3 Modul IP7.1
4 Modul IP7.3

5 Grundzüge der SPARC-Architektur V8

5.1 Überblick

5.2 Zwei einfache Assembler-Programme

5.3 Zur Assembler-Sprache

5.4 Arithmetische Befehle

5.5 Logische Befehle

5.6 Schiebebefehle

5.7 Sprungbefehle

5.8 Lade- und Speicherbefehle

5.9 Aufruf von Unterprogrammen

5.10 Synthetische Instruktionen

5.11 Gleitpunktrechnung

Überblick über SPARC-Architektur:

1. **SPARC = Scalable Processor Architecture**

2. **Definierendes Dokument:**

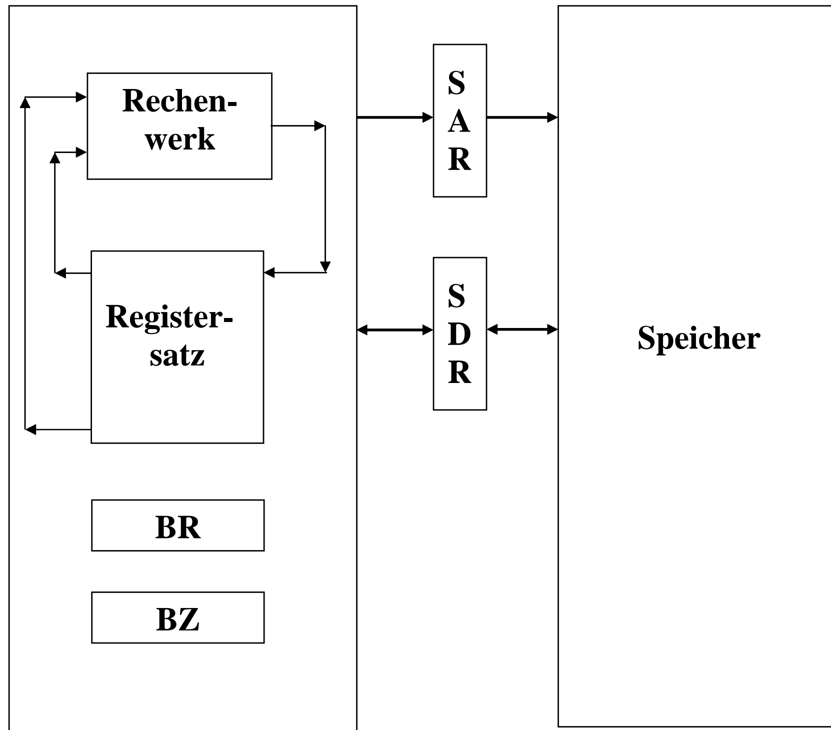
   D. Weaver, T. Germond (eds): The SPARC architecture manual, version 8,
   SPARC International, Inc.
   Englewood Cliffs, N. J., Prentice Hall, 1992
   ISBN-10: 0-13-825001-4

3. **RISC-Architektur:**

   Kleine Anzahl an Befehlen,
   wenige Adreßformate,
   "Load/Store"-Architektur,
   große Anzahl an Registern.

Bemerkung: **RISC ist die Abkürzung von Reduced Instruction Set Computer, dies kann man auf zwei Arten interpretieren. "Load /Store"-Architektur bedeutet, daß die Dateneingabe für das Rechenwerk nur aus Registern der Zentraleinheit erfolgt.**

Schaubild einer Maschine in "Load/Store"-Architektur:



Legende:

SAR	=	**Speicheradreßregister**
SDR	=	**Speicherdatenregister**
BZ	=	**Befehlszähler**
BR	=	**Befehlsregister**

4. Befehlsformate:

**Es existieren nur vier Befehlsformate.
Die meisten Befehle sind Drei-Operanden-Befehle, in Assembler-Syntax:**

```
add     regsc1,  regsc2,  regdn
add     regsc1,  imm13,  regdn
```

5. Fünf Datenformate:

Byte	**8**	**Bit**
Halbwort	**16**	**Bit**
Wort	**32**	**Bit**
Doppelwort	**64**	**Bit**
Vierfachwort	**128**	**Bit**

6. Register:

8 globale Register:	**%g0 . . . %g7**
8 "output" Register:	**%o0 . . . %o7**
8 lokale Register:	**%l0 . . . %l7**
8 "input" Register:	**%i0 . . . %i7**

Bemerkung: **Die o-, l- und i-Register sind Bestandteil des lokalen Registerfensters. Das Register %g0 repräsentiert die Konstante Null. Daneben existieren noch Hilfsregister zur Ausführung ausgewählter Befehle und zustandsbeschreibende Register wie das Maschinenstatuswort.**

7. Adressierungsmodi:

Eine Adresse wird gemäß einer der folgenden Regeln gebildet:

- Register,
- Register + Register,
- Register ± Konstante,
- Konstante.

Bemerkung: Jeder Befehl auf der Sparc wird in 32 Bit codiert. Je nach Befehl stehen für obige Konstante 30 Bit, 22 Bit oder 13 Bit zur Verfügung.

Beispiele:

```
ld      [%r16],  %r9
ld      [%r16 + %r17],  %r9
ld      [%r16 + 24],  %r9
```

8. Befehlszähler:

Die Sparc verfügt über zwei Befehlszähler, daher ist eine verzögerte Ausführung von Sprungbefehlen möglich.

9. Der Befehl sethi:

Syntax:

```
sethi   const22,  reg
sethi   %hi(value),  reg
```

Semantik: Der Befehl sethi nulliert die 10 niederwertigsten Bit und ersetzt die 22 höchstwertigen Bit durch die angegebene Konstante.

```
!  5.2a  ! Ein einfaches Assemblerprogramm

        ! Informationen fuer Binder
        .global   main
        .global   printf

        ! Deklaration von Daten
        .section  ".data"
a2:     .word     41
a1:     .half     17
fmt:    .asciz    "Ergebnis:  %d * %d  = %d\n"

        ! Ausfuehrbares Programm
        .section  ".text"
main:   set       a1,  %l1
        ldsh      [%l1],  %l1
        set       a2,  %l2
        ld        [%l2],  %l2
        set       fmt,  %o0          ! Erster Operand
                                     ! in %o0
        smul      %l1,  %l2,  %o3    ! Vierter Operand
                                     ! in %o3
        mov       %l1,  %o1
        mov       %l2,  %o2
        call      printf
        ! Es wird die C-Routine printf aus stdio.h
        ! aufgerufen.  Eine äquivalente Java-Routine
        ! printf findet man in java.io.PrintStream.
        nop

        mov       1,  %g1   ! Rueckkehr ins Betriebssytem
        ta        0

        ! Aufruf mit           cc  einfach.s
        ! Ausfuehren mit       a.out
        ! Ausgabe:             Ergebnis:  17 * 41  = 697
```

```
! 5.2b    !  Beispielprogramm zur Gleitpunktrechnung

          !  Informationen fuer Binder
          .global   main
          .global   printf

          !  Deklaration von Daten
          .section  ".data"
b1:       .single        0r3.050502
b2:       .single        0r0.202020
          .double        0r0
fmt:      .asciz    "Ergebnis = %g\n"   ! Gleitpunktformat

          !  Ausfuehrbares Programm
          .section  ".text"
main:     sethi     %hi(b1), %l1
          or        %l1, %lo(b1), %l1
          ld        [%l1], %f1
          ld        [%l1+4], %f2
          fadds     %f1, %f2, %f3
          fstod     %f3, %f4              !  Zahlwandlung
          std       %f4, [%l1+8]   ! 2 Register gespeichert.
          ld        [%l1+8], %o1          !  Besonderheit
          ld        [%l1+12], %o2         !  des C-Systems
          set       fmt,  %o0             !  auf der Sparc.
          call      printf
          nop
          !  Rueckkehr zum Betriebssystem
          mov       1,  %g1
          ta        0

          !  Ausgabe:
          !  Ergebnis  =  3.25252
          !  Taschenrechner  =  3.252522,  warum ?
```

5.3 Zur Assembler-Sprache

Einige Bemerkungen:

(i) Assemblersprachen sind einfache zeilenorientierte Sprachen. Ein Assemblerprogramm besteht aus Direktiven, Marken, Befehlen und Kommentaren; es ist in Segmente gegliedert.

(ii) Format einer Assemblerzeile:

```
Marke:    Befehl       !Kommentar
          Direktive    Parameter
```

(iii) Beispiele für Kommentare:

```
!  Dies ist ein Kommentar
/* Auch dies ist ein Kommentar */
```

(iv) Marken dienen der Kennzeichnung von Speicher-adressen; eine Marke beschreibt eine Variable, einen Abschnitt oder ein Sprungziel. Marken werden durch Symbolbezeichner, gefolgt von einem Doppelpunkt, eingeführt. Eine besondere Rolle als Programmstart spielt in C-Systemen die Marke main.

(v) Direktiven stellen Organisationsanweisungen für den Assembler dar. Beispiele sind: .word, .section, .global.

(vi) Beispiele elementarer Direktiven:

```
x:    .word     0x3fce     !  32 Bit Größe
y:    .half     0x1f3      !  16 Bit Größe
z:    .byte     0x1f       !   8 Bit Größe
a:    .single   0r3.2      !  32 Bit Größe
b:    .double   0r2E10     !  64 Bit Größe
c:    .quad     0r1.414    ! 128 Bit Größe
d:    .skip     n          ! reserviert n Byte

! Erzeugung einer Zeichenkette von
! ASCII-Zeichen.
a:    .ascii    “Hallo”

! Erzeugung einer nullterminierten Zeichenkette
! von ASCII-Zeichen.
b:    .asciz    "Hallo“

! Markiert ein Symbol als global sichtbar.
      .global   alpha

      .section  ".bss"
! BSS =  Block Storage Segment oder
!        Block Started by Symbol
! Speicherbereich für nicht initialisierte globale
! Variablen.

      .section  ".data"
      .section  ".rodata"
      .section  ".init"
      .section  ".fini"
      .section  ".text"

      .align    Grenze
! Grenze muß eine Zweierpotenz sein.
```

(vii) Innerhalb des Assemblers lassen sich Ganzzahlausdrücke berechnen. Es stehen dafür die folgenden Operatoren zur Verfügung:

+	**(binäre Addition),**
–	**(binäre Subtraktion),**
*****	**(Multiplikation),**
/	**(Division),**
%	**(Modulo-Operation),**
<<	**(Links-Schieben),**
>>	**(Rechts-Schieben),**
^	**(Bitweises XOR),**
&	**(Bitweises UND),**
|	**(Bitweises ODER),**
+	**(unäres Plus),**
–	**(unäres Minus),**
~	**(Bitweises NICHT).**
%lo	**niederwertige 10 Bit**
%hi	**höchstwertige 22 Bit**

Beispiele:

```
–1   <<  6     =  –64
–64  >>  28    =   15
264  %  10     =    4
264  %  –10    =    4
–264  %  10    =   –4
–264  %  –10   =   –4
~21            =  4294967274
%lo (~21)      =  1002
%hi (~21)      =  4194303
```

(viii) Für Ausdrücke lassen sich Abkürzungen einführen: Symbol = Ausdruck.

5.4 Arithmetische Befehle

Die arithmetischen Befehle werden auf der Sparc in zwei Varianten angeboten, einmal ohne Ergebnisanzeigen und einmal mit Ergebnisanzeigen.

Liste (ohne Befehle für kennzeichenbehaftete Daten):

```
add      rs1,  rs2/imm13,   rd
addcc    rs1,  rs2/imm13,   rd
addx     rs1,  rs2/imm13,   rd
addxcc   rs1,  rs2/imm13,   rd
sub      rs1,  rs2/imm13,   rd
subcc    rs1,  rs2/imm13,   rd
subx     rs1,  rs2/imm13,   rd
subxcc   rs1,  rs2/imm13,   rd
mulscc   rs1,  rs2/imm13,   rd
umul     rs1,  rs2/imm13,   rd
umulcc   rs1,  rs2/imm13,   rd
smul     rs1,  rs2/imm13,   rd
smulcc   rs1,  rs2/imm13,   rd
udiv     rs1,  rs2/imm13,   rd
udivcc   rs1,  rs2/imm13,   rd
sdiv     rs1,  rs2/imm13,   rd
sdivcc   rs1,  rs2/imm13,   rd
```

Man kennt vier Ergebnisanzeigen:

Negativ, Null, Übertrag, Überlauf

Bedingungsanzeigen:

Die Sparc V8 kennt vier Bedingungsanzeigen für Ganzzahlrechnungen. Sie werden im Prozessorstatuswort im Bitfeld 20 .. 23 gespeichert. Die Bedingungsanzeigen werden durch einen Teil der Befehle gesetzt, in Assembler-Notation tragen diese den Zusatz cc.

Negativ-Anzeige (n):

- **n = 1, falls das Ergebnis der letzten berücksichtigten Operation negativ war.**
- **n = 0, falls das Ergebnis der letzten berücksichtigten Operation nicht negativ war.**

Null-Anzeige (z):

- **z = 1, falls das Ergebnis der letzten berücksichtigten Operation 0 war.**
- **z = 0, falls das Ergebnis der letzten berücksichtigten Operation nicht 0 war.**

Überlauf-Anzeige (v):

- **v = 1, falls das Ergebnis der letzten berücksichtigten Operation nicht darstellbar war.**
- **v = 0, falls das Ergebnis der letzten berücksichtigten Operation darstellbar war.**

Übertrag-Anzeige (c):

- **c = 1, falls bei der letzten berücksichtigten Operation ein Übertrag gemäß Zweierkomplementrechnung auftrat.**
- **c = 0, falls bei der letzten berücksichtigten Operation kein Übertrag auftrat.**

Beispiele zum Additionsbefehl addcc:

dezimal: 111 + 666 = 777
sedezimal: 6f + 29a = 309
Gesetzte Anzeigen: keine

dezimal: 0 + 0 = 0
sedezimal: 0 + 0 = 0
Gesetzte Anzeigen: Z

dezimal: 111 + 4294967184 = 4294967295
sedezimal: 6f + ffffff90 = ffffffff
Gesetzte Anzeigen: N

dezimal: 2147483997 + 2147483650 = 351
sedezimal: 8000015d + 80000002 = 15f
Gesetzte Anzeigen: C V

dezimal: 4294967295 + 6 = 5
sedezimal: ffffffff + 6 = 5
Gesetzte Anzeigen: C

dezimal: 4294967185 + 111 = 0
sedezimal: ffffff91 + 6f = 0
Gesetzte Anzeigen: Z C

dezimal: 2147483644 + 12 = 2147483656
sedezimal: 7ffffffc + c = 80000008
Gesetzte Anzeigen: N V

Ganzzahlmultiplikation:

Die Sparc V8 kennt zwei Multiplikationsbefehle:

Multiplikation vorzeichenloser Zahlen,
Multiplikation vorzeichenbehafteter Zahlen,
in beiden Fällen werden 32-Bit-Zahlen verarbeitet.

Syntax:

```
umul   rs1, rs2,    rd
umul   rs1, imm13,  rd
smul   rs1, rs2,    rd
smul   rs1, imm13,  rd
```

Bemerkungen:

(i) **Das Multiplikationsergebnis ist 64 Bit lang, die höherwertigen Bit befinden sich im Y-Register, die niederwertigen im Register rd.**

(ii) **Kontrolle eines Überlaufs bei umul:**
$Y \neq 0$.

(iii) **Kontrolle eines Überlaufs bei smul:**
$Y \neq$ ((rd) arith>> 31).

(iv) **Die Varianten**
umulcc rs1, rs2/imm13, rd und
smulcc rs1, rs2/imm13, rd
setzen die Anzeigen Z und N nur nach den niederen 32 Bit des Ergebnisses, die Anzeigen C und V sind 0.

Ganzzahldivision:

Die Sparc V8 kennt zwei Divisionsbefehle:

Division vorzeichenloser Zahlen,
Division vorzeichenbehafteter Zahlen,
in beiden Fällen ist der Dividend eine 64-Bit-Zahl und der Divisor eine 32-Bit-Zahl. Der Quotient ist immer eine 32-Bit-Zahl, der Rest muß nicht bestimmt werden, Rundung des Quotienten erfolgt in Richtung Null, bei Überlauf werden die Grenzwerte geliefert.

Syntax:

```
udiv    rs1,  rs2,     rd
udiv    rs1,  imm13,   rd
sdiv    rs1,  rs2,     rd
sdiv    rs1,  imm13,   rd
```

Bemerkung:

(i) **Die 64-Bit-Kette, die sich als Verkettung der Inhalte des Y-Registers und des rs1-Registers ergibt, ist der Dividend. Der Quotient wird im Register rd abgelegt. Implementationsabhängig kann das Y-Register den Divisionsrest enthalten.**

(ii) **Die Varianten**

```
udivcc  rs1,  rs2/imm13,  rd  und
sdivcc  rs1,  rs2/imm13,  rd
```

setzen die Anzeigen Z und N gemäß den 32 Bit des Quotienten, die Anzeige V wird bei Ergebnisüberlauf gesetzt, die Anzeige C ist 0.

Lesen und Schreiben des Y-Registers:

Syntax für Lesebefehl:

```
rd      %y,  rd
```

Syntax für Schreibbefehl:

```
wr      rs1,  rs2,     %y
wr      rs1,  imm13,   %y
```

Semantik des Schreibbefehls:

%y := (rs1) xor (rs2)
oder
%y := (rs1) xor (sign-extended imm13)

Bemerkung:

Das Schreiben des Y-Registers erfolgt verzögert. Zwischen einem Schreiben in das und einem nachfolgenden Lesen aus dem Y-Register sollten n (n ist implementationsabhängig, oft 3) Befehle liegen. Die Befehle sdiv und udiv lesen das Y-Register.

5.5 Logische Befehle

not	rs1, rd
not	rd

and	rs1, rs2/simm13, rd
andcc	rs1, rs2/simm13, rd
andn	rs1, rs2/simm13, rd
andncc	rs1, rs2/simm13, rd

or	rs1, rs2/simm13, rd
orcc	rs1, rs2/simm13, rd
orn	rs1, rs2/simm13, rd
orncc	rs1, rs2/simm13, rd

xor	rs1, rs2/simm13, rd
xorcc	rs1, rs2/simm13, rd
xnor	rs1, rs2/simm13, rd
xnorcc	rs1, rs2/simm13, rd

Bemerkung: Die cc-Varianten setzen nur die Anzeigen N und Z, not ist eine synthetische Instruktion.

Logische Operationen auf der Sparc:

a b	0 0 1 1 0 1 0 1	Logische Operation	Sparc-Befehl
0	0 0 0 0	false	
1	0 0 0 1	a and b	and
2	0 0 1 0	a and (not b)	andn
3	0 0 1 1	a	
4	0 1 0 0	b and (not a)	
5	0 1 0 1	b	
6	0 1 1 0	a xor b	xor
7	0 1 1 1	a or b	or
8	1 0 0 0	a nor b	
9	1 0 0 1	a xor (not b)	xnor
10	1 0 1 0	not b	
11	1 0 1 1	a or (not b)	orn
12	1 1 0 0	not a	
13	1 1 0 1	b or (not a)	
14	1 1 1 0	a nand b	
15	1 1 1 1	true	

Bemerkung: Nur die Funktionen nand und nor bietet die Sparc nicht.

5.6 Schiebebefehle

Die V8-Architektur kennt 3 Schiebebefehle:

SLL	**rs1, rs2/imm13, rd**
SRL	**rs1, rs2/imm13, rd**
SRA	**rs1, rs2/imm13, rd**

Bemerkungen:

(i) **Die Schiebebefehle setzen die Bedingungsanzeigen nicht.**

(ii) **Die Schiebebefehle nutzen nur die untersten 5 Bit der Schiebezahl.**

(iii) **Verschieben um 0 Positionen ist zulässig.**

(iv) **Die Befehle SLL und SRL füllen die freiwerdenden Positionen mit Nullen, der Befehl SRA füllt die freiwerdenden Positionen mit dem höchstwertigen Bit von rs1.**

5.7 Sprungbefehle

Bemerkung: **Der Jump-Befehl und seine Verwandten werden in 5.9 besprochen.**

Bedingte Sprungbefehle der Sparc V8:

Kürzel	Bezeichnung	Bedingung
BA	Branch Always	true
BN	Branch Never	false
BNZ	Branch on Not Zero	not Z
BZ	Branch on Zero	Z
BG	Branch on Greater	not (Z or (N xor V))
BLE	Branch on Less or Equal	Z or (N xor V)
BGE	Branch on Greater or Equal	not (N xor V)
BL	Branch on Less	N xor V
BGU	Branch on Greater Unsigned	not (C or Z)
BLEU	Branch on Less or Equal Unsigned	C or Z
BCC	Branch on Carry Clear	not C
BCS	Branch on Carry Set	C
BPOS	Branch on Positive	not N
BNEG	Branch on Negative	N
BVC	Branch on Overflow Clear	not V
BVS	Branch on Overflow Set	V

Sprungbefehle werden oft unmittelbar nach Vergleichen eingesetzt.

Bedingte Sprungbefehle nach Vergleich vorzeichen-behafteter Größen:

Kürzel	Bezeichnung	Bedingung
BE	Branch on Equal	Z
BNE	Branch on Not Equal	not Z
BL	Branch on Less	N xor V
BLE	Branch on Less or Equal	Z or (N xor V)
BGE	Branch on Greater or Equal	not (N xor V)
BG	Branch on Greater	not (Z or (N xor V))

Bedingte Sprungbefehle nach Vergleich vorzeichenloser Größen:

Kürzel	Bezeichnung	Bedingung
BE	Branch on Equal	Z
BNE	Branch on Not Equal	not Z
BLU	Branch on Less Unsigned	C
BLEU	Branch on Less or Equal Unsigned	C or Z
BGEU	Branch on Greater or Equal Unsigned	not C
BGU	Branch on Greater Unsigned	not (C or Z)

Sprungbefehle benötigt man bei der Übersetzung von Schleifen und Fallunterscheidungen aus Hochsprachen in Maschinencode.

Beispiel:

```
int  a = 8;
int  c = 0;
while  (a <= 12) {
        a = a + 2;          //  Anweisung 1
        c = c + 1;          //  Anweisung 2
}
```

Ein einfaches Übersetzungsschema führt zu:

```
            mov     8,      %i5
            mov     %g0,  %l0

            !  Überprüfen der Schleifenbedingung
            cmp     %i5,  12
            bg      sende
            nop
schleife:
            add     %i5,  2,  %i5       ! Anweisung 1
            add     %l0,  1,  %l0       ! Anweisung 2
            !  Überprüfen auf Schleifenende
            cmp     %i5,  12
            ble     schleife
            nop
sende:
```

Sucht man nach Vereinfachungen für den Maschinencode, dann sieht man sofort, daß der Vergleich vor dem ersten Sprung eingespart werden kann.

```
            ba      sende
            nop
schleife:   add     %i5,  2,  %i5      !  Anweisung 1
            add     %l0,  1,  %l0      !  Anweisung 2
sende:      cmp     %i5,  12
            ble     schleife
            nop
```

Ein spezielles Problem der Sparc-Architektur ist das Füllen der "Delay-Slots." Der erste nop-Befehl kann durch den cmp-Befehl ersetzt werden.

```
            ba  sende
            cmp     %i5,  12
schleife:   add     %i5,  2,  %i5      !  Anweisung 1
            add     %l0,  1  ,%l0      !  Anweisung 2
            cmp     %i5,  12
sende:      ble     schleife
            nop
```

Der zweite nop-Befehl, der innerhalb einer Schleife ausgeführt wird, kann ebenfalls sinnvoll ersetzt werden. Man nutzt hier eine Eigenschaft der Sparc, die man Annulling nennt.

```
            ba      sende
            cmp     %i5,  12
schleife:   add     %i5,  2,  %i5      !  Anweisung 1
            cmp     %i5,  12
sende:      ble,a   schleife
            add     %l0,  1,  %l0      !  Anweisung 2
```

Der Befehl nach dem bedingten Sprung wird genau dann ausgeführt, falls der bedingte Sprung ausgeführt wird. In Assembler-Notation wird dies durch ",a" angegeben. Bei unbedingten Sprüngen ist die Bedeutung des Annulling entgegengesetzt.

Sind die Steuerausdrücke für Schleifen oder Fallunterscheidungen zusammengesetzt, dann benutzt man häufig Sprungfolgen, die bedeutend schwieriger zu verbessern sind.

```
Beispiel:   while  ((a>3) && (c < 15))  {
                   /*  Schleifenkörper  */  }
            cmp     %i5,  3
            ble     sende
            nop
            cmp     %l0,  15
            bge     sende
            nop
schleife:   !  Schleifenkörper
            cmp     %i5,  3
            ble     sende
            nop
            cmp     %l0,  15
            bl      schleife
sende:
```

5.8 Ladebefehle und Speicherbefehle

Ladebefehle der Ganzzahleinheit:

ldsb **load signed byte - rechts ausgerichtet und vorzeichengerecht erweitert.**
```
ldsb   [address], rd
```

ldsh **load signed halfword - rechts ausgerichtet und vorzeichengerecht erweitert.**
```
ldsh   [address], rd
```

ldub **load unsigned byte - rechts ausgerichtet und mit Nullen erweitert.**
```
ldub   [address], rd
```

lduh **load unsigned halfword - rechts ausgerichtet und mit Nullen erweitert.**
```
lduh   [address], rd
```

ld **load word**
```
ld     [address], rd
```

ldd **load doubleword - lade in gerades-ungerades Registerpaar.**
```
ldd    [address], rd
```

sethi **Ersetze die 22 höchstwertigen Bit durch den Operandenwert , die 10 niederwertigen Bit durch 0.**
```
sethi  immediate22, rd
```

rdy **Lese den Inhalt des Y-Registers.**
```
rd     %y, rd
```

Speicherbefehle der Ganzzahleinheit:

stb **store byte**
```
stb    rs, [address]
```

sth **store halfword**
```
sth    rs, [address]
```

st **store word**
```
st     rs, [address]
```

std **store doubleword from an even-odd register pair**
```
std    rs, [address]
```

wry **Ersetze den Inhalt des Y-Registers durch rs xor op2.**
```
wr     rs, op2, %y
```

5.9 Aufruf von Unterprogrammen

Aufgaben beim Aufruf einer Routine:

Im Hauptprogramm:

⋮
Bereitstellung der Aufrufparameter,
Aufruf der Routine,
Übernahme der Rückgabewerte.
⋮

Im Unterprogramm:

Retten der Rücksprungadresse,
Besorgen lokalen Speicherplatzes,
Übernahme der Aufrufparameter.
⋮

Text für Routinenaufgabe

⋮
Bereitstellen der Rückgabewerte,
Rückkehr zum Aufrufer.

Bemerkung: Die Verteilung der Aufgaben zwischen Hauptprogramm und Unterprogramm kann auch anders erfolgen.

Konventionen der SPARC-Nutzung:

Bei jedem Unterprogrammaufruf wird dem Unterprogramm ein Satz von 8 lokalen Registern zur beliebigen Nutzung zur Verfügung gestellt. Die Registerinhalte bleiben über innere Prozeduraufrufe erhalten.

Normaler Aufruf von Routinen:

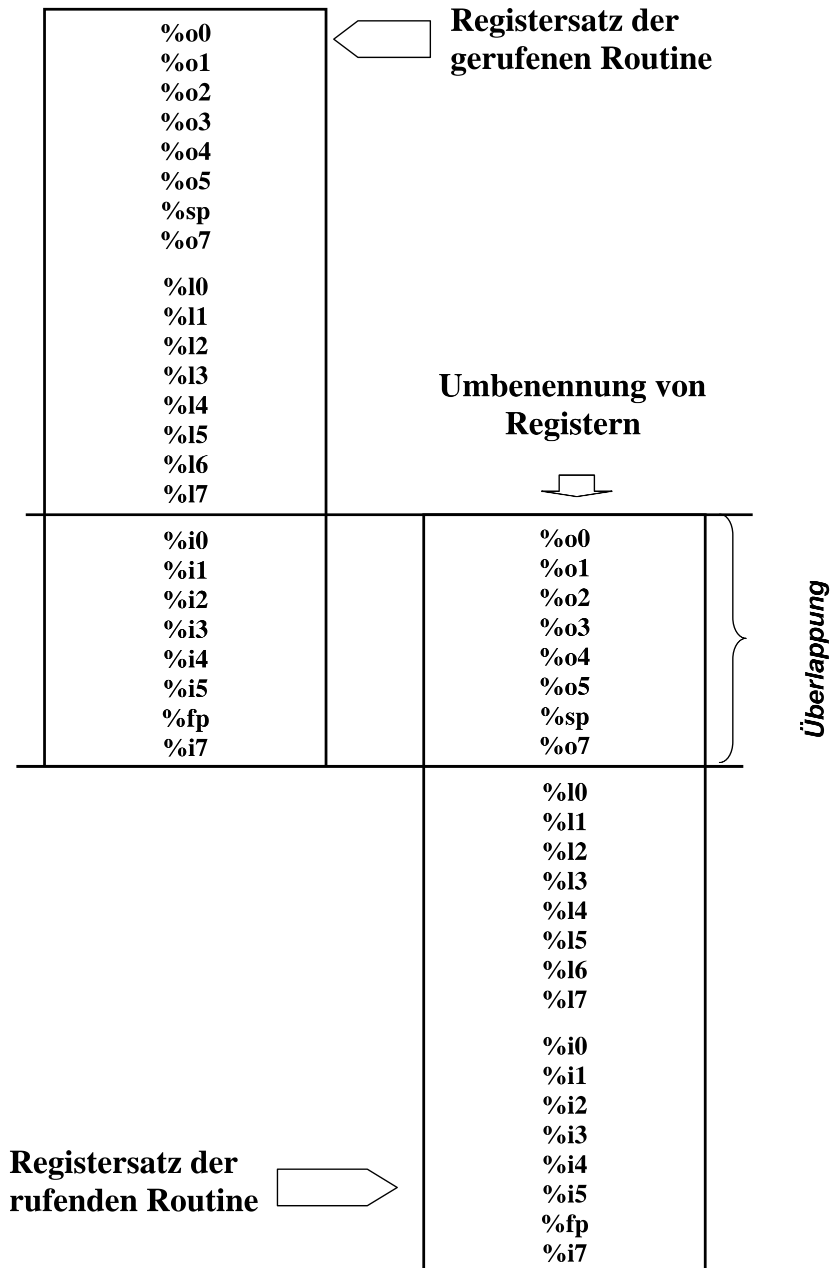
```
        call    routine
        nop                     !  Ziel:  sinnvolle Nutzung
```

Normaler Prolog und Epilog für Routinen:

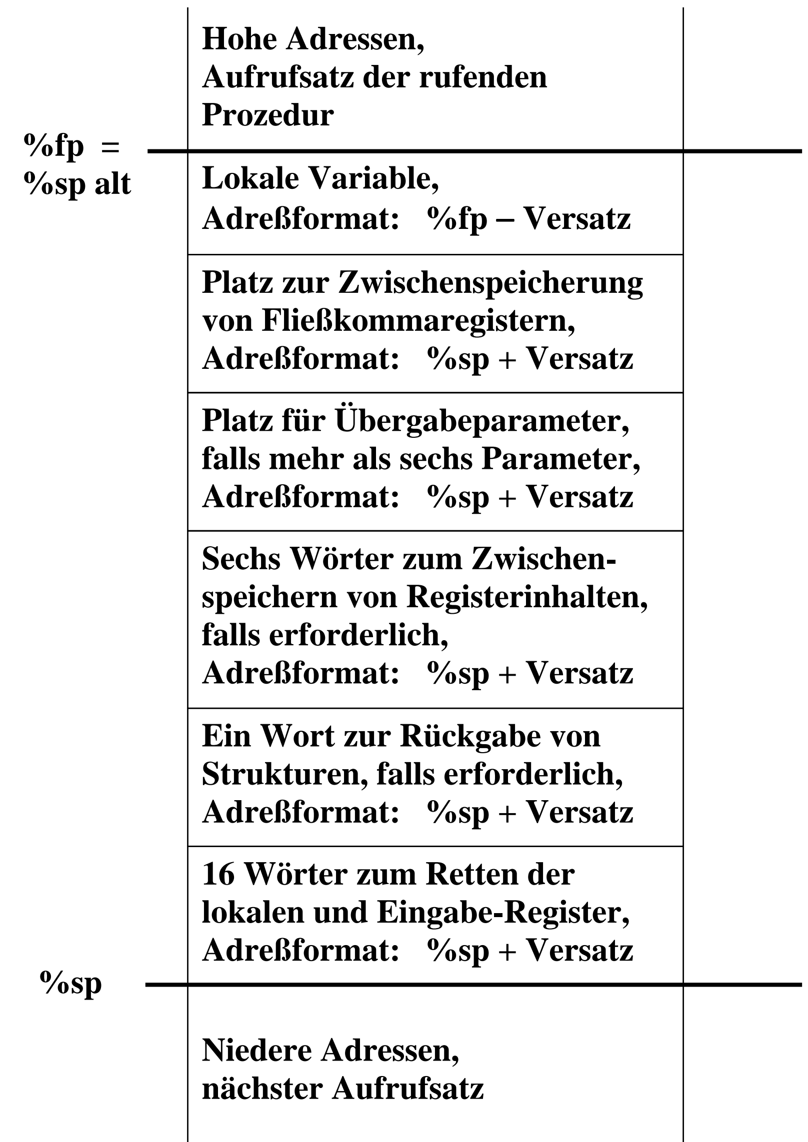
```
routine:
        save    %sp,   -C,   %sp

                ! Konstante C steht für benötigten
                ! Speicherplatz,  64 ist der Minimal-
                ! wert,  ferner sollte C  durch acht
                ! ohne Rest teilbar sein.
                ! Ausführen der Aufgabe,
                ! Ergebnis in Register  %i0
                ! speichern.

        ret         ! jmpl      %i7+8,   %g0
        restore     ! restore   %g0,  %g0,  %g0
```



Aufbau eines Aufrufsatzes:



Bemerkung: Sinnvoll ist eine Mindestgröße von 96 Byte.

Save, Restore, Jmpl (Jump and Link):

Assemblernotation:

```
save      %rs1, %rs2/imm13, %rd
restore   %rs1, %rs2/imm13, %rd
jmpl      %rs1, %rs2/imm13, %rd
```

Bemerkung: Der Befehl save stellt ein neues Registerfenster zur Verfügung. Gleichzeitig führt er eine Addition durch: %rd = (%rs1) + (%rs2)/imm13. Hierbei stammen die Register %rs1 und %rs2 aus dem aktuellen Registerfenster, falls zu einem Registerfenster zugehörig, und das Register %rd aus dem neuen Fenster. Eingesetzt wird dieser Befehl, um Speicherplatz auf einem Aufrufkeller zur Verfügung zu stellen. Der Befehl restore kehrt zum Registerfenster vor dem letzten Aufruf von save zurück. Ansonsten führt er wie save auch eine Addition durch. Benutzt wird dieser Befehl meistens nur in der trivialen Form restore %g0, %g0, %g0. Es sei darauf hingewiesen, daß sowohl save als auch restore eine Ausnahmebehandlung auslösen können, in diesem Fall sollte feststehen, wo die zwischenzeitlich anders genutzten Register zwischengespeichert werden. Der Befehl jmpl speichert seine Adresse im Register %rd und verzweigt zur Speicherstelle (%rs1) + (%rs2)/imm13.

```
!       Beispiel eines Unterprogramms
!       Addition dreier Zahlen

        .section   ".text"
        .align     8
        .global    add3
        .global    main
        .global    printf
!       add3 (x, y, z)  =  x + y + z
add3:
        !  Neues Registerfensters
        !  Anlegen eines Aufrufsatzes
        !  Uebernahme der Parameter aus %i0, %i1, %i2
        save      %sp,  -96,  %sp
        add       %i0,  %i1,  %o1
        add       %o1,  %i2,  %i0
        !  Wertrueckgabe in %i0
        jmp       %i7+8              ! Ruecksprung
        restore

main:

        !  Vorbereitung der Parameteruebergabe
        !  add3 (4, 5, 6)
        mov       4,  %o0
        mov       5,  %o1
        call      add3
        mov       6,  %o2
        mov       %o0,  %o1
        !  Ausdruck des Ergebnisses
        set       fmta,  %l0
        call      printf
        !  Nutzung des Sprungfensters
        mov       %l0,  %o0
```

```
        !  Vorbereitung eines zweiten Aufrufs von add3
        !  add3 (7, 8, 9);

        mov     7,  %o0
        mov     8,  %o1
        call    add3
        !  Der folgende Befehl wird vor Ausfuehrung von
        !  add3 ausgefuehrt.
        mov     9,  %o2
        mov     %o0,  %o1


        call    printf
        add     %l0,  8,  %o0

        !  Rueckkehr zum Betriebssystem
        mov     1,  %g1
        ta      0


!  Anlage Datenbereich für Nurlesedaten
        .section  ".rodata1"
        .align    4
fmta:
        .ascii    "a = %d\n\000"        !  \000 ist Oktal 0.
        .align    4
fmtb:
        .ascii    "b = %d\n\000"

!       Ergebnis
!       a = 15
!       b = 24
```

```
!   Rekursive Berechnung der Fakultät
        .global   fakul
        .global   main
        .global   printf

        .section  ".text"
        .align    8
fakul:
!  Normaler Prozedurbeginn
        save      %sp,  -96,  %sp
!  Pruefung,  ob Parameter gleich 0
        mov       %i0,  %l1
        cmp       %l1,  0
        bne       else
        nop
        ba        ende
        mov       1,  %i5              !  Fakultaet (0) = 1
else:
        call      fakul
        sub       %l1,  1,  %o0        !  Fakultaet (n-1)
        smul      %l1,  %o0,  %i5
ende:
        mov       %i5,  %i0
        jmp       %i7+8
        restore

        .align    8
main:
        call      fakul                !  Fakultaet (7)
        mov       7, %o0
        mov       %o0,  %i4            !  Ergebnis retten
```

```
        call     fakul              !  Fakultaet (10)
        mov      10,  %o0
        mov      %o0,  %i5          !  Ergebnis retten

!  Ausdruck der Ergebnisse
        sethi    %hi(fmta),  %l0
        or       %l0,  %lo(fmta),  %l0
        mov      %l0,  %o0
        call     printf
        mov      %i4,  %o1

        add      %l0,  8,  %o0           !  Format aendern
        call     printf
        mov      %i5,  %o1

        mov      1,  %g0
        ta       0

!       Formate
        .section ".rodata1"
        .align   4
fmta:
        .asciz   "a = %d\n"
        .align   4
fmtb:
        .asciz   "b = %d\n"


!   Ergebnis:
!   a = 5040
!   b = 3628800
```

```
!  Beispiel zum Unterprogramm main
!  Parameter auf der Kommandozeile
        .section  ".data"
        .align  8
        korr  =  0x30303030
fmt:    .asciz  "%s + %s = %s\n"
erg:    .asciz  "abcd"

!  Addition zweier ASCII-Zahlen,  manchmal!
        .global  main
        .global printf

        .section  ".text"
main:      save %sp, -96, %sp
!  %io verweist auf Parameterzahl
!  %i1 verweist auf Liste von Zeigern auf Parameter
           ld    [%i1+4],  %o1
           ld    [%o1],  %l4
           ld    [%i1+8],  %o2
           ld    [%o2],  %l5
           add  %l4,  %l5,  %l4
           set   korr,  %l5
           sub  %l4,  %l5,  %l4
           set   erg,  %o3
           set   fmt,  %o0
           call  printf
           st     %l4,  [%o3]
!  Rueckkehr zum Betriebssystem
           mov  0, %i0
           ret
           restore
!  Beispielaufruf:          a.out    3214     4321
!  Ausgabe:                 3214  +  4321  =  7535
```

5.10 Synthetische Instruktionen

Synthetische Befehle sind Kurzformen für etwas längere Hardware-Befehle.

Auswahl:

Bit test	btst	reg/imm, rs
Bit set	bset	reg/imm, rd
Bit clear	bclr	reg/imm, rd
Bit toggle	btog	reg/imm, rd
Call	call	reg/imm
Clear register	clr	rd
Clear byte	clrb	[address]
Clear halfword	clrh	[address]
Clear word	clr	[address]
Compare	cmp	reg, reg/imm
Decrement by 1	dec	rd
Decrement by const13	dec	const13, rd
Increment by 1	inc	rd
Increment by const13	inc	const13, rd
Jump	jmp	address
Move	mov	reg/imm, rd
Not	not	rs, rd
Not	not	rd
Negate	neg	rs, rd
Negate	neg	rd
Set	set	value, rd

Bemerkung: Der synthetische Befehl set value, rd wird manchmal in zwei Hardware-Befehle transformiert.

Zuordnung Synthetischer Befehl – Hardware-Befehl:

Synthetischer Befehl		Hardware-Äquivalent	
btst	reg/imm, rs	andcc	rs, reg/imm, %g0
bset	reg/imm, rd	or	rd, reg/imm, rd
bclr	reg/imm, rd	andn	rd, reg/imm, rd
btog	reg/imm, rd	xor	reg/imm, rd
call	reg/imm	jmpl	reg/imm, %o7
clr	rd	or	%g0, %g0, rd
clrb	[address]	stb	%g0, [address]
clrh	[address]	sth	%g0, [address]
clr	[address]	st	%g0, [address]
cmp	reg, reg/imm	subcc	rs, reg/imm, %g0
dec	rd	sub	rd, 1, rd
dec	const13, rd	sub	rd, const13, rd
inc	rd	add	rd, 1, rd
inc	const13, rd	add	rd, const13, rd
jmp	address	jmpl	address, %g0
mov	reg/imm, rd	or	%g0, reg/imm, rd
not	rs, rd	xnor	rs, %g0, rd
not	rd	xnor	rd, %g0, rd
neg	rs, rd	sub	%g0, rs, rd
neg	rd	sub	%g0, rd, rd

```
! Falls -4097 < value < 4096
set    value, rd        or      %g0, value, rd
! Falls (value&0x3f) = 0
set    value, rd        sethi   %hi(value), rd
! sonst
                        sethi   %hi(value), rd
                        or      rd, %lo(value), rd
```

5.11 Gleitpunktrechnung

```
!     Beispiel zur Rechengenauigkeit bei Fließkommazahlen
!      Programm ohne Optimierungen

            .global   printf
            .global    main

            .align    8
            .section  ".rodata"

!           Benutzte Formate

fma:        .asciz    "Berechnung von 1/3:\n"

fmb:        .asciz    "\n\nRechnung in double:\n\n"

fmc:        .asciz    "Stellenzahl:%3d:%24.*f\n"

fmd:        .asciz    "\n\nRechnung in single:\n\n"

fme:        .asciz    "Stellenzahl:%3d:%24.*f\n"


            .section  ".data"
            .align    8
d1:         .double   0r1.0
d2:         .double   0r3.0
d3:         .double   0r0
            .double   0r0             !  Hilfsspeicher
s1:         .single   0r0

            .section ".text"
main:
            set       d1,  %l0
            ldd       [%l0],  %f6         !  1.0  in  %f6, %f7
            set       d2,  %l0
            ldd       [%l0],  %f4         !  3.0  in  %f4, %f5
            fdivd     %f6,  %f4,  %f4
            set       d3,  %l0
            std       %f4,  [%l0]         !  (double)  1/3
            fdtos     %f4,  %f4           !  Formatwandlung
            set       s1,  %l0
            st        %f4,  [%l0]         !  (single)  1/3

            set       fma,  %l0
            call      printf
            mov       %l0,  %o0

            set       fmb,  %l0
            call      printf
            mov       %l0,  %o0

            mov       8,  %i5             !  Stellenzahl

            set       fmc,  %l1
SL01:
            set       d3,  %l5
            ldd       [%l5],  %f4
            mov       %l1,  %o0
            mov       %i5,  %o1
            mov       %i5,  %o2
            std       %f4,  [%l5+8]
            ld        [%l5+8],  %o3
            call      printf
            ld        [%l5+12],  %o4
```

```
        inc     %i5
        cmp     %i5,  16
        bl      SL01
        nop

        set     fmd,  %l1
        call    printf
        mov     %l1,  %o0

        mov     5,  %i5

        set     fme,  %l1
SL02:
        set     s1,  %l5
        ld      [%l5],  %f4
        fstod   %f4,  %f4          !  Formatwandlung
        mov     %l1,  %o0          !  fuer printf
        mov     %i5,  %o1
        mov     %i5,  %o2
        std     %f4,  [%l5-8]
        ld      [%l5-8],  %o3
        call    printf
        ld      [%l5-4],  %o4
        inc     %i5
        cmp     %i5,  11
        bl      SL02
        nop

!       Rueckkehr ins Betriebssystem
        mov     1,    %g1
        ta      0
```

Berechnung von 1/3:

Rechnung in double:

```
Stellenzahl:  8:        0.33333333
Stellenzahl:  9:        0.333333333
Stellenzahl: 10:        0.3333333333
Stellenzahl: 11:        0.33333333333
Stellenzahl: 12:        0.333333333333
Stellenzahl: 13:        0.3333333333333
Stellenzahl: 14:        0.33333333333333
Stellenzahl: 15:        0.333333333333333
```

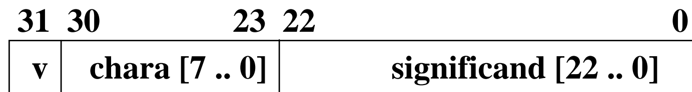
Rechnung in single:

```
Stellenzahl:  5:        0.33333
Stellenzahl:  6:        0.333333
Stellenzahl:  7:        0.3333333
Stellenzahl:  8:        0.33333334
Stellenzahl:  9:        0.333333343
Stellenzahl: 10:        0.3333333433
```

Frage: Woher kommt die 4 in der Dezimaldarstellung von 1.0/3.0 für das Single-Format?

Drei Arten von Fließkommazahlen auf der Sparc V8:

(a) Floating-Point Singleword Format:



Bezeichnungen:

- v = Vorzeichen (1 Bit)
- c = Charakteristik (8 Bit)
- s = Signifikand (23 Bit)

Normale Zahl: $(0 < c < 255)$	$(-1)^v * 2^{c-127} * 1.s$
Subnormale Zahl: $(c = 0)$	$(-1)^v * 2^{-126} * 0.s$
Null: $(c = 0)$	$(-1)^v * 0$

Signalisierende NaN:

v = undefiniert, c = 255,
s = 0uu .. uu, mindestens eine Ziffer u ≠ 0

Stille NaN:

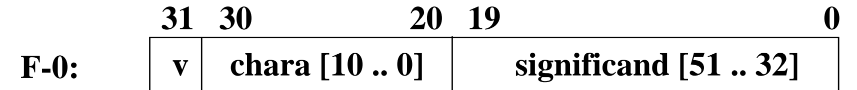
v = undefiniert, c = 255, s = 1uu .. uu

–∞	v = 1, c = 255, s = 00 ... 00
+∞:	v = 0, c = 255, s = 00 ... 00

Drei Arten von Fließkommazahlen auf der Sparc V8:

(b) Floating-Point Doubleword Format:

F-0:



F-1:

Bezeichnungen:

- v = Vorzeichen (1 Bit)
- c = Charakteristik (11 Bit)
- s = Signifikand (52 Bit)

Normale Zahl: $(0 < c < 2047)$	$(-1)^v * 2^{c-1023} * 1.s$
Subnormale Zahl: $(c = 0)$	$(-1)^v * 2^{-1022} * 0.s$
Null: $(c = 0)$	$(-1)^v * 0$

Signalisierende NaN:

v = undefiniert, c = 2047,
s = 0uu .. uu, mindestens eine Ziffer u ≠ 0

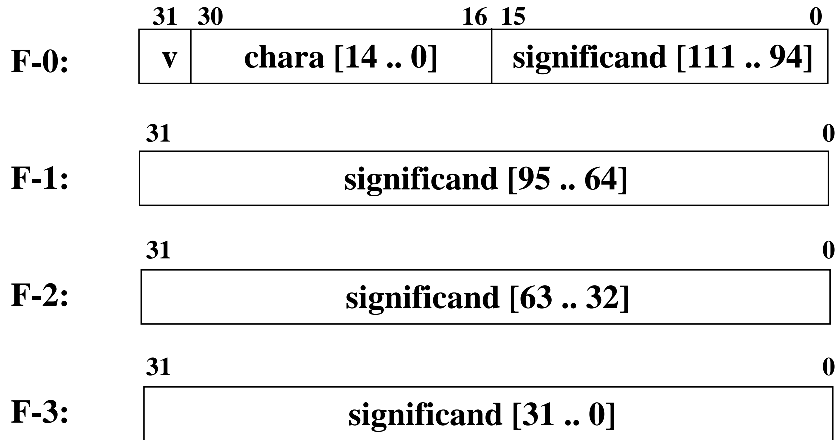
Stille NaN:

v = undefiniert, c = 2047, s = 1uu .. uu

–∞	v = 1, c = 2047, s = 00 ... 00
+∞:	v = 0, c = 2047, s = 00 ... 00

Drei Arten von Fließkommazahlen auf der Sparc V8:

(c) Floating-Point Quadword Format:



Bezeichnungen:

- v = Vorzeichen (1 Bit)
- c = Charakteristik (15 Bit)
- s = Signifikand (112 Bit)

Normale Zahl: $(0 < c < 32767)$ $(-1)^v * 2^{c-16383} * 1.s$

Subnormale Zahl: $(c = 0)$ $(-1)^v * 2^{-16382} * 0.s$

Null: $(c = 0)$ $(-1)^v * 0$

Signalisierende NaN:
v = undefiniert, c = 32767,
s = 0uu .. uu, mindestens eine Ziffer $u \neq 0$

Stille NaN: v = undefiniert, c = 32767, s = 1uu .. uu

–∞ v = 1, c = 32767, s = 00 ... 00
+∞: v = 0, c = 32767, s = 00 ... 00

```
Beispiel einer Fließkomma-Addition:

c :=  a + b  mit    a = 0xc1ff00aa (-31.875324)
             und    b = 0x4400ffbb (515.995789)

a   =  -  131  (1,Signifikand)
      1 100.0001.1 1,111.1111.0000.0000.1010.1010

b   =  +  136  (1,Signifikand)
      0 100.0100.0 1,000.0000.1111.1111.1011.1011

b          1,000.0000.1111.1111.1011.1011.0000.0
|a|               111.1111.1000.0000.0101.0101.0
-|a|       1,111.1000.0000.0111.1111.1010.1011.0
b-|a|      0,111 1001.0000.0111.1011.0101.1011.0

a+b   =  +  135  (1,111.0010.0000.1111.0110.1011)

c = 0x43f20f6b  (484.120453)
```

Fließkomma-Status-Register:

rd	u	tem	ns	res	ver	ftt	qn	u	fcc	aexc	cexc
31 30	29 28	27..23	22	21 20	19..17	16..14	13	12	11 10	9..5	4..0

Abkürzungen:

- **rd = rounding direction**
- **u = unused**
- **tem = trap enable mask**
- **ns = nonstandard**
- **res = reserved**
- **ver = version**
- **ftt = floating point trap type**
- **qn = deferred trap queue**
- **fcc = floating point condition code**
- **aexc = accrued exception**
- **cexc = current exception**

Zur Fließkommarechnung:

Es existieren vier Rundungsmodi:

Rundungsmodi:	**rd**	**= 00**	**runden zum nächsten Wert**
		= 01	**runden nach 0**
		= 10	**runden nach +∞**
		= 11	**runden nach –∞**

Beispiel: Runden zum nächsten Wert:

10,1 bis 10,4	**==>**	**10**
10,5	**==>**	**10**
10,6 bis 10,9	**==>**	**11**
11		
11,1 bis 11,4	**==>**	**11**
11,5	**==>**	**12**
11,6 bis 11,9	**==>**	**12**

Bemerkung: **Die verschiedenen Rundungsmodi kann man zum Beispiel nutzen, um eine Intervallarithmetik zu implementieren.**

Für Zahlvergleiche existieren vier Ergebnisanzeigen:

Anzeige:	**fcc**	**= 00**	**Zahl1 = Zahl2**
		= 01	**Zahl1 < Zahl2**
		= 10	**Zahl1 > Zahl2**
		= 11	**Zahl1 und Zahl2 sind unvergleichbar.**

Es existiert ein Befehl zum Schreiben des Fließkomma-Status-Registers: **ld [address], %fsr**

Beispiele zum Vergleich von Fließkommazahlen in Hochsprachen:

(12.12 < 24.06) = true
(12.12 = 24.06) = false
(12.12 > 24.06) = false

(12.12 < Inf) = true
(12.12 = Inf) = false
(12.12 > Inf) = false

(12.12 < NaN) = false
(12.12 = NaN) = false
(12.12 > NaN) = false

(Inf < Inf) = false
(Inf = Inf) = true
(Inf > Inf) = false

(Inf < -Inf) = false
(Inf = -Inf) = false
(Inf > -Inf) = true

(Inf < NaN) = false
(Inf = NaN) = false
(Inf > NaN) = false

(NaN < NaN) = false
(NaN = NaN) = false
(NaN > NaN) = false

Fließkomma-Register:

Ex existieren 32 globale Fließkommaregister:
%f0, %f1, . . ., %f31

%f0
%f1
⋮
%f31

Bemerkungen:

(i) Die Registerbreite beträgt 32 Bit.

(ii) Single-Zahlen werden in einem Register gespeichert, Double-Zahlen in zwei aufeinanderfolgenden Registern, Quadruple-Zahlen in vier aufeinanderfolgenden Registern.

(iii) In den F-Registern sind die Operanden auszurichten, Double-Operanden beginnen in einem geradzahligen Register, Quadruple-Operanden in einem durch 4 teilbaren Register.

(iv) Die Fließkommaregister %f0 und %f1 werden im "Application Binary Interface" ABI nur zur Rückgabe von Fließkommawerten genutzt.

```
!       Beispiel zu Registerkonventionen
!       für den SPARC-Assembler
!       bei Nutzung von Fließkomma-Unterprogrammen.
!       hier:  double addf2 (double a,  double b) {
!                          return a + b;
!                   }
                .section   ".text"
                .align     8
                .global    addf2
addf2:
        save        %sp,  -112,  %sp

!       Es existieren keine Befehle,  die Inhalte von
!       Ganzzahlregistern in Fließkommaregister
!       und umgekehrt transportieren.
        st          %i0,  [%fp-8]
        st          %i1,  [%fp-4]
        st          %i2,  [%fp-16]
        st          %i3,  [%fp-12]
        ldd         [%fp-8],  %f6
        ldd         [%fp-16],  %f4

        faddd       %f6,  %f4,  %f4

!       Ergebnisrueckgabe in  %f0 und %f1
        fmovs       %f5,  %f1
        fmovs       %f4,  %f0
        !  Manchmal ist Rueckgabe ueber Keller sinnvoll.
        !  std      %f4,  [%fp-8]
        jmp         %i7+8
        restore
```

Fließkomma-Operationen:

Addition und Subtraktion:

```
fadds,  faddd,  faddq         fregs1,  fregs2,  fregd
fsubs,  fsubd,  fsubq           "        "        "
```

Multiplikation und Division:

```
fmuls,  fmuld,  fmulq
fsmuld,  fdmulq               fregs1,  fregs2,  fregd
fdivs,  fdivd,  fdivq           "        "        "
```

(fsmuld berechnet das exakte Produkt zweier single Größen, analog fdmulq)

Quadratwurzel:

```
fsqrts,  fsqrtd,  fsqrtq      fregs,  fregd
```

Vergleiche:

```
fcmps,  fcmpd,  fcmpq         fregs1,  fregs2
```

Verschieben ("nur single"):

```
fmovs,  fnegs,  fabss         fregs1,  fregd
```

Zahlwandlungen:

```
fitos,  fitod,  fitoq         fregs1,  fregds
fstoi,  fdtoi,  fqtoi         fregs1,  fregds
fstod,  fstoq                   "        "
fdtos,  fdtoq                   "        "
fqtos,  fqtod                   "        "
```

Ladebefehle:

ldf, lddf	**[address], fregd**

Speicherbefehle:

stf, stdf,	**fregs, [address]**

Die Assemblernotation für Lade- und Speicherbefehle der Fließkommaeinheit entspricht den Ganzzahlbefehlen.

Sprungbefehle:

fba	**floating branch always**
fbn	**floating branch never**
fbu	**floating branch on unordered**
fbo	**floating branch on ordered**
fbe	**floating branch on equal**
fbue	**floating branch on unordered or equal**
fbne	**floating branch on not equal**
fblg	**floating branch on less or greater**
fbl	**floating branch on less**
fbul	**floating branch on unordered or less**
fble	**floating branch on less or equal**
fbule	**floating branch on unordered, less or equal**
fbge	**floating branch on greater or equal**
fbuge	**floating branch on unordered, greater or equal**
fbg	**floating branch on greater**
fbug	**floating branch on unordered or greater**

Bemerkung: **Geht einer fbxx-Instruktion unmittelbar eine fcmxx-Instruktion voraus, dann ist das Ergebnis nicht vorhersehbar.**