C Program:

```c
void main(void) {
    hello();
    printf("Finished\n");
}

void hello(void) {
    printf("Hello world!\n");
}
```

When a procedure call is made:

1. System saves the `return address`.
2. Transfers control to the beginning of the called procedure.
3. Completes the procedure.
4. Transfers control back to the caller using the saved `return address`.
#Function main.
.data
mstr: .asciiz "Finished\n"
.text
.globl main
main: jal hello #Procedure call.
la $a0, mstr #Print string.
li $v0, 4
syscall
li $v0, 10 #exit.
sySCALL
#Function hello.
.data
hstr: .asciiz "Hello World!\n"
.text
hello: la $a0, hstr
li $v0, 4
syscall
jr $31 #Return.
Procedure Calls in MAL (continued)

**Actions Performed by jal:**

1. Saves the address of the following instruction in $31.
2. Causes a jump to the specified label.

**Effect of jr $31:** Causes a jump to the address stored in $31.

**Difficulty:** The above mechanism cannot handle nested procedure calls.
void main(void) {  
    hello();  
    printf("Finished\n");  
}   

void hello(void) {  
    printf("Hello world!\n");  
    pzero();  
}  

void pzero(void) {  
    printf("%d\n", 0);  
}
#Function main.

.data
mstr: .asciiz "Finished\n"

.text
.globl main
main: jal hello  #Proc. call
la $a0, mstr  #Print str.
li $v0, 4
syscall
li $v0, 10  #exit.
syscall
#Function hello.

.data
hstr: .asciiz "Hello World!\n"

.text
hello:  la $a0, hstr
li $v0, 4
syscall
jal pzero  #New call.
jr $31  #Won’t work.
#Function pzero.

.data

nl:   .asciiz "\n"

.text

pzero: li $v0, 1
      move $a0, $0
      syscall
      la $a0, nl
      li $v0, 4  #Print "\n".
      syscall
      jr $31
MAL Version of the Program (continued)

Difficulty: The jal instruction in hello overwrites the value stored in $31.

Solution:

1. A procedure X should save register $31 before calling another procedure Y.

2. Before returning to its caller, X must restore register $31.
Handling Nested Calls

Implementation of the Solution:

1. Procedure X pushes $31$ on to the stack.

2. Procedure X calls procedure Y.

3. Procedure Y completes and returns control to X (using the jr $31$ instruction).

4. Procedure X pops the stack to restore $31$.

5. Procedure X returns to its caller (using the jr $31$ instruction).
The System Stack

**Note:** The system stack is part of the memory.

Stack Pointer:

- \$sp: Synonym for \$29.
- Contains the address of the next *free* location on the stack.
The System Stack (continued)

**Saving $31 on Stack:** (push)

```
sw $31, 0($sp)
addi $sp, $sp, -4
```

**Restoring $31 from Stack:** (pop)

```
lw $31, 4($sp)
addi $sp, $sp, 4
```

**MAL Program with Nested Calls:** (Example 4)

Method I: Choose suitable registers for parameters and return value(s).

Example: See Handout 14.2 (Example 5)

Notes Regarding Example 5:

- main: Uses $5$ for $r$ and $6$ for $y$.
- sum: Saves and restores $6$ and $8$ since they are used as temporaries within the procedure.

Method II: Use the system stack for passing parameters.
Example:

Calling Procedure:

#Push parameters on stack.

sw $5, 0(sp)
sw $6, -4(sp)
sw $7, -8(sp)
sw $8, -12(sp)
addi $sp, $sp, -16

Called Procedure:

#Obtain parameters by popping stack.

lw $20, 4(sp)
lw $21, 8(sp)
lw $22, 12(sp)
lw $23, 16(sp)
addi $sp, $sp, 16