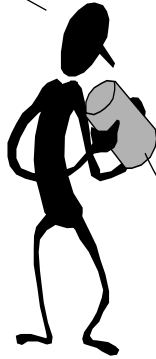
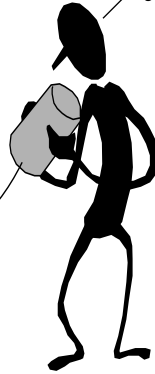


Stacks and Procedures

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.

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).

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

Oh, recursion gives me a headache.



Hints From Register Usage

The SYMBOLIC register names give some hints for writing procedures.

The MIPS REGISTER USAGE conventions designate registers for arguments, return values, return addresses, etc.

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

Procedure Linkage: First Try

Callee/Caller

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

```
sqr: slti    $t0,$a0,2
     beq     $t0,$0,else
     move   $v0,$a0
     beq    $0,$0,rtn
```

Caller

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

OOPS!
 \$t0 is clobbered on successive calls. Will saving "x" at some fixed location in memory help? (Nope)

```
else:
    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 ...;
}
    
```

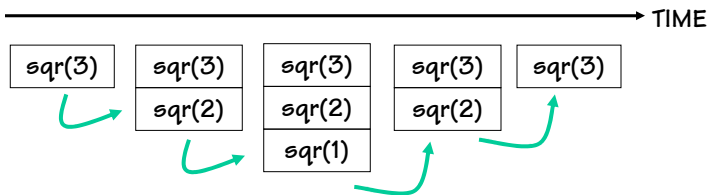
Each of these is specific to a "particular" invocation or *activation* of the Callee. Collectively, the arguments passed in, the return address, and the callee's local variables are its *activation record*, or *call frame*, or *stack frame*.

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"

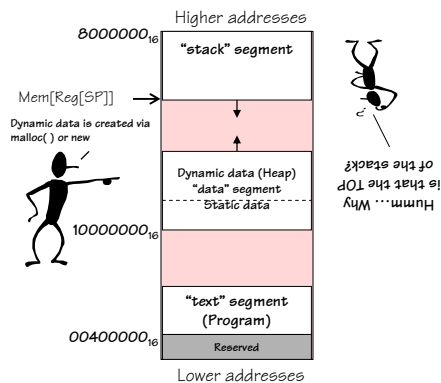
Add things by PUSHING new values on top.

Remove things by POPPING off values.

MIPS Stack Convention

CONVENTIONS:

- Waste a register for the Stack Pointer ($\$sp = \29).
- Builds DOWN (towards lower addresses) on pushes and allocates
- SP points to the TOP 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 Macros

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 RX: 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  RX, 0($sp)
```

POP RX: 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  RX, 0($sp)
addiu $sp,$sp,-4
```

Fun with Stacks

We can squirrel away variables for latter. 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:
addi  $s1,$s1,-1
bne   $s1,$0,suspense
sw    $s0,warp_engines
lw    $s0,4($sp)     # Restore s0
lw    $s1,0($sp)     # Restore s1
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...

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
- \$31 = **\$ra**. Return address 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):

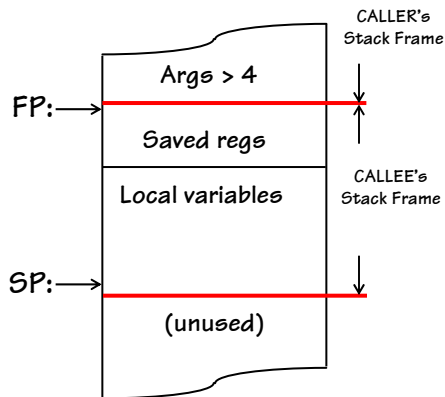
Stack Frame Overview

The STACK FRAME contains storage for the CALLER's state that is 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.



In theory it's possible to use SP to access stack frame, but offsets will change due to ALLOCATEs and DEALLOCATEs. For convenience \$fp is used to provide CONSTANT offsets to local variables and arguments

MIPS Stack Frame Details

EVERY stack frame must have at LEAST 6 words (24 bytes).

This REQUIRED stack frame includes space for

1. `$ra`, the return address, used only if the callee is itself a caller
2. Caller's `$fp`, used only if the callee needs its own frame pointer (i.e. it has local variables or needs access to arguments on stack)
3. Storage for the 4 "incoming" argument registers, `$a0-$a3`, (used if callee is caller and args need to be preserved, OR if the address of an argument is needed).

Stack Frames are extended by multiples of 8 bytes (2-words)

```
int foo(int x)
{
    int *a = &x;
    return (int) a;
}
```



We need to be able to compute the address of any argument. Thus, we need memory locations to keep them in, even if they are never used.

More MIPS Stack Frame Details

ADDITIONAL space must be allocated in the stack frame for:

1. Those SAVED registers the procedure uses (`$s0-$s7`)
2. TEMPORARY registers the procedure wants preserved IF it calls other procedures (`$t0-$t9`)
3. LOCAL variables declared in the procedure
4. Other TEMP space IF the procedure runs out of registers (RARE)
5. Enough "outgoing" arguments to satisfy the worst 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 this 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

Stack Snap Shots

Shown on the right is a snap shot of the stack's 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)

CALLER's \$fp →

\$sp (prior to call)

CALLEE's \$fp

\$sp (after call) →

Space for \$ra	CALLER'S FRAME
Space for \$fp	
Space for \$a3	
Space for \$a2	
Space for \$a1	
Space for \$a0	
\$s3	
\$t1	
Caller's local 1	
...	
Caller's local n	
Arg[5]	CALLEE'S FRAME
Arg[4]	
Space for \$ra	
Space for \$fp	
Space for \$a3	
Space for \$a2	
Space for \$a1	
Space for \$a0	
Callee's local 1	

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.

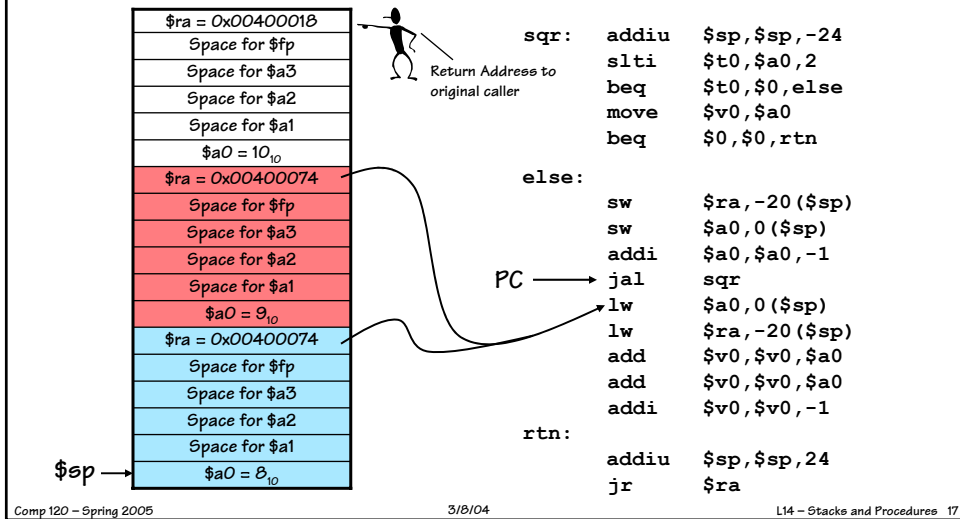
```
sqr:    addiu   $sp,$sp,-24
        slti    $t0,$a0,2
        beq    $t0,$0,else
        move   $v0,$a0
        beq    $0,$0,rtn

else:
        sw     $ra,-20($sp)
        sw     $a0,0($sp)
        Pass arguments →
        addi   $a0,$a0,-1
        jal   $ra,$sp
        lw     $a0,0($sp)
        lw     $ra,-20($sp)
        add    $v0,$v0,$a0
        add    $v0,$v0,$a0
        addi   $v0,$v0,-1
        Restore saved registers.
        DEALLOCATE stack frame.

rtn:
        addiu  $sp,$sp,24
        jr    $ra
        ALLOCATE minimum stack frame.
        Save registers that must survive the call.
```

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 of software, like hardware, 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, yet we will want to design SOFTWARE SYSTEM COMPONENTS that interoperate, much like combinational, and sequential logic blocks. How did we insure 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 the call in its stack frame (\$a0-\$a3, \$t0-\$t9, and \$ra)
- 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).
- Restore saved temp registers

Code Inspector

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

The CALLER will:

- Save all temp registers that it wants to survive the call in its stack frame (\$a0-\$a3, \$t0-\$t9, and \$ra)
- 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).
- Restore saved temp registers

```
sqr:  addiu  $sp,$sp,-24
      slti  $t0,$a0,2
      beq   $t0,$0,else
      move  $v0,$a0
      beq   $0,$0,rtn
```

```
else:
      sw    $ra,-20($sp)
      sw    $a0,0($sp)
      addi  $a0,$a0,-1
      jal   sqr
      lw    $a0,0($sp)
      lw    $ra,-20($sp)
      add   $v0,$v0,$a0
      add   $v0,$v0,$a0
      addi  $v0,$v0,-1
```

```
rtn:
      addiu $sp,$sp,24
      jr    $ra
```

Procedure Linkage: Callee Contract

The CALLEE will:

- Allocate a stack frame of at least 6 words adding extra space (multiples of 2 words) for:

- 1) CALLEE saved registers (\$s0-\$s7)
- 2) Space for temps if CALLER (\$t0-\$t9)
- 3) Local variables

- If CALLEE has local variables or needs access to arguments on stack, save CALLER's frame pointer, and set \$fp to \$sp + (stack frame size minus 4)

- EXECUTE procedure
- Place return value in \$v0
- Restore CALLEE saved registers
- Add stack frame size to \$sp
- Return to CALLER with jr \$ra

More Inspections

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

The CALLEE will:

- Allocate a stack frame of at least 6 words adding extra space (multiples of 2 words) for:

- 1) CALLEE saved registers (\$s0-\$s7)
- 2) Space for temps if CALLER (\$t0-\$t9)
- 3) Local variables

- If CALLEE has local variables or needs access to arguments on stack, save CALLER's frame pointer, and set \$fp to \$sp + (stack frame size minus 4)

- EXECUTE procedure
- Place return value in \$v0
- Restore CALLEE saved registers
- Add stack frame size to \$sp
- Return to CALLER with jr \$ra



```
sqr:  addiu $sp, $sp, -24
      slti $t0, $a0, 2
      beq $t0, $0, else
      move $v0, $a0
      beq $0, $0, rtn
```

```
else:
      sw $ra, -20($sp)
      sw $a0, 0($sp)
      addi $a0, $a0, -1
      jal sqr
      lw $a0, 0($sp)
      lw $ra, -20($sp)
      add $v0, $v0, $a0
      add $v0, $v0, $a0
      addi $v0, $v0, -1
rtn:
```

```
      addiu $sp, $sp, 24
      jr $ra
```

Remaining Tough Problems

1. NON-LOCAL variable access, particularly in nested procedure definitions.

"FUNarg" problem of LISP.

Conventional solution: "static links" in stack frames, pointing to frames of statically enclosing blocks. This allows a run-time discipline which correctly accesses variables in enclosing blocks.

ANALOG: LISP Environments, closures.

(C avoids this problem by outlawing nested procedure declarations!)

2. "Dangling References" - - -

```

int x, y, z;

g(int x) {
  int z;
  f(int x) {
    int y;
    ...
    z = x * y;
  }
  ...
  f(4);
}
    
```

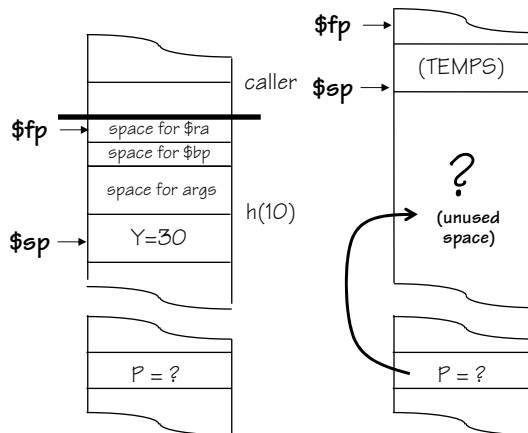
Dangling References

```

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?

The Word on Dangling References

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...