Subroutine/function

Introduction

- open subroutine
  - ✓ replace/extend text
  - ✓ macro
  - ✓ no run-time overhead w.r.t registers
- **closed** subroutine
  - ✓ call/return
  - ✓ context switching (register saving)
  - ✓ (de)allocation of stack frame
  - ✓ parameter passing: register or stack
  - ✓ return value/address

```
      call  subr
      ! delay slot
      ! return address
          ...
subr: save %sp, ..., %sp
          ...

      ret
      restore  ! delay slot
```

Subroutine call

1) call label

- ✓ transfer control to label (%pc update)
- ✓ store %pc(before update) to %o7 as return address

2) jmpl R+A, S

- ✓ transfer control to R+A (R: register, A: register/imm)
- ✓ store %pc to register S

Ex) jmpl %o0, %o7

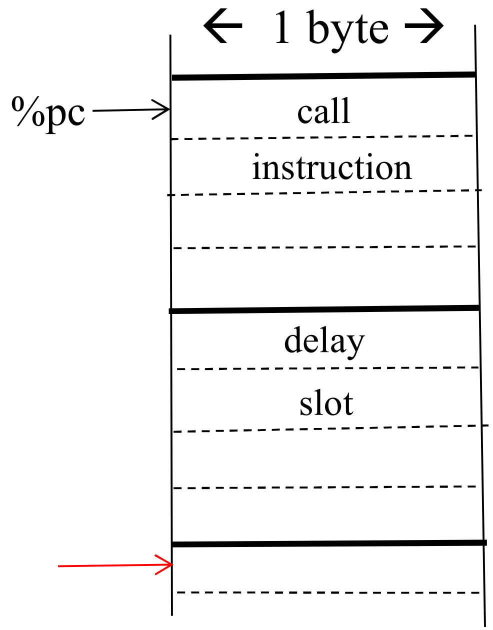
→ %o7 ← %pc
%pc ← %o0

(Be aware of delay slots!)

- Return instructions
  - ✓ delay slot

1) ret
2) jmpl %i7 + 8, %g0
3) jmpl %S+8, %g0
4) retl (return from leaf subroutine)

(%pc) ← (%i7 + 8)



Subroutine call/return example

```
        .global     main
main:   save   %sp, -96, %sp
        mov    2, %o0
        mov    3, %o1
        call   add2
        nop
        ret
        restore

add2:   save   %sp, -96, %sp
        add    %i0, %i1, %i0
        ret
        restore
```

< Instruction execution sequence >

1. call instruction

   %o7 ← %pc

   %pc ← address ‘add2'

2. delay slot instruction
3. first instruction in subroutine
4. ret (%pc ← %i7 + 8)
5. delay slot instruction
6. instruction in main routine

call instruction (machine) format

Bit number	31 30	29 0
field	op	displacement

- ✓ op = 01, displacement= address (**pc-relative**)
- ✓ Ex

```
        call  add4
        mov     5, %o1
        ret
        restore
add4: save %sp, -96, %sp
```

+ 16

→ 01 000000000000000000000000000100

Stack frame structure

%sp + n area	%sp →	register window saving area (64B)	callee
		Return Structure pointer (4B)	
		First 6 parameters (24B)	
		rest of parameters (as needed)	
%fp - n area		**locals** (as needed)	
%fp + n area	%fp →	register window saving area (64B)	caller
		Return Structure pointer(4B)	
		First 6 parameters (24B)	
		rest of parameters (as needed)	
		locals (as needed)	

- save %sp, -**96**, %sp: stack frame and register set allocation

Register set

- 32 registers through mapping

Group	name/mnemonics	function	mapping
global	%r0 - %r7(%g?)	global register	No
out	%r8 - %r15(%o?)	outgoing params	Yes
local	%r16 - %r23(%l?)	local vars	Yes
in	%r24 - %r31(%i?)	incoming params	Yes

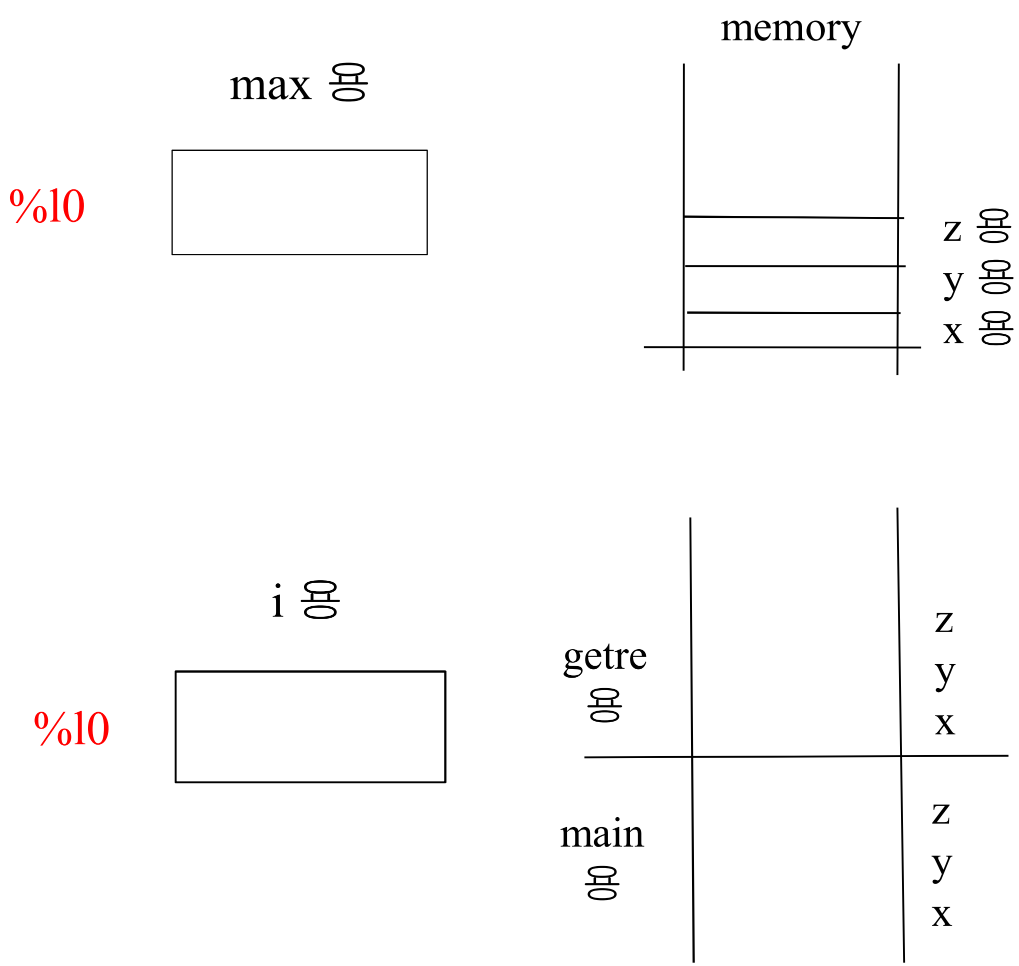
- A typical SPARC processor has 128 registers for mapping

  = 16 registers per set * 8 sets

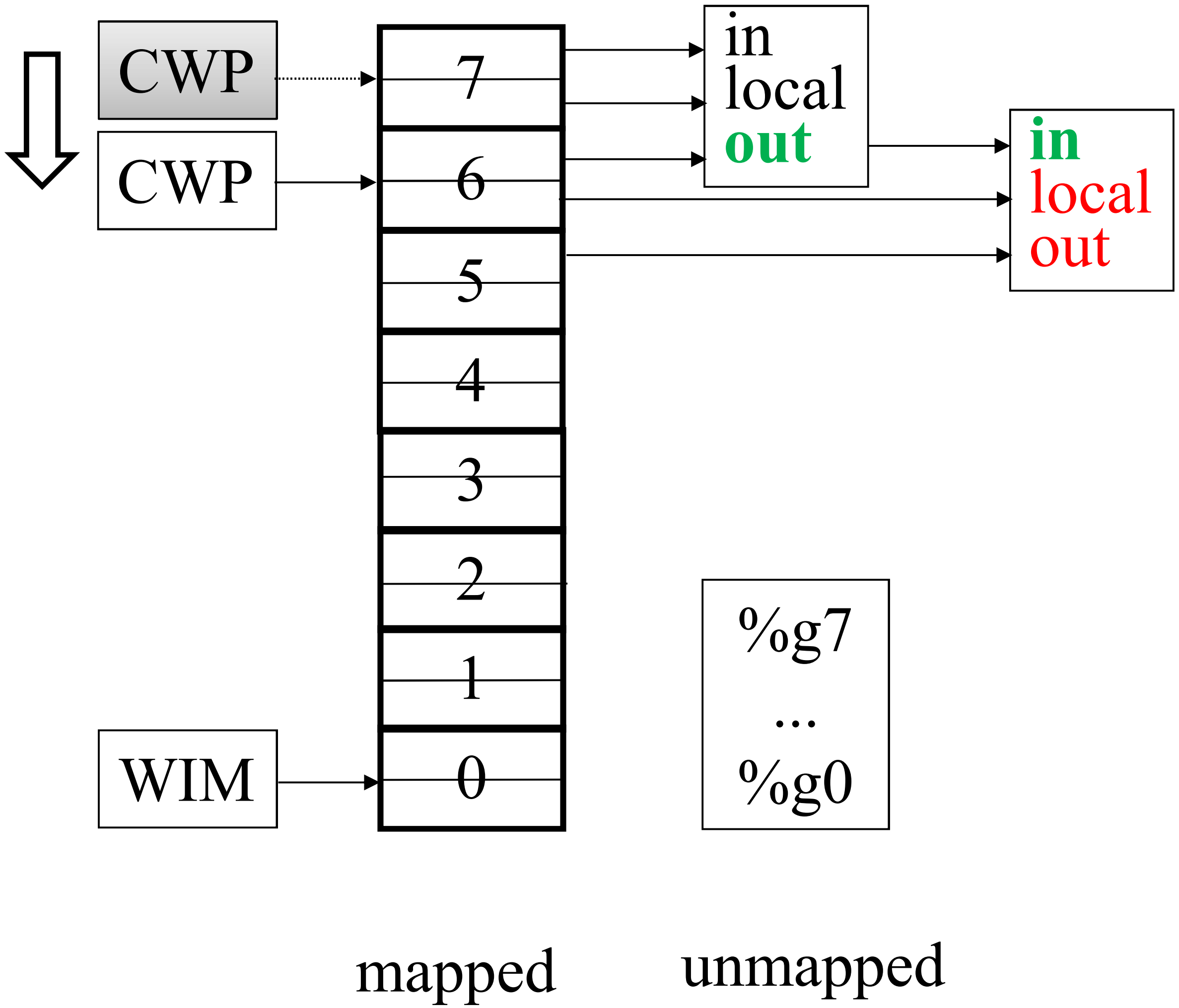
- SPARC allows calling subroutine without register saving
  - ✓ register saving: execution time overhead
- Related instruction: **save**/**restore**

Register saving

```
main()
{
  int x, y, z;
  register int max;
      :
  result=getre(x,y)
      :
}
getre(x,y)
{
int x, y, z;
register int i;
:
}
```



Register file structure

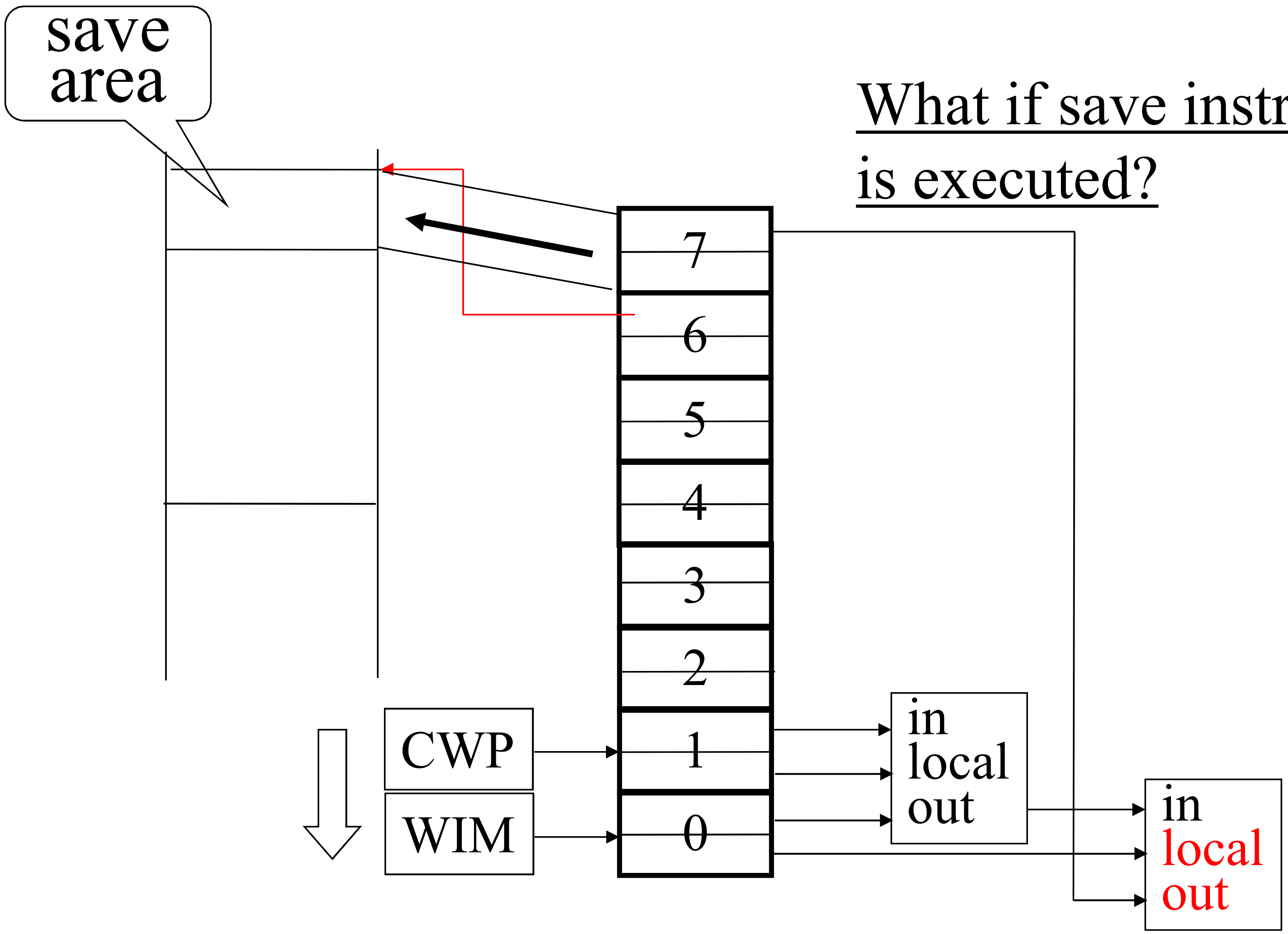


Register window

- CWP(current window pointer)
  - ✓ pointing current active register set
- WIM(window invalid mask)
  - ✓ pointing the last available register set
- Effect of save instruction
  - register set allocation
  - out registers of caller is the same as in registers of callee
    - ➢ overlapped register window
    - ➢ %sp (%o6), %fp(%i6)
- Effect of restore instruction
- Register windows overflow / underflow

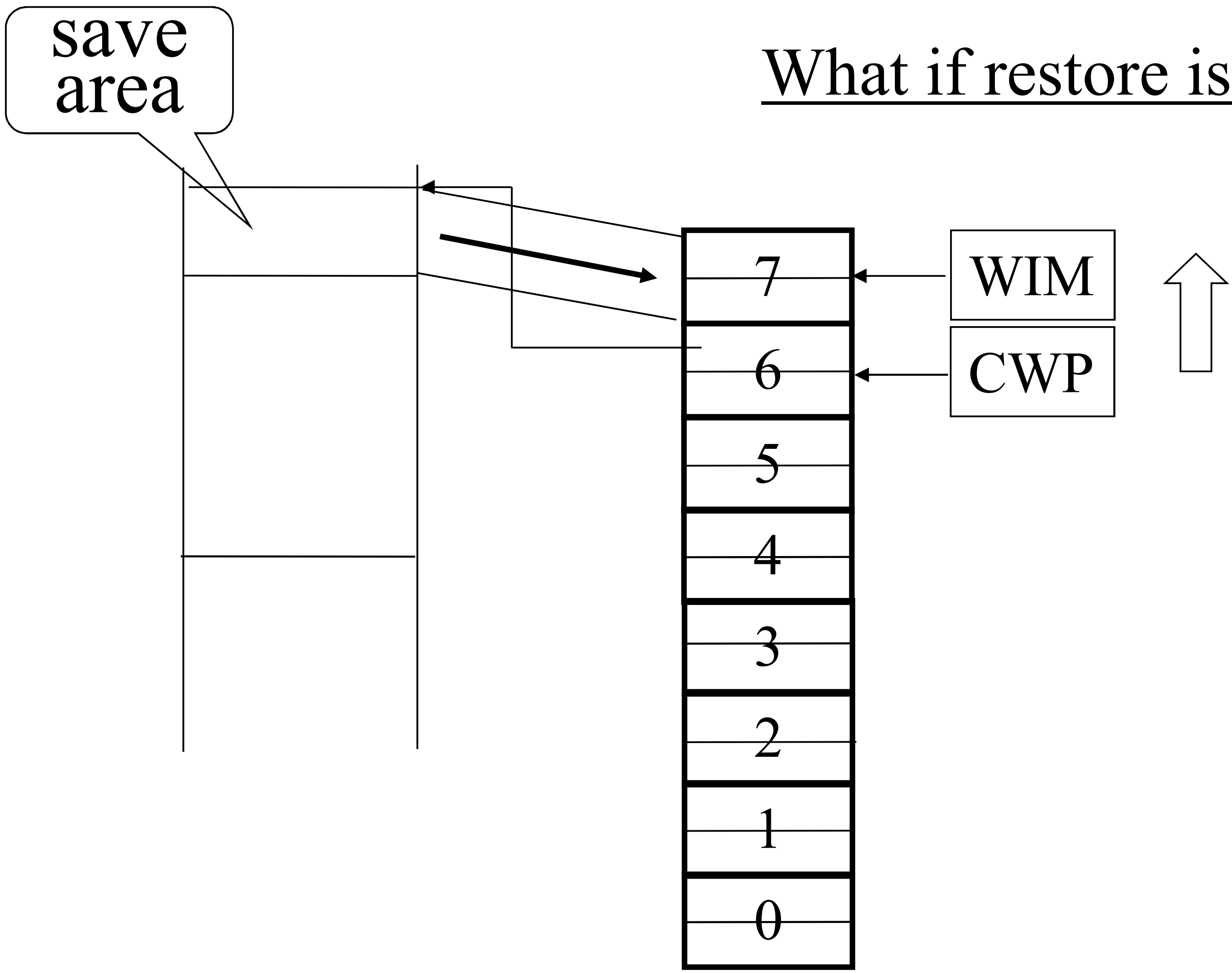
Register window overflow

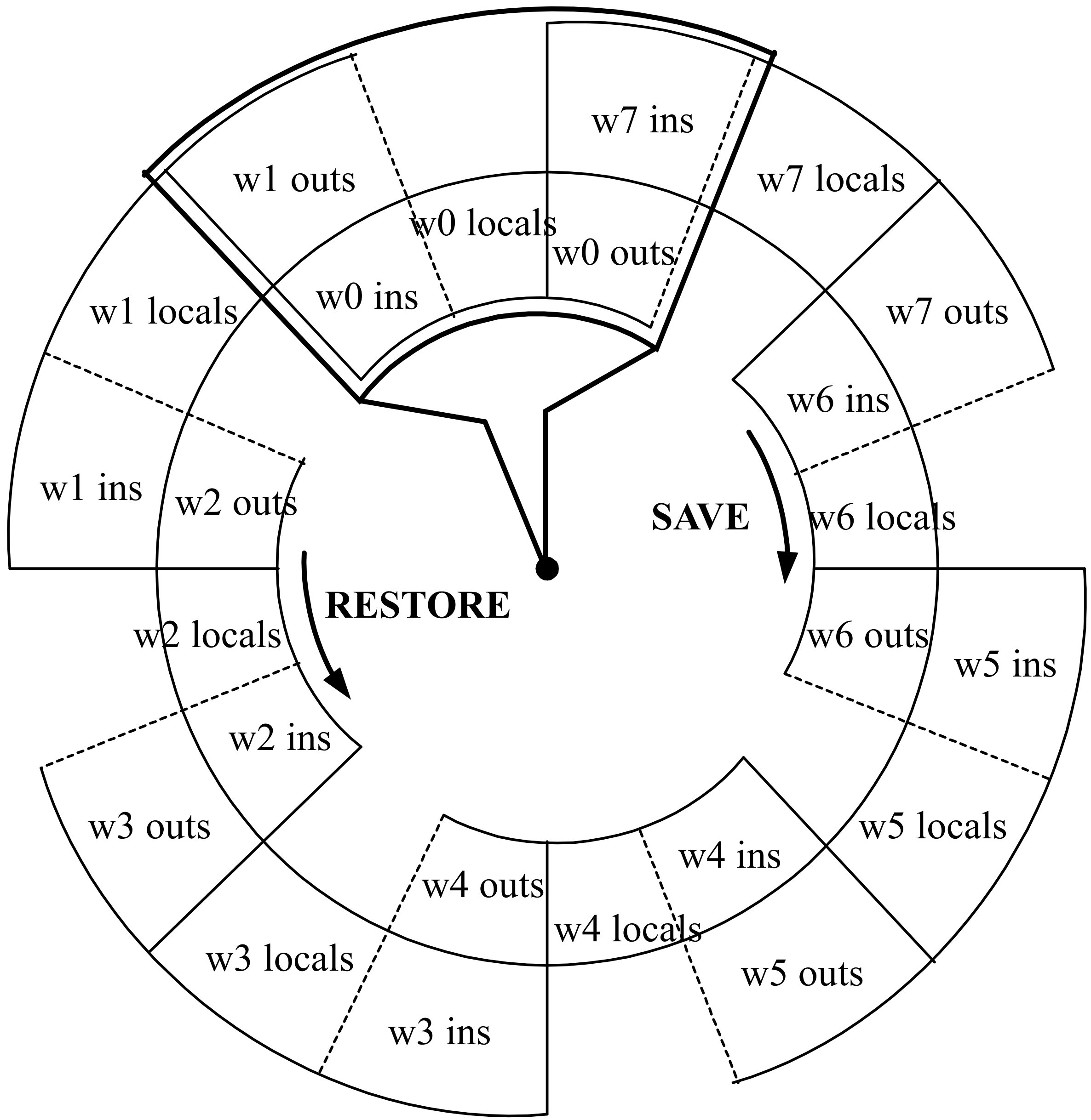
What if save instruction is executed?



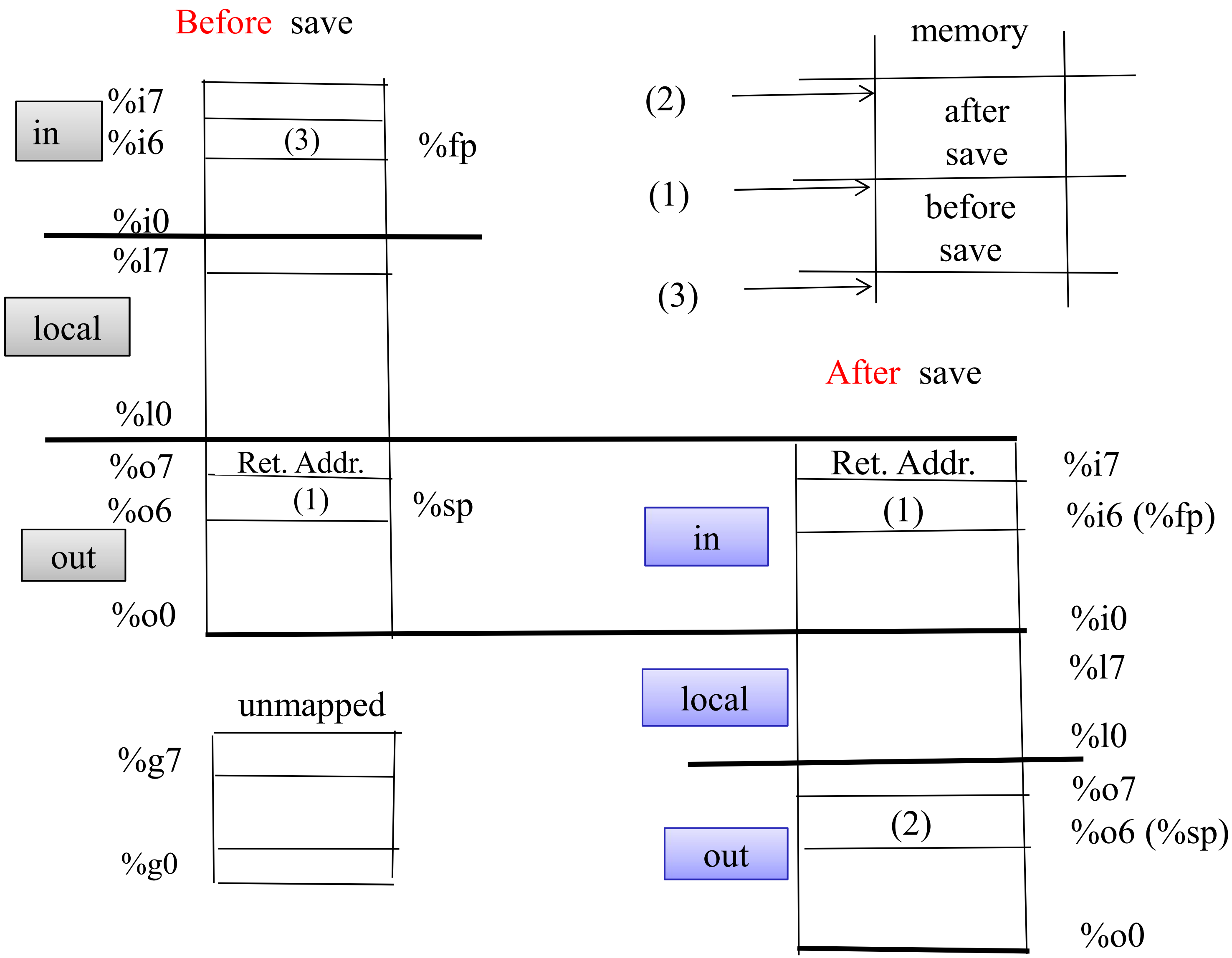
Register window underflow

What if restore is executed?




w7 ins
w1 outs
w7 locals
w0 locals
w0 outs
w0 ins
w1 locals
w7 outs
w6 ins
w1 ins
w2 outs
SAVE
w6 locals
RESTORE
w2 locals
w6 outs
w5 ins
w2 ins
w3 outs
w5 locals
w4 ins
w4 outs
w4 locals
w3 locals
w5 outs
w3 ins

Overlapped register window

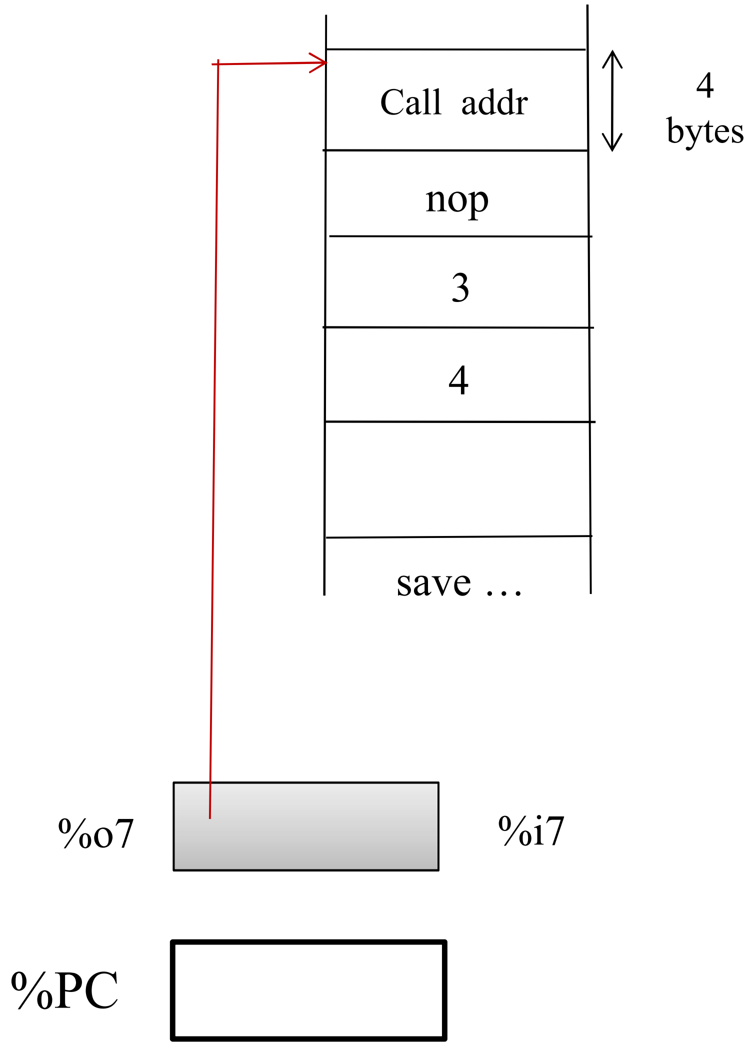


Arguments passing

1) in-line method

✓ Embedded in code

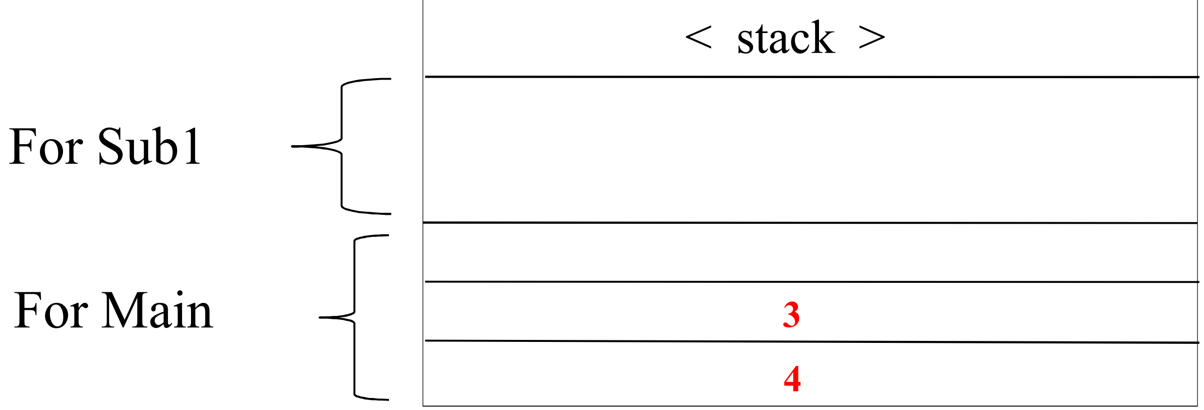
```
        call   addr
        nop
        .word  3, 4
        ...
addr:   save   %sp, -96, %sp
        ld     [%i7 + 8], %i0
        ld     [%i7 + 12], %i1
        add    %i0, %i1, %i0
        jmpl   %i7 + 16, %g0
        restore
```



2) Using stack

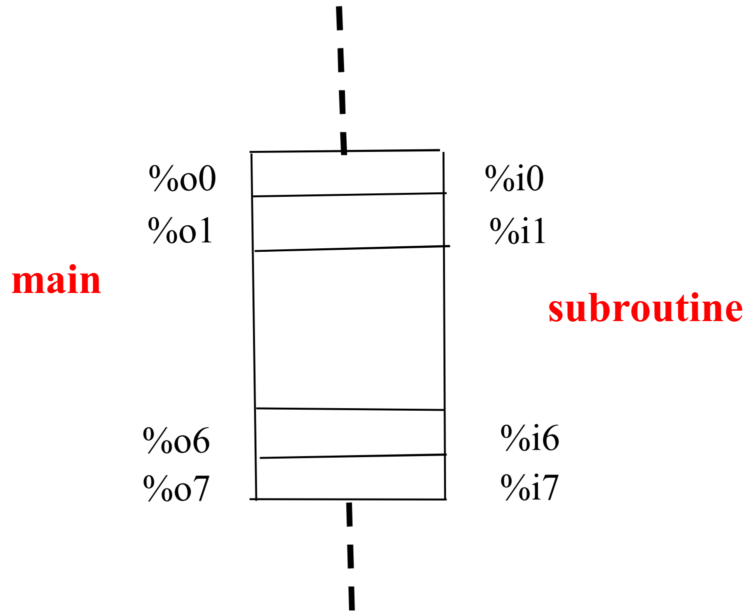
- ✓ Excessive memory access
- ✓ Most widely used method

```
main:
        store in < stack >
        call    sub1
         nop
         ...
sub1:   save %sp, ..., %sp
        read from < stack >
         ...
        ret
        restore
```



3) Using registers: SPARC case

- ✓ use out registers
- ✓ maximum number: 6
  - ✓ %o6: %sp
  - ✓ %o7: return address



- ✓ arguments more than 6 are stored in stack

- Example 1

```
mov   3,  %o0
mov   5,  %o1
call  .mul
nop
```

- Example 2

```
int main(){
    int sum;
    sum = add4(1, 2, 3, 4);
    }

int add4(int a, int b, int c, int d){
    return a + b + c + d;
}
```

```
        .global     main
main:   save   %sp, -96, %sp
        mov    1, %o0
        mov    2, %o1
        mov    3, %o2
        call   add4
        mov    4, %o3
        ret
        restore

        .global     add4
add4:   save   %sp, -96, %sp
        add    %i0, %i1, %i0
        add    %i2, %i0, %i0
        ret
        restore     %i3, %i0, %o0
```

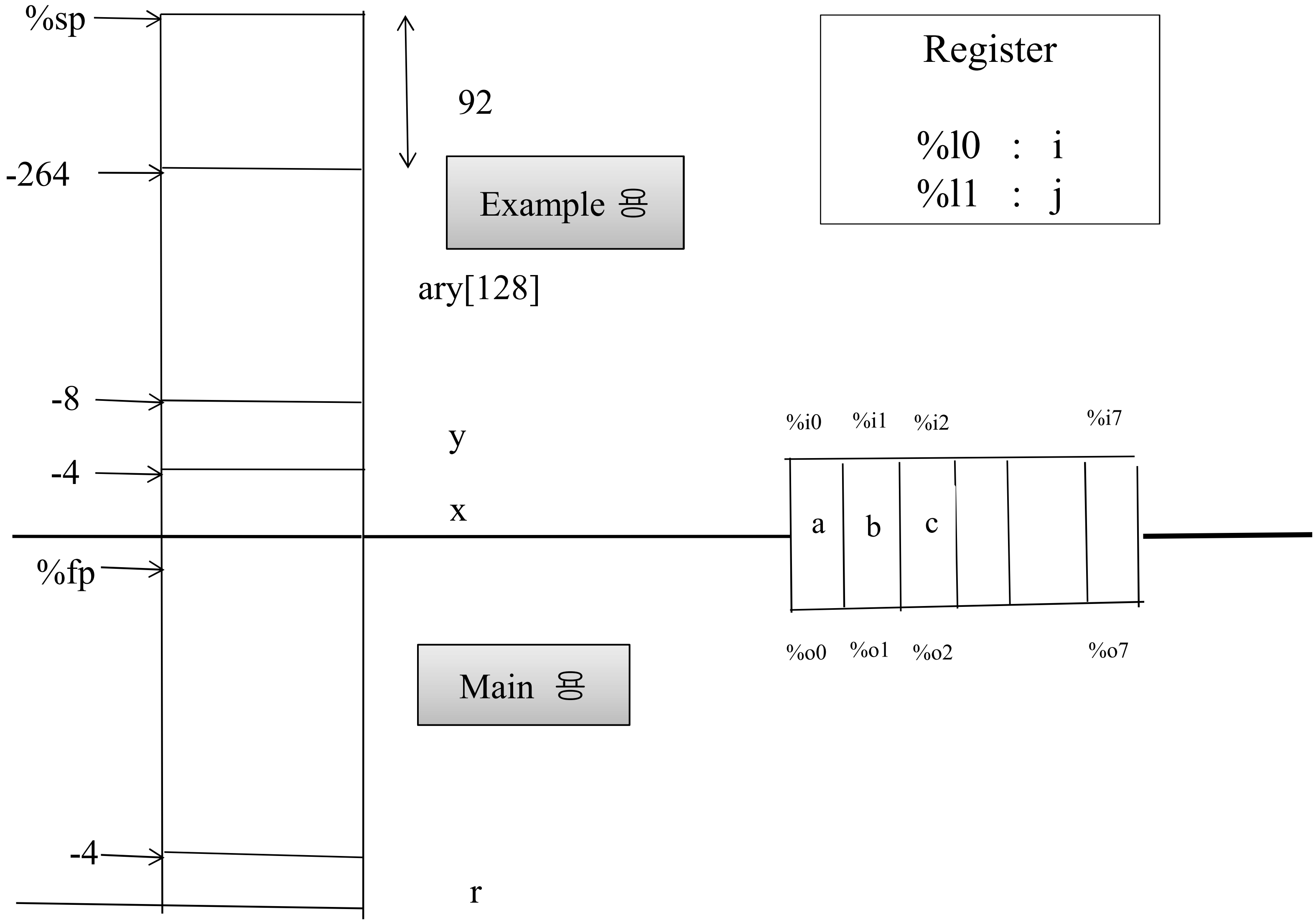
```
add    %i3,  %i0, %i0
ret
restore
```

Program Example

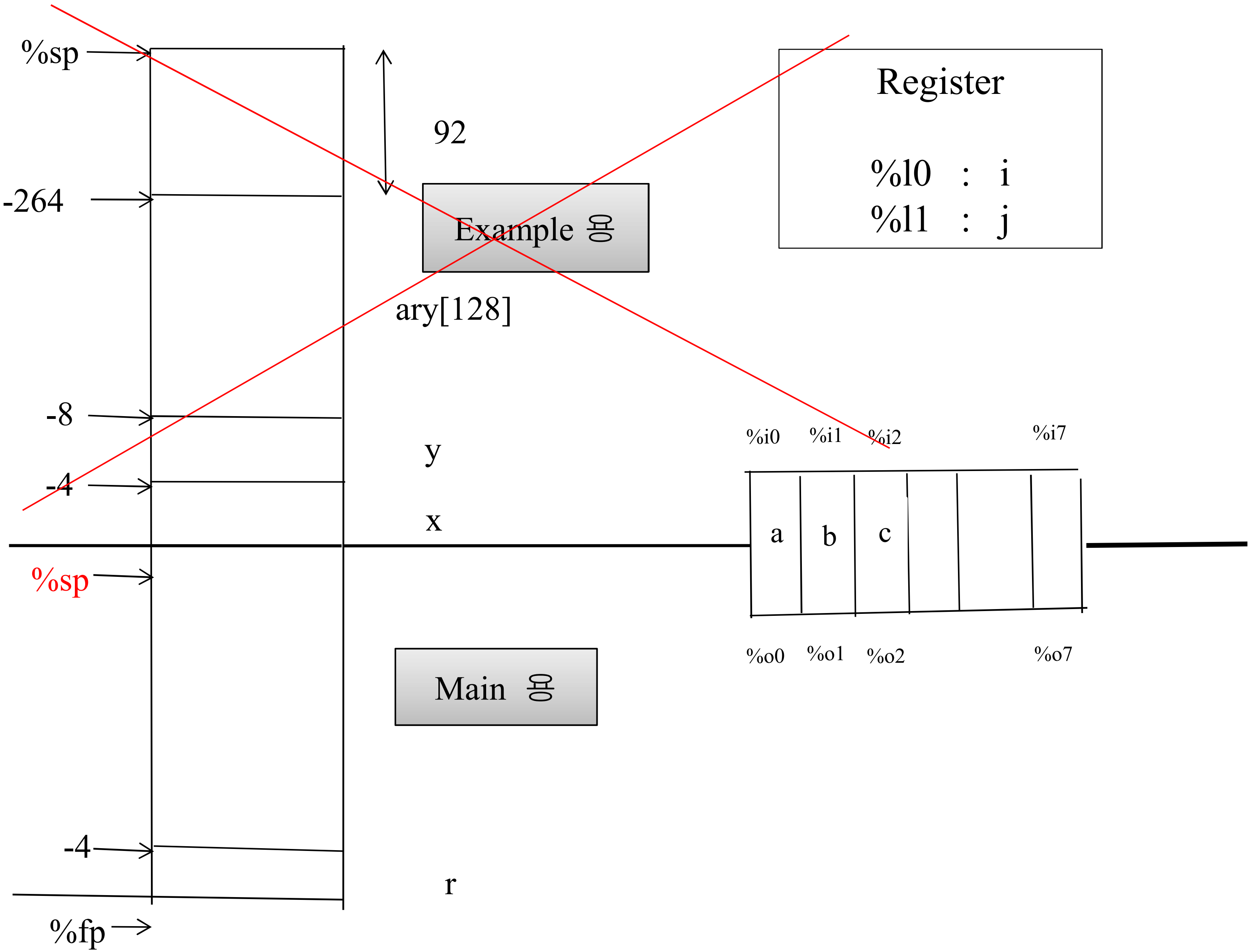
```
int example(int a, int b, char c) {
  int x, y;
  short ary[128];
  register int i, j;

  x = a + b;
  i = c + 64;
  ary[i] = c + a;
  y = x * a;
  j = x + i;
  return x + y;
}
```

```
main() {
   int r;
   r = example(3, 5, 4);
}
```

- Variables in stack when subroutine example is executed



- When control is returned to main



```
! arguments
        ! a_r in %i0
        ! b_r in %i1
        ! c_r in %i2
! local variables
    x_s = -4
    y_s = -8
    ary_s = -264
! register variables
        ! i_r in %l0
        ! j_r in %l1

    .global example

example: save %sp, -360, %sp   ! –(64+4+24+264) & -8
         add  %i0, %i1, %o0        !   x = a + b
         st   %o0, [%fp + x_s]
```

```
add      %i2, 64, %l0            ! i = c + 64
add      %i0, %i2, %o0           !  c + a
sll      %l0, 1, %o1             ! i*2
add      %fp, ary_s, %o2
sth      %o0, [%o1 + %o2]        ! store in ary[i]
ld       [%fp + x_s], %o0        ! y = x * a
call     .mul
mov      %i0, %o1
st       %o0, [%fp + y_s]
ld       [%fp + x_s], %o0        ! j = x + i
add      %l0, %o0, %l1
ld       [%fp + x_s], %o0        ! return x + y
ld       [%fp + y_s], %o1
ret
restore  %o0, %o1, %o0           ! add %o0, %o1, %o0
```

```
add %o0, %o1, %i0
ret
restore
```

```
        .global main
main:   save %sp, -(64 + 4 + 24 + 4) & -8, %sp

        mov     3, %o0
        mov     5, %o1
        mov     4, %o2

        call    example
        nop

        st %o0, [%fp - 4]

        mov     1, %g1
        ta      0
```