Procedure Call

CS 217

Procedure Call

- Involves following actions
  - pass arguments
  - save a return address
  - transfer control to callee
  - transfer control back to caller
  - return results

```
int add(int a, int b)
{
  return a + b;
}


int main()
{
  int c = add(3, 4);
  printf("%d\n", c);
  return 0;
}
```

Procedure Call

- Involves following actions
  - pass arguments
  - save a return address
  - transfer control to callee
  - transfer control back to caller
  - return results

```
int add(int a, int b)
{
  return a + b;
}

int main()
{
  int c = add(3, 4);
  printf("%d\n", c);
  return 0;
}
```

Jump Instruction

`jmpl` *`address, reg`*

10	reg	111000	rs1	0	0	rs2
10	reg	111000	rs1	1	simm13	

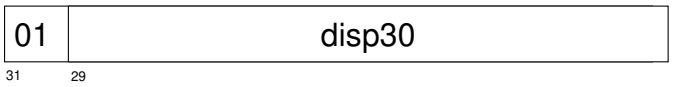
31 29 24 18 13 12 4

leaves PC in *reg*

```
reg = PC; /* return address */
PC = nPC;
nPC = rs1 + op2;
```

Call Instruction

`call label`



leaves PC (location of **`call`**) in **`%o7 (%r15)`**

```
%o7 = PC; /* return address */
PC = nPC;
nPC = PC + sign_extend(disp30) << 2;
```

Like: **`jmpl label, %o7`**

Procedure Call Example

- Simplest example: leaf procedure (**`c=a+b`**)

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

Return Instruction

`jmpl %o7+8,%g0`

- transfers control from caller to callee
- synthetic instructions: **`ret`** and **`retl`**
- why **`+8`**?

```
_add: save %sp, ..., %sp
      add %o0, %o1, %o1
      ret
      restore
```

Calls with Function Pointers

`jmpl reg,%o7`

- jumps to the 32-bit address specified in *reg*
- leaves PC (return address) in **`%o7 (%r15)`**
- example: **`c = (*apply)(a,b);`**

```
ld   b,%o0
ld   c,%o1
ld   apply,%o3
jmpl %o3,%o7; nop
st   %o0,a
```

Procedure Call

- Involves following actions
  - pass arguments
  - save a return address
  - transfer control to callee
  - transfer control back to caller
  - return results

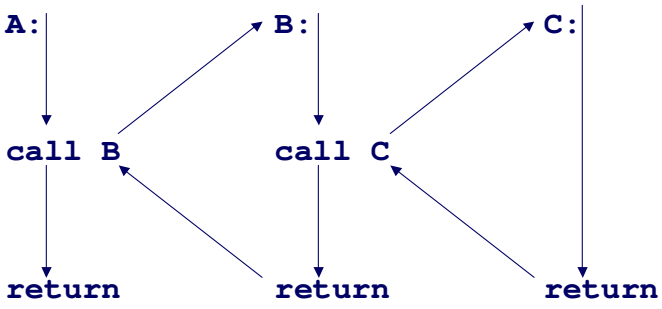
```
int add(int a, int b)
{
  return a + b;
}


int main()
{
  int c = add(3, 4);
  printf("%d\n", c);
  return 0;
}
```

Procedure Call

- Requirements
  - Pass a variable number of arguments
  - Pass and return structures
  - Allocate and deallocate space for local variables
  - Save and restore caller's registers
  - Handle nested procedure calls
- Entry and exit sequences collaborate to implement these requirements

Nested/Recursive Calls

- **A** calls **B**, which calls **C**



Must work when **B** is **A**

Arguments and Return Values

- By convention
  - caller places arguments in the “out” registers
  - callee finds its arguments in the “in” registers
  - only the first 6 arguments are passed in registers
  - the rest are passed on the stack

Arguments and Return Value (cont)

- Registers at call time

caller	callee	
%o7	%i7	**return address -8 (%r15)**
%o6	%i6	**stack/frame pointer (%r14)**
%o5	%i5	**sixth argument**
…	…	
%o0	%i0	**first argument**

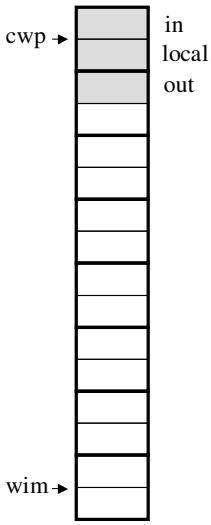
Arguments and Return Value (cont)

- Registers at return time

caller	callee	
%o5	%i5	**sixth return value**
%o4	%i4	**fifth return value**
…	…	
%o0	%i0	**first return value**

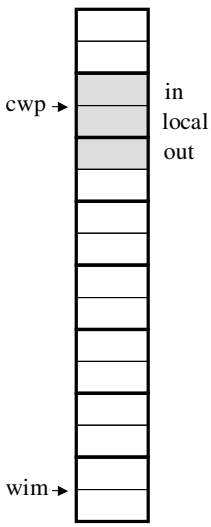
Register Windows

- Machine has more than 32 registers
  - Each procedure gets 16 “new” registers
  - All procedures can use globals
- The window “slides” at call time
  - caller’s out registers become callee’s in registers
- Instructions
  - **`save`** slides the window forward
  - **`restore`** slides the window backwards
  - decrement/increments CWP register
- Finite number of windows (usually 8)



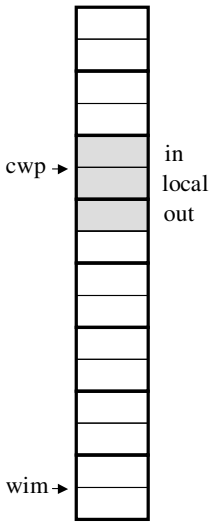
Register Windows

- Machine has more than 32 registers
  - Each procedure gets 16 “new” registers
  - All procedures can use globals
- The window “slides” at call time
  - caller’s out registers become callee’s in registers
- Instructions
  - **`save`** slides the window forward
  - **`restore`** slides the window backwards
  - decrement/increments CWP register
- Finite number of windows (usually 8)



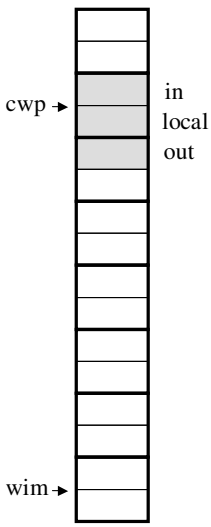
Register Windows

- Machine has more than 32 registers
  - Each procedure gets 16 “new” registers
  - All procedures can use globals
- The window “slides” at call time
  - caller’s out registers become callee’s in registers
- Instructions
  - **`save`** slides the window forward
  - **`restore`** slides the window backwards
  - decrement/increments CWP register
- Finite number of windows (usually 8)



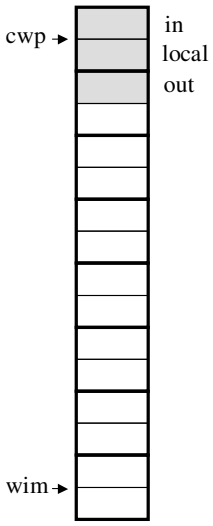
Register Windows

- Machine has more than 32 registers
  - Each procedure gets 16 “new” registers
  - All procedures can use globals
- The window “slides” at call time
  - caller’s out registers become callee’s in registers
- Instructions
  - **`save`** slides the window forward
  - **`restore`** slides the window backwards
  - decrement/increments CWP register
- Finite number of windows (usually 8)

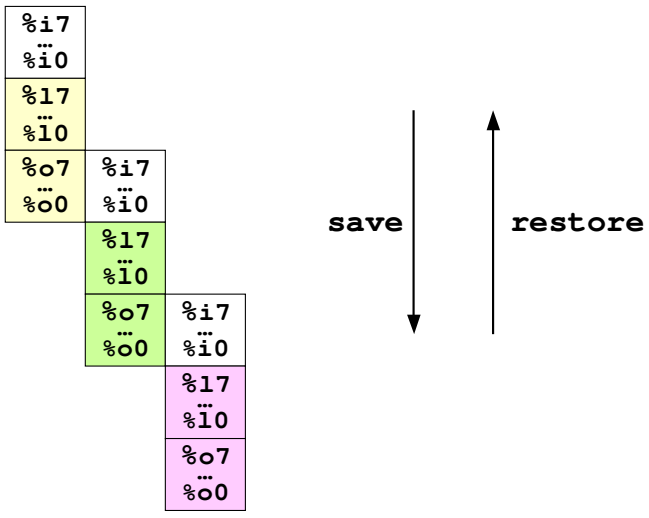


Register Windows

- Machine has more than 32 registers
  - Each procedure gets 16 “new” registers
  - All procedures can use globals
- The window “slides” at call time
  - caller’s out registers become callee’s in registers
- Instructions
  - **`save`** slides the window forward
  - **`restore`** slides the window backwards
  - decrement/increments CWP register
- Finite number of windows (usually 8)



Register Windows (cont)



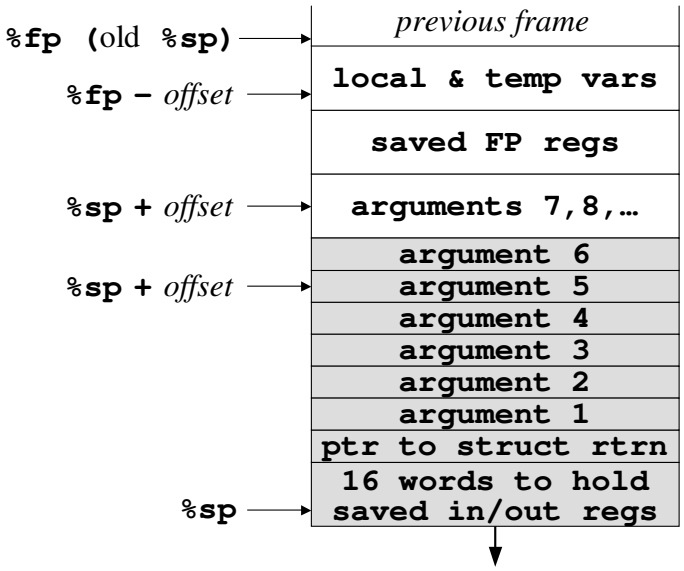
Window Managment

- Call time (**`save`**)
  - **`save %sp,N,%sp`**
  - current window becomes previous window
  - decrements CWP and checks for overflow
  - adds *N* to the stack pointer (allocates *N* bytes if *N*<0)
  - if overflow occurs, save registers on the stack (must be enough stack space)
- Return time (**`restore`**)
  - previous window becomes current window
  - increments CWP and checks for underflow

Stack

- Procedure call information stored on stack
  - locals, including compiler temporaries
  - caller's registers, if necessary
  - callee's arguments, if necessary
- Sparc's stack grows "down" from high to low address
- The stack pointer (**`%sp`**) points to top word on the stack (must be multiple of 8)

Stack Frame



Example Stack Frames

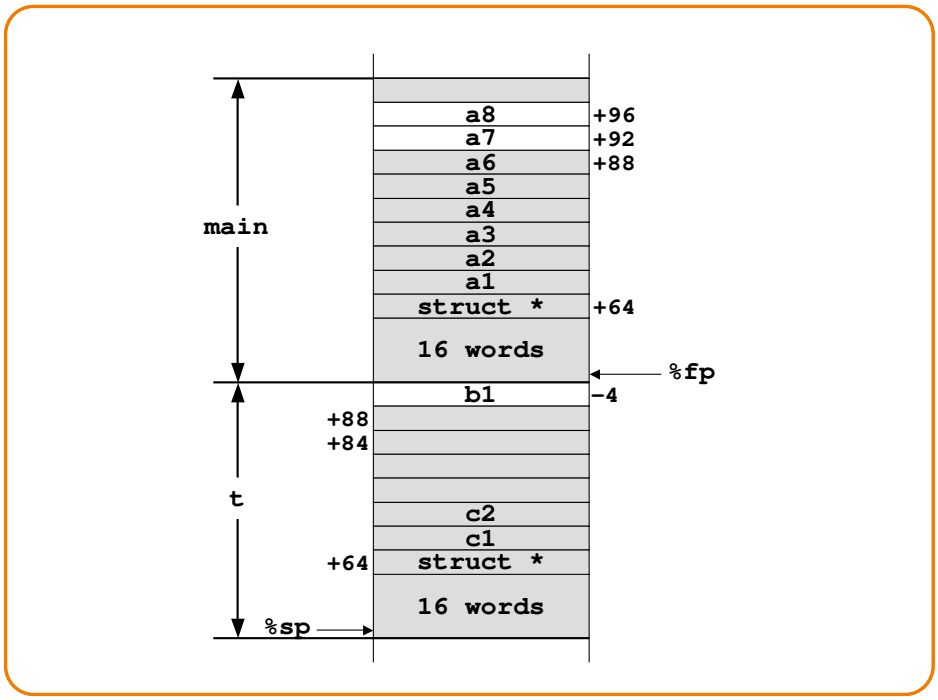
```
main() {
   t(1,2,3,4,5,6,7,8);
}

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;
}
```

Example (cont)

```
_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
```

Example (cont)

```
_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,%01,%o0
    retl; nop
```


main
t
a8
a7
a6
a5
a4
a3
a2
a1
struct *
16 words
b1
c2
c1
struct *
16 words
+96
+92
+88
+64
%fp
-4
+88
+84
+64
%sp