LEON-REX Instruction Set Extension

Technical note 2016-05-27

Doc. No GRLIB-TN-0001

Issue 1.1

CHANGE RECORD

Issue	Date	Section / Page	Description
1.0	2015-10-27		First issue of document
1.1	2016-05-26	6.3.4, 6.3.7, 6.3.8, 6.3.10, 6.3.11, 6.3.12, 6.3.13, 6.3.14, 6.3.15, 6.4.6, 6.5.1, 6.5.2	Correct typos in opcode binary values and formats
		6.3.15	rta0 replaced with rta that takes argument 0-7
		6.3.13	The immediate field in relative loads is unsigned
		6.4.1, 6.4.2, 6.4.4	Show bit shuffling of immediate bits in format
		6.5.2	Add note to clarify encoding

TABLE OF CONTENTS

1 INTRODUCTION

1.1 Scope of the Document

This document describes the opcode format and instruction set for LEON-REX extension to the SPARC V8 instruction set, that has been developed as an optional add-on to some versions of the LEON processor family.

Only the software visible parts of the extension is covered, other aspects such as implementation details and exact instruction timing are outside the scope of this document. Also the exact details on how to detect if the extension is implemented in the hardware and how to enable it is left to the hardware and not in this document.

The architecture extension described in this document has been developed by Cobham Gaisler AB, Göteborg, Sweden.

1.2 Reference Documents

[RD1] “The SPARC architecture manual, version 8”, SPARC International inc.
[RD2] “GRLIB IP Core User's Manual”, Cobham Gaisler AB

2 ABBREVIATIONS

ABI	Application Binary Interface
RISC	Reduced Instruction Set Computing
SPARC	Scalable Processor ARChitecture, open processor architecture standard maintained by SPARC International
TBC	To Be Confirmed
TBD	To Be Defined

3 OVERVIEW

3.1 Background

The SPARC architecture was designed according to a RISC philosophy, where the instruction set is very regular and symmetric to allow fast hardware implementations and compilers with aggressive optimization techniques. However, this leads to an instruction set where there are many redundant instructions and the total code size per line of source code therefore can become quite large.

For embedded designs where very limited on-chip RAM is available, or in execute-from-ROM environments where the amount of ROM is limited, the code density can limit the maximum program size. Also, the low code density results in an increase in the amount of data needed to be fetched by the instruction cache, both due to the initial fetch size and due to the increased miss rate of the cache.

3.2 Design goals

The LEON-REX extension was developed with the main goal of reducing code size to allow larger programs to fit into RAM and off-line storage, and improve instruction cache hit rate by fitting more code into the same cache space. However, there were also additional constraints that needed to be taken into consideration.

One additional design goal was to allow full interoperability between REX code and regular V8 code at procedure level. This was in part for practical reasons, to allow the REX extension to be rolled out gradually while the software environment catches up, but also to allow keeping code in uncompressed format where this is preferred for performance reasons, or where the code has been extensively qualified/validated and it would be a major effort to re-do that work for the compressed version.

In addition to the above, the extension needed to be defined so that it can be added to the existing LEON3 processor pipeline without major redesign (for example to add additional pipeline stages) and without significantly reducing the maximum frequency of the processor for a given technology.

3.3 Key features

The above design inputs have resulted in an extension with the following key features, described further in the following sections of the document:

- Variable-length instruction set with 16/32/48 bits per instruction. Vast majority of SPARC V8 32-bit instructions can still be encoded in 32 bits or less.
- Branch delay slots removed to reduce code size (as delay slots can not always be filled).
- REX procedures follows the standard SPARC calling convention on both entry and outgoing calls, and REX and regular SPARC procedures can therefore call each other using the normal SPARC C ABI. Bit 0 of the return address is used to track REX state in function

calls and trap handlers.
- Backward-compatible modification to SAVE and ADD instruction using previously unused bits in opcode to enter REX mode in compressed function prologue.
- Backward-compatible modification to JMPL instruction to allow function return to REX function and return from trap handler to work using existing calling convention.

4 ADDITIONS TO V8 INSTRUCTION BEHAVIOR

4.1 Detecting and enabling REX functionality

How to detect and/or enable REX support is not specified here but left to the specific implementation. It is recommended to have three modes for enabling REX: enabled, illegal and transparent. Enabled means that all the additions in this section are available to software, illegal means that an attempt to use the features here will cause an exception, and transparent meaning that the pipeline behaves as if the extension did not exist at all for maximum backward compatibility.

Also it is recommended to have a version field to allow detecting future revisions of the REX extension. This document describes revision 1 of the REX extension.

4.2 SAVEREX

The SAVE instruction in SPARC is available in two forms:

- Immediate form, with one destination register, one source register and one signed 13-bit immediate offset
- Register form, with one destination register, two source registers, and 8 unused bits that are set to zero.

The REX extension adds a third form of SAVE called SAVEREX, which has the same function as SAVE but additionally has the side effect of switching to REX mode. The switch is instant, so the instruction following the SAVEREX is expected to be in REX encoding.

SAVEREX has one destination register, one source register and one 13-bit immediate offset, however the offset must be negative. The encoding used for SAVEREX is the same as the immediate form, except the immediate bit is inverted. Bit 12 of the opcode is used to distinguish between the SAVEREX form and the regular three-register form of SAVE.

In theory there may be a compatibility issue if software uses the unused bits of the three-register SAVE instruction for some other purpose, however no such usage is known to exist. The GNU disassembler does not even recognize the save with unused bits set as a valid opcode which is further evidence that such usage is very uncommon. In the unlikely event this causes an issue, it is

recommended to implement a transparent option as suggested in section 4.1.

4.3 ADDREX

Analogous to the SAVEREX, the ADD instruction adds a third form of the opcode called ADDREX. The intent of this variant is to allow implementing leaf functions where save is not desired.

4.4 JMPL additions

The JMPL instruction is extended to allow functions to return into the middle of a REX code block. To indicate this the lowest bit of the target address is set to 1 when jumping to a REX instruction.

Table 1 JMPL behavior in REX enabled and disabled mode

Target address	REX enabled	REX disabled, illegal	REX disabled, transparent
4N+0	Jump to 4N, V8 mode	Jump to 4N, V8 mode	Jump to 4N, V8 mode
4N+1	Jump to 4N, REX mode	Unaligned address or illegal instruction exception	Unaligned address exception
4N+2	Unaligned address exception	Unaligned address exception	Unaligned address exception
4N+3	Jump to 4N+2, REX mode	Unaligned address or illegal instruction exception	Unaligned address exception

When jumping to REX mode code, any control transfer instruction in the delay slot will not affect the execution flow of the REX code. This in particular applies to the RETT instruction when returning from a trap handler, the address supplied to RETT will be ignored however the other side effects of RETT (to PSR bits) will be performed as usual.

5 EXECUTION FLOW IN REX MODE

5.1 General

After entering REX mode via any of the methods described in section 4, the processor will begin decoding and executing REX instructions at the location pointed to by the program counter. After each instruction, the PC is advanced 2, 4 or 6 bytes to the following instruction, or it is changed to the target address in case of a taken branch or other control transfer instruction.

5.2 Instruction alignment

All REX instructions are aligned to halfword (two-byte) boundaries but there are otherwise no requirements on alignment. This means that instructions are allowed to cross cache-line and MMU page boundaries. The instructions are designed so the first two bytes are sufficient to decode the instruction size, so it is always possible to determine whether more data needs to be fetched.

Note that for object file compatibility with existing linkers that have not been adapted for the REX extensions, the call instruction may need to be aligned to a 4-byte rather than 2-byte boundary. However this is not a restriction in the hardware so this may be lifted once the linker can be modified to accept call instructions on odd addresses.

5.3 Delay slot removal

In REX mode, control transfer instructions have no delay slot, the next executed instruction after a branch will be either the branch target in the case of a taken branch, or the following instruction in case of a not taken branch.

This also means that the concept of an independent nPC register becomes meaningless in REX mode as the next instruction address is always determined by the current instruction. Another, equivalent, way to look at this is to say that nPC still exists but is locked down to always point to the following instruction and all branches have their delay slot annulled in case of taken branch.

5.4 Reduced register window fields

Most of the 16-bit instructions use a four-bit field for the destination and source registers, in order to increase the number of possible operations in the small 16-bit opcode space. This restricts these opcodes to only be able to specify half of the possible registers in the register window.

To simplify the decoding logic, the REX opcodes that have 5 bits and can specify any of the 32 registers in some cases use the same reduced 4-bit encoding with an extra bit added to select between the reduced and the “complementary” set of 16 registers. The mapping from reduced register number to register in both the reduced and the complementary set is tabulated below.

For floating-point registers the same scheme is used just to allow common decoding logic to the integer instructions, however the standard and complementary sets have been reversed to allow accessing register %f0-1 with most load/store instructions.

Table 2 Mapping of 4-bit register field to SPARC window registers

r4 field value (decimal)	Register	Complementary register
0-3	%o0-3 (%r8-11)	%l0-3 (%r16-19)
4-7	%l4-7 (%r20-23)	%o4-7 (%r12-15)
8-15	%i0-7 (%r24-31)	%g0-7 (%r0-7)

Table 3 Mapping of 4-bit register field to SPARC floating-point registers

r4 field value (decimal)	Register	Complementary register
0-3	%f16-19	%f8-11
4-7	%f12-15	%f20-23
8-15	%f0-7	%f24-31

5.5 Subroutine calls

Subroutines are called using the CALL instruction, that has the exact same opcode representation as the regular SPARC counterpart. The CALL instructions sets up the return address so that the standard SPARC return sequence will cause execution to return to the instruction after the CALL and resume in REX mode.

5.6 Traps

In case of a taken trap, the behaviour is similar to the regular case. The CWP will be decremented one step and the S and PS bits are updated as described in the SPARC manual. The execution jumps to the trap handler address determined by the %tbr register and the trap type.

The %l0 register in the trap handler's window is updated to point to the REX instruction that caused the trap, but with the least significant bit set to 1. The %l1 register is set to point to the following instruction, also with the least significant bit set to 1 if the following instruction is a REX mode instruction or 0 if it is not.

In the special case of a trap on the r_retrest instruction, %l1 is allowed to either point to the following instruction (the return address) or have the same value as the %l0 register (in other words, also point to the r_retrest instruction) in order to simplify the hardware implementation. This is normally not a problem as the window_underflow handler returns to the %l0 instruction.

6 INSTRUCTION DEFINITIONS

6.1 Overview

This section lists the different instructions added by the LEON-REX extension and their opcode representation. The instructions are arranged by mode (regular or REX) and size, and then subdivided by category.

The suggested assembly language syntax uses a prefix of "r" on all mnemonics to distinguish them from their regular SPARC counterpart. Another possibility could be to have an assembler directive such as ".rex" to indicate that the following code should be represented with REX opcodes and then use the ordinary SPARC syntax.

6.2 Regular SPARC mode instructions

6.2.1 SAVEREX and ADDREX

Opcode	op3	Description
SAVEREX	111100	Save caller's window and enter REX mode
ADDREX	000000	Add and enter REX mode

Format:

10	rd	op3	rs1	imm=0	simm13(12)=1	Simm13(11:0)
31	*29*	*24*	*18*	*13*	*12*	*11* *0*

Suggested assembly language syntax:

saverex reg_{rs1}, imm, reg_{rd}
addrex reg_{rs1}, imm, reg_{rd}

Description:

Same behavior as SAVE and ADD instructions but with the added side effect of entering REX mode. The most significant bit of the immediate must be 1, in other words the immediate must be negative.

Traps:

window_overflow (SAVE only)

6.3 REX instructions, 16-bit

6.3.1 Branch on Integer Condition Codes, Short

Format:

00	cond	bsz=0	btype=0	disp8
15	*13*	*9*	*8*	*7* *0*

Suggested assembly language syntax:

```
rba     label
rbn     label
rbne    label
rbe     label
rbg     label
rble    label
rbge    label
rbl     label
rbgu    label
rbleu   label
rbcc    label
rbcs    label
rbpos   label
rbneg   label
rbvc    label
rbvs    label
```

Description:

Branch to “PC + (2 x sign_ext(disp8))” if the condition is met. Value of cond has same format as for SPARC V8 Bicc instructions.

Traps:

(none)

6.3.2 Branch on Floating-Point Condition Codes, Short

Format:

00	cond	bsz=0	btype=1	disp8
15	*13*	*9*	*8*	*7* *0*

Suggested assembly language syntax:

rfba *label*
rfbn *label*
rfbu *label*
rfbg *label*
rfbug *label*
rfbl *label*
rfbul *label*
rfblg *label*
rfbne *label*
rfbe *label*
rfbue *label*
rfbge *label*
rfbuge *label*
rfble *label*
rfbule *label*
rfbo *label*

Description:

Branch to “PC + (2 x sign_ext(disp8))” if the floating-point condition is met. Value of cond has same format as for SPARC V8 FBfcc instructions.

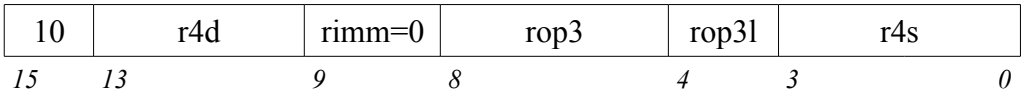
Traps:

fp_disabled
fp_exception

6.3.3 Arithmetic operations, accumulator with register

Opcode	rop3(r)	rop3l	Description
Radd	0000	0	Add
Raddcc	0000	1	Add and modify icc
Rsub	0001	0	Subtract
Rsubcc	0001	1	Subtract and modify icc
Rand	0010	0	And
Randcc	0010	1	And and modify icc
Randn	1010	0	And-not
Randncc	1010	1	And-not and modify icc
Ror	0100	0	Or
Rorcc	0100	1	Or and modify icc
Rorn	1100	0	Or-not
Rorncc	1100	1	Or-not and modify icc
Rxor	0110	0	Exclusive-or
Rxorcc	0110	1	Exclusive-or and modify icc
Rsll	1011	0	Shift left logical
Rsrl	1101	0	Shift right logical

Format:



Suggested assembly language syntax:

radd reg_{rs}, reg_{rd}
raddcc reg_{rs}, reg_{rd}
rsub reg_{rs}, reg_{rd}
rsubcc reg_{rs}, reg_{rd}
rand reg_{rs}, reg_{rd}
randcc reg_{rs}, reg_{rd}
randn reg_{rs}, reg_{rd}
randncc reg_{rs}, reg_{rd}
ror reg_{rs}, reg_{rd}

rorcc	reg_{rs}, reg_{rd}
rorn	reg_{rs}, reg_{rd}
rorncc	reg_{rs}, reg_{rd}
rxor	reg_{rs}, reg_{rd}
rxorcc	reg_{rs}, reg_{rd}
rsll	reg_{rs}, reg_{rd}
rsrl	reg_{rs}, reg_{rd}

Description:

Perform arithmetic/logical operation on %rd and %rs, then store the result in %rd. Since %rd is both source and destination, it acts as an accumulator in these instructions. %rd is used as %rs1 and %rs is used as %rs2 in the equivalent SPARC instructions, so for example “rsub %r1, %r2” is equivalent to “sub %r2, %r1, %r2”.

Traps:

(none)

6.3.4 Arithmetic operations, accumulator with immediate

Opcode	rop3(i)	Description
Raddcc	0000	Add and modify icc
Rsll	1011	Shift left logical
Rsrl	1101	Shift right logical

Format:

10	r4d	rimm=1	rop3	simm5 / imm5
15	*13*	*9*	*8*	*4* *0*

Suggested assembly language syntax:

raddcc imm, reg_{rd}
rsll imm, reg_{rd}
rsrl imm, reg_{rd}

Description:

Raddcc adds the simm5 sign-extended 5-bit constant to the register %rd and updates the condition codes.

SLL/SRL performs logical left/right shift of the destination register by the number of bits indicated in the constant. In this case the constant is treated as an unsigned number.

Traps:

(none)

6.3.5 Comparison, with register

Opcode	rop3(r)	rop3l	Description
Rcmp	1011	1	Compare registers and modify icc

Format:

10	r4s1	rimm=0	rop3	rop3l	r4s2
15	*13*	*9*	*8*	*4*	*3* *0*

Suggested assembly language syntax:

rcmp reg_{rs1}, reg_{rs2}

Description:

Perform the equivalent of "subcc %rs1, %rs2, %g0"

Traps:

(none)

6.3.6 Comparison, with immediate

Opcode	rop3(i)	Description
Rcmp	1000	Compare and modify icc

Format:

10	r4s1	rimm=1	rop3	simm5
15	*13*	*9*	*8*	*4* *0*

Suggested assembly language syntax:

rcmp reg_{rs1}, imm

Description:

Perfoms the equivalent of "subcc %rs1, imm, %g0". The 5-bit constant is sign-extended.

Traps:

(none)

NOTE: rop3 field value is different for immediate and register version of rcmp opcode.

6.3.7 Constant assignment

Opcode	rop3(i)	Description
Rset5	0001	Assign constant value to register
Rone	0101	Create value with one bit set and assign to register

Format:

10	r4d	rimm=1	rop3	simm5 / imm5
15	*13*	*9*	*8*	*4* *0*

Suggested assembly language syntax:

rset5 imm, reg_{rd}
rone imm, reg_{rd}

Description:

RSET5 sets the destination register to the 5-bit sign-extended constant value.
RONE sets the destination register to (1 << imm5), a mask with one bit set. The value in this case is not sign-extended.

Traps:

(none)

6.3.8 Bit-mask operations

Opcode	rop3(i)	Description
Rsetbit	0100	Modify register to set selected bit
Rclrbit	1010	Modify register to clear selected bit
Rinvbit	0110	Modify register to invert selected bit
Rtstbit	0011	Test selected bit in register and modify icc
Rmasklo	0010	Mask away upper bits in register

Format:

10	r4d	rimm=1	rop3	imm5
15	*13*	*9*	*8*	*4* *0*

Suggested assembly language syntax:

rsetbit imm, reg_{rd}
rclrbit imm, reg_{rd}
rinvbit imm, reg_{rd}
rtstbit imm, reg_{rd}
rmasklo imm, reg_{rd}

Description:

RSETBIT, RCLRBIT and RINVBIT modifies the selected bit in a register without affecting the remaining bits. The bit is provided as an unsigned 5-bit constant.

RTSTBIT updates the icc flags depending on whether the selected bit is set or not. The Z flag is set if the bit was zero and not set if the bit was one. Other bits in icc are undefined (implementation dependent).

RMASKLO keeps the bits up to and including the bit selected by the constant, and remaining bits get cleared. For example, if the immedate is 3, only the lowest four bits (bits 3:0) are kept and the remaining 28 bits (31:4) are cleared. If the immediate is 31 all bits are kept and the opcode has no effect.

Traps:

(none)

6.3.9 Register to register copy

Opcode	rop3(r)	rop3l	Description
Rmov	100x	x	Copy data from register to register

Format:

10	r4d	rimm=0	Rop3(3:1)	rdalt	rsalt	r4s
15	*13*	*9*	*8*	*5*	*4*	*3* *0*

Suggested assembly language syntax:

rmov reg_{rs}, reg_{rd}

Description:

Copies the contents of one register into another register. This opcode can access any register in the register window, the rdalt bit selects whether the r4d field refers to 0) the reduced or 1) the complementary reduced register set, and rsalt in the same way selects for the r4s field.

Traps:

(none)

6.3.10 Negation

Opcode	rop3(i)	rop4	Description
Rneg	1110	00100	Negate register
Rnot	1110	00101	Invert register

Format:

10	r4d	rimm=1	rop3	rop4
15	*13*	*9*	*8*	*4* *0*

Suggested assembly language syntax:

rneg reg_{rd}
rnot reg_{rd}

Description:

The RNEG and RNOT instructions perform simple 2-complements negation or bit-wise inversion of the selected register.

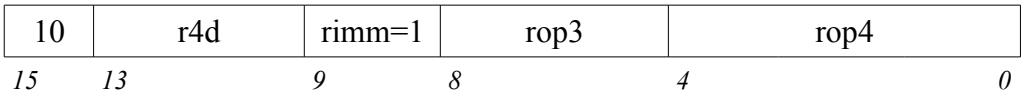
Traps:

(none)

6.3.11 Return instructions

Opcode	rop3(i)	rop4	Description
Rretrest	1110	00000	Return to caller and restore
Rretl	1110	00001	Return from leaf function

Format:



Suggested assembly language syntax:

rretrest
rretl

Description:

The RRETREST instruction performs the “jmpl %i7+8, %g0; restore” sequence normally used in the SPARC ABI when returning from a subroutine to the calling procedure. It is possible to get a window underflow trap if the window restore results in pointing to an invalid window.

The RRETL instruction performs the “jmpl %o7+8; nop” sequence normally used in the SPARC ABI when returning from a leaf function.

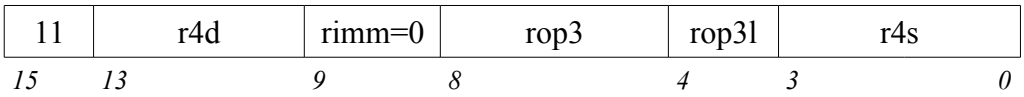
Traps:

window_underflow (RRETREST only)
mem_address_not_aligned (due to the jmpl if i7/o7 are not aligned correctly, see section 4.4)

6.3.12 Load/Store, 8/16/32/64 bits, one register

Opcode	rop3(r)	rop3l	Description
Rld	0000	0	Load 32-bit value
Rldub	0010	0	Load 8-bit unsigned value
Rlduh	0100	0	Load 16-bit unsigned value
Rldd	0110	0	Load 64-bit value
Rldf	0001	0	Load 32-bit floating-point value
Rlddf	0101	0	Load 64-bit floating-point value
Rst	1000	0	Store 32-bit value
Rstb	1010	0	Store 8-bit unsigned value
Rsth	1100	0	Store 16-bit unsigned value
Rstd	1110	0	Store 64-bit value
Rstf	1001	0	Store 32-bit floating-point value
Rstdf	1101	0	Store 64-bit floating-point value

Format:



Suggested assembly language syntax:

rld	[reg_{rs}], reg_{rd}
rldub	[reg_{rs}], reg_{rd}
rlduh	[reg_{rs}], reg_{rd}
rldd	[reg_{rs}], reg_{rd}
rldf	[reg_{rs}], reg_{rd}
rlddf	[reg_{rs}], reg_{rd}
rst	reg_{rd}, [reg_{rs}]
rstb	reg_{rd}, [reg_{rs}]
rsth	reg_{rd}, [reg_{rs}]
rstd	reg_{rd}, [reg_{rs}]
rstf	reg_{rd}, [reg_{rs}]
rstdf	reg_{rd}, [reg_{rs}]

Description:

Stores or load data to the address given in the %rs register and the transferred data source or destination in the %rd register.

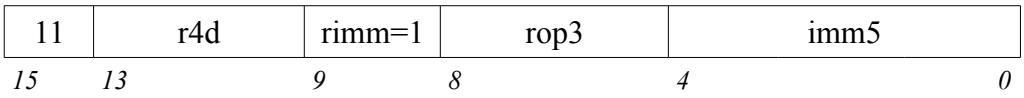
Traps:

illegal_instruction (rldd/rstd with odd rd)
mem_address_not_aligned
data_access_exception
data_access_error

6.3.13 Load/Store, 32 bits, fixed register plus immediate

Opcode	rop3(i)	Description
Rldi0	0100	Load 32-bit value from address relative to i0
Rldo0	0110	Load 32-bit value from address relative to o0
Rldfp	0000	Load 32-bit value from address relative to fp
Rldsp	0010	Load 32-bit value from address relative to sp
Rldfi0	0101	Load 32-bit floating-point value from address relative to i0
Rldffp	0001	Load 32-bit floating-point value from address relative to fp
Rldfsp	0011	Load 32-bit floating-point value from address relative to sp
Rsti0	1100	Store 32-bit value to address relative to i0
Rsto0	1110	Store 32-bit value to address relative to o0
Rstfp	1000	Store 32-bit value to address relative to fp
Rstsp	1010	Store 32-bit value to address relative to sp
Rstfi0	1101	Store 32-bit floating-point value to address relative to i0
Rstffp	1001	Store 32-bit floating-point value to address relative to fp
Rstfsp	1011	Store 32-bit floating-point value to address relative to sp

Format:



Suggested assembly language syntax:

```
rld     [%i0+imm], reg_rd
rld     [%o0+imm], reg_rd
rld     [%fp+imm], reg_rd
rld     [%sp+imm], reg_rd
rldf    [%i0+imm], reg_rd
rldf    [%fp+imm], reg_rd
rldf    [%sp+imm], reg_rd
rst     reg_rd, [%i0+imm]
rst     reg_rd, [%o0+imm]
rst     reg_rd, [%fp+imm]
rst     reg_rd, [%sp+imm]
rstf    reg_rd, [%i0+imm]
rstf    reg_rd, [%fp+imm]
rstf    reg_rd, [%sp+imm]
```

Description:

Stores or load data to the address given by adding the fixed register selected by the opcode and 4 x imm. The immediate is treated as an unsigned 5-bit offset.

Note: This is intended to support several common scenarios, for example accessing elements in a struct passed as the first argument of a subroutine and loading/storing temporary values from the stack.

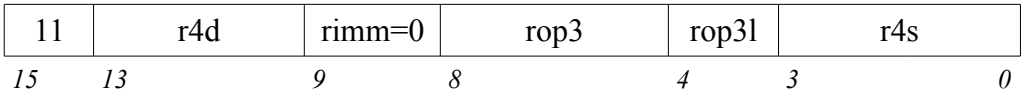
Traps:

mem_address_not_aligned
data_access_exception
data_access_error

6.3.14 Load/Store, 8/16/32/32-bits, one register, auto-incrementing

Opcode	rop3(r)	rop3l	Description
Rldinc	0000	1	Load 32-bit value and increment address register by 4
Rldubinc	0010	1	Load 8-bit unsigned value and increment address register by 1
Rlduhinc	0100	1	Load 16-bit unsigned value and increment address register by 2
Rlddinc	0110	1	Load 64-bit value and increment address register by 8
Rldfinc	0001	1	Load 32-bit floating-point value and increment address register by 4
Rlddfinc	0101	1	Load 64-bit floating-point value and increment address register by 8
Rstinc	1000	1	Store 32-bit value and increment address register by 4
Rstbinc	1010	1	Store 8-bit unsigned value and increment address register by 1
Rsthinc	1100	1	Store 16-bit unsigned value and increment address register by 2
Rstdinc	1110	1	Store 64-bit value and increment address register by 8
Rstfinc	1001	1	Store 32-bit floating-point value and increment address register by 4
Rstdfinc	1101	1	Store 64-bit floating-point value

Format:



Suggested assembly language syntax:

rldinc [reg_{rs}], reg_{rd}
rldubinc [reg_{rs}], reg_{rd}
rlduhinc [reg_{rs}], reg_{rd}
rlddinc [reg_{rs}], reg_{rd}
rldfinc [reg_{rs}], reg_{rd}
rlddfinc [reg_{rs}], reg_{rd}
rstinc reg_{rd}, [reg_{rs}]
rstbinc reg_{rd}, [reg_{rs}]
rsthinc reg_{rd}, [reg_{rs}]
rstdinc reg_{rd}, [reg_{rs}]
rstfinc reg_{rd}, [reg_{rs}]

rstdfinc reg_{rd}, [reg_{rs}]

Description:

Just as the regular non-incrementing load/store, the instruction stores or loads data to the address given in the %rs register and the transferred data source or destination in the %rd register. As a side effect, the register holding the address is incremented by the size of the data element loaded or stored. This instruction is equivalent to first performing a regular load/store and then an addition, for example “lduh [%rs], %rd; add %rs, 2, %rs”, except that you can not get an interrupt between the load/store and the add.

Note that the floating-point load/store instructions also use the reduced 4-bit register format and the register field is interpreted in the same way (for example if the field would decode to integer register %r5, for the floating-point load/store it decodes to %f5).

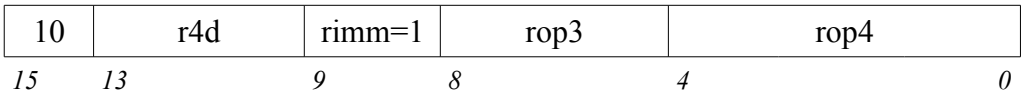
Traps:

illegal_instruction (rlddinc/rstdinc with odd rd)
mem_address_not_aligned
data_access_exception
data_access_error

6.3.15 Miscellaneous operations, no source operands

Opcode	rop3(i)	rop4	Description
Rpush	1110	00010	Store value to sp and decrement sp
Rpop	1110	00011	Load value from sp and increment sp
Rta	1110	00110	Software trap
Rleave	1110	00111	Leave REX mode
Rgetpc	1110	01001	Get current value of program counter

Format:



Suggested assembly language syntax:

rpush reg_{rd}
rpop reg_{rd}
rta software_trap#
rleave
rgetpc reg_{rd}

Description:

The RPUSH and RPOP intstructions are included to support implementing alternative calling conventions. They load/store the value pointed to by %sp and then increment/decrements %sp.

The RTA instruction can be used to generate software trap 0-7.

The RLEAVE instruction can be used to immediately leave REX mode for the next instruction. However note that normally RRETREST/RRETL should be used.

The RGETPC instruction stores the current PC (in other words, the address of the GETPC opcode itself) into the destination register.

Traps:

mem_address_not_aligned (RPUSH/RPOP only)
data_access_exception (RPUSH/RPOP only)
data_access_error (RPUSH/RPOP only)
trap_instruction (RTA only)

6.4 REX instructions, 32-bit

6.4.1 Branch on integer condition codes, long

Format:

00	cond	bsz=1	btype=0	disp24(7:0)	disp24(23:16)
31	*29*	*25*	*24*	*23*	*15* *0*

Suggested assembly language syntax:

```
rba,l    label
rbn,l    label
rbne,l   label
rbe,l    label
rbg,l    label
rble,l   label
rbge,l   label
rbl,l    label
rbgu,l   label
rbleu,l  label
rbcc,l   label
rbcs,l   label
rbpos,l  label
rbneg,l  label
rbvc,l   label
rbvs,l   label
```

Description:

Branch to “PC + (2 x sign_ext(disp24))” if the condition is met. Value of cond has same format as for SPARC V8 Bicc instructions.

Traps:

(none)

6.4.2 Branch on floating-point condition codes, long

Format:

00	cond	bsz=1	btype=1	disp24(7:0)	disp24(23:16)
31	*29*	*25*	*24*	*23*	*15* *0*

Suggested assembly language syntax:

rfba,l *label*
rfbn,l *label*
rfbu,l *label*
rfbg,l *label*
rfbug,l *label*
rfbl,l *label*
rfbul,l *label*
rfblg,l *label*
rfbne,l *label*
rfbe,l *label*
rfbue,l *label*
rfbge,l *label*
rfbuge,l *label*
rfble,l *label*
rfbule,l *label*
rfbo,l *label*

Description:

Branch to “PC + (2 x sign_ext(disp24))” if the floating-point condition is met. Value of cond has same format as for SPARC V8 FBfcc instructions.

Traps:

fp_disabled
fp_exception

6.4.3 Call and Link

Format:

01	disp30

31 *29* *0*

Suggested assembly language syntax:

call *label*

Description:

Performs an unconditional PC-relative control transfer to address “PC + (4 x disp30)” and at the same time leaves REX mode. The next instruction executed after the CALL will be the regular instruction located at the target address with the nPC pointing to the instruction following it. If the CALL instruction is not on aligned on a word boundary, the target address will be rounded down to the nearest lower word address (the two lowest bits of the computed target address become zeroed).

The CALL instruction stores the value of (PC-3) into register %o7.

Note:

The value of (PC-3) is chosen so that the regular ret/retl instructions (jmpl %i7/%o7+8) will return to the instruction following the CALL and re-enable REX mode.

Traps:

(none)

6.4.4 Constant assignment

Opcode	rop3(i)	Description
Rset21	0111	Assign constant value to register

Format:

10	r4d	rimm=1	rop3	simm21(4:0)	simm21(20:5)
31	*29*	*25*	*24*	*20*	*15* *0*

Suggested assembly language syntax:

rset21 imm, reg_{rd}

Description:

RSET21 sets the destination register to the 21-bit sign-extended constant value.

Traps:

(none)

6.4.5 Generic format 3 SPARC operation

Opcode	rop3(r)	rop3l	Description
Rgop	0111	0	Generic SPARC integer operation

Format:

1x	r4d	rimm =0	rop3	rop3l	r4s1	ximm =0	xop3	rdalt	rs1alt	(zero)	rs2
31	*29*	*25*	*24*	*20*	*19*	*15*	*14*	*8*	*7*	*6*	*4* *0*

1x	r4d	rimm =0	rop3	rop3l	r4s1	ximm =1	xop3	rdalt	rs1alt	simm7
31	*29*	*25*	*24*	*20*	*19*	*15*	*14*	*8*	*7*	*6* *0*

Suggested assembly language syntax:

rgop sparc_insn, reg_{rs1}, reg_or_imm, reg_{rd}

Description:

This opcode allows any regular SPARC “format 3” (memory, arithmetic, logical, shift, and remaining) to be represented also with 32 bits in the REX encoding, with the restriction that SPARC opcode bits 12:7 can not be controlled. This limits immediate constants to 7 bits instead of the full 13.

The 32-bit SPARC opcode is composed based on the rgop opcode as below:
Bits 31:30 (op type) are copied over directly from the same position.
Bits 29:25 (rd) is taken from the r4d / rdalt fields (converted from reduced/alternate format)
Bits 24:19 (op3) is taken from xop3 field
Bits 18:14 (rs1) is taken from the r4s1 / rs1alt fields (converted from reduced/alternate format)
Bit 13 (imm) is taken from the ximm bit
Bit 12:7 (simm13 high bits) is set to same value as the highest bit of simm7 if ximm=1
Bit 12:7 (unused) is always set to 0 if ximm=0
Bit 6:0 (simm13 low bits or unused+rs1) is copied over directly from the same position.

Traps:

any trap the corresponding SPARC opcode can generate.

6.4.6 Floating-point operations

Opcode	rop3(r)	rop3l	rfpop	Description
FiTos	0111	1	1100100	
FiTod	0111	1	1101000	
FiToq	0111	1	1101100	
FsToi	0111	1	1010001	
FdToi	0111	1	1010010	
FqToi	0111	1	1010011	
FMOVs	0111	1	0000001	
FNEGs	0111	1	0000101	
FABSs	0111	1	0001001	
FSQRTs	0111	1	0011001	
FSQRTd	0111	1	0011010	
FSQRTq	0111	1	0011011	
FADDs	0111	1	0100001	
FADDd	0111	1	0100010	
FADDq	0111	1	0100011	
FSUBs	0111	1	0100101	
FSUBd	0111	1	0100110	
FSUBq	0111	1	0100111	
FMULs	0111	1	0101001	
FMULd	0111	1	0101010	
FMULq	0111	1	0101011	
FSMULD	0111	1	0111001	
FSMULQ	0111	1	0111010	
FDIVs	0111	1	0101101	
FDIVd	0111	1	0101110	
FDIVq	0111	1	0101111	
FCMPs	0111	1	1000001	
FCMPd	0111	1	1000010	
FCMPq	0111	1	1000011	

Opcode	rop3(r)	rop3l	rfpop	Description
FCMPEs	0111	1	1000101	
FCMPEd	0111	1	1000110	
FCMPEq	0111	1	1000111	

Format:

10	r4d	rimm =0	rop3	rop3l	r4s1	xfpop	rdalt	rs1alt	(zero)	rs2
31	*29*	*25*	*24*	*20*	*19*	*15*	*8*	*7*	*6*	*4* *0*

Suggested assembly language syntax:

rflop sparc_insn, reg_{rs1}, reg_or_imm, reg_{rd}

Description:

This opcode allows any of the floating-point operations (FPOps) defined in the SPARC standard to be represented also with 32 bits in the REX encoding, without any additional restrictions. The same restrictions as would have applied in uncompressed code on the same system should be expected.

The %rd and %rs1 registers are represented using the 4+1-bit reduced/alternate representation, but %rs2 is represented directly as the SPARC register number.

Traps:

any trap the corresponding SPARC opcode can generate.

6.5 REX instructions, 48-bit

6.5.1 Set 32-bit constant

Opcode	rop3(i)	rop4	Description
Rset32	1111	01000	Set 32-bit constant
Rset32pc	1111	01001	Set 32-bit constant, plus PC

Format:

Suggested assembly language syntax:

rset32 imm, reg_{rd}
rset32pc imm, reg_{rd}

Description:

RSET32 assigns a 32-bit constant into the destination register. RSET32PC also adds the address of the instruction to the immediate to allow relative address calculations.

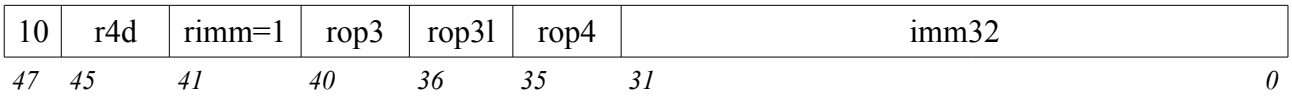
Traps:

(none)

6.5.2 Load from 32-bit address

Opcode	rop3(i)	rop4	Description
Rld32	1111	01010	Load from 32-bit absolute address
Rld32pc	1111	01011	Load from 32-bit relative address

Format:



Suggested assembly language syntax:

rld32 [imm], reg_{rd}
rld32pc [imm], reg_{rd}

Description:

RLD32 loads a 32-bit word from the address given by the immediate into the destination register. RSET32PC also adds the address of the instruction to the immediate to allow relative addresses.

NOTE: These instructions have the op field assigned as 10 despite being load instructions.

Traps:

(none)