# Chapter 6
# Function Calling Mechanisms

Arthur B. Maccabe
Department of Computer Science
The University of New Mexico



## Overview

**Slide 1**

- Indirect branching
- Basic Mechanisms: call and return
- Argument passing
- C Functions
- Activation records
- Register windows on the SPARC

**Slide 2**

## Indirect Branching

- Jump and link, *jmpl*
  - target of branch is specified using a SPARC addressing mode
  - like load instruction, no square brackets
  - address of jmpl instruction is saved in the destination register
- Synthetic instruction, *jmp* (destination register is %r0)
- Encoding: uses load/store format, $op_3$ is 111000

**Slide 3**

## Implementing switch Statements

- Sample switch statement

```
switch (i) {
case 1:
    /* statement 1 */
    break;
case 2:
case 4:
    /* statement 2 */
    break;
case 6:
    /* statement 3 */
}
```

- Cascaded comparisons
- Branch table

## Cascaded Comparisons

**Slide 4**

```
.seg    "text"
sethi   %hi(i), %r2
ldw     [%r2+%lo(i)], %r3
cmp     %r3, 1
bne     %xcc, try2
nop                             ! branch delay slot
⋮                               ! code for stmt1
bra     %xcc, last
nop                             ! branch delay slot
```

## Cascaded Comparisons (continued)

**Slide 5**

```
try2:
        cmp     %r3, 2
        bre     %xcc, lab2
        nop                     ! branch delay slot
        cmp     %r3, 4
        bne     try3
        nop                     ! branch delay slot
lab2:
        ⋮                       ! code for stmt2
        bra     %xcc, last
        nop                     ! branch delay slot
```

## Cascaded Comparisons (continued)

**Slide 6**

```
try3:
        cmp     %r3, 6
        bne     %xcc, last
        nop                              ! branch delay slot
         ⋮                               ! code for stmt3
last:
```

## Branch Table (the table)

**Slide 7**

```
        .seg    "data"
tab:                                ; initialize the branch table
        .word   lab1                ; (i = 1) offset 0 → stmt1
        .word   lab2                ; (i = 2) offset 1 → stmt2
        .word   last                ; (i = 3) offset 2 → last
        .word   lab2                ; (i = 4) offset 3 → stmt2
        .word   last                ; (i = 5) offset 4 → last
        .word   lab3                ; (i = 6) offset 5 → stmt3
```

## Branch Table (branching)

**Slide 8**

```
.seg    "text"
sethi   %hi(i), %r2
ldsw    [%r2+%lo(i)], %r3
cmp     %r3, 1
blt     %xcc, last          ! nothing to do
cmp     %r3, 6
bgt     last                ! nothing to do
set     tab−4, %r2          ! normalized table base address
sll     %r3, 2, %r3         ! scale the index
ldw     [%r2+%r3], %r4      ! %r4 holds target address
jmp     %r4
nop                         ! branch delay slot
```

## Branch Table (the statements)

**Slide 9**

```
lab1:
        ⋮                       ! code for stmt1
        bra     %xcc, last
        nop                     ! branch delay slot
lab2:
        ⋮                       ! code for stmt2
        bra     %xcc, last
        nop                     ! branch delay slot
lab3:
        ⋮                       ! code for stmt3
 last:
```

## Preliminaries

**Slide 10**

- Functions
  - procedures
  - subroutines
- Arguments
  - formal arguments
  - actual arguments
  - parameters

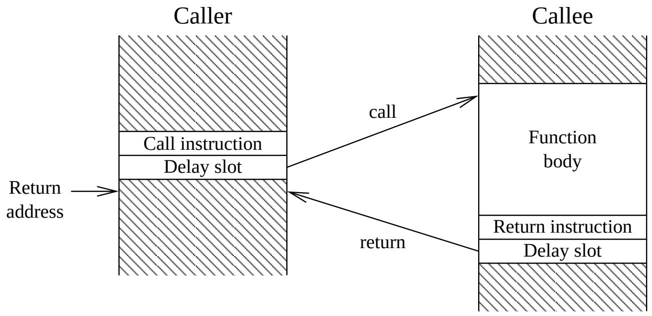
## Call and Return

**Slide 11**

- Basic terminology
- Linkage
- Linking the return address
- The SPARC call instruction
- Saving the return address
- Saving the registers

## Call and Return: Basic Terminology

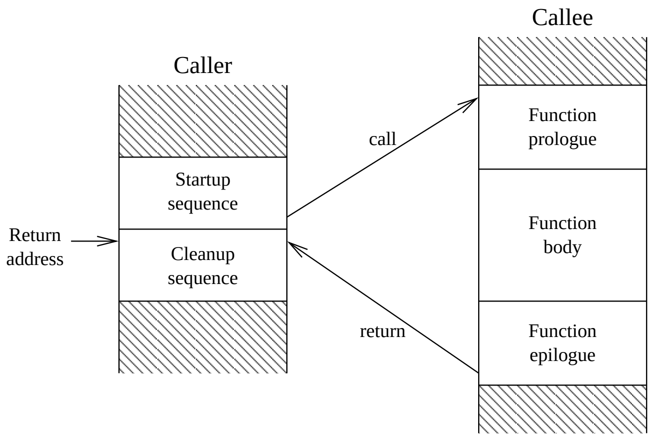
**Slide 12**




- Caller, Callee
- Function Call, Function Return
- Return address
- Function invocation

## Call and Return: Linkage

**Slide 13**




- Startup, Cleanup
- Prologue, Epilogue
- Calling convention

## Call and Return: Linking the Return Address

**Slide 14**

| | |
|---|---|
| Startup sequence | Store the return address in %r15. |
| | Branch to the function. |
| Function prologue | *nothing* |
| Function epilogue | Branch to the address stored in %r15. |
| Cleanup sequence | *nothing* |

## Call and Return: Linking the Return Address (SPARC Code, using *jmp*)

**Slide 15**

```
        set   ret, %r15     ! the startup sequence
        bra   %xcc, sub1    !
        nop                 ! delay slot
ret:
                            ! no cleanup sequence
        ⋮                   !
sub1:
                            ! no prologue
        ⋮                   ! the function body
        jmp   %r15          ! the epilogue
        nop
```

## The *call* and *retl* Instructions

**Slide 16**

- *call* saves address of call instruction in %r15
- *retl* (return from leaf) is a synthetic instruction

    jmpl %r15+8, %r0

## Code Example

**Slide 17**

```
        call  sub1   !
        nop          ! delay slot
                     ! no cleanup sequence
        ⋮            !
sub1:
                     ! no prologue
        ⋮            ! the function body
        retl         ! the epilogue
        nop
```

## Encoding Call Instructions

**Slide 18**

| 31 30 | 29 … 0 |
|---|---|
| 01 | disp30 |

| Field | Meaning |
|---|---|
| disp30 | Displacement value in 30-bit 2's complement representation |

The displacement is measured in instructions.

## Call and Return: Saving the Return Address

**Slide 19**

| | |
|---|---|
| Startup sequence | Branch to the function (save the call address in *%r15*). |
| Function prologue | Save the value in *%r15* in a memory location. |
| Function epilogue | Restore the call address and return. |
| Cleanup sequence | *Nothing.* |

## Call and Return: Saving the Return Address (SPARC Code)

**Slide 20**

```
        .seg    "data"
ret1:   .skip   4                            ! space to save the return addres
        .seg    "text"
sub1:
        sethi   %hi(ret1), %r1               ! prologue, save the return address
        stw     %r15, [%r1+%lo(ret1)]
        ⋮                                    ! the function body
        call    sub2
        ⋮                                    ! the function body
        sethi   %hi(ret1), %r1               ! epilogue, restore the return address
        lduw    [%r1+%lo(ret1)], %r15
        retl
        nop                                  ! delay slot
```

## Call and Return: Issues in Saving the Return Address

**Slide 21**

- Saving return address not needed for *leaf* functions
- Simple strategy not sufficient for *cyclic* calling sequences
- Recursive functions have cyclic calling sequences

**Slide 22**

## Call and Return: Saving the Registers

- Nonintrusive strategy (callee saves)

| | |
|---|---|
| Startup sequence | Branch to the function (save the call address in *%r15*). |
| Function prologue | For nonleaf functions, save *%r15* in memory. |
| | Save any registers used in the body of the function. |
| Function epilogue | Restore the registers saved in the prologue. |
| | Restore the call address and return. |
| Cleanup sequence | *Nothing.* |

- Defensive (caller saves)
- Limit the registers saved

**Slide 23**

## Call and Return: SPARC Register Usage

| Register(s) | Use | Volatility |
|---:|---|---|
| %r0 | Always zero | Unchanged |
| %r1–%r7 | Temporaries | Volatile |
| %r8 | Argument 1/Return Value | Volatile |
| %r9–%r13 | Arguments 2 through 6 | Volatile |
| %r14 | Stack Pointer | Unchanged |
| %r15 | Return Address | Volatile |
| %r16–%r29 | Local and Input registers | Unchanged |
| %r30 | Frame Pointer | Unchanged |
| %r31 | Input register | Unchanged |

## Argument Passing

**Slide 24**

- Pass-by-Value
- Pass-by-Value/Result
- Pass-by-Reference
- Contrasts

## An Example: String Length

**Slide 25**

- Assume the string starts at the location "str"
- Calling strlen

```
set   str, %r8   ! put the argument in R8
call  strlen     ! call the function
nop              ! delay slot
  ⋮              ! %r8 holds the return value
```

- Pass-by-value

**Slide 26**

## String Length: Body

```
        .seg    "text"
strlen:
        clr     %r9                     ! initialize the result
   top:
        ldub    [%r8+%r9], %r10         ! load the next character
        cmp     %r10, 0                 ! test for end of string
        bne,a   top
        inc     %r9                     ! one more character (delay slot)

        retl                            ! return
        mov     %r9, %r8                ! delay slot
```

**Slide 27**

## Value/Result

- AKA Copy-in/Copy-out
- Caller: copy inputs to argument registers; retrieve results from argument registers
- Callee: retrieve inputs from argument registers; put results in output registers
- Problem: %r8 holds the function return value, not the returned value of argument 1.
- We'll use %r9–%r13 for arguments

## Strange Example

Slide 28

- Behavior
  - takes two arguments
  - returns the product of the two arguments
  - modifies the first argument by adding the value of the second
- Implementation

```
        .seg    "text"
strange:
        ! the body
        mulx    %r9, %r10, %r8   ! calculate the result
        add     %r9, %r10, %r9   ! modify the first argument

        ! the epilogue
        retl                     ! return
        nop                      ! delay slot
```

## Calling *strange*: value/result

Slide 29

```
! z = strange( x, y ) – startup sequence
sethi   %hi(x), %r1
ldsw    [%r1+%lo(x)], %r9     ! first argument
sethi   %hi(y), %r1
call    strange
ldsw    [%r1+%lo(y)], %r10    ! second argument
! cleanup sequence
sethi   %hi(z), %r1
stw     %r8, [%r1+%lo(z)]     ! return value
sethi   %hi(x), %r1
stw     %r9, [%r1+%lo(x)]     ! first argument
```

## Reference

- Caller passes the address of the argument, i.e., a reference to the argument
- Callee manipulates the argument directly
- Implementing *strange*

**Slide 30**

```
strange:
        ldsw   [%r8], %r11          ! load value of first argument
        mulx   %r11, %r9, %r10      ! calculate the result
        add    %r11, %r9, %r11      ! calculate the sum
        stw    %r11, [%r8]          ! change the first argument

        ! the epilogue
        retl                        ! return
        mov    %r10, %r8            ! delay slot
```

## Calling *strange*: reference

**Slide 31**

```
! z = strange( x, y )
set     x, %r8                    ! the address of x
sethi   %hi(y), %r1               ! the value of y
call    strange
ldsw    [%r1+%lo(y)], %r9         ! delay slot
! cleanup sequence
sethi   %hi(z), %r1
stw     %r8, [%r1+%lo(z)]
```

## Summary

**Slide 32**

- Types of arguments: constant and variable
- Mechanisms: value, value/result, and reference
- Pass-by-value
  - appropriate for small constant arguments
  - called function should not alter the values passed by the caller
- Pass-by-value/result
  - appropriate for small variable arguments
  - caller's cleanup sequence stores the values of variable arguments
- Pass-by-value
  - appropriate for variable arguments and large constant arguments
  - called function should not alter the address values passed by the caller (i.e., addresses are passed by value)

## C Functions

**Slide 33**

- Function definition
  - header specifies return type and formal arguments
  - body is a compound statement and may declare "local" variables
- Arguments are always passed by value

  pass addresses to attain reference arguments

## The *strange* Function in C

**Slide 34**

```
int strange( int *p1, int p2 )
{
    int res;

    res = *p1 * p2;
    *p1 = *p1 + p2;
    return res;
}
```

## Calling *strange* in C

**Slide 35**

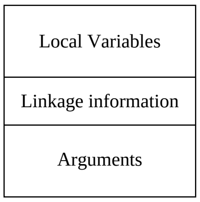
```
{
    int x, y, z;

    z = strange( &x, y );
}
```

## Activation Records

- Limitations of using registers
  - large arguments (more than a register)
  - large number of arguments (more than %r8 through %r13)
  - large number of local variables (more than %r1 through %r7)
  - calling a function in a function body

**Slide 36**

- Activation record




- Linkage information, linkage between caller and callee: return address, return value, etc.

## Allocating Activation Records

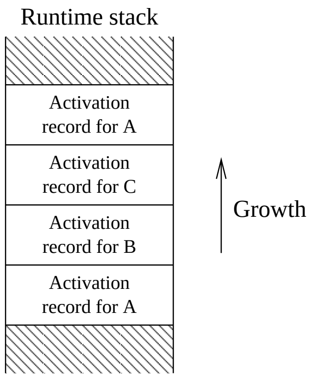
- Need dynamic allocation to support cyclic calling sequences (recursion)
- Need variable sized allocation (different activation records have different sizes)

**Slide 37**

- Fortunately we don't need general purpose allocation
  - allocations are associated with function calls
  - de-allocations are associated with function returns
  - functions returns are LIFO
  - we can use a stack of activation records

## The Stack of Activation Records

**Slide 38**




A calls B, B calls C, C calls A

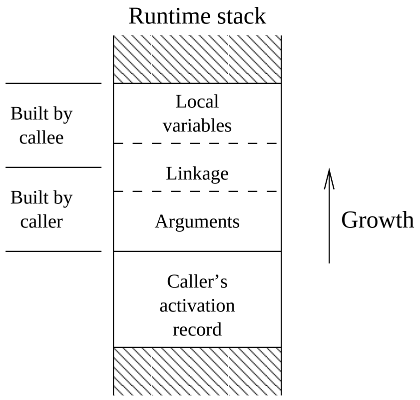
## Calling Convention

**Slide 39**

| | |
|---|---|
| Startup | Allocate stack space for the argument portion of the activation record (include space for all arguments). |
| | Copy arguments (beyond the first 6) to the activation record. |
| | Branch to the function (storing the return address in *%r15*). |
| Prologue | Allocate stack space needed for linkage information and local variables. |
| | For nonleaf functions, save the value in *%r15* in the linkage part of the activation record. |
| | Save any nonvolatile registers used in the body of the function. |
| Epilogue | Restore the registers saved in the prologue. |
| | Restore the return address. |
| | Deallocate the stack space allocated in the prologue. |
| Cleanup | Deallocate the stack space allocated in the startup. |

## Graphical Interpretation

**Slide 40**




## The Stack Pointer

**Slide 41**

- Points to the top of the stack
- In SPARC assembly language %sp is an alias for %r14
- Details are operating system dependent
  - initialized by the OS when the program is loaded
  - minimal alignment is 8 (e.g., an extended word), OS may require more

## Example

Slide 42

- Translate x = f( y, &z, w );
- Assume that x, y, z, and w are 32-bit globals
- SPARC code

```
add    %sp, −16, %sp          ! allocate argument space
sethi  %hi(y), %r1            ! the value of y
ldw    [%r1+%lo(y)], %r8
set    z, %r9                 ! the address of z
sethi  %hi(w), %r1            ! the value of w
call   f
ldw    [%r1+%lo(w)], %r10     ! delay slot
! cleanup sequence
add    %sp, 16, %sp           ! deallocate argument space
sethi  %hi(x), %r1
stw    %r8, [%r1+%lo(x)]
```

## The Frame Pointer

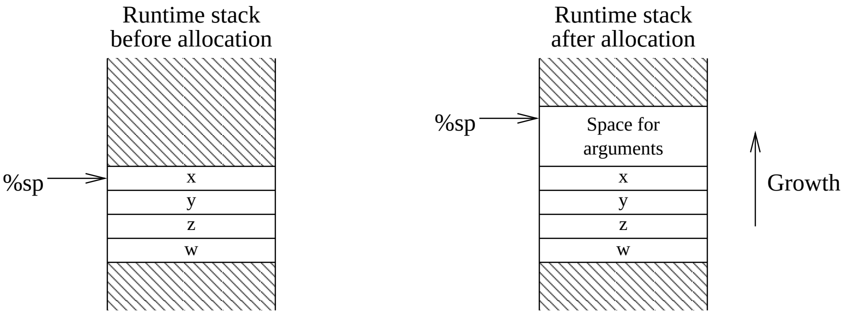
Slide 43

- The stack pointer changes, the frame pointer doesn't
- Assume x, y, z, and w are at offsets 0, 4, 8, and 12, respectively
- SPARC code

```
! x = f( y, &z, w )
add   %sp, −16, %sp          ! allocate argument space
ldw   [%sp+4+16], %r8        ! the value of y
add   %sp, 8+16, %r9         ! the address of z
call  f
ldw   [%sp+12+16], %r10      ! delay slot
! cleanup sequence
add   %sp, 16, %sp           ! deallocate argument space
stw   %r8, [%sp+0]           ! save the return value
```

## Graphical Interpretation

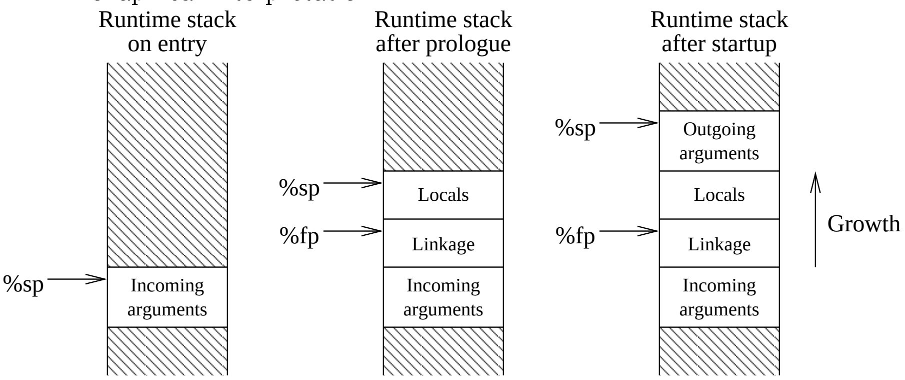
**Slide 44**




## Using a Frame Pointer

**Slide 45**

- In SPARC assembly language %fp is an alias for %r30
- Graphical Interpretation




- Arguments have positive offsets, locals have negative offsets

## Example Using the Frame Pointer

Slide 46

- Assume x, y, z, and w are at offsets −4, −8, −12, and −16, respectively
- SPARC code

```
! x = f( y, &z, w )
add  %sp, −16, %sp    ! allocate argument space
ldw  [%fp−8], %r8     ! the value of y
add  %fp, −12, %r9    ! the address of z
call f
ldw  [%fp−16], %r10   ! delay slot
! cleanup sequence
add  %sp, 16, %sp     ! deallocate argument space
stw  %r8, [%fp−4]     ! save the return value
```

## Calling Convention

Slide 47

| | |
|---|---|
| Startup | Allocate stack space for the argument portion of the activation record (include space for all arguments). |
| | Copy arguments (beyond the first 6) to the activation record. |
| | Branch to the function (storing the return address in *%r15*). |
| Prologue | Allocate stack space needed for linkage information and local variables. |
| | For nonleaf functions, save the return address (*%r15*) and the frame pointer (*%r30*)in the linkage part of the activation record. |
| | Save any nonvolatile registers used in the body of the function. |
| Epilogue | Restore the registers saved in the prologue. |
| | Restore the frame pointer and return address. |
| | Deallocate the stack space allocated in the prologue. |
| Cleanup | Deallocate the stack space allocated in the startup. |

**Slide 48**

## Example

- Show prologue and epilogue for *fun1* with 48 bytes of local space
- SPARC code

```
fun1:
        ! the prologue
        stx   %r15, [%sp−8]    ! save the return address
        stx   %fp, [%sp−16]    ! save the frame pointer
        mov   %sp, %fp         ! establish the frame pointer
        add   %sp, −64, %sp    ! alloc space for linkage and locals
        ...
        ! the epilogue
        add   %sp, 64, %sp     ! deallocate stack space
        ldx   [%sp−8], %r15    ! restore the return address
        retl
        ldx   [%sp−16], %fp    ! restore frame pointer (delay slot)
```

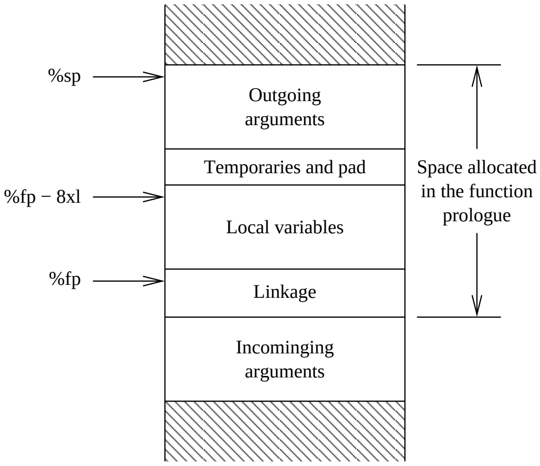
**Slide 49**

## Allocating Argument Space in the Prologue

| | |
|---|---|
| Startup | Copy arguments (beyond the first 6) to the activation record. |
| | Branch to the function (storing the return address in *%r15*). |
| Prologue | Allocate stack space for linkage information, local variables, and outgoing arguments. |
| | For nonleaf functions, save the return address (*%r15*) and frame pointer (*%r30*) in the linkage part of the activation record. |
| | Save any nonvolatile registers used in the body of the function. |
| Epilogue | Restore the registers saved in the prologue. |
| | Restore the frame pointer and return address. |
| | Deallocate the stack space allocated in the prologue. |
| Cleanup | *Nothing* |

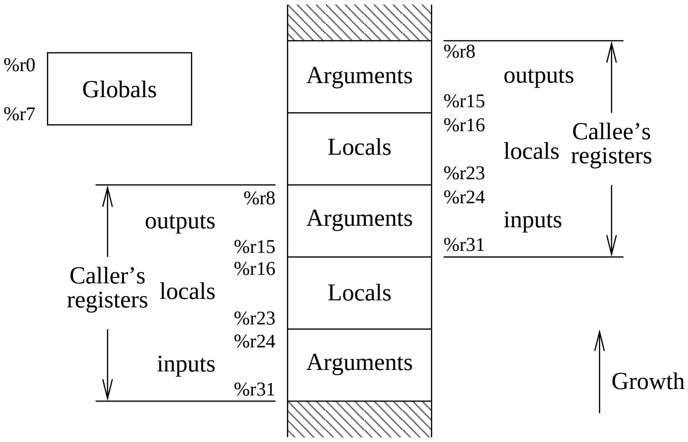
## Graphical Interpretation

**Slide 50**




## Register Windows on the SPARC

**Slide 51**



## Alternate Register Names

**Slide 52**

| Basic names | Alternate names | Uses |
|---|---|---|
| %r0–%r7 | %g0–%g7 | Globals, shared by all functions |
| %r8–%r15 | %o0–%o7 | Outputs, output arguments and temporaries |
| %r16–%r23 | %l0–%l7 | Locals, temporaries |
| %r24–%r31 | %i0–%i7 | Inputs, input arguments |

## Register Uses in Light of Register Windows

**Slide 53**

| Register(s) | Use | Window usage |
|---:|---|---|
| %r0 | Always zero | %g0 |
| %r1–%r7 | Temporaries | Global registers |
| %r8 | Argument 1/Return Value | Output registers |
| %r9–%r13 | Arguments 2 through 6 | Output registers |
| %r14 | Stack Pointer | Output registers |
| %r15 | Return Address | Output registers |
| %r16–%r23 | Local registers | Local registers |
| %r24–%r29 | Input registers | Input registers |
| %r30 | Frame Pointer | Input, matches %r14 |
| %r31 | Input register | Input, matches %r15 |

## Simpler Calling Convention

**Slide 54**

| | |
|---|---|
| Startup | Copy arguments (beyond the first 6) to the activation record. |
| | Branch to the function (storing the return address in *%r15*). |
| Prologue | For nonleaf functions, allocate a new set of registers saving the return address (*%r15*) and current stack pointer (*%r14*). |
| | Allocate stack space for local variables and outgoing arguments. |
| Epilogue | Deallocate the set of registers allocated in the prologue, restoring the old stack pointer and return address. |
| Cleanup | *Nothing* |

## SPARC Instructions

**Slide 55**

- Save
  - allocate a register set (save caller's registers)
  - addition, sources from old regs, destination to new regs
  - frequently used to allocate stack space
  - prologue is one instruction, e.g.,

    save %sp, 128, %sp
- Restore
  - deallocate a register set (restore caller's registers)
  - also does and addition, but not often used
- Synthetic versions use %r0 for all operands
- Encoded using arithmetic format
  - $\text{op}_3$ is 111000 for save and 111001 for restore

## The *ret* Instruction

**Slide 56**

- Using *retl*

```
! the epilogue
restore                  ! restore the caller's register set
retl                     ! return, using the caller's %r15
nop                      ! delay slot
```

- Putting restore in the delay slot

```
! the epilogue
jmpl    %r31+8,%r0       ! return, using the caller's %r15
restore                  ! restore the caller's register set
```

- Using *ret*

```
! the epilogue
ret                      ! return, using the caller's %r15
restore                  ! restore the caller's register set
```

## Stack Frame Organization

**Slide 57**