



THE AUSTRALIAN NATIONAL UNIVERSITY

TR-CS-00-03

**SPARC V9 Instruction Set
Specification**

Bill Clarke

October 2000

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports
Department of Computer Science
Faculty of Engineering and Information Technology
The Australian National University
Canberra ACT 0200
Australia

or send email to:

`Technical.Reports@cs.anu.edu.au`

A list of technical reports, including some abstracts and copies of some full reports may be found at:

<http://cs.anu.edu.au/techreports/>

Recent reports in this series:

- TR-CS-01-02 Jeremy E. Dawson and Rajeev Gore. *Mechanising cut-elimination for display logic*. October 2001.
- TR-CS-01-01 Stephen Roberts Peter Christen, Markus Hegland and Irfan Altas. *A scalable parallel fem surface fitting algorithm for data mining*. October 2001.
- TR-CS-00-02 Peter Strazdins. *A survey of simulation tools for cap project phase iii*. October 2000.
- TR-CS-00-03 Bill Clarke. *Sparc v9 instruction set specification*. October 2000.
- TR-CS-00-01 Jens Gustedt, Ole A. Maehle, and Jan Arne Telle. *Java programs do not have bounded treewidth*. February 2000.
- TR-CS-99-02 Samuel Taylor. *A distributed visualisation tool for digital terrain models*. July 1999.

SPARC-V9 Instruction-Set Syntax Specification

Bill Clarke
llib@computer.org
CAP Project
Australian National University

November 22, 2000

Contents

1	Introduction	2
2	Fields	3
2.1	Field specifications	3
2.2	Names and subfields	5
2.2.1	Integer registers	5
2.2.2	Ancillary state registers	5
2.2.3	Privileged registers	5
2.2.4	Condition code registers	6
2.2.5	Annul and predict fields	6
2.2.6	Condition fields	6
3	Structured operands and typed constructors	8
3.1	Register-or-immediate structured operands	8
3.2	Addresses and other address-like structured operands	9
3.3	Floating-point register encoding	10
3.3.1	Non-decoding floating-point registers	12
4	Opcodes	13
4.1	Branches and related opcodes	13
4.2	Arithmetic and logical opcodes	15
4.3	Load and store opcodes	18
4.4	Floating-point opcodes	20
5	Constructors	22
5.1	Load and store constructors	22
5.2	Read and write constructors	25
5.3	Shift, logic, and arithmetic	26
5.4	Branches and call	27
5.5	Conditional moves and integer register conditions	28
5.6	Floating point	30
5.7	Miscellany	31
6	UltraSPARC implementation-dependent instructions	32
6.1	Ancillary state registers	32
6.2	Shutdown	33
6.3	Graphics instructions	33
6.3.1	Edge handling, array and align address	34
6.3.2	Pixel compare, partitioned multiply, pack and pixel distance	35
6.3.3	Align data, merge, expand and partitioned add	36
6.3.4	Logical	37
6.3.5	Constructors	38
6.4	Assembler syntax	39
6.5	Validating	40
7	SPARC64 implementation-dependent instructions	41
7.1	Ancillary state registers	41
7.2	Floating-point multiply-add	45
7.3	Assembler syntax	46
8	Synthetic instructions	47
8.1	Assembler syntax	48
8.2	Validating	49
9	Assembler syntax	50
10	Application-specific specifications for simulation	52
11	Validating against the SPARC-V9 assembler	53
A	List of Chunks	55
B	Index	56

1 Introduction

This document specifies the SPARC-V9 instruction set syntax, adapted by Bill Clarke from the `njmetk-v0.5` SPARC-V8 instruction set [1, ch.2]. For more info regarding the special commands used herein, see that code/document. For more information on the New Jersey Machine Code Toolkit, see [2].

The context of this specification is with regard to simulating SPARC-V9 cpu's, and hence may be different to other contexts. All valid instructions in SPARC-V9 ought to be recognised by this specification. Any instruction not directly matched ought to generate an `illegal_instruction` trap when executed or emulated. Some instructions matched may later generate an `illegal_instruction` trap. Any instruction that could cause some other trap and could be detected here (possibly in equations, e.g., the quad-precision floating-point encoding of registers) will have a constructor that accepts these invalid instructions and a constructor that accepts only the truly "valid" instructions.

This specification also includes implementation-dependent instructions, such as the VIS instruction-set implemented by the UltraSPARC family of CPUs.

This specification also includes special constructors for simulation. For example, all the `BPr` opcodes are matched with a single `BPr` constructor with the condition type as an extra field.

2 Fields

The SPARC-V9 is a RISC architecture and thus uses a single token class for all of its fields.

```
3a <fields.spec 3a>≡
    # fields.spec
    # this goes first
    fields of instruction (32) <field specifications 3b>
```

Information about instruction formats and fields is taken from Chapter 6 of the SPARC-V9 architecture manual [3].

2.1 Field specifications

Working from top to bottom of pages 64 and 65 of the SPARC-V9 architecture manual, we have format 1,

```
3b <field specifications 3b>≡ (3a) 3c>
    inst 0:31 op 30:31 disp30 0:29
```

Defines:

```
    disp30, used in chunk 28.
    inst, never used.
    op, used in chunk 13d.
```

format 2,

```
3c <field specifications 3b>+≡ (3a) <3b 4a>
    rd 25:29 op2 22:24 imm22 0:21 a 29:29 icond_f2 25:28 fcond_f2 25:28
    disp22 0:21 icc_f2 20:21 fcc_f2 20:21 p 19:19
    disp19 0:18 mbz_f2 28:28 rcond_f2 25:27 d16hi 20:21 d16lo 0:13
```

Defines:

```
    a, used in chunks 6c, 27, 29, 30c, 47–49, 52, and 53.
    d16hi, used in chunk 29.
    disp19, used in chunk 27.
    disp22, used in chunk 27.
    d16lo, used in chunk 29.
    fcc_f2, used in chunks 6b and 27.
    fcond_f2, used in chunks 6d and 27.
    icc_f2, used in chunks 6b and 27.
    icond_f2, used in chunks 6d, 27, and 28.
    imm22, used in chunks 13e, 27a, and 31.
    mbz_f2, used in chunk 14.
    op2, used in chunk 13e.
    p, used in chunks 6c, 27, and 29.
    rcond_f2, used in chunks 7 and 29.
    rd, used in chunks 5c, 11a, 13e, 16, 19a, 23, 25, 26, 28, 30–33, 39a, 42, 44, and 47.
```

format 3,

4a *(field specifications 3b)*+≡ (3a) <3c 4b>
 op3 19:24 rs1 14:18 i 13:13 imm_asi 5:12 rs2 0:4
 simm13 0:12 rcond_f3 10:12 simm10 0:9 membar_mask 0:6
 impldep1 25:29 impldep2 0:18 x 12:12 shcnt32 0:4
 shcnt64 0:5 opf 5:13 mbz_f3 27:29 fcc_f3 25:26 fcn 25:29

Defines:

fcc_f3, used in chunks 6b and 31a.
 fcn, used in chunks 17a and 24.
 i, used in chunks 8, 10b, 16, and 44.
 imm_asi, used in chunk 10.
 impldep1, never used.
 impldep2, never used.
 mbz_f3, used in chunk 22a.
 membar_mask, used in chunk 25a.
 op3, used in chunks 15 and 18.
 opf, used in chunks 20, 22a, and 33b.
 rcond_f3, used in chunks 7 and 30.
 rs1, used in chunks 5c, 9, 11a, 16, 17d, 23e, 25, 26, 29, 30, 32b, 33a, 39a, 42, 44, and 47.
 rs2, used in chunks 5c, 8, 9, 11a, 23e, 39a, 44, and 47.
 shcnt32, used in chunk 8c.
 shcnt64, used in chunk 8c.
 simm10, used in chunk 8c.
 simm13, used in chunks 8, 9, and 25a.
 x, used in chunk 17b.

and format 4

4b *(field specifications 3b)*+≡ (3a) <4a 5d>
 icc_f4 11:12 fcc_f4 11:12 simm11 0:10 cc2_f4 18:18 mbz_f4_18 18:18
 icond_f4 14:17 fcond_f4 14:17
 simm7 0:6 mbz_f4_13 13:13 rcond_f4 10:12 opf_low5 5:9 opf_cc 11:13
 opf_low6 5:10

Defines:

cc2_f4, used in chunk 28.
 fcc_f4, used in chunks 6b and 28e.
 fcond_f4, used in chunks 6d, 28e, and 29a.
 icc_f4, used in chunks 6b and 28.
 icond_f4, used in chunks 6d, 28, and 29.
 mbz_f4_13, used in chunk 22a.
 mbz_f4_18, used in chunk 22a.
 opf_cc, used in chunks 6b and 29.
 opf_low5, used in chunk 22a.
 opf_low6, used in chunk 22a.
 rcond_f4, used in chunks 7 and 30.
 simm11, used in chunk 8c.
 simm7, used in chunks 8c and 9b.

Unfortunately, some of the new (V9) instructions have similarly named bitfields but are located in different places (cond, cc, rcond and opf_low): these are labelled here instead by suffixing _f2, _f3 or _f4 depending on whether the field is defined in Format 2, 3 or 4. This convention breaks down for opf_low since they are both in format 4, so they are labelled opf_low5 and opf_low6 (ick!).

The cc fields are also sometimes labelled icc or fcc to identify whether they're using the integer or floating-point condition codes, and also to aid in generating assembly language.

Like the cc fields, the cond fields are also labelled with an i or f prefix to indicate which condition-code the condition is to be tested against.

We have also added some mbz_f{2, 3, 4}_{18, 13} fields which Must-Be-Zero for some instructions (the mbz_f4_X fields identify which bit must-be-zero).

Other fields, such as floating-point registers and specialised register fields are defined elsewhere.

2.2 Names and subfields

To aid in generating assembly language, we identify names to particular fields. This way they will print according to the format specified in the SPARC-V9 manual. Some of the names/assembly syntax are identified by typed constructors in section 3 (such as `reg_or_imm` and floating-point registers), or explicitly in the generating constructors (such as the `reg_plus_imm/%asi` variants of the load and store instructions in section 5.1), or converted from opcode syntax to assembler syntax in section 11.

We store these `fieldinfo` specifications in a separate file, so that we can concatenate other field specifications with the above ones.

```
5a <fields-names.spec 5a>≡
    # fields-names.spec
    # requires fields.spec
    <fieldinfo specifications 5c>
```

2.2.1 Integer registers

Firstly, we have the integer registers which are separated into `globals`, `outs`, `locals` and `ins`. Additionally, out register 6 is specified as the stack pointer (`sp`), and in register 6 is specified as the frame pointer (`fp`) [3, pp291–292].

```
5b <properties of integer-register fields 5b>≡ (5c)
    names [ "%g0" "%g1" "%g2" "%g3" "%g4" "%g5" "%g6" "%g7"
            "%o0" "%o1" "%o2" "%o3" "%o4" "%o5" "%sp" "%o7"
            "%l0" "%l1" "%l2" "%l3" "%l4" "%l5" "%l6" "%l7"
            "%i0" "%i1" "%i2" "%i3" "%i4" "%i5" "%fp" "%i7" ]
```

```
5c <fieldinfo specifications 5c>≡ (5a) 6a>
    fieldinfo
    [ rd rs1 rs2 ] is [ <properties of integer-register fields 5b> ]
    Uses rd 3c, rs1 4a, and rs2 4a.
```

There are several related “integer-register-like” sets of registers, including floating-point registers, ancillary state registers (`asrs`) and privileged registers.

Floating-point registers are encoded (including their assembly encoding) using typed constructors in section 3.3.

2.2.2 Ancillary state registers

Ancillary state registers are only manipulated using `{RD,WR}ASR`, and are identified by `%asrX` where `X` is the number of the register. Since the encoding is so simple, we let the toolkit deal with printing the `X` and explicitly include the `%asr` in the instructions’ constructors (see section 5.2). Hence we need register-like fields that don’t have explicit names (the `i` implies integer):

```
5d <field specifications 3b>+≡ (3a) <4b 5e>
    rsl1 14:18 rdi 25:29
    Defines:
    rdi, used in chunk 25.
    rsl1, used in chunk 25.
```

2.2.3 Privileged registers

The privileged registers are encoded in the same sort of field as plain registers, and are identified by appending a `p` to the register identifiers. The `%pX!` in the specification are illegal values (which are enforced in the pattern or constructor).

```
5e <field specifications 3b>+≡ (3a) <5d>
    rslp 14:18 rdp 25:29
    Defines:
    rdp, used in chunks 6a and 26.
    rslp, used in chunks 6a and 26.
```

```

6a <fieldinfo specifications 5c>+≡ (5a) <5c 6b>
    fieldinfo
        [ rslp rdp ] is [ names [
            "%tpc"      "%tnpc"      "%tstate"  "%tt"
            "%tick"    "%tba"      "%pstate"  "%tl"
            "%pil"     "%cwp"      "%cansave" "%canrestore"
            "%cleanwin" "%otherwin" "%wstate"  "%fq"
            "%p16!"    "%p17!"    "%p18!"   "%p19!"
            "%p20!"    "%p21!"    "%p22!"   "%p23!"
            "%p24!"    "%p25!"    "%p26!"   "%p27!"
            "%p28!"    "%p29!"    "%p30!"   "%ver" ] ]

```

Uses rdp 5e and rslp 5e.

2.2.4 Condition code registers

The condition code registers are identified (like the other registers) by names preceded by a %. There are two integer condition code registers (*icc* and *xcc*), which are represented by a two-bit field, so two values are not used (and are illegal; this is identified by restrictions in patterns or equations in constructors, and `{i,x}cc!` in the names). There are four floating-point condition code registers, and are just labelled by *fccX* where *X* is the value of the field. Other condition code values are a combination of the two.

```

6b <fieldinfo specifications 5c>+≡ (5a) <6a 6c>
    fieldinfo
        [ icc_f2 icc_f4 ] is [ names [ "%icc"  "%icc!" "%xcc"  "%xcc!" ] ]
        [ fcc_f2 fcc_f3 fcc_f4 ]
            is [ names [ "%fcc0" "%fcc1" "%fcc2" "%fcc3" ] ]
        opf_cc          is [ names [ "%fcc0" "%fcc1" "%fcc2" "%fcc3"
            "%icc"  "%icc!" "%xcc"  "%xcc!" ] ]

```

Uses fcc_f2 3c, fcc_f3 4a, fcc_f4 4b, icc_f2 3c, icc_f4 4b, and opf_cc 4b.

2.2.5 Annul and predict fields

The annul and predict fields of various branch instructions are specified by suffixing `{ , , a }` and `{ "pn" , "pt" }` respectively, depending on whether the field is not or is set. We can use field names to identify these, for which the instructions are declared in section 5.4.

```

6c <fieldinfo specifications 5c>+≡ (5a) <6b 6d>
    fieldinfo
        a is [ names [ " " , "a" ] ]
        p is [ names [ " ,pn" , "pt" ] ]

```

Uses a 3c and p 3c.

2.2.6 Condition fields

The `{i,f}cond_f{2,4}` fields specify the condition to check for a given condition code register (see above for the condition code registers). Each of these conditions is given a name which is used as a suffix for opcodes and assembly language and which is enumerated here. We can then use these fields as parts of constructors to produce the various instructions that depend on the fields.

```

6d <fieldinfo specifications 5c>+≡ (5a) <6c 7>
    fieldinfo
        [ icond_f2 icond_f4 ] is
            [ names [ N E LE L LEU CS NEG VS A NE G GE GU CC POS VC ] ]

        [ fcond_f2 fcond_f4 ] is
            [ names [ N NE LG UL L UG G U A E UE GE UGE LE ULE O ] ]

```

Uses fcond_f2 3c, fcond_f4 4b, icond_f2 3c, and icond_f4 4b.

The `rcond_f{2,3,4}` fields specify the integer register value condition to check, and like the `{i,f}cond_*` fields the names of that condition give a name for the opcode. Two of the possible values are invalid and are labelled with `!`.

```
7 <fieldinfo specifications 5c>+≡ (5a) <6d
  fieldinfo
    [ rcond_f2 rcond_f3 rcond_f4 ] is
      [ names [ "0!" Z LEZ LZ "4!" NZ GZ GEZ ] ]
```

Uses `rcond_f2 3c`, `rcond_f3 4a`, and `rcond_f4 4b`.

3 Structured operands and typed constructors

SPARC-V9 has instructions whose operands are not simple integers or fields. For example, the integer-arithmetic instructions take an operand that may be a register or an immediate value, and the load and store instructions take an operand that computes an address. The formats for these operands appear on pages 295–296 in Appendix G of the SPARC-V9 manual.

8a $\langle types.spec\ 8a \rangle \equiv$
 # types.spec
 # requires fields.spec
 $\langle type\ specifications\ 8b \rangle$

3.1 Register-or-immediate structured operands

We specify such operands by creating a constructor type for them, giving a constructor for each format. We use the “operand syntax” name in the SPARC-V9 manual as the name of the type; for example, the constructors for a “register or immediate” operand are:

8b $\langle type\ specifications\ 8b \rangle \equiv$ (8a) 8c >
 constructors
 immROI simm13! : reg_or_imm is i = 1 & simm13
 regROI rs2 : reg_or_imm is i = 0 & rs2

Defines:

immROI, used in chunks 9 and 47.
 reg_or_imm, used in chunks 9a, 25, 26, 33a, 44, and 47.
 regROI, used in chunks 9 and 47.
 Uses i 4a, rs2 4a, and simm13 4a.

Similarly, we define constructors for other register-or-immediate operand syntax:

8c $\langle type\ specifications\ 8b \rangle + \equiv$ (8a) < 8b 9a >
 constructors
 immROI10 simm10! : reg_or_imm10 is i = 1 & simm10
 regROI10 rs2 : reg_or_imm10 is i = 0 & rs2

 immROI11 simm11! : reg_or_imm11 is i = 1 & simm11
 regROI11 rs2 : reg_or_imm11 is i = 0 & rs2

 immROS32 shcnt32 : reg_or_shcnt32 is i = 1 & shcnt32
 regROS32 rs2 : reg_or_shcnt32 is i = 0 & rs2

 immROS64 shcnt64 : reg_or_shcnt64 is i = 1 & shcnt64
 regROS64 rs2 : reg_or_shcnt64 is i = 0 & rs2

 immROI7 simm7! : reg_or_imm7 is i = 1 & simm7
 regROI7 rs2 : reg_or_imm7 is i = 0 & rs2

Defines:

immROI1, never used.
 immROI7, used in chunk 9b.
 immROS3, never used.
 immROS6, never used.
 reg_or_imm10, used in chunk 30.
 reg_or_imm11, used in chunk 28.
 reg_or_imm7, used in chunk 9b.
 reg_or_shcnt32, used in chunk 26c.
 reg_or_shcnt64, used in chunk 26c.
 regROI1, never used.
 regROI7, used in chunk 9b.
 regROS3, never used.
 regROS6, never used.
 Uses i 4a, rs2 4a, shcnt32 4a, shcnt64 4a, simm10 4a, simm11 4b, and simm7 4b.

3.2 Addresses and other address-like structured operands

Specifying addresses is a bit problematic because it's not clear what convention to follow. The underlying general mechanism is that a register is added to a value of type `reg_or_imm`, but there are many useful abbreviations:

```
9a (type specifications 8b)+≡ (8a) <8c 9b>
constructors
# generalA  rsl + reg_or_imm : address_
generalA  rsl + reg_or_imm : address_ is rsl & reg_or_imm
dispA     rsl + simm13!     : address_ is generalA(rsl, immROI(simm13!))
absoluteA simm13!         : address_ is generalA(0,   immROI(simm13!))
indexA    rsl + rs2       : address_ is generalA(rsl, regROI(rs2))
indirectA rsl             : address_ is generalA(rsl, regROI(0))
```

Defines:

```
absoluteA, never used.
address_, used in chunks 23, 24, 31c, and 47.
dispA, used in chunk 47.
generalA, never used.
indexA, never used.
indirectA, never used.
```

Uses `immROI 8b`, `reg_or_imm 8b`, `regROI 8b`, `rs1 4a`, `rs2 4a`, and `simm13 4a`.

(the commented-out line is what should be used; the second is a workaround for a bug in the m1 version of the toolkit!!!).

Unfortunately we can't call the type `address` because `address` is reserved for the toolkit to describe the treatment of addresses in decoding specifications.

There are several other operand syntax specifications like `address_`, given below (names taken from `address_`, even if they don't make much sense if they're not addresses). `reg_plus_imm` and `regaddr` are only used in conjunction with alternate space instructions (`reg_plus_imm` uses the ASI register, `regaddr` has an immediate asi). `software_trap_number` is used for a software trap (Tcc).

```
9b (type specifications 8b)+≡ (8a) <9a 10a>
constructors
dispRPI    rsl + simm13! : reg_plus_imm is rsl & immROI(simm13!)
absoluteRPI simm13!     : reg_plus_imm is dispRPI(0,   simm13!)
indirectRPI rsl         : reg_plus_imm is dispRPI(rsl, 0)

indexRA    rsl + rs2    : regaddr      is rsl & regROI(rs2)
indirectRA rsl         : regaddr      is indexRA(rsl, 0)

# generalSTN rsl + reg_or_imm7 : software_trap_number
generalSTN  rsl + reg_or_imm7 : software_trap_number is rsl & reg_or_imm7
dispSTN     rsl + simm7!     : software_trap_number is generalSTN(rsl, immROI7(simm7!))
absoluteSTN simm7!         : software_trap_number is generalSTN(0,   immROI7(simm7!))
indexSTN    rsl + rs2       : software_trap_number is generalSTN(rsl, regROI7(rs2))
indirectSTN rsl             : software_trap_number is generalSTN(rsl, regROI7(0))
```

Defines:

```
absoluteRPI, never used.
absoluteSTN, never used.
dispRPI, never used.
dispSTN, never used.
generalSTN, never used.
indexRA, never used.
indexSTN, never used.
indirectRA, never used.
indirectRPI, never used.
indirectSTN, never used.
regaddr, used in chunk 10a.
reg_plus_imm, used in chunk 10a.
software_trap_number, used in chunk 28.
```

Uses `immROI 8b`, `immROI7 8c`, `reg_or_imm7 8c`, `regROI 8b`, `regROI7 8c`, `rs1 4a`, `rs2 4a`, `simm13 4a`, and `simm7 4b`.

One operand syntax not defined in the assembly language syntax is the asi alternative instructions. Since SLED cannot handle overloaded constructors, we could either specialise each instruction for either immediate or implicit asi's (ugly), or add a constructor to switch between them (slightly less ugly). So we do the latter:

```
10a <type specifications 8b>+≡ (8a) <9b 10b>
constructors
# imm_asi_address [regaddr] imm_asi : asi_address
  imm_asi_address [regaddr] imm_asi : asi_address is regaddr & imm_asi
# imp_asi_address [reg_plus_imm] "%asi" : asi_address
  imp_asi_address [reg_plus_imm] "%asi" : asi_address is reg_plus_imm
```

Defines:

```
asi_address, used in chunks 23 and 24.
imm_asi_address, never used.
imp_asi_address, never used.
```

Uses imm_asi 4a, regaddr 9b, and reg_plus_imm 9b.

The compare-and-swap instructions have their own specialised asi-related syntax. Strangely, it seems contradictory with the reg_or_imm specification: reg_or_imm defines i=1 to have an immediate value whereas casa_asi defines i=0 to have an immediate asi.

```
10b <type specifications 8b>+≡ (8a) <10a
constructors
  imm_casa_asi imm_asi : casa_asi is i = 0 & imm_asi
  imp_casa_asi "%asi" : casa_asi is i = 1
```

Defines:

```
casa_asi, used in chunk 23e.
imm_casa_asi, used in chunk 47.
imp_casa_asi, never used.
```

Uses i 4a and imm_asi 4a.

3.3 Floating-point register encoding

SPARC-V9 also has an encoded representation for floating-point registers. The actual register number is a 6-bit number, but is stored in the instruction as a 5-bit encoded value. The actual encoding is:

Type	6-bit actual register (5 to 0)	5-bit encoding (4 to 0)
single	[0 b4 b3 b2 b1 b0]	[b4 b3 b2 b1 b0]
double	[b5 b4 b3 b2 b1 0]	[b4 b3 b2 b1 b5]
quad	[b5 b4 b3 b2 0 0]	[b4 b3 b2 0 b5]

The single and double encodings are effectively foolproof for decoding: all possible values are valid. The quad encoding requires that bit 1 of the encoded field be zero, and hence the decoded field is divisible by 4; however, illegal encodings of quad fields do not result in an `illegal_instruction` trap, they result in an `fp_exception_other` trap, with `FSR.ftt = 6 (invalid_fp_register)` [3, p.40]. One could use the same encoding as for double, and then test elsewhere that the resultant register is divisible by 4, or, as I've done here, apply an extra constructor which allows for the explicit (de/en)coding of an invalid register¹.

We store the floating-point register encoding in a separate file so we can use a non-decoding version for speed.

```
10c <types-fpre.spec 10c>≡
# types-fpre.spec
# after types.spec (requires fields.spec)
<fpre type specifications 11a>
```

The single encoding is straightforward: a direct copy of the bits required.

```
10d <fseqnv 10d>≡ (11a)
{ fr@[5:31] = 0 }
```

¹While the Solaris 2.6/7 assemblers (WorkShop Compilers 4.X dev 18 Sep 1996/5.0 Alpha 03/27/98 Build) do not allow invalid (ie. divisible by 2, but not by 4) quad registers to be assembled, it seems that when binary encoded and run (on an UltraSPARC) it works as expected (not fully checked); I have not attempted to determine what traps occur however. Indeed, all the UltraSPARC processors trap on executing a quad instruction and emulate the instruction in software; this software may not even check for invalid quad encoding. A note on page 240/246 of [4] states that UltraSPARC does not detect or generate an `invalid_fp_register` trap directly in hardware.

11a $\langle \text{fpre type specifications } 11a \rangle \equiv$ (10c) 12b>
 constructors
 $\text{rs1fsv } \%f \text{ fr!} : \text{rs1fs } \langle \text{fseqnv } 10d \rangle \text{ is rs1} = \text{fr}$
 $\text{rs2fsv } \%f \text{ fr!} : \text{rs2fs } \langle \text{fseqnv } 10d \rangle \text{ is rs2} = \text{fr}$
 $\text{rdfsv } \%f \text{ fr!} : \text{rdfs } \langle \text{fseqnv } 10d \rangle \text{ is rd} = \text{fr}$

Defines:

rdfs , used in chunks 24a, 29–31, 39a, and 46a.

rdfsv , never used.

rs1fs , used in chunks 31a, 39a, and 46a.

rs2fs , used in chunks 29–31, 39a, and 46a.

rs1fsv , never used.

rs2fsv , never used.

Uses $\text{rd } 3c$, $\text{rs1 } 4a$, and $\text{rs2 } 4a$.

The "%f" is for generating assembly language for testing purposes, and for disassembly. Also, the ! specify that the field is signed: it stops the toolkit complaining; the equation (fseqnv) ensures that only positive fields are accepted/generated anyway.

The double and quad encodings are less simple.

Firstly, we define some helper equations: fr is the decoded register value, which must be in range and even (frdqlegal); fhi and flo are temporary variables which hold the encoded parts of the register (frdqtemp).

11b $\langle \text{frdqlegal } 11b \rangle \equiv$ (11)
 $\text{fr}[6:31] = 0, \text{fr}[0] = 0$

11c $\langle \text{frdqtemp } 11c \rangle \equiv$ (11)
 $\text{fhi} = \text{fr}[1:4], \text{flo} = \text{fr}[5]$

The next three chunks define the equations that satisfy double, valid-quad, and invalid-quad encoding respectively (fdeqnv , fseqnv , fseqni).

11d $\langle \text{fdeqnv } 11d \rangle \equiv$ (12b)
 $\{ \langle \text{frdqlegal } 11b \rangle, \langle \text{frdqtemp } 11c \rangle \}$

11e $\langle \text{fseqnv } 11e \rangle \equiv$ (12b)
 $\{ \langle \text{frdqlegal } 11b \rangle, \langle \text{frdqtemp } 11c \rangle, \text{fr}[1] = 0 \}$

11f $\langle \text{fseqni } 11f \rangle \equiv$ (12b)
 $\{ \langle \text{frdqlegal } 11b \rangle, \langle \text{frdqtemp } 11c \rangle, \text{fr}[1] = 1 \}$

In order to simplify the actual field assignments, we define some helper fields which cover the standard registers.

11g $\langle \text{types-fpre-fields.spec } 11g \rangle \equiv$
 $\# \text{ types-fpre-fields.spec}$
 $\# \text{ this goes after fields.spec (in fields declaration)}$
 $\text{rs1fhi } 15:18 \text{ rs1flo } 14:14$
 $\text{rs2fhi } 1:4 \text{ rs2flo } 0:0$
 $\text{rdfhi } 26:29 \text{ rdflo } 25:25$

Defines:

rdfhi , used in chunk 12a.

rdflo , used in chunk 12a.

rs1fhi , used in chunk 11h.

rs2fhi , used in chunk 11i.

rs1flo , used in chunk 11h.

rs2flo , used in chunk 11i.

These last three chunks do the required field assignments from the temporaries $\text{f}\{\text{hi}, \text{lo}\}$ to the actual fields $\text{r}\{\text{s}\{1, 2\}, \text{d}\}\text{f}\{\text{hi}, \text{lo}\}$.

11h $\langle \text{rs1fass } 11h \rangle \equiv$ (12b)
 $\text{rs1fhi} = \text{fhi} \ \& \ \text{rs1flo} = \text{flo}$
 Uses $\text{rs1fhi } 11g$ and $\text{rs1flo } 11g$.

11i $\langle \text{rs2fass } 11i \rangle \equiv$ (12b)
 $\text{rs2fhi} = \text{fhi} \ \& \ \text{rs2flo} = \text{flo}$
 Uses $\text{rs2fhi } 11g$ and $\text{rs2flo } 11g$.

12a $\langle rdfass\ 12a \rangle \equiv$ (12b)
`rdfhi = fhi & rdflo = flo`
 Uses `rdfhi` 11g and `rdflo` 11g.

All that remains is to specify the constructors for double and quad registers.

12b $\langle fpre\ type\ specifications\ 11a \rangle + \equiv$ (10c) <11a
`constructors`
`rs1fdv "%f" fr! : rs1fd <fdeqnv 11d> is <rs1fass 11h>`
`rs2fdv "%f" fr! : rs2fd <fdeqnv 11d> is <rs2fass 11i>`
`rdfdv "%f" fr! : rdfd <fdeqnv 11d> is <rdfass 12a>`

`rs1fqv "%f" fr! : rs1fq <fqeqnv 11e> is <rs1fass 11h>`
`rs2fqv "%f" fr! : rs2fq <fqeqnv 11e> is <rs2fass 11i>`
`rdfqv "%f" fr! : rdfq <fqeqnv 11e> is <rdfass 12a>`

`rs1fqi "%f" fr! : rs1fq <fqeqni 11f> is <rs1fass 11h>`
`rs2fqi "%f" fr! : rs2fq <fqeqni 11f> is <rs2fass 11i>`
`rdfqi "%f" fr! : rdfq <fqeqni 11f> is <rdfass 12a>`

Defines:

`rdfd`, used in chunks 24a, 29–31, 39a, and 46a.
`rdfdv`, never used.
`rdfq`, used in chunks 24a and 29–31.
`rdfqi`, never used.
`rdfqv`, never used.
`rs1fd`, used in chunks 31a, 39a, and 46a.
`rs2fd`, used in chunks 29–31, 39a, and 46a.
`rs1fdv`, never used.
`rs2fdv`, never used.
`rs1fq`, used in chunk 31a.
`rs2fq`, used in chunks 29–31.
`rs1fqi`, never used.
`rs2fqi`, never used.
`rs1fqv`, never used.
`rs2fqv`, never used.

3.3.1 Non-decoding floating-point registers

If we do not want to use the floating-point encoding types (for example, to speed up generation of decoders) then we can specify alternate fields instead, which are equivalent to the integer registers. Since these are true fields and not typed constructors, their values can be used immediately in matching statements.

12c $\langle types\ -nofpre\ -fields\ .spec\ 12c \rangle \equiv$
`# types-nofpre-fields.spec`
`# this goes after fields.spec (in fields declaration)`
`# incompatible with types-fpre*.spec`
`rs1fs 14:18 rs2fs 0:4 rdfs 25:29`
`rs1fd 14:18 rs2fd 0:4 rdfd 25:29`
`rs1fq 14:18 rs2fq 0:4 rdfq 25:29`

Defines:

`rdfd`, used in chunks 24a, 29–31, 39a, and 46a.
`rdfq`, used in chunks 24a and 29–31.
`rdfs`, used in chunks 24a, 29–31, 39a, and 46a.
`rs1fd`, used in chunks 31a, 39a, and 46a.
`rs2fd`, used in chunks 29–31, 39a, and 46a.
`rs1fq`, used in chunk 31a.
`rs2fq`, used in chunks 29–31.
`rs1fs`, used in chunks 31a, 39a, and 46a.
`rs2fs`, used in chunks 29–31, 39a, and 46a.

Clearly, if we want to use the values of the registers, then we need to transform them into floating-point register values, as described above.

4 Opcodes

13a `<core.spec 13a>≡`
`# core.spec`
`# requires fields.spec and types.spec`
`<pattern and constructor specifications 13d>`

We also use two additional files, one that contains specifications for simulation, and one that contains the full specifications for usual purposes; these files cannot be used together.

13b `<core-sim.spec 13b>≡`
`# core-sim.spec`
`# goes after core.spec (incompatible with core-full.spec)`
`<sim pattern and constructor specifications 23d>`

13c `<core-full.spec 13c>≡`
`# core-full.spec`
`# goes after core.spec (incompatible with core-sim.spec)`
`<full pattern and constructor specifications 23c>`

The following opcode tables are derived from the tables in Appendix E of the SPARC-V9 manual [3].

Where an entry in a table refers to another table, we define a pattern with the name of that table (e.g., TABLE_31). That pattern is not useful by itself, but is used to define more opcodes in a pattern-binding statement that resembles the eponymous table.

The SPARC-V9 opcode tables are organized hierarchically; the first table in Appendix E is at the top of the hierarchy, and it has four entries corresponding to the four possible values of the (two-bit) `op` field. Only one of these entries (CALL) is an opcode; the others refer the reader to subsequent tables.

13d `<pattern and constructor specifications 13d>≡` (13a) 13e>
`patterns`
`[TABLE_31 CALL TABLE_32 TABLE_33] is op = {0 to 3}`

Defines:

CALL, used in chunk 28.
 TABLE_31, used in chunk 13e.
 TABLE_32, used in chunk 15.
 TABLE_33, used in chunk 18.

Uses `op 3b`.

TABLE_{31, 32, 33} are further reduced in sections 4.1, 4.2 and 4.3 respectively.

4.1 Branches and related opcodes

Table 31 is short, but it presents an oddity: an opcode with two names. SETHI means NOP when `rd` and `imm22` are zero. On the MIPS, no-ops were treated as synthetic instructions, but here we define NOP as a separate opcode, reflecting the presentation in the manual.

13e `<pattern and constructor specifications 13d>+≡` (13a) <13d 14>
`patterns`
`[ILLTRAP BPcc Bicc BPrx SETHI FBPFcc FBfcc]`
`is TABLE_31 & op2 = {0 to 6}`
`NOP is SETHI & rd = 0 & imm22 = 0`

Defines:

Bicc, used in chunk 27.
 BPcc, used in chunk 27.
 BPrx, used in chunk 14.
 FBfcc, used in chunk 27.
 FBPFcc, used in chunk 27.
 ILLTRAP, used in chunks 27a, 31c, and 53.
 NOP, used in chunks 31b and 53.
 SETHI, used in chunks 31d and 53.

Uses `imm22 3c`, `op2 3c`, `rd 3c`, and TABLE_31 13d.

Since `ILLTRAP` is defined to always produce an `illegal_instruction` trap, and because of our deliberate design decision to avoid matching instructions that are illegal, we could possibly not match it here. We do match it to make placeholders easier to define.

`BPrx` is a placeholder for the `BPr` instructions, which also require a particular bit to be zero:

```
14 <pattern and constructor specifications 13d> +≡ (13a) <13e 15>
    patterns
    BPr is BPrx & mbz_f2 = 0
```

Defines:

`BPr`, used in chunks 29 and 53.
Uses `BPrx` 13e and `mbz_f2` 3c.

Table 36 includes the branch and trap opcodes (the `Tcc` pattern is defined below in section 4.2). The rows of each table vary with the values of the `{i, f}cond_f2` fields but they are similar in many ways, and so we have aggressively reduced the table into two patterns based on the suffixes to the names; to reduce namespace pollution (and to allow overloading) we use `fieldinfo` to label the values of `{i, f}cond_f2`. These names are defined in section 2.2. This reduction means that all the production occurs in the constructors: see section 5.4. Since all the patterns have been reduced in preceding sections nothing more needs to be done.

Table 37 shows the encoding of the `rcond` instruction field for the integer register condition opcodes (`BPr`, `MOVr` and `FMOVr`; the latter two are defined below in sections 4.2 and 4.4). Like the `cond` fields for the branch and trap opcodes, the value of the `rcond` field depends on a suffix of the opcode so we can use the same `fieldinfo/constructor` trick. The fields are named in section 2.2. Unlike the branch opcodes, the `rcond` field is not fully populated, so the illegal values (0 and 4, named as “0!” and “4!”) must be disallowed in the constructors: see section 5.5.

4.2 Arithmetic and logical opcodes

Table 32 includes most of the arithmetic and logical opcodes. The `_`'s are illegal. Any patterns suffixed with an `x` must be reduced further (below).

15 $\langle \text{pattern and constructor specifications } 13d \rangle + \equiv$ (13a) $\langle 14 \ 16 \rangle$

```

patterns
[ ADD  ADDcc  TADDcc  WRxxx
  AND  ANDcc  TSUBcc  SAVEDx
  OR   ORcc   TADDccTV WRPRx
  XOR  XORcc  TSUBccTV _
  SUB  SUBcc  MULSc  FPop1
  ANDN ANDNcc  SLLx   FPop2
  ORN  ORNcc  SRLx   IMPDEP1
  XNOR XNORcc  SRAX   IMPDEP2
  ADDC ADDCcc  RDxxx   JMPL
  MULX _      _      RETURN
  UMUL UMULcc  RDPRx   Tcc
  SMUL SMULcc  FLUSHW  FLUSH
  SUBC SUBCcc  MOVcc   SAVE
  UDIVX _      SDIVX   RESTORE
  UDIV UDIVcc  POPCx   DONEx
  SDIV SDIVcc  MOVr    _      ]
is TABLE_32 & op3 = {0 to 63 columns 4}

```

Defines:

ADD, used in chunks 26c, 47, and 53.
 ADDC, used in chunks 26c and 53.
 ADDcc, used in chunks 26c, 47, and 53.
 ADDCcc, used in chunks 26c and 53.
 AND, used in chunks 26c and 53.
 ANDcc, used in chunks 26c, 47, and 53.
 ANDN, used in chunks 26c, 47, and 53.
 ANDNcc, used in chunks 26c and 53.
 DONEx, used in chunk 17a.
 FLUSH, used in chunks 31c and 53.
 FLUSHW, used in chunks 31b and 53.
 FPop1, used in chunk 20.
 FPop2, used in chunk 22a.
 IMPDEP1, used in chunks 33b and 34a.
 IMPDEP2, used in chunk 46a.
 JMPL, used in chunks 31c, 47, and 53.
 MOVcc, used in chunk 28.
 MOVr, used in chunk 17e.
 MULSc, used in chunks 26c and 53.
 MULX, used in chunks 26c and 53.
 OR, used in chunks 26c, 47, and 53.
 ORcc, used in chunks 26c, 47, and 53.
 ORN, used in chunks 26c and 53.
 ORNcc, used in chunks 26c and 53.
 POPCx, used in chunk 17d.
 RDPRx, used in chunk 17c.
 RDxxx, used in chunk 16.
 RESTORE, used in chunks 26c, 47, and 53.
 RETURN, used in chunks 31c and 53.
 SAVE, used in chunks 26c, 47, and 53.
 SAVEDx, used in chunk 17a.
 SDIV, used in chunks 26c and 53.
 SDIVcc, used in chunks 26c and 53.
 SDIVX, used in chunks 26c and 53.
 SLLx, used in chunk 17b.
 SMUL, used in chunks 26c and 53.
 SMULcc, used in chunks 26c and 53.
 SRAX, used in chunk 17b.
 SRLx, used in chunk 17b.
 SUB, used in chunks 26c, 47, and 53.

SUBC, used in chunks 26c and 53.
 SUBcc, used in chunks 26c, 47, and 53.
 SUBccc, used in chunk 26c.
 TADDcc, used in chunks 26c and 53.
 TADDccTV, used in chunks 26c and 53.
 Tcc, used in chunk 28.
 TSUBcc, used in chunks 26c and 53.
 TSUBccTV, used in chunks 26c and 53.
 UDIV, used in chunks 26c and 53.
 UDIVcc, used in chunks 26c and 53.
 UDIVX, used in chunks 26c and 53.
 UMUL, used in chunks 26c and 53.
 UMULcc, used in chunks 26c and 53.
 WRPRx, used in chunk 17c.
 WRxxx, used in chunk 16.
 XNOR, used in chunks 26c, 47, and 53.
 XNORcc, used in chunks 26c and 53.
 XOR, used in chunks 26c, 47, and 53.
 XORcc, used in chunks 26c and 53.
 Uses op3 4a and TABLE_32 13d.

We have used WRxxx and RDxxx to stand for the groups of opcodes that appear in the corresponding positions in Table 32. These opcodes define variants of the wr and rd instructions, which are overloaded. The overloading is resolved by looking at the values of operands, as shown by the details in Table 32. It probably would have been simpler to specify these purely as synthetic instructions, but we've chosen to be slaves to the SPARC-V9 manual. The _'s are illegal. The {RD,WR}ASR constructors define which values are legal.

16 \langle pattern and constructor specifications 13d $\rangle + \equiv$ (13a) \langle 15 17a \rangle

```

patterns
[ WRY _ WRCCR WRASI _ _ WRFPRS ]
    is WRxxx & rd = {0 to 6}
WRASR is WRxxx
SIR   is WRxxx & rd = 15 & rs1 = 0 & i = 1
[ RDY _ RDCCR RDASI RDTICK RDPC RDFPRS ]
    is RDxxx & rs1 = {0 to 6} & i = 0
RDASR is RDxxx & i = 0
[ STBAR MEMBAR ]
    is RDxxx & rs1 = 15 & rd = 0 & i = [0 1]
  
```

Defines:

MEMBAR, used in chunks 25a and 53.
 RDASI, used in chunks 25a and 53.
 RDASR, used in chunks 25, 32b, 42, and 53.
 RDCCR, used in chunks 25a and 53.
 RDFPRS, used in chunks 25a and 53.
 RDPC, used in chunks 25a and 53.
 RDTICK, used in chunks 25a and 53.
 RDY, used in chunks 25a and 53.
 SIR, used in chunks 25a and 53.
 STBAR, used in chunks 25a and 53.
 WRASI, used in chunks 25a and 53.
 WRASR, used in chunks 25, 33a, 44, and 53.
 WRCCR, used in chunk 25a.
 WRFPRS, used in chunks 25a and 53.
 WRY, used in chunks 25a and 53.
 Uses i 4a, rd 3c, RDxxx 15, rs1 4a, and WRxxx 15.

SAVED_x represents the SAVED and RESTORED instructions, which depend on fcn. Similarly, DONE_x represents the DONE and RETRY instructions. All other patterns of fcn are illegal. Note that DONE and RETRY are illegal if TL = 0 (but that's not specifiable here since that's a runtime constraint).

17a \langle pattern and constructor specifications 13d $\rangle + \equiv$ (13a) \langle 16 17b \rangle

```
patterns
  [ SAVED RESTORED ] is SAVEDx & fcn = [ 0 1 ]
  [ DONE  RETRY    ] is DONEx  & fcn = [ 0 1 ]
```

Defines:

DONE, used in chunks 31b and 53.
 RESTORED, used in chunks 31b and 53.
 RETRY, used in chunks 31b and 53.
 SAVED, used in chunks 31b and 53.

Uses DONE_x 15, fcn 4a, and SAVED_x 15.

SLL_x, SRL_x and SRA_x are shifts on double or single words depending on x.

17b \langle pattern and constructor specifications 13d $\rangle + \equiv$ (13a) \langle 17a 17c \rangle

```
patterns
  [ SLL SLLX ] is SLLx & x = [ 0 1 ]
  [ SRL SRLX ] is SRLx & x = [ 0 1 ]
  [ SRA SRAX ] is SRAx & x = [ 0 1 ]
```

Defines:

SLL, used in chunks 26c and 53.
 SLLX, used in chunks 26c and 53.
 SRA, used in chunks 26c, 47, and 53.
 SRAX, used in chunks 26c and 53.
 SRL, used in chunks 26c, 47, and 53.
 SRLX, used in chunks 26c and 53.

Uses SLL_x 15, SRA_x 15, SRL_x 15, and x 4a.

RDPR_x and WRPR_x depend on values of encoded registers being valid for the corresponding {RD, WR}PR opcode. Reads are legal for registers in the range 0 . . 15, and 31, and writes are legal for registers in the range 0 . . 14; unfortunately, due to restrictions in the toolkit we cannot check these patterns so we add some equations to the constructors in section 5.2. The registers rslp and rdpr are versions of rsl and rd that have special names related to the registers (see section 2.2).

17c \langle pattern and constructor specifications 13d $\rangle + \equiv$ (13a) \langle 17b 17d \rangle

```
patterns
  RDPR is RDPRx
  WRPR is WRPRx
```

Defines:

RDPR, used in chunks 26 and 53.
 WRPR, used in chunks 26 and 53.

Uses RDPR_x 15 and WRPR_x 15.

POPC_x is a placeholder for POPC which is only defined if rsl is 0. All other encodings of rsl are illegal. Note that all known implementations of SPARC-V9 do not implement POPC in hardware, but generate illegal instruction and are emulated in kernel software.

17d \langle pattern and constructor specifications 13d $\rangle + \equiv$ (13a) \langle 17c 17e \rangle

```
patterns
  POPC is POPCx & rsl = 0
```

Defines:

POPC, used in chunks 26c and 53.

Uses POPC_x 15 and rsl 4a.

MOV_r is the name for the family of MOV_R opcodes. For this specification, they are the same thing.

17e \langle pattern and constructor specifications 13d $\rangle + \equiv$ (13a) \langle 17d 18 \rangle

```
patterns
  MOVR is MOVr
```

Defines:

MOV_R, used in chunk 30.

Uses MOV_r 15.

4.3 Load and store opcodes

Table 33 includes the load and store opcodes. The `_`'s are illegal. Patterns suffixed with an `x` are reduced further below.

18 $\langle \text{pattern and constructor specifications } 13d \rangle + \equiv$ (13a) $\langle 17e \ 19a \rangle$

```

patterns
[ LDUW  LDUWA  LDF      LDFA
  LDUB  LDUBA  LDFSRx   _
  LDUH  LDUHA  LDQF     LDQFA
  LDD   LDDA   LDDF     LDDFA
  STW   STWA   STF      STFA
  STB   STBA   STF SRx  _
  STH   STHA  STQF     STQFA
  STD   STDA  STDF     STDFA
  LDSW  LDSWA  _        _
  LDSB  LDSBA  _        _
  LDSH  LDSHA  _        _
  LDX   LDXA  _        _
  _     _     _        CASA
  LDSTUB LDSTUBA PREFETCHx PREFETCHAx
  STX   STXA  _        CASXA
  SWAP  SWAPA  _        _
]
is TABLE_33 & op3 = {0 to 63 columns 4}

```

Defines:

CASA, used in chunks 19c and 47.
CASXA, used in chunks 19c, 47, and 53.
LDD, used in chunks 23 and 53.
LDDA, used in chunks 23 and 53.
LDDF, used in chunks 24a and 53.
LDDFA, used in chunks 24a and 53.
LDF, used in chunks 24a and 53.
LDFA, used in chunks 24a and 53.
LDFSRx, used in chunk 19a.
LDQF, used in chunks 24a and 53.
LDQFA, used in chunks 24a and 53.
LDSB, used in chunk 22b.
LDSBA, used in chunks 22b and 53.
LDSH, used in chunks 22b and 53.
LDSHA, used in chunks 22b and 53.
LDSTUB, used in chunks 22b and 53.
LDSTUBA, used in chunks 22b and 53.
LDSW, used in chunks 22b and 53.
LDSWA, used in chunks 22b and 53.
LDUB, used in chunks 22b and 53.
LDUBA, used in chunks 22b and 53.
LDUH, used in chunks 22b and 53.
LDUHA, used in chunks 22b and 53.
LDUW, used in chunks 22b and 53.
LDUWA, used in chunks 22b and 53.
LDX, used in chunks 22b, 50b, and 53.
LDXA, used in chunks 22b and 53.
PREFETCHAx, used in chunk 19b.
PREFETCHx, used in chunk 19b.
STB, used in chunks 22b, 47, and 53.
STBA, used in chunks 22b and 53.
STD, used in chunks 23 and 53.
STDA, used in chunks 23 and 53.
STDF, used in chunks 24a and 53.
STDFA, used in chunks 24a and 53.
STF, used in chunks 24a and 53.
STFA, used in chunks 24a and 53.
STFSRx, used in chunk 19a.
STH, used in chunks 22b, 47, and 53.
STHA, used in chunks 22b and 53.

STQF, used in chunks 24a and 53.
 STQFA, used in chunks 24a and 53.
 STW, used in chunks 22b, 47, and 53.
 STWA, used in chunks 22b and 53.
 STX, used in chunks 22b, 47, 50b, and 53.
 STXA, used in chunks 22b and 53.
 SWAP, used in chunks 22b and 53.
 SWAPA, used in chunks 22b and 53.
 Uses `op3_4a` and `TABLE_33_13d`.

Actual LDFSRx and STFSRx instructions depend on `rd`. All other encodings of `rd` are illegal. We group the `LD{,X}FSR` and `ST{,X}FSR` instructions together for generating constructors in section 5.1.

19a \langle *pattern and constructor specifications* 13d $\rangle + \equiv$ (13a) \langle 18 19b \rangle

```
patterns
  ldfsr is any of [ LDFSR LDXFSR ],
  which is LDFSRx & rd = [ 0 1 ]
  stfsr is any of [ STFSR STXFSR ],
  which is STFSRx & rd = [ 0 1 ]
```

Defines:

LDFSR, used in chunk 53.
 ldfsr, used in chunk 24b.
 LDXFSR, used in chunk 53.
 stfsr, used in chunk 24b.
 STFSR, used in chunk 53.
 STXFSR, used in chunk 53.
 Uses LDFSRx 18, `rd_3c`, and STFSRx 18.

The PREFETCH and PREFETCHA instructions depend on `fcn`. `fcn` values in the ranges 5 . . 15 are reserved and are illegal. `fcn` values in the range 16 . . 31 are implementation-dependent (and are NOPs if not defined). These checks are performed in the constructors (section 5.1).

19b \langle *pattern and constructor specifications* 13d $\rangle + \equiv$ (13a) \langle 19a 19c \rangle

```
patterns
  [ PREFETCH PREFETCHA ] is
  [ PREFETCHx PREFETCHAx ]
```

Defines:

PREFETCH, used in chunks 24 and 53.
 PREFETCHA, used in chunks 24 and 53.
 Uses PREFETCHAx 18 and PREFETCHx 18.

We group the CASA and CASXA instructions together because they have the same syntax.

19c \langle *pattern and constructor specifications* 13d $\rangle + \equiv$ (13a) \langle 19b 20 \rangle

```
patterns
  casa is CASA | CASXA
```

Defines:

casa, used in chunk 23e.
 Uses CASA 18 and CASXA 18.

4.4 Floating-point opcodes

Unlike the SPARC-V8 specification, the SPARC-V9 specification of the floating-point arithmetic and conversion opcodes is a full table (Table 34). We use the SPARC-V8 way here, since the table is very similar (only a few extra ops). We've also chosen to divide the opcodes into two groups: two-operand and three-operand instructions.

We further divide the opcodes based on their operand types. This division makes it easy to use the names `float2`, `float3s`, etc. to refer to the groups, without having to re-enumerate the members of each group; these names are used later in the specification to specify complete instructions.

20 *<pattern and constructor specifications 13d>+≡* (13a) <19c 22a>

```
patterns
float2 is any of
  [ FMOVs FMOVd FMOVq FNEGs FNEGd FNEGq
    FABSSs FABSSd FABSSq FSQRTs FSQRTd FSQRTq
    FSTOx FdTOx FqTOx FxTOs FxTOd FxTOq
    FiTOs FdTOs FqTOs FiTOd FSTOd FqTOd
    FiTOq FSTOq FdTOq FSTOi FdTOi FqTOi ],
which is FPopl & opf =
  [ 0x1  0x2  0x3  0x5  0x6  0x7
    0x9  0xa  0xb  0x29 0x2a 0x2b
    0x81 0x82 0x83 0x84 0x88 0x8c
    0xc4 0xc6 0xc7 0xc8 0xc9 0xcb
    0xcc 0xcd 0xce 0xd1 0xd2 0xd3 ]
```

```
funarys is FMOVs | FNEGs | FABSSs | FSQRTs
funaryd is FMOVd | FNEGd | FABSSd | FSQRTd
funaryq is FMOVq | FNEGq | FABSSq | FSQRTq
```

```
f2sTOs is funarys | FSTOi | FiTOs
f2sTOd is FSTOx | FiTOd | FSTOd
f2sTOq is FiTOq | FSTOq
f2dTOs is FxTOs | FdTOi | FdTOs
f2dTOd is funaryd | FdTOx | FxTOd
f2dTOq is FxTOq | FdTOq
f2qTOs is FqTOi | FqTOs
f2qTOd is FqTOx | FqTOd
f2qTOq is funaryq
```

```
float3 is any of
  [ FADDs FADDd FADDq FSUBs FSUBd FSUBq
    FMULs FMULd FMULq FDIVs FDIVd FDIVq
    FsmULd FdMULq ],
which is FPopl & opf =
  [ 0x41 0x42 0x43 0x45 0x46 0x47
    0x49 0x4a 0x4b 0x4d 0x4e 0x4f
    0x69 0x6e ]
float3s is FADDs | FSUBs | FMULs | FDIVs
float3d is FADDd | FSUBd | FMULd | FDIVd
float3q is FADDq | FSUBq | FMULq | FDIVq
```

Defines:

```
FABSSd, used in chunk 53.
FABSSq, used in chunk 53.
FABSSs, used in chunk 53.
FADDd, never used.
FADDq, used in chunk 53.
FADDs, used in chunk 53.
FDIVd, used in chunk 53.
FDIVq, used in chunk 53.
```

FDIVs, used in chunk 53.
FdMULq, used in chunks 31a and 53.
f2dTOD, used in chunk 30d.
FdTOi, used in chunk 53.
f2dTOq, used in chunk 30d.
FdTOq, used in chunk 53.
f2dTOS, used in chunk 30d.
FdTOS, used in chunk 53.
FdTOx, used in chunk 53.
FiTOD, used in chunk 53.
FiTOq, used in chunk 53.
FiTOS, never used.
float2, never used.
float3, never used.
float3d, used in chunk 31a.
float3q, used in chunk 31a.
float3s, used in chunk 31a.
FMOVd, used in chunk 53.
FMOVq, used in chunk 53.
FMOVs, used in chunk 53.
FMULD, used in chunk 53.
FMULq, used in chunk 53.
FMULs, used in chunk 53.
FNEGd, used in chunk 53.
FNEGq, used in chunk 53.
FNEGs, used in chunk 53.
f2qTOD, used in chunk 30d.
FqTOD, used in chunk 53.
FqTOi, used in chunk 53.
f2qTOq, used in chunk 30d.
f2qTOS, used in chunk 30d.
FqTOS, used in chunk 53.
FqTOx, used in chunk 53.
FSMULD, used in chunks 31a and 53.
FSQRTd, used in chunk 53.
FSQRTq, used in chunk 53.
FSQRTs, used in chunk 53.
f2sTOD, used in chunk 30d.
FsTOD, used in chunk 53.
FsTOi, used in chunk 53.
f2sTOq, used in chunk 30d.
FsTOq, used in chunk 53.
f2sTOS, used in chunk 30d.
FsTOx, used in chunk 53.
FSUBd, used in chunk 53.
FSUBq, used in chunk 53.
FSUBs, used in chunk 53.
funaryd, never used.
funaryq, never used.
funarys, never used.
FxTOD, used in chunk 53.
FxTOq, used in chunk 53.
FxTOS, used in chunk 53.
Uses FPop1 15 and opf 4a.

Table 35 includes the floating-point comparison opcodes, but it is too explicit, expanding out fields that are not necessary; we can use the `opf_low{5,6}` fields to get all the information we need.

22a *(pattern and constructor specifications 13d)*+≡ (13a) <20 22b>

```

patterns
  fcompares is any of      [ FCMPs FCMPEs ],
    which is FPop2 & opf = [ 0x51 0x55 ] & mbz_f3 = 0
  fcompared is any of     [ FCMPd FCMPEd ],
    which is FPop2 & opf = [ 0x52 0x56 ] & mbz_f3 = 0
  fcompareq is any of    [ FCMPq FCMPEq ],
    which is FPop2 & opf = [ 0x53 0x57 ] & mbz_f3 = 0

  [ FMOVScC FMOVDcC FMOVQcC ]
    is FPop2 & opf_low6 = { 1 to 3 } & mbz_f4_18 = 0

  [ FMOVSR FMOVDR FMOVQR ]
    is FPop2 & opf_low5 = { 5 to 7 } & mbz_f4_13 = 0

```

Defines:

```

FCMPd, used in chunk 53.
FCMPEd, used in chunk 53.
FCMPEq, used in chunk 53.
FCMPq, used in chunk 53.
fcompared, used in chunk 31a.
fcompareq, used in chunk 31a.
fcompares, used in chunk 31a.
FCPEs, never used.
FCPs, never used.
FMOVDcC, used in chunk 29.
FMOVDR, used in chunk 30.
FMOVQcC, used in chunk 29.
FMOVQR, used in chunk 30.
FMOVScC, used in chunk 29.
FMOVSR, used in chunk 30.

```

Uses FPop2 15, mbz_f3 4a, mbz_f4_13 4b, mbz_f4_18 4b, opf 4a, opf_low5 4b, and opf_low6 4b.

5 Constructors

5.1 Load and store constructors

We group the load and store opcodes by their assembly syntax. These groupings reflect the information provided on pages 178–183 for the load instructions and pages 229–232 for the stores. Each group uses a different set of names for their register operands. Those names are defined below.

22b *(pattern and constructor specifications 13d)*+≡ (13a) <22a 23a>

```

patterns
  loadg is LDSB | LDSH | LDSW | LDUB | LDUH | LDUW | LDX |
    LDSTUB | SWAP
  loada is LDSBA | LDSHA | LDSWA | LDUBA | LDUHA | LDUWA | LDXA |
    LDSTUBA | SWAPA
  storeg is STB | STH | STW | STX
  storea is STBA | STHA | STWA | STXA

```

Defines:

```

loada, used in chunk 23b.
loadg, used in chunk 23a.
storea, used in chunk 23b.
storeg, used in chunk 23a.

```

Uses LDSB 18, LDSBA 18, LDSH 18, LDSHA 18, LDSTUB 18, LDSTUBA 18, LDSW 18, LDSWA 18, LDUB 18, LDUBA 18, LDUH 18, LDUHA 18, LDUW 18, LDUWA 18, LDX 18, LDXA 18, STB 18, STBA 18, STH 18, STHA 18, STW 18, STWA 18, STX 18, STXA 18, SWAP 18, and SWAPA 18.

The constructors for the load and store instructions illustrate the use of the typed constructor `address_` as an operand. Addresses are bracketed, as in SPARC-V9 assembly language. Because the constructors' output patterns are the conjunctions of the opcodes and operands, they can be omitted.

23a \langle *pattern and constructor specifications 13d* $\rangle + \equiv$ (13a) \langle 22b 23b \rangle
 constructors
 loadg [address_], rd
 storeg rd, [address_]

Uses `address_9a`, `loadg 22b`, `rd 3c`, and `storeg 22b`.

The `loada` and `storea` only accept an `asi_address` (constructor defined in section 3.2).

23b \langle *pattern and constructor specifications 13d* $\rangle + \equiv$ (13a) \langle 23a 23e \rangle
 constructors
 loada asi_address, rd
 storea rd, asi_address

Uses `asi_address 10a`, `loada 22b`, `rd 3c`, and `storea 22b`.

Loads and stores of double words (for the deprecated LDD, STD, LDDA and STDA) require even-odd register pairs for the destination and source registers; their equations specify that constraint. We move these constructors to another file so we can have different specifications for simulation.

23c \langle *full pattern and constructor specifications 23c* $\rangle \equiv$ (13c) 24c \rangle
 constructors
 LDD [address_], rd { rd = 2 * _ }
 STD rd, [address_] { rd = 2 * _ }
 LDDA asi_address, rd { rd = 2 * _ }
 STDA rd, asi_address { rd = 2 * _ }

Uses `address_9a`, `asi_address 10a`, `LDD 18`, `LDDA 18`, `rd 3c`, `STD 18`, and `STDA 18`.

For simulation, we don't check the register pairing (it is checked later). (is this correct??? if not, just use the full version)

23d \langle *sim pattern and constructor specifications 23d* $\rangle \equiv$ (13b) 24d \rangle
 constructors
 LDD [address_], rd
 STD rd, [address_]
 LDDA asi_address, rd
 STDA rd, asi_address

Uses `address_9a`, `asi_address 10a`, `LDD 18`, `LDDA 18`, `rd 3c`, `STD 18`, and `STDA 18`.

CASA and CASXA have their own special syntax, using all three registers explicitly, and an optional immediate `asi` (`casa_asi`).

23e \langle *pattern and constructor specifications 13d* $\rangle + \equiv$ (13a) \langle 23b 24a \rangle
 constructors
 casa [rs1] casa_asi, rs2, rd

Uses `casa 19c`, `casa_asi 10b`, `rd 3c`, `rs1 4a`, and `rs2 4a`.

Floating-point loads and stores use the encoded forms of register access. In SPARC-V8 LDDF and STDF required an even register to store into; this is handled in SPARC-V9 by the encoding mechanism.

24a *(pattern and constructor specifications 13d)* +≡ (13a) <23e 24b>

```
constructors
  LDF  [address_], rdfs
  LDDF [address_], rdfd
  LDQF [address_], rdfq
  STF  rdfs, [address_]
  STDF rdfd, [address_]
  STQF rdfq, [address_]
  LDFA asi_address, rdfs
  LDDFA asi_address, rdfd
  LDQFA asi_address, rdfq
  STFA rdfs, asi_address
  STDFA rdfd, asi_address
  STQFA rdfq, asi_address
```

Uses address_9a, asi_address 10a, LDDF 18, LDDFA 18, LDF 18, LDFA 18, LDQF 18, LDQFA 18, rdfd 12b 12c, rdfq 12b 12c, rdfs 11a 12c, STDF 18, STDFA 18, STF 18, STFA 18, STQF 18, and STQFA 18.

The SPARC-V9 also has a couple of specialized load and store instructions, related to the floating-point state register (FSR). The registers loaded and stored are implicit in the opcodes, but we put them into the specifications as “assembly-language syntax.” This technique helps us generate a disassembler, but more importantly, it makes the specification easier to read and understand.

24b *(pattern and constructor specifications 13d)* +≡ (13a) <24a 25a>

```
constructors
  ldfsr [address_], "%fsr"
  stfsr "%fsr", [address_]
```

Uses address_9a, ldfsr 19a, and stfsr 19a.

The PREFETCH{ ,A } instructions look like loads, but without any observable effects. Due to restrictions in the toolkit, the checks on fcn must be performed here instead of in the patterns (the equations must be commented out when doing checking). We do not do the checking in simulation.

24c *(full pattern and constructor specifications 23c)* +≡ (13c) <23c 25b>

```
constructors
  PREFETCH [address_], fcn
    when { fcn < 5 } is PREFETCH & address_ & fcn
    when { fcn > 15 } is PREFETCH & address_ & fcn
  PREFETCHA asi_address, fcn
    when { fcn < 5 } is PREFETCHA & asi_address & fcn
    when { fcn > 15 } is PREFETCHA & asi_address & fcn
```

Uses address_9a, asi_address 10a, fcn 4a, PREFETCH 19b, and PREFETCHA 19b.

24d *(sim pattern and constructor specifications 23d)* +≡ (13b) <23d 25c>

```
constructors
  PREFETCH [address_], fcn
  PREFETCHA asi_address, fcn
```

Uses address_9a, asi_address 10a, fcn 4a, PREFETCH 19b, and PREFETCHA 19b.

5.2 Read and write constructors

The RDxxx and WRxxx group of instructions move information between special-purpose registers and general-purpose registers, instead of special-purpose registers and memory (unlike {ld, st}f_{sr}). Several other miscellaneous instructions are also defined by the same opcodes and are defined here (it is not clear whether `simm13` in SIR is signed or not, despite its name, nor what it is used for, but we assume it is signed since that is what the `s` in `simm13` stands for!).

25a \langle pattern and constructor specifications 13d $\rangle + \equiv$ (13a) \langle 24b 26c \rangle

```
constructors
  RDY      "%y",      rd
  RDCCR    "%ccr",    rd
  RDASI    "%asi",    rd
  RDTICK   "%tick",   rd
  RDPC     "%pc",     rd
  RDFPRS   "%fprs",   rd
  STBAR
  MEMBAR   membar_mask

  WRY      rsl, reg_or_imm, "%y"
  WRCCR    rsl, reg_or_imm, "%ccr"
  WRASI    rsl, reg_or_imm, "%asi"
  WRFPRS   rsl, reg_or_imm, "%fprs"
  SIR      simm13!
```

Uses MEMBAR 16, membar_mask 4a, rd 3c, RDASI 16, RDCCR 16, RDFPRS 16, RDPC 16, RDTICK 16, RDY 16, reg_or_imm 8b, rsl 4a, simm13 4a, SIR 16, STBAR 16, WRASI 16, WRCCR 16, WRFPRS 16, and WRY 16.

The instructions that read and write the ancillary state registers (`asr`) have their own special syntax.

25b \langle full pattern and constructor specifications 23c $\rangle + \equiv$ (13c) \langle 24c 26a \rangle

```
constructors
  RDASR    "%asr"rqli, rd { rqli > 15 }
  WRASR    rsl, reg_or_imm, "%asr"rdi { rdi > 15 }
```

Uses rd 3c, RDASR 16, rdi 5d, reg_or_imm 8b, rsl 4a, rqli 5d, and WRASR 16.

The registers `rqli` and `rdi` are versions of `rsl` and `rd` that print as integers (so no `fieldinfo` specifications) and are defined in section 2.2.

The equations are not needed for simulation.

25c \langle sim pattern and constructor specifications 23d $\rangle + \equiv$ (13b) \langle 24d 26b \rangle

```
constructors
  RDASR    "%asr"rqli, rd
  WRASR    rsl, reg_or_imm, "%asr"rdi
```

Uses rd 3c, RDASR 16, rdi 5d, reg_or_imm 8b, rsl 4a, rqli 5d, and WRASR 16.

It may be easier to change {RD, WR}xxx to {RD, WR}ASR and specialise the names, and also specialise for {ST, MEM}BAR and SIR; the names specialisation should also include implementation-dependent ASR's. The only problem with this idea is that implementations are free to specify special syntax for each ASR in impl deps 47 and 48. In fact, UltraSPARC seems to specify a slightly alternate syntax for reading ASR's [4, p152], in which no `reg_or_imm` is allowed (implying that `reg_or_imm` is in fact `regROI(0)`). Contradicting itself, it also specifies that the `gsr` (graphics status register) may be written to including `reg_or_imm` [4, p192]. I suspect that the UltraSPARC chips will accept standard WR instructions for all the writable ASR's (and operate as per the standard WR instructions), but if it does it is not documented.

The $\{RD, WR\}PR$ instructions are for reading and writing privileged registers. Notice the use of when to check that $rs1p$ is legal (this is not in the pattern equations due to restrictions in the toolkit). The equations are not needed in simulation.

26a $\langle \text{full pattern and constructor specifications 23c} \rangle + \equiv$ (13c) $\langle 25b \ 27a \rangle$

constructors

```
RDPR rs1p, rd  when { rs1p < 16 } is RDPR & rs1p & rd
                when { rs1p = 31 } is RDPR & rs1p & rd
WRPR rs1, reg_or_imm, rdp { rdp < 15 }
```

Uses $rd \ 3c$, $rdp \ 5e$, $RDPR \ 17c$, $reg_or_imm \ 8b$, $rs1 \ 4a$, $rs1p \ 5e$, and $WRPR \ 17c$.

26b $\langle \text{sim pattern and constructor specifications 23d} \rangle + \equiv$ (13b) $\langle 25c \ 27c \rangle$

constructors

```
RDPR rs1p, rd
WRPR rs1, reg_or_imm, rdp
```

Uses $rd \ 3c$, $rdp \ 5e$, $RDPR \ 17c$, $reg_or_imm \ 8b$, $rs1 \ 4a$, $rs1p \ 5e$, and $WRPR \ 17c$.

See section 2.2 for the definitions of $rs1p$ and rdp .

5.3 Shift, logic, and arithmetic

The logical and arithmetic instructions share identical assembly language syntax, so we can get away with only a single constructor declaration. Unlike SPARC-V8, we cannot include the shift instructions as well since the SPARC-V9 shift instructions have an extra bit x that coincides with the most-significant $simm13$ bit.

26c $\langle \text{pattern and constructor specifications 13d} \rangle + \equiv$ (13a) $\langle 25a \ 30d \rangle$

patterns

```
logical is AND | ANDcc | ANDN | ANDNcc | OR | ORcc | ORN | ORNcc |
          XOR | XORcc | XNOR | XNORcc
arith   is ADD | ADDcc | ADDC | ADDCcc | TADDcc | TADDccTV |
          SUB | SUBcc | SUBC | SUBCcc | TSUBcc | TSUBccTV |
          MULScc | UMUL | SMUL | UMULcc | SMULcc |
          UDIV | SDIV | UDIVcc | SDIVcc |
          MULX | SDIVX | UDIVX |
          SAVE | RESTORE
alu     is logical | arith
```

```
shift32 is SLL | SRL | SRA
shift64 is SLLX | SRLX | SRAX
```

constructors

```
alu      rs1, reg_or_imm,      rd
shift32  rs1, reg_or_shcnt32, rd
shift64  rs1, reg_or_shcnt64, rd
POPC     reg_or_imm,          rd
```

Defines:

```
alu, never used.
arith, never used.
logical, never used.
shift32, never used.
shift64, never used.
```

Uses $ADD \ 15$, $ADDC \ 15$, $ADDcc \ 15$, $ADDCcc \ 15$, $AND \ 15$, $ANDcc \ 15$, $ANDN \ 15$, $ANDNcc \ 15$, $MULScc \ 15$, $MULX \ 15$, $OR \ 15$, $ORcc \ 15$, $ORN \ 15$, $ORNcc \ 15$, $POPC \ 17d$, $rd \ 3c$, $reg_or_imm \ 8b$, $reg_or_shcnt32 \ 8c$, $reg_or_shcnt64 \ 8c$, $RESTORE \ 15$, $rs1 \ 4a$, $SAVE \ 15$, $SDIV \ 15$, $SDIVcc \ 15$, $SDIVX \ 15$, $SLL \ 17b$, $SLLX \ 17b$, $SMUL \ 15$, $SMULcc \ 15$, $SRA \ 17b$, $SRAX \ 17b$, $SRL \ 17b$, $SRLX \ 17b$, $SUB \ 15$, $SUBC \ 15$, $SUBcc \ 15$, $SUBCcc \ 15$, $TADDcc \ 15$, $TADDccTV \ 15$, $TSUBcc \ 15$, $TSUBccTV \ 15$, $UDIV \ 15$, $UDIVcc \ 15$, $UDIVX \ 15$, $UMUL \ 15$, $UMULcc \ 15$, $XNOR \ 15$, $XNORcc \ 15$, $XOR \ 15$, and $XORcc \ 15$.

5.4 Branches and call

A placeholder pattern must be specified before the declaration of any constructor that refers to relocatable addresses; such a constructor may emit a placeholder in lieu of the relocated instruction. We choose the `ILLTRAP` instruction because it causes an illegal-instruction trap if executed. We also define `reloc` to be a relocatable variable so it can be used with the various branching instructions (it is the absolute address of the destination). For simulation, we only use displacements since the toolkit is not foolproof with 64-bit addresses.

```
27a <full pattern and constructor specifications 23c>+≡ (13c) <26a 27b>
    #fields of addrtoken (64) reloc 0:63
    placeholder for instruction is ILLTRAP & imm22 = 0xbad
    relocatable reloc
```

Defines:

`reloc`, used in chunks 27–29, 47, and 48a.
Uses `ILLTRAP` 13e and `imm22` 3c.

The commented-out line is a (failed) attempt to get the toolkit to deal with addresses as 64 bits.

All of the condition-code branch instructions depend on the `{i, f}cond_f2` fields, which have special names used to identify them (see section 2.2). We use those names as a suffix to identify each of the branch variations, except in simulation where we use the top-level opcode and pass the fields as parameters.

All these branches also come in two variants depending on the setting of the annul (`a`) bit. In the assembly language, the `a` bit is notated with the suffix `,a` when set, and with no suffix when clear (defined in section 2.2). We then use `a` as a suffix to the branch instructions.

The deprecated SPARC-V8 branch instructions (simpler than the new predicted branch instructions) `Bicc` and `FBfcc` have a fixed condition-code to check.

```
27b <full pattern and constructor specifications 23c>+≡ (13c) <27a 27d>
    constructors
    B^icond_f2^a reloc { reloc = label + 4 * disp22! } is
        label: Bicc & icond_f2 & a & disp22
```

```
    FB^fcond_f2^a reloc { reloc = label + 4 * disp22! } is
        label: FBfcc & fcond_f2 & a & disp22
```

Uses `a` 3c, `Bicc` 13e, `disp22` 3c, `FBfcc` 13e, `fcond_f2` 3c, `icond_f2` 3c, and `reloc` 27a.

```
27c <sim pattern and constructor specifications 23d>+≡ (13b) <26b 27e>
    constructors
    Bicc icond_f2 a disp22!
    FBfcc fcond_f2 a disp22!
```

Uses `a` 3c, `Bicc` 13e, `disp22` 3c, `FBfcc` 13e, `fcond_f2` 3c, and `icond_f2` 3c.

The prediction-based branches depend on the prediction (`p`) bit (as well as the annul bit), which in assembly are indicated by either `,pn` or `,pt`. These branches also depend on particular condition codes `{i, f}cc_f2` (see tables 39 and 40 respectively).

```
27d <full pattern and constructor specifications 23c>+≡ (13c) <27b 28a>
    constructors
    BP^icond_f2^a^p icc_f2, reloc {
        icc_f2@[0] = 0,
        reloc = label + 4 * disp19!
    } is label: BPcc & icond_f2 & a & p & icc_f2 & disp19
```

```
    FBP^fcond_f2^a^p fcc_f2, reloc { reloc = label + 4 * disp19! } is
        label: FBPFcc & fcond_f2 & a & p & fcc_f2 & disp19
```

Uses `a` 3c, `BPcc` 13e, `disp19` 3c, `FBPFcc` 13e, `fcc_f2` 3c, `fcond_f2` 3c, `icc_f2` 3c, `icond_f2` 3c, `p` 3c, and `reloc` 27a.

```
27e <sim pattern and constructor specifications 23d>+≡ (13b) <27c 28b>
    constructors
    BPcc icond_f2 a p icc_f2 disp19!
    FBPFcc fcond_f2 a p fcc_f2 disp19!
```

Uses `a` 3c, `BPcc` 13e, `disp19` 3c, `FBPFcc` 13e, `fcc_f2` 3c, `fcond_f2` 3c, `icc_f2` 3c, `icond_f2` 3c, and `p` 3c.

The Tcc instructions depend on the `icc_f4` field to determine which condition code to check (see Table 40), and on the `icond_f2` field to determine how to check it:

28a *<full pattern and constructor specifications 23c>+≡* (13c) <27d 28c>
 constructors
`T^icond_f2 icc_f4, software_trap_number { icc_f4@[0] = 0 }`
`is Tcc & icond_f2 & icc_f4 & software_trap_number`
 Uses `icc_f4` 4b, `icond_f2` 3c, `software_trap_number` 9b, and Tcc 15.

28b *<sim pattern and constructor specifications 23d>+≡* (13b) <27e 28d>
 constructors
`Tcc icond_f2 icc_f4 software_trap_number`
 Uses `icc_f4` 4b, `icond_f2` 3c, `software_trap_number` 9b, and Tcc 15.

The CALL instruction is like the branches, except it uses a 30-bit displacement, and there is no annul bit.

28c *<full pattern and constructor specifications 23c>+≡* (13c) <28a 28e>
 constructors
`CALL reloc { reloc = label + 4 * disp30! } is label: CALL & disp30`
 Uses CALL 13d, `disp30` 3b, and `reloc` 27a.

28d *<sim pattern and constructor specifications 23d>+≡* (13b) <28b 28f>
 constructors
`CALL disp30!`
 Uses CALL 13d and `disp30` 3b.

5.5 Conditional moves and integer register conditions

The condition-code instructions MOVCC and FMOVCC are very similar to the branch instructions, but the organisation of the `{i, f}cond_f4` fields are not expanded out nicely like it is for the branches (i.e., there is no separate table like Table 36). Firstly, we split them into two parts, depending on whether we are checking an integer or floating-point condition code (hence the `{i, f}`).

By inspection of the definitions of `{, F}MOVCC` and Table 36, we can see there is a direct correspondence. The integer condition code ordering is equivalent to that in Table 36 columns 1, 2 and 5 (BPCC, BiCC and TCC). The floating-point condition code ordering is equivalent to Table 36 columns 3 and 4 (FBPFCC and FBfCC). Hence `{i, f}cond_f4` are directly equivalent to `{i, f}cond_f2` (and hence are named just like them in section 2.2).

We specialise constructors for MOVCC depending on whether we are checking an integer or floating-point condition code. For integer condition-codes, we also ensure that the condition code is legal. The `cc2_f4` is implied by (and distinguishes between) the conditions (integer or floating-point condition code).

28e *<full pattern and constructor specifications 23c>+≡* (13c) <28c 29a>
 constructors
`MOV^icond_f4 icc_f4, reg_or_imm11, rd { icc_f4@[0] = 0 } is`
`MOVcc & cc2_f4 = 1 & icond_f4 & icc_f4 & reg_or_imm11 & rd`
`MOVF^fcond_f4 fcc_f4, reg_or_imm11, rd is`
`MOVcc & cc2_f4 = 0 & fcond_f4 & fcc_f4 & reg_or_imm11 & rd`
 Uses `cc2_f4` 4b, `fcc_f4` 4b, `fcond_f4` 4b, `icc_f4` 4b, `icond_f4` 4b, MOVCC 15, `rd` 3c, and `reg_or_imm11` 8c.

For simulation, we pass all fields as parameters, including the switch for floating-point or integer condition-codes (note that `icond_f4` is not necessarily an integer condition-code!).

28f *<sim pattern and constructor specifications 23d>+≡* (13b) <28d 29b>
 constructors
`MOVcc cc2_f4 icond_f4 icc_f4 reg_or_imm11 rd`
 Uses `cc2_f4` 4b, `icc_f4` 4b, `icond_f4` 4b, MOVCC 15, `rd` 3c, and `reg_or_imm11` 8c.

As for MOVcc, we specialise constructors for FMOV{S,D,Q}cc in a similar fashion. This time, however, the opf_cc is one field instead of two.

29a *{full pattern and constructor specifications 23c}+≡* (13c) <28e 29c>
constructors

```
FMOVS^icond_f4 opf_cc, rs2fs, rdfs {
    opf_cc@[2] = 1, opf_cc@[0] = 0
} is FMOVSc & icond_f4 & opf_cc & rs2fs & rdfs
FMOVSF^fcond_f4 opf_cc, rs2fs, rdfs { opf_cc@[2] = 0 }
    is FMOVSc & fcond_f4 & opf_cc & rs2fs & rdfs
```

```
FMOVD^icond_f4 opf_cc, rs2fd, rdfd {
    opf_cc@[2] = 1, opf_cc@[0] = 0
} is FMOVDc & icond_f4 & opf_cc & rs2fd & rdfd
FMOVDF^fcond_f4 opf_cc, rs2fd, rdfd { opf_cc@[2] = 0 }
    is FMOVDc & fcond_f4 & opf_cc & rs2fd & rdfd
```

```
FMOVQ^icond_f4 opf_cc, rs2fq, rdfq {
    opf_cc@[2] = 1, opf_cc@[0] = 0
} is FMOVQc & icond_f4 & opf_cc & rs2fq & rdfq
FMOVQF^fcond_f4 opf_cc, rs2fq, rdfq { opf_cc@[2] = 0 }
    is FMOVQc & fcond_f4 & opf_cc & rs2fq & rdfq
```

Uses fcond_f4 4b, FMOVDc 22a, FMOVQc 22a, FMOVSc 22a, icond_f4 4b, opf_cc 4b, rdfd 12b 12c, rdfq 12b 12c, rdfs 11a 12c, rs2fd 12b 12c, rs2fq 12b 12c, and rs2fs 11a 12c.

29b *{sim pattern and constructor specifications 23d}+≡* (13b) <28f 29d>
constructors

```
FMOVSc icond_f4 opf_cc rs2fs rdfs
FMOVDc icond_f4 opf_cc rs2fd rdfd
FMOVQc icond_f4 opf_cc rs2fq rdfq
```

Uses FMOVDc 22a, FMOVQc 22a, FMOVSc 22a, icond_f4 4b, opf_cc 4b, rdfd 12b 12c, rdfq 12b 12c, rdfs 11a 12c, rs2fd 12b 12c, rs2fq 12b 12c, and rs2fs 11a 12c.

Integer register condition branches: the difficulty with BPr is that the displacement is assigned in two parts: dl6{hi,lo}. We want to ensure that the displacement is not too large, but it is difficult to ensure that the toolkit will do the right thing. After many iterations, we have arrived at this: note the use of a typed constructor (disp16) to simplify things (it is only used internally so does not belong with the other types in section 3).

29c *{full pattern and constructor specifications 23c}+≡* (13c) <29a 30a>
constructors

```
disp16 disp! : disp16t {
    disp@[0:1] = 0,
    dl6lo = disp@[2:15],
    dl6hi! = disp@[16:31]!
} is dl6lo & dl6hi
BR^rcond_f2^a^p rs1, reloc #{ rcond_f2 != 0, rcond_f2 != 4 }
    is label: BPr & rcond_f2 & a & p & rs1 & disp16(reloc - label)
```

Uses a 3c, BPr 14, dl6hi 3c, dl6lo 3c, p 3c, rcond_f2 3c, reloc 27a, and rs1 4a.

The disp@[0:1] = 0 line can be commented out for speed in encoding if it doesn't need to be checked (it is equivalent to saying disp = 4 * _). The rcond_f2 equations are commented out since they confuse the toolkit; the user must ensure that only the correct conditions are used.

29d *{sim pattern and constructor specifications 23d}+≡* (13b) <29b 30b>
constructors

```
BPr rcond_f2 a p rs1 disp16!
    { dl6lo = disp16@[0:13], dl6hi! = disp16@[14:31] }
    is BPr & rcond_f2 & a & p & rs1 & dl6lo & dl6hi
```

Uses a 3c, BPr 14, dl6hi 3c, dl6lo 3c, p 3c, rcond_f2 3c, and rs1 4a.

The integer register condition moves are fairly trivial. The equations are commented out since they confuse the toolkit.

30a *(full pattern and constructor specifications 23c)* +≡ (13c) <29c 30c>
constructors

```
MOVR^rcond_f3  rsl, reg_or_imm10, rd  #{ rcond_f3 != 0, rcond_f3 != 4 }
FMOVSR^rcond_f4 rsl, rs2fs, rdfs  #{ rcond_f4 != 0, rcond_f4 != 4 }
FMOVDR^rcond_f4 rsl, rs2fd, rdfd  #{ rcond_f4 != 0, rcond_f4 != 4 }
FMOVQR^rcond_f4 rsl, rs2fq, rdfq  #{ rcond_f4 != 0, rcond_f4 != 4 }
```

Uses FMOVDR 22a, FMOVQR 22a, FMOVSR 22a, MOVR 17e, rcond.f3 4a, rcond.f4 4b, rd 3c, rdfd 12b 12c, rdfq 12b 12c, rdfs 11a 12c, reg_or_imm10 8c, rs1 4a, rs2fd 12b 12c, rs2fq 12b 12c, and rs2fs 11a 12c.

30b *(sim pattern and constructor specifications 23d)* +≡ (13b) <29d
constructors

```
MOVR rcond_f3  rsl reg_or_imm10 rd
FMOVSR rcond_f4 rsl rs2fs rdfs
FMOVDR rcond_f4 rsl rs2fd rdfd
FMOVQR rcond_f4 rsl rs2fq rdfq
```

Uses FMOVDR 22a, FMOVQR 22a, FMOVSR 22a, MOVR 17e, rcond.f3 4a, rcond.f4 4b, rd 3c, rdfd 12b 12c, rdfq 12b 12c, rdfs 11a 12c, reg_or_imm10 8c, rs1 4a, rs2fd 12b 12c, rs2fq 12b 12c, and rs2fs 11a 12c.

Due to the toolkit problems with not checking the validity of rcond, we discard the illegal constructors.

30c *(full pattern and constructor specifications 23c)* +≡ (13c) <30a
discard

```
"BR0!,pt" "BR0!,pn" "BR0!,a,pt" "BR0!,a,pn"
"BR4!,pt" "BR4!,pn" "BR4!,a,pt" "BR4!,a,pn"
"MOVR0!" "MOVR4!"
"FMOVSR0!" "FMOVSR4!" "FMOVDR0!" "FMOVDR4!" "FMOVQR0!" "FMOVQR4!"
```

Uses a 3c.

5.6 Floating point

Floating-point arithmetic instructions include two-operand and three-operand variants. Their output patterns are the implicit conjunctions of their opcodes and operands so the output pattern is omitted. We use the floating-point register constructor to enforce the encoding.

Firstly we enumerate the two-operand variants, based on their operand types.

30d *(pattern and constructor specifications 13d)* +≡ (13a) <26c 31a>
constructors

```
f2sTOs rs2fs, rdfs
f2sTOd rs2fs, rdfd
f2sTOq rs2fs, rdfq
f2dTOs rs2fd, rdfs
f2dTOd rs2fd, rdfd
f2dTOq rs2fd, rdfq
f2qTOs rs2fq, rdfs
f2qTOd rs2fq, rdfd
f2qTOq rs2fq, rdfq
```

Uses f2dTOd 20, f2dTOq 20, f2dTOs 20, f2qTOd 20, f2qTOq 20, f2qTOs 20, f2sTOd 20, f2sTOq 20, f2sTOs 20, rdfd 12b 12c, rdfq 12b 12c, rdfs 11a 12c, rs2fd 12b 12c, rs2fq 12b 12c, and rs2fs 11a 12c.

The three-operand variants, based on their operand types are listed here.

31a \langle pattern and constructor specifications 13d $\rangle + \equiv$ (13a) \langle 30d 31b \rangle

```
constructors
  float3s  rs1fs, rs2fs, rdfs
  float3d  rs1fd, rs2fd, rdfd
  float3q  rs1fq, rs2fq, rdfq
  FsmULD  rs1fs, rs2fs, rdfd
  FdMULq  rs1fd, rs2fd, rdfq

  fcompares fcc_f3, rs1fs, rs2fs
  fcompared fcc_f3, rs1fd, rs2fd
  fcompareq fcc_f3, rs1fq, rs2fq
```

Uses `fcc_f3` 4a, `fcompared` 22a, `fcompareq` 22a, `fcompares` 22a, `FdMULq` 20, `float3d` 20, `float3q` 20, `float3s` 20, `FsmULD` 20, `rdfd` 12b 12c, `rdfq` 12b 12c, `rdfs` 11a 12c, `rs1fd` 12b 12c, `rs2fd` 12b 12c, `rs1fq` 12b 12c, `rs2fq` 12b 12c, `rs1fs` 11a 12c, and `rs2fs` 11a 12c.

5.7 Miscellany

The remaining instructions don't belong to any particular grouping. Firstly there are the instructions which don't take any operands:

31b \langle pattern and constructor specifications 13d $\rangle + \equiv$ (13a) \langle 31a 31c \rangle

```
patterns
  nooperands is NOP | DONE | RETRY | SAVED | RESTORED | FLUSHW

constructors
  nooperands
```

Defines:

`nooperands`, never used.

Uses `DONE` 17a, `FLUSHW` 15, `NOP` 13e, `RESTORED` 17a, `RETRY` 17a, and `SAVED` 17a.

Then some of the remaining instructions:

31c \langle pattern and constructor specifications 13d $\rangle + \equiv$ (13a) \langle 31b 31d \rangle

```
constructors
  FLUSH  address_
  JMPL  address_, rd
  RETURN address_
  ILLTRAP imm22
```

Uses `address_` 9a, `FLUSH` 15, `ILLTRAP` 13e, `imm22` 3c, `JMPL` 15, `rd` 3c, and `RETURN` 15.

The SPARC-V9 architecture manual defines `sethi` such that it destroys the least significant ten bits on encoding. Therefore, no single bi-directional definition of `sethi` can be written without loss of information. The SPARC-V8 spec used two constructors; our solution is to provide the simpler constructor, with an explicit immediate value.

31d \langle pattern and constructor specifications 13d $\rangle + \equiv$ (13a) \langle 31c

```
constructors
  SETHI imm22, rd
```

Uses `imm22` 3c, `rd` 3c, and `SETHI` 13e.

6 UltraSPARC implementation-dependent instructions

The SPARC-V9 manual specifies that the `IMPDEP{1,2}` instructions are completely implementation dependent, and that their dependency on the `impldep{1,2}` fields is implementation dependent. If the implementation does not define the instruction, then an `illegal_instruction` trap occurs. Hence we do not define constructors for `IMPDEP{1,2}`; implementation-dependent instructions should be defined explicitly here.

All of the UltraSPARC “extended instructions” are defined here. These include `SHUTDOWN`, the additional ancillary state registers (ASR’s) and the graphics instructions. Most of the extra instructions are defined in Chapter 13 of the UltraSPARC User’s Manual [4].

```
32a <ultrasparc.spec 32a>≡
    # requires fields.spec and core.spec
    # (depends on table 32 and fp register encoding)
    <ultrasparc patterns and constructors 32b>
```

Worthy of note is the fact that all the UltraSPARC extended instructions are based on `IMPDEP1`.

6.1 Ancillary state registers

The ancillary state registers used by the UltraSPARC chips are:

ASR Value	Name	Assembler	Access
0x10	PERF_CONTROL_REG	%pcr	RW
0x11	PERF_COUNTER	%pic	RW
0x12	DISPATCH_CONTROL_REG	%dcr	RW
0x13	GRAPHIC_STATUS_REG	%gsr	RW
0x14	SET_SOFTINT	%set_softint	W
0x15	CLEAR_SOFTINT	%clear_softint	W
0x16	SOFTINT_REG	%softint	RW
0x17	TICK_CMPR_REG	%tick_cmpr	RW

Other registers used to control and monitor the UltraSPARC are manipulated using implementation-dependent address space identifiers (ASI’s). This is not interesting for specification of the instruction syntax, but does need to be addressed for simulation.

```
32b <ultrasparc patterns and constructors 32b>≡ (32a) 33a>
    patterns
    [ RDPCR RDPIC RDDCR RDGSR _ _ RDSOFTINT RDTICK_CMPR ]
    is RDASR & rsl = {0x10 to 0x17}
    constructors
    RDPCR "%pcr", rd
    RDPIC "%pic", rd
    RDDCR "%dcr", rd
    RDGSR "%gsr", rd
    RDSOFTINT "%softint", rd
    RDTICK_CMPR "%tick_cmpr", rd
```

Defines:

```
    RDDCR, used in chunk 40.
    RDGSR, used in chunk 40.
    RDPCR, never used.
    RDPIC, used in chunk 40.
    RDSOFTINT, used in chunk 40.
    RDTICK_CMPR, used in chunk 40.
    Uses rd 3c, RDASR 16, and rsl 4a.
```

I have given each ASR an opcode name, which is not necessary, but tidies up the specification. For simulation, it may be easier to just use the `RDASR` constructor.

For some strange reason, the suggested assembly language syntax for the writing to the ASR's [4, p.152/158] (second page number is new edition) specifies no second source register, except in the case of the GSR which is separately allowed to have a `reg_or_imm` second source register [4, p.192/p.198]. Currently, we ignore the specification and use the standard `reg_or_imm` syntax.

```
33a <ultrasparc patterns and constructors 32b>+≡ (32a) <32b 33b>
patterns
  [ WRPCR WRPIC WRDCR WRGSR
    WRSET_SOFTINT WRCLEAR_SOFTINT WRSOFTINT WRTICK_CMPR ]
  is WRASR & rd = {0x10 to 0x17}
constructors
  WRPCR rsl, reg_or_imm, "%pcr"
  WRPIC rsl, reg_or_imm, "%pic"
  WRDCR rsl, reg_or_imm, "%dcr"
  WRGSR rsl, reg_or_imm, "%gsr"
  WRSET_SOFTINT rsl, reg_or_imm, "%set_softint"
  WRCLEAR_SOFTINT rsl, reg_or_imm, "%clear_softint"
  WRSOFTINT rsl, reg_or_imm, "%softint"
  WRTICK_CMPR rsl, reg_or_imm, "%tick_cmpr"
```

Defines:

```
WRCLEAR_SOFTINT, used in chunk 40.
WRDCR, used in chunk 40.
WRGSR, used in chunk 40.
WRPCR, used in chunk 40.
WRPIC, used in chunk 40.
WRSET_SOFTINT, never used.
WRSOFTINT, used in chunk 40.
WRTICK_CMPR, used in chunk 40.
```

Uses `rd` 3c, `reg_or_imm` 8b, `rsl` 4a, and `WRASR` 16.

6.2 Shutdown

The SHUTDOWN instruction is decidedly simple.

```
33b <ultrasparc patterns and constructors 32b>+≡ (32a) <33a 34a>
patterns
  SHUTDOWN is IMPDEP1 & opf = 0x80
constructors
  SHUTDOWN
```

Defines:

```
SHUTDOWN, used in chunk 40.
```

Uses `IMPDEP1` 15 and `opf` 4a.

6.3 Graphics instructions

All the graphics instructions (also known as VIS instructions) have the same structure as floating-point instructions, with three register fields and an instruction switch based on the `opf` field. For the VIS instructions, `opf` must be less than `0x80`.

To enable easy reading and decoding of the VIS instructions, we split the `opf` field into pieces — `vismbz`, `visop1` and `visop2` — of two, two and five bits respectively.

```
33c <ultrasparc-fields.spec 33c>≡
# ultrasparc-fields.spec
# goes after fields.spec (in fields declaration)
vismbz 12:13 visop1 10:11 visop2 5:9
```

Defines:

```
vismbz, used in chunk 34a.
```

```
visop1, used in chunk 34a.
```

```
visop2, used in chunks 34–37.
```

34a \langle *ultrasparc patterns and constructors 32b* $\rangle + \equiv$ (32a) \langle 33b 34b \rangle
 patterns
 [VISOP0 VISOP1 VISOP2 VISOP3] is
 IMPDEP1 & vismbz = 0 & visop1 = {0 to 3}

Defines:

VISOP0, used in chunk 34b.
 VISOP1, used in chunk 35.
 VISOP2, used in chunk 36.
 VISOP3, used in chunk 37.

Uses IMPDEP1 15, vismbz 33c, and visop1 33c.

6.3.1 Edge handling, array and align address

34b \langle *ultrasparc patterns and constructors 32b* $\rangle + \equiv$ (32a) \langle 34a 35 \rangle
 patterns
 visop0rrr is any of
 [EDGE8 EDGE32 ARRAY8 ALIGNADDRESS
 — — — —
 EDGE8L EDGE32L ARRAY16 ALIGNADDRESS_LITTLE
 — — — —
 EDGE16 — ARRAY32 —
 — — — —
 EDGE16L — — —
 — — — —] ,
 which is VISOP0 & visop2 = {0 to 31 columns 4}

Defines:

ALIGNADDRESS, used in chunk 40.
 ALIGNADDRESS_LITTLE, never used.
 ARRAY16, used in chunk 40.
 ARRAY32, used in chunk 40.
 ARRAY8, used in chunk 40.
 EDGE16, used in chunk 40.
 EDGE32, used in chunk 40.
 EDGE8, used in chunk 40.
 EDGE16L, used in chunk 40.
 EDGE32L, used in chunk 40.
 EDGE8L, used in chunk 40.
 visop0rrr, used in chunk 38.

Uses VISOP0 34a and visop2 33c.

We group the instructions by their parameter types. The r's in visop0rrr refer to a plain integer register; three r's indicates that all three registers are used.

6.3.2 Pixel compare, partitioned multiply, pack and pixel distance

35 *(ultrasparc patterns and constructors 32b)* +≡ (32a) <34b 36>
 patterns

```
[ FCMPL16 FCMPT16 _ FMULD8SUx16
  _ _ FMUL8x16 FMULD8ULx16
  FCMPL16 FCMPEQ16 _ FPACK32
  _ _ FMUL8x16AU FPACK16
  FCMPL32 FCMPT32 _ _
  _ _ FMUL8x16AL FPACKFIX
  FCMPL32 FCMPEQ32 FMULD8SUx16 PDIST
  _ _ FMUL8ULx16 _ ]
is VISOP1 & visop2 = {0 to 31 columns 4}
```

```
visopl1ddr is FCMPL16 | FCMPT16 | FCMPL16 | FCMPEQ16 |
              FCMPL32 | FCMPT32 | FCMPL32 | FCMPEQ32
visopl1ssd is FMUL8x16 | FMUL8x16
visopl1ssd is FMUL8x16AU | FMUL8x16AL | FMULD8SUx16 | FMULD8ULx16
visopl1ddd is FMULD8SUx16 | FMULD8ULx16 | FPACK32 | PDIST
visopl1ds is FPACK16 | FPACKFIX
```

Defines:

FCMPEQ16, used in chunk 40.
 FCMPEQ32, never used.
 FCMPT16, used in chunk 40.
 FCMPT32, used in chunk 40.
 FCMPL16, used in chunk 40.
 FCMPL32, used in chunk 40.
 FCMPL16, used in chunk 40.
 FCMPL32, used in chunk 40.
 FCMPL16, used in chunk 40.
 FCMPL32, used in chunk 40.
 FMULD8SUx16, used in chunk 40.
 FMULD8ULx16, used in chunk 40.
 FMULD8SUx16, never used.
 FMULD8ULx16, used in chunk 40.
 FMUL8x16, used in chunk 40.
 FMUL8x16AL, used in chunk 40.
 FMUL8x16AU, never used.
 FPACK16, used in chunk 40.
 FPACK32, used in chunk 40.
 FPACKFIX, used in chunk 40.
 PDIST, used in chunk 40.
 visopl1ddd, used in chunk 38.
 visopl1ddr, used in chunk 38.
 visopl1ds, used in chunk 38.
 visopl1ssd, used in chunk 38.
 visopl1ssd, used in chunk 38.

Uses VISOP1 34a and visop2 33c.

As above, we group by parameter types; *s* refers to a single-precision floating-point parameter and *d* refers to a double-precision floating-point parameter (the order is {source 1, source 2, destination} and is right-associative). The duplicate of FMUL8x16 in the pattern definition of visopl1ssd is a workaround to force expansion of constructors; ignore any warnings when the constructor is generated.

6.3.3 Align data, merge, expand and partitioned add

36 *(ultrasparc patterns and constructors 32b)* +≡ (32a) <35 37>
 patterns

```
[ _ FALIGNDATA FPADD16 _
  _ _ FPADD16S _
  _ _ FPADD32 _
  _ FPMERGE FPADD32S _
  _ _ FPSUB16 _
  _ FEXPAND FPSUB16S _
  _ _ FPSUB32 _
  _ _ FPSUB32S _ ]
is VISOP2 & visop2 = {0 to 31 columns 4}
```

```
visop2ddd is FALIGNDATA | FPADD16 | FPADD32 | FPSUB16 | FPSUB32
visop2ssd is FPMERGE | FPMERGE
visop2sd is FEXPAND | FEXPAND
visop2sss is FPADD16S | FPADD32S | FPSUB16S | FPSUB32S
```

Defines:

FALIGNDATA, used in chunk 40.
 FEXPAND, used in chunk 40.
 FPADD16, used in chunk 40.
 FPADD32, used in chunk 40.
 FPADD16S, used in chunk 40.
 FPADD32S, used in chunk 40.
 FPMERGE, used in chunk 40.
 FPSUB16, used in chunk 40.
 FPSUB32, never used.
 FPSUB16S, used in chunk 40.
 FPSUB32S, used in chunk 40.
 visop2ddd, used in chunk 38.
 visop2sd, used in chunk 38.
 visop2ssd, used in chunk 38.
 visop2sss, used in chunk 38.
 Uses visop2 33c and VISOP2 34a.

The duplicates of FPMERGE and FEXPAND in the patterns are workarounds to force expansion of constructors; ignore any warnings when constructors are generated.

6.3.4 Logical

37 *(ultrasparc patterns and constructors 32b)* +≡ (32a) <36 38>

```

patterns
[ FZERO      FANDNOT1  FAND      FSRC2
  FZEROS     FANDNOT1S FANDS     FSRC2S
  FNOR       FNOT1     FXNOR     FORNOT1
  FNORS      FNOT1S    FXNORS    FORNOT1S
  FANDNOT2   FXOR      FSRC1     FOR
  FANDNOT2S  FXORS     FSRC1S    FORS
  FNOT2      FNAND     FORNOT2   FONE
  FNOT2S     FNANDS    FORNOT2S  FONES   ]
is VISOP3 & visop2 = {0 to 31 columns 4}

```

```

visop3d is  FZERO | FONE
visop3s is  FZEROS | FONES
visop3dd is FNOT2 | FSRC2
visop3dxd is FSRC1 | FNOT1
visop3ss is FNOT2S | FSRC2S
visop3sxs is FNOT1S | FSRC1S
visop3ddd is FNOR | FANDNOT2 | FANDNOT1 | FXOR | FNAND |
              FAND | FXNOR | FORNOT2 | FORNOT1 | FOR
visop3sss is FNORS | FANDNOT2S | FANDNOT1S | FXORS | FNANDS |
              FANDS | FXNORS | FORNOT2S | FORNOT1S | FORS

```

Defines:

FAND, used in chunk 40.
 FANDNOT1, used in chunk 40.
 FANDNOT2, used in chunk 40.
 FANDNOT1S, used in chunk 40.
 FANDNOT2S, never used.
 FANDS, used in chunk 40.
 FNAND, used in chunk 40.
 FNANDS, used in chunk 40.
 FNOR, used in chunk 40.
 FNORS, used in chunk 40.
 FNOT1, used in chunk 40.
 FNOT2, never used.
 FNOT1S, used in chunk 40.
 FNOT2S, used in chunk 40.
 FONE, used in chunk 40.
 FONES, never used.
 FOR, never used.
 FORNOT1, used in chunk 40.
 FORNOT2, used in chunk 40.
 FORNOT1S, used in chunk 40.
 FORNOT2S, used in chunk 40.
 FORS, used in chunk 40.
 FSRC1, used in chunk 40.
 FSRC2, used in chunk 40.
 FSRC1S, used in chunk 40.
 FSRC2S, used in chunk 40.
 FXNOR, used in chunk 40.
 FXNORS, used in chunk 40.
 FXOR, used in chunk 40.
 FXORS, used in chunk 40.
 FZERO, used in chunk 40.
 FZEROS, used in chunk 40.
 visop3d, used in chunk 38.
 visop3dd, used in chunk 38.
 visop3ddd, used in chunk 38.
 visop3dxd, used in chunk 38.
 visop3s, used in chunk 38.
 visop3ss, used in chunk 38.
 visop3sss, used in chunk 38.

`visop3sxs`, used in chunk 38.
 Uses `visop233c` and `VISOP334a`.

The `x` “type” suffix indicates an unused second source. This is because (for some strange reason) `VIS` defines operations which copy (with optional negation) either `rs1f` or `rs2f`. Normally we’d expect just the one choice using `rs2f`. I don’t know why the `rs1f` option is provided, but that’s how it goes!

6.3.5 Constructors

Putting all of the `VIS` instruction variants together, we get the following patterns grouped by parameter types.

38 `<ultrasparc patterns and constructors 32b>+≡` (32a) `<37 39a>`

```
patterns
  visops   is visop3s
  visopd   is visop3d
  visopss  is visop3ss
  visopsd  is visop2sd
  visopds  is visopl1ds
  visopdd  is visop3dd
  visopsxs is visop3sxs
  visopdxd is visop3dxd
  visopsss is visop2sss | visop3sss
  visopssd is visopl1ssd | visop2ssd
  visopsdd is visopl1sdd
  visopddd is visopl1ddd | visop2ddd | visop3ddd
  visopddr is visopl1ddr
  visoprdr is visop0rdr
```

Defines:

```
visopd, used in chunk 39a.
visopdd, used in chunk 39a.
visopddd, used in chunk 39a.
visopddr, used in chunk 39a.
visopds, used in chunk 39a.
visopdxd, used in chunk 39a.
visoprdr, used in chunk 39a.
visops, used in chunk 39a.
visopsd, used in chunk 39a.
visopsdd, used in chunk 39a.
visopss, used in chunk 39a.
visopssd, used in chunk 39a.
visopsss, used in chunk 39a.
visopsxs, used in chunk 39a.
```

Uses `visop3d` 37, `visop3dd` 37, `visopl1ddd` 35, `visop2ddd` 36, `visop3ddd` 37, `visopl1ddr` 35, `visopl1ds` 35, `visop3dxd` 37, `visop0rdr` 34b, `visop3s` 37, `visop2sd` 36, `visopl1sdd` 35, `visop3ss` 37, `visopl1ssd` 35, `visop2ssd` 36, `visop2sss` 36, `visop3sss` 37, and `visop3sxs` 37.

These patterns can then be turned into constructors with the correct parameters.

39a \langle ultrasparc patterns and constructors 32b $\rangle + \equiv$ (32a) \triangleleft 38

```
constructors
  visops  rdfs
  visopd  rdfd
  visopss rs2fs, rdfs
  visopsd rs2fs, rdfd
  visopds rs2fd, rdfs
  visopdd rs2fd, rdfd
  visopxs rslfs, rdfs
  visopxd rslfd, rdfd
  visopsss rslfs, rs2fs, rdfs
  visopssd rslfs, rs2fs, rdfd
  visopssdd rslfs, rs2fd, rdfd
  visopddd rslfd, rs2fd, rdfd
  visopddr rslfd, rs2fd, rd
  visopr   rsl,  rs2,  rd
```

Uses rd 3c, rdfd 12b 12c, rdfs 11a 12c, rs1 4a, rs2 4a, rs1fd 12b 12c, rs2fd 12b 12c, rslfs 11a 12c, rs2fs 11a 12c, visopd 38, visopdd 38, visopddd 38, visopddr 38, visopds 38, visopdx 38, visopr 38, visops 38, visopsd 38, visopssdd 38, visopss 38, visopssd 38, visopsss 38, and visopxs 38.

6.4 Assembler syntax

39b \langle ultrasparc-asm.spec 39b $\rangle \equiv$

```
# ultrasparc-asm.spec
# goes after asm.spec
 $\langle$ ultrasparc assembler syntax 39c $\rangle$ 
```

The UltraSPARC ASR opcodes are all disambiguated in the parameters for assembly.

39c \langle ultrasparc assembler syntax 39c $\rangle \equiv$ (39b) 39d \triangleright

```
assembly opcode
  {RD}{PCR,PIC,DCR,GSR,SOFTINT,TICK_CMPR} is $1
  {WR}{PCR,PIC,DCR,GSR,{SET_,CLEAR_,}SOFTINT,TICK_CMPR} is $1
```

Some of the graphics assembler instructions (VIS instruction set) for the UltraSPARC are different to the opcodes:

39d \langle ultrasparc assembler syntax 39c $\rangle + \equiv$ (39b) \triangleleft 39c

```
assembly opcode
  {ALIGNADDR}ESS          is $1
  {ALIGNADDR}ESS_{L}ITLLE is $1$2
```

6.5 Validating

See section 11 for rationale. All UltraSPARC implementation dependent instructions were tested on 20000627.

```
40 <ultrasparc-check.spec 40>≡
  ## VIS instructions: tested 20000627
  #discard
  #RDPCR RDPIC RDDCR RDGSR RDSOFTINT RTICK_CMPR WRPCR WRPIC WRDCR WRGSR
  #WRSET_SOFTINT WRCLEAR_SOFTINT WRSOFTINT WRTICK_CMPR SHUTDOWN FZEROS
  #FONES FZERO FONE FNOT2S FNOT1S FSRC1S FSRC2S FEXPAND FPACK16 FPACKFIX
  #FNOT2 FNOT1 FSRC1 FSRC2 FPADD16S FPADD32S FPSUB16S FPSUB32S FNORS
  #FANDNOT2S FANDNOT1S FXORS FNANDS FANDS FXNORS FORNOT2S FORNOT1S FORS
  #FMUL8x16AU FMUL8x16AL FMULD8SUx16 FMULD8ULx16 FPMERGE FMUL8x16
  #FMUL8SUx16 FMUL8ULx16 FPACK32 PDIST FALIGNDATA FPADD16 FPADD32 FPSUB16
  #FPSUB32 FNOR FANDNOT2 FANDNOT1 FXOR FNAND FAND FXNOR FORNOT2 FORNOT1
  #FOR FCMPE16 FCMPGT16 FCMPE16 FCMPEQ16 FCMPE32 FCMPGT32 FCMPE32
  #FCMPEQ32 EDGE8 EDGE32 ARRAY8 ALIGNADDRESS EDGE8L EDGE32L ARRAY16
  #ALIGNADDRESS_LITTLE EDGE16 ARRAY32 EDGE16L
```

Uses ALIGNADDRESS 34b, ARRAY16 34b, ARRAY32 34b, ARRAY8 34b, EDGE16 34b, EDGE32 34b, EDGE8 34b, EDGE16L 34b, EDGE32L 34b, EDGE8L 34b, FALIGNDATA 36, FAND 37, FANDNOT1 37, FANDNOT2 37, FANDNOT1S 37, FANDS 37, FCMPEQ16 35, FCMPGT16 35, FCMPGT32 35, FCMPE16 35, FCMPE32 35, FCMPE16 35, FCMPE32 35, FEXPAND 36, FMULD8SUx16 35, FMULD8ULx16 35, FMUL8ULx16 35, FMUL8x16 35, FMUL8x16AL 35, FNAND 37, FNANDS 37, FNOR 37, FNORS 37, FNOT1 37, FNOT1S 37, FNOT2S 37, FONE 37, FORNOT1 37, FORNOT2 37, FORNOT1S 37, FORNOT2S 37, FORS 37, FPACK16 35, FPACK32 35, FPACKFIX 35, FPADD16 36, FPADD32 36, FPADD16S 36, FPADD32S 36, FPMERGE 36, FPSUB16 36, FPSUB16S 36, FPSUB32S 36, FSRC1 37, FSRC2 37, FSRC1S 37, FSRC2S 37, FXNOR 37, FXNORS 37, FXOR 37, FXORS 37, FZERO 37, FZEROS 37, PDIST 35, RDDCR 32b, RDGSR 32b, RDPIC 32b, RDSOFTINT 32b, RTICK_CMPR 32b, SHUTDOWN 33b, WRCLEAR_SOFTINT 33a, WRDCR 33a, WRGSR 33a 44, WRPCR 33a, WRPIC 33a, WRSOFTINT 33a, and WRTICK_CMPR 33a.

7 SPARC64 implementation-dependent instructions

The HAL SPARC64 implementation-dependent instructions are based on IMPDEP2, and are all based on the floating-point multiply-adder. The information on the HAL SPARC64 chips is from [5].

None of these instructions have been tested.

- 41a `<sparc64.spec 41a>≡`
`# requires fields.spec, types.spec and core.spec`
`# (depends on table 32 and fp register encoding)`
`<sparc64 patterns and constructors 42>`
- 41b `<sparc64-fields.spec 41b>≡`
`# sparc64-fields.spec`
`# goes after fields.spec (in fields declaration)`
`<sparc64 fields 41c>`

7.1 Ancillary state registers

The ancillary state registers used by the SPARC64 chips are [5, pp303–305,337–339]:

ASR Value	Name	Assembler	Access
0x12	HDW_MODE	%hardware_mode	RW
0x13	GSR	%graphic_status	RW
0x14	SET_SCHED_INT	%set_sched_int	W
0x15	CLEAR_SCHED_INT	%clear_sched_int	W
0x16	SCHED_INT	%sched_int	RW
0x17	TICK_MATCH	%tick_match	RW
0x19	SCRATCH	%scratch[0-3]	RW
0x1a	{BRKPT, BRK}_{ADDR, MASK}	{%dbreak, brk}_{addr, mask}	RW
0x1c	DFAULT_ADDR	%dfaddr	R
0x1d	DFTYPE	%dftype	R
0x1e	PM_{ { , CLR_ } { DIS, EN } , REG[0-5] }	%pm_{ { { , clr_ } { dis, en } } , [0-6] }	RW
0x1f	SCR	%scr	

Note the GSR (graphics status register): this is defined to allow emulation of the UltraSPARC VIS instruction set.

Some of these ASR's use non-standard instruction specifications requiring additional field specifications for selecting subregisters.

- 41c `<sparc64 fields 41c>≡` (41b) 45a▷
`asrsub 8:12 rdasrmbz 0:7 wrasrmbz 5:7`

Defines:

- asrsub, used in chunks 42 and 44.
- rdasrmbz, used in chunk 42.
- wrasrmbz, used in chunk 44.

For some strange reason, the prefix for the opcodes for some of the RDASR's have changed from RD to just R or to RD_. They are also inconsistent in the opcode and assembly naming; e.g., see 0x1a in the above table: for reading, they're calling BRKPT, but for writing they're called BRK; the assembler names them dbreak and brk. The performance monitoring PM ASR instructions are quite different depending whether reading or writing. All in all, these inconsistencies make it difficult to define a clean and simple specification.

42 *(sparc64 patterns and constructors 42)* ≡ (41a) 44▷

```

patterns
  [ RHDW_MODE RGSR _ _ RSCHED_INT RTICK_MATCH RIFTYPE RSCRATCHx
    RBRKPTx _ RDFFAULT_ADDR RDFTYPE RD_PMx RDSCR ]
  is RDASR & rsl = {0x12 to 0x1f}
constructors
  RHDW_MODE      "%hardware_mode", rd
  RGSR           "%graphic_status", rd
  RSCHED_INT     "%sched_int",      rd
  RTICK_MATCH    "%tick_match",     rd
  RIFTYPE        "%iftype",         rd
  RDFFAULT_ADDR "%dfaddr",          rd
  RDFTYPE        "%dftype",         rd
  RDSCR          "%scr",             rd
patterns
  [ RSCRATCH0 RSCRATCH1 RSCRATCH2 RSCRATCH3 ]
  is RSCRATCHx & asrsub = {0 to 3} & rdasrmbz = 0
  [ RBRKPT_ADDR RBRKPT_MASK ]
  is RBRKPTx & asrsub = [0 1] & rdasrmbz = 0
  [ RD_PM_VN RD_PM_REG0 RD_PM_REG1 RD_PM_REG2 RD_PM_REG3
    RD_PM_REG4 RD_PM_REG5 ]
  is RD_PMx & asrsub = {0 to 6} & rdasrmbz = 0
constructors
  RSCRATCH0 "%scratch0", rd
  RSCRATCH1 "%scratch1", rd
  RSCRATCH2 "%scratch2", rd
  RSCRATCH3 "%scratch3", rd
  RBRKPT_ADDR "%dbreak_addr", rd
  RBRKPT_MASK "%dbreak_mask", rd
  RD_PM_VN    "%pm0", rd
  RD_PM_REG0  "%pm1", rd
  RD_PM_REG1  "%pm2", rd
  RD_PM_REG2  "%pm3", rd
  RD_PM_REG3  "%pm4", rd
  RD_PM_REG4  "%pm5", rd
  RD_PM_REG5  "%pm6", rd

```

Defines:

```

RBRKPT_ADDR, never used.
RBRKPT_MASK, never used.
RBRKPTx, never used.
RDFFAULT_ADDR, never used.
RDFTYPE, never used.
RD_PM_REG0, never used.
RD_PM_REG1, never used.
RD_PM_REG2, never used.
RD_PM_REG3, never used.
RD_PM_REG4, never used.
RD_PM_REG5, never used.
RD_PM_VN, never used.
RD_PMx, never used.
RDSCR, never used.
RGSR, never used.
RHDW_MODE, never used.

```

RIFTYPE, never used.
RSCHED_INT, never used.
RSCRATCH0, never used.
RSCRATCH1, never used.
RSCRATCH2, never used.
RSCRATCH3, never used.
RSCRATCHx, never used.
RTICK_MATCH, never used.

Uses `asrsub 41c`, `rd 3c`, `RDASR 16`, `rdasrmbz 41c`, and `rs1 4a`.

Notice that the performance monitor register assembly naming is quite inconsistent with the opcodes, but I'm following the manual (it isn't actually clear whether `RD_PM_VN` is `%pm0` or `%pm6`).

The SPARC64 WRASR opcodes are just as confused as the RDASR ones. Some have a WR_ prefix, some have a WR prefix, and some don't have a prefix. Some of the performance monitor instructions don't have any parameters.

44 *(sparc64 patterns and constructors 42)+≡* (41a) <42 46a>

```

patterns
  [ WR_HDW_MODE WRGSR SET_SCHED_INT CLEAR_SCHED_INT WR_SCHED_INT
    WR_TICK_MATCH _ WR_SCRATCHx WR_BRKx _ _ WR_PMx WRSCR ]
  is WRASR & rd = {0x12 to 0x1f}
constructors
  WR_HDW_MODE      rsl, reg_or_imm, "%hardware_mode"
  WRGSR            rsl, reg_or_imm, "%graphic_status"
  SET_SCHED_INT   rsl, reg_or_imm, "%set_sched_int"
  CLEAR_SCHED_INT rsl, reg_or_imm, "%clear_sched_int"
  WR_SCHED_INT    rsl, reg_or_imm, "%sched_int"
  WR_TICK_MATCH   rsl, reg_or_imm, "%tick_match"
  WRSCR           rsl, reg_or_imm, "%scr"
patterns
  [ WR_SCRATCH0 WR_SCRATCH1 WR_SCRATCH2 WR_SCRATCH3 ]
  is WR_SCRATCHx & asrsub = {0 to 3} & wrasrmbz = 0 & i = 0
  [ WR_BRK_ADDR WR_BRK_MASK ]
  is WR_BRKx & asrsub = [0 1] & wrasrmbz = 0 & i = 0
  [ WR_PM_DIS WR_PM_CLR_DIS WR_PM_EN WR_PM_CLR_EN WR_PM_VN ]
  is WR_PMx & asrsub = {0 to 4} & wrasrmbz = 0 & i = 0
constructors
  WR_SCRATCH0 rsl, rs2, "%scratch0"
  WR_SCRATCH1 rsl, rs2, "%scratch1"
  WR_SCRATCH2 rsl, rs2, "%scratch2"
  WR_SCRATCH3 rsl, rs2, "%scratch3"
  WR_BRK_ADDR rsl, rs2, "%brk_addr"
  WR_BRK_MASK rsl, rs2, "%brk_mask"
  WR_PM_DIS   "%pm_dis"
  WR_PM_CLR_DIS "%pm_clr_dis"
  WR_PM_EN    "%pm_en"
  WR_PM_CLR_EN "%pm_clr_en"
  WR_PM_VN rsl, rs2, "%pm_vn"

```

Defines:

```

CLEAR_SCHED_INT, never used.
SET_SCHED_INT, never used.
WR_BRK_ADDR, never used.
WR_BRK_MASK, never used.
WR_BRKx, never used.
WRGSR, used in chunk 40.
WR_HDW_MODE, never used.
WR_PM_CLR_DIS, never used.
WR_PM_CLR_EN, never used.
WR_PM_DIS, never used.
WR_PM_EN, never used.
WR_PM_VN, never used.
WR_PMx, never used.
WR_SCHED_INT, never used.
WRSCR, never used.
WR_SCRATCH0, never used.
WR_SCRATCH1, never used.
WR_SCRATCH2, never used.
WR_SCRATCH3, never used.
WR_SCRATCHx, never used.
WR_TICK_MATCH, never used.

```

Uses asrsub 41c, i 4a, rd 3c, reg_or_imm 8b, rsl 4a, rs2 4a, WRASR 16, and wrasrmbz 41c.

Just to add to the confusion, the SPARC64 manual occasionally refers to the SIR instruction as WRSIR as though it is actually a register write to the SIR register (e.g., see [5, page 339, paragraph 3]).

7.2 Floating-point multiply-add

The floating-point multiply-add instructions are based on IMPDEP2. They require three additional fields (*var*, *size* and *rs3*).

```
45a <sparc64-fields 41c>+≡ (41b) <41c
    rs3 9:13 var 7:8 size 5:6
```

Defines:

```
rs3, used in chunk 45c.
size, used in chunk 46a.
var, used in chunk 46a.
```

They also require floating-point encoding of *rs3*: this is defined below.

```
45b <sparc64-fpre-fields.spec 45b>≡
    # sparc64-fpre-fields.spec
    # goes after fields.spec (in fields declaration)
    rs3fhi 10:13 rs3flo 9:9
```

Defines:

```
rs3fhi, used in chunk 45c.
rs3flo, used in chunk 45c.
```

```
45c <sparc64-fpre.spec 45c>≡
    # sparc64-fpre.spec
    constructors
    rs3fsv "%f"fr! : rs3fs { fr@[5:31] = 0 } is rs3 = fr
    rs3fdv "%f"fr! : rs3fd
        { fr@[6:31] = 0, fr@[0] = 0,
          fhi = fr@[1:4], flo = fr@[5] }
    is rs3fhi = fhi & rs3flo = flo
```

Defines:

```
rs3fd, used in chunk 46a.
rs3fdv, never used.
rs3fs, used in chunk 46a.
rs3fsv, never used.
```

Uses *rs3* 45a, *rs3fhi* 45b, and *rs3flo* 45b.

```
45d <sparc64-nofpre-fields.spec 45d>≡
    # sparc64-nofpre-fields.spec
    # goes after fields.spec (in fields declaration)
    # incompatible with sparc64-fpre*.spec
    rs3fs 9:13 rs3fd 9:13
```

Defines:

```
rs3fd, used in chunk 46a.
rs3fs, used in chunk 46a.
```

46a \langle sparc64 patterns and constructors 42 \rangle + \equiv (41a) <44

```

patterns
  fpmas is any of
    [ FMADDs FMSUBs FNMSUBs FNMADDs ],
    which is IMPDEP2 & size = 1 & var = {0 to 3}
  fpmad is any of
    [ FMADDd FMSUBd FNMSUBd FNMADDd ],
    which is IMPDEP2 & size = 2 & var = {0 to 3}
constructors
  fpmas rslfs, rs2fs, rs3fs, rdfs
  fpmad rslfd, rs2fd, rs3fd, rdfd

```

Defines:

```

FMADDd, never used.
FMADDs, never used.
FMSUBd, never used.
FMSUBs, never used.
FNMADDd, never used.
FNMADDs, never used.
FNMSUBd, never used.
FNMSUBs, never used.
fpmad, never used.
fpmas, never used.

```

Uses IMPDEP2 15, rdfd 12b 12c, rdfs 11a 12c, rslfd 12b 12c, rs2fd 12b 12c, rs3fd 45c 45d, rslfs 11a 12c, rs2fs 11a 12c, rs3fs 45c 45d, size 45a, and var 45a.

7.3 Assembler syntax

46b \langle sparc64-asm.spec 46b \rangle \equiv

```

# sparc64-asm.spec
# goes after asm.spec
 $\langle$ sparc64 assembler syntax 46c $\rangle$ 

```

The SPARC64 ASR opcodes, like the UltraSPARC's, are all rd or wr.

46c \langle sparc64 assembler syntax 46c \rangle \equiv (46b)

```

assembly opcode
  {R}{HDW_MODE,GSR,SCHED_INT,TICK_MATCH,IFTYPE} is $1D
  {RD}{FAULT_ADDR,FTYPE,SCR,_PM_{VN,REG{0,1,2,3,4,5}}}} is $1
  {R}{SCRATCH{0,1,2,3},BRKPT_{ADDR,MASK}} is $1D

  {WR}{_{HDW_MODE,SCHED_INT,TICK_MATCH},GSR,SCR} is $1
  {SET,CLEAR}_SCHED_INT is RD
  {WR}_{SCRATCH{0,1,2,3},BRK_{ADDR,MASK},PM_{_{CLR_}{DIS,EN},VN}} is $1

```

8 Synthetic instructions

The synthetic instructions are defined on pages 297–299 in the SPARC-V9 manual; their definitions appear below. When dealing with overloaded instructions like `call`, `mov`, and `clr`, we've reserved the standard name (e.g., `call`) for the most common variant, using either semi-mnemonic names (e.g., `movr` for move-register or `clrw` for clear-word, `calla` for call-address) or names with trailing underscores (e.g., `save_`) for other variants.

There's some problems with synthetics and the toolkit, so these haven't been tested.

```
47 <synth.spec 47>≡
constructors
# there's something wrong with synthetics.
# cmp rs1, reg_or_imm    is SUBcc(rs1, reg_or_imm, "%g0")
# jmp address_          is JMPL (address_, "%g0")
# calla address_        is JMPL (address_, "%o7")
  cmp rs1, reg_or_imm    is SUBcc(rs1, reg_or_imm, 0)
  jmp address_          is JMPL (address_, 0)
  calla address_        is JMPL (address_, 15)
# iprefetch reloc      is "BPN,a,pt" ("%xcc", reloc)
iprefetch reloc        is "BPN,a,pt" (2, reloc)
  tst rs2                is ORcc ("%g0", regROI(rs2), "%g0")
  ret                    is JMPL (dispA("%i7",8), "%g0")
  retl                   is JMPL (dispA("%o7",8), "%g0")
  restore_               is RESTORE ("%g0", regROI("%g0"), "%g0")
  save_                  is SAVE ("%g0", regROI("%g0"), "%g0")
  signx rd               is SRA (rd, regROS32("%g0"), rd)
  signx2 rs1, rd         is SRA (rs1, regROS32("%g0"), rd)
  not rd                 is XNOR(rd, regROI("%g0"), rd)
  not2 rs1, rd           is XNOR(rs1, regROI("%g0"), rd)
  neg rd                 is SUB ("%g0", regROI(rd), rd)
  neg2 rs2, rd           is SUB ("%g0", regROI(rs2), rd)
  cas [rs1], rs2, rd     is CASA (rs1, imm_casa_asi(0x80), rs2, rd)
  casl [rs1], rs2, rd    is CASA (rs1, imm_casa_asi(0x88), rs2, rd)
  casx [rs1], rs2, rd    is CASXA(rs1, imm_casa_asi(0x80), rs2, rd)
  casxl [rs1], rs2, rd   is CASXA(rs1, imm_casa_asi(0x88), rs2, rd)
  inc rd                 is ADD (rd, immROI(1), rd)
  inc2 val!, rd          is ADD (rd, immROI(val), rd)
  inccc rd               is ADDcc (rd, immROI(1), rd)
  inccc2 val!, rd        is ADDcc (rd, immROI(val), rd)
  dec rd                 is SUB (rd, immROI(1), rd)
  dec2 val!, rd          is SUB (rd, immROI(val), rd)
  deccc rd               is SUBcc (rd, immROI(1), rd)
  deccc2 val!, rd        is SUBcc (rd, immROI(val), rd)
  btst reg_or_imm, rs1   is ANDcc(rs1, reg_or_imm, "%g0")
  bset reg_or_imm, rd    is OR (rd, reg_or_imm, rd)
  bclr reg_or_imm, rd    is ANDN (rd, reg_or_imm, rd)
  btog reg_or_imm, rd    is XOR (rd, reg_or_imm, rd)
  clr rd                 is OR ("%g0", regROI("%g0"), rd)
  clrb [address_]        is STB ("%g0", address_)
  clrh [address_]        is STH ("%g0", address_)
  clrw [address_]        is STW ("%g0", address_)
  clrx [address_]        is STX ("%g0", address_)
  clruw rd               is SRL (rd, regROS32("%g0"), rd)
  clruw2 rs1, rd         is SRL (rs1, regROS32("%g0"), rd)
  mov reg_or_imm, rd     is OR ("%g0", reg_or_imm, rd)
  movr rs2, rd           is OR ("%g0", regROI(rs2), rd)
```

Defines:

bclr, used in chunk 49b.
 bset, used in chunk 49b.
 btog, used in chunk 49b.
 btst, used in chunk 49b.
 calla, used in chunk 49b.
 cas, used in chunk 49b.
 casl, used in chunk 49b.
 casx, used in chunk 49b.
 casxl, used in chunk 49b.
 clr, used in chunk 49.
 clrb, used in chunk 49b.
 clrh, used in chunk 49b.
 clruw, used in chunk 49.
 clruw2, used in chunk 49b.
 clrw, used in chunk 49b.
 clrx, used in chunk 49b.
 cmp, used in chunk 49b.
 dec, used in chunk 49.
 dec2, used in chunk 49b.
 deccc, used in chunk 49.
 deccc2, used in chunk 49b.
 inc, used in chunk 49.
 inc2, used in chunk 49b.
 inccc, used in chunk 49.
 inccc2, used in chunk 49b.
 iprefetch, used in chunk 49b.
 jmp, used in chunk 49b.
 mov, used in chunk 49.
 movr, used in chunk 49b.
 neg, used in chunk 49.
 neg2, used in chunk 49b.
 not, used in chunk 49.
 not2, used in chunk 49b.
 restore_, used in chunk 49b.
 ret, used in chunk 49b.
 retl, used in chunk 49b.
 save_, used in chunk 49b.
 signx, used in chunk 49.
 signx2, used in chunk 49b.
 tst, used in chunk 49b.

Uses a 3c, ADD 15, ADDcc 15, address_9a, ANDcc 15, ANDN 15, CASA 18, CASXA 18, dispA 9a, imm_casa_asi 10b, immROI 8b, JMLP 15, OR 15, ORcc 15, rd 3c, reg_or_imm 8b, regROI 8b, reloc 27a, RESTORE 15, rs1 4a, rs2 4a, SAVE 15, SRA 17b, SRL 17b, STB 18, STH 18, STW 18, STX 18, SUB 15, SUBcc 15, XNOR 15, and XOR 15.

Some of the more esoteric synthetic instructions (e.g., `mov` the `y` or `asr` registers) were not defined.

There are a lot of synonyms defined in the SPARC-V9 specification, such as for the condition-code checks, `Xnz` is equivalent to `Xne`. We ignore this part of the specification, since we're not interested in actual assembly, merely decoding and encoding. It would not be difficult to add these, I think, but would require additional testing. For example, (mind you this won't work)

48a *<example of synonyms 48a>* ≡

```

constructors
  fbnz^a reloc is FBNE(a, reloc)
  fbz^a  reloc is FBE (a, reloc)

```

Uses a 3c and reloc 27a.

The SPARC-V9 has several conditionally assembled instructions (`setuw = set`, `setsw` and `setx`), unlike SPARC-V8 which only has one. We currently don't encode or decode any of them (see the SPARC-V8 spec for how to do it).

8.1 Assembler syntax

48b *<synth-asm.spec 48b>* ≡

```

# synth-asm.spec
# goes after asm.spec
<synth assembler syntax 49a>

```

The synthetic assembly codes were named differently due to overloading. We can allow them to be overloaded in assembler.

```
49a <synth assembler syntax 49a>≡ (48b)
    assembly opcode
      {call}a          is $1
      {save,restore}_ is $1
      {clr}w           is $1
      {mov}r           is $1
      {signx,not,neg,inc,inccc,dec,deccc,clruw}2 is $1
```

Uses a 3c, clr 47, clruw 47, dec 47, deccc 47, inc 47, inccc 47, mov 47, neg 47, not 47, and signx 47.

8.2 Validating

See section 11 for rationale. There seems to be a problem with the synthetics: don't use them (20000809).

```
49b <synth-check.spec 49b>≡
    ## there is something wrong with synthetics
    discard
    cmp jmp calla iprefetch tst ret retl restore_ save_ signx signx2 not not2 neg neg2 cas ca
    ccc inccc2 dec dec2 deccc deccc2 btst bset bclr btog clr clrb clrh clrw clrx clruw clruw2
```

Uses bclr 47, bset 47, btog 47, btst 47, calla 47, cas 47, casl 47, casx 47, casxl 47, clr 47, clrb 47, clrh 47, clruw 47, clruw2 47, clrw 47, clrx 47, cmp 47, dec 47, dec2 47, deccc 47, deccc2 47, inc 47, inc2 47, inccc 47, inccc2 47, iprefetch 47, jmp 47, mov 47, movr 47, neg 47, neg2 47, not 47, not2 47, restore_ 47, ret 47, retl 47, save_ 47, signx 47, signx2 47, and tst 47.

9 Assembler syntax

This section is for the generation of assembler encoders, so clients can emit assembly. In order to do so, we must convert some instructions from the opcode syntax to assembler syntax. These are different since there is extra syntax in assembly language to allow instructions to be overloaded.

```
50a <asm.spec 50a>≡
    # asm.spec
    # requires fields.spec, core.spec
    <assembler syntax 50b>
```

Several of the floating-point assembly names are different to the opcodes. The assembler can use additional information (the operands) to disambiguate. Firstly, there are the load and store floating-point instructions, which we reduce down to a few cases (essentially involving removing F or FSR):

```
50b <assembler syntax 50b>≡ (50a) 50c>
    assembly opcode
    {LD,ST}{F,FSR} is $1
    {LD,ST}{D,Q}F is $1$2
    {LDX,STX}FSR is $1
    {LD,ST}F{A} is $1$2
    {LD,ST}{D,Q}F{A} is $1$2$3
```

Uses LDX 18 and STX 18.

The FMOV variants are many and confusing. I'm not sure if the last one has to be here, but it makes more sense to read it this way. Firstly there is a special case to deal with the simple moves, then there's the floating-point condition moves, then the integer register value moves, then the integer register condition moves. Note the order of the third variant is correct! (the FMOV{S,D,Q}R may need to change to assembly component?)

```
50c <assembler syntax 50b>+≡ (50a) <50b 50d>
    assembly opcode
    {FMOV}{s,d,q} is $1$2
    {FMOV}{S,D,Q}F{*} is $1$2$3
    {FMOV}{S,D,Q}{R}{*} is $1$3$2$4
    {FMOV}{S,D,Q}{*} is $1$2$3
```

As for FMOV, there are many MOV variants. Firstly there are the floating-point register condition moves, then the integer register value moves, then the integer register condition moves.

```
50d <assembler syntax 50b>+≡ (50a) <50c 50e>
    assembly opcode
    {MOV}F{*} is $1$2
    {MOV}{R}{*} is $1$2$3
    {MOV}{*} is $1$2
```

The lines which simply match and reproduce everything are probably not necessary, but make it easier to understand.

The RDxxx and WRxxx assembler opcodes are all the same; they are disambiguated by strings in the operands (generated in the constructors).

```
50e <assembler syntax 50b>+≡ (50a) <50d 51>
    assembly opcode
    {RD}{Y,CCR,ASI,TICK,PC,FPRS,ASR} is $1
    {WR}{Y,CCR,ASI,FPRS,ASR} is $1
```

The branches with prediction have the same assembler codes as those without (as the ones without prediction are deprecated, they shouldn't be generated by assembler anyway, unless forced to by large enough displacements). We have to remove the `P` from the constructor, using the `assembly component` syntax since the branches are combinations of opcodes and other components (`annul` and `predict`). We specialise the `BPOS` since it would otherwise be replaced with `BOS`!

```
51 <assembler syntax 50b>+≡ (50a) <50e
    assembly component
      {BPOS}          is $1
      {,F}{B}P{*}   is $1$2$3
```

10 Application-specific specifications for simulation

See the SPARC-V8 spec for some application-specific specifications. We may need some specification for emitting relocatable addresses (64-bit!) if we want to use this specification for encoding (e.g., SPARC-on-SPARC). We also may want to guarantee some of the register values.

```
52 <decode.spec 52>≡  
    address type is "unsigned"  
    address to integer using "%a"  
    address add using "%a+%o"  
    fetch 32 using "fetch_word(%a)"
```

Uses a 3c.

11 Validating against the SPARC-V9 assembler

The syntax for all of the SPARC-V9 instructions seems to be recognised by the Solaris assembler (unlike the SPARC-V8/SunOS assembler). We use `discard` to remove the instructions that we have already tested from checking.

```
53 <check.spec 53>≡
# method:
# list all constructors, then test some by commenting out one of these lines.
# some constructors have been separated out for individual testing.

# 20000610: all tested (except synthetics), BPr is dodgy.
# 20000620: after rearranging, everything is ok except synthetics and BPr
# 20000627: tested VIS instructions: ok
# 20000629: tested all fp instructions with new constructors: ok
# 20000630: BPr now works

#discard
## these need to be individually and/or manually tested:

## prefetch/rdasr/rdpr: due to problems in test generation, these need
## to be tested by removing the equations from their constructors
## PREFETCHA: there is a bug in sparcworks 4.X assembler/fixed in 5.0
# PREFETCH PREFETCHA
# RDASR WRASR
# RDPR WRPR

## all quads have to be tested by commenting out the "invalid" constructors
# FiTOq FsTOq FxTOq FdTOq FqTOi FqTOs FqTOx FqTOd FMOVq FNEGq FABSq FSQRTq FADDq FSUBq F
MULq FCMpq FCMPEq

## BPr is tricky but now works (20000630)
# "BRZ,pt" "BRZ,pn" "BRZ,a,pt" "BRZ,a,pn" "BRLEZ,pt" "BRLEZ,pn" "BRLEZ,a,pt" "BRLEZ,a,pn"

#discard
## these have been checked:
#LDSB LDSH LDSW LDUB LDUH LDUW LDX LDSTUB SWAP LDD STB STH STW STX STD LDSBA LDSHA LDSWA
STUBA SWAPA LDDA STBA STHA STWA STXA STDA
#CASA CASXA LDF LDDF LDQF STF STDF STQF LDFA LDDFA LDQFA STFA STDFA STQFA LDFSR LDXFSR ST
CCR RDASI RTICK RDPC RDFPRS STBAR MEMBAR WRY
#WRCCR WRASI WRFPRS SIR AND ANDcc ANDN ANDNcc OR ORcc ORN ORNcc XOR XORcc XNOR XNORcc ADD
Dcc ADDC ADDCcc TADDcc TADDccTV SUB SUBcc SUBC
#SUBCcc TSUBcc TSUBccTV MULScc UMUL SMUL UMULcc SMULcc UDIV SDIV UDIVcc SDIVcc MULX SDIVX
STORE SLL SRL SRA SLLX SRLX SRAX POPC BNE "BNE,a" BNEG "BNEG,a" BPOS
#"BPOS,a" BE "BE,a" BGE "BGE,a" BGU "BGU,a" BG "BG,a" BCC "BCC,a" BCS "BCS,a" BL "BL,a" B
#FBLG "FBLG,a" FBNE "FBNE,a" FBULE "FBULE,a" FBE "FBE,a" FBU "FBU,a" FBGE "FBGE,a" FBUG "
#"FBLE,a" "BPNE,pt" "BPNE,pn" "BPNE,a,pt" "BPNE,a,pn" "BPNEG,pt" "BPNEG,pn" "BPNEG,a,pt"
NEG,a,pn" "BPPOS,pt" "BPPOS,pn" "BPPOS,a,pt" "BPPOS,a,pn" "BPE,pt" "BPE,pn" "BPE,a,pt" "B
#"BPCC,pt" "BPCC,pn" "BPCC,a,pt" "BPCC,a,pn" "BPCS,pt" "BPCS,pn" "BPCS,a,pt" "BPCS,a,pn"
#"BPVC,pn" "BPVC,a,pt" "BPVC,a,pn" "BPVS,pt" "BPVS,pn" "BPVS,a,pt" "BPVS,a,pn" "FBPUE,pt"
PLG,pt" "FBPLG,pn" "FBPLG,a,pt" "FBPLG,a,pn" "FBPNE,pt" "FBPNE,pn" "FBPNE,a,pt" "FBPNE,a,
#"FBPU,a,pt" "FBPU,a,pn" "FBPGE,pt" "FBPGE,pn" "FBPGE,a,pt" "FBPGE,a,pn" "FBPUG,pt" "FBPU
#"FBPL,a,pn" "FBPA,pt" "FBPA,pn" "FBPA,a,pt" "FBPA,a,pn" "FBPN,pt" "FBPN,pn" "FBPN,a,pt"
PLE,pt" "FBPLE,pn" "FBPLE,a,pt" "FBPLE,a,pn" TNE TNEG TPOS TE TGE TGU TG TCC TCS TL TLEU
#CALL MOVNE MOVNEG MOVPOS MOVE MOVGE MOVGU MOVG MOVCC MOVCS MOVL MOVLEU MOVA MOVN MOVLE M
```

```

FUE MOVFLG MOVFNE MOVFULE MOVFE MOVFU MOVFGE MOVFUG MOVFUL MOVFG MOVFO MOVFUGE
#MOVFL MOVFA MOVFN MOVFLE FMOVSNE FMOVSNEG FMOVSPS FMOVSE FMOVSGE FMOVSGU FMOVSG FMOVSCO
FUE FMOVSFLG FMOVSFNE FMOVSFULE FMOVSFLE FMOVSFU FMOVSFGE FMOVSFUG FMOVSFUL
#FMOVSTG FMOVSTO FMOVSTFUGE FMOVSTFL FMOVSTFA FMOVSTFN FMOVSTFLE FMOVSDNE FMOVSDNEG FMOVSDPOS FMO
DLE FMOVSDVC FMOVSDVS FMOVDFUE FMOVDFLG FMOVDFNE FMOVDFULE FMOVDFE FMOVDFU
#FMOVDFGE FMOVDFUG FMOVDFUL FMOVDFG FMOVDFO FMOVDFUGE FMOVDFL FMOVDFFA FMOVDFN FMOVDFLE FM
POS FMOVQE FMOVQGE FMOVQGU FMOVQG FMOVQCC FMOVQCS FMOVQL FMOVQLEU FMOVQA FMOVQN FMOVQLE F
FUE FMOVQFLG FMOVQFNE
#FMOVQFULE FMOVQFE FMOVQFU FMOVQFGE FMOVQFUG FMOVQFUL FMOVQFG FMOVQFO FMOVQFUGE FMOVQFL F
#MOVRLZ MOVRLZ MOVRLZ MOVRLZ MOVRLZ MOVRLZ FMOVSRZ FMOVSRLEZ FMOVSRLEZ FMOVSRLEZ FMOVSRLEZ FMOVSRGZ
RGEZ FMOVDRZ FMOVDRLEZ FMOVDRLEZ FMOVDRLEZ FMOVDRLEZ FMOVDRLEZ FMOVDRLEZ FMOVDRLEZ FMOVDRLEZ FMOVDRLEZ FMOVDRLEZ
#FITOs FSTOx FITOd FSTOd FxTOs FdTOi FdTOs FMOVd FNEGd FABSD FSQRtd FdTOx FxTOd FADDs FSU
#FADDd FSUBd FMULD FDIVd FsmULD FCMPs FCMPEs FCMPd FCMPed NOP DONE RETRY SAVED RESTORED F
TURN ILLTRAP SETHI

```

Uses a 3c, ADD 15, ADDC 15, ADDcc 15, ADDCcc 15, AND 15, ANDcc 15, ANDN 15, ANDNcc 15, BPr 14, CASXA 18, DONE 17a, FABSD 20, FABSQ 20, FABSS 20, FADDq 20, FADDs 20, FCMPd 22a, FCMPed 22a, FCMPEq 22a, FCMPq 22a, FDIvd 20, FDIvq 20, FDIvs 20, FdMULq 20, FdTOi 20, FdTOq 20, FdTOs 20, FdTOx 20, FiTOd 20, FiTOq 20, FLUSH 15, FLUSHW 15, FMOVd 20, FMOVq 20, FMOVs 20, FMULD 20, FMULq 20, FMULs 20, FNEGd 20, FNEGq 20, FNEGs 20, FqTOd 20, FqTOi 20, FqTOs 20, FqTOx 20, FsmULD 20, FSQRtd 20, FSQRtq 20, FSQRts 20, FstOd 20, FstOi 20, FstOq 20, FstOx 20, FSUBd 20, FSUBq 20, FSUBs 20, FxTOd 20, FxTOq 20, FxTOs 20, ILLTRAP 13e, Jmpl 15, LDD 18, LDDA 18, LDDF 18, LDDFA 18, LDF 18, LDFA 18, LDFSR 19a, LDQF 18, LDQFA 18, LDSBA 18, LDSH 18, LDSHA 18, LDSTUB 18, LDSTUBA 18, LDSW 18, LDSWA 18, LDUB 18, LDUBA 18, LDUH 18, LDUHA 18, LDUW 18, LDUWA 18, LDX 18, LDXA 18, LDXFSR 19a, MEMBAR 16, MULScC 15, MULX 15, NOP 13e, OR 15, ORcc 15, ORN 15, ORNcc 15, POPC 17d, PREFETCH 19b, PREFETCHA 19b, RDASI 16, RDASR 16, RDCCR 16, RDFPRS 16, RDPC 16, RDPR 17c, RTICK 16, RDY 16, RESTORE 15, RESTORED 17a, RETRY 17a, RETURN 15, SAVE 15, SAVED 17a, SDIV 15, SDIVcc 15, SDIVX 15, SETHI 13e, SIR 16, SLL 17b, SLLX 17b, SMUL 15, SMULcc 15, SRA 17b, SRAX 17b, SRL 17b, SRLX 17b, STB 18, STBA 18, STBAR 16, STD 18, STDA 18, STDF 18, STDFa 18, STF 18, STFA 18, STFSR 19a, STH 18, STHA 18, STQF 18, STQFA 18, STW 18, STWA 18, STX 18, STXA 18, STXFSR 19a, SUB 15, SUBC 15, SUBcc 15, SWAP 18, SWAPA 18, TADDcc 15, TADDccTV 15, TSUBcc 15, TSUBccTV 15, UDIV 15, UDIVcc 15, UDIVX 15, UMUL 15, UMULcc 15, WRASI 16, WRASR 16, WRFPRS 16, WRPR 17c, WRY 16, XNOR 15, XNORcc 15, XOR 15, and XORcc 15.

We don't have to put any special headers in assembly source used by the checker.

54 *(sparc-v9-checker.s 54)*≡

References

- [1] Ramsey, Norman and Mary F. Fernandez. *New Jersey Machine-Code Toolkit Architecture Specifications*. Technical Report TR-470-94, Department of Computer Science, Princeton University, 1994.
- [2] Ramsey, Norman and Mary F. Fernandez. The New Jersey Machine-Code Toolkit. In *Proceedings of the 1995 USENIX Technical Conference*, pages 289–302. New Orleans, LA, 1995.
The New Jersey Machine-Code Toolkit web-page is at <http://www.eecs.harvard.edu/~nr/toolkit/>.
- [3] SPARC International, Inc. *The SPARC Architecture Manual (version SAV09R1459912)*. Prentice-Hall, New Jersey, 1994–2000.
Download from <http://www.sparc.com/standards.html> (direct: <http://www.sparc.com/standards/v9.ps.Z>).
- [4] Sun Microelectronics. *UltraSPARC(tm)-I User's Manual (Revision 1.4)*. Sun Microsystems, Inc, California, 1996.
New version 1997-02 (including UltraSPARC-II) download from <http://www.sun.com/microelectronics/UltraSPARC-II/> (direct: <http://www.sun.com/microelectronics/manuals/ultrasparc/802-7220-02.pdf>). References to this document may include two page references; the first refers to Revision 1.4, the second to 1997-02.
- [5] HAL Computer Systems, Inc. *SPARC64-III User's Guide*. HAL Computer Systems, Inc, California, 1998.
Download from <http://www.sparc.com/standards.html> (direct: <http://www.sparc.com/standards/sparc64.ps.Z>).

A List of Chunks

<i><asm.spec 50a></i>	<i><sparc64 assembler syntax 46c></i>
<i><assembler syntax 50b></i>	<i><sparc64 fields 41c></i>
<i><check.spec 53></i>	<i><sparc64 patterns and constructors 42></i>
<i><core-full.spec 13c></i>	<i><sparc64-asm.spec 46b></i>
<i><core-sim.spec 13b></i>	<i><sparc64-fields.spec 41b></i>
<i><core.spec 13a></i>	<i><sparc64-fpre-fields.spec 45b></i>
<i><decode.spec 52></i>	<i><sparc64-fpre.spec 45c></i>
<i><example of synonyms 48a></i>	<i><sparc64-nofpre-fields.spec 45d></i>
<i><fdeqnv 11d></i>	<i><sparc64.spec 41a></i>
<i><field specifications 3b></i>	<i><sparc-v9-checker.s 54></i>
<i><fieldinfo specifications 5c></i>	<i><synth assembler syntax 49a></i>
<i><fields-names.spec 5a></i>	<i><synth-asm.spec 48b></i>
<i><fields.spec 3a></i>	<i><synth-check.spec 49b></i>
<i><fpre type specifications 11a></i>	<i><synth.spec 47></i>
<i><fqeqni 11f></i>	<i><type specifications 8b></i>
<i><fqeqnv 11e></i>	<i><types-fpre-fields.spec 11g></i>
<i><frdqlegal 11b></i>	<i><types-fpre.spec 10c></i>
<i><frdqtemp 11c></i>	<i><types-nofpre-fields.spec 12c></i>
<i><fseqnv 10d></i>	<i><types.spec 8a></i>
<i><full pattern and constructor specifications 23c></i>	<i><ultrasparc assembler syntax 39c></i>
<i><pattern and constructor specifications 13d></i>	<i><ultrasparc patterns and constructors 32b></i>
<i><properties of integer-register fields 5b></i>	<i><ultrasparc-asm.spec 39b></i>
<i><rdfass 12a></i>	<i><ultrasparc-check.spec 40></i>
<i><rs1fass 11h></i>	<i><ultrasparc-fields.spec 33c></i>
<i><rs2fass 11i></i>	<i><ultrasparc.spec 32a></i>
<i><sim pattern and constructor specifications 23d></i>	

B Index

a: [3c](#), [6c](#), [27b](#), [27c](#), [27d](#), [27e](#), [29c](#),
[29d](#), [30c](#), [47](#), [48a](#), [49a](#), [52](#), [53](#)
absoluteA: [9a](#)
absoluteRPI: [9b](#)
absoluteSTN: [9b](#)
ADD: [15](#), [26c](#), [47](#), [53](#)
ADDC: [15](#), [26c](#), [53](#)
ADDcc: [15](#), [26c](#), [47](#), [53](#)
ADDcc: [15](#), [26c](#), [53](#)
address.: [9a](#), [23a](#), [23c](#), [23d](#), [24a](#),
[24b](#), [24c](#), [24d](#), [31c](#), [47](#)
ALIGNADDRESS: [34b](#), [40](#)
ALIGNADDRESS_LITTLE: [34b](#)
alu: [26c](#)
AND: [15](#), [26c](#), [53](#)
ANDcc: [15](#), [26c](#), [47](#), [53](#)
ANDN: [15](#), [26c](#), [47](#), [53](#)
ANDNcc: [15](#), [26c](#), [53](#)
arith: [26c](#)
ARRAY16: [34b](#), [40](#)
ARRAY32: [34b](#), [40](#)
ARRAY8: [34b](#), [40](#)
asi_address: [10a](#), [23b](#), [23c](#),
[23d](#), [24a](#), [24c](#), [24d](#)
asrsub: [41c](#), [42](#), [44](#)
bclr: [47](#), [49b](#)
Bicc: [13e](#), [27b](#), [27c](#)
BPcc: [13e](#), [27d](#), [27e](#)
BPr: [14](#), [29c](#), [29d](#), [53](#)
BPrx: [13e](#), [14](#)
bset: [47](#), [49b](#)
btog: [47](#), [49b](#)
btst: [47](#), [49b](#)
CALL: [13d](#), [28c](#), [28d](#)
calla: [47](#), [49b](#)
cas: [47](#), [49b](#)
casa: [19c](#), [23e](#)
CASA: [18](#), [19c](#), [47](#)
casa_asi: [10b](#), [23e](#)
casl: [47](#), [49b](#)
casx: [47](#), [49b](#)
CASXA: [18](#), [19c](#), [47](#), [53](#)
casxl: [47](#), [49b](#)
cc2_f4: [4b](#), [28e](#), [28f](#)
CLEAR_SCHED_INT: [44](#)
clr: [47](#), [49a](#), [49b](#)
clrb: [47](#), [49b](#)
clrh: [47](#), [49b](#)
clruw2: [47](#), [49b](#)
clruw: [47](#), [49a](#), [49b](#)
clrw: [47](#), [49b](#)
clrx: [47](#), [49b](#)
cmp: [47](#), [49b](#)
dec2: [47](#), [49b](#)
dec: [47](#), [49a](#), [49b](#)
deccc2: [47](#), [49b](#)
deccc: [47](#), [49a](#), [49b](#)
d16hi: [3c](#), [29c](#), [29d](#)
disp19: [3c](#), [27d](#), [27e](#)
disp22: [3c](#), [27b](#), [27c](#)
disp30: [3b](#), [28c](#), [28d](#)
dispA: [9a](#), [47](#)
dispRPI: [9b](#)
dispSTN: [9b](#)
d16lo: [3c](#), [29c](#), [29d](#)
DONE: [17a](#), [31b](#), [53](#)
DONEx: [15](#), [17a](#)
EDGE16: [34b](#), [40](#)
EDGE32: [34b](#), [40](#)
EDGE8: [34b](#), [40](#)
EDGE16L: [34b](#), [40](#)
EDGE32L: [34b](#), [40](#)
EDGE8L: [34b](#), [40](#)
FABSd: [20](#), [53](#)
FABSq: [20](#), [53](#)
FABSs: [20](#), [53](#)
FADDd: [20](#)
FADDq: [20](#), [53](#)
FADDs: [20](#), [53](#)
FALIGNDATA: [36](#), [40](#)
FAND: [37](#), [40](#)
FANDNOT1: [37](#), [40](#)
FANDNOT2: [37](#), [40](#)
FANDNOT1S: [37](#), [40](#)
FANDNOT2S: [37](#)
FANDS: [37](#), [40](#)
FBfcc: [13e](#), [27b](#), [27c](#)
FBPfcc: [13e](#), [27d](#), [27e](#)
fcc_f2: [3c](#), [6b](#), [27d](#), [27e](#)
fcc_f3: [4a](#), [6b](#), [31a](#)
fcc_f4: [4b](#), [6b](#), [28e](#)
FCMPd: [22a](#), [53](#)
FCMPed: [22a](#), [53](#)
FCMPEQ16: [35](#), [40](#)
FCMPEQ32: [35](#)
FCMPEq: [22a](#), [53](#)
FCMPGT16: [35](#), [40](#)
FCMPGT32: [35](#), [40](#)
FCMPLE16: [35](#), [40](#)
FCMPLE32: [35](#), [40](#)
FCMPNE16: [35](#), [40](#)
FCMPNE32: [35](#), [40](#)
FCMPq: [22a](#), [53](#)
fcn: [4a](#), [17a](#), [24c](#), [24d](#)
fcompared: [22a](#), [31a](#)
fcompareq: [22a](#), [31a](#)
fcompares: [22a](#), [31a](#)
fcond_f2: [3c](#), [6d](#), [27b](#), [27c](#), [27d](#),
[27e](#)
fcond_f4: [4b](#), [6d](#), [28e](#), [29a](#)
FCPEs: [22a](#)
FCPs: [22a](#)
FDIVd: [20](#), [53](#)
FDIVq: [20](#), [53](#)
FDIVs: [20](#), [53](#)
FdMULq: [20](#), [31a](#), [53](#)
f2dTOd: [20](#), [30d](#)
FdTOi: [20](#), [53](#)
f2dTOq: [20](#), [30d](#)
FdTOq: [20](#), [53](#)
f2dTOS: [20](#), [30d](#)
FdTOS: [20](#), [53](#)
FdTOx: [20](#), [53](#)
FEXPAND: [36](#), [40](#)
FiTOd: [20](#), [53](#)
FiTOq: [20](#), [53](#)
FiTOS: [20](#)
float2: [20](#)
float3: [20](#)
float3d: [20](#), [31a](#)
float3q: [20](#), [31a](#)
float3s: [20](#), [31a](#)
FLUSH: [15](#), [31c](#), [53](#)
FLUSHW: [15](#), [31b](#), [53](#)
FMADDd: [46a](#)
FMADDs: [46a](#)
FMOVd: [20](#), [53](#)
FMOVdcc: [22a](#), [29a](#), [29b](#)
FMOVDR: [22a](#), [30a](#), [30b](#)
FMOVq: [20](#), [53](#)
FMOVQcc: [22a](#), [29a](#), [29b](#)
FMOVQR: [22a](#), [30a](#), [30b](#)
FMOVs: [20](#), [53](#)
FMOVSc: [22a](#), [29a](#), [29b](#)
FMOVSR: [22a](#), [30a](#), [30b](#)
FMSUBd: [46a](#)
FMSUBs: [46a](#)
FMULd: [20](#), [53](#)
FMULD8SUx16: [35](#), [40](#)
FMULD8ULx16: [35](#), [40](#)
FMULq: [20](#), [53](#)
FMULs: [20](#), [53](#)
FMUL8SUx16: [35](#)
FMUL8ULx16: [35](#), [40](#)
FMUL8x16: [35](#), [40](#)
FMUL8x16AL: [35](#), [40](#)

FMUL8x16AU: [35](#)
 FNAND: [37](#), [40](#)
 FNANDS: [37](#), [40](#)
 FNEGd: [20](#), [53](#)
 FNEGq: [20](#), [53](#)
 FNEGs: [20](#), [53](#)
 FNMADDd: [46a](#)
 FNMADDs: [46a](#)
 FNMSUBd: [46a](#)
 FNMSUBs: [46a](#)
 FNOR: [37](#), [40](#)
 FNORS: [37](#), [40](#)
 FNOT1: [37](#), [40](#)
 FNOT2: [37](#)
 FNOT1S: [37](#), [40](#)
 FNOT2S: [37](#), [40](#)
 FONE: [37](#), [40](#)
 FONES: [37](#)
 FOR: [37](#)
 FORNOT1: [37](#), [40](#)
 FORNOT2: [37](#), [40](#)
 FORNOT1S: [37](#), [40](#)
 FORNOT2S: [37](#), [40](#)
 FORS: [37](#), [40](#)
 FPACK16: [35](#), [40](#)
 FPACK32: [35](#), [40](#)
 FPACKFIX: [35](#), [40](#)
 FPADD16: [36](#), [40](#)
 FPADD32: [36](#), [40](#)
 FPADD16S: [36](#), [40](#)
 FPADD32S: [36](#), [40](#)
 fpmad: [46a](#)
 fpmas: [46a](#)
 FPMERGE: [36](#), [40](#)
 FPop1: [15](#), [20](#)
 FPop2: [15](#), [22a](#)
 FPSUB16: [36](#), [40](#)
 FPSUB32: [36](#)
 FPSUB16S: [36](#), [40](#)
 FPSUB32S: [36](#), [40](#)
 f2qTOd: [20](#), [30d](#)
 FqTOd: [20](#), [53](#)
 FqTOi: [20](#), [53](#)
 f2qTOq: [20](#), [30d](#)
 f2qTOs: [20](#), [30d](#)
 FqTOs: [20](#), [53](#)
 FqTOx: [20](#), [53](#)
 FsMULd: [20](#), [31a](#), [53](#)
 FSQRTd: [20](#), [53](#)
 FSQRTq: [20](#), [53](#)
 FSQRTs: [20](#), [53](#)
 FSRC1: [37](#), [40](#)
 FSRC2: [37](#), [40](#)
 FSRC1S: [37](#), [40](#)
 FSRC2S: [37](#), [40](#)
 f2sTOd: [20](#), [30d](#)
 FsTOd: [20](#), [53](#)
 FsTOi: [20](#), [53](#)
 f2sTOq: [20](#), [30d](#)
 FsTOq: [20](#), [53](#)
 f2sTOs: [20](#), [30d](#)
 FsTOx: [20](#), [53](#)
 FSUBd: [20](#), [53](#)
 FSUBq: [20](#), [53](#)
 FSUBs: [20](#), [53](#)
 funaryd: [20](#)
 funaryq: [20](#)
 funarys: [20](#)
 FXNOR: [37](#), [40](#)
 FXNORS: [37](#), [40](#)
 FXOR: [37](#), [40](#)
 FXORS: [37](#), [40](#)
 FxTOd: [20](#), [53](#)
 FxTOq: [20](#), [53](#)
 FxTOs: [20](#), [53](#)
 FZERO: [37](#), [40](#)
 FZEROS: [37](#), [40](#)
 generalA: [9a](#)
 generalSTN: [9b](#)
 i: [4a](#), [8b](#), [8c](#), [10b](#), [16](#), [44](#)
 icc_f2: [3c](#), [6b](#), [27d](#), [27e](#)
 icc_f4: [4b](#), [6b](#), [28a](#), [28b](#), [28e](#), [28f](#)
 icond_f2: [3c](#), [6d](#), [27b](#), [27c](#), [27d](#),
 [27e](#), [28a](#), [28b](#)
 icond_f4: [4b](#), [6d](#), [28e](#), [28f](#), [29a](#),
 [29b](#)
 ILLTRAP: [13e](#), [27a](#), [31c](#), [53](#)
 imm22: [3c](#), [13e](#), [27a](#), [31c](#), [31d](#)
 imm_asi: [4a](#), [10a](#), [10b](#)
 imm_asi_address: [10a](#)
 imm_casa_asi: [10b](#), [47](#)
 immROI1: [8c](#), [8c](#)
 immROI7: [8c](#), [9b](#)
 immROI: [8b](#), [9a](#), [9b](#), [47](#)
 immROS3: [8c](#)
 immROS6: [8c](#)
 imp_asi_address: [10a](#)
 imp_casa_asi: [10b](#)
 IMPDEP1: [15](#), [33b](#), [34a](#)
 IMPDEP2: [15](#), [46a](#)
 impldep1: [4a](#)
 impldep2: [4a](#)
 inc2: [47](#), [49b](#)
 inc: [47](#), [49a](#), [49b](#)
 inccc2: [47](#), [49b](#)
 inccc: [47](#), [49a](#), [49b](#)
 indexA: [9a](#)
 indexRA: [9b](#)
 indexSTN: [9b](#)
 indirectA: [9a](#)
 indirectRA: [9b](#)
 indirectRPI: [9b](#)
 indirectSTN: [9b](#)
 inst: [3b](#)
 iprefetch: [47](#), [49b](#)
 jmp: [47](#), [49b](#)
 JMPL: [15](#), [31c](#), [47](#), [53](#)
 LDD: [18](#), [23c](#), [23d](#), [53](#)
 LDDA: [18](#), [23c](#), [23d](#), [53](#)
 LDDF: [18](#), [24a](#), [53](#)
 LDDFA: [18](#), [24a](#), [53](#)
 LDF: [18](#), [24a](#), [53](#)
 LDFA: [18](#), [24a](#), [53](#)
 LDFSR: [19a](#), [53](#)
 ldfsr: [19a](#), [24b](#)
 LDFSR: [19a](#), [53](#)
 LDFSRx: [18](#), [19a](#)
 LDQF: [18](#), [24a](#), [53](#)
 LDQFA: [18](#), [24a](#), [53](#)
 LDSB: [18](#), [22b](#)
 LDSBA: [18](#), [22b](#), [53](#)
 LDSH: [18](#), [22b](#), [53](#)
 LDSHA: [18](#), [22b](#), [53](#)
 LDSTUB: [18](#), [22b](#), [53](#)
 LDSTUBA: [18](#), [22b](#), [53](#)
 LDSW: [18](#), [22b](#), [53](#)
 LDSWA: [18](#), [22b](#), [53](#)
 LDUB: [18](#), [22b](#), [53](#)
 LDUBA: [18](#), [22b](#), [53](#)
 LDUH: [18](#), [22b](#), [53](#)
 LDUHA: [18](#), [22b](#), [53](#)
 LDUW: [18](#), [22b](#), [53](#)
 LDUWA: [18](#), [22b](#), [53](#)
 LDX: [18](#), [22b](#), [50b](#), [53](#)
 LDXA: [18](#), [22b](#), [53](#)
 LDXFSR: [19a](#), [53](#)
 loada: [22b](#), [23b](#)
 loadg: [22b](#), [23a](#)
 logical: [26c](#)
 mbz_f2: [3c](#), [14](#)
 mbz_f3: [4a](#), [22a](#)
 mbz_f4_13: [4b](#), [22a](#)
 mbz_f4_18: [4b](#), [22a](#)
 MEMBAR: [16](#), [25a](#), [53](#)
 membar_mask: [4a](#), [25a](#)
 mov: [47](#), [49a](#), [49b](#)
 MOVcc: [15](#), [28e](#), [28f](#)
 MOVr: [15](#), [17e](#)
 MOVr: [17e](#), [30a](#), [30b](#)
 movr: [47](#), [49b](#)
 MULSc: [15](#), [26c](#), [53](#)
 MULX: [15](#), [26c](#), [53](#)
 neg2: [47](#), [49b](#)
 neg: [47](#), [49a](#), [49b](#)
 nooperands: [31b](#)

NOP: [13e](#), [31b](#), [53](#)
 not2: [47](#), [49b](#)
 not: [47](#), [49a](#), [49b](#)
 op2: [3c](#), [13e](#)
 op3: [4a](#), [15](#), [18](#)
 op: [3b](#), [13d](#)
 opf: [4a](#), [20](#), [22a](#), [33b](#)
 opf_cc: [4b](#), [6b](#), [29a](#), [29b](#)
 opf_low5: [4b](#), [22a](#)
 opf_low6: [4b](#), [22a](#)
 OR: [15](#), [26c](#), [47](#), [53](#)
 ORcc: [15](#), [26c](#), [47](#), [53](#)
 ORN: [15](#), [26c](#), [53](#)
 ORNcc: [15](#), [26c](#), [53](#)
 p: [3c](#), [6c](#), [27d](#), [27e](#), [29c](#), [29d](#)
 PDIST: [35](#), [40](#)
 POPC: [17d](#), [26c](#), [53](#)
 POPCx: [15](#), [17d](#)
 PREFETCH: [19b](#), [24c](#), [24d](#), [53](#)
 PREFETCHA: [19b](#), [24c](#), [24d](#), [53](#)
 PREFETCHAx: [18](#), [19b](#)
 PREFETCHx: [18](#), [19b](#)
 RBRKPT_ADDR: [42](#)
 RBRKPT_MASK: [42](#)
 RBRKPTx: [42](#)
 rcond_f2: [3c](#), [7](#), [29c](#), [29d](#)
 rcond_f3: [4a](#), [7](#), [30a](#), [30b](#)
 rcond_f4: [4b](#), [7](#), [30a](#), [30b](#)
 rd: [3c](#), [5c](#), [11a](#), [13e](#), [16](#), [19a](#), [23a](#),
 [23b](#), [23c](#), [23d](#), [23e](#), [25a](#), [25b](#), [25c](#),
 [26a](#), [26b](#), [26c](#), [28e](#), [28f](#), [30a](#), [30b](#),
 [31c](#), [31d](#), [32b](#), [33a](#), [39a](#), [42](#), [44](#),
 [47](#)
 RDASI: [16](#), [25a](#), [53](#)
 RDASR: [16](#), [25b](#), [25c](#), [32b](#), [42](#), [53](#)
 rdasrmbz: [41c](#), [42](#)
 RDCCR: [16](#), [25a](#), [53](#)
 RDDCR: [32b](#), [40](#)
 RDFFAULT_ADDR: [42](#)
 rdfd: [12b](#), [12c](#), [24a](#), [29a](#), [29b](#),
 [30a](#), [30b](#), [30d](#), [31a](#), [39a](#), [46a](#)
 rdfdvdv: [12b](#)
 rdfhi: [11g](#), [12a](#)
 rdflo: [11g](#), [12a](#)
 RDFPRS: [16](#), [25a](#), [53](#)
 rdfq: [12b](#), [12c](#), [24a](#), [29a](#), [29b](#),
 [30a](#), [30b](#), [30d](#), [31a](#)
 rdfqi: [12b](#)
 rdfqv: [12b](#)
 rdfs: [11a](#), [12c](#), [24a](#), [29a](#), [29b](#),
 [30a](#), [30b](#), [30d](#), [31a](#), [39a](#), [46a](#)
 rdfsvdv: [11a](#)
 RDFTYPE: [42](#)
 RDGSR: [32b](#), [40](#)
 rdi: [5d](#), [25b](#), [25c](#)
 rdp: [5e](#), [6a](#), [26a](#), [26b](#)
 RDPC: [16](#), [25a](#), [53](#)
 RDPCR: [32b](#)
 RDPIC: [32b](#), [40](#)
 RD_PM_REG0: [42](#)
 RD_PM_REG1: [42](#)
 RD_PM_REG2: [42](#)
 RD_PM_REG3: [42](#)
 RD_PM_REG4: [42](#)
 RD_PM_REG5: [42](#)
 RD_PM_VN: [42](#)
 RD_PMx: [42](#)
 RDPR: [17c](#), [26a](#), [26b](#), [53](#)
 RDPRx: [15](#), [17c](#)
 RDSCR: [42](#)
 RDSOFTINT: [32b](#), [40](#)
 RDTICK: [16](#), [25a](#), [53](#)
 RDTICK_CMPR: [32b](#), [40](#)
 RDxxx: [15](#), [16](#)
 RDY: [16](#), [25a](#), [53](#)
 regaddr: [9b](#), [10a](#)
 reg_or_imm10: [8c](#), [30a](#), [30b](#)
 reg_or_imm11: [8c](#), [28e](#), [28f](#)
 reg_or_imm7: [8c](#), [9b](#)
 reg_or_imm: [8b](#), [9a](#), [25a](#), [25b](#),
 [25c](#), [26a](#), [26b](#), [26c](#), [33a](#), [44](#), [47](#)
 reg_or_shcnt32: [8c](#), [26c](#)
 reg_or_shcnt64: [8c](#), [26c](#)
 reg_plus_imm: [9b](#), [10a](#)
 regROI1: [8c](#), [8c](#)
 regROI7: [8c](#), [9b](#)
 regROI: [8b](#), [9a](#), [9b](#), [47](#)
 regROS3: [8c](#)
 regROS6: [8c](#)
 reloc: [27a](#), [27b](#), [27d](#), [28c](#), [29c](#),
 [47](#), [48a](#)
 restore_: [47](#), [49b](#)
 RESTORE: [15](#), [26c](#), [47](#), [53](#)
 RESTORED: [17a](#), [31b](#), [53](#)
 ret: [47](#), [49b](#)
 ret1: [47](#), [49b](#)
 RETRY: [17a](#), [31b](#), [53](#)
 RETURN: [15](#), [31c](#), [53](#)
 RGSR: [42](#)
 RHDW_MODE: [42](#)
 RIFTYPE: [42](#)
 rs1: [4a](#), [5c](#), [9a](#), [9b](#), [11a](#), [16](#), [17d](#),
 [23e](#), [25a](#), [25b](#), [25c](#), [26a](#), [26b](#), [26c](#),
 [29c](#), [29d](#), [30a](#), [30b](#), [32b](#), [33a](#), [39a](#),
 [42](#), [44](#), [47](#)
 rs2: [4a](#), [5c](#), [8b](#), [8c](#), [9a](#), [9b](#), [11a](#),
 [23e](#), [39a](#), [44](#), [47](#)
 rs3: [45a](#), [45c](#)
 RSCHED_INT: [42](#)
 RSCRATCH0: [42](#)
 RSCRATCH1: [42](#)
 RSCRATCH2: [42](#)
 RSCRATCH3: [42](#)
 RSCRATCHx: [42](#)
 rs1fd: [12b](#), [12c](#), [31a](#), [39a](#), [46a](#)
 rs2fd: [12b](#), [12c](#), [29a](#), [29b](#), [30a](#),
 [30b](#), [30d](#), [31a](#), [39a](#), [46a](#)
 rs3fd: [45c](#), [45d](#), [46a](#)
 rs1fdvdv: [12b](#)
 rs2fdvdv: [12b](#)
 rs3fdvdv: [45c](#)
 rs1fhi: [11g](#), [11h](#)
 rs2fhi: [11g](#), [11i](#)
 rs3fhi: [45b](#), [45c](#)
 rs1flo: [11g](#), [11h](#)
 rs2flo: [11g](#), [11i](#)
 rs3flo: [45b](#), [45c](#)
 rs1fq: [12b](#), [12c](#), [31a](#)
 rs2fq: [12b](#), [12c](#), [29a](#), [29b](#), [30a](#),
 [30b](#), [30d](#), [31a](#)
 rs1fqi: [12b](#)
 rs2fqi: [12b](#)
 rs1fqv: [12b](#)
 rs2fqv: [12b](#)
 rs1fs: [11a](#), [12c](#), [31a](#), [39a](#), [46a](#)
 rs2fs: [11a](#), [12c](#), [29a](#), [29b](#), [30a](#),
 [30b](#), [30d](#), [31a](#), [39a](#), [46a](#)
 rs3fs: [45c](#), [45d](#), [46a](#)
 rs1fsv: [11a](#)
 rs2fsv: [11a](#)
 rs3fsv: [45c](#)
 rsl: [5d](#), [25b](#), [25c](#)
 rslp: [5e](#), [6a](#), [26a](#), [26b](#)
 RTICK_MATCH: [42](#)
 save_: [47](#), [49b](#)
 SAVE: [15](#), [26c](#), [47](#), [53](#)
 SAVED: [17a](#), [31b](#), [53](#)
 SAVEDx: [15](#), [17a](#)
 SDIV: [15](#), [26c](#), [53](#)
 SDIVcc: [15](#), [26c](#), [53](#)
 SDIVX: [15](#), [26c](#), [53](#)
 SETHI: [13e](#), [31d](#), [53](#)
 SET_SCHED_INT: [44](#)
 shcnt32: [4a](#), [8c](#)
 shcnt64: [4a](#), [8c](#)
 shift32: [26c](#)
 shift64: [26c](#)
 SHUTDOWN: [33b](#), [40](#)
 signx2: [47](#), [49b](#)
 signx: [47](#), [49a](#), [49b](#)
 simm10: [4a](#), [8c](#)
 simm11: [4b](#), [8c](#)
 simm13: [4a](#), [8b](#), [9a](#), [9b](#), [25a](#)
 simm7: [4b](#), [8c](#), [9b](#)
 SIR: [16](#), [25a](#), [53](#)

size: [45a](#), [46a](#)
 SLL: [17b](#), [26c](#), [53](#)
 SLLX: [17b](#), [26c](#), [53](#)
 SLLx: [15](#), [17b](#)
 SLLX: [17b](#), [26c](#), [53](#)
 SMUL: [15](#), [26c](#), [53](#)
 SMULcc: [15](#), [26c](#), [53](#)
 software_trap_number: [9b](#),
 [28a](#), [28b](#)
 SRA: [17b](#), [26c](#), [47](#), [53](#)
 SRAX: [17b](#), [26c](#), [53](#)
 SRAx: [15](#), [17b](#)
 SRAX: [17b](#), [26c](#), [53](#)
 SRL: [17b](#), [26c](#), [47](#), [53](#)
 SRLx: [15](#), [17b](#)
 SRLX: [17b](#), [26c](#), [53](#)
 STB: [18](#), [22b](#), [47](#), [53](#)
 STBA: [18](#), [22b](#), [53](#)
 STBAR: [16](#), [25a](#), [53](#)
 STD: [18](#), [23c](#), [23d](#), [53](#)
 STDA: [18](#), [23c](#), [23d](#), [53](#)
 STDF: [18](#), [24a](#), [53](#)
 STDFA: [18](#), [24a](#), [53](#)
 STF: [18](#), [24a](#), [53](#)
 STFA: [18](#), [24a](#), [53](#)
 STFSR: [19a](#), [53](#)
 stfsr: [19a](#), [24b](#)
 STFSR: [19a](#), [53](#)
 STFSRx: [18](#), [19a](#)
 STH: [18](#), [22b](#), [47](#), [53](#)
 STHA: [18](#), [22b](#), [53](#)
 storea: [22b](#), [23b](#)
 storeg: [22b](#), [23a](#)
 STQF: [18](#), [24a](#), [53](#)
 STQFA: [18](#), [24a](#), [53](#)
 STW: [18](#), [22b](#), [47](#), [53](#)
 STWA: [18](#), [22b](#), [53](#)
 STX: [18](#), [22b](#), [47](#), [50b](#), [53](#)
 STXA: [18](#), [22b](#), [53](#)
 STXFSR: [19a](#), [53](#)
 SUB: [15](#), [26c](#), [47](#), [53](#)
 SUBC: [15](#), [26c](#), [53](#)
 SUBcc: [15](#), [26c](#), [47](#), [53](#)
 SUBCcc: [15](#), [26c](#)
 SWAP: [18](#), [22b](#), [53](#)
 SWAPA: [18](#), [22b](#), [53](#)
 TABLE_31: [13d](#), [13e](#)
 TABLE_32: [13d](#), [15](#)
 TABLE_33: [13d](#), [18](#)
 TADDcc: [15](#), [26c](#), [53](#)
 TADDccTV: [15](#), [26c](#), [53](#)
 Tcc: [15](#), [28a](#), [28b](#)
 tst: [47](#), [49b](#)
 TSUBcc: [15](#), [26c](#), [53](#)
 TSUBccTV: [15](#), [26c](#), [53](#)
 UDIV: [15](#), [26c](#), [53](#)
 UDIVcc: [15](#), [26c](#), [53](#)
 UDIVX: [15](#), [26c](#), [53](#)
 UMUL: [15](#), [26c](#), [53](#)
 UMULcc: [15](#), [26c](#), [53](#)
 var: [45a](#), [46a](#)
 vismbz: [33c](#), [34a](#)
 VISOP0: [34a](#), [34b](#)
 visop1: [33c](#), [34a](#)
 VISOP1: [34a](#), [35](#)
 visop2: [33c](#), [34b](#), [35](#), [36](#), [37](#)
 VISOP2: [34a](#), [36](#)
 VISOP3: [34a](#), [37](#)
 visop3d: [37](#), [38](#)
 visopd: [38](#), [39a](#)
 visop3dd: [37](#), [38](#)
 visopdd: [38](#), [39a](#)
 visoplddd: [35](#), [38](#)
 visop2ddd: [36](#), [38](#)
 visop3ddd: [37](#), [38](#)
 visopddd: [38](#), [39a](#)
 visoplddd: [35](#), [38](#)
 visopddr: [38](#), [39a](#)
 visoplds: [35](#), [38](#)
 visopds: [38](#), [39a](#)
 visop3dxd: [37](#), [38](#)
 visopdxd: [38](#), [39a](#)
 visop0rrr: [34b](#), [38](#)
 visopr: [38](#), [39a](#)
 visop3s: [37](#), [38](#)
 visops: [38](#), [39a](#)
 visop2sd: [36](#), [38](#)
 visopsd: [38](#), [39a](#)
 visoplstd: [35](#), [38](#)
 visopstd: [38](#), [39a](#)
 visop3ss: [37](#), [38](#)
 visopss: [38](#), [39a](#)
 visoplssd: [35](#), [38](#)
 visop2ssd: [36](#), [38](#)
 visopssd: [38](#), [39a](#)
 visop2sss: [36](#), [38](#)
 visop3sss: [37](#), [38](#)
 visopsss: [38](#), [39a](#)
 visop3sxs: [37](#), [38](#)
 visopsxs: [38](#), [39a](#)
 WRASI: [16](#), [25a](#), [53](#)
 WRASR: [16](#), [25b](#), [25c](#), [33a](#), [44](#), [53](#)
 wrasrmbz: [41c](#), [44](#)
 WR_BRK_ADDR: [44](#)
 WR_BRK_MASK: [44](#)
 WR_BRKx: [44](#)
 WRCCR: [16](#), [25a](#)
 WRCLEAR_SOFTINT: [33a](#), [40](#)
 WRDCR: [33a](#), [40](#)
 WRFPRS: [16](#), [25a](#), [53](#)
 WRGSR: [33a](#), [40](#), [44](#)
 WR_HDW_MODE: [44](#)
 WRPCR: [33a](#), [40](#)
 WRPIC: [33a](#), [40](#)
 WR_PM_CLR_DIS: [44](#)
 WR_PM_CLR_EN: [44](#)
 WR_PM_DIS: [44](#)
 WR_PM_EN: [44](#)
 WR_PM_VN: [44](#)
 WR_PMx: [44](#)
 WRPR: [17c](#), [26a](#), [26b](#), [53](#)
 WRPRx: [15](#), [17c](#)
 WR_SCHED_INT: [44](#)
 WRSCR: [44](#)
 WR_SCRATCH0: [44](#)
 WR_SCRATCH1: [44](#)
 WR_SCRATCH2: [44](#)
 WR_SCRATCH3: [44](#)
 WR_SCRATCHx: [44](#)
 WRSET_SOFTINT: [33a](#)
 WR_SOFTINT: [33a](#), [40](#)
 WRTICK_CMPR: [33a](#), [40](#)
 WR_TICK_MATCH: [44](#)
 WRxxx: [15](#), [16](#)
 WRY: [16](#), [25a](#), [53](#)
 x: [4a](#), [17b](#)
 XNOR: [15](#), [26c](#), [47](#), [53](#)
 XNORcc: [15](#), [26c](#), [53](#)
 XOR: [15](#), [26c](#), [47](#), [53](#)
 XORcc: [15](#), [26c](#), [53](#)