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.

```
a = b*c;
```

```
ld b,%o0
ld c,%o1
call .mul
nop
st %o0,a
```

optimized

```
ld b,%o0
call .mul
ld c,%o1
st %o0,a
```


Call/Return Instructions

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

`call` *label*

a format 1 instruction

01	*disp30*					
31	**29**	**24**	**18**	**13**	**12**	**4**

jumps to ***PC*** $+ 4 \times \mathrm{zeroextend}($***disp30***$)$
leaves ***PC***, i.e. the location of the **`call`**, in **`%o7`** (**`%r15`**)

- jmpl instruction

`jmpl` *address*, *reg*

format 3 instruction

10	*reg*	*111000*	*rs1*	*i=0*	*0*	*rs2*
10	*reg*	*111000*	*rs1*	*i=1*	*simm13*	
31	**29**	**24**	**18**	**13**	**12**	**4**

jumps to 32-bit address by ***address***, which may be any addressing mode
leaves PC in ***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`

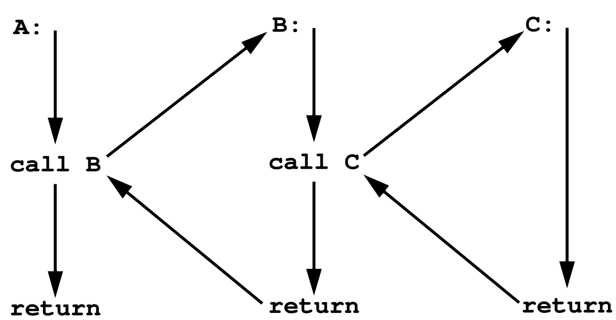
  transfers control from the callee to the caller (see also **`ret`** and **`retl`**)

  why **`+8`**?

Procedure Calls

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

  e.g., **`A`** calls **`B`**, **`B`** calls **`C`**



  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

caller	***what***	***callee***	
`%o7`	return address **– 8**	`%i7`	
`%o6`	***stack*** pointer	`%i6`	***frame*** pointer
`%o5`	sixth argument	`%i5`	
...	...	...	
`%o1`	second argument	`%i1`	
`%o0`	first argument	`%i0`	

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

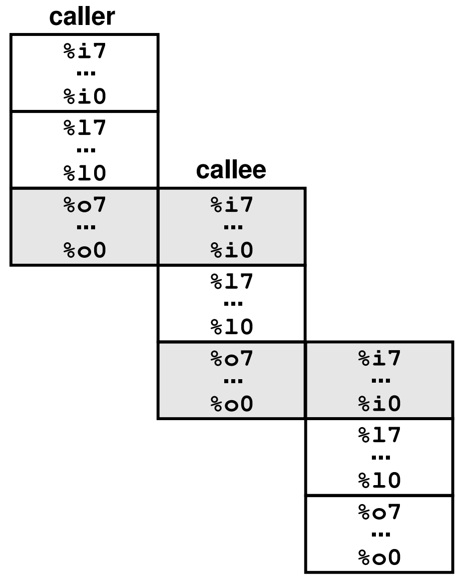
caller	***what***	***callee***
`%o5`	sixth return value	`%i5`
...	...	...
`%o1`	second return value	`%i1`
`%o0`	first return value	`%i0`

Register Windows

- SPARC ***register windows***: each procedure gets 16 “new” registers
- The window “slides” at a call

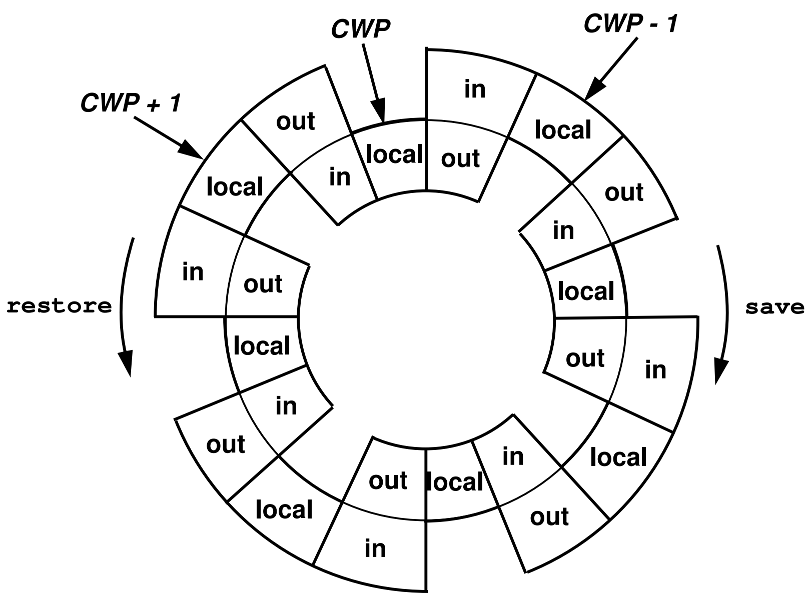
  callee’s in registers become synonymous with the caller’s out registers

- 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 <u>c</u>urrent <u>w</u>indow <u>p</u>ointer, ***CWP***

Window Management

- **`save`** instruction

  `save %sp,`*N*`,%sp` e.g., **`save %sp,-4*16,%sp`**

  slides the register window so the current window becomes the previous window

  decrements the current window pointer (***CWP***) and checks for window ***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 ***must*** be enough stack space

- **`restore`** instruction

  slides the register window so the previous window becomes the current window

  increments the current window pointer (***CWP***) and checks for window ***underflow***

- In **`save`** and **`restore`**

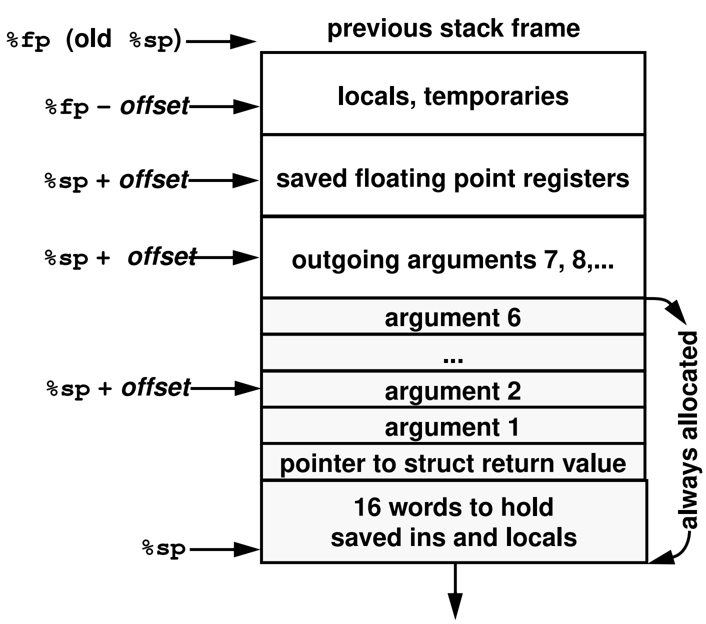
  source registers refer to the ***current*** window

  destination registers refer to the ***new*** window

Copyright ©1997

Stack Frame

- see §7.5 in Paul



Copyright ©1997

C Calling Convention

- First 6 arguments are passed in **%o0** — **%o5**, the rest in the stack

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

- Assembly language

```
.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,%o0
    set _str,%o1
    call _bcopy
    set 24,%o2
    ret; restore
.global _bcopy
_bcopy:...
    retl; nop
```


Example Stack Frames

```
main() {
    t(1,2,3,4,5,6,7,8);
}
```

```
_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(int a1, int a2,
  int a3, int a4,
  int a5, int a6,
  int a7, int a8) {
    int b1 = a1;
    return s(b1, a8);
}
```

```
_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(int c1, int c2) {
    return c1 + c2;
}
```

```
_s:
    add %o0,%o1,%o0
    retl; nop
```

Example Stack Frames, cont'd