Selected SPARC Instruction Set

This document describes a useful subset of SPARC (version 8 or 9) instructions; it is by no means complete. The SPARC Architecture Manual and your assembler reference manual (both available from the course web page) are also important; in particular you’ll need the list of assembler pseudo-operators. Instruction syntax conforms to the notation used in the architecture manual. Only actual instructions are shown here; see also the list of synthetic instructions in Appendix A.3 of the architecture manual.

Many of the following instructions take a \texttt{reg} as first operand and either a \texttt{reg} or a 13 bit signed immediate value (\texttt{simm13}) as second operand. The value of the second operand, \texttt{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 \texttt{cc} appended to it), that does. The SPARC’s condition codes are:

\begin{itemize}
  \item N (last result negative)
  \item Z (last result 0)
  \item V (last result overflowed in two’s complement)
  \item C (last result carried)
\end{itemize}

The codes are tested by the various conditional branch instructions.

Register \texttt{/%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

\begin{verbatim}
add\{cc\}    regrs1, reg_or_immed, regrd     add
sub\{cc\}    regrs1, reg_or_immed, regrd     subtract
umul\{cc\}   regrs1, reg_or_immed, regrd     unsigned multiply
smul\{cc\}   regrs1, reg_or_immed, regrd     signed multiply
udiv\{cc\}   regrs1, reg_or_immed, regrd     unsigned divide
sdiv\{cc\}   regrs1, reg_or_immed, regrd     signed divide
\end{verbatim}

Logical Instructions

\begin{verbatim}
and\{cc\}    regrs1, reg_or_immed, regrd     bitwise and
andn\{cc\}   regrs1, reg_or_immed, regrd     bitwise and with NOT(op2)
or\{cc\}     regrs1, reg_or_immed, regrd     bitwise or
orn\{cc\}    regrs1, reg_or_immed, regrd     bitwise or with NOT(op2)
xor\{cc\}    regrs1, reg_or_immed, regrd     bitwise xor
xnor\{cc\}   regrs1, reg_or_immed, regrd     bitwise xor with NOT(op2)
\end{verbatim}
Shifts

\[
\begin{align*}
\text{sll} & \quad \text{reg}_{rs1}, \text{reg}_{or\text{-immed}}, \text{reg}_{rd} & \quad \text{shift left by op2} \\
\text{srl} & \quad \text{reg}_{rs1}, \text{reg}_{or\text{-immed}}, \text{reg}_{rd} & \quad \text{shift right by op2; zero fill} \\
\text{sra} & \quad \text{reg}_{rs1}, \text{reg}_{or\text{-immed}}, \text{reg}_{rd} & \quad \text{shift right by op2; sign extend}
\end{align*}
\]

Only the five low-order bits of the shift count (\textit{op2}) matter.

Miscellaneous

\[
\begin{align*}
\text{sethi} & \quad \text{const}^{22}, \text{reg}_{rd} \\
\text{sethi} & \quad \%\text{hi (value)}, \text{reg}_{rd}
\end{align*}
\]

Zero low-order 10 bits of \textit{reg}_{rd} and set high-order 22 bits to \textit{const}^{22}. The \%\textit{hi} pseudo-op can be used to extract and right-shift the high-order 22 bits of a literal value.

\text{nop}

No operation.

Control

\[
\begin{align*}
\text{save} & \quad \text{reg}_{rs1}, \text{reg}_{or\text{-immed}}, \text{reg}_{rd} \\
\text{restore} & \quad \text{reg}_{rs1}, \text{reg}_{or\text{-immed}}, \text{reg}_{rd}
\end{align*}
\]

Adjust register window as described in class notes. Otherwise, instructions act like \textit{add}, except that source operands are read from old window, and result is written into target register in new window.

\text{call} \quad \text{label}

Write \%\textit{pc} to \%\textit{o7} and perform a delayed jump to specified label. Any address in whole 32-bit space is legal.

\text{jmpl} \quad \text{address, reg}_{rd}

Write \%\textit{pc} to \textit{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)
bc{,a}  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 PC + 4 * sign ext(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], regrd      load signed byte
ldsh  [address], regrd      load signed halfword (2 bytes)
ldub  [address], regrd      load unsigned byte
lduh  [address], regrd      load unsigned halfword
ld  [address], regrd        load word (4 bytes)
ldd  [address], regrd       load double word (8 bytes)

stb   regrd, [address]       store byte
sth   regrd, [address]       store halfword
st    regrd, [address]       store word
std   regrd, [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.

```
fadd  fregrs1, fregrs2, fregrd  add double
fsubd fregrs1, fregrs2, fregrd  subtract double
fmuld fregrs1, fregrs2, fregrd  multiply double
fdivd fregrs1, fregrs2, fregrd  divide double
fmovd fregrs, fregrd  move
fnegd fregrs, fregrd  negate
fabsd fregrs, fregrd  absolute value
fitod fregrs, fregrd  convert integer to double
fdivd fregrs, fregrd  convert double to integer
std  fregrd, [address]  store double
ldd  [address], fregrd  load double
fcmpd fregrs1, fregrs2  compare double
fba{} label  branch always
fbn{} label  branch never
fbu{} label  branch on unordered
fbg{} label  branch on greater
fbug{} label  branch on unordered or greater
fbl{} label  branch on less
fbu{} label  branch on unordered or less
fbig{} label  branch on less or greater
fbne{} label  branch on not equal
fbe{} label  branch on equal
fbue{} label  branch on unordered or equal
fbge{} label  branch on greater or equal
fbuge{} label  branch on unordered or greater or equal
fble{} label  branch on less or equal
fbule{} label  branch on unordered or less or equal
fbo{} 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.