SPARC Architecture

The SPARC architecture is called a load/store (RISC) architecture. This means that only the load and store instructions can access memory locations. All other instructions access only registers. There are 32 registers available at any one time.

Registers

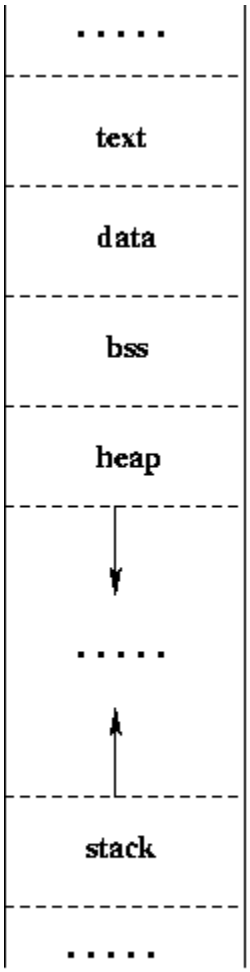
Registers are indicated by a leading percent sign, "%".

%g0 == %r0 (always zero)	g stands for global. Except for %g0 always being 0
%g1 == %r1	the other 7 registers are for data with a global context.
...	
%g7 == %r7	
%o0 == %r8	o stands for output, note not 0. First six registers are for
...	local data and arguments used to call subroutines.
%o6 == %r14	%sp (Stack Pointer)
%o7 == %r15	Called subroutine's return address
%l0 == %r16	l stands for local, note not 1. All eight registers are used
...	for local variables.
%l7 == %r23	
%i0 == %r24	i stands for input. First six registers are for incoming
...	subroutine arguments.
%i6 == %r30	%fp (Frame Pointer)
%i7 == %r31	Subroutine's return address

Note: Do not use %o6, %o7, %i6, or %i7.
Use %g1 - %g7 for your global variables.
Use %l0 - %l7 for your local variables.

Program Layout in Memory

The following graphic depicts how your program resides in memory



Program's Layout
In Memory

Program sections reflect the different logical parts of a program.

text	program code
data	initialized variables
bss	(block started by symbol) uninitialized variables
stack	automatically allocated variables, local variables, and other stack frame entries
heap	dynamically allocated variables

Pseudo-ops can be used to identify the sections in your assembly source code.

.section ".text"
This is where your program code goes.

.section “.data”
This is where your variables that have initial values go.

.section ".bss"
This is where your variables could go.

.word
Followed by a list of initialized values

.asciz
Null-terminated character string, useful for strings when using routines from the C library like printf and scanf.

.skip
Useful for declaring space for a variable or an array.

.align 4
Variables of word size 32 must be aligned, i.e. .align 4.

Skeleton for SPARC Assembly Language Program

Execution always starts at the label "main", just like C programs. An initial save instruction provides a new register window. End by a normal procedure exit of return and restore instructions.

```
        .global main
main:   save    %sp, -120, %sp

        ... your assembly language code ...

        ret
          restore
```

Statement Syntax

```
<label:>        mnemonic_opcode         operands        ! comments
```

C-style comments, i.e. /* and */, are acceptable. ! comments are also acceptable. Some additional formats are

```
<op>  src_reg,  src_reg,  dest_reg

<op>  src_reg,  immediate,  dest_reg    ! -4096 <= immediate < 4096
```

where <op> is the operation code, src_reg is a source register, dest_reg is the destination register, and immediate is a small integer constant. See below for a brief listing of some of the SPARC operation codes.

There are also synthetic instructions such as

```
clr  reg
mov  immediate, dest_reg
mov  src_reg,  dest_reg
```

```
set  value,  dest_reg
```

Other statements, called "pseudo-ops", are available to control the assembler's actions or to specify data. These instructions generally start with a period. Specifically,

```
            .section  ".data"
            .global     <label>
            .align 4
<label:>     .word      <argument(s)>    ! could be several arguments
```

Instructions

Load and Store

You will need to use the load instruction to get a value from a memory location into a register and a store instruction to get a value from a register to a memory location. Since we are only dealing with full word entities, the appropriate format for the load is

```
ld  [ memory location ],  dest_reg
```

Note: The brackets, i.e. [‘s and]’s, must surround the memory location.

For the store instruction we have

```
st   src_reg,  [ memory location ]
```

Note: The brackets, i.e. [‘s and]’s must surround the memory location.

Basic Arithmetic Operations

The basic arithmetic operations are add, subtract, multiple, and divide. Add and subtract can be achieved directly as

```
add  src_reg,  src_reg,  dest_reg
```

or

```
add src_reg,  immediate,  dest_reg
```

The subtract is

```
sub  src_reg,  src_reg,  dest_reg
```

or

```
sub  src_reg,  immediate,  dest_reg
```

The multiple and divide require a call. For example, if we wanted to multiple the contents of %o0 with %o1, then we would have

```
call  .mul
  nop
```

The results are returned in register %o0.

Note: The operands must be in registers %o0 and %o1.

For divide we would have

```
call   .div
  nop
```

The results are returned in register %o0.

Note: The operands must be in registers %o0 and %o1.

Logical Operations

The three logical operations that you will need are

and or not

Branches

Some of the branching instructions are:

ba -- branch always

testing result of compare or other operation (signed arithmetic):

bl -- branch on less than
ble -- branch on less than or equal
be -- branch on equal
bne -- branch on not equal
bge -- branch on greater than or equal
bg -- branch on greater than

Note: Always put a nop after each and every branch.

Control Structures in C and Their Assembly Language Templates

Note: In the examples below %a_r is the register that the variable a is in, %b_r is the register that the variable b is in, and %c_r is the register that the variable c is in.

while loop

```
                        !  WHILE LOOP
while ( a <= 17 ) {     L$17                    !  (L$17, LABEL, -, -)
  ….                        cmp  %a_r, 17       !  (B$15, <=, a, 17)
}                           bg   L$18           !  (L$18, CJUMP, B$15, -)
                              nop
                            ….
                            ba   L$17           !  (L$17, JUMP, -, -)
                              nop
                        L$18:                   !  (L$18, LABEL, -, -)
```

if-then

```
                          !  IF THEN
 if ( (a + b) > c ) {          add  %a_r, %b_r, %o0        !  (I$22, IADD, a, b)
   ….                          cmp  %o0, %c_r              !  (B$11, GT, I$22, c)
 }                             ble  L$15                   !  (L$15, CJUMPF, B$11, -)
                                 nop

                               ….
                          L$15:                            !  (L$15, LABEL, -, -)
```

if-then-else

```
                          !  IF THEN ELSE
 if ( a != b ) {               cmp  %a_r, %b_r      !  ( B$23, NE, a, b)
   ……                          be   L$14            !  ( L$14, CJUMPF, B$23, -)
 } else {                        nop
   ….                          ……
 }                             ba  L$15             !  ( L$15, JUMP, -, -)
                                 nop
                          L$14:                     !  (L$14, LABEL, -, -)
                               ….
                          L$15:                     !  (L$15, LABEL, -, -)
```

printf

You can call printf from a SPARC assembly language program. The following skeleton can be used. Be sure that you use registers %o0 and %o1 for the call to printf.

```
main: save  %sp, -120, %sp

      <your program goes here>

      set   fmt, %o0                    ! what kind of data you want to print
      mov   <register to print>, %o1    ! what value you want to print
      call  printf                      ! now print it
       nop

      <rest of your program goes here>

      ret
       restore


      .section ".rodata"
fmt:  .asciz "%d\n"
```

scanf

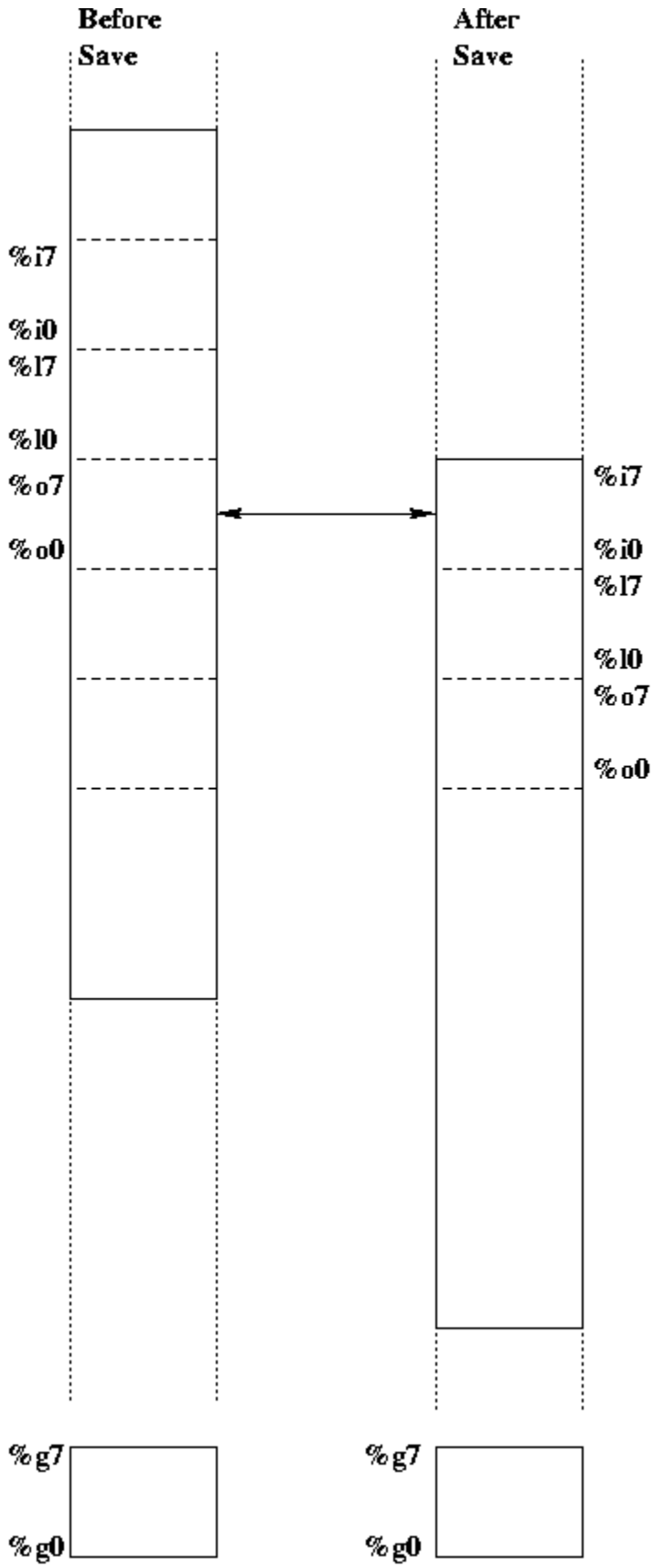
You can call scanf from a SPARC assembly language program. The following skeleton can be used. Be sure that you use registers %o0 and %o1 for the call to scanf.

```
 main: save  %sp, -120, %sp

        <your program goes here>

        set  format,  %o0                ! what kind of data you want to get
        set  input,  %o1                 ! location for the input number to be stored
        set  nl,  %o2                    ! location to dump the input newline
        call  scanf                      ! now the user input is in the data block
          nop                            ! specified by the label input and nl

        <rest of your program goes here>

        ret
          restore

        .section  ".data"
input:  .word  0

           .section  ".rodata"
 format:  .asciz  "%d\n"
 nl:      .asciz   "\n"
```

Subroutines

SPARC Registers

The SPARC registers are composed of 8 global registers that are available at any time and 24 registers that are available to a procedure. These 24 registers are in a register window that gets shifted by 16 registers for each procedure call and return. The graphic below depicts the SPARC registers.

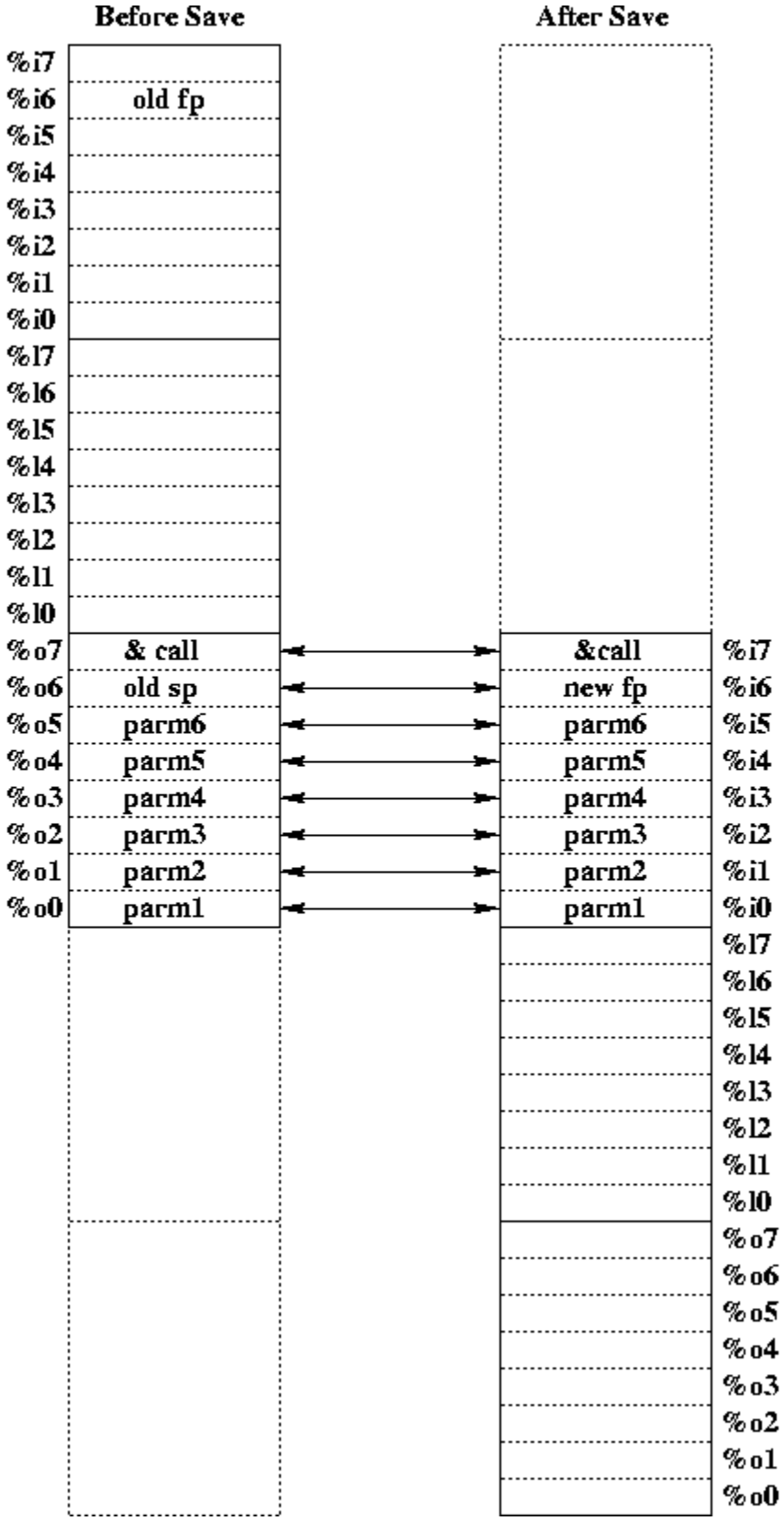


The save instruction changes the register mapping. The restore instruction changes the register mapping back to the previous mapping. If we have register window overflow, the registers will be written to memory, i.e. the stack, automatically.

Register Usage Conventions

%i0	incoming parameter, also used to pass return value back
%i1-%i5	additional incoming parameters
%i6	frame pointer
%i7	address of call instruction, return to %i7 + 8 if saved
%l0-%l7	all yours
%o0	all yours, often used as temp register, but also used to pass a parameter to and get a return value back from a subroutine.
%o1-%o5	all yours, but also used to pass parameters to a subroutine
%o6	stack pointer
%o7	address of call instruction, return to %o7 + 8 if no save

A much more detailed diagram is below. The subroutine cannot access the registers, %i0 thru %i7 and %l0 thru %l7 of the calling program. The subroutine gets a fresh set of local and output registers. The input registers in the subroutine overlap the output registers in the calling routine. The return instruction will return to %i7 + 8. The old sp becomes the new fp in the subroutine. Up to 6 parameters can be passed through the registers. %o7 in the subroutine is available for saved address if the subroutine has any nested calls. The new sp is equal to the old sp plus the value in the save instruction. %o0 thru %o5 are available for parameters if the subroutine calls another subroutine.


Before Save
After Save
%i7
%i6
old fp
%i5
%i4
%i3
%i2
%i1
%i0
%l7
%l6
%l5
%l4
%l3
%l2
%l1
%l0
%o7
& call
%o6
old sp
%o5
parm6
%o4
parm5
%o3
parm4
%o2
parm3
%o1
parm2
%o0
parm1
&call
%i7
new fp
%i6
parm6
%i5
parm5
%i4
parm4
%i3
parm3
%i2
parm2
%i1
parm1
%i0
%l7
%l6
%l5
%l4
%l3
%l2
%l1
%l0
%o7
%o6
%o5
%o4
%o3
%o2
%o1
%o0

Typical SPARC call

```
   /* pre-call */
      move parameter1 to %o0
      move parameter2 to %o1
      call subroutine            /* places address of call instruction in %o7 */
       nop                       /* call requires a delay slot instruction    */

                        subroutine:
                          /* prologue */
                            save %sp,-120,%sp

                          /* body */

                            body of subroutine, in which you access parameters in %i0, %i1, ...,
                             and register allocated local variables in %l0, %l1, ...

                          /* epilogue */
                            ret
                             restore

   /* post-call */
      /* empty */
```

Pass By Value Example

The following contains some sample c-like code and the equivalent SPARC assembly code that does pass by value.

```
C code                    equivalent assembly code
------                    ------------------------

int a;                            .section ".bss"      ! .bss since a is not
                                  .align 4             ! assigned an initial
                          a:      .skip 4              ! value

                                  .section ".text"
int main(void){           main:   save  %sp, -120, %sp ! standard prologue

  a = 0;                          set   a, %l0         ! a = 0
                                  st    %g0, [%l0]     !

  a = sub( a );                   ld    [%l0], %o0     !\
                                  call  sub            ! | sub( a ) by value
}                                  nop                 !/

                                  st    %o0, [%l0]     ! a = return value

                                  ret
                                   restore


int sub( int x ){         sub:    save, %sp, -120, %sp   ! standard prologue

  return( x + 1);                 add   %i0, 1, %i0    ! return value = x + 1
}
                                  ret                  ! note that first parm and return
                                    restore            ! value both use reg %i0
```

Pass by Reference Example

The following contains some sample c-like code and the equivalent SPARC assembly code that does pass by reference.

```
C code                  equivalent assembly code
------                  ------------------------

int a;                          .section ".bss"      ! .bss since a is not assigned
                        a:      .skip 4              ! an initial value

                                .section ".text"
int main(void){         main:   save  %sp, -120, %sp

  a = 0;                        set   a, %l0         ! a = 0
                                st    %g0, [%l0]     !

  a = sub( &a );                mov   %l0, %o0       !\
                                call  sub            ! | sub( &a ) by ref
}                                nop                 !/

                                st    %o0, [%l0]     ! a = return value

                                ret
                                 restore

int sub( int *x ){      sub:    save  %sp, -120, %sp  ! standard prologue

  return( *x + 1 );             ld    [%i0], %l1     ! deref first parm
}                               add   %l1, 1, %i0    ! return value = *x + 1

                                ret                  ! note that first parm and return
                                 return              ! value both use reg %i0 and %l1
```

Recursion

The following example is a recursive Fibonacci Sequence. Briefly, the Fibonacci Sequence is defined recursively as

Fib(0) = 1

Fib(1) = 1

Fib(n) = Fib(n-1) + Fib(n-2)

We will assume that n is non-negative and is in register %o0 in the calling routine.

```
!       Recursive Fibonacci routine
!

fib:    save    %sp, -120, %sp          ! Prologue

        cmp     $i0, 1                  ! test for terminating condition
        bg      then                    ! recursive call
         nop

        add     %g0, 1, %i0             ! return 1

        ret                             ! Epilogue
         restore


  then:
        sub     %i0, 1, %o0             ! decrement n by 1
        call    fib                     ! call with n-1
         nop

        mov     %o0, %l0                ! save results in %l0

        sub     %i0, 2, %o0             ! decrement n by 2
        call    fib                     ! call with n-2
         nop

        add     %l0, %o0, %i0           ! add fib(n-1) + fib(n-2)

        ret
         restore
```