Subroutines

• Also called procedures or functions

*Example C code:*

```c
int main()
{
    int i, j;
    i = 5;
    j = celtokel(i);
    i = j;
    return 0;
}

// subroutine converts Celsius to kelvin
int celtokel(int i)
{
    return (i + 273);
}
```

Arguments/Parameters

Result of the subroutine
Subroutine

- Important for modularizing programs
- separation of concerns, smaller coding units
- Each subroutine should be able to perform independently without worrying about what registers are in use by its callee subroutines.
- reuse of code – call from anywhere
Subroutines

- Basic Steps:
  1. Place arguments in a place where the subroutine can access them
  2. Store the current PC+4 in a place so that the program can return to it at the end
  3. Jump from current PC to the PC at the start of subroutine
  4. Perform the instructions in the subroutine and store the result in a place so that the main program can access it.
  5. Return to the main program at the point saved in step 2.
Subroutines

• Special register for supporting subroutines
  • $31: return address register - stores the return address

• Subroutine Call - Instruction for jumping to the subroutine and saving the current PC:
  
  ```
  jal L1 ; jump-and-link
  ```

  • copy (current PC + 4) to $31
  • Jump to L1

• Subroutine Return - Instruction for jumping back to the saved point
  ```
  jr $31
  ```

  • Jump to address in $31
Subroutines

- Problem! – If after jal, $31 stores the return address, what have we lost? – the return address of the emulator (to end our program) or another subroutine call’s return address

- This means if a piece of code wants to call a subroutine before doing so it must store its own return address or else it will lose it! Where should we store it?

- Register – No! ; Because, a subroutine might use that register for its work, and we want to do nested subroutines and recursion.

- We use a Stack – basically, an area in the memory
Nested Subroutines

- Notice function calls and returns occur in a stack-like order: the most recently called function is the first one to return.

  1. Someone calls A
  2. A calls B
  3. B calls C
  4. C returns to B
  5. B returns to A
  6. A returns

- Here, for example, C must return to B before B can return to A.
Stack

LIFO (last in, first out) principle
Stack

• For each function call, a block of stack space, called a stack frame, is allocated from the memory. This stack frame will be used by the function
• Before the function returns, it must pop the stack fully
• We don't need to remember specific memory addresses for each value, just store address of the last thing we pushed (top of the stack) and remember what order we pushed things in!
• Normally, the current top of stack address is stored by $29 (Stack Pointer register (SP)), but in our MIPS assembler, stack pointer is $30.
• MIPS does not provide push and pop instructions. These have to be done by the programmer
- Convention – stack grows from higher memory address to lower

<table>
<thead>
<tr>
<th>Empty stack</th>
<th>Push 24</th>
<th>Push 86</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Higher address</td>
<td>sp →</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Push values onto stack by subtracting from sp
- Pop values from stack by adding to sp
Stack Save/Restore

- Push (save register to memory)
  - decrement stack pointer to make room
  - copy value to stack pointer memory location
    
    ```
    addi $30, $30, -4
    sw $x, 0($30)
    ```

- Pop (restore register from stack)
  - copy value from stack pointer memory location
  - increment stack pointer to free up space
    
    ```
    lw $x, 0($30)
    addi $30, $30, 4
    ```
Arguments Passing and Return values

- need to pass arguments and return the result(s)
- can use registers and/or stack
  - There needs to be agreement between caller and callee on the protocol
  - MIPS convention
    - first 4 arguments in registers
    - remainder on the stack
MIPS Register Conventions!

- which registers to save?
- which registers to use for arguments/results?
- MIPS
  - $at (1) – assembler temporary
  - $v0 - $v1 (2 - 3) – return values
  - $a0 - $a3 (4 - 7) – arguments
  - $t0 - $t9 (8 - 15, 24, 25) – temporary
  - $s0 - $s7 (16 - 23) – saved temporary
  - $k0 - $k1 (26 - 27) – OS kernel
  - $sp - $30 – Stack Pointer
  - $ra - $31 – Return Address
MIPS Register Conventions

- $t0 - $t9 (8 - 15, 24, 25) – *temporary*: the subroutine you called does NOT promise to preserve the values in these registers.

- $s0 - $s7 (16 - 23) – *saved temporary*: the subroutine you called promises that when it gives execution back to the caller these registers will be in the same state they are when the call was made. Achieved by either not touching the registers or by saving them on the stack before using them, then popping off before returning.

- Simply put, unsaved temporaries if required after subroutine call must be saved by caller, saved temporaries are saved by callee.
int main()
{
    int i, j;
    i = 5;
    j = celtokel(i);
    i = j;
    return 0;
}

// Celsius to kelvin
int celtokel(int i)
{
    return (i + 273);
}

addi $17, $0, 5 ;i=5
add $4, $17, $0 ;Argument
addi $30, $30, -4 ;sp-4
sw $31, 0($30) ;push ra
jal ctkok ;jump to c2k
add $18, $2, $0 ;return val
addi $30, $30, 4 ;sp+4
lw $31, 0($30) ;pop ra
jr $31 ;jump to $ra

c2k:addi $30, $30, -4 ;sp-4
sw $17, 0($30) ;push s1
val
addi $17, $4, 273 ;$4+273
add $2, $17, $0 ;ret val
lw $17, 0($30) ;pop s1
val
addi $17, $17, 4 ;sp+4
jr $31 ;jump to $ra