Procedure Call Instructions

- Procedure calls involve the following actions:
  1. passing arguments
  2. saving a "return address"
  3. transferring from the caller to the callee
  4. returning from the callee to the caller
  5. returning the results

- Simplest examples include assembly-language intrinsic procedures, like:
  ```
  st %0, a
  ld c, %0
  call .mul
  id b, %0
  id p, %0
  
  optimized
  st %0, a
  nop
  call .mul
  id c, %0
  id b, %0
  id p, %0
  
  a = b * c;
  ```

- The arithmetic intrinsics .mul, etc. must be used.

- Procedure calls involve the following actions:
  1. passing arguments
  2. saving a "return address"
  3. transferring from the caller to the callee
  4. returning from the callee to the caller
  5. returning the results

- Simplest examples include assembly-language intrinsic-"leaf" procedures, like:
  ```
  a = b * c;
  ```
Call/Return Instructions

• Procedures are called with either **call** or **jmpl**

**call** instruction: format 1 instruction

- Leaves PC to 32-bit address by **address**, which may be any addressing mode
- Leaves PC in **reg**, i.e., the location of the **call**, in %r7 (%r15)

**jmpl** instruction: format 3 instruction

- Jumps to 32-bit address by **address reg**, which may be any addressing mode
- Jumps to **PC + 4 x zeroextend(disp30)**

<table>
<thead>
<tr>
<th>4</th>
<th>12</th>
<th>18</th>
<th>24</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>disp30</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Calls are called with either **call** or **jmpl**
Indirect Calls

`jmpl reg, %r15` implements indirect calls. It jumps to the 32-bit address specified in `reg`.

For example, consider the following code snippet:

```assembly
ld b, %o0
ld c, %o1
ld apply, %o3
jmpl %o3, %r15; nop
```

This code snippet demonstrates how `jmpl` is used to transfer control from the callee to the caller, similar to `ret` and `retl`.

Why `+8`?

This `+8` ensures that the control transfer is performed correctly, considering the alignment and structure of the function pointer.

In summary, `jmpl` is a powerful instruction for implementing indirect procedure calls and returning control to the caller. It provides a flexible way to handle function calls without the need for explicit stack manipulation.
Procedure Calls

- **Entry and exit sequences** collaborate to implement these requirements:
  - Saving and restoring caller's registers
  - Allocating and deallocating space for locals
  - Passing and returning structures
  - Passing a variable number of arguments

- **Other requirements**:
  - Must work when, e.g., B is A, etc.

- Procedure implementation must handle **nested and recursive calls**.

```plaintext
A: call B
    \( \text{C returns} \)
B: call C
    \( \text{C returns} \)
```

Copyright ©1995 D. Hanson, K. Li, J.P. Singh

October 22, 1997
• Procedure call information is stored in the stack, locals, including compiler “temporaries”
• SPARC’s stack grows downwards, i.e., from high to low addresses
• callee’s arguments, if necessary
callee’s registers, if necessary
locals, including compiler “temporaries”

procedure call information is stored in the stack
Arguments and Return Values

By convention, the first 6 arguments are passed in registers; the rest are passed on the stack (97% of procedures have 6 or fewer arguments). The first 6 arguments are passed in registers; the rest are passed on the stack.

caller

callee

caller finds the arguments in the "out" registers;
callee finds the return value in the "in" registers;
callee places the arguments in the "out" registers;
caller finds the return value in the "in" registers;
caller places its return value in the "in" registers;
callee finds the return value in the "out" registers;

%0 first argument
%1 second argument
%2 ... ... ...
%5 sixth argument

%0 first return value
%1 second return value
%2 ... ... ...
%5 sixth return value

frame pointer

stack pointer

return address
Register Windows

- SPARC register windows: each procedure gets 16 "new" registers
- The SPARCS have 2–32 windows
- The window "slides" at a call
- "backwards" restore slides the window
- "forward" save slides the window
- callee's in registers become synonymous with the caller's out registers
- " callee" register windows: each
- caller
• Most SPARC's have 8 windows

Register Windows, cont'd
Window Management

In save and restore instructions, registers refer to the new window:

- **Save instruction**:save %sp, N, %sp
  - e.g., save %sp, -4*16, %sp

- If an overflow occurs, the registers are saved on the stack.

  - If an overflow occurs, the registers are saved on the stack.

- In save and restore instructions, registers refer to the current window:

  - **Save instruction**:save %sp, N, %sp

- In save and restore instructions, registers refer to the current window:

  - **Save instruction**:save %sp, N, %sp

- In save and restore instructions, registers refer to the current window:

  - **Save instruction**:save %sp, N, %sp

- In save and restore instructions, registers refer to the current window:

  - **Save instruction**:save %sp, N, %sp

- In save and restore instructions, registers refer to the current window:

  - **Save instruction**:save %sp, N, %sp

- In save and restore instructions, registers refer to the current window:

  - **Save instruction**:save %sp, N, %sp

Stack Frame

- Saved ins and locals
  - 16 words to hold
- Pointer to struct return value
- Argument 1
- Argument 2
- ...
- Argument 6
- Outgoing arguments 7, 8...
- Saved floating point registers
- Locals, temporaries
- Previous stack frame

Always allocated

See page 189 in the SPARC Architecture Manual, §7.5 in Paul
First 6 arguments are passed in %00—%05, the rest in the Stack

C Calling Convention
Example Stack Frames

```c
main() {
t(1,2,3,4,5,6,7,8);
}
t(int a1, int a2, int a3, int a4, int a5, int a6, int a7, int a8) {
    int b1 = a1;return s(b1, a8);
}
s(int c1, int c2) {
    return c1 + c2;
}
```
Example Stack Frames, cont'd