Calling an Assembly funciton from a C Program using the SPARC ABI

Sanjose Mary[1] and Gauri Patrikar[1]

[1]Project Research Engineer, IITB

July 27, 2019

Abstract

This document discusses ABI (application binary interface), which defines the standard function calling sequence. Here, we specifically call an assembly language program from a C program. A calling sequence includes steps to ensure seamless transition from one function to another without any loss of important data or any situation causing traps.

We will first take a brief look into the various registers involved and their specified roles. The stack memory organization is also discussed. We then look into the rules specified the SPARC ABI, followed by some examples.

This a document giving steps to follow to call an assembly program from a C one and for further information the SPARC ABI will need to be referred. It has been mentioned in the document wherever things get out of scope.

1 Introduction

We will refer to the C program as the caller, as it is the one calling the assembly program, which in turn is referred to as the callee.

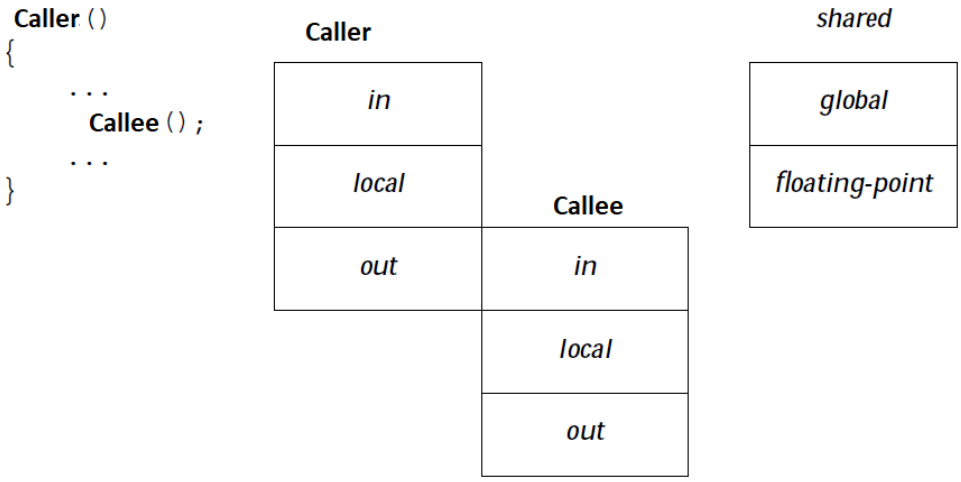

Figure 1: Register Windows

1.1 Registers

The SPARC architecture provides windowed integer registers (in, out, local), global integer registers and floating-point registers. At any time, there are 32 integer and 32 floating point registers visible to the program. The integer registers are arranged as follows-

- g0-g7 — for global data
- o0-o7 — for arguments to a subroutine
- l0-l7 — for local data
- i0-i7 — for incoming arguments

Table 1 gives the layout of all visible registers and their specific roles if any.

1.2 *In* and *Out* Register

Here, we are assuming that the callee will have its own register window. It will do so by executing a SAVE instruction, which shifts a register window.

in	i7(r31)	return address-8
	fp, i6(r30)	frame pointer
	i5(r29)	incoming parameter 6
	i4(r28)	incoming parameter 5
	i3(r27)	incoming parameter 4
	i2(r26)	incoming parameter 3
	i1(r25)	incoming parameter 2
	i0(r24)	incoming parameter 1/return value to caller
local	l7(r23)	local 7
	l6(r22)	local 6
	l5(r21)	local 5
	l4(r20)	local 4
	l3(r19)	local 3
	l2(r18)	local 2
	l1(r17)	local 1
	l0(r16)	local 0
out	o7(r15)	temporary value/ address of CALL instruction
	sp, o6(r14)	stack pointer
	o5(r13)	outgoing parameter 6
	o4(r12)	outgoing parameter 5
	o3(r11)	outgoing parameter 4
	o2(r10)	outgoing parameter 3
	o1(r9)	outgoing parameter 2
	o0(r8)	outgoing parameter 1/return value from callee
global	g7(r7)	global 7(SPARC ABI: use reserved)
	g6(r6)	global 6(SPARC ABI: use reserved)
	g5(r5)	global 5(SPARC ABI: use reserved)
	g4(r4)	global 7(SPARC ABI: global register variable)
	g3(r3)	global 7(SPARC ABI: global register variable)
	g2(r2)	global 7(SPARC ABI: global register variable)
	g1(r1)	temporary value
	g0(r0)	0
floating point	f31	floating point value
	.	.
	.	.
	.	.
	.	.
	f0	floating point value

Table 1: Registers

When a register window shifts, the second function gets its own local and out registers, but the outs of the previous window become the ins of the new register window, i.e., the two windows share 8 registers. This is very useful for parameter passing, as outgoing parameters are stored in the out registers of the caller, which the callee will find its in registers.

Figure 1 shows windowing of registers.

The *in* and *out* registers are used primarily for passing parameters to subroutines and receiving results from them.

- o0-o5 of the caller—Up to six parameters passed by placing them in out registers o0 to o5, additional parameters are passed in the memory stack.
- o6 of the caller—The stack pointer is implicitly passed in o6 of the caller. It points to an area in which the system can store r16 to r31 (l0 to l7 and i0 to i7) when the register file overflows (window overflow trap), and is used to address most values located on the stack.
- o7 of the caller—a CALL instruction places its own address in o7.

A procedure may store temporary values in its *out* registers, with the exception of sp, which are volatile across procedure calls. sp cannot be used for temporary values.

After a callee is entered and its SAVE instruction has been executed, as the register window shifts, the caller's stack pointer sp(o6 of the caller) automatically becomes the current procedure's frame pointer fp(i6 of the callee). If there are more than 6 arguments, then they are stored in the memory stack, from where they can be accessed.

1.3 Local Registers

The locals are used for automatic variables, and for most temporary values. For access efficiency, a compiler may also copy parameters (i.e. those past the sixth) from the memory stack into the locals and use them from there.

1.4 Global Registers

The globals are a set of eight registers with global scope. The globals(except g0) are conventionally assumed to be volatile across procedure calls.The

global registers, other than g0, can be used for temporaries, global variables, or global pointers — either user variables, or values maintained as part of the program's execution environment.

- g0—hardwired value of zero. It always reads as zero, and writes to it have no effect.
- g1—volatile across procedure calls.
- g2-g4—reserved for use by the application program.
- g5-g7— nonvolatile and reserved for (as-yet-undefined) use by the execution environment.

1.5 Floating-Point Registers

There are thirty-two 32-bit floating-point registers. Floating-point registers are accessed with different instructions than the integer registers, their contents can be moved among themselves, and to or from memory.

Across a procedure call, either the caller must save its live floating-point registers, or the callee must save the ones it is going to use and restore them before returning. Current compilers use the caller-save convention.

2 The Memory Stack

Every function gets its own to stack space. We allocate stack space at the start of the program itself. Figure 2 shows the stack frame organization.

In this figure, the stack frame points to the bottom most address of the current stack and frame pointer points to the topmost address of the current stack.

Also, the current frame pointer of the callee is the stack pointer of the caller.

The following are always allocated at compile time in a procedure's stack frame (values in brackets give the starting address):

- 16 words, always starting at sp, for saving the procedure's in and local registers, should a register window overflow occur (sp+0).

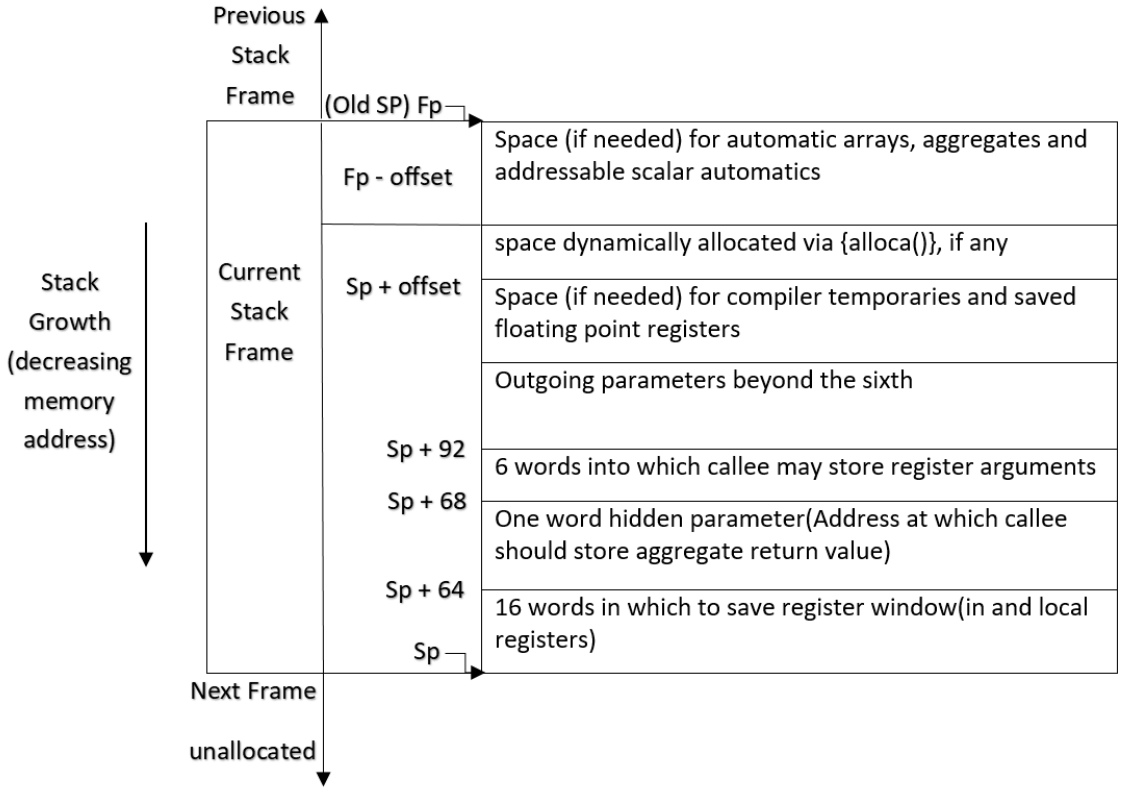

Figure 2: Stack Frame Organization

- One word, for passing a hidden (implicit) parameter. This is used when the caller is expecting the callee to return a data aggregate by value; the hidden word contains the address of stack space allocated (if any) by the caller for that purpose (sp+64).
- 6 words into which callee can write outgoing parameters 0 to 5(sp +68).
- Outgoing parameters beyond the sixth (sp+92).
- Compiler-generated temporary values (typically when there are too many for the compiler to keep them all in registers)
- Floating-point registers being saved across calls (occurs if floating-point instructions are used by a procedure)
- All automatic arrays, automatic data aggregates, automatic scalars which must be addressable, and automatic scalars for which there is no room in registers

Space can be allocated dynamically (at runtime) in the stack frame for the fol- lowing:

- Memory allocated using the alloca() function of the C library

Addressable automatic variables on the stack are addressed with negative offsets relative to fp, dynamically allocated space is addressed with positive offsets from the pointer returned by alloca(), everything else in the stack frame is addressed with positive offsets relative to sp. Incoming arguments are stored in the previous stack frame, which we will reference from positive offsets of frame pointer.

The stack pointer sp must always be doubleword-aligned. This allows window overflow and underflow trap handlers to use the more efficient STD and LDD instructions to store and reload register windows.

3 Passing and Returning Arguments

3.1 Integer Arguments

The first six input arguments are received through the in registers of the callee, and the rest of the arguments are passed through the stack. Functions

pass all integer-valued arguments as words, expanding signed or unsigned bytes and halfwords as needed.

The output is kept in the i0 register of the callee, the caller finds the value in o0.

3.2 Floating point Arguments

The in registers also hold the input floating point arguments, single precision need one register and double precision need two registers. As the floating point operations cannot use integer registers, compilers usually store them in the stack.

A floating point return value appears in the floating point registers for both the caller and the callee. Single precision values appear in f0, while double precision in f0 and f1.

4 Prologue and Epilogue

Each function that has to be called has a set epilogue and prologue.

The prologue is as follows-

```
SAVE %sp, -96, %sp
```

Here, with the SAVE instruction, we are shifting to a new register window, making the out of the callers, the in of the callees, and giving the callee its own register window. Also we are allocating stack space to the callee. The stack size can be variable.

The epilogue is as follows-

```
RESTORE
RETL
NOP
```

Here, the RESTORE instruction restores the previous register window of the callee. And the RETL instruction returns control to the caller.

5 Coding Examples

5.1 Passing Two Integer Arguments

In this example, we will be adding two integers. Here, the C program is the caller, giving the two values to be added. The assembly program is the one being called, doing the addition and returning a scalar value.

C code:- main.c

```
int add(int a, int b);
int main(){
    return add(5,7);
}
```

SPARC Assembly code:- add.s

```
.section \.text"
.align 4
.global add
.type add, #function
add:
    save %sp,-96, %sp
    add %i0, %i1, %g1
    mov %g1, %i0
    restore
    retl
    nop
```

Summary

- Two register windows are being used, when save is executed the register window will be shifted. The callee will hence find the input in i0 and i1.
- The output will be again stored in i0 of the callee. When restore is executed the control will transfer to the previous window, and the calleer will find the ouput in its o0 register.

5.2 Passing more than 6 integer arguments

Here we will be adding 7 numbers. This will help in understanding stack utilization, as the 7th argument will reside in the stack.

C code:- main.c

```
int add(int a, int b, int c, int d, int e, int f, int g);
int main(){
    int a=5;
    int b=7;
    int c=2;
    int d=1;
    int e=3;
    int f=4;
    int g=2;
    return add( a,b,c,d,e,f,g );
}
```

Assembly code:- add.s

```
.section ".text"
.align 4
.global add
.type add, #function
add:
    save %sp, -96, %sp
    add %i0, %i1, %g1
    add %g1, %i2, %g2
    add %g2, %i3, %g3
    add %g3, %i4, %g4
    add %g4, %i5, %g5
    ld [ %fp + 92 ], %l0
    add %g5, %l0, %g5
    mov %g5, %i0
    restore
    retl
    nop
```

Summary

- The first 6 arguments are being accessed from the registers, but the seventh argument is being taken from the stack.
- The stack is being accessed from offsets of the frame pointer.
- The other steps are the same as that explained previously.

5.3 Passing two floating point arguments

In this program we will add two floating point numbers. The floating point numbers need to be accessed from the memory directly.

```
C code:- main.c
  extern float add(float a, float b);

float main()
{
float a=2.5;
float b=5.8;
return add(a,b);
}

Assembly code:- add.s
  .text
.align 4
.global add
.type add, #function

add:
        save  %sp, -104, %sp
        ld [%fp+100], %f2
        ld [%fp+96], %f1
        fadds %f2,%f1,%f0
        restore
        retl
        nop
```

Summary

- The input is stored in memory, from where we are loading it into the floating point register f0. The same procedure is followed for f1 register.
- The result is stored f0. Both the caller and callee will find the result in fo itself, as it is a shared register.

5.4 Passing more floating point arguments

In this program we will more or seven floating point numbers. The floating point numbers need to be accessed from the memory directly.

```
C code:- main.c
extern float add(float a, float b, float c, float d, float e, float f, float g);

float main()
{
        float a=1.1;
        float b=1.2;
float c=1.3;
        float d=1.4;
float e=1.5;
        float f=1.6;
float g=1.7;
        return add(a,b,c,d,e,f,g);
}

Assembly code:- add.s
.text
.align 4
.global add
.type add, #function

add:
        save  %sp, -96, %sp
        ld [%fp+128], %f2
        ld [%fp+132], %f1
        fadds %f2,%f1,%f0
ld [%fp+124], %f1
fadds %f0, %f1, %f0
ld [%fp+120], %f1
fadds %f0, %f1, %f0
ld [%fp+116], %f1
fadds %f0, %f1, %f0
ld [%fp+112], %f1
fadds %f0, %f1, %f0
```

```
ld [%fp+108], %f1
fadds %f0, %f1, %f0
        restore
        retl
        nop
```

6 Conclusion

In this document we have explained the usage of the SPARC ABI specifically for calling an assembly function. Following points have been summarized-

- Usage of various registers and register windows.
- Usage of stack memory.
- Argument passing, i.e., how the assembly language will receive the input values, and how the C program will access the output values.

References

[1] SYSTEM V APPLICATION BINARY INTERFACE
SPARC Processor Supplement,
https://www.gaisler.com/doc/sparc-abi.pdf

[2] SPARC Architecture Manual
Version 8,
https://www.gaisler.com/doc/sparcv8.pdf

[3] SPARC Assembly Language Reference Manual,