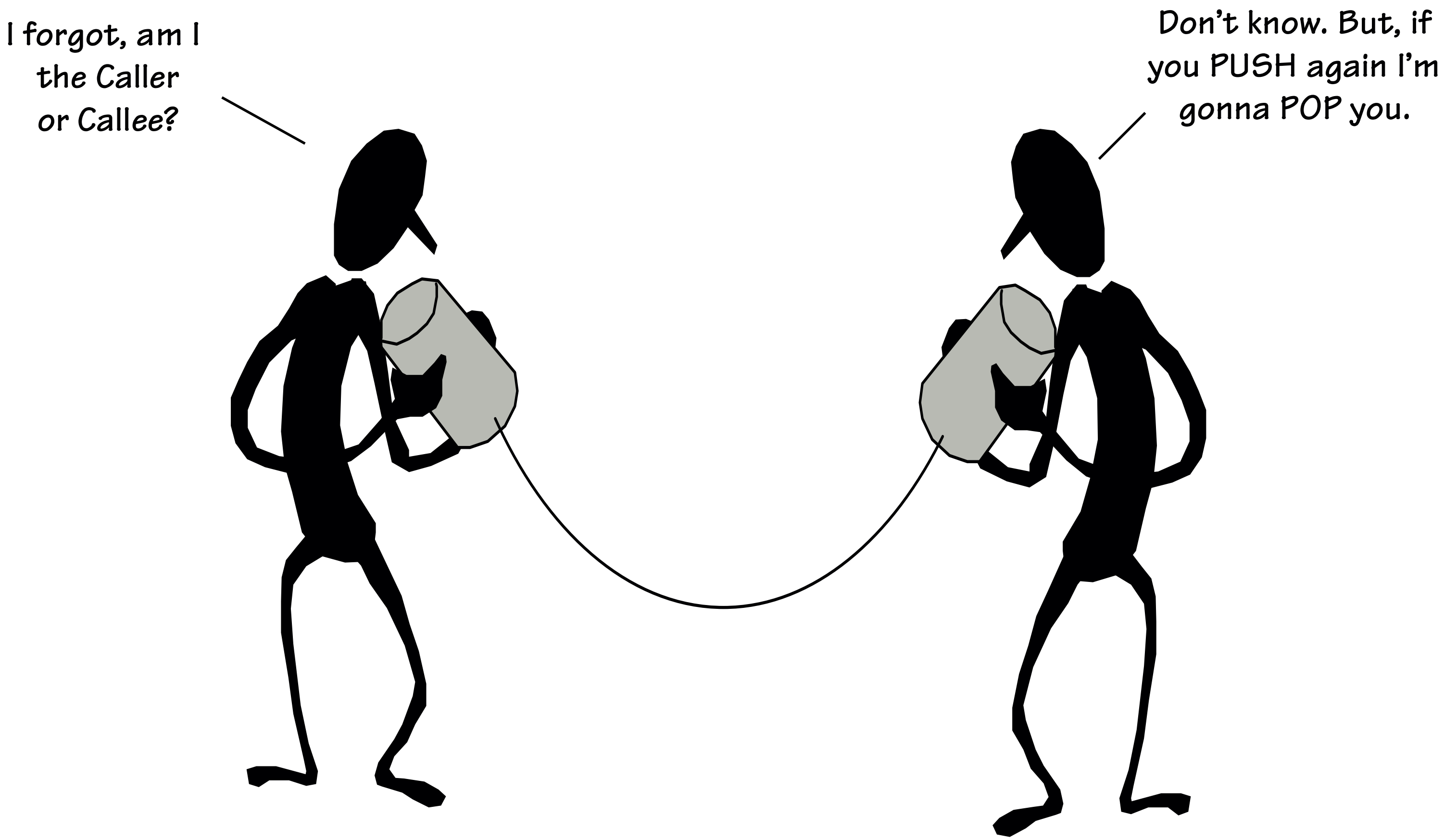
Stacks and Procedures



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

The Beauty of Procedures

- Reusable code fragments (modular design)

```
clear_screen();
  ...                  # code to draw a bunch of lines
clear_screen();
  ...
```

- Parameterized functions (variable behaviors)

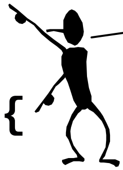
```
line(x1, y1, x2, y2, color);
line(x2,y2,x3,y3, color);
  ...
```

```
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);
```

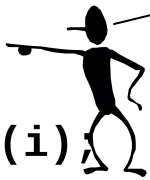
More Procedure Power

- Local scope (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"

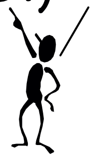
How do we keep track of all the variables

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):
  - 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:

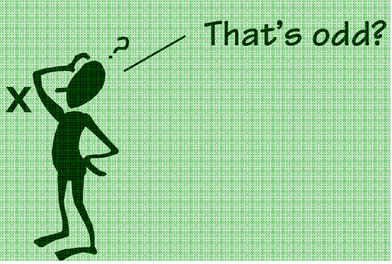
```
   .globl x
.data
x:   .word 9
```

Callee

```
.globl fee
.text
fee:
    add    $v0,$a0,$a0
    addi   $v0,$v0,-1
    j      $ra
```

Caller

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



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

This type of function 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?

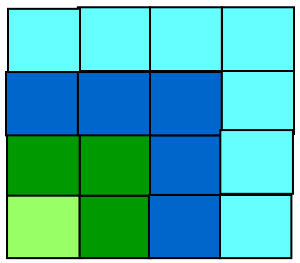
Let's consider the worst case of a Callee as a Caller…

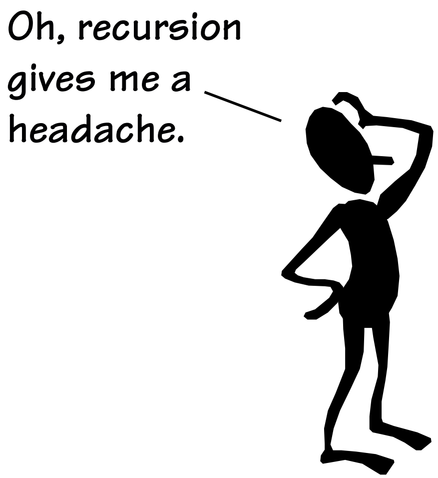
Writing Procedures

```
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).

sqr(10) = sqr(9)+10+10-1 = 100
sqr(9) = sqr(8)+9+9-1 = 81
sqr(8) = sqr(7)+8+8-1 = 64
sqr(7) = sqr(6)+7+7-1 = 49
sqr(6) = sqr(5)+6+6-1 = 36
sqr(5) = sqr(4)+5+5-1 = 25
sqr(4) = sqr(3)+4+4-1 = 16
sqr(3) = sqr(2)+3+3-1 = 9
sqr(2) = sqr(1)+2+2-1 = 4
sqr(1) = 1
sqr(0) = 0





Procedure Linkage: First Try

Callee/Caller

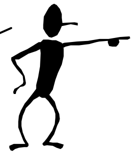
```
int sqr(int x) {
  if (x > 1)
    x = sqr(x-1)+x+x-1;
  return x;
}
```

Caller

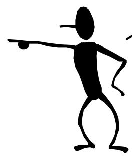
```
main()
{
  sqr(10);
}
```

$t0 is clobbered on successive calls.

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



```
sqr:   slti   $t0,$a0,2
       beq    $t0,$0,then   #!(x<2)
       move   $v0,$a0
       beq    $0,$0,rtn
then:
       add    $t0,$0,$a0
       addi   $a0,$a0,-1
       jal    sqr
       add    $v0,$v0,$t0
       add    $v0,$v0,$t0
       addi   $v0,$v0,-1
rtn:
       jr     $ra
```



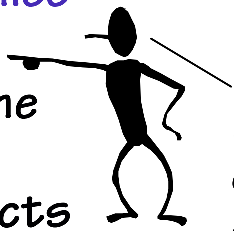
We also clobber our return address, so there's no way back!

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:

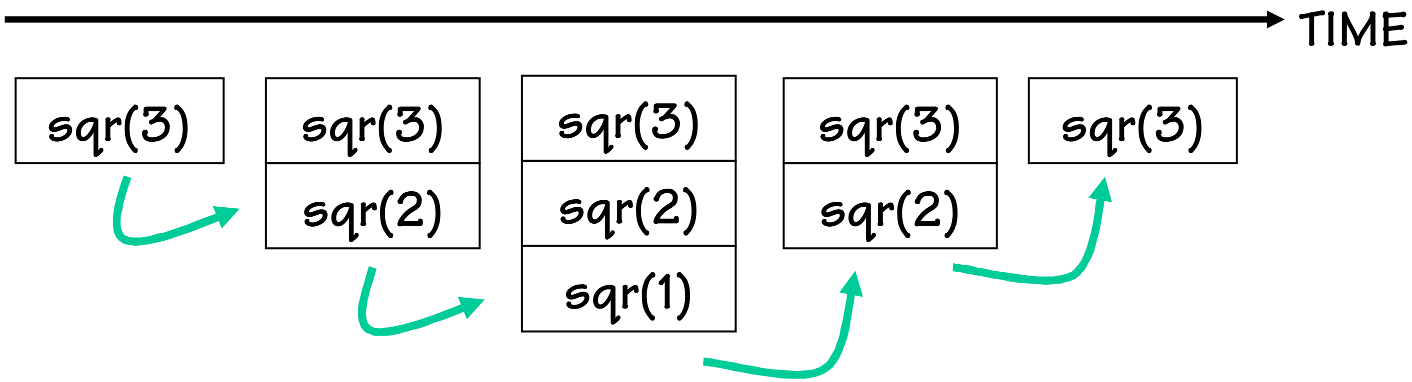
```
...
{
  int x, y;
  ... x ... y ...;
}
```

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.

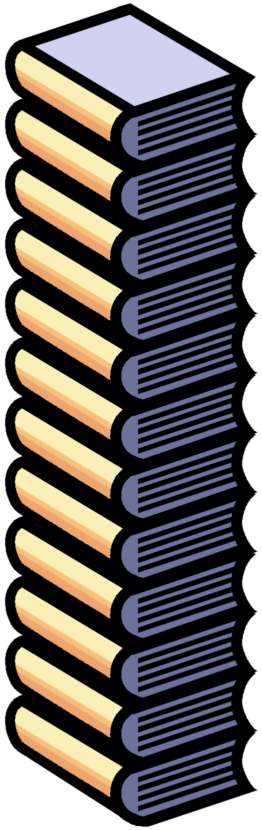
We Need Dynamic Storage!

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. Only the top is directly visible, the so-called
"top-of-stack"

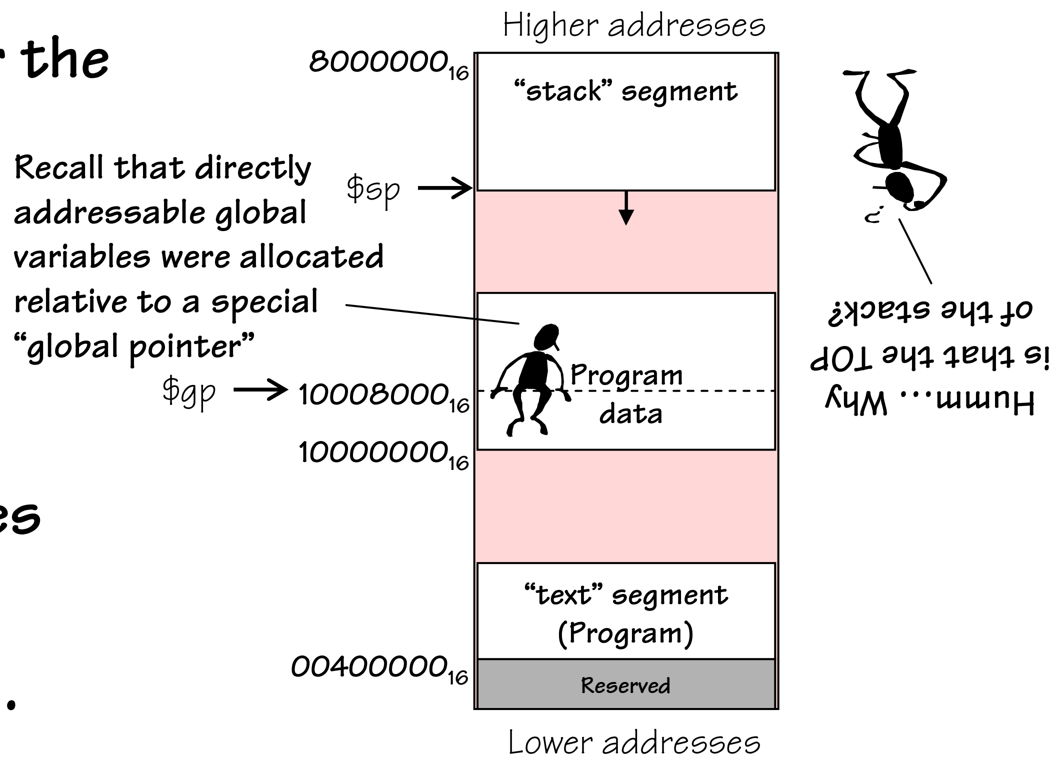
Add things by PUSHING new values on top.

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.
- Place stack far away from our 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

Reg[SP] = Reg[SP] - 4*k

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

DEALLOCATE k: release k WORDS of stack

Reg[SP] = Reg[SP] + 4*k

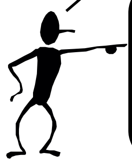
```
addi $sp,$sp,4*k
```

PUSH rx: push Reg[x] onto stack

Reg[SP] = Reg[SP] - 4

Mem[Reg[SP]] = Reg[x]

An ALLOCATE 1 followed by a store



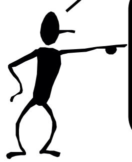
```
addi $sp,$sp,-4
sw   RX, 0($sp)
```

POP rx: pop the value on the top of the stack into Reg[x]

Reg[x] = Mem[Reg[SP]]

Reg[SP] = Reg[SP] + 4;

A load followed by a DEALLOCATE 1

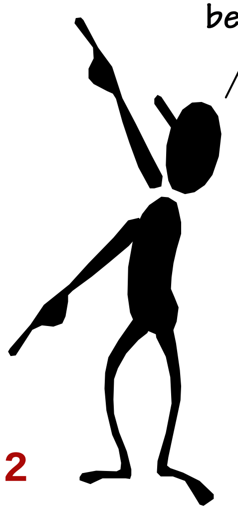


```
lw   RX, 0($sp)
addi $sp,$sp,4
```

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!!
            #
            addi   $sp,$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:   addi   $s1,$s1,-1
            bne    $s1,$0,suspense
            sw     $s0,warp_engines
            lw     $s0,-4($sp)       # Restore s0
            lw     $s1,0($sp)        # Restore s1
            addi   $sp,$sp,8         # deallocate 2
```



AND Stacks can also be used to solve other problems…

Solving Procedure Linkage "Problems"

In case you forgot, a reminder of our problems:

1) We need a way to pass arguments into procedures
2) Procedures need storage for their LOCAL variables
3) Procedures need to call other procedures
4) Procedures might call themselves (Recursion)

BUT FIRST, WE'LL WASTE SOME MORE REGISTERS:

\$30 = \$fp.	Frame ptr, points to the callee's local variables on the stack, we also use it to access extra args (>4)
\$31 = \$ra.	Return address back to caller
\$29 = \$sp.	Stack ptr, points to "TOP" of stack

Now we can define a STACK FRAME
(a.k.a. the procedure's Activation Record):

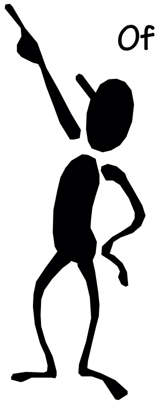
More MIPS Procedure Conventions

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.
(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.)



Of course, the MIPS convention is to do this case.

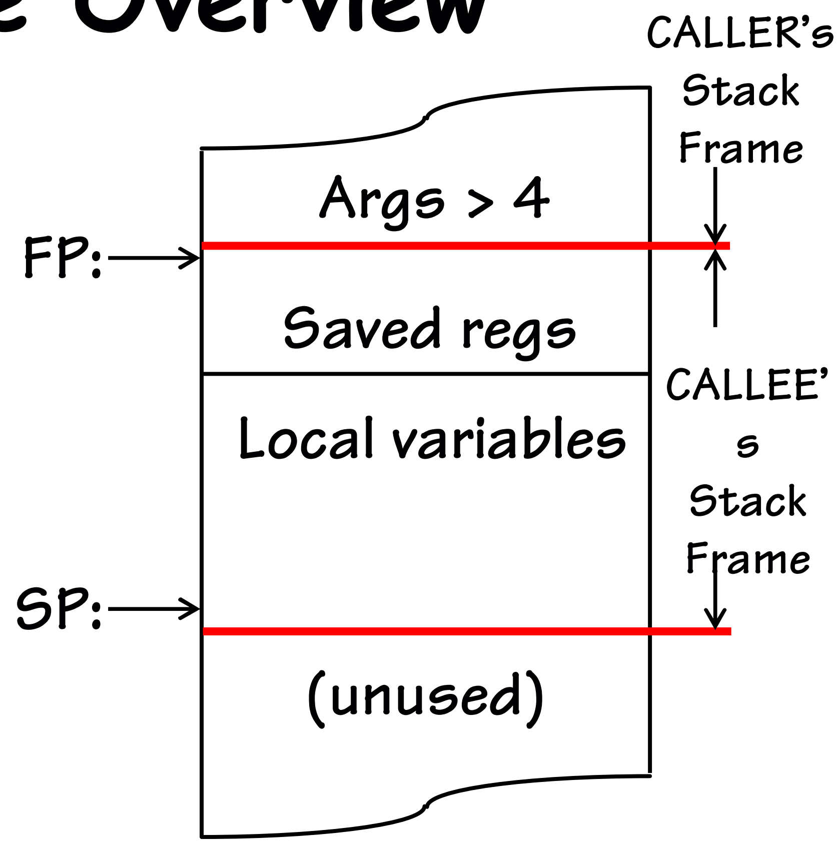
Stack Frame Overview

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 5^{th} 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 5^{th} argument (if used) passed in from the CALLER. The FRAME POINTER keeps track of this boundary between stack frames.



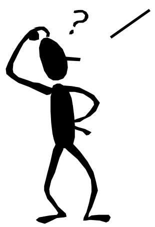
It's possible to use only the SP to access a stack frame, but offsets may change due to ALLOCATEs and DEALLOCATEs. For convenience a $fp is used to provide CONSTANT offsets to local variables and arguments

Procedure Stack Usage

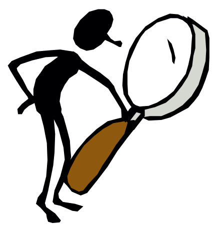
ADDITIONAL space must be allocated in the stack frame for:

1. Any SAVED registers the procedure uses ($s0-$s7)
2. Any TEMPORARY registers that the procedure wants preserved IF it calls other procedures ($t0-$t9)
3. Any LOCAL variables declared within the procedure
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 words.
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.



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 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

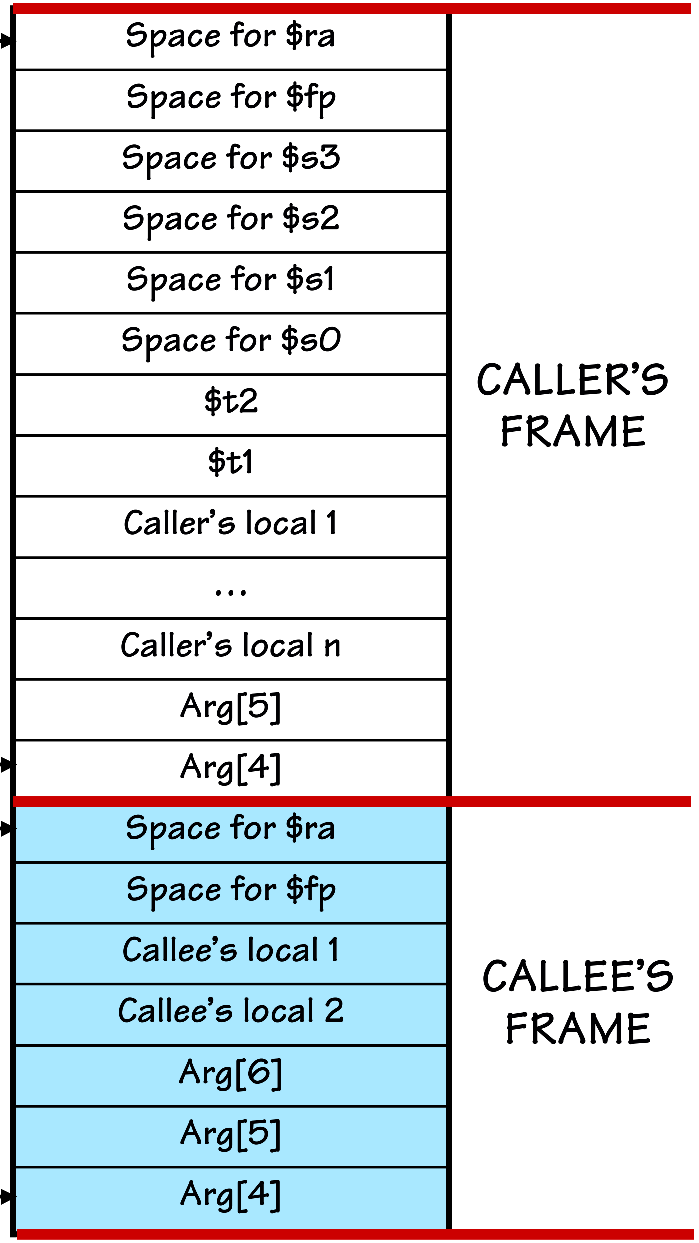
Stack Snap Shots

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

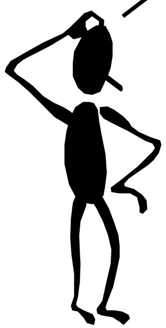
Where in the CALLEE's stack frame might one find the CALLER's $fp? It MIGHT be at -4($fp)



Back to Reality

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

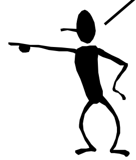
```
int sqr(int x) {
  if (x > 1)
    x = sqr(x-1)+x+x-1;
  return x;
}

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



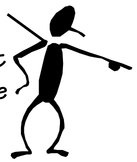
Q: Why didn't we save and update $fp?

A: Don't have local variables or spilled args.

```
sqr:  addiu  $sp,$sp,-8
      sw     $ra,4($sp)
      sw     $a0,0($sp)
      slti   $t0,$a0,2
      beq    $t0,$0,then
      move   $v0,$a0
      beq    $0,$0,rtn

then:
      addi   $a0,$a0,-1
      jal    sqr
      lw     $a0,0($sp)
      add    $v0,$v0,$a0
      add    $v0,$v0,$a0
      addi   $v0,$v0,-1
rtn:
      lw     $ra,4($sp)
      addiu  $sp,$sp,8
      jr     $ra
```

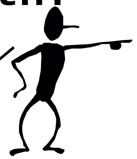


ALLOCATE minimum stack frame. With room for the return address and the passed in argument.

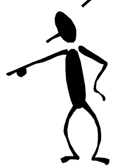


Save registers that must survive the call.

Pass arguments



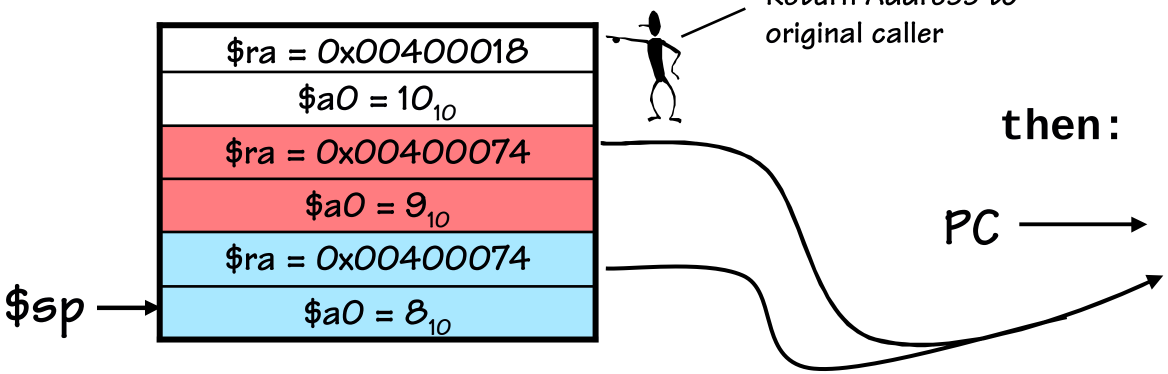
Restore saved registers.



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.




```
sqr:    addiu   $sp,$sp,-8
        sw      $ra,4($sp)
        sw      $a0,0($sp)
        slti    $t0,$a0,2
        beq     $t0,$0,then
        move    $v0,$a0
        beq     $0,$0,rtn

then:
        addi    $a0,$a0,-1
        jal     sqr
        lw      $a0,0($sp)
        add     $v0,$v0,$a0
        add     $v0,$v0,$a0
        addi    $v0,$v0,-1
rtn:
        lw      $ra,4($sp)
        addiu   $sp,$sp,8
        jr      $ra
```

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 settle on the details? Not just the HOWs, but WHENs.

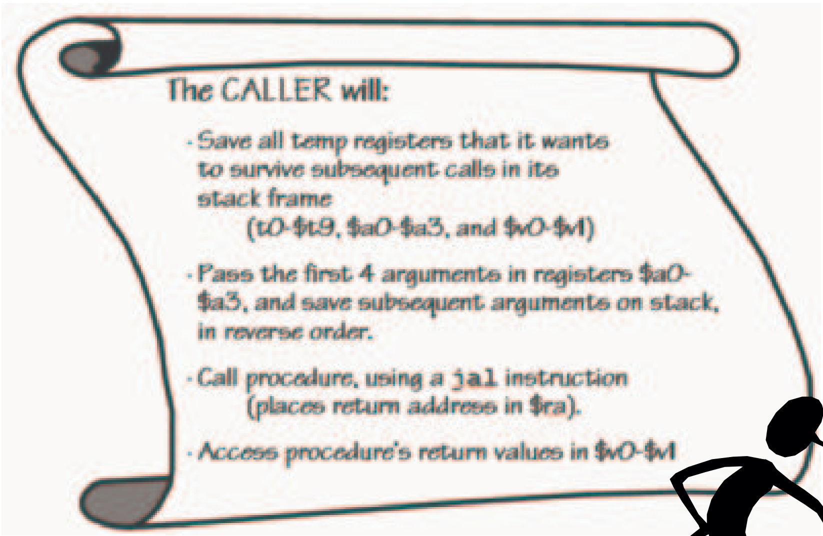
Procedure Linkage: Caller Contract

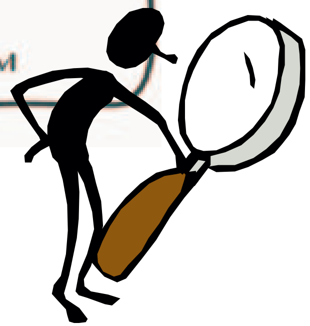
The CALLER will:

- 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





```
sqr:    addiu   $sp,$sp,-8
        sw      $ra,4($sp)
        sw      $a0,0($sp)
        slti    $t0,$a0,2
        beq     $t0,$0,then
        move    $v0,$a0
        beq     $0,$0,rtn
then:
        addi    $a0,$a0,-1
        jal     sqr
        lw      $a0,0($sp)
        add     $v0,$v0,$a0
        add     $v0,$v0,$a0
        addi    $v0,$v0,-1
rtn:
        lw      $ra,4($sp)
        addiu   $sp,$sp,8
        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

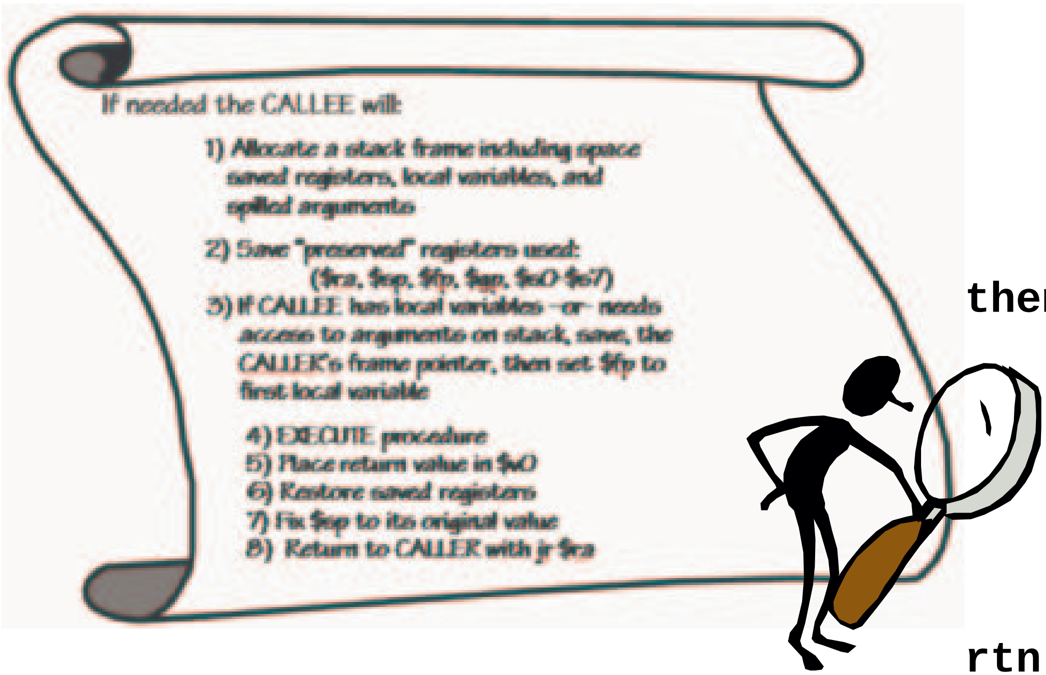
2) Save any "preserved" registers used:
   ($ra, $sp, $fp, $gp, $s0-$s7)

3) If CALLEE has local variables -or- needs access to arguments on the stack, save the CALLER's frame pointer and set $fp to 1^{st} 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 satified?



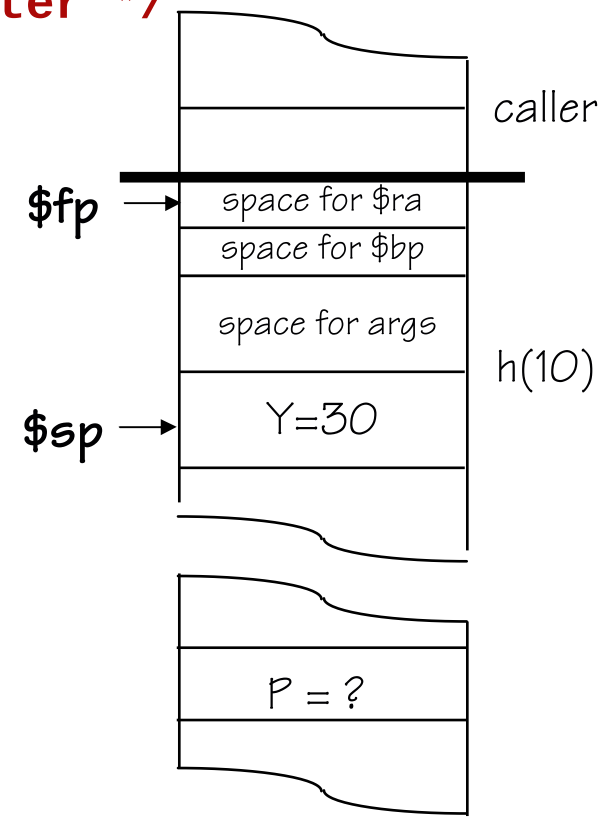

```
sqr:    addiu   $sp,$sp,-8
        sw      $ra,4($sp)
        sw      $a0,0($sp)
        slti    $t0,$a0,2
        beq     $t0,$0,then
        move    $v0,$a0
        beq     $0,$0,rtn
then:
        addi    $a0,$a0,-1
        jal     sqr
        lw      $a0,0($sp)
        add     $v0,$v0,$a0
        add     $v0,$v0,$a0
        addi    $v0,$v0,-1
rtn:
        lw      $ra,4($sp)
        addiu   $sp,$sp,8
        jr      $ra
```

On Last Point: Dangling References

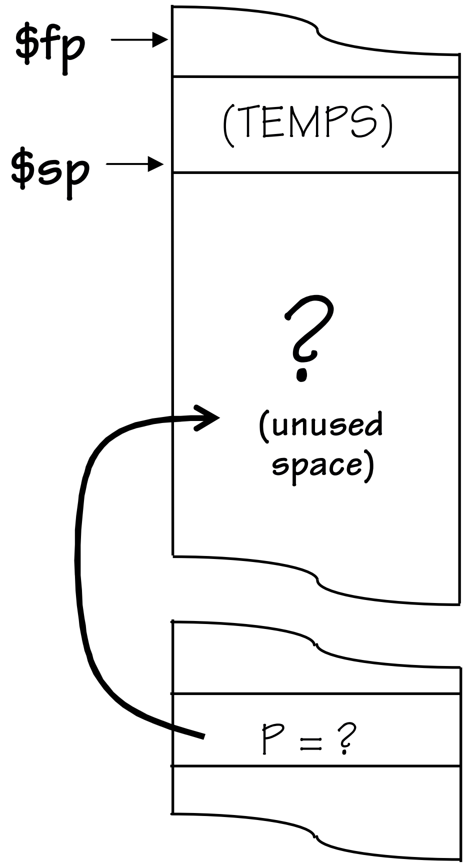
Stacks can be an unreliable place to put things....

```
int *p; /* a pointer */

int h(x)
{
    int y = x*3;
    p = &y;
    return 37;
}

h(10);
print(*p);
```

What do we expect to be printed?



"During Call"



"After Call"

Dangling Reference Solutions

Java & PASCAL: Kiddy scissors only.

No "ADDRESS OF" operator: language restrictions *forbid* constructs which could lead to dangling references.

C and C++: real tools, real dangers.

"You get what you deserve".

SCHEME/LISP: throw cycles at it.

Activation records allocated from a HEAP, reclaimed transparently by garbage collector (at considerable cost).

"You get what you pay for"

Of course, there's a stack hiding there somewhere...