Selected SPARC Instruction Set

This document describes a useful subset of SPARC (version 8 or 9) instructions; it is by no means complete. Reference to the SPARC Assembly Reference Manual (available from the course web page) is also important; in particular the list of assembler pseudo-operators in Appendix A. Instruction syntax conforms to the notation used in that manual. Only actual instructions are shown here; see also the list of synthetic instructions in manual section 5.5

Many of the following instructions take a *reg* as first operand and either a *reg* or a 13 bit signed immediate value (*simm13*) as second operand. The value of the second operand, *op2*, is either the contents of the register or the sign extended value of the immediate value.

Many of these instructions have one form that doesn't affect condition codes and the other (with `cc` appended to it), that does. The SPARC's condition codes are:

N	(last result negative)
Z	(last result 0)
V	(last result overflowed in two's complement)
C	(last result carried)

The codes are tested by the various conditional branch instructions.

Register %g0 (%r0) behaves specially. If it is a source operand, the constant value 0 is read; if it is a destination operand, the data written is discarded.

Arithmetic Instructions

`add{cc}`	reg_{rs1}, *reg_or_immed*, reg_{rd}	*add*
`sub{cc}`	reg_{rs1}, *reg_or_immed*, reg_{rd}	*subtract*
`umul{cc}`	reg_{rs1}, *reg_or_immed*, reg_{rd}	*unsigned multiply*
`smul{cc}`	reg_{rs1}, *reg_or_immed*, reg_{rd}	*signed multiply*
`udiv{cc}`	reg_{rs1}, *reg_or_immed*, reg_{rd}	*unsigned divide*
`sdiv{cc}`	reg_{rs1}, *reg_or_immed*, reg_{rd}	*signed divide*

Logical Instructions

`and{cc}`	reg_{rs1}, *reg_or_immed*, reg_{rd}	*bitwise and*
`andn{cc}`	reg_{rs1}, *reg_or_immed*, reg_{rd}	*bitwise and with NOT(op2)*
`or{cc}`	reg_{rs1}, *reg_or_immed*, reg_{rd}	*bitwise or*
`orn{cc}`	reg_{rs1}, *reg_or_immed*, reg_{rd}	*bitwise or with NOT(op2)*
`xor{cc}`	reg_{rs1}, *reg_or_immed*, reg_{rd}	*bitwise xor*
`xnor{cc}`	reg_{rs1}, *reg_or_immed*, reg_{rd}	*bitwise xor with NOT(op2)*

Shifts

`sll`	*reg_{rs1}, reg_or_immed, reg_{rd}*	*shift left by op2*
`srl`	*reg_{rs1}, reg_or_immed, reg_{rd}*	*shift right by op2; zero fill*
`sra`	*reg_{rs1}, reg_or_immed, reg_{rd}*	*shift right by op2; sign extend*

Only the five low-order bits of the shift count (*op2*) matter.

Miscellaneous

`sethi` *const22, reg_{rd}*
`sethi` `%hi` *(value), reg_{rd}*
Zero low-order 10 bits of reg_{rd} and set high-order 22 bits to *const22*. The `%hi` pseudo-op can be used to extract and right-shift the high-order 22 bits of a literal value.
`nop`
No operation.

Control

`save` *reg_{rs1}, reg_or_immed, reg_{rd}*
`restore` *reg_{rs1}, reg_or_immed, reg_{rd}*
Adjust register window as described in class notes. Otherwise, instructions act like `add`, except that source operands are read from old window, and result is written into target register in new window.
`call` *label*
Write `%pc` to `%o7` and perform a delayed jump to specified label. Any address in whole 32-bit space is legal.
`jmpl` *address, reg_{rd}*
Write `%pc` to *rd* and perform a delayed jump to specified address,

Branch Instructions

`ba{,a}`	*label*	*branch always*
`bn{,a}`	*label*	*branch never*
`bne{,a}`	*label*	*branch on not equal*
`be{,a}`	*label*	*branch on equal*
`bg{,a}`	*label*	*branch on greater*
`ble{,a}`	*label*	*branch on less or equal*
`bge{,a}`	*label*	*branch on greater or equal*
`bl{,a}`	*label*	*branch on less*
`bgu{,a}`	*label*	*branch on greater unsigned*
`bleu{,a}`	*label*	*branch on less or equal unsigned*
`bcc{,a}`	*label*	*branch on carry clear (greater or equal unsigned)*
`bcs{,a}`	*label*	*branch on carry set (less unsigned)*
`bpos{,a}`	*label*	*branch on positive*
`bneg{,a}`	*label*	*branch on negative*
`bvc{,a}`	*label*	*branch on overflow clear*
`bvs{,a}`	*label*	*branch on overflow set*

If condition is met (according to current condition codes), perform PC-relative, delayed branch to label, which must be expressible as $\texttt{PC} + 4 * \mathrm{sign\_ext}(\mathit{disp22})$. Appending `,a` sets the "annul" bit for these instructions, which has this effect: if a conditional branch is executed and the branch is not taken, or if a `ba` or `bn` is executed, the delay slot instruction is annulled (not executed).

Load and Store Instructions

`ldsb`	*[address], reg_{rd}*	*load signed byte*
`ldsh`	*[address], reg_{rd}*	*load signed halfword (2 bytes)*
`ldub`	*[address], reg_{rd}*	*load unsigned byte*
`lduh`	*[address], reg_{rd}*	*load unsigned halfword*
`ld`	*[address], reg_{rd}*	*load word (4 bytes)*
`ldd`	*[address], reg_{rd}*	*load double word (8 bytes)*
`stb`	*reg_{rd}, [address]*	*store byte*
`sth`	*reg_{rd}, [address]*	*store halfword*
`st`	*reg_{rd}, [address]*	*store word*
`std`	*reg_{rd}, [address]*	*store double word*

All addresses must be aligned (i.e., halfword addresses must be divisible by 2, word addresses by 4, and double-word address by 8. Unsigned loads zero-fill high-order bits; signed loads sign-extend. Register numbers for double word instructions must be even, and two registers are read/written.

Floating Point Operations

This lists only the double-precision (8 byte) operations; there are also single and quad precision operations. Double-precision operators act on pairs of floating registers, specified by the (lower) even-numbered register. Loads and stores of doubles must be to 8-byte aligned memory addresses. Note that there is no way to move a value directly between integer and float registers; it must transmitted through memory.

`faddd`	$freg_{rs1}$, $freg_{rs2}$,$freg_{rd}$	*add double*
`fsubd`	$freg_{rs1}$, $freg_{rs2}$,$freg_{rd}$	*subtract double*
`fmuld`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	*multiply double*
`fdivd`	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$	*divide double*
`fmovd`	$fregrs$, $freg_{rd}$	*move*
`fnegd`	$fregrs$, $freg_{rd}$	*negate*
`fabsd`	$fregrs$, $freg_{rd}$	*absolute value*
`fitod`	$fregrs$, $freg_{rd}$	*convert integer to double*
`fdtoi`	$fregrs$, $freg_{rd}$	*convert double to integer*
`std`	$freg_{rd}$, *[address]*	*store double*
`ldd`	*[address]*, $freg_{rd}$	*load double*
`fcmpd`	$freg_{rs1}$, $freg_{rs2}$	*compare double*
`fba{,a}`	*label*	*branch always*
`fbn{,a}`	*label*	*branch never*
`fbu{,a}`	*label*	*branch on unordered*
`fbg{,a}`	*label*	*branch on greater*
`fbug{,a}`	*label*	*branch on unordered or greater*
`fbl{,a}`	*label*	*branch on less*
`fbul{,a}`	*label*	*branch on unordered or less*
`fblg{,a}`	*label*	*branch on less or greater*
`fbne{,a}`	*label*	*branch on not equal*
`fbe{,a}`	*label*	*branch on equal*
`fbue{,a}`	*label*	*branch on unordered or equal*
`fbge{,a}`	*label*	*branch on greater or equal*
`fbuge{,a}`	*label*	*branch on unordered or greater or equal*
`fble{,a}`	*label*	*branch on less or equal*
`fbule{,a}`	*label*	*branch on unordered or less or equal*
`fbo{,a}`	*label*	*branch on ordered*

Floating point comparisons are made explicitly using `fcmpd`, which sets the floating point condition codes; the results are then tested by the floating conditional branches. A comparison returns "unordered" if one or both operands is NaN ("not a number"). The annul bit operates the same way as for integer branches.