SPARC Instruction Types

There are very few addressing modes on the SPARC, and they may be used only in certain very restricted combinations. The three main types of SPARC instructions are given below, along with the valid combinations of addressing modes. There are only a few unusual instructions which do not fall into these catagories.

1. Arithmetic/Logical/Shift instructions

```
opcode reg1,reg2,reg3    !reg1 op reg2    -> reg3
opcode reg1,const13,reg3 !reg1 op const13 -> reg3
```

- All "action" instructions (add, sub, and, or, etc.) take three operands.
- The destination is always the third operand.
- The middle operand may be a 13-bit signed constant (-4096...+4095).
- Otherwise, all operands are registers.
- Examples:

```
add %L1,%L2,%L3 !%L1+%L2->%L3
add %L1,1,%L1   !increment L1
sub %g0,%i3,%i3 !negate i3
sub %L1,10,%G0  !compare %L1 to 10 (discard result)
add %L1,%G0,%L2 !move %L1 to %L2 (add 0 to it)
add %G0,%G0,%L4 !clear L4 (0+0 ->%L4)
```

- To do the above things in the 680x0, 6 different opcodes would be needed (move, add, addi, clr, neg, cmp)

2. Load/Store Instructions

```
opcode [reg1+reg2],reg3
opcode [reg1+const13],reg3
```

- **Only** load and store instructions can access memory.
- The contents of reg3 is read/written from/to the address in memory formed by adding reg1+reg2, or else reg1+const13 (a 13- bit signed constant as above).
- The operands are written in the reverse direction for store instructions, so that the destination is always last.
- One of reg1 or const13 can be omitted. The assembler will supply $g0 or 0. (This is a shorthand provided by the assembler. Both are always there in machine language.)
- Examples:

```
ld [%L1+%L2],%L3 !word at address [%L1+%L2]->%L3
ld [%L1+8],%L2   !word at address [%L1+8]->%L2
ld [%L1],%L2     !word at address [%L1]->%L2
st %g0,[%i2+0]   !0 -> word at address in %i2
st %g0,[%i2]     !same as above
```

3. Branch Instructions

```
opcode address
```

- Branch to (or otherwise use) the address given.
- There are actually 2 types of addresses (see "relocatability" later) - but they look the same.
- Examples:

```
call printf
be   Loop
```

That's it. Period. No other modes or combinations of modes are possible. This is a RISC machine and R stands for "Reduced".

```
add %L1,[%L2],%L3 !Invalid. No memory access allowed.
ld  5,%L4         !Invalid. Must be a memory access.
```

SPARC Fundamental Instructions

Load/Store Instructions

- **Only these instructions access memory.**
- All 32 bits of the register are always affected by a load. If a shorter data item is loaded, it is padded by either adding zeroes (for unsigned data), or by sign extension (for signed data).
- In effect, data in memory may be 1, 2, or 4 bytes long, but data in registers is always 4 bytes long.

```
ld   - load (load a word into a register)
st   - store (store a word into memory)
ldub - load unsigned byte (fetch a byte, pad with 0's)
ldsb - load signed byte (fetch a byte, sign extend it)
lduh - load unsigned halfword (fetch 2 bytes, pad)
ldsh - load signed halfword (fetch 2 bytes, sign extend)
stb  - store byte (store only the LSB)
sth  - store halfword (store only the 2 LSB's)
```

- There are also two instructions for double words. The register number must be even, and 8 bytes are loaded or stored. The MSW goes to the even register and the LSW to the odd register that follows it.

```
ldd - load double (load 2 words into 2 registers)
std - store double (store 2 words from 2 registers)
```

Arithmetic/Logical Instructions

- All 32 bits of every register is used.
- Setting the condition code is always optional. Add "cc" to the opcode to set the condition code. By default, it is **not** set.

```
add  - a+b
sub  - a-b
and  - a&b (bitwise AND)
andn - a&~b (bitwise and - second operand complemented)
or   - a|b (bitwise OR)
orn  - a|~b (bitwise or - second operand complemented)
xor  - a^b (bitwise exclusive or)
xnor - a^~b (bitwise exor - second operand complemented)
```

- Examples:

```
add   %L1,%L2,%L3  ;add %L1+%L2 -> %L3
subcc %L4,10,%G0   ;sub %L4-10, set cc, discard result
or    %o3,0xFF,%o3 ;set lowest 8 bits of %o3 to 1's
xnor  %L6,%G0,%L6  ;complement %L6 (same as NOT in 680x0)
```

Comparison of CISC vs. RISC (680x0 vs. SPARC)

A RISC (Reduced Instruction Set Computer) achieves the same functionality with a much smaller (and more consistent) instruction set. For example, the sets of instructions below can do roughly the same jobs. (Size modifiers like "ub" after "ld" in SPARC and ".w" after "move" in 680x0 are ignored.)

680x0:

```
add,adda,addi,addq,clr,cmp,cmpa,cmpi,cmpm,exg,
ext,extb,move,movea,move16,movem,moveq,neg,negx,
sub,suba,aubi,subq,tst
```

SPARC:

```
ld,st,addcc,subcc
```

How is this reduction made possible?

- Three operands per instruction
- The fact that %g0 gives easy access to a 0 value
- The use of more instructions is expected in some cases. (Example,
  `move (a3)+,-(a4)`
  in 68000 would require
  ```
  ld [%L3],%G1
  add %L3,4,%L3
  sub %L4,4,%L4
  st %G1,[%L4]
  ```
  in SPARC.
- But, in the example above, both machines would likely use about the same number of clock cycles, and the SPARC would have a faster clock because it is simpler, so the SPARC would likely be faster for roughly equivalent technology.

CALL Instruction

This instruction is used to call subprograms. As for the 680x0, we will leave the details for later. For now, it will be used only to call library routines.

```
call printf
```

SETHI Instruction (and the SET synthetic instruction)

This instruction is one of the few that has a slightly different assembly-language format. The syntax looks like this:

```
sethi const22,%reg
```

where "const22" is a 22-bit integer constant (signed or unsigned is not relevant). It places the constant into the high-order 22 bits of the register, and sets the low-order 10 bits of the register to 0's. (?!) For example,

```
sethi 0x333333,%L1; 0x333333 is 1100110011001100110011
```

would set register %L1 to

```
1100110011001100110011 0000000000
```

Q: Why would you want to do this? A: In order to load a 32-bit constant (such as an address) into a register. This can't possibly be done in one instruction (since all instructions are 32 bits long, there isn't room for a 32-bit constant and also an opcode and a register number). There are instructions that can set the lower part of a register (add, or, etc), so this one complements those nicely.

For example, to set %L1 to 0x89ABCDEF, do the following:

1. Split up 0x89ABCDEF into the top 22 bits and the bottom 10 bits

```
89ABCDEF = 10001001101010111100110111101111
Top 22 bits are 1000100110101011110011 = 226AF3
Low 10 bits are 0111101111 = 1EF
```

2. Place the two halves into %L1 using separate instructions:

```
sethi 0x226AF3,%L1
or    %L1,0x1EF,%L1 ;or is better than add. (WHY?)
```

Shortcut #1: The SPARC assembler provides two special "functions" to make this easier. %hi(X) will give the top 22 bits of the constant X and %lo(X) will give the bottom 10 bits. This is an assembler feature. It is not part of the SPARC machine language. So we could use

```
sethi %hi(0x89ABCDEF),%L1
or    %L1,%lo(0x89ABCDEF),%L1
```

The most common use of this instruction is to place the address of something into a register. For example, if there is a character string in memory with the label "Prompt" on it, you can put the address of that string into %o1 using

```
sethi %hi(Prompt),%o1
or    %o1,%lo(Prompt),%o1
```

Shortcut #2: The above pair of instructions is used quite a lot, so the assembler provides a "synthetic instruction" which will generate them for you. The "instruction"

```
set   const32,%reg
```

will accept any 32-bit constant (const32) such as an address, and any register (%reg) and will generate

```
sethi %hi(const32),%reg
or    %reg,%lo(const32),%reg
```

However, it should be remembered that **SET is not a real SPARC instruction**, and that **it produces two machine language instructions**, not one.

NOP Instruction (No OPeration)

This is another "synthetic instruction". The syntax looks like this:

```
nop
```

but it really generates the instruction

```
sethi 0,%g0
```

which does **absolutely nothing**. All instruction sets have a NOP instruction. In RISC machines, it is often a very essential instruction.

SPARC Synthetic Instructions

RISC machine languages do not have many instructions that are common in CISC machine languages (move, negate, clear, compare, etc.) because all of these can be done quite easily with 3-operand add, subtract, and logical instructions. However, the assembler provides "synthetic instructions" to improve convenience and readability. These are not real machine language instructions, but the assembler will automatically translate them into the proper instruction(s) for you.

In SPARC, some of the most common synthetic instructions are:

```
Synthetic Instruction   Assembled As
---------------------   -----------------------------
  clr     %reg          or      %g0,%g0,%reg
  cmp     %reg,%reg     subcc   %reg,%reg,%g0
  cmp     %reg,const    subcc   %reg,const,%g0
  mov     %reg,%reg     or      %g0,%reg,%reg
  mov     const,%reg    or      %g0,const,%reg

  set     const,%reg    sethi   %hi(const),%reg
                        or      %reg,%lo(const22),%reg
```

And here are some others that may be useful:

```
Synthetic Instruction   Assembled As
---------------------   -----------------------------
  clr     [address]     st      %g0,[address]
  clrh    [address]     sth     %g0,[address]
  clrb    [address]     stb     %g0,[address]
  dec     %reg          sub     %reg,1,%reg
  deccc   %reg          subcc   %reg,1,%reg
  inc     %reg          add     %reg,1,%reg
  inccc   %reg          addcc   %reg,1,%reg
  not     %reg          xnor    %reg,%g0,%reg
  neg     %reg          sub     %g0,%reg,%reg
  tst     %reg          orcc    %reg,%g0,%g0
```

Here are two that will be used for subprograms later:

```
Synthetic Instruction   Assembled As
---------------------   -----------------------------
  restore               restore %g0,%g0,%g0
  ret                   jmpl    %i7+8,%g0
```