A Laboratory Manual for the SPARC

Arthur B. Maccabe
Jeff Vandyke
Department of Computer Science
The University of New Mexico
Albuquerque, NM 87131

January 16, 1996

Introduction

This laboratory manual was developed to provide a "hands on" introduction to the SPARC architecture. The labs are based on ISEM, an instructional SPARC emulator developed at the University of New Mexico.

The ISEM package is available via anonymous ftp. To obtain a copy ftp to cs.unm.edu and cd to pub/ISEM. The README file in this directory should provide you with the information needed to obtain a working copy for your environment. ISEM currently runs on most Unix boxes., There are plans to port ISEM to the DOS/Windows environment as well as the Mac. If you have any difficulty getting a copy of ISEM or would like more information regarding the status of the ports, send email to isem@cs.unm.edu.

In addition to an instruction set emulator for the SPARC, the ISEM package includes emulations for several devices (a character mapped display, a bitmapped display, a UART, etc.), an assembler, and a linker. The assember is a slightly modified version of the GNU assembler (gas version 2.1.1). The primary modification is the addition of several synthetic operations to support loads and stores from/to arbitrary locations in memory. These operations are described in the first few laboratory write ups.

The lab maunal is complete in that it covers all of the SPARC operations and instruction formats. As such, students should not require individual copies of *The SPARC Architecture Manual*. However, we have found it useful to have copies of the SPARC Architecture Manual available to students on a reference basis.

This lab manual has been designed to accompany *Computer Systems: Architecture, Organization, and Programming* by Maccabe (Richard D. Irwin, 1993). However, the manual does not directly reference the text and, as such, could be used with other text books.

Laboratory 1
Using ISEM (the Instructional SPARC Emulator)

1.1 Goal

To describe the translation of assembly language programs and introduce basic features of ISEM, the instructional SPARC emulator.

1.2 Objectives

After completing this lab, you will be able to:

- Assemble and link SPARC programs,
- Load SPARC programs,
- Run SPARC programs,
- Examine/modify memory locations, and
- Examine/modify registers.

1.3 Discussion

In this lab we describe the translation process and introduce the basic features of ISEM. We begin by describing the translation process: the steps used to translate an assembly language program into an executable program. After describing the translation process, we describe how you can execute and test executable programs using ISEM.

The first thing you need is a simple SPARC program. Figure 1.1 illustrates a simple SPARC program. We will use this program in the remainder of this lab.

Activity 1.1 *Using a text editor, enter the program shown in Figure 1.1 into a file called* foo.s.

1.3.1 Assembling and Linking Programs

Once you have a SPARC program, you will need to assemble and link your program before you can run it using ISEM. The assembler translates *assembly language programs* into *object program*. The linker transforms object programs into *executable programs* that you can execute using ISEM

To assemble a SPARC program, you need to specify the file that holds the assembly language program and the file that will hold the resulting object program. Assembly commands start with the name of the assembler, *isem-as*, followed by (optional) listing specifications, followed by an output file specification, followed by the name of the source file.

```
        .data                   ! variables
x:      .word 0x42              ! initialize x to 0x42
y:      .word 0x20              ! initialize y to 0x20
z:      .word 0                 ! initialize z to 0

        .text                   ! instructions
start:
        set    x, %r2           ! &x --> %r2
        ld     [%r2], %r2       ! x --> %r2
        set    y, %r3           ! &y --> %r3
        ld     [%r3], %r3       ! y --> %r3
        add    %r2, %r2, %r2    ! r2 + r2 --> %r2
        add    %r2, %r3, %r2    ! r2 + r3 --> %r2
        set    z, %r3           ! &z --> %r3
        st     %r2, [%r3]       ! r2 --> x

end:    ta     0
```

Figure 1.1 A sample SPARC program implementing $z = 2x + y$

Hexadecimal ISEM reports all of its results in hexadecimal. To simplify your interaction with ISEM, we will use hexadecimal notation in our programs. The program shown in Figure 1.1 uses the hexadecimal constants: 0x42 and 0x20.

We will use the listing specification "−als" (to generate a source listing and a list of symbols). The output specification consists of a -o followed by the name of the output file. Figure 1.2 illustrates the interaction that results when you assemble the program in foo.s.

Activity 1.2 *Assemble the program in* foo.s, *placing the object code in the file* foo.o.

The linker is called *isem-ld*. Linker commands start with the name of the linker, isem-ld, followed by an output specification, followed by the name of the file containing the object program.

Activity 1.3 *Link the object code in* foo.o, *placing the executable program in the file* foo.

1.3.2 Testing Your Program

Once you have assembled and linked your assembly language program, you can execute your program using ISEM. To start ISEM, you need to issue the ISEM command, "isem".

When you start ISEM, you will see an introductory message followed by the ISEM prompt ("ISEM>"). When you see this prompt, you can issue an ISEM command. Figure 1.3 illustrates the interaction you should expect to see when you start up ISEM.

The load command

The *load* command is used to load executable programs into the ISEM environment. The load command consists of the name of the command (load) followed by the name of a file containing an executable program.

Figure 1.4 illustrates the ISEM load command. In examining this interaction, note that ISEM tells you where it loaded the program text (instructions) and data (variables). ISEM

```
%  isem-as -als -o foo.o foo.s
SPARC GAS  foo.s                         page 1

   1                           .data                  ! variables
   2 0030 00000042      x:     .word 0x42             ! initialize x
to 0x42
   3 0034 00000020      y:     .word 0x20             ! initialize y
to 0x20
   4 0038 00000000      z:     .word 0                ! initialize z
to 0
   5
   6                           .text                  ! instructions
   7                    start:
   8 0000 05000000             set    x, %r2          ! &x --> %r2
   8      8410A000
   9 0008 C4008000             ld     [%r2], %r2      ! x --> %r2
  10 000c 07000000             set    y, %r3          ! &y --> %r3
  10      8610E000
  11 0014 C600C000             ld     [%r3], %r3      ! y --> %r3
  12 0018 84008002             add    %r2, %r2, %r2   ! r2 + r2 -->
%r2
  13 001c 84008003             add    %r2, %r3, %r2   ! r2 + r3 -->
%r2
  14 0020 07000000             set    z, %r3          ! &z --> %r3
  14      8610E000
  15 0028 C420C000             st     %r2, [%r3]      ! r2 --> x
  16 002c 91D02000             ta     0
  17                    end:

SPARC GAS  foo.s                         page 2


DEFINED SYMBOLS
               foo.s:2        2:00000030 x
               foo.s:3        2:00000034 y
               foo.s:4        2:00000038 z
               foo.s:7        1:00000000 start
               foo.s:17       1:00000030 end

UNDEFINED SYMBOLS
```

Figure 1.2 Illustrating the isem-as command.

File Names Traditionally, file name suffixes indicate the type of program stored in a file. Files that contain assembly language programs have a suffix of ".s". Files that contain object programs have a suffix of ".o". Files that contain executable programs do not usually have a suffix.

In the preceding paragraphs, we used the files *foo.s* (for the assembly language program), *foo.o* (for the object program), and *foo* (for the executable program).

---

also tells you the current value of the program counter (PC) and the next program counter (nPC). Finally, ISEM shows you the next instruction that it will execute (i.e., the instruction pointed to by the PC).

```
% isem

Instructional SPARC Emulator
Copyright 1993 - Computer Science Department
                 University of New Mexico

ISEM comes with ABSOLUTELY NO WARRANTY

ISEM Ver 1.00a : Mon Nov 1 20:25:01 MST 1993
```

Figure 1.3 Illustrating the isem command.

```
ISEM> load foo
Loading File: foo
2000 bytes loaded into Text region at address 8:0
2000 bytes loaded into Data region at address a:2000

  PC: 08:00000020    nPC: 00000024    PSR: 0000003e    N:0 Z:0 V:0
C:0

              start    :  sethi   0x8, %g2
```

Figure 1.4 Illustrating the load command.

Activity 1.4 *Start ISEM and load the file* foo*.*

Note that the instruction (sethi 0x8, %g2) doesn't look like the first instruction in the sample program (set x, %r2). We will discuss the reason for this when we consider *synthetic operations* in Lab 9. For now, it is sufficient to know that set instruction may be implemented using two instructions: a sethi instruction followed by an or instruction.

The trace command

You can execute your program, one instruction at a time, using the *trace* command. The trace command executes a single instruction and reports the values stored in the registers, followed by the next instruction to be executed.

Figure 1.5 illustrates three successive executions of the trace command. Note that register 2 (first row, third column) now has the value 0x00000042—the value used in the initialization of x.

To complete the execution of the sample program, you need to issue nine more trace commands (a total of twelve trace commands). As you issue trace commands, note how the values in registers 2 and 3 change. When you have executed all of the instructions in the sample program, ISEM will print the message "Program exited normally." Figure 1.6 illustrates the execution of the last two trace commands.

Activity 1.5 *Execute the sample program by issuing twelve trace commands.*

The dump command

The trace command is useful because it lets you see how each instruction affects the registers when it is executed. You can also examine the contents of memory using the *dump* command. You can issue a dump command any time you see the ISEM> prompt.

To use the dump command, you need to specify the range of memory values that you want to examine. A range of memory locations is specified using two memory addresses

```
     ISEM> trace
       ----0--- ----1--- ----2--- ----3--- ----4--- ----5--- ----6---
----7---
     G 00000000 00000000 00002000 00000000 00000000 00000000 00000000
00000000
     O 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
     L 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
     I 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
       PC: 08:00000024    nPC: 00000028    PSR: 0000003e    N:0 Z:0 V:0
C:0

                 start+04 :  or      %g2, 0x60, %g2

     ISEM> trace
       ----0--- ----1--- ----2--- ----3--- ----4--- ----5--- ----6---
----7---
     G 00000000 00000000 00002060 00000000 00000000 00000000 00000000
00000000
     O 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
     L 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
     I 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
       PC: 08:00000028    nPC: 0000002c    PSR: 0000003e    N:0 Z:0 V:0
C:0

                 start+08 :  ld      [%g2], %g2

     ISEM> trace
       ----0--- ----1--- ----2--- ----3--- ----4--- ----5--- ----6---
----7---
     G 00000000 00000000 00000042 00000000 00000000 00000000 00000000
00000000
     O 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
     L 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
     I 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
       PC: 08:0000002c    nPC: 00000030    PSR: 0000003e    N:0 Z:0 V:0
C:0

                 start+0c :  sethi   0x8, %g3
```

Figure 1.5 Illustrating the trace command.

separated by a comma. Memory addresses can be specified using an integer value or the name of a label. For example, to see the final value stored in the memory location associated with the label “z” you could use the range “z, z”.

Figure 1.7 illustrates the dump command. The dump command reports memory values in pairs of hexadecimal digits. Each word of memory is 32 bits and, as such, requires four

```
ISEM> trace
  ----0--- ----1--- ----2--- ----3--- ----4--- ----5--- ----6---
----7---
G 00000000 00000000 000000a4 00002068 00000000 00000000 00000000
00000000
O 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
L 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
I 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
  PC: 08:0000004c    nPC: 00000050    PSR: 0000003e    N:0 Z:0 V:0
C:0

            start+2c :  ta      [%g0 + 0x0]

ISEM> trace
Program exited normally.
  ----0--- ----1--- ----2--- ----3--- ----4--- ----5--- ----6---
----7---
G 00000000 00000000 000000a4 00002068 00000000 00000000 00000000
00000000
O 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
L 00000000 0000004c 00000050 00000000 00000000 00000000 00000000
00000000
I 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
  PC: 09:00000800    nPC: 00000804    PSR: 0000009d    N:0 Z:0 V:0
C:0

             end+7b0 :  jmpl    [%l2], %g0
```

Figure 1.6 Completing execution of the sample program.

pairs of hexadecimal digits. In examining Figure 1.7, note that "z" holds the value 0xa4 (the final value in register %r2).

```
ISEM> dump z,z
0a:00002068  00 00 00 a4 00 00 00 00                         ........
```

Figure 1.7 Illustrating the dump command.

Activity 1.6 *Use the dump command to examine the values stored in the memory locations* x, y, *and* z.

Memory Addresses When you are interacting with ISEM, memory addresses can be specified using integer constants or the labels defined in an assembly language program.

The symb command

Because memory addresses can be specified using the labels defined in an executable program, you may be interested in knowing which labels have defined values. After you have loaded an executable program, you can use the *symb* command to display the values defined by the program. Figure 1.8 illustrates the symb command.

```
ISEM> symb
Symbol List
                    end : 00000050
                  start : 00000020
                      x : 00002060
                      y : 00002064
                      z : 00002068
```

Figure 1.8 Illustrating the symb command.

Activity 1.7 *Use the symb command to examine the labels defined by the sample program.*

The edit command

The *edit* command sets the values stored in memory locations. Each edit command requires two arguments, the memory location to edit and the value to store in the memory location. For example, you could use the command "edit x 0x20" to set the value of the memory location labeled "x" to the value 0x20.

Activity 1.8 *Use the edit command to change the values associated with* x *and* y*.*

The reg command

You can use the *reg* command to set the values in the registers. The reg command can have zero, one or two arguments. The first argument (if present) must be the name of a SPARC register. The second argument (if present) must be a value. With zero arguments the reg command prints the contents of all of the SPARC registers. With one argument the reg command prints the value of the specified register. With three arguments the reg command sets the register specified by the first argument to the value specified by the second argument.

Table 1.1 summarizes the names of the SPARC registers. For now, you only need to be familiar with the integer registers and the program counters. We will consider the remaining registers in later labs.

Table 1.1 SPARC Registers

Group name	Register names
Integer registers	%r0–%r31
Program counters	%pc, %npc
Multiply/divide register	%y
Floating point registers	%f0–%f31
Floating point queue	%fq
Floating point status register	%fsr
Processor status register	%psr
Window invalid mask	%wim
Trap base register	%tbr

Activity 1.9 *Use the reg command to examine the values of the registers.*

For example, the command "reg %pc start" resets the PC to the start of the program. You can use this command when you want to rerun the sample program.

The run command

You can use the *run* command to execute your program, starting with the instruction pointed to by the PC. This command does not take any arguments and executes instructions until it encounters a breakpoint, an illegal instruction, or a program termination instruction (ta 0).

Activity 1.10 *Use the reg command to reset the value of %pc. Then use the run command to rerun the sample program.*

Figure 1.9 illustrates the run command. This interaction starts by setting the %pc and then issuing the run command. Note that the run command produces an error message. In this case, the run command stopped executing program instructions because it encountered an illegal instruction. Whenever you load a program, ISEM makes sure that there is an illegal instruction following the last instruction in your program.

```
ISEM> reg %pc start
Register: %pc = 20

ISEM> run
Program exited normally.
```

Figure 1.9 Illustrating the reg and run commands.

The break command

You can use the *break* command to set breakpoints (the run command terminates when it encounters a breakpoint). To set a breakpoint, you can issue a break command with a single argument, the address of the breakpoint. After you have set a breakpoint, the run command will stop executing your program just *after* it executes the instruction at the specified address.

Figure 1.10 illustrates the use of the break command. In this case, the breakpoint stops the program 20 bytes after the start label—just before the "add %r2, %r2, %r2" instruction. Note that ISEM reports the breakpoint address in hexadecimal.

```
ISEM> reg %pc start
Register: %pc = 20

ISEM> break start+20

ISEM> run
Breakpoint encountered at start+14
```

Figure 1.10 Illustrating the break command.

The help command

The help command may be the most important command provided by ISEM. The help command takes a single, optional argument. When it is supplied, the argument is the name of the item that you would like more information about. Without any arguments, the help command tells you the items that help knows about.

The quit command

When you are done interacting with ISEM, you can issue the *quit* command.

1.3.3 Memory regions

The ISEM/SPARC architecture provides two separate regions of memory: one for text (code) and another for data. This means that there are two memory locations with the address 100, one in the program memory and another in the data memory. ISEM is reasonably intelligent about when it uses each of these memories. ISEM always fetches instructions from the program memory and the load and store operations always refer to the data memory. In addition, the load command always loads programs into the program memory and the dump and edit commands use the data memory by default.

1.4 Summary

In this lab we have described the steps in the translation process and introduced that basic functions provided by ISEM.

The ISEM assembler, isem-as, translates an assembly language program into object code. The linker, isem-ld, translates an object code program into an executable program. Figure 1.11 summarizes the steps uses to translate an assembly language program into an executable program.

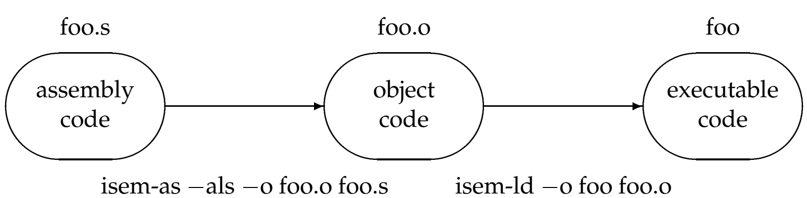


Figure 1.11 Translation steps

When you have an executable program, you can use ISEM to execute and test your program. Figure 1.12 illustrates the basic components of ISEM and shows the commands that manipulate these components. Table 1.2 summarizes the ISEM commands that we have discussed in this lab.

1.5 Review Questions

1. What is the name of the tool that translates an assembly language program into object code? What is the name of the tool that translates object code into an executable program?
2. What does the trace command do?
3. What does the dump command do?
4. What does the symb command do?
5. What does the edit command do?
6. What does the reg command do?
7. What does the run command do?
8. What is a breakpoint? Explain how to set a breakpoint in ISEM.

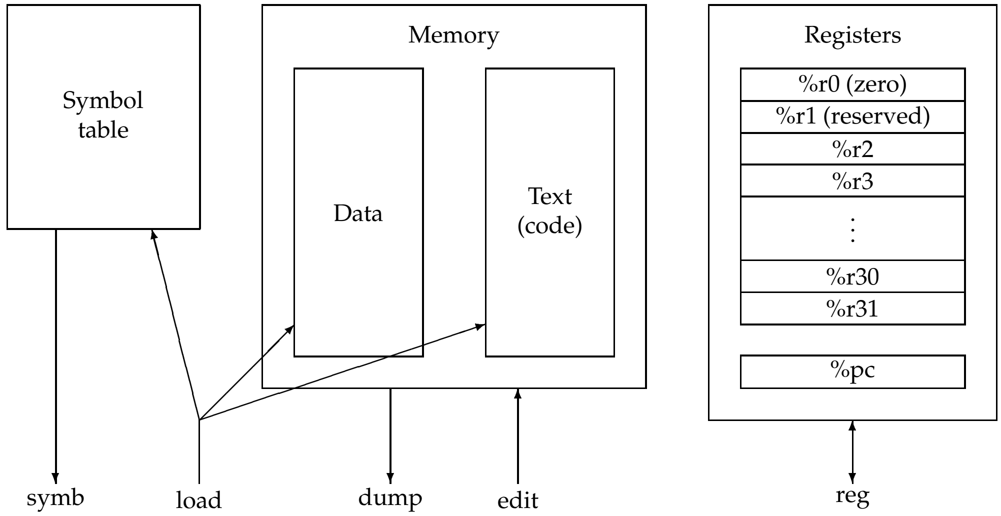


Figure 1.12 The components of ISEM

Table 1.2 The commands of ISEM

Syntax	Meaning
break *address*	Set an execution breakpoint.
dump [*address* [, *address*]]	Display the contents of memory
edit *address value*	Set the contents of a memory location.
help [*topic*]	Provide information about an isem topic or command.
load *filename*	Load an executable program.
quit	Exit ISEM.
symb	Display the symbol table.
reg [*register* [, *value*]]	Display or set the contents of a register.
run	Execute instructions to a breakpoint or illegal instruction.
trace	Execute the next instruction.

Notes:
Items in square brackets (“[” … “]”) are optional.

address	a memory address (may be a label or number
value	an integer value (0x. . . means hexadecimal notation)
filename	the name of a file
register	the name of a register

1.6 Exercises

1. After you have successfully assembled the sample program, perform the following modifications to the second line of the program (x: .word 0x42). In each case, you should start with the original sample program and you should write down the error message (if any) produced by the assembler.
   a. remove the “:” following “x”,
   b. remove the “.” preceding “word”.
2. After you have successfully assembled the sample program, perform the following modifications to the eleventh line of the program (ld [%r3], %r3). In each case, you should start with the original sample program and you should write down the error

message (if any) produced by the assembler.

a. remove the “[” and “]” surrounding first “%r3”,
b. remove the “%” preceding the first “r3”.

3. Load the sample program into ISEM, run it, and then issue the following ISEM commands. In each case, describe the output produced by ISEM and explain why ISEM produced the results that it produced (you may need to use the help command to get further information about the commands).

   a. dump x,z
   b. dump z,x
   c. dump z
   d. dump x
   e. dump start,end

4. In examining Figure 1.4, note that the value of the PC is given as “08:00000020”. What does the “08:” mean?
5. In examining Figure 1.7, note that the address for *z* is given as “0a:00002068”. What does the “0a:” mean?

Laboratory 2
Assembly Language Programming

2.1 Goal

To introduce the fundamentals of assembly language programming.

2.2 Objectives

After completing this lab, you will be able to write assembly language programs that use:

- The .text and .data assembler directives,
- The .word assembler directive,
- The integer registers (%r0–%r31),
- The (synthetic) set operation,
- The load and store operations,
- The signed integer addition and subtraction operations, and
- The (synthetic) mov operation.

2.3 Discussion

In this lab we introduce the fundamentals of SPARC assembly language programming. In particular, we consider basic assembler directives, register naming conventions, the (synthetic) load and store operations, the integer addition and subtraction operations, and the (synthetic) register copy and register set operations. We begin by considering the structure of assembly language programs.

2.3.1 Assembly language

Assembly language programs are line-oriented. That is, the assembler translates an assembly language program one line at a time. The assembler recognizes four types of lines: empty lines, label definition lines, directive lines, and instruction lines.

- A line that only has spaces or tabs (i.e., white space) is an *empty line*. Empty lines are ignored by the assembler.
- A *label definition line* consists of a label definition. A label definition consists of an identifier followed by a colon (":"). As in most programming languages, an identifier must start with a letter (or an underscore) and may be followed by any number of letters, underscores, and digits.
- A *directive line* consists of an optional label definition, followed by the name of an assembler directive, followed by the arguments for the directive. In this lab we will consider three assembler directives: .data, .word, and .text.

- An *instruction line* consists of an optional label definition, followed by the name of an operation, followed by the operands. In this lab we will consider five operations: load, store, set, add, and sub.

Every line can conclude with a comment. Comments begin with the character "!". Whenever it encounters a "!", the assembler ignores the "!" and the remaining characters on the line.

Activity 2.1 *Consider the SPARC program presented in Figure 1.1. For each nonempty line in the program, identify any labels defined and identify any assembler directives and assembly language instructions.*

2.3.2 Directives

In this lab we introduce three directives: .data, .text, and .word. The first two (.data and .text) are used to separate variable declarations and assembly language instructions. The .word directive is used to allocate and initialize space for a variable.

Each group of variable declarations should be preceded by a .data directive. Each group of assembly language instructions should be preceded by a .text directive. Using these directives, you could mix variable declarations and assembly language instructions; however, for the present, your assembly language programs should consist of a group of variable declarations followed by a group of assembly language instructions.

A variable declaration starts with a label definition (the name of the variable), followed by a .word directive, followed by the initial value for the variable. The assembler supports a fairly flexible syntax for specifying the initial value. For now, we will use simple integer values to initialize our variables. By default, the assembler assumes that numbers are expressed using decimal notation. You can use hexadecimal notation if you use the "0x" prefix. Example 2.1 illustrates a group of variable declarations.

Example 2.1 *Give directives to allocate space for three variables, x, y, and z. You should initialize these variables to decimal 23, hexadecimal 3fce, and decimal 42, respectively.*

```
        .data            ! start a group of variable declarations
x:      .word   23       ! int x = 23;
y:      .word   0x3fce   ! int y = 0x3fce;
z:      .word   42       ! int z = 42;
```

2.3.3 Labels

In an assembly language program, a label is simply a name for an address. For example, given the declarations shown in Example 2.1, "x" is a name for the address of a memory location that was initialized to 23. On the SPARC an address is a 32-bit value. As such, labels are 32-bit values when they are used in assembly language programs.

2.3.4 Integer registers

The SPARC integer unit provides thirty-two general purpose registers. Each integer register holds 32-bits. The integer registers are called %r0 through %r31. In addition to the names %r0 through %r31, the integer registers have alternate names (aliases) as shown in Table 2.1.

The letter used in each group of aliases (g, o, l, or i) denotes the name for the group of registers. The group names are related to procedure calling conventions. We will discuss

Table 2.1 Aliases for the integer registers

Integer registers	Alternate names	Group name
%r0–%r7	%g0–%g7	Global registers
%r8–%r15	%o0–%o7	Output registers
%r16–%r23	%l0–%l7	Local registers
%r24–%r31	%i0–%i7	Input registers

the meanings of these group names when we consider register windows in Lab 11. In the meantime, we will use the %r names in our assembly language programs. As you may have noted, ISEM uses the alternate names when it reports the contents of the registers and when it shows the next instruction to execute.

Register Names Register names on the SPARC always start with a percent sign ("%"). For example, the integer registers are named %r0 through %r31.

2.3.5 %r0

The value stored in %r0 is always zero and cannot be altered. If an instruction specifies %r0 is used as the destination, the result is simply discarded. It is not an error to execute an instruction that specifies %r0 as the destination for the result; however, the contents of %r0 will not be altered when this instruction is executed.

%r0 Register %r0 always holds the value zero. The value stored in this register cannot be altered.

2.3.6 The set operation

The set operation can be used to load a 32-bit signed integer constant into a register. Every set instruction has two operands: the 32-bit value followed by the destination register. Table 2.2 summarizes the set operation.

Table 2.2 The set operation

Operation	Assembler syntax	Operation implemented
register set	set $siconst_{32}$, rd	reg[rd] = $siconst_{32}$

Notes:
$siconst_{32}$ a 32-bit signed integer constant (can be specified by a label)
rd the destination register

Destination last In SPARC assembly language instructions, the destination is specified as the last operand.

Example 2.2 *Using set instructions, write code that will load the value 0x42 into register %r2 and the address of x (from Example 2.1) into register %r3.*

```
set   0x42, %r2
set   x, %r3
```

2.3.7 The load and store operations

The SPARC is based on the load/store architecture. This means that registers are used as the operands for all data manipulation operations. The operands for these operations cannot be in memory locations. Table 2.3 summarizes simple versions of the load and store operations. (We will cover these operations in more detail in later labs.)

Table 2.3 The load and store operations

Operation	Assembler syntax	Operation implemented
load word	ld [*rs*], *rd*	reg[*rd*] = memory[reg[*rs*]]
store word	st *rs*, [*rd*]	memory[reg[*rd*]] = reg[*rs*]

Notes:
rd the destination register
rs the source register

Source and destination registers When we introduce assembly language syntax, the names *rs* and *rd* are used to denote source and destination registers, respectively. When an instruction uses multiple source registers, we use subscripts to distinguish these registers.

2.3.8 The addition and subtraction operations

The SPARC uses 2's complement representation for signed integer values. Signed additions and subtractions are performed using 32-bit arithmetic (the source and destination values are 32 bits).

Table 2.4 summarizes the signed addition and subtraction operations provided by the SPARC. The SPARC provides two instruction formats for each of the arithmetic operations. Both formats use three explicit operands—two source operands, and a destination operand. In the first format both of the source operands are in registers. In the second format, one of the source operands is in a register while the other is a small constant value. This constant value may be negative or positive; however, its 2's complement representation must fit in 13 bits. Example 2.3 presents a SPARC assembly language program that illustrates variable declarations and the operations (load, store, add, and sub) that we have described in this lab.

Integer Constants We use the name $siconst_n$ to denote a signed integer constant in assembly language syntax. The subscript indicates, n, the number of bits used in the 2's complement representation of this value.

Table 2.4 The addition and subtraction operations

Operation	Assembler syntax		Operation implemented
integer addition	add	rs_1,rs_2, *rd*	reg[*rd*] = reg[rs_1] + reg[rs_2]
	add	*rs*, $siconst_{13}$, *rd*	reg[*rd*] = reg[*rs*] + $siconst_{13}$
integer subtraction	sub	rs_1, rs_2, *rd*	reg[*rd*] = reg[rs_1] − reg[rs_2]
	sub	*rs*, $siconst_{13}$, *rd*	reg[*rd*] = reg[*rs*] − $siconst_{13}$

Note:
$siconst_{13}$ a 13-bit (2's complement) signed integer constant.

2.3.9 Program termination

Programs to be run in the ISEM environment should terminate their execution by executing the instruction "ta 0". Whenever this instruction is executed, ISEM will stop executing instructions and print the message "Program exited normally".

Example 2.3 *Write a SPARC assembly language program to evaluate the statement* $a = (a + b) - (c - d)$.

```
        .data
a:      .word   0x42
b:      .word   0x43
c:      .word   0x44
d:      .word   0x45

        .text
start:  set     a, %r1
        ld      [%r1], %r2          ! a --> %r2
        set     b, %r1
        ld      [%r1], %r3          ! b --> %r3
        set     c, %r1
        ld      [%r1], %r4          ! c --> %r4
        set     d, %r1
        ld      [%r1], %r5          ! d --> %r5

        add     %r2, %r3, %r2       ! a + b --> %r2
        sub     %r4, %r5, %r3       ! c - d --> %r3
        sub     %r2, %r3, %r2       ! (a + b) - (c - d) --> %r2
        set     a, %r1
        st      %r2, [%r1]          ! (a + b) - (c - d) --> a

end:    ta      0
```

Activity 2.2 *Using a text editor, enter the program shown in Example 2.3 into a file, assemble it, link it, and test it using isem.*

2.3.10 The mov operation

We conclude this lab by considering another (synthetic) operation: mov. The mov operation is used to copy the value stored in one register to another register. This operation can also be used to load a small integer value into a register. Table 2.5 summarizes the mov operations.

Table 2.5 The register copy and register set operations

Operation	Assembler syntax		Operation implemented
register copy	mov	*rs*, *rd*	reg[*rd*] = reg[*rs*]
load constant	mov	$siconst_{13}$, *rd*	reg[*rd*] = signextend($siconst_{13}$)

Because you can always use the set operation to load a 13-bit value to an integer register, the second version of the mov operation is redundant for integer registers. However, as we will discuss, this version of the mov operation is used to load the other state registers on the SPARC.

2.4 Summary

In this lab we have introduced the basics of SPARC assembly language programming. We began by considering the structure of an assembly language program. We then considered the names and uses of the integer registers. We then introduced three assembler directives: .text, .data, and .word. The first two (.text and .data) are used to identify sections of an assembly language program. The last two (.data and .word) are used to declare and initialize variables. We will consider additional assembler directives in later labs. We concluded the lab by introducing six assembly language operations: set, load, store, add, sub, and mov.

Figure 2.1 provides a graphical illustration for several of the operations that we have introduced in this lab. In particular, this figure illustrates the data paths used in the load, store, addition, and subtraction operations.

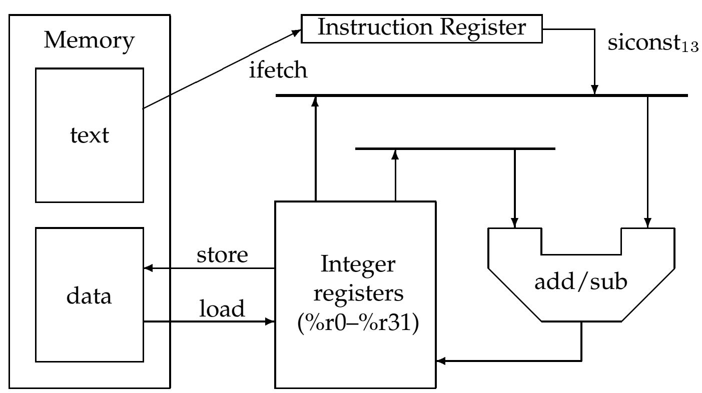


Figure 2.1 Illustrating the load, store, add, and subtract operations

The set and mov operations are synthetic (or pseudo) operations. That is, these operations are not really SPARC operations. Instead, the assembler translates these operations into one or more SPARC operations when it assembles your program. We will consider synthetic operations in Lab 9

2.5 Review Questions

1. Explain the difference between a directive, an operation, and an instruction. Give an example of each.

2. How are the integer registers named on the SPARC?
3. How many integer registers are there on the SPARC?
4. For each of the integer registers that have special attributes, explain the special attributes.
5. What does $siconst_{13}$ mean when used to specify assembly language syntax?
6. The subtraction operation has two source operands. Which operand is subtracted from the other?
7. Describe when you would use the set operation.

2.6 Exercises

1. Suppose that the SPARC did not provide a (synthetic) register copy operation, explain how you could emulate this operation.
2. For each of the following statements, write, assemble, and test a SPARC assembly language fragment that implements the statement. Be certain to declare and initialize all variables in your assembly language programs.
    a. $a = c + d$.
    b. $a = (c + d) - (c + b + d - e)$.
    c. $a = (d - 13) + (a + 23)$.
    d. $a = d + 9832$.
    e. $a = 87765 - c$.

Laboratory 3
Implementing Control Structures

3.1 Goal

To cover the implementation of control structures using the SPARC instruction set.

3.2 Objectives

After completing this lab, you will be able to write assembly language programs that use:

- The condition code register,
- Operations that set the condition code register,
- The conditional and unconditional branching operations of the SPARC, and
- Conditional nullification.

3.3 Discussion

In this lab we introduce a subset of the SPARC branching operations. In particular, we introduce the operations that provide conditional and unconditional branching based on the bits in a condition code register.

We begin by considering the bits in the condition code register of a SPARC processor. After introducing these bits, we consider the operations that affect the bits in the condition code register. We then consider the conditional and unconditional branching operations that use the bits in the condition code register to control branching. Next, we introduce nullification (annulment) in the branching operations of the SPARC. We conclude by considering several examples to illustrate the SPARC operations and by introducing the compare operation provided by the SPARC.

3.3.1 The condition code register

The condition code register on the SPARC has four bits: Z (Zero), N (Negative), C (Carry), and V (oVerflow). The standard arithmetic operations (e.g., addition and subtraction) do not update the bits in the condition code register. Instead, there are special operations that update the condition code register. Table 3.1 summarizes a collection of operations that update the bits in the condition code register. The names for these operations have a suffix of "cc" to indicate that they update the bits in the condition code register.

In most cases, the effect that an operation has on the condition codes is just what you would expect. Most of these operations set the Z bit when the result of the operation is zero, and clear this bit when the result is nonzero. Similarly, most of these operations set the N bit when the result of the operation is negative, and clear this bit when the result is nonnegative. The V bit is usually set when the (signed integer) result of the operation

Table 3.1 Updating the condition code register

Operation	Operation name
Signed integer addition	addcc
Signed integer subtraction	subcc

cannot be stored in 32 bits, and cleared when the result can be stored in 32 bits. Finally, the C bit is set when the operation generates a carry out of the most significant bit, and cleared otherwise.

In most contexts, you will be most interested in the N and Z bits of the condition code register and we will emphasize these bits in the remainder of this lab. We will consider the remaining bits in the condition code register (the C and V bits) at greater length in Lab 13.

3.3.2 Branching operations

The SPARC provides 16 basic branching operations. These operations are summarized in Table 3.2. Note that the first two operations, ba (branch always) and bn (branch never), are unconditional—the operation specifies whether the branch is taken. The remaining operations are conditional branching operations. When these operations are used, the branch is only taken when the specified condition in met. In last column of Table 3.2 we use a boolean expression involving the bits of the condition code register to specify the condition. The condition is satisfied if the boolean expression results in the value 1; otherwise (if the expression results in 0), the condition is not satisfied and the processor continues with sequential execution of instructions. The target specified in an assembly language instruction is a label defined by the program.

Table 3.2 Branching operations on the SPARC

Operation	Assembler syntax		Branch condition
Branch always	ba	*target*	1 (always)
Branch never	bn	*target*	0 (never)
Branch not equal	bne	*target*	not Z
Branch equal	be	*target*	Z
Branch greater	bg	*target*	not (Z or (N xor V))
Branch less or equal	ble	*target*	Z or (N xor V)
Branch greater or equal	bge	*target*	not (N xor V)
Branch less	bl	*target*	N xor V
Branch greater, unsigned	bgu	*target*	not (C or Z)
Branch less or equal, unsigned	bleu	*target*	C or Z
Branch carry clear	bcc	*target*	not C
Branch carry set	bcs	*target*	C
Branch positive	bpos	*target*	not N
Branch negative	bneg	*target*	N
Branch overflow clear	bvc	*target*	not V
Branch overflow set	bvs	*target*	V

In addition to the operation names defined in Table 10, the SPARC defines several synonyms for these operations. These synonyms are summarized in Table 3.3.

Like most RISC machines, the SPARC uses a branch delay slot. By default, the instruction following a branch instruction is executed whenever the branch instruction is

Table 3.3 Synonyms for branching operations

Operation	Operation name	Synonym for
Branch nonzero	bnz	bne
Branch zero	bz	be
Branch greater or equal, unsigned	bgeu	bcc
Branch less, unsigned	blu	bcs

executed.

SPARC assemblers provide a special (synthetic) operation, *nop*, for situations when it is not convenient to put a useful instruction in the delay slot of a branch instruction. In assembly language a nop instruction has no operands (i.e., a nop instruction is fully specified by the name of the operation). When a nop instruction is executed, it does not alter any of the registers or values stored in memory. However, the use of nop instructions causes the processor to execute more instructions and, as such, increases the time required to execute the program. Example 3.1 illustrates the conditional and unconditional branching operations.

Example 3.1 *Translate the following C code fragment into SPARC assembly language.*

```
int temp;
int x = 0;
int y = 0x9;
int z = 0x42;

temp = y;
while( temp > 0 ) {
    x = x + z;
    temp = temp - 1;
}
```

To simplify the translation, we fill the branch delay slots with nop instructions.

```
        .data
x:      .word   0
y:      .word   0x9
z:      .word   0x42

        .text
start:  set     y, %r1
        ld      [%r1], %r2          ! we'll use %r2 for temp
        set     z, %r1
        ld      [%r1], %r3          ! we'll use %r3 for z
        mov     %r0, %r4            ! we'll use %r4 for x

        add     %r2, 1, %r2         ! set up for decrement
        ba      test                ! test the loop condition
        nop                         ! BRANCH DELAY SLOT
top:    add     %r4, %r3, %r4       ! x + z --> x
test:   subcc   %r2, 1, %r2         ! temp - 1 --> temp
        bg      top                 ! temp > 0 ?
        nop                         ! BRANCH DELAY SLOT

        set     x, %r1
        st      %r4, [%r1]          ! store x
end:    ta      0
```

Activity 3.1 *After each trace command, ISEM reports the values of the bit in the condition code register. Type the program shown Example 3.1 into a file, assemble it, link it, and load it into ISEM. Trace the program execution, noting how each instruction affects the bits in the SPARC condition code register.*

The SPARC keeps track of the instructions to execute using two program counters: PC, and nPC. The first program counter, PC, holds the address of the next instruction to execute. The second program counter, nPC, holds the next value for PC. Usually, the SPARC updates the program counters at the end of each instruction execution by assigning the current value of nPC to PC, and adding 4 to the value of nPC. When it executes a branching operation, the SPARC assigns the current value of nPC to PC and then updates the value of nPC. If the branch is taken, nPC is assigned the value of the *target* specified in the instruction; otherwise, nPC is incremented by 4. The branch delay slot arises because the PC is assigned the old value of nPC (before nPC is assigned the target of the branch).

Activity 3.2 *After each trace command, ISEM reports the values of PC and nPC. Run the program shown in Example 3.1 noting the changes to PC and nPC.*

3.3.3 Nullification

Every branching instruction can specify that the affect of the instruction in the branch delay slot is to be nullified (annulled in SPARC terminology) if the branch specified by the conditional branching instruction is not taken. In assembly language, this conditional nullification is specified by appending a suffix of ",a" to the name of the branching operation. Example 3.2 illustrates conditional nullification.

Example 3.2 *Rewrite the code fragment shown in Example 3.1 so that the code has meaningful instructions in the branch delay slots.*

```
        .data
x:      .word   0
y:      .word   0x9
z:      .word   0x42

        .text
start:  set     y, %r1
        ld      [%r1], %r2          ! we'll use %r2 for temp
        set     z, %r1
        ld      [%r1], %r3          ! we'll use %r3 for z
        mov     %r0, %r4            ! we'll use %r4 for x

        add     %r2, 1, %r2         ! set up for decrement
top:    subcc   %r2, 1, %r2         ! temp - 1 --> temp
        bg,a    top                 ! temp > 0 ?
        add     %r4, %r3, %r4       ! x + z --> x

        set     x, %r1
        st      %r4, [%r1]          ! store x
end:    ta      0
```

3.3.4 The (synthetic) integer comparison operation

Assemblers for the SPARC provide a synthetic integer comparison operation. You can use this operation when the data manipulation operations do not establish the needed values in the condition code register. Table 3.4 summarizes the integer comparison operation. This operation can be used to compare the contents of two registers or to compare the contents of a register with a small integer constant.

Table 3.4 Signed integer comparison

Operation	Assembler syntax		Operation implemented
integer comparison	cmp	sr_1, sr_2	reg[sr_1] $-$ reg[sr_2]
	cmp	sr, $siconst_{13}$	reg[dr] = reg[sr] $-$ $siconst_{13}$

3.3.5 Delayed control-transfer couples

When a branch instruction is in the delay slot of another branch instruction, the pair of branch instructions is called a "delayed control-transfer couple". If you use a delayed control-transfer couple on the SPARC, the first branch operation should be an unconditional branch; otherwise, the sequence of instructions executed when the delayed control-transfer couple is executed is not defined. We will consider delayed control-transfer couples in greater depth when we consider traps and exceptions in Lab 16.

3.4 Summary

In this lab we have introduced the condition code register, the basic branching operations, and the integer comparison operation. The branching operations include two unconditional branch operations (*ba* and *bn*) and a host of conditional branching operations. The SPARC branching operations have a branch delay slot. That is, the instruction following a branch instruction is executed whenever the branch instruction is executed. The SPARC provides conditional annulment of the instruction in the branch delay slot. When the branch operation specifies annulment (using the operator suffix ",a"), the affects of the instruction are canceled (note, the instruction is executed, but the execution has no affect).

3.5 Review Questions

1. What are the bits in the condition code register.
2. Name two operations that affect the bits in the condition code register and explain how they affect these bits.
3. What are the two program counters on the SPARC. Explain how these program counters are used.
4. Is the affect of the instruction in the delay slot of an annulled branch canceled when the branch is take or when the branch is not taken? Explain why the designers of the SPARC chose the this version of nullification.

3.6 Exercises

1. Suppose that your assembler did not provide an integer comparison operation. Explain how you could implement this operation using the other SPARC operations that we have considered in this and previous labs.
2. Consider the SPARC code presented in Example 3.2. Currently, the loop is executed "y" times. If "y" is larger than "z" it would be better to execute the loop "z" times. Rewrite the code shown in Example 3.2 to take advantage of this observation.
3. Write a SPARC program that has four variables: $x \geq 0$, $y > 0$, z, and w. Your program should assign the quotient of x/y to z and the remainder to w. (You should write this code using the operations presented in this and previous labs. Do not use the SPARC integer multiplication or division operations.
4. Write a SPARC program that will compute the greatest common divisor of a and b and assign this value to c.

Laboratory 4
Multiplication and Division

4.1 Goal

To cover the SPARC operations related to multiplication and division.

4.2 Objectives

After completing this lab, you will be able to write assembly language programs that use:

- The signed and unsigned multiplication and division operations,

4.3 Discussion

In this lab we consider the SPARC operations related to integer multiplication and division. We begin by considering the signed integer multiplication and division operations.

4.3.1 The multiplication and division operations

The integer multiplication operations multiply two 32-bit source values and produce a 64-bit result. The most significant 32 bits of the result are stored in the Y register (%y) and the remaining 32 bits are stored in one of the integer registers. Figure 4.1 illustrates the use of the Y register during a multiplication operation.

The integer division operations divide a 32-bit value into a 64-bit value and produce a 32-bit result. The Y register provides the most significant 32 bits of the 64-bit dividend. One of the source values provides the least significant 32 bits, while the other provides the 32 bit divisor. A implementation of the SPARC may optionally store the remainder in the Y register. ISEM does *not* store the remainder in the Y register, so we will adopt this convention in our presentation. Figure 4.2 illustrates the use of the Y register during the division operation.

Table 4.1 summarizes the assembly language syntax for the integer multiplication and division operations provided by the SPARC. Like the addition and subtraction operations, the multiplication and division operations have two assembly language formats: one that uses registers for both source operands and another that uses a register and a small constant value for the source operands.

Table 4.2 summarizes the names for the signed and unsigned integer multiplication and division operations. Note that each operation has two SPARC operations: one that affects bits in the condition code register (e.g., smulcc), and another that does not (e.g., smul). Example 4.1 illustrates the use of these operations.

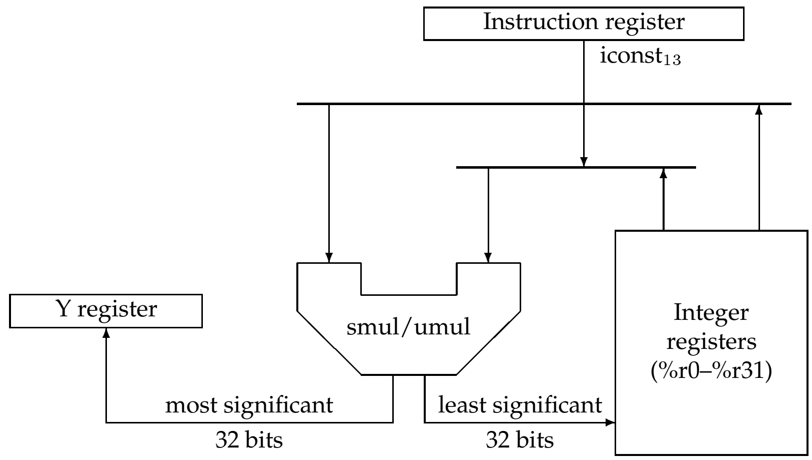


Figure 4.1 Integer multiplication

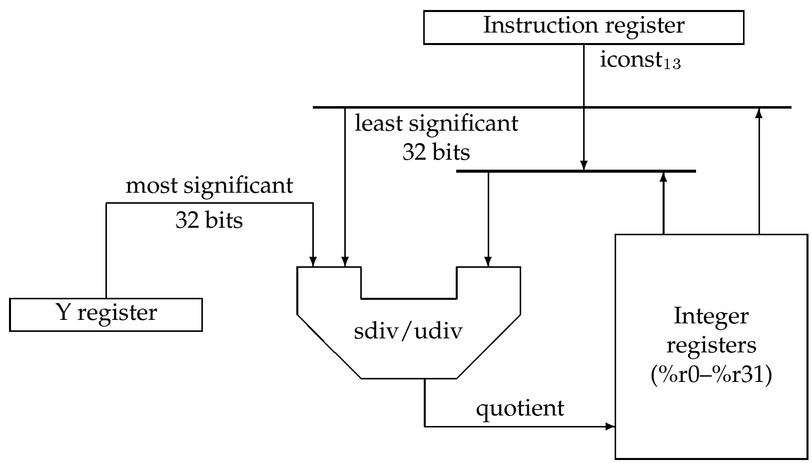


Figure 4.2 Integer division

Table 4.1 Assembly language formats for the integer multiplication and division operations

Operation	Assembler syntax		Operation implemented
integer multiplication	*mul-op*	rs_1, rs_2, rd	{%y, reg[rd]} = reg[rs_1] × reg[rs_2]
	mul-op	rs, $iconst_{13}$, rd	{%y, reg[rd]} = reg[rs] × $iconst_{13}$
integer division	*div-op*	rs_1, rs_2, rd	reg[rd] = {%y, reg[rs_1]} / reg[rs_2]
	div-op	rs, $iconst_{13}$, rd	reg[rd] = {%y, reg[rs]} / $iconst_{13}$

Notes:

- $iconst_{13}$ denotes an integer constant. This constant is signed when it used with a signed operation (e.g., smul) and unsigned when it is used with an unsigned operation (e.g., umul). The value must be represented in 13 bits.
- $\{x, y\}$ denotes a 64-bit value (or storage location) constructed from two 32-bit values x and y. The first of these values, x, is the most significant.

Table 4.2 The signed and unsigned integer multiplication and division operations

Operation	Operation names	
signed integer multiplication	smul	smulcc
unsigned integer multiplication	umul	umulcc
signed integer division	sdiv	sdivcc
unsigned integer division	udiv	udivcc

Example 4.1 *Write a SPARC program to evaluate the statement $a = (a * b)/c$. In writing this code you should assume that a, b, and c are signed integer values and that all results can be represented in 32 bits.*

```
        .data
a:      .word   0x42
b:      .word   0x43
c:      .word   0x44

        .text
start:  set     a, %r1
        ld      [%r1], %r2
        set     b, %r1
        ld      [%r1], %r3
        set     c, %r1
        ld      [%r1], %r4

        smul    %r2, %r3, %r2    ! a* b --> %y, %r2
        sdiv    %r2, %r4, %r2    ! %y, %r2 / c --> %r2

        set     a, %r1
        st      %r2, [%r1]       ! %r2 --> a

end:    ta      0
```

The signed and unsigned operations are distinguished by the way they interpret their operands. The signed operations interpret their source operands as signed integers and produce signed integer results. The unsigned operations interpret their source operands as unsigned integers and produce unsigned integer results.

4.3.2 Updating the condition code bits

The operations that have names ending in “cc” update the bits in the condition code register. The integer multiplication operations (smulcc and umulcc) always clear the V (overflow) and C (carry) bits of the condition code register. In addition, these operations update the N (negative) and Z (zero) bits of the condition code register. Although the multiplication operations produce a 64-bit result, updates to the N and Z flags are only based on the least significant 32 bits of the result. Like the multiplication operations, the division operations (sdivcc and udivcc) also clear the C bit in the condition code register. In addition, the division operations update the N, Z, and V bits in the condition code register based on the value of the 32-bit result.

4.3.3 Examining and setting the Y register

Note that the code presented in Example 4.1 does not need to examine or set the value in the Y register. The multiplication sets the Y register and the division uses the value set by the multiplication. In many other cases you will need to examine or set the contents of the Y register. In particular, you may need to examine the contents of the Y register after a multiplication or set the contents of the Y register before a division. You can use the (synthetic) mov operation introduced in Lab 2 to copy the contents of the Y register to an integer register or vice versa. You can also use the mov operation to set the contents of the Y register. Table 4.3 summarizes the mov operation as it applies to the Y register. The first format copies the contents of the Y register to an integer register. The second format copies the contents of an integer register to the Y register. The third instruction format stores a small integer constant into the Y register.

Table 4.3 The mov operation applied to the Y register.

Operation	Assembler syntax		Operation implemented
register copy	mov	%y, *rd*	reg[*rd*] = reg[%y]
	mov	*rs*, %y	reg[%y] = reg[*rs*]
register set	mov	$siconst_{13}$, %y	reg[%y] = $iconst_{13}$

When you store a value into the Y register (using a mov instruction), it takes three instruction cycles before the Y register is actually updated. This means that you need to make sure there are at least three instructions between an instruction that uses the Y register as a destination and an instruction that uses the value stored in the Y register.

Writing to the Y register Remember, always make sure that there are at least three instructions between any instruction that writes to the %y register and an instruction that uses the value in the %y register.

Example 4.2 *Write a SPARC assembly language fragment to evaluate the statement $a = (a + b)/c$. Again, you should assume that a, b, and c are signed integers and that all results can be represented in 32 bits.*

```
        .data
a:      .word   0x42
b:      .word   0x43
c:      .word   0x44

        .text
start:  mov     %r0, %y         ! clear the Y register -- THERE MUST BE
AT
                                ! LEAST 3 INSTRUCTIONS BETWEEN THE MOV
AND
                                ! SDIV INSTRUCTIONS
        set     a, %r1
        ld      [%r1], %r2
        set     b, %r1
        ld      [%r1], %r3
        set     c, %r1
        ld      [%r1], %r4
```

```
            add     %r2, %r3, %r2   ! a + b --> %r2
            sdiv    %r2, %r4, %r2   ! %r2 / c --> %r2

            set     a, %r1
            st      %r2, [%r1]      ! %r2 --> a
    end:    ta      0
```

4.3.4 The multiply step operation

Prior to Version 8 the SPARC did not have integer multiplication or division operations. These operations had to be performed using more primitive operations. To simplify integer multiplication, earlier versions of the SPARC provided a multiply step operation, "mulscc". We will consider this operation in Lab 13 when take a closer look at integer arithmetic on the SPARC.

4.4 Summary

In this lab we have covered the integer multiplication and division operations provided by the SPARC. As with the other arithmetic operation (add and sub), there are versions of the multiplication and division operations that update the condition code bits and other multiplication and division operations that do not alter the condition code bits.

All of the multiplication and division operations operations use a special purpose register, the Y register (%y). You can use the mov operation to examine and set the contents of the Y register.

4.5 Review Questions

1. How is the Y register used in the integer multiplication operations?
2. How is the Y register used in the integer division operations?

4.6 Exercises

Laboratory 5
Bit Manipulation and Character I/O

5.1 Goal

To cover the bit manipulation operations provided by the SPARC and the character I/O traps provided by ISEM.

5.2 Objectives

After completing this lab, you will be able to write assembly language programs that use:

- The bitwise operations,
- The shift operations, and
- The character I/O traps provided by ISEM.

5.3 Discussion

In this lab we introduce the bit manipulation operations of the SPARC. In particular, we consider the logical operations *and*, *or*, and *xor*. In addition, we consider the synthetic operation not. We follow this with a discussion of the shift operations *sll*, *srl*, and *sra*. The lab ends with a short discussion of the character I/O facilities provided by ISEM.

5.3.1 Bitwise operations

Table 5.1 summarizes the bitwise operations of the SPARC. Like the other data manipulation operations (e.g., add, sub, smul, sdiv, etc.), there are two instruction formats for each of the bitwise operations. Both formats use three explicit operands—two source operands and a destination operand. In the first format, both of the source operands are in registers. In the second format, one of the source operands is in a register, the other is a small constant value. This constant may be positive or negative; however its 2's complement representation must fit in 13 bits—the SPARC does sign extend this value. Examples 5.1 and 5.2 illustrate uses of the bitwise operations.

Example 5.1 *Write a SPARC program to evaluate the statement:* $a = (a\&b) \wedge (c|d)$.

```
        .data
a:      .word   0x42
b:      .word   0x43
c:      .word   0x44
d:      .word   0x45
```

Table 5.1 Bitwise operations

Operation	Assembler syntax		Operation implemented
and	and	rs_1, rs_2, rd	reg[rd] = reg[rs_1] & reg[rs_2]
	and	$rs_1, siconst_{13}, rd$	reg[rd] = reg[rs_1] & $siconst_{13}$
or	or	rs_1, rs_2, rd	reg[rd] = reg[rs_1] | reg[rs_2]
	or	$rs_1, siconst_{13}, rd$	reg[rd] = reg[rs_1] | $siconst_{13}$
exclusive or	xor	rs_1, rs_2, rd	reg[rd] = reg[rs_1] ∧ reg[rs_2]
	xor	$rs_1, siconst_{13}, rd$	reg[rd] = reg[rs_1] ∧ $siconst_{13}$
and not	andn	rs_1, rs_2, rd	reg[rd] = reg[rs_1] & ∼reg[rs_2]
	andn	$rs_1, siconst_{13}, rd$	reg[rd] = reg[rs_1] & ∼$siconst_{13}$
or not	orn	rs_1, rs_2, rd	reg[rd] = reg[rs_1] | ∼reg[rs_2]
	orn	$rs_1, siconst_{13}, rd$	reg[rd] = reg[rs_1] | ∼$siconst_{13}$
exclusive nor	xnor	rs_1, rs_2, rd	reg[rd] = reg[rs_1] ∧ ∼reg[rs_2]
	xnor	$rs_1, siconst_{13}, rd$	reg[rd] = reg[rs_1] ∧ ∼$siconst_{13}$

```
        .text
start:  set     a, %r1
        ld      [%r1], %r2              ! a --> %r2
        set     b, %r1
        ld      [%r1], %r3              ! b --> %r3
        set     c, %r1
        ld      [%r1], %r4              ! c --> %r4
        set     d, %r1
        ld      [%r1], %r5              ! d --> %r5

        and     %r2, %r3, %r2           ! a & b --> %r2
        or      %r4, %r5, %r4,          ! c | d --> %r4
        xor     %r2, %r4, %r2           ! %r2 ∧ %r4 --> r2

        set     a, %r1
        st      %r2, [%r1]              ! %r2 --> a
end:    ta      0
```

Example 5.2 *Write a SPARC program to clear bits 5 through 12 of the word n (bit 0 is the least significant bit).*

```
        .data
n:      .word   0xaaaaaaaa

        .text
start:  set     n, %r1
        ld      [%r1], %r2              ! n --> %r2

        andn    %r2, 0x1fe0, %r2        ! %r2 & ∼0x1fe0 --> %r2

        st      %r2, [%r1]              ! %r2 --> n
end:    ta      0
```

In this case, the mask (0x1fe0) can be represented using 12 bits and, as such, we can use an andn instruction with an immediate value.

Activity 5.1 *Write a SPARC program to clear bits 5 through 13 of the word n.*

5.3.2 Condition codes

Like the other data manipulation operations, there are separate bitwise operations that update the condition code register after they perform the specified operation. Table 5.2 summarizes the collection of bitwise operations that set the condition codes. The names for these operations have a suffix of "cc" to indicate that they update the bits in the condition code register.

Table 5.2 Bitwise operations—condition code versions

Operation	Operation name
and	andcc
or	orcc
exclusive or	xorcc
and not	andncc
or not	orncc
exclusive nor	xnorcc

In all cases, the V and C flags are cleared.
If the result is zero, the Z flag is set, otherwise Z is cleared.
The N flag is set to the most significant bit in the result.

5.3.3 Synthetic operation not

Bitwise inversion is provided by a synthetic operation. Table 5.3 summarizes this operation. Bitwise inversion does not affect the bits in the condition code register.

Table 5.3 Bitwise inversion

Operation	Assembler syntax	Operation implemented
not	not *rs*, *rd*	reg[*rd*] = ∼reg[*rs*]
	not *rd*	reg[*rd*] = ∼reg[*rd*]

5.3.4 Shift operations

Table 5.4 summarizes the shift operations of the SPARC. The SPARC provides two instruction formats for each of the shift operations. The shift operations provide a mechanism for moving every bit in a value to the left or right by a specified number of positions. The shift operations do not affect the bits in the condition code register.

Table 5.4 The shift operations

Operation	Assembler syntax	Operation implemented
left shift logical	sll rs_1, rs_2, rd	$reg[rd] = reg[rs_1] << reg[rs_2]$
	sll $rs_1, siconst_{13}, rd$	$reg[rd] = reg[rs_1] << siconst_{13}$
right shift logical	srl rs_1, rs_2, rd	$reg[rd] = reg[rs_1] >> reg[rs_2]$
	srl $rs_1, siconst_{13}, rd$	$reg[rd] = reg[rs_1] >> siconst_{13}$
right shift arithmetic	sra rs_1, rs_2, rd	$reg[rd] = reg[rs_1]_{31} \mid reg[rs_1] >> reg[rs_2]$
	sra $rs_1, siconst_{13}, rd$	$reg[rd] = reg[rs_1]_{31} \mid reg[rs_1] >> siconst_{13}$

As shown in Table 5.4, the shift operations have two source operands. The first operand specifies the value to be shifted. This value must be stored in an integer register. The second source operand specifies the amount of the shift. This operand may be stored in an integer register or it may be a small constant value. The processor only uses the least significant 5 bits of the second source operand (and ignores the remaining bits of this operand). Example 5.3 illustrates the shift operations.

Example 5.3 *Write a SPARC program to count the number of bits that are set (i.e., 1) in the memory location n (a variable). The result should be stored in %r2. In writing this code, you may use any of the remaining registers as temporaries.*

```
        .data
n:      .word 0xaaaaaaaa

        .text
start:  set     n, %r1
        ld      [%r1], %r3
        clr     %r2             ! clear the result

loop:   andcc   %r3, 1, %r0     ! check the lowest bit
        be      cont            !   skip if zero
        nop                     ! (delay slot)
        inc     %r2             ! increment the 1's count
cont:   srl     %r3, 1, %r3     ! shift data right, creating a new low
bit
        cmp     %r3, 0          ! if %r3 == 0, we're done
        bne     loop
        nop

end:    ta      0
```

5.3.5 Character data and character I/O in ISEM

When you use isem-as, character values are delimited using single quotes. For example, 'B' denotes the value associated with the letter B. When it encounters a character value, isem-as converts the character value into its ASCII representation. As such, writing 'B' in an assembly language program is equivalent to writing 0x42 (or 66). You should use the notation that best expresses the intent of your code.

ISEM provides the user with a primitive form of character input/output using the trap mechanism. We will discuss the trap mechanism in detail in Lab 16. For now, we note that the following instructions can be used to get and put characters from the standard I/O devices.

```
ta      1       ! %r8 --> putchar
ta      2       ! getchar --> %r8
```

The first of these instructions prints the character in the least significant byte of register %r8 (= %o0) to standard output and the second reads a character from standard input and places the result in the least significant byte of %r8, clearing the most significant 24 bits of this register. Example 5.4 illustrates the use of these I/O instructions.

Example 5.4 *Write a SPARC/ISEM program that reads a one digit number, adds five to the number and prints the result.*

```
        .text
start:  ta      2               ! digit --> %r8
```

```
        sub     %r8, '0', %r8    ! convert from digit to number
        add     %r8, 5, %r8      ! add 5
        cmp     %r8, 9           ! is the result greater than 9?
        ble     one_dg
        nop

        mov     %r8, %r7         ! copy %r8 into %r7
        set     '1', %r8         ! the most significant digit must be '1'
        ta      1                ! '1' --> putchar
        mov     %r7, %r8         ! restore %r8
        sub     %r8, 10, %r8

one_dg:                          ! %r8 holds a number less than 10
        add     %r8, '0', %r8    ! convert number to digit
        ta      1                ! print the least significant digit

        ta      0
```

5.4 Summary

In this lab we have introduced the bitwise and shift operations provided by the SPARC. In addition, we have introduced the notation used for character data and primitive mechanisms for character input and output. Example 5.5 illustrates all of these operations.

Example 5.5 *Write a SPARC/ISEM program to print the binary representation of the unsigned integer in memory location* n. *In writing the code, you may use any of the registers as temporaries.*

```
        .data
n:      .word   0xaaaaaaaa

        .text
start:  set     n, %r1
        ld      [%r1], %r2
        set     1 << 31, %r3   ! initialize the mask to start
                                 !   with the most significant bit
        set     32, %r4          ! number of bits to print

loop:   andcc   %r2, %r3, %g0    ! check for one
        be      print            !   branch on zero
        set     '0', %r8         ! (delay slot)

        set     '1', %r8         ! must have been a one

print:  ta      1                ! print %r8

        deccc   %r4              ! decrement count
        bg      loop             ! continue until count == 0
        srl     %r3, 1, %r3      ! shift mask right (bd)

end:    ta      0
```

5.5 Exercises

1. Write a SPARC program that compares the memory contents of the word pointed to by register $\%r2$ to the contents of register $\%r3$ on a bit-by-bit basis. For all bits i, if the value of bit i of $[\%r2]$ is smaller than the value of bit i of $\%r3$, set bit i of $\%r3$ to 1; otherwise, set bit i of $\%r3$ to 0.
2. Write a SPARC program that distinguishes ASCII-coded hexadecimal digits from other bytes and which converts valid digit codes to the corresponding hexadecimal value. **Only valid digit codes are to be converted. Leave invalid characters unaltered.** Suppose that register $\%r2$ holds the character on program entry, and that $\%r2$ holds the result on program exit. Use register $\%r3$ to report on validity. If valid, $\%r3$ must equal 0; if invalid, $\%r3$ must equal 1.

Valid Digits, Their Codes and Values

Digit	Digit Code	Hex Value
'0'	0x30	0x0
⋮	⋮	⋮
'9'	0x39	0x9
'a','A'	0x61,0x41	0xa
⋮	⋮	⋮
'f','F'	0x66,0x46	0xf

3. Write a SPARC program to print the hexadecimal representation of an unsigned integer. The unsigned integer should be named n and declared in the data segment. Your program should finish by printing a newline.

Laboratory 6
Assembler Directives, Assembler Expressions, and Addressing Modes

6.1 Goal

To cover several assembler directives and assembler expressions provided by the GNU assembler (gas) and the SPARC addressing modes.

6.2 Objectives

After completing this lab, you will be able to write assembly language programs that use:

- The assembler directives provided by the GNU assembler,
- Assemble expressions,
- The SPARC addressing modes in load and store instructions, and
- And use the SETHI instruction.

6.3 Discussion

In this lab we introduce three new assembler directives: a directive to allocate space, a directive to define symbolic constants, and a directive to include header files. After we describing these directives, we discuss assembler expressions and introduce the distinction between *relocatable* values and *absolute* values. We conclude this lab by discussing the memory addressing modes provided by the SPARC and the SETHI instruction.

6.3.1 Assembler directives

In Lab 2 we introduced three assembler directives: *.data*, *.text*, and *.word*. In this Lab, we introduce three more directives: *.skip*, *.set*, and *.include*. The *.skip* directive is used to allocate space. The *.set* directive is used to define a symbolic constant. The *.include* directive is used to include source (header) files.

You can use the *.skip* directive to allocate space in the current assembler segment (data or text). This directive takes one or two arguments. The first argument specifies the number of bytes to skip in the current assembler segment. The second argument specifies the value to be deposited in the skipped bytes. If the second argument is omitted, it is assumed to be zero.

You can define symbolic constants using the *.set* directive. This directive takes two arguments. The first argument is the name of the symbol to be defined. The second argument is an expression that defines the value of the symbol. This directive can be written using standard directive syntax (e.g., .set *symbol*, *expression*), or it can be written using the infix '=' operator (e.g., *symbol* = *expression*).

In many cases, you will want to collect a group of definitions for symbolic constants into a header file that can be included in several different programs or modules. (By including the same file in each of the programs or modules, you can be sure that all of the programs and modules use the same values for the symbolic constants.) The *.include* directive supports this style of programming. This directive takes a single argument, a string that gives the name of the file to include. The code from the included file logically replaces the *.include* directive. When the assembler is finished processing the included file, it resumes after the *.include* directive in the original file.

Table 6.1 summarizes the directives that we have introduced in this section.

Table 6.1 Assembler directives

Operation	Assembler syntax	
Allocate space	.skip	*n*
Symbolic constant	.set	*symbol*, *expression*
	symbol = *expression*	
Include file	.include	"*filename*"

6.3.2 Assembler Expressions

The *.set* and *.skip* directives use assembler expressions. In addition, you can use assembler expressions whenever you use a constant value in an assembly language instruction. Table 6.2 summarizes the operators that you can use constructing expressions. The operands can be expressions (using parentheses to override precedence), symbols (defined as labels or using the *.set* directive), or numbers.

Table 6.2 Assembler expressions

Operator	Operation	Precedence
−	Unary minus	highest
+	Unary plus	
*	Multiplication	high middle
/	Division	
<<	Left shift	
>>	Right shift	
|	Bitwise inclusive or	low middle
&	Bitwise and	
ˆ	Bitwise exclusive or	
!	Bitwise and not	
+	Addition	lowest
−	Binary subtraction	

When considering assembler expressions, it is useful to distinguish between *relocatable* values and *absolute* values. Labels are the simplest examples of relocatable values. They are relocatable because their final values depend on where your program is loaded into memory. Numbers are the simplest examples of absolute values. Absolute values do not depend on where your program is loaded into memory.

You can use absolute values with any of the operators. If all of the operands for an operator are absolute values, the expression using the operator is an absolute value.

You can only use relocatable values in expressions using the binary addition and subtraction operators. When you use binary addition, at most one operand can be a relocatable value, the other operand must be an absolute value. If one operand is relocatable, the value of the expression is relocatable.

When you use binary subtraction, you cannot subtract a relocatable value from an absolute value. When subtract an absolute value from a relocatable value, the result is a relocatable value. When you subtract two relocatable values, the two values must be defined in the same assembler segment (e.g., text or data), and the result is an absolute value.

6.3.3 Addressing modes

The SPARC supports two addressing modes: register indirect with index and register indirect with displacement. In the first mode, the effective address is calculated by adding the contents two integer registers. This addressing mode is commonly used to access an array element: one of the registers holds the base address of the array, the other holds the (scaled) index of the element.

In the second mode, the effective address is calculated by adding a 13-bit signed integer constant to a register. This addressing mode can be used with pointers to structures: the register holds the address of the structure and the integer constant specifies the offset of the member (field) being accessed. Register indirect with displacement addressing is also commonly used when accessing items on the runtime stack (e.g., parameters and local variables). We consider access to the runtime stack in Lab 12 when we consider standard procedure calling conventions for the SPARC.

Table 6.3 summarizes the addressing modes supported by SPARC assemblers. In addition to the two basic addressing modes, SPARC assemblers recognize register indirect addressing and a limited form of direct memory addressing. Direct memory addresses are limited to values that can be expressed in 13-bits when sign-extended (i.e., very small addresses and very large addresses).

Table 6.3 Assembler address specifications

Addressing mode	Assembler syntax	Implementation	Effective address
register indirect with index	$[r_1+r_2]$	basic mode	reg$[r_1]$+reg$[r_2]$
register indirect with displacement	$[r_1+siconst_{13}]$	basic mode	reg$[r_1]+siconst_{13}$
	$[siconst_{13}+r_1]$		$siconst_{13}+$reg$[r_1]$
	$[r_1-siconst_{13}]$		reg$[r_1]-siconst_{13}$
register indirect	$[r]$	$[r$+%r0$]$	reg$[r]$
direct memory	$[siconst_{13}]$	[%r0+$siconst_{13}$]	$siconst_{13}$

Addressing modes can only be used with the load and store instructions. Table 6.4 summarizes the load word and store word operations.

6.3.4 The SETHI instruction

We conclude this discussion by introducing another SPARC instruction, sethi, and the %hi and %lo operators provided by SPARC assemblers. The sethi instruction takes two arguments: a 22-bit constant and a destination register. This instruction sets the most significant 22 bits of the destination register and clears the least significant 10 bits of this register.

Table 6.4 The SPARC ld and st operations

Operation	Assembler syntax		Operation implemented
load word	ld	*address*, *rd*	reg[*rd*] = memory[eff_addr(*address*)]
store word	st	*rs*, *address*	memory[eff_addr(*address*)] = reg[*rs*]

Notes:
address an address specification (see Table 6.3)
eff_addr(x) the result of the effective address calculation

Example 6.1 *Show how you would load a 32-bit, bignum, into %r2 without using the set instruction.*

```
.set    bignum, 0x87654321
sethi   bignum>>10, %r2
or      %r2, bignum&0x3ff, %r2
ta      0
```

Note the use of the expression "bignum>>10" to extract the most significant 22 bits of bignum and the expression "bignum&0x3ff" to extract the least significant 10 bits of bignum. To make your code more readable, SPARC assemblers provide two special operators: %hi(x) yields the most significant 22 bits of x, while %lo(x) yields the least significant 10 bits of x. Note that these operators are written using function call notation.

Example 6.2 *Rewrite the code fragment given in Example 6.1, using the %hi and %lo operators.*

```
.set    bignum, 0x87654321
sethi   %hi(bignum), %r2
or      %r2, %lo(bignum), %r2
ta      0
```

Example 6.3 *Write an assembly language fragment to sum up the elements in the array. Give directives to declare an array of 20 words and an additional word to hold the sum.*

```
        .data
arr:    .skip   20*4                ! allocate an array of 20 words
sum:    .word   0                   ! allocate a word to hold the sum

        .text
start:  set     arr, %r2            ! %r2 is the base address
        mov     %r0, %r3            ! %r3 is the index value
        mov     %r0, %r4            ! %r4 is the running sum
        set     20, %r5             ! %r5 is the number of elems to add

loop:   ld      [%r2+%r3], %r6      ! fetch the next element
        add     %r4, %r6, %r4       ! add it to the running sum
        subcc   %r5, 1, %r5         ! one fewer element
        bne     loop                ! if %r5 > 0 get next element
        add     %r3, 4, %r3         ! increment the index (DELAY SLOT)

        sethi   %hi(sum), %r1       ! store the result in sum
        st      %r4, [%r1+%lo(sum)]
end:    ta      0
```

Note that the code in Example 6.3 stores the result into sum using a *sethi* instruction followed by a *st* instruction. In previous examples we have used a *set* instruction followed by a *st* instruction to accomplish the same task. However, the *set* instruction is actually a synthetic instruction and the assembler implements this instruction using a *sethi* instruction followed by an *or* instruction (as we showed in Example 6.1). As such, our earlier code actually requires three instruction for every (load or) store. Using the *sethi* instruction directly, we can avoid an unnecessary instruction.

6.4 Summary

6.5 Review Questions

6.6 Exercises

1. The %hi operator yields the most significant 22 bits while %lo operator yields the least significant 10 bits of a 32-bit value. Considering that the SPARC uses 13-bit signed integers in lots of contexts, it might seem that it would be better to have the %hi and %lo operators yield 19 and 13 bits respectively. What problems would this cause?
2. Write a SPARC assembly language program to count the number of ones in a bit string. The bit string should be named BitString and the length (in bits) of the bit string should be named Length. The result should be stored as a word named Count. Note: there is no limit on the number of bits in the bit string.
3. Given an array, A, and the number of elements in the array, n, write a SPARC program to sort the array. You may use any method you like to sort the array.

Laboratory 7
Operand Sizes and Unsigned Values

7.1 Goal

To complete our coverage of the load and store instructions provided by the SPARC and to cover a collection of useful synthetic operations.

7.2 Objectives

After completing this lab, you will be able to write assembly language programs that use:

- Byte, halfword, word, and double word operands,
- The assembler directives .byte, .hword, .quad, and .align,
- Signed and unsigned operands, and
- The (synthetic) operations clear, negate, increment, and decrement.

7.3 Discussion

In this lab we introduce the assembler directives and operations associated with different sized operands and unsigned operands. We begin by considering the operand sizes supported by the SPARC. Then we consider assembler directives used to allocate different amounts of memory. Next, we consider the load and store operations for different sized operations. The we consider the load operations for unsigned operands. Finally, we conclude by considering a small collection of useful synthetic operations.

7.3.1 Operand sizes

On the SPARC, all data manipulation operations (e.g., integer addition and integer subtraction) manipulate 32-bit values. However, the data transfer operations (e.g., load and store) can transfer different sized values between the memory and the integer registers. Table 7.1 summarizes the operand sizes supported by the SPARC load and store operations.

Assembler directives

The .word directive introduced in Lab 2 allocates and initializes memory in 32-bit (word) units. Table 7.2 summarizes the assembler directives for allocating and initializing different sized units of memory. The name of the directive used to allocate and initialize a 64 bit doubleword, ".quad", is historical. The name dates to a time when words were 16 bits. Quad is short for quad-word, that is, four 16-bit words.

Table 7.1 Operand sizes for the data transfer operations

Name	Size
byte	8 bits
halfword	16 bits
word	32 bits
double word	64 bits

Table 7.2 Directives for allocating and initializing memory

Directive	Size
.byte	8 bits
.hword	16 bits
.word	32 bits
.quad	64 bits

The SPARC load and store operations require that halfword values be aligned on even addresses (i.e., halfword alignment) and that word and double word values be aligned on addresses that are a multiple of four (i.e., word aligned). You can use the .align directive to make sure that your variables are aligned as needed. This directive takes a two arguments, a number and an optional pad value. When the assembler encounters an align directive, it makes sure that the next address in the current assembler segment is a multiple of the first argument. For example, the directive ".align 8" will ensure that the next address is a multiple of 8. To ensure that the next address meets the alignment requirements, the assembler emits "pad" bytes. If the second argument is supplied, the assembler uses this value when it emits pad bytes; otherwise, the assembler emits zeros. Example 7.1 illustrates the size and alignment directives.

Example 7.1 *Consider the following C declarations. Assuming that a character is one byte, a short integer is two bytes, and an integer is four bytes, give assembler directives to allocate and initialize the memory specified by these directives.*

```
short int short1 = 22;
char ch1 = 'a';
short int short2 = 33;
char ch2 = 'A';
int int1 = 0;
```

```
        .data
        .align 2        ! halfword align
short1: .hword 22       ! allocate and initialize a halfword
ch1:    .byte 'a'       ! allocate and initialize a byte
        .align 2        ! halfword align
short2: .hword 33       ! allocate and initialize a second halfword
ch2:    .byte 'A'       ! allocate and initialize a second byte
        .align 4        ! word align
int1:   .word 0         ! allocate and initialize a word
```

The load and store operations

The size of the operand for a load or store operation is specified using a suffix: "d" for doubleword, "h" for halfword, or "b" byte. The operation names are summarized in Table 7.3.

Table 7.3 The load and store operations

Size	Load	Store	Notes
byte	ldb	stb	
halfword	ldh	sth	the address must be halfword aligned
word	ld	st	the address must be word aligned
doubleword	ldd	std	the address must be word aligned and the register must be an even number

The load operations take two operands: the (source) memory address followed by the (destination) register. Similarly, the store operations take two operands: the (source) register followed by the (destination) memory address.

The load byte, halfword, and word operations set all 32 bits of the destination register. The load byte operation (ldb) fetches an 8-bit value, sign extends this value to 32 bits, and loads the resulting value into destination register. The load halfword operation (ldh) fetches a 16-bit value and sign extends this value to 32 bits. The load word operation (load) fetches a 32-bit value and load this value into the destination register. The load doubleword operation fetches two 32-bit values and loads them into consecutive registers—starting with the register specified in the instruction (e.g., %r2 and %r3).

The store byte operation (stb) stores the least significant 8 bits of the source register to the destination memory location. The store halfword operation (sth) stores the least significant 16 bits of the source register into the destination memory location. The store word operation (st) stores the contents of a register to the destination memory address. The store doubleword operation (stored) stores the contents two consecutive registers (starting with the register specified in the instruction).

The memory addresses used with the operations that load and store halfwords must be even (i.e., halfword aligned). The memory addresses used with the operations that load and store words and doublewords must be multiples of four (i.e., word aligned). The register used with the operations that load and store doublewords must be even (e.g., %r2 but not %r3). When these operations are used, the most significant 32 bits are stored in the even register.

Table 7.4 The load and store instructions

Operation	Instruction syntax	Operation implemented
load byte	ldb *address*, *rd*	reg[*rd*] = signextend(memory[*address*]$_8$)
load halfword	ldh *address*, *rd*	reg[*rd*] = signextend(memory[*address*]$_{16}$)
load word	ld *address*, *rd*	reg[*rd*] = memory[*address*]$_{32}$
load doubleword	ldd *address*, *rd*	reg[*rd*] = memory[*address*]$_{32}$ reg[*rd*+1] = memory[*address*+4]$_{32}$
store byte	stb *rs*, *address*	memory[*address*]$_8$ = reg[*rs*]$_8$
store halfword	sth *rs*, *address*	memory[*address*]$_{16}$ = reg[*rs*]$_{16}$
store word	st *rs*, *address*	memory[*address*]$_{32}$ = reg[*rs*]$_{32}$
store doubleword	std *rs*, *address*	memory[*address*]$_{32}$ = reg[*rs*]$_{32}$ memory[*address*+4]$_{32}$ = reg[*rs*+1]$_{32}$

Example 7.2 *Rewrite the code presented in Example 6.3 using an array of 20 bytes (i.e., chars) instead of words. (You should still store the sum in a word.)*

```
        .data
```

```
arr:    .skip   20                    ! allocate an array of 20 bytes
sum:    .word   0                     ! allocate a word to hold the sum

        .text
start:  set     arr, %r2              ! %r2 is the base address
        mov     %r0, %r3              ! %r3 is the index value
        mov     %r0, %r4              ! %r4 is the running sum
        set     20, %r5               ! %r5 is the number of elems to add

loop:   ldb     [%r2+%r3], %r6        ! fetch the next element
        add     %r4, %r6, %r4         ! add it to the running sum
        subcc   %r5, 1, %r5           ! one fewer element
        bne     loop                  ! if %r5 > 0 get next element
        add     %r3, 4, %r3           ! increment the index (DELAY SLOT)

        sethi   %hi(sum), %r1         ! store the result in sum
        st      %r4, [%r1+%lo(sum)]
end:    ta      0
```

7.3.2 Unsigned operands

The SPARC instruction set also provides a load unsigned byte operation (loadub) and a load unsigned halfword operation (loaduh). In contrast to the standard load operations (loadb and loadh), these operations do not sign extend their values. They always set the most significant bits to zero.

Table 7.5 The unsigned load operations

Operation	Instruction syntax	Operation implemented
load unsigned byte	ldub *address*, *rd*	reg[*rd*] = zerofill(memory[*address*]$_8$)
load unsigned halfword	lduh *address*, *rd*	reg[*rd*] = zerofill(memory[*address*]$_{16}$)

7.3.3 Useful (synthetic) operations

We conclude this lab by considering a collection of useful operations. In particular, we consider the (synthetic) operations clear, negate, increment, and decrement. These operations are summarized in Table 7.6.

Example 7.3 *Rewrite the code presented in Example 6.3 using the operations defined in Table 7.6.*

```
        .data
arr:    .skip   20*4                  ! allocate an array of 20 words
sum:    .word   0                     ! allocate a word to hold the sum

        .text
start:  set     arr, %r2              ! %r2 is the base address
        clr     %r3                   ! %r3 is the index value
        clr     %r4                   ! %r4 is the running sum
        set     20, %r5               ! %r5 is the number of elems to add

loop:   ld      [%r2+%r3], %r6        ! fetch the next element
        add     %r4, %r6, %r4         ! add it to the running sum
```

Table 7.6 Some useful operations

Operation	Instruction syntax		Operation implemented
clear register	clr	*rd*	reg[*rd* = 0
clear memory word	clr	*address*	memory[*address*]$_{32}$ = 0
clear memory halfword	clrh	*address*	memory[*address*]$_{16}$ = 0
clear memory byte	clrb	*address*	memory[*address*]$_{8}$ = 0
negate register	neg	*rd*	reg[*rd*] = −reg[*rd*]
	neg	*rs,rd*	reg[*rd*] = −reg[*rs*]
increment register	inc	*rd*	reg[*rd*] = reg[*rd*] + 1
	inc	$siconst_{13}$, *rd*	reg[*rd*] = reg[*rd*] + $siconst_{13}$
increment register, set cc	inccc	*rd*	reg[*rd*] = reg[*rd*] + 1
	inccc	$siconst_{13}$, *rd*	reg[*rd*] = reg[*rd*] + $siconst_{13}$
decrement register	dec	*rd*	reg[*rd*] = reg[*rd*] − 1
	dec	$siconst_{13}$, *rd*	reg[*rd*] = reg[*rd*] − $siconst_{13}$
decrement register, set cc	deccc	*rd*	reg[*rd*] = reg[*rd*] − 1
	deccc	$siconst_{13}$, *rd*	reg[*rd*] = reg[*rd*] − $siconst_{13}$

```
        deccc   %r5                     ! one fewer element
        bne     loop                    ! if %r5 > 0 get next element
        inc     4, %r3                  ! increment the index (DELAY SLOT)

        sethi   %hi(sum), %r1           ! store the result in sum
        st      %r4, [%r1+%lo(sum)]
end:    ta       0
```

7.4 Summary

7.5 Review Questions

1. Explain why the SPARC does not provide unsigned store operations.

7.6 Exercises

1. Suppose that the SPARC did not have a load unsigned byte operation, explain how you could implement this operation using the load byte operation (recall, this operation always sign extends the value being loaded). Note, the Intel i860 processor provides a load byte operation but does not provide a load unsigned byte operation.

Laboratory 8
The ISEM Graphics Accelerator

8.1 Goal

To cover uses of the graphics accelerator device provided by ISEM. (Currently, this device is only available in the X11 environment.)

8.2 Objectives

After completing this lab, you will be able to write assembly language programs that use:

- the graphics accelerator.

8.3 Discussion

In the previous labs, we have focused on assembly language programming, presenting SPARC instructions and assembler directives. Now, it's time for some fun! In this lab we present the ISEM graphics accelerator device, gx. In addition to showing you how a simple device works, this lab will give you an opportunity to review the assembly language constructs covered in the previous labs.

When you open the gx device, it creates a black and white graphics window. By issuing gx commands, you can instruct the gx device to draw lines, fill rectangles, and copy rectangles in the window. The visible gx display is 512×512 pixels. Individual pixels in the visible region are addressed by an (x, y) pair. The pixel in the upper left corner of the display is addressed by the pair $(0, 0)$. The pixel in the lower right corner is addressed by the pair $(511, 511)$.

In addition to the visible pixels, the gx device provides a 512x64 rectangle of pixels that are not displayed. These pixels are commonly used with the *blt* operation (described later in this lab). They are addressed using the pixel addresses $(0, 512)$ through $(511, 575)$. Figure 8.1 illustrates the pixel addresses provided by the gx device.

The gx device is a "memory mapped" device. This means that the gx device registers are mapped into memory locations and can be accessed using the standard load and store operations. (The SPARC architecture doesn't provide any special I/O instructions, so all devices must be memory mapped when they are used with a SPARC.) The gx device has 256 registers: a status register, a command register, and 254 argument registers. Each register is one word (four bytes, 32 bits) wide.

The status register is mapped into memory location 0x100000. Storing a value into this location has no affect; however, when you load a register using this memory location, you will actually read the status register of the gx device. The value of the status register is 0 when the gx device hasn't been (opened and) displayed. This register has the value 1 when the device has been opened and mapped onto the display. The gx command register is

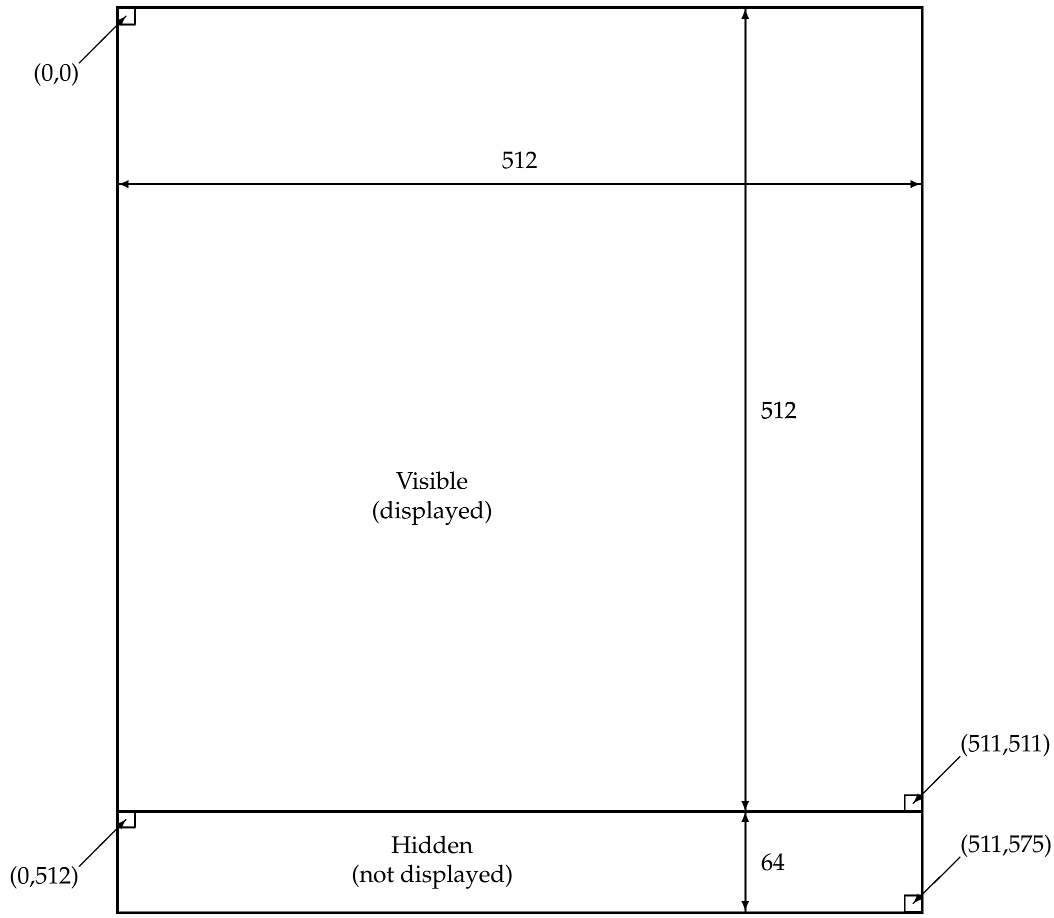


Figure 8.1 The gx display

mapped to memory location 0x100004 while the argument registers are mapped to memory locations 0x100008–0x1007fe. Figure 8.2 illustrates the mapping of gx registers into the ISEM memory.

The gx device provides commands to draw lines, fill rectangles, and copy rectangles on the display. Table 8.1 summarizes the commands provided by the gx device.

Before you issue any other gx commands, you must first open the device. When you issue the open command, the gx device creates an X11 window and initializes the display memory. The display is initially is solid white, and the drawing color is black.

After you are done using the gx device, you can explicitly close the device using the close command. If you don't issue a close command, the display window will be destroyed when you exit isem, as such, closing the gx device is not critical.

You can use the color command to set the color used to draw lines and fill rectangles. This command takes a single argument, the color. If the argument is 0, the gx device draws lines an fills rectangles in black; otherwise, if the argument is 1, the gx device draws lines and fills rectangles in white.

The gx_op command has a single argument. This command sets the drawing function to the value of the argument. When the gx device is initialized, the drawing function is "copy". That is, the gx device simply copies the drawing color (when drawing a line or

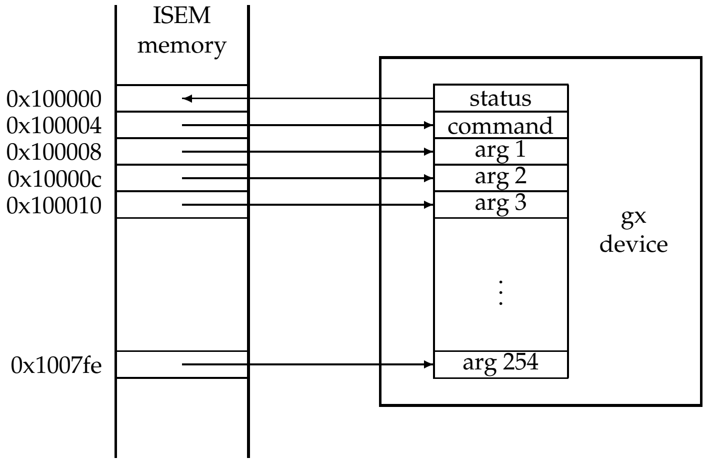


Figure 8.2 The gx register/memory map

Table 8.1 Commands provided by the graphics accelerator

Command value	Command name	Arguments	Operation performed
0	open		open the device
1	close		close the device
2	color	n	set the drawing color ($n = 0$ for black, $n = 1$ for white)
3	gx_op	n	set the “drawing function” (see Table 8.2)
4	line	$x_1\ y_1\ x_2\ y_2$	draw a line from pixel (x_1, y_1) to (x_2, y_2)
5	fill	$x\ y\ w\ h$	fill the rectangle with upper left corner (x, y), width w, and height h
6	blt	$x_1\ y_1\ w\ h\ x_2\ y_2$	copy the rectangle with upper left corner (x_1, y_1), width w, and height h to the rectangle with upper left corner (x_2, y_2)

filling a rectangle) or the source pixel (when copying a rectangle) to the destination pixel. Table 8.2 summarizes the other drawing functions provided by the gx device.

The line function draws a line from one pixel to another based on the current drawing color and function. This command takes four arguments: two arguments to specify the the coordinates for each pixel.

The fill command fills a rectangle based on the current drawing color and function. This command takes four arguments: the first two arguments specify the coordinates of the upper left corner of the rectangle, the third argument specifies the width of the rectangle, and the fourth argument specifies the height of the rectangle.

The blt command copies (subject to the drawing function) a rectangle from one part of the gx memory to another. This command takes six arguments: the first four arguments specify the source rectangle, the last two specify the upper left corner of the destination rectangle.

To issue a gx command, you first store the arguments in the gx argument registers and then store the command into the gx command register. The order in which you perform

Table 8.2 Drawing operation supported by the graphics accelerator

Name	Value	Drawing operation
clear	0x0	0
and	0x1	source AND destination
andReverse	0x2	source AND NOT destination
copy	0x3	source
andInverted	0x4	NOT source AND destination
noop	0x5	destination
xor	0x6	source XOR destination
or	0x7	source OR destination
nor	0x8	NOT source AND NOT destination
equiv	0x9	NOT source XOR destination
invert	0xa	NOT destination
orReverse	0xb	source OR NOT destination
copyInverted	0xc	NOT source
orInverted	0xd	NOT source OR destination
nand	0xe	NOT source OR NOT destination
set	0xf	1

these stores is critical. You must load the command register after you have loaded the argument registers. The gx device reads its argument registers as soon as you store a value in the command register.

Symbolic constant that make devices, like the gx device, easier to use are commonly defined in header files. Figure 8.3 presents a header file for the gx device. Example 8.1 illustrates the gx device and a use of the gx header file.

Example 8.1 *Write an ISEM program that opens the gx device and draws a line from the upper left corner to lower right corner.*

```
        .include "gx.h"

        .text
main:   set     BX_BUFFER, %r1          ! %r1 points to gx registers

        st      %r0, [%r1+GX_CMD]       ! open display
        ld      [%r1+GX_STATUS], %r2    ! load status word
wait:   cmp     %r2, 0
        be      wait                    ! wait until window is mapped
        ld      [%r1], %r2              ! load status word

        ! set up the command arguments
        st      %r0, [%r1+GX_LINE_X1]   ! x1 = 0
        st      %r0, [%r1+GX_LINE_Y1]   ! y1 = 0
        mov     511, %r2
        st      %r2, [%r1+GX_LINE_X2]   ! x2 = 511
        st      %r2, [%r1+GX_LINE_Y2]   ! y2 = 511

        ! now, issue the command
        mov     GX_LINE, %r2
        st      %r2, [%r1+GX_CMD]
```

```
!
! gx.h -- symbolic constants for the gx device
!
.set GX_BUFFER,          0x100000  ! start address for GX registers

.set GX_OPEN,             0         ! command numbers
.set GX_CLOSE,            1
.set GX_COLOR,            2
.set GX_OP,               3
.set GX_LINE,             4
.set GX_FILL,             5
.set GX_BLIT,             6

.set GX_STATUS,           0         ! symbolic buffer offsets
.set GX_CMD,              4
.set GX_ARG,              8

.set GX_FILL_X,           8         !   for fill
.set GX_FILL_Y,          12
.set GX_FILL_W,          16
.set GX_FILL_H,          20

.set GX_LINE_X1,          8         !   for line
.set GX_LINE_Y1,         12
.set GX_LINE_X2,         16
.set GX_LINE_Y2,         20

.set GX_BLIT_X1,          8         !   for blit
.set GX_BLIT_Y1,         12
.set GX_BLIT_W,          16
.set GX_BLIT_H,          20
.set GX_BLIT_X2,         24
.set GX_BLIT_Y2,         28

! drawing functions for the GX_OP command
.set GX_CLEAR,           0x0        ! 0
.set GX_AND,             0x1        ! source AND destination
.set GX_AND_REVERSE,     0x2        ! source AND NOT destination
.set GX_COPY,            0x3        ! source
.set GX_AND_INVERTED,    0x4        ! NOT source AND destination
.set GX_NOOP,            0x5        ! destination
.set GX_XOR,             0x6        ! source XOR destination
.set GX_OR,              0x7        ! source OR destination
.set GX_NOR,             0x8        ! NOT source AND NOT destination
.set GX_EQUIV,           0x9        ! NOT source XOR destination
.set GX_INVERT,          0xa        ! NOT destination
.set GX_OR_REVERSE,      0xb        ! source OR NOT destination
.set GX_COPY_INVERTED,   0xc        ! NOT source
.set GX_OR_INVERTED,     0xd        ! NOT source OR destination
.set GX_NAND,            0xe        ! NOT source OR NOT destination
```

Figure 8.3 The gx header file. gx.h

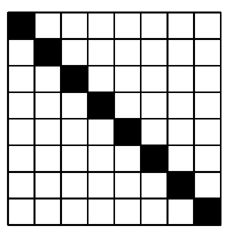
```
        ! all done -- the display will remain until you exit isem
end:    ta      0
```

8.4 Summary

8.5 Review Questions

8.6 Exercises

1. A bitmap is rectangle of black and white pixels. In X11, bitmaps are stored row-by-row in an array of bytes, i.e., consecutive memory locations. Each bit in this array represents a pixel on the screen—1 for black and 0 for white (the inverse of the color convention used for the gx device). Each row of the bitmap is stored in an integral number of bytes. If the number of columns is not a multiple of 8, the last byte is padded with zeros. The first byte of the array represents the leftmost 8 pixels in the top row. The next byte represents the next 8 pixels in the top row or the first 8 pixels of the next row if there are fewer than 9 columns in the bitmap. Within a byte, the least significant bit represents the leftmost pixel.
   As an example, Figure 8.4 illustrates a simple bitmap and assembly language declarations for the X11 representation of this bitmap.



```
height:  .word   8
width:   .word   8
bits:    .byte   0x01, 0x02, 0x04, 0x08
         .byte   0x10, 0x20, 0x40, 0x80
```

Figure 8.4 A simple bitmap

   You can use the "bitmap" program under X11 to create bitmaps and save their representation in a file. When you save a bitmap, the bitmap program generates C declarations for the bitmap. You can easily convert these declarations into assembly language declarations.
   Write a program to draw a bitmap on the gx display. The gx device does not provide a simple way to copy bytes from standard memory to the device memory. You will need to "draw" the bitmap into the display memory, using GX_LINE commands. Your program should draw the bitmap with its upper left corner at position (0,0) of the display. You can test your program with the bitmap shown in Figure 8.4, but make sure you can also display bitmaps with widths that are not a multiple of 8. Figure 8.5 gives you another bitmap to display.

```
width:    .word  31
height:   .word  13
bits:     .byte  0x00, 0x00, 0x00, 0x00, 0x7e, 0xbf, 0xdf, 0x18, 0x7e
          .byte  0xbf, 0xdf, 0x1d, 0x18, 0x83, 0xc1, 0x1f, 0x18, 0x83
          .byte  0xc1, 0x1f, 0x18, 0xbf, 0xc7, 0x1a, 0x18, 0xbf, 0xc7
          .byte  0x18, 0x18, 0xb0, 0xc1, 0x18, 0x18, 0xb0, 0xc1, 0x18
          .byte  0x7e, 0xbf, 0xdf, 0x18, 0x7e, 0xbf, 0xdf, 0x18, 0x00
          .byte  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
```

Figure 8.5 A simple bitmap

Laboratory 9
The SPARC Instruction Formats

9.1 Goal

To cover the instruction encoding and decoding for the SPARC.

9.2 Objectives

After completing this lab, you will be able to:

- Hand assemble SPARC assembly language instructions, and
- Hand disassemble SPARC machine language instructions.

9.3 Discussion

In this lab we consider instruction encoding and decoding for the operations that we have introduced in previous labs. In particular, we will consider encodings for instructions that use the data manipulation and branching operations. After we introduce instruction encoding, we consider the translation of synthetic operations. Finally, we conclude this lab by considering instruction decoding on the SPARC.

All SPARC instructions are encoded in a single 32-bit instruction word, there are no extension words.

9.3.1 Encoding load and store instructions

The SPARC machine language uses two different formats for load and store instructions. These formats are shown in Figure 9.1. The first format is used for instructions that use one or two registers in the effective address. The second format is used for instructions that use an integer constant in the effective address.

In the first format the 32-bit instruction is divided into seven fields. The first field (reading from the left) holds the 2-bit value 11, while the fifth field (bit 13) holds the 1-bit value 0. These bits are the same for all load and store instructions that use two source registers. The sixth field (bits 5 through 12) holds the address space indicator, *asi*. For the present, we will always set the *asi* field to zero. The remaining fields, rd, op_3, rs_1, and rs_2, hold encodings for the destination register, the operation, and the two source registers, respectively.

Registers are encoded using the 5-bit binary representation of the register number. Table 9.1 summarizes the operation encodings for the load and store operations.

Example 9.1 *Hand assemble the instruction:*

```
ldd     [%r4+%r7], %r11
```

A. Instructions of the form: op $[rs_1+rs_2]$, rd (load instructions) or
op rd, $[rs_1+rs_2]$ (store instructions)

31 30	29 … 25	24 … 19	18 … 14	13	12 … 5	4 … 0
11	rd	op_3	rs_1	0	asi	rs_2

B. Instructions of the form: op $[rs_1+siconst_{13}]$, rd (load instructions) or
op rd, $[rs_1+siconst_{13}]$ (store instructions)

31 30	29 … 25	24 … 19	18 … 14	13	12 … 0
11	rd	op_3	rs_1	1	$siconst_{13}$

Figure 9.1 Instruction formats for load and store instructions

Table 9.1 Operation encodings for the load and store operations

Operation	op_3	Operation	op_3
ld	000000	st	000100
ldub	000001	stb	000101
lduh	000010	sth	000110
ldd	000011	std	000111
ldsb	001001		
ldsh	001010		

Because this instruction uses two registers in the address specification, it is encoded using the first format shown in Figure 9.1. As such, we must determine the values for the *rd*, op_3, rs_1, and rs_2 fields. The following table summarizes these encodings:

Field	Symbolic value	Encoded value
rd	%r11	01011
op_3	ldd	000011
rs_1	%r4	00100
rs_2	%r7	00111

These encodings lead to the following machine instruction:

31 30	29 … 25	24 … 19	18 … 14	13	12 … 5	4 … 0
11	01011	000011	00100	0	00000000	00111

That is, 1101 0110 0001 1001 0000 0000 0000 0111 in binary, or 0xD6190007.

If the assembly language instruction only uses a single register in the address specification (e.g., register indirect addressing), the register is encoded in one of the source register fields (i.e., sr_1 or sr_2) while %r0 is encoded in the other. It doesn't matter which field holds the register specified in the assembly language instruction and which field holds the encoding for %r0. However, *isem-as* encodes %r0 in sr_2.

Example 9.2 *Hand assemble the instruction:*

```
ldub    [%r23], %r19
```

Because this instruction uses registers in the address specification, it is encoded using the first format shown in Figure 9.1. As such, we must determine the values for the *rd*, op_3, rs_1, and rs_2 fields. The following table summarizes these encodings:

Field	Symbolic value	Encoded value
rd	%r19	10011
op_3	ldub	000001
rs_1	%r23	10111
rs_2	%r0	00000

These encodings lead to the following machine instruction:

31 30	29 25	24 19	18 14	13	12 5	4 0
11	10011	000001	10111	0	00000000	00000

That is, 1110 0110 0000 1101 1100 0000 0000 0000 in binary, or 0xE60DC000.

In the second format the 32-bit instruction is divided into six fields. As in the previous format, the first field holds the 2-bit value 11. However, unlike the previous format, the fifth field holds the 1-bit value 1. The remaining fields, *rd*, op_3, rs_1, and $siconst_{13}$, hold encodings for the destination register, the operation, the source register, and the constant value, respectively. When this format is used, the integer constant is encoded using the 13-bit 2's complement representation and stored in the $siconst_{13}$ field of the instruction.

9.3.2 Encoding sethi instructions

The format used to encode sethi instructions is shown in Figure 9.2. Sethi instructions are encoded in four fields. The first field holds the 2-bit value 00. The next field, *rd*, holds the 5-bit encoding of the destination register. The third field holds the 3-bit value 100. The final filed holds the 22-bit binary encoding of the value specified in the instruction.

31 30	29 25	24 22	21 0
00	rd	100	$const_{22}$

Figure 9.2 Instruction format for sethi instructions

Example 9.3 *Hand assemble the instruction:*

```
sethi   %hi(0x87654321), %r2
```

This instruction is encoded using the format shown in Figure 9.2. As such, we need to determined the values for the *rd* and $const_{22}$ fields. The following table summarizes these encodings:

Field	Symbolic value	Encoded value
rd	%r2	00010
$const_{22}$	%hi(0x87654321)	1000 0111 0110 0101 0100 00

These encodings lead to the following machine instruction:

31 30	29 25	24 22	21 0
00	00010	100	1000 0111 0110 0101 0100 00

That is, 0000 0101 0010 0001 1101 1001 0101 0000 in binary, or 0x0521D950.

9.3.3 Encoding integer data manipulation instructions

Data manipulation instructions are encoded using two formats: one for instructions that use two source registers and another for instructions that use a source register and a small integer constant. The formats used for integer data manipulation instructions are shown in Figure 9.3

In the first format the 32-bit instruction is divided into seven fields. The first field (reading from the left) holds the 2-bit value 10, while the fifth field (bit 13) holds the 1-bit value 0.

A. Instructions of the form: op rs_1, rs_2, rd

31 30	29 25	24 19	18 14	13	12 5	4 0
10	rd	op_3	rs_1	0	unused(zero)	rs_2

B. Instructions of the form: op rs_1, $siconst_{13}$, rd

31 30	29 25	24 19	18 14	13	12 0
10	rd	op_3	rs_1	1	$siconst_{13}$

Figure 9.3 Instruction formats for data manipulation instructions

These bits are the same for all data manipulation instructions that use two source registers. The sixth field (bits 5 through 12) is unused-the bits in this field must be zero. The remaining fields, *rd*, op_3, rs_1, and rs_2, hold encodings for the destination register, the operation, and the two source registers, respectively

In the second format the 32-bit instruction is divided into six fields. As in the previous format, the first field holds the first field holds the 2-bit value 01. However, unlike the previous format, the fifth field holds the 1-bit value 1. The remaining fields, *rd*, op_3, rs_1, and $siconst_{13}$, hold encodings for the destination register, the operation, the source register, and the constant value, respectively. When this format is used, the integer constant is encoded using the 13-bit 2's complement representation and stored in the $siconst_{13}$ field of the instruction.

Recall that a SPARC assembly language instruction begins with the name of the operation, followed by the two source operands, followed by the destination operand. In considering the translation from an assembly language instruction into machine language, there are a few points to keep in mind:

- The operation is encoded in the op_3 field.
- The first source operand must be a register and it is encoded in the rs_1 field.
- The second source operand can be a register or a constant value. If it is a register, it is encoded in the rs_2 field; otherwise, it is encoded in the $siconst_{13}$ field.
- The destination register is encoded in the *rd* field.

Table 9.2 summarizes the operation encodings for the data manipulation operations that we have covered in the previous labs. When an instruction using one of these operations is encoded, the operator encoding is placed in the op_3 field of the machine instruction.

Example 9.4 *Hand Assemble the following SPARC instructions.*

```
sub     %r27, %r16, %r26
```

Because this instruction uses two source registers, it is encoded using the first format shown in Figure 9.3. As such, we must determine the values for the op_3, *rd*, rs_1, and rs_2 fields. The following table summarizes these encodings:

Field	Symbolic value	Encoded value
rd	%r27	11011
op_3	sub	000100
rs_1	%r16	10000
rs_2	%r26	11010

These encodings lead to the following machine instruction:

31 30	29 25	24 19	18 14	13	12 5	4 0
10	11011	000100	10000	0	00000000	11010

That is, 1011 0110 0010 0100 0000 0000 0001 1010 in binary, or 0xB624001A.

Table 9.2 Operation encodings for the data manipulation operations

Operation	op_3	Operation	op_3
add	000000	addcc	010000
and	000001	andcc	010001
andn	000101	andncc	010101
or	000010	orcc	010010
orn	000110	orncc	010110
udiv	001110	udivcc	011110
umul	001010	umulcc	011010
smul	001011	smulcc	011011
sdiv	001111	sdivcc	011111
sub	000100	subcc	010100
xor	000011	xorcc	010011
xnor	000111	xnorcc	010111
sll	100101		
rsl	100101		
rsa	100111		

Example 9.5 *Hand Assemble the following SPARC instructions.*

```
smulcc  %r19, -23, %r29
```

Because this instruction uses one source register and a signed integer constant, it is encoded using the second format shown in Figure 9.3. As such, we must determine the values for the op_3, rd, rs_1, and $siconst_{13}$ fields. The following table summarizes these encodings:

Field	Symbolic value	Encoded value
rd	%r19	10011
op_3	smulcc	011011
rs_1	%r29	11101
$siconst_{13}$	−23	1111 1111 0100 1

These encodings lead to the following machine instruction:

31 30	29 25	24 19	18 14	13	12 0
10	10011	011011	11101	1	1111 1111 0100 1

That is, 1010 0110 1101 1111 0111 1111 1110 1001 in binary, or 0xA6DF7FE9.

9.3.4 Encoding conditional branching instructions

The machine language format for the conditional branching operations on the SPARC is shown in Figure 9.4. This format divides the machine instruction into five fields. The first and fourth fields hold the fixed values 102 and 0102. The remaining fields, *a*, *cond*, and $disp_{22}$, hold the encoded values for the annul bit, the branching condition, and program counter displacement.

31 30	29	28 25	24 22	21 0
00	a	cond	010	$disp_{22}$

Figure 9.4 Instruction format for conditional branch instructions

The *a* field of a machine instruction is set (i.e., 1) for instructions that use the annul suffix (",a"). This field is clear (i.e, 0) for conditional branching instructions that do not nullify the results of the next instruction. The *cond* field of a machine instruction encodes the condition under which the branch is taken. Table 9.3 summarizes the operation encodings for the branching operations supported by the SPARC.

Table 9.3 Operation encodings for the conditional branching operations

Operation	cond	Operation	cond
ba	1000	bn	0000
bne (bnz)	1001	be (bz)	0001
bg	1010	ble	0010
bge	1011	bl	0011
bgu	1100	bleu	0100
bcc (bgeu)	1101	bcs (blu)	0101
bpos	1110	bneg	0110
bvc	1111	bvs	0111

To complete the encoding of an assembly language instruction that uses conditional branching, you need to determine the value of the $disp_{22}$ field. We address this issue by considering how a processor uses this value. When the processor determines that the branching condition is satisfied, it multiplies the value in the $disp_{22}$ field by 4 and adds it to the program counter (PC). To be more precise, the processor sign extends the 22-bit value stored in the $disp_{22}$ field to 30 bits and concatenates two zeros to construct a 32-bit which which it adds to the PC. In effect, the $disp_{22}$ field holds the distance from the target to the destination measured in instructions.

Example 9.6 *Hand Assemble the branch instruction in the following SPARC code fragment.*

```
        cmp     %r2, 8
        bne     l1
        nop
        inc     %r3
l1:
```

In this case, the target is 3 instructions from the branch instruction, so the $disp_{22}$ field will be the 22-bit binary encoding of 3.

Field	Symbolic value	Encoded value
a		0
cond	bne	1001
disp$_2$2	l1	0000 0000 0000 0000 0000 11

These encodings lead to the following machine instruction:

31 30	29	28 25	24 22	21 0
00	0	1001	010	0000 0000 0000 0000 0000 11

That is, 0001 0010 1000 0000 0000 0000 0000 0011 in binary, or 0x12800002.

Example 9.7 *Hand Assemble the branch instruction in the following SPARC code fragment.*

```
top:    add     %r2, %r3, %r2
        deccc   %r4
        bne     top
```

In this case, the target is 2 instructions (back) from the branch instruction, so the $disp_{22}$ field will be the 22-bit binary encoding of -2.

Field	Symbolic value	Encoded value
a		0
cond	bne	1001
disp$_2$2	l1	1111 1111 1111 1111 1111 10

These encodings lead to the following machine instruction:

31 30	29	28 25	24 22	21 0
00	0	1001	010	1111 1111 1111 1111 1111 10

That is, 0001 0010 1011 1111 1111 1111 1111 1110 in binary, or 0x12BFFFFE.

9.3.5 Synthetic Instructions

In most cases, an assembly language instruction is simply a symbolic representation of a machine language instruction. The SPARC architecture also defines a number of assembly language instructions that do not correspond directly to SPARC machine language instructions. These are called synthetic instructions. The assembler translates synthetic instructions one or more machine language instructions. Using synthetic instructions can frequently make your programs easier to read. Table 9.4 summarizes the translation provided by the assembler for most of the synthetic instructions on the SPARC.

Most of the translations shown in Table 9.4 are straightforward. However, the implementation of the *set* instruction merits further discussion. The assembler will always try to use one of the first two translations if it can. That is, if the constant value can be represented in 13 bits, the assembler will select the first translation. If the least significant 10 bits of the constant value are 0, it will used the second translation. Otherwise, the assembler will use the third translation. Note, if the constant value is relocatable, the assembler will always select the third translation.

9.3.6 The read and write instructions

The Y register, introduced in Lab 4 is one of the SPARC state registers. As shown in Table 9.4, when you use a state register as the destination in a mov instruction, it is translated to a wr (write) instruction. Similarly, when you use a state register as the source register in a mov instruction it is translated to a rd (read) instruction.

Write instructions are encoded using the formats shown in Figure 9.3. When the destination register is the Y register, the *rd* field is set to the 5-bit value 00000 and the op_3 field is set to the 6-bit value 110000.

Read instructions are encoded using the second format shown in Figure 9.3. When the source register is the Y register, the op_3 field is set to the 6-bit value 101000 and the rs_1 field is set to the 5-bit value 00000.

9.3.7 Relocatable expressions

In this lab, we have limited our discussion to the translation of instructions that use absolute expressions. We will consider the translation of relocatable expressions when we consider linking and loading in Lab 15.

9.3.8 Decoding SPARC instructions

We conclude our discussion of instruction formats by considering instruction decoding. That is, the process by which a SPARC processor determines the instruction it is executing.

The SPARC uses a distributed opcode. The two most significant bits in an instruction represent the primary opcode. If the primary opcode is 00, bits 22–24 of the instruction

Table 9.4 The synthetic instructions

Synthetic instruction		Implementation	
bclr	*rs*, *rd*	andn	*rd*, *rs*, *rd*
bclr	*rs*, $siconst_{13}$	andn	*rs*, $siconst_{13}$, *rd*
bset	*rs*, *rd*	or	*rd*, *rs*, *rd*
bset	$siconst_{13}$, *rd*	or	*rd*, $siconst_{13}$, *rd*
btst	rs_1, rs_2	andcc	rs_1, rs_2, %g0
btst	*rs*, $siconst_{13}$	andcc	*rs*, $siconst_{13}$, %g0
btog	*rs*, *rd*	xor	*rd*, *rs*, *rd*
btog	*rs*, $siconst_{13}$	xor	*rs*, $siconst_{13}$, *rd*
clr	*rd*	or	%g0, %g0, *rd*
clrb	[*address*]	stb	%g0, [*address*]
clrh	[*address*]	sth	%g0, [*address*]
clr	[*address*]	st	%g0, [*address*]
cmp	rs_1, rs_2	subcc	rs_1, rs_2, %g0
cmp	*rs*, $siconst_{13}$	subcc	*rs*, $siconst_{13}$, %g0
dec	*rd*	sub	*rd*, 1, *rd*
dec	$siconst_{13}$, *rd*	sub	*rd*, $siconst_{13}$, *rd*
deccc	*rd*	subcc	*rd*, 1, *rd*
deccc	$siconst_{13}$, *rd*	subcc	*rd*, $siconst_{13}$, *rd*
inc	*rd*	add	*rd*, 1, *rd*
inc	$siconst_{13}$, *rd*	add	*rd*, $siconst_{13}$, *rd*
inccc	*rd*	addcc	*rd*, 1, *rd*
inccc	$siconst_{13}$, *rd*	addcc	*rd*, $siconst_{13}$, *rd*
mov	*rs*, *rd*	or	%g0, *rs*, *rd*
mov	$siconst_{13}$, *rd*	or	%g0, $siconst_{13}$, *rd*
mov	*statereg*, *rd*	rd	*statereg*, *rd*
mov	*rs*, *statereg*	wr	%g0, *rs*, *statereg*
mov	$siconst_{13}$, *statereg*	wr	%g0, $siconst_{13}$, *statereg*
neg	*rs*, *rd*	sub	%g0, *rs*, *rd*
neg	*rd*	sub	%g0, *rd*, *rd*
not	*rd*	xnor	*rd*, %g0, *rd*
not	*rs*, *rd*	xnor	*rs*, %g0, *rd*
set	*iconst*, *rd*	or	%g0, *iconst*, *rd*
			—*or*—
		sethi	%hi(*iconst*), *rd*
			—*or*—
		sethi	%hi(*iconst*), *rd*
		or	*rd*, %lo(*iconst*), *rd*
tst	*rs*	orcc	%g0, *rs*, %g0

provide the secondary opcode. If the primary opcode is 01, the instruction is a call instruction and the remaining bits (bits 0–29) are a displacement for the program counter (we will discuss the call instruction at greater length in Lab 10). Otherwise, if the primary opcode is either 10 or 11, bits 19–24 of the instruction provide the secondary opcode. Figure 9.5 illustrates the positions of the secondary opcodes based on the primary opcode.

Once you have determined the primary and secondary opcodes, you'll be able to to determined the instruction and, knowing the instruction, decode the remaining fields of the instruction. If the primary opcode is 01, the instruction is a call instruction and you can easily complete the decoding of the instruction.

If the primary opcode is 00, the instruction is an unimplemented instruction, a condi-

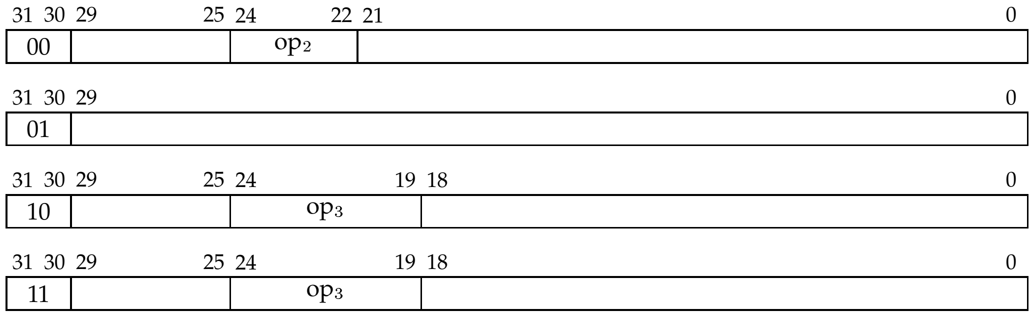


Figure 9.5 The primary opcode ina SPARC instruction

tional branch instruction, or a sethi instruction. Table 9.5 summarizes how the 3-bit value in op_2 is used to identify the instruction.

Table 9.5 Decoding the op_2 field

Value	Instruction
000	The unimplemented instruction
001	*illegal*
010	Conditional branch—integer unit
011	*illegal*
100	SETHI
101	*illegal*
110	Conditional branch—floating point unit
111	Conditional branch—coprocessor

The data manipulation instructions are encoded with a primary opcode of 10. Table 9.6 shows how the 6-bit value in the op_3 field is used to determine the instruction when the primary opcode is 10.

Table 9.6 Decoding the op_3 field when the primary opcode is 10

	000xxx	001xxx	010xxx	011xxx	100xxx	101xxx	110xxx	111xxx
xxx000	add	addx	addcc	addxcc	taddcc	rd	wr	jmpl
xxx001	and	—	andcc	—	tsubcc	rd	wr	rett
xxx010	or	umul	orcc	umulcc	taddcctv	rd	wr	trap
xxx011	xor	smul	xorcc	smulcc	tsubcctv	rd	wr	flush
xxx100	sub	subx	subcc	subxcc	mulscc	—	FPU op	save
xxx101	andn	—	andncc	—	sll	—	FPU op	restore
xxx110	orn	udiv	orncc	udivcc	srl	—	CP op	—
xxx111	xnor	sdiv	xnorcc	sdivcc	sra	—	CP op	—

Instructions that access memory are encoded with a primary opcode of 11. Table 9.7 shows how the 6-bit value in the op_3 field is used to determine the instruction when the primary opcode is 11.

When you decode an instrcution that has a primary opcode of 10 or 11, you will need to examine bit 13 to determine whether bits 0–12 of the instruction hold an immediate value

Table 9.7 Decoding the op_3 field when the primary opcode is 11

	000xxx	001xxx	010xxx	011xxx	100xxx	101xxx	110xxx	111xxx
xxx000	ld	—	lda	—	ldf	—	ldc	—
xxx001	ldub	ldsb	lduba	ldsba	ldfsr	—	ldcsr	—
xxx010	lduh	ldsh	lduha	ldsha	—	—	—	—
xxx011	ldd	—	ldda	—	lddf	—	lddc	—
xxx100	st	—	sta	—	stf	—	stc	—
xxx101	stb	ldstub	stba	ldstuba	stfsr	—	stcsr	—
xxx110	sth	—	stha	—	stdfq	—	scdfq	—
xxx111	std	swap	stda	swapa	stdf	—	scdf	—

or a register. If bit 13 is 1, bits 0–12 hold an immediate value.

Example 9.8 *Give an instruction that will assemble to the value 0x09012345.*

In binary, this instruction is 00 00100 100 000100.... That is, the primary opcode is 00 and op_2 is 100. From Table 9.5, this is a sethi instruction. Using the sethi format to partition the bits yields:

31 30	29 25	24 22	21 0
00	rd = 00100	100	$iconst_{22}$ = 01 0010 0011 0100 0101

Thus, the destination register is %r4, and the integer constant is 0x12345. The following instruction will be assembled as 0x09012345.

```
sethi   %hi(0x12345<<10), %r4
```

Example 9.9 *Give an instruction that will assemble to the value 0x10800006.*

In binary, this instruction is 00 01000 010 000000.... That is, the primary opcode is 00 and op_2 is 010. From Table 9.5, this is a conditional branch instruction. Using the conditional branch format to partition the bits yields:

31 30	29	28 25	24 22	21 0
00	0	cd=1000	010	disp=0000 0000 0000 0000 0001 10

Thus, the operator is "ba" and the displacement is +6 words. The following instruction will be assembled as 0x10800006.

```
ba      .+(6*4)
```

(When you use isem-as, '.' is the address of the current instruction.

Example 9.10 *Give an instruction that will assemble to the value 0x8601600E*

In binary, the instruction is 10 00011 000000 0001.... That is, the primary opcode is 10 and op_3 is 000000. From Table 9.6, this is an add instruction. Because bit 13 is 1, we use the second format in Figure 9.3 to decode this instruction.

31 30	29 25	24 19	18 14	13	12 0
10	rd=00011	000000	rs_1=00101	1	$siconst_{13}$=0 0000 0000 1110

Thus, the destination is %r3, the source register is %r5, and the constant is 0xE. The following instruction will be assembled as 0x8601600E.

```
add     %r5, 14, %r3
```

9.4 Summary

9.5 Review Questions

9.6 Exercises

Laboratory 10
Leaf Procedures on the SPARC

10.1 Goal

To introduce the calling conventions associated with leaf procedures on the SPARC.

10.2 Objectives

After completing this lab, you will be able to write assembly language programs that:

- Use leaf procedures,
- Use the call and link operation, and
- Use the (synthetic) return from leaf operation.

10.3 Discussion

This lab is the first of three labs that cover procedure calling conventions on the SPARC. In this lab we consider the conventions associated with leaf procedures: procedures that do not make calls to other procedures. In Lab 11 we consider the use of register windows on the SPARC. In Lab 12 we complete our coverage of procedure calling conventions by considering the standard calling conventions used by compilers.

10.3.1 Register usage

The SPARC Architecture Manual describes a class of procedures called "optimized leaf procedures." As we have noted, a leaf procedure is a procedure that does not call any other procedures. Optimized refers to restrictions placed on the procedure's use of the registers.

Table 10.1 summarizes register uses for optimized leaf procedures. This table specifies which registers can be modified by a leaf procedure. Registers %r8–%r13 are used for parameters passed to the leaf procedure. The first parameter (i.e., the leftmost parameter) should be placed in %r8, the next in %r9, and so forth. Note that %r8 is used for the first parameter and the return value.

Beyond the registers used to pass parameters to the leaf procedure, there are a few other conventions in Table 10.1 that are worth noting. Register %r1 can always be used as a temporary register and the caller cannot assume that it will retain its value across a call to a leaf procedure. We have frequently used this register in the sethi instructions used with the load and store instructions, and you can continue to do this in leaf procedures. The stack pointer is stored in %r14. We will discuss the use of the stack pointer when we consider stack based calling conventions in Lab 12. For the moment, note that a leaf procedure should not alter the value stored in register %r14. Finally, note that the return address

Table 10.1 Register usage for optimized leaf procedures

Register(s)	Use	Changed by leaf procedure
%r0	zero	No
%r1	temporary	Yes
%r2–%r7	caller's variables	No
%r8	return value	Yes
%r8–%r13	parameters	Yes
%r14	stack pointer	No
%r15	return address	Yes
%r16–%r31	caller's variable	No

(actually, the address of the call instruction used to call the leaf procedure) is stored in register %r15. A leaf procedure can alter this register; however, it will be difficult to return to the point of the call if you alter the value in%r15.

An optimized leaf procedure should only alter the values stored in registers %r1 and %r8–%r13. If the leaf procedure requires more local storage than these registers provide, or if the parameters do not fit in these registers, the leaf procedure cannot be implemented as an "optimized" leaf procedure. We will discuss the techniques use to implement other types of procedures in the next two labs.

10.3.2 Calling sequence

In assembly language, a procedure is a block of instructions. The first instruction in this block is labeled by the name of the procedure.

Table 10.2 summarizes the operations used to call and return from optimized leaf procedures. Like the branching operations introduced in Lab 3, these operations have a branch delay slot. The call and link operation saves the current value of the PC in %r15, updates the PC, and sets the nPC to the address specified in the call. The retl operation updates the PC and sets to nPC to the contents of %r15 plus eight. The "plus eight" is needed to skip over the call instruction and the instruction in the delay slot of the call. Examples 10.1 and 10.2 illustrate the use of these operations.

Table 10.2 The call and retl operations

Operation	Assembler syntax	Operation implemented
call and link	call label	%r15 = PC PC = nPC nPC = label
return from leaf	retl	PC = nPC nPC = %r15 + 8

Example 10.1 *Write SPARC procedure that prints a NULL terminated string. The address of the string to print will be passed as the first parameter (i.e., in %r8).*

```
        .text
! pr_str - print a null terminated string
!
! Parameters:   %r8 - pointer to string (initially)
!
```

```
! Temporaries:  %r8 - the character to be printed
!               %r9 - pointer to string
!

pr_str: mov     %r8, %r9         ! we need %r8 for the "ta 1"
pr_lp:  ldub    [%r9], %r8       ! load character
        cmp     %r8, 0           ! check for null
        be      pr_dn
        nop
        ta      1                ! print character
        ba      pr_lp
        inc     %r9              ! increment the pointer

pr_dn:  retl                     ! return from leaf procedure
        nop                      !   (branch delay)
```

Example 10.2 *Write a SPARC assembly language fragment that calls the procedure presented in Example 10.1.*

```
        .data
str:    .asciz  "Hello, World!\n"

        .text
main:   set     str, %r8         ! setup the first argument
        call    pr_str           ! call print string
        nop                      !   (branch delay)
end:    ta      0                ! exit gracefully
```

10.3.3 Instruction Encodings

We have introduced two new instructions in this lab: *call* and *retl*. Figure 10.1 illustrates the format used to encode call instructions. This instruction encoding uses two fields. The first field holds the 2-bit value 01, the second holds a 30-bit displacement for the program counter. This field is encoded in the same fashion as the displacement field for the conditional branch instructions (see Lab 9 for more details).

31 30	29 … 0
01	$disp_{30}$

Figure 10.1 Instruction format for call instructions

Example 10.3 *Show how the call instruction in the following SPARC assembly code fragment is encoded.*

```
        .text
main:   set     str, %r8         ! setup the first argument
        call    pr_string        ! call print string
        nop                      !   (branch delay)
        ta      0                ! exit gracefully

pr_string:
        mov     %r8, %r9         ! we need %r8 for the "ta 1"
```

In this case, the target is 3 instructions from the call instruction, so the $disp_{30}$ field is set to the 30-bit binary encoding of 3.
This leads to the following machine instruction:

31 30	29 ... 0
01	0000 0000 0000 0000 0000 0000 0000 11

That is, 0100 0000 0000 0000 0000 0000 0000 0011 in binary, or 0x40000003.

The *retl* instruction is actually a synthetic instruction that is translated to a *jmpl* (jump and link) instruction. The *jmpl* instruction has two operands: an address, and a destination register. The address is similar to the addresses used in the load and store instructions; however, the brackets surrounding the address in the load and store instructions are omitted in the jmpl instruction. When a SPARC processor executes a jmpl instruction, it saves the address of the jmpl instruction in the destination register and sets the next program counter to the address specified in the instruction. Figure 10.2 illustrates the formats used to encode jmpl instructions.

A. Instructions of the form: jmpl rs_1+rs_2, rd

31 30	29 25	24 19	18 14	13	12 5	4 0
10	rd	111000	rs_1	0	unused(zero)	rs_2

B. Instructions of the form: jmpl rs_1+$siconst_{13}$, rd

31 30	29 25	24 19	18 14	13	12 0
10	rd	111000	rs_1	1	$siconst_{13}$

Figure 10.2 Instruction formats for jmpl instructions

The retl instruction is translated to a jmpl instruction with the address set to %r15+8 and the destination register set to %r0. That is, the target of the branch is instruction after the branch delay slot of the call instruction (remember, call instruction saved its address in %r15), and the address of the retl instruction is discarded (saved in %r0).

Example 10.4 *Show how a retl instruction is encoded.*

A retl instruction is translated to the instruction jmpl %r15+8, %r0. This instruction is encoded using the second format shown in Figure 10.2. Figure 9.3. As such, we must determine the values for the *rd*, rs_1, and $siconst_{13}$ fields. The following table summarizes these encodings:

Field	Symbolic value	Encoded value
rd	%r0	00000
rs_1	%r15	01111
$siconst_{13}$	8	0000 0000 0100 0

These encodings lead to the following machine instruction:

31 30	29 25	24 19	18 14	13	12 0
10	00000	111000	01111	1	0000 0000 0100 0

That is, 1000 0001 1100 0011 1110 0000 0000 1000 in binary, or 0x81C3E008.

10.4 Summary

A leaf procedure is a procedure that never calls any other procedure. In this lab we have introduced the SPARC instructions used to write leaf procedures: call and retl. In the next two labs, we will examine more general procedure calling conventions.

10.5 Review Questions

1. xxx

10.6 Exercises

1. Write a SPARC assembly language program consisting of of a main program and a procedure, "pr_octal", that prints an unsigned integer in octal notation.
2. Write a SPARC assembly language program consisting of of a main program and a procedure, "pr_hex", that prints an unsigned integer in hexadecimal notation.
3. Write a SPARC procedure, called "strcmp", that compares two strings. Your procedure should accept two parameters, s1 and s2, both pointers to NULL terminated strings. Your procedure should return an integer based on the comparison. In particular,
   - if (s1 < s1) return -1;
   - if (s1 > s2) return 1;
   - if (s1 == s2) return 0;

   You should also provide a small driver to test your procedure.
4. Write a SPARC procedure, called "strchr", that returns a pointer to the first occurrence of a character within a string. The first parameter should be a pointer to a string. The second parameter should be the character search for. If the character is not present in the string, the procedure should return NULL (i.e., 0).
   You should also provide a small driver to test your procedure.
5. Write a procedure to draw a black pixel at an arbitrary (x, y) location on the GX device. The C declaration for the procedure would be:
   void draw_pixel(int x, int y)

Laboratory 11
Register Windows

11.1 Goal

To introduce register windows.

11.2 Objectives

After completing this lab, you will be able to use:

- Register windows,
- The save and restore operations,
- The (synthetic) return operation.

11.3 Discussion

In this lab we introduce a more general procedure calling mechanism that uses register windows. We introduce the save and restore instructions and another synthetic instruction, ret, for returning from procedures.

11.3.1 Register Windows

To this point, we have used the %r names for the integer registers. From this point on, we will use the alternate names for these registers. The alternate names are shown in Table 11.1.

Table 11.1 Names for the integer registers

Integer registers	Alternate names	Group name
%r0–%r7	%g0–%g7	Global registers
%r8–%r15	%o0–%o7	Output registers
%r16–%r23	%l0–%l7	Local registers
%r24–%r31	%i0–%i7	Input registers

The alternate names reflect the uses of the registers when procedures use register windows. The global registers (%g0–%g7) are shared by all procedures. The output registers are used for parameters when calling another procedure. That is, the output registers are outputs from the caller to the called procedure. The local registers (%l0–%l7) are used to store local values used by a procedure. The input registers (%i0–%i7) are used for the parameters passed into the procedure. That is, the input registers are inputs passed from the caller to the called procedure.

When you consider the relationship between the output and input registers, the trick is to make the caller's output registers the same as the called procedure's input registers. On the SPARC, this is done using *overlapping register windows*.

All procedures share the global registers (%g0–%g7). The remaining registers, %o0–%o7, %l0–%l7, and %i0–%i7, are called a *register window*. When a procedure starts its execution, it allocates a set of 16 registers (using the save instruction as described in the following section). The new register set provides the procedure with its own output and local registers (%o0–%o7 and %l0–%l7). The procedure's input registers (%i0–%i7) are overlapped with the caller's output registers. Figure 11.1 illustrates the overlapping of register windows between the caller and the callee.

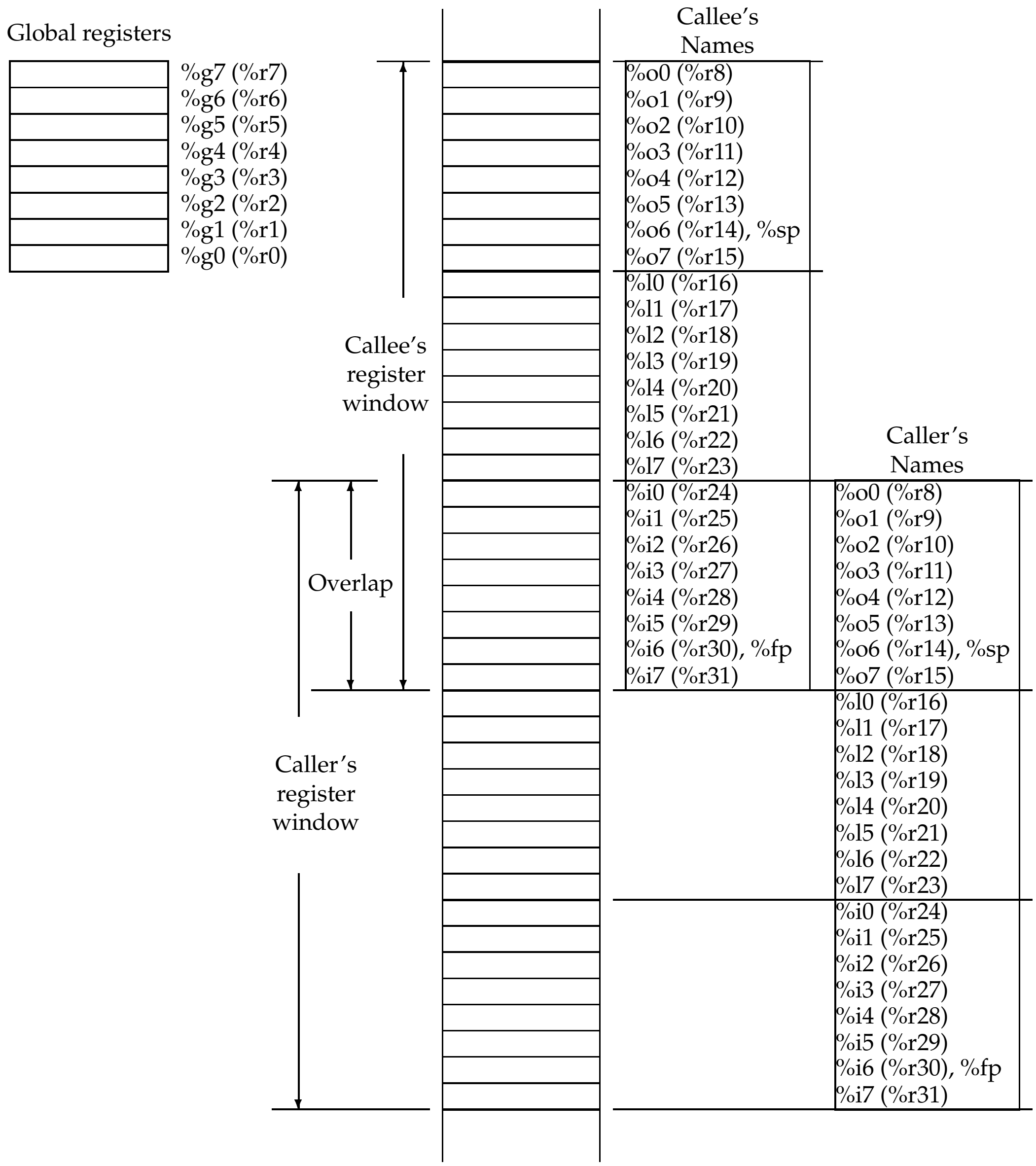


Figure 11.1 Overlapping register windows

In addition to the register names we have discussed, Figure 11.1 introduces two new names: %sp and %fp. The first of these, %sp, denotes the stack pointer. In an assembly language program, %sp is simply another name for %o6. Similarly, %fp denotes the frame pointer and is simply another name for %i6. We will discuss the special uses of these registers (and hence the additional names) in the next lab when we consider stack frame organization.

An implementation of the SPARC integer unit may have between 40 and 520 integer registers. Every SPARC has 8 global registers, plus a circular stack of 2 to 32 register sets. Each register set has 16 registers. The number of registers sets is implementation dependent. Number of register sets provided a particular implementation of the SPARC architecture has been given the name NWINDOWS. ISEM provides 32 register sets, the maximum number supported by the SPARC Architecture. Most hardware implementations provide 7 or 8 register sets.

11.3.2 Save and Restore

The current window in the integer registers is given by the current window pointer (CWP). The CWP is stored in the lower five bits of the processor status register (PSR). The save and restore instructions let the programmer to manipulate the CWP.

Table 11.2 summarizes the SPARC save and restore operations. SPARC assemblers provide two instruction formats for save instructions and three formats for restore instructions. Both formats for the save instruction and the first two formats for the restore instruction use three explicit operands—two source operands and a destination operand. In the first format, both source operands are in registers. In the second format, one source operand is in a register, the other is specified using a small constant value. This constant may be positive or negative; however its 2's complement representation must fit in 13 bits. Its important to note, for both the save and restore operations, that the destination register is in the register window *after* the CWP has been modified. The restore instruction also has a format that doesn't have any operands.

Table 11.2 Saving and restoring register windows

Operation	Syntax	Operation implemented
save caller's register window	save rs_1, rs_2, rd	res = reg[rs_1]+reg[rs_2] CWP = (CWP−1) % NWINDOWS reg[rd] = res
	save rs_1, $siconst_{13}$, rd	res = reg[rs_1]+$siconst_{13}$ CWP = (CWP−1) % NWINDOWS reg[rd] = res
restore caller's register window	restore rs_1, rs_2, rd	res = reg[sr_1]+reg[sr_2] CWP = (CWP+1) % NWINDOWS reg[rd] = res
	restore rs, $siconst_{13}$, rd	res = reg[rs]+$siconst_{13}$ CWP = (CWP+1) % NWINDOWS reg[rd] = res
	restore	CWP = (CWP+1) % NWINDOWS

11.3.3 Stack management

The operands of the save instruction are commonly used to allocate space for a stack frame. We'll discuss the stack and stack frames at greater length in the next lab. However, to the save and restore instructions correctly, you must allocate and maintain a runtime stack.

The runtime stack grows from higher to lower addresses. The stack pointer (%r14, %o6) should be aligned on an 8 byte address at all times. For now, we'll use the operands of the save instruction to subtract 96 from the stack pointer (%sp). The reason for this will be clearer in the next lab.

In most cases, you will simply use the restore instruction with no arguments to restore the caller's register window. However, you can occasionally use the other versions of this instruction to return a value from a procedure.

11.3.4 Procedure calling conventions

From the caller's perspective using register windows does not change anything about how it interacts with the called procedure. The caller still puts the outgoing parameters in registers %o0–%o5 (%r8–%r13) and expect the result in %o0 (%r8). Moreover, the caller may assume that registers %g0, %g2–%g7, %o6 (%sp), %l0–%l7, and %i0–%i7 will not be altered by the called procedure. Finally, the caller uses the call instruction introduced in the previous lab to transfer control to the called procedure.

From the perspective of the called procedure, things have changed quite a bit. As soon as control is transferred to the procedure, it needs to save to the caller's register window. In the body of the procedure, you can modify %g1, %i0–%i5, %l0–%l7, and %o0–%o6. You could also modify %i7; however, %i7 holds the return address, so it's not a good idea to change it. Finally, just before returning control to the caller, the called procedure needs to restore the caller's register window.

The instruction used to restore the caller's register window is usually put in the branch delay slot of the instruction used to return control to the caller. Because the caller's register window has not been restored when the called procedure issues the return instruction, the return instruction needs to use %i7 in calculating the return address instead of %o7 (%r15). SPARC assemblers provide the *ret* instruction for this purpose. Table 11.3 summarizes the call and return instructions.

Table 11.3 The SPARC call and ret operations

Operation	Syntax	Operation implemented
call and link	call label	%o7 = PC
		PC = nPC
		nPC = label
return from procedure	ret	PC = nPC
		nPC = %i7 + 8

Example 11.1 *Write a SPARC assembly language procedure, pr_str, that will print a NULL terminated string. Your procedure should take a single argument, the address of the string to print. In writing this procedure, you should assume that the procedure pr_ch is available for printing a character.*

```
        .text
! pr_str - print a null terminated string
!
! Temporaries:  %i0 - pointer to string
```

```
!               %o0 - character to be printed
!
pr_str: save    %sp, -96, %sp   ! PROLOGUE - save the current window and
                                !   allocate the minimum stack frame

pr_lp:  ldub    [%i0], %o0      ! load character
        cmp     %o0, 0          ! check for null
        be      pr_dn
        nop
        call    pr_char         ! print character
        nop
        ba      pr_lp
        inc     %i0             ! increment the pointer (branch delay)

pr_dn:  ret                     ! EPILOGUE return from procedure and
        restore                 !   restore old window; no return value
```

Example 11.2 *Write a "main" SPARC assembly language fragment that allocates space for the stack and calls the pr_str procedure in the previous example.*

```
        .data
str:    .asciz  "Hello, World!\n"

        .align  8
stack_top:
        . = . + 2048
stack_bot:

        .text
start:
        set     stack_bot-96, %sp ! initialize the stack and allocate
                                  !  the minimum stack frame
        set     str, %o0          ! initialize pointer to str
        call    pr_str            ! call print string
        nop                       !   (branch delay)
end:    ta      0
```

Example 11.3 *Write a procedure that recursively calculates the Nth Fibonacci number. You may assume that N is non-negative and will be small enough that register overflow will not occur.*

```
! fib - calculate the Nth Fibonacci number
!
!       fib(N) = fib(N-1) + fib(N-2)
!       fib(0) = fib(1) = 1

fib:    save    %sp, -96, %sp   ! PROLOGUE

        cmp     %i0, 1
        bg      fib_call        ! call recursively
        nop

        ret                     ! EPILOGUE
        restore %g0, 1, %o0     ! return 1
```

```
fib_call:
        call    fib             ! call with N-1
        sub     %i0, 1, %o0     !  (branch delay)
        mov     %o0, %l0        ! %l0 = fib(N-1)

        call    fib             ! call with N-2
        sub     %i0, 2, %o0     !  (branch delay)

        ret                     ! EPILOGUE
        restore %l0, %o0, %o0   ! return fib(N-1) + fib(N-2)
```

11.3.5 Exceptions

Both the save and restore operations can generate exceptions (or traps). Before the CWP is modified, the bit in the WIM cooresponding to the new value for the CWP is tested. If the bit in the WIM is 1, an exception is generated. For a save instruction, this causes a window overflow trap. For a restore instruction, this causes a window underflow trap.

These traps are normally handled by the operating system and are transparent to the application programmer. In *tkisem* these traps are handled by the *rom* code. We will discuss the code used to handle these traps in Lab 17.

11.3.6 Instruction encoding

We have introduced three new instructions in this lab: save, restore, and ret. The save and restore instructions are encoded as data manipulation instructions (the instruction formats are shown in Figure 9.3). The restore instruction with no operands is actually a synthetic instruction in which all of the operands are %g0. Table 11.4 summarizes the encodings of the op_3 field for the save and restore instructions.

Table 11.4 Encoding op_3 in save and restore instructions

Instruction	op_3
save	111100
restore	111101

Like the *retl* instruction, the *ret* instruction is a synthetic instruction, based on the *jmpl* instruction. The ret instruction is translated to jmpl %i7+8, %g0.

11.4 Summary

This lab presents a more general mechanism for procedures on the SPARC. Register windows provide easy access to a large collection of registers and can reduce the need to save registers in memory. While this mechanism has many advantages there are several disadvantages to keep in mind. The mechanism only provides six registers for procedure parameters. If you write a procedure with more than six parameters, you will need to to use the stack for any parameters beyond six. Secondly, most implementations only have 7 or 8 register sets. So, if your call sequence gets deeper than NWINDOWS (as it probably will in most recursive procedures), you are again forced to use the stack.

11.5 Review Questions

11.6 Exercises

1. Write a procedure which will draw a bitmap at an arbitrary (x, y) location on the GX device. The bitmap is described by an array of chars, a width, and a height (i.e., the X Windows bitmap format) The C declaration for the procedure would be:
   void draw_bitmap(char* bits, int w, int h, int x, int y)
   The draw_bitmap procedure should make use of the draw_pixel procedure (from the previous lab).
2. Write a procedure that "tiles" a bitmap to the GX display. The procedure should call draw_bitmap for the first tile. But, for all of the subsequent tiles, you should use the GX_BLIT operation. The C declaration for the procedure would be:
   void tile_bitmap(char* bits, int w, int h)
   Don't forget to clip the border tiles appropriately.

Laboratory 12
Standard Calling Conventions

12.1 Goal

To cover the standard procedure calling conventions for the SPARC.

12.2 Objectives

After completing this lab, you will be able to write assembly language procedures that:

- Follow the standard calling conventions for the SPARC,
- Call C functions, and
- Can be called from C.

12.3 Discussion

In most cases, you will not want to write entire programs in assembly language. Instead, you will want to write most of the program in a high-level language (like C) and only write a few procedures in assembly language—the procedures that cannot be easily optimized in the high-level language or that need to take advantage of special features provided by the machine.

In this lab, we complete our presentation of the SPARC application binary interface (ABI). The SPARC ABI is a set of conventions that are expected to be folowed by all compilers and assembly language programmers. These conventions cover the uses of registers and the structure of the stack frame. If you follow the conventions specified by the SPARC ABI in your assembly language procedures, it will be possible to call your proceudres from procedures written in high-level languages. You will also be able to call procedures written in high-level languages from your assembly language procedures.

In Lab 10 we covered the portion of the SPARC ABI that deals with optimized leaf procedures. In Lab 11 we covered the conventions related to register usage for procedures that are not implementd as optimized leaf procedures. In this lab we cover the conventions related to the allocation and structure of stack frames. Throughout this lab we will assume that we are not implementing an optimized leaf procedure.

12.3.1 The allocation of stack frames

The stack pointer is stored in register %o6. In assembly language, this register can be referenced using the alias %sp. Due to the overlap of register windows, the stack pointer for the calling procedure is always available in register %i6. In SPARC terminology, the previous stack pointer is called the frame pointer and can be accessed using the alias %fp. The stack grows from addresses with larger numbers to addresses with smaller numbers.

As such, allocation of a stack frame is implemented by subtracting a value from the current stack pointer (actually, this usually done by adding a negative number to the stack pointer).

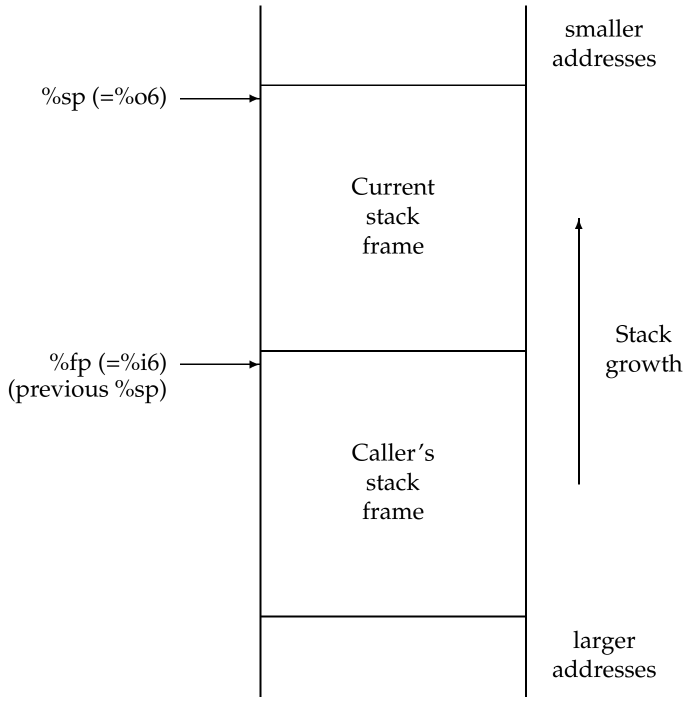


Figure 12.1 Stack frame allocation in the SPARC ABI

12.3.2 Parameters

As we noted in the previous two labs, registers %o0–%o5 are used for the first six parameters passed to a procedure. If a procedure has more than 6 parameters, the remaining parameters are passed on the stack. SPARC procedures do not push parameters (beyond the sixth parameter) onto the procedure call stack. Instead, they allocate space in their stack frame for the parameters and copy parameters into this space. This means that the called procedure will find its parameters (beyond the sixth) in the caller's stack frame. The called procedure can access these parameters using the frame pointer (%fp) with positive offsets.

12.3.3 Stack frame organization

As a minimum, every procedure that is not implemented as an optimized leaf procedure (i.e., any procudure that executes a save instruction) must allocate a stack frame or 64 bytes. This space will be used to store the input and local registers (%i0–%i7 and %l0–%l7) allocated by this procedure should you run out of register windows in a later procedure call.

Every nonleaf procedure must allocate an additional 7 words (28 bytes) in its stack frame. The first word of this space is used to store a "hidden" parameter. The hidden parameter is used for procedures that return structured values. Procedures that return simple values use %i0 (the caller's %o0) to return the result. However, if a procedure returns a structured value, the result may not fit in a register. In this case, the ccalling procedure must allocate space for the return value (probably in its stack frame). The calling procedure then puts the address of this space into the hidden parameter before making the call.

The remaining 6 words can be used by the called procedure to store the first six arguments (the ones passed in %o0–%o5). In most cases, the called procedure will be able to access these parameters in the registers %i0–%i5 and will not need to store them in the caller's stack frame. However, if the called procedure needs to take the address of a parameter, it needs to store the parameter into memory (you can't take the address of a register).

In addition to the regions that we have discussed, a procedure may allocate additional stack space for: alignment (the stack pointer should always be a multiple of 8), outgoing parameters (beyond the sixth parameter), automatic local arrays and other automatic local variables that don't fit in the local registers %l0–%l7, temporaries, and floating point registers. Figure 12.2 illustrates the organization of a SPARC stack frame.

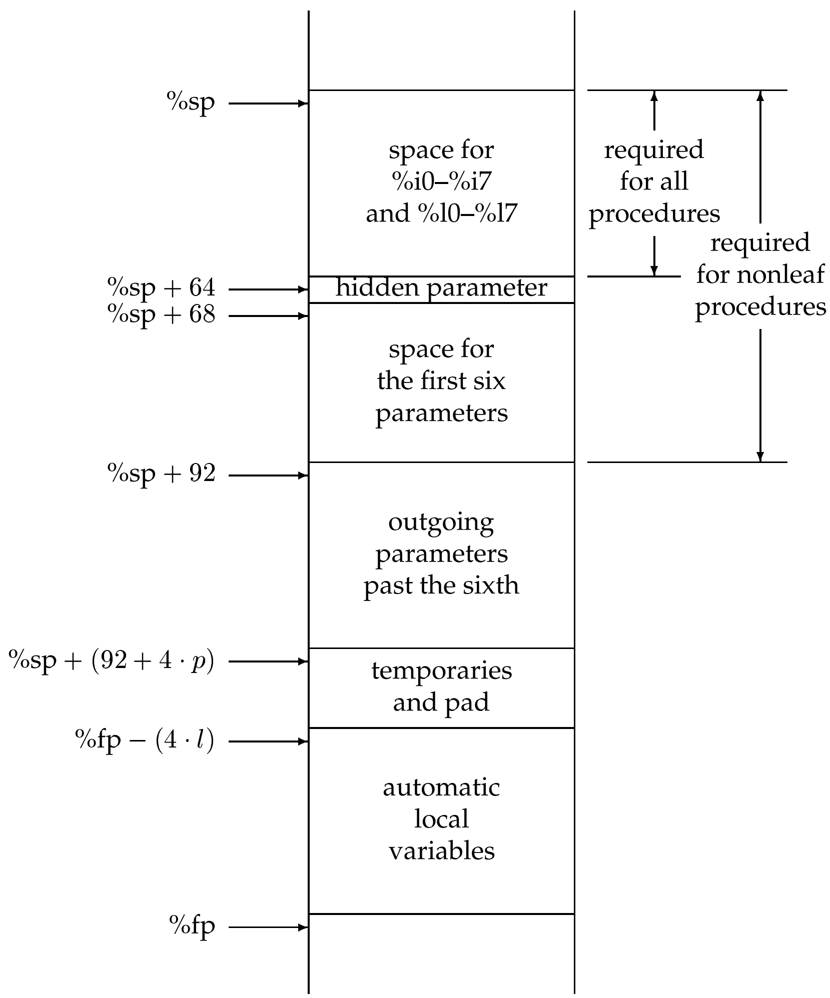


Figure 12.2 Stack frame organization

Example 12.1 *Translate the following C function into a SPARC procedure.*

```
int add7( int p1, int p2, int p3, int p4, int p5, int p6, int p7 )
{
  return p1 + p2 + p3 + p4 + p5 + p6 + p7;
}
```

In this case, the parameters p1–p6 will be in registers %i0–%i5. The parameter p7 will be in the caller's stack frame at offset 92 (that is, %fp + 92).

```
        .text
add7:   save    %sp, -64, %sp     ! this is a leaf procedure

        ld      [%fp+92], %l0     ! we'll eventually need p7

        add     %i0, %i1, %i0     ! add in p2
        add     %i0, %i2, %i0     ! add in p3
        add     %i0, %i3, %i0     ! add in p4
        add     %i0, %i4, %i0     ! add in p5
        add     %i0, %i5, %i0     ! add in p6

        ret
        restore %i0, %l0, %o0     ! add in p7
```

Example 12.2 *Translate the following C function which includes a call to the function defined in example 12.1*

```
int test( int x1, int x2 )
{
  int l1, l2;

  l1 = x1 + x2;
  l2 = x2 - x1;
  return add7( x1, x2, l1, l2, l1+l2, l2-l1, l2+l2 );
}
```

```
        .text
test:   save    %sp, -(92+4), %sp     ! allocate the minimum stack frame
                                      !  (includes a 4 byte ''alignment''
pad)

        add     %i0, %i1, %l0         ! l1 = x1 + x2;
        sub     %i1, %i0, %l1         ! l2 = x2 - x1;

        mov     %i0, %o0              ! first parameter
        mov     %i1, %o1              ! second parameter
        mov     %l0, %o2              ! third parameter
        mov     %l1, %o3              ! fourth parameter
        add     %l0, %l1, %o4         ! fifth parameter
        sub     %l1, %l0, %o5         ! sixth parameter

        add     %l1, %l1, %l2         ! temp = l2+l2
        call    add7
        st      %l2, [%sp+92]         ! seventh parameter (delay slot)

        ret
        restore %g0, %o0, %o0         ! return the result to caller's %o0
```

It is also common to access local variables stored in the stack using negative offsets from the frame pointer.

Example 12.3 *Translate the following C function into a SPARC procedure. You should assume that the procedures "read_int" and "write_int" are defined elsewhere.*

```
void read10( )
{
    int i;
    int a[10];

    for( i = 0 ; i < 10 ; i++ ) {
        a[i] = read_int();
    }

    for( i = 9 ; i >= 0 ; i-- ) {
        write_int( a[i] );
    }
}
```

In this case, we will use %l0 for *i* (scaled by 4) and the array *a* will be stored in the local space starting at %fp−40.

```
        .text
add7:   save    %sp, -(92+4*10+4), %sp  ! we need 40 words for the array

        sub     %fp, 40, %l1            ! l1 points to the start of the
array

        clr     %l0                     ! i = 0

top1:   call    read_int
        nop
        st      %o0, [%l1+%l0]          ! a[i] = read_int();

        inc     %l0, 4                  ! increment i += 4
        cmp     %l0, 40                 ! i < 10*4
        bl      top1
        nop

        mov    36, %l0                  ! i = 9*4
top2:   ld     [%l1+%l0], %o0           ! write_int( a[i] )
        call   write_int
        nop

        deccc  %l0, 4                   ! i -= 4
        bge    top2                     ! i >= 0
        nop

        ret
        restore
```

12.4 Summary

12.5 Review Questions

12.6 Exercises

Laboratory 13
Integer Arithmetic on the SPARC

13.1 Goal

To cover

13.2 Objectives

After completing this lab, you will be able to write SPARC programs that:

- Implement multiple precision arithmetic.

13.3 Discussion

13.4 Summary

13.5 Review Questions

13.6 Exercises

Laboratory 14
The Floating Point Coprocessor

14.1 Goal

To cover the floating point coprocessor on the SPARC.

14.2 Objectives

After completing this lab, you will be able to write SPARC programs that:

- Use floating point operations.

14.3 Discussion

14.4 Summary

14.5 Review Questions

14.6 Exercises

Laboratory 15
Linking and Loading

15.1 Goal

To cover the translation process implemented by the ISEM tools.

15.2 Objectives

After completing this lab, you will:

- coff

15.3 Discussion

15.4 Summary

15.5 Review Questions

15.6 Exercises

Laboratory 16
Traps

16.1 Goal

To cover the basic SPARC trap mechanism and trap instructions.

16.2 Objectives

After completing this lab, you will be able to write trap handlers:

- trap always
- conditional traps

16.3 Discussion

16.3.1 The Processor Status Register (PSR)

Figure 16.1 presents the fields in the processor status register.

Figure 16.1 The processor status register

16.3.2 Address Spaces

As noted in Lab 1, the SPARC uses separate address spaces for data and text (code). In fact, the SPARC provides (at least) four address spaces: the user instruction space, the supervisor instruction space, the user data space, and the supervisor data space. When the processor is in user state, instructions are fetched from the user instruction space while data values are loaded from and stored to user data memory. Similarly, when the processor is in supervisor mode, instructions are fetched from supervisor instruction space and data values are, by default, loaded from and stored to supervisor data memory.

When the processor is in supervisor state, you can use special load and store instructions to access data values in alternate memory spaces. For examples, you can load a value from the user data space, or store a value into the user instruction space. These instructions require an explicit address space indicator (ASI). Table 16.1 summarizes the ASI values used for these instructions.

Table 16.1 Address Spaces on the SPARC

Address space	ASI
User Instructions	7
Supervisor Instructions	8
User Data	9
Supervisor Data	10

16.3.3 The ROM Code

16.4 Review Questions

16.5 Exercises

1. Write a trap handler that will print a NULL terminated string in the user data space. (Because the string is in the user data space and not in supervisor data space, you cannot use "puts" function in rom.s to implement this trap.) The starting address of the string will be available in
   The functionality provided by this trap is not absolutely necessary. Application programs could attain equivalent functionality using multiple invocations of trap 1 (putc). What is the advantage of providing this as an separate trap?

Laboratory 17
Exceptions and Exception Handling

17.1 Goal

To cover exception handling on the SPARC.

17.2 Objectives

After completing this lab, you will:

- exception handlers.

17.3 Discussion

17.4 Summary

17.5 Review Questions

17.6 Exercises

Laboratory 18
Interrupts and Interrupt Handling

18.1 Goal

To cover interrupts and interrupt handling on the SPARC.

18.2 Objectives

After completing this lab, you will:

- interrupt handlers.

18.3 Discussion

18.4 Summary

18.5 Review Questions

18.6 Exercises

Laboratory 19
Context Switching

19.1 Goal

To cover context switching on the SPARC.

19.2 Objectives

After completing this lab, you will:

- multitasking.

19.3 Discussion

19.4 Summary

19.5 Review Questions

19.6 Exercises