Chapter 6
Function Calling Mechanisms

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


Overview

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

- Jump and link, \texttt{jmp}
  - target of branch is specified using a SPARC addressing mode
- Like load instruction, no square brackets
  - address of \texttt{jmp} instruction is saved in the destination register
- Synthetic instruction, \texttt{jmp} (destination register is \texttt{\%r0})
- Encoding: uses load/store format, \texttt{op\_3} is 111000

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

```
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)

```
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)

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

last:

Branch Table (the table)

     .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)

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)

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

- Functions
  - procedures
  - subroutines

Slide 10

- Arguments
  - formal arguments
  - actual arguments
  - parameters

Call and Return

- Basic terminology
- Linkage

Slide 11

- Linking the return address
- The SPARC call instruction
- Saving the return address
- Saving the registers
Call and Return: Basic Terminology

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

Call and Return: Linkage

- Startup, Cleanup
- Prologue, Epilogue
- Calling convention
Call and Return: Linking the Return Address

**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)**

```
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

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

```assembly
jmpl   %r15+8, %r0
```

Code Example

```assembly
call   sub1       !
nop    ! delay slot
        ! no cleanup sequence
:
sub1:
        ! no prologue
:
    ! the function body
retl   ! the epilogue
nop
```
Encoding Call Instructions

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>disp30</td>
<td>Displacement value in 30-bit 2's complement representation</td>
</tr>
</tbody>
</table>

The displacement is measured in instructions.

Call and Return: Saving the Return Address

- **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)

```
seg    "data"
ret1:  skip 4                      ! space to save the return address
seg    "text"

sub1:
  sethi %hi(ret1), %r1           ! prologue, save the return address
  stw   %r15, [%r1+%lo(ret1)]
  call  sub2                      ! the function body
  :                                 
  :                                 ! the function body
  sethi %hi(ret1), %r1           ! epilogue, restore the return address
  lduw  [%r1+%lo(ret1)], %r15
  ret1
  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
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

---

Call and Return: SPARC Register Usage

<table>
<thead>
<tr>
<th>Register(s)</th>
<th>Use</th>
<th>Volatility</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r0</td>
<td>Always zero</td>
<td>Unchanged</td>
</tr>
<tr>
<td>%r1-%r7</td>
<td>Temporaries</td>
<td>Volatile</td>
</tr>
<tr>
<td>%r8</td>
<td>Argument 1/Return Value</td>
<td>Volatile</td>
</tr>
<tr>
<td>%r9-%r13</td>
<td>Arguments 2 through 6</td>
<td>Volatile</td>
</tr>
<tr>
<td>%r14</td>
<td>Stack Pointer</td>
<td>Unchanged</td>
</tr>
<tr>
<td>%r15</td>
<td>Return Address</td>
<td>Volatile</td>
</tr>
<tr>
<td>%r16-%r29</td>
<td>Local and Input registers</td>
<td>Unchanged</td>
</tr>
<tr>
<td>%r30</td>
<td>Frame Pointer</td>
<td>Unchanged</td>
</tr>
<tr>
<td>%r31</td>
<td>Input register</td>
<td>Unchanged</td>
</tr>
</tbody>
</table>
Argument Passing

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

An Example: String Length

- Assume the string starts at the location “str”
- Calling strlen

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

- Pass-by-value
String Length: Body

```
.slide
  .seg  "text"
  strlen:
  
    dr   %r9                    ! initialize the result
  
  top:
  
    klub [%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
```

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

- 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
  ret1  ! return
  nop   ! delay slot
```

Calling *strange*: value/result

```
! 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`

```
strange:
  ldsw [%r8], %r11       ! load value of first argument
  mulp %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

```
! 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)]
```

Slide 30

Slide 31

16
Summary

- 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

Slide 32

- 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

- 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

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

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

Calling *strange* in C

```c
{
    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

<table>
<thead>
<tr>
<th>Local Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linkage information</td>
</tr>
<tr>
<td>Arguments</td>
</tr>
</tbody>
</table>

- 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

A calls B, B calls C, C calls A

Calling Convention

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

The Stack Pointer

- 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

- 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
seti %hi(y), %r1        ! the value of y
ldw  [%r1+lo(y)], %r8
set  z, %r9             ! the address of z
seti %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
seti %hi(x), %r1
stw  %r8, [%r1+lo(x)]
```

The Frame Pointer

- 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

Using a Frame Pointer

- 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

- Assume x, y, z, and w are at offsets -4, -8, -12, and -16, respectively
- SPARC code
  
  Slide 46
  
  \[
  \begin{align*}
  &! x = f(y, e, z, w) \\
  &\text{add} \ %s, -16, %s \ ! \text{allocate argument space} \\
  &\text{ldw} \ [\%fp-8], %r8 \ ! \text{the value of y} \\
  &\text{add} \ %fp, -12, %r9 \ ! \text{the address of z} \\
  &\text{call} \ f \\
  &\text{ldw} \ [\%fp-16], %r10 \ ! \text{delay slot} \\
  &! \text{cleanup sequence} \\
  &\text{add} \ %s, 16, %s \ ! \text{deallocate argument space} \\
  &\text{stw} \ %r8, [\%fp-4] \ ! \text{save the return value}
  \end{align*}
  \]

Calling Convention

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.
Dealocate the stack space allocated in the prologue.

Cleanup
Dealocate the stack space allocated in the startup.
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)
  ```

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

%sp
Outgoing arguments
Temporaries and pad
Local variables
Linkage
Incoming arguments
Space allocated in the function prologue

Register Windows on the SPARC

%r0 %r7
Globals

%r8 %r15 %r16 %r23 %r24 %r31
Arguments outputs
Locals
Callee’s registers

%r0 %r7
outputs
%r8 %r15 %r16 %r23 %r24 %r31
Arguments inputs
Locals

%r0 %r7
inputs
%r8 %r15 %r16 %r23 %r24 %r31
Arguments

Caller’s registers locals

Growth
### Alternate Register Names

<table>
<thead>
<tr>
<th>Basic names</th>
<th>Alternate names</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r0-%r7</td>
<td>%g0-%g7</td>
<td>Globals, shared by all functions</td>
</tr>
<tr>
<td>%r8-%r15</td>
<td>%o0-%o7</td>
<td>Outputs, output arguments and temporaries</td>
</tr>
<tr>
<td>%r16-%r23</td>
<td>%l0-%l17</td>
<td>Locals, temporaries</td>
</tr>
<tr>
<td>%r24-%r31</td>
<td>%i0-%i17</td>
<td>Inputs, input arguments</td>
</tr>
</tbody>
</table>

### Register Uses in Light of Register Windows

<table>
<thead>
<tr>
<th>Register(s)</th>
<th>Use</th>
<th>Window usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r0</td>
<td>Always zero</td>
<td>%g0</td>
</tr>
<tr>
<td>%r1-%r7</td>
<td>Temporaries</td>
<td>Global registers</td>
</tr>
<tr>
<td>%r8</td>
<td>Argument 1/Return Value</td>
<td>Output registers</td>
</tr>
<tr>
<td>%r9-%r13</td>
<td>Arguments 2 through 6</td>
<td>Output registers</td>
</tr>
<tr>
<td>%r14</td>
<td>Stack Pointer</td>
<td>Output registers</td>
</tr>
<tr>
<td>%r15</td>
<td>Return Address</td>
<td>Output registers</td>
</tr>
<tr>
<td>%r16-%r23</td>
<td>Local registers</td>
<td>Local registers</td>
</tr>
<tr>
<td>%r24-%r29</td>
<td>Input registers</td>
<td>Input registers</td>
</tr>
<tr>
<td>%r30</td>
<td>Frame Pointer</td>
<td>Input, matches %r14</td>
</tr>
<tr>
<td>%r31</td>
<td>Input register</td>
<td>Input, matches %r15</td>
</tr>
</tbody>
</table>
Simpler Calling Convention

**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**
Dealocate the set of registers allocated in the prologue,

restoring the old stack pointer and return address.

**Cleanup**
*Nothing*

---

SPARC Instructions

- 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
  - op3 is 111000 for save and 111001 for restore

---

Slide 55

---

Slide 54
The `ret` Instruction

- Using `ret`  
  ```
  ! the epilogue
  restore! restore the caller’s register set
  ret! return, using the caller’s %r15
  nop! delay slot
  ```

- Putting restore in the delay slot  
  ```
  ! the epilogue
  jmp! return, using the caller’s %r15
  %r31+8,%r0! restore the caller’s register set
  restore
  ```

- Using `ret`  
  ```
  ! the epilogue
  ret! return, using the caller’s %r15
  restore! restore the caller’s register set
  ```

Stack Frame Organization

- `%sp`  
  - Space for the register window (128 bytes)
  - Locals and outputs

- `%sp + 128`  
  - Hidden parameter (8 bytes)

- `%sp + 136`  
  - Space for the first six outgoing arguments (48 bytes)

- `%sp + 184`  
  - Other outgoing arguments

- `%fp`  
  - Local variables

- `%fp - 8xl`  
  - Temporaries and pad

Slide 56

Slide 57

29