

# Princeton University

## COS 217: Introduction to Programming Systems

### A Subset of SPARC Assembly Language

Abbreviations Used in Instruction Formats	
rs1	Source register 1.
rs2	Source register 2.
rd	Destination register.
exprX	Expression that the assembler (and linker) evaluates to a constX.
labelX	Label that the assembler (and linker) evaluates to a constX instruction displacement.

Abbreviations Used in Instruction Descriptions	
r[X]	The contents of register X. The instruction descriptions view r as an array of ints.
mem[X]	The contents of memory at location X. The instruction descriptions view mem as an array of chars.
constX	Constant that fits into X bits.
Z	Zero condition code. The instruction descriptions view Z as an int whose value is either 0 (FALSE) or 1 (TRUE).
N	Negative condition code. The instruction descriptions view N as an int whose value is either 0 (FALSE) or 1 (TRUE).
V	oVerflow condition code. The instruction descriptions view V as an int whose value is either 0 (FALSE) or 1 (TRUE).
C	Carry condition code. The instruction descriptions view C as an int whose value is either 0 (FALSE) or 1 (TRUE).

Load and Store Mnemonics (Format 3)	
ldub [rs1],rd ldub [rs1+rs2],rd ldub [rs1+expr13],rd ldub [rs1-expr13],rd ldub [expr13+rs1],rd ldub [expr13],rd	<b>Load unsigned byte</b> r[rd] = (unsigned int)mem[r[rs1]]; r[rd] = (unsigned int)mem[r[rs1] + r[rs2]]; r[rd] = (unsigned int)mem[r[rs1] + const13]; r[rd] = (unsigned int)mem[r[rs1] - const13]; r[rd] = (unsigned int)mem[r[rs1] + const13]; r[rd] = (unsigned int)mem[const13];
ldsb [rs1],rd ldsb [rs1+rs2],rd ldsb [rs1+expr13],rd ldsb [rs1-expr13],rd ldsb [expr13+rs1],rd ldsb [expr13],rd	<b>Load signed byte</b> r[rd] = (int)mem[r[rs1]]; r[rd] = (int)mem[r[rs1] + r[rs2]]; r[rd] = (int)mem[r[rs1] + const13]; r[rd] = (int)mem[r[rs1] - const13]; r[rd] = (int)mem[r[rs1] + const13]; r[rd] = (int)mem[const13];
lduh [rs1],rd lduh [rs1+rs2],rd lduh [rs1+expr13],rd lduh [rs1-expr13],rd lduh [expr13+rs1],rd lduh [expr13],rd	<b>Load unsigned halfword</b> r[rd] = *(unsigned short*)(mem + r[rs1]); r[rd] = *(unsigned short*)(mem + r[rs1] + r[rs2]); r[rd] = *(unsigned short*)(mem + r[rs1] + const13); r[rd] = *(unsigned short*)(mem + r[rs1] - const13); r[rd] = *(unsigned short*)(mem + r[rs1] + const13); r[rd] = *(unsigned short*)(mem + const13);
ldsh [rs1],rd ldsh [rs1+rs2],rd ldsh [rs1+expr13],rd ldsh [rs1-expr13],rd ldsh [expr13+rs1],rd ldsh [expr13],rd	<b>Load signed halfword</b> r[rd] = *(short*)(mem + r[rs1]); r[rd] = *(short*)(mem + r[rs1] + r[rs2]); r[rd] = *(short*)(mem + r[rs1] + const13); r[rd] = *(short*)(mem + r[rs1] - const13); r[rd] = *(short*)(mem + r[rs1] + const13); r[rd] = *(short*)(mem + const13);
ld [rs1],rd ld [rs1+rs2],rd ld [rs1+expr13],rd ld [rs1-expr13],rd ld [expr13+rs1],rd ld [expr13],rd	<b>Load word</b> r[rd] = *(int*)(mem + r[rs1]); r[rd] = *(int*)(mem + r[rs1] + r[rs2]); r[rd] = *(int*)(mem + r[rs1] + const13); r[rd] = *(int*)(mem + r[rs1] - const13); r[rd] = *(int*)(mem + r[rs1] + const13); r[rd] = *(int*)(mem + const13);

<pre> ldd [rs1],rd ldd [rs1+rs2],rd ldd [rs1+expr13],rd ldd [rs1-expr13],rd ldd [expr13+rs1],rd ldd [expr13],rd </pre>	<p><b>Load doubleword</b></p> <pre> r[rd] = *(int*)(mem + r[rs1]); r[rd+1] = *(int*)(mem + r[rs1] + 4); r[rd] = *(int*)(mem + r[rs1] + r[rs2]); r[rd+1] = *(int*)(mem + r[rs1] + r[rs2] + 4); r[rd] = *(int*)(mem + r[rs1] + const13); r[rd+1] = *(int*)(mem + r[rs1] + const13 + 4); r[rd] = *(int*)(mem + r[rs1] - const13); r[rd+1] = *(int*)(mem + r[rs1] - const13 + 4); r[rd] = *(int*)(mem + r[rs1] + const13); r[rd+1] = *(int*)(mem + r[rs1] + const13 + 4); r[rd] = *(int*)(mem + const13); r[rd+1] = *(int*)(mem + const13 + 4); </pre>
<pre> swap [rs1],rd swap [rs1+rs2],rd swap [rs1+expr13],rd swap [rs1-expr13],rd swap [expr13+rs1],rd swap [expr13],rd </pre>	<p><b>Swap</b></p> <pre> temp=mem[r[rs1]]; mem[r[rs1]]=r[rd]; r[rd]=temp; temp=mem[r[rs1]+r[rs2]]; mem[r[rs1]+r[rs2]]=r[rd]; r[rd]=temp; temp=mem[r[rs1]+const13]; mem[r[rs1]+const13]=r[rd]; r[rd]=temp; temp=mem[r[rs1]-const13]; mem[r[rs1]-const13]=r[rd]; r[rd]=temp; temp=mem[r[rs1]+const13]; mem[r[rs1]+const13]=r[rd]; r[rd]=temp; temp=mem[expr13]; mem[expr13]=r[rd]; r[rd]=temp; </pre>
<pre> stb rd,[rs1] stb rd,[rs1+rs2] stb rd,[rs1+expr13] stb rd,[rs1-expr13] stb rd,[expr13+rs1] stb rd,[expr13] </pre>	<p><b>Store byte</b></p> <pre> mem[r[rs1]] = r[rd]; mem[r[rs1] + r[rs2]] = r[rd]; mem[r[rs1] + const13] = r[rd]; mem[r[rs1] - const13] = r[rd]; mem[r[rs1] + const13] = r[rd]; mem[const13] = r[rd]; </pre>
<pre> sth rd,[rs1] sth rd,[rs1+rs2] sth rd,[rs1+expr13] sth rd,[rs1-expr13] sth rd,[expr13+rs1] sth rd,[expr13] </pre>	<p><b>Store halfword</b></p> <pre> *(short*)(mem + r[rs1]) = r[rd]; *(short*)(mem + r[rs1] + r[rs2]) = r[rd]; *(short*)(mem + r[rs1] + const13) = r[rd]; *(short*)(mem + r[rs1] - const13) = r[rd]; *(short*)(mem + r[rs1] + const13) = r[rd]; *(short*)(mem + const13) = r[rd]; </pre>
<pre> st rd,[rs1] st rd,[rs1+rs2] st rd,[rs1+expr13] st rd,[rs1-expr13] st rd,[expr13+rs1] st rd,[expr13] </pre>	<p><b>Store word</b></p> <pre> *(int*)(mem + r[rs1]) = r[rd]; *(int*)(mem + r[rs1] + r[rs2]) = r[rd]; *(int*)(mem + r[rs1] + const13) = r[rd]; *(int*)(mem + r[rs1] - const13) = r[rd]; *(int*)(mem + r[rs1] + const13) = r[rd]; *(int*)(mem + const13) = r[rd]; </pre>
<pre> std rd,[rs1] std rd,[rs1+rs2] std rd,[rs1+expr13] std rd,[rs1-expr13] std rd,[expr13+rs1] std rd,[expr13] </pre>	<p><b>Store doubleword</b></p> <pre> *(int*)(mem + r[rs1]) = r[rd]; *(int*)(mem + r[rs1] + 4) = r[rd+1]; *(int*)(mem + r[rs1] + r[rs2]) = r[rd]; *(int*)(mem + r[rs1] + r[rs2] + 4) = r[rd+1]; *(int*)(mem + r[rs1] + const13) = r[rd]; *(int*)(mem + r[rs1] + const13 + 4) = r[rd+1]; *(int*)(mem + r[rs1] - const13) = r[rd]; *(int*)(mem + r[rs1] - const13 + 4) = r[rd+1]; *(int*)(mem + r[rs1] + const13) = r[rd]; *(int*)(mem + r[rs1] + const13 + 4) = r[rd+1]; *(int*)(mem + const13) = r[rd]; *(int*)(mem + const13 + 4) = r[rd+1]; </pre>

### Shift Mnemonics (Format 3)

<pre> sll rs1,rs2,rd sll rs1,expr13,rd </pre>	<p><b>Shift left logical</b></p> <pre> r[rd] = r[rs1] &lt;&lt; r[rs2]; r[rd] = r[rs1] &lt;&lt; const13; </pre>
<pre> srl rs1,rs2,rd srl rs1,expr13,rd </pre>	<p><b>Shift right logical</b></p> <pre> r[rd] = (unsigned int)r[rs1] &gt;&gt; r[rs2]; r[rd] = (unsigned int)r[rs1] &gt;&gt; const13; </pre>
<pre> sra rs1,rs2,rd sra rs1,expr13,rd </pre>	<p><b>Shift right arithmetic</b></p> <pre> r[rd] = r[rs1] &gt;&gt; r[rs2]; r[rd] = r[rs1] &gt;&gt; const13; </pre>

Arithmetic Mnemonics (Format 3)	
add rs1,rs2,rd add rs1,expr13,rd	<b>Add</b> r[rd] = r[rs1] + r[rs2]; r[rd] = r[rs1] + const13;
addcc rs1,rs2,rd  addcc rs1,expr13,rd	<b>Add, and set condition codes</b> r[rd] = r[rs1] + r[rs2]; N = r[rd]<0; Z = r[rd]==0; V = (r[rs1]<0 & r[rs2]<0 & r[rd]>0)   (r[rs1]>0 & r[rs2]>0 & r[rd]<0); C = (r[rs1]<0 & r[rs2]<0)   (r[rd]>0 & (r[rs1]<0   r[rs2]<0)); r[rd] = r[rs1] + const13; N = r[rd]<0; Z = r[rd]==0; V = (r[rs1]<0 & const13<0 & r[rd]>0)   (r[rs1]>0 & const13>0 & r[rd]<0); C = (r[rs1]<0 & const13<0)   (r[rd]>0 & (r[rs1]<0   const13<0));
addx rs1,rs2,rd addx rs1,expr13,rd	<b>Add extended</b> r[rd] = r[rs1] + r[rs2] + C; r[rd] = r[rs1] + const13 + C;
addxcc rs1,rs2,rd  addxcc rs1,expr13,rd	<b>Add extended, and set condition codes</b> r[rd] = r[rs1] + r[rs2] + C; N = r[rd]<0; Z = r[rd]==0; V = (r[rs1]<0 & r[rs2]<0 & r[rd]>0)   (r[rs1]>0 & r[rs2]>0 & r[rd]<0); C = (r[rs1]<0 & r[rs2]<0)   (r[rd]>0 & (r[rs1]<0   r[rs2]<0)); r[rd] = r[rs1] + const13 + C; N = r[rd]<0; Z = r[rd]==0; V = (r[rs1]<0 & const13<0 & r[rd]>0)   (r[rs1]>0 & const13>0 & r[rd]<0); C = (r[rs1]<0 & const13<0)   (r[rd]>0 & (r[rs1]<0   const13<0));
sub rs1,rs2,rd sub rs1,expr13,rd	<b>Subtract</b> r[rd] = r[rs1] - r[rs2]; r[rd] = r[rs1] - const13;
subcc rs1,rs2,rd  subcc rs,expr13,rd	<b>Subtract, and set condition codes</b> r[rd] = r[rs1] - r[rs2]; N = r[rd]<0; Z = r[rd]==0; V = (r[rs1]<0 & r[rs2]>0 & r[rd]>0)   (r[rs1]>0 & r[rs2]<0 & r[rd]<0); C = (r[rs1]>0 & r[rs2]<0)   (r[rd]<0 & (r[rs1]>0   r[rs2]<0)); r[rd] = r[rs1] - const13; N = r[rd]<0; Z = r[rd]==0; V = (r[rs1]<0 & const13>0 & r[rd]>0)   (r[rs1]>0 & const13<0 & r[rd]<0); C = (r[rs1]>0 & const13<0)   (r[rd]<0 & (r[rs1]>0   const13<0));
subx rs1,rs2,rd subx rs1,expr13,rd	<b>Subtract extended</b> r[rd] = r[rs1] - r[rs2] - C; r[rd] = r[rs1] - const13 - C;
subxcc rs1,rs2,rd  subxcc rs,expr13,rd	<b>Subtract extended, and set condition codes</b> r[rd] = r[rs1] - r[rs2] - C; N = r[rd]<0; Z = r[rd] == 0; V = (r[rs1]<0 & r[rs2]>0 & r[rd]>0)   (r[rs1]>0 & r[rs2]<0 & r[rd]<0); C = (r[rs1]>0 & r[rs2]<0)   (r[rd]<0 & (r[rs1]>0   r[rs2]<0)); r[rd] = r[rs1] - const13 - C; N = r[rd]<0; Z = r[rd]==0; V = (r[rs1]<0 & const13>0 & r[rd]>0)   (r[rs1]>0 & const13<0 & r[rd]<0); C = (r[rs1]>0 & const13<0)   (r[rd]<0 & (r[rs1]>0   const13<0));
neg rs2,rd neg rd	<b>Negate</b> Synthetic instruction for: sub %g0, rs2, rd Synthetic instruction for: sub %g0, rd, rd
inc rd inc expr13,rd	<b>Increment</b> Synthetic instruction for: add rd, 1, rd Synthetic instruction for: add rd, expr13, rd
incc rd incc expr13,rd	<b>Increment, and set condition codes</b> Synthetic instruction for: addcc rd, 1, rd Synthetic instruction for: addcc rd, is, rd
dec rd dec expr13,rd	<b>Decrement</b> Synthetic instruction for: sub rd, 1, rd Synthetic instruction for: sub rd, expr13, rd

deccc rd deccc expr13,rd	<b>Decrement, and set condition codes</b> Synthetic instruction for: subcc rd, 1, rd Synthetic instruction for: subcc rd, expr13, rd
cmp rs,rs2 cmp rs,expr13	<b>Compare</b> Synthetic instruction for: subcc rs, rs2, %g0 Synthetic instruction for: subcc rs, expr13, %g0

<b>Logical Mnemonics (Format 3)</b>	
and rs1,rs2,rd and rs1,expr13,rd	<b>And</b> $r[rd] = r[rs1] \& r[rs2]$ $r[rd] = r[rs1] \& \text{const13}$
andcc rs1,rs2,rd andcc rs1,expr13,rd	<b>And, and set condition codes</b> $r[rd] = r[rs1] \& r[rs2]$ ; $N = r[rd]<0$ ; $Z = r[rd]==0$ ; $V=0$ ; $C=0$ ; $r[rd] = r[rs1] \& \text{const13}$ ; $N = r[rd]<0$ ; $Z = r[rd]==0$ ; $V=0$ ; $C=0$ ;
andn rs1,rs2,rd andn rs1,expr13,rd	<b>And negative</b> $r[rd] = r[rs1] \& \sim r[rs2]$ ; $r[rd] = r[rs1] \& \sim \text{expr13}$ ;
andncc rs1,rs2,rd andncc rs1,expr13,rd	<b>And negative, and set condition codes</b> $r[rd] = r[rs1] \& \sim r[rs2]$ ; $N = r[rd]<0$ ; $Z = r[rd]==0$ ; $V=0$ ; $C=0$ ; $r[rd] = r[rs1] \& \sim \text{const13}$ ; $N = r[rd]<0$ ; $Z = r[rd]==0$ ; $V=0$ ; $C=0$ ;
or rs1,rs2,rd or rs1,expr13,rd	<b>Or</b> $r[rd] = r[rs1]   r[rs2]$ $r[rd] = r[rs1]   \text{expr13}$
orcc rs1,rs2,rd orcc rs1,expr13,rd	<b>Or, and set condition codes</b> $r[rd] = r[rs1]   r[rs2]$ ; $N = r[rd]<0$ ; $Z = r[rd]==0$ ; $V=0$ ; $C=0$ ; $r[rd] = r[rs1]   \text{const13}$ ; $N = r[rd]<0$ ; $Z = r[rd]==0$ ; $V=0$ ; $C=0$ ;
orn rs1,rs2,rd orn rs1,expr13,rd	<b>Or negative</b> $r[rd] = r[rs1]   \sim r[rs2]$ ; $r[rd] = r[rs1]   \sim \text{expr13}$ ;
orncc rs1,rs2,rd orncc rs1,expr13,rd	<b>Or negative, and set condition codes</b> $r[rd] = r[rs1]   \sim r[rs2]$ ; $N = r[rd]<0$ ; $Z = r[rd]==0$ ; $V=0$ ; $C=0$ ; $r[rd] = r[rs1]   \sim \text{const13}$ ; $N = r[rd]<0$ ; $Z = r[rd]==0$ ; $V=0$ ; $C=0$ ;
xor rs1,rs2,rd xor rs1,expr13,rd	<b>Exclusive or</b> $r[rd] = r[rs1] \wedge r[rs2]$ ; $r[rd] = r[rs1] \wedge \text{const13}$ ;
xorcc rs1,rs2,rd xorcc rs1,expr13,rd	<b>Exclusive or, and set condition codes</b> $r[rd] = r[rs1] \wedge r[rs2]$ ; $N = r[rd]<0$ ; $Z = r[rd]==0$ ; $V=0$ ; $C=0$ ; $r[rd] = r[rs1] \wedge \text{const13}$ ; $N = r[rd]<0$ ; $Z = r[rd]==0$ ; $V=0$ ; $C=0$ ;
xnor rs1,rs2,rd xnor rs1,expr13,rd	<b>Exclusive nor</b> $r[rd] = \sim(r[rs1] \wedge r[rs2])$ ; $r[rd] = \sim(r[rs1] \wedge \text{expr13})$ ;
xnorcc rs1,rs2,rd xnorcc rs1,expr13,rd	<b>Exclusive nor, and set condition codes</b> $r[rd] = \sim(r[rs1] \wedge r[rs2])$ ; $N = r[rd]<0$ ; $Z = r[rd]==0$ ; $V=0$ ; $C=0$ ; $r[rd] = \sim(r[rs1] \wedge \text{const13})$ ; $N = r[rd]<0$ ; $Z = r[rd]==0$ ; $V=0$ ; $C=0$ ;
clr rd	<b>Clear</b> Synthetic instruction for: or %g0, %g0, rd
mov rs2,rd mov expr13,rd	<b>Move</b> Synthetic instruction for: or %g0, rs2, rd Synthetic instruction for: or %g0, expr13, rd
tst rs2	<b>Test</b> Synthetic instruction for: orcc %g0, rs2, %g0
btst rs2,rs1 btst expr13,rs1	<b>Bit test</b> Synthetic instruction for: andcc rs1, rs2, %g0 Synthetic instruction for: andcc rs1, expr13, %g0
bset rs2,rd bset expr13,rd	<b>Bit set</b> Synthetic instruction for: or rd, rs2, rd Synthetic instruction for: or rd, expr13, rd
bclr rs2,rd bclr expr13,rd	<b>Bit clear</b> Synthetic instruction for: andn rd, rs2, rd Synthetic instruction for: andn rd, expr13, rd
btog rs2,rd btog expr13,rd	<b>Bit toggle</b> Synthetic instruction for: xor rd, rs2, rd Synthetic instruction for: xor rd, expr13, rd
not rs1,rd not rd	<b>Not</b> Synthetic instruction for: xnor rs1, %g0, rd Synthetic instruction for: xnor rd, %g0, rd

## Integer Branch Mnemonics (Format 2)

Unconditional branching:	
ba{,a} label22	<b>Branch to label always</b> pc = npc; npc = const22 << 2; if (a == 1) { pc = npc; npc += 4; }
Signed number branching:	
be{,a} label22	<b>Branch if equal</b> pc = npc; if (Z) npc = const22 << 2; else if (a == 0) npc += 4; else { pc += 4; npc += 8; }
bne{,a} label22	<b>Branch if not equal</b> pc = npc; if (! Z) npc = const22 << 2; else if (a == 0) npc += 4; else { pc += 4; npc += 8; }
bl{,a} label22	<b>Branch if less than</b> pc = npc; if (N ^ V) npc = const22 << 2; else if (a == 0) npc += 4; else { pc += 4; npc += 8; }
ble{,a} label22	<b>Branch if less than or equal to</b> pc = npc; if (Z   (N ^ V)) npc = const22 << 2; else if (a == 0) npc += 4; else { pc += 4; npc += 8; }
bge{,a} label22	<b>Branch if greater than or equal to</b> pc = npc; if (!(N ^ V)) npc = const22 << 2; else if (a == 0) npc += 4; else { pc += 4; npc += 8; }
bg{,a} label22	<b>Branch if greater than</b> pc = npc; if (!(Z   (N ^ V))) npc = const22 << 2; else if (a == 0) npc += 4; else { pc += 4; npc += 8; }
Unsigned number branching:	
blu{,a} label22	<b>Branch if less than (unsigned)</b> Synonym for: bcs{,a} label22
bleu{,a} label22	<b>Branch if less than or equal to (unsigned)</b> pc = npc; if (C   Z) npc = const22 << 2; else if (a == 0) npc += 4; else { pc += 4; npc += 8; }
bgeu{,a} label22	<b>Branch if greater than or equal to (unsigned)</b> Synonym for: bcc{,a} label22
bgu{,a} label22	<b>Branch if greater than (unsigned)</b> pc = npc; if (!(C   Z)) npc = const22 << 2; else if (a == 0) npc += 4; else { pc += 4; npc += 8; }
Individual condition code branching:	
bpos{,a} label22	<b>Branch if positive (or zero)</b> pc = npc; if (! N) npc = const22 << 2; else if (a == 0) npc += 4; else { pc += 4; npc += 8; }
bneg{,a} label22	<b>Branch if negative</b> pc = npc; if (N) npc = const22 << 2; else if (a == 0) npc += 4; else { pc += 4; npc += 8; }
bcs{,a} label22	<b>Branch if C is set</b> pc = npc; if (C) npc = const22 << 2; else if (a == 0) npc += 4; else { pc += 4; npc += 8; }

bcc{,a} label22	<b>Branch if C is clear</b> pc = npc; if (! C) npc = const22 << 2; else if (a == 0) npc += 4; else { pc += 4; npc += 8; }
bvs{,a} label22	<b>Branch if V is set</b> pc = npc; if (V) npc = const22 << 2; else if (a == 0) npc += 4; else { pc += 4; npc += 8; }
bvc{,a} label22	<b>Branch if V is clear</b> pc = npc; if (! V) npc = const22 << 2; else if (a == 0) npc += 4; else { pc += 4; npc += 8; }
bz{,a} label22	<b>Branch if zero</b> Synonym for: be{,a} label22
bnz{,a} label22	<b>Branch if not zero</b> Synonym for: bne{,a} label22

### Control Mnemonics (Format 3)

jmpl rs1,rd jmpl rs1+rs2,rd jmpl rs1+expr13,rd jmpl rs1-expr13,rd jmpl expr13+rs1,rd jmpl expr13,rd	<b>Jump and link</b> r[rd] = pc; pc = npc; npc = r[rs1]; r[rd] = pc; pc = npc; npc = r[rs1] + r[rs2]; r[rd] = pc; pc = npc; npc = r[rs1] + const13; r[rd] = pc; pc = npc; npc = r[rs1] - const13; r[rd] = pc; pc = npc; npc = r[rs1] + const13; r[rd] = pc; pc = npc; npc = const13;
jmp rs1 jmp rs1+rs2 jmp rs1+expr13 jmp rs1-expr13 jmp expr13+rs1 jmp expr13	<b>Jump</b> Synthetic instruction for jmpl rs1, %g0 Synthetic instruction for jmpl rs1+rs2, %g0 Synthetic instruction for jmpl rs1+expr13, %g0 Synthetic instruction for jmpl rs1-expr13, %g0 Synthetic instruction for jmpl expr13+rs1, %g0 Synthetic instruction for jmpl expr13, %g0
call rs1	<b>Call indirect</b> Synthetic instruction for: jmpl rs1, %o7
ret	<b>Return from subroutine</b> Synthetic instruction for: jmpl %i7 + 8, %g0
retl	<b>Return from leaf subroutine</b> Synthetic instruction for: jmpl %o7 + 8, %g0
save rs1,rs2,rd  save rs1,expr13,rd	<b>Save register window and add</b> temp = r[rs1] + r[rs2]; save the register window r[rd] = temp; temp = r[rs1] + expr13; (save the register window) r[rd] = temp;
restore rs1,rs2,rd  restore rs1,expr13,rd	<b>Restore register window and add</b> temp = r[rs1] + r[rs2]; (restore the register window) r[rd] = temp; temp = r[rs1] + r[rs2]; (restore the register window) r[rd] = temp;
restore	<b>Restore register window and add</b> Synthetic instruction for: restore %g0, %g0, %g0

### Control Mnemonics (Format 2)

nop	<b>No operation</b>
sethi expr22,rd	<b>Set high-order bits</b> r[rd] = const22 << 10;
set expr32,rd	<b>Set</b> Synthetic instruction for: sethi %hi(const32), rd or rd, %lo(const32), rd where: %hi(const32) is equivalent to (const32 >> 10) %lo(const32) is equivalent to (const32   0x3ff)

## Control Mnemonics (Format 1)

call label30	<b>Call</b> r[o7] = pc; pc = npc; npc = pc + const30
--------------	---

## Assembler Directives

label:	Record the fact that label marks the current location within the current section
.section ".sectionname"	Make the sectionname section the current section
.skip n	Skip n bytes of memory in the current section
.align n	Increase the current section's location counter so it is evenly divisible by n
.byte bytevalue1, bytevalue2, ...	Allocate memory containing bytevalue1, bytevalue2, ... in the current section
.half halfvalue1, halfvalue2, ...	Allocate memory containing halfvalue1, halfvalue2, ... in the current section
.word wordvalue1, wordvalue2, ...	Allocate memory containing wordvalue1, wordvalue2, ... in the current section
.ascii "string1", "string2", ...	Allocate memory containing the characters from string1, string2, ... in the current section
.asciz "string1", "string2", ...	Allocate memory containing string1, string2, ..., where each string is NULL terminated, in the current section
.global label1, label2, ...	Mark label1, label2, ... so they are available to the linker

Copyright © 2003 by Robert M. Dondero, Jr.