Procedure Calls in MAL
```cpp
{ cout << 0 << endl;
  void hello()
  {
    cout << "Hello World\n";
  }
  hello();
  cout << "Finished\n";
  hello();
  void main() {
    cout << "main\n";
  }

Example:
  nested procedure calls.
  Difficulty: The above mechanism cannot handle
  dress stored in $31.$
  What if $331$ does: Causes a jump to the address
  in $r10b$ which is $331.$
  What if $31$ does: Causes a jump to the specified label.
  I saves the address of the following instruction
  label $331.$
```

Diagram:
```
\[ \text{hello} \rightarrow \text{main} \]
```

Conclusion:
Difficult: The jal instruction in hello causes trouble - it overwrites what was stored in §31.

```
.text
    .data
        hello: 40, 'w', 'o', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '
        finished: 40, 'f', 'i', 'n', 'i', 's', 'h', 'e', 'd
    .text
        start: jal finished
    .text
        main
        jal pzero
        jal hello
        syscall
    .data
        pzero
        hello
    .text
        main
        syscall
        move $40, 0
        jal print-strt
        syscall
        move $40, 1
        jal print-strt
        syscall
        move $40, 2
        jal print-strt
        syscall
        move $40, 3
        jal print-strt
        syscall
        move $40, 4
        jal print-strt
        syscall
        move $40, 5
        jal print-strt
        syscall
        move $40, 6
        jal print-strt
        syscall
        move $40, 7
        jal print-strt
        syscall
        move $40, 8
        jal print-strt
        syscall
        move $40, 9
        jal print-strt
        syscall
    .text
        function pzero.
       jal pzero
       new call
       syscall
        # print-string done
        syscall
        $40, mstr
        $40, 4
        print-string
        syscall
        $40, 3
        print-string
        syscall
        $40, 2
        print-string
        syscall
        $40, 1
        print-string
        syscall
        $40, 0
        print-string
        syscall
```

I. Procedure X pushes $31 on the stack.

II. Procedure Y calls procedure X.

III. Procedure Y completes and returns control to X (using JF $31 instruction).

IV. Procedure X pops stack to restore $31.

V. Procedure X returns to its caller.

The usual implementation:

Register $31.

2. Before returning to its caller, X must restore calling another procedure Y.

Solution:

Stack pointer:

• $sp : 5ynym for $29.

• Contains the address of the last used 4 byte

• Word on the stack:

Removing $31 from stack: (pop)

sw $ra, 0($sp)

add $sp, $sp, 4

lw $31, 0($sp) (pop)

Restoring $31 from stack: (pop)

add $sp, $sp, 4

Saving $31 on stack: (push)

The system stack: (a region of memory)
Method 1: Use the system stack for passing parameters.

Parameters are used as temporary within the procedure.

- sum: Saves and restores $6$ and $8$ as they pass.
- main: Uses $5$ for $x$ and $6$ for $y$.

**Notes Regarding the Program:**

MAL program with parameters: Handout.

Method 1: Choose suitable registers for parameters.

Tests and return values(s).

# Obtain parameters by popping stack.

```assembly
addt $sp, $gp, 16
lw $t23, $2, $sp
lw $t22, $8, $sp
lw $t21, $4, $sp
lw $t20, $0, $sp
```

# Push parameters on stack.

```assembly

called procedure:
```

```assembly

calling procedure:
```

**Example**

```assembly
pushq $r10
```

MAL program with nested calls: Handout.