Procedures

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

C code

```
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

pass parameters?	use register $4
pass back the return value?	use register $2
start executing the function?	use j
return from the function?	use j
use registers?	use any

Assembly

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


Second Attempt

Might want to call function from multiple places...

start executing the function?	use jal
return from the function?	use jr

Assembly

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


What About Recursion?

C code

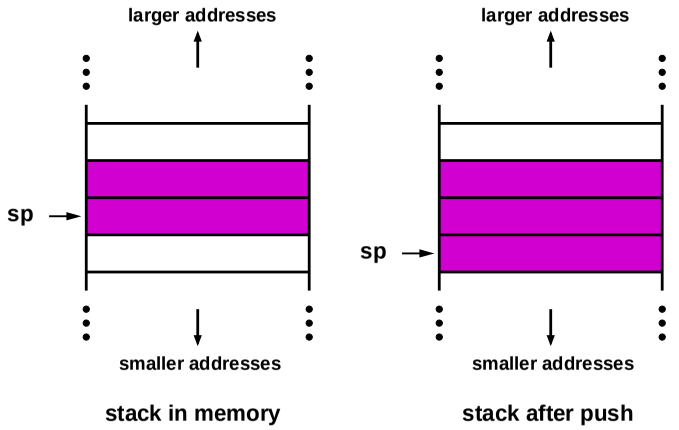
```
int NumSpaces (char *s)
{
   int count;
   if (!(*s)) return 0;
   count = NumSpaces (s+1);
   if (*s == ' ') count++;
   return count;
}
```

see also: Recursion

What About Recursion?

```
NumSpaces:  addu $17,$0,$4      # s = argument1
            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
                                # count is $2
            li $9,32            # temp = ' '
            bne $9,$8,$skipinc  # if *s != ' '
                                #  goto skipinc
            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...

start executing the function?	use `jal`, but save return address on stack
return from the function?	use `jr`, but pop return address first

Third Attempt

```
NumSpaces:  addiu $29,$29,-4     # allocate stack space
            sw $31,0($29)        # save return addr
            addu $17,$0,$4       # s = argument0
            lbu $8, 0($17)       # temp = *s
            beq $8,$0,$done      # if (temp == 0) goto done
            addiu $4,$4,1        # argument0 = s+1
            jal NumSpaces        # recursive call
            li $9,32             # temp2 = ' '
            bne $9,$8,$skipinc   # if (temp != ' ') goto skipinc
            addiu $2,$2,1        # count++
  $skipinc  lw $31,0($29)        # pop return address
            addiu $29,$29,4      # pop stack
            jr $31               # return
    $done:  li $2,0              # return val = 0
            lw $31,0($29)        # pop return address
            addiu $29,$29,4      # pop stack
            jr $31               # return
```


Fourth Attempt

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

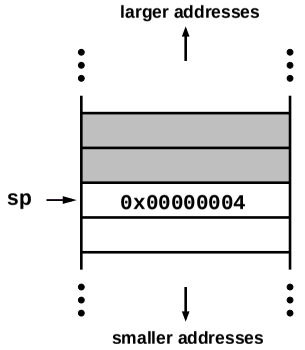
- First 4 integer arguments: $4–$7 ($a0–$a3)
- Return address: $31 ($ra)
- Stack pointer: $29 ($sp)
- Frame pointer: $30 ($fp)
- Return value: $2, $3 ($v0,$v1)
- Callee saved: $16–$23 ($s0–$s7)
- Caller saved: $8–$15,$24,$25 ($t0–$t7,$t8,$t9)
- Reserved: $26,$27 ($k0,$k1)
- Global pointer: $28 ($gp)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

Argument Passing

What if there are >4 arguments?

Use the stack.

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




```
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,-4     # on stack
jal f
```

Argument Passing

How do we handle variable-length parameters?

Example:

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

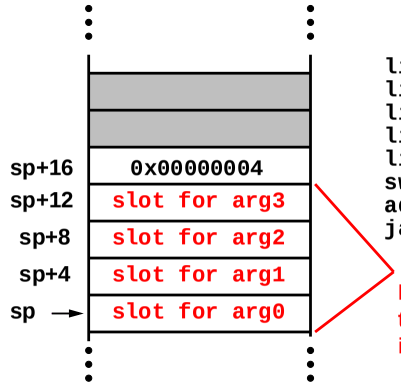
- Special-purpose code?

```
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

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

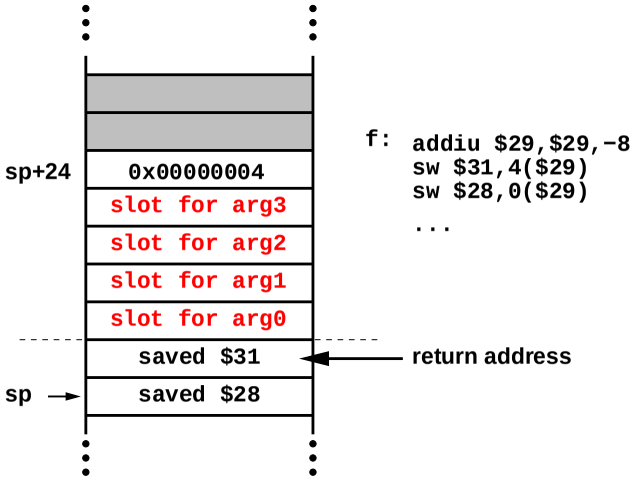



```
li $4,0              # arg0 = 0
li $5,1              # arg1 = 1
li $6,2              # arg2 = 2
li $7,3              # arg3 = 3
li $8,4              # temp = 4
sw $8,-4($29)        # arg4 = 4
addiu $29,$29,-20    # on stack
jal f
```

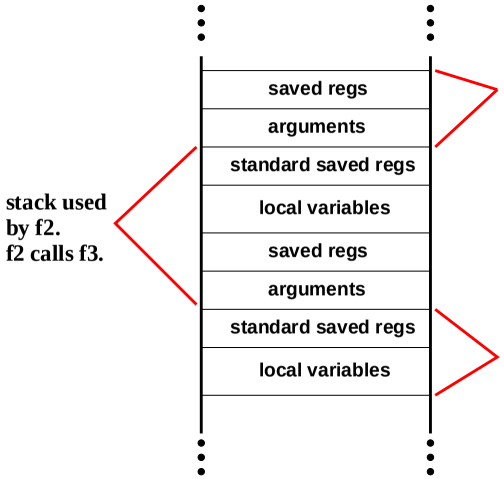
... what about bytes/half-words/double-words?

Argument Passing

Body of function `f`:



What Else Goes On The Stack?



- Local variables that don't fit in regs.
- Saved registers

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.