INTRODUCTION TO
SPARC ASSEMBLY LANGUAGE

SOURCE CODE FORMAT

A SPARC assembly language source code file is composed of zero or more lines of text, where each line may contain the following elements:

label: instruction ! comment

All three of the elements are optional. Thus, blank lines are acceptable.

A label is a symbol which must begin with a letter or a period; subsequent characters must be letters, digits, or underscores. A colon is used to mark the end of the label field.

An instruction must be one of the following:

-- a symbolic machine instruction
-- a synthetic instruction
-- an assembler directive

The synthetic instructions and the symbolic machine instructions recognized by most translators are listed in Appendix A and Appendix B of the SPARC Architecture Manual; commonly used assembler directives are listed below.

A comment begins with an exclamation point and continues to the end of that line of text. In addition, some translators accept C-style comments.

Most instructions require one or more operands, including:

-- symbols (normally labels)
-- special symbols (normally register designators)
-- literal constants

Special symbols begin with a percent sign and usually designate a register:

-- general-purpose registers %r0 -- %r31
-- floating-point registers %f0 -- %f31
-- general-purpose global registers %g0 -- %g7 (same as %r0 -- %r7)
-- general-purpose out registers %o0 -- %o7 (same as %r8 -- %r15)
-- general-purpose local registers %l0 -- %l7 (same as %r16 -- %r23)
-- general-purpose in registers %i0 -- %i7 (same as %r24 -- %r31)
-- stack-pointer register %sp (same as %o6)
-- frame-pointer register %fp (same as %i6)

Most translators accept C-style literal constants, including:

-- octal integer constants
-- decimal integer constants
-- hexadecimal integer constants
-- character constants

In addition, most translators recognize a set of operators which can be applied to operands in constant expressions. The most useful operators are:

-- %lo(operand) extract the least significant 10 bits of the operand
-- %hi(operand) extract the most significant 22 bits of the operand
ASSEMBLER DIRECTIVES

Most translators which process SPARC assembly language source files recognize a rich set of assembler directives. The directives which are most useful to the assembly language programmer are listed below.

<table>
<thead>
<tr>
<th>directive (and operands)</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.global  symbol</td>
<td>Declares the symbol to be global (the symbol is defined externally, or the symbol is defined internally and is accessible externally).</td>
</tr>
<tr>
<td>.section &quot;name&quot;</td>
<td>Makes the named section the current section.</td>
</tr>
<tr>
<td>.align  N</td>
<td>Aligns the location counter on the next N-byte boundary. The value of N must be a power of 2.</td>
</tr>
<tr>
<td>.skip  N</td>
<td>Increments the location counter by N, which allocates N bytes in the current section.</td>
</tr>
<tr>
<td>.ascii  &quot;string&quot;</td>
<td>Generates the sequence of ASCII characters.</td>
</tr>
<tr>
<td>.asciz  &quot;string&quot;</td>
<td>Generates the sequence of ASCII characters and appends a null byte.</td>
</tr>
<tr>
<td>.byte  I [,I]*</td>
<td>Generates the 8-bit internal representation of each integer value in the sequence.</td>
</tr>
<tr>
<td>.half  I [,I]*</td>
<td>Generates the 16-bit internal representation of each integer value in the sequence.</td>
</tr>
<tr>
<td>.word  I [,I]*</td>
<td>Generates the 32-bit internal representation of each integer value in the sequence.</td>
</tr>
<tr>
<td>.single  0rR [,0rR]*</td>
<td>Generates the 32-bit internal representation of each floating point value in the sequence.</td>
</tr>
<tr>
<td>.double  0rR [,0rR]*</td>
<td>Generates the 64-bit internal representation of each floating point value in the sequence.</td>
</tr>
</tbody>
</table>

In addition to these directives, most translators allow the programmer to define symbolic constants, as shown below:

```
MAX = 20
```

These symbolic constants can be used to replace literal constants. For example, the following two instructions are identical (assuming that "MAX" is defined as shown above):

```
cmp  %10, MAX
cmp  %10, 20
```

Symbolic constants make it easier to maintain source code files, and should be used whenever possible.
FLOATING POINT PROGRAMMING

The SPARC floating point processing unit (FPU) supports two floating point data formats that are of interest to the C and assembly language programmer: single precision (32-bit reals) and double precision (64-bit reals).

The FPU contains 32 floating point registers, each of which is 32 bits in length. Two consecutive registers are used to hold double precision values, where the first register must be an even-numbered register. The floating point registers are global; they are not windowed via the SAVE and RESTORE instructions.

The following assembly language instructions can be used to perform either single or double precision arithmetic operations. Replacing "s" with "d" specifies double precision format instead of single precision.

- fadds %fx, %fy, %fz       Add register fx to fy, place result in fz
- fsubs %fx, %fy, %fz       Subtract register fy from fx, place result in fz
- fmuls %fx, %fy, %fz       Multiply register fx by fy, place result in fz
- fdivs %fx, %fy, %fz       Divide register fx by fy, place result in fz
- fsqrts %fx, %fy           Compute square root of fx

The following instructions can be used to perform type conversions:

- fitos %fx, %fy        Convert integer to single precision
- fstoi %fx, %fy        Convert single precision to integer
- fitod %fx, %fy        Convert integer to double precision
- fdtoi %fx, %fy        Convert double precision to integer
- fstod %fx, %fy        Convert single to double precision
- fdtos %fx, %fy        Convert double to single precision

The following instructions can be used to compare two values and set the FPU condition code bits:

- fcmps %fx, %fy       Compare register fx to fy (single precision)
- fcmpd %fx, %fy       Compare register fx to fy (double precision)

The FBCC instructions permit conditional branching based on the FPU condition code bits:

- fba       Branch Always
- fbn       Branch Never
- fbu       Branch on Unordered
- fbg       Branch on Greater
- fbug      Branch on Unordered or Greater
- fbl       Branch on Less
- fbul      Branch on Unordered or Less
- fblg      Branch on Less or Greater
- fbne      Branch on Not Equal
- fbe       Branch on Equal
- fbue      Branch on Unordered or Equal
- fbge      Branch on Greater or Equal
- fbuge     Branch on Unordered or Greater or Equal
- fble      Branch on Less or Equal
- fbule     Branch on Unordered or Less or Equal
- fbo       Branch on Ordered

Note that at least one non-FPU instruction must be executed between an FCMP instruction and an FBCC instruction.
The following instructions can be used to perform special-purpose operations:

- `fmovs %fx, %fy`: Copy contents of fx to fy
- `fnegs %fx, %fy`: Complement upper bit of fx and copy to fy
- `fabss %fx, %fy`: Clear upper bit of fx and copy to fy

Movement of data values between floating point registers and memory is accomplished using LOAD and STORE instructions:

- `ld [adr], %fx`: Load single precision value from memory
- `ldd [adr], %fx`: Load double precision value from memory
- `st %fx, [adr]`: Store single precision value into memory
- `std %fx, [adr]`: Store double precision value into memory

The conventions used as the interface between program units are similar to those used for integer operands: all arguments (including floating point operands) are passed to functions in the integer registers, which are managed using the SAVE and RESTORE instructions. However, a floating point value is returned from a function in register f0, rather than register i0/o0.