
11: int main(void)
12: {
13:     int i;
14:
15:     for (i=0; i<SIZE; ++i)
16:         ua[i] = rand();
17:     printf("%d\n", onecnt(ua, SIZE));
18:     return 0;
19: }
20: /* ------------------------------------------------------------ */
```

Programm 5

```
 1: ! heron.s - Heron's algorithm to calculate square root    br 1/2002
 2: !---------------------------------------------------------------
 3: ! double heron(double r)
 4: !---------------------------------------------------------------
 5:     .section   ".text"
 6:     .align  8
 7: const:
 8:     .double 0r0.5                   ! constant for division by 2
 9:
10:     .align  8
11:     .global heron
12: heron:
13:     sub     %sp, 72, %sp            ! get new stackframe
14:     sethi   %hi(const), %g1
15:     std     %o0, [%sp+64]           ! r           -> stack
16:     ldd     [%g1+%lo(const)], %f12  ! 0.5d        -> %f12, %f13
17:     ldd     [%sp+64], %f6           ! r           -> %f6, %f7
18:     fmuld   %f6, %f12, %f0          ! x = r/2     -> %f0, %f1
19:     fdivd   %f6, %f0, %f10          ! r/x         -> %f10, %f11
20:     faddd   %f10, %f0, %f2          ! r/x + x     -> %f2, %f3
21:     fmuld   %f2, %f12, %f14         ! (r/x + x)/2 -> %f14, %f15
22:     fcmpd   %f0, %f14               ! old x == new x ?
23:     nop
24:     fbe     .done
25:     nop
26:
27:     fmovd   %f14, %f8               ! x           -> %f8, %f9
28:     fdivd   %f6, %f8, %f4           ! r/x         -> %f4, %f5
29: .loop:
30:     fmovd   %f8, %f0                ! save old x  -> %f0, %f1
31:     faddd   %f4, %f8, %f4
32:     fmuld   %f4, %f12, %f8          ! (r/x+x)/2   -> %f8, %f9
33:     fcmpd   %f0, %f8                ! old x == new x ?
34:     nop
35:     fbne,a  .loop
36:     fdivd   %f6, %f8, %f4           ! r/x         -> %f4, %f5
37:
38: .done:
39:     jmpl    %o7+8, %g0              ! return result in %f0, %f1
40:     add     %sp, 72, %sp            ! release stackframe
41: !---------------------------------------------------------------
```

```
 1: /* t_heron.c  -   test driver for heron.s       br 3/2002 */
 2: /* ------------------------------------------------------ */
 3: #include <stdio.h>
 4:
 5: double heron(double r);
 6:
 7: int main(int argc, char *argv[])
 8: {
 9:     int i;
10:
11:     for (i=1; i<101; ++i)
12:         printf("%-20.015g\n", heron(i));
13:     return 0;
14: }
15: /* ------------------------------------------------------ */
```

Die Routine heron() ermittelt die Quadratwurzel nach »Herons Algorithmus«, der eine Spezialisierung des sog. Newton-Raphson-Verfahrens der iterativen Näherung darstellt. Der implementierte Algorithmus entspricht in C folgender Formulierung:

```
double heron(double r)
{
    double x;

    for (x=r/2; x!=(r/x+x)/2; x=(r/x+x)/2)
        ;
    return x;
}
```

Hier wird das Arbeiten mit der Fließkommaeinheit demonstriert. Auf einen separaten Registerbelegungsplan wurde aus Platzgründen verzichtet; die Kommentare müssen diesmal dafür herhalten. Der *double*-Eingabeparameter kommt über die Register *o0, o1* herein und muß zunächst in den Speicher gebracht werden, weil kein direkter Transfer zwischen den Registern von IU und FPU möglich ist. Dazu wird durch Herunterziehen des Stackpointers (hier ohne neues Fenster) Speicherplatz bereitgestellt. Zum Verarbeiten des SPARC-Datentyps Doubleword (entspr. dem C-Datentyp *double*) werden die Register der FPU paarweise (gerade, ungerade) genutzt. Bei Lade- und Speicheroperationen ist auf die korrekte Ausrichtung im Speicher zu achten (`.align 8`).

Nach Abspeichern des einkommenden Parameters, kann die FPU nun geladen werden. Anschließend werden die reichlich vorhandenen Registerpaare zum Speichern der Konstanten und diverser in der Iteration gebrauchter Zwischenwerte genutzt. Die Rückgabe des *double*-Ergebnisses erfolgt konventionsgemäß in *f0, f1*, so daß sich weiteres Umspeichern erübrigt.

Programm 6

```
 1: ! fp_rnd.s - set IEEE rounding mode in floating point state register
 2: !------------------------------------------------------- br 1/2002
 3: ! void fp_rnd(unsigned int rnd);
 4: !-----------------------------------------------------------------
 5: ! FPU's rounding mode is controlled by 2 hi bits in 32 bit wide FSR
 6: ! rounding mode of FPU can be:
 7: ! 0 - round to nearest (even if tie)
 8: ! 1 - round to zero (truncate)
 9: ! 2 - round to +infinity
10: ! 3 - round to -infinity
11: ! only lowest two bits of parameter's value are used
12: !-----------------------------------------------------------------
13: ! register map:
14: ! %o0   rounding mode, 2 bits, later shifted to position <31, 30>
15: ! %o1   address of word for storing fsr
16: ! %o2   mask for clearing rounding mode
17: ! %o3   used for manipulating fsr bits
18: !-----------------------------------------------------------------
19:     .section        ".data"
20:     .align  4
21: fsr:
22:     .word   0                   ! place to store fsr
23: !-----------------------------------------------------------------
24:     .section    ".text"
25:     .align  4
26:     .global fp_rnd
27:
28: fp_rnd:
29:     sethi   %hi(fsr), %o1       ! get hi part of address
30:     sethi   %hi(0xC0000000), %o2 ! prepare mask with two hi bits set
31:     or      %o1, %lo(fsr), %o1  ! get lo part of address
32:     st      %fsr, [%o1]         ! put contents of fsr into storage
33:     ld      [%o1], %o3          ! pull it from storage into register
34:     sll     %o0, 30, %o0        ! get rounding mode bits up high
35:     andn    %o3, %o2, %o3       ! clear rounding mode bits
36:     or      %o3, %o0, %o3       ! set rounding mode bits
37:     st      %o3, [%o1]          ! store word with modified bits
38:
39:     jmpl    %o7+8, %g0
40:     ld      [%o1], %fsr         ! load fsr with modified word
41: !-----------------------------------------------------------------
```

```
 1: ! getfsr.s - write contents of FSR to address      br 1/2002
 2: !-------------------------------------------------------------
 3: ! void getfsr(unsigned int *fsrbits);
 4: !-------------------------------------------------------------
 5:     .section    ".text"
 6:     .align  4
 7:     .global getfsr
 8: getfsr:
 9:     jmpl    %o7+8, %g0
10:     st      %fsr, [%o0]    ! store FSR to address
11: !-------------------------------------------------------------
```

```
 1: ! setfsr.s - load word at address into FSR         br 1/2002
 2: !-------------------------------------------------------------
 3: ! void setfsr(unsigned int *fsrbits);
 4: !-------------------------------------------------------------
 5:     .section    ".text"
 6:     .align  4
 7:     .global setfsr
 8: setfsr:
 9:     jmpl    %o7+8, %g0
10:     ld      [%o0], %fsr    ! load FSR from address
11: !-------------------------------------------------------------
```

Hier sind drei nützliche Routinen angegeben, die zeigen, wie man mit dem FSR in der FPU arbeiten kann. Mit fp_rnd() kann man den IEEE-Rundungsmodus der FPU beeinflussen. Die beiden anderen Routinen lesen bzw. schreiben das FSR. Interessant ist hier, daß die eigentliche Aktion im *delay slot* des Rücksprungbefehls untergebracht ist. Ein Testrahmen kann aus Platzgründen leider nicht angeboten werden, zumal die verschiedenen Nutzungsmöglichkeiten sehr zahlreich sind. Man betrachte dies als Aufforderung zu eigenem Experimentieren. Viel Spaß dabei!

A Befehlsübersicht

Opcode	*Name*
LDSB (LDSBA†)	Load Signed Byte (from Alternate space)
LDSH (LDSHA†)	Load Signed Halfword (from Alternate space)
LDUB (LDUBA†)	Load Unsigned Byte (from Alternate space)
LDUH (LDUHA†)	Load Unsigned Halfword (from Alternate space)
LD (LDA†)	Load Word (from Alternate space)
LDD (LDDA†)	Load Doubleword (from Alternate space)
LDF	Load Floating-point
LDDF	Load Double Floating-point
LDFSR	Load Floating-point State Register
LDC	Load Coprocessor
LDDC	Load Double Coprocessor
LDCSR	Load Coprocessor State Register
STB (STBA†)	Store Byte (into Alternate space)
STH (STHA†)	Store Halfword (into Alternate space)
ST (STA†)	Store Word (into Alternate space)
STD (STDA†)	Store Doubleword (into Alternate space)
STF	Store Floating-point
STDF	Store Double Floating-point
STFSR	Store Floating-point State Register
STDFQ†	Store Double Floating-point deferred-trap Queue
STC	Store Coprocessor
STDC	Store Double Coprocessor
STCSR	Store Coprocessor State Register
STDCQ†	Store Double Coprocessor deferred-trap Queue
LDSTUB (LDSTUBA†)	Atomic Load-Store Unsigned Byte (in Alternate space)
SWAP (SWAPA†)	Swap r Register with Memory (in Alternate space)
SETHI	Set High 22 bits of r Register
NOP	No Operation
AND (ANDcc)	And (and modify *icc*)
ANDN (ANDNcc)	And Not (and modify *icc*)
OR (ORcc)	Inclusive-Or (and modify *icc*)
ORN (ORNcc)	Inclusive-Or Not (and modify *icc*)
XOR (XORcc)	Exclusive-Or (and modify *icc*)
XNOR (XNORcc)	Exclusive-Nor (and modify *icc*)
SLL	Shift Left Logical
SRL	Shift Right Logical
SRA	Shift Right Arithmetic

Opcode	*Name*
ADD (ADDcc)	Add (and modify *icc*)
ADDX (ADDXcc)	Add with Carry(and modify *icc*)
TADDcc (TADDccTV)	Tagged Add and modify *icc* (and Trap on overflow)
SUB (SUBcc)	Subtract (and modify *icc*)
SUBX (SUBXcc)	Subtract with Carry (and modify *icc*)
TSUBcc (TSUBccTV)	Tagged Subtract and modify *icc* (and Trap on overflow)
MULScc	Multiply Step and modify *icc*
UMUL (UMULcc)	Unsigned Integer Multiply (and modify *icc*)
SMUL (SMULcc)	Signed Integer Multiply (and modify *icc*)
UDIV (UDIVcc)	Unsigned Integer Divide (and modify *icc*)
SDIV (SDIVcc)	Signed Integer Divide (and modify *icc*)
SAVE	Save caller's window
RESTORE	Restore caller's window
Bicc	Branch on integer condition codes
FBfcc	Branch on floating-point condition codes
CBccc	Branch on coprocessor condition codes
CALL	Call and Link
JMPL	Jump and Link
RETT†	Return from Trap
Ticc	Trap on integer condition codes
RDASR‡	Read Ancillary State Register
RDY	Read Y Register
RDPSR†	Read Processor State Register
RDWIM†	Read Window Invalid Mask Register
RDTBR†	Read Trap Base Register
WRASR‡	Write Ancillary State Register
WRY	Write Y Register
WRPSR†	Write Processor State Register
WRWIM†	Write Window Invalid Mask Register
WRTBR†	Write Trap Base Register
STBAR	Store Barrier
UNIMP	Unimplemented Instruction
FLUSH	Flush Instruction Memory
FPop	Floating-point Operate: see table overleaf
CPop	Coprocessor Operate: implementation-dependent

† privilegierter Befehl
‡ privilegierter Befehl, wenn referenziertes ASR privilegiert ist

FPop-Befehl	*Name*
FiTO(s,d,q)	Floating-point integer To (single, double, quad)
F(s,d,q)TOi	Floating-point (single, double, quad) To integer
FsTOd	Floating-point single To double
FsTOq	Floating-point single To quad
FdTOs	Floating-point double To single
FdTOq	Floating-point double To quad
FqTOs	Floating-point quad To single
FqTOd	Floating-point quad To double
FMOVs	Floating-point Move single
FNEGs	Floating-point Negate single
FABSs	Floating-point Absolute single
FSQRT(s,d,q)	Floating-point Square Root (single, double, quad)
FADD(s,d,q)	Floating-point Add (single, double, quad)
FSUB(s,d,q)	Floating-point Subtract (single, double, quad)
FMUL(s,d,q)	Floating-point Multiply (single, double, quad)
FDIV(s,d,q)	Floating-point Divide (single, double, quad)
FsMULd	Floating-point single Multiply double
FdMULq	Floating-point double Multiply quad
FCMP(s,d,q)	Floating-point Compare (single, double, quad)
FCMPE(s,d,q)	Floating-point Compare with Exception (single, double, quad)

Synthetischer Befehl		*SPARC-Befehl*		*Kommentar*
`cmp`	*reg*, *reg_or_imm*	`subcc`	*reg*, *reg_or_imm*, `%g0`	compare
`jmp`	*address*	`jmpl`	*address*, `%g0`	
`call`	*address*	`jmpl`	*address*, `%o7`	
`tst`	*reg*	`orcc`	`%g0`, *reg*, `%g0`	test
`ret`		`jmpl`	`%i7+8, %g0`	return from subroutine
`retl`		`jmpl`	`%o7+8, %g0`	.. from leaf subroutine
`restore`		`restore`	`%g0, %g0, %g0`	trivial restore
`set`	*value*, *reg*	`sethi`	`%hi(`*value*`)`, *reg*	(*value* & 0x3FF) == 0
		`or`	`%g0`, *value*, *reg*	-4096 $\leq$ *value* $\leq$ 4095
		`sethi`	`%hi(`*value*`)`, *reg*	andernfalls
		`or`	*reg*, `%lo(`*value*`)`, *reg*	Achtung: 2 Befehle!!!
`not`	*reg*	`xnor`	*reg*, `%g0`, *reg*	Einerkomplement
`neg`	*reg*	`sub`	`%g0`, *reg*, *reg*	Zweierkomplement
`inc`	*reg*	`add`	*reg*, `1`, *reg*	increment
`dec`	*reg*	`sub`	*reg*, `1`, *reg*	decrement
`btst`	*reg_or_imm*, *reg*	`andcc`	*reg*, *reg_or_imm*, `%g0`	bit test
`bset`	*reg_or_imm*, *reg*	`or`	*reg*, *reg_or_imm*, *reg*	bit set
`bclr`	*reg_or_imm*, *reg*	`andn`	*reg*, *reg_or_imm*, *reg*	bit clear
`btog`	*reg_or_imm*, *reg*	`xor`	*reg*, *reg_or_imm*, *reg*	bit toggle
`mov`	*reg_or_imm*, *reg*	`or`	`%g0`, *reg_or_imm*, *reg*	move, i.e. copy

B Abkürzungen

ABI	Application Binary Interface
aexc	accrued exception
ASI	Address Space Identifier
ccc	coprocessor condition codes
cexc	current exception
CP	Coprocessor
CPop	Coprocessor operate
CPU	Central Processor Unit
CWP	Current Window Pointer
CTI	Control Transfer Instruction
DCTI	Delayed Control Transfer Instruction
ELF	Executable and Linking Format
ET	Enable Traps
fcc	floating-point condition codes
ftt	floating-point trap type
FPop	Floating-Point operate
FPU	Floating-Point Unit
FSR	Floating-Point State Register
icc	integer condition codes
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronic Engineers
I/O	Input/Output
ISA	Instruction Set Architecture
ISO	International Organization for Standardization (keine Abk.!)
IU	Integer Unit
MMU	Memory Management Unit
NaN	Not a Number
nPC	next Program Counter
OS	Operating System
PC	Program Counter
PIL	Processor Interrupt Level
PSR	Processor State Register
qne	(floating-point deferred trap) queue not empty
RD	Rounding (mode of the FPU)
SCD	SPARC Compliance Definition
SPARC	Scalable Processor Architecture
TEM	Trap Enable Mask
TBA	Trap Base Address
TBR	Trap Base Register
tt	trap type
VIS	Visual Instruction Set
WIM	Window Invalid Mask

Literatur

[Cypress90] Ross Technology, Inc., Cypress Semiconductor Corp., SPARC RISC User's Guide, 2nd Edition, Austin, 1990
Herstellerhandbuch einer Multichip-Implementation der v7-Achitektur.

[Dewar90] Dewar,Robert B.K., and Matthew Smosna, Microprocessors: A Programmer's View, McGraw-Hill, 1990
Einführung und Vergleiche repräsentativer Architekturen (CISC: 80386, 80387, 68030. RISC: MIPS, SPARC, i860, IBM POWER, INMOS Transputer).

[Dobb92] Dobberpuhl, Daniel W., et alii, A 200-MHz 64-bit Dual-issue CMOS Microprocessor, Digital Technical Journal, Vol.4 No.4, Special Issue 1992, p.35 ff.
Nachdruck aus dem IEEE Journal of Solid-State Circuits*, vol. 27, no. 11, pp. 1555-1567, Nov. 1992. Beschreibt die HW-Implementation des 64-Bit DEC Alpha 21064, des ersten Chips dieser Reihe mit damals sensationellen 200 MHz.*

[Fuji92] Fujitsu Inc., MB86930 - SPARClite User's Manual, Section 1, 1992
Spezialversion einer SPARC v8 Implementation für eingebettete Anwendung. Integer-Division und Fließkommaeinheit werden in Software emuliert.

[Gil93] Giloi, Wolfgang K., Rechnerarchitektur, 2. Auflage, Springer Verlag, 1993 – *Der Autor war u.a. Architekt des Superrechners SUPRENUM.*

[HAL98] HAL Computer Systems, Inc., A Fujitsu Company, SPARC64-III User's Guide, Campbell, California, 1998
Beschreibung der SPARC v9 Implementation von Fujitsu mit 4 parallelen Ausführungseinheiten in der IU. Sehr gut geeignet als Ergänzung zu [SPARC95], zeigt dieses Handbuch die Entwicklung einer konkreten Implementation aus der abstrakten Architekturbeschreibung.

[Harb95] Harbison, Samuel P., and Guy L. Steele Jr., C, A Reference Manual, 4th edition, Prentice-Hall, 1995
Renommiertes Standardwerk zu C, enthält schon die Erweiterungen von C94.

[ISO99] International Organization for Standardization (ISO), ISO/IEC Standard 9899:1999, Genf, 1999
Der definitive Text des C-Standards, inzwischen auf dem Entwicklungsstand C99. Nur zu bestellen bei der entspr. Standardorganisation.

[Kern89] Kernighan,Brian W., and Dennis M. Ritchie, The C Programming Language, Prentice-Hall, 1989
Das Einführungsbuch. D.M.Ritchie ist der Autor der Sprache C. Das Buch beschreibt ANSI-C 89 und gibt ein Gefühl für das Flair und die Eleganz dieser heute wohl am weitesten verbreiteten Systemsprache.

[Knuth99] Knuth, Donald E., The Art of Computer Programming, Fascicle 1, MMIX, Addison-Wesley, 1999
Beschreibung eines für didaktische Zwecke entwickelten fiktiven RISC-Microprozessors zur Arbeit mit den Beispielen in seinen Grundlagenwerken.

[Marg90] Margulis, Neal, i860 Microprocessor Architecture, Intel/Osborne/McGraw-Hill, 1990
Neal Margulis war Chef-Anwendungsingenieur für das i860-Projekt bei Intel.

[Plau95] Plauger, P.J., and Jim Brodie, Standard C, A Reference, Prentice-Hall, 1995
Jim Brodie (von Motorola) veranlaßte die Inangriffnahme der Standardisierung der Sprache C beim ANSI, P.J.Plauger leitete den Ausschuß für die Standardisierung der C-Bibliothek.

[Sites92] Sites, Richard L., Alpha AXP Architecture, Digital Technical Journal, Vol.4 No.4, Special Issue 1992, p.19 ff.
Sites ist Architekt für den Alpha (mit Richard Witek). Die autoritative frühe Beschreibung der Alpha-Architektur mit Erläuterung der Entwicklungsschritte und Vorstellung des Programmiermodells.

[SPARC92] SPARC International, Inc., The SPARC Architecture Manual, Version 8, Menlo Parc, 1992
Das dem ersten Teil dieser Einführung zugrundeliegende Hauptwerk.

[SPARC95] SPARC International, Inc., The SPARC Architecture Manual, Version 9, Menlo Parc, 1994-2000
Die Weiterentwicklung und Erweiterung der SPARC v8 auf 64-Bit.

[SPARC00] Sun Microsystems, Inc., SPARC Assembly Language Reference Manual, Palo Alto, 2000
Beschreibt den Assembler mit Aufrufoptionen, Syntax und Dateiformaten sowohl für v8, als auch v9, mit diversen Zwischenstufen. Suns v9-Erweiterungen, wie UltraSPARC und VIS, sind auch enthalten.

[Stev92] Stevens, W. Richard, Advanced Programming in the UNIX Environment, Addison-Wesley, 1992
Das Standardwerk zur systemnahen Programmierung unter allen Unix-Derivaten. Präzise und umfassend.

[Weiss94] Weiss, Shlomo, and James E. Smith, POWER and PowerPC, Morgan-Kaufman, 1994
Eine weitere hochinteressante RISC-Architektur. Enthält am Schluß einen interessanten Vergleich mit der Alpha-Architektur, sowie einen Anhang mit der Beschreibung des Standards IEEE-754.

Index