C functions

```c
main() {
  int i,j,k,m;
  ...
  i = mult(j,k); ...
  m = mult(i,i); ...
}
/* really dumb mult function */
int mult(int mcand, int mlier){
  int product;
  product = 0;
  while (mlier > 0)  {
    product = product + mcand;
    mlier = mlier -1; }
  return product;
}
```

Function Call Bookkeeping

- Registers play a major role in keeping track of information for function calls.
- **Register conventions:**
  - Return address \( \$ra \)
  - Arguments \( \$a0, \$a1, \$a2, \$a3 \)
  - Return value \( \$v0, \$v1 \)
  - Local variables \( \$s0, \$s1, \ldots, \$s7 \)
  - The stack is also used; more later.

Instruction Support for Functions (1/6)

```c
int sum(int x, int y) {
  return x+y;
}
```

In MIPS, all instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.

Instruction Support for Functions (2/6)

```
address
M 1000
P 1000
S 1008
   1012
      1016
  2000
     2004
```

Instruction Support for Functions (3/6)

```
address
M 1000
P 1004
S 1008
   1012
      1016
  2000
     2004
```

- Question: Why use \( \text{j}r \) here? Why not simply use \( \text{j} \)?
- Answer: sum might be called by many functions, so we can’t return to a fixed place. The calling proc to sum must be able to say “return here” somehow.

```
address
M 1000
P 1004
S 1008
   1012
      1016
  2000
     2004
```

Instruction Support for Functions (4/6)

- Single instruction to jump and save return address: jump and link (\( \text{j}al \))
- **Before:**
  ```
  1008 addi $ra,$zero,1016 #$ra=1016
  1012 j sum #jump to sum
  ```
- **After:**
  ```
  1008 jal sum # $ra=1012,goto sum
  ```
- Why have a \( \text{j}al \)? Make the common case fast: function calls are very common. Also, you don’t have to know where the code is loaded into memory with \( \text{j}al \).
Instruction Support for Functions (5/6)

• Syntax for jal (jump and link) is same as for j (jump):
  
  jal label

• jal should really be called laj for "link and jump":
  
  • Step 1 (link): Save address of next instruction into $ra (Why next instruction? Why not current one?)
  • Step 2 (jump): Jump to the given label

Instruction Support for Functions (6/6)

• Syntax for jr (jump register):
  
  jr register

• Instead of providing a label to jump to, the jr instruction provides a register which contains an address to jump to.

• Only useful if we know exact address to jump to.

• Very useful for function calls:
  
  • jal stores return address in register ($ra)
  • jr $ra jumps back to that address

# load parameters for test call to foo(4,6)
li   $a0,4       # set up first parameter
li   $a1,6       # set up second parameter

# call foo
jal   foo        # call function

# on return from foo, the result is in $v0, save it and print it
move  $v0,$v0    # move result into argument
li    $v0,1      # syscall code for print integer
syscall

# end program
li    $v0,10     # terminate execution
syscall

### MY FIRST SPIM PROCEDURE

## Simple procedure example: not more than 4 arguments, only 1 return value, no calls from within the procedure, and no local variables!

#### Procedure foo

foo:

# get arguments
move  $t0,$a0       # get first argument
move  $t1,$a1       # get second argument

# perform body of the procedure
mul   $t2,$t0,2     #compute 2*first argument
mul   $t2,$t2,$t1    #compute 2*first argument*second argument

# set up return value in register $v0
move  $v0,$t2

# return
jr    $ra           #return