SPARC Assembly Language

Program Organization Summary

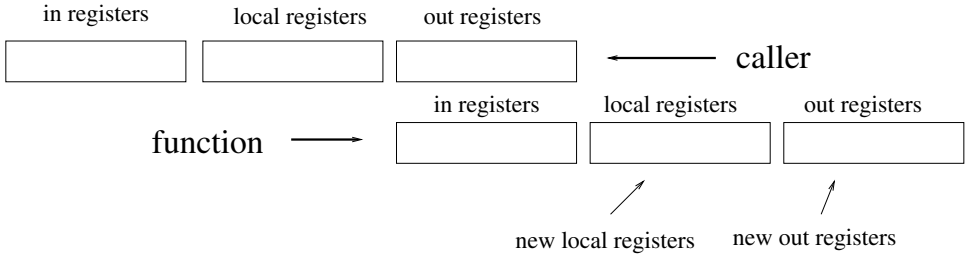
- Programs are organized into smaller units often called sub-programs, and also often called functions.
- Transferring the thread of execution (i.e., the next instruction) to a function is referred to as **calling** a function.
- While the thread of execution is in a function, the unit that called the function is "on hold", i.e., doing nothing. The caller will resume from the point of the call when the function completes.
- Transferring the thread of execution (i.e., the next instruction) from a function back to the caller is referred to as the function **return**.
- Information is passed to a function as a set of parameters. The mechanism for passing parameters varies among different computer designs.
- Functions perform the task for which they were designed
- Functions pass information back to the program unit that called it.

Memory Organization Summary

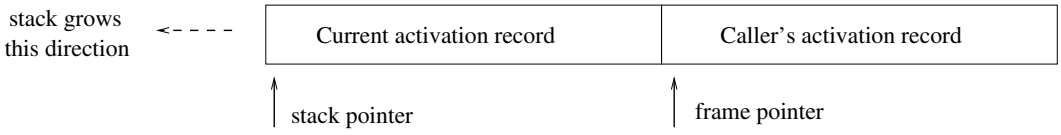
- When a function is called, a region of memory is allocated to serve as a working memory space for the function. This working memory space is known as the **activation record**.
- When a function ends the activation record is discarded (de-allocated).
- The allocation and de-allocation process is based on a **stack**. A **stack** is essentially a list in which all things added to the list and all things removed from the list are added (or removed) from one end of the list ONLY.
- Activation records contain space for any **local variables** that the function might use. You should think of a local variable as something that is private to the function. No other program unit should inspect or change a local variable.
- Activation records may also hold the address of the instruction to which the thread of execution will return on completion of the function. On SPARC systems, a different method is used to store the return address.

SPARC CPU Registers

- At any given moment in time, a function has access to 32 general purpose registers. Some of them are reserved for special functions.
- The 32 registers are organized in groups of 8: 8 **in** registers, 8 **out** registers, 8 **local** registers, and 8 **global** registers.
- Register names:
  - **in** registers: %i0, %i1, %i2, %i3 %i4, %i5, %i6, %i7
  - **out** registers: %o0, %o1, %o2, %o3 %o4, %o5, %o6, %o7
  - **local** registers: %l0, %l1, %l2, %l3 %l4, %l5, %l6, %l7
  - **global** registers: %g0, %g1, %g2, %g3 %g4, %g5, %g6, %g7
- A special mechanism exists in SPARC architecture for passing parameters to functions. The caller places the desired parameters into its **out** registers.
- As part of the function call, the registers are shifted according to the following diagram.



  I.e., the caller's **out** registers become the function's **in** registers. A function finds its input parameters in its input registers.
- As part of the function return, the registers are shifted back to the original configuration. The function returns information by storing it into its **in** registers. On return to the caller, the caller will find the result in its **out** registers.
- Register %g0 is hard-wired to the number zero. If you try to store a value into %g0 it will have no effect. Using %g0 as a source of zero for initializations (starting values) is sometimes useful.
- Do not store anything into %i7 or %o7. The call instruction saves its address into %o7. The called function has access to this address in register %i7 (in the context of the called function). This allows the called function to return to the caller using the instruction: `jmp %i7+8`
- Do not use registers %i6 or %o6. Those registers are used to store the frame pointer and the stack pointer (respectively) as part of the implementation of activation records. The frame and stack pointers are illustrated below.



SPARC Instructions

For illustration purposes we use registers `r0`, `r1`, and `r2`. In an actual program, you will use the names of the SPARC registers described earlier. For example, `%i0`, `%i1`, `%o0`, or `%l3`.

- `add  r0, r1, r2`
  - The `add` instruction requires three registers. On completion, register `r2` will contain the sum of `r0` plus `r1`.
- `sub  r0, r1, r2`
  - Similar to the `add` instruction, it requires three registers. On completion, register `r2` will contain the difference of `r0` minus `r1`. Optionally, a constant in the range -4096 to 4095 may be used in place of `r1`.
- `smul  r0, r1, r2`
  - Performs a multiply[1] with signed registers. On completion, register `r2` will contain the product of `r0` times `r1`.
- `mov  r0, r1`
  - The `mov` instruction makes a copy of `r0` into `r1`. On completion, register `r0` and `r1` will contain the same value. Optionally, `r0` may be replaced by a constant in the range -4096 to 4095. This is useful for initializations.
- `nop`
  - The `nop` instruction is the "no operation" instruction. I.e., do nothing. It is used immediately after a branch instruction.
- `ld  [ r0 ],r1`
  - Register `r0` must contain the **address** of the desired data. The `ld` instruction copies data from the memory location (address) indicated by `r0` into register `r1`. This operation is known as a "load".
- `st  r0,[ r1 ]`
  - Register `r1` must contain the **address** of a valid memory location. The `st` instruction copies data from register `r0` to the memory location indicated by `r1`. This operation is known as a "store".

SPARC Compare and Branching Instructions

Conditional execution of a sequence of instructions (if branches) and repetition of a sequence of instructions (loops) are implemented with the compare instruction (`cmp`), and various branch statements. Branch statements require a destination point within the program. Points within a program are established by labels followed by a colon. A label can be almost anything you want to use to name a point in a program; it must begin with a letter, and consist only of letters, digits, and possibly the underscore character.

[1] The divide instruction is complicated. For simplicity we will skip it.

Example of a label

```
L1:
          add   %i0, %i1, %l0
          smul  %l0,   3, %l1
```

A branch instruction which specifies `L1` will jump to point `L1` in the program and begin executing the `add` instruction shown above. On completion of the `add` instruction, processing will proceed to the `smul` instruction.

- `cmp  r0, r1`
  - The `cmp` instruction compares the contents of `r0` and `r1`, and sets the condition code register. If `r0` is less than `r1`, the condition code register will record the condition "less". Similar conditions exist to record "less or equal", "equal", "not equal", etc. The `cmp` instruction is usually used immediately before a conditional branch instruction.
- `ba  label`
  - The `ba` instruction is the "branch always" instruction. Control will be transferred to the point in the program indicated by the `label`. However, due to instruction pipelining, the instruction following the `ba` instruction will be loaded into the instruction register and on its way to the decoder before the program counter is set to the address indicated by the `label`. For this reason, all branching instructions should be followed by a `nop` instruction.
- `be  label`
  - The `be` instruction is the "branch equal" instruction. If the condition code register has recorded the condition "equals", then control will be transferred to the point in the program indicated by the `label`.
- `bne label`
  - The `bne` instruction is the "branch not equal" instruction.
- `bl  label`
  - The `bl` instruction is the "branch less" instruction.
- `ble  label`
  - The `ble` instruction is the "branch less or equal" instruction.
- `bg  label`
  - The `bg` instruction is the "branch greater" instruction.
- `bge  label`
  - The `ble` instruction is the "branch greater or equal" instruction.

Examples

Example 1: Suppose registers `%i0` and `%i1` each contain an integer. Suppose also that we would like to select the larger of those two numbers and store it in register `%l0`. Here is a segment of assembly code that will do that for us:[2]

```
        cmp %i0, %i1
        bl  L1          ! - Takes the jump to L1 only if %i0 less than %i1
        nop
! -        This is the "Not Less Than" branch.
        mov %i0, %l0
        ba  L2
        nop
L1:
! -        This is the "Less Than" branch.
        mov %i1, %l0
L2:
! - The rest of the program continues here.
```

Example 2: Suppose registers `%i0` and `%i1` each contain an integer. In this example, we will illustrate a complete function which adds the two numbers and returns their sum.

```
! - Header information for proper assembly.
    .align 8
    .skip 16
    .global f

f:                              ! -  Entry point (name) of the function.

       save %sp,-104,%sp        ! -  Create new context

! - Add the two inputs, store in %l0
        add %i0,%i1,%l0
! - Copy the result to the input register to return it.
       mov %l0,%i0
! - Do the return protocol
       jmp %i7+8
       restore
```

[2]Recall that the exclamation point at the begining of a line indicates that the line is a comment. The assembler (software to convert our program into machine language) ignores comment lines.

Example 3: Suppose we have two inputs: a number `n` and an array of length `n` containing integers. We want to add up the numbers in that array. On entry to our function, register `%i0` will contain `n`, and register `%i1` will contain the address of the beginning of the array (in memory). We want to put the sum of the numbers in the array into register `%l0`. The following code segment (complete function) will do this:

```
        .align 8
        .skip 16
        .global f

f:                                  ! -  Entry point.

            save %sp,-104,%sp       ! -  Create new context

! - Initialize %l0 to zero.  Register %l0 will hold the total
            mov %g0, %l0
! - Initialize %l1 to zero.  Register %l1 will count to n
            mov %g0, %l1
! - Start our loop.
StartLoop:
            cmp  %l1,%i0        ! - Compare counter %l1 to n
            bge  OutOfLoop
            nop
! - At this point we need to get our data item.  We begin
!   by computing the appropriate address.  Count by 4's.
            smul %l1,4,%l2
            add %i1,%l2,%l2
            ld [ %l2 ], %l3
            add %l0, %l3, %l0
! - We have accumulated our data item into our total. Now
!   we add one to our counter and go back to the beginning
!   of the loop.
            add %l1,1,%l1
            ba  StartLoop
            nop
OutOfLoop:
! - Rest of function goes here.  To return the sum, we copy
!   %l0 to %i0 and return to the caller.
            mov %l0,%i0
            jmp %i7+8
            restore
```