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 “leaf” procedures, like the arithmetic intrinsics `.mul`, etc.

```plaintext
a = b*c;
ld b, %0
ld c, %1
call .mul
nop
st %0, a

optimized
ld b, %0
call .mul
ld c, %1
ld %0, a
st %0, a
```
Call/Return Instructions

- Procedures are called with either `call` or `jmpl`

- **call** instruction

  \[
  \text{call} \quad \text{label}
  \]

  a format 1 instruction

  \[
  \begin{array}{cccccc}
  \text{01} & \text{disp30} \\
  31 & 29 & 24 & 18 & 13 & 12 & 4
  \end{array}
  \]

  jumps to \( \textbf{PC} + 4 \times \text{zeroextend}(\text{disp30}) \)

  leaves \( \textbf{PC} \), i.e. the location of the \textbf{call}, in \( \%07(\%r15) \)

- **jmpl** instruction

  \[
  \text{jmpl} \quad \text{address}, \text{reg}
  \]

  format 3 instruction

  \[
  \begin{array}{ccccccc}
  \text{10} & \text{reg} & 111000 & \text{rs1} & i=0 & 0 & \text{rs2} \\
  \text{10} & \text{reg} & 111000 & \text{rs1} & i=1 & \text{simm13} & \text{rs2}
  \end{array}
  \]

  jumps to 32-bit address by \textbf{address}, which may be any addressing mode

  leaves PC in \text{reg}
Indirect Calls

- `jmpl` implements *indirect calls*

  ```
  jmpl  reg, %r15
  ```

  jumps to the 32-bit address specified in `reg`

  leaves `PC` — the return address — in `%r15`

  e.g., for function pointers

  ```
  a = (*apply)(b, c);
  ld b, %o0
  ld c, %o1
  ld apply, %o3
  jmpl %o3, %r15; nop
  st %o0, a
  ```

- `jmpl` implements procedure return

  ```
  jmpl  %r15+8, %g0
  ```

  transfers control from the callee to the caller (see also `ret` and `retl`)

  why +8?
Procedure Calls

- Procedure implementation must handle **nested** and **recursive** calls
e.g., A calls B, B calls C

\[
\begin{align*}
A: & \quad \text{call } B \\
& \quad \text{return} \\
B: & \quad \text{call } C \\
& \quad \text{return} \\
C: & \quad \text{return}
\end{align*}
\]

must work when, e.g., B is A, etc.

- Other requirements
  - passing a variable number of arguments
  - passing and returning structures
  - allocating and deallocating space for locals
  - saving and restoring caller’s registers

- **Entry** and **exit** sequences collaborate to implement these requirements
Stack

- Procedure call information is stored in the stack locals, including compiler “temporaries”
  caller’s registers, if necessary
  callee’s arguments, if necessary

- SPARC’s stack grows downwards, i.e. from high to low addresses

- The stack pointer, %sp (%r14) points to the top 32-bit word on the stack
  %sp **must** always be a multiple of 8

- Stack operations
  to push %o1
    ```
    dec 4,%sp
    st %o1,[%sp]
    ```
  to pop top word into %o1
    ```
    ld [%sp],%o1
    inc 4,%sp
    ```
  to allocate \( N \) bytes of stack space
    ```
    sub %sp,\( N \),%sp
    ```
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)

- Caller places the arguments in the “out” registers; callee finds its arguments in the “in” registers

<table>
<thead>
<tr>
<th>caller</th>
<th>what</th>
<th>callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>%07</td>
<td>return address - 8</td>
<td>%17</td>
</tr>
<tr>
<td>%06</td>
<td>stack pointer</td>
<td>%16</td>
</tr>
<tr>
<td>%05</td>
<td>sixth argument</td>
<td>%15</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>%01</td>
<td>second argument</td>
<td>%11</td>
</tr>
<tr>
<td>%00</td>
<td>first argument</td>
<td>%10</td>
</tr>
</tbody>
</table>

- Callee places its return value in the “in” registers; caller finds the return value in the “out” registers

<table>
<thead>
<tr>
<th>caller</th>
<th>what</th>
<th>callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>%05</td>
<td>sixth return value</td>
<td>%15</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>%01</td>
<td>second return value</td>
<td>%11</td>
</tr>
<tr>
<td>%00</td>
<td>first return value</td>
<td>%10</td>
</tr>
</tbody>
</table>
Register Windows

- **SPARC register windows**: each procedure gets 16 “new” registers

- The window “slides” at a call
  - callee’s in registers become synonymous with the caller’s out registers

- The SPARCs have 2–32 windows

- **save** slides the window “forward”

- **restore** slides the window “backwards”
Register Windows, cont’d

- Most SPARCs have 8 windows

- **save/restore** decrement/increment the current window pointer, **CWP**
Window Management

• **save** instruction

  \[
  \text{save } \%sp, N, \%sp \quad \text{e.g., save } \%sp, -4*16, \%sp
  \]

  slides the register window so the current window becomes the previous window
decrements the current window pointer (\textit{CWP}) and checks for window \textbf{overflow}
adds \( N \) to the stack pointer, \( \%sp \); i.e., allocates \( N \) bytes if \( N < 0 \)

• If an overflow occurs, the registers are saved on the stack
  there \textbf{must} be enough stack space

• **restore** instruction

  slides the register window so the previous window becomes the current window
  increments the current window pointer (\textit{CWP}) and checks for window \textbf{underflow}

• In \textit{save} and \textit{restore}

  \textbf{source} registers refer to the \textit{current} window
  \textbf{destination} registers refer to the \textit{new} window
Stack Frame

- see page 189 in the SPARC Architecture Manual, §7.5 in Paul
C Calling Convention

- First 6 arguments are passed in %0 — %05, the rest in the stack

```c
char out[30], str[] = "this is a sample string";
main() { bcopy(out, str, sizeof str); }
bcopy(char *dst, char *src, int nbytes) { ... }
```

- Assembly language

```assembly
.seg "bss"
.global _out
.common _out,30
.seg "data"
.global _str
_str:.ascii "this is a sample string\000"
.seg "text"
.global _main
_main: save %sp,-96,%sp
    set _out,%0
    set _str,%01
    call _bcopy
    set 24,%02
    ret; restore
.global _bcopy
_bcopy:...
    retl; nop
```
Example Stack Frames

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;
}

_main: save %sp,-104,%sp
    set 1,%o0
    set 2,%o1
    set 3,%o2
    set 4,%o3
    set 5,%o4
    set 6,%o5
    set 7,%i5
    st %i5,[%sp+4*6+68]
    set 8,%i5
    st %i5,[%sp+4*7+68]
    call _t; nop
    ret; restore

_t: save %sp,-96,%sp
    st %i0,[%fp-4]
    ld [%fp-4],%o0
    ld [%fp+96],%o1
    call _s; nop
    mov %o0,%i0
    ret; restore

_s:
    add %o0,%o1,%o0
    retl; nop
Example Stack Frames, cont’d

```
main

+96
a8
+92
a7
+88
(a6)
+84
(a5)
+80
(a4)
+76
(a3)
+72
(a2)
+68
(a1)
+64
struct *

16 words

+88
b1
+84
(argument 6)
+80
(argument 5)
+76
(argument 4)
+72
(argument 3)
+68
(c2)
+64
(c1)
struct *

16 words

+4
%fp

%sp
```