Overview of the SPARC Architecture

Harry Porter
Computer Science Department
Portland State University

Abstract

This document provides an overview of the SPARC architecture and instruction set. Only a portion of the architecture is covered. The view presented here is simplified to provide an abstraction that should be adequate for user-level programming.

Memory

Main memory is byte addressable. Addresses are 4 bytes (i.e., 32 bits), allowing for up to 4 gigabytes of memory to be addressed.

Every instruction is 4 bytes (i.e., 32 bits) long. Instructions must be word aligned. That is, their addresses must be divisible by 4.

Bytes, Halfwords, Words, and Doublewords

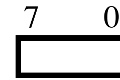
Data is manipulated in 1, 2, 4, and 8 byte units called bytes, halfwords, words, and doublewords.

	number of bytes	number of bits	required alignment	aligned address in binary
byte	1	8	none	(any)
half	2	16	2	-----------0
word	4	32	4	----------00
double	8	64	8	---------000

The bits within a word are numbered from 0 (least significant) to 31 (most significant).

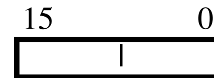
byte:

7 0



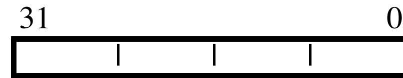
half (2 bytes):

15 0



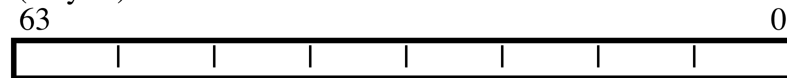
word (4 bytes):

31 0



double (8 bytes):

63 0



Alignment

Halfword, word, and doubleword data must be aligned in memory. For example, a 4 byte word value must be placed at an address that is divisible by 4, and a doubleword must be placed at an address that is divisible by 8. The load and store instructions will cause error exceptions if an attempt is made to move a halfword, word, or doubleword quantity to/from an address that is not properly aligned.

Big Endian

The SPARC architecture is “Big Endian,” meaning that when a word-sized quantity is stored in 4 bytes of memory, the most significant byte is stored in the first byte (i.e., lowest numbered address) and the least significant byte is stored in the last byte (i.e., the byte with the greatest address). In other words, if a word is stored at address x, the most significant byte will be stored in byte x and the least significant byte will be stored in address x+3.

Some other (non-SPARC) computers use “Little Endian” order, in which the least significant byte is stored in the lowest numbered address.

Binary, Decimal, and Hex Notation

An integer may be represented many ways. Normally, we represent integers in decimal using 10 numeral symbols (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9). The position (i.e., “place”) of each digit in the representation is significant. The place values are: ones, tens, hundreds, thousands, and so on.

Integers represented in binary use only two numeral symbols (0 and 1). The bit in each place is multiplied by a power of 2. Thus, the place values are: 1's, 2's, 4's, 8's, 16's, and so on.

Integers represented in hex use sixteen digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F). The hex digit in each place is multiplied by a power of 16. Thus, the place values are: 1's, 16's, 256's, 4096's, 65536's, and so on.

The first 16 numbers are shown represented in decimal, hex, and binary.

decimal	hex	binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Note that is takes exactly 4 binary digits to represent one hex digit. Thus, there is a simple and immediate correspondence between numbers represented in binary and in hex. As a result, we generally use hex notation to represent binary quantities. The translation between hex notation and binary notation is trivial; the translation between binary and decimal requires computation.

	number of bytes	number of bits	example in hex	same example in binary
byte	1	8	3A	0011,1010
half	2	16	3A0F	0011,1010,0000,1111
word	4	32	3A0F 12D8	0011,1010,0000,1111,0001,0010,1101,1000

In octal notation, three binary bits are represented with a number between 0 and 7. The octal system is problematic since there is a mismatch between the 3 bits in an octal digit and the 8 bits in a byte.

In the SPARC assembler, numbers can be represented in any base; the assembler will convert them all to binary.

Decimal: 1234
Hex: 0x04D2, 0x04d2 (capitalization is not important)
Octal: 02322 (A leading zero indicates octal)

Signed Numbers

A byte (or halfword, or word) contains a bit pattern which may be interpreted either as an unsigned integer or as a signed integer.

Signed integers are represented using the “two's complement” system. The following discussion refers to 32 bit quantities, but the ideas apply analogously to other sizes of data, too.

The table below shows how 4-byte signed numbers are represented.

bit pattern	hex representation	value (in decimal)
0111...1111	7FFF FFFF	2,147,483,647
0111...1110	7FFF FFFE	2,147,483,646
0111...1101	7FFF FFFD	2,147,483,645
•••		
0000...0011	0000 0003	3
0000...0010	0000 0002	2
0000...0001	0000 0001	1
0000...0000	0000 0000	0
1111...1111	FFFF FFFF	-1
1111...1110	FFFF FFFE	-2
1111...1101	FFFF FFFD	-3
•••		
1000...0010	8000 0002	-2,147,483,646
1000...0001	8000 0001	-2,147,483,647
1000...0000	8000 0000	-2,147,483,648

The most significant bit (bit 31) is the sign bit: 0=positive (or zero), 1=negative. Note that there is always one more negative number than there are positive numbers.

The algorithm for addition and subtraction is the same for signed and unsigned numbers. Thus, there is only one instruction for addition (and not one for signed addition and one for unsigned addition).

To negate a signed number, the algorithm is: flip all the bits and then add one. For example, to negate 2:

```
2 =               0000 0000 0000 0000 0000 0000 0000 0010
flipping bits =   1111 1111 1111 1111 1111 1111 1111 1101
adding one =      1111 1111 1111 1111 1111 1111 1111 1110
```

The result is just the two's complement representation of -2. Going from negative to positive works, too:

```
-2 =              1111 1111 1111 1111 1111 1111 1111 1110
flipping bits =   0000 0000 0000 0000 0000 0000 0000 0001
adding one =      0000 0000 0000 0000 0000 0000 0000 0010
```

Every number can be negated except the most negative number. The result of trying to negate the most negative number gives that number itself:

```
-2,147,483,648 =  1000 0000 0000 0000 0000 0000 0000 0000
flipping bits =   0111 1111 1111 1111 1111 1111 1111 1111
adding one =      1000 0000 0000 0000 0000 0000 0000 0000
```

The range of values that can be represented depends on the number of bits being used:

	byte	half	word
total number of values	2^8 = 256	2^{16} = 65,536 = 64K	2^{32} = 4,294,967,296 = 4G
largest positive number	2^7-1 = 127	2^{15}-1 = 32,767 = 32K-1	2^{31}-1 = 2,147,483,647 = 2G-1
most negative number	-2^7 = -128	-2^{15} = -32,768 = -32K	-2^{31} = -2,147,483,648 = -2G

Load Store Architecture

The SPARC uses a "load-store" architecture. Operations (such as arithmetic and logical functions) can only be performed on data stored in registers. It is necessary to move the data from memory to registers (load it) and back to memory (store it) with separate instructions. The instructions in the following example are not exactly SPARC instructions, but they give you the idea of load-store architectures. Here, "x," "y," and "z" are the names of some memory locations and "r1," "r2," and "r3" are the names of registers.

```
load    x,r1
load    y,r2
add     r1,r2,r3
store   r3,z
```

The first instruction moves data from memory into register "r1." The second instructions moves a number into register "r2." The third instruction adds the values in "r1" and "r2," placing the result into register "r3." The final instructions moves data from register "r3" back to main memory.

Each SPARC instruction either does a memory access or does a computation. The idea is that each instruction will be executed in one machine cycle. (Actually, in one machine cycle on average; due to pipelining—i.e., instruction overlapping—the execution of each instruction will be spread over several cycles.)

This architecture works best if all (or most all) the data can be kept in registers. Toward this end, the SPARC provides a large number of registers. Accessing memory slows microprocessors down and the goal is to minimize memory accessing by keeping all relevant data in registers.

Integer Registers

Each running process may access 32 "integer" registers. ("Floating-point" register are discussed later.) Each integer register contains one word (32 bits) of data. Actually, each integer register has

64 bits in a newer version of the architecture, but this is completely hidden for backward compatibility. Thus, you can safely imagine that the registers are simply 4 bytes (32 bits) long.

The registers are divided into 4 groups of 8 registers each. Registers are given names that begin with a % sign. Here are the names of the registers. Note the similarity of the numbers and letters `1` and `l`, and of `0` and `o`.

Global Registers:	`%g0, %g1, ... %g7`
Local Registers:	`%l0, %l1, ... %l7`
In Registers:	`%i0, %i1, ... %i7`
Out Registers:	`%o0, %o1, ... %o7`

Any instruction that uses an integer register can use any one of the registers. However, some registers have special uses. For example, the "in" and "out" registers are used for passing arguments to/from subroutines. The "local" registers are generally used for data local to a subroutine, while the "global" registers are used for data shared by several subroutines.

When doubleword (64 bit) data is manipulated, a pair of registers is used. In the instruction, the programmer specifies the lower numbered register of the pair; the instruction will then operate on that register and the register with the next higher number. The register pair must be aligned in the sense that the lower numbered register should be an even numbered register. For example, specifying %g2 would be legal and would indicate the register pair %g2 and %g3. The most significant word will be in %g2 and the least significant word will be in %g3. Specifying %g3 would be in error for an instruction operating on doubleword data.

Register %g0

Register %g0 is special. Its value is always zero. Any attempt to read from this register will result in zero being returned. Any attempt to store data into this register is perfectly legal; the data will simply be discarded.

The Register File and Register Windows

The SPARC processor has a large set of 32-bit integer registers. While a routine is executing, it has access to only 32 registers (the "in", "out," "local," and "global" sets of 8 registers each) although there are many more registers on the processor.

The set of registers that can be accessed at any moment is called the current "register window." You can imagine the registers as being organized in a long, linear list and the window being 24 registers big. This window slides up and down along this list, but at any time, only 24 registers may be seen through it. The 24 registers visible through the window are the "in," "local," and "out" registers (3 sets of 8 registers each).

The "global" registers do not participate in the window mechanism. There are simply 8 "global" registers, and any one of them (such as "%g3) may be accessed from anywhere at any time. However, a reference to an "in," "local," or "out" register (such as "%i3") is context dependent. It may refer to one register in one routine and a different register in another routine as discussed next.

When a new routine is called, it is given its own set new set of "out" and "local" registers. These are allocated from the above-mentioned file of registers. (More accurately, the window is moved,

making a new set of registers visible. Thus, each routine invocation can be thought of as allocating or creating 16 new registers. When the routine returns, these 16 registers disappear. They don't really go away, they just become inaccessible once more.)

Specifically, the "save" instruction (which is normally the first instruction in any routine and is not used anywhere else) causes the 16 new "out" and "local" registers to be created. The "restore" instruction (which is the last instruction in any routine) causes the registers to go away.

During a routine, any reference to a "local" or "out" register will refer to one of the 16 registers that have been most recently created for the currently executing routine. Also, any reference to an "in" register (in a *called* routine) will refer to a register that was previously referred to (in the *caller* routine) as an "out" register. Within the called routine it is impossible to access the "in" and "local" registers of the caller routine.

This mechanism should be used as follows:

- The caller places data to be passed to the called routine in its "out" registers.
- The new routine (call it "foo") is invoked. The foo routine executes a "save" instruction, thereby moving the register window.
- The foo routine gets its arguments from the same registers, which are now called the "in" registers.
- The foo routine may use its "local" registers for computation, resting assured that any routines it calls can not modify them.
- If foo wishes to call any routines, it will use its "out" register to pass arguments to those routines. It will also use its "out" registers to receive results returned from those routines.
- The foo routine may use its "in" registers for computation, resting assured that any routines it calls can not modify them. However, foo's caller may have certain expectations about what foo will leave in these registers after foo returns. In general, foo will use the "in" registers freely, trashing them all with the exception of the "in" register(s) used to return results to its caller.
- The foo routine may use its "out" registers for computations, but must remember that any calls foo makes will allow some other routine access to these register. In general, foo should assume that any call it makes may change any and all of its "out" registers. Therefore, it is a good idea to use the "out" registers just for communication with routines foo will call.
- The foo routine places its results, if any, in the "in" registers.
- Finally, the foo routine executes a "restore" instruction and returns. This gets rid of foo's register window and restores the caller's window.
- The caller can be sure that foo did not alter its "local" and "in" registers. The caller will look for foo's results in what that caller refers to as the "out" registers.

This clever register-window mechanism makes it possible to call and return from routines without ever touching main memory, thereby potentially speeding up subroutine invocation dramatically. As long as there are 6 or fewer arguments passed to a routine, and 6 or fewer results returned from a routine, and as long as each routine can get by with 8 or few local variables, main memory need not ever be accessed.

The global registers do not participate in the register windows and are intended to be used to store static, globally-accessible variables that are shared across routine calls.

Connection Between Register Window and Frame Stack in Memory

Register “%o6” is also called “%sp” and either name may be used in a program. Likewise, “%i6” is also called “%fp” and either can be used. It is possible that the above limits on parameters and local storage are inadequate and that a traditional “activation record” (also called a “stack frame”) will be allocated in main memory.

During the execution of routines, a stack of frames in main memory will be maintained. Each frame in the stack may have a different size (however, the frames must always be a multiple of 8 bytes, i.e., doubleword aligned). The stack of frames in main memory should not be confused with the “stack” of registers on the SPARC chip, and a stack frame should not be confused with a register window, although the two mechanisms a related and used for similar purposes (namely, to provide each routine with space for local variables).

During the execution of any routine, the “%sp” register will point to the top of the stack of frames. This stack generally grows downward (toward low memory), so “%sp” points to the first word in the current frame. Also, the “%fp” register points to the previous frame, i.e., to the first word of the frame of the routine that called the current routine. The routine may use either register to access fields in the frames on the stack. Typically, you might use [%sp+42] to address a field at offset 42 in the current (top) frame. You would use [%fp+56] to access something in the frame under the top frame.

A new frame may be allocated on the stack by using the “save” instruction, which conveniently subtracts a value from the old value of the stack top pointer (actually, “save” adds a negative number), saving the result in the new stack top pointer. This is done at the same time a new register window is opened up.

When a “restore” instruction is executed at the end of a routine, the “%sp” pointer is discarded (since it is lost when the register window is moved) and the old top pointer (previously in “%i6” = “%fp”) becomes the new stack top, as the register window is moved. When the window is moved, this register once again becomes accessible as “%o6” = “%sp” thereby making it the curent stack top pointer.

Saving the Return Pointer

To facilitate quick subroutine calling and return, the registers “%i7” and “%o7” are dedicated to saving the return address. They are manipulated by the “call” and “ret” instructions and are generally not manipulated by other instructions.

Floating-Point Registers

The SPARC can operate on three sizes of floating-point number:

single precision:	1 word (32 bits)
double precision:	2 word (64 bits)
quad precision:	4 word (128 bits)

There are separate instructions to operate on the floating-point registers. For example, there is an add instruction (called “add”) to add the contents of two integer registers, using the rules for integer addition. There is also a floating-point add instruction (called “fadd”) which adds two floating-point values from the floating-point register set, using the rules for floating-point addition.

There are also instructions to move data to/from the floating-point registers and to perform conversions from two's complement signed number representation to floating-point representation.

There are 32 floating-point registers of 32 bits each. Each can hold one single precision value.

Floating-point Registers `%f0, %f1, ... %f31`

To manipulate double precision values, you specify an even numbered register (n); the instruction will then act on the pair of registers (n and n+1). This allows for 16 double precision values to be stored.

To manipulate quad precision values, you specify a register divisible by 4 (n); the instruction will then act on the four registers (n, n+1, n+2, n+3). This allows for 8 quad precision values to be stored.

Note the way the floating-point registers are numbered. Registers %f4, %f5, %f6, %f7 can hold either four single precision values, or two double precision values (addressed as %f4 and %f6), or a single quad precision value (addressed as %f4).

Single Precision Registers:	%f0,	%f1,	%f2,	%f3,	%f4,	%f5,	%f6,	%f7,	%f8,	...
Double Precision Registers:	%f0,		%f2,		%f4,		%f6		%f8,	...
Quad Precision Registers:	%f0,				%f4,				%f8,	...

In addition to the 32 single precision registers mentioned above, there is an equal amount of register storage that can be used only for double and quad precision values (but not for single precision values). This allows for a total of 32 double precision values or 16 quad precision values to be stored. The numbering of these additional registers follows the pattern established by the lower numbered registers. To complete the chart given above:

Single Precision Registers:	%f0,	%f1,	%f2,	%f3,	%f4,	%f5,	%f6,	%f7,	%f8,	...	%f31
Double Precision Registers:	%f0,		%f2,		%f4,		%f6		%f8,	...	%f62
Quad Precision Registers:	%f0,				%f4,				%f8,	...	%f60

Integer Condition Code Register

There is a 4 bit register (called “CC” or “CCR”) which contains the following bits. These bits are set after certain instructions are executed to reflect the properties of the result of the operation:

Z (zero bit)	1=result is zero
	0=not zero
N (negative)	1=result is negative
	0=not negative
V (overflow)	1=operation resulted in overflow
	0=no overflow
C (carry)	1=operation resulted in a carry out
	0=no carry

These bits are tested with the various integer branch instructions, which will conditionally branch according to how these bits are set.

Floating-point Condition Code Register

There is a 4 bit register (called “FCC”) which contains the following bits. These bits are set after the floating-point compare instruction to reflect the relationship between the two operands:

E (equal bit)	1: op1=op2
	0: otherwise
L (less)	1: op1<op2
	0: otherwise
G (greater)	1: op1>op2
	0: otherwise
U (unordered)	1: op1 and op2 are unordered
	0: otherwise

These bits are tested with the various floating-point branch instructions, which will conditionally branch according to how these bits are set.

The Y Register

There is a 32-bit register called the “Y” register, which is used in the multiply and divide instructions. These instructions each require one operand to be 64 bits long. This register provides the upper (most significant) 32-bits and is loaded and read with the “`wr`” and “`rd`” instructions.

The Program Counter (PC) Register

There is a 32-bit register called the “PC”, which contains the address of the next instruction to execute. It is modified with instructions that branch or transfer the flow of control.

13-Bit Sign-Extended Immediate Values

Many of the instructions allow for an immediate (or “literal”) value to be included directly in the instruction. The subtract instruction, for example:

```
sub      %l2,27,%l6
```

This instruction subtracts 27 from the value stored in register %l2 and stores the result in %l6. The data (27) is specified immediately; it is not stored in a register. Instructions such as these allocate a 13 bit field in the instruction to contain the value. When executed, the 13 bits are interpreted as a signed integer. The 13 bit value is “sign-extended.” In other words, the most significant bit (bit 12) is duplicated to fill the value out to 32 bits. This means that any value between -4096 and 4095 can be specified literally. If you wish to subtract a larger number, you can not use a literal value; you will have to place it in a register of its own. (You can use the “set” instruction to accomplish this.)

Addressing Memory

In order to fetch or store data to / from memory, the address must first be placed in a register. The "set" instruction is ideal for moving the address into a register. For example, the following code first moves the address of "myVar" into register %l1. Then it uses that register to fetch from memory. This code increments the word whose address is "myVar".

```
set      myVar,%l1      ! Move address of myVar into %l1
ld       [%l1],%l2      ! Move data from myVar into %l2
add      %l2,1,%l2      ! Increment %l2
st       %l2,[%l1]      ! Store data back, using same address
```

A second addressing mode allows an integer offset to be added to a register to compute the effective address. For example, assume "myArr" is the address of the first element of an array of words. We wish to increment the sixth element of this array. We will need an offset of 24, since each element is 4 bytes long.

```
set      myArr,%l1
ld       [%l1+24],%l2
add      %l2,1,%l2
st       %l2,[%l1+24]
```

A third addressing mode allows us to add the contents of two registers together to compute the effective address. For example, if the offset into the array has been computed and placed in register %l3, then we can use the following code to access the desired element.

```
set      myArr,%l1
ld       [%l1+%l3],%l2
add      %l2,1,%l2
st       %l2,[%l1+%l3]
```

A common thing to do is to access fields in the activation record (or "frame"). Assume that variable x is a local variable stored at offset -8 in the frame. Then the following code would be used to increment x. (In the following, "%fp" is shorthand for referring to one of the integer registers.)

```
ld       [%fp-8],%l2
add      %l2,1,%l2
st       %l2,[%fp-8]
```

There are several "synthetic instructions." Technically, these are not valid SPARC instructions. Instead, they are introduced to make life easier for the programmer. Whenever the assembler sees a "synthetic" instruction, it will translate it into one or two legal SPARC instructions and will use those instructions instead.

For example, "set" is a synthetic instruction. It will be expanded into two instructions. Consider the following code:

```
set      myArr,%l1
ld       [%l1],%l2
```

The "set" instruction will be expanded into two instructions, giving:

```
sethi    %hi(myArr),%l1
```

```
        or      %l1,%lo(myArr),%l1
        ld      [%l1],%l2
```

The “sethi” and “or” instructions are discussed later, but in short, “sethi” moves 22 bits of data into the register, and the “or” instructions brings in the remaining 10 bits of data. The “%hi” and “%lo” functions are macros that isolate the relevant bits from the actual value provided, namely an address in memory called “myArr.”

A more efficient code sequence would be:

```
        sethi   %hi(myArr),%l1
        ld      [%l1+%lo(myArr)],%l2
```

Assembler Comments

The exclamation point (!) is used to begin comments. The comment runs through the end-of-line. I feel that every line of assembler code should be commented.

When a single comment applies to several instructions, I prefer to use a period as shown below to indicate that a multi-line comment applies to several instructions. For example:

```
        set     myVal,%l1       ! Increment myVal
        ld      [%l1],%l2       ! .  by one
        add     %l2,1,%l2       ! .
        st      %l2,[%l1]       ! .
        cmp     %l2,47          ! If myVal >= 47
        bge     loop            ! .  then goto loop
        nop                     ! .
```

Labels and White Space

Each instruction may be preceded by an optional label. This label may then be the target of a “branch” or “call” instruction. Data locations in memory may also be labeled, in which case the label could be used in “load” or “store” instructions.

If a label is present, it should be at the beginning of the line and should be followed by a colon. The instruction op-code is separated from the label by some white-space (usually a single tab character). The instruction op-code is followed by white space (usually a tab or two) and the operands. These can optionally be followed by some white space and a comment. Comments consist of an exclamation mark through end-of-line.

A label may also appear on a line by itself, in which case it applies to the current location counter, and thus will therefore be set to the address of the next instruction (notwithstanding things like “.align” pseudo-ops that may change the location counter). Op-codes (both legal SPARC instructions and pseudo-ops) must be preceded by white space, typically a tab or two.

The convention is to try to make things line up neatly. For example:

```
myLoop:                             ! myLoop:
        set     myVal,%l1           !  Increment myVal
        ld      [%l1],%l2           !  .  by one
        add     %l2,1,%l2           !  .
        st      %l2,[%l1]           !  .
        cmp     %l2,47              !  If myVal >= 47
        bge     myLoop              !  .  then goto myLoop
        nop                         !  .
        ...
myVal:  .word   0                   ! myVal: counter of things
```

The Program Counter

Every computer processor contains a register called the Program Counter (or "PC") which contains the address of the next instruction to be executed. In the simplest model of computer operation, the processor repeatedly executes this algorithm:

```
pc := 0;
loop
  fetch next instruction from memory[pc];
  pc := pc + 1;
  execute the instruction:
         Decode the instruction
         Fetch operands from memory or registers
         Perform computation
         Store results back to memory or registers
endLoop
```

Sometimes the instruction will modify the flow of control (a "call" or "branch" instruction). For such instructions, results will be stored into the PC register, causing the next instruction to be fetched from a new address.

In this simple model, each instruction executes to completion before the next instruction begins.

The Delay Slot and Annulled Instructions

In reality, the SPARC architecture is pipelined, which means that the execution of the previous instruction is overlapped with the execution of the next instruction. For the most part, this is transparent. If one instruction writes to a register and the next instruction reads from that same register, the computer will automatically delay as necessary to give the same result as if there were no pipelining.

However, for branch and call instructions (and any instruction which alters the flow of control), the pipelining will be visible to the programmer. The SPARC is always pre-fetching the next instruction and decoding it while finishing the execution of the previous instruction. The pre-fetch and decode doesn't cause any problems except when a control flow transfer is made.

When a transfer is made, the next instruction (the one after the branch) is so far along that it cannot be reasonably stopped. Therefore, the instruction immediately after the branch is always executed,

regardless of whether the branch is taken or not. This instruction is called the delay instruction and the position directly after a branch or call instruction is called the “delay slot.”

One option is to put a “nop” instruction in the delay slot. The “nop” instruction (pronounced KNOW-OP) simply does nothing. This solution is simple but wastes a machine cycle. This is what we did in the example above in the section titled “Assembler Comments.”

Another option is to move a real instruction into the delay slot. The instruction will then be executed after the branch instruction but before the target instruction. You may either select an instruction that was previously located somewhere before the branch instruction and move it down into the delay slot, or you may take the instruction that was previously the target of the branch instruction and copy (not move) it into the delay slot. In the second case, you would alter the branch instruction to jump to the instruction following the previous target instruction.

In the first case, when you select an instruction before the branch to move down into the delay slot, you must ensure that the results of the moved instruction are not needed before the branch is taken. For example, in the following sequence:

```
cmp      %l3,17
ble      loop
<delay slot>
```

we cannot move the compare instruction since it sets the condition codes, which are needed by the branch instruction.

In the second case, when you move the instruction at the target of the branch, you must keep in mind that the branch may not always be taken, but that the delay slot will always be executed.

Consider the following code:

```
loop: add    %l5,1,%l5
      ...
      cmp    %l3,17
      ble    loop
      <delay slot>
```

First, we will copy the target instruction (which increments register %l5) into the delay slot and alter the branch instruction to jump to the instruction following the add:

```
      add    %l5,1,%l5
loop: ...
      cmp    %l3,17
      ble    loop
      add    %l5,1,%l5
```

Unfortunately this code is incorrect: the add instruction will be executed an extra time, when the branch is not taken. To facilitate code like this, branch instructions allow the programmer to “annul” the delay slot. A single bit in the branch instruction tells whether the delay slot is annulled. If branch instruction has annulling turned on, and if the branch is not taken, the instruction in the delay slot will not be executed. The processor will do whatever is necessary to avoid executing the instruction, or will undo any of the effects of partially executing the delay instruction.

To turn on annulling, the programmer uses the following syntax. Here is the correct version:

```
        add    %l5,1,%l5
loop: ...
        cmp    %l3,17
        ble,a  loop
        add    %l5,1,%l5
```

Although annulling will slow the execution down when the branch is not taken, it will have no effect on execution when the branch is taken. This architecture is designed to speed up repetitive looping constructs, where many programs will spend most of their time. In a typical loop, the branch is taken many times (once for each iteration), but only not taken once (when the loop terminates).

The Location Counter

As the assembler scans and processes each line of code, it keeps track of the current memory location into which each instruction will be placed. To do this, the assembler keeps a variable called the "Location Counter," which should not be confused with the "Program Counter."

When an instruction is encountered, the location counter is advanced by 4 (i.e., by the size of the instruction, 4 bytes). When a branch instruction is processed, the location counter is always advanced by 4 and the instruction on the line following the branch is then processed.

It is important to understand that the location counter is an assembly-time concept; it does not exist at run time. On the other hand, the program counter is a run-time register and is only active when the program is running. When the assembler encounters a branch instruction, it never "takes" the branch; instead it simply moves on to the next instruction in the source file.

When the assembler encounters a label, it adds a new definition to its symbol table, using the current value of the location counter as the definition of the symbol. When the assembler encounters a branch instruction or a load or store instruction using some label, it looks that label up in the symbol table. It will use the value of that symbol in generating the instruction.

Assemblers generally make two passes over the source program. In the first pass, the location counter is set to zero, and the source code file is scanned line-by-line. Each time an instruction is encountered, the location counter is advanced by the length of the instruction, but no code is generated in the first pass. Whenever a label is encountered, it is entered into the symbol table, using the current value of the location counter as the symbol definition. In the second pass, the source code is again examined line-by-line from beginning to end. This time, machine code is generated and written to the output as each instruction is scanned.

This two-pass approach allows branches and other instructions to use labels that are defined either before or after the instruction. As the second pass begins, all symbols are already defined. As the second pass proceeds, the machine code for each instruction can be determined by plugging in the symbols' values as necessary.

Actually, the assembler can be used to place data and instructions into several segments. The assembler maintains a location counter for each segment. Thus, there is a ".data" location counter, a ".text" location counter, and a ".bss" location counter. At any one time, the assembler is "in" only one segment, as determined by the **`.data`**, **`.text`**, and **`.bss`** pseudo-ops (assembler directives). For example, if the last pseudo-op was **`.text`**, all generated data following it will be placed in the ".text" segment and the corresponding location counter will be incremented.

Branch and call instructions in the SPARC architecture are relative, and not absolute. For example, the branch instructions contain a 22 bit field to specify the target address for the branch. This 22 bit field does not contain the absolute address of the target instruction. Instead, this field is interpreted as a signed number and is added to the current program counter (at run-time) to give the target address. The benefit of using relative addresses is that the code can be "relocated," i.e., moved into any address of memory without having to be modified.

Consider the following code:

```
loop: add   %l5,5,%l5
      add   %l3,1,%l3
      cmp   %l3,17
      ble   loop
      <delay slot>
```

The branch instruction will contain a displacement of -12, meaning that the target instruction is 12 bytes before the branch instruction. At runtime, this code may be loaded at any address. For example, it might be loaded at addresses starting at 0x65432100:

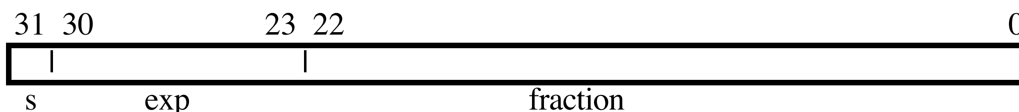
```
6543 2100    loop:   add  %l5,5,%l5
6543 2104            add  %l3,1,%l3
6543 2108            cmp  %l3,17
6543 210C            ble  loop
6543 2110            <delay slot>
```

When the processor executes the branch instruction, it takes the current value of the program counter (6543210C) and adds the relative offset in the instruction (i.e., -12), giving the address of the instruction labeled "loop." Obviously, if this code had been loaded at some other address at run-time, it would still work correctly.

To be precise, the 22 bit displacement is signed extended to 32 bits, and then multiplied by 4 (since all instructions must be word aligned). Finally, this value is added to the address of the branch instruction. This means that the target of a branch instruction must lie within -16,777,216 to 16,777,215 bytes of the branch instruction.

Floating-Point Number Representations

A "single precision" floating-point number is represented as:



The sign bit is 0=positive, 1=negative. The exponent is an 8 bit value with an interpretation described below. The fraction is a 23 bit field.

The range of numbers representable with single precision floating-point is:

Smallest Number: $1.17549435 \times 10^{-38}$

Largest Number: $3.40282347 \times 10^{+38}$

Number of digits of accuracy: about 9

The exp field can be interpreted as an unsigned integer between 0 and 255. If the exp field is between 1 and 254, this floating-point number is representing the number N:

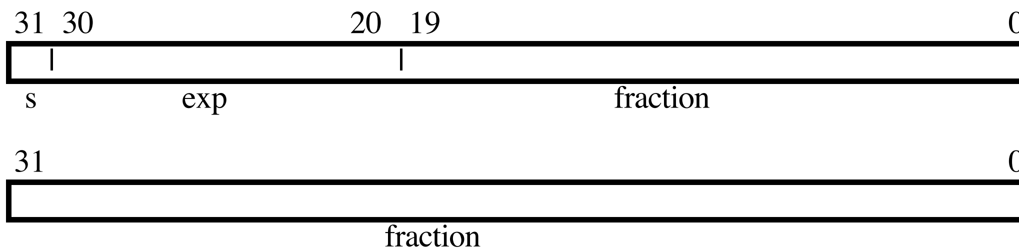
$$N = (-1)^{\mathbf{s}} \times 1.\mathbf{fraction} \times 2^{\mathbf{exp}-127}$$

If the exponent is 255 (i.e., all 1's), it signals a special case. Examples with (exp = all ones) are:

0x7FF00000 = $+\infty$
0xFFF00000 = $-\infty$
0xFFFFFFFF = NaN (i.e., "Not a Number")

If the exponent is all zeros, it signals a "subnormal" number. These are tiny numbers, close to zero. They are represented slightly differently than shown in the formula above.

A "double precision" floating-point number is represented with two words (8 bytes):



The exponent is 11 bits and the fraction is 52 bits.

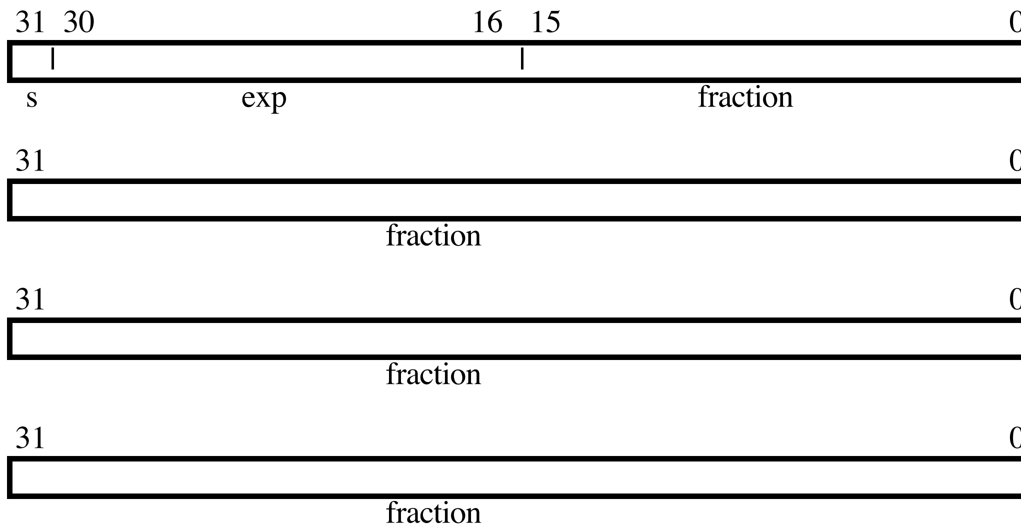
The exp field can be interpreted as an unsigned integer between 0 and 2047. The number represented is:

$$N = (-1)^{\mathbf{s}} \times 1.\mathbf{fraction} \times 2^{\mathbf{exp}-1023}$$

The range of numbers representable with double precision floating-point is:

Smallest Number: $2.2250738585072014 \times 10^{-308}$
Largest Number: $1.7976931348623157 \times 10^{+308}$
Precision: about 17 decimal digits of accuracy

A "quad precision" floating-point number is represented with four words (16 bytes):



The exponent is 15 bits and the fraction is 112 bits.

The exp field can be interpreted as an unsigned integer between 0 and 32,768. The number represented is:

$$N = (-1)^{\mathbf{s}} \times 1.\mathbf{fraction} \times 2^{\mathbf{exp}-16,383}$$

Pseudo-Ops (Assembler Directives)

The following sections describe several "pseudo-operations," which control and direct the assembler in its task. Pseudo-ops are sometimes called "assembler directives."

Pseudo-ops are included in the assembly language program file, mixed in among the real SPARC instructions. However, pseudo-ops are not executed at runtime; instead they tell the assembler how to assemble other instructions and what the put into memory before program execution begins. For example, a pseudo-op could be used to initialize a variable with some particular value. The value can be coded using convenient (C-like) notation, freeing the programmer from having to specify the precise bit-pattern used to represent the value.

Pseudo-ops begin with a period to make it easy to distinguish them from SPARC instructions.

Character Data

Here are examples of the ".ascii" and ".asciz" pseudo-ops:

```
.ascii  "abc"
.asciz  "SPARC programming is fun!\n"
```

The ".ascii" and ".asciz" pseudo-ops place N bytes of data in memory, where N is the number of bytes in the character string. In the case of .asciz, an additional byte of zero is placed after the last character (the "C" string convention), resulting in N+1 bytes. The standard "C" string escapes (such as "\n") can be used.

Data

```
.byte        expression
.half        expression
.word        expression
.double      expression
```

These pseudo-ops place data values in memory. In each case, arbitrary expressions may be provided, as long as they can be fully evaluated at assembly-time. These expressions may include many of the operators (such as + and *) available in the C language.

Also, lists of data values may be provided, in which case several byte (or half or word or double) values are placed in memory. For example:

```
.word        10,20,30
```

would initialize three consecutive words of memory.

Alignment

The ".align" pseudo-op will force the location counter to be aligned according to the argument (which should be 2, 4, or 8). In other words, one or more bytes of padding may be inserted to round the location counter up to an address divisible by 2, 4, or 8.

```
.align       2
.align       4
.align       8
```

In the first line of the following example, up to 3 bytes will be inserted (if necessary) to bring the location counter up to a multiple of 4. In the second line, 5 word-sized values will be allocated. These will be followed by a padding word (if necessary), followed by 2 double word constants.

```
.align       4
.word        43,"cs30",-1,0x0123,0
.align       8
.double      12.34,-123.456e12
```

It never hurts to place an .align pseudo-op directly before instructions that require a particular alignment, and it is necessary if the preceding instructions may have left the alignment wrong.

The .half, .word, and .double instructions require proper alignment. In this example, the second .align instruction is necessary. (Consider the case when the location counter is initially set to doubleword alignment, i.e., divisible by 8. The .word pseudo-op will allocate 5 words—20 bytes—and will leave the location counter not divisible by 8.)

Uninitialized Data Space

```
.skip       n
```

This pseudo-op skips over N bytes, where N is any expression.

Segment Control

A running process is loaded into main memory before execution begins. In Unix, a running process is divided into four "segments." These segments have the following names:

```
.data
.text
.bss
.stack
```

The assembler is used to determine which bytes will initially be loaded into the "data," "text," and "bss" segments. For example, the "data" segment might be 100 bytes long, the "text" segment might be 13,000 bytes long, and the "bss" segment might be 0 bytes long. After loading, the bytes in the "data" segment will be marked read/write, and the bytes in the "text" segment will be marked "read-only." Any attempt at run-time to modify a byte in the "text" segment will result in a "segment fault." Any attempt to access a byte outside of any segment will also result in a "segment fault." The "stack" segment will be marked read/write, and has no fixed size; instead any attempt to access bytes beyond the end of the "stack" segment will result in new pages being added to that segment.

There are three pseudo-ops that allow control over which segment to place data and instructions into. They are:

```
.data
.text
.bss
```

These pseudo-ops determine into which segment the following instructions and/or data will be placed. Each "remains in effect" until a new `.data` or `.text` or `.bss` pseudo-op is encountered.

At assembly time, each of the three segments is assumed to start at location zero. Later, when the program is loaded into main memory, an address will be selected. Also, the size of the segment is rounded up to the nearest page boundary. In a machine with 8K byte pages, our "data" segment with 100 bytes would be rounded up to 8K bytes. The "text" segment, with 13,000 bytes would be rounded up to 16K bytes. Then an address is selected for each of the four segments and the initial data is loaded into the appropriate memory locations.

Symbols

Symbols may be defined in several ways. The most common way is when an instruction is labeled. Another way is when the location of data in memory is allocated (e.g., with a ".word" pseudo-op) and a label is present.

When the assembler encounters a label, it makes a new entry in its symbol table, associating the symbol with the current value of the location counter. It also makes a note in its symbol table of which segment this address is in. Later, when the program is linked, the segment will be assigned a specific address. This will require that the value of the symbol be adjusted. Furthermore, every instruction that uses the symbol must be modified to reflect the fact that the symbol is being adjusted. Obviously, there is a lot of information in the ".o" files that indicates which symbols are used as well as where and how they are used, so that the linker can make these adjustments when the various segments are put together.

Symbols may also be defined directly using the "=" pseudo-op. Here is an example:

```
            set     myVal,%l1          !  Increment myVal
            ld      [%l1],%l2          !  .  by "incrAmt"
            add     %l2,incrAmt,%l2    !  .
            st      %l2,[%l1]          !  .
            ...
incrAmt     =       3
```

Note that "=" is a pseudo-op just like "`.align`" or "`.text`", although "=" differs in that it is not given an alphabetic name beginning with a period.

In this example, we have introduced a symbol called "incrAmt" and set its value to 3. Such a symbol is an assemble-time constant: it can never change. This symbol is absolute, which means it is not relative to any segment. During linking, when the segments are assigned addresses, no adjustment will be needed to the "add" instruction. The value of 3 will be used. The symbol "incrAmt" may be used any place the constant 3 may be used.

In this example, the operand of the "=" pseudo-op is "3." In general, the operand can be any expression, as long as it can be evaluated at assemble-time.

External Symbols

```
.global anySymbol
```

The ".global" pseudo-op tells the assembler to include information in the ".o" file to identify the named symbol and give information about its location. It is assumed that "`anySymbol`" is defined as an address within this module. This pseudo-op will make it visible to other ".o" files with which this module is linked. Consider this example:

```
        .global   main
        .align    4
main:   save      ...
        ...
        call      printf
        ...
```

If a symbol (such as "printf" in this example) is not defined in this module, the linker will look for a definition of it in other ".o" modules and will patch up the instructions in this module to point to it.

It is a good idea to put an ".align" pseudo-op before the first instruction, in case previous data left us on a non-word alignment boundary.

Note that the following is in error since the symbol “main” would be set to point to the “padding” bytes inserted by the “.align”.

```
main:
        .align    4
        save      ...
```

SPARC Instruction Set

The next sections describe the instructions available on the SPARC architecture. Not all SPARC instructions are discussed here, but there should be enough instructions to allow you to create a large range of programs.

Each section describes one, or several related, instructions. In the discussion, we use a number of abbreviations:

```
reg1
reg2
reg2_or_immed
regs
regd
freg
address
```

The terms “reg1,” “reg2,” “regs,” “regd” stand for any integer registers. The term “regs” stands for a source register, that is, a register that supplies the data. The term “regd” stands for a destination register, i.e., a register into which a result will be moved. The term “freg” stands for a floating-point register. The term “reg2_or_immed” means that the programmer may supply either an integer register or may supply a literal (immediate) value. The immediate value must be given by an expression that can be evaluated at assembly time. The expression may contain symbolic names, but must contain a value computed at run-time or a value stored in memory.

For example, one of the formats for the “add” instruction is:

```
add     reg1,reg2_or_immed,regd
```

Here are some valid examples of this instruction format:

```
add     %g2,%i3,%g3
add     %o2,1,%o2
add     %l6, (numWords*17) >> 2, %l5
```

The term “address” can take three different forms:

```
[reg1+reg2]
[reg1+offset]
[reg1]
```

In the first case, the contents of register reg1 and the contents of register reg2 are added together to give the memory address. In the second case, the 13-bit sign-extended immediate offset is added to

the contents of register reg1 to give the memory address. In the third case, the contents of register reg1 alone provide the memory address.

The registers in an address are always non-floating-point registers.

For example, one of the formats for the “load” instruction is:

```
ld      address,regd
```

Here are some valid examples of this instruction format:

```
ld      [%g2],%g3
ld      [%i2+%i3],%o3
ld      [%i4+8],%i2
ld      [%i4+(numWords*4)],%i2
```

In the following sections, the bold heading is a descriptive name for an instruction or group of instructions. There may be several closely related instructions and these will be grouped under a single heading. For example, under the heading “add,” there are two instructions, whose op-codes are “`add`” and “`addcc`”.

add

```
add     reg1,reg2_or_immed,regd
addcc   reg1,reg2_or_immed,regd
```

The contents of register reg2 or the immediate value is added to the contents of reg1 and the result is placed in register regd. In the case of addcc, the condition codes are set according to the resulting value. The immediate value is encoded as a sign-extended 13-bit value; therefore it must lie between -4096 and 4095.

sub

```
sub     reg1,reg2_or_immed,regd
subcc   reg1,reg2_or_immed,regd
```

The contents of register reg2 or the immediate value is subtracted from the contents of reg1 and the result is placed in register regd. In the case of subcc, the condition codes are set according to the resulting value. The immediate value is encoded as a sign-extended 13-bit value; therefore it must lie between -4096 and 4095.

Multiplication

```
smul    reg1,reg2_or_immed,regd
smulcc  reg1,reg2_or_immed,regd
```

This instruction takes two 32-bit arguments and produces a 64-bit result. The contents of register reg2 or the immediate value is multiplied by the contents of reg1. The least-significant 32 bits of the result are placed in register regd. The most significant 32 bits are placed in the “Y” register. (Refer to the “WR” and “RD” instructions for manipulating the “Y” register.)

If present, the immediate value is encoded as a sign-extended 13-bit value; therefore it must lie between -4096 and 4095.

All values are treated as signed quantities. There are also similar instructions to perform unsigned multiplication:

```
umul    reg1,reg2_or_immed,regd
umulcc  reg1,reg2_or_immed,regd
```

In the case of "smulcc" and "umulcc," the condition codes are set according to the result. Note that they are set according 32-bits left in regd, not the full 64-bit result.

Division

```
sdiv    reg1,reg2_or_immed,regd
sdivcc  reg1,reg2_or_immed,regd
```

This instruction divides a 64-bit number by a 32-bit number, producing a 32-bit result. The 64-bit number is constructed by using the "Y" register as the most-significant word and reg1 as the least significant word. This is divided by the contents of register reg2 or the immediate value. The 32-bit result is placed in register regd. (Refer to the "WR" and "RD" instructions for manipulating the "Y" register.)

If present, the immediate value is encoded as a sign-extended 13-bit value; therefore it must lie between -4096 and 4095.

All values are treated as signed quantities. There are also similar instructions to perform unsigned division:

```
udiv    reg1,reg2_or_immed,regd
udivcc  reg1,reg2_or_immed,regd
```

In the case of "sdivcc" and "udivcc," the condition codes are set according to the result.

and

```
and     reg1,reg2_or_immed,regd
andcc   reg1,reg2_or_immed,regd
```

The contents of register reg2 or the 13-bit immediate value is logically ANDed with the contents of reg1 and the result is placed in register regd. In the case of andcc, the condition codes are set according to the resulting value. If a 13-bit immediate value is used, it is signed extended.

or

```
or      reg1,reg2_or_immed,regd
orcc    reg1,reg2_or_immed,regd
```

The contents of register reg2 or the 13-bit immediate value is logically ORed with the contents of reg1 and the result is placed in register regd. In the case of orcc, the condition codes are set according to the resulting value. If a 13-bit immediate value is used, it is signed extended.

xor

```
xor     reg1,reg2_or_immed,regd
xorcc   reg1,reg2_or_immed,regd
```

The contents of register reg2 or the 13-bit immediate value is exclusively-or'ed with the contents of reg1 and the result is placed in register regd. In the case of xorcc, the condition codes are set according to the resulting value. If a 13-bit immediate value is used, it is signed extended.

sll - Shift Left Logical

```
sll     reg1,reg2_or_immed,regd
```

The contents of register reg1 is shifted left by N bits and the result is placed in regd. N is a number from 0 through 31 and may be in register reg2 or may be specified literally. If N is zero, the data is not shifted. Zeros will be shifted in from the right. The condition codes are not altered.

srl - Shift Right Logical

```
srl     reg1,reg2_or_immed,regd
```

The contents of register reg1 is shifted right by N bits and the result is placed in regd. N is a number from 0 through 31 and may be in register reg2 or may be specified literally. If N is zero, the data is not shifted. Zeros will be shifted in from the left. The condition codes are not altered.

sra - Shift Right Arithmetic

```
sra     reg1,reg2_or_immed,regd
```

The contents of register reg1 is shifted right by N bits and the result is placed in regd. N is a number from 0 through 31 and may be in register reg2 or may be specified literally. If N is zero, the data is not shifted. Vacated bits on the left will be filled with the previous sign bit. Thus, this instruction serves as a division by 2^N, with rounding toward minus infinity. The condition codes are not altered.

mov

```
mov     reg1_or_immed,regd
```

The contents of register reg1 or the immediate value is moved into register regd. If used, the immediate value is encoded as a sign-extended 13-bit value; therefore it must lie between -4096 and 4095. The condition codes are not altered.

This is a synthetic instruction; the assembler substitutes the following:

```
or      %g0,reg1_or_immed,regd
```

ld - Load Word

```
ld      address,regd
ld      address,fregd
```

A 32-bit (4 byte) word is moved from the memory location given by address into register regd. The destination register may be an integer register or a floating-point register. (Floating-point registers are specified as %f0 through %f31).

The condition codes are not altered. The address must be word-aligned.

The address may by given as:

```
[reg1+reg2]
[reg1+offset]
[reg1]
```

In the first case, the contents of register reg1 and the contents of register reg2 are added together to give the memory address. In the second case, the 13-bit sign-extended immediate offset is added to the contents of register reg1 to give the memory address. In the third case, the contents of register reg1 alone provide the memory address.

The registers in an address are always non-floating-point registers.

ldd - Load DoubleWord

```
ldd     address,regd
ldd     address,fregd
```

A 64-bit (8 byte) doubleword is moved from the memory location given by address into registers regd and regd+1. The destination register pair may be an integer register pair or a floating-point register pair. Regd (or fregd) must be even. The first 4 bytes (the most significant bytes) are moved into the even-numbered register.

The condition codes are not altered. The address must be doubleword-aligned.

st - Store Word into Memory

```
st      reg,address
st      freg,address
```

A 32-bit (4 byte) word is moved from register reg into the memory location given by address. The data may be moved from an integer register or a floating-point register.

The condition codes are not altered. The address must be word-aligned.

std - Store Doubleword into Memory

```
std     reg,address
std     freg,address
```

A 64-bit (8 byte) doubleword is moved from registers reg and reg+1 into the memory location given by address. The source register may be an integer register pair or a floating-point register pair. Reg (or freg) must be even. The contents of the even-numbered register is moved into the first 4 bytes (the most significant bytes).

The condition codes are not altered. The address must be doubleword-aligned.

Read and Write "Y" Register

```
wr      regs,%y
rd      %y,regd
```

The "wr" instruction moves 32-bits from any integer register "regs" into the "Y" register. The "rd" instruction moves 32-bits from the "Y" register into any integer register "regd." There are other forms of these instructions used to move data to/from registers that are not discussed in this document.

The condition codes are not altered.

cmp - Compare

```
cmp     reg1,reg2_or_immed
```

The contents of register reg1 is compared to register reg2 or to the immediate value and the condition codes are set accordingly. If an immediate value is used, it is encoded as a sign-extended 13-bit value; therefore it must lie between -4096 and 4095.

This is a synthetic instruction. The assembler generates code as if the following instruction had been supplied:

```
subcc   reg1,reg2_or_immed,%g0
```

Conditional Branching

The relevant instructions are:

```
be      label
be,a    label
bne     label
bne,a   label
bl      label
bl,a    label
ble     label
ble,a   label
bg      label
bg,a    label
bge     label
bge,a   label
bcc     label
bcc,a   label
bcs     label
bcs,a   label
bvc     label
bvc,a   label
bvs     label
bvs,a   label
```

These instructions test the current setting of the 4 condition code bits and the branch is taken if the indicated condition holds. The conditions are defined as follows:

```
be      equal
bne     not equal
bl      less than
ble     less than or equal
bg      greater than
bge     greater than or equal
bcc     carry bit clear
bcs     carry bit set
bvc     no overflow
bvs     overflow
```

There are four condition code bits, which are set in various combination to give the conditions listed above.

```
N       1=negative
Z       1=zero
C       1=carry
V       1=overflow
```

Due to pipelining, the instruction immediately following the branch instruction in memory (i.e., the instruction in the "delay slot") is normally executed before the target instruction is executed. If the annul bit in the instruction is set (as indicated by ",a" in the instruction), the delay slot is only executed if the branch is taken; if the branch is not taken, the delay instruction is annulled - that is, it is not executed.

The label is given as a 22-bit displacement. The 22-bits are shifted left two bits, sign-extended and then added to the address of this instruction to give the address of the target instruction. In assembly code, a symbolic label is normally provided.

Unconditional Branching

```
ba      label
ba,a    label
```

The branch is always taken. If the ",a" is provided, the instruction in the delay slot is not executed; otherwise, it will be executed before the branch is taken. Note that this annulling behavior is slightly different than for conditional branch instructions.

sethi

```
sethi   immed_22,regd
```

This instruction takes a 22-bit value. It zeros the lo-order (least significant) 10-bits of the register and moves the 22-bit value into the hi-order (most significant) bits. The condition codes are not altered.

The built-in assembler macro functions %hi() and %lo() are defined as:

```
%hi(x)  x >> 10
%lo(x)  x & 0x3ff
```

The %hi() and %lo() functions may be used in any expressions in any instructions processed by the assembler, just like operators +, *, etc. They will be evaluated during the assembly process, not at run-time.

set

```
set     immed_value,regd
```

The "immed_value" may be any 32-bit quantity. This instruction moves the 32-bit immediate data into register "regd." The condition codes are not set.

This is a "synthetic instruction." In general, the assembler will replace a "set" instructions with the instructions:

```
sethi   %hi(value),regd
or      regd,%lo(value),regd
```

If the hi bits are all zero, the assembler will simply substitute the following single instruction:

```
or      %g0,value,regd
```

If the lo bits are all zero, the assembler will substitute the following instruction:

```
sethi   %hi(value),regd
```

Note that since the assembler may replace a “set” instruction with two instructions, you should not use a “set” in the delay slot. The delay slot has room for only one instruction.

nop

```
nop
```

No operation is executed. This is a “synthetic instruction.” The assembler substitutes the instruction:

```
sethi   0,%g0
```

call

```
call    label
```

Control is transferred to the given instruction. The instruction in the delay slot immediately following this instruction will be executed before control is transferred. The address is a 30-bit relative offset from the location of this instruction. The address of the call instruction is stored into %o7 so it may be used for returning later.

This instruction is encoded using only 2 bits for the op-code and 30 bits for the target field. The target field is multiplied by 4 (i.e., padded on the right with 2 zero bits), since the target instruction must always be word-aligned. Thus, a call can be made to any instruction in the 4G byte (32-bit) address space. The target field is relative; that is, it is added to the program counter to give an absolute address.

call – (second form)

```
call    address
```

Here, address may be:

```
        [reg1+reg2]
        [reg1+offset]
        [reg1]
```

In the first case, the contents of register reg1 and the contents of register reg2 are added together to give the address of the routine. In the second case, the 13-bit sign-extended immediate offset is added to the contents of register reg1 to give the address. In the third case, the contents of register reg1 alone provide the address.

This is a synthetic instruction. The assembler generates code for the following:

```
jmpl    address,%o7
```

jmpl

```
jmpl    address,regd
```

Control is transferred to the instruction at address. The instruction in the delay slot immediately following this instruction will be executed before control is transferred.

The address may by given as:

```
        [reg1+reg2]
        [reg1+offset]
        [reg1]
```

In the first case, the contents of register reg1 and the contents of register reg2 are added together to give the memory address. In the second case, the 13-bit sign-extended immediate offset is added to the contents of register reg1 to give the memory address. In the third case, the contents of register reg1 alone provide the memory address.

The address of this instruction is stored into regd.

ret - Return from Routine

```
ret
```

This instruction is used to return from a (non-leaf) routine. The instruction in the delay slot immediately following this instruction will be executed before control is transferred. It is assumed that this routine executed a "save" earlier, and that the delay slot will be filled by a "restore." Thus, the saved return address is in %i7, not %o7.

This is a synthetic instruction. The assembler generates code for the following.

```
jmpl    %i7+8,%g0
```

The addition of 8 causes control to skip past the "call" instruction (4 bytes) and past the instruction in the delay slot after the "call" instruction (4 bytes).

retl - Return from Leaf Routine

```
retl
```

This instruction is used to return from a (leaf) routine. The instruction in the delay slot immediately following this instruction will be executed before control is transferred. It is assumed that this routine did not execute a "save," so the return address is in %o7.

This is a synthetic instruction. The assembler generates code for the following.

```
jmpl    %o7+8,%g0
```

save

```
save    reg1,reg_or_immed,regd
```

This instruction is normally executed as the first instruction of a non-leaf routine and creates a new activation record (i.e., frame) on the calling stack.

This instruction creates a new register window: The previous “in” and “local” registers are saved. A new set of “local” and “out” registers are allocated. The previous contents of the “out” registers become available as the “in” registers.

The reg_or_immed will normally be a 13-bit sign-extended immediate value. It will be added to the value of reg1 and the result will be stored in regd. Reg1 and the register of the second operand (if any) come from the previous register window. Regd comes from the new register window.

The normal usage of this instruction is:

```
save    %sp,-(64+4+24+args+locals)&-8,%sp
```

The 64 is the number of bytes in the activation record for saving the register window (16 registers). The 4 is the number of bytes in the activation record for the structure pointer. The 24 is the number of bytes for saving %i0 through %i5. Args is the number of bytes for additional arguments (to be passed to routines called by the routine containing this save). Locals is the number of bytes required for local and temporary variables in this routine. The &-8 is to assure that doubleword alignment is followed.

After the save, the new stack top will be computed by adding this immediate value to the old stack top and will be stored in %sp (=%o6). The previous stack top will be left in %fp (=%i6).

restore

```
restore reg1,reg_or_immed,regd
```

This instruction is normally executed as the last instruction of a non-leaf routine by being placed in the delay slot after a “ret” instruction.

This instruction restores the register window, which has the side-effect of popping the top activation record from the stack. (Before this instruction, the current stack top is in %o6 (=%sp). The old stack top was saved in %i6 (=%fp). After the register window is shifted, the %i6 (=%fp) register becomes %o6 (=%sp), thereby shrinking the stack back to its older size.)

The returning routine's out and local registers are lost (freed). The calling routine's saved in and local registers are restored. The in registers of the returning routine become (once again) the out registers of the calling routine.

Normally, the arguments are omitted:

```
restore
```

If arguments are supplied, this instruction acts like an add instruction except that the source registers come from the old register window and the destination register regd is from the new register window.

ta - Trap to the Operating System

The following instruction traps to the operating system.

```
ta       immed
```

To invoke a kernel routine (i.e., to call an operating system function from user-mode code), use the trap instruction. First place an integer in %g1 indicating the service requested. There is no delay slot following the trap instruction. The trap instruction takes an integer in the range 0..127. For example:

```
mov      6,%g1
ta       0
```

The trap instruction changes the mode to “privileged” mode (aka: “supervisor” or “system” mode) and jumps into the operating system's address space. There are a number of trap instructions, based on the condition codes. “ta” stands for “trap always.”

Args to a kernel call are passed in %o0, %o01, %o2, etc. The kernel may set the CARRY condition code to indicate the results of the call:

C=0	OK
C=1	Problems

Some of the service requests are:

3	read
4	write
5	open
6	creat
8	close
0	exit (?)
1	exit (?)

Moving Data Between Memory and Floating Registers

To move data into a floating-point register, use a load instruction and specify a floating-point register as the destination. To move data out of a floating-point register, use a store instruction and specify a floating-point register as the source.

```
ld       [address],fregd
ldd      [address],fregd
st       fregs,[address]
std      fregs,[address]
```

Here, [address] may be:

```
[reg1+reg2]
[reg1+offset]
[reg1]
```

For version 9, you may also move quad word quantities using the following instructions:

```
ldq     [address],fregd
stq     fregs,[address]
```

Moving Data Between Integer and Floating Registers

It is apparently impossible to move data directly (with one instruction) between an integer register and a floating-point register. The data must first be moved into memory, and from there to the register.

Clearing Floating Registers

To clear a floating-point register, it is necessary to load a value from memory. For example:

```
            set         myZero,%l6
            ld          [%l6],%f4
            ...
myZero:     .single     0r0
```

It might be tempting to try subtracting a register from itself; however this fails if the floating-point register contains NaN ("not-a-number").

Floating-Point Instructions

In this section, we list several instructions that perform floating-point computations.

In the notation used here, "freg" stands for any floating-point register. Of course, the register number must be aligned properly for double or quad precision operations. In the notation used

here, “freg1” and “freg2” stand for the two source operands (for binary operations); “fregs” stands for a single source register (for unary operations); and “fregd” stands for a destination register (i.e., where the result is stored).

Most of the floating-point instructions end with a letter (either “s,” “d,” or “q”) to indicate whether the operation is performed using single, double, or quad precision.

The following instructions perform floating-point addition, subtraction, multiplication, and division.

```
fadds   freg1,freg2,fregd
fsubs   freg1,freg2,fregd
fmuls   freg1,freg2,fregd
fdivs   freg1,freg2,fregd

faddd   freg1,freg2,fregd
fsubd   freg1,freg2,fregd
fmuld   freg1,freg2,fregd
fdivd   freg1,freg2,fregd

faddq   freg1,freg2,fregd
fsubq   freg1,freg2,fregd
fmulq   freg1,freg2,fregd
fdivq   freg1,freg2,fregd
```

The following instructions move data from the source register to the destination register.

```
fmovs   fregs,fregd
fmovd   fregs,fregd
fmovq   fregs,fregd
```

The following instructions negate the value in the source register and place the result in the destination register. (Note: since only the sign bit is affected, only “fnegs” is strictly necessary.)

```
fnegs   fregs,fregd
fnegd   fregs,fregd
fnegq   fregs,fregd
```

The following instructions compute the absolute value of the value in the source register and place the result in the destination register.

```
fabss   fregs,fregd
fabsd   fregs,fregd
fabsq   fregs,fregd
```

The following instructions compute the square root of the value in the source register and place the result in the destination register.

```
fsqrts  fregs,fregd
fsqrtd  fregs,fregd
fsqrtq  fregs,fregd
```

Floating-Point Comparison

The following instructions compare the value in the register1 with the value in register 2.

```
fcmps   freg1,freg2
fcmpd   freg1,freg2
fcmpq   freg1,freg2
```

The floating-point condition codes are set as follows:

E	is set to 1 iff	arg1 = arg2
L	is set to 1 iff	arg1 < arg2
G	is set to 1 iff	arg1 > arg2
U	is set to 1 iff	arg1 and arg2 are incomparable (e.g., one arg is a "NaN.")

This instruction causes an exception if either number is "signaling" NaN; no exception if the numbers are "quiet" NaNs.

A non-floating-point instruction must be executed between a floating-point compare and a floating-point branch instruction.

Note: the other floating-point arithmetic instructions (such as "fadds") do not set the floating-point condition codes.

Floating-Point Branch Instructions

There is a set of instructions to branch conditionally, depending on the state of the floating-point condition codes.

```
fbl     label
fble    label
fbg     label
fbge    label
fbe     label
fbne    label
fbo     label
fbu     label
```

The last two branch instructions test whether the two values compared in the last compare instruction were ordered or unordered.

Just like the integer branch instructions, there is a delay slot following the branch instruction. Due to pipelining, the instruction immediately following the branch instruction in memory (i.e., the instruction in the "delay slot") is normally executed before the target instruction is executed. If the annul bit in the instruction is set, the delay slot is only executed if the branch is taken; if the branch is not taken, the delay instruction is annulled - that is, it is not executed. To set the annul bit, ",a" is added to the instruction op-code. For example:

```
fble,a  label
```

Note: In older versions of the SPARC (i.e., for Sparc-V8), the results of a floating-point compare instruction were not available immediately. The programmer is required to insert an extra

instruction between the compare and the following branch instruction. For Sparc-V9, this restriction is removed.

f?to? - Conversions Between Number Representations

The following instructions convert numbers from one representation to another.

```
fstoi   fregs,fregd
fdtoi   fregs,fregd
fqtoi   fregs,fregd
fitos   fregs,fregd
fitod   fregs,fregd
fitoq   fregs,fregd
fstod   fregs,fregd
fstoq   fregs,fregd
fdtos   fregs,fregd
fdtoq   fregs,fregd
fqtos   fregs,fregd
fqtod   fregs,fregd
```

These instructions can be understood using the following abbreviations. For example, the instruction "fstoq" takes a single precision floating-point number (s) stored in register "fregs" and produces a quad precision (q) representation of the same number, and stores it in the four registers "fregd," "fregd+1," "fregd+2," and "fregd+3."

i = signed integer word
s = single precision floating-point
d = double precision floating-point
q = quad precision floating-point

Both source and destination registers are floating registers, even in the case of signed integer data. Condition codes are not set. The number will be rounded when it doesn't fit.

gdb - A Debugger for Unix Programs

To use the debugger, you should compile / assemble with the "-g" option. This will include symbol table information in the executable file. (Normally, symbol information is included in the .o file for the linker, but is excluded from the executable file.) Gdb will work with assembly code programs and with programs written in high-level languages like C and Pascal.

```
     gcc -g myProg.s
or:  gcc -g myProg.c
```

To invoke the debugger, type:

```
gdb a.out
```

Gdb accepts commands from the user. Each command may be spelled out fully and many commands may be abbreviated. Some of the gdb commands are:

help *[keyword]*

Displays help info.

r *[args]*

Short for “run.” Begin program execution. Normally, when gdb begins, it loads the program, but does not begin its execution. Command-line arguments may be provided optionally.

b ****addr*

Sets a breakpoint. Execution will proceed up to the indicated instruction and will then be suspended. You will then be thrown into the gdb command line interpreter.

i b

Short for “info break.” This displays a list of all the breakpoints that have been set.

c

Short for “continue.” Resumes execution after a breakpoint has been reached.

p $g4

Short for “print.” Prints the value of a register. Note that registers are specified with “ $ ” instead of “%”. Data may be displayed in a number of different formats. The format codes are:

d - decimal
x - hex
t - binary
f - floating-point
i - instruction
c - character

The default is to display in decimal. To display the value of register %i5 in hexadecimal, type:

```
p/x $i5
```

To look at the current value of the program counter type:

```
p/x $pc
```

The following command performs a computation and displays the result:

```
p 23+17
```

x addr

Short for "examine." This command is used to dump memory.

In the next example, the first word of the "main" procedure is displayed as an instruction:

```
x/i main
```

Hitting "enter" will repeat the previous command, generally incrementing the "current location." Thus, by hitting enter several times, you can display several instructions from the "main" routine.

You may also display the values of variables. For example, to print the value of a variable in hex type:

```
x/x myVar
```

disass

This instruction will disassemble and display the current routine.

set $g4=0x456789AB

This instruction will move data into register %g4. You may also use this instruction to alter memory. Consider the following code:

```
        .data
        .global     myVar
myVar:  .word
```

To display memory contents type:

```
p myVar
```

That would print the variable in decimal; to see it in hex type:

```
p/x myVar
```

To alter memory type:

```
set myVar=0xFEDCBA34
```

To display the contents of memory at an arbitrary location (e.g., $1C000_{16}$) type:

```
x/x 0x1C000
```

ni *[count]*

Short for "next instruction." This command executes a single instruction and the returns to the gdb command line interpreter. It steps "over" calls—that is, if the current instruction is a "call," this command will execute the entire routine and stop just after it returns. The count is optional; if provided, this command will execute "count" instructions before stopping.

si *[count]*

Short for "step into." This command executed a single instruction and the returns to the gdb command line interpreter. It steps "into" calls—that is, if the current instruction is a "call," this command moves to the first instruction of the routine and suspends execution. You may then single step through the routine. The count is optional; if provided, this command will execute "count" instructions before stopping.

next *[count]*

This command single steps a single source-code line, stepping over called subroutines.

step *[count]*

This command single steps a single source-code line, stepping into called subroutines.

q

This command quits gdb.

where
bt

These two commands print information from the activation record stack, indicating which routines are currently active. "bt" stands for "backtrace." They each display slightly different information.

call foo(1,2,3)

From gdb you can call source-level routines. For example, to call a routine named "foo" providing three arguments, you can type the above command. If "foo" returns a value, it will be printed.

A file called ".gdbinit" will be consulted upon starting gdb. For example, the following file will set a breakpoint at the beginning of the main routine, set up things so the gdb will display the current instruction whenever a breakpoint is hit, begin the program (which will immediately stop at the first instruction), and finally, disassemble and print the main routine.

.gdbinit

```
break *&main
display/i $pc
r
disass
```