CSE-413 Computer Architecture
Lecture 7

Nested Procedure

Nested Procedure

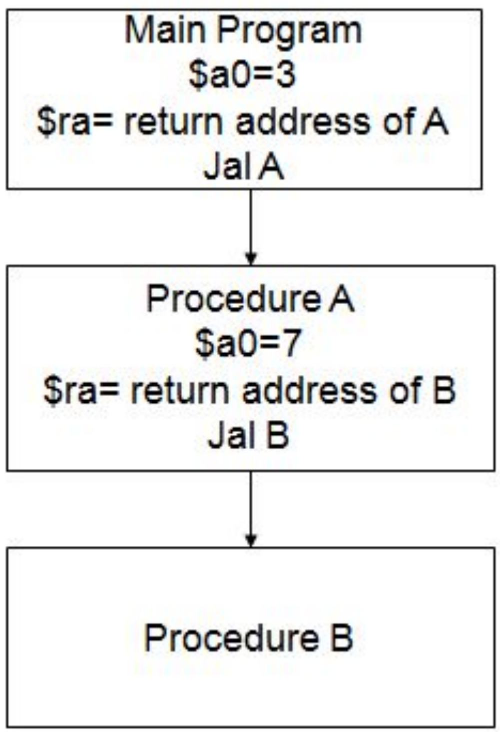
- Suppose that the main program calls procedure A with an argument of 3, by placing the value 3 into register $a0 and then using jal A.

- Then suppose that procedure A calls procedure B via jal B with an argument of 7, also placed in $a0.

- Since A hasn’t finished its task yet, there is a conflict over the use of register $a0.

- Similarly, there is a conflict over the return address in register $ra, since it now has the return address for B.

Cont.



Solution:

- The caller pushes any argument registers ($a0-$a3) or temporary registers ($t0-$t9) that are needed after the call.
- The callee pushes the return address register $ra and any saved registers ($s0-$s7) used by the callee.
- The stack pointer $sp is adjusted to account for the number of registers placed on the stack.

Example

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

The parameter variable n corresponds to the argument register $a0. The compiled program starts with the label of the procedure and then saves two registers on the stack, the return address and $a0:

```
fact:
addi $sp, $sp, –8   # adjust stack for 2 items
sw $ra, 4($sp)      # save the return address
sw $a0, 0($sp)      # save the argument n
```

Cont.

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

The next two instructions test whether n is less than 1, going to L1 if n ≥ 1.

slti \$t0,\$a0,1 # test for n < 1 (slti=set on less than)
beq \$t0,\$zero, L1 # if n >= 1, go to L1 (branch if registers are equal instruction (beq))

BEQ (short for "Branch if EQual") is the mnemonic for a machine language **instruction** which branches, or "jumps", to the address specified if, and only if the zero flag is **set**.

Cont.

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

If n is less than 1, fact returns 1 by putting 1 into a value register: it adds 1 to 0 and places that sum in $v0. It then pops the two saved values off the stack and jumps to the return address:

```
lw $a0, 0($sp)
lw $ra, 4($sp)          # restore the return address
addi $v0,$zero,1        # return 1
addi $sp,$sp,8          # pop 2 items off stack
jr $ra                  # return to caller
```

Cont.

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

If n is not less than 1, the argument n is decremented and then fact is called again with the decremented value:

```
L1: addi $a0,$a0,–1    # n >= 1: argument gets (n – 1)
jal fact               # call fact with (n – 1)
```

Cont.

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

The next instruction is where fact returns. Now the old return address and old argument are restored, along with the stack pointer:

```
lw $a0, 0($sp) # return from jal: restore argument n
lw $ra, 4($sp)      # restore the return address
addi $sp, $sp, 8    # adjust stack pointer to pop 2 items
```

Cont.

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

Next, the value register \$v0 gets the product of old argument \$a0 and the current value of the value register.

```
mul $v0,$a0,$v0 # return n * fact (n – 1)
```

Finally, fact jumps again to the return address:

```
jr $ra # return to the caller
```