The Calling Sequence

Typical code to call a subroutine:

```
              <...move arguments into %o0,%o1, ..., %o5>
%o7   -->     call foo
%o7+4 -->     nop
%o7+8 -->     add   %o0,43,%l4
```

Return to here (result is in %o0)

The "call" instruction:

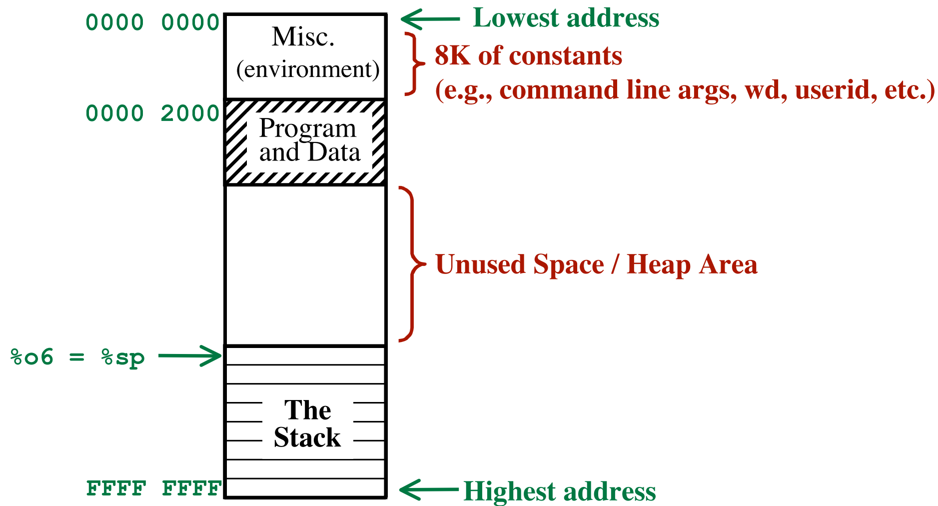
- Save address of the "call" instruction in %o7.
- Move address of "foo" into PC.
- Execute the instruction in the delay slot.
- Execute first instruction in "foo".

The "ret" instruction:

- Add 8 to %o7.
- Move result into PC.
- Execute the instruction after the delay slot.

© Harry H. Porter, 2006

Memory Usage In Unix



© Harry H. Porter, 2006

The In-Memory Stack

The stack grows "downward"
From high memory, towards low memory

The *Stack Top Pointer* (%sp) points to the lowest used byte.
i.e., the item on the "top" of the stack.

The Stack Top Pointer (%sp) must be double-word aligned
i.e., last 3 bits must be = 000

To ensure proper alignment:
`& 0xFFFFFFF8`

To grow the stack

```
        add     %sp,-93&-8,%sp
```

The minimum frame size:
64 bytes = 16 regs * 4 bytes
If on-chip registers run out...
The OS will save registers in this area.
The programmer / compiler must leave at least this much space.
Programmer / compiler may allocate additional bytes in frame

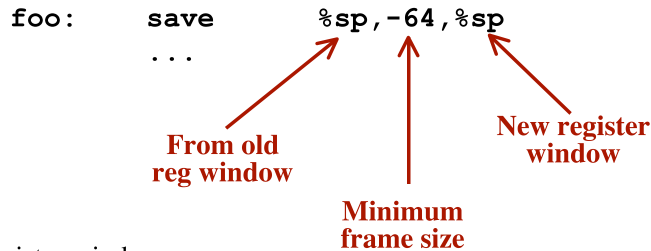
© Harry H. Porter, 2006

The "save" Instruction

Grow the in-memory stack
Grow the on-chip register stack
(allocate a new register window)

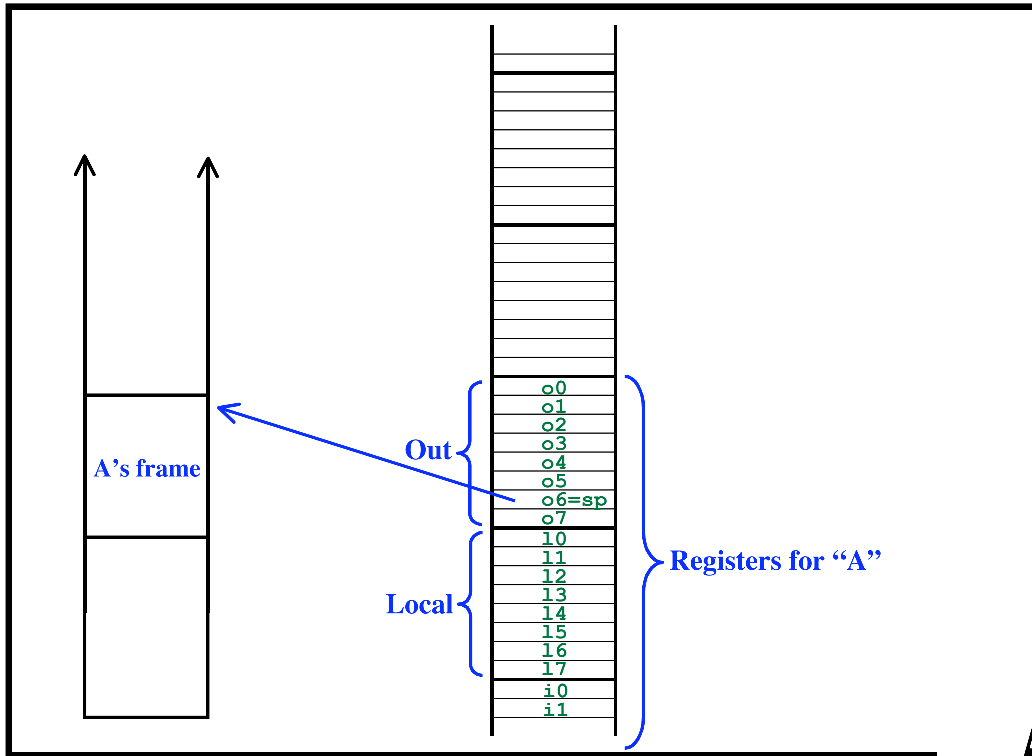
Typical beginning of a routine:

```
        foo:     save       %sp,-64,%sp
                 ...
```

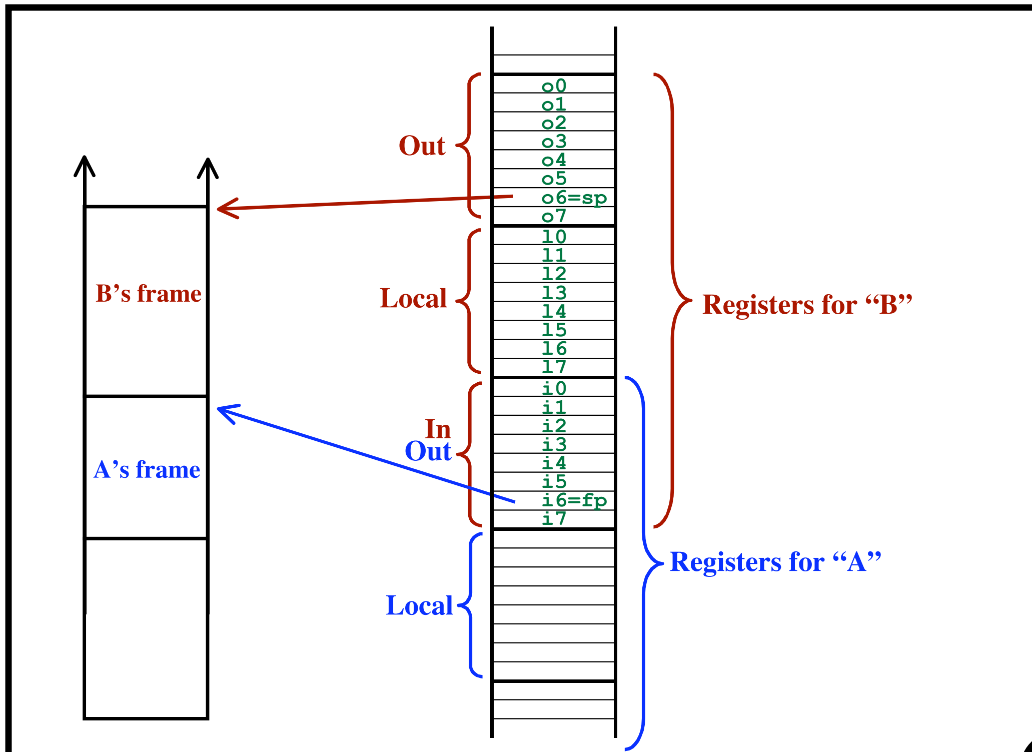


Adjust register window
"out" registers become "in" registers
"new set of "local" registers
`%sp` becomes `%fp`
$\%sp_{NEW} = \%sp_{OLD} - 64$
`%fp` points to beginning of previous frame

© Harry H. Porter, 2006



© Harry H. Porter, 2006



© Harry H. Porter, 2006

%sp and %fp

Stack Frame
= "Activation Record"

The newly called routine (B)...
Will use %sp as a pointer to its newly allocated frame
Will use %fp to access the frame of its caller (A)
where it may find additional arguments

© Harry H. Porter, 2006

Returning From a Routine

```
restore
```

Go back to old register window
The in-memory stack shrinks automatically
(since we go back to old value of %sp)

May also function as an "add" instruction

```
restore    reg1,reg2_or_immed,regD
```

(Not often used.)
NOTE: operands from old window; result to new window

`ret` -- The return instruction
`retl` -- A variation used (to be discussed later)

The "ret" instruction is synthetic:

```
jmpl          %i7,8,%g0
<delay slot>
```

The typical return sequence:

```
ret
restore
```

Put "restore" in the delay slot!

© Harry H. Porter, 2006

Addressing Memory

```
        ld      myVar,%g2
```

Not a legal SPARC instruction
(Used on previous slides, for simplicity)
Why?
Addresses are 32-bits in length.
Instructions are 32-bits in length.
Not enough room to put an address into a single instruction.
To move data into a register...

Must be -4096..+4095

Option #1:

```
        mov     reg_or_immed,regD
```

Synthetic: expanded to:

```
        or      %g0,reg_or_immed,regD
```

Examples:

```
        mov     123,%o5
        mov     %o3,%l2
```

Expanded to "sethi"

Option #2:

```
        set     32_immed,regD
```

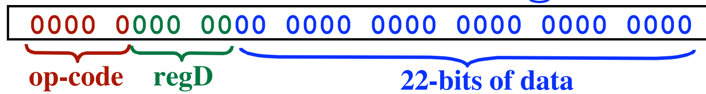
© Harry H. Porter, 2006

The "sethi" Instruction

```
    sethi    22_immed,regD
```

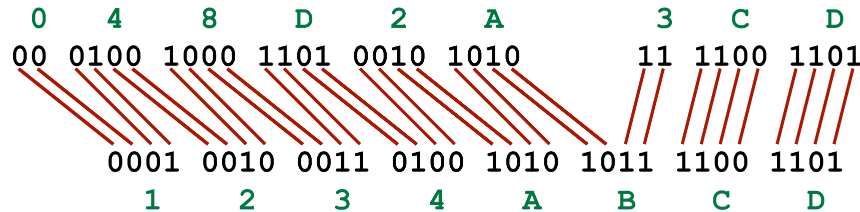
Instruction includes 22 bits of immediate (literal) data.
Moves 22-bits into high-order 22 bits of regD
Moves zeros into low-order 10 bits of regD

Instruction Encoding (approximately)



Example: Move 0x1234ABCD into %g5

```
    sethi    0x48D2A,%g5
    or       %g5,0x000003CD,%g5
```



© Harry H. Porter, 2006

Built-In Macros

`%hi(x)`
Defined as
```
x >> 10
```

`%lo(x)`
Defined as
```
x & 0x3ff
```

Within any instruction...
can use complex expressions

© Harry H. Porter, 2006

The "set" Synthetic Instruction

```
set     value,regD
```

Expands to...

- **If -4096 ≤ value ≤ +4095**
```
or      %g0,value,regD
```

- **If least significant 10 bits of value are zero**
```
sethi   %hi(value),regD
```

- **Otherwise**
```
sethi   %hi(value),regD
or      regD,%lo(value),regD
```

Note:
"set" may expand into 2 instructions
Do not use "set" in a delay slot!

© Harry H. Porter, 2006

Accessing Memory

Goal: increment a word in memory

```
myVar: .word  123
       ...
       set    myVar,%l2
       ld     [%l2],%l3
       add    %l3,1,%l3
       st     %l3,[%l2]
```

Brackets indicate Memory accesses

To move a word from /to memory...

```
       ld     [reg1],regD
       ld     [reg1+reg2],regD
       ld     [reg1+offset],regD
       st     regS,[reg1]
       st     regS,[reg1+reg2]
       st     regS,[reg1+offset]
```

Offset must be 13-bits i.e., within -4096 .. +4095

© Harry H. Porter, 2006

Load / Store Instructions

To move a byte

ldsb -- signed (i.e., sign extend to 32-bits)
ldub -- unsigned (i.e, zero-fill the high-order 3 bytes)
stb

Address must be halfword aligned

To move a halfword

ldsh -- signed (i.e., sign extend to 32-bits)
lduh -- unsigned (i.e., zero-fill the high-order 2 bytes)
sth

Address must be word aligned

To move a word

ld
st

Address must be doubleword aligned

To move a doubleword

ldd
std

Specify an even numbered register. Will move to / from the register pair
%n --> MSW
%n+1 --> LSW

© Harry H. Porter, 2006

Subroutine Linkage

To Call

```
call        foo
<delay slot>
```

All instructions must be word aligned

(1) Multiply 30-bit displacement by 4 (i.e., “sll 2”)
(2) Add to PC

Range is -2^{31} .. $+(2^{31})-1$

32-bit architecture

Can call to any address

64-bit architecture

(1) Multiply 30 bit displacement by 4
(2) Sign-extend to 64-bits
(3) Add to PC

Any register:
Register contains addr of routine.
Call to address stored in that reg.

Also:

```
call        %l5
<delay slot>
```

© Harry H. Porter, 2006

Subroutine Linkage

Call saves the return address in %o7

Saves address of the call instruction itself
The return should be to %o7+8

The routine will look like this:

```
foo: save   %sp,(-64-XXX)&-8,%sp
     ...
     ret
     restore
```

The amount of additional storage, beyond the minimum, to put into the activation record.

old reg window

new reg window

The stack must be doubleword aligned.

```
&-8   =   &0xFFFFFFF8
```

will round down to a multiple of 8

© Harry H. Porter, 2006

Save and Restore

The "save" instruction:

Allocate new "local" and "out" registers
Renames "out" registers to become new "in" registers

```
%o6 = %sp          ⇒  %i6 = %fp
%o7 (ret addr)     ⇒  %i7 (ret addr)
```

The "restore" instruction:

Goes back to the old register window
[We get the old stack to back for free!]

© Harry H. Porter, 2006

Activation Records

(Also called "Frames")

Need Space For...

- 64 bytes
  (16 × 32-bit registers)
- Additional arguments
- Local variables
- Temporary (compiler generated) variables
- Additional register saving
- "Structure Pointer" (4 bytes)

© Harry H. Porter, 2006

Argument Passing

Caller's View:

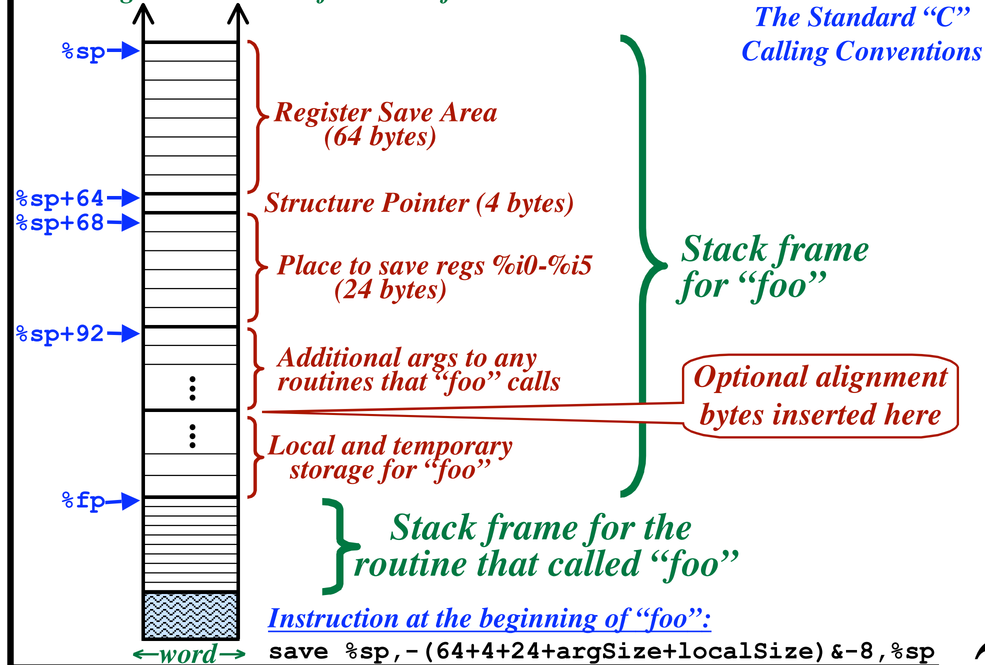
Put argument values in %o0 ... %o5
The subroutine may change the "out" registers during execution
    Assume "out" registers are trashed.
Find the result value (if any) in %o0.
Assume caller will execute "save" and "return"
    Assume "in" and "local" registers are saved / unchanged
Assume the subroutine will trash the condition codes

	caller	callee	
arg1	%o0	%i0	⟵ result returned here
arg2	%o1	%i1	
arg3	%o2	%i2	
arg4	%o3	%i3	
arg5	%o4	%i4	
arg6	%o5	%i5	
	%o6	%i6	⟵ top of stack before the call
	%o7	%i7	⟵ addr of the "call" instruction

© Harry H. Porter, 2006

A Typical Stack Frame

During the execution of routine "foo"...



© Harry H. Porter, 2006

Within "foo"

To access "foo"'s arguments:

arg7	`%fp+92`
arg8	`%fp+96`
arg9	`%fp+100`
arg10	`%fp+104`
...	

To access "foo"'s local storage:

Var 1	`%fp-4`
Var 2	`%fp-8`
Var 3	`%fp-12`
Var 4	`%fp-16`
...	

© Harry H. Porter, 2006

Returning Results

If the returned value is one word or less...

- Callee puts it in %i0
- Caller retrieves it from %o0

If the returned value is larger...

```
struct S { ... };
struct S foo( ) {
  struct S temp = ...;
  ...
  return temp;
}
main () {
  struct S x;
  ...
  x = foo( );
  ...
}
```

A "record" whose size could be 1000 bytes

Returning 1000 bytes of data here

The caller must provide storage for the returned value.
The caller passes a pointer to this space to "foo" in the...

"Structure Pointer"

© Harry H. Porter, 2006

- Caller will allocate space
  ... and pass a pointer to it to "foo"
- Caller will place the structure size (i.e., the # of bytes to be returned) *INLINE*, after the "call" instruction and its delay slot.

In the caller:

```
        add     %fp,x,%o0          Get ptr to x (a local variable)
        call    foo
        st      %o0,[%sp+64]       ... and store it in the frame
        .word   1000
        <...next instruction...>
```

In "foo":

```
foo:    save    %sp,-64,%sp        Allocate frame (e.g., min. size)
        ld      [%i7+8],%o1        Test the size
        cmp     %o1,1000           .
        bne     return / error     .
...........................................................
        ld      [%fp+64],%o0       Load ptr to "x" into %o0
        st      ...,[%o0+0]        Move return value into "x"
        st      ...,[%o0+4]        .
        ...                        .
        st      ...,[%o0+996]      .
return: jmpl    %i7+12,%g0         Return, jumping over delay
        restore                    .  instruction and size
```

Recall that `ret = jmpl %i7+8,%g0`

© Harry H. Porter, 2006

Integer Multiply Instructions

Multiplying two 32-bit numbers... 64-bit result

Decimal:	Binary / Hex:	
999,999	FFFF FFFF	
× 999,999	× FFFF FFFF	
999,998,000,001	FFFF FFFE 0000 0001	← Unsigned
	0000 0000 0000 0001	← Signed

SPARC (two versions)

%y register -- 32-bits

```
umul    reg1,reg2_or_immed,regD
smul    reg1,reg2_or_immed,regD
```

Resulting 64 bits into `%y || regD`

- `%y`: Most Significant Bytes
- `regD`: Least Significant Bytes

To read from %y:

```
rd    %y,regD
```

To move data into %y:

```
wr    reg_or_immed,%y
```

© Harry H. Porter, 2006

Division

For division, the 64-bit dividend is in `%y || reg1`

```
udiv   reg1,reg2_or_immed,regD
sdiv   reg1,reg2_or_immed,regD
```

Result is 32-bits long.
Result in "regD"

Non-integer quotients?
Result rounded toward zero.

Dividing by zero?
Will cause the "divide-by-zero" exception.

Overflow?
The result will be the largest representable integer.

© Harry H. Porter, 2006

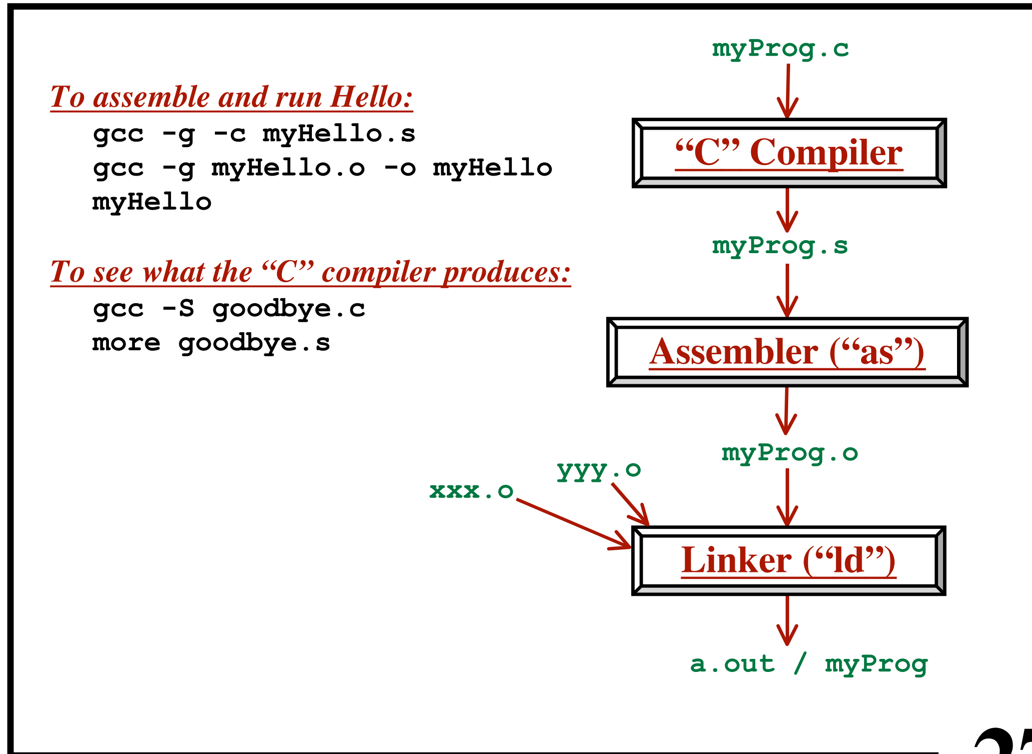
Example SPARC Program

```
! myHello.s
!
! Harry Porter - 12/23/97
!
! This program demonstrates the basics of assembling and running a SPARC
! assembly language program.  It prints out a message when run.
!
        .data                           ! Data will go into 'data' segment
str:    .asciz  "Hello world!!!\n"      ! A string argument to printf

        .text                           ! Code will go into 'text' segment
        .global main                    ! Make this symbol externally known

main:   save    %sp,-(64+4+24)&-8,%sp ! Set up a new activation record
        set     str,%o0                 ! Move arg1 (ptr to str) into %o0
        call    printf                  ! Call printf
        nop                             ! .
        ret                             ! Return to caller after
        restore                         ! .  restoring the stack
```

© Harry H. Porter, 2006



© Harry H. Porter, 2006

Trapping to the O.S.

The "Trap Always" Instruction:

```
mov     6,%g1
ta      0
<no delay slot>
```

The "service request" is in %g1:

```
3 read
4 write
5 open
6 creat
8 close
0 exit (?)
1 exit (?)
```

Changes CPU to "System" mode
and jumps into the O.S. kernel
O.S. will return to "User" program
after executing the request.

Args passed in `%o0, %o1, %o2`, ...
Condition Codes will be set to reflect the overall status: **C=0** *okay*
C=1 *problems*

Unix Kernel Routines:

```
int read (int fd, char * buf, int n) → #bytes
int write (int fd, char * buf, int n) → #bytes
int open (char * name, int flags, int perms) → fileDesc
int creat (char * name, int perms) → fileDesc
void close (int fd)
```

© Harry H. Porter, 2006

Leaf Routines

A "*Leaf Routine*" does not call any other routine.

Don't need a new set of regs.
- No new "register window"
- Avoid the overhead of "save" and "restore"

Args will be in `%o0, %o1, %o2, ... , %o5`

Result should be placed in `%o0`

Must not modify `%i0, %i1, %i2, ... , %i7`
`%l0, %l1, %l2, ... , %l7`

To return:

```
retl
<delay slot>
```

This is a synthetic instruction:
`jmpl %o7,8,%g0`
Add 8 to `%o7` and jump to that address
Store current address in `%g0` (i.e., discard it)

© Harry H. Porter, 2006

Floating Point Data Sizes

Single Precision	word	32-bits (4 bytes)
Double Precision	doubleword	64-bits (8 bytes)
Quad Precision	quadword	128-bits (16 bytes)

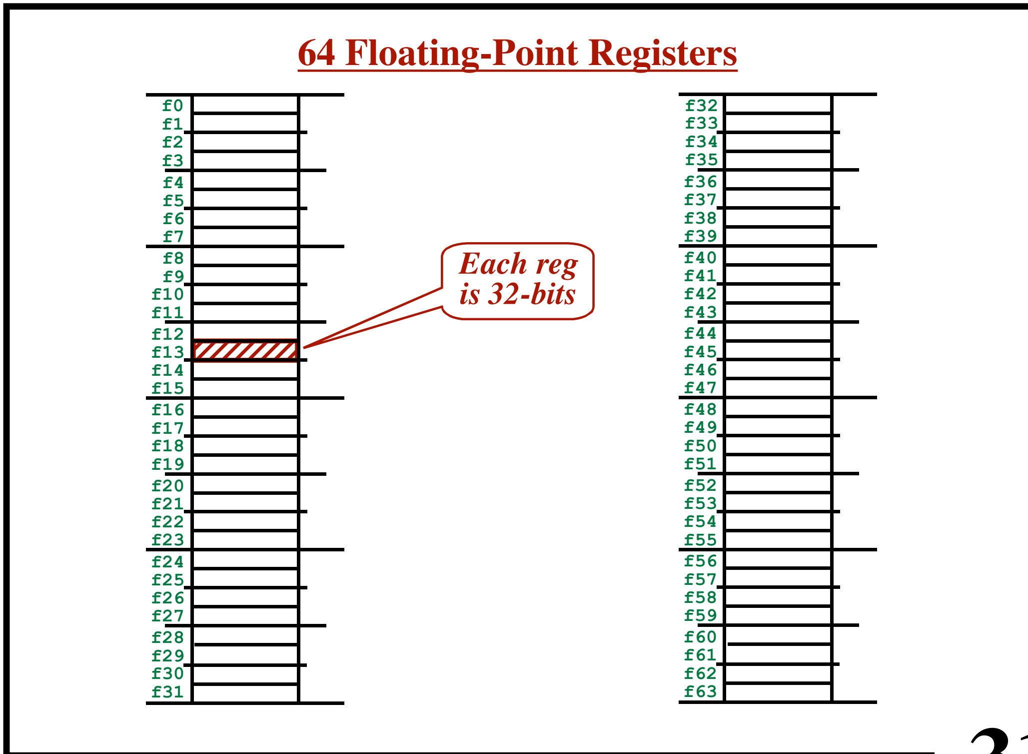
Floating-Point Register File
- Total: 64 words of storage

Organization:

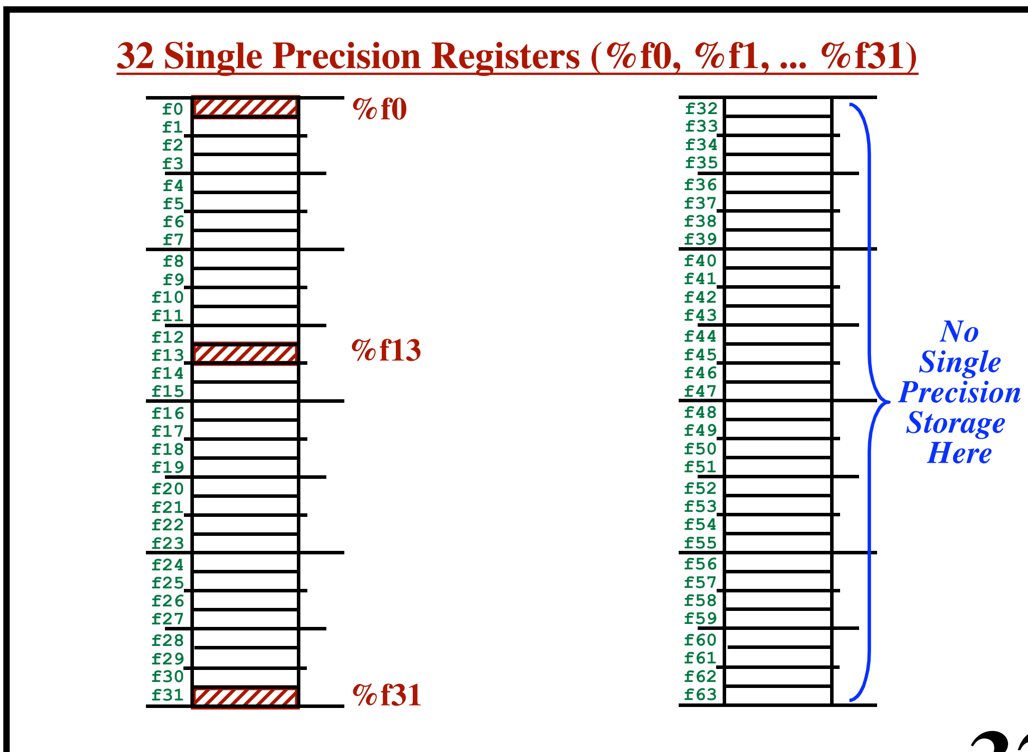
32 single precision regs	`%f0,%f1,%f2, ... %f31`
32 double precision regs	`%f0,%f2,%f4, ... %f62`
16 quad precision regs	`%f0,%f4,%f8, ... %f60`

These registers overlap!

© Harry H. Porter, 2006

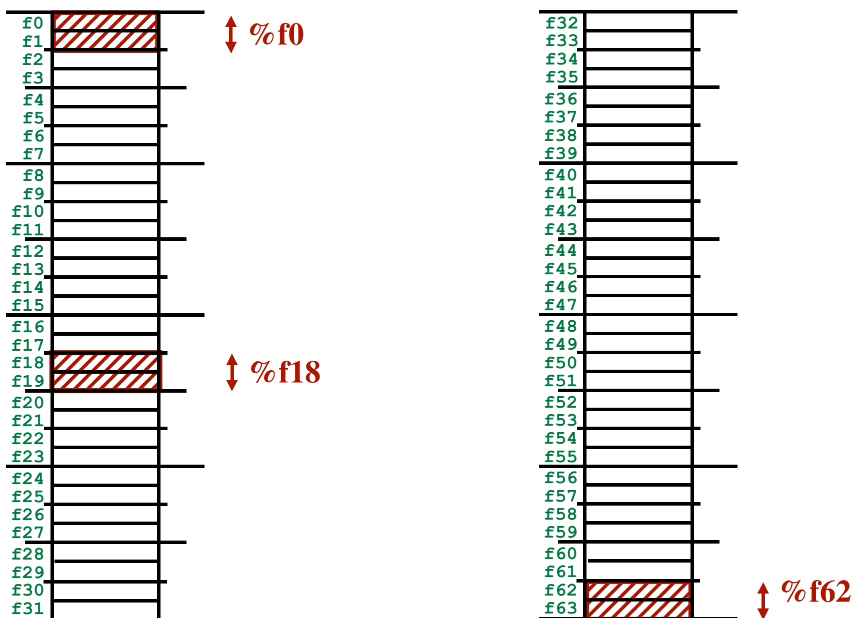


© Harry H. Porter, 2006



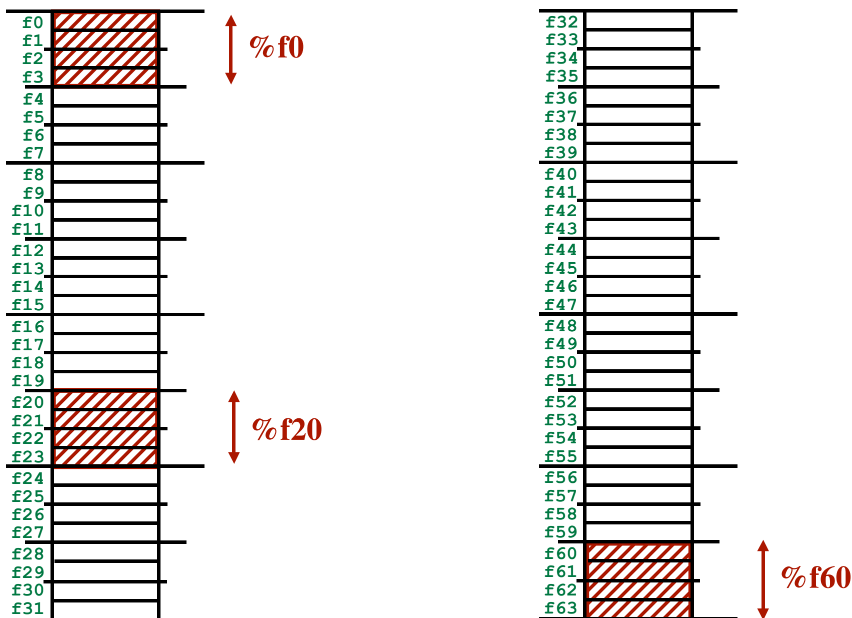
© Harry H. Porter, 2006

32 Double Precision Registers (%f0, %f2, ... %f62)



© Harry H. Porter, 2006

16 Quad Precision Registers (%f0, %f4, ... %f60)



© Harry H. Porter, 2006

Loading and Storing Floating Registers

Example:

```
ldf     [%l4],%f3
```

Destination floating-point register

Addresses use integer registers

Instructions:

```
ldf     [address],freg
lddf    [address],freg
ldqf    [address],freg
stf     freg,[address]
stdf    freg,[address]
stqf    freg,[address]
```

Proper alignment required

Where **address** is:

```
[reg]              [%i3]
[reg+reg]          [%i3+%l4]
[reg+offset]       [%i3+x]
```

If the assembler sees a floating point register...

You may use:	*In place of:*
`ld`	`ldf`
`ldd`	`lddf`
`st`	`stf`
`std`	`stdf`

© Harry H. Porter, 2006

Fixed-Point Numbers

Decimal

123.456

... 10^2 10^1 10^0 10^{-1} 10^{-2} 10^{-3} ...

... 100 10 1 $^1/_{10}$ $^1/_{100}$ $^1/_{1000}$ ...

Binary

101.0101

... 2^2 2^1 2^0 2^{-1} 2^{-2} 2^{-3} 2^{-4} ...

... 4 2 1 $^1/_2$ $^1/_4$ $^1/_8$ $^1/_{16}$ ...

What is this number? $4 + 1 + {}^1/_4 + {}^1/_{16} = 5\ {}^5/_{16} = 5.3125$

© Harry H. Porter, 2006

"Every binary fraction can be represented exactly with a decimal fraction."

$1001.01_2 = 9.25_{10}$

(And the decimal representation will use no more decimal digits to the right of "." than the binary number has bits.)

"Some decimal fractions cannot be represented exactly using binary fractions."

$0.3_{10} = 0.01\underbrace{0011}\underbrace{0011}\underbrace{0011}\underbrace{0011}\underbrace{0011}..._2$

$= 0.01\overline{0011}_2$

(of finite length)

© Harry H. Porter, 2006

CS-322 SPARC-Part 3

Floating Point Numbers

Decimal

```
  123.456
= 1.2345 × 10^2
  6.0225 × 10^23
```

Limited precision: Rounded to the nearest $\mathbf{10^{19}}$
The leading digit will always be 1,2,3, ..., 9 (never 0).

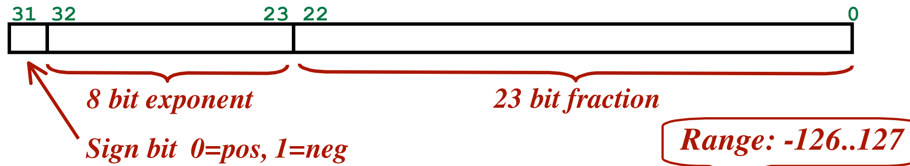
Binary

```
  101.0101
= 1.010101 × 2^2
= 1.328125 × 4 = 5.3125
```

Note: The leading bit will always be "1" (never "0").
No need to store the first bit!

© Harry H. Porter, 2006

Single Precision Floating Point Number Representation



Range: -126..127

$$N = (-1)^{sign} \times 1.fraction \times 2^{exp}$$

The exponent is an 8 bit number interpreted as follows...

```
0000 0000      "sub-normal"
0000 0001      -126
   ...           ...
0111 1110        -1
0111 1111         0
1000 0000         1
   ...           ...
1111 1110       127
1111 1111      "not a number"
```

Single-Precision

Smallest non-zero number:
$1.17549435 \times 10^{-38}$

Largest number:
$3.40282347 \times 10^{+38}$

About 9 digits of accuracy!

© Harry H. Porter, 2006

NaN: Not-A-Number

When
exp = `1111 1111`
a special meaning arises

`0xFFFF FFFF`
(= -1 as a signed number)
Not A Number
"**NaN**"
Will cause an exception when used.

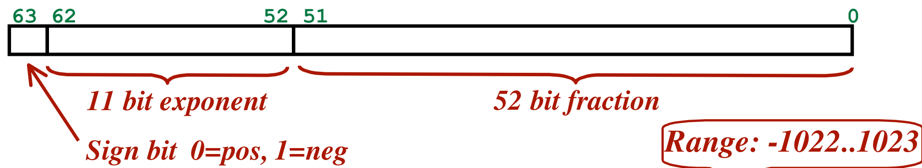
`0x7FF0 0000`
Positive Infinity, $+\infty$
"**+inf**"

`0xFFF0 0000`
Negative Infinity, $-\infty$
"**-inf**"

Divide $^{1}/_{0} \Rightarrow +\infty$
Divide $-^{1}/_{0} \Rightarrow -\infty$
You can compare other numbers to $+\infty$ and $-\infty$.

© Harry H. Porter, 2006

Double Precision Floating Point Number Representation



Range: -1022..1023

$$N = (-1)^{sign} \times 1.fraction \times 2^{exp}$$

Double-Precision

Smallest non-zero number:

2.225,073,858,507,201,4 × 10^{-308}

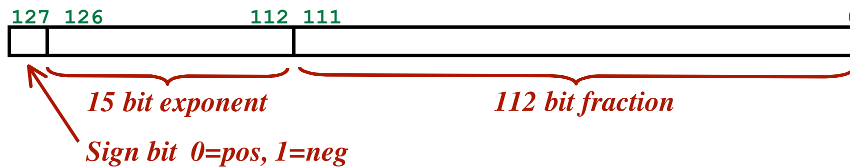
Largest number:

1.797,693,134,862,315,7 × 10^{+308}

About 17 digits of accuracy!

© Harry H. Porter, 2006

Quad Precision Floating Point Number Representation

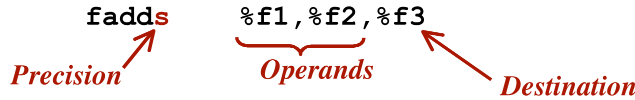


$$N = (-1)^{sign} \times 1.fraction \times 2^{exp}$$

© Harry H. Porter, 2006

Floating-Point Computation

Example:



Arithmetic Instructions:

```
fadds
fsubs
fmuls
fdivs
```
Single Precision

```
faddd
fsubd
fmuld
fdivd
```
Double Precision

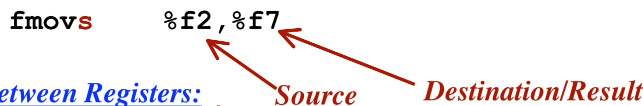
```
faddq
fsubq
fmulq
fdivq
```
Quad Precision

Both operands and result must be in floating-point registers (no literal data).

© Harry H. Porter, 2006

Misc. Floating-Point Instructions

Example:



Move Between Registers:

```
fmovs
fmovd
fmovq
```

Negation:

```
fnegs
fnegd
fnegq
```

Absolute Value:

```
fabss
fabsd
fabsq
```

Square Root:

```
fsqrts
fsqrtd
fsqrtq
```

All take 2 registers

source → destination

Example

```
fnegd     %f4,%f6
```

© Harry H. Porter, 2006

Comparing

Compare two floating-point numbers:

```
fcmps  freg1,freg2
fcmpd  freg1,freg2
fcmpq  freg1,freg2
```

Floating-Point Condition Codes

- **E** set to **1** iff `freg1 = freg2`
- **L** set to **1** iff `freg1 < freg2`
- **G** set to **1** iff `freg1 > freg2`
- **U** set to **1** iff `freg1` and `freg2` cannot be ordered (e.g., NaN)

> Note: These bits are different from the integer condition codes
>
> **Integer Condition Codes**
>
> - **Z** zero
> - **C** carry
> - **N** negative
> - **V** overflow

© Harry H. Porter, 2006

Branching

Compare two floating-point numbers:

`fbl`	$<$
`fble`	$\leq$
`fbg`	$>$
`fbge`	$\geq$
`fbe`	$=$
`fbne`	$\neq$
`fbu`	*unordered*
`fbo`	*ordered*

(Other combinations)

> *Example:*
>
> `fble    label`
>
> Test the floating-point conditions
> ...and branch to "label" if the condition is true.

Delay Slot:

Just like all other branch instructions (i.e., next instruction is executed first)

Annul Bit

Just like the integer branch instructions:

If the annul bit is set (`fble,a   label`)
and if the condition is false (i.e., branch not taken)
then do not execute instruction in the delay slot!

© Harry H. Porter, 2006

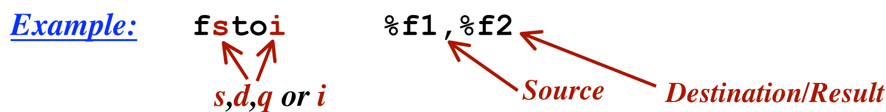
Older Version of the architecture...
- SPARC-V8
  - The branch may not immediately follow the compare instruction
  - You need an intervening instruction (e.g., nop)

SPARC-V9
- No restriction

Conversions

Conversion between integer and floating-point representation

Both operands must be in a floating-point register.



The Instructions:

Source \ Destination	int	single	double	quad
int	-	fitos	fitod	fitoq
single	fstoi	-	fstod	fstoq
double	fdtoi	fdtos	-	fdtoq
quad	fqtoi	fqtos	fqtod	-

gdb

A debugger for C, Assembly, etc.
Command line
`%gdb a.out`
- Loads the program in main memory
- Does not begin execution
- Accepts gdb commands

`%gdb a.out core`
- Loads the core file back into main memory
- You can see the state of things at the time of the fault.

`help [command]`
Displays help info.
`help run`

`run [args]`
`r [args]`
Begin execution at program's beginning.
Optionally supply command line arguments
May abbreviate any command
`r < test1.pcat > test1.out`

© Harry H. Porter, 2006

gdb

`b *addr`
Set a breakpoint.
```
b iscan
b *&iscan+44
```

`info break`
`i b`
Show all breakpoints.

`continue`
`c`
Continue / resume execution after reaching a breakpoint.

© Harry H. Porter, 2006

Print Command

`print value`

Display data values

`p $g6`

Registers use `$` instead of `%`.

Default: print value in decimal

```
p/x $g6   ← Prints value in hex
p/d $g6   ← decimal
p/t $g6   ← binary
p/f $g6   ← floating point
p/c $g6   ← character
p/i $g6   ← Interpret data as a SPARC instruction and print it
```

`p/x iscan`

Print out the value of a symbol

Print the address of the instruction labeled "iscan"

`p/x $pc`

Show where execution is currently stopped (i.e., value of "program counter")

`p/x 123*5`

Perform computation and print result in hex

© Harry H. Porter, 2006

Examine and Set

`x addr`

Examine memory contents

`x/i main`

Disassemble and display first instruction

`<cr>`

Hit enter to repeat last instruciton, stepping through memory

`x/x myVar`

Display the value of "myVar" in hex

`x/x 0x20844`

Display the contents of memory at address 00020844_{16}

`set $g4=0x1234abcd`

Move a data value into a register

`set myVar=0x4321`

Move a data value into memory

© Harry H. Porter, 2006

More Commands

disass
Disassemble and print the current routine

ni
"Next Instruction": Execute a single SPARC instruction
(Steps over calls; i.e., execute the entire routine then stop)
`ni 13` Execute 13 instructions and stop

si
Single-steps the CPU
Execute a single SPARC instruction
"Step Into": Steps through subroutines too (ugh...)

info all-registers
i all
Display the contents of all registers.

display
Like "examine" command, except done whenever a breakpoint it encountered.

q
Quit gdb

© Harry H. Porter, 2006

Commands Related To High-Level Languages

call foo(3,5,7)
Invoke a routine with the given arguments.
Will print the returned value.

print myVar
Print a variable.
Print using the format that is appropriate for its source-code type.

step
Single-steps a single source code line.

next
Single-steps a single source code line.
(Steps over calls)

where
Print the calling stack.

bt
"Backtrace"
Print the calling stack (slightly different info)

© Harry H. Porter, 2006

Unix Usage

```
gcc -g myProg.s / myProg.c
```

Causes symbol info to be added to .o files (and a.out)

... so gdb can learn about symbolic names and values

```
gdb a.out
```

The `.gdbinit` file

Automatically read and executed upon gdb start-up.

Contains gdb commands.

Example

```
break *&main      ← Set a breakpoint
display/i $pc     ← Display current instruction when b.p. hit
r                 ← Start execution (and stop immediately)
disass            ← Disassemble and display the main routine
```

© Harry H. Porter, 2006

Example

```
! myFloat.s
!
! Harry Porter - 1/13/00
!
! This program demonstrates floating point computation.
! It also demonstrates the preferred style of commenting.
!
! This program performs this function:
!
!       main () {
!         double x = 12.34;
!         double y = 10;
!         printf ("Result = %g\n", x + y);
!       }
!
! Frame layout:
!   %fp-24   ==> y     (a floating double)
!   %fp-16   ==> x     (a floating double)
!   %fp-8    ==> temp  (a floating double)
! (Size of local storage = 3*8 = 24 bytes)
!
```

© Harry H. Porter, 2006

Frame Layout

```
    64    Register Save Area
    4     Structure Pointer
    24    %i0 - %i5
    0     Additional arguments to routines "myFloat" calls (none)
    4     Alignment
    8     y
    8     x
    8     temp
==============
    120   bytes
```

%fp-24 "y"
%fp-16 "x"
%fp-8 "temp"

© Harry H. Porter, 2006

%fp-24 "y"
%fp-16 "x"
%fp-8 "temp"

```
         .text
strArg:  .asciz    "Result = %g\n"          ! String constant

         .align    8
xConst:  .double   0r+1234.0e-2             ! Constant = 12.34
yConst:  .double   0r+1.00E1                ! Constant = 10.00

         .global   main                     ! Begin main function
main:    save      %sp,-120,%sp             ! .

         sethi     %hi(xConst),%o0          ! Initialize x (%fp-16)
         ldd       [%o0+%lo(xConst)],%o2    ! .  from stored
         std       %o2,[%fp-16]             ! .  constant "xConst"

         sethi     %hi(yConst),%o0          ! Initialize y (%fp-24)
         ldd       [%o0+%lo(yConst)],%o2    ! .  from stored
         std       %o2,[%fp-24]             ! .  constant "yConst"
```

© Harry H. Porter, 2006

%fp-24 "y"
%fp-16 "x"
%fp-8 "temp"

```
        ldd     [%fp-16],%f2        ! Add x+y, storing result in
        ldd     [%fp-24],%f4        ! .  temp var (%fp-8)
        faddd   %f2,%f4,%f6         ! .
        std     %f6,[%fp-8]         ! .

        ldd     [%fp-8],%o2         ! Load value of temp
        mov     %o2,%o1             ! .  into %o1 and %o2
        mov     %o3,%o2             ! .
        set     strArg,%o0          ! Load addr of strArg into %o0
        call    printf              ! Call printf
        nop                         ! .

        ret                         ! Return from "main"
        restore                     ! .
```

© Harry H. Porter, 2006