Support for High-Level Language constructs are an integral part of modern computer organization. In particular, support for 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)**
  ```java
  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

How do we keep track of all the variables?

This is yet another “x”

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 (at 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 $\text{ra}$ instruction.

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

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.
And It “Sort Of” Works

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

main()
{
    sqr(10);
}

Oh, recursion gives me a headache.

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

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

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
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);
}

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
Procedure Linkage: First Try

**Callee/Caller**

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

**Caller**

```c
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

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

```c
tqr:  slti $t0,$a0,2
    beq $t0,$0,then  #! (x<2)
    add $v0,$0,$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
```
Procedure Linkage: First Try

Callee/Caller

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

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

Caller

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

OOPS!

```
sqr:    slti    $t0,$a0,2
       beq $t0,$0,then  #!(x<2)
       add $v0,$0,$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
```
```

OOPS!

```
rtn:
       jr $ra
```


OOPS!

```
add $t0,$0,$a0
addi $a0,$a0,-1
jal sqr
add $v0,$v0,$t0
add $v0,$v0,$t0
addi $v0,$v0,-1
```

OOPS!

```
rtn:
       jr $ra
```

OOPS!
Procedure Linkage: First Try

Callee/Caller

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

```mips
sqr: slti $t0,$a0,2
    beq $t0,$0,then #! (x<2)
    add $v0,$0,$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
```

OOPS!

```mips
$\text{t0 is clobbered on successive calls.}$
```

Caller

```c
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

Will saving “x” in some register or at some fixed location in memory help?
Procedure Linkage: First Try

Callee/Caller

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

```c
int sqr(int x) {                      sqr:  slti  $t0,$a0,2
    if (x > 1)                          beq  $t0,$0,then  #=> (x<2)
        x = sqr(x-1)+x+x-1;             add  $v0,$0,$a0
    return x;                          beq  $0,$0,rtn
}
```

```c
main() {                            then:
    sqr(10);                         add  $t0,$0,$a0
    Will saving “x” in some           addi  $a0,$a0,-1
    register or at some fixed location in memory help? (Nope)
    return x;                      jal  sqr
}
```

Caller

```c
main() {                            rtn:
    sqr(10);                          jr  $ra
    $t0 is clobbered on successive calls.
    Will saving “x” in some register or at some fixed location in memory 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
Procedure Linkage: First Try

Callee/Caller

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

```mips
sqr:    slti     $t0,$a0,2
        beq      $t0,$0,then  #!(x<2)
        add      $v0,$0,$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
```

Caller

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

```mips
    $t0 is clobbered on successive calls.
    Will saving “x” in some register or at some fixed location in memory 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

OOPS!

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**
  
  \[ f(x,y,z) \text{ or worse... } \sin(a+b) \]

- **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**.

Local variables of Callee:

```c

{ 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.

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.
Lives of Activation Records

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

sqr(3)

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.
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.

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

```
sqr(3)  sqr(3)  sqr(2)
```

TIME
Lives of Activation Records

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

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.
Lives of Activation Records

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

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.
Lives of Activation Records

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

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.
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.
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.
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.
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.
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.
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.
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.
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.
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.
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.
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
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
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
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

Recall that directly addressable global variables were allocated relative to a special “global pointer”

Higher addresses

800000016

“stack” segment

$sp

Recall that directly addressable global variables were allocated relative to a special “global pointer”

Data

100800016

100000016

Lower addresses

004000016

“text” segment (Program)

Reserved

Other possible implementations include:
1) stacks that grow “UP”
2) SP points to first UNUSED location

Hummm... Why is that the TOP of the stack?
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** rx: push Reg[x] onto stack

\[ \begin{align*}
\text{Reg[SP]} & = \text{Reg[SP]} - 4 \\
\text{Mem[Reg[SP]]} & = \text{Reg[x]}
\end{align*} \]

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

\[ \begin{align*}
\text{Reg[x]} & = \text{Mem[Reg[SP]]} \\
\text{Reg[SP]} & = \text{Reg[SP]} + 4;
\end{align*} \]
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** $rx$: push $\text{Reg}[x]$ onto stack

\[ \text{Reg}[SP] = \text{Reg}[SP] - 4 \]
\[ \text{Mem} [\text{Reg}[SP]] = \text{Reg}[x] \]

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

\[ \text{Reg}[x] = \text{Mem}[\text{Reg}[SP]] \]
\[ \text{Reg}[SP] = \text{Reg}[SP] + 4; \]
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**  \( rx \): push \( \text{Reg}[x] \) onto stack

\[ \text{Reg[SP]} = \text{Reg[SP]} - 4 \]
\[ \text{Mem}[\text{Reg[SP]}] = \text{Reg}[x] \]

**POP**  \( rx \): pop the value on the top of the stack into \( \text{Reg}[x] \)

\[ \text{Reg}[x] = \text{Mem}[\text{Reg[SP]}] \]
\[ \text{Reg[SP]} = \text{Reg[SP]} + 4; \]
Stack Management Primitives

**ALLOCATE**  \( k \): reserve \( k \) WORDS of stack

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

\[
\text{addi } $sp, $sp, -4^*k
\]

**DEALLOCATE**  \( k \): release \( k \) WORDS of stack

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

\[
\text{addi } $sp, $sp, 4^*k
\]

**PUSH**  \( rx \): push \( \text{Reg}[x] \) onto stack

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

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

\[
\text{addi } $sp, $sp, -4
\]

\[
\text{sw } $rx, 0($sp)
\]

**POP**  \( rx \): pop the value on the top of the stack into \( \text{Reg}[x] \)

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

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

An ALLOCATE 1 followed by a store
Stack Management Primitives

ALLOCATE $k$: reserve $k$ WORDS of stack
Reg[SP] = Reg[SP] - 4*$k

DEALLOCATE $k$: release $k$ WORDS of stack
Reg[SP] = Reg[SP] + 4*$k

PUSH $rx$: push Reg[$x$] onto stack
Reg[SP] = Reg[SP] - 4
Mem[Reg[SP]] = Reg[$x$]

POP $rx$: pop the value on the top of the stack into Reg[$x$]
Reg[$x$] = Mem[Reg[SP]]
Reg[SP] = Reg[SP] + 4;

An ALLOCATE 1 followed by a store
addi $sp,$sp,-4
sw $rx, 0($sp)

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 save those registers it expects to be unchanged)

CHOICE 3… Something in between.
(Give the Callee some registers to play with. But, make it 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.)
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 save those registers it expects to be unchanged)

**CHOICE 3**... Something in between.
(Give the Callee some registers to play with. But, make it 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 extra arguments (>4, 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.
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.
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, 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.

<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 instant in time. One can mine a lot of information by inspecting its contents.

Can we determine the number of CALLEE arguments?

Can we determine the maximum number of arguments needed by any procedure called by the CALLER?

Where in the CALLEE's stack frame might one find the CALLER's $fp?
Stack Snap Shots

Shown on the right is a snap shot of a program's stack contents, taken at some instant 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?

Where in the CALLEE's stack frame might one find the CALLER's $fp?

$sp (prior to call)
Stack Snap Shots

Shown on the right is a snapshot of a program's stack contents, taken at some instant 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?

<table>
<thead>
<tr>
<th>CALLER’S FRAME</th>
<th>CALLEE’S FRAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>$sp (prior to call)</td>
<td>$sp (after call)</td>
</tr>
<tr>
<td>CALLER’s $fp</td>
<td>CALLEE’s $fp</td>
</tr>
<tr>
<td>$sp (after call)</td>
<td>$sp (prior to call)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CALLER’S FRAME</th>
<th>CALLEE’S FRAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space for $ra</td>
<td>Space for $ra</td>
</tr>
<tr>
<td>Space for $fp</td>
<td>Space for $fp</td>
</tr>
<tr>
<td>Space for $s3</td>
<td>Space for $s3</td>
</tr>
<tr>
<td>Space for $s2</td>
<td>Space for $s2</td>
</tr>
<tr>
<td>Space for $s1</td>
<td>Space for $s1</td>
</tr>
<tr>
<td>$t2</td>
<td>$t2</td>
</tr>
<tr>
<td>$t1</td>
<td>$t1</td>
</tr>
<tr>
<td>Caller's local 1</td>
<td>Callee's local 1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Caller's local n</td>
<td>Callee's local 2</td>
</tr>
</tbody>
</table>
Stack Snap Shots

Shown on the right is a snap shot of a program’s stack contents, taken at some instant 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.

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

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

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

```
sqr:  addiu $sp,$sp,-8
       sw   $ra,4($sp)
       sw   $a0,0($sp)
       slti $t0,$a0,2
       beq  $t0,$0,then
       add  $v0,$0,$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
```
Back to Reality

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);
}
```

```
sqr:  addiu $sp,$sp,-8
sw    $ra,4($sp)
sw    $a0,0($sp)
slti  $t0,$a0,2
beq   $t0,$0,then
add   $v0,$0,$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
```
Back to Reality

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);
}
```

```
sqr:  addiu  $sp,$sp,-8
       sw     $ra,4($sp)
       sw     $a0,0($sp)
       slti   $t0,$a0,2
       beq    $t0,$0,then
       add    $v0,$0,$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.
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);
}
```

```
ALLOCATION of 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.
```

```
ALLOCATE stack frame.

DEALLOCATE stack frame.
```
Back to Reality

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);
}
```

```assembly
sqr:  addiu $sp,$sp,-8
      sw  $ra,4($sp)
      sw  $a0,0($sp)
      slti $t0,$a0,2
      beq $t0,$0,then
      add  $v0,$0,$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
```
Back to Reality

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);
}
```

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

Q: Why didn't we save and update $fp?
A: Don't have local variables or spilled args.

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

DEALLOCATE stack frame.

Save registers that must survive the call.
Now let's take a look at the active stack frames at some point during the procedure's execution.

```assembly
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
```
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
```
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?

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?
Procedure Linkage is Nontrivial

The details can be overwhelming.
What’s the solution for managing this complexity?

Abstraction!

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?
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?
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!**
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
  \((t0-t9, a0-a3, \text{and } v0-v1)\)

• 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 \(v0-v1\)
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 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.

Our contractual obligations:

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

int sqr(int x) {
  if (x > 1)
    x = sqr(x-1)+x+x-1;
  return x;
}
Our running example is a CALLER. Let’s make sure it obeys its contractual obligations

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

```
sqr:   addiu $sp,$sp,-8
        sw $ra,4($sp)
        sw $a0,0($sp)
        slti $t0,$a0,2
        beq $t0,$0,then
        add $v0,$0,$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
```
Our running example is a CALLER. Let's make sure it obeys its contractual obligations.

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

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

```assembly
sqr:  addiu $sp,$sp,-8
sw    $ra,4($sp)
sw    $a0,0($sp)
slti  $t0,$a0,2
beq   $t0,$0,then
add   $v0,$0,$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
```
Our running example is a CALLER. Let’s make sure it obeys its contractual obligations.

Code Lawyer

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

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

\[
\text{addiu } \$sp, \$sp, -8 \\
\text{sw } \$ra, 4(\$sp) \\
\text{sw } \$a0, 0(\$sp) \\
\text{slti } \$t0, \$a0, 2 \\
\text{beq } \$t0, \$0, \text{then} \\
\text{add } \$v0, \$0, \$a0 \\
\text{beq } \$0, \$0, \text{rtn} \\
\text{addi } \$a0, \$a0, -1 \\
\text{jal } \text{sqr} \\
\text{lw } \$a0, 0(\$sp) \\
\text{add } \$v0, \$v0, \$a0 \\
\text{add } \$v0, \$v0, \$a0 \\
\text{addi } \$v0, \$v0, -1 \\
\text{rtn: } \\
\text{lw } \$ra, 4(\$sp) \\
\text{addiu } \$sp, \$sp, 8 \\
\text{jr } \$ra
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,-8
        sw     $ra,4($sp)
        sw     $a0,0($sp)
        slti   $t0,$a0,2
        beq    $t0,$0,then
        add    $v0,$0,$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 1st entry of the CALLEE’s stack

4) EXECUTE procedure

5) Place return values in $v0-$v1

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?

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

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

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

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

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

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

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

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

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

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

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

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

sqr:

```assembly
addiu $sp,$sp,-8
sw $ra,4($sp)
sw $a0,0($sp)
slt $t0,$a0,2
beq $t0,$0,then
add $v0,$0,$a0
beq $0,$0,rtn
```

then:

```assembly
addi $a0,$a0,-1
jal sqr
lw $a0,0($sp)
add $v0,$v0,$a0
add $v0,$v0,$a0
addi $v0,$v0,-1
```

rtn:

```assembly
lw $ra,4($sp)
addiu $sp,$sp,8
jr $ra
```

```c
int sqr(int x) {
    if (x > 1)
        x = sqr(x-1)+x+x-1;
    return x;
}
```
Our running example is also a CALLEE. Are these contractual obligations satisfied?

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

If needed the CALLEE will:

1. Allocate a stack frame including space saved registers, local variables, and spilled arguments
2. Save "preserved" registers used:  
   ($ra, $sp, $s0, $s1, $a0, ..., $v7)  
3. If CALLEE has local variables - or needs access to arguments on stack, save the CALLER's frame pointer, then set $sp to first local variable
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

```assembly
sqr:
  addiu $sp,$sp,-8  
  sw $ra,4($sp)
  sw $a0,0($sp)
  slti $t0,$a0,2
  beq $t0,$0,then
  add $v0,$0,$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
```
More Legalese

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

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

If needed the CALLEE will:

1) Allocate a stack frame including space saved registers, local variables, and spilled arguments
2) Save "preserved" registers used: ($ra, $sp, $fp, $gp, $s0-$s7)
3) IF CALLEE has local variables - or needs access to arguments on stack, save, the CALLER's frame pointer, then set $sp to first local variable
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

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

```c
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"
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...