Subroutine Calls

CS 217

Subroutine Calls

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

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


int main()
{
  int d = add(3, 4, 5);
  printf("%d\n", d);
  return 0;
}
```

Subroutine Calls

- Requirements
  - Set PC to arbitrary address
  - Return PC to instruction after call sequence
  - Handle nested subroutine calls
  - Save and restore caller's registers
  - Pass an arbitrary number of arguments
  - Pass and return structures
  - Allocate and deallocate space for local variables
- Subroutine call and return sequences collaborate to implement these requirements

Subroutine Calls

- Requirements
  - Ø Set PC to arbitrary address
  - Ø Return PC to instruction after call sequence
  - Handle nested subroutine calls
  - Save and restore caller's registers
  - Pass an arbitrary number of arguments
  - Pass and return structures
  - Allocate and deallocate space for local variables
- Subroutine call and return sequences collaborate to implement these requirements

Call Instruction

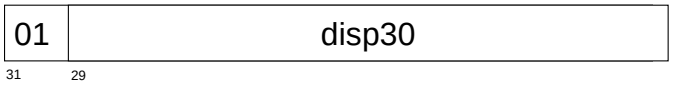
`call` *`label`*

Example:

```
ld   a,%o0
ld   b,%o1
ld   c,%o2
call addthree
nop
st   %o0,d
```

Call Instruction

`call` *`label`*



Jumps to address of label and
leaves PC (location of **`call`**) in **`%o7`** (**`%r15`**)

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

Example:

```
ld   a,%o0
ld   b,%o1
ld   c,%o2
call addthree
nop
st   %o0,d
```

Calls with Function Pointers

`jmpl reg,%o7`

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

```
ld   a,%o0
ld   b,%o1
ld   c,%o2
ld   apply,%o3
jmpl %o3,%o7; nop
st   %o0,d
```

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;
```

Return Instructions

`ret and retl`

- returns to caller at end of subroutine
- **`retl`** is synthetic instructions for **`jmpl %o7+8, %g0`**

Example:

```
                ld   a,%o0
                ld   b,%o1
                ld   c,%o2
                call addthree
                nop
                st   %o0,d

addthree:       add %o0,%o1,%o0
                add %o0,%o2,%o0
                retl
```

This is a "leaf" subroutine

Leaf Subroutines

- Simply use caller's registers
  - Do not save/restore
  - Use only %o0-%o5, %g0-%g1
- Advantages
  - Simple
  - Little overhead

```
                ld   a,%o0
                ld   b,%o1
                ld   c,%o2
                call addthree
                nop
                st   %o0,d

addthree:       add %o0,%o1,%o0
                add %o0,%o2,%o0
                retl
```

Leaf Subroutines

- Limitations
  - Cannot take more than 6 parameters
  - Cannot have arbitrary number of local variables
  - Cannot call another function
  - Cannot return structure

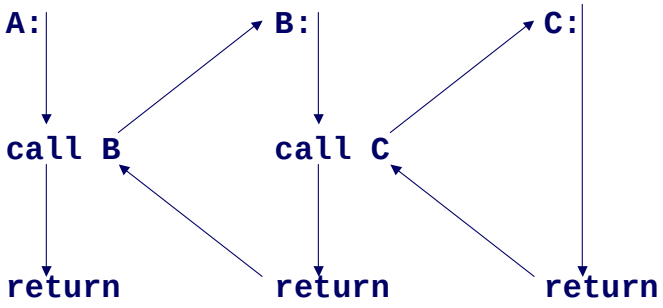
```
main() {
   t(1,2,3,4,5,6,7,8);
}

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

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

Nested Subroutine Calls

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



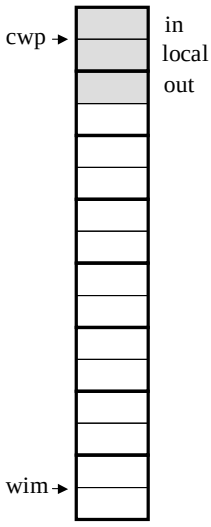
Must even work when **B** is **A**

Subroutine Calls

- Requirements
  - Set PC to arbitrary address
  - Return PC to instruction after call sequence
  - Ø Handle nested subroutine calls
  - Ø Save and restore caller's registers
  - Pass an arbitrary number of arguments
  - Pass and return structures
  - Allocate and deallocate space for local variables
- Subroutine call and return sequences collaborate to implement these requirements

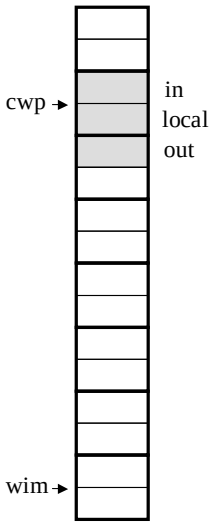
Register Windows

- Machine has more than 32 registers
  - Each subroutine gets 16 "new" registers
  - All subroutines 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)



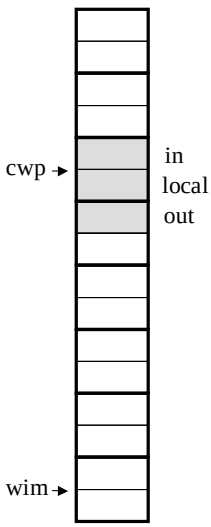
Register Windows

- Machine has more than 32 registers
  - Each subroutine gets 16 “new” registers
  - All subroutines 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)



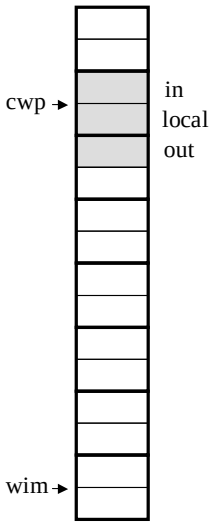
Register Windows

- Machine has more than 32 registers
  - Each subroutine gets 16 “new” registers
  - All subroutines 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)



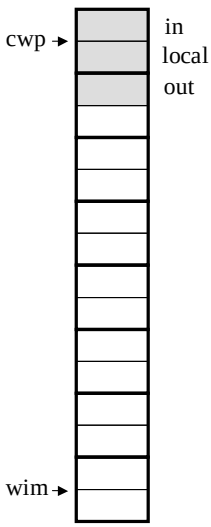
Register Windows

- Machine has more than 32 registers
  - Each subroutine gets 16 “new” registers
  - All subroutines 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)

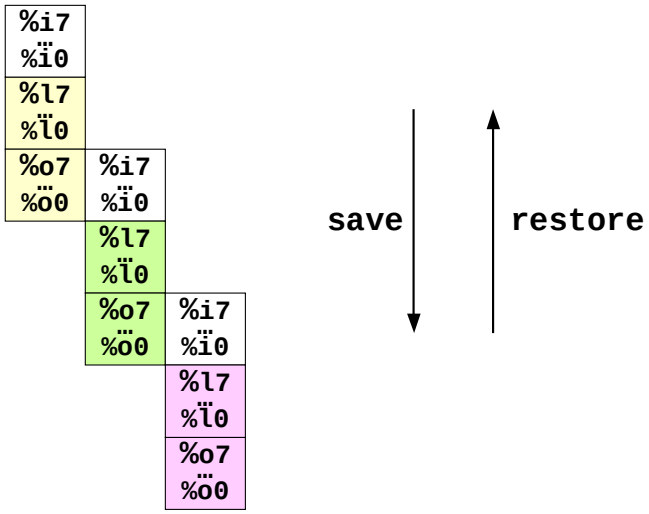


Register Windows

- Machine has more than 32 registers
  - Each subroutine gets 16 “new” registers
  - All subroutines 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)



Save and Restore Commands

- Call time (save %sp, N, %sp)
  - decrements CWP
    - » current register window becomes previous window
    - » if overflow occurs, save registers on the stack
  - adds N to the stack pointer
    - » frame pointer becomes previous stack pointer
- Return time (restore)
  - increments CWP
    - » previous window becomes current window
  - subtracts N from the stack pointer
    - » restores previous stack pointer and frame pointer

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

```
                    ld   a,%o0
                    ld   b,%o1
                    ld   c,%o2
                    call addthree
                    nop
                    st   %o0,d

addthree:  save %sp, -64, %sp
           add %i0,%i1,%i0
           ret
           restore
```

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**

- Registers at return time

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

Comparison of Subroutine Types

- Example:

```
        ld   a,%o0
        ld   b,%o1
        ld   c,%o2
        call add
        nop
        st   %o0,c

add:    add %o0,%o1,%o0
        add %o0,%o2,%o0
        retl
```

Leaf Subroutine

```
        ld   a,%o0
        ld   b,%o1
        ld   c,%o2
        call add
        nop
        st   %o0,c

add:    save %sp, -64, %sp
        add %i0,%i1,%i0
        add %i0,%i2,%i0
        ret
        restore
```

"Regular" Subroutine

Subroutine Calls

- Requirements
  - Set PC to arbitrary address
  - Return PC to instruction after call sequence
  - Handle nested subroutine calls
  - Save and restore caller's registers

  Ø Pass an arbitrary number of arguments

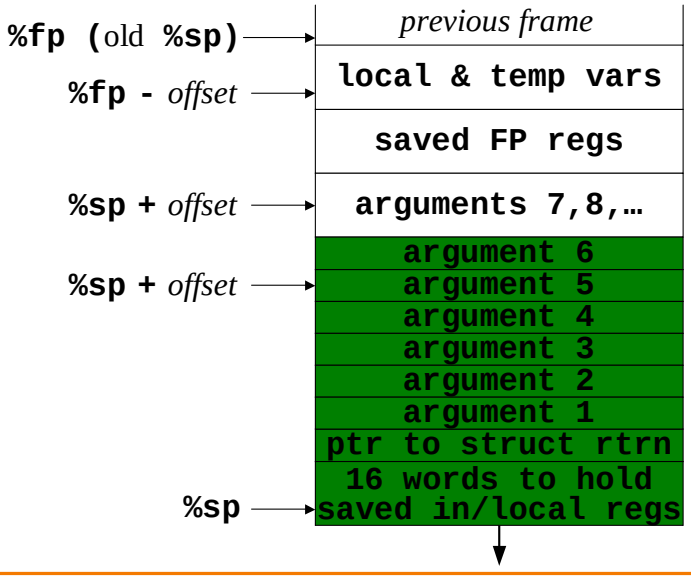
  Ø Pass and return structures

  Ø Allocate and deallocate space for local variables
- Subroutine call and return sequences collaborate to implement these requirements

Stack

- Subroutine call information stored on stack
  - callee's arguments, if necessary
  - local variables, if necessary
  - caller's registers, if necessary
- Information added to stack before call and removed from stack upon return
  - The stack pointer (**%sp**) points to top word on the stack (must be multiple of 8)
  - The frame pointer (**%fp**) points to the position of the stack pointer in the caller's frame (before the **save** instruction)

Stack Frame



Example Stack Frames

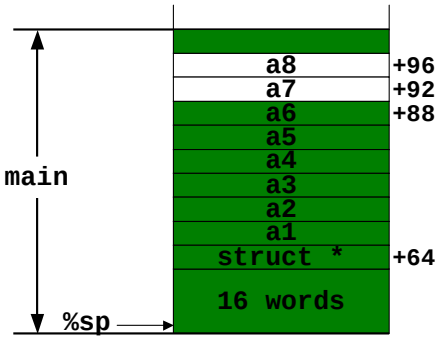
```
main() {
   t(1,2,3,4,5,6,7,8);
}

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)

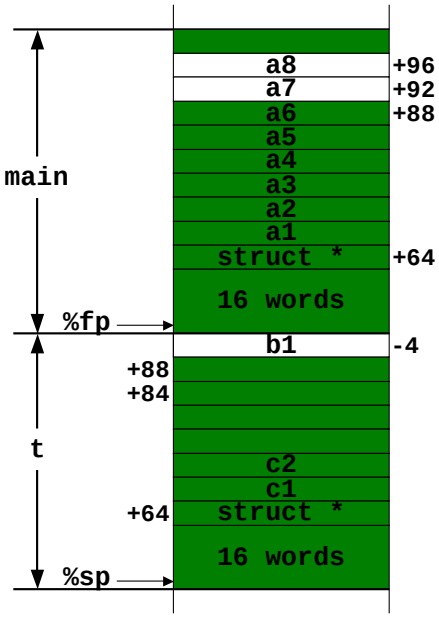
```
_main: save %sp,-104,%sp
       set 1,%o0
       set 2,%o1
       set 3,%o2
       set 4,%o3
       set 5,%o4
       set 6,%05
       set 7,%i5
       st %i5,[%sp+4*6+68]
       set 8,%i5
       st %i5,[%sp+4*7+68]
       call _t; nop
       ret; restore
```



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
```



Summary

- Caller
  - Loads first six arguments into %o0, %o1, …
  - Loads other arguments onto stack (%sp + 68 + 4*i)
  - Uses call or jmpl to set PC (saves old PC in %o7)
  - Receives return values in %o0, %o1, …
- Callee
  - Executes save instruction at beginning
  - Accesses first six arguments in registers
  - Accesses other arguments on stack (%fp + 68 + 4*i)
  - Accesses local variables on stack (%fp – 4*i)
  - Uses **`ret`** to return (%jmpl %o7+8, %g0)
  - Executes restore instruction at end