Procedures (first attempt)

main program
main:
  j helper
back:
stop:

helper procedure
helper:
done: j back

Why?

Code reuse?

main program
main:
  j helper
back:
done: j back

helper procedure
helper:
back:

Fixing it up (not the real way)

main program
main:
  addi $ra, $pc, 8
  j helper

helper procedure
helper:
done: move $pc, $ra
stop:

MIPS registers

Jump-and-link (the actual solution)

main program
main:
  fc: jal helper
  sc: jal helper
  stop:

helper procedure
helper:
  done: jr $ra

MIPS registers
Remember whence we came.

Code reuse!
Calling Conventions and Call Stacks
How do I pass (lots of) arguments to a procedure?

```c
void doStuff(int a, int b, int c, int d, int e, int f, int g, ... ) { ... }
```

How do I store (lots of) local variables?

What if a procedure I call uses the same registers I do?

What if the procedure that called me uses the same registers I do?

When a procedure returns, how does it know where to return?*

Answer: calling conventions and call stacks

---

**Procedure Call Overview**

- **Caller**
  - Create local vars>
  - Set up return value in $v0>
  - Identity local vars>
  - $ra

- **Callee**
  - Set up args>
  - Call callee>
  - Clean up args>
  - Find return value in $v0>
  - Call

**Calling convention** (or procedure call linkage).

- **Callee** must know where to find arguments, return address
- **Caller** must know where to find return value
- **Caller** and **Callee** run on same CPU, use the same registers.
  - How do we deal with register reuse?
**Procedure Call Overview**

Calling convention (or procedure call linkage).

- **Caller** must know where to find arguments, return address
- **Callee** must know where to find return value
- **Caller** and **Callee** must use register saving conventions.

**MIPS Register Conventions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Register Number</th>
<th>Usage</th>
<th>Preserve on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>constant 0 (hardware)</td>
<td>n/a</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>reserved for assembler</td>
<td>n/a</td>
</tr>
<tr>
<td>$v0 - $v1</td>
<td>2-3</td>
<td>returned values</td>
<td>no</td>
</tr>
<tr>
<td>$a0 - $a3</td>
<td>4-7</td>
<td>arguments</td>
<td>yes</td>
</tr>
<tr>
<td>$t0 - $t7</td>
<td>8-15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0 - $s7</td>
<td>16-23</td>
<td>stored values</td>
<td>yes</td>
</tr>
<tr>
<td>$t8 - $t9</td>
<td>24-25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return addr (hardware)</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Stacks (the rest of the world)**

Last-in, first-out (LIFO) data structure.

Operations:
- push - add element to top
- pop - remove element from top
Call Stack (HW/SW interface)

Region of memory managed like a stack. Grows toward lower addresses.

Register $sp$ contains lowest stack address = address of "top" element.

MIPS Call Stack: Push

- `addi $sp, $sp, -4`
  - Decrement $sp$ by 4 (why 4?)
- `sw $register, ($sp)`
  - Store value at address given by $sp$

MIPS Call Stack: Pop

- `lw $register, ($sp)`
  - Load value from address given by $sp$
- `addi $sp, $sp, 4`
  - Increment $sp$ by 4

Memory Layout

- Stack
- Dynamic Data (Heap)
- Static Data
- Literals
- Instructions
- Data

Those bits are still there; we're just not using them.

The program

Local variables; procedure context

Objects allocated with `new` or `malloc`

Static variables (global variables)

Literals (e.g., "example")
Memory Layout

- **Stack**: Managed by compiler (or cs240 programmer...)
- **Dynamic Data (Heap)**: Managed by programmer with help from run-time systems.
- **Static Data**: Initialized when process starts
- **Literals**: Initialized when process starts
- **Instructions**: Read-only; not executable

The stack pointer

In MIPS registers and memory:

- pc
- $ra
- $sp
- $a0
- $a1
- $a2
- $a3
- $v0
- $v1

Procedure call in action (via translation)

```c
int leaf_example (int g, int h, int i, int j) {
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

Space for locals

```c
int leaf_example (int g, int h, int i, int j) {
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

**leaf_example**:

- addi $sp, $sp, -12
- sw $t1, 8($sp)
- sw $t0, 4($sp)
- sw $s0, 0($sp)

Memory layout:

- Stack: Managed by compiler (or cs240 programmer...)
- Dynamic Data (Heap): Managed by programmer with help from run-time systems.
- Static Data: Initialized when process starts
- Literals: Initialized when process starts
- Instructions: Read-only; not executable
Perform the task

```c
int leaf_example (int g, int h, int i, int j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

Store return value

```c
int leaf_example (int g, int h, int i, int j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

Deallocate locals, return control

```assembly
leaf_example:
    addi $sp, $sp, -12
    sw $t1, 8($sp)
    sw $t0, 4($sp)
    sw $s0, 0($sp)
    add $t0, $a0, $a1
    add $t1, $a2, $a3
    sub $s0, $t0, $t1
```

Recursive/nested procedures

```assembly
main:
    jal helper
helper:
    jal helper
    jr $ra
```

*And how does the helper procedure avoid clobbering main’s variables (much less its own)?*
Nested procedures

```c
int fact (int n)
{
    if (n < 1) return 1;
    else return n * fact(n-1);
}
```

Push return address and argument reg

```c
int fact (int n)
{
    if (n < 1) return 1;
    else return n * fact(n-1);
}
```

Test n < 1

```c
int fact (int n)
{
    if (n < 1) return 1;
    else return n * fact(n-1);
}
```

If n < 1 then return 1

```c
int fact (int n)
{
    if (n < 1) return 1;
    else return n * fact(n-1);
}
```
If \( n \geq 1 \), \(-n\) and call fact

```c
int fact (int n)
{
    if (n < 1) return 1;
    else return n * fact(n-1);
}
```

When we return from recursive call

```c
addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)
slt $t0, $a0, 1
beq $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr $ra
L1: addi $a0, $a0, -1
jal fact
lw $a0, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, 8
mul $v0, $a0, $v0
jr $ra
```

Finally, multiply and return

```c
addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)
slt $t0, $a0, 1
beq $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr $ra
L1: addi $a0, $a0, -1
jal fact
lw $a0, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, 8
mul $v0, $a0, $v0
jr $ra
```