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-00-01 Jens Gustedt, Ole A. Maehle, and Jan Arne Telle.  Java programs do not have bounded treewidth. February 2000.

SPARC-V9 Instruction-Set Syntax Specification

Bill Clarke
llib@computer.org
CAP Project
Australian National University
November 22, 2000

Contents

1 Introduction
2 Fields
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
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
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 njmctk-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 an 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.

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,

```
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,

```
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,

\[ \langle \text{field specifications 3b} \rangle + \equiv \]

\begin{align*}
op & 19:24 \text{ rs1 14:18 } i 13:13 \text{ imm}_\text{asi} 5:12 \text{ rs2 0:4} \\
\text{elmet} & 0:12 \text{ rcond}_3 10:12 \text{ simm} 10 0:9 \text{ member_mask 0:6} \\
\text{impldep} & 1 25:29 \text{ impldep} 2 0:18 x 12:12 \text{ shcnt} 32 0:4 \\
\text{shcnt} & 0:5 \text{ opf} 5:13 \text{ mbz}_3 27:29 \text{ fcc}_3 25:26 \text{ fcn} 25:29 \\
\end{align*}

Defines:

- \text{fcc}_3, used in chunks 6b and 31a.
- \text{fcn}, used in chunks 17a and 24.
- \text{i}, used in chunks 8, 10b, 16, and 44.
- \text{imm}_\text{asi}, used in chunk 10.
- \text{impldep}, never used.
- \text{impldep} 2, never used.
- \text{mbz}_3, used in chunk 22a.
- \text{member_mask}, used in chunk 25a.
- \text{opf}, used in chunks 20, 22a, and 33b.
- \text{rcond}, used in chunks 7 and 30.
- \text{rs1}, used in chunks 5c, 9, 11a, 16, 17d, 23e, 25, 26, 29, 30, 32b, 33a, 39a, 42, 44, and 47.
- \text{rs2}, used in chunks 5c, 8, 9, 11a, 23e, 39a, 44, and 47.
- \text{shcnt} 32, used in chunk 8c.
- \text{shcnt} 64, used in chunk 8c.
- \text{simm} 10, used in chunk 8c.
- \text{simm} 13, used in chunks 8, 9, and 25a.
- \text{x}, used in chunk 17b.

and format 4

\[ \langle \text{field specifications 3b} \rangle + \equiv \]

\begin{align*}
icc & 11:12 \text{ fcc}_4 11:12 \text{ simm} 11 0:10 \text{ cc2}_4 18:18 \text{ mbz}_4 18:18 \\
icond & 14:17 \text{ fcond}_4 14:17 \\
simm & 0:6 \text{ mbz}_4 13 13:13 \text{ rcond}_4 10:12 \text{ opf}_\text{low} 5:9 \text{ opf}_\text{cc} 11:13 \\
opf & 5:10 \\
\end{align*}

Defines:

- \text{cc2}_4, used in chunk 28.
- \text{fccc}_4, used in chunks 6b and 28e.
- \text{fcond}_4, used in chunks 6d, 28e, and 29a.
- \text{icc}_4, used in chunks 6b and 28.
- \text{icond}_4, used in chunks 6d, 28, and 29.
- \text{mbz}_4 13, used in chunk 22a.
- \text{mbz}_4 18, used in chunk 22a.
- \text{opf}_\text{cc}, used in chunks 6b and 29.
- \text{opf}_\text{low} 5, used in chunk 22a.
- \text{opf}_\text{low} 6, used in chunk 22a.
- \text{rcond}_4, used in chunks 7 and 30.
- \text{simm} 10, used in chunk 8c.
- \text{simm} 7, used in chunks 8c and 9b.

Unfortunately, some of the new (V9) instructions have similarly named bitfields but are located in different places (\text{cond}, \text{cc}, \text{rcond} and \text{opf}_\text{low}): these are labelled here instead by suffixing \_2, \_3 or \_4 depending on whether the field is defined in Format 2, 3 or 4. This convention breaks down for \text{opf}_\text{low} since they are both in format 4, so they are labelled \text{opf}_\text{low} 5 and \text{opf}_\text{low} 6 (ick!).

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

Like the \text{cc} fields, the \text{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 \text{mbz}_f{2, 3, 4, \{18, 13\}} fields which Must-Be-Zero for some instructions (the \text{mbz}_f{2, 3, 4, \{18, 13\}} 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 \texttt{reg} or \texttt{imm} and floating-point registers), or explicitly in the generating constructors (such as the \texttt{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 \texttt{fieldinfo} specifications in a separate file, so that we can concatenate other field specifications with the above ones.

\begin{verbatim}
(fields-names.spec 5a)≡
  # fields-names.spec
  # requires fields.spec
  (fieldinfo specifications 5c)
\end{verbatim}

2.2.1 Integer registers

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

\begin{verbatim}
(properties of integer-register fields 5b)≡
  names [ "%g0" "%g1" "%g2" "%g3" "%g4" "%g5" "%g6" "%g7"
  "%o0" "%o1" "%o2" "%o3" "%o4" "%o5" "%sp" "%07"
  "%10" "%11" "%12" "%13" "%14" "%15" "%16" "%17"
  "%i0" "%i1" "%i2" "%i3" "%i4" "%fp" "%17"
  ]
\end{verbatim}

\begin{verbatim}
(fieldinfo specifications 5c)≡
  fieldinfo
  [ rd rs1 rs2 ] is [ (properties of integer-register fields 5b) ]
\end{verbatim}

Uses \texttt{rd}, \texttt{rs1}, \texttt{rs2}.

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

\begin{verbatim}
(field specifications 3b)+≡
  rs1i 14:18 rdi 25:29
\end{verbatim}

Defines:
\texttt{rdi}, used in chunk 25.
\texttt{rs1i}, 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 \texttt{p} to the register identifiers. The \texttt{pX} in the specification are illegal values (which are enforced in the pattern or constructor).

\begin{verbatim}
(field specifications 3b)+≡
  rs1p 14:18 rdp 25:29
\end{verbatim}

Defines:
\texttt{rdp}, used in chunks 6a and 26.
\texttt{rs1p}, used in chunks 6a and 26.
2 FIELDS

6a \{fieldinfo specifications sc\}+≡

fieldinfo

   [ rs1p rdp ] is [ names [
      "$\text{tpc}"   "$\text{tnpc}"  "$\text{tstate}"  "$\text{tt}"
      "$\text{tick}"  "$\text{tba}"   "$\text{pstate}"  "$\text{tl}"
      "$\text{cwp}"   "$\text{cansave}" "$\text{canrestore}"
      "$\text{cleanwin}" "$\text{otherwin}" "$\text{wstate}"  "$\text{fq}"
      "$\text{p16}!"   "$\text{p17}!"  "$\text{p18}!"  "$\text{p19}!"
      "$\text{p20}!"   "$\text{p21}!"  "$\text{p22}!"  "$\text{p23}!"
      "$\text{p24}!"   "$\text{p25}!"  "$\text{p26}!"  "$\text{p27}!"
      "$\text{p28}!"   "$\text{p29}!"  "$\text{p30}!"  "$\text{ver}" ] ]

Uses rdp 5e and rs1p 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 sc\}+≡

fieldinfo

   [ icc_f2 icc_f4 ] is [ names [ "$\text{icc}" "$\text{icc}!" "$\text{xcc}" "$\text{xcc}!" ] ]
   [ fcc_f2 fcc_f3 fcc_f4 ]
     is [ names [ "$\text{fcc0}" "$\text{fcc1}" "$\text{fcc2}" "$\text{fcc3}" ] ]
   opf_cc
     is [ names [ "$\text{fcc0}" "$\text{fcc1}" "$\text{fcc2}" "$\text{fcc3}" "$\text{icc}" "$\text{icc}!" "$\text{xcc}" "$\text{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 sc\}+≡

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 sc\}+≡

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 U A E UE GE UGE LE ULE O ] ]

Uses fcond_f2 3c, fcond_f4 4b, icond_f2 3c, and icond_f4 4b.
The \texttt{rcond}\_f(2, 3, 4) fields specify the integer register value condition to check, and like the \texttt{(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 !.

\begin{verbatim}
7 \langle\text{fieldinfo specifications sc}\rangle \equiv (5a) \land 6d
  fieldinfo
    \[ \text{rcond\_f2 rcond\_f3 rcond\_f4 } \] is
    \[ \text{names } \left[ "0!" Z LEZ LZ "4!" NZ GZ GEZ } \right] \]
Uses \texttt{rcond\_f2 3c, rcond\_f3 4a, and rcond\_f4 4b.}
\end{verbatim}
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.

\[\langle \text{types.spec} \rangle \equiv\]

\[
\begin{align*}
\text{# types.spec} \\
\text{# requires fields.spec} \\
\langle \text{type specifications} \rangle &
\end{align*}
\]

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:

\[\langle \text{type specifications} \rangle \equiv\]

\[
\begin{align*}
\text{constructors} \\
\text{immROI simm13!} & : \text{reg_or_imm} \quad i = 1 & \text{simm13} \\
\text{regROI rs2} & : \text{reg_or_imm} \quad i = 0 & \text{rs2}
\end{align*}
\]

Defines:

\text{immROI}, used in chunks 9 and 47.
\text{regROI}, used in chunks 9 and 47.
\text{regROI}, used in chunks 9 and 47.
\text{regROI}, used in chunks 9 and 47.
\text{regROI}, used in chunks 9 and 47.
\text{regROI}, used in chunks 9 and 47.
\text{regROI}, used in chunks 9 and 47.
\text{regROI}, used in chunks 9 and 47.

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

\[\langle \text{type specifications} \rangle \equiv\]

\[
\begin{align*}
\text{constructors} \\
\text{immROI10 simm10!} & : \text{reg_or_imm10} \quad i = 1 & \text{simm10} \\
\text{regROI10 rs2} & : \text{reg_or_imm10} \quad i = 0 & \text{rs2}
\end{align*}
\]

\[
\begin{align*}
\text{immROI11 simm11!} & : \text{reg_or_imm11} \quad i = 1 & \text{simm11} \\
\text{regROI11 rs2} & : \text{reg_or_imm11} \quad i = 0 & \text{rs2}
\end{align*}
\]

\[
\begin{align*}
\text{immROS32 shcnt32} & : \text{reg_or_shcnt32} \quad i = 1 & \text{shcnt32} \\
\text{regROS32 rs2} & : \text{reg_or_shcnt32} \quad i = 0 & \text{rs2}
\end{align*}
\]

\[
\begin{align*}
\text{immROS64 shcnt64} & : \text{reg_or_shcnt64} \quad i = 1 & \text{shcnt64} \\
\text{regROS64 rs2} & : \text{reg_or_shcnt64} \quad i = 0 & \text{rs2}
\end{align*}
\]

\[
\begin{align*}
\text{immROI7 simm7!} & : \text{reg_or_imm7} \quad i = 1 & \text{simm7} \\
\text{regROI7 rs2} & : \text{reg_or_imm7} \quad i = 0 & \text{rs2}
\end{align*}
\]

Defines:

\text{immROI10}, never used.
\text{immROI11}, used in chunk 9b.
\text{immROS32}, never used.
\text{immROS64}, never used.
\text{regROI10}, used in chunk 30.
\text{regROI11}, used in chunk 28.
\text{regROI7}, used in chunk 9b.
\text{regROS32}, used in chunk 26c.
\text{regROS64}, used in chunk 26c.
\text{regROI1}, never used.
\text{regROI7}, used in chunk 9b.
\text{regROS3}, never used.
\text{regROS6}, never used.

Uses i 4a, rs2 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:

\[
\begin{align*}
\text{ctor} & : \text{address} \equiv \\
\text{ctor} & : \text{address} \equiv \text{ctor}(\text{reg}, \text{imm}) \\
\text{ctor} & : \text{address} \equiv \text{ctor}(\text{imm}, \text{reg})
\end{align*}
\]

Defines:
- `ctor`, used in chunks 23, 24, 31c, and 47.
- `ctor`, used in chunks 23, 24, 31c, and 47.
- `ctor`, never used.
- `ctor`, never used.
- `ctor`, never used.

Uses `immROI`, `reg_or_imm`, `regROI`, `rs1`, `rs2`, and `simm`.

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).

\[
\begin{align*}
\text{ctor} & : \text{address} \equiv \\
\text{ctor} & : \text{address} \equiv \text{ctor}(\text{reg}, \text{imm}) \\
\text{ctor} & : \text{address} \equiv \text{ctor}(\text{imm}, \text{reg})
\end{align*}
\]

Defines:
- `ctor`, used in chunks 23, 24, 31c, and 47.
- `ctor`, used in chunks 23, 24, 31c, and 47.
- `ctor`, never used.
- `ctor`, never used.
- `ctor`, never used.
- `ctor`, never used.

Uses `immROI`, `reg_or_imm`, `regROI`, `rs1`, `rs2`, and `simm`.

Uses `immROI`, `reg_ROI`, `regROI`, `rs1`, `rs2`, `simm`, `simmROI`, and `simmROI7`.

---

9a, 9b

\[
\begin{align*}
\text{ctor} & : \text{address} \equiv \\
\text{ctor} & : \text{address} \equiv \text{ctor}(\text{reg}, \text{imm}) \\
\text{ctor} & : \text{address} \equiv \text{ctor}(\text{imm}, \text{reg})
\end{align*}
\]

Defines:
- `ctor`, used in chunks 23, 24, 31c, and 47.
- `ctor`, used in chunks 23, 24, 31c, and 47.
- `ctor`, never used.
- `ctor`, never used.
- `ctor`, never used.
- `ctor`, never used.

Uses `immROI`, `reg_or_imm`, `regROI`, `rs1`, `rs2`, `simm`, `simmROI`, and `simmROI7`.

Uses `immROI`, `reg_ROI`, `regROI`, `rs1`, `rs2`, `simm`, `simmROI`, and `simmROI7`.

---

9b

\[
\begin{align*}
\text{ctor} & : \text{address} \equiv \\
\text{ctor} & : \text{address} \equiv \text{ctor}(\text{reg}, \text{imm}) \\
\text{ctor} & : \text{address} \equiv \text{ctor}(\text{imm}, \text{reg})
\end{align*}
\]

Defines:
- `ctor`, used in chunks 23, 24, 31c, and 47.
- `ctor`, used in chunks 23, 24, 31c, and 47.
- `ctor`, never used.
- `ctor`, never used.
- `ctor`, never used.
- `ctor`, never used.

Uses `immROI`, `reg_or_imm`, `regROI`, `rs1`, `rs2`, `simm`, `simmROI`, and `simmROI7`.

Uses `immROI`, `reg_ROI`, `regROI`, `rs1`, `rs2`, `simm`, `simmROI`, and `simmROI7`.

---

9b

\[
\begin{align*}
\text{ctor} & : \text{address} \equiv \\
\text{ctor} & : \text{address} \equiv \text{ctor}(\text{reg}, \text{imm}) \\
\text{ctor} & : \text{address} \equiv \text{ctor}(\text{imm}, \text{reg})
\end{align*}
\]

Defines:
- `ctor`, used in chunks 23, 24, 31c, and 47.
- `ctor`, used in chunks 23, 24, 31c, and 47.
- `ctor`, never used.
- `ctor`, never used.
- `ctor`, never used.
- `ctor`, never used.

Uses `immROI`, `reg_or_imm`, `regROI`, `rs1`, `rs2`, `simm`, `simmROI`, and `simmROI7`.

Uses `immROI`, `reg_ROI`, `regROI`, `rs1`, `rs2`, `simm`, `simmROI`, and `simmROI7`.

---

9b

\[
\begin{align*}
\text{ctor} & : \text{address} \equiv \\
\text{ctor} & : \text{address} \equiv \text{ctor}(\text{reg}, \text{imm}) \\
\text{ctor} & : \text{address} \equiv \text{ctor}(\text{imm}, \text{reg})
\end{align*}
\]

Defines:
- `ctor`, used in chunks 23, 24, 31c, and 47.
- `ctor`, used in chunks 23, 24, 31c, and 47.
- `ctor`, never used.
- `ctor`, never used.
- `ctor`, never used.
- `ctor`, never used.

Uses `immROI`, `reg_or_imm`, `regROI`, `rs1`, `rs2`, `simm`, `simmROI`, and `simmROI7`.

Uses `immROI`, `reg_ROI`, `regROI`, `rs1`, `rs2`, `simm`, `simmROI`, and `simmROI7`.

---

9b

\[
\begin{align*}
\text{ctor} & : \text{address} \equiv \\
\text{ctor} & : \text{address} \equiv \text{ctor}(\text{reg}, \text{imm}) \\
\text{ctor} & : \text{address} \equiv \text{ctor}(\text{imm}, \text{reg})
\end{align*}
\]

Defines:
- `ctor`, used in chunks 23, 24, 31c, and 47.
- `ctor`, used in chunks 23, 24, 31c, and 47.
- `ctor`, never used.
- `ctor`, never used.
- `ctor`, never used.
- `ctor`, never used.

Uses `immROI`, `reg_or_imm`, `regROI`, `rs1`, `rs2`, `simm`, `simmROI`, and `simmROI7`.

Uses `immROI`, `reg_ROI`, `regROI`, `rs1`, `rs2`, `simm`, `simmROI`, and `simmROI7`.

---

9b

\[
\begin{align*}
\text{ctor} & : \text{address} \equiv \\
\text{ctor} & : \text{address} \equiv \text{ctor}(\text{reg}, \text{imm}) \\
\text{ctor} & : \text{address} \equiv \text{ctor}(\text{imm}, \text{reg})
\end{align*}
\]

Defines:
- `ctor`, used in chunks 23, 24, 31c, and 47.
- `ctor`, used in chunks 23, 24, 31c, and 47.
- `ctor`, never used.
- `ctor`, never used.
- `ctor`, never used.
- `ctor`, never used.

Uses `immROI`, `reg_or_imm`, `regROI`, `rs1`, `rs2`, `simm`, `simmROI`, and `simmROI7`.

Uses `immROI`, `reg_ROI`, `regROI`, `rs1`, `rs2`, `simm`, `simmROI`, and `simmROI7`.

---

9b

\[
\begin{align*}
\text{ctor} & : \text{address} \equiv \\
\text{ctor} & : \text{address} \equiv \text{ctor}(\text{reg}, \text{imm}) \\
\text{ctor} & : \text{address} \equiv \text{ctor}(\text{imm}, \text{reg})
\end{align*}
\]

Defines:
- `ctor`, used in chunks 23, 24, 31c, and 47.
- `ctor`, used in chunks 23, 24, 31c, and 47.
- `ctor`, never used.
- `ctor`, never used.
- `ctor`, never used.
- `ctor`, never used.

Uses `immROI`, `reg_or_imm`, `regROI`, `rs1`, `rs2`, `simm`, `simmROI`, and `simmROI7`.

Uses `immROI`, `reg_ROI`, `regROI`, `rs1`, `rs2`, `simm`, `simmROI`, and `simmROI7`.
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:

```
(type specifications 8b) +≡

constructors
  # imm_asi_address [regaddr] imm_asi : as_address
  imm_asi_address [regaddr] imm_asi : as_address is regaddr & imm_asi
  # imp_asi_address [reg_plus_imm] "%asi" : as_address
  imp_asi_address [reg_plus_imm] "%asi" : as_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.

```
(type specifications 8b) +≡

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:

<table>
<thead>
<tr>
<th>Type</th>
<th>6-bit actual register (5 to 0)</th>
<th>5-bit encoding (4 to 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>single</td>
<td>[0 b4 b3 b2 b1 b0]</td>
<td>[b4 b3 b2 b1 b0]</td>
</tr>
<tr>
<td>double</td>
<td>[b5 b4 b3 b2 b1 0]</td>
<td>[b4 b3 b2 b1 b5]</td>
</tr>
<tr>
<td>quad</td>
<td>[b5 b4 b3 b2 0 0]</td>
<td>[b4 b3 b2 0 b5]</td>
</tr>
</tbody>
</table>

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 register.

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

```
(types-fpre.spec 10c) ≡

# types-fpre.spec
# after types.spec (requires fields.spec)

(fpre type specifications 1la)
```

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

```
(fseqn 10d) ≡

( 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.
**Structured Operands and Typed Constructors**

11a  \[\text{(fpre type specifications 11a)}\]\

**constructors**
- rs1fsv \("%f"fr! : rs1fs (fseqnv 10d) is rs1 = fr
- rs2fsv \("%f"fr! : rs2fs (fseqnv 10d) is rs2 = fr
- rdfs v \("%f"fr! : rdfs (fseqnv 10d) is rd = fr

**Defines:**
- rdfs, used in chunks 24a, 29–31, 39a, and 46a.
- rs1fsv, never used.
- rs2fsv, used in chunks 31a, 39a, and 46a.
- rs1fsv, never used.
- rs2fsv, never used.
- Uses rd 3c, rs1 4a, and 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 (frdlegal); fhi and flo are temporary variables which hold the encoded parts of the register (frdqtemp).

11b  \[\text{(frdqlegal 11b)}\]\
\(fr@[6:31] = 0, fr@[0] = 0\)

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

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

11d  \[\text{(fdeqnv 11d)}\]\
\(\{ \text{(frdqlegal 11b), (frdqtemp 11c) }\)

11e  \[\text{(fqeqnv 11e)}\]\
\(\{ \text{(frdqlegal 11b), (frdqtemp 11c), fr@[1] = 0 }\)

11f  \[\text{(fqeqni 11f)}\]\
\(\{ \text{(frdqlegal 11b), (frdqtemp 11c), fr@[1] = 1 }\)

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

11g  \[\text{(types-fpre-fields.spec 11g)}\]\
\# types-fpre-fields.spec
\# this goes after fields.spec (in fields declaration)
- rs1fhi 15:18 rs1flo 14:14
- rs2fhi 1:4 rs2flo 0:0
- rdfhi 26:29 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 \(f\{hi, lo\}\) to the actual fields \(r\{s\{1,2\},d\}f\{hi, lo\}\).

11h  \[\text{(rs1fass 11h)}\]\
\(rs1fhi = fhi & rs1flo = flo\)

**Uses** rs1fhi 11g and rs1flo 11g.

11i  \[\text{(rs2fass 11i)}\]\
\(rs2fhi = fhi & rs2flo = flo\)

**Uses** rs2fhi 11g and rs2flo 11g.
All that remains is to specify the constructors for double and quad registers.

\[
\text{\texttt{rdfhi}} = \text{fhi} \& \text{rdflo} = \text{flo}
\]

Uses \texttt{rdfhi} and \texttt{rdflo}.

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.

\[
\text{\texttt{types-nofpre-fields.spec}}
\]

\[
\begin{align*}
\text{rs1fdv} & \, \text{"f"fr!} : \text{rs1fd} & \text{\texttt{fdeqnv}} \, \text{(rs1fass 11h)} \\
\text{rs2fdv} & \, \text{"f"fr!} : \text{rs2fd} & \text{\texttt{fdeqnv}} \, \text{(rs2fass 11l)} \\
\text{rdfdv} & \, \text{"f"fr!} : \text{rdfd} & \text{\texttt{fdeqnv}} \, \text{(rdfass 12a)} \\
\text{rs1fqv} & \, \text{"f"fr!} : \text{rs1fq} & \text{\texttt{fqeqnv}} \, \text{(rs1fass 11h)} \\
\text{rs2fqv} & \, \text{"f"fr!} : \text{rs2fq} & \text{\texttt{fqeqnv}} \, \text{(rs2fass 11i)} \\
\text{rdfqv} & \, \text{"f"fr!} : \text{rdfq} & \text{\texttt{fqeqnv}} \, \text{(rdfass 12a)} \\
\text{rs1fqi} & \, \text{"f"fr!} : \text{rs1fq} & \text{\texttt{fqeqni}} \, \text{(rs1fass 11h)} \\
\text{rs2fqi} & \, \text{"f"fr!} : \text{rs2fq} & \text{\texttt{fqeqni}} \, \text{(rs2fass 11i)} \\
\text{rdfqi} & \, \text{"f"fr!} : \text{rdfq} & \text{\texttt{fqeqni}} \, \text{(rdfass 12a)}
\end{align*}
\]

Defines:

\texttt{rdfd}, used in chunks 24a, 29–31, 39a, and 46a.
\texttt{rdfdv}, never used.
\texttt{rdfq}, used in chunks 24a and 29–31.
\texttt{rdfq}, never used.
\texttt{rdfqv}, never used.
\texttt{rs1fd}, used in chunks 31a, 39a, and 46a.
\texttt{rs2fd}, used in chunks 29–31, 39a, and 46a.
\texttt{rs1fdv}, never used.
\texttt{rs2fdv}, never used.
\texttt{rs1fq}, used in chunk 31a.
\texttt{rs2fq}, used in chunks 29–31.
\texttt{rs1fqi}, never used.
\texttt{rs2fqi}, never used.
\texttt{rs1fsv}, never used.
\texttt{rs2fsv}, never used.

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

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.

**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.
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:

\[
\begin{align*}
BPr & \equiv BPrx \land \text{mbz}_f = 0 \\
\end{align*}
\]

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 field-info/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).

\[
\{\text{pattern and constructor specifications }13d\} + \Xi
\]

\[
\begin{array}{cccc}
\text{ADD} & \text{ADDcc} & \text{TADDcc} & \text{WRxxx} \\
\text{AND} & \text{ANDcc} & \text{TSUBcc} & \text{SAVEDx} \\
\text{OR} & \text{ORcc} & \text{TADDccTV} & \text{WRPRx} \\
\text{XOR} & \text{XORcc} & \text{TSUBccTV} & \\
\text{SUB} & \text{SUBcc} & \text{MULScc} & \text{FPop1} \\
\text{ANDN} & \text{ANDNcc} & \text{SLLx} & \text{FPop2} \\
\text{ORN} & \text{ORNcc} & \text{SRLx} & \text{IMPDEP1} \\
\text{XNOR} & \text{XNORcc} & \text{SRAx} & \text{IMPDEP2} \\
\text{ADDC} & \text{ADDCcc} & \text{RDxxx} & \text{JMPL} \\
\text{MULX} & \_ & \_ & \text{RETURN} \\
\text{UMUL} & \text{UMULcc} & \text{RDPRx} & \text{Tcc} \\
\text{SMUL} & \text{SMULcc} & \text{FLUSHW} & \text{FLUSH} \\
\text{SUBC} & \text{SUBCc} & \text{MOVcc} & \text{SAVE} \\
\text{UDIVX} & \_ & \text{SDIVX} & \text{RESTORE} \\
\text{UDIV} & \text{UDIVcc} & \text{POPCx} & \text{DONEx} \\
\text{SDIV} & \text{SDIVcc} & \text{MOVr} & \_ \\
\end{array}
\]

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 33b and 34a.

MOVcc, used in chunk 28.

MOVr, used in chunk 17c.

MULScc, 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.
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.

\[
\text{patterns}
\]

\[
\begin{align*}
&[ \text{WRY } \_ \text{ WRCCR WRASI } \_ \_ \text{ WRFPRS } ] \\
& \quad \text{is WRxxx & rd = \{0 to 6\}} \\
& \text{WRASR is WRxxx} \\
& \text{SIR is WRxxx & rd = 15 & rs1 = 0 & i = 1} \\
& [ \text{RDY } \_ \text{ RDCCR RDAISI RDTICK RDFPRS } ] \\
& \quad \text{is RDxxx & rs1 = \{0 to 6\} & i = 0} \\
& \text{RDASR is RDxxx & i = 0} \\
& [ \text{STBAR MEMBAR } ] \\
& \quad \text{is RDxxx & rs1 = 15 & rd = 0 & i = [0 1]} \\
\end{align*}
\]

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.
SAVEDx represents the SAVED and RESTORED instructions, which depend on fcn. Similarly, DONEx represents the DONE and REPLY 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).

\[ \langle \text{pattern and constructor specifications} \rangle^+ \]
\[
\begin{align*}
\text{17a} & \quad \text{SAVEDx} & \text{is SAVEDx} & \& \text{fcn} = [0 1] \\
\text{17b} & \quad \text{DONEx} & \text{is DONEx} & \& \text{fcn} = [0 1]
\end{align*}
\]

Defines:
- SAVED, 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 DONEx 15, fcn 4a, and SAVEDx 15.

SLLx, SRLx and SRAx are shifts on double or single words depending on x.

\[ \langle \text{pattern and constructor specifications} \rangle^+ \]
\[
\begin{align*}
\text{17b} & \quad \text{[ SLL SLLx ] is SLLx} & \& \text{x} = [0 1] \\
\text{17c} & \quad \text{[ SRL SRLx ] is SRLx} & \& \text{x} = [0 1] \\
\text{17d} & \quad \text{[ SRA SRAX ] is SRAX} & \& \text{x} = [0 1]
\end{align*}
\]

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 SLLx 15, SRAx 15, SRLx 15, and x 4a.

RDPRx and WRPRx 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 rs1p and rdp are versions of rs1 and rd that have special names related to the registers (see section 2.2).

\[ \langle \text{pattern and constructor specifications} \rangle^+ \]
\[
\begin{align*}
\text{17c} & \quad \text{RDPR} & \text{is RDPRx} \\
\text{17d} & \quad \text{WRPR} & \text{is WRPRx}
\end{align*}
\]

Defines:
- RDPR, used in chunks 26 and 53.
- WRPR, used in chunks 26 and 53.

Uses RDPRx 15 and WRPRx 15.

POPCx is a placeholder for POPC which is only defined if rs1 is 0. All other encodings of rs1 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.

\[ \langle \text{pattern and constructor specifications} \rangle^+ \]
\[
\begin{align*}
\text{17e} & \quad \text{POPC} & \text{is POPCx} & \& \text{rs1} = 0
\end{align*}
\]

Defines:
- POPC, used in chunks 26c and 53.

Uses POPCx 15 and rs1 4a.

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

\[ \langle \text{pattern and constructor specifications} \rangle^+ \]
\[
\begin{align*}
\text{17f} & \quad \text{MOVr} & \text{is MOVR}
\end{align*}
\]

Defines:
- MOVR, used in chunk 30.

Uses MOVR 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.

\[
\begin{align*}
\text{patterns} & \\
[LDUW LDUWA LDF LDFA] & \\
LDUB & LDUBA LDFSRx _ \\
LDUH & LDUHA LDQF LDQFA \\
LDD & LDDA LDDF LDDFA \\
STW & STWA STF STFA \\
STB & STBA STFSRx _ \\
STH & STHA STQF STQFA \\
STD & STDIA STDF STDFA \\
LDSD & LSDWA _ _ \\
LDSD & LSDBA _ _ \\
LDSD & LDSHA _ _ \\
LDX & LDX A _ _ \\
LDSTUB & LDSTUBA PREFETCHx PREFETCHAx \\
STX & STXA _ CASXA \\
SWAP & SWAPA _ _ \\
\end{align*}
\]

is TABLE_33 & op3 = \{0 to 63 columns 4\}

Defines:

- CASA, used in chunks 19c and 47.
- CASAX, 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.
- LDDS, used in chunk 22b.
- LDDS, 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.
- LDSTUBA, used in chunks 22b and 53.
- LDSTUBA, used in chunks 22b and 53.
- LDSTUBA, used in chunks 22b and 53.
- LDSTUBA, used in chunks 22b and 53.
- STX, used in chunks 23 and 53.
- STDERR, used in chunks 23 and 53.
- STDERR, used in chunks 23 and 53.
- STDERR, used in chunks 23 and 53.
- STDERR, used in chunks 23 and 53.
- STDERR, used in chunks 23 and 53.
- STDERR, used in chunks 23 and 53.
- STDERR, used in chunks 23 and 53.
- STDERR, used in chunks 23 and 53.
- STDERR, used in chunks 23 and 53.
- STDERR, used in chunks 23 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 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\text{pattern and constructor specifications} 13d\rangle + \equiv
\langle\text{patterns}\rangle
\langle\text{ldfsr is any of [ LDFSR LDXFSR ]},
\text{which is LDFSRx & rd = [0 1]}\rangle
\langle\text{stfsr is any of [ STFSR STXFSR ]},
\text{which is STFSRx & rd = [0 1]}\rangle

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\text{pattern and constructor specifications} 13d\rangle + \equiv
\langle\text{patterns}\rangle
\langle[\text{PREFETCH PREFETCHA }]\rangle
\langle[\text{PREFETCHx PREFETCHAx }]\rangle

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\text{pattern and constructor specifications} 13d\rangle + \equiv
\langle\text{patterns}\rangle
\langle\text{casa is CASA | CASXA}\rangle

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.

\[ \langle \text{pattern and constructor specifications} \rangle = \]

\[
\text{patterns} \quad \text{float2 is any of} \quad [\text{FMOVs} \ FMOVd \ FMOVq \ \text{FNEGs} \ \text{FNEGd} \ \text{FNEGq} \\
\text{FABSs} \ \text{FABSd} \ \text{FABsq} \ \text{FSQRTs} \ \text{FSQRTd} \ \text{FSQRTq} \\
\text{FsTOx} \ \text{FdT0x} \ \text{FqTOx} \ \text{FxTOs} \ \text{FxTOd} \ \text{FxTOq} \\
\text{FiTOs} \ \text{FdT0s} \ \text{FqTOs} \ \text{FiTOd} \ \text{FsTOd} \ \text{FqTOd} \\
\text{FiTOq} \ \text{FsTOq} \ \text{FdT0q} \ \text{FsTOi} \ \text{FqTOi} \ ],
\]

which is FPop1 & opf =

\[
[ \ 0x1 \ 0x2 \ 0x3 \ 0x5 \ 0x6 \ 0x7 \\
0x9 \ 0xa \ 0xb \ 0x29 \ 0x2a \ 0x2b \\
0x81 \ 0x82 \ 0x83 \ 0x84 \ 0x88 \ 0x8c \\
0xc4 \ 0xc6 \ 0xc7 \ 0xc8 \ 0xc9 \ 0xcb \\
0xcc \ 0xce \ 0xd1 \ 0xd2 \ 0xd3 \ ]
\]

\[
\text{funarys} \ is \ \text{FMOVs} \ \text{FNEGs} \ \text{FABSs} \ \text{FSQRTs} \\
\text{funaryd} \ is \ \text{FMOVd} \ \text{FNEGd} \ \text{FABsd} \ \text{FSQRTd} \\
\text{funaryq} \ is \ \text{FMOVq} \ \text{FNEGq} \ \text{FABsq} \ \text{FSQRTq}
\]

\[
\text{f2sTOs is funarys} \quad \text{FsTOi} \quad \text{FiTOs} \\
\text{f2sT0d is FsTOx} \quad \text{FiTOd} \quad \text{FsTOd} \\
\text{f2sT0q is FiTOq} \quad \text{FsTOq} \\
\text{f2dT0s is FxTOs} \quad \text{FdTOi} \quad \text{FsTOd} \\
\text{f2dT0d is funaryd} \quad \text{FdTOx} \quad \text{FxTOd} \\
\text{f2dT0q is FxTOq} \quad \text{FdTOq} \\
\text{f2qT0s is FqTOi} \quad \text{FqTOs} \\
\text{f2qT0d is FqTOx} \quad \text{FqTOd} \\
\text{f2qT0q is funaryq}
\]

\[
\text{float3 is any of} \quad [\text{FADDs} \ \text{FADDD} \ \text{FADDq} \ \text{FSUBs} \ \text{FSUBd} \ \text{FSUBq} \\
\text{FMULs} \ \text{FMULD} \ \text{FMULq} \ \text{FDIVs} \ \text{FDIVd} \ \text{FDIVq} \\
\text{FsMULd} \ \text{FDMULq} ],
\]

which is FPop1 & opf =

\[
[ \ 0x41 \ 0x42 \ 0x43 \ 0x45 \ 0x46 \ 0x47 \\
0x49 \ 0x4a \ 0x4b \ 0x4d \ 0x4e \ 0x4f \\
0x69 \ 0x6e \ ]
\]

\[
\text{float3s} \ is \ \text{FADDs} \ \text{FSUBs} \ \text{FMULs} \ \text{FDIVs} \\
\text{float3d} \ is \ \text{FADDD} \ \text{FSUBd} \ \text{FMULd} \ \text{FDIVd} \\
\text{float3q} \ is \ \text{FADDq} \ \text{FSUBq} \ \text{FMULq} \ \text{FDIVq}
\]

Defines:
- FABSd, used in chunk 53.
- FABSq, used in chunk 53.
- FABSSs, used in chunk 53.
- FADDs, never used.
- 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.
FDTOl, 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.
FiTQq, used in chunk 53.
FiTQs, 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.
FNEDd, used in chunk 53.
FNEGq, used in chunk 53.
FNEGq, used in chunk 53.
f2qTOd, used in chunk 30d.
FqTOD, used in chunk 53.
FqTQL, 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.
FsTQL, used in chunk 53.
f2sTOq, used in chunk 30d.
FsTOq, used in chunk 53.
f2sTOs, used in chunk 30d.
FsTOS, 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.
FxTQq, used in chunk 53.
FxTOS, used in chunk 53.
Uses FPpopl 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.

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

\[
\text{[ FMOVScc, FMOVDcc, FMOVQcc ]}
\]
  - is FPop2 & opf_{low6} = {1 to 3} & mbz_f4_18 = 0

\[
\text{[ FMOVSr, FMOVDsr, FMOVQsr ]}
\]
  - 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.
- FCMPeq, used in chunk 53.
- fcmpared, used in chunk 31a.
- fcompared, used in chunk 31a.
- fcompares, used in chunk 31a.
- FCPes, never used.
- FCPer, never used.
- FMOVDcc, used in chunk 29.
- FMOVDR, used in chunk 30.
- FMOVQcc, used in chunk 29.
- FMOVQR, used in chunk 30.
- FMOVScce, used in chunk 29.
- FMOVsr, used in chunk 30.

Uses FPop2, 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.

Patterns

- loadg is LDSB | LDSH | LDSW | LDUB | LDUH | LDUW | LDX |
  - LDSTUB | SWAP
- loada is LDSBA | LDSHA | LDWSA | LDUA | LDUBA | LDUWA | LDXA |
  - LDSTUBA | SWAPA
- storeg is STB | STH | STW | STX
- storea is STBA | STA | 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, LDSTUBA 18, LDWSA 18, LDUA 18, LDUBA 18, LDUH 18, LDUWA 18, LDUX 18, LDXA 18, LDX 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.

```c
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).

```c
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.

```c
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)

```c
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 (casaasi)`.

```c
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.

\[
\text{(pattern and constructor specifications 13d)} + \equiv
\]

constructors
\[
\begin{align*}
\text{LDF} \ [\text{address}_\text{d}], \ rdfs \\
\text{LDDF} \ [\text{address}_\text{d}], \ rdfd \\
\text{LDQF} \ [\text{address}_\text{d}], \ rdfq \\
\text{STF} \ rdfs, \ [\text{address}_\text{d}] \\
\text{STDF} \ rdfd, \ [\text{address}_\text{d}] \\
\text{STQF} \ rdfq, \ [\text{address}_\text{d}] \\
\text{LDFA} \ \text{asi}_\text{address}, \ rdfs \\
\text{LDDFA} \ \text{asi}_\text{address}, \ rdfd \\
\text{LDQFA} \ \text{asi}_\text{address}, \ rdfq \\
\text{STFA} \ rdfs, \ \text{asi}_\text{address} \\
\text{STDF} \ rdfd, \ \text{asi}_\text{address} \\
\text{STQFA} \ rdfq, \ \text{asi}_\text{address}
\end{align*}
\]

Uses \text{address}_9a, \text{asi}_\text{address}_10a, \text{LDDF}_18, \text{LDDFA}_18, \text{LDF}_18, \text{LDFA}_18, \text{LDQF}_18, \text{LQFA}_18, \text{rdfd}_12b \text{12c}, \text{rdfq}_12b \text{12c}, \text{rdfs}_11a \text{12c}, \text{STDF}_18, \text{STDF}_18, \text{STF}_18, \text{STFA}_18, \text{STQF}_18, \text{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.

\[
\text{PREFETCH} \ [\text{address}_\text{d}], \ fcn
\]

when \{ fcn < 5 \} is \text{PREFETCH} \ & \text{address}_\text{d} \ & \ fcn

when \{ fcn > 15 \} is \text{PREFETCH} \ & \text{address}_\text{d} \ & \ fcn

\[
\text{PREFETCHA} \ \text{asi}_\text{address}, \ fcn
\]

when \{ fcn < 5 \} is \text{PREFETCHA} \ & \text{asi}_\text{address} \ & \ fcn

when \{ fcn > 15 \} is \text{PREFETCHA} \ & \text{asi}_\text{address} \ & \ fcn

Uses \text{address}_9a, \text{asi}_\text{address}_10a, \text{fcn}_4a, \text{PREFETCH}_19b, \text{PREFETCHA}_19b.

The \text{PREFETCH}{,A} instructions look like loads, but without any observable effects. Due to restrictions in the toolkit, the checks on \text{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.

\[
\text{PREFETCH} \ [\text{address}_\text{d}], \ fcn
\]

\[
\text{PREFETCHA} \ \text{asi}_\text{address}, \ fcn
\]

Uses \text{address}_9a, \text{asi}_\text{address}_10a, \text{fcn}_4a, \text{PREFETCH}_19b, \text{PREFETCHA}_19b.

\[
\text{sim pattern and constructor specifications 23d} + \equiv
\]

constructors
\[
\begin{align*}
\text{PREFETCH} \ [\text{address}_\text{d}], \ fcn \\
\text{PREFETCHA} \ \text{asi}_\text{address}, \ fcn
\end{align*}
\]

Uses \text{address}_9a, \text{asi}_\text{address}_10a, \text{fcn}_4a, \text{PREFETCH}_19b, \text{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}fsr). 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!).

\[(pattern and constructor specifications 13d) + \equiv \quad (13a) \langle 24b \ 26c \rangle\]

25a

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

WRY    rs1, reg_or_imm, "%y"
WRCCR rs1, reg_or_imm, "%ccr"
WRASI rs1, reg_or_imm, "%asi"
WRFPRS rs1, reg_or_imm, "%fprs"
SIR    simm13!
```

Uses MEMBAR 16, membar_mask 4a, rd 3c, RDASR 16, RDCCR 16, RDFSRS 16, RDPC 16, RDTICK 16, RDY 16, reg_or_imm 8b, rs1 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.

\[(full pattern and constructor specifications 23c) + \equiv \quad (13c) \langle 24c \ 26a \rangle\]

25b

```
constructors
RDASR "%asr"rs1i, rd { rsli > 15 }
WRASR rs1, reg_or_imm, "%asr"rdi { rdi > 15 }
```

Uses rd 3c, RDASR 16, rdi 5d, reg_or_imm 8b, rs1 4a, rsli 5d, and WRASR 16.

The registers rsli and rdi are versions of rs1 and rd that print as integers (so no fieldinfo specifications) and are defined in section 2.2.

The equations are not needed for simulation.

\[(sim pattern and constructor specifications 23d) + \equiv \quad (13b) \langle 24d \ 26b \rangle\]

25c

```
constructors
RDASR "%asr"rs1i, rd
WRASR rs1, reg_or_imm, "%asr"rdi
```

Uses rd 3c, RDASR 16, rdi 5d, reg_or_imm 8b, rs1 4a, rsli 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 read 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 \(\text{when}\) 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.

\[\{\text{full pattern and constructor specifications} 23c\} \equiv\]

\(\text{constructors}\)
\[
\begin{align*}
\text{RDPR } rs1p, \ rd \ &\text{ when } \{ rs1p < 16 \} \text{ is } RDPR &\& rs1p &\& rd \\
\text{when } \{ rs1p = 31 \} \text{ is } RDPR &\& rs1p &\& rd \\
\text{WRPR } rsl, \ \text{reg_or_imm}, \ rdp \ &\text{ when } rdp < 15 \\
\end{align*}
\]

Uses \(rd 3c, rdp 5e, RDPR 17c, \text{reg_or_imm} 8b, rl 4a, rs1p 5e, \) and \(\text{WRPR} 17c.\)

\[\{\text{sim pattern and constructor specifications} 23d\} \equiv\]

\(\text{constructors}\)
\[
\begin{align*}
\text{RDPR } rs1p, \ rd \\
\text{WRPR } rsl, \ \text{reg_or_imm}, \ rdp \\
\end{align*}
\]

Uses \(rd 3c, rdp 5e, RDPR 17c, \text{reg_or_imm} 8b, rl 4a, rs1p 5e, \) and \(\text{WRPR} 17c.\)

See section 2.2 for the definitions of \(rs1p\) and \(rdp.\)

\subsection*{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 \(\text{sim}m 13\) bit.

\[\{\text{pattern and constructor specifications} 13d\} \equiv\]

\(\text{patterns}\)
\[
\begin{align*}
\text{logical} &\text{ is } \text{AND } | \ \text{ANDcc } | \ \text{ANDN } | \ \text{ANDNcc } | \ \text{OR } | \ \text{ORcc } | \ \text{ORN } | \ \text{ORNcc } | \\
\text{XOR } | \ \text{XORcc } | \ \text{XNOR } | \ \text{XNORcc } \\
\text{arith} &\text{ is } \text{ADD } | \ \text{ADDC } | \ \text{ADDCc } | \ \text{TADDcc } | \ \text{TADDccTV } | \\
\text{ SUB } | \ \text{SUBcc } | \ \text{SUBC } | \ \text{SUBCc } | \ \text{TSUBcc } | \ \text{TSUBccTV } | \\
\text{MULSc } | \ \text{UMUL } | \ \text{SMUL } | \ \text{SMULcc } | \\
\text{UDIV } | \ \text{SDIV } | \ \text{UDIVcc } | \ \text{SDIVcc } | \\
\text{MULX } | \ \text{SDIVX } | \ \text{UDIVX } | \\
\text{SAVE } | \ \text{RESTORE } \\
\text{alu } &\text{ is logical } | \ \text{arith } \\
\text{shift32} &\text{ is } \text{SLL } | \ \text{SRL } | \ \text{SRA } \\
\text{shift64} &\text{ is } \text{SLLX } | \ \text{SRLX } | \ \text{SRAX } \\
\end{align*}
\]

\(\text{constructors}\)
\[
\begin{align*}
\text{alu } &\text{ rs1, reg_or_imm, } \ rd \\
\text{shift32} &\text{ rs1, reg_or_shcnt32, } \ rd \\
\text{shift64} &\text{ rs1, reg_or_shcnt64, } \ rd \\
\text{POPC } &\text{ reg_or_imm, } \ rd \\
\end{align*}
\]

Defines:
\(\text{alu, never used.}\)
\(\text{arith, never used.}\)
\(\text{logical, never used.}\)
\(\text{shift32, never used.}\)
\(\text{shift64, never used.}\)

Uses \(\text{ADD } 15, \text{ADDC } 15, \text{ADDCc } 15, \text{AND } 15, \text{ANDcc } 15, \text{ANDN } 15, \text{ANDNcc } 15, \text{MULSc } 15, \text{MULX } 15, \text{OR } 15, \text{ORcc } 15, \text{ORN } 15, \text{ORNcc } 15, \text{POPC } 17d, \text{rd 3c, reg_or_imm} 8b, \text{reg_or_shcnt32 8c, reg_or_shcnt64 8c, RESTORE } 15, \text{rl 4a, SAVE } 15, \text{SDIV } 15, \text{SDIVcc } 15, \text{SDIVX } 15, \text{SLL } 17b, \text{SLLX } 17b, \text{SMUL } 15, \text{SMULcc } 15, \text{SRA } 17b, \text{SRAx } 17b, \text{SRL } 17b, \text{SRLX } 17b, \text{SUB } 15, \text{SUBcc } 15, \text{SUBCc } 15, \text{TADDcc } 15, \text{TADDccTV } 15, \text{TSUBcc } 15, \text{TSUBccTV } 15, \text{UDIV } 15, \text{UDIVcc } 15, \text{UDIVX } 15, \text{UMUL } 15, \text{UMULcc } 15, \text{XNOR } 15, \text{XNORcc } 15, \text{XOR } 15, \) and \(\text{XORcc } 15.\)
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 \texttt{ILLTRAP} instruction because it causes an illegal-instruction trap if executed. We also define \texttt{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.

\begin{verbatim}
/fields of addrtoken (64) reloc 0:63
placeholder for instruction is ILLTRAP & imm22 = 0xbad
relocatable reloc
\end{verbatim}

Defines:
\texttt{reloc}, used in chunks 27–29, 47, and 48a.
Uses \texttt{ILLTRAP} 13e and \texttt{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 \((i,f)\texttt{cond}_f2\) 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.

\begin{verbatim}
\langle full pattern and constructor specifications 23c\rangle+\equiv
\end{verbatim}

\texttt{B}\^{}ic\texttt{cond}_f2\^{}a\ reloc \ ( \texttt{reloc} = \texttt{label} + 4 \ast \texttt{disp22}! \) is \ label: \texttt{Bicc} \ & \texttt{icond}_f2 \ & \texttt{a} \ & \texttt{disp22}

\begin{verbatim}
\texttt{FB}\^{}f\texttt{cond}_f2\^{}a\ reloc \ ( \texttt{reloc} = \texttt{label} + 4 \ast \texttt{disp22}! \) is \ label: \texttt{FBfcc} \ & \texttt{fcond}_f2 \ & \texttt{a} \ & \texttt{disp22}
\end{verbatim}

Uses a 3c, Bicc 13e, disp22 3c, FBfcc 13e, fcond\_f2 3c, icond\_f2 3c, and reloc 27a.

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 \((i,f)\texttt{cc}_f2\) (see tables 39 and 40 respectively).

\begin{verbatim}
\langle full pattern and constructor specifications 23c\rangle+\equiv
\end{verbatim}

\texttt{BP}\^{}ic\texttt{cond}_f2\^{}\texttt{a}\^{}p \texttt{icc}_f2, reloc \ { \texttt{icc}_f2\&[0] = 0, \ reloc = \texttt{label} + 4 \ast \texttt{disp19}! \} is \ label: \texttt{BPcc} \ & \texttt{icond}_f2 \ & \texttt{a} \ & \texttt{p} \ & \texttt{icc}_f2 \ & \texttt{disp19}

\begin{verbatim}
\texttt{FBP}\^{}f\texttt{cond}_f2\^{}\texttt{a}\^{}p \texttt{fcc}_f2, reloc \ { \texttt{reloc} = \texttt{label} + 4 \ast \texttt{disp19}! \} is \ label: \texttt{FBPfcc} \ & \texttt{fcond}_f2 \ & \texttt{a} \ & \texttt{p} \ & \texttt{fcc}_f2 \ & \texttt{disp19}
\end{verbatim}

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.

The \texttt{sim pattern and constructor specifications} (23d)+\equiv

\begin{verbatim}
\langle sim pattern and constructor specifications 23d\rangle+\equiv
\end{verbatim}

\texttt{BP}\^{}ic\texttt{cond}_f2\^{}\texttt{a}\^{}p \texttt{icc}_f2 \ & \texttt{disp19}!

\begin{verbatim}
\texttt{FBP}\^{}f\texttt{cond}_f2\^{}\texttt{a}\^{}p \texttt{fcc}_f2 \ & \texttt{disp19}!
\end{verbatim}

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 \text{Tcc} instructions depend on the \text{icc\_f4} field to determine which condition code to check (see Table 40), and on the \text{icond\_f2} field to determine how to check it:

28a \begin{align*}
&\text{\texttt{\{full pattern and constructor specifications\} 23c}}+\equiv \\
&\text{\texttt{constructors}} \\
&\text{T\texttt{\_icond\_f2 icc\_f4, \text{software\_trap\_number}}} \ (\text{icc\_f4}[0] = 0) \\
&\text{is Tcc \& icond\_f2 \& icc\_f4 \& \text{software\_trap\_number}}
\end{align*}

Uses \text{icc\_f4} 4b, \text{icond\_f2} 3c, \text{software\_trap\_number} 9b, and \text{Tcc} 15.

28b \begin{align*}
&\text{\texttt{\{sim pattern and constructor specifications\} 23d}}+\equiv \\
&\text{\texttt{constructors}} \\
&\text{Tcc icond\_f2 icc\_f4 \text{software\_trap\_number}}
\end{align*}

Uses \text{icc\_f4} 4b, \text{icond\_f2} 3c, \text{software\_trap\_number} 9b, and \text{Tcc} 15.

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

28c \begin{align*}
&\text{\texttt{\{full pattern and constructor specifications\} 23c}}+\equiv \\
&\text{\texttt{constructors}} \\
&\text{CALL reloc \ \{(reloc = label + 4 \ast \text{disp30})\ \} is label: CALL \& \text{disp30}}
\end{align*}

Uses \text{CALL} 13d, \text{disp30} 3b, and \text{reloc} 27a.

28d \begin{align*}
&\text{\texttt{\{sim pattern and constructor specifications\} 23d}}+\equiv \\
&\text{\texttt{constructors}} \\
&\text{CALL \text{disp30}}
\end{align*}

Uses \text{CALL} 13d and \text{disp30} 3b.

\subsection*{5.5 \textit{Conditional moves and integer register conditions}}

The condition-code instructions \text{MOVcc} and \text{FMOVcc} are very similar to the branch instructions, but the organisation of the \{i,f\}\text{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 \{i,f\}).

By inspection of the definitions of \{(F\text{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\}\text{cond\_f4} are directly equivalent to \{i,f\}\text{cond\_f2} (and hence are named just like them in section 2.2).}

We specialise constructors for \text{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 \text{cc2\_f4} is implied by (and distinguishes between) the conditions (integer or floating-point condition code).

28e \begin{align*}
&\text{\texttt{\{full pattern and constructor specifications\} 23c}}+\equiv \\
&\text{\texttt{constructors}} \\
&\text{MOV\_icond\_f4 icc\_f4, reg\_or\_imm11, rd} \ (\text{icc\_f4}[0] = 0) \\
&\text{is MOVcc \& cc2\_f4 = 1 \& \text{icond\_f4} \& \text{icc\_f4} \& \text{reg\_or\_imm11} \& \text{rd}}
\end{align*}

Uses \text{cc2\_f4} 4b, \text{fcc\_f4} 4b, \text{fcond\_f4} 4b, \text{icc\_f4} 4b, \text{icc\_f4} 4b, \text{icc\_f4} 4b, \text{MOVcc} 15, \text{rd} 3c, and \text{reg\_or\_imm11} 8c.

For simulation, we pass all fields as parameters, including the switch for floating-point or integer condition-codes (note that \text{icond\_f4} is not necessarily an integer condition-code!).

28f \begin{align*}
&\text{\texttt{\{sim pattern and constructor specifications\} 23d}}+\equiv \\
&\text{\texttt{constructors}} \\
&\text{MOVcc cc2\_f4 iccond\_f4 icc\_f4 reg\_or\_imm11 rd}
\end{align*}

Uses \text{cc2\_f4} 4b, \text{icc\_f4} 4b, \text{icc\_f4} 4b, \text{icc\_f4} 4b, \text{MOVcc} 15, \text{rd} 3c, and \text{reg\_or\_imm11} 8c.
As for MOVcc, we specialise constructors for $\text{FMOV}(S,D,Q)cc$ in a similar fashion. This time, however, the $\text{opf}_cc$ is one field instead of two.

Constructors

$$\text{FMOVS}^{\text{icond}_f4 \text{opf}_cc, \text{rs2fs}, \text{rdfs}} \begin{cases} \text{opf}_cc[2] = 1, \text{opf}_cc[0] = 0 \end{cases}$$

is $\text{FMOVS}cc$ & $\text{icond}_f4$ & $\text{opf}_cc$ & $\text{rs2fs}$ & $\text{rdfs}$

$$\text{FMOVSF}^{\text{fcond}_f4 \text{opf}_cc, \text{rs2fs}, \text{rdfs}} \begin{cases} \text{opf}_cc[2] = 0 \end{cases}$$

is $\text{FMOVS}cc$ & $\text{fcond}_f4$ & $\text{opf}_cc$ & $\text{rs2fs}$ & $\text{rdfs}$

$$\text{FMOVD}^{\text{icond}_f4 \text{opf}_cc, \text{rs2fd}, \text{rdfd}} \begin{cases} \text{opf}_cc[2] = 1, \text{opf}_cc[0] = 0 \end{cases}$$

is $\text{FMOVD}cc$ & $\text{icond}_f4$ & $\text{opf}_cc$ & $\text{rs2fd}$ & $\text{rdfd}$

$$\text{FMOVDF}^{\text{fcond}_f4 \text{opf}_cc, \text{rs2fd}, \text{rdfd}} \begin{cases} \text{opf}_cc[2] = 0 \end{cases}$$

is $\text{FMOVD}cc$ & $\text{fcond}_f4$ & $\text{opf}_cc$ & $\text{rs2fd}$ & $\text{rdfd}$

$$\text{FMOVQ}^{\text{icond}_f4 \text{opf}_cc, \text{rs2fq}, \text{rdfq}} \begin{cases} \text{opf}_cc[2] = 1, \text{opf}_cc[0] = 0 \end{cases}$$

is $\text{FMOVQ}cc$ & $\text{icond}_f4$ & $\text{opf}_cc$ & $\text{rs2fq}$ & $\text{rdfq}$

$$\text{FMOVQF}^{\text{fcond}_f4 \text{opf}_cc, \text{rs2fq}, \text{rdfq}} \begin{cases} \text{opf}_cc[2] = 0 \end{cases}$$

is $\text{FMOVQ}cc$ & $\text{fcond}_f4$ & $\text{opf}_cc$ & $\text{rs2fq}$ & $\text{rdfq}$

Integer register condition branches: the difficulty with $\text{BPr}$ is that the displacement is assigned in two parts: $\text{d16}{\text{hi,lo}}$. 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 ($\text{disp16}$) to simplify things (it is only used internally so does not belong with the other types in section 3).

Constructors

$$\text{disp16 disp!} : \text{disp16t} \begin{cases} \text{disp}[0:1] = 0, \text{disp}[2:15] = \text{disp}[16:31] \end{cases}$$

is $\text{disp16}$ & $\text{disp16t}$

$$\text{BPr rcond}_f2 \text{a p rs1, reloc} \begin{cases} \text{rcond}_f2 = 0, \text{rcond}_f2 = 4 \end{cases}$$

is label: $\text{BPr}$ & $\text{rcond}_f2$ & $\text{a}$ & $\text{p}$ & $\text{rs1}$ & $\text{disp16}($reloc - label$)$

Uses a $3c$, $\text{BPr}$ 14, $\text{d16hi}$ 3c, $\text{d16lo}$ 3c, $\text{p}$ 3c, $\text{rcond}_f2$ 3c, $\text{reloc}$ 27a, and $\text{rs1}$ 4a.

The $\text{disp}[0:1] = 0$ line can be commented out for speed in encoding if it doesn’t need to be checked (it is equivalent to saying $\text{disp} = 4 * \_\_ \_$. The $\text{rcond}_f2$ equations are commented out since they confuse the toolkit; the user must ensure that only the correct conditions are used.
The integer register condition moves are fairly trivial. The equations are commented out since they confuse the toolkit.

The equations are commented out since they confuse the toolkit.

\[
\begin{align*}
\text{MOVR}^{rcond_f3} & \quad rs1, \text{reg_or_imm10}, \text{rd} \quad \{ rcond_f3 != 0, rcond_f3 != 4 \} \\
\text{FMOVSR}^{rcond_f4} & \quad rs1, \text{rs2fs}, \text{rdfs} \quad \{ rcond_f4 != 0, rcond_f4 != 4 \} \\
\text{FMOVDR}^{rcond_f4} & \quad rs1, \text{rs2fd}, \text{rdfd} \quad \{ rcond_f4 != 0, rcond_f4 != 4 \} \\
\text{FMOVQR}^{rcond_f4} & \quad rs1, \text{rs2fq}, \text{rdfq} \quad \{ rcond_f4 != 0, rcond_f4 != 4 \}
\end{align*}
\]

Uses \text{FMOVDR} 22a, \text{FMOVQR} 22a, \text{MOVR} 17e, \text{rcond_f3} 4a, \text{rcond_f4} 4b, \text{rd} 3c, \text{rdfd} 12b 12c, \text{rdfq} 12b 12c, \text{rdfs} 11a 12c, \text{reg_or_imm10} 8c, \text{rs1} 4a, \text{rs2fd} 12b 12c, \text{rs2fq} 12b 12c, and \text{rs2fs} 11a 12c.

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

\[
\begin{align*}
\text{MOVR}^{rcond_f3} & \quad rs1, \text{reg_or_imm10}, \text{rd} \\
\text{FMOVSR}^{rcond_f4} & \quad rs1, \text{rs2fs}, \text{rdfs} \\
\text{FMOVDR}^{rcond_f4} & \quad rs1, \text{rs2fd}, \text{rdfd} \\
\text{FMOVQR}^{rcond_f4} & \quad rs1, \text{rs2fq}, \text{rdfq}
\end{align*}
\]

Uses \text{FMOVDR} 22a, \text{FMOVQR} 22a, \text{MOVR} 17e, \text{rcond_f3} 4a, \text{rcond_f4} 4b, \text{rd} 3c, \text{rdfd} 12b 12c, \text{rdfq} 12b 12c, \text{rdfs} 11a 12c, \text{reg_or_imm10} 8c, \text{rs1} 4a, \text{rs2fd} 12b 12c, \text{rs2fq} 12b 12c, and \text{rs2fs} 11a 12c.

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

\[
\begin{align*}
\text{MOVR}^{rcond_f3} & \quad rs1, \text{reg_or_imm10}, \text{rd} \\
\text{FMOVSR}^{rcond_f4} & \quad rs1, \text{rs2fs}, \text{rdfs} \\
\text{FMOVDR}^{rcond_f4} & \quad rs1, \text{rs2fd}, \text{rdfd} \\
\text{FMOVQR}^{rcond_f4} & \quad rs1, \text{rs2fq}, \text{rdfq}
\end{align*}
\]

Uses a 3c.

\section*{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.

\[
\begin{align*}
\text{f2sTOS} \quad & \text{rs2fs}, \text{rdfs} \\
\text{f2sTOD} \quad & \text{rs2fs}, \text{rdfd} \\
\text{f2sTOq} \quad & \text{rs2fs}, \text{rdfq} \\
\text{f2dTOS} \quad & \text{rs2fd}, \text{rdfs} \\
\text{f2dTOD} \quad & \text{rs2fd}, \text{rdfd} \\
\text{f2dTOq} \quad & \text{rs2fd}, \text{rdfq} \\
\text{f2qTOS} \quad & \text{rs2fq}, \text{rdfq} \\
\text{f2qTOD} \quad & \text{rs2fq}, \text{rdfd} \\
\text{f2qTOq} \quad & \text{rs2fq}, \text{rdfs}
\end{align*}
\]

Uses \text{f2dTOD} 20, \text{f2dTOq} 20, \text{f2dTOS} 20, \text{f2qTOD} 20, \text{f2qTOq} 20, \text{f2qTOS} 20, \text{f2sTOD} 20, \text{f2sTOS} 20, \text{rdfd} 12b 12c, \text{rdfq} 12b 12c, \text{rdfs} 11a 12c, \text{rs2fd} 12b 12c, \text{rs2fq} 12b 12c, and \text{rs2fs} 11a 12c.
The three-operand variants, based on their operand types are listed here.

5.7 Miscellany

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

Then some of the remaining instructions:

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.
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].

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:

<table>
<thead>
<tr>
<th>ASR Value</th>
<th>Name</th>
<th>Assembler</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10</td>
<td>PERF_CONTROL_REG</td>
<td>%pcr</td>
<td>RW</td>
</tr>
<tr>
<td>0x11</td>
<td>PERF_COUNTER</td>
<td>%pic</td>
<td>RW</td>
</tr>
<tr>
<td>0x12</td>
<td>DISPATCH_CONTROL_REG</td>
<td>%dcr</td>
<td>RW</td>
</tr>
<tr>
<td>0x13</td>
<td>GRAPHIC_STATUS_REG</td>
<td>%gsr</td>
<td>RW</td>
</tr>
<tr>
<td>0x14</td>
<td>SET_SOFTINT</td>
<td>%set_softint</td>
<td>W</td>
</tr>
<tr>
<td>0x15</td>
<td>CLEAR_SOFTINT</td>
<td>%clear_softint</td>
<td>W</td>
</tr>
<tr>
<td>0x16</td>
<td>SOFTINT_REG</td>
<td>%softint</td>
<td>RW</td>
</tr>
<tr>
<td>0x17</td>
<td>TICK_CMPR_REG</td>
<td>%tick_cmpr</td>
<td>RW</td>
</tr>
</tbody>
</table>

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.

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.

6.2 Shutdown

The `SHUTDOWN` instruction is decidedly simple.

```
<ultrasparc patterns and constructors 32b>+
patterns
    SHUTDOWN is IMPDEP1 & opf = 0x80
constructors
    SHUTDOWN
Defines:
    SHUTDOWN, used in chunk 40.
Uses: rd 3c, reg_or_imm 8b, rs1 4a, and WRASR 16.
```

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`, `visopl` and `visop2` — of two, two and five bits respectively.

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

Defines:
- `vismbz`, used in chunk 34a.
- `visopl`, used in chunk 34a.
- `visop2`, used in chunks 34–37.
\[(\text{ultrasparc patterns and constructors } 32b)\] \(\equiv\)
\[\text{patterns}\]
\[
\begin{align*}
\{ \text{ VISOP0 VISOP1 VISOP2 VISOP3 } \} & \text{ is} \\
\text{IMPDEP1 & vismbz = 0 & visop1 = \{0 to 3\}}
\end{align*}
\]
Defines:
- VISOP0, used in chunk 34b.
- VISOP1, used in chunk 35.
- VISOP2, used in chunk 36.
- VISOP3, used in chunk 37.
Uses IMPDEP1, vismbz, and visop1.

6.3.1 Edge handling, array and align address

\[(\text{ultrasparc patterns and constructors } 32b)\] \(\equiv\)
\[\text{patterns}\]
\[
\begin{align*}
\text{visop0rrr is any of} \\
\{ \text{ EDGE8 EDGE32 ARRAY8 ALIGNADDRESS} \\
\phantom{\text{visop0rrr is any of}} \text{ EDGE8L EDGE32L ARRAY16 ALIGNADDRESS_LITTLE} \\
\phantom{\text{visop0rrr is any of}} \text{ EDGE16 _ ARRAY32 _} \\
\phantom{\text{visop0rrr is any of}} \text{ EDGE16L _ _ _} \\
\phantom{\text{visop0rrr is any of}} \text{ which is VISOP0 & visop2 = \{0 to 31 columns 4\}}
\end{align*}
\]
Defines:
- ALIGNADDRESS, used in chunk 40.
- ALIGNADDRESS_LITTLE, never used.
- ARRAY16, used in chunk 40.
- ARRAY32, used in chunk 40.
- EDGE8, used in chunk 40.
- EDGE16, used in chunk 40.
- EDGE32, used in chunk 40.
- EDGE8L, used in chunk 40.
- EDGE16L, used in chunk 40.
- visop0rrr, used in chunk 38.
Uses VISOP0, visop2, and VISOP3.

We group the instructions by their parameter types. The \text{r}'s in \text{visop0rrr} refer to a plain integer register; three \text{r}'s indicates that all three registers are used.
6.3.2 Pixel compare, partitioned multiply, pack and pixel distance

\begin{align*}
\text{patterns} & \quad \begin{array}{l}
[ \text{FCMPEQ16} \quad \text{FCMPGT16} \quad \ldots \quad \text{FMULD8SUx16} \\
\phantom{[} \quad \ldots \quad \text{FMUL8x16} \quad \text{FMULD8ULx16} \\
\text{FCMPE16} \quad \text{FCMPEQ16} \quad \ldots \quad \text{FPACK32} \\
\phantom{[} \quad \ldots \quad \text{FMUL8x16AU} \quad \text{FPACK16} \\
\text{FCMPE32} \quad \text{FCMPGT32} \quad \ldots \quad \\
\phantom{[} \quad \ldots \quad \text{FMUL8x16AL} \quad \text{FPACKFIX} \\
\text{FCMPE32} \quad \text{FCMPEQ32} \quad \text{FMUL8SUx16} \quad \text{PDIST} \\
\phantom{[} \quad \ldots \quad \text{FMUL8ULx16} \quad \\
\end{array} \\
\text{is VISOP1 & visop2 = \{0 to 31 columns 4\}}
\end{align*}

\begin{align*}
\text{visop1ddr} & \quad \text{FCMPE16} \quad \text{FCMPEQ16} \mid \text{FCMPGT16} \mid \text{FCMPE16} \mid \text{FCMPEQ16} \\
\phantom{\text{visop1ddr}} & \quad \text{FCMPE32} \quad \text{FCMPEQ32} \mid \text{FCMPGT32} \mid \text{FCMPE32} \mid \text{FCMPEQ32} \\
\text{visop1sdd} & \quad \text{FMUL8x16} \mid \text{FMUL8x16AU} \\
\phantom{\text{visop1sdd}} & \quad \text{FMUL8x16AL} \mid \text{FMULD8SUx16} \mid \text{FMULD8ULx16} \\
\text{visop1ddd} & \quad \text{FMUL8SUx16} \mid \text{FMUL8ULx16} \mid \text{FPACK32} \mid \text{PDIST} \\
\phantom{\text{visop1ddd}} & \quad \text{FPACK16} \mid \text{FPACKFIX} \\
\text{visop1ds} & \quad \text{FPACK32} \mid \text{FPACKFFIX} \\
\end{align*}

Defines:
FCMPEQ16, used in chunk 40.
FCMPEQ32, never used.
FCMPE16, used in chunk 40.
FCMPE32, used in chunk 40.
FCMPE16, used in chunk 40.
FCMPE32, used in chunk 40.
FCMPE16, used in chunk 40.
FCMPE32, used in chunk 40.
FMULD8SUx16, used in chunk 40.
FMULD8ULx16, used in chunk 40.
FMUL8x16, 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.
visoplddd, used in chunk 38.
visopl1ddr, used in chunk 38.
visop1ds, used in chunk 38.
visop1sdd, used in chunk 38.
visop1ssd, used in chunk 38.
Uses VISOP1 33a 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 visop1sdd 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

Patterns:

```plaintext
[_] 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 | FPADD32 | 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.
Visop2ssd, used in chunk 38.
Visop2sd, 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

\[ \langle \text{ultrasparc patterns and constructors} \rangle^{32b} + \equiv \]

\begin{verbatim}
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 ]
\end{verbatim}

\begin{verbatim}
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
visop3sss 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
\end{verbatim}

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.
FNOT1S, never used.
FNOT1S, used in chunk 40.
FNOT2, 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.
visop3ss, used in chunk 38.
visop3sss, used in chunk 38.

The × “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.

\[
\text{ultrasparc patterns and constructors } 32b + \equiv (32a) + 37 39ab
\]

\[
\begin{align*}
\text{visops} & \text{ is } \text{visop3s} \\
\text{visopd} & \text{ is } \text{visop3d} \\
\text{visopss} & \text{ is } \text{visop3ss} \\
\text{visopsd} & \text{ is } \text{visop2sd} \\
\text{visopds} & \text{ is } \text{visop1ds} \\
\text{visopdd} & \text{ is } \text{visop3dd} \\
\text{visopsxs} & \text{ is } \text{visop3sxs} \\
\text{visopdxd} & \text{ is } \text{visop3dxd} \\
\text{visopsds} & \text{ is } \text{visop3dxd} \\
\text{visopdd} & \text{ is } \text{visop1ddd} \\
\text{visopdr} & \text{ is } \text{visop1ddr} \\
\text{visoprr} & \text{ is } \text{visop0rr}
\end{align*}
\]

Defines:

\[
\begin{align*}
\text{visopd}, \text{used in chunk 39a.} \\
\text{visopdd}, \text{used in chunk 39a.} \\
\text{visopddd}, \text{used in chunk 39a.} \\
\text{visopddr}, \text{used in chunk 39a.} \\
\text{visopds}, \text{used in chunk 39a.} \\
\text{visopdxd}, \text{used in chunk 39a.} \\
\text{visoprrr}, \text{used in chunk 39a.} \\
\text{visops}, \text{used in chunk 39a.} \\
\text{visopsd}, \text{used in chunk 39a.} \\
\text{visopsdd}, \text{used in chunk 39a.} \\
\text{visopsas}, \text{used in chunk 39a.} \\
\text{visopss}, \text{used in chunk 39a.} \\
\text{visopss}, \text{used in chunk 39a.} \\
\text{visopss}, \text{used in chunk 39a.}
\end{align*}
\]

Uses visop3d 37, visop3dd 37, visop1ddd 35, visop2ddd 36, visop3ddd 37, visop1ddr 35, visop1ds 35, visop3dxd 37, visop0rr 34b, visops 37, visop2sd 36, visop1add 35, visop3ss 37, visop1ssd 35, visop2ssd 36, visop2sss 36, visop3sss 37, and visop3sxs 37.
These patterns can then be turned into constructors with the correct parameters.

\[
\langle \text{ultrasparc patterns and constructors 32b} \rangle + \equiv
\]

constructors

<table>
<thead>
<tr>
<th>constructor</th>
<th>parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>visops rdfs</td>
<td></td>
</tr>
<tr>
<td>visopd rdfd</td>
<td></td>
</tr>
<tr>
<td>visopsrs rs2fs, rdfs</td>
<td></td>
</tr>
<tr>
<td>visopsd rs2fs, rdfd</td>
<td></td>
</tr>
<tr>
<td>visopds rs2fd, rdfs</td>
<td></td>
</tr>
<tr>
<td>visopdd rs2fd, rdfd</td>
<td></td>
</tr>
<tr>
<td>visopsxs rs1fs, rdfs</td>
<td></td>
</tr>
<tr>
<td>visopdxr rs1fd, rdfd</td>
<td></td>
</tr>
<tr>
<td>visopssrs rs1fs, rs2fs, rdfs</td>
<td></td>
</tr>
<tr>
<td>visopssdr rs1fs, rs2fs, rdfd</td>
<td></td>
</tr>
<tr>
<td>visopssdd rs1fs, rs2fd, rdfd</td>
<td></td>
</tr>
<tr>
<td>visopdddr rs1fd, rs2fd, rdfd</td>
<td></td>
</tr>
<tr>
<td>visoprrrr rs1, rs2, rd</td>
<td></td>
</tr>
</tbody>
</table>

Uses \( \text{rd} \), \( \text{rdfd} \), \( \text{rs1} \), \( \text{rs2} \), \( \text{rs1} \), \( \text{rs2} \), \( \text{rs2} \), \( \text{rs1} \), \( \text{rs2} \), \( \text{rs2} \), \( \text{rs1} \), \( \text{rs2} \), \( \text{rs2} \), \( \text{rs2} \), \( \text{rs1} \), \( \text{rs2} \), \( \text{rs2} \), \( \text{rs1} \), \( \text{rs2} \), \( \text{rs2} \), \( \text{rs1} \), \( \text{rs2} \), \( \text{rs2} \), and \( \text{visopsxs} \).

### 6.4 Assembler syntax

\[
\langle \text{ultrasparc-asm.spec 39b} \rangle \equiv
\]

- \# ultrasparc-asm.spec
- \# goes after asm.spec
- \langle \text{ultrasparc assembler syntax 39c} \rangle

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

\[
\langle \text{ultrasparc assembler syntax 39c} \rangle \equiv
\]

assembly opcode

- \{\text{RD}\} \{\text{PCR, PIC, DCR, GSR, SOFTINT, TICK_CMPR}\} is \( \text{\$1} \)
- \{\text{WR}\} \{\text{PCR, PIC, DCR, GSR, \{SET_, CLEAR_,\} SOFTINT, TICK_CMPR}\} is \( \text{\$1} \)

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

\[
\langle \text{ultrasparc assembler syntax 39c} \rangle + \equiv
\]

assembly opcode

- \{\text{ALIGNADDR}\} ESS is \( \text{\$1} \)
- \{\text{ALIGNADDR}\} ESS_\text{\( \text{L}\)}ITTLE is \( \text{\$1\$2} \)
6.5 Validating

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

```sh
## VIS instructions: tested 20000627
#discard
#RDPCR RDPIC RDDCR RDGSR RDSOFTINT RDTICK_CMPR WRPCR WRDCR WRGSR
#WRSET_SOFTINT WRCLEAR_SOFTINT WRSOFTINT WRTICK_CMPR SHUTDOWN FZEROS
#FONES FZERO FONE FN0T2S FN0T1S FSRC1S FSRC2S FEXPAND FPACK16 FPACKFIX
#FN0T2 FN0T1 FSRC1 FSRC2 FPADD16S FPADD32S FPSUB16S FPSUB32S FN0RS
#FANDNOT2S FANDNOT1S FXORS FNANDS FANDS FXNORS FORNOT2S FORNOT1S F0RS
#FMUL8x16AU FMUL8x16AL FMULD8SUx16 FMULD8ULx16 FMERGE FMUL8x16
#FMUL8SUx16 FMUL8ULx16 FPACK32 PDIST FALIGNDATA FPADD16 FPADD32 FPSUB16
#FPSUB32 FNOR FANDNOT2 FANDNOT1 FXOR FNAND FAND FXNOR FORNOT2 FORNOT1
#FOR FCMPLE16 FCMPGT16 FCMPNE16 FCMPEQ16 FCMPGT32 FCMPNE32
#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, FCMPTG16 35, FCMPGT32 35, FCMPLE16 35, FCMPLE32 35, FCMPNE16 35, FCMPNE32 35, FEXPAND 36, FMULD8SUx16 35, FMULD8ULx16 35, FMUL8x16 35, FMUL8x16AL 35, FNAND 37, FNANDS 37, FNOR 37, FNORS 37, FNOT 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, FMERGE 36, FPSUB16 36, FPSUB32 36, FPSUB32S 36, FZEROS 37, FZEROS 37, PDIST 35, RDDCR 32b, RDGSR 32b, RDPCRTCMPR 32b, RDPCR 33a, RDCR 33a, WRGSR 33a, WRGSR 33a, 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.

\[\text{sparc64.spec} 41a\]
\# requires fields.spec, types.spec and core.spec
\# (depends on table 32 and fp register encoding)
\(\langle\text{sparc64 patterns and constructors} 42\rangle\)

\[\text{sparc64-fields.spec} 41b\]
\# sparc64-fields.spec
\# goes after fields.spec (in fields declaration)
\(\langle\text{sparc64 fields} 41c\rangle\)

7.1 Ancillary state registers

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

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

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.

\[\text{sparc64 fields} 41c\] (41b) 45a-
\text{asrsub} 8:12 \text{rdasrmbz} 0:7 \text{wrasrmbz} 5:7

Defines:
- \text{asrsub}, used in chunks 42 and 44.
- \text{rdasrmbz}, used in chunk 42.
- \text{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.

```
(patterns)  \[ RHDW_MODE RGSR RSCHED_INT RTICK_MATCH RIFTYPE RSCRATCHx
                      RBRKPTx RDAFAULT_ADDR RDTYPE RD_PMx RDSCR \]
        is RDASR & rs1 = {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
                    RDAFAULT_ADDR "%dfaddr", rd
                    RDTYPE "%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.
RDAFAULT_ADDR, never used.
RDTYPE, 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, rdasrm2 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.

(patterns)

\[
\begin{align*}
\text{WR_HDW\_MODE} & \quad \text{WRGSR} & \quad \text{SET\_SCHED\_INT} & \quad \text{CLEAR\_SCHED\_INT} & \quad \text{WR\_SCHED\_INT} \\
\text{WR\_TICK\_MATCH} & \quad \text{WR\_SCRATCHx} & \quad \text{WR\_BRKx} & \quad \_ \_ \_ & \quad \text{WR\_PMx} & \quad \text{WRSCR} \\
is & \quad \text{WRASR} & \quad \text{rd} = \{0x12 \text{ to } 0x1f\}
\end{align*}
\]

(constructors)

\[
\begin{align*}
\text{WR\_HDW\_MODE} & \quad rs1, \; \text{reg\_or\_imm}, \; \"\text{hardware\_mode}\" \\
\text{WRGSR} & \quad rs1, \; \text{reg\_or\_imm}, \; \"\text{graphic\_status}\" \\
\text{SET\_SCHED\_INT} & \quad rs1, \; \text{reg\_or\_imm}, \; \"\text{set\_sched\_int}\" \\
\text{CLEAR\_SCHED\_INT} & \quad rs1, \; \text{reg\_or\_imm}, \; \"\text{clear\_sched\_int}\" \\
\text{WR\_SCHED\_INT} & \quad rs1, \; \text{reg\_or\_imm}, \; \"\text{sched\_int}\" \\
\text{WR\_TICK\_MATCH} & \quad rs1, \; \text{reg\_or\_imm}, \; \"\text{tick\_match}\" \\
\text{WRSCR} & \quad rs1, \; \text{reg\_or\_imm}, \; \"\text{scr}\"
\end{align*}
\]

(patterns)

\[
\begin{align*}
\text{WR\_SCRATCH0} & \quad \text{WR\_SCRATCH1} & \quad \text{WR\_SCRATCH2} & \quad \text{WR\_SCRATCH3} \\
is \quad \text{WR\_SCRATCHx} & \quad \text{asrsub} = \{0 \text{ to } 3\} & \quad \text{wrasrmzb} = 0 & \quad i = 0
\end{align*}
\]

(constructors)

\[
\begin{align*}
\text{WR\_SCRATCH0} & \quad rs1, \; rs2, \; \"\text{scratch0}\" \\
\text{WR\_SCRATCH1} & \quad rs1, \; rs2, \; \"\text{scratch1}\" \\
\text{WR\_SCRATCH2} & \quad rs1, \; rs2, \; \"\text{scratch2}\" \\
\text{WR\_SCRATCH3} & \quad rs1, \; rs2, \; \"\text{scratch3}\" \\
\text{WR\_BRK\_ADDR} & \quad rs1, \; rs2, \; \"\text{brk\_addr}\" \\
\text{WR\_BRK\_MASK} & \quad rs1, \; rs2, \; \"\text{brk\_mask}\" \\
\text{WR\_PM\_DIS} & \quad \"\text{pm\_dis}\" \\
\text{WR\_PM\_CLR\_DIS} & \quad \"\text{pm\_clr\_dis}\" \\
\text{WR\_PM\_EN} & \quad \"\text{pm\_en}\" \\
\text{WR\_PM\_CLR\_EN} & \quad \"\text{pm\_clr\_en}\" \\
\text{WR\_PM\_VN} & \quad rs1, \; rs2, \; \"\text{pm\_vn}\"
\end{align*}
\]

Defines:

\text{CLEAR\_SCHED\_INT}, \text{never used.}
\text{SET\_SCHED\_INT}, \text{never used.}
\text{WR\_BRK\_ADDR}, \text{never used.}
\text{WR\_BRK\_MASK}, \text{never used.}
\text{WR\_BRK\_X}, \text{never used.}
\text{WRGSR}, \text{used in chunk 40.}
\text{WR\_HDW\_MODE}, \text{never used.}
\text{WR\_PM\_CLR\_DIS}, \text{never used.}
\text{WR\_PM\_CLR\_EN}, \text{never used.}
\text{WR\_PM\_DIS}, \text{never used.}
\text{WR\_PM\_EN}, \text{never used.}
\text{WR\_PM\_VN}, \text{never used.}
\text{WR\_PMx}, \text{never used.}
\text{WR\_SCHED\_INT}, \text{never used.}
\text{WRSCR}, \text{never used.}
\text{WR\_SCRATCH0}, \text{never used.}
\text{WR\_SCRATCH1}, \text{never used.}
\text{WR\_SCRATCH2}, \text{never used.}
\text{WR\_SCRATCH3}, \text{never used.}
\text{WR\_SCRATCHx}, \text{never used.}
\text{WR\_TICK\_MATCH}, \text{never used.}

Uses asrsub 41c, i 4a, rd 3c, reg\_or\_imm 8b, rs1 4a, rs2 4a, WRASR 16, and \text{Wrasrmzb} 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).

\[ \text{sparc64 fields 41c} \]
\[ rs3 \ 9:13 \ var \ 7:8 \ size \ 5:6 \]

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.

\[ \text{sparc64-fpre-fields.spec 45b} \]
\[ \text{sparc64-fpre.spec 45c} \]
\[ \text{sparc64-nofpre-fields.spec 45d} \]

Uses rs3 45a, rs3fhi 45b, and rs3flo 45b.
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 rs1fs, rs2fs, rs3fs, rdfs
    fpmad rs1fd, rs2fd, rs3fd, rdfd

Defines:
FMADDd, never used.
FMADDs, never used.
FMSUBd, never used.
FMSUBs, never used.
FNMAADDd, never used.
FNMAADDs, never used.
FMSUBd, never used.
FMSUBs, never used.
fpmad, never used.
fpmas, never used.
Uses IMPDEP2 15, rdfd 12b 12c, rdfs 11a 12c, rs1fd 12b 12c, rs2fd 12b 12c, rs3fd 45c 45d, rs1fs 11a 12c, rs2fs 11a 12c, rs3fs 45c 45d, size 45a, and var 45a.

7.3 Assembler syntax

### Sparc64-asm.spec

```
# sparc64-asm.spec
# goes after asm.spec
```

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

```
{R}{HDW_MODE,GSR,SCHED_INT,TICK_MATCH,IETYPE} 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.

```plaintext
# there’s something wrong with synthetics.
# cmp rsi, reg_or_imm is SUBcc(rsi, reg_or_imm, "%g0")
# jmp address_ is JMPL (address_, "%g0")
# calla address_ is JMPL (address_, "%o7")
cmp rsi, reg_or_imm is SUBcc(rsi, 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 rsi is ORcc ("%g0", regROI(rsi), "%g0")
ret is JMPL (dispA("%17", 8), "%g0")
retl is JMPL (dispA("%o7", 8), "%g0")
restore_ is RESTORE ("%g0", regROI("%g0"), "%g0")
save_ is SAVE("%g0", regROI("%g0"), "%g0")
signx rsi is SRA (rd, regROS32("%g0"), rd)
signx2 rsi, rdi is SRA (rs1, regROS32("%g0"), rd)
not rsi is XNOR(rsi, regROI("%g0"), rd)
neg rdi is SUB ("%g0", regROI(rdi), rd)
neg2 rsi, rdi is SUB ("%g0", regROI(rsi), rd)
clas [rs1], rs2, rdi is CASA (rs1, imm_casa_asi(0x80), rs2, rdi)
casia [rs1], rs2, rdi is CASA (rs1, imm_casa_asi(0x88), rs2, rdi)
casx [rs1], rs2, rdi is CASA (rs1, imm_casa_asi(0x80), rs2, rdi)
casxl [rs1], rs2, rdi is CASA (rs1, imm_casa_asi(0x88), rs2, rdi)
inc rsi is ADD (rd, immROI(1), rd)
inc2 val!, rdi is ADD (rd, immROI(val), rd)
incccc rd is ADDcc (rd, immROI(1), rd)
incccc2 val!, rdi is ADDcc (rd, immROI(val), rd)
dec rsi is SUB (rd, immROI(1), rd)
dec2 val!, rdi is SUB (rd, immROI(val), rd)
decccc rd is SUBcc (rd, immROI(1), rd)
decccc2 val!, rdi is SUBcc (rd, immROI(val), rd)
bts rsi, reg_or_imm, rsi is ANDcc (rs1, reg_or_imm, "%g0")
bset rsi, reg_or_imm, rdi is OR (rd, reg_or_imm, rdi)
bclr rsi, reg_or_imm, rdi is ANDN (rd, reg_or_imm, rdi)
bto reg_or_imm, rsi is XOR (rd, reg_or_imm, rd)
clr rd is OR ("%g0", regROI("%g0"), rd)
clb [address_] is STB ("%g0", address_)
clrh [address_] is STH ("%g0", address_)
clrw [address_] is STW ("%g0", address_)
clr [address_] is STX ("%g0", address_)
clruw rd is SRL (rd, regROS32("%g0"), rd)
clruw2 rs1, rd is SRL (rs1, regROS32("%g0"), rd)
mov rsi, reg_or_imm, rdi is OR ("%g0", reg_or_imm, rd)
movr rsi, reg_or_imm, rd is OR ("%g0", regROI(rsi), rd)
```

Defines:
bclr, used in chunk 49b.
bset, used in chunk 49b.
btoq, used in chunk 49b.
btsr, used in chunk 49b.
calla, used in chunk 49b.
cas, used in chunk 49b.
cas1, used in chunk 49b.
casx, used in chunk 49b.
casx1, used in chunk 49b.
cir, used in chunk 49b.
cirb, used in chunk 49b.
cirh, used in chunk 49b.
ciruw, used in chunk 49b.
ciruw2, used in chunk 49b.
cirw, used in chunk 49b.
cirx, used in chunk 49b.
cmp, used in chunk 49b.
dec, used in chunk 49b.
dec2, used in chunk 49b.
deccc, used in chunk 49.
decccc2, used in chunk 49b.
derestore, 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, CASX 18, dispA 9a, imm_casa asi 10b, immROI 8b, JMPL 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, XOR 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)

\[ \text{example of synonyms 48a} \equiv \]
\[
 \text{constructors}
\]
\[
 \text{fbnz}^\wedge a \text{ reloc is FBNE}(a, \text{ reloc})
\]
\[
 \text{fbs}^\wedge a \text{ reloc is FBE} (a, \text{ reloc})
\]

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

\[ \text{synth-asm.spec 48b} \equiv \]
\[
 \text{# synth-asm.spec}
\]
\[
 \text{# goes after asm.spec}
\]
\[
 \text{\{synth assembler syntax 49a\}}
\]
The synthetic assembly codes were named differently due to overloading. We can allow them to be overloaded in assembler.

\[ \text{synth assembler syntax} \]
\[ \text{assembly opcode} \]
\{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).

\[ \text{synth-check.spec} \]
\[ \text{## there is something wrong with synthetics} \]
discard
cmp jmp calla iprefetch tst ret retl restore_ save_ signx signx2 not not2 neg neg2 cas cacc 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, casx1 47, clr 47, clrb 47, clrh 47, clruw 47, clruw2 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.

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):

\[
\begin{align*}
&\{LD,ST\}(F,FSR) \text{ is } $1 \\
&\{LD,ST\}(D,Q)F \text{ is } $1$2 \\
&\{LDX,STX\}FSR \text{ is } $1 \\
&\{LD,ST\}F[A] \text{ is } $1$2 \\
&\{LD,ST\}(D,Q)F[A] \text{ is } $1$2$3
\end{align*}
\]

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?)

\[
\begin{align*}
&\{FMOV\}(s,d,q) \text{ is } $1$2 \\
&\{FMOV\}(S,D,Q)F(*) \text{ is } $1$2$3 \\
&\{FMOV\}(S,D,Q)\{R\}{*} \text{ is } $1$3$2$4 \\
&\{FMOV\}(S,D,Q){*} \text{ is } $1$2$3
\end{align*}
\]

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.

\[
\begin{align*}
&\{MOV\}F(*) \text{ is } $1$2 \\
&\{MOV\}\{R\}{*} \text{ is } $1$2$3 \\
&\{MOV\}{*} \text{ is } $1$2
\end{align*}
\]

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).

\[
\begin{align*}
&\{RD\}\{Y,CCR,ASI,TICK,PC,FPRS,ASR\} \text{ is } $1 \\
&\{WR\}\{Y,CCR,ASI,FPRS,ASR\} \text{ is } $1
\end{align*}
\]
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 $\text{BPOS}$ since it would otherwise be replaced with $\text{BOS}$!

\[
\text{assembly component} \\
\{ \text{BPOS} \} \text{ is } $1$ \\
\{,F\}{B}\{P\}{} \text{ is } $1$\$2$\$3$
\]
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.

\[
\langle \text{decode.spec} \quad 52 \rangle \equiv \\
\text{address type is } "\text{unsigned}" \\
\text{address to integer using } "\%a" \\
\text{address add using } "\%a+\%o" \\
\text{fetch 32 using } "\text{fetch}_\text{word}(\%a)"
\]

Uses a 3c.
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.

```
(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
# F1Toq FsToq FxToq FdToq FqToq FqTox FqTox FqTod FMOVq FNEGq FABSq FSQRTq FADDq FSBq FUBq 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 LDSD LDUB LDUH LDUW LDUX LDSTUB SWAP LDD STB STH STW STX STDA LDSBA LDSHA LDSW LDBA LDBH
# STUBA SWAPA LDDA STBA STHA STWA STX A STDA
FUE MOVFLG MOVFNE MOVFULE MOVFE MOVFUG MOVFU MOVFGE MOVFUG MOVFUL MOVFG MOVFO MOVFUGE
#MOVFL MOVFA MOVFN MOVFLE FMVVSNE FMVVSNEG FMVVSPOS FMVVSGE FMVVSIG FMVVSQG FMVVSQUG
FUE FMVVSFLG FMVVSFNE FMVVSFULE FMVVSFUGE FMVVSFUG FMVVSFUL
#FMVVSFPG FMVVSFPG FMVVSFLF FMVVSFPA FMVVSFEN FMVVSFLE FMVVSFNEG FMVVSFPOS FMVVSFQLE
FULE FMVVSFLG FMVVSFUG FMVVSFUNG FMVVSFUE FMVVSFUGE FMVVSFULE
#FMVVSFLG FMVVSFUG FMVVSFUNG FMVVSFUE FMVVSFUGE FMVVSFULE
POS FMVVSFGE FMVVSFGU FMVVSFGQ FMVVSFGCC FMVVSFGCS FMVVSFGVL FMVVSFGLEU FMVVSFGQA FMVVSFGLE
FUE FMVVSFLG FMVVSFUG
#FMVVSFQLE FMVVSFQGE FMVVSFQUG FMVVSFQULL FMVVSFQUGE FMVVSFQUGE FMVVSFQQLE FMVVSFQQGV
#FMVRS MOVRLZ MOVRNL MOVRGZ MOVRSZ MOVRSRLZ MOVRSRNLZ MOVRSRZ MOVRSRZLZ MOVRSRZLNLZ
#MOVZ MOVREZ MOVRLZ MOVRGZ MOVRSZ MOVRSRLZ MOVRSRNLZ MOVRSRZ MOVRSRZLZ MOVRSRZLNLZ
Uses a 3c, ADD 15, ADDCC 15, ADDCCc 15, AND 15, ANDCC 15, ANDCCc 15, BPR 14, CASX 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,
FMULq 20, FDTOi 20, FDToq 20, FDTDq 20, FDTDs 20, FDTTo 20, FDTToq 20, FDTo 20, FDToq 20, FDTq 20, FDTqs 20,
FDTDq 20, FSQRTd 20, FSQRTq 20, FSQRTs 20, FSQRTc 20, FSQRTcc 20, FSTDq 20, FSTDs 20, FSTDTo 20, FSTDToq 20,
FSTDq 20, FSTDqs 20, FSTDTo 20, FSTDToq 20, ILLTRAP 13c, IMPL 15, LDD 18, LDLAQ 18, LDLDF 18, LDF 18, LDFSR 19a,
LDQ 18, LDQFA 18, LDQSA 18, LDQSH 18, LDSTB 18, LDSTBA 18, LDSTUA 18, LDSTU 18, LDUB 18, LDUBA 18, LDUA 18,
LDUN 18, LDUNA 18, LDX 18, LDXA 18, LDXF 18, LDXFA 19a, MEMBAR 16, MILd 18, MULX 15, NOP 13e, OR 15, ORcc 15,
ORN 15, ORNcc 15, POC 17d, POCPECT 17b, POCPECTA 19b, POCPECTCA 19b, RDAS 16, RDASR 16, RDCR 16, RDPFR 16, RDPR 17c,
RDTICK 16, RDY 16, RESTORE 15, RESTORED 17a, RETRY 17a, RETURN 15, SAVE 15, SAVED 17a, SDIV 15, SDIVc 15, SDIVX 15,
SETHI 13c, STR 16, SL 17b, SLN 17b, SMUL 15, SMULc 15, SRA 17b, SRAx 17b, SRL 17b, SRLx 17b, STH 18, STBA 18,
STBAR 16, STDA 18, STDF 18, STDF 18, STF 18, STFG 19a, STG 18, STHA 18, STFQ 18, STFA 18, STFQ 18, STFQ 18,
STH 18, STW 18, STX 18, STX 18, STXFSR 19a, SU 15, SUB 15, SUBC 15, SWAP 18, SWAPA 15, TADDc 15, TADDct 15,
TSD 15, TSDT 15, UDIV 15, UDIVC 15, UDIVX 15, UMUL 15, UMULc 15, WRAS 16, WRASR 16, WFPPR 16, WRPR 17c, WRY 16,
XNOR 15, XNORc 15, XOR 15, and XORc 15.

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


New version 1997-02 (including UltraSPARC-II) download from http://www.sun.com/microelectronics/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.


A List of Chunks

\langle asm.spec \rangle
\langle assembler syntax \rangle
\langle check.spec \rangle
\langle core-full.spec \rangle
\langle core-sim.spec \rangle
\langle core.spec \rangle
\langle decode.spec \rangle
\langle example of synonyms \rangle
\langle fdeqnv \rangle
\langle field specifications \rangle
\langle fields-names.spec \rangle
\langle fields.spec \rangle
\langle fpre type specifications \rangle
\langle fseqnv \rangle
\langle full pattern and constructor specifications \rangle
\langle pattern and constructor specifications \rangle
\langle properties of integer-register fields \rangle
\langle rdfass \rangle
\langle rs1fass \rangle
\langle rs2fass \rangle
\langle sim pattern and constructor specifications \rangle
\langle Sparc64 assembler syntax \rangle
\langle Sparc64 fields \rangle
\langle Sparc64 patterns and constructors \rangle
\langle Sparc64-asm.spec \rangle
\langle Sparc64-fields.spec \rangle
\langle Sparc64-fpre-fields.spec \rangle
\langle Sparc64-nofpre-fields.spec \rangle
\langle Sparc64.spec \rangle
\langle Synth assembler syntax \rangle
\langle Synth fields \rangle
\langle Synth-check.spec \rangle
\langle Type specifications \rangle
\langle Types-fpre.spec \rangle
\langle Types-nofpre-specs.spec \rangle
\langle Types.spec \rangle
\langle Ultrasparc assembler syntax \rangle
\langle Ultrasparc patterns and constructors \rangle
\langle Ultrasparc-as spec \rangle
\langle Ultrasparc-check.spec \rangle
\langle Ultrasparc-fields.spec \rangle
\langle Ultrasparc.spec \rangle
B Index

a: 3c, 6c, 27b, 27c, 27d, 27e, 29c, 29d, 30c, 47, 48a, 49a, 52, 53
absolutenA: 9h
absoluterP: 9b
absolutestN: 9b
ADD: 15, 26c, 47, 53
ADDCC: 15, 26c, 53
ADDCCc: 15, 26c, 47, 53
ADDCCc: 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
ANDNCc: 15, 26c, 53
arshi: 26c
ARRAY16: 34b, 40
ARRAY32: 34b, 40
ARRAY8: 34b, 40
asIaddress: 10a, 23b, 23c, 23d, 24a, 24c, 24d
asrsib: 41c, 42, 44
bclr: 47, 49b
Bic: 13e, 27b, 27c
BPcc: 13e, 27d, 27e
BPx: 14, 29c, 29d, 53
BPRx: 13e, 14
bset: 47, 49b
b-to-g: 47, 49b
bts: 47, 49b
CALL: 13d, 28c, 28d
calla: 47, 49b
cas: 47, 49b
casa: 19c, 23e
CASA: 18, 19c, 47
casa_asl: 10b, 23e
caslu: 47, 49b
cask: 47, 49b
cASX: 18, 19c, 47, 53
casx1: 47, 49b
cc2_f4: 4b, 28e, 28f
CLEAR_Sched_INT: 44
clr: 47, 49a, 49b
Clrbl: 47, 49b
Clrhi: 47, 49b
clrw2: 47, 49b
clrw: 47, 49a, 49b
clrw: 47, 49b
clrwx: 47, 49b
cmp: 47, 49b
dec2: 47, 49b
dec: 47, 49a, 49b
decce: 47, 49b
decce: 47, 49a, 49b
dlehi: 3c, 29c, 29d
disp19: 3c, 27d, 27c
disp22: 3c, 27b, 27c
disp30: 3b, 28c, 28d
dispA: 9a, 47
dispRI: 9b
dispSTN: 9b
dleilo: 3c, 29c, 29d
DONE: 17a, 31b, 53
donEX: 15, 17a
EDGE16: 34b, 40
EDGE32: 34b, 40
EDGE8: 34b, 40
EDGE161: 34b, 40
EDGE321: 34b, 40
EDGE81: 34b, 40
FABSd: 20, 53
FABSQ: 20, 53
FABS: 20, 53
FADD: 20
FADDD: 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
FBf: 13e, 27b, 27c
FBfpc: 13e, 27d, 27e
f2c_f3: 3c, 6b, 27d, 27e
f2c_f3: 3a, 6b, 31a
f2c_f4: 4b, 6b, 28e
FCDp: 22a, 53
FCDP: 22a, 53
FCMEQ16: 35, 40
FCMEQ16: 35, 40
FCMEQ32: 35
FCMEQ32: 35
FCMEQ: 22a, 53
FCMPGT16: 35, 40
FCMPGT32: 35, 40
FCMPLTE16: 35, 40
FCMPLTE32: 35, 40
FCMPNE16: 35, 40
FCMPNE32: 35, 40
FCMPQ: 22a, 53
fcn: 4a, 17a, 24c, 24d
fcompared: 22a, 31a
fcompareq: 22a, 31a
fcompared: 22a, 31a
fcond_2f: 3c, 6d, 27b, 27c, 27d, 27e
fcond_f4: 4b, 6d, 28e, 29a
FCPE: 22a
FCP: 22a
FDIVd: 20, 53
FDIVQ: 20, 53
FDIVS: 20, 53
FdMU: 20, 31a, 53
f2dToD: 20, 30d
FDT: 20, 53
f2dToQ: 20, 30d
FDTQ: 20, 53
f2dToS: 20, 30d
FDTS: 20, 53
FDTS: 20, 53
FDTS: 20, 53
FDTS: 20, 53
FEXPAND: 36, 40
FIToD: 20, 53
FIToQ: 20, 53
FIToS: 20
float2: 20
float3: 20
float3: 20, 31a
float3: 20, 31a
float3: 20, 31a
FLUSH: 15, 31c, 53
FLUSHW: 15, 31b, 53
FMADD: 46a
FMADDS: 46a
FMADD: 20, 53
FMADD: 20, 53
FMADD: 20, 53
FMADD: 22a, 29a, 29b
FMADD: 22a, 30a, 30b
FMAD: 20, 53
FMADV: 20, 53
FMADVCC: 22a, 29a, 29b
FMADV: 20, 53
FMADV: 20, 53
FMADV: 22a, 30a, 30b
FMADV: 20, 53
FMADV: 20, 53
FMADV: 22a, 29a, 29b
FMADV: 22a, 30a, 30b
FMADV: 20, 53
FMADV: 20, 53
FMADV: 22a, 29a, 29b
FMADV: 22a, 30a, 30b
FMADV: 46a
FMADV: 46a
FMULD: 20, 53
FMULD: 20, 53
FMULD8SUX16: 35, 40
FMULD8SUX16: 35, 40
FMULQ: 20, 53
FMULS: 20, 53
FMULS: 20, 53
FMUL8SUX16: 35
FMUL8SUX16: 35
FMUL8SUX16: 35, 40
FMUL8X16: 35, 40
FMUL8X16: 35, 40
size: 45a, 46a
SLL: 17b, 26c, 53
SLLX: 17b, 26c, 53
SMUL: 17b, 26c, 53
SMULcc: 15, 26c, 53
software_trap_num: 9b, 28a, 28b
SRA: 17b, 26c, 47, 53
Srax: 17b, 26c, 53
SRAx: 15, 17b
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
STFSRc: 19a, 24b
STFSRd: 19a, 53
STFSRex: 18, 19a
ST: 18, 22b, 47, 53
STHA: 18, 22b, 53
storea: 22b, 23b
storeg: 22b, 23a
STGF: 18, 24a, 53
STGFA: 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
SWAP: 18, 22b, 53
SWAPA: 18, 22b, 53
TABLE_31: 13d, 13e
TABLE_32: 13d, 15
TABLE_33: 13d, 18
TADDC: 15, 26c, 53
TADDCCTV: 15, 26c, 53
tcc: 15, 28a, 28b
tst: 47, 49b
TSUBcc: 15, 26c, 53
TSUBCCCTV: 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
visop1dd: 35, 38
visop2dd: 36, 38
visop3dd: 37, 38
visopdd: 38, 39a
visop3d: 37, 38
visopdd: 38, 39a
visop1dd: 35, 38
visop2dd: 36, 38
visop3dd: 37, 38
visopdd: 38, 39a
visop0rr: 34b, 38
visoprr: 38, 39a
visop3s: 37, 38
visops: 38, 39a
visop2sd: 36, 38
visopsd: 38, 39a
visop1sdd: 35, 38
visopsdd: 38, 39a
visop3ss: 37, 38
visopsss: 38, 39a
visop1ssd: 35, 38
visop2ssd: 36, 38
visopss: 38, 39a