The Q&D First Time Compiler Writer's Guide to the SPARC V.8 Instruction Set Architecture

by

Peter Graham
Dept. of Computer Science
University of Manitoba
Winnipeg, MB
CANADA R3T 2N2
pgraham@cs.umanitoba.ca

© Peter Graham, 1994

Introduction

This document has been written to provide the fundamental information needed for a student in a compiler writing course to produce code for the SPARC architecture. SPARC is an architecture not an implementation. The difference is that the "architecture", sometimes referred to as the "instruction set architecture" (ISA), specifies only those features which are visible to a programmer of the machine. It does not specify details of the "implementation" which do not affect the programming interface. Such features may affect performance, but do not affect how code is written. Designers of chips which implement the SPARC architectue are free to implement the features of the ISA in any manner they choose. This can lead to highly efficient (and expensive) implementations or less efficient (but more cost-effective) ones. There are several versions of the SPARC architecture which permit for the evolution of the architecture and prevent the rapid decline of it. This document describes version 8 of the SPARC architecture. It does not consider any implementation details and thus lacks the details required to effectively implement certain advanced compiler optimizations that depend on implementation details such as cache sizes. This should not be a problem for a student compiler project.

SPARC is a RISC architecture as are most modern architectures (at the time of writing). RISC stands for Reduced Instruction Set Computer. This name is only partially indicative of the overall design philosophy for the architecture. The name RISC implies an architecture with a small, simple set of instructions. Of course, the instruction set is only one aspect of a computer's architecture. There is also the register set, the addressing modes, instruction formats, etc. All of these features are typically also "reduced" (or simplified) in a typical RISC architecture.

The name originates from early research into effective microprocessor design done at various locations including U.C. Berkeley (the RISC-1 microprocessor which eventually evolved into the SPARC architecture), Stanford (a RISC-like architecture which led to the MIPS architecture), and IBM (an effort which after much thrashing led to the POWER architecture). The motivation for RISC machines arose from the observation that existing machines spent over 90% of the time executing instructions from only about 15% of the instruction set. The rapid advances being made in VLSI technology at the time were making more powerful microprocessors (32 bit machines) possible and cost-effective. Such machines could conceivably rival mini and mainframe computer systems if they could provide similar performance. To do this though it was important to make the best possible use of the available "silicon real estate". This meant that instructions (and other "complex" features of the architecture) had to be sacrificed to make room for performance related implementation features such as pipelines and caches. The RISC philosophy provided a guideline for what to sacrifice.

RISC machines typically have large register sets (not strictly a component of RISCs but a good use of silicon), a simple instruction set (which can be used to do complex operations through instruction sequences), a fixed instruction format (which helps with pipelining), and limited but general purpose addressing modes (normally used only by load and store instructions). A key philosophy behind RISC is that the language compilers which produce code for these architectures should be relied upon to exploit architectural knowledge to produce code sequences which will execute efficiently. To accomplish this,

RISC architectures often make more details of the architecture visible to the programmer (and compiler) than older CISC (Complex Instruction Set Computer) architectures do. Some architectures take this idea to the extreme (e.g. the I960 where the operation of the pipeline is exposed and under compiler control - the compiler says when the next pipeline "tick" occurs) while others, including the SPARC, take a more moderate approach. The exposure of architectural detail complicates the programmer or compiler writer's life but the other aspects of the RISC philosophy simplify it (e.g. a compiler has fewer code generation choices in a RISC than in a CISC). For better or worse, RISC architectures are the current trend and compiler writers must be prepared to generate code for them.

Normally compilers may generate either assembly or machine code. For a student compiler, in particular, it is preferable to produce assembly code. This simplifies code generation because symbolic names rather than absolute addresses may be used and it also aids in readability, understandability, and debugging. Since this document is geared towards the writing of student compilers, no information concerning machine code details (e.g. opcodes and operand formats) will be given. All discussions will address the assembly code level. If more detailed information is required concerning machine level or other details the definitive reference souce is "The SPARC Architecture Manual - Version 8" by SPARC International Ltd., published by Prentice Hall, ISBN 0-13-825001-4. A similar manual for the 64 bit version of the SPARC architecture (version 9) should also be available from Prentice Hall.

The rest of this document is organized into several sections. The first of these is an overview which discusses the features of the ISA at a high level. The architecture is then presented in detail in the following four sections which discuss data formats, the register set, operand addressing, and the instruction set respectively. Immediately thereafter, the assembly language syntax is presented. Finally, the document concludes with some sample code illustrating the use of the assembly language.

Overview

The SPARC architecture defines a modern 32 bit RISC processor family featuring a complete set of computational instructions, a load-store architecture, and a windowed register set. The key features of SPARC of interest to compiler writers include the following:

1. A linear address space of 2^{32} bytes - this is important because it means that the compiler writer need not be concerned about addressing limitations introduced by architectures such as the old 80x86 family.
2. Only two addressing modes (register+register and register+immediate) - this means that the ways in which data may be accessed by generated code are limited and thus simplifies the process of deciding on code to address program data.
3. Simple operand format - all instructions (except loads and stores, and ...) operate using three registers (source1, source2, and destination) - this precludes the need to select a "location" (register or memory) for each operand and thus simplifies not only code generation but also register management.

4. Cleanly separated integer and floating point sub-units - this means that in many cases, floating point code and nearby integer code may be dealt with more or less separately (e.g. there are separate integer and floating point register sets so instructions in the two groups do not conflict for register access)
5. Few and simple instruction formats - this is of concern only to compilers generating machine code but in this case offers simplified code generation
6. Overlapped windowed integer register set - this register organization complicates the handling of code related to register management (in particular, handling register spills is more difficult).
7. Delayed control transfer - this feature is an example of architectural detail made visible to the compiler and complicates code generation by asking the compiler to do some simple code reorganization

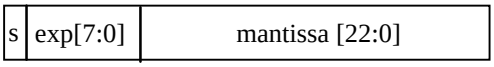
Data Formats

The SPARC architecture defines several different data types with corresponding formats. There are the traditional signed and unsigned integer and floating point types which are available in various sizes to support differing needs for range and precision. The supported operand sizes are:

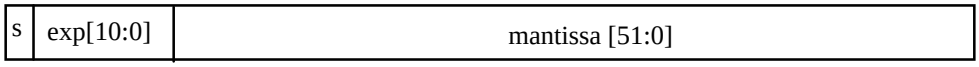
Name	Size
byte	8 bits
halfword	16 bits
word	32 bits
doubleword	64 bits
quadword	128 bits

Signed integer data may be represented in any of four different sizes; byte, halfword, word, and doubleword. Unsigned integers are available in the same sizes. Floating point data is available in any of three sizes; word, doubleword, and quadword.

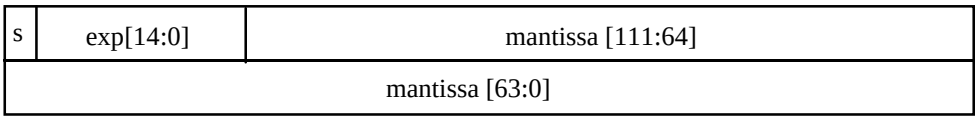
Signed integers are represented using 2’s complement while unsigned are represented as simple binary numbers. The floating point representations vary with the data size selected. A single precision floating point number (stored in a word) has the following format:



A double precision number has the format:



An extended precision number has the format:



The resulting ranges of the various floating point formats are summarized in the following table:

Precision	Value	Sizes
single	$(-1)^s \times 2^{e-127} \times 1.f$	e(8 bits), f(23 bits)
double	$(-1)^s \times 2^{e-1023} \times 1.f$	e(11 bits), f(52 bits)
extended	$(-1)^s \times 2^{e-16383} \times 1.f$	e(15 bits), f(112 bits)

The SPARC architecture definition requires that data be aligned on certain byte boundaries in order to ensure efficient access. The rules for alignment are simple halfwords must be aligned on addresses which are divisible by two, words on addresses divisible by four, etc. Related to alignment is the question of how bytes are stored within halfwords, words, etc. SPARC is a “big-endian” architecture. Thus, the most significant byte is stored first in a larger unit of storage (i.e. at a lower memory address).

Register Set

The SPARC architecture’s register set may be conveniently divided into two parts: those registers which are used for special purposes, and the “general purpose” registers which are structured as a set of overlapping *register windows*. We focus on the general purpose registers since the special purpose ones are of only limited interest to the compiler writer and are typically related to *advanced* functions which are beyond the scope of this document.

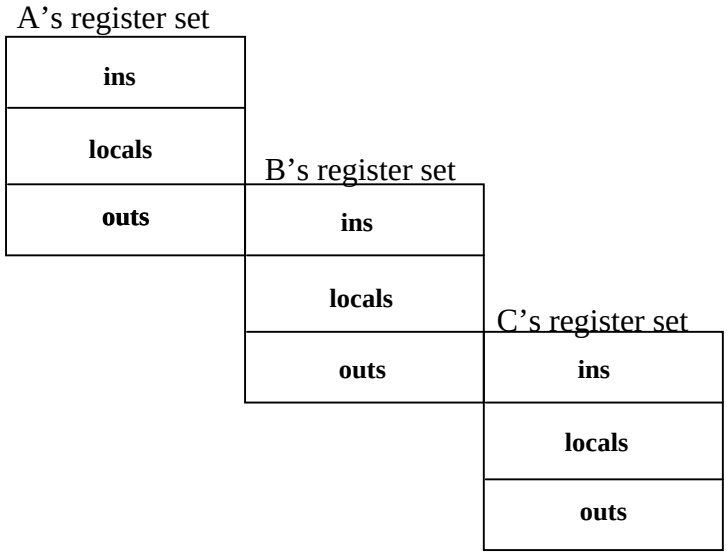
The general purpose registers may also be divided into two groups: the integer register set and the floating point register set. The floating point register set is a conventional collection of 32, 32 bit registers. The integer registers however are structured as a set of overlapped register “windows” together with 8 *global* integer registers.

The SPARC architecture supports from 2 to 32 sets of register windows each consisting of 24 registers. The 24 registers are subdivided into three groups of 8 registers each; the *in* registers, the *out* registers, and the *local* registers. The *in* and *out* registers are used to permit efficient passing of arguments to subroutines as well as the efficient return of results. To accomplish this, the *out* registers of a caller are mapped to the same physical registers as the *in* registers of the callee. Thus, any data written into the caller’s *out* registers appears in the callee’s *in* registers when it is invoked. Similarly, the callee may write return values in its *in* registers and after return, the caller will find that data in its *out* registers. The use of the overlapped register set is ideal for passing small data values such as integers, characters, and pointers to more complex structures. The passing of large structures and (optionally - integer registers could be used) floating point values is accomplished using conventional stack based techniques. The *local* registers are strictly for the use of the current subroutine and are not shared with any others.

The registers may be addressed using two different forms. The *in* registers may be referred to as ‘in[0]’ through ‘in[7]’. Similarly, the *out* registers, the *local* registers, and the *global* registers may be referred to as ‘out[0]’ through ‘out[7]’, ‘local[0]’ through

‘local[7]’, and ‘global[0]’ through ‘global[7]’ respectively. Alternatively, the registers may be referred to as ‘r[0]’ through ‘r[31]’ with ‘global[0]’ through ‘global[7]’ mapping to ‘r[0]’ through ‘r[7]’, ‘out[0]’ through ‘out[7]’ mapping to ‘r[8]’ through ‘r[15]’, ‘local[0]’ through ‘local[7]’ mapping to ‘r[16]’ through ‘r[23]’, and ‘in[0]’ through ‘in[7]’ mapping to ‘r[24]’ through ‘r[31]’.

The following example illustrate overlapping register windows. In what follows, assume that procedure ‘A’ calls procedure ‘B’ which calls procedure ‘C’.



Certain of the registers are used for special purposes during particular processor operations and thus, caution should be used when generating code which uses them. In particular, if ‘r[0]’ is referenced as a source operand, the constant zero is returned. Further, ‘r[15]’ is modified by the “CALL” instruction and ‘r[17]’ and ‘r[18]’ are set when a trap occurs. This last special use of registers should not be a concern to the normal compiler writer since the modification occurs only in the register window for the trap handler and since code is not being generated for any trap handlers, there is no need for concern during code generation.

Finally, certain operations (integer multiply and divide) also require register pairs as operands. These are specified by supplying the *even* numbered register of the pair. Special attention should be paid to this during register allocation.

Operand Addressing

The SPARC is a *load store* machine architecture. That means that computational instructions such as adds and compares never reference memory. All such operations occur within the general purpose registers. Only the load and store instructions actually reference memory. This is a typical feature of RISC machine architectures. Thus, the form of operand addressing is only a concern for load and store instructions. In keeping with the RISC philosophy, the available addressing modes are limited and simple.

A memory location may be specified using either two registers or a register and an immediate value (in both cases, the two values are added together to form an effective address from which data is loaded or to which data is stored).

Operands may also be "addressed" in registers for computational instructions. In this case, the registers specification may be given in either form discussed previously.

Instruction Set

The fundamental principle of a RISC architecture, as the name suggests, is a restricted set of orthogonal instructions. The SPARC architecture adheres to this principle. The following describes an *incomplete* subset of the instructions offered by the SPARC architecture. Those instructions which are not of concern to the beginning compiler writer are simply omitted. For example, the instructions for optional co-processors are not presented. On the other hand, there are definitely more instructions discussed here than will be needed for a typical compiler project. You must select an appropriate instruction for whatever must be done.

In what follows, the notation '$\texttt{reg}_{\texttt{rd}}$' refers to an integer destination register specification. Similarly, '$\texttt{reg}_{\texttt{rs1}}$' refers to the first integer source register and '$\texttt{reg}_{\texttt{rs2}}$' refers to the second integer source register. Similar notations prefixed by an 'f' identify floating point registers (e.g. '$\texttt{freg}_{\texttt{rs1}}$'). The notation '`reg_or_imm`' corresponds to an operand which may be either a register specification or a 13 bit signed immediate (i.e. constant) value. The notation 'imm22' refers to an unsigned 22 bit immediate value. The '`address`' specifications may be either two registers or a register and an immediate value (as described previously). Bear in mind that addresses specified must satisfy the alignment requirements specified earlier. Finally, the notation '`label`' refers to a label in the assembly language program.

For instructions which operate on two operands, the first operand specification refers to the left operand while the second refers to the right operand. This is important for non-commutative operations.

The SPARC architecture supports condition codes, both integer and floating point which are separate. Certain instructions set the condition codes while others do not. Integer instructions have opcodes suffixed with 'cc' if they set the integer condition codes. Floating point operations always set the floating point condition codes.

Integer Load Instructions

ldsb load signed byte - right justified in register and sign extended
```
ldsb [address], reg_rd
```
ldsh load signed halfword - right justified in register and sign extended
```
ldsh [address], reg_rd
```
ldub load unsigned byte - right justified in register and zero filled
```
ldub [address], reg_rd
```
lduh load unsigned halfword - right justified in register and zero filled
```
lduh [address], reg_rd
```
ld load word
```
ld   [address], reg_rd
```

ldd load doubleword - load into an even-odd register pair

```
ldd   [address], reg_rd
```

Integer Store Instructions

stb store byte - signed and unsigned are the same (writes to a byte)

```
stb   reg_rd, [address]
```

sth store halfword - signed and unsigned are the same (writes to a halfword)

```
sth   reg_rd, [address]
```

st store word

```
st    reg_rd, [address]
```

std store doubleword - store doubleword from an even-odd register pair

```
std   reg_rd, [address]
```

Floating Point Load Instructions

ld load floating point - note overloading of ‘ld’ (but ‘freg’ not ‘reg’)

```
ld    [address], freg_rd
```

ldd load double floating point - note overloading of ‘ldd’ (but ‘freg’ not ‘reg’)

```
ldd   [address], freg_rd
```

Floating Point Store Instructions

st store floating point - note overloading of ‘st’ (but ‘freg’ not ‘reg’)

```
st    freg_rd, [address]
```

std store double floating point - note overloading of ‘std’ (but ‘freg’ not ‘reg’)

```
std   freg_rd, [address]
```

Logical Instructions

and logical AND of two operands, condition codes not set

```
and       reg_rs1, reg_or_imm, reg_rd
```

andcc logical AND of two operands, condition codes are set

```
andcc     reg_rs1, reg_or_imm, reg_rd
```

andn logical AND with second operand negated, condition codes not set

```
andn      reg_rs1, reg_or_imm, reg_rd
```

andncc logical AND with second operand negated, condition codes are set

```
andncc    reg_rs1, reg_or_imm, reg_rd
```

or logical OR of two operands, condition codes not set

```
or        reg_rs1, reg_or_imm, reg_rd
```

orcc logical OR of two operands, condition codes are set

```
orcc      reg_rs1, reg_or_imm, reg_rd
```

orn logical OR with second operand negated, condition codes not set

```
orn       reg_rs1, reg_or_imm, reg_rd
```

orncc logical OR with second operand negated, condition codes are set

```
orncc     reg_rs1, reg_or_imm, reg_rd
```

xor logical XOR of two operands, condition codes not set

```
xor       reg_rs1, reg_or_imm, reg_rd
```

xorcc logical XOR of two operands, condition codes are set

```
xorcc     reg_rs1, reg_or_imm, reg_rd
```

xnor logical XNOR of two operands, condition codes not set

```
        xnor    reg_rs1, reg_or_imm, reg_rd
```
xnorcc logical XNOR of two operands, condition codes are set
```
        xnorcc  reg_rs1, reg_or_imm, reg_rd
```

Shift Instructions

sll shift left logical, zero filled from the right
```
        sll     reg_rs1, reg_or_imm, reg_rd
```
srl shift right logical, zero filled from the left
```
        srl     reg_rs1, reg_or_imm, reg_rd
```
sra shift right arithmetic, sign bit propogated from the right
```
        sra     reg_rs1, reg_or_imm, reg_rd
```

Arithmetic Instructions

add add, condition codes not set
```
        add     reg_rs1, reg_or_imm, reg_rd
```
addcc add, condition codes are set
```
        addcc   reg_rs1, reg_or_imm, reg_rd
```
addx add extended (with carry), condition codes not set
```
        addx    reg_rs1, reg_or_imm, reg_rd
```
addxcc add extended (with carry), condition codes are set
```
        addxcc  reg_rs1, reg_or_imm, reg_rd
```
sub subtract, condition codes not set
```
        sub     reg_rs1, reg_or_imm, reg_rd
```
subcc subtract, condition codes are set
```
        subcc   reg_rs1, reg_or_imm, reg_rd
```
subx subtract extended (with carry), condition codes not set
```
        subx    reg_rs1, reg_or_imm, reg_rd
```
subxcc subtract extended (with carry), condition codes are set
```
        subxcc  reg_rs1, reg_or_imm, reg_rd
```
umul unsigned 32x32 bit multiply, condition codes not set
```
        umul    reg_rs1, reg_or_imm, reg_rd
```
umulcc unsigned 32x32 bit multiply, condition codes are set
```
        umulcc  reg_rs1, reg_or_imm, reg_rd
```
smul signed 32x32 bit multiply, condition codes not set
```
        smul    reg_rs1, reg_or_imm, reg_rd
```
smulcc signed 32x32 bit multiply, condition codes are set
```
        smulcc  reg_rs1, reg_or_imm, reg_rd
```
udiv unsigned 64x32 bit divide, condition codes not set
```
        udiv    reg_rs1, reg_or_imm, reg_rd
```
udivcc unsigned 64x32 bit divide, condition codes are set
```
        udivcc  reg_rs1, reg_or_imm, reg_rd
```
sdiv signed 64x32 bit divide, condition codes not set
```
        sdiv    reg_rs1, reg_or_imm, reg_rd
```
sdivcc signed 64x32 bit divide, condition codes are set
```
        sdivcc  reg_rs1, reg_or_imm, reg_rd
```

Miscellaneous Instructions

sethi zero the least significant 10 bits of ‘reg_{rd}’ and replaces its high order 22 bits with the value from its 22 bit immediate value

```
sethi    imm22, reg_rd
```

nop no operation

```
nop
```

Branch Instructions

The SPARC architecture is designed for pipelined execution and to prevent stalls in the pipeline, branch delay slots are implemented. This means that in most cases, the instruction immediately *following* the branch in memory is executed regardless of whether or not the branch is taken. It is the compiler writer’s responsibility to place a “useful” instruction in the branch delay slot following branches instructions. Moving an instruction from before the branch to after the branch is what is normally attempted, but this is a difficult optimization. Placing a ‘`nop`’ in the branch delay slot is always correct if not optimal. Another way to deal (sub-optimally) with the branch delay slot is to annul the instruction following the branch in which case it is not executed but a pipeline stall is required. The suffix ‘,a’ in the following indicates that the annul bit is set.

ba{,a} branch always

```
ba       label
```

bn{,a} branch never

```
bn       label
```

bne{,a} branch not equal

```
bne      label
```

be{,a} branch equal

```
be       label
```

bg{,a} branch greater than

```
bg       label
```

bge{,a} branch greater or equal

```
bge      label
```

bl{,a} branch less than

```
bl       label
```

ble{,a} branch less or equal

```
ble      label
```

bleu{,a} branch less or equal unsigned

```
bleu     label
```

bcc{,a} branch carry clear (i.e. no carry)

```
bcc      label
```

bcs{,a} branch carry set

```
bcs      label
```

bpos{,a} branch positive

```
bpos     label
```

bneg{,a} branch negative

```
bneg     label
```

bvc{,a} branch overflow clear

```
        bvc     label
```
bvs{,a} branch overflow set
```
        bvs     label
```
fba{,a} floating point branch always
```
        fba     label
```
fbn{,a} floating point branch never
```
        fbn     label
```
fbg{,a} floating point branch greater than
```
        fbg     label
```
fbl{,a} floating point branch less than
```
        fbl     label
```
fbne{,a} floating point branch not equal
```
        fbne    label
```
fbe{,a} floating point branch equal
```
        fbe     label
```
fbge{,a} floating point branch greater or equal
```
        fbge    label
```
fble{,a} floating point branch less or equal
```
        fble    label
```

Subroutine-related Instructions

save save caller's window and add operands (from old context) and write the result (in the new context)
```
        save    reg_rs1, reg_or_imm, reg_rd
```
restore restore caller's window and add operands (from old context) and write the result (in the new context)
```
        restore reg_rs1, reg_or_imm, reg_rd
```
call call and link - PC relative immediate address
```
        call    label
```
jumpl jump and link - register indirect (2 regs or reg + immed)
```
        jmpl    address, reg_rd
```

Floating Point Instructions

fmovs move '$freg_{rs2}$' to '$freg_{rd}$'
```
        fmovs   freg_rs2, freg_rd
```
fnegs move '$freg_{rs2}$' to after negating it '$freg_{rd}$'
```
        fnegs   freg_rs2, freg_rd
```
fabss move absolute value of '$freg_{rs2}$' to '$freg_{rd}$'
```
        fabss   freg_rs2, freg_rd
```
fsqrts single precision floating point square root
```
        fqrts   freg_rs2, freg_rd
```
fsqrtd double precision floating point square root
```
        fsqrtd  freg_rs2, freg_rd
```
fsqrtq quad precision floating point square root
```
        fsqrtq  freg_rs2, freg_rd
```
fadds single precision floating point add
```
        fadds   freg_rs1, freg_rs2, freg_rd
```

faddd double precision floating point add

```
faddd    freg_rs1, freg_rs2, freg_rd
```

faddq quad precision floating point add

```
faddq    freg_rs1, freg_rs2, freg_rd
```

fsubs single precision floating point subtract

```
fsubs    freg_rs1, freg_rs2, freg_rd
```

fsubd double precision floating point subtract

```
fsubd    freg_rs1, freg_rs2, freg_rd
```

fsubq quad precision floating point subtract

```
fsubq    freg_rs1, freg_rs2, freg_rd
```

fmuls single precision floating point multiply

```
fmuls    freg_rs1, freg_rs2, freg_rd
```

fmuld double precision floating point multiply

```
fmuld    freg_rs1, freg_rs2, freg_rd
```

fmulq quad precision floating point multiply

```
fmulq    freg_rs1, freg_rs2, freg_rd
```

fsmuld single precision floating point multiply to double precision result

```
fsmuld   freg_rs1, freg_rs2, freg_rd
```

fdmulq double precision floating point multiply to quad precision result

```
fdmulq   freg_rs1, freg_rs2, freg_rd
```

fdivs single precision floating point divide

```
fdivs    freg_rs1, freg_rs2, freg_rd
```

fdivd double precision floating point divide

```
fdivd    freg_rs1, freg_rs2, freg_rd
```

fdivq quad precision floating point divide

```
fdivq    freg_rs1, freg_rs2, freg_rd
```

fcmps single precision floating point compare

```
fcmps    freg_rs1, freg_rs2
```

fcmpd double precision floating point compare

```
fcmpd    freg_rs1, freg_rs2
```

fcmpq quad precision floating point compare

```
fcmpq    freg_rs1, freg_rs2
```

Conversion Instructions

fitos convert integer to single precision floating point

```
fitos    freg_rs2, freg_rd
```

fitod convert integer to double precision floating point

```
fitod    freg_rs2, freg_rd
```

fitoq convert integer to quad precision floating point

```
fitoq    freg_rs2, freg_rd
```

fstoi convert single precision floating point to integer

```
fstoi    freg_rs2, freg_rd
```

fdtoi convert double precision floating point to integer

```
fdtoi    freg_rs2, freg_rd
```

fqtoi convert quad precision floating point to integer

```
fqtoi    freg_rs2, freg_rd
```

fstod convert single to double precision floating point

```
        fstod     freg_rs2, freg_rd
```
fstoq convert single to quad precision floating point
```
        fstoq     freg_rs2, freg_rd
```
fdtos convert double to single precision floating point
```
        fdtos     freg_rs2, freg_rd
```
fdtoq convert double to quad precision floating point
```
        fdtoq     freg_rs2, freg_rd
```
fqtos convert quad to single precision floating point
```
        fqtos     freg_rs2, freg_rd
```
fqtod convert quad to double precision floating point
```
        fqtod     freg_rs2, freg_rd
```

Synthetic Instructions

There are certain *synthetic* instructions that the assembler may provide for the convenience of assembly language programmers (or compiler writers). These are intended to provide common operations which an assembly language programmer might expect but which are not directly supported by the machine architecture. The assembler maps these synthetic instructions to less obvious machine instructions or to short sequences of machine instructions. These operations are presented here without a specification of their machine instruction equivalents. This should not be a problem for an introductory compiler writer. Some operations are ommitted to prevent possible trouble.

cmp compare
```
        cmp  reg_rs1, reg_or_imm
```
jmp jump to an address discarding “return address”
```
        jmp  address
```
call call a subroutine and save “return address” in ‘%o7’
```
        call address
```
tst test a value to set condition codes
```
        tst  reg_rs2
```
ret return from subroutine
```
        ret
```
set set register to value without concern for value’s size
```
        set  value,reg_rd
```
not one’s complement
```
        not  reg_rs1, reg_rd
```
not one’s complement in place
```
        not  reg_rd
```
neg two’s complement
```
        neg  reg_rs1, reg_rd
```
neg two’s complement in place
```
        neg  reg_rd
```
inc increment
```
        inc  reg_rd
```
inc increment by amount

```
                inc  const13, reg_rd
```

inccc — increment and set condition codes

```
                inccc     reg_rd
```

inccc — increment by amount and set condition codes

```
                inccc     const13, reg_rd
```

dec — decrement

```
                dec  reg_rd
```

dec — decrement by amount

```
                dec  const13, reg_rd
```

deccc — decrement and set condition codes

```
                deccc     reg_rd
```

deccc — decrement by amount and set condition codes

```
                deccc     const13, reg_rd
```

btst — bit test

```
                btst reg_or_imm, reg_rs1
```

bset — bit set

```
                bset reg_or_imm, reg_rd
```

bclr — bit clear

```
                bclr reg_or_imm, reg_rd
```

btog — bit toggle

```
                btog reg_or_imm, reg_rd
```

clr — clear register

```
                clr  reg_rd
```

clrb — clear byte

```
                clr  [address]
```

clrh — clear halfword

```
                clr  [address]
```

clr — clear word

```
                clr  [address]
```

mov — move data

```
                mov  reg_or_imm, reg_rd
```

Assembly Language Syntax

The SPARC architecture, naturally, does not define an assembly language syntax (its not part of the ISA). Thus, there are many possible assembly language formats that may be implemented in different operating environments running on or targetting the SPARC architecture. The SPARC Architecture Manual does define a “suggested” assembly language syntax and that is what will be used throughout this document. Be aware that syntactic details may change from system to system.

The opcodes are as specified in the preceding section. Register specifications are preceded by the percent sign as in ‘`%i7`’, ‘`%f27`’ or ‘`%r16`’. In addition to the ‘r’, ‘i’, ‘l’, ‘o’, and ‘g’ register specification, the notations ‘`%sp`’ and ‘`%fp`’ may be used to refer to the stack pointer and frame pointer respectively. There are not special purpose registers for the stack and frame pointer. Instead, by convention these registers are mapped to ‘`%o6`’ and ‘`%i6`’ respectively.

The syntax ‘`%hi(32bitValue)`’ and ‘`%lo(32bitValue)`’ is also supported. These expressions extract the high order 22 and low order 10 bits of the 32 bit value (normally an address) specified. Of course, the use of the synthetic ‘`set`’ instruction precludes the need for these in most cases.

Labels are composed of a sequence of characters including letters, digits, underscores, dollar signs, and periods. A label may not begin with a digit but otherwise, there are no restrictions.

References to data at a specified address is made by using the syntax ‘[address]’ while references to addresses themselves omit the square brackets. When more than one source is used to form an address (i.e. two registers or a register and an immediate field) either a ‘+’ or a ‘-’ is placed between them as required.

Comments are introduced using the exclamation point (‘!’) and continue from that point to the end of line.

Assembler directives vary greatly between OS platforms. Consult the “SUN-4 Assembly Language Reference Manual” for details on directives.

Example Code

Knowing the individual details of the SPARC architecture is sufficient information to permit an experienced compiler writer to begin writing a code generator. For a first time student compiler writer however, more information is typically needed. This section presents some example code generated by a compiler for a conventional high level language. While the code is presented, the method of generating the code is not.

The following code was generated by the Gnu C compiler without optimization and is indicative of the results of simple code generation. The original source code is shown and the resulting assembly code has been documented with explanatory notes. The Gnu C compiler (‘gcc’) has an option to enable the generation of assembly code in a file with suffix ‘.s’. This is a useful approach to learning about what kind of code should be generated for various constructs.

```
! The original source code was:
!
!  void main()
!
!  {
!        int   i,j;
!
!        for (i=;i<100;i++) {
!              j=i*3+44;
!              if (j>60) {
!                    j--;
!              } else {
!                    j++;
!              }
!              printf("j is %d.\n",j);
!        }
!  }
!
! The generated SPARC assembly code is:
!
```

```
gcc2_compiled.:
___gnu_compiled_c:
.text
        .align 8
LC0:
        .ascii "j is %d.\12\0"  ! string literal - printf
        .align 4
        .global _main
        .proc 020
_main:                          ! code entry point
        !#PROLOGUE# 0
        save %sp,-120,%sp       ! reserve stack space
        !#PROLOGUE# 1
        call ___main,0
        nop                     ! branch delay slot
        st %g0, [%fp-20]        ! zero 'i'
L2:                             ! top of loop
        ld [%fp-20],%o0         ! load 'i'
        cmp %o0,99              ! test against loop bound
        bg L3                   ! branch if done
        nop                     ! branch delay slot
        ld [%fp-20],%o0         ! get 'i' again
        move %o0,%o2
        sll %o2,1,%o1           ! multiple by two and
        add %o1,%o0,%o1         ! add one to get multiply by three
        add %o1,44,%o0          ! add 44 to that
        st %o0,[%fp-24]         ! and store it in 'j'
        ld [%fp-24],%o0         ! load 'j'
        cmp %o0,60              ! test in if (is it greater than 60)
        ble L5                  ! if less or equal do ELSE part
        nop                     ! branch delay slot
                                ! the THEN part of the IF
        ld [%fp-24],%o1         ! load 'j'
        add %o1,-1,%o0          ! 'j--'
        mov %o0,%o1
        st %01,[%fp-24]         ! store result
        b L6                    ! branch around ELSE part of the IF
        nop                     ! branch delay slot
L5:
        ld [%fp-24],%o1         ! load 'j'
        add %o1,1,%o0           ! 'j++'
        mov %o0,%o1
        st %o1,[%fp-24]         ! store result
L6:                             ! end of IF
        sethi %hi(LC0),%o1      ! setup for call to printf
        or %o1,%lo(LC0),%o0
        ld [%fp-24],%o1         ! point to 'j' for printf
        call _printf,0
        nop                     ! branch delay slot
L4:
        ld [%fp-20],%o1         ! load 'I'
        add %o1,1,%o0           ! increment loop index 'I'
        mov %o0,%o1
        st %o1,[%fp-20]         ! store 'I'
        b L2                    ! back to top of loop
        nop                     ! branch delay slot
L3:
L1:
        ret                     ! return to O/S (executed as s/r)
        restore                 ! restore stack (in delay slot)
```