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:

- 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
- sge $d,$s,$t becomes slt $d,$t,$s
- nop becomes sll $0,$0,0

Do Nothing

Why both?

Which, BTW, assembles to 0x00000000
Uber Pseudoinstruction

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

1) \texttt{la \$d, offset($base)}
2) \texttt{la \$d, offset}
3) \texttt{la \$d, ($base)}

It mimics the format of \texttt{lw/sw} instructions, but rather than reading/writing the contents of memory, it loads it destination register with the “effective address” that would have been accessed.

As a result it can generate any one of the following five sequences:

1) \texttt{lui \$d,offset}
   \texttt{ori \$d,$d,offset}
2) \texttt{ori \$d,$0,offset}
3) \texttt{addiu \$d,$base,$1}

The MIPS compiler loves this pseudoinstruction. It often uses it to load a constant into a register.

The MIPS compiler loves this pseudoinstruction. It often uses it to load a constant into a register.
The Beauty of Procedures

- Reusable code fragments (modular design)
  
  ```
  clear_screen();
  ...
  # code to draw a bunch of lines
  clear_screen();
  ...
  ```

- Parameterized procedures (variable behaviors)
  
  ```
  line(x_1, y_1, x_2, y_2, color);
  line(x_2, y_2, x_3, y_3, color);
  ...
  ```

- Functions (procedures that return values)
  
  ```
  xMax = max(max(x_1, x_2), x_3);
  yMax = max(max(y_1, y_2), y_3);
  ```

  ```
  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)**

```c
int x = 9;  // These are different “x”s

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));
}
```

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, and provide a “link” back

• A “called” procedure (Callee) must:
  – Acquire/create resources needed to perform the function (local variables, registers, etc.)
  – Perform the function
  – Place results in a place where the Caller can find them
  – Return control back to the Caller through the supplied link

• 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 part #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 jr $31 or jr $ra instruction.

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>assembler temporary</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>procedure return values</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>procedure arguments</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved by callee</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>more temporaries</td>
</tr>
<tr>
<td>$k0-$k1</td>
<td>26-27</td>
<td>reserved for operating system</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
</tr>
</tbody>
</table>
And It “Sort Of” Works

- Example:
  .globl x
  .data
  x: .word 9

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

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

Works for cases where the Callees need few resources and call no other functions.

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

But there are still a few issues:
- How can a Callee 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…
Recursion! A Callee who calls itself!

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

```c
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.
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
     addu $t0,$0,$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.

Caller
main()
{
    sqr(10);
}

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

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

We also clobber our return address, so there’s no way back!
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 a “return address” to get back
- **Callee** saves stuff that **Caller** expects to remain unchanged
- **Callee** executes
- **Callee** passes results back to **Caller**.

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**.
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:

• Dedicate a register for the Stack Pointer ($sp = $29).

• Stack grows DOWN (towards lower addresses) on pushes and allocates

• $sp points to the last or 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
  \[ \text{Reg}[SP] = \text{Reg}[SP] - 4 \times k \]

**DEALLOCATE**  $k$: release $k$ WORDS of stack
  \[ \text{Reg}[SP] = \text{Reg}[SP] + 4 \times k \]

**PUSH**  $x$: push Reg[$x$] onto stack
  \[ \text{Reg}[SP] = \text{Reg}[SP] - 4 \]
  \[ \text{Mem}[\text{Reg}[SP]] = \text{Reg}[x] \]

**POP**  $x$: pop the value on the top of the stack into Reg[$x$]
  \[ \text{Reg}[x] = \text{Mem}[\text{Reg}[SP]] \]
  \[ \text{Reg}[SP] = \text{Reg}[SP] + 4; \]

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

```
addiu $sp,$sp,-4\times k
addiu $sp,$sp,4\times k
```

An ALLOCATE 1 followed by a store

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

A load followed by a DEALLOCATE 1
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.

```assembly
#  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
```

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

You should ALWAYS allocate prior to saving, and deallocate after restoring in order to be SAFE!
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.)
Stack Frame Overview

The STACK FRAME contains storage for the CALLER’s volatile state that must be preserved after the invocation of CALLEES. In addition, the CALLEE uses the stack for:

1) Accessing the arguments that the CALLER passes to it (specifically, the 5\textsuperscript{th} and greater)
2) Saving non-temporary registers that it needs 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 1\textsuperscript{st} argument passed in from the CALLER. A FRAME POINTER, $fp$, (if needed) keeps track of this boundary between stack frames. It also tracks where local variables are allocated.

It’s possible to use only the SP to access a stack frame, but the 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

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

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 memory contents “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.

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

Shown on the right is a snapshot 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+8]**
Simple Cases

A leaf needing minimal resources:

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

Assembly code:

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

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

A function that calls others and has local variables:

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

Assembly code:

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

No stack funny business at all?
Simple Cases

A leaf needing minimal resources:

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

A function that calls others and has local variables:

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

Assembly code:

```
isOdd:  andi    $2,$4,1
    L_1:   jr      $31
parity: addiu   $sp,$sp,-32
    sw      $31,20($sp)
    sw      $4,0+32($sp)
    sw      $5,4+32($sp)
    sw      $6,8+32($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
    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?

Contents

<table>
<thead>
<tr>
<th>$sp</th>
<th>$a3 = parity.d</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a2 = parity.c</td>
</tr>
<tr>
<td></td>
<td>$a1 = parity.b</td>
</tr>
<tr>
<td></td>
<td>$a0 = parity.a</td>
</tr>
</tbody>
</table>

Notice that the caller allocates space for the first four arguments to functions that it might call in its stack frame, even though it never pushes values into them. However, the callee can safely save them if needed.
Optimized Simple Case

A leaf needing minimal resources:

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

A function that calls others and has local variables:

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

Optimized assembly code:

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

parity: addiu   $sp,$sp,-32
        sw      $31,20($sp)
        addu    $24,$4,$5
        addu    $24,$24,$6
        addu    $4,$24,$7
        sw      $4,-4+32($sp)
        jal     isOdd
        lw      $31,20($sp)
        addiu   $sp,$sp,32
        jr      $31
        L_1:
        jr      $31
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>$sp+44</td>
<td>$a3</td>
</tr>
<tr>
<td>$sp+40</td>
<td>$a2</td>
</tr>
<tr>
<td>$sp+36</td>
<td>$a1</td>
</tr>
<tr>
<td>$sp+32</td>
<td>$a0</td>
</tr>
<tr>
<td>$sp+28</td>
<td>“sum”</td>
</tr>
<tr>
<td>$sp+24</td>
<td><strong>unused</strong></td>
</tr>
<tr>
<td>$sp+20</td>
<td>$ra</td>
</tr>
<tr>
<td>$sp+16</td>
<td><strong>unused</strong></td>
</tr>
<tr>
<td>$sp+12</td>
<td>$a3</td>
</tr>
<tr>
<td>$sp+8</td>
<td>$a2</td>
</tr>
<tr>
<td>$sp+4</td>
<td>$a1</td>
</tr>
<tr>
<td>$sp</td>
<td>$a0</td>
</tr>
</tbody>
</table>

Callee doesn’t save its arguments because, they are not referenced again after the call to isOdd(sum). But it still needs to allocate space for local variables (i.e. sum), the return address, and any arguments that callees might need storage for.
Back to our Recursive Example

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

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

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

### Stack Frame

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

### Code Example

```
sqr:  addiu $sp,$sp,-32
      sw  $ra,20($sp)
      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
```

### Address Contents

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>$sp+43</td>
<td>$a3</td>
</tr>
<tr>
<td>$sp+40</td>
<td>$a2</td>
</tr>
<tr>
<td>$sp+36</td>
<td>$s1</td>
</tr>
<tr>
<td>$sp+32</td>
<td>$a0 = x</td>
</tr>
<tr>
<td>$sp+28</td>
<td>&quot;unused&quot;</td>
</tr>
<tr>
<td>$sp+24</td>
<td>$v0</td>
</tr>
<tr>
<td>$sp+20</td>
<td>$ra</td>
</tr>
<tr>
<td>$sp+16</td>
<td>$v0</td>
</tr>
<tr>
<td>$sp+12</td>
<td>$a3</td>
</tr>
<tr>
<td>$sp+8</td>
<td>$a2</td>
</tr>
<tr>
<td>$sp+4</td>
<td>$s1</td>
</tr>
<tr>
<td>$sp</td>
<td>$a0</td>
</tr>
</tbody>
</table>

Q: Why isn't the $fp being used?
A: Stack frame size within a call remains constant, so the compiler makes all accesses relative to $sp.

### Diagram

- **Save registers that must survive the call.**
- **Pass arguments.**
- **Restore saved registers.**
Testing Reality's Boundaries

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

```assembly
  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 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 any temp registers that it needs after calls in its stack frame (t0-$t9, $a0-$a3, $v0-$v1, and $ra)
- Pass the first 4 arguments in registers $a0-$a3, and save subsequent arguments on stack, in *reversed* order.
- Call procedure, using a jal instruction (places return address in $ra).
- Access procedure’s return values in $v0-$v1
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;
}
```

The CALLER will:
- Save any temp registers that it needs after calls in its stack frame (t0-t9, $a0-$a3, $v0-$v1, and $ra)
- Pass the first 4 arguments in registers $a0-$a3, and save subsequent arguments on stack, in “reversed” order.
- Call procedure, using a jal instruction (places return address in $ra).
- Access procedure’s return values if $.

```
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
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 local variables, any saved registers it uses, and room for the saving arguments of procedures it calls

2) Save any “preserved” registers it uses:
   ($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 any registers saved in step 2

7) Deallocate space on stack (add to $sp)

8) Return to CALLER with jr $ra

A leaf function might not need to do anything for steps 1, 2, 3, and 6.
More Legalese

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

If needed the CALLEE will:

1) Allocate a stack frame including space for local variables, any saved registers it uses, and room for the saving arguments of procedures it calls

2) Save any “preserved” registers it uses:
   ($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 any registers saved in step 2.
7) Deallocate space on stack (add to $sp)
8) Return to CALLER with jr $ra

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

```assembly
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
```