Let’s try to make the code reusable...

**C code**

```c
int NumSpaces (char *s)
{
    int count = 0;
    while (*s) {
        if (*s++ == ' ') count++;
    }
    /* count contains the number of spaces */
    return count;
}
```
Procedures

Questions:

How does one...

• pass parameters?
• pass back the return value?
• start executing the function?
• return from the function?
• use registers?
First Attempt

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>pass parameters?</td>
<td>use register $4</td>
</tr>
<tr>
<td>pass back the return value?</td>
<td>use register $2</td>
</tr>
<tr>
<td>start executing the function?</td>
<td>use j</td>
</tr>
<tr>
<td>return from the function?</td>
<td>use j</td>
</tr>
<tr>
<td>use registers?</td>
<td>use any</td>
</tr>
</tbody>
</table>

Assembly

**callee**

```
NumSpaces:  addu $17,$0,$4
    ...
$done:      addu $2,$0,$16
    j Return
```

**caller**

```
...  
j NumSpaces
Return: ...
...  
```
**Second Attempt**

Might want to call function from multiple places...

<table>
<thead>
<tr>
<th>start executing the function?</th>
<th>use jal</th>
</tr>
</thead>
<tbody>
<tr>
<td>return from the function?</td>
<td>use jr</td>
</tr>
</tbody>
</table>

**Assembly**

```assembly
callee
NumSpaces: addu $17,$0,$4
... $done: addu $2,$0,$16
jr $31

caller
... jal NumSpaces
... $31
...```

What About Recursion?

C code

```c
int NumSpaces (char *s)
{
    int count;
    if (!(*s)) return 0;
    count = NumSpaces (s+1);
    if (*s == ' ') count++;
    return count;
}
```

see also: Recursion
What About Recursion?

NumSpaces:  addu $17,$0,$4  # s = argument
lbu $8, 0($17)  # temp = *s
beq $8,$0,$done  # if *s == 0 goto done
addiu $4,$4,1  # argument = s+1
jal NumSpaces  # call NumSpaces

li $9,32  # temp = ''

bne $9,$8,$skipinc  # if *s != ''

addiu $2,$2,1  # count++

$skipinc:  jr $31  # return

$done: li $2,0  # return value = 0
jr $31  # return
Stacks: Last-In First-Out

- **Push**: save a value/add entry
- **Pop**: restore a value/remove entry
Stacks

- Use stack to save return address, registers
- Stack pointer: register 29 (what’s push/pop?)
- Stack frames
  - Groups of elements pushed/popped for a single call

Once again...

<table>
<thead>
<tr>
<th>start executing the function?</th>
<th>use jal, but save return address on stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>return from the function?</td>
<td>use jr, but pop return address first</td>
</tr>
</tbody>
</table>
**Third Attempt**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>addiu $29,$29,-4</td>
<td>allocate stack space</td>
</tr>
<tr>
<td>sw $31,0($29)</td>
<td>save return addr</td>
</tr>
<tr>
<td>addu $17,$0,$4</td>
<td>s = argument0</td>
</tr>
<tr>
<td>lub $8, 0($17)</td>
<td>temp = *s</td>
</tr>
<tr>
<td>beq $8,$0,$done</td>
<td>if (temp == 0) goto done</td>
</tr>
<tr>
<td>addiu $4,$4,1</td>
<td>argument0 = s+1</td>
</tr>
<tr>
<td>jal NumSpaces</td>
<td>recursive call</td>
</tr>
<tr>
<td>li $9,32</td>
<td>temp2 = ''</td>
</tr>
<tr>
<td>bne $9,$8,$skipinc</td>
<td>if (temp != '') goto skipinc</td>
</tr>
<tr>
<td>addiu $2,$2,1</td>
<td>count++</td>
</tr>
</tbody>
</table>

**$skipinc**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $31,0($29)</td>
<td>pop return address</td>
</tr>
<tr>
<td>addiu $29,$29,4</td>
<td>pop stack</td>
</tr>
<tr>
<td>jr $31</td>
<td>return</td>
</tr>
</tbody>
</table>

**$done:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>li $2,0</td>
<td>return val = 0</td>
</tr>
<tr>
<td>lw $31,0($29)</td>
<td>pop return address</td>
</tr>
<tr>
<td>addiu $29,$29,4</td>
<td>pop stack</td>
</tr>
<tr>
<td>jr $31</td>
<td>return</td>
</tr>
</tbody>
</table>
Register usage convention:

- Who saves registers?
  - Caller vs callee

- Where are the registers saved?
  - Must be in memory
  - Stack!

- Which registers should be saved?
  - In general, all those ones that are modified...

(FORTRAN 77 does not support recursion, saves variables in globals)
Fourth Attempt

Example: a function that modifies $8, $9, $18

Function:  
```
addiu $29, $29, -16  # create space on stack
sw $31, 12($29)  # save ret addr
sw $8, 8($29)  # save $8
sw $9, 4($29)  # save $9
sw $18, 0($29)  # save $18
...
$ret: lw $18, 0($29)  # restore $18
lw $9, 4($29)  # restore $9
lw $8, 8($29)  # restore $8
lw $31, 12($29)  # restore ret addr
addiu $29, $29, 16  # pop stack
jr $31  # return
```
Prolog and Epilog

Functions are assembled in a standard form:

- **Prolog**
  - Template code at the beginning
  - Allocates space on the stack, saves registers

- **Epilog**
  - Template code at the end
  - Deallocates space on the stack, restores registers

**Problem:** too much call/return overhead.
### MIPS Calling Convention

- **First 4 integer arguments:** $4$–$7$ ($a0$–$a3$)
- **Return address:** $31$ ($ra$)
- **Stack pointer:** $29$ ($sp$)
- **Frame pointer:** $30$ ($fp$)
- **Return value:** $2$, $3$ ($v0$, $v1$)
- **Callee saved:** $16$–$23$ ($s0$–$s7$)
- **Caller saved:** $8$–$15$, $24$, $25$ ($t0$–$t7$, $t8$, $t9$)
- **Reserved:** $26$, $27$ ($k0$, $k1$)
- **Global pointer:** $28$ ($gp$)
- **Assembler temporary:** $1$ ($at$)
Argument Passing

What if there are >4 arguments?

Use the stack.

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

```
0x00000004

sp

larger addresses

li $4,0
li $5,1
li $6,2
li $7,3
li $8,4
sw $8,-4($29)
jal f
addiu $29,$29,-4

smaller addresses

# arg0 = 0
# arg1 = 1
# arg2 = 2
# arg3 = 3
# temp = 4
# arg4 = 4
# on stack
```
Argument Passing

How do we handle variable-length parameters?

Example:

```c
printf ("Avg:%f, Mean:%f, Med:%f\n",x,y,z);
```

- **Special-purpose code?**
  ```c
  if (num == 1) use $4; else if (num == 2) use $5;
  ...
  ```

- **Put all arguments on stack?**

- **MIPS: leave space on the stack for 4 args**
  - caller may not know function is varargs
  - callee can copy args to stack if necessary
Argument Passing

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

```plaintext
0x00000004
sp
is varargs.
jal f
addiu $29,$29,−20
sp+16
sp+12
sp+8
sp+4
slot for arg0
slot for arg1
slot for arg2
slot for arg3

li $4,0      # arg0 = 0
li $5,1      # arg1 = 1
li $6,2      # arg2 = 2
li $7,3      # arg3 = 3
li $8,4      # temp = 4
sw $8,−4($29) # arg4 = 4
addiu $29,$29,−20 # on stack
jal f

... what about bytes/half-words/double-words?
```

Four words of "space" on the stack, in case callee is varargs.
Body of function $f$:

```
sp+24     0x00000004
slot for arg3
slot for arg2
slot for arg1
slot for arg0

saved $31
saved $28
```

$f$:  
```
addiu $29, $29, -8
sw $31, 4($29)
sw $28, 0($29)
...
```
What Else Goes On The Stack?

- Saved registers
- Arguments
- Standard saved regs
- Local variables
- Saved registers
- Arguments
- Standard saved regs
- Local variables

- Stack used by f1. f1 calls f2.
  - Local variables that don’t fit in regs.
  - Saved registers
- Stack used by f3.
Stack Frames

Register $30$ is the frame pointer.

- Value of stack pointer at function entry
- Used to restore stack pointer before returning

Part of stack owned by a function is the frame.

Frame pointers not really required. Needs to be saved/restored if used.