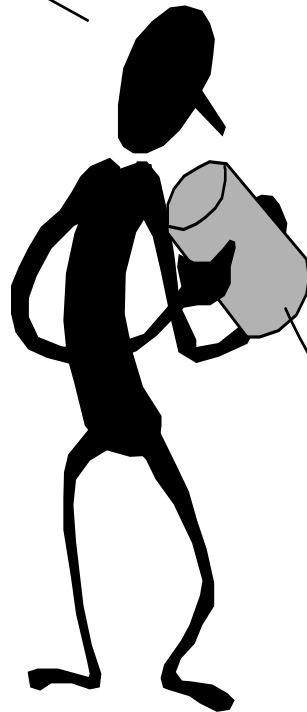
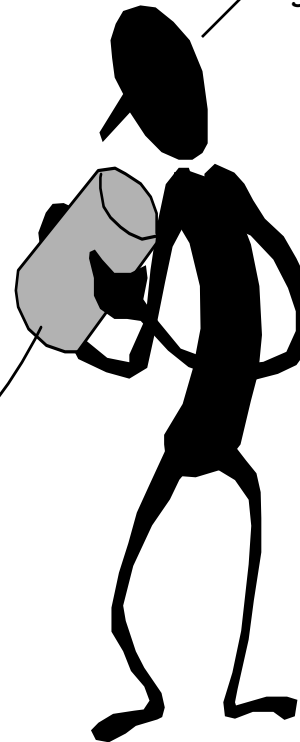


Stacks and Procedures

I forgot, am I
the Caller
or Callee?



Don't know. But, if
you PUSH again I'm
gonna POP you.



Support for High-Level Language constructs are an integral part of modern computer organization. In particular, support for subroutines, procedures, and functions.

An Aside: Pseudoinstructions

MIPS has relatively few instructions, however, it is possible to “fake” new instructions by taking advantage of special ISA properties (i.e. %0 is always zero, clever use of immediate values)

Examples:

Why both?



move \$d,\$s

becomes

addi \$d,\$s,0

neg \$d,\$s

becomes

sub \$d,\$0,\$s

negu \$d,\$s

becomes

subu \$d,\$0,\$s

not \$d,\$s

becomes

nor \$d,\$s,\$0

subiu \$d,\$s,imm16

becomes

addiu \$d,\$s,-imm16

b label

becomes

beq \$0,\$0,label

Do Nothing



sge \$d,\$s,\$t

becomes

slt \$d,\$t,\$s

nop

becomes

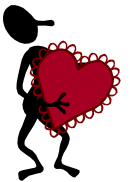
sll \$0,\$0,0

Uber Pseudoinstruction

There is one pseudo instruction where MIPS goes crazy. It essentially generates different instructions depending on the context:

- 1) `la $d, offset($base)`
- 2) `la $d, offset`
- 3) `la $d, ($base)`

The MIPS compiler
loves this
pseudoinstruction



It mimics the format of lw/sw instructions, but rather than reading/writing the contents of memory, it loads its destination register with the “effective address” that would have been accessed. As a result it can generate any one of the following five sequences:

2) `lui $d,offset`
`ori $d,$d,offset`

1) `lui $1,offset`
`ori $1,$1,offset`
`addu $d,$base,$1`

`ori $d,$0,offset`

`ori $1,$0,offset`
`addu $d,$base,$1`

`addiu $d,$base,offset`

The Beauty of Procedures

- Reusable code fragments (modular design)

```
clear_screen();
```

```
...
```

```
# code to draw a bunch of lines
```

```
clear_screen();
```

```
...
```



- Parameterized functions (variable behaviors)

```
line(x1, y1, x2, y2, color);
```

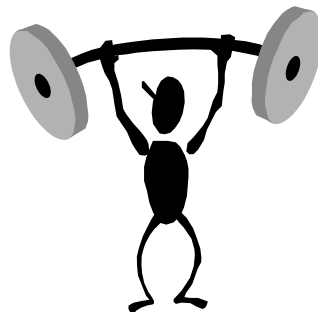
```
line(x2,y2,x3,y3, color);
```

```
...
```

```
for (i=0; i < N-1; i++)
```

```
    line(x[i],y[i],x[i+1],y[i+1],color);
```

```
line(x[i],y[i],x[0],y[0],color);
```



More Procedure Power

- Global vs. Local scope (Name Independence)

```
int x = 9;
```

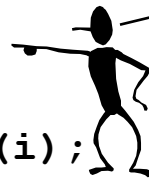
```
int fee(int x) {  
    return x+x-1;  
}
```

```
int foo(int i) {  
    int x = 0;  
    while (i > 0) {  
        x = x + fee(i);  
        i = i - 1;  
    }  
    return x;  
}
```

```
main() {  
    fee(foo(x));  
}
```



These are different "x"s



This is yet another "x"

How do we
keep track of
all the
variables



That "fee()" seems odd
to me? And, foo()'s a
little square.



Using Procedures

- A “calling” program (**Caller**) must:
 - Provide procedure parameters. In other words, put the arguments in a place where the procedure can access them
 - Transfer control to the procedure. Jump to it
- A “called” procedure (**Callee**) must:
 - Acquire the resources needed to perform the function
 - Perform the function
 - Place results in a place where the Caller can find them
 - Return control back to the Caller
- Solution (a least a partial one):
 - WE NEED CONVENTIONS, agreed upon standards for how arguments are passed in and how function results are retrieved
 - Solution #1: Allocate registers for these specific functions

MIPS Register Usage

- Conventions designate registers for procedure arguments (\$4-\$7) and return values (\$2-\$3).
- The ISA designates a “linkage register” for calling procedures (\$31)
- Transfer control to Callee using the jal instruction
- Return to Caller with the j \$31 or j \$ra instruction

The “linkage register” is where the return address of back to the callee is stored.



This allows procedures to be called from any place, and for the caller to come back to the place where it was invoked.

Name	Register number	Usage
\$zero	0	the constant value 0
\$at	1	assembler temporary
\$v0-\$v1	2-3	procedure return values
\$a0-\$a3	4-7	procedure arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved by callee
\$t8-\$t9	24-25	more temporaries
\$k0-\$k1	26-27	reserved for operating system
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

And It "Sort Of" Works

- Example:

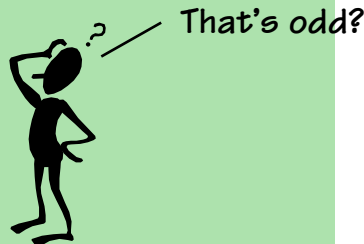
```
.globl x
.data
x:      .word 9
```

```
.globl fee
.text
fee:
    addu   $v0,$a0,$a0
    addiu  $v0,$v0,-1
    jr     $ra
```

Callee

```
.globl main
.text
main:
    lw     $a0,x
    jal    fee
    jr     $ra
```

Caller



Works for special cases where the Callee needs few resources and calls no other functions.

This type of function (one that calls no others) is called a **LEAF** function.

But there are lots of issues:

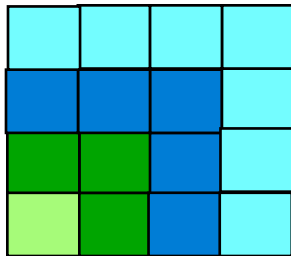
- How can fee call functions?
- More than 4 arguments?
- Local variables?
- Where will main return to?

Let's consider the worst case of a Callee as a Caller...

Writing Procedures

```
int sqr(int x) {  
    if (x > 1)  
        x = sqr(x-1)+x+x-1;  
    return x;  
}
```

```
main()  
{  
    sqr(10);  
}
```



How do we go about writing callable procedures? We'd like to support not only LEAF procedures, but also procedures that call other procedures, ad infinitum (e.g. a recursive function).

Oh, recursion gives me a headache.


$$\begin{aligned} \text{sqr}(10) &= \text{sqr}(9)+10+10-1 = 100 \\ \text{sqr}(9) &= \text{sqr}(8)+9+9-1 = 81 \\ \text{sqr}(8) &= \text{sqr}(7)+8+8-1 = 64 \\ \text{sqr}(7) &= \text{sqr}(6)+7+7-1 = 49 \\ \text{sqr}(6) &= \text{sqr}(5)+6+6-1 = 36 \\ \text{sqr}(5) &= \text{sqr}(4)+5+5-1 = 25 \\ \text{sqr}(4) &= \text{sqr}(3)+4+4-1 = 16 \\ \text{sqr}(3) &= \text{sqr}(2)+3+3-1 = 9 \\ \text{sqr}(2) &= \text{sqr}(1)+2+2-1 = 4 \\ \text{sqr}(1) &= 1 \\ \text{sqr}(0) &= 0 \end{aligned}$$

Procedure Linkage: First Try

Callee/Caller

```
int sqr(int x) {
    if (x > 1)
        x = sqr(x-1)+x+x-1;
    return x;
}
```

```
sqr:    addiu   $t0,$0,1
        slt    $t0,$t0,$a0    # 1 < x ?
        beq   $t0,$0,endif
        move  $t0,$a0        # save x
        addiu $a0,$a0,-1
        jal   sqr            # sqr(x-1)
        addu  $v0,$v0,$t0
        addu  $v0,$v0,$t0
        addiu $v0,$v0,-1
        b     rtn
endif:  move  $v0,$a0
rtn:    jr    $ra
```

\$t0 is clobbered on successive calls.



Will saving "x" in some register or at some fixed location in memory help?

(Nope)

We also clobber our return address, so there's no way back!



Caller

```
main()
{
    sqr(10);
}
```

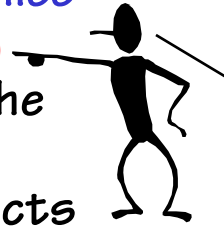
MIPS Convention:

- pass 1st arg x in \$a0
- save return addr in \$ra
- return result in \$v0
- use only temp registers to avoid saving stuff

A Procedure's Storage Needs

Basic Overhead for Procedures/Functions:

- **Caller** sets up ARGUMENTs for **callee**
`f(x, y, z)` or worse... `sin(a+b)`
- **Caller** invokes **Callee** while saving the Return Address to get back
- **Callee** saves stuff that **Caller** expects to remain unchanged
- **Callee** executes
- **Callee** passes results back to **Caller**.



In C it's the caller's job to evaluate its arguments as expressions, and pass the resulting values to the callee... Therefore, the CALLEE has to save arguments if it wants access to them after calling some other procedure, because they might not be around in any variable, to look up later.

Local variables of Callee:

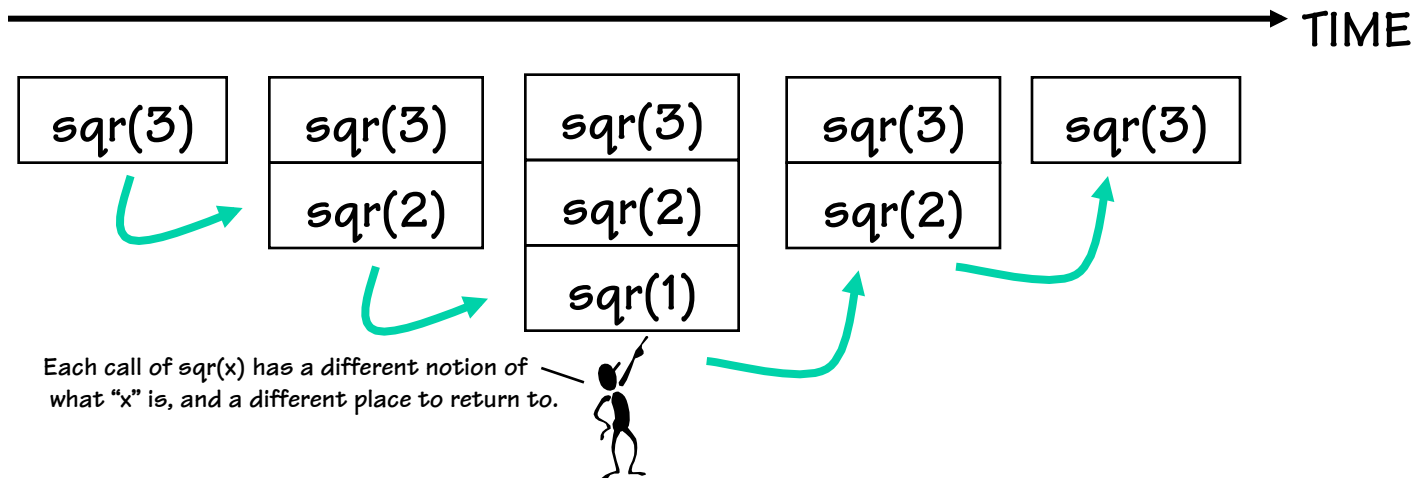
```
...  
{  
  int x, y;  
  ... x ... y ...;  
}
```

Each of these is specific to a “particular” invocation or **activation** of the Callee. Collectively, the arguments passed in, the return address, and the callee's local variables are its **activation record**, or **call frame**.

Lives of Activation Records

```
int sqr(int x) {  
    if (x > 1)  
        x = sqr(x-1)+x+x-1;  
    return x;  
}
```

Where do we store
activation
records?



A procedure call creates a new activation record. Caller's record is preserved because we'll need it when call finally returns.

Return to previous activation record when procedure finishes, permanently discarding activation record created by call we are returning from.

We Need Dynamic Storage!

What we need is a *SCRATCH* memory for holding temporary variables. We'd like for this memory to *grow and shrink as needed*. And, we'd like it to have an *easy management policy*.

One possibility is a

STACK

A last-in-first-out (LIFO) data structure.



Some interesting properties of stacks:

SMALL OVERHEAD. Everything is referenced relative to the top, the so-called “top-of-stack”

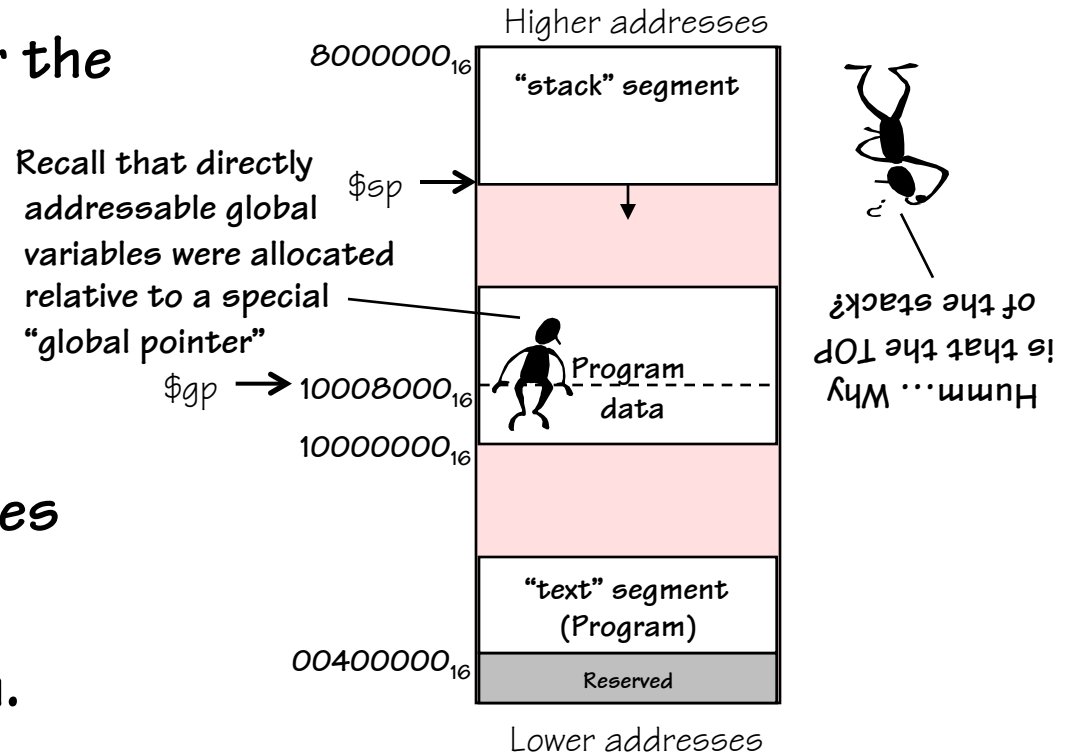
Add things by **PUSHING** new values on top.

Remove things by **POPPING** off values.

MIPS Stack Convention

CONVENTIONS:

- Waste a register for the Stack Pointer ($\$sp = \29).
- Stack grows **DOWN** (towards lower addresses) on pushes and allocates
- $\$sp$ points to the **TOP** *used* location.
- Stack is placed far away from the program and its data



Other possible implementations include:

- 1) stacks that grow "UP"
- 2) SP points to first UNUSED location

Stack Management Primitives

ALLOCATE **k**: reserve k WORDS of stack

$$\text{Reg}[SP] = \text{Reg}[SP] - 4*k$$

```
addiu $sp,$sp,-4*k
```

DEALLOCATE **k**: release k WORDS of stack

$$\text{Reg}[SP] = \text{Reg}[SP] + 4*k$$

```
addiu $sp,$sp,4*k
```

PUSH **\$x**: push Reg[x] onto stack

$$\text{Reg}[SP] = \text{Reg}[SP] - 4$$

$$\text{Mem}[\text{Reg}[SP]] = \text{Reg}[x]$$

An ALLOCATE 1 followed by a store



```
addiu $sp,$sp,-4  
sw $x, ($sp)
```

POP **\$x**: pop the value on the top of the stack into Reg[x]

$$\text{Reg}[x] = \text{Mem}[\text{Reg}[SP]]$$

$$\text{Reg}[SP] = \text{Reg}[SP] + 4;$$

A load followed by a DEALLOCATE 1



```
lw $x, ($sp)  
addiu $sp,$sp,4
```

Fun with Stacks

Stacks can be used to squirrel away variables for later. For instance, the following code fragment can be inserted anywhere within a program.

```
#  
# Argh!!! I'm out of registers Scotty!!  
#  
addiu    $sp,$sp,-8      # allocate 2  
sw       $s0,4($sp)      # Free up s0  
sw       $s1,0($sp)      # Free up s1  
lw       $s0,dilithum_xtals  
lw       $s1,seconds_til_explosion  
suspense: addiu    $s1,$s1,-1  
bne      $s1,$0,suspense  
sw       $s0,warp_engines  
lw       $s1,0($sp)      # Restore s1  
lw       $s0,4($sp)      # Restore s0  
addiu    $sp,$sp,8      # deallocate 2
```

You should
ALWAYS
allocate
prior to
saving, and
deallocate
after
restoring
in order to
be SAFE!



AND Stacks can also be used to solve other problems...

More MIPS Procedure Conventions

What needs to be saved?

CHOICE 1... anything that a Callee touches
(except the return value registers)

CHOICE 2... Give the Callee access to everything
(make the Caller will save those
registers it expects to be unchanged)

CHOICE 3... Something in between.

Of course, the
MIPS convention
is this case.



(Give the Callee some registers to play with. But, make him save others if they are not enough, and also provide a few registers that the caller can assume will not be changed by the callee.)

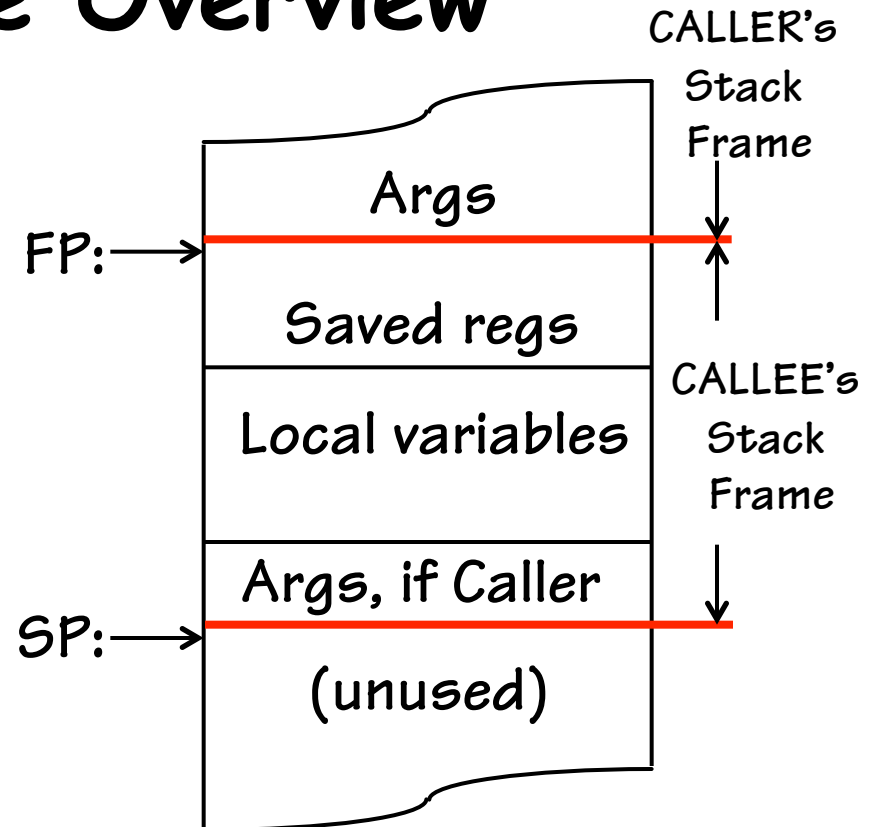
Stack Frame Overview

The STACK FRAME contains storage for the CALLER's volatile state that it wants preserved after the invocation of CALLEEs.

In addition, the CALLEE will use the stack for the following:

- 1) Accessing the arguments that the CALLER passes to it (specifically, the 5th and greater)
- 2) Saving non-temporary registers that it wishes to modify
- 3) Accessing its own local variables

The boundary between stack frames falls at the first word of state saved by the CALLEE, and just after the 5th argument (if used) passed in from the CALLER. The FRAME POINTER keeps track of this boundary between stack frames.



It's possible to use only the SP to access a stack frame, but offsets may change due to ALLOCATEs and DEALLOCATEs. For convenience a \$fp is used to provide CONSTANT offsets to local variables and arguments

Procedure Stack Usage

ADDITIONAL space must be allocated in the stack frame for:

1. Any LOCAL variables declared within the procedure
2. Any SAVED registers the procedure uses (\$s0-\$s7, \$ra, \$fp)
3. Any TEMPORARY registers that the procedure wants preserved IF it calls other procedures (\$t0-\$t9)
4. Other TEMP space IF the procedure runs out of registers (RARE)
5. Enough “outgoing” arguments to satisfy the worse case **ARGUMENT SPILL** of ANY procedure it calls.
(SPILL is the number of arguments greater than 4).

Reminder; stack frames are extended by multiples of 2 word (8 bytes). By convention, the above order is the order in which storage is allocated



Each procedure has keep track of how many SAVED and TEMPORARY registers are on the stack in order to calculate the offsets to LOCAL VARIABLES.



2/4/13

PRO: The MIPS stack frame convention minimizes the number of stack ALLOCATES

CON: The MIPS stack frame convention tends to allocate larger stack frames than needed, thus wasting memory

More MIPS Register Usage

- The registers $\$s0-\$s7$, $\$sp$, $\$ra$, $\$gp$, $\$fp$, and the stack above the memory above the stack pointer must be preserved by the CALLEE
- The CALLEE is free to use $\$t0-\$t9$, $\$a0-\$a3$, and $\$v0-\$v1$, and the memory below the stack pointer.
- No “user” program can use $\$k0-\$k1$, or $\$at$

Name	Register number	Usage
$\$zero$	0	the constant value 0
$\$at$	1	assembler temporary
$\$v0-\$v1$	2-3	procedure return values
$\$a0-\$a3$	4-7	procedure arguments
$\$t0-\$t7$	8-15	temporaries
$\$s0-\$s7$	16-23	saved by callee
$\$t8-\$t9$	24-25	more temporaries
$\$k0-\$k1$	26-27	reserved for operating system
$\$gp$	28	global pointer
$\$sp$	29	stack pointer
$\$fp$	30	frame pointer
$\$ra$	31	return address

Stack Snap Shots

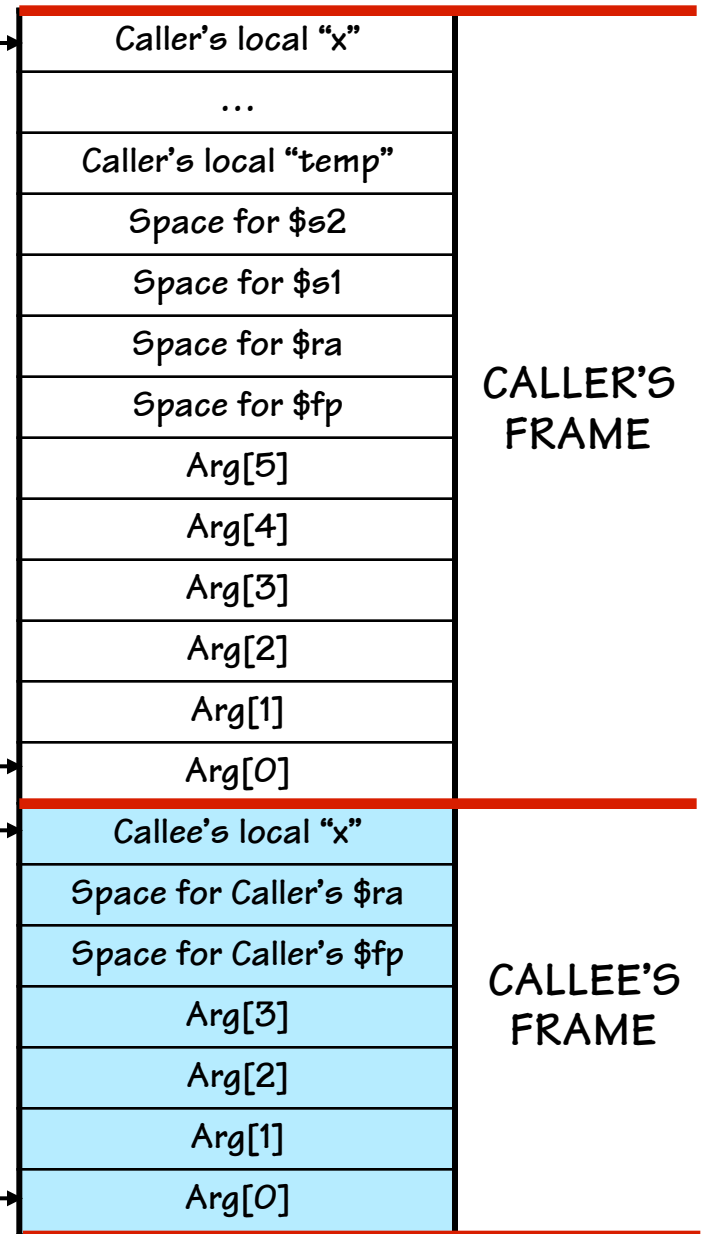
Shown on the right is a snap shot of a program's stack contents, taken at some instance in time. One can mine a lot of information by inspecting its contents.

Can we determine the number of CALLEE arguments? **NOPE**

Can we determine the maximum number of arguments needed by any procedure called by the CALLER? **Yes, there can be no more than 6**

Where in the CALLEE's stack frame might one find the CALLER's \$fp? **It MIGHT be at Mem[\$fp+4]+4**

CALLER's \$fp →



\$sp (prior to call) →

CALLEE's \$fp →

\$sp (after call) →

Simple Cases

A leaf needing minimal resources: Generated assembly code:

```
int isOdd(int x) {
    return (x & 1);
}
```


A function that calls others and has local variables:

```
int parity(a,b,c,d) {
    int sum = a + b + c + d;
    return isOdd(sum);
}
```

```
isOdd:  andi    $2,$4,1
L_1:    jr     $31

parity: addiu   $sp,$sp,-32
        sw     $31,20($sp)
        sw     $4,32($sp)
        sw     $5,36($sp)
        sw     $6,40($sp)
        sw     $7,44($sp)
        lw     $24,0+32($sp)
        lw     $15,4+32($sp)
        addu   $24,$24,$15
        lw     $15,8+32($sp)
        addu   $24,$24,$15
        lw     $15,12+32($sp)
        addu   $24,$24,$15
        sw     $24,-4+32($sp)
        lw     $4,-4+32($sp)
        jal   isOdd
L_2:    lw     $31,20($sp)
        addiu  $sp,$sp,32
        jr     $31
```

No stack funny business at all?



Address	Contents
\$sp+44	\$a3 = d
\$sp+40	\$a2 = c
\$sp+36	\$a1 = b
\$sp+32	\$a0 = a
\$sp+28	"sum"
\$sp+24	**unused**
\$sp+20	\$ra
\$sp+16	\$fp
\$sp+12	\$a3
\$sp+8	\$a2
\$sp+4	\$a1
\$sp →	\$a0

Back to our Recursive Example

Now let's make our example work, using the MIPS procedure linking and stack conventions.

```
int sqr(int x) {
    if (x > 1)
        x = sqr(x-1)+x+x-1;
    return x;
}
```

```
main()
{
    sqr(10);
}
```

Q: Why isn't the \$fp being used?
 A: Stack frame is remaining constant, so the compiler is making all accesses relative to \$sp.

```

sqr:   addiu   $sp, $sp, -32
       sw     $ra, 20($sp)
       sw     $s0, 24($sp)
       move  $s0, $a0
       addiu $t8, $0, 1
       slt   $1, $t8, $s0
       beq   $1, $0, endif
       addiu $a0, $s0, -1
       jal   sqr
       addu  $v0, $v0, $s0
       addu  $v0, $v0, $s0
       addiu $v0, $v0, -1
       b     rtn
endif: move  $v0, $s0
rtn:   lw     $s0, 24($sp)
       lw     $ra, 20($sp)
       addiu $sp, $sp, 32
       jr    $ra
    
```

Save registers that must survive the call.

Pass arguments

Restore saved registers.

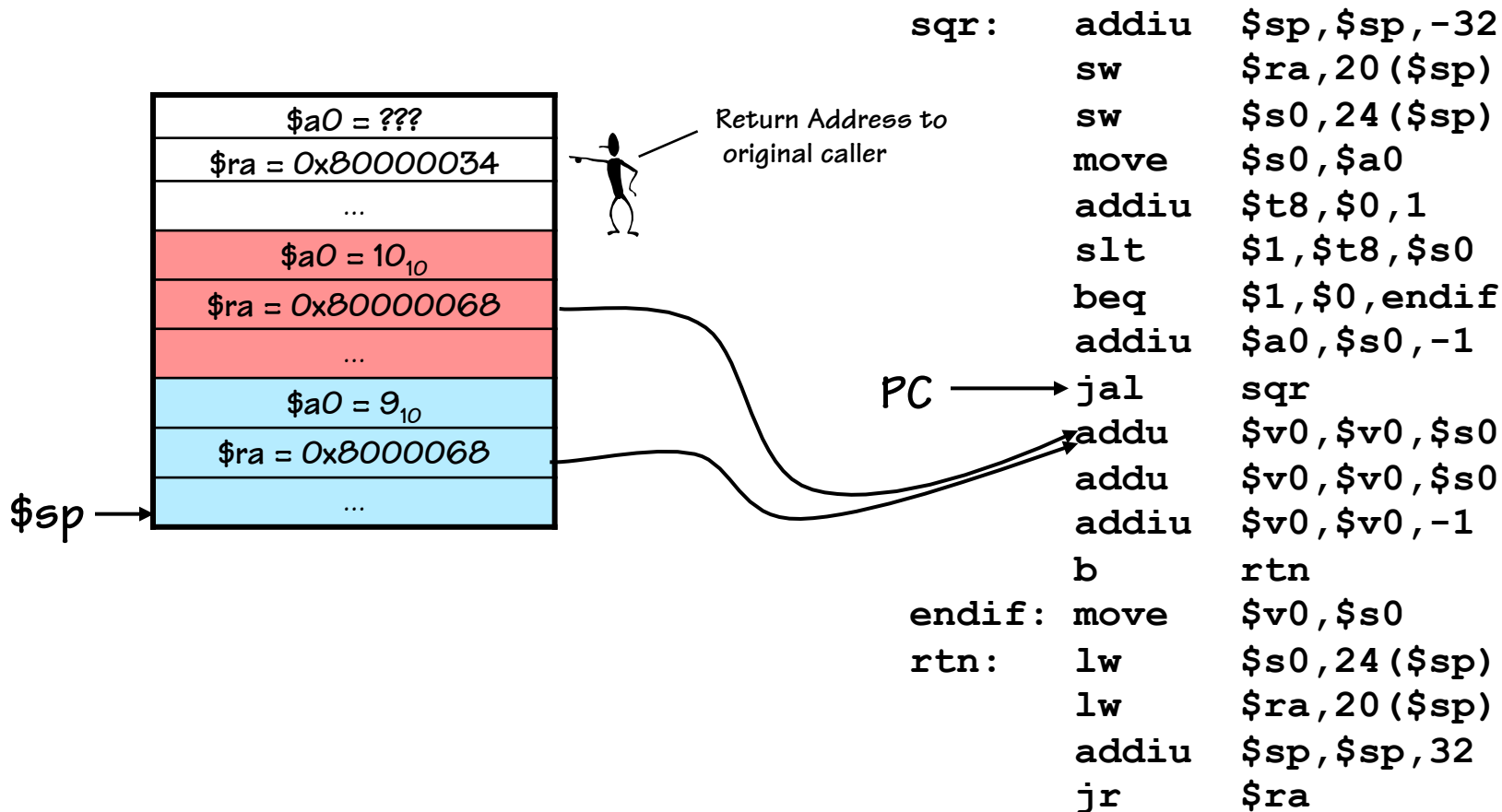
ALLOCATE stack frame. With room for the return address, a local saved register, and the argument spill for calls.

Address	Contents
\$ep+44	\$a3
\$ep+40	\$a2
\$ep+36	\$a1
\$ep+32	\$a0 = x
\$ep+28	**unused**
\$ep+24	\$s0
\$ep+20	\$ra
\$ep+16	\$fp
\$ep+12	\$a3
\$ep+8	\$a2
\$ep+4	\$a1
\$ep →	\$a0

DEALLOCATE stack frame.

Testing Reality's Boundaries

Now let's take a look at the active stack frames at some point during the procedure's execution.



Procedure Linkage is Nontrivial

The details can be overwhelming.

What's the solution for managing this complexity?

Abstraction!

- High-level languages can provide compact notation that hides the details.

We have another problem, there are great many CHOICES that we can make in realizing a procedure (which variables are saved, who saves them, etc.), yet we will want to design SOFTWARE SYSTEM COMPONENTS that interoperate. How did we enable composition in that case?

Contracts!

- But, first we must agree on the details?
Not just the HOWs, but WHENs.

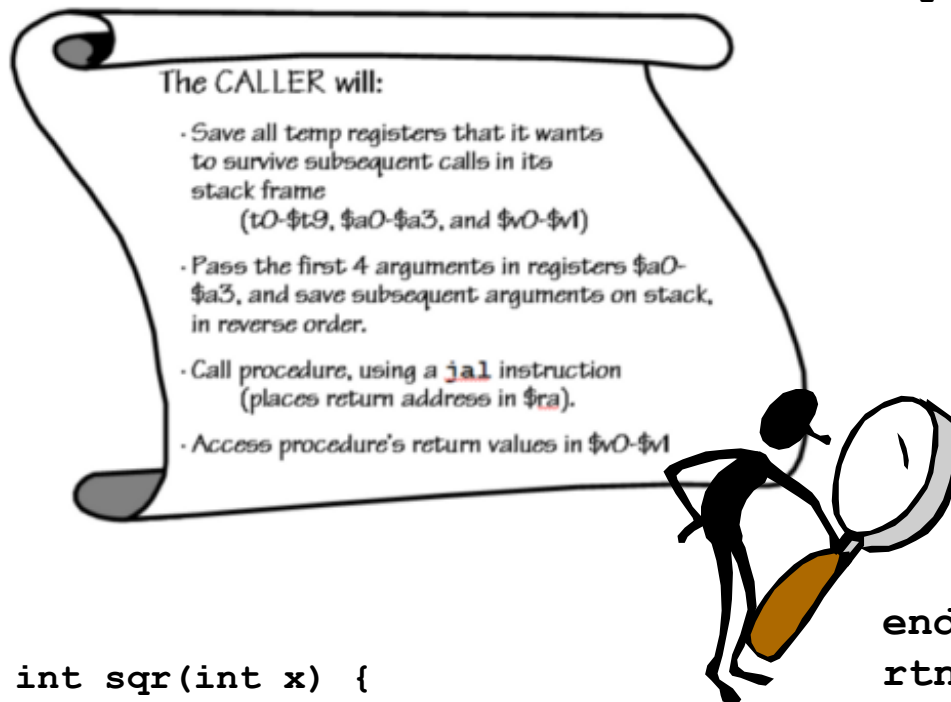
Procedure Linkage: Caller Contract

The CALLER will:

- Save all temp registers that it wants to survive subsequent calls in its stack frame
($t0-t9$, $a0-a3$, and $v0-v1$)
- Pass the first 4 arguments in registers $a0-a3$, and save subsequent arguments on stack, in *reverse* order.
- Call procedure, using a `jal` instruction (places return address in ra).
- Access procedure's return values in $v0-v1$

Code Lawyer

Our running example is a CALLER. Let's make sure it obeys its contractual obligations



```
int sqr(int x) {  
    if (x > 1)  
        x = sqr(x-1)+x+x-1;  
    return x;  
}
```

```
sqr:    addiu    $sp,$sp,-32  
       sw      $ra,20($sp)  
       sw      $s0,24($sp)  
       move   $s0,$a0  
       addiu  $t8,$0,1  
       slt   $1,$t8,$s0  
       beq   $1,$0,endif  
       addiu  $a0,$s0,-1  
       jal   sqr  
       addu  $v0,$v0,$s0  
       addu  $v0,$v0,$s0  
       addiu  $v0,$v0,-1  
       b     rtn  
endif: move   $v0,$s0  
rtn:    lw     $s0,24($sp)  
       lw     $ra,20($sp)  
       addiu  $sp,$sp,32  
       jr    $ra
```

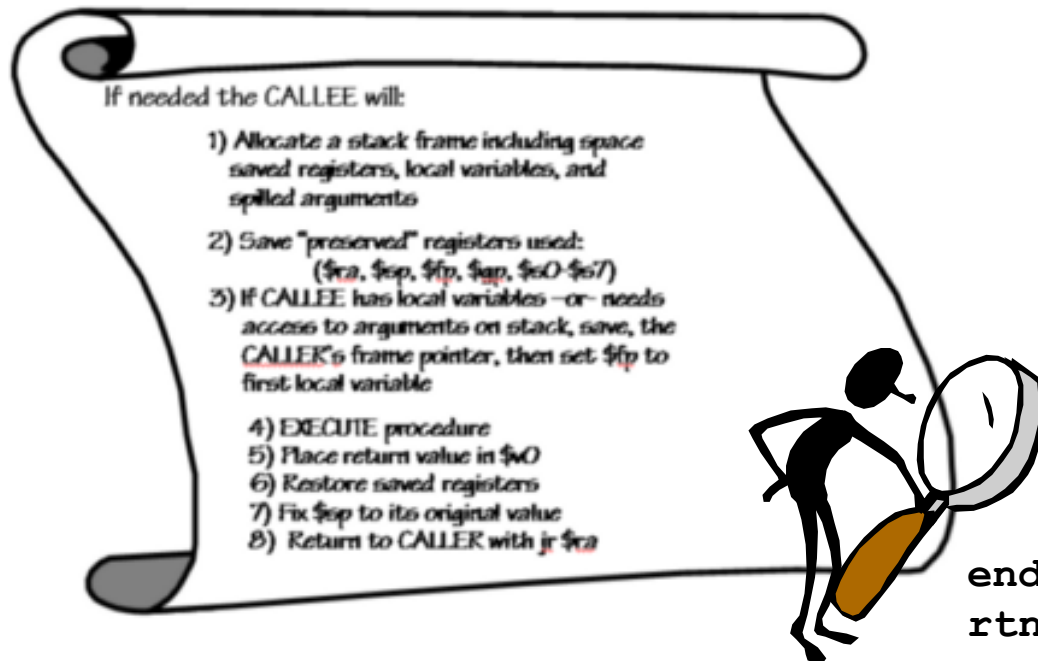
Procedure Linkage: Callee Contract

If needed the CALLEE will:

- 1) Allocate a stack frame including space for saved registers, local variables, and spilled arguments
- 2) Save any “preserved” registers used:
(\$ra, \$sp, \$fp, \$gp, \$s0-\$s7)
- 3) If CALLEE has local variables -or- needs access to arguments on the stack, save the CALLER’s frame pointer and set \$fp to 1st entry of the CALLEE’s stack
- 4) EXECUTE procedure
- 5) Place return value in \$v0
- 6) Restore saved registers
- 7) Fix \$sp to its original value
- 8) Return to CALLER with jr \$ra

More Legalese

Our running example is also a CALLEE. Are these contractual obligations satisfied?



```
int sqr(int x) {
    if (x > 1)
        x = sqr(x-1)+x+x-1;
    return x;
}
```

```
sqr:
    addiu $sp,$sp,-32
    sw    $ra,20($sp)
    sw    $s0,24($sp)
    move  $s0,$a0
    addiu $t8,$0,1
    slt   $1,$t8,$s0
    beq   $1,$0,endif
    addiu $a0,$s0,-1
    jal   sqr
    addu  $v0,$v0,$s0
    addu  $v0,$v0,$s0
    addiu $v0,$v0,-1
    b     rtn
endif:
    move  $v0,$s0
rtn:
    lw    $s0,24($sp)
    lw    $ra,20($sp)
    addiu $sp,$sp,32
    jr    $ra
```