Lecture 5: Procedures
Function call book-keeping in C

```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;
}
```

What information must compiler/programmer keep track of?

What instructions can accomplish this?
Function Call Book-keeping in Assembly

• 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, ... , $s7

• The stack is also used; more later.
In MIPS,
- instructions are 4 bytes
- stored in memory

Here are addresses where the instructions are stored.
... sum(a,b);... /* a,b:$s0,$s1 */

C

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

M

```assembly
address
1000 add $a0,$s0,$zero       # x = a
1004 add $a1,$s1,$zero       # y = b
1008 addi $ra,$zero,1016     #$ra=1016
1012 j          sum          #jump to sum
1016 ...
```

P

```assembly
2000 sum: add $v0,$a0,$a1
2004 jr $ra     # new instruction
```
C

... sum(a,b);... /* a,b:$s0,$s1 */
}

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

M

• Question: Why use \texttt{jr} instead of using \texttt{j}?

• Answer: \texttt{sum} might be called by many functions, so we can’t return to a fixed location/address. The calling proc to \texttt{sum} must be able to specify “return address”.

2000

\texttt{sum: add $v0,$a0,$a1}

2001

\texttt{jr $ra} # new instruction
Instruction Support for Functions (4/6)

- Single instruction to jump and save return address: jump and link (jal)

- Earlier approach:
  
  1008  addi $ra,$zero,1016  #$ra=1016
  1012  j sum  # goto sum

- Faster approach:
  
  1008  jal sum  # $ra=1012, goto sum

- Why have a new instruction (jal)?
  1) Make the common case fast, function calls are very common.
  2) Also, you don’t need to know the memory address of individual instructions with jal.
• 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 \texttt{jr} (jump register):
  \[
  \texttt{jr \ register}
  \]

• Instead of providing a label to jump to, the \texttt{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:
  • \texttt{jal} stores return address in register (\$ra)
  • \texttt{jr \ $ra} jumps back to that address
Rules for Procedures

• Called with a jal instruction, returns with a jr $ra

• Accepts up to 4 arguments in $a0, $a1, $a2 and $a3

• Return value is always in $v0 (and if necessary in $v1)

• Must follow register conventions (even in functions that only you will call)!
  - More details in next lecture
## 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!

In C it would be:

```c
int foo (int ain, int bin) {
    int n = 2*ain*bin;
    return n;
}
```
# 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 $a0,$v0  # move result into argument
    li $v0,1      # syscall code for print integer
    syscall

# end program
    li $v0,10     # terminate execution
    syscall
## 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 arg*second arg

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

# return
jr $ra  # return
“And in Conclusion…”

- Functions called with jal, return with jr $ra.

- Instructions we know so far
  - Arithmetic: add, addi, sub, addu, addiu, subu
  - Memory:  lw, sw, lh, sh, lb, sb, li, la
  - Decision:  beq, bne, (blt, ble, bgt, bge)
  - slt, slti, sltu, sltiu
  - Unconditional Branches (Jumps):  j, jal, jr

- Registers we know so far
  - All of them (almost)!